개발자 안내서

Borland®
C++Builder™ 6
Windows 용

볼랜드 코리아 주식회사
서울특별시 강남구 삼성동 159-1 ASEM 타워 30 층
연락처 : (02)6001-3162

C++Builder 라이센스 조항과 하자 보증에 따라 배포할 수 있는 전체 파일 리스트를 보시려면 C++Builder™ 제품의 루트 디렉토리에 있는 DEPLOY.TXT 파일을 참조하십시오.

Borland는 이 문서에 포함된 내용에 대해서 특허권을 가지고 있거나 특허 출원 중에 있습니다. 이 문서를 공급한다고 해서 미국 및 기타 국가에서 특허권에 대한 라이센서가 부여되는 것은 아닙니다. 다른 모든 표시는 해당 소유자의 자산입니다.

COPYRIGHT © 1983–2001 Borland Software Corporation.All rights reserved. All Borland products are trademarks or registered trademarks of Inprise Corporation.

Printed in the U.S.A.

CPE1340WW21001 4E0R1001
0102030405-9 8 7 6 5 4 3 2 1
D3

목차

1장
서문 **1-1**
안내서 내용 1-1
설명서 규칙 1-3
개발자 지원 서비스 1-3
인쇄된 설명서 주문 1-3

I 부
C++Builder 프로그래밍

2장
C++Builder를 사용한 애플리케이션 개발 **2-1**
통합 개발 환경 2-1
애플리케이션 디자인 2-2
프로젝트 생성 2-3
코드 편집 2-3
애플리케이션 컴파일 2-4
애플리케이션 디버깅 2-4
애플리케이션 배포 2-5

3장
클래스 라이브러리 사용 **3-1**
클래스 라이브러리 이해 3-1
속성, 메소드 및 이벤트 3-2
속성 3-2
메소드 3-3
이벤트 3-3
사용자 이벤트 3-3
시스템 이벤트 3-3
객체, 컴포넌트 및 컨트롤 3-3
TObject 분기 3-5
TPersistent 분기 3-6
TComponent 분기 3-7
TControl 분기 3-8
TWinControl/TWidgetControl 분기 3-9

4장
BaseCLX 사용 **4-1**
스트림 사용 4-2
스트림을 사용하여 데이터 읽기 또는 쓰기 . . 4-2
읽기 및 쓰기용 스트림 메소드 4-2
컴포넌트 읽기 및 쓰기 4-3
한 스트림에서 다른 스트림으로 데이터 복사 4-3
스트림 위치 및 크기 지정 4-3
특정 위치 찾기 4-4
Position 및 Size 속성 사용 4-4
파일 작업 4-4
파일 I/O에 대한 접근 방법 4-5
파일 스트림 사용 4-5
파일 스트림을 사용하여 파일 생성 및 열기 4-6
파일 핸들 사용 4-7
파일 처리 4-7
파일 삭제 4-7
파일 찾기 4-7
파일 이름 변경 4-9
파일의 date-time 루틴 4-9
파일 복사 4-9
ini 파일 및 시스템 레지스트리 작업 4-10
TIniFile 및 TMemIniFile 사용 4-10
TRegistryIniFile 사용 4-11
TRegistry 사용 4-12
리스트 작업 4-12
일반적인 리스트 작업 4-13
리스트 항목 추가 4-13
리스트 항목 삭제 4-14
리스트 항목 액세스 4-14
리스트 항목 재정렬 4-14
영구적(persistent) 리스트 4-14
문자열 리스트 작업 4-15
문자열 리스트의 로드 및 저장 4-15
새 문자열 리스트 작성 4-16
기간이 짧은 문자열 리스트 4-16
기간이 긴 문자열 리스트 4-16
리스트에서 문자열 처리 4-17
리스트의 문자열 수 계산 4-18
특정 문자열 액세스 4-18
문자열 리스트에서 항목 찾기 4-18
리스트에서 문자열을 통한 반복 4-18
리스트에 문자열 추가 4-18
리스트 내에서 문자열 이동 4-18
리스트에서 문자열 삭제 4-19
문자열 리스트에 객체 연결 4-19
문자열 작업 4-19
와이드 문자 루틴 4-20
AnsiStrings에 대해 일반적으로 사용되는 루틴 4-21
Null 종료 문자열에 일반적으로 사용되는 루틴 4-23

인쇄 4-25
측정값 변환 4-26
변환 수행 4-26
단순한 변환 수행 4-26
복잡한 변환 수행 4-26
새 측정 타입 추가 4-27
간단한 변환 패밀리 작성 및 단위 추가 4-27
변수 선언 4-27
변환 패밀리 등록 4-28
측정 단위 등록 4-28
새 단위 사용 4-28
변환 함수 사용 4-28
변수 선언 4-28
변환 패밀리 등록 4-29
기본 단위 등록 4-29
메소드를 작성하여 기본 단위 간의 변환 수행 4-29
다른 단위 등록 4-29
새 단위 사용 4-30
클래스를 사용하여 변환 관리 4-30
변환 클래스 작성 4-30
변수 선언 4-31
변환 패밀리 및 기타 유닛 등록 4-32
새 단위 사용 4-33
그리기 공간 생성 4-33

5장
컴포넌트 작업 5-1
컴포넌트 속성 설정 5-2
디자인 타임에 속성 설정 5-2
속성 에디터 사용 5-2
런타임에 속성 설정 5-3
메소드 호출 5-3
이벤트 및 이벤트 핸들러 작업 5-3
새 이벤트 핸들러 생성 5-4
컴포넌트의 디폴트 이벤트에 대한 핸들러 생성 5-4
이벤트 핸들러 찾기 5-4
이벤트를 기존 이벤트 핸들러에 연결 5-4
Sender 매개변수 사용 5-5
공유 이벤트 표시 및 코딩 5-5
메뉴 이벤트를 이벤트 핸들러에 연결 5-5
이벤트 핸들러 삭제 5-6
크로스 플랫폼 컴포넌트 및 크로스 플랫폼이 아닌 컴포넌트 5-6
컴포넌트 팔레트에 사용자 정의 컴포넌트 추가 5-8

6장
컨트롤 작업 6-1
컨트롤에서 드래그 앤 드롭 구현 6-1
끌기 작업 시작 6-1
끌어온 항목 승인 6-2
항목 놓기 6-2
끌기 작업 끝내기 6-3
끌기 객체로 드래그 앤 드롭 사용자 정의 6-3
끌기 마우스 포인터 변경 6-4
컨트롤에 드래그 앤 도킹 구현 6-4
윈도우 컨트롤을 도킹 사이트로 만들기 6-4
컨트롤을 도킹 가능한 자식으로 만들기 6-4
자식 컨트롤의 도킹 방법 제어 6-5
자식 컨트롤의 도킹 해제 방법 제어 6-6
드래그 앤 도킹 작업에 대한 자식 컨트롤의 응답 방법 제어 6-6
컨트롤 내에서 텍스트 작업 6-6
텍스트 정렬 설정 6-7
런타임 시 스크롤 막대 추가 6-7
클립보드 객체 추가 6-8
텍스트 선택 6-8
모든 텍스트 선택 6-8
텍스트 잘라내기, 복사 및 붙여넣기 6-9
선택한 텍스트 삭제 6-9
메뉴 항목 비활성화 6-9
팝업 메뉴 제공 6-10
OnPopup 이벤트 처리 6-10
컨트롤에 그래픽 추가 6-11
컨트롤이 owner-draw 항목임을 나타내기 . . 6-12
문자열 리스트에 그래픽 객체 추가 6-12
애플리케이션에 이미지 추가 6-12
문자열 리스트에 이미지 추가 6-13
owner-draw 항목 그리기 6-13
owner-draw 항목의 크기 지정 6-14
owner-draw 항목 그리기 6-15

7장
애플리케이션, 컴포넌트 및 라이브러리 생성 7-1
애플리케이션 생성 7-1
GUI 애플리케이션 7-1
사용자 인터페이스 모델 7-2
SDI 애플리케이션 7-2
MDI 애플리케이션 7-2
IDE, 프로젝트 및 컴파일 옵션 설정 7-3
프로그래밍 템플릿 7-3

콘솔 애플리케이션 7-4
콘솔 애플리케이션에서 VCL 및 CLX 사용 7-4
서비스 애플리케이션 7-4
서비스 스레드 7-7
서비스 이름 속성 7-8
서비스 애플리케이션 디버깅 7-9
패키지 및 DLL 생성 7-10
패키지 및 DLL 사용 시기 7-10
C++Builder에서 DLL 사용 7-11
C++Builder에서 DLL 생성 7-11
VCL 및 CLX 컴포넌트를 포함하는 DLL 생성 . . 7-12
DLL 연결 7-15
데이터베이스 애플리케이션 생성 7-15
분산 데이터베이스 애플리케이션 7-16
웹 서버 애플리케이션 생성 7-16
Web Broker 사용 7-17
WebSnap 애플리케이션 생성 7-18
InternetExpress 사용 7-18
웹 서비스 애플리케이션 생성 7-18
COM을 사용한 애플리케이션 생성 7-19
COM 및 DCOM 사용 7-19
MTS 및 COM+ 사용 7-19
데이터 모듈 사용 7-20
표준 데이터 모듈 생성 및 편집 7-20
데이터 모듈 및 해당 유닛 파일 이름 지정 7-21
컴포넌트 배치 및 이름 지정 7-22
데이터 모듈에서 컴포넌트 속성 및 이벤트 사용 7-22
데이터 모듈에서 비즈니스 룰 생성 7-23
폼에서 데이터 모듈 액세스 7-23
애플리케이션 서버 프로젝트에 원격 데이터 모듈 추가 7-23
Object Repository 사용 7-24
프로젝트 내에서 항목 공유 7-24
Object Repository에 항목 추가 7-24
팀 환경에서 객체 공유 7-25
프로젝트에서 Object Repository 항목 사용 . . 7-25
항목 복사 7-25
항목 상속 7-25
항목 사용 7-25
프로젝트 템플릿 사용 7-26
공유 항목 수정 7-26
디폴트 프로젝트, 새 폼 및 메인 폼 지정 7-26
애플리케이션에서 도움말 사용 7-27
도움말 시스템 인터페이스 7-27
ICustomHelpViewer 구현 7-28
Help Manager와 통신 7-28
Help Manager에 정보 요청 7-29
키워드 방식 도움말 표시 7-29
목차 표시 7-30
IExtendedHelpViewer 구현 7-30
IhelpSelector 구현 7-31
도움말 시스템 객체 등록 7-32
도움말 뷰어 등록 7-32
도움말 선택자 등록 7-32
VCL 애플리케이션에서 도움말 사용 7-33
TApplication이 VCL 도움말을 처리하는 방법 7-33
VCL 컨트롤이 도움말을 처리하는 방법 7-33
CLX 애플리케이션에서 도움말 사용 7-34
TApplication이 도움말을 처리하는 방법 . . . 7-34
CLX 컨트롤이 도움말을 처리하는 방법 7-34
도움말 시스템 직접 호출 7-35
IHelpSystem 사용 7-35
IDE 도움말 시스템 사용자 정의 7-36

8장

애플리케이션 사용자 인터페이스 개발 8-1

애플리케이션 동작 제어 8-1
애플리케이션 레벨 작업 8-1
화면 처리 8-2
폼 설정 8-2
메인 폼 사용 8-2
메인 폼 숨기기 8-3
폼 추가 8-3
폼 연결 8-3
레이아웃 관리 8-3
폼 사용 8-4
메모리에 있는 폼 제어 8-5
자동 생성된 폼 표시 8-5
동적으로 폼 생성 8-5
윈도우 등의 modeless 폼 생성 8-6
로컬 변수를 사용하여 폼 인스턴스 생성 . . 8-6
폼에 추가 인수 전달 8-7
폼에서 데이터 검색 8-8
modeless 폼에서 데이터 검색 8-8
modal 폼에서 데이터 검색 8-9
컴포넌트와 컴포넌트 그룹의 재사용 8-11
컴포넌트 템플릿 생성 및 사용 8-12
프레임 사용 8-13
프레임 생성 8-13
컴포넌트 팔레트에 프레임 추가 8-13
프레임 사용 및 수정 8-14
프레임 공유 8-15

다이얼로그 박스 개발 8-15
Open 다이얼로그 박스 사용 8-15
툴바와 메뉴에 대한 액션 구성 8-16
액션의 개념 8-17
액션 밴드 설정 8-18
툴바 및 메뉴 생성 8-19
메뉴, 버튼 및 툴바에 색, 패턴 또는 그림 추가 8-20
메뉴와 툴바에 아이콘 추가 8-21
사용자가 정의할 수 있는 툴바 및 메뉴 생성 8-21
액션 밴드에서 사용하지 않는 항목 및 범주 숨기기 8-22
액션 리스트 사용 8-23
액션 리스트 설정 8-23
액션 실행 시 발생하는 이벤트 8-24
이벤트에 응답 8-24
액션이 해당 대상을 찾는 방법 8-26
액션 업데이트 8-26
미리 정의된 액션 클래스 8-26
액션 컴포넌트 작성 8-28
액션 등록 8-28
메뉴 생성 및 관리 8-29
메뉴 디자이너 열기 8-30
메뉴 생성 8-30
메뉴 이름 지정 8-31
메뉴 항목 이름 지정 8-31
메뉴 항목 추가, 삽입 및 삭제 8-31
구분자 표시줄 추가 8-33
가속키 및 단축키 지정 8-33
하위 메뉴 생성 8-33
기존 메뉴를 밑으로 내려 하위 메뉴 생성 8-34
메뉴 항목 이동 8-34
메뉴 항목에 이미지 추가 8-35
메뉴 보기 8-35
Object Inspector에서 메뉴 항목 편집 8-36
메뉴 디자이너 컨텍스트 메뉴 사용 8-36
컨텍스트 메뉴의 명령 8-36
디자인 타임 시 메뉴 간의 전환 8-37
메뉴 템플릿 사용 8-38
메뉴를 템플릿으로 저장 8-39
템플릿 메뉴 항목 및 이벤트 핸들러의 이름 지정 규칙 8-39
런타임 시 메뉴 항목 처리 8-40
메뉴 병합 8-40
활성 메뉴 지정: Menu 속성 8-40
병합된 메뉴 항목의 순서 결정: GroupIndex 속성 8-40
리소스 파일 임포트 8-41
툴바 및 쿨바(cool bar) 디자인 8-42
패널 컴포넌트를 사용하여 툴바 추가 8-43
패널에 스피드 버튼 추가 8-43
스피드 버튼의 문자 모양 할당 8-43
스피드 버튼의 초기 상태 설정 8-44
스피드 버튼 그룹 생성 8-44
버튼 토글 허용 8-44
툴바 컴포넌트를 사용하여 툴바 추가 8-45
툴 버튼 추가 8-45
툴 버튼에 이미지 할당 8-45
툴 버튼 모양 및 초기 상태 설정 8-46
툴 버튼 그룹 생성 8-46
툴 버튼 토글 허용 8-46
쿨바(cool bar) 컴포넌트 추가 8-47
쿨바(cool bar)의 모양 설정 8-47
클릭 이벤트에 응답 8-48
툴 버튼에 메뉴 할당 8-48
숨겨진 툴바 추가 8-48
툴바 숨기기 및 표시 8-48

9장
컨트롤 타입 9-1
텍스트 컨트롤 9-1
편집 컨트롤 9-2
편집 컨트롤 속성 9-2
메모 및 리치 에디트(rich edit) 컨트롤 9-3
텍스트 보기 컨트롤(CLX만 해당) 9-3
레이블 9-4
특화된 입력 컨트롤 9-4
스크롤 막대 9-4
트랙 표시줄 9-5
업다운(up-down) 컨트롤(VCL만 해당) 9-5
스핀 편집 컨트롤(CLX만 해당) *9-5*
단축키 컨트롤(VCL만 해당) 9-6
스플리터 컨트롤 9-6
버튼 및 이와 유사한 컨트롤 9-6
버튼 컨트롤 9-7
비트맵 버튼 9-7
스피드 버튼 9-7
체크 박스 9-8
라디오 버튼 9-8
툴바 9-8
쿨바(cool bar)(VCL만 해당) 9-8
리스트 컨트롤 9-9
리스트 박스와 체크 리스트 박스 9-9

콤보 박스 9-10
트리 뷰 9-11
리스트 뷰 9-11
Date-Time Picker 및 Month Calendar (VCL만 해당) 9-11
그룹화 컨트롤 9-12
그룹 박스와 라디오 그룹 9-12
패널 9-12
스크롤 박스 9-13
탭 컨트롤 9-13
페이지 컨트롤 9-13
헤더 컨트롤 9-14
디스플레이 컨트롤 9-14
상태 표시줄 9-14
진행 표시줄 9-14
도움말과 힌트 속성 9-15
그리드 9-15
그리기 그리드 9-15
문자열 그리드 9-16
값 리스트 에디터(VCL만 해당) 9-16
그래픽 컨트롤 9-17
이미지 9-17
도형 9-17
3D 9-17
그리기 박스 9-17
애니메이션 컨트롤 (VCL만 해당) 9-17

10장

그래픽 및 멀티미디어 작업 10-1

그래픽 프로그래밍 개요 10-1
화면 새로 고침 10-2
그래픽 객체 타입 10-3
캔버스의 일반적인 속성과 메소드 10-4
캔버스 객체의 속성 사용 10-5
펜 사용 10-5
브러시 사용 10-8
픽셀 읽기 및 설정 10-9
캔버스 메소드를 사용하여 그래픽 객체 그리기 10-10
선 및 다각선 그리기 10-10
도형 그리기 10-11
애플리케이션에서 여러 그리기 객체 처리 . . 10-12
사용할 드로잉 툴 파악 10-12
스피드 버튼으로 툴 변경 10-13
드로잉 툴 사용 10-14
그래픽에서 그리기 10-17
스크롤할 수 있는 그래픽 만들기 10-17
이미지 컨트롤 추가 10-17
그래픽 파일 로드 및 저장 10-19
파일에서 그림 로드 10-20
파일에 그림 저장 10-20
그림 바꾸기 10-21
그래픽에서 클립보드 사용 10-22
클립보드에 그래픽 복사 10-22
그림을 잘라내어 클립보드에 넣기 10-22
클립보드에서 그림 붙여넣기 10-23
Rubber banding 예제 10-24
마우스에 응답 10-24
마우스 다운 동작에 응답 10-25
마우스 동작 추적을 위해 폼 객체에 필드 추가 10-26
선 그리기 다듬기 10-27
멀티미디어 작업 10-29
애플리케이션에 무성 비디오 클립 추가 . . . 10-29
무성 비디오 클립 추가 예제 10-30
애플리케이션에 오디오 및/또는 비디오 클립 추가 10-31
오디오 및/또는 비디오 클립 추가 예제 (VCL만 해당) 10-33

11장

멀티 스레드 애플리케이션 개발 11-1

스레드 객체 정의 11-1
스레드 초기화 11-2
디폴트 우선 순위 할당 11-3
스레드 해제 시기 표시 11-3
스레드 함수 작성 11-4
메인 VCL/CLX 스레드 사용 11-4
스레드 로컬 변수 사용 11-5
다른 스레드로 종료 확인 11-6
스레드 함수에서의 예외 처리 11-6
지우기 코드 작성 11-7
스레드 조정 11-7
동시 액세스 피하기 11-7
객체 잠금 11-7
임계 구역 사용 11-8
동시 읽기는 허용하고 쓰기는 배타적인 동기화 장치(multi-read exclusive-write synchronizer) 사용 11-8
메모리 공유에 이용하는 기타 기술 11-9
다른 스레드 기다리기 11-9
스레드의 실행 종료 대기 11-9
작업 완료 대기 11-10
스레드 객체 실행 11-11
디폴트 우선 순위 오버라이드 11-11
스레드 시작 및 중지 11-11

멀티 스레드 애플리케이션 디버깅 11-12
스레드 이름 지정 11-12
이름 없는 스레드를 이름이 지정된 스레드로 변환 11-12
비슷한 스레드에 각각의 이름 할당 11-13

12장
예외 처리 12-1
C++ 예외 처리 12-1
예외 처리 구문 12-1
try 블록 12-2
throw 문 12-2
catch 문 12-3
예외 다시 throw 12-4
예외 규정 12-4
예외 해제 12-5
안전 포인터 12-5
예외 처리에서의 생성자 12-5
catch되지 않은 예외와 예기치 않은 예외 처리 12-6
Win32에서의 구조적 예외 12-6
구조적 예외 구문 12-7
구조적 예외 처리 12-8
예외 필터 12-8
C++와 구조적 예외 혼합 12-10
C++ 프로그램 예제에서의 C 기반 예외 . 12-11
예외 정의 12-12
예외 발생 12-12
종료 블록 12-13
C++Builder 예외 처리 옵션 12-14
VCL/CLX 예외 처리 12-15
C++ 및 VCL/CLX 예외 처리 간의 차이점 . . 12-15
운영 체제 예외 처리 12-15
VCL 및 CLX 예외 처리 12-16
VCL 및 CLX 예외 클래스 12-16
이식성 고려 사항 12-18

13장
VCL 및 CLX에 대한 C++ 랭귀지 지원 13-1
C++ 및 오브젝트 파스칼 객체 모델 13-1
상속 및 인터페이스 13-2
복수 상속 대신 인터페이스 사용 13-2
인터페이스 클래스 선언 13-2
IUnknown 및 IInterface 13-3
IUnknown을 지원하는 클래스 생성 13-4
인터페이스 클래스 및 수명 관리 13-5
객체 구분 인스턴스화 13-5
C++ 및 오브젝트 파스칼 참조 구별 13-5
객체 복사 13-6
함수 인수로서의 객체 13-7
C++Builder VCL/CLX 클래스의 객체 생성 . 13-7
C++ 객체 생성 13-7
오브젝트 파스칼 객체 생성 13-8
C++Builder 객체 생성 13-8
기본 클래스 생성자에서의 가상 메소드 호출 13-10
오브젝트 파스칼 모델 13-11
C++ 모델 13-11
C++Builder 모델 13-11
예제: 가상 메소드 호출 13-11
가상 함수의 데이터 멤버 초기화 13-12
객체 소멸 13-13
생성자에서 발생되는 예외 13-13
소멸자에서 호출되는 가상 메소드 13-14
AfterConstruction 및 BeforeDestruction . . 13-15
클래스 가상 함수 13-15
오브젝트 파스칼 데이터 타입 및 랭귀지 개념 지원 13-15
Typedef 13-16
오브젝트 파스칼 랭귀지를 지원하는 클래스 13-16
오브젝트 파스칼 랭귀지에 대한 C++ 랭귀지 대응 부분 13-16
Var 매개변수 13-16
타입이 지정되지 않은 매개변수 13-17
개방형 배열 13-17
요소 수 계산 13-17
임시 개방형 배열 13-18
array of const 13-18
OPENARRAY 매크로 13-19
EXISTINGARRAY 매크로 13-19
개방형 배열 인수를 가져오는 C++ 함수 13-19
다르게 정의되는 타입 13-20
부울 데이터 타입 13-20
Char 데이터 타입 13-20
Delphi 인터페이스 13-20
리소스 문자열 13-21
디폴트 매개변수 13-21
런타임 타입 정보 13-22
매핑되지 않은 타입 13-23
6바이트 실수 타입 13-23
함수의 반환 타입으로서의 배열 13-23
키워드 확장 13-23
__classid 13-23
__closure 13-24

__property 13-26
__published 13-27
__declspec 키워드 확장 13-28
__declspec(delphiclass) 13-28
__declspec(delphireturn) 13-29
__declspec(delphirtti) 13-29
__declspec(dynamic) 13-29
__declspec(hidesbase) 13-29
__declspec(package) 13-29
__declspec(pascalimplementation) 13-30
__declspec(uuid) 13-30

14장

크로스 플랫폼 애플리케이션 개발 14-1

크로스 플랫폼 애플리케이션 생성 14-1
Windows 애플리케이션을 Linux로 이식 14-2
이식 기법 14-2
플랫폼별 이식 14-2
크로스 플랫폼 이식 14-3
Windows 에뮬레이션 이식 14-3
애플리케이션 이식 14-3
CLX와 VCL 비교 14-5
CLX에서 다르게 구현되는 기능 14-5
룩앤필(Look and feel) 14-6
스타일 14-6
가변 타입 14-6
레지스트리 14-6
그 밖의 차이점 14-7
CLX에서 구현되지 않는 기능 14-7
직접 이식되지 않는 기능 14-8
CLX 및 VCL 유닛 비교 14-8
CLX 객체 생성자의 차이점 14-11
시스템 및 widget 이벤트 처리 14-12
Windows와 Linux 간의 소스 파일 공유 . . . 14-12
Windows와 Linux 간의 환경적 차이점 . . . 14-13
Linux의 디렉토리 구조 14-15
이식 가능한 코드 작성 14-15
조건 지시어 사용 14-16
메시지 표시 14-17
인라인 어셈블러 코드 포함 14-18
Linux에서의 프로그래밍 차이점 14-19
크로스 플랫폼 데이터베이스 애플리케이션 . . 14-19
dbExpress 차이점 14-20
컴포넌트 레벨 차이점 14-21
사용자 인터페이스 레벨 차이점 14-21
데이터베이스 애플리케이션을 Linux로 이식 14-22
dbExpress 애플리케이션에서 데이터 업데이트 14-24
크로스 플랫폼 인터넷 애플리케이션 14-25
인터넷 애플리케이션을 Linux로 이식 14-26

15장

패키지와 컴포넌트 사용 15-1

패키지를 사용하는 이유 15-2
패키지 및 표준 DLL 15-2
런타임 패키지 15-3
애플리케이션에서 패키지 사용 15-3
동적 패키지 로딩 15-4
사용할 런타임 패키지 결정 15-4
사용자 정의 패키지 15-4
디자인 타임 패키지 15-5
컴포넌트 패키지 설치 15-5
패키지 생성 및 편집 15-6
패키지 생성 15-6
기존 패키지 편집 15-7
패키지 소스 파일 및 프로젝트 옵션 파일 . . . 15-8
컴포넌트 패키지화 15-8
패키지 구조 이해 15-9
패키지 이름 지정 15-9
Requires 리스트 15-9
Contains 리스트 15-10
패키지 생성 15-10
패키지별 컴파일러 지시어 15-11
명령줄 컴파일러와 링커 사용 15-12
컴파일하여 생성된 패키지 파일 15-12
패키지 배포 15-13
패키지를 사용하는 애플리케이션 배포 . . . 15-13
다른 개발자에게 패키지 배포 15-13
패키지 컬렉션 파일 15-13

16장

국제적인 애플리케이션 생성 16-1

국제화 및 지역화 16-1
국제화 16-1
지역화 16-1
애플리케이션 국제화 16-2
애플리케이션 코드 활성화 16-2
문자 집합 16-2
OEM 및 ANSI 문자 집합 16-2
멀티바이트 문자 집합 16-3
와이드 문자 16-3
애플리케이션에 양방향 기능 포함 16-4
BiDiMode 속성 16-6
로케일 특정 기능 16-8

사용자 인터페이스 디자인 16-8
텍스트 16-8
그래픽 이미지 16-9
형식 및 정렬 순서 16-9
키보드 매핑 16-9
리소스 분리 16-9
리소스 DLL 생성 16-10
리소스 DLL 사용 16-10
리소스 DLL의 동적 전환 16-12
애플리케이션 지역화 16-12
리소스 지역화 16-12

17장
애플리케이션 배포 17-1
일반적인 애플리케이션 배포 17-1
설치 프로그램 사용 17-2
애플리케이션 파일 식별 17-2
애플리케이션 파일 17-3
패키지 파일 17-3
병합 모듈 17-3
ActiveX 컨트롤 17-5
Helper 애플리케이션 17-5
DLL 위치 17-5
CLX 애플리케이션 배포 17-6
데이터베이스 애플리케이션 배포 17-6
dbExpress 데이터베이스 애플리케이션 배포 17-7
BDE 애플리케이션 배포 17-8
Borland Database Engine 17-8
SQL Links 17-9
멀티 티어 데이터베이스 애플리케이션 (DataSnap) 배포 17-9
웹 애플리케이션 배포 17-10
Apache 서버에 배포 17-10
다양한 호스트 환경을 위한 프로그래밍 17-11
화면 해상도와 색상 깊이 17-12
동적으로 크기가 조정되지 않는 경우의 고려 사항 17-12
폼과 컨트롤 크기를 동적으로 조정하는 경우의 고려 사항 17-12
다양한 색상 깊이 지원 17-13
글꼴 17-14
운영 체제 버전 17-14
소프트웨어 사용권 요구 사항 17-15
배포 17-15
README 17-15
사용권 계약서 17-15
서드파티 제품 설명서 17-15

II 부
데이터베이스 애플리케이션 개발

18장
데이터베이스 애플리케이션 디자인 18-1
데이터베이스 사용 18-1
데이터베이스의 타입 18-2
데이터베이스 보안 18-4
트랜잭션 18-4
참조 무결성, 내장 프로시저 및 트리거 18-5
데이터베이스 아키텍처 18-5
일반적인 구조 18-5
사용자 인터페이스 폼 18-6
데이터 모듈 18-6
데이터베이스 서버에 직접 연결 18-7
디스크에 있는 전용 파일 사용 18-9
다른 데이터셋 연결 18-10
클라이언트 데이터셋을 같은 애플리케이션의 다른 데이터셋에 연결 18-11
멀티 티어 아키텍처 사용 18-12
방법 조합(Combining approaches) 18-14
사용자 인터페이스 디자인 18-15
데이터 분석 18-15
보고서 작성 18-15

19장
데이터 컨트롤 사용 19-1
공통 데이터 컨트롤 기능 사용 19-2
데이터 컨트롤을 데이터셋과 연결 19-3
런타임 시 연결 데이터셋 변경 19-3
데이터 소스 활성화 및 비활성화 19-4
데이터 소스에 따른 변경 내용에 대한 응답 19-4
데이터 편집 및 업데이트 19-5
사용자 엔트리의 컨트롤에서 편집 활성화 19-5
컨트롤에서 데이터 편집 19-5
데이터 표시 활성화 및 비활성화 19-6
데이터 표시 새로 고침 19-6
마우스, 키보드 및 타이머 이벤트 활성화 . . . 19-7
데이터 구성 방법 선택 19-7
단일 레코드 표시 19-7
데이터를 레이블로 표시 19-8
에디트 박스에서 필드 표시 및 편집 19-8
메모 컨트롤에서 텍스트 표시 및 편집 . . . 19-8
리치 에디트(rich edit) 메모 컨트롤에서 텍스트 표시 및 편집 19-9

이미지 컨트롤에서 그래픽 필드 표시 및 편집 19-9
리스트 박스와 콤보 박스에서 데이터 표시 및 편집 19-10
체크 박스와 함께 부울 필드 값 처리 . . . 19-13
라디오 컨트롤을 사용하여 필드 값 제한 19-13
여러 레코드 표시 19-14
TDBGrid를 사용하여 데이터 보기 및 편집 . . . 19-15
디폴트 상태로 그리드 컨트롤 사용 19-15
사용자 정의된 그리드 생성 19-16
영구적 열 이해 19-16
영구적 열 생성 19-17
영구적 열 삭제 19-18
영구적 열의 순서 정렬 19-18
디자인 타임 시 열 속성 설정 19-19
조회 리스트 열 정의 19-20
열에 단추 놓기 19-21
열에 대한 기본값 복구 19-21
ADT 및 배열 필드 표시 19-22
그리드 옵션 설정 19-24
그리드에서 편집 19-25
그리드 그리기 제어 19-25
런타임 시 사용자 동작에 응답 19-26
기타 데이터 인식 컨트롤을 포함하는 그리드 생성 19-27
레코드 탐색 및 처리 19-28
표시할 탐색기 버튼 선택 19-29
디자인 타임 시 탐색기 버튼 표시 및 숨기기 19-29
런타임 시 탐색기 표시 및 숨기기 19-29
플라이오버(fly-over) 도움말 표시 19-30
여러 데이터셋에서 단일 탐색기 사용 19-30

20장
decision 지원 컴포넌트 사용 20-1
개요 20-1
크로스탭 정보 20-2
1차원 크로스탭 20-3
다차원 크로스탭 20-3
decision 지원 컴포넌트 사용 지침 20-3
decision 지원 컴포넌트를 사용하여 데이터셋 사용 20-4
TQuery나 TTable을 사용하여 decision 데이터셋 생성 20-5
Decision Query Editor를 사용하여 decision 데이터셋 생성 20-6
decision cube 사용 20-7
decision cube 속성과 이벤트 20-7
Decision Cube Editor 사용 20-7
차원 설정 보기 및 변경 20-7
사용 가능한 최대 차원 및 요약 수 설정 . . 20-8
디자인 옵션 보기 및 변경 20-8
decision source 사용 20-8
속성과 이벤트 20-9
decision pivot 사용 20-9
decision pivot 속성 20-10
decision grid 생성 및 사용 20-10
decision grid 생성 20-10
decision grid 사용 20-11
decision grid 필드 열기 및 닫기 20-11
decision grid의 행과 열 재구성 20-11
decision grid에서 세부 사항을 드릴다운 20-11
decision grid의 차원 선택 제한 20-12
decision grid 속성 20-12
decision 그래프 생성 및 사용 20-13
decision 그래프 생성 20-13
decision 그래프 사용 20-13
decision 그래프 표시 20-14
decision 그래프 사용자 정의 20-15
decision 그래프 템플릿 기본값 설정 . . . 20-16
decision 그래프 시리즈 사용자 정의 . . . 20-17
런타임 시 decision 지원 컴포넌트 20-18
런타임 시 decision pivot 20-18
런타임 시 decision grid 20-18
런타임 시 decision 그래프 20-19
decision 지원 컴포넌트와 메모리 제어 20-19
최대 차원, 요약 및 셀 설정 20-19
차원 상태 설정 20-19
페이징된 차원 사용 20-20

21장
데이터베이스에 연결 21-1
암시적 연결 사용 21-2
연결 제어 21-2
데이터베이스 서버에 연결 21-3
데이터베이스 서버로부터 연결 끊기 21-3
서버 로그인 제어 21-4
트랜잭션 관리 21-6
트랜잭션 시작 21-6
트랜잭션 끝내기 21-7
성공적인 트랜잭션 끝내기 21-8
실패한 트랜잭션 끝내기 21-8
트랜잭션 분리 레벨 지정 21-9

서버에 명령 보내기 21-10
연결된 데이터셋을 사용한 작업 21-11
서버와의 연결을 끊지 않고 모든 데이터셋 닫기 21-11
연결된 데이터셋을 통한 반복 21-12
메타데이터 얻기 21-12
사용 가능한 테이블 열거 21-12
테이블의 필드 열거 21-13
사용 가능한 내장 프로시저 열거 21-13
사용 가능한 인덱스 열거 21-13
내장 프로시저 매개변수 열거 21-13

22장

데이터셋 이해 22-1

TDataSet 자손 사용 22-2
데이터셋 상태 결정 22-2
데이터셋 열기 및 닫기 22-4
데이터셋 탐색 22-4
First와 Last 메소드 사용 22-6
Next와 Prior 메소드 사용 22-6
MoveBy 메소드 사용 22-6
Eof와 Bof 속성 사용 22-7
Eof 22-7
Bof 22-8
레코드 표시 및 레코드로 돌아가기 22-9
Bookmark 속성 22-9
GetBookmark 메소드 22-9
GotoBookmark 및 BookmarkValid 메소드 22-9
CompareBookmarks 메소드 22-9
FreeBookmark 메소드 22-9
북마킹 예제 22-10
데이터셋 검색 22-10
Locate 사용 22-10
조회 사용 22-11
필터를 사용하여 데이터의 부분 집합 표시 및 편집 22-12
필터링 활성화 및 비활성화 22-13
필터 생성 22-13
Filter 속성 설정 22-14
OnFilterRecord 이벤트 핸들러 작성 . . . 22-15
런타임 시 필터 이벤트 핸들러 전환 22-15
필터 옵션 설정 22-15
필터링된 데이터셋에서 레코드 탐색 22-16
데이터 수정 22-17
레코드 편집 22-17
새 레코드 추가 22-18
레코드 삽입 22-19
레코드 추가 22-19
레코드 삭제 22-19
데이터 포스트 22-20
변경 취소 22-20
전체 레코드 수정 22-21
필드 계산 22-22
데이터셋의 타입 22-23
테이블 타입 데이터셋 사용 22-24
테이블 타입 데이터셋의 장점 22-25
인덱스를 사용하여 레코드 정렬 22-25
인덱스 관련 정보 얻기 22-26
IndexName을 사용하여 인덱스 지정 . . 22-26
IndexFieldNames를 사용하여 인덱스 생성 22-27
인덱스를 사용하여 레코드 검색 22-27
Goto 메소드를 사용하여 검색 수행 . . . 22-28
Find 메소드를 사용하여 검색 실행 22-28
성공적인 검색 이후 현재 레코드 지정 . . 22-29
부분 키 검색 22-29
검색 반복 또는 확장 22-29
범위를 사용하여 레코드 제한 22-30
범위와 필터의 차이점 이해 22-30
범위 지정 22-30
범위 수정 22-33
범위의 적용 또는 취소 22-33
마스터/디테일 관계 생성 22-34
테이블을 다른 데이터셋의 디테일로 만들기 22-34
중첩 디테일 테이블 사용 22-36
테이블의 읽기/쓰기 액세스 제어 22-37
테이블 생성 및 삭제 22-37
테이블 생성 22-37
테이블 삭제 22-39
테이블 비우기 22-39
테이블 동기화 22-40
쿼리 타입 데이터셋 사용 22-40
쿼리 지정 22-41
SQL 속성을 사용하여 쿼리 지정 22-42
CommandText 속성을 사용하여 쿼리 지정 22-43
쿼리 매개변수 사용 22-43
디자인 타임 시 매개변수 제공 22-44
런타임 시 매개변수 제공 22-45
매개변수를 사용하여 마스터/디테일 관계 설정 22-45
쿼리 준비 22-46
결과 집합을 반환하지 않는 쿼리 실행 22-47
단방향 결과 집합 사용 22-48
내장 프로시저 타입의 데이터셋 사용 22-48
내장 프로시저 매개변수 사용 22-49

디자인 타임 시 매개변수 설정 22-50
런타임 시 매개변수 사용 22-51
내장 프로시저 준비 22-52
결과 집합을 반환하지 않는 내장 프로시저 실행 22-52
여러 결과 집합 가져오기 22-53

23장
필드 컴포넌트 작업 23-1
동적 필드 컴포넌트 23-2
영구적 필드(persistent field) 컴포넌트 23-3
영구적 필드(persistent field) 생성 23-4
영구적 필드(persistent field) 정렬 23-5
영구적 새 필드 정의 23-5
데이터 필드 정의 23-6
계산된 필드 정의 23-7
계산된 필드 프로그래밍 23-7
조회 필드 정의 23-8
집계 필드 정의 23-10
영구적 필드(persistent field) 컴포넌트 삭제 23-10
영구적 필드 속성 및 이벤트 설정 23-10
디자인 타임 시 표시 및 편집 속성 설정 . 23-11
런타임 시 필드 컴포넌트 속성 설정 23-12
필드 컴포넌트에 대한 어트리뷰트(attribute) 집합 생성 23-12
어트리뷰트(attribute) 집합과 필드 컴포넌트 연결 23-13
어트리뷰트(attribute) 연결 제거 23-14
사용자 입력 제어 및 마스킹 23-14
숫자, 날짜 및 시간 필드에 대한 디폴트 서식 사용 23-14
이벤트 처리 23-15
런타임 시 필드 컴포넌트 메소드 작업 23-16
필드 값 표시, 변환 및 액세스 23-17
표준 컨트롤에 필드 컴포넌트 값 표시 23-17
필드 값 변환 23-18
디폴트 데이터셋 속성으로 필드 값 액세스 23-19
데이터셋의 Fields 속성으로 필드 값 액세스 23-19
데이터셋의 FieldByName 메소드로 필드 값 액세스 23-20
필드에 대한 기본값 설정 23-20
제약 조건 사용 23-21
사용자 정의 제약 조건 생성 23-21
서버 제약 조건 사용 23-21
객체 필드 사용 23-22
ADT 및 배열 필드 표시 23-23
ADT 필드 작업 23-23
영구적 필드(persistent field) 컴포넌트 사용 23-23
데이터셋의 FieldByName 메소드 사용 . 23-24
데이터셋의 FieldValues 속성 사용 23-24
ADT 필드의 FieldValues 속성 사용 . . . 23-24
ADT 필드의 Fields 속성 사용 23-24
배열 필드 작업 23-24
영구적 필드(persistent field) 사용 23-25
배열 필드의 FieldValues 속성 사용 23-25
배열 필드의 Fields 속성 사용 23-25
데이터셋 필드 작업 23-25
데이터셋 필드 표시 23-25
중첩된 데이터셋의 데이터 액세스 23-26
참조 필드 작업 23-26
참조 필드 표시 23-26
참조 필드의 데이터 액세스 23-26

24장
Borland Database Engine 사용 24-1
BDE 기반 아키텍처 24-1
BDE 호환 데이터셋 사용 24-2
데이터베이스 및 세션 연결에 데이터셋 연결 24-3
BLOB 캐싱 24-4
BDE 핸들 가져오기 24-4
TTable 사용 24-4
로컬 테이블의 테이블 타입 지정 24-5
로컬 테이블에 대한 읽기/쓰기 액세스 제어 24-6
dBASE 인덱스 파일 지정 24-6
로컬 테이블 이름 변경 24-7
다른 테이블에서 데이터 임포트 24-7
TQuery 사용 24-8
이질적 쿼리 생성 24-9
편집할 수 있는 결과 집합 얻기 24-10
읽기 전용 결과 집합 업데이트 24-11
TStoredProc 사용 24-11
매개변수 연결 24-11
Oracle 오버로드된 내장 프로시저 작업 24-12
TDatabase를 사용한 데이터베이스 연결 . . 24-12
데이터베이스 컴포넌트를 세션에 연결 . 24-12
데이터베이스 및 세션 컴포넌트의 상호 작용 이해 24-13

데이터베이스 식별 24-13
Tdatabase를 사용하여 연결 열기 24-15
데이터 모듈의 데이터베이스 컴포넌트 사용 24-16
데이터베이스 세션 관리 24-16
세션 활성화 24-17
디폴트 데이터베이스 연결 동작 지정 . . 24-18
데이터베이스 연결 관리 24-19
암호 사용 Paradox 및 dBASE 테이블 . . . 24-21
Paradox 디렉토리 위치 지정 24-23
BDE 알리아스 작업 24-24
세션에 대한 정보 검색 24-26
추가 세션 생성 24-26
세션 이름 지정 24-27
여러 세션 관리 24-28
BDE에서 트랜잭션 사용 24-29
통과(passthrough) SQL 사용 24-30
로컬 트랜잭션 사용 24-31
BDE를 사용하여 업데이트 캐싱 24-31
BDE 기반의 캐싱된 업데이트 사용 24-33
BDE 기반의 캐싱된 업데이트 적용 24-33
데이터베이스를 사용하여 캐싱된 업데이트 적용 24-34
데이터셋 컴포넌트 메소드로 캐싱된 업데이트 적용 24-35
OnUpdateRecord 이벤트 핸들러 생성 . . 24-35
캐싱된 업데이트 오류 처리 24-37
업데이트 객체를 사용하여 데이터셋 업데이트 24-39
업데이트 컴포넌트를 위한 SQL 문 작성 24-40
여러 업데이트 객체 사용 24-43
SQL 문 실행 24-44
TBatchMove 사용 24-47
일괄 이동 컴포넌트 생성 24-47
일괄 이동 모드 지정 24-48
레코드 추가 24-48
레코드 업데이트 24-49
레코드 추가 및 업데이트 24-49
데이터셋 복사 24-49
레코드 삭제 24-49
데이터 타입 매핑 24-49
일괄 이동 실행 24-50
일괄 이동 오류 처리 24-50
Data Dictionary 24-51
BDE 작업 도구 24-53

25장

ADO 컴포넌트 사용 25-1
ADO 컴포넌트 개요 25-1
ADO 데이터 저장소에 연결 25-2
TADOConnection을 사용하여 데이터 저장소에 연결 25-3
연결 객체 액세스 25-4
연결 미세 조정 25-5
비동기 연결 강제 적용 25-5
시간 제한 제어 25-5
연결에서 지원하는 작업 타입 표시 25-6
연결 시 트랜잭션 자동 시작 여부 지정 . . 25-6
연결의 명령 액세스 25-7
ADO 연결 이벤트 25-7
연결을 설정할 때 발생하는 이벤트 25-7
연결을 끊을 때 발생하는 이벤트 25-8
트랜잭션을 관리할 때 발생하는 이벤트 . . 25-8
기타 이벤트 25-8
ADO 데이터셋 사용 25-8
데이터 저장소에 ADO 데이터셋 연결 . . . 25-9
레코드셋 작업 25-10
북마크를 사용한 레코드 필터링 25-10
레코드 비동기 페치 25-11
배치 업데이트 사용 25-11
파일에서 데이터 로드 및 파일에 데이터 저장 25-14
TADODataSet 사용 25-15
명령 객체 사용 25-16
명령 지정 25-17
Execute 메소드 사용 25-17
명령 취소. 25-17
명령을 사용하여 결과 집합 검색 25-18
명령 매개변수 처리 25-18

26장

단방향 데이터셋 사용 26-1
단방향 데이터셋의 타입 26-2
데이터베이스 서버에 연결 26-2
TSQLConnection 설정 26-3
드라이버 식별 26-3
연결 매개변수 지정 26-4
연결 이름 지정 설명 26-4
Connection Editor 사용 26-5
표시할 데이터 지정 26-5
쿼리의 결과 표시 26-6

테이블의 레코드 표시 26-6
TSQLDataSet을 사용하여 테이블 표시 . . . 26-6
TSQLTable을 사용하여 테이블 표시 26-7
내장 프로시저의 결과 표시 26-7
데이터 가져오기 26-8
데이터셋 준비 26-8
여러 데이터셋 가져오기 26-9
레코드를 반환하지 않는 명령 실행 26-9
실행할 명령 지정 26-9
명령 실행 26-10
서버 메타데이터 작성 및 수정 26-10
마스터/디테일 연결된 커서 설정 26-11
스키마 정보 액세스 26-12
메타데이터를 단방향 데이터셋으로 가져오기 26-12
메타데이터에 대한 데이터셋 사용 후 데이터 가져오기 26-13
메타데이터 데이터셋의 구조 26-13
dbExpress 애플리케이션 디버깅 26-17
TSQLMonitor를 사용하여 SQL 명령 모니터 26-17
콜백을 사용하여 SQL 명령 모니터 26-18

27장
클라이언트 데이터셋 사용 27-1
클라이언트 데이터셋을 사용하는 데이터 작업 . . 27-2
클라이언트 데이터셋에서 데이터 탐색 27-2
레코드 표시 제한 27-2
데이터 편집 27-5
변경 취소 27-5
변경 내용 저장 27-6
데이터 값 제약 27-6
사용자 정의 제약 조건 지정 27-7
정렬과 인덱싱 27-7
새 인덱스 추가 27-8
인덱스 삭제와 전환 27-9
인덱스를 사용하여 데이터 그룹화 27-9
계산된 값 표시 27-10
클라이언트 데이터셋에서 내부적으로 계산된 필드 사용 27-10
유지 보수된 집계 사용 27-11
집계 지정 27-11
여러 레코드 그룹의 집계 27-12
집계 값 얻기 27-13
다른 데이터셋에서 데이터 복사 27-13
데이터 직접 할당 27-14
클라이언트 데이터셋 커서 복제 27-14
데이터에 애플리케이션 특정 정보 추가 . . . 27-15
업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용 27-15
캐싱된 업데이트 사용의 개요 27-16
업데이트 캐싱을 위한 데이터셋 타입 선택 . 27-17
수정된 레코드 표시 27-18
레코드 업데이트 27-19
업데이트 적용 27-20
업데이트 적용 시 간섭 27-21
업데이트 오류 조정 27-22
프로바이더와 함께 클라이언트 데이터셋 사용 27-24
프로바이더 지정 27-24
소스 데이터셋이나 문서에서 데이터 요청 27-25
증분 페치 27-25
요청 시 가져오기 27-26
소스 데이터셋에서 매개변수 얻기 27-26
소스 데이터셋에 매개변수 전달 27-27
쿼리나 내장 프로시저 매개변수 보내기 27-28
매개변수로 레코드 제한 27-28
서버로부터의 제약 조건 처리 27-29
레코드 새로 고침 27-30
사용자 정의 이벤트를 사용하여 프로바이더와 통신 27-30
소스 데이터셋 오버라이드 27-31
파일 기반 데이터와 함께 클라이언트 데이터셋 사용 27-32
새로운 데이터셋 생성 27-32
파일이나 스트림에서 데이터 로드 27-33
데이터에 변경 내용 병합 27-33
파일이나 스트림에 데이터 저장 27-34

28장
프로바이더 컴포넌트 사용 28-1
데이터의 소스 정하기 28-2
데이터 소스로 데이터셋 사용 28-2
데이터 소스로 XML 문서 사용 28-2
클라이언트 데이터셋과의 통신 28-3
데이터셋 프로바이더를 사용하여 업데이트를 적용하는 방법 선택 28-4
데이터 패킷에 포함될 정보 제어 28-4
데이터 패킷에 표시되는 필드 지정 28-4
데이터 패킷에 영향을 주는 옵션 설정 28-5
데이터 패킷에 사용자 정의 정보 추가 28-6
클라이언트 데이터 요청에 대한 응답 28-7
클라이언트 업데이트 요청에 대한 응답 28-8
데이터베이스 업데이트 이전의 델타 패킷 편집 28-9

업데이트 적용 방법 변경 28-9
개별 업데이트 검열 28-11
프로바이더에서 업데이트 오류 해결 28-11
단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용 28-11
클라이언트 생성 이벤트에 응답 28-12
서버 제약 조건 처리 28-12

29장

멀티 티어 애플리케이션 생성 29-1

멀티 티어 데이터베이스 모델의 장점 29-2
프로바이더 기반 멀티 티어 애플리케이션 이해 29-2
3티어 애플리케이션의 개요 29-3
클라이언트 애플리케이션의 구조 29-4
애플리케이션 서버의 구조 29-5
원격 데이터 모듈의 내용 29-6
트랜잭션 데이터 모듈 사용 29-6
원격 데이터 모듈 풀링(pooling) 29-8
연결 프로토콜 선택 29-9
DCOM 연결 사용 29-9
소켓 연결 사용 29-9
웹 연결 사용 29-10
SOAP 연결 사용 29-11
멀티 티어 애플리케이션 생성 29-11
애플리케이션 서버 생성 29-12
원격 데이터 모듈 설정 29-13
트랜잭션이 아닐 경우 원격 데이터 모듈 구성 29-13
트랜잭션 원격 데이터 모듈 구성 29-14
TSoapDataModule 구성 29-15
애플리케이션 서버의 인터페이스 확장 . . . 29-16
애플리케이션 서버의 인터페이스에 콜백 추가 29-16
트랜잭션 애플리케이션 서버의 인터페이스 확장 29-17
멀티 티어 애플리케이션의 트랜잭션 관리 . . 29-17
마스터/디테일 관계 지원 29-18
원격 데이터 모듈에서 상태 정보 지원 29-18
여러 원격 데이터 모듈 사용 29-20
애플리케이션 서버 등록 29-21
클라이언트 애플리케이션 생성 29-21
애플리케이션 서버에 연결 29-22
DCOM을 사용하여 연결 지정 29-23
소켓을 사용하여 연결 지정 29-23
HTTP를 사용하여 연결 지정 29-24
SOAP를 사용하여 연결 지정 29-25
연결 브로커 29-25
서버 연결 관리 29-26
서버에 연결 29-26
서버 연결 끊기 또는 변경 29-26
서버 인터페이스 호출 29-27
여러 데이터 모듈을 사용하는 애플리케이션 서버에 연결 29-28
웹 기반 클라이언트 애플리케이션 생성 29-28
클라이언트 애플리케이션을 ActiveX 컨트롤로 분산 29-30
클라이언트 애플리케이션용 Active Form 생성 29-30
InternetExpress를 이용한 웹 애플리케이션 개발 29-31
InternetExpress 애플리케이션 생성 29-31
Javascript 라이브러리 사용 29-33
애플리케이션 서버를 액세스 및 시작할 수 있는 권한 부여 29-33
XML 브로커 사용 29-34
XML 데이터 패킷 가져오기 29-34
XML 델타 패킷의 업데이트 적용 29-35
InternetExpress 페이지 프로듀서로 웹 페이지 생성 29-36
웹 페이지 에디터 사용 29-37
웹 항목 속성 설정 29-37
InternetExpress 페이지 프로듀서 템플릿 사용자 정의 29-38

30장

데이터베이스 애플리케이션에서 XML 사용 30-1

변환 정의 30-1
XML 노드와 데이터 패킷 필드 사이의 매핑 30-2
XMLMapper 사용 30-4
XML 스키마 또는 데이터 패킷 로드 30-4
매핑 정의 30-4
변환 파일 생성 30-5
XML 문서를 데이터 패킷으로 변환 30-6
소스 XML 문서 지정 30-6
변환 지정 30-6
결과 데이터 패킷 가져오기 30-6
사용자 정의 노드 변환 30-7
XML 문서를 프로바이더의 소스로 사용 30-7
XML 문서를 프로바이더의 클라이언트로 사용 30-8
프로바이더로부터 XML 문서 가져오기 30-9
XML 문서의 업데이트 내용을 프로바이더에 적용 30-10

III 부
인터넷 애플리케이션 작성

31장
CORBA 애플리케이션 작성 31-1

CORBA 애플리케이션 개요 31-1
스텁 및 스켈레톤 이해 31-2
Smart Agent 사용 31-3
서버 애플리케이션 활성화 31-3
인터페이스 호출의 동적 연결 31-4
CORBA 서버 작성 31-4
객체 인터페이스 정의 31-5
CORBA 서버 마법사 사용 31-5
IDL 파일에서 스텁 및 스켈레톤 생성 31-6
CORBA Object Implementation 마법사 사용 31-6
CORBA 객체 인스턴스화 31-7
위임 모델 사용 31-8
변경 내용 확인 및 편집 31-9
CORBA 객체 구현 31-9
스레드 충돌 방지 31-11
CORBA 인터페이스 변경 31-12
서버 인터페이스 등록 31-12
CORBA 클라이언트 작성 31-13
스텁 사용 31-14
동적 호출 인터페이스 사용 31-15
CORBA 서버 테스트 31-16
테스트 도구 설정 31-16
테스트 스크립트 기록 및 실행 31-17

32장
인터넷 서버 애플리케이션 생성 32-1

Web Broker 및 WebSnap 32-1
용어 및 표준 32-3
URL(Uniform Resource Locator)의 각 부분 . . 32-3
URI와 URL 비교 32-4
HTTP 요청 헤더 정보 32-4
HTTP 서버의 동작 32-5
클라이언트 요청 작성 32-5
클라이언트 요청 처리 32-5
클라이언트 요청에 응답하기 32-6
웹 서버 애플리케이션 타입 32-6
ISAPI 및 NSAPI 32-6
CGI 독립 실행형 32-6
Win-CGI 독립 실행형 32-7
Apache 32-7
Web App Debugger 32-7
웹 서버 애플리케이션 대상 타입 변환 32-8
서버 애플리케이션 디버깅 32-9
Web Application Debugger 사용 32-9
애플리케이션을 다른 타입의 웹 서버 애플리케이션으로 변환 32-10
DLL 웹 애플리케이션 디버깅 32-10
DLL 디버깅에 필요한 사용자 권한 32-10

33장
Web Broker 사용 33-1

Web Broker를 사용하여 웹 서버 애플리케이션 생성 33-1
웹 모듈 33-2
웹 애플리케이션 객체 33-3
Web Broker 애플리케이션의 구조 33-3
웹 디스패처 33-4
디스패처에 액션 추가 33-5
요청 메시지 디스패칭 33-5
액션 항목 33-6
액션 항목이 실행되는 조건 33-6
대상 URL 33-6
요청 메소드 타입 33-6
액션 항목 활성화 및 비활성화 33-7
디폴트 액션 항목 선택 33-7
액션 항목으로 요청 메시지에 응답 33-8
응답 보내기 33-8
여러 액션 항목 사용 33-8
클라이언트 요청 정보 액세스하기 33-9
요청 헤더 정보를 포함하는 속성 33-9
대상을 식별하는 속성 33-9
웹 클라이언트를 설명하는 속성 33-9
요청 목적을 식별하는 속성 33-10
기대 응답을 설명하는 속성 33-10
컨텐트를 설명하는 속성 33-10
HTTP 요청 메시지의 컨텐트 33-11
HTTP 응답 메시지 생성 33-11
응답 헤더 채우기 33-11
응답 상태 표시 33-11
클라이언트 액션에 대한 요구 표시 33-12
서버 애플리케이션 설명 33-12
컨텐트 설명 33-12
응답 컨텐트 설정 33-12
응답 보내기 33-13
응답 메시지 컨텐트 생성 33-13
페이지 프로듀서 컴포넌트 사용 33-14
HTML 템플릿 33-14
HTML 템플릿 지정 33-15
HTML 투명 태그 변환 33-15
액션 항목에서 페이지 프로듀서 사용 . . 33-15
페이지 프로듀서의 연결 33-16

응답에서 데이터베이스 정보 사용 33-17
웹 모듈에 세션 추가 33-18
HTML에 데이터베이스 정보 표시 33-18
데이터셋 페이지 프로듀서 사용 33-18
테이블 프로듀서 사용 33-19
테이블 어트리뷰트(attribute) 지정 33-19
행 어트리뷰트 지정 33-19
열 지정 33-20
HTML 문서에 테이블 포함 33-20
데이터셋 테이블 프로듀서 설정 33-20
쿼리 테이블 프로듀서 설정 33-20

34장

WebSnap을 사용하여 웹 서버 애플리케이션 생성 34-1

기본 WebSnap 컴포넌트 34-2
웹 모듈 34-2
웹 애플리케이션 모듈 타입 34-3
웹 페이지 모듈 34-4
웹 데이터 모듈 34-4
어댑터 34-5
필드 34-5
액션 34-6
오류 34-6
레코드 34-6
페이지 프로듀서 34-6
WebSnap을 사용하여 웹 서버 애플리케이션 생성 34-7
서버 타입 선택 34-8
애플리케이션 모듈 컴포넌트 지정 34-8
웹 애플리케이션 모듈 옵션 선택 34-10
WebSnap 자습서 34-11
새 애플리케이션 생성 34-11
단계 1. WebSnap 애플리케이션 마법사 시작 34-11
단계 2. 생성된 파일 및 프로젝트 저장 . . 34-11
단계 3. 애플리케이션 제목 지정 34-12
CountryTable 페이지 생성 34-12
단계 1. 새 웹 페이지 모듈 추가 34-12
단계 2. 새 웹 페이지 모듈 저장 34-13
CountryTable 모듈에 데이터 컴포넌트 추가 34-13
단계 1. 데이터 인식 컴포넌트 추가 34-13
단계 2. 키 필드 지정 34-14
단계 3. 어댑터 컴포넌트 추가 34-14
데이터를 표시할 그리드 생성 34-15
단계 1. 그리드 추가 34-15
단계 2. 그리드에 편집 커맨드 추가 34-17
Edit 폼 추가 34-18
단계 1. 새 웹 페이지 모듈 추가 34-18
단계 2. 새 모듈 저장 34-19
단계 3. 새 모듈에서 CountryTableU를 액세스할 수 있게 만들기 34-19
단계 4. 입력 필드 추가 34-19
단계 5. 버튼 추가 34-20
단계 6. 그리드 페이지에 폼 액션 연결 . . 34-21
단계 7. 폼 페이지에 그리드 액션 연결 . . 34-21
완성된 애플리케이션 실행 34-22
오류 리포트 추가 34-22
단계 1. 그리드에 오류 지원 추가 34-22
단계 2. 폼에 오류 지원 추가 34-23
단계 3. 오류 리포트 메커니즘 테스트 . . 34-23
고급 HTML 디자인 34-24
HTML 파일에서 서버사이드 스크립트 조작 34-25
로그인 지원 34-25
로그인 지원 추가 34-25
세션 서비스 사용 34-26
로그인 페이지 34-27
로그인이 필요한 페이지 설정 34-29
사용자 액세스 권한 34-29
필드를 동적으로 에디트 박스 또는 텍스트로 표시 34-30
필드 및 컨텐트 숨기기 34-30
페이지 액세스 금지 34-30
WebSnap의 서버사이드 스크립트 34-31
액티브 스크립트 34-32
스크립트 엔진 34-32
스크립트 블록 34-32
스크립트 생성 34-32
마법사 템플릿 34-32
TAdapterPageProducer 34-33
스크립트 편집 및 보기 34-33
페이지에 스크립트 포함 34-33
스크립트 객체 34-33
요청 및 응답 디스패칭 34-34
디스패처 컴포넌트 34-34
어댑터 디스패처 작업 34-35
어댑터 컴포넌트를 사용하여 컨텐트 생성 34-35
어댑터 요청 받기 및 응답 생성 34-36
이미지 요청 34-38
이미지 응답 34-38
액션 항목 디스패칭 34-39
페이지 디스패처 작업 34-40

35장

XML 문서 작업 35-1
DOM(Document Object Model) 사용 35-2
XML 컴포넌트 작업 35-3
TXMLDocument 사용 35-3
XML 노드 작업 35-4
노드 값 작업 35-4
노드 어트리뷰트(attribute) 작업 35-5
자식 노드 추가 및 삭제 35-5
Data Binding 마법사로 XML 문서 고유화 35-5
XML Data Binding 마법사 사용 35-7
XML Data Binding 마법사가 생성하는 코드 사용 35-8

36장

웹 서비스 사용 36-1
인보커블(invocable) 인터페이스 이해 36-2
인보커블(invocable) 인터페이스에 넌스칼라 (nonscalar) 타입 사용 36-3
넌스칼라(nonscalar) 타입 등록 36-4
typedef로 선언된 타입 및 열거 타입 등록 36-6
리모터블(remotable) 객체 사용 36-7
리모터블 객체 예제 36-8
웹 서비스를 지원하는 서버 작성 36-9
웹 서비스 서버 생성 36-10
SOAP 애플리케이션 마법사 사용 36-11
새 웹 서비스 추가 36-12
생성된 코드 수정 36-12
다른 기본 클래스 사용 36-12
Web Services Importer 사용 36-13
웹 서비스를 위한 사용자 정의 예외 클래스 생성 36-14
웹 서비스 애플리케이션을 위한 WSDL 문서 생성 36-15
웹 서비스용 클라이언트 작성 36-16
WSDL 문서 임포트 36-16
인보커블(invocable) 인터페이스 호출 36-16

37장

소켓 작업 37-1
서비스 구현 37-1
서비스 프로토콜 이해 37-2
애플리케이션과 통신 37-2
서비스와 포트 37-2
소켓 연결의 타입 37-2
클라이언트 연결 37-3
수신 대기 연결 37-3
서버 연결 37-3
소켓 설명 37-3
호스트 설명 37-4
호스트 이름과 IP 주소 중에서 선택 37-4
포트 사용 37-5
소켓 컴포넌트 사용 37-5
연결에 대한 정보 얻기 37-6
클라이언트 소켓 사용 37-6
대상 서버 지정 37-6
연결 구성 37-6
연결에 대한 정보 얻기 37-6
연결 끊기 37-7
서버 소켓 사용 37-7
포트 지정 37-7
클라이언트 요청 수신 대기 37-7
클라이언트에 연결 37-7
서버 연결 끊기 37-7
소켓 이벤트에 응답 37-8
오류 이벤트 37-8
클라이언트 이벤트 37-8
서버 이벤트 37-9
수신 대기 시의 이벤트 37-9
클라이언트 연결 시의 이벤트 37-9
소켓 연결을 통한 읽기 및 쓰기 37-9
넌블로킹 연결 37-10
읽기 및 쓰기 이벤트 37-10
블로킹 연결 37-10

IV 부

COM 기반 애플리케이션 개발

38장

COM 기술 개요 38-1
스펙 및 구현에 대한 COM 38-1
COM 확장 38-2
COM 애플리케이션 요소 38-2
COM 인터페이스 38-3
기본 COM 인터페이스, IUnknown 38-4
COM 인터페이스 포인터 38-4
COM 서버 38-5
CoClasses 및 클래스 팩토리 38-6
in-process, out-of-process 및 원격 서버 . . 38-6
마샬링 메커니즘 38-8
집합체(aggregation) 38-9
COM 클라이언트 38-9

COM 확장 38-10
Automation 서버 38-12
Active Server Pages 38-13
ActiveX 컨트롤 38-13
Active Document 38-14
트랜잭션 객체 38-14
COM+ 이벤트 및 event subscriber object . . 38-15
타입 라이브러리 38-16
타입 라이브러리의 내용 38-16
타입 라이브러리 생성 38-17
타입 라이브러리 사용 시기 38-17
타입 라이브러리 액세스 38-17
타입 라이브러리 사용의 이점 38-18
타입 라이브러리 도구 사용 38-19
마법사로 COM 객체 구현 38-19
마법사에 의해 작성된 코드 38-22

39장

타입 라이브러리 사용 39-1

Type Library Editor 39-2
Type Library Editor의 구성 요소 39-3
툴바 39-3
객체 리스트 창 39-5
상태 표시줄 39-5
타입 정보 페이지 39-6
타입 라이브러리 요소 39-8
인터페이스 39-8
Dispinterface 39-9
CoClass 39-9
타입 정의 39-9
모듈 39-10
Type Library Editor 사용 39-10
유효한 타입 39-11
새 타입 라이브러리 생성 39-13
기존의 타입 라이브러리 열기 39-13
타입 라이브러리에 인터페이스 추가 . . . 39-14
타입 라이브러리를 사용하여 인터페이스 수정 39-14
인터페이스 또는 dispinterface에 속성 및 메소드 추가 39-14
타입 라이브러리에 CoClass 추가 39-15
CoClass에 인터페이스 추가 39-16
타입 라이브러리에 열거 추가 39-16
타입 라이브러리에 알리아스 추가 39-16
타입 라이브러리에 레코드 또는 합집합 추가 39-17
타입 라이브러리에 모듈 추가 39-17
타입 라이브러리 정보 저장 및 등록 39-17
타입 라이브러리 저장 39-18
타입 라이브러리 새로 고침 39-18
타입 라이브러리 등록 39-18
IDL 파일 엑스포트 39-19
타입 라이브러리 배포 39-19

40장

COM 클라이언트 생성 40-1

타입 라이브러리 정보 임포트 40-2
Import Type Library 다이얼로그 박스 사용 . 40-3
Import ActiveX 다이얼로그 박스 사용 40-4
타입 라이브러리 정보를 임포트할 때 생성되는 코드 40-5
임포트된 객체 제어 40-6
컴포넌트 랩퍼 사용 40-6
ActiveX 랩퍼(ActiveX wrapper) 40-7
Automation 객체 랩퍼 40-7
데이터 인식 ActiveX 컨트롤 사용 40-8
예제: Microsoft Word를 사용한 문서 인쇄 40-10
단계 1: 이 예제를 위해 C++Builder 준비 40-10
단계 2: Word 타입 라이브러리 임포트 . . 40-10
단계 3: Vtable 또는 dispatch 인터페이스 객체를 사용하여 Microsoft Word 제어 40-11
단계 4: 예제 삭제 40-12
타입 라이브러리 정의를 기반으로 한 클라이언트 코드 작성 40-13
서버에 연결 40-13
이중 인터페이스를 사용한 Automation 서버 제어 40-13
dispatch 인터페이스를 사용하여 Automation 서버 제어 40-13
Automation 컨트롤러에서 이벤트 처리 40-14
타입 라이브러리가 없는 서버를 위한 클라이언트 생성 40-16

41장

간단한 COM 서버 생성 41-1

COM 객체 생성 개요 41-1
COM 객체 디자인 41-2
COM Object 마법사 사용 41-2
Automation Object 마법사 사용 41-4
스레드 모델 선택 41-5
Free 스레드 모델을 지원하는 객체 작성 41-6

Apartment 스레드 모델을 지원하는 객체 작성 41-7
Neutral 스레드 모델을 지원하는 객체 작성 41-8
ATL 옵션 지정 41-8
COM 객체 인터페이스 정의 41-9
객체 인터페이스에 속성 추가 41-9
객체 인터페이스에 메소드 추가 41-10
클라이언트에 이벤트 제공 41-10
Automation 객체의 이벤트 관리 41-11
Automation 인터페이스 41-12
이중 인터페이스 41-12
dispatch 인터페이스 41-13
사용자 정의 인터페이스 41-14
데이터 마샬링 41-14
Automation과 호환 가능한 타입 41-14
자동 마샬링에 대한 타입 제한 41-15
사용자 정의 마샬링 41-15
COM 객체 등록 41-16
in-process 서버 등록 41-16
out-of-process 서버 등록 41-16
애플리케이션 테스트 및 디버깅 41-17

42장

Active Server Page 생성 42-1
Active Server Object 생성 42-2
ASP 내장 함수 사용 42-3
Application 객체 42-3
Request 42-4
Response 42-5
Session 42-5
Server 42-6
in-process 또는 out-of-process 서버를 위한 ASP 생성 42-7
Active Server Object 등록 42-7
in-process 서버 등록 42-7
out-of-process 서버 등록 42-8
Active Server Page 애플리케이션 테스트 및 디버깅 42-8

43장

ActiveX 컨트롤 생성 43-1
ActiveX 컨트롤 생성 개요 43-2
ActiveX 컨트롤의 요소 43-2
VCL 컨트롤 43-3
ActiveX 랩퍼(ActiveX wrapper) 43-3
타입 라이브러리 43-3
속성 페이지 43-3
ActiveX 컨트롤 디자인 43-4
VCL 컨트롤에서 ActiveX 컨트롤 생성 43-4
VCL 폼을 기반으로 ActiveX 컨트롤 생성 43-6
ActiveX 컨트롤 라이센스 43-7
ActiveX 컨트롤 인터페이스 사용자 정의 43-8
속성, 메소드 및 이벤트 추가 43-9
속성 및 메소드 추가 43-9
이벤트 추가 43-10
타입 라이브러리를 사용하여 간단한 데이터 연결 사용 43-11
ActiveX 컨트롤을 위한 속성 페이지 생성 43-13
새 속성 페이지 생성 43-13
속성 페이지에 컨트롤 추가 43-14
속성 페이지 컨트롤을 ActiveX 컨트롤 속성에 연결 43-14
속성 페이지 업데이트 43-14
객체 업데이트 43-15
속성 페이지를 ActiveX 컨트롤에 연결 . . . 43-15
ActiveX 컨트롤 등록 43-15
ActiveX 컨트롤 테스트 43-16
ActiveX 컨트롤을 웹에 배포 43-16
옵션 설정 43-17

44장

MTS 객체 또는 COM+ 객체 생성 44-1
트랜잭션 객체 이해 44-2
트랜잭션 객체의 요구 사항 44-3
리소스 관리 44-3
객체 컨텍스트 액세스 44-4
Just-in-time 활성화 44-4
리소스 풀링 44-5
데이터베이스 리소스 디스펜서 44-6
Shared Property Manager 44-6
리소스 릴리스 44-9
객체 풀링 44-9
MTS 및 COM+ 트랜잭션 지원 44-10
트랜잭션 어트리뷰트(attribute) 44-11
트랜잭션 어트리뷰트 설정 44-11
stateful 객체와 stateless 객체 44-12
트랜잭션 종료 방법 변경 44-12
트랜잭션 초기화 44-13
클라이언트에 트랜잭션 객체 설정 44-14
서버에 트랜잭션 객체 설정 44-16
트랜잭션 제한 시간 44-17
역할 기반 보안 44-17
트랜잭션 객체 작성 개요 44-18
Transactional Object 마법사 사용 44-18
트랜잭션 객체에 대한 스레드 모델 선택 . . 44-19
작업 44-20

COM+에서 이벤트 생성 44-21
Event Object 마법사 사용 44-23
COM+ Event Subscription Object 마법사 . . 44-24
COM+ 이벤트 객체를 사용하여 이벤트 발생 44-25
객체 참조 전달 44-25
SafeRef 메소드 사용 44-26
콜백 44-26
트랜잭션 객체 디버깅 및 테스트 44-27
트랜잭션 객체 설치 44-27
트랜잭션 객체 관리 44-28

V 부
사용자 정의 컴포넌트 생성

45장
컴포넌트 생성 개요 45-1
클래스 라이브러리 45-1
컴포넌트 및 클래스 45-2
컴포넌트 생성 방법 45-2
기존 컨트롤 수정 45-3
윈도우 컨트롤 생성 45-3
그래픽 컨트롤 생성 45-4
Windows 컨트롤 서브클래싱 45-4
넌비주얼(nonvisual) 컴포넌트 생성 45-4
컴포넌트 생성 시 고려 사항 45-5
종속성 제거 45-5
속성, 메소드 및 이벤트 설정 45-5
속성 45-6
이벤트 45-6
메소드 45-6
그래픽 캡슐화 45-7
컴포넌트 등록 45-8
새 컴포넌트 생성 45-8
컴포넌트 마법사로 컴포넌트 생성 45-9
수동으로 컴포넌트 생성 45-11
유닛 파일 생성 45-11
컴포넌트 파생 45-12
새 생성자 선언 45-13
컴포넌트 등록 45-13
컴포넌트용 비트맵 생성 45-14
설치되지 않은 컴포넌트 테스트 45-16
설치된 컴포넌트 테스트 45-18
컴포넌트 팔레트에 컴포넌트 설치 45-18
소스 파일을 사용할 수 있게 만들기 45-19
컴포넌트 추가 45-19

46장
컴포넌트 작성자를 위한 객체 지향 프로그래밍 46-1
새 클래스 정의 46-1
새 클래스 파생 46-2
반복을 피하기 위한 클래스 기본값 변경 46-2
클래스에 새 기능 추가 46-2
새 컴포넌트 클래스 선언 46-3
조상, 자손 및 클래스 계층 구조 46-3
액세스 제어 46-4
구현 세부 사항 숨기기 46-4
컴포넌트 작성자의 인터페이스 정의 46-6
런타임 인터페이스 정의 46-7
디자인 타임 인터페이스 정의 46-7
메소드 디스패칭 46-8
일반적인 메소드 46-8
가상 메소드 46-9
메소드 오버라이드 46-9
추상 클래스 멤버 46-10
클래스 및 포인터 46-10

47장
속성 생성 47-1
속성을 만드는 이유 47-1
속성 타입 47-2
상속된 속성 게시 47-2
속성 정의 47-3
속성 선언 47-3
내부 데이터 저장소 47-4
직접 액세스 47-4
액세스 메소드 47-5
read 메소드 47-6
write 메소드 47-6
디폴트 속성 값 47-7
기본값을 갖지 않도록 지정 47-7
배열 속성 생성 47-8
하위 컴포넌트의 속성 생성 47-9
속성 저장 및 로드 47-10
저장 및 로드 메커니즘 사용 47-11
기본값 지정 47-11
저장할 대상 결정 47-12
로드 후 초기화 47-13
published가 아닌 속성 저장 및 로드 47-13
속성 값을 저장 및 로드하는 메소드 생성 47-13
DefineProperties 메소드 오버라이드 . . 47-14

48장

이벤트 생성 48-1

이벤트 정의 48-1
이벤트는 클로저(closure) 48-2
이벤트는 속성 48-2
이벤트 타입은 클로저(closure) 타입 48-3
이벤트 핸들러의 반환 타입은 void 48-3
이벤트 핸들러는 옵션 48-3
표준 이벤트 구현 48-4
표준 이벤트 식별 48-4
모든 컨트롤에 대한 표준 이벤트 48-4
표준 컨트롤에 대한 표준 이벤트 48-5
이벤트 표시 48-5
표준 이벤트 처리 변경 48-5
새로운 이벤트 정의 48-6
이벤트 트리거 48-6
두 종류의 이벤트 48-7
핸들러 타입 정의 48-7
일반적인 통지 48-7
이벤트별 핸들러 48-7
핸들러의 정보 반환 48-7
이벤트 선언 48-8
"On"으로 시작하는 이벤트 이름 48-8
이벤트 호출 48-8

49장

메소드 생성 49-1

종속성 피하기 49-1
메소드 이름 지정 49-2
메소드 보호 49-3
public이어야 하는 메소드 49-3
protected여야 하는 메소드 49-3
가상 메소드 만들기 49-3
메소드 선언 49-4

50장

컴포넌트에서 그래픽 사용 50-1

그래픽 개요 50-1
캔버스 사용 50-3
그림 작업 50-3
그림, 그래픽 또는 캔버스 사용 50-3
그래픽 로드 및 저장 50-4
팔레트 처리 50-4
컨트롤의 팔레트 지정 50-5
오프스크린 비트맵 50-5
오프스크린 비트맵 생성 및 관리 50-6
비트맵 이미지 복사 50-6
변경에 대한 응답 50-6

51장

메시지 및 시스템 통지 처리 51-1

메시지 처리 시스템 이해 51-1
Windows 메시지의 내용 51-2
메시지 디스패칭 51-3
메시지 흐름 추적 51-3
메시지 처리 변경 51-4
핸들러 메소드 오버라이드 51-4
메시지 매개변수 사용 51-5
메시지 트래핑 51-5
새 메시지 핸들러 생성 51-6
새로운 메시지 정의 51-6
메시지 식별자 선언 51-6
메시지 구조체 타입 선언 51-6
새 메시지 처리 메소드 선언 51-7
메시지 보내기 51-8
메시지를 폼에 있는 모든 컨트롤에 브로드캐스트 51-8
컨트롤의 메시지 핸들러 직접 호출 51-9
Windows 메시지 대기열을 사용하여 메시지 보내기 51-9
곧바로 실행되지 않는 메시지 보내기 . . 51-10
CLX를 사용하여 시스템 통지에 응답 51-10
신호에 응답 51-10
사용자 정의 신호 핸들러 할당 51-11
시스템 이벤트에 응답 51-12
일반적으로 사용하는 이벤트 51-13
EventFilter 메소드 오버라이드 51-14
Qt 이벤트 생성 51-15

52장

디자인 타임 시 컴포넌트 사용 52-1

컴포넌트 등록 52-1
Register 함수 선언 52-2
Register 함수 작성 52-2
컴포넌트 지정 52-2
팔레트 페이지 지정 52-3
RegisterComponents 함수 사용 52-3
팔레트 비트맵 추가 52-4
컴포넌트를 위한 도움말 제공 52-4
도움말 파일 생성 52-5
항목 작성 52-5
상황에 맞는 컴포넌트 도움말 작성 52-6
컴포넌트 도움말 파일 추가 52-7
속성 에디터 추가 52-7
속성 에디터 클래스 파생 52-7
텍스트로 속성 편집 52-8
속성 값 표시 52-9

속성 값 설정 52-9
전체 속성 편집 52-9
에디터 어트리뷰트(attribute) 지정 52-10
속성 에디터 등록 52-11
속성 범주 52-12
한 번에 하나의 속성 등록 52-12
한 번에 여러 속성 등록 52-13
속성 범주 지정 52-14
IsPropertyInCategory 함수 사용 52-15
컴포넌트 에디터 추가 52-15
컨텍스트 메뉴에 항목 추가 52-16
메뉴 항목 지정 52-16
명령 구현 52-16
더블 클릭 동작 변경 52-17
클립보드 형식 추가 52-18
컴포넌트 에디터 등록 52-18
컴포넌트를 패키지로 컴파일 52-19
사용자 정의 컴포넌트의 문제 해결 52-19

53장
기존 컴포넌트 수정 53-1
컴포넌트 생성 및 등록 53-1
컴포넌트 클래스 수정 53-3
생성자 오버라이드 53-3
새 디폴트 속성 값 지정 53-4

54장
그래픽 컴포넌트 생성 54-1
컴포넌트 생성 및 등록 54-1
상속된 속성 게시 54-3
그래픽 기능 추가 54-3
그릴 대상 결정 54-3
속성 타입 선언 54-4
속성 선언 54-4
구현 메소드 작성 54-5
생성자 및 소멸자 오버라이드 54-5
디폴트 속성 값 변경 54-5
펜과 브러시 게시 54-6
데이터 멤버 선언 54-6
액세스 속성 선언 54-6
소유된 클래스 초기화 54-7
소유된 클래스의 속성 설정 54-8
컴포넌트 이미지 그리기 54-9
정교한 도형 그리기 54-10

55장
그리드 사용자 정의 55-1
컴포넌트 생성 및 등록 55-1
상속된 속성 게시 55-3
초기 값 변경 55-3
셀 크기 조정 55-4
셀 채우기 55-6
날짜 추적 55-6
내부 날짜 저장 55-7
년, 월, 일 액세스 55-7
일(day) 수 생성 55-9
현재 날짜 선택 55-11
연도와 월 탐색 55-12
일 탐색 55-12
선택 항목 이동 55-13
OnChange 이벤트 제공 55-13
비어 있는 셀 제외 55-14

56장
데이터 인식 컨트롤 만들기 56-1
데이터 찾아보기 컨트롤 생성 56-1
컴포넌트 생성 및 등록 56-2
읽기 전용 컨트롤 만들기 56-3
ReadOnly 속성 추가 56-3
필요한 업데이트 허용 56-4
데이터 연결 추가 56-5
데이터 멤버 선언 56-5
액세스 속성 선언 56-6
액세스 속성 선언의 예제 56-6
데이터 연결 초기화 56-7
데이터 변경 내용에 응답 56-7
데이터 편집 컨트롤 생성 56-9
FReadOnly의 기본값 변경 56-9
마우스 다운(mouse-down) 및
키 다운(key-down) 이벤트 처리 56-9
마우스 다운(mouse-down) 이벤트에
응답 56-10
키 다운(key-down) 이벤트에 응답 56-10
필드 데이터 연결 클래스 업데이트 56-12
Change 메소드 수정 56-12
데이터셋 업데이트 56-13

57장
다이얼로그 박스를 컴포넌트로 만들기 57-1
컴포넌트 인터페이스 정의 57-1
컴포넌트 생성 및 등록 57-2
컴포넌트 인터페이스 생성 57-3
폼 유닛 파일 포함 57-3
인터페이스 속성 추가 57-4
Execute 메소드 추가 57-5
컴포넌트 파생 57-6

58장
IDE 확장 58-1
Tools API의 개요 58-1
마법사 클래스 작성 58-3
마법사 인터페이스 구현 58-4
인터페이스 구현 단순화 58-6
마법사 패키지 설치 58-6
Tools API 서비스 사용 58-7
원시 IDE 객체 사용 58-8
INTAServices 인터페이스 사용 58-8
이미지 리스트에 이미지 추가 58-8
액션 리스트에 작업 추가 58-9
툴바 버튼 삭제 58-9
마법사 디버깅 58-10
인터페이스 버전 번호 58-11
파일 및 에디터 사용 58-11
모듈 인터페이스 사용 58-12
에디터 인터페이스 사용 58-12
폼 및 프로젝트 생성 58-13
모듈 생성 58-13
마법사에게 IDE 이벤트 통지 58-17
마법사 DLL 설치 58-21
런타임 패키지 없이 DLL 사용 58-22

부록 A
ANSI 구현 특정 표준 A-1

부록 B
WebSnap 서버사이드 스크립트 참조 B-1
객체 타입 B-1
Adapter 타입 B-2
속성 B-2
AdapterAction 타입 B-4
속성 B-4
메소드 B-5
AdapterErrors 타입 B-6
속성 B-6
AdapterField 타입 B-6
속성 B-6
메소드 B-9
AdapterFieldValues 타입 B-9
속성 B-10
메소드 B-10
AdapterFieldValuesList 타입 B-10
속성 B-10
메소드 B-11
AdapterHiddenFields 타입 B-11
속성 B-11
메소드 B-11
AdapterImage 타입 B-11
속성 B-11
Module 타입 B-12
속성 B-12
Page 타입 B-12
속성 B-12
전역 객체 B-13
Application 객체 B-14
속성 B-14
메소드 B-15
EndUser 객체 B-15
속성 B-15
Modules 객체 B-16
Page 객체 B-16
Pages 객체 B-16
Producer 객체 B-16
속성 B-16
메소드 B-17
Request 객체 B-17
속성 B-17
Response 객체 B-17
속성 B-17
메소드 B-18
Session 객체 B-18
속성 B-18
JScript 예제 B-18
예제 1 . . . . . B-19
예제 2 . . . . . B-20
예제 3 . . . . . B-20
예제 4 . . . . . B-20
예제 5 . . . . . B-21
예제 6 . . . . . B-21
예제 7 . . . . . B-22
예제 8 . . . . . B-22
예제 9 . . . . . B-23
예제 10 . . . . . B-24
예제 11 . . . . . B-26
예제 12 . . . . . B-27
예제 13 . . . . . B-28
예제 14 . . . . . B-29
예제 15 . . . . . B-31
예제 16 . . . . . B-32
예제 17 . . . . . B-33
예제 18 . . . . . B-35
예제 19 . . . . . B-35
예제 20 . . . . . B-35
예제 21 . . . . . B-36
예제 22 . . . . . B-37

표

1.1 글꼴과 기호 1-3
3.1 중요한 기본 클래스 3-5
4.1 개방형 모드 4-6
4.2 공유 모드 4-6
4.3 각 개방형 모드에서 사용할 수 있는 공유 모드 4-6
4.4 어트리뷰트 상수 및 값 4-8
4.5 리스트 관리용 클래스 4-13
4.6 문자열 비교 루틴 4-22
4.7 대소문자 변환 루틴 4-22
4.8 문자열 수정 루틴 4-23
4.9 하위 문자열 루틴 4-23
4.10 Null 종료 문자열 비교 루틴 4-24
4.11 Null 종료 문자열 대소문자 변환 루틴 4-24
4.12 문자열 수정 루틴 4-24
4.13 하위 문자열 루틴 4-24
4.14 문자열 복사 루틴 4-25
5.1 컴포넌트 팔레트 페이지 5-6
6.1 선택한 텍스트의 속성 6-8
6.2 Fixed 및 Variable owner-draw 스타일 . . .6-12
7.1 라이브러리에 대한 컴파일러 지시어 7-10
7.2 컴포넌트 팔레트의 데이터베이스 페이지 7-15
7.3 웹 서버 애플리케이션 7-17
7.4 데이터 모듈의 컨텍스트 메뉴 옵션 7-21
7.5 Tapplication의 도움말 메소드 7-33
8.1 액션 설정 용어 8-17
8.2 Action Manager의 PrioritySchedule 속성의 기본값 8-22
8.3 액션 클래스 8-27
8.4 예제 캡션 및 파생된 이름 8-31
8.5 메뉴 디자이너 컨텍스트 메뉴 명령 8-36
8.6 스피드 버튼의 모양 설정 8-44
8.7 툴 버튼 모양 설정 8-46
8.8 쿨(cool) 버튼 모양 설정 8-47
9.1 편집 컨트롤 속성 9-2
10.1 그래픽 객체 타입 10-3
10.2 캔버스 객체의 일반적인 속성 10-4
10.3 캔버스 객체의 일반 메소드 10-4
10.4 CLX MIME 타입과 상수 10-22
10.5 마우스 이벤트 매개변수 10-24
10.6 멀티미디어 장치 타입과 기능 10-32
11.1 스레드 우선 순위 11-3
11.2 WaitFor 반환 값 11-10
12.1 예외 처리 컴파일러 옵션 12-14
12.2 선택된 예외 클래스 12-17
13.1 객체 모델 비교 13-10
13.2 BOOL 변수의 동등 비교 !A == !B 13-20
13.3 오브젝트 파스칼에서 C++로의 RTTI 매핑 예제 13-22
14.1 이식 기법 14-2
14.2 CLX 부분 14-5
14.3 변경되었거나 다른 기능 14-8
14.4 VCL 유닛과 이에 해당하는 CLX 유닛 . . .14-9
14.5 CLX 전용 유닛 14-9
14.6 VCL 전용 유닛 14-10
14.7 Linux 및 Windows 운영 환경 간의 차이점 14-13
14.8 공통된 Linux 디렉토리 14-15
14.9 유사한 데이터 액세스 컴포넌트 14-21
15.1 패키지 파일 15-2
15.2 패키지별 컴파일러 지시어 15-11
15.3 패키지별 명령줄 링커 스위치 15-12
15.4 패키지와 함께 배포되는 파일 15-13
16.1 BiDi를 지원하는 VCL 객체 16-4
16.2 예상 문자열 길이 16-9
17.1 애플리케이션 파일 17-3
17.2 병합 모듈 및 종속 관계 17-4
17.3 독립 실행형 실행 파일로 dbExpress 배포 . 17-7
17.4 드라이버 DLL과 dbExpress 배포 17-8
17.5 SQL 데이터베이스 클라이언트 소프트웨어 파일 17-9
19.1 데이터 컨트롤 19-2
19.2 열 속성 19-20
19.3 확장된 TColumn Title 속성 19-20
19.4 복합 필드의 표시 방법에 영향을 미치는 속성 19-23
19.5 Expanded TDBGrid Options 속성 19-24
19.6 그리드 컨트롤 이벤트 19-26
19.7 선택한 데이터베이스 컨트롤 그리드 속성 19-28
19.8 TDBNavigator 버튼 19-29
21.1 데이터베이스 연결 컴포넌트 21-1
22.1 데이터셋 State 속성의 값 22-3
22.2 데이터셋의 탐색 메소드 22-5
22.3 데이터셋의 탐색 속성 22-5
22.4 필터에 표시될 수 있는 비교 및 논리 연산자 22-14

22.5 FilterOptions 값 22-15
22.6 필터링된 데이터셋 탐색 메소드 22-16
22.7 데이터 삽입, 업데이트 및 삭제를 위한 데이터셋 메소드 22-17
22.8 전체 레코드를 사용하는 메소드 22-21
22.9 인덱스 기반 검색 메소드 22-27
23.1 데이터 표시에 영향을 주는 TFloatField 속성 23-1
23.2 영구적 특수 필드 종류 23-6
23.3 필드 컴포넌트 속성 23-11
23.4 필드 컴포넌트 서식 루틴 23-15
23.5 필드 컴포넌트 이벤트 23-15
23.6 선택된 필드 컴포넌트 메소드 23-16
23.7 특수한 변환 결과 23-19
23.8 객체 필드 컴포넌트의 타입 23-22
23.9 공통된 객체 필드 자손 속성 23-22
24.1 파일 확장자에 따라 BDE가 인식하는 테이블 타입 24-5
24.2 테이블 타입 값 24-5
24.3 BatchMove 임포트 모드 24-8
24.4 세션 컴포넌트의 데이터베이스 관련 정보 메소드 24-26
24.5 TSessionList 속성 및 메소드 24-28
24.6 캐싱된 업데이트를 위한 속성, 메소드 및 이벤트 24-32
24.7 UpdateKind 값 24-38
24.8 일괄 이동 모드 24-48
24.9 Data Dictionary 인터페이스 24-52
25.1 ADO 컴포넌트 25-2
25.2 ADO 연결 모드 25-6
25.3 ADO 데이터셋의 실행 옵션 25-11
25.4 ADO 및 클라이언트 데이터셋에 캐싱된 업데이트 비교 25-12
26.1 테이블을 나열하는 메타데이터 테이블의 열 26-13
26.2 내장 프로시저를 나열하는 메타데이터 테이블의 열 26-14
26.3 필드를 나열하는 메타데이터 테이블의 열 26-14
26.4 인덱스를 나열하는 메타데이터 테이블의 열 26-15
26.5 매개변수를 나열하는 메타데이터 테이블의 열 26-16
27.1 클라이언트 데이터셋의 필터 지원 27-3
27.2 유지 보수된 집계의 요약 연산자 27-12
27.3 업데이트 캐싱을 위한 특화된 클라이언트 데이터셋 27-17
28.1 AppServer 인터페이스 멤버 28-3
28.2 프로바이더 옵션 28-5
28.3 UpdateStatus 값 28-9
28.4 UpdateMode 값 28-10
28.5 ProviderFlags 값 28-10
29.1 멀티 티어 애플리케이션에서 사용하는 컴포넌트 29-3
29.2 연결 컴포넌트 29-5
29.3 Javascript 라이브러리 29-33
32.1 Web Broker와 WebSnap 비교 32-2
33.1 MethodType 값 33-6
34.1 웹 애플리케이션 모듈 타입 34-3
34.2 웹 서버 애플리케이션 타입 34-8
34.3 웹 애플리케이션 컴포넌트 34-9
34.4 스크립트 객체 34-33
34.5 액션 요청에 있는 요청 정보 34-36
36.1 리모터블 클래스 36-7
38.1 COM 객체 요구 사항 38-12
38.2 COM, Automation 및 ActiveX 객체 구현을 위한 C++Builder 마법사 38-21
39.1 Type Library Editor 파일 39-2
39.2 Type Library Editor의 구성 요소 39-3
39.3 타입 라이브러리 페이지 39-6
41.1 COM 객체에 대한 스레드 모델 41-5
42.1 IApplicationObject 인터페이스 멤버 42-4
42.2 IRequest 인터페이스 멤버 42-4
42.3 IResponse 인터페이스 멤버 42-5
42.4 ISessionObject 인터페이스 멤버 42-6
42.5 IServer 인터페이스 멤버 42-6
44.1 트랜잭션 지원을 위한 IObjectContext 메소드 44-13
44.2 트랜잭션 객체의 스레드 모델 44-19
44.3 호출 동기화 옵션 44-21
44.4 이벤트 퍼블리셔 반환 코드 44-25
45.1 컴포넌트 생성 방법 45-3
46.1 객체 내의 가시성 레벨 46-4
47.1 Object Inspector에 속성이 표시되는 방법 47-2
50.1 캔버스 기능 요약 50-3
50.2 이미지 복사 메소드 50-6
51.1 시스템 통지에 응답하기 위한 TWidgetControl protected 메소드 51-13
52.1 미리 정의된 속성 에디터 타입 52-8
52.2 속성 값을 읽고 쓰기 위한 메소드 52-9
52.3 속성 에디터 어트리뷰트 플래그 52-10
52.4 속성 범주 52-14
58.1 네 종류의 마법사 58-3

58.2 Tools API 서비스 인터페이스 58-7
58.3 통지자 인터페이스 58-17
A.1 ANSI 규정 준수에 필요한 옵션 A-1
A.2 C++에서의 진단 식별 A-3
58.4 WebSnap 객체 타입 B-2
58.5 WebSnap 전역 객체 B-14
58.6 서버사이드 스크립트의 JScript 예제 B-18

그림

3.1 객체, 컴포넌트 및 컨트롤 3-4
3.2 단순화된 계층 다이어그램 3-4
8.1 데이터 인식 컨트롤과 데이터 소스 컴포넌트가 있는 프레임 8-14
8.2 Action Manager Editor 8-20
8.3 메뉴 용어 8-29
8.4 MainMenu 컴포넌트 및 PopupMenu 컴포넌트 8-30
8.5 메인 메뉴의 메뉴 디자이너 8-30
8.6 메인 메뉴에 메뉴 항목 추가 8-32
8.7 중첩 메뉴 구조 8-34
8.8 Select Menu 다이얼로그 박스 8-37
8.9 메뉴를 위한 예제 Insert Template 다이얼로그 박스 8-38
8.10 메뉴에 대한 Save Template 다이얼로그 박스 8-39
9.1 트랙 표시줄 컴포넌트의 세 가지 뷰 9-5
9.2 진행 표시줄 9-15
10.1 BMPDlg 유닛의 Bitmap Dimensions 다이얼로그 박스 10-21
13.1 VCL 스타일 객체 생성 순서 13-9
16.1 bdLeftToRight로 설정된 TListBox 16-6
16.2 bdRightToLeft로 설정된 TListBox 16-6
16.3 bdRightToLeftNoAlign으로 설정된 TListBox 16-7
16.4 bdRightToLeftReadingOnly로 설정된 TListBox 16-7
18.1 일반적인 데이터베이스 아키텍처 18-6
18.2 데이터베이스 서버에 직접 연결 18-8
18.3 파일 기반의 데이터베이스 애플리케이션 .18-9
18.4 클라이언트 데이터셋과 다른 데이터셋을 결합하는 아키텍처 18-12
18.5 멀티 티어 데이터베이스 아키텍처 18-13
19.1 TDBGrid 컨트롤 19-15
19.2 ObjectView가 false로 설정되어 있는 TDBGrid 컨트롤 19-23
19.3 Expanded가 false로 설정된 TDBGrid 컨트롤 19-23
19.4 Expanded가 true로 설정된 TDBGrid 컨트롤 19-23
19.5 디자인 타임 시 TDBCtrlGrid 19-27
19.6 TDBNavigator 컨트롤의 버튼 19-28
20.1 디자인 타임 시 decision 지원 컴포넌트 . . .20-2
20.2 1차원 크로스탭 20-3
20.3 3차원 크로스탭 20-3
20.4 다른 decision source에 연결된 decision graph 20-14
24.1 BDE 기반 애플리케이션의 컴포넌트 24-2
29.1 웹 기반 멀티 티어 데이터베이스 애플리케이션 29-29
31.1 CORBA 애플리케이션 구조 31-2
32.1 URL(Uniform Resource Locator)의 각 부분 32-3
33.1 서버 애플리케이션의 구조 33-4
34.1 New WebSnap 애플리케이션 다이얼로그 박스 34-8
34.2 Web App Components 다이얼로그 박스 34-9
34.3 CountryTable 페이지의 New WebSnap Page Module 다이얼로그 박스 34-13
34.4 CountryTable 웹 페이지 모듈 34-15
34.5 CountryTable Preview 탭 34-16
34.6 CountryTable HTML Script 탭 34-16
34.7 편집 커맨드를 추가한 이후의 CountryTable 미리 보기 34-18
34.8 CountryForm 미리 보기 34-20
34.9 폼 전송 버튼이 있는 CountryForm 34-21
34.10 로그인 지원 옵션이 선택되어 있는 Web App Components 다이얼로그 박스 34-26
34.11 웹 페이지 에디터에 표시되는 로그인 페이지의 예제 34-28
34.12 컨텐트 흐름 생성 34-36
34.13 액션 요청 및 응답 34-38
34.14 이미지 요청에 대한 응답 34-39
34.15 페이지 디스패칭 34-40
38.1 COM 인터페이스 38-3
38.2 인터페이스 vtable 38-4
38.3 In-process 서버 38-7
38.4 Out-of-process 및 원격 서버 38-8
38.5 COM 기반 기술 38-11
38.6 간단한 COM 객체 인터페이스 38-20
38.7 Automation 객체 인터페이스 38-20
38.8 ActiveX 객체 인터페이스 38-20
39.1 Type Library Editor 39-3
39.2 객체 리스트 창 39-5
41.1 이중 인터페이스 VTable 41-13
43.1 디자인 모드의 Mask Edit 속성 페이지 . . 43-14

44.1 COM+ Event 시스템 44-23
45.1 비주얼 컴포넌트 라이브러리 클래스 계층 45-2
45.2 Component 마법사 45-9
51.1 신호 라우팅 51-11
51.2 시스템 이벤트 라우팅 51-12

장

1

서문

*개발자 안내서*에서는 클라이언트 / 서버 데이터베이스 애플리케이션 작성, 사용자 정의 컴포넌트 작성, 인터넷 웹 서버 애플리케이션 작성, SOAP, TCP/IP, COM+ 및 ActiveX와 같은 업계 표준 규정 지원 등의 중급 및 고급 개발 주제에 대해 설명합니다. 웹 개발, 고급 XML 기술 및 데이터베이스 개발을 지원하는 대부분의 고급 기능에는 일부 C++Builder 버전에서 사용할 수 없는 컴포넌트 또는 마법사가 필요합니다.

*개발자 안내서*에서는 독자가 C++Builder 사용에 익숙하고 기본적인 C++Builder 프로그래밍 기법을 이해하고 있다고 가정합니다. C++Builder 프로그래밍과 통합 개발 환경(IDE)에 대한 소개는 *입문서*와 온라인 도움말을 참조하십시오.

안내서 내용

이 안내서에는 다음과 같은 다섯 부분이 포함되어 있습니다.

- **I부, "C++Builder 프로그래밍"**에서는 범용 C++Builder 애플리케이션을 생성하는 방법에 대해 설명합니다. 이 부분에서는 C++Builder 애플리케이션에서 사용할 수 있는 프로그래밍 기법에 대한 세부적인 내용을 제공합니다. 예를 들어, 일반적인 비주얼 컴포넌트 라이브러리(VCL) 또는 크로스 플랫폼 컴포넌트 라이브러리(CLX) 객체를 사용하여 문자열 처리, 텍스트 처리, 일반적인 다이얼로그 박스 구현, 그래픽 사용, 오류 및 예외 처리, DLL 사용, OLE Automation, 국제적인 애플리케이션 작성 등과 같은 인터페이스 프로그래밍을 쉽게 할 수 있는 방법에 대해 설명합니다.

  일반적으로 C++Builder의 원본으로 사용하는 VCL을 오브젝트 파스칼에서 작성하는 것은 문제가 되지 않습니다. 그러나 이로 인해 C++Builder 프로그램이 영향을 받는 경우가 가끔 있습니다. C++ 랭귀지 지원 및 VCL에 관한 장에서는 VCL 클래스를 사용할 때 C++ 클래스 인스턴스화의 차이점, 프로그래밍의 C++Builder "컴포넌트-속성-이벤트" 모델을 지원하기 위해 추가된 C++ 랭귀지 확장 등과 같은 랭귀지 문제에 대해 설명합니다.

또 다른 장에서는 Borland 크로스 플랫폼 컴포넌트 라이브러리(CLX)에서 객체를 사용하여 Windows 또는 Linux 플랫폼에서 컴파일 및 실행할 수 있는 애플리케이션을 개발하는 방법에 대해 설명합니다.

배포에 대한 장에서는 애플리케이션 사용자에게 애플리케이션을 배포하는 것과 관련된 작업에 대해 설명합니다. 예를 들어, 효과적인 컴파일러 옵션, InstallShield Express 사용, 라이센스 문제, 애플리케이션 정식 버전을 생성할 때 사용할 패키지, DLL 및 기타 라이브러리 결정 등에 대한 정보가 포함되어 있습니다.

- **II부, "데이터베이스 애플리케이션 개발"**에서는 데이터베이스 도구와 컴포넌트를 사용하여 데이터베이스 애플리케이션을 생성하는 방법에 대해 설명합니다. C++Builder에서는 Paradox 및 dBASE와 같은 로컬 데이터베이스와 InterBase, Oracle 및 Sybase와 같은 네트워크 SQL 서버 데이터베이스를 비롯한 여러 가지 타입의 데이터베이스를 액세스할 수 있습니다. dbExpress, Borland Database Engine, InterBaseExpress, ADO(ActiveX Data Object) 등의 다양한 데이터 액세스 메커니즘을 선택할 수 있습니다. 고급 데이터베이스 애플리케이션을 구현하려면 일부 에디션에서만 사용할 수 있는 C++Builder 기능이 필요합니다.
- **III부, "인터넷 애플리케이션 작성"**에서는 인터넷 상에서 배포되는 애플리케이션을 만드는 방법에 대해 설명합니다. C++Builder에는 웹 서버 애플리케이션을 작성하기 위한 일련의 도구가 포함되어 있습니다. 크로스 플랫폼 서버 애플리케이션을 만들 수 있는 아키텍처인 Web Broker, GUI 환경에서 웹 페이지를 디자인할 수 있는 WebSnap, XML 문서 작업 지원, SOAP 기반 웹 서비스를 사용하기 위한 아키텍처인 BizSnap 등을 예로 들 수 있습니다.

  또한 이 부분에서는 C++Builder 소켓 컴포넌트에 대한 장이 제공됩니다. 이러한 소켓 컴포넌트를 사용하면 TCP/IP 및 관련 프로토콜을 통해 다른 시스템과 통신할 수 있는 애플리케이션을 만들 수 있습니다. 소켓은 TCP/IP 프로토콜에 기반하여 연결되지만 XNS(Xerox Network System), Digital의 DECnet 또는 Novell의 IPX/SPX 제품군과 같은 관련 프로토콜에서 일반적으로 작동합니다.
- **IV부, "COM 기반 애플리케이션 개발"**에서는 다른 COM 기반 API 객체와 상호 운용할 수 있는 애플리케이션을 생성하는 방법에 대해 설명합니다. C++Builder는 ATL(Active Template Library) 마법사 및 Type Library Editor에 기초하여 COM 서버 개발을 쉽게 하는 COM 애플리케이션과 클라이언트 애플리케이션을 신속하게 작성할 수 있는 임포트 도구를 지원합니다. COM 클라이언트에 대한 지원은 모든 C++Builder 에디션에서 사용할 수 있지만, COM 서버에 대한 지원은 일부 C++Builder 에디션에서만 사용할 수 있습니다.
- **V부, "사용자 정의 컴포넌트 생성"**에서는 고유한 컴포넌트를 디자인 및 구현하는 방법과 이러한 컴포넌트를 IDE의 컴포넌트 팔레트에서 사용할 수 있게 만드는 방법에 대해 설명합니다. 컴포넌트는 디자인 타임에 처리하려는 거의 모든 프로그램 요소를 나타낼 수 있습니다. 사용자 정의 컴포넌트의 구현은 VCL 또는 CLX 클래스 라이브러리의 기존 클래스 타입에서 새 클래스를 파생함을 의미합니다.

설명서 규칙

이 설명서는 특별한 텍스트를 표시하기 위하여 표 1.1에 설명된 글꼴과 기호를 사용합니다.

표 1.1 글꼴과 기호

글꼴 또는 기호	의미
`Monospace type`	고정 폭 텍스트는 텍스트를 화면이나 C++ 코드에 나타나는 대로 표시합니다. 사용자가 입력해야 하는 내용 역시 이 글꼴로 나타냅니다.
[ ]	텍스트나 구문 리스트에서 대괄호는 옵션 항목을 묶습니다. 이러한 종류의 텍스트는 그대로 입력하면 안됩니다.
Boldface	텍스트 또는 코드 리스트에서 굵게 쓰여진 글자는 C++ 예약어 또는 컴파일러 옵션을 나타냅니다.
Italics	텍스트에서 이탤릭체 글자는 변수 또는 타입 이름 같은 C++ 식별자를 나타냅니다. 이탤릭체는 새로운 용어 같은 일부 단어를 강조하기 위해 사용하기도 합니다.
Keycaps	이 글꼴은 키보드에 있는 특정 키를 나타냅니다. 예를 들면, "*Esc* 키를 눌러 메뉴를 종료합니다."와 같이 사용합니다.

개발자 지원 서비스

Borland는 인터넷 상에서 무료 서비스를 비롯한 다양한 지원 옵션을 제공합니다. 개발자는 인터넷에서 광범위한 정보를 검색할 수 있고 Borland 제품, 기술 지원 및 유료 컨설턴트 지원을 사용하는 다른 개발자와 통신을 할 수 있습니다.

Borland의 개발자 지원 서비스에 대한 자세한 내용을 알아보려면 웹 사이트(http://www.borland.com/devsupport/bcppbuilder)를 참조하거나, Borland Assist((800) 523-7070)에 전화하거나, 판매 부서((831) 431-1064)로 문의하십시오. 미국 이외의 지역에 거주하는 고객은 http://www.borland.com/bww/intlcust.html을 참조하십시오.

지원을 받으려면 현재 작업 환경에 대한 자세한 정보, 사용 중인 제품의 버전 및 에디션, 문제에 대한 자세한 설명 등을 제공해야 합니다.

인쇄된 설명서 주문

추가 설명서를 주문하려면 Borland 웹 사이트인 shop.borland.com을 참조하십시오.

부

I

C++Builder 프로그래밍

"C++Builder 프로그래밍"에 포함된 장에서는 제품의 모든 에디션에서 C++Builder 애플리케이션을 만드는 데 필요한 개념과 기술에 대해 소개합니다.

장

2

C++Builder를 사용한 애플리케이션 개발

Borland C++Builder는 Windows와 Linux에서 배포용 32비트 애플리케이션을 개발할 수 있는 시각적인 객체 지향 프로그램입니다. C++Builder를 사용하면 수동 코딩 작업을 최소화하면서 성능이 뛰어난 애플리케이션을 작성할 수 있습니다.

C++Builder는 프로그래밍 마법사와 애플리케이션 및 폼 템플릿을 비롯한 신속한 애플리케이션 개발(RAD) 디자인 도구를 제공하고 다음과 같은 포괄적인 클래스 라이브러리를 사용하여 객체 지향 프로그램을 지원합니다.

- *비주얼 컴포넌트 라이브러리*(VCL) - Windows의 다른 유용한 프로그래밍 기법을 비롯한 Windows API를 캡슐화하는 객체를 포함
- *Borland 크로스 플랫폼 컴포넌트 라이브러리*(CLX) - Windows나 Linux의 Qt 라이브러리를 캡슐화하는 객체를 포함

이 장에서는 C++Builder 개발 환경 및 이러한 개발 환경이 개발 라이프 사이클에 어떻게 부합되는지를 간략하게 설명합니다. 이 설명서의 나머지 부분에서는 범용 데이터베이스, 인터넷 및 인트라넷 애플리케이션의 개발과 ActiveX와 COM 컨트롤 생성 및 고유 컴포넌트 작성에 대한 기술 정보를 제공합니다.

통합 개발 환경

C++Builder를 시작하면 IDE라고도 하는 통합 개발 환경 내에 바로 놓이게 됩니다. 이 IDE에서는 애플리케이션을 디자인, 개발, 테스트, 디버그 및 배포하는 데 필요한 모든 도구를 제공하므로 프로토타입을 빠르게 만들 수 있고 개발 시간을 단축시킬 수 있습니다.

IDE에는 애플리케이션을 디자인하는 데 필요한 다음과 같은 도구가 모두 포함됩니다.

- 애플리케이션에 필요한 UI를 디자인할 수 있는, 비어 있는 윈도우인 폼 디자이너(*폼*)
- 사용자 인터페이스를 디자인하는 데 사용할 수 있는 비주얼(visual) 및 넌비주얼(nonvisual) 컴포넌트를 표시하기 위한 컴포넌트 팔레트
- 객체의 속성과 이벤트를 확인하고 변경하기 위한 Object Inspector
- 컴포넌트의 논리 관계를 표시하고 변경하기 위한 Object TreeView
- 원본으로 사용하는 프로그램 로직을 작성하고 편집하기 위한 코드 에디터
- 하나 이상의 프로젝트를 구성하는 파일을 관리하기 위한 Project Manager
- 개발자 코드의 오류를 찾아 고치기 위한 통합 디버거
- 객체의 속성 값을 변경하기 위한 속성 에디터와 같은 많은 다른 도구
- 컴파일러, 링커 및 기타 유틸리티를 포함한 명령줄 도구
- 재사용할 수 있는 객체가 많은 광범위한 클래스 라이브러리. 클래스 라이브러리에 제공된 많은 객체는 컴포넌트 팔레트에서 IDE에 액세스할 수 있습니다. 규칙에 따라 클래스 라이브러리의 객체 이름은 *TStatusBar*처럼 T로 시작합니다.

모든 툴이 모든 제품 에디션에 포함되어 있는 것은 아닙니다.

개발 환경에 대한 자세한 설명은 제품과 함께 제공되는 *입문서*에 나와 있습니다. 그리고 온라인 도움말 시스템은 모든 메뉴, 다이얼로그 박스 및 윈도우에 대한 도움말을 제공합니다.

애플리케이션 디자인

C++Builder를 사용하여 범용 유틸리티에서 복잡한 데이터 액세스 프로그램이나 분산 애플리케이션에 이르는 모든 종류의 32비트 애플리케이션을 디자인할 수 있습니다.

애플리케이션의 사용자 인터페이스를 비주얼하게 디자인하면 C++Builder에서 원본으로 사용하는 C++ 코드를 생성하여 애플리케이션을 지원합니다. 컴포넌트와 폼의 속성을 선택하고 수정하면 변경한 내용의 결과가 소스 코드에 자동으로 나타나고 그 반대도 마찬가지입니다. 기본 코드 에디터를 비롯한 텍스트 에디터로 소스 파일을 직접 수정할 수 있습니다. 변경한 내용은 비주얼 환경에 즉시 반영됩니다.

C++Builder에서 새로운 컴포넌트를 만들 수 있습니다. 제공된 대부분의 컴포넌트는 오브젝트 파스칼로 작성됩니다. 작성한 컴포넌트를 컴포넌트 팔레트에 추가한 다음 필요하면 새로운 탭을 포함하여 팔레트를 사용자 정의할 수 있습니다.

C++Builder를 사용하여 CLX에 의해 Linux와 Windows에서 실행되는 애플리케이션을 디자인할 수도 있습니다. CLX에는 VCL의 클래스 집합 대신에 사용되면 프로그램을 Windows와 Linux 사이에 이식할 수 있게 해주는 클래스 집합이 포함되어 있습니다. 크로스 플랫폼 프로그래밍과 Windows와 Linux 환경 간의 차이에 대한 자세한 내용은 14장, "크로스 플랫폼 애플리케이션 개발"을 참조하십시오.

7장, "애플리케이션, 컴포넌트 및 라이브러리 생성"에서는 다양한 애플리케이션 타입에 대한 C++Builder 지원을 설명합니다.

프로젝트 생성

C++Builder 애플리케이션 개발의 모든 과정은 프로젝트에 따라 달라집니다. C++Builder에서 애플리케이션을 만들면 프로젝트를 만드는 것입니다. 프로젝트는 애플리케이션을 구성하는 파일을 모아 놓은 것입니다. 이러한 파일 중 일부는 디자인 타임 시 만들어집니다. 다른 파일은 프로젝트 소스 코드를 컴파일할 때 자동으로 생성됩니다.

Project Manager라는 프로젝트 관리 도구에서 프로젝트의 내용을 볼 수 있습니다. Project Manager는 유닛 이름과 유닛이 한 개인 경우, 유닛에 포함된 폼을 계층 뷰 방식으로 나열하고 프로젝트에 있는 파일의 경로를 보여 줍니다. 이러한 파일 중 많은 파일은 직접 편집할 수 있지만 C++Builder의 시각적 도구를 사용하는 것이 좀더 쉽고 확실합니다.

프로젝트 계층의 맨 위는 그룹 파일입니다. 여러 프로젝트를 한 프로젝트 그룹으로 결합할 수 있습니다. 결합하면 Project Manager에서 한 번에 여러 프로젝트를 열 수 있습니다. 프로젝트 그룹을 사용하면 함께 작동하거나 멀티 티어 애플리케이션의 일부로 작동하는 관련 프로젝트를 구성하여 작업할 수 있습니다. 한 프로젝트에서만 작업하는 경우 애플리케이션을 만들기 위해 프로젝트 그룹 파일을 사용할 필요가 없습니다.

각 프로젝트와 파일 및 관련 옵션을 설명하는 프로젝트 파일에는 .bpr 확장자가 있습니다. 프로젝트 파일에는 애플리케이션과 공유 객체 생성을 위한 설명이 들어 있습니다. Project Manager를 추가하고 제거하면 프로젝트 파일이 업데이트됩니다. 폼, 애플리케이션, 컴파일러와 같은 다양한 프로젝트 작업에 대한 탭이 있는 Project Options 다이얼로그 박스를 사용하여 프로젝트 옵션을 지정합니다. 이러한 프로젝트 옵션은 프로젝트와 함께 프로젝트 파일에 저장됩니다.

유닛과 폼은 C++Builder 애플리케이션의 기본 빌딩 블록입니다. 프로젝트는 프로젝트 디렉토리 트리 외부에 거주하는 폼과 유닛 파일 이외에도 기존 폼과 유닛 파일을 공유할 수 있습니다. 독립 루틴으로 작성된 사용자 정의 프로시저와 함수도 공유할 수 있습니다.

공유 파일을 프로젝트에 추가하면 파일이 현재 프로젝트 디렉토리로 복사되는 것이 아니라 현재 위치에 남아 있습니다. 공유 파일을 현재 프로젝트에 추가하면 파일 이름과 경로가 프로젝트 파일에 등록됩니다. 사용자가 유닛을 프로젝트에 추가하면 C++Builder에서 자동으로 등록합니다.

프로젝트를 컴파일할 때 프로젝트를 구성하는 파일의 장소는 중요하지 않습니다. 컴파일러는 공유 파일을 프로젝트 자체에서 만든 파일과 똑같이 취급합니다.

코드 편집

C++Builder 코드 에디터는 모든 기능을 갖춘 아스키 에디터입니다. 비주얼(visual) 프로그래밍 환경에서는 폼이 자동으로 새 프로젝트의 일부로 표시됩니다. 객체를 폼에 두고 Object Inspector에서 작동하는 방법을 수정하면 애플리케이션 인터페이스 디자인을 시작할 수 있습니다. 하지만 객체의 이벤트 핸들러 작성과 같은 다른 프로그래밍 작업은 코드를 입력해야 합니다.

폼의 내용과 모든 속성 및 컴포넌트 그리고 컴포넌트의 속성은 코드 에디터에서 텍스트로 보고 편집할 수 있습니다. 코드 에디터에서 생성된 코드를 조정하고 에디터 내에서 코드를 입력하여 컴포넌트를 더 추가할 수 있습니다. 코드를 에디터에 입력할 때 컴파일러에서 즉시 변경한 내용을 검사하여 새 레이아웃으로 폼을 업데이트합니다. 그리고 나서 폼으로 돌아가서 에디터에서 변경한 내용을 보고 테스트한 다음 그 부분부터 폼을 계속 조정할 수 있습니다.

C++Builder 코드 생성 및 속성 스트리밍 시스템은 검사할 수 있도록 완전히 개방됩니다. VCL 객체, CLX 객체, RTL 소스 및 프로젝트 파일 등 최종 실행 파일에 포함되어 있는 모든 내용의 소스 코드를 코드 에디터에서 보고 편집할 수 있습니다.

애플리케이션 컴파일

폼에서 애플리케이션 인터페이스를 디자인한 다음 이 인터페이스에서 원하는 작업을 수행하도록 추가 코드를 작성하였으면 IDE나 명령줄에서 해당 프로젝트를 컴파일할 수 있습니다.

모든 프로젝트는 한 개의 배포 가능한 실행 파일을 대상으로 갖습니다. 다음과 같이 애플리케이션을 컴파일하고 생성, 실행하여 다양한 개발 단계에서 애플리케이션을 보거나 테스트할 수 있습니다.

- 컴파일할 때는 마지막 컴파일 후 변경한 유닛만 다시 컴파일됩니다.
- 생성할 때는 마지막 컴파일 후 변경 여부에 관계 없이 프로젝트의 모든 유닛이 컴파일됩니다. 이러한 기술은 변경된 파일을 정확히 알 수 없을 때나 모든 파일이 있으며 동기화되었는지만 확인하고 싶은 경우에 유용합니다. 전역 컴파일러 지시어를 변경한 경우에도 모든 코드가 올바른 단계에서 컴파일되었는지 확인하기 위해 생성해야 합니다. 그리고 프로젝트를 컴파일하지 않고 소스 코드의 유효성을 테스트할 수 있습니다.
- 실행할 때 애플리케이션을 컴파일한 다음 실행합니다. 마지막 컴파일 이후에 소스 코드를 수정한 경우 컴파일러는 변경된 모듈을 다시 컴파일하여 애플리케이션을 다시 연결합니다.

여러 프로젝트를 함께 그룹화하면 모든 프로젝트를 단일 프로젝트 그룹으로 한 번에 컴파일하거나 생성할 수 있습니다. Project Manager에서 선택한 프로젝트 그룹과 함께 Project | Compile All Projects나 Project | Build All Projects를 선택합니다.

CLX Linux에서 CLX 애플리케이션을 컴파일하기 위한 Linux C++ 애플리케이션은 아직 사용할 수 없지만 이제 C++Builder를 사용하여 애플리케이션을 개발할 수 있습니다.

애플리케이션 디버깅

C++Builder는 애플리케이션에서 오류를 찾아 수정할 수 있게 해주는 통합 디버거를 제공합니다. 통합 디버거를 사용하면 프로그램 실행을 제어하고 데이터 구조의 다양한 값과 항목을 모니터할 수 있으며 디버깅하는 동안 데이터 값을 수정할 수도 있습니다.

통합 디버거는 런타임 오류와 논리 오류를 모두 추적할 수 있습니다. 특정 프로그램 위치로 실행하고 변수 값과 호출 스택의 함수, 프로그램 출력을 확인하여 프로그램의 동작 방법을 모니터하고 디자인한 대로 작동하지 않는 부분을 찾을 수 있습니다. 디버거는 온라인 도움말에 설명되어 있습니다.

예외 처리를 사용하여 오류를 인식하고 찾아 처리할 수도 있습니다. C++Builder의 예외는 규칙에 따라 T가 아닌 E로 시작한다는 점만 다르고 C++Builder의 다른 클래스처럼 클래스입니다. 예외 처리에 대한 자세한 내용은 12장, "예외 처리"를 참조하십시오.

애플리케이션 배포

C++Builder에는 애플리케이션 배포에 도움이 되는 추가 도구가 있습니다. 예를 들어, 모든 에디션에서 사용할 수 있는 것은 아니지만 InstallShield Express를 사용하면 분산 애플리케이션을 실행하는 데 필요한 모든 파일을 포함하는, 애플리케이션 설치 패키지를 만들 수 있습니다. 모든 에디션에서 사용할 수 있는 것은 아니지만 TeamSource 소프트웨어를 사용해도 애플리케이션 업데이트 내용을 추적할 수 있습니다.

참고 C++Builder의 모든 에디션에 배포 기능이 있는 것은 아닙니다.

CLX Linux에서 CLX 애플리케이션을 배포하는 Linux C++ 애플리케이션은 아직 사용할 수 없지만 이제 C++Builder를 사용하여 애플리케이션을 개발할 수 있습니다.

배포에 대한 자세한 내용은 17장, "애플리케이션 배포"를 참조하십시오.

장

3

클래스 라이브러리 사용

이 장에서는 클래스 라이브러리의 개요를 제공하고, 애플리케이션 개발에 사용할 수 있는 여러 컴포넌트를 소개합니다. C++Builder에는 비주얼 컴포넌트 라이브러리(VCL)와 Borland 크로스 플랫폼 컴포넌트 라이브러리(CLX)가 모두 포함되어 있습니다. VCL은 Windows 개발에 사용되고, CLX는 Windows와 Linux 모두의 크로스 플랫폼 개발에 사용됩니다. 이 두 클래스 라이브러리는 서로 다르지만 여러 가지 면에서 비슷합니다.

클래스 라이브러리 이해

VCL 및 CLX는 애플리케이션 개발에 사용하는 객체로 구성된 클래스 라이브러리입니다. 라이브러리는 서로 비슷하며 동일한 여러 객체를 포함합니다. 컴포넌트 팔레트에서 ADO, BDE, QReport, COM+ 및 Servers 탭에 표시되는 객체와 같은 VCL의 일부 객체는 Windows 전용 기능을 구현합니다. 실제로 모든 CLX 객체는 Windows와 Linux 모두에서 사용할 수 있습니다.

모든 VCL 및 CLX 객체는 해당 메소드가 생성, 소멸 및 메시지 처리와 같은 기본 동작을 캡슐화하는 추상 클래스인 *TObject*의 자손입니다. 클래스를 직접 작성할 경우 클래스는 사용할 클래스 라이브러리에 있는 *TObject*의 자손이어야 합니다.

*컴포넌트*는 VCL이나 CLX의 부분 집합이며 추상 클래스인 *TComponent*의 자손입니다. 컴포넌트를 폼이나 데이터 모듈에 배치하여 디자인 타임 시 처리할 수 있습니다. 대부분의 컴포넌트는 런타임 시 보이는지 여부에 따라 비주얼(visual) 컴포넌트이거나 넌비주얼(nonvisual) 컴포넌트입니다. 일부 컴포넌트는 컴포넌트 팔레트에 나타납니다.

TForm 및 *TSpeedButton*과 같은 비주얼 컴포넌트를 *컨트롤*이라고 하며, 이러한 컴포넌트는 *TControl*의 자손입니다. *TControl*은 높이나 너비와 같은 컨트롤의 비주얼 어트리뷰트(attribute)를 지정하는 속성을 제공합니다.

넌비주얼(nonvisual) 컴포넌트는 여러 가지 작업에 사용됩니다. 예를 들면, 데이터베이스에 연결하는 애플리케이션을 작성할 경우 폼에 *TDataSource* 컴포넌트를 배치하여 컨트롤과 컨트롤이 사용하는 데이터셋을 연결할 수 있습니다. 사용자가 이 연결을 볼 수 없으므로 *TDataSource*는 넌비주얼입니다. 디자인 타임 시 넌비주얼 컴포넌트는 아이콘으로 표시됩니다. 이 아이콘을 사용하여 보이는 컨트롤을 처리할 때와 같이 해당 속성과 이벤트를 처리할 수 있습니다.

프로그래밍 도중 온라인 도움말을 사용하여 VCL 및 CLX의 모든 객체에 대한 자세한 참조 자료에 액세스할 수 있습니다. 코드 에디터에서 커서를 객체의 아무데나 놓고 F1을 누르면 도움말 항목이 나타납니다. VCL의 객체, 속성, 메소드 및 이벤트는 "VCL Reference"로 표시되며, CLX의 객체, 속성, 메소드 및 이벤트는 "CLX Reference"로 표시됩니다.

속성, 메소드 및 이벤트

VCL과 CLX 모두 IDE에 연결된 객체의 계층을 이룹니다. 이 IDE에서 애플리케이션을 빨리 개발할 수 있습니다. 두 컴포넌트 라이브러리에 있는 객체는 속성, 메소드 및 이벤트 기반입니다. 각 객체는 데이터 멤버(속성), 데이터에 작동하는 함수(메소드), 클래스의 사용자들과 상호 작용하는 방법(이벤트)을 포함합니다. VCL은 Windows API 기반이고, CLX는 Qt widget 라이브러리 기반이지만 VCL과 CLX는 오브젝트 파스칼로 작성됩니다.

속성

*속성*은 객체의 비주얼한 동작 또는 작동에 영향을 주는 객체의 특성입니다. 예를 들어, *Visible* 속성은 객체가 애플리케이션 인터페이스에서 보이는지 여부를 결정합니다. 속성을 잘 디자인하면 다른 사용자가 컴포넌트를 쉽게 사용할 수 있으며, 컴포넌트를 유지 보수하기도 쉽습니다.

속성의 장점은 다음과 같습니다.

- 런타임에만 사용 가능한 메소드와는 달리, 디자인 타임에 속성을 보고 변경할 수 있으며, IDE에서 컴포넌트가 변경될 때 즉시 피드백을 얻을 수 있습니다.
- 객체 값을 비주얼하게 수정할 수 있는 Object Inspector에서 속성에 액세스할 수 있습니다. 디자인 타임에 속성을 설정하면 코드를 직접 작성하는 것보다 쉽고, 코드를 간편하게 유지 보수할 수 있습니다.
- 데이터가 캡슐화되어 있으므로 데이터가 보호되며 실제 객체에 private으로 사용됩니다.
- 값을 얻고 설정하는 실제 호출은 메소드이므로, 특별한 처리를 수행하여 객체의 사용자에게 보이지 않도록 할 수 있습니다. 예를 들어, 데이터는 테이블에 상주할 수 있지만 프로그래머에게 일반적인 데이터 멤버로 나타날 수 있습니다.
- 속성을 액세스하는 동안 이벤트를 발생시키거나 다른 데이터를 수정하는 로직을 구현할 수 있습니다. 예를 들어, 특정 속성 값을 변경하려면 다른 속성을 수정해야 합니다. 속성에 대해 만들어진 메소드를 변경할 수 있습니다.
- 속성은 가상(virtual)일 수 있습니다.
- 속성은 단일 객체에 국한되지 않습니다. 한 객체에 대한 속성을 변경하는 것은 여러 객체에 영향을 미칠 수 있습니다. 예를 들어, 하나의 라디오 버튼에 *Checked* 속성을 설정하면 그룹의 모든 라디오 버튼에 적용됩니다.

메소드

*메소드*는 클래스의 멤버인 함수입니다. 메소드는 객체의 동작을 정의합니다. 클래스 메소드는 모든 *public*, *protected* 및 *private* 속성과 클래스의 데이터 멤버에 액세스할 수 있으며, 일반적으로 클래스 메소드를 멤버 함수라고 합니다. 46-4페이지의 "액세스 제어"를 참조하십시오.

이벤트

*이벤트*는 프로그램에 의해 탐지된 동작이나 발생한 사건입니다. 대부분의 최신 애플리케이션은 이벤트에 응답하도록 디자인되기 때문에 이벤트 방식이라고 합니다. 프로그램에서 프로그래머는 사용자가 다음에 수행할 동작의 순서를 정확하게 예측할 수 없습니다. 예를 들어, 사용자가 메뉴 항목을 선택하거나, 버튼을 누르거나, 텍스트 일부를 표시할 수도 있습니다. 항상 똑같은 순서로 실행되는 코드를 작성하기보다는 개발자가 원하는 이벤트를 처리할 코드를 작성할 수 있습니다.

이벤트 호출 방법에 상관 없이, C++Builder에서는 이벤트를 처리할 코드를 작성했는지 확인합니다. 이러한 코드를 작성한 경우에는 코드가 실행되지만, 그렇지 않으면 디폴트 이벤트 처리 동작이 실행됩니다.

발생할 수 있는 이벤트 종류는 두 가지 주요 범주로 나뉠 수 있습니다.

- 사용자 이벤트
- 시스템 이벤트

사용자 이벤트

사용자 이벤트는 사용자가 시작하는 동작입니다. 사용자 이벤트의 예로는 *OnClick*(사용자가 마우스를 클릭), *OnKeyPress*(사용자가 키보드의 키를 누름) 및 *OnDblClick*(사용자가 마우스 버튼을 더블 클릭)이 있습니다.

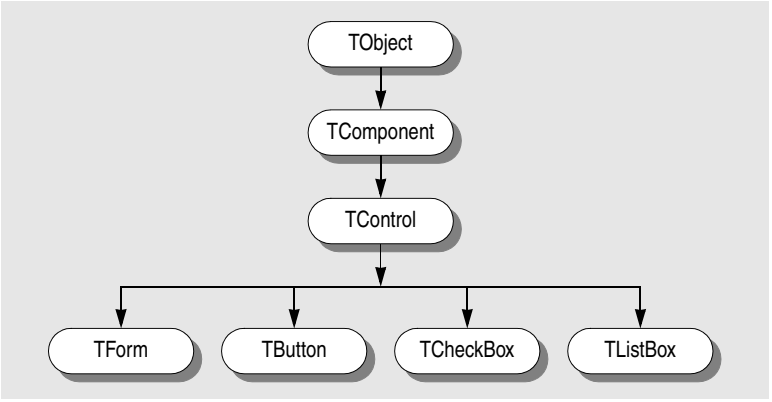
시스템 이벤트

시스템 이벤트는 운영 체제에서 시작하는 이벤트입니다. 예를 들어, *OnTimer* 이벤트(Timer 컴포넌트는 정해진 시간이 경과되면 이벤트를 발생시킴), *OnCreate* 이벤트(컴포넌트 생성 중), *OnPaint* 이벤트(다시 그려야 하는 컴포넌트 또는 윈도우) 등이 있습니다. 대개 시스템 이벤트는 사용자 동작을 통해 직접 시작되지 않습니다.

객체, 컴포넌트 및 컨트롤

그림 3.1은 객체, 컴포넌트 및 컨트롤의 관계를 나타내는 상속 계층을 매우 간단하게 표시한 것입니다.

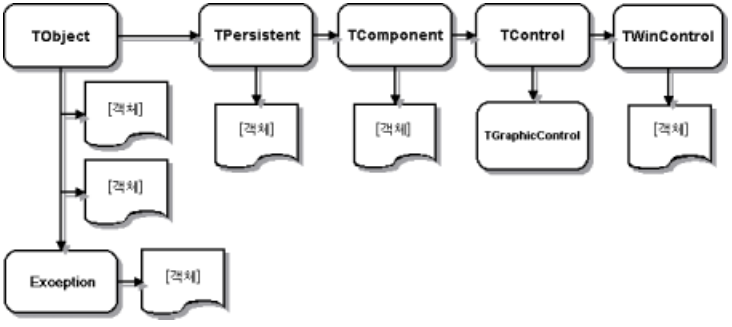
그림 3.1 객체, 컴포넌트 및 컨트롤



모든 객체는 *TObject*로부터 상속되며, 많은 객체가 *TComponent*로부터 상속됩니다. 컨트롤은 *TControl*에서 상속되며 런타임에 표시될 수 있습니다. *TCheckBox*와 같은 컨트롤은 *TObject*, *TComponent* 및 *TControl*의 모든 기능을 상속하고, 특화된 자체 기능을 추가합니다.

그림 3.2는 상속 트리의 주요 분기를 표시하는 비주얼 컴포넌트 라이브러리(VCL)의 개요입니다. *TWinControl*이 *TWidgetControl*로 대체되지만 이 수준에서는 Borland 크로스 플랫폼 컴포넌트 라이브러리(CLX)가 VCL과 매우 유사합니다.

그림 3.2 단순화된 계층 다이어그램



여러 중요한 기본 클래스가 그림에 나타나 있습니다. 다음 표는 기본 클래스를 설명한 것입니다.

표 3.1 중요한 기본 클래스

클래스	설명
TObject	VCL이나 CLX에서 모든 클래스의 기본 클래스이며 최고 조상입니다. *TObject*는 객체의 인스턴스 생성, 유지 보수 및 소멸과 같은 기본 기능을 수행하는 메소드를 소개하여 VCL/CLX의 모든 객체에 공통되는 기본 동작을 캡슐화합니다.
Exception	예외와 관련된 모든 클래스의 기본 클래스를 지정합니다. *Exception*는 오류 상태에 대해 일관된 인터페이스를 제공하고, 애플리케이션이 오류 상태를 잘 처리할 수 있게 합니다.
TPersistent	속성을 구현하는 모든 객체에 대한 기본 클래스를 지정합니다. *TPersistent*의 자손 클래스는 데이터를 스트림에 보내는 것을 처리하고, 클래스의 할당을 허용합니다.
TComponent	*TApplication*과 같은 모든 넌비주얼(nonvisual) 컴포넌트에 대한 기본 클래스를 지정합니다. *TComponent*는 모든 컴포넌트의 공통 조상입니다. 이 클래스를 사용하면 컴포넌트 팔레트에 컴포넌트를 표시할 수 있고, 컴포넌트가 다른 컴포넌트를 소유할 수 있으며, 컴포넌트를 폼에서 직접 처리할 수 있습니다.
TControl	런타임에 보이는 모든 컨트롤에 대한 기본 클래스를 나타냅니다. *TControl*은 모든 비주얼 컴포넌트의 공통 조상이며, 위치나 커서 같은 표준 비주얼 컨트롤을 제공합니다. 또한 이 클래스는 마우스 동작에 응답하는 이벤트를 제공합니다.
TWinControl	모든 사용자 인터페이스 객체의 기본 클래스를 지정합니다. *TWinControl*의 자손 컨트롤은 키보드 입력을 캡처할 수 있는 윈도우 컨트롤입니다. CLX에서는 이러한 컨트롤을 widget이라고 하며, *TWidgetControl*이 *TWinControl*을 대체합니다.

다음 단원들에서는 각 분기에 포함되는 클래스 타입에 대한 일반적인 사항을 설명합니다. VCL 및 CLX 객체 계층의 전체 개요에 대해서는 이 제품과 함께 포함된 VCL 객체 계층 및 CLX 객체 계층 차트를 참조하십시오.

TObject 분기

TObject 분기에는 *TObject*의 자손인 모든 VCL 및 CLX 객체가 포함되지만 *TPersistent*의 자손 객체는 포함되지 않습니다. VCL 및 CLX 객체가 가진 대부분의 강력한 기능은 *TObject*의 메소드를 통해 구현됩니다. *TObject*는 다음을 제공하는 메소드를 통해 VCL 및 CLX의 모든 객체에 공통적인 기본 동작을 캡슐화합니다.

- 객체가 생성되거나 소멸될 때 응답하는 기능
- 객체의 클래스 타입과 인스턴스 정보 및 객체의 published 속성에 대한 런타임 타입 정보(RTTI)
- 메시지 처리(VCL) 또는 시스템 이벤트(CLX)에 대한 지원

*TObject*는 여러 단순 클래스의 직계 조상입니다. *TObject* 분기 내에 포함된 클래스는 일시적이라는 공통된 중요한 특성이 있습니다. 즉, 이러한 클래스는 일시적입니다. 클래스에 소멸되기 전의 상태를 저장할 메소드가 없다는 의미이므로 이 클래스는 영구적이지 않습니다.

이 분기에 있는 클래스의 주요 그룹 중 하나는 *Exception* 클래스입니다. 이 클래스는 0으로 나누기 오류, 파일 I/O 오류, 잘못된 타입 변환 및 많은 기타 예외 상태를 자동으로 처리하여 대규모의 기본 예외 클래스 집합을 제공합니다.

TObject 분기에 있는 다른 그룹의 타입은 데이터 구조를 캡슐화하는 다음과 같은 클래스입니다.

- 부울 값 "배열"을 저장하는 클래스인 *TBits*
- 연결 리스트(linked list) 클래스인 *TList*
- 포인터의 후입선출(LIFO) 배열을 유지 보수하는 클래스인 *TStack*
- 포인터의 선입선출(FIFO) 배열을 유지 보수하는 클래스인 *TQueue*

또한 VCL에는 Windows 프린터 인터페이스를 캡슐화하는 *TPrinter*와 같은 외부 객체에 대한 랩퍼 및 *TRegistry*와 같이 레지스트리에서 작동하는 시스템 레지스트리와 함수에 대한 저수준 랩퍼가 있습니다. 이러한 랩퍼는 Windows 환경 전용입니다.

*TStream*은 이 분기에 있는 다른 클래스 타입에 대한 좋은 예입니다. *TStream*은 디스크 파일, 동적 메모리 등과 같은 다양한 저장 매체로부터 읽고 쓸 수 있는 스트림 객체에 대한 기본 클래스 타입입니다.

대개 이 분기에는 개발자에게 유용한 여러 가지 다른 클래스 타입이 포함됩니다.

TPersistent 분기

TPersistent 분기에는 *TPersistent*의 자손이지만 *TComponent*의 자손은 아닌 모든 VCL 및 CLX 객체가 포함됩니다. 영구성은 폼 파일이나 데이터 모듈에 저장되는 것과 메모리에서 검색할 때 폼이나 데이터 모듈로 로드되는 것을 결정합니다.

이 분기에 있는 객체는 컴포넌트의 속성을 구현합니다. 속성은 소유자(owner)가 있는 경우에 폼과 함께 로드되고 저장됩니다. 소유자는 특정 컴포넌트여야 합니다. 이 분기에는 속성의 소유자를 결정하는 데 사용되는 *GetOwner* 함수가 있습니다.

또한 이 분기에 있는 객체는 속성이 자동으로 로드되고 저장될 수 있는 published 섹션을 포함하는 첫 번째 객체입니다. *DefineProperties* 메소드를 사용하여 속성을 로드하고 저장하는 방법을 지시할 수도 있습니다.

다음은 계층의 TPersistent 분기에 있는 몇몇 다른 클래스입니다.

- *TGraphicsObject*는 *TBrush*, *TFont* 및 *TPen*과 같은 그래픽 객체의 기본 추상 클래스입니다.
- *TGraphic*은 *TBitmap* 및 *TIcon*과 같은 객체의 기본 추상 클래스로서 비주얼 이미지를 저장하고 표시합니다.
- *TStrings*는 문자열 리스트를 나타내는 객체의 기본 클래스입니다.
- *TClipboard*는 애플리케이션에서 잘라내거나 복사한 텍스트 또는 그래픽을 포함하는 클래스입니다.
- *TCollection*, *TOwnedCollection* 및 *TCollectionItem*은 특별히 정의된 항목의 인덱스된 컬렉션을 유지 보수하는 클래스입니다.

TComponent 분기

TComponent 분기는 *TControl*이 아닌 *TComponent*의 자손인 객체를 포함합니다. 이 분기에 있는 객체는 디자인 타임에 폼에서 처리할 수 있는 컴포넌트입니다. 이 객체는 다음을 수행할 수 있는 영구적 객체입니다.

- 컴포넌트 팔레트에 나타나고 폼 디자이너에서 변경할 수 있습니다.
- 다른 컴포넌트를 소유하고 관리합니다.
- 로드 및 저장됩니다.

*TComponent*의 여러 메소드는 디자인 타임 동안 컴포넌트가 작동하는 방법과 그 컴포넌트와 함께 저장되는 정보를 지시합니다. VCL 및 CLX의 이 분기에 스트리밍이 도입되었습니다. C++Builder는 대부분의 스트리밍 작업을 자동으로 처리합니다. 속성이 published인 경우 속성은 영구적이고, published 속성은 자동으로 스트리밍됩니다.

TComponent 클래스는 VCL 및 CLX 를 통해 전파되는 소유권의 개념을 처음으로 사용한 클래스입니다. *Owner*와 *Components* 속성은 소유권을 지원합니다. 각 컴포넌트는 다른 컴포넌트를 자신의 소유자로 참조하는 *Owner* 속성을 가집니다. 컴포넌트는 다른 컴포넌트를 소유할 수 있습니다. 이 경우에 모든 소유된 컴포넌트는 컴포넌트의 *Array* 속성으로 참조됩니다.

컴포넌트의 생성자는 새 컴포넌트의 소유자를 지정하는 데 사용되는 매개변수를 하나 가지고 있습니다. 전달된 소유자가 있으면 새 컴포넌트는 소유자의 컴포넌트 리스트에 추가됩니다. 소유된 컴포넌트를 참조하는 컴포넌트 리스트를 사용하는 것과 별도로 , 이 속성은 또한 소유된 컴포넌트의 자동 소멸을 제공합니다. 컴포넌트의 소유자가 있을 경우 소유자가 소멸되면 소유된 컴포넌트도 함께 소멸됩니다. 예를 들어, *TForm*은 *TComponent*의 자손이므로, 폼이 소멸되면 폼이 소유한 모든 컴포넌트가 소멸되고 컴포넌트의 메모리는 해제됩니다. 이것은 컴포넌트의 소멸자를 호출할 때 폼의 모든 컴포넌트가 제대로 해제된다는 것을 가정합니다.

속성 타입이 *TComponent* 또는 자손인 경우, 스트리밍 시스템은 읽을 때 해당 타입의 인스턴스를 생성합니다. 속성 타입이 *TComponent*가 아닌 *TPersistent*인 경우, 스트리밍 시스템은 속성을 통해 사용 가능한 기존 인스턴스를 사용하고 해당 인스턴스의 속성 값을 읽습니다.

폼 파일(폼에 있는 컴포넌트에 대한 정보를 저장하는 데 사용하는 파일)을 만들 때, 폼 디자이너는 폼의 컴포넌트 배열을 반복하고 폼의 모든 컴포넌트를 저장합니다. 각 컴포넌트는 변경된 속성을 스트림(이 경우에 텍스트 파일)에 쓰는 방법을 "압니다". 반면에, 폼 디자이너는 컴포넌트의 속성을 폼 파일에 로드할 때, 컴포넌트 배열을 반복하고 각 컴포넌트를 로드합니다.

이 분기에서 찾을 수 있는 클래스 타입은 다음과 같습니다.

- *TActionList*는 메뉴 항목 및 버튼과 같이 컴포넌트 및 컨트롤과 함께 사용되는 동작의 리스트를 유지 보수하는 클래스입니다.
- *TMainMenu*는 폼에 대한 메뉴 바와 메뉴 바의 드롭다운 메뉴를 제공하는 클래스입니다.
- *TOpenDialog*, *TSaveDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog* 등은 공용으로 사용되는 다이얼로그 박스를 제공합니다.
- *TScreen*은 애플리케이션에 의해 인스턴스화된 폼과 데이터 모듈, 활성 폼, 폼의 활성 컨트롤, 화면 크기 및 해상도, 애플리케이션에 사용 가능한 커서 및 글꼴 등을 추적하는 클래스입니다.

비주얼 인터페이스가 필요 없는 컴포넌트는 *TComponent*에서 직접 파생될 수 있습니다. *TTimer* 장치와 같은 도구를 만들기 위해 *TComponent*에서 파생시킬 수 있습니다. 이러한 타입의 컴포넌트는 컴포넌트 팔레트에 있지만 런타임 시 사용자 인터페이스에 나타나지 않고 코드를 통해 액세스되는 내부 함수를 수행합니다.

또한 CLX에서는 *TComponent* 분기가 *THandleComponent*를 포함할 수 있습니다. 이 클래스는 다이얼로그 박스와 메뉴 등 원본으로 사용하는 Qt 객체에 대한 핸들을 필요로 하는 넌비주얼(nonvisual) 컴포넌트의 기본 클래스입니다.

속성 설정, 메소드 호출 및 컴포넌트에 대한 이벤트를 사용하는 작업에 대한 자세한 내용은 5장, "컴포넌트 작업"을 참조하십시오.

TControl 분기

TControl 분기는 *TWinControl*(CLX에서는 *TWidgetControl*)이 아닌 *TControl*의 자손인 컴포넌트로 구성됩니다. 이 분기의 객체는 애플리케이션 사용자가 런타임 시 보고 처리할 수 있는 비주얼 객체인 컨트롤입니다. 모든 컨트롤에는 컨트롤의 위치, 컨트롤의 윈도우(CLX에서는 widget) 와 연결된 커서, 컨트롤을 그리거나 이동하는 메소드 및 마우스 동작에 응답하는 이벤트와 같은 컨트롤의 모양과 관련 있는 공통적인 속성, 메소드 및 이벤트가 있습니다. 컨트롤은 키보드 입력을 받을 수 없습니다.

*TComponent*는 모든 컴포넌트에 대한 동작을 정의하는 반면에 *TControl*은 모든 비주얼 컨트롤에 대한 동작을 정의합니다. 여기에는 드로잉 루틴, 표준 이벤트 및 포함 관계(containership) 등이 포함됩니다.

모든 비주얼 컨트롤은 특정 속성을 공유합니다. 이러한 속성이 *TControl*에서 상속되지만 적용 가능한 컴포넌트에 대해서만 게시되어 Object Inspector에 나타납니다. 예를 들어, *TImage*는 표시하는 그래픽에 의해 해당 색이 결정되므로 *Color* 속성을 게시하지 않습니다.

컨트롤 타입에는 다음 두 가지가 있습니다.

- 각각의 윈도우(또는 widget)가 있는 컨트롤
- 부모의 윈도우(또는 widget)를 사용하는 컨트롤

각각의 윈도우를 가진 컨트롤을 "윈도우" 컨트롤(VCL) 또는 "widget 기반" 컨트롤(CLX)이라고 하며, 이러한 컨트롤은 *TWinControl*(CLX에서는 *TWidgetControl*)의 자손입니다. 버튼과 체크 박스는 이 클래스에 속합니다.

부모의 윈도우(또는 widget)를 사용하는 컨트롤은 그래픽 컨트롤이라고 하며, 이러한 컨트롤은 *TGraphicControl*의 자손입니다. 이미지와 도형이 이 클래스에 속합니다. 그래픽 컨트롤에는 핸들이 없으며 입력 포커스를 받을 수 없습니다. 그래픽 컨트롤은 핸들을 필요로 하지 않으므로 시스템 리소스가 적게 사용됩니다. 그래픽 컨트롤은 스스로 그려지며 다른 컨트롤의 부모가 될 수 없습니다.

다른 그래픽 컨트롤에 대한 자세한 내용은 9-17페이지의 "그래픽 컨트롤"을 참조하고, 다른 컨트롤 타입에 대한 자세한 내용은 9장, "컨트롤 타입"을 참조하십시오. 런타임에 컨트롤과 상호작용하는 방법에 대한 자세한 내용은 6장, "컨트롤 작업"을 참조하십시오.

TWinControl/TWidgetControl 분기

VCL에서 *TWinControl* 분기는 *TWinControl*의 자손인 모든 컨트롤을 포함합니다. *TWinControl*은 버튼, 레이블, 스크롤 막대 등 애플리케이션의 사용자 인터페이스에 사용할 항목인 모든 윈도우 컨트롤의 기본 클래스입니다. 윈도우 컨트롤은 Windows 컨트롤의 랩퍼입니다.

CLX에서 *TWinControl*을 대신하는 *TWidgetControl*은 *widget*의 랩퍼인 모든 widget 컨트롤의 기본 클래스입니다.

윈도우 컨트롤 및 widget 컨트롤의 특징

- 애플리케이션이 실행되는 동안 포커스를 받을 수 있습니다. 즉, 애플리케이션 사용자의 키보드 입력을 받을 수 있습니다. 그러나 다른 컨트롤은 데이터를 표시할 수만 있습니다.
- 하나 이상의 자식 컨트롤의 부모가 될 수 있습니다.
- 핸들이나 고유한 식별자가 있습니다.

TWinControl/*TWidgetControl* 분기는 자동으로 그려지는 컨트롤(*TEdit*, *TListBox*, *TComboBox*, *TPageControl* 등)과 *TDBNavigator*, *TMediaPlayer*(VCL 전용) 및 *TGauge*(VCL 전용)와 같이 C++Builder가 그려야 하는 사용자 정의 컨트롤을 둘 다 포함합니다. *TWinControl*/*TWidgetControl*의 직계 자손은 일반적으로 에디트 필드, 콤보 박스, 리스트 박스, 페이지 컨트롤 등과 같은 표준 컨트롤을 구현하므로 자신을 그리는 방법을 이미 알고 있습니다.

TCustomControl 클래스는 윈도우 핸들을 필요로 하지만 핸들 자체를 다시 그리는 기능이 있는 표준 컨트롤을 캡슐화하지 않는 컴포넌트에 대해 제공됩니다. 컨트롤이 렌더링되는 방법이나 이벤트에 응답하는 방법은 문제가 되지 않습니다. C++Builder가 이 동작을 완전히 캡슐화합니다.

장

4

BaseCLX 사용

VCL과 CLX에서 공통으로 사용되고 두 컴포넌트 라이브러리에 기본 지원을 제공하는 유닛이 많이 있으며, 이런 유닛을 총칭하여 BaseCLX라고 합니다. BaseCLX는 컴포넌트 팔레트에 나타나는 컴포넌트를 포함하지 않습니다. 대신 컴포넌트 팔레트에 나타나는 컴포넌트에 의해 사용되는 여러 가지 클래스와 전역 루틴을 포함합니다. 또한 이러한 클래스와 루틴은 애플리케이션 코드에서 사용할 수 있거나 고유의 클래스를 작성할 때 사용할 수 있습니다.

참고 BaseCLX를 구성하는 전역 루틴은 종종 런타임 라이브러리로 불립니다. 이러한 루틴과 C++ 런타임 라이브러리를 혼동하지 마십시오. 이러한 루틴 중에는 C++ 런타임 라이브러리와 유사한 역할을 하는 것도 많으나 함수 이름이 대문자로 시작하며 유닛의 헤더에서 선언되는 점으로 구분할 수 있습니다.

다음 부분에서는 BaseCLX를 구성하는 다양한 클래스와 루틴에 대한 설명 및 사용 방법을 설명합니다. 사용 방법에는 다음이 포함됩니다.

- 스트림 사용
- 파일 작업
- .ini 파일 및 시스템 레지스트리 작업
- 리스트 작업
- 문자열 리스트 작업
- 문자열 작업
- 측정값 변환
- 그리기 공간 생성

참고 이 작업 리스트는 전체 리스트가 아닙니다. BaseCLX의 런타임 라이브러리에는 이 리스트에 언급되지 않은 작업을 수행하는 여러 가지 루틴이 있습니다. 즉, Math 유닛에 정의된 수학 함수의 호스트, SysUtils 유닛과 DateUtils 유닛에 정의된 날짜/시간 값을 사용하는 루틴 및 Variants 유닛에 정의된 오브젝트 파스칼 가변 타입을 사용하는 루틴 등이 있습니다.

스트림 사용

스트림은 데이터를 읽고 쓸 수 있게 하는 클래스입니다. 스트림은 메모리, 문자열, 소켓 및 데이터베이스 내의 BLOB 필드와 같은 다른 매체에 읽고 쓰기 위한 공통적인 인터페이스를 제공합니다. *TStream*의 자손인 몇몇 스트림 클래스가 있습니다. 각 스트림 클래스는 하나의 매체 타입에만 한정됩니다. 예를 들면, *TMemoryStream*은 메모리 이미지에서 읽고 쓰며 *TFileStream*은 파일에서 읽고 씁니다.

스트림을 사용하여 데이터 읽기 또는 쓰기

스트림 클래스는 모두 데이터를 읽고 쓰기 위해 몇몇 메소드를 공유합니다. 이러한 방법들은 다음을 수행하는지에 따라 구별됩니다.

- 읽거나 쓴 바이트 수를 반환합니다.
- 바이트 수가 알려져 있어야 합니다.
- 오류에 대한 예외를 발생시킵니다.

읽기 및 쓰기용 스트림 메소드

Read 메소드는 현재 *Position*에서 스트림으로부터 지정된 수의 바이트를 버퍼로 읽습니다. 그리고 나서 *Read*는 현재 위치를 실제로 전송된 바이트 수만큼 앞으로 나가게 합니다. 다음은 *Read*에 대한 프로토타입입니다.

```
virtual int __fastcall Read(void *Buffer, int Count);
```

*Read*는 파일의 바이트 수를 모를 때 유용합니다. *Read*는 실제로 전송된 바이트 수를 반환하며 스트림이 현재 위치에 전달된 *Count* 바이트를 모를 때 *Count*보다 적을 수 있습니다.

Write 메소드는 현재 *Position*에서 시작하여 버퍼로부터 스트림에 *Count* 바이트를 작성합니다. 다음은 *Write*에 대한 프로토타입입니다.

```
virtual int __fastcall Write(const void *Buffer, int Count);
```

파일에 기록한 후, *Write*는 현재 위치를 기록된 바이트 수만큼 앞으로 나가게 하고, 실제로 기록된 바이트 수를 반환하는데, 버퍼의 끝에 도달하거나 스트림이 더 이상의 바이트를 수용할 수 없으면 *Count*보다 작아집니다.

이에 대응되는 프로시저는 *ReadBuffer*와 *WriteBuffer*이며 *Read*, *Write*와 달리 읽거나 기록된 바이트 숫자를 반환하지 않습니다. 이러한 프로시저는 구조체에서 읽을 때처럼 바이트 수가 알려져 있고 또 필요한 경우에 유용합니다. *ReadBuffer* 및 *WriteBuffer*는 바이트 계산이 정확하게 일치하지 않으면 *EReadError* 및 *EWriteError* 예외가 발생합니다. 요청된 값과 다른 바이트 계산을 반환하는 *Read* 및 *Write* 메소드와는 대조적입니다. 다음은 *ReadBuffer*와 *WriteBuffer*에 대한 프로토타입입니다.

```
virtual int __fastcall ReadBuffer(void *Buffer, int Count);
virtual int __fastcall WriteBuffer(const void *Buffer, int Count);
```

이러한 메소드들은 *Read* 메소드와 *Write* 메소드를 호출하여 실제 읽기와 쓰기를 수행합니다.

컴포넌트 읽기 및 쓰기

*TStream*은 컴포넌트를 읽고 쓰기 위해 특화된 메소드, 즉 *ReadComponent* 및 *WriteComponent*를 정의합니다. 개발자의 애플리케이션 내에서 이러한 메소드를 사용하여 런타임시 해당 메소드를 생성하거나 변경할 때 컴포넌트 및 그 속성을 저장할 수 있습니다.

ReadComponent 및 *WriteComponent*는 IDE가 파일에서 컴포넌트를 읽거나 파일에 컴포넌트를 기록하기 위해 사용하는 메소드입니다. 컴포넌트를 파일에 스트리밍하는 경우 스트림 클래스가 *TFiler* 클래스, *TReader* 및 *TWriter*를 사용하여 파일로부터 객체를 읽거나 디스크에 객체를 기록합니다. 컴포넌트 스트리밍 시스템 사용에 대한 자세한 내용은 *TStream*, *TFiler*, *TReader*, *TWriter* 및 *TComponent* 클래스의 온라인 도움말을 참조하십시오.

한 스트림에서 다른 스트림으로 데이터 복사

한 스트림에서 다른 스트림으로 데이터를 복사하는 경우 명시적으로 데이터를 읽은 다음 기록하지 않아도 됩니다. 그 대신 다음 예제에서 보듯이 *CopyFrom* 메소드를 사용하면 됩니다.

애플리케이션에는 두 가지 편집 컨트롤(From 및 To)과 Copy File 버튼이 있습니다.

```
void __fastcall TForm1::CopyFileClick(TObject *Sender)
{
  TStream* stream1=TFileStream::Create(From.Text,fmOpenRead |
fmShareDenyWrite);
  try
  {
    TStream* stream2 -> TFileStream::Create(To.Text fmOpenCreate |
fmShareDenyRead);
    try
    {
      stream2 -> CopyFrom(stream1, stream1->Size);
    }
    __finally
    {
      delete stream2;
    }
  }
  __finally
  {
    delete stream1;
  }
}
```

스트림 위치 및 크기 지정

읽기 및 쓰기용 메소드 외에 스트림을 사용하여 애플리케이션이 스트림에서 임의의 위치를 찾거나 스트림의 크기를 변경할 수 있도록 허용할 수 있습니다. 일단 지정된 위치를 찾으면 다음 읽기 또는 쓰기 작업이 해당 위치로부터 스트림을 읽거나 해당 위치에 스트림을 기록합니다.

특정 위치 찾기

Seek 메소드는 스트림에서 특정 위치로 이동할 수 있는 가장 일반적인 메커니즘입니다. *Seek* 메소드에는 두 개의 오버로드가 있습니다.

```
virtual int __fastcall Seek(int Offset, Word Origin);
virtual __int64 __fastcall Seek(const __int64 Offset, TSeekOrigin
Origin);
```

두 오버로드는 동일한 방법으로 작업합니다. 차이점은 한 버전은 32 비트 정수를 사용하여 위치 및 오프셋을 나타내는 반면 다른 버전은 64 비트 정수를 사용한다는 점입니다.

Origin 매개변수는 *Offset* 매개변수를 해석하는 방법을 나타냅니다. *Origin* 매개변수는 다음 값 중의 하나가 되어야 합니다.

값	의미
soFromBeginning	리소스의 시작 지점에서 Offset이 시작합니다. Seek는 Offset 위치로 이동합니다. Offset은 >= 0이어야 합니다.
soFromCurrent	리소스의 현재 위치에서 Offset이 시작합니다. Seek는 Position + Offset으로 이동합니다.
soFromEnd	리소스의 끝에서 Offset이 시작합니다. 파일 끝에서부터의 바이트 수를 나타내려면 Offset은 <= 0이어야 합니다.

*Seek*는 스트림의 현재 위치를 재설정하여 표시된 오프셋만큼 이동합니다. *Seek*는 스트림에서의 새 현재 위치를 반환합니다.

Position 및 Size 속성 사용

모든 스트림에는 스트림의 현재 위치와 크기를 가지는 속성이 있습니다. 이러한 속성은 스트림을 읽거나 스트림에 기록하는 메소드와 마찬가지로 *Seek* 메소드에 의해 사용됩니다.

Position 속성은 스트림되는 데이터의 시작으로부터 스트림으로 현재 오프셋을 바이트 단위로 나타냅니다.

Size 속성은 스트림의 크기를 바이트 단위로 표시합니다. 이러한 표시로 인해 읽기에 사용할 수 있는 바이트 수를 결정하거나 스트림의 데이터를 자를 수 있습니다.

*Size*는 스트림에 읽고 쓰는 루틴에 의해서 내부적으로 사용됩니다.

Size 속성을 설정하면 스트림 내의 데이터 크기가 변경됩니다. 예를 들어, 파일 스트림에서 *Size*를 설정하면 파일 끝을 나타내는 표식을 삽입하여 파일을 자릅니다. 스트림의 *Size*가 변경되지 않으면 예외가 발생합니다. 예를 들어, 읽기 전용 파일 스트림의 *Size*를 변경하려고 시도하면 예외가 발생합니다.

파일 작업

BaseCLX는 여러 가지 방법으로 파일 작업을 지원합니다. 파일 스트림을 사용하는 것 외에 파일 I/O 수행에는 여러 가지 런타임 라이브러리 루틴이 있습니다. 파일로부터 읽고 파일에 기록하는 데 필요한 파일 스트림과 전역 루틴 모두 4-5페이지의 "파일 I/O에 대한 접근 방법"에서 설명하고 있습니다.

입/출력 작업 외에 디스크 상의 파일을 조작해야 하는 경우도 있을 것입니다. 파일의 내용이 아니라 파일 상의 작업 자체에 대한 지원은 4-7페이지의 "파일 처리"에서 설명합니다.

참고 크로스 플랫폼 애플리케이션에서 CLX를 사용하는 경우 오브젝트 파스칼 랭귀지는 대소문자를 구별하지 않지만 Linux 운영 체제는 구별합니다. 파일을 사용하는 객체 및 루틴을 사용하는 경우 파일 이름의 대소문자를 구별하십시오.

파일 I/O에 대한 접근 방법

파일로부터 읽거나 파일에 기록하는 경우 사용할 수 있는 방법은 세 가지가 있습니다.

- 파일 사용에 대해 권장하는 방법은 파일 스트림을 사용하는 것입니다. 파일 스트림은 디스크 파일의 정보를 액세스하는 데 사용되는 *TFileStream* 클래스의 객체 인스턴스입니다. 파일 스트림은 파일 핸들을 사용할 수 있게 하고 다른 방법과 함께 사용할 수 있으므로 이식 가능한 높은 수준의 파일 I/O 접근 방법입니다. 다음 "파일 스트림 사용" 단원에서는 *TFileStream*에 대해 자세히 설명합니다.
- 핸들에 기반을 둔 접근 방법으로 파일을 사용할 수 있습니다. 파일 핸들은 개발자가 파일을 생성하거나 열어서 컨텐트를 사용하는 경우 운영 체제에 의해 제공됩니다. SysUtils 유닛은 파일 핸들을 사용하여 파일로 작업하는 파일 처리 루틴의 수를 정의합니다. Windows에서는 이러한 방법이 Windows API 함수에 관한 일반적인 랩퍼입니다. Delphi 함수는 오브젝트 파스칼 구문을 사용하고 종종 디폴트 매개변수 값을 제공하므로 Windows API 에 대한 편리한 인터페이스입니다. 더 나아가 Linux에 해당되는 버전이 있으므로 크로스 플랫폼 애플리케이션에서 이러한 루틴을 사용할 수 있습니다. 핸들에 기반을 둔 접근 방법을 사용하려면 먼저 *FileOpen* 함수를 사용하여 파일을 열거나 *FileCreate* 함수를 사용하여 새 파일을 작성해야 합니다. 일단 핸들이 있으면 핸들에 기반한 루틴을 사용하여 명령줄 쓰기, 텍스트 읽기 등의 작업을 수행할 수 있습니다.
- C 런타임 라이브러리 및 표준 C++ 라이브러리에는 파일 사용에 필요한 다양한 함수와 클래스가 포함되어 있습니다. 따라서 VCL 또는 CLX를 사용하지 않는 애플리케이션에서 사용할 수 있다는 장점이 있습니다. 이러한 함수에 대한 자세한 내용은 C 런타임 라이브러리 또는 표준 C++ 라이브러리에 대한 문서를 참조하십시오.

파일 스트림 사용

*TFileStream*은 애플리케이션에서 디스크 상의 파일을 읽고 쓸 수 있게 하는 클래스입니다. 이것은 파일 스트림의 수준 높은 객체 표현에 사용됩니다. *TFileStream*은 스트림 객체이므로 공통 스트림 메소드를 공유합니다. 이러한 메소드를 사용하여 파일을 읽고 쓰거나 다른 스트림 클래스로부터 데이터를 복사하거나 다른 스트림 클래스에 데이터를 복사하거나 컴포넌트 값을 읽고 쓸 수 있습니다. 파일 스트림이 스트림 클래스가 됨으로써 상속할 수 있는 기능에 대한 자세한 내용은 4-2페이지의 "스트림 사용"을 참조하십시오.

또한 파일 스트림을 사용하면 파일 핸들에 액세스할 수 있으므로 파일 핸들이 필요한 루틴을 처리하는 전역 파일과 함께 사용할 수 있습니다.

파일 스트림을 사용하여 파일 생성 및 열기

파일을 생성하거나 열고 파일 핸들에 대한 액세스 권한을 얻기 위해서는 단순히 *TFileStream*을 인스턴스화하면 됩니다. 이렇게 하면 명명된 파일을 열거나 작성하며 파일을 읽고 쓸 수 있는 메소드를 제공할 수 있습니다. 파일이 열리지 않으면, *TFileStream* 생성자가 예외를 발생시킵니다.

```
__fastcall TFileStream(const AnsiString FileName, Word Mode);
```

Mode 매개변수는 파일 스트림을 생성할 때 파일을 여는 방법을 지정합니다. *Mode* 매개변수는 OR 연산된 개방형 모드와 공유 모드로 구성됩니다. 개방형 모드는 다음 값 중의 하나여야 합니다.

표 4.1 개방형 모드

값	의미
fmCreate	주어진 이름으로 파일을 TfileStream합니다. 주어진 이름을 가진 파일이 있을 경우 쓰기 모드에서 파일을 엽니다.
fmOpenRead	읽기 전용으로 파일을 엽니다.
fmOpenWrite	쓰기 전용으로 파일을 엽니다. 파일에 쓰는 내용은 현재 내용을 완전히 대체합니다.
fmOpenReadWrite	현재 내용을 대체하는 대신 수정하기 위해 파일을 엽니다.

공유 모드는 아래에 나열된 제한 사항을 가진 다음 값 중의 하나가 될 수 있습니다.

표 4.2 공유 모드

값	의미
fmShareCompat	FCB가 열리는 방법과 공유가 호환됩니다.
fmShareExclusive	다른 애플리케이션에서 파일을 절대 열 수 없습니다.
fmShareDenyWrite	다른 애플리케이션에서 읽기 위해 파일을 열 수는 있지만 쓰기 위해 열 수는 없습니다.
fmShareDenyRead	다른 애플리케이션에서 쓰기 위해 파일을 열 수는 있지만 읽기 위해 열 수는 없습니다.
fmShareDenyNone	다른 애플리케이션에서 파일 읽기나 쓰기가 금지되어 있습니다.

사용할 수 있는 공유 모드는 사용했던 개방형 모드에 따라 다르다는 것에 유의하십시오. 다음 표는 각 개방형 모드에서 사용할 수 있는 공유 모드를 나타냅니다.

표 4.3 각 개방형 모드에서 사용할 수 있는 공유 모드

개방형 모드	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	사용할 수 없음	사용할 수 없음	사용 가능	사용할 수 없음	사용 가능
fmOpenWrite	사용 가능	사용 가능	사용할 수 없음	사용 가능	사용 가능
fmOpenReadWrite	사용 가능	사용 가능	사용 가능	사용 가능	사용 가능

파일 개방형 모드 및 공유 모드 상수는 SysUtils 유닛에 정의되어 있습니다.

파일 핸들 사용

*TFileStream*을 인스턴스화하면 파일 핸들에 액세스할 수 있습니다. 파일 핸들은 *Handle* 속성에 포함되어 있습니다. Windows에서 *Handle*은 Windows 파일 핸들입니다. CLX Linux 버전에서는 Linux 파일 핸들입니다. *Handle*은 읽기 전용이며 파일이 열린 모드를 나타냅니다. 파일 *Handle*의 어트리뷰트(attribute)를 변경하려면 새로운 파일 스트림 객체를 만들어야 합니다.

일부 파일 처리 루틴은 파일 핸들을 매개변수로 사용합니다. 일단 파일 스트림을 갖게 되면 파일 핸들을 사용할 어떠한 상황에서도 *Handle* 속성을 사용할 수 있습니다. 파일 스트림은 핸들 스트림과 달리 객체가 소멸될 때 파일 핸들을 닫는다는 점에 유의하십시오.

파일 처리

몇몇 공통된 파일 작업은 BaseCLX 런타임 라이브러리로 생성됩니다. 파일 작업에 관한 프로시저와 함수는 수준 높은 작업을 수행합니다. 대부분의 루틴에서는 파일 이름을 지정하면 루틴이 개발자 대신 운영 체제에 필요한 호출을 합니다. 그 대신 파일 핸들을 사용하는 경우도 있습니다.

주의 오브젝트 파스칼 랭귀지는 대소문자를 구별하지 않지만 Linux 운영 체제는 구별합니다. 크로스 플랫폼 애플리케이션에서 파일을 사용하여 작업하는 경우 대소문자에 유의하십시오.

파일 삭제

파일 삭제는 디스크에서 파일을 지우고 디스크의 디렉토리에서 항목을 제거합니다. 삭제된 파일을 복구하는 작업은 없으므로 일반적으로 사용자가 애플리케이션에서 파일의 삭제를 확인할 수 있습니다. 파일을 삭제하려면, 다음과 같이 파일 이름을 *DeleteFile* 함수로 전달합니다.

```
DeleteFile(FileName);
```

*DeleteFile*은 파일을 삭제한 경우 *true*를 반환하며, 삭제하지 않은 경우(예: 파일이 없거나 읽기 전용인 경우)에는 *false*를 반환합니다. *DeleteFile*은 *FileName*에 의해 명명된 파일을 디스크에서 지웁니다.

파일 찾기

파일 찾기에 사용되는 루틴은 *FindFirst*, *FindNext* 및 *FindClose* 등 세 가지입니다. *FindFirst*는 지정된 디렉토리에서 주어진 어트리뷰트(attribute) 집합을 사용하여 파일 이름의 첫 번째 인스턴스를 찾습니다. *FindNext*는 이전 *FindFirst* 호출에서 지정된 이름과 어트리뷰트에 맞는 다음 항목을 반환합니다. *FindClose*는 *FindFirst*에 의해 할당된 메모리를 해제합니다. 항상 *FindClose*를 사용하여 *FindFirst*/*FindNext*를 종료해야 합니다. 파일이 존재하는지 알고 싶은 경우에는 *FileExists* 함수를 사용하십시오. 이 함수는 파일이 있으면 *true*를 반환하고 파일이 없으면 *false*를 반환합니다.

세 가지 파일 찾기 루틴은 *TSearchRec*을 매개변수 중의 하나로 사용합니다. *TSearchRec*은 *FindFirst* 또는 *FindNext*로 찾아낸 파일 정보를 정의합니다. 다음은 *TSearchRec*에 대한 선언입니다.

```
struct TSearchRec
{
  int Time; // time stamp of the file
  int Size; // size of the file in bytes
  int Attr; // file attribute flags
  AnsiString Name; // filename and extension
```

```
    int ExcludeAttr; // file attribute flags for files to ignore
    unsigned FindHandle;
    _WIN32_FIND_DATAA FindData; // structure with addition information
} ;
```

파일을 찾으면, *TSearchRec* 타입의 매개변수 필드는 찾아낸 파일을 설명하도록 수정됩니다. 파일에 특정한 어트리뷰트(attribute)가 있는지 확인하기 위해 다음 어트리뷰트 상수 또는 값에 대한 *Attr*을 검사할 수 있습니다.

표 4.4 어트리뷰트 상수 및 값

상수	값	설명
faReadOnly	0x00000001	읽기 전용 파일
faHidden	0x00000002	숨김 파일
faSysFile	0x00000004	시스템 파일
faVolumeID	0x00000008	볼륨 ID 파일
faDirectory	0x00000010	디렉토리 파일
faArchive	0x00000020	아카이브 파일
faAnyFile	0x0000003F	모든 파일

어트리뷰트를 테스트하려면 **&** 연산자를 사용하여 *Attr* 필드의 값을 어트리뷰트 상수와 조합하십시오. 파일에 해당 어트리뷰트가 있으면 결과가 0보다 커집니다. 예를 들어, 찾은 파일이 히든 파일이면 *SearchRec.Attr* & *faHidden* > 0 표현식이 결과적으로 *true*로 분석됩니다. 어트리뷰트는 해당 상수나 값을 OR 연산하여 결합할 수 있습니다. 예를 들면, 일반적인 파일 이외에 읽기 전용 파일과 히든 파일을 찾으려는 경우 *faReadOnly* | *faHidden*을 *Attr* 매개변수로 전달하십시오.

예제 다음 예제는 폼에서 레이블과 *Search*라는 버튼, *Again*이라는 버튼을 사용합니다. 사용자가 *Search* 버튼을 누르면, 지정된 경로에 있는 첫 번째 파일이 발견되고 파일의 이름과 바이트 수가 레이블의 캡션에 나타납니다. 사용자가 *Again* 버튼을 누를 때마다 그 다음으로 일치하는 파일 이름과 크기가 레이블에 나타납니다.

```
  TSearchRec SearchRec; // global variable
void __fastcall TForm1::SearchClick(TObject *Sender)
{
  FindFirst("c:\\Program Files\\bcb6\\bin\\*.*", faAnyFile, SearchRec);
  Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) +
" bytes in size";
}
void __fastcall TForm1::AgainClick(TObject *Sender)
{
  if (FindNext(SearchRec) == 0)
    Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size)
+ " bytes in size";
  else
    FindClose(SearchRec);
}
```

참고 크로스 플랫폼 애플리케이션에서 하드 코딩된 경로 이름을 나타내려면 이러한 경로 이름을 모두 올바른 시스템 경로 이름으로 바꾸거나 Tools|Environment Options를 선택한 다음 Environment Variables 페이지에서 환경 변수를 사용해야 합니다.

파일 이름 변경

파일 이름을 변경하려면 *RenameFile* 함수를 사용하십시오.

```
extern PACKAGE bool __fastcall RenameFile(const AnsiString OldName, const
AnsiString NewName);
```

여기서, *RenameFile*은 *OldFileName*으로 식별한 파일 이름을 *NewFileName*에서 지정한 파일 이름으로 변경합니다. 이 작업이 성공하면 *RenameFile*은 *true*를 반환합니다. 만일 파일 이름을 바꾸지 못하면, 가령 *NewFileName*이라는 파일이 이미 있을 경우에는 *RenameFile*이 *false*를 반환합니다. 예를 들면, 다음과 같습니다.

```
if (!RenameFile("OLDNAME.TXT","NEWNAME.TXT"))
  ErrorMsg("Error renaming file!");
```

여기서 *RenameFile*을 사용하여 다른 드라이브로 파일 이름을 바꿀(이동) 수는 없습니다. 먼저 파일을 복사한 다음 이전 파일을 삭제해야 합니다.

참고 BaseCLX 런타임 라이브러리의 *RenameFile*이 Windows API *MoveFile* 함수의 랩퍼로 *MoveFile* 함수도 다른 드라이버에서는 사용할 수 없습니다.

파일의 date-time 루틴

FileAge, *FileGetDate*, *FileSetDate* 루틴은 운영 체제 date-time 값을 조작합니다. *FileAge*는 파일의 date-and-time 스탬프를 반환하거나, 파일이 없으면 -1을 반환합니다. *FileSetDate*는 지정된 파일의 date-and-time 스탬프를 설정한 다음, 성공하면 0을 반환하고 실패하면 오류 코드를 반환합니다. *FileGetDate*는 지정된 파일의 date-and-time 스탬프를 반환하거나 핸들이 유효하지 않으면 -1을 반환합니다.

대부분의 파일 처리 루틴처럼, *FileAge*는 문자열 파일 이름을 사용합니다. 그러나 *FileGetDate* 및 *FileSetDate*는 파일 핸들을 사용하는 정수 매개변수를 사용합니다. 파일 핸들을 가져오려면 다음 방법 중 하나를 수행하십시오.

- *FileOpen* 또는 *FileCreate* 함수를 사용하여 새 파일을 작성하거나 기존 파일을 엽니다. *FileOpen* 및 *FileCreate* 모두 파일 핸들을 반환합니다.
- *TFileStream*을 인스턴스화하여 파일을 작성하거나 엽니다. 그런 다음 *Handle* 속성을 사용합니다. 자세한 내용은 4-5페이지의 "파일 스트림 사용"을 참조하십시오.

파일 복사

BaseCLX 런타임 라이브러리는 파일 복사에 필요한 루틴을 제공하지 않습니다. 그러나 Windows 전용 애플리케이션을 작성하는 경우에는 직접 Windows API *CopyFile* 함수를 호출하여 파일을 복사할 수 있습니다. 대부분의 런타임 라이브러리 파일 루틴과 같이 *CopyFile*은 파일 핸들이 아니라 파일 이름을 매개변수로 사용합니다. 파일을 복사하는 경우 기존 파일에 대한 파일 어트리뷰트(attribute)가 새 파일에 복사되나 보안 어트리뷰트는 복사되지 않는다는 점에 유의하십시오. 또한 *RenameFile* 함수와 Windows API *MoveFile* 함수 모두 파일 이름을 변경하거나 다른 드라이브로 옮길 수 없으므로 파일을 다른 드라이브로 옮기는 경우에도 *CopyFile*이 유용합니다. 자세한 내용은 Microsoft Windows 온라인 도움말을 참조하십시오.

ini 파일 및 시스템 레지스트리 작업

많은 애플리케이션이 ini 파일을 사용하여 구성 정보를 저장합니다. BaseCLX에는 ini 파일을 사용하는 데 필요한 두 개의 클래스, 즉 *TIniFile* 및 *TMemIniFile*이 포함되어 있습니다. ini 파일을 사용하면 크로스 플랫폼 애플리케이션에서도 사용할 수 있으며 읽기와 편집이 쉽다는 이점이 있습니다. 이러한 클래스에 대한 자세한 내용은 4-10페이지의 "TIniFile 및 TMemIniFile 사용"을 참조하십시오.

많은 Windows 애플리케이션에서는 ini 파일 대신 시스템 레지스트리를 사용합니다. Windows 시스템 레지스트리는 구성 정보에 필요한 중앙 저장 공간 역할을 하는 계층적 데이터베이스입니다. VCL에는 시스템 레지스트리 사용에 필요한 클래스가 포함되어 있습니다. 이러한 클래스는 Windows에서 사용할 수 없으므로 기술적으로는 BaseCLX의 일부가 아니지만 *TRegistryIniFile* 및 *TRegistry*라는 두 개의 클래스는 ini 파일을 사용하는 클래스와 유사하므로 이에 대한 설명을 제공합니다.

*TRegistryIniFile*은 ini 파일을 사용하는 클래스와 공통적인 조상(*TCustomIniFile*)을 공유하므로 크로스 플랫폼 애플리케이션에 유용합니다. 공통 조상(*TCustomIniFile*)의 메소드에 한정하면 애플리케이션이 최소한의 조건부 코드가 있는 양 애플리케이션에서 작업할 수 있습니다. *TRegistryIniFile*에 대해서는 4-11페이지의 "TRegistryIniFile 사용"에서 설명합니다.

여러 플랫폼에서 사용할 수 없는 애플리케이션의 경우, *TRegistry* 클래스를 사용할 수 있습니다. 시스템 레지스트리는 ini 파일에 대한 클래스와 호환될 필요가 없으므로 *TRegistry*의 속성 및 메소드는 시스템 레지스트리가 구성된 방법에 보다 직접적으로 해당되는 이름을 갖습니다. *TRegistry*에 대해서는 4-12페이지의 "TRegistry 사용"에서 설명합니다.

TIniFile 및 TMemIniFile 사용

ini 파일 형식은 지금도 널리 사용되고 있으며, DSK Desktop 설정 파일 등 많은 설정 파일이 이 형식을 사용합니다. 이 형식은 특히 구성 정보를 저장하기 위해 항상 시스템 레지스트리를 계산할 수는 없는 크로스 플랫폼 애플리케이션에서 유용합니다. BaseCLX는 아주 쉽게 ini 파일을 읽고 쓸 수 있는 *TIniFile* 및 *TMemIniFile*이라는 두 개의 클래스를 제공합니다.

Linux에서는 *TMemIniFile*과 *TIniFile*이 동일합니다. Windows에서는 *TMemIniFile* 버퍼가 메모리를 모두 변경하고 개발자가 *UpdateFile* 메소드를 호출하기 전까지 이를 디스크에 기록하지 않는 반면 *TIniFile*은 디스크에서 ini 파일을 직접 사용합니다.

TIniFile 또는 *TMemIniFile* 객체를 인스턴스화하는 경우 ini 파일의 이름을 매개변수로 생성자에 전달합니다. 이 때 파일이 없으면 자동으로 생성됩니다. 그러면 *ReadString*, *ReadDate*, *ReadInteger* 또는 *ReadBool* 등의 다양한 읽기 메소드를 사용하여 자유롭게 값을 읽을 수 있습니다. 또는 ini 파일의 전체 섹션을 읽으려면 *ReadSection* 메소드를 사용할 수 있습니다. 이와 유사하게 *WriteBool*, *WriteInteger*, *WriteDate* 또는 *WriteString* 등의 메소드를 사용하여 값을 쓸 수 있습니다.

다음은 폼의 생성자 및 *OnClose* 이벤트 핸들러의 쓰기 값에 있는 ini 파일로부터 구성 정보를 읽는 예제입니다.

```
__fastcall TForm1::TForm1(TComponent *Owner) : TForm(Owner)
{
   TIniFile *ini;
   ini = new TIniFile( ChangeFileExt( Application->ExeName, ".INI" ) );
```

```
    Top     =  ini->ReadInteger( "Form", "Top", 100 );
    Left    =  ini->ReadInteger( "Form", "Left", 100 );
    Caption =  ini->ReadString( "Form", "Caption",
                                "Default Caption" );
    ini->ReadBool( "Form", "InitMax", false ) ?
          WindowState = wsMaximized :
          WindowState = wsNormal;

    delete ini;
}

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
    TIniFile *ini;
    ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
    ini->WriteInteger( "Form", "Top", Top );
    ini->WriteInteger( "Form", "Left", Left );
    ini->WriteString ( "Form", "Caption", Caption );
    ini->WriteBool   ( "Form", "InitMax",
                       WindowState == wsMaximized );

    delete ini;
}
```

각 Read 루틴은 세 개의 매개변수를 사용합니다. 첫 번째 매개변수는 ini 파일의 섹션을 식별하고 두 번째 매개변수는 개발자가 읽으려는 값을 식별하며 세 번째 매개변수는 섹션이나 값이 ini 파일에 없는 경우의 기본값입니다. 섹션이나 값이 없을 때 Read 메소드가 상황을 처리하듯이 Write 루틴 또한 섹션이나 값이 없을 때 섹션이나 값을 생성합니다. 다음은 ini 파일을 만들 때 처음 실행되는 코드 예제입니다.

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

이 애플리케이션의 이후 실행에서는 폼이 생성될 때 ini 값을 읽고 다시 *OnClose* 이벤트에 기록합니다.

TRegistryIniFile 사용

레지스트리는 계층적이며 ini 파일의 크기 제한이 없으므로 많은 32 비트 Windows 애플리케이션이 정보를 ini 파일 대신 시스템 레지스트리에 저장합니다. ini 파일 사용에 익숙하며 구성 정보를 ini 파일 대신 레지스트리로 옮기려면 *TRegistryIniFile* 클래스를 사용하면 됩니다. 또한 Windows에서 시스템 레지스트리를 사용하고 Linux에서 ini 파일을 사용하려면 크로스 플랫폼 애플리케이션에서 *TRegistryIniFile*을 사용하면 됩니다. 대부분의 애플리케이션을 작성할 수 있으므로 *TCustomIniFile* 타입을 사용합니다. 개발자는 Windows에서 *TRegistryIniFile*의 인스턴스를 만들거나 Linux에서 *TMemIniFile*을 만들고, 이 인스턴스를 애플리케이션이 사용하는 *TCustomIniFile*에 할당하는 코드를 조건적으로 지정하기만 하면 됩니다.

*TRegistryIniFile*은 ini 파일 항목처럼 보이는 레지스트리 항목을 만듭니다. *TIniFile* 및 *TMemIniFile*(읽기 및 쓰기)의 모든 메소드는 *TRegistryIniFile*에 있습니다.

TRegistryIniFile 객체를 생성하는 경우 *IniFile* 또는 *TMemIniFile* 객체의 파일 이름에 해당되는 생성자에 전달하는 매개변수가 레지스트리 내의 사용자 키에서 키 값이 됩니다. 모든 섹션 및 값은 이 루트에서 파생됩니다. *TRegistryIniFile*은 레지스트리 인터페이스를 크게 단순화하므로 기존 코드를 이식하거나 크로스 플랫폼 애플리케이션을 작성하는 경우가 아니더라도 *TRegistry* 컴포넌트 대신 이를 사용하고자 하는 경우가 있습니다.

TRegistry 사용

Windows 전용 애플리케이션을 작성하려는 경우이며 시스템 레지스트리의 구조에 익숙하다면 *TRegistry*를 사용할 수 있습니다. 다른 ini 파일 컴포넌트와 동일한 속성 및 메소드를 사용하는 *TRegistryIniFile*과 달리 *TRegistry*의 속성 및 메소드는 시스템 레지스트리의 구조에 보다 직접적으로 해당됩니다. 예를 들어, *TRegistry*가 HKEY_CURRENT_USER를 루트 키로 가정하는 데 비해 *TRegistry*를 사용하면 루트 키와 하위 키를 지정할 수 있습니다. 열기, 닫기, 저장, 이동, 복사 및 삭제 키용 메소드 외에 *TRegistry*를 사용하면 원하는 액세스 레벨을 지정할 수 있습니다.

참고 *TRegistry*는 크로스 플랫폼 프로그래밍에 사용할 수 없습니다.

다음은 레지스트리 항목에서 값을 검색하는 예제입니다.

```
#include <Registry.hpp>

AnsiString GetRegistryValue(AnsiString KeyName)
{
  AnsiString S;
  TRegistry *Registry = new TRegistry(KEY_READ);
  try
  {
    Registry->RootKey = HKEY_LOCAL_MACHINE;
    // False because we do not want to create it if it doesn't exist
    Registry->OpenKey(KeyName,false);
    S = Registry->ReadString("VALUE1");
  }
  __finally
  {
    delete Registry;
  }
  return S;
}
```

리스트 작업

BaseCLX에는 리스트나 항목 컬렉션을 나타내는 다양한 클래스가 포함되어 있습니다. 이러한 클래스는 포함한 항목의 타입, 지원하는 작업 및 영구적인지 여부에 따라 매우 다양합니다.

다음은 다양한 리스트 클래스 및 해당 클래스에 포함된 항목 타입을 나타내는 표입니다.

표 4.5 리스트 관리용 클래스

객체	보유 리스트
TList	포인터 리스트
TThreadList	스레드 안전 포인터 리스트
TBucketList	해시된 포인터 리스트
TObjectBucketList	해시된 객체 인스턴스 리스트
TObjectList	메모리에서 관리하는 객체 인스턴스 리스트
TComponentList	컴포넌트의 메모리 관리 리스트(즉, *TComponent*의 자손 클래스의 인스턴스)
TClassList	클래스 참조(메타클래스) 리스트
TInterfaceList	인터페이스 포인터 리스트
TQueue	포인터의 선입선출(FIFO) 리스트
TStack	포인터의 후입선출(LIFO) 리스트
TObjectQueue	객체의 선입선출 리스트
TObjectStack	객체의 후입선출 리스트
TCollection	타입화된 항목의 많은 특화된 클래스에 대한 기본 클래스
TStringList	문자열 리스트
THashedStringList	성능을 위해 해시된, Name=Value 형식을 가진 문자열 리스트

일반적인 리스트 작업

다양한 리스트 클래스에는 다른 타입의 항목 및 조상이 있으나 대부분은 리스트 내의 항목 추가, 삭제, 재정렬 및 액세스에 대해 공통되는 메소드 집합을 공유합니다.

리스트 항목 추가

대부분의 리스트 클래스에는 *Add* 메소드가 있으며 이 메소드를 사용하여 정렬되지 않은 경우에는 항목을 리스트 끝에 추가하며 정렬된 경우에는 적절한 위치에 추가합니다. 일반적으로 *Add* 메소드는 개발자가 리스트에 추가하는 항목을 매개변수로 사용하며 항목이 추가된 리스트 내 위치를 반환합니다. 버킷 리스트(*TBucketList* 및 *TObjectBucketList*)에서는 Add가 추가할 항목뿐만 아니라 추가할 항목과 연결할 수 있는 데이터도 사용합니다. 컬렉션의 경우, *Add*는 매개변수를 사용하지 않으나 추가하는 새 항목을 생성합니다. 컬렉션 상의 *Add* 메소드는 추가된 항목을 반환하므로 새 항목의 속성에 값을 할당할 수 있습니다.

일부 리스트 클래스에는 *Add* 메소드 외에 *Insert* 메소드가 있습니다. Insert 메소드는 *Add* 메소드와 같은 방식으로 작동하나 추가 매개변수를 사용하여 새 항목을 표시할 리스트 상의 위치를 지정할 수 있습니다. 클래스에 Add 메소드가 있으면 항목의 위치가 미리 결정되지 않는 한 Insert 메소드도 있습니다. 예를 들어, 항목이 정렬 순서로 배열되어야 하므로 정렬된 리스트에서는 *Insert*를 사용할 수 없으며 버킷 리스트에서는 해시 알고리즘이 항목 위치를 결정하므로 *Insert*를 사용할 수 없습니다.

Add 메소드가 없는 유일한 클래스는 대기열과 스택 등 정렬된 리스트입니다. 정렬된 리스트에 항목을 추가하려면 Add 메소드 대신 *Push* 메소드를 사용하십시오. *Push*는 *Add*처럼 항목을 매개변수로 사용하여 올바른 위치에 삽입합니다.

리스트 항목 삭제

한 리스트 클래스에서 단일한 항목을 삭제하려면 *Delete* 메소드 또는 *Remove* 메소드 중 하나를 사용하십시오. *Delete*는 제거할 항목의 인덱스인 단일 매개변수를 사용합니다. *Remove*도 단일 매개변수를 사용하지만 이 매개변수는 제거할 항목의 인덱스가 아니라 항목에 대한 참조입니다. *Delete* 메소드나 *Remove* 메소드만 지원하는 리스트 클래스도 있고, 둘 다 지원하는 리스트 클래스도 있습니다.

항목 추가 작업에서와 마찬가지로 정렬된 리스트는 다른 리스트와 다르게 사용됩니다. 정렬된 리스트에서 항목을 제거하는 경우에는 *Delete* 또는 *Remove* 메소드를 사용하는 대신 *Pop* 메소드를 호출하여 정렬된 리스트에서 항목을 제거하십시오. *Pop*에는 제거할 수 있는 항목이 하나뿐이므로 인수를 사용하지 않습니다.

리스트에서 모든 항목을 삭제하려면 *Clear* 메소드를 호출하십시오. Clear 메소드는 정렬된 리스트 외의 모든 리스트에 대해 사용할 수 있습니다.

리스트 항목 액세스

TThreadList 및 정렬된 리스트를 제외한 모든 리스트 클래스는 개발자가 리스트 내의 항목에 액세스하는 데 필요한 속성을 갖고 있습니다. 일반적으로 이 속성을 *Items*이라고 합니다. 문자열 리스트의 경우, 이 속성을 *Strings*이라고 하며 버킷 리스트의 경우는 *Data*라고 합니다. *Items*, *Strings* 또는 *Data* 속성은 인덱싱된 속성이므로 엑세스하고자 하는 항목을 지정할 수 있습니다.

*TThreadList*에서는 반드시 항목에 액세스하기 전에 리스트를 잠궈야 합니다. 리스트를 잠그는 경우 *LockList* 메소드는 항목 액세스에 사용하는 *TList* 객체를 반환합니다.

정렬된 리스트에서는 리스트의 "맨 위" 항목에만 액세스할 수 있습니다. 개발자는 *Peek* 메소드를 호출하여 이 항목에 대한 참조를 얻을 수 있습니다.

리스트 항목 재정렬

일부 리스트에는 리스트 내의 항목을 재정렬할 수 있는 메소드가 포함되어 있습니다. 일부 리스트에는 두 항목의 위치를 바꾸는 *Exchange* 메소드가 있고 어떤 리스트에는 항목을 지정된 위치로 이동시키는 *Move* 메소드가 있으며 또 다른 리스트에는 리스트 내의 항목을 정렬하는 *Sort* 메소드가 있습니다.

어떤 메소드를 사용할 수 있는지 보려면 사용하고 있는 리스트 클래스의 온라인 도움말을 참조하십시오.

영구적(persistent) 리스트

영구적 리스트는 폼 파일에 저장할 수 있습니다. 따라서 컴포넌트에 있는 published 속성의 타입으로 사용됩니다. 디자인 시에 리스트에 항목을 추가할 수 있으며, 이러한 항목은 객체로 저장되어 항목을 사용하는 컴포넌트가 런타임 시 메모리로 로드될 때 해당 위치에 놓이게 됩니다. 영구적 리스트에는 두 가지 기본 타입, 즉 문자열 리스트와 컬렉션이 있습니다.

문자열 리스트의 예로는 *TStringList* 및 *THashedStringList*이 있습니다. 문자열 리스트는 이름이 암시하는 것처럼 문자열을 포함합니다. 또한 Name=Value 형식의 문자열에 대한 특별한 지원을 제공하여 개발자가 이름과 연결된 값을 찾을 수 있게 해줍니다. 대부분의 문자열 리스트를 통해 객체와 리스트 내의 각 문자열을 연결할 수도 있습니다. 문자열 리스트는 4-15페이지의 "문자열 리스트 작업"에 더 자세히 설명되어 있습니다.

컬렉션은 *TCollection* 클래스의 자손입니다. 각 *TCollection* 자손은 특정 항목 클래스를 관리하도록 특화되어 있습니다. 여기서 해당 클래스는 *TCollectionItem*의 자손입니다. 컬렉션은 많은 공통적인 리스트 작업을 지원합니다. 모든 컬렉션은 published 속성의 타입으로 디자인되며 대부분은 속성을 구현하기 위해 이를 사용하는 객체와 독립적으로 작동할 수 없습니다. 디자인 타임에서 값이 컬렉션인 속성은 컬렉션 에디터를 사용하여 항목을 추가, 제거 및 재정렬할 수 있습니다. 컬렉션 에디터는 컬렉션 구현에 필요한 공통 사용자 인터페이스를 제공합니다.

문자열 리스트 작업

가장 일반적으로 사용되는 리스트 타입 중 하나는 문자열 리스트입니다. 문자열 리스트의 예로는 콤보 박스의 항목, 메모의 줄, 글꼴 이름, 문자열 그리드의 열과 행 이름이 있습니다. BaseCLX는 *TStrings* 객체와 *TStringList* 및 *THashedStringList* 등의 이 객체의 자손을 통해 모든 문자열 리스트에 공통 인터페이스를 제공합니다. *TStringList*는 *TStrings*에 있는 추상 속성과 메소드를 구현하고, 속성, 이벤트 및 메소드를 사용하여 다음을 수행합니다.

- 리스트에서 문자열을 정렬합니다.
- 정렬된 리스트에서 문자열 중복을 금지합니다.
- 리스트 내용의 변경 사항에 응답합니다.

문자열 리스트를 보유하는 기능을 제공하는 것 외에도 이러한 객체들은 상호 운용성을 쉽게 합니다. 예를 들면, *TStrings*의 자손인 메모의 줄을 편집한 다음 이러한 줄을 *TStrings*의 자손인 콤보 박스의 항목으로 사용할 수 있습니다.

문자열 리스트 속성은 Value 열의 *TStrings*와 함께 Object Inspector에 나타납니다. *TStrings*를 더블 클릭하여 String List Editor를 연 다음, 줄을 편집, 추가 또는 삭제할 수 있습니다.

런타임에 문자열 리스트 객체를 사용하여 다음과 같은 작업을 수행할 수 있습니다.

- 문자열 리스트의 로드 및 저장
- 새 문자열 리스트 작성
- 리스트에서 문자열 처리
- 문자열 리스트에 객체 연결

문자열 리스트의 로드 및 저장

문자열 리스트 객체는 *SaveToFile* 및 *LoadFromFile* 메소드를 제공하므로 사용자는 문자열 리스트를 텍스트 파일에 저장하고 텍스트 파일을 문자열 리스트에 로드할 수 있습니다. 텍스트 파일의 각 줄은 리스트의 문자열에 해당합니다. 이러한 메소드를 사용하면, 파일을 메모 컴포넌트에 로드하여 단순한 텍스트 에디터를 만들거나 콤보 박스의 항목 리스트를 저장할 수 있습니다.

다음은 WIN.INI 파일의 복사본을 메모 필드에 로드하고 WIN.BAK이라는 백업 복사본을 만드는 예제입니다.

```
void __fastcall EditWinIni()
{
  AnsiString FileName = "C:\\WINDOWS\\WIN.INI";// set the file name
  Form1->Memo1->Lines->LoadFromFile(FileName);      // load from file
  Form1->Memo1->Lines->SaveToFile(ChangeFileExt(FileName, ".BAK"));  //
save to backup
}
```

새 문자열 리스트 작성

문자열 리스트는 일반적으로 컴포넌트의 일부입니다. 그러나, 예를 들어 조회 테이블용 문자열을 저장하는 것과 같이 독립적인 문자열 리스트를 만드는 것이 편리할 때가 있습니다. 문자열 리스트의 작성 및 관리 방법은 단일 루틴 내에서 생성, 사용 및 소멸되는 리스트의 기간이 짧은지 애플리케이션이 종료할 때까지 사용할 만큼 기간이 긴지에 따라 다릅니다. 생성한 문자열 리스트의 종류에 상관 없이 사용자는 종료 시 리스트를 해제해야 합니다.

기간이 짧은 문자열 리스트

문자열 리스트를 단일 루틴 기간 동안만 사용하는 경우 해당 문자열 리스트를 한 루틴 내에서 생성, 사용 및 제거할 수 있습니다. 문자열 리스트를 사용할 때는 이 방법이 가장 안전합니다. 문자열 리스트 객체는 리스트 자체와 리스트의 문자열에 대해 메모리를 할당하기 때문에 예외가 발생하더라도 **try...__finally** 블록을 사용하여 메모리가 해제되었는지 확인해야 합니다.

1 문자열 리스트 객체를 만듭니다.

2 **try...__finally** 블록의 **try** 부분에서 문자열 리스트를 사용합니다.

3 **__finally** 부분에서 문자열 리스트 객체를 해제합니다.

다음 이벤트 핸들러는 버튼 클릭에 응답하여 문자열 리스트를 생성, 사용 및 제거합니다.

```
void __fastcall TForm1::ButtonClick1(TObject *Sender)
{
   TStringList *TempList = new TStringList; // declare the list
   try{
      //use the string list
   }
   __finally{
    delete TempList; // destroy the list object
   }
}
```

기간이 긴 문자열 리스트

애플리케이션을 실행하는 동안 항상 문자열 리스트를 사용할 수 있게 하려면, 애플리케이션을 시작할 때 리스트를 생성하고 애플리케이션 종료 전에 제거하십시오.

1 애플리케이션의 메인 폼에 대한 유닛 파일에서 *TStrings* 타입의 필드를 폼의 선언에 추가합니다.

2 폼이 나타나기 전에 실행되는 메인 폼에 대한 *생성자*를 작성합니다. 생성자는 문자열 리스트를 만든 다음, 첫 번째 단계에서 선언한 필드에 할당해야 합니다.

3 폼의 *OnClose* 이벤트용 문자열 리스트를 해제하는 이벤트 핸들러를 작성합니다.

이 예제에서는 기간이 긴 문자열 리스트를 사용하여 메인 폼에서 사용자의 마우스 클릭을 기록한 다음 애플리케이션이 종료되기 전에 리스트를 파일에 저장합니다.

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
   ClickList = new TStringList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   ClickList->SaveToFile(ChangeFileExt(Application->ExeName, ".LOG"));//
Save the list
   delete ClickList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button,
      TShiftState Shift, int X,int Y)
{
   TVarRec v[] = {X,Y};
   ClickList->Add(Format("Click at (%d, %d)",v,ARRAYSIZE(v) - 1));//add a
string to the list }
```

리스트에서 문자열 처리

다음은 문자열 리스트에서 공통으로 수행되는 작업입니다.

- 리스트의 문자열 수 계산
- 특정 문자열 액세스
- 리스트에서 문자열 위치 찾기
- 리스트에서 문자열을 통한 반복
- 리스트에 문자열 추가
- 리스트 내에서 문자열 이동
- 리스트에서 문자열 삭제
- 문자열 리스트 전체 복사

리스트의 문자열 수 계산

읽기 전용 *Count* 속성은 리스트의 문자열 수를 반환합니다. 문자열 리스트는 인덱스가 0부터 시작하므로 *Count*는 마지막 문자열의 인덱스보다 하나 더 많습니다.

특정 문자열 액세스

Strings 배열 속성에는 0부터 시작하는 인덱스에 의해 참조되는 리스트의 문자열을 포함합니다.

```
StringList1->Strings[0] = “This is the first string.”;
```

문자열 리스트에서 항목 찾기

문자열 리스트에서 문자열을 찾으려면 *IndexOf* 메소드를 사용합니다. *IndexOf*는 전달된 매개변수와 일치하는 리스트의 첫 번째 문자열 인덱스를 반환하고, 일치하는 매개변수 문자열을 찾지 못하면 -1을 반환합니다. *IndexOf*는 정확하게 일치하는 문자열만을 찾습니다. 리스트에서 일치하는 부분 문자열을 찾으려면 문자열 리스트를 반복해야 합니다.

예를 들어, 다음과 같이 *IndexOf*를 사용하여 리스트 박스의 *Items* 중에서 해당 파일 이름을 찾았는지 여부를 확인할 수 있습니다.

```
if (FileListBox1->Items->IndexOf("WIN.INI") > -1) ...
```

리스트에서 문자열을 통한 반복

리스트의 문자열을 통해 반복하려면 0에서 *Count* – 1까지 실행되는 **for** 순환문을 사용합니다.

다음은 리스트 박스에 있는 각 문자열을 대문자로 변환하는 예제입니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for (int i = 0; i< ListBox1->Items->Count; i++)
    ListBox1->Items->Strings[i] =UpperCase(ListBox1->Items->Strings[i]);
}
```

리스트에 문자열 추가

문자열을 문자열 리스트의 끝에 추가하려면 *Add* 메소드를 호출하여 매개변수로 새 문자열을 전달하십시오. 문자열을 리스트에 삽입하려면 *Insert* 메소드를 호출하여 두 매개변수, 즉 문자열과 문자열을 두고자 하는 위치의 인덱스를 전달하십시오. 예를 들어, 문자열 "Three"를 리스트에서 세 번째 문자열로 만들려면 다음과 같이 합니다.

```
StringList1->Insert(2, "Three");
```

문자열을 한 리스트에서 다른 리스트로 추가하려면 *AddStrings*를 호출하십시오.

```
StringList1->AddStrings(StringList2);  // append the strings from
StringList2 to StringList1
```

리스트 내에서 문자열 이동

문자열 리스트에서 문자열을 이동하려면 *Move* 메소드를 호출하여 두 매개변수, 즉 문자열의 현재 인덱스와 문자열에 할당할 인덱스를 전달하십시오. 예를 들어, 리스트에서 세 번째 문자열을 다섯 번째 위치로 이동하려면 다음과 같이 합니다.

```
StringListObject->Move(2, 4);
```

리스트에서 문자열 삭제

문자열 리스트에서 문자열을 삭제하려면 리스트의 *Delete* 메소드를 호출하여 삭제하려는 문자열의 인덱스를 전달합니다. 삭제하려는 문자열의 인덱스를 모르는 경우 *IndexOf* 메소드를 사용하여 위치를 찾아냅니다. 문자열 리스트에서 모든 문자열을 삭제하려면 *Clear* 메소드를 사용하십시오.

다음은 *IndexOf*와 *Delete*를 사용하여 문자열을 찾아서 삭제하는 예제입니다.

```
int BIndex = ListBox1->Items->IndexOf("bureaucracy");
if (BIndex > -1)
  ListBox1->Items->Delete(BIndex);
```

문자열 리스트 전체 복사

Assign 메소드를 사용하여 소스 리스트에서 대상 리스트로 문자열을 복사할 수 있습니다. 이 메소드는 대상 리스트의 내용을 덮어 씁니다. 대상 리스트에 덮어 쓰지 않고 문자열을 추가하려면 *AddStrings*를 사용합니다. 예를 들면, 다음과 같습니다.

```
Memo1->Lines->Assign(ComboBox1->Item)s; //overwrites original strings
```

이 구문은 콤보 박스의 줄을 메모에 복사합니다(메모를 덮어씀). 반면에

```
Memo1->Lines->AddStrings(ComboBox1->Items);//appends strings to end
```

이 구문은 콤보 박스의 줄을 메모에 추가합니다.

문자열 리스트의 로컬 복사본을 만들 때 *Assign* 메소드를 사용합니다. 다음과 같이 한 문자열 리스트 변수를 다른 문자열 리스트 변수에 할당하는 경우

```
StringList1 = StringList2;
```

원래 문자열 리스트 객체를 잃게 되고 가끔 예상치 못한 결과가 발생합니다.

문자열 리스트에 객체 연결

문자열 리스트는 *Strings* 속성에 저장된 문자열 외에 객체에 대한 참조를 가질 수 있으며, 이를 *Objects* 속성에 저장합니다. *Strings*와 마찬가지로 *Objects* 속성은 인덱스가 0부터 시작하는 배열입니다. *Objects*는 비트맵을 owner-draw 컨트롤의 문자열에 연결할 때 가장 많이 사용됩니다.

문자열과 연결 객체를 리스트에 한 번에 추가하려면 *AddObject* 메소드 또는 *InsertObject* 메소드를 사용합니다. *IndexOfObject*는 지정된 객체에 연결된 리스트에서 첫 번째 문자열의 인덱스를 반환합니다. *Delete*, *Clear* 및 *Move*와 같은 메소드는 문자열과 객체 모두에서 작동합니다. 예를 들어, 문자열을 삭제하면 해당 객체가 있는 경우 이 객체가 제거됩니다.

기존 문자열에 객체를 연결하려면 동일한 인덱스에서 객체를 *Objects* 속성에 할당합니다. 해당 문자열을 추가하지 않으면 객체를 추가할 수 없습니다.

문자열 작업

BaseCLX 런타임 라이브러리는 어떤 문자열 타입에만 해당되는 많은 특화된 문자열 처리 루틴을 제공합니다. 와이드 문자열, AnsiStrings 및 Null 종료 문자열(char *)용 루틴들이 여기에 해당됩니다. Null 종료 문자열을 사용하는 루틴은 Null 종료를 사용하여 문자열의 길이를 결정합니다. 이후 단원들에서는 런타임 라이브러리 내의 다양한 문자열 처리 루틴에 대한 개요를 제공합니다.

와이드 문자 루틴

와이드 문자열은 여러 상황에서 사용됩니다. XML 등의 기술은 와이드 문자열을 원시 타입으로 사용합니다. 또한 여러 대상 로케일이 있는 애플리케이션에서 문자열 처리 관련 사항의 일부를 단순화하기 때문에 사용하기도 합니다. 와이드 문자 인코딩 구성을 사용하면 MBCS(멀티바이트 문자 집합) 시스템에서는 작동하지 않는 문자열에 대한 일반적인 가정을 만들 수 있다는 이점이 있습니다. 문자열의 바이트 수와 문자 수 사이에는 직접적인 관계가 있습니다. 따라서 문자가 반으로 잘리거나 문자의 뒷 부분을 다른 문자의 앞 부분으로 오인하는 실수에 대해서 걱정할 필요가 없습니다.

와이드 문자를 사용할 때의 단점은 대부분의 VCL 컨트롤이 문자열 값을 단일 바이트나 MBCS 문자열로 나타낸다는 점입니다(CLX 컨트롤은 일반적으로 와이드 문자열을 사용함). 와이드 문자 시스템과 MBCS 시스템 간에 변환을 수행하려면 매 번 문자열 속성을 설정하거나 그 값을 읽어야 하므로 엄청난 양의 추가적인 코드 작업이 필요하며 애플리케이션 속도가 느려집니다. 그러나 문자와 *WideChars* 간의 1:1 매핑을 사용해야 하는 일부 특수한 문자열 처리 알고리즘의 경우 와이드 문자열로 변환해야 하는 경우도 있습니다.

다음은 표준 단일 바이트 문자열(또는 MBCS 문자열)과 유니코드 문자열 간의 변환을 수행하는 함수입니다.

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

다음 함수도 WideStrings와 기타 표현 간의 변환을 수행하는 함수입니다.

- UCS4StringToWideString
- WideStringToUCS4String
- VarToWideStr
- VarToWideStrDef

다음 루틴은 WideStrings에 직접 사용됩니다.

- WideCompareStr
- WideCompareText
- WideSameStr
- WideSameText
- WideSameCaption(CLX만 해당)
- WideFmtStr
- WideFormat
- WideLowerCase
- WideUpperCase

마지막으로 일부 루틴에는 와이드 문자열 사용에 필요한 오버로드가 포함됩니다.

- UniqueString
- Length
- Trim
- TrimLeft
- TrimRight

AnsiStrings에 대해 일반적으로 사용되는 루틴

AnsiString 처리 루틴은 여러 함수 영역을 처리합니다. 이 영역 중 일부는 동일한 목적으로 사용되는데, 처리 루틴들 간의 차이점은 계산에서 특정 기준을 사용하는지 여부입니다. 다음 표에는 이러한 루틴이 함수 영역별로 나열되어 있습니다.

- 비교
- 대소문자 변환
- 수정
- 하위 문자열

해당되는 경우에 한해 테이블은 루틴이 다음 기준을 만족하는지 여부를 나타내는 열도 제공합니다.

- 대소문자 구별 사용: 로케일 설정이 사용되면 대소문자의 정의를 결정합니다. 루틴이 로케일 설정을 사용하지 않으면, 분석은 문자의 순서 값에 기반합니다. 루틴이 대소문자를 구별하지 않으면, 미리 정의된 패턴에 의해 결정된 대소문자를 논리적으로 병합합니다.
- 로케일 설정 사용: 로케일 설정을 사용하면 특정 로케일, 특히 아시아 랭귀지 환경에 대해 애플리케이션을 사용자 정의할 수 있습니다. 대부분의 로케일 설정에서는 소문자가 해당 대문자보다 값이 더 적다고 간주합니다. 이것은 소문자가 대문자보다 값이 더 큰 ASCII 순서와는 반대입니다. Windows 로케일을 사용하는 루틴은 일반적으로 Ansi로 시작합니다. 예를 들어, Ansi*XXX*로 사용합니다.
- MBCS(멀티바이트 문자 집합 지원): MBCS는 극동 아시아 로케일용 코드를 작성할 때 사용됩니다. 멀티바이트 문자는 1바이트와 2바이트 문자 코드의 혼합으로 표현되므로, 바이트 길이가 반드시 문자열의 길이에 해당되는 것은 아닙니다. MBCS를 지원하는 루틴은 1바이트 및 2바이트 문자를 분석합니다.

*ByteType*과 *StrByteType*은 특정 바이트가 2바이트의 선행 바이트인지 여부를 결정합니다. 멀티바이트 문자를 사용할 때 문자열의 2바이트 문자가 절반으로 잘리지 않도록 주의합니다. 문자 크기는 미리 결정할 수 없으므로 문자를 함수 또는 프로시저에 매개변수로 전달하지 마십시오. 대신 문자 또는 문자열에 대한 포인터를 전달합니다. MBCS에 대한 자세한 내용은 16장, "국제적인 애플리케이션 생성"에서 16-2페이지의 "애플리케이션 코드 활성화"를 참조하십시오.

표 4.6 문자열 비교 루틴

루틴	대소문자 구별	로케일 설정 사용	MBCS 지원
AnsiStrLComp	예	예	예
AnsiStrLastChar	아니오	예	예
AnsiStrIComp	아니오	예	예
AnsiMatchStr	예	예	예
AnsiMatchText	아니오	예	예
AnsiContainsStr	예	예	예
AnsiContainsText	아니오	예	예
AnsiStartsStr	예	예	예
AnsiStartsText	아니오	예	예
AnsiEndsStr	예	예	예
AnsiEndsText	아니오	예	예
AnsiIndexStr	예	예	예
AnsiIndexText	아니오	예	예
CompareStr	예	아니오	아니오
CompareText	아니오	아니오	아니오
AnsiResemblesText	아니오	아니오	아니오

표 4.7 대소문자 변환 루틴

루틴	로케일 설정 사용	MBCS 지원
WideString	예	예
AnsiLowerCaseFileName	예	예
AnsiUpperCaseFileName	예	예
AnsiUpperCase	예	예
LowerCase	아니오	아니오
UpperCase	아니오	아니오

참고 *AnsiCompareFileName*, *AnsiLowerCaseFileName*, *AnsiUpperCaseFileName* 등 문자열 파일 이름에 사용되는 루틴은 모두 Windows 로케일을 사용합니다. 파일 이름에 사용된 로케일(문자 집합)은 디폴트 사용자 인터페이스와 다를 수 있으므로 항상 이식할 수 있는 파일 이름을 사용해야 합니다.

표 4.8 문자열 수정 루틴

루틴	대소문자 구별	MBCS 지원
AdjustLineBreaks	해당 없음	예
AnsiQuotedStr	해당 없음	예
AnsiReplaceStr	예	예
AnsiReplaceText	아니오	예
StringReplace	플래그별 옵션	예
ReverseString	해당 없음	아니오
StuffString	해당 없음	아니오
Trim	해당 없음	예
TrimLeft	해당 없음	예
TrimRight	해당 없음	예
WrapText	해당 없음	예

표 4.9 하위 문자열 루틴

루틴	대소문자 구별	MBCS 지원
AnsiExtractQuotedStr	해당 없음	예
AnsiPos	예	예
IsDelimiter	예	예
IsPathDelimiter	예	예
LastDelimiter	예	예
LeftStr	해당 없음	아니오
RightStr	해당 없음	아니오
MidStr	해당 없음	아니오
QuotedStr	아니오	아니오

Null 종료 문자열에 일반적으로 사용되는 루틴

Null 종료 문자열 처리 루틴은 여러 함수 영역을 처리합니다. 이 영역 중 일부는 동일한 목적으로 사용되는데, 처리 루틴들 간의 차이점은 계산에서 특정 기준을 사용하는지 여부입니다. 다음에는 이러한 루틴이 함수 영역별로 나열되어 있습니다.

- 비교
- 대소문자 변환
- 수정
- 하위 문자열
- 복사

또한 이들 표에서는 해당되는 경우에 한해 루틴이 대소문자를 구별하는지, 현재 로케일을 사용하는지, MBCS를 지원하는지 여부도 표시합니다.

표 4.10 Null 종료 문자열 비교 루틴

루틴	대소문자 구별	로케일 설정 사용	MBCS 지원
AnsiStrComp	예	예	예
AnsiStrIComp	아니오	예	예
AnsiStrLComp	예	예	예
ByteType	아니오	예	예
StrComp	예	아니오	아니오
StrIComp	아니오	아니오	아니오
StrLComp	예	아니오	아니오
StrLIComp	아니오	아니오	아니오

표 4.11 Null 종료 문자열 대소문자 변환 루틴

루틴	로케일 설정 사용	MBCS 지원
AnsiStrLower	예	예
AnsiStrUpper	예	예
StrLower	아니오	아니오
StrUpper	아니오	아니오

표 4.12 문자열 수정 루틴

루틴
StrCat
StrLCat

표 4.13 하위 문자열 루틴

루틴	대소문자 구별	MBCS 지원
AnsiStrPos	예	예
AnsiStrScan	예	예
AnsiStrRScan	예	예
StrPos	예	아니오
StrScan	예	아니오
StrRScan	예	아니오

표 4.14 문자열 복사 루틴

루틴
StrCopy
StrLCopy
StrECopy
StrMove
StrPCopy
StrPLCopy

인쇄

엄밀하게 말하자면 두 개의 독립적인 버전이 있어서 하나는 Printers 유닛에서 VCL 용이며 다른 하나는 QPrinters 유닛에서 CLX용이므로 TPrinter 클래스는 BaseCLX에 속하지 않습니다. VCL *TPrinter* 객체는 Windows 프린터의 세부 사항을 캡슐화합니다. CLX *TPrinter* 객체는 프린터에 그리는 그리기 장치입니다. 이 장치는 포스트스크립트를 생성하여 lpr, lp 또는 다른 인쇄 명령으로 보냅니다. 그러나 *TPrinter*의 두 버전은 거의 유사합니다.

설치되어 있는 프린터 리스트를 보려면 *Printers* 속성을 사용하십시오. 두 프린터 객체는 모두 폼의 *TCanvas*와 동일한 *TCanvas*를 사용하므로 폼에 그릴 수 있는 모든 것을 인쇄할 수 있습니다. 이미지를 인쇄하려면 *BeginDoc* 메소드를 호출한 뒤에 인쇄하려는 캔버스 그래픽(*TextOut* 메소드를 사용한 텍스트 포함)을 지정하여 해당 메소드를 호출한 다음, *EndDoc* 메소드를 호출하여 인쇄할 작업을 프린터에 보내십시오.

이 예제에서는 폼에 버튼과 메모를 사용합니다. 사용자가 버튼을 누르면 메모의 내용이 인쇄되며 페이지의 가장자리에는 200픽셀의 테두리가 나타납니다.

이 예제를 성공적으로 실행하려면 유닛 파일에 <Printers.hpp>을 포함하십시오.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TPrinter *Prntr = Printer();
  TRect r = Rect(200,200,Prntr->PageWidth - 200,Prntr->PageHeight- 200);
  Prntr->BeginDoc();
  for( int i = 0; i< Memo1->Lines->Count; i++)
    Prntr->Canvas->TextOut(200,200 + (i *
  Prntr->Canvas->TextHeight(Memo1->Lines->Strings[i])),
  Memo1->Lines->Strings[i]);
  Prntr->Canvas->Brush->Color = clBlack;
  Prntr->Canvas->FrameRect(r);
  Prntr->EndDoc();
}
```

TPrinter 객체의 사용 방법에 대한 자세한 내용은 *TPrinter*의 온라인 도움말을 참조하십시오.

측정값 변환

ConvUtils 유닛은 한 단위 집합으로부터 다른 단위 집합으로 측정값을 변환하는 데 사용할 수 있는 일반적인 용도의 *Convert* 함수를 선언합니다. 피트와 인치 또는 일과 주 등 서로 호환되는 측정 단위 간에 변환을 수행할 수 있습니다. 동일한 타입의 대상을 측정하는 단위는 동일한 *변환 패밀리*에 속한다고 할 수 있습니다. 서로 변환하는 단위는 반드시 동일한 변환 패밀리에 속해야 합니다. 변환 수행에 대한 자세한 내용은 다음 단원인 변환 수행과 온라인 도움말의 *Convert*를 참조하십시오.

StdConvs 유닛은 몇몇 변환 패밀리 및 각 패밀리 내의 측정 단위를 정의합니다. 또한 *RegisterConversionType* 및 *RegisterConversionFamily* 함수를 사용하여 정의된 변환 패밀리 및 관련된 단위를 만들 수 있습니다. 변환 및 변환 단위 확장에 대한 자세한 내용은 새 측정 타입 추가 단원과 온라인 도움말의 Convert를 참조하십시오.

변환 수행

Convert 함수를 사용하여 단순한 변환 및 복잡한 변환을 모두 수행할 수 있습니다. 여기에는 단순한 구문 및 복잡한 측정값 타입 간의 변환 수행에 필요한 두 번째 구문이 포함됩니다.

단순한 변환 수행

Convert 함수를 사용하여 한 단위 집합에서 다른 단위 집합으로 측정값을 변환할 수 있습니다. *Convert* 함수는 거리, 넓이, 시간, 온도 등 동일한 타입의 대상을 측정하는 단위 간의 변환을 수행합니다.

*Convert*를 사용하려면 반드시 어디서 어디로 변환할 것인지 지정해야 합니다. *TConvType* 타입을 사용하여 측정값의 단위를 확인할 수 있습니다.

예를 들면, 다음은 화씨에서 절대 온도로 변환하는 예제입니다.

```
TempInKelvin = Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

복잡한 변환 수행

또한 *Convert* 함수를 사용하여 두 측정 타입의 비율 간에 보다 복잡한 변환을 수행할 수도 있습니다. 이러한 변환이 필요한 경우를 예로 들면, 속도를 계산하기 위해 시간당 마일을 분당 미터로 변환하거나 흐름을 계산하기 위해 분당 갤론을 시간당 리터로 변환하는 경우 등이 있습니다.

예를 들어, 다음은 갤론당 마일을 리터당 킬로미터로 변환하는 예제입니다.

```
double nKPL = Convert(StrToFloat(Edit1.Text), duMiles, vuGallons,
duKilometers, vuLiter);
```

변환하는 단위는 반드시 동일한 변환 패밀리에 속해야 합니다. 즉, 동일한 대상을 측정해야 합니다. 단위가 서로 호환되지 않으면 *Convert*가 *EConversionError* 예외를 발생시킵니다. 두 *TConvType* 값이 동일한 변환 패밀리에 속하는지 검사하려면 *CompatibleConversionTypes*를 호출하십시오.

StdConvs 유닛은 몇몇 *TConvType* 값 패밀리를 정의합니다. 미리 정의된 측정 단위 패밀리 및 각 패밀리의 측정 단위에 대한 리스트를 보려면 온라인 도움말에서 Conversion family variables를 참조하십시오.

새 측정 타입 추가

StdConvs 유닛에서 아직 정의되지 않은 측정 단위 간의 변환을 수행하려면 새 변환 패밀리를 만들어 측정 단위(*TConvType* 값)를 나타내야 합니다. 두 *TConvType* 값이 동일한 변환 패밀리로 등록될 경우 *Convert* 함수는 이러한 *TConvType* 값에 의해 표시되는 단위를 사용하여 측정값 사이의 변환을 수행할 수 있습니다.

먼저 *RegisterConversionFamily* 함수를 사용하여 변환 패밀리를 등록하면 *TConvFamily* 값을 얻을 수 있습니다. 새 변환 패밀리를 등록하거나 StdConvs 유닛의 전역 변수 중 하나를 사용하여 *TConvFamily* 값을 얻은 후에는 *RegisterConversionType* 함수를 사용하여 변환 패밀리에 새 단위를 추가할 수 있습니다. 다음은 이러한 작업을 보여 주는 예제입니다.

간단한 변환 패밀리 작성 및 단위 추가

새 변환 패밀리를 작성하고 새 측정 단위를 추가할 수 있는 예로 수개월에서 수세기로의 변환과 같이 장기간 간의 변환을 수행하여 정확도가 떨어지는 경우를 가정할 수 있습니다.

즉, *cbTime* 패밀리는 일(day)을 기본 단위로 사용합니다. 기본 단위는 해당 패밀리 내의 모든 변환 수행에 사용됩니다. 따라서 모든 변환은 일(day)이라는 기간에서 수행되어야 합니다. 일과 월, 일과 년 등은 정확히 변환되지 않으므로 월 또는 더 큰 단위(월, 년, 10년, 세기, 1000년)를 사용하여 변환하면 부정확할 수 있습니다. 각 월은 길이가 다르고 년에는 윤년, 윤초 등에 대한 수정 요인이 있습니다.

월 이상의 측정 단위만 사용하는 경우에는 년을 기본 단위로 사용하여 보다 정확한 변환 패밀리를 만들 수 있습니다. 다음은 *cbLongTime*라는 새 변환 패밀리를 만드는 예제입니다.

변수 선언

먼저 식별자에 대한 변수 선언이 필요합니다. 식별자는 새 LongTime 변환 패밀리 및 그 멤버인 측정 단위에서 사용됩니다.

```
tConvFamily cbLongTime;
TConvType ltMonths;
TConvType ltYears;
TConvType ltDecades;
TConvType ltCenturies;
TConvType ltMillennia;
```

변환 패밀리 등록

다음은 변환 패밀리를 등록합니다.

```
cbLongTime = RegisterConversionFamily (“Long Times”);
```

UnregisterConversionFamily 프로시저가 제공되지만 변환 패밀리를 정의하는 단위가 런타임 시 제거되지 않는 한 변환 패밀리를 등록 취소할 필요는 없습니다. 애플리케이션이 종료되면 자동으로 제거됩니다.

측정 단위 등록

다음에는 방금 작성한 변환 패밀리 내에 측정 단위를 등록해야 합니다. 지정된 패밀리 내에서 측정 단위를 등록하는 *RegisterConversionType* 함수를 사용하십시오. 기본 단위(예제에서는 년)를 정의해야 하며 다른 단위는 기본 단위에 대한 관계를 나타내는 요인을 사용하여 정의됩니다. 따라서 LongTime 패밀리에 대한 기본 단위가 년이므로 *ltMonths*에 대한 요인은 1/12입니다. 또한 변환하는 단위에 대한 설명을 포함시킬 수도 있습니다.

측정 단위를 등록하기 위한 코드는 다음과 같습니다.

```
ltMonths = RegisterConversionType(cbLongTime,”Months”,1/12);
ltYears = RegisterConversionType(cbLongTime,”Years”,1);
ltDecades = RegisterConversionType(cbLongTime,”Decades”,10);
ltCenturies = RegisterConversionType(cbLongTime,”Centuries”,100);
ltMillennia = RegisterConversionType(cbLongTime,”Millennia”,1000);
```

새 단위 사용

이제 새로 등록된 단위를 사용하여 변환을 수행할 수 있습니다. 전역 *Convert* 함수는 개발자가 *cbLongTime* 변환 패밀리에 등록한 모든 타입의 변환 간에 변환을 수행할 수 있습니다.

따라서 다음 *Convert* 호출을 사용하는 대신,

```
Convert(StrToFloat(Edit1->Text),tuMonths,tuMillennia);
```

보다 정확하게 변환하기 위해 다음과 같은 새 호출을 사용할 수 있습니다.

```
Convert(StrToFloat(Edit1->Text),ltMonths,ltMillennia);
```

변환 함수 사용

변환이 보다 복잡한 경우, 변환 요인을 사용하는 대신 다른 구문을 사용하여 변환을 수행할 함수를 지정할 수도 있습니다. 예를 들어, 다른 온도 단위는 출발점이 다르므로 변환 요인을 사용하여 온도 값을 변환할 수 없습니다.

StdConvs 유닛에서 변환된 이 예제는 함수를 제공하여 기본 단위 간에 변환을 수행함으로써 변환 타입을 등록하는 방법을 보여 줍니다.

변수 선언

먼저 식별자에 대한 변수를 선언합니다. 식별자는 *cbTemperature* 변환 패밀리에서 사용되며 측정 단위는 해당 멤버입니다.

```
TConvFamily cbTemperature;
TConvType tuCelsius;
TConvType tuKelvin;
TConvType tuFahrenheit;
```

참고 여기에 나열된 측정 단위는 실제로 *StdConvs* 유닛에 등록된 온도 단위의 부분 집합입니다.

변환 패밀리 등록

다음에는 변환 패밀리를 등록합니다.

```
cbTemperature = RegisterConversionFamily (“Temperature”);
```

기본 단위 등록

다음은 변환 패밀리의 기본 단위를 정의하고 등록합니다. 이 예제에서는 섭씨 온도입니다. 기본 단위의 경우에는 수행할 실제 변환이 없으므로 간단한 변환 요인을 사용할 수 있습니다.

```
tuCelsius = RegisterConversionType(cbTemperature,”Celsius”,1);
```

메소드를 작성하여 기본 단위 간의 변환 수행

이 변환은 단순 변환 요인에 의존하지 않으므로 각 온도 단위에서 섭씨 온도로 변환을 수행하는 코드를 작성해야 합니다. 이러한 함수는 StdConvs 유닛으로부터 변환됩니다.

```
double __fastcall FahrenheitToCelsius(const double AValue)
{
  return (((AValue - 32) * 5) / 9);
}
double __fastcall CelsiusToFahrenheit(const double AValue)
{
  return (((AValue * 9) / 5) + 32);
}
double __fastcall KelvinToCelsius(const double AValue)
{
  return (AValue - 273.15);
}
double __fastcall CelsiusToKelvin(const double AValue)
{
  return (AValue + 273.15);
}
```

다른 단위 등록

이제 변환 함수를 작성하였으므로 변환 패밀리 내에 다른 측정 단위를 등록할 수 있습니다. 또한 단위에 대한 설명을 포함시킬 수도 있습니다.

다른 단위를 패밀리에 등록하는 코드는 다음과 같습니다.

```
tuKelvin = RegisterConversionType(cbTemperature, “Kelvin”,
KelvinToCelsius, CelsiusToKelvin);
  tuFahrenheit = RegisterConversionType(cbTemperature, “Fahrenheit”,
FahrenheitToCelsius, CelsiusToFahrenheit);
```

새 단위 사용

이제 새로 등록된 단위를 사용하여 애플리케이션에서 변환을 수행할 수 있습니다. 전역 *Convert* 함수는 *cbTemperature* 변환 패밀리에 등록한 모든 변환 타입 간의 변환을 수행할 수 있습니다. 예를 들어 다음 코드는 값을 화씨 온도에서 절대 온도로 변환합니다.

```
Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

클래스를 사용하여 변환 관리

언제든지 변환 함수를 사용하여 변환 단위를 등록할 수 있습니다. 단, 본질적으로 같은 작업을 수행하는 불필요한 많은 수의 함수를 작성해야 하는 경우가 있습니다.

매개변수나 변수의 값만 다른 변환 함수 집합을 작성할 수 있다면 이러한 변환을 처리하는 클래스를 만들 수 있습니다. 예를 들어, 유로 도입 이후 다양한 유럽 통화 간의 변환에 필요한 표준 기술 집합이 있습니다. 달러와 유로 간의 변환 요인과 달리 변환 요인이 일정하지만 다음 두 가지 이유로 인해 유럽 통화 간의 적절한 변환에 간단한 변환 요인 접근법을 사용할 수 없습니다.

- 통화별 특정 자릿수로 변환을 반올림하거나 우수리를 버려야 합니다.
- 변환 요인 접근법은 표준 유로 변환에 의해 지정된 통화에 대해 역 요인을 사용합니다.

그러나 다음과 같은 변환 함수를 사용하면 이러한 문제를 모두 처리할 수 있습니다.

```
double __fastcall FromEuro(const double AValue, const double Factor,
TRoundToRange FRound)
{
  return(RoundTo(AValue * Factor, FRound));
}
double __fastcall ToEuro(const double AValue, const double Factor)
{
  return (AValue / Factor);
}
```

문제는 이 접근법에는 변환 함수에 대한 추가 매개변수가 필요하다는 점입니다. 즉, 모든 유럽 통화에 대해 간단히 동일한 함수를 등록할 수 없습니다. 모든 유럽 통화에 대해 새로운 두 함수를 작성하지 않으려면 동일한 두 함수를 클래스의 멤버로 만들어 사용할 수 있습니다.

변환 클래스 작성

클래스는 반드시 *TConvTypeFactor*의 자손이어야 합니다. *TConvTypeFactor*는 변환 패밀리의 기본 단위 간의 변환, 이 예제에서는 유로 간의 변환을 수행하기 위해 *ToCommon* 및 *FromCommon*이라는 두 메소드를 정의합니다. 변환 유닛을 등록할 때 직접 함수를 사용하듯이 이러한 메소드에는 추가 매개변수가 없으므로 반올림하거나 우수리를 버릴 자릿수를 제공해야 하며 변환 요인은 변환 클래스의 private 멤버로 제공해야 합니다. 이러한 사항은 demos\ConvertIt 디렉토리(euroconv.pas 참조)에 있는 EuroConv 예제에 나와 있습니다.

```
class PASCALIMPLEMENTATION TConvTypeEuroFactor : public
Convutils::TConvTypeFactor
{
  private:
    TRoundToRange FRound;
  public:
    __fastcall TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const
TRoundToRange ARound);
                TConvTypeFactor(AConvFamily, ADescription, AFactor);
    virtual double ToCommon(const double AValue);
    virtual double FromCommon(const double AValue);
}
```

생성자는 이러한 private 멤버에 값을 할당합니다.

```
__fastcall TConvTypeEuroFactor::TConvTypeEuroFactor(const TConvFamily
AConvFamily,
        const AnsiString ADescription, const double AFactor, const
TRoundToRange ARound):
                TConvTypeFactor(AConvFamily, ADescription, AFactor);
{
  FRound = ARound;
}
```

두 변환 함수는 단순히 이러한 private 멤버를 사용합니다.

```
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (RoundTo(AValue * Factor, FRound));
}

virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (AValue / Factor);
}
```

변수 선언

이제 변환 클래스가 작성되었으므로 식별자를 선언하여 다른 변환 패밀리로 시작합니다.

```
TConvFamily cbEuro;
TConvType euEUR; // EU euro
TConvType euBEF; // Belgian francs
TConvType euDEM; // German marks
TConvType euGRD; // Greek drachmas
TConvType euESP; // Spanish pesetas
TConvType euFFR; // French francs
TConvType euIEP; // Irish pounds
TConvType euITL; // Italian lire
TConvType euLUF; // Luxembourg francs
TConvType euNLG; // Dutch guilders
TConvType euATS; // Austrian schillings
TConvType euPTE; // Protuguese escudos
TConvType euFIM; // Finnish marks
```

변환 패밀리 및 기타 단위 등록

이제 새 변환 클래스를 사용하여 변환 패밀리와 유럽 통화 단위를 등록할 준비가 되었습니다. 다른 변환 패밀리를 등록한 것과 동일한 방법으로 변환 패밀리를 등록합니다.

```
cbEuro = RegisterConversionFamily (“European currency”);
```

각 변환 타입을 등록하려면 해당 통화의 요인과 반올림 속성을 반영하는 변환 클래스의 인스턴스를 만들고 RegisterConversionType 메소드를 호출하십시오.

```
TConvTypeInfo *pInfo = new TConvTypeEuroFactor(cbEuro, “EUEuro”, 1.0, -
2);
if (!RegisterConversionType(pInfo, euEUR))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “BelgianFrancs”, 40.3399, 0);
if (!RegisterConversionType(pInfo, euBEF))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “GermanMarks”, 1.95583, -2);
if (!RegisterConversionType(pInfo, euDEM))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “GreekDrachmas”, 340.75, 0);
if (!RegisterConversionType(pInfo, euGRD)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “SpanishPesetas”, 166.386, 0);
if (!RegisterConversionType(pInfo, euESP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “FrenchFrancs”, 6.55957, -2);
if (!RegisterConversionType(pInfo, euFFR)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “IrishPounds”, 0.787564, -2);
if (!RegisterConversionType(pInfo, euIEP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “ItalianLire”, 1936.27, 0);
if (!RegisterConversionType(pInfo, euITL)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “LuxembourgFrancs”, 40.3399, -2);
if (!RegisterConversionType(pInfo, euLUF)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “DutchGuilders”, 2.20371, -2);
if (!RegisterConversionType(pInfo, euNLG)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “AutstrianSchillings”, 13.7603, -
2);
if (!RegisterConversionType(pInfo, euATS)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “PortugueseEscudos”, 200.482, -
2);
if (!RegisterConversionType(pInfo, euPTE)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, “FinnishMarks”, 5.94573, 0);
if (!RegisterConversionType(pInfo, euFIM)
  delete pInfo;
```

새 단위 사용

이제 새로 등록된 단위를 사용하여 애플리케이션에서 변환을 수행할 수 있습니다. 전역 *Convert* 함수는 새 cbEuro 패밀리에 등록한 유럽 통화 중 어느 것과도 변환할 수 있습니다. 예를 들어, 다음 코드는 이탈리아 리라에서 독일 마르크로 값을 변환합니다.

```
Edit2->Text = FloatToStr(Convert(StrToFloat(Edit1->Text), euITL, euDEM));
```

그리기 공간 생성

TCanvas 클래스는 VCL 내의 Windows 장치 컨텍스트 및 CLX 내의 그리기 장치(Qt painter)를 캡슐화하며 패널 등의 시각적 컨테이너와 프린터 객체라는 두 폼에 대한 그리기를 모두 처리합니다(4-25페이지의 "인쇄" 참조). 캔버스 객체를 사용하면 사용자를 위해 모든 할당 및 할당 해제가 처리되므로 펜, 브러쉬, 팔레트 등의 할당을 염려하지 않아도 됩니다.

*TCanvas*는 캔버스를 포함하는 컨트롤에 선, 도형, 다각형 및 글꼴 등을 그리는 여러 가지 기본적인 그래픽 루틴을 포함합니다. 예를 들어, 다음은 폼의 왼쪽 상단 모서리에서 가운데로 선을 그리고 원래의 텍스트를 폼에 출력하는 버튼 이벤트 핸들러를 보여 줍니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo( 10, 10 );
   Canvas->LineTo( 100, 100 );
   Canvas->Brush->Color = clBtnFace;
   Canvas->Font->Name = "Arial";
   Canvas->TextOut( Canvas->PenPos.x,Canvas->PenPos.y,"This is the end of
the line" );
}
```

Windows 애플리케이션에서 *TCanvas* 객체는 장치 컨텍스트, 펜, 브러쉬 등을 그리기 작업 이전의 값으로 복구하는 등 일반적인 Windows 그래픽 오류를 방지하기도 합니다. *TCanvas*는 C++Builder에서 그리기가 필요하거나 그리가가 가능한 모든 곳에서 사용되고, 그래픽을 안전하고 쉽게 그릴 수 있습니다.

속성 및 메소드에 대한 전체 리스트를 보려면 온라인 도움말 참조에서 *TCanvas*를 참조하십시오.

장

5

컴포넌트 작업

통합 개발 환경(IDE)에서 컴포넌트 팔레트는 여러 가지 컴포넌트를 제공합니다. 컴포넌트 팔레트에서 컴포넌트를 선택하여 폼이나 데이터 모듈에 놓을 수 있습니다. 버튼, 리스트 박스 등과 같은 비주얼(visual) 컴포넌트를 폼에 배열하여 애플리케이션 사용자 인터페이스를 디자인합니다. 데이터 액세스 컴포넌트와 같은 넌비주얼(nonvisual) 컴포넌트를 폼이나 데이터 모듈에 넣을 수도 있습니다.

언뜻 보면, C++Builder 컴포넌트는 다른 C++ 클래스와 똑같아 보입니다. 그러나 C++Builder 컴포넌트와 대부분의 C++ 프로그래머가 사용하는 표준 C++ 클래스 계층 구조 사이에는 차이점이 있습니다. 몇 가지 차이점은 다음과 같습니다.

- 모든 C++Builder 컴포넌트는 *TComponent*의 자손입니다.
- 컴포넌트는 기능을 추가하거나 변경하기 위해 서브 클래스로 지정하는 "기본 클래스"로 사용되기보다는 대부분 있는 그대로 사용되고 속성을 통해 변경됩니다. 상속된 컴포넌트인 경우 일반적으로 기존 이벤트 처리 멤버 함수에 특정 코드를 추가해야 합니다.
- 컴포넌트는 스택이 아닌 힙에만 할당할 수 있으므로 *반드시* **new** 연산자를 사용하여 컴포넌트를 생성해야 합니다.
- 컴포넌트의 속성은 본질적으로 런타임 타입 정보를 포함합니다.
- 컴포넌트를 C++Builder 사용자 인터페이스의 컴포넌트 팔레트에 추가하여 폼에서 처리할 수 있습니다.

컴포넌트는 일반적으로 표준 C++ 클래스에서보다 나은 캡슐화 기능을 제공하는 경우가 많습니다. 예를 들어 누름 버튼을 포함하는 다이얼로그 박스를 사용하는 경우를 생각해 보십시오. VCL 컴포넌트를 사용하여 개발하는 C++ Windows 프로그램인 경우 사용자가 버튼을 클릭하면 시스템은 WM_LBUTTONDOWN 메시지를 생성합니다. 프로그램은 보통 **switch** 문, 메시지 맵 또는 응답 테이블에 이 메시지를 catch하고 메시지에 응답하여 실행될 루틴으로 메시지를 디스패치해야 합니다.

대부분의 Windows 메시지(VCL) 또는 시스템 이벤트(CLX)는 C++Builder 컴포넌트에서 처리합니다. 메시지에 응답하려는 경우 이벤트 핸들러만 제공하면 됩니다.

8장, "애플리케이션 사용자 인터페이스 개발"에서는 동적으로 modal 폼 생성, 폼에 매개변수 전달 및 폼에서 데이터 검색 등과 같은 폼 사용에 대해 자세히 설명합니다.

컴포넌트 속성 설정

디자인 타임에 published 속성을 설정하려면 Object Inspector를 사용할 수 있으며 일부 경우에 특수 속성 에디터를 사용할 수 있습니다. 런타임에 속성을 설정하려면 애플리케이션 소스 코드에서 속성 값을 할당하십시오.

각 컴포넌트의 속성에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

디자인 타임에 속성 설정

디자인 타임에 폼에서 컴포넌트를 선택하면 Object Inspector는 이 컴포넌트의 published 속성을 표시하고 필요한 경우 속성을 편집할 수 있게 합니다. *Tab* 키를 사용하면 왼쪽 Property 열과 오른쪽 Value 열을 토글할 수 있습니다. 커서가 Property 열에 있는 경우 속성 이름의 첫 글자를 입력하여 속성을 탐색할 수 있습니다. 부울 또는 열거 타입의 속성인 경우 드롭다운 리스트에서 값을 선택하거나 Value 열을 더블 클릭하여 설정을 토글할 수 있습니다.

속성 이름 옆에 + 기호가 표시되어 있는 경우 이 + 기호를 클릭하거나 포커스가 이 속성에 있을 때 '+'를 입력하면 해당 속성의 하위 값 리스트가 표시됩니다. 마찬가지로 속성 이름 옆에 - 기호가 표시되어 있는 경우 - 기호를 클릭하거나 '-'를 입력하면 하위 값을 숨깁니다.

디폴트로, 레거시 범주의 속성은 표시되지 않습니다. 디스플레이 필터를 변경하려면 Object Inspector에서 마우스 오른쪽 버튼을 클릭하고 View를 선택합니다. 자세한 내용은 온라인 도움말에서 "property categories"를 참조하십시오.

두 개 이상의 컴포넌트를 선택하면 Object Inspector는 선택한 컴포넌트에서 공유하는 모든 속성(*Name* 제외)을 표시합니다. 공유하는 속성의 값이 선택한 컴포넌트 간에 다른 경우 Object Inspector는 기본값이나 첫째로 선택한 컴포넌트의 값을 표시합니다. 공유 속성을 변경하면 이 변경 내용은 선택된 모든 컴포넌트에 적용됩니다.

Object Inspector에서 이벤트 핸들러 이름과 같은 코드 관련 속성을 변경하면 해당 소스 코드가 자동으로 변경됩니다. 또한 폼 클래스 선언에서 이벤트 핸들러 메소드 이름을 다시 지정하는 것과 같이 소스 코드를 변경하면 Object Inspector에서 그 변경 내용이 곧바로 반영됩니다.

속성 에디터 사용

일부 속성(예: *Font*)에는 특수 속성 에디터가 있습니다. Object Inspector에서 이런 속성을 선택하면 속성 값 옆에 생략 부호(...)가 표시됩니다. 속성 에디터를 열려면 Value 열을 더블 클릭하거나, 생략 부호를 클릭하거나, 포커스가 해당 속성이나 속성 값에 있을 때 *Ctrl+Enter*를 누르십시오. 일부 컴포넌트의 경우 폼에서 컴포넌트를 더블 클릭해도 속성 에디터가 열립니다.

속성 에디터를 사용하면 단일 다이얼로그 박스에서 복잡한 속성을 설정할 수 있습니다. 속성 에디터는 입력 확인을 제공하며 종종 할당 결과를 미리 보여 주기도 합니다.

런타임에 속성 설정

쓰기 가능한 속성은 런타임에 소스 코드에서 설정할 수 있습니다. 예를 들어 다음과 같이 동적으로 폼에 캡션을 할당할 수 있습니다.

```
Form1->Caption = MyString;
```

메소드 호출

메소드는 일반적인 프로시저 및 함수와 똑같이 호출됩니다. 예를 들어 비주얼(visual) 컨트롤은 화면상의 컨트롤 이미지를 새로 고치는 *Repaint* 메소드를 가집니다. 다음과 같이 draw-grid 객체에서 *Repaint* 메소드를 호출할 수 있습니다.

```
DrawGrid1->Repaint;
```

속성에서와 마찬가지로 메소드 이름의 범위에 따라 한정자가 필요한지 여부가 결정됩니다. 예를 들어 폼의 자식 컨트롤 중 하나의 이벤트 핸들러 내에서 폼을 다시 그리는 경우 메소드 호출 앞에 폼 이름을 붙이지 않아도 됩니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Repaint;
}
```

이벤트 및 이벤트 핸들러 작업

C++Builder에서 작성되는 거의 모든 코드는 *이벤트*에 직접 또는 간접적으로 응답하여 실행됩니다. 이벤트는 런타임에 발생하는 일(종종 사용자 작업임)을 나타내는 특수 유형의 속성이며, 이벤트에 직접 응답하는 코드(*이벤트 핸들러*라고 함)는 객체의 메소드입니다. 다음 단원에서는 아래와 같은 작업을 하는 방법에 대해 설명합니다.

- 새 이벤트 핸들러 생성
- 컴포넌트의 디폴트 이벤트에 대한 핸들러 생성
- 이벤트 핸들러 찾기
- 이벤트를 기존 이벤트 핸들러에 연결
- 메뉴 이벤트를 이벤트 핸들러에 연결
- 이벤트 핸들러 삭제

새 이벤트 핸들러 생성

C++Builder는 폼과 다른 컴포넌트를 위한 스켈레톤 이벤트 핸들러를 생성할 수 있습니다. 다음과 같은 방법으로 이벤트 핸들러를 생성합니다.

1 컴포넌트를 선택합니다.

2 Object Inspector의 Events 탭을 클릭합니다. Object Inspector의 Events 페이지는 컴포넌트에 대해 정의된 모든 이벤트를 표시합니다.

3 원하는 이벤트를 선택하고 Value 열을 더블 클릭하거나 *Ctrl+Enter*를 누릅니다.

4 이벤트가 발생할 때 실행할 코드를 입력합니다.

컴포넌트의 디폴트 이벤트에 대한 핸들러 생성

일부 컴포넌트에는 컴포넌트가 가장 일반적으로 처리해야 하는 *디폴트* 이벤트가 있습니다. 예를 들어 버튼의 디폴트 이벤트는 *OnClick*입니다. 디폴트 이벤트 핸들러를 생성하려면 폼 디자이너에서 컴포넌트를 더블 클릭합니다. 그러면 스켈레톤 이벤트 처리 프로시저가 생성되고 코드 에디터가 열리는데, 이때 커서는 프로시저 바디에 놓여 있게 됩니다. 여기서 쉽게 코드를 추가할 수 있습니다.

모든 컴포넌트에 디폴트 이벤트가 있는 것은 아닙니다. 일부 컴포넌트(예: *TBevel*)는 이벤트에 응답하지 않으며, 일부 컴포넌트는 폼 디자이너에서 컴포넌트를 더블 클릭할 때 다르게 응답합니다. 예를 들어 디자인 타임에 컴포넌트를 더블 클릭하면 많은 컴포넌트가 디폴트 속성 에디터를 열지만 어떤 것은 다이얼로그 박스를 열기도 합니다.

이벤트 핸들러 찾기

폼 디자이너에서 컴포넌트를 더블 클릭하여 디폴트 이벤트 핸들러를 생성한 경우 같은 방법으로 해당 이벤트 핸들러를 찾을 수 있습니다. 컴포넌트를 더블 클릭하면 코드 에디터가 열리며, 이벤트 핸들러 바디의 시작 부분에 커서가 놓여 있습니다.

다음과 같은 방법으로 디폴트가 아닌 이벤트 핸들러를 찾습니다.

1 폼에서 찾을 이벤트 핸들러가 있는 컴포넌트를 선택합니다.

2 Object Inspector에서 Events 탭을 클릭합니다.

3 보려는 핸들러가 있는 이벤트를 선택하고 Value 열을 더블 클릭합니다. 코드 에디터가 열리고 커서는 이벤트 핸들러 바디의 시작 부분에 놓여 있습니다.

이벤트를 기존 이벤트 핸들러에 연결

둘 이상의 이벤트에 응답하는 이벤트 핸들러를 작성하면 코드를 다시 사용할 수 있습니다. 예를 들어 많은 애플리케이션은 드롭다운 메뉴 명령과 동일한 스피드 버튼을 제공합니다. 버튼이 메뉴 명령과 같은 작업을 시작하는 경우 하나의 이벤트 핸들러를 작성하여 버튼과 메뉴 항목의 *OnClick* 이벤트 모두에 이 작업을 할당할 수 있습니다.

다음과 같은 방법으로 이벤트를 기존 이벤트 핸들러에 연결합니다.

1 폼에서 처리할 이벤트가 있는 컴포넌트를 선택합니다.

2 Object Inspector의 Events 페이지에서 핸들러를 연결할 이벤트를 선택합니다.

3 이벤트 옆의 Value 열에서 아래쪽 화살표를 클릭하여 이전에 작성한 이벤트 핸들러 리스트를 엽니다. 이 리스트에는 같은 폼에서 같은 이름의 이벤트에 대해 작성된 이벤트 핸들러만 표시됩니다. 리스트에서 이벤트 핸들러 이름을 클릭하여 선택합니다.

위의 절차를 사용하면 이벤트 핸들러를 쉽게 다시 사용할 수 있습니다. 그러나 *액션 리스트* (VCL인 경우 *액션 밴드*)와 같은 강력한 도구를 사용하면 사용자 명령에 응답하는 코드를 중앙에서 구성할 수 있습니다. 액션 리스트는 다른 플랫폼에서 사용할 수 있지만 액션 밴드는 다른 플랫폼에서 사용할 수 없습니다. 액션 리스트 및 액션 밴드에 대한 자세한 내용은 8-16페이지의 "툴바와 메뉴에 대한 액션 구성"을 참조하십시오.

Sender 매개변수 사용

이벤트 핸들러에서 *Sender* 매개변수는 이벤트를 받고 핸들러를 호출한 컴포넌트를 나타냅니다. 때로는 호출하는 컴포넌트에 따라 다르게 동작하는 이벤트 핸들러를 여러 컴포넌트가 공유하게 하면 유용할 수 있습니다. 이렇게 하려면 *Sender* 매개변수를 사용하십시오.

공유 이벤트 표시 및 코딩

컴포넌트들이 이벤트를 공유하는 경우 Object Inspector에 이 공유 이벤트를 표시할 수 있습니다. 우선 폼 디자이너에서 *Shift* 키를 누른 채로 컴포넌트들을 클릭하여 선택합니다. 그러면 Object Inspector의 Value 열에서 공유 이벤트에 대해 새 이벤트 핸들러를 생성하거나 기존 이벤트를 할당할 수 있습니다.

메뉴 이벤트를 이벤트 핸들러에 연결

메뉴 디자이너에서 *MainMenu* 및 *PopupMenu* 컴포넌트를 사용하여 애플리케이션에 드롭다운 및 팝업 메뉴를 쉽게 제공할 수 있습니다. 그러나 메뉴가 작동하려면 각 메뉴 항목은 사용자가 메뉴 항목을 선택하거나 해당 가속키 또는 단축키를 누를 때마다 발생하는 *OnClick* 이벤트에 응답해야 합니다. 이 단원에서는 이벤트 핸들러와 메뉴 항목을 연결하는 방법에 대해 설명합니다. 메뉴 디자이너 및 관련 컴포넌트에 대한 자세한 내용은 8-29페이지의 "메뉴 생성 및 관리"를 참조하십시오.

다음과 같은 방법으로 메뉴 항목에 대한 이벤트 핸들러를 생성합니다.

1 *MainMenu* 또는 *PopupMenu* 컴포넌트를 더블 클릭하여 Menu Designer를 엽니다.

2 Menu Designer에서 메뉴 항목을 선택합니다. Object Inspector에서 항목의 *Name* 속성에 값을 할당합니다.

3 Menu Designer에서 메뉴 항목을 더블 클릭합니다. C++Builder는 코드 에디터에 이벤트 핸들러를 생성합니다.

4 사용자가 메뉴 명령을 선택할 때 실행할 코드를 입력합니다.

다음과 같은 방법으로 메뉴 항목을 기존 *OnClick* 이벤트 핸들러에 연결합니다.

1. *MainMenu* 또는 *PopupMenu* 컴포넌트를 더블 클릭하여 Menu Designer를 엽니다.
2. Menu Designer에서 메뉴 항목을 선택합니다. Object Inspector에서 항목의 *Name* 속성에 값을 할당합니다.
3. Object Inspector의 Events 페이지에서 *OnClick* 옆에 있는 아래쪽 화살표를 클릭하여 기존에 작성된 이벤트 핸들러를 엽니다. 이 리스트에는 이 폼에서 *OnClick* 이벤트에 대해 작성된 이벤트 핸들러만 표시됩니다. 리스트에서 이벤트 핸들러 이름을 클릭하여 선택합니다.

이벤트 핸들러 삭제

폼 디자이너를 사용하여 폼에서 컴포넌트를 삭제하면 C++Builder는 폼의 타입 선언에서 해당 컴포넌트를 제거합니다. 그러나 해당 컴포넌트와 연결된 메소드는 유닛 파일에서 삭제되지 않습니다. 폼에 있는 다른 컴포넌트에서 이 메소드를 여전히 호출할 수 있기 때문입니다. 이벤트 핸들러와 마찬가지로 메소드를 수동으로 삭제할 수도 있지만 이렇게 하는 경우에는 메소드의 forward 선언과 구현도 같이 삭제해야 합니다. 그렇지 않으면 프로젝트를 생성할 때 컴파일러 오류가 발생합니다.

크로스 플랫폼 컴포넌트 및 크로스 플랫폼이 아닌 컴포넌트

컴포넌트 팔레트에는 광범위한 프로그래밍 작업을 처리하는 여러 가지 컴포넌트들이 포함되어 있습니다. 컴포넌트는 해당 용도와 기능에 따라 페이지에 배열됩니다. 예를 들어 메뉴, 에디트 박스 또는 버튼을 만드는 컴포넌트와 같이 일반적으로 사용되는 컴포넌트는 Standard 페이지에 있습니다. 디폴트 구성에 표시되는 페이지는 실행하는 제품의 에디션에 따라 다릅니다.

표 3.3은 다른 플랫폼에서 사용할 수 없는 컴포넌트를 포함하여 애플리케이션을 만드는 데 사용할 수 있는 일반적인 디폴트 페이지와 컴포넌트를 나열합니다. 모든 CLX 컴포넌트는 Windows 및 Linux 애플리케이션에서 사용할 수 있습니다. 일부 VCL 특정 컴포넌트는 Windows 전용의 CLX 애플리케이션에서만 사용할 수 있으며, 애플리케이션의 코드 부분을 분리하지 않는 한 다른 플랫폼에서는 사용할 수 없습니다.

표 5.1 컴포넌트 팔레트 페이지

페이지 이름	설명	크로스 플랫폼 여부
Standard	표준 컨트롤, 메뉴	예
Additional	특화된 컨트롤	예(ApplicationEvents, ActionManager, ActionMain-MenuBar, ActionToolBar 및 CustomizeDlg 제외). LCDNumber는 CLX에만 있습니다.

표 5.1 컴포넌트 팔레트 페이지 (계속)

페이지 이름	설명	크로스 플랫폼 여부
Win32(VCL)/ Common Controls(CLX)	Windows 공용 컨트롤	Win32 페이지에 있는 컴포넌트 중 많은 수가 CLX 애플리케이션을 만들 때 표시되는 Common Controls 페이지에도 있습니다. RichEdit, UpDown, HotKey, Ani-mate, DataTimePicker, MonthCalen-dar, Coolbar, PageScroller 및 ComboBoxEx는 VCL에만 있습니다. TextBrowser, TextViewer, Icon-Viewer 및 SpinEdit는 CLX에만 있습니다.
System	타이머, 멀티미디어, DDE 등을 포함하는 시스템 수준 액세스를 위한 컴포넌트 및 컨트롤	아니오(CLX 애플리케이션을 만들 때 Additional 페이지에 있는 Timer 및 PaintBox 제외)
Data Access	특정 데이터 액세스 메커니즘에 연결되지 않은 데이터베이스 데이터 작업을 위한 컴포넌트	예(XMLTransform, XMLTransformProvider 및 XML-TransformClient 제외)
Data Controls	비주얼(visual), 데이터 인식 컨트롤	예(DBRichEdit, DBCtrlGrid 및 DBChart 제외)
dbExpress	동적 SQL 처리를 위한 메소드를 제공하며 크로스 플랫폼이고 데이터베이스 독립적인 레이어인 dbExpress를 사용하는 데이터베이스 컨트롤. SQL 서버를 액세스하기 위한 공용 인터페이스를 정의합니다.	예
DataSnap	멀티 티어 데이터베이스 애플리케이션을 만드는 데 사용되는 컴포넌트	아니오
BDE	Borland Database Engine을 통한 데이터 액세스를 제공하는 컴포넌트	아니오
ADO	ADO 프레임워크를 통한 데이터 액세스를 제공하는 컴포넌트	아니오
InterBase	InterBase 데이터베이스에 직접 액세스할 수 있게 하는 컴포넌트	예
InterBaseAdmin	InterBase Services API 호출을 액세스하는 컴포넌트	예
InternetExpress	웹 서버 애플리케이션인 동시에 멀티 티어 데이터베이스 애플리케이션의 클라이언트이기도 한 컴포넌트	아니오
Internet	인터넷 통신 프로토콜과 웹 애플리케이션용 컴포넌트	아니오
WebSnap	웹 서버 애플리케이션 작성을 위한 컴포넌트	아니오
FastNet	NetMasters Internet 컨트롤	아니오
QReport	포함된 보고서(embedded report)를 작성하기 위한 QuickReport 컴포넌트	아니오

표 5.1 컴포넌트 팔레트 페이지 (계속)

페이지 이름	설명	크로스 플랫폼 여부
Dialogs	일반적으로 사용되는 다이얼로그 박스	예(OpenPictureDialog, SavePictureDialog, PrintDialog 및 PrinterSetupDialog 제외)
Win 3.1	이전 스타일 Win 3.1 컴포넌트	아니오
Samples	예제 사용자 정의 컴포넌트	아니오
ActiveX	예제 ActiveX 컨트롤(Microsoft 문서 (msdn.microsoft.com) 참조)	아니오
COM+	COM+ 이벤트를 처리하기 위한 컴포넌트	아니오
WebServices	SOAP 기반 웹 서비스를 구현 또는 사용하는 애플리케이션을 작성하기 위한 컴포넌트	아니오
Servers	Microsoft Excel, Word 등을 위한 COM 서버 예제(Microsoft MSDN 문서 참조)	아니오
Indy Clients	클라이언트를 위한 크로스 플랫폼 Internet 컴포넌트(오픈 소스 Winshoes Inter-net 컴포넌트)	예
Indy Servers	서버를 위한 크로스 플랫폼 Internet 컴포넌트(오픈 소스 Winshoes Inter-net 컴포넌트)	예
Indy Misc	추가 크로스 플랫폼 Internet 컴포넌트 (오픈 소스 Winshoes Internet 컴포넌트)	예

팔레트에서 컴포넌트를 추가, 제거 및 재정렬할 수 있고 컴포넌트 *템플릿*및 *프레임*을 만들어 여러 컴포넌트를 그룹화할 수 있습니다.

온라인 도움말은 컴포넌트 팔레트의 컴포넌트에 대한 정보를 제공합니다. 그러나 ActiveX, Servers 및 Samples 페이지에 있는 컴포넌트 중 일부는 예제로만 제공되므로 설명이 없습니다.

VCL와 CLX의 차이점에 대한 자세한 내용은 14장, "크로스 플랫폼 애플리케이션 개발"을 참조하십시오.

컴포넌트 팔레트에 사용자 정의 컴포넌트 추가

직접 또는 업체에서 작성한 사용자 정의 컴포넌트를 컴포넌트 팔레트에 설치하고 애플리케이션에서 사용할 수 있습니다. 사용자 정의 컴포넌트를 생성하려면 V부, "사용자 정의 컴포넌트 생성"을 참조하고, 기존의 컴포넌트를 설치하려면 15-5페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

장

6

컨트롤 작업

컨트롤이란 사용자가 런타임 시에 상호 작용할 수 있는 비주얼(visual) 컴포넌트입니다. 이 장에서는 대부분의 컨트롤에서 일반적으로 찾아볼 수 있는 다양한 기능에 대해 설명합니다.

컨트롤에서 드래그 앤 드롭 구현

드래그 앤 드롭은 객체를 조작할 때 사용할 수 있는 편리한 방법입니다. 드래그 앤 드롭을 사용하면 사용자가 전체 컨트롤을 끌어 놓거나 컨트롤 간에 리스트 박스나 트리 뷰와 같은 항목을 끌어 놓을 수 있습니다.

- 끌기 작업 시작
- 끌어온 항목 승인
- 항목 놓기
- 끌기 작업 끝내기
- 끌기 객체로 드래그 앤 드롭 사용자 정의
- 끌기 마우스 포인터 변경

끌기 작업 시작

모든 컨트롤에는 끌기 작업이 시작되는 방법을 결정하는 *DragMode*라는 속성이 있습니다. *DragMode*가 *dmAutomatic*이면 사용자가 컨트롤에 커서를 놓고 마우스 버튼을 누를 때 끌기가 자동으로 시작됩니다. *dmAutomatic*은 일반적인 마우스 동작을 방해할 수 있으므로 *DragMode*를 *dmManual*(기본값)로 설정한 다음 마우스 다운(mouse-down) 이벤트를 처리하여 끌기를 시작할 수 있습니다.

컨트롤을 수동으로 끌기 시작하려면 컨트롤의 *BeginDrag* 메소드를 호출하십시오. *BeginDrag*는 *Immediate*라는 부울 매개변수와 *Threshold*라는 정수 매개변수를 사용합니다. *Immediate*에 **true**를 전달하면 즉시 끌기가 시작됩니다. **false**를 전달하면 사용자가 *Threshold*에 지정된 픽셀 수만큼 마우스를 이동할 때까지 끌기가 시작되지 않습니다. *Threshold*가 -1이면 기본값이 사용됩니다. 다음 메소드를 호출하면

```
BeginDrag (false, -1);
```

컨트롤이 끌기 작업을 시작하지 않고 마우스 클릭을 승인할 수 있습니다.

*BeginDrag*를 호출하기 전에 마우스 다운(mouse-down) 이벤트의 매개변수를 테스트하여 사용자가 누른 마우스 버튼 확인과 같은 다른 조건을 지정하여 끌기 시작 여부를 결정할 수 있습니다. 예를 들어, 다음 코드는 마우스 왼쪽 버튼을 눌렀을 때만 끌기 작업을 시작하여 파일 리스트 박스의 마우스 다운(mouse-down) 이벤트를 처리합니다.

```
void __fastcall TFMForm::FileListBox1MouseDown(TObject *Sender,
  TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (Button == mbLeft)// drag only if left button pressed
  {
    TFileListBox *pLB = (TFileListBox *)Sender; // cast to TFileListBox
    if (pLB->ItemAtPos(Point(X,Y), true) >= 0) // is there an item here?
      pLB->BeginDrag(false, -1);                  // if so, drag it
  }
}
```

끌어온 항목 승인

사용자가 컨트롤로 항목을 끌면 해당 컨트롤이 *OnDragOver* 이벤트를 받습니다. 이 때 사용자가 항목을 놓는 경우 승인할 수 있는지 여부를 나타내야 합니다. 끌어온 항목을 컨트롤이 승인할 수 있는지 여부를 나타내도록 끌기 커서가 변경됩니다. 컨트롤로 끌어온 항목을 승인하려면 컨트롤의 *OnDragOver* 이벤트에 이벤트 핸들러를 추가하십시오.

드래그 오버(drag-over) 이벤트에는 항목을 승인할 경우 이벤트 핸들러가 **true**로 설정할 수 있는 *Accept*라는 매개변수가 있습니다. *Accept*는 커서 타입을 승인 커서로 변경하거나 변경하지 않습니다.

드래그 오버(drag-over) 이벤트에는 끌기 소스와 마우스 커서의 현재 위치 등 이벤트 핸들러가 끌기 승인 여부를 결정하는 데 사용할 수 있는 다른 매개변수가 있습니다. 다음 예제에서는 디렉토리 트리 뷰에서 파일 리스트 박스에서 끌어온 항목만 승인합니다.

```
void __fastcall TForm1::TreeView1DragOver(TObject *Sender, TObject
*Source,
  int X, int Y, TDragState State, bool &Accept)
{
  if (Source->InheritsFrom(__classid(TFileListBox)))
    Accept = true;
}
```

항목 놓기

컨트롤이 끌어온 항목을 승인할 수 있다고 표시하면 항목을 놓도록 처리해야 합니다. 놓은 항목을 처리하려면 이벤트 핸들러를 놓기를 승인한 컨트롤의 *OnDragDrop* 이벤트에 추가하십시오. 드래그 오버(drag-over) 이벤트와 마찬가지로 드래그 앤 드롭(drag-and-drop) 이벤트는 끌어온 항목의 소스와 승인하는 컨트롤 위의 마우스 커서 좌표를 나타냅니다. 두 번째 매개변수를 사용하면 항목을 끄는 동안의 경로를 모니터할 수 있습니다. 예를 들면 항목을 놓는 경우에 이 정보를 사용하여 컴포넌트의 색을 변경할 수 있습니다.

다음 예제에서는 파일 리스트 박스에서 끌어온 항목을 승인한 디렉토리 트리 뷰에서 항목을 놓는 디렉토리로 파일을 이동하여 응답합니다.

```
void __fastcall TForm1::TreeView1DragDrop(TObject *Sender, TObject
*Source,
int X, int Y){
  if (Source->InheritsFrom(__classid(TFileListBox)))
  {
    TTreeNode *pNode = TreeView1->GetNodeAt(X,Y); // pNode is drop target
    AnsiString NewFile = pNode->Text + AnsiString("//") +
      ExtractFileName(FileListBox1->FileName); // build file name for
drop target
    MoveFileEx(FileListBox1->FileName.c_str(), NewFile.c_str(),
      MOVEFILE_REPLACE_EXISTING | MOVEFILE_COPY_ALLOWED); // move the
file
  }
}
```

끌기 작업 끝내기

항목을 놓거나 승인할 수 없는 컨트롤 위에 릴리스하면 끌기 작업이 끝납니다. 이 때 엔드 드래그(end-drag) 이벤트가 끌기를 시작한 컨트롤로 전달됩니다. 항목을 끌었을 때 컨트롤에서 응답하게 하려면 이벤트 핸들러를 컨트롤의 *OnEndDrag* 이벤트에 추가하십시오.

OnEndDrag 이벤트에서 가장 중요한 매개변수는 놓기를 승인할 컨트롤을 나타내는 *Target* 입니다. *Target*이 **Null**이면 끌어온 항목을 승인할 컨트롤이 없다는 의미입니다. *OnEndDrag* 이벤트에는 받은 컨트롤의 좌표도 포함됩니다.

이 예제에서 파일 리스트 박스는 해당 파일 리스트를 새로 고쳐서 엔드 드래그(end-drag) 이벤트를 처리합니다.

```
void __fastcall TFMForm::FileListBox1EndDrag(TObject *Sender, TObject
*Target,   int X, int Y)
  if (Target)
    FileListBox1->Update();
};
```

끌기 객체로 드래그 앤 드롭 사용자 정의

TDragObject 자손을 사용하여 객체의 드래그 앤 드롭 동작을 사용자 정의할 수 있습니다. 표준 드래그 오버(drag-over) 및 드래그 앤 드롭(drag-and-drop) 이벤트는 끌어온 항목의 소스와 승인한 컨트롤 위에 있는 마우스 커서의 좌표를 나타냅니다. 자세한 상태 정보를 보려면 *TDragObject* 또는 *TDragObjectEx*(VCL만 해당)에서 사용자 정의 끌기 객체를 파생시켜 해당 가상 메소드를 오버라이드하십시오. *OnStartDrag* 이벤트에서 사용자 정의 끌기 객체를 만듭니다.

일반적으로 드래그 오버(drag-over) 및 드래그 앤 드롭(drag-and-drop) 이벤트의 소스 매개변수는 끌기 작업을 시작한 컨트롤입니다. 여러 종류의 컨트롤이 같은 종류의 데이터와 관련된 동작을 시작할 수 있으면 소스에서 각 컨트롤 종류를 지원해야 합니다. *TDragObject*의 자손을 사용할 때 소스가 끌기 객체 자체라면, 각 컨트롤이 해당 *OnStartDrag* 이벤트에 같은 종류의 끌기 객체를 만드는 경우 대상에서 한 가지 종류의 객체만 처리해야 합니다. 드래그 오버(drag-over) 및 드래그 앤 드롭(drag-and-drop) 이벤트는 *IsDragObject* 함수를 호출하여 소스가 컨트롤이 아니라 끌기 객체인지 여부를 알려줄 수 있습니다.

TDragObjectEx 자손(VCL만 해당)은 자동으로 해제되지만 *TDragObject*의 자손은 그렇지 않습니다. 명시적으로 해제하지 않은 *TDragObject* 자손이 있는 경우 *TDragObjectEx*의 자손으로 변경하여 메모리 손실을 방지할 수 있습니다.

끌기 객체를 사용하면 애플리케이션의 메인 실행 파일에 구현된 폼과 DLL을 사용하여 구현된 폼 간에, 또는 서로 다른 DLL을 사용하여 구현된 폼 간에 항목을 끌 수 있습니다.

끌기 마우스 포인터 변경

소스 컴포넌트의 *DragCursor* 속성(VCL만 해당)을 설정하여 끌기 작업 중 마우스 포인터의 모양을 사용자 정의할 수 있습니다.

컨트롤에 드래그 앤 도킹 구현

*TWinControl*의 자손은 도킹 사이트 역할을 하고 *TControl*의 자손은 도킹 사이트에 도킹된 자식 윈도우 역할을 합니다. 예를 들어, 폼 윈도우의 왼쪽 가장자리에 도킹 사이트를 제공하려면 패널을 폼의 왼쪽 가장자리에 맞추고 패널을 도킹 사이트로 만듭니다. 도킹 가능한 컨트롤을 패널로 끌어 릴리스하면 패널의 자식 컨트롤이 됩니다.

- 윈도우 컨트롤을 도킹 사이트로 만들기
- 컨트롤을 도킹 가능한 자식으로 만들기
- 자식 컨트롤의 도킹 방법 제어
- 자식 컨트롤의 도킹 해제 방법 제어
- 드래그 앤 도킹 작업에 대한 자식 컨트롤의 응답 방법 제어

참고 드래그 앤 도킹 속성은 VCL에서는 사용할 수 있으나 CLX에서는 사용할 수 없습니다.

윈도우 컨트롤을 도킹 사이트로 만들기

다음과 같은 방법으로 윈도우 컨트롤을 도킹 사이트로 만듭니다.

1 *DockSite* 속성을 **true**로 설정합니다.

2 도킹된 클라이언트를 포함하는 경우를 제외하고는 도킹 사이트 객체가 나타나지 않아야 하는 경우 해당 *AutoSize* 속성을 **true**로 설정합니다. *AutoSize*가 **true**이면 자식 컨트롤을 도킹할 수 있도록 승인할 때까지 도킹 사이트의 크기가 0이 됩니다. 그런 다음 자식 컨트롤에 맞게 크기를 다시 조정합니다.

컨트롤을 도킹 가능한 자식으로 만들기

다음과 같은 방법으로 컨트롤을 도킹 가능한 자식 컨트롤로 만듭니다.

1 해당 *DragKind* 속성을 *dkDock*으로 설정합니다. *DragKind*가 *dkDock*인 경우 컨트롤을 끌면 새 도킹 사이트로 컨트롤이 이동하거나 컨트롤이 해제되어 이동식 윈도우가 됩니다. *DragKind*가 *dkDrag*(기본값)인 경우 컨트롤을 끌면 *OnDragOver*, *OnEndDrag* 및 *OnDragDrop* 이벤트를 사용하여 구현해야 할 드래그 앤 드롭 작업이 시작됩니다.

2 해당 *DragMode*를 *dmAutomatic*으로 설정합니다. *DragMode*가 *dmAutomatic*인 경우에는 사용자가 마우스로 컨트롤을 끌기 시작할 때 끌기(*DragKind*에 따라 드래그 앤 드롭 또는 도킹)가 자동으로 시작됩니다. *DragMode*가 *dmManual*인 경우에도 계속 *BeginDrag* 메소드를 호출하여 드래그 앤 도킹 또는 드래그 앤 드롭 작업을 시작할 수 있습니다.

3 도킹이 해제되거나 이동식 윈도우로 남아 있는 경우 컨트롤을 호스트해야 하는 *TWinControl* 자손을 나타내도록 해당 *FloatingDockSiteClass* 속성을 설정합니다. 컨트롤이 릴리스되고 도킹 사이트 위에 있지 않으면 이 클래스의 윈도우 컨트롤이 동적으로 만들어져서 도킹 가능한 자식의 부모가 됩니다. 도킹 가능한 자식 컨트롤이 *TWinControl* 의 자손이면 테두리 또는 제목 표시줄을 만들 폼을 지정하려는 경우에도 컨트롤을 호스트할 각각의 부동 도킹 사이트를 만들 필요가 없습니다. 동적 컨테이너 윈도우를 생략하려면 *FloatingDockSiteClass*를 컨트롤과 같은 클래스로 설정하십시오. 그러면 부모가 없는 이동식 윈도우가 됩니다.

자식 컨트롤의 도킹 방법 제어

도킹 사이트에서 자식 컨트롤을 릴리스하면 도킹 사이트에서 자동으로 이 컨트롤을 승인합니다. 대부분의 컨트롤은 첫 번째 자식이 클라이언트 영역에 맞게 도킹되고 두 번째 자식부터는 이 영역을 각각의 영역으로 분할합니다. 페이지 컨트롤은 자식을 새 탭 시트에 도킹시키거나 자식이 다른 페이지 컨트롤인 경우 탭 시트에 병합시킵니다.

세 가지 이벤트를 통해 도킹 사이트에 자식 컨트롤을 도킹시키는 방법을 제한할 수 있습니다.

```
__property TGetSiteInfoEvent OnGetSiteInfo = {read=FOnGetSiteInfo,
write=FOnGetSiteInfo};
typedef void __fastcall (__closure *TGetSiteInfoEvent)(System::TObject*
Sender, TControl* DockClient, Windows::TRect &InfluenceRect, const
Windows::TPoint &MousePos, bool &CanDock);
```

사용자가 도킹 가능한 자식을 컨트롤 위로 끌면 도킹 사이트에서 *OnGetSiteInfo*가 발생합니다. 이를 통해 사이트에서 *DockClient* 매개변수에서 지정한 컨트롤을 자식으로 승인할지 여부를 나타낼 수 있고, 이 경우 자식의 위치가 도킹된 것으로 간주되어야 합니다. *OnGetSiteInfo*가 발생하면 *InfluenceRect*가 도킹 사이트의 화면 좌표로 초기화되고 *CanDock*가 **true**로 초기화됩니다. *InfluenceRect*를 변경하여 도킹 영역을 더 제한할 수 있으며 *CanDock*를 **false**로 설정하여 자식이 거부되게 할 수 있습니다.

```
__property TDockOverEvent OnDockOver = {read=FOnDockOver,
write=FOnDockOver};
typedef void __fastcall (__closure *TDockOverEvent)(System::TObject*
Sender, TDragDockObject* Source, int X, int Y, TDragState State, bool
&Accept);
```

*OnDockOver*는 사용자가 도킹 가능한 자식을 컨트롤로 끌 때 도킹 사이트에서 발생합니다. 이는 *OnDragOver* 이벤트의 드래그 앤 드롭 작업과 유사합니다. 이 이벤트의 *Accept* 매개변수를 설정하면 자식을 릴리스하여 도킹할 수 있다는 것을 알릴 수 있습니다. 잘못된 컨트롤 타입 등의 이유로 *OnGetSiteInfo* 이벤트 핸들러에서 도킹 가능한 컨트롤을 거부하면 *OnDockOver*가 발생하지 않습니다.

```
__property TDockDropEvent OnDockDrop = {read=FOnDockDrop,
write=FOnDockDrop};
typedef void __fastcall (__closure *TDockDropEvent)(System::TObject*
Sender, TDragDockObject* Source, int X, int Y);
```

사용자가 도킹 가능한 자식을 컨트롤에서 릴리스하면 도킹 사이트에서 *OnDockDrop*이 발생합니다. 이는 *OnDragDrop* 이벤트의 일반적인 드래그 앤 드롭 작업과 동일합니다. 이 이벤트를 사용하여 컨트롤을 자식 컨트롤로 승인하는 데 필요한 작업을 수행합니다. 자식 컨트롤에 액세스하려면 *Source* 매개변수에서 지정한 *TDockObject*의 *Control* 속성을 사용할 수 있습니다.

자식 컨트롤의 도킹 해제 방법 제어

도킹 사이트에서는 *dmAutomatic*의 *DragMode* 속성이 있는 자식 컨트롤을 끌 때 자동으로 도킹이 해제되게 할 수 있습니다. 도킹 사이트는 자식 컨트롤을 끌어낼 때 응답할 수 있으며 *OnUnDock* 이벤트 핸들러에서 도킹을 해제하지 못하게 할 수 있습니다.

```
__property TUnDockEvent OnUnDock = {read=FOnUnDock, write=FOnUnDock};
typedef void __fastcall (__closure *TUnDockEvent)(System::TObject*
Sender, TControl* Client, TWinControl* NewTarget, bool &Allow);
```

Client 매개변수는 도킹을 해제하려는 자식 컨트롤을 나타내며 *Allow* 매개변수를 통해 도킹 사이트(*Sender*)에서 도킹 해제를 거부할 수 있습니다. *OnUnDock* 이벤트 핸들러를 구현할 때 다른 자식이 현재 도킹되어 있는지 알면 유용할 수 있습니다. 이 정보는 *TControl*의 인덱싱된 배열인 읽기 전용 *DockClients* 속성에서 사용할 수 있습니다. 도킹 클라이언트 수는 읽기 전용 *DockClientCount* 속성에서 지정합니다.

드래그 앤 도킹 작업에 대한 자식 컨트롤의 응답 방법 제어

도킹 가능한 자식 컨트롤에서는 드래그 앤 도킹 작업 중 두 가지 이벤트가 발생할 수 있습니다. *OnStartDock*는 드래그 앤 드롭 작업의 *OnStartDrag* 이벤트와 같이, 도킹 가능한 자식 컨트롤이 사용자 정의 끌기 객체를 만들 수 있게 합니다. *OnEndDock*는 *OnEndDrag*처럼 끌기가 종료될 때 발생합니다.

컨트롤 내에서 텍스트 작업

다음 단원에서는 다양한 편집 및 메모 컨트롤 기능 사용 방법을 설명합니다. 이 기능 중 일부는 편집 컨트롤에서도 사용합니다.

- 텍스트 정렬 설정
- 런타임 시 스크롤 막대 추가
- 클립보드 객체 추가
- 텍스트 선택
- 모든 텍스트 선택
- 텍스트 잘라내기, 복사 및 붙여넣기
- 선택한 텍스트 삭제
- 메뉴 항목 비활성화
- 팝업 메뉴 제공
- OnPopup 이벤트 처리

텍스트 정렬 설정

리치 에디트(rich edit)나 메모 컴포넌트에서 텍스트는 왼쪽, 오른쪽 또는 가운데로 정렬할 수 있습니다. 텍스트 정렬을 변경하려면 편집 컴포넌트의 *Alignment* 속성을 설정하십시오. 이 속성은 *WordWrap* 속성이 **true**일 때에만 적용되며 자동 줄 바꿈 처리가 해제되어 있으면 정렬할 여백이 없습니다.

예를 들어, RichEdit 예제의 다음 코드는 선택한 버튼에 따라 정렬을 설정합니다.

```
switch((int)RichEdit1->Paragraph->Alignment)
{
  case 0: LeftAlign->Down   = true; break;
  case 1: RightAlign->Down  = true; break;
  case 2: CenterAlign->Down = true; break;
}
```

런타임 시 스크롤 막대 추가

필요한 경우 리치 에디트와 메모 컴포넌트에 수평 및/또는 수직 스크롤 막대를 포함시킬 수 있습니다. 자동 줄 바꿈 기능이 설정되어 있으면 컴포넌트에 수직 스크롤 막대만 필요합니다. 사용자가 자동 줄 바꿈 기능을 해제하면 텍스트가 에디터의 왼쪽 끝에 의해 제한되지 않으므로 수평 스크롤 막대도 필요합니다.

다음과 같은 방법으로 런타임 시 스크롤 막대를 추가합니다.

1. 텍스트가 오른쪽 여백을 초과하는지 확인합니다. 대부분의 경우 이것은 자동 줄 바꿈 처리의 설정 여부 확인을 의미합니다. 또한 텍스트 행이 실제로 컨트롤의 너비를 초과하는지 여부를 확인할 수도 있습니다.
2. 리치 에디트 또는 메모 컴포넌트의 *ScrollBars* 속성에 스크롤 막대를 포함할지 또는 제외할지를 설정합니다.

다음 예제에서는 *OnClick* 이벤트 핸들러를 Character|WordWrap 메뉴 항목에 추가합니다.

```
void __fastcall TEditForm::WordWrap1Click(TObject *Sender)
{
  Editor->WordWrap = !(Editor->WordWrap);   // toggle word wrapping
  if (Editor->WordWrap)
    Editor->ScrollBars = ssVertical;        // wrapped requires only
vertical
  else
    Editor->ScrollBars = ssBoth;            // unwrapped can need both
  WordWrap1->Checked = Editor->WordWrap;    // check menu item to match
property
}
```

리치 에디트와 메모 컴포넌트는 스크롤 막대를 약간 다른 방법으로 처리합니다. 리치 에디트 컴포넌트는 텍스트가 컴포넌트 경계 내에 모두 포함될 경우 스크롤 막대를 숨길 수 있습니다. 스크롤 막대가 활성화되어 있으면 메모는 항상 스크롤 막대를 표시합니다.

클립보드 객체 추가

텍스트를 다루는 대부분의 애플리케이션은 다른 애플리케이션의 문서를 비롯해서 문서 간에 선택한 텍스트를 이동하는 방법을 제공합니다. C++Builder의 *Clipboard* 객체는 Windows 클립보드와 같은 클립보드를 캡슐화하고 텍스트(그래픽 등의 다른 형식도 포함)를 잘라내고 복사하고 붙여넣기 위한 메소드를 포함합니다. *Clipboard* 객체는 *Clipbrd* 유닛에 선언되어 있습니다.

다음과 같은 방법으로 *Clipboard* 객체를 애플리케이션에 포함시킵니다.

1 클립보드를 사용할 유닛을 선택합니다.

2 폼의 .h 파일에 다음을 추가합니다.

```
#include <vcl\Clipbrd.hpp>
```

텍스트 선택

편집 컨트롤의 텍스트는 텍스트를 클립보드로 보내기 전에 선택해야 합니다. 선택한 텍스트의 강조 기능은 편집 컴포넌트에 내장되어 있습니다. 사용자가 텍스트를 선택하면 강조되어 나타납니다.

표 6.1은 선택한 텍스트를 처리하는 데 자주 사용되는 속성을 나열한 것입니다.

표 6.1 선택한 텍스트의 속성

속성	설명
SelText	컴포넌트 내에서 선택된 텍스트를 나타내는 문자열이 들어 있습니다.
SelLength	선택한 문자열의 길이가 들어 있습니다.
SelStart	편집 컨트롤의 텍스트 버퍼 시작과 관련된 시작 위치가 들어 있습니다.

모든 텍스트 선택

SelectAll 메소드는 리치 에디트(rich edit)나 메모 컴포넌트 등 편집 컨트롤의 모든 내용을 선택합니다. 이 메소드는 컴포넌트의 내용이 컴포넌트의 가시 영역을 벗어날 때 특히 유용합니다. 대부분의 다른 경우에서는 사용자가 키스트로크나 마우스를 끌어 텍스트를 선택합니다.

리치 에디트나 메모 컨트롤의 전체 내용을 선택하려면 *RichEdit1* 컨트롤의 *SelectAll* 메소드를 호출하십시오.

예를 들면, 다음과 같습니다.

```
void __fastcall TMainForm::SelectAll(TObject *Sender)
{
  RichEdit1->SelectAll();     // select all text in RichEdit
}
```

텍스트 잘라내기, 복사 및 붙여넣기

Clipbrd 유닛을 사용하는 애플리케이션은 클립보드를 통해 텍스트, 그래픽 및 객체를 잘라내고 복사하고 붙여넣을 수 있습니다. 표준 텍스트 처리 컨트롤을 캡슐화하는 편집 컴포넌트는 모두 클립보드와 상호 작용할 수 있도록 내장되어 있습니다. 클립보드에서 그래픽을 사용하는 방법에 대한 자세한 내용은 10-22페이지의 "그래픽에서 클립보드 사용"을 참조하십시오.

클립보드로 텍스트를 잘라내거나 복사하거나 붙여넣으려면 편집 컴포넌트의 *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 각각 호출합니다.

예를 들면, 다음 코드는 Edit|Cut, Edit|Copy 및 Edit|Paste 명령의 *OnClick* 이벤트에 각각 이벤트 핸들러를 추가합니다.

```
void __fastcall TMainForm::EditCutClick(TObject* Sender)
{    RichEdit1->CutToClipboard();
}
void __fastcall TMainForm::EditCopyClick(TObject* Sender)
{    RichEdit1->CopyToClipboard();
}
void __fastcall TMainForm::EditPasteClick(TObject* Sender)
{    RichEdit1->PasteFromClipboard();
}
```

선택한 텍스트 삭제

클립보드로 잘라내지 않고 편집 컴포넌트에서 선택한 텍스트를 삭제할 수 있습니다. 이렇게 하려면 *ClearSelection* 메소드를 호출하십시오. 예를 들어, Edit 메뉴에 Delete 항목이 있으면 코드가 다음과 같을 수 있습니다.

```
void __fastcall TMainForm::EditDeleteClick(TObject *Sender)
{
  RichEdit1->ClearSelection();
}
```

메뉴 항목 비활성화

메뉴에서 메뉴 명령을 제거하지 않고 비활성화하면 편리할 때가 있습니다. 예를 들어, 텍스트 에디터에서 현재 선택한 텍스트가 없으면 Cut, Copy 및 Delete 명령을 사용할 수 없습니다. 메뉴 항목은 사용자가 메뉴를 선택할 때 활성화 또는 비활성화되는 것이 좋습니다. 메뉴 항목을 비활성화하려면 해당 *Enabled* 속성을 **false**로 설정하십시오.

다음 예제에서는 이벤트 핸들러가 자식 폼의 메뉴 모음에 있는 Edit 항목의 *OnClick* 이벤트에 추가됩니다. *RichEdit1*에 선택한 텍스트가 있는지 여부에 따라 Edit 메뉴의 Cut, Copy 및 Delete 메뉴 항목에 *Enabled*가 설정됩니다. Paste 명령은 클립보드에 텍스트가 있는지 여부에 따라 활성화 또는 비활성화됩니다.

```
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  bool HasSelection = (RichEdit1->SelLength > 0);  // true if text is
selected
```

```
    Cut1->Enabled = HasSelection;  // enable menu items if HasSelection is
  true
    Copy1->Enabled = HasSelection;
    Delete1->Enabled = HasSelection;
  }
```

클립보드의 *HasFormat* 메소드는 클립보드에 특정 형식의 객체, 텍스트 또는 이미지가 있는지 여부에 따라 부울 값을 반환합니다. 매개변수 *CF_TEXT*가 있는 *HasFormat*을 호출하면 클립보드에 텍스트가 있는지 알 수 있으며 이에 따라 Paste 항목을 적절히 활성화 또는 비활성화할 수 있습니다.

클립보드에서 그래픽을 사용하는 데 대한 자세한 내용은 10장, "그래픽 및 멀티미디어 작업"을 참조하십시오.

팝업 메뉴 제공

팝업 또는 로컬 메뉴는 모든 애플리케이션에서 사용하기 쉬운 일반적인 기능입니다. 이 기능을 사용하면 애플리케이션 작업 영역에서 마우스 오른쪽 단추를 클릭함으로써 마우스 움직임을 최소화하여 자주 사용하는 명령 리스트에 액세스할 수 있습니다.

예를 들어, 텍스트 에디터 애플리케이션에서 Cut, Copy 및 Paste 편집 명령을 반복하는 팝업 메뉴를 추가할 수 있습니다. 이 팝업 메뉴 항목에서 Edit 메뉴의 해당 항목과 동일한 이벤트 핸들러를 사용할 수 있습니다. 일반적으로 해당 메뉴 항목에 이미 바로 가기가 있으므로 팝업 메뉴에 대한 가속키나 단축키를 만들 필요가 없습니다.

폼의 *PopupMenu* 속성은 사용자가 폼의 항목을 마우스 오른쪽 단추로 클릭할 때 표시할 팝업 메뉴를 지정합니다. 또한 각 컨트롤에는 특정 컨트롤에 대해 사용자 정의 메뉴를 허용하는 폼의 속성을 오버라이드할 수 있는 *PopupMenu* 속성이 있습니다.

다음과 같은 방법으로 폼에 팝업 메뉴를 추가합니다.

1. 폼에 팝업 메뉴 컴포넌트를 놓습니다.
2. Menu Designer를 사용하여 팝업 메뉴의 항목을 정의합니다.
3. 메뉴를 팝업 메뉴 컴포넌트의 이름에 표시하는 폼이나 컨트롤의 *PopupMenu* 속성을 설정합니다.
4. 팝업 메뉴 항목의 *OnClick* 이벤트에 핸들러를 추가합니다.

OnPopup 이벤트 처리

일반 메뉴에서 항목을 활성화 또는 비활성화하는 것처럼 메뉴를 표시하기 전에 팝업 메뉴 항목을 조정할 수 있습니다. 6-9페이지의 "메뉴 항목 비활성화"에 설명된 것처럼 일반 메뉴에서 메뉴 상단의 항목에서 *OnClick* 이벤트를 처리할 수 있습니다.

그러나 팝업 메뉴의 경우 최상위 레벨 메뉴 모음이 없으므로 팝업 메뉴 명령을 준비하려면 메뉴 컴포넌트 자체에서 이벤트를 처리해야 합니다. 팝업 메뉴 컴포넌트는 이러한 용도만을 위해 *OnPopup*이라는 이벤트를 제공합니다.

다음과 같은 방법으로 메뉴 항목을 표시하기 전에 팝업 메뉴의 항목을 조정합니다.

1 팝업 메뉴 컴포넌트를 선택합니다.

2 해당 *OnPopup* 이벤트에 이벤트 핸들러를 추가합니다.

3 이벤트 핸들러에 메뉴 항목을 활성화, 비활성화, 숨기기 또는 표시하는 코드를 작성합니다.

다음 코드에서는 앞의 6-9페이지의 "메뉴 항목 비활성화"에서 설명되었던 *Edit1Click* 이벤트 핸들러가 팝업 메뉴 컴포넌트의 *OnPopup* 이벤트에 추가됩니다. 한 줄의 코드가 팝업 메뉴에 있는 각 항목에 대한 *Edit1Click*에 추가됩니다.

```
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  Paste2->Enabled = Paste1->Enabled;            // add this line
  bool HasSelection = (RichEdit1->SelLength > 0);  // true if text is
selected
  Cut1->Enabled = HasSelection;    // enable menu items if HasSelection
is true
  Cut2->Enabled = HasSelection;    // add this line
  Copy1->Enabled = HasSelection;
  Copy2->Enabled = HasSelection;   // add this line
  Delete1->Enabled = HasSelection;
}
```

컨트롤에 그래픽 추가

여러 가지 컨트롤을 통해 컨트롤이 렌더링되는 방법을 사용자 정의할 수 있습니다. 이러한 컨트롤에는 리스트 박스, 콤보 박스, 메뉴, 헤더, 탭 컨트롤, 리스트 뷰, 상태 표시줄, 트리 뷰 및 툴바가 있습니다. 컨트롤이나 해당 항목을 그리는 표준 메소드를 사용하지 않고 컨트롤의 소유자(일반적으로 폼)가 런타임 시 항목을 그립니다. owner-draw 컨트롤을 사용하는 가장 일반적인 방법은 항목의 텍스트 외에 그래픽을 제공하는 것입니다. owner-draw를 사용하여 메뉴에 이미지를 추가하는 방법에 대한 자세한 내용은 8-35페이지의 "메뉴 항목에 이미지 추가"를 참조하십시오.

모든 owner-draw 컨트롤에는 항목 리스트가 있습니다. 일반적으로 이러한 리스트는 텍스트로 표시되는 문자열 리스트거나 텍스트로 표시되는 문자열을 포함한 객체 리스트입니다. 객체를 리스트의 각 항목과 연결하여 항목을 그릴 때 해당 객체를 쉽게 사용할 수 있습니다.

일반적으로 C++Builder에서 owner-draw 컨트롤을 작성하려면 다음과 같은 단계가 필요합니다.

1 컨트롤이 owner-draw 항목임을 나타내기

2 문자열 리스트에 그래픽 객체 추가

3 owner-draw 항목 그리기

컨트롤이 owner-draw 항목임을 나타내기

컨트롤 그리기를 사용자 정의하려면 색칠할 때 컨트롤의 이미지를 렌더링하는 이벤트 핸들러를 제공해야 합니다. 일부 컨트롤은 이러한 이벤트를 자동으로 받습니다. 예를 들어 리스트 뷰, 트리 뷰 및 툴바는 속성을 전혀 설정하지 않아도 그리기 프로세스의 다양한 단계에서 이벤트를 받습니다. 이러한 이벤트는 "OnCustomDraw" 또는 "OnAdvancedCustomDraw"와 같은 이름을 가집니다.

그러나 다른 컨트롤은 속성을 설정해야만 owner draw 이벤트를 받습니다. 리스트 박스, 콤보 박스, 헤더 컨트롤 및 상태 표시줄에는 *Style*이라는 속성이 있습니다. *Style*은 컨트롤에 "표준" 스타일인 default drawing을 사용할지 owner drawing을 사용할지 여부를 결정합니다. 그리드는 *DefaultDrawing* 속성을 사용하여 default drawing을 활성화하거나 비활성화합니다. 리스트 뷰와 탭 컨트롤에는 default drawing을 활성화하거나 비활성화하는 *OwnerDraw* 속성이 있습니다.

리스트 박스와 콤보 박스에는 표 6.2에서 설명하는 것처럼 *fixed*와 *variable*이라는 추가 owner-draw 스타일이 있습니다. 텍스트가 들어 있는 항목의 크기는 다양하지만 다른 컨트롤은 항상 고정되어 있으며 각 항목의 크기는 컨트롤을 그리기 전에 결정됩니다.

표 6.2 Fixed 및 Variable owner-draw 스타일

owner-draw 스타일	의미	예
Fixed	각 항목은 *ItemHeight* 속성에서 지정된 동일한 높이를 갖습니다.	*lbOwnerDrawFixed*, *csOwnerDrawFixed*
Variable	각 항목은 런타임 시 데이터에 의해 지정된 다른 높이를 가질 수도 있습니다.	*lbOwnerDrawVariable*, *csOwnerDrawVariable*

문자열 리스트에 그래픽 객체 추가

각 문자열 리스트에는 문자열 리스트뿐 아니라 객체 리스트도 포함될 수 있습니다.

예를 들어, 파일 관리자 애플리케이션에 드라이브 문자와 드라이브 타입을 나타내는 비트맵을 추가할 수 있습니다. 이렇게 하려면 다음 단원의 설명에 따라 애플리케이션에 비트맵 이미지를 추가한 다음 이 이미지를 해당 위치로 복사해야 합니다.

애플리케이션에 이미지 추가

이미지 컨트롤은 비트맵과 같은 그래픽 이미지를 포함하는 넌비주얼(nonvisual) 컨트롤입니다. 이미지 컨트롤을 사용하여 폼에 그래픽 이미지를 표시할 수 있습니다. 또한 애플리케이션에서 사용할 숨겨진 이미지를 포착하는 데에도 이미지 컨트롤을 사용할 수 있습니다. 예를 들면, 다음과 같이 숨겨진 이미지 컨트롤에 owner-draw 컨트롤의 비트맵을 저장할 수 있습니다.

1. 메인 폼에 이미지 컨트롤을 추가합니다.
2. *Name* 속성을 설정합니다.
3. 각 이미지 컨트롤의 *Visible* 속성을 **false**로 설정합니다.
4. Object Inspector의 Picture Editor를 사용하여 각 이미지의 *Picture* 속성을 원하는 비트맵으로 설정합니다.

애플리케이션을 실행하면 이미지 컨트롤은 보이지 않습니다.

문자열 리스트에 이미지 추가

애플리케이션에 그래픽 이미지가 있으면 이 이미지를 문자열 리스트에 있는 문자열에 연결할 수 있습니다. 문자열과 동시에 객체를 추가하거나 기존 문자열에 객체를 연결할 수 있습니다. 필요한 데이터를 모두 사용할 수 있으면 객체와 문자열을 동시에 추가하는 방법이 더 많이 사용됩니다.

다음 예제는 문자열 리스트에 이미지를 추가하는 방법을 보여 줍니다. 이 예제는 적절한 각각의 드라이브 문자와 함께 각 드라이브 타입을 나타내는 비트맵을 추가하는 파일 관리자 애플리케이션의 일부입니다. *OnCreate* 이벤트 핸들러는 다음과 같습니다.

```
void __fastcall TFMForm::FormCreate(TObject *Sender)
{
  int AddedIndex;
  char DriveName[4] = "A:\\";
  for (char Drive = 'A'; Drive <= 'Z'; Drive++) // try all possible
drives
  {
    DriveName[0] = Drive;
    switch (GetDriveType(DriveName))
    {
      case DRIVE_REMOVABLE:// add a list item
        DriveName[1] = '\0'; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Floppy->Picture->Graphic);
        DriveName[1] = ':' // replace the colon
        break;
      case DRIVE_FIXED:// add a list item
        DriveName[1] = '\0'; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Fixed->Picture->Graphic);
        DriveName[1] = ':' // replace the colon
        break;
      case DRIVE_REMOTE:// add a list item
        DriveName[1] = '\0'; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Network->Picture->Graphic);
        DriveName[1] = ':' // replace the colon
        break;
    }
    if ((int)(Drive - 'A') == getdisk()) // current drive?
      DriveList->ItemIndex = AddedIndex; // then make that the current
list item
  }
}
```

owner-draw 항목 그리기

속성을 설정하거나 사용자 정의 그리기 이벤트 핸들러를 제공하여 컨트롤이 owner-drawn이라는 것을 나타내면 컨트롤이 더 이상 화면에 나타나지 않습니다. 대신 운영 체제에서 컨트롤의 각 가시적 항목에 대해 이벤트를 생성합니다. 애플리케이션은 이벤트를 처리하여 항목을 그립니다.

owner-draw 컨트롤에 항목을 그리려면 컨트롤에서 보이는 각 항목에 대해 다음을 수행하십시오. 모든 항목에 대해 하나의 이벤트 핸들러를 사용합니다.

1 필요에 따라 항목의 크기를 조정합니다.

   항목의 크기가 같으면(예를 들어, *lsOwnerDrawFixed* 스타일의 리스트 박스) 크기를 조정하지 않아도 됩니다.

2 항목을 그립니다.

owner-draw 항목의 크기 지정

애플리케이션이 다양한 owner-draw 컨트롤에 각 항목을 그릴 수 있게 하기 전에 운영 체제는 measure-item 이벤트를 생성합니다. measure-item 이벤트는 컨트롤에서 항목이 나타나는 위치를 애플리케이션에 알립니다.

C++Builder는 항목의 크기를 결정합니다. 일반적으로 현재 글꼴로 항목의 텍스트를 표시하기에는 충분할 정도의 크기를 지정합니다. 애플리케이션은 이벤트를 처리하고 선택된 사각형을 변경할 수 있습니다. 예를 들어, 항목 텍스트의 비트맵을 대체하려면 사각형을 비트맵 크기로 변경합니다. 비트맵과 텍스트를 표시하려면 모두 표시되도록 사각형의 크기를 조정합니다.

owner-draw 항목의 크기를 변경하려면 owner-draw 컨트롤 내의 measure-item 이벤트에 이벤트 핸들러를 추가합니다. 컨트롤에 따라 이벤트의 이름이 다를 수 있습니다. 리스트 박스와 콤보 박스는 *OnMeasureItem*을 사용합니다. 그리드에는 measure-item 이벤트가 없습니다.

크기 조정 이벤트에는 항목의 인덱스 번호와 항목 크기라는 두 가지 중요한 매개변수가 있습니다. 크기는 다양하며 애플리케이션에서 크기를 조정할 수 있습니다. 연속되는 항목의 위치는 선행 항목의 크기에 따라 다릅니다.

예를 들어, variable owner-draw 리스트 박스에서 애플리케이션이 첫 항목의 높이를 5픽셀로 설정하면 두 번째 항목은 위로부터 6픽셀 아래에서 시작합니다. 리스트 박스와 콤보 박스에서 애플리케이션은 항목의 높이만 변경할 수 있습니다. 항목의 너비는 항상 컨트롤의 너비입니다.

owner-draw 그리드는 그릴 때 셀 크기를 변경할 수 없습니다. 각 행과 열의 크기는 그리기 전에 *ColWidths* 및 *RowHeights* 속성에 의해 설정됩니다.

owner-draw 리스트 박스의 *OnMeasureItem* 이벤트에 추가된 다음 코드는 연결된 비트맵에 맞추기 위해 각 리스트 항목의 높이를 늘립니다.

```
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control, int
Index,
  int &Height)      // note that Height is passed by reference
{
  int BitmapHeight = ((TBitmap *)ListBox1->Items->Objects[Index])->Height
+ 2;
  // make sure list item has enough room for bitmap (plus 2)
  if (BitmapHeight > Height)
    Height = BitmapHeight;
}
```

참고 문자열 리스트에 있는 *Objects* 속성의 항목을 타입 변환해야 합니다. *Objects*는 *TObject* 타입의 속성으로 모든 형식의 객체를 포함할 수 있습니다. 배열에서 객체를 검색할 때 객체 타입을 항목의 실제 타입으로 변환해야 합니다.

owner-draw 항목 그리기

애플리케이션이 owner-draw 컨트롤을 그리거나 다시 그릴 때 Windows는 각 컨트롤의 각 가시적 항목에 대해 draw-item 이벤트를 생성합니다. 컨트롤에 따라 항목은 전체 또는 하위 항목으로 해당 항목의 그리기 이벤트를 받을 수도 있습니다.

owner-draw 컨트롤에서 각 항목을 그리려면 해당 컨트롤의 draw-item 이벤트에 이벤트 핸들러를 추가합니다.

일반적으로 owner drawing의 이벤트 이름은 다음 중 하나로 시작합니다.

- *OnDrawItem*이나 *OnDrawCell*의 경우처럼 *OnDraw*로 시작
- *OnCustomDrawItem*의 경우처럼 *OnCustomDraw*로 시작
- *OnCustomDrawItem*의 경우처럼 *OnCustomDraw*로 시작

draw-item 이벤트에는 그릴 항목을 식별하는 매개변수, 그려 넣을 사각형 및 항목에 포커스가 있는지 여부와 같은 항목의 상태에 관한 몇 가지 정보가 들어 있습니다. 애플리케이션은 주어진 사각형 내에서 적절한 항목을 렌더링하여 각 이벤트를 처리합니다.

예를 들어, 다음 코드는 각 문자열과 연결된 비트맵이 있는 리스트 박스에 항목을 그리는 방법을 표시합니다. 다음과 같이 리스트 박스의 *OnDrawItem* 이벤트에 이 핸들러를 추가합니다.

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control, int Index,
  TRect &Rect, TOwnerDrawState State)

  TBitmap *Bitmap = (TBitmap *)ListBox1->Items->Objects[Index];
  ListBox1->Canvas->Draw(R.Left, R.Top + 2, Bitmap); // draw the bitmap
  ListBox1->Canvas->TextOut (R.Left + Bitmap->Width + 2, R.Top + 2,
    ListBox1->Items->Strings[Index]);        // and write the text to its
right
}
```

장

7

애플리케이션, 컴포넌트 및 라이브러리 생성

이 장에서는 C++Builder를 사용하여 애플리케이션, 라이브러리 및 컴포넌트를 생성하는 방법에 대한 개요를 제공합니다.

애플리케이션 생성

C++Builder는 주로 다음과 같은 유형의 애플리케이션을 디자인하고 생성하는 데 사용합니다.

- GUI 애플리케이션
- 콘솔 애플리케이션
- 서비스 애플리케이션(Windows 애플리케이션만 해당)
- 패키지 및 DLL

GUI 애플리케이션은 일반적으로 사용하기 쉬운 인테페이스를 가집니다. 콘솔 애플리케이션은 콘솔 윈도우에서 실행됩니다. 서비스 애플리케이션은 Windows 서비스로 실행됩니다. 이런 유형의 애플리케이션은 시작 코드가 있는 실행 파일로 컴파일됩니다.

패키지 및 DLL과 같이 패키지나 동적으로 연결할 수 있는 라이브러리를 생성하는 다른 프로젝트 타입을 만들 수도 있습니다. 이 애플리케이션은 시작 코드 없이 실행할 수 있는 코드를 만듭니다. 자세한 내용은 7-10페이지의 "패키지 및 DLL 생성"을 참조하십시오.

GUI 애플리케이션

그래픽 사용자 인터페이스(GUI) 애플리케이션은 윈도우, 메뉴, 다이얼로그 박스 등과 같은 그래픽 기능과 애플리케이션 사용을 쉽게 하는 기능을 사용하여 디자인합니다. GUI 애플리케이션을 컴파일하면 시작 코드가 있는 실행 파일이 만들어집니다. 실행 파일은 보통 사용자 프로그램의 기본적인 기능을 제공하며, 간단한 프로그램은 대체로 실행 파일 하나로만 구성됩니다.

실행 파일로부터 DLL, 패키지 및 다른 지원 파일을 호출하여 애플리케이션을 확장할 수 있습니다.

C++Builder는 다음과 같은 두 가지 애플리케이션 UI 모델을 제공합니다.

- 단일 문서 인터페이스(SDI)
- 다중 문서 인터페이스(MDI)

애플리케이션의 구현 모델과 함께 프로젝트의 디자인 타임 동작 및 애플리케이션의 런타임 동작은 IDE에서 프로젝트 옵션을 설정하여 처리할 수 있습니다.

사용자 인터페이스 모델

폼은 단일 문서 인터페이스(SDI)나 다중 문서 인터페이스(MDI) 폼으로 구현할 수 있습니다. MDI 애플리케이션인 경우 하나의 부모 윈도우 내에서 둘 이상의 문서나 자식 윈도우를 열 수 있습니다. MDI는 스프레드시트나 워드 프로세서 같은 애플리케이션에서는 일반적인 인터페이스입니다. 반면에 SDI 애플리케이션은 일반적으로 단일 문서 뷰를 포함합니다. 폼을 SDI 애플리케이션으로 만들려면 *Form* 객체의 *FormStyle* 속성을 *fsNormal*로 설정합니다.

애플리케이션의 UI 개발에 대한 자세한 내용은 8장, "애플리케이션 사용자 인터페이스 개발"을 참조하십시오.

SDI 애플리케이션

다음과 같은 방법으로 새 SDI 애플리케이션을 만듭니다.

1 File|New|Other를 선택하여 New Items 다이얼로그 박스를 표시합니다.

2 Projects 페이지를 클릭하고 SDI Application을 더블 클릭합니다.

3 OK를 클릭합니다.

디폴트로 *Form* 객체의 *FormStyle* 속성은 *fsNormal* 로 설정되므로 C++Builder 는 모든 새 애플리케이션이 SDI 애플리케이션인 것으로 간주합니다.

MDI 애플리케이션

다음과 같은 방법으로 MDI 애플리케이션을 만듭니다.

1 File|New|Other를 선택하여 New Items 다이얼로그 박스를 표시합니다.

2 Projects 페이지를 클릭하고 MDI Application을 더블 클릭합니다.

3 OK를 클릭합니다.

MDI 애플리케이션은 SDI 애플리케이션보다 디자인하기가 다소 복잡하며 더 많은 계획이 필요합니다. MDI 애플리케이션은 클라이언트 윈도우 내에 있는 자식 윈도우를 만듭니다. 즉, 메인 폼은 자식 폼을 포함합니다. *TForm* 객체의 *FormStyle* 속성을 설정하여 폼이 자식(*fsMDIChild*)인지 메인 폼(*fsMDIForm*)인지를 지정합니다. 자식 폼을 위한 기본 클래스를 정의하고 이 클래스에서 각각의 자식 폼을 파생하는 것이 좋습니다. 이렇게 하면 자식 폼의 속성을 다시 설정할 필요가 없습니다.

MDI 애플리케이션은 메인 메뉴에 윈도우 팝업을 포함하며, 여기에는 여러 윈도우를 다양한 스타일로 표시하기 위한 Cascade, Tile 등과 같은 항목이 들어 있습니다. 자식 윈도우를 최소화하면 그 아이콘은 MDI 부모 폼에 표시됩니다.

MDI 애플리케이션을 위한 윈도우 작성에 필요한 작업을 요약하면 다음과 같습니다.

1 메인 윈도우 폼이나 MDI 부모 윈도우를 만듭니다. *FormStyle* 속성을 *fsMDIForm*으로 설정합니다.

2 메인 윈도우의 메뉴를 만들고 Cascade, Tile, Arrange All 항목 등이 있는 윈도우와 File|Open 및 File|Save를 메뉴에 포함시킵니다.

3 MDI 자식 폼을 생성하고 해당 *FormStyle* 속성을 *fsMDIChild*로 설정합니다.

IDE, 프로젝트 및 컴파일 옵션 설정

Project|Options를 선택하여 프로젝트에 대해 다양한 옵션을 지정합니다. 자세한 내용은 온라인 도움말을 참조하십시오.

디폴트 프로젝트 옵션 설정

이후의 모든 프로젝트에 적용되는 디폴트 옵션을 변경하려면 Project Options 다이얼로그 박스에서 윈도우 오른쪽 아래에 있는 Default 체크 박스에 선택 표시를 합니다. 모든 새 프로젝트는 디폴트로 선택된 현재 옵션을 사용하게 됩니다.

프로그래밍 템플릿

프로그래밍 템플릿은 일반적으로 소스 코드에 추가한 다음 채워 넣을 수 있는 *스켈레톤* 구조로 사용됩니다. 배열, 클래스, 함수 선언 및 많은 명령문 등에 사용하는 템플릿과 같은 몇몇 표준 코드 템플릿은 C++Builder에 포함되어 있습니다.

자주 사용하는 코딩 구조를 위한 템플릿을 직접 작성할 수도 있습니다. 예를 들어 코드에 **for** 루프를 사용하려는 경우 다음과 같은 템플릿을 삽입할 수 있습니다.

```
for (; ;)
{

}
```

코드 에디터에 코드 템플릿을 삽입하려면 *Ctrl-j*를 누르고 사용할 템플릿을 선택합니다. 직접 만든 템플릿도 이 컬렉션에 추가할 수 있습니다. 다음과 같은 방법으로 템플릿을 추가합니다.

1 Tools|Editor Options를 선택합니다.

2 Code Insight 탭을 클릭합니다.

3 Templates 섹션에서 Add를 클릭합니다.

4 Shortcut 이름 뒤에 템플릿 이름을 입력하고 새 템플릿에 대한 간단한 설명을 입력한 다음 OK를 클릭합니다.

5 템플릿 코드를 Code 텍스트 박스에 추가합니다.

6 OK를 클릭합니다.

콘솔 애플리케이션

콘솔 애플리케이션은 일반적으로 콘솔 윈도우에서 그래픽 인터페이스 없이 실행되는 32비트 프로그램입니다. 콘솔 애플리케이션은 일반적으로 사용자 입력이 많지 않으며, 제한된 함수 집합을 수행합니다.

다음과 같은 방법으로 새 콘솔 애플리케이션을 만듭니다.

1 File|New|Other를 선택하고 New Items 다이얼로그 박스에서 Console Wizard를 더블 클릭합니다.

2 Console Wizard 다이얼로그 박스에서 Console Application 옵션에 선택 표시를 하고, 프로젝트의 메인 모듈에 사용할 소스 타입(C 또는 C++)을 선택하거나 main 또는 winmain 함수를 포함하는 이전부터 존재하는 파일을 지정한 다음 OK 버튼을 클릭합니다.

C++Builder는 이러한 소스 타입을 위한 프로젝트 파일을 만들고 코드 에디터를 표시합니다.

콘솔 애플리케이션에서 VCL 및 CLX 사용

참고 새 콘솔 애플리케이션을 만드는 경우 IDE는 새 폼을 만들지 않습니다. 코드 에디터만 표시됩니다.

그러나 콘솔 애플리케이션에서 VCL 및 CLX 객체를 사용할 수 있습니다. 이렇게 하려면 Console Wizard에서 VCL이나 CLX를 사용하려 한다는 것을 나타내야 합니다(Use VCL 또는 Use CLX 옵션 선택). VCL이나 CLX를 사용하려 한다는 것을 마법사에서 표시하지 않으면 나중에 이 애플리케이션에서 VCL이나 CLX 클래스를 사용할 수 없습니다. 나중에 사용하려 하면 링커 오류가 발생합니다.

콘솔 애플리케이션은 모든 예외를 처리하여 애플리케이션이 실행되는 동안 윈도우에서 다이얼로그 박스를 표시하지 않도록 해야 합니다.

서비스 애플리케이션

서비스 애플리케이션은 클라이언트 애플리케이션의 요청을 받고, 이 요청을 처리하여 그 결과를 클라이언트 애플리케이션에게 반환합니다. 서비스 애플리케이션은 일반적으로 백그라운드에서 실행되며 사용자 입력이 많지 않습니다. Web, FTP, 전자 메일 서버 등이 서비스 애플리케이션의 예입니다.

다음과 같은 방법으로 Win32 서비스를 구현하는 애플리케이션을 만듭니다.

1 File|New|Other를 선택하고 New Items 다이얼로그 박스에서 Service Application을 더블 클릭합니다. 그러면 *TServiceApplication* 타입의 *Application* 이라는 전역 변수가 프로젝트에 추가됩니다.

2 서비스(*TService*)에 해당하는 Service 윈도우가 표시됩니다. Object Inspector에서 서비스의 속성과 이벤트를 설정하여 서비스를 구현합니다.

3 File|New|Other를 선택하고 New Items 다이얼로그 박스에서 Service를 더블 클릭하여 서비스 애플리케이션에 추가 서비스를 추가할 수 있습니다. 서비스 애플리케이션이 아닌 애플리케이션에 서비스를 추가하지 마십시오. *TService* 객체를 추가할 수는 있지만 애플리케이션은 필요한 이벤트를 생성하지 않거나 서비스를 대신해서 적절한 Windows 호출을 하지 않습니다.

4 서비스 애플리케이션이 생성되면 SCM(Service Control Manager)을 사용하여 해당 서비스를 설치할 수 있습니다. 그러면 다른 애플리케이션은 SCM에 요청을 보내서 이 서비스를 실행할 수 있습니다.

애플리케이션의 서비스를 설치하려면 /INSTALL 옵션을 사용하여 애플리케이션을 실행하십시오. 애플리케이션은 서비스를 설치하고 종료할 때 서비스가 성공적으로 설치되면 확인 메시지를 표시합니다. 서비스 애플리케이션을 실행할 때 /SILENT 옵션을 사용하면 확인 메시지를 표시하지 않을 수 있습니다.

서비스를 제거하려면 명령줄에서 /UNINSTALL 옵션을 사용하여 애플리케이션을 실행하십시오. 마찬가지로 /SILENT 옵션을 사용하여 제거할 때 확인 메시지를 표시하지 않을 수 있습니다.

예제 이 서비스에는 포트가 80으로 설정되는 *TServerSocket*이 있습니다. 이 디폴트 포트를 통해 웹 브라우저가 웹 서버에 요청을 보내고 웹 서버가 웹 브라우저에 응답합니다. 이 특별한 예제는 C:\Temp 디렉토리에 WebLog*xxx*.log(여기서 *xxx*는 ThreadID임)라는 텍스트 문서를 생성합니다. 지정된 포트에서 수신하는 서버는 하나만 있어야 하므로 웹 서버가 있는 경우에는 이 서버가 수신하고 있지 않아야 합니다. 서비스가 중지되어 있는지 확인합니다.

결과를 확인하려면, 로컬 컴퓨터에서 웹 브라우저를 열고 주소에 따옴표 없이 'localhost'를 입력합니다. 브라우저는 결국 종료되지만 C:\Temp 디렉토리에 Weblog*xxx*.log라는 파일이 생성됩니다.

1 예제를 만들려면 File|New|Other를 선택하고 New Items 다이얼로그 박스에서 Service Application을 선택합니다.

2 컴포넌트 팔레트의 Internet 페이지에서 ServerSocket 컴포넌트를 서비스 윈도우(Service1)에 추가합니다.

3 TMemoryStream 타입의 개인 데이터 멤버를 TService1 클래스에 추가합니다. 이제 유닛의 헤더는 다음과 같이 표시됩니다.

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <SvcMgr.hpp>
#include <ScktComp.hpp>
//---------------------------------------------------------------------------
class TService1 : public TService
{
__published:// IDE-managed Components
        TServerSocket *ServerSocket1;
private:// User declarations
        TMemoryStream *Stream; // add this line here
public:// User declarations
  __fastcall TService1(TComponent* Owner);
  PServiceController __fastcall GetServiceController(void);

  friend void __stdcall ServiceController(unsigned CtrlCode);
};
```

```
//---------------------------------------------------------------------------
extern PACKAGE TService1 *Service1;
//---------------------------------------------------------------------------
#endif
```

4 단계 1에서 추가한 ServerSocket1을 선택합니다. Object Inspector에서 *OnClientRead* 이벤트를 더블 클릭하고 다음과 같은 이벤트 핸들러를 추가합니다.

```
void __fastcall TService1::ServerSocket1ClientRead(TObject *Sender,
      TCustomWinSocket *Socket)
{
  char *Buffer = NULL;
  int len = Socket->ReceiveLength();
  while (len > 0)
  {
    try
    {
      Buffer = (char *)malloc(len);
      Socket->ReceiveBuf((void *)Buffer, len);
      Stream->Write(Buffer, len);
    }
    __finally
    {
      free(Buffer);
    }
    Stream->Seek(0, soFromBeginning);
    AnsiString LogFile = "C:\\Temp\\WebLog";
    LogFile = LogFile + IntToStr(ServiceThread->ThreadID) + ".log";
    Stream->SaveToFile(LogFile);
  }
}
```

5 끝으로 윈도우의 클라이언트 영역에서 ServiceSocket 이외의 위치를 클릭하여 Service1을 선택합니다. Object Inspector에서 *OnExecute* 이벤트를 더블 클릭하고 다음과 같은 이벤트 핸들러를 추가합니다.

```
void __fastcall TService1::Service1Execute(TService *Sender)
{
  Stream = new TMemoryStream();
  try
  {
    ServerSocket1->Port = 80; // WWW port
    ServerSocket1->Active = true;
    while (!Terminated)
       ServiceThread->ProcessRequests(true);
    ServerSocket1->Active = false;
   }
   __finally
   {
     delete Stream;
   }
  }
```

서비스 애플리케이션을 작성할 때 다음 사항에 대해 알고 있어야 합니다.

- 서비스 스레드
- 서비스 이름 속성
- 서비스 애플리케이션 디버깅

참고 서비스 애플리케이션은 Windows 전용입니다.

서비스 스레드

서비스마다 고유한 스레드(*TServiceThread*)를 가지므로 서비스 애플리케이션이 둘 이상의 서비스를 구현하는 경우 서비스 구현이 스레드에 안전한지 확인해야 합니다. *TServiceThread*는 *TService OnExecute* 이벤트 핸들러에서 서비스를 구현할 수 있도록 설계됩니다. 서비스 스레드는 고유한 *Execute* 메소드를 가지는데 이 메소드에는 새 요청을 처리하기 전에 서비스의 *OnStart* 및 *OnExecute* 핸들러를 호출하는 루프가 포함됩니다.

서비스 요청을 처리하는 데는 긴 시간이 걸릴 수 있고 서비스 애플리케이션은 둘 이상의 클라이언트로부터 동시에 요청을 받을 수 있으므로 각 요청에 대해 새 스레드(*TServiceThread*가 아닌 *TThread*에서 파생)를 만들어 해당 서비스 구현을 새 스레드의 *Execute* 메소드로 옮기는 방법이 더 효율적입니다. 이렇게 하면 서비스 스레드의 *Execute* 루프에서 서비스의 *OnExecute* 핸들러가 마칠 때까지 기다리지 않고 계속적으로 새 요청을 처리할 수 있습니다. 다음 예제에서 이런 방법을 보여 줍니다.

예제 이 서비스는 표준 스레드 내에서 500밀리초마다 신호음을 울립니다. 이 서비스는 일시 중지, 계속 또는 중지하도록 명령을 받을 때 스레드의 일시 중지, 계속 및 중지를 처리합니다.

1. File|New|Other를 선택하고 New Items 다이얼로그 박스에서 Service Application을 더블 클릭합니다. Service1 윈도우가 나타납니다.
2. 유닛 헤더 파일에서 TsparkyThread라는 *TThread*의 새 자손을 선언합니다. 이 스레드가 서비스의 작업을 수행하는 스레드입니다. 선언은 다음과 같이 표시되어야 합니다.

```
class TSparkyThread : public TThread
{
private:
protected:
        void __fastcall Execute();
public:
        __fastcall TSparkyThread(bool CreateSuspended);
};
```

3. 유닛의 .cpp 파일에서 TSparkyThread 인스턴스에 대한 전역 변수를 만듭니다.

```
TSparkyThread *SparkyThread;
```

4. TSparkyThread 생성자의 .cpp 파일에 다음 코드를 추가합니다.

```
__fastcall TSparkyThread::TSparkyThread(bool CreateSuspended)
        : TThread(CreateSuspended)
{
}
```

5. TSparkyThread Execute 메소드(스레드 함수)의 .cpp 파일에 다음 코드를 추가합니다.

```
void __fastcall TSparkyThread::Execute()
{
  while (!Terminated)
  {
    Beep();
    Sleep(500);
  }
}
```

6 서비스 윈도우(Service1)를 선택하고 Object Inspector에서 OnStart 이벤트를 더블 클릭합니다. 다음과 같은 OnStart 이벤트 핸들러를 추가합니다.

```
void __fastcall TService1::Service1Start(TService *Sender, bool &Started)
{
  SparkyThread = new TSparkyThread(false);
  Started = true;
}
```

7 Object Inspector에서 *OnContinue* 이벤트를 더블 클릭합니다. 다음과 같은 OnContinue 이벤트 핸들러를 추가합니다.

```
void __fastcall TService1::Service1Continue(TService *Sender, bool
&Continued)
{
  SparkyThread->Resume();
  Continued = true;
}
```

8 Object Inspector에서 OnPause 이벤트를 더블 클릭합니다. 다음과 같은 OnPause 이벤트 핸들러를 추가합니다.

```
void __fastcall TService1::Service1Pause(TService *Sender, bool &Paused)
{
  SparkyThread->Suspend();
  Paused = true;
}
```

9 끝으로 Object Inspector에서 OnStop 이벤트를 더블 클릭하고 다음과 같은 OnStop 이벤트 핸들러를 추가합니다.

```
void __fastcall TService1::Service1Stop(TService *Sender, bool &Stopped)
{
  SparkyThread->Terminate();
  Stopped = true;
}
```

서버 애플리케이션을 개발할 때 새 스레드를 생성할지 여부는 제공할 서비스 특성, 예상되는 연결 수 및 서비스를 실행하는 컴퓨터에 있어야 하는 프로세서 수 등에 따라 결정됩니다.

서비스 이름 속성

VCL은 Windows 플랫폼에서 서비스 애플리케이션을 만들기 위한 클래스를 제공합니다(다른 플랫폼 애플리케이션에서는 사용할 수 없음). 여기에는 *TService* 및 *TDependency* 등이 포함됩니다. 이 클래스를 사용할 때 다양한 이름 속성이 혼동을 줄 수 있습니다. 이 단원에서는 그 차이점에 대해 설명합니다.

서비스는 암호에 연결되는 사용자 이름(서비스 시작 이름이라고 함)과 관리자 및 에디터 윈도우에 표시하기 위한 표시 이름 및 실제 이름(서비스 이름) 등을 가집니다. 종속 관계는 서비스가 되거나 로드 정렬 그룹이 될 수 있습니다. 또한 종속 관계에는 이름과 표시 이름이 있습니다. 그리고 서비스 객체는 *TComponent*에서 파생하므로 *Name* 속성을 상속받습니다. 다음 단원에서는 이름 속성에 대해 요약합니다.

TDependency 속성

*TDependency DisplayName*은 서비스의 표시 이름이면서 실제 이름입니다. 이 속성은 거의 항상 *TDependency Name* 속성과 같습니다.

TService Name 속성

TService Name 속성은 *TComponent*에서 상속됩니다. 이 속성은 컴포넌트 이름이면서 서비스 이름이기도 합니다. 서비스인 종속 관계인 경우 이 속성은 *TDependency Name* 및 *DisplayName* 속성과 같습니다.

*TService*의 *DisplayName*은 Service Manager 윈도우에 표시되는 이름입니다. 이 이름은 실제 서비스 이름(*TService::Name*, *TDependency::DisplayName*, *TDependency::Name*)과 다른 경우가 많습니다. 종속 관계의 *DisplayName*과 서비스의 *DisplayName*은 일반적으로 다릅니다.

서비스 시작 이름은 서비스 표시 이름 및 실제 서비스 이름과 다릅니다. *ServiceStartName*은 Service Control Manager에서 선택된 Start 다이얼로그 박스에 입력된 사용자 이름입니다.

서비스 애플리케이션 디버깅

이미 실행 중인 서비스 애플리케이션 프로세스에 연결하여 서비스 애플리케이션을 디버깅할 수 있습니다. 즉 서비스를 먼저 시작한 다음 디버거에 연결합니다. 서비스 애플리케이션 프로세스에 연결하려면 Run|Attach To Process를 선택하고 표시되는 다이얼로그 박스에서 서비스 애플리케이션을 선택합니다.

권한이 충분하지 않아서 이 방법이 실패하는 경우도 있습니다. 이런 경우에는 Service Control Manager를 사용하여 서비스가 디버거와 함께 작동하게 할 수 있습니다.

1. 우선 다음 레지스트리 위치에 **Image File Execution Options**라는 키를 만듭니다.
   `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`
2. 서비스와 동일한 이름으로 하위 키를 만듭니다(예: MYSERV.EXE). 이 하위 키에 Debugger라는 REG_SZ 타입의 값을 추가합니다. BCB.exe에 대한 전체 경로를 문자열 값으로 사용합니다.
3. 제어판의 **서비스**에서 해당 애플릿에서 서비스를 선택하고 **시작**을 클릭한 다음 **서비스와 데스크톱 상호 작용 허용**을 선택합니다.

Windows NT 시스템인 경우 다른 방법을 사용하여 서비스 애플리케이션을 디버깅할 수 있습니다. 그러나 이 방법에서는 짧은 간격을 두어야 하므로 까다로울 수 있습니다.

1. 우선 디버거에서 애플리케이션을 실행합니다. 애플리케이션이 로드될 때까지 몇 초 간 기다립니다.
2. 명령줄에서 다음 명령을 실행하거나 제어판을 사용하여 서비스를 빨리 시작합니다.
   `start MyServ`

서비스가 시작되지 않으면 애플리케이션이 종료되므로 서비스를 빨리(애플리케이션 시작 후 15-30초 내에) 시작해야 합니다.

패키지 및 DLL 생성

동적 연결 라이브러리(DLL)는 애플리케이션에 기능을 제공하기 위해 실행 파일과 함께 작동하는 컴파일된 코드 모듈입니다. 크로스 플랫폼 프로그램에서 DLL을 생성할 수 있습니다. 그러나 Linux에서 DLL 및 패키지는 공유 객체로 다시 컴파일됩니다.

패키지는 C++Builder 애플리케이션이나 IDE 또는 둘다에서 사용되는 특수 DLL입니다. 패키지는 런타임 패키지와 디자인 타임 패키지 등 두 가지 종류가 있습니다. 런타임 패키지는 프로그램 실행 중에 이 프로그램에 기능을 제공합니다. 디자인 타임 패키지는 IDE의 기능을 확장합니다.

DLL과 라이브러리는 모든 예외를 처리하여 Windows 다이얼로그 박스를 통해 오류와 경고가 표시되지 않게 해야 합니다.

다음 컴파일러 지시어를 라이브러리 프로젝트 파일에 사용할 수 있습니다.

표 7.1 라이브러리에 대한 컴파일러 지시어

컴파일러 지시어	설명
{$LIBPREFIX 'string'}	지정한 접두사를 출력 파일 이름에 추가합니다. 예를 들면, 디자인 타임 패키지에 {$LIBPREFIX 'dcl'}을 지정할 수 있고, {$LIBPREFIX ' '}를 사용하여 접두사를 없앨 수도 있습니다.
{$LIBSUFFIX 'string'}	출력 파일 이름에서 확장자 앞에 지정된 접미사를 추가합니다. 예를 들면, something.cpp에 {$LIBSUFFIX '-2.1.3'}을 사용하면 something-2.1.3.bpl이 생성됩니다.
{$LIBVERSION 'string'}	출력 파일 이름에서 .bpl 확장자 뒤에 두 번째 확장자를 추가합니다. 예를 들면, .cpp에서 {$LIBVERSION '2.1.3'}를 사용하여 something.bpl.2.1.3을 생성합니다.

패키지에 대한 자세한 내용은 15장, "패키지와 컴포넌트 사용"을 참조하십시오.

패키지 및 DLL 사용 시기

C++Builder로 작성된 대부분의 애플리케이션에서 패키지는 DLL보다 유연성이 더 크고 만들기도 더 쉽습니다. 그러나 다음과 같은 경우에는 패키지보다 DLL이 프로젝트에 더 적합합니다.

- C++Builder로 작성되지 않은 애플리케이션에서 코드 모듈을 호출하는 경우
- 웹 서버의 기능을 확장하는 경우
- 서드파티 개발자가 사용할 코드 모듈을 생성하는 경우
- 프로젝트가 OLE 컨테이너인 경우

DLL 간이나 DLL에서 실행 파일로 런타임 타입 정보(RTTI)를 전달할 수 없습니다. DLL은 모두 자체의 기호 정보를 유지하기 때문입니다. DLL에서 **is** 또는 **as** 연산자를 사용하여 *TStrings* 객체를 전달해야 하는 경우에는 DLL이 아니라 패키지를 만들어야 합니다. 패키지는 기호 정보를 공유합니다.

C++Builder에서 DLL 사용

Windows DLL은 다른 C++ 애플리케이션에서와 마찬가지로 C++Builder 애플리케이션에서 사용할 수 있습니다.

C++Builder 애플리케이션이 로드될 때 DLL을 정적으로 로드하려면 연결 시 DLL의 임포트 라이브러리 파일을 C++Builder 애플리케이션에 연결합니다. 임포트 라이브러리를 C++Builder 애플리케이션에 추가하려면 Project|Add를 클릭하고 추가할 .LIB 파일을 선택하십시오.

그러면 해당 DLL의 엑스포트된 함수를 애플리케이션에서 사용할 수 있게 됩니다. 다음과 같이 **__declspec** (`dllimport`) 변경자를 사용하여 애플리케이션이 사용하는 DLL 함수의 프로토타입을 만듭니다.

```
__declspec(dllimport) return_type imported_function_name(parameters);
```

C++Builder 애플리케이션이 실행되는 동안 DLL을 동적으로 로드하려면 정적 로딩과 마찬가지로 임포트 라이브러리를 포함시키고 Project|Options|Advanced Linker 탭에서 Delay Load Linker 옵션을 설정합니다. Windows API 함수 *LoadLibrary()*를 사용하여 DLL을 로드한 다음 API 함수 *GetProcAddress()*를 사용하여 사용할 개별 함수에 대한 포인터를 가져올 수도 있습니다.

DLL 사용에 대한 추가 정보는 *Microsoft® Win32 SDK Reference*를 참조하십시오.

C++Builder에서 DLL 생성

C++Builder에서 DLL을 만들려면 다음과 같이 표준 C++에서와 같은 방법을 사용합니다.

1. File|NewOther를 선택하여 New Items 다이얼로그 박스를 표시합니다.
2. DLL Wizard 아이콘을 더블 클릭합니다.
3. 메인 모듈의 소스 타입(C 또는 C++)을 선택합니다.
4. DLL 엔트리 포인트가 DllMain, MSVC++ 스타일이 되게 하려면 VC++ 스타일 옵션을 선택합니다. 선택하지 않으면 DllEntryPoint가 엔트리 포인트로 사용됩니다.
5. Use VCL 또는 Use CLX를 클릭하여 VCL 또는 CLX 컴포넌트를 포함하는 DLL을 만듭니다. 이 옵션은 C++ 소스 모듈에서만 사용할 수 있습니다. 7-12페이지의 "VCL 및 CLX 컴포넌트를 포함하는 DLL 생성"을 참조하십시오.
6. DLL이 멀티 스레드가 되게 하려면 Multi-threaded 옵션을 선택합니다.
7. OK를 클릭합니다.

코드에서 엑스포트된 함수는 Borland C++ 또는 Microsoft Visual C++에서와 마찬가지로 **__declspec** (dllexport) 변경자로 식별해야 합니다. 예를 들어 다음 코드는 C++Builder 및 다른 Windows C++ 컴파일러에서 사용할 수 있습니다.

```
// MyDLL.cpp
double dblValue(double);
double halfValue(double);
extern "C" __declspec(dllexport) double changeValue(double, bool);

double dblValue(double value)
{
  return value * value;
};

double halfValue(double value)
{
  return value / 2.0;
}

double changeValue(double value, bool whichOp)
{
  return whichOp ? dblValue(value) : halfValue(value);
}
```

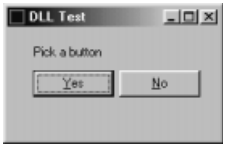
위의 코드에서 *changeValue* 함수는 엑스포트되므로 호출 애플리케이션에서 사용할 수 있습니다. *dblValue* 및 *halfValue* 함수는 내부 함수이므로 DLL 외부에서 호출할 수 없습니다.

DLL 생성에 대한 추가 정보는 *Microsoft® Win32 SDK Reference*를 참조하십시오.

VCL 및 CLX 컴포넌트를 포함하는 DLL 생성

DLL의 장점 중 하나는 하나의 개발 도구로 만든 DLL을 다른 개발 도구로 작성된 애플리케이션에서 사용할 수 있다는 점입니다. 호출 애플리케이션에서 사용될, 폼과 같은 VCL 또는 CLX 컴포넌트가 DLL에 포함되어 있는 경우 표준 호출 규칙을 사용하고 C++ 이름이 손상되는 것을 방지하며 작동하기 위해 호출 애플리케이션에서 VCL 및 CLX 라이브러리를 지원할 필요가 없는 엑스포트된 인터페이스 루틴을 제공해야 합니다. 엑스포트할 수 있는 VCL 또는 CLX 컴포넌트를 만들려면 런타임 패키지를 사용하십시오. 자세한 내용은 15장, "패키지와 컴포넌트 사용"을 참조하십시오.

예를 들어 다음과 같은 간단한 다이얼로그 박스를 표시하는 DLL을 만든다고 가정합니다.



이 다이얼로그 박스 DLL의 코드는 다음과 같습니다.

```
// DLLMAIN.H
//---------------------------------------------------------------------------
#ifndef dllMainH
#define dllMainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TYesNoDialog : public TForm
{
__published:    // IDE-managed Components
   TLabel *LabelText;
   TButton *YesButton;
   TButton *NoButton;
   void __fastcall YesButtonClick(TObject *Sender);
   void __fastcall NoButtonClick(TObject *Sender);
private:        // User declarations
   bool returnValue;
public:         // User declarations
   virtual __fastcall TYesNoDialog(TComponent *Owner);
   bool __fastcall GetReturnValue();
};

// exported interface function
extern "C" __declspec(dllexport) bool InvokeYesNoDialog();

//---------------------------------------------------------------------------
 extern TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
#endif


// DLLMAIN.CPP
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "dllMain.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
__fastcall TYesNoDialog::TYesNoDialog(TComponent *Owner)
 : TForm(Owner)
{
  returnValue = false;
}
```

```
//---------------------------------------------------------------------
void __fastcall TYesNoDialog::YesButtonClick(TObject *Sender)
{
   returnValue = true;
   Close();
}
//---------------------------------------------------------------------
void __fastcall TYesNoDialog::NoButtonClick(TObject *Sender)
{
   returnValue = false;
   Close();
}
//---------------------------------------------------------------------
bool __fastcall TYesNoDialog::GetReturnValue()
{
   return returnValue;
}
//---------------------------------------------------------------------
// exported standard C++ interface function that calls into VCL
bool InvokeYesNoDialog()
{
   bool returnValue;
     TYesNoDialog *YesNoDialog = new TYesNoDialog(NULL);
   YesNoDialog->ShowModal();
     returnValue = YesNoDialog->GetReturnValue();
     delete YesNoDialog;
   return returnValue;
}
//---------------------------------------------------------------------
```

이 예제의 코드는 다이얼로그 박스를 표시하고 "Yes" 버튼을 누르는 경우 private 데이터 멤버 *returnValue*에 **true** 값을 저장합니다. 그렇지 않으면 *returnValue*는 *false*입니다. public *GetReturnValue()* 함수는 *returnValue*의 현재 값을 검색합니다.

다이얼로그 박스를 호출하고 어떤 버튼을 눌렀는지 확인하기 위해 호출 애플리케이션은 엑스포트된 함수 *InvokeYesNoDialog()*를 호출합니다. 이 함수는 C 링크(C++ 이름이 손상되는 것을 방지) 및 표준 C 호출 규칙을 사용하여 DLLMAIN.H에서 엑스포트된 함수로 선언되고 DLLMAIN.CPP에 정의됩니다.

표준 C 함수를 DLL에 대한 인터페이스로 사용하므로 C++Builder로 만들었는지 여부에 관계없이 모든 호출 애플리케이션이 DLL을 사용할 수 있습니다. 다이얼로그 박스를 지원하기 위해 필요한 VCL 및 CLX 기능은 DLL 자체에 연결되므로 호출 애플리케이션은 이 사항에 대해 알 필요가 없습니다.

VCL 또는 CLX를 사용하는 DLL을 만드는 경우 필요한 VCL 또는 CLX 컴포넌트를 DLL에 연결하는 데 일정한 양의 오버헤드가 발생합니다. 이 오버헤드가 애플리케이션의 전체 크기에 미치는 영향을 최소화하려면 VCL 및 CLX 지원 컴포넌트의 복사본 하나만을 필요로 하는 DLL 한 개에 여러 컴포넌트를 연결하십시오.

DLL 연결

Project Options 다이얼로그 박스의 Linker 페이지에서는 DLL에 대한 링커 옵션을 설정할 수 있습니다. 이 페이지의 디폴트 체크 박스는 DLL에 대한 임포트 라이브러리도 생성합니다. 명령줄에서 컴파일하는 경우에는 `-Tpd` 스위치를 사용하여 ILINK32.EXE 링커를 호출합니다. 예를 들면, 다음과 같습니다.

```
ilink32 /c /aa /Tpd c0d32.obj mydll.obj, mydll.dll, mydll.map,
import32.lib cw32mt.lib
```

임포트 라이브러리가 필요하면 -Gi 스위치를 사용하여 임포트 라이브러리를 생성합니다.

명령줄 유틸리티 IMPLIB.EXE를 사용하여 임포트 라이브러리를 만들 수도 있습니다. 예를 들면, 다음과 같습니다.

```
implib mydll.lib mydll.dll
```

DLL을 연결하고 런타임 라이브러리에 정적 또는 동적으로 연결되어 있는 다른 모듈과 함께 DLL을 사용하는 여러 옵션에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

데이터베이스 애플리케이션 생성

C++Builder의 장점 중 하나는 고급 데이터베이스 애플리케이션을 만들 수 있다는 점입니다. C++Builder는 SQL 서버와 데이터베이스(예: Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, DB2 등)에 연결할 수 있는 도구를 지원하며 애플리케이션 간에 투명한 데이터 공유를 제공합니다.

C++Builder에는 데이터베이스를 액세스하고 데이터베이스가 포함하는 정보를 나타내기 위한 많은 컴포넌트가 들어 있습니다. 컴포넌트 팔레트에서 데이터베이스 컴포넌트는 데이터 액세스 메커니즘 및 기능에 따라 그룹화되어 있습니다.

표 7.2 컴포넌트 팔레트의 데이터베이스 페이지

팔레트 페이지	내용
BDE	데이터베이스와 상호 작용하기 위한 대형 API인 BDE(Borland Database Engine)를 사용하는 컴포넌트. BDE는 가장 넓은 범위의 함수를 지원하며 Database Desktop, Database Explorer, SQL Monitor, BDE Administrator 등을 포함하여 가장 강력한 지원 유틸리티를 제공합니다. 자세한 내용은 24장, "Borland Database Engine 사용"을 참조하십시오.
ADO	Microsoft에서 개발한 ADO(ActiveX Data Objects)를 사용하여 데이터베이스 정보를 액세스하는 컴포넌트. 다양한 데이터베이스 서버를 연결하는 데 사용할 수 있는 많은 ADO 드라이버가 있습니다. ADO 기반 컴포넌트를 사용하면 애플리케이션을 ADO 기반 환경에 통합할 수 있습니다. 자세한 내용은 25장, "ADO 컴포넌트 사용"을 참조하십시오.
dbExpress	dbExpress를 사용하여 데이터베이스 정보를 액세스하는 크로스 플랫폼 컴포넌트. dbExpress 드라이버는 데이터베이스에 빠르게 액세스할 수 있게 하지만 업데이트를 수행하려면 *TClientDataSet* 및 *TDataSetProvider*와 함께 사용해야 합니다. 자세한 내용은 26장, "단방향 데이터셋 사용"을 참조하십시오.
InterBase	각각의 엔진 레이어를 통하지 않고 InterBase 데이터베이스를 직접 액세스하는 컴포넌트. InterBase 컴포넌트 사용에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

표 7.2 컴포넌트 팔레트의 데이터베이스 페이지

팔레트 페이지	내용
Data Access	*TClientDataSet* 및 *TDataSetProvider*와 같은 데이터 액세스 메커니즘과 함께 사용할 수 있는 컴포넌트. 클라이언트 데이터셋에 대한 자세한 내용은 27장, "클라이언트 데이터셋 사용"을 참조하고 프로바이더에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오.
Data Controls	데이터 소스의 정보를 액세스할 수 있는 데이터 인식 컨트롤. 자세한 내용은 19장, "데이터 컨트롤 사용"을 참조하십시오.

데이터베이스 애플리케이션을 디자인할 때 사용할 데이터 액세스 메커니즘을 결정해야 합니다. 각 데이터 액세스 메커니즘에 따라 기능 지원 범위, 배포의 편리함 및 다른 데이터베이스 서버를 지원하기 위한 드라이버 사용 가능성 등이 다릅니다.

C++Builder를 사용하여 데이터베이스 클라이언트 애플리케이션과 애플리케이션 서버를 만드는 방법에 대한 자세한 내용은 이 설명서에서 II부, "데이터베이스 애플리케이션 개발"을 참조하십시오. 배포에 관한 내용은 17-6페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

참고 모든 C++Builder 에디션에 데이터베이스 지원이 포함되어 있지는 않습니다.

분산 데이터베이스 애플리케이션

C++Builder는 통합된 컴포넌트 집합을 사용하여 분산 데이터베이스 애플리케이션을 만들 수 있도록 지원합니다. 분산 데이터베이스 애플리케이션은 DCOM, TCP/IP 및 SOAP 등을 포함하는 다양한 통신 프로토콜에 구축할 수 있습니다.

분산 데이터베이스 애플리케이션 생성에 대한 자세한 내용은 29장, "멀티 티어 애플리케이션 생성"을 참조하십시오.

데이터베이스 애플리케이션을 분산하기 위해 애플리케이션 파일과 함께 BDE(Borland Database Engine)도 배포해야 하는 경우가 종종 있습니다. BDE 배포에 대한 자세한 내용은 17-6페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

웹 서버 애플리케이션 생성

웹 서버 애플리케이션은 HTML 웹 페이지나 XML 문서와 같은 웹 컨텐트를 인터넷을 통해 배달하는 서버에서 실행되는 애플리케이션입니다. 웹 서버 애플리케이션의 예로는 웹 사이트에 대한 액세스 제어, 주문서 생성, 정보 요청에 대한 응답 등을 수행하는 애플리케이션을 들 수 있습니다.

다음과 같은 C++Builder 기술을 사용하여 다양한 타입의 웹 서버 애플리케이션을 만들 수 있습니다.

- Web Broker
- WebSnap
- InternetExpress
- 웹 서비스

Web Broker 사용

Web Broker(NetCLX 아키텍처라고도 함)를 사용하여 CGI 애플리케이션이나 동적 연결 라이브러리(DLL) 등과 같은 웹 서버 애플리케이션을 만들 수 있습니다. 이러한 웹 서버 애플리케이션은 넌비주얼(nonvisual) 컴포넌트를 포함할 수 있습니다. 컴포넌트 팔레트의 Internet 페이지에 있는 컴포넌트를 사용하여 이벤트 핸들러를 만들고 HTML 또는 XML 문서를 프로그램에서 작성할 수 있으며 이것을 클라이언트에 전송할 수 있습니다.

Web Broker 아키텍처를 사용하여 새 웹 서버 애플리케이션을 만들려면 File|New|Other를 선택하고 New Items 다이얼로그 박스에서 Web Server Application을 더블 클릭하십시오. 그런 다음 웹 서버 애플리케이션 타입을 다음 중에서 선택합니다.

표 7.3 웹 서버 애플리케이션

웹 서버 애플리케이션 타입	설명
ISAPI 및 NSAPI 동적 연결 라이브러리	ISAPI 및 NSAPI 웹 서버 애플리케이션은 웹 서버에서 로드하는 DLL입니다. 클라이언트 요청 정보는 구조체로 DLL에 전달되고 TISAPIApplication이 분석합니다. 각 요청 메시지는 각각의 실행 스레드에서 처리됩니다. 이 애플리케이션 타입을 선택하면 프로젝트 파일의 라이브러리 헤더와 필요한 항목이 프로젝트 파일의 uses 리스트 및 exports 절에 추가됩니다.
CGI 독립 실행형 실행 파일	CGI 웹 서버 애플리케이션은 표준 입력에서 클라이언트의 요청을 받아 처리한 다음 클라이언트에 보낼 표준 출력으로 결과를 서버에 돌려 보내는 콘솔 애플리케이션입니다.
Win-CGI 독립 실행형 실행 파일	Win-CGI 웹 서버 애플리케이션은 서버에서 작성하는 구성 설정(INI) 파일로부터 클라이언트의 요청을 받고 그 결과를 서버가 클라이언트에 돌려 보내는 파일에 기록하는 Windows 애플리케이션입니다. INI 파일은 TCGIApplication이 분석합니다. 요청 메시지는 각각의 애플리케이션 인스턴스에 의해 처리됩니다.
Apache 공유 모듈 (DLL)	이 애플리케이션 타입을 선택하면 프로젝트를 DLL로 설정합니다. Apache 웹 서버 애플리케이션은 웹 서버에서 로드하는 DLL입니다. 웹 서버가 정보를 DLL에 전달하고 처리한 다음 클라이언트에 반환합니다.
Web App Debugger 독립 실행형 실행 파일	이 애플리케이션 타입을 선택하면 웹 서버 애플리케이션을 개발하고 테스트하기 위한 환경을 설정합니다. Web App Debugger 애플리케이션은 웹 서버에서 로드하는 실행 파일입니다. 이 애플리케이션 타입은 배포용이 아닙니다.

CGI 및 Win-CGI 애플리케이션은 서버의 시스템 리소스를 더 많이 사용하므로 복잡한 애플리케이션은 ISAPI, NSAPI 또는 Apache DLL 애플리케이션으로 작성하는 것이 더 좋습니다. 크로스 플랫폼 애플리케이션을 작성하는 경우 웹 서버 개발을 위해 CGI 독립 실행형이나 Apache Shared Module(DLL)을 선택해야 합니다. WebSnap 및 Web Service 애플리케이션을 만드는 경우에도 이와 동일한 옵션이 제공됩니다.

웹 서버 애플리케이션 생성에 대한 자세한 내용은 32장, "인터넷 서버 애플리케이션 생성"을 참조하십시오.

WebSnap 애플리케이션 생성

WebSnap은 웹 브라우저와 상호 작용하는 고급 웹 서버를 생성하는 데 사용하는 일련의 컴포넌트와 마법사를 제공합니다. WebSnap 컴포넌트는 웹 페이지의 HTML 또는 다른 MIME 컨텐트를 생성합니다. WebSnap은 서버사이드 개발에 사용됩니다. 현재 WebSnap은 크로스 플랫폼 애플리케이션에서 사용할 수 없습니다.

새 WebSnap 애플리케이션을 만들려면 File|New|Other를 선택하고 New Items 다이얼로그 박스에서 WebSnap을 선택합니다. WebSnap Application을 선택합니다. 그런 다음 웹 서버 애플리케이션 타입(ISAPI/NSAPI, CGI, Win-CGI, Apache)을 선택합니다. 자세한 내용은 표 7.3, "웹 서버 애플리케이션"을 참조하십시오.

WebSnap에 대한 자세한 내용은 34장, "WebSnap을 사용하여 웹 서버 애플리케이션 생성"을 참조하십시오.

InternetExpress 사용

InternetExpress는 기본 웹 서버 애플리케이션 아키텍처를 확장하여 애플리케이션 서버의 클라이언트로 작동하는 컴포넌트 집합입니다. InternetExpress는 클라이언트에서 실행되는 동안 브라우저 기반 클라이언트가 프로바이더에서 데이터를 가져오고 프로바이더에 대한 업데이트를 확인할 수 있는 애플리케이션에 사용합니다.

InternetExpress 애플리케이션은 HTML, XML 및 javascript를 함께 포함하는 HTML 페이지를 생성합니다. HTML은 엔드 유저의 브라우저에 표시되는 페이지의 레이아웃과 모양을 결정합니다. XML은 데이터베이스 정보를 나타내는 데이터 패킷과 델타 패킷을 인코딩합니다. javascript는 HTML 컨트롤이 클라이언트 컴퓨터의 XML 데이터 패킷에 있는 데이터를 해석하고 처리할 수 있게 합니다.

InternetExpress에 대한 자세한 내용은 29-31페이지의 "InternetExpress를 이용한 웹 애플리케이션 개발"을 참조하십시오.

웹 서비스 애플리케이션 생성

웹 서비스는 네트워크(예: World Wide Web)를 통해 게시하고 호출할 수 있는 독립적인 모듈식 애플리케이션입니다. 웹 서비스는 제공되는 서비스를 설명하는 잘 정의된 인터페이스를 제공합니다. 웹 서비스를 사용하면 XML, XML 스키마, SOAP(Simple Object Access Protocol) 및 WSDL(Web Service Definition Language) 등과 같은 새로운 표준을 사용하여 인터넷 상에서 프로그램 가능한 서비스를 만들거나 사용할 수 있습니다.

웹 서비스는 분산 환경에서 정보를 교환하기 위한 표준 lightweight 프로토콜인 SOAP를 사용합니다. 웹 서비스는 HTTP를 통신 프로토콜로 사용하고, XML을 사용하여 원격 프로시저 호출을 인코드합니다.

C++Builder를 사용하여 웹 서비스를 구현하는 서버와 이 서비스에 호출하는 클라이언트를 생성할 수 있습니다. SOAP 메시지에 응답하는 웹 서비스를 구현하는 임의의 서버와 임의의 클라이언트에서 사용할 수 있도록 웹 서비스를 게시하는 C++Builder 서버에 대한 클라이언트를 작성할 수 있습니다.

웹 서비스에 대한 자세한 내용은 36장, "웹 서비스 사용"을 참조하십시오.

COM을 사용한 애플리케이션 생성

COM(Component Object Model)은 인터페이스라는 미리 정의된 루틴을 사용하여 객체의 상호 운용성을 제공하기 위해 설계된 Windows 기반 분산 객체 아키텍처입니다. COM 애플리케이션은 다른 프로세스 또는 다른 컴퓨터(DCOM을 사용하는 경우)에서 구현되는 객체를 사용합니다. 또한 COM+, ActiveX 및 Active Server Pages 등도 사용할 수 있습니다.

COM은 Windows 플랫폼에서 실행되는 애플리케이션과 소프트웨어 컴포넌트 간의 상호 작용을 가능하게 하는 랭귀지 독립 소프트웨어 컴포넌트입니다. COM의 주요한 특징은 명확하게 정의된 인터페이스를 통해 컴포넌트 간, 애플리케이션 간 및 클라이언트와 서버 간의 통신을 가능하게 한다는 것입니다. 인터페이스를 통해 클라이언트는 COM 컴포넌트에게 런타임 시 지원하는 기능을 질문할 수 있습니다. 컴포넌트에 추가 기능을 제공하려면 이 기능를 위한 추가 인터페이스를 추가하면 됩니다.

COM 및 DCOM 사용

C++Builder에는 COM, OLE 또는 ActiveX 애플리케이션을 쉽게 만들 수 있게 도와주는 클래스와 마법사가 있습니다. COM 객체, Automation 서버(Active Server Objects 포함), ActiveX 컨트롤 또는 ActiveForms 등을 구현하는 COM 클라이언트나 서버를 만들 수 있습니다. COM은 Automation, ActiveX 컨트롤, Active Documents 및 Active Directories 등과 같은 다른 기술의 기초로도 사용됩니다.

COM 기반 애플리케이션을 만들기 위해 C++Builder를 사용하면 애플리케이션 내부에서 인터페이스를 사용하여 소프트웨어 디자인을 개선하는 것에서부터 시스템의 다른 COM 기반 API 객체와 상호 작용할 수 있는 객체(예: Win9x 셸 확장 및 DirectX 멀티미디어 지원 등)를 만들 수 있는 것까지 다양한 가능성을 제공합니다. 애플리케이션은 해당 애플리케이션과 같은 컴퓨터에 있는 COM 컴포넌트 인터페이스에 액세스할 수도 있고 DCOM이라는 메커니즘을 사용하여 네트워크상의 다른 컴퓨터에 있는 COM 컴포넌트 인터페이스에 액세스할 수도 있습니다.

COM 및 Active X 컨트롤에 대한 자세한 내용은 38장, "COM 기술 개요", 43장, "ActiveX 컨트롤 생성" 및 29-30페이지의 "클라이언트 애플리케이션을 ActiveX 컨트롤로 분산"을 참조하십시오.

DCOM에 대한 자세한 내용은 29-9페이지의 "DCOM 연결 사용"을 참조하십시오.

MTS 및 COM+ 사용

대규모 분산 환경에서 객체를 관리하기 위한 특수 서비스를 사용하여 COM 애플리케이션의 기능을 강화할 수 있습니다. 이 서비스는 Microsoft Transaction Server(MTS)(Windows 2000 이전의 Windows 버전) 또는 COM+(Windows 2000 이후 버전)에서 제공하는 트랜잭션 서비스, 보안 및 리소스 관리 등을 포함합니다.

MTS 및 COM+에 대한 자세한 내용은 44장, "MTS 객체 또는 COM+ 객체 생성" 및 29-6페이지의 "트랜잭션 데이터 모듈 사용"을 참조하십시오.

데이터 모듈 사용

데이터 모듈은 넌비주얼(nonvisual) 컴포넌트를 포함하는 특수한 폼과 비슷합니다. 데이터 모듈의 모든 컴포넌트는 비주얼(visual) 컨트롤과 함께 일반적인 폼에 둘 수있습니다. 그러나 데이터베이스와 시스템 객체 그룹을 재사용하거나 데이터베이스 연결과 비즈니스 룰을 처리하는 애플리케이션의 일부를 분리하려는 경우 데이터 모듈은 간편한 구성 도구를 제공합니다.

C++Builder 에디션에 따라 표준, 원격, 웹 모듈, 애플릿 모듈, 서비스 등 여러 가지 타입의 데이터 모듈이 있습니다. 각 타입의 데이터 모듈은 특수한 용도로 사용됩니다.

- 표준 데이터 모듈은 단일 티어 및 2 티어 데이터베이스 애플리케이션에 특히 유용하지만 모든 애플리케이션에서 넌비주얼 컴포넌트를 구성하는 데 사용할 수 있습니다. 자세한 내용은 7-20페이지의 "표준 데이터 모듈 생성 및 편집"을 참조하십시오.
- 원격 데이터 모듈은 멀티 티어 데이터베이스에서 애플리케이션 서버의 기초를 형성합니다. 이 모듈은 모든 에디션에서 사용할 수 있는 것은 아닙니다. 원격 데이터 모듈은 애플리케이션 서버에 넌비주얼 컴포넌트를 유지하는 것 외에 클라이언트가 애플리케이션 서버와 통신하기 위해 사용하는 인터페이스를 노출합니다. 원격 데이터 모듈 사용에 대한 자세한 내용은 7-23페이지의 "애플리케이션 서버 프로젝트에 원격 데이터 모듈 추가"를 참조하십시오.
- 웹 모듈은 웹 서버 애플리케이션의 기초를 형성합니다. 웹 모듈은 HTTP 응답 메시지의 내용을 만드는 컴포넌트를 유지하는 것 외에 클라이언트 애플리케이션에서 HTTP 메시지의 디스패칭을 처리합니다. 웹 모듈 사용에 대한 자세한 내용은 32장, "인터넷 서버 애플리케이션 생성"을 참조하십시오.
- 애플릿 모듈은 제어판 애플릿의 기초를 형성합니다. 애플릿 모듈은 제어판 애플릿을 구현하는 넌비주얼 컨트롤을 유지하는 것 외에 애플릿 아이콘이 제어판에 표시되는 모양을 결정하는 속성을 정의하고 사용자가 애플릿을 실행할 때 호출되는 이벤트를 포함시킵니다. 이러한 프로시저에 대한 자세한 내용은 온라인 도움말을 참조하십시오.
- 서비스는 NT 서비스 애플리케이션에 있는 개별 서비스를 캡슐화합니다. 서비스는 이를 구현하는 데 사용되는 넌비주얼 컨트롤을 유지하는 것 외에 서비스를 시작하거나 중지할 때 호출되는 이벤트를 포함시킵니다. 서비스에 대한 자세한 내용은 7-4페이지의 "서비스 애플리케이션"을 참조하십시오.

표준 데이터 모듈 생성 및 편집

프로젝트의 표준 데이터 모듈을 만들려면 File|New|Data Module을 선택합니다. C++Builder는 데스크탑에 데이터 모듈 컨테이너를 열고, 코드 에디터에 새 모듈의 유닛 파일을 표시하고, 이 모듈을 현재 프로젝트에 추가합니다.

디자인 타임에 데이터 모듈은 배경이 흰색이고 맞춤 그리드가 없는 표준 C++Builder 폼과 비슷합니다. 폼에서와 마찬가지로 컴포넌트 팔레트에서 넌비주얼 컴포넌트를 모듈에 놓고 Object Inspector에서 그 속성을 편집할 수 있습니다. 추가하는 컴포넌트에 맞게 데이터 모듈의 크기를 다시 조정할 수 있습니다.

또한 모듈을 마우스 오른쪽 버튼으로 클릭하여 모듈의 컨텍스트 메뉴를 표시할 수 있습니다. 다음 표는 데이터 모듈의 컨텍스트 메뉴 옵션을 요약한 것입니다.

표 7.4 데이터 모듈의 컨텍스트 메뉴 옵션

메뉴 항목	용도
Edit	데이터 모듈의 컴포넌트를 잘라내기, 복사, 붙여넣기, 삭제 및 선택할 수 있는 컨텍스트 메뉴를 표시합니다.
Position	보이지 않는 모듈 그리드(*Align To Grid)* 또는 Alignment 다이얼로그 박스에서 지정한 기준에 따라(*Align*) 넌비주얼(nonvisual) 컴포넌트를 정렬합니다.
Tab Order	탭 키를 누를 때 포커스가 컴포넌트에서 컴포넌트로 이동하는 순서를 변경할 수 있습니다.
Creation Order	시작 시 데이터 액세스 컴포넌트가 만들어지는 순서를 변경할 수 있습니다.
Revert to Inherited	Object Repository의 다른 모듈에서 상속된 모듈에 대한 변경 내용을 버리고 원래 상속된 모듈로 되돌립니다.
Add to Repository	데이터 모듈에 대한 연결을 Object Repository에 저장합니다.
View as Text	데이터 모듈 속성의 텍스트 표현을 표시합니다.
Text DFM	이 특정 폼 파일이 저장되는 형식(바이너리 또는 텍스트) 간을 토글합니다.

데이터 모듈에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

데이터 모듈 및 해당 유닛 파일 이름 지정

데이터 모듈의 제목 표시줄은 모듈 이름을 표시합니다. 데이터 모듈의 디폴트 이름은 "DataModule*N*"이며 여기서 *N*은 프로젝트에서 사용되지 않는 유닛 번호 중 가장 낮은 번호를 나타내는 숫자입니다. 예를 들어 새 프로젝트를 시작하고 다른 애플리케이션을 생성하기 전에 프로젝트에 모듈을 추가하면 이 모듈의 이름은 디폴트로 "DataModule2"가 됩니다. *DataModule2*의 해당 유닛 파일은 디폴트로 "Unit2"가 됩니다.

디자인 타임에 데이터 모듈과 해당 유닛 파일의 이름을 다시 지정하여 의미를 좀더 명확하게 해야 합니다. 특히 Object Repository에 추가할 데이터 모듈인 경우 모듈을 사용하는 애플리케이션이나 Repository에 있는 다른 데이터 모듈과 이름이 충돌하지 않도록 이름을 다시 지정해야 합니다.

다음과 같은 방법으로 데이터 모듈의 이름을 다시 지정합니다.

1 모듈을 선택합니다.

2 Object Inspector에서 모듈의 *Name* 속성을 편집합니다.

Object Inspector의 *Name* 속성에 포커스가 없으면 모듈의 새 이름이 제목 표시줄에 표시됩니다.

디자인 타임에 데이터 모듈 이름을 변경하면 코드의 인터페이스 섹션에 있는 해당 변수 이름이 변경됩니다. 프로시저 선언에서 타입 이름의 사용도 변경됩니다. 작성한 코드에서 데이터 모듈에 대한 참조는 수동으로 변경해야 합니다.

다음과 같은 방법으로 데이터 모듈의 유닛 파일 이름을 다시 지정합니다.

1 유닛 파일을 선택합니다.

컴포넌트 배치 및 이름 지정

비주얼(visual) 컴포넌트를 폼에 놓는 것처럼 넌비주얼(nonvisual) 컴포넌트를 데이터 모듈에 놓습니다. 컴포넌트 팔레트의 해당 페이지에서 원하는 컴포넌트를 클릭한 다음 컴포넌트를 놓을 데이터 모듈을 클릭합니다. 그리드와 같은 비주얼 컨트롤은 데이터 모듈에 놓을 수 없습니다. 이렇게 하려고 하면 오류 메시지가 반환됩니다.

사용하기 편리하도록 데이터 모듈에서 컴포넌트는 이름으로 표시됩니다. 컴포넌트를 처음으로 놓는 경우 C++Builder는 컴포넌트의 종류를 식별하는 일반적인 이름을 할당하고 그 뒤에 *1*을 붙입니다. 예를 들어 *TDataSource* 컴포넌트는 *DataSource1*이라는 이름을 받아들입니다. 따라서 작업할 속성과 메소드가 있는 특정 컴포넌트를 선택하기가 쉬워집니다.

또한 컴포넌트 타입과 용도를 반영하도록 컴포넌트에 다른 이름을 지정할 수도 있습니다.

다음과 같은 방법으로 데이터 모듈에서 컴포넌트 이름을 변경합니다.

1 컴포넌트를 선택합니다.

2 Object Inspector에서 컴포넌트의 *Name* 속성을 편집합니다.

Object Inspector의 *Name* 속성에 포커스가 없으면 컴포넌트의 새 이름이 해당 아이콘 아래에 표시됩니다.

예를 들어, 데이터베이스 애플리케이션이 CUSTOMER 테이블을 사용한다고 가정합니다. 테이블을 액세스하려면 최소한 데이터 소스 컴포넌트(*TDataSource)* 및 테이블 컴포넌트 (*TClientDataSet)* 등 두 개의 데이터 액세스 컴포넌트가 있어야 합니다. 이 두 컴포넌트를 데이터 모듈에 놓으면 C++Builder는 이 컴포넌트에 *DataSource1* 및 *ClientDataSet1*이라는 이름을 할당합니다. 컴포넌트 타입과 컴포넌트에서 액세스하는 데이터베이스(CUSTOMER)를 반영하려면 이 이름을 *CustomerSource* 및 *CustomerTable*로 변경해야 합니다.

데이터 모듈에서 컴포넌트 속성 및 이벤트 사용

데이터 모듈에 컴포넌트를 놓으면 전체 애플리케이션을 위해 컴포넌트 동작을 한 곳에 모읍니다. 예를 들어 데이터셋 컴포넌트의 속성(예: *TClientDataSet*)을 사용하여 해당 데이터셋을 사용하는 데이터 소스 컴포넌트에서 사용할 수 있는 데이터를 제어할 수 있습니다. 데이터셋에 대해 *ReadOnly* 속성을 *true*로 설정하면 사용자가 폼의 데이터 인식 비주얼(visual) 컨트롤에 표시되는 데이터를 편집할 수 없습니다. *ClientDataSet1*을 더블 클릭하여 데이터셋용 Fields Editor를 호출하여 테이블이나 쿼리 내에서 데이터 소스와 폼의 데이터 인식 컨트롤에서 사용할 수 있는 필드를 제한할 수 있습니다. 데이터 모듈에서 컴포넌트에 대해 설정한 속성은 모듈을 사용하는 애플리케이션의 모든 폼에 일관성 있게 적용됩니다.

컴포넌트에 대한 속성 외에도 이벤트 핸들러를 작성할 수 있습니다. 예를 들어 *TDataSource* 컴포넌트에는 *OnDataChange*, *OnStateChange*, *OnUpdateData* 등 세 가지 이벤트가 있을 수 있습니다. *TClientDataSet* 컴포넌트는 잠재적인 이벤트가 20개가 넘습니다. 이 이벤트를 사용하여 전체 애플리케이션에서 데이터 처리를 제어하는 일관성 있는 비즈니스 룰 집합을 만들 수 있습니다.

데이터 모듈에서 비즈니스 룰 생성

데이터 모듈에 있는 컴포넌트의 이벤트 핸들러를 작성하는 것 외에 데이터 모듈의 유닛 파일에서는 직접 메소드를 코딩할 수도 있습니다. 이 메소드는 데이터 모듈을 비즈니스 룰로 사용하는 폼에 적용할 수 있습니다. 예를 들어 월말, 분기말 또는 연말 부기를 수행하는 프로시저를 작성할 수 있습니다. 이 프로시저를 데이터 모듈에 있는 컴포넌트의 이벤트 핸들러에서 호출할 수 있습니다.

폼에서 데이터 모듈 액세스

폼에 있는 비주얼(visual) 컨트롤을 데이터 모듈에 연결하려면 먼저 데이터 모듈의 헤더 파일을 폼의 cpp 파일에 추가해야 합니다. 다음과 같은 여러 가지 방법으로 이렇게 할 수 있습니다.

- 코드 에디터에서 폼의 유닛 파일을 열고 *#include* 지시어를 사용하여 데이터 모듈의 헤더 파일을 포함시킵니다.
- 폼의 유닛 파일을 클릭하고 File|Include Unit Hdr을 선택한 다음 모듈 이름을 입력하거나 Use Unit 다이얼로그 박스의 리스트 박스에서 선택합니다.
- 데이터베이스 컴포넌트인 경우, 데이터 모듈에서 데이터셋이나 쿼리 컴포넌트를 클릭하여 Fields Editor를 열고 기존 필드를 에디터에서 폼으로 끌어 놓습니다. C++Builder는 폼에 모듈을 추가할지 묻는 메시지를 표시한 다음 필드의 컨트롤(예: 에디트 박스)을 만듭니다.

예를 들어 *TClientDataSet* 컴포넌트를 데이터 모듈에 추가한 경우 이 컴포넌트를 더블 클릭하여 Fields Editor를 엽니다. 필드를 선택하여 폼으로 끌어 놓습니다. 에디트 박스 컴포넌트가 나타납니다.

데이터 소스가 아직 정의되지 않았으므로 C++Builder는 새 데이터 소스 컴포넌트(*DataSource1*)를 폼에 추가하고 에디트 박스의 *DataSource* 속성을 *DataSource1*로 설정합니다. 데이터 소스의 *DataSet* 속성은 자동으로 데이터 모듈의 데이터셋 컴포넌트(*ClientDataSet1*)로 설정됩니다.

필드를 폼에 끌어 놓기 *전*에 데이터 소스를 정의하려면 데이터 모듈에 *TDataSource* 컴포넌트를 추가하면 됩니다. 그러고 데이터 소스의 *DataSet* 속성을 *ClientDataSet1*로 설정합니다. 필드를 폼에 끌어 놓으면 에디트 박스는 *TDataSource* 속성이 이미 *DataSource1*로 설정된 상태로 표시됩니다. 이 방법을 사용하면 데이터 액세스 모델을 좀더 확실하게 유지할 수 있습니다.

애플리케이션 서버 프로젝트에 원격 데이터 모듈 추가

일부 에디션의 C++Builder에서는 애플리케이션 서버 프로젝트에 *원격 데이터 모듈*을 추가할 수 있습니다. 원격 데이터 모듈에는 멀티 티어 애플리케이션의 클라이언트가 네트워크를 통해 액세스할 수 있는 인터페이스가 있습니다.

다음과 같은 방법으로 원격 데이터 모듈을 프로젝트에 추가합니다.

1. File|New|Other를 선택합니다.
2. New Items 다이얼로그 박스에서 Multitier 페이지를 선택합니다.
3. Remote Data Module 아이콘을 더블 클릭하여 Remote Data Module 마법사를 엽니다.

원격 데이터 모듈을 프로젝트에 추가하면 이 모듈을 표준 데이터 모듈처럼 사용합니다.

멀티 티어 데이터베이스 애플리케이션에 대한 자세한 내용은 29장, "멀티 티어 애플리케이션 생성"을 참조하십시오.

Object Repository 사용

Object Repository(Tools|Repository)를 사용하면 폼, 다이얼로그 박스, 프레임 및 데이터 모듈을 쉽게 공유할 수 있습니다. Object Repository는 폼과 프로젝트를 만들 때 사용자를 안내하는 마법사 및 새 프로젝트에 대한 템플릿도 제공합니다. 리포지토리는 Repository 및 New Items 다이얼로그 박스에 나타나는 항목에 대한 레퍼런스가 들어 있는 텍스트 파일인 BCB.DRO (디폴트로 BIN 디렉토리에 있음)에서 유지 보수됩니다.

프로젝트 내에서 항목 공유

항목을 Object Repository에 추가하지 않고 프로젝트 *내에서* 공유할 수 있습니다. New Items 다이얼로그 박스(File|New|Other)를 열면 현재 프로젝트의 이름이 있는 페이지 탭을 볼 수 있습니다. 이 페이지에서는 프로젝트에 있는 모든 폼, 다이얼로그 박스 및 데이터 모듈을 나열합니다. 기존 항목에서 새 항목을 파생시키고 필요하면 항목을 사용자 정의할 수 있습니다.

Object Repository에 항목 추가

Object Repository에 새로운 프로젝트, 폼, 프레임 및 데이터 모듈을 추가할 수 있습니다. 다음과 같은 방법으로 Object Repository에 항목을 추가합니다.

1. 항목이 프로젝트이거나 프로젝트에 있는 경우에 해당 프로젝트를 엽니다.
2. 프로젝트인 경우 Project|Add To Repository를 선택합니다. 폼 또는 데이터 모듈인 경우 항목을 마우스 오른쪽 버튼으로 클릭하고 Add To Repository를 선택합니다.
3. 설명, 제목 및 작성자를 입력합니다.
4. New Items 다이얼로그 박스에서 항목을 나타내려는 페이지를 결정한 다음, 페이지의 이름을 입력하거나 Page 콤보 박스에서 페이지의 이름을 선택합니다. 존재하지 않는 페이지 이름을 입력하면 C++Builder가 새 페이지를 만듭니다.
5. Browse를 선택하여 Object Repository에 있는 객체를 나타내는 아이콘을 선택합니다.
6. OK를 선택합니다.

팀 환경에서 객체 공유

네트워크상에서 리포지토리를 사용 가능하게 하여 작업 그룹 또는 개발 팀과 객체를 공유할 수 있습니다. 공유 리포지토리를 사용하려면 팀원 모두 Environment Options 다이얼로그 박스에서 동일한 Shared Repository 디렉토리를 선택해야 합니다.

1 Tools | Environment Options를 선택합니다.

2 Preferences 페이지에서 Shared Repository 패널을 찾습니다. Directory 에디트 박스에서 공유 리포지토리를 검색하려는 디렉토리를 입력합니다. 반드시 모든 팀원이 액세스할 수 있는 디렉토리를 지정해야 합니다.

리포지토리에 항목을 처음 추가하는 경우 BCB.DRO 파일이 없으면 C++Builder가 Shared Repository 디렉토리에 이 파일을 만듭니다.

프로젝트에서 Object Repository 항목 사용

Object Repository의 항목에 액세스하려면 File | New | Other를 선택합니다. New Items 다이얼로그 박스가 나타나 사용 가능한 항목을 모두 보여 줍니다. 사용하려는 항목의 타입에 따라 프로젝트에 항목을 추가하기 위한 다음 세 가지 옵션이 있습니다.

- Copy
- Inherit
- Use

항목 복사

Copy를 선택하여 선택한 항목의 정확한 복사본을 만들어 복사본을 프로젝트에 추가합니다. Object Repository에 있는 항목의 차후 변경 내용은 복사본에 반영되지 않고 복사본에 대해 변경한 내용은 원래의 Object Repository 항목에 영향을 미치지 않습니다.

Copy는 프로젝트 템플릿에서 사용할 수 있는 유일한 옵션입니다.

항목 상속

Inherit를 선택하여 Object Repository의 선택된 항목에서 새 클래스를 파생하고 이 클래스를 프로젝트에 추가합니다. 프로젝트를 다시 컴파일할 때 Object Repository의 항목에서 변경된 모든 내용은 프로젝트 항목의 변경 내용과 더불어 파생된 클래스에 반영됩니다. 파생된 클래스의 변경 내용은 Object Repository의 공유 항목에 영향을 미치지 않습니다.

Inherit는 폼, 다이얼로그 박스 및 데이터 모듈에 사용할 수 있지만 프로젝트 템플릿에는 사용할 수 없습니다. Inherit는 동일한 프로젝트 내에서 항목을 재사용하기 위해 사용할 수 있는 *유일한* 옵션입니다.

항목 사용

선택된 항목 자체를 프로젝트의 일부가 되게 하려면 Use를 선택합니다. 프로젝트 항목의 변경 내용은 Inherit 옵션 또는 Use 옵션으로 항목을 추가한 다른 모든 프로젝트에 반영됩니다. 이 옵션은 신중하게 선택해야 합니다.

Use 옵션은 폼, 다이얼로그 박스 및 데이터 모듈에 사용할 수 있습니다.

프로젝트 템플릿 사용

템플릿은 애플리케이션 개발의 첫 단계에서 사용할 수 있도록 미리 디자인된 프로젝트입니다. 다음과 같은 방법으로 템플릿에서 새 프로젝트를 만듭니다.

1 File|New|Other를 선택하여 New Items 다이얼로그 박스를 표시합니다.

2 Projects 탭을 선택합니다.

3 원하는 프로젝트 템플릿을 선택하고 OK를 선택합니다.

4 Select Directory 다이얼로그 박스에서 새 프로젝트 파일의 디렉토리를 지정합니다.

C++Builder는 템플릿 파일을 지정된 디렉토리에 복사하며, 여기서 템플릿 파일을 수정할 수 있습니다. 변경 내용은 원래 프로젝트 템플릿에 영향을 주지 않습니다.

공유 항목 수정

Object Repository에서 항목을 수정하는 경우, 이 변경 내용은 Use 옵션 또는 Inherit 옵션으로 항목을 추가한 기존 프로젝트 뿐만 아니라 해당 항목을 사용하는 이후의 모든 프로젝트에 영향을 미칩니다. 변경 내용이 다른 프로젝트에 전파되는 것을 막으려면 다음 중 하나를 수행합니다.

- 항목을 복사하고 현재 프로젝트 내에서만 수정합니다.
- 항목을 현재 프로젝트로 복사하고 수정한 다음 항목을 다른 이름으로 Repository에 추가합니다.
- 항목에서 컴포넌트, DLL, 컴포넌트 템플릿 또는 프레임을 만듭니다. 컴포넌트 또는 DLL을 만드는 경우 다른 개발자와 공유할 수 있습니다.

디폴트 프로젝트, 새 폼 및 메인 폼 지정

디폴트로, File|New|Application 또는 File|New|Form을 선택하면 C++Builder는 비어 있는 폼을 표시합니다. 다음과 같이 Repository를 재구성하여 이 동작을 변경할 수 있습니다.

1 Tools|Repository를 선택합니다.

2 디폴트 프로젝트를 지정하려면 Projects 페이지를 선택하고 Objects에 있는 항목을 선택합니다. 그런 다음 New Project 체크 박스를 선택합니다.

3 디폴트 폼을 지정하려면 Forms와 같은 Repository 페이지를 선택한 다음 Objects 아래에 있는 폼을 선택합니다. 새 디폴트 폼(File|New Form)을 지정하려면 New Form 체크 박스를 선택합니다. 새 프로젝트에 사용할 디폴트 메인 폼을 지정하려면 Main Form 체크 박스를 선택합니다.

4 OK를 클릭합니다.

애플리케이션에서 도움말 사용

VCL 및 CLX는 모두 도움말 요청을 여러 외부 도움말 뷰어 중 하나로 전달할 수 있는 객체 기반 메커니즘을 사용하여 애플리케이션에서 도움말을 표시할 수 있게 지원합니다. 이 기능을 지원하려면 애플리케이션에 *ICustomHelpViewer* 인터페이스 및 옵션으로 이 인터페이스의 여러 자손 중 하나를 구현하고 전역 Help Manager에 등록하는 클래스가 있어야 합니다.

VCL은 이러한 모든 인터페이스를 구현하고 애플리케이션과 WinHelp 사이의 링크를 제공하는 *TWinHelpViewer*의 인스턴스를 모든 애플리케이션에 제공합니다. CLX는 개발자가 직접 구현해야 합니다. Windows에서 CLX 애플리케이션은 VCL의 일부로서 제공되는 WinHelpViewer 유닛에 애플리케이션을 정적으로 연결한 경우 이 유닛을 사용할 수 있습니다. 즉, 이 유닛을 VCL 패키지에 연결하는 대신 프로젝트의 일부로 포함시키는 것입니다.

Help Manager는 등록된 뷰어 리스트를 유지 보수하고 다음 두 단계 과정으로 요청을 도움말 뷰어에 전달합니다. 먼저 각 뷰어가 특정 도움말 키워드 또는 컨텍스트를 지원할 수 있는지를 확인한 다음 지원할 수 있다고 응답하는 뷰어에 도움말 요청을 전달합니다.

Windows의 WinHelp와 HyperHelp 또는 Linux의 Man과 Info에 대한 등록된 뷰어가 있는 애플리케이션의 경우와 같이 두 개 이상의 뷰어가 키워드를 지원하는 경우 Help Manager는 애플리케이션 사용자가 호출할 도움말 뷰어를 결정할 수 있는 선택 박스를 표시할 수 있습니다. 그렇지 않으면 Help Manager는 먼저 응답하는 도움말 시스템을 표시합니다.

도움말 시스템 인터페이스

도움말 시스템은 일련의 인터페이스를 통해 애플리케이션과 도움말 뷰어 간의 통신을 가능하게 합니다. 이 인터페이스는 모두 HelpIntfs에 정의되며, 여기에는 Help Manager의 구현도 포함되어 있습니다.

*ICustomHelpViewer*는 제공된 키워드에 기반한 도움말을 표시하는 기능과 특정 뷰어에서 사용할 수 있는 모든 도움말을 나열하는 목차를 표시하는 기능을 제공합니다.

*IExtendedHelpViewer*는 숫자 도움말 컨텍스트에 따른 도움말 표시와 항목 표시를 제공합니다. 대부분의 도움말 시스템에서 항목은 상위 수준 키워드로 사용됩니다. 예를 들면, "IntToStr"는 도움말 시스템에서 키워드가 될 수 있지만 "String manipulation routines"는 항목 이름이 될 수 있습니다.

*ISpecialWinHelpViewer*는 Windows에서 실행하는 애플리케이션이 받을 수 있지만 쉽게 일반화할 수 없는 특화된 WinHelp 메시지에 응답할 수 있습니다. 일반적으로 Windows 환경에서 운영되는 애플리케이션만 이러한 인터페이스를 구현해야 하며, 더욱이 이 인터페이스는 비표준 WinHelp 메시지를 광범위하게 사용하는 애플리케이션에만 필요합니다.

*IHelpManager*는 도움말 뷰어가 애플리케이션의 Help Manager와 통신하고 추가 정보를 요청하기 위한 메커니즘을 제공합니다. 도움말 뷰어가 자신을 등록할 때 *IHelpManager*를 얻게 됩니다.

*IHelpSystem*은 *TApplication*이 도움말 요청을 도움말 시스템으로 전달하는 메커니즘을 제공합니다. *TApplication*은 애플리케이션을 로드할 때 *IHelpSystem*과 *IHelpManager*를 모두 구현하는 객체의 인스턴스를 가져와서 이 인스턴스를 속성으로 엑스포트합니다. 이를 통해 애플리케이션 내의 다른 코드가 필요한 경우에 도움말 요청을 직접 보낼 수 있습니다.

*IHelpSelector*는 둘 이상의 뷰어가 도움말 요청을 처리할 수 있는 경우에 도움말 시스템이 사용자 인터페이스를 호출하여 어떤 도움말 뷰어를 사용할지 묻고 목차를 표시할 수 있는 메커니즘을 제공합니다. Help Manager 코드가 사용 중인 widget 집합이나 클래스 라이브러리와 상관 없이 동일하도록, 이러한 표시 기능은 Help Manager로 직접 생성되지 않습니다.

ICustomHelpViewer 구현

ICustomHelpViewer 인터페이스에는 시스템 수준 정보(예: 특정 도움말 요청과 관련되지 않는 정보)를 Help Manager와 통신하는 데 사용하는 메소드, Help Manager가 제공하는 키워드에 따라 도움말을 보여주는 것과 관련된 메소드, 목차를 표시하기 위한 메소드 등 세 가지 타입의 메소드가 있습니다.

Help Manager와 통신

*ICustomHelpViewer*는 시스템 정보를 Help Manager와 통신하는 데 사용할 수 있는 네 가지 함수를 제공합니다.

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

Help Manager는 다음과 같은 경우에 이 함수들을 사용하여 호출합니다.

- AnsiString *ICustomHelpViewer::GetViewerName*()은 Help Manager가 뷰어의 이름을 알고 싶은 경우(예를 들어, 애플리케이션이 등록된 모든 뷰어 리스트를 표시하도록 요청 받을 경우)에 호출됩니다. 이 정보는 문자열로 반환되며 논리적으로 정적이어야 합니다. 즉, 애플리케이션 실행 중에 변경할 수 없습니다. 멀티바이트 문자 집합은 지원되지 않습니다.
- **void** *ICustomHelpViewer::NotifyID*(**const** int ViewerID)는 뷰어를 식별하는 고유한 쿠키를 뷰어에 제공하기 위해 등록한 **직후**에 호출됩니다. 이 정보는 나중에 사용하기 위해 저장해야 하고, 뷰어가 Help Manager의 통지에 응답하는 것이 아니라 스스로 종료하는 경우에는 Help Manager에 식별 쿠키를 제공하여 Help Manager가 뷰어에 대한 모든 참조를 해제할 수 있도록 해야 합니다. 쿠키를 제공하는 데 실패하거나 잘못된 쿠키를 제공하면 Help Manager는 잘못된 뷰어에 대한 참조를 해제할 수도 있습니다.
- **void** *ICustomHelpViewer::ShutDown*()은 Help Manager가 종료 중이며 도움말 뷰어가 할당한 모든 리소스를 해제해야 한다는 것을 도움말 뷰어에게 알리기 위해 Help Manager에서 호출합니다. 모든 리소스 해제를 이 메소드에 위임하는 것이 좋습니다.
- **void** *ICustomHelpViewer::SoftShutDown*()은 뷰어를 언로드하지 않은 채 외부에서 보이는 도움말 시스템(예:도움말 정보를 표시하는 윈도우)을 닫도록 도움말 뷰어에 요청하기 위해 Help Manager에서 호출합니다.

Help Manager에 정보 요청

도움말 뷰어는 *IHelpManager* 인터페이스를 통해 Help Manager와 통신합니다. 도움말 뷰어가 Help Manager에 등록할 때 이 인터페이스의 인스턴스가 도움말 뷰어에 반환됩니다. 도움말 뷰어는 *IHelpManager*를 사용하여 다음 네 가지를 전달할 수 있습니다.

- 현재 활성인 컨트롤의 윈도우 핸들에 대한 요청
- Help Manager가 현재 활성인 컨트롤의 도움말이 포함되어 있을 것이라고 생각하는 도움말 파일의 이름에 대한 요청
- 이 도움말 파일의 경로에 대한 요청
- 도움말 뷰어가 Help Manager에서 종료하도록 요청한 것이 아닌 다른 것에 응답하여 스스로 종료하려고 한다는 통지

int *IHelpManager::GetHandle*()은 현재 활성인 컨트롤의 핸들을 알아야 하는 경우에 도움말 뷰어에서 호출하며 윈도우 핸들이 반환됩니다.

AnsiString *IHelpManager::GetHandle*()은 현재 활성인 컨트롤의 핸들을 포함하고 있을 것이라고 생각되는 도움말 파일의 이름을 알고 싶은 경우에 도움말 뷰어에서 호출합니다.

void *IHelpManager::Release*()는 도움말 뷰어 연결을 끊는 중이라는 것을 Help Manager에게 알리기 위해 호출합니다. *IHelpManager::ShutDown*()을 통한 요청에 대한 응답으로 호출해서는 안됩니다. 예기치 않는 연결 종료를 Help Manager에게 알리는 데에만 사용됩니다.

키워드 방식 도움말 표시

도움말 요청은 일반적으로 다음 중 하나로 도움말 뷰어에 전달됩니다. 뷰어가 특정한 문자열을 기반으로 도움말을 제공하도록 요청 받는 경우에는 *키워드 방식* 도움말로 전달되고 뷰어가 특정 숫자 식별자를 기반으로 도움말을 제공하도록 요청 받는 경우에는 *컨텍스트 방식* 도움말로 전달됩니다.

CLX 숫자 도움말 컨텍스트는 WinHelp 시스템을 사용하는 Windows 애플리케이션에서 도움말을 요청하는 디폴트 형태입니다. CLX도 숫자 도움말 컨텍스트를 지원하지만, 대부분의 Linux 도움말 시스템에서는 이 컨텍스트를 인식하지 못하므로 CLX 애플리케이션에는 사용하지 않는 것이 좋습니다.

ICustomHelpViewer 구현은 키워드 방식 도움말 요청을 지원하는 데 필요하고, *IExtendedHelpViewer* 구현은 컨텍스트 방식 도움말 요청을 지원하는 데 필요합니다.

*ICustomHelpViewer*는 키워드 방식 도움말을 처리하기 위해 다음과 같은 세 가지 메소드를 제공합니다.

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
int__fastcall ICustomHelpViewer::UnderstandsKeyword(const AnsiString
HelpString)
```

이 구문은 Help Manager에서 호출하는 세 가지 메소드 중 첫 번째이며, 뷰어가 해당 문자열에 대한 도움말을 제공하는지 묻기 위해 같은 문자열을 사용하여 등록된 *각* 도움말을 호출합니다. 뷰어는 해당 도움말 요청에 응답하여 표시할 수 있는 다른 도움말 페이지의 수를 나타내는

정수로 응답합니다. 뷰어는 이 수를 확인하기 위해 필요한 모든 메소드를 사용할 수 있습니다. IDE 내부에서 HyperHelp 뷰어는 고유한 인덱스를 유지하며 이를 검색합니다. 뷰어가 이 키워드에 대한 도움말을 지원하지 않으면 뷰어는 0을 반환합니다. 음수는 현재 0를 의미하는 것으로 해석되지만 이 동작은 이후 릴리스에서는 보장되지 않습니다.

```
Classes::TStringList*__fastcall ICustomHelpViewer::GetHelpStrings(const
AnsiString HelpString)
```

이 구문은 하나 이상의 뷰어가 항목에 대한 도움말을 제공할 수 있는 경우에 Help Manager에서 호출합니다. 뷰어는 Help Manager에서 해제하는 *TStringList*를 반환합니다. 반환된 리스트에 있는 문자열은 해당 키워드에 사용할 수 있는 페이지에 매핑되어야 하지만 매핑의 특성은 뷰어에서 결정할 수 있습니다. Windows의 WinHelp 뷰어와 Linux의 HyperHelp 뷰어의 경우 문자열 리스트는 항상 하나의 항목만을 포함합니다. HyperHelp는 자체 인덱싱을 제공하며 인덱싱을 다른 곳에 복제하는 것은 의미가 없습니다. Man 페이지 뷰어(Linux)의 경우 문자열 리스트는 해당 키워드에 대한 페이지를 포함하는 매뉴얼의 각 섹션에 대해 하나씩, 여러 문자열로 구성됩니다.

```
void__fastcall ICustomHelpViewer::ShowHelp(const AnsiString HelpString)
```

이 구문은 도움말 뷰어에서 특정 키워드에 대한 도움말을 표시하도록 해야 하는 경우에 Help Manager에서 호출합니다. 이것은 이 작업에서 마지막 메소드 호출이므로 *UnderstandsKeyword*가 먼저 호출되지 않은 경우에는 호출할 수 없습니다.

목차 표시

*ICustomHelpViewer*는 목차 표시와 관련된 두 가지 메소드를 제공합니다.

- *CanShowTableOfContents*
- *ShowTableOfContents*

이 메소드 작업의 이론은 다음 키워드 도움말 요청 함수의 작업과 유사합니다. 즉, Help Manager는 먼저 *ICustomHelpViewer::CanShowTableOfContents()*를 호출하여 모든 도움말 뷰어를 쿼리한 다음 *ICustomHelpViewer::ShowTableOfContents*()를 호출하여 특정 도움말 뷰어를 호출합니다.

목차 지원에 대한 요청은 특정 뷰어에서 거부하는 것이 합리적입니다. 예를 들면 목차의 개념이 Man 페이지의 작동 방식에 제대로 매핑되지 않기 때문에 Man 페이지 뷰어는 목차 지원 요청을 거부합니다. 이와 대조적으로 HyperHelp 뷰어는 목차를 표시하라는 요청을 HyperHelp(Linux) 및 WinHelp(Windows)에 직접 전달함으로써 목차를 지원합니다. 그러나 *ICustomHelpViewer*의 구현이 *CanShowTableOfContents*를 통한 쿼리에 *true*로 응답하고 *ShowTableOfContents*를 통한 요청을 무시하는 것은 합리적이지 *않습니다*.

IExtendedHelpViewer 구현

*ICustomHelpViewer*는 키워드 방식 도움말에 대한 직접적인 지원만을 제공합니다. 일부 도움말 시스템(특히, WinHelp)은 도움말 시스템 내부에서 즉, 애플리케이션에 보이지 않는 방식으로 번호(*컨텍스트 ID*라고 함)를 키워드에 연결하여 작동합니다. 이런 도움말 시스템에서는 애플리케이션이 문자열이 아니라 컨텍스트로 도움말 시스템을 호출하는 컨텍스트 방식 도움말을 애플리케이션에서 지원해야 하고 도움말 시스템은 번호 자체를 번역합니다.

VCL 또는 CLX로 작성된 애플리케이션은 *ICustomHelpViewer*를 구현하는 객체를 확장함으로써 컨텍스트 방식 도움말을 요구하는 시스템과 통신하여 *IExtendedHelpViewer*도 구현할 수 있습니다. *IExtendedHelpViewer*를 사용하면 키워드 검색을 사용하지 않고 상위 수준 항목으로 직접 이동할 수 있도록 하는 도움말 시스템과 통신할 수도 있습니다. 기본 WinHelp 뷰어는 이런 작업을 자동으로 수행합니다.

*IExtendedHelpViewer*는 네 가지 함수를 노출합니다. 이 함수 중 두 개(*UnderstandsContext* 및 *DisplayHelpByContext*)는 컨텍스트 방식 도움말을 지원하는 데 사용하고 나머지 두 개(*UnderstandsTopic* 및 *DisplayTopic*)는 항목을 지원하는 데 사용합니다.

애플리케이션 사용자가 F1 키를 누르면 Help Manager는 다음을 호출합니다.

```
int__fastcall IExtendedHelpViewer::UnderstandsContext(const int
ContextID, AnsiString HelpFileName)
```

그리고 현재 활성화된 컨트롤은 키워드 방식 도움말이 아닌 컨텍스트 방식 도움말을 지원합니다. *ICustomHelpViewer::UnderstandsKeyword*()에서와 마찬가지로 Help Manager는 등록된 모든 도움말 뷰어를 반복해서 쿼리합니다. *ICustomHelpViewer::UnderstandsKeyword*()에서와는 달리 둘 이상의 뷰어가 지정된 컨텍스트를 지원하는 경우 등록된 뷰어 중 지정된 컨텍스트를 지원하는 *첫 번째* 뷰어가 호출됩니다.

Help Manager는 다음 구문을 호출하기 전에

```
void__fastcall IExtendedHelpViewer::DisplayHelpByContext(const int
ContextID, AnsiString
HelpFileName)
```

등록된 도움말 뷰어를 폴링(polling)합니다.

항목 지원 함수는 다음과 같이 동일한 방식으로 작동합니다.

```
bool__fastcall IExtendedHelpViewer::UnderstandsTopic(const AnsiString
Topic)
```

위 구문은 항목을 지원하는지 묻는 도움말 뷰어를 폴링(polling)하는 데 사용되고,

```
void__fastcall IExtendedHelpViewer::DisplayTopic(const AnsiString Topic)
```

위 구문은 항목에 대한 도움말을 제공할 수 있다고 보고하는 첫째로 등록된 뷰어를 호출하는 데 사용됩니다.

IhelpSelector 구현

*IHelpSelector*는 *ICustomHelpViewer*의 짝(companion)입니다. 둘 이상의 등록된 뷰어가 지정한 키워드, 컨텍스트 또는 항목에 대한 지원을 제공하거나 목차를 제공하겠다고 요청하면 Help Manager는 이들 중에서 하나를 선택해야 합니다. 컨텍스트 또는 항목의 경우 Help Manager는 *항상* 먼저 지원을 제공하겠다고 주장하는 첫 번째 도움말 뷰어를 선택합니다. 키워드나 컨텍스트 테이블의 경우 Help Manager는 디폴트로 첫 번째 도움말 뷰어를 선택합니다. 이 동작은 애플리케이션에서 오버라이드할 수 있습니다.

이런 경우 애플리케이션이 Help Manager의 선택을 오버라이드하려면 *IHelpSelector* 인터페이스의 구현을 제공하는 클래스를 등록해야 합니다. *IHelpSelector*는 *SelectKeyword* 및 *TableOfContents* 등 두 개의 함수를 엑스포트합니다. 두 함수는 가능한 키워드 일치 사항이나 목차 제공을 요청하는 뷰어의 이름을 하나씩 포함하는 *TStrings*를 인수로 취합니다. 구현자(implementor)는 선택된 문자열을 나타내는 *TStringList*의 인덱스를 반환해야 합니다. 그러면 Help Manager에서 *TStringList*를 해제합니다.

참고 문자열이 다시 정렬되면 Help Manager에서 혼동할 수 있으므로 *IHelpSelector* 구현자의 이러한 동작은 삼가는 것이 좋습니다. 도움말 시스템은 *하나의* 도움말 선택자만 지원하므로 새 선택자(selector)가 등록되면 기존 선택자는 연결이 끊어집니다.

도움말 시스템 객체 등록

Help Manager가 *ICustomHelpViewer*, *IExtendedHelpViewer*, *ISpecialWinHelpViewer* 및 *IHelpSelector*를 구현하는 객체와 통신하려면 해당 객체를 Help Manager에 등록해야 합니다.

도움말 시스템 객체를 Help Manager에 등록하려면 다음 작업을 수행해야 합니다.

- 도움말 뷰어 등록
- 도움말 선택자 등록

도움말 뷰어 등록

객체 구현을 포함하는 유닛은 HelpIntfs를 사용해야 합니다. 객체의 인스턴스는 구현 유닛의 헤더 파일에서 선언해야 합니다.

구현 유닛은 인스턴스 변수를 할당하여 함수 *RegisterViewer*에 전달하는 메소드를 호출하는 pragma 시작 지시어를 포함해야 합니다. *RegisterViewer*는 *ICustomHelpViewer*를 인수로 사용하고 *IHelpManager*를 반환하는 HelpIntfs.pas에서 엑스포트되는 단일 차원 함수입니다. *IHelpManager*는 나중에 사용하기 위해 저장해야 합니다.

해당 .cpp 파일은 인터페이스를 등록하기 위한 코드를 포함합니다. 위에서 설명한 인터페이스를 위한 등록 코드는 다음과 같습니다.

```
void InitServices()
{
  THelpImplementor GlobalClass;
  Global = dynamic_cast<ICustomHelpViewer*>(GlobalClass);
  Global->AddRef;
  HelpIntfs::RegisterViewer(Global, GlobalClass->Manager);
}
#pragma startup InitServices
```

참고 Help Manager 객체가 아직 해제되지 않은 경우 GlobalClass 객체의 소멸자에서 이 객체를 해제해야 합니다.

도움말 선택자 등록

객체 구현을 포함하는 유닛은 Forms(VCL) 또는 Qforms(CLX)를 사용해야 합니다. 객체의 인스턴스는 구현 유닛의 .cpp 파일에서 선언해야 합니다.

구현 유닛은 전역 Application 객체의 *HelpSystem* 속성을 통해 도움말 선택자를 등록해야 합니다.

```
Application->HelpSystem->AssignHelpSelector(myHelpSelectorInstance)
```

이 프로시저는 값을 반환하지 않습니다.

VCL 애플리케이션에서 도움말 사용

다음 단원에서는 VCL 애플리케이션 내에서 도움말을 사용하는 방법에 대해 설명합니다.

- TApplication이 VCL 도움말을 처리하는 방법
- VCL 컨트롤이 도움말을 처리하는 방법
- 도움말 시스템 직접 호출
- IHelpSystem 사용

TApplication이 VCL 도움말을 처리하는 방법

VCL의 *TApplication*은 애플리케이션 코드에서 액세스할 수 있는 네 가지 메소드를 제공합니다.

표 7.5 Tapplication의 도움말 메소드

HelpCommand	Windows 도움말 스타일인 HELP_COMMAND를 가져와서 WinHelp에 전달합니다. 이 메커니즘을 통해 전달된 도움말 요청은 IspecialWinHelpViewer 구현에만 전달됩니다.
HelpContext	컨텍스트 방식 도움말을 요청하며 도움말 시스템을 호출합니다.
HelpKeyword	키워드 방식 도움말을 요청하며 도움말 시스템을 호출합니다.
HelpJump	특성 항목 표시를 요청합니다.

네 가지 함수 모두 자신에게 전달된 데이터를 가져와서 도움말 시스템을 나타내는 *TApplication*의 데이터 멤버를 통해 이 데이터를 전달합니다. 이 데이터 멤버는 *HelpSystem* 속성을 통해 직접 액세스할 수 있습니다.

VCL 컨트롤이 도움말을 처리하는 방법

*TControl*에서 파생하는 모든 VCL 컨트롤은 *HelpType*, *HelpContext* 및 *HelpKeyword* 등 도움말 시스템에서 사용되는 여러 속성을 노출합니다.

HelpType 속성은 컨트롤 디자이너가 키워드 방식 도움말이나 컨텍스트 방식 도움말 중 어떤 도움말을 예상할지 결정하는 열거 타입의 인스턴스를 포함합니다. *HelpType*이 *htKeyword*로 설정되는 경우, 도움말 시스템은 컨트롤이 키워드 방식 도움말을 사용하는 것으로 예상하고 도움말 시스템은 *HelpKeyword* 속성의 내용만 봅니다. 이와 반대로 *HelpType*이 *htContext*로 설정되는 경우, 도움말 시스템은 컨트롤이 컨텍스트 방식 도움말을 사용하는 것으로 예상하고 도움말 시스템은 *HelpContext* 속성의 내용만 봅니다.

컨트롤은 속성 이외에 도움말 시스템에 요청을 전달하기 위해 호출할 수 있는 하나의 메소드(*InvokeHelp*)를 노출합니다. 이 메소드는 매개변수를 사용하지 않고 전역 Application 객체의 메소드 중 컨트롤이 지원하는 도움말 타입에 해당하는 메소드를 호출합니다.

*TWidgetControl*의 *KeyDown* 메소드가 *InvokeHelp*를 호출하므로 F1 키를 누르면 도움말 메시지가 자동으로 호출됩니다.

CLX 애플리케이션에서 도움말 사용

다음 단원에서는 CLX 애플리케이션 내에서 도움말을 사용하는 방법에 대해 설명합니다.

- TApplication이 도움말을 처리하는 방법
- CLX 컨트롤이 도움말을 처리하는 방법
- 도움말 시스템 직접 호출
- IHelpSystem 사용

TApplication이 도움말을 처리하는 방법

*TApplication*은 애플리케이션 코드에서 액세스할 수 있는 다음과 같은 두 가지 메소드를 제공합니다.

- *ContextHelp*는 컨텍스트 방식 도움말 요청을 사용하여 도움말 시스템을 호출합니다.
- *KeywordHelp*는 키워드 방식 도움말 요청을 사용하여 도움말 시스템을 호출합니다.

이 두 함수는 전달되는 컨텍스트 또는 키워드를 인수로 사용하고 도움말 시스템을 나타내는 *TApplication*의 데이터 멤버를 통해 요청을 전달합니다. 이 데이터 멤버는 읽기 전용 속성인 *HelpSystem*을 통해 직접 액세스할 수 있습니다.

CLX 컨트롤이 도움말을 처리하는 방법

*TControl*에서 파생하는 모든 컨트롤은 *HelpType*, *HelpFile*, *HelpContext* 및 *HelpKeyword* 등 도움말 시스템에서 사용되는 네 가지 속성을 노출합니다. *HelpFile*은 컨트롤의 도움말이 있는 파일 이름을 포함하는 것으로 간주됩니다. 파일 이름에 대해 신경 쓰지 않는 외부 도움말 시스템(예 : Man 페이지 시스템)이 있는 경우 이 속성은 비어 있어야 합니다.

HelpType 속성은 컨트롤의 디자이너가 키워드 방식 도움말이나 컨텍스트 방식 도움말 중 어떤 도움말이 제공될 것인지를 결정하는 열거 타입의 인스턴스를 포함합니다. 나머지 두 속성은 HelpType 속성에 연결됩니다. *HelpType*이 *htKeyword*로 설정되는 경우 도움말 시스템은 컨트롤이 키워드 방식 도움말을 사용하는 것으로 예상하고 *HelpKeyword* 속성의 내용만 봅니다. 이와 반대로 *HelpType*이 *htContext*로 설정되는 경우 도움말 시스템은 컨트롤이 컨텍스트 방식 도움말을 사용하는 것으로 예상하고 *HelpContext* 속성의 내용만 봅니다.

컨트롤은 속성 이외에 요청을 도움말 시스템으로 전달하기 위해 호출되는 하나의 메소드인 *InvokeHelp*를 노출합니다. 이 메소드는 매개변수를 사용하지 않고 전역 Application 객체의 메소드 중 컨트롤이 지원하는 도움말 타입에 해당하는 메소드를 호출합니다.

*TWidgetControl*의 *KeyDown* 메소드가 *InvokeHelp*를 호출하므로 F1 키를 누르면 도움말 메시지가 자동으로 호출됩니다.

도움말 시스템 직접 호출

VCL 또는 CLX에서 제공되지 않는 추가 도움말 시스템 기능을 위해 *TApplication*은 도움말 시스템을 직접 액세스할 수 있도록 하는 읽기 전용 속성을 제공합니다. 이 속성은 인터페이스 *IHelpSystem* 구현의 인스턴스입니다. *IHelpSystem*과 *IHelpManager*는 동일한 객체에서 구현되지만, 한 인터페이스는 애플리케이션이 Help Manager와 통신할 수 있게 하는 데 사용되고 다른 인터페이스는 도움말 뷰어가 Help Manager와 통신할 수 있게 하는 데 사용됩니다.

IHelpSystem 사용

*IHelpSystem*을 사용하면 VCL 또는 CLX 애플리케이션이 다음 세 가지 작업을 할 수 있습니다.

- Help Manager에 Path Info 제공
- 새 도움말 선택자 제공
- Help Manager에게 도움말을 표시하도록 요청

도움말 선택자를 할당하면 여러 외부 도움말 시스템에서 동일한 키워드에 대한 도움말을 제공할 수 있는 경우 Help Manager가 의사 결정을 위임할 수 있습니다. 자세한 내용은 7-31페이지의 "IhelpSelector 구현" 단원을 참조하십시오.

*IHelpSystem*은 Help Manager에게 도움말을 표시하도록 요청하기 위해 다음과 같은 네 개의 프로시저와 한 개의 함수를 엑스포트합니다.

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

*Hook*는 전적으로 WinHelp 호환성을 위해 고안되었으므로 CLX 애플리케이션에서 사용해서는 안됩니다. 이를 사용하여 키워드 방식, 컨텍스트 방식 및 항목 방식 도움말에 대한 요청에 직접적으로 매핑할 수 없는 WM_HELP 메시지를 처리할 수 있습니다. 다른 메서드마다 도움말을 요청할 키워드, 컨텍스트 ID 또는 항목 중 하나와 도움말이 있을 것으로 예상되는 도움말 파일 등 두 가지 인수를 취합니다.

일반적으로 항목 방식 도움말을 요청하지 않는 경우 컨트롤의 *InvokeHelp* 메소드를 통해 도움말 요청을 Help Manager에 전달하는 것이 효율적이고 더욱 확실합니다.

IDE 도움말 시스템 사용자 정의

C++Builder IDE 는 VCL 또는 CLX 애플리케이션과 정확히 같은 방법으로 여러 도움말 뷰어를 지원합니다. 즉, 도움말 요청을 Help Manager에 위임하고 Help Manager는 요청을 등록된 도움말 뷰어에 전달합니다. IDE는 VCL이 사용하는 것과 동일한 WinHelpViewer를 사용합니다.

IDE에서 새 도움말 뷰어를 설치하려면 한 가지만 제외하고는 VCL이나 CLX 애플리케이션에서에서 하는 것과 똑같이 수행하십시오. *ICustomHelpViewer* 및 원하는 경우 *IExtendedHelpViewer*를 구현하는 객체를 작성하여 도움말 요청을 자신이 선택한 외부 뷰어에 전달하고 *ICustomHelpViewer*를 IDE에 등록합니다.

다음과 같은 방법으로 사용자 정의 도움말 뷰어를 IDE에 등록합니다.

1. 도움말 뷰어를 구현하는 유닛이 HelpIntfs.cpp를 포함하는지 확인합니다.
2. 유닛을 IDE에 등록된 디자인 타임 패키지로 생성하고 런타임 패키지를 선택한 상태에서 해당 패키지를 생성합니다. 이것은 유닛이 사용하는 Help Manager 인스턴스가 IDE가 사용하는 Help Manager 인스턴스와 동일하다는 것을 확인하는 데 필요합니다.
3. 도움말 뷰어가 유닛 내에서 전역 인스턴스로 존재하는지 확인합니다.
4. #pragma 시작으로 차단된 초기화 함수에서 인스턴스가 *RegisterHelpViewer* 함수로 전달되는지 확인합니다.

장

8

애플리케이션 사용자 인터페이스 개발

C++Builder를 열거나 새 프로젝트를 만들면 화면에 비어 있는 폼이 표시됩니다. 애플리케이션의 사용자 인터페이스(UI)를 디자인할 때는 윈도우, 메뉴 및 다이얼로그 박스와 같은 비주얼(visual) 컴포넌트를 컴포넌트 팔레트에서 폼으로 가져옵니다.

비주얼(visual) 컴포넌트가 폼에 있으면 위치, 크기 및 기타 디자인 타임 속성을 조정하고 이벤트 핸들러를 코딩할 수 있습니다. C++Builder는 원본으로 사용하는 세부적인 프로그래밍을 제공합니다.

다음 단원에서는 폼 작업, 컴포넌트 템플릿 생성, 다이얼로그 박스 추가 및 메뉴와 툴바에 대한 액션 구성과 같은 주요 인터페이스 작업에 대해 설명합니다.

애플리케이션 동작 제어

TApplication, *TScreen* 및 *TForm*은 프로젝트 동작을 제어하여 모든 C++Builder 애플리케이션의 백본을 형성하는 클래스입니다. *TApplication* 클래스는 표준 프로그램의 동작을 캡슐화하는 속성과 메소드를 제공하여 애플리케이션의 기초를 형성합니다. *TScreen*은 화면 해상도 및 사용 가능한 디스플레이 글꼴과 같은 시스템 관련 정보 관리 뿐만 아니라 로드된 폼과 데이터 모듈 추적을 위해 런타임에서 사용됩니다. *TForm* 클래스의 인스턴스는 애플리케이션 사용자 인터페이스의 빌딩 블록입니다. 애플리케이션의 윈도우와 다이얼로그 박스는 *TForm*을 기반으로 합니다.

애플리케이션 레벨 작업

TApplication 타입의 전역 변수 *Application*은 모든 VCL 또는 CLX 기반의 애플리케이션에 있습니다. *Application*은 프로그램의 백그라운드에서 발생하는 많은 기능을 제공하고 애플리케이션을 캡슐화합니다. 예를 들어, *Application*은 프로그램 메뉴에서 도움말 파일을 호출하는 방법을 처리합니다. *TApplication*의 작동 방법에 대한 이해는 독립 실행형 애플리케이션 개발자보다는 컴포넌트 작성자에게 더 중요하지만 프로젝트를 작성하는 경우 Project|Options Application 페이지에서 *Application*이 처리하는 옵션을 설정해야 합니다.

또한 *Application*은 애플리케이션 전체에 적용되는 많은 이벤트를 받습니다. 예를 들어, *OnActivate* 이벤트를 사용하면 애플리케이션이 처음 시작될 때 액션을 수행할 수 있고, *OnIdle* 이벤트는 애플리케이션이 사용되지 않을 때 백그라운드 프로세스를 수행할 수 있고, *OnMessage* 이벤트는 Windows 메시지를 인터셉트할 수 있고(Windows에서만), *OnEvent* 이벤트는 이벤트를 인터셉트할 수 있습니다. IDE를 사용하여 전역 *Application* 변수의 이벤트와 속성을 검사할 수 없지만 다른 컴포넌트인 *TApplicationEvents*는 IDE를 사용하여 이벤트 핸들러를 제공하고 이벤트를 인터셉트할 수 있습니다.

화면 처리

TScreen 타입의 전역 변수 *Screen*은 프로젝트를 만들 때 만들어집니다. *Screen*은 애플리케이션이 실행되는 화면의 상태를 캡슐화합니다. *Screen*을 사용하여 다음과 같은 일반적인 작업을 수행할 수 있습니다.

- 커서 모양 지정
- 애플리케이션이 실행 중인 윈도우 크기 지정
- 화면 장치에서 사용 가능한 글꼴 리스트 지정
- 여러 화면 동작 지정(Windows만 해당)

Windows 애플리케이션이 여러 모니터에서 실행되는 경우에는 사용자 인터페이스의 레이아웃을 효과적으로 관리할 수 있도록 *Screen*이 모니터 및 해당 치수 리스트를 관리합니다.

크로스 플랫폼 프로그래밍에 CLX를 사용하는 경우에는 디폴트로 애플리케이션이 현재 화면 장치에 대한 정보에 따라 화면 컴포넌트를 만들어 *Screen*에 할당합니다.

폼 설정

*TForm*은 GUI 애플리케이션을 작성하기 위한 주요 클래스입니다. 디폴트 프로젝트를 표시하는 C++Builder 를 열거나 새 프로젝트를 만들 때는 UI 디자인을 시작할 수 있는 폼이 나타납니다.

메인 폼 사용

디폴트로, 프로젝트에서 처음 만들어서 저장하는 폼이 메인 폼이 되며 런타임에 처음 만들어집니다. 프로젝트에 폼을 추가함에 따라 다른 폼을 애플리케이션의 메인 폼으로 지정할 수 있습니다. 또한 폼을 메인 폼으로 지정하면 폼 작성 순서를 바꾸지 않는 한 메인 폼이 실행 중인 애플리케이션에 나타나는 첫 번째 폼이 되므로 런타임에 쉽게 테스트할 수 있습니다.

다음과 같은 방법으로 프로젝트 메인 폼을 변경합니다.

1 Project|Options를 선택한 다음 Forms 페이지를 선택합니다.

2 Main Form 콤보 박스에서 프로젝트 메인 폼으로 사용할 폼을 선택한 다음 OK를 선택합니다.

이제 애플리케이션을 실행하면 메인 폼으로 선택한 폼이 먼저 표시됩니다.

메인 폼 숨기기

애플리케이션을 처음 시작할 때 메인 폼이 나타나지 않게 할 수 있습니다. 이렇게 하려면 8-1페이지의 "애플리케이션 레벨 작업"에서 설명한 전역 *Application* 변수를 사용해야 합니다.

다음과 같은 방법으로 시작할 때 메인 폼을 숨깁니다.

1 Project | View Source를 선택하여 메인 프로젝트 파일을 표시합니다.

2 `Application->CreateForm()` 호출과 `Application->Run()` 호출 사이에 다음 줄을 추가합니다.

```
Application->ShowMainForm = false;
```

3 Object Inspector를 사용하여 메인 폼의 *Visible* 속성을 **false**로 설정합니다.

폼 추가

프로젝트에 폼을 추가하려면 File | New | Form을 선택하십시오. Project Manager(View | Project Manager)에 나열된 프로젝트의 폼과 해당 관련 유닛을 모두 볼 수 있으며 View | Forms를 선택하여 폼 리스트만 볼 수도 있습니다.

폼 연결

프로젝트에 폼을 추가하면 해당 폼에 대한 참조가 프로젝트 파일에는 추가되지만 프로젝트에 있는 다른 유닛에는 추가되지 않습니다. 새 폼을 참조하는 코드를 작성하려면 먼저 참조하는 폼의 유닛 파일에 해당 폼에 대한 참조를 추가해야 합니다. 이것을 *폼 연결*이라고 합니다.

폼을 연결하는 이유는 일반적으로 해당 폼에 있는 컴포넌트에 대한 액세스를 제공하기 위해서입니다. 예를 들어, 폼 연결을 사용하여 데이터 인식 컴포넌트가 있는 폼을 데이터 모듈에 있는 data-access 컴포넌트에 연결할 수 있습니다.

다음과 같은 방법으로 하나의 폼을 다른 폼에 연결합니다.

1 다른 폼을 참조할 폼을 선택합니다.

2 File | Include Unit Hdr을 선택합니다.

3 참조할 폼의 폼 유닛 이름을 선택합니다.

4 OK를 선택합니다.

하나의 폼을 다른 폼에 연결한다는 것은 단지 하나의 폼 유닛에 다른 폼 유닛에 대한 헤더가 들어 있어 연결된 폼과 해당 컴포넌트가 이제 연결 폼의 유효 범위에 있다는 것을 의미합니다.

레이아웃 관리

폼에서 컨트롤을 둘 위치를 통해 사용자 인터페이스의 레이아웃을 간단히 제어합니다. 선택한 위치는 컨트롤의 *Top*, *Left*, *Width* 및 *Height* 속성에 반영됩니다. 런타임에 이러한 값을 변경하여 폼에서의 컨트롤 위치와 크기를 변경할 수 있습니다.

그러나 컨트롤에는 내용이나 컨테이너에 따라 자동으로 조정할 수 있는 여러 가지 다른 속성이 있습니다. 이러한 속성을 사용하면 각 요소가 하나의 통합체로서 맞춰지도록 폼의 레이아웃을 설정할 수 있습니다.

컨트롤이 해당 부모와 관련하여 위치와 크기가 조정되게 하는 방법에 영향을 주는 속성에는 두 가지가 있습니다. *Align* 속성을 사용하면 컨트롤이 해당 부모 내에서 특정 가장자리를 따라 맞춰지거나, 다른 컨트롤이 모두 정렬된 다음 전체 클라이언트 영역을 채우도록 할 수 있습니다. 부모의 크기가 조정되면 부모에 맞게 정렬된 컨트롤이 특정 가장자리에 비례하여 맞춰지도록 위치는 그대로 유지하면서 크기가 자동으로 조정됩니다.

컨트롤이 해당 부모의 특정 가장자리에 따라 상대적으로 위치를 유지하면서 가장자리에 붙거나 항상 전체 가장자리를 따라 실행되도록 크기를 조정하고 싶지는 않을 때는 *Anchors* 속성을 설정합니다.

컨트롤이 너무 크거나 작아지지 않게 하려면 *Constraints* 속성을 사용합니다. *Constraints* 를 사용하면 컨트롤의 최대 높이, 최소 높이, 최대 너비 및 최소 너비를 지정할 수 있습니다. 이러한 값을 설정하여 컨트롤의 높이와 너비의 크기(픽셀)를 제한합니다. 예를 들어, 컨테이너 객체의 제약 조건에서 *MinWidth* 및 *MinHeight*를 설정하여 자식 객체가 항상 표시되게 할 수 있습니다.

객체에는 크기 제약 조건이 있는 정렬된 자식이 있으므로 *Constraints* 값은 객체의 크기가 제한되도록 부모/자식 계층을 따라 전달됩니다. 또한 *Constraints*를 사용하면 해당 *ChangeScale* 메소드가 호출될 때 컨트롤이 특정 치수로 배율 조정되는 것을 방지할 수 있습니다.

*TControl*은 *TConstrainedResizeEvent* 타입의 protected 이벤트인 *OnConstrainedResize*를 제공합니다.

```
void __fastcall (__closure *TConstrainedResizeEvent)(System::TObject* Sender,
int &MinWidth, int &MinHeight, int &MaxWidth, int &MaxHeight);
```

이 이벤트를 사용하면 컨트롤의 크기를 조정하려고 할 때 크기 제약 조건을 오버라이드할 수 있습니다. 제약 조건 값은 이벤트 핸들러 내에서 변경될 수 있는 `var` 매개변수로 전달됩니다. *OnConstrainedResize*는 컨테이너 객체 (*TForm*, *TScrollBox*, *TControlBar* 및 *TPanel*)를 위해 게시됩니다. 또한 컴포넌트 작성자는 *TControl*의 자손을 위해 이 이벤트를 사용하거나 게시할 수 있습니다.

크기를 변경할 수 있는 항목이 있는 컨트롤은 컨트롤 자체 크기를 글꼴에 맞게 조정하거나 포함된 객체에 맞게 조정할 수 있도록 *AutoSize* 속성을 가집니다.

폼 사용

IDE에서 C++Builder의 폼을 만들 때 C++Builder는 애플리케이션 기능의 메인 엔트리 포인트에 코드를 포함시켜 자동으로 메모리에 폼을 만듭니다. 보통 이 동작은 매우 바람직하며 다른 부분을 변경할 필요가 없습니다. 즉, 메인 윈도우가 프로그램이 실행되는 동안 계속 표시되므로 메인 윈도우를 위해 폼을 만들 때 디폴트 C++Builder 동작을 변경할 필요가 없습니다.

그러나 프로그램이 실행되는 동안 메모리에서 애플리케이션의 모든 폼을 원하지 않을 수도 있습니다. 즉, 한 번에 애플리케이션의 모든 다이얼로그 박스가 메모리에 있는 것을 원하지 않을 경우 다이얼로그 박스가 나타나기를 원할 때 동적으로 해당 다이얼로그 박스를 만들 수 있습니다.

폼은 modal 폼이거나 modeless 폼일 수 있습니다. modal 폼은 사용자 입력이 필요한 다이얼로그 박스처럼 사용자가 다른 폼으로 전환하기 전에 상호 작용해야 하는 폼입니다. modeless 폼은 다른 윈도우에 의해 가려지거나 사용자가 닫거나 최소화할 때까지 표시되는 윈도우입니다.

메모리에 있는 폼 제어

디폴트로, C++Builder는 다음 코드를 애플리케이션의 메인 엔트리 포인트에 포함시켜 자동으로 메모리에 애플리케이션의 메인 폼을 만듭니다.

```
Application ->CreateForm(__classid(TForm1), &Form1);
```

이 기능은 폼과 같은 이름의 전역 변수를 만듭니다. 따라서 애플리케이션의 모든 폼에는 관련 전역 변수가 있습니다. 이 변수는 폼 클래스의 포인터이고 애플리케이션이 실행되는 동안 폼을 참조하는 데 사용됩니다. 폼의 헤더(.h) 파일을 포함하는 소스 코드(.cpp) 파일은 이 변수를 통해 폼에 액세스할 수 있습니다.

폼은 애플리케이션의 메인 엔트리 포인트에 추가되므로 프로그램을 호출할 때 나타나며 애플리케이션이 실행되는 동안 메모리에 존재합니다.

자동 생성된 폼 표시

시작할 때 폼을 만들도록 선택하고, 프로그램이 일정 시간 동안 실행될 때까지 이 폼이 표시되지 않게 하는 경우 폼의 이벤트 핸들러는 *ShowModal* 메소드를 사용하여 메모리에 이미 로드되어 있는 폼을 표시합니다.

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  ResultsForm->ShowModal();
}
```

이 경우에는 메모리에 이미 폼이 있으므로 다른 인스턴스를 만들거나 해당 인스턴스를 소멸할 필요가 없습니다.

동적으로 폼 생성

애플리케이션의 폼이 메모리에 동시에 있는 것을 원하지 않을 때도 있습니다. 로드 시간에 필요한 메모리 양을 줄이려면 사용해야 할 때만 일부 폼을 작성합니다. 예를 들어, 다이얼로그 박스를 사용자가 상호 작용할 때만 메모리에 있게 할 수 있습니다.

다음과 같은 방법으로 IDE를 사용하여 실행하는 동안 다른 단계에서 폼을 만듭니다.

1. 메인 메뉴에서 File|New|Form을 선택하여 새로운 폼을 표시합니다.
2. Project|Options|Forms 페이지의 자동 생성 폼 리스트에서 해당 폼을 제거합니다.

   이렇게 하면 폼 호출이 제거됩니다. 또는 프로그램의 메인 엔트리 포인트에서 다음 줄을 수동으로 제거할 수도 있습니다.

   ```
   Application->CreateForm(__classid(TResultsForm), &ResultsForm);
   ```

3. modeless 폼일 경우 폼의 *Show* 메소드를 사용하고, modal 폼의 경우에는 *ShowModal* 메소드를 사용하여 필요할 때 폼을 호출합니다.

메인 폼의 이벤트 핸들러는 결과 폼의 인스턴스를 만들어 삭제해야 합니다. 결과 폼을 호출하는 한 가지 방법은 다음과 같이 전역 변수를 사용하는 것입니다. *ResultsForm*은 modal 폼이므로 핸들러에서 *ShowModal* 메소드를 사용할 수 있습니다.

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  ResultsForm = new TResultsForm(this);
 ResultsForm->ShowModal();
 delete ResultsForm;
}
```

예제의 이벤트 핸들러는 폼을 닫은 다음 삭제하므로 애플리케이션의 다른 곳에서 *ResultsForm*을 사용해야 할 경우에는 **new**를 사용하여 다시 만들어야 합니다. 폼이 *Show*를 사용하여 표시된 경우에는 폼이 열려 있는 동안 *Show*가 폼을 반환하므로 이벤트 핸들러 내에서 폼을 삭제할 수 없습니다.

참고 **new** 연산자를 사용하여 폼을 만드는 경우에는 해당 폼이 Project Options|Forms 페이지의 Auto-create forms 리스트에 없어야 합니다. 특히 리스트에서 같은 이름의 폼을 삭제하지 않고 새 폼을 만들었는데 C++Builder에서 시작할 때 폼을 만들고 이 이벤트 핸들러가 폼의 새 인스턴스를 만들면 자동으로 생성된 인스턴스에 대한 참조가 덮어써집니다. 자동으로 생성된 인스턴스는 계속 존재하지만 애플리케이션에서 액세스할 수 없게 됩니다. 이벤트 핸들러가 종료되고 나면 전역 변수에서 더 이상 유효한 폼을 가리킬 수 없습니다. 전역 변수를 역참조하려고 하면 애플리케이션에서 오류가 발생합니다.

윈도우 등의 modeless 폼 생성

폼을 사용하는 동안 modeless 폼의 참조 변수가 존재하게 해야 합니다. 즉 이러한 변수가 전역 범위를 가져야 합니다. 대부분의 경우 폼을 만들 때 생성된 전역 참조 변수를 사용합니다. 이 변수의 이름은 해당 폼의 Name 속성과 일치합니다. 애플리케이션에서 폼의 추가 인스턴스가 필요하면 인스턴스에 대한 각각의 전역 변수(폼 클래스에 대한 타입 포인터)를 선언합니다.

로컬 변수를 사용하여 폼 인스턴스 생성

*modal 폼*의 고유 인스턴스를 더 안전한 방법으로 만들려면 이벤트 핸들러에서 로컬 변수를 새 인스턴스에 대한 참조로 사용하십시오. 로컬 변수를 사용하면 *ResultsForm*이 자동으로 생성되었는지의 여부는 상관 없습니다. 이벤트 핸들러의 코드는 전역 폼 변수에 대한 참조를 만들지 않습니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  TResultsForm *rf = new TResultsForm(this);// rf is local form instance
 rf->ShowModal();
 delete rf;    // form safely destroyed
}
```

폼의 전역 인스턴스는 이 버전의 이벤트 핸들러에서 사용되지 않습니다.

일반적으로 애플리케이션은 폼의 전역 인스턴스를 사용합니다. 그러나 modal 폼의 새 인스턴스가 필요하고 하나의 함수와 같은 애플리케이션의 제한된 별도 섹션에서 폼을 사용하는 경우 로컬 인스턴스는 폼을 사용하는 가장 안전하고 효율적인 방법입니다.

물론, modeless 폼을 위한 이벤트 핸들러의 경우 폼을 사용하는 동안 계속 폼이 존재하도록 전역 범위를 가져야 하기 때문에 로컬 변수를 사용할 수 없습니다. *Show*는 폼을 열자마자 반환하므로 로컬 변수를 사용하는 경우에는 로컬 변수가 즉시 범위를 벗어납니다.

폼에 추가 인수 전달

일반적으로 IDE 내에서 애플리케이션을 위한 폼을 만듭니다. 이러한 방법으로 만들 때는 작성 중인 폼의 소유자에 대한 포인터인 하나의 인수 *Owner*를 취하는 생성자가 폼에 있습니다. 소유자는 호출하는 애플리케이션 객체 또는 폼 객체입니다. *Owner*는 NULL이 될 수 있습니다.

폼에 추가 인수를 전달하려면 **new** 연산자를 사용하여 각각의 생성자를 만들고 인스턴스화합니다. 다음 예제 폼 클래스는 추가 인수 *whichButton*이 있는 추가 생성자를 보여 줍니다. 이 새 생성자는 폼 클래스에 수동으로 추가됩니다.

```
class TResultsForm : public TForm
{
__published:    // IDE-managed Components
   TLabel *ResultsLabel;
   TButton *OKButton;
   void __fastcall OKButtonClick(TObject *Sender);
private:        // User declarations
public:         // User declarations
   virtual __fastcall TResultsForm(TComponent* Owner);
  virtual __fastcall TResultsForm(int whichButton, TComponent* Owner);
};
```

다음은 추가 인수인 *whichButton*을 전달하는 수동으로 코딩된 생성자입니다. 이 생성자는 *whichButton* 매개변수를 사용하여 폼에 있는 *Label* 컨트롤의 *Caption* 속성을 설정합니다.

```
void__fastcall TResultsForm::TResultsForm(int whichButton, TComponent*
Owner)
 : TForm(Owner)
{
  switch (whichButton) {
    case 1:
      ResultsLabel->Caption = "You picked the first button!";
      break;
    case 2:
      ResultsLabel->Caption = "You picked the second button!";
      break;
    case 3:
      ResultsLabel->Caption = "You picked the third button!";
  }
}
```

여러 생성자가 있는 폼의 인스턴스를 만들면 용도에 가장 적합한 생성자를 선택할 수 있습니다. 예를 들어, 폼에 있는 버튼에 대한 다음 *OnClick* 핸들러는 추가 매개변수를 사용하는 *TResultsForm* 인스턴스를 만듭니다.

```
void __fastcall TMainMForm::SecondButtonClick(TObject *Sender)
{
  TResultsForm *rf = new TResultsForm(2, this);
  rf->ShowModal();
  delete rf;
}
```

폼에서 데이터 검색

대부분의 실제 애플리케이션은 여러 개의 폼으로 구성됩니다. 이러한 폼 간에 정보를 전달해야 할 때도 있습니다. 정보를 전달할 때는 받는 폼의 생성자로 매개변수를 전달하거나 폼의 속성에 값을 할당합니다. 폼으로부터 정보를 받는 방법은 modal 폼인지 modeless 폼인지에 따라 달라집니다.

modeless 폼에서 데이터 검색

폼의 public 멤버 함수를 호출하거나 폼의 속성을 쿼리하여 modeless 폼에서 정보를 쉽게 추출할 수 있습니다. 예를 들어, 애플리케이션에 있는 modeless 폼 *ColorForm*에 "Red", "Green", "Blue" 등의 색상 리스트가 있는 *ColorListBox* 리스트 박스가 있다고 가정합니다. *ColorListBox*에서 선택한 색 이름 문자열은 사용자가 새 색을 선택할 때마다 자동으로 *CurrentColor*라는 속성에 저장됩니다. 폼의 클래스 선언은 다음과 같습니다.

```
class TColorForm : public TForm
{
  __published:    // IDE-managed Components
    TListBox *ColorListBox;
    void __fastcall ColorListBoxClick(TObject *Sender);
  private:        // User declarations
    String getColor();
    void   setColor(String);
    String curColor;
  public:         // User declarations
    virtual __fastcall TColorForm(TComponent* Owner);
    __property String CurrentColor = {read=getColor, write=setColor};
};
```

리스트 박스 *ColorListBoxClick*의 *OnClick* 이벤트 핸들러는 리스트 박스에서 새 항목을 선택할 때마다 *CurrentColor* 속성 값을 설정합니다. 이벤트 핸들러는 색 이름이 있는 리스트 박스에서 문자열을 가져와서 *CurrentColor*에 할당합니다. *CurrentColor* 속성은 setter 함수 *setColor*를 사용하여 속성의 실제 값을 private 데이터 멤버 *curColor*에 저장합니다.

```
void __fastcall TColorForm::ColorListBoxClick(TObject *Sender)
{
  int index = ColorListBox->ItemIndex;
  if (index >= 0) {// make sure a color is selected
    CurrentColor = ColorListBox->Items->Strings[index];
  }
  else        // no color selected
    CurrentColor = "";
}
//---------------------------------------------------------------------
void TColorForm::setColor(String s)
{
  curColor = s;
}
```

이제 *ResultsForm*에서 *UpdateButton* 버튼을 클릭할 때마다 애플리케이션의 다른 폼인 *ResultsForm*에서 현재 *ColorForm*에서 선택된 색을 알아야 한다고 가정합니다. *UpdateButton*의 *OnClick* 이벤트 핸들러는 다음과 같을 수 있습니다.

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
  if (ColorForm) {// verify ColorForm exists
    String s = ColorForm->CurrentColor;
    // do something with the color name string
  }
}
```

이벤트 핸들러는 먼저 해당 지점이 NULL인지 여부를 점검하여 *ColorForm*이 있는지 확인합니다. 그런 다음 *ColorForm*의 *CurrentColor* 속성 값을 가져옵니다. *CurrentColor* 쿼리는 다음과 같이 해당 getter 함수 *getColor*를 호출합니다.

```
String TColorForm::getColor()
{
  return curColor;
}
```

또는 *ColorForm*의 *getColor* 함수가 public이면 다른 폼에서 *CurrentColor* 속성(예: `String s = ColorForm->getColor();`)을 사용하지 않고 현재 색을 가져올 수 있습니다. 실제로는 리스트 박스에서 직접 선택하여 다른 폼에서 *ColorForm*의 현재 선택된 색상을 자유롭게 가져올 수 있습니다.

```
String s = ColorListBox->Items->Strings[ColorListBox->ItemIndex];
```

그러나 속성을 사용하면 *ColorForm*의 인터페이스가 매우 간단해집니다. 폼에서는 *ColorForm*의 *CurrentColor* 값을 확인하기만 하면 됩니다.

modal 폼에서 데이터 검색

modeless 폼처럼 modal 폼에도 다른 폼에 필요한 정보가 포함됩니다. 가장 일반적인 예는 폼 A가 modal 폼 B를 시작할 때입니다. 폼 B를 닫을 때는 폼 A에서 사용자가 폼 B로 한 작업을 확인하여 폼 A 처리 시 어떻게 함께 처리할지 결정해야 합니다. 폼 B가 여전히 메모리에 있으면 위의 modeless 폼 예제에서처럼 속성이나 멤버 함수를 통해 쿼리할 수 있습니다. 그러나 폼 B를 닫을 때 메모리에서 삭제되는 경우를 생각해 보십시오. 폼은 명시적인 반환 값을 갖지 않으므로 폼을 소멸하기 전에 해당 폼의 중요 정보를 보존해야 합니다.

예를 들어, modal 폼으로 디자인된 *ColorForm* 폼의 수정된 버전을 살펴 봅니다. 클래스 선언은 다음과 같습니다.

```
class TColorForm : public TForm
{
  __published:    // IDE-managed Components
    TListBox *ColorListBox;
    TButton *SelectButton;
    TButton *CancelButton;
    void __fastcall CancelButtonClick(TObject *Sender);
    void __fastcall SelectButtonClick(TObject *Sender);
  private:        // User declarations
    String* curColor;
  public:         // User declarations
    virtual __fastcall TColorForm(TComponent* Owner);
    virtual __fastcall TColorForm(String* s, TComponent* Owner);
};
```

폼에는 색상 리스트가 있는 *ColorListBox* 리스트 박스가 있습니다. *SelectButton* 버튼을 클릭하면 현재 *ColorListBox*에서 선택된 색 이름을 기록한 다음 폼을 닫습니다. *CancelButton*은 그냥 폼을 닫는 버튼입니다.

사용자 정의 생성자는 *String** 인수를 취하는 클래스에 추가됩니다. 이 *String**은 *ColorForm*을 시작한 폼에서 알고 있는 문자열을 가리킬 것입니다. 이 생성자를 구현하려면 다음과 같이 하십시오.

```
void__fastcall TColorForm::TColorForm(String* s, TComponent* Owner)
 : TForm(Owner)
{
  curColor = s;
  *curColor = "";
}
```

생성자는 private 데이터 멤버 *curColor*에 대한 포인터를 저장하고 문자열을 빈 문자열로 초기화합니다.

참고 위의 사용자 정의 생성자를 사용하려면 폼을 명시적으로 만들어야 합니다. 애플리케이션을 시작할 때 자동으로 생성되는 폼에서는 사용할 수 없습니다. 자세한 내용은 8-5페이지의 "메모리에 있는 폼 제어"를 참조하십시오.

애플리케이션에서 사용자는 리스트 박스에서 색을 선택하고 *SelectButton*을 클릭하여 선택 내용을 저장한 다음 폼을 닫습니다. *SelectButton*의 *OnClick* 이벤트 핸들러는 다음과 같습니다.

```
void __fastcall TColorForm::SelectButtonClick(TObject *Sender)
{
  int index = ColorListBox->ItemIndex;
  if (index >= 0)
    *curColor = ColorListBox->Items->Strings[index];
  Close();
}
```

이벤트 핸들러는 선택한 색 이름을 생성자에 전달된 문자열 주소에 저장합니다.

*ColorForm*을 효과적으로 사용하려면 호출한 폼이 생성자에게 기존 문자열에 대한 포인터를 전달해야 합니다. 예를 들어, *ColorForm*이 *ResultsForm*에 있는 *UpdateButton* 버튼에 대한 응답으로 *ResultsForm*이라는 폼에 의해 인스턴스화되었다고 가정합니다. 이벤트 핸들러는 다음과 같습니다.

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
  String s;
  GetColor(&s);
  if (s != "") {
    // do something with the color name string
  }
  else {
    // do something else because no color was picked
  }
}
//---------------------------------------------------------------------------
void TResultsForm::GetColor(String *s)
{
  ColorForm = new TColorForm(s, this);
  ColorForm->ShowModal();
  delete ColorForm;
  ColorForm = 0; // NULL the pointer
}
```

*UpdateButtonClick*은 *s*라는 문자열을 만듭니다. *s*의 주소는 *ColorForm*을 만드는 *GetColor*로 전달되어 *s*에 대한 포인터가 생성자에 대한 인수로 전달됩니다. *ColorForm*은 닫히자마자 삭제되지만 색 이름을 선택한 경우 선택한 색 이름은 아직 *s*에 남아 있습니다. 그렇지 않으면 *s*에 사용자가 색을 선택하지 않고 *ColorForm*을 끝냈다는 것을 나타내는 빈 문자열이 있습니다.

이 예제에서는 한 문자열 변수를 사용하여 modal 폼의 정보를 저장합니다. 물론 필요에 따라 더 복잡한 객체를 사용할 수도 있습니다. 빈 문자열에 *s* 기본값을 설정하는 등, 항목이 변경되거나 선택되지 않고 modal 폼이 닫혔는지 여부를 호출하는 폼이 항상 알 수 있게 해야 합니다.

컴포넌트와 컴포넌트 그룹의 재사용

C++Builder에서는 컴포넌트로 수행했던 작업을 저장하고 재사용하는 여러 가지 방법을 제공합니다.

- *컴포넌트 템플릿*은 컴포넌트 그룹을 구성하고 저장하기 위한 간단하고 쉬운 방법을 제공합니다. 8-12페이지의 "컴포넌트 템플릿 생성 및 사용"을 참조하십시오.
- *리포지토리*에 폼, 데이터 모듈 및 프로젝트를 저장할 수 있습니다. 리포지토리는 재사용 가능한 요소들이 모여 있는 데이터베이스를 제공하고 폼을 상속하여 변경 내용을 전달할 수 있습니다.
- 컴포넌트 팔레트나 Object Repository에 *프레임*을 저장할 수 있습니다. 프레임은 폼 상속을 사용하며 폼이나 다른 프레임에 포함될 수 있습니다. 8-13페이지의 "프레임 사용"을 참조하십시오.

- *사용자 정의 컴포넌트*를 만들어 코드를 재사용하는 방법은 가장 복잡하기는 하지만 가장 유연성이 큽니다. 45장, "컴포넌트 생성 개요"를 참조하십시오.

컴포넌트 템플릿 생성 및 사용

하나 이상의 컴포넌트로 구성된 템플릿을 만들 수 있습니다. 컴포넌트를 폼에 정렬한 후 속성을 설정하고 코드를 작성한 다음 *컴포넌트 템플릿*으로 저장합니다. 나중에 컴포넌트 팔레트에서 템플릿을 선택하여 미리 구성된 컴포넌트를 폼 위에 두면, 연결된 모든 속성과 이벤트 처리 코드가 프로젝트에 동시에 추가됩니다.

일단 템플릿을 폼에 두면 컴포넌트를 각각의 작업으로 배치한 것처럼 각 컴포넌트의 위치를 바꾸고, 속성을 다시 설정하고, 컴포넌트의 이벤트 핸들러를 만들거나 수정할 수 있습니다.

다음과 같은 방법으로 컴포넌트 팔레트를 만듭니다.

1. 폼에 컴포넌트를 두고 정렬합니다. Object Inspector에서 컴포넌트의 속성 및 이벤트를 설정합니다.
2. 컴포넌트를 선택합니다. 여러 컴포넌트를 선택하는 가장 쉬운 방법은 모든 컴포넌트 위로 마우스를 끄는 것입니다. 회색 핸들이 선택된 각 컴포넌트의 모서리에 나타납니다.
3. Component|Create Component Template를 선택합니다.
4. Component Template Information 에디트 박스에서 컴포넌트 템플릿의 이름을 지정합니다. 디폴트로, 단계 2에서 선택된 첫 번째 컴포넌트의 컴포넌트 타입 뒤에 "Template"이라는 단어가 추가됩니다. 예를 들어, 레이블을 선택한 다음 에디트 박스를 선택하면 "TLabelTemplate"이라는 이름이 제시됩니다. 이 이름을 변경할 수는 있지만 기존의 컴포넌트 이름과 중복되지 않도록 주의합니다.
5. Palette Page 에디트 박스에서 템플릿을 넣으려는 컴포넌트 팔레트 페이지를 지정합니다. 존재하지 않는 페이지를 지정하면 템플릿을 저장할 때 새 페이지가 만들어집니다.
6. Palette Icon 옆에서 팔레트의 템플릿을 나타내는 비트맵을 선택합니다. 디폴트로, 단계 2에서 선택된 첫 번째 컴포넌트의 컴포넌트 타입에서 사용하는 비트맵이 사용됩니다. 다른 비트맵을 찾으려면 Change를 클릭합니다. 선택한 비트맵은 24x24픽셀 이하여야 합니다.
7. OK를 클릭합니다.

컴포넌트 팔레트에서 템플릿을 제거하려면 Component|Configure Palette를 선택합니다.

프레임 사용

폼과 같은 프레임(*TFrame*)은 다른 컴포넌트의 컨테이너입니다. 프레임은 자동 인스턴스화 및 컴포넌트 삭제에 대해 폼과 같은 소유권 메커니즘을 사용하며 컴포넌트 속성의 동기화에 대해 같은 부모-자식 관계를 사용합니다

그러나 어떤 면에서 볼 때 프레임은 폼에 비해 사용자 정의하기가 더 수월한 컴포넌트입니다. 프레임은 쉽게 다시 사용하기 위해 컴포넌트 팔레트에 저장할 수 있으며 폼, 다른 프레임 또는 다른 컨테이너 객체 내에 중첩시킬 수 있습니다. 프레임을 만들고 저장하고 나면 프레임이 계속 하나의 유닛으로 기능을 수행하고 이 프레임을 포함하는 컴포넌트(다른 프레임 포함)로부터 변경 내용을 상속합니다. 프레임이 다른 프레임이나 폼에 포함되면 프레임은 이전 프레임의 변경 내용이 계속 상속됩니다.

프레임은 애플리케이션에서 여러 곳에서 사용되는 컨트롤 그룹을 구성할 때 유용합니다. 예를 들면, 여러 폼에서 사용되는 비트맵을 프레임에 두어 애플리케이션의 리소스에 해당 비트맵 복사본 한 개만 포함되게 할 수 있습니다. 프레임으로 테이블을 편집하고 테이블에 데이터를 입력하고자 할 때마다 사용할 수 있는 일련의 편집 필드를 설명할 수도 있습니다.

프레임 생성

비어 있는 프레임을 만들려면 File|New|Frame을 선택하거나 File|New|Other를 선택하고 Frame을 더블 클릭하십시오. 그런 다음 새 프레임으로 다른 프레임을 포함한 컴포넌트를 놓을 수 있습니다.

반드시 그럴 필요는 없지만 프레임을 프로젝트의 일부로 저장하는 것이 가장 좋은 방법입니다. 폼 없이 프레임만 포함하는 프로젝트를 만들려면 File|New| Application을 선택하고 새 폼과 유닛을 저장하지 않고 닫은 다음 File|New|Frame을 선택하고 프로젝트를 저장합니다.

참고 프레임을 저장할 때 *Unit1*, *Project1* 등 디폴트 이름을 사용하지 않아야 합니다. 이러한 디폴트 이름을 사용하면 나중에 프레임을 사용하려고 할 때 충돌을 일으킬 수 있기 때문입니다.

디자인 타임에 View|Forms를 선택하고 프레임을 선택하여 현재 프로젝트에 포함된 모든 프레임을 표시할 수 있습니다. 폼과 데이터 모듈처럼 View as Form 또는 View as Text를 마우스 오른쪽 버튼으로 클릭하여 폼 디자이너와 프레임의 폼 파일을 토글할 수 있습니다.

컴포넌트 팔레트에 프레임 추가

프레임은 컴포넌트 템플릿으로 컴포넌트 팔레트에 추가됩니다. 컴포넌트 팔레트에 프레임을 추가하려면 폼 디자이너에서 프레임을 열고 프레임을 마우스 오른쪽 버튼으로 클릭한 다음, Add to Palette를 선택하십시오. 이 경우에는 다른 컴포넌트에 포함된 프레임을 사용할 수 없습니다. Component Template Information 다이얼로그 박스가 열리면 새 템플릿의 이름, 팔레트 페이지 및 아이콘을 선택합니다.

프레임 사용 및 수정

애플리케이션에서 프레임을 사용하려면 직접적이거나 간접적으로 폼에 프레임을 두어야 합니다. 프레임을 폼, 다른 프레임 또는 패널이나 스크롤 박스와 같은 다른 컨테이너 객체에 직접 추가할 수 있습니다.

폼 디자이너에서 애플리케이션에 프레임을 추가하는 방법에는 두 가지가 있습니다.

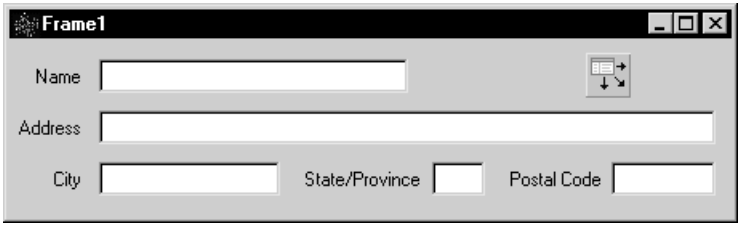
- 컴포넌트 팔레트에서 프레임을 선택하여 폼, 다른 프레임 또는 다른 컨테이너 객체에 가져다 놓습니다. 필요한 경우에는 폼 디자이너에서 프레임의 유닛 파일을 프로젝트에 포함할지 여부를 묻습니다.
- 컴포넌트 팔레트의 Standard 페이지에서 *Frames*를 선택하고 폼이나 다른 프레임을 클릭합니다. 이미 프로젝트에 포함된 프레임 리스트가 있는 다이얼로그 박스가 나타납니다. 하나를 선택한 다음 OK를 클릭합니다.

폼이나 다른 컨테이너에 프레임을 가져다 놓으면 C++Builder에서 선택한 프레임의 자손인 새 클래스를 선언합니다. 마찬가지로, 프로젝트에 새 폼을 추가할 때는 C++Builder에서 *TForm*의 자손인 새 클래스를 선언합니다. 즉, 나중에 원본(조상) 프레임을 변경하면 포함된 프레임에 전달되지만 포함된 프레임을 변경하면 조상으로 다시 전달되지는 않습니다.

예를 들어, data-access 컴포넌트와 데이터 인식 컨트롤 그룹을 여러 애플리케이션에서 반복적으로 사용하기 위해 조합하려는 경우를 가정합니다. 이렇게 하려면 컴포넌트를 하나의 컴포넌트 템플릿으로 모읍니다. 그러나 템플릿을 사용하여 시작한 다음 나중에 컨트롤 배열을 변경하려면 다시 돌아 가서 템플릿을 둔 각 프로젝트를 수동으로 변경해야 합니다.

한편, 데이터베이스 컴포넌트를 프레임에 두면 나중에 한 곳에서만 변경해도 프로젝트를 다시 컴파일할 때 원본 프레임의 변경 내용이 자동으로 포함된 자손으로 전달됩니다. 또한 원본 프레임이나 포함된 다른 자손에 영향을 주지 않고 포함된 프레임을 얼마든지 수정할 수 있습니다. 단, 포함된 프레임을 수정할 때는 컴포넌트를 추가할 수 없습니다.

그림 8.1 데이터 인식 컨트롤과 데이터 소스 컴포넌트가 있는 프레임



프레임을 사용하면 관리가 간편해질 뿐 아니라 리소스를 더 효과적으로 사용할 수 있습니다. 예를 들어, 애플리케이션에서 비트맵이나 다른 그래픽을 사용하려면 그래픽을 *TImage* 컨트롤의 *Picture* 속성으로 로드할 수 있습니다. 그러나 한 애플리케이션에서 같은 그래픽을 반복적으로 사용하는 경우에는 폼에 각 *Image* 객체를 두고 폼의 리소스 파일에 그래픽의 다른 복사본을 추가할 수 있습니다. 이는 *TImage::Picture*를 한 번 설정하고 *Image* 컨트롤을 컴포넌트 템플릿으로 저장하는 경우에도 적용됩니다. 더 나은 방법으로는 *Image* 객체를 프레임에 가져다 놓고 그래픽을 로드한 다음 그래픽이 나타날 위치에 프레임을 사용할 수 있습니다. 이렇게 하면 폼 파일이 더 작아지고 원본 프레임의 *Image*를 수정하여 간단하게 모든 위치에 나타나는 그래픽을 변경할 수 있다는 장점이 있습니다.

프레임 공유

두 가지 방법으로 다른 개발자와 프레임을 공유할 수 있습니다.

- Object Repository에 프레임을 추가합니다.
- 프레임 유닛(.cpp and .h)과 폼(.dfm 또는 .xfm) 파일을 배포합니다.

리포지토리에 프레임을 추가하려면 프레임에 포함된 프로젝트를 열고 폼 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Add to Repository를 선택하십시오. 자세한 내용은 7-24페이지의 "Object Repository 사용"을 참조하십시오.

프레임 유닛과 폼 파일을 다른 개발자에게 보내는 경우 이 유닛과 파일을 열고 컴포넌트 팔레트에 추가할 수 있습니다. 프레임에 다른 프레임이 포함되어 있으면 이 프레임을 프로젝트의 한 부분으로 열어야 합니다.

다이얼로그 박스 개발

컴포넌트 팔레트의 Dialog 페이지에 있는 다이얼로그 박스 컴포넌트를 사용하면 다양한 다이얼로그 박스를 애플리케이션에서 사용할 수 있습니다. 이러한 다이얼로그 박스는 사용자가 파일 열기, 저장, 인쇄와 같이 공통적인 파일 작업을 수행할 수 있는 친숙하고 일관적인 인터페이스를 애플리케이션에 제공합니다. 다이얼로그 박스는 데이터를 표시하거나 얻습니다.

Execute 메소드가 호출되면 각 다이얼로그 박스가 열립니다. *Execute*는 부울 값을 반환합니다. 사용자가 OK를 선택하여 다이얼로그 박스의 변경 사항을 적용하면 *Execute*가 **true**를 반환하고, Cancel을 선택하여 변경 사항을 저장하지 않고 다이얼로그 박스를 빠져 나가면 *Execute*가 **false**를 반환합니다.

CLX 크로스 플랫폼 애플리케이션을 개발할 때는 QDialogs 유닛의 CLX에서 제공하는 다이얼로그 박스를 사용할 수 있습니다. 운영 체제에 파일 열기 또는 저장, 글꼴이나 색 변경과 같은 공통적인 작업을 위한 원시 다이얼로그 박스 타입이 있는 경우 *UseNativeDialog* 속성을 사용할 수 있습니다. 애플리케이션이 이러한 환경에서 실행되고 운영 체제에서 Qt 다이얼로그 박스 대신 원시 다이얼로그 박스를 사용하게 하려면 *UseNativeDialog*를 **true**로 설정합니다.

Open 다이얼로그 박스 사용

공통으로 사용되는 다이얼로그 박스 컴포넌트 중의 하나는 *TOpenDialog* 입니다. 이 컴포넌트는 보통 폼의 메인 메뉴 바에 있는 File 옵션 아래의 New 또는 Open 메뉴 항목에 의해 호출됩니다. 다이얼로그 박스에는 와일드카드 문자를 사용하여 파일 그룹을 선택하고 디렉토리를 통해 탐색할 수 있는 컨트롤이 포함됩니다.

TOpenDialog 컴포넌트를 사용하면 애플리케이션에서 Open 다이얼로그 박스를 사용할 수 있습니다. 이 다이얼로그 박스의 용도는 사용자에게 열려는 파일을 선택하도록 하는 것입니다. *Execute* 메소드를 사용하여 다이얼로그 박스를 표시합니다.

사용자가 다이얼로그 박스에서 OK를 선택하면 사용자의 파일은 *TopenDialog*의 *FileName* 속성에 저장된 후에 원하는대로 처리할 수 있습니다.

다음 코드를 *Action*에 두고 *TMainMenu* 하위 항목의 *Action* 속성에 연결하거나 하위 항목의 *OnClick* 이벤트에 둘 수 있습니다.

```
if(OpenDialog1->Execute()){
   filename = OpenDialog1->FileName;
};
```

이 코드는 다이얼로그 박스를 보여주고, 사용자가 OK 버튼을 클릭할 경우 이전에 선언한 `filename`이라는 *AnsiString* 변수에 파일 이름을 복사합니다.

툴바와 메뉴에 대한 액션 구성

C++Builder는 메뉴와 툴바를 만들고, 사용자 정의하고 관리하는 작업을 간단히 수행할 수 있는 여러 가지 기능을 제공합니다. 이러한 기능을 사용하면 애플리케이션 사용자가 툴바에서 버튼을 클릭하거나, 메뉴에서 명령을 선택하거나, 아이콘을 가리키거나 클릭하여 시작할 수 있는 액션 리스트를 구성할 수 있습니다.

여러 사용자 인터페이스 요소에서 한 액션 집합이 사용될 때도 있습니다. 예를 들면, Cut, Copy 및 Paste 명령이 Edit 메뉴와 툴바에 모두 나타날 때가 있습니다. 액션을 한 번만 추가하면 애플리케이션의 여러 UI 요소에서 사용할 수 있습니다.

Windows 플랫폼에서는 디자인 타임이나 런타임에 쉽게 액션을 정의 및 그룹화하고, 다양한 레이아웃을 만들고, 메뉴를 사용자 정의할 수 있도록 툴을 제공합니다. 이러한 툴을 통칭해서 ActionBand 툴이라고 하며 툴을 사용하여 만든 메뉴와 툴바를 액션 밴드라고 합니다. 일반적으로 다음과 같이 ActionBand 사용자 인터페이스를 만들 수 있습니다.

- 액션 리스트를 작성하여 애플리케이션에서 사용할 수 있는 일련의 액션을 만듭니다 (Action Manager, *TActionManager* 사용).
- 사용자 인터페이스 요소를 애플리케이션에 추가합니다(*TActionMainMenuBar* 및 *TActionToolBar*와 같은 ActionBand 사용).
- Action Manager에서 액션을 사용자 인터페이스 요소로 드래그 앤 드롭합니다.

다음 표는 메뉴 및 툴바 설정과 관련된 용어를 정의한 것입니다.

표 8.1 액션 설정 용어

용어	정의
액션	메뉴 항목을 클릭하는 등 사용자가 수행한 작업에 대한 응답입니다. 애플리케이션에서 사용할 수 있도록 자주 필요한 많은 표준 액션이 제공됩니다. 예를 들어, File Open, File SaveAs, File Run 및 File Exit와 같은 파일 작업이 편집, 서식 설정, 검색, 도움말, 다이얼로그 박스 및 윈도우 액션을 위한 다른 작업들과 함께 포함됩니다. 액션 리스트와 Action Manager로 사용자 정의 액션을 프로그래밍하고 액세스할 수 있습니다.
액션 밴드	사용자 정의 가능한 메뉴나 툴바와 관련된 일련의 액션을 위한 컨테이너입니다. 액션 밴드의 예로는 메인 메뉴와 툴바에 대한 ActionBand 컴포넌트(*TActionMainMenuBar* 및 *TActionToolBar*)가 있습니다.
액션 범주	액션을 그룹화하고 그룹화된 액션을 메뉴나 툴바에 가져다 놓을 수 있습니다. 예를 들어, 표준 액션 범주인 Search에는 Find, FindFirst, FindNext 및 Replace 액션에 모두 포함됩니다.
액션 클래스	애플리케이션에서 사용되는 액션을 수행하는 클래스입니다. 모든 표준 액션은 *TEditCopy*, *TEditCut* 및 *TEditUndo*와 같은 액션 클래스에서 정의됩니다. Customize 다이얼로그 박스에서 액션 밴드로 이러한 클래스를 드래그 앤 드롭하여 사용할 수 있습니다.
액션 클라이언트	일반적으로 액션을 시작하는 통지를 받는 메뉴 항목이나 버튼을 나타냅니다. 클라이언트에서 마우스 클릭과 같은 사용자 명령을 받으면 관련된 액션을 시작합니다.
액션 리스트	사용자 작업에 대한 응답으로 애플리케이션에서 수행할 수 있는 액션의 리스트입니다.
Action Manager	ActionBand 컴포넌트에서 다시 사용할 수 있는 로컬 액션 집합을 그룹화하고 구성합니다. *TActionManager*를 참조하십시오.
메뉴	애플리케이션 사용자가 클릭하여 실행할 수 있는 명령을 나열합니다. ActionBand 메뉴 클래스 *TActionMainMenuBar*를 사용하거나 *TMainMenu* 또는 *TPopupMenu* 등의 크로스 플랫폼 컴포넌트를 사용하여 메뉴를 만들 수 있습니다.
대상	액션이 수행되는 항목을 나타냅니다. 대상은 보통 메모나 데이터 컨트롤과 같은 컨트롤이며 모든 액션에 대상이 필요하지는 않습니다. 예를 들어, 표준 도움말 액션은 대상을 무시하며 간단히 도움말 시스템을 시작합니다.
툴바	클릭할 때 프로그램이 현재 문서 인쇄 등 특정 액션을 수행하게 하는 버튼 아이콘의 시각적 행을 표시합니다. ActionBand 툴바 컴포넌트 *TActionToolBar*를 사용하거나 크로스 플랫폼 컴포넌트 *TToolBar*를 사용하여 툴바를 만들 수 있습니다.

크로스 플랫폼을 개발할 때는 8-23페이지의 "액션 리스트 사용"을 참조하십시오.

액션의 개념

애플리케이션을 개발할 때 다양한 UI 요소에서 사용할 수 있는 액션 집합을 만들 수 있습니다. 액션을 하나의 집합으로(예: Cut, Copy 및 Paste) 또는 한 번에 하나씩(예: Tools|Customize) 메뉴에 가져다 놓을 수 있는 범주로 구성할 수 있습니다.

하나의 액션은 메뉴 항목이나 툴바 버튼 등 사용자 인터페이스에 있는 하나 이상의 요소에 해당됩니다. 액션은 두 가지 기능을 제공합니다. (1) 컨트롤이 활성화되었는지 또는 선택되었는지 여부와 같이 사용자 인터페이스 요소에 공통적인 속성을 나타내고, (2) 예를 들어 애플리케이션 사용자가 버튼을 클릭하거나 메뉴 항목을 선택할 때와 같이 컨트롤이 실행될 때 응답합니다. 메뉴, 버튼, 툴바, 컨텍스트 메뉴 등을 통해 애플리케이션에서 사용할 수 있는 액션 리스트를 만들 수 있습니다.

액션은 다른 컴포넌트와 연결됩니다.

- **클라이언트:** 하나 이상의 클라이언트에서 액션을 사용합니다. 클라이언트는 보통 메뉴 항목이나 버튼(예: *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton* 등)을 나타냅니다. 액션은 *TActionMainMenuBar* 및 *TActionToolBar*와 같은 *ActionBand*에 상주할 수도 있습니다. 클라이언트에서 마우스 클릭 등의 사용자 명령을 받으면 관련된 액션이 시작됩니다. 일반적으로 클라이언트의 *OnClick* 이벤트는 해당 액션의 *OnExecute* 이벤트와 연결됩니다.
- **대상:** 액션은 대상에서 작동됩니다. 대상은 보통 메모나 데이터 컨트롤과 같은 컨트롤입니다. 컴포넌트 작성자는 디자인하고 사용하는 컨트롤의 필요에 맞는 액션을 만든 다음 이 유닛을 패키지화하여 모듈식 애플리케이션을 만들 수 있습니다. 모든 액션이 대상을 사용하는 것은 아닙니다. 예를 들어, 표준 도움말 액션은 대상을 무시하며 간단히 도움말 시스템을 시작합니다.

  대상도 컴포넌트가 될 수 있습니다. 예를 들어, 데이터 컨트롤은 대상을 관련 데이터셋으로 변경합니다.

액션은 클라이언트에 영향을 받습니다. 액션은 클라이언트가 액션을 실행할 때 응답합니다. 또한 액션도 클라이언트에 영향을 줍니다. 액션 속성에 따라 클라이언트 속성이 동적으로 업데이트됩니다. 예를 들어, 런타임에 액션이 해당 *Enabled* 속성이 **false**로 설정되어 비활성화되면 해당 액션의 모든 클라이언트가 비활성화되어 회색으로 나타납니다.

Action Manager 또는 Action List Editor(액션 리스트 객체 *TActionList*를 더블 클릭하여 표시)를 사용하여 액션을 추가하거나 삭제하거나 다시 정렬할 수 있습니다. 이러한 액션은 나중에 클라이언트 컨트롤에 연결됩니다.

액션 밴드 설정

액션에는 모양이나 위치 등의 "레이아웃" 정보가 유지되지 않으므로 C++Builder는 이 데이터를 저장할 수 있는 액션 밴드를 제공합니다. 액션 밴드는 레이아웃 정보와 일련의 컨트롤을 지정할 수 있는 메커니즘을 제공합니다. 액션을 툴바와 메뉴와 같은 UI 요소로 렌더링할 수 있습니다.

Action Manager(*TActionManager*)를 사용하여 액션 집합을 구성합니다. 제공된 표준 액션을 사용하거나 직접 새 액션을 만들 수 있습니다.

그런 다음 액션 밴드를 만듭니다.

- *TActionMainMenuBar*를 사용하여 메인 메뉴를 만듭니다.
- *TActionToolBar*를 사용하여 툴바를 만듭니다.

액션 밴드는 액션 집합을 저장하고 렌더링하는 컨테이너 역할을 합니다. 디자인 타임에 Action Manager Editor에서 액션 밴드로 항목을 드래그 앤 드롭할 수 있습니다. 런타임에 애플리케이션 사용자는 Action Manager Editor와 비슷한 다이얼로그 박스를 사용하여 애플리케이션의 메뉴나 툴바를 사용자 정의할 수도 있습니다.

툴바 및 메뉴 생성

참고 이 단원에서는 Windows 애플리케이션에서 메뉴와 툴바를 만들 때 권장할 만한 방법을 소개합니다. 크로스 플랫폼 개발의 경우 *TToolBar*와 *TMainMenu*와 같은 메뉴 컴포넌트를 사용하고 액션 리스트(*TActionList*)를 사용하여 이들을 구성해야 합니다. 8-23페이지의 "액션 리스트 설정"을 참조하십시오.

Action Manager를 사용하여 애플리케이션에 포함된 액션을 기준으로 툴바와 메인 메뉴를 자동으로 생성합니다. Action Manager는 표준 액션과 직접 작성한 모든 사용자 정의 액션을 관리합니다. 그런 다음 이러한 액션을 기준으로 UI 요소를 작성하고 액션 밴드를 사용하여 액션 항목을 메뉴 항목이나 툴바의 버튼으로 렌더링합니다.

메뉴, 툴바 및 기타 액션 밴드를 만드는 일반적인 절차는 다음과 같습니다.

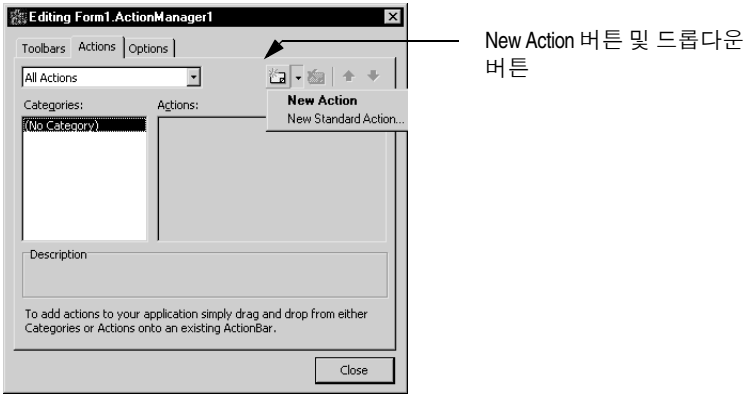
- Action Manager를 폼에 가져다 놓습니다.
- Action Manager에 액션을 추가하여 적절한 액션 리스트로 구성합니다.
- 사용자 인터페이스를 위한 액션 밴드(메뉴 또는 툴바)를 만듭니다.
- 액션을 애플리케이션 인터페이스로 드래그 앤 드롭합니다.

다음 절차는 이 단계를 더 자세히 설명한 것입니다.

다음과 같은 방법으로 액션 밴드를 사용하여 메뉴와 툴바를 만듭니다.

1. 컴포넌트 팔레트의 Additional 페이지에서 Action Manager 컴포넌트(*TActionManager*)를 툴바나 메뉴를 만들 폼에 가져다 놓습니다.
2. 메뉴나 툴바에 이미지가 나타나게 하려면 컴포넌트 팔레트의 Win32 페이지의 ImageList 컴포넌트를 폼에 가져다 놓습니다. ImageList에 사용할 이미지를 추가하거나 제공된 이미지를 사용해야 합니다.
3. 컴포넌트 팔레트의 Additional 페이지에서 다음 액션 밴드 중 하나 이상을 폼에 가져다 놓습니다.
   - *TActionMainMenuBar*(메인 메뉴 디자인용)
   - *TActionToolBar*(툴바 디자인용)
4. ImageList를 Action Manager에 연결합니다. Action Manager와 Object Inspector에 포커스를 두고 Images 속성에서 ImageList 이름을 선택합니다.
5. Action Manager Editor의 액션 창에 액션을 추가합니다.
   - Action Manager를 더블 클릭하여 Action Manager Editor를 표시합니다.
   - New Action 버튼(그림 8.2처럼 Actions 탭의 오른쪽 상단 모서리의 왼쪽 끝에 있는 버튼) 옆의 드롭다운 화살표를 클릭하고 New Action 또는 New Standard Action을 선택합니다. 트리 뷰가 나타납니다. 하나 이상의 액션이나 액션 범주를 Action Manager의 액션 창에 추가합니다. Action Manager는 액션을 액션 리스트에 추가합니다.

그림 8.2 Action Manager Editor



6 Action Manager Editor에서 한 액션이나 액션 범주를 디자인하고 있는 메뉴나 툴바로 드래그 앤 드롭합니다.

사용자 정의 액션을 추가하려면 New Action 버튼을 누르고 실행할 때 응답하는 방법을 정의한 이벤트 핸들러를 작성하여 새 *TAction*을 만드십시오. 자세한 내용은 8-24페이지의 "액션 실행 시 발생하는 이벤트"를 참조하십시오. 한 번 정의된 액션은 표준 액션처럼 메뉴나 툴바에 드래그 앤 드롭할 수 있습니다.

메뉴, 버튼 및 툴바에 색, 패턴 또는 그림 추가

Background 및 *BackgroundLayout* 속성을 사용하여 메뉴 항목이나 버튼에 사용할 색, 패턴 또는 비트맵을 지정할 수 있습니다. 이러한 속성을 사용하여 메뉴의 왼쪽이나 오른쪽에서 실행되는 배너를 설정할 수도 있습니다.

액션 클라이언트 객체에서 배경과 레이아웃을 하위 항목에 할당합니다. 메뉴에 있는 항목의 배경을 설정하려면 폼 디자이너에서 항목이 포함된 메뉴 항목을 클릭합니다. 예를 들어, File을 선택하면 File 메뉴에 나타나는 항목의 배경을 변경할 수 있습니다. Object Inspector의 *Background* 속성에서 색, 패턴 또는 비트맵을 할당할 수 있습니다.

BackgroundLayout 속성을 사용하여 요소에 배경을 설정하는 방법을 설명합니다. 색과 이미지는 일반적으로 캡션 뒤에 두거나, 항목 영역에 맞게 확장하거나, 작은 사각형 형태로 바둑판식 배열하여 영역을 가릴 수 있습니다.

보통(blNormal), 확장(blStretch) 또는 바둑판식(blTile) 배경은 투명한 배경과 함께 렌더링됩니다. 배너를 만들면 전체 이미지가 항목의 왼쪽(blLeftBanner)이나 오른쪽(blRightBanner)에 배치됩니다. 배너는 확장하거나 축소할 수 없으므로 정확한 크기인지 확인해야 합니다.

액션 밴드(메인 메뉴 또는 툴바의)의 배경을 변경하려면 액션 밴드를 선택하고 액션 밴드 컬렉션 에디터에서 *TActionClientBar*를 선택하십시오. *Background* 및 *BackgroundLayout* 속성을 설정하여 전체 툴바나 메뉴에서 사용할 색, 패턴 또는 비트맵을 지정할 수 있습니다.

메뉴와 툴바에 아이콘 추가

메뉴 항목 옆에 아이콘을 추가하거나 툴바의 캡션을 아이콘으로 대신할 수 있습니다. *ImageList* 컴포넌트를 사용하여 비트맵이나 아이콘을 구성합니다.

1 컴포넌트 팔레트의 Win32 페이지에서 *ImageList* 컴포넌트를 폼에 가져다 놓습니다.

2 사용할 이미지를 이미지 리스트에 추가합니다. *ImageList* 아이콘을 더블 클릭합니다. Add를 클릭하고 사용할 이미지를 탐색한 다음 OK를 클릭합니다. Program Files\Common Files\Borland Shared\Images에 일부 예제 이미지가 포함되어 있습니다. 버튼 이미지는 활성화된 버튼과 비활성화된 버튼 두 가지 보기로 포함되어 있습니다.

3 컴포넌트 팔레트의 Additional 페이지에서 다음 액션 밴드 중 하나 이상을 폼에 가져다 놓습니다.

- *TActionMainMenuBar*(메인 메뉴 디자인용)
- *TActionToolBar*(툴바 디자인용)

4 이미지 리스트를 Action Manager에 연결합니다. 먼저 Action Manager에 포커스를 설정합니다. 그런 다음 Object Inspector의 *Images* 속성에서 `ImageList1`과 같은 이미지 리스트 이름을 선택합니다.

5 Action Manager Editor를 사용하여 액션을 Action Manager에 추가합니다. 해당 *ImageIndex* 속성을 이미지 리스트의 번호로 설정하여 이미지와 액션을 연결할 수 있습니다.

6 Action Manager Editor에서 한 액션이나 액션 범주를 메뉴나 툴바로 드래그 앤 드롭합니다.

7 툴바에 캡션 없이 아이콘만 표시하려면 Toolbar 액션 밴드를 선택하고 해당 Items 속성을 더블 클릭합니다. 컬렉션 에디터에서 하나 이상의 항목을 선택하거나 해당 *Caption* 속성을 설정할 수 있습니다.

8 이미지가 자동으로 메뉴나 툴바에 나타납니다.

사용자가 정의할 수 있는 툴바 및 메뉴 생성

액션 밴드와 Action Manager를 사용하여 정의할 수 있는 툴바와 메뉴를 만들 수 있습니다. 런타임에 애플리케이션 사용자는 Action Manager Editor 와 비슷한 사용자 정의 가능한 다이얼로그 박스를 사용하여 애플리케이션 사용자 인터페이스의 툴바와 메뉴(액션 밴드)를 사용자 정의할 수 있습니다.

애플리케이션 사용자는 다음과 같은 방법으로 애플리케이션에서 액션 밴드를 사용자 정의할 수 있습니다.

1 Action Manager 컴포넌트를 폼에 가져다 놓습니다.

2 액션 밴드 컴포넌트(*TActionMainMenuBar*, *TActionToolBar*)를 가져다 놓습니다.

3 Action Manager를 더블 클릭하여 Action Manager Editor를 표시합니다.

- 애플리케이션에서 사용할 액션을 추가합니다. 표준 액션 리스트 아래쪽에 나타날 Customize 액션도 추가합니다.
- Additional 탭의 *TCustomizeDlg* 컴포넌트를 폼에 가져다 놓고 해당 ActionManager 속성을 사용하여 Action Manager에 연결합니다. 사용자가 지정한 내용을 스트리밍할 파일 이름을 지정합니다.
- 액션을 액션 밴드 컴포넌트로 드래그 앤 드롭합니다. Customize 액션을 툴바나 메뉴에 추가해야 합니다.

4 애플리케이션을 완료합니다.

애플리케이션을 컴파일하고 실행하면 사용자가 Action Manager Editor와 비슷한 사용자 정의 다이얼로그 박스를 표시하는 Customize 명령에 액세스할 수 있습니다. Action Manager에서 제공한 액션을 사용하여 툴바를 만들거나 메뉴 항목을 드래그 앤 드롭할 수 있습니다.

액션 밴드에서 사용하지 않는 항목 및 범주 숨기기

ActionBand를 사용하면 사용하지 않는 항목과 범주를 사용자로부터 숨길 수 있다는 장점이 있습니다. 나중에 애플리케이션 사용자가 액션 밴드를 사용자 정의하여 사용하는 항목만 표시하고 나머지는 보기에서 숨길 수 있습니다. 숨겨진 항목은 사용자가 드롭다운 버튼을 눌러 다시 표시할 수 있습니다. 또한 사용자는 사용자 정의 다이얼로그 박스에서 사용량 통계를 다시 설정하여 모든 액션 밴드 항목의 가시성을 복원할 수 있습니다. 항목 숨기기는 액션 밴드의 디폴트 동작이지만 각 항목, File 메뉴와 같은 특정 컬렉션의 모든 항목 또는 특정 액션 밴드의 모든 항목을 숨길 수 없도록 변경할 수 있습니다.

Action Manager는 사용자가 액션을 호출한 횟수를 추적합니다. 이 횟수는 해당 *TActionClientItem*의 *UsageCount* 필드에 저장됩니다. Action Manager는 애플리케이션이 실행된 횟수인 세션 번호는 물론 액션이 마지막으로 사용된 세션 번호도 기록합니다. *UsageCount* 값은 항목이 숨겨지기 전에 사용하지 않을 수 있는 최대 세션 수를 알아내어 이 값을 현재 세션 번호와 항목을 마지막으로 사용한 세션 번호 간의 차이와 비교합니다. 차이가 *PrioritySchedule*에서 지정한 수보다 크면 항목이 숨겨집니다. 다음 표에는 *PrioritySchedule*의 기본값이 나와 있습니다.

표 8.2 Action Manager의 PrioritySchedule 속성의 기본값

액션 밴드 항목이 사용된 세션 수	마지막으로 사용한 후 항목이 숨겨지지 않는 세션 수
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29
25 이상	31

디자인 타임에 항목 숨기기를 비활성화할 수 있습니다. 특정 액션과 해당 액션을 포함하는 모든 컬렉션이 숨겨지지 않게 하려면 해당 *TActionClientItem* 객체를 찾아 *UsageCount*를 -1로 설정하십시오. File 메뉴나 메인 메뉴 바와 같은 전체 항목 컬렉션이 숨겨지지 않게 하려면 컬렉션과 연결된 *TActionClients* 객체를 찾아 해당 *HideUnused* 속성을 **false**로 설정하십시오.

액션 리스트 사용

참고 이 단원의 내용은 크로스 플랫폼 개발을 위한 툴바와 메뉴 설정에 적용됩니다. Windows 개발의 경우에도 여기서 설명한 방법을 사용할 수 있습니다. 그러나 액션 밴드를 사용하면 더 간단하고 다양한 옵션을 사용할 수 있습니다. 나열된 액션이 Action Manager에 의해 자동으로 처리됩니다. 액션 밴드 및 Action Manager 사용에 대한 자세한 내용은 8-16페이지의 "툴바와 메뉴에 대한 액션 구성"을 참조하십시오.

액션 리스트에는 사용자 작업에 대한 응답으로 애플리케이션에서 수행할 수 있는 액션이 저장됩니다. 액션 객체를 사용하여 사용자 인터페이스에서 애플리케이션에 의해 수행될 기능을 중앙 집중화합니다. 이렇게 하면 액션을 수행하기 위한 공통 코드를 공유할 수 있으며(예: 툴바 버튼과 메뉴 항목이 같은 작업을 수행할 때) 애플리케이션 상태에 따라 액션을 활성화하거나 비활성화하는 중앙화된 방법을 제공합니다.

액션 리스트 설정

액션 리스트 설정은 다음과 같은 기본 단계를 이해하고 나면 매우 쉽습니다.

- 액션 리스트을 만듭니다.
- 액션 리스트에 액션을 추가합니다.
- 액션의 속성을 설정합니다.
- 액션에 클라이언트를 연결합니다.

자세한 단계는 다음과 같습니다.

1 *TActionList* 객체를 폼이나 데이터 모듈에 가져다 놓습니다. ActionList는 컴포넌트 팔레트의 Standard 페이지에 있습니다.

2 *TActionList* 객체를 더블 클릭하여 Action List Editor를 표시합니다.

   a 에디터에 나열된 미리 정의된 액션 중 하나를 사용합니다. 마우스 오른쪽 버튼을 클릭한 다음 New Standard Action을 선택합니다.

   b 미리 정의된 액션은 Standard Action Classes 다이얼로그 박스에서 범주(Dataset, Edit, Help 및 Window 등)별로 구성됩니다. 액션 리스트에 추가할 표준 액션을 모두 선택하고 OK를 클릭합니다.

   또는

   c 새 액션을 직접 만듭니다. 마우스 오른쪽 버튼을 클릭하고 New Action을 선택합니다.

3 Object Inspector에서 각 액션의 속성을 설정합니다. 설정한 속성은 액션의 모든 클라이언트에 적용됩니다.

Name 속성은 액션을 식별하고 다른 속성과 이벤트(*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible* 및 *Execute*)는 해당 클라이언트 컨트롤의 속성과 이벤트에 해당됩니다. 클라이언트의 해당 속성은 대부분 해당 클라이언트 속성 이름과 같습니다. 예를 들어, 액션의 *Enabled* 속성은 *TToolButton*의 *Enabled* 속성에 해당합니다. 그러나 액션의 *Checked* 속성은 *TToolButton*의 *Down* 속성에 해당합니다.

4 미리 정의된 액션을 사용하면 액션에 자동으로 발생되는 표준 응답이 포함됩니다. 액션을 직접 만드는 경우에는 액션이 실행될 때 응답하는 방법을 정의하는 이벤트 핸들러를 작성해야 합니다. 자세한 내용은 8-24페이지의 "액션 실행 시 발생하는 이벤트"를 참조하십시오.

5 액션 리스트의 액션을 해당 액션이 필요한 클라이언트에 연결합니다.

- 폼이나 데이터 모듈에서 버튼이나 메뉴 항목 등의 컨트롤을 클릭합니다. Object Inspector에서 *Action* 속성은 사용 가능한 액션을 나열합니다.
- 원하는 액션을 선택합니다.

TEditDelete 또는 *TDataSetPost* 등의 표준 액션은 모두 예상 액션을 수행합니다. 필요하면 모든 표준 액션의 작업 방법에 대한 자세한 내용을 온라인 참조 도움말에서 참조할 수 있습니다. 직접 액션을 작성하려면 액션이 실행될 때 일어나는 동작을 더 깊이 이해해야 합니다.

액션 실행 시 발생하는 이벤트

이벤트가 발생하면 주로 일반적인 액션을 위한 일련의 이벤트가 발생합니다. 그런 다음 이벤트에서 액션을 처리하지 않으면 다른 이벤트 시퀀스가 발생합니다.

이벤트에 응답

클라이언트 컴포넌트 또는 컨트롤이 클릭되거나 다른 방법으로 활성화되면 응답할 수 있는 일련의 이벤트가 발생합니다. 예를 들어 다음 코드는 액션이 실행될 때 툴바의 표시 여부를 토글하는, 액션에 대한 이벤트 핸들러를 나타냅니다.

```
void __fastcall TForm1::Action1Execute(TObject *Sender)
{
  // Toggle Toolbar1's visibility
  ToolBar1->Visible = !ToolBar1->Visible;
}
```

액션, 액션 리스트 또는 애플리케이션 등 세 가지 레벨 중 하나로 응답하는 이벤트 핸들러를 제공할 수 있습니다. 이것은 미리 정의된 표준 액션을 사용하지 않고 새로운 일반 액션을 사용하려는 경우에만 문제가 됩니다. 표준 액션에는 이러한 이벤트가 발생할 때 실행되는 기본 동작이 있으므로 표준 액션을 사용하는 경우에는 신경쓸 필요가 없습니다.

이벤트 핸들러가 이벤트에 대해 응답하는 순서는 다음과 같습니다.

- 액션 리스트
- 애플리케이션
- 액션

사용자가 클라이언트 컨트롤을 클릭하면 C++Builder에서 가장 먼저 액션 리스트의 Execute 메소드를 호출한 다음 Application 객체를 호출하고, 액션 리스트나 Application 객체에서 처리하지 못하면 액션을 호출합니다. 이 과정을 더 자세히 설명하면, C++Builder가 사용자 액션에 응답하는 방법을 찾는 경우 다음 디스패칭 시퀀스를 따릅니다.

1 액션 리스트에 *OnExecute* 이벤트 핸들러를 제공하고 이 이벤트 핸들러가 액션을 처리하면 애플리케이션에서 진행됩니다.

   액션 리스트의 이벤트 핸들러에는 디폴트로 **false**를 반환하는 *Handled*라는 매개변수가 있습니다. 핸들러가 할당되고 핸들러에서 이벤트를 처리하면 **true**를 반환하고 처리 과정은 여기서 끝납니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TForm1::ActionList1ExecuteAction(TBasicAction *Action, bool
&Handled)
{
  Handled = true;
}
```

   액션 리스트 이벤트 핸들러에서 *Handled*를 **true**로 설정하지 않으면 처리 과정이 계속 진행됩니다.

2 액션 리스트의 *OnExecute* 이벤트 핸들러를 작성하지 않은 경우 또는 이벤트 핸들러가 액션을 처리하지 않는 경우, 애플리케이션의 *OnActionExecute* 이벤트 핸들러가 실행됩니다. 이벤트 핸들러가 액션을 처리하면 애플리케이션이 진행됩니다.

   애플리케이션의 액션 리스트에서 이벤트를 처리하지 못할 경우 전역 *Application* 객체가 *OnActionExecute* 이벤트를 받습니다. 액션 리스트의 *OnExecute* 이벤트 핸들러처럼 *OnActionExecute* 핸들러에는 디폴트로 **false**를 반환하는 *Handled*라는 매개변수가 있습니다. 이벤트 핸들러가 할당되고 이벤트가 처리되면 **true**가 반환되고 처리 과정은 여기서 끝납니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TForm1::ApplicationExecuteAction(TBasicAction *Action, bool
&Handled)
{
  // Prevent execution of all actions in Application
  Handled = true;
}
```

3 애플리케이션의 *OnExecute* 이벤트 핸들러가 액션을 처리하지 않으면 액션의 *OnExecute* 이벤트 핸들러가 실행됩니다.

기본 액션을 사용하거나 편집 컨트롤과 같은 특정 대상 클래스에서의 작업 방법을 아는 액션 클래스를 직접 만들 수 있습니다. 모든 레벨에서 이벤트 핸들러를 찾을 수 없으면 애플리케이션이 다음에 액션을 실행할 대상을 찾습니다. 애플리케이션에서 액션에서 작업 방법을 아는 대상을 찾으면 액션을 호출합니다. 애플리케이션에서 미리 정의된 액션 클래스에 응답할 수 있는 대상을 찾는 자세한 방법에 대해서는 다음 단원을 참조하십시오.

액션이 해당 대상을 찾는 방법

8-24페이지의 "액션 실행 시 발생하는 이벤트"에서는 사용자가 액션을 호출할 때 발생하는 실행 주기에 대해 설명합니다. 액션 리스트, 애플리케이션 또는 액션 레벨에서 액션에 응답하도록 이벤트 핸들러가 할당되지 않으면 애플리케이션에서 액션이 자체적으로 적용될 수 있는 대상 객체를 찾습니다.

애플리케이션은 다음 순서로 대상을 찾습니다.

1. 활성화된 컨트롤: 애플리케이션은 먼저 활성화된 컨트롤을 잠재적 대상으로 찾습니다.
2. 활성화된 폼 : 애플리케이션이 활성화된 컨트롤을 찾지 못하거나 활성화된 컨트롤이 대상 역할을 할 수 없는 경우 화면의 *ActiveForm*을 찾습니다.
3. 폼의 컨트롤: 활성화된 폼이 적절한 대상이 아닌 경우 애플리케이션은 대상의 활성화된 폼에서 다른 컨트롤을 찾습니다.

대상이 없으면 이벤트 실행 시 아무 일도 일어나지 않습니다.

일부 컨트롤은 검색을 확장하여 관련된 컴포넌트가 실행되는 시점까지 대상을 연기할 수 있습니다. 예를 들어, 데이터 인식 컨트롤은 관련 데이터셋 컴포넌트가 실행될 때까지 대상을 연기합니다. 또한 File Open 다이얼로그 박스 등 미리 정의된 어떤 액션에서는 대상을 사용하지 않습니다.

액션 업데이트

애플리케이션이 유휴 상태이면 나타난 컨트롤이나 메뉴 항목에 연결된 모든 액션에 대해 *OnUpdate* 이벤트가 발생합니다. 이를 통해 애플리케이션은 활성화 및 비활성화, 선택 및 선택 해제 등을 위해 중앙화된 코드를 실행할 수 있습니다. 예를 들어, 다음 코드는 툴바가 표시될 때 "선택된" 액션에 대한 *OnUpdate* 이벤트 핸들러를 설명한 것입니다.

```
void __fastcall TForm1::Action1Update(TObject *Sender)
{
  // Indicate whether ToolBar1 is currently visible
  ((TAction *)Sender)->Checked = ToolBar1->Visible;
}
```

경고 *OnUpdate* 이벤트 핸들러에는 시간이 많이 걸리는 코드를 추가하지 마십시오. 이 이벤트 핸들러는 애플리케이션이 유휴 상태일 때마다 실행됩니다. 이벤트 핸들러의 실행 시간이 너무 길면 전체 애플리케이션의 성능에 영향을 줍니다.

미리 정의된 액션 클래스

Action Manager를 마우스 오른쪽 버튼으로 클릭하거나 New Standard Action을 선택하여 애플리케이션에 미리 정의된 액션을 추가할 수 있습니다. 미리 정의된 액션 클래스와 해당 표준 액션이 나열된 New Standard Action Classes 다이얼로그 박스가 나타납니다. 이 액션들은 C++Builder에 포함되어 있으며 자동으로 액션을 수행하는 객체입니다. 미리 정의된 액션은 다음 클래스로 구성됩니다.

표 8.3 액션 클래스

클래스	설명
Edit	표준 편집 액션: 편집 컨트롤 대상에 사용됩니다. *TEditAction*은 각 클래스가 *ExecuteTarget* 메소드를 오버라이드하여 클립보드를 통해 복사, 잘라내기 및 붙여넣기 등을 구현하는 자손 클래스의 기본 클래스입니다.
Format	표준 서식 지정 액션: 서식있는 텍스트에 사용되어 굵게, 이탤릭체, 밑줄, 취소선 등의 텍스트 서식 지정 옵션을 적용합니다. *TRichEditAction*은 각 클래스가 *ExecuteTarget* 및 *UpdateTarget* 메소드를 오버라이드하여 대상의 서식 지정을 구현하는 자손 클래스의 기본 클래스입니다.
Help	표준 도움말 액션: 모든 대상에 사용됩니다. *THelpAction*은 각 클래스가 *ExecuteTarget* 메소드를 오버라이드하여 도움말 시스템에 명령을 전달하는 자손 클래스의 기본 클래스입니다.
Window	표준 윈도우 액션: 폼과 함께 MDI 애플리케이션의 대상으로 사용됩니다. *TWindowAction*은 각 클래스가 *ExecuteTarget* 메소드를 오버라이드하여 MDI 자식 폼 정렬, 계단식 정렬, 닫기, 바둑판식 배열 및 최소화 등을 구현하는 자손 클래스의 기본 클래스입니다.
File	파일 액션: File Open, File Run 또는 File Exit 같은 파일 작업에서 사용합니다.
Search	검색 액션: 검색 옵션과 함께 사용합니다. *TSearchAction*은 사용자가 편집 컨트롤 검색을 위해 검색 문자열을 입력할 수 있는 modeless 다이얼로그 박스를 표시하는 액션의 일반적인 동작을 구현합니다.
Tab	탭 컨트롤 액션: 마법사의 Prev 및 Next 버튼처럼 탭 컨트롤에서 탭 간을 이동하는 데 사용됩니다.
List	리스트 컨트롤 액션: 리스트 뷰에서 항목을 관리하는 데 사용됩니다.
Dialog	다이얼로그 박스 액션: 다이얼로그 박스 컴포넌트와 사용됩니다. *TDialogAction*은 실행 시 다이얼로그 박스를 표시하는 액션의 일반적인 동작을 구현합니다. 각 자손 클래스는 특정 다이얼로그 박스를 나타냅니다.
Internet	인터넷 액션: 인터넷 찾아보기, 다운로드 및 메일 보내기 등의 기능에 사용됩니다.
DataSet	데이터셋 액션: 데이터셋 컴포넌트 대상과 함께 사용됩니다. *TDataSetAction*은 각 클래스가 *ExecuteTarget* 및 *UpdateTarget* 메소드를 오버라이드하여 대상의 탐색과 편집 작업을 구현하는 자손 클래스의 기본 클래스입니다. *TDataSetAction*은 액션에는 반드시 해당 데이터셋에서 수행되게 하는 *DataSource* 속성이 있습니다. *DataSource*가 NULL이면 현재 포커스가 있는 데이터 인식 컨트롤이 사용됩니다.
Tools	툴: 자동으로 액션 밴드를 위한 사용자 정의 다이얼로그 박스를 표시하는 *TCustomizeActionBars* 등의 추가 툴입니다.

온라인 참조 도움말의 각 액션 객체 이름 아래에서 모든 액션 객체에 대해 설명합니다. 액션 객체의 작업 방법에 대한 자세한 내용은 도움말을 참조하십시오.

액션 컴포넌트 작성

미리 정의된 새로운 액션 클래스를 만들 수도 있습니다. 새로운 액션 클래스를 작성하면 객체의 특정 대상 클래스에 실행되는 기능을 생성할 수 있습니다. 그런 다음 미리 정의된 액션 클래스를 사용할 때와 같은 방법으로 사용자 정의 액션을 사용할 수 있습니다. 즉, 액션이 자신을 대상 클래스에 인식시키거나 적용할 수 있는 경우에는 이벤트 핸들러를 작성할 필요 없이 이 액션을 클라이언트 컨트롤에 할당하기만 하면 대상에서 작업하게 할 수 있습니다.

컴포넌트 작성자는 QStdActns 및 DBActns 유닛의 클래스를 특정 컨트롤이나 컴포넌트 고유의 동작을 구현하기 위해 액션 클래스를 파생하기 위한 예제로 사용할 수 있습니다. 이러한 특화된 액션(*TEditAction*, *TWindowAction* 등)의 기본 클래스는 일반적으로 *HandlesTarget*, *UpdateTarget* 및 기타 메소드를 오버라이드하여 액션의 대상을 특정 객체 클래스로 제한합니다. 자손 클래스는 일반적으로 *ExecuteTarget*을 오버라이드하여 특화된 작업을 수행합니다. 여기서는 다음 메소드에 대해 설명합니다.

메소드	설명
HandlesTarget	사용자가 액션에 연결된 객체(툴 버튼이나 메뉴 항목)를 호출할 때 자동으로 호출됩니다. *HandlesTarget* 메소드를 사용하면 액션 객체가 *Target* 매개변수에서 지정한 객체와 함께 지금 실행하는 것이 적절한지의 여부를 "대상"으로 나타내게 할 수 있습니다. 자세한 내용은 8-26페이지의 "액션이 해당 대상을 찾는 방법"을 참조하십시오.
UpdateTarget	애플리케이션이 유휴 상태일 때 호출되어 액션이 현재 상태에 따라 업데이트할 수 있게 합니다. *OnUpdateAction* 대신 사용합니다. 자세한 내용은 8-26페이지의 "액션 업데이트"를 참조하십시오.
ExecuteTarget	사용자 액션(예를 들어, 사용자가 이 액션에 연결된 메뉴 항목을 선택하거나 툴 버튼을 누를 때)에 대한 응답으로 액션이 실행될 때 *OnExecute* 대신 호출됩니다. 자세한 내용은 8-24페이지의 "액션 실행 시 발생하는 이벤트"를 참조하십시오.

액션 등록

직접 액션을 작성할 때 Action List Editor에 나타나도록 액션을 등록할 수 있습니다. Actnlist 유닛에서 전역 루틴을 사용하여 액션을 등록하거나 등록을 취소할 수 있습니다.

```
extern PACKAGE void __fastcall RegisterActions(const AnsiString
CategoryName, TMetaClass* const * AClasses, const int AClasses_Size,
TMetaClass* Resource);

extern PACKAGE void __fastcall UnRegisterActions(TMetaClass* const *
AClasses, const int AClasses_Size);
```

*RegisterActions*를 호출하면 등록한 액션이 애플리케이션에서 사용할 수 있도록 Action List Editor에 나타납니다. 디폴트 속성 값을 제공할 수 있게 해주는 *Resource* 매개변수는 물론 범주 이름을 제공하여 액션을 구성할 수 있습니다.

예를 들어, 다음 코드는 액션을 *MyAction* 유닛의 IDE에 등록합니다.

```
namespace MyAction
{
  void __fastcall PACKAGE Register()
  {
     // code goes here to register any components and editors
      TMetaClass classes[2] = {__classid(TMyAction1),
__classid(TMyAction2)};
      RegisterActions("MySpecialActions", classes, 1, NULL);
  }
}
```

*UnRegisterActions*를 호출하면 액션이 더 이상 Action List Editor에 나타나지 않습니다.

메뉴 생성 및 관리

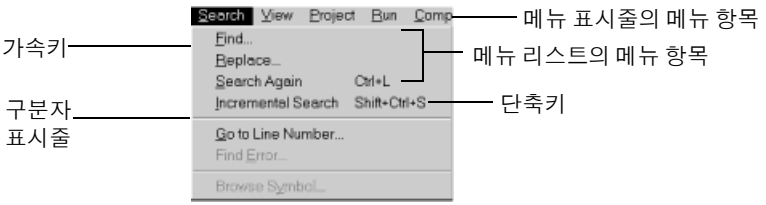
메뉴는 사용자가 그룹화된 명령을 로컬로 실행할 수 있는 쉬운 방법을 제공합니다. 메뉴 디자이너를 사용하면 미리 디자인된 메뉴나 사용자 정의 메뉴를 쉽게 폼에 추가할 수 있습니다. 폼에 메뉴 컴포넌트를 추가하고 메뉴 디자이너를 연 다음 메뉴 디자이너 윈도우에 직접 메뉴 항목을 입력합니다. 디자인 타임 중 메뉴 항목을 추가 또는 삭제하거나 드래그 앤 드롭하여 다시 정렬할 수 있습니다.

결과를 보기 위해 프로그램을 실행할 필요도 없습니다. 디자인은 런타임 중 나타날 모양 그대로 즉시 폼에 나타납니다. 사용자에게 더 자세한 정보나 옵션을 제공하기 위해 런타임 중 메뉴를 변경하도록 코딩할 수 있습니다.

이 단원에서는 메뉴 디자이너를 사용하여 메뉴 표시줄과 팝업(로컬) 메뉴를 디자인하는 방법에 대해 설명합니다. 다음과 같이 디자인 타임과 런타임에 메뉴를 사용하여 작업하는 방법에 대해 설명합니다.

- 메뉴 디자이너 열기
- 메뉴 생성
- Object Inspector에서 메뉴 항목 편집
- 메뉴 디자이너 컨텍스트 메뉴 사용
- 메뉴 템플릿 사용
- 메뉴를 템플릿으로 저장
- 메뉴 항목에 이미지 추가

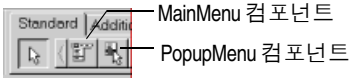
그림 8.3 메뉴 용어



메뉴 디자이너 열기

메뉴 디자이너를 사용하여 애플리케이션을 위한 메뉴를 디자인합니다. 메뉴 디자이너를 사용하려면 먼저 *MainMenu* 또는 *PopupMenu* 컴포넌트를 폼에 추가해야 합니다. 두 메뉴 컴포넌트는 컴포넌트 팔레트의 Standard 페이지에 있습니다.

그림 8.4 MainMenu 컴포넌트 및 PopupMenu 컴포넌트



MainMenu 컴포넌트는 폼의 제목 표시줄과 붙어 있는 메뉴를 만듭니다. PopupMenu 컴포넌트는 사용자가 폼에서 마우스 오른쪽 버튼을 클릭할 때 나타나는 메뉴를 만듭니다. 팝업 메뉴에는 메뉴 표시줄이 없습니다.

메뉴 디자이너를 열려면 폼에서 메뉴 컴포넌트를 선택한 다음

- 메뉴 컴포넌트를 더블 클릭합니다.

  또는

- Object Inspector의 Properties 페이지에서 *Items* 속성을 선택하고 Value 열의 [Menu]를 더블 클릭하거나 생략 부호(...) 버튼을 클릭합니다.

  메뉴 디자이너를 처음 열면 디자이너에서 선택한 첫 번째(비어 있는) 메뉴 항목과 Object Inspector에서 선택한 *Caption* 속성이 표시됩니다.

그림 8.5 메인 메뉴의 메뉴 디자이너



메뉴 생성

애플리케이션에 포함시킬 모든 메뉴마다 폼에 메뉴 컴포넌트를 추가합니다. 각 메뉴 구조를 처음부터 생성하거나 미리 디자인된 메뉴 템플릿 중 하나를 사용하여 시작할 수 있습니다.

이 단원에서는 디자인 타임에 메뉴를 작성하기 위한 기본 사항에 대해 설명합니다. 메뉴 템플릿에 대한 자세한 내용은 8-38페이지의 "메뉴 템플릿 사용"을 참조하십시오.

메뉴 이름 지정

다른 모든 컴포넌트와 마찬가지로 메뉴 컴포넌트를 폼에 추가하면 C++Builder가 *MainMenu1* 과 같은 디폴트 이름을 제공합니다. 그러나 보다 쉽게 알아볼 수 있는 이름을 메뉴에 지정할 수도 있습니다.

C++Builder에서 폼의 타입 선언에 메뉴 이름을 추가하면 메뉴 이름이 Component 리스트에 나타납니다.

메뉴 항목 이름 지정

메뉴 컴포넌트와는 반대로, 메뉴 항목을 폼에 추가할 때는 이름을 명시적으로 지정해야 합니다. 다음 두 가지 방법 중 하나로 지정할 수 있습니다.

- *Name* 속성에 값을 직접 입력합니다.
- 먼저 *Caption* 속성에 값을 입력하고 C++Builder가 캡션의 *Name* 속성을 파생시키게 합니다.

  예를 들어, 메뉴 항목에 *File*의 *Caption* 속성 값을 지정하면 C++Builder가 메뉴 항목을 *File1* 의 *Name* 속성에 할당합니다. *Caption* 속성을 채우기 전에 *Name* 속성을 채운 경우에는 값을 입력할 때까지 C++Builder가 *Caption* 속성을 비워 둡니다.

참고 C++ 식별자로 사용할 수 없는 글자를 *Caption* 속성에 입력하면 C++Builder가 *Name* 속성을 적절하게 수정합니다. 예를 들어, 캡션이 숫자로 시작되게 하려면 C++Builder에서 숫자 앞에 문자를 추가하여 *Name* 속성을 파생시킵니다.

다음 표에서는 표시된 모든 메뉴 항목이 동일한 메뉴 표시줄에 나타난다는 가정 하에 이에 관한 예제를 보여 줍니다.

표 8.4 예제 캡션 및 파생된 이름

컴포넌트 캡션	파생된 이름	설명
&File	File1	앰퍼샌드(&) 제거
&File(두 번째 나타남)	File2	중복된 항목을 번호로 지정
1234	N12341	선행 문자와 번호 추가
1234(두 번째 나타남)	N12342	파생된 이름을 명확하게 나타내는 숫자 추가
$@@@#	N1	모든 비표준 문자를 제거하고 선행 문자 및 번호 추가
- (하이픈)	N2	비표준 문자가 있는 캡션이 두 번째 나타날 때 순차적으로 정렬

메뉴 컴포넌트에서 C++Builder가 폼의 타입 선언에 메뉴 항목 이름을 추가하면 Component 리스트에 추가된 메뉴 항목 이름이 나타납니다.

메뉴 항목 추가, 삽입 및 삭제

다음은 메뉴 구조 생성과 관련된 기본 작업을 수행하는 방법을 설명한 절차입니다. 각 절차에서는 메뉴 디자이너 윈도우가 열려 있다는 것을 전제로 합니다.

다음과 같은 방법으로 디자인 타임에 메뉴 항목을 추가합니다.

1 메뉴 항목을 만들 위치를 선택합니다.

  메뉴 디자이너를 열면 메뉴 표시줄의 첫 번째 위치가 이미 선택되어 있습니다.

2 캡션 입력을 시작합니다. 또는 Object Inspector에서 커서를 특정 위치에 두고 값을 입력하여 *Name* 속성을 먼저 입력합니다. 이 경우에는 다음에 *Caption* 속성을 다시 선택하여 값을 입력해야 합니다.

3 *Enter* 키를 누릅니다.

그러면 메뉴 항목의 다음 위치 표시자가 선택됩니다.

Caption 속성을 먼저 입력했으면 화살표 키를 사용하여 방금 입력한 메뉴 항목으로 돌아 갑니다. C++Builder가 캡션에 입력한 값을 기초로 *Name* 속성을 입력한 것을 볼 수 있습니다. 8-31페이지의 "메뉴 항목 이름 지정"을 참조하십시오.

4 계속해서 만들려는 새 항목마다 *Name* 및 *Caption* 속성에 값을 입력하거나 ***Esc***를 눌러 메뉴 표시줄로 돌아갑니다.

화살표 키를 사용하여 메뉴 표시줄에서 메뉴로 이동한 다음 리스트 항목 사이를 이동하고 *Enter*를 눌러 액션을 끝냅니다. 메뉴 표시줄로 돌아 오려면 ***Esc***를 누릅니다.

다음과 같은 방법으로 비어 있는 새 메뉴 항목을 삽입합니다.

1 메뉴 항목에 커서를 둡니다.

2 *Ins*를 누릅니다.

메뉴 표시줄에서 선택한 항목 왼쪽 및 메뉴 리스트에서 선택한 항목 위에 메뉴 항목이 삽입됩니다.

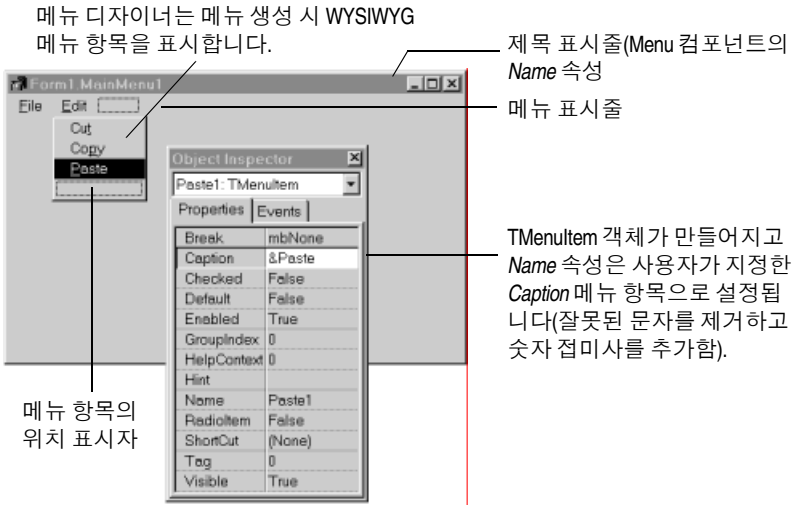
다음과 같은 방법으로 메뉴 항목 또는 명령을 삭제합니다.

1 삭제할 메뉴 항목에 커서를 둡니다.

2 *Del*을 누릅니다.

참고 메뉴 리스트에 마지막으로 입력한 항목 아래 또는 메뉴 표시줄의 마지막 항목 옆에 있는 디폴트 위치 표시자는 삭제할 수 없습니다. 이 위치 표시자는 런타임에 메뉴에 나타나지 않습니다.

그림 8.6 메인 메뉴에 메뉴 항목 추가



구분자 표시줄 추가

구분자 표시줄은 메뉴 항목 간에 줄을 삽입합니다. 구분자 표시줄을 사용하여 메뉴 항목 내의 그룹화를 나타내거나 단순히 리스트에서 시각적인 구분을 표시할 수 있습니다.

메뉴 항목을 구분자 표시줄로 만들려면 캡션에 하이픈(-)을 입력합니다.

가속키 및 단축키 지정

가속키를 사용하면 사용자가 *Alt* 키와 코드에서 앰퍼샌드 뒤에 지정한 해당 문자를 눌러 키보드를 사용하여 메뉴 명령에 액세스할 수 있습니다. 앰퍼샌드 뒤의 문자는 메뉴에서 밑줄로 표시됩니다.

C++Builder는 자동으로 중복된 가속키를 검사하여 런타임에 이를 조정합니다. 이렇게 하면 런타임에 동적으로 생성된 메뉴에 중복된 가속키가 포함되지 않고 모든 메뉴 항목이 가속키를 갖게 됩니다. 메뉴 항목의 AutoHotkeys 속성을 maManual로 설정하여 이 자동 검사를 해제할 수 있습니다.

다음과 같은 방법으로 가속키를 지정합니다.

- 해당 문자 앞에 앰퍼샌드를 추가합니다.

  예를 들어, 가속키가 *S*인 Save 메뉴 명령을 추가하려면 `&Save`를 입력합니다.

키보드 단축키를 사용하여 메뉴를 직접 사용하지 않고 단축키 조합으로 입력하여 액션을 수행할 수 있습니다.

다음과 같은 방법으로 단축키를 지정합니다.

- Object Inspector를 사용하여 *ShortCut* 속성에 값을 입력하거나 드롭다운 리스트에서 키 조합을 선택합니다.

  이 리스트는 사용자가 입력할 수 있는 키 조합들을 표시합니다.

추가한 단축키는 메뉴 항목 캡션 옆에 나타납니다.

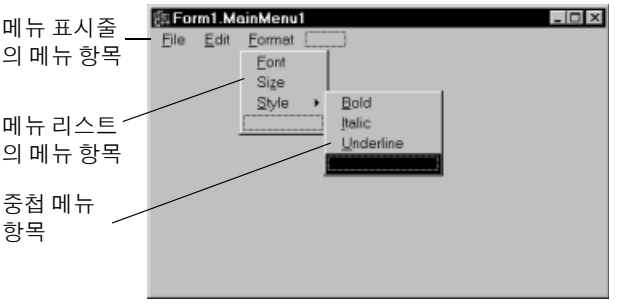
주의 단축키는 가속키와 달리 중복 여부가 자동으로 검사되지 않습니다. 고유성을 직접 확인해야 합니다.

하위 메뉴 생성

많은 애플리케이션 메뉴에는 메뉴 항목 옆에 나타나 추가로 관련 명령을 제공하는 드롭다운 리스트가 있습니다. 이러한 리스트가 있는 메뉴 항목에는 오른쪽에 화살표가 나타납니다. C++Builder에서는 메뉴에 원하는 만큼의 하위 메뉴 레벨을 지원합니다.

이러한 방법으로 메뉴 구조를 구성하면 세로 방향의 화면 공간을 절약할 수 있습니다. 그러나 인터페이스를 디자인할 때 메뉴 레벨 수가 2-3개를 넘지 않는 것이 좋습니다. 팝업 메뉴의 경우 하위 메뉴는 하나까지만 사용할 수 있습니다.

그림 8.7 중첩 메뉴 구조



다음과 같은 방법으로 하위 메뉴를 만듭니다.

1 하위 메뉴를 만들 메뉴 항목을 선택합니다.

2 *Ctrl*→ 키를 눌러 첫 번째 위치 표시자를 만들거나 마우스 오른쪽 버튼을 클릭한 다음 Create Submenu를 선택합니다.

3 하위 메뉴 항목의 이름을 입력하거나 기존 메뉴 항목을 이 위치 표시자로 끌어 놓습니다.

4 *Enter* 또는 ↓를 눌러 다음 위치 표시자를 만듭니다.

5 하위 메뉴에서 만들 각 항목에 대해 단계 3과 단계 4를 반복합니다.

6 *Esc*를 눌러 이전 메뉴 레벨로 돌아갑니다.

기존 메뉴를 밑으로 내려 하위 메뉴 생성

메뉴 표시줄이나 메뉴 템플릿의 메뉴 항목을 리스트에 있는 메뉴 항목 간에 삽입하여 하위 메뉴를 만들 수 있습니다. 기존 메뉴 구조로 메뉴를 이동하면 관련 항목이 모두 함께 이동되어 하위 메뉴가 그대로 유지됩니다. 이는 하위 메뉴에도 적용됩니다. 메뉴 항목을 기존 하위 메뉴로 이동하면 하나 이상의 중첩 레벨만 만들어집니다.

메뉴 항목 이동

디자인 타임 중 드래그 앤 드롭하여 간단히 메뉴 항목을 이동할 수 있습니다. 메뉴 표시줄을 따라, 메뉴 리스트의 다른 위치로, 또는 완전히 다른 메뉴로 메뉴 항목을 이동할 수 있습니다.

단, 계층이 있는 메뉴는 예외이며, 메뉴 항목을 메뉴 표시줄에서 각각의 메뉴로 레벨을 낮추거나 각각의 하위 메뉴로 이동할 수 없습니다. 그러나 원래의 위치에 관계 없이 항목을 *다른* 메뉴로 이동할 수는 있습니다.

커서를 끄는 동안에는 메뉴 항목을 새 위치에 놓을 수 있는지 여부를 알 수 있도록 모양이 변경됩니다. 메뉴 항목을 이동하면 모든 하위 항목도 함께 이동합니다.

다음과 같은 방법으로 메뉴 표시줄에서 메뉴 항목을 이동합니다.

1 커서의 화살표 끝이 새 위치를 가리킬 때까지 메뉴 표시줄을 따라 메뉴 항목을 끕니다.

2 마우스 버튼을 놓아 메뉴 항목을 새 위치에 놓습니다.

다음과 같은 방법으로 메뉴 리스트로 메뉴 항목을 이동합니다.

1 커서의 화살표 끝이 새 메뉴를 가리킬 때까지 메뉴 표시줄에서 메뉴 항목을 끕니다.

이렇게 하면 메뉴가 열리므로 메뉴 항목을 새 위치로 끌어 놓을 수 있습니다.

2 메뉴 항목을 리스트로 끌어 놓은 다음 마우스 버튼을 놓아 메뉴 항목을 새 위치에 놓습니다.

메뉴 항목에 이미지 추가

이미지를 사용하면 툴바의 이미지처럼 사용자가 문자 모양과 이미지를 메뉴 항목 액션과 연관시켜 메뉴를 탐색할 수 있습니다. 메뉴 항목에 하나의 비트맵을 추가하거나, 애플리케이션의 이미지를 이미지 리스트로 구성하여 이미지 리스트에서 메뉴에 추가할 수 있습니다. 애플리케이션에서 같은 크기의 비트맵을 여러 개 사용하는 경우에는 이미지 리스트에 두는 것이 좋습니다.

메뉴 또는 메뉴 항목에 단일 이미지를 추가하려면 이미지의 *Bitmap* 속성이 메뉴 또는 메뉴 항목에서 사용할 비트맵 이름을 참조하도록 설정합니다.

다음과 같은 방법으로 이미지 리스트를 사용하여 메뉴 항목에 이미지를 추가합니다.

1 *TMainMenu* 또는 *TPopupMenu* 객체를 폼에 가져다 놓습니다.

2 *TImageList* 객체를 폼에 가져다 놓습니다.

3 *TImageList* 객체를 더블 클릭하여 ImageList Editor를 엽니다.

4 Add를 클릭하여 메뉴에서 사용할 비트맵 또는 비트맵 그룹을 선택합니다. OK를 클릭합니다.

5 *TMainMenu* 또는 *TPopupMenu* 객체의 *Images* 속성을 방금 만든 ImageList로 설정합니다.

6 앞에서 설명한 대로 메뉴 항목과 하위 메뉴 항목을 만듭니다.

7 Object Inspector에서 이미지를 갖고자 하는 메뉴 항목을 선택하고 *ImageIndex* 속성을 *ImageList*의 해당 이미지 번호로 설정합니다. *ImageIndex*의 기본값은 -1이며 이 값은 이미지를 나타내지 않습니다.

참고 메뉴에 제대로 표시하려면 16x16픽셀의 이미지를 사용합니다. 메뉴 이미지에 대해 다른 크기를 사용할 수 있으나 16x16픽셀 이외의 이미지를 사용하면 배열 및 일관성 문제가 발생할 수 있습니다.

메뉴 보기

먼저 프로그램 코드를 실행하지 않고 디자인 타임에 폼에서 메뉴를 볼 수 있습니다. 디자인 타임에 팝업 메뉴 컴포넌트는 폼에 나타나지만 팝업 메뉴 자체는 나타나지 않습니다. 디자인 타임에 팝업 메뉴를 보려면 메뉴 디자이너를 사용하십시오.

다음과 같은 방법으로 메뉴를 봅니다.

1 폼이 보이는 경우 폼을 클릭하거나 View 메뉴에서 보고자 하는 메뉴의 폼을 선택합니다.

2 폼에 메뉴가 둘 이상 있으면 폼의 *Menu* 속성 드롭다운 리스트에서 보고자 하는 메뉴를 선택합니다.

   프로그램을 실행할 때와 똑같이 폼에 메뉴가 나타납니다.

Object Inspector에서 메뉴 항목 편집

이 단원에서는 메뉴 항목의 여러 속성을 설정하는 방법에 대해 설명했습니다. 예를 들어, 메뉴 디자이너를 사용하여 *Name* 및 *Caption* 속성을 설정하는 방법을 살펴 보았습니다.

또한 이 단원에서는 폼에서 선택한 컴포넌트의 속성을 설정하는 것처럼 Object Inspector에서 직접 *ShortCut* 속성과 같은 메뉴 항목 속성을 설정하는 방법에 대해 설명했습니다.

메뉴 디자이너를 사용하여 메뉴 항목을 편집해도 해당 속성이 Object Inspector에 나타납니다. Object Inspector로 포커스를 전환하여 여기서 메뉴 항목 속성을 계속 편집할 수 있습니다. 또는 Object Inspector의 Component 리스트에서 메뉴 항목을 선택하고 메뉴 디자이너를 열지 않고 속성을 편집할 수 있습니다.

다음과 같은 방법으로 메뉴 디자이너 윈도우를 닫고 메뉴 항목을 계속 편집합니다.

1 Object Inspector의 속성 페이지를 클릭하여 메뉴 디자이너 윈도우에서 Object Inspector로 포커스를 전환합니다.

2 일반적인 방법으로 메뉴 디자이너를 닫습니다.

   포커스는 아직 Object Inspector에 있습니다. 여기서 선택한 메뉴 항목의 속성을 계속 편집할 수 있습니다. 다른 메뉴 항목을 편집하려면 Component 리스트에서 메뉴 항목을 선택합니다.

메뉴 디자이너 컨텍스트 메뉴 사용

메뉴 디자이너 컨텍스트 메뉴를 사용하면 가장 자주 사용하는 메뉴 디자이너 명령과 메뉴 템플릿 옵션에 빠르게 액세스할 수 있습니다. 메뉴 템플릿에 대한 자세한 내용은 8-38페이지의 "메뉴 템플릿 사용"을 참조하십시오.

컨텍스트 메뉴를 표시하려면 메뉴 디자이너 윈도우를 마우스 오른쪽 버튼으로 클릭하거나 커서가 메뉴 디자이너 윈도우에 있을 때 *Alt+F10*을 누르십시오.

컨텍스트 메뉴의 명령

다음 표는 메뉴 디자이너 컨텍스트 메뉴의 명령에 대해 요약하여 설명한 것입니다.

표 8.5 메뉴 디자이너 컨텍스트 메뉴 명령

메뉴 명령	액션
Insert	커서의 위쪽 또는 왼쪽에 위치 표시자를 삽입합니다.
Delete	선택한 메뉴 항목 및 해당 하위 항목을 모두 삭제합니다.
Create Submenu	중첩 레벨에 위치 표시자를 만들고 선택한 메뉴 항목 오른쪽에 화살표를 추가합니다.

표 8.5 메뉴 디자이너 컨텍스트 메뉴 명령 (계속)

메뉴 명령	액션
Select Menu	현재 폼에서 메뉴 리스트를 엽니다. 메뉴 이름을 더블 클릭하면 메뉴 디자이너 윈도우가 열립니다.
Save As Template	다음에 재사용할 메뉴를 저장할 수 있는 Save Template 다이얼로그 박스가 열립니다.
Insert From Template	재사용할 템플릿을 선택할 수 있는 Insert Template 다이얼로그 박스가 열립니다.
Delete Templates	기존 템플릿을 삭제할 수 있는 Delete Templates 다이얼로그 박스가 열립니다.
Insert From Resource	현재 폼에서 열 .rc 또는 .mnu 파일을 선택할 수 있는 Insert Menu from Resource File 다이얼로그 박스가 열립니다.

디자인 타임 시 메뉴 간의 전환

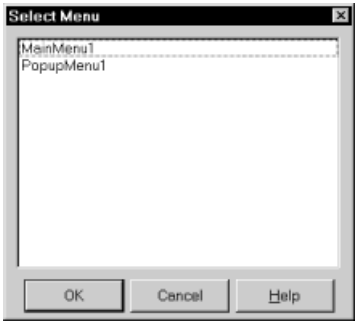
폼에 대한 여러 메뉴를 디자인하는 경우, 메뉴 디자이너 컨텍스트 메뉴 또는 Object Inspector를 사용하여 메뉴를 쉽게 선택하고 메뉴 사이를 전환할 수 있습니다.

다음과 같은 방법으로 컨텍스트 메뉴를 사용하여 폼과 메뉴 간에 전환합니다.

1 메뉴 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Select Menu를 선택합니다.

Select Menu 다이얼로그 박스가 나타납니다.

그림 8.8 Select Menu 다이얼로그 박스



이 다이얼로그 박스에는 메뉴 디자이너에서 현재 메뉴가 열려 있는 폼과 관련된 메뉴 리스트가 모두 나열됩니다.

2 Select Menu 다이얼로그 박스의 리스트에서 보거나 편집할 메뉴를 선택합니다.

다음과 같은 방법으로 Object Inspector를 사용하여 폼의 메뉴 간을 전환합니다.

1 선택할 메뉴의 폼에 포커스를 맞춥니다.

2 컴포넌트 리스트에서 편집할 메뉴를 선택합니다.

3 Object Inspector의 Properties 페이지에서 이 메뉴에 대한 *Items* 속성을 선택한 다음 생략 부호 버튼을 클릭하거나 [Menu]를 더블 클릭합니다.

메뉴 템플릿 사용

C++Builder에서는 여러 가지 미리 디자인된 메뉴 또는 메뉴 템플릿을 제공하는데 여기에는 자주 사용하는 명령이 들어 있습니다. 애플리케이션에서 이러한 메뉴를 수정하지 않고 사용하거나(코드 작성 제외), 이 메뉴를 시작점으로 사용하여 원하는 대로 사용자 정의할 수 있습니다. 메뉴 템플릿에는 이벤트 핸들러 코드가 없습니다.

C++Builder와 함께 제공된 메뉴 템플릿은 디폴트 설치의 BIN 하위 디렉토리에 저장됩니다. 이러한 파일에는 .dmt(C++Builder 메뉴 템플릿) 확장자가 있습니다.

메뉴 디자이너를 사용하여 직접 디자인한 메뉴를 템플릿으로 저장할 수도 있습니다. 메뉴를 템플릿으로 저장하면 미리 디자인된 메뉴처럼 사용할 수 있습니다. 특정 메뉴 템플릿을 더 이상 사용하지 않으려면 리스트에서 삭제할 수 있습니다.

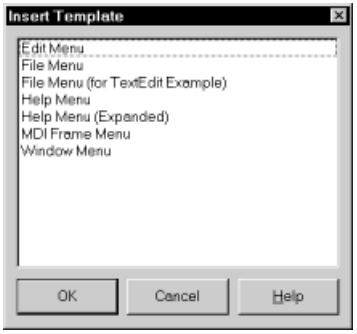
다음과 같은 방법으로 애플리케이션에 메뉴 템플릿을 추가합니다.

1 메뉴 디자이너를 마우스 오른쪽 버튼으로 클릭한 다음 Insert From Template을 선택합니다.

템플릿이 없는 경우에는 컨텍스트 메뉴에 Insert From Template 옵션이 흐리게 나타납니다.

Insert Template 다이얼로그 박스가 열리면서 사용 가능한 메뉴 템플릿 리스트가 표시됩니다.

그림 8.9 메뉴를 위한 예제 Insert Template 다이얼로그 박스



2 삽입할 메뉴 템플릿을 선택한 다음 *Enter* 키를 누르거나 OK를 선택합니다.

이렇게 하면 폼에서 현재 커서가 있는 위치에 메뉴가 삽입됩니다. 예를 들어, 커서가 리스트의 메뉴 항목에 있으면 메뉴 템플릿이 선택한 항목 위에 삽입됩니다. 커서가 메뉴 표시줄에 있으면 메뉴 템플릿이 커서 왼쪽에 삽입됩니다.

다음과 같은 방법으로 메뉴 템플릿을 삭제합니다.

1 메뉴 디자이너를 마우스 오른쪽 버튼으로 클릭한 다음 Delete Templates를 선택합니다.

템플릿이 없는 경우, 컨텍스트 메뉴에 Delete Templates 옵션이 흐리게 나타납니다.

Delete Templates 다이얼로그 박스가 열리면서 사용 가능한 템플릿 리스트가 나열됩니다.

2 삭제할 메뉴 템플릿을 선택한 다음 *Delete* 키를 누릅니다.

C++Builder는 템플릿 리스트와 하드 디스크에서 템플릿을 삭제합니다.

메뉴를 템플릿으로 저장

디자인한 메뉴를 템플릿으로 저장하여 다시 사용할 수 있습니다. 메뉴 템플릿을 사용하여 애플리케이션에 일관성 있는 모양을 제공하거나, 메뉴 템플릿에서 시작하여 추가로 사용자 정의할 수 있습니다.

저장한 메뉴 템플릿은 BIN 하위 디렉토리에 .dmt 파일로 저장됩니다.

다음과 같은 방법으로 메뉴를 템플릿으로 저장합니다.

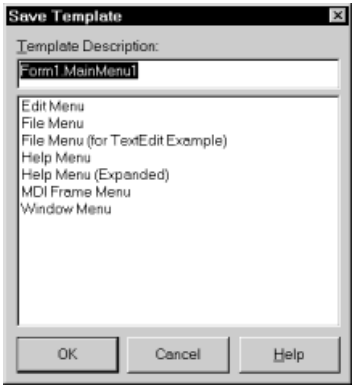
1 재사용할 수 있는 메뉴를 디자인합니다.

이 메뉴에는 원하는 만큼의 항목, 명령 및 하위 메뉴를 추가할 수 있습니다. 활성화된 메뉴 디자이너 윈도우의 모든 것이 재사용 가능한 메뉴로 저장됩니다.

2 메뉴 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Save As Template을 선택합니다.

Save Template 다이얼로그 박스가 나타납니다.

그림 8.10 메뉴에 대한 Save Template 다이얼로그 박스



3 Template Description 에디트 박스에 이 메뉴에 대한 간단한 설명을 입력한 다음 OK를 선택합니다.

Save Template 다이얼로그 박스가 닫히면서 메뉴 디자인을 저장하고 메뉴 디자이너 윈도우로 돌아갑니다.

참고 입력한 설명은 Save Template, Insert Template 및 Delete Templates 다이얼로그 박스에만 나타납니다. 메뉴의 *Name* 또는 *Caption* 속성과는 연관이 없습니다.

템플릿 메뉴 항목 및 이벤트 핸들러의 이름 지정 규칙

메뉴를 템플릿으로 저장하면 모든 메뉴가 해당 소유자(폼) 범위 내에서 고유 이름을 가지므로 C++Builder가 해당 *Name* 속성을 저장할 수 없습니다. 그러나 메뉴 디자이너를 사용하여 메뉴를 템플릿 형태로 새 폼에 삽입하면 C++Builder가 새 이름과 모든 해당 항목을 만듭니다.

예를 들어, File 메뉴를 템플릿으로 저장하는 경우를 가정합니다. 원본 메뉴에서 *MyFile*이라고 이름을 지정합니다. 이 메뉴를 템플릿 형태로 새 메뉴에 삽입하면 C++Builder가 *File1*이라는 이름을 지정합니다. *File1*이라는 기존 메뉴 항목이 있는 메뉴에 삽입하면 C++Builder가 *File2*라는 이름을 지정합니다.

또한 코드가 새 폼에서 제대로 작동하는지 여부를 테스트할 방법이 없으므로 C++Builder는 템플릿으로 저장한 메뉴와 관련된 *OnClick* 이벤트 핸들러를 저장하지 않습니다. 메뉴 템플릿 항목에 대해 새 이벤트 핸들러를 만들면 C++Builder가 이벤트 핸들러 이름을 만듭니다.

메뉴 템플릿의 항목을 폼의 기존 *OnClick* 이벤트 핸들러와 쉽게 연결할 수 있습니다.

런타임 시 메뉴 항목 처리

애플리케이션이 실행 중일 때 메뉴 항목을 기존 메뉴 구조에 추가하여 사용자에게 정보나 옵션을 더 제공해야 할 때가 있습니다. 메뉴 항목의 *Add* 또는 *Insert* 메소드를 사용하여 메뉴 항목을 삽입하거나, *Visible* 속성을 변경하여 메뉴에서 항목을 숨기거나 표시할 수 있습니다. *Visible* 속성은 메뉴 항목에 메뉴에 표시되는지 여부를 결정합니다. 메뉴 항목을 숨기지 않고 흐리게 나타내려면 *Enabled* 속성을 사용하십시오.

메뉴 항목의 *Visible* 및 *Enabled* 속성을 사용하는 예제는 6-9페이지의 "메뉴 항목 비활성화"를 참조하십시오.

다중 문서 인터페이스(MDI) 및 객체 연결 및 포함(OLE) 애플리케이션에서 메뉴 항목을 기존 메뉴 표시줄로 병합할 수도 있습니다. 다음 단원에서는 이 내용에 대해 자세히 설명합니다.

메뉴 병합

텍스트 에디터 예제 애플리케이션과 같은 MDI 애플리케이션과 OLE 클라이언트 애플리케이션의 경우에는 애플리케이션의 메인 메뉴에서 다른 폼이나 OLE 서버 객체로부터 메뉴 항목을 받을 수 있어야 합니다. 이를 *메뉴 병합*이라고 합니다. OLE 기술은 Windows 애플리케이션으로만 제한되어 있으므로 크로스 플랫폼 프로그래밍에서는 사용할 수 없습니다.

두 속성에 대해 값을 지정하여 병합을 위한 메뉴를 준비합니다.

- 폼의 속성인 *Menu*
- 메뉴의 메뉴 항목 속성인 *GroupIndex*

활성 메뉴 지정: Menu 속성

Menu 속성은 폼의 활성 메뉴를 지정합니다. 메뉴 병합 작업은 활성 메뉴에만 적용됩니다. 폼에 메뉴 컴포넌트가 둘 이상 있으면 코드에서 *Menu* 속성을 설정하여 런타임에 활성 메뉴를 변경할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Form1->Menu = SecondMenu;
```

병합된 메뉴 항목의 순서 결정: GroupIndex 속성

GroupIndex 속성은 병합 메뉴 항목이 공유 메뉴 표시줄에 나타나는 순서를 결정합니다. 병합 메뉴 항목은 메인 메뉴 표시줄의 메뉴 항목 대신 사용되거나 삽입될 수 있습니다.

*GroupIndex*의 기본값은 0입니다. *GroupIndex*의 값을 지정할 때는 여러 가지 규칙이 적용됩니다.

- 낮은 숫자가 메뉴에서 먼저(왼쪽부터) 나타납니다.

  예를 들어, File 메뉴처럼 항상 가장 왼쪽에 나타낼 메뉴의 *GroupIndex* 속성을 0으로 설정합니다. 마찬가지로, Help 메뉴처럼 가장 오른쪽에 나타낼 메뉴에는 높은 숫자를 지정합니다.

- 메인 메뉴의 항목을 바꾸려면 자식 메뉴의 항목에 같은 *GroupIndex* 값을 지정하십시오.

  이 방법은 그룹화 또는 단일 항목에 적용될 수 있습니다. 예를 들어, 메인 폼에 *GroupIndex* 값이 1로 설정된 Edit 메뉴가 있으면 자식 폼의 메뉴에 있는 하나 이상의 항목에 *GroupIndex* 값을 1로 지정하여 대신 사용할 수 있습니다.

  자식 메뉴의 여러 항목에 같은 *GroupIndex* 값을 지정하면 메인 메뉴로 병합할 때 순서가 그대로 유지됩니다.

- 메인 메뉴의 항목을 대체하지 않고 삽입하려면 메인 메뉴 항목의 번호 범위에 자리를 남겨두고 자식 폼에서 번호를 삽입합니다.

  예를 들어, 메인 메뉴 항목에 0과 5라는 번호를 지정하고 자식 메뉴의 항목에 1, 2, 3, 4를 지정한 다음 삽입합니다.

리소스 파일 임포트

C++Builder는 표준 Windows 리소스(.RC) 파일 형식을 따르는 다른 애플리케이션의 메뉴 사용을 지원합니다. 이러한 메뉴를 직접 C++Builder 프로젝트로 임포트하면 메뉴를 다시 생성해야 하는 시간과 수고를 절약할 수 있습니다.

다음과 같은 방법으로 .RC 메뉴 파일을 로드합니다.

1. 메뉴 디자이너에서 메뉴를 표시할 위치에 커서를 둡니다.

   임포트된 메뉴는 디자인하고 있는 메뉴의 일부가 되거나 전체 메뉴 자체가 될 수 있습니다.

2. 마우스 오른쪽 버튼을 클릭하고 Insert From Resource를 선택합니다.

   Insert Menu From Resource 다이얼로그 박스가 나타납니다.

3. 다이얼로그 박스에서 로드할 리소스 파일을 선택하고 OK를 선택합니다.

   메인 디자이너 윈도우에 메뉴가 나타납니다.

참고 리소스 파일에 메뉴가 둘 이상 있으면 메뉴를 임포트하기 전에 먼저 각각의 리소스 파일로 저장해야 합니다.

툴바 및 쿨바(cool bar) 디자인

*툴바*는 보통 폼의 위쪽 메뉴 표시줄 아래에 가로질러 나타나는 패널로 버튼과 다른 컨트롤들을 포함합니다. *쿨바(cool bar)*(rebar라고도 함)는 이동 가능하고 크기를 조정할 수 있는 밴드에 컨트롤을 표시하는 일종의 툴바입니다. 여러 패널을 폼의 위쪽에 배열한 경우에는 추가한 순서대로 세로로 쌓입니다.

참고 쿨바는 CLX에서 크로스 플랫폼 애플리케이션에 사용할 수 없습니다.

모든 종류의 컨트롤을 툴바에 둘 수 있습니다. 버튼 외에도 색상 그리드, 스크롤 막대, 레이블 등을 둘 수 있습니다.

여러 가지 방법으로 폼에 툴바를 추가할 수 있습니다.

- 폼에 패널(*TPanel*)을 두고 패널에 컨트롤(일반적으로 스피드 버튼)을 추가합니다.
- *TPanel* 대신 툴바 컴포넌트(*TToolBar*)를 사용하여 컨트롤을 추가합니다. *TToolBar*는 버튼과 다른 컨트롤을 관리하여 행으로 정렬하고 크기위 위치를 자동으로 조정합니다. 툴바에서 툴 버튼(*TToolButton*)을 사용하는 경우에는 *TToolBar*를 사용하여 쉽게 버튼을 기능별로 그룹화하고 다른 디스플레이 옵션을 제공할 수 있습니다.
- 쿨바(*TCoolBar*) 컴포넌트를 사용하여 컨트롤을 추가합니다. 쿨바에는 독립적으로 이동 가능하고 크기를 조정할 수 있는 밴드에 컨트롤이 나타납니다.

툴바를 구현하는 방법은 애플리케이션에 따라 다릅니다. 패널 컴포넌트를 사용하면 툴바의 모양과 느낌을 전체적으로 제어할 수 있다는 장점이 있습니다.

원시 Windows 컨트롤을 사용하고 있으므로 툴바와 쿨바 컴포넌트를 사용하여 애플리케이션이 Windows 애플리케이션의 모양과 느낌을 갖게 합니다. 나중에 이러한 운영 체제 컨트롤이 변경되면 애플리케이션도 변경됩니다. 또한 툴바와 쿨바는 Windows의 공통 컴포넌트에 거의 의존하지 않으므로 애플리케이션에 COMCTL32.DLL이 필요합니다. 툴바와 쿨바는 WinNT 3.51 애플리케이션에서 지원되지 않습니다.

다음 단원에서는 다음과 같은 액션을 수행하는 방법에 대해 설명합니다.

- 패널 컴포넌트를 사용하여 툴바와 해당 스피드 버튼 컨트롤 추가
- 툴바 컴포넌트를 사용하여 툴바와 해당 툴 버튼 컨트롤 추가
- 쿨바 컴포넌트를 사용하여 쿨바 추가
- 클릭에 응답
- 숨겨진 툴바와 쿨바 추가
- 툴바와 쿨바 숨기기/표시

패널 컴포넌트를 사용하여 툴바 추가

다음과 같은 방법으로 패널 컴포넌트를 사용하여 폼에 툴바를 추가합니다.

1 컴포넌트 팔레트의 Standard 페이지에 있는 패널 컴포넌트를 폼에 추가합니다.

2 패널의 *Align* 속성을 *alTop*으로 설정합니다.패널이 폼 맨 위에 정렬되면 윈도우 크기가 변경되어도 패널 높이는 유지되지만 패널 너비는 폼 클라이언트 영역의 전체 너비에 맞춰집니다.

3 패널에 스피드 버튼 또는 다른 컨트롤을 추가합니다.

스피드 버튼은 툴바 패널에서 작동하도록 디자인되어 있습니다. 스피드 버튼에는 보통 캡션이 없고 버튼의 기능을 나타내는 작은 그래픽(*문자 모양*)만 있습니다.

스피드 버튼에는 세 가지 작동 모드가 있습니다. 스피드 버튼은 다음 작동을 수행할 수 있습니다.

- 일반적인 누름 버튼처럼 동작
- 클릭하여 켜고 끄기
- 라디오 버튼 집합처럼 동작

툴바에서 스피드 버튼을 구현하려면 다음을 수행하십시오.

- 패널에 스피드 버튼 추가
- 스피드 버튼의 문자 모양 할당
- 스피드 버튼의 초기 상태 설정
- 스피드 버튼 그룹 생성
- 버튼 토글 허용

패널에 스피드 버튼 추가

툴바 패널에 스피드 버튼을 추가하려면 컴포넌트 팔레트의 Additional 페이지에 있는 스피드 버튼 컴포넌트를 패널에 둡니다.

폼이 아닌 패널은 스피드 버튼을 "소유"하므로 패널을 이동하거나 숨기면 스피드 버튼도 이동하거나 숨겨집니다.

패널의 디폴트 높이는 41이며 스피드 버튼의 디폴트 높이는 25입니다. 각 버튼의 *Top* 속성을 8로 설정하면 세로 방향으로 가운데 맞춰집니다. 디폴트 그리드 설정은 스피드 버튼을 해당 세로 위치로 맞춥니다.

스피드 버튼의 문자 모양 할당

각 스피드 버튼에는 사용자에게 버튼의 기능을 알릴 수 있도록 *문자 모양*이라는 그래픽 이미지가 필요합니다. 스피드 버튼에 하나의 이미지만 제공하면 버튼이 해당 이미지를 조작하여 버튼이 눌렸는지, 눌리지 않았는지, 선택되었는지, 비활성화되었는지를 나타냅니다. 원한다면 상태에 대한 각각의 특정 이미지를 제공할 수도 있습니다.

런타임에 다른 문자 모양을 할당할 수 있지만 일반적으로 디자인 타임에 문자 모양을 스피드 버튼에 지정합니다.

다음과 같은 방법으로 디자인 타임에 문자 모양을 스피드 버튼에 할당합니다.

1 스피드 버튼을 선택합니다.

2 Object Inspector에서 *Glyph* 속성을 선택합니다.

3 *Glyph* 옆의 Value 열을 더블 클릭하여 Picture Editor를 열고 원하는 비트맵을 선택합니다.

스피드 버튼의 초기 상태 설정

사용자는 스피드 버튼의 모양을 통해 상태와 용도에 대한 정보를 얻을 수 있습니다. 스피드 버튼에는 캡션이 없으므로 사용자가 쉽게 알 수 있는 이미지를 제공해야 합니다.

표 8.6은 스피드 버튼의 모양을 변경하기 위해 설정할 수 있는 액션을 나열한 것입니다.

표 8.6 스피드 버튼의 모양 설정

스피드 버튼의 모양 변경	툴바의 속성 설정
눌려진 상태로 표시	*GroupIndex* 속성을 0이 아닌 값으로 설정하고 *Down* 속성을 **true**로 설정합니다.
비활성화 상태로 표시	*Enabled* 속성을 **false**로 설정합니다.
왼쪽 여백 표시	*Indent* 속성 값을 0보다 큰 값으로 설정합니다.

애플리케이션에 디폴트 드로잉 툴이 있을 경우, 애플리케이션을 시작할 때 툴바에서 해당 버튼이 눌려져 있어야 합니다. 이렇게 하려면 해당 *GroupIndex* 속성을 0이 아닌 값으로 설정하고 *Down* 속성을 **true**로 설정하십시오.

스피드 버튼 그룹 생성

일련의 스피드 버튼은 종종 동시에 선택할 수 없는 집합을 나타냅니다. 이러한 경우에는 버튼을 그룹으로 연결하여 그룹에서 버튼을 클릭할 때 그룹의 다른 항목이 팝업되게 해야 합니다.

임의 수의 스피드 버튼을 그룹으로 연결하려면 각 스피드 버튼의 *Group Index* 속성에 동일한 수를 할당합니다.

이렇게 하려면 그룹에서 원하는 버튼을 모두 선택한 다음 선택한 전체 그룹에서 *GroupIndex*를 고유 값으로 설정하는 것입니다.

버튼 토글 허용

그룹에서 눌려 있는 버튼을 클릭하여 팝업되게 하여 그룹에서 눌린 버튼이 하나도 없게 할 수 있습니다. 이러한 버튼을 *토글*이라고 합니다. *AllowAllUp*을 사용하여 토글로 작동되는 그룹화된 버튼을 만듭니다. 토글 버튼은 한 번 클릭하면 눌려지고 다시 클릭하면 팝업됩니다.

그룹화된 스피드 버튼을 토글로 만들려면 해당 *AllowAllUp* 속성을 **true**로 설정합니다.

그룹에 있는 스피드 버튼의 *AllowAllUp*을 **true**로 설정하면 그룹에 있는 모든 버튼의 속성 값이 자동으로 동일하게 설정됩니다. 따라서 그룹이 일반적인 그룹처럼 동작하고 한 번에 한 버튼만 눌려지지만 모든 버튼이 동시에 눌려지지 않게 할 수도 있습니다.

툴바 컴포넌트를 사용하여 툴바 추가

툴바 컴포넌트(*TToolBar*)는 패널 컴포넌트와 달리 버튼 관리와 디스플레이 기능을 제공합니다. 다음과 같은 방법으로 툴바 컴포넌트를 사용하여 폼에 툴바를 추가합니다.

1 컴포넌트 팔레트의 Win32 페이지에 있는 툴바 컴포넌트를 폼에 추가합니다. 툴바가 자동으로 폼의 위쪽에 정렬됩니다.

2 툴바에 툴 버튼 또는 다른 컨트롤을 추가합니다.

툴 버튼은 툴바 컴포넌트에서 동작하도록 디자인되어 있습니다. 스피드 버튼과 마찬가지로 툴바 버튼을 다음 기능을 수행할 수 있습니다.

- 일반적인 누름 버튼처럼 동작
- 클릭하여 켜고 끄기
- 라디오 버튼 집합처럼 동작

툴바에서 툴 버튼을 구현하려면 다음을 수행하십시오.

- 툴 버튼 추가
- 툴 버튼에 이미지 할당
- 툴 버튼의 모양 설정
- 툴 버튼 그룹 생성
- 툴 버튼 토글 허용

툴 버튼 추가

툴바에 툴 버튼을 추가하려면 툴바를 마우스 오른쪽 버튼으로 클릭한 후 New Button을 선택합니다.

툴바는 툴 버튼을 "소유"하므로 툴바를 이동하거나 숨기면 버튼도 이동하거나 숨겨집니다. 또한 툴바의 모든 툴 버튼은 자동으로 같은 높이와 너비로 유지됩니다. 컴포넌트 팔레트에서 다른 컨트롤을 툴바로 끌면 자동으로 동일한 높이로 유지됩니다. 툴바에서 가로 방향으로 맞지 않으면 컨트롤도 줄을 바꿔 새 줄을 시작됩니다.

툴 버튼에 이미지 할당

각 툴 버튼에는 런타임에 나타나는 이미지를 결정하는 *ImageIndex* 속성이 있습니다. 툴 버튼에 하나의 이미지만 제공하면 버튼이 해당 이미지를 조작하여 버튼이 비활성화되었는지 여부를 나타냅니다. 다음과 같은 방법으로 디자인 타임에 툴 버튼에 이미지를 할당할 수 있습니다.

1 버튼을 표시할 툴바를 선택합니다.

2 Object Inspector에서 *TImageList* 객체를 툴바의 *Images* 속성에 할당합니다. 이미지 리스트는 같은 크기의 아이콘이나 비트맵의 컬렉션입니다.

3 툴 버튼을 선택합니다.

4 Object Inspector에서 버튼에 할당할 이미지 리스트의 이미지에 해당하는 툴 버튼의 *ImageIndex* 속성에 정수를 할당합니다.

툴 버튼이 비활성화되고 마우스 포인터로 가리켜지면 각각의 이미지가 툴 버튼에 나타나게 지정할 수도 있습니다. 이렇게 하려면 각각의 이미지 리스트를 툴바의 *DisabledImages* 및 *HotImages* 속성에 할당하십시오.

툴 버튼 모양 및 초기 상태 설정

표 8.7은 스피드 버튼의 모양을 변경하기 위해 설정할 수 있는 액션을 나열한 것입니다.

표 8.7 툴 버튼 모양 설정

툴 버튼 모양	툴바의 속성 설정
눌려진 상태로 표시	(툴 버튼에서) *Style* 속성을 *tbsCheck*로, *Down* 속성을 **true**로 설정합니다.
비활성화 상태로 표시	*Enabled* 속성을 **false**로 설정합니다.
왼쪽 여백 표시	*Indent* 속성 값을 0보다 큰 값으로 설정합니다.
"팝업" 테두리가 있어 툴바를 투명하게 보이게 표시	*Flat* 속성을 **true**로 설정합니다.

참고 *TToolBar*의 *Flat* 속성을 사용하려면 버전 4.70 이상의 COMCTL32.DLL이 필요합니다.

특정 툴 버튼 다음에 새로운 행의 컨트롤을 놓으려면 행의 마지막에 나타날 툴 버튼을 선택하고 *Wrap* 속성을 **true**로 설정합니다.

툴바의 자동 줄 바꿈 기능을 해제하려면 툴바의 *Wrapable* 속성을 **false**로 설정합니다.

툴 버튼 그룹 생성

툴 버튼 그룹을 만들려면 연결할 버튼을 선택하고 해당 *Style* 속성을 *tbsCheck*로 설정한 다음 *Grouped* 속성을 **true**로 설정합니다. 그룹화된 툴 버튼을 선택하면 그룹의 다른 버튼이 팝업되므로 동시에 선택할 수 없는 집합을 쉽게 할 수 있습니다.

인접한 버튼의 시퀀스가 끊어지지 않고 *Style* 속성이 *tbsCheck*로, *Grouped*가 **true**로 설정되면 단일 그룹이 형성됩니다. 툴 버튼의 그룹을 해제하려면 다음 중 하나로 버튼을 분리하십시오.

- *Grouped* 속성이 **false**인 툴 버튼
- *Style* 속성이 *tbsCheck*로 설정되지 않은 툴 버튼. 툴바에 공백이나 구분줄을 만들려면 *Style* 속성이 *tbsSeparator* 또는 *tbsDivider*로 설정된 툴 버튼을 추가하십시오.
- 툴 버튼 이외의 다른 컨트롤

툴 버튼 토글 허용

*AllowAllUp*을 사용하여 토글로 작동하는 그룹화된 툴 버튼을 만듭니다. 토글 버튼은 한 번 클릭하면 눌려지고 다시 클릭하면 팝업됩니다. 그룹화된 툴 버튼을 토글로 만들려면 *AllowAllUp* 속성을 **true**로 설정합니다.

스피드 버튼과 마찬가지로, 그룹에 있는 툴 버튼의 *AllowAllUp*을 **true**로 설정하면 그룹에 있는 모든 버튼의 속성 값이 자동으로 동일하게 설정됩니다.

쿨바(cool bar) 컴포넌트 추가

참고 *TCoolBar* 컴포넌트를 사용하려면 버전 4.70 이상의 COMCTL32.DLL이 필요하며 CLX에서는 사용할 수 없습니다.

쿨바 컴포넌트(*TCoolBar*)는 *리바(rebar)*라고도 하는데, 독립적으로 이동하거나 크기를 조정할 수 있는 밴드에 윈도우 컨트롤을 표시합니다. 사용자는 각 밴드의 왼쪽에 있는 크기 조정 핸들을 끌어 밴드의 위치를 지정할 수 있습니다.

다음과 같은 방법으로 Windows 애플리케이션의 폼에 쿨바를 추가합니다.

1 컴포넌트 팔레트의 Win32 페이지에 있는 쿨바 컴포넌트를 폼에 추가합니다. 쿨바는 자동으로 폼의 위쪽에 정렬됩니다.

2 컴포넌트 팔레트에서 윈도우 컨트롤을 쿨바에 추가합니다.

*TWinControl*의 자손인 VCL 컴포넌트만 윈도우 컨트롤입니다. 레이블이나 스피드 버튼 등의 그래픽 컨트롤도 쿨바에 추가할 수 있으나 각각의 밴드에 나타나지는 않습니다.

쿨바(cool bar)의 모양 설정

쿨바 컴포넌트는 여러 가지 유용한 구성 옵션을 제공합니다. 표 8.8은 쿨 버튼의 모양을 변경하기 위해 설정할 수 있는 액션을 나열한 것입니다.

표 8.8 쿨(cool) 버튼 모양 설정

쿨바의 모양	툴바의 속성 설정
포함하는 밴드에 맞게 자동으로 크기 조정	*AutoSize* 속성을 **true**로 설정합니다.
밴드에서 동일한 높이 유지	*FixedSize* 속성을 **true**로 설정합니다.
가로 방향이 아닌 세로 방향으로 방향 다시 지정	*Vertical* 속성을 **true**로 설정합니다. 이렇게 변경하면 *FixedSize* 속성에도 영향을 줍니다.
런타임에 밴드의 *Text* 속성 표시 방지	*ShowText* 속성을 **false**로 설정합니다. 쿨바의 각 밴드에는 고유한 *Text* 속성이 있습니다.
쿨바 주위의 테두리 제거	*BandBorderStyle*을 *bsNone*으로 설정합니다.
런타임에 사용자의 밴드 테두리 변경 방지 (밴드 이동 및 크기 조정은 가능)	*FixedOrder*를 **true**로 설정합니다.
쿨바의 배경 이미지 생성	*Bitmap* 속성을 *TBitmap* 객체로 설정합니다.
밴드 왼쪽에 나타낼 이미지 리스트 선택	*Images* 속성을 *TImageList* 객체로 설정합니다.

각 밴드에 이미지를 할당하려면 쿨바를 선택하고 Object Inspector 에서 *Bands* 속성을 더블 클릭하십시오. 그런 다음 밴드를 선택하고 해당 *ImageIndex* 속성에 값을 할당합니다.

클릭 이벤트에 응답

사용자가 툴바의 버튼 등 컨트롤을 클릭하면 애플리케이션에서 이벤트 핸들러와 함께 응답할 수 있는 *OnClick* 이벤트를 만듭니다. *OnClick*은 버튼의 디폴트 이벤트이므로 디자인 타임에 버튼을 더블 클릭하여 이벤트에 스켈레톤 핸들러를 만들 수 있습니다.

툴 버튼에 메뉴 할당

툴 버튼(*TToolButton*)이 있는 툴바(*TToolBar*)를 사용하는 경우 특정 버튼과 메뉴를 연결할 수 있습니다.

1. 툴 버튼을 선택합니다.
2. Object Inspector에서 툴 버튼의 *DropDownMenu* 속성에 팝업 메뉴(*TPopupMenu*)를 할당합니다.

메뉴의 *AutoPopup* 속성을 **true**로 설정하는 경우 버튼을 누르면 메뉴가 자동으로 나타납니다.

숨겨진 툴바 추가

툴바는 항상 표시되지 않아도 됩니다. 실제로, 여러 툴바를 사용할 수 있지만 사용자가 필요할 때만 표시되게 하면 편리할 때가 많습니다. 여러 툴바가 있는 폼을 만든 다음 일부 또는 모두 숨길 수 있습니다.

다음과 같은 방법으로 숨겨진 툴바를 만듭니다.

1. 폼에 툴바, 쿨바 또는 패널 컴포넌트를 추가합니다.
2. 컴포넌트의 *Visible* 속성을 **false**로 설정합니다.

디자인 타임에는 툴바를 볼 수 있고 수정할 수 있을지라도 런타임에는 애플리케이션에서 특별히 툴바를 보이게 할 때까지는 숨겨진 채로 있습니다.

툴바 숨기기 및 표시

애플리케이션에 여러 툴바가 있지만 모든 툴바를 동시에 표시하여 혼잡해 보이지 않게 할 수 있습니다. 또는 사용자가 툴바 표시 여부를 결정하게 할 수도 있습니다. 모든 컴포넌트처럼 툴바도 런타임에 필요에 따라 표시하거나 숨길 수 있습니다.

런타임에 툴바를 숨기거나 표시하려면 해당 *Visible* 속성을 각각 **false** 또는 **true**로 설정하십시오. 일반적으로 특정 사용자 이벤트나 애플리케이션의 작업 모드 변경에 대한 응답으로 이 작업을 수행합니다. 이렇게 할 수 있도록 보통 각 툴바에는 닫기 버튼이 있습니다. 사용자가 이 버튼을 클릭하면 애플리케이션이 해당 툴바를 숨깁니다.

툴바의 토글 방법을 제공할 수도 있습니다. 다음 예제에서는 메인 툴바의 버튼을 통해 펜 툴바가 토글됩니다. 클릭할 때마다 버튼이 눌리거나 해제되므로 *OnClick* 이벤트 핸들러로 버튼이 눌렸는지의 여부에 따라 Pen 툴바를 표시하거나 숨길 수 있습니다.

```
void __fastcall TForm1::PenButtonClick(TObject *Sender)
{
  PenBar->Visible = PenButton->Down;
}
```

장

9

컨트롤 타입

컨트롤은 인터페이스를 디자인할 때 유용하게 사용할 수 있는 비주얼(visual) 컴포넌트입니다.

이 장에서는 텍스트 컨트롤, 입력 컨트롤, 버튼, 리스트 컨트롤, 그룹화 컨트롤, 디스플레이 컨트롤, 그리드, 값 리스트 에디터 및 그래픽 컨트롤 등 사용할 수 있는 여러 가지 컨트롤에 대해 설명합니다.

그래픽 컨트롤을 만들려면 54장, "그래픽 컴포넌트 생성"을 참조하고 이러한 컨트롤을 구현하는 방법에 대해서는 9장 "컨트롤 작업"을 참조하십시오.

텍스트 컨트롤

대부분의 애플리케이션에서는 텍스트 컨트롤을 사용하여 사용자에게 텍스트를 표시합니다. 다음 컨트롤을 사용할 수 있습니다.

- 사용자가 텍스트를 추가하게 할 수 있는 편집 컨트롤

컴포넌트	수행 작업
TEdit	한 줄의 텍스트를 편집합니다.
TMemo	여러 줄의 텍스트를 편집합니다.
TMaskEdit	우편 번호 또는 전화 번호 같은 특정 서식을 사용합니다.
TRichEdit	서식있는 텍스트 형식을 사용하여 여러 줄의 텍스트를 편집합니다 (VCL만 해당).

- 사용자가 텍스트를 추가할 수 없는 텍스트 보기 컨트롤 및 레이블

컴포넌트	수행 작업
TTextBrowser	사용자가 스크롤할 수 있는 텍스트 파일이나 간단한 HTML 페이지를 표시합니다.
TTextViewer	텍스트 파일이나 간단한 HTML 페이지를 표시합니다. 사용자는 페이지를 스크롤하거나 링크를 클릭하여 다른 페이지와 이미지를 볼 수 있습니다.
TLCDNumber	디지털 디스플레이 폼에 숫자 정보를 표시합니다.
TLabel	비윈도우 컨트롤에 텍스트를 표시합니다.
TStaticText	윈도우 컨트롤에 텍스트를 표시합니다.

편집 컨트롤

편집 컨트롤을 사용하면 사용자가 텍스트를 표시하고 텍스트를 입력할 수 있습니다. 이러한 목적으로 사용된 컨트롤 타입은 정보의 크기와 서식에 따라 다릅니다.

TEdit 와 *TMaskEdit* 는 정보를 입력할 수 있는 한 줄의 텍스트 에디트 박스가 포함된 간단한 편집 컨트롤입니다. 에디트 박스에 포커스가 있으면 깜빡이는 삽입 지점이 나타납니다.

문자열 값을 텍스트의 *Text* 속성에 할당하여 텍스트를 에디트 박스에 포함할 수 있습니다. *Font* 속성에 값을 할당하여 에디트 박스에 있는 텍스트 모양을 제어합니다. 글꼴의 서체, 크기, 색상 및 어트리뷰트(attribute)를 지정할 수 있습니다. 어트리뷰트는 에디트 박스의 모든 텍스트에 영향을 주며 개별 문자에만 적용될 수는 없습니다.

포함하는 글꼴의 크기에 따라 에디트 박스의 크기가 변경되도록 디자인할 수 있습니다. *AutoSize* 속성을 true로 설정하면 글꼴 크기에 따라 에디트 박스의 크기가 변합니다. *MaxLength* 속성에 값을 할당하여 에디트 박스의 문자 수를 제한할 수 있습니다.

*TMaskEdit*는 텍스트의 유효한 폼을 인코드하는 마스크에 대하여 입력된 텍스트를 검증하는 특수 편집 컨트롤입니다. 마스크는 사용자에게 표시되는 텍스트의 서식을 지정할 수도 있습니다.

TMemo 및 *TRichEdit* 를 통해 사용자는 여러 줄의 텍스트를 추가할 수 있습니다.

편집 컨트롤 속성

다음은 편집 컨트롤의 중요한 속성 중 일부를 보여 줍니다.

표 9.1 편집 컨트롤 속성

속성	설명
Text	에디트 박스 또는 메모 컨트롤에 나타나는 텍스트를 결정합니다.
Font	에디트 박스나 메모 컨트롤에 쓰인 텍스트의 어트리뷰트(attribute)를 제어합니다.
AutoSize	현재 선택된 글꼴에 맞게 에디트 박스의 높이를 동적으로 변경할 수 있게 합니다.
ReadOnly	사용자가 텍스트를 변경할 수 있는지 여부를 지정합니다.
MaxLength	간단한 편집 컨트롤의 문자 수를 제한합니다.
SelText	텍스트의 현재 선택된(강조된) 부분을 포함합니다.
SelStart, SelLength	선택한 텍스트 부분의 위치와 길이를 나타냅니다.

메모 및 리치 에디트(rich edit) 컨트롤

TMemo 및 *TRichEdit* 컨트롤 모두 여러 줄의 텍스트를 처리합니다.

VCL 리치 에디트(rich edit) 컨트롤은 VCL에만 있습니다.

*TMemo*는 여러 줄의 텍스트를 처리하는 다른 타입의 에디트 박스입니다. 메모 컨트롤의 줄은 에디트 박스의 오른쪽 테두리 밖으로 확장하거나 다음 줄로 줄 바꿈할 수 있습니다. *WordWrap* 속성을 사용하여 줄 바꿈 여부를 제어합니다.

*TRichEdit*는 서식있는 텍스트 서식, 인쇄, 검색 및 텍스트 드래그 앤 드롭을 지원하는 메모 컨트롤입니다. 이 컨트롤을 사용하여 글꼴 속성, 정렬, 탭, 들여쓰기 및 번호 매기기를 지정할 수 있습니다.

메모와 리치 에디트(rich edit) 컨트롤에는 모든 편집 컨트롤이 갖는 속성 외에 다음 속성이 포함됩니다.

- *Alignment*는 컴포넌트에서 텍스트를 정렬하는 방법(왼쪽, 오른쪽 또는 가운데)을 지정합니다.
- *Text* 속성은 컨트롤에 텍스트를 포함시킵니다. 애플리케이션은 *Modified* 속성을 확인하여 텍스트가 바뀌는지 여부를 알려줄 수 있습니다.
- *Lines*는 텍스트를 문자열 리스트로 포함합니다.
- *OEMConvert*는 텍스트를 입력할 때 ANSI에서 OEM으로 임시 변환되는지 여부를 결정합니다. 이 속성은 파일 이름을 확인하는 데 유용합니다(VCL만 해당).
- *WordWrap*은 텍스트가 오른쪽 여백을 만나면 줄 바꿈할지 여부를 결정합니다.
- *WantReturns*는 사용자가 텍스트에 하드 리턴을 삽입할 수 있는지 여부를 결정합니다.
- *WantTabs*는 사용자가 텍스트에 탭을 삽입할 수 있는지 여부를 결정합니다.
- *AutoSelect*는 컨트롤이 활성화될 때 텍스트가 자동으로 선택(강조)되는지 여부를 결정합니다.

런타임에 *SelectAll* 메소드를 사용하여 메모의 모든 텍스트를 선택할 수 있습니다.

텍스트 보기 컨트롤(CLX만 해당)

텍스트 보기 컨트롤은 텍스트를 표시할 수 있으나 읽기 전용입니다. *TTextViewer*는 사용자가 문서를 읽고 스크롤할 수 있는 간단한 뷰어 역할을 합니다. *TTextBrowser*를 사용하면 사용자가 링크를 클릭하여 다른 문서나 같은 문서의 다른 부분으로 이동할 수 있습니다. 방문한 문서는 열어 본 페이지 리스트에 저장되며 *Backward*, *Forward* 및 *Home* 메소드를 사용하여 다시 볼 수 있습니다. *TTextViewer* 및 *TTextBrowser*는 HTML 기반의 텍스트를 표시하거나 HTML 기반의 도움말 시스템을 구현하는 데 가장 유용합니다.

*TTextBrowser*에는 *TTextViewer*의 속성 외에도 *Factory* 속성이 있습니다. *Factory*는 포함된 이미지용 파일 타입을 결정하는 데 사용되는 MIME 팩토리 객체를 결정합니다. 예를 들어, .txt, .html 및 .xml과 같은 파일 이름 확장자를 MIME 타입에 연결하고 팩토리에서 이 데이터를 컨트롤로 로드하게 할 수 있습니다.

FileName 속성을 사용하여 .html과 같은 텍스트 파일을 추가하여 런타임에 컨트롤에 나타나게 할 수 있습니다.

레이블

레이블(*TLabel* 및 *TStaticText* (VCL만 해당))은 텍스트를 표시하며 항상 다른 컨트롤 옆에 위치합니다. 에디트 박스와 같은 다른 컴포넌트를 식별하거나 주석을 표시하거나 또는 텍스트를 폼에 포함하려고 하는 경우 폼에 레이블을 둡니다. 표준 레이블 컴포넌트인 *TLabel*은 비윈도우 컨트롤(CLX에서는 widget 기반)이므로 포커스를 받을 수 없습니다. 윈도우 핸들에 레이블이 필요하면 *TStaticText*를 대신 사용하십시오.

Label 속성에는 다음이 포함됩니다.

- *Caption*은 레이블의 텍스트 문자열을 포함합니다.
- *Font*, *Color* 및 다른 속성은 레이블의 모양을 결정합니다. 각 레이블은 글꼴, 크기 및 색상을 하나만 사용할 수 있습니다.
- *FocusControl*은 레이블을 폼에 있는 다른 컨트롤에 연결합니다. *Caption*에 가속키가 있으면 사용자가 가속키를 누를 때 *FocusControl*에 의해 지정된 컨트롤이 포커스를 받습니다.
- *ShowAccelChar*는 레이블이 밑줄이 있는 가속 문자를 표시할 수 있는지 결정합니다. *ShowAccelChar*가 *true*이면 앰퍼샌드(&) 뒤에 오는 모든 글자에 밑줄이 추가되므로 가속키를 사용할 수 있습니다.
- *Transparent*는 그래픽과 같은 레이블 아래의 항목이 보이는지 여부를 결정합니다.

레이블은 보통 애플리케이션 사용자가 변경할 수 없는 읽기 전용 정적 텍스트를 표시합니다. *Caption* 속성에 새 값을 지정하여 애플리케이션은 실행 도중에 텍스트를 변경할 수 있습니다. 사용자가 스크롤하거나 편집할 수 있는 폼에 텍스트를 추가하려면 *TEdit*를 사용하십시오.

특화된 입력 컨트롤

다음 컴포넌트는 입력을 캡처하는 추가 방법을 제공합니다.

컴포넌트	수행 작업
TScrollBar	연속적인 범위의 값을 선택합니다.
TTrackBar	스크롤 막대보다 시각적으로 더 효과적으로 연속 범위의 값을 선택합니다.
TUpDown	편집 컴포넌트에 연결된 스피너(spinner)에서 값을 선택합니다(VCL만 해당).
THotKey	*Ctrl*/*Shift*/*Alt* 키를 누릅니다(VCL만 해당).
TSpinEdit	스피너(spinner) widget에서 값을 선택합니다(CLX만 해당).

스크롤 막대

스크롤 막대 컴포넌트는 윈도우, 폼 또는 다른 컨트롤의 내용을 스크롤하는 데 사용할 수 있는 스크롤 막대를 만듭니다. *OnScroll* 이벤트 핸들러에서, 사용자가 스크롤 막대를 움직일 때 컨트롤의 동작을 결정하는 코드를 작성합니다.

많은 비주얼(visual) 컴포넌트에는 자체적으로 스크롤 막대가 포함되어 있어 추가 코딩이 필요 없습니다. 따라서 스크롤 막대 컴포넌트는 자주 사용되지 않습니다. 예를 들어, *TForm*에는 자동으로 폼의 스크롤 막대를 만드는 *VertScrollBar*와 *HorzScrollBar* 속성이 있습니다. 폼 내에서 스크롤할 수 있는 영역을 따로 만들려면 *TScrollBox*를 사용하십시오.

트랙 표시줄

트랙 표시줄은 연속적인 범위의 정수 값을 설정할 수 있으므로색상, 부피 및 밝기 같은 속성을 제어하는 데 유용합니다. 사용자는 슬라이드 지시자를 특정 위치로 끌어 놓거나 표시줄 내부를 클릭하여 슬라이드 지시자를 옮깁니다.

- *Max* 속성과 *Min* 속성을 사용하여 트랙 표시줄 범위의 시작과 끝을 설정합니다.
- *SelEnd*와 *SelStart*를 사용하여 선택 범위를 강조 표시합니다. 그림 9.1을 참조하십시오.
- *Orientation* 속성은 트랙 표시줄이 수직인지 수평인지 결정합니다.
- 디폴트로, 트랙 표시줄에는 아래쪽을 따라 하나의 눈금 행이 있습니다. *TickMarks* 속성을 사용하여 눈금 위치를 변경합니다. 눈금 간격을 제어하려면 *TickStyle* 속성과 *SetTick* 메소드를 사용하십시오.

그림 9.1 트랙 표시줄 컴포넌트의 세 가지 뷰



- *Position*은 트랙 표시줄의 디폴트 위치를 설정하고 런타임 시 위치를 추적합니다.
- 디폴트로, 사용자는 위쪽 화살표와 아래쪽 화살표 키를 눌러서 눈금을 위나 아래로 움직일 수 있습니다. *LineSize*를 설정하여 눈금의 이동 단위를 변경합니다.
- *PageSize*를 설정하여 사용자가 *Page Up* 키와 *Page Down* 키를 누를 때 눈금의 이동 정도를 결정할 수 있습니다.

업다운(up-down) 컨트롤(VCL만 해당)

업다운(up-down) 컨트롤(*TUpDown*)은 사용자가 고정된 증분으로 정수 값을 변경할 수 있는 한 쌍의 화살표 버튼으로 구성되어 있습니다. 현재 값은 *Position* 속성에 의해 지정되고, 기본값이 1인 증분은 *Increment* 속성에 의해 지정됩니다. *Associate* 속성을 사용하여 편집 컨트롤 같은 다른 컴포넌트를 업다운(up-down) 컨트롤에 추가합니다.

스핀 편집 컨트롤(CLX만 해당)

스핀 편집 컨트롤(*TSpinEdit*)은 업다운 widget, 작은 화살표 widget 또는 스핀 버튼이라고도 합니다. 이 컨트롤을 사용하여 위쪽 또는 아래쪽 화살표 버튼을 클릭하여 현재 표시된 값을 증가 또는 감소시키거나 스핀 박스에 직접 값을 입력하여, 애플리케이션 사용자가 고정된 증분으로 정수 값을 변경할 수 있습니다.

현재 값은 *Value* 속성에 의해 지정되고, 기본값이 1인 증분은 *Increment* 속성에 의해 지정됩니다.

단축키 컨트롤(VCL만 해당)

단축키 컴포넌트(*THotKey*)를 사용하여 포커스를 다른 컨트롤로 전달하는 단축키를 할당합니다. *HotKey* 속성에는 현재 키 조합이 포함되어 있고 *Modifiers* 속성은 *HotKey*에 사용할 수 있는 키를 결정합니다.

단축키 컴포넌트는 메뉴 항목의 *ShortCut* 속성으로 할당될 수 있습니다. 그런 다음 사용자가 *HotKey* 및 *Modifiers* 속성에서 지정한 키 조합을 입력하면 Windows가 메뉴 항목을 활성화합니다.

스플리터 컨트롤

정렬된 컨트롤 사이에 있는 스플리터(*TSplitter*)를 통해 사용자는 컨트롤의 크기를 조정할 수 있습니다. 패널이나 그룹 박스와 같은 컴포넌트와 함께 스플리터를 사용하면 각 창에 여러 개의 컨트롤이 있는 여러 개의 창으로 폼을 나눌 수 있습니다.

패널이나 다른 컨트롤을 폼에 추가한 후에 컨트롤과 동일한 정렬을 가진 스플리터를 추가합니다. 마지막 컨트롤은 클라이언트 정렬이 되어야 하므로, 다른 컨트롤의 크기가 조정되면 나머지 공간을 모두 채웁니다. 예를 들어, 패널을 폼의 왼쪽 가장자리에 둘 수 있고, 패널의 *Alignment*를 *alLeft*로 설정한 다음 *alLeft*로 정렬된 스플리터를 패널의 오른쪽에 두고, 마지막으로 *alLeft* 또는 *alClient*로 정렬된 다른 패널을 스플리터의 오른쪽에 둡니다.

*MinSize*를 설정하여 이웃 컨트롤의 크기를 조정할 때 스플리터가 남겨두어야 하는 최소 크기를 지정합니다. *Beveled*를 *true*로 설정하여 스플리터의 가장자리를 3차원 모양으로 만들 수 있습니다.

버튼 및 이와 유사한 컨트롤

메뉴와 함께 버튼은 애플리케이션에서 명령을 호출하는 가장 일반적인 방법을 제공합니다. C++Builder는 다음과 같은 몇 가지 버튼 유형의 컨트롤을 제공합니다.

컴포넌트	수행 작업
TButton	텍스트가 있는 버튼으로 명령 선택 사항을 제공합니다.
TBitBtn	텍스트와 문자 모양이 있는 버튼으로 명령 선택 사항을 제공합니다.
TSpeedButton	그룹화된 툴바 버튼을 만듭니다.
TCheckBox	설정/해제 옵션을 제공합니다.
TRadioButton	동시에 설정할 수 없는 선택 사항들의 집합을 제공합니다.
TToolBar	행에서 툴 버튼 및 기타 컨트롤을 정렬하고 크기와 위치를 자동으로 조정합니다.
TCoolBar	이동 및 크기 조정이 가능한 밴드 내에서 윈도우 컨트롤 집합을 나타냅니다(VCL만 해당).

버튼 컨트롤

마우스로 버튼 컨트롤을 클릭하여 액션을 초기화합니다. 버튼에는 액션을 나타내는 텍스트 레이블이 있습니다. 이 텍스트는 *Caption* 속성에 문자열 값을 할당하여 지정합니다. 또한 단축키를 눌러서 대부분의 버튼을 선택할 수 있습니다. 단축키는 버튼에서 밑줄 친 문자로 나타납니다.

버튼 컨트롤을 클릭하여 액션을 초기화합니다. *OnClick* 이벤트 핸들러를 만들어 *TButton* 컴포넌트에 액션을 할당할 수 있습니다. 디자인 타임에 버튼을 더블 클릭하여 코드 에디터에 있는 버튼의 *OnClick* 이벤트 핸들러로 이동할 수 있습니다.

- *Esc* 키를 누를 때 버튼의 *OnClick* 이벤트가 실행되도록 하려면 *Cancel*을 *true*로 설정합니다.
- *Enter* 키를 누를 때 버튼의 *OnClick* 이벤트가 실행되도록 하려면 *Default*를 *true*로 설정합니다.

비트맵 버튼

비트맵 버튼(*BitBtn*)은 비트맵 이미지가 있는 버튼 컨트롤입니다.

- 버튼에 표시할 비트맵을 선택하려면 *Glyph* 속성을 설정합니다.
- *Kind*를 사용하여 문자 모양과 디폴트 동작을 지닌 버튼을 자동으로 구성합니다.
- 디폴트로, 문자 모양은 텍스트의 왼쪽에 나타납니다. 문자 모양을 이동하려면 *Layout* 속성을 사용합니다.
- 문자 모양과 텍스트는 자동으로 버튼의 가운데에 표시됩니다.위치를 이동하려면 *Margin* 속성을 사용합니다. *Margin*은 이미지의 가장자리와 버튼의 가장자리 사이의 픽셀 수를 결정합니다.
- 디폴트로, 이미지와 텍스트는 4개의 픽셀로 분리됩니다. *Spacing*을 사용하여 간격을 늘리거나 줄입니다.
- 비트맵 버튼에는 업, 다운 및 누르고 있음(held down) 세 가지 상태가 있습니다. *NumGlyphs* 속성을 3으로 설정하면 각 상태에 따라 다른 비트맵이 표시됩니다.

스피드 버튼

스피드 버튼은 보통 이미지를 가지고 있고 그룹으로 묶어서 사용합니다. 일반적으로 스피드 버튼을 패널과 함께 사용하여 툴바를 만듭니다.

- 스피드 버튼이 그룹으로 동작하게 하려면, 모든 버튼의 *GroupIndex* 속성에 0이 아닌 동일한 값을 할당합니다.
- 디폴트로, 스피드 버튼은 선택하지 않은 상태인 up 상태로 나타납니다. 스피드 버튼이 선택된 상태를 초기값으로 지정하려면 *Down* 속성을 *true*로 설정합니다.
- *AllowAllUp*이 *true*인 경우에 그룹의 모든 스피드 버튼은 선택되지 않은 상태가 될 수 있습니다. 버튼 그룹이 라디오 그룹처럼 동작하게 하려면 *AllowAllUp*을 *false*로 설정합니다.

스피드 버튼에 대한 자세한 내용은 8-43페이지의 "패널 컴포넌트를 사용하여 툴바 추가" 및 8-16페이지의 "툴바와 메뉴에 대한 액션 구성" 단원을 참조하십시오.

체크 박스

체크 박스는 사용자가 선택 또는 선택 해제할 수 있는 토글입니다. 선택하면 체크 박스에 체크 표시가 나타나고 선택하지 않으면 비어 있는 상태가 됩니다. *TCheckBox*를 사용하여 체크 박스를 만듭니다.

- 디폴트로 체크 박스를 선택 표시되게 하려면 *Checked*를 *true*로 설정합니다.
- *AllowGrayed*를 *true*로 설정하여 체크 박스에 선택, 선택 해제 및 비활성이라는 세 가지의 상태를 부여합니다.
- *State* 속성은 체크 박스가 선택되었는지(*cbChecked*), 선택되지 않았는지(*cbUnchecked*) 또는 비활성인지(*cbGrayed*) 여부를 나타냅니다.

참고 체크 박스 컨트롤은 두 가지 바이너리 상태 중 하나를 표시합니다. 다른 설정으로 체크 박스의 현재 값을 결정할 수 없는 경우 미결정 상태가 사용됩니다.

라디오 버튼

라디오 버튼은 동시에 설정할 수 없는 선택 집합을 나타냅니다. *TRadioButton*을 사용하여 라디오 버튼을 개별적으로 만들거나 *라디오 그룹* 컴포넌트(*TRadioGroup*)를 사용하여 자동으로 라디오 버튼을 그룹으로 정렬합니다. 라디오 버튼을 그룹화하여 사용자가 제한된 선택 집합 내에서 하나를 선택하게 할 수 있습니다. 자세한 내용은 9-12페이지의 "그룹화 컨트롤"을 참조하십시오.

선택된 라디오 버튼은 가운데가 채워진 원으로 나타나고 선택되지 않은 경우에는 비어 있는 원으로 나타납니다. Checked 속성에 *true* 또는 *false* 값을 할당하여 라디오 버튼의 시각적 상태를 변경할 수 있습니다.

툴바

툴바는 비주얼(visual) 컨트롤을 정렬하고 관리하는 쉬운 방법을 제공합니다. 패널 컴포넌트와 스피드 버튼으로 툴바를 만들거나 *TToolBar* 컴포넌트를 사용한 다음 마우스 오른쪽 버튼을 클릭하여 New Button을 선택하여 툴바에 툴 버튼을 추가할 수 있습니다.

TToolBar 컴포넌트에는 여러 가지 장점이 있습니다. 다른 컨트롤은 상대적인 위치와 높이를 유지하는 반면 툴바에 있는 버튼은 자동으로 일정한 크기와 간격을 유지하고, 컨트롤들이 수평 정렬되지 않으면 자동으로 랩어라운드되어 새 줄에서 시작하며, *TToolBar*는 투명도, 팝업 테두리 및 공간과 스플리터 등 표시 옵션을 제공하여 컨트롤을 그룹화합니다.

*액션 리스트*나 *액션 밴드*를 사용하여 툴바와 메뉴에 있는 액션 집합을 사용할 수 있습니다. 버튼과 툴바로 액션 리스트를 사용하는 데 대한 자세한 내용은 8-23페이지의 "액션 리스트 사용"을 참조하십시오.

툴바는 에디트 박스, 콤보 박스 등과 같은 다른 컨트롤의 부모가 될 수도 있습니다.

쿨바(cool bar)(VCL만 해당)

쿨바에는 독립적으로 이동하거나 크기를 조정할 수 있는 자식 컨트롤이 있습니다. 컨트롤은 각 밴드에 있습니다. 사용자는 크기 조정 핸들을 각 밴드의 왼쪽으로 끌어 크기를 조정할 수 있습니다.

쿨바를 사용하려면 디자인 타임과 런타임 모두에서 버전 4.70 이상의 COMCTL32.DLL(보통 Windows\System 또는 Windows\System32 디렉토리에 있음)이 필요합니다. 쿨바는 크로스 플랫폼 애플리케이션에서는 사용할 수 없습니다.

- *Bands* 속성에는 *TCoolBand* 객체 컬렉션이 포함되어 있습니다. 디자인 타임 시 Bands Editor로 밴드를 추가하거나 제거하거나 수정할 수 있습니다. 밴드 에디터를 열려면 Object Inspector에서 *Bands* 속성을 선택한 다음 오른쪽의 Value 열을 더블 클릭하거나 생략 부호(...) 버튼을 클릭하십시오. 팔레트에서 새 윈도우 컨트롤을 추가하여 밴드를 만들 수도 있습니다.
- *FixedOrder* 속성은 사용자가 밴드를 재정렬할 수 있는지 여부를 결정합니다.
- *FixedSize* 속성은 밴드에서 일정한 높이가 유지되는지 여부를 지정합니다.

리스트 컨트롤

리스트는 선택할 항목들을 사용자에게 보여 줍니다. 리스트를 표시하는 일부 컴포넌트는 다음과 같습니다.

컴포넌트	표시 내용
TListBox	텍스트 문자열 리스트
TCheckListBox	각 항목 앞에 체크 박스가 있는 리스트
TComboBox	스크롤할 수 있는 드롭다운 리스트가 있는 에디트 박스
TTreeView	계층 리스트
TListView	선택적 아이콘, 열 및 헤더가 있는 끌어 놓을 수 있는 항목 리스트
TIconView *(CLX 만 해당)*	작거나 큰 아이콘으로 표시되는 행과 열 형태의 항목 또는 데이터 리스트
TDateTimePicker	날짜나 시간을 입력하기 위한 리스트 박스(VCL만 해당)
TMonthCalendar	날짜를 선택하기 위한 달력(VCL만 해당)

넌비주얼(nonvisual) *TStringList*와 *TImageList* 컴포넌트를 사용하여 문자열과 이미지 집합을 관리합니다. 문자열 리스트에 대한 자세한 내용은 4-15페이지의 "문자열 리스트 작업"을 참조하십시오.

리스트 박스와 체크 리스트 박스

리스트 박스(*TListBox*)와 체크 리스트 박스는 사용자가 항목을 선택할 수 있는 리스트를 표시합니다.

- *Items*는 *TStrings* 객체를 사용하여 컨트롤을 값으로 채웁니다.
- *ItemIndex*는 리스트에서 선택된 항목을 나타냅니다.
- *MultiSelect*는 한 번에 둘 이상의 항목을 선택할 수 있는지 여부를 지정합니다.
- *Sorted*는 리스트가 알파벳 순서로 정렬되는지 여부를 결정합니다.
- *Columns*는 리스트 컨트롤에 있는 열의 수를 지정합니다.

- *IntegralHeight*는 리스트 박스에 세로 공간을 완전히 채우는 항목만 표시할지 여부를 지정합니다(VCL만 해당).
- *ItemHeight*는 각 항목의 높이를 픽셀 단위로 지정합니다. *Style* 속성은 *ItemHeight*를 무시하도록 할 수 있습니다.
- *Style* 속성은 리스트 컨트롤이 항목을 표시하는 방법을 결정합니다. 디폴트로, 항목은 문자열로 표시됩니다. *Style* 값을 변경하여 항목을 그래픽으로 또는 변하는 높이로 항목을 표시하는 *owner-draw* 리스트 박스를 만들 수 있습니다. owner-draw 컨트롤에 대한 자세한 내용은 6-11페이지의 "컨트롤에 그래픽 추가"를 참조하십시오.

다음과 같은 방법으로 간단한 리스트 박스를 만듭니다.

1. 프로젝트 내에서 컴포넌트 팔레트의 리스트 박스 컴포넌트를 폼에 가져다 놓습니다.
2. 리스트 박스의 크기를 정하고 필요하면 리스트 박스를 정렬합니다.
3. *Items* 속성의 오른쪽을 더블 클릭하거나 생략 부호(...) 버튼을 선택하여 String List Editor를 표시합니다.
4. String List Editor 를 사용하여 리스트 박스의 내용에 대해 줄로 정렬된 자유 형식 텍스트를 입력합니다.
5. OK를 선택합니다.

사용자가 리스트 박스에서 여러 항목을 선택하게 하려면 *ExtendedSelect* 속성과 *MultiSelect* 속성을 사용할 수 있습니다.

콤보 박스

콤보 박스(*TComboBox*)는 에디트 박스를 스크롤 가능한 리스트와 결합합니다. 사용자가 입력하거나 리스트에서 선택하여 데이터를 컨트롤에 입력하면 *Text* 속성 값이 변경됩니다. *AutoComplete*를 사용 가능으로 설정하면 애플리케이션은 사용자가 데이터를 입력할 때 리스트에서 가장 일치하는 것을 찾아서 표시합니다.

콤보 박스에는 표준, 드롭다운(기본값) 및 드롭다운 리스트 등 세 가지 타입이 있습니다.

- *Style* 속성을 사용하여 필요한 콤보 박스 타입을 선택합니다.
- 드롭다운 리스트가 있는 에디트 박스를 사용하려면 *csDropDown*을 사용하십시오. *csDropDownList*를 사용하여 에디트 박스를 읽기 전용으로 만듭니다. 그러면 사용자가 리스트에서만 선택하게 할 수 있습니다. *DropDownCount* 속성을 설정하여 리스트에 표시된 항목의 수를 변경합니다.
- *csSimple*을 사용하여 닫히지 않은 고정된 리스트를 가진 콤보 박스를 만듭니다. 콤보 박스의 크기를 조정하여 리스트 항목이 모두 표시되는지 확인하십시오.
- *csOwnerDrawFixed* 또는 *csOwnerDrawVariable*을 사용하여 항목을 그래픽으로 또는 변하는 높이로 항목을 표시하는 *owner-draw* 콤보 박스를 만듭니다. owner-draw 컨트롤에 대한 자세한 내용은 6-11페이지의 "컨트롤에 그래픽 추가"를 참조하십시오.

런타임 시 CLX 콤보 박스는 VCL 콤보 박스와 다르게 동작합니다. CLX(VCL 콤보 박스 아님)에서는 콤보 박스의 편집 필드에 텍스트를 입력하고 Enter 키를 눌러서 드롭다운에 항목을 추가할 수 있습니다. *InsertMode*를 ciNone으로 설정하여 이 기능을 해제할 수 있습니다. 콤보 박스의 리스트에 비어 있는(문자열이 아닌) 항목을 추가할 수도 있습니다. 또한 아래쪽 화살표 키를 계속 누르면 콤보 박스 리스트의 마지막 항목에서 멈추지 않고 위에서부터 다시 반복하여 스크롤됩니다.

트리 뷰

트리 뷰*(TTreeView)*는 들여쓰기로 항목을 표시합니다. 컨트롤에는 노드를 확장하거나 축소시키는 버튼이 있습니다. 사용자는 항목의 텍스트 레이블이 있는 아이콘을 포함할 수 있고, 노드가 확장 또는 축소되는지에 따라 각각 다른 아이콘을 표시할 수 있습니다. 항목에 대한 상태 정보를 반영하는 체크 박스와 같은 그래픽을 포함할 수도 있습니다.

- *Indent*는 항목을 부모로부터 수평으로 분리하는 픽셀 수를 설정합니다.
- *ShowButtons*는 '+'와 '-' 버튼을 표시하여 항목을 확장할 수 있는지 여부를 나타냅니다.
- *ShowLines*는 연결선을 표시하여 계층 관계를 표시합니다(VCL만 해당).
- *ShowRoot*는 최상위 레벨 항목을 연결하는 선을 표시할지 여부를 결정합니다(VCL만 해당).

디자인 타임에 항목을 트리 뷰 컨트롤에 추가하려면 컨트롤을 더블 클릭하여 TreeView Items Editor를 표시합니다. 추가한 항목은 *Items* 속성 값이 됩니다. 런타임 시 *Items* 속성의 메소드를 사용하여 *TTreeNodes* 타입의 객체인 항목을 변경할 수 있습니다. *TTreeNodes*에는 트리 뷰에 항목을 추가하고, 삭제하고, 탐색하기 위한 메소드가 있습니다.

트리 뷰는 vsReport 모드에 있는 리스트 뷰와 유사한 열과 하위 항목을 표시할 수 있습니다.

리스트 뷰

*TListView*를 사용하여 만든 리스트 뷰는 다양한 형식으로 리스트를 표시합니다. 다음과 같이 *ViewStyle* 속성을 사용하여 원하는 리스트의 종류를 선택합니다.

- *vsIcon* 및 *vsSmallIcon*은 각 항목을 레이블이 있는 아이콘으로 표시합니다. 사용자는 리스트 뷰 내에서 항목을 끌 수 있습니다(VCL만 해당).
- *vsList*는 끌어 놓을 수 없는, 레이블이 있는 아이콘으로 항목을 표시합니다.
- *vsReport*는 각각의 행에 항목을 표시하며 각 열에 해당 정보를 제공합니다. 가장 왼쪽에 있는 열에는 작은 아이콘과 레이블이 있고, 다음 열에는 애플리케이션에 의해 지정된 하위 항목이 있습니다. *ShowColumnHeaders* 속성을 사용하여 열에 대한 헤더를 표시합니다.

Date-Time Picker 및 Month Calendar(VCL만 해당)

DateTimePicker 컴포넌트는 날짜나 시간을 입력할 수 있는 리스트 박스를 표시하는 반면 MonthCalendar 컴포넌트는 날짜나 날짜 범위를 입력할 수 있는 달력을 표시합니다. 이러한 컴포넌트를 사용하려면 디자인 타임과 런타임 모두에서 버전 4.70 이상의 COMCTL32.DLL (보통 Windows\System 또는 Windows\System32 디렉토리에 있음)이 필요합니다. 크로스 플랫폼 애플리케이션에서는 사용할 수 없습니다.

그룹화 컨트롤

그래픽 인터페이스에서 관련 컨트롤과 정보를 그룹화하면 사용하기가 쉽습니다. C++Builder는 컴포넌트를 그룹화하기 위해 다음과 같은 여러 컴포넌트를 제공합니다.

컴포넌트	수행 작업
TGroupBox	제목이 있는 표준 그룹 박스
TRadioGroup	단순한 라디오 버튼 그룹
TPanel	시각적으로 더 유연한 컨트롤 그룹
TScrollBox	컨트롤을 포함하는 스크롤 가능한 영역
TTabControl	동시에 설정할 수 없는 노트북 스타일 탭 집합
TPageControl	탭마다 페이지를 가지는 동시에 설정할 수 없는 노트북 스타일 탭의 집합 (각 페이지에는 다른 컨트롤을 포함)
THeaderControl	크기 조정 가능한 열 헤더

그룹 박스와 라디오 그룹

그룹 박스(*TGroupBox*)는 폼에서 관련 있는 컨트롤을 정렬합니다. 그룹화된 컨트롤 중에 가장 일반적인 것은 라디오 버튼입니다. 폼에 그룹 박스를 추가한 다음 컴포넌트 팔레트에서 컴포넌트를 선택하여 그룹 박스에 추가합니다. *Caption* 속성에는 런타임에 그룹 박스에 레이블을 표시하는 텍스트를 포함합니다.

라디오 그룹 컴포넌트(*TRadioGroup*)를 사용하면 간편하게 라디오 버튼을 조합하여 함께 작동하도록 만들 수 있습니다. 라디오 버튼을 라디오 그룹에 추가하려면 Object Inspector에서 *Items* 속성을 편집하십시오. 즉, *Items*에 있는 각 문자열은 캡션 문자열이 있는 그룹 박스에 라디오 버튼을 나타나게 합니다. *ItemIndex* 속성 값은 현재 선택된 라디오 버튼을 지정합니다. *Columns* 속성 값을 설정하여 단일 열 또는 여러 열에서 라디오 버튼을 표시합니다. 버튼을 재배치하려면 라디오 그룹 컴포넌트의 크기를 조정합니다.

패널

TPanel 컴포넌트는 다른 컨트롤에 대한 일반적인 컨테이너를 제공합니다. 패널은 보통 폼에서 컴포넌트를 시각적으로 그룹화하는 데 사용됩니다. 패널을 폼과 함께 정렬하여 폼의 크기를 조정할 때 상대적으로 동일한 위치를 유지하게 할 수 있습니다. *BorderWidth* 속성은 패널 주위의 테두리 너비를 픽셀 단위로 지정합니다.

또한 다른 컨트롤을 패널에 놓고 *Align* 속성을 사용하여 폼에 있는 그룹의 모든 컨트롤 위치가 적절한지 확인할 수 있습니다. 폼의 크기가 변경되더라도 패널의 위치가 유지되도록 패널을 alTop 정렬할 수 있습니다.

BevelOuter 속성과 *BevelInner* 속성을 사용하여 패널의 모양을 올라가거나 내려간 모양으로 변경할 수 있습니다. 이러한 속성 값들을 변경하여 다양한 비주얼 3차원 효과를 낼 수 있습니다. 단순히 올라가거나 내려간 3 차원 상태를 원한다면 대신 리소스를 덜 사용하는 *TBevel* 컨트롤을 사용하면 됩니다.

또한 하나 이상의 패널을 사용하여 다양한 상태 표시줄 또는 정보 표시 영역을 만들 수 있습니다.

스크롤 박스

스크롤 박스(*TScrollBox*)는 폼 내에 스크롤 영역을 만듭니다. 경우에 따라 애플리케이션은 지정된 영역에 비해 더 많은 정보를 표시해야 합니다. 리스트 박스, 메모 및 폼 자체 등 일부 컨트롤은 자동으로 내용을 스크롤할 수 있습니다.

스크롤 박스의 다른 용도는 윈도우에서 다양한 스크롤 영역(뷰)을 만드는 것입니다. 뷰는 상용 워드 프로세서, 스프레트시트 및 프로젝트 관리 애플리케이션에서 일반적입니다. 스크롤 박스는 폼의 임의 스크롤 하위 영역을 정의할 수 있는 추가 유연성을 제공합니다.

패널이나 그룹 박스와 마찬가지로 스크롤 박스는 *TButton* 객체 및 *TCheckBox* 객체와 같은 다른 컨트롤을 포함합니다. 그러나 스크롤 박스는 일반적으로 보이지 않습니다. 스크롤 박스에 있는 컨트롤이 가시적인 영역보다 크면 스크롤 박스에 자동으로 스크롤 막대가 표시됩니다.

스크롤 박스의 다른 용도는 툴바 또는 상태 표시줄(*TPanel* 컴포넌트)과 같은 윈도우 영역에서 스크롤을 제한하는 것입니다. 툴바와 상태 표시줄이 스크롤되는 것을 방지하려면, 스크롤 막대를 숨긴 다음 스크롤 박스를 툴바와 상태 표시줄 사이에 있는 윈도우의 클라이언트 영역에 둡니다. 스크롤 박스에 연결된 스크롤 막대는 윈도우에 속하는 것처럼 보이지만 스크롤 박스 내의 영역만 스크롤합니다.

탭 컨트롤

탭 컨트롤 컴포넌트(*TTabControl*)는 노트북 칸막이처럼 보이는 탭 집합을 만듭니다. Object Inspector에서 *Tabs* 속성을 편집하여 탭을 만들 수 있는데 *Tabs*에 있는 각 문자열은 탭을 나타냅니다. 탭 컨트롤은 컴포넌트 집합 하나를 가지는 단일 패널입니다. 탭을 클릭할 때 컨트롤의 모양을 변경하려면 *OnChange* 이벤트 핸들러를 작성해야 합니다. 멀티페이지 다이얼로그 박스를 만들려면 탭 컨트롤 대신 페이지 컨트롤을 사용합니다.

페이지 컨트롤

페이지 컨트롤 컴포넌트(*TPageControl*)는 멀티페이지 다이얼로그 박스에 적합한 페이지 집합입니다. 페이지 컨트롤은 *TTabSheet* 객체인 여러 개의 오버랩된 페이지를 표시합니다. 컨트롤의 상단에 있는 탭을 클릭하여 사용자 인터페이스에서 페이지를 선택합니다.

디자인 타임에 페이지 컨트롤에서 새 페이지를 만들려면 컨트롤을 마우스 오른쪽 버튼으로 클릭하고 New Page를 선택합니다. 페이지에 대한 객체를 만들고 해당 *Page Control* 속성을 설정하여 런타임에 새 페이지를 추가합니다.

```
TTabSheet *pTabSheet = new TTabSheet(PageControl1);
pTabSheet->PageControl = PageControl1;
```

활성 페이지에 액세스하려면 *ActivePage* 속성을 사용하십시오. 활성 페이지를 변경하기 위해 *ActivePage* 속성이나 *ActivePageIndex* 속성을 설정할 수 있습니다.

헤더 컨트롤

헤더 컨트롤(*THeaderControl*)은 런타임에 사용자가 선택하거나 크기를 조정할 수 있는 열 헤더 집합입니다. 컨트롤의 *Sections* 속성을 편집하여 헤더를 추가하거나 수정합니다. 헤더 섹션은 열 또는 필드 위에 둘 수 있습니다. 예를 들어, 헤더 섹션은 리스트 박스(*TListBox*) 위에 둘 수 있습니다.

디스플레이 컨트롤

사용자에게 애플리케이션의 상태 정보를 제공하는 방법은 여러 가지가 있습니다. 예를 들어, *TForm*을 비롯한 일부 컴포넌트에는 런타임에 설정할 수 있는 *Caption* 속성이 있습니다. 다이얼로그 박스를 만들어 메시지를 표시할 수도 있습니다. 그 밖에, 다음 컴포넌트가 런타임에 시각적 피드백을 제공하는 데 특히 유용합니다.

컴포넌트 또는 속성	수행 작업
TStatusBar	일반적으로 윈도우 하단에 상태 영역을 표시합니다.
TProgressBar	특정 작업에 대해 완료된 작업량을 보여 줍니다.
*Hint*와 *ShowHint*	플라이바이(fly-by) 또는 "툴팁" 도움말을 활성화합니다.
*HelpContext*와 *HelpFile*	상황에 맞는 온라인 도움말을 연결합니다.

상태 표시줄

패널을 사용하여 상태 표시줄을 만들 수 있지만 상태 표시줄 컴포넌트를 사용하는 것이 더 간단합니다. 디폴트로, 상태 표시줄의 *Align* 속성은 *alBottom*으로 설정되어 위치와 크기를 모두 제어합니다.

상태 표시줄에서 텍스트 문자열을 한 번에 하나씩만 표시하려면 해당 *SimplePanel* 속성을 *true*로 설정하고 *SimpleText* 속성을 사용하여 상태 표시줄에 표시된 텍스트를 제어할 수 있습니다.

상태 표시줄을 패널이라는 여러 개의 텍스트 영역으로 분리할 수 있습니다. 패널을 만들려면 Object Inspector에서 *Panels* 속성을 편집하고 Panels Editor에서 각 패널의 *Width*, *Alignment* 및 *Text* 속성을 설정합니다. 각 패널의 *Text* 속성에는 패널에 표시된 텍스트가 포함됩니다.

진행 표시줄

애플리케이션에서 시간이 많이 소요되는 작업을 수행할 때, 진행 표시줄을 사용하여 작업의 완료된 정도를 표시할 수 있습니다. 진행 표시줄에는 왼쪽에서 오른쪽으로 증가하는 점선이 표시됩니다.

그림 9.2 진행 표시줄



Position 속성은 점선의 길이를 지정합니다. *Max*와 *Min*은 *Position*의 범위를 결정합니다. 선을 길게 늘리려면 *StepBy* 또는 *StepIt* 메소드를 호출하여 *Position*을 증가시킵니다. *Step* 속성은 *StepIt*가 사용하는 증가 단위를 결정합니다.

도움말과 힌트 속성

대부분의 비주얼(visual) 컨트롤은 런타임 시 플라이바이(fly-by) 힌트 뿐만 아니라 상황에 맞는 도움말도 표시할 수 있습니다. *HelpContext* 및 *HelpFile* 속성은 컨트롤에 대한 도움말 컨텍스트 번호와 도움말 파일을 만듭니다.

Hint 속성에는 마우스 포인터로 컨트롤 또는 메뉴 항목를 가리킬 때 나타나는 텍스트 문자열이 포함됩니다. 힌트를 사용할 수 있게 하려면 *ShowHint*를 *true*로 설정하십시오. *ParentShowHint*를 *true*로 설정하면 컨트롤의 *ShowHint* 속성이 해당 부모와 같은 값을 갖습니다.

그리드

그리드는 정보를 행과 열로 표시합니다. 데이터베이스 애플리케이션을 작성하는 경우에는 14장 "데이터 컨트롤 사용"에서 설명한 *TDBGrid* 또는 *TDBCtrlGrid* 컴포넌트를 사용합니다. 그 외의 경우 표준 그리기 그리드 또는 문자열 그리드를 사용합니다.

그리기 그리드

그리기 그리드(*TDrawGrid*)는 임의의 데이터를 테이블 형식으로 표시합니다. *OnDrawCell* 이벤트 핸들러를 작성하여 그리드의 셀을 채웁니다.

- *CellRect* 메소드는 지정된 셀의 화면 좌표를 반환하지만 *MouseToCell* 메소드는 지정된 화면 좌표에 있는 셀의 열과 행을 반환합니다. *Selection* 속성은 현재 선택한 셀의 테두리를 나타냅니다.
- *TopRow* 속성은 현재 그리드의 상단에 있는 행을 결정합니다. *LeftCol* 속성은 왼쪽에서 첫 번째 보이는 열을 결정합니다. *VisibleColCount*와 *VisibleRowCount*는 그리드에 보이는 열과 행의 수입니다.
- *ColWidths* 속성과 *RowHeights* 속성을 사용하여 열이나 행의 너비 또는 높이를 변경할 수 있습니다. *GridLineWidth* 속성을 사용하여 그리드 선의 너비를 설정합니다. *ScrollBars* 속성을 사용하여 스크롤 막대를 그리드에 추가합니다.
- *FixedCols* 속성과 *FixedRows* 속성을 사용하여 고정되거나 스크롤이 없는 열과 행을 갖도록 선택할 수 있습니다. *FixedColor* 속성을 사용하여 고정된 열과 행에 색상을 지정합니다.
- *Options, DefaultColWidth* 및 *DefaultRowHeight* 속성은 그리드의 모습과 동작에 영향을 미칠 수 있습니다.

문자열 그리드

문자열 그리드 컴포넌트는 문자열의 표시를 단순화하는 특화된 기능을 추가하는 *TDrawGrid*의 자손입니다. *Cells* 속성은 그리드의 각 셀에 대한 문자열을 나열하고 *Objects* 속성은 각 문자열에 연결된 객체를 나열합니다. 모든 문자열과 특정한 열이나 행에 관련된 객체는 *Cols* 속성 또는 *Rows* 속성을 통해 액세스할 수 있습니다.

값 리스트 에디터(VCL만 해당)

*TValueListEditor*는 Name=Value 형식의 이름/값 쌍을 포함하는 문자열 리스트를 편집하기 위해 특화된 그리드입니다. 이름과 값은 *Strings* 속성의 값인 *TStrings* 자손으로 저장됩니다. *Values* 속성을 사용하여 이름의 값을 찾을 수 있습니다. *TValueListEditor*는 크로스 플랫폼 프로그래밍에 사용할 수 없습니다.

그리드에는 이름과 값을 위한 두 개의 열이 있습니다. 디폴트로, Name 열의 이름은 "Key"이고 Value 열의 이름은 "Value"로 지정됩니다. *TitleCaptions* 속성을 설정하여 이러한 기본값을 변경할 수 있습니다. *DisplayOptions* 속성을 사용하여 이러한 제목을 생략할 수 있습니다. 이 속성은 컨트롤의 크기를 조정할 때 크기 조정을 제어하기도 합니다.

사용자가 *KeyOptions* 속성을 사용하여 Name 열을 편집할 수 있는지 여부를 제어할 수 있습니다. *KeyOptions*에는 편집, 새 이름 추가, 이름 삭제 및 새 이름의 고유성 여부를 제어하는 각각의 옵션이 있습니다.

ItemProps 속성을 사용하여 사용자가 Value 열의 항목을 편집하는 방법을 제어할 수 있습니다. 항목에는 다음을 수행할 수 있게 해주는 각각의 *TItemProp* 객체가 있습니다.

- 유효한 입력을 제한하는 편집 마스크 제공
- 값의 최대 길이 지정
- 값을 읽기 전용으로 표시
- 값 리스트 에디터에 사용자가 값을 선택할 수 있는 값 선택 리스트를 여는 드롭다운 화살표를 표시할지, 사용자가 값을 입력하는 다이얼로그 박스를 표시하기 위한 이벤트를 실행하는 생략 부호(...) 버튼을 표시할지 지정합니다.

  드롭다운 화살표가 표시되도록 지정한 경우에는 사용자가 선택할 수 있는 값 리스트를 제공해야 합니다. 이 리스트는 정적 리스트(*TItemProp* 객체의 *PickList* 속성)거나 값 리스트 에디터의 *OnGetPickList* 이벤트를 사용하여 런타임 시 동적으로 추가될 수 있습니다. 이러한 두 방법을 결합하여 *OnGetPickList* 이벤트 핸들러가 정적 리스트를 수정하게 할 수도 있습니다.

  생략 부호 버튼이 표시되도록 지정한 경우에는 값 설정을 비롯하여 사용자가 버튼을 클릭할 때 발생할 응답을 제공해야 합니다. *OnEditButtonClick* 이벤트 핸들러를 작성하여 이 응답을 제공합니다.

그래픽 컨트롤

다음 컴포넌트를 사용하면 그래픽을 애플리케이션에 쉽게 통합할 수 있습니다.

컴포넌트	표시 내용
TImage	그래픽 파일
TShape	기하학적 도형
TBevel	3차원 선 및 프레임
TPaintBox	런타임에 프로그램이 그리는 그래픽
TAnimate	AVI 파일(VCL만 해당)

여기에는 포커스를 받을 필요가 없는 공통 그리기 루틴(*Repaint*, *Invalidate* 등)이 포함된다는 것에 유의하십시오.

이미지

이미지 컴포넌트는 비트맵, 아이콘 또는 메타파일과 같은 그래픽 이미지를 표시합니다. *Picture* 속성은 표시할 그래픽을 결정합니다. *Center*, *AutoSize*, *Stretch*, *Transparent*를 사용하여 표시 옵션을 설정합니다. 자세한 내용은 10-1페이지의 "그래픽 프로그래밍 개요"를 참조하십시오.

도형

도형 컴포넌트는 기하학적 도형을 표시합니다. 이 컴포넌트는 비윈도우 컨트롤(CLX의 widget 기반이 아님)이므로 사용자 입력을 받을 수 없습니다. *Shape* 속성은 컨트롤이 가정하는 도형을 결정합니다. 도형의 색상을 변경하거나 패턴을 추가하려면 *TBrush* 객체의 *Brush* 속성을 사용합니다. 도형을 그리는 방법은 *TBrush*의 *Color* 속성과 *Style* 속성에 따라 다릅니다.

3D

3D 컴포넌트(*TBevel*)는 음각과 양각 모습으로 나타날 수 있는 선입니다. *TPanel*과 같은 일부 컴포넌트는 3D 테두리를 만들 수 있는 기본 속성을 가지고 있습니다. 기본 속성을 사용할 수 없으면 *TBevel*을 사용하여 3D 윤곽, 박스 또는 프레임을 만듭니다.

그리기 박스

그리기 박스 컴포넌트(*TPaintBox*)를 사용하면 애플리케이션이 폼에 그릴 수 있습니다. *OnPaint* 이벤트 핸들러를 작성하여 그리기 박스의 *Canvas*에 이미지를 직접 렌더링할 수 있습니다. 그리기 박스의 테두리 밖에는 그릴 수 없습니다. 자세한 내용은 10-1페이지의 "그래픽 프로그래밍 개요"를 참조하십시오.

애니메이션 컨트롤 (VCL만 해당)

애니메이션 컴포넌트는 소리 없이 AVI(Audio Video Interleaved) 클립을 표시하는 윈도우입니다. AVI 클립은 동영상과 같은 일종의 비트맵 프레임입니다. AVI 클립에 소리가 있을 수는

있지만 애니메이션 컨트롤은 소리 없는 AVI 클립에서만 작동합니다. 사용하는 파일은 압축되지 않은 AVI 파일이거나 RLE(Run-Length Encoding)을 사용하여 압축된 AVI 클립이어야 합니다. 애니메이션 컨트롤은 크로스 플랫폼 프로그래밍에 사용할 수 없습니다.

다음은 애니메이션 컴포넌트의 일부 속성입니다.

- *ResHandle*은 AVI 클립을 리소스로 포함하는 모듈을 위한 Windows 핸들입니다. 런타임에 애니메이션 리소스를 포함하는 모듈의 모듈 핸들이나 인스턴스 핸들에 *ResHandle*을 설정합니다. *ResHandle*을 설정한 다음 *ResID* 또는 *ResName* 속성을 설정하여 지정된 모듈에서 애니메이션 컨트롤이 표시해야 할 AVI 클립을 지정합니다.
- *AutoSize*를 **true**로 설정하여 애니메이션 컨트롤에서 해당 크기를 AVI 클립의 프레임 크기로 조정하게 합니다.
- *StartFrame* 및 *StopFrame*은 클립을 시작하고 중지하는 프레임을 지정합니다.
- *CommonAVI*를 설정하여 Shell32.DLL에 제공된 일반적인 Windows AVI 클립 중 하나를 표시합니다.
- *Active* 속성을 각각 **true**와 **false**로 설정하여 애니메이션을 시작하거나 중지할 시기를 지정하고 *Repetitions* 속성을 설정하여 여러 반복 부분을 재생할 방법을 지정합니다.
- *Timers* 속성을 사용하면 타이머를 사용하여 프레임을 표시할 수 있습니다. 이 속성은 애니메이션 시퀀스와 사운드 트랙 재생 등 다른 액션을 동기화할 때 유용합니다.

장

10

그래픽 및 멀티미디어 작업

그래픽 요소 및 멀티미디어 요소를 사용하여 애플리케이션을 더욱 멋지게 꾸밀 수 있습니다. C++Builder는 다양한 방법으로 애플리케이션에 그래픽 및 멀티미디어를 추가할 수 있습니다. 그래픽 요소를 추가하려면 디자인 타임에 미리 그려놓은 그림을 삽입하거나 그래픽 컨트롤을 사용하여 그래픽 요소를 만들거나 런타임 시 동적으로 그릴 수 있습니다. 멀티미디어 기능을 추가하기 위해 C++Builder에는 오디오 및 비디오 클립을 재생할 수 있는 특수 컴포넌트가 들어 있습니다.

CLX 멀티미디어 컴포넌트는 VCL에서만 사용할 수 있습니다.

그래픽 프로그래밍 개요

Graphics 유닛에서 정의한 VCL 그래픽 컴포넌트는 Windows GDI(Graphics Device Interface)를 캡슐화하여 Windows 애플리케이션에 쉽게 그래픽을 추가할 수 있도록 합니다. QGraphics 유닛에서 정의된 CLX 그래픽 컴포넌트는 크로스 플랫폼 애플리케이션에 그래픽을 추가하기 위한 Qt 그래픽 widget을 캡슐화합니다.

C++Builder 애플리케이션에서 그래픽을 그리려면 객체에 직접 그리는 것이 아니라 객체의 *캔버스* 위에 그립니다. 캔버스는 객체의 속성이면서 그 자체가 객체이기도 합니다. 캔버스 객체는 리소스를 효과적으로 처리하고 장치 컨텍스트를 관리하므로 화면이나 프린터, 비트맵 또는 메타파일(CLX에서는 드로잉) 중 어디에서 그림을 그리든 관계 없이 프로그램이 동일한 메소드를 사용할 수 있다는 이점이 있습니다. 캔버스는 런타임 시에만 사용할 수 있으므로 코드를 작성하면 캔버스에서 모든 작업을 할 수 있습니다.

VCL *TCanvas*는 Windows 장치 컨텍스트 주변의 랩퍼(wrapper) 리소스 관리자이므로 캔버스에서 Windows GDI의 모든 기능을 사용할 수 있습니다. 캔버스의 *Handle* 속성은 장치 컨텍스트 핸들입니다.

CLX *TCanvas*는 Qt painter 주변의 랩퍼 리소스 관리자입니다. 캔버스의 *Handle* 속성은 Qt painter 객체의 인스턴스에 대한 타입이 지정된 포인터입니다. 이 속성을 노출하면 QpainterH를 필요로 하는 저수준 Qt 그래픽 라이브러리 함수를 사용할 수 있습니다.

애플리케이션에 그래픽 이미지가 표시되는 모양은 그리는 캔버스의 객체 타입에 따라 다릅니다. 컨트롤 캔버스에 직접 그림을 그리면 그림이 즉시 표시됩니다. 그러나 *TBitmap* 캔버스와 같은 오프스크린 이미지에서 그리는 경우 컨트롤이 비트맵에서 컨트롤의 캔버스로 이미지를 복사해야 이미지가 표시됩니다. 즉, 비트맵을 그려 이미지 컨트롤에 할당하는 경우 이미지는 컨트롤이 해당 *OnPaint* 메시지(VCR) 또는 이벤트(CLX)를 처리할 수 있을 때에만 표시됩니다.

그래픽을 사용하다 보면 *그리기*(Drawing)와 *색칠*(Painting)이라는 용어가 종종 나옵니다.

- 그리기(Drawing)는 코드를 사용하여 선이나 도형과 같은 특정 단일 그래픽 요소를 작성하는 것입니다. 코드에서 캔버스의 그리기 메소스를 호출하여 객체가 캔버스의 특정 위치에 특정 그래픽을 그리게 할 수 있습니다.
- 색칠(Painting)은 객체의 전체 모양을 만드는 것입니다. 색칠은 일반적으로 그리기를 필요로 합니다. 즉, *OnPaint* 이벤트의 응답으로 객체는 일반적으로 일부 그래픽을 그립니다. 예를 들어, 에디트 박스는 사각형을 그린 다음 내부에 일부 텍스트를 그려 자신을 색칠합니다. 반면에 도형 컨트롤은 단일 그래픽을 그려서 자신을 색칠합니다.

이 장의 처음에 나오는 예제는 다양한 그래픽을 그리는 방법을 보여주지만 이러한 작업은 *OnPaint* 이벤트에 대한 응답을 수행하는 것입니다. 이후 단원에서는 다른 이벤트에 응답하여 같은 종류의 그리기를 수행하는 방법을 보여 줍니다.

화면 새로 고침

특정한 경우 운영 체제는 화면상의 객체 모양을 새로 고칠 필요가 있다고 결정하고 Windows에서 WM_PAINT 메시지를 생성합니다. 이 메시지는 다시 VCL에 의해 *OnPaint* 이벤트로 라우트됩니다. 크로스 플랫폼 개발을 위해 CLX를 사용하는 경우 CLX가 *OnPaint* 이벤트로 라우트하는 그리기 이벤트가 생성됩니다. 해당 객체에 대해 *OnPaint* 이벤트 핸들러를 미리 작성해 놓은 경우 *Refresh* 메소드를 사용하면 이 이벤트 핸들러를 호출합니다. 폼의 *OnPaint* 이벤트 핸들러에 생성된 디폴트 이름은 *FormPaint*입니다. 경우에 따라 *Refresh* 메소드를 사용하여 컴포넌트나 폼을 새로 고칠 수도 있습니다. 예를 들어, 폼의 *OnResize* 이벤트 핸들러에서 *Refresh*를 호출하여 모든 그래픽을 다시 표시할 수 있으며 VCL을 사용하는 경우 폼의 배경을 색칠할 수 있습니다.

일부 운영 체제에서는 무효화된 윈도우의 클라이언트 영역을 자동으로 다시 그리지만 Windows는 이 작업을 수행하지 않습니다. Windows 운영 체제에서는 화면에 그린 모든 그림에 영구적 속성이 있습니다. 예를 들어, 윈도우를 끄는 동안 폼이나 컨트롤이 일시적으로 흐려지면 해당 폼이나 컨트롤을 제 위치에 다시 놓을 때 흐려진 영역을 다시 색칠해야 합니다. WM_PAINT 메시지에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

TImage 컨트롤을 사용하여 폼에 그래픽 이미지를 표시하는 경우 *TImage*에 들어 있는 그래픽의 색칠 및 새로 고침은 자동으로 처리됩니다. *Picture* 속성은 *TImage*가 표시하는 실제 비트맵, 그리기 또는 다른 그래픽 객체를 지정합니다. 또한 *Proportional* 속성을 설정하여 이미지를 비틀림 없이 이미지 컨트롤 안에 완전히 표시할 수 있습니다. *TImage*에 그린 이미지는 영구적입니다. 따라서 Timage에 포함된 이미지를 다시 그릴 필요가 없습니다. 반면에 *TPaintBox*의 캔버스는 스크린 장치(VCL) 또는 painter(CLX)에 직접 연결되어 있으므로 *PaintBox*의 캔버스에 그려진 모든 그림은 일시적입니다. 폼 자체를 비롯하여 거의 모든 컨트롤 또한 일시적입니다. 따라서 해당 생성자에 있는 *TPaintBox*에 그리거나 색칠하는 경우 클라이언트 영역이 무효화될 때마다 이미지가 다시 칠해지도록 *OnPaint* 이벤트 핸들러에 해당 코드를 추가해야 합니다.

그래픽 객체 타입

VCL/CLX는 표 10.1의 그래픽 객체를 제공합니다.이러한 객체는 10-10페이지의 "캔버스 메소드를 사용하여 그래픽 객체 그리기"에 설명된 것처럼 캔버스에 그릴 수 있고, 10-19페이지의 "그래픽 파일 로드 및 저장"에서 설명한대로 그래픽 파일로 로드 및 저장할 수 있는 메소드를 가집니다.

표 10.1 그래픽 객체 타입

객체	설명
Picture	그림은 모든 그래픽 이미지를 갖기 위해 사용합니다. 추가 그래픽 파일 형식을 추가하려면 Picture *Register* 메소드를 사용합니다. 이 메소드를 사용하면 이미지 컨트롤에 이미지를 표시하는 것과 같이 임의의 파일을 처리할 수 있습니다.
Bitmap	비트맵 객체는 이미지를 만들고, 처리(크기 변경, 스크롤, 회전 및 색칠)하며 이미지를 파일로서 디스크에 저장하는 데 사용하는 강력한 그래픽 객체입니다. 이미지가 아니라 *handle*을 복사하므로 비트맵 복사본을 만드는 것은 빠릅니다.
Clipboard	클립보드는 애플리케이션에서 또는 애플리케이션으로 잘라내거나, 복사하거나 붙여넣은 텍스트나 그래픽의 컨테이너를 나타냅니다. 클립보드를 사용하면 적절한 형식에 따라 데이터를 얻고 검색하고, 참조 카운팅을 처리하며 클립보드를 열거나 닫고, 클립보드 내의 객체 형식을 관리하고 처리할 수 있습니다.
Icon	아이콘은 아이콘 파일(::ICO 파일)에서 로드된 값을 나타냅니다.
Metafile(VCL만 해당) Drawing(CLX만 해당)	이미지의 실제 비트맵 픽셀이 들어 있는 것이 아니라 이미지를 생성하는 데 필요한 작업을 기록하는 파일이 들어 있습니다. 메타파일 또는 드로잉은 이미지의 상세 정보를 잃지 않고 자유 자재로 크기를 조정할 수 있으며 특히 프린터와 같은 고해상도 장치에서는 비트맵보다 훨씬 적은 메모리를 필요로 합니다. 그러나 메타파일과 드로잉은 비트맵만큼 빨리 표시되지 않습니다. 성능보다는 다양한 기능이나 정밀도가 더 중요한 경우에 메타파일이나 드로잉을 사용합니다.

캔버스의 일반적인 속성과 메소드

표 10.2에서는 캔버스 객체에서 공통적으로 사용되는 속성을 나열합니다. 속성과 메소드의 전체 리스트는 온라인 도움말의 *TCanvas* 컴포넌트를 참조하십시오.

표 10.2 캔버스 객체의 일반적인 속성

속성	설명
Font	이미지 위에 텍스트를 작성할 때 사용할 글꼴을 지정합니다. TFont 객체의 속성을 설정하여 글꼴, 색상, 크기 및 스타일을 지정합니다.
Brush	캔버스가 그래픽 모양과 배경을 채우는 데 사용하는 색상과 패턴을 결정합니다. TBrush 객체의 속성을 설정하여 캔버스의 공간을 채울 때 사용할 색상, 패턴 또는 비트맵을 지정합니다.
Pen	캔버스가 선과 윤곽 모양을 그리는 데 사용할 펜의 종류를 지정합니다. TPen 객체의 속성을 설정하여 펜의 색상, 스타일, 너비 및 모드를 지정합니다.
PenPos	펜의 현재 그리기 위치를 지정합니다.
Pixels	현재 ClipRect 내에서의 픽셀 영역의 색상을 지정합니다.

이러한 속성은 10-5페이지의 "캔버스 객체의 속성 사용"에 보다 자세하게 설명되어 있습니다.

표 10.3은 사용할 수 있는 메소드 리스트입니다.

표 10.3 캔버스 객체의 일반 메소드

메소드	설명
Arc	지정된 사각형에 내접한 타원의 둘레를 따라 이미지에 원호를 그립니다.
Chord	선과 타원의 교차점으로 표시되는 닫힌 그림을 그립니다.
CopyRect	다른 캔버스에서 현재 캔버스로 이미지의 일부를 복사합니다.
Draw	좌표(X, Y)에 의해 지정된 위치의 캔버스에 Graphic 매개변수에 의해 지정된 그래픽 객체를 렌더링합니다.
Ellipse	경계 사각형에 의해 정의된 타원을 캔버스에 그립니다.
FillRect	현재 브러시를 사용하여 캔버스의 지정된 사각형을 채웁니다.
FloodFill(VCL만 해당)	현재 브러시를 사용하여 캔버스의 영역을 채웁니다.
FrameRect	캔버스에서 Brush를 사용하여 사각형을 그려 테두리를 그립니다.
LineTo	캔버스에 PenPos에서 X와 Y로 지정된 지점까지 선을 그리고 펜 위치를 (X, Y) 지점으로 설정합니다.
MoveTo	현재 그리기 위치를 (X,Y) 지점으로 변경합니다.
Pie	좌표 (X1, Y1)과 (X2, Y2)에 해당하는 사각형에 내접하는 파이 모양의 타원 섹션을 캔버스위에 그립니다.

표 10.3 캔버스 객체의 일반 메소드 (계속)

메소드	설명
Polygon	마지막 지점에서 첫 번째 지점까지 선을 그려서 통과 지점을 연결하고 도형을 닫는 일련의 선을 캔버스에 그립니다.
Polyline	현재 펜을 사용하여 캔버스 위에 일련의 선을 그리고 펜으로 전달된 각 지점을 Points에서 연결합니다.
Rectangle	왼쪽 위 모서리 지점 (X1, Y1)과 오른쪽 아래 모서리 지점 (X2, Y2)으로 이루어진 사각형을 캔버스에 그립니다. 펜을 사용하여 *Rectangle*로 상자를 그리고 브러시로 상자를 채웁니다.
RoundRect	캔버스에 모서리가 둥근 사각형을 그립니다.
StretchDraw	이미지가 지정된 사각형에 맞도록 캔버스에 그래픽을 그립니다. 그래픽 이미지의 크기나 가로 세로 비율을 알맞게 변경해야 할 수도 있습니다.
TextHeight, TextWidth	현재 글꼴로 된 문자열의 높이와 너비를 각각 반환합니다. 높이에는 줄 간격이 포함됩니다.
TextOut	(X,Y) 지점에서 시작하는 문자열을 캔버스에 작성하고 PenPos를 문자열의 끝으로 업데이트합니다.
TextRect	영역 내에 문자열을 작성하며 영역 밖으로 벗어난 문자열은 표시하지 않습니다.

이러한 메소드는 10-10페이지의 "캔버스 메소드를 사용하여 그래픽 객체 그리기"에 보다 자세하게 설명되어 있습니다.

캔버스 객체의 속성 사용

캔버스 객체를 사용하여 선을 그리는 펜, 도형을 채우는 브러시, 텍스트를 작성하는 글꼴, 이미지를 나타내는 픽셀의 배열 속성을 설정할 수 있습니다.

이 단원에서는 다음을 설명합니다.

- 펜 사용
- 브러시 사용
- 픽셀 읽기 및 설정

펜 사용

캔버스의 *Pen* 속성은 도형의 윤곽선을 포함하여 선이 표현되는 방법을 제어합니다. 직선을 그리는 것은 두 지점 간에 있는 픽셀 그룹을 변경하는 것입니다.

펜 자체에는 변경할 수 있는 네 가지 속성인 *Color*, *Width*, *Style*, *Mode*가 있습니다.

- *Color* 속성은 펜 색상을 변경합니다.
- *Width* 속성은 펜 너비를 변경합니다.
- *Style* 속성은 펜 스타일을 변경합니다.
- *Mode* 속성은 펜 모드를 변경합니다.

이 속성 값들에 따라 펜이 선의 픽셀을 변경하는 방법이 결정됩니다. 디폴트로, 모든 펜은 검은색, 1픽셀의 너비, 실선 스타일 및 캔버스에 이미 존재하는 모든 것을 덮어 쓰는 copy 모드로 시작합니다.

*TPenRecall*을 사용하여 펜 속성을 빠르게 저장하고 복원할 수 있습니다.

펜 색상 변경

런타임 시 다른 *Color* 속성을 설정하는 것처럼 펜의 색상을 설정할 수 있습니다. 펜 색상은 일반선 및 다각선의 색상 뿐만 아니라 도형의 경계로 그어진 선 등, 펜이 그리는 선의 색상을 결정합니다. 펜 색상을 변경하려면 펜의 *Color* 속성에 값을 할당합니다.

사용자가 새 색상의 펜을 선택할 수 있도록 하려면 펜의 툴바에 색상 그리드를 넣습니다. 색상 그리드를 사용하여 전경색과 배경색을 모두 설정할 수 있습니다. 그리드가 없는 펜 스타일의 경우 선분 간의 틈에 그려진 배경색을 고려해야 합니다. 배경색은 Brush 색상 속성에 있습니다.

사용자는 그리드를 클릭하여 새 색상을 선택하므로 다음 코드는 *OnClick* 이벤트에 응답하여 펜 색상을 변경합니다.

```
void __fastcall TForm1::PenColorClick(TObject *Sender)
{
  Canvas->Pen->Color = PenColor->ForegroundColor;
}
```

펜 너비 변경

펜 너비는 펜이 그리는 선의 두께를 픽셀 단위로 결정합니다.

참고 두께가 1픽셀보다 크면 Windows에서는 펜의 *Style* 속성 값과 관계 없이 항상 실선을 그립니다.

펜 너비를 변경하려면 펜의 *Width* 속성에 숫자 값을 할당합니다.

펜의 너비 값을 설정하기 위해 펜의 툴바에 스크롤 막대를 넣는다고 가정합니다. 그리고 사용자에게 피드백을 제공하기 위해 스크롤 막대 옆에 있는 레이블을 업데이트한다고 가정합니다. 스크롤 막대의 위치를 사용하여 펜 너비를 결정하면 위치가 변경될 때마다 펜 너비를 업데이트합니다.

스크롤 막대의 *OnChange* 이벤트를 처리하는 방법은 다음과 같습니다.

```
void __fastcall TForm1::PenWidthChange(TObject *Sender)
{
  Canvas->Pen->Width = PenWidth->Position;         // set the pen width
directly
  PenSize->Caption = IntToStr(PenWidth->Position); // convert to string
}
```

펜 스타일 변경

펜의 *Style* 속성을 사용하여 실선, 점괘선, 점선 등을 설정할 수 있습니다.

VCL 참고 Windows에 배포된 크로스 플랫폼 애플리케이션인 경우 Windows는 1픽셀보다 두꺼운 펜의 점괘선, 점선 스타일을 지원하지 않으며 지정한 스타일과 관계 없이 실선을 그립니다.

펜 속성을 설정하는 작업은 이벤트를 처리하기 위해 서로 다른 컨트롤이 동일한 이벤트 핸들러를 공유하도록 하는 경우 이상적입니다. 어느 컨트롤이 실제로 이벤트를 갖는지 보려면 *Sender* 매개변수를 확인합니다.

펜의 툴바에 있는 6개의 펜 스타일 버튼에 대해 하나의 클릭 이벤트(click-event) 핸들러를 만들려면 다음을 수행합니다.

1 여섯 개의 펜 스타일 버튼을 모두 선택하고 Object Inspector | Events | *OnClick* 이벤트를 선택한 다음 Handler 열에 `SetPenStyle`을 입력합니다.

C++Builder 는 *SetPenStyle* 이라는 비어 있는 클릭 이벤트 핸들러를 생성하고 모든 여섯 개 버튼의 *OnClick* 이벤트에 연결합니다.

2 클릭 이벤트를 보낸 컨트롤인 *Sender* 값에 따라 펜 스타일을 설정하여 클릭 이벤트 핸들러를 다음과 같은 방법으로 채웁니다.

```
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender == SolidPen)
    Canvas->Pen->Style = psSolid;
  else if (Sender == DashPen)
    Canvas->Pen->Style = psDash;
  else if (Sender == DotPen)
    Canvas->Pen->Style = psDot;
  else if (Sender == DashDotPen)
    Canvas->Pen->Style = psDashDot;
  else if (Sender == DashDotDotPen)
    Canvas->Pen->Style = psDashDotDot;
‘  else if (Sender == ClearPen)
    Canvas->Pen->Style = psClear;
}
```

펜 스타일 상수를 펜 스타일 버튼의 *Tag* 속성에 넣으면 위의 이벤트 핸들러 코드가 더욱 간단해집니다. 그러면 이벤트 코드는 다음과 같이 나타납니다.

```
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Pen->Style = (TPenStyle) ((TSpeedButton *)Sender)->Tag;
}
```

펜 모드 변경

펜의 *Mode* 속성을 사용하여 펜 색상과 캔버스 위의 색상을 결합하기 위한 다양한 방법을 지정할 수 있습니다. 예를 들어, 펜이 항상 검은색이거나 캔버스 배경색의 반대색이 되거나 펜 색상의 반대색이 되도록 설정할 수 있습니다. 자세한 내용은 온라인 도움말의 *TPen*을 참조하십시오.

펜 위치 파악

펜이 다음 선을 그리기 시작하는 위치인 현재 그리기 위치를 펜 위치라고 합니다. 캔버스는 펜 위치를 *PenPos* 속성에 저장합니다. 도형과 텍스트의 경우 필요한 모든 좌표를 입력하므로 펜 위치는 선 그리기에만 영향을 미칩니다.

펜 위치를 설정하려면 캔버스의 *MoveTo* 메소드를 호출합니다. 예를 들어, 다음 코드는 펜 위치를 캔버스의 왼쪽 위 모서리로 이동합니다.

```
Canvas->MoveTo(0, 0);
```

참고 *LineTo* 메소드로 선을 그려도 현재 위치가 선의 끝점으로 이동합니다.

브러시 사용

캔버스의 *Brush* 속성은 도형의 내부를 포함하여 영역을 채우는 방법을 제어합니다. 브러시로 영역을 채우는 것은 지정된 방법으로 많은 인접한 픽셀 을 변경하는 방법입니다.

브러시에는 처리할 수 있는 다음과 같은 세 가지 속성이 있습니다.

- *Color* 속성은 채우기 색상을 변경합니다.
- *Style* 속성은 브러시 스타일을 변경합니다.
- *Bitmap* 속성은 비트맵을 브러시 패턴으로 사용합니다.

이러한 속성 값으로 캔버스가 도형이나 다른 영역을 채우는 방법을 결정합니다. 디폴트로, 모든 브러시는 흰색의 실선 스타일로 시작하며 패턴 비트맵을 사용하지 않습니다.

*TBrushRecall*을 사용하여 브러시 속성을 빠르게 저장하고 복원할 수 있습니다.

브러시 색상 변경

브러시 색상은 캔버스가 도형을 채우는 데 사용할 색상을 결정합니다. 채우기 색상을 변경하려면 브러시의 *Color* 속성에 값을 할당합니다. 브러시는 텍스트의 배경색과 선 그리기에 사용되므로 일반적으로 브러시를 사용하여 배경색 속성을 설정합니다.

브러시 툴바의 색상 그리드를 클릭하여 펜 색상과 동일한 방법으로 브러시 색상을 설정할 수 있습니다(10-6페이지의 "펜 색상 변경" 참조).

```
void __fastcall TForm1::BrushColorClick(TObject *Sender)
{
  Canvas->Brush->Color = BrushColor->BackgroundColor;
}
```

브러시 스타일 변경

브러시 스타일은 캔버스가 도형을 채우는 데 사용하는 패턴을 결정합니다. 브러시 스타일을 사용하여 브러시 색상과 캔버스에 이미 있는 색상을 결합하는 다양한 방법을 지정할 수 있습니다. 미리 정의된 스타일에는 단색, 무색 및 다양한 선과 해칭 패턴이 있습니다.

브러시 스타일을 변경하려면, *Style* 속성을 미리 정의된 *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, *bsDiagCross* 값 중 하나로 설정합니다.

이 예제는 8개의 브러시 스타일 버튼 집합에서 클릭 이벤트 핸들러를 공유하여 브러시 스타일을 설정합니다. 8개의 버튼을 모두 선택하고 Object Inspector | Events | *OnClick* 으로 설정한 다음 *OnClick* 핸들러의 이름을 *SetBrushStyle*로 지정합니다. 핸들러 코드는 다음과 같습니다.

```
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
  if (Sender == SolidBrush)
    Canvas->Brush->Style = bsSolid;
  else if (Sender == ClearBrush)
    Canvas->Brush->Style = bsClear;
  else if (Sender == HorizontalBrush)
    Canvas->Brush->Style = bsHorizontal;
  else if (Sender == VerticalBrush)
    Canvas->Brush->Style = bsVertical;
  else if (Sender == FDiagonalBrush)
    Canvas->Brush->Style = bsFDiagonal;
  else if (Sender == BDiagonalBrush)
    Canvas->Brush->Style = bsBDiagonal;
  else if (Sender == CrossBrush)
    Canvas->Brush->Style = bsCross;
  else if (Sender == DiagCrossBrush)
    Canvas->Brush->Style = bsDiagCross;
}
```

브러시 스타일 상수를 브러시 스타일 버튼의 *Tag* 속성에 넣으면 위의 이벤트 핸들러 코드가 더욱 간단해집니다. 그러면 이벤트 코드는 다음과 같이 나타납니다.

```
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
   if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Brush->Style = (TBrushStyle) ((TSpeedButton *)Sender)->Tag;
}
```

브러시의 비트맵 속성 설정

브러시의 *Bitmap* 속성을 사용하여 브러시가 도형 및 다른 영역을 채우기 위한 패턴으로 사용하는 비트맵 이미지를 지정합니다.

다음 예제는 파일에서 비트맵을 로드하고 비트맵을 Form1 Canvas의 Brush에 할당합니다.

```
BrushBmp->LoadFromFile("MyBitmap.bmp");
  Form1->Canvas->Brush->Bitmap = BrushBmp;
  Form1->Canvas->FillRect(Rect(0,0,100,100));
```

참고 브러시는 자신의 *Bitmap* 속성에 할당된 비트맵 객체의 소유권을 갖지 않습니다. 브러시를 사용하는 동안에는 비트맵 객체가 유효한 상태로 있는지를 확인해야 하고 사용한 후에는 직접 비트맵 객체를 해제해야 합니다.

픽셀 읽기 및 설정

모든 캔버스에는 캔버스의 이미지를 구성하는 색상을 입힌 점을 각각 나타내는 인덱싱된 *Pixels* 속성이 있습니다. 이 속성은 픽셀의 색상을 찾거나 설정하는 등의 단순한 작업의 편의성을 위해서만 사용할 수 있으며 *Pixels*을 직접 액세스하는 일은 거의 없습니다.

참고 각 픽셀을 설정하고 가져오는 것은 영역에서 그래픽 작업을 하는 것보다 훨씬 오래 걸립니다. Pixel 배열 속성을 사용하여 일반 배열의 이미지 픽셀에 액세스하지 마십시오. 이미지 픽셀에 대한 고성능 액세스에 대한 내용은 *TBitmap::ScanLine* 속성을 참조하십시오.

캔버스 메소드를 사용하여 그래픽 객체 그리기

이 단원에서는 일반적인 일부 메소드를 사용하여 그래픽 객체를 그리는 방법을 설명합니다. 이 단원에서는 다음과 같은 내용을 다룹니다.

- 선 및 다각선 그리기
- 도형 그리기
- 모서리가 둥근 사각형 그리기
- 다각형 그리기

선 및 다각선 그리기

캔버스에서는 직선과 다각선을 그릴 수 있습니다. 직선은 두 점을 연결하는 픽셀의 선입니다. 다각선은 끝과 끝이 연결된 일련의 직선입니다. 캔버스에서는 펜을 사용하여 모든 선을 그립니다.

선 그리기

캔버스에 직선을 그리려면 캔버스의 *LineTo* 메소드를 사용합니다.

*LineTo*는 현재의 펜 위치에서 지정하는 펜 위치까지 선을 그리고 선의 끝점을 현재 위치로 나타냅니다. 캔버스에서는 펜을 사용하여 선을 그립니다.

예를 들어, 다음 메소드는 폼을 색칠할 때마다 폼에 서로 교차하는 두 개의 대각선을 그립니다.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->MoveTo(0,0);
  Canvas->LineTo(ClientWidth, ClientHeight);
  Canvas->MoveTo(0, ClientHeight);
  Canvas->LineTo(ClientWidth, 0);
}
```

다각선 그리기

개별적인 선 외에도 캔버스에서는 연결된 여러 선분의 그룹인 다각선을 그릴 수 있습니다.

캔버스에 다각선을 그리려면 캔버스의 *Polyline* 메소드를 호출합니다.

Polyline 메소드에 전달된 매개변수는 점의 배열입니다. 다각선이 첫 번째 점에서 *MoveTo*를 수행하고 각 연속된 점에서는 *LineTo*를 수행하는 경우를 생각해 볼 수 있습니다. 여러 선을 그리는 경우 *Polyline* 메소드를 사용하면 *MoveTo* 메소드와 *LineTo* 메소드를 사용하는 것보다 더 빠릅니다. 왜냐하면 Polyline 메소드의 경우 많은 호출 오버헤드를 제거하기 때문입니다.

예를 들어, 다음 메소드는 폼에 마름모꼴을 그립니다.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TPoint vertices[5];
  vertices[0] = Point(0, 0);
  vertices[1] = Point(50, 0);
  vertices[2] = Point(75, 50);
  vertices[3] = Point(25, 50);
  vertices[4] = Point(0, 0);
  Canvas->Polyline(vertices, 4);
}
```

*Polyline*에 대한 마지막 매개변수는 점의 수가 아니라 마지막 점의 인덱스입니다.

도형 그리기

캔버스에는 여러 종류의 도형을 그리는 데 사용하는 메소드가 있습니다. 캔버스에서는 펜으로 도형의 윤곽을 그린 다음 브러시로 내부를 채웁니다. 도형의 테두리를 형성하는 선은 현재의 *Pen* 객체가 제어합니다.

이 단원에서는 다음과 같은 내용을 다룹니다.

- 사각형 및 타원 그리기
- 모서리가 둥근 사각형 그리기
- 다각형 그리기

사각형 및 타원 그리기

캔버스에 사각형이나 타원을 그리려면 캔버스의 *Rectangle* 메소드나 *Ellipse* 메소드를 호출하여 경계 사각형의 좌표를 전달하십시오.

Rectangle 메소드는 경계 사각형을 그리고, *Ellipse* 메소드는 사각형의 모든 면에 내접하는 타원을 그립니다.

다음 메소드는 폼의 왼쪽 위 사분면을 채우는 사각형을 그린 다음 동일한 영역에 타원을 그립니다.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Rectangle(0, 0, ClientWidth/2, ClientHeight/2);
  Canvas->Ellipse(0, 0, ClientWidth/2, ClientHeight/2);
}
```

모서리가 둥근 사각형 그리기

캔버스에 모서리가 둥근 사각형을 그리려면 캔버스의 *RoundRect* 메소드를 호출하십시오.

*RoundRect*에 전달된 첫 번째 네 개의 매개변수는 *Rectangle* 메소드나 *Ellipse* 메소드의 경우처럼 경계 사각형입니다. *RoundRect*는 둥근 모서리를 그리는 방법을 나타내는 두 개의 매개변수를 추가로 취합니다.

예를 들어, 다음 메소드는 폼의 왼쪽 위 사분면에 지름이 10 픽셀인 원의 부분을 사용하여 모서리를 둥글게 한 사각형을 그립니다.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->RoundRect(0, 0, ClientWidth/2, ClientHeight/2, 10, 10);
}
```

다각형 그리기

캔버스에 여러 개의 면을 갖는 다각형을 그리려면 캔버스의 *Polygon* 메소드를 호출하십시오.

Polygon 메소드는 점의 배열을 유일한 매개변수로 가지며 펜으로 점을 연결한 다음 마지막 점을 첫 번째 점과 연결하여 다각형을 닫습니다. 선을 그린 후에는, *Polygon* 메소드가 브러시를 사용하여 다각형 내부 영역을 채웁니다.

예를 들어, 다음 코드는 폼의 왼쪽 아래 중간 부분에 직각 삼각형을 그립니다.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TPoint vertices[3];
  vertices[0] = Point(0, 0);
  vertices[1] = Point(0, ClientHeight);
  vertices[2] = Point(ClientWidth,ClientHeight);
  Canvas->Polygon(vertices,2);
}
```

애플리케이션에서 여러 그리기 객체 처리

일반적으로 툴바와 버튼 패널에서는 사각형, 도형, 선 등 다양한 그리기 메소드를 사용할 수 있습니다. 애플리케이션은 원하는 그리기 객체를 설정하기 위해 스피드 버튼 클릭에 응답할 수 있습니다. 이 단원에서는 다음을 수행하는 방법을 설명합니다.

- 사용할 드로잉 툴 파악
- 스피드 버튼으로 도구 변경
- 드로잉 툴 사용

사용할 드로잉 툴 파악

그래픽 프로그램은 사용자가 특정 상황에서 사용하기를 원하는 드로잉 툴(선, 사각형, 타원 또는 모서리가 둥근 사각형)의 종류를 파악해야 합니다. 일반적으로 C++ 열거형을 사용하여 사용 가능한 툴을 나열합니다. 열거형은 타입 선언이기도 하므로 해당 지정 값 *만* 할당하도록 C++의 형식 검사를 사용할 수 있습니다.

예를 들어, 다음 코드는 그래픽 애플리케이션에서 사용 가능한 각 드로잉 툴의 열거 타입을 선언합니다.

```
typedef enum {dtLine, dtRectangle, dtEllipse, dtRoundRect} TDrawingTool;
```

TDrawingTool 타입의 변수는 상수 *dtLine, dtRectangle, dtEllipse,* , 또는 *dtRoundRect* 중 하나에만 할당될 수 있습니다.

규칙에 따라 타입 식별자는 *T*로 시작하고 유사한 상수 그룹(열거 타입을 구성하는 상수 등)은 두 글자의 접두사로 시작합니다. 예를 들어, "드로잉 툴(drawing tool)"의 경우 *dt*로 시작합니다.

다음 코드에서 폼에 추가된 필드는 폼의 드로잉 툴을 파악합니다.

```
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};

class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X,int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int
X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X,int Y);
private:// User declarations
public:// User declarations
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  //field to track whether button was pressed
  TPoint Origin, MovePt;  // fields to store points
  TDrawingTool DrawingTool;  // field to hold current tool
};
```

스피드 버튼으로 툴 변경

각 드로잉 툴은 연결된 *OnClick* 이벤트 핸들러가 필요합니다. 애플리케이션에 네 개의 드로잉 툴, 즉 선, 사각형, 타원 또는 모서리가 둥근 사각형 각각에 대한 버튼이 있다고 가정합니다. 다음 이벤트 핸들러를 네 개의 드로잉 툴 버튼의 *OnClick* 이벤트에 연결하고 *DrawingTool*을 각 드로잉 툴 버튼에 적절한 값으로 설정합니다.

```
void __fastcall TForm1::LineButtonClick(TObject *Sender)  // LineButton
{
  DrawingTool = dtLine;
}

void __fastcall TForm1::RectangleButtonClick(TObject *Sender) //
RectangleButton
{
  DrawingTool = dtRectangle;
}

void __fastcall TForm1::EllipseButtonClick(TObject *Sender)  //
EllipseButton
{
  DrawingTool = dtEllipse;
}

void __fastcall TForm1::RoundedRectButtonClick(TObject *Sender) //
RoundRectBtn
{
  DrawingTool = dtRoundRect;
}
```

드로잉 툴 사용

이제 사용할 툴을 파악하였으므로 여러 도형을 그리는 방법을 나타내야 합니다. 그리기를 수행하는 유일한 메소드는 마우스 이동(mouse-move) 및 마우스업(mouse-up) 핸들러이며 어떤 툴을 선택하든지 한 개의 그리기 코드가 선을 그립니다.

다른 드로잉 툴을 사용하려면 선택한 툴을 기반으로 코드에서 그리는 방법을 지정해야 합니다. 각 툴의 이벤트 핸들러에 이 명령을 추가합니다.

이 단원에서는 다음을 설명합니다.

- 도형 그리기
- 이벤트 핸들러 간의 코드 공유

도형 그리기

도형을 그리는 것은 선을 그리는 것만큼이나 쉽습니다. 각 도형은 단일문을 취하므로 좌표만 있으면 됩니다.

네 개의 툴 모두에 대한 도형을 그리는 *OnMouseUp* 이벤트 핸들러의 수정 코드는 다음과 같습니다.

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
                                    TShiftState Shift, int X, int Y){
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(Origin.x,Origin.y);
      Canvas->LineTo (X, Y);
      break;
    case dtRectangle:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y);
      break;
    case dtEllipse:
      Canvas->Ellipse(Origin.x, Origin.y, X, Y);
      break;
    case dtRoundRect:
      Canvas->Rectangle(Origin.x,Origin.y, X, Y);Origin.x - X)/2,
                        (Origin.y - Y)/2);
      break;
  }
  Drawing = false;
}
```

물론 도형을 그리려면 *OnMouseMove* 핸들러를 다음과 같이 업데이트해야 합니다.

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X,int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    switch (DrawingTool)
    {
```

```
        case dtLine:
          Canvas->MoveTo(Origin.x, Origin.y);
          Canvas->LineTo(MovePt.x, MovePt.y);
          Canvas->MoveTo(Origin.x, Origin.y);
          Canvas->LineTo(X, Y);
          break;
        case dtRectangle:
          Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y);
          Canvas->Rectangle(Origin.x, Origin.y, X, Y);
          break;
        case dtEllipse:
          Canvas->Ellipse(Origin.x, Origin.y, MovePt.x, MovePt.y);
          Canvas->Ellipse(Origin.x, Origin.y, X, Y);
          break;
        case dtRoundRect:
          Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y,
                            (Origin.x - MovePt.x)/2,(Origin.y - MovePt.y)/2);
          Canvas->Rectangle(Origin.x, Origin.y, X, Y,
                            (Origin.x - X)/2, (Origin.y - Y)/2);
          break;
      }
      MovePt = Point(X, Y);
    }
    Canvas->Pen->Mode = pmCopy;
  }
```

일반적으로 위 예제에 있는 반복적인 코드는 모두 각각의 루틴에 있습니다. 다음 단원에서는 모든 마우스 이벤트 핸들러에서 호출할 수 있는 단일 루틴에 있는 도형 그리기 코드를 모두 보여줍니다.

이벤트 핸들러 간의 코드 공유

대다수의 이벤트 핸들러가 동일한 코드를 사용할 경우 모든 이벤트 핸들러가 공유할 수 있는 루틴으로 반복 코드를 이동시키면 애플리케이션의 효율성이 향상됩니다.

다음과 같은 방법으로 폼에 메소드를 추가합니다.

1 폼 객체에 메소드 선언을 추가합니다.

   폼 객체의 선언 마지막에 있는 **public**이나 **private** 부분에 선언을 추가할 수 있습니다. 코드가 단순히 일부 이벤트 처리의 세부 사항을 공유하는 경우 공유 메소드를 **private**으로 하는 것이 가장 안전합니다.

2 폼의 유닛에 대한 .cpp 파일에 메소드 구현을 작성합니다.

메소드 구현의 헤더는 해당 선언과 정확히 일치해야 합니다. 즉, 동일한 매개변수가 동일한 순서로 사용되어야 합니다.

다음 코드는 *DrawShape*라는 폼에 메소드를 추가하고 각 핸들러에서 추가된 메소드를 호출합니다. 먼저 *DrawShape*의 선언이 폼 객체의 선언에 다음과 같은 방법으로 추가됩니다.

```
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int
X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X,int Y);
private:// User declarations
  void __fastcall DrawShape(TPoint TopLeft, TPoint BottomRight, TPenMode
AMode);
public:// User declarations
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  //field to track whether button was pressed
  TPoint Origin, MovePt;  // fields to store points
  TDrawingTool DrawingTool;  // field to hold current tool
};
```

그런 다음 유닛에 대한 .cpp 파일에 다음과 같이 *DrawShape* 구현을 작성합니다.

```
void __fastcall TForm1::DrawShape(TPoint TopLeft, TPoint BottomRight,
  TPenMode AMode)
{
  Canvas->Pen->Mode = AMode;
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(TopLeft.x, TopLeft.y);
      Canvas->LineTo(BottomRight.x, BottomRight.y);
      break;
    case dtRectangle:
      Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x,
BottomRight.y);
      break;
    case dtEllipse:
      Canvas->Ellipse(TopLeft.x, TopLeft.y, BottomRight.x,
BottomRight.y);
      break;
    case dtRoundRect:
      Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x,
BottomRight.y,
                  (TopLeft.x - BottomRight.x)/2,(TopLeft.y -
BottomRight.y)/2);
      break;
  }
}
```

다른 이벤트 핸들러는 *DrawShape*를 호출하도록 수정됩니다.

```
void __fastcall TForm1::FormMouseUp(TObject *Sender)
{
  DrawShape(Origin, Point(X,Y), pmCopy); // draw the final shape
  Drawing = false;
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    DrawShape(Origin, MovePt, pmNotXor); // erase previous shape
    MovePt = Point(X, Y);
    DrawShape(Origin, MovePt, pmNotXor); // draw current shape
  }
}
```

그래픽에서 그리기

애플리케이션의 그래픽 객체를 처리하는 데에는 컴포넌트가 필요 없습니다. 화면에 아무 것도 그리지 않고 그래픽 객체를 생성, 저장 및 소멸시킬 수 있습니다. 사실 애플리케이션에서 폼에 직접 그리는 경우는 거의 없습니다. 오히려 애플리케이션은 그래픽에서 작동하고 이미지 컨트롤 컴포넌트를 사용하여 폼에 그래픽을 표시합니다.

일단 애플리케이션의 드로잉을 이미지 컨트롤의 그래픽으로 이동하면 그래픽 객체에 대한 인쇄, 클립보드, 로딩 및 저장 작업을 쉽게 추가할 수 있습니다. 그래픽 객체는 비트맵 파일, 드로잉, 아이콘 또는 jpeg 그래픽과 같이 설치되어 있는 다른 그래픽 클래스일 수 있습니다.

참고 *TBitmap* 캔버스와 같은 오프스크린 이미지에서 그리는 것이므로 이미지는 컨트롤이 비트맵에서 컨트롤의 캔버스에 복사되어야 표시됩니다. 즉, 비트맵을 그려서 이미지 컨트롤에 할당하면 이미지는 컨트롤이 해당 그림 메시지를 처리할 수 있는 경우에만 나타납니다. 그러나 컨트롤의 캔버스 속성에 직접 그리면 그림 객체가 즉시 표시됩니다.

스크롤할 수 있는 그래픽 만들기

그래픽은 폼과 크기가 같지 않아도 됩니다. 즉 폼보다 더 크거나 더 작아도 됩니다. 폼에 스크롤 박스 컨트롤을 추가하고 내부에 그래픽 이미지를 배치하면 폼보다 더 크거나 심지어 화면보다 큰 그래픽을 표시할 수 있습니다. 스크롤할 수 있는 그래픽을 추가하려면 먼저 *TScrollBox* 컴포넌트를 추가한 다음 이미지 컨트롤을 추가합니다.

이미지 컨트롤 추가

이미지 컨트롤은 비트맵 객체를 표시할 수 있는 컨테이너 컴포넌트입니다. 항상 표시할 필요가 없거나 애플리케이션이 다른 그림을 생성하는 데 사용할 필요가 있는 비트맵은 이미지 컨트롤을 사용하여 보관합니다.

참고 6-11 페이지의 "컨트롤에 그래픽 추가"에서는 컨트롤에서 그래픽을 사용하는 방법을 보여 줍니다.

컨트롤 배치

이미지 컨트롤은 폼의 어느 곳에나 배치할 수 있습니다. 그림에 자신의 크기를 맞추는 이미지 컨트롤 기능을 사용할 경우, 왼쪽 위 모서리만 설정하면 됩니다. 이미지 컨트롤이 비트맵에서 보이지 않는 표시자인 경우에는 넌비주얼(nonvisual) 컴포넌트와 마찬가지로 아무 곳에나 배치할 수 있습니다.

폼의 클라이언트 영역에 이미 정렬되어 있는 스크롤 박스에 이미지 컨트롤을 놓는 경우 이미지의 그림 중 오프스크린 부분에 액세스할 수 있도록 스크롤 박스에서 스크롤 막대를 추가합니다. 그런 다음 이미지 컨트롤의 속성을 설정합니다.

초기 비트맵 크기 설정

이미지 컨트롤을 배치하면 이미지 컨트롤은 단지 컨테이너에 불과합니다. 그러나 디자인 타임에 이미지 컨트롤의 *Picture* 속성을 설정하여 정적 그래픽을 포함할 수 있습니다. 그리고 10-19 페이지의 "그래픽 파일 로드 및 저장"에서 설명한 대로 컨트롤은 런타임 시 파일에서 컨트롤에 있는 그림을 로드할 수도 있습니다.

다음과 같은 방법으로 애플리케이션을 시작할 때 비어 있는 비트맵을 만듭니다.

1 이미지가 들어 있는 폼의 *OnCreate* 이벤트에 핸들러를 연결합니다.

2 비트맵 객체를 만들고 비트맵 객체를 이미지 컨트롤의 *Picture->Graphic* 속성에 할당합니다.

이 예제에서 이미지는 애플리케이션의 메인 폼인 *Form1*에 있으므로 코드는 다음과 같이 *Form1*의 *OnCreate* 이벤트에 핸들러를 연결합니다.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Graphics::TBitmap *Bitmap = new Graphics::TBitmap();   // create the
bitmap object
  Bitmap->Width = 200;               // assign the initial width...
  Bitmap->Height = 200;              // ...and the initial height
  Image->Picture->Graphic = Bitmap;  // assign the bitmap to the image
control
  delete Bitmap;                     // free the bitmap object
}
```

그림의 *Graphic* 속성에 비트맵을 할당하면 그림 객체에 비트맵이 복사됩니다. 그러나 그림 객체는 비트맵의 소유권을 갖지 않으므로 할당한 다음 해제해야 합니다.

지금 애플리케이션을 실행하는 경우 폼의 클라이언트 영역에 비트맵을 나타내는 흰색 영역이 나타납니다. 클라이언트 영역에 전체 이미지가 표시되지 않도록 윈도우 크기를 지정하면 이미지의 나머지 부분을 표시할 수 있도록 스크롤 박스에서 스크롤 막대를 자동으로 표시하는 것을 볼 수 있습니다. 그러나 이미지 위에서 그리는 경우 그래픽이 나타나지 않습니다. 이미지와 스크롤 박스 뒤에 있는 폼에서 애플리케이션이 아직 그리기 작업을 진행 중이기 때문입니다.

비트맵에 그리기

비트맵에서 그리려면 이미지 컨트롤의 캔버스를 사용하고 이미지 컨트롤의 해당 이벤트에 마우스 이벤트 핸들러를 연결하십시오. 일반적으로 채우기, 사각형, 다각선 등 영역 작업을 사용합니다. 영역 작업은 빠르고 효율적인 그리기 방법입니다.

각 픽셀에 액세스해야 할 때 이미지를 그리는 효율적인 방법은 비트맵 *ScanLine* 속성을 사용하는 것입니다. 일반적인 용도로 사용하는 경우 비트맵 픽셀 형식을 24비트로 설정한 다음 *ScanLine*에서 반환된 포인터를 RGB 배열로 처리합니다. 그렇지 않으면 *ScanLine* 속성의 원시

형식을 알아야 합니다. 이 예제는 *ScanLine*을 사용하여 픽셀을 한 번에 한 줄씩 가져오는 방법을 보여 줍니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Graphics::TBitmap *pBitmap = new Graphics::TBitmap();
// This example shows drawing directly to the Bitmap
  Byte *ptr;
  try
  {
    pBitmap->LoadFromFile("C:\\Program Files\\Borland\\CBuilder\\Images\\
Splash\\256color\\factory.bmp ");
    for (int y = 0; y <pBitmap->Height; y++)
    {
      ptr = pBitmap->ScanLine[y];
      for (int x = 0; x <pBitmap->Width; x++)
        ptr[x] = (Byte)y;
    }
    Canvas->Draw(0,0,pBitmap);
  }
  catch (...)
  {
    ShowMessage("Could not load or alter bitmap");
  }
  delete pBitmap;
}
```

CLX 크로스 플랫폼 애플리케이션의 경우 Windows 특정 코드 및 VCL 특정 코드를 변경하여 Linux에서도 애플리케이션을 실행할 수 있습니다. 예를 들어, Linux에서는 경로 이름에 슬래시(/)를 구분 기호로 사용합니다. CLX 및 크로스 플랫폼 애플리케이션에 대한 자세한 내용은 14장, "크로스 플랫폼 애플리케이션 개발"을 참조하십시오.

그래픽 파일 로드 및 저장

하나의 애플리케이션이 실행되는 동안에만 존재하는 그래픽 이미지는 별로 가치가 없습니다. 동일한 그림을 항상 사용해야 하거나 만들어진 그림을 나중에 사용할 수 있도록 저장해야 할 경우가 많습니다. 이미지 컴포넌트를 사용하면 쉽게 파일에서 그림을 로드하고 다시 저장할 수 있습니다.

그래픽 이미지를 로드 및 저장하고 바꾸는 데 사용되는 컴포넌트는 비트맵 파일, 메타파일, 문자 모양 등 많은 그래픽 형식을 지원합니다. 또한 CLX 컴포넌트는 설치 가능한 그래픽 클래스를 지원합니다.

그래픽 파일을 로드하고 저장하는 방법은 다른 파일과 유사하며 다음 단원에서 설명합니다.

- 파일에서 그림 로드
- 그림을 파일로 저장
- 그림 바꾸기

파일에서 그림 로드

애플리케이션에서 그림을 수정해야 하거나 애플리케이션 외부에 그림을 저장하려는 경우 파일에서 그림을 로드하는 기능을 제공해야 합니다. 그러면 다른 개발자나 애플리케이션이 그림을 수정할 수 있습니다.

그래픽 파일을 이미지 컨트롤로 로드하려면 이미지 컨트롤 *Picture* 객체의 *LoadFromFile* 메소드를 호출하십시오.

다음 코드는 그림 파일 열기 다이얼로그 박스에서 파일 이름을 가져온 다음 해당 파일을 *Image* 라는 이름의 이미지 컨트롤로 로드합니다.

```
void __fastcall TForm1::Open1Click(TObject *Sender)
{
  if (OpenPictureDialog1->Execute())
  {
    CurrentFile = OpenPictureDialog1->FileName;
    Image->Picture->LoadFromFile(CurrentFile);
  }
}
```

파일에 그림 저장

그림 객체는 여러 형식으로 그래픽을 로드하고 저장할 수 있으며, 자신의 그래픽 파일 형식을 만들고 등록하여 그림 객체가 이러한 형식도 로드하고 저장할 수 있게 할 수도 있습니다.

파일에 이미지 컨트롤의 내용을 저장하려면 이미지 컨트롤 *Picture* 객체의 *SaveToFile* 메소드를 호출하십시오.

SaveToFile 메소드에는 저장할 파일의 이름이 필요합니다. 그림을 새로 만든 경우 파일 이름이 없거나 사용자가 기존 그림을 다른 파일에 저장하려고 할 수도 있습니다. 두 경우 모두, 다음 단원의 설명과 같이 애플리케이션이 파일을 저장하기 전에 사용자가 파일 이름을 지정해야 합니다.

File|Save 및 File|Save As 메뉴 항목에 연결된 다음의 이벤트 핸들러 쌍은 각각, 이름이 지정된 파일 다시 저장, 이름이 지정되지 않은 파일 저장 및 기존 파일을 새 이름으로 저장하는 기능을 처리합니다.

```
void __fastcall TForm1::Save1Click(TObject *Sender)
{
  if (!CurrentFile.IsEmpty())
    Image->Picture->SaveToFile(CurrentFile);   // save if already named
else SaveAs1Click(Sender);                     // otherwise get a name
}
void __fastcall TForm1::SaveAs1Click(TObject *Sender)
{
  if (SaveDialog1->Execute())              // get a file name
  {
    CurrentFile = SaveDialog1->FileName;  // save user-specified name
    Save1Click(Sender);                   // then save normally
  }
}
```

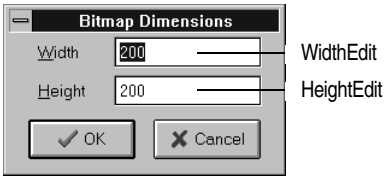
그림 바꾸기

이미지 컨트롤의 그림은 언제든지 바꿀 수 있습니다.이미 그래픽이 있는 그림에 새 그래픽을 할당하면 기존 그래픽이 새 그래픽으로 바뀝니다.

이미지 컨트롤에서 그림을 바꾸려면 이미지 컨트롤의 *Picture* 객체에 새 그래픽을 할당하십시오.

새 그래픽을 만드는 경우 초기 그래픽을 만드는 것과 동일한 과정을 거치지만(10-18페이지의 "초기 비트맵 크기 설정" 참조), 사용자가 초기 그래픽에 사용된 디폴트 크기가 아닌 새 크기를 선택할 수 있는 방법을 제공해야 합니다. 크기 선택 옵션을 제공하는 쉬운 방법은 그림 10.1과 같이 다이얼로그 박스를 표시하는 것입니다.

그림 10.1 BMPDlg 유닛의 Bitmap Dimensions 다이얼로그 박스



위 다이얼로그 박스는 EXAMPLES\DOC\GRAPHEX 디렉토리에 있는 *GraphEx* 프로젝트에 포함된 *BMPDlg* 유닛에서 만들어진 것입니다.

프로젝트에 이러한 다이얼로그 박스를 사용하여 메인 폼의 .cpp 파일에 BMPDlg.hpp에 대한 include 문을 추가합니다. 그런 다음 File|New 메뉴 항목의 *OnClick* 이벤트에 이벤트 핸들러를 연결할 수 있습니다. 예제는 다음과 같습니다.

```
void __fastcall TForm1::New1Click(TObject *Sender)
{
     Graphics::TBitmap *Bitmap;
  // make sure focus is on width field
     NewBMPForm->ActiveControl = NewBMPForm->WidthEdit;
  // initialize to current dimensions as default ...
     NewBMPForm->WidthEdit->Text = IntToStr(Image->Picture->Graphic-
>Width);
     NewBMPForm->HeightEdit->Text = IntToStr(Image->Picture->Graphic-
>Height);
    if (NewBMPForm->ShowModal() != IDCANCEL){        // if user does not
cancel dialog...
        Bitmap = new Graphics::TBitmap();             // create a new
bitmap object
       // use specified dimensions
       Bitmap->Width = StrToInt(NewBMPForm->WidthEdit->Text);
        Bitmap->Height = StrToInt(NewBMPForm->HeightEdit->Text);
        Image->Picture->Graphic = Bitmap;            // replace graphic
with new bitmap
        CurrentFile = EmptyStr;                      //indicate unnamed
file
        delete Bitmap;
     }
}
```

참고 그림 객체의 *Graphic* 속성에 새 비트맵을 할당하면 그림 객체가 새 그래픽을 복사하지만 새 그래픽에 대한 소유권을 갖지는 않습니다. 그림 객체는 자신의 내부 그래픽 객체를 유지합니다. 이로 인해 이전 코드는 비트맵을 할당한 다음 비트맵 객체를 해제합니다.

그래픽에서 클립보드 사용

Windows 클립보드를 사용하여 애플리케이션 내에서 그래픽을 복사하고 붙여넣거나 다른 애플리케이션과 그래픽을 교환할 수 있습니다. VCL의 클립보드 객체를 사용하면 그래픽을 포함한 여러 종류의 정보를 쉽게 처리할 수 있습니다.

애플리케이션에서 클립보드 객체를 사용하려면 클립보드 데이터를 액세스하는 데 필요한 모든 .cpp 파일에 Clipbrd.hpp에 대한 include 문을 추가해야 합니다.

크로스 플랫폼 애플리케이션의 경우, CLX를 사용할 때 클립보드에 저장된 데이터는 연결된 *TStream* 객체와 함께 MIME 타입으로 저장됩니다. 다음과 같은 MIME 타입에 대해 CLX는 미리 정의된 상수를 제공합니다.

표 10.4 CLX MIME 타입과 상수

MIME 타입	CLX 상수
'image/delphi.bitmap'	SDelphiBitmap
'image/delphi.component'	SDelphiComponent
'image/delphi.picture'	SDelphiPicture
'image/delphi.drawing'	SDelphiDrawing

클립보드에 그래픽 복사

이미지 컨트롤의 컨텐트를 포함하는 모든 그림을 클립보드에 복사할 수 있습니다. 클립보드에 저장된 그림은 모든 애플리케이션에서 사용할 수 있습니다.

클립보드에 그림을 복사하려면 *Assign* 메소드를 사용하여 클립보드 객체에 그림을 할당하십시오.

다음 코드는 Edit|Copy 메뉴 항목 클릭에 응답하여 *Image*라는 이미지 컨트롤에서 클립보드로 그림을 복사하는 방법을 보여 줍니다.

```
void __fastcall TForm1::Copy1Click(TObject *Sender)
{
  Clipboard()->Assign(Image->Picture);
}
```

그림을 잘라내어 클립보드에 넣기

그림을 잘라 클립보드에 넣는 것은 그림을 복사하는 것과 같지만 잘라내기의 경우 원본에서 그래픽을 지웁니다.

그림에서 그래픽을 잘라내어 클립보드에 넣으려면 먼저 클립보드에 그래픽을 복사한 다음 원본을 지우십시오.

대부분의 경우 잘라내기의 유일한 문제점은 원본 이미지가 지워졌다는 것을 보여주는 방법에 있습니다. Edit|Cut 메뉴 항목의 *OnClick* 이벤트에 이벤트 핸들러를 연결하는 다음의 코드에서처럼 해당 영역을 하얀색으로 설정하는 것이 일반적인 해결 방법입니다.

```
void __fastcall TForm1::Cut1Click(TObject *Sender)
{
  TRect ARect;
  Copy1Click(Sender);          // copy picture to clipboard
  Image->Canvas->CopyMode = cmWhiteness; // copy everything as white
  ARect = Rect(0, 0, Image->Width, Image->Height); // get dimensions of
image
  Image->Canvas->CopyRect(ARect, Image->Canvas, ARect); // copy bitmap
over self
  Image->Canvas->CopyMode = cmSrcCopy; // restore default mode
}
```

클립보드에서 그림 붙여넣기

클립보드에 비트맵 그래픽이 들어 있는 경우 이미지 컨트롤과 폼의 표면을 비롯한 모든 이미지 객체에 클립보드를 붙여 넣을 수 있습니다.

다음과 같은 방법으로 클립보드에서 그래픽을 붙여 넣습니다.

1 VCL을 사용하는 경우 클립보드의 *HasFormat* 메소드를 호출하고, CLX를 사용하는 경우에는 *Provides* 메소드를 호출하여 클립보드가 그래픽을 포함하고 있는지 확인합니다.

   HasFormat(CLX에서는 *Provides*)은 부울 함수입니다. 클립보드에 매개변수에서 지정한 타입의 항목이 들어 있는 경우 **true**를 반환합니다. Windows 플랫폼에서 그래픽을 테스트하려면 *CF_BITMAP*을 전달합니다. 크로스 플랫폼 애플리케이션의 경우 *SDelphiBitmap*을 전달합니다.

2 대상에 클립보드를 할당합니다.

VCL 다음 VCL 코드는 Edit|Paste 메뉴 항목 클릭에 응답하여 클립보드에서 이미지 컨트롤로 그림을 붙여 넣는 방법을 보여 줍니다.

```
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
  Graphics::TBitmap *Bitmap;
  if (Clipboard()->HasFormat(CF_BITMAP)){
   Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

CLX 크로스 플랫폼 개발을 위한, CLX에서의 동일한 예제는 다음과 같습니다.

```
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
  QGraphics::TBitmap *Bitmap;
  if (Clipboard()->Provides(SDelphiBitmap)){
   Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

클립보드의 그래픽은 이 애플리케이션에서 만들어졌거나 Microsoft Paint와 같은 다른 애플리케이션에서 복사한 것일 수 있습니다. 클립보드에 지원하는 형식이 없을 때에는 붙여넣기 메뉴를 사용할 수 없으므로 이런 경우에는 클립보드 형식을 확인할 필요가 없습니다.

Rubber banding 예제

이 예제에서는 사용자가 런타임 시 그래픽을 그릴 때 마우스 이동을 추적하는 그래픽 애플리케이션에서의 "rubber banding" 효과를 구현하는 방법에 대한 자세한 내용을 설명합니다. 이 단원의 예제 코드는 Examples\Doc\GraphEx 디렉토리에 있는 예제 애플리케이션에서 발췌한 것입니다. 애플리케이션에서는 클릭과 끌기에 응답하여 윈도우 캔버스에 선과 도형을 그립니다. 즉, 마우스 버튼을 누르면 그리기를 시작하고 버튼을 릴리스하면 그리기를 끝냅니다.

처음에는 예제 코드에서 메인 폼의 표면에 그리는 방법을 보여 줍니다. 이후의 예제에서는 비트맵에 그리는 방법을 보여 줍니다.

다음 항목에서는 해당 예제를 설명합니다.

- 마우스에 응답
- 마우스 동작을 추적하기 위해 폼 객체에 필드 추가
- 선 그리기 다듬기

마우스에 응답

애플리케이션은 마우스 버튼 다운, 마우스 이동, 마우스 버튼 업과 같은 마우스 동작에 응답할 수 있습니다. 또한 modal 다이얼로그 박스에서 *Enter* 키를 누르는 경우처럼 키 입력으로 생성할 수 있는 클릭(한 위치에서 완전히 눌렀다가 놓기)에도 응답할 수 있습니다.

이 단원에서는 다음과 같은 내용을 다룹니다.

- 마우스 이벤트의 내용
- 마우스 다운 동작에 응답
- 마우스 업 동작에 응답
- 마우스 이동에 응답

마우스 이벤트의 내용

C++Builder에는 *OnMouse Down* 이벤트, *OnMouseMove* 이벤트, *OnMouseUp* 이벤트 등 세 개의 마우스 이벤트가 있습니다.

애플리케이션이 마우스 동작을 감지하면 해당 이벤트에 대해 정의된 이벤트 핸들러를 호출하고 다섯 개의 매개변수를 전달합니다. 이 매개변수의 정보를 사용하여 이벤트에 대한 응답을 사용자 정의합니다. 다섯 개의 매개변수는 다음과 같습니다.

표 10.5 마우스 이벤트 매개변수

매개변수	의미
Sender	마우스 동작을 감지한 객체입니다.
Button	버튼 매개변수는 관련된 마우스 버튼을 나타내며 *mbLeft*, *mbMiddle*, 또는 *mbRight*가 있습니다.
Shift	마우스 동작 시 *Alt*, *Ctrl* 및 *Shift* 키의 상태를 나타냅니다.
X, *Y*	이벤트가 발생한 좌표입니다.

어떤 마우스 버튼이 이벤트를 유발했는지 알려면 대부분 마우스 이벤트 핸들러에 반환된 좌표만 있으면 되지만 경우에 따라 *Button*도 확인해야 합니다.

참고 C++Builder는 Microsoft Windows와 같은 기준을 사용하여 눌러진 마우스 버튼을 확인합니다. 따라서 디폴트로 설정된 "기본" 마우스 버튼과 "보조" 마우스 버튼을 서로 바꿔 오른쪽 마우스 버튼이 기본 버튼이 되면 기본(오른쪽) 버튼을 클릭했을 때 *mbLeft*가 *Button* 매개변수의 값으로 기록됩니다.

마우스 다운 동작에 응답

사용자가 마우스 버튼을 누를 때마다 *OnMouseDown* 이벤트가 포인터가 가리키는 객체로 전달됩니다. 그러면 객체는 이벤트에 응답할 수 있습니다.

마우스 다운 동작에 응답하려면, *OnMouseDown* 이벤트에 이벤트 핸들러를 연결합니다.

C++Builder에서는 다음과 같은 방법으로 마우스 다운 이벤트에 대한 비어 있는 핸들러를 폼에 생성합니다.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{

}
```

마우스 다운 동작에 응답

다음 코드를 사용하면 마우스로 클릭한 폼의 해당 위치에 'Here!' 문자열이 표시됩니다.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->TextOut(X, Y, "Here!");// write text at (X, Y)
}
```

애플리케이션이 실행되면 폼에 마우스 커서를 두고 마우스 버튼을 눌러 클릭한 지점에 "Here!" 문자열이 나타나게 할 수 있습니다. 다음 코드는 사용자가 버튼을 누른 좌표에 현재 그리기 위치를 설정합니다.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(X, Y);// set pen position
}
```

이제 마우스 버튼을 누르면 펜 위치가 설정되고 선의 시작점이 설정됩니다. 버튼을 놓는 지점까지 선을 그리려면 마우스 업 이벤트에 응답해야 합니다.

마우스 업 동작에 응답

OnMouseUp 이벤트는 마우스 버튼을 놓을 때마다 발생합니다. 사용자가 버튼을 누를 때 마우스 커서가 가리키는 객체로 이벤트가 전달되지만 버튼을 놓을 때 커서가 가리키는 객체가 동일할 필요는 없습니다. 이러한 방식을 사용하면 선이 폼의 테두리를 넘은 것처럼 선을 그릴 수 있습니다.

마우스 업 동작에 응답하려면 *OnMouseUp* 이벤트의 핸들러를 정의해야 합니다.

다음은 마우스 버튼을 놓는 지점까지 선을 그리는 간단한 *OnMouseUp* 이벤트 핸들러입니다.

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);//draw line from PenPos to (X, Y)
}
```

이 코드를 통해 사용자는 클릭, 끌기 그리고 릴리스를 사용하여 선을 그릴 수 있습니다. 이 경우에는 사용자가 마우스 버튼을 놓을 때까지 선을 볼 수 없습니다.

마우스 이동에 응답

OnMouseMove 이벤트는 사용자가 마우스를 이동할 때 정기적으로 발생합니다. 사용자가 버튼을 누르면 이벤트가 마우스 포인터 아래에 있던 객체로 전달됩니다. 이를 통해 마우스가 움직이는 동안 임시 선을 그림으로써 사용자에게 중간 피드백을 줄 수 있습니다.

마우스 이동에 응답하려면 *OnMouseMove* 이벤트에 대한 이벤트 핸들러를 정의해야 합니다. 다음 예제에서는 마우스 이동 이벤트를 사용하여 사용자가 마우스 버튼을 누르고 있는 동안 폼에 중간 도형을 그려서 사용자에게 중간 피드백을 제공합니다. *OnMouseMove* 이벤트 핸들러는 *OnMouseMove* 이벤트가 발생한 위치까지 폼에 선을 그립니다.

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);// draw line to current position
}
```

위의 코드를 사용하면 마우스 버튼을 누르기 전에도, 마우스를 폼 위로 이동하면 마우스를 따라 선을 그리게 됩니다.

마우스 이동 이벤트는 마우스 버튼을 누르지 않았을 때에도 발생합니다.

눌러진 마우스 버튼이 있는지 추적하려면 폼 객체에 객체 필드를 추가해야 합니다.

마우스 동작 추적을 위해 폼 객체에 필드 추가

눌러진 마우스 버튼이 있는지 추적하려면 폼 객체에 객체 필드를 추가해야 합니다. 폼에 컴포넌트를 추가할 때 C++Builder는 폼 객체에 해당 컴포넌트를 나타내는 필드를 추가하여 해당 필드 이름으로 컴포넌트를 참조할 수 있도록 합니다. 폼 유닛의 헤더 파일에서 타입 선언을 편집하여 폼에 직접 만든 필드를 추가할 수도 있습니다.

다음 예제에서 폼은 사용자가 마우스 버튼을 눌렀는지 여부를 추적해야 합니다. 그렇게 하려면 사용자가 마우스 버튼을 누를 때 폼에서 부울 필드를 추가하고 해당 값을 설정합니다.

객체에 필드를 추가하려면 선언 아래의 **public** 지시어 뒤에 필드 식별자와 타입을 지정하여 객체 타입 정의를 편집합니다.

C++Builder는 **public** 지시어 앞의 모든 선언을 "소유"합니다. 그러므로 public 지시어 앞에 컨트롤을 나타내는 필드와 이벤트에 응답하는 메소드를 넣습니다.

다음 코드는 폼 객체의 선언에서 폼에 *부울* 타입의 *Drawing*이라는 필드를 제공합니다. 또한 TPoint 타입의 *Origin* 및 *MovePt* 지점을 저장하는 두 개의 필드도 추가합니다.

```
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:// User declarations
public:// User declarations
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  //field to track whether button was pressed
   TPoint Origin, MovePt;  // fields to store points
};
```

그릴 것인지 여부를 추적하는 *Drawing* 필드가 있으면 다음과 같은 방법으로 사용자가 마우스 버튼을 누를 때 **true**로 설정하고 릴리스할 때 **false**로 설정합니다.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
}

void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);      // draw line from PenPos to (X, Y)
  Drawing = false;           // clear the Drawing flag
}
```

그런 다음 다음과 같은 방법으로 *OnMouseMove* 이벤트 핸들러를 수정하여 *Drawing*이 **true**일 때에만 그리게 할 수 있습니다.

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
    Canvas->LineTo(X, Y);// only draw if mouse is down
}
```

이렇게 하면 마우스 다운과 마우스 업 이벤트 간에만 그릴 수 있지만 이 때 직선 대신 마우스 이동을 추적하는 자유 곡선이 나타납니다.

이 경우에는 마우스를 이동할 때마다 마우스 이동 이벤트 핸들러가 펜 위치를 이동시키는 *LineTo*를 호출하여 마우스 버튼을 놓을 때 직선이 시작해야 하는 지점을 잃게 된다는 문제가 있습니다.

선 그리기 다듬기

다양한 지점을 추적하는 필드를 적절히 사용하면 애플리케이션의 선 그리기를 다듬을 수 있습니다.

시작 지점 추적

선을 그릴 때 *Origin* 필드를 사용하여 선이 시작되는 지점을 추적합니다.

*Origin*은 마우스 다운 이벤트가 발생하는 지점으로 설정되어야만 마우스 업 이벤트 핸들러가 *Origin*을 사용하여 다음 코드처럼 선의 시작 위치를 지정할 수 있습니다.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
  Canvas->LineTo(X, Y);                // draw line from PenPos to (X, Y)
  Drawing = false;                     // clear the Drawing flag
}
```

이렇게 변경하면 애플리케이션에서는 마지막 선을 다시 그리지만 중간 동작을 그리지 않습니다. 즉 애플리케이션이 "rubber banding"을 아직 지원하지 않는 것입니다.

이동 추적

이 예제의 문제점은 현재 작성된 *OnMouseMove* 이벤트 핸들러가 시작 위치로부터가 아니라 마지막 *마우스 위치*로부터 현재 마우스 위치로 선을 그린다는 것입니다. 시작 지점으로 그리기 위치를 이동한 다음 현재 마우스 위치로 그리면 이 문제를 해결할 수 있습니다.

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(X, Y);
  }
}
```

위의 코드는 현재 마우스 위치를 추적하지만 중간 선이 사라지지 않으므로 마지막 선을 거의 볼 수 없습니다. 이 예제에서는 이전의 선이 있던 위치를 계속 추적하여 다음 선을 그리기 전에 선을 하나씩 지워야 합니다. *MovePt* 필드를 사용하여 선을 지우는 작업을 할 수 있습니다.

*MovePt*는 각 중간 선의 끝점으로 설정되어야 하므로 *MovePt*와 *Origin*을 함께 사용하여 다음에 선이 그려질 때 해당 선을 지웁니다.

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton
Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
```

```
  MovePt = Point(X, Y);      // record last endpoint
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(MovePt.x, MovePt.y);  // erase old line
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
again
    Canvas->LineTo(X, Y);                // draw new line
  }
  MovePt = Point(X, Y);      // record new endpoint
  Canvas->Pen->Mode = pmCopy;
}
```

이제 선을 그릴 때 "rubber banding" 효과가 나타납니다. 펜의 모드를 *pmNotXor*로 변경하면 펜의 모드를 사용하여 선을 배경 픽셀과 조합할 수 있습니다. 선을 지우면 픽셀을 이전 상태로 설정하게 됩니다. 선을 그린 후 펜 모드를 기본값인 *pmCopy*로 다시 변경하면 마우스 버튼을 놓을 때 펜은 마지막 그리기 작업을 준비합니다.

멀티미디어 작업

C++Builder를 사용하여 CLX나 Linux가 아닌 Windows 애플리케이션에 멀티미디어 컴포넌트를 추가할 수 있습니다. Win32 페이지에 있는 *TAnimate* 컴포넌트나 컴포넌트 팔레트의 System 페이지에 있는 *TMediaPlayer* 컴포넌트를 사용하여 멀티미디어 컴포넌트를 추가할 수 있습니다. 애니메이션 컴포넌트를 사용하여 애플리케이션에 무성 비디오 클립을 추가할 수 있습니다. 미디어 플레이어 컴포넌트를 사용하면 애플리케이션에 오디오 및/또는 비디오 클립을 추가할 수 있습니다.

TAnimate 및 *TMediaPlayer* 컴포넌트에 대한 자세한 내용은 VCL 온라인 도움말을 참조하십시오.

이 단원에서는 다음 항목에 대해 설명합니다.

- 애플리케이션에 무성 비디오 클립 추가
- 애플리케이션에 오디오 및/또는 비디오 클립 추가

애플리케이션에 무성 비디오 클립 추가

C++ Builder에 있는 애니메이션 컨트롤을 사용하여 애플리케이션에 무성 비디오 클립을 추가할 수 있습니다.

다음과 같은 방법으로 애플리케이션에 무성 비디오 클립을 추가합니다.

1 컴포넌트 팔레트의 Win32 페이지에 있는 애니메이션 아이콘을 더블 클릭합니다. 그러면 비디오 클립을 표시할 폼 윈도우에 애니메이션 컨트롤이 자동으로 들어갑니다.

2 Object Inspector를 사용하여 *Name* 속성을 선택하고 애니메이션 컨트롤에 대한 새 *이름*을 입력합니다. 애니메이션 컨트롤을 호출할 때 이 이름을 사용합니다(C++ 식별자를 명명하는 표준 규칙을 따름).

디자인 타임 속성을 설정하고 이벤트 핸들러를 만들 때는 항상 Object Inspector를 사용하여 직접 작업합니다.

3 다음 중 하나를 수행합니다.

- *Common AVI* 속성을 선택하고 드롭다운 리스트에서 사용 가능한 AVI 중 하나를 선택합니다.
- *FileName* 속성을 선택하고 생략 부호(…) 버튼을 클릭하여 로컬 또는 네트워크의 모든 사용 가능한 디렉토리에서 하나의 AVI 파일을 선택하고 Open AVI 다이얼로그 박스에서 Open을 클릭합니다.
- *ResName* 또는 *ResID* 속성을 사용하여 AVI 리소스를 선택합니다. *ResHandle*을 사용하여 *ResName* 또는 *ResID*에서 확인한 리소스가 들어있는 모듈을 나타냅니다.

이 작업을 통해 AVI 파일을 메모리로 로드합니다. *Active* 속성 또는 *Play* 메소드를 사용하여 클립을 재생할 때까지 AVI 클립의 첫 프레임을 표시하려면 *Open* 속성을 **true**로 설정합니다.

4 *Repetitions* 속성을 AVI 클립을 재생할 횟수로 설정합니다. 값이 0이면 *Stop* 메소드가 호출될 때까지 시퀀스가 반복됩니다.

5 기타 모든 애니메이션 컨트롤 설정을 변경합니다. 예를 들어, 애니메이션 컨트롤을 열 때 표시되는 첫 프레임을 변경하려는 경우 *StartFrame* 속성을 원하는 프레임 값으로 설정합니다.

6 드롭다운 리스트를 사용하여 *Active* 속성을 **true**로 설정하거나 특정 이벤트가 런타임 시 발생할 때 AVI 클립을 실행하기 위한 이벤트 핸들러를 작성합니다. 예를 들어, 버튼 객체를 클릭할 때 AVI 클립을 활성화하려면 버튼의 *OnClick* 이벤트를 작성할 때 지정하면 됩니다. 또한 *Play* 메소드를 호출하여 AVI를 재생할 시간을 지정할 수 있습니다.

참고 *Active*를 **true**로 설정한 후 폼 또는 폼에 있는 컴포넌트를 변경하면 *Active* 속성이 **false**가 되므로 속성을 **true**로 다시 설정해야 합니다. 위의 속성은 런타임 전 혹은 런타임 시 설정합니다.

무성 비디오 클립 추가 예제

개발한 애플리케이션이 시작될 때 애니메이션 로고를 첫 화면으로 표시하려 한다고 가정합니다. 로고가 다 표시되면 화면이 사라집니다.

이 예제를 실행하려면 새 프로젝트를 만들고 Unit1.cpp 파일을 Frmlogo.cpp로 저장하고 Project1.bpr 파일을 Logo.bpr로 저장합니다. 그리고 다음을 수행합니다.

1 컴포넌트 팔레트의 Win32 페이지에 있는 애니메이션 아이콘을 더블 클릭합니다.

2 Object Inspector를 사용하여 Name 속성을 *Logo1*로 설정합니다.

3 해당 FileName 속성을 선택하고 생략 부호(…) 버튼을 클릭하여 ..\Examples\MFC\General\Cmnctrls 디렉토리에서 dillo.avi 파일을 선택합니다. 그리고 나서 Open AVI 다이얼로그 박스에서 Open을 클릭합니다.

위의 작업을 마치면 dillo.avi 파일이 메모리로 로드됩니다.

4 애니메이션 컨트롤 박스를 클릭하여 폼의 오른쪽 위로 끌어 놓습니다.

5 Repetitions 속성을 5로 설정합니다.

6 폼을 클릭하여 포커스를 폼으로 가져오고 폼의 Name 속성을 *LogoForm1* 그리고 Caption 속성을 *Logo Window*로 설정합니다. 이제 폼의 높이를 줄여 애니메이션 컨트롤이 폼의 오른쪽 가운데에 오게 합니다.

7 폼의 *OnActivate* 이벤트를 더블 클릭하고 다음의 코드를 작성하여 런타임 시 폼에 포커스가 있을 때 AVI 클립을 실행합니다.

```
Logo1->Active = true;
```

8 컴포넌트 팔레트의 Standard 페이지에 있는 Label 아이콘을 더블 클릭합니다. 해당 Caption 속성을 선택하고 *Welcome to Armadillo Enterprises 4.0*을 입력합니다. 이제 해당 Font 속성을 선택하고 생략 부호(…) 버튼을 클릭한 다음 Font 다이얼로그 박스에서 Font Style을 Bold로, 크기를 18로, 색을 Navy로 선택한 다음 OK를 클릭합니다. 레이블 컨트롤을 클릭하여 폼의 가운데로 끌어 놓습니다.

9 애니메이션 컨트롤을 클릭하여 포커스를 다시 가져옵니다. *OnStop* 이벤트를 더블 클릭하고 다음 코드를 작성하여 AVI 파일이 정지할 때 폼을 닫습니다.

```
LogoForm1->Close();
```

10 Run|Run을 선택하여 애니메이션 로고 윈도우를 실행합니다.

애플리케이션에 오디오 및/또는 비디오 클립 추가

C++ Builder의 미디어 플레이어 컴포넌트를 사용하여 애플리케이션에 오디오 및/또는 비디오 클립을 추가할 수 있습니다. 컴포넌트는 미디어 장치를 열고 미디어 장치가 사용하는 오디오 및/또는 비디오 클립을 재생, 정지, 일시 정지, 녹음합니다. 이 미디어 장치는 하드웨어일 수도 있고 소프트웨어일 수도 있습니다.

참고 오디오 및 비디오 클립은 클래스 플랫폼 프로그래밍에서는 제공되지 않습니다.

다음과 같은 방법으로 애플리케이션에 오디오 및/또는 비디오 클립을 추가합니다.

1 컴포넌트 팔레트의 System 페이지에 있는 미디어 플레이어 아이콘을 더블 클릭합니다. 그러면 미디어 기능이 필요한 폼 윈도우에 미디어 플레이어 컨트롤이 자동으로 들어갑니다.

2 Object Inspector를 사용하여 *Name* 속성을 선택하고 미디어 플레이어에 대한 새 이름을 입력합니다. 미디어 플레이어 컨트롤을 호출할 때 이 이름을 사용합니다(C++ 식별자를 명명하는 표준 규칙을 따름).

디자인 타임 속성을 설정하고 이벤트 핸들러를 만들 때는 항상 Object Inspector를 사용하여 직접 작업합니다.

3 *DeviceType* 속성을 선택하고, *AutoOpen* 속성 또는 *Open* 메소드를 사용하여 열 적절한 장치 타입을 선택합니다. *DeviceType*이 dtAutoSelect인 경우 *FileName* 속성이 지정한 미디어 파일에 대한 파일 확장자에 기반하여 장치 타입이 결정됩니다. 장치 타입 및 장치 기능에 대한 자세한 내용은 표 10.6을 참조하십시오.

4 장치가 미디어를 파일에 저장하는 경우 *FileName* 속성을 사용하여 미디어 파일의 이름을 지정합니다. *FileName* 속성을 선택하고 생략 부호(…) 버튼을 클릭하여 로컬 또는 네트워크의 모든 사용 가능한 디렉토리에서 하나의 미디어 파일을 선택하고 Open 다이얼로그 박스에서 Open을 클릭합니다. 그렇지 않은 경우 런타임 시 디스크, 카세트 등 미디어가 저장된 하드웨어를 선택된 미디어 장치에 삽입합니다.

5 *AutoOpen* 속성을 **true**로 설정합니다. 이러한 방법으로 미디어 플레이어가 들어 있는 폼이 런타임 시 만들어질 때 미디어 플레이어가 지정된 장치를 자동으로 열게 됩니다. *AutoOpen*이 **false**인 경우 장치를 열기 위해서는 *Open* 메소드를 호출해야 합니다.

6 *AutoEnable* 속성을 **true**로 설정하여 런타임 시 요구 사항에 따라 미디어 플레이어를 활성화 또는 비활성화하거나 *EnabledButtons* 속성을 더블 클릭하여 각 버튼을 **true** 또는 **false**로 설정하여 버튼을 활성화하거나 비활성화합니다.

멀티미디어 장치는 사용자가 미디어 플레이어 컴포넌트의 해당 버튼을 누르면 각각 재생, 일시 정지, 정지 등의 동작을 합니다. 멀티미디어 장치는 Play, Pause, Stop, Next, Previous 등의 버튼에 해당하는 메소드를 사용하여 제어할 수도 있습니다.

7 미디어 플레이어 컨트롤 막대를 클릭하여 폼의 적절한 위치로 끌어 놓거나 *Align* 속성을 선택하여 표시되는 드롭다운 리스트에서 적절한 정렬 위치를 선택하는 방법으로 미디어 플레이어 컨트롤 막대를 폼에 배치할 수 있습니다.

런타임 시 미디어 플레이어를 보이지 않게 하려면 *Visible* 속성을 **false**로 설정하고 *Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *Start Recording*, *Eject* 중 적절한 메소드를 호출하여 장치를 제어합니다.

8 기타 모든 미디어 플레이어 컨트롤 설정을 변경합니다. 예를 들어, 미디어에 표시 윈도우가 필요하면 *Display* 속성을 미디어를 표시하는 컨트롤로 설정합니다. 장치가 다중 트랙을 사용하는 경우 *Tracks* 속성을 원하는 트랙으로 설정합니다.

표 10.6 멀티미디어 장치 타입과 기능

장치 타입	사용하는 소프트웨어/ 하드웨어	재생 대상	트랙 사용	표시 윈도우 사용
dtAVIVideo	Windows용 AVI Video Player	AVI 비디오 파일	아니오	예
dtCDAudio	Windows용 CD Audio Player 또는 CD Audio Player	CD 오디오 디스크	예	아니오
dtDAT	Digital Audio Tape Player	디지털 오디오 테이프	예	아니오
dtDigitalVideo	Windows용 Digital Video Player	AVI, MPG, MOV 파일	아니오	예
dtMMMovie	MM Movie Player	MM 동영상	아니오	예
dtOverlay	Overlay 장치	아날로그 비디오	아니오	예
dtScanner	Image Scanner	재생 기능 없음(Record의 이미지 스캔)	아니오	아니오
dtSequencer	Windows용 MIDI Sequencer	MIDI 파일	예	아니오
dtVCR	Video Cassette Recorder	비디오 카세트	아니오	예
dtWaveAudio	Windows용 Wave Audio Player	WAV 파일	아니오	아니오

오디오 및/또는 비디오 클립 추가 예제(VCL만 해당)

이 예제에서는 C++Builder 멀티미디어 광고의 AVI 비디오 클립을 실행합니다. 이 예제를 실행하려면 새 프로젝트를 만들고 Unit1.cpp 파일을 FrmAd.cpp로 저장하고 Project1.bpr 파일을 MmediaAd.bpr로 저장합니다. 그리고 다음을 수행합니다.

1. 컴포넌트 팔레트의 System 페이지에 있는 미디어 플레이어 아이콘을 더블 클릭합니다.
2. Object Inspector를 사용하여 미디어 플레이어의 Name 속성을 *VideoPlayer1*로 설정합니다.
3. DeviceType 속성을 선택하여 표시되는 드롭다운 리스트에서 dtAVIVideo를 선택합니다.
4. 해당 FileName 속성을 선택하고 생략 부호(…) 버튼을 클릭하여 ..\Examples\Coolstuf 디렉토리에서 해당 파일을 선택합니다. Open 다이얼로그 박스에서 Open을 클릭합니다.
5. AutoOpen 속성을 **true**로 Visible 속성을 **false**로 설정합니다.
6. 컴포넌트 팔레트의 Win32 페이지에 있는 애니메이션 아이콘을 더블 클릭합니다. AutoSize 속성을 **false**로, Height 속성을 *175*로, Width 속성을 *200*으로 설정합니다. 애니메이션 컨트롤을 클릭하여 폼의 왼쪽 위 모서리로 끌어 놓습니다.
7. 미디어 플레이어를 클릭하여 포커스를 다시 가져옵니다. Display 속성을 선택하여 표시되는 드롭다운 리스트에서 Animate1을 선택합니다.
8. 폼을 클릭하여 포커스를 가져오고 Name 속성을 선택하여 *C++_Ad*를 입력합니다. 이제 폼의 크기를 애니메이션 컨트롤 크기에 맞춰 조정합니다.
9. 폼의 *OnActivate* 이벤트를 더블 클릭하고 다음의 코드를 작성하여 런타임 시 폼에 포커스가 있을 때 AVI 비디오를 실행합니다.

   ```
   VideoPlayer1->Play();
   ```

10. Run|Run을 선택하여 AVI 비디오를 실행합니다.

장

11

멀티 스레드 애플리케이션 개발

C++Builder는 멀티 스레드 애플리케이션을 쉽게 작성할 수 있도록 여러 가지 객체를 제공합니다. 멀티 스레드 애플리케이션은 동시에 실행할 수 있는 여러 개의 경로가 들어 있는 애플리케이션입니다. 멀티 스레드를 사용하는 데에는 세심한 주의가 필요하지만 다음과 같은 작업을 수행하여 프로그램을 향상시킬 수 있습니다.

- **병목 현상 방지.** 스레드가 하나인 경우에는 디스크의 파일 액세스, 다른 컴퓨터와의 통신 또는 멀티미디어 컨텐트 표시 등의 느린 프로세스를 기다리는 동안 프로그램에서 모든 실행을 중지해야 합니다. CPU는 프로세스가 완료될 때까지 유휴 상태로 있습니다. 그러나 멀티 스레드를 사용하면 애플리케이션에서 하나의 스레드가 느린 프로세스의 결과를 기다리는 동안 각각의 스레드에서 다른 프로세스를 계속 실행할 수 있습니다.
- **프로그램 동작 조직화.** 경우에 따라 프로그램의 동작을 독립적으로 기능을 수행하는 여러 개의 병렬 프로세스로 조직화할 수 있습니다. 스레드를 사용하면 각 병렬 프로세스에 대해 단일 코드 섹션을 동시에 실행할 수 있습니다. 보다 중요한 작업에 CPU 시간을 더욱 배분할 수 있도록 스레드를 사용하여 여러 프로그램 작업의 우선 순위를 할당하십시오.
- **멀티프로세싱.** 프로그램을 실행하는 시스템에 다중 프로세서가 있으면 작업을 여러 스레드로 나누고 각각의 프로세서에서 동시에 실행하도록 하여 성능을 향상시킬 수 있습니다.

참고 기초가 되는 하드웨어에서 멀티프로세싱을 지원하더라도 모든 운영 체제가 진정한 멀티프로세싱을 구현하는 것은 아닙니다. 예를 들어, 기초가 되는 하드웨어에서 멀티프로세싱을 지원하지만 Windows 9x는 멀티프로세싱을 시뮬레이트할 뿐입니다.

스레드 객체 정의

대부분의 애플리케이션에서 스레드 객체를 사용하면 애플리케이션에 실행 스레드를 나타낼 수 있습니다. 스레드 객체는 가장 일반적으로 필요한 스레드 사용을 캡슐화하여 멀티 스레드 애플리케이션 생성 과정을 단순화합니다.

참고 스레드 객체를 통해 보안 어트리뷰트(attribute)나 스레드의 스택 크기를 제어할 수는 없습니다. 이를 제어하려면 Windows API *CreateThread 또는 BeginThread* 함수를 사용해야 합니다. Windows Thread API 호출 또는 *BeginThread*를 사용하는 경우에도 일부 스레드 동기화 객체와 11-7페이지의 "스레드 조정"에서 설명된 메소드를 사용하여 이점을 얻을 수 있습니다. *CreateThread* 또는 *BeginThread* 사용에 대한 자세한 내용은 Windows 온라인 도움말을 참조하십시오.

애플리케이션에서 스레드 객체를 사용하려면 *TThread*의 새 자손을 만들어야 합니다. *TThread*의 자손을 만들려면 메인 메뉴에서 File|New|Other를 선택합니다. New Items 다이얼로그 박스에서 Thread Object를 더블 클릭하고 *TMyThread*와 같은 클래스 이름을 입력합니다. 새 스레드의 이름을 지정하려면 Named Thread 체크 박스에 체크 표시를 하고 스레드 이름을 입력합니다. 스레드의 이름을 지정하면 디버깅 동안 스레드를 더 쉽게 추적할 수 있습니다. OK를 클릭하면 C++Builder는 스레드를 구현할 새로운 .cpp와 header 파일을 만듭니다. 스레드 이름 지정에 대한 자세한 내용은 11-12페이지의 "스레드 이름 지정"을 참조하십시오.

참고 클래스 이름을 필요로 하는 대부분의 IDE 다이얼로그 박스와는 달리 New Thread Object 다이얼로그 박스는 개발자가 입력한 클래스 이름 앞에 "T"를 자동으로 붙이지 않습니다.

자동으로 생성된 .cpp 파일에는 새 스레드 클래스의 스켈레톤 코드가 들어 있습니다. 스레드의 이름을 *TMyThread*로 지정한 경우 다음과 같이 나타납니다.

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended):
TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
  // ---- Place thread code here ----
}
//---------------------------------------------------------------------------
```

생성자와 *Execute* 메소드에 대한 코드를 채워야 합니다. 그 단계에 대해서는 다음 단원에서 설명합니다.

스레드 초기화

생성자를 사용하여 새 스레드 클래스를 초기화합니다. 여기서 스레드의 디폴트 우선 순위를 할당하고 스레드가 실행 종료될 때 자동으로 해제되어야 할지 여부를 나타낼 수 있습니다.

디폴트 우선 순위 할당

우선 순위는 운영 체제가 애플리케이션의 모든 스레드에 CPU 시간을 배분할 때 해당 스레드가 갖는 우선 순위를 나타냅니다. 속도가 중요한 작업을 처리할 때에는 높은 우선 순위의 스레드를 사용하고 기타 작업을 수행할 때에는 낮은 우선 순위의 스레드를 사용합니다. 스레드 객체의 우선 순위를 나타내려면 *Priority* 속성을 설정하십시오.

Windows 애플리케이션을 작성하는 경우 *Priority* 값은 표 11.1에 설명된 대로 한 기준에 따라 적용됩니다.

표 11.1 스레드 우선 순위

값	우선 순위
tpIdle	스레드는 시스템이 유휴 상태일 때만 실행됩니다. Windows에서는 다른 스레드가 *tpIdle* 우선 순위를 가진 스레드를 실행하는 것을 중단하지 않습니다.
tpLowest	스레드의 우선 순위는 보통 우선 순위보다 두 단계 낮습니다.
tpLower	스레드의 우선 순위는 보통 우선 순위보다 한 단계 낮습니다.
tpNormal	스레드는 보통 우선 순위를 가집니다.
tpHigher	스레드의 우선 순위는 보통 우선 순위보다 한 단계 높습니다.
tpHighest	스레드의 우선 순위는 보통 우선 순위보다 두 단계 높습니다.
tpTimeCritical	스레드는 가장 높은 우선 순위를 가집니다.

참고 크로스 플랫폼 애플리케이션을 작성하는 경우 Windows와 Linux에서 우선 순위를 할당하는 각각의 코드를 사용해야 합니다. Linux에서 *Priority*는 루트에서만 변경할 수 있는 스레드 정책에 종속된 숫자 값입니다. 자세한 내용은 *TThread* 및 *Priority* 온라인 도움말 CLX 버전을 참조하십시오.

경고 CPU를 많이 사용하는 작업의 스레드 우선 순위를 높이면 애플리케이션의 다른 스레드가 CPU를 사용하지 못하는 "기아" 상태가 될 수 있습니다. 외부 이벤트를 기다리는 데 대부분의 시간을 소비하는 스레드에만 높은 우선 순위를 적용합니다.

다음 코드는 다른 애플리케이션의 성능에 영향을 주지 않아야 하는 백그라운드 작업을 수행하는 우선 순위가 낮은 스레드의 생성자를 보여 줍니다.

```
//---------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended):
TThread(CreateSuspended)
{
  Priority = tpIdle;
}
//---------------------------------------------------------------------
```

스레드 해제 시기 표시

일반적으로 스레드가 작업을 마치면 간단히 해제됩니다. 이러한 경우 스레드 객체가 스스로 해제되도록 하는 것이 가장 쉽습니다. 스스로 해제되게 하려면 *FreeOnTerminate* 속성을 **true**로 설정합니다.

그러나 한 스레드의 종료 시기를 다른 스레드와 조정해야 하는 경우가 있습니다. 예를 들어, 다른 스레드에서 작업을 수행하기 전에 한 스레드가 값을 반환하기를 기다려야 할 수 있습니다. 그러려면 두 번째 스레드가 반환 값을 받은 다음 첫 번째 스레드를 해제해야 합니다. 이러한 상

황은 *FreeOnTerminate*를 **false**로 설정한 다음 두 번째 스레드에서 첫 번째 스레드를 명시적으로 해제함으로써 해결할 수 있습니다.

스레드 함수 작성

Execute 메소드가 개발자의 스레드 함수입니다. 동일한 프로세스 공간을 공유하는 점 외에는 스레드 함수를 애플리케이션에 의해 실행되는 프로그램이라고 생각할 수 있습니다. 스레드 함수를 작성하는 것은, 애플리케이션에서 다른 스레드가 사용하는 메모리를 겹쳐 쓰지 않아야 하므로, 각각의 프로그램을 작성하는 것보다 다소 까다롭습니다. 반면에 스레드가 다른 스레드와 동일한 프로세스 공간을 공유하므로 공유 메모리를 사용하여 스레드 간에 통신할 수 있습니다.

메인 VCL/CLX 스레드 사용

VCL 또는 CLX 객체 계층에서 객체를 사용할 때 해당 속성과 메소드는 스레드에 대해 안전하지 않습니다. 즉, 속성에 액세스하거나 메소드를 실행하면 다른 스레드의 동작으로부터 보호되지 않는 메모리를 사용하는 일부 동작을 수행할 수 있습니다. 따라서 메인 스레드는 VCL 및 CLX 객체를 액세스하기 위해 따로 존재합니다. 메인 스레드는 애플리케이션의 컴포넌트로부터 받은 모든 Windows 메시지를 처리하는 스레드입니다.

모든 객체가 단일 스레드에서 해당 속성에 액세스하고 메소드를 실행할 경우에는 개발자의 객체들이 서로 영향을 줄지 여부에 대해 걱정하지 않아도 됩니다. 메인 스레드를 사용하려면 필요한 동작을 수행하는 각각의 루틴을 만드십시오. 스레드의 *Synchronize* 메소드 내에서 각각의 루틴을 호출합니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TMyThread::PushTheButton(void)
{
  Button1->Click();
}
f
void __fastcall TMyThread::Execute()
{
  f
  Synchronize((TThreadMethod)PushTheButton);
  f
}
```

*Synchronize*는 메인 스레드가 메시지 루프로 들어올 때까지 기다린 다음 전달된 메소드를 실행합니다.

참고 *Synchronize*는 메시지 루프를 사용하므로 콘솔 애플리케이션에서는 작동하지 않습니다. 콘솔 애플리케이션에서 VCL 또는 CLX 객체에 대한 액세스를 보호하려면 임계 구역 같은 다른 메커니즘을 사용해야 합니다.

항상 메인 스레드를 사용할 필요는 없습니다. 일부 객체에는 스레드 인식 기능이 있습니다. 객체의 메소드가 스레드에 대해 안전하다는 것을 알 경우 *Synchronize* 메소드를 사용하지 않으면 VCL 또는 CLX 스레드가 메시지 루프로 들어올 때까지 기다릴 필요가 없으므로 성능이 향상됩니다. 다음과 같은 객체에 대해서는 *Synchronize* 메소드를 사용할 필요가 없습니다.

- 데이터 액세스 컴포넌트는 다음과 같이 스레드에 대해 안전합니다. BDE 호환 데이터셋의 경우 스레드마다 각각의 데이터베이스 세션 컴포넌트가 있어야 합니다. 한 가지 예외는 Microsoft Access 드라이버를 사용할 경우입니다. Microsoft Access 드라이버는 스레드에

대해 안전하지 않은 Microsoft 라이브러리를 사용하여 구성되었습니다. dbDirect의 경우 업체 클라이언트 라이브러리가 스레드에 대해 안전하기만 하면 dbDirect 컴포넌트는 스레드에 대해 안전합니다. ADO 및 InterbaseExpress 컴포넌트는 스레드에 대해 안전합니다.

데이터 액세스 컴포넌트를 사용할 때 *Synchronize* 메소드의 데이터 인식 컨트롤을 포함하는 모든 호출을 랩핑(wrap)해야 합니다. 그러므로 예를 들어, 데이터 소스 객체의 *DataSet* 속성을 설정하여 데이터셋에 데이터 컨트롤을 연결하는 호출을 동기화해야 하지만 데이터셋 필드의 데이터에 액세스하기 위해 동기화할 필요는 없습니다.

데이터베이스 세션과 BDE 호환 애플리케이션의 스레드를 함께 사용하는 방법에 대한 자세한 내용은 24-28페이지의 "여러 세션 관리"를 참조하십시오.

- VisualCLX 객체는 스레드에 대해 안전하지 않은 반면 DataCLX 객체는 스레드에 대해 안전합니다.
- 그래픽 객체는 스레드에 대해 안전합니다. 메인 VCL 스레드 또는 CLX 스레드를 사용하여 *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile* (VCL만 해당), *TDrawing*(CLX만 해당), 또는 *TIcon*에 액세스할 필요가 없습니다. 캔버스 객체는 TFont, TPen, TBrush, TBitmap, TDrawing, TIcon 등을 잠금으로써 *Synchronize* 메소드 외부에서 사용할 수 있습니다(11-7페이지의 "객체 잠금" 참조).
- 리스트 객체는 스레드에 대해 안전하지 않지만 *TList* 대신 스레드에 대해 안전한 버전인 *TThreadList*를 사용할 수 있습니다.

백그라운드 스레드의 실행을 메인 스레드와 동기화할 수 있도록 애플리케이션의 메인 스레드 내에서 *CheckSynchronize* 루틴을 정기적으로 호출하십시오. *CheckSynchronize*는 애플리케이션이 유휴 상태 일 때, 즉 *OnIdle* 이벤트 핸들러 등의 위치에서 호출하는 것이 가장 좋습니다. 이렇게 하면 백그라운드 스레드에서 안전하게 메소드를 호출할 수 있습니다.

스레드 로컬 변수 사용

개발자의 *Execute* 메소드 및 메소드가 호출하는 모든 루틴은 다른 C++ 루틴과 마찬가지로 자체 로컬 변수를 가집니다. 또한 이 루틴은 모든 전역 변수에도 액세스할 수 있습니다. 사실 전역 변수는 스레드 간의 통신을 위한 강력한 메커니즘을 제공합니다.

그러나 경우에 따라 해당 스레드에서 실행되는 모든 루틴에 대한 전역 변수이면서 동일한 스레드 클래스의 다른 인스턴스와는 공유되지 않는 변수를 사용하고자 할 수도 있습니다. 스레드 로컬 변수를 선언하면 위와 같은 변수를 사용할 수 있습니다. **__thread** 변경자를 변수 선언에 추가하여 변수를 스레드 로컬로 만듭니다. 예를 들면,

```
int __thread x;
```

는 애플리케이션의 각 스레드에 개별적이지만 각 스레드 내에서는 전역인 정수형 변수를 선언합니다.

__thread 변경자는 전역(파일 범위) 및 정적 변수에 대해서만 사용할 수 있습니다. Pointer 및 Function 변수는 스레드 변수가 될 수 없습니다. AnsiStrings와 같이 copy-on-write 의미론을 사용하는 타입도 스레드 변수로 사용할 수 없습니다. 런타임 초기화 및 런타임 완료를 필요로 하는 프로그램 요소는 **__thread** 타입으로 선언될 수 없습니다.

다음 선언은 런타임 초기화를 필요로 하고 따라서 오류가 있습니다.

```
int f( );
int __thread x = f( );   // illegal
```

사용자 정의 생성자 및 소멸자를 사용하는 클래스의 인스턴스화는 런터임 초기화를 필요로 하고 따라서 다음 선언은 오류가 있습니다.

```
class X  {
   X( );
   ~X( );
};
X __thread myclass;   // illegal
```

다른 스레드로 종료 확인

스레드는 *Execute* 메소드가 호출될 때 실행을 시작하고(11-11페이지의 "스레드 객체 실행" 참조) *Execute*가 종료될 때까지 계속됩니다. 이는 스레드가 특정 작업을 수행한 다음 끝나면 중지하는 모델을 보여 줍니다. 그러나 경우에 따라 어떤 애플리케이션은 일부 외부 기준을 충족할 때까지 실행되는 스레드를 필요로 합니다.

Terminated 속성을 확인함으로써 스레드가 실행을 끝내야 할 시간이라는 것을 다른 스레드를 통해 신호를 보낼 수 있습니다. 다른 스레드가 개발자의 스레드를 종료하려고 할 때 *Terminate* 메소드를 호출합니다. *Terminate*는 개발자 스레드의 *Terminated* 속성을 **true**로 설정합니다. *Terminated* 속성 확인 및 응답을 통해 *Terminate* 메소드를 구현할지 여부는 *Execute* 메소드가 결정합니다. 다음 예제는 여러 방법 중 하나를 보여 줍니다.

```
void __fastcall TMyThread::Execute()
{
  while (!Terminated)
    PerformSomeTask();
}
```

스레드 함수에서의 예외 처리

Execute 메소드는 스레드에서 발생하는 모든 예외를 감지해야 합니다. 스레드 함수에서 예외를 감지하지 못할 경우 애플리케이션에서 액세스 위반이 발생할 수 있습니다. IDE가 예외를 감지하므로 애플리케이션을 개발할 때는 예외가 분명하지 않을 수 있으나 디버거 외부에서 애플리케이션을 실행하면 예외로 인해 런타임 오류가 발생하고 애플리케이션의 실행이 중지됩니다.

스레드 함수 내부에서 발생하는 예외를 감지하려면 *Execute* 메소드의 구현에 **try**...**catch** 블록을 추가합니다.

```
void __fastcall TMyThread::Execute()
{
  try
  {
    while (!Terminated)
      PerformSomeTask();
  }
  catch (...)
  {
    // do something with exceptions
  }
}
```

지우기 코드 작성

스레드 실행이 끝났을 때 해당 스레드를 지우는 코드를 하나로 통일할 수 있습니다. 스레드가 종료되기 전에 *OnTerminate* 이벤트가 발생합니다. *OnTerminate* 이벤트 핸들러에 지우기 코드를 넣어 *Execute* 메소드 앞에 어떤 실행 경로가 오더라도 항상 실행되도록 하십시오.

OnTerminate 이벤트 핸들러는 스레드의 일부로는 실행되지 않습니다. 대신 애플리케이션에 있는 메인 VCL 스레드 또는 CLX 스레드의 컨텍스트에서 실행됩니다. 이것은 다음과 같은 두 가지 의미를 가집니다.

- 메인 VCL 스레드 또는 CLX 스레드 값을 원하는 경우를 제외하고는 *OnTerminate* 이벤트 핸들러에서는 스레드 로컬 변수를 사용할 수 없습니다.
- 다른 스레드와의 충돌에 대한 걱정 없이 *OnTerminate* 이벤트 핸들러에서 모든 컴포넌트 및 VCL 또는 CLX 객체에 액세스할 수 있습니다.

메인 VCL 스레드 또는 CLX 스레드에 대한 자세한 내용은 11-4페이지의 "메인 VCL/CLX 스레드 사용"을 참조하십시오.

스레드 조정

스레드가 실행될 때 실행되는 코드를 작성할 경우 동시에 실행될 수 있는 다른 스레드의 동작을 고려해야 합니다. 특히 두 개의 스레드가 동일한 전역 객체나 변수를 동시에 사용하지 못하도록 주의를 기울여야 합니다. 뿐만 아니라 한 스레드의 코드는 다른 스레드에 의해 수행되는 작업 결과에 따라 달라질 수 있습니다.

동시 액세스 피하기

전역 객체나 변수에 액세스할 때 다른 스레드와의 충돌을 피하려면 스레드 코드가 작업을 끝낼 때까지 다른 스레드의 실행을 막아야 합니다. 이 때 실행 중인 다른 스레드를 불필요하게 막지 않도록 주의해야 합니다. 그렇게 할 경우 성능이 심하게 저하되어 멀티 스레드를 사용하여 얻을 수 있는 대부분의 이점을 얻지 못할 수 있습니다.

객체 잠금

일부 객체에는 다른 스레드가 실행될 때 해당 객체 인스턴스를 사용하지 못하게 하는 잠금 기능이 제공되어 있습니다.

예를 들어, *TCanvas* 및 Tcanvas의 자손 등의 캔버스 객체에는 *Unlock* 메소드가 호출되기 전까지 다른 스레드가 캔버스에 액세스하지 못하도록 막아주는 *Lock* 메소드가 있습니다.

VCL와 CLX는 모두 스레드에 대해 안전한 리스트 객체인 *TThreadList*를 포함하고 있습니다. *TThreadList::LockList*를 호출하면 *UnlockList* 메소드가 호출될 때까지 실행 중인 다른 스레드가 리스트를 사용하지 못하도록 막으면서 리스트 객체를 반환합니다. *TCanvas::Lock* 또는 *TThreadList::LockList* 호출은 안전하게 중첩될 수 있습니다. 마지막 잠금 호출이 동일한 스레드 내의 해당 잠금 해제 호출과 일치해야만 잠금이 해제됩니다.

임계 구역 사용

객체가 기본 잠금 기능을 제공하지 않는 경우 임계 구역을 사용할 수 있습니다. 임계 구역은 한 번에 하나의 스레드만 들어갈 수 있게 하는 문과 같습니다. 임계 구역을 사용하려면 *TCriticalSection*의 전역 인스턴스를 만듭니다. *TCriticalSection*에는 다른 스레드가 구역을 실행하지 못하게 막는 *Acquire* 메소드와 블록을 제거하는 *Release* 메소드 등 두 개의 메소드가 있습니다.

각 임계 구역은 보호하고자 하는 전역 메모리와 연결됩니다. 해당 전역 메모리에 액세스하는 모든 스레드는 먼저 *Acquire* 메소드를 사용하여 다른 스레드가 Acquire 메소드를 사용하지 못하도록 해야 합니다. 끝나면 스레드는 다른 스레드에서 *Acquire*를 호출하여 전역 메모리에 액세스할 수 있도록 *Release* 메소드를 호출합니다.

경고 임계 구역은 모든 스레드가 임계 구역을 사용하여 연결된 전역 메모리에 액세스할 때만 작동합니다. 임계 구역을 무시하고 *Acquire*를 호출하지 않은 채 전역 메모리에 액세스하는 스레드는 동시 액세스 문제를 유발할 수 있습니다.

예를 들어, 전역 변수인 X와 Y에 대한 액세스를 막는 임계 구역 전역 변수인 *pLockXY*를 갖고 있는 애플리케이션의 경우를 생각해 보십시오. X와 Y를 사용하는 스레드는 다음과 같은 방법으로 임계 구역 호출을 사용하여 다른 스레드가 X와 Y를 사용하지 못하게 막아야 합니다.

```
pLockXY->Acquire(); // lock out other threads
try
{
  Y = sin(X);
}
__finally
{
  pLockXY->Release();
}
```

동시 읽기는 허용하고 쓰기는 배타적인 동기화 장치(multi-read exclusive-write synchronizer) 사용

임계 구역을 사용하여 전역 메모리를 보호할 경우 한 번에 단 하나의 스레드만이 해당 메모리를 사용할 수 있습니다. 특히 객체나 변수를 종종 읽기는 하지만 객체나 변수에 쓰는 경우가 거의 없는 경우에는 필요 이상으로 메모리를 보호하는 것일 수도 있습니다. 스레드가 메모리에 쓰는 경우가 아니면 동일한 메모리를 동시에 읽는 멀티 스레드에 대한 위험은 없습니다.

가끔 읽기는 하지만 스레드가 쓰는 경우가 적은 전역 메모리를 가지고 있는 경우 *TMultiReadExclusiveWriteSynchronizer*를 사용하여 보호할 수 있습니다. 이 객체는 임계 구역처럼 동작하지만 다른 스레드가 쓰고 있지만 않으면 멀티 스레드를 통해 객체가 보호하는 메모리를 읽을 수 있습니다. 스레드는 *TMultiReadExclusiveWriteSynchronizer*에서 보호하는 메모리에 배타적으로 쓸 수 있도록 액세스해야 합니다.

동시 읽기는 허용하고 쓰기는 배타적인 동기화 장치를 사용하려면 보호하려는 전역 메모리와 연결된 *TMultiReadExclusiveWriteSynchronizer*의 전역 인스턴스를 만듭니다. 이 메모리에서 읽히는 모든 스레드는 먼저 *BeginRead* 메소드를 호출해야 합니다. 그러면 *BeginRead*에서는 다른 스레드가 메모리에 쓰지 못하게 합니다. 스레드가 보호된 메모리 읽기를 끝내면 *EndRead* 메소드를 호출합니다. 보호된 메모리에 쓰는 모든 스레드는 먼저 *BeginWrite*를 호출해야 합니다. 그러면 *BeginWrite*에서는 다른 스레드가 메모리를 읽거나 쓰지 못하게 합니다. 스레드가 보호된 메모리에 쓰기를 끝내면 메모리를 읽기 위해 기다린 다른 스레드가 시작할 수 있도록 *EndWrite* 메소드를 호출합니다.

경고 동시 읽기는 허용하고 쓰기는 배타적인 동기화 장치는 임계 구역처럼 모든 스레드가 이 동기화 장치를 사용하여 연결된 전역 메모리에 액세스하는 경우에만 작동합니다. 동기화 장치를 무시하고 *BeginRead*나 *BeginWrite*를 호출하지 않은 채 전역 메모리에 액세스하는 스레드는 동시 액세스 문제를 유발합니다.

메모리 공유에 이용하는 기타 기술

VCL 또는 CLX에서 객체를 사용하는 경우에는 메인 스레드를 사용하여 코드를 실행합니다. 메인 스레드를 사용하면 객체가 다른 스레드의 VCL 또는 CLX 객체에서 사용하는 모든 메모리에 간접적으로 액세스하지 못하게 합니다. 메인 스레드에 대한 자세한 내용은 11-4페이지의 "메인 VCL/CLX 스레드 사용"을 참조하십시오.

전역 메모리를 멀티 스레드로 공유할 필요가 없을 경우 전역 변수 대신 스레드 로컬 변수 사용을 고려해 보십시오. 스레드 로컬 변수를 사용하면 개발자의 스레드가 다른 스레드를 기다리거나 잠글 필요가 없습니다. 스레드 로컬 변수에 대한 자세한 내용은 11-5페이지의 "스레드 로컬 변수 사용"을 참조하십시오.

다른 스레드 기다리기

스레드가 다른 스레드의 작업이 끝나기를 기다려야 하는 경우 스레드가 일시적으로 실행을 중지하도록 할 수 있습니다. 다른 스레드가 실행이 완전히 끝날 때까지 기다리거나 다른 스레드가 작업 완료 신호를 낼 때까지 기다릴 수 있습니다.

스레드의 실행 종료 대기

다른 스레드의 실행이 끝날 때까지 기다리려면 해당 다른 스레드의 *WaitFor* 메소드를 사용합니다. *WaitFor*는 다른 스레드가 *Execute* 메소드를 종료하거나 예외로 인한 종료를 통해 다른 스레드가 종료되면 반환됩니다. 예를 들면, 다음 코드는 리스트에 있는 객체에 액세스하기 전에 스레드 리스트 객체를 다른 스레드가 채울 때까지 기다립니다.

```
if (pListFillingThread->WaitFor())
{
  TList *pList = ThreadList1->LockList();
  for (int i = 0; i< pList->Count; i++)
    ProcessItem(pList->Items[i]);
  ThreadList1->UnlockList();
}
```

이전의 예제에서 보면 *WaitFor* 메소드가 리스트가 성공적으로 채워졌음을 나타낼 때만 리스트 항목을 액세스할 수 있습니다. 이 반환값은 기다린 스레드의 *Execute* 메소드에 의해 할당되어야 합니다. 그러나 *WaitFor*를 호출하는 스레드가 *Execute*를 호출하는 코드가 아니라 스레드 실행 결과를 알고자 하므로 *Execute* 메소드는 값을 반환하지 않습니다. 대신 *Execute* 메소드는 *ReturnValue* 속성을 설정합니다. *WaitFor* 메소드는 다른 스레드에 의해 호출될 때 *ReturnValue*를 반환합니다. 반환 값은 정수입니다. 애플리케이션이 반환 값의 의미를 결정합니다.

작업 완료 대기

간혹 특정 스레드의 실행 완료를 기다리는 대신 스레드의 일부 작업이 종료되는 것을 기다려야 할 경우도 있습니다.이 경우 이벤트 객체를 사용합니다. 이벤트 객체(*TEvent*)는 모든 스레드에서 볼 수 있는 신호처럼 동작할 수 있도록 전역 유효 범위(scope)를 사용하여 만들어야 합니다.

한 스레드에서 다른 스레드가 종속된 작업을 완료할 경우 스레드는 *TEvent::SetEvent*를 호출합니다. *SetEvent*는 신호를 켜서 기다리고 있는 다른 스레드에 작업이 완료되었음을 알립니다. 신호를 끄려면 *ResetEvent* 메소드를 사용합니다.

예를 들어, 단일 스레드가 아니라 여러 스레드의 실행이 완료되는 것을 기다려야 하는 상황을 생각해 보십시오. 어떤 스레드가 마지막으로 완료될지 모르기 때문에 단순히 스레드 하나에 *WaitFor* 메소드를 사용할 수는 없습니다. 그 대신 각 스레드가 완료될 때 카운터를 증가시키도록 할 수 있으며 이벤트를 설정하여 마지막 스레드가 모든 작업이 완료되었다는 신호를 보내게 할 수 있습니다.

다음 코드는 완료되어야 하는 모든 스레드의 마지막 *OnTerminate* 이벤트 핸들러를 보여 줍니다 . *CounterGuard* 는 멀티 스레드가 동시에 카운터를 사용하지 못하도록 막는 임계 구역 전역 객체입니다. *Counter*는 완료된 스레드의 수를 계산하는 전역 변수입니다.

```
void __fastcall TDataModule::TaskThreadTerminate(TObject *Sender)
{
  f
  CounterGuard->Acquire(); // lock the counter
  if (--Counter == 0)    // decrement the global counter
    Event1->SetEvent(); // signal if this is the last thread
  CounterGuard->Release(); // release the lock on the counter
}
```

메인 스레드는 Counter 변수를 초기화하고, 작업 스레드를 실행하며 *WaitFor* 메소드를 호출하여 모든 작업이 완료되었음을 나타내는 신호를 기다립니다. *WaitFor*는 신호가 설정될 때까지 지정된 시간 동안 기다리고 표 11.2의 값 중 하나를 반환합니다.

표 11.2 WaitFor 반환 값

값	의미
wrSignaled	이벤트 신호가 설정됩니다.
wrTimeout	신호가 설정되지 않은 상태로 지정된 시간이 경과합니다.
wrAbandoned	시간 초과 기간이 경과하기 전에 이벤트 객체가 소멸됩니다.
wrError	대기 중 오류가 발생합니다.

다음 코드는 메인 스레드가 작업 스레드를 실행하고 모든 작업이 완료되었을 때 다시 시작하는 방법을 보여 줍니다.

```
Event1->ResetEvent(); // clear the event before launching the threads
for (int i = 0; i< Counter; i++)
  new TaskThread(false); // create and launch task threads
if (Event1->WaitFor(20000) != wrSignaled)
  throw Exception;
// now continue with the main thread, all task threads have finished
```

참고 지정된 시간 후에도 이벤트 대기 중지를 원하지 않는 경우 *WaitFor* 메소드를 INFINITE의 매개변수 값에 전달합니다. INFINITE를 사용할 때에는 예상한 신호가 수신되지 않는 경우 스레드가 정지되므로 주의해야 합니다.

스레드 객체 실행

Execute 메소드를 부여하여 스레드 클래스를 구현하면 애플리케이션에서 스레드 클래스를 사용하여 *Execute* 메소드에서 코드를 실행할 수 있습니다. 스레드를 사용하려면 먼저 스레드 클래스의 인스턴스를 만듭니다. 즉시 실행되는 스레드 인스턴스를 만들거나 *Resume* 메소드를 호출할 때만 시작하도록 일시 중지된 상태에서 스레드를 작성할 수 있습니다. 즉시 시작하도록 스레드를 작성하려면 생성자의 *CreateSuspended* 매개변수를 **false**로 설정합니다. 예를 들어, 다음은 스레드를 작성하고 실행합니다.

```
TMyThread *SecondThread = new TMyThread(false); // create and run the
thread
```

경고 애플리케이션에 너무 많은 스레드를 만들지 마십시오. 멀티 스레드를 관리하는 과정에서의 오버헤드가 성능을 저하시킬 수 있습니다. 단일 프로세서 시스템의 경우 프로세스당 권장 스레드 수는 16개까지입니다. 이러한 제한은 대부분의 스레드가 외부 이벤트를 기다리는 것을 가정한 것입니다. 모든 스레드가 활성화되었을 경우에는 더 적게 사용합니다.

동일한 스레드 형식의 다중 인스턴스를 만들어 병렬 코드를 실행할 수 있습니다. 예를 들어, 일부 사용자 동작에 대한 응답으로 새 스레드 인스턴스를 실행하여 각 스레드에서 예상된 응답을 수행하게 할 수 있습니다.

디폴트 우선 순위 오버라이드

스레드가 수신해야 하는 CPU 시간의 양이 스레드 작업에 암시된 경우 생성자에 해당 우선 순위를 설정해 둡니다. 관련 내용은 11-2페이지의 "스레드 초기화"에 설명되어 있습니다. 그러나 스레드 실행 시기에 따라 스레드 우선 순위가 달라지는 경우 일시 중지된 상태에서 스레드를 만들고 우선 순위를 설정한 다음 다음과 같은 방법으로 스레드 실행을 시작합니다.

```
TMyThread *SecondThread = new TMyThread(true); // create but don't run
SecondThread->Priority = tpLower; // set the priority lower than normal
SecondThread->Resume(); // now run the thread
```

참고 크로스 플랫폼 애플리케이션을 작성하는 경우 Windows와 Linux에서 우선 순위를 할당하는 각각의 코드를 사용해야 합니다. Linux에서 *Priority*는 루트에서만 변경할 수 있는 스레드 정책에 종속된 숫자 값입니다. 자세한 내용은 *TThread* 및 *Priority* 온라인 도움말 CLX 버전을 참조하십시오.

스레드 시작 및 중지

스레드는 실행을 완료하기 전에 몇 번이고 시작하고 중지할 수 있습니다. 스레드를 일시적으로 중지하려면 스레드의 *Suspend* 메소드를 호출합니다. 스레드를 다시 시작하는 것이 안전하면 스레드의 *Resume* 메소드를 호출합니다. *Suspend*는 내부 카운터를 증가시켜 *Suspend*와 *Resume*에 대한 호출을 중첩시킬 수 있습니다. 스레드는 모든 보류 시간(suspension)이 *Resume*에 대한 호출과 일치할 경우 다시 실행됩니다.

Terminate 메소드를 호출하여 스레드가 미리 실행 종료되도록 요청할 수 있습니다. *Terminate*는 스레드의 *Terminated* 속성을 **true**로 설정합니다. *Execute* 메소드를 적절히 구현했을 경우 Execute는 *Terminated* 속성을 정기적으로 확인하고 *Terminated*가 **true**일 때 실행을 중지합니다.

멀티 스레드 애플리케이션 디버깅

멀티 스레드 애플리케이션을 디버깅할 경우 동시에 실행되는 모든 스레드의 상태를 추적하기가 어려우며 심지어는 브레이크포인트에서 중지할 때 어떤 스레드가 실행 중인지 확인하는 것조차 어려울 수 있습니다. Thread Status 박스를 사용하여 애플리케이션의 모든 스레드를 추적하고 처리할 수 있습니다. Thread Status 박스를 표시하려면 메인 메뉴에서 View | Debug Windows | Threads를 선택합니다.

브레이크포인트, 예외, 일시 중단 등의 디버그 이벤트가 발생하면 스레드 상태 뷰에서 각 스레드의 상태를 나타냅니다. Thread Status 박스를 마우스 오른쪽 버튼으로 클릭하여 해당 소스 위치를 찾는 명령에 액세스하거나 다른 스레드를 현재 스레드로 사용합니다. 스레드가 현재 사용 중인 것으로 표시되면 해당 스레드에 따라 다음 단계나 실행 작업이 결정됩니다.

Thread Status 박스는 스레드 ID로 애플리케이션의 모든 실행 스레드를 나열합니다. 스레드 객체를 사용할 경우 스레드 ID는 *ThreadID* 속성 값입니다. 스레드 객체를 사용하지 않을 경우 각 스레드의 스레드 ID는 *BeginThread* 또는 *BeginThread*에 대한 호출에 의해 반환됩니다.

Thread Status 박스에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

스레드 이름 지정

Thread Status 박스에서는 스레드 ID가 어떤 스레드를 나타내는지 알기 어렵기 때문에 개발자가 스레드 클래스의 이름을 지정할 수 있습니다. Thread Object 다이얼로그 박스에서 스레드 클래스를 만드는 경우 클래스 이름을 입력한 다음 Named Thread 체크 박스를 선택하고 스레드 이름을 입력한 다음 OK를 클릭합니다.

스레드 클래스의 이름을 지정하면 스레드 클래스에 *SetName*이라고 하는 메소드가 추가됩니다. 스레드가 실행되면 *SetName* 메소드를 제일 먼저 호출합니다.

CLX VCL 애플리케이션에서만 스레드의 이름을 지정할 수 있습니다.

이름 없는 스레드를 이름이 지정된 스레드로 변환

이름 없는 스레드를 이름이 지정된 스레드로 변환할 수 있습니다. 예를 들어 C++Builder 5 또는 이전 버전을 사용하여 만든 스레드 클래스는 다음과 같은 방법으로 이름이 지정된 스레드로 변환할 수 있습니다.

1 *SetName* 메소드를 스레드 클래스에 추가합니다.

```
//---------------------------------------------------------------------------
----
void TMyThread::SetName()
{
        THREADNAME_INFO info;
```

```
        info.dwType = 0x1000;
        info.szName = "MyThreadName";
        info.dwThreadID = -1;
        info.dwFlags = 0;

        __try
        {
                RaiseException( 0x406D1388, 0, sizeof(info)/
sizeof(DWORD),(DWORD*)&info );
        }
        __except (EXCEPTION_CONTINUE_EXECUTION)
        {
        }
}
//---------------------------------------------------------------------
```

참고 `info.szName`을 스레드 클래스의 이름으로 설정합니다.

디버거는 예외를 발견하고 개발자가 전달한 구조에서 스레드 이름을 알아냅니다. 디버깅 동안 디버거는 Thread Status 박스의 스레드 ID 필드에 스레드 이름을 표시합니다.

2 스레드의 *Execute* 메소드 시작 시 새로운 *SetName* 메소드를 호출에 추가합니다.

```
//---------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
        SetName();
        //---- Place existing Execute method code here ----
}
//---------------------------------------------------------------------
```

비슷한 스레드에 각각의 이름 할당

같은 스레드 클래스로부터의 모든 스레드 인스턴스는 같은 이름을 갖고 있습니다. 하지만 런타임 시 다음과 같은 방법으로 각 스레드 인스턴스에 대해 다른 이름을 할당할 수 있습니다.

1 클래스 정의에 다음을 추가하여 *ThreadName* 속성을 스레드 클래스에 추가합니다.

```
__property AnsiString ThreadName = {read=FName, write=FName};
```

2 *SetName* 메소드에서 아래의 코드를 찾습니다.

```
info.szName = “MyThreadName”;
```

위 코드를 다음과 같이 변경합니다.

```
info.szName = ThreadName;
```

3 다음과 같은 방법으로 스레드 객체를 만듭니다.

1 일시 중지된 상태에서 만듭니다. 11-11페이지의 "스레드 객체 실행"을 참조하십시오.

2 `MyThread.ThreadName=”SearchForFiles”;`와 같이 이름을 할당합니다.

3 스레드를 다시 시작합니다. 11-11페이지의 "스레드 시작 및 중지"를 참조하십시오.

장

12

예외 처리

C++Builder는 C++ 예외 처리, C 기반 구조적 예외 처리, VCL 및 CLX 예외 처리 등을 지원합니다.

개발자는 ANSI 표준 C++ 예외 뿐만 아니라 기본 오류 처리 루틴이 포함된 VCL 타입 예외를 발생시킬 수 있습니다.

또한 C 기반 Win32 구조적 예외에 대한 지원도 제공되므로 코드에서 Windows 운영 체제가 thorw한 예외에 적절하게 대응할 수 있습니다.

C++ 예외 처리

특수한 처리가 필요한 예외적 조건을 나타내는 예외에는 0으로 나누거나 여유 저장소를 다 써 버리는 등의 런타임에 발생하는 오류가 포함될 수 있습니다. *예외 처리*는 오류를 처리하고 예상된 문제나 예기치 않은 문제 모두를 검색하는 표준 방법을 제공하며, 개발자가 버그를 인식, 추적 및 수정할 수 있게 해 줍니다.

오류가 발생하면 프로그램은 예외를 *throw*합니다. 일반적으로 예외는 발생한 문제에 대한 정보를 갖고 있으므로, 프로그램의 다른 부분에서 예외의 원인을 진단할 수 있습니다.

프로그램은 예외를 발생시킬 수 있는 명령문을 *try 블록*에 배치하여 예외를 준비합니다. 이러한 명령문 중 하나가 예외를 발생시키면, 컨트롤이 해당 타입의 예외를 처리하는 *예외 핸들러*에 전달됩니다. 예외 핸들러는 예외를 *catch*하라는 지시를 받고 프로그램을 종료하기 전에 수행할 작업을 지정합니다.

예외 처리 구문

예외 처리에는 세 가지 키워드, 즉 **try**, **throw** 및 **catch**를 사용해야 합니다. throw 키워드는 예외를 생성하는 데 사용됩니다. try 블록은 예외를 발생시킬 수 있는 명령문을 포함하고 그 뒤에 하나 이상의 catch 문이 옵니다. 각 catch 문은 특정 타입의 예외를 처리합니다.

참고 **try, catch** 및 **throw** 키워드는 C 프로그램에서 허용되지 않습니다.

try 블록

try 블록에는 예외를 발생시킬 수 있는 하나 이상의 명령문이 포함됩니다. 프로그램은 **throw** 문을 실행하여 예외를 발생시킵니다. **throw** 문은 대개 함수 내에서 발생합니다. 예를 들면, 다음과 같습니다.

```
void SetFieldValue(DF *dataField, int userValue)
{
    if ((userValue < 0) || userValue > 10)
     throw EIntegerRange(0, 10, userValue);
     . . .
}
```

프로그램의 다른 부분에서는 발생된 예외 객체를 catch하여 적절하게 처리할 수 있습니다. 예를 들면, 다음과 같습니다.

```
try
{
    SetFieldValue(dataField, userValue);
}
catch (EIntegerRange &rangeErr)
{
    printf("Expected value between %dand %d, but got %d\n",
        rangeErr.min, rangeErr.max, rangeErr.value);
}
```

앞의 예제에서 *SetFieldValue* 함수는 자신의 입력 매개변수가 무효한 것으로 확인되면 예외를 발생시켜 이 사실을 나타낼 수 있습니다. try/catch 블록은 *SetFieldValue*를 랩핑하여 *SetFieldValue*가 발생시킨 예외를 catch하고 printf 문을 실행합니다. 예외가 발생되지 않으면 printf 문은 실행되지 않습니다.

try에 의해 지정된 try 블록 바로 다음에는 **catch**에 의해 지정된 *핸들러*가 와야 합니다. try 블록은 프로그램이 실행될 때 컨트롤의 흐름을 지정하는 명령문입니다. try 블록에서 예외가 발생되면 프로그램 컨트롤이 해당 예외 핸들러로 전달됩니다.

*핸들러*는 예외 처리를 위해 디자인된 코드 블록입니다. C++ 랭귀지에는 try 블록 바로 다음에 최소한 하나 이상의 핸들러가 와야 합니다. 프로그램은 자신이 생성할 수 있는 각 예외에 대한 핸들러를 포함해야 합니다.

throw 문

throw 문은 다양한 타입의 객체를 발생시킬 수 있습니다. C++에서 객체는 일반적으로 값, 참조 또는 포인터에 의해 발생될 수 있습니다. 예를 들면, 다음과 같습니다.

```
// throw an object, to be caught by value or reference
throw EIntegerRange(0, 10, userValue);

// throw an object to be caught by pointer
throw new EIntegerRange(0, 10, userValue);
```

다음 두 예제는 주로 throw 문을 완전하게 하기 위해 제공되는 기능을 보여 줍니다. 보다 자세한 설명이 포함된 예외를 발생시키는 것이 좋습니다. 경우에 따라서는 정수와 같은 기본 타입을 발생시킬 수 있으며, 포인터에 의해 예외를 발생시키는 것은 좋지 않습니다.

```
// throw an integer
throw 1;

// throw a char *
throw "foo";
```

대부분의 경우, 참조에 의해 예외를 catch하는 것이 일반적이며 특히 **const** 참조가 많이 사용됩니다. 객체를 값에 의해 catch하는 것에 대해 주의를 기울여야 할 경우가 있습니다. 예를 들어, 값에 의해 catch하는 객체는 catch 매개변수에 할당하기 전에 복사해야 합니다. 이는 사용자가 복사 생성자를 제공할 경우 이 생성자가 호출되어 비효율성을 초래할 수 있기 때문입니다.

catch 문

catch 문의 형태는 다양합니다. 객체는 값, 참조 또는 포인터에 의해 catch될 수 있습니다. 또한 **const** 변경자를 catch 매개변수에 적용할 수 있습니다. 단일 try 블록에 여러 **catch** 문이 존재하면 특정 블록에서 서로 다른 여러 종류의 예외를 catch할 수 있습니다. 이 경우, 발생될 수 있는 각 예외에 대한 **catch** 문이 존재해야 합니다. 예를 들면, 다음과 같습니다.

```
try
   CommitChange(dataBase, recordMods);
catch (const EIntegerRange &rangeErr)
   printf("Got an integer range exception");
catch (const EFileError &fileErr)
   printf("Got afile I/O error");
```

CommitChange 함수가 여러 하위 시스템을 사용하고 이러한 하위 시스템이 다른 타입의 예외를 발생시킬 수 있는 경우, 여러 타입의 예외를 각각 처리할 수 있습니다. 단일 **try** 문에 여러 **catch** 문을 사용하면 각 타입의 예외에 대한 핸들러를 가질 수 있습니다.

예외 객체가 일부 기본 클래스에서 파생된 경우에는 파생된 일부 예외를 위한 특화된 핸들러를 추가할 수 있지만 기본 클래스에 대한 일반적인 핸들러를 포함할 수도 있습니다. 이렇게 하려면 예외가 발생될 때 검색하려는 순서대로 catch 문을 배치합니다. 예를 들어, 다음 코드는 먼저 *EIntegerRange*를 처리한 후 *EIntegerRange*를 파생시킨 *ERange*를 처리합니다.

```
try
   SetFieldValue(dataField, userValue);
catch (const EIntegerRange &rangeErr)
   printf("Got an integer range exception");
catch (const ERange &rangeErr)
   printf("Got a range exception");
```

마지막으로 핸들러가 try 블록을 넘어서 발생될 수 있는 모든 예외를 catch하게 하려면 특수한 형태인 **catch**(…)를 사용합니다. 이 형태는 모든 예외에 대해 핸들러를 호출하도록 예외 처리 시스템에 명령합니다. 예를 들면, 다음과 같습니다.

```
try
   SetFieldValue(dataField, userValue);
catch (...)
   printf("Got an exception of some kind");
```

예외 다시 throw

경우에 따라 예외 핸들러에서 예외를 처리한 다음 동일한 예외를 다시 발생시키거나 다른 예외를 발생시킬 수 있습니다.

현재 예외를 다시 발생시키려는 경우, 예외 핸들러는 매개변수 없이 **throw** 문을 사용할 수 있습니다. 이렇게 하면 컴파일러/RTL이 현재 예외 객체를 가져와서 다시 발생시킵니다. 예를 들면, 다음과 같습니다.

```
catch (EIntegerRange &rangeErr)
{
    // Code here to do local handling for the exception
    throw;  // rethrow the exception
}
```

예외 핸들러가 다른 예외를 발생시키려는 경우에는 일반적으로 **throw** 문을 사용합니다.

예외 규정

함수가 발생시킬 수 있는 예외를 지정할 수 있습니다. 함수를 넘어서 잘못된 타입의 예외를 발생시키는 것은 런타임 오류입니다. *예외 규정*을 위한 구문은 다음과 같습니다.

```
exception-specification:
  throw (type-id-list)    //type-id-list is optional
  type-id-list:
    type-id
    type-id-list, type-id
```

다음 예제는 예외 규정이 포함된 함수입니다.

```
void f1();                // The function can throw any exception
void f2() throw();        // Should not throw any exceptions
void f3() throw( A, B* ); //Can throw exceptions publicly derived from A,
                          // or apointer to publicly derived B
```

이러한 함수의 정의와 모든 선언은 동일한 *type-id* 집합을 포함하는 예외 규정을 가져야 합니다. 함수가 자체 규정에 나열되어 있지 않은 예외를 발생시킬 경우, 프로그램은 *unexpected*를 호출합니다.

경우에 따라서는 다음과 같은 이유 때문에 예외를 지정하지 않을 수 있습니다.

첫째로 예외 규정을 함수에 제공하면 Windows에서 런타임 성능이 떨어집니다.

둘째로 런타임에 예기치 않은 오류가 발생할 수 있습니다. 예를 들어, 시스템에서 예외 규정을 사용하고 구현에서 또 다른 하위 시스템을 사용한다고 가정합니다. 그런데 이 하위 시스템이 변경되어 새 예외 타입을 발생시킨다고 가정해 보십시오. 이 경우, 새 하위 시스템 코드를 사용하면 컴파일러로부터 오류에 대한 어떠한 정보도 얻지 못한 상태에서 런타임 오류가 발생할 수 있습니다.

셋째로 가상 함수를 사용하는 경우 프로그램 디자인을 위반할 수 있습니다. 이것은 예외 규정이 함수 타입 부분으로 간주되지 않기 때문입니다. 예를 들어, 다음 코드에서 파생된 클래스인 *BETA::vfunc*는 어떠한 예외도 발생시키지 않도록 정의되며, 이것은 원래의 함수 선언에서 벗어난 것입니다.

```
class ALPHA {
public:
  struct ALPHA_ERR {};
  virtual void vfunc(void) throw (ALPHA_ERR) {} // Exception
specification
};

class BETA : public ALPHA {
  void vfunc(void) throw() {} // Exception specification is changed
};
```

예외 해제

예외가 발생되면 런타임 라이브러리는 발생된 객체를 가져와 객체의 타입을 얻은 다음, 호출 스택에서 이 타입과 일치하는 타입의 핸들러를 찾습니다. 핸들러가 발견되면 RTL은 핸들러 지점까지 스택을 해제하고 핸들러를 실행합니다.

해제 프로세스에서 RTL은 예외가 발생된 위치와 catch되는 위치 사이의 스택 프레임에 있는 모든 로컬 객체에 대한 소멸자를 호출합니다. 소멸자가 스택 해제 도중 발생하는 예외의 원인이 되고 이 예외를 처리하지 않을 경우 *terminate*가 호출됩니다. 소멸자는 디폴트로 호출되지만 **-xd** 컴파일러 옵션을 사용하여 이 디폴트 동작을 해제할 수 있습니다.

안전 포인터

객체에 대한 포인터인 로컬 변수가 있고 예외가 발생될 경우 이러한 포인터는 자동으로 삭제되지 않습니다. 이것은 해당 함수에만 할당된 데이터에 대한 포인터와 다른 포인터를 컴파일러에서 구별할 수 있는 방법이 없기 때문입니다. 객체를 로컬 사용에 할당하는 데 사용할 수 있는 클래스는 예외가 auto_ptr인 경우에도 삭제됩니다. 함수에서 할당된 포인터에 대해 메모리가 해제되는 다음과 같은 특별한 경우가 있습니다.

```
TMyObject *pMyObject = new TMyObject;
```

이 예제에서 *TMyObject*의 생성자가 예외를 발생시키면 RTL은 예외를 해제할 때 *TMYObject*에 할당된 객체에 대한 포인터를 삭제합니다. 이것은 컴파일러가 포인터 값을 자동으로 삭제하는 유일한 경우입니다.

예외 처리에서의 생성자

클래스 생성자는 객체를 성공적으로 생성할 수 없는 경우 예외를 발생시킬 수 있습니다. 생성자가 예외를 발생시키면 해당 객체의 소멸자는 호출되지 않을 수 있습니다. 소멸자는 기본 클래스와 **try** 블록을 입력한 기본 클래스 내에서 완전하게 생성된 객체에 대해서만 호출됩니다.

catch되지 않은 예외와 예기치 않은 예외 처리

예외가 발생되고 예외 핸들러가 발견되지 않으면, 즉 예외가 catch되지 않으면 프로그램은 종료 함수를 호출합니다. 종료 함수는 *set_terminate*를 사용하여 고유하게 지정할 수 있습니다. 종료 함수를 지정하지 않으면 *terminate* 함수가 호출됩니다. 예를 들어, 다음 코드는 *my_terminate* 함수를 사용하여 핸들러가 catch하지 않은 예외를 처리합니다.

```
void SetFieldValue(DF *dataField, int userValue)
{
  if ((userValue < 0) || (userValue) > 10));
    throw EIntegerRange(0, 10, userValue);
  . . .
}

void my_terminate()
{
  printf("Exception not caught");
  abort();
}
// Set my_terminate() as the termination function
set_terminate(my_terminate);
// Call SetFieldValue. This generates an exception because the user value
is greater
// than 10. Because the call is not in a try block, my_terminate is
called.
SetFieldValue(DF, 11);
```

함수가 발생시키는 예외를 지정했는데 지정되지 않은 예외를 발생시키면 예기치 않은 함수가 호출됩니다. *set_unexpected*를 사용하여 새로운 예기치 않은 함수를 지정할 수 있습니다. 예기치 않은 함수를 지정하지 않으면 *unexpected* 함수가 호출됩니다.

Win32에서의 구조적 예외

Win32는 C++ 예외와 유사한 C 기반 구조적 예외 처리를 지원합니다. 그러나 예외 인식 C++ 코드와 혼합하여 사용할 경우에는 몇 가지 주요 차이점에 주의해야 합니다.

C++Builder 애플리케이션에서 구조적 예외 처리를 사용할 때는 다음을 염두에 두어야 합니다.

- C 구조적 예외는 C++ 프로그램에서 사용할 수 있습니다.
- C++ 예외는 C 프로그램에서 사용할 수 없습니다. 이는 C++ 예외의 경우 자체 핸들러를 **catch** 키워드로 지정해야 하지만 C 프로그램에서 **catch**가 허용되지 않기 때문입니다.
- *RaiseException* 함수 호출에 의해 생성된 예외는 **try**/**__except**(C++) 또는 **__try**/**__except**(C) 블록에서 처리됩니다. 또한 **try/__finally** 또는 **__try/__finally** 블록을 사용할 수도 있습니다(12-7페이지의 "구조적 예외 구문" 참조). *RaiseException*이 호출되면 **try**/**catch** 블록의 모든 핸들러가 무시됩니다.

- 애플리케이션이 처리하지 않는 예외는 *terminate()* 를 호출하는 대신 운영 체제로 전달되며 대개 프로세스가 종료되는 결과를 가져옵니다.
- 예외 핸들러는 요청하지 않은 경우 예외 객체의 복사본을 받지 않습니다.

C 또는 C++ 프로그램에서 다음 C 예외 helper 함수를 사용할 수 있습니다.

- *GetExceptionCode*
- *GetExceptionInformation*
- *SetUnhandledExceptionFilter*
- *UnhandledExceptionFilter*

C++Builder에서는 *UnhandledExceptionFilter* 함수의 사용이 **__try**/**__except** 또는 **try**/**__except** 블록의 except 필터로 제한되지 않습니다. 그러나 이 함수가 **__try**/**__except** 또는 **try**/**__except** 블록 외부에서 호출되면 프로그램 동작이 정의되지 않습니다.

CLX Linux에서 실행되는 CLX 애플리케이션은 C/C++ 구조적 예외를 지원하지 않습니다.

구조적 예외 구문

C 프로그램에서 구조적 예외를 구현하는 데 사용되는 ANSI 규격 키워드는 **__except**, **__finally** 및 **__try**입니다.

참고 **__try** 키워드는 C 프로그램에만 나타날 수 있습니다. 이식 가능한 코드를 작성하려면 C++ 프로그램에서 구조적 예외 처리를 사용하지 마십시오.

try-except 예외 처리 구문

try-except 예외 처리의 경우 구문은 다음과 같습니다.

```
try-block:
    __try compound-statement (in a C module)
    try compound-statement (in a C++ module)
handler:
    __except (expression) compound-statement
```

try-finally 종료 구문

try-finally 종료의 경우 구문은 다음과 같습니다.

```
try-block:
    __try compound-statement (in a C module)
    try compound-statement (in a C++ module)
termination:
    __finally compound-statement
```

구조적 예외 처리

다음과 같이 C++ 예외 처리의 확장을 사용하여 구조적 예외를 처리할 수 있습니다.

```
try {
        foo();
    }
    __except(__expr__) {
        // handler here
    }
```

__expr__은 다음 세 개의 값 중 하나로 분석되는 표현식입니다.

값	설명
EXCEPTION_CONTINUE_SEARCH (0)	핸들러가 입력되지 않고 OS가 예외 핸들러 검색을 계속합니다.
EXCEPTION_CONTINUE_EXECUTION (-1)	예외 지점에서 실행을 계속합니다.
EXCEPTION_EXECUTE_HANDLER (1)	예외 핸들러를 입력합니다. 소멸자 제거가 활성화된 상태(즉, -xd가 디폴트로 설정된 경우)로 코드가 컴파일되지 않았다면, 예외 지점과 예외 핸들러 간에 작성된 로컬 객체의 모든 소멸자가 스택 해제 시 호출됩니다. 스택 해제는 핸들러를 입력하기 전에 완료됩니다.

Win32는 활성 예외에 대한 정보를 쿼리하는 데 사용할 수 있는 두 가지 함수, 즉 *GetExceptionCode()*와 *GetExceptionInformation()*을 제공합니다. 특정 함수를 위에 나온 "필터" 표현식의 일부로 호출하려면 다음과 같이 이러한 함수를 *__except()*의 컨텍스트 내에서 직접 호출해야 합니다.

```
#include <Windows.h>
#include <excpt.h>

int filter_func(EXCEPTION_POINTERS *);
...
EXCEPTION_POINTERS *xp = 0;
    try {
        foo();
    }
    __except(filter_func(xp = GetExceptionInformation())) {
        //...
    }
```

또는 함수 호출에서 중첩된 할당에 대한 쉼표 연산자를 사용하려면 다음 예제를 참조하십시오.

```
__except((xp = GetExceptionInformation()), filter_func(xp))
```

예외 필터

필터 표현식은 필터 함수를 호출할 수 있지만 필터 함수는 *GetExceptionInformation*을 호출할 수 없습니다. *GetExceptionInformation*의 반환값을 매개변수로 필터 함수에 전달할 수 있습니다.

EXCEPTION_POINTERS 정보를 예외 핸들러에 전달하려면 필터 표현식이나 필터 함수는 포인터나 데이터를 *GetExceptionInformation*에서 핸들러가 나중에 액세스할 수 있는 위치로 복사해야 합니다.

중첩된 **try-except** 문의 경우, 각 명령문의 필터 표현식은 EXCEPTION_EXECUTE_HANDLER 또는 EXCEPTION_CONTINUE_EXECUTION을 찾을 때까지 분석됩니다. 필터 표현식은 *GetExceptionInformation*을 호출하여 예외 정보를 가져올 수 있습니다.

GetExceptionInformation 또는 *GetExceptionCode*가 **__except**에 제공된 표현식에서 직접 호출되면, 복잡한 C++ 표현식을 만들려고 노력하는 대신 함수를 사용하여 예외와 관련하여 수행할 작업을 결정할 수 있습니다. 예외를 처리하는 데 필요한 거의 모든 정보는 *GetExceptionInformation()*의 결과에서 추출할 수 있습니다. *GetExceptionInformation()*은 다음과 같이 EXCEPTION_POINTERS 구조체에 대한 포인터를 반환합니다.

```
struct EXCEPTION_POINTERS {
        EXCEPTION_RECORD *ExceptionRecord;
        CONTEXT *Context;
    };
```

다음과 같이 EXCEPTION_RECORD에는 컴퓨터에 독립적인 상태가 포함되어 있습니다.

```
struct EXCEPTION_RECORD {
        DWORD ExceptionCode;
        DWORD ExceptionFlags;
        struct EXCEPTION_RECORD *ExceptionRecord;
        void *ExceptionAddress;
        DWORD NumberParameters;
        DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
    };
```

일반적으로 필터 함수는 ExceptionRecord에서 정보를 찾아 응답 방법을 결정합니다. 경우에 따라 더 특정한 정보가 필요할 수 있습니다. 특히 수행할 작업이 EXCEPTION_CONTINUE_EXECUTION인 경우, 수행되는 작업이 없다면 예외의 원인이 된 코드가 다시 실행됩니다. 이러한 경우, EXCEPTION_POINTERS 구조체의 다른 필드는 예외 발생 시점에 프로세서 상태를 제공합니다. 이 구조체가 수정되거나 필터가 EXCEPTION_CONTINUE_EXCEPTION을 반환하면 실행을 계속하기 전에 스레드 상태를 설정하는 데 이 구조체가 사용됩니다. 예를 들면, 다음과 같습니다.

```
static int xfilter(EXCEPTION_POINTERS *xp)
    {
        int rc;

        EXCEPTION_RECORD *xr = xp->ExceptionRecord;
        CONTEXT *xc = xp->Context;

        switch (xr->ExceptionCode) {
            case EXCEPTION_BREAKPOINT:
                // whoops, someone left an embedded breakpoint.
                // just step over it (1 byte on x86)
                ++xc->Eip;
                rc = EXCEPTION_CONTINUE_EXECUTION;
                break;
```

```
            case EXCEPTION_ACCESS_VIOLATION:
                rc = EXCEPTION_EXECUTE_HANDLER;
                break;

            default:
                // give up
                rc = EXCEPTION_CONTINUE_SEARCH;
                break;
        };

        return rc;
    }
    ...

    EXCEPTION_POINTERS *xp;

    try {
        func();
    }
    __except(xfilter(xp = GetExceptionInformation())) {
        abort();
    }
```

C++와 구조적 예외 혼합

C++ 프로그램에서 구조적 예외를 사용할 때는 몇 가지 문제를 알아야 합니다. 우선 C++Builder가 Win32 구조적 예외를 사용하여 C++ 예외를 구현하지만 C++ 예외는 **__except** 블록에서 분명하게 나타납니다.

try 블록 다음에는 정확하게 **except** 블록 한 개가 오거나 최소 하나 이상의 catch 블록이 올 수 있습니다. 이 두 블록을 혼합하려고 하면 컴파일러 오류가 발생합니다. 예외의 두 타입을 모두 처리해야 하는 코드는 단순히 **try** 블록 두 개 내에서 중첩되어야 합니다.

```
try {
        EXCEPTION_POINTERS *xp;

        try {
            func();
        }
        __except(xfilter(xp = GetExceptionInformation())) {
            //...
        }
    }
    catch (...) {
        //...
    }
```

함수의 *throw()* 지정은 Win32 예외와 관련하여 프로그램 동작에 영향을 주지 않습니다. 또한 *terminate()*를 호출하는 C++ 프로그램과 달리 처리되지 않은 예외는 최종적으로 운영 체제에 의해 처리됩니다(디버거나 예외를 그 전에 처리하지 않는 경우).

-xd 컴파일러 옵션(디폴트로 설정됨)을 사용하여 컴파일된 모든 모듈은 **auto** 저장소를 가진 모든 객체에 대해 소멸자를 호출합니다. 스택 해제는 예외가 thorw된 지점에서 예외가 catch되는 지점으로 발생합니다.

C++ 프로그램 예제에서의 C 기반 예외

```
/* Program results:
Another exception:
Caught a C-based exception.
Caught C++ exception[Hardware error: Divide by 0]
C++ allows __finally too!
*/
#include <stdio.h>
#include <string.h>
#include <windows.h>

class Exception
{
  public:
  Exception(char* s = "Unknown"){ what = strdup(s);      }
  Exception(const Exception& e ){what = strdup(e.what); }
 ~Exception()                    { delete[] what;         }
  char* msg() const              { return what;           }
private:
  char* what;
};
int main()
{
  float e, f, g;
  try
  {
    try
    {
      f = 1.0;
      g = 0.0;
      try
      {
        puts("Another exception:");
        e = f / g;
      }
      __except(EXCEPTION_EXECUTE_HANDLER)
      {
        puts("Caught a C-based exception.");
        throw(Exception("Hardware error: Divide by 0"));
      }
    }
    catch(const Exception&e)
    {
      printf("Caught C++ Exception: %s :\n", e.msg());
    }
  }
```

```
  __finally
  {
    puts("C++ allows __finally too!");
  }
return e;
}
```

예외 정의

일반적으로 동일한 프로그램 내에서 처리되는 Win32 예외를 발생시키는 것은 비합리적입니다. 이는 C++ 예외가 더 뛰어난 작업 수행 능력과 이식성을 제공하고 더 간단한 구문을 사용하기 때문입니다. 그러나 Win32 예외는 동일한 C++ 컴파일러로 컴파일되지 않는 컴포넌트가 처리할 수 있다는 이점이 있습니다.

첫 번째 단계는 예외를 정의하는 것입니다. 예외는 0비트에서 시작하는 다음 형식을 갖는 32비트 정수입니다.

비트	의미
31-30	11 = 오류(정상)
	00 = 성공
	01 = 정보
	10 = 경고
29	1 = 사용자 정의
28	예약
27-0	사용자 정의

예외 코드를 정의하는 것 외에도 예외 레코드의 필터/핸들러에서 액세스할 수 있는 예외에 추가 정보를 포함할 것인지 여부를 결정해야 합니다. 예외 코드에서 추가 매개변수를 인코딩하는 것에 대한 기본 방법은 존재하지 않습니다. 자세한 내용은 C++Builder 온라인 도움말에서 제공되는 Win32 설명서를 참조하십시오.

예외 발생

Win32 예외는 다음과 같이 선언되는 *RaiseException()*을 호출하여 발생시킵니다.

```
void RaiseException(DWORD ec, DWORD ef, DWORD na, const DWORD *a);
```

여기서 각 항목의 내용은 다음과 같습니다.

ec 예외 코드

ef 예외 플래그(0 또는 EXCEPTION_NONCONTINUABLE)
예외가 계속할 수 없는 것으로 표시된 상태에서 필터가 예외를 계속하려고 시도하면 EXCEPTION_NONCONTINUABLE_EXCEPTION이 발생합니다.

na 인수 배열에 있는 요소 수

a 인수 배열에 있는 첫 번째 요소의 포인터(이러한 인수의 의미는 특정 예외에 따라 달라짐)

종료 블록

구조적 예외 처리 모델은 보호된 블록이 정상적으로 종료되는지, 아니면 예외를 통해 종료되는지 여부에 상관 없이 실행되는"종료 블록"을 지원합니다. C++Builder 컴파일러는 다음 구문을 사용하여 C에서 이러한 지원을 구현합니다.

```
__try {
        func();
    }
    __finally {
        // this happens whether func() raises an exception or not
    }
```

종료 블록은 다음과 같이 **__finally** 블록에서 정리를 처리할 수 있는 C++ 확장에 의해 지원됩니다.

```
try {
        func();
    }
    __finally {
        // this happens whether func() raises an exception or not
    }
```

다음 예제는 종료 블록을 보여 줍니다.

```
/* Program results:
An exception:
Caught an exception.
The __finally is executed too!
No exception:
No exception happened, but __finally still executes!
*/
#include <stdio.h>
#include <windows.h>

int main()
{
  float e, f, g;

  try
  {
    f = 1.0;
    g = 0.0;
    try
    {
      puts("An exception:");
      e = f / g;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
      puts("Caught an exception.");
    }
  }
```

```
  __finally
  {
    puts("The __finally is executed too!");
  }
  try
  {
    f = 1.0;
    g = 2.0;
    try
    {
      puts("No exception:");
      e = f / g;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
      puts("Caught an exception.");
    }
  }
  __finally
  {
    puts("No exception happened, but __finally still executes!");
  }
return
}
```

또한 C++ 코드는 범위를 벗어났을 때 호출되는 소멸자가 포함된 로컬 객체를 만들어 "종료 블록"을 처리할 수 있습니다. C++Builder 구조적 예외가 소멸자 정리를 지원하므로 이 로컬 객체는 발생한 예외 타입에 상관 없이 작동합니다.

참고 발생한 예외를 프로그램에서 처리하지 않는 경우 수행되는 작업과 관련하여 한 가지 특수한 상황이 있습니다. 즉, 처리되지 않은 C++ 예외의 경우에는 C++Builder 컴파일러가 랭귀지 정의에 필요하지 않은 로컬 객체에 대한 소멸자를 호출하지만, 처리되지 않은 Win32 예외의 경우에는 소멸자 정리가 발생하지 않습니다.

C++Builder 예외 처리 옵션

다음은 C++Builder 컴파일러의 예외 처리 옵션입니다.

표 12.1 예외 처리 컴파일러 옵션

명령줄 스위치	설명
-x	C++ 예외 처리를 활성화합니다. 기본값은 on입니다.
-xd	소멸자 정리를 활성화합니다. 예외가 발생될 때 catch 및 throw 문의 범위 간에 자동으로 선언된 모든 객체에 대한 소멸자를 호출합니다. 고급 옵션이며 기본값은 on입니다.
-xp	예외 위치 정보를 활성화합니다. 예외 위치의 소스 코드에 있는 줄 번호를 제공하여 예외의 런타임 식별을 사용할 수 있게 만듭니다. 이렇게 하면 프로그램은 __ThrowFileName 및 __ThrowLineNumber 전역을 사용하여 C++ 예외가 발생한 파일 및 줄 번호를 쿼리할 수 있습니다. 고급 옵션입니다.

VCL/CLX 예외 처리

애플리케이션에서 VCL 및 CLX 컴포넌트를 사용하는 경우, VCL/CLX 예외 처리 메커니즘을 이해해야 합니다. 이는 많은 클래스에 예외가 생성되고, 예기치 않은 현상이 일어나면 자동으로 예외가 발생되기 때문입니다. 개발자가 예외를 처리하지 않으면 VCL 및 CLX는 예외를 디폴트 방식으로 처리합니다. 일반적으로 발생한 오류의 타입을 설명하는 메시지가 표시됩니다.

발생된 예외의 타입을 나타내는 메시지를 표시하는 예외가 발생하면 온라인 도움말에서 예외 클래스를 찾아볼 수 있습니다. 대개 제공된 정보는 오류가 발생한 위치와 그 원인을 확인하는 데 도움을 줍니다.

이 외에도 13장, "VCL 및 CLX에 대한 C++ 랭귀지 지원"에서는 예외의 원인이 되는 미묘한 랭귀지 차이점에 대해 설명합니다. 13-13페이지의 "생성자에서 발생되는 예외" 섹션은 객체 생성 도중 예외가 발생될 경우 어떤 작업이 수행되는지 예제를 통해 보여 줍니다.

C++ 및 VCL/CLX 예외 처리 간의 차이점

다음은 C++ 및 VCL/CLX 예외 처리 간의 몇 가지 두드러진 차이점을 나열한 것입니다.

생성자에서 발생된 예외:

- C++ 소멸자는 완전하게 생성된 기본 클래스 및 멤버에 대해 호출됩니다.
- VCL 및 CLX 기본 클래스 소멸자는 객체 또는 기본 클래스가 완전히 생성되지 않는 경우에도 호출됩니다.

예외 catch 및 throw:

- C++ 예외는 참조, 포인터 또는 값에 의해 catch할 수 있습니다. *TObject*에서 파생된 VCL 및 CLX 예외는 참조 또는 포인터에 의해서만 catch할 수 있습니다. *TObject* 예외를 값에 의해 catch하려고 하면 컴파일 타임 오류가 발생합니다. *EAccessViolation*과 같은 하드웨어 또는 운영 체제 예외는 참조에 의해 catch해야 합니다.
- VCL 및 CLX 예외는 참조에 의해 catch됩니다.
- **throw**를 사용하여 VCL 또는 CLX 코드 내에서 catch된 예외를 다시 발생시킬 수 없습니다.

운영 체제 예외 처리

C++Builder를 사용하면 운영 체제에서 발생시킨 예외를 처리할 수 있습니다. 운영 체제 예외에는 액세스 위반, 정수 계산 오류, 부동 소수점 계산 오류, 스택 오버플로, *Ctrl+C* 중단 등이 포함됩니다. 이러한 예외는 C++ RTL에서 처리되고 애플리케이션으로 디스패치되기 전에 VCL 및 CLX 예외 클래스 객체로 변환됩니다. 그런 다음 아래와 같은 C++ 코드를 작성할 수 있습니다.

```
try
{
  char * p = 0;
  *p = 0;
}
```

```
// You should always catch by reference.
catch (const EAccessViolation &e)
{
  printf("You can't do that!\n");
}
```

C++Builder가 사용하는 클래스는 Delphi가 사용하는 클래스와 동일하고 C++Builder VCL 및 CLX 애플리케이션에서만 사용할 수 있습니다. *TObject*에서 파생되는 이러한 클래스에는 VCL 및 CLX 지원이 필요합니다.

다음은 C++Builder 예외 처리의 몇 가지 특성입니다.

- 예외 객체를 해제할 필요가 **없습니다**.
- 운영 체제 예외는 참조에 의해 catch되어야 합니다.
- catch 프레임을 벗어났거나 VCL 및 CLX catch 프레임에 끼어들어 운영 체제 예외를 catch하도록 만든 경우 이 예외를 다시 발생시킬 수 없습니다.
- catch 프레임을 벗어났거나 운영 체제 catch 프레임에 끼어들어 운영 체제 예외를 catch하도록 만든 경우 이 예외를 다시 발생시킬 수 없습니다.

마지막 특성 두 개는 좀더 개략적으로 살펴보면, 개발자가 catch하는 스택 프레임에 있는 경우를 제외하고 운영 체제 예외가 C++ 예외로 catch되면 운영 체제 예외나 VCL 또는 CLX 예외인 것처럼 다시 발생시킬 수 없다는 의미입니다.

VCL 및 CLX 예외 처리

C++Builder는 VCL 및 CLX에서 발생된 소프트웨어 예외, 즉 발생되는 예외 클래스가 *TObject*에서 파생되는 C++에서 발생된 예외를 처리하기 위해 의미론을 확장합니다. 이 경우, VCL 스타일 클래스를 힙(heap) 상에서만 할당할 수 있다는 사실에서 몇 가지 규칙이 파생됩니다.

- VCL 스타일 예외 클래스는 포인터(소프트웨어 예외인 경우) 또는 참조(참조가 선호됨)에 의해서만 catch할 수 있습니다.
- VCL 스타일 예외는 "by value" 구문을 통해 발생되어야 합니다.

VCL 및 CLX 예외 클래스

C++Builder에는 0으로 나누기 오류, 파일 I/O 오류, 잘못된 타입 변환 및 다른 여러 예외 상태를 자동으로 처리하기 위한 대규모의 기본 예외 클래스 집합이 포함되어 있습니다. 모든 VCL 및 CLX 예외 클래스는 *Exception*이라는 단일 루트 객체의 자손입니다. *Exception*은 모든 VCL 타입 예외에 대한 기본 속성과 메소드를 캡슐화하고 예외를 처리하기 위한 일관된 인터페이스를 애플리케이션에 제공합니다.

Exception 타입의 매개변수를 가져오는 **catch** 블록에 예외를 전달할 수 있습니다. 다음 구문을 사용하여 VCL 및 CLX 예외를 catch합니다.

```
catch (exception_class &exception_variable)
```

catch할 예외 클래스를 지정하고 예외를 참조하는 데 사용할 변수를 제공합니다.

다음 예제는 VCL 또는 CLX 예외를 발생시키는 방법을 보여 줍니다.

```
void __fastcall TForm1::ThrowException(TObject *Sender)
{
  try
  {
    throw Exception("An error has occurred");
  }
  catch(const Exception &E)
  {
    ShowMessage(AnsiString(E.ClassName())+ E.Message);
  }
}
```

위 예제의 **throw** 문은 *Exception* 클래스의 인스턴스를 만들고 이 클래스의 생성자를 호출합니다. *Exception*의 자손인 모든 예외는 표시할 수 있고, 생성자를 통해 전달할 수 있고, *Message* 속성을 통해 검색할 수 있는 메시지를 가집니다.

표 12.2는 선택된 VCL/CLX 예외 클래스를 나열한 것입니다.

표 12.2 선택된 예외 클래스

예외 클래스	설명
EAbort	오류 메시지 다이얼로그 박스를 표시하지 않고 일련의 이벤트를 중단합니다.
EAccessViolation	잘못된 메모리 액세스 오류를 확인합니다.
EBitsError	부울 배열을 액세스하려는 잘못된 시도를 방지합니다.
EComponentError	컴포넌트를 등록하거나 그 이름을 바꾸려는 잘못된 시도를 알립니다.
EConvertError	문자열 또는 객체 변환 오류를 나타냅니다.
EDatabaseError	데이터베이스 액세스 오류를 지정합니다.
EDBEditError	지정된 마스크와 호환되지 않는 데이터를 catch합니다.
EDivByZero	정수를 0으로 나누는 오류를 catch합니다.
EExternalException	인식할 수 없는 예외 코드를 나타냅니다.
EInOutError	파일 I/O 오류를 나타냅니다.
EIntOverflow	할당된 레지스터에 비해 너무 큰 결과를 가져오는 정수 계산을 지정합니다.
EInvalidCast	유효하지 않은 타입 변환을 확인합니다.
EInvalidGraphic	인식할 수 없는 그래픽 파일 형식을 사용하려는 시도를 나타냅니다.
EInvalidOperation	컴포넌트에서 잘못된 작업을 시도하는 경우에 발생합니다.
EInvalidPointer	잘못된 포인터 작업에서 발생합니다.
EMenuError	메뉴 항목 문제를 포함합니다.
EOleCtrlError	ActiveX 컨트롤에 대한 연결 문제를 감지합니다.
EOleError	OLE Automation 오류를 지정합니다.
EPrinterError	인쇄 오류를 알립니다.
EPropertyError	속성 값을 설정하려는 시도가 실패할 경우 발생합니다.
ERangeError	정수 값이 할당되는 선언된 타입에 비해 너무 큰 정수 값을 나타냅니다.
ERegistryException	레지스트리 오류를 나타냅니다.
EZeroDivide	부동 소수점을 0으로 나누는 오류를 catch합니다.

위의 선택된 리스트에서 볼 수 있는 것처럼 기본 VCL 및 CLX 예외 클래스는 많은 예외 처리를 수행하여 코드를 단순화할 수 있습니다. 경우에 따라서는 고유한 상황을 처리하는 자신만의 예외 클래스를 만들어야 할 수 있습니다. 예외 클래스를 *Exception* 타입의 자손으로 만들고 필요한 만큼 생성자를 만들거나 Sysutils.hpp의 기존 클래스에서 생성자를 복사하여 새 예외 클래스를 선언할 수 있습니다.

이식성 고려 사항

C++Builder는 여러 개의 런타임 라이브러리(RTL)를 제공합니다. 이러한 라이브러리는 대개 C++Builder 애플리케이션에 속하지만, cw32mt.lib는 VCL 또는 CLX에 대한 참조를 만들지 않는 일반적인 멀티 스레드 RTL입니다. 이 RTL은 프로젝트의 일부이면서 VCL 또는 CLX에 의존하지 않는 레거시 애플리케이션을 지원하기 위해 제공됩니다 . 이 RTL 은 운영 체제 예외의 catch를 지원하지 않습니다. 이는 이러한 예외 객체가 *TObject*에서 파생되고 VCL 및 CLX의 일부가 애플리케이션에 연결되어야 하기 때문입니다.

멀티 스레드 런타임 라이브러리인 cp32mt.lib 라이브러리를 사용할 수 있습니다. 이 RTL은 VCL 및 CLX를 통한 메모리 관리 및 예외 처리를 제공합니다.

RTL DLL을 사용하기 위해 두 개의 임포트 라이브러리인 cw32mti.lib 및 cp32mti.lib를 사용할 수 있습니다. VCL 및 CLX 예외 지원에는 cp32mti.lib를 사용합니다.

장

13

VCL 및 CLX에 대한 C++ 랭귀지 지원

C++Builder는 오브젝트 파스칼에서 작성된 비주얼 컴포넌트 라이브러리(VCL) 및 크로스 플랫폼 컴포넌트 라이브러리(CLX)의 RAD(Rapid Application Development) 기능을 사용합니다. 이 장에서는 오브젝트 파스칼 랭귀지 기능, 구조 및 개념이 C++Builder에서 구현되어 VCL 및 CLX를 지원하는 방법에 대해 설명합니다. 이 장의 내용은 애플리케이션에서 VCL 및 CLX를 사용하는 프로그래머와 VCL 및 CLX 클래스의 자손인 새 클래스를 만드는 개발자를 위한 것입니다.

이 장의 전반부에서는 C++ 및 오브젝트 파스칼 모델을 비교하고 C++Builder가 이러한 두 가지 방식을 결합하는 방법에 대해 설명합니다. 이어서 후반부에서는 C++Builder에서 오브젝트 파스칼 랭귀지 구조가 C++ 구조로 변환된 방법에 대해 설명합니다. 여기에는 VCL 및 CLX를 지원하기 위하여 추가된 키워드 확장에 대한 자세한 내용도 포함됩니다. 이러한 확장 중에서 클로저(closure) 및 속성과 같은 일부 확장은 VCL 및 CLX 기반 코드 지원에 독립적인 유용한 기능입니다.

참고 *TObject*에서 파생된 C++ 클래스에 대한 참조는 *TObject*가 궁극적인 조상이지만 직계 조상일 필요는 없는 클래스를 참조합니다. 컴파일러 진단과의 일관성을 유지하기 위해 이러한 클래스를 "VCL 스타일 클래스"라고도 합니다.

C++ 및 오브젝트 파스칼 객체 모델

C++와 오브젝트 파스칼 객체 모델 간에는 명확한 차이점과 미세한 차이점이 모두 있습니다. 가장 명확한 차이점 중 하나는 C++에서 다중 상속을 허용하는 것과 달리 오브젝트 파스칼은 단일 상속 모델로 제한된다는 점입니다. 또한 C++와 오브젝트 파스칼에는 객체 작성, 초기화, 참조, 복사 및 삭제하는 방법에서 약간의 차이가 있습니다. 이 단원에서는 이러한 차이점이 무엇이고, C++Builder VCL 스타일 클래스에 어떤 영향을 주는지에 대해 설명합니다.

상속 및 인터페이스

C++와 달리 오브젝트 파스칼 랭귀지는 복수 상속을 지원하지 않습니다. VCL 또는 CLX 조상을 갖는 작성된 모든 클래스는 이 제한을 상속합니다. 즉, VCL 또는 CLX 클래스가 직계 조상이 아닌 경우에도 VCL 스타일 C++ 클래스에 여러 기본 클래스를 사용할 수 없습니다.

복수 상속 대신 인터페이스 사용

C++에서 복수 상속을 사용하는 많은 상황을 오브젝트 파스칼은 대신 인터페이스를 사용하여 처리합니다. 오브젝트 파스칼의 인터페이스 개념에 직접 매핑되는 C++ 구조는 존재하지 않습니다. 오브젝트 파스칼 인터페이스는 구현이 없는 클래스처럼 동작합니다. 즉, 모든 메소드가 순수 가상 메소드이고 데이터 멤버가 존재하지 않는 클래스로 볼 수 있습니다. 오브젝트 파스칼 클래스는 단일 부모 클래스만 가질 수 있지만 인터페이스는 그 수에 상관 없이 지원할 수 있습니다. 오브젝트 파스칼 코드는 클래스 인스턴스를 모든 조상 클래스 타입의 변수에 할당할 수 있는 것처럼 모든 인터페이스 타입의 변수에 클래스 인스턴스를 할당할 수 있습니다. 결과적으로 클래스의 다형적인 동작은 공통된 조상이 없더라도 동일한 인터페이스를 공유할 수 있습니다.

C++Builder에서 컴파일러는 순수 가상 메소드만 있고 데이터 멤버가 없는 클래스를 오브젝트 파스칼 인터페이스에 해당하는 것으로 인식합니다. 따라서 VCL 스타일 클래스를 만들 때 복수 상속을 사용할 수 있지만, 이것은 VCL, CLX 또는 VCL 스타일 클래스를 제외한 모든 기본 클래스에 순수 가상 메소드만 있고 데이터 멤버가 없는 경우에만 가능합니다.

참고 인터페이스 클래스에는 VCL 스타일 클래스는 필요하지 않으며, 순수 가상 메소드만 있고 데이터 멤버가 없어야 한다는 요구 사항만 충족하면 됩니다.

인터페이스 클래스 선언

다른 C++ 클래스와 마찬가지로 인터페이스를 나타내는 클래스를 선언할 수 있습니다. 그러나 클래스를 인터페이스로 사용하고자 하는 의도를 특정 규칙을 사용하여 더 분명하게 나타낼 수 있습니다. 이러한 규칙은 다음과 같습니다.

- 클래스 키워드를 사용하는 대신 __interface를 사용하여 인터페이스를 선언합니다. __interface는 클래스 키워드에 매핑되는 매크로입니다. __interface는 꼭 필요한 것은 아니지만, 클래스를 인터페이스로 사용하고자 하는 의도를 분명하게 나타냅니다.
- 일반적으로 인터페이스는 문자 'I'로 시작되는 이름을 가집니다. *IComponentEditor* 또는 *IDesigner*를 예로 들 수 있습니다. 이 규칙을 따르면 클래스가 인터페이스로 작동하는 시점을 확인하기 위해 클래스 선언을 다시 살펴볼 필요가 없습니다.
- 일반적으로 인터페이스는 연결된 GUID를 가집니다. 이것은 절대 요구 사항은 아니지만, 인터페이스를 지원하는 대부분의 코드는 GUID의 존재를 예상합니다. **uuid** 인수와 함께 **__declspec** 변경자를 사용하면 인터페이스와 GUID를 연결할 수 있습니다. 인터페이스의 경우, INTERFACE_UUID 매크로가 동일한 작업을 수행합니다.

다음 인터페이스 선언은 이러한 규칙을 보여 줍니다.

```
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}")
IHelloWorld :
    public IInterface
{
public:
```

```
  virtual void __stdcall SayHelloWorld(void) = 0 ;
};
```

일반적으로 인터페이스 클래스가 선언될 때, C++Builder 코드는 인터페이스 작업을 더 편리하게 만드는 해당 *DelphiInterface* 클래스도 다음과 같이 선언합니다.

```
typedef System::DelphiInterface< IHelloWorld >  _di_IHelloWorld;
```

DelphiInterface 클래스에 대한 자세한 내용은 13-20페이지의 "Delphi 인터페이스"를 참조하십시오.

IUnknown 및 IInterface

모든 오브젝트 파스칼 인터페이스는 공통된 단일 조상인 *IInterface*의 자손입니다. VCL 스타일 클래스가 C++ 인터페이스 클래스를 추가 기본 클래스로 사용할 수 있다는 점에서 C++ 인터페이스 클래스는 *IInterface*를 기본 클래스로 사용할 필요가 없지만, 인터페이스를 다루는 VCL 및 CLX 코드는 *IInterface* 메소드가 존재한다고 가정합니다.

COM 프로그래밍에서는 모든 인터페이스가 *IUnknown*의 자손입니다. VCL에서의 COM 지원은 *IInterface*로 직접 매핑되는 *IUnknown*의 정의에 기초합니다. 즉, 오브젝트 파스칼에서는 *IUnknown*과 *IInterface*가 동일합니다.

그러나 *IUnknown*의 오브젝트 파스칼 정의는 C++Builder에 사용되는 *IUnknown* 정의에 해당되지 않습니다. unknwn.h 파일에서는 다음 세 가지 메소드가 포함되도록 *IUnknown*이 정의됩니다.

```
virtual HRESULT STDCALL QueryInterface( const GUID &guid, void ** ppv ) =
0;
virtual ULONG STDCALL AddRef() = 0;
virtual ULONG STDCALL Release() = 0;
```

이것은 Microsoft에서 COM 규정의 일부로 정의한 *IUnknown*의 정의와 일치합니다.

참고 *IUnknown*과 그 사용에 대한 자세한 내용은 38장, "COM 기술 개요" 또는 Microsoft 설명서를 참조하십시오.

오브젝트 파스칼의 경우와 달리 *IInterface*의 C++Builder 정의는 *IUnknown* 정의와 동등하지 않습니다. 대신 *IInterface*는 추가 메소드인 *Supports*를 포함하는 *IUnknown*의 자손입니다.

```
template <typename T>
HRESULT __stdcall Supports(DelphiInterface<T>& smartIntf)
{
  return QueryInterface(__uuidof(T),
         reinterpret_cast<void**>(static_cast<T**>(&smartIntf)));
}
```

*Supports*는 *IInterface*를 구현하는 객체에서 다른 지원되는 인터페이스의 *DelphiInterface*를 얻을 수 있게 해 줍니다. 예를 들어, *IMyFirstInterface* 인터페이스의 *DelphiInterface*가 있고 구현 객체가 *IMySecondInterface*(*DelphiInterface* 타입은 *_di_IMySecondInterface*)도 구현할 경우 두 번째 인터페이스의 *DelphiInterface*를 다음과 같이 얻을 수 있습니다.

```
_di_IMySecondInterface MySecondIntf;
MyFirstIntf->Supports(MySecondIntf);
```

VCL 및 CLX는 오브젝트 파스칼로의 *IUnknown* 매핑을 사용합니다. 이 매핑은 *IUnknown* 메소드에 사용된 타입을 오브젝트 파스칼 타입으로 변환하고 *AddRef* 및 *Release* 메소드의 이름을 *_AddRef* 및 *_Release*로 바꿔서 이러한 메소드를 직접 호출할 수 없다는 것을 나타냅니다. 오브젝트 파스칼에서 컴파일러는 *IUnknown* 메소드에 필요한 호출을 자동으로 생성합니다. VCL 및 CLX 객체가 지원하는 *IUnknown* 또는 *IInterface* 메소드가 C++로 다시 매핑되면 다음 메소드 시그너처가 결과로 얻어집니다.

```
virtual HRESULT __stdcall QueryInterface(const GUID &IID, void *Obj);
int __stdcall _AddRef(void);
int __stdcall _Release(void);
```

이것은 오브젝트 파스칼에서 *IUnknown* 또는 *IInterface*를 지원하는 VCL 및 CLX 객체가 C++Builder에 표시되는 *IUnknown* 및 *IInterface*의 버전을 지원하지 않는다는 것을 의미합니다.

IUnknown을 지원하는 클래스 생성

VCL 및 CLX 클래스는 대부분 인터페이스 지원을 포함합니다. 실제로 모든 VCL 스타일 클래스에 대한 기본 클래스인 *TObject*는 객체 인스턴스가 지원하는 모든 인터페이스의 *DelphiInterface* 클래스를 얻게 해 주는 *GetInterface* 메소드를 가집니다. *TComponent* 및 *TInterfacedObject*는 둘 다 *IInterface*(*IUnknown*) 메소드를 구현합니다. 따라서 작성된 *TComponent* 또는 *TInterfacedObject*의 모든 자손은 모든 오브젝트 파스칼 인터페이스의 공통 메소드에 대한 지원을 자동으로 상속합니다. 오브젝트 파스칼에서 이러한 클래스 중 하나의 자손을 만들 때는 새 인터페이스가 사용한 이러한 메소드만 구현하고 상속된 *IInterface* 메소드에 대한 디폴트 구현에 의존하여 새 인터페이스를 지원할 수 있습니다.

불행하게도 *IUnknown* 및 *IInterface* 메소드의 시그너처가 C++Builder와 VCL 및 CLX 클래스에 사용되는 버전 간에 다르기 때문에 작성된 VCL 스타일 클래스는 *TComponent* 또는 *TInterfacedObject*에서 직간접적으로 파생된 경우에도 *IInterface* 또는 *IUnknown* 지원을 자동으로 포함하지 않습니다. 즉, *IUnknown* 또는 *IInterface*의 자손이고 C++에서 정의되는 모든 인터페이스를 지원하려면 이전과 같이 *IUnknown* 메소드의 구현을 추가해야 합니다.

TComponent 또는 *TInterfacedObject*의 자손인 클래스에서 *IUnknown* 메소드를 구현하는 가장 쉬운 방법은 함수 시그너처에 차이가 있더라도 기본 *IUnknown* 지원을 활용하는 것입니다. 단순히 *IUnknown* 메소드의 C++ 버전에 대한 구현을 추가하여 상속된 오브젝트 파스칼 기반 버전으로 위임하면 됩니다. 예를 들면, 다음과 같습니다.

```
virtual HRESULT __stdcall QueryInterface(const GUID& IID, void **Obj)
{
   return TInterfacedObject::QueryInterface(IID, (void *)Obj);
}
virtual ULONG __stdcall AddRef()
{
   return TInterfacedObject::_AddRef();
}
virtual ULONG __stdcall Release()
{
   return TInterfacedObject::_Release();
}
```

이러한 세 가지 메소드 구현을 *TInterfacedObject*의 자손에 추가함으로써 클래스는 완벽한 *IUnknown* 또는 *IInterface* 지원을 갖게 됩니다.

인터페이스 클래스 및 수명 관리

인터페이스 클래스의 *IUnknown* 메소드는 인터페이스를 구현하는 객체를 할당 및 해제하는 방법에 대한 함축 정보를 갖고 있습니다. *IUnknown*이 COM 객체에 의해 구현되면 이 객체는 *IUnknown* 메소드를 사용하여 해당 인터페이스에 대해 사용 중인 참조 수를 추적합니다. 이 참조 카운트가 0에 도달하면 COM 객체는 자신을 자동으로 해제합니다. *TInterfacedObject*는 같은 타입의 수명 관리를 수행하기 위해 *IUnknown* 메소드를 구현합니다.

그러나 이러한 *IUnknown* 메소드 구현은 반드시 필요한 것은 아닙니다. 예를 들어, *TComponent*가 제공하는 *IUnknown* 메소드의 디폴트 구현은 인터페이스에 대한 참조 카운트를 무시합니다. 따라서 이 컴포넌트는 참조 카운트가 0에 도달하더라도 자신을 해제하지 않습니다. 이것은 *TComponent*가 해당 *Owner* 속성에 지정된 객체에 의존하여 자신을 해제하기 때문입니다.

일부 컴포넌트는 이러한 두 모델을 혼합하여 사용합니다. 해당 *Owner* 속성이 NULL인 경우, 이러한 컴포넌트는 수명 관리를 위해 인터페이스에 대한 참조 카운트를 사용하여 참조 카운트가 0에 도달하면 자신을 해제합니다. 이러한 컴포넌트에 소유자가 있는 경우, 참조 카운팅은 무시되고 소유자가 컴포넌트를 해제할 수 있습니다. 그러나 애플리케이션 객체만 만들고 객체로부터 인터페이스를 얻지 않을 경우에는 이러한 혼합 객체 뿐만 아니라 수명 관리를 위해 참조 카운팅을 사용하는 다른 객체에서도 객체가 자동으로 해제되지 않는다는 점에 주의합니다.

객체 구분 인스턴스화

C++에서 클래스의 인스턴스는 실제 객체입니다. 이 객체는 직접 처리하거나 객체에 대한 참조 또는 포인터를 통해 간접적으로 액세스할 수 있습니다. 예를 들어, 인수가 없는 생성자가 포함된 *CPP_class*라는 C++ 클래스의 경우 다음 인스턴스 변수는 이 클래스에 대해 모두 유효합니다.

```
CPP_class by_value;// an object of type CPP_class
CPP_class& ref = by_value;// a reference to the object by_value, above
CPP_class* ptr = new CPP_class();// a pointer to an object of type
CPP_class
```

반대로 오브젝트 파스칼에서는 *object* 타입 변수가 객체를 항상 간접적으로 참조합니다. 모든 객체의 메모리는 동적으로 할당됩니다. 예를 들어, 다음과 같은 *OP_class* 오브젝트 파스칼 클래스를 가정할 수 있습니다.

```
ref: OP_class;
ref := OP_class.Create;
```

여기서 *ref*는 *OP_class* 타입 객체에 대한 "참조"입니다. 위 클래스는 C++Builder 코드로 변환되면 다음과 같이 나타납니다.

```
OP_class* ref = new OP_class;
```

C++ 및 오브젝트 파스칼 참조 구별

설명서에서는 흔히 오브젝트 파스칼 클래스 인스턴스 변수를 참조로 나타내지만 그 동작은 포인터의 동작과 동일하다고 설명합니다. 이것은 참조와 포인터의 속성을 이 변수가 모두 갖고 있기 때문입니다. 오브젝트 파스칼 객체 참조는 다음 예외를 가진 C++ 포인터와 동일합니다.

- 오브젝트 파스칼 참조는 암시적으로 역참조됩니다. 이 경우 C++ 참조와 더 유사하게 작동합니다.
- 오브젝트 파스칼 참조에는 정의된 작업과 같이 포인터 산술이 없습니다.

오브젝트 파스칼 참조를 C++ 참조와 비교할 때는 유사점과 차이점도 존재합니다. 두 랭귀지 모두에서 참조는 암시적으로 역참조되지만, 다음과 같은 차이점이 있습니다.

- 오브젝트 파스칼 참조는 다시 연결할 수 있지만 C++ 참조는 다시 연결할 수 없습니다.
- 오브젝트 파스칼 참조는 **nil**이 될 수 있지만 C++ 참조는 유효한 객체를 참조해야 합니다.

VCL 및 CLX 프레임워크의 기초가 되는 일부 디자인 결정은 이러한 타입의 인스턴스 변수 사용을 바탕으로 합니다. 포인터는 오브젝트 파스칼 참조와 가장 유사한 C++ 랭귀지 구조입니다. 결과적으로 C++Builder에서 거의 모든 VCL 및 CLX 객체 구분 식별자는 C++ 포인터로 변환됩니다.

참고 오브젝트 파스칼 **var** 매개변수 타입은 C++ 참조와 거의 일치합니다. **var** 매개변수에 대한 자세한 내용은 13-16페이지의 "Var 매개변수"를 참조하십시오.

객체 복사

C++와 달리 오브젝트 파스칼에는 객체 복사를 위한 기본 컴파일러 지원이 없습니다. 이 단원에서는 VCL 스타일 클래스의 할당 생성자 및 복사 생성자에 대한 이러한 차이점이 어떤 영향을 미치는지에 대해 설명합니다.

할당 연산자

오브젝트 파스칼 할당 연산자(:=)는 클래스 할당 연산자(연산자=())가 아닙니다. 이 할당 연산자는 객체가 아니라 참조를 복사합니다. 다음 코드에서 *B*와 *C*는 모드 동일한 객체를 참조합니다.

```
B, C: TButton;
B:= TButton.Create(ownerCtrl);
C:= B;
```

위 예제는 C++Builder에서 다음 코드로 변환됩니다.

```
TButton *B = NULL;
TButton *C = NULL;
B = new TButton(ownerCtrl);
C = B;// makes a copy of the pointer, not the object
```

C++Builder에서 VCL 스타일 클래스는 할당 연산자에 대한 오브젝트 파스칼 랭귀지 규칙을 따릅니다. 이것은 다음 코드에서 역참조된 포인터 간의 할당이 포인터가 아닌 객체를 복사하려고 시도하기 때문에 유효하지 않다는 것을 의미합니다.

```
TVCLStyleClass *p = new TVCLStyleClass;
TVCLStyleClass *q = new TVCLStyleClass;
*p = *q;// not allowed for VCL style class objects
```

참고 VCL 스타일 클래스의 경우, C++ 구문을 사용한 참조 연결은 계속 유효합니다. 예를 들면, 다음 코드는 유효합니다.

```
TVCLStyleClass *ptr = new TVCLStyleClass;
TVCLStyleClass &ref = *ptr;// OK for VCL style classes
```

이 구문은 할당 연산자를 사용하는 것과 다르지만 여기에서 설명 및 비교를 위해 언급했습니다.

복사 생성자

오브젝트 파스칼에는 기본 복사 생성자가 없습니다. 결과적으로 C++Builder에서 VCL 스타일 클래스에는 기본 복사 생성자가 존재하지 않습니다. 다음 예제 코드는 복사 생성자를 사용하여 *TButton* 포인터의 작성을 시도합니다.

```
TButton *B = new TButton(ownerCtrl);
TButton *C = new TButton(*B);// not allowed for VCL style class objects
```

VCL 및 CLX 클래스의 기본 복사 생성자에 의존하는 코드를 작성해서는 안됩니다. C++Builder에서 VCL 스타일 클래스 객체의 복사본을 만들려면 객체를 복사하는 멤버 함수에 대한 코드를 작성합니다. 또는 VCL 및 CLX *TPersistent* 클래스의 자손이 *Assign* 메소드를 오버라이드하여 특정 객체에서 다른 객체로 데이터를 복사할 수 있습니다. 일반적으로 이 방법은 리소스 이미지를 포함하는 *TBitmap* 및 *TIcon*과 그래픽 클래스에서 사용됩니다. 결국 객체 복사 방법은 프로그래머(컴포넌트 작성자)가 결정할 수 있지만, 표준 C++에서 사용되는 일부 복사 방법을 VCL 스타일 클래스에 사용할 수 없다는 것에 주의해야 합니다.

함수 인수로서의 객체

앞에서 언급한 것처럼 C++와 오브젝트 파스칼에서의 인스턴스 변수는 서로 다릅니다. 객체를 함수 인수로 전달할 때 이 점을 고려해야 합니다. C++에서 객체는 값, 참조 또는 포인터에 의해 함수에 전달할 수 있습니다. 오브젝트 파스칼에서는 객체를 값에 의해 함수에 전달할 때, 객체 인수가 이미 객체에 대한 참조라는 점에 주의합니다. 따라서 실제 객체가 아니라 참조가 값에 의해 전달됩니다. 오브젝트 파스칼에는 C++와 같이 실제 객체를 값에 의해 전달하는 기능이 없습니다. 함수에 전달되는 VCL 스타일 객체는 오브젝트 파스칼 규칙을 따릅니다.

C++Builder VCL/CLX 클래스의 객체 생성

C++와 오브젝트 파스칼은 객체를 다르게 생성합니다. 이 단원에서는 이러한 차이점과 C++Builder에서 두 가지 객체 생성 방식을 결합하는 방법에 대해 설명합니다.

C++ 객체 생성

표준 C++에서는 가상 기본 클래스, 기본 클래스 및 파생된 클래스의 순서로 생성이 이루어집니다. C++ 구문은 생성자 초기화 리스트를 사용하여 기본 클래스 생성자를 호출합니다. 객체의 런타임 타입은 호출되고 있는 현재 생성자의 클래스에 대한 타입입니다. 가상 메소드 디스패칭은 객체의 런타임 타입 이후에 오고 생성 도중 적절하게 변경됩니다.

오브젝트 파스칼 객체 생성

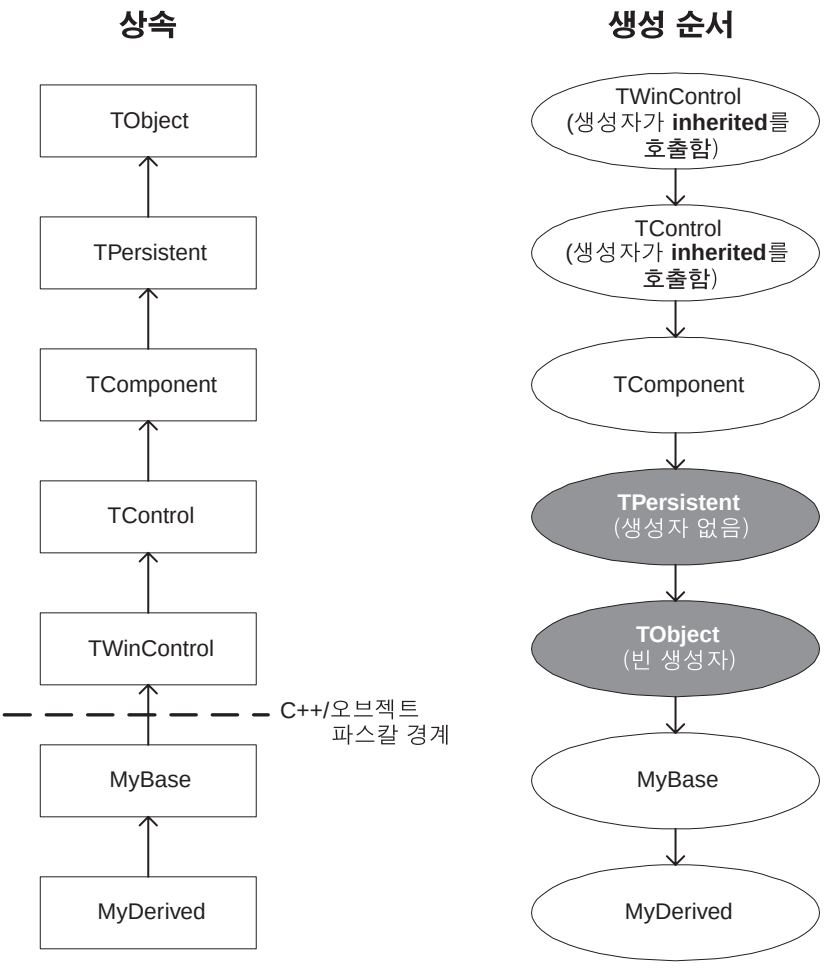
오브젝트 파스칼에서는 인스턴스화된 클래스의 생성자만 확실하게 호출되며, 기본 클래스 메모리의 할당이 수행됩니다. 각 직계 기본 클래스의 생성은 파생된 클래스의 해당 생성자에서 **inherited**를 호출하여 수행합니다. 규칙에 따라 VCL 및 CLX 클래스는 **inherited**를 사용하여 비어 있지 않은 기본 클래스 생성자를 호출합니다. 그러나 이 규칙은 오브젝트 파스칼에서 강제적인 요구 사항이 아니라는 점에 주의하십시오. 객체의 런타임 타입은 인스턴스화된 클래스의 런타임 타입처럼 즉시 설정되고, 기본 클래스 생성자가 호출될 때 바뀌지 않습니다. 가상 메소드 디스패칭은 객체의 런타임 타입 이후에 오기 때문에 생성 도중에 변경되지 않습니다.

C++Builder 객체 생성

VCL 스타일 객체는 오브젝트 파스칼 객체와 같이 생성되지만 C++ 구문을 사용합니다. 이것은 기본 클래스 생성자 호출의 방법 및 순서가 모든 VCL이 아닌 기본 클래스 및 CLX 기본 클래스와 첫 번째 직계 VCL 또는 CLX 조상에 대한 초기화 리스트를 사용하여 C++ 구문을 따른다는 것을 의미합니다. 이 VCL 또는 CLX 기본 클래스는 가정 먼저 생성되는 클래스이며, 경우에 따라 **inherited**를 사용하여 오브젝트 파스칼 메소드 뒤에 고유한 기본 클래스를 생성합니다. 따라서 VCL 및 CLX 기본 클래스는 C++와 반대 순서로 생성됩니다. 그런 다음 파생된 클래스에서 가장 먼 조상으로부터 모든 C++ 기본 클래스가 생성됩니다. 객체의 런타임 타입과 가상 메소드 디스패칭은 오브젝트 파스칼을 기본으로 합니다.

그림 13.1은 VCL 스타일 클래스인 *MyDerived*의 인스턴스가 생성되는 순서를 보여 줍니다. 이 VCL 스타일 클래스는 *TWinControl*의 직계 자손인 *MyBase*의 자손입니다. *MyDerived* 및 *MyBase*는 C++에서 구현됩니다. *TWinControl*은 오브젝트 파스칼에서 구현되는 VCL 클래스입니다.

그림 13.1 VCL 스타일 객체 생성 순서



실제 VCL 및 CLX 클래스의 *TObject*에 대한 최하위 조상에서 시작하여 *MyBase*를 생성한 다음 파생된 클래스를 최종적으로 생성하기 때문에 C++ 프로그래머에게는 생성 순서가 반대로 보일 수 있다는 점에 주의합니다.

참고 *TPersistent*에 생성자가 없기 때문에 *TComponent*는 **inherited**를 호출하지 않습니다. *TObject*는 비어 있는 생성자를 갖기 때문에 호출되지 않습니다. 이러한 클래스 생성자를 호출할 경우, 해당 클래스가 회색으로 표시된 위 다이어그램의 순서가 적용됩니다.

다음의 표 13.1은 C++, 오브젝트 파스칼 및 C++Builder에서의 객체 생성 모델을 요약한 것입니다.

표 13.1 객체 모델 비교

C++	오브젝트 파스칼	C++Builder
생성 순서		
가상 기본 클래스, 기본 클래스 및 파생된 클래스의 순서로 생성됩니다.	인스턴스화된 클래스 생성자가 자동으로 호출되는 첫 번째이자 유일한 생성자입니다. 이후에 생성되는 클래스는 최하위에서 루트 방향으로 생성됩니다.	직계 VCL 또는 CLX 기본 클래스가 가장 먼저 생성되고 오브젝트 파스칼 모델과 C++ 모델(가상 기본 클래스는 허용되지 않음)의 생성 순서가 차례로 적용됩니다.
기본 클래스 생성자 호출 방법		
생성자 초기화 리스트에서 자동으로 호출합니다.	경우에 따라 언제든지, 파생된 클래스 생성자의 바디에서 **inherited** 키워드를 사용하여 명시적으로 호출합니다.	VCL 또는 CLX 기본 클래스 생성자인 직계 조상을 통해 생성자 초기화 리스트에서 자동으로 호출합니다. 그런 다음 오브젝트 파스칼 메소드에 따라 **inherited**와 함께 생성자를 호출합니다.
생성 시 객체의 런타임 타입		
현재 생성자 클래스의 타입을 반영하여 변경됩니다.	인스턴스화된 클래스의 객체 런타임 타입과 같이 즉시 설정됩니다.	인스턴스화된 클래스의 객체 런타임 타입과 같이 즉시 설정됩니다.
가상 메소드 디스패칭		
기본 클래스 생성자가 호출될 때 객체의 런타임 타입에 따라 변경됩니다.	모든 클래스 생성자를 호출하는 과정에서 동일하게 남아 있는 객체의 런타임 타입을 따릅니다.	모든 클래스 생성자를 호출하는 과정에서 동일하게 남아 있는 객체의 런타임 타입을 따릅니다.

다음 단원에서는 이러한 차이점의 의미를 설명합니다.

기본 클래스 생성자에서의 가상 메소드 호출

VCL 또는 CLX 기본 클래스 생성자의 바디에서 호출된 가상 메소드, 즉 오브젝트 파스칼에서 구현되는 클래스는 객체의 런타임 타입에 따라 C++에서와 같이 디스패치됩니다. 객체의 런타임 타입을 즉시 설정하는 오브젝트 파스칼 모델과 파생된 클래스가 생성되기 전에 기본 클래스를 생성하는 C++ 모델이 C++Builder에서 결합되기 때문에, VCL 스타일 클래스에 대한 기본 클래스 생성자에서 가상 메소드를 호출하면 모호한 부작용이 발생할 수 있습니다. 아래에서는 최소한 하나 이상의 기본 클래스에서 파생된 인스턴스화된 클래스 예제를 통해 이러한 부작용에 대해 설명합니다. 여기서는 인스턴스화된 클래스를 파생된 클래스로 간주합니다.

오브젝트 파스칼 모델

오브젝트 파스칼에서 프로그래머는 파생된 클래스 생성자 바디의 어디에서든 기본 클래스 생성자를 호출할 수 있는 **inherited** 키워드를 사용할 수 있습니다. 따라서 파생된 클래스가 객체 설정이나 데이터 멤버 초기화에 의존하는 임의의 가상 메소드를 오버라이드하는 경우, 기본 클래스 생성자 및 가상 메소드를 호출하기 전에 이 키워드를 사용할 수 있습니다.

C++ 모델

C++ 구문에는 파생된 클래스의 생성 도중에 언제든지 기본 클래스 생성자를 호출하기 위한 **inherited** 키워드가 없습니다. C++ 모델의 경우에는 객체의 런타임 타입이 파생된 클래스가 아니라 생성 중인 현재 클래스의 런타임 타입이므로 **inherited**를 사용할 필요가 없습니다. 따라서 파생된 클래스가 아니라 현재 클래스의 가상 메소드가 호출됩니다. 결과적으로 이러한 메소드를 호출하기 전에 데이터 멤버를 초기화하거나 파생된 클래스 객체를 설정할 필요가 없습니다.

C++Builder 모델

C++Builder에서 VCL 스타일의 객체는 기본 클래스 생성자가 호출되는 동안 파생된 클래스의 런타임 타입을 가집니다. 따라서 기본 클래스 생성자가 가상 메소드를 호출할 경우 파생된 클래스가 가상 메소드를 오버라이드하면 파생된 클래스 메소드가 호출됩니다. 이 가상 메소드가 초기화 리스트나 파생된 클래스 생성자의 바디에 있는 임의의 항목에 의존할 경우 메소드는 이보다 앞서 호출됩니다. 예를 들어, *CreateParams*는 *TWinControl*의 생성자에서 간접적으로 호출되는 가상 멤버 함수입니다. *TWinControl*에서 클래스를 파생시키고 *CreateParams*를 오버라이드하여 생성자에 의존하게 만들면, 이 코드는 *CreateParams*가 호출된 후 처리됩니다. 이 상황은 기본 클래스의 모든 파생된 클래스에 적용됩니다. *A*에서 파생된 *B*에서 파생되는 클래스 *C*를 가정해 보십시오. *C*의 인스턴스를 만들 때, *B*만 메소드를 오버라이드하고 *C*는 그렇지 않을 경우 *A*는 *B*의 오버라이드된 메소드도 호출할 것입니다.

참고 *CreateParams*와 같은 가상 메소드는 생성자에서 분명히 호출되는 것이 아니라 간접적으로 호출됩니다.

예제: 가상 메소드 호출

다음 예제는 오버라이드된 가상 메소드가 있는 C++ 및 VCL 스타일 클래스를 비교합니다. 이 예제는 기본 클래스 생성자에서 이러한 가상 메소드에 대한 호출이 두 경우에 어떻게 해결되는지 보여 줍니다. *MyBase*와 *MyDerived*는 표준 C++ 클래스입니다. *MyVCLBase*와 *MyVCLDerived*는 *TObject*의 자손인 VCL 스타일 클래스입니다. 가상 메소드 *what_am_I()*는 파생된 두 클래스에서 오버라이드되지만, 파생된 클래스 생성자가 아니라 *단지* 기본 클래스 생성자에서만 호출됩니다.

```
#include <sysutils.hpp>
#include <iostream.h>
```

```
// non-VCL style classes
class MyBase {
public:
  MyBase() { what_am_I(); }
  virtual void what_am_I() {cout << "I am a base" << endl; }
};
class MyDerived : public MyBase {
public:
  virtual void what_am_I() {cout << "I am a derived" << endl; }
};
// VCL style classes
class MyVCLBase : public TObject {
public:
  __fastcall MyVCLBase() { what_am_I(); }
  virtual void __fastcall what_am_I() {cout << "I am a base" << endl; }
};
class MyVCLDerived : public MyVCLBase {
public:
  virtual void __fastcall what_am_I() {cout << "I am a derived" << endl;
}
};
int main(void)
{
  MyDerived d;// instantiation of the C++ class
  MyVCLDerived *pvd = new MyVCLDerived;// instantiation of the VCL style
class
  return 0;
}
```

이 예제의 출력은 다음과 같이 나타납니다.

```
I am a base
I am a derived
```

이 같은 출력이 나타나는 이유는 각각의 기본 클래스 생성자가 호출되는 동안 *MyDerived* 및 *MyVCLDerived*의 런타임 타입에 차이가 있기 때문입니다.

가상 함수의 데이터 멤버 초기화

가상 함수에서는 데이터 멤버가 사용될 수 있기 때문에 데이터 멤버를 초기화하는 시기 및 방법을 이해하는 것이 중요합니다. 오브젝트 파스칼에서 초기화되지 않은 모든 데이터는 0으로 초기화됩니다. 예를 들어, 이것은 생성자를 **inherited**를 통해 호출하지 않는 기본 클래스에 적용됩니다. C++에서는 초기화되지 않은 데이터 멤버에 적용되는 지정된 값이 없습니다. 다음 타입의 클래스 데이터 멤버는 클래스 생성자의 초기화 리스트에서 초기화해야 합니다.

- 참조
- 디폴트 생성자가 없는 데이터 멤버

그러나 이러한 데이터 멤버의 값 또는 생성자 바디에서 초기화되는 데이터 멤버는 기본 클래스 생성자가 호출될 때 정의되지 않습니다. C++Builder에서 VCL 스타일 클래스의 메모리는 0으로 초기화됩니다.

참고 기술적인 측면에서 보면 VCL 또는 CLX 클래스의 메모리가 0이며(즉, 비트가 0임) 실제로 값이 정의되는 것은 아닙니다. 예를 들어, 참조는 0입니다.

생성자 바디나 초기화 리스트에서 초기화되는 멤버 변수의 값에 의존하는 가상 함수는 변수가 0으로 초기화된 것처럼 동작할 수 있습니다. 이것은 초기화 리스트를 처리하거나 생성자 바디를 입력하기 전에 기본 클래스 생성자가 호출되기 때문입니다. 다음 예제에서 이러한 내용을 알 수 있습니다.

```
#include <sysutils.hpp>

class Base : public TObject {
public:
  __fastcall Base() { init(); }
  virtual void __fastcall init() { }
};

class Derived : public Base {
public:
  Derived(int nz) : not_zero(nz) { }
  virtual void __fastcall init()
  {
    if (not_zero == 0)
      throw Exception("not_zero is zero!");
  }
private:
  int not_zero;
};

int main(void)
{
  Derived *d42 = new Derived(42);
  return 0;
}
```

이 예제는 *Base*의 생성자에서 예외를 발생시킵니다. *Base*가 *Derived*보다 먼저 생성되므로 not_zero가 생성자에 전달된 값 42로 아직 초기화되지 않았습니다. 해당 기본 클래스 생성자를 호출하기 전에는 VCL 스타일 클래스의 데이터 멤버를 초기화할 수 없다는 것에 주의합니다.

객체 소멸

객체 소멸과 관련된 다음 두 개의 메커니즘이 C++와 오브젝트 파스칼에서 다르게 작동합니다.

- 생성자에서 발생된 예외로 인해 호출되는 소멸자
- 소멸자에서 호출되는 가상 메소드

VCL 스타일 클래스는 이러한 두 랭귀지의 방법을 결합합니다. 아래에는 이와 관련된 문제가 설명되어 있습니다.

생성자에서 발생되는 예외

객체 생성 도중에 예외가 발생될 경우 C++에서 소멸자는 오브젝트 파스칼에서와 다르게 호출됩니다. 예를 들어, 클래스 *A*에서 파생된 클래스 *B*에서 파생되는 클래스 *C*는 다음과 같습니다.

```
class A
{
  // body
};
class B: public A
{
  // body
};
class C: public B
{
  // body
};
```

*C*의 인스턴스를 생성할 때 클래스 *B*의 생성자에서 예외가 발생하는 경우를 가정합니다. 아래에는 C++, 오브젝트 파스칼 및 VCL 스타일 클래스에서 어떤 결과가 얻어지는지 설명되어 있습니다.

- C++에서는 *B*의 완전하게 생성된 객체 데이터 멤버에 대한 소멸자가 호출되고 *A*의 생성자가 호출된 다음 *A*의 완전하게 생성된 모든 데이터 멤버에 대한 소멸자가 호출됩니다. 그러나 *B*와 *C*의 소멸자는 호출되지 않습니다.
- 오브젝트 파스칼에서는 인스턴스화된 클래스 소멸자만 자동으로 호출됩니다. 이 소멸자는 *C*에 대한 소멸자입니다. 생성자의 경우와 마찬가지로 소멸자에서 **inherited**를 호출하는 것은 전적으로 프로그래머에게 달려 있습니다. 이 예제에서 모든 소멸자가 **inherited** 를 호출한다고 가정하면 *C, B* 및 *A*에 대한 소멸자가 이 순서대로 호출됩니다. 게다가 **inherited**가 *B*의 생성자에서 예외가 발생하기 전에 이미 호출되었는지 여부에 상관 없이 *A*의 소멸자가 호출됩니다. 이것은 **inherited**가 *B*의 소멸자에서 호출되었기 때문입니다. *A*의 소멸자 호출은 해당 생성자가 실제로 호출되었는지 여부에 상관하지 않습니다. 더 중요한 것은 생성자에서 **inherited**를 즉시 호출하는 것이 일반적이기 때문에 *C*의 소멸자는 해당 생성자의 바디가 완전히 실행되었는지 여부에 상관 없이 호출된다는 점입니다.
- VCL 스타일 클래스에서는 오브젝트 파스칼에서 구현되는 VCL 또는 CLX 기본 클래스가 오브젝트 파스칼의 소멸자 호출 방법을 따릅니다. C++에서 구현되는 파생된 클래스는 각 랭귀지에서의 방법을 정확하게 따르지 않습니다. 실제로 모든 소멸자를 호출하지만 C++ 랭귀지 규칙에 따라 호출되지 않는 이러한 소멸자의 바디는 입력되지 않습니다.

따라서 오브젝트 파스칼에서 구현되는 클래스는 소멸자의 바디에 작성하는 모든 정리 코드를 처리하기 위한 기회를 제공합니다. 여기에는 생성자 예외가 발생하기 전에 생성되는 하위 객체(객체인 데이터 멤버)의 메모리를 해제하는 코드가 포함됩니다. VCL 스타일 클래스의 경우, 소멸자가 호출된 경우에도 인스턴스화된 클래스나 C++에서 구현되는 자체의 기본 클래스에 대해 정리 코드가 처리되지 않을 수 있다는 것에 주의합니다.

C++Builder에서 예외를 처리하는 것에 대한 자세한 내용은 12-15페이지의 "VCL/CLX 예외 처리"를 참조하십시오.

소멸자에서 호출되는 가상 메소드

소멸자에서의 가상 메소드 디스패칭은 생성자와 동일한 패턴을 따릅니다. 이것은 VCL 스타일 클래스의 경우, 파생된 클래스가 가장 먼저 소멸되지만 객체의 런타임 타입이 기본 클래스 소멸자에 대한 이후의 호출 과정에서 파생된 클래스의 런타임 타입으로 남아 있는다는 것을 의미합니다. 따라서 VCL 또는 CLX 기본 클래스 소멸자에서 가상 메소드를 호출할 경우, 이미 자신을 소멸시킨 클래스로의 디스패칭이 수행될 수 있습니다.

AfterConstruction 및 BeforeDestruction

*TObject*는 두 개의 가상 메소드인 *BeforeDestruction*과 *AfterConstruction*을 도입합니다. 이러한 가상 메소드를 사용하면 프로그래머는 객체가 소멸 및 생성되기 전에 처리되는 코드를 작성할 수 있습니다. *AfterConstruction*은 마지막 생성자가 호출된 다음에 호출되고, *BeforeDestruction*은 첫 번째 소멸자가 호출되기 전에 호출됩니다. 이러한 메소드는 public이며 자동으로 호출됩니다.

클래스 가상 함수

오브젝트 파스칼에는 클래스 가상 함수의 개념이 있습니다. C++에는 정적 가상 함수가 존재할 수 있지만 이 타입의 함수와 정확하게 일치하는 것은 없습니다. 이러한 함수는 VCL 및 CLX에서 안전하게 내부적으로 호출됩니다. 그러나 C++Builder에서는 이 타입의 함수를 호출해서는 안됩니다. 이러한 함수의 앞에는 다음 주석이 오므로 헤더 파일에서 쉽게 식별할 수 있습니다.

```
/* virtual class method */
```

오브젝트 파스칼 데이터 타입 및 랭귀지 개념 지원

VCL 및 CLX를 지원하기 위하여 C++Builder는 대부분의 오브젝트 파스칼 데이터 타입, 구조 및 랭귀지 개념을 C++ 랭귀지로 구현, 변환 또는 매핑합니다. 이 작업은 다음과 같은 방법으로 수행됩니다.

- 원시 C++ 타입에 대한 Typedef
- 클래스, 구조체 및 클래스 템플릿
- C++ 랭귀지의 대응 부분
- 매크로
- ANSI 규격 랭귀지 확장인 키워드

오브젝트 파스칼 랭귀지의 모든 측면이 C++에 분명하게 매핑되는 것은 아닙니다. 경우에 따라 오브젝트 파스칼 랭귀지의 이러한 부분을 사용하면 애플리케이션에서 예기치 않은 동작이 발생할 수 있습니다. 예를 들면, 다음과 같습니다.

- 일부 타입은 오브젝트 파스칼과 C++에 모두 존재하지만 다르게 정의됩니다. 두 랭귀지 간에 코드를 공유할 때는 이러한 타입에 주의해야 합니다.
- C++Builder를 지원하기 위하여 오브젝트 파스칼에 일부 확장이 추가되었습니다. 경우에 따라 이러한 확장은 모호한 상호 운용성 문제를 일으킬 수 있습니다.
- C++ 랭귀지로 매핑되지 않는 오브젝트 파스칼 타입 및 랭귀지 구조는 이러한 랭귀지 간에 코드를 공유할 때 C++Builder에서 사용해서는 안됩니다.

이 단원에서는 C++Builder가 오브젝트 파스칼 랭귀지를 구현하는 방법을 개괄적으로 살펴보고 주의를 기울여야 하는 시점에 대해 설명합니다.

Typedef

대부분의 오브젝트 파스칼 기본 데이터 타입은 원시 C++ 타입에 대한 **typedef**를 통해 C++Builder에서 구현됩니다. 이러한 데이터 타입은 sysmac.h에서 확인할 수 있습니다. 가능하다면 오브젝트 파스칼 타입이 아니라 원시 C++ 타입을 사용해야 합니다.

오브젝트 파스칼 랭귀지를 지원하는 클래스

기본 C++ 대응 부분이 없는 일부 오브젝트 파스칼 데이터 타입 및 랭귀지 구조는 클래스나 구조체로 구현됩니다. 또한 특정 타입을 선언할 수 있는 **set**과 같은 클래스 템플릿을 사용하여 오브젝트 파스칼 데이터 타입과 랭귀지 구조를 구현할 수도 있습니다. 이러한 요소에 대한 선언은 다음 헤더 파일에서 확인할 수 있습니다.

- dstring.h
- wstring.h
- sysclass.h
- syscomp.h
- syscurr.h
- sysdyn.h
- sysopen.h
- sysset.h
- systdate.h
- systobj.h
- systvar.h
- sysvari.h

이러한 헤더 파일에서 구현되는 클래스는 오브젝트 파스칼 루틴에 사용되는 원시 타입을 지원하기 위해 작성되었으며, VCL 또는 CLX 기반 코드에서 이러한 루틴을 호출할 때 사용됩니다.

오브젝트 파스칼 랭귀지에 대한 C++ 랭귀지 대응 부분

오브젝트 파스칼의 **var** 매개변수와 타입이 지정되지 않은 매개변수는 C++에 대해 원시가 아니며, C++Builder에서 사용되는 C++ 랭귀지 대응 부분을 갖고 있습니다.

Var 매개변수

C++와 오브젝트 파스칼에는 수정 가능한 인수를 나타내는 "참조에 의한 전달"이라는 개념이 존재합니다. 오브젝트 파스칼에서는 이러한 인수를 **var** 매개변수라고 부릅니다. **var** 매개변수를 갖는 함수 구문은 다음과 같이 나타납니다.

```
procedure myFunc(var x : Integer);
```

C++에서는 이러한 타입의 매개변수를 참조에 의해 전달해야 합니다.

```
void myFunc(int& x);
```

객체를 수정하기 위하여 C++ 참조 및 포인터를 모두 사용할 수 있습니다. 참조나 포인터는 자신이 참조하는 대상의 *값*을 변경할 수 있지만, 참조는 포인터와 달리 다시 연결할 수 없고 **var** 매개변수는 다시 할당할 수 없기 때문에 **var** 매개변수에 더 가까운 것은 참조입니다.

타입이 지정되지 않은 매개변수

오브젝트 파스칼은 타입이 지정되지 않은 매개변수를 허용합니다. 이러한 매개변수는 타입이 정의되지 않은 상태로 함수에 전달됩니다. 받는 함수는 매개변수를 사용하기 전에 알려진 타입으로 변환해야 합니다. C++Builder는 타입이 지정되지 않은 매개변수를 void 포인터(*void* *)로 해석합니다. 받는 함수는 void 포인터를 원하는 타입의 포인터로 변환해야 합니다. 예를 들면, 다음과 같습니다.

```
int myfunc(void* MyName)
{
  // Cast the pointer to the correct type; then dereference it.
  int* pi = static_cast<int*>(MyName);
  return 1 + *pi;
}
```

개방형 배열

오브젝트 파스칼은 지정되지 않은 크기의 배열을 함수로 전달하도록 허용하는 "개방형 배열" 구조를 가집니다. C++에는 이 타입에 대한 직접적인 지원이 없지만, 배열의 첫 번째 요소에 대한 포인터와 마지막 인덱스 값(배열 요소 개수에서 1을 뺀 수)을 명시적으로 전달하여 개방형 배열 매개변수를 가진 오브젝트 파스칼 함수를 호출할 수 있습니다. 예를 들어, math.hpp의 *Mean* 함수는 오브젝트 파스칼에서 다음 선언을 가집니다.

```
function Mean(Data: array of Double): Extended;
```

C++ 선언은 다음과 같습니다.

```
Extended __fastcall Mean(const double * Data, const int Data_Size);
```

다음 코드는 C++에서 *Mean* 함수를 호출하는 것을 보여 줍니다.

```
double d[] = { 3.1, 4.4, 5.6 };

// explicitly specifying last index
long double x = Mean(d, 2);

// better: use sizeof to ensure that the correct value is passed
long double y = Mean(d, (sizeof(d) / sizeof(d[0])) - 1);

// use macro in sysopen.h
long double z = Mean(d, ARRAYSIZE(d) - 1);
```

참고 위 예제와 비슷하지만 오브젝트 파스칼 함수가 **var** 매개변수를 갖는 경우, C++ 함수 선언 매개변수는 **const**가 되지 않습니다.

요소 수 계산

sizeof(), ARRAYSIZE 매크로 또는 EXISTINGARRAY 매크로를 사용하여 배열의 요소 수를 계산할 때는 배열에 대한 포인터를 사용하지 않도록 주의합니다. 대신 배열 자체의 이름을 전달합니다.

```
double d[] = { 3.1, 4.4, 5.6 };
int n = ARRAYSIZE(d); // sizeof(d)/ sizeof(d[0]) => 24/8 => 3

double *pd = d;
int m = ARRAYSIZE(pd); // sizeof(pd)/sizeof(pd[0]) => 4/8 => 0 => Error!
```

배열의 "sizeof"를 가져오는 것은 포인터의 "sizeof"를 가져오는 것과 다릅니다. 예를 들어, 다음과 같이 선언했다고 가정합니다.

```
double d[3];
double *p = d;
```

배열의 크기를 가져오는 다음 코드는

```
sizeof(d)/ sizeof d[0]
```

포인트 크기를 가져오는 다음 코드와 다르게 분석됩니다.

```
sizeof(p)/ sizeof(p[0])
```

위의 예제와 이후의 코드에서는 *sizeof()* 연산자 대신 ARRAYSIZE 매크로가 사용됩니다. ARRAYSIZE 매크로에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

임시 개방형 배열

오브젝트 파스칼은 이름 없는 임시 개방형 배열을 함수에 전달하는 기능을 지원합니다. C++에는 이 작업을 수행하기 위한 구문이 없습니다. 그러나 변수 정의가 다른 명령문과 혼합될 수 있으므로 단순히 변수에 이름을 제공하는 방법을 사용할 수 있습니다.

오브젝트 파스칼의 경우는 다음과 같습니다.

```
Result := Mean([3.1, 4.4, 5.6]);
```

명명된 "임시 개방형 배열"을 사용하는 C++의 경우는 다음과 같습니다.

```
double d[] = { 3.1, 4.4, 5.6 };
return Mean(d, ARRAYSIZE(d) - 1);
```

명명된 "임시 개방형 배열" 범위를 제한하여 다른 로컬 변수와의 충돌을 방지하려면 다음과 같이 새 범위를 적절하게 여십시오.

```
long double x;
{
  double d[] = { 4.4, 333.1, 0.0 };
  x = Mean(d, ARRAYSIZE(d) - 1);
}
```

다른 해결 방법에 대한 자세한 내용은 13-19페이지의 "OPENARRAY 매크로"를 참조하십시오.

array of const

오브젝트 파스칼은 **array of const**라는 랭귀지 구조를 지원합니다. 이 인수 타입은 *TVarRec*의 개방형 배열을 값에 의해 가져오는 것과 동일합니다.

다음은 **array of const**를 승인하기 위해 선언된 오브젝트 파스칼 코드 세그먼트입니다.

```
function Format(const Format: string; Args: array of const): string;
```

C++에서의 프로토타입은 다음과 같습니다.

```
AnsiString __fastcall Format(const AnsiString Format,
     TVarRec const *Args, const int Args_Size);
```

함수는 개방형 배열을 가져오는 다른 함수와 동일하게 호출됩니다.

```
void show_error(int error_code, AnsiString const &error_msg)
{
  TVarRec v[] = { error_code, error_msg };
  ShowMessage(Format("%d: %s", v, ARRAYSIZE(v) - 1));
}
```

OPENARRAY 매크로

sysopen.h에 정의되는 OPENARRAY 매크로를 명명된 변수의 대안으로 사용하여 값에 의해 개방형 배열을 가져오는 함수에 임시 개방형 배열을 전달할 수 있습니다. 이 매크로는 다음과 같이 사용됩니다.

```
OPENARRAY(T, (value1, value2, value3)) // up to 19 values
```

여기서 *T*는 생성할 개방형 배열의 타입이고 *value* 매개변수는 배열을 채우는 데 사용됩니다. *value* 인수에는 괄호를 사용해야 합니다. 예를 들면, 다음과 같습니다.

```
void show_error(int error_code, AnsiString const &error_msg)
{
  ShowMessage(Format("%d: %s", OPENARRAY(TVarRec, (error_code,
error_msg))));
}
```

OPENARRAY 매크로를 사용하면 최대 19개의 값을 전달할 수 있습니다. 더 큰 배열이 필요한 경우에는 명시적 변수를 정의해야 합니다. 또한 OPENARRAY 매크로를 사용하면 원본으로 사용되는 배열을 할당하는 비용과 각 값의 추가 복사로 인해 약간의 런타임 비용이 추가로 발생합니다.

EXISTINGARRAY 매크로

sysopen.h에 정의되는 EXISTINGARRAY 매크로는 개방형 배열이 예상되는 곳에서 기존 배열을 전달하는 데 사용할 수 있습니다. 이 매크로는 다음과 같이 사용됩니다.

```
long double Mean(const double *Data, const int Data_Size);
double d[] = { 3.1, 3.14159, 2.17128 };
Mean(EXISTINGARRAY (d));
```

참고 13-17페이지의 "요소 수 계산" 단원에서도 EXISTINGARRAY 매크로에 대해 설명합니다.

개방형 배열 인수를 가져오는 C++ 함수

오브젝트 파스칼에서 개방형 배열이 전달되는 C++ 함수를 작성할 때는 "값에 의한 전달" 의미론을 명시적으로 유지하는 것이 중요합니다. 특히 함수에 대한 선언이 "값에 의한 전달"에 해당할 경우, 임의의 요소를 수정하기 전에 명시적으로 복사해야 합니다. 오브젝트 파스칼에서 개방형 배열은 기본 타입이며 값에 의해 전달할 수 있습니다. C++에서 개방형 배열 타입은 포인터를 통해 구현되므로 로컬 복사본을 만들지 않으면 원본 배열이 수정됩니다.

다르게 정의되는 타입

일반적으로 오브젝트 파스칼과 C++에서 다르게 정의되는 타입은 크게 중요하지 않지만, 이러한 타입이 문제가 되는 드문 경우를 파악하기가 어려울 수 있습니다. 따라서 이 단원에서는 이러한 타입에 대해 설명합니다.

부울 데이터 타입

오브젝트 파스칼 *ByteBool*, *WordBool* 및 *LongBool* 데이터 타입의 *True* 값은 오브젝트 파스칼에서 –1로 표시되고, *False*는 0으로 표시됩니다.

참고 부울 데이터 타입은 바뀌지 않고 그대로입니다(true = 1, false = 0).

C++ BOOL 타입이 이러한 오브젝트 파스칼 타입을 정확하게 변환하지만, WinAPI 함수나 1로 표시되는 Windows의 BOOL 타입을 사용하는 다른 함수를 공유할 때 문제가 발생합니다. 즉, BOOL 타입의 매개변수에 전달된 값은 오브젝트 파스칼에서는 –1로 분석되고 C++에서는 1로 분석됩니다. 따라서 이러한 두 랭귀지 간에 코드를 공유할 때, 두 식별자는 모두 0(*False*, false)이 아닌 경우 실패할 수 있습니다. 이에 대한 해결 방법으로 다음 비교 방법을 사용할 수 있습니다.

```
!A == !B;
```

표 13.2는 이러한 동등 비교 방법을 사용한 결과를 보여 줍니다.

표 13.2 BOOL 변수의 동등 비교 !A == !B

오브젝트 파스칼	C++	!A == !B
0(false)	0(false)	!0 == !0(TRUE)
0(false)	1(true)	!0 == !1(FALSE)
–1(true)	0(false)	!–1 == !0(FALSE)
–1(true)	1(true)	!–1 == !1(TRUE)

이 비교 방법을 사용하면 모든 값 집합이 정확하게 분석됩니다.

Char 데이터 타입

Char 타입은 C++에서는 부호 있는 타입이고 오브젝트 파스칼에서는 부호 없는 타입입니다. 코드를 공유할 때 이 차이점으로 인해 문제가 발생하는 경우는 거의 없습니다.

Delphi 인터페이스

오브젝트 파스칼 컴파일러는 인터페이스 사용을 위한 많은 세부 사항을 자동으로 처리합니다. 이 컴파일러는 애플리케이션 코드가 인터페이스 포인터를 획득하면 인터페이스에 대한 참조 카운트를 자동으로 증가시키고 인터페이스가 범위에서 벗어나면 참조 카운트를 자동으로 감소시킵니다.

C++Builder에서 *DelphiInterface* 템플릿 클래스는 이러한 편의성 중에서 일부를 C++ 인터페이스 클래스에 제공합니다. 오브젝트 파스칼에서 인터페이스 타입을 사용하는 VCL 및 CLX 속성과 메소드의 경우, C++ 랩퍼는 원본으로 사용하는 인터페이스 클래스를 통해 생성되는 *DelphiInterface*를 사용합니다.

DelphiInterface 생성자, 복사 생성자, 할당 연산자 및 소멸자는 모두 필요에 따라 참조 카운트를 증가 또는 감소시킵니다. 그러나 *DelphiInterface*는 오브젝트 파스칼의 인터페이스에 대한 컴파일러 지원만큼 편리하지는 않습니다. 원본으로 사용하는 인터페이스 포인터에 대한 액세스를 제공하는 다른 연산자는 참조 카운팅을 처리하지 않습니다. 이것은 클래스가 적절한 참조 카운팅에 대한 정보를 항상 제공할 수 있는 것은 아니기 때문입니다. 따라서 AddRef 또는 Release를 명시적으로 호출하여 적절한 참조 카운팅을 확인해야 할 수 있습니다.

리소스 문자열

리소스 문자열을 사용하는 파스칼 유닛에 코드가 있는 경우, 파스칼 컴파일러(DCC32)는 헤더 파일을 생성할 때 각 리소스 문자열에 대한 전역 변수와 해당 전처리기 매크로를 생성합니다. 리소스 문자열을 자동으로 로드하는 데 사용되는 이러한 매크로는 리소스 문자열이 참조되는 모든 곳에서 C++ 코드에 사용하도록 되어 있습니다. 예를 들어, 오브젝트 파스칼 코드의 **resourcestring** 섹션은 다음을 포함할 수 있습니다.

```
unit borrowed;
interface
resourcestring
  Warning = 'Be careful when accessing string resources.';
implementation
begin
end.
```

C++Builder의 파스칼 컴파일러가 생성하는 해당 코드는 다음과 같이 나타납니다.

```
extern PACKAGE System::Resource ResourceString _Warning;
#define Borrowed_Warning System::LoadResourceString(&Borrowed::_Warning)
```

이러한 코드는 *LoadResourceString*을 명시적으로 호출할 필요 없이 엑스포트된 오브젝트 파스칼 리소스 문자열을 사용할 수 있게 해 줍니다.

디폴트 매개변수

이제 파스칼 컴파일러는 생성자와 관련하여 C++와의 호환성을 위해 디폴트 매개변수를 승인합니다. C++와 달리 오브젝트 파스칼 생성자는 고유한 이름이 지정되기 때문에 매개변수의 개수 및 타입이 동일할 수 있습니다. 이 경우에는 C++ 헤더 파일이 생성될 때 오브젝트 파스칼 생성자를 구분하기 위하여 더미 매개변수가 이러한 생성자에 사용됩니다. 예를 들면, *TInCompatible*이라는 클래스의 경우 오브젝트 파스칼 생성자는 다음과 같을 수 있습니다.

```
constructor Create(AOwner: TComponent);
constructor CreateNew(AOwner: TComponent);
```

이러한 생성자는 C++에서 디폴트 매개변수 없이 다음의 모호한 코드로 변환될 것입니다.

```
__fastcall TInCompatible(Classes::TComponent* Owner); // C++ version of
the Pascal Create constructor

__fastcall TInCompatible(Classes::TComponent* Owner);// C++ version of
the Pascal CreateNew constructor
```

그러나 *TCompatible*이라는 클래스에 디폴트 매개변수가 사용될 경우 오브젝트 파스칼 생성자는 다음과 같습니다.

```
constructor Create(AOwner: TComponent);
constructor CreateNew(AOwner: TComponent; Dummy: Integer = 0);
```

이러한 생성자는 C++Builder에서 다음의 명확한 코드로 변환됩니다.

```
__fastcall TCompatible(Classes::TComponent* Owner);// C++ version of the
Pascal Create constructor

__fastcall TCompatible(Classes::TComponent* Owner, int Dummy);// C++
version of the Pascal CreateNew constructor
```

참고 디폴트 매개변수에 관한 주요 문제는 DCC32가 디폴트 매개변수의 기본값을 제거한다는 것입니다. 기본값을 제거하는 데 실패하면 기본값이 존재하지 않을 때 발생하는 모호함을 겪게 됩니다. VCL 또는 CLX 클래스를 사용하거나 외부 컴포넌트를 사용할 경우에는 이러한 점에 주의해야 합니다.

런타임 타입 정보

오브젝트 파스칼에는 RTTI를 처리하는 랭귀지 구조가 있으며, C++에 대응하는 구조가 존재할 수 있습니다. 표 13.3은 이러한 랭귀지 구조를 나열한 것입니다.

표 13.3 오브젝트 파스칼에서 C++로의 RTTI 매핑 예제

오브젝트 파스칼 RTTI	C++ RTTI
if Sender **is** TButton...	**if** (dynamic_cast <TButton*> (Sender) // dynamic_cast returns **NULL** on failure.
b := Sender **as** TButton; (* raises an exception on failure *)	TButton& ref_b = dynamic_cast <TButton&> (*Sender) // throws an exception on failure.
ShowMessage(Sender.ClassName);	ShowMessage(typeid(*Sender).name());

표 13.3에서 *ClassName*은 선언된 변수 타입에 상관 없이 객체의 실제 타입 이름이 포함된 문자열을 반환하는 *TObject* 메소드입니다. *TObject*에 사용되는 다른 RTTI 메소드는 대응하는 메소드가 C++에 존재하지 않습니다. 모두 public인 이러한 메소드를 나열하면 다음과 같습니다.

- *ClassInfo*는 객체 타입의 런타임 타입 정보(RTTI) 테이블에 대한 포인터를 반환합니다.
- *ClassNameIs*는 객체가 특정 타입인지 여부를 확인합니다.
- *ClassParent*는 클래스에 대해 직계 조상의 타입을 반환합니다. *TObject*의 경우에는 *TObject*에 부모가 없기 때문에 *ClassParent*는 nil을 반환합니다. 이 메소드는 **is** 및 **as** 연산자와 *InheritsFrom* 메소드에서 사용됩니다.
- *ClassType*은 객체의 실체 타입을 동적으로 확인합니다. 이 메소드는 오브젝트 파스칼 **is** 및 **as** 연산자에서 내부적으로 사용됩니다.
- *FieldAddress*는 RTTI를 사용하여 published 필드의 주소를 얻습니다. 이 메소드는 스트리밍 시스템에서 내부적으로 사용됩니다.

- *InheritsFrom*은 두 객체의 관계를 확인합니다. 이 메소드는 오브젝트 파스칼 **is** 및 **as** 연산자에서 내부적으로 사용됩니다.
- *MethodAddress*는 RTTI를 사용하여 메소드의 주소를 찾습니다. 이 메소드는 스트리밍 시스템에서 내부적으로 사용됩니다.

이러한 *TObject* 메소드 중 일부는 주로 컴파일러 또는 스트리밍 시스템에서 내부적으로 사용됩니다. 이러한 메소드에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

매핑되지 않은 타입

6바이트 실수 타입

이전의 오브젝트 파스칼 6바이트 부동 소수점 형식은 이제 *Real48*이라고 부릅니다. 이전의 *Real* 타입은 Double이 되었습니다. C++에는 *Real48* 타입에 해당하는 타입이 없습니다. 결과적으로 이 타입을 포함하는 오브젝트 파스칼 코드를 C++ 코드와 함께 사용하면 안됩니다. 그렇지 않으면 헤더 파일 생성기에서 경고를 생성합니다.

함수의 반환 타입으로서의 배열

오브젝트 파스칼에서 함수는 배열을 인수로 가져오거나 타입으로 반환할 수 있습니다. 예를 들어, 문자 80개의 배열을 반환하는 *GetLine* 함수의 구문은 다음과 같습니다.

```
type
  Line_Data = array[0..79] of char;
function GetLine: Line_Data;
```

C++에는 이 개념에 해당하는 부분이 없습니다. C++에서는 배열이 함수의 반환 타입으로 허용되지 않습니다. 또한 C++에서는 배열을 함수 인수의 타입으로 승인하지도 않습니다.

VCL 및 CLX은 배열인 속성을 갖지 않지만 오브젝트 파스칼 랭귀지에서 이를 허용한다는 점에 주의합니다. 속성 타입의 값을 가져와 반환하는 Get과 Set 읽기 및 쓰기 메소드가 속성에 사용될 수 있기 때문에, C++Builder에서는 타입 배열의 속성을 가질 수 없습니다.

참고 오브젝트 파스칼에서도 유효한 배열 속성은 C++에서 문제가 되지 않습니다. 이는 Get 메소드가 인덱스 값을 매개변수로 가져오고 Set 메소드가 배열에 포함된 타입의 객체를 반환하기 때문입니다. 배열 속성에 대한 자세한 내용은 47-8페이지의 "배열 속성 생성"을 참조하십시오.

키워드 확장

이 단원에서는 VCL 및 CLX를 지원하기 위해 C++Builder에서 구현되는 ANSI 규격의 키워드 확장에 대해 설명합니다. C++Builder의 키워드 및 키워드 확장에 대한 전체 리스트를 보려면 온라인 도움말을 참조하십시오.

__classid

컴파일러는 **__classid** 연산자를 사용하여 지정된 *classname*의 vtable에 대한 포인터를 생성합니다. 이 연산자는 클래스에서 메타클래스를 얻는 데 사용됩니다.

구문

```
__classid(classname)
```

예를 들어, **__classid**는 속성 에디터, 컴포넌트 및 클래스를 등록할 때 *TObject*의 *InheritsFrom* 메소드와 함께 사용됩니다. 다음 코드는 *TWinControl*에서 파생된 새 컴포넌트를 만들기 위하여 **__classid**가 사용되는 것을 보여 줍니다.

```
namespace Ywndctrl
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(MyWndCtrl)};
    RegisterComponents("Additional", classes, 0);
  }
}
```

__closure

__closure 키워드는 멤버 함수에 대한 특정 타입의 포인터를 선언하는 데 사용됩니다. 표준 C++에서 멤버 함수에 대한 포인터를 얻는 유일한 방법은 다음 예제에 표시된대로 전체 멤버 이름을 사용하는 것입니다.

```
class base
{
  public:
    void func(int x) { };
};

typedef void (base::* pBaseMember)(int);

int main(int argc, char* argv[])
{
  base        baseObject;
  pBaseMember m = &base::func; // Get pointer to member 'func'

  // Call 'func' through the pointer to member
  (baseObject.*m)(17);
  return 0;
}
```

그러나 파생된 클래스의 멤버에 대한 포인터를 기본 클래스의 멤버에 대한 포인터에 할당할 수 없습니다. *contravariance*라 부르는 이 규칙은 다음 예제에 나와 있습니다.

```
class derived: public base
{
  public:
    void new_func(int i) { };
};

int main(int argc, char* argv[])
{
  derived         derivedObject;
  pBaseMember m = &derived::new_func; // ILLEGAL

  return 0;
}
```

__closure 키워드 확장을 사용하면 이 제한을 벗어나는 등의 효과를 거둘 수 있습니다. 클로저(closure)를 사용하면 객체(즉, 클래스의 특정 인스턴스)의 멤버 함수에 대한 포인터를 얻을 수 있습니다. 여기서 객체는 상속 계층에 상관 없이 *모든* 객체가 될 수 있습니다. 객체의 **이** 포인터는 클로저(closure)를 통해 멤버 함수를 호출할 때 자동으로 사용됩니다. 다음 예제는 클로

저(closure)를 선언 및 사용하는 방법을 보여 줍니다. 이전에 제공된 *기본* 및 *파생된* 클래스는 정의된 것으로 가정합니다.

```
int main(int argc, char* argv[])
{
  derived         derivedObject;
  void (__closure *derivedClosure)(int);

     derivedClosure = derivedObject.new_func; // Get a pointer to the
'new_func' member.
                                              // Note the closure is
associated with the
                                              // particular object,
'derivedObject'.
     derivedClosure(3);  // Call 'new_func' through the closure.
     return 0;
}
```

또한 다음 예제에 표시된대로 클로저(closure)는 객체에 대한 포인터에도 사용됩니다.

```
void func1(base *pObj)
{
  // A closure taking an int argument and returning void.
  void ( __closure *myClosure )(int);

  // Initialize the closure.
  myClosure = pObj->func;

  // Use the closure to call the member function.
  myClosure(1);
  return;
}

int main(int argc, char* argv[])
{
  derived         derivedObject;
  void (__closure *derivedClosure)(int);

     derivedClosure = derivedObject.new_func; // Same as before...
     derivedClosure(3);

     // We can use pointers to initialize a closure, too.
     // We can also get a pointer to the 'func' member function
     // in the base class.
     func1(&derivedObject);
     return 0;
}
```

여기서는 *파생된* 클래스의 인스턴스에 대한 포인터를 전달하고, 이를 사용하여 *기본* 클래스에서 멤버 함수에 대한 포인터를 가져온다는 점에 주의합니다. 이는 표준 C++에서 허용하지 않는 작업입니다.

클로저(closure)는 C++ Builder RAD 환경의 핵심 부분입니다. 클로저(closure)를 사용하면 Object Inspector에서 이벤트 핸들러를 할당할 수 있습니다. 예를 들어, *TButton* 객체는 *OnClick* 이라는 이벤트를 가집니다. *TButton* 클래스에서 *OnClick* 이벤트는 자체의 선언에서 **__closure**

키워드 확장을 사용하는 속성입니다. **__closure** 키워드를 사용하면 다른 클래스의 멤버 함수(대개 *TForm* 객체이 있는 멤버 함수)를 이 속성에 할당할 수 있습니다. *TButton* 객체를 폼에 두고 버튼의 *OnClick* 이벤트에 대한 핸들러를 만들면 C++ Builder는 버튼의 *TForm* 부모에서 멤버 함수를 만들고 이 멤버 함수를 *TButton*의 *OnClick* 이벤트에 할당합니다. 이와 같은 방법으로 이벤트 핸들러는 *TButton*의 특정 인스턴스에 연결됩니다.

이벤트와 클로저(closure)에 대한 자세한 내용은 48장, "이벤트 생성"을 참조하십시오.

__property

__property 키워드는 클래스의 어트리뷰트(attribute)를 선언합니다. 속성은 클래스의 다른 어트리뷰트(필드)처럼 프로그래머에게 나타납니다. 그러나 오브젝트 파스칼에서와 마찬가지로 C++ Builder의 **__property** 키워드는 어트리뷰트 값을 검사 및 변경하는 것 이상의 많은 기능을 추가합니다. 속성 어트리뷰트가 속성에 대한 액세스를 완벽하게 제어하므로 클래스 자체 내에서 속성을 구현하는 방법에는 제한을 받지 않습니다.

구문

__property *type propertyName*[*index1Type index1*][*indexNType indexN*] = { *attributes* };

여기서

- *type*은 기본 또는 이전에 선언된 데이터 타입입니다.
- *propertyName*은 유효한 식별자입니다.
- *indexNType*은 기본 또는 이전에 선언된 데이터 타입입니다.
- *indexN*은 속성의 **read** 및 **write** 함수에 전달되는 인덱스 매개변수의 이름입니다.
- *attributes*는 **read**, **write**, **stored**, **default**(또는 **nodefault**) 또는 **index**의 쉼표로 구분된 시퀀스입니다.

대괄호 안의 *indexN* 매개변수는 옵션입니다. 이러한 인덱스 매개변수는 배열 속성을 선언하며, 배열 속성의 **read** 및 **write** 메소드에 전달됩니다.

다음 예제는 몇 가지 간단한 속성 선언을 보여 줍니다.

```
class PropertyExample {
  private:
    int Fx,Fy;
    float Fcells[100][100];

  protected:
    int  readX()          { return(Fx); }
    void writeX(int newFx) { Fx = newFx;  }
```

```
    double computeZ() {
        // Do some computation and return a floating point value...
        return(0.0);
    }


    float cellValue(int row, int col) { return(Fcells[row][col]); }

  public:
    __property int    X = { read=readX, write=writeX };
    __property int    Y = { read=Fy };
    __property double Z = { read=computeZ };
    __property float Cells[int row][int col] = { read=cellValue };
};
```

이 예제는 몇 가지 속성 선언을 보여 줍니다. 속성 *X*는 각각 *readX* 및 *writeX* 멤버 함수를 통해 읽기/쓰기 액세스를 가집니다. 속성 *Y*는 *Fy* 멤버 변수에 직접 대응되고 읽기 전용입니다. 속성 *Z*는 계산되는 읽기 전용 값입니다. 이 속성은 클래스에서 데이터 멤버로 저장되지 않습니다. 마지막으로 속성 *Cells*는 인덱스가 두 개인 배열 속성을 나타냅니다. 다음 예제는 코드에서 이러한 속성을 액세스하는 방법을 보여 줍니다.

```
PropertyExample myPropertyExample;
myPropertyExample.X = 42;            // Evaluates to:
myPropertyExample.WriteX(42);
int    myVal1 = myPropertyExample.Y; // Evaluates to: myVal1 =
myPropertyExample.Fy;
double myVal2 = myPropertyExample.Z; // Evaluates to: myVal2 =
myPropertyExample.ComputeZ();
float  cellVal = myPropertyExample[3][7]; // Evaluates to:
                                          // cellVal =
myPropertyExample.cellValue(3,7);
```

속성은 이 예제에 나타나지 않은 다른 많은 변형과 기능을 가집니다. 또한 속성은 다음 작업을 수행할 수 있습니다.

- **index** 어트리뷰트(attribute)를 사용하여 동일한 읽기 또는 쓰기 메소드를 둘 이상의 속성에 연결할 수 있습니다.
- 기본값을 가질 수 있습니다.
- 폼 파일에 저장할 수 있습니다.
- 기본 클래스에 정의된 속성을 확장할 수 있습니다.

속성에 대한 자세한 내용은 47장, "속성 생성"을 참조하십시오.

__published

__published 키워드는 클래스가 컴포넌트 팔레트에 있는 경우 해당 섹션의 속성이 Object Inspector에 표시되도록 지정합니다. *TObject*에서 파생된 클래스만 **__published** 섹션을 가질 수 있습니다.

published 멤버의 가시성 규칙은 public 멤버의 가시성 규칙과 동일합니다. published 멤버와 public 멤버의 유일한 차이점은 오브젝트 파스칼 스타일의 런타임 타입 정보(RTTI)가 **__published** 섹션에서 선언된 데이터 멤버 및 속성에 대해서 생성된다는 것입니다. RTTI는 애플리케이션이 알 수 없는 클래스 타입의 데이터 멤버, 멤버 함수 및 속성을 동적으로 쿼리할 수 있게 합니다.

참고 **__published** 섹션에서는 생성자 또는 소멸자가 허용되지 않습니다. 속성, 파스칼 기본 데이터 멤버나 파생된 VCL 또는 CLX 데이터 멤버, 멤버 함수, 클로저(closure) 등은 **__published** 섹션에서 사용할 수 있습니다. **__published** 섹션에서 정의되는 필드는 클래스 타입이어야 합니다. **__published** 섹션에서 정의되는 속성은 배열 속성이 될 수 없습니다. 또한 **__published** 섹션에서 정의되는 속성 타입은 순서 타입, 실수 타입, 문자열 타입, 작은 집합 타입, 클래스 타입 또는 메소드 포인터 타입이어야 합니다.

__declspec 키워드 확장

__declspec 키워드 확장에 대한 일부 인수는 VCL 및 CLX에 대한 랭귀지 지원을 제공합니다. 이러한 인수는 아래 나열되어 있습니다. declspec 인수와 그 조합에 사용되는 매크로는 sysmac.h에 정의됩니다. 대부분의 경우 이러한 인수와 그 조합은 지정할 필요가 없으며, 추가할 필요가 있는 경우에는 해당 매크로를 사용해야 합니다.

__declspec(delphiclass)

delphiclass 인수는 *TObject*에서 파생된 클래스의 선언에 사용됩니다. 이러한 클래스는 다음 호환성을 포함하도록 만들어집니다.

- 오브젝트 파스칼과 호환 가능한 RTTI
- VCL(CLX)과 호환 가능한 생성자/소멸자 동작
- VCL(CLX)과 호환 가능한 예외 처리

VCL 또는 CLX과 호환 가능한 클래스에는 다음 제한이 적용됩니다.

- 가상 기본 클래스가 허용되지 않습니다.
- 13-2 페이지의 "상속 및 인터페이스"에 설명된 경우 이외에는 복수 상속이 허용되지 않습니다.
- 새로운 전역 연산자를 사용하여 동적으로 할당해야 합니다.
- 소멸자를 가져야 합니다.
- 파생된 VCL 또는 CLX 클래스에 대해 복사 소멸자 및 할당 연산자가 컴파일러에서 생성되지 않습니다.

*TObject*에서 클래스가 파생된다는 사실을 컴파일러가 알아야 할 경우, 오브젝트 파스칼에서 변환되는 클래스 선언에는 이 변경자가 필요합니다.

__declspec(delphireturn)

delphireturn 인수는 C++Builder에서 VCL 및 CLX에 의해 내부적으로만 사용됩니다. 이 인수는 원시 C++ 타입이 없는 오브젝트 파스칼의 기본 데이터 타입과 랭귀지 구조를 지원하기 위해 C++Builder에서 작성된 클래스를 선언하는 데 사용되며, *Currency*, *AnsiString*, *Variant*, *TDateTime* 및 *Set*이 여기에 포함됩니다. **delphireturn** 인수는 함수 호출에서 VCL 또는 CLX와 호환 가능한 처리를 위한 C++ 클래스를 매개변수 및 반환값으로 표시합니다. 오브젝트 파스칼과 C++ 간에 구조를 값에 의해 함수로 전달할 때 이 변경자가 필요합니다.

__declspec(delphirtti)

delphirtti 인수는 컴파일러가 컴파일 시에 런타임 타입 정보를 클래스에 포함하도록 지시합니다. 이 변경자가 사용되면 컴파일러는 published 섹션에서 선언된 모든 필드, 메소드 및 속성에 대해 런타임 타입 정보를 생성합니다. 인터페이스의 경우, 컴파일러는 인터페이스의 모든 메소드에 대한 런타임 타입 정보를 생성합니다. 클래스가 런타임 타입 정보와 함께 컴파일되면 클래스의 모든 자손도 런타임 타입 정보를 포함합니다. *TPersistent* 클래스가 런타임 타입 정보와 함께 컴파일되기 때문에, 이는 *TPersistent*를 조상으로 갖는 모든 작성된 클래스에서 이 변경자를 사용할 필요가 없다는 것을 의미합니다. 이 변경자는 주로 웹 서비스를 구현 또는 사용하는 애플리케이션의 인터페이스에 사용됩니다.

__declspec(dynamic)

dynamic 인수는 동적 함수의 선언에 사용됩니다. 동적 함수는 vtable이 정의되는 객체의 vtable에만 저장되고 자손 vtable에는 저장되지 않는다는 점을 제외하고 가상 함수와 비슷합니다. 객체에 정의되지 않은 동적 함수를 호출할 경우, 이 함수가 발견될 때까지 해당 조상의 vtable에서 검색이 이루어집니다. 동적 함수는 *TObject*에서 파생된 클래스에만 유효합니다.

__declspec(hidesbase)

hidesbase 인수는 오브젝트 파스칼 가상 및 오버라이드 함수를 C++Builder에 이식할 때 오브젝트 파스칼 프로그램 의미론을 유지합니다. 오브젝트 파스칼에서 기본 클래스의 가상 함수는 파생된 클래스에 동일한 이름의 함수로 표시될 수 있지만, 이전 함수와의 명시적 관계가 존재하지 않는 완전히 새로운 함수가 되어야 합니다.

컴파일러는 sysmac.h에 정의된 HIDESBASE 매크로를 사용하여 이러한 타입의 함수 선언이 완전히 별개라는 사실을 지정합니다. 예를 들어, 기본 클래스인 T1이 인수가 없는 *func*라는 가상 함수를 선언하고 파생된 클래스인 T2가 이름 및 시그너처가 동일한 함수를 선언한 경우, DCC32 -jphn은 다음 프로토타입이 포함된 HPP 파일을 생성합니다.

```
virtual void T1::func(void);
HIDESBASE void T2::func(void);
```

C++ 프로그램 의미론은 HIDESBASE 선언을 사용하지 않고 *T1::func()* 가상 함수가 *T2::func()*에 의해 오버라이드되고 있다는 것을 나타냅니다.

__declspec(package)

package 인수는 클래스를 정의하는 코드를 package에서 컴파일할 수 있다는 것을 나타냅니다. 이 변경자는 IDE에서 패키지를 만들 때 컴파일러에 의해 자동으로 생성됩니다. 패키지에 대한 자세한 내용은 15장, "패키지와 컴포넌트 사용"을 참조하십시오.

__declspec(pascalimplementation)

pascalimplementation 인수는 클래스를 정의하는 코드가 오브젝트 파스칼에서 구현되었다는 것을 나타냅니다. 이 변경자는 확장자가 .hpp인 오브젝트 파스칼 이식성 헤더 파일에 표시됩니다.

__declspec(uuid)

uuid 인수는 클래스를 GUID(Globally Unique Identifier)와 연결합니다. 이 변경자는 모든 클래스에서 사용할 수 있지만, 일반적으로 오브젝트 파스칼 인터페이스 또는 COM 인터페이스를 나타내는 클래스에 사용됩니다. 이 변경자를 사용하여 선언한 클래스의 GUID는 **__uuidof** 지시어를 호출하여 검색할 수 있습니다.

장

14

크로스 플랫폼 애플리케이션 개발

C++Builder를 사용하면 Windows 및 Linux 운영 체제 모두에서 실행되는 크로스 플랫폼 32비트 애플리케이션을 개발할 수 있습니다. 크로스 플랫폼 애플리케이션은 CLX 컴포넌트를 사용하며 특정 운영 체제의 API를 호출하지 않습니다. 크로스 플랫폼 애플리케이션을 개발하려면 새 CLX 애플리케이션을 만들거나 기존 Windows 애플리케이션을 수정하십시오. 그런 다음 애플리케이션이 실행되는 플랫폼에서 해당 애플리케이션을 컴파일 및 배포합니다. Windows의 경우에는 C++Builder를 사용합니다. Linux의 경우, 아직 Borland C++ 솔루션을 사용할 수 없지만 C++Builder로 애플리케이션을 개발하여 미리 준비할 수 있습니다.

이 장에서는 Linux에서 컴파일할 수 있도록 C++Builder 애플리케이션을 변경하는 방법과 플랫폼에 독립적이고 두 환경 간에 이식할 수 있는 코드를 작성하는 방법에 대해 설명합니다. 또한 Windows 및 Linux에서의 애플리케이션 개발 간에 어떤 차이점이 있는지에 대해 설명합니다.

크로스 플랫폼 애플리케이션 생성

크로스 플랫폼 애플리케이션은 다른 C++Builder 애플리케이션과 동일한 방법으로 만듭니다. CLX 비주얼(visual) 및 넌비주얼(nonvisual) 컴포넌트를 사용해야 하고, 완벽한 크로스 플랫폼 애플리케이션을 원할 경우 특정 운영 체제의 API를 사용해서는 안됩니다. 크로스 플랫폼 애플리케이션 생성에 대한 팁을 보려면 14-15페이지의 "이식 가능한 코드 작성"을 참조하십시오.

다음과 같은 방법으로 크로스 플랫폼 애플리케이션을 만듭니다.

1 IDE에서 File|New|CLX application을 선택합니다.
컴포넌트 팔레트에 CLX 애플리케이션에서 사용할 수 있는 페이지와 컴포넌트가 나타납니다.

2 IDE 내에서 애플리케이션을 개발합니다.

3 애플리케이션을 컴파일하여 테스트합니다 . 추가로 변경해야 할 곳이 없는지를 알아보기 위해서 모든 오류 메시지를 검토합니다.

참고 C++Builder 솔루션을 Linux에서 사용할 수 있게 되면 Linux에서 애플리케이션을 컴파일하여 테스트할 수 있습니다.

애플리케이션을 Linux로 이식한 경우 프로젝트 옵션을 다시 설정해야 합니다. 이것은 프로젝트 옵션을 저장하는 .dof 파일이 Linux에서 디폴트 옵션이 설정된 상태에서 다른 확장자로 다시 만들어지기 때문입니다.

크로스 플랫폼 애플리케이션의 폼 파일은 .dfm 대신 .xfm이라는 확장자를 가집니다. 따라서 CLX 컴포넌트를 사용하는 크로스 플랫폼 폼은 VCL 컴포넌트를 사용하는 폼과 구별됩니다. .xfm 폼 파일은 Windows와 Linux에서 모두 작동하지만 .dfm 폼 파일은 Windows에서만 작동합니다.

플랫폼 독립적인 데이터베이스 또는 인터넷 애플리케이션을 작성하는 것에 대한 자세한 내용은 14-19페이지의 "크로스 플랫폼 데이터베이스 애플리케이션" 및 14-25페이지의 "크로스 플랫폼 인터넷 애플리케이션"을 참조하십시오.

Windows 애플리케이션을 Linux로 이식

Windows 환경에 맞게 작성된 C++Builder 애플리케이션의 경우, Linux 환경에 맞게 준비할 수 있습니다. 이 작업의 난이도는 애플리케이션의 특성 및 복잡성과 명시적인 Windows 종속 정도에 따라 다릅니다.

다음 단원에서는 Windows와 Linux 환경 간의 주요한 차이점 몇 가지를 설명하고 애플리케이션 이식 입문에 대한 지침을 제공합니다.

이식 기법

다음은 애플리케이션을 특정 플랫폼에서 다른 플랫폼으로 이식할 때 취할 수 있는 여러 가지 접근 방법입니다.

표 14.1 이식 기법

기법	설명
플랫폼별 이식	운영 체제와 원본으로 사용하는 API를 대상으로 함
크로스 플랫폼 이식	크로스 플랫폼 API를 대상으로 함
Windows 에뮬레이션	코드는 그대로 놔두고 코드에서 사용하는 API를 이식함

플랫폼별 이식

플랫폼별 이식은 시간이 오래 걸리고, 비용이 많이 들며 주로 해당 플랫폼에서만 애플리케이션을 사용할 수 있습니다. 플랫폼별 이식은 서로 다른 코드 기반을 생성하므로 유지 보수가 특히 어렵습니다. 그러나 각 이식은 특정 운영 체제를 위해 디자인된 것으로 플랫폼별 기능을 활용할 수 있습니다. 따라서 일반적으로 애플리케이션이 보다 빠르게 실행됩니다.

크로스 플랫폼 이식

크로스 플랫폼 이식에서는 이식된 애플리케이션이 여러 플랫폼을 대상으로 하기 때문에 시간이 절약되는 경향이 있습니다. 그러나 크로스 플랫폼 애플리케이션 개발에 포함되는 작업의 양은 기존 코드에 따라 결정됩니다. 코드가 플랫폼 독립성을 염두에 두지 않고 개발되었다면 플랫폼 독립적인 로직과 플랫폼 독립적인 구현이 같이 섞이는 경우가 있을 수 있습니다.

크로스 플랫폼 접근 방법은 비즈니스 로직이 플랫폼 독립적인 관점에서 개발되므로 더 선호됩니다. 일부 서비스는 모든 플랫폼에서 같아 보이지만 개별적으로는 특정한 구현을 가진 내부 인터페이스를 통해 추출됩니다. C++Builder의 런타임 라이브러리가 바로 이러한 예입니다. 두 플랫폼에서 인터페이스가 매우 유사하지만 구현 내용은 크게 다릅니다. 크로스 플랫폼 부분을 분리한 후 그 위에 특정 서비스를 구현해야 합니다. 결국 이 접근 방법은 폭 넓게 공유되는 소스 기반과 개선된 애플리케이션 아키텍처로 인한 유지 비용 감소를 통해 가장 비용이 적게 드는 솔루션입니다.

Windows 에뮬레이션 이식

Windows 에뮬레이션은 가장 복잡한 방법이며 비용이 많이 들 수 있지만 결과물인 Linux 애플리케이션은 기존의 Windows 애플리케이션과 가장 유사하게 보입니다. 이 접근 방법은 Linux에 Windows 기능을 구현합니다. 엔지니어링의 관점에서 볼 때 이 솔루션은 유지 보수가 매우 어렵습니다.

Windows API를 에뮬레이트할 곳에서는 **#ifdef**를 사용하여 각각의 섹션 두 개를 포함시켜 Windows 또는 Linux 전용 코드 섹션을 나타낼 수 있습니다.

애플리케이션 이식

Windows 및 Linux 모두에서 실행할 애플리케이션을 이식하는 경우, 코드를 수정하거나 **#ifdef**를 사용하여 Windows 또는 Linux에 특별히 적용되는 코드의 섹션을 나타냅니다.

VCL 애플리케이션을 CLX로 이식하려면 다음과 같은 일반적인 단계를 따르십시오.

1. C++Builder에서 변경할 애플리케이션이 포함된 프로젝트를 엽니다.
2. .dfm 파일을 동일한 이름의 .xfm 파일로 복사합니다(예를 들어, unit1.dfm의 이름을 unit1.xfm으로 변경). 헤더 파일의 .dfm 파일 참조를 #pragma resource "*.dfm"에서 #pragma resource "*.xfm"으로 변경(또는 **#ifdef**)합니다. 이제 .xfm 파일은 C++Builder 및 Linux 애플리케이션에서 모두 작동합니다.
3. CLX의 올바른 유닛을 참조하도록 소스 파일의 모든 헤더 파일을 변경(또는 **#ifdef**)합니다(자세한 내용은 14-8페이지의 "CLX 및 VCL 유닛 비교" 참조).

   예를 들어, Windows 애플리케이션의 헤더 파일에 다음 #include 문이 있다고 가정합니다.

   ```
   #include <vcl.h>
   #include <Controls.hpp>
   #include <StdCtrls.hpp>
   #include <Forms.hpp>
   ```

위 #include 문을 CLX 애플리케이션에 맞게 다음과 같이 변경합니다.

```
#include <clx.h>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
```

4 프로젝트를 저장하고 다시 엽니다. 그러면 컴포넌트 팔레트에 CLX 애플리케이션에서 사용할 수 있는 컴포넌트가 표시됩니다.

참고 일부 Windows 전용 넌비주얼(nonvisual) 컴포넌트는 CLX 애플리케이션에서 사용할 수 있지만, 이러한 컴포넌트에는 Windows CLX 애플리케이션에서 작동하는 기능만 들어 있습니다. Linux에서도 애플리케이션을 컴파일할 계획이면 애플리케이션에서 넌비주얼 VCL 컴포넌트를 사용하지 않거나 **#ifdef**를 사용하여 이러한 코드 섹션을 Windows 전용으로 표시합니다. 이 경우, 동일한 애플리케이션에서 VisualCLX와 함께 VCL의 비주얼 부분을 사용할 수 없습니다.

5 Windows 종속성이 필요한 코드를 다시 작성하여 코드를 더 플랫폼 독립적으로 만듭니다. 이 작업은 런타임 라이브러리 루틴 및 상수를 사용하여 수행합니다(자세한 내용은 14-15페이지의 "이식 가능한 코드 작성" 참조).

6 Linux에서 다른 기능에 대해 동등한 기능을 찾습니다. 필요한 경우 **#ifdef**를 사용하여 Windows 특정 정보를 구분합니다(자세한 내용은 14-16페이지의 "조건 지시어 사용" 참조).

예를 들어, 소스 파일에서 다음과 같이 플랫폼 특정 코드를 **#ifdef**할 수 있습니다.

```
#ifdef WINDOWS //If using the C++Builder compiler, use __WIN32__
IniFile->LoadfromFile("c:\\x.txt");
#endif

#ifdef __linux__
IniFile->LoadfromFile("/home/name/x.txt");
#endif
```

7 모든 프로젝트 파일에서 경로명에 대한 참조를 검색합니다.

- Linux의 경로명은 슬래시(/)를 구분자로 사용하며(예를 들어, /usr/lib) Linux 시스템에서는 파일들이 다른 디렉토리에 위치할 수 있습니다. SysUtils에서 PathDelim 상수를 사용하여 시스템에 맞는 경로 구분자를 지정합니다. Linux 상의 파일에 대한 정확한 위치를 정합니다.
- 드라이브 문자에 대한 참조(예를 들어, C:\)와 문자열에서 두 번째 위치에 있는 콜론을 찾아서 드라이브 문자를 검색하는 코드를 변경합니다. SysUtils에서 DriveDelim 상수를 사용하여 시스템에 맞게 위치를 지정합니다.
- 여러 경로를 지정하는 경우 경로 구분자를 세미콜론(;)에서 콜론(:)으로 변경합니다. SysUtils에서 PathSep 상수를 사용하여 시스템에 맞는 경로 구분자를 지정합니다.
- Linux에서는 파일 이름의 대소문자를 구별하므로 애플리케이션이 파일 이름의 대소문자를 변경하거나 대소문자 중 하나인 것으로 가정하지 않았는지 확인합니다.

8 애플리케이션을 컴파일, 테스트 및 디버깅합니다.

CLX와 VCL 비교

CLX 애플리케이션은 VCL(Visual Component Library) 대신 CLX(Cross Platform)용 Borland 컴포넌트 라이브러리를 사용합니다. VCL 내에서 대부분의 컨트롤은 Windows API 라이브러리로 호출을 하여 Windows 컨트롤을 액세스하는 쉬운 방법을 제공합니다. 마찬가지로 CLX는 Qt 공유 라이브러리로 호출을 하여 Qt widget(window와 gadget을 합한 것)에 대한 액세스를 제공합니다. C++Builder에는 CLX와 VCL이 모두 포함되어 있습니다.

CLX는 VCL과 매우 유사합니다. 대부분의 컴포넌트와 속성은 동일한 이름을 가집니다. 또한 C++Builder의 에디션에 따라 VCL 뿐만 아니라 CLX를 Windows에서 사용할 수 있습니다.

CLX 컴포넌트는 다음 부분으로 그룹화할 수 있습니다.

표 14.2 CLX 부분

부분	설명
VisualCLX	원시 크로스 플랫폼 GUI 컴포넌트와 그래픽
DataCLX	클라이언트 데이터 액세스 컴포넌트. 이 부분의 컴포넌트는 클라이언트 데이터셋에 기반한 로컬, 클라이언트/서버 및 n 티어의 부분 집합입니다. 코드는 Linux와 Windows에서 동일합니다.
NetCLX	Apache DSO 및 CGI Web Broker를 포함하는 인터넷 컴포넌트. Linux와 Windows에서 동일합니다.
BaseCLX	Classes 유닛까지를 포함하는 런타임 라이브러리. 코드는 Linux와 Windows에서 동일합니다.

VisualCLX의 widget은 Windows 컨트롤을 대체합니다. 예를 들어, CLX의 *TWidgetControl*은 VCL의 *TWinControl*을 대체합니다. *TScrollingWinControl*과 같은 다른 VCL 컴포넌트는 CLX에 *TScrollingWidget*과 같은 해당 이름을 가집니다. 그러나 *TWinControl*을 *TWidgetControl*로 변경할 필요는 없습니다. 예를 들어, 다음과 같은 타입 선언의 경우

```
TWinControl = TWidgetControl;
```

QControls 유닛 파일에 사용하여 소스 코드의 공유를 간소화합니다. *TWidgetControl*과 이 컨트롤의 모든 자손은 Qt 객체를 참조하는 *Handle* 속성과 이벤트 메커니즘을 처리하는 후크 객체를 참조하는 *Hooks* 속성을 가집니다.

CLX의 경우에는 유닛 이름과 일부 클래스의 위치가 다릅니다. 소스 파일에 포함하는 헤더 파일을 수정하여 CLX에 존재하지 않는 유닛에 대한 참조를 제거하고 이름을 CLX 유닛으로 변경해야 합니다.

CLX에서 다르게 구현되는 기능

많은 CLX가 VCL과 일관되도록 구현되지만 일부 기능은 다르게 구현됩니다. 이 단원에서는 크로스 플랫폼 애플리케이션을 작성할 때 알아야 하는 CLX 및 VCL 구현 간의 몇 가지 차이점을 개괄적으로 설명합니다.

룩앤필(Look and feel)

Linux의 비주얼 환경은 Windows와 약간 다르게 보입니다. 다이얼로그 박스의 모습은 KDE나 Gnome 등 사용 중인 윈도우 관리자에 따라 달라질 수 있습니다.

스타일

OwnerDraw 속성 외에 애플리케이션 전체 스타일을 사용할 수 있습니다. *TApplication::Style* 속성을 사용하면 애플리케이션의 그래픽 요소에 대한 룩앤필(look and feel)을 지정할 수 있습니다. 스타일을 사용하여 widget이나 애플리케이션이 완전히 새로운 모습을 갖출 수 있습니다. Linux에서 owner draw를 사용할 수 있지만 스타일을 사용하도록 권합니다.

가변 타입

시스템 유닛에 있던 모든 가변 타입/안전 배열 코드는 이제는 다음 두 개의 새로운 유닛입니다.

- Variants
- VarUtils

이제 운영 체제 종속 코드는 VarUtils 유닛으로 분리되고 Variants 유닛에 필요한 모든 것의 일반적인 버전을 포함합니다. Windows 호출이 포함된 VCL 애플리케이션을 CLX 애플리케이션으로 변환하는 경우, Windows 호출을 VarUtils 유닛의 호출로 바꿔야 합니다.

가변 타입을 사용하려면 헤더 파일에 대한 Variants 유닛을 소스 파일에 포함해야 합니다.

*VarIsEmpty*는 *varEmpty*에 대해 간단한 테스트를 수행하여 가변 타입이 분명한지 확인합니다. Linux에서는 *VarIsClear* 함수를 사용하여 가변 타입 값의 정의 여부를 확인할 수 있습니다.

레지스트리

Linux에서는 구성 정보를 저장하기 위한 레지스트리는 사용하지 않습니다. 레지스트리 대신 텍스트 설정 파일과 환경 변수를 사용합니다. 때로는 Linux에서 시스템 설정 파일이 /etc/hosts와 같이 /etc에 있는 경우도 있습니다. 다른 사용자 프로파일은 bash 셸 설정이 들어 있는 .bashrc 또는 X 프로그램의 기본값을 설정하는 데 사용하는 .XDefaults와 같은 히든 파일(점이 앞에 옴)에 있습니다.

레지스트리 종속 코드를 변경하여 로컬 구성 텍스트 파일을 대신 사용할 수 있습니다. 사용자가 변경할 수 있는 설정은 홈 디렉토리에 저장해야만 사용자가 쓰기 권한을 가질 수 있습니다. 루트가 설정해야 하는 구성 옵션은 /etc 디렉토리에 있어야 합니다. 모든 레지스트리 함수를 포함하지만 모든 출력을 로컬 설정 파일로 전달하는 유닛을 작성하는 것은 레지스트리에 대한 이전 종속 관계를 처리하는 한 방법입니다.

Linux에서 정보를 전역 위치에 두려면 전역 설정 파일을 /etc 디렉토리나 사용자의 홈 디렉토리에 숨김 파일로 저장합니다. 이렇게 하면 모든 애플리케이션이 동일한 설정 파일을 액세스할 수 있습니다. 그러나 파일 권한과 액세스 권한이 올바르게 설정되었는지 확인해야 합니다.

크로스 플랫폼 애플리케이션에서는 .ini 파일을 사용할 수도 있습니다. 그러나 CLX에서는 *TRegIniFile* 대신 *TMemIniFile*을 사용해야 합니다.

그 밖의 차이점

컴포넌트 작동에 영향을 미치는 방법에 있어서도 CLX 구현은 VCL과 차이가 있습니다. 이 단원에서는 이러한 차이점에 대해 설명합니다.

- 컴포넌트 팔레트에서 CLX 컴포넌트를 선택하고 왼쪽 또는 오른쪽 마우스 버튼을 클릭하여 폼에 추가할 수 있습니다. VCL 컴포넌트의 경우에는 왼쪽 마우스 버튼만 클릭할 수 있습니다.
- CLX *TButton* 컨트롤은 해당 VCL 컨트롤과 달리 *ToggleButton* 속성을 가집니다.
- CLX의 *TColorDialog*에는 설정할 *TColorDialog::Options* 속성이 없습니다. 따라서 색상 선택 다이얼로그 박스의 모양 및 기능을 사용자 정의할 수 없습니다. 또한 Linux에서는 사용 중인 윈도우 관리자에 따라 *TColorDialog*가 modal이 아니거나 크기 조정이 가능할 수 있습니다. Windows의 경우, *TColorDialog*는 항상 modal이고 그 크기를 조정할 수 없습니다.
- 런타임에서 콤보 박스는 CLX와 VCL에서 각각 다르게 동작합니다. VCL과 달리 CLX에서는 콤보 박스의 편집 필드에서 텍스트를 입력하고 *Enter*를 눌러 항목을 드롭다운 리스트에 추가할 수 있습니다. *InsertMode*를 *ciNone*으로 설정하여 이 기능을 선택 해제할 수 있습니다. 콤보 박스의 리스트에 비어 있는(문자열이 아닌) 항목을 추가할 수도 있습니다. 또한 에디트 박스가 닫힐 때 아래쪽 화살표 키를 계속 누르고 있으면 콤보 박스 리스트의 마지막 항목에서 멈추지 않고, 위에서부터 다시 반복합니다.
- *TCustomEdit*는 *Undo*, *ClearUndo* 또는 *CanUndo*를 구현하지 않습니다. 따라서 프로그램에서 편집을 취소할 방법이 없습니다. 그러나 애플리케이션 사용자는 런타임에 에디트 박스를 마우스 오른쪽 버튼으로 클릭하고 Undo 명령을 선택하여 에디트 박스(*TEdit*)의 편집 내용을 취소할 수 있습니다.
- 이벤트에 사용되는 키 값은 VCL과 CLX 간에 달라질 수 있습니다. 예를 들어, Enter 키의 값은 VCL에서 13이고 CLX에서 4100입니다. CLX 애플리케이션에서 키 값을 하드 코딩할 경우, Windows에서 Linux로 이식하거나 그 반대로 이식할 때 이러한 값을 변경해야 합니다.

이외에도 다른 차이점이 존재합니다. 모든 CLX 객체에 대한 자세한 내용이나 {install directory}\C++Builder6\Source\CLX에 소스 코드가 들어 있는 C++Builder 에디션에 대한 자세한 내용은 CLX 온라인 설명서를 참조하십시오.

CLX에서 구현되지 않는 기능

VCL 대신 CLX를 사용할 때는 많은 객체가 동일합니다. 그러나 이러한 객체에 속성, 메소드, 이벤트 등의 일부 기능이 없을 수도 있습니다. 다음과 같은 일반적인 기능이 CLX에서 구현되지 않습니다.

- 오른쪽에서 왼쪽 텍스트 입력이나 출력을 위한 양방향 속성(*BidiMode*)
- 공용 컨트롤의 일반적인 베벨 속성(일부 객체에는 아직도 베벨 속성이 있음)
- 도킹 속성 및 메소드
- Win3.1 탭 및 *Ctl3D* 상의 컴포넌트와 같은 역 호환성 기능
- *DragCursor*와 *DragKind*(끌어서 놓기는 포함되어 있음)

직접 이식되지 않는 기능

C++Builder에서 지원되는 일부 Windows 특정 기능은 Linux 환경으로 직접 전송되지 않습니다. 예를 들어, COM, ActiveX, OLE, BDE 및 ADO는 Windows 기술에 의존하므로 Linux에서 사용할 수 없습니다. 다음 표는 두 플랫폼 간에 다르게 나타나는 기능들과, Linux 또는 CLX에 해당 기능이 있는 경우 그 기능들을 나열한 것입니다.

표 14.3 변경되었거나 다른 기능

Windows/VCL 기능	Linux/CLX 기능
ADO 컴포넌트	일반적인 데이터베이스 컴포넌트
Automation 서버	사용할 수 없음
BDE	dbExpress 및 일반적인 데이터베이스 컴포넌트
COM+ 컴포넌트(ActiveX 포함)	사용할 수 없음
DataSnap	사용할 수 없음
FastNet	사용할 수 없음
레거시 컴포넌트(예: Win 3.1 컴포넌트 팔레트 탭의 항목)	사용할 수 없음
MAPI(Messaging Application Programming Interface)에는 Windows 메시징 기능의 표준 라이브러리가 있음	SMTP 및 POP3로 전자 메일 메시지를 보내고 받고 저장할 수 있음
Quick Report	사용할 수 없음
웹 서비스(SOAP)	사용할 수 없음
WebSnap	사용할 수 없음
Windows API 호출	CLX 메소드, Qt 호출, libc 호출 또는 다른 시스템 라이브러리에 대한 호출
Windows 메시징	Qt 이벤트
Winsock	BSD 소켓

Linux에서 Windows DLL에 해당하는 것은 공유 객체 라이브러리(.so 파일)로서 위치 독립 코드(PIC)를 포함합니다. 따라서 외부 함수에 대한 전역 메모리 참조와 호출은 호출 간에 보존되어야 하는 EBX 레지스터에 상대적으로 만들어집니다.

C++Builder 는 정확한 코드를 생성하므로 어셈블러를 사용할 경우 외부 함수에 대한 전역 메모리 참조와 호출만 신경쓰면 됩니다. 자세한 내용은 14-18페이지의 "인라인 어셈블러 코드 포함"을 참조하십시오.

CLX 및 VCL 유닛 비교

VCL 또는 CLX의 모든 객체는 헤더 파일에 정의됩니다. 예를 들어, 시스템 유닛에는 *TObject*의 구현이 있으며 클래시스 유닛은 기본 *TComponent* 클래스를 정의합니다 . 폼에 객체를 갖다 놓거나 애플리케이션 내에서 객체를 사용하면 해당 유닛의 이름이 소스 파일에 포함된 헤더 파일에 추가되어 컴파일러에게 프로젝트에 어느 유닛을 연결해야 하는지 알려줍니다.

이 단원에는 VCL 유닛과 이에 해당하는 CLX 유닛, CLX 전용 유닛 및 VCL 전용 유닛을 나열한 표 세 개가 있습니다.

다음 표는 VCL 유닛과 이에 해당하는 CLX 유닛을 나열한 것입니다. VCL과 CLX에서 동일한 유닛이나 외부 유닛은 나열하지 않았습니다.

표 14.4 VCL 유닛과 이에 해당하는 CLX 유닛

VCL 유닛	CLX 유닛
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts 및 RTLConsts
Controls	QControls
DBActns	QDBActns
DBCtrls	QDBCtrls
DBGrids	QDBGrids
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Mask	QMask
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdActns	QStdActns
StdCtrls	QStdCtrls
Types	Types와 QTypes
VclEditors	ClxEditors

다음은 CLX에만 있고 VCL에는 없는 유닛입니다.

표 14.5 CLX 전용 유닛

유닛	설명
DirSel	디렉토리 선택
QStyle	GUI 룩앤필(look and feel)
Qt	Qt 라이브러리 인터페이스

다음 Windows VCL 유닛은 ADO, BDE 및 COM 같은 Linux에는 없는 Windows 특정 기능과 관련이 있으므로 대부분 CLX에 포함되어 있지 않습니다.

표 14.6 VCL 전용 유닛

Unit	포함되지 않은 이유
ADOConst	ADO 기능 없음
ADODB	ADO 기능 없음
AppEvnts	TApplicationEvent 객체 없음
AxCtrls	COM 기능 없음
BdeConst	BDE 기능 없음
Calendar	현재 지원되지 않음
Chart	현재 지원되지 않음
CmAdmCtl	COM 기능 없음
ColorGrd	현재 지원되지 않음
ComStrs	COM 기능 없음
ConvUtils	사용할 수 없음
CorbaCon	CORBA 기능 없음
CorbaStd	CORBA 기능 없음
CorbaVCL	CORBA 기능 없음
CtlPanel	Windows 제어판 없음
CustomizeDlg	현재 지원되지 않음
DataBkr	현재 지원되지 않음
DBCGrids	BDE 기능 없음
DBExcept	BDE 기능 없음
DBInpReq	BDE 기능 없음
DBLookup	폐기됨
DbOleCtl	COM 기능 없음
DBPWDlg	BDE 기능 없음
DBTables	BDE 기능 없음
DdeMan	DDE 기능 없음
DRTable	BDE 기능 없음
ExtActns	현재 지원되지 않음
ExtDlgs	그림 다이얼로그 박스 기능 없음
FileCtrl	폐기됨
ListActns	현재 지원되지 않음
MConnect	COM 기능 없음
Messages	Windows 메시징 없음
MidasCon	폐기됨
MPlayer	Windows 미디어 플레이어 없음
Mtsobj	COM 기능 없음
MtsRdm	COM 기능 없음

표 14.6 VCL 전용 유닛

Unit	포함되지 않은 이유
Mtx	COM 기능 없음
mxConsts	COM 기능 없음
ObjBrkr	현재 지원되지 않음
OleConstMay	COM 기능 없음
OleCtnrs	COM 기능 없음
OleCtrls	COM 기능 없음
OLEDB	COM 기능 없음
OleServer	COM 기능 없음
Outline	폐기됨
Registry	Windows 레지스트리 기능 없음
ScktCnst	소켓에 의해 교체됨
ScktComp	소켓에 의해 교체됨
SConnect	지원되는 연결 프로토콜 없음
SHDocVw_ocx	ActiveX 기능 없음
StdConvs	현재 지원되지 않음
SvcMgr	Windows NT 서비스 기능 없음
TabNotbk	폐기됨
Tabs	폐기됨
ToolWin	도킹 기능 없음
ValEdit	현재 지원되지 않음
VarCmplx	현재 지원되지 않음
VarConv	현재 지원되지 않음
VCLCom	COM 기능 없음
WebConst	Windows 상수 없음
Windows	Windows API 호출 없음

CLX 객체 생성자의 차이점

CLX 객체를 암시적으로 폼에 두는 방법으로 만들거나 객체 생성자를 사용하여 명시적으로 코드에서 만들면 원본으로 사용하는 연결된 widget의 인스턴스도 함께생성됩니다. CLX 객체는 이 widget 인스턴스를 소유합니다. CLX 객체가 삭제되면 원본으로 사용하는 widget도 함께 삭제됩니다. 이 기능은 Windows 애플리케이션의 VCL에서 볼 수 있는 것과 동일한 타입의 기능입니다.

QWidget_Create()와 같은 Qt 인터페이스 라이브러리로 호출하여 CLX 객체를 코드에서 명시적으로 만들면 CLX 객체가 소유하지 않는 Qt widget의 인스턴스가 만들어집니다. 이 경우, 기존 Qt widget의 인스턴스가 CLX 객체에 전달되어 객체 생성 동안 사용됩니다. 이 CLX 객체는 자신에게 전달된 Qt widget을 소유하지 않습니다. 따라서 객체를 이 방법으로 만들면 CLX 객체만 삭제되고 원본으로 사용하는 Qt widget 인스턴스는 삭제되지 않습니다. 이것이 VCL과 다른 점입니다.

TBrush 및 *TPen*과 같은 일부 CLX 그래픽 객체를 사용하면 *OwnHandle* 메소드를 통해 원본으로 사용하는 widget의 소유권을 가정할 수 있습니다. *OwnHandle*을 호출한 후 CLX 객체를 삭제하면 원본으로 사용하는 widget도 삭제됩니다.

CLX에서의 일부 속성 할당은 생성자에서 *InitWidget*으로 이동했습니다. 따라서 실제로 필요할 때까지 Qt 객체의 생성을 지연시킬 수 있습니다. 예를 들어, *Color*라는 이름의 속성이 있다고 가정합니다. *SetColor*에서 Qt 핸들이 있는지 보려면 *HandleAllocated*에서 확인할 수 있습니다. 이 핸들이 할당되어 있으면 Qt를 적절히 호출하여 색상을 설정할 수 있습니다. 할당되지 않았으면 값을 private 필드 변수에 저장하고 *InitWidget*에서 속성을 설정합니다.

객체 생성에 대한 자세한 내용은 13-7페이지의 "C++Builder VCL/CLX 클래스의 객체 생성"을 참조하십시오.

시스템 및 widget 이벤트 처리

컴포넌트를 작성할 때 많이 다루어지는 시스템 및 widget 이벤트는 VCL과 CLX에서 다르게 처리됩니다. 가장 중요한 차이점은 CLX 컨트롤은 Windows에서 실행 중인 경우에도 Windows 메시징에 직접 응답하지 않는다는 것입니다(51장, "메시지 및 시스템 통지 처리" 참조). 대신 CLX 컨트롤은 원본으로 사용하는 widget 레이어의 통지에 응답합니다. 이 통지가 다른 시스템을 사용하기 때문에 이벤트의 순서와 시간은 해당 CLX 및 VCL 객체 간에 달라질 수 있습니다. 이 차이점은 CLX 애플리케이션이 Linux가 아니라 Windows에서 실행 중인 경우에도 발생합니다. VCL 애플리케이션을 CLX로 이식하는 중이면 이러한 차이점을 수용하도록 이벤트 핸들러의 응답 방법을 변경해야 할 수 있습니다.

CLX 컴포넌트의 published 이벤트에서 반영되는 컴포넌트가 아니라 시스템 및 widget 이벤트에 응답하는 컴포넌트를 작성하는 것에 대한 자세한 내용은 51-10페이지의 "CLX를 사용하여 시스템 통지에 응답"을 참조하십시오.

Windows와 Linux 간의 소스 파일 공유

애플리케이션이 Windows와 Linux 모두에서 실행되게 하려면 두 운영 체제에서 액세스할 수 있도록 소스 파일을 공유하면 됩니다. 두 컴퓨터에서 액세스할 수 있는 서버에 소스 파일을 저장하거나 Linux 컴퓨터의 Samba를 사용하여 Linux와 Windows용 Microsoft 네트워크 파일을 공유함으로써 파일에 대한 액세스를 제공하는 등 소스 파일을 공유하는 방법은 여러 가지가 있습니다. Linux에서 소스를 보관하도록 하고 Linux에 공유 드라이브를 만들 수 있습니다. 또는 Windows에 소스를 보관하고 공유를 만들어 Linux 컴퓨터가 액세스할 수 있게 할 수 있습니다.

VCL과 CLX 모두에서 지원하는 객체를 사용하면 C++Builder에서 파일을 계속적으로 개발하고 컴파일할 수 있습니다. 이 작업이 끝나면 Windows에서 컴파일할 수 있습니다. Linux C++ 솔루션을 사용할 수 있게 되면 Linux에서도 컴파일할 수 있습니다.

새 CLX 애플리케이션을 만들 경우, C++Builder 에서는 .dfm 파일 대신 .xfm 폼 파일을 만듭니다.

Windows와 Linux 간의 환경적 차이점

현재 크로스 플랫폼은 Windows 및 Linux 운영 체제 모두에서 실제로 변경하지 않고 컴파일할 수 있는 애플리케이션을 의미합니다. 다음 표는 Linux 및 Windows 운영 환경 간의 몇 가지 차이점을 나열한 것입니다.

표 14.7 Linux 및 Windows 운영 환경 간의 차이점

차이점	설명
파일 이름 대소문자 구별	Linux에서는 파일 이름의 대소문자를 구별합니다. Test.txt 파일은 test.txt와 *다릅니다*. Linux에서는 파일 이름의 대소문자에 많은 주의를 기울여야 합니다.
줄 끝 문자	Windows에서 텍스트 행은 CR/LF(즉 ASCII 13 + ASCII 10)로 종료되지만 Linux에서는 LF로 종료됩니다. 코드 에디터가 이 차이점을 처리하기는 하지만 Windows에서 코드를 임포트할 때 이러한 차이점을 인식하고 있어야 합니다.
파일 끝 문자	MS-DOS와 Windows에서 #26(Ctrl-Z) 문자 값은 해당 문자 뒤에 데이터가 있더라도 텍스트 파일의 끝으로 취급됩니다. Linux에서는 Ctrl+D가 파일 끝 문자로 사용됩니다.
배치 파일/셸 스크립트	Linux에서 .bat 파일에 해당하는 것은 셸 스크립트입니다. 스크립트는 명령어를 포함하는 텍스트 파일로, 저장되어 `chmod +x scriptfile>` 명령으로 실행 가능하게 됩니다. 스크립트 언어는 Linux에서 사용하는 셸에 따라 다릅니다. Bash를 일반적으로 사용합니다.
명령 확인	MS-DOS나 Windows에서는 파일이나 폴더를 삭제하려는 경우 "정말로 삭제하시겠습니까?"라고 확인을 합니다. Linux는 일반적으로 묻지 않고 바로 실행합니다. 그러면 실수로 파일이나 전체 파일 시스템을 삭제해버릴 수 있습니다. 파일을 다른 매체에 백업하지 않는 한 Linux에서는 삭제를 취소할 방법이 없습니다.
명령 피드백	Linux에서는 명령이 성공하면 상태 메시지 없이 명령 프롬프트를 다시 표시합니다.
명령 스위치	DOS에서 슬래시(/)나 대시(-)를 사용하는 대신 Linux에서는 대시(-)를 사용하여 명령 스위치를 표시하거나 이중 대시(--)를 사용하여 여러 문자 옵션을 표시합니다.
설정 파일	Windows에서 구성은 레지스트리나 autoexec.bat과 같은 파일에서 이루어집니다. Linux에서 설정 파일은 사용자의 홈 디렉토리에 숨김 파일로 만들어집니다. /etc 디렉토리에 있는 설정 파일은 대개 히든 파일이 아닙니다. Linux도 LD_LIBRARY_PATH(라이브러리의 경로를 찾음)와 같은 환경 변수를 사용합니다. 기타 중요한 환경 변수는 다음과 같습니다. HOME 사용자의 홈 디렉토리(/home/sam) TERM 터미널 타입(xterm, vt100, console) SHELL 사용자 셸의 경로(/bin/bash) USER 사용자의 로그인 이름(sfuller) PATH 프로그램을 검색하는 경로 이 환경 변수들은 셸이나 .bashrc 같은 파일에 지정되어 있습니다.
DLL	Linux에서는 공유 객체 파일(.so)을 사용합니다. Windows에서 이러한 파일들은 동적 연결 라이브러리(DLL)입니다.

표 14.7 Linux 및 Windows 운영 환경 간의 차이점(계속)

차이점	설명
드라이브 문자	Linux에는 드라이브 문자가 없습니다. Linux의 경로명을 예를 들면 /lib/security입니다. 런타임 라이브러리에서 DriveDelim을 참조하십시오.
예외	운영 체제 예외는 Linux에서 시그널이라고 합니다.
실행 파일	Linux의 실행 파일은 확장자가 없습니다. Windows의 실행 파일은 exe 확장자를 갖습니다.
파일 이름 확장자	Linux는 파일 이름 확장자를 사용하여 파일 형식을 식별하거나 애플리케이션과 연결하지 않습니다.
파일 권한	Linux에서는 파일(및 디렉토리)에 파일 소유자, 그룹 및 기타에 대해 읽고, 쓰고 실행할 권한이 할당됩니다. 예를 들면, 다음과 같습니다. `-rwxr-xr-x`는 왼쪽에서 오른쪽으로 다음과 같은 의미입니다. `-`은 파일 타입(`-` = 일반 파일, `d` = 디렉토리, `l` = 링크)이고, `rwx`는 파일 소유자의 권한(읽기, 쓰기, 실행)이며 `r-x`는 파일 소유자 그룹의 권한(읽기, 실행)이고 `r-x`는 다른 모든 사용자의 권한(읽기, 실행)입니다. 루트 사용자(superuser)는 이 권한을 무시할 수 있습니다. 애플리케이션이 올바른 사용자에게서 실행되고 필요한 파일에 대한 적절한 액세스 권한이 있는지 확인해야 합니다.
Make 유틸리티	Borland의 make 유틸리티는 Linux 플랫폼에서는 사용할 수 없습니다. 그 대신 Linux의 GNU make 유틸리티를 사용하면 됩니다.
멀티태스킹	Linux는 멀티태스킹을 완벽하게 지원합니다. 동시에 여러 프로그램(Linux에서는 프로세스라고 부름)을 실행할 수 있습니다. 명령 뒤에 &를 사용하면 프로세스를 백그라운드로 실행하고 또 그대로 계속하여 작업할 수도 있습니다. Linux에서는 또한 여러 세션을 지원합니다.
경로명	DOS에서 역슬래시(\)를 사용하는 곳에 Linux는 슬래시(/)를 사용합니다. PathDelim 상수를 사용하면 플랫폼의 해당 문자를 지정할 수 있습니다. 런타임 라이브러리에서 PathDelim을 참조하십시오.
검색 경로	프로그램 실행 시 Windows는 항상 현재 디렉토리를 먼저 찾은 다음 PATH 환경 변수를 찾습니다. Linux는 현재 디렉토리를 찾지 않고 PATH에 나열된 디렉토리만을 검색합니다. 현재 디렉토리에서 프로그램을 실행하려면 그 앞에 ./를 써주어야 합니다. PATH를 수정하여 검색할 첫 경로로서 ./를 포함할 수도 있습니다.
검색 경로 구분자	Windows에서는 검색 경로 구분자로 세미콜론을 사용합니다. Linux에서는 콜론을 사용합니다. 런타임 라이브러리에서 PathDelim을 참조하십시오.
심볼릭 링크	Linux에서 심볼릭 링크는 디스크의 다른 파일을 가리키는 특별한 파일입니다. 애플리케이션의 주요 파일을 가리키는 전역 bin 디렉토리에 심볼릭 링크를 넣으면 시스템 검색 경로를 수정할 필요가 없습니다. 심볼릭 링크는 ln(link) 명령으로 만듭니다. Windows에는 GUI 데스크탑의 단축키가 들어 있습니다. 명령줄에서 프로그램을 사용하도록 하기 위해 Windows 설치 프로그램은 일반적으로 시스템 검색 경로를 수정합니다.

Linux의 디렉토리 구조

Linux에서는 디렉토리가 다릅니다. 파일 시스템 상의 어느 곳에서든지 모든 파일 또는 장치를 마운트할 수 있습니다.

참고 Windows 경로명이 역슬래시를 사용하는 반면 Linux 경로명은 슬래시를 사용합니다. 첫 번째 슬래시는 루트 디렉토리를 의미합니다.

다음은 Linux에서 일반적으로 사용되는 디렉토리입니다.

표 14.8 공통된 Linux 디렉토리

디렉토리	내용
/	전체 Linux 파일 시스템의 루트 또는 최상위 디렉토리
/root	루트 파일 시스템인 superuser의 홈 디렉토리
/bin	명령, 유틸리티
/sbin	시스템 유틸리티
/dev	파일로 보여지는 장치
/lib	라이브러리
/home/username	사용자의 로그인 이름이 username인 곳에서 사용자가 소유한 파일
/opt	옵션
/boot	시스템 시동 시 호출되는 커널
/etc	설정 파일
/usr	애플리케이션, 프로그램. 보통 /usr/spool, /usr/man, /usr/include, /usr/local 같은 디렉토리 포함
/mnt	CD나 플로피 디스크 드라이브 같은 시스템에 마운트되는 기타 매체
var	로그, 메시지, 스풀 파일
/proc	가상 파일 시스템 및 시스템 통계 보고
/tmp	임시 파일

참고 Linux는 배포판에 따라 종종 다른 위치에 파일을 둡니다. Red Hat 배포판에서는 /bin에 유틸리티 프로그램이 있지만 Debian 배포판에서는 /usr/local/bin에 있습니다.

UNIX/Linux의 계층적 파일 시스템 구성에 대한 자세한 내용 및 *Filesystem Hierarchy Standard*를 읽으려면 www.pathname.com을 참조하십시오.

이식 가능한 코드 작성

Windows 및 Linux에서 실행 가능한 크로스 플랫폼 애플리케이션을 작성하는 경우 다른 조건에서 컴파일하는 코드를 작성할 수 있습니다. 조건부 컴파일을 사용하면 Windows 코드를 유지하면서도 Linux 운영 체제와의 차이점을 수용하도록 할 수 있습니다.

Windows와 Linux 간에 쉽게 이식할 수 있는 애플리케이션을 만들려면 반드시 다음 사항을 염두에 두어야 합니다.

- 플랫폼 특정(Win32 또는 Linux) API에 대한 호출을 줄이거나 분리합니다. 즉, Qt 라이브러리에 대한 CLX 메소드 또는 호출을 사용합니다.
- 애플리케이션 내에서 구성되는 Windows 메시징(PostMessage, SendMessage)을 제거합니다. 대신 CLX에서 *QApplication_postEvent* 및 *QApplication_sendEvent* 메소드를 호출합니다.

시스템 및 widget 이벤트에 응답하는 컴포넌트를 작성하는 것에 대한 자세한 내용은 51-10 페이지의 "CLX를 사용하여 시스템 통지에 응답"을 참조하십시오.

- *TRegIniFile* 대신 *TMemIniFile*을 사용합니다.
- 파일과 디렉토리 이름의 대소문자 구분에 유의합니다.
- 모든 외부 어셈블러 TASM 코드를 이식합니다. GNU 어셈블러 "as"는 TASM 구문을 지원하지 않습니다(14-18페이지의 "인라인 어셈블러 코드 포함" 참조).

플랫폼 독립적인 런타임 라이브러리 루틴을 사용하고 시스템, SysUtils 및 기타 런타임 라이브러리 유닛의 상수를 사용하는 코드를 작성하도록 하십시오. 예를 들면, PathDelim 상수를 사용하여 '/'와 '\' 플랫폼 차이로 코드를 구분합니다.

두 플랫폼에서 멀티바이트 문자를 사용하는 것이 또 다른 방법입니다. Windows 코드는 오래 전부터 멀티바이트 문자당 2바이트를 사용해왔습니다. Linux에서 멀티바이트 문자 인코딩은 문자당 더 많은 바이트(UTF-8에 대해 최고 6바이트까지)를 가질 수 있습니다. 두 플랫폼 모두 SysUtils의 StrNextChar 함수를 사용하여 조정할 수 있습니다. 다음과 같은 기존의 Windows 코드가 있다고 가정해 보십시오.

```
while(*p != 0)
{
   if(LeadBytes.Contains(*p))
      p++;
   p++;
}
```

위의 코드를 플랫폼 독립적인 코드로 대체할 수 있습니다.

```
while(*p != 0)
{
   if(LeadBytes.Contains(*p))
     p = StrNextchar(p);
   else
     p++;
}
```

이 예제는 다른 플랫폼으로 이식 가능하면서도 멀티바이트가 아닌 로케일에 대한 프로시저 호출로 인해 성능이 저하되는 문제를 여전히 방지할 수 있습니다.

런타임 라이브러리 함수를 사용해도 해결되지 않으면 루틴의 플랫폼 특정 코드를 청크 하나로 넣어 분리하거나 서브루틴으로 넣어 분리합니다. 소스 코드의 가독성과 이식성을 유지하려면 **#ifdef** 블록의 수를 제한하도록 합니다. 조건 심볼 WIN32는 Linux에는 정의되어 있지 않습니다. 조건 심볼 LINUX는 소스 코드가 Linux 플랫폼용으로 컴파일되고 있는 것으로 정의되어 있습니다.

조건 지시어 사용

#ifdef 컴파일러 지시어를 사용하는 것은 Windows 및 Linux 플랫폼용으로 코드를 조건화하기에 적합한 방법입니다. 그러나 **#ifdef**로는 소스 코드를 이해하고 유지하기가 어렵기 때문에 **#ifdef**를 언제 사용하는 것이 적당한지를 이해해야 합니다. **#ifdef** 사용을 고려할 때 가장 중요한 문제는 "이 코드에 왜 **#ifdef**가 필요한가?"와 "**#ifdef**를 쓰지 않고도 이 코드를 작성할 수 있을까?"입니다.

크로스 플랫폼 애플리케이션 내에서 **#ifdef**를 사용하려면 다음 지침을 따릅니다.

- **#ifdef**는 반드시 필요한 경우가 아니면 사용하지 않도록 합니다. 소스 파일에 있는 **#ifdef**는 소스 코드가 컴파일될 때에만 분석됩니다. 프로젝트를 컴파일하기 위해서는 C/C++에 유닛 소스(헤더 파일)가 필요합니다. 모든 소스 코드를 완전히 다시 컴파일하여 생성하는 것은 대부분의 C++Builder 프로젝트에서 일반적인 작업이 아닙니다.
- 패키지(.bpk) 파일에서 **#ifdef**를 사용해서는 안됩니다. 대신 소스 파일에만 사용해야 합니다. 컴포넌트 작성자는 크로스 플랫폼 개발 시 **#ifdef**를 이용하여 하나의 패키지를 만드는 것이 아니라 두 개의 디자인 타임 패키지를 만들어야 합니다.
- 일반적으로는 **#ifdef** WINDOWS를 사용하여 WIN32를 포함한 Windows 플랫폼에 대해 테스트합니다. **#ifdef** WIN32는 32비트와 64비트 Windows 같은 특정 Windows 플랫폼 간의 구분을 위해 사용합니다. WIN64에서 사용하지 않는 것이 확실하지 않다면 코드를 WIN32로 국한시키지 마십시오.
- **#ifndef**와 같은 부정 테스트는 반드시 필요한 경우 외에는 하지 않습니다. **#ifndef** __linux__는 **#ifdef** WINDOWS와 동등하지 *않습니다*.
- **#ifndef/#else** 조합을 사용하지 않습니다. 그 대신 가독성을 높이려면 긍정 테스트(**#ifdef**)를 사용합니다.
- 플랫폼을 구분하는 **#ifdef**에서 **#else** 절을 사용하지 않습니다. **#ifdef** __linux__/**#else** 또는 **#ifdef** WINDOWS/**#else** 대신 Linux 및 Windows 전용 코드에 각각의 **#ifdef** 블록을 사용합니다.

  예를 들어, 이전 코드에는 다음과 같은 내용이 있을 수 있습니다.

```
#ifdef WIN32
   (32-bit Windows code)
#else
   (16-bit Windows code)   //!! By mistake, Linux could fall into this
code.
#endif
```

  #ifdef 의 모든 이식 불가 코드의 경우 플랫폼이 **#else** 절에 의해 런타임 시 알 수 없는 이유로 실패하는 것 보다는 소스 코드가 컴파일되지 않는 것이 낫습니다. 컴파일 실패가 런타임 실패보다 찾기가 더 쉽습니다.

- 복잡한 테스트에는 **#if** 구문을 사용합니다. 중첩된 **#ifdef**를 **#if** 지시어의 부울 표현식으로 바꿉니다. **#if** 지시어를 **#endif**를 사용하여 종료해야 합니다. 이렇게 하면 이전 컴파일러로부터 새로운 **#if** 구문을 숨기기 위해 **#if** 표현식을 **#ifdef** 안에 둘 수 있습니다.

모든 조건 지시어가 온라인 도움말에 있습니다. 자세한 내용은 도움말의 "Conditional Compilation" 항목을 참조하십시오.

메시지 표시

#pragma message 컴파일러 지시어를 사용하면 컴파일러와 마찬가지로 소스 코드에서 경고와 오류를 표시할 수 있습니다. #pragma message를 이용하여 사용자 정의 메시지를 다음 형식 중 하나로 프로그램 코드 내에서 지정합니다.

문자열 상수 수가 가변적인 경우에는 다음을 사용합니다.

```
#pragma message( “hi there” )
#pragma message( “hi” “ there” )
```

메시지 다음에 오는 텍스트를 작성하려면 다음을 사용하십시오.

```
#pragma message text
```

이전에 정의한 값을 확장하려면 다음을 사용하십시오.

```
#pragma message (text)
  #define text “a test string”
  #pragma message (text)
```

예를 들어, 다음과 같은 방법으로 이러한 메시지를 버튼에 표시합니다.

```
  void __fastcall TForm1::Button1Click(TObject *Sender)
  {
      #pragma message( “hi there 1” )
      #pragma message( “hi “ “there 2” )

      #pragma message hi there 3

      #define text “a test string”
      #pragma message (text)
  }
```

IDF에서 메시지를 표시하려면 Projects|Options|Compiler를 선택하고 Compiler 탭을 클릭한 다음 Show General Messages 체크 박스를 선택하십시오.

인라인 어셈블러 코드 포함

Windows 애플리케이션에 인라인 어셈블러 코드를 포함하면 Linux의 위치 독립 코드(PIC) 요구 사항 때문에 해당 코드를 Linux에서 사용하지 못할 수도 있습니다. Linux 공유 객체 라이브러리(DLL에 해당됨)는 모든 코드를 수정하지 않고 메모리에 재배치할 수 있어야 합니다. 이는 전역 변수 또는 기타 절대 주소를 사용하는 루틴이나 외부 함수를 호출하는 인라인 어셈블러 루틴에 1차적인 영향을 줍니다.

C++ 코드만을 포함하는 유닛의 경우 컴파일러는 필요할 때 자동으로 PIC를 생성합니다. 모든 소스 파일은 PIC와 PIC가 아닌 두 가지 형식으로 컴파일하는 것이 좋습니다. 즉, -VP 컴파일러 스위치를 사용하여 PIC를 생성합니다.

실행 파일이나 공유 라이브러리 중 어느 것으로 컴파일하는지에 따라 어셈블러 루틴을 다르게 코딩하려는 경우 **#ifdef**__PIC__를 사용하여 어셈블러 코드를 두 버전으로 나눕니다. 또는 이 문제를 피하기 위해 C++에서 루틴을 다시 작성할 수도 있습니다.

다음은 인라인 어셈블러 코드에 대한 PIC 규칙입니다.

- Linux에서 전역 오프셋 테이블 또는 GOT로 불리는 현재 모듈의 기본 주소 포인터가 EBX 레지스터에 들어 있으므로, PIC에서는 모든 메모리 참조가 이 레지스터와 관련되어야 합니다. 따라서 다음과 같이 하지 않는 대신

  ```
  MOV EAX,GlobalVar
  ```

  다음을 사용합니다

  ```
  MOV EAX,[EBX].GlobalVar
  ```

- PIC에서는 Win32에서와 같이 호출 간에 EBX 레지스터를 어셈블리 코드에 남겨두어야 하고 또 Win32에서와 달리 외부 함수를 호출하기 *전에* EBX 레지스터를 복원해야 합니다.

- PIC 코드는 기본 실행 파일에서 작동되기는 하지만 성능이 떨어지고 더 많은 코드를 생성합니다. 공유 객체에서는 선택의 여지가 없지만 실행 파일에서는 가장 높은 수준의 성능을 바랄 것입니다.

Linux에서의 프로그래밍 차이점

Linux wchar_t widechar는 문자당 32비트입니다. VCL 및 CLX가 지원하는 16비트 유니코드 표준은 Linux와 GNU 라이브러리가 지원하는 32비트 UCS 표준의 부분 집합입니다. WideString 타입은 wchar_t로 OS 함수에 전달되기 전에 문자당 32비트로 확장되어야 합니다.

Linux에서 WideString은 긴 문자열처럼 카운트되는 참조입니다(Windows에서는 아님).

Windows에서 멀티바이트 문자(MBCS)는 1바이트와 2바이트 char 코드로 표시됩니다. Linux에서는 1에서 6바이트로 표시됩니다.

AnsiString는 멀티바이트 문자 구조를 지닐 수 있지만 사용자의 로케일 설정에 따릅니다. 일본어, 중국어, 히브리어 및 아라비아어와 같은 멀티바이트 문자의 Linux 인코딩은 동일한 로케일에 대한 Windows 인코딩과 호환되지 않습니다. 멀티바이트는 이식할 수 없지만 유니코드는 이식이 가능합니다. 국제적인 애플리케이션에서 다양한 로케일의 문자열을 처리하는 것이 대한 자세한 내용은 16-2페이지의 "애플리케이션 코드 활성화"를 참조하십시오.

크로스 플랫폼 데이터베이스 애플리케이션

Windows에서 C++Builder는 데이터베이스 정보 액세스 방법에 대한 여러 선택 사항을 제공합니다. ADO, BDE(Borland Database Engine) 및 InterBase Express 사용이 들어 있습니다. Windows와 Linux에서는 현재 보유한 C++Builder의 에디션에 따라 크로스 플랫폼 데이터 액세스 기술인 *dbExpress*를 사용할 수 있습니다.

*dbExpress*가 Linux에서도 실행되도록 데이터베이스 애플리케이션을 이식하기 전에 *dbExpress*의 사용 방법과 사용자가 이전에 사용하던 데이터 액세스 메커니즘 사용 방법 사이의 차이점을 이해해야 합니다. 그 차이는 여러 수준에서 발생합니다.

- 가장 낮은 수준에는 애플리케이션과 데이터베이스 서버 간에 통신하는 레이어가 있습니다. 이 레이어는 ADO, BDE 또는 InterBase 클라이언트 소프트웨어일 수 있습니다. 이 레이어는 동적 SQL 처리를 위한 lightweight 드라이버 집합인 *dbExpress*로 대체됩니다.
- 낮은 레벨의 데이터 액세스는 데이터 모듈이나 폼에 추가하는 컴포넌트 집합에 랩핑됩니다. 이러한 컴포넌트들은 데이터베이스 서버로의 연결을 나타내는 데이터베이스 연결 컴포넌트 및 서버에서 가져온 데이터를 나타내는 데이터셋을 포함합니다. *dbExpress* 커서의 단방향성 때문에 중요한 차이점이 있기는 하지만, 데이터베이스 연결 컴포넌트처럼 모든 데이터셋은 공통 조상을 공유하기 때문에 이 레벨에서는 그 차이가 분명하게 드러나지 않습니다.

- 사용자 인터페이스 수준에는 차이점이 거의 없습니다. CLX 데이터 인식 컨트롤은 해당 Windows 컨트롤에 가능한 유사하게 디자인되었습니다. 사용자 인터페이스 수준에서의 주요 차이점은 캐싱된 업데이트의 사용을 적용하기 위해 변경이 필요할 때 나타납니다.

기존 데이터베이스 애플리케이션을 *dbExpress*로 이식하는 것에 대한 자세한 내용은 14-22페이지의 "데이터베이스 애플리케이션을 Linux로 이식"을 참조하십시오. 새 *dbExpress* 애플리케이션을 디자인하는 것에 대한 자세한 내용은 18장, "데이터베이스 애플리케이션 디자인"을 참조하십시오.

dbExpress 차이점

Linux에서 *dbExpress*는 데이터베이스 서비와의 통신을 관리합니다. *dbExpress*는 공통 인터페이스 집합을 구현하는 lightweight 드라이버 집합으로 구성됩니다. 각 드라이버는 애플리케이션에 연결해야 하는 공유 객체(.so 파일)입니다. *dbExpress*는 크로스 플랫폼용으로 디자인되었으므로 Windows에서도 동적 연결 라이브러리(.dll) 집합으로 사용할 수 있습니다.

모든 데이터 액세스 레이어와 마찬가지로 *dbExpress*는 데이터베이스 업체가 제공하는 클라이언트사이드 소프트웨어를 필요로 합니다. 또한 데이터베이스별 드라이버와 두 개의 설정 파일인 dbxconnection과 dbxdriver를 사용합니다. 이것은 기본 Borland Database Engine 라이브러리(Idapi32.dll) 외에도 데이터베이스별 드라이버와 많은 수의 기타 지원 라이브러리를 필요로 하는 BDE 등의 경우보다 요구 사항이 훨씬 적은 것입니다.

또한 *dbExpress*는 다음과 같은 특징을 갖고 있습니다.

- 원격 데이터베이스를 보다 간단하고 신속하게 연결할 수 있게 해줍니다. 그 결과 데이터를 한층 더 신속하고 간단하게 액세스할 수 있게 됩니다.
- 쿼리와 내장 프로시저를 처리할 수 있지만 개방형 테이블의 개념은 지원하지 않습니다.
- 단방향 커서만 반환합니다.
- INSERT, DELETE 또는 UPDATE 쿼리를 실행하는 기능 말고는 기본 업데이트 지원이 없습니다.
- 메타데이터 캐싱을 하지 않으며 디자인 타임 메타데이터 액세스 인터페이스는 코어 데이터 액세스 인터페이스를 사용하여 구현됩니다.
- 사용자가 요청한 쿼리만을 실행하므로 다른 쿼리를 끌어들이지 않고 데이터베이스 액세스를 최적화합니다.
- 레코드 버퍼나 레코드 버퍼 블록을 내부적으로 관리합니다. 이것이 레코드 버퍼에 사용되는 메모리를 클라이언트가 할당해야 하는 BDE와 다른 점입니다.
- *I*nterBase 및 Oracle과 같은 SQL 기반의 로컬 테이블만 지원합니다.
- DB2, Informix, InterBase, MySQL 및 Oracle용 드라이버를 사용합니다. 다른 데이터베이스 서버를 사용하고 있을 경우에는 이 데이터베이스들 중 하나로 데이터를 변환하고 사용 중인 데이터베이스 서버용 *dbExpress* 드라이버를 작성하거나 사용자의 데이터베이스 서버용으로 사용할 외부 *dbExpress* 드라이버를 구해야 합니다.

컴포넌트 레벨 차이점

dbExpress 애플리케이션을 작성할 때는 기존 데이터베이스 애플리케이션에 사용된 것과 다른 데이터 액세스 컴포넌트 집합이 필요합니다. *dbExpress* 컴포넌트는 다른 데이터 액세스 컴포넌트(*TDataSet* 및 *TCustomConnection*)와 동일한 기본 클래스를 공유하며, 이것은 많은 속성, 메소드 및 이벤트가 기존 애플리케이션에 사용되는 컴포넌트와 동일하다는 것을 의미합니다.

표 14.9는 Windows 환경의 InterBase Express, BDE 및 ADO에서 사용되는 중요한 데이터베이스 컴포넌트를 나열하며 Linux와 크로스 플랫폼 애플리케이션에서 사용되는 유사한 *dbExpress* 컴포넌트를 보여 줍니다.

표 14.9 유사한 데이터 액세스 컴포넌트

InterBase Express 컴포넌트	BDE 컴포넌트	ADO 컴포넌트	dbExpress 컴포넌트
TIBDatabase	*TDatabase*	*TADOConnection*	*TSQLConnection*
TIBTable	*TTable*	*TADOTable*	*TSQLTable*
TIBQuery	*TQuery*	*TADOQuery*	*TSQLQuery*
TIBStoredProc	*TStoredProc*	*TADOStoredProc*	*TSQLStoredProc*
TIBDataSet		*TADODataSet*	*TSQLDataSet*

dbExpress 데이터셋(*TSQLTable*, *TSQLQuery*, *TSQLStoredProc* 및 *TSQLDataSet*)은 편집을 지원하지 않고 포워드 탐색만 허용하기 때문에 다른 데이터셋보다 더 제한적입니다. *dbExpress* 데이터셋과 Windows에서 사용할 수 있는 다른 데이터셋 간의 차이점에 대한 자세한 내용은 26장, "단방향 데이터셋 사용"을 참조하십시오.

편집 및 탐색 지원이 없으므로 대부분의 *dbExpress* 애플리케이션은 *dbExpress* 데이터셋과 직접 연동하지 않습니다. 오히려 *dbExpress* 데이터셋을 메모리에 레코드를 버퍼링하고 편집과 탐색 지원을 제공하는 클라이언트 데이터셋에 연결합니다. 이 아키텍처에 대한 자세한 내용은 18-5페이지의 "데이터베이스 아키텍처"를 참조하십시오.

참고 아주 간단한 애플리케이션의 경우 클라이언트 데이터셋에 연결된 *dbExpress* 데이터셋 대신 *TSQLClientDataSet*을 사용할 수 있습니다. 이는 이식하려는 애플리케이션의 데이터셋과 이식된 애플리케이션의 데이터셋 간에 일대일로 대응되기 때문에 간단하다는 이점이 있지만 *dbExpress* 데이터셋을 클라이언트 데이터셋에 명시적으로 연결하는 것보다는 유연하지 못합니다. 대부분의 애플리케이션에서는 *TClientDataSet* 컴포넌트에 연결된 *dbExpress* 데이터셋을 사용하는 것이 좋습니다.

사용자 인터페이스 레벨 차이점

CLX 데이터 인식 컨트롤은 해당 Windows 컨트롤과 최대한 유사하게 디자인되어 있습니다. 그 결과 데이터베이스 애플리케이션의 사용자 인터페이스 부분을 이식하면 Windows 애플리케이션을 CLX로 이식하는 것 외에 추가로 고려할 사항이 거의 없습니다.

사용자 인터페이스 수준의 주요 차이점은 *dbExpress* 데이터셋이나 클라이언트 데이터셋이 데이터를 제공하는 방법에 있습니다.

dbExpress 데이터셋만을 사용하는 경우 편집을 지원하지 않고 포워드 탐색만을 지원한다는 사실에 맞추어 사용자 인터페이스를 조정해야 합니다. 따라서 사용자를 이전 레코드로 이동하도록 허용하는 컨트롤을 제거해야 할 수 있습니다. *dbExpress* 데이터셋은 데이터를 버퍼링하지 않으므로 데이터 인식 그리드에 데이터를 표시할 수 없고,한 번에 레코드 하나만 표시할 수 있습니다.

dbExpress 데이터셋을 클라이언트 데이터셋에 연결했을 경우 편집 및 탐색과 관련된 사용자 인터페이스 요소는 계속 작동합니다. 클라이언트 데이터셋에 다시 연결하기만 하면 됩니다. 이 경우에 제일 먼저 고려해야 할 것은 데이터베이스에 업데이트를 쓰는 방법을 처리하는 것입니다. 디폴트로, Windows의 대부분의 데이터셋은 포스트될 때(예를 들어 사용자가 새 레코드로 이동할 때) 자동으로 데이터베이스 서버에 업데이트를 씁니다. 반면 클라이언트 데이터셋은 항상 메모리에 업데이트를 캐싱합니다. 이 차이점을 수용하는 방법에 대한 자세한 내용은 14-24페이지의 "dbExpress 애플리케이션에서 데이터 업데이트"를 참조하십시오.

데이터베이스 애플리케이션을 Linux로 이식

데이터베이스 애플리케이션을 *dbExpress*로 이식하면 Windows와 Linux 모두에서 실행되는 크로스 플랫폼 애플리케이션을 만들 수 있습니다. 이식 기법이 다르기 때문에 이식 과정에는 애플리케이션을 변경하는 과정이 포함됩니다. 이식이 얼마나 어려운지는 애플리케이션의 타입, 복잡한 정도 및 필요 조건 등에 따라 다릅니다. ADO 같은 Windows 특정 기술을 많이 사용하는 애플리케이션은 C++Builder 데이터베이스 기술을 사용하는 애플리케이션을 이식하는 것보다 더 어렵습니다.

Windows/VCL 데이터베이스 애플리케이션을 Linux/CLX 로 이식하려면 다음과 같은 일반적인 단계를 따르십시오.

1. DB2, Informix, InterBase, MySQL, Oracle 등과 같이 *dbExpress*가 지원하는 데이터베이스에 데이터가 저장되어 있는지 확인합니다. 데이터는 이러한 SQL 서버 중 하나에 두어야 합니다. 이러한 데이터베이스 중 하나에 아직 데이터를 저장하지 않았다면 유틸리티를 찾아 전송을 수행합니다.

   예를 들어, C++Builder의 Data Pump 유틸리티(일부 에디션에서는 제공하지 않음)를 사용하여 dBase, FoxPro, Paradox 등의 데이터베이스를 dbExpress 지원 데이터베이스로 변환합니다. 이 유틸리티의 사용 방법에 대한 자세한 내용은 Program Files\Common Files\Borland\Shared\BDE에 있는 datapump.hlp 파일을 참조하십시오.

2. 사용자 인터페이스 폼과 컴포넌트에서 분리되도록 데이터셋 및 연결 컴포넌트를 포함하는 데이터 모듈을 만듭니다. 이렇게 하면 데이터 모듈에 완전히 새로운 컴포넌트 집합을 필요로 하는 애플리케이션의 부분을 분리할 수 있습니다. 그런 다음 사용자 인터페이스를 나타내는 폼을 다른 애플리케이션과 마찬가지로 이식할 수 있습니다. 자세한 내용은 14-3페이지의 "애플리케이션 이식"을 참조하십시오.

   나머지 단계에서는 데이터셋과 연결 컴포넌트가 각각의 데이터 모듈에서 분리되어 있는 것으로 가정합니다.

3. 데이터셋의 CLX 버전과 연결 컴포넌트를 보유할 새로운 데이터 모듈을 만듭니다.

4 원본 애플리케이션의 각 데이터셋의 경우 *dbExpress* 데이터셋, *TDataSetProvider* 컴포넌트 및 *TClientDataSet* 컴포넌트를 추가합니다. 표 14.9의 내용을 참조하여 사용할 *dbExpress* 데이터셋을 결정합니다. 컴포넌트에 이름을 부여합니다.

- *TClientDataSet* 컴포넌트의 *ProviderName* 속성을 *TDataSetProvider* 컴포넌트의 이름으로 설정합니다.
- *TDataSetProvider* 컴포넌트의 *DataSet* 속성을 *dbExpress* 데이터셋으로 설정합니다.
- 원본 데이터셋으로 참조되는 데이터 소스 컴포넌트의 *DataSet* 속성을 변경하여 클라이언트 데이터셋을 바로 참조할 수 있게 합니다.

5 새 데이터셋이 원본 데이터셋과 일치하도록 속성을 설정합니다.

- 원본 데이터셋이 *TTable*, *TADOTable* 또는 *TIBTable* 컴포넌트였을 경우 새 *TSQLTable*의 *TableName* 속성을 원본 데이터셋의 *TableName*으로 설정합니다. 또한 마스터/디테일 관계를 설정하거나 인덱스를 지정하는 데 사용한 모든 속성을 복사합니다. 범위와 필터를 지정하는 속성은 새로운 *TSQLTable* 컴포넌트보다는 클라이언트 데이터셋에 설정해야 합니다.
- 원본 데이터셋이 *TQuery*, *TADOQuery* 또는 *TIBQuery* 컴포넌트였을 경우 새 *TSQLQuery* 컴포넌트의 *SQL* 속성을 원본 데이터셋의 *SQL* 속성으로 설정합니다. 새 *TSQLQuery*의 *Params* 속성을 원본 데이터셋의 *Params*나 *Parameters* 속성과 일치하도록 설정합니다. 마스터/디테일 관계 성립을 위해 *DataSource* 속성을 설정했으면 이 속성도 복사합니다.
- 원본 데이터셋이 *TStoredProc*, *TADOStoredProc* 또는 *TIBStoredProc* 컴포넌트였으면 새 *TSQLStoredProc* 컴포넌트의 *StoredProcName*을 원본 데이터셋의 *StoredProcName*이나 *ProcedureName* 속성으로 설정합니다. 새 *TSQLStoredProc* 의 *Params* 속성을 원본 데이터셋의 *Params*나 *Parameters* 속성 값과 일치하도록 설정합니다.

6 원본 애플리케이션에 있는 모든 데이터베이스 연결 컴포넌트(*TDatabase*, *TIBDatabase* 또는 *TADOConnection*)의 경우 새 데이터 모듈에 *TSQLConnection* 컴포넌트를 추가합니다. 또한 연결 컴포넌트없이 연결한(예를 들어 ADO 데이터셋의 *ConnectionString* 속성을 사용하거나 BDE 데이터셋의 *DatabaseName* 속성을 BDE 앨리어스로 설정함으로써) 모든 데이터베이스 서버의 *TSQLConnection* 컴포넌트도 추가해야 합니다.

7 단계 4의 각 *dbExpress* 데이터셋마다 해당 *SQLConnection* 속성을 적절한 데이터베이스 연결에 해당하는 *TSQLConnection* 컴포넌트로 설정합니다.

8 각 *TSQLConnection* 컴포넌트마다 데이터베이스 연결에 필요한 정보를 지정합니다. 이렇게 하려면 *TSQLConnection* 컴포넌트를 더블 클릭하여 Connection Editor를 표시하고 매개변수 값을 설정하여 적절한 설정을 나타내십시오. 단계 1의 새로운 데이터베이스 서버로 데이터를 전송했으면 새 서버에 맞는 설정을 지정합니다. 이전과 동일한 서버를 사용할 예정이면 원본 연결 컴포넌트에서 이 정보의 일부를 찾아볼 수 있습니다.

- 원본 애플리케이션이 *TDatabase*를 사용했을 경우에는 *Params*와 *TransIsolation* 속성에 나타나는 정보를 전송해야 합니다.
- 원본 애플리케이션이 *TADOConnection*을 사용했을 경우에는 *ConnectionString*과 *IsolationLevel* 속성에 나타나는 정보를 전송해야 합니다.

- 원본 애플리케이션이 *TIBDatabase*를 사용했을 경우에는 *DatabaseName*과 *Params* 속성에 나타나는 정보를 전송해야 합니다.
- 원본 연결 컴포넌트가 없을 경우 BDE 알리어스와 관련되었거나 데이터셋의 *ConnectionString* 속성에 나타난 정보를 전송해야 합니다.

이 매개변수 집합을 새로운 연결 이름으로 저장할 수 있습니다. 이 과정에 대한 자세한 내용은 21-2페이지의 "연결 제어"를 참조하십시오.

dbExpress 애플리케이션에서 데이터 업데이트

dbExpress 애플리케이션은 클라이언트 데이터셋을 사용하여 수정 기능을 지원합니다. 변경 사항을 클라이언트 데이터셋에 포스트하면 클라이언트 데이터셋의 메모리 데이터 스냅샷에는 이 변경 사항이 기록되지만 데이터베이스 서버에는 자동으로 기록되지 않습니다. 원본 애플리케이션이 업데이트 캐싱을 위해 클라이언트 데이터셋을 사용한 경우 Linux 에서 수정 기능을 지원하기 위해 어떤 항목을 변경할 필요가 없습니다. 그러나 Windows에서 레코드를 포스트할 때 데이터베이스 서버에 변경 사항을 기록하는 대부분의 데이터셋의 디폴트 동작을 사용한 경우에는, 클라이언트 데이터셋을 사용하기 위해 필요한 항목을 변경해야 합니다.

이전에 업데이트를 캐싱하지 않은 애플리케이션을 변환하는 방법에는 두 가지가 있습니다.

- 각 업데이트된 레코드가 포스트되자마자 데이터베이스 서버에 적용하기 위한 코드를 작성하여 Windows에서 데이터셋의 동작을 흉내낼 수 있습니다. 이렇게 하려면 업데이트를 데이터베이스 서버에 적용하는 *AfterPost* 이벤트 핸들러가 있는 클라이언트 데이터셋을 다음과 같이 사용하십시오.

```
void __fastcall TForm1::ClientDataSet1AfterPost(TDataSet *DataSet)
{
  TClientDataSet *pCDS = dynamic_cast<TClientDataSet *>(DataSet);
  if (pCDS)
    pCDS->ApplyUpdates(1);
}
```

- 캐싱된 업데이트를 다루기 위해 사용자 인터페이스를 조정할 수 있습니다. 이 접근 방법은 네트워크 트래픽의 양을 줄이고 트랜잭션 시간을 최소화하는 등의 이점이 있습니다. 그러나 캐싱된 업데이트 사용으로 전환하면 해당 업데이트를 데이터베이스 서버에 다시 적용할 시기를 정해야 하고, 사용자 인터페이스를 변경하여 업데이트된 애플리케이션을 사용자가 초기화하게 하거나 사용자의 편집 내용이 데이터베이스에 기록했는지 여부에 대한 정보를 제공해야 할 것입니다. 또한 사용자가 레코드를 포스트할 때 업데이트 오류가 감지되지 않으므로 사용자에 대해 이러한 오류를 보고하는 방법을 변경하여 어떤 업데이트로 문제를 일으켰는지와 어떤 타입의 문제가 일어났는지를 사용자가 알 수 있게 합니다.

원본 애플리케이션이 업데이트를 캐싱하기 위해 BDE나 ADO가 제공하는 지원을 사용했을 경우 코드를 수정하여 클라이언트 데이터셋 사용으로 전환해야 합니다. 다음 표는 BDE와 ADO 데이터셋에서 캐싱된 업데이트를 지원하는 속성, 이벤트 및 메소드와 *TClientDataSet*의 해당 속성, 메소드 및 이벤트를 나열한 것입니다.

BDE 데이터셋 (또는 TDatabase)	ADO 데이터셋	TclientDataSet	용도
CachedUpdates	*LockType*	필요없음, 클라이언트 데이터셋은 항상 업데이트를 캐싱함	캐싱된 업데이트가 효력이 있는지 결정합니다.
지원하지 않음	*CursorType*	지원하지 않음	서버의 변경으로 인해 데이터셋이 분리된 방법을 지정합니다.
UpdatesPending	지원하지 않음	*ChangeCount*	로컬 캐시가 데이터베이스에 적용하는 데 필요한 업데이트된 레코드를 포함하는지를 나타냅니다.
UpdateRecordTypes	*FilterGroup*	*StatusFilter*	업데이트된 레코드의 종류를 나타내어 캐싱된 업데이트를 적용할 때 보이게 합니다.
UpdateStatus	*RecordStatus*	*UpdateStatus*	레코드가 변경되지 않았는지, 수정되었는지, 삽입되었는지 또는 삭제되었는지를 나타냅니다.
OnUpdateError	지원하지 않음	*OnReconcileError*	레코드별로 업데이트 오류를 처리하기 위한 이벤트입니다.
ApplyUpdates (데이터셋이나 데이터베이스에서)	*UpdateBatch*	*ApplyUpdates*	로컬 캐시의 레코드를 데이터베이스에 적용합니다.
CancelUpdates	*CancelUpdates* 또는 *CancelBatch*	*CancelUpdates*	대기 중인 업데이트를 적용하지 않고 로컬 캐시에서 제거합니다.
CommitUpdates	자동 처리됨	*Reconcile*	성공적으로 업데이트된 애플리케이션 다음에 이어지는 업데이트 캐시를 지웁니다.
FetchAll	지원하지 않음	*GetNextPacket* (및 *PacketRecords*)	데이터베이스 레코드를 편집 및 업데이트할 목적으로 로컬 캐시에 복사합니다.
RevertRecord	*CancelBatch*	*RevertRecord*	업데이트가 아직 적용되지 않았으면 현재 레코드에 대한 업데이트를 취소합니다.

크로스 플랫폼 인터넷 애플리케이션

인터넷 애플리케이션은 클라이언트를 서버에 연결하기 위해 표준 인터넷 프로토콜을 사용하는 클라이언트/서버 애플리케이션입니다. 클라이언트/서버 통신용 표준 인터넷 프로토콜을 사용하므로 애플리케이션을 크로스 플랫폼으로 만들 수 있습니다. 예를 들어, 인터넷 애플리케이션의 서버사이드 프로그램은 컴퓨터의 웹 서버 소프트웨어를 통해 클라이언트와 통신합니다. 서버 애플리케이션은 일반적으로 Linux 또는 Windows용으로 작성되지만 크로스 플랫폼이 될 수 있습니다. 클라이언트는 두 플랫폼 중 어느 하나에 둘 수 있습니다.

C++Builder를 사용하면 웹 서버 애플리케이션을 나중에 Linux에서 배포하기 위하여 CGI 또는 Apache 애플리케이션으로 만들 수 있습니다. Windows에서는 Microsoft Server DLL(ISAPI), Netscape Server DLL(NSAPI) 및 Windows CGI 애플리케이션과 같은 다른 타입의 웹 서버를 만들 수 있습니다. 직접적인 CGI 애플리케이션과 Web Broker를 사용하는 일부 애플리케이션만이 Windows와 Linux 모두에서 실행됩니다.

인터넷 애플리케이션을 Linux로 이식

크로스 플랫폼으로 만들려는 기존 인터넷 애플리케이션이 있는 경우 웹 서버 애플리케이션을 Linux로 이식할 수 있습니다. 또는 Borland C++ 솔루션을 사용할 수 있게 되면 Linux에서 새 애플리케이션을 만들 수 있습니다. 웹 서버 작성에 대한 자세한 내용은 32장, "인터넷 서버 애플리케이션 생성"을 참조하십시오. 애플리케이션이 Web Broker를 사용하고 Web Broker 인터페이스에 쓰지만 원시 API 호출을 사용하지 않는 경우에는 Linux로 이식하는 것만큼 어렵지 않습니다.

애플리케이션이 ISAPI, NSAPI, Windows CGI 또는 기타 웹 API를 사용하는 경우에는 이식하기가 더 어렵습니다. 소스 파일을 통해 검색하고 이 API 호출을 Apache(Apache API용 함수 프로토타입에 대해서는 ..\Include\Vcl\httpd.hpp 참조)나 CGI 호출로 변환해야 합니다. 또한 14-2페이지의 "Windows 애플리케이션을 Linux로 이식"에 설명되어 있는 다른 모든 변경 내용을 수행해야 합니다.

장

15

패키지와 컴포넌트 사용

*패키지*란 C++Builder 애플리케이션, IDE 또는 둘 다에 의해 사용되는 특수한 동적 연결 라이브러리입니다. *런타임 패키지*는 사용자가 애플리케이션을 실행할 때 기능을 제공합니다. *디자인 타임 패키지*는 IDE에 컴포넌트를 설치하고 사용자 정의 컴포넌트에 대한 특수 속성 에디터를 만드는 데 사용합니다. 단일 패키지는 디자인 타임과 런타임 모두에서 사용할 수 있으며, 디자인 타임 패키지는 종종 런타임 패키지를 호출하여 작동합니다. 패키지 라이브러리는 다른 DLL과 구별하기 위해 확장자가 .bpl(Borland package library)인 파일로 저장됩니다.

다른 런타임 라이브러리와 마찬가지로 패키지는 애플리케이션 간에 공유할 수 있는 코드를 포함합니다. 예를 들어, 가장 자주 사용되는 C++Builder 컴포넌트는 vcl이라는 패키지에 있으며 개발자가 새 디폴트 애플리케이션을 생성할 때마다 자동으로 vcl을 사용합니다. 이러한 방법으로 만든 애플리케이션을 생성하는 경우, 애플리케이션의 실행 가능한 이미지에는 코드 및 코드에 고유한 데이터만 포함되고 일반 코드는 vcl60.bpl이라는 런타임 패키지에 포함됩니다. 여러 개의 패키지를 사용할 수 있는 애플리케이션이 설치된 컴퓨터에는 모든 애플리케이션 및 C++Builder IDE 자체가 공유하는 vcl60.bpl의 복사본 한 부만 있으면 됩니다.

C++Builder는 VCL 및 CLX 컴포넌트를 캡슐화하는 여러 개의 런타임 패키지를 제공합니다. 또한 C++Builder는 디자인 타임 패키지를 사용하여 IDE에서 컴포넌트를 처리합니다.

개발자는 패키지를 사용하거나 사용하지 않고 애플리케이션을 생성할 수 있습니다. 단, IDE에 사용자 정의 컴포넌트를 추가하려면 반드시 컴포넌트를 디자인 타임 패키지로 설치해야 합니다.

애플리케이션 간에 공유되는 새로운 런타임 패키지를 만들 수 있습니다. C++Builder 컴포넌트를 작성하는 경우에는 컴포넌트를 설치하기 전에 디자인 타임 패키지에 컴포넌트를 생성해야 합니다.

패키지를 사용하는 이유

디자인 타임 패키지는 사용자 정의 컴포넌트 배포 및 설치 작업을 단순화합니다. 옵션인 런타임 패키지는 일반적인 프로그래밍에 적용되는 여러 가지 장점을 제공합니다. 재사용되는 코드를 런타임 라이브러리에 생성하여 애플리케이션 간에 공유할 수 있습니다. 예를 들어, C++Builder 자체를 포함한 모든 애플리케이션은 패키지를 통해 표준 컴포넌트에 액세스할 수 있습니다. 애플리케이션마다 해당 실행 파일에 컴포넌트 라이브러리의 복사본을 연결할 필요가 없으므로 실행 파일의 크기가 훨씬 작으며 시스템 리소스와 하드 디스크 저장 공간을 모두 절약할 수 있습니다. 또한 애플리케이션에 고유한 코드만 각 빌드와 함께 컴파일되므로 패키지를 사용하면 빠르게 컴파일할 수 있습니다.

패키지 및 표준 DLL

IDE를 통해 사용할 수 있는 사용자 정의 컴포넌트를 만들려면 패키지를 만드십시오. 애플리케이션 생성에 사용한 개발 도구와 상관 없이 모든 애플리케이션에서 호출할 수 있는 라이브러리를 생성하려면 표준 DLL을 만드십시오.

다음 표는 패키지와 관련된 파일 타입을 나열한 것입니다.

표 15.1 패키지 파일

파일 확장자	설명
bpk	프로젝트 옵션 소스 파일. 이 파일은 패키지 프로젝트의 XML 부분입니다. 결합된 *ProjectName*.bpk 및 *ProjectName*.cpp는 패키지 프로젝트에 의해 사용되는 설정, 옵션 및 파일 관리에 사용됩니다.
bpl	런타임 패키지입니다. 이 파일은 특수한 C++Builder 특정 기능이 있는 Windows .dll입니다. bpl에 대한 기본 이름은 bpk 소스 파일의 기본 이름입니다.
cpp	*ProjectName*.cpp에는 패키지에 대한 엔트리 포인트가 포함되어 있습니다. 또한 패키지에 포함된 각 컴포넌트는 일반적으로 .cpp 파일 내에 상주합니다.
h	헤더 파일 또는 컴포넌트용 인터페이스입니다. *ComponentName*.h는 *ComponentName*.cpp와 같이 제공됩니다.
lib	애플리케이션이 런타임 패키지를 사용하지 않을 때 .bpi 대신 사용되는 정적 라이브러리 또는 .objs의 컬렉션입니다. **-Gl**(Generate .LIB 파일)을 선택한 경우에만 생성됩니다.
obj	패키지에 포함된 유닛 파일에 대한 바이너리 이미지입니다. 필요할 때마다 각 .cpp 파일에 대해 하나의 obj가 만들어집니다.
bpi	Borland 패키지 임포트 라이브러리입니다. .bpi는 각 패키지에 대해 만들어집니다. bpl에 대한 bpi의 관계는 dll에 대한 임포트 라이브러리와 유사합니다. 이 파일은 패키지를 사용하는 애플리케이션에 의해 링커로 전달되어 패키지 내의 함수에 대한 참조를 해결합니다. bpi의 기본 이름은 패키지 소스 파일의 기본 이름입니다.

패키지에 VCL이나 CLX 또는 둘 다를 포함할 수 있습니다. 여러 플랫폼에서 사용할 수 있도록 설계된 패키지는 CLX 컴포넌트만 포함해야 합니다.

참고 패키지는 애플리케이션에 있는 다른 모듈과 전역 데이터를 공유합니다.

런타임 패키지

런타임 패키지는 C++Builder 애플리케이션과 함께 배포되며 사용자가 애플리케이션을 실행할 때 기능을 제공합니다.

패키지를 사용하는 애플리케이션을 실행하려면 컴퓨터에 애플리케이션의 실행 파일과 애플리케이션이 사용하는 모든 패키지(.bpl 파일)가 설치되어 있어야 합니다. .bpl 파일은 반드시 애플리케이션이 .bpl 파일을 사용하는 시스템 경로에 있어야 합니다. 애플리케이션을 배포할 때는 반드시 사용자가 필요한 .bpl 파일의 올바른 버전을 갖고 있는지 확인해야 합니다.

애플리케이션에서 패키지 사용

다음과 같은 방법으로 애플리케이션에서 패키지를 사용합니다.

1. IDE에서 프로젝트를 로드하거나 만듭니다.
2. Project|Options를 선택합니다.
3. Packages 탭을 선택합니다.
4. Build with Runtime Packages 체크 박스를 선택한 다음, 아래의 에디트 박스에 하나 이상의 패키지 이름을 입력합니다. 설치된 디자인 타임 패키지와 연결된 런타임 패키지는 이미 에디트 박스에 나열되어 있습니다.
5. 기존의 리스트에 패키지를 추가하려면 Add 버튼을 클릭하고 Add Runtime Package 다이얼로그 박스에 새 패키지 이름을 입력합니다. 사용 가능한 패키지 리스트에서 검색하려면 Add 버튼을 클릭한 다음 Add Runtime Package 다이얼로그 박스의 Package Name 에디트 박스 옆에 있는 Browse 버튼을 클릭합니다.

   Add Runtime Package 다이얼로그 박스의 Search Path 에디트 박스를 편집하면 C++Builder의 전역 Library Path가 변경됩니다.

   패키지 이름에 파일 확장자 또는 C++Builder 릴리스를 나타내는 버전 번호를 포함시킬 필요는 없습니다. 예를 들면, vcl60.bpl은 vcl로 표시하면 됩니다. Runtime Packages 에디트 박스에 여러 개의 이름을 직접 입력하는 경우 세미콜론으로 분리합니다. 예를 들면, 다음과 같습니다.

   ```
   rtl;vcl;vcldb;vclado;vclx;vclbde;
   ```

Runtime Packages 에디트 박스에 나열된 패키지는 자동으로 개발자의 애플리케이션에 연결됩니다. 중복되는 패키지 이름은 무시되며 Build with runtime packages 체크 박스의 선택을 해제하면 애플리케이션이 패키지 없이 연결됩니다.

런타임 패키지는 현재 프로젝트에 대해서만 선택할 수 있습니다. 현재 선택된 항목이 새 프로젝트에 대해서도 자동으로 기본값으로 적용되도록 하려면 다이얼로그 박스 하단에 있는 Defaults 체크 박스를 선택하십시오.

패키지를 사용하여 생성된 애플리케이션은 여전히 애플리케이션이 사용하는 패키지화된 유닛의 헤더 파일을 포함해야 합니다. 예를 들면, 데이터베이스 컨트롤을 사용하는 애플리케이션은 vcldb 패키지를 사용하더라도 다음 명령문이 필요합니다.

```
#include "vcldb.h"
```

생성된 소스 파일에서 C++Builder는 자동으로 이러한 **#include** 문을 만듭니다.

동적 패키지 로딩

런타임에서 패키지를 로드하려면 *LoadPackage* 함수를 호출하십시오. *LoadPackage*는 패키지를 로드하고 중복 유닛을 검사하며 패키지에 포함된 모든 유닛의 초기화 블록을 호출합니다. 예를 들어, 다음 코드는 특정 파일이 파일 선택 다이얼로그 박스에서 선택될 때 실행될 수 있습니다.

```
if (OpenDialog1->Execute())
  PackageList->Items->AddObject(OpenDialog1->FileName, (TObject
*)LoadPackage(OpenDialog1->FileName));
```

패키지를 동적으로 언로드하려면 *UnloadPackage*를 호출하십시오. 패키지에서 정의된 클래스의 인스턴스를 소멸시키고 인스턴스에 의해 등록된 클래스의 등록을 해제하는 경우에는 주의해야 합니다.

사용할 런타임 패키지 결정

C++Builder는 rtl 및 vcl을 포함하여 기본 랭귀지 및 컴포넌트 지원을 제공하는 여러 개의 런타임 패키지와 함께 제공됩니다.

vcl 패키지에는 가장 일반적으로 사용되는 컴포넌트가 포함되어 있으며 rtl 패키지에는 컴포넌트외의 모든 시스템 함수와 Windows 인터페이스 요소가 포함되어 있습니다. 그러나 데이터베이스 또는 각각의 패키지에서 사용할 수 있는 특수 컴포넌트는 포함되지 않습니다.

패키지를 사용하는 클라이언트/서버 데이터베이스 애플리케이션을 만들려면 vcl, vcldb, rtl 및 dbrtl을 포함하여 몇 개의 런타임 패키지가 필요합니다. 애플리케이션 내에서 Outline 컴포넌트를 사용하려는 경우에도 vclx가 필요합니다. 이러한 패키지를 사용하려면 Project|Options를 선택하고 Packages 탭을 선택한 다음 Runtime Packages 에디트 박스에 다음 리스트를 입력하십시오.

```
rtl;vcl;vcldb;vclx;
```

실제로 vcl 및 rtl은 vcldb의 Requires 리스트에서 참조되므로 포함시키지 않아도 됩니다(15-9페이지의 "Requires 리스트" 참조). 애플리케이션 컴파일은 vcl 및 rtl이 Runtime Packages 에디트 박스에 포함되어 있는지에 상관 없이 동일하게 생성될 수 있습니다.

사용자 정의 패키지

사용자 정의 패키지는 개발자가 직접 코딩하고 생성한 bpl이거나 서드파티 업체의 기존 패키지입니다. 애플리케이션과 함께 사용자 정의 런타임 패키지를 사용하려면 Project|Options를 선택하고 Packages 페이지의 Runtime Packages 에디트 박스에 패키지 이름을 추가하십시오. 예를 들어, stats.bpl이라는 통계 패키지가 있다고 가정할 때 이를 애플리케이션에서 사용하려면 Runtime Packages 에디트 박스에 다음과 같이 입력해야 합니다.

```
rtl;vcl;vcldb;stats
```

사용자가 패키지를 직접 만드는 경우 필요에 따라 리스트에 추가할 수 있습니다.

디자인 타임 패키지

디자인 타임 패키지는 IDE의 컴포넌트 팔레트에 컴포넌트를 설치하고 사용자 정의 컴포넌트에 대한 특수 속성 에디터를 만드는 데 사용합니다.

C++Builder는 IDE에 미리 설치된 여러 개의 디자인 타임 컴포넌트 패키지와 함께 제공됩니다. 설치되는 패키지는 사용하는 C++Builder의 버전 및 이를 사용자 정의했는지 여부에 따라 결정됩니다. Component|Install Packages를 선택하면 시스템에 설치되는 패키지 리스트를 확인할 수 있습니다.

디자인 타임 패키지는 Requires 리스트에서 참조하는 런타임 패키지를 호출하여 작업합니다 (15-9페이지의 "Requires 리스트" 참조). 예를 들면, dclstd는 vcl을 참조합니다. dclstd 자체는 컴포넌트 팔레트에서 사용할 수 있는 표준 컴포넌트를 최대한 활용할 수 있는 추가 기능을 포함하고 있습니다.

참고 규칙에 따라 IDE 디자인 패키지는 dcl과 함께 시작되며 bin 디렉토리에 상주합니다. ..\bin\Dclstd 등의 디자인 패키지에는 ..\lib\vcl.lib, ..\lib\vcl.bpi 및 런타임 패키지 자체인 Windows\System\vcl60.bpl 등 링커에 대한 대응 요소가 있습니다.

미리 설치된 패키지 외에도, IDE에 자신의 컴포넌트 패키지나 서드파티 개발자의 컴포넌트 패키지를 설치할 수 있습니다. dclusr 디자인 타임 패키지는 새 컴포넌트에 대한 디폴트 컨테이너로 제공됩니다.

컴포넌트 패키지 설치

모든 컴포넌트는 IDE에 패키지로 설치됩니다. 자체 컴포넌트를 생성한 경우에는 자체 컴포넌트를 포함한 패키지를 만들고 생성합니다(15-6페이지의 "패키지 생성 및 편집" 참조). 컴포넌트 소스 코드는 V부, "사용자 정의 컴포넌트 생성"에서 설명한 모델을 따라야 합니다. 단일 패키지에 여러 유닛을 추가하고 유닛 각각에 컴포넌트가 있는 경우 네임스페이스에 해당 패키지의 이름이 있는 있는 모든 컴포넌트에 대해 단일한 Register 함수를 만들어야 합니다.

자신의 컴포넌트 또는 서드파티 업체의 컴포넌트를 설치하거나 제거하려면 다음 단계를 따르십시오.

1. 새 패키지를 설치하는 경우 패키지 파일을 로컬 디렉토리로 복사하거나 이동합니다. 패키지가 .bpl, .bpi, .lib 및 .obj 파일과 함께 제공되는 경우에는 반드시 이들 파일을 모두 복사합니다. .bpl, .bpi, .lib 및 .obj 파일에 대한 자세한 내용은 "패키지 및 표준 DLL"을 참조하십시오.

   .bpi, 헤더 파일, .lib 또는 .obj(배포에 포함된 경우에 한함) 파일을 저장하는 디렉토리는 반드시 C++Builder Library Path에 있어야 합니다.
2. IDE 메뉴에서 Component|Install Packages를 선택하거나 Project|Options를 선택하고 Packages 탭을 클릭합니다.
3. 사용 가능한 패키지 리스트가 "Design packages" 아래에 나타납니다.
   - IDE에 패키지를 설치하려면 옆에 있는 체크 박스를 선택하십시오.
   - 패키지를 제거하려면 체크 박스의 선택을 해제하십시오.

- 설치된 패키지에 포함된 컴포넌트 리스트를 보려면 패키지를 선택하고 Components를 클릭하십시오.
- 리스트에 패키지를 추가하려면 Add를 클릭하고 Add Design Package 다이얼로그 박스에서 .bpl 파일이 있는(단계 1 참조) 디렉토리로 이동하십시오. .bpl 파일을 선택하고 Open을 클릭합니다.
- 리스트에서 패키지를 제거하려면 패키지를 선택하고 Remove를 클릭하십시오.

4 OK를 클릭합니다.

패키지의 컴포넌트는 컴포넌트의 *RegisterComponents* 프로시저에서 지정한 컴포넌트 팔레트 페이지에 동일한 프로시저에서 할당된 이름으로 설치됩니다.

디폴트 설정을 변경하지 않는 한 새 프로젝트는 사용 가능한 모든 패키지를 설치하여 만들어집니다. 현재의 설치 선택 사항이 자동으로 새 프로젝트에 기본값으로 적용되게 하려면 Project Options 다이얼로그 박스의 Packages 탭 하단에 있는 Default 체크 박스를 선택하십시오.

패키지를 제거하지 않고 컴포넌트 팔레트에서 컴포넌트를 제거하려면 Component|Configure Palette를 선택하거나 Tools|Environment Options를 선택하고 Palette 탭을 클릭하십시오. Palette Options 탭에는 설치된 컴포넌트와 함께 이러한 컴포넌트가 표시되는 컴포넌트 팔레트 페이지의 이름이 나열되어 있습니다. 컴포넌트를 선택하고 Hide를 클릭하면 팔레트에서 해당 컴포넌트가 제거됩니다.

패키지 생성 및 편집

패키지를 만들려면 다음을 지정해야 합니다.

- 패키지 *이름*
- 새 패키지에 *필요*하거나 새 패키지에 연결되는 다른 패키지 리스트
- 패키지가 생성될 때 그 패키지에 *포함*되거나 연결된 유닛 파일 리스트. 패키지는 본질적으로 이러한 소스 코드 유닛에 대한 랩퍼입니다 . Contains 리스트는 패키지로 생성하고자 하는 사용자 정의 컴포넌트에 대한 소스 코드 유닛을 입력하는 곳입니다.

패키지는 C++ 소스(.cpp) 파일 및 확장자가 .bpk인 프로젝트 옵션 파일에 의해 정의됩니다. 이러한 파일은 패키지 에디터에 의해 생성됩니다.

패키지 생성

패키지를 만들려면 아래의 절차에 따르십시오. 여기서 개괄적으로 설명되는 각 단계에 대한 자세한 내용은 15-9페이지의 "패키지 구조 이해"를 참조하십시오.

참고 크로스 플랫폼 개발 등의 경우에는 패키지 파일(.bpk)에서 ifdef를 사용하지 마십시오. 단, 소스 코드에서는 사용할 수 있습니다.

1 File|New|Other를 선택하고 Package 아이콘을 선택한 다음 OK를 클릭합니다.

2 패키지 에디터에 생성된 패키지가 나타납니다.

3 패키지 에디터는 새 패키지의 *Requires* 노드와 *Contains* 노드를 보여 줍니다.

4 Contains 리스트에 유닛을 추가하려면 Add to package 스피드 버튼을 클릭하십시오. Add unit 페이지의 Unit file name 에디트 박스에 .cpp 파일 이름을 입력하거나 Browse를 클릭하여 파일을 찾은 다음 OK 를 클릭하십시오 . 그러면 선택한 유닛이 패키지 에디터의 Contains 노드 아래에 표시됩니다. 이 단계를 반복하면 다른 유닛을 추가할 수 있습니다.

5 Requires 리스트에 패키지를 추가하려면 Add to package 스피드 버튼을 클릭하십시오. Requires 페이지의 Package name 에디트 박스에 .bpi 파일 이름을 입력하거나 Browse를 클릭하여 해당 파일을 찾은 다음 OK를 클릭합니다. 그러면 선택한 유닛이 패키지 에디터의 Requires 노드 아래에 표시됩니다. 이 단계를 반복하면 다른 유닛을 추가할 수 있습니다.

6 Options 스피드 버튼을 클릭하고 생성하려는 패키지 종류를 결정합니다.

- 디자인 타임 전용 패키지 (런타임에서 사용할 수 없는 패키지) 를 만들려면 Designtime only 라디오 버튼을 선택하십시오. 또는 **-Gpd** 링커 스위치를 bpk 파일에 추가합니다 (`LFLAGS = ... -Gpd ...`).
- 런타임 전용 패키지(설치할 수 없는 패키지)를 만들려면 Runtime only 라디오 버튼을 선택하십시오. 또는 **-Gpr** 링커 스위치를 bpk 파일에 추가합니다(`LFLAGS = ... -Gpr ...`).
- 디자인 타임과 런타임 모두에 사용할 수 있는 패키지를 만들려면 Designtime and runtime 라디오 버튼을 선택하십시오.

7 패키지 에디터에서 Compile package 스피드 버튼을 클릭하여 패키지를 컴파일합니다.

참고 Install 버튼을 클릭하여 강제로 컴파일하도록 할 수도 있습니다.

기존 패키지 편집

다음과 같은 여러 가지 방법으로 기존 패키지를 열어 편집할 수 있습니다.

- File|Open(또는 File|Reopen)을 선택하고 cpp 또는 bpk 파일을 선택합니다.
- Component|Install Packages를 선택하고 Design Packages 리스트에서 패키지를 선택한 다음 Edit 버튼을 클릭합니다.
- 패키지 에디터가 열리면 Requires 노드에서 패키지 중 하나를 선택하고 마우스 오른쪽 버튼을 클릭한 다음 Open을 선택합니다.

패키지의 설명을 편집하거나 사용 옵션을 설정하려면 패키지 에디터에서 Options 스피드 버튼을 클릭하고 Description 탭을 선택하십시오.

Project Options 다이얼로그 박스의 왼쪽 하단에는 Default 체크 박스가 있습니다. 이 체크 박스가 선택되어 있을 때 OK를 클릭하면 선택한 옵션이 새 패키지 프로젝트에 대한 디폴트 설정으로 저장됩니다. 원래의 기본값을 복원하려면 default.bpk 파일을 삭제하거나 이름을 바꾸십시오.

패키지 소스 파일 및 프로젝트 옵션 파일

패키지 소스 파일의 확장자는 .cpp입니다. 패키지 프로젝트 옵션 파일은 XML 형식을 사용하여 만들어지며 .bpk(Borland 패키지) 확장자를 사용합니다. Contains 또는 Requires 절을 마우스 오른쪽 버튼으로 클릭하고 Edit Option Source를 선택하여 패키지 에디터에서 패키지 프로젝트 옵션 파일을 표시합니다.

참고 C++Builder가 .bpk 파일을 유지 보수하므로, 일반적으로 개발자가 수동으로 수정할 필요는 없습니다. Project Options 다이얼로그 박스의 Packages 탭을 사용하여 원하는 항목을 변경해야 합니다.

MyPack이라는 패키지에 대한 프로젝트 옵션 파일의 일부는 다음과 같이 나타날 수 있습니다.

```
<MACROS>
    <VERSION value="BCB.05.02"/>
    <PROJECT value="MyPack.bpl"/>
    <OBJFILES value="MyPack.obj Unit2.obj Unit3.obj"/>
    <RESFILES value="MyPack.res"/>
    <IDLFILES value=""/>
    <IDLGENFILES value=""/>
    <DEFFILE value=""/>
    <RESDEPEN value="$(RESFILES)"/>
    <LIBFILES value=""/>
    <LIBRARIES value=""/>
    <SPARELIBS value="Vcl60.lib"/>
    <PACKAGES value="Vcl60.bpi vcldbx60.bpi"/>
    .
    .
    .
```

이런 경우, MYPACK.cpp는 다음 코드를 포함합니다.

```
USERES("MyPack.res");
USEPACKAGE("vcl60.bpi");
USEPACKAGE("vcldbx60.bpi");
USEUNIT("Unit2.cpp");
USEUNIT("Unit3.cpp");
```

MyPack의 Contains 리스트에는 MyPack 자체, Unit2, Unit3 등 세 개의 유닛이 포함됩니다. MyPack의 Requires 리스트에는 VCL 및 VCLDBX가 포함됩니다.

컴포넌트 패키지화

New Component 마법사를 사용하여 새 컴포넌트를 만들면(Component|New Component 선택) C++Builder는 필요한 곳에 PACKAGE 매크로를 삽입합니다. 그러나 C++Builder 이전 버전의 사용자 정의 컴포넌트가 있는 경우 수동으로 두 위치에 PACKAGE를 추가해야 합니다.

C++Builder 컴포넌트에 대한 헤더 파일 선언에는 다음과 같이 **class**라는 단어 뒤에 반드시 미리 정의된 PACKAGE 매크로가 포함되어야 합니다.

```
class PACKAGE MyComponent : ...
```

또한 컴포넌트가 정의되는 cpp 파일의 *Register* 함수 선언에 PACKAGE 매크로가 포함되어야 합니다.

```
void __fastcall PACKAGE Register()
```

PACKAGE 매크로는 클래스를 임포트하고 결과 bpl 파일에서 엑스포트할 수 있는 명령문으로 확장됩니다.

패키지 구조 이해

패키지는 다음을 포함합니다.

- 패키지 이름
- Requires 리스트
- Contains 리스트

패키지 이름 지정

패키지 이름은 프로젝트 내에서 고유해야 합니다. 패키지 이름을 Stats라고 지정하면 패키지 에디터가 각각 Stats.cpp 및 Stats.bpk라는 소스 파일과 프로젝트 옵션 파일을 생성하고 컴파일러와 링크가 각각 Stats.bpl, Stats.bpi 및 Stats.lib라는 실행 파일, 바이너리 이미지 및 정적 라이브러리(옵션)를 생성합니다. 애플리케이션에서 패키지를 사용하려면 Project|Options를 선택하고 Packages 탭을 클릭하여 Stats를 Runtime Packages 에디트 박스에 추가하십시오.

또한 패키지 이름에 접두사, 접미사 및 버전 번호를 추가할 수 있습니다. 패키지 에디터가 열려 있을 때 Options 버튼을 클릭합니다. Project Options 다이얼로그 박스의 Description 페이지에서 LIB Suffix, LIB Prefix 또는 LIB Version에 해당되는 텍스트 또는 값을 입력합니다. 예를 들어, 패키지 프로젝트에 버전 번호를 추가하려면 LIB Version 뒤에 6을 입력하여 *Package1*이 *Package1*.bpl.6을 생성하도록 하십시오.

Requires 리스트

Requires 리스트에는 현재 패키지에서 사용하는 기타 외부 패키지가 나열되어 있습니다. Requires 리스트에 포함된 외부 패키지는 컴파일 시에 자동으로 현재 패키지와 외부 패키지에 포함된 유닛 중 하나를 사용하는 애플리케이션에 연결됩니다.

패키지에 포함된 유닛 파일이 다른 패키지 유닛을 참조하면 다른 패키지가 개발자 패키지의 Requires 리스트에 표시되어야 하거나 개발자가 직접 이러한 패키지를 추가해야 합니다. 다른 패키지가 Requires 리스트에서 생략되면 컴파일러가 이들을 '암시적으로 포함된 유닛'으로 패키지로 임포트합니다.

참고 개발자가 만드는 대부분의 패키지에는 rtl이 필요합니다. VCL 컴포넌트를 사용하는 경우에는 vcl 패키지도 필요합니다. 클로스 플랫폼 프로그램용으로 CLX 컴포넌트를 사용하는 경우 VisualCLX를 포함시켜야 합니다.

순환 패키지 참조 방지

패키지는 Requires 리스트에 순환 참조를 포함할 수 없습니다. 즉, 다음과 같은 의미입니다.

- 패키지는 각각의 Requires 리스트에서 자신을 참조할 수 없습니다.

- 참조 체인은 해당 체인에 있는 패키지를 참조하지 않고 종료되어야 합니다. 패키지 A가 패키지 B를 요청하면 패키지 B는 패키지 A를 요청할 수 없습니다. 패키지 A가 패키지 B를 요청하고 패키지 B가 패키지 C를 요청하면 패키지 C는 패키지 A를 요청할 수 없습니다.

중복 패키지 참조 처리

패키지의 Requires 리스트 또는 Runtime Packages 에디트 박스 내의 중복 참조는 링커에 의해 무시됩니다. 그러나 프로그래밍의 명확성과 가독성을 높이려면 중복 패키지 참조를 찾아서 제거해야 합니다.

Contains 리스트

Contains 리스트는 패키지에 연결되는 유닛 파일을 식별합니다. 새로운 패키지를 작성하는 경우 cpp 파일에 소스 코드를 두고 cpp 파일을 Contains 리스트에 포함시킵니다.

중복 소스 코드 사용 방지

패키지는 다른 패키지의 Contains 리스트에 표시될 수 없습니다.

패키지의 Contains 리스트에 직접적으로 포함되거나 유닛에 간접적으로 포함된 모든 유닛은 링크 시 패키지에 연결됩니다.

유닛은 *C++Builder IDE* 등을 *포함하여* 같은 애플리케이션에서 사용하는 둘 이상의 패키지에 직간접적으로 포함될 수 없습니다. 다시 말해서 vcl의 유닛 중 하나를 포함하는 패키지를 만드는 경우 IDE에 패키지를 설치할 수 없습니다. 다른 패키지에서 이미 패키지화된 유닛 파일을 사용하려면 첫 번째 패키지를 두 번째 패키지의 Requires 리스트에 두십시오.

패키지 생성

IDE 또는 명령줄에서 패키지를 생성할 수 있습니다. 다음과 같은 방법으로 IDE에서 패키지만을 다시 생성합니다.

1. File|Open을 선택하고 패키지 소스 파일이나 프로젝트 옵션 파일을 선택한 다음 Open을 클릭합니다.
2. 에디터가 열리면 Project|Make 또는 Project|Build를 선택합니다.

참고 또는 File|New|Other를 선택하고 Package Editor를 더블 클릭할 수도 있습니다. Install 버튼을 클릭하여 패키지 프로젝트를 만듭니다. 설치, 작성 또는 컴파일하여 생성할 옵션을 보려면 패키지 프로젝트 노드를 마우스 오른쪽 버튼으로 클릭합니다.

생성되지 않은 .cpp 파일을 추가하는 경우에는 컴파일러 지시어 중의 하나를 패키지 소스 코드에 추가합니다. 자세한 내용은 아래의 "패키지별 컴파일러 지시어"를 참조하십시오.

명령줄에서 연결하는 경우 여러 개의 패키지별 링커 스위치를 사용할 수 있습니다. 자세한 내용은 15-12페이지의 "명령줄 컴파일러와 링커 사용"을 참조하십시오.

패키지별 컴파일러 지시어

다음 표는 소스 코드에 삽입할 수 있는 패키지별 컴파일러 지시어를 나열한 것입니다.

표 15.2 패키지별 컴파일러 지시어

지시어	용도
#pragma package(smart_init)	패키지화된 유닛이 족송성에 따라 결정된 순서대로 초기화되게 합니다(디폴트로, 패키지 소스 파일에 포함되어 있음).
#pragma package(smart_init, weak)	유닛을 "약하게" 패키지화합니다. 아래의 "약한 패키징"을 참조하십시오. 지시어는 유닛 소스 파일에 둡니다.

모든 라이브러리에서 사용할 수 있는 추가 지시어에 대한 자세한 내용은 7-10 페이지의 "패키지 및 DLL 생성"을 참조하십시오.

약한 패키징

#pragma package(smart_init, weak) 지시어는 .obj 파일이 패키지의 .bpi 및 .bpl 파일에 저장되는 방식에 영향을 줍니다(컴파일러 및 링커에 의해 생성된 파일에 대한 자세한 내용은 15-12 페이지의 "컴파일하여 생성된 패키지 파일" 참조). **#pragma package(smart_init, weak)**가 유닛 파일에 포함되어 있으면 가능한 경우에 한해 링커가 bpl 파일에서 유닛을 생략하며, 다른 애플리케이션이나 패키지가 요청하면 유닛의 패키지화되지 않은 로컬 복사본을 만듭니다. 이 지시어로 컴파일된 유닛을 *약하게 패키지화된* 유닛이라고 합니다.

예를 들어, UNIT1이라는 유닛 하나만을 포함하는 PACK 패키지를 만들었으며 UNIT1은 추가 유닛을 사용하지 않지만 RARE.dll을 호출한다고 가정합니다. 개발자가 새로운 패키지를 생성할 때 UNIT1.cpp에 **#pragma package(smart_init, weak)**를 포함시키면 UNIT1이 PACK.bpl에 포함되지 않으며 PACK과 함께 RARE.dll의 복사본을 배포할 필요가 없습니다. 그러나 UNIT1은 여전히 PACK.bpi에 포함됩니다. UNIT1이 PACK을 사용하는 다른 패키지나 애플리케이션에 의해 참조되면 PACK.bpi로부터 복사되며 직접 프로젝트로 연결됩니다.

이제 두 번째 유닛인 UNIT2를 PACK에 추가하며 UNIT2가 UNIT1을 사용한다고 가정합니다. 이런 경우 개발자가 UNIT1.cpp에 **#pragma package(smart_init, weak)**를 포함시켜 PACK을 컴파일하더라도 링커는 UNIT1을 PACK.bpl에 포함시킵니다. 그러나 UNIT1을 참조하는 다른 패키지나 애플리케이션은 PACK.bpi로부터 가져온 패키지화되지 않은 복사본을 사용하게 됩니다.

참고 **#pragma package(smart_init, weak)** 지시어를 포함한 유닛 파일은 전역 변수를 가져서는 안됩니다.

#pragma package(smart_init, weak)는 자신의 bpl 파일을 다른 C++Builder 프로그래머에게 배포하는 개발자를 위한 고급 기능입니다. 이 기능을 사용하면 가끔씩 사용되는 DLL의 배포를 방지하고 동일한 외부 라이브러리에 의존하는 패키지 간의 충돌을 방지할 수 있습니다.

예를 들어, C++Builder의 PenWin 유닛은 PenWin.dll을 참조합니다. 대부분의 프로젝트는 PenWin을 사용하지 않으며 컴퓨터에 PenWin.dll이 설치되어 있지 않은 경우가 많습니다. 이 때문에 PenWin 유닛은 vcl에서 약하게 패키지화되어 있습니다. PenWin과 vcl 패키지를 사용하는 프로젝트를 연결하면 PenWin이 vcl60.bpi에서 복사되며 개발자 프로젝트에 직접 연결됩니다. 그 결과로 만들어지는 실행 파일은 PenWin.dll에 정적으로 연결됩니다.

PenWin을 약하게 패키지화하지 않으면 두 가지 문제가 발생합니다. 첫째, vcl 자체가 정적으로 PenWin.dll에 링크되어 PenWin.dll이 설치되지 않은 컴퓨터에는 로드할 수 없습니다. 둘째, PenWin을 포함한 패키지를 만들려고 하면 생성 오류가 발생하여 PenWin 유닛이 vcl과 개발자의 패키지에 모두 포함됩니다. 따라서 약한 패키징을 사용하지 않고는 PenWin을 vcl의 표준 배포에 포함시킬 수 없습니다.

명령줄 컴파일러와 링커 사용

명령줄에서 연결하는 경우 **-Tpp** 링커 스위치를 사용하여 프로젝트가 패키지로 생성되었는지 확인합니다. 다음 표는 기타 패키지별 스위치를 나열한 것입니다.

표 15.3 패키지별 명령줄 링커 스위치

스위치	용도
-Tpp	프로젝트를 패키지로 생성합니다. 디폴트로, 패키지 프로젝트 파일에 포함됩니다.
-Gi	생성된 bpi 파일을 저장합니다. 디폴트로, 패키지 프로젝트 파일에 포함됩니다.
-Gpr	런타임 전용 패키지를 생성합니다.
-God	디자인 타임 전용 패키지를 생성합니다.
-Gl	.lib 파일을 생성합니다.
-D "*description*"	패키지와 함께 지정된 설명을 저장합니다.

-Gpr 및 **-Gpd** 스위치는 패키지 프로젝트에서만 사용할 수 있는 Project Options 다이얼로그 박스의 Description 페이지에 있는 Runtime Package 및 Design Package 체크 박스에 해당됩니다. **-Gpr**이나 **-Gpd** 중 아무것도 사용하지 않으면 결과 패키지가 디자인 타임과 런타임에서 모두 작동합니다. **-D** 스위치는 동일한 페이지의 Description 에디트 박스에 해당되며 **-Gl** 스위치는 Project Options 다이얼로그 박스의 Linker 페이지에 있는 Generate .lib File 체크 박스에 해당됩니다.

참고 Project|Export Makefile을 선택하면 makefile을 생성하여 명령줄에서 사용할 수 있습니다.

컴파일하여 생성된 패키지 파일

패키지를 만들려면 .bpk 확장자가 있는 프로젝트 옵션 파일을 사용하여 소스(.cpp) 파일을 컴파일합니다. 소스 파일의 기본 이름은 컴파일러에 의해 생성된 파일의 기본 이름과 일치해야 합니다. 즉, 소스 파일이 TRAYPAK.cpp이면 프로젝트 옵션 파일인 TRAYPAK.bpk가 포함되어야 합니다.

```
<PROJECT value="Traypak.bpl"/>
```

이런 경우, 프로젝트를 컴파일하고 연결하면 TRAYPAK.bpl이라는 패키지가 만들어집니다.

패키지를 컴파일하고 연결하면 bpi, bpl, obj 및 lib 파일이 만들어집니다. bpi, bpl 및 lib 파일은 Tools|Environment Options 다이얼로그 박스의 Library 페이지에서 지정된 디렉토리에 디폴트로 생성됩니다. 패키지 에디터에서 Options 스피드 버튼을 클릭하여 Project Options 다이얼로그 박스를 표시하고 Directories/Conditionals 페이지를 변경하면 디폴트 설정을 오버라이드할 수 있습니다.

패키지 배포

패키지를 배포하는 것은 다른 애플리케이션을 배포하는 것과 아주 유사합니다. 패키지와 함께 배포되는 파일은 다를 수 있습니다. bpl 및 임의의 패키지 또는 bpl에 의해 요청된 dll은 반드시 배포되어야 합니다.

다음 표는 패키지의 사용 용도에 따라 반드시 필요할 수 있는 파일을 나열한 것입니다.

표 15.4 패키지와 함께 배포되는 파일

파일	설명
ComponentName.h	엔드 유저가 클래스 인터페이스에 액세스할 수 있게 해줍니다.
ComponentName.cpp	엔드 유저가 컴포넌트 소스에 액세스할 수 있게 해줍니다.
bpi, obj 및 lib	엔드 유저가 링크 애플리케이션에 연결할 수 있게 해줍니다.

일반적인 배포 정보에 대한 내용은 17장, "애플리케이션 배포"를 참조하십시오.

패키지를 사용하는 애플리케이션 배포

런타임 패키지를 사용하는 애플리케이션을 배포하는 경우 개발자의 사용자가 애플리케이션이 호출하는 모든 라이브러리(.bpl 또는 .dll) 파일 뿐만 아니라 애플리케이션의 .exe 파일을 갖고 있는지 확인합니다. 라이브러리 파일이 .exe 파일과 다른 디렉토리에 있는 경우 반드시 사용자의 경로를 통해 액세스할 수 있어야 합니다. 라이브러리 파일을 Windows\System 디렉토리에 넣는 규칙을 따르는 경우도 있습니다. InstallShield Express를 사용하면 맹목적으로 다시 설치하기 전에 설치 스크립트가 사용자의 시스템에 필요한 패키지가 있는지 확인합니다.

다른 개발자에게 패키지 배포

다른 C++Builder 개발자에게 런타임 또는 디자인 타임 패키지를 배포하는 경우 필요한 헤더 파일뿐만 아니라 .bpi 및 .bpl 파일을 모두 제공해야 합니다. 컴포넌트를 애플리케이션에 정적으로 연결하려면, 즉 런타임 패키지를 사용하지 않는 애플리케이션을 생성하려면 공급하는 모든 패키지에 대해 .lib 또는 .obj 파일도 필요합니다.

패키지 컬렉션 파일

패키지 컬렉션(.dpc 파일)을 사용하면 편리하게 다른 개발자에게 패키지를 배포할 수 있습니다. 각 패키지 컬렉션에는 bpl 파일 및 bpl 파일과 함께 배포하고자 하는 임의의 추가 파일을 포함하여 하나 이상의 패키지가 포함되어 있습니다. IDE 설치에 대해 패키지 컬렉션을 선택한 경우 이를 구성하는 파일이 자동으로 .pce 컨테이너에서 추출되며 Installation 다이얼로그 박스를 사용하여 컬렉션 내의 모든 패키지를 설치할지 또는 패키지를 선택적으로 설치할지 선택할 수 있습니다.

다음과 같은 방법으로 패키지 컬렉션을 만듭니다.

1. Tools|Package Collection Editor를 선택하여 Package Collection Editor를 엽니다.
2. Add a Package 스피드 버튼을 클릭한 다음 Select Package 다이얼로그 박스에서 bpl을 선택하고 Open을 클릭합니다. 컬렉션에 bpl을 더 추가하려면 Add a Package 스피드 버튼을 다시 클릭하십시오. 패키지 에디터의 왼쪽에 있는 트리 형태에 추가된 bpl이 표시됩니다. 패키지를 제거하려면 제거할 패키지를 선택하고 Remove Package 스피드 버튼을 클릭하십시오.
3. 트리 형태의 맨 위에서 Collection 노드를 선택합니다. Package Collection Editor의 오른쪽에 필드 두 개가 표시됩니다.
   - Author/Vendor Name 에디트 박스에 패키지 컬렉션에 관한 선택 정보를 입력하면 사용자가 패키지를 설치할 때 Installation 다이얼로그 박스에 해당 정보가 표시됩니다.
   - Directory List 아래에 패키지 컬렉션의 파일이 설치될 디폴트 디렉토리가 나열됩니다. Add, Edit 및 Delete 버튼을 사용하여 리스트를 편집합니다. 예를 들어, 모든 소스 코드 파일을 동일한 디렉토리에 복사하려는 경우를 가정합니다. 이런 경우, Directory Name을 Source로, Suggested Path를 C:\MyPackage\Source로 입력할 수 있습니다. 그러면 Installation 다이얼로그 박스가 디렉토리에 대한 제안 경로로 C:\MyPackage\Source를 표시합니다.
4. 패키지 컬렉션에는 bpl 외에도 .bpi, .obj, .cpp(유닛) 파일, 설명서 및 기타 개발자가 함께 배포하고자 하는 파일들이 포함될 수 있습니다. 보조 파일은 특정 패키지(bpl)와 연결된 파일 그룹에 두며 그룹 내의 파일은 연결된 bpl이 설치될 때에만 설치됩니다. 보조 파일을 패키지 컬렉션에 두려면 트리 형태에서 bpl 을 선택하고 Add File Group 스피드 버튼을 클릭한 다음 파일 그룹 이름을 입력하십시오. 파일 그룹을 더 추가하려면 같은 방법을 반복하십시오. 파일 그룹을 선택할 때 새 필드는 Package Collection Editor 의 오른쪽에 표시됩니다.
   - Install Directory 리스트 박스에서 설치하고자 하는 그룹 내 파일이 있는 디렉토리를 선택합니다. 앞의 단계 3에서 Directory List에 입력한 디렉토리가 드롭다운 리스트에 포함됩니다.
   - 그룹 내 파일 설치를 옵션으로 만들려면 Optional Group 체크 박스를 선택합니다.
   - Include Files 아래에 이 그룹에 포함시킬 파일들이 나열됩니다. Add, Delete 및 Auto 버튼을 사용하여 리스트를 편집합니다. Auto 버튼을 사용하면 패키지의 Contains 리스트에 나열된 지정된 확장자를 가진 모든 파일을 선택할 수 있으며 Package Collection Editor는 C++Builder의 전역 Library Path를 사용하여 해당 파일을 찾습니다.
5. 컬렉션에 있는 모든 패키지의 Requires 리스트에 나열된 패키지의 설치 디렉토리를 선택할 수 있습니다. 트리 형태에서 bpl을 선택하면 Package Collection Editor의 오른쪽에 새 필드 네 개가 표시됩니다.
   - Required Executables 리스트 박스에서 Requires 리스트에 나열된 패키지에 대한 .bpl 파일을 설치할 디렉토리를 선택합니다. 앞의 단계 3에서 Directory List에 입력한 디렉토리가 드롭다운 리스트에 포함됩니다. Package Collection Editor는 C++Builder의 전역 Library Path를 사용하여 이러한 파일을 찾아 Required Executable Files에 나열합니다.

- Required Libraries 리스트 박스에서 Requires 리스트에 나열된 패키지에 대한 .obj 및 .bpi 파일을 설치할 디렉토리를 선택합니다. 앞의 단계 3에서 Directory List에 입력한 디렉토리가 드롭다운 리스트에 포함됩니다. Package Collection Editor는 C++Builder의 전역 Library Path를 사용하여 이러한 파일을 찾아 Required Library Files에 나열합니다.

6 패키지 컬렉션 소스 파일을 저장하려면 File|Save를 선택하십시오. 패키지 컬렉션 소스 파일은 반드시 .pce 확장자를 사용하여 저장해야 합니다.

7 패키지 컬렉션을 생성하려면 Compile 스피드 버튼을 클릭하십시오. Package Collection Editor는 소스(.pce) 파일과 같은 이름을 가진 .dpc 파일을 생성합니다. 아직 소스 파일을 저장하지 않은 경우 에디터는 생성하기 전에 개발자에게 파일 이름을 묻습니다

기존 .pce 파일을 편집하거나 다시 생성하려면 Package Collection Editor에서 File|Open을 선택하고 사용할 파일을 찾으십시오.

장

16

국제적인 애플리케이션 생성

이 장에서는 국제적인 시장으로 배포할 애플리케이션을 작성하기 위한 지침을 설명합니다. 미리 계획하면 국내 시장 뿐만 아니라 해외 시장에서 애플리케이션을 작동시키는 데 필요한 시간과 코드를 줄일 수 있습니다.

국제화 및 지역화

해외 시장으로 배포할 수 있는 애플리케이션을 만들려면 다음 두 가지 기본 단계를 수행해야 합니다.

- 국제화
- 지역화

C++Builder 의 에디션에 Translation Tools 가 포함된 경우 이를 사용하여 지역화를 관리할 수 있습니다. 자세한 내용은 Translation Tools의 온라인 도움말(ETM.hlp)을 참조하십시오.

국제화

국제화란 프로그램이 여러 로케일에서 작동할 수 있도록 하는 과정입니다. 로케일은 언어뿐 아니라 대상 국가의 문화적 전통을 비롯한 사용자의 환경을 의미합니다. Windows는 언어와 국가의 쌍으로 기술되는 많은 로케일을 지원합니다.

지역화

지역화란 애플리케이션이 특정 로케일에서 작동할 수 있도록 번역하는 과정입니다. 지역화에는 사용자 인터페이스를 번역하는 것 외에도 기능을 사용자 정의하는 것이 포함됩니다. 예를 들어, 재무 애플리케이션을 다른 국가의 세법에 맞게 수정할 수 있습니다.

애플리케이션 국제화

국제화된 애플리케이션을 만들려면 다음 단계를 완료해야 합니다.

- 코드에서 국제적인 문자 집합의 문자열을 처리할 수 있게 합니다.
- 지역화로 인한 변경 내용을 수용할 수 있도록 사용자 인터페이스를디자인합니다.
- 지역화해야 할 모든 리소스를 분리합니다.

애플리케이션 코드 활성화

애플리케이션의 코드가 다양한 대상 로케일에서 나타나는 문자열을 처리할 수 있도록 해야 합니다.

문자 집합

Windows의 영어권 에디션(영어, 프랑스어, 독일어 포함)에서는 ANSI Latin-1(1252) 문자 집합을 사용합니다. 그러나 Windows의 다른 에디션에서는 다른 문자 집합을 사용합니다. 예를 들어, Windows의 일본어 버전에서는 일본어 문자를 멀티바이트 문자 코드로 나타내는 Shift-JIS 문자 집합(코드 페이지 932)을 사용합니다.

일반적으로 문자 집합에는 다음 세 가지 유형이 있습니다.

- 싱글바이트
- 멀티바이트
- 와이드 문자

Windows와 Linux는 모두 유니코드 뿐만 아니라 싱글바이트 및 멀티바이트 문자 집합을 지원합니다. 싱글바이트 문자 집합에서 문자열의 각 바이트는 문자 하나를 나타냅니다. 영어권 운영 체제에서 사용하는 ANSI 문자 집합은 싱글바이트 문자 집합입니다.

멀티바이트 문자 집합에서, 일부 문자는 1바이트로 표시하고 그 외 문자들은 2바이트 이상으로 표시합니다. 멀티바이트 문자의 첫 바이트는 선행 바이트라고 합니다. 일반적으로 멀티바이트 문자 집합의 하위 128 문자는 7비트의 ASCII 문자에 매핑됩니다. 그리고 순서값이 127 이상인 모든 바이트는 멀티바이트 문자의 선행 바이트입니다. 싱글바이트 문자만 Null 값(#0)을 가질 수 있습니다. 멀티바이트 문자 집합, 특히 더블바이트 문자 집합(DBCS)은 아시아 언어권에서 널리 사용됩니다.

OEM 및 ANSI 문자 집합

Windows 문자 집합(ANSI)과 사용자 시스템의 코드 페이지에서 지정한 문자 집합(OEM 문자 집합) 사이에서 변환해야 할 경우가 있습니다.

멀티바이트 문자 집합

아시아권에서 사용하는 표의 문자 집합에서는 언어의 문자와 1바이트(8비트) *char* 타입 사이의 단순한 1:1 매핑을 사용할 수 없습니다. 표의 언어에는 싱글바이트 *char*를 사용하여 나타내는 문자가 너무 많습니다. 대신 멀티바이트 문자열에는 문자 당 하나 이상의 바이트가 포함될 수 있습니다. AnsiStrings에는 싱글바이트와 멀티바이트 문자가 혼합되어 포함될 수 있습니다.

모든 멀티바이트 문자 코드의 선행 바이트는 특정한 문자 집합에 따라 달라지는 예약된 범위에서 가져옵니다. 두 번째 이후의 바이트는 각각의 1바이트 문자의 문자 코드와 같거나 멀티바이트 문자의 첫 번째 바이트에 예약된 범위에 해당할 수 있습니다. 따라서 문자열의 특정 바이트가 싱글바이트를 나타내는지 또는 멀티바이트 문자의 일부인지 여부를 확인할 수 있는 유일한 방법은 처음부터 문자열을 읽고 예약된 범위에서 선행 바이트가 발생할 경우 2바이트 이상의 문자로 분석하는 것입니다.

아시아 로케일용 코드를 작성할 경우 문자열을 멀티바이트 문자로 분석할 수 있는 함수를 사용하여 모든 문자열 처리를 할 수 있어야 합니다. 멀티바이트 문자와 사용할 수 있는 RTL 함수 리스트는 온라인 도움말의 "International API"를 참조하십시오.

바이트로 표시된 문자열 길이가 문자로 표시된 문자열 길이와 반드시 일치하는 것은 아닙니다. 멀티바이트 문자를 반으로 잘라서 문자열을 자르지 마십시오. 일선에서는 문자 크기를 알 수 없기 때문에 문자를 매개변수로 함수나 프로시저에 전달하지 마십시오. 대신 항상 문자나 문자열에 대한 포인터를 전달합니다.

와이드 문자

표의 문자 집합을 사용하는 또 다른 방법은 모든 문자를 유니코드 같은 와이드 문자 인코딩 구성으로 변환하는 것입니다. 유니코드 문자와 문자열은 와이드 문자 및 와이드 문자열이라고도 합니다. 유니코드 문자 집합에서 각 문자는 2바이트로 표시합니다. 따라서 유니코드 문자열은 개별적인 바이트의 연속이 아니라 2바이트 워드의 연속입니다.

첫 256 유니코드 문자는 ANSI 문자 집합에 매핑됩니다. Windows 운영 체제는 유니코드(UCS-2)를 지원합니다. Linux 운영 체제는 UCS-2의 상위 집합인 UCS-4를 지원합니다. C++Builder는 두 플랫폼 모두에서 UCS-2를 지원합니다. 와이드 문자는 한 바이트가 아니라 두 바이트이므로 문자 집합에서 훨씬 많은 다른 문자들을 표현할 수 있습니다.

와이드 문자 인코딩 구성을 사용하면 MBCS(멀티바이트 문자 집합) 시스템에서는 작동하지 않는 문자열에 대한 일반적인 가정을 만들 수 있는 이점이 있습니다. 문자열의 바이트 수와 문자 수 사이에는 직접적인 관계가 있습니다. 따라서 문자가 반으로 잘리거나 문자의 뒷 부분을 다른 문자의 앞 부분으로 오인하는 실수에 대해서 걱정할 필요가 없습니다.

와이드 문자 사용의 가장 큰 단점은 Windows에서 적은 와이드 문자 API 함수 호출을 지원한다는 점입니다. 이 때문에 VCL 컴포넌트에서는 모든 문자열 값을 싱글바이트나 멀티바이트 문자 집합(MBCS) 문자열로 표시합니다. 와이드 문자 시스템과 MBCS 시스템 사이에서 변환할 경우 문자열 속성을 설정하거나 문자열 속성 값을 읽을 때마다 다른 코드가 필요하고 애플리케이션이 느려집니다. 그러나 문자와 *WideChars* 사이의 1대1 매핑을 이용해야 할 일부 특수한 문자열 처리 알고리즘의 경우 와이드 문자로 변환하고자 할 수 있습니다.

유니코드 문자와 사용할 수 있는 RTL 함수 리스트는 온라인 도움말의 "International API"를 참조하십시오.

애플리케이션에 양방향 기능 포함

일부 언어에서는 영어권 언어에서 일반적인 왼쪽에서 오른쪽으로 읽기 순서를 따르지 않고, 단어는 오른쪽에서 왼쪽으로 읽고 숫자는 왼쪽에서 오른쪽으로 읽습니다. 이러한 구분 때문에 이런 언어를 양방향(BiDi)이라고 합니다. 다른 중동 언어도 양방향이지만 가장 일반적인 양방향 언어는 아랍어와 히브리어입니다.

*TApplication*에는 키보드 레이아웃을 지정할 수 있게 해주는 두 가지 속성 *BiDiKeyboard*와 *NonBiDiKeyboard*가 있습니다. 그리고 VCL에서는 *BiDiMode* 속성과 *ParentBiDiMode* 속성을 통해 양방향 지역화를 지원합니다. 다음 표는 이러한 속성을 가진 VCL 객체를 나열한 것입니다.

표 16.1 BiDi를 지원하는 VCL 객체

컴포넌트 팔레트 페이지	VCL 객체
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar
Additional	TActionMainMenuBar
	TActionToolBar
	TBitBtn
	TCheckListBox
	TColorBox
	TDrawGrid
	TLabeledEdit
	TMaskEdit
	TScrollBox
	TSpeedButton
	TStaticLabel
	TStaticText
	TStringGrid
	TValueListEditor

표 16.1 BiDi를 지원하는 VCL 객체(계속)

컴포넌트 팔레트 페이지	VCL 객체
Win32	TComboBoxEx
	TDateTimePicker
	THeaderControl
	THotKey
	TListView
	TMonthCalendar
	TPageControl
	TRichEdit
	TStatusBar
	TTabControl
	TTreeView
Data Controls	TDBCheckBox
	TDBComboBox
	TDBEdit
	TDBGrid
	TDBListBox
	TDBLookupComboBox
	TDBLookupListBox
	TDBMemo
	TDBRadioGroup
	TDBRichEdit
	TDBText
QReport	TQRDBText
	TQRExpr
	TQRLabel
	TQRMemo
	TQRPreview
	TQRSysData
Other classes	TApplication(*ParentBiDiMode* 없음)
	TBoundLabel
	TControl(*ParentBiDiMode* 없음)
	TCustomHeaderControl(*ParentBiDiMode* 없음)
	TForm
	TFrame
	THeaderSection
	THintWindow(*ParentBiDiMode* 없음)
	TMenu
	TStatusPanel

참고 *THintWindow*에서는 힌트를 활성화한 컨트롤의 *BiDiMode*를 선택합니다.

양방향 속성

16-4페이지의 표 16.1, "BiDi를 지원하는 VCL 객체"에 나열된 객체에는 *BiDiMode* 및 *ParentBiDiMode* 속성이 있습니다. *TApplication*의 *BiDiKeyboard* 및 *NonBiDiKeyboard*와 함께 이 속성에서는 양방향 지역화를 지원합니다.

참고 CLX에서는 크로스 플랫폼 프로그래밍을 위해 양방향 속성을 사용할 수 없습니다.

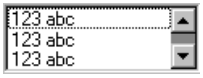
BiDiMode 속성

BiDiMode 속성은 *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign* 및 *bdRightToLeftReadingOnly*의 네 가지 상태를 가진 새로운 열거 타입 *TBiDiMode*입니다.

bdLeftToRight

*bdLeftToRight*에서는 왼쪽에서 오른쪽으로 읽기 순서를 사용하여 텍스트를 그립니다. 정렬이나 스크롤 막대는 바뀌지 않습니다. 예를 들어, 아랍어나 히브리어처럼 오른쪽에서 왼쪽으로 텍스트를 입력할 경우 커서는 푸시 모드가 되고 텍스트는 오른쪽에서 왼쪽으로 입력됩니다. 영어나 프랑스어 같은 라틴 텍스트는 왼쪽에서 오른쪽으로 입력됩니다. *bdLeftToRight*가 기본값입니다.

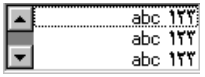
그림 16.1 bdLeftToRight로 설정된 TListBox



bdRightToLeft

*bdRightToLeft*에서는 오른쪽에서 왼쪽으로 읽기 순서를 사용하여 텍스트를 그립니다. 정렬이 바뀌고 스크롤 막대는 이동합니다. 아랍어나 히브리어같이 오른쪽에서 왼쪽으로 읽는 언어에서처럼 텍스트가 입력됩니다. 키보드가 라틴 언어로 바뀌면 커서는 푸시 모드가 되고 텍스트는 왼쪽에서 오른쪽으로 입력됩니다.

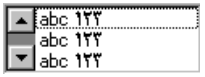
그림 16.2 bdRightToLeft로 설정된 TListBox



bdRightToLeftNoAlign

*bdRightToLeftNoAlign*에서는 오른쪽에서 왼쪽으로 읽기 순서를 사용하여 텍스트를 그립니다. 정렬은 바뀌지 않지만 스크롤 막대는 이동합니다.

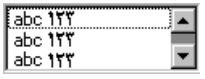
그림 16.3 bdRightToLeftNoAlign으로 설정된 TListBox



bdRightToLeftReadingOnly

*bdRightToLeftReadingOnly*에서는 오른쪽에서 왼쪽으로 읽기 순서를 사용하여 텍스트를 그리고, 정렬과 스크롤 막대가 바뀌지 않습니다.

그림 16.4 bdRightToLeftReadingOnly로 설정된 TListBox



ParentBiDiMode 속성

*ParentBiDiMode*는 부울 속성입니다. **true**(기본값)일 경우 컨트롤은 각각의 부모를 알아내어 사용할 *BiDiMode*를 결정합니다. 컨트롤이 *TForm* 객체일 경우 폼에서는 *Application*의 *BiDiMode* 설정을 사용합니다. 모든 *ParentBiDiMode* 속성이 **true**일 경우 *Application*의 *BiDiMode* 속성을 변경하면 프로젝트의 모든 폼과 컨트롤이 새로운 설정으로 업데이트됩니다.

FlipChildren 메소드

FlipChildren 메소드를 사용하면 컨테이너 컨트롤의 자식 위치를 플립할 수 있습니다. 컨테이너 컨트롤은 *TForm*, *TPanel* 및 *TGroupBox* 같은 다른 컨트롤을 승인할 수 있는 컨트롤입니다. *FlipChildren*에는 하나의 부울 매개변수가 있는데 바로 *AllLevels*입니다. **false**일 경우에는 컨테이너 컨트롤의 직계 자식만 플립합니다. **true**일 경우 컨테이너 컨트롤의 모든 레벨의 자식을 플립합니다.

C++Builder에서는 Left 속성과 컨트롤의 정렬을 변경하여 컨트롤을 플립합니다. 컨트롤의 왼쪽이 부모 컨트롤의 왼쪽 가장자리에서 5 픽셀인 경우 플립하면 컨트롤의 오른쪽은 부모 컨트롤의 오른쪽 가장자리에서 5 픽셀입니다. 편집 컨트롤을 왼쪽으로 정렬한 경우 *FlipChildren*을 호출하면 컨트롤이 오른쪽으로 정렬됩니다.

디자인 타임 시 컨트롤을 플립하려면 Edit|Flip Children을 선택하고 모든 컨트롤을 플립할지 여부에 따라 All 또는 Selected를 선택하거나 선택한 컨트롤의 자식을 선택합니다. 폼에서 컨트롤을 선택하고, 마우스 오른쪽 버튼을 클릭한 다음 컨텍스트 메뉴에서 Flip Children을 선택하여 컨트롤을 플립할 수도 있습니다.

참고 편집 컨트롤을 선택하고 Flip Children|Selected 명령을 선택하면 아무 것도 수행되지 않습니다. 편집 컨트롤은 컨테이너가 아니기 때문입니다.

추가 메소드

양방향 사용자를 위해 애플리케이션을 개발하는 데 유용한 여러 메소드가 있습니다.

메소드	설명
OkToChangeFieldAlignment	데이터베이스 컨트롤과 함께 사용합니다. 컨트롤 정렬을 변경할 수 있는지 확인합니다.
DBUseRightToLeftAlignment	정렬을 확인하기 위한 데이터베이스 컨트롤용 랩퍼입니다.

메소드	설명
ChangeBiDiModeAlignment	전달된 정렬 매개변수를 변경합니다. *BiDiMode* 설정에 대한 확인을 수행하지 않습니다. 왼쪽 정렬을 오른쪽 정렬로 변환하거나 그 반대로 변환하고, center-aligned 컨트롤은 그대로 둡니다.
IsRightToLeft	오른쪽에서 왼쪽으로 옵션을 선택한 경우 **true**를 반환합니다. **false**를 반환한 경우 컨트롤은 왼쪽에서 오른쪽으로 모드입니다.
UseRightToLeftReading	컨트롤에서 오른쪽에서 왼쪽으로 읽기를 사용할 경우 **true**를 반환합니다.
UseRightToLeftAlignment	컨트롤에서 오른쪽에서 왼쪽으로 정렬을 사용할 경우 **true**를 반환합니다. 오버라이드하여 사용자 정의할 수 있습니다.
UseRightToLeftScrollBar	컨트롤에서 왼쪽 스크롤 막대를 사용할 경우 **true**를 반환합니다.
DrawTextBiDiModeFlags	컨트롤의 BiDiMode에 대해 올바른 그리기 텍스트 플래그를 반환합니다.
DrawTextBiDiModeFlagsReadingOnly	컨트롤의 *BiDiMode*에 대해 올바른 그리기 텍스트 플래그를 반환하고 플래그를 해당하는 읽기 순서로 제한합니다.
AddBiDiModeExStyle	해당하는 *ExStyle* 플래그를 만들고 있는 컨트롤에 추가합니다.

로케일 특정 기능

특정 로케일의 애플리케이션에 다른 기능을 추가할 수 있습니다. 특히 아시아 언어 환경에서는 사용자가 입력한 키스트로크를 문자열로 변환하는 데 사용하는 IME(Input Method Editor)를 애플리케이션에서 제어하고자 할 수 있습니다.

VCL 및 CLX 컴포넌트에서는 IME 프로그래밍에 대한 지원을 제공합니다. 텍스트 입력을 직접 사용하는 대부분의 윈도우 컨트롤에는 컨트롤에 입력 포커스가 있을 때 사용해야 할 특정 IME를 지정할 수 있게 해주는 *ImeName* 속성이 있습니다. 그리고 IME에서 키보드 입력을 변환하는 방법을 지정하는 *ImeMode* 속성도 제공합니다. *TWinControl*에서는 사용자가 정의한 클래스에서 IME를 제어하는 데 사용할 수 있는 여러 protected 메소드를 도입합니다. 그리고 전역 *Screen* 변수에서는 사용자 시스템에서 사용할 수 있는 IME에 대한 정보를 제공합니다.

VCL과 CLX에서 사용할 수 있는 전역 *Screen* 변수에서는 사용자 시스템에 설치된 키보드 매핑에 대한 정보도 제공합니다. 이 매개변수를 사용하여 애플리케이션이 실행 중인 환경에 대한 로케일 특정 정보를 가져올 수 있습니다.

사용자 인터페이스 디자인

여러 해외 시장을 위한 애플리케이션을 만들 때는 번역 중 발생하는 변경 내용을 수용할 수 있도록 사용자 인터페이스를 디자인하는 것이 중요합니다.

텍스트

사용자 인터페이스에 나타나는 모든 텍스트를 번역해야 합니다. 영어 텍스트는 항상 번역보다 짧습니다. 문자열이 늘어날 수 있는 공간이 있도록 텍스트를 표시하는 사용자 인터페이스 요소를 디자인합니다. 더 긴 문자열을 쉽게 표시할 수 있도록 텍스트를 표시하는 다이얼로그 박스,

메뉴, 상태 표시줄 및 기타 사용자 인터페이스 요소를 만듭니다. 약어를 사용하지 마십시오. 표의 문자를 사용하는 언어에는 약어가 없습니다.

짧은 문자열은 긴 절보다 번역문에서 더 늘어나는 경향이 있습니다. 표 16.2에서는 영어 문자열 길이를 감안하여 확장할 대략적인 예상값을 제공합니다.

표 16.2 예상 문자열 길이

영어 문자열 길이(문자)	예상되는 증가치
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
50이상	10%

그래픽 이미지

이상적으로는 번역이 필요없는 이미지를 사용하고자 할 것입니다. 이것은 항상 번역이 필요한 텍스트가 그래픽 이미지에 포함되지 말아야함을 의미합니다. 이미지에 텍스트를 포함시켜야 할 경우 이미지 일부로 텍스트를 포함시키는 것보다 이미지에 투명한 배경을 가진 레이블 객체를 사용하는 것이 좋습니다.

그래픽 이미지를 만들 때 다른 점도 고려해야 합니다. 특정 문화에 관련된 이미지를 사용하지 마십시오. 예를 들어 서로 다른 국가의 사서함은 서로 다른 모양을 하고 있습니다. 다른 국교를 갖고 있는 국가를 위해 애플리케이션을 만들 경우에는 신앙에 관련된 기호는 적절하지 않습니다. 또한 색상에도 다른 문화에서 서로 다른 상징적인 의미가 있을 수 있습니다.

형식 및 정렬 순서

애플리케이션에서 사용하는 날짜, 시간, 숫자 및 통화 형식을 대상 로케일에 맞게 지역화해야 합니다. Windows 형식만 사용할 경우 사용자의 Windows 레지스트리에서 가져오기 때문에 형식을 변환할 필요가 없습니다. 그러나 고유한 서식 문자열을 지정한 경우에는 이 서식 문자열을 지역화할 수 있도록 리소스 상수로 선언해야 합니다.

문자열의 정렬 순서도 국가마다 다릅니다. 대부분의 유럽 언어에는 로케일에 따라 다르게 정렬되는 분음 기호가 포함되어 있습니다. 그리고 일부 국가에서는 두 문자 조합을 정렬 순서에서 한 문자로 취급하기도 합니다. 예를 들어, 스페인에서 *ch* 조합은 *c*와 *d* 사이의 고유한 단일 문자로 정렬됩니다. 때로는 독일어 *eszett*처럼 단일 문자가 두 개의 독립적인 문자처럼 정렬되기도 합니다.

키보드 매핑

키 조합 단축키 지정에 주의하십시오. US 키보드에서 사용할 수 있는 모든 문자를 모든 국제적인 키보드에서 쉽게 재현할 수 있는 것은 아닙니다. 가능하다면 단축키에 숫자 키와 기능 키를 사용하십시오. 이 키들은 실제로 모든 키보드에서 사용할 수 있기 때문입니다.

리소스 분리

애플리케이션을 지역화하는 데 있어서 가장 눈에 띄는 작업은 사용자 인터페이스의 문자열을 번역하는 것입니다. 코드를 변경하지 않고 번역할 수 있는 애플리케이션을 만들려면 사용자

인터페이스의 문자열을 단일 모듈로 분리합니다. C++Builder에서는 메뉴, 다이얼로그 박스 및 비트맵의 리소스가 들어있는 .dfm (CLX 애플리케이션에서는 .xfm) 파일을 자동으로 만듭니다.

이렇게 명백한 사용자 인터페이스 요소 외에도 사용자에게 제공하는 오류 메시지 같은 문자열도 분리해야 합니다. 문자열 리소스는 폼 파일에 포함되지 않지만 RC 파일로 분리할 수 있습니다.

리소스 DLL 생성

리소스를 분리하면 번역 과정이 간단해집니다. 리소스 분리의 다음 단계는 리소스 DLL을 만드는 것입니다. 리소스 DLL에는 모든 리소스와 프로그램의 리소스만 포함됩니다. 리소스 DLL을 사용하면 리소스 DLL을 스와핑하여 많은 번역을 지원하는 프로그램을 만들 수 있습니다.

프로그램을 위한 리소스 DLL을 만들려면 Resource DLL 마법사를 사용합니다. Resource DLL 마법사에는 열리고, 저장되고, 컴파일된 프로젝트가 필요합니다. 이 마법사에서는 RC 파일에서 사용하는 문자열 테이블과 프로젝트의 **resourcestring** 문자열이 포함된 RC 파일을 만들고, 관련된 폼과 만들어진 RES 파일이 포함된 리소스 전용 DLL을 위한 프로젝트를 만듭니다. RES 파일은 새로운 RC 파일로 컴파일됩니다.

지원할 번역에 대한 리소스 DLL을 만들어야 합니다. 모든 리소스 DLL은 대상 로케일에 관련된 파일 이름 확장자를 갖고 있어야 합니다. 처음 두 문자는 대상 언어를 가리키고, 세 번째 문자는 로케일 국가를 가리킵니다. Resource DLL 마법사를 사용할 경우 마법사에서 대신 처리해줍니다. 그렇지 않으면 다음 코드를 사용하여 대상 번역의 로케일 코드를 가져옵니다.

```
/* This callback fills a listbox with the strings and their associated
languages and countries*/
BOOL __stdcall EnumLocalesProc(char* lpLocaleString)
{
  AnsiString LocaleName, LanguageName, CountryName;
  LCID lcid;
  lcid = StrToInt("$" + AnsiString(lpLocaleString));
  LocaleName = GetLocaleStr(lcid, LOCALE_SABBREVLANGNAME, "");
  LanguageName = GetLocaleStr(lcid, LOCALE_SNATIVELANGNAME, "");
  CountryName = GetLocaleStr(lcid, LOCALE_SNATIVECTRYNAME, "");
  if (lstrlen(LocaleName.c_str()) > 0)
    Form1->ListBox1->Items->Add(LocaleName + ":" + LanguageName + "-" +
CountryName);
  return TRUE;
}
/* This call causes the callback to execute for every locale */
  EnumSystemLocales((LOCALE_ENUMPROC)EnumLocalesProc, LCID_SUPPORTED);
```

리소스 DLL 사용

실행 파일, DLL 및 애플리케이션을 구성하는 패키지(bpl)에는 필요한 모든 리소스가 포함되어 있습니다. 그러나 이러한 리소스를 지역화된 버전으로 바꾸려면 실행 파일, DLL 또는 패키지 파일과 같은 이름을 가진 지역화된 리소스 DLL과 함께 애플리케이션을 제공해야 합니다.

애플리케이션을 시작하면 로컬 시스템의 로케일을 확인합니다. 사용하고 있는 EXE, DLL 또는 BPL 파일과 같은 이름을 가진 리소스 DLL을 찾으면 해당 DLL의 확장자를 확인합니다. 리소스 모듈의 확장자가 시스템 로케일의 언어 및 국가와 일치하면 애플리케이션에서는 실행 파일,

DLL 또는 패키지의 리소스 대신 해당 리소스 모듈의 리소스를 사용합니다. 언어 및 국가 모두와 일치하는 리소스 모듈이 없으면 애플리케이션에서는 언어와 일치하는 리소스 모듈을 찾습니다. 언어와 일치하는 리소스 모듈이 없으면 애플리케이션에서는 실행 파일, DLL 또는 패키지와 같이 컴파일한 리소스를 사용합니다.

애플리케이션에서 로컬 시스템의 로케일과 일치하는 리소스 모듈 대신 다른 리소스 모듈을 사용하게 하려면 Windows 레지스트리에 다른 로케일 오버라이드 항목을 설정할 수 있습니다. HKEY_CURRENT_USER\Software\Borland\Locales key에서 애플리케이션의 경로와 파일 이름을 문자열 값으로 추가하고 데이터 값을 리소스 DLL의 확장자로 설정합니다. 시작할 때 애플리케이션에서 시스템 로케일을 사용하기 전에 이 확장자를 가진 리소스 DLL을 찾습니다. 이 레지스트리 항목을 설정하면 시스템의 로케일을 변경하지 않고 지역화된 버전의 애플리케이션을 테스트할 수 있습니다.

예를 들어, 설치 또는 설정 프로그램에서 다음 프로시저를 사용하여 C++Builder 애플리케이션을 로드할 때 사용할 로케일을 가리키는 레지스트리 키 값을 설정할 수 있습니다.

```
void SetLocalOverrides(char* FileName, char* LocaleOverride)
{
  HKEY Key;
  const char* LocaleOverrideKey = "Software\\Borland\\Locales";
  if (RegOpenKeyEx(HKEY_CURRENT_USER, LocaleOverrideKey, 0,
KEY_ALL_ACCESS, &Key)
        == ERROR_SUCCESS) {
    if (lstrlen(LocaleOverride) == 3)
      RegSetValueEx(Key, FileName, 0, REG_SZ, (const
BYTE*)LocaleOverride, 4);
    RegCloseKey(Key);
  }
}
```

애플리케이션에서 전역 *FindResourceHInstance* 함수를 사용하여 현재 리소스 모듈의 핸들을 가져옵니다. 예를 들면, 다음과 같습니다.

```
LoadString(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery,
sizeof(szQuery));
```

적절한 리소스 DLL만 제공하면 애플리케이션이 실행 중인 시스템의 로케일에 맞게 자동으로 적용하는 단일 애플리케이션을 만들 수 있습니다.

리소스 DLL의 동적 전환

애플리케이션을 시작할 때 리소스 DLL을 찾는 것 외에도 런타임 시 리소스 DLL을 동적으로 전환할 수 있습니다. 이 기능을 애플리케이션을 추가하려면 프로젝트에 ReInit 유닛을 포함시켜야 합니다. ReInit는 Examples 디렉토리의 Richedit 샘플에 있습니다. 언어를 전환하려면 *LoadResourceModule*을 호출하여 새 언어의 LCID를 전달한 다음 *ReinitializeForms*를 호출합니다.

예를 들어, 다음 코드에서는 인터페이스 언어를 프랑스어로 전환합니다.

```
const  FRENCH = (SUBLANG_FRENCH << 10) | LANG_FRENCH;
if (LoadNewResourceModule(FRENCH))
  ReinitializeForms();
```

이 기술의 장점은 애플리케이션의 현재 인스턴스와 모든 폼을 사용할 수 있다는 점입니다. 데이터베이스 서버에 로그인하는 것처럼 레지스트리 설정을 업데이트하고 애플리케이션을 다시 시작하거나 애플리케이션에서 요구하는 리소스를 다시 가져올 필요가 없습니다.

리소스 DLL을 전환하면 새로운 DLL에서 지정한 속성이 폼의 실행 중인 인스턴스의 속성을 덮어 씁니다.

참고 런타임 시의 폼 속성에 대한 변경 사항은 손실됩니다. 일단 새로운 DLL을 로드하면 기본값은 다시 설정되지 않습니다. 지역화로 인한 차이는 별도로 하고, 폼 객체를 시작 상태로 다시 초기화하는 것으로 가정하는 코드는 사용하지 마십시오.

애플리케이션 지역화

애플리케이션을 국제화하면 애플리케이션을 배포할 다른 해외 시장을 위한 지역화된 버전을 만들 수 있습니다.

리소스 지역화

이상적으로 폼 파일(VCL의 .dfm 또는 CLX의 .xfm)과 리소스 파일이 포함된 리소스 DLL로 리소스를 분리했습니다. IDE에서 폼을 열고 관련된 속성을 변환할 수 있습니다.

참고 리소스 DLL 프로젝트에서 컴포넌트를 추가하거나 삭제할 수 있습니다. 그러나 런타임 오류가 발생할 수 있는 방법으로 속성을 변경할 수 있기 때문에 번역이 필요한 속성만 수정하도록 주의하십시오. 실수를 방지하기 위해 Object Inspector에서 Localizable 속성만 표시하도록 구성할 수 있습니다. 그렇게 하려면 Object Inspector에서 마우스 오른쪽 버튼을 클릭하고 View 메뉴를 사용하여 원하지 않는 속성 분류는 필터링합니다.

RC 파일을 열고 관련된 문자열을 번역할 수 있습니다. Project Manager에서 RC 파일을 열어 StringTable Editor를 사용합니다.

장

17

애플리케이션 배포

C++Builder 애플리케이션을 구축하고 실행한 후에는 배포할 수 있습니다. 즉 다른 사용자가 애플리케이션을 설치해서 실행하게 할 수 있습니다. 애플리케이션을 다른 컴퓨터에 배포하여 애플리케이션이 완전한 기능을 수행하게 하려면 여러 단계를 거쳐야 합니다. 애플리케이션의 종류에 따라 필요한 단계가 각각 다릅니다. 다음 단원에서는 여러 가지 타입의 애플리케이션을 배포할 때 고려해야 할 사항에 대해 설명합니다.

- 일반적인 애플리케이션 배포
- CLX 애플리케이션 배포
- 데이터베이스 애플리케이션 배포
- 웹 애플리케이션 배포
- 다양한 호스트 환경을 위한 프로그래밍
- 소프트웨어 사용권 요구 사항

참고 이 단원에서는 Windows에 애플리케이션을 배포하는 데 대한 정보를 제공합니다.

일반적인 애플리케이션 배포

실행 파일 외에도 애플리케이션에는 DLL, 패키지 파일 및 helper 애플리케이션과 같은 많은 지원 파일이 필요할 수 있습니다. 또한 Windows 레지스트리에는 지원 파일의 위치 지정부터 간단한 프로그램 설정에 이르기까지 애플리케이션에 대한 항목이 포함되어야 합니다. 애플리케이션의 파일을 컴퓨터에 복사하는 과정과 필요한 레지스트리 설정 값을 만드는 과정을 InstallShield Express와 같은 설치 프로그램을 사용하여 자동화할 수 있습니다. 다음은 거의 모든 타입의 애플리케이션에 공통적인 주요 배포 관련 사항입니다.

- 설치 프로그램 사용
- 애플리케이션 파일 식별

데이터베이스에 액세스하는 C++Builder 애플리케이션과 웹에서 실행되는 애플리케이션은 일반적인 애플리케이션에 적용되는 단계 외에 추가 설치 단계가 필요합니다. 데이터베이스 애플리케이션 설치에 대한 자세한 내용은 17-6페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오. 웹 애플리케이션 설치에 대한 자세한 내용은 17-10페이지의 "웹 애플리케이션 배포"를 참조하십시오. ActiveX 컨트롤 설치에 대한 자세한 내용은 43-16페이지의 "ActiveX 컨트롤을 웹에 배포"를 참조하십시오. CORBA 애플리케이션 배포에 대한 자세한 내용은 *VisiBroker Installation and Administration Guide*를 참조하십시오.

설치 프로그램 사용

실행 파일로만 구성된 간단한 C++Builder 애플리케이션은 대상 컴퓨터에 쉽게 설치할 수 있습니다. 실행 파일을 컴퓨터에 복사하기만 하면 됩니다. 그러나 여러 파일이 포함된 더 복잡한 애플리케이션은 추가 설치 과정이 필요합니다 . 이러한 애플리케이션에는 전용 설치 프로그램이 필요합니다.

설치 도구 키트는 설치 프로그램 작성 과정을 자동화하며 코드를 전혀 작성할 필요가 없는 경우도 많습니다. 설치 도구 키트로 작성된 설치 프로그램은 C++Builder 애플리케이션을 설치하는 데 기본적으로 필요한 여러 가지 작업을 수행합니다. 여기에는 실행 파일과 지원 파일을 호스트 컴퓨터로 복사하고, Windows 레지스트리 항목을 만들고, BDE 데이터베이스 애플리케이션을 위한 Borland Database Engine을 설치하는 작업도 포함됩니다.

InstallShield Express는 C++Builder에 포함된 설치 도구 키트입니다. InstallShield Express는 C++Builder 및 Borland Database Engine과 함께 사용할 수 있도록 인증되었습니다. InstallShield Express는 Windows Installer(MSI) 기술을 기반으로 합니다.

InstallShield Express는 C++Builder를 설치할 때 자동으로 설치되지 않으므로 이 프로그램을 사용하여 설치 프로그램을 만들려면 수동으로 설치해야 합니다. C++Builder CD에서 설치 프로그램을 실행하여 InstallShield Express 를 설치합니다 . InstallShield Express 를 사용하여 설치 프로그램을 만드는 데 대한 자세한 내용은 InstallShield Express 온라인 도움말을 참조하십시오.

다른 설치 도구 키트도 사용할 수 있습니다. 그러나 BDE 데이터베이스 애플리케이션을 배포하는 경우에는 MSI 기술과 Borland Database Engine 배포를 위해 인증된 기술을 기반으로 하는 도구 키트만 사용해야 합니다.

애플리케이션 파일 식별

실행 파일 외에도 많은 파일이 애플리케이션과 함께 배포되어야 합니다.

- 애플리케이션 파일
- 패키지 파일
- 병합 모듈
- ActiveX 컨트롤

애플리케이션 파일

다음 타입의 파일이 애플리케이션과 함께 배포되어야 합니다.

표 17.1 애플리케이션 파일

타입	파일 이름 확장자
프로그램 파일	.exe 및 .dll
패키지 파일	.bpl 및 .bcp
도움말 파일	.hlp, .cnt 및 .toc(사용되는 경우) 또는 애플리케이션에서 지원하는 다른 모든 도움말 파일
ActiveX 파일	.ocx(종종 DLL에서 지원)
로컬 테이블 파일	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

패키지 파일

애플리케이션이 런타임 패키지를 사용하는 경우 패키지 파일은 애플리케이션과 함께 배포되어야 합니다. InstallShield Express는 패키지 파일의 설치를 DLL과 같은 방법으로 처리하여 파일을 복사하고 Windows 레지스트리에 필요한 항목을 만듭니다. 병합 모듈을 사용하여 InstallShield Express를 포함하는 런타임 패키지와 MSI 기반의 설치 도구를 배포할 수도 있습니다. 자세한 내용은 다음 단원을 참조하십시오.

Borland에서 제공한 런타임 패키지 파일은 Windows\System 디렉토리에 설치하는 것이 좋습니다. 이 디렉토리는 여러 애플리케이션이 파일의 단일 인스턴스에 액세스할 수 있는 공통 위치로 사용됩니다. 직접 만든 패키지의 경우 애플리케이션과 같은 디렉토리에 설치하는 것이 좋습니다. .bpl 파일만 배포하면 됩니다.

참고 CLX 애플리케이션과 패키지를 배포하는 경우에는 vcl60.bpl 대신 clx60.bpl을 포함해야 합니다.

패키지를 다른 개발자에게 배포하는 경우에는 .bpl과 .bcp 파일을 모두 제공합니다.

병합 모듈

InstallShield Express 3.0은 Windows Installer(MSI) 기술을 기반으로 합니다. 그렇기 때문에 C++Builder에는 병합 모듈이 포함되어 있습니다. 병합 모듈은 공유 코드, 파일, 리소스, 레지스트리 항목 및 설치 로직을 단일 복합 파일로 애플리케이션에 전달하는 데 사용할 수 있는 표준 방법을 제공합니다. 병합 모듈을 사용하여 InstallShield Express와 같은 MSI 기반의 설치 도구로 런타임 패키지를 배포할 수 있습니다.

런타임 라이브러리는 그룹화되는 방식으로 인해 약간의 상호 종속 관계를 갖고 있습니다. 그 결과, 한 패키지가 설치 프로젝트에 추가되고 설치 도구가 자동으로 하나 이상의 다른 패키지에 대한 종속 관계를 추가하거나 보고합니다. 예를 들어, VCLInternet 병합 모듈을 설치 프로젝트에 추가하면 설치 도구가 VCLDatabase 및 StandardVCL 모듈에 대한 종속 관계도 자동으로 추가하거나 보고해야 합니다.

다음 표는 각 병합 모듈의 종속 관계를 나열한 것입니다. 설치 도구에 따라 이러한 종속 관계에 대해 다르게 반응합니다. Windows Installer용 InstallShield는 필요한 모듈을 찾을 수 있는 경우 자동으로 이 모듈을 추가합니다. 다른 도구들은 단순히 종속 관계를 보고하거나 필요한 모듈이 프로젝트에 모두 포함되어 있지 않은 경우 생성 오류를 발생시킵니다.

표 17.2 병합 모듈 및 종속 관계

병합 모듈	포함된 BPL	종속 관계
ADO	adortl60.bpl	DatabaseRTL, BaseRTL
BaseClientDataSet	cds60.bpl	DatabaseRTL, BaseRTL, DataSnap, dbExpress
BaseRTL	rtl60.bpl	종속 관계 없음
BaseVCL	vcl60.bpl, vclx60.bpl	BaseRTL
BDEClientDataSet	bdecds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress, BDERTL
BDEInternet	inetdbbde60.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl60.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl60.bpl	BaseRTL
DatabaseVCL	vcldb60.bpl	BaseVCL, BaseRTL
DataSnap	dsnap60.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsnapcon60.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapCorba	dsnapcrba60.bpl	DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DataSnapEntera	dsnapent60.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx60.bpl	DatabaseVCL, BaseVCL, BaseRTL, DatabaseRTL
dbExpress	dbexpress60.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress60.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss60.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
FastNet	nmfast60.bpl	BaseVCL, BaseRTL
InterbaseVCL	ibxpress60.bpl	BaseClientDataSet, BaseRTL, BaseVCL, DatabaseRTL, DatabaseVCL, DataSnap, dbExpress
Internet	inet60.bpl, inetdb60.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy60.bpl	BaseVCL, BaseRTL
Office2000Components	dcloffice2k60.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
QuickReport	qrpt60.bpl	BaseVCL, BaseRTL, BDERTL, DatabaseRTL
SampleVCL	vclsmp60.bpl	BaseVCL, BaseRTL
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	BaseVCL, BaseRTL
VCLIE	vclie60.bpl	BaseVCL, BaseRTL
VisualCLX	visualclx60.bpl	BaseRTL

표 17.2 병합 모듈 및 종속 관계 (계속)

병합 모듈	포함된 BPL	종속 관계
VisualDBCLX	visualdbclx60.bpl	BaseRTL, DatabaseRTL, VisualCLX
WebDataSnap	webdsnap60.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL
WebSnap	websnap60.bpl, vcljpg60.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmlrtl60.bpl	Internet, DatabaseRTL, BaseRTL

ActiveX 컨트롤

C++Builder에 포함된 컴포넌트 중에는 ActiveX 컨트롤이 있습니다. 컴포넌트 랩퍼는 애플리케이션의 실행 파일(또는 런타임 패키지)에 연결되어 있지만 컴포넌트의 .ocx 파일도 애플리케이션과 함께 배포해야 합니다. 이러한 컴포넌트에는 다음 종류가 있습니다.

- Chart FX, copyright SoftwareFX Inc.
- VisualSpeller Control, copyright Visual Components, Inc.
- Formula One(스프레드시트), copyright Visual Components, Inc.
- First Impression(VtChart), copyright Visual Components, Inc.
- Graph Custom Control, copyright Bits Per Second Ltd.

직접 만든 ActiveX 컨트롤을 사용하려면 먼저 배포용 컴퓨터에 등록해야 합니다. InstallShield Express와 같은 설치 프로그램을 사용하면 이 등록 과정이 자동으로 진행됩니다. ActiveX 컨트롤을 수동으로 등록하려면 IDE에서 Run|ActiveX Server를 선택하고 TRegSvr 데모 애플리케이션을 사용하거나 Microsoft 유틸리티 REGSRV32.EXE(Windows 9x 버전에 포함되지 않음)를 사용하십시오.

ActiveX 컨트롤을 지원하는 DLL도 애플리케이션과 함께 배포해야 합니다.

Helper 애플리케이션

Helper 애플리케이션이란 이 애플리케이션이 없으면 C++Builder 애플리케이션이 부분적으로 또는 전혀 작동되지 않는 독립적인 프로그램입니다. Helper 애플리케이션은 Borland에서 운영 체제와 함께 제공되거나 서드파티 제품일 수 있습니다. Helper 애플리케이션의 예로는 InterBase 데이터베이스, 사용자 및 보안을 관리하는 InterBase 유틸리티 프로그램인 Server Manager가 있습니다.

애플리케이션이 Helper 프로그램에 종속된 경우에는 가능한 한 애플리케이션과 함께 배포해야 합니다. Helper 프로그램 배포는 재배포 사용권 계약에 의해 제한될 수 있습니다. 자세한 내용은 Helper 프로그램 설명서를 참조하십시오.

DLL 위치

한 애플리케이션에 의해서만 사용되는 DLL 파일은 애플리케이션과 같은 디렉토리에 설치할 수 있습니다. 여러 애플리케이션에서 사용할 DLL은 이러한 애플리케이션이 모두 액세스할 수 있는 위치에 설치해야 합니다. 일반적으로 이러한 커뮤니티 DLL은 Windows 또는 Windows\System 디렉토리에 둡니다. 또는 Borland Database Engine을 설치하는 것과 비슷한 방법으로 공통 .DLL 파일의 전용 디렉토리를 만드는 것도 더 좋은 방법입니다.

CLX 애플리케이션 배포

Windows에 CLX 애플리케이션을 배포하려면 일반적인 애플리케이션과 같은 단계를 따르십시오. 일반적인 애플리케이션 설치에 대한 자세한 내용은 17-1페이지의 "일반적인 애플리케이션 배포"를 참조하십시오. 데이터베이스 CLX 애플리케이션 설치에 대한 자세한 내용은 17-6페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

CLX 런타임을 포함하려면 qtintf.dll을 애플리케이션에 포함시켜야 합니다. CLX 애플리케이션과 함께 패키지를 배포하는 경우에는 vcl60.bpl 대신 clx60.bpl을 포함시켜야 합니다.

CLX 애플리케이션 생성에 대한 자세한 내용은 14장, "크로스 플랫폼 애플리케이션 개발"을 참조하십시오.

데이터베이스 애플리케이션 배포

데이터베이스에 액세스하는 애플리케이션을 설치할 때는 애플리케이션의 실행 파일을 호스트 컴퓨터에 복사하는 것 외에 특별히 고려해야 할 사항이 있습니다.데이터베이스 액세스는 각각의 데이터베이스 엔진에 의해 처리되는 경우가 가장 많으며 이러한 엔진의 파일은 애플리케이션의 실행 파일과 연결될 수 없습니다. 데이터 파일이 미리 생성되지 않은 경우에는 애플리케이션에서 사용할 수 있게 해야 합니다. 애플리케이션을 구성하는 파일은 보통 여러 컴퓨터에 설치되기 때문에 멀티 티어 데이터베이스 애플리케이션을 설치할 때는 훨씬 특화된 처리가 필요합니다.

여러 가지 다른 데이터베이스 기술(ADO, BDE, dbExpress 및 InterBase Express)이 지원되므로 각 기술마다 배포 요구 사항이 다릅니다. 어떤 기술을 사용하든지 클라이언트사이드 소프트웨어를 데이터베이스 애플리케이션을 실행할 시스템에 설치해야 합니다. ADO, BDE, dbExpress 및 InterBase Express가 데이터베이스의 클라이언트사이드 소프트웨어와 상호 작용하기 위해서는 드라이버가 필요합니다.

dbExpress, BDE 및 멀티 티어 데이터베이스 애플리케이션 배포 방법에 대한 자세한 내용은 다음 단원에서 설명합니다.

- dbExpress 데이터베이스 애플리케이션 배포
- BDE 애플리케이션 배포
- 멀티 티어 데이터베이스 애플리케이션(DataSnap) 배포

*TClientDataSet*과 같은 클라이언트 데이터셋이나 데이터셋 프로바이더를 사용하는 데이터베이스 애플리케이션을 사용하려면 midaslib.dcu 및 crtl.dcu(독립 실행형 실행 파일을 제공할 때의 정적 연결을 위해)를 포함해야 합니다. 실행 파일 및 필요한 모든 DLL과 함께 애플리케이션을 패키지화하는 경우에는 Midas.dll을 포함해야 합니다.

ADO 를 사용하는 데이터베이스 애플리케이션을 배포할 경우에는 애플리케이션을 실행할 시스템에 MDAC 버전 2.1 이상이 설치되어 있는지 확인해야 합니다. MDAC는 Windows 2000 및 Internet Explorer 버전 5 이상과 같은 소프트웨어와 함께 자동으로 설치됩니다. 연결할 데이터베이스 서버용 드라이버가 클라이언트에 설치되어 있는지도 확인해야 합니다. 다른 배포 단계는 필요하지 않습니다.

InterBase Express를 사용하는 데이터베이스 애플리케이션을 배포할 경우에는 애플리케이션을 실행할 시스템에 InterBase 클라이언트가 설치되어 있는지 확인해야 합니다. InterBase를 사용하려면 gd32.dll 및 interbase.msg가 액세스 가능한 디렉토리에 있어야 합니다. 다른 배포 단계는 필요하지 않습니다. InterBase Express 컴포넌트는 InterBase Client API와 직접 통신하므로 추가 드라이버가 필요하지 않습니다. 자세한 내용은 Borland 웹 사이트에 포스트된 Embedded Installation Guide를 참조하십시오.

여기서 설명된 기술 외에도, 서드파티 데이터베이스 엔진을 사용하여 C++Builder 애플리케이션에 데이터베이스 액세스를 제공할 수 있습니다. 재배포 권한, 설치 및 구성에 대한 자세한 내용은 해당 데이터베이스 엔진의 업체나 설명서를 참조하십시오.

dbExpress 데이터베이스 애플리케이션 배포

dbExpress는 데이터베이스 정보에 대해 빠른 액세스를 제공하는 씬 원시 드라이버 집합입니다. dbExpress는 Linux에서도 사용할 수 있으므로 크로스 플랫폼 개발을 지원합니다. dbExpress 컴포넌트 사용에 대한 자세한 내용은 26장, "단방향 데이터셋 사용"을 참조하십시오.

dbExpress 애플리케이션은 독립 실행형 실행 파일이나, 관련 dbExpress 드라이버 DLL을 포함하는 실행 파일로 배포할 수 있습니다.

dbExpress 애플리케이션을 독립 실행형 실행 파일로 배포하려면 dbExpress 객체 파일이 정적으로 실행 파일에 연결되어 있어야 합니다. 이 작업을 수행하려면 lib 디렉토리에 있는 다음 DCU를 포함시키십시오.

표 17.3 독립 실행형 실행 파일로 dbExpress 배포

데이터베이스 단위	포함할 시기
dbExpINT	InterBase 데이터베이스에 연결하는 애플리케이션
dbExpORA	Oracle 데이터베이스에 연결하는 애플리케이션
dbExpDB2	DB2 데이터베이스에 연결하는 애플리케이션
dbExpMYS	MySQL 3.22.x 데이터베이스에 연결하는 애플리케이션
dbExpMYSQL	MySQL 3.23.x 데이터베이스에 연결하는 애플리케이션
crtl	dbExpress를 사용하는 모든 실행 파일에 필요
MidasLib	*TClientDataSet*과 같은 클라이언트 데이터셋을 사용하는 dbExpress 실행 파일에 필요

참고 Informix를 사용하는 데이터베이스 애플리케이션은 독립 실행형으로 배포할 수 없습니다. 대신 다음 표에 나열된 드라이버 DLL dbexpinf.dll과 함께 실행 파일로 배포하십시오.

독립 실행형으로 배포하지 않을 경우에는 관련 dbExpress 드라이버와 DataSnap DLL을 실행 파일과 배포할 수 있습니다. 다음 표는 해당 DLL과 배포 시기를 나열한 것입니다.

표 17.4 드라이버 DLL과 dbExpress 배포

데이터베이스 DLL	배포 시기
dbexpinf.dll	Informix 데이터베이스에 연결하는 애플리케이션
dbexpint.dll	InterBase 데이터베이스에 연결하는 애플리케이션
dbexpora.dll	Oracle 데이터베이스에 연결하는 애플리케이션
dbexpdb2.dll.	DB2 데이터베이스에 연결하는 애플리케이션
dbexpmys.dll	MySQL 3.22.x 데이터베이스에 연결하는 애플리케이션
dbexpmysql.dll	MySQL 3.23.x 데이터베이스에 연결하는 애플리케이션
Midas.dll	클라이언트 데이터셋을 사용하는 데이터베이스 애플리케이션에 필요

BDE 애플리케이션 배포

Borland Database Engine(BDE)은 데이터베이스와 상호 작용하기 위한 대형 API를 정의합니다. 모든 데이터 액세스 메커니즘 중에서 BDE는 가장 광범위한 함수를 지원하며 가장 많은 지원 유틸리티를 제공합니다 . Paradox 또는 dBASE 테이블에 있는 데이터 작업을 할 때 가장 적합한 방법입니다.

애플리케이션의 데이터베이스 액세스는 다양한 데이터베이스 엔진에 의해 제공됩니다. 애플리케이션은 BDE 또는 서드파티 데이터베이스 엔진을 사용할 수 있습니다. SQL 데이터베이스 시스템에 대한 원시 액세스를 사용할 수 있도록 SQL Links가 제공되지만, 모든 에디션에서 사용할 수 있는 것은 아닙니다. 다음 단원에서는 애플리케이션의 데이터베이스 액세스 요소 설치에 대해 설명합니다.

- Borland Database Engine
- SQL Links

Borland Database Engine

표준 C++Builder 데이터 컴포넌트가 데이터베이스에 액세스할 수 있게 하려면 Borland Database Engine(BDE)이 표시되고 액세스 가능해야 합니다. BDE 재배포에 대한 권한과 제한 사항에 대해서는 BDEDEPLOY 설명서를 참조하십시오.

BDE를 설치할 때는 InstallShield Express나 기타 인증된 설치 프로그램을 사용하는 것이 좋습니다. InstallShield Express는 필요한 레지스트리 항목을 작성하고 애플리케이션에 필요한 모든 알리아스를 정의합니다. 인증된 설치 프로그램을 사용하여 BDE 파일과 기타 하위 파일을 배포하는 것이 중요한 이유는 다음과 같습니다.

- BDE 또는 BDE 하위 파일을 제대로 설치하지 않으면 BDE를 사용하는 다른 애플리케이션에 오류가 발생할 수 있습니다. 이러한 애플리케이션에는 Borland 제품뿐 아니라 BDE를 사용하는 많은 서드파티 프로그램이 포함되어 있습니다.
- 32비트 Windows 95/NT 이상에서 BDE 구성 정보는 16비트 Windows에서처럼 .ini 파일에 저장되는 것이 아니라 Windows 레지스트리에 저장됩니다. 설치 및 제거를 위해 정확한 항목을 추가하거나 삭제하는 것은 복잡한 작업입니다.

애플리케이션에 실제로 필요한 만큼만 BDE를 설치할 수 있습니다. 예를 들어, 애플리케이션에 Paradox 테이블이 필요하면 BDE에서 Paradox 테이블에 액세스하는 데 필요한 부분만 설치해야 합니다. 따라서 애플리케이션에 필요한 디스크 공간을 줄일 수 있습니다. InstallShield Express와 같은 인증된 설치 프로그램은 부분적인 BDE 설치를 수행할 수 있습니다. 배포된 애플리케이션에서는 사용되지 않지만 다른 프로그램에 필요한 BDE 시스템 파일은 남겨두어야 합니다.

SQL Links

SQL Links는 Borland Database Engine을 통해 애플리케이션과 SQL 데이터베이스용 클라이언트 소프트웨어를 연결하는 드라이버를 제공합니다. SQL Links 재배포에 대한 권한과 제한 사항은 DEPLOY 설명서를 참조하십시오. Borland Database Engine(BDE)에서처럼 SQL Links는 InstallShield Express나 기타 인증된 설치 프로그램을 사용하여 배포해야 합니다.

참고 SQL Links는 BDE를 클라이언트 소프트웨어에만 연결할 뿐, SQL 데이터베이스 자체에는 연결하지 않습니다. 그러나 사용되는 SQL 데이터베이스 시스템을 위해 클라이언트 소프트웨어를 설치해야 합니다. 클라이언트 소프트웨어 설치 및 구성에 대한 자세한 내용은 SQL 데이터베이스 설명서나 SQL 데이터베이스를 제공하는 공급업체에 문의하십시오.

표 17.5는 SQL Links에서 다양한 SQL 데이터베이스 시스템에 연결하는 데 사용하는 드라이버와 설정 파일 이름을 나열한 것입니다. 이러한 파일은 SQL Links와 함께 제공하며 C++Builder 사용권 계약에 따라 재배포됩니다.

표 17.5 SQL 데이터베이스 클라이언트 소프트웨어 파일

공급업체	재배포 가능한 파일
Oracle 7	SQLORA32.DLL 및 SQL_ORA.CNF
Oracle 8	SQLORA8.DLL 및 SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL 및 SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL 및 SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL 및 SQL_MSS.CNF
Informix 7	SQLINF32.DLL 및 SQL_INF.CNF
Informix 9	SQLINF9.DLL 및 SQL_INF9.CNF
DB/2 2	SQLDB232.DLL 및 SQL_DB2.CNF
DB/2 5	SQLDB2V5.DLL 및 SQL_DB2V5.CNF
InterBase	SQLINT32.DLL 및 SQL_INT.CNF

InstallShield Express나 기타 인증된 설치 프로그램을 사용하여 SQL Links를 설치합니다. SQL Links 설치 및 구성에 대한 자세한 내용은 디폴트로 메인 BDE 디렉토리에 설치되는 도움말 파일 SQLLNK32.HLP를 참조하십시오.

멀티 티어 데이터베이스 애플리케이션(DataSnap) 배포

DataSnap은 클라이언트 애플리케이션이 애플리케이션 서버에 있는 프로바이더에 연결할 수 있게 함으로써 C++Builder에 멀티 티어 데이터베이스 기능을 제공합니다.

InstallShield Express나 기타 Borland 인증 설치 스크립팅 유틸리티를 사용하여 DataSnap을 멀티 티어 애플리케이션과 함께 설치합니다. 애플리케이션과 함께 재배포해야 할 파일에 대한 자세한 내용은 메인 C++Builder 디렉토리에 있는 DEPLOY 설명서를 참조하십시오. 또한 DataSnap 파일을 재배포할 수 있는 대상과 방법에 대한 내용은 REMOTE 설명서를 참조하십시오.

웹 애플리케이션 배포

일부 C++Builder 애플리케이션은 World Wide Web을 통해 서버사이드 확장 DLL(ISAPI 및 Apache), CGI 애플리케이션 및 ActiveForms 형태로 실행되도록 디자인되어 있습니다.

웹 애플리케이션을 배포하는 단계는 애플리케이션의 파일을 웹 서버에 배포하는 것만 다를 뿐, 일반적인 애플리케이션을 배포하는 단계와 동일합니다. 일반적인 애플리케이션 설치에 대한 자세한 내용은 17-1페이지의 "일반적인 애플리케이션 배포"를 참조하십시오. 데이터베이스 웹 애플리케이션 배포에 대한 자세한 내용은 17-6페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오.

웹 애플리케이션을 배포할 때 특별히 주의해야 할 사항은 다음과 같습니다.

- BDE 데이터베이스 애플리케이션의 경우 Borland Database Engine 또는 대체 데이터베이스 엔진은 웹 서버에 애플리케이션 파일과 함께 설치됩니다.
- dbExpress 애플리케이션의 경우 dbExpress DLL을 경로에 포함시켜야 합니다. dbExpress DLL을 포함시킨 경우 dbExpress 드라이버는 웹 서버에 애플리케이션 파일과 함께 설치됩니다.
- 애플리케이션에서 필요한 모든 데이터베이스 파일에 액세스할 수 있도록 디렉토리에 대한 보안을 설정해야 합니다.
- 애플리케이션이 포함된 디렉토리에는 읽기 및 실행 어트리뷰트(attribute)가 있어야 합니다.
- 애플리케이션은 데이터베이스나 기타 파일에 액세스하기 위해 하드 코딩된 경로를 사용해서는 안됩니다.
- ActiveX 컨트롤의 위치는 <OBJECT> HTML 태그의 CODEBASE 매개변수에 의해 지정됩니다.

Apache에 배포하는 내용에 대해서는 다음 단원에서 설명합니다.

Apache 서버에 배포

WebBroker는 DLL 및 CGI 애플리케이션을 위해 Apache 버전 1.3.9 이상을 지원합니다. Apache는 conf 디렉토리에 있는 파일에 의해 구성됩니다.

Apache DLL을 만드는 경우에는 Apache 서버 설정 파일인 httpd.conf에 해당 지시어를 설정해야 합니다. .dll은 Apache 소프트웨어의 Modules 하위 디렉토리에 두는 것이 좋습니다.

CGI 애플리케이션을 만드는 경우 CGI 스크립트가 실행될 수 있도록 프로그램 실행을 허용하려면 httpd.conf 파일의 Directory 지시어에서 지정한 실제 디렉토리에 ExecCGI 옵션이 있어야 합니다. 사용 권한을 제대로 설정하려면 ScriptAlias 지시어를 사용하거나 Options ExecCGI를 on으로 설정해야 합니다.

ScriptAlias 지시어는 서버에 가상 디렉토리를 만들고 대상 디렉토리에 CGI 스크립트가 포함되어 있는 것으로 표시합니다. 예를 들어, httpd.conf 파일에 다음 줄을 추가할 수 있습니다.

```
ScriptAlias /cgi-bin "c:\inetpub\cgi-bin"
```

이렇게 하면 스크립트 c:\inetpub\cgi-bin\mycgi를 실행하여 /cgi-bin/mycgi와 같은 요청을 만족시킬 수 있습니다.

또한 httpd.conf의 Directory 지시어를 사용하여 Options를 All 또는 ExecCGI로 설정할 수도 있습니다. Options 지시어는 특정 디렉토리에서 사용할 수 있는 서버 기능을 제어합니다. Directory 지시어는 명명된 디렉토리와 그 하위 디렉토리에 적용된 일련의 지시어를 포함하는 데 사용됩니다. 다음은 Directory 지시어의 예제입니다.

```
<Directory <apache-root-dir>\cgi-bin>
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

이 예제에서 Options는 디렉토리 cgi-bin에서 CGI 스크립트 실행을 허용하는 ExecCGI로 설정됩니다.

참고 Apache는 httpd.conf 파일의 User 지시어에 지정된 계정 내의 서버에서 로컬로 실행됩니다. 사용자가 애플리케이션에 필요한 리소스에 액세스할 수 있도록 적절한 권한을 부여해야 합니다.

Apache 소프트웨어 배포와 관련된 정보는 Apache 배포에 포함된 Apache LICENSE 파일에서 찾을 수 있습니다. 또한 Apache 웹 사이트 www.apache.org에서 구성 정보를 찾아볼 수도 있습니다.

다양한 호스트 환경을 위한 프로그래밍

다양한 운영 체제 환경의 특성으로 인해 사용자 기본 설정이나 구성에 영향을 주는 요인이 많이 있습니다. 다음 요인은 다른 컴퓨터에 배포된 애플리케이션에 영향을 줄 수 있습니다.

- 화면 해상도와 색상 깊이
- 글꼴
- 운영 체제 버전
- Helper 애플리케이션
- DLL 위치

화면 해상도와 색상 깊이

데스크탑의 크기와 컴퓨터에서 사용 가능한 색상은 구성 가능하며 설치된 하드웨어에 따라 다릅니다. 이러한 어트리뷰트(attribute)는 개발용 컴퓨터와 배포용 컴퓨터에서 서로 다를 수도 있습니다.

다양한 화면 해상도를 위해 구성된 컴퓨터의 애플리케이션 모양(윈도우, 객체 및 글꼴 크기)은 다양한 방법으로 처리할 수 있습니다.

- 사용자가 사용할 가장 낮은 해상도(보통 640x480)에 맞춰 애플리케이션을 디자인합니다. 특별한 작업을 하지 않아도 호스트 컴퓨터의 화면 표시에 비례하여 동적으로 객체의 크기가 조절됩니다. 해상도를 높게 설정할수록 객체의 크기가 시각적으로 작아집니다.
- 개발용 컴퓨터에서 임의의 화면 해상도를 사용하여 디자인하고 런타임에 개발용 컴퓨터와 배포용 컴퓨터 사이의 화면 해상도 차이에 비례(화면 해상도 차이 비율)하여 모든 폼과 객체의 크기를 동적으로 조정합니다.
- 개발용 컴퓨터에서 임의의 화면 해상도를 사용하여 디자인하고 런타임에 애플리케이션의 폼 크기만 동적으로 조정합니다. 폼에서의 비주얼(visual) 컨트롤 위치에 따라 사용자가 폼의 모든 컨트롤에 액세스할 수 있도록 폼을 스크롤할 수 있어야 합니다.

동적으로 크기가 조정되지 않는 경우의 고려 사항

애플리케이션을 구성하는 폼과 비주얼(visual) 컨트롤의 크기가 런타임에 동적으로 조정되지 않으면 애플리케이션의 요소를 가장 낮은 해상도에 맞춰 디자인합니다. 그렇지 않으면 개발용 컴퓨터보다 낮은 화면 해상도로 구성된 컴퓨터에서 애플리케이션을 실행할 경우 애플리케이션의 폼이 화면 경계와 겹쳐 보일 수 있습니다.

예를 들어 개발용 컴퓨터가 1024x768의 화면 해상도로 설정되고 폼이 700픽셀 너비로 디자인된 경우 640x480 화면 해상도로 구성된 컴퓨터의 데스크탑에서는 해당 폼의 일부만 보입니다.

폼과 컨트롤 크기를 동적으로 조정하는 경우의 고려 사항

애플리케이션의 폼과 비주얼(visual) 컨트롤을 동적으로 크기 조정하는 경우 가능한 모든 화면 해상도에서 최적의 애플리케이션 모습이 되도록 크기 조정 과정의 모든 측면을 고려합니다. 애플리케이션의 비주얼(visual) 요소를 동적으로 크기 조정할 때는 다음 요인을 고려합니다.

- 개발용 컴퓨터의 화면 해상도와 애플리케이션이 설치된 컴퓨터의 화면 해상도를 비교하여 계산된 비율에 따라 폼과 비주얼(visual) 컨트롤의 크기를 조정합니다. 개발용 컴퓨터의 화면 해상도에 따라 한 치수를 나타낼 때는 상수를 사용하여 높이나 너비를 픽셀로 지정합니다. *TScreen::Height* 또는 *TScreen::Width* 속성을 사용하여 런타임에 사용자 컴퓨터에 대해 같은 치수를 검색합니다. 두 컴퓨터의 화면 해상도 사이의 차이 비율을 계산하려면 개발용 컴퓨터의 값을 사용자 컴퓨터의 값으로 나눕니다.
- 폼에 있는 요소의 크기와 위치를 줄이거나 늘려 애플리케이션(폼과 컨트롤)에 있는 비주얼(visual) 요소의 크기를 조정합니다. 이 경우 개발용 컴퓨터와 사용자 컴퓨터 사이의 화면 해상도 차이에 비례하여 크기를 조정합니다. *TCustomForm::Scaled* 속성을 **true**로 설정하고 *TWinControl::ScaleBy* 메소드(크로스 플랫폼 애플리케이션을 위해서는 *TWidgetControl.ScaleBy*)를 호출하여 폼에 있는 비주얼(visual) 컨트롤의 크기와 위치를 조정합니다. *ScaleBy* 메소드는 폼의 높이와 너비를 변경하지 않습니다. *Height*와 *Width* 속성의 현재 값을 화면 해상도 차이 비율로 곱하여 수동으로 폼의 높이와 너비를 변경합니다.

- 폼의 컨트롤은 *TWinControl::ScaleBy* 메소드(크로스 플랫폼 애플리케이션을 위해서는 *TWidgetControl.ScaleBy*) 메소드로 자동으로 크기를 조정하지 않고, 순환문에서 각 비주얼(visual) 컨트롤을 참조하고 치수와 위치를 설정하여 수동으로 크기를 조정할 수 있습니다. 비주얼(visual) 컨트롤의 *Height* 및 *Width* 속성 값을 화면 해상도 차이 비율로 곱합니다. *Top* 및 *Left* 속성 값을 같은 비율로 곱하여 화면 해상도 차이에 비례하여 비주얼(visual) 컨트롤을 재배치합니다.
- 사용자 컴퓨터의 화면 해상도보다 더 높은 화면 해상도로 구성된 컴퓨터에서 애플리케이션이 디자인된 경우 비주얼(visual) 컨트롤의 비례적인 크기 조정 과정에서 글꼴 크기가 줄어듭니다. 디자인 타임의 글꼴 크기가 너무 작으면 런타임에 크기 조정된 글꼴을 읽을 수 없습니다. 예를 들어, 폼의 디폴트 글꼴 크기는 8입니다. 개발용 컴퓨터의 화면 해상도가 1024x768이고 사용자의 컴퓨터가 640x480인 경우 비주얼(visual) 컨트롤 치수는 0.625(640 / 1024 = 0.625)로 감소됩니다. 따라서 원래의 글꼴 크기는 8에서 5(8 * 0.625 = 5)로 줄어듭니다. 애플리케이션의 텍스트는 줄어든 글꼴 크기로 표시되어 읽을 수 없게 됩니다.
- *TLabel*과 *TEdit*같은 일부 비주얼(visual) 컨트롤은 컨트롤의 글꼴 크기가 변경될 때 동적으로 크기가 조정됩니다. 그 결과 폼과 컨트롤이 동적으로 크기가 조정될 때 배포된 애플리케이션에 영향을 줄 수 있습니다. 글꼴 크기 변경으로 인한 컨트롤의 크기 조정이 화면 해상도의 비례적 크기 조정으로 인한 크기의 변경에 추가됩니다. 이 영향은 이들 컨트롤의 *AutoSize* 속성을 **false**로 설정하여 상쇄할 수 있습니다.
- 캔버스에 직접 그리는 것처럼 명시적인 픽셀 좌표를 사용하는 것은 피해야 합니다. 대신 비율 비례에 의한 조정으로 개발용 컴퓨터와 사용자 컴퓨터 사이의 화면 해상도 차이를 수정합니다. 예를 들어 애플리케이션이 10x20픽셀로 캔버스에 직사각형을 그리는 경우 화면 해상도 차이 비율에 10과 20을 곱합니다. 이렇게 하면 다른 화면 해상도에서도 같은 크기로 나타납니다.

다양한 색상 깊이 지원

색상 표현 능력이 서로 다르게 설정된 모든 배포용 컴퓨터에 대해 가장 안전한 방법은 가능한 한 최소의 색상 수로 그래픽을 사용하는 것입니다. 이 방법은 전형적인 16색 그래픽을 사용해야 하는 컨트롤 문자 모양에 대해 적합합니다. 그림을 표시하기 위해서는 다양한 해상도와 다른 색상 깊이를 사용하여 이미지 복사본을 여러 개 제공하거나 애플리케이션에 대해 요구되는 조건으로 최소 해상도와 색상 수를 나타냅니다.

글꼴

Windows에서는 표준 TrueType 및 래스터 글꼴 집합을 제공합니다. Linux에서는 배포에 따라 표준 글꼴 집합을 제공합니다. 다른 컴퓨터에 배포할 애플리케이션을 디자인할 때 일부 컴퓨터에는 표준 집합의 글꼴만 있다는 점을 염두에 두어야 합니다.

애플리케이션에 사용되는 텍스트 컴포넌트는 모든 배포용 컴퓨터에서 사용 가능한 글꼴을 모두 사용해야 합니다.

애플리케이션에서 비표준 글꼴을 꼭 사용해야 하는 경우 해당 글꼴을 애플리케이션과 함께 배포해야 합니다. 설치 프로그램이나 애플리케이션은 배포용 컴퓨터에 글꼴을 설치해야 합니다. 서드파티 글꼴을 배포할 경우 글꼴 작성자가 지정한 제한 사항에 따라야 할 수 있습니다.

Windows에는 컴퓨터에 없는 글꼴을 사용하려 하는 경우를 대비한 안전한 방법이 있습니다. 없는 글꼴과 가장 비슷한 기존의 다른 글꼴로 대체합니다. 이 방법은 없는 글꼴로 인한 오류를 방지할 수 있지만 결과적으로 애플리케이션의 시각적 품질이 떨어질 수 있습니다. 따라서 디자인 타임 시 이러한 결과에 대비하여 미리 대비하는 것이 좋습니다.

Windows 애플리케이션에서 비표준 글꼴을 사용할 수 있게 하려면 Windows API 함수 *AddFontResource* 및 *DeleteFontResource*를 사용하십시오. 비표준 글꼴의 .fot 파일을 애플리케이션과 함께 배포합니다.

운영 체제 버전

운영 체제 API를 사용하거나 애플리케이션에서 운영 체제 영역에 액세스할 때는 다른 운영 체제 버전을 사용하는 컴퓨터에서 기능, 작업 또는 영역을 사용하지 못할 수 있습니다.

이러한 가능성을 배제하려면 다음 두 방법을 사용하십시오.

- 애플리케이션의 시스템 요구 사항에서 애플리케이션을 실행할 수 있는 운영 체제 버전을 지정합니다. 호환 가능한 운영 체제 버전에서만 애플리케이션을 설치하고 사용하는 것은 사용자의 책임입니다.
- 애플리케이션이 설치될 때 운영 체제 버전을 확인합니다. 호환되지 않는 버전의 운영 체제가 있으면 설치 과정을 중단하거나 이 문제에 대해 경고 메시지를 표시하게 합니다.
- 런타임 시 모든 버전에서 사용할 수 없는 작업을 실행하기 직전에 운영 체제 버전을 확인합니다. 호환되지 않는 버전의 운영 체제가 있으면 설치 과정을 중단하고 사용자에게 경고 메시지를 표시합니다. 또는 다른 운영 체제 버전에서 실행되도록 다른 코드를 제공합니다.

참고 일부 작업은 Windows 95/98와 Windows NT/2000/XP에서 다르게 수행됩니다. Windows 버전을 확인하려면 Windows API 함수 *GetVersionEx*를 사용하십시오.

소프트웨어 사용권 요구 사항

C++Builder 애플리케이션과 관련된 일부 파일의 배포는 제한 사항이 있거나 전혀 재배포할 수 없습니다. 다음 문서는 이런 파일의 배포에 관한 법적 규정을 설명합니다.

배포

DEPLOY 설명서에서는 다양한 컴포넌트와 유틸리티, C++Builder 애플리케이션의 일부 또는 관련된 다른 제품의 배포에 관련된 일부 법률적인 내용이 들어 있습니다. DEPLOY 설명서는 메인 C++Builder 디렉토리에 설치됩니다. 이 설명서는 다음에 대해 다룹니다.

- .exe, .dll 및 .bpl 파일
- 컴포넌트 및 디자인 타임 패키지
- Borland Database Engine
- ActiveX 컨트롤
- 샘플 이미지
- SQL Links

README

README 설명서에는 컴포넌트, 유틸리티 또는 기타 제품의 재배포 권리에 관련된 정보를 비롯한 C++Builder에 대한 최종 정보를 포함합니다. README 설명서는 메인 C++Builder 디렉토리에 설치됩니다.

사용권 계약서

인쇄된 문서인 C++Builder 사용권 계약서는 C++Builder에 관련된 기타 법적 권리와 의무를 규정합니다.

서드파티 제품 설명서

서드파티의 컴포넌트, 유틸리티, Helper 애플리케이션, 데이터베이스 엔진 및 기타 제품에 대한 재배포 권리는 제품을 공급하는 업체에게 있습니다. 제품을 배포하기 전에 먼저 제품 설명서를 참조하거나 업체에 문의하여 C++Builder 애플리케이션을 사용한 제품 재배포에 대한 정보를 확인하십시오.

부

II

데이터베이스 애플리케이션 개발

"데이터베이스 애플리케이션 개발"에 포함된 장에서는 C++Builder 데이터베이스 애플리케이션을 만드는 데 필요한 개념과 기술을 설명합니다.

참고 데이터베이스 애플리케이션 구축에 대한 지원 수준은 구입한 C++Builder 에디션에 따라 다릅니다. 특히 클라이언트 데이터셋을 사용하려면 최소 전문가용 에디션 이상이 필요하고, 멀티티어 데이터베이스 애플리케이션을 개발하려면 기업용 에디션이 필요합니다.

장

18

데이터베이스 애플리케이션 디자인

데이터베이스 애플리케이션을 사용하면 사용자가 데이터베이스에 저장된 정보와 상호 작용할 수 있습니다. 데이터베이스는 이러한 정보에 대한 구조를 제공하여 각기 다른 애플리케이션 간에 정보를 공유할 수 있게 합니다.

C++Builder는 관계형 데이터베이스 애플리케이션을 지원합니다. 관계형 데이터베이스는 행(레코드)과 열(필드)을 포함하는 테이블로 정보를 구성합니다. 이러한 테이블은 관계형 연산 같은 간단한 작업으로 조작될 수 있습니다.

데이터베이스 애플리케이션을 디자인할 때는 데이터의 구성 방법을 이해해야 합니다. 이 구조를 기반으로 사용자에게 데이터를 표시할 사용자 인터페이스를 디자인하고 사용자가 새 정보를 입력하거나 기존 데이터를 수정하게 할 수 있습니다.

이 장에서는 데이터베이스 애플리케이션 디자인에 대한 일반적인 고려 사항과 사용자 인터페이스 디자인과 관련하여 결정할 내용에 대해 설명합니다.

데이터베이스 사용

C++Builder에는 데이터베이스에 액세스하고 데이터베이스에 포함된 정보를 표시하기 위한 여러 가지 컴포넌트가 있습니다. 이 컴포넌트는 데이터 액세스 메커니즘에 따라 그룹화됩니다.

- 컴포넌트 팔레트의 BDE 페이지에는 BDE(Borland Database Engine)를 사용하는 컴포넌트가 있습니다. BDE는 데이터베이스와 상호 작용하기 위한 대형 API를 정의합니다. 모든 데이터 액세스 메커니즘 중에서 BDE 는 가장 광범위한 기능을 지원하며 대부분의 가장 강력한 유틸리티를 제공합니다. Paradox나 dBASE 테이블의 데이터로 작업하는 것은 가장 이상적인 방법이기는 하지만 배포하기에 가장 복잡한 메커니즘이기도 합니다. BDE 컴포넌트 사용에 대한 자세한 내용은 24장, "Borland Database Engine 사용"을 참조하십시오.

- 컴포넌트 팔레트의 ADO 페이지에는 ADO(ActiveX Data Objects)를 사용하여 OLEDB를 통해 데이터베이스 정보에 액세스하는 컴포넌트가 있습니다. ADO는 Microsoft 표준입니다. 여러 가지 데이터베이스 서버에 연결하는 광범위한 ADO 드라이버를 사용할 수 있습니다. ADO 기반의 컴포넌트를 사용하면 애플리케이션을 ADO 기반의 환경에 통합할 수 있습니다. 예를 들어, ADO 기반의 애플리케이션 서버를 사용할 수 있습니다. ADO 컴포넌트 사용에 대한 자세한 내용은 25장, "ADO 컴포넌트 사용"을 참조하십시오.
- 컴포넌트 팔레트의 dbExpress 페이지에는 dbExpress를 사용하여 데이터베이스 정보에 액세스하는 컴포넌트가 있습니다. dbExpress는 데이터베이스 정보를 가장 빠르게 액세스할 수 있는 lightweight 드라이버 집합입니다. 또한 dbExpress 컴포넌트는 Linux에서도 사용할 수 있으므로 크로스 플랫폼 개발을 지원합니다. 그러나 dbExpress 데이터베이스 컴포넌트는 가장 적은 범위의 데이터 조작 기능도 지원합니다. dbExpress 컴포넌트 사용에 대한 자세한 내용은 26장, "단방향 데이터셋 사용"을 참조하십시오.
- 컴포넌트 팔레트의 InterBase 페이지에는 각각의 엔진 레이어를 통과하지 않고 직접 InterBase 데이터베이스에 액세스하는 컴포넌트가 들어 있습니다.
- 컴포넌트 팔레트의 Data Access 페이지에는 데이터 액세스 메커니즘과 함께 사용할 수 있는 컴포넌트가 있습니다. 이 페이지에는 디스크에 저장된 데이터를 사용하거나, 역시 이 페이지에 있는 *TDataSetProvider* 컴포넌트를 사용하여 다른 그룹의 컴포넌트와 작업할 수 있는 *TClientDataset*이 있습니다. 클라이언트 데이터셋 사용에 대한 자세한 내용은 27장, "클라이언트 데이터셋 사용"을 참조하십시오. *TDataSetProvider*에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오.

참고 여러 가지 버전의 C++Builder에는 BDE, ADO 또는 dbExpress를 사용하여 데이터베이스 서버에 액세스하기 위한 다양한 드라이버가 포함되어 있습니다.

데이터베이스 애플리케이션을 디자인할 때는 사용할 컴포넌트 집합을 결정해야 합니다. 각 데이터 액세스 메커니즘마다 지원하는 기능 범위, 배포의 용이성 및 다양한 데이터베이스 서버를 지원하기 위한 드라이버 가용성이 서로 다릅니다.

데이터 액세스 메커니즘과 함께 데이터베이스 서버도 선택해야 합니다. 여러 가지 데이터베이스 타입이 있으므로 각 타입의 장단점을 고려하여 적절한 데이터베이스 서버를 결정해야 합니다.

타입에 관계 없이 모든 데이터베이스에는 정보를 저장하는 테이블이 있습니다. 또한 대부분의 서버는 다음과 같은 추가 기능을 지원합니다.

- 데이터베이스 보안
- 트랜잭션
- 참조 무결성, 내장 프로시저 및 트리거

데이터베이스의 타입

관계형 데이터베이스 서버마다 정보를 저장하는 방법과 여러 사용자가 동시에 해당 정보를 액세스하는 방법이 다릅니다. C++Builder는 다음과 같은 두 가지 타입의 관계형 데이터베이스 서버를 지원합니다.

- **원격 데이터베이스 서버**는 일반적으로 각각의 시스템에 상주합니다. 경우에 따라 원격 데이터베이스 서버의 데이터가 한 시스템에 상주하지 않고 여러 서버에 분산될 때도 있습니다. 원격 데이터베이스 서버마다 정보를 저장하는 방법은 다르지만 클라이언트에 공통적인 논리 인터페이스를 제공합니다. 이러한 공통적인 인터페이스는 SQL(Structured Query Language)입니다. 데이터베이스 서버는 SQL을 사용하여 액세스하므로 SQL 서버라고 하기도 하고 RDBMS(Remote Database Management system)라고도 합니다. SQL을 구성하는 공통적인 명령 외에 대부분의 원격 데이터베이스 서버는 고유한 SQL "언어"를 지원합니다. SQL 서버의 예로는 InterBase, Oracle, Sybase, Informix, Microsoft SQL server 및 DB2가 있습니다.
- **로컬 데이터베이스**는 로컬 드라이브 또는 LAN에 상주합니다. 로컬 데이터베이스에는 데이터를 액세스하기 위한 전용 API가 있는 경우가 있습니다. 여러 사용자가 로컬 데이터베이스를 공유할 때는 파일 기반 잠금 메커니즘을 사용합니다. 이 메커니즘 때문에 로컬 데이터베이스를 파일 기반 데이터베이스라고도 합니다. 로컬 데이터베이스의 예로는 Paradox, dBASE, FoxPro 및 Access가 있습니다.

로컬 데이터베이스를 사용하는 애플리케이션은 애플리케이션과 데이터베이스가 하나의 파일 시스템을 공유하기 때문에 **단일 티어 애플리케이션**이라고 합니다. 원격 데이터베이스 서버를 사용하는 애플리케이션은 애플리케이션과 데이터베이스가 독립 시스템(또는 티어)에서 작동되므로 **2티어 애플리케이션** 또는 **멀티 티어 애플리케이션**이라고 합니다.

사용할 데이터베이스 타입을 선택할 때는 여러 가지 요인을 고려해야 합니다. 예를 들어, 데이터가 기존 데이터베이스에 이미 저장되어 있을 수도 있습니다. 애플리케이션에서 사용하는 데이터베이스 테이블을 만들 때는 다음 사항을 고려해야 합니다.

- 이 테이블을 공유할 사용자는 몇 명인가? 원격 데이터베이스 서버는 여러 사용자가 동시에 액세스할 수 있도록 고안되었습니다. 원격 데이터베이스 서버는 트랜잭션이라는 메커니즘을 통해 여러 사용자를 지원합니다. Local InterBase 같은 일부 로컬 데이터베이스도 트랜잭션을 지원하지만 대부분의 로컬 데이터베이스는 파일 기반 잠금 메커니즘만 제공하고 클라이언트 데이터셋 파일 같은 일부 로컬 데이터베이스는 다중 사용자를 지원하지 않습니다.
- 테이블에 저장할 데이터 양은 어느 정도인가? 원격 데이터베이스 서버에는 로컬 데이터베이스보다 많은 데이터를 저장할 수 있습니다. 일부 원격 데이터베이스 서버는 많은 양의 데이터베이스를 저장하도록 디자인되어 있지만 일부 원격 데이터베이스 서버는 빠른 업데이트 등 다른 기능을 위해 최적화되어 있습니다.
- 데이터베이스에 어떤 성능(속도)이 필요한가? 로컬 데이터베이스는 데이터베이스 애플리케이션과 동일한 시스템에 상주하기 때문에 일반적으로 원격 데이터베이스 서버보다 빠릅니다. 원격 데이터베이스 서버마다 다른 타입의 작업 지원을 위해 최적화되어 있으므로 원격 데이터베이스 서버를 선택할 때는 성능을 고려해야 합니다.
- 데이터베이스 관리에 어떤 지원이 가능한가? 로컬 데이터베이스는 원격 데이터베이스 서버에 비해 필요한 지원 정도가 적습니다. 일반적으로 로컬 데이터베이스는 서버를 따로 설치하거나 비싼 사이트 사용권이 필요하지 않기 때문에 운영하는 데 적은 비용이 듭니다.

데이터베이스 보안

데이터베이스에는 종종 중요한 정보가 저장되기도 합니다. 각각의 데이터베이스에서 이러한 정보를 보호하기 위한 보안 방법을 제공합니다. Paradox와 dBASE와 같은 일부 데이터베이스는 테이블이나 필드 레벨의 보안만 제공합니다. 사용자가 보호된 테이블에 액세스하려면 암호를 제공해야 합니다. 일단 인증된 사용자는 사용 권한이 있는 필드(열)만 볼 수 있습니다.

대부분의 SQL 서버는 데이터베이스 서버를 사용하는 데 암호와 사용자 이름이 필요합니다. 사용자가 데이터베이스에 로그인하면 사용자 이름과 암호를 통해 사용할 수 있는 테이블이 결정됩니다. SQL 서버에 암호 제공에 대한 자세한 내용은 21-4페이지의 "서버 로그인 제어"를 참조하십시오.

데이터베이스 애플리케이션을 디자인할 때는 데이터베이스 서버에 어떤 타입의 인증이 필요한지를 고려해야 합니다. 애플리케이션은 종종 애플리케이션 자체에 대한 로그인만 필요한 사용자에게 명시적인 데이터베이스 로그인을 표시하지 않도록 디자인됩니다. 사용자에게 데이터베이스 암호를 요구하지 않으려면 암호가 필요하지 않은 데이터베이스를 사용하거나, 프로그램에서 암호와 사용자 이름을 서버에 제공해야 합니다. 프로그램에서 암호를 제공하는 경우 애플리케이션에서 암호를 읽어와서 보안이 침해되지 않도록 주의해야 합니다.

사용자가 암호를 입력하게 하려면 암호를 언제 요구할 것인지 결정해야 합니다. 지금은 로컬 데이터베이스를 사용하지만 나중에 대용량 SQL 서버로 확장할 계획이 있으면 각 테이블을 열 때 보다는 SQL 데이터베이스에 로그인할 때 암호를 요구하는 것이 좋습니다.

여러 개의 보호된 시스템이나 데이터베이스에 로그인해야 하므로 애플리케이션에 여러 암호가 필요한 경우에는 사용자가 보호된 시스템에 필요한 암호 테이블에 액세스하는 데 사용되는 하나의 마스터 암호를 제공하도록 할 수 있습니다. 그러면 사용자가 여러 암호를 제공하지 않아도 애플리케이션에서 프로그램을 통해 암호를 제공합니다.

멀티 티어 애플리케이션에서는 다양한 보안 모델을 함께 사용할 수 있습니다. HTTP나 COM+를 사용하여 미들 티어에 대한 액세스를 제어할 수 있고, 미들 티어에서 데이터베이스 서버 로그인에 대한 상세한 부분을 모두 처리하도록 할 수 있습니다.

트랜잭션

트랜잭션은 데이터베이스에 있는 하나 이상의 테이블이 커밋되기 전에 성공적으로 모두 수행되어야 하는 액션 그룹입니다. 그룹 내의 작업이 하나라도 실패할 경우 모든 액션이 롤백(취소)됩니다.

트랜잭션을 사용하면 다음을 수행할 수 있습니다.

- 단일 트랜잭션의 모든 업데이트가 커밋되거나 중지되고 이전 상태로 롤백됩니다. 이것을 **원자성**이라고 합니다.
- 트랜잭션은 상태의 불변값을 유지하는 시스템 상태의 바람직한 변환입니다. 이것을 **일관성**이라고 합니다.
- 동시 트랜잭션은 서로 부분적인 결과 또는 커밋되지 않은 결과를 알지 못하므로 애플리케이션 상태에서 일관되지 않은 부분이 있을 수 있습니다. 이것을 **분리**라고 합니다.
- 통신 실패, 프로세스 실패 및 서버 시스템 실패 등의 실패에도 레코드에 커밋된 업데이트는 남아 있습니다. 이것을 **지속성**이라고 합니다.

따라서 트랜잭션은 데이터베이스 명령이나 일련의 명령 중에 발생하는 하드웨어 오류에 대해 보호합니다. 트랜잭션 로깅을 사용하면 디스크 미디어 실패 후에도 지속적인 상태를 복구할 수 있습니다. 또한 트랜잭션은 SQL 서버에서 여러 사용자 동시성 제어의 기초를 형성합니다. 각 사용자가 트랜잭션을 통해서만 데이터베이스와 상호 작용할 때는 한 사용자의 명령이 다른 사용자의 트랜잭션 단위를 방해할 수 없습니다. 대신 SQL 서버는 전체적으로 성공하거나 실패하는 수신 트랜잭션을 계획합니다.

로컬 InterBase에서는 지원되지만 대부분의 로컬 데이터베이스에서는 트랜잭션을 지원하지 않습니다. 또한 BDE 드라이버는 일부 로컬 데이터베이스에 대해 제한된 트랜잭션 지원을 제공합니다. 데이터베이스 트랜잭션 지원은 데이터베이스 연결을 나타내는 컴포넌트에 의해 제공됩니다. 데이터베이스 연결 컴포넌트를 사용하여 트랜잭션을 관리하는 데 대한 자세한 내용은 21-6페이지의 "트랜잭션 관리"를 참조하십시오.

멀티 티어 애플리케이션에서는 데이터베이스 작업 이외의 액션을 포함하거나 여러 데이터베이스로 확장되는 트랜잭션을 만들 수 있습니다. 멀티 티어 애플리케이션에서 트랜잭션을 사용하는 데 대한 자세한 내용은 29-17페이지의 "멀티 티어 애플리케이션의 트랜잭션 관리"를 참조하십시오.

참조 무결성, 내장 프로시저 및 트리거

모든 관계형 데이터베이스에는 애플리케이션이 데이터를 저장하고 조작할 수 있는 기능을 공통적으로 가지고 있습니다. 또한 데이터베이스는 경우에 따라 데이터베이스 테이블 간에 일관적인 관계를 유지하는 데 유용한 데이터베이스별 기능을 제공합니다. 다음과 같은 기능이 있습니다.

- **참조 무결성.** 참조 무결성은 테이블 간의 마스터/디테일 관계가 손상되지 않도록 하는 메커니즘을 제공합니다. 고아 디테일 레코드를 발생시킬 수 있는 마스터 테이블의 필드를 사용자가 삭제하려는 경우, 참조 무결성 규칙은 삭제를 막거나 자동으로 고아 디테일 레코드를 삭제합니다.
- **내장 프로시저.** 내장 프로시저는 이름이 지정되어 SQL 서버에 저장된 SQL 문의 집합입니다. 내장 프로시저는 보통 서버에서 공통 데이터베이스 관련 작업을 수행하고 때로 레코드 집합(데이터셋)을 반환합니다.
- **트리거.** 트리거는 특정 명령에 응답하여 자동으로 실행되는 SQL 문의 집합입니다.

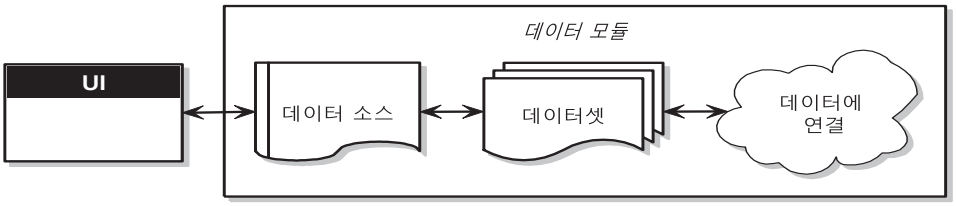
데이터베이스 아키텍처

데이터베이스 애플리케이션은 사용자 인터페이스 요소, 데이터 정보를 나타내는 컴포넌트(데이터셋) 및 이들을 서로 연결하고 데이터베이스 정보 소스에 연결하는 컴포넌트로 구축됩니다. 이러한 요소들을 구성하는 방법이 데이터베이스 애플리케이션의 아키텍처입니다.

일반적인 구조

데이터베이스 애플리케이션의 컴포넌트를 구성하는 많은 다양한 방법들이 있지만 대부분은 다음 그림 18.1에 설명되어 있는 일반적인 방법을 따릅니다.

그림 18.1 일반적인 데이터베이스 아키텍처



사용자 인터페이스 폼

애플리케이션의 나머지 부분과 완전히 분리된 폼에 사용자 인터페이스를 분리하는 것이 좋습니다. 이렇게 하면 여러 가지 장점이 있습니다. 데이터베이스 정보 자체를 나타내는 컴포넌트에서 사용자 인터페이스를 분리하여 디자인에 보다 많은 유연성을 부여합니다. 데이터베이스 정보를 관리하는 방법을 바꾸기 위해 사용자 인터페이스를 다시 작성하거나, 사용자 인터페이스를 변경하기 위해 데이터베이스를 사용하는 애플리케이션의 일부를 변경할 필요가 없습니다. 또한 이러한 타입의 분리를 사용하여 다중 애플리케이션 간에 공유할 수 있는 공통 폼을 개발할 수 있어서 일관성 있는 사용자 인터페이스를 제공합니다. 개발자들은 Object Repository에 잘 디자인된 폼에 대한 연결을 저장하여 매번 처음부터 새 프로젝트를 시작하지 않고 기존의 프로젝트를 기초로 하여 새 프로젝트를 생성할 수 있습니다. 또한 폼을 공유하여 애플리케이션 인터페이스에 대한 기업 표준을 개발할 수 있습니다. 데이터베이스 애플리케이션용 사용자 인터페이스 작성에 대한 자세한 내용은 18-15페이지의 "사용자 인터페이스 디자인"을 참조하십시오.

데이터 모듈

사용자 인터페이스를 자체 폼으로 분리한 경우, 데이터 모듈을 사용하여 데이터 정보를 나타내는 컴포넌트(데이터셋)와 이러한 데이터셋을 애플리케이션의 다른 부분에 연결하는 컴포넌트를 수용할 수 있습니다. 사용자 인터페이스 폼처럼 애플리케이션 간에 데이터 모듈을 재사용하고 공유할 수 있도록 Object Repository의 데이터 모듈을 공유할 수 있습니다.

데이터 소스

데이터 모듈의 첫 번째 항목은 데이터 소스입니다. 데이터 소스는 사용자 인터페이스와 데이터베이스의 정보를 나타내는 데이터셋 간의 통로 역할을 합니다. 폼에 있는 여러 가지 데이터 인식 컨트롤은 하나의 데이터 소스를 공유할 수 있습니다. 이 경우 각 컨트롤의 디스플레이가 동기화되므로 사용자가 레코드를 스크롤하고 현재 레코드의 필드에 있는 해당 값을 각 컨트롤에 표시할 수 있습니다.

데이터셋

데이터베이스 애플리케이션의 핵심은 데이터셋입니다. 이 컴포넌트는 원본으로 사용하는 데이터베이스의 레코드 집합을 나타냅니다. 이러한 레코드는 단일 데이터베이스 테이블의 데이터, 테이블의 필드나 레코드의 부분 집합, 또는 싱글 뷰로 조인된 하나 이상의 테이블 정보일 수 있습니다. 데이터셋을 사용함으로써 애플리케이션 로직은 데이터베이스의 물리적 테이블을 재구성해서 버퍼링됩니다. 원본으로 사용하는 데이터베이스가 변경되면 데이터셋 컴포넌트가 포함하고 있는 데이터를 지정하는 방법을 변경해야 할 수도 있지만 애플리케이션의 나머지 부분은 변경하지 않아도 계속 작동됩니다. 데이터셋의 일반적인 속성과 메소드에 대한 자세한 내용은 22장, "데이터셋 이해"를 참조하십시오.

데이터 연결

데이터셋이 원본으로 사용하는 데이터베이스 정보에 연결할 때 사용하는 메커니즘은 데이터셋의 타입에 따라 차이가 있습니다. 또한, 이처럼 서로 다른 메커니즘을 사용함에 따라 개발자가 생성할 수 있는 데이터베이스 애플리케이션의 아키텍처도 크게 달라집니다. 데이터를 연결하는 기본 메커니즘에는 다음 네 가지가 있습니다.

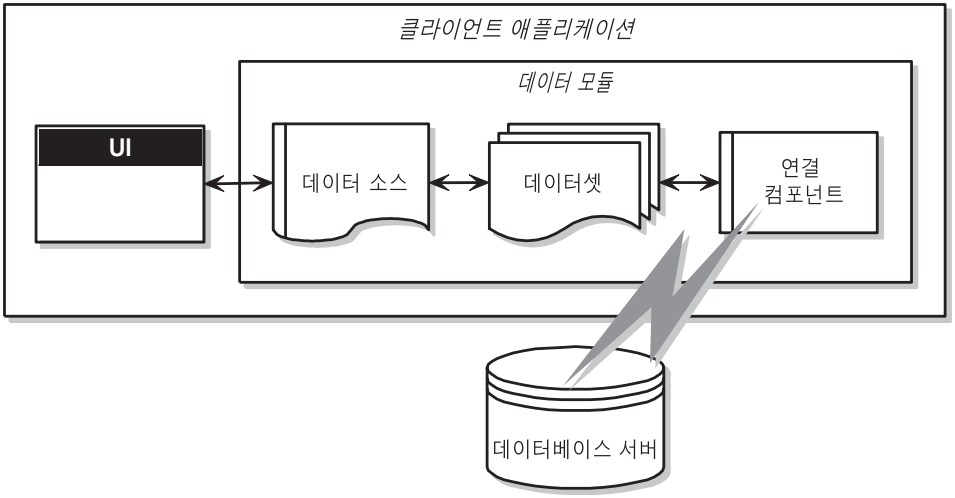
- 데이터베이스 서버에 직접 연결합니다. 대부분의 데이터셋은 *TCustomConnection*의 자손을 사용하여 데이터베이스 서버에 대한 연결을 나타냅니다.
- 디스크에 있는 전용 파일을 사용합니다. 클라이언트 데이터셋은 디스크에 있는 전용 파일로 작업할 수 있는 기능을 지원합니다. 클라이언트 데이터셋 자체에서 파일을 읽고 쓸 수 있으므로 전용 파일 작업을 할 때는 각각의 연결 컴포넌트를 필요로 하지 않습니다.
- 다른 데이터셋에 연결합니다. 클라이언트 데이터셋은 다른 데이터셋에 의해 제공된 데이터 작업을 할 수 있습니다. *TDataSetProvider* 컴포넌트는 클라이언트 데이터셋과 해당 소스 데이터셋 간의 매개체 역할을 합니다. 이 데이터셋 프로바이더는 클라이언트 데이터셋과 같은 데이터 모듈에 상주하거나 다른 시스템에서 실행 중인 애플리케이션 서버의 일부가 될 수 있습니다 . 프로바이더가 애플리케이션 서버의 일부이면 애플리케이션 서버 연결을 나타내는 데 *TCustomConnection*의 특수 자손이 필요합니다.
- RDS DataSpace 객체에서 데이터를 구합니다. ADO 기반의 애플리케이션 서버를 사용하여 생성한 멀티 티어 데이터베이스 애플리케이션의 데이터를 마샬링하기 위해 ADO 데이터셋은 *TRDSConnection* 컴포넌트를 사용할 수 있습니다.

때로 이러한 메커니즘은 하나의 애플리케이션으로 결합될 수 있습니다.

데이터베이스 서버에 직접 연결

대부분의 일반적인 데이터베이스 아키텍처는 데이터셋이 연결 컴포넌트를 사용하여 데이터베이스 서버에 대한 연결을 구축하는 데 사용됩니다. 그러면 데이터셋이 서버로부터 직접 데이터를 가져와서 편집 내용을 서버에 직접 포스트합니다. 이 내용은 그림 18.2에 설명되어 있습니다.

그림 18.2 데이터베이스 서버에 직접 연결



각 타입의 데이터셋은 하나의 데이터 액세스 메커니즘을 나타내는 고유한 연결 컴포넌트 타입을 사용합니다.

- 데이터셋이 *TTable*, *TQuery* 또는 *TStoredProc* 등의 BDE 데이터셋이면 연결 컴포넌트가 *TDataBase* 객체입니다. 해당 *Database* 속성을 설정하여 데이터셋을 데이터베이스 컴포넌트에 연결합니다. BDE 데이터셋을 사용할 때는 데이터베이스 컴포넌트를 명시적으로 추가할 필요가 없습니다. 데이터셋의 *DatabaseName* 속성을 설정하면 데이터베이스 컴포넌트가 런타임에 자동으로 생성됩니다.
- 데이터셋이 *TADODataSet*, *TADOTable*, *TADOQuery* 또는 *TADOStoredProc* 등의 ADO 데이터셋이면 연결 컴포넌트가 *TADOConnection* 객체입니다 . 해당 *ADOConnection* 속성을 설정하여 ADO 연결 컴포넌트에 데이터셋을 연결합니다. BDE 데이터셋을 사용하면 연결 컴포넌트를 명시적으로 추가하지 않고 대신 데이터셋의 *ConnectionString* 속성을 설정할 수 있습니다.
- 데이터셋이 *TSQLDataSet*, *TSQLTable*, *TSQLQuery* 또는 *TSQLStoredProc* 등의 dbExpress 데이터셋이면 연결 컴포넌트가 *TSQLConnection* 객체입니다. 해당 *SQLConnection* 속성을 설정하여 SQL 연결 컴포넌트에 데이터셋을 연결합니다. dbExpress 데이터셋을 사용할 때는 연결 컴포넌트를 명시적으로 추가해야 합니다. dbExpress 데이터셋과 다른 데이터셋의 또 다른 차이점은 dbExpress 데이터셋은 항상 읽기 전용이고 단방향이라는 점입니다. 즉, 레코드를 순서대로 반복하여 탐색할 수만 있으며 편집을 지원하는 데이터셋 메소드를 사용할 수 없습니다.
- 데이터셋이 *TIBDataSet*, *TIBTable*, *TIBQuery* 또는 *TIBStoredProc* 등의 InterBase Express 데이터셋이면 연결 컴포넌트가 *TIBDatabase* 객체입니다. 해당 *Database* 속성을 설정하여 데이터셋을 IB 데이터베이스 컴포넌트에 연결합니다. dbExpress 데이터셋을 사용할 때는 연결 컴포넌트를 명시적으로 추가해야 합니다.

위에 나열된 컴포넌트 외에 *TBDEClientDataSet*, *TSQLClientDataSet* 또는 *TIBClientDataSet*과 같은 특화된 클라이언트 데이터셋을 사용할 수 있습니다. 이러한 클라이언트 데이터셋 중 하나를 사용할 때는 해당 연결 컴포넌트 타입을 *DBConnection* 속성 값으로 지정하십시오.

각 데이터셋 타입마다 다른 연결 컴포넌트를 사용하지만 모두 같은 작업을 수행하는 일이 많고 같은 속성, 메소드 및 이벤트를 발생시키는 경우가 많습니다. 여러 데이터베이스 연결 컴포넌트 간의 공통성에 대한 자세한 내용은 21장, "데이터베이스에 연결"을 참조하십시오.

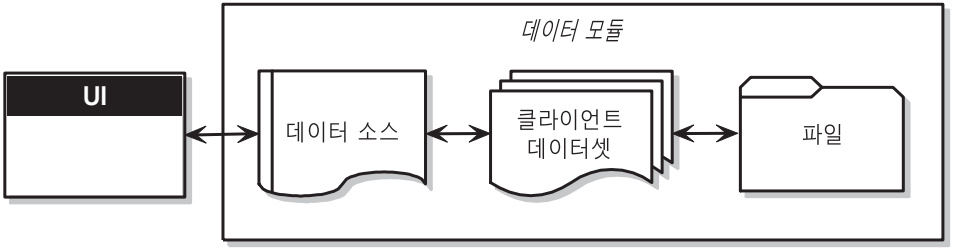
이 아키텍처는 데이터베이스 서버가 로컬 데이터베이스인지 원격 데이터베이스 서버인지에 따라 단일 티어 또는 2티어 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

참고 2티어 애플리케이션을 만드는 데 필요한 연결 컴포넌트나 드라이버를 모든 버전의 C++Builder에서 사용할 수 있는 것은 아닙니다.

디스크에 있는 전용 파일 사용

작성 가능한 데이터베이스 애플리케이션의 가장 단순한 형태에서는 데이터베이스 서버를 사용하지 않습니다. 그 대신 클라이언트 데이터셋의 기능인 MyBase를 사용하여 파일에 저장하고 파일에서 데이터를 로드합니다. 이 아키텍처는 그림 18.3에서 설명합니다.

그림 18.3 파일 기반의 데이터베이스 애플리케이션



이 파일 기반의 방법을 사용할 때는 애플리케이션에서 클라이언트 데이터셋의 *SaveToFile* 메소드를 사용하여 디스크에 변경 내용을 기록합니다. *SaveToFile*은 하나의 매개변수를 취하는데 이 매개변수는 테이블을 포함하는 작성되거나 겹쳐 쓴 파일의 이름입니다. *SaveToFile* 메소드를 사용하여 이전에 작성한 테이블을 읽으려면 *LoadFromFile* 메소드를 사용하십시오. *LoadFromFile* 또한 하나의 매개변수를 취하는데 이 매개변수는 테이블을 포함하는 파일의 이름입니다.

항상 같은 파일에서 로드하고 저장하는 경우에는 *SaveToFile* 및 *LoadFromFile* 메소드 대신 *FileName* 속성을 사용할 수 있습니다. *FileName*이 유효한 파일 이름으로 설정된 경우 클라이언트 데이터셋이 열려 있으면 데이터가 자동적으로 파일에서 로드되고, 클라이언트 데이터셋이 닫혀 있으면 파일에 저장됩니다.

이 간단한 파일 기반의 아키텍처가 단일 티어 애플리케이션입니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

파일 기반 접근 방법의 장점은 단순함입니다. 클라이언트 데이터셋에 midas.so가 필요하기는 하지만 설치, 구성 또는 배포할 데이터베이스 서버가 없습니다. 사이트 사용권이나 데이터베이스 관리도 필요하지 않습니다.

또한 일부 버전의 C++Builder를 사용하면 임의의 XML 문서와 클라이언트 데이터셋에서 사용하는 데이터 패킷 간을 변환할 수 있습니다. 따라서 파일 기반의 접근 방법은 전용 데이터셋과 XML 문서로 작업할 수 있습니다. XML 문서와 클라이언트 데이터셋 데이터 패킷 간을 변환하는 데 대한 자세한 내용은 30장, "데이터베이스 애플리케이션에서 XML 사용"을 참조하십시오.

파일 기반의 접근 방법은 여러 사용자 지원을 제공하지 않습니다. 데이터셋은 애플리케이션 전용이어야 합니다. 데이터는 디스크에 있는 파일에 저장되었다가 나중에 로드되지만 기본적으로 여러 사용자가 각각 서로의 데이터 파일을 겹쳐 쓰는 것을 방지하는 보호 기능은 없습니다.

디스크에 저장된 클라이언트 데이터셋 사용에 대한 자세한 내용은 27-32페이지의 "파일 기반 데이터와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

다른 데이터셋 연결

BDE나 *dbExpress*를 사용하여 데이터베이스 서버에 연결하는 특화된 클라이언트 데이터셋이 있습니다. 이러한 특화된 클라이언트 데이터셋은 실제로 데이터와 내부 프로바이더 컴포넌트에 액세스하는 다른 데이터셋을 포함하는 컴포넌트를 내부적으로 결합하여 소스 데이터셋의 데이터를 패키지화하고 데이터베이스 서버로 다시 업데이트를 적용합니다. 이러한 복합 컴포넌트에는 약간의 추가 오버헤드가 필요하지만 장점도 있습니다.

- 클라이언트 데이터셋은 캐싱된 업데이트 작업을 할 수 있는 가장 강력한 방법을 제공합니다. 디폴트로, 다른 데이터셋 타입은 편집 내용을 데이터베이스 서버에 직접 포스트합니다. 업데이트를 로컬로 캐싱하고 이를 나중에 단일 트랜잭션으로 적용하는 데이터셋을 사용하여 네트워크 트래픽을 줄일 수 있습니다. 클라이언트 데이터셋을 사용하여 업데이트를 캐싱할 때의 장점에 대한 자세한 내용은 27-15페이지의 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용"을 참조하십시오.
- 클라이언트 데이터셋은 데이터베이스가 읽기 전용일 때 편집 내용을 직접 데이터베이스 서버에 적용합니다. *dbExpress*를 사용할 때는 이 방법이 데이터셋의 데이터를 편집하는 유일한 방법입니다. 또한 *dbExpress*를 사용할 때 데이터에서 자유롭게 탐색하는 한 방법이기도 합니다. *dbExpress*를 사용하지 않을 때도 일부 쿼리와 내장 프로시저의 결과는 읽기 전용입니다. 클라이언트 데이터셋을 사용하면 이러한 데이터를 편집할 수 있게 만드는 표준화된 방법을 제공합니다.
- 클라이언트 데이터셋은 디스크에 있는 전용 파일로 직접 작업할 수 있으므로 클라이언트 데이터셋을 파일 기반 모델과 함께 사용하여 유연성 있는 "Briefcase" 애플리케이션에 허용할 수 있습니다 . briefcase model 에 대한 자세한 내용은 18-14 페이지의 " 방법 조합 (Combining approaches)"을 참조하십시오.

이러한 특화된 클라이언트 데이터셋 외에도 내부 데이터셋과 데이터셋 프로바이더를 포함하지 않는 일반적인 클라이언트 데이터셋(*TClientDataSet*)이 있습니다. *TClientDataSet*에는 기본 데이터베이스 액세스 메커니즘이 없지만 데이터를 가져오거나 데이터를 주고 받는 다른 외부 데이터셋에 연결할 수 있습니다. 이 방법은 약간 복잡하지만 가끔 아주 적절할 때가 있습니다.

- 소스 데이터셋과 데이터셋 프로바이더는 서로 외부에 있으므로 데이터를 가져오고 업데이트를 적용하는 방법을 더욱 제어할 수 있습니다. 예를 들어, 프로바이더 컴포넌트는 특화된 클라이언트 데이터셋을 사용하여 데이터에 액세스할 때 사용할 수 없는 여러 가지 이벤트를 발생시킵니다.
- 소스 데이터셋이 서로 외부에 있으면 이 데이터셋을 마스터/디테일 관계로 다른 데이터셋에 연결할 수 있습니다. 외부 프로바이더는 자동으로 이 배열을 중첩된 디테일이 있는 단일 데이터셋으로 변환합니다. 소스 데이터셋이 서로 내부에 있으면 이러한 방법으로 중첩된 디테일을 작성할 수 없습니다.
- 외부 데이터셋에 클라이언트 데이터셋을 연결하는 것은 멀티 티어로 쉽게 확장할 수 있는 아키텍처입니다. 티어 수가 늘어나면 개발 프로세스가 더 복잡하고 비용도 많이 들기 때문에 애플리케이션 개발을 처음 시작할 때는 단일 티어나 2티어 애플리케이션으로 시작하는 것이 좋습니다. 데이터 양, 사용자 수 및 데이터에 액세스하는 다른 애플리케이션의 수가 늘어나면 나중에 멀티 티어 아키텍처로 확장해야 할 수 있습니다. 궁극적으로 멀티 티어 아키텍처를 사용할 계획을 갖고 있다면 외부 소스 데이터셋이 있는 클라이언트 데이터셋을 사용하여 시작하는 것이 효과적일 수 있습니다. 이러한 방법으로 데이터 액세스와 조작 로직을 미들 티어로 이동하면 애플리케이션이 증가해도 코드를 다시 사용할 수 있으므로 개발에 소요된 투자를 보호할 수 있습니다.
- *TClientDataSet*은 다른 모든 소스 데이터셋에 연결할 수 있습니다. 즉 해당되는 특화된 클라이언트 데이터셋이 없는 사용자 정의 데이터셋(서드파티 컴포넌트)을 사용할 수 있습니다. 일부 버전의 C++Builder에는 클라이언트 데이터셋을 다른 데이터셋이 아닌 XML 문서에 연결하는 특수 프로바이더 컴포넌트가 포함되어 있습니다. XML 문서에 연결하는 방법은 XML 프로바이더가 데이터셋 대신 XML 문서를 사용하는 것을 제외하면 클라이언트 데이터셋을 다른 데이터셋에 연결할 때와 같습니다. 이러한 XML 프로바이더에 대한 자세한 내용은 30-7페이지의 "XML 문서를 프로바이더의 소스로 사용"을 참조하십시오.

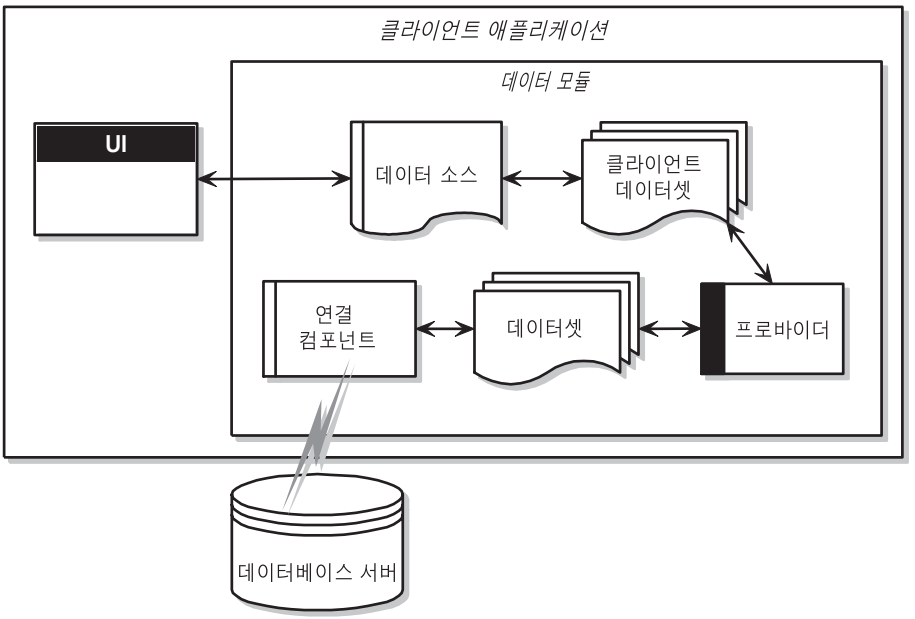
클라이언트 데이터셋을 외부 데이터셋에 연결하는 아키텍처에는 두 가지 버전이 있습니다.

- 클라이언트 데이터셋을 같은 애플리케이션의 다른 데이터셋에 연결
- 멀티 티어 아키텍처 사용

클라이언트 데이터셋을 같은 애플리케이션의 다른 데이터셋에 연결

프로바이더 컴포넌트를 사용하여 *TClientDataSet*을 다른(소스) 데이터셋에 연결할 수 있습니다. 프로바이더는 데이터베이스 정보를 클라이언트 데이터셋이 사용할 수 있는 전송 가능한 데이터 패킷으로 패키지화하고 클라이언트 데이터셋이 만든 델타 패킷에 수신된 업데이트를 다시 데이터베이스 서버에 적용합니다. 이를 위한 아키텍처는 그림 18.4에 설명되어 있습니다.

그림 18.4 클라이언트 데이터셋과 다른 데이터셋을 결합하는 아키텍처



데이터베이스 서버가 로컬 데이터베이스인지 원격 데이터베이스 서버인지에 따라 아키텍처가 단일 티어 또는 2 티어 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

클라이언트 데이터셋을 프로바이더에 연결하려면 해당 *ProviderName* 속성을 프로바이더 컴포넌트 이름으로 설정합니다. 프로바이더는 클라이언트 데이터셋과 동일한 데이터 모듈에 있습니다. 프로바이더를 소스 데이터셋에 연결하려면 해당 *DataSet* 속성을 설정하십시오.

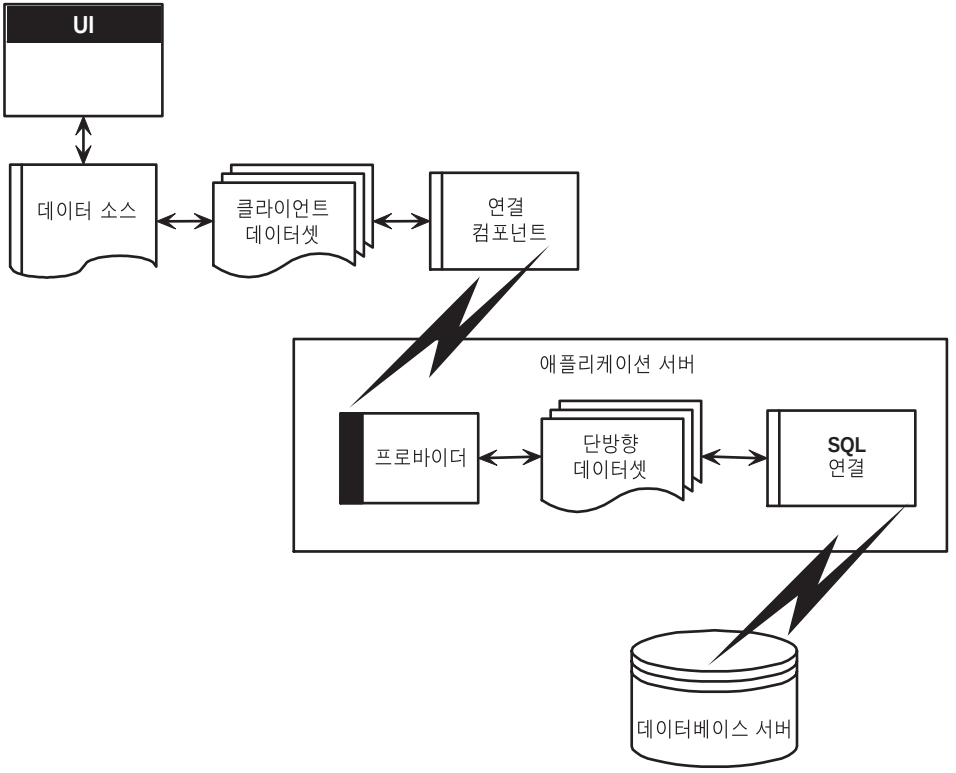
클라이언트 데이터셋이 프로바이더에 연결되고 프로바이더가 소스 데이터셋에 연결되면, 소스 데이터셋이 데이터베이스에 연결되어 있다는 가정 하에 이러한 컴포넌트들이 데이터베이스 레코드를 통해 가져오고, 표시하고, 탐색하는 데 필요한 모든 세부 사항을 자동으로 처리합니다. 사용자가 수정한 내용을 다시 데이터베이스에 적용하려면 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출하기만 하면 됩니다.

프로바이더와 클라이언트 데이터셋 사용에 대한 자세한 내용은 27-24페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

멀티 티어 아키텍처 사용

데이터베이스에 정보가 여러 테이블 간의 복합 관계가 포함되어 있거나 클라이언트 수가 증가할 때는 멀티 티어 모델을 사용할 수 있습니다. 멀티 티어 애플리케이션에는 클라이언트 애플리케이션과 데이터베이스 서버 사이에 미들 티어가 있습니다. 멀티 티어 애플리케이션의 아키텍처는 그림 18.5에 설명되어 있습니다.

그림 18.5 멀티 티어 데이터베이스 아키텍처



앞의 그림은 3티어 애플리케이션을 나타낸 것입니다. 데이터베이스 정보를 처리하는 로직은 각각의 시스템이나 티어에 있습니다. 이 미들 티어는 데이터베이스 상호 작용을 관리하는 로직을 중앙화하여 데이터베이스 작동을 중앙에서 제어합니다. 이렇게 하면 일관적인 데이터 로직을 유지하면서 다른 클라이언트 애플리케이션이 같은 데이터를 사용하게 할 수 있습니다. 또한 많은 프로세스가 미들 티어로 분담되기 때문에 보다 작은 클라이언트 애플리케이션도 만들 수 있습니다. 이러한 작은 클라이언트 애플리케이션은 설치, 구성 및 유지 보수가 더 쉽습니다. 또한 멀티 티어 애플리케이션은 데이터 처리 작업을 여러 시스템으로 분산시켜 성능을 향상시킬 수 있습니다.

멀티 티어 아키텍처는 이전 모델과 매우 비슷합니다. 이 아키텍처의 가장 큰 차이점은 소스 데이터셋과 클라이언트 데이터셋 간의 매개체 역할을 하는 프로바이더와 데이터베이스 서버에 연결하는 소스 데이터셋이 모두 각각의 애플리케이션으로 이동한다는 점입니다. 이러한 각각의 애플리케이션을 애플리케이션 서버(또는 "원격 데이터 브로커")라고 합니다.

프로바이더가 각각의 애플리케이션으로 이동했기 때문에 클라이언트 데이터셋은 더 이상 간단히 해당 *ProviderName* 속성을 설정하여 소스 데이터셋에 연결할 수 없습니다. 또한 애플리케이션 서버를 검색하고 연결하는 데 특정 타입의 연결 컴포넌트를 사용해야 합니다.

클라이언트 데이터셋을 애플리케이션 서버에 연결할 수 있는 연결 컴포넌트에는 여러 종류가 있습니다. 이 컴포넌트들은 모두 *TCustomRemoteServer*의 모든 자손이며 주로 사용하는 통신 프로토콜(TCP/IP, HTTP, DCOM, 또는 SOAP)에 따라 구분됩니다. *RemoteServer* 속성을 설정하여 클라이언트 데이터셋을 해당 연결 컴포넌트에 연결합니다.

연결 컴포넌트는 애플리케이션 서버와의 연결을 설정하고 클라이언트 데이터셋이 사용하는 인터페이스를 반환하여 해당 *ProviderName* 속성으로 지정된 프로바이더를 호출합니다. 클라이언트 데이터셋은 애플리케이션 서버를 호출할 때마다 *ProviderName*의 값을 전달하고 애플리케이션 서버는 프로바이더에 호출을 전달합니다.

애플리케이션 서버에 클라이언트 데이터셋을 연결하는 데 대한 자세한 내용은 29장, "멀티 티어 애플리케이션 생성"을 참조하십시오.

방법 조합(Combining approaches)

이전 단원에서는 데이터베이스 애플리케이션을 작성할 때 사용할 수 있는 여러 가지 아키텍처에 대해서 설명했습니다. 그러나 단일 애플리케이션에서 둘 이상의 사용 가능한 아키텍처를 조합할 수 있습니다. 실제로 조합을 통해 매우 강력한 기능을 발휘할 수도 있습니다.

예를 들어, 18-9페이지의 "디스크에 있는 전용 파일 사용"에서 설명한 디스크 기반의 아키텍처와 18-11페이지의 "클라이언트 데이터셋을 같은 애플리케이션의 다른 데이터셋에 연결" 또는 18-12페이지의 "멀티 티어 아키텍처 사용"에서 설명한 다른 방법을 조합할 수 있습니다. 모든 모델은 클라이언트 데이터셋을 사용하여 사용자 인터페이스에 나타나는 데이터를 표현하므로 이렇게 조합하는 것은 쉽습니다. 이 결과를 briefcase model(경우에 따라 연결 해제된 모델 또는 모바일 컴퓨팅)이라고 합니다.

briefcase model은 다음과 같은 상황에서 유용합니다. 온사이트에 있는 회사의 데이터베이스에는 영업 사원이 밖에서 사용하고 업데이트할 수 있는 고객 연락처 데이터가 포함되어 있습니다. 영업 사원은 온사이트에서 데이터베이스로부터 정보를 다운로드합니다. 나중에 외국으로 가거나 또는 기존 고객이나 새 고객 사이트의 레코드가 업데이트되면 랩톱에서 이 정보로 작업했다가 온사이트로 돌아오면 모든 사람들이 사용할 수 있도록 회사 데이터베이스에 데이터 변경 내용을 업로드합니다.

온사이트에서 사용할 때 briefcase model 애플리케이션의 클라이언트 데이터셋은 프로바이더의 데이터를 가져옵니다. 따라서 클라이언트 데이터셋은 데이터베이스 서버에 연결되어 프로바이더를 통해 서버 데이터를 가져오고 업데이트 내용을 다시 서버로 보냅니다. 프로바이더와의 연결을 끊기 전에 클라이언트 데이터셋은 해당 정보 스냅샷을 디스크에 있는 파일에 저장합니다. 오프사이트에서는 클라이언트 데이터셋이 파일에서 해당 데이터를 로드한 다음 변경 내용을 다시 해당 파일에 저장합니다. 마지막으로, 다시 온사이트에서 클라이언트 데이터셋은 프로바이더에 다시 연결하여 업데이트된 내용을 데이터베이스 서버에 적용하거나 데이터 스냅샷을 새로 고칠 수 있습니다.

사용자 인터페이스 디자인

컴포넌트 팔레트의 Data Controls 페이지에서는 데이터베이스 레코드에 있는 필드의 데이터를 나타내는 일련의 데이터 인식 컨트롤을 제공하며 사용자가 데이터를 편집하고 변경된 내용을 다시 데이터베이스로 포스트할 수 있게 합니다. 데이터 인식 컨트롤을 사용하여 사용자가 정보를 보고 액세스할 수 있도록 데이터베이스 애플리케이션의 사용자 인터페이스(UI)를 생성합니다. 데이터 인식 컨트롤에 대한 자세한 내용은 19장, "데이터 컨트롤 사용"을 참조하십시오.

기본 데이터 컨트롤 외에도 다른 요소를 사용자 인터페이스에 도입할 수 있습니다.

- 애플리케이션에서 데이터베이스에 있는 데이터를 분석하게 할 수 있습니다. 데이터 분석 애플리케이션은 데이터를 단지 데이터베이스에 표시하는 것 이상의 작업을 수행하며 사용자가 해당 데이터의 효과를 쉽게 알 수 있도록 유용한 형식으로 정보를 요약합니다.
- 보고서를 인쇄하여 사용자 인터페이스에 표시된 정보를 하드 카피로 제공할 수 있습니다.
- 웹 브라우저에서 볼 수 있는 사용자 인터페이스를 만들 수 있습니다. 가장 간단한 웹 기반 데이터베이스 애플리케이션에 대해서는 33-17페이지의 "응답에서 데이터베이스 정보 사용"에서 설명합니다. 또한 "웹 기반 클라이언트 애플리케이션 생성"의 설명처럼 웹 기반 방법과 멀티 티어 아키텍처를 함께 사용할 수도 있습니다.

데이터 분석

일부 데이터베이스 애플리케이션에서는 데이터베이스의 정보가 사용자에게 직접 표시되지 않습니다. 대신 사용자가 이 데이터를 사용하여 의사를 결정할 수 있도록 데이터베이스의 정보를 분석하고 요약합니다.

컴포넌트 팔레트의 Data Controls 페이지에 있는 *TDBChart* 컴포넌트를 사용하면 사용자가 데이터베이스 정보의 임포트를 쉽게 이해할 수 있도록 데이터베이스 정보를 그래프 형태로 나타낼 수 있습니다.

또한 일부 버전의 C++Builder에는 컴포넌트 팔레트에 Decision Cube 페이지가 포함되어 있습니다. 이 페이지에는 decision 지원 애플리케이션을 작성할 때 데이터를 분석하고 크로스탭을 수행할 수 있는 6개의 컴포넌트가 포함되어 있습니다. Decision Cube 컴포넌트 사용에 대한 자세한 내용은 20장, "decision 지원 컴포넌트 사용"을 참조하십시오.

다양한 그룹화 조건에 따라 데이터 요약을 표시하는 새로운 컴포넌트를 생성하려면 클라이언트 데이터셋에 유지 보수된 집계를 사용할 수 있습니다. 유지 보수된 집계에 대한 자세한 내용은 27-11페이지의 "유지 보수된 집계 사용"을 참조하십시오.

보고서 작성

사용자가 애플리케이션의 데이터셋에서 데이터베이스 정보를 인쇄하게 하려면 컴포넌트 팔레트의 QReport 페이지에서 보고서 컴포넌트를 사용할 수 있습니다. 이러한 컴포넌트를 사용하여 데이터베이스 테이블의 정보를 표시하고 요약하는 개요 보고서를 비주얼하게 생성할 수 있습니다. 그룹 헤더와 바닥글에 요약을 추가하여 그룹화 조건을 기준으로 데이터를 분석할 수 있습니다.

New Items 다이얼로그 박스에서 QuickReport 아이콘을 선택하여 애플리케이션에 대한 보고서를 시작합니다. 메인 메뉴에서 File|New|Other를 선택하고 Business라는 레이블의 페이지로 이동합니다. QuickReport Wizard 아이콘을 더블 클릭하여 마법사를 시작합니다.

참고 QReport 페이지의 컴포넌트를 사용하는 방법의 예는 C++Builder 에서 제공한 QuickReport 데모를 참조하십시오.

장

19

데이터 컨트롤 사용

컴포넌트 팔레트의 Data Controls 페이지에서는 데이터베이스 레코드 필드의 데이터를 나타내고 데이터셋에서 허용하면 사용자가 해당 데이터를 편집하여 변경 내용을 다시 데이터베이스로 포스트할 수 있게 해주는 데이터 인식 컨트롤 집합을 제공합니다. 데이터베이스 애플리케이션의 폼에 데이터 컨트롤을 두면 데이터베이스 애플리케이션의 사용자 인터페이스(UI)를 구축하여 개발자가 정보를 보고 액세스할 수 있습니다.

사용자 인터페이스에 추가하는 데이터 인식 컨트롤은 다음과 같은 여러 가지 요인에 따라 달라집니다.

- 표시하는 데이터 타입. 일반 텍스트를 표시하고 편집하기 위한 컨트롤과 서식화된 텍스트를 사용하는 컨트롤, 그래픽용 컨트롤, 멀티미디어 요소 등에서 선택할 수 있습니다. 다양한 정보 타입을 표시하는 컨트롤은 19-7페이지의 "단일 레코드 표시"에서 설명합니다.
- 정보를 구성하는 방법. 화면의 단일 레코드에서 정보를 표시하거나 그리드를 사용하는 여러 레코드에서 정보를 나열하는 방법 중에서 선택할 수 있습니다. 19-7페이지의 "데이터 구성 방법 선택"에서는 몇 가지 가능성을 설명합니다.
- 데이터를 컨트롤에 제공하는 데이터셋 타입. 원본으로 사용하는 데이터셋의 제약을 반영하는 컨트롤을 사용해야 할 경우가 있습니다. 예를 들어, 단방향 데이터셋에서는 한 번에 한 레코드만 제공할 수 있으므로 단방향 데이터셋을 사용할 경우에는 그리드를 사용하지 않습니다.
- 사용자가 데이터셋 레코드를 탐색하고 데이터를 추가하거나 편집할 수 있도록 할지 여부 또는 방법. 고유의 컨트롤이나 메커니즘을 추가하여 탐색 및 편집을 하려는 경우나 데이터 탐색기와 같은 기본 컨트롤을 사용하려는 경우가 있습니다. 데이터 탐색기 사용에 대한 자세한 내용은 19-28페이지의 "레코드 탐색 및 처리"를 참조하십시오.

참고 decision 지원을 위한 다소 복잡한 데이터 인식 컨트롤은 20장, "decision 지원 컴포넌트 사용"에서 설명합니다.

사용자 인터페이스에 추가하려고 선택한 데이터 인식 컨트롤 종류에 상관없이 특정 공통 기능이 적용됩니다. 이러한 내용은 아래에서 설명합니다.

공통 데이터 컨트롤 기능 사용

다음 작업은 대부분의 데이터 컨트롤에서 공통적으로 사용되는 기능입니다.

- 데이터 컨트롤을 데이터셋과 연결
- 데이터 편집 및 업데이트
- 데이터 표시 활성화 및 비활성화
- 데이터 표시 새로 고침
- 마우스, 키보드 및 타이머 이벤트 활성화

데이터 컨트롤을 사용하면 데이터셋의 현재 레코드와 연결된 데이터 필드를 표시하고 편집할 수 있습니다. 표 19.1은 컴포넌트 팔레트의 Data Controls 페이지에 나타난 데이터 컨트롤을 요약한 것입니다.

표 19.1 데이터 컨트롤

데이터 컨트롤	설명
TDBGrid	데이터 소스의 정보를 테이블 형식으로 표시합니다. 그리드의 열은 원본으로 사용하는 테이블이나 쿼리 데이터셋의 열에 해당하며 그리드의 행은 레코드에 해당합니다.
TDBNavigator	레코드 업데이트, 레코드 포스트, 레코드 삭제, 레코드 편집 취소 및 데이터 표시를 새로 고치면서 데이터셋의 데이터 레코드를 탐색합니다.
TDBText	필드의 데이터를 레이블로 표시합니다.
TDBEdit	필드의 데이터를 에디트 박스에 표시합니다.
TDBMemo	메모나 BLOB 필드의 데이터를 스크롤 가능한 여러 줄 에디트 박스에 표시합니다.
TDBImage	데이터 필드의 그래픽을 그래픽 박스에 표시합니다.
TDBListBox	현재 데이터 레코드의 필드를 업데이트할 항목 리스트를 표시합니다.
TDBComboBox	필드를 업데이트할 항목 리스트를 표시하고 표준 데이터 인식 에디트 박스와 같이 텍스트 직접 입력도 허용합니다.
TDBCheckBox	부울 필드 값을 나타내는 체크 박스를 표시합니다.
TDBRadioGroup	필드의 상호 배타적 옵션을 표시합니다.
TDBLookupListBox	필드 값을 기반으로 다른 데이터셋에서 알아낸 항목 리스트를 표시합니다.
TDBLookupComboBox	필드 값을 기반으로 다른 데이터셋에서 알아낸 항목 리스트를 표시하고 표준 데이터 인식 에디트 박스처럼 텍스트 직접 입력도 허용합니다.
TDBCtrlGrid	그리드 내에서 구성 가능한, 데이터 인식 컨트롤의 반복되는 집합을 표시합니다.
TDBRichEdit	필드의 서식화된 데이터를 에디트 박스에 표시합니다.

데이터 컨트롤은 디자인 타임 시 데이터 인식입니다. 애플리케이션을 생성하는 동안 활성 데이터셋과 데이터 컨트롤을 연결하면 컨트롤에서 바로 라이브 데이터를 볼 수 있습니다. Fields Editor를 사용하면 애플리케이션을 컴파일하고 실행하지 않아도 디자인 타임 시 데이터셋을 스크롤하여 애플리케이션에서 데이터를 올바르게 표시하는지 확인할 수 있습니다. Fields Editor에 대한 자세한 내용은 23-4페이지의 "영구적 필드(persistent field) 생성"을 참조하십시오.

런타임 시 데이터 컨트롤은 데이터를 표시합니다. 애플리케이션, 컨트롤 및 데이터셋 모두에서 이 데이터를 허용하는 경우 사용자는 컨트롤을 통해 데이터를 편집할 수 있습니다.

데이터 컨트롤을 데이터셋과 연결

데이터 컨트롤은 데이터 소스를 사용하여 데이터셋에 연결합니다. 데이터 소스 컴포넌트(*TDataSource*)는 컨트롤과 데이터가 포함된 데이터셋 사이의 매개체 역할을 합니다. 데이터를 표시하고 처리할 수 있도록 각 데이터 인식 컨트롤을 데이터 소스 컴포넌트에 연결해야 합니다. 마찬가지로 폼의 데이터 인식 컨트롤에서 해당 데이터를 표시하고 처리할 수 있도록 모든 데이터셋을 데이터 소스 컴포넌트에 연결해야 합니다.

참고 데이터 소스 컴포넌트는 마스터/디테일 관계에서 중첩되지 않은 데이터셋을 연결할 때도 필요합니다.

다음과 같은 방법으로 데이터 컨트롤을 데이터셋에 연결합니다.

1. 데이터셋을 데이터 모듈이나 폼에 두고 해당 속성을 적절하게 설정합니다.
2. 데이터 소스를 같은 데이터 모듈이나 폼에 둡니다. Object Inspector를 사용하여 *DataSet* 속성을 단계 1에서 두었던 데이터셋으로 설정합니다.
3. 컴포넌트 팔레트 Data Access 페이지의 데이터 컨트롤을 폼에 둡니다.
4. Object Inspector를 사용하여 컨트롤의 *DataSource* 속성을 단계 2에서 두었던 데이터 소스 컴포넌트로 설정합니다.
5. 컨트롤의 *DataField* 속성을 표시할 필드 이름으로 설정하거나 속성의 드롭다운 리스트에서 필드 이름을 선택합니다 . *TDBGrid*, *TDBCtrlGrid* 및 *TDBNavigator* 는 데이터셋에 있는 모든 사용 가능한 필드에 액세스할 수 있으므로 이러한 단계가 적용되지 않습니다.
6. 데이터셋의 *Active* 속성을 **true**로 설정하여 컨트롤의 데이터를 표시합니다.

런타임 시 연결 데이터셋 변경

앞의 예제에서 데이터 소스는 디자인 타임 시 *DataSet* 속성을 설정하여 데이터셋과 연결되었습니다. 런타임 시 데이터 소스 컴포넌트의 데이터셋을 필요에 따라 전환할 수 있습니다. 예를 들어, 다음 코드에서는 *Customers*와 *Orders*라는 데이터셋 컴포넌트 사이에서 *CustSource* 데이터 소스 컴포넌트의 데이터셋을 바꿉니다.

```
if (CustSource->DataSet == Customers)
  CustSource->DataSet = Orders;
else
  CustSource->DataSet = Customers;
```

DataSet 속성을 다른 폼의 데이터셋으로 설정하여 두 개의 폼에 있는 데이터 컨트롤을 동기화할 수도 있습니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  DataSource1->DataSet = Form1->Table1;
}
```

데이터 소스 활성화 및 비활성화

데이터 소스에는 데이터셋에 연결되었는지 확인하는 *Enabled* 속성이 있습니다. *Enabled*가 **true**이면 데이터 소스가 데이터셋에 연결됩니다.

*Enabled*를 **false**로 설정하여 단일 데이터 소스와 데이터셋의 연결을 일시적으로 끊을 수 있습니다. *Enabled*가 **false**이면 데이터 소스 컴포넌트에 연결된 모든 데이터 컨트롤은 비워지며 *Enabled*가 **true**로 설정될 때까지 비활성됩니다. 하지만 데이터셋 컴포넌트의 *DisableControls*와 *EnableControls* 메소드는 연결된 모든 데이터 소스에 영향을 미치므로 이러한 메소드를 통해 데이터셋에 대한 액세스를 제어하는 것이 좋습니다.

데이터 소스에 따른 변경 내용에 대한 응답

데이터 소스는 데이터 컨트롤과 해당 데이터셋 사이에 연결을 제공하므로 둘 사이에 발생하는 모든 통신을 조정합니다 . 일반적으로 데이터 인식 컨트롤은 자동으로 데이터셋의 변경에 응답합니다. 하지만 사용자 인터페이스에서 데이터 인식가 아닌 컨트롤을 사용하고 있으면 데이터 소스 컴포넌트의 이벤트를 사용하여 동일한 종류의 응답을 수동으로 제공할 수 있습니다.

OnDataChange 이벤트는 필드 편집을 비롯하여 레코드의 데이터가 변경되거나 커서가 새 레코드로 이동할 때 발생합니다. 이 이벤트는 변경 사항이 있을 때마다 항상 실행되므로 컨트롤이 데이터셋의 현재 필드 값을 반영하는지 확인할 때 유용합니다. 일반적으로 *OnDataChange* 이벤트 핸들러는 필드 데이터를 표시하는, 데이터 인식이 아닌 컨트롤 값을 새로 고칩니다.

OnUpdateData 이벤트는 현재 레코드의 데이터가 포스트되려고 할 때 발생합니다. 예를 들어, *OnUpdateData* 이벤트는 *Post*를 호출한 후에 발생하지만 데이터가 실제로 원본으로 사용하는 데이터베이스 서버나 로컬 캐시로 포스트되기 전에 이 이벤트가 발생합니다.

데이터셋의 상태가 변경되면 *OnStateChange* 이벤트가 발생합니다. 이 이벤트가 발생하면 데이터셋의 *State* 속성을 검사하여 현재 상태를 확인할 수 있습니다.

예를 들어, 다음 *OnStateChange* 이벤트 핸들러는 현재 상태를 기반으로 버튼이나 메뉴 항목을 활성화하거나 비활성화합니다.

```
void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
  CustTableActivateBtn->Enabled = (CustTable->State == dsInactive);
  CustTableEditBtn->Enabled = (CustTable->State == dsBrowse);
  CustTableCancelBtn->Enabled = (CustTable->State == dsInsert ||
                                 CustTable->State == dsEdit ||
                                 CustTable->State == dsSetKey);
  f
}
```

참고 데이터셋 상태에 대한 자세한 내용은 22-2페이지의 "데이터셋 상태 결정"을 참조하십시오.

데이터 편집 및 업데이트

탐색기를 제외한 모든 데이터 컨트롤에서는 데이터베이스 필드의 데이터를 표시합니다. 또한 원본으로 사용하는 데이터셋에서 허용하는 한 데이터 컨트롤을 사용하여 데이터를 편집하고 업데이트할 수 있습니다.

참고 단방향 데이터셋에서는 사용자가 데이터를 편집하고 업데이트할 수 없습니다.

사용자 엔트리의 컨트롤에서 편집 활성화

데이터셋은 데이터 편집을 허용할 수 있도록 *dsEdit* 상태에 있어야만 합니다. 데이터 소스의 *AutoEdit* 속성이 **true**(기본값)이면 사용자가 데이터를 편집하려고 시도하자마자 데이터 컨트롤에서는 데이터셋을 *dsEdit* 모드로 넣는 작업을 합니다.

*AutoEdit*가 **false**이면 데이터셋을 편집 모드로 설정할 대체 메커니즘을 제공해야 합니다. 이러한 메커니즘 중 하나는 사용자가 명시적으로 데이터셋을 편집 모드로 설정해주는 *Edit* 버튼과 함께 *TDBNavigator* 컨트롤을 사용하는 것입니다. *TDBNavigator*에 대한 자세한 내용은 19-28페이지의 "레코드 탐색 및 처리"를 참조하십시오. 또한 데이터셋을 편집 모드로 설정해야 할 때 데이터셋의 *Edit* 메소드를 호출하는 코드를 작성할 수 있습니다.

컨트롤에서 데이터 편집

데이터셋의 *CanModify* 속성이 **true**인 경우에만 데이터 컨트롤에서 편집 내용을 연결된 데이터셋으로 포스트할 수 있습니다. 단방향 데이터셋에서는 *CanModify*가 항상 **false**입니다. 일부 데이터셋에는 *CanModify*가 **true**인지 지정할 수 있게 해주는*ReadOnly* 속성이 있습니다.

참고 데이터셋에서 데이터를 업데이트할 수 있는지는 원본으로 사용하는 데이터베이스 테이블에서 업데이트를 허용하는지에 따라 달라집니다.

데이터셋의 *CanModify* 속성이 **true** 이더라도 컨트롤에서 데이터베이스 테이블로 업데이트를 다시 포스트하기 전에, 데이터셋을 컨트롤에 연결하는 데이터 소스의 *Enabled* 속성도 **true**가 되어야 합니다. 데이터 소스의 *Enabled* 속성은 컨트롤에서 데이터셋의 필드 값을 표시할 수 있는지 여부와 사용자가 값을 편집하여 포스트할 수 있는지 여부를 결정합니다. *Enabled*가 **true**(기본값)인 경우 컨트롤에서 필드 값을 표시할 수 있습니다.

마지막으로 컨트롤에 표시된 데이터에 사용자가 편집 내용을 입력할 수 있는지 여부를 제어할 수 있습니다. 데이터 컨트롤의 *ReadOnly* 속성은 컨트롤에서 표시한 데이터를 사용자가 편집할 수 있는지 여부를 결정할 수 있습니다. **false**(기본값)이면 사용자는 데이터를 편집할 수 있습니다. 데이터셋의 *CanModify* 속성이 **false**이면 확실하게 컨트롤의 *ReadOnly* 속성이 **true**인지 확인해야 합니다. 그렇지 않으면 원본으로 사용하는 데이터베이스 테이블의 데이터에 잘못된 영향을 미칠 수 있습니다.

*TDBGrid*를 제외한 모든 데이터 컨트롤에서 필드를 수정할 때 컨트롤에서 *Tab* 키를 누르면 수정 내용이 원본으로 사용하는 데이터셋으로 복사됩니다. 필드에서 *Tab* 키를 누르기 전에 *Esc* 키를 누르면 데이터 컨트롤에서 수정을 중단하고 수정하기 전의 가졌던 값으로 필드 값을 되돌립니다.

*TDBGrid*에서 다른 레코드로 이동하면 수정 내용이 포스트됩니다. 다른 레코드로 이동하기 전에 필드의 아무 레코드에서나 *Esc* 키를 눌러 레코드에 대한 모든 변경 내용을 취소할 수 있습니다.

레코드가 포스트될 때 C++Builder 는 상태의 변화에 대해 데이터셋과 연결된 모든 데이터 인식 컨트롤을 검사합니다. 수정된 데이터가 포함된 필드를 업데이트할 때 문제가 있으면 C++Builder는 예외를 발생시키고 레코드가 수정되지 않습니다.

참고 예를 들어, 클라이언트 데이터셋을 사용하여 애플리케이션에서 업데이트를 캐싱하면 모든 수정 내용은 내부 캐시에 포스트됩니다. 데이터셋의 *ApplyUpdates* 메소드를 호출할 때까지 이러한 수정 내용은 원본으로 사용하는 데이터베이스 테이블에 적용되지 않습니다.

데이터 표시 활성화 및 비활성화

애플리케이션에서 데이터셋을 반복하거나 검색을 수행하는 경우 현재 레코드가 변경될 때마다 데이터 인식 컨트롤에 표시된 값을 새로 고치는 작업을 일시적으로 할 수 없게 해야 합니다. 값을 새로 고칠 수 없게 하면 반복이나 검색 속도가 빨라지고 성가신 화면 깜빡임이 없어집니다.

DisableControls 는 데이터셋에 연결된 모든 데이터 인식 컨트롤에 대한 표시를 비활성화하는 데이터셋 메소드입니다. 반복이나 검색이 끝나자마자 애플리케이션에서 바로 데이터셋의 *EnableControls* 메소드를 호출해야 합니다.

일반적으로는 반복 프로세스를 입력하기 전에 컨트롤을 사용할 수 없게 만듭니다. 처리 중에 예외가 발생하더라도 컨트롤을 다시 사용 가능으로 만들 수 있도록 반복 프로세스 자체는 **try...__finally** 문 안에서 발생해야 합니다. **__finally** 절은 *EnableControls* 를 호출해야 합니다. 다음 코드에서는 이런 식으로 *DisableControls*와 *EnableControls*를 사용할 수 있는 방법을 설명합니다.

```
CustTable->DisableControls();
try
{
  // cycle through all records of the dataset
  for (CustTable->First(); !CustTable->EOF; CustTable->Next())
  {
  // Process each record here
  }
}
__finally
{
  CustTable->EnableControls();
}
```

데이터 표시 새로 고침

데이터셋의 *Refresh* 메소드는 로컬 버퍼를 플러시하고 개방형 데이터셋의 데이터를 다시 가져옵니다. 다른 애플리케이션에서 개발자 애플리케이션이 사용하는 데이터에 동시에 액세스하였기 때문에 원본으로 사용하는 데이터가 변경되었다고 생각되면 이 메소드를 사용하여 데이터 인식 컨트롤의 표시를 업데이트할 수 있습니다. 캐싱된 업데이트를 사용하는 경우 데이터셋을 새로 고치기 전에 데이터셋에서 현재 캐싱한 업데이트를 모두 적용해야 합니다.

새로 고침 기능은 때때로 예기치 않은 결과를 초래합니다. 예를 들어, 사용자가 다른 애플리케이션에서 삭제한 레코드를 보고 있는 경우 해당 애플리케이션에서 *Refresh*를 호출하면 해당 레코드가 사라집니다. 그리고 개발자가 데이터를 가져온 다음 *Refresh*를 호출하기 전에 다른 사용자가 레코드를 변경하면 데이터가 변경을 위해 표시될 수도 있습니다.

마우스, 키보드 및 타이머 이벤트 활성화

데이터 컨트롤의 *Enabled* 속성은 마우스나 키보드, 타이머 이벤트에 응답할지 여부를 결정한 다음 정보를 데이터 소스에 전달합니다. 이러한 속성의 디폴트 설정은 **true**입니다.

마우스나 키보드, 타이머 이벤트가 데이터 컨트롤에 도달하는 것을 막으려면 데이터 컨트롤의 *Enabled* 속성을 **false**로 설정합니다. *Enabled*가 **false**이면 컨트롤을 데이터셋에 연결하는 데이터 소스는 데이터 컨트롤로부터 정보를 받지 못합니다. 데이터 컨트롤에서는 데이터를 계속 표시하지만 컨트롤에 표시된 텍스트는 희미해집니다.

데이터 구성 방법 선택

개발자 데이터베이스 애플리케이션의 사용자 인터페이스를 구축할 때 정보 표시 및 해당 정보를 처리하는 컨트롤의 구성 방법을 선택합니다.

처음으로 결정해야 할 내용은 한 번에 한 레코드를 표시할 것인지 여러 레코드를 표시할 것인지 여부입니다.

또한 컨트롤을 추가하여 레코드를 탐색하고 처리해야 할 수도 있습니다. *TDBNavigator* 컨트롤은 수행하고자 하는 많은 함수에 대한 기본 지원을 제공합니다.

단일 레코드 표시

한 번에 데이터의 한 레코드에 관한 정보를 제공하는 애플리케이션이 많이 있습니다. 예를 들어, 주문 입력 애플리케이션은 현재 로그한 다른 주문을 표시하지 않고 단일 주문에 대한 정보만 표시합니다. 이 정보는 주문 데이터셋의 단일 레코드에서 온 것일 수 있습니다.

단일 레코드를 표시하는 애플리케이션의 경우 데이터베이스 정보는 모두 같은 내용(이전 예제에서는 같은 주문)에 대한 것이므로 대개 읽고 이해하기 쉽습니다. 이러한 사용자 인터페이스의 데이터 인식 컨트롤은 데이터베이스 레코드의 단일 필드를 나타냅니다 . 컴포넌트 팔레트의 Data Controls 페이지에서는 다양한 종류의 필드를 나타내는 컨트롤을 광범위하게 선택할 수 있습니다. 이러한 컨트롤은 일반적으로 컴포넌트 팔레트에서 사용할 수 있는 다른 컨트롤의 데이터 인식 버전입니다. 예를 들어, *TDBEdit* 컨트롤은 사용자가 텍스트 문자열을 보고 편집할 수 있게 해주는 표준 *TEdit* 컨트롤의 데이터 인식 버전입니다.

사용하는 컨트롤 종류는 필드에 포함된 데이터 타입(텍스트, 서식화된 텍스트, 그래픽 및 부울 정보 등)에 따라 달라집니다.

데이터를 레이블로 표시

TDBText 는 컴포넌트 팔레트의 Standard 페이지에 있는 *TLabel* 컴포넌트와 유사한 읽기 전용 컨트롤입니다. *TDBText* 컨트롤은 다른 컨트롤에서 사용자 입력을 허용하는 폼에 표시 전용 데이터를 제공해야 할 때 유용합니다. 예를 들어, 고객 리스트 테이블의 필드에 대해 폼을 만든다고 가정하면 사용자가 폼에 시/도, 구/군/시 등의 정보를 입력한 다음에는 동적 조회를 사용하여 각각의 테이블에서 우편 번호 필드를 자동으로 확인하려고 할 것입니다. 우편 번호 테이블에 연결된 *TDBText* 컴포넌트를 사용하면 사용자가 입력한 주소와 일치하는 우편 번호 필드를 표시할 수 있습니다.

*TDBText*는 데이터셋의 현재 레코드에 있는 특정 필드에서 표시할 텍스트를 가져옵니다. *TDBText*는 데이터셋에서 텍스트를 가져오므로 표시하는 텍스트가 동적입니다. 즉, 사용자가 데이터베이스 테이블을 탐색함에 따라 텍스트가 변경됩니다. 그러므로 *TLabel*에서처럼 디자인 타임 시 *TDBText*의 표시 텍스트를 지정할 수 없습니다.

참고 *TDBText* 컴포넌트를 폼에 둘 때는 다양한 너비의 데이터를 표시할 수 있도록 컨트롤 자체에서 크기를 조정할 수 있게 *AutoSize* 속성을 **true**(기본값)로 설정해야 합니다. *AutoSize*가 **false**이고 컨트롤이 너무 작으면 데이터 표시가 잘립니다.

에디트 박스에서 필드 표시 및 편집

*TDBEdit*은 에디트 박스 컴포넌트의 데이터 인식 버전입니다. *TDBEdit*은 자신이 연결된 데이터 필드의 현재 값을 표시하고 표준 에디트 박스 기술을 사용하여 편집할 수 있게 해줍니다.

예를 들어, *CustomersSource*가 *CustomersTable*이라는 개방형 *TClientDataSet*에 연결된 활성 *TDataSource* 컴포넌트라고 가정합니다. 그리고 폼에 *TDBEdit* 컴포넌트를 둔 다음 다음과 같이 속성을 설정합니다.

- *DataSource*: CustomersSource
- *DataField*: CustNo

데이터 인식 에디트 박스는 디자인 타임과 런타임에 모두 *CustomersTable* 데이터셋에 있는 *CustNo* 열의 현재 행 값을 표시합니다.

메모 컨트롤에서 텍스트 표시 및 편집

*TDBMemo*는 긴 텍스트 데이터를 표시할 수 있는 표준 *TMemo* 컴포넌트와 유사한 데이터 인식 컴포넌트입니다. *TDBMemo*는 여러 줄 텍스트를 표시할 뿐 아니라 사용자가 여러 줄 텍스트도 입력할 수 있게 해줍니다. *TDBMemo* 컨트롤을 사용하여 큰 바이너리 객체(BLOB) 필드에 포함된 큰 텍스트 필드나 텍스트 데이터를 표시할 수 있습니다.

디폴트로, *TDBMemo*에서는 사용자의 메모 텍스트 수정을 허용합니다. 수정을 방지하려면 메모 컨트롤의 *ReadOnly* 속성을 **true**로 설정합니다. 탭을 표시하고 사용자가 메모에서 입력할 수 있게 허용하려면 *WantTabs* 속성을 **true**로 설정합니다. 사용자가 입력할 수 있는 데이터베이스 메모에 입력할 수 있는 문자 수를 제한하려면 *MaxLength* 속성을 사용해야 합니다. *MaxLength*의 기본값이 0이면 운영 체제에서 부여한 문자 제한 이외에는 다른 문자 제한이 없다는 의미입니다.

데이터베이스 메모 표시 방법과 텍스트 입력 방법에 영향을 미치는 속성은 몇 가지가 있습니다. *ScrollBars* 속성을 사용하여 메모에 스크롤 막대를 제공할 수 있습니다. 자동 줄 바꿈이 안되게 하려면 *WordWrap* 속성을 **false**로 설정합니다. *Alignment* 속성은 컨트롤 내에서 텍스트를 정렬하는 방법을 결정합니다. 정렬 방법은 *taLeftJustify*(기본값), *taCenter* 및 *taRightJustify* 중 하나입니다. 텍스트의 글꼴을 변경하려면 *Font* 속성을 사용합니다.

런타임 시 사용자는 텍스트를 데이터베이스 컨트롤에서 자르고 복사하여 붙일 수 있습니다. *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 사용하여 프로그래밍 방식으로 같은 작업을 완료할 수 있습니다.

*TDBMemo*는 많은 양의 데이터를 표시할 수 있으므로 런타임 시 표시할 내용을 채우는 데 시간이 많이 걸립니다. 데이터 레코드를 스크롤하는 데 걸리는 시간을 줄이기 위해 *TDBMemo*에는 액세스된 데이터를 자동으로 표시해야 할지를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 **false**로 설정하면 *TDBMemo*에서는 실제 데이터가 아닌 필드 이름을 표시합니다. 실제 데이터를 표시하려면 컨트롤 내부를 더블 클릭합니다.

리치 에디트(rich edit) 메모 컨트롤에서 텍스트 표시 및 편집

*TDBRichEdit*는 큰 바이너리 객체(BLOB) 필드에 저장된 서식화된 텍스트를 표시할 수 있는 표준 *TRichEdit* 컴포넌트와 마찬가지로 데이터 인식 컴포넌트입니다. *TDBRichEdit*는 서식화된, 여러 줄 텍스트를 표시하므로 서식화된 여러 줄의 텍스트를 사용자가 입력할 수 있게 해줍니다.

참고 *TDBRichEdit*는 서식있는 텍스트를 입력하고 사용할 수 있는 속성과 메소드를 제공하지만, 이러한 서식 지정 옵션을 사용자가 사용할 수 있게 해주는 사용자 인터페이스 컴포넌트는 제공하지 않습니다. 서식있는 텍스트 기능을 사용할 수 있게 해주는 사용자 인터페이스를 애플리케이션에서 구현해야 합니다.

디폴트로, *TDBRichEdit*에서는 사용자의 메모 텍스트 수정을 허용합니다. 수정을 방지하려면 리치 에디트(rich edit) 컨트롤의 *ReadOnly* 속성을 **true**로 설정합니다. 탭을 표시하여 사용자가 탭을 메모에 입력할 수 있게 하려면 *WantTabs* 속성을 **true**로 설정합니다. 사용자가 데이터베이스 메모에 입력할 수 있는 문자 수를 제한하려면 *MaxLength* 속성을 사용합니다. MaxLength의 기본값이 0이라는 것은 운영 체제에서 부여한 문자 제한 이외에는 다른 문자 제한이 없다는 의미입니다.

*TDBRichEdit*는 많은 양의 데이터를 표시할 수 있으므로 런타임 시 표시할 내용을 채우는 데 시간이 많이 걸립니다. 데이터 레코드를 스크롤하는 데 걸리는 시간을 줄이기 위해 *TDBRichEdit*에는 액세스된 데이터를 자동으로 표시해야 할지를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 **false**로 설정하면 *TDBRichEdit*에서는 실제 데이터가 아닌 필드 이름을 표시합니다. 실제 데이터를 표시하려면 컨트롤 내부를 더블 클릭합니다.

이미지 컨트롤에서 그래픽 필드 표시 및 편집

*TDBImage*는 BLOB 필드에 포함된 그래픽을 표시하는 데이터 인식 컨트롤입니다.

디폴트로, *TDBImage*에서는 *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 사용하여 사용자가 클립보드의 내용을 자르고 붙여넣어 그래픽 이미지를 수정할 수 있게 해줍니다. 대신 컨트롤의 이벤트 핸들러에 연결된 사용자 고유의 편집 메소드를 제공할 수도 있습니다.

디폴트로, 이미지 컨트롤은 그래픽이 너무 크면 잘라서 컨트롤에 맞는 만큼만 그래픽을 표시합니다. *Stretch* 속성을 **true**로 설정하여 그래픽을 다시 조정할 때 이미지 컨트롤에 맞게 그래픽을 다시 조정합니다.

*TDBImage*에서는 많은 양의 데이터를 표시할 수 있기 때문에 런타임 시 표시할 내용을 채울 때 많은 시간이 걸립니다. 데이터 레코드를 스크롤하는 데 걸리는 시간을 줄이기 위해 *TDBImage*에는 액세스된 데이터를 자동으로 표시해야 할지를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 **false**로 설정하면 *TDBImage*에서는 실제 데이터가 아닌 필드 이름을 표시합니다. 실제 데이터를 표시하려면 컨트롤 내부를 더블 클릭합니다.

리스트 박스와 콤보 박스에서 데이터 표시 및 편집

런타임 시 선택할 수 있는 디폴트 데이터 값 집합을 사용자에게 제공하는 데이터 컨트롤에는 네 가지가 있습니다. 이러한 데이터 컨트롤은 다음과 같이 표준 리스트 박스와 콤보 박스 컨트롤의 데이터 인식 버전입니다.

- *TDBListBox*, 사용자가 데이터 필드에 입력하기 위해 선택할 수 있는 스크롤 가능한 항목의 리스트를 표시. 데이터 인식 리스트 박스는 현재 레코드의 필드에 대한 기본값을 표시하고 리스트에서 해당 엔트리를 강조 표시합니다. 현재 행의 필드 값이 리스트에 없으면 리스트 박스에서 어떤 값도 강조 표시되지 않습니다. 사용자가 리스트 항목을 선택할 때 해당 필드 값이 원본으로 사용하는 데이터셋에서 변경됩니다.
- *TDBComboBox*, 데이터 인식 편집 컨트롤과 드롭다운 리스트의 기능을 결합. 런타임 시 미리 정의된 값 집합에서 사용자가 선택할 수 있는 드롭다운 리스트를 표시하고 사용자가 완전히 다른 값을 입력할 수 있게 합니다.
- *TDBLookupListBox*, 표시 항목의 리스트를 다른 데이터셋에서 알아낼 수 있다는 점만 제외하면 *TDBListBox*처럼 작동
- *TDBLookupComboBox*, 표시 항목의 리스트를 다른 데이터셋에서 알아낼 수 있다는 점만 제외하면 *TDBComboBox*처럼 작동

참고 런타임 시 사용자는 증분 검색을 사용하여 리스트 박스 항목을 찾을 수 있습니다. 예를 들어, 컨트롤에 포커스가 있을 때 'ROB'를 입력하면 문자 'ROB'로 시작하는 리스트 박스의 첫 번째 항목이 선택됩니다. 추가로 'E'를 입력하면 'Robert Johnson'과 같이 'ROBE'로 시작하는 첫 번째 항목이 선택됩니다. 검색할 때는 대소문자를 구분하지 않습니다. *Backspace*와 *Esc*는 키스트로크 사이의 2초 일시 정지처럼 현재 검색 문자열을 취소하지만 선택한 내용은 그대로 둡니다.

TDBListBox와 TDBComboBox 사용

*TDBListBox*나 *TDBComboBox*를 사용할 때 런타임 시 String List Editor를 사용하여 표시할 항목 리스트를 만들어야 합니다. String List Editor를 불러오려면 Object Inspector의 *Items* 속성에 대한 생략 부호 버튼을 클릭합니다. 그리고 나서 리스트에 표시할 항목을 입력합니다. 런타임 시 *Items* 속성의 메소드를 사용하여 문자열 리스트를 처리할 수 있습니다.

DataField 속성을 통해 *TDBListBox*나 *TDBComboBox* 컨트롤이 필드에 연결되어 있으면 필드 값은 리스트에 선택되어 표시됩니다. 현재 값이 리스트에 없으면 선택되어 표시되는 항목이 없습니다. 하지만 *TDBComboBox*는 필드의 현재 값이 *Items* 리스트에 표시되는지에 상관 없이 에디트 박스에 필드의 현재 값을 표시합니다.

*TDBListBox*의 경우 *Height* 속성은 한 번에 리스트 박스에서 볼 수 있는 항목 수를 결정합니다. *IntegralHeight* 속성은 마지막 항목이 표시되는 방법을 제어합니다. *IntegralHeight*가 **false**(기본값)인 경우 리스트 박스의 하단은 *ItemHeight* 속성에 의해 결정되므로 하단 항목이 완전히 표시되지 않을 수도 있습니다. *IntegralHeight*가 **true**이면 리스트 박스에서 표시할 수 있는 하단 항목이 모두 표시됩니다.

*TDBComboBox*의 경우 *Style* 속성은 사용자의 컨트롤 사용 방법을 결정합니다. 디폴트로, *Style*은 *csDropDown*인데 이는 사용자가 키보드에서 값을 입력하거나 드롭다운 리스트에서 항목을 선택할 수 있다는 의미입니다. 다음 속성은 런타임 시 *Items* 리스트가 표시되는 방법을 결정합니다.

- *Style*은 다음과 같은 컴포넌트의 표시 스타일을 결정합니다.
  - *csDropDown*(기본값): 사용자가 텍스트를 입력할 수 있는 에디트 박스와 함께 드롭다운 리스트를 표시합니다. 모든 항목은 문자열로 높이가 같습니다.
  - *csSimple*: 항상 표시되는 고정 크기 항목 리스트와 편집 컨트롤을 결합합니다. *Style*을 *csSimple*로 설정하는 경우 리스트가 표시되도록 *Height* 속성을 증가시켜야 합니다.
  - *csDropDownList*: 드롭다운 리스트와 에디트 박스를 표시하지만 런타임 시 드롭다운 리스트에 없는 값을 입력하거나 변경할 수 없습니다.
  - *csOwnerDrawFixed* 와 *csOwnerDrawVariable*: 항목 리스트에서 비트맵과 같은 문자열 이외의 값을 표시하거나 리스트의 개별 항목에 대해 다양한 글꼴을 사용할 수 있게 해줍니다.
- *DropDownCount*: 리스트에 표시되는 항목의 최대 수를 나타냅니다. *Items*의 수가 *DropDownCount*보다 크면 사용자는 해당 리스트를 스크롤할 수 있습니다. *Items* 수가 *DropDownCount*보다 작으면 리스트의 크기는 모든 항목을 표시할 정도가 됩니다.
- *ItemHeight*: 스타일이 *csOwnerDrawFixed*일 때의 각 항목의 높이입니다.
- *Sorted*: **true**이면 *Items* 리스트가 알파벳 순으로 표시됩니다.

조회 리스트 박스 및 콤보 박스에서 데이터 표시 및 편집

조회 리스트 박스 및 조회 콤보 박스(*TDBLookupListBox*와 *TDBLookupComboBox*)는 유효한 필드 값을 설정할 제한된 선택 리스트를 사용자에게 제시합니다. 사용자가 리스트 항목을 선택하면 해당 필드 값은 원본으로 사용하는 데이터셋에서 변경됩니다.

예를 들어, *OrdersTable*에 연결된 필드가 있는 주문 폼이 있다고 가정합니다. *OrdersTable*에는 고객 ID에 해당하는 *CustNo* 필드가 있지만 *OrdersTable*에 다른 고객 정보는 없습니다. 반면 *CustomersTable*에는 고객 ID에 해당하는 *CustNo* 필드가 있는데 고객 회사와 주소 등의 추가 정보도 포함되어 있습니다. 사무원이 송장을 만들 때 주문 양식에서 고객 ID 대신 회사 이름으로 고객을 선택할 수 있게 해주면 편리합니다. *CustomersTable*의 모든 회사 이름을 표시하는 *TDBLookupListBox*를 사용하면 사용자는 리스트에서 회사 이름을 선택한 다음 주문 양식에서 *CustNo*를 적절하게 설정할 수 있습니다.

이러한 조회 컨트롤은 다음과 같은 두 개의 소스 중 하나에서 표시 항목 리스트를 파생시킵니다.

- **데이터셋에 대해 정의되어 있는 조회 필드**
  조회 필드를 사용하여 리스트 박스 항목을 지정하려면 컨트롤을 연결할 데이터셋에 조회 필드가 미리 정의되어 있어야 합니다. 이 프로세스는 23-8페이지의 "조회 필드 정의"에 설명되어 있습니다. 다음과 같은 방법으로 리스트 박스 항목의 조회 필드를 지정합니다.

  1 리스트 박스의 *DataSource* 속성을 사용할 조회 필드가 포함된 데이터셋의 데이터 소스로 설정합니다.

  2 사용할 조회 필드를 *DataField* 속성의 드롭다운 리스트에서 선택합니다.

  조회 컨트롤과 연결된 테이블을 활성화하면 컨트롤에서 테이블의 데이터 필드가 조회 필드임을 인식하고 조회 결과에서 적절한 값을 표시합니다.

- **보조 데이터 소스, 데이터 필드 및 키**
  데이터셋의 조회 필드를 정의하지 않았으면 보조 데이터 소스와 보조 데이터 소스에서 검색할 필드 값 및 리스트 항목으로 반환할 필드 값을 사용하여 유사한 관계를 설정할 수 있습니다. 다음과 같은 방법으로 리스트 박스 항목에 대한 보조 데이터 소스를 지정합니다.

  1 리스트 박스의 *DataSource* 속성을 컨트롤의 데이터 소스로 설정합니다.

  2 *DataField* 속성을 드롭다운 리스트에서 조회 값을 삽입할 필드를 선택하여 설정합니다. 선택한 필드는 조회 필드가 될 수 없습니다.

  3 리스트 박스의 *ListSource* 속성을 알아내려는 필드 값이 포함된 데이터셋의 데이터 소스로 설정합니다.

  4 *KeyField* 속성을 드롭다운 리스트에서 조회 키로 사용할 필드를 선택하여 설정합니다. 드롭다운 리스트는 단계 3에서 지정한 데이터 소스와 연결된 데이터셋의 필드를 표시합니다. 선택한 필드가 인덱스의 일부일 필요는 없지만 인덱스인 경우 조회 성능이 훨씬 더 빨라집니다.

  5 *ListField* 속성을 드롭다운 리스트에서 값을 반환할 필드를 선택하여 설정합니다. 드롭다운 리스트는 단계 3에서 지정한 데이터 소스와 연결된 데이터셋의 필드를 표시합니다.

  조회 컨트롤과 연결된 테이블을 활성화하면 컨트롤에서 테이블의 리스트 항목이 보조 소스에서 파생되었음을 인식하고 해당 소스에서 적절한 값을 표시합니다.

한 번에 표시할 항목 수를 *TDBLookupListBox* 컨트롤에서 지정하려면 *RowCount* 속성을 사용합니다. 리스트 박스의 높이는 정확히 행 수에 맞춰 조정됩니다.

*TDBLookupComboBox*의 드롭다운 리스트에 표시되는 항목 수를 지정하려면 대신 *DropDownRows* 속성을 사용합니다.

참고 데이터 그리드의 열을 설정하여 조회 콤보 박스처럼 작동하게 할 수 있는데 이러한 방법에 대한 자세한 내용은 19-20페이지의 "조회 리스트 열 정의"를 참조하십시오.

체크 박스와 함께 부울 필드 값 처리

*TDBCheckBox*는 데이터 인식 체크 박스 컨트롤입니다. 데이터셋의 부울 필드 값을 설정하는 데 사용될 수 있습니다. 예를 들어, 선택되어 있으면 비과세되는 고객을 나타내고 선택되어 있지 않으면 과세되는 고객을 나타내는 체크 박스 컨트롤이 고객 송장 양식에 있을 수 있습니다.

데이터 인식 체크 박스 컨트롤은 현재 필드의 값을 *ValueChecked* 와 *ValueUnchecked* 속성의 내용과 비교하여 선택되거나 선택되지 않은 상태를 관리합니다. 필드 값이 *ValueChecked* 속성과 일치하면 컨트롤이 선택됩니다. 그렇지 않고 필드 값이 *ValueUnchecked* 속성과 일치하면 컨트롤이 선택 해제됩니다.

참고 *ValueChecked*와 *ValueUnchecked*의 값은 동일할 수 없습니다.

사용자가 다른 레코드로 이동할 때 컨트롤이 선택되어 있으면 *ValueChecked* 속성을 데이터베이스로 포스트할 컨트롤 값으로 설정합니다. 디폴트로, 이 값은 "true"로 설정되어 있지만 개발자 필요에 따라 적절한 영숫자 값으로 바꿀 수 있습니다. 세미콜론으로 구분된 항목 리스트를 *ValueChecked*의 값으로 입력할 수도 있습니다. 현재 레코드에 있는 해당 필드의 내용과 일치하는 항목이 있으면 체크 박스가 선택됩니다. 예를 들어, 다음과 같은 *ValueChecked* 문자열을 지정할 수 있습니다.

```
DBCheckBox1->ValueChecked = “true;Yes;On”;
```

현재 레코드의 필드에 "true"나 "Yes", "On" 중 하나가 있으면 체크 박스가 선택됩니다. *ValueChecked* 문자열과 필드 비교 시 대소문자를 구분합니다. 여러 *ValueChecked* 문자열이 있는 박스를 사용자가 선택하면 첫 번째 문자열은 데이터베이스에 포스트할 값이 됩니다.

사용자가 다른 레코드로 이동할 때 컨트롤이 선택되어 있지 않으면 컨트롤에서 데이터베이스로 포스트할 값으로 *ValueUnchecked* 속성을 설정합니다. 디폴트로, 이 값은 "false"로 설정되어 있지만 개발자의 필요에 따라 적절한 영숫자 값으로 바꿀 수 있습니다. 세미콜론으로 구분된 항목을 *ValueUnchecked*의 값으로 입력할 수도 있습니다. 현재 레코드에 있는 해당 필드의 내용과 일치하는 항목이 있으면 체크 박스가 선택 해제됩니다.

*ValueChecked*나 *ValueUnchecked* 속성에 나열되어 있는 값 중 하나가 현재 레코드 필드에 포함되어 있지 않으면 데이터 인식 체크 박스를 사용할 수 없습니다.

체크 박스가 연결되어 있는 필드가 논리 필드인 경우 필드 내용이 **true**이면 체크 박스가 항상 선택되며 필드 내용이 **false**이면 체크 박스가 선택 해제됩니다. 이 경우 *ValueChecked*와 *ValueUnchecked* 속성에 입력된 문자열은 논리 필드에 영향을 미치지 않습니다.

라디오 컨트롤을 사용하여 필드 값 제한

*TDBRadioGroup*은 라디오 그룹 컨트롤의 데이터 인식 버전입니다. TDBRadioGroup을 사용하면 필드에 대한 가능한 값의 수가 제한되어 있는 라디오 버튼 컨트롤로 데이터 필드 값을 설정할 수 있습니다. 라디오 그룹은 필드에서 받아들일 수 있는 각 값에 대한 단추를 하나 포함합니다. 사용자는 원하는 라디오 단추를 선택하여 데이터 필드의 값을 설정할 수 있습니다.

Items 속성은 그룹에 표시된 라디오 버튼을 결정합니다. *Items*는 문자열 리스트입니다. *Items*에 있는 각 문자열에 대해 한 개의 라디오 버튼이 표시됩니다. 그리고 각 문자열은 라디오 버튼의 오른쪽에 버튼 레이블로 표시됩니다.

라디오 그룹과 연결된 필드의 현재 값이 *Items* 속성의 문자열 중 하나와 일치하면 해당 라디오 버튼이 선택됩니다. 예를 들어, *Items*에 "Red, "Yellow", "Blue"라는 세 개의 문자열이 나열되어 있는데 현재 레코드의 필드에 "Blue"라는 값이 포함되어 있으면 그룹의 세 번째 버튼이 선택된 것으로 표시됩니다.

참고 필드가 *Items* 에 있는 문자열 중 어느 것과도 일치하지 않지만 *Values* 속성의 문자열과 일치하면 라디오 버튼은 선택된 것으로 표시될 수 있습니다. 하지만 현재 레코드의 필드가 *Items* 나 *Values*의 어느 문자열과도 일치하지 않으면 라디오 버튼이 선택되지 않습니다.

Values 속성은 사용자가 라디오 버튼을 선택하고 레코드를 포스트할 때 데이터셋에 반환할 수 있는 선택적 문자열 리스트를 포함할 수 있습니다. 문자열은 순서대로 버튼과 연결됩니다. 첫 번째 문자열은 첫 번째 버튼과 연결되고 두 번째 문자열은 두 번째 버튼과 연결되는 식입니다. 예를 들어, *Items*에는 "Red", "Yellow", "Blue"가 있고 *Values*에는 "Magenta", "Yellow", "Cyan"이 있을 때 사용자가 "Red"라는 레이블의 버튼을 선택하면 "Magenta"가 데이터베이스에 포스트됩니다.

*Values*의 문자열이 제공되어 있지 않은 경우 레코드가 포스트될 때 선택한 라디오 버튼의 *Item* 문자열이 데이터베이스에 반환됩니다.

여러 레코드 표시

많은 레코드를 같은 형식으로 표시해야 할 경우가 가끔 있습니다. 예를 들어, 송장 작성 애플리케이션에서 동일한 고객의 주문은 모두 동일한 형식으로 표시해야 합니다.

여러 레코드를 표시하려면 그리드 컨트롤을 사용합니다. 그리드 컨트롤은 애플리케이션의 사용자 인터페이스를 좀 더 강력하고 효과적으로 만들 수 있는 여러 필드, 여러 레코드 데이터 뷰를 제공합니다. 이러한 내용은 19-15페이지의 "TDBGrid를 사용하여 데이터 보기 및 편집"과 19-27페이지의 "기타 데이터 인식 컨트롤을 포함하는 그리드 생성"에서 설명합니다.

참고 단방향 데이터셋을 사용할 때는 여러 레코드를 표시할 수 없습니다.

단일 레코드의 필드와 여러 레코드를 나타내는 그리드를 모두 표시하는 사용자 인터페이스를 디자인할 수 있습니다. 이러한 두 가지 방법을 결합하는 두 가지 모델은 다음과 같습니다.

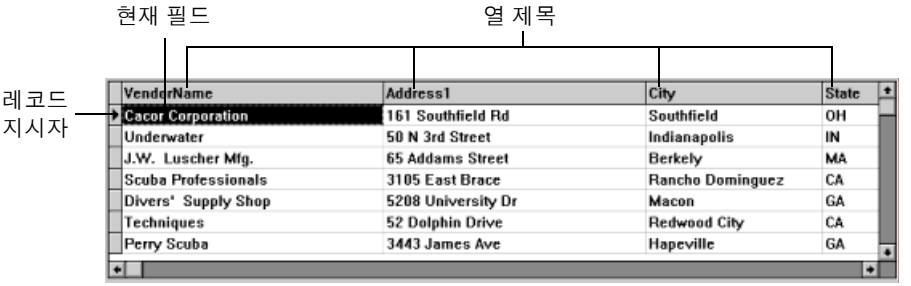
- **마스터/디테일 폼:** 단일 필드를 표시하는 컨트롤과 그리드 컨트롤을 모두 포함하여 마스터 테이블과 디테일 테이블 양쪽의 정보를 나타낼 수 있습니다. 예를 들어, 해당 고객의 주문을 표시하는 디테일 그리드를 사용하여 한 고객에 대한 정보를 표시할 수 있습니다. 마스터/디테일 폼에서 원본으로 사용하는 테이블에 연결하는 방법에 대한 자세한 내용은 22-34 페이지의 "마스터 / 디테일 관계 생성"과 22-45페이지의 "매개변수를 사용하여 마스터 / 디테일 관계 설정"을 참조하십시오.
- **드릴다운 폼:** 여러 레코드를 표시하는 폼에서 현재 레코드의 디테일 정보만 표시하는 단일 필드 컨트롤을 포함할 수 있습니다. 이러한 방법은 레코드에 긴 메모나 그래픽 정보가 있을 때 특히 유용합니다. 사용자가 그리드 레코드를 스크롤함에 따라 메모나 그래픽이 현재 레코드의 값을 나타내기 위해 업데이트됩니다. 설정 방법은 아주 쉽습니다. 그리드 컨트롤과 메모나 이미지 컨트롤에서 공통 데이터 소스를 공유하고 있다면 두 가지 디스플레이의 동기화가 자동으로 이루어집니다.

팁 이러한 두 가지 방법을 단일 폼에서 결합하는 것은 일반적으로 좋은 방법이 아닙니다. 이러한 폼에서는 사용자가 데이터 관계를 이해하기가 어렵습니다.

TDBGrid를 사용하여 데이터 보기 및 편집

TDBGrid 컨트롤을 사용하면 테이블 형식의 그리드 형식에서 데이터셋의 레코드를 보고 편집할 수 있습니다.

그림 19.1 TDBGrid 컨트롤



다음과 같은 세 가지 요소가 그리드 컨트롤에 표시된 레코드 모양에 영향을 미칩니다.

- Columns Editor를 사용하여 그리드에 대해 정의한 영구적 열 객체의 존재. 영구적 열 객체는 그리드와 데이터 모양을 설정하는 데 많은 융통성을 제공합니다. 영구적 열 사용에 대한 자세한 내용은 19-16페이지의 "사용자 정의된 그리드 생성"을 참조하십시오.
- 그리드에 표시된 데이터셋에 대한 영구적 필드(persistent field) 컴포넌트의 생성. Fields Editor를 사용하여 영구적 필드 컴포넌트 생성에 대한 자세한 내용은 23장, "필드 컴포넌트 작업"을 참조하십시오.
- ADT와 배열 필드를 표시하는 그리드에 대한 데이터셋의 *ObjectView* 속성 설정. 19-22페이지의 "ADT 및 배열 필드 표시"를 참조하십시오.

그리드 컨트롤에는 자체가 *TDBGridColumns* 객체의 랩퍼인 *Columns* 속성이 있습니다. *TDBGridColumns*는 그리드 컨트롤의 모든 열을 나타내는 *TColumn* 객체의 컬렉션입니다. Columns Editor를 사용하여 다지인 타임 시 열 어트리뷰트(attribute)를 설정하거나 그리드의 *Columns* 속성을 사용하여 런타임 시 *TDBGridColumns*의 속성, 이벤트 및 메소드에 액세스할 수 있습니다.

디폴트 상태로 그리드 컨트롤 사용

그리드 *Columns* 속성의 *State* 속성은 그리드에 대해 영구적 열 객체가 있는지 여부를 표시합니다. *Columns->State*는 그리드에 대해 자동으로 설정되는 런타임 전용 속성입니다. 디폴트 상태는 *csDefault*로, 영구적 열 객체가 해당 그리드에 대해 존재한지 않는다는 의미입니다. 그런 경우 그리드의 데이터 디스플레이는 그리드 데이터셋의 필드 속성에 의해 주로 결정되거나 영구적 필드 컴포넌트가 없으면 디폴트 디스플레이 특징에 의해 결정됩니다.

그리드의 *Columns->State* 속성이 *csDefault*이면 그리드 열은 데이터셋의 보이는 필드로부터 동적으로 생성되고 그리드의 열 순서는 데이터셋의 필드 순서와 일치합니다. 그리드의 각 열은 필드 컴포넌트와 연결되어 있습니다. 필드 컴포넌트에 대한 속성 변경이 바로 그리드에 표시됩니다.

동적으로 생성된 열과 함께 그리드 컨트롤을 사용하면 런타임 시 선택된 임의의 테이블 내용을 보고 편집할 때 유용합니다. 그리드의 구조가 설정되어 있지 않기 때문에 다른 데이터셋과의 조화를 위해 동적 변경이 가능합니다. 동적으로 생성된 열이 있는 단일 그리드에서는 한 번에 한 Paradox 테이블만 표시한 다음 그리드의 *DataSource* 속성이 변경되거나 데이터 소스 자체의 *DataSet* 속성이 변경될 때 SQL 쿼리의 결과를 표시할 수 있습니다.

디자인 타임이나 런타임 시 동적 열의 모양을 변경할 수 없지만 실제로 수정하고 있는 내용은 열에 표시된 필드 컴포넌트의 해당 속성입니다. 열이 단일 데이터셋의 특정 필드에 연결되어 있는 동안만 동적 열의 속성이 존재합니다. 예를 들어, 특정 열의 *Width* 속성을 변경하면 해당 열과 연결된 필드의 *DisplayWidth* 속성이 변경됩니다. *Font*처럼 필드 속성에 기반을 두고 있지 않은 열 속성의 변경 내용은 열의 수명 동안만 존재합니다.

그리드의 데이터셋이 동적 필드 컴포넌트로 구성되어 있으면 데이터셋이 닫힐 때마다 필드가 소멸됩니다. 필드 컴포넌트가 소멸되면 이와 연결된 동적 열도 모두 소멸됩니다. 그리드의 데이터셋이 영구적 필드 컴포넌트로 구성되어 있으면 데이터셋이 닫혀도 필드 컴포넌트는 남아 있습니다. 따라서 영구적 필드와 연결된 열도 데이터셋이 닫힐 때 각각의 속성을 그대로 보유합니다.

참고 그리드의 *Columns->State* 속성을 *csDefault*로 런타임 시 변경하면 영구적 열을 비롯한 그리드의 열 객체가 모두 삭제되고 그리드 데이터셋의 보이는 필드를 기반으로 동적 열이 다시 생성됩니다.

사용자 정의된 그리드 생성

사용자 정의된 그리드는 열 표시 방법과 열의 데이터 표시 방법을 설명하는 영구적 열 객체를 정의하는 그리드입니다. 사용자 정의된 그리드를 사용하면 여러 그리드를 구성하여 같은 데이터셋의 다양한 뷰(예: 다양한 열 순서, 다양한 필드 선택, 다양한 열 색상 및 글꼴)를 제시할 수 있습니다. 또한 그리드에서 사용하는 필드나 데이터셋의 필드 순서에 영향을 미치지 않고 런타임 시 그리드 모양을 수정할 수도 있습니다.

사용자 정의된 그리드는 디자인 타임 시 구조를 알 수 있는 데이터셋과 가장 잘 사용됩니다. 사용자 정의된 그리드에서는 디자인 타임 시 만든 필드 이름이 데이터셋에 존재하길 기대하므로 런타임 시 선택한 임의의 테이블을 찾는 데는 적합하지 않습니다.

영구적 열 이해

그리드에 대한 영구적 열 객체를 만들면 이들은 원본으로 사용하는 그리드 데이터셋의 필드와 느슨하게 연결됩니다. 열 속성에 값이 할당될 때까지 영구적 열의 디폴트 속성 값은 디폴트 소스(연결된 필드나 그리드 자체)에서 동적으로 가져오게 됩니다. 열 속성에 값을 할당할 때 디폴트 소스가 변경됨에 따라 값도 변경됩니다. 열 속성에 값을 할당한 다음에는 디폴트 소스가 변경되더라도 더 이상 값이 변경되지 않습니다.

예를 들어, 열 제목의 디폴트 소스는 연결된 필드의 *DisplayLabel* 속성입니다. *DisplayLabel* 속성을 수정하면 열 제목은 바로 변경되는 내용을 반영합니다. 그런 다음 열 제목의 캡션에 문자열을 할당하면 제목 캡션은 연결된 필드의 *DisplayLabel* 속성과 분리됩니다. 필드의 *DisplayLabel* 속성을 차후에 변경하면 열 제목에 더 이상 영향을 미치지 않습니다.

영구적 열은 연결된 필드 컴포넌트와 독립적으로 존재합니다. 사실, 영구적 열은 필드 객체와 연결될 필요가 전혀 없습니다. 영구적 열의 *FieldName* 속성이 비어 있거나 필드 이름이 그리드의 현재 데이터셋에 있는 필드 이름과 일치하지 않으면 , 해당 열의 *Field* 속성은 NULL 이 되고 해당 열이 비어 있는 셀로 그려집니다. 셀의 디폴트 그리기 메소드를 오버라이드하면 비어 있는 셀에 사용자 고유의 사용자 정의 정보를 표시할 수 있습니다. 예를 들어, 비어 있는 열을 사용하면 요약을 집계하는 레코드 그룹의 마지막 레코드에서 집계 값을 표시할 수 있습니다. 또는 레코드 데이터의 특정 면을 그래픽으로 나타내는 비트맵이나 막대 차트를 표시할 수 있습니다.

두 개 이상의 영구적 열은 데이터셋의 동일한 필드와 연결할 수 있습니다. 예를 들어, 와이드 그리드의 왼쪽과 오른쪽 끝에 부품 번호 필드를 표시하면 그리드를 스크롤하지 않고 부품 번호를 쉽게 찾을 수 있습니다.

참고 영구적 열은 데이터셋의 필드와 연결되지 않아도 되고 여러 열에서 같은 필드를 참조할 수 있으므로 사용자 정의 그리드의 *FieldCount* 속성은 그리드의 열 개수 이하가 될 수 있습니다. 또한 사용자 정의 그리드에서 현재 선택된 열이 필드와 연결되어 있지 않으면 그리드의 *SelectedField* 속성은 NULL이 되고 *SelectedIndex* 속성은 –1이 됨에 유의하십시오.

영구적 열을 구성하여 그리드 셀을 다른 데이터셋이나 정적 선택 리스트에서 조회 값의 콤보 박스 드롭다운 리스트로 표시할 수 있습니다. 또는 현재 셀에 관련된 특수 데이터 뷰어나 다이얼로그 박스를 시작할 때 클릭할 수 있는 셀의 생략 부호 버튼(…)으로 표시할 수 있습니다.

영구적 열 생성

디자인 타임 시 그리드 모양을 사용자 정의하려면 Columns Editor를 호출하여 그리드에 대한 영구적 열 객체 집합을 만듭니다. 런타임 시 영구적 열 객체를 갖는 그리드의 *State* 속성은 자동으로 *csCustomized*로 설정됩니다.

다음과 같은 방법으로 그리드 컨트롤의 영구적 열을 만듭니다.

1. 폼의 그리드 컴포넌트를 선택합니다.
2. Object Inspector에서 그리드의 *Columns* 속성을 더블 클릭하여 Columns Editor를 호출합니다.

Columns 리스트 박스는 선택한 그리드에 정의했던 영구적 열을 표시합니다. 먼저 Columns Editor를 불러오면 동적 열만 포함하는 그리드가 디폴트 상태에 있기 때문에 해당 리스트가 비어 있습니다.

데이터셋의 모든 필드에 대해 영구적 열을 동시에 만들거나 개별적으로 영구적 열을 만들 수 있습니다. 다음과 같은 방법으로 모든 필드에 대해 영구적 열을 만듭니다.

1. 마우스 오른쪽 버튼으로 그리드를 클릭하여 컨텍스트 메뉴를 호출한 다음 Add All Fields를 선택합니다. 그리드가 아직 데이터 소스와 연결되어 있지 않으면 Add All Fields를 사용할 수 없음에 유의하십시오. Add All Fields를 선택하기 전에 활성 데이터셋을 갖는 데이터 소스와 그리드를 연결해야 합니다.
2. 그리드에 영구적 열이 이미 포함되어 있으면 다이얼로그 박스에서 기존 열을 삭제할 것인지 또는 컬럼 셋에 추가할 것인지 묻습니다. Yes를 선택하면 기존의 영구적 열 정보가 제거되고 현재 데이터셋의 모든 필드가 데이터셋의 필드 순서에 따라 필드 이름에 의해 삽입됩니다. No를 선택하면 기존의 영구적 열 정보가 유지되고 데이터셋의 추가 필드를 기반으로 새 열 정보가 데이터셋에 추가됩니다.
3. Close를 클릭하면 영구적 열이 그리드에 적용된 다음 다이얼로그 박스가 닫힙니다.

다음과 같은 방법으로 영구적 열을 개별적으로 만듭니다.

1 Columns Editor에서 Add 버튼을 선택합니다. 새 열이 리스트 박스에 선택됩니다. 새 열에는 0 - TColumn 형식의 순차적 번호와 디폴트 이름이 부여됩니다.

2 필드를 새 열과 연결하려면 Object Inspector의 *FieldName* 속성을 설정합니다.

3 새 열의 제목을 설정하려면 Object Inspector 의 *Title* 속성을 확장한 다음 *Caption* 속성을 설정합니다.

4 Columns Editor를 닫으면 영구적 열이 그리드에 적용된 다음 다이얼로그 박스가 닫힙니다.

런타임 시 *Columns::State* 속성에 *csCustomized*를 할당하여 영구적 열로 전환할 수 있습니다. 그리드의 기존 열은 소멸되고 그리드 데이터셋의 각 필드에 대해 새 영구적 열이 생성됩니다. 그러면 다음과 같이 열 리스트에 대한 *Add* 메소드를 호출하여 런타임 시 영구적 열을 추가할 수 있습니다.

```
DBGrid1->Columns->Add();
```

영구적 열 삭제

그리드에서 영구적 열을 삭제하면 표시할 필요가 없는 필드를 제거하는 데 도움이 됩니다. 다음과 같은 방법으로 영구적 열을 그리드에서 제거합니다.

1 그리드를 더블 클릭하여 Columns Editor를 표시합니다.

2 Columns 리스트 박스에서 제거할 필드를 선택합니다.

3 Delete를 클릭합니다. 또는 컨텍스트 메뉴나 *Del* 키를 사용하여 열을 제거할 수 있습니다.

참고 그리드의 모든 열을 삭제하면 *Columns->State* 속성이 *csDefault* 속성으로 바뀌고 데이터셋의 각 필드에 대한 동적 열을 자동으로 생성합니다.

다음과 같이 열 객체를 해제하기만 하면 런타임 시 영구적 열을 삭제할 수 있습니다.

```
delete DBGrid1->Columns->Items[5];
```

영구적 열의 순서 정렬

Columns Editor에 열이 표시되는 순서는 그리드에서 열이 표시되는 순서와 같습니다. Columns 리스트 박스 내에서 열을 끌어다 놓아서 열 순서를 바꿀 수 있습니다.

다음과 같은 방법으로 열 순서를 바꿉니다.

1 Columns 리스트 박스의 열을 선택합니다.

2 리스트 박스의 새 위치로 열을 끕니다.

열 제목을 클릭한 다음 해당 열을 새 위치로 끌어서 런타임 시 열 순서를 바꿀 수 있습니다.

참고 Fields Editor의 영구적 필드(persistent field)를 재정렬하면 디폴트 그리드의 열도 재정렬되지만 사용자 그리드의 열은 재정렬되지 않습니다.

중요 동적 열과 동적 필드가 모두 포함되어 있는 그리드에서는 디자인 타임 시 열을 재정렬할 수 없습니다. 대체된 필드나 열 순서를 기록할 영구적 것이 없기 때문입니다.

DragMode 속성이 *dmManual*로 설정되어 있으면 런타임 시 사용자는 마우스를 사용하여 특정 열을 그리드의 새 위치로 끌 수 있습니다. *State* 속성이 *csDefault* 상태인 그리드의 열을 재정렬하면 그리드의 원본으로 사용하는 데이터셋의 필드 컴포넌트도 재정렬됩니다. 하지만 실제 테이블의 필드 순서는 영향을 받지 않습니다 . 사용자가 런타임 시 열을 재정렬하는 것을 막으려면 그리드의 *DragMode* 속성을 *dmAutomatic*으로 설정합니다.

런타임 시 그리드의 *OnColumnMoved* 이벤트는 열이 이동된 다음에 시작됩니다.

디자인 타임 시 열 속성 설정

열 속성은 해당 열의 셀에서 데이터가 표시되는 방법을 결정합니다. 대부분의 열 속성은 다른 컴포넌트(예: 그리드나 연결된 필드 컴포넌트)와 연결된 속성의 기본값을 갖습니다. 이를 *디폴트* 소스라고 합니다.

열 속성을 설정하려면 Columns Editor에서 해당 열을 선택한 다음 Object Inspector에서 열의 속성을 설정합니다. 다음 표는 설정할 수 있는 주요 열 속성을 요약한 것입니다.

표 19.2 열 속성

속성	용도
Alignment	열의 필드 데이터를 왼쪽이나 오른쪽 또는 가운데로 정렬. 디폴트 소스: *TField::Alignment*
ButtonStyle	*cbsAuto*: (기본값) 연결된 필드가 조회 필드이거나 해당 열의 *PickList* 속성에 데이터가 있으면 드롭다운 리스트를 표시
	cbsEllipsis: 셀의 오른쪽에 생략(...) 버튼 표시. 버튼을 클릭하면 그리드의 *OnEditButtonClick* 이벤트가 시작
	cbsNone: 열의 데이터 편집 시 정상 편집 컨트롤만 사용하는 열
Color	열에 있는 셀의 배경색을 지정. 디폴트 소스: *TDBGrid::Color.* 텍스트 기본색에 대해서는 Font 속성 참조
DropDownRows	드롭다운 리스트에서 표시하는 텍스트의 줄 수. 기본값: 7
Expanded	열의 확장 여부 지정. ADT나 배열 필드를 나타내는 열에만 적용
FieldName	해당 열과 연결된 필드 이름 지정. 공백일 수 있음
ReadOnly	**true**: 열의 데이터를 사용자가 편집할 수 없음
	false: (기본값) 열의 데이터를 편집할 수 있음
Width	열 너비를 화면 픽셀 단위로 지정. 디폴트 소스: *TField::DisplayWidth*
Font	열에서 텍스트를 그릴 때 사용되는 글꼴 타입, 크기, 색을 지정. 디폴트 소스: *TDBGrid::Font*
PickList	열 드롭다운 리스트에 표시할 값 리스트를 포함
Title	선택한 열의 제목에 대한 속성 설정

다음 표는 *Title* 속성에 지정할 수 있는 옵션을 요약한 것입니다.

표 19.3 확장된 TColumn Title 속성

속성	용도
Alignment	열 제목에서 캡션 텍스트를 왼쪽(기본값)이나 오른쪽, 가운데로 정렬
Caption	열 제목에 표시할 텍스트를 지정. 디폴트 소스: *TField::DisplayLabel*
Color	열 제목 셀을 그리는 데 사용할 배경색을 지정. 디폴트 소스: *TDBGrid::FixedColor*
Font	열 제목에서 텍스트를 그릴 때 사용되는 글꼴 타입, 크기 및 색을 지정. 디폴트 소스: *TDBGrid::TitleFont*

조회 리스트 열 정의

조회 콤보 박스 컨트롤과 유사한, 드롭다운 리스트를 표시하는 열을 만들 수 있습니다. 해당 열이 콤보 박스처럼 작동하도록 지정하려면 열의 *ButtonStyle* 속성을 *cbsAuto*로 설정합니다. 리스트를 값으로 채우고 나면 해당 열의 셀이 편집 모드에 있을 때 그리드는 콤보 박스 같은 드롭다운 버튼을 자동으로 디스플레이합니다.

사용자가 선택한 값으로 리스트를 채우는 방법은 다음 두 가지입니다.

- 조회 테이블에서 값을 가져올 수 있습니다. 각각의 조회 테이블에서 그린 드롭다운 리스트 값을 셀에서 표시하게 하려면 데이터셋에서 조회 필드를 정의해야 합니다. 조회 필드 생성에 대한 자세한 내용은 23-8페이지의 "조회 필드 정의"를 참조하십시오. 조회 필드를 정의한 다음에는 열의 *FieldName*을 조회 필드 이름으로 설정합니다. 조회 필드에서 정의한 조회 값으로 드롭다운 리스트가 자동으로 채워집니다.
- 디자인 타임 시 값 리스트를 명시적으로 지정할 수 있습니다. 디자인 타임 시 값 리스트를 입력하려면 Object Inspector에서 해당 열의 *PickList* 속성을 더블 클릭합니다. 그러면 해당 열의 선택 리스트를 채우는 값을 입력할 수 있는 String List Editor가 나타납니다.

디폴트로, 드롭다운 리스트에는 7개의 값이 표시됩니다. *DropDownRows* 속성을 설정하여 이 리스트의 길이를 변경할 수 있습니다.

참고 명시적 선택 리스트를 갖는 열의 기본 동작을 복구하려면 String List Editor 를 사용하여 선택 리스트에서 모든 텍스트를 삭제합니다.

열에 단추 놓기

열에서는 기본 셀 에디터의 오른쪽에 생략 부호 버튼(…)을 표시할 수 있습니다. *Ctrl+Enter*나 마우스를 클릭하면 그리드의 *OnEditButtonClick* 이벤트가 시작합니다. 생략 부호 버튼을 사용하여 열 데이터의 좀 더 자세한 뷰가 포함된 폼을 불러올 수 있습니다. 예를 들어, 송장 요약을 표시하는 테이블에서 송장 합계 열에 생략 부호 버튼을 설정하면 송장의 항목을 표시하는 폼이나 세금 계산 메소드 등을 불러올 수 있습니다. 그래픽 필드에서는 생략 부호 버튼을 사용하여 이미지를 표시하는 폼을 불러올 수 있습니다.

다음과 같은 방법으로 열에 생략 부호 버튼을 만듭니다.

1. *Columns* 리스트 박스에서 열을 선택합니다.
2. *ButtonStyle*을 *cbsEllipsis*로 설정합니다.
3. *OnEditButtonClick* 이벤트 핸들러를 작성합니다.

열에 대한 기본값 복구

런타임 시 열의 *AssignedValues* 속성을 테스트하여 열 속성이 명시적으로 할당되었는지 결정합니다. 명시적으로 정의되어 있지 않은 값은 연결된 필드나 그리드 기본값을 동적으로 기반으로 합니다.

하나 이상의 열에 만든 속성 변경을 취소할 수 있습니다. Columns Editor에서 복구할 열을 선택한 다음 컨텍스트 메뉴에서 Restore Defaults를 선택합니다. 기본값을 복구하면 할당된 속성 설정을 버리고 원본으로 사용하는 필드 컴포넌트에서 파생된 속성으로 열의 속성을 복구합니다.

런타임 시 열의 *RestoreDefaults* 메소드를 호출하여 단일 열의 모든 디폴트 속성을 다시 설정할 수 있습니다. 다음과 같이 열 리스트의 *RestoreDefaults* 메소드를 호출하여 그리드의 모든 열에 대한 디폴트 속성을 다시 설정할 수 있습니다.

```
DBGrid1->Columns->RestoreDefaults();
```

ADT 및 배열 필드 표시

가끔 그리드 데이터셋 필드에서 텍스트, 그래픽, 숫자 값 등 단순한 값을 나타내지 않을 때가 있습니다. 일부 데이터베이스 서버에서는 간단한 데이터 타입의 복합 필드인, ADT 필드나 배열 필드 등을 허용합니다.

그리드에서 복합 필드를 표시할 수 있는 방법은 다음과 같이 두 가지입니다.

- 필드를 구성하는 보다 간단한 타입이 데이터셋에서 각각의 필드로 나타나도록 "단순화"할 수 있습니다. 복합 필드가 단순화되면 필드의 구성 요소는 각각의 필드 이름 앞에 원본으로 사용하는 데이터베이스 테이블의 공통 상위 필드의 이름이 온다는 점에서만 공통 소스를 반영하는, 독립적인 필드로 나타납니다.

  복합 필드를 단순화된 것처럼 표시하려면 데이터셋의 *ObjectView* 속성을 **false**로 설정합니다. 데이터셋에서는 복합 필드를 각각의 필드 집합으로 저장하고 그리드는 구성 요소 부분에 각각의 열을 할당하여 이를 반영합니다.

- 복합 필드가 단일 필드임을 반영하면서 단일 열에 표시할 수 있습니다. 복합 필드가 단일 열에 표시되어 있는 경우 다음과 같이 필드의 제목 표시줄에서 화살표를 클릭하거나 해당 열의 *Expanded* 속성을 설정하여 열을 확장 및 축소할 수 있습니다.

  - 열을 확장하면 각 자식 필드가 부모 필드의 제목 표시줄 아래에, 제목 표시줄이 있는 고유의 하위 열에 나타납니다. 즉, 복합 필드의 이름을 부여하는 첫 번째 행 그리고 개별 부분으로 분리하는 두 번째 행과 함께 그리드의 제목 표시줄은 높이가 증가합니다. 복합 필드가 아닌 필드는 제목 표시줄의 높이가 매우 크게 표시됩니다. 이러한 확장은 디테일 테이블에 중첩된 디테일 테이블의 경우처럼 복합 필드인 구성 요소마다 차례로 계속되고 제목 표시줄의 높이도 계속 증가합니다.

  - 필드를 축소하면 자식 필드가 포함된 쉼표로 구분된 편집할 수 없는 문자열과 함께 한 열만 표시됩니다.

  복합 필드를 확장 및 축소 열에서 표시하려면 데이터셋의 *ObjectView* 속성을 **true**로 설정합니다. 데이터셋은 복합 필드를 중첩된 하위 필드 집합을 포함하는 단일 필드 컴포넌트로 저장합니다. 그리드에서는 이러한 내용을 확장이나 축소가 가능한 열에 반영합니다.

그림 19.2는 ADT 필드와 배열 필드가 있는 그리드를 보여 줍니다. 자식 필드마다 열이 한 개씩 있게 데이터셋의 *ObjectView* 속성을 **false**로 설정합니다.

그림 19.2 ObjectView가 false로 설정되어 있는 TDBGrid 컨트롤

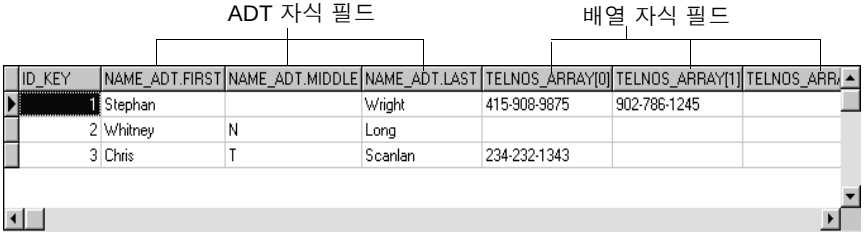


그림 19.3과 19.4는 ADT 필드와 배열 필드가 있는 그리드를 보여 줍니다. 그림 19.3은 축소된 필드를 보여 줍니다. 이 상태에서는 편집할 수 없습니다. 그림 19.4는 확장된 필드를 보여 줍니다. 필드 제목 표시줄의 화살표를 클릭하면 필드가 확장 및 축소됩니다.

그림 19.3 Expanded가 false로 설정된 TDBGrid 컨트롤

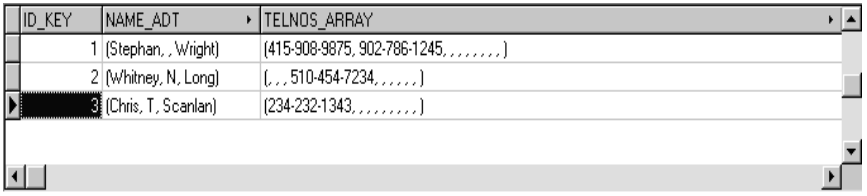
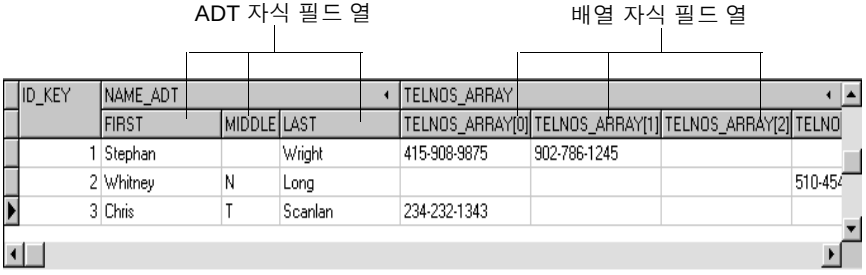


그림 19.4 Expanded가 true로 설정된 TDBGrid 컨트롤



다음 표는 *TDBGrid*에 ADT 및 배열 필드의 표시 방법에 영향을 미치는 속성을 나열합니다.

표 19.4 복합 필드의 표시 방법에 영향을 미치는 속성

속성	객체	용도
Expandable	TColumn	자식 필드를 각각의 편집 가능한 열에 표시하기 위해 열을 확장할 수 있는지 표시 (읽기 전용)
Expanded	TColumn	열을 확장할 것인지 지정
MaxTitleRows	TDBGrid	그리드에서 표시할 수 있는 제목 행의 최대 수 지정
ObjectView	TDataSet	필드를 단순하게 표시할 것인지 아니면 각 객체 필드를 확장 및 축소할 수 있는 객체 모드로 표시할 것인지 지정
ParentColumn	TColumn	자식 필드 열을 갖는 TColumn 객체를 참조

참고 ADT와 배열 필드 이외에도 일부 데이터셋에는 다른 데이터셋(데이터셋 필드)이나 다른 데이터셋(참조) 필드의 레코드를 참조하는 필드를 포함합니다. 데이터 인식 그리드에서는 이러한 필드를 각각 "(DataSet)"이나 "(Reference)"로 표시합니다. 런타임 시 생략 부호 버튼은 오른쪽에 표시됩니다. 생략 부호 버튼을 클릭하면 필드 내용을 표시하는 그리드와 함께 새 폼이 나타납니다. 데이터셋 필드의 경우 이 그리드는 필드 값인 데이터셋을 표시합니다. 참조 필드의 경우 이 그리드는 다른 데이터셋의 레코드를 표시하는 단일 행을 포함합니다.

그리드 옵션 설정

디자인 타임 시 *Options* 속성을 사용하여 런타임의 기본 그리드 동작과 모양을 제어할 수 있습니다. 디자인 타임 시 그리드 컴포넌트를 먼저 폼에 두면 Object Inspector의 *Options* 속성은 +(더하기) 부호로 표시됩니다. 이는 *Options* 속성을 확장하여 개별적으로 설정할 수 있는 부울 속성 집합을 표시할 수 있음을 나타냅니다. 이러한 속성을 보고 설정하려면 + 부호를 클릭합니다. 옵션 리스트는 *Options* 속성 아래의 Object Inspector에 표시됩니다. + 부호는 클릭하면 리스트를 축소할 수 있는 –(빼기) 부호로 변경됩니다.

다음 표는 설정할 수 있는 *Options* 속성을 나열하고 런타임 시 그리드에 영향을 미치는 방법을 설명합니다.

표 19.5 Expanded TDBGrid Options 속성

옵션	용도
dgEditing	**true**: (기본값). 그리드에서 레코드 편집, 삽입 및 삭제를 사용할 수 있게 만듭니다.
	false: 그리드에서 레코드 편집, 삽입 및 삭제를 사용할 수 없게 만듭니다.
dgAlwaysShowEditor	**true**: 필드를 선택하면 Edit 상태가 됩니다.
	false: (기본값). 선택하면 필드가 자동으로 Edit 상태가 되지 않습니다.
dgTitles	**true**: (기본값). 그리드의 맨 위에 필드 이름을 표시합니다.
	false: 필드 이름 표시 기능이 사용되지 않습니다.
dgIndicator	**true**: (기본값). 지시자 열이 그리드 왼쪽에 표시되고 현재 레코드를 표시하기 위해 현재 레코드 지시자(그리드 왼쪽의 화살표)가 활성화됩니다. 삽입할 때 화살표는 별표가 되고 편집할 때는 화살표가 I형 포인터가 됩니다.
	false: 지시자 열이 사용되지 않습니다.
dgColumnResize	**true**: (기본값). 제목 영역에서 열 눈금자를 끌어서 열의 크기를 조정할 수 있습니다. 크기를 조정하면 원본으로 사용하는 *TField* 컴포넌트의 해당 너비가 변경됩니다.
	false: 그리드에서 열의 크기를 조정할 수 없습니다.
dgColLines	**true**: (기본값). 열 사이에 세로 구분선을 표시합니다.
	false: 열 사이에 구분선을 표시하지 않습니다.
dgRowLines	**true**: (기본값). 레코드 사이에 가로 구분선을 표시합니다.
	false: 레코드 사이에 구분선을 표시하지 않습니다.
dgTabs	**true**: (기본값). Tab 키를 사용하여 레코드의 필드 간을 이동할 수 있습니다.
	false: Tab 키를 누르면 포커스가 그리드 컨트롤에서 다른 컨트롤로 이동합니다.

표 19.5 Expanded TDBGrid Options 속성(계속)

옵션	용도
dgRowSelect	**true**: 선택 표시줄이 그리드의 전체 너비로 확장됩니다.
	false: (기본값). 레코드의 필드를 선택하면 해당 필드만 선택됩니다.
dgAlwaysShowSelection	**true**: (기본값). 다른 컨트롤에 포커스가 있더라도 그리드의 선택 표시줄을 항상 볼 수 있습니다.
	false: 그리드에 포커스가 있을 때만 그리드의 선택 표시줄을 볼 수 있습니다.
dgConfirmDelete	**true**: (기본값). 레코드를 삭제할 것인지 확인하는 메시지를 표시합니다 (*Ctrl+Del*).
	false: 확인하지 않고 레코드를 삭제합니다.
dgCancelOnExit	**true**: (기본값). 그리드에서 포커스가 이동하면 보류 중인 삽입을 취소합니다. 이 옵션에서는 부분 레코드나 비어 있는 레코드의 부주의한 포스트를 막습니다.
	false: 보류 중인 삽입을 허용합니다.
dgMultiSelect	**true**: *Ctrl+Shift*나 *Shift+ arrow* 키를 사용하여 그리드에서 여러 곳의 행들을 선택하는 것을 허용합니다.
	false: (기본값). 행을 여러 개 선택하는 것을 허용하지 않습니다.

그리드에서 편집

런타임 시 다음 디폴트 조건이 충족되면 그리드를 사용하여 기존 데이터를 수정하고 새 레코드를 입력할 수 있습니다.

- *Dataset*의 *CanModify* 속성이 **true**여야 합니다.
- 그리드의 *ReadOnly* 속성이 *false*여야 합니다.

사용자가 그리드의 레코드를 편집하면 각 필드에 대한 변경 내용은 내부 레코드 버퍼에 포스트되지만 사용자가 그리드의 다른 레코드로 이동할 때까지는 포스트되지 않습니다. 포커스를 폼의 다른 컨트롤로 변경하더라도 해당 데이터셋의 커서를 다른 레코드로 이동해야만 그리드에서 변경 내용을 포스트합니다. 레코드를 포스트하면 데이터셋에서 현재 변경 내용에 연결된 데이터 인식 컴포넌트를 모두 검사합니다. 수정된 데이터를 포함하는 필드를 업데이트하는 데 문제가 발생하면 그리드에서 예외를 발생하고 해당 레코드를 수정하지 않습니다.

참고 애플리케이션에서 업데이트를 캐싱하는 경우 레코드 변경 내용을 포스트하면 내부 캐시에만 추가됩니다. 애플리케이션에서 업데이트를 적용할 때까지 원본으로 사용하는 데이터베이스 테이블로 다시 포스트되지 않습니다.

다른 레코드로 이동하기 전에 아무 필드에서나 *Esc* 키를 눌러 레코드에 대한 모든 편집을 취소할 수 있습니다.

그리드 그리기 제어

그리드 컨트롤에서 그리드를 그리는 방법을 제어하기 위한 첫 단계는 열 속성을 설정하는 것입니다. 그리드에서는 열의 글꼴, 색 및 정렬 속성을 자동으로 사용하여 해당 열의 셀을 그립니다. 데이터 필드의 텍스트는 열과 연결된 필드 컴포넌트의 *DisplayFormat*이나 *EditFormat* 속성을 사용하여 그려집니다.

그리드의 *OnDrawColumnCell* 이벤트에 있는 코드를 사용하여 디폴트 그리드 디스플레이 로직을 늘릴 수 있습니다. 그리드의 *DefaultDrawing* 속성이 **true**이면 *OnDrawColumnCell* 이벤트가 호출되기 전에 일반적인 그리기 작업이 모두 수행됩니다. 그러면 디폴트 디스플레이 위에 코딩한 내용이 그려질 수 있습니다. 이 방법은 비어 있는 영구적 열을 정의한 다음 해당 열의 셀에 특수 그래픽을 그릴 때 아주 유용합니다.

그리드의 그리기 로직을 완전히 대체하려면 *DefaultDrawing*을 **false**로 설정한 다음 그리기 코드를 그리드의 *OnDrawColumnCell* 이벤트에 둡니다. 특정 열이나 특정 필드 데이터 타입에서만 그리기 로직을 대체하려면 *OnDrawColumnCell* 이벤트 핸들러 내부에서 *DefaultDrawColumnCell*을 호출하여, 선택한 열에 대한 기본 그리기 코드를 그리드에서 사용하게 만들 수 있습니다. 예를 들어, 부울 필드 타입이 그려지는 방법만 변경하려는 경우 이렇게 하면 작업 내용이 줄어듭니다.

런타임 시 사용자 동작에 응답

이벤트 핸들러를 작성하여 그리드 동작을 수정하면 런타임 시 그리드 내에서의 특정 동작에 응답할 수 있습니다. 그리드는 일반적으로 한 번에 많은 필드와 레코드를 표시하므로 각각의 열의 변경 내용에 응답해야만 할 필요도 있습니다. 예를 들어, 사용자가 특정 열에 들어왔다 나갈 때마다 폼의 어디에서나 버튼을 활성화 및 비활성화할 수 있어야 합니다.

다음 표는 Object Inspector에서 사용할 수 있는 그리드 이벤트를 나열합니다.

표 19.6 그리드 컨트롤 이벤트

이벤트	용도
OnCellClick	사용자가 그리드의 특정 셀을 클릭할 때 발생
OnColEnter	사용자가 그리드의 특정 열로 이동할 때 발생
OnColExit	사용자가 그리드의 특정 열을 떠날 때 발생
OnColumnMoved	사용자가 새 위치의 특정 열로 이동할 때 발생
OnDblClick	사용자가 그리드에서 더블 클릭할 때 발생
OnDragDrop	사용자가 그리드에서 끌어다 놓을 때 발생
OnDragOver	사용자가 그리드 위를 끌 때 발생
OnDrawColumnCell	애플리케이션에서 개별 셀을 그려야 할 때 발생
OnDrawDataCell	(폐기됨) *State*가 *csDefault*인 경우 애플리케이션에 개별 셀을 그려야할 때 발생
OnEditButtonClick	사용자가 열에서 생략 부호 버튼을 클릭할 때 발생
OnEndDrag	사용자가 그리드에서 끄는 것을 멈출 때 발생
OnEnter	그리드에서 포커스를 가질 때 발생
OnExit	그리드에서 포커스를 잃을 때 발생
OnKeyDown	그리드에서 사용자가 키보드의 키나 키 조합을 누를 때 발생
OnKeyPress	그리드에서 사용자가 키보드의 단일 영숫자 키를 누를 때 발생
OnKeyUp	그리드에서 사용자가 키를 릴리스할 때 발생
OnStartDrag	사용자가 그리드에서 끌기를 시작할 때 발생
OnTitleClick	사용자가 열의 제목을 클릭할 때 발생

이러한 이벤트의 용도는 많습니다. 예를 들어, 사용자가 열에 입력할 값을 선택할 수 있는 리스트를 팝업하는 *OnDblClick* 이벤트의 핸들러를 작성할 수 있습니다. 이러한 핸들러에서는 *SelectedField* 속성을 사용하여 현재 행과 열을 확인할 수 있습니다.

기타 데이터 인식 컨트롤을 포함하는 그리드 생성

TDBCtrlGrid 컨트롤은 테이블 형식 그리드 형식에서 여러 레코드의 여러 필드를 표시합니다. 그리드의 각 셀은 단일 행의 여러 필드를 표시합니다 . 다음과 같은 방법으로 데이터베이스 컨트롤을 사용할 수 있습니다.

1. 데이터베이스 컨트롤 그리드를 폼에 둡니다.
2. 그리드의 *DataSource* 속성을 데이터 소스의 이름으로 설정합니다.
3. 개별 데이터 컨트롤을 그리드의 디자인 셀 내에 둡니다. 그리드의 디자인 셀은 그리드의 맨 위나 가장 왼쪽의 셀이며 다른 컨트롤을 둘 수 있는 유일한 셀입니다.
4. 각 데이터 컨트롤의 *DataField* 속성을 필드 이름으로 설정합니다. 이러한 데이터 컨트롤의 데이터 소스는 데이터베이스 컨트롤 그리드의 데이터 소스로 설정되어 있습니다.
5. 셀 내의 컨트롤을 원하는 대로 배열합니다.

데이터베이스 컨트롤 그리드를 포함하는 애플리케이션을 컴파일하고 실행하면, 런타임 시 디자인 셀에 설정한 데이터 컨트롤의 배열이 그리드의 각 셀에 복제됩니다. 각 셀은 데이터셋의 다른 레코드에 표시됩니다.

그림 19.5 디자인 타임 시 TDBCtrlGrid



다음 표는 디자인 타임 시 설정할 수 있는 데이터베이스 컨트롤 그리드에 대한 일부 고유 속성을 요약한 것입니다.

표 19.7 선택한 데이터베이스 컨트롤 그리드 속성

속성	용도
AllowDelete	**true**(기본값): 레코드 삭제를 허용
	false: 레코드 삭제를 방지
AllowInsert	**true**(기본값): 레코드 삽입을 허용
	false: 레코드 삽입을 방지
ColCount	그리드의 열 수를 설정. 기본값 = 1
Orientation	*goVertical* (기본값): 위에서 아래로 레코드 표시
	goHorizontal: 왼쪽에서 오른쪽으로 레코드 표시
PanelHeight	각 패널의 높이를 설정. 기본값 = 72
PanelWidth	각 패널의 너비 설정. 기본값 = 200
RowCount	표시할 패널 수 설정. 기본값 = 3
ShowFocus	**true**(기본값): 런타임 시 현재 레코드 패널 주위에 포커스 사각형 표시
	false: 포커스 사각형을 표시하지 않음

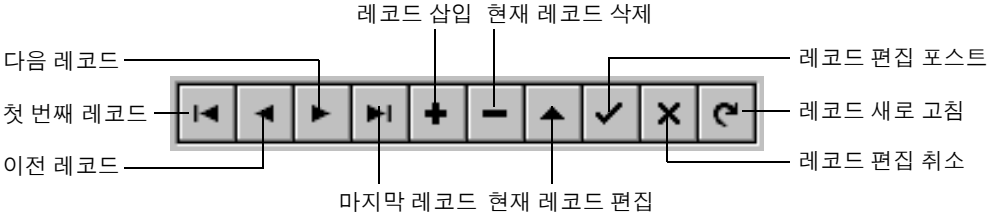
데이터베이스 컨트롤 그리드 속성과 메소드에 대한 자세한 내용은 온라인 *VCL Reference*를 참조하십시오.

레코드 탐색 및 처리

TDBNavigator 는 데이터셋의 레코드를 탐색하고 레코드를 처리하는 간단한 컨트롤을 사용자에게 제공합니다. 탐색기는 사용자가 한 번에 레코드를 앞 뒤로 스크롤, 첫 번째 레코드로 이동, 마지막 레코드로 이동, 새 레코드 삽입, 기존 레코드 업데이트, 데이터 변경 내용 포스트, 데이터 변경 취소, 레코드 삭제 및 레코드 표시 새로 고침을 가능하게 하는 버튼 집합으로 구성되어 있습니다.

그림 19.6은 디자인 타임 시 폼에 두면 디폴트로 표시되는 탐색기를 나타냅니다. 탐색기는 사용자가 데이터셋의 한 레코드에서 다른 레코드로 탐색한 다음 레코드를 편집, 삭제, 삽입 및 포스트할 수 있게 해주는 버튼 집합으로 구성되어 있습니다. 탐색기의 *VisibleButtons* 속성을 사용하면 이러한 버튼의 부분 집합을 동적으로 표시하거나 숨길 수 있습니다.

그림 19.6 TDBNavigator 컨트롤의 버튼



다음 표는 탐색기의 버튼을 설명합니다.

표 19.8 TDBNavigator 버튼

버튼	용도
First	데이터셋의 *First* 메소드를 호출하여 현재 레코드를 첫 번째 레코드로 설정
Prior	데이터셋의 *Prior* 메소드를 호출하여 현재 레코드를 이전 레코드로 설정
Next	데이터셋의 *Next* 메소드를 호출하여 현재 레코드를 다음 레코드로 설정
Last	데이터셋의 *Last* 메소드를 호출하여 현재 레코드를 마지막 레코드로 설정
Insert	데이터셋의 *Insert* 메소드를 호출하여 현재 레코드 앞에 새 레코드를 삽입하고 Insert 상태에서 데이터셋 설정
Delete	현재 레코드를 삭제. *ConfirmDelete* 속성이 **true**이면 삭제하기 전에 확인하는 메시지 표시
Edit	현재 레코드를 수정할 수 있도록 데이터셋을 Edit 상태에 둠
Post	현재 레코드의 변경 내용을 데이터베이스에 기록
Cancel	현재 레코드에 대한 편집을 취소하고 데이터셋을 Browse 상태로 반환
Refresh	데이터 컨트롤 표시 버퍼를 지운 다음 실제 테이블이나 쿼리로부터 버퍼를 새로 고침. 다른 애플리케이션에서 원본으로 사용하는 데이터를 변경한 경우에 유용

표시할 탐색기 버튼 선택

디자인 타임 시 *TDBNavigator*를 폼에 두면 폼의 모든 버튼을 볼 수 있습니다. *VisibleButtons* 속성을 사용하여 폼에서 사용하지 않을 버튼을 끌 수 있습니다. 예를 들어, 단방향 데이터셋에서 작업하는 경우 *First*와 *Next*, *Refresh* 버튼만 의미가 있습니다. 편집이 아니라 보기 위한 폼에서 *Edit*, *Insert*, *Delete*, *Post* 및 *Cancel* 버튼을 사용할 수 없게 만들 수 있습니다.

디자인 타임 시 탐색기 버튼 표시 및 숨기기

Object Inspector의 *VisibleButtons*속성이 + 부호와 함께 표시되면 탐색기의 각 버튼에 대한 부울 값을 표시하기 위해 확장이 가능하는 것을 의미합니다. 이러한 값을 보고 설정하려면 + 부호를 클릭합니다. 켜거나 끌 수 있는 버튼 리스트는 *VisibleButtons* 속성 아래의 Object Inspector에 표시됩니다. + 부호는 클릭하면 속성 리스트를 축소할 수 있는 –(빼기) 부호로 바뀝니다.

버튼 가시성은 버튼 값의 *부울* 상태에 의해 표시됩니다. 값이 **true**로 설정되면 버튼은 *TDBNavigator*에 나타납니다. **false**로 설정되면 버튼은 디자인 타임과 런타임 시 탐색기에서 제거됩니다.

참고 버튼 값이 **false**로 설정되어 있으면 폼의 *TDBNavigator*에서 제거되므로 나머지 버튼은 컨트롤을 채우기 위해 너비가 확장됩니다. 컨트롤의 핸들을 끌어서 버튼 크기를 조정할 수 있습니다.

런타임 시 탐색기 표시 및 숨기기

런타임 시 사용자 동작이나 애플리케이션 상태에 응답하여 탐색기 버튼을 표시하거나 숨길 수 있습니다. 예를 들어, 한 데이터셋은 사용자가 레코드를 편집할 수 있고 다른 데이터셋은 읽기 전용인, 두 개의 다른 데이터셋을 탐색하는 단일 탐색기를 제공한다고 가정합니다. 데이터셋 사이를 전환할 때 읽기 전용 데이터셋에 대해서는 탐색기의 *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, *Refresh* 버튼을 숨기고 다른 데이터셋에 대해서 이러한 버튼을 표시해야 할 것입니다.

예를 들어, 탐색기의 *Insert, Delete, Edit, Post, Cancel* 및 *Refresh* 버튼을 숨겨서 *OrdersTable*에 대한 편집을 막지만 *CustomersTable*에 대해서는 편집을 허용한다고 가정합니다. *VisibleButtons* 속성은 탐색기에 표시될 버튼을 제어합니다. 이벤트 핸들러를 작성하는 한 가지 방법은 다음과 같습니다.

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
 if (Sender == (TObject *)CustomerCompany)
 {
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbFirst << nbPrior
<< nbNext << nbLast;
  }
 else
  {
    DBNavigatorAll->DataSource = OrderNum->DataSource;
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbInsert << nbDelete
<< nbEdit
                                   << nbPost << nbCancel << nbRefresh;
  }
}
```

플라이오버(fly-over) 도움말 표시

런타임 시 각 탐색기 버튼에 대한 플라이오버(fly-over) 도움말을 표시하려면 *ShowHint* 속성을 **true**로 설정합니다. *ShowHint*가 **true**이면 마우스 커서가 탐색기 버튼을 지나갈 때마다 탐색기에서 플라이바이(fly-by) 도움말 힌트를 표시합니다. *ShowHint*의 기본값은 **false**입니다.

Hints 속성은 각 버튼에 대한 플라이오버(fly-over) 도움말 텍스트를 제어합니다. 디폴트로, *Hints*는 비어 있는 문자열 리스트입니다. *Hints*가 비어 있으면 각 탐색기 버튼은 디폴트 도움말 텍스트를 표시합니다. 탐색기 버튼에 사용자 정의 플라이오버 도움말을 제공하려면 String List Editor 를 사용하여 *Hints* 속성에 버튼에 대한 각각의 힌트 텍스트 줄을 입력합니다. 표시할 때 입력한 문자열이 탐색기 컨트롤에서 입력한 디폴트 힌트를 오버라이드합니다.

여러 데이터셋에서 단일 탐색기 사용

다른 데이터 인식 컨트롤과 마찬가지로 탐색기의 *DataSource* 속성도 데이터셋에 컨트롤을 연결하는 데이터 소스를 지정합니다. 런타임 시 탐색기의 *DataSource* 속성을 변경하면 단일 탐색기에서 여러 데이터셋에 대한 레코드 탐색 및 처리 기능을 제공합니다.

*CustomersSource*와 *OrdersSource* 데이터 소스 각각을 통해 *CustomersTable*과 *OrdersTable* 데이터셋에 연결된 두 개의 편집 컨트롤을 갖는 폼이 있다고 가정합니다. 사용자가 *CustomersSource*에 연결된 편집 컨트롤을 입력하면 탐색기에서도 *CustomersSource*를 사용하고, 사용자가 *OrdersSource*에 연결된 편집 컨트롤을 입력하면 탐색기에서도 *OrdersSource*를 사용합니다. 편집 컨트롤 중 하나에 대한 *OnEnter* 이벤트 핸들러를 작성한 다음 해당 이벤트를 다른 편집 컨트롤과 공유할 수 있습니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
 if (Sender == (TObject *)CustomerCompany)
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
 else
    DBNavigatorAll->DataSource = OrderNum->DataSource;
}
```

장

20

decision 지원 컴포넌트 사용

decision 지원 컴포넌트를 사용하면 크로스 테이블 형식이나 크로스탭 형식의 테이블 및 그래프를 만드는 데 도움이 됩니다. 이러한 테이블과 그래프를 사용하여 다양한 관점에서 데이터를 보고 요약할 수 있습니다. 크로스 테이블 형식 데이터에 대한 자세한 내용은 20-2페이지의 "크로스탭 정보"를 참조하십시오.

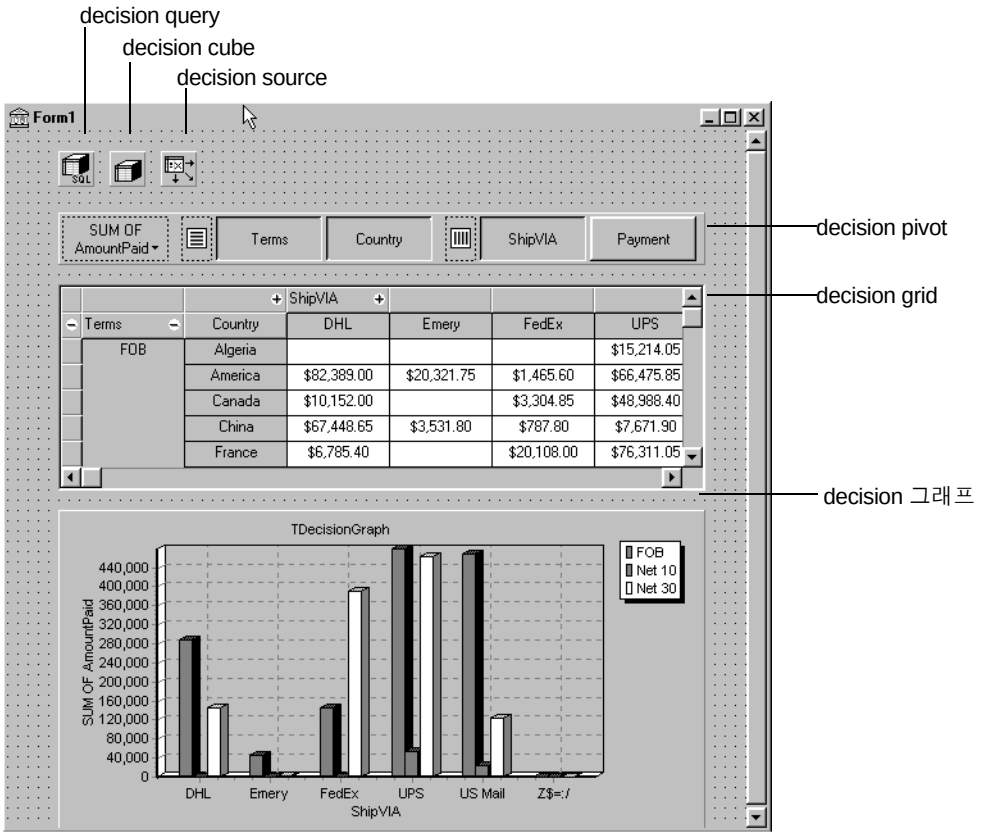
개요

decision 지원 컴포넌트는 다음과 같이 컴포넌트 팔레트의 Decision Cube 페이지에 나타납니다.

- decision cube인 *TDecisionCube*는 다차원 데이터 저장소입니다.
- decision source인 *TDecisionSource*는 decision grid나 decision 그래프의 현재 피벗 상태를 정의합니다.
- decision query인 *TDecisionQuery*는 decision cube의 데이터를 정의하는 데 사용되는 *TQuery*의 특화된 폼입니다.
- decision pivot인 *TDecisionPivot*을 사용하면 버튼을 눌러 decision cube 차원 또는 필드를 열거나 닫을 수 있습니다.
- decision grid인 *TDecisionGrid*는 테이블 폼에서 단일 및 다차원 데이터를 표시합니다.
- decision 그래프인 *TDecisionGraph* 는 decision grid 의 필드를 데이터 차원을 수정함에 따라 변경되는 동적 그래프로 표시합니다.

그림 20.1은 디자인 타임 시 폼에 위치한 모든 decision 지원 컴포넌트를 보여 줍니다.

그림 20.1 디자인 타임 시 decision 지원 컴포넌트



크로스탭 정보

크로스 테이블 형식이나 크로스탭은 관계와 경향을 쉽게 볼 수 있도록 데이터의 부분 집합을 표현하는 방법입니다. 테이블 필드는 크로스탭의 차원이 되고 필드 값은 차원 내에서 범주와 요약을 정의합니다.

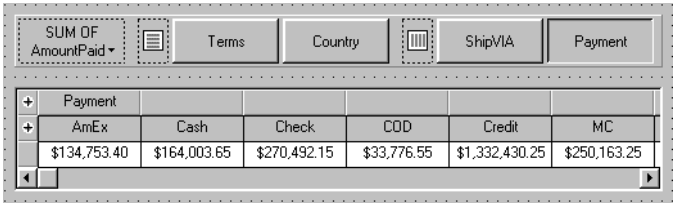
decision 지원 컴포넌트를 사용하여 폼에서 크로스탭을 설정할 수 있습니다. *TDecisionGrid*는 테이블의 데이터를 보여 주고 *TDecisionGraph*는 이 데이터를 그래픽으로 차트화합니다. *TDecisionPivot*에는 차원을 쉽게 표시하고 숨기며 열과 행 사이를 이동할 수 있게 하는 버튼이 있습니다.

크로스탭은 1차원이나 다차원이 될 수 있습니다.

1차원 크로스탭

1차원 크로스탭은 1차원 범주에 대한 요약 행 또는 열을 표시합니다. 예를 들어, Payment가 선택된 열 차원이고 Amount Paid가 요약 범주이면 그림 20.2의 크로스탭에는 각 메소드를 사용하여 지불된 금액이 표시됩니다.

그림 20.2 1차원 크로스탭

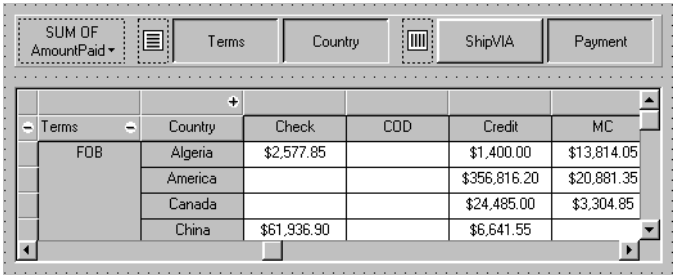


다차원 크로스탭

다차원 크로스탭은 행 및/또는 열에 대해 추가 차원을 사용합니다. 예를 들어, 2차원 크로스탭에서는 국가별로 지불 방법에 의해 지불된 금액을 표시할 수 있습니다.

3차원 크로스탭에서는 그림 20.3에 나타난 것처럼 국가별로 지불 방법과 조건에 따라 지불한 금액을 표시할 수 있습니다.

그림 20.3 3차원 크로스탭



decision 지원 컴포넌트 사용 지침

20-1 페이지에 나열된 decision 지원 컴포넌트를 함께 사용하면 다차원 데이터를 테이블과 그래프로 나타낼 수 있습니다. 두 개 이상의 그리드나 그래프를 각 데이터셋에 연결할 수 있습니다. 두 개 이상의 *TDecisionPivot* 인스턴스를 사용하여 런타임 시 다양한 관점에서 데이터를 표시할 수 있습니다.

다차원 데이터 테이블과 그래프가 있는 폼을 만들려면 다음 단계를 따릅니다.

1. 폼을 만듭니다.
2. 다음과 같은 컴포넌트를 폼에 추가하고 Object Inspector를 사용하여 표시된 대로 컴포넌트를 묶습니다.
   - 데이터셋인 *TDecisionQuery*(자세한 내용은 20-6페이지의 "Decision Query Editor를 사용하여 decision 데이터셋 생성" 참조) 또는 *TQuery*

- *DataSet* 속성을 데이터셋 이름으로 설정하여 데이터셋에 연결된 decision cube인 *TDecisionCube*
- *DecisionCube* 속성을 decision cube 이름으로 설정하여 decision cube에 연결된 decision source인 *TDecisionSource*

3 decision pivot인 *TDecisionPivot*을 추가한 다음 *DecisionSource* 속성을 적절한 decision source 이름으로 설정하여 Object Inspector에서 decision pivot을 decision source에 연결할 수 있습니다. decision pivot은 선택적이지만 유용합니다. decision pivot을 사용하면 폼 개발자와 엔드 유저가 버튼을 눌러 decision grid나 decision 그래프에 표시된 차원을 변경할 수 있습니다.

디폴트 방향인 가로 방향에서 decision pivot의 왼쪽 버튼은 decision grid(행)의 왼쪽 필드에 적용되고 오른쪽 버튼은 decision grid(열)의 맨 위에 있는 필드에 적용됩니다.

decision pivot의 *GroupLayout* 속성을 *xtVertical*, *xtLeftTop* 또는 *xtHorizontal*(기본값) 중 하나로 설정하여 decision pivot 버튼이 표시될 위치를 결정할 수 있습니다. decision pivot 속성에 대한 자세한 내용은 20-9페이지의 "decision pivot 사용"을 참조하십시오.

4 decision source에 연결된 하나 이상의 decision grid를 추가합니다. 자세한 내용은 20-10페이지의 "decision grid 생성 및 사용"과 20-13페이지의 "decision 그래프 생성 및 사용"을 참조하십시오.

5 Decision Query Editor나 *TDecisionQuery*(또는 *TQuery*)의 *SQL* 속성을 사용하여 테이블, 필드 및 요약을 지정하고 그리드나 그래프에 표시합니다. SQL SELECT의 마지막 필드는 요약 필드가 되어야 합니다. SELECT의 다른 필드는 GROUP BY 필드가 되어야 합니다. 자세한 내용은 20-6 페이지의 "Decision Query Editor를 사용하여 decision 데이터셋 생성"을 참조하십시오.

6 decision query나 대체 데이터셋 컴포넌트의 *Active* 속성을 **true**로 설정합니다.

7 decision grid와 decision 그래프를 사용하여 다양한 데이터 차원을 표시하고 차트화합니다. 자세한 지침 및 제안은 20-11 페이지의 "decision grid 사용"과 20-13 페이지의 "decision 그래프 사용"을 참조하십시오.

폼의 모든 decision 지원 컴포넌트에 대한 자세한 내용은 20-2페이지의 그림 20.1을 참조하십시오.

decision 지원 컴포넌트를 사용하여 데이터셋 사용

데이터셋에 직접 연결하는 유일한 decision 지원 컴포넌트는 decision cube인 *TDecisionCube*입니다. *TDecisionCube*에서는 받아들일 수 있는 형식의 SQL 문에서 정의한 그룹 및 요약과 함께 데이터를 받기를 기대합니다. GROUP BY 구에는 SELECT 구와 같은 요약되지 않은 필드가 같은 순서로 포함되어 있어야 하며 요약 필드는 정의되어야 합니다.

decision query 컴포넌트인 *TDecisionQuery*는 *TQuery*의 특화된 폼입니다. TDecisionQuery를 사용하여 차원 설정(열과 행) 정의와 decision cube(*TDecisionCube*)에 데이터를 제공하는 데 사용되는 요약 값 정의를 더 간단하게 할 수 있습니다. 일반적인 *TQuery*나 기타 BDE 호환 데이터셋을 *TDecisionCube*의 데이터셋으로 사용할 수도 있지만 데이터셋과 *TDecisionCube*의 올바른 설정에 대한 책임은 디자이너에게 있습니다.

decision cube로 올바르게 작업하려면 데이터셋의 모든 Projected field는 차원이거나 요약 중 하나이어야 합니다. 요약은 합계나 카운트처럼 가산 값이어야 하며 차원 값의 조합별로 합계를 나타내야 합니다. 설정 과정을 최대한 단순화하려면 카운트의 이름은 "Count..."여야 하며 합계의 이름은 데이터셋에서 "Sum..."이어야 합니다.

decision cube에서는 요약 셀이 가산 요약에 대해서만 올바르게 피벗, 부분합, 드릴인(drill-in)이 가능합니다. SUM과 COUNT는 누적되지만 AVERAGE, MAX, MIN은 누적되지 않습니다. 빌드 피벗팅 크로스탭에서는 가산 aggregator만 포함되어 있는 그리드만 표시합니다. 비가산 aggregator를 사용하고 있다면 피벗이나 드릴, 부분합을 하지 않는 정적 decision 드릴을 사용해야 합니다.

평균은 COUNT로 나눈 SUM을 사용하여 계산이 가능하므로 필드에 대한 SUM과 COUNT 차원이 데이터셋에 포함되면 평균 피벗팅이 자동으로 추가됩니다. 이러한 평균 계산 방식이 AVERAGE 문을 사용하여 평균을 사용하는 것보다 선호됩니다.

COUNT(*)를 사용하여 평균을 계산할 수도 있습니다. COUNT(*)를 사용하여 평균을 계산하려면 쿼리에 "COUNT(*) COUNTALL" 선택자를 포함해야 합니다. COUNT(*)를 사용하여 평균을 계산하면 모든 필드에 하나의 aggregator를 사용할 수 있습니다. 요약할 필드에 비어 있는 값이 없거나 모든 필드에 대해 COUNT aggregator를 사용할 수 있는 경우가 아닐 때만 COUNT(*)를 사용해야 합니다.

TQuery나 TTable을 사용하여 decision 데이터셋 생성

일반적인 *TQuery* 컴포넌트를 decision 데이터셋으로 사용하는 경우 SELECT 구와 같은 필드를 같은 순서로 포함하는 GROUP BY 구를 제공하면서 SQL 문을 수동으로 설정해야 합니다.

SQL의 형태는 다음과 같아야 합니다.

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

SELECT 필드의 순서는 GROUP BY 필드의 순서와 일치해야 합니다.

*TTable*을 사용하는 경우 쿼리의 필드 중 어느 것이 그룹화 필드이며 요약인지에 관한 정보를 decision cube에 제공해야 합니다. 이 작업을 하려면 decision cube의 *DimensionMap*에서 각 필드의 Dimension Type을 채워야 합니다. 각 필드가 차원인지 아니면 요약인지를 표시해야 하며 요약인 경우 요약 타입을 입력해야 합니다. 평균 피벗팅은 SUM/COUNT 계산에 따라 달라지므로 기본 필드 이름도 제공하여 decision cube에서 SUM과 COUNT aggregator 쌍을 비교할 수 있게 해야 합니다.

Decision Query Editor를 사용하여 decision 데이터셋 생성

decision 지원 컴포넌트에서 사용하는 모든 데이터는 decision cube를 통과하며, decision cube는 SQL 쿼리에서 가장 쉽게 만들 수 있는 특별한 형식의 데이터셋을 받아들입니다. 자세한 내용은 20-4페이지의 "decision 지원 컴포넌트를 사용하여 데이터셋 사용"을 참조하십시오.

*TTable*과 *TQuery*는 모두 decision 데이터셋으로 사용될 수 있지만 *TDecisionQuery*를 사용하는 것이 더 쉽습니다. 왜냐하면 TDecisionQuery와 함께 제공되는 Decision Query Editor를 사용하면 테이블, 필드 및 요약이 decision cube에 나타나도록 지정할 수 있으며 SQL의 SELECT와 GROUP BY 부분을 올바르게 설정하도록 도와 줍니다.

다음과 같은 방법으로 Decision Query Editor를 사용합니다.

1. 폼에서 의사 결정 쿼리 컴포넌트를 선택한 다음 마우스 오른쪽 버튼을 클릭하여 Decision Query Editor를 선택합니다. Decision Query Editor 다이얼로그 박스가 나타납니다.
2. 사용할 데이터베이스를 선택합니다.
3. 단일 테이블 쿼리의 경우 Select Table 버튼을 클릭합니다.

   다중 테이블 조인과 관련된 다소 복잡한 쿼리의 경우 Query Builder 버튼을 클릭하여 SQL Builder를 표시하거나 SQL 문을 SQL 탭 페이지의 에디트 박스에 입력합니다.
4. Decision Query Editor 다이얼로그 박스로 돌아갑니다.
5. Decision Query Editor 다이얼로그 박스에서 Available Fields 리스트 박스의 필드를 선택한 다음 적절한 오른쪽 화살표 버튼을 클릭하여 Dimensions나 Summaries 중 하나를 할당합니다. Summaries 리스트에 필드를 추가할 때 합, 카운트, 평균 등 사용할 요약 타입이 표시되어 있는 메뉴에서 선택합니다.
6. 디폴트로, decision query의 *SQL* 속성에 정의되어 있는 모든 필드와 요약은 Active Dimensions와 Active Summaries 리스트 박스에 나타납니다. 차원이나 요약을 제거하려면 리스트에서 선택한 다음 리스트 옆의 왼쪽 화살표를 클릭하거나 제거할 항목을 더블 클릭합니다. 차원이나 요약을 다시 추가하려면 Available Fields 리스트 박스에서 선택한 다음 적절한 오른쪽 화살표를 클릭합니다.

decision cube 내용을 정의한 다음에는 차원 *DimensionMap* 속성과 *TDecisionPivot* 버튼을 사용하여 차원 표시를 계속 처리할 수 있습니다. 자세한 내용은 다음 단원인 "decision cube 사용"과 20-8페이지의 "decision source 사용" 및 20-9페이지의 "decision pivot 사용"을 참조하십시오.

참고 Decision Query Editor를 사용할 때 쿼리는 ANSI-92 SQL 구문에서 먼저 처리된 다음 필요하면 서버에서 사용하는 언어로 번역됩니다. Decision Query Editor에서는 ANSI 표준 SQL만 읽고 표시합니다. 해당 언어의 번역문은 *TDecisionQuery*의 SQL 속성에 자동으로 할당됩니다. 쿼리를 수정하려면 SQL 속성이 아닌 Decision Query의 ANSI-92 버전을 편집해야 합니다.

decision cube 사용

decision cube 컴포넌트인 *TDecisionCube*는 데이터셋에서 데이터를 가져오는 다차원 데이터 저장소입니다. 반면 특수 구조 SQL 문은 일반적으로 *TDecisionQuery*나 *TQuery*를 통해 입력됩니다. 데이터는 쿼리를 다시 실행하지 않아도 쉽게 피벗, 즉 데이터 구성 및 요약 방식을 변경할 수 있는 폼에 저장됩니다.

decision cube 속성과 이벤트

*TDecisionCube*의 *DimensionMap* 속성에서는 표시될 차원과 요약을 제어할 뿐 아니라 데이터 범위를 설정하고 decision cube에서 지원할 최대 차원 수를 지정합니다. 그리고 디자인하는 동안 데이터 표시 여부도 나타낼 수 있습니다. 이름이나 범주 값, 부분 합, 데이터를 표시할 수 있습니다. 디자인 타임 시 데이터를 표시하면 데이터 소스에 따라 시간이 많이 걸릴 수 있습니다.

Object Inspector의 *DimensionMap* 옆의 생략 부호를 클릭하면 Decision Cube Editor 다이얼로그 박스가 나타납니다 . 이 다이얼로그 박스의 페이지와 컨트롤을 사용하여 *DimensionMap* 속성을 설정할 수 있습니다.

decision cube 캐시가 다시 생성될 때마다 *OnRefresh* 이벤트가 발생합니다. 개발자는 새 차원 맵을 액세스하여 메모리를 해제하도록 변경하거나 최대 요약이나 차원 수를 변경하는 등의 작업을 할 수 있습니다. 사용자가 Decision Cube Editor 에 액세스하는 경우에도 *OnRefresh* 가 유용합니다. 애플리케이션 코드에서 그 순간에 사용자 변경에 반응할 수 있기 때문입니다.

Decision Cube Editor 사용

Decision Cube Editor를 사용하여 decision cube의 *DimensionMap* 속성을 설정할 수 있습니다. 이전 단원에서 설명한 대로 Object Inspector를 통해서나 디자인 시 폼의 decision cube를 마우스 오른쪽 버튼으로 클릭하고 Decision Cube Editor를 선택하여 Decision Cube Editor를 표시할 수 있습니다.

Decision Cube Editor 다이얼로그 박스에는 다음과 같은 두 개의 탭이 있습니다.

- Dimension Settings - 사용 가능한 차원을 활성화하거나 비활성화, 차원의 이름을 바꾸고 다시 포맷, 차원을 영구적 드릴 상태에 두거나 표시할 날짜 범위를 설정하는 데 사용
- Memory Control - 한 번에 활성화될 수 있는 차원과 요약 수를 설정하고 메모리 사용에 대한 정보를 표시하며 디자인 타임시 나타나는 이름과 데이터를 결정하는 데 사용

차원 설정 보기 및 변경

차원 설정을 보려면 Decision Cube Editor를 표시한 다음 Dimension Settings 탭을 클릭합니다. 그리고 나서 Available Fields 리스트에서 차원이나 요약을 선택합니다. 관련 정보는 에디터의 오른쪽 박스에 다음과 같이 나타납니다.

- decision pivot이나 decision grid, decision 그래프에 나타나는 차원이나 요약을 변경하려면 Display Name 에디트 박스에 새 이름을 입력합니다.
- 선택한 필드가 차원인지 요약인지 확인하려면 Type 에디트 박스에서 텍스트를 읽습니다. 데이터셋이 *TTable* 컴포넌트인 경우 Type을 사용하여 선택한 필드가 차원인지 아니면 요약인지를 지정할 수 있습니다.
- 선택한 차원이나 요약을 활성화하거나 비활성화하려면 Active Type 드롭다운 리스트 박스의 설정을 Active, As Needed 또는 Inactive로 변경합니다. 차원을 비활성화하거나 As Needed로 설정하면 메모리가 절약됩니다.
- 해당 차원이나 요약의 형식을 변경하려면 Format 에디트 박스에 서식 문자열을 입력합니다.
- Year나 Quarter, Month별로 차원이나 요약을 표시하려면 Binning 드롭다운 리스트 박스의 설정을 변경합니다. Binning 리스트 박스의 Set을 선택하여 선택한 차원이나 요약을 영구적으로 "드릴다운" 상태에 둘 수 있습니다. 차원에 값이 많이 있는 경우 드릴다운 상태에 두면 메모리를 절약할 수 있습니다. 자세한 내용은 20-19페이지의 "decision 지원 컴포넌트와 메모리 제어"를 참조하십시오.
- 범위 시작 값이나 "Set" 차원의 드릴다운 값을 확인하려면 먼저 Grouping 드롭다운에서 먼저 적절한 Grouping 값을 선택한 다음 Initial Value 드롭다운 리스트에 시작 범위 값이나 영구적 드릴다운 값을 입력합니다.

사용 가능한 최대 차원 및 요약 수 설정

선택한 decision cube에 연결된 decision pivot, decision grid 및 decision 그래프에서 사용할 수 있는 차원 및 요약의 최대 수를 확인하려면 Decision Cube Editor를 표시하고 Memory Control 탭을 클릭합니다. 필요한 편집 컨트롤을 사용하여 현재 설정을 조정합니다. 이러한 설정은 decision cube에서 필요한 메모리 크기를 제어하도록 도와 줍니다. 자세한 내용은 20-19페이지의 "decision 지원 컴포넌트와 메모리 제어"를 참조하십시오.

디자인 옵션 보기 및 변경

디자인 타임 시 표시되는 정보 양을 결정하려면 Decision Cube Editor를 표시하여 Memory Control 탭을 클릭합니다. 그리고 나서 표시할 이름과 데이터를 나타내는 설정을 확인합니다. 디자인 타임 시 데이터나 필드 표시는 데이터를 가져오는 데 필요한 시간 때문에 성능이 저하되는 경우도 있습니다.

decision source 사용

decision source 컴포넌트인 *TDecisionSource*는 decision grid나 decision 그래프의 현재 피벗 상태를 정의합니다. 같은 decision source를 사용하는 두 개의 객체도 피벗 상태를 공유할 수 있습니다.

속성과 이벤트

다음에는 decision source의 모양 및 동작을 제어하는 몇 가지 특수 속성과 이벤트가 나와 있습니다.

- *TDecisionSource*의 *ControlType* 속성에서는 decision pivot 버튼이 여러 개를 선택할 수 있는 체크 박스나 상호 배타적으로 선택하는 라디오 버튼 중 어떤 형식으로 동작해야 하는지 표시합니다.
- *TDecisionSource*의 *SparseCols*와 *SparseRows* 속성은 값이 없는 열이나 행을 표시할 것인지 여부를 표시합니다. 이 속성 값이 **true**인 경우 값이 드문 드문 비어있는 열이나 행이 표시됩니다.
- *TDecisionSource*에는 다음과 같은 이벤트가 있습니다.
  - *OnLayoutChange*는 데이터를 다시 구성하는 피벗이나 드릴다운을 수행할 때 발생합니다.
  - *OnNewDimensions*는 요약이나 차원 필드가 대체되는 경우처럼 데이터가 완전히 대체될 때 발생합니다.
  - *OnSummaryChange*는 현재 요약이 변경될 때 발생합니다.
  - *OnStateChange*는 Decision Cube가 활성화되거나 비활성화될 때 발생합니다.
  - *OnBeforePivot*은 변경 내용이 커밋되었지만 아직 사용자 인터페이스에 반영되지 않았을 때 발생합니다. 애플리케이션 사용자들이 이전 작업의 결과를 보기 전에 개발자는 용량이나 피벗 상태 등을 변경할 수 있습니다.
  - *OnAfterPivot*은 피벗 상태의 변경 내용 다음에 발생합니다. 개발자는 그 순간에 정보를 캡처할 수 있습니다.

decision pivot 사용

decision pivot 컴포넌트인 *TDecisionPivot*을 사용하면 버튼을 눌러 decision cube 차원이나 필드를 열고 닫을 수 있습니다. *TDecisionPivot* 버튼을 눌러 행이나 열을 열면 이에 해당하는 차원이 *TDecisionGrid*나 *TDecisionGraph* 컴포넌트에 나타납니다. 차원이 닫히면 디테일 데이터는 나타나지 않고 다른 차원의 합계로 축소 표시됩니다. 차원은 차원 필드의 특정 값에 대한 요약만 표시되는 "드릴" 상태에 있을 수도 있습니다.

decision pivot을 사용하여 decision grid와 decision 그래프에 표시된 차원을 다시 구성할 수도 있습니다. 버튼을 행이나 열 영역으로 끌거나 같은 영역 내에서 버튼 순서를 바꿀 수 있습니다.

디자인 타임 시 decision pivot에 대한 자세한 내용은 표 20.1, 20.2, 및 20.3을 참조하십시오.

decision pivot 속성

다음에는 decision pivot의 모양 및 동작을 제어하는 몇 가지 특수한 속성이 나와 있습니다.

- *TDecisionPivot*에 대해 나열되어 있는 첫 번째 속성들은 전반적인 모양과 동작을 정의합니다. *TDecisionPivot*에 대해 *ButtonAutoSize*를 **false**로 설정하면 컴포넌트의 크기를 조정할 때 버튼이 확장 및 축소되는 것을 막을 수 있습니다.
- *TDecisionPivot*의 *Groups* 속성은 표시될 차원 버튼을 정의합니다. 행, 열, 요약 선택 버튼 그룹을 어떤 조합으로든지 표시할 수 있습니다. 이러한 그룹 배치에 좀 더 유용성이 필요한 경우 행만 있는 폼의 *TDecisionPivot*을 한 위치에 배치하고 열만 있는 두 번째 폼의 TDecisionPivot을 다른 위치에 배치할 수 있습니다.
- 일반적으로 *TDecisionPivot*은 *TDecisionGrid* 위에 추가됩니다. 디폴트 방향인 가로 방향에서 *TDecisionPivot*의 왼쪽 버튼은 *TDecisionGrid*(행)의 왼쪽 필드에 적용되고, 오른쪽 버튼은 *TDecisionGrid*(열)의 맨 위에 있는 필드에 적용됩니다.
- *TDecisionPivot* 버튼의 *GroupLayout* 속성을 *xtVertical*, *xtLeftTop*, *xtHorizontal*(기본값, 이전 단락에서 설명) 중 하나로 설정하여 TDecisionPivot 버튼이 표시될 위치를 결정할 수 있습니다.

decision grid 생성 및 사용

decision grid 컴포넌트인 *TDecisionGrid*는 테이블 폼의 크로스 테이블 형식 데이터를 나타냅니다. 이러한 테이블은 20-2페이지에서 설명한 대로 크로스탭이라고도 합니다. 20-2페이지의 그림 20.1은 디자인 타임 시 폼의 decision grid를 보여 줍니다.

decision grid 생성

다음과 같은 방법으로 하나 이상의 크로스 테이블 형식 데이터의 테이블을 사용하여 폼을 만듭니다.

1. 20-3페이지의 "decision 지원 컴포넌트 사용 지침"에 있는 단계 1-3을 따릅니다.
2. Object Inspector를 사용하여 decision grid 컴포넌트의 *DecisionSource* 속성을 적절한 decision source 컴포넌트로 설정하면, 한 개 이상의 decision grid 컴포넌트(*TDecisionGrid*)를 추가한 다음 decision source인 *TDecisionSource*에 연결할 수 있습니다.
3. "decision 지원 컴포넌트 사용 지침"에 나열되어 있는 단계 5-7을 계속 진행합니다.

decision grid에 나타나는 모양 및 그 사용 방법에 대한 자세한 내용은 20-11페이지의 "decision grid 사용"을 참조하십시오.

그래프를 폼에 추가하려면 20-13페이지의 "decision 그래프 생성"의 지침을 따릅니다.

decision grid 사용

decision grid 컴포넌트인 *TDecisionGrid*는 decision source(*TDecisionSource*)에 연결된 decision cube(*TDecisionCube*)의 데이터를 표시합니다.

디폴트로, 그리드는 데이터셋에 정의되어 있는 그룹화 지침에 따라 왼쪽 및/또는 맨 위에 차원 필드와 함께 표시됩니다. 각 데이터 값에 대한 범주는 각 필드 아래 표시되며 다음 작업을 할 수 있습니다.

- 차원 열기 및 닫기
- 행과 열 재구성이나 피벗
- 세부 사항 드릴다운
- 차원 선택을 각 축에 대한 단일 차원으로 제한

decision grid의 특수 속성 및 이벤트에 대한 자세한 내용은 20-12페이지의 "decision grid 속성"을 참조하십시오.

decision grid 필드 열기 및 닫기

더하기 부호(+)가 차원이나 요약 필드에 나타나면 오른쪽에 있는 하나 이상의 필드가 닫히거나 숨겨집니다. 더하기 부호를 클릭하여 추가 필드와 범주를 열 수 있습니다. 빼기 부호(-)는 완전히 열려 있는, 확장된 필드를 표시합니다. 빼기 부호를 클릭하면 해당 필드가 닫힙니다. 이러한 윤곽 기능을 사용할 수 없게 만들 수 있는데 자세한 내용은 20-12페이지의 "decision grid 속성"을 참조하십시오.

decision grid의 행과 열 재구성

행과 열의 머리글을 같은 축 내의 새 위치나 다른 축으로 끌 수 있습니다. 이러한 식으로 그리드를 재구성하여 데이터 그룹화가 변경되면 새로운 관점에서 그리드를 볼 수 있습니다. 이러한 피벗팅 기능을 사용할 수 없게 만들 수 있는데 자세한 내용은 20-12페이지의 "decision grid 속성"을 참조하십시오.

decision pivot이 포함되어 있으면 해당 버튼을 누르거나 끌어 디스플레이를 재구성할 수 있습니다. 자세한 내용은 20-9페이지의 "decision pivot 사용"을 참조하십시오.

decision grid에서 세부 사항을 드릴다운

드릴다운하면 차원에 대한 세부 사항을 볼 수 있습니다.

예를 들어, 다른 내용이 밑에 숨겨져 있는 차원의 범주 레이블(열 머리글)을 마우스 오른쪽 버튼으로 클릭하면 드릴다운을 선택하여 해당 범주의 데이터만 볼 수 있습니다. 차원이 드릴되는 경우 단일 범주 값에 대한 레코드만 표시되므로 그리드에 표시되는 해당 차원에 대한 범주 레이블은 볼 수 없습니다. 폼에 decision pivot이 있는 경우 범주 값이 표시되므로 원하면 다른 값으로 변경할 수 있습니다.

다음과 같은 방법으로 차원에 드릴다운합니다.

- 범주 레이블을 마우스 오른쪽 버튼으로 클릭한 다음 Drill In To This Value를 선택하거나
- 피벗 버튼을 마우스 오른쪽 버튼으로 클릭한 다음 Drilled In을 선택합니다.

다음과 같은 방법으로 완전한 차원을 다시 활성화합니다.

- 해당 피벗 버튼을 마우스 오른쪽 버튼으로 클릭하거나 왼쪽 위 모서리의 decision grid를 클릭하여 차원을 선택합니다.

decision grid의 차원 선택 제한

decision source 의 *ControlType* 속성을 변경하여 그리드의 각 축에 대해 두 개 이상의 차원을 선택할 수 있는지 결정할 수 있습니다. 자세한 내용은 20-8페이지의 "decision source 사용"을 참조하십시오.

decision grid 속성

decision grid 컴포넌트인 *TDecisionGrid*는 *TDecisionSource*에 연결된 *TDecisionCube* 컴포넌트의 데이터를 표시합니다. 디폴트로, 데이터는 그리드의 왼쪽과 맨 위에 범주 필드가 있는 그리드에 표시됩니다.

다음에는 decision grid의 모양 및 동작을 제어하는 몇 가지 특수 속성이 나와 있습니다.

- *TDecisionGrid*에는 각 차원에 대한 고유 속성이 있습니다. 이러한 속성을 설정하려면 Object Inspector에서 *Dimensions*를 선택한 다음 차원을 선택합니다. 그러면 해당 속성이 Object Inspector에 표시됩니다. *Alignment*는 해당 차원의 범주 레이블 정렬을 정의합니다. *Caption*을 사용하면 디폴트 차원 이름을 오버라이드할 수 있습니다. *Color*는 범주 레이블의 색을 정의합니다. *FieldName*은 활성 차원의 이름을 표시합니다. *Format*은 해당 데이터 타입의 표준 형식을 유지합니다. 그리고 *Subtotals*는 해당 차원에 대한 부분합을 표시할지 여부를 표시합니다. 요약 필드를 사용하는 경우 이러한 동일한 속성을 사용하여 그리드의 요약 영역에 나타나는 데이터 모양을 변경할 수 있습니다. 차원 속성 설정이 끝났으면 폼의 컴포넌트를 클릭하거나 Object Inspector의 맨 위에 있는 드롭다운 리스트 박스에서 컴포넌트를 선택합니다.
- *TDecisionGrid*의 *Options* 속성을 사용하면 +와 - 지시자로 차원을 축소 및 확장하여 윤곽 기능을 활성화하고(*cgOutliner* = **true**) 드래그 앤 드롭 피벗팅을 활성화하면서(*cgPivotable* = **true**) 그리드 라인의 표시를 제어할 수 있습니다(*cgGridLines* = **true**).
- *TDecisionGrid*의 *OnDecisionDrawCell* 이벤트를 사용하면 각 셀의 모양을 그려진 대로 변경할 수 있습니다. 이벤트에서는 현재 셀의 *String*, *Font* 및 *Color*를 참조 매개변수로 전달합니다. 이러한 매개변수를 마음대로 대체하여 음수 값에 대해 특수한 색을 만드는 등의 효과를 낼 수 있습니다. *TCustomGrid*에서 전달하는 *DrawState*와 함께 이벤트에는 그려질 셀 타입을 결정하는 데 사용할 수 있는 *TDecisionDrawState*를 전달합니다. 셀에 대한 자세한 내용은 *Cells*, *CellValueArray*나 *CellDrawState* 함수를 사용하여 가져올 수 있습니다.
- *TDecisionGrid* 의 *OnDecisionExamineCell* 이벤트를 사용하면 마우스 오른쪽 버튼으로 클릭한 이벤트(right-click-on-event)를 데이터 셀로 훅하여 프로그램에서 특정 데이터 셀에 대한 디테일 레코드 등의 정보를 표시하게 할 수 있습니다. 사용자가 데이터 셀을 마우스 오른쪽 버튼으로 클릭하면 현재 활성 중인 요약 값과 요약 값을 만들 때 사용되었던 모든 차원 값의 *ValueArray* 를 비롯하여 데이터 값을 구성하는 데 사용되었던 모든 정보가 이벤트에 제공됩니다.

decision 그래프 생성 및 사용

decision 그래프 컴포넌트인 *TDecisionGraph* 는 크로스 테이블 형식 데이터를 그래픽 형식으로 나타냅니다. 각 decision 그래프에는 하나 이상의 차원에 대해 차트화된 Sum이나 Count, Avg와 같은 단일 요약 값이 표시됩니다. 크로스탭에 대한 자세한 내용은 20-3페이지를 참조하십시오. 디자인 타임 시 decision 그래프에 대한 자세한 내용은 20-2페이지의 그림 20.1과 20-14페이지의 그림 20.4를 참조하십시오.

decision 그래프 생성

다음과 같은 방법으로 하나 이상의 decision 그래프로 폼을 만듭니다.

1. 20-3페이지의 "decision 지원 컴포넌트 사용 지침" 아래에 나열된 단계 1-3을 따릅니다.
2. 하나 이상의 decision 그래프 컴포넌트 (*TDecisionGraph*) 를 추가한 다음 Object Inspector 에서 *DecisionSource* 속성을 적절한 decision source 컴포넌트로 설정하여 decision 그래프 컴포넌트를 decision source인 *TDecisionSource*와 연결합니다.
3. "decision 지원 컴포넌트 사용 지침" 아래에 나열된 단계 5-7을 따릅니다.
4. 마지막으로 마우스 오른쪽 버튼으로 그래프를 클릭한 다음 Edit Chart를 선택하여 그래프 시리즈의 모양을 수정합니다. 각 그래프 차원의 템플릿 속성을 설정한 다음 개별 시리즈 속성을 설정하여 이러한 기본값을 오버라이드할 수 있습니다. 자세한 내용은 20-15페이지의 "decision 그래프 사용자 정의"를 참조하십시오.

decision 그래프의 모양 및 그래프 사용 방법에 대한 자세한 내용은 다음 단원인 "decision 그래프 사용"을 참조하십시오.

decision grid 나 크로스탭 테이블을 폼에 추가하려면 20-10 페이지의 "decision grid 생성 및 사용"에 나와 있는 지침을 따릅니다.

decision 그래프 사용

decision 그래프 컴포넌트인 *TDecisionGraph*는 decision source(*TDecisionSource*)의 필드를 decision pivot(*TDecisionPivot*)과 함께 데이터 차원이 열리고 닫히며 드래그 앤 드롭되고 재정렬될 때마다 변경되는 동적 그래프로 나타냅니다.

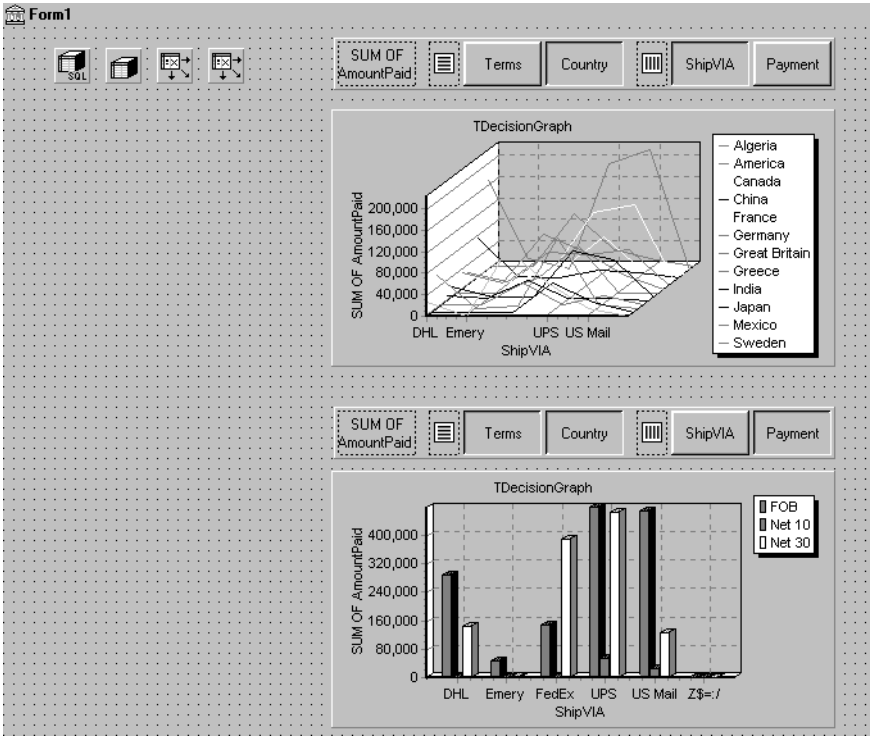
그래프 데이터는 *TDecisionQuery* 와 같은 특수한 서식의 데이터셋에서 제공됩니다. decision 지원 컴포넌트에서 이러한 데이터를 처리하고 정렬하는 방법에 대한 개요는 20-1페이지를 참조하십시오.

디폴트로, 첫 행 차원은 x축으로 첫 열 차원은 y축으로 표시됩니다.

크로스탭 데이터를 테이블 형식 폼으로 나타내는 decision grid 대신, 또는 decision grid 와 함께 decision 그래프를 사용할 수 있습니다. 동일한 decision source에 연결된 decision grid와 decision 그래프는 동일한 데이터 차원을 나타냅니다. 동일한 차원에 대해 다른 요약 데이터를 표시하려면 두 개 이상의 decision 그래프를 동일한 decision source에 연결하면 됩니다. 다른 차원을 표시하려면 decision 그래프를 decision source에 연결합니다.

예를 들어, 그림 20.4에서는 첫 번째 decision pivot과 decision 그래프가 첫 번째 decision source에 연결되어 있고 두 번째 decision pivot 과 decision 그래프는 두 번째 decision source 에 연결되어 있습니다. 따라서 그래프마다 다른 차원을 표시할 수 있습니다.

그림 20.4 다른 decision source에 연결된 decision 그래프



decision 그래프에 나타나는 내용에 대한 자세한 내용은 다음 단원인 "decision 그래프 표시"를 참조하십시오.

decision 그래프를 만들려면 이전 단원인 "decision 그래프 생성"을 참조하십시오.

decision 그래프 속성과 decision 그래프의 모양 및 동작을 변경하는 방법에 대한 자세한 내용은 20-15페이지의 "decision 그래프 사용자 정의"를 참조하십시오.

decision 그래프 표시

디폴트로, decision 그래프는 범주의 요약 값을 첫 번째 활성 열 필드(x축을 따라)의 값에 대한 첫 번째 활성 행 필드(y축을 따라)에 그립니다. 그래프 범주는 각각의 시리즈로 나타납니다.

차원을 한 개만 선택하면, 예를 들어, 한 *TDecisionPivot* 버튼만 클릭하면 한 시리즈만 그래프로 표시됩니다.

decision pivot을 사용한 경우에는 decision pivot 버튼을 눌러 그래프로 나타낼 decision cube 필드(차원)를 결정할 수 있습니다. 그래프 축을 바꾸려면 구분자 영역의 한 쪽에서 다른 쪽으로 decision pivot 버튼을 끌어 놓습니다. 모든 버튼이 구분자 영역의 한 쪽에 있는 1차원 그래프가 있다면 Row나 Column 아이콘을 구분자의 다른 쪽으로 버튼을 추가하기 위한 드롭 대상으로 사용하여 해당 그래프를 다차원으로 만들 수 있습니다.

한 번에 한 열과 행만 활성화되게 하려면 *TDecisionSource*의 *ControlType* 속성을 *xtRadio*로 설정하면 됩니다. 그러면 각 decision cube 축에 대해 한 번에 한 개의 활성 필드만 있게 되며 decision pivot의 기능은 그래프 작동에 대응하게 됩니다. *xtRadioEx*는 *xtRadio*와 똑같이 작동하지만 행이나 열 차원이 모두 닫히는 것은 허용하지 않습니다.

decision grid와 decision 그래프가 모두 동일한 *TDecisionSource*에 연결되어 있는 경우 *TDecisionGrid*의 보다 유연한 동작에 대응하기 위해 *ControlType*을 다시 *xtCheck*로 설정할 수 있습니다.

decision 그래프 사용자 정의

decision 그래프 컴포넌트인 *TDecisionGraph*에서는 decision source(*TDecisionSource*)의 필드를 decision pivot(*TDecisionPivot*)과 함께 데이터 차원이 열리고 닫히며 드래그 앤 드롭되고 재정렬될 때마다 변경되는 동적 그래프로 표시할 수 있습니다. 선 그래프의 타입, 색 및 표시 타입과 decision 그래프의 다른 많은 속성을 변경할 수 있습니다.

다음과 같은 방법으로 그래프를 사용자 정의합니다.

1. 마우스 오른쪽 버튼으로 그래프를 클릭한 다음 Edit Chart를 선택합니다. Chart Editing 다이얼로그 박스가 나타납니다.
2. Chart Editing 다이얼로그 박스의 Chart 페이지를 사용하여 표시되는 시리즈 리스트를 보고, 동일한 시리즈에 사용할 수 있는 정의가 두 개 이상일 경우 사용할 시리즈 정의를 선택하고, 템플릿이나 시리즈의 그래프 타입을 변경한 다음 전체 그래프 속성을 설정합니다.

   Chart 페이지의 Series 리스트는 Template: 다음에 모든 decision cube 차원과 현재 보이는 범주를 모두 표시합니다. 각각의 범주나 시리즈는 독립적인 객체입니다. 다음 작업을 수행할 수 있습니다.

   - 기존의 decision 그래프 시리즈에서 파생된 시리즈를 추가하거나 삭제합니다. 파생된 시리즈는 기존 시리즈에 대한 주석을 제공하거나 다른 시리즈로부터 계산된 값을 나타냅니다.
   - 디폴트 그래프 타입을 변경하고 템플릿과 시리즈의 제목을 변경합니다.

   기타 Chart 페이지 탭에 대한 자세한 내용을 보려면 온라인 도움말에서 "Chart 페이지(Chart Editing 다이얼로그 박스)"라는 주제를 검색하십시오.
3. Series 페이지를 사용하여 차원 템플릿을 구축한 다음 개별 그래프 시리즈에 대한 속성을 사용자 정의합니다.

디폴트로, 모든 시리즈는 막대 그래프로 표현되며 기본 16색이 지정됩니다. 템플릿 타입과 속성을 편집하여 새 기본값을 만들 수 있습니다. 그리고 나서 decision source를 다른 상태로 피벗하면 새로운 상태별로 시리즈를 동적으로 만들기 위해 템플릿이 사용됩니다. 템플릿에 대한 자세한 내용은 20-16페이지의 "decision 그래프 템플릿 기본값 설정"을 참조하십시오.

개별 시리즈를 사용자 정의하려면 20-17페이지의 "decision 그래프 시리즈 사용자 정의"의 지침을 따르십시오.

Series 페이지 탭에 대한 자세한 내용을 보려면 온라인 도움말에서 "Series 페이지(Chart Editing 다이얼로그 박스)"라는 주제를 검색하십시오.

decision 그래프 템플릿 기본값 설정

decision 그래프는 decision cube의 두 가지 차원, 즉 그래프의 축으로 표시되는 한 차원과 시리즈 집합을 만드는 데 사용되는 다른 차원의 값을 표시합니다. 해당 차원의 템플릿은 이러한 시리즈의 디폴트 속성(예: 막대, 선, 면 등 시리즈의 형태)을 제공합니다. 사용자가 한 상태에서 다른 상태로 피벗하면 해당 차원에 필요한 시리즈가 시리즈 타입과 템플릿에 지정된 기타 기본값을 사용하여 만들어집니다.

사용자가 한 차원만 활성 중인 상태로 피벗하는 경우 별도의 템플릿이 제공됩니다. 1차원 상태는 종종 파이 차트로 표시되므로 이 경우에 대한 별도의 템플릿이 제공됩니다.

다음 작업을 할 수 있습니다.

- 디폴트 그래프 타입을 변경합니다.
- 다른 그래프 템플릿 속성을 변경합니다.
- 전반적 그래프 속성을 보고 설정합니다.

디폴트 decision 그래프 타입 변경

다음과 같은 방법으로 디폴트 decision 그래프 타입을 변경할 수 있습니다.

1. Chart Editing 다이얼로그 박스에 있는 Chart 페이지의 Series 리스트에서 템플릿을 선택합니다.
2. Change 버튼을 클릭합니다.
3. 새 타입을 선택하고 Gallery 다이얼로그 박스를 닫습니다.

기타 decision 그래프 템플릿 속성 변경

다음과 같은 방법으로 템플릿의 색상이나 기타 속성을 변경합니다.

1. Chart Editing 다이얼로그 박스의 맨 위에서 Series 페이지를 선택합니다.
2. 페이지 맨 위에 있는 드롭다운 리스트에서 템플릿을 선택합니다.
3. 해당 속성 탭을 선택하고 설정을 선택합니다.

전반적인 decision 그래프 속성 보기

다음과 같은 방법으로 타입과 시리즈 이외의 decision 그래프 속성을 보고 설정합니다

1. Chart Editing 다이얼로그 박스의 맨 위에서 Chart 페이지를 선택합니다.
2. 해당 속성 탭을 선택하고 설정을 선택합니다.

decision 그래프 시리즈 사용자 정의

템플릿은 그래프 타입과 시리즈 표시 방법 등 각 decision cube 차원에 대한 많은 기본값을 제공합니다. 시리즈 색상 등의 다른 기본값은 *TDecisionGraph*에서 정의합니다. 원하면 각 시리즈에 대한 기본값을 오버라이드할 수 있습니다.

템플릿은 필요에 따라 범주에 대한 시리즈를 만든 다음 더 이상 필요하지 않으면 삭제하는 프로그램이 필요한 경우 사용할 용도로 만든 것입니다. 원하면 특정 범주 값에 대한 사용자 정의 시리즈를 설정할 수 있습니다. 그러려면 그래프의 현재 디스플레이에서 사용자 정의할 범주에 대한 시리즈를 갖도록 그래프를 피벗하면 됩니다. 시리즈가 그래프에 표시되면 Chart Editor를 사용하여 다음 작업을 할 수 있습니다.

- 그래프 타입을 변경합니다.
- 기타 시리즈 속성을 변경합니다.
- 사용자 정의했던 특정 그래프 시리즈를 저장합니다.

시리즈 템플릿을 정의하고 전반적 그래프 기본값을 설정하려면 20-16 페이지의 "decision 그래프 템플릿 기본값 설정"을 참조하십시오.

시리즈 그래프 타입 변경

디폴트로, 각 시리즈는 차원 템플릿에서 정의한 대로 같은 그래프 타입을 갖습니다. 모든 시리즈를 같은 그래프 타입으로 변경하려면 템플릿 타입을 변경할 수 있습니다. 자세한 내용은 20-16 페이지의 "디폴트 decision 그래프 타입 변경"을 참조하십시오.

다음과 같은 방법으로 단일 시리즈의 그래프 타입을 변경할 수 있습니다.

1. Chart Editor의 Chart 페이지에 있는 Series 리스트에서 시리즈를 선택합니다.
2. Change 버튼을 클릭합니다.
3. 새 타입을 선택하고 Gallery 다이얼로그 박스를 닫습니다.
4. Save Series 체크 박스를 선택합니다.

기타 decision 그래프 시리즈 속성 변경

다음과 같은 방법으로 decision 그래프 시리즈의 색상이나 기타 속성을 변경할 수 있습니다.

1. Chart Editing 다이얼로그 박스의 맨 위에서 Series 페이지를 선택합니다.
2. 해당 페이지의 맨 위에 있는 드롭다운 리스트에서 시리즈를 선택합니다.
3. 해당 속성 탭을 선택하고 설정을 선택합니다.
4. Save Series 체크 박스를 선택합니다.

decision 그래프 시리즈 설정 저장

디폴트로, 템플릿의 설정만 디자인 타임 시 저장됩니다. 특정 시리즈에 대한 변경 내용은 Chart Editing 다이얼로그 박스에서 해당 시리즈에 대해 Save 박스가 선택되어 있을 경우에만 저장됩니다.

시리즈 저장에는 메모리가 많이 사용되므로 시리즈를 저장할 필요가 없으면 Save 체크 박스를 선택 해제할 수 있습니다.

런타임 시 decision 지원 컴포넌트

런타임 시 마우스 왼쪽 버튼을 클릭하거나 오른쪽 버튼을 클릭한 다음 보이는 decision 지원 컴포넌트를 끌어서 많은 작업을 수행할 수 있습니다. 이 장의 앞 부문에서 설명한 이러한 작업이 아래에 요약되어 있습니다.

런타임 시 decision pivot

다음 작업을 할 수 있습니다.

- decision pivot의 왼쪽 끝에 있는 요약 버튼을 마우스 왼쪽 버튼으로 클릭하면 사용 가능한 요약 리스트가 표시됩니다. 이러한 리스트를 사용하여 decision grid 와 decision 그래프에 표시된 요약 데이터를 변경할 수 있습니다.
- 마우스 오른쪽 버튼으로 차원 버튼을 클릭하면 다음을 선택할 수 있습니다.
  - 행 영역에서 열 영역으로 차원 버튼을 이동하거나 그 반대 작업이 가능
  - Drill In을 선택하여 디테일 데이터 표시
- Drill In 명령 다음에 마우스 왼쪽 버튼으로 차원을 클릭하면 다음을 선택할 수 있습니다.
  - Open Dimension을 선택하여 해당 차원의 맨 위 레벨로 다시 이동
  - All Values를 선택하여 decision grid에서 요약만 표시와 요약을 비롯한 기타 모든 값 표시 사이에서 토글
  - 해당 차원에 대해 사용 가능한 범주 리스트에서 디테일 값으로 드릴인(drill-in)하기 위한 한 범주
- 마우스 왼쪽 버튼으로 차원 버튼을 클릭하면 해당 차원을 열거나 닫습니다.
- 차원 버튼을 행 영역에서 열 영역으로 끌어다 놓거나 그 반대 작업을 할 수 있습니다. 해당 영역의 기존 버튼 옆에 놓거나 행이나 열 아이콘에 놓을 수 있습니다.

런타임 시 decision grid

다음 작업을 할 수 있습니다.

- decision grid 내에서 마우스 오른쪽 버튼으로 클릭하면 다음 작업을 선택할 수 있습니다.
  - 개별 데이터 그룹이나 차원의 모든 값 또는 전체 그리드에 대해 부분합 기능을 켜고 끕니다.
  - 20-7페이지에서 설명한 Decision Cube Editor를 표시합니다.
  - 차원과 요약을 열고 닫습니다.
- 행과 열 내에서 +와 -를 클릭하여 차원을 열고 닫습니다.
- 행에서 열로 또는 열에서 행으로 차원을 끌어다 놓습니다.

런타임 시 decision 그래프

사용자는 그래프 그리드 영역에서 한 쪽에서 다른 쪽으로나 위, 아래로 끌어서 화면 밖 범주와 값을 스크롤할 수 있습니다.

decision 지원 컴포넌트와 메모리 제어

차원이나 요약이 decision cube에 로드될 때 메모리를 차지합니다. 새 요약을 추가하면 메모리 소비가 선형으로 증가합니다. 즉, 두 개의 요약이 있는 decision cube는 한 개의 요약이 있는 동일한 decision cube 보다 두 배의 메모리를 차지하며 세 개의 요약이 있는 decision cube는 한 개의 요약이 있는 동일한 decision cube 보다 세 배의 메모리를 차지하는 식입니다. 차원의 경우 메모리 소비는 더욱 빠릅니다. 10개의 값이 있는 차원을 추가하면 10배만큼 메모리 소비가 증가합니다. 100개의 값이 있는 차원을 추가하면 메모리 소비가 100배 증가합니다. 따라서 decision cube에 차원을 추가하면 메모리 사용에 큰 영향을 미칠 뿐 아니라 성능 문제가 쉽게 발생할 수 있습니다. 많은 값이 있는 차원을 추가할 때 특히 이러한 부정적 효과가 발생합니다.

decision 지원 컴포넌트에는 메모리가 사용되는 방법 및 시기를 제어하는 데 도움이 되는 여러 가지 설정이 있습니다. 여기에서 언급한 속성 및 기술에 대한 자세한 내용은 온라인 도움말에서 *TDecisionCube*를 찾아보십시오.

최대 차원, 요약 및 셀 설정

decision cube의 *MaxDimensions*과 *MaxSummaries* 속성을 *CubeDim->ActiveFlag* 속성과 함께 사용하면 한 번에 로드할 수 있는 차원 및 요약 수를 제어할 수 있습니다. Decision Cube Editor의 Cube Capacity 페이지에서 최대 값을 설정하여 동시에 메모리로 가져올 수있는 차원이나 요약 수를 전반적으로 제어할 수 있습니다.

차원이나 요약 수를 제한하면 decision cube에서 사용하는 메모리 양을 대략적으로 제한할 수 있습니다. 하지만 값이 많이 있는 차원과 값이 별로 들어있지 않은 차원은 구분하지 못합니다. decision cube에 대한 절대 메모리 수요를 더 많이 제어하기 위해 decision cube의 셀 수를 제한할 수도 있습니다. Decision Cube Editor의 Cube Capacity 페이지에서 최대 셀 수를 설정합니다.

차원 상태 설정

ActiveFlag 속성은 로드될 차원을 제어합니다. Decision Cube Editor의 Dimension Settings 탭에서 Activity Type 컨트롤을 사용하여 이 속성을 설정할 수 있습니다. 이 컨트롤이 *Active*로 설정되어 있으면 무조건 차원이 로드되어 항상 공간을 차지하게 됩니다. 이 상태의 차원 수는 항상 *MaxDimensions* 보다 적어야 하며 *Active* 로 설정된 요약 수는 *MaxSummaries* 보다 적어야 합니다. 차원이나 요약을 항상 사용해야 할 경우에만 *Active*로 설정해야 합니다. *Active* 설정은 사용 가능한 메모리를 관리하는 큐브의 기능을 감소시킵니다.

*ActiveFlag*가 *AsNeeded*로 설정되어 있으면 *MaxDimensions*나 *MaxSummaries*, *MaxCells* 제한을 초과하지 않고 로드될 수 있는 경우에만 차원이나 요약이 로드됩니다. decision cube는 *MaxCells*나 *MaxDimensions*, *MaxSummaries*에서 부과한 제한 내에서 *AsNeeded*로 표시된 차원과 요약을 바꿔가며 메모리에 넣었다가 뺍니다. 따라서 차원이나 요약이 현재 사용되고 있지 않으면 메모리에 로드되지 않습니다. 자주 사용되지 않는 차원을 *AsNeeded*로 설정하면 현재 로드되어 있지 않은 차원을 액세스할 때 시간이 지체되는 단점이 있지만 로드 및 피벗 성능은 향상됩니다.

페이징된 차원 사용

Decision Cube Editor의 Dimension Settings 탭에서 Binning이 Set으로 설정되어 있고 Start Value가 NULL이 아니면 차원이 "페이징"되었거나 "영구적으로 드릴다운"되었다고 합니다. 프로그래밍 방식으로 일련의 값을 순차적으로 액세스할 수는 있지만 한 번에 해당 차원의 단일 값에 대한 데이터를 액세스할 수 있습니다. 그러한 차원은 피벗되거나 열리지 않을 수 있습니다.

값이 상당히 많이 있는 차원에 대한 차원 데이터를 포함하면 상당히 메모리가 많이 소모됩니다. 이러한 차원을 페이징하면 한 번에 한 값에 대한 요약 정보를 표시할 수 있습니다. 이런 식으로 표시하면 정보가 읽기 쉬워지며 메모리 소비도 훨씬 더 관리하기 쉽습니다.

장

21

데이터베이스에 연결

모든 데이터셋 컴포넌트는 데이터베이스 서버에 직접 연결할 수 있습니다. 일단 연결되면 데이터셋은 서버와 자동으로 통신합니다. 데이터셋을 열면 데이터셋이 서버의 데이터로 채워지고 레코드를 포스트하면 레코드는 서버로 다시 보내진 다음 적용됩니다. 단일 연결 컴포넌트를 여러 데이터셋에서 공유할 수 있거나 각 데이터셋에서 자체의 고유한 연결을 사용할 수 있습니다.

각 데이터셋 타입은 고유한 연결 컴포넌트 타입을 사용하여 데이터베이스 서버에 연결되며 단일 데이터 액세스 메커니즘과 함께 작동하도록 디자인됩니다. 다음 표에는 이러한 데이터 액세스 메커니즘 및 관련된 연결 컴포넌트가 열거되어 있습니다.

표 21.1 데이터베이스 연결 컴포넌트

데이터 액세스 메커니즘	연결 컴포넌트
Borland Database Engine(BDE)	TDatabase
ActiveX Data Objects(ADO)	TADOConnection
dbExpress	TSQLConnection
InterBase Express	TIBDatabase

참고 이러한 메커니즘의 장, 단점에 대한 자세한 내용은 18-1페이지의 "데이터베이스 사용"을 참조하십시오.

연결 컴포넌트는 데이터베이스를 연결하는 데 필요한 모든 정보를 제공합니다. 이러한 정보는 다음과 같이 연결 컴포넌트 타입에 따라 다릅니다.

- BDE 기반 연결에 대한 자세한 내용은 24-13페이지의 "데이터베이스 식별"을 참조하십시오.
- ADO 기반 연결에 대한 자세한 내용은 25-3페이지의 "TADOConnection을 사용하여 데이터 저장소에 연결"을 참조하십시오.
- dbExpress 연결에 대한 자세한 내용은 26-3페이지의 "TSQLConnection 설정"을 참조하십시오.
- InterBase Express 연결에 대한 자세한 내용은 *TIBDatabase*의 온라인 도움말을 참조하십시오.

각 데이터셋 타입은 다른 연결 컴포넌트를 사용하지만 모두 *TCustomConnection*의 자손이므로 다수의 동일한 작업을 수행하고 많은 동일한 속성, 메소드 및 이벤트를 사용합니다. 이 장에서는 이러한 많은 공통 작업에 대해 설명합니다.

암시적 연결 사용

사용 중인 데이터 액세스 메커니즘에 상관없이 명시적으로 연결 컴포넌트를 만들어 데이터베이스와의 연결 및 통신 관리에 사용할 수 있습니다. BDE 호환 및 ADO 기반 데이터셋의 경우 데이터셋 속성을 통해 데이터베이스 연결을 설명하고 데이터셋에서 암시적으로 연결하게 하는 옵션도 있습니다. BDE 호환 데이터셋의 경우 *DatabaseName* 속성을 사용하여 암시적 연결을 지정합니다. ADO 기반 데이터셋의 경우 *ConnectionString* 속성을 지정합니다.

암시적 연결을 사용하는 경우 연결 컴포넌트를 명시적으로 만들 필요가 없습니다. 따라서 애플리케이션 개발이 간단해지고 개발자가 지정한 디폴트 연결에서 다양한 상황을 처리할 수 있습니다. 하지만 사용자가 많고 데이터베이스 연결 요구 사항이 다양한, 복잡하고 중요한 임무를 갖는 클라이언트/서버 애플리케이션의 경우 사용자 고유의 연결 컴포넌트를 만들어 애플리케이션 요구에 맞춰 각 데이터베이스 연결을 조정해야 합니다. 명시적 연결 컴포넌트를 사용하면 컨트롤을 더 잘할 수 있습니다. 예를 들어, 다음 작업을 수행하려면 연결 컴포넌트에 액세스해야 합니다.

- 데이터베이스 서버 로그인 지원을 사용자 정의(암시적 연결에서는 사용자에게 사용자 이름과 암호를 입력하라는 디폴트 로그인 다이얼로그 박스 표시)
- 트랜잭션을 제어하고 트랜잭션 분리 레벨을 지정
- 데이터셋을 사용하지 않고 서버에서 SQL 명령을 실행
- 같은 데이터베이스에 연결된 열려 있는 모든 데이터셋에서 작업을 수행

그리고 모두 같은 서버를 사용하는 여러 데이터셋이 있는 경우 연결 컴포넌트를 사용하면 한 곳에서만 사용할 서버를 지정하면 되기 때문에 사용하기가 더 쉽습니다. 마찬가지로 나중에 서버를 변경하는 경우에도 여러 데이터셋 컴포넌트를 업데이트하지 않고 연결 컴포넌트만 업데이트하면 됩니다.

연결 제어

데이터베이스 서버에 연결하려면 원하는 서버를 설명하는 특정 정보를 애플리케이션에서 제공해야 합니다. 연결 컴포넌트의 타입에 따라 서버를 정의하는 속성 집합이 다릅니다. 하지만 일반적으로 연결 컴포넌트는 원하는 서버의 이름을 지정하고 연결 구성 방법을 제어하는 연결 매개변수를 입력하는 방식을 각각 제공합니다. 연결 매개변수는 서버에 따라 다르며 사용자 이름과 암호, BLOB 필드의 최대 크기, SQL 역할 등의 정보를 포함할 수 있습니다.

원하는 서버와 연결 매개변수를 정의한 다음에는 연결 컴포넌트를 사용하여 명시적으로 연결을 열거나 닫을 수 있습니다. 데이터베이스 연결의 변경 내용에 따라 애플리케이션의 응답을 사용자 정의하는 데 사용할 수 있도록 연결을 열거나 닫을 때 연결 컴포넌트에서 이벤트를 생성합니다.

데이터베이스 서버에 연결

연결 컴포넌트를 사용하여 데이터베이스 서버에 연결할 수 있는 방법은 다음 두 가지입니다.

- *Open* 메소드를 호출합니다.
- *Connected* 속성을 **true**로 설정합니다.

Open 메소드를 호출하면 *Connected*를 **true**로 설정합니다.

참고 연결 컴포넌트가 서버에 연결되어 있지 않고 애플리케이션에서 관련 데이터셋 중 하나를 열려고 하면 데이터셋은 자동으로 연결 컴포넌트의 *Open* 메소드를 호출합니다.

*Connected*를 **true**로 설정하면 연결 컴포넌트는 먼저 *BeforeConnect* 이벤트를 생성하므로 초기화를 수행할 수 있습니다. 예를 들어, 이 이벤트를 사용하여 연결 매개변수를 대체할 수 있습니다.

BeforeConnect 이벤트 다음에는 서버 로그인 제어를 선택한 방법에 따라 연결 컴포넌트에서 디폴트 로그인 다이얼로그 박스를 표시합니다. 그리고 나서 사용자 이름과 암호를 드라이버에 전달하여 연결을 엽니다.

일단 연결이 열리면 연결 컴포넌트는 *AfterConnect* 이벤트를 생성하므로 열린 연결이 필요한 작업을 수행할 수 있습니다.

참고 일부 연결 컴포넌트는 연결할 때 추가 이벤트도 생성합니다.



연결된 다음에는 연결을 사용하는 활성 데이터셋이 하나 이상이면 연결이 계속 유지됩니다. 이상 활성 데이터셋이 없으면 연결 컴포넌트에서 연결을 끊습니다. 일부 연결 컴포넌트는 연결을 사용하는 모든 데이터셋이 닫혀 있더라도 연결이 열린 상태로 있게 하는 *KeepConnection* 속성을 사용합니다. *KeepConnection*이 **true**이면 연결이 유지됩니다. 원격 데이터베이스 서버에 대한 연결의 경우나 자주 데이터셋을 열고 닫는 애플리케이션의 경우 *KeepConnection*을 **true**로 설정하면 네트워크 트래픽이 줄고 애플리케이션의 속도가 빨라집니다. *KeepConnection*이 **false**인 경우 데이터베이스를 사용하는 활성 데이터셋이 없으면 연결이 끊어집니다. 데이터베이스를 사용하는 데이터셋을 나중에 열면 다시 연결이 되고 초기화됩니다.

데이터베이스 서버로부터 연결 끊기

연결 컴포넌트를 사용하여 서버의 연결을 끊을 수 있는 방법은 다음 두 가지입니다.

- *Connected* 속성을 **false**로 설정합니다.
- *Close* 메소드를 호출합니다.

*Close*를 호출하면 *Connected*가 **false**로 설정됩니다.

*Connected*가 **false**로 설정되어 있으면 연결 컴포넌트는 *BeforeDisconnect* 이벤트를 생성하므로 연결이 닫히기 전에 정리 작업을 수행할 수 있습니다. 예를 들어, 이 이벤트를 사용하여 연결이 닫히기 전에 모든 열려 있는 데이터셋에 대한 정보를 캐싱할 수 있습니다.

BeforeConnect 이벤트 다음에는 연결 컴포넌트에서 열려 있는 모든 데이터셋을 닫고 서버와의 연결을 끊습니다.

마지막으로 연결 컴포넌트는 *AfterDisconnect* 이벤트를 생성하므로 사용자 인터페이스에서 Connect 버튼을 사용 가능하게 하는 등 연결 상태의 변경 내용에 따라 응답할 수 있습니다.

참고 *Close*를 호출하거나 *Connected*를 **false**로 설정하면 연결 컴포넌트의 *KeepConnection* 속성이 **true**인 경우라도 데이터베이스 서버와의 연결이 끊어집니다.

서버 로그인 제어

대부분의 원격 데이터베이스 서버에는 승인되지 않은 액세스를 막는 보안 기능이 포함되어 있습니다. 일반적으로 서버에서 데이터베이스 액세스를 허용하기 전에 사용자 이름과 암호 로그인을 요구합니다.

디자인 타임 시 서버에서 로그인을 요구하면 사용자가 데이터베이스에 처음으로 연결을 시도할 때 사용자 이름과 암호를 입력하라는 표준 로그인 다이얼로그 박스가 표시됩니다.

런타임 시 서버의 로그인 요청을 처리할 수 있는 세 가지 방법은 다음과 같습니다.

- 디폴트 로그인 다이얼로그 박스 및 프로세스에서 로그인을 처리하게 합니다. 이것이 기본 방법입니다. 연결 컴포넌트의 *LoginPrompt* 속성을 **true**(기본값)로 설정하고 연결 컴포넌트를 선언한 유닛에 DBLogDlg.hpp를 포함합니다. 서버에서 사용자 이름과 암호를 요청할 때 사용자 애플리케이션에서 표준 로그인 다이얼로그 박스를 표시합니다.
- 로그인을 시도하기 전에 로그인 정보를 입력합니다. 연결 컴포넌트 타입에 따라 다음과 같이 사용자 이름과 암호를 지정할 때 다양한 메커니즘을 사용합니다.
  - BDE, dbExpress 및 InterBase express 데이터셋의 경우 사용자 이름 및 암호 연결 매개변수는 *Params* 속성을 통해 액세스할 수 있습니다. dbExpress 데이터셋의 경우 매개변수 값을 연결 이름과도 연결할 수 있는 반면, BDE 데이터셋의 경우 매개변수 값을 BDE 알리아스와 연결할 수 있습니다.
  - ADO 데이터셋의 경우 사용자 이름과 암호는 *ConnectionString* 속성에 포함될 수 있거나 *Open* 메소드의 매개변수로 제공될 수 있습니다.

  서버에서 사용자 이름과 암호를 요청하기 전에 지정하려면 *LoginPrompt*를 **false**로 설정하여 디폴트 로그인 다이얼로그 박스가 표시되지 않도록 해야 합니다. 예를 들어, 다음 코드는 *BeforeConnect* 이벤트 핸들러의 SQL 연결 컴포넌트에서 사용자 이름과 암호를 설정하고 현재 연결 이름과 연결되어 있는 암호를 해독합니다.

```
void __fastcall TForm1::SQLConnectionBeforeConnect(TObject *Sender)
{
  if (SQLConnection1->LoginPrompt == false)
  {
    SQLConnection1->Params->Values["User_Name"] = "SYSDBA";
    SQLConnection1->Params->Values["Password"] =
        Decrypt(SQLConnection1->Params->Values["Password"]);
  }
}
```

디자인 시 사용자 이름과 암호를 설정하거나 코드의 하드 코딩된 문자열을 사용하면 해당 값이 애플리케이션 실행 파일에 포함되므로 값을 찾기 쉽고 서버 보안이 손상됩니다.

- 로그인 이벤트에 대해 고유한 사용자 정의 처리 방법을 제공하십시오. 연결 컴포넌트는 사용자 이름과 암호가 필요할 때 이벤트를 생성합니다.
  - *TDatabase*, *TSQLConnection* 및 *TIBDatabase*의 경우 로그인 이벤트는 *OnLogin* 이벤트입니다. 이벤트 핸들러에는 두 개의 매개변수, 즉 연결 컴포넌트와 문자열 리스트에 있는 사용자 이름과 암호 매개변수의 로컬 복사본이 있습니다. *TSQLConnection*에는 데이터베이스 매개변수도 포함됩니다. 이 이벤트가 발생하게 하려면 *LoginPrompt* 속성을 **true**로 설정해야 합니다. *LoginPrompt* 값을 **false**로 하고 *OnLogin* 이벤트 핸들러를 지정하면 디폴트 다이얼로그 박스가 표시되지 않고 *OnLogin* 이벤트 핸들러가 실행되지 않으므로 데이터베이스에 로그인할 수 있는 상황이 만들어집니다.
  - *TADOConnection*의 경우 로그인 이벤트는 *OnWillConnect* 이벤트입니다. 이벤트 핸들러에는 5개의 매개변수, 즉 연결에 영향을 주기 위해 값을 반환하는 연결 컴포넌트와 네 개의 매개변수가 있습니다. 이 중에 두 개의 매개변수는 사용자 이름과 암호에 사용됩니다. 이러한 이벤트는 *LoginPrompt* 값에 관계 없이 항상 발생합니다.

로그인 매개변수를 설정한 이벤트에 대해 이벤트 핸들러를 작성합니다. 전역 변수(*UserName*)와 사용자 이름이 주어진 암호(*PasswordSearch*)를 반환하는 메소드로부터 USER NAME과 PASSWORD 매개변수 값이 입력되는 예가 다음에 나와 있습니다.

```
void __fastcall TForm1::Database1Login(TDatabase *Database, TStrings
*LoginParams)
{
  LoginParams->Values["USER NAME"] = UserName;
  LoginParams->Values["PASSWORD"] = PasswordSearch(UserName);
}
```

*OnLogin*이나 *OnWillConnect* 이벤트 핸들러를 작성할 때 로그인 매개변수를 입력하는 다른 메소드의 경우처럼 애플리케이션 코드에 암호를 하드 코딩하지 마십시오. 암호는 암호화된 값, 즉 사용자 애플리케이션에서 해당 값을 조사하기 위해 사용하는 보안 데이터베이스의 엔트리로만 표시되거나 사용자로부터 동적으로 입력을 받아야 합니다.

트랜잭션 관리

*트랜잭션*은 데이터베이스에 있는 하나 이상의 테이블들이 *커밋*(영구적으로 적용)되기 전에 성공적으로 모두 수행되어야 하는 작업의 그룹입니다. 그룹 내 작업이 하나라도 실패할 경우 모든 작업이 *롤백*(취소)됩니다. 트랜잭션을 사용하면 트랜잭션을 구성하는 작업 중 하나를 완료하면서 문제가 발생하더라도 데이터베이스가 일관성을 잃지 않도록 보장할 수 있습니다.

예를 들어, 은행 애플리케이션의 경우 한 계정에서 다른 계정으로 자금을 이체하는 작업은 트랜잭션으로 보호해야 할 필요가 있습니다. 한 계정의 잔고를 감소시킨 다음 다른 계정의 잔고를 증가시키려고 할 때 오류가 발생한 경우 데이터베이스에서 계속 올바른 수지를 유지하게 하려면 트랜잭션을 롤백해야 할 것입니다.

SQL 명령을 직접 데이터베이스로 전송하여 항상 트랜잭션을 관리할 수 있습니다. 일부 데이터베이스는 전혀 트랜잭션을 지원하지 않지만 대부분의 데이터베이스는 고유한 트랜잭션 관리 모델을 제공합니다. 트랜잭션을 지원하는 서버의 경우 스키마 캐싱과 같은 특정 데이터베이스 서버의 고급 트랜잭션 관리 기능을 이용하여 사용자 고유의 트랜잭션 관리를 직접 코딩해야 하는 경우도 있습니다.

고급 트랜잭션 관리 기능을 사용할 필요가 없는 경우 명시적으로 SQL 명령을 전송하지 않아도 트랜잭션을 관리할 때 사용할 수 있는 일련의 메소드와 속성을 연결 컴포넌트에서 제공합니다. 이러한 속성과 메소드를 사용하면 서버에서 트랜잭션을 지원하는 한, 데이터베이스 서버 타입별로 애플리케이션을 사용자 정의하지 않아도 되는 이점이 있습니다. BDE의 경우 서버 트랜잭션이 지원되지 않는 로컬 테이블에 대해서도 제한적으로 트랜잭션을 지원합니다. BDE를 사용하지 않는 경우, 트랜잭션을 지원하지 않는 데이터베이스에서 트랜잭션을 시작하려고 하면 연결 컴포넌트에서 예외가 발생하게 됩니다.

경고 데이터셋 프로바이더 컴포넌트에서 업데이트를 적용할 때 업데이트에 대해 트랜잭션이 암시적으로 생성됩니다. 따라서 사용자가 명시적으로 시작한 트랜잭션과 프로바이더에서 생성한 트랜잭션이 충돌하지 않도록 주의해야 합니다.

트랜잭션 시작

트랜잭션을 시작할 때 데이터베이스를 읽거나 데이터베이스에 기록하는 그 다음 문들은 트랜잭션이 명시적으로 중단되거나(오버랩된 트랜잭션의 경우) 다른 트랜잭션이 시작되기 전에는 모두 트랜잭션의 컨텍스트 내에서 발생합니다. 각 문은 그룹의 일부로 간주됩니다. 변경 내용을 데이터베이스로 성공적으로 커밋하거나 그룹의 모든 변경 내용을 취소해야 합니다.

트랜잭션을 처리 중일때 데이터베이스 테이블의 사용자 데이터 뷰는 사용자의 트랜잭션 분리 레벨에 따라 결정됩니다. 트랜잭션 분리 레벨에 대한 자세한 내용은 21-9페이지의 "트랜잭션 분리 레벨 지정"을 참조하십시오.

*TADOConnection*의 경우 다음과 같이 *BeginTrans* 메소드를 호출하여 트랜잭션을 시작합니다.

```
Level = ADOConnection1->BeginTrans();
```

*BeginTrans*는 시작된 트랜잭션의 중첩 레벨을 반환합니다. 중첩된 트랜잭션은 다른 부모 트랜잭션 내에서 중첩된 트랜잭션입니다. 서버에서 트랜잭션을 시작한 다음 ADO 연결에서는 *OnBeginTransComplete* 이벤트를 받습니다.

*TDatabase*의 경우는 대신에 *StartTransaction* 메소드를 사용합니다. *TDataBase*에서는 중첩 트랜잭션이나 오버랩 트랜잭션을 지원하지 않습니다. 다른 트랜잭션이 진행 중일 때 *TDatabase* 컴포넌트의 *StartTransaction* 메소드를 호출하면 예외가 발생합니다. *StartTransaction*을 호출하는 것을 막으려면 다음과 같이 *InTransaction* 속성을 검사할 수 있습니다.

```
if (!Database1->InTransaction)
  Database1->StartTransaction();
```

*TSQLConnection*에서도 *StartTransaction* 메소드를 사용하지만 사용자가 더 많이 제어할 수 있는 버전을 사용합니다. 특히, *StartTransaction*에서는 트랜잭션 정보를 취하므로 동시에 여러 트랜잭션을 관리하고 트랜잭션별로 트랜잭션 분리 레벨을 지정할 수 있습니다. 트랜잭션 레벨에 대한 자세한 내용은 21-9페이지의 "트랜잭션 분리 레벨 지정"을 참조하십시오. 동시에 여러 트랜잭션을 관리하려면 트랜잭션 설명의 *TransactionID* 필드를 고유한 값으로 설정합니다.
*TransactionID*가 현재 진행 중인 다른 트랜잭션과 충돌하지 않고 고유하다면 어느 값이나 가능합니다. 서버에 따라 *TSQLConnection*에서 시작한 트랜잭션은 중첩(ADO를 사용하는 경우)되거나 오버랩될 수 있습니다.

```
TTransactionDesc TD;
TD.TransactionID = 1;
TD.IsolationLevel = xilREADCOMMITTED;
SQLConnection1->StartTransaction(TD);
```

디폴트로 , 오버랩 트랜잭션을 사용하면 두 번째 트랜잭션이 시작할 때 첫 번째 트랜잭션은 비활성화됩니다. 하지만 첫 번째 트랜잭션의 커밋이나 롤백을 나중으로 연기할 수 있습니다. InterBase 데이터베이스와 함께 *TSQLConnection*을 사용하면 *TransactionLevel* 속성을 설정하여 사용자 애플리케이션의 각 데이터셋을 특정 활성 트랜잭션과 동일시할 수 있습니다. 즉, 두 번째 트랜잭션을 시작한 다음 원하는 트랜잭션과 함께 데이터셋을 연결하기만 하면 양쪽 트랜잭션을 동시에 계속 작업할 수 있습니다.

참고 *TADOConnection*과는 달리 *TSQLConnection*과 *TDatabase*는 트랜잭션이 시작할 때 이벤트를 받지 않습니다.

InterBase Express는 연결 컴포넌트를 사용하여 트랜잭션을 시작하지 않고 각각의 트랜잭션 컴포넌트를 사용하므로 *TSQLConnection*보다 훨씬 큰 제어력을 제공합니다. 하지만 *TIBDatabase*를 사용하면 다음과 같이 디폴트 트랜잭션을 시작할 수 있습니다.

```
if (!IBDatabase1->DefaultTransaction->InTransaction)
  IBDatabase1->DefaultTransaction->StartTransaction();
```

두 개의 독립적인 트랜잭션 컴포넌트를 사용하여 오버랩된 트랜잭션을 가질 수 있습니다. 각 트랜잭션 컴포넌트에는 트랜잭션을 구성할 수 있게 해주는 일련의 매개변수가 있습니다. 이러한 매개변수를 사용하면 트랜잭션 분리 레벨 및 다른 트랜잭션 속성을 지정할 수 있습니다.

트랜잭션 끝내기

이상적으로, 트랜잭션은 필요한 동안만 지속되어야 합니다. 트랜잭션이 오랫동안 활성화되면 데이터베이스를 액세스하는 동시 사용자 수가 더 많아지고, 사용자 트랜잭션의 수명 동안 시작하고 끝나는 동시 발생 트랜잭션 수가 더 많아지므로 사용자가 변경 내용을 커밋하려고 할 때 트랜잭션이 다른 트랜잭션과 충돌하게 됩니다.

성공적인 트랜잭션 끝내기

트랜잭션을 구성하는 동작이 모두 성공하면 트랜잭션을 커밋하여 데이터베이스 변경 내용을 영구적으로 만들 수 있습니다. *TDatabase*의 경우 *Commit* 메소드를 사용하여 트랜잭션을 커밋합니다.

```
MyOracleConnection->Commit();
```

*TSQLConnection*의 경우에도 *Commit* 메소드를 사용하지만 다음과 같이 *StartTransaction* 메소드에 제공한 트랜잭션 설명을 입력하여 커밋할 트랜잭션을 지정해야 합니다.

```
MyOracleConnection->Commit(TD);
```

*TIBDatabase*의 경우 트랜잭션 객체의 *Commit* 메소드를 사용하여 트랜잭션 객체를 커밋합니다.

```
IBDatabase1->DefaultTransaction->Commit();
```

*TADOConnection*의 경우 다음과 같이 *CommitTrans* 메소드를 사용하여 트랜잭션을 커밋합니다.

```
ADOConnection1->CommitTrans();
```

참고 부모 트랜잭션이 롤백될 때 변경 내용을 나중에 롤백하기만 하면 되므로 중첩된 트랜잭션도 커밋할 수 있습니다.

트랜잭션을 성공적으로 커밋한 다음에는 ADO 연결 컴포넌트에서 *OnCommitTransComplete* 이벤트를 받습니다. 다른 연결 컴포넌트는 이와 유사한 이벤트를 받지 않습니다.

현재 트랜잭션을 커밋하는 호출은 대개 **try...catch** 문에서 시도됩니다. 그 때 트랜잭션을 성공적으로 커밋할 수 없으면 **catch** 블록을 사용하여 오류를 처리한 다음 해당 작업을 다시 시작하거나 트랜잭션을 롤백할 수 있습니다.

실패한 트랜잭션 끝내기

트랜잭션의 일부인 변경을 하거나 트랜잭션을 커밋하려고할 때 오류가 발생하면 트랜잭션을 구성하는 모든 변경 내용을 버려야 할 것입니다. 이렇게 변경 내용을 버리는 것을 트랜잭션을 롤백한다고 합니다.

*TDatabase*의 경우 다음과 같이 *Rollback* 메소드를 호출하여 트랜잭션을 롤백합니다.

```
MyOracleConnection->Rollback();
```

*TSQLConnection*의 경우에도 *Rollback* 메소드를 사용하지만 다음과 같이 *StartTransaction* 메소드에 제공한 트랜잭션 정보를 입력하여 롤백하려는 트랜잭션을 지정해야 합니다.

```
MyOracleConnection->Rollback(TD);
```

*TIBDatabase*의 경우 다음과 같이 *Rollback* 메소드를 호출하여 트랜잭션 객체를 롤백합니다.

```
IBDatabase1->DefaultTransaction->Rollback();
```

*TADOConnection*의 경우 다음과 같이 *RollbackTrans* 메소드를 호출하여 트랜잭션을 롤백합니다.

```
ADOConnection1->RollbackTrans();
```

트랜잭션이 성공적으로 롤백되면 ADO 연결 컴포넌트는 *OnRollbackTransComplete* 이벤트를 받습니다. 다른 연결 컴포넌트는 유사한 이벤트를 받지 않습니다.

현재 트랜잭션을 롤백하는 호출은 대개 다음과 같은 경우에 발생합니다.

- 데이터베이스 오류에서 회복할 수 없는 예외 처리 코드
- 사용자가 Cancel 버튼을 클릭한 경우의 버튼이나 메뉴 이벤트 코드

트랜잭션 분리 레벨 지정

트랜잭션 분리 레벨은 여러 트랜잭션에서 같은 테이블을 작업할 때 한 트랜잭션이 동시에 발생한 다른 트랜잭션과 상호 작용하는 방법을 결정합니다. 특히 한 트랜잭션에서 변경한 테이블 내용을 다른 트랜잭션에서 볼 수 있는 정도에 영향을 미칩니다.

서버 타입별로 가능한 트랜잭션 분리 레벨이 다릅니다. 가능한 트랜잭션 분리 레벨은 다음 세 가지입니다.

- *DirtyRead*: 분리 레벨이 *DirtyRead*이면 다른 트랜잭션이 커밋되지 않았더라도 사용자 트랜잭션에서 다른 트랜잭션이 변경한 내용을 모두 봅니다. 커밋되지 않은 변경 내용은 영구적이지 않으므로 언제든지 롤백될 수 있습니다. 이 값은 최소 분리를 제공하므로 Oracle, Sybase, MS-SQL, InterBase 등 많은 데이터베이스 서버에서는 사용할 수 없습니다.
- *ReadCommitted*: 분리 레벨이 *ReadCommitted*이면 다른 트랜잭션에서 커밋한 변경 내용만 볼 수 있습니다. 이러한 설정은 롤백 가능성이 있는 커밋되지 않은 변경 내용은 사용자 트랜잭션에서 볼 수 없지만, 읽고 있는 중에 다른 트랜잭션이 커밋되는 경우 여전히 데이터베이스 상태에 대해 일치하지 않는 뷰를 받을 수 있습니다. 이러한 레벨은 BDE에서 관리하는 로컬 트랜잭션만 제외하고 모든 트랜잭션에서 사용할 수 있습니다.
- *RepeatableRead*: 분리 레벨이 *RepeatableRead*이면 사용자 트랜잭션에서는 데이터베이스 데이터의 일치하는 상태를 볼 수 있습니다. 사용자 트랜잭션에서는 데이터의 단일 스냅샷을 봅니다. 동시에 발생한 다른 트랜잭션에서 곧바로 변경 내용을 커밋했더라도 이 내용을 볼 수 없습니다. 이 분리 레벨은 일단 트랜잭션에서 레코드를 읽은 다음에는 해당 레코드의 뷰가 변경되지 않게 합니다. 이 레벨에서는 사용자 트랜잭션이 다른 트랜잭션에 의한 변경 내용과는 대부분 분리됩니다. 이러한 레벨은 Sybase와 MS-SQL과 같은 일부 서버에서는 사용할 수 없으며 BDE에서 관리하는 로컬 트랜잭션에서도 사용할 수 없습니다.

그리고 *TSQLConnection*을 사용하면 데이터베이스 특정 사용자 정의 분리 레벨을 지정할 수 있습니다. 사용자 정의 분리 레벨은 *dbExpress* 드라이버에서 정의합니다. 자세한 내용은 드라이버 설명서를 참조하십시오.

참고 각 분리 레벨의 구현 방법에 대한 자세한 내용은 서버 설명서를 참조하십시오.

*TDatabase*와 *TADOConnection*을 사용하면 *TransIsolation* 속성을 설정하여 트랜잭션 분리 레벨을 지정할 수 있습니다. 데이터베이스 서버에서 지원하지 않는 값으로 *TransIsolation*을 설정하면 가능한 경우 다음으로 가장 높은 분리 레벨을 얻습니다. 사용 가능한 더 높은 레벨이 없으면 트랜잭션을 시작하려고 할 때 연결 컴포넌트에서 예외를 발생시킵니다.

*TSQLConnection*을 사용할 때 트랜잭션 분리 레벨은 트랜잭션 설명의 *IsolationLevel* 필드에 의해 제어됩니다.

InterBase express를 사용할 때 트랜잭션 분리 레벨은 트랜잭션 매개변수에 의해 제어됩니다.

서버에 명령 보내기

*TIBDatabase*를 제외한 모든 데이터베이스 연결 컴포넌트를 사용하면 *Execute* 메소드를 호출하여 연결 서버에서 SQL 문을 실행할 수 있습니다. SQL 문이 SELECT 문일때 *Execute*에서 커서를 반환할 수 있지만 이러한 사용 방법은 좋지 않습니다. 데이터를 반환하는 문을 실행하기 위한 더 좋은 방법은 데이터셋을 사용하는 것입니다.

Execute 메소드는 레코드를 반환하지 않는 간단한 SQL 문을 실행할 때 아주 편리합니다. 이러한 문에는 CREATE INDEX, ALTER TABLE 및 DROP DOMAIN과 같은 데이터베이스 메타데이터에서 작동하거나 이를 만드는 데이터 정의 언어(DDL) 문이 포함됩니다. 일부 데이터 조작 언어(DML) SQL 문에서도 결과 집합을 반환하지 않습니다. 데이터에서 동작을 수행하지만 결과 집합을 반환하지 않는 DML 문은 INSERT, DELETE, UPDATE입니다.

Execute 메소드의 구문은 다음과 같이 연결 타입에 따라 다릅니다.

- *TDatabase*의 경우 *Execute*는 실행할 단일 SQL 문을 지정하는 AnsiString, 해당 명령문의 매개변수 값을 제공하는 *TParams* 객체, 다시 호출하기 위해 해당 명령문이 캐싱되어야 하는지 여부를 표시하는 부울, 돌려줄 수 있는 BDE 커서에 대한 포인터(NULL을 전달할 것을 권장) 등 네 개의 매개변수를 취합니다.
- *TADOConnection*의 경우 *Execute*의 두 가지 버전이 있습니다. 첫 번째 버전에서는 SQL 문을 지정하는 WideString과 명령문이 비동기적으로 실행될 것인지 여부와 레코드를 반환할 것인지 여부를 제어하는 일련의 옵션을 지정하는 두 번째 매개변수를 취합니다. 이러한 첫 번째 구문은 반환된 레코드의 인터페이스를 반환합니다. 두 번째 구문은 SQL 문을 지정하는 WideString, 문을 실행할 때 영향을 받는 레코드 수를 반환하는 두 번째 매개변수 그리고 명령문이 비동기적으로 실행될 것인지 여부에 관한 옵션을 지정하는 세 번째 매개변수를 취합니다. 두 가지 구문 모두 매개변수 전달은 하지 않습니다.
- *TSQLConnection* 의 경우 *Execute*는 실행할 단일 SQL 문을 지정하는 AnsiString, 해당 명령문의 매개변수 값을 제공하는 *TParams* 객체, 레코드에 NULL을 반환하도록 만들어진 *TCustomSQLDataSet*을 받을 수 있는 포인터 등 세 개의 매개변수를 취합니다.

참고 *Execute*는 한 번에 한 개의 SQL 문만 실행할 수 있습니다. SQL 스크립팅 유틸리티의 경우처럼 *Execute*를 한 번만 호출하여 여러 SQL 문을 실행하는 것은 불가능합니다. 두 개 이상의 문을 실행하려면 *Execute*를 반복적으로 호출합니다.

매개변수를 포함하지 않는 문을 실행하기는 아주 쉽습니다. 예를 들어, 다음 코드는 *TSQLConnection* 컴포넌트의 매개변수 없이 CREATE TABLE 문(DDL)을 실행합니다.

```
void __fastcall TDataForm::CreateTableButtonClick(TObject *Sender)
{
  SQLConnection1->Connected = true;
  AnsiString SQLstmt = "CREATE TABLE NewCusts " +
    "( " +
    " CustNo INTEGER, " +
    " Company CHAR(40), " +
    " State CHAR(2), " +
    " PRIMARY KEY (CustNo) " +
    ")";
  SQLConnection1->Execute(SQLstmt, NULL, NULL);
}
```

매개변수를 사용하려면 *TParams* 객체를 만들어야 합니다. 매개변수 값별로 *TParams::CreateParam* 메소드를 사용하여 *TParam* 객체를 추가합니다. 그리고 나서 *TParam* 속성을 사용하여 매개변수를 설명하고 값을 설정합니다.

이러한 과정은 *TDatabase*를 사용하여 INSERT 문을 실행하는 다음 예제에서 설명됩니다. INSERT 문에는 *StateParam*이라는 단일 매개변수가 있습니다. 해당 매개변수에 "CA" 값을 입력하기 위해 *stmtParams*라는 *TParams* 객체가 만들어집니다.

```
void __fastcall TForm1::INSERT_WithParamsButtonClick(TObject *Sender)
{
  AnsiString SQLstmt;
  TParams *stmtParams = new TParams;
  try
  {
    Database1->Connected = true;
    stmtParams->CreateParam(ftString, "StateParam", ptInput);
    stmtParams->ParamByName("StateParam")->AsString = "CA";
    SQLstmt = "INSERT INTO 'Custom.db' ";
    SQLstmt += "(CustNo, Company, State) ";
    SQLstmt += "VALUES (7777, 'Robin Dabank Consulting', :StateParam)";
    Database1->Execute(SQLstmt, stmtParams, false, NULL);
  }
  __finally
  {
    delete stmtParams;
  }
}
```

SQL 문은 매개변수는 포함하지만 그 값을 제공하기 위해 *TParam* 객체를 입력하지는 않으므로 실행될 때 SQL 문에서 오류가 발생할 수 있습니다. 오류 발생 여부는 사용되는 특정 데이터베이스 백 엔드에 따라 달라집니다. *TParam* 객체가 제공되었지만 SQL 문에 해당 매개변수가 없으면 애플리케이션에서 *TParam*을 사용하려고 할 때 예외가 발생됩니다.

연결된 데이터셋을 사용한 작업

모든 데이터베이스 연결 컴포넌트는 데이터베이스에 연결할 때 자신을 사용하는 모든 데이터셋의 리스트를 유지합니다. 예를 들어, 연결 컴포넌트는 이러한 리스트를 사용하여 데이터베이스 연결을 닫을 때 데이터셋을 모두 닫습니다.

특정 데이터베이스에 연결하기 위해 특정 연결 컴포넌트를 사용하는 모든 데이터셋에서 이 리스트를 사용하여 동작을 수행할 수도 있습니다.

서버와의 연결을 끊지 않고 모든 데이터셋 닫기

연결 컴포넌트의 연결을 끊으면 자동으로 데이터셋도 모두 닫힙니다. 하지만 데이터베이스 서버와의 연결을 끊지 않고 데이터셋을 모두 닫고자 하는 경우가 있습니다.

서버와의 연결을 끊지 않고 열려 있는 데이터셋을 모두 닫으려면 *CloseDataSets* 메소드를 사용할 수 있습니다.

*TADOConnection*과 *TIBDatabase*의 경우 *CloseDataSets*을 호출하면 연결이 열린 채로 있습니다. *TDatabase*와 *TSQLConnection*의 경우 *KeepConnection* 속성을 **true**로 설정해야 합니다.

연결된 데이터셋을 통한 반복

연결 컴포넌트를 사용하는 모든 데이터셋에서 모두 닫지 않고 동작을 수행하려면 *DataSets*와 *DataSetCount* 속성을 사용합니다. *DataSets*는 연결 컴포넌트에 연결된 모든 데이터셋의 인덱싱된 배열입니다. *TADOConnection*을 제외한 모든 연결 컴포넌트의 경우 이러한 리스트에는 활성 데이터셋만 포함됩니다. *TADOConnection*은 비활성 데이터셋도 열거합니다. *DataSetCount*는 이러한 배열에서 데이터셋의 개수입니다.

참고 일반적인 클라이언트 데이터셋인 *TClientDataSet*과 대조적으로, 특화된 클라이언트 데이터셋을 사용하여 업데이트 내용을 캐싱하면 *DataSets* 속성은 클라이언트 데이터셋 자체가 아닌 클라이언트 데이터셋에서 소유하는 내부 데이터셋을 열거합니다.

*DataSetCount*와 함께 *DataSets*을 사용하여 코드에서 현재 활성 중인 모든 데이터셋을 순환할 수 있습니다. 예를 들어, 다음 코드는 모든 활성 데이터셋을 순환하며 데이터셋에서 제공한 데이터를 사용하는 컨트롤을 사용할 수 없게 합니다.

```
for (int i = 0; i< MyDBConnection->DataSetCount; i++)
  MyDBConnection->DataSets[i]->DisableControls();
```

참고 *TADOConnection*은 데이터셋과 명령 객체를 지원합니다. *Commands*와 *CommandCount* 속성을 사용하여 데이터셋을 통해 반복하는 것처럼 명령 객체를 통해 반복할 수 있습니다.

메타데이터 얻기

모든 데이터베이스 연결 컴포넌트에서는 검색하는 메타데이터 타입별로 리스트가 다르긴 하지만 데이터베이스에서 메터데이터 리스트를 검색할 수 있습니다. 메타데이터를 검색한 메소드는 문자열 리스트를 서버에서 사용 가능한 다양한 엔티티 이름으로 채웁니다. 그리고 나서 이 정보를 사용하여 런타임 시 사용자들이 테이블을 동적으로 선택하게도 할 수 있습니다.

TADOConnection 컴포넌트를 사용하여 ADO 데이터 저장소에서 사용 가능한 테이블과 내장 프로시저에 대해 메타데이터를 검색할 수 있습니다. 그리고 나서 이러한 정보를 사용하여 사용자들이 테이블이나 내장 프로시저를 동적으로 선택하게도 할 수 있습니다.

사용 가능한 테이블 열거

GetTableNames 메소드는 테이블 이름 리스트를 기존의 문자열 리스트 객체에 복사합니다. 이러한 메소드는 열 테이블을 선택할 때 사용할 수 있는 테이블 이름으로 리스트 박스를 채울 때 사용할 수 있습니다. 다음 행은 데이터베이스의 모든 테이블의 이름으로 리스트 박스를 채웁니다.

```
MyDBConnection->GetTableNames(ListBox1->Items, false);
```

*GetTableNames*에는 테이블 이름으로 채울 문자열 리스트 및 리스트에서 시스템 테이블이나 일반적인 테이블을 포함할지 여부를 표시하는 부울 등 두 개의 매개변수가 있습니다. 모든 서버에서 메타데이터를 저장하는 시스템 테이블을 사용하는 것은 아니므로 시스템 테이블을 요청하면 비어 있는 리스트가 생길 수도 있습니다.

참고 대부분의 데이터베이스 연결 컴포넌트의 경우 *GetTableNames*는 두 번째 매개변수가 **false**일때 사용 가능한 비시스템(non-system) 테이블 리스트를 반환합니다. 하지만 *TSQLConnection*의 경우 시스템 테이블 이름만 가져오는 것이 아니라면 리스트에 추가된 타입에 대한 추가적인 제어가 필요합니다. *TSQLConnection*을 사용할 때 리스트에 추가된 이름의 타입은 *TableScope* 속성에 의해 제어됩니다. *TableScope*은 리스트에서 일반적인 테이블, 시스템 테이블, 동의어 및 뷰 중 하나나 모두를 포함할 것인지를 표시합니다.

테이블의 필드 열거

GetFieldNames 메소드는 기존의 문자열 리스트를 지정된 테이블의 모든 필드(열) 이름으로 채웁니다. *GetFieldNames*는 다음과 같이 필드를 나열할 테이블의 이름과 필드 이름으로 채워질 기존 문자열 리스트인, 두 개의 매개변수를 취합니다.

```
MyDBConnection->GetTableNames("Employee", ListBox1->Items);
```

사용 가능한 내장 프로시저 열거

데이터베이스에 포함된 모든 내장 프로시저를 열거하려면 *GetProcedureNames* 메소드를 사용합니다. 이 메소드는 다음과 같이 채워야 할 기존 문자열 리스트인 단일 매개변수를 취합니다.

```
MyDBConnection->GetProcedureNames(ListBox1->Items);
```

참고 *GetProcedureNames*는 *TADOConnection*과 *TSQLConnection*에서만 사용할 수 있습니다.

사용 가능한 인덱스 열거

특정 테이블에 정의된 모든 인덱스를 열거하려면 *GetIndexNames* 메소드를 사용합니다. 이 메소드는 다음과 같이 인덱스를 구해야 할 테이블과 채워야 할 기존 문자열 리스트인, 두 개의 매개변수를 취합니다.

```
MyDBConnection1->GetIndexNames("Employee", ListBox1->Items);
```

참고 대부분의 테이블 타입 데이터셋에는 동일한 메소드가 있지만 *GetIndexNames*는 *TSQLConnection*에서만 사용할 수 있습니다.

내장 프로시저 매개변수 열거

특정 내장 프로시저에 대해 정의된 모든 매개변수의 리스트를 구하려면 *GetProcedureParams* 메소드를 사용합니다. *GetProcedureParams*는 *TList* 객체를 매개변수 설명 구조에 대한 포인터로 채웁니다. 여기서 각 구조는 지정한 내장 프로시저의 이름, 인덱스, 매개변수 타입, 필드 타입 등 매개변수에 대해 설명합니다.

*GetProcedureParams*는 다음과 같이 내장 프로시저의 이름과 채워야 할 기존 *TList* 객체인 두 개의 매개변수를 취합니다.

```
MyDBConnection1->GetIndexNames("GetInterestRate", List1);
```

리스트에 추가한 매개변수 설명을 좀 더 익숙한 *TParams* 객체로 변환하려면 전역 *LoadParamListItems* 프로시저를 호출합니다. *GetProcedureParams*는 동적으로 개별 구조를 할당하므로 해당 정보를 사용한 작업이 끝나면 사용자 애플리케이션에서 구조를 해제해야 합니다. 전역 *FreeProcParams* 루틴에서 이러한 해제 작업을 할 수 있습니다.

참고 *GetProcedureParams*는 *TSQLConnection*에서만 사용할 수 있습니다.

장

22

데이터셋 이해

데이터를 액세스할 때의 기본 유닛은 객체의 데이터셋 패밀리입니다. 애플리케이션에서는 모든 데이터베이스 액세스에 데이터셋을 사용합니다. 데이터베이스 객체는 논리 테이블로 구성된 데이터베이스의 레코드 집합을 나타냅니다. 이러한 레코드는 단일 데이터베이스 테이블의 레코드가 되거나 쿼리나 내장 프로시저의 실행 결과를 나타낼 수 있습니다.

데이터베이스 애플리케이션에서 사용하는 모든 데이터셋 객체는 *TDataSet*의 자손으로 이 클래스에서 데이터 필드, 속성, 이벤트 및 메소드를 상속받습니다. 이 장에서는 데이터베이스 애플리케이션에서 사용하는 데이터셋 객체가 상속받는 *TDataSet*의 기능을 설명합니다. 데이터셋 객체를 사용하려면 이러한 공유 기능을 이해해야 합니다.

*TDataSet*은 가상화된 데이터셋이므로 이 속성 중 많은 속성이 **가상**이거나 **순수 가상**임을 의미합니다. *가상 메소드*는 해당 메소드의 구현이 자손 객체에서 오버라이드될 수 있거나 대개는 오버라이드되는 함수이거나 프로시저 선언입니다. *순수 가상 메소드*는 실제로 구현되지 않는 함수이거나 프로시저 선언입니다. 선언은 모든 자손 데이터셋 객체에서 구현되어야 하지만 각각의 객체에 의해 다르게 구현될 수 있는 메소드를 설명(해당 매개변수 및 반환 함수가 있는 경우 이에 대해서도 설명)하는 프로토타입입니다.

*TDataSet*에는 **순수 가상(virtual)** 메소드가 포함되어 있으므로 애플리케이션에서 직접 사용하는 경우 런타임 오류가 생성됩니다. 대신 기본 *TDataSet* 자손의 인스턴스를 만들어 애플리케이션에서 사용하거나 사용자 고유의 데이터셋 객체를 *TDataSet* 이나 그 자손에서 파생시켜 모든 **순수 가상** 메소드에 대해 구현을 작성해야 합니다.

*TDataSet*은 모든 데이터셋 객체에 공통되는 많은 내용을 정의합니다. 예를 들어, *TDataSet*은 하나 이상의 데이터베이스 테이블의 실제 열에 해당하는 *TField* 컴포넌트의 배열, 애플리케이션에서 제공하는 조회 필드 또는 계산된 필드와 같은 모든 데이터셋의 기본 구조를 정의합니다. *TField* 컴포넌트에 대한 자세한 내용은 23장, "필드 컴포넌트 작업"을 참조하십시오.

이 장에서는 *TDataSet*에서 도입하는 일반적인 데이터베이스 기능을 사용하는 방법을 설명합니다. *TDataSet*에서는 이러한 기능의 메소드를 도입하지만 모든 *TDataSet* 자손이 이러한 메소드를 구현하는 것은 아닙니다. 특히, 단방향 데이터셋은 제한된 부분 집합만 구현합니다.

TDataSet 자손 사용

*TDataSet*에는 여러 개의 직계 자손이 있으며 각각은 다른 데이터 액세스 메커니즘에 해당합니다. 이러한 자손을 직접 사용하는 것이 아니라 각 자손이 특정 데이터 액세스 메커니즘 사용을 위한 속성과 메소드를 도입합니다. 그런 다음 이러한 속성과 메소드는 다양한 서버 데이터 타입에 사용할 수 있는 자손 클래스에 의해 노출됩니다. *TDataSet*의 직계 자손은 다음과 같습니다.

- *TBDEDataSet*. 데이터베이스 서버와 통신하는 BDE(Borland Database Engine)를 사용합니다. 사용되는 *TBDEDataSet* 자손은 *TTable*, *TQuery*, *TStoredProc* 및 *TNestedTable*입니다. BDE 호환 데이터셋의 고유한 기능은 24장, "Borland Database Engine 사용"에서 설명합니다.
- *TCustomADODataSet*. ADO(ActiveX Data Objects)를 사용하여 OLEDB 데이터 저장소와 통신합니다. 사용되는 *TCustomADODataSet* 자손은 *TADODataSet*, *TADOTable*, *TADOQuery* 및 *TADOStoredProc*입니다. ADO 기반 데이터셋의 고유 기능은 25장, "ADO 컴포넌트 사용"에서 설명합니다.
- *TCustomSQLDataSet*. dbExpress를 사용하여 데이터베이스 서버와 통신합니다. 사용되는 *TCustomSQLDataSet* 자손은 *TSQLDataSet*, *TSQLTable*, *TSQLQuery* 및 *TSQLStoredProc*입니다. dbExpress 데이터셋의 고유 기능은 26장, "단방향 데이터셋 사용"에서 설명합니다.
- *TIBCustomDataSet*. InterBase 데이터베이스 서버와 직접 통신합니다. 사용되는 *TIBCustomDataSet* 자손은 *TIBDataSet*, *TIBTable*, *TIBQuery* 및 *TIBStoredProc*입니다.
- *TCustomClientDataSet*. 다른 데이터셋 컴포넌트의 데이터나 디스크에 있는 전용 파일의 데이터를 나타냅니다. 사용되는 *TCustomClientDataSet* 자손은 외부(소스) 데이터셋에 연결할 수 있는 *TClientDataSet*과 내부 소스 데이터셋을 사용하는, 특정 데이터 액세스 메커니즘에 특화된 클라이언트 데이터셋(*TBDEClientDataSet*, *TSQLClientDataSet* 및 *TIBClientDataSet*)입니다. 클라이언트 데이터셋의 고유 기능은 27장, "클라이언트 데이터셋 사용"에서 설명합니다.

이러한 *TDataSet* 자손이 갖는 다양한 데이터 액세스 메커니즘의 몇 가지 장단점은 18-1페이지의 "데이터베이스 사용"에서 설명합니다.

기본 데이터셋 이외에도 고유의 사용자 정의 *TDataSet* 자손을 만들 수 있습니다. 예를 들어, 데이터베이스 서버 이외의 프로세스(예:스프레드시트)로부터 데이터를 제공하는 경우입니다. 사용자 정의 데이터셋을 작성하면 여전히 사용자 인터페이스를 구축하는 데 VCL 데이터 컨트롤을 사용할 수 있는 반면에 선택한 메소드를 사용하는 데이터를 관리하는 데 유연성을 부여할 수 있습니다. 사용자 정의 컴포넌트를 만드는 방법에 관한 자세한 내용은 45장, "컴포넌트 생성 개요"를 참조하십시오.

각 *TDataSet* 자손에는 자체의 고유한 속성과 메소드가 있지만 자손 클래스에서 도입한 일부 속성과 메소드는 다른 데이터 액세스 메커니즘을 사용하는 다른 자손에서 도입한 내용과 같습니다. 예를 들어, "테이블" 컴포넌트(*TTable*, *TADOTable*, *TSQLTable* 및 *TIBTable*)에는 유사성이 있습니다. *TDataSet* 자손에서 도입한 공통 기능에 대해서는 22-23페이지의 "데이터셋의 타입"을 참조하십시오.

데이터셋 상태 결정

데이터셋의 *상태* 또는 *모드*는 데이터에서 이루어질 수 있는 작업을 결정합니다. 예를 들어, 데이터셋이 닫히면 데이터셋의 상태는 *dsInactive*가 되는데 이는 데이터에서 이루어질 수 있는 작업이

없다는 의미입니다. 런타임시 데이터셋의 읽기 전용 *State* 속성을 사용하여 현재 상태를 결정할 수 있습니다. 다음 표는 *State* 속성에서 사용할 수 있는 값과 그 의미를 요약한 것입니다.

표 22.1 데이터셋 State 속성의 값

값	상태	의미
dsInactive	Inactive	데이터셋이 닫혀 있습니다. 데이터를 사용할 수 없습니다.
dsBrowse	Browse	DataSet이 열려 있습니다. 데이터를 볼 수는 있지만 변경할 수는 없습니다. 열려 있는 데이터셋의 디폴트 상태입니다.
dsEdit	Edit	DataSet이 열려 있습니다. 현재 행을 수정할 수 있습니다(단방향 데이터셋에서는 지원되지 않음).
dsInsert	Insert	DataSet이 열려 있습니다. 새 행을 삽입하거나 추가할 수 있습니다 (단방향 데이터셋에서는 지원되지 않음).
dsSetKey	SetKey	DataSet이 열려 있습니다. 범위와 *GotoKey* 작업에서 사용되는 범위 및 키 값의 설정을 사용 가능하게 합니다(모든 데이터셋에서 지원되는 것은 아님).
dsCalcFields	CalcFields	DataSet이 열려 있습니다. *OnCalcFields* 이벤트가 진행 중임을 표시합니다. 계산되지 않은 필드에 대한 변경을 못하게 합니다.
dsCurValue	CurValue	DataSet이 열려 있습니다. 캐싱된 업데이트를 적용할 때 발생한 오류에 응답하는 이벤트 핸들러에 대해 필드의 CurValue 속성을 가져오는 중임을 표시합니다.
dsNewValue	NewValue	DataSet이 열려 있습니다. 캐싱된 업데이트를 적용할 때 발생한 오류에 응답하는 이벤트 핸들러에 대해 필드의 NewValue 속성을 가져오는 중임을 표시합니다.
dsOldValue	OldValue	DataSet이 열려 있습니다. 캐싱된 업데이트를 적용할 때 발생한 오류에 응답하는 이벤트 핸들러에 대해 필드의 OldValue 속성을 가져오는 중임을 표시합니다.
dsFilter	Filter	DataSet이 열려 있습니다. 필터 작업이 진행 중임을 표시합니다. 제한된 데이터 집합을 볼 수 있지만 데이터를 변경할 수는 없습니다 (단방향 데이터셋에서는 지원되지 않음).
dsBlockRead	Block Read	DataSet이 열려 있습니다. 현재 레코드가 변경될 때 데이터 인식 컨트롤이 업데이트되지 않으며 이벤트가 실행되지 않습니다.
dsInternalCalc	Internal Calc	DataSet이 열려 있습니다. 레코드와 함께 저장되는 계산된 값에 대해 *OnCalcFields* 이벤트가 진행 중입니다(클라이언트 데이터셋만 해당).
dsOpening	Opening	DataSet을 열고 있는 중인데 아직 완료되지 않았습니다. 비동기 페치(fetch)를 위해 데이터셋이 열려 있을 때 이 상태가 발생합니다.

일반적으로 애플리케이션에는 데이터셋 상태를 검사하여 특정 작업을 수행할 시기를 결정합니다. 예를 들어, *dsEdit*이나 *dsInsert* 상태를 검사하여 업데이트를 포스트해야 할지 여부를 확인합니다.

참고 데이터셋의 상태가 변경될 때마다 데이터셋과 연결된 데이터 소스 컴포넌트에 대해 *OnStateChange* 이벤트가 호출됩니다. 데이터 소스 컴포넌트와 *OnStateChange*에 대한 자세한 내용은 19-4페이지의 "데이터 소스에 따른 변경 내용에 대한 응답"을 참조하십시오.

데이터셋 열기 및 닫기

데이터셋에서 데이터를 읽거나 쓰려면 애플리케이션에서 먼저 데이터셋을 열어야 합니다. 다음과 같은 두 가지 방법으로 데이터셋을 열 수 있습니다.

- 디자인 타임 시 Object Inspector에서 또는 런타임 시 코드에서 데이터셋의 *Active* 속성을 **true**로 설정합니다.

  ```
  CustTable->Active = true;
  ```

- 런타임 시 데이터셋의 *Open* 메소드를 호출합니다.

  ```
  CustQuery->Open();
  ```

데이터셋을 열면 데이터셋에서 먼저 *BeforeOpen* 이벤트를 받고 커서를 열어 데이터로 채운 다음 마지막으로 *AfterOpen* 이벤트를 받습니다.

새로 연 데이터셋은 찾아보기 모드에 있으며 이는 애플리케이션에서 데이터를 읽고 탐색할 수 있음을 의미합니다.

다음과 같은 두 가지 방법으로 데이터셋을 닫을 수 있습니다.

- 디자인 타임 시 Object Inspector에서 또는 런타임 시 코드에서 데이터셋의 *Active* 속성을 **false**로 설정합니다.

  ```
  CustQuery->Active = false;
  ```

- 런타임 시 데이터셋의 *Close* 메소드를 호출합니다.

  ```
  CustTable->Close();
  ```

데이터셋을 열면 데이터셋에서 *BeforeOpen*과 *AfterOpen* 이벤트를 받는 것과 마찬가지로 데이터셋을 닫으면 데이터셋에 대한 *Close* 메소드에 응답하는 핸들러인 *BeforeClose*와 *AfterClose* 이벤트를 받습니다. 예를 들어, 이러한 이벤트를 사용하여 변경 내용 보류를 포스트할 것인지 아니면 데이터셋을 닫기 전에 변경 내용을 취소할 것인지 묻는 메시지를 사용자에게 표시합니다. 다음 코드에서는 그러한 핸들러를 설명합니다.

```
void __fastcall TForm1::VerifyBeforeClose(TDataSet *DataSet)
{
  if (DataSet->State == dsEdit || DataSet->State == dsInsert)
  {
    TMsgDlgButtons btns;
    btns << mbYes << mbNo;
  if (MessageDlg("Post changes before closing?", mtConfirmation, btns, 0)
== mrYes)
      DataSet->Post();
    else
      DataSet->Cancel();
  }
}
```

참고 *TTable* 컴포넌트의 *TableName*의 경우처럼 데이터셋의 특정 속성을 변경하려면 데이터셋을 닫아야 합니다. 데이터셋을 다시 열면 새 속성 값이 적용됩니다.

데이터셋 탐색

각 활성 데이터셋에는 데이터셋의 현재 행에 대한 *커서*나 포인터가 있습니다. 데이터셋의 *현재 행*은 *TDBEdit*, *TDBLabel* 및 *TDBMemo*와 같이 필드 값이 폼의 현재 단일 필드인 데이터 인식 컨트

롤에 표시되는 행입니다. 데이터셋에서 편집을 지원하면 현재 레코드는 편집, 삽입, 삭제 메소드에 의해 처리될 수 있는 값을 포함합니다.

커서가 다른 행을 가리키도록 이동하여 현재 행을 변경할 수 있습니다. 다음 표는 다른 레코드로 이동하기 위해 애플리케이션 코드에서 사용할 수 있는 메소드를 나열합니다.

표 22.2 데이터셋의 탐색 메소드

메소드	커서 이동 위치
First	데이터셋의 첫 행
Last	데이터셋의 마지막 행(단방향 데이터셋에서는 사용할 수 없음)
Next	데이터셋의 다음 행
Prior	데이터셋의 이전 행(단방향 데이터셋에서는 사용할 수 없음)
MoveBy	데이터셋에서 지정한 수만큼 앞이나 뒤에 있는 행

데이터 인식, 비주얼(visual) 컴포넌트인 *TDBNavigator*는 이러한 메소드를 런타임 시 레코드 사이를 이동할 때 클릭할 수 있는 버튼으로 캡슐화합니다. 탐색기 컴포넌트에 대한 자세한 내용은 19-28페이지의 "레코드 탐색 및 처리"를 참조하십시오.

이러한 메소드 중 하나를 사용하거나 검색 조건을 기준으로 탐색하는 다른 메소드를 사용하여 현재 레코드를 변경할 때마다 데이터셋에서는 *BeforeScroll*(현재 레코드를 떠나기 전에)과 *AfterScroll*(새 레코드에 도착하기 전에) 등, 두 가지 이벤트를 받습니다. 이러한 이벤트를 사용하여 사용자 인터페이스를 업데이트할 수 있습니다. 예를 들어, 현재 레코드의 정보를 표시하는 상태 표시줄을 업데이트하는 경우입니다.

또한 *TDataSet*은 데이터셋의 레코드를 반복할 때 유용한 정보를 제공하는 두 개의 부울 속성을 정의합니다.

표 22.3 데이터셋의 탐색 속성

속성	설명
Bof(Beginning-of-file)	**true**: 커서가 데이터셋의 첫 번째 행에 있습니다.
	false: 커서가 데이터셋의 첫 행에 있는지 알 수 없습니다.
Eof(End-of-file)	**true**: 커서가 데이터셋의 마지막 행에 있습니다.
	false: 커서가 데이터셋의 마지막 행에 있는지 알 수 없습니다.

First와 Last 메소드 사용

First 메소드는 커서를 데이터셋의 첫 행으로 이동한 다음 *Bof* 속성을 **true**로 설정합니다. 커서가 이미 데이터셋의 첫 행에 있으면 *First*는 아무 작업도 하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 첫 번째 레코드로 이동합니다.

```
CustTable->First();
```

Last 메소드는 커서를 데이터셋의 마지막 행으로 이동한 다음 *Eof* 속성을 **true**로 설정합니다. 이미 커서가 데이터셋의 마지막 행에 있으면 *Last*는 아무 작업도 하지 않습니다.

다음 코드는 *CustTable*의 마지막 레코드로 이동합니다.

```
CustTable->Last();
```

참고 단방향 데이터셋에서는 *Last* 메소드가 예외를 발생합니다.

팁 사용자 간섭없이 데이터셋의 첫 행이나 마지막 행으로 이동해야 하는 프로그래밍 상의 이유가 있을 수도 있지만 *TDBNavigator* 컴포넌트를 사용하여 사용자가 레코드 간을 탐색할 수 있게 만들 수도 있습니다. 탐색기 컴포넌트에는, 활성화되어 볼 수 있게되면, 사용자가 활성 데이터셋의 첫 번째 행과 마지막 행으로 이동할 수 있게 하는 버튼이 있습니다. 이러한 버튼의 *OnClick* 이벤트는 데이터셋의 *First*와 *Last* 메소드를 호출합니다. 탐색기 컴포넌트를 효과적으로 사용하는 방법에 대한 자세한 내용은 19-28페이지의 "레코드 탐색 및 처리"를 참조하십시오.

Next와 Prior 메소드 사용

Next 메소드는 데이터셋의 한 행 앞으로 커서를 이동한 다음 데이터셋이 비어 있지 않으면 *Bof* 속성을 **false**로 설정합니다. *Next*를 호출할 때 커서가 이미 데이터셋의 마지막 행에 있으면 아무 일도 발생하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 다음 레코드로 이동합니다.

```
CustTable->Next();
```

Prior 메소드는 데이터셋의 한 행 뒤로 커서를 이동한 다음 데이터셋이 비어 있지 않으면 *Eof*를 **false**로 설정합니다. *Prior*를 호출할 때 커서가 이미 데이터셋의 첫 행에 있으면 *Prior*에서는 아무 작업도 하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 이전 레코드로 이동합니다.

```
CustTable->Prior();
```

참고 단방향 데이터셋에서 *Prior* 메소드는 예외를 발생합니다.

MoveBy 메소드 사용

*MoveBy*를 사용하면 데이터셋에서 커서 이동을 위한 앞뒤의 행 수를 지정할 수 있습니다. *MoveBy*를 호출할 때의 현재 레코드를 기준으로 이동이 이루어집니다. *MoveBy*는 데이터셋의 *Bof*와 *Eof* 속성을 적절하게 설정하기도 합니다.

이 함수에서는 정수 매개변수인 이동할 레코드 수를 취합니다. 양의 정수는 앞으로 이동을 나타내고 음의 정수는 뒤로 이동을 나타냅니다.

참고 음의 인수를 사용하는 경우 단방향 데이터셋에서는 *MoveBy*가 예외를 발생합니다.

*MoveBy*는 이동해야 할 행 수를 반환합니다. 데이터셋의 처음이나 끝을 지나 이동하려고 시도하면 *MoveBy*에서 반환하는 행 수는 이동을 요청했던 행 수와 달라집니다. 이유는 *MoveBy*가 데이터셋의 첫 레코드나 마지막 레코드에 도달하면 멈추기 때문입니다.

다음 코드는 *CustTable*에서 뒤로 두 레코드 이동합니다.

```
CustTable->MoveBy(-2);
```

참고 애플리케이션에서 *MoveBy*를 여러 사용자 데이터베이스 환경에서 사용하는 경우 데이터셋이 유동함에 유의하십시오. 여러 사용자가 동시에 데이터베이스에 액세스하여 데이터를 변경하면 좀 전에 5 레코드 뒤로 이동하려던 레코드가 이제 4나 6 또는 알 수 없는 레코드 수만큼 뒤로 이동합니다.

Eof와 Bof 속성 사용

두 개의 읽기 전용, 런타임 속성인 *Eof*(End-of-file)와 *Bof*(Beginning-of-file)는 데이터셋의 모든 레코드를 반복할 때 유용합니다.

Eof

*Eof*가 **true**이면 커서가 확실히 데이터셋의 마지막 행에 있음을 의미합니다. 다음과 같은 경우에 *Eof*가 **true**로 설정됩니다.

- 비어 있는 데이터셋을 연 경우입니다.
- 데이터셋의 *Last* 메소드를 호출한 경우입니다.
- 데이터셋의 *Next* 메소드를 호출하였는데 커서가 현재 데이터셋의 마지막 행에 있기 때문에 메소드가 실패한 경우입니다.
- 비어 있는 범위나 데이터셋에서 *SetRange*를 호출한 경우입니다.

다른 모든 경우에는 *Eof*가 **false**로 설정됩니다. 위의 조건 중 하나가 맞지 *않았고* 속성을 직접 테스트했다면 *Eof*가 **false**인 것으로 가정해야 합니다.

*Eof*는 데이터셋의 모든 레코드를 반복적으로 처리하는 루프 조건에서 일반적으로 테스트됩니다. 레코드가 들어 있는 데이터셋을 열거나 *First*를 호출하면 *Eof*가 **false**입니다. 한 번에 한 레코드씩 데이터셋을 반복하려면 *Next*를 호출하여 한 레코드씩 진행하는 루프를 만들고 *Eof*가 **true**일 때 종료해야 합니다. 커서가 이미 마지막 레코드에 있으면 *Next*를 호출할 때까지 *Eof*는 **false**로 남아 있습니다.

다음 코드에서는 *CustTable*이라는 데이터셋의 레코드 처리 루프를 코딩하는 한 가지 방법을 설명합니다.

```
CustTable->DisableControls();
try
{
  for (CustTable->First(); !CustTable->Eof; CustTable->Next())
  (
    // Process each record here
    f
  }
}
__finally
{
  CustTable->EnableControls();
}
```

팁 이 예제에서는 데이터셋에 연결된 데이터 인식 비주얼(visual) 컨트롤을 활성화하거나 비활성화하는 방법을 보여 줍니다. 데이터셋 반복 중에 비주얼 컨트롤을 비활성화하려면 애플리케이션에서 현재 레코드가 변경될 때마다 컨트롤의 내용을 업데이트할 필요가 없으므로 처리 속도가 빨라집니다. 반복이 완료되면 새 현재 행 값으로 컨트롤을 다시 업데이트할 수 있도록 컨트롤을 활성화해야 합니다. 비주얼 컨트롤의 활성화는 **try...__finally** 문의 **__finally** 절에서 발생해야 함에 유의하십시오. 그래야만 예외로 인해 루프 처리가 조기에 종료되더라도 컨트롤이 사용 가능 상태로 남을 수 있습니다.

Bof

*Bof*가 **true**이면 커서가 확실히 데이터셋의 첫 번째 행에 있음을 의미합니다. 다음과 같은 경우에 *Bof*가 **true**로 설정됩니다.

- 데이터셋을 여는 경우입니다.
- 데이터셋의 *First* 메소드를 호출하는 경우입니다.
- 데이터셋의 *Prior* 메소드를 호출하였는데 커서가 현재 데이터셋의 첫 행에 있기 때문에 메소드가 실패한 경우입니다.
- 비어 있는 범위나 데이터셋에서 *SetRange*를 호출한 경우입니다.

다른 모든 경우에는 *Bof*가 **false**로 설정됩니다. 위의 조건 중 하나가 맞지 *않았고* 속성을 직접 테스트했다면 *Bof*가 **false**인 것으로 가정해야 합니다.

*Eof*처럼 *Bof*는 데이터셋의 레코드를 반복적으로 처리하는 루프 조건에 있을 수 있습니다. 다음 코드에서는 *CustTable*이라는 데이터셋의 레코드 처리 루프를 코딩하는 한 가지 방법을 설명합니다.

```
CustTable->DisableControls(); // Speed up processing; prevent screen
flicker
try
{
  while (!CustTable->Bof) // Cycle until Bof is true
  (
    // Process each record here
    f
    CustTable->Prior();
    // Bof false on success; Bof true when Prior fails on first record
  }
}
__finally
{
  CustTable->EnableControls();
}
```

레코드 표시 및 레코드로 돌아가기

데이터셋에서 레코드 간을 이동하거나 레코드의 특정 번호로 레코드 간을 이동하는 것 이외에도 원하는 때에 즉시 데이터셋의 특정 위치로 돌아갈 수 있도록 데이터셋의 특정 위치를 표시하는 것도 유용한 방법입니다 . *TDataSet* 은 *Bookmark* 속성과 다섯 개의 북마크 메소드로 이루어진 북마킹 기능을 도입합니다.

*TDataSet*은 **가상** 북마크 메소드를 구현합니다. 이러한 메소드는 북마크를 호출할 때 *TDataSet*에서 파생된 데이터셋 객체에서 값을 반환하도록 하지만 반환 값은 단순히 기본값이므로 현재 위치를 추적하지는 않습니다. *TDataSet* 자손은 북마크를 제공하는 지원 레벨이 다릅니다. dbExpress 데이터셋에서는 북마크를 전혀 추가하지 않습니다. ADO 데이터셋에서는 원본으로 사용하는 데이터베이스 테이블에 따라 북마크를 지원할 수 있습니다. BDE 데이터셋, InterBase Express 데이터셋 및 클라이언트 데이터셋에서는 항상 북마크를 지원합니다.

Bookmark 속성

Bookmark 속성은 애플리케이션에 있는 많은 북마크 중에서 사용 중인 북마크를 표시합니다. *Bookmark*는 현재 북마크를 식별하는 문자열입니다. 다른 북마크를 추가할 때마다 이 북마크가 현재 북마크가 됩니다.

GetBookmark 메소드

북마크를 만들려면 애플리케이션에서 *TBookmark* 타입의 변수를 선언한 다음 *GetBookmark*를 호출하여 변수에 필요한 저장소를 할당하고 데이터셋의 특정 위치로 변수 값을 설정해야 합니다. *TBookmark* 타입은 포인터(void *)입니다.

GotoBookmark 및 BookmarkValid 메소드

북마크를 전달하면 *GotoBookmark*는 데이터셋의 커서를 북마크에서 지정한 위치로 이동시킵니다. *GotoBookmark*를 호출하기 전에 북마크가 레코드를 가리키고 있는지 확인하기 위해 *BookmarkValid*를 호출할 수 있습니다. 지정한 북마크가 레코드를 가리키면 *BookmarkValid*는 **true**를 반환합니다.

CompareBookmarks 메소드

이동하려는 북마크가 다른 북마크 또는 현재 북마크와 다른지 알아보려면 *CompareBookmarks*를 호출할 수도 있습니다. 두 개의 북마크에서 같은 레코드를 참조하거나 두 개의 북마크가 모두 NULL이면 *CompareBookmarks*는 0을 반환합니다.

FreeBookmark 메소드

*FreeBookmark*는 특정 북마크가 더 이상 필요하지 않을 때 이 북마크에 할당된 메모리를 해제합니다. 기존 북마크를 다시 사용하기 전에도 *FreeBookmark*를 호출해야 합니다.

북마킹 예제

다음 코드에서는 북마킹 사용 방법 중 하나를 설명합니다.

```
void DoSomething (const TTable *Tbl)
{
  TBookmark Bookmark = Tbl->GetBookmark(); // Allocate memory and assign
a value
  Tbl->DisableControls(); // Turn off display of records in data controls
  try
  {
    for (Tbl->First(); !Tbl->Eof; Tbl->Next()) // Iterate through each
record in table
    {
      // Do your processing here
      f
    }
  }
  __finally
  {
    Tbl->GotoBookmark(Bookmark);
    Tbl->EnableControls(); // Turn on display of records in data controls
    Tbl->FreeBookmark(Bookmark); // Deallocate memory for the bookmark
  }
}
```

레코드를 반복하기 전에 컨트롤은 비활성화되었습니다. 레코드를 반복할 때 오류가 발생하면 **__finally** 절은 루프가 조기에 종료되더라도 북마크를 항상 해제하고 컨트롤을 항상 활성화합니다.

데이터셋 검색

데이터셋이 단방향이 아니면 *Locate*와 *Lookup* 메소드를 사용하여 데이터셋을 검색할 수 있습니다. 이러한 메소드를 사용하면 데이터셋의 모든 열 타입에서 검색할 수 있습니다.

참고 일부 *TDataSet* 자손은 인덱스를 기반으로 검색할 때 추가 메소드 패밀리를 도입합니다. 이러한 추가 메소드에 대한 자세한 내용은 22-27페이지의 "인덱스를 사용하여 레코드 검색"을 참조하십시오.

Locate 사용

*Locate*에서는 특정 검색 기준 집합에 일치하는 첫 번째 행으로 커서를 이동합니다. 검색할 열 이름, 일치하는 필드 값 및 검색 시 대소문자를 구분할 것인지 또는 부분 키 일치를 사용할 수 있는지를 지정하는 옵션 플래그를 가장 간단한 형식으로 *Locate*에 전달하면 됩니다. 부분 키 일치는 기준 문자열에 필드 값의 접두사만 있는 경우 사용합니다. 예를 들어, 다음 코드에서는 *Company* 열의 값이 "Professional Divers, Ltd."인 경우 *CustTable*의 첫 행으로 커서를 이동합니다.

```
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey;
bool LocateSuccess = CustTable->Locate(“Company”, “Professional Divers,
Ltd.”,
    SearchOptions);
```

*Locate*에서 일치하는 내용을 찾으면 일치하는 내용을 포함하는 첫 번째 레코드가 현재 레코드가 됩니다. *Locate*에서 일치하는 레코드를 찾으면 **true**를 반환하고 레코드를 찾지 못하면 **false**를 반환합니다. 검색에 실패하면 현재 레코드는 변경되지 않습니다.

*Locate*의 진가는 여러 열에서 검색하고 검색할 여러 값을 지정할 때 발휘됩니다. 검색 값은 Variants이므로 검색 기준에서 다양한 데이터 타입을 지정할 수 있습니다. 검색 문자열에서 여러 열을 지정하려면 문자열에서 개별 항목을 세미콜론으로 구분해야 합니다.

검색 값이 Variants이기 때문에 여러 값을 전달하는 경우 Variant 배열을 인수(예를 들어, *Lookup* 메소드의 반환 값)로 전달하거나 *VarArrayOf* 함수를 사용하여 코드에서 Variant 배열을 생성해야 합니다. 다음 코드에서는 여러 검색 값과 부분 키 일치를 사용하여 여러 열을 검색하는 경우를 설명합니다.

```
TLocateOptions Opts;
Opts.Clear();
Opts << loPartialKey;
Variant locvalues[2];
locvalues[0] = Variant(“Sight Diver”);
locvalues[1] = Variant(“P”);
CustTable->Locate(“Company;Contact”, VarArrayOf(locvalues, 1), Opts);
```

*Locate*에서는 일치하는 레코드를 찾기 위해 가장 빠른 메소드를 사용합니다. 검색할 열이 인덱싱되어 있고 인덱스가 지정한 검색 옵션과 호환되면 *Locate*에서는 인덱스를 사용합니다.

조회 사용

*Lookup*에서는 지정한검색 기준에 일치하는 첫 번째 행을 검색합니다. 일치하는 행을 찾으면 데이터셋과 연결된 계산된 필드와 조회 필드를 다시 계산한 다음 일치하는 열에서 한 개 이상의 필드를 반환합니다. *Lookup*에서는 커서를 일치하는 행으로 반환하지 않고 행의 값만 반환합니다.

검색할 열 이름, 일치하는 필드 값 및 반환할 필드를 가장 간단한 형식으로 *Lookup*에 전달하면 됩니다. 예를 들어, 다음 코드에서는 *Company* 필드 값이 "Professional Divers, Ltd"인 *CustTable*의 첫 번째 레코드를 찾아 회사 이름, 연락 담당자 및 회사 전화 번호를 반환합니다.

```
Variant LookupResults = CustTable->Lookup(“Company”, “Professional
Divers, Ltd”,
    “Company;Contact;Phone”);
```

*Lookup*에서는 찾은 첫 번째 일치하는 레코드에서 지정한 필드의 값을 반환합니다. 값은 Variants로 반환됩니다. 반환 값이 두 개 이상 요청되면 *Lookup*에서는 Variant 배열을 반환합니다. 일치하는 레코드가 없으면 *Lookup*에서는 Null Variant를 반환합니다. Variant 배열에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

*Lookup*의 진가는 여러 열에서 검색하고 검색할 여러 값을 지정할 때 발휘됩니다. 여러 열이나 결과 필드를 포함하는 문자열을 지정하려면 문자열에서 각각의 필드를 세미콜론으로 구분해야 합니다.

검색 값이 Variants이기 때문에 여러 값을 전달하는 경우 Variant 배열을 인수(예를 들어, *Lookup* 메소드의 반환 값)로 전달하거나 *VarArrayOf* 함수를 사용하여 코드에서 Variant 배열을 생성해야 합니다. 다음 코드에서는 여러 열을 조회 검색하는 경우를 설명합니다.

```
Variant LookupResults;
Variant locvalues[2];
Variant v;

locvalues[0] = Variant(“Sight Diver”);
locvalues[1] = Variant(“Kato Paphos”);
LookupResults = CustTable->Lookup(“Company;City”, VarArrayOf(locvalues,
1),
                  “Company;Addr1;Addr2;State;Zip”);
// now put the results in a global stringlist (created elsewhere)
pFieldValues->Clear();
for (int i = 0; i < 5; i++) // Lookup call requested 5 fields
{
  v =LookupResults.GetElement(i);
  if (v.IsNull())
    pFieldValues->Add(“”);
  else
    pFieldValues->Add(v);
}
```

*Locate*와 같이 *Lookup*에서도 일치하는 레코드를 찾기 위해 가장 빠른 메소드를 사용합니다. 검색할 열이 인덱싱되어 있으면 *Lookup*에서는 인덱스를 사용합니다.

필터를 사용하여 데이터의 부분 집합 표시 및 편집

특정 애플리케이션에서는 데이터셋 레코드의 부분 집합에만 관심을 갖습니다. 예를 들어, 고객 데이터베이스에서 캘리포니아에 기반을 둔 회사의 레코드만 검색해서 봐야 할 경우가 있거나 특정 필드 값 집합만 갖는 레코드를 찾아야 하는 경우도 있습니다. 이러한 경우 필터를 사용하여 애플리케이션의 액세스를 데이터셋의 모든 레코드의 부분 집합으로 제한할 수 있습니다.

단방향 데이터셋을 사용하면 데이터셋의 레코드를 제한하는 쿼리를 사용하여 데이터셋의 레코드만 제한할 수 있습니다. 하지만 다른 *TDataSet* 자손을 사용하면 이미 가져온 데이터의 부분 집합을 정의할 수 있습니다. 애플리케이션의 액세스를 데이터셋의 모든 레코드의 부분 집합으로 제한하려면 필터를 사용할 수 있습니다.

레코드에서 맞춰야 할 필터의 특정 조건이 표시되어야 합니다. 필터 조건은 데이터셋의 *Filter* 속성에서 규정하거나 이 속성의 *OnFilterRecord* 이벤트 핸들러에서 코딩할 수 있습니다. 필터 조건은 데이터셋의 필터가 인덱싱되었는지 여부에 상관없이 특정 수의 필드에 있는 값을 기반으로 합니다. 예를 들어, 캘리포니아에 기반을 둔 회사에 관한 레코드만 보려는 경우 간단한 필터에서는 State 필드 값에 "CA"를 포함하는 레코드만 필요로 할 것입니다.

참고 필터는 데이터셋에서 검색된 레코드마다 적용됩니다. 많은 양의 데이터를 필터링해야 하는 경우 필터를 사용하는 것보다는 쿼리를 사용하여 레코드 검색을 제한하거나 인덱싱된 테이블에 범위를 설정하는 것이 더 효율적일 수 있습니다.

필터링 활성화 및 비활성화

데이터셋에서 필터를 활성화하려면 다음 3단계 과정을 거쳐야 합니다.

1. 필터를 만듭니다.
2. 필요하면 문자열 기반 필터 테스트를 위해 필터 옵션을 설정합니다.
3. *Filtered* 속성을 **true**로 설정합니다.

필터링이 활성화되면 필터 기준에 맞는 레코드만 애플리케이션에서 사용할 수 있습니다. 필터링은 항상 일시적 조건입니다. *Filtered* 속성을 *false*로 설정하여 필터링 기능을 끌 수 있습니다.

필터 생성

데이터셋의 필터를 만드는 방법은 다음과 같이 두 가지입니다.

- *Filter* 속성에서 간단한 필터 조건을 지정합니다. *Filter*는 런타임 시 필터를 만들어 적용하는 데 특히 유용합니다.
- 간단하거나 복잡한 필터 조건에 대해 *OnFilterRecord* 이벤트 핸들러를 작성합니다. *OnFilterRecord*를 사용하면 디자인 타임 시 필터 조건을 지정합니다. 필터 로직을 포함하는 단일 문자열로만 제한하는 *Filter* 속성과는 달리, *OnFilterRecord* 이벤트는 분기 및 루프 로직을 이용하여 복잡한, 다중 레벨의 필터 조건을 만들 수 있습니다.

Filter 속성을 사용하여 필터를 만들 때의 주요 장점은 사용자 입력에 응답하는 경우처럼, 애플리케이션에서 필터를 동적으로 만들고 변경하여 적용할 수 있다는 것입니다. 반면 주요 단점은 필터 조건을 단일 텍스트 문자열로 표현해야 하므로 분기 및 루프 구조를 사용할 수 없고 데이터셋에 아직 없는 값과 필터 값을 테스트하거나 비교할 수 없다는 것입니다.

OnFilterRecord 이벤트의 강점은 필터가 복잡하거나 변수일 수 있고 분기 및 루프 구조를 사용하는 여러 줄의 코드를 기반으로 할 수 있으며 에디트 박스의 텍스트와 같은, 데이터셋 외부의 값에 대해 데이터셋 값을 테스트할 수 있다는 것입니다. *OnFilterRecord*를 사용할 때의 가장 큰 단점은 디자인 타임 시 필터를 설정하기 때문에 사용자 입력에 응답하여 수정할 수 없다는 것입니다. 하지만 여러 개의 필터 핸들러를 만든 다음 일반적인 애플리케이션 조건에 응답하여 핸들러 중 하나로 전환할 수 있습니다.

다음 단원에서는 *Filter* 속성과 *OnFilterRecord* 이벤트 핸들러를 사용하여 필터를 만드는 방법을 설명합니다.

Filter 속성 설정

Filter 속성을 사용하여 필터를 만들려면 속성 값을 필터 테스트 조건이 포함된 문자열로 설정합니다. 예를 들어, 다음 명령문에서는 데이터셋의 State 문에 캘리포니아 주에 대한 값이 포함되어 있는지 알아보는 테스트를 하기 위해 필터를 만듭니다.

```
Dataset1->Filter = “State = ‘CA’”;
```

사용자가 입력한 텍스트를 기반으로 *Filter* 값을 입력할 수도 있습니다. 예를 들어, 다음 명령문에서는 에디트 박스로부터 입력된 텍스트를 *Filter*에 할당합니다.

```
Dataset1->Filter = Edit1->Text;
```

물론 하드 코딩된 텍스트와 사용자 입력 데이터를 기반으로 문자열을 만들 수도 있습니다.

```
Dataset1->Filter = AnsiString(“State = ‘”) + Edit1->Text + “’”;
```

비어 있는 필드 값은 다음과 같이 명시적으로 필터에 포함되어 있지 않으면 표시되지 않습니다.

```
Dataset1->Filter = “State <> ‘CA’ or State = BLANK”;
```

참고 *Filter*의 값을 지정한 다음 필터를 데이터셋에 적용하려면 *Filtered* 속성을 **true**로 설정합니다.

필터는 다음과 같은 비교 및 논리 연산자를 사용하여 필드 값을 리터럴 및 상수와 비교할 수 있습니다.

표 22.4 필터에 표시될 수 있는 비교 및 논리 연산자

연산자	의미
<	작다
>	크다
>=	크거나 같다
<=	작거나 같다
=	같다
<>	같지 않다
AND	두 개의 명령문이 모두 **true**인지 테스트합니다.
NOT	다음 명령문이 **true**가 아닌지 테스트합니다.
OR	두 개의 명령문 중 하나가 적어도 **true**인지 테스트합니다.
+	숫자를 추가하거나 문자열을 연결하고 숫자를 날짜/시간 값에 추가합니다(일부 드라이버에서만 사용 가능).
-	숫자를 빼거나 날짜를 빼거나 날짜에서 숫자를 뺍니다(일부 드라이버에서만 사용 가능).
*	두 개의 숫자를 곱합니다(일부 드라이버에서만 사용 가능).
/	두 개의 숫자를 나눕니다(일부 드라이버에서만 사용 가능).
*	부분 비교용 와일드카드입니다(*FilterOptions* 에 *foPartialCompare* 가 포함되어 있어야만 함).

이러한 연산자를 조합하여 아주 복잡한 필터를 만들 수 있습니다. 예를 들어, 다음 명령문은 표시할 레코드를 받아들이기 전에 두 개의 테스트 조건이 맞는지 확인하기 위해 검사합니다.

```
(Custno > 1400) AND (Custno < 1500);
```

참고 필터링 기능이 켜져 있는 경우, 사용자가 레코드를 편집하는 것은 해당 레코드가 더 이상 필터의 테스트 조건에 맞지 않음을 의미할 수 있습니다. 다음에 레코드를 데이터셋에서 검색할 때 해당 레코드는 사라질 수 있습니다. 그런 경우 필터 조건을 전달하는 다음 레코드가 현재 레코드가 됩니다.

OnFilterRecord 이벤트 핸들러 작성

검색할 각 레코드에 대해 데이터셋에서 생성한 *OnFilterRecord* 이벤트를 사용하여 레코드를 필터링하는 코드를 작성할 수 있습니다. 이 이벤트 핸들러에서는 애플리케이션에서 볼 수 있는 레코드에 특정 레코드가 포함되어야 하는지를 결정하는 테스트를 구현합니다.

특정 레코드에서 필터 조건을 전달할 지 여부를 표시하는 경우 해당 레코드를 포함하려면 *OnFilterRecord* 핸들러의 *Accept* 매개변수를 **true** 로 설정하고 해당 레코드를 제외하려면 **false** 로 설정합니다. 예를 들어, 다음 필터는 State 필드가 "CA"로 설정된 레코드만 표시합니다.

```
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet; bool &Accept)
{
  Accept = DataSet->FieldByName[“State”]->AsString == “CA”;
}
```

필터링이 활성화된 경우 검색된 모든 레코드에 대해 *OnFilterRecord* 이벤트가 생성됩니다. 이벤트 핸들러는 각 레코드를 테스트한 다음 필터 조건에 맞는 레코드만 표시합니다. 데이터셋의 각 레코드에 대해 *OnFilterRecord* 이벤트가 생성되므로 성능에 악영향을 미치지 않도록 가능한 한 빈틈없이 이벤트 핸들러를 코딩해야 합니다.

런타임 시 필터 이벤트 핸들러 전환

개수에 상관없이 *OnFilterRecord* 이벤트 핸들러를 코딩하여 런타임 시 이 중 하나로 전환할 수 있습니다. 예를 들어, 다음 명령문은 *NewYorkFilter*라는 *OnFilterRecord* 이벤트 핸들러로 전환합니다.

```
DataSet1->OnFilterRecord = NewYorkFilter;
Refresh();
```

필터 옵션 설정

FilterOptions 속성을 사용하면 문자열 기반 필드를 비교하는 필터에서 부분 비교를 기반으로 하는 레코드를 받아들일 것인지와 문자열 비교 시 대소문자를 구분할 것인지를 지정할 수 있습니다. *FilterOptions*는 비어 있는 집합(기본값)이나 다음 값 중 하나 또는 두 개를 모두 포함할 수 있는 집합 속성입니다.

표 22.5 FilterOptions 값

값	의미
foCaseInsensitive	문자열을 비교할 때 대소문자를 무시합니다.
foNoPartialCompare	부분 문자열 일치를 비활성화합니다. 즉, 별표(*)로 끝나는 문자열을 찾지 않습니다.

예를 들어, 다음 명령문은 *State* 필드의 값을 비교할 때 대소문자를 무시하는 필터를 설정합니다.

```
TFilterOptions FilterOptions;
FilterOptions->Clear();
FilterOptions << foCaseInsensitive;
Table1->FilterOptions = FilterOptions;
Table1->Filter = “State = 'CA'”;
```

필터링된 데이터셋에서 레코드 탐색

필터링된 데이터셋의 레코드 사이를 탐색하는 데이터셋 메소드는 네 가지가 있습니다. 다음 표는 이러한 메소드를 나열하고 그러한 메소드의 작업을 설명한 것입니다.

표 22.6 필터링된 데이터셋 탐색 메소드

메소드	용도
FindFirst	현재 필터 기준에 일치하는 첫 번째 레코드로 이동합니다. 첫 번째로 일치하는 레코드에 대한 검색은 항상 필터링되지 않은 데이터셋의 첫 번째 레코드에서 시작합니다.
FindLast	현재 필터 기준에 일치하는 마지막 레코드로 이동합니다.
FindNext	필터링된 데이터셋의 현재 레코드에서 다음 레코드로 이동합니다.
FindPrior	필터링된 데이터셋의 현재 레코드에서 이전 레코드로 이동합니다.

예를 들어, 다음 명령문은 데이터셋에서 첫 번째로 필터링된 레코드를 찾습니다.

```
DataSet1->FindFirst();
```

사용하는 애플리케이션에서 *Filter* 속성을 설정하거나 *OnFilterRecord* 이벤트 핸들러를 만든다고 가정하면 이러한 메소드는 필터링이 현재 활성화되었는지에 상관없이 특정 레코드의 위치로 커서를 이동시킵니다. 필터링이 활성화되어 있지 않을 때 이러한 메소드를 호출하면 다음과 같은 작업이 발생합니다.

- 일시적으로 필터링을 활성화합니다.
- 일치하는 레코드를 찾으면 해당 레코드의 위치로 커서가 이동됩니다.
- 필터링을 비활성화합니다.

참고 필터링이 비활성화되었는데 *Filter* 속성을 설정하지 않았거나 *OnFilterRecord* 이벤트 핸들러를 만들면 이러한 메소드는 *First*(), *Last*(), *Next*() 및 *Prior*()와 같은 작업을 합니다.

모든 탐색 필터 메소드는 일치하는 레코드를 찾으면 해당 레코드의 위치로 커서를 이동하고 해당 레코드를 현재 레코드로 만든 다음 **true**를 반환합니다. 일치하는 레코드를 찾지 못하면 커서 위치는 변경되지 않으며 이러한 메소드에서는 **false**를 반환합니다. *Found* 속성의 상태를 검사하여 이러한 호출을 랩핑한 다음 *Found*가 **true**일 때만 동작을 취합니다. 예를 들어, 커서가 이미 데이터셋의 마지막 일치하는 레코드에 있는데 *FindNext*를 호출하면 메소드에서 **false**를 반환하고 현재 레코드는 변경되지 않습니다.

데이터 수정

읽기 전용인 *CanModify* 속성이 **true**인 경우 다음 데이터셋 메소드를 사용하여 데이터를 삽입, 업데이트 및 삭제할 수 있습니다. 데이터셋이 단방향이 아니거나, 데이터셋의 원본으로 사용하는 데이터베이스가 쓰기 및 읽기 권한을 허용하지 않거나, 일부 다른 요인 (일부 데이터셋의 *ReadOnly* 속성 또는 *TQuery* 컴포넌트의 *RequestLive* 속성)이 없다면, *CanModify*는 **true**입니다.

표 22.7 데이터 삽입, 업데이트 및 삭제를 위한 데이터셋 메소드

메소드	설명
Edit	데이터셋이 아직 *dsEdit*이나 *dsInsert* 상태에 없으면 해당 데이터셋을 *dsEdit* 상태에 둡니다.
Append	보류 중인 데이터를 포스트하고 현재 레코드를 데이터셋의 끝으로 이동한 다음 데이터셋을 *dsInsert* 상태에 둡니다.
Insert	보류 중인 데이터를 포스트하고 데이터셋을 *dsInsert* 상태에 둡니다.
Post	새 레코드나 대체된 레코드를 데이터베이스로 포스트하려고 시도합니다. 성공하면 데이터셋은 *dsBrowse* 상태에 놓이고 성공하지 못하면 데이터셋은 현재 상태로 남습니다.
Cancel	현재 작업을 취소하고 데이터셋을 *dsBrowse* 상태에 둡니다.
Delete	현재 레코드를 삭제하고 데이터셋을 *dsBrowse* 상태에 둡니다.

레코드 편집

애플리케이션에서 레코드를 수정하기 전에 데이터셋은 *dsEdit* 모드에 있어야 합니다. 데이터셋의 읽기 전용 *CanModify* 속성이 **true**이면 코드에서 *Edit* 메소드를 사용하여 데이터셋을 *dsEdit* 모드에 둘 수 있습니다.

데이터셋이 *dsEdit* 모드로 전환되면 데이터셋에서는 먼저 *BeforeEdit* 이벤트를 받습니다. 편집 모드로의 전환이 성공적으로 완료되면 데이터셋은 *AfterEdit* 이벤트를 받습니다. 일반적으로 이러한 이벤트는 데이터셋의 현재 상태를 표시하는 사용자 인터페이스를 업데이트할 때 사용됩니다. 데이터셋이 몇 가지 이유로 편집 모드에 있을 수 없으면 문제를 일으킨 사용자에게 알리거나 데이터셋이 편집 모드로 들어가지 못하도록 상황을 수정할 수 있는 *OnEditError* 이벤트가 발생합니다.

애플리케이션의 폼에서 일부 데이터 인식 컨트롤은 다음과 같은 경우에 데이터셋을 자동으로 *dsEdit* 상태에 둡니다.

- 컨트롤의 *ReadOnly* 속성이 **false**(기본값)인 경우입니다.
- 컨트롤에 대한 데이터 소스의 *AutoEdit* 속성이 **true**이고
- 데이터셋의 *CanModify*가 *true*인 경우입니다.

참고 데이터셋이 *dsEdit* 상태에 있더라도 애플리케이션 사용자가 적절한 SQL 액세스 권한을 갖고 있지 않으면 SQL 기반 데이터베이스에 대한 레코드 편집에 실패할 수도 있습니다.

일단 데이터셋이 *dsEdit* 모드에 있으면 사용자는 폼의 데이터 인식 컨트롤에 표시되는 현재 레코드의 필드 값을 수정할 수 있습니다. 사용자가 그리드의 다른 레코드로 이동하는 등, 현재 레코드를 변경하는 동작을 실행하면 편집이 활성화되어 있는 데이터 인식 컨트롤에서는 자동으로 *Post*를 호출합니다.

폼에 탐색기 컴포넌트가 있으면 사용자는 탐색기의 Cancel 버튼을 클릭하여 편집을 취소할 수 있습니다. 편집을 취소하면 데이터셋은 *dsBrowse* 상태로 돌아갑니다.

코드에서는 적절한 메소드를 호출하여 편집을 기록하거나 취소해야 합니다. 변경 내용을 기록하려면 *Post*를 호출합니다. 변경 내용을 취소하려면 *Cancel*을 호출합니다. 코드에서 *Edit*와 *Post*는 종종 함께 사용됩니다. 예를 들면, 다음과 같습니다.

```
Table1->Edit();
Table1->FieldValues[“CustNo”] = 1234;
Table1->Post();
```

앞의 예제에서 코드의 첫 번째 줄은 데이터셋을 *dsEdit* 모드에 둡니다. 코드의 그 다음 줄은 1234라는 숫자를 현재 레코드의 *CustNo* 필드에 할당합니다. 마지막으로 마지막 줄은 수정된 레코드를 기록하거나 포스트합니다. 업데이트를 캐싱하지 않은 경우 포스트하면 변경 내용이 다시 데이터베이스로 기록됩니다. 업데이트를 캐싱하면 변경 내용이 임시 버퍼에 기록되어 데이터셋의 *ApplyUpdates* 메소드를 호출할 때까지 버퍼에 남아 있습니다.

새 레코드 추가

애플리케이션에서 새 레코드를 추가할 수 있으려면 데이터셋이 먼저 *dsInsert* 모드에 있어야 합니다. 데이터셋의 읽기 전용 *CanModify* 속성이 **true**이면 코드에서 *Insert*나 *Append* 메소드를 사용하여 데이터셋을 *dsInsert* 모드에 둘 수 있습니다.

데이터셋이 *dsInsert* 모드로 전환되면 먼저 *BeforeInsert* 이벤트를 받습니다. 삽입 모드로의 전환이 성공적으로 완료되면 데이터셋에서는 먼저 *OnNewRecord* 이벤트 핸들러를 받은 다음 *AfterInsert* 이벤트를 받습니다. 예를 들어, 이러한 이벤트를 사용하여 다음과 같이 새로 삽입한 레코드에 초기 값을 제공할 수 있습니다.

```
void __fastcall TForm1::OrdersTableNewRecord(TDataSet *DataSet)
{
  DataSet->FieldByName("OrderDate")->AsDateTime = Date();
}
```

애플리케이션의 폼에서 데이터 인식 그리드 및 탐색기 컨트롤은 다음과 같은 경우에 데이터셋을 *dsInsert* 상태에 둡니다.

- 컨트롤의 *ReadOnly* 속성이 **false**(기본값)이고
- 데이터셋의 *CanModify*가 **true**인 경우입니다.

참고 데이터셋이 *dsInsert* 상태에 있더라도 애플리케이션 사용자가 적절한 SQL 액세스 권한을 갖고 있지 않으면 SQL 기반 데이터베이스에 대한 레코드 추가에 실패할 수도 있습니다.

일단 데이터셋이 *dsInsert* 모드에 있으면 사용자나 애플리케이션에서는 새 레코드와 연결된 필드에 값을 입력할 수 있습니다. 그리드와 탐색 컨트롤을 제외하면 *Insert*와 *Append*는 사용자에게 뚜렷한 차이가 없습니다. *Insert*를 호출하면 그리드에서 현재 레코드 위에 비어 있는 행이 표시됩니다. *Append*를 호출하면 그리드가 데이터셋의 마지막 레코드로 스크롤되어 그리드의 마지막에 비어 있는 행이 표시되며, 데이터셋과 연결된 탐색기 컴포넌트에서 *Next*와 *Last* 버튼이 흐리게 표시됩니다.

사용자가 그리드의 다른 레코드로 이동하는 등, 현재 레코드를 변경하는 동작을 실행하면 삽입이 활성화되어 있는 데이터 인식 컨트롤에서는 자동으로 *Post*를 호출합니다. 그렇지 않은 경우에는 코드에서 *Post*를 호출해야 합니다.

*Post*는 새 레코드를 데이터베이스에 기록합니다. 또는 업데이터를 캐싱하면 *Post*는 레코드를 인메모리 캐시에 기록합니다. 캐싱된 삽입 및 추가 내용을 데이터베이스에 기록하려면 데이터셋의 *ApplyUpdates* 메소드를 호출합니다.

레코드 삽입

*Insert*에서는 현재 레코드 앞에 비어 있는 새로운 레코드를 열고, 비어 있는 레코드를 현재 레코드로 만들어 사용자나 애플리케이션 코드에서 해당 레코드의 필드 값을 입력할 수 있게 만듭니다.

애플리케이션에서 *Post*를 호출하거나, 캐싱된 업데이트를 사용할 때 *ApplyUpdates*를 호출하면 새로 삽입된 레코드가 다음과 같은 세 가지 방법 중 하나로 데이터베이스에 기록됩니다.

- 인덱싱된 Paradox와 dBASE 테이블의 경우 레코드의 인덱스를 기반으로 한 위치에서 레코드가 데이터셋에 삽입됩니다.
- 인덱싱되지 않은 Paradox와 dBASE 테이블의 경우 레코드는 현재 위치에서 데이터셋으로 삽입됩니다.
- SQL 데이터베이스의 경우 삽입할 때의 실제 위치는 구현 방법에 따라 다릅니다. 테이블이 인덱싱되어 있으면 인덱스는 새 레코드 정보로 업데이트됩니다.

레코드 추가

*Append*는 데이터셋의 끝에서 비어 있는 새로운 레코드를 열고, 비어 있는 레코드를 현재 레코드로 만들어 사용자나 애플리케이션 코드에서 해당 레코드의 필드 값을 입력할 수 있게 만듭니다.

애플리케이션에서 *Post*를 호출하거나, 캐싱된 업데이트를 사용할 때 *ApplyUpdates*를 호출하면 새로 삽입된 레코드가 다음과 같은 세 가지 방법 중 하나로 데이터베이스에 기록됩니다.

- 인덱싱된 Paradox와 dBASE 테이블의 경우 레코드의 인덱스를 기반으로 한 위치에서 레코드가 데이터셋에 삽입됩니다.
- 인덱싱되지 않은 Paradox와 dBASE 테이블의 경우 레코드는 데이터셋의 끝에 삽입됩니다.
- SQL 데이터베이스의 경우 추가할 때의 실제 위치는 구현 방법에 따라 다릅니다. 테이블이 인덱싱되어 있으면 인덱스는 새 레코드 정보로 업데이트됩니다.

레코드 삭제

활성 데이터셋에서 현재 레코드를 삭제하려면 *Delete* 메소드를 호출합니다. *Delete* 메소드가 호출되면 다음 작업이 수행됩니다.

- 데이터셋에서 *BeforeDelete* 이벤트를 받습니다.
- 데이터셋에서 현재 레코드를 삭제하려고 시도합니다.
- 데이터셋이 *dsBrowse* 상태로 돌아갑니다.
- 데이터셋에서 *AfterDelete* 이벤트를 받습니다.

BeforeDelete 이벤트 핸들러에서 삭제할 수 없게 하려면 다음과 같이 전역 *Abort* 프로시저를 호출할 수 있습니다.

```
void __fastcall TForm1::TableBeforeDelete (TDataSet *Dataset)
{
  if (MessageBox(0, "Delete This Record?", "CONFIRM", MB_YESNO) != IDYES)
    Abort();
}
```

*Delete*가 실패하면 *OnDeleteError* 이벤트를 생성합니다. *OnDeleteError* 이벤트 핸들러에서 문제를 해결할 수 없으면 데이터셋은 *dsEdit* 상태로 남습니다. *Delete*가 성공하면 데이터셋이 *dsBrowse* 상태로 되돌아가고 삭제된 레코드 뒤의 레코드는 현재 레코드가 됩니다.

업데이트를 캐싱하는 경우 *ApplyUpdates*를 호출할 때까지 삭제된 레코드는 원본으로 사용하는 데이터베이스 테이블에서 제거되지 않습니다.

탐색기 컴포넌트를 폼에 제공하면 사용자는 탐색기의 Delete 버튼을 클릭하여 현재 레코드를 삭제할 수 있습니다. 코드에서는 *Delete*를 명시적으로 호출하여 현재 레코드를 제거해야 합니다.

데이터 포스트

레코드 편집을 마친 다음에는 *Post* 메소드를 호출하여 변경 내용을 기록해야 합니다. *Post* 메소드는 데이터셋의 상태와 업데이트 캐싱 여부에 따라 다르게 동작합니다.

- 업데이트를 캐싱하지 않는 경우 데이터셋이 *dsEdit*이나 *dsInsert* 상태에 있으면 *Post*는 현재 레코드를 데이터베이스에 기록하고 데이터셋을 *dsBrowse* 상태로 되돌립니다.
- 업데이트를 캐싱하는 경우 데이터셋이 *dsEdit*이나 *dsInsert* 상태에 있으면 *Post*는 현재 레코드를 내부 캐시에 기록하고 데이터셋을 *dsBrowse* 상태로 되돌립니다. *ApplyUpdates*를 호출할 때까지 편집 내용은 데이터베이스에 기록되지 않습니다.
- 데이터셋이 *dsSetKey* 상태에 있으면 *Post*는 데이터셋을 *dsBrowse* 상태로 되돌립니다.

데이터셋의 초기 상태에 상관없이 현재 변경 내용을 기록하기 전이나 기록한 다음에 *Post*는 *BeforePost*와 *AfterPost* 이벤트를 생성합니다. 이러한 이벤트를 사용하여 사용자 인터페이스를 업데이트하거나 데이터셋에서 *Abort* 프로시저를 호출하여 변경 내용을 포스트하지 못하게 할 수 있습니다. *Post*에 대한 호출에 실패하면 데이터셋에서는 *OnPostError* 이벤트를 받으므로 문제의 사용자에게 알리거나 문제를 해결하려고 시도할 수 있습니다.

포스트는 명시적으로 이루어지거나 다른 프로시저의 일부로 암시적으로 이루어질 수 있습니다. 애플리케이션에서 현재 레코드를 떠나면 *Post*는 암시적으로 호출됩니다. *First*, *Next*, *Prior* 및 *Last* 메소드를 호출하면 테이블이 *dsEdit*이나 *dsInsert* 모드에 있는 경우 *Post*를 수행합니다. 또한 *Append*와 *Insert* 메소드는 보류 중인 데이터를 명시적으로 포스트합니다.

경고 *Close* 메소드는 *Post*를 암시적으로 호출하지 않습니다. 보류 중인 편집을 명시적으로 포스트하려면 *BeforeClose* 이벤트를 사용합니다.

변경 취소

아직 *Post*를 직접 또는 간접으로 호출하지 않았으면 현재 레코드에 대한 변경 내용을 언제든지 애플리케이션에서 취소할 수 있습니다. 예를 들어, 데이터셋이 *dsEdit* 모드에 있고 사용자가 하나 이상의 필드에서 데이터를 변경한 경우 애플리케이션에서는 데이터셋의 *Cancel* 메소드를 호출하여 해당 레코드를 원래 값으로 되돌릴 수 있습니다. *Cancel*을 호출하면 데이터셋을 항상 *dsBrowse* 상태로 되돌립니다.

애플리케이션에서 *Cancel*을 호출했을 때 데이터셋이 *dsEdit*이나 *dsInsert* 모드에 있었으면 현재 레코드가 원래 값으로 복구되기 전에 그리고 복구된 다음 데이터셋은 *BeforeCancel*과 *AfterCancel* 이벤트를 받습니다.

폼에서는, 데이터셋과 연결된 탐색기 컴포넌트에 Cancel 버튼을 포함하여 사용자가 편집이나 삽입, 추가 작업을 취소할 수 있게 해주거나 개발자 고유의 Cancel 버튼에 대한 코드를 해당 폼에 제공할 수 있습니다.

전체 레코드 수정

폼에서는, 그리드를 제외한 모든 데이터 인식 컨트롤과 탐색기는 레코드의 단일 필드에 대한 액세스를 제공합니다.

하지만 코드에서는 데이터셋의 원본으로 사용하는 데이터베이스 테이블의 구조가 안정적이고 변경되지 않는다는 가정 하에 전체 레코드 구조를 사용하는 다음 메소드를 사용할 수 있습니다. 다음 표는 전체 레코드의 개별 필드가 아닌 전체 레코드를 작업할 때 사용할 수 있는 메소드를 요약한 것입니다.

표 22.8 전체 레코드를 사용하는 메소드

메소드	설명
AppendRecord([값의 배열])	지정한 열 값을 갖는 레코드를 테이블 끝에 추가합니다. *Append*와 유사합니다. 암시적 *Post*를 수행합니다.
InsertRecord([값의 배열])	지정한 값을 레코드로 테이블의 현재 커서 위치 앞에 삽입합니다. *Insert*와 유사합니다. 암시적 *Post*를 수행합니다.
SetFields([값의 배열])	해당 필드의 값을 설정합니다 . *TField* 에 값을 할당하는 것과 유사합니다. 애플리케이션에서는 명시적 *Post*를 수행해야 합니다.

이러한 메소드는 각 값이 원본으로 사용하는 데이터셋의 열에 해당하는, 값의 배열을 인수로 취합니다. 이러한 배열을 만들려면 ARRAYOFCONST 매크로를 사용합니다. 값은 리터럴이나 변수, NULL이 될 수 있습니다. 인수에서 값의 개수가 데이터셋의 열 개수보다 작으면 나머지 값은 NULL로 간주됩니다.

인덱싱되지 않은 데이터셋의 경우 *AppendRecord*는 레코드를 데이터셋의 끝에 추가하고 *InsertRecord*는 레코드를 현재 커서 위치 다음에 삽입합니다. 인덱싱된 데이터셋의 경우 두 개의 메소드는 모두 인덱스를 기반으로 테이블의 올바른 위치에 레코드를 둡니다. 두 가지 경우 모두에서 메소드는 커서를 레코드의 위치로 이동합니다.

*SetFields*는 매개변수 배열에 지정된 값을 데이터셋의 필드에 할당합니다. *SetFields*를 사용하려면 애플리케이션에서는 먼저 *Edit*을 호출하여 데이터셋을 *dsEdit* 모드에 두어야 합니다. 변경 내용을 현재 레코드에 적용하려면 *Post*를 수행해야 합니다.

*SetFields*를 사용하여 기존 레코드의 모든 필드가 아닌 일부 필드를 수정하는 경우 변경하지 않을 필드에는 NULL 값을 전달할 수 있습니다. 레코드의 모든 필드에 충분한 값을 제공하지 않으면 SetFields에서는 필드에 NULL 값을 할당합니다. NULL 값은 해당 필드에 있던 기존 값을 오버라이드합니다.

예를 들어, 데이터베이스에 Name, Capital, Continent, Area 및 Population 열이 있는 COUNTRY 테이블이 있다고 가정합니다. *CountryTable*이라는 *TTable* 컴포넌트가 COUNTRY 테이블에 연결되어 있는 경우 다음 명령문은 COUNTRY 테이블에 레코드를 삽입할 것입니다.

```
CountryTable->InsertRecord(ARRAYOFCONST((“Japan”, “Tokyo”, “Asia”)));
```

이 명령문에서는 Area와 Population의 값은 지정하지 않으므로 NULL 값이 삽입됩니다. 테이블은 Name으로 인덱싱되어 있으므로 해당 명령문에서는 "Japan"의 알파벳순 데이터 정렬을 기반으로 레코드를 삽입할 것입니다.

레코드를 업데이트하려면 애플리케이션에서 다음 코드를 사용할 수 있습니다.

```
TLocateOptions SearchOptions;
SearchOptions->Clear();
SearchOptions << loCaseInsensitive;
if (CountryTable->Locate("Name", "Japan", SearchOptions))
{
  CountryTable->Edit();
  CountryTable->SetFields(ARRAYOFCONST(((void *)NULL, (void *)NULL, (void
*)NULL,
        344567, 164700000)));
  CountryTable->Post();
}
```

이 코드에서는 Area와 Population 필드에 값을 할당한 다음 데이터베이스에 이들 값을 포스트합니다. 세 개의 NULL 포인터는 현재 내용을 유지하기 위해 처음의 세 열에 대한 위치 표시자 역할을 합니다.

경고 몇 가지 필드 값을 그대로 두기 위해 *SetFields*와 함께 NULL 포인터를 사용하려면 NULL을 void *로 변환해야 합니다. 변환하지 않고 NULL을 매개변수로 사용하면 필드를 비어 있는 값으로 설정하게 됩니다.

필드 계산

Fields Editor를 사용하여 데이터셋의 계산된 필드를 정의할 수 있습니다. 데이터셋에 계산된 필드가 포함되어 있으면 해당 필드 값을 계산하는 코드를 *OnCalcFields* 이벤트 핸들러에 입력해야 합니다. Fields Editor를 사용하여 계산된 필드를 정의하는 방법에 대한 자세한 내용은 23-7페이지의 "계산된 필드 정의"를 참조하십시오.

AutoCalcFields 속성은 *OnCalcFields*가 호출되는 시기를 결정합니다. *AutoCalcFields*가 **true**이면 다음과 같은 경우 *OnCalcFields*가 호출됩니다.

- 데이터셋이 열린 경우
- 데이터셋이 편집 모드로 들어간 경우
- 데이터베이스에서 레코드를 검색한 경우
- 데이터 인식 그리드 컨트롤에서 한 열에서 다른 열로 포커스를 이동하거나 비주얼 (visual) 컴포넌트에서 다른 컴포넌트로 포커스를 이동하고 현재 레코드가 수정된 경우

*AutoCalcFields*가 **false**인 경우 레코드 내에서 개별 필드를 편집하면(위에 있는 네 번째 조건) *OnCalcFields*는 호출되지 않습니다.

주의 *OnCalcFields*는 자주 호출되므로 이에 대한 코드를 작성할 때는 간단하게 해야 합니다. 그리고 *AutoCalcFields*가 **true**인 경우 *OnCalcFields*에서 데이터셋 또는 마스터/디테일 관계의 일부인 경우 연결된 데이터셋을 수정하면 재귀될 수 있으므로 수정하는 작업을 수행하지 않습니다. 예를 들어, *OnCalcFields*에서 *Post*를 수행하고 *AutoCalcFields*가 **true**인 경우 *OnCalcFields*는 다른 *Post*를 일으키면서 다시 호출되는 식으로 진행됩니다.

*OnCalcFields*가 실행되면 데이터셋은 *dsCalcFields* 모드로 들어갑니다. 이러한 상태는 핸들러에서 수정하려는 계산된 필드 이외의 레코드에 대한 수정이나 추가를 막습니다. 다른 수정을 막는 이유는 *OnCalcFields*에서 계산된 필드 값을 구하기 위해 다른 필드의 값을 사용하기 때문입니다. 따라서 이러한 필드를 변경하면 계산된 필드에 할당된 값을 무효화할 수 있습니다. *OnCalcFields*가 완료된 다음 데이터셋은 *dsBrowse* 상태로 돌아갑니다.

데이터셋의 타입

22-2페이지의 "TDataSet 자손 사용"에서는 *TDataSet* 자손이 데이터에 액세스할 때 사용하는 메소드에 따라 TDataSet 자손을 분류합니다. *TDataSet* 자손을 분류하는 다른 유용한 방법은 TDataSet 자손이 나타내는 서버 타입을 고려하는 것입니다. 이러한 방법을 살펴보면 데이터셋에는 다음과 같은 세 개의 기본 클래스가 있습니다.

- **테이블 타입 데이터셋:** 테이블 타입 데이터셋은 모든 행과 열을 비롯하여 데이터베이스 서버의 단일 테이블을 나타냅니다. 테이블 타입 데이터셋에는 *TTable*, *TADOTable*, *TSQLTable* 및 *TIBTable*이 포함됩니다.

  테이블 타입 데이터셋을 사용하면 서버에 정의되어 있는 인덱스를 이용할 수 있습니다. 데이터베이스 테이블과 데이터셋은 일대일 대응을 하기 때문에 데이터베이스 테이블에 대해 정의되어 있는 서버 인덱스를 사용할 수 있습니다. 인덱스를 사용하면 애플리케이션에서 테이블의 레코드를 정렬할 수 있고 검색 및 조회 속도를 높일 수 있으며 마스터/디테일 관계의 기초를 만들 수 있습니다 . 그리고 일부 테이블 타입 데이터셋에서도 데이터셋과 데이터베이스 테이블 사이의 일대일 관계를 이용하므로 데이터베이스 테이블 생성 및 삭제와 같은 테이블 수준 작업을 수행할 수 있게 해줍니다.

- **쿼리 타입 데이터셋:** 쿼리 타입 데이터셋에서는 단일 SQL 명령이나 쿼리를 나타냅니다. 쿼리는 명령(일반적으로 SELECT 문)을 실행할 때 결과 집합을 나타내거나 레코드를 반환하지 않는 명령(예를 들어, UPDATE 문)을 실행할 수 있습니다. 쿼리 타입 데이터셋에는 *TQuery*, *TADOQuery*, *TSQLQuery* 및 *TIBQuery*가 포함됩니다.

  쿼리 타입 데이터셋을 효과적으로 사용하려면 SQL-92 표준에 대한 제한 사항 및 확장을 비롯하여 SQL 및 서버의 SQL 구현을 잘 알아야 합니다. SQL을 잘 모른다면 SQL을 깊이 있게 다룬 서드파티 서적을 구입해 보는 것도 좋습니다. 추천할 만한 서적은 Jim Melton과 Alan R. Simpson이 공저하고 Morgan Kaufmann Publishers에서 발행된 *Understanding the New SQL: A Complete Guide*입니다.

- **내장 프로시저 타입 데이터셋:** 내장 프로시저 타입 데이터셋은 데이터베이스 서버에서 내장 프로시저를 나타냅니다. 내장 프로시저 타입 데이터셋에는 *TStoredProc*, *TADOStoredProc*, *TSQLStoredProc* 및 *TIBStoredProc*이 포함됩니다.

  내장 프로시저는 프로시저에 작성된 독립적인 프로그램이며, 사용 중인 데이터베이스 시스템에 따른 언어를 실행합니다. 일반적으로 내장 프로시저는 반복된 데이터베이스 관련 작업을 처리하며, 많은 양의 레코드를 작업하거나 집계 또는 수식 함수를 사용하는 작업에서 특히 유용합니다. 내장 프로시저를 사용할 경우, 일반적으로 다음과 같은 이유에서 데이터베이스 애플리케이션의 성능이 향상됩니다.

- 대개는 서버의 높은 처리 능력과 속도를 이용할 수 있습니다.
- 서버에서 처리하도록 하여 네트워크 트래픽을 줄입니다.

내장 프로시저는 데이터를 반환할 수도 있고 반환하지 않을 수도 있습니다. 데이터를 반환하는 내장 프로시저는 커서(SELECT 쿼리의 결과와 유사), 다중 커서(여러 데이터셋을 효과적으로 반환)로 반환하거나 출력 매개변수에 데이터를 반환합니다. 이러한 차이는 부분적으로 서버에 따라 다르게 나타납니다. 일부 서버는 내장 프로시저의 데이터 반환을 허용하지 않거나 출력 매개변수만 허용합니다. 내장 프로시저를 아예 지원하지 않는 서버도 있습니다. 서버 설명서를 참조하여 사용 가능한 작업을 확인하십시오.

참고 대부분의 서버에서는 내장 프로시저를 사용할 수 있는 SQL에 대한 확장 기능을 제공하므로 쿼리 타입 데이터셋을 사용하여 내장 프로시저를 실행할 수 있습니다. 하지만 서버별로 내장 프로시저에 대한 고유 구문을 사용합니다. 내장 프로시저 타입 데이터셋 대신 쿼리 타입 데이터셋 사용을 선택한 경우 필요한 구문은 서버 설명서를 참조하십시오.

이러한 세 개의 범주에 확실하게 해당하는 데이터셋 이외에 *TDataSet*에는 다음과 같이 두 개 이상의 범주에 해당하는 몇 가지 자손이 있습니다.

- *TADODataSet*과 *TSQLDataSet*에는 이들이 테이블이나 쿼리, 내장 프로시저 중 어느 것을 나타내는지 지정할 수 있게 해주는 *CommandType* 속성이 있습니다. *TADODataSet*을 사용하면 테이블 타입 데이터셋과 같은 인덱스를 지정할 수 있게 해주지만 속성 및 메소드 이름은 쿼리 타입 데이터셋과 아주 유사합니다.
- *TClientDataSet*은 다른 데이터셋의 데이터를 나타냅니다. 따라서 테이블이나 쿼리, 내장 프로시저를 나타낼 수 있습니다. *TClientDataSet*은 인덱스를 지원하기 때문에 대개는 테이블 타입 데이터셋처럼 동작합니다. 하지만 쿼리와 내장 프로시저의 일부 기능, 즉, 매개변수 관리 기능 및 결과 집합을 검색하지 않고 실행하는 기능도 갖습니다.
- 기타 다른 클라이언트 데이터셋(*TBDEClientDataSet*과 *TSQLClientDataSet*)에는 이들이 테이블이나 쿼리, 내장 프로시저 중 어느 것을 나타내는지 지정할 수 있는 *CommandType* 속성이 있습니다. 매개변수 지원, 인덱스 및 결과 집합을 검색하지 않고 실행하는 기능을 가지며 속성 및 메소드 이름은 *TClientDataSet*과 유사합니다.
- *TIBDataSet*은 쿼리와 내장 프로시저를 모두 나타낼 수 있습니다. 사실상, 이것은 각각에 대한 속성을 가지고 여러 개의 쿼리와 내장 프로시저를 동시에 나타낼 수 있습니다.

테이블 타입 데이터셋 사용

다음과 같은 방법으로 테이블 타입 데이터셋을 사용할 수 있습니다.

1. 해당 데이터셋 컴포넌트를 데이터 모듈이나 폼에 배치하고 *Name* 속성을 애플리케이션에 적절한 고유 값으로 설정합니다.
2. 사용할 테이블을 포함하는 데이터베이스 서버를 식별합니다. 테이블 타입 데이터셋별로 이러한 작업을 다르게 수행하지만 일반적으로 다음과 같이 데이터베이스 연결 컴포넌트를 지정할 수 있습니다.

- *TTable*에서는 *DatabaseName* 속성을 사용하여 *TDatabase* 컴포넌트나 BDE 알리아스를 지정합니다.
- *TADOTable*에서는 *Connection* 속성을 사용하여 *TADOConnection* 컴포넌트를 지정합니다.
- *TSQLTable*에서는 *SQLConnection* 속성을 사용하여 *TSQLConnection* 컴포넌트를 지정합니다.
- *TIBTable*에서는 *Database* 속성을 사용하여 *TIBConnection* 컴포넌트를 지정합니다.

데이터베이스 연결 컴포넌트에 대한 자세한 내용은 21장, "데이터베이스에 연결"을 참조하십시오.

3. *TableName* 속성을 데이터베이스의 테이블 이름으로 설정합니다. 데이터베이스 연결 컴포넌트를 이미 식별하였으면 드롭다운 리스트에서 테이블을 선택할 수 있습니다.
4. 데이터 소스 컴포넌트를 데이터 모듈이나 폼에 두고 *DataSet* 속성을 데이터셋의 이름으로 설정합니다 . 데이터셋의 결과 집합을 표시할 데이터 인식 컴포넌트에 전달할 때 데이터 소스 컴포넌트가 사용됩니다.

테이블 타입 데이터셋의 장점

테이블 타입 데이터셋 사용의 주요 장점은 인덱스의 사용입니다. 인덱스를 사용하면 애플리케이션에서 다음 작업을 할 수 있습니다.

- 데이터셋에서 레코드를 정렬합니다.
- 레코드를 빨리 찾습니다.
- 보이는 레코드를 제한합니다.
- 마스터/디테일 관계를 설정합니다.

그리고 테이블 타입 데이터셋과 데이터베이스 테이블의 일대일 관계는 다음과 같은 경우에 데이터셋과 테이블을 많이 사용하게 해줍니다.

- 테이블의 읽기/쓰기 액세스 제어
- 테이블 생성 및 삭제
- 테이블 비우기
- 테이블 동기화

인덱스를 사용하여 레코드 정렬

인덱스는 테이블에서 레코드 표시 순서를 결정합니다. 일반적으로 레코드는 주 인덱스 또는 디폴트 인덱스를 기반으로 오름차순으로 표시됩니다. 이러한 디폴트 동작에는 애플리케이션 간섭이 필요하지 않습니다. 하지만 다른 정렬 순서를 사용하려면 다음 방법 중 하나를 지정해야 합니다.

- 대체 인덱스
- 정렬할 열의 리스트(SQL 기반이 아닌 서버에서는 사용할 수 없음)

인덱스를 사용하면 테이블의 데이터를 다양한 순서로 나타낼 수 있습니다. SQL 기반 테이블의 경우 이러한 정렬 순서는 테이블 레코드를 가져오는 쿼리에서 ORDER BY 절을 생성하는 인덱스를 사용하여 구현됩니다. Paradox 및 dBASE 테이블과 같은 다른 테이블에서는 원하는 순서로 레코드를 나타내는 데이터 액세스 메커니즘에서 인덱스를 사용합니다.

인덱스 관련 정보 얻기

애플리케이션에서는 모든 테이블 타입 데이터셋의 서버 정의 인덱스에 대한 정보를 얻을 수 있습니다. 데이터셋에서 사용할 수 있는 인덱스 리스트를 얻으려면 *GetIndexNames* 메소드를 호출합니다. *GetIndexNames*는 유효한 인덱스 이름으로 문자열 리스트를 만듭니다. 예를 들어, 다음 코드는 *CustomersTable* 데이터셋에 대해 정의한 모든 인덱스 이름으로 리스트 박스를 채웁니다.

```
CustomersTable->GetIndexNames(ListBox1->Items);
```

참고 Paradox 테이블의 경우 기본 인덱스는 이름이 지정되어 있지 않으므로 *GetIndexNames*에 의해 반환되지 않습니다. 하지만 대체 인덱스를 사용한 다음 *IndexName* 속성을 빈 문자열로 설정하여 인덱스를 다시 Paradox 테이블의 기본 인덱스로 변경할 수 있습니다.

현재 인덱스의 필드에 대한 정보를 얻으려면 다음 내용을 사용합니다.

- *IndexFieldCount* 속성을 사용하여 인덱스에 있는 열 수를 확인합니다.
- *IndexFields* 속성을 사용하여 인덱스를 구성하는 열의 필드 컴포넌트 리스트를 검사합니다.

다음 코드에서는 *IndexFieldCount*와 *IndexFields*를 사용하여 애플리케이션에서 열 이름 리스트를 반복하는 방법을 설명합니다.

```
AnsiString ListOfIndexFields[20];
for (int i = 0; i< CustomersTable->IndexFieldCount; i++)
  ListOfIndexFields[i] =CustomersTable->IndexFields[i]->FieldName;
```

참고 표현식 인덱스에 열려 있는 dBASE 테이블에서는 *IndexFieldCount*가 유효하지 않습니다.

IndexName을 사용하여 인덱스 지정

IndexName 속성을 사용하여 인덱스를 활성화합니다. 일단 활성화되면 인덱스는 데이터셋의 레코드 순서를 결정합니다. 그리고 마스터/디테일 연결이나 인덱스 기반 검색, 인덱스 기반 필터링의 기초로 사용할 수 있습니다.

인덱스를 활성화하려면 *IndexName* 속성을 인덱스 이름으로 설정합니다. 일부 데이터베이스 시스템에서는 기본 인덱스가 이름을 갖지 않습니다. 이러한 인덱스 중 하나를 활성화하려면 *IndexName*을 빈 문자열로 설정합니다.

디자인 타임 시 Object Inspector에서 해당 속성의 생략 부호 버튼을 클릭하여 사용 가능한 인덱스 리스트에서 인덱스를 선택할 수 있습니다. 런타임 시 *AnsiString* 리터럴이나 변수를 사용하여 *IndexName*을 설정합니다. *GetIndexNames* 메소드를 호출하여 사용 가능한 인덱스 리스트를 얻을 수 있습니다.

다음 코드는 *CustomersTable*의 리스트를 *CustDescending*으로 설정합니다.

```
CustomersTable->IndexName = “CustDescending”;
```

IndexFieldNames를 사용하여 인덱스 생성

원하는 정렬 순서를 구현하는 정의되어 있는 인덱스가 없으면 *IndexFieldNames* 속성을 사용하여 pseudo-index를 만들 수 있습니다.

참고 *IndexName*과 *IndexFieldNames*는 상호 배타적입니다. 한 속성을 설정하면 다른 속성에 설정된 값을 지웁니다.

*IndexFieldNames*의 값은 AnsiString입니다. 정렬 순서를 지정하려면 원하는 순서로 사용할 각 열 이름을 나열하고 각 이름을 세미콜론으로 구분해야 합니다. 정렬은 오름차순만 가능합니다. 정렬 시 대소문자 구분 여부는 서버의 기능에 따라 다릅니다. 자세한 내용은 서버 설명서를 참조하십시오.

다음 코드는 *LastName*을 기반으로 한 다음 *FirstName*을 기반으로 하여 *PhoneTable*의 정렬 순서를 설정합니다.

```
PhoneTable->IndexFieldNames = "LastName;FirstName";
```

참고 Paradox와 dBASE 테이블에서 *IndexFieldNames*를 사용하는 경우 데이터셋에서는 지정한 열을 사용하는 인덱스를 찾기 위해 시도합니다. 데이터셋에서 그러한 인덱스를 찾을 수 없으면 예외를 발생합니다.

인덱스를 사용하여 레코드 검색

*TDataSet*의 *Locate*와 *Lookup* 메소드를 사용하여 데이터셋을 검색할 수 있습니다. 하지만 인덱스를 명시적으로 사용하여 일부 테이블 타입 데이터셋에서는 *Locate*와 *Lookup* 메소드에서 제공하는 검색 성능을 개선할 수 있습니다.

ADO 데이터셋에서는 모두 *Seek* 메소드를 지원하는데 이 메소드는 현재 인덱스의 필드에 대한 필드 값 집합을 기반으로 레코드를 이동합니다. *Seek*을 사용하면 처음이나 마지막 일치하는 레코드를 기준으로 커서를 이동할 장소를 지정할 수 있습니다.

*TTable*과 모든 클라이언트 데이터셋 타입에서는 유사한 인덱스 기반 검색을 지원하지만 관련 메소드의 조합을 사용합니다. 다음 표는 인덱스 기반 검색을 지원하기 위해 *TTable*과 클라이언트 데이터셋에서 제공하는 6개의 관련 메소드를 요약한 것입니다.

표 22.9 인덱스 기반 검색 메소드

메소드	용도
EditKey	검색 키 버퍼의 현재 내용을 보관하고 데이터셋을 *dsSetKey* 상태로 두어 사용하는 애플리케이션에서 검색을 수행하기 전에 기존 검색 조건을 수정할 수 있습니다.
FindKey	*SetKey* 메소드와 *GotoKey* 메소드를 하나의 메소드로 결합합니다.
FindNearest	*SetKey* 메소드와 *GotoNearest* 메소드를 하나의 메소드로 결합합니다.
GotoKey	데이터셋에서 검색 조건에 정확하게 일치하는 첫 번째 레코드를 검색하고 찾은 레코드로 커서를 이동합니다.
GotoNearest	문자열 기반 필드에서 부분 키 값에 기반하여 가장 가깝게 일치하는 레코드를 검색하고 해당 레코드로 커서를 이동합니다.
SetKey	검색 키 버퍼를 지우고 테이블을 *dsSetKey* 상태로 두어 사용하는 애플리케이션에서 검색을 수행하기 전에 새로운 검색 조건을 지정할 수 있습니다.

*GotoKey*와 *FindKey*는 성공하면 커서를 일치하는 레코드로 이동하고 **true**를 반환하는 부울 함수입니다. 검색이 실패하는 경우 커서는 이동하지 않으며 **false**를 반환합니다.

*GotoNearest*와 *FindNearest*는 처음으로 찾은 정확한 일치 내용으로 또는 찾지 못한 경우에는 지정한 검색 기준보다 큰 첫 번째 레코드로 커서를 항상 다시 배치합니다.

Goto 메소드를 사용하여 검색 수행

Goto 메소드를 사용하여 검색을 수행하려면 다음과 같은 일반적인 단계를 따릅니다.

1. 검색에 사용할 인덱스를 지정합니다. 이 인덱스는 데이터셋에서 레코드를 정렬한 인덱스와 같은 인덱스입니다(22-25페이지의 "인덱스를 사용하여 레코드 정렬" 참조). 인덱스를 지정하려면 *IndexName*이나 *IndexFieldNames* 속성을 사용합니다.
2. 해당 데이터셋을 엽니다.
3. *SetKey* 메소드를 호출하여 데이터셋을 *dsSetKey* 상태에 둡니다.
4. *Fields* 속성에서 검색할 값을 지정합니다. *Fields*는 *TFields* 객체이며, 열에 해당하는 서수를 지정함으로써 액세스할 수 있는 필드 컴포넌트의 인덱스 리스트를 가지고 있습니다. 데이터셋의 첫 번째 열 번호는 0입니다.
5. *GotoKey*나 *GotoNearest*를 사용하여 찾은 첫 번째 일치하는 레코드를 검색하여 이동합니다.

예를 들어, 버튼의 *OnClick* 이벤트에 추가된 다음 코드는 *GotoKey* 메소드를 사용하여 인덱스의 첫 번째 필드의 값이 에디트 박스의 텍스트와 정확하게 일치하는 첫 번째 레코드로 이동합니다.

```
void __fastcall TSearchDemo::SearchExactClick(TObject *Sender)
{
  ClientDataSet1->SetKey();
  ClientDataSet1->Fields->Fields[0]->AsString = Edit1->Text;
  if (!ClientDataSet1->GotoKey())
    ShowMessage("Record not found");
}
```

*GotoNearest*도 유사합니다. 부분 필드 값에 가장 일치하는 내용을 검색합니다. 문자열 필드에서만 사용될 수 있습니다. 예를 들면, 다음과 같습니다.

```
Table1->SetKey();
Table1->Fields->Fields[0]->AsString = "Sm";
Table1->GotoNearest();
```

첫 인덱싱된 필드 값의 처음 두 글자가 "Sm"인 레코드가 있는 경우 이 레코드에 커서가 위치합니다. 해당 레코드가 없는 경우 커서의 위치는 변하지 않고 *GotoNearest*는 **false**를 반환합니다.

Find 메소드를 사용하여 검색 실행

Find 메소드는 검색할 키 필드 값을 지정하기 위해 명시적으로 데이터셋을 *dsSetKey* 상태로 두지 않아도 된다는 점을 제외하면 *Goto* 메소드와 동일합니다. *Find* 메소드를 사용하여 검색을 수행하려면 다음과 같은 일반적인 단계를 따릅니다.

1. 검색에 사용할 인덱스를 지정합니다. 이 인덱스는 데이터셋의 레코드를 정렬하는 것과 같은 인덱스입니다(22-25페이지의 "인덱스를 사용하여 레코드 정렬" 참조). 인덱스를 지정하려면 *IndexName*이나 *IndexFieldNames* 속성을 사용합니다.
2. 해당 데이터셋을 엽니다.
3. *FindKey*나 *FindNearest*를 사용하여 첫 번째 레코드나 가장 가까운 레코드를 검색하여 이동합니다. 두 메소드에는 하나의 매개변수, 즉 필드 값이 쉼표로 분리된 리스트가 필요하며 각 필드 값은 원본으로 사용하는 테이블의 인덱싱된 열에 해당합니다.

참고 *FindNearest*는 문자열 필드에 대해서만 사용할 수 있습니다.

성공적인 검색 이후 현재 레코드 지정

디폴트로, 검색이 성공하면 검색 조건에 일치하는 첫 번째 레코드로 커서가 이동합니다. 원한다면 *KeyExclusive* 속성을 **true**로 설정하여 첫 번째 일치하는 레코드의 다음 레코드에 커서를 둘 수 있습니다.

디폴트로 *KeyExclusive*는 **false**이며, 이는 검색이 성공하면 첫 번째 일치하는 레코드로 커서가 이동한다는 의미입니다.

부분 키 검색

데이터셋에 두 개 이상의 키 열이 있고 해당 키의 부분 집합에서 값을 검색하려면 *KeyFieldCount*를 검색할 열의 수로 설정합니다. 예를 들어, 데이터셋의 현재 인덱스에 세 개의 열이 있고 첫 번째 열만 사용하여 값을 검색하려면 *KeyFieldCount*를 1로 설정합니다.

여러 열 키를 가진 테이블 타입 데이터셋의 경우 첫 번째 열부터 시작하여 이웃하는 열의 값만을 검색할 수 있습니다. 예를 들어, 3열 키의 경우 첫 번째 열에서 검색하거나, 첫 번째와 두 번째 열에서 검색하거나, 첫 번째, 두 번째, 세 번째 열에서 값을 검색할 수 있지만 첫 번째와 세 번째 열에서는 검색할 수 없습니다.

검색 반복 또는 확장

*SetKey*나 *FindKey*를 호출할 때마다 메소드에서 *Fields* 속성의 이전 값을 지웁니다. 이전에 설정한 필드를 사용하여 검색을 반복하거나 검색에 사용되는 필드에 추가하려면 *SetKey*와 *FindKey* 대신 *EditKey*를 호출합니다.

예를 들어, "CityIndex" 인덱스의 City 필드를 기반으로 Employee 테이블 검색을 이미 실행했다고 가정합니다. 계속해서 "CityIndex"는 *City* 필드와 *Company* 필드를 모두 포함한다고 가정합니다. 특정 도시에 있는 특정 회사 이름으로 레코드를 찾으려면 다음과 같은 코드를 사용합니다.

```
Employee->KeyFieldCount = 2;
Employee->EditKey();
Employee->FieldValues[“Company”] = Variant(Edit2->Text);
Employee->GotoNearest();
```

범위를 사용하여 레코드 제한

필터를 사용하여 특정 데이터셋의 부분 집합 데이터를 일시적으로 보고 편집할 수 있습니다 (22-12페이지의 "필터를 사용하여 데이터의 부분 집합 표시 및 편집" 참조). 어떤 테이블 타입 데이터셋에서는 사용 가능한 레코드의 부분 집합에 액세스할 수 있는 추가 방법인 범위를 제공합니다.

범위는 *TTable*과 클라이언트 데이터셋에만 적용됩니다. 범위와 필터는 서로 유사하지만 다른 용도를 가집니다. 다음 항목에서는 범위와 필터의 차이점 및 범위를 사용하는 방법에 대해 설명합니다.

범위와 필터의 차이점 이해

범위와 필터는 보이는 레코드를 모든 사용 가능 레코드의 부분 집합으로 제한하지만 이를 수행하는 방법은 서로 다릅니다. 범위는 지정된 경계 값들 사이의 연속적으로 인덱싱된 레코드 집합입니다. 예를 들어, 성에 따라 인덱싱된 직원 데이터베이스에서 "Jones"와 "Smith" 사이에 있는 성을 가진 직원들을 모두 표시하도록 범위를 적용할 수 있습니다. 범위는 인덱스에 따라 다르기 때문에 현재 인덱스를 범위를 정의하는 데 사용할 수 있는 인덱스로 설정해야 합니다. 레코드를 정렬하기 위해 인덱스를 지정하는 것과 마찬가지로, *IndexName*이나 *IndexFieldNames* 속성 중 하나를 사용하여 범위를 정의할 대상에 인덱스를 할당할 수 있습니다.

반면, 필터는 인덱스에 상관없이 특정 데이터 포인트를 공유하는 레코드 집합을 말합니다. 예를 들어, 캘리포니아에 살면서 회사에 근무한지 5년 이상되는 직원들을 모두 표시하도록 직원 데이터베이스를 필터링할 수 있습니다. 필터가 적용될 때 인덱스를 사용할 수 있지만, 필터가 인덱스에 의존하지는 않습니다. 필터는 애플리케이션에서 데이터셋을 스크롤할 때 한 레코드씩 적용됩니다.

일반적으로 필터가 범위보다 더 유연합니다. 하지만 데이터셋이 크고 애플리케이션에서 관심있는 레코드가 연속적인 인덱싱된 그룹으로 이미 블록화되어 있는 경우는 범위가 더 효과적일 수 있습니다. 아주 큰 데이터셋에서는 데이터를 선택할 때 쿼리 타입 데이터셋의 WHERE 절을 사용하는 것이 훨씬 더 효율적일 수 있습니다. 쿼리 지정에 대한 자세한 내용은 22-40페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오.

범위 지정

범위를 지정하는 두 가지의 상호 배타적인 방법은 다음과 같습니다.

- *SetRangeStart*와 *SetRangeEnd*를 사용하여 범위의 시작과 끝을 각각 지정합니다.
- *SetRange*를 사용하여 양쪽 끝점을 한 번에 지정합니다.

범위의 시작 설정

SetRangeStart 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 시작 값 리스트를 만들기 시작합니다. 일단 *SetRangeStart*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용할 시작 인덱스 값으로 간주됩니다. 지정된 필드는 현재 인덱스에 적용해야 합니다.

예를 들어, 사용하는 애플리케이션에서 CUSTOMER 테이블에 연결된 *Customers*라는 *TSQLClientDataSet* 컴포넌트를 사용하며 사용자가 *Customers* 데이터셋의 각 필드에 대해 영구적 필드(persistent field) 컴포넌트를 작성했다고 가정합니다. CUSTOMER는 첫 번째 열(*CustNo*)에 인덱싱되어 있습니다. 애플리케이션의 폼에는 범위의 시작과 끝 값을 지정하는 데 사용되는 *StartVal*과 *EndVal*이라는 두 개의 편집 컴포넌트가 있습니다. 다음 코드는 범위를 만들어 적용하는 데 사용할 수 있습니다.

```
Customers->SetRangeStart();
Customers->FieldValues[“CustNo”] = StrToInt(StartVal->Text);
Customers->SetRangeEnd();
if (!EndVal->Text.IsEmpty())
  Customers->FieldValues[“CustNo”] = StrToInt(EndVal->Text);
Customers->ApplyRange();
```

이 코드는 *Fields*에 값을 할당하기 전에 *EndVal*에 입력된 텍스트가 Null이 아님을 확인합니다. *StartVal* 에 입력된 텍스트가 Null인 경우 어떠한 값도 Null 보다는 크기 때문에 데이터셋의 시작부터 모든 레코드가 포함됩니다. 그러나 *EndVal*에 입력된 텍스트가 Null인 경우 어떠한 값도 Null보다는 크므로 레코드가 하나도 포함되지 않습니다.

여러 열 인덱스의 경우 인덱스의 모든 필드나 일부 필드에 대해 시작 값을 지정할 수 있습니다. 인덱스에 사용된 필드에 값을 지정하지 않으면, 범위를 적용할 때 Null로 간주됩니다. 인덱스에 없는 필드에 값을 설정하려고 하면 데이터셋은 예외를 발생합니다.

팁 데이터셋의 처음에서 시작하려면 *SetRangeStart* 호출을 생략합니다.

범위의 시작 지정을 완료하기 위해서는 *SetRangeEnd*를 호출하거나 범위를 적용 또는 취소하십시오. 범위 적용 및 취소에 대한 자세한 내용은 22-33페이지의 "범위의 적용 또는 취소"를 참조하십시오.

범위의 끝 설정

SetRangeEnd 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 끝 값 리스트를 만들기 시작합니다. 일단 *SetRangeEnd*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용하는 끝 인덱스 값으로 간주됩니다. 지정된 필드는 현재 인덱스에 적용해야 합니다.

경고 범위의 끝을 데이터셋의 마지막 레코드로 정하고자 하더라도 항상 범위의 끝 값을 지정하십시오. 끝 값을 제공하지 않으면 C++Builder에서는 범위의 끝 값을 Null 값으로 간주합니다. 끝 값이 Null인 범위는 항상 비어 있습니다.

끝 값을 할당하는 가장 쉬운 방법은 *FieldByName* 메소드를 호출하는 것입니다. 예를 들면, 다음과 같습니다.

```
Contacts->SetRangeStart();
Contacts->FieldByName(“LastName”)->Value = Edit1->Text;
Contacts->SetRangeEnd();
Contacts->FieldByName(“LastName”)->Value = Edit2->Text;
Contacts->ApplyRange();
```

범위 값의 시작을 지정할 때 인덱스에 없는 필드에 대해 값을 설정하고자 하면 데이터셋은 예외가 발생합니다.

범위의 끝을 지정하였으면 해당 범위를 적용하거나 취소합니다. 범위 적용 및 취소에 대한 자세한 내용은 22-33페이지의 "범위의 적용 또는 취소"를 참조하십시오.

시작 및 끝 범위 값 설정

*SetRangeStart*와 *SetRangeEnd*를 각각 호출하여 범위 경계를 지정하는 대신, *SetRange* 프로시저를 한 번 호출하면 데이터셋을 *dsSetKey* 상태로 두고 범위의 시작과 끝 값을 설정할 수 있습니다.

*SetRange*에서는 두 개의 상수 배열 매개변수, 즉 시작 값 집합과 끝 값 집합을 취합니다. 예를 들어, 다음과 같은 명령문은 2열 인덱스를 기반으로 범위를 지정합니다.

```
TVarRec StartVals[2];
TVarRec EndVals[2];
StartVals[0] = Edit1->Text;
StartVals[1] = Edit2->Text;
EndVals[0] = Edit3->Text;
EndVals[1] = Edit4->Text;

Table1->SetRange(StartVals, 1, EndVals, 1);
```

여러 열 인덱스의 경우 인덱스의 모든 필드나 일부 필드에 시작과 끝 값을 지정할 수 있습니다. 인덱스에 사용된 필드에 값을 지정하지 않으면, 범위를 적용할 때 Null로 간주됩니다. 인덱스의 첫 번째 필드에 대한 값을 생략하고 그 다음 필드의 값을 지정하려면 생략된 필드에 Null 값을 전달합니다.

범위의 끝을 데이터셋의 마지막 레코드로 정하고자 하더라도 항상 범위의 끝 값을 지정하십시오. 끝 값을 지정하지 않으면 데이터셋은 범위의 끝 값을 Null로 간주합니다. 끝 값이 Null인 범위는 시작 값이 끝 값보다 크거나 같기 때문에 항상 비어 있습니다.

부분 키에 따른 범위 지정

키가 하나 이상의 문자열 필드로 구성되어 있는 경우 *SetRange* 메소드는 부분 키를 지원합니다. 예를 들어 인덱스가 *LastName*과 *FirstName* 열에 기반하는 경우 다음과 같은 범위 지정이 유효합니다.

```
Contacts->SetRangeStart();
Contacts->FieldValues[“LastName”] = “Smith”;
Contacts->SetRangeEnd();
Contacts->FieldValues[“LastName”] = “Zzzzzz”;
Contacts->ApplyRange();
```

이 코드는 *LastName*이 "Smith"보다 크거나 같은 범위의 모든 레코드를 포함시킵니다. 값 규정은 다음과 같을 수 있습니다.

```
Contacts->FieldValues[“LastName”] = “Sm”;
```

이 명령문은 *LastName*이 "Sm"보다 크거나 같은 레코드를 포함시킵니다.

경계 값과 일치하는 레코드 포함 또는 제외

디폴트로, 범위에는 지정된 시작 범위보다 크거나 같은 레코드와 지정된 끝 범위보다 작거나 같은 모든 레코드가 포함됩니다. 이러한 동작은 *KeyExclusive* 속성에서 제어합니다. *KeyExclusive*는 기본값이 **false**입니다.

원한다면 데이터셋의 *KeyExclusive* 속성을 **true**로 설정하여 끝 범위와 동일한 레코드를 제외할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Contacts->SetRangeStart();
Contacts->KeyExclusive = true;
Contacts->FieldValues[“LastName”] = “Smith”;
Contacts->SetRangeEnd();
Contacts->FieldValues[“LastName”] = “Tyler”;
Contacts->ApplyRange();
```

이 코드는 *LastName*이 "Smith"보다 크거나 같고 "Tyler"보다 작은 모든 레코드를 범위에 포함시킵니다.

범위 수정

두개의 함수, 범위의 시작 값을 변경하려면 *EditRangeStart*를 그리고 범위의 끝 값을 변경하려면 *EditRangeEnd*를 사용하면 범위에 대한 기존의 경계 조건을 수정할 수 있습니다.

범위를 편집하고 적용하기 위한 프로세스에는 다음과 같은 일반적인 단계가 있습니다.

1. 데이터셋을 *dsSetKey* 상태로 두고 범위의 시작 인덱스 값을 수정합니다.
2. 범위의 끝 인덱스 값을 수정합니다.
3. 데이터셋에 범위를 적용합니다.

범위의 시작 또는 끝 값을 수정하거나 또는 두 가지 경계 조건을 모두 수정할 수 있습니다. 현재 데이터셋에 적용된 범위에 대해 경계 조건을 수정하는 경우 *ApplyRange*를 다시 호출할 때까지 변경 내용이 적용되지 않습니다.

범위의 시작 편집

EditRangeStart 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 만들고 범위 시작 값의 현재 리스트를 수정하기 시작합니다. 일단 *EditRangeStart*를 호출한 다음 이어서 *Fields* 속성에 할당하면 범위를 적용할 때 사용할 현재 인덱스 값을 덮어씁니다.

팁 처음에 부분 키에 따라 시작 범위를 만든 경우 *EditRangeStart*를 사용하여 범위의 시작 값을 확장할 수 있습니다. 부분 키에 따른 범위에 대한 자세한 내용은 22-32페이지의 "부분 키에 따른 범위 지정"을 참조하십시오.

범위의 끝 편집

EditRangeEnd 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 끝 값 리스트를 만들기 시작합니다. 일단 *EditRangeEnd*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용하는 끝 인덱스 값으로 간주됩니다.

범위의 적용 또는 취소

범위의 시작을 지정하기 위해 *SetRangeStart* 또는 *EditRangeStart*를 호출하거나, 범위의 끝을 지정하기 위해 *SetRangeEnd* 또는 *EditRangeEnd*를 호출할 경우 데이터셋은 *dsSetKey* 상태가 됩니다. 개발자가 범위를 적용 또는 취소할 때까지 계속 이 상태를 유지합니다.

범위 적용

범위를 지정할 때 사용자가 정의하는 경계 조건은 사용자가 범위를 적용하기 전까지는 효력이 없습니다. 범위가 효력을 가지려면 *ApplyRange* 메소드를 호출합니다. *ApplyRange*는 사용자가 지정된 데이터셋의 부분 집합에 있는 데이터를 보거나 액세스하는 것을 즉시 제한합니다.

범위 취소

CancelRange 메소드는 범위의 애플리케이션을 끝내고 전체 데이터셋에 대한 액세스를 복구합니다. 범위를 취소하면 데이터셋의 모든 레코드에 대한 액세스가 복구되더라도 해당 범위의 경계 조건은 계속 유지되어 사용자가 나중에 범위를 다시 적용할 수 있습니다. 범위 경계는 사용자가 새로운 범위 경계를 제시하거나 기존 경계를 수정하기 전까지는 계속 유지됩니다. 예를 들어 다음과 같은 코드는 유효합니다.

```
f
MyTable->CancelRange();
f
// later on, use the same range again. No need to call SetRangeStart,
etc.
```

```
MyTable->ApplyRange();
f
```

마스터/디테일 관계 생성

테이블 타입 데이터셋은 마스터/디테일 관계로 연결될 수 있습니다. 사용자가 마스터/디테일 관계를 설정할 경우, 한 테이블(디테일)의 모든 레코드가 다른 테이블(마스터)의 하나의 현재 레코드에 일치하도록 두 데이터셋을 연결합니다.

테이블 타입 데이터셋은 다음의 두 가지 방식 중 하나를 이용해서 마스터/디테일 관계를 지원합니다.

- 모든 테이블 타입 데이터셋은 커서를 연결하여 다른 데이터셋의 디테일 역할을 할 수 있습니다. 이 프로세스는 다음에 나오는 "테이블을 다른 데이터셋의 디테일로 만들기"에 설명되어 있습니다.
- *TTable*, *TSQLTable* 및 모든 클라이언트 데이터셋은 중첩된 디테일 테이블을 사용하는 마스터/디테일 관계에서 마스터의 역할을 할 수 있습니다. 이 프로세스는 22-36페이지의 "중첩 디테일 테이블 사용"에 설명되어 있습니다.

이러한 방법은 저마다 고유한 장점이 있습니다. 커서를 연결하면 마스터 테이블이 데이터셋 타입인 마스터/디테일 관계를 만들 수 있습니다. 중첩된 디테일을 사용하면 디테일 테이블로 사용될 수 있는 데이터셋 타입은 제한되지만 데이터를 표시하는 방법 옵션은 더 많이 제공합니다. 마스터가 클라이언트 데이터셋이면 중첩된 디테일은 캐싱된 업데이트를 제공하는 데 좀 더 강력한 메커니즘을 제공합니다.

테이블을 다른 데이터셋의 디테일로 만들기

테이블 타입 데이터셋의 *MasterSource*와 *MasterFields* 속성은 두 데이터셋 간의 일대다 관계를 설정하는 데 사용될 수 있습니다.

MasterSource 속성은 테이블에서 마스터 테이블의 데이터를 가져오는 데이터 소스를 지정하는 데 사용됩니다. 이러한 데이터 소스는 모든 데이터셋 타입에 연결될 수 있습니다. 예를 들어, 쿼리의 데이터 소스를 이러한 속성에서 지정하면 클라이언트 데이터셋을 쿼리의 디테일로 연결할 수 있으므로 클라이언트 데이터셋에서는 쿼리에서 발생하는 이벤트를 추적합니다.

데이터셋은 현재 인덱스를 기반으로 마스터 테이블에 연결됩니다. 디테일 데이터셋에서 추적하는 마스터 데이터셋의 필드를 지정하기 전에 해당 필드로 시작하는 디테일 데이터셋의 인덱스를 우선 지정하십시오. *IndexName*이나 *IndexFieldNames* 속성 중 하나를 사용할 수 있습니다.

사용할 인덱스를 지정하고 나면 *MasterFields* 속성을 사용하여 디테일 테이블의 인덱스 필드에 해당하는 마스터 데이터셋의 열을 나타냅니다. 여러 열 이름에 데이터셋을 연결하려면 다음과 같이 세미콜론으로 필드 이름을 구분합니다.

```
Parts->MasterFields = “OrderNo;ItemNo”;
```

두 데이터셋 사이의 의미 있는 연결을 돕는 Field Link 디자이너를 사용할 수 있습니다. Field Link 디자이너를 사용하려면 *MasterSource*와 인덱스를 할당한 후 Object Inspector의 *MasterFields* 속성을 더블 클릭합니다.

다음 단계에서는 사용자가 고객 레코드를 스크롤하여 현재 고객에 대한 모든 순서를 표시할 수 있는 간단한 폼을 만듭니다. 마스터 테이블은 *CustomersTable*이고, 디테일 테이블은 *OrdersTable*입니다. 예제에서는 BDE 기반 *TTable* 컴포넌트를 사용하지만 같은 메소드를 사용하여 모든 테이블 타입 데이터셋에 연결할 수 있습니다.

1. 두 개의 *TTable* 컴포넌트와 두 개의 *TDataSource* 컴포넌트를 데이터 모듈에 둡니다.
2. 첫 번째 *TTable* 컴포넌트의 속성을 다음과 같이 설정합니다.
   - *DatabaseName*: BCDEMOS
   - *TableName*:CUSTOMER
   - *Name*:CustomersTable
3. 다음과 같이 두 번째 *TTable* 컴포넌트의 속성을 설정합니다.
   - *DatabaseName*: BCDEMOS
   - *TableName*: ORDERS
   - *Name*:OrdersTable
4. 첫 번째 *TDataSource* 컴포넌트의 속성을 다음과 같이 설정합니다.
   - *Name*:CustSource
   - *DataSet*:CustomersTable
5. 다음과 같이 두 번째 *TDataSource* 컴포넌트의 속성을 설정합니다.
   - *Name*:OrdersSource
   - *DataSet*:OrdersTable
6. 두 개의 *TDBGrid* 컴포넌트를 폼에 둡니다.
7. File | Include Unit Hdr를 선택하여 폼에서 데이터 모듈을 사용한다고 지정합니다.
8. 첫 번째 그리드 컴포넌트의 *DataSource* 속성을 "CustSource"로 설정하고 두 번째 그리드의 *DataSource* 속성을 "OrdersSource"로 설정합니다.
9. *OrdersTable*의 *MasterSource* 속성을 "CustSource"로 설정합니다. 이렇게 하면 CUSTOMER 테이블(마스터 테이블)이 ORDERS 테이블(디테일 테이블)에 연결됩니다.
10. Object Inspector에서 *MasterFields* 속성 값 박스를 더블 클릭하여 다음 속성을 설정하는 Field Link 디자이너를 호출합니다.

- Available Indexes 필드에서 *CustNo* 필드로 두 개의 테이블을 연결할 *CustNo*를 선택합니다.
- Detail Fields 필드 리스트와 Master Fields 필드 리스트 모두에서 *CustNo*를 선택합니다.
- Add 버튼을 클릭하여 조인 조건을 추가합니다. Joined Fields 리스트에 "CustNo -> CustNo"가 나타납니다.
- OK를 선택하여 선택한 내용을 커밋하고 Field Link 디자이너를 종료합니다.

11 *CustomersTable*과 *OrdersTable*의 *Active* 속성을 **true**로 설정하여 폼의 그리드에 데이터를 표시합니다.

12 애플리케이션을 컴파일하여 실행합니다.

이제 애플리케이션을 실행하면 테이블이 함께 연결되어서 CUSTOMER 테이블의 새 레코드로 이동하면 현재 고객에 속한 ORDERS 테이블의 레코드만 볼 수 있게 됩니다.

중첩 디테일 테이블 사용

중첩 테이블은 다른(마스터) 데이터셋에서 단일 데이터셋 필드의 값이 되는 디테일 데이터셋입니다. 서버 데이터를 나타내는 데이터셋의 경우 중첩 디테일 데이터셋은 서버의 데이터셋 필드에 대해서만 사용할 수 있습니다. *TClientDataSet* 컴포넌트는 서버 데이터를 나타내지 않지만, 중첩 디테일을 포함하는 TClientDataSet 컴포넌트에 대해 데이터셋을 만들거나 TClientDataSet 컴포넌트가 마스터/디테일 관계의 마스터 테이블에 연결하는 프로바이더에서 데이터를 받으면 데이터셋 필드를 포함할 수도 있습니다.

참고 *TClientDataSet*의 경우, 마스터 테이블과 디테일 테이블에서 데이터베이스 서버로 업데이트 내용을 적용하려면 중첩 디테일 셋을 사용해야 합니다.

중첩 디테일 셋을 사용하려면 마스터 데이터셋의 *ObjectView* 속성이 **true**이어야만 합니다. 테이블 타입 데이터셋에 중첩 디테일 데이터셋이 포함된 경우 *TDBGrid*에서 중첩 디테일을 팝업 메뉴에 표시하도록 지원합니다. 이러한 표시 방법에 대한 자세한 내용은 23-25페이지의 "데이터셋 필드 표시"를 참조하십시오.

그 대신, 디테일 셋에 대해 각각의 데이터셋 컴포넌트를 사용하여 데이터 인식 컨트롤에 디테일 데이터셋을 표시하거나 편집할 수 있습니다. 디자인 타임 시 마우스 오른쪽 버튼으로 마스터 데이터셋을 클릭한 다음 Fields Editor를 선택하면 나타나는, Fields Editor를 사용하여 개발자(마스터) 데이터셋의 필드에 대해 영구적 필드(persistent field)를 만듭니다. Add Fields를 마우스 오른쪽 버튼으로 클릭하여 개발자 데이터셋에 새로운 영구적 필드를 추가합니다. DataSet Field 타입의 새로운 필드를 정의합니다. Fields Editor에서 디테일 테이블 구조를 정의합니다. 마스터 데이터셋에서 사용되는 다른 필드에 대해서도 영구적 필드를 추가해야 합니다.

디테일 테이블의 데이터셋 컴포넌트는 마스터 테이블에서 허용하는 타입의 데이터셋 자손입니다. *TTable* 컴포넌트는 *TNestedDataSet* 컴포넌트를 중첩된 데이터셋으로만 허용합니다. *TSQLTable* 컴포넌트는 다른 *TSQLTable* 컴포넌트를 허용합니다. *TClientDataset* 컴포넌트는 다른 클라이언트 데이터셋을 허용합니다. 컴포넌트 팔레트에서 적절한 타입의 데이터셋을 선택한 다음 폼이나 데이터 모듈에 추가합니다. 이러한 디테일 데이터셋의 *DataSetField* 속성을 마스터 데이터셋의 영구적 DataSet 필드로 설정합니다. 마지막으로 데이터 소스 컴포넌트를 폼이나 데이터 모듈에 둔 다음 이 컴포넌트의 *DataSet* 속성을 디테일 데이터셋으로 설정합니다.
데이터 인식 컨트롤에서는 이러한 데이터 소스를 사용하여 디테일 셋의 데이터에 액세스할 수 있습니다.

테이블의 읽기/쓰기 액세스 제어

디폴트로, 테이블 타입 데이터셋이 열려 있으면 원본으로 사용하는 데이터베이스 테이블에 대한 읽기 및 쓰기 액세스를 요청합니다. 원본으로 사용하는 데이터베이스 테이블의 특징에 따라 요청한 쓰기 권한은 승인되지 않을 수 있습니다. 예를 들어, 원격 서버에서 SQL 테이블에 대해 쓰기 액세스를 요청하면 서버에서는 테이블의 액세스를 읽기 전용으로 제한합니다.

참고 데이터셋 프로바이더에서 데이터 패킷과 함께 제공하는 정보를 통해 사용자가 데이터를 편집할 수 있는지 여부를 결정하는 *TClientDataSet*에서는 위의 내용이 적용되지 않습니다. 그리고 단방향 데이터셋이기 때문에 항상 읽기 전용인 *TSQLTable*에도 적용되지 않습니다.

테이블이 열리면 *CanModify* 속성을 검사하여 원본으로 사용하는 데이터베이스나 데이터셋 프로바이더에서 사용자에게 테이블 데이터의 편집을 허용하는지 여부를 확인할 수 있습니다. *CanModify*가 **false**이면 애플리케이션에서는 데이터베이스에 기록할 수 없습니다. *CanModify*가 **true**인 경우 테이블의 *ReadOnly* 속성이 **false**이면 애플리케이션에서는 데이터베이스에 기록할 수 있습니다.

*ReadOnly*는 사용자가 데이터를 보고 편집할 수 있는지를 모두 결정합니다. *ReadOnly*가 **false**(기본값)이면 사용자는 데이터를 보고 편집할 수 있습니다. 사용자가 데이터를 보기만 하게 제한하려면 테이블을 열기 전에 *ReadOnly*를 **true**로 설정합니다.

참고 *ReadOnly*는 항상 읽기 전용인 *TSQLTable*만 제외하고 모든 테이블 타입 데이터셋에서 구현됩니다.

테이블 생성 및 삭제

일부 테이블 타입 데이터셋을 사용하면 디자인 타임이나 런타임 시 원본으로 사용하는 테이블을 만들고 삭제할 수 있습니다. 일반적으로 데이터베이스 테이블은 데이터베이스 관리자에 의해 만들어지고 삭제됩니다. 하지만 애플리케이션 개발 및 테스트 중에는 애플리케이션에서 사용할 수 있는 데이터베이스 테이블을 만들고 삭제할 수 있습니다.

테이블 생성

*TTable*과 *TIBTable*은 모두 SQL을 사용하지 않고 원본으로 사용하는 데이터베이스 테이블을 만듭니다. 마찬가지로 *TClientDataSet*을 사용하면 데이터셋 프로바이더를 사용하지 않을 때 데이터셋을 만들 수 있습니다. *TTable*과 *TClientDataSet*을 사용하면 디자인 타임이나 런타임 시 테이블을 만들 수 있습니다. *TIBTable*을 사용해야만 런타임 시 테이블을 만들 수 있습니다.

테이블을 만들기 전에 만들고 있는 테이블의 구조를 지정하기 위해 속성을 설정해야 합니다. 특히, 다음과 같은 내용을 지정해야 합니다.

- 새 테이블을 호스트할 데이터베이스. *TTable*에서는 *DatabaseName* 속성을 사용하여 데이터베이스를 지정합니다. *TIBTable*에서는 *Database* 속성에 할당된 *TIBDatabase* 컴포넌트를 사용해야 합니다(클라이언트 데이터셋에서는 데이터베이스를 사용하지 않음).
- 데이터베이스 타입(*TTable*만 해당). *TableType* 속성을 테이블의 원하는 타입으로 설정합니다. Paradox, dBASE 또는 ASCII 테이블의 경우 *TableType*을 *ttParadox*, *ttDBase* 또는 *ttASCII*로 각각 설정합니다. 모든 다른 테이블 타입의 경우 *TableType*을 *ttDefault*로 설정합니다.

- 만들려는 테이블의 이름. *TTable*과 *TIBTable*은 모두 새 테이블의 이름에 *TableName* 속성을 갖습니다. 클라이언트 데이터셋은 테이블 이름을 사용하지 않지만 새 테이블을 저장하기 전에 *FileName* 속성을 지정해야 합니다. 기존 테이블과 이름이 중복되는 테이블을 만드는 경우 새로 만든 테이블이 기존 테이블과 테이블의 모든 데이터를 덮어씁니다. 이전 테이블과 데이터는 복원할 수 없습니다. 기존 테이블을 덮어쓰는 것을 피하려면 런타임 시 *Exists* 속성을 검사할 수 있습니다. *Exists*는 *TTable*과 *TIBTable*에서만 사용할 수 있습니다.
- 새 테이블의 필드. 이를 수행하는 방법은 두 가지가 있습니다.
  - 필드 정의를 *FieldDefs* 속성에 추가할 수 있습니다. 디자인 타임 시 Object Inspector에서 *FieldDefs* 속성을 클릭하여 컬렉션 에디터를 불러옵니다. 컬렉션 에디터를 사용하여 필드 정의의 속성을 추가, 제거 또는 변경합니다. 런타임 시 기존 필드 정의를 지운 다음 *AddFieldDef* 메소드를 사용하여 새 필드 정의를 각각 추가합니다. 새 필드를 각각 정의하는 경우 *TFieldDef* 객체의 속성을 설정하여 원하는 필드 어트리뷰트(attribute)를 지정합니다.
  - 대신 영구적 필드 컴포넌트를 사용할 수 있습니다. 디자인 타임 시 데이터셋을 더블 클릭하여 *Fields* Editor를 불러옵니다. Fields Editor에서 마우스 오른쪽 버튼을 클릭하고 New Field 명령을 선택합니다. 필드의 기본 속성을 설명합니다. 필드가 생성되면 Fields Editor에서 필드를 선택하여 Object Inspector에서 속성을 변경할 수 있습니다.
- 새 테이블의 인덱스(옵션). 디자인 타임 시, Object Inspector에서 *IndexDefs* 속성을 더블 클릭하여 컬렉션 에디터를 불러옵니다. 컬렉션 에디터를 사용하여 인덱스 정의의 속성을 추가, 제거 또는 변경합니다. 런타임 시 기존 필드 정의를 지운 다음 *AddIndexDef* 메소드를 사용하여 새 인덱스 정의를 각각 추가합니다. 새 필드를 각각 정의하는 경우 *TIndexDef* 객체의 속성을 설정하여 원하는 인덱스 어트리뷰트를 지정합니다.

참고 필드 정의 객체 대신 영구적 필드 컴포넌트를 사용하여 새 테이블의 인덱스를 정의할 수 없습니다.

디자인 타임 시 테이블을 만들려면 마우스 오른쪽 버튼으로 데이터셋을 클릭하고 Create Table(*TTable*)이나 Create Data Set(*TClientDataSet*)을 선택합니다. 모든 필요한 정보를 지정할 때까지 이 명령은 컨텍스트 메뉴에 나타나지 않습니다.

런타임 시 테이블을 만들려면 *CreateTable* 메소드(*TTable*과 *TIBTable*)나 *CreateDataSet* 메소드(*TClientDataSet*)를 호출합니다.

참고 디자인 타임 시 정의를 설정한 다음 런타임 시 *CreateTable* 또는 *CreateDataSet* 메소드를 호출하여 테이블을 만들 수 있습니다. 하지만 테이블을 만들려면 런타임 시 지정한 정의를 데이터셋 컴포넌트와 함께 저장해야 한다고 표시해야 합니다. (디폴트로, 필드와 인덱스 정의는 런타임 시 동적으로 생성됩니다.) *StoreDefs* 속성을 **true**로 설정하여 정의를 데이터셋과 함께 저장하도록 지정합니다.

팁 *TTable*을 사용하고 있는 경우 디자인 타임 시 기존 테이블의 필드 정의와 인덱스 정의를 미리 로드할 수 있습니다. *DatabaseName*과 *TableName* 속성을 설정하여 기존 테이블을 지정합니다. 마우스 오른쪽 버튼으로 테이블 컴포넌트를 클릭하여 Update Table Definition 을 선택합니다. 이렇게 하면 자동으로 *FieldDefs*와 *IndexDefs* 속성을 설정하여 기존 테이블의 필드와 인덱스를 설명합니다. 다음으로는 *DatabaseName*과 *TableName*을 다시 설정하여 기존 테이블의 이름을 바꾸는 메시지를 취소하고 만들려는 테이블을 지정합니다.

참고 Oracle8 테이블을 만들 때는 객체 필드(ADT 필드, 배열 필드 및 데이터셋 필드)를 만들 수 없습니다.

다음 코드는 런타임 시 새 테이블을 만들어 BCDEMOS 알리아스와 연결합니다. 코드에서 새 테이블을 만들기 전에 제공한 테이블 이름이 기존 테이블 이름과 일치하지 않는지 다음과 같이 확인합니다.

```
TTable *NewTable = new TTable(Form1);
NewTable->Active = false;
NewTable->DatabaseName = "BCDEMOS";
NewTable->TableName = Edit1->Text;
NewTable->TableType = ttDefault;
NewTable->FieldDefs->Clear();
TFieldDef *NewField = NewTable->FieldDefs->AddFieldDef(); // define first
field
NewField->DataType = ftInteger;
NewField->Name = Edit2->Text;
NewField = NewTable->FieldDefs->AddFieldDef(); // define second field
NewField->DataType = ftString;
NewField->Size = StrToInt(Edit3->Text);
NewField->Name = Edit4->Text;
NewTable->IndexDefs->Clear();
TIndexDef *NewIndex = NewTable->IndexDefs->AddIndexDef(); // add an index
NewIndex->Name = "PrimaryIndex";
NewIndex->Fields = Edit2->Text;
NewIndex->Options << ixPrimary << ixUnique;
// Now check for prior existence of this table
bool CreateIt = (!NewTable->Exists);
if (!CreateIt)
   if (Application->MessageBox((AnsiString("Overwrite table ") + Edit1-
>Text +
                                AnsiString("?")).c_str(),
                                "Table Exists", MB_YESNO) == IDYES)
    CreateIt = true;
if (CreateIt)
  NewTable->CreateTable(); // create the table
```

테이블 삭제

*TTable*과 *TIBTable*을 사용하면 SQL을 사용하지 않고 원본으로 사용하는 데이터베이스 테이블에서 테이블을 삭제할 수 있습니다. 런타임 시 테이블을 삭제하려면 데이터셋의 *DeleteTable* 메소드를 호출합니다. 예를 들어, 다음 명령문은 데이터셋의 원본으로 사용하는 테이블을 제거합니다.

```
CustomersTable->DeleteTable();
```

주의 *DeleteTable*을 사용하여 테이블을 삭제하면 테이블과 데이터는 영구히 사라집니다.

*TTable*을 사용하는 경우 다음과 같이 디자인 타임 시에도 테이블을 삭제할 수 있습니다. 마우스 오른쪽 버튼으로 테이블 컴포넌트를 클릭하여 컨텍스트 메뉴에서 Delete Table을 선택합니다. 테이블 컴포넌트에서 기존 데이터베이스 테이블을 나타낼 때만, 즉 *DatabaseName*과 *TableName* 속성에서 기존 테이블을 지정할 때만 Delete Table 메뉴 선택이 가능합니다.

테이블 비우기

많은 테이블 타입 데이터셋에서는 테이블의 모든 데이터 행을 삭제할 수 있게 해주는 단일 메소드를 제공합니다.

- *TTable*과 *TIBTable*에서는 런타임 시 *EmptyTable* 메소드를 호출하여 모든 레코드를 삭제할 수 있습니다.

```
PhoneTable->EmptyTable();
```

- *TADOTable*에서는 *DeleteRecords* 메소드를 사용할 수 있습니다.

  ```
  PhoneTable->DeleteRecords(arAll);
  ```

- *TSQLTable*에서도 *DeleteRecords* 메소드를 사용할 수 있습니다. 하지만 *DeleteRecords* 의 *TSQLTable* 버전에서는 매개변수를 취하지 않음에 유의하십시오.

  ```
  PhoneTable->DeleteRecords();
  ```

- 클라이언트 데이터셋의 경우 *EmptyDataSet* 메소드를 사용할 수 있습니다.

  ```
  PhoneTable->EmptyDataSet();
  ```

참고 SQL 서버 테이블의 경우 테이블에 대한 DELETE 권한이 있어야만 이러한 메소드가 성공합니다.

주의 데이터셋을 비우면 삭제한 데이터는 영구히 사라집니다.

테이블 동기화

동일한 데이터베이스 테이블을 나타내지만 데이터 소스 컴포넌트를 공유하지 않는 두 개 이상의 데이터셋이 있으면 각 데이터셋은 데이터의 고유한 뷰와 고유한 현재 레코드를 갖습니다. 각 데이터셋을 통해 사용자가 레코드에 액세스할 때 컴포넌트의 현재 레코드가 달라집니다.

데이터셋이 *TTable*의 모든 인스턴스, *TIBTable*의 모든 인스턴스 또는 모든 클라이언트 데이터셋이면 *GotoCurrent* 메소드를 호출하여 이러한 데이터셋 각각의 현재 레코드를 같게 만들 수 있습니다. *GotoCurrent*는 자체 데이터셋의 현재 레코드를 일치하는 데이터셋의 현재 레코드로 설정합니다. 예를 들어, 다음 코드는 *CustomerTableOne*의 현재 레코드를 *CustomerTableTwo*의 현재 레코드와 같게 설정합니다.

```
CustomerTableOne->GotoCurrent(CustomerTableTwo);
```

팁 애플리케이션에서 데이터셋을 이러한 방식으로 동기화해야 하는 경우 데이터셋을 데이터 모듈에 놓은 다음 테이블에 액세스하는 단위별로 데이터 모듈의 헤더를 포함해야 합니다.

각각의 폼에서 가져온 데이터셋을 동기화하려면 한 폼의 헤더 파일을 다른 폼의 소스 유닛에 포함해야 하며 한 개 이상의 데이터셋 이름을 폼 이름으로 한정해야 합니다. 예를 들면, 다음과 같습니다.

```
CustomerTableOne->GotoCurrent(Form2->CustomerTableTwo);
```

쿼리 타입 데이터셋 사용

다음과 같은 방법으로 쿼리 타입 데이터셋을 사용합니다.

1. 해당 데이터셋 컴포넌트를 데이터 모듈이나 폼에 배치하고 *Name* 속성을 애플리케이션에 적절한 고유 값으로 설정합니다.
2. 쿼리할 데이터베이스 서버를 식별합니다. 쿼리 타입 데이터셋별로 이러한 식별 방법은 다르지만 일반적으로 다음과 같이 데이터베이스 연결 컴포넌트를 지정하면 됩니다.
   - *TQuery*에서는 *DatabaseName* 속성을 사용하여 *TDatabase* 컴포넌트나 BDE 알리아스를 지정합니다.
   - *TADOQuery*에서는 *Connection* 속성을 사용하여 *TADOConnection* 컴포넌트를 지정합니다.

- *TSQLQuery*에서는 *SQLConnection* 속성을 사용하여 *TSQLConnection* 컴포넌트를 지정합니다.
- *TIBQuery*에서는 *Database* 속성을 사용하여 *TIBConnection* 컴포넌트를 지정합니다.

데이터베이스 연결 컴포넌트에 대한 자세한 내용은 21장, "데이터베이스에 연결"을 참조하십시오.

3. 데이터셋의 *SQL* 속성에서 SQL 문을 지정한 다음 해당 명령문의 매개변수를 선택적으로 지정합니다. 자세한 내용은 22-41페이지의 "쿼리 지정" 및 22-43페이지의 "쿼리 매개변수 사용"을 참조하십시오.
4. 쿼리 데이터를 비주얼(visual) 데이터 컨트롤과 함께 사용하려면 데이터 소스 컴포넌트를 데이터 모듈에 추가하고 *DataSet* 속성을 쿼리 타입 데이터셋으로 설정합니다. 데이터 소스 컴포넌트는 *result set*이라는 쿼리 결과를 표시할 데이터 인식 컴포넌트로 전달합니다. *DataSource* 및 *DataField* 속성을 사용하여 데이터 인식 컴포넌트를 데이터 소스에 연결합니다.
5. 쿼리 컴포넌트를 활성화합니다. 결과 집합을 반환하는 쿼리에서는 *Active* 속성이나 *Open* 메소드를 사용합니다. 데이블에서 동작만 수행하고 결과 집합을 반환하지 않는 쿼리를 실행하려면 런타임 시 *ExecSQL* 메소드를 사용합니다. 쿼리를 두 번 이상 실행할 계획이면 *Prepare*를 호출하여 데이터 액세스 레이어를 초기화하고 매개변수 값을 쿼리에 연결할 수 있습니다. 쿼리 준비에 관한 자세한 내용은 22-46페이지의 "쿼리 준비"를 참조하십시오.

쿼리 지정

true 쿼리 타입 데이터셋에서는 *SQL* 속성을 사용하여 실행할 데이터셋의 SQL 문을 지정합니다. *TADODataSet*, *TSQLDataSet* 및 클라이언트 데이터셋과 같은 일부 데이터셋에서는 *CommandText* 속성을 사용하여 실행할 데이터셋의 SQL 문을 지정합니다.

레코드를 반환하는 대부분의 쿼리가 SELECT 명령입니다. 일반적으로 이런 쿼리는 포함할 필드, 이들 필드를 선택할 테이블, 포함할 레코드를 제한하는 조건 및 결과 데이터셋의 순서를 정의합니다. 예를 들면, 다음과 같습니다.

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

레코드를 반환하는 쿼리에는 SELECT 문(예를 들어, INSERT, DELETE, UPDATE, CREATE INDEX 및 ALTER TABLE 명령은 쿼리를 반환하지 않음)을 비롯하여 데이터 정의 언어(DDL)나 데이터 조작 언어(DML) 문을 사용하는 명령문이 포함됩니다. 명령에 사용되는 랭귀지는 서버 특정이지만, 일반적으로 SQL 랭귀지의 SQL-92 표준과 호환됩니다.

실행하는 SQL 명령은 사용 중인 서버에서 수용 가능해야 합니다. 데이터셋은 SQL을 분석하거나 실행하지 않습니다. 실행을 위해 서버에 명령을 전달할 뿐입니다. SQL 문은 필요에 따라 복잡해질 수도 있지만(예를 들어, AND와 OR같은 여러 개의 중첩 논리 연산자를 사용하는 WHERE 절이 있는 SELECT 문) 대부분의 경우 명령은 하나의 완전한 SQL 문이 되어야 합니다. 일부 서버에서는 여러 명령문을 허용하는 "일괄" 구문도 지원합니다. 서버에서 그러한 구문을 지원하면 쿼리를 지정할 때 여러 명령문을 입력할 수 있습니다.

쿼리에서 사용하는 SQL 문은 실제 단어로 작성되거나 사용되거나 대체 가능한 매개변수를 포함할 수 있습니다. 매개변수를 사용하는 쿼리는 *매개변수화된 쿼리*라고 합니다. 매개변수화된 쿼리를 사용할 때 매개변수에 할당된 실제 값은 쿼리를 실행하기 전에 쿼리에 삽입됩니다. 매개변수화된 쿼리를 사용하면 SQL 문을 대체할 필요없이 런타임 시 사용자 뷰를 변경하여 바로 데이터에 액세스할 수 있으므로 아주 유연합니다. 매개변수화된 쿼리에 대한 자세한 내용은 22-43페이지의 "쿼리 매개변수 사용"을 참조하십시오.

SQL 속성을 사용하여 쿼리 지정

true 쿼리 타입 데이터셋 *(TQuery, TADOQuery, TSQLQuery* 또는 *TIBQuery)*을 사용하는 경우 해당 쿼리를 *SQL* 속성에 할당합니다. *SQL* 속성은 *TStrings* 객체입니다. 이 *TStrings* 객체에서 각각의 문자열은 쿼리 각 줄을 나타냅니다. 여러 줄을 사용하더라도 서버에서 쿼리가 실행되는 방법에는 영향을 미치지 않지만 다음과 같이 해당 명령문을 논리적 단위로 나누면 쿼리를 수정하고 디버깅할 때 훨씬 더 쉽습니다.

```
MyQuery->Close();
MyQuery->SQL->Clear();
MyQuery->SQL->Add("SELECT CustNo, OrderNO, SaleDate");
MyQuery->SQL->Add("FROM Orders");
MyQuery->SQL->Add("ORDER BY SaleDate");
MyQuery->Open();
```

아래에 나오는 코드는 기존 SQL 문의 단일 줄만 수정하는 방법을 보여 줍니다. 이 경우 ORDER BY 절은 해당 명령문의 세 번째 줄에 이미 존재하며 인덱스 2를 사용하여 *SQL* 속성에서 참조합니다.

```
MyQuery->SQL->Strings[2] = "ORDER BY OrderNO";
```

참고 SQL 속성을 지정하거나 수정할 때 데이터셋을 반드시 닫아야 합니다.

디자인 타임 시 String List Editor를 사용하여 쿼리를 지정합니다. Object Inspector의 *SQL* 속성의 생략 부호 버튼을 클릭해서 String List Editor를 표시합니다.

참고 C++Builder의 일부 버전에서 *TQuery*를 사용하는 경우 SQL Builder를 사용하여 데이터베이스에 있는 테이블과 필드의 시각적 표현을 기반으로 쿼리를 구성할 수도 있습니다. SQL Builder를 사용하려면 쿼리 컴포넌트를 선택하여 마우스 오른쪽 버튼으로 클릭한 다음 컨텍스트 메뉴를 호출하고 Graphical Query Editor를 선택합니다. SQL Builder를 사용하는 방법을 배우려면 SQL Builder를 연 다음 온라인 도움말을 사용합니다.

SQL 속성은 *TStrings* 객체이기 때문에 다음과 같이 *TStrings*::*LoadFromFile* 메소드를 호출하여 파일에서 쿼리 텍스트를 로드할 수 있습니다.

```
MyQuery->SQL->LoadFromFile("custquery.sql");
```

또한 *SQL* 속성의 *Assign* 메소드를 사용하여 문자열 리스트 객체의 내용을 *SQL* 속성으로 복사할 수 있습니다. *Assign* 메소드는 다음과 같이 새 명령문을 복사하기 전에 *SQL* 속성의 현재 내용을 자동으로 지웁니다.

```
MyQuery->SQL->Assign(Memo1->Lines);
```

CommandText 속성을 사용하여 쿼리 지정

TADODataSet, *TSQLDataSet* 또는 클라이언트 데이터셋을 사용하는 경우 다음과 같이 쿼리 문의 텍스트를 *CommandText* 속성에 할당합니다.

```
MyQuery->CommandText = "SELECT CustName, Address FROM Customer";
```

디자인 타임 시 Object Inspector에 직접 쿼리를 입력할 수 있습니다. 또한, 데이터베이스에 대한 데이터셋의 연결이 이미 활성화되어 있으면 *CommandText* 속성의 생략 부호 버튼을 클릭해서 Command Text Editor를 표시할 수 있습니다. Command Text Editor는 사용 가능한 테이블과 이들 테이블의 필드를 나열하여 쿼리를 쉽게 작성할 수 있게 해줍니다.

쿼리 매개변수 사용

매개변수화된 SQL 문은 매개변수나 변수 또는 디자인 타임이나 런타임 시 달라질 수 있는 매개변수나 변수의 값을 포함합니다. 매개변수는 비교를 위해 WHERE 절에 사용되는 데이터 값 등, SQL 문에 나타나는 데이터 값을 대체할 수 있습니다. 대개 매개변수는 해당 문자에 전달되는 데이터 값을 대신합니다. 예를 들어, 다음 INSERT 문에서 삽입할 값은 매개변수로 전달됩니다.

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

이 SQL 문에서 *:Name*, *:Capital* 및 *:Population*은 애플리케이션에서 런타임 시 해당 명령문에 입력할 실제 값의 위치 표시자입니다. 매개변수의 이름이 콜론으로 시작하고 있습니다. 매개변수 이름을 리터럴 값과 구분하기 위해 콜론이 필요합니다. 쿼리에 물음표(?)를 추가하면 명명되지 않은 매개변수를 포함시킬 수 있습니다. 명명되지 않은 매개변수는 고유한 이름이 없기 때문에 위치로 식별합니다.

데이터셋이 쿼리를 실행하기 전에, 쿼리 텍스트의 매개변수에 대한 값을 제공해야 합니다. *TQuery*, *TIBQuery*, *TSQLQuery* 및 클라이언트 데이터셋의 경우 *Params* 속성을 사용하여 이러한 값을 저장합니다. *TADOQuery*에서는 대신 *Parameters* 속성을 사용합니다. *Params*(또는 *Parameters*)는 매개변수 객체(*TParam* 또는 *TParameter*)의 컬렉션으로, 여기서 각 객체는 단일 매개변수를 나타냅니다. 쿼리의 텍스트를 지정할 때 데이터셋은 이러한 매개변수 객체 집합을 생성하고 데이터셋 타입에 따라 쿼리에서 추론할 수 있는 속성을 초기화합니다.

참고 *ParamCheck* 속성을 **false**로 설정하면, 쿼리 텍스트 변경에 응답하여 매개변수 객체를 자동으로 생성하지 않을 수 있습니다. 이러한 방법은 매개변수가 쿼리 자체의 매개변수가 아닌 데이터 정의 언어(DDL) 문의 일부로 포함되어 있는 DDL 문에서 유용합니다. 예를 들어, 내장 프로시저를 만드는 DDL 문에서는 내장 프로시저의 일부인 매개변수를 정의할 수 있습니다. *ParamCheck*를 **false**로 설정하면 이러한 매개변수를 쿼리의 매개변수로 혼동하지 않게 할 수 있습니다.

매개변수 값은 처음으로 실행되기 전에 SQL 문에 연결되어야 합니다. 쿼리 컴포넌트에서는 쿼리를 실행하기 전에 *Prepare* 메소드를 명시적으로 호출하지 않더라도 이러한 작업을 자동으로 합니다.

팁 매개변수가 연결된 열의 실제 이름에 해당하는 변수 이름을 매개변수에 부여하는 것은 좋은 프로그래밍 방법입니다. 예를 들어, 열 이름이 "Number"이면 이에 해당하는 매개변수는 ":Number"가 될 수 있습니다. 다른 데이터셋에서 매개변수 값을 가져와야 하는 데이터 소스를 사용하는 데이터셋의 경우 일치하는 이름을 사용하는 것이 특히 중요합니다. 이 프로세스는 22-45페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"에 설명되어 있습니다.

디자인 타임 시 매개변수 제공

디자인 타임 시 매개변수 컬렉션 에디터를 사용하여 매개변수를 지정할 수 있습니다. 매개변수 컬렉션 에디터를 표시하려면 Object Inspector의 *Params* 또는 *Parameters* 속성의 생략 부호 버튼을 클릭합니다. SQL 문에 매개변수가 없으면, 컬렉션 에디터에 객체가 나열되지 않습니다.

참고 매개변수 컬렉션 에디터는 다른 컬렉션 속성에 나타나는 것과 동일한 컬렉션 에디터입니다. 에디터가 다른 속성과 공유되기 때문에, 오른쪽 버튼을 클릭해서 표시되는 컨텍스트 메뉴에는 Add 및 Delete 명령이 포함됩니다. 하지만 이 명령들은 쿼리 매개변수에 대해서는 활성화되지 않습니다. 매개변수를 추가하거나 삭제할 수 있는 유일한 방법은 SQL 문 자체에 있습니다.

각 매개변수를 매개변수 컬렉션 에디터에서 선택합니다. 그런 다음 Object Inspector를 사용하여 해당 속성을 수정합니다.

Params 속성(*TParam* 객체)을 사용할 때 다음 내용을 검사하거나 수정해야 하는 경우가 있습니다.

- *DataType* 속성은 매개변수 값의 데이터 타입을 나열합니다. 일부 데이터셋의 경우 이 값을 올바르게 초기화할 수 있습니다. 데이터셋이 타입을 추론할 수 없을 경우 *DataType*은 *ftUnknown*이며 이를 변경하여 매개변수 값의 타입을 나타내야 합니다.

  DataType 속성은 매개변수의 논리 데이터 타입을 나열합니다. 일반적으로 이러한 데이터 타입은 서버 데이터 타입을 따릅니다. 서버 데이터 타입에 대한 특정 논리 타입 매핑에 대해서는 데이터 액세스 메커니즘(BDE, dbExpress, InterBase) 설명서를 참조하십시오.

- *ParamType* 속성은 선택한 매개변수를 나열합니다. 쿼리의 경우, 입력 매개변수만 포함할 수 있기 때문에 항상 *ptInput*입니다. *ParamType*의 값이 *ptUnknown*이면 이 값을 *ptInput*으로 변경합니다.

- *Value* 속성은 선택한 매개변수의 값을 지정합니다. 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우에는 *Value*를 공백으로 둘 수 있습니다.

Parameters 속성(*TParameter* 객체)을 사용할 때 다음 내용을 검사하거나 수정할 수 있습니다.

- *DataType* 속성은 매개변수 값에 대한 데이터 타입을 나열합니다. 일부 데이터 타입에서는 다음과 같은 추가 정보를 제공해야 합니다.
  - *NumericScale* 속성은 숫자 매개변수의 소수 자릿수를 나타냅니다.
  - *Precision* 속성은 숫자 매개변수의 전체 자릿수를 나타냅니다.
  - *Size* 속성은 문자열 매개변수의 문자 수를 나타냅니다.

- *Direction* 속성은 선택한 매개변수의 타입을 나열합니다. 쿼리의 경우에는, 입력 매개변수만 포함할 수 있기 때문에 항상 *pdInput*으로 초기화됩니다.
- *Attributes* 속성은 매개변수에서 받아들일 값 타입을 제어합니다. *Attributes*는 *psSigned*, *psNullable* 및 *psLong*의 조합으로 설정될 수 있습니다.
- *Value* 속성은 선택한 매개변수의 값을 지정합니다. 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우에는 *Value*를 공백으로 둘 수 있습니다.

런타임 시 매개변수 제공

런타임 시 매개변수를 만들기 위해 다음을 사용할 수 있습니다.

- *ParamByName* 메소드를 사용하여 이름을 기반으로 매개변수에 값을 할당합니다 (*TADOQuery*에서는 사용할 수 없음).
- *Params::Items* 또는 *Parameters::Items* 속성을 사용하여 SQL 문 내에서 매개변수의 위치 상의 순서를 기반으로 매개변수에 값을 할당합니다.
- *Params::ParamValues* 또는 *Parameters::ParamValues* 속성은 각 매개변수 집합의 이름을 기반으로 단일 명령 줄의 하나 이상의 매개변수에 대한 값을 할당합니다.

다음 코드에서는 *ParamByName*을 사용하여 :Capital 매개변수에 에디트 박스의 텍스트를 할당합니다.

```
SQLQuery1->ParamByName("Capital")->AsString = Edit1->Text;
```

인덱스를 0(:Capital 매개변수가 SQL 문의 첫 번째 매개변수라고 가정)으로 하여 *Params* 속성을 사용하는 동일한 코드를 다시 작성할 수 있습니다.

```
SQLQuery1->Params->Items[0]->AsString = Edit1->Text;
```

아래의 명령 줄은 *Params::ParamValues* 속성을 사용하여 한 번에 세 개의 매개변수를 설정합니다.

```
Query1->Params->ParamValues["Name;Capital;Continent"] =
  VarArrayOf(OPENARRAY(Variant, (Edit1->Text, Edit2->Text, Edit3-
>Text)));
```

*ParamValues*가 값을 변환하지 않아도 되는 Variants를 사용한다는 점에 유의하십시오.

매개변수를 사용하여 마스터/디테일 관계 설정

디테일 셋이 쿼리 타입 데이터셋인 마스터 / 디테일 관계를 설정하려면 매개변수를 사용하는 쿼리를 지정해야 합니다. 이들 매개변수는 마스터 데이터셋의 현재 필드 값을 참조합니다. 마스터 데이터셋의 현재 필드 값은 런타임 시 동적으로 변경되기 때문에, 마스터 레코드가 변경될 때마다 디테일 셋의 매개변수를 다시 연결해야 합니다. 이벤트 핸들러를 사용하여 이러한 작업을 하는 코드를 기록할 수도 있지만 *TIBQuery*를 제외한 모든 쿼리 타입 데이터셋에서는 *DataSource* 속성을 사용하여 좀 더 쉬운 메커니즘을 제공합니다.

매개변수화된 쿼리의 매개변수 값이 디자인 타임 시 연결되지 않거나 런타임 시 지정되지 않을 경우, 쿼리 타입 데이터셋은 *DataSource* 속성에 따라 이에 대한 값을 제공하려고 시도합니다. *DataSource*는 연결되지 않은 매개변수의 이름과 일치하는 필드 이름으로 검색된 다른 데이터셋을 식별합니다. 이러한 검색 데이터셋은 어떤 데이터셋이든 가능합니다. 검색 데이터셋은 이를 사용하는 디테일 데이터셋을 작성하기 전에 만들어져 채워져야 합니다. 검색 데이터셋에서 일치하는 항목을 찾으면, 디테일 데이터셋은 매개변수 값을 데이터 소스에 의해 지시된 현재 레코드의 필드 값에 연결합니다.

이러한 방법을 설명하기 위해 두 개의 테이블, Customer 테이블과 Orders 테이블을 고려해 봅니다. 모든 고객에 대해, Orders 테이블은 고객이 만드는 주문 집합을 포함합니다. Customer 테이블은 고유한 고객 ID를 지정하는 ID 필드를 포함합니다. Orders 테이블은 주문을 작성한 고객의 ID를 지정하는 CustID 필드를 포함합니다.

첫 번째 단계는 다음과 같이 Customer 데이터셋을 설정하는 것입니다.

1. 테이블 타입 데이터셋을 애플리케이션에 추가한 다음 Customer 테이블에 연결합니다.
2. 이름이 *CustomerSource*인 *TDataSource*를 추가합니다. *DataSet* 속성을 단계 1에서 추가한 데이터셋으로 설정합니다. 이 데이터 소스는 이제 Customer 데이터셋을 나타냅니다.
3. 쿼리 타입 데이터셋을 추가하고 *SQL* 속성을 다음과 같이 설정합니다.

   ```
   SELECT CustID, OrderNo, SaleDate
   FROM Orders
   WHERE CustID = :ID
   ```

   매개변수 이름이 마스터(Customer) 테이블의 필드 이름과 동일하다는 것에 유의하십시오.
4. 디테일 데이터셋의 *DataSource* 속성을 *CustomerSource*로 설정합니다. 이 속성을 설정하면 디테일 데이터셋이 연결된 쿼리가 됩니다.

런타임 시 디테일 데이터셋의 SQL 문에 있는 *:ID* 매개변수에는 값이 할당되지 않습니다. 따라서 데이터셋은 *CustomersSource*로 인식되는 데이터셋의 열과 매개변수를 이름으로 비교하려고 합니다. *CustomersSource*는 마스터 데이터셋에서 데이터를 가져오며, 마스터 데이터셋은 다시 Customer 테이블에서 데이터를 파생시킵니다. Customer 테이블에는 "ID" 열이 있으므로 마스터 데이터셋의 현재 레코드에 있는 *ID* 필드의 값은 디테일 데이터셋 SQL 문의 *:ID* 매개변수에 할당됩니다. 데이터셋은 마스터/디테일 관계로 연결됩니다. Customers 데이터셋에서 현재 레코드가 변경될 때마다 디테일 데이터셋의 SELECT 문은 현재 고객의 ID에 따라 모든 주문을 검색합니다.

쿼리 준비

쿼리를 준비하는 것은 쿼리 실행 전에 이루어지는 옵션 단계입니다. 쿼리를 준비할 때 SQL 문과 매개변수가 있으면 분석과 리소스 할당 및 최적화를 위해 데이터 액세스 레이어와 데이터베이스 서버로 보냅니다. 일부 데이터셋의 경우 데이터셋에서 쿼리를 준비할 때 추가 설치 작업을 수행할 수도 있습니다. 이러한 작업은 업데이트 가능한 쿼리를 사용하는 경우 특히 애플리케이션 속도를 증가시키면서 쿼리 성능을 향상시킵니다.

애플리케이션에는 *Prepared* 속성을 **true**로 설정하여 쿼리를 준비할 수 있습니다. 쿼리를 실행하기 전에 쿼리를 준비하지 않으면 데이터셋에서는 *Open*이나 *ExecSQL*을 호출할 때마다 개발자를 위해 자동으로 준비합니다. 데이터셋에서 쿼리를 대신 준비하기는 하지만 처음으로 데이터셋을 열기 전에 명시적으로 데이터셋을 준비하는 것이 성능을 개선시킬 수 있습니다.

```
CustQuery->Prepared = true;
```

데이터셋을 명시적으로 준비하면, *Prepared*를 **false**로 설정하기 전에는 구문 실행을 위해 할당된 리소스가 해제되지 않습니다.

데이터셋에서 실행하기 전에(예를 들어, 매개변수를 추가하는 경우) 데이터셋이 다시 준비되게 하려면 *Prepared* 속성을 **false**로 설정합니다.

참고 쿼리에 대한 *SQL* 속성의 텍스트를 변경하면 데이터셋은 자동으로 닫히고 쿼리 준비를 취소합니다.

결과 집합을 반환하지 않는 쿼리 실행

쿼리(예: SELECT 쿼리)에서 레코드 집합을 반환하는 경우, *Active*를 **true**로 설정하거나 *Open* 메소드를 호출하여 레코드로 데이터셋을 채우는 방식으로 쿼리를 실행합니다.

하지만 SQL 명령에서 레코드를 반환하지 않는 경우도 있습니다. 이러한 명령은 SELECT 문(예를 들면 INSERT, DELETE, UPDATE, CREATE INDEX 및 ALTER TABLE 명령은 어떤 레코드도 반환하지 않음)이 아닌 데이터 정의 언어(DDL) 또는 데이터 조작 언어(DML) 문을 사용하는 명령문을 포함합니다.

모든 쿼리 타입 데이터셋의 경우 다음과 같이 *ExecSQL*을 호출하여 결과 집합을 반환하지 않는 쿼리를 실행할 수 있습니다.

```
CustomerQuery->ExecSQL(); // Does not return a result set
```

팁 쿼리를 여러 번 실행해야 하는 경우 *Prepared* 속성을 **true**로 설정하는 것이 좋습니다.

쿼리에서 레코드를 반환하지 않지만 쿼리의 영향을 받는 레코드 수(예를 들어, DELETE 쿼리로 삭제되는 레코드 수)는 알아야 할 것입니다. *RowsAffected* 속성은 *ExecSQL*을 호출한 후에 영향을 받은 레코드 수를 제공합니다.

팁 디자인 타임 시, 쿼리에서 결과 집합을 반환할 지 여부를 모르는 경우(예를 들어, 사용자가 쿼리를 런타임 시 동적으로 입력하는 경우), 양쪽 타입의 쿼리 실행 문을 **try...catch** 블록 내에 코딩할 수 있습니다. **try** 절에 *Open* 메소드에 대한 호출을 둡니다. *Open* 메소드와 함께 쿼리가 활성화되면 실행 쿼리가 실행되지만 그와 함께 예외가 발생합니다. 예외를 검사하여 결과 집합이 없음을 표시할 뿐이면 예외를 표시하지 않습니다. 예를 들어, *TQuery*는 이러한 내용을 *ENoResultSet* 예외를 사용하여 표시합니다.

단방향 결과 집합 사용

쿼리 타입 데이터셋에서 결과 집합을 반환할 때 결과 집합의 첫 레코드에 대한 커서나 포인터도 받습니다. 커서가 가리키는 레코드는 현재 활성 중인 레코드입니다. 현재 레코드는 필드 값이 결과 집합의 데이터 소스와 연결된 데이터 인식 컴포넌트에 표시된 레코드입니다.
dbExpress를 사용하는 경우가 아니라면 이 커서는 디폴트로 양방향입니다. 양방향 커서는 레코드를 앞과 뒤로 모두 탐색할 수 있습니다. 양방향 커서를 지원하려면 약간의 추가 처리 오버헤드가 필요하므로 일부 쿼리의 속도가 늦어질 수 있습니다.

결과 집합을 역으로 탐색할 필요가 없는 경우, *TQuery*와 *TIBQuery*에서 대신 단방향 커서를 요청하여 쿼리 성능을 향상시킬 수 있습니다. 단방향 커서를 요청하려면 *UniDirectional* 속성을 **true**로 설정합니다.

쿼리를 준비하고 실행하기 전에 *UniDirectional*을 설정합니다. 다음 코드는 쿼리를 준비하고 실행하기 전에 *UniDirectional*을 설정하는 것을 보여 줍니다.

```
if (!CustomerQuery->Prepared)
{
  CustomerQuery->UniDirectional = true;
  CustomerQuery->Prepared = true;
}
CustomerQuery->Open(); // Returns aresult set with a one-way cursor
```

참고 *UniDirectional* 속성을 단방향 데이터셋과 혼동하지 마십시오. 단방향 데이터셋(*TSQLDataSet*, *TSQLTable*, *TSQLQuery* 및 *TSQLStoredProc*)은 단방향 커서만 반환하는 dbExpress를 사용합니다. 단방향 데이터셋은 역으로 탐색하는 기능을 제한할 뿐 아니라 레코드를 버퍼링하지 않으므로 추가 제한(필터를 사용할 수 없음)이 있습니다.

내장 프로시저 타입의 데이터셋 사용

애플리케이션에서 내장 프로시저를 사용하는 방법은 내장 프로시저가 코딩된 방법, 데이터의 반환 여부 및 반환 방법, 사용하는 특정 데이터베이스 서버 또는 이러한 요소의 조합에 따라 달라집니다.

일반적으로 내장 프로시저에 액세스하려면 애플리케이션에서는 다음 작업을 수행해야 합니다.

1. 해당 데이터셋 컴포넌트를 데이터 모듈이나 폼에 배치하고 *Name* 속성을 애플리케이션에 적합한 고유 값으로 설정합니다.
2. 내장 프로시저를 정의하는 데이터베이스 서버를 식별합니다. 내장 프로시저 타입 데이터셋별로 이러한 식별 방법은 다르지만 일반적으로 다음과 같이 데이터베이스 연결 컴포넌트를 지정하면 됩니다
   - *TStoredProc*에서는 *DatabaseName* 속성을 사용하여 *TDatabase* 컴포넌트나 BDE 알리아스를 지정합니다.
   - *TADOStoredProc*에서는 *Connection* 속성을 사용하여 *TADOConnection* 컴포넌트를 지정합니다.

- *TSQLStoredProc*에서는 *SQLConnection* 속성을 사용하여 *TSQLConnection* 컴포넌트를 지정합니다.
- *TIBStoredProc*에서는 *Database* 속성을 사용하여 *TIBConnection* 컴포넌트를 지정합니다.

데이터베이스 연결 컴포넌트에 대한 자세한 내용은 21장, "데이터베이스에 연결"을 참조하십시오.

3. 실행할 내장 프로시저를 지정합니다. 내장 프로시저 타입의 데이터셋에서는 대부분 *StoredProcName* 속성을 설정하여 이를 지정할 수 있습니다. 단, 이 속성 대신 *ProcedureName* 속성을 갖는 *TADOStoredProc*는 예외입니다.
4. 비주얼(visual) 데이터 컨트롤과 함께 사용되는 커서를 내장 프로시저에서 반환하면, 데이터 소스 컴포넌트를 데이터 모듈에 추가한 다음 *DataSet* 속성을 내장 프로시저 타입의 데이터셋으로 설정합니다. *DataSource* 및 *DataField* 속성을 사용하여 데이터 인식 컴포넌트를 데이터 소스에 연결합니다.
5. 필요하면 내장 프로시저의 입력 매개변수 값을 입력합니다. 서버에서 모든 내장 프로시저 매개변수에 대한 정보를 제공하지 않으면 매개변수 이름과 데이터 타입과 같은 추가 입력 매개변수를 제공해야 합니다. 내장 프로시저 매개변수 사용에 대한 자세한 내용은 22-49페이지의 "내장 프로시저 매개변수 사용"을 참조하십시오.
6. 내장 프로시저를 실행합니다. 커서를 반환하는 내장 프로시저의 경우 *Active* 속성이나 *Open* 메소드를 사용합니다. 결과 집합을 반환하지 않거나 출력 매개변수만 반환하는 내장 프로시저를 실행하려면 런타임 시 *ExecProc* 메소드를 사용합니다. 두 번 이상 내장 프로시저를 실행할 계획이면 *Prepare*를 호출하여 데이터 액세스 레이어를 초기화하고 매개변수 값을 내장 프로시저에 연결할 수 있습니다. 내장 프로시저 실행에 대한 자세한 내용은 22-52페이지의 "결과 집합을 반환하지 않는 내장 프로시저 실행"을 참조하십시오.
7. 결과를 처리합니다. 이러한 결과는 결과 및 출력 매개변수로 반환되거나 내장 프로시저 타입 데이터셋을 채우는 결과 집합으로 반환될 수 있습니다. 일부 내장 프로시저는 여러 커서를 반환합니다. 추가 커서에 액세스하는 방법에 대한 자세한 내용은 22-53페이지의 "여러 결과 집합 가져오기"를 참조하십시오.

내장 프로시저 매개변수 사용

내장 프로시저와 연결될 수 있는 매개변수 타입은 네 가지가 있습니다.

- *입력 매개변수*는 처리를 위해 내장 프로시저에 값을 전달할 때 사용됩니다.
- *출력 매개변수*는 내장 프로시저에서 애플리케이션에 반환값을 전달하기 위해 사용됩니다.
- *입/출력 매개변수*는 처리를 위해 내장 매개변수에 값을 전달할 때 사용되며, 내장 프로시저에서 애플리케이션에 반환값을 전달하기 위해 사용됩니다.
- *결과 매개변수*는 일부 내장 프로시저에서 애플리케이션에 오류나 상태값을 반환하기 위해 사용됩니다. 내장 프로시저는 하나의 결과 매개변수만 반환할 수 있습니다.

내장 프로시저에서 특정 타입의 매개변수를 사용할지는 데이터베이스 서버에 있는 내장 프로시저의 일반적인 랭귀지 구현과 내장 프로시저의 특정 인스턴스에 따라 달라집니다. 서버에서는 각각의 내장 프로시저에서 입력 매개변수를 사용할 수도 있고 사용하지 않을 수도 있습니다. 반면에 일부 매개변수는 서버마다 다르게 사용됩니다. 예를 들어, MS-SQL Server와 Sybase에서 내장 프로시저는 항상 결과 매개변수를 반환하지만 InterBase에서 내장 프로시저를 구현할 때는 결과 매개변수를 반환하지 않습니다.

내장 프로시저 매개변수에 대한 액세스는 *Params* 속성(*TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*에서) 또는 *Parameters* 속성 (*TADOStoredProc*에서)에서 제공합니다. *StoredProcName*(또는 *ProcedureName*) 속성에 값을 할당하면 데이터셋은 내장 프로시저의 각 매개변수에 대해 객체를 자동으로 생성합니다. 일부 데이터셋에서 내장 프로시저 이름을 런타임까지 지정하지 않았으면 각 매개변수의 객체를 런타임에 프로그래밍 방식으로 만들어야 합니다. 내장 프로시저를 지정하지 않고 *TParam*이나 *TParameter* 객체를 수동으로 만들면 많은 내장 프로시저에서 단일 데이터셋을 사용할 수 있습니다.

참고 일부 내장 프로시저는 출력 및 결과 매개변수 이외에 데이터셋을 반환합니다. 애플리케이션은 데이터 인식 컨트롤에서 데이터셋 레코드를 표시할 수 있지만 출력과 결과 매개변수를 각각 처리해야 합니다.

디자인 타임 시 매개변수 설정

매개변수 컬렉션 에디터를 사용하여 디자인 타임 시 내장 프로시저 매개변수 값을 지정할 수 있습니다. 매개변수 컬렉션 에디터를 표시하려면 Object Inspector에서 *Params* 또는 *Parameters* 속성의 생략 부호 버튼을 클릭합니다.

중요 매개변수 컬렉션 에디터에서 값을 선택하고 Object Inspector를 사용하여 *Value* 속성을 설정하여 입력 매개변수에 값을 할당할 수 있습니다. 그러나 서버에서 보고된 입력 매개변수에 대한 이름이나 데이터 타입은 변경하지 마십시오. 그렇지 않으면 내장 프로시저를 실행할 때 예외가 발생할 수 있습니다.

일부 서버는 매개변수 이름 또는 데이터 타입을 보고하지 않습니다. 이러한 경우에는 매개변수 컬렉션 에디터를 사용하여 매개변수를 직접 설정해야 합니다. 오른쪽 버튼을 클릭하고 Add를 선택하여 매개변수를 추가합니다. 추가하는 각 매개변수에 대해 매개변수를 자세하게 설명해야 합니다. 매개변수를 추가할 필요가 없더라도 개별 매개변수 객체의 속성을 검사하여 올바른지 확인해야 합니다.

데이터셋에 *Params* 속성(*TParam* 객체)이 있는 경우 다음 속성을 올바르게 지정해야 합니다.

- *Name* 속성은 내장 프로시저에서 정의한 대로 매개변수의 이름을 나타냅니다.
- *DataType* 속성은 매개변수 값의 데이터 타입을 제공합니다. *TSQLStoredProc*을 사용할 때 일부 데이터 타입에는 다음과 같은 추가 정보가 필요합니다.
  - *NumericScale* 속성은 숫자 매개변수의 소수 자릿수를 나타냅니다.
  - *Precision* 속성은 숫자 매개변수의 전체 자릿수를 나타냅니다.
  - *Size* 속성은 문자열 매개변수의 문자 수를 나타냅니다.
- *ParamType* 속성은 선택한 매개변수 타입을 나타냅니다. 이러한 타입에는 *ptInput*(입력 매개변수의 경우), *ptOutput*(출력 매개변수의 경우), *ptInputOutput*(입/출력 매개변수의 경우) 또는 *ptResult*(결과 매개변수의 경우)가 있을 수 있습니다.

- *Value* 속성은 선택한 매개변수의 값을 지정합니다. 출력 매개변수와 결과 매개변수에 대한 값은 설정할 수 없습니다. 이러한 타입의 매개변수는 내장 프로시저 실행에 의해 설정된 값을 가집니다. 입력 및 입/출력 매개변수의 경우, 애플리케이션에서 런타임 시 매개변수를 입력하면 *Value*를 비워 둘 수 있습니다.

데이터셋에서 *Parameters* 속성(*TParameter* 객체)을 사용하는 경우 다음 속성을 올바르게 지정해야 합니다.

- *Name* 속성은 내장 프로시저에서 정의한 대로 매개변수 이름을 표시합니다.
- *DataType* 속성은 매개변수 값에 대한 데이터 타입을 제공합니다. 일부 데이터 타입에서는 다음과 같은 추가 정보를 제공해야 합니다.
  - *NumericScale* 속성은 숫자 매개변수의 소수 자릿수를 나타냅니다.
  - *Precision* 속성은 숫자 매개변수의 전체 자릿수를 나타냅니다.
  - *Size* 속성은 문자열 매개변수의 문자 수를 나타냅니다.
- *Direction* 속성은 선택한 매개변수의 타입을 제공합니다. 이러한 타입에는 *pdInput*(입력 매개변수의 경우), *pdOutput*(출력 매개변수의 경우), *pdInputOutput*(입/출력 매개변수의 경우) 또는 *pdReturnValue*(결과 매개변수의 경우)가 있을 수 있습니다.
- *Attributes* 속성은 매개변수에서 받아들일 값 타입을 제어합니다. *Attributes*는 *psSigned*, *psNullable* 및 *psLong*의 조합으로 설정될 수 있습니다.
- *Value* 속성은 선택한 매개변수의 값을 지정합니다. 출력 및 결과 매개변수의 값은 설정하지 마십시오. 입력 및 입/출력 매개변수의 경우 애플리케이션에서 런타임 시 매개변수를 입력하면 *Value*를 비워 둘 수 있습니다.

런타임 시 매개변수 사용

일부 데이터셋에서 내장 프로시저 이름을 런타임까지 지정하지 않았으면 매개변수에 대해 *TParam* 객체가 자동으로 만들어지지 않으므로 객체를 프로그래밍 방식으로 만들어야 합니다. 다음과 같이 새 *TParam* 객체나 *TParams::AddParam* 메소드를 인스턴스화하여 객체를 프로그래밍 방식으로 만들 수 있습니다.

```
TParam *P1, *P2;
StoredProc1->StoredProcName = "GET_EMP_PROJ";
StoredProc1->Params->Clear();
P1 = new TParam(StoredProc1->Params, ptInput);
P2 = new TParam(StoredProc1->Params, ptOutput);
try
{
  StoredProc1->Params->Items[0]->Name = "EMP_NO";
  StoredProc1->Params->Items[1]->Name = "PROJ_ID";
  StoredProc1->ParamByName("EMP_NO")->AsSmallInt = 52;
  StoredProc1->ExecProc();
  Edit1->Text = StoredProc1->ParamByName("PROJ_ID")->AsString;
}
__finally
{
  delete P1;
  delete P2;
}
```

개별 매개변수 객체를 런타임 시 추가할 필요가 없더라도 개별 매개변수 객체에 액세스하여 입력 매개변수에 값을 할당하고 출력 매개변수로부터 값을 검색해야 하는 경우가 있습니다. 데이터셋의 *ParamByName* 메소드를 사용하여 매개변수 이름을 기반으로 개별 매개변수에 액세스할 수 있습니다. 예를 들어, 다음 코드는 입/출력 매개변수의 값을 설정하고, 내장 프로시저를 실행하며 반환된 값을 검색합니다.

```
SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger = 103;
SQLDataSet1->ExecSQL();
int Result = SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger;
```

내장 프로시저 준비

쿼리 타입 데이터셋의 경우처럼 내장 프로시저를 실행하기 전에 내장 프로시저 타입 데이터셋을 준비해야 합니다. 내장 프로시저를 준비하면 내장 프로시저의 리소스를 할당하고 매개변수를 연결해야 함을 데이터 액세스 레이어와 데이터베이스 서버에게 알립니다. 이러한 작업은 성능을 향상시킬 수 있습니다.

개발자가 내장 프로시저를 준비하기 전에 실행하려고 시도하는 경우, 데이터셋에서는 개발자를 위해 자동으로 내장 프로시저를 준비하여 실행한 다음 준비를 취소합니다. 내장 프로시저를 여러 번 실행할 계획이면 Prepared 속성을 **true**로 설정하여 내장 프로시저를 명시적으로 준비하는 것이 더 효율적입니다.

```
MyProc->Prepared = true;
```

데이터셋을 명시적으로 준비하면, *Prepared*를 **false**로 설정하기 전에는 내장 프로시저 실행을 위해 할당된 리소스가 해제되지 않습니다.

데이터셋이 실행되기 전에 다시 준비되는지 확인하려면(예를 들어, Oracle 오버로드된 프로시저를 사용할 때 매개변수를 변경하는 경우), *Prepared* 속성을 **false**로 설정합니다.

결과 집합을 반환하지 않는 내장 프로시저 실행

내장 프로시저에서 커서를 반환하는 경우 *Active*를 **true**로 설정하거나 *Open* 메소드를 호출하여 레코드로 데이터셋을 채우는 방식으로 내장 프로시저를 실행합니다

하지만 내장 프로시저에서는 가끔 데이터를 반환하지 않거나 결과를 출력 매개변수로만 반환합니다. *ExecProc*을 호출하여 결과 집합을 반환하지 않는 내장 프로시저를 실행할 수 있습니다. 다음과 같이 내장 프로시저를 실행한 다음 *ParamByName* 메소드를 사용하여 결과 매개변수의 값이나 출력 매개변수의 값을 읽을 수 있습니다.

```
MyStoredProcedure->ExecProc(); // Does not return a result set
Edit1->Text = MyStoredProcedure->ParamByName("OUTVAR")->AsString;
```

참고 *TADOStoredProc*에는 *ParamByName* 메소드가 없습니다. ADO를 사용할 때 출력 매개변수를 얻으려면 *Parameters* 속성을 사용하여 매개변수 객체에 액세스해야 합니다.

팁 프로시저를 여러 번 실행하는 경우 *Prepared* 속성을 **true**로 설정하는 것이 좋습니다.

여러 결과 집합 가져오기

일부 내장 프로시저는 레코드의 여러 집합을 반환합니다. 데이터셋은 개발자가 이를 열 때 첫 번째 집합만 가져옵니다. *TSQLStoredProc*이나 *TADOStoredProc*을 사용하는 경우 *NextRecordSet*이나 *NextRecordset* 메소드를 호출하여 다른 레코드 집합에 액세스할 수 있습니다.

```
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

*TSQLStoredProc*에서 *NextRecordSet*은 다음 레코드 집합에 대한 액세스를 제공하는 새로 생성된 *TCustomSQLDataSet* 컴포넌트를 반환합니다. *TADOStoredProc*에서 *NextRecordset*은 기존 ADO 데이터셋의 *RecordSet* 속성에 할당할 수 있는 인터페이스를 반환합니다. 양쪽 클래스에서 모두, 해당 메소드는 반환된 데이터셋의 레코드 수를 출력 매개변수로 반환합니다.

처음으로 *NextRecordSet*이나 *NextRecordset*을 호출하면 두 번째 레코드 집합을 반환합니다. *NextRecordSet*이나 *NextRecordset*을 다시 호출하면 세 번째 데이터셋을 반환하며, 이러한 작업은 더 이상 남은 레코드 집합이 없을 때까지 반복됩니다. 더 이상 추가 커서가 없으면 *NextRecordSet*이나 *NextRecordset*에서 NULL을 반환합니다.

장

23

필드 컴포넌트 작업

이 장에서는 *TField* 객체와 그 자손이 공통적으로 사용하는 속성, 이벤트 및 메소드를 설명합니다. 필드 컴포넌트는 데이터셋의 개별 필드(열)를 나타냅니다. 또한 필드 컴포넌트를 사용하여 애플리케이션에서 데이터를 표시 및 편집하는 것을 제어하는 방법을 설명합니다.

필드 컴포넌트는 항상 데이터셋과 연결되어 있습니다. *TField* 객체는 개발자의 애플리케이션에서 직접 사용하지 않습니다. 대신, 애플리케이션의 각 필드 컴포넌트는 데이터셋에 있는 열의 데이터 타입에 해당하는 *TField* 자손입니다. 필드 컴포넌트는 연결된 데이터셋의 특정 열의 데이터에 대한 *TDBEdit* 및 *TDBGrid* 액세스와 같이 데이터 인식 컨트롤을 제공합니다.

일반적으로, 단일 필드 컴포넌트는 데이터 타입 및 크기와 같은 데이터셋의 단일 열이나 필드의 특성을 나타냅니다. 단일 필드 컴포넌트는 또한 정렬, 표시 형식 및 편집 형식과 같은 필드의 표시 특성을 나타냅니다. 예를 들어, *TFloatField* 컴포넌트에는 데이터 표시에 직접적으로 영향을 주는 속성이 네 가지 있습니다.

표 23.1 데이터 표시에 영향을 주는 TFloatField 속성

속성	용도
Alignment	왼쪽 정렬, 가운데 정렬, 오른쪽 정렬의 데이터 표시 방법을 지정합니다.
DisplayWidth	한 번에 컨트롤에 표시할 수 있는 숫자 자릿수를 지정합니다.
DisplayFormat	소수점 이하 자릿수를 얼마나 표시할 것인가와 같이 표시할 데이터의 서식을 지정합니다.
EditFormat	편집 중에 값을 표시하는 방법을 지정합니다.

데이터셋의 레코드에서 레코드로 스크롤할 때, 필드 컴포넌트를 사용하면 현재 레코드의 필드 값을 보거나 변경할 수 있습니다.

필드 컴포넌트는 *DisplayWidth* 및 *Alignment*와 같이 서로 공통적인 속성을 많이 갖고 있으며 *TFloatField*에 대한 *Precision*과 같이 데이터 타입에 특정한 속성도 갖고 있습니다. 이들 각 속성은 폼에서 애플리케이션 사용자에게 데이터가 나타나는 방식에 영향을 줍니다. 또한 *Precision*과 같은 일부 속성은 데이터 수정이나 입력 시 사용자가 컨트롤에 입력할 수 있는 데이터 값에도 영향을 줍니다.

데이터셋의 모든 필드 컴포넌트는 *동적*(원본으로 사용하는 데이터베이스 테이블의 구조에 따라 자동으로 생성)이거나 *영구적*(Fields Editor에서 설정하는 특정 필드 이름과 속성에 따라 생성)입니다. 동적 필드와 영구적 필드(persistent field)는 각각의 장점을 갖고 있으며 각기 다른 애플리케이션 타입에 적당합니다. 다음 단원에서는 동적 필드와 영구적 필드를 좀더 자세히 설명하고, 둘 중 어느 필드를 선택해야 할 지에 대해 조언을 제공합니다.

동적 필드 컴포넌트

필드 컴포넌트는 디폴트로 동적으로 생성됩니다. 사실상, 데이터셋을 데이터 모듈에 처음으로 배치한 다음 데이터셋이 데이터를 가져오고 여는 방법을 지정하면, 데이터셋의 모든 필드 컴포넌트는 동적 필드가 됩니다. 데이터셋이 나타내는 데이터의 물리적인 기본 특성을 바탕으로 자동으로 생성되는 필드 컴포넌트는 *동적* 필드 컴포넌트입니다. 데이터셋은 원본으로 사용하는 데이터의 각 열에 대해 하나의 필드 컴포넌트를 생성합니다. 각 열에 대해 만들어지는 정확한 *TField* 자손은 데이터베이스 또는 *TClientDataSet*의 경우 프로바이더 컴포넌트로부터 받은 필드 타입 정보에 의해 결정됩니다.

동적 필드는 임시 필드입니다. 동적 필드는 데이터셋이 열려 있는 동안에만 존재합니다. 동적 필드를 사용하는 데이터셋을 다시 열 때마다, 데이터셋의 원본으로 사용하는 데이터의 현재 구조에 따라 완전히 새로운 동적 필드 컴포넌트 집합을 다시 만듭니다. 원본으로 사용하는 데이터의 열이 변경된 후 동적 필드 컴포넌트를 사용하는 데이터셋을 다시 열면, 자동으로 생성되는 필드 컴포넌트도 이를 반영하기 위해 변경됩니다.

데이터 표시 및 편집 작업을 융통성 있게 수행해야 하는 애플리케이션에 동적 필드를 사용해야 합니다. 예를 들어, SQL 탐색기와 같은 데이터베이스 찾아보기 도구를 만들려면, 모든 데이터베이스 테이블은 열의 숫자와 타입이 다르기 때문에 동적 필드를 사용해야 합니다. 또한 사용자의 데이터와의 상호 작용이 대부분 그리드 컴포넌트 내에서 발생하고, 애플리케이션에서 사용하는 데이터셋이 자주 변경되는 애플리케이션의 경우에도 동적 필드를 사용합니다.

애플리케이션에서 동적 필드를 사용하려면 다음과 같이 합니다.

1. 데이터셋과 데이터 소스를 데이터 모듈에 배치합니다.
2. 데이터셋을 데이터에 연결합니다. 이 과정에서 연결 컴포넌트나 프로바이더를 사용하여 데이터 소스에 연결하게 되며 데이터셋이 어떤 데이터를 나타낼 지 지정하는 속성을 설정할 수 있습니다.
3. 데이터 소스를 데이터셋에 연결합니다.
4. 데이터 인식 컨트롤을 애플리케이션의 폼에 배치하고, 데이터 모듈의 헤더를 각 폼의 유닛에 포함시킨 다음 각 데이터 인식 컨트롤을 모듈의 데이터 소스와 연결합니다. 또한 필드를 필요로 하는 각각의 데이터 인식 컨트롤에 필드를 연결합니다. 동적 필드 컴포넌트를 사용하기 때문에, 데이터셋을 열 때 지정한 필드 이름이 계속 남아 있지 않을 수도 있습니다.
5. 데이터셋을 엽니다.

사용이 간편한 반면 동적 필드의 사용은 제한적입니다. 코드를 작성하지 않으면, 동적 필드에 대한 표시 및 편집 기본값을 변경할 수 없고, 동적 필드가 표시되는 순서를 안전하게 변경할 수 없으며, 데이터셋의 모든 필드에 대한 액세스를 방지할 수 없습니다. 계산된 필드나 조회 필드와 같이 데이터셋에 대한 추가 필드를 만들 수 없고, 동적 필드의 디폴트 데이터 타입을 오버라이드할 수 없습니다. 데이터베이스 애플리케이션에서 필드를 조정하고 필드에 대한 유동성을 얻으려면, Fields Editor를 호출하여 데이터셋에 대한 영구적 필드(persistent field) 컴포넌트를 만들어야 합니다.

영구적 필드(persistent field) 컴포넌트

디폴트로, 데이터셋 필드는 동적입니다. 필드의 속성과 가용성은 자동으로 설정되고 어떠한 방식으로든 변경될 수 없습니다. 필드의 속성과 이벤트를 제어하려면 데이터셋에 대해 영구적 필드를 만들어야 합니다. 영구적 필드를 사용하면 다음을 수행할 수 있습니다.

- 디자인 타임 또는 런타임 시 필드의 표시 및 편집 특성을 설정하거나 변경합니다.
- 데이터셋의 기존 필드의 값을 바탕으로 조회 필드, 계산된 필드 및 집계 필드와 같은 새 필드를 만듭니다.
- 데이터 입력을 검증합니다.
- 영구적 컴포넌트 리스트에서 필드 컴포넌트를 제거하여 원본으로 사용하는 데이터베이스의 특정 열에 애플리케이션이 액세스하지 못하도록 합니다.
- 데이터셋의 원본으로 사용하는 테이블이나 쿼리의 열에 따라 새 필드를 정의하며 기존 필드를 바꿉니다.

디자인 타임 시 애플리케이션의 데이터셋에서 사용하는 필드 컴포넌트의 영구적 리스트를 만들려면 Fields Editor를 사용해야 합니다. 영구적 필드 컴포넌트 리스트는 애플리케이션에 저장되며, 데이터셋의 원본으로 사용하는 데이터베이스 구조가 변경되더라도 변경되지 않습니다. 일단 Fields Editor로 영구적 필드를 만들면 데이터 값의 변경에 응답하고 데이터 입력을 검증하는 이벤트 핸들러를 만들 수 있습니다.

참고 데이터셋에 대한 영구적 필드를 만들 때 개발자가 선택하는 필드만이 디자인 타임 및 런타임 시 애플리케이션에서 사용될 수 있습니다. 디자인 타임에, 항상 Fields Editor를 사용하여 데이터셋에 대한 영구적 필드를 추가하거나 제거할 수 있습니다.

단일 데이터셋에서 사용하는 모든 필드는 영구적이거나 동적입니다. 단일 데이터셋에서 여러 필드 타입을 사용할 수는 없습니다. 데이터셋에 대한 영구적 필드를 만든 다음 동적 필드로 변환하려면, 데이터셋에서 모든 영구적 필드를 제거해야 합니다. 동적 필드에 대한 자세한 내용은 23-2페이지의 "동적 필드 컴포넌트"를 참조하십시오.

참고 영구적 필드의 주 용도 중 하나는 데이터의 모양과 표시를 제어하는 것입니다. 또한 데이터 인식 그리드의 열 모양을 조정할 수도 있습니다. 그리드의 열 모양 제어에 대한 내용은 19-16페이지의 "사용자 정의된 그리드 생성"을 참조하십시오.

영구적 필드(persistent field) 생성

Fields Editor로 만들어진 영구적 필드 컴포넌트는 원본으로 사용하는 데이터에 대해 효과적이고 타입이 안정적이며(type-safe) 읽기 쉬운 프로그램 액세스를 제공합니다. 영구적 필드 컴포넌트를 사용하면, 애플리케이션이 실행될 때마다 원본으로 사용하는 데이터베이스의 실제 구조가 변경되더라도 항상 동일한 순서로 동일한 열이 사용되고 표시됩니다. 특정 필드에 의존하는 데이터 인식 컴포넌트와 프로그램 코드는 항상 예상한 대로 작동합니다. 영구적 필드 컴포넌트의 기반이 되는 열이 삭제되거나 변경되면 , 존재하지 않는 열이나 일치하지 않는 데이터에 대해 애플리케이션을 실행하지 않고 C++Builder가 예외를 생성합니다.

다음과 같은 방법으로 데이터셋에 대한 영구적 필드를 만듭니다.

1. 데이터 모듈에 데이터셋을 둡니다.
2. 원본으로 사용하는 데이터에 데이터셋을 연결합니다. 일반적으로 이 과정에서 데이터셋이 연결 컴포넌트나 프로바이더에 연결되고 데이터를 설명하는 모든 속성이 지정됩니다. 예를 들어, *TADODataSet*을 사용하는 경우에는 적절히 구성된 *TADOConnection* 컴포넌트에 대한 *Connection* 속성을 설정하고 , 유효한 쿼리에 대한 *CommandText* 속성을 설정할 수 있습니다.
3. 데이터 모듈의 데이터셋 컴포넌트를 더블 클릭하여 Fields Editor를 호출합니다. Fields Editor에는 제목 표시줄, 탐색기 버튼 및 리스트 박스가 있습니다.

   Fields Editor의 제목 표시줄은 데이터 모듈 또는 데이터셋을 포함하는 폼의 이름과 데이터셋 자체의 이름을 모두 표시합니다. 예를 들어, *CustomerData* 데이터 모듈의 *Customers* 데이터셋을 열면, 제목 표시줄은 'CustomerData.>Customers' 또는 제목 표시줄에 들어가는 만큼 이름을 표시합니다.

   제목 표시줄 아래에는 디자인 타임에 활성 데이터셋의 레코드를 하나씩 스크롤하고 첫 번째나 마지막 레코드로 이동할 수 있도록 하는 탐색 버튼 집합이 있습니다. 데이터셋이 활성화되어 있지 않거나 비어 있으면 탐색 버튼이 흐리게 표시됩니다. 데이터셋이 단방향인 경우에는 마지막 레코드와 이전 레코드로 이동하는 버튼이 항상 흐리게 표시됩니다.

   리스트 박스에는 데이터셋에 대한 영구적 필드 컴포넌트의 이름을 표시합니다. 새 데이터셋에 대해 Fields Editor 를 처음 호출하면 데이터셋에 대한 필드 컴포넌트가 영구적이 아니고 동적이기 때문에 리스트가 비어 있습니다. 이미 영구적 필드 컴포넌트가 있는 데이터셋에 대해 Fields Editor를 호출하면, 리스트 박스에 필드 컴포넌트 이름이 표시됩니다.
4. Fields Editor 컨텍스트 메뉴에서 Add Fields를 선택합니다.
5. Add Fields 다이얼로그 박스에서 영구적으로 만들 필드를 선택합니다. 디폴트로, 다이얼로그 박스가 열릴 때에는 모든 필드가 선택되어 있습니다. 선택하는 모든 필드는 영구적 필드가 됩니다.

Add Fields 다이얼로그 박스가 닫히면, 개발자가 선택한 필드가 Fields Editor 리스트 박스에 나타납니다. Fields Editor 리스트 박스의 필드는 영구적입니다. 또한 데이터셋이 활성 상태이면, 리스트 박스 위의 다음 및 Last 탐색 버튼(데이터셋이 단방향이 아닌 경우)을 사용할 수 있습니다.

그리고 나면 데이터셋을 열 때마다, 원본으로 사용하는 데이터베이스의 모든 열에 대해 동적 필드 컴포넌트를 더 이상 만들지 않습니다. 그 대신, 개발자가 지정한 필드에 대한 영구적 컴포넌트만을 만듭니다.

데이터셋을 열 때마다, 각각의 계산되지 않은 영구적 필드(persistent field)가 데이터베이스에 있거나 데이터베이스의 데이터에서 만들어질 수 있는지 검증합니다 . 그렇지 않으면 데이터셋은 필드가 유효하지 않아 열리지 않는다는 것을 경고하는 예외를 발생시킵니다.

영구적 필드(persistent field) 정렬

영구적 필드 컴포넌트가 Fields Editor 리스트 박스에 나열되는 순서는 필드가 데이터 인식 그리드 컴포넌트에 나타나는 디폴트 순서입니다. 리스트 박스에서 필드를 끌어다 놓으면 필드 순서를 변경할 수 있습니다.

다음과 같은 방법으로 필드 순서를 변경합니다.

1 필드를 선택합니다. 한 번에 하나 이상의 필드를 선택하고 순서를 바꿀 수 있습니다.

2 필드를 새 위치로 끌어 놓습니다.

비연속적인 필드 집합을 선택하여 새로운 위치로 끌어 놓으면 연속적인 블록으로 삽입됩니다. 이 블록 내에서 필드의 순서는 변경되지 않습니다.

다른 방법으로는, 필드를 선택하고 *Ctrl+Up* 및 *Ctrl+Dn*을 사용하여 리스트에서 개별 필드의 순서를 변경할 수 있습니다.

영구적 새 필드 정의

기존의 데이터셋 필드를 영구적 필드로 만드는 것 외에도, 데이터셋의 다른 영구적 필드에 대한 추가 필드나 교체 필드로서 영구적 특수 필드를 만들 수 있습니다.

개발자가 만드는 새 영구적 필드는 표시용으로만 사용합니다. 이 필드가 런타임 시 포함하는 데이터는 이미 데이터베이스의 다른 곳에 있거나 임시적이기 때문에 유지되지 않습니다. 데이터셋의 원본으로 사용하는 데이터의 실제 구조는 어떤 방식으로든 변경되지 않습니다.

새 영구적 필드 컴포넌트를 만들려면, Fields Editor에서 컨텍스트 메뉴를 호출하여 New Field를 선택합니다. New Field 다이얼로그 박스가 나타납니다.

New Field 다이얼로그 박스는 세 가지 그룹 박스, 즉 Field properties, Field type 및 Lookup definition을 가지고 있습니다.

- Field properties 그룹 박스를 사용하면 일반적인 필드 컴포넌트 정보를 입력할 수 있습니다. Name 에디트 박스에 필드 이름을 입력합니다. 여기서 입력하는 이름은 필드 컴포넌트의 *FieldName* 속성과 일치합니다. New Field 다이얼로그 박스는 이 이름을 사용하여 Component 에디트 박스에 컴포넌트 이름을 만듭니다. Component 에디트 박스에 표시되는 이름은 필드 컴포넌트의 *Name* 속성과 일치하고, 오직 정보 제공의 목적을 위해서만 사용됩니다(*Name*은 개발자가 소스 코드의 필드 컴포넌트를 참조할 수 있도록 하는 식별자입니다). 다이얼로그 박스는 Component 에디트 박스에 직접 입력하는 모든 내용을 버립니다.

- Field properties 그룹의 Type 콤보 박스를 사용하면 필드 컴포넌트의 데이터 타입을 지정할 수 있습니다. 만드는 모든 새로운 필드 컴포넌트에 대해 데이터 타입을 제공해야 합니다. 예를 들어, 필드에 부동 소수점 통화 값을 표시하려면, 드롭다운 리스트에서 *Currency*를 선택합니다. Size 에디트 박스를 사용하여 문자열 기반 필드에 표시 또는 입력될 수 있는 문자의 최대 수나 *Bytes* 및 *VarBytes* 필드의 크기를 지정합니다 . 다른 모든 데이터 타입에서 Size는 무의미합니다.
- Field type 라디오 그룹을 사용하면 작성할 새로운 필드 컴포넌트의 타입을 지정할 수 있습니다. 디폴트 타입은 Data입니다. Lookup을 선택하면 Lookup Definition 그룹 박스의 Dataset 및 Source Fields 에디트 박스가 활성화됩니다. 또한 Calculated 필드를 만들 수 있으며, 클라이언트 데이터셋에서 작업하는 경우에는 InternalCalc 필드나 Aggregate 필드를 작성할 수도 있습니다. 다음 표는 생성할 수 있는 필드 타입을 설명한 것입니다.

표 23.2 영구적 특수 필드 종류

필드 종류	용도
Data	기존의 필드를 대체합니다(예를 들어, 데이터 타입 변경).
Calculated	런타임 시 데이터셋의 *OnCalcFields* 이벤트 핸들러로 계산된 값을 표시합니다.
Lookup	런타임 시 지정한 검색 기준을 기반으로 하여 지정된 데이터셋에서 값을 검색합니다(단방향 데이터셋에서는 지원되지 않음).
InternalCalc	런타임 시 클라이언트 데이터셋이 계산하고 해당 데이터와 함께 저장된 값을 표시합니다.
Aggregate	클라이언트 데이터셋에서 레코드 집합 데이터를 요약하는 값을 표시합니다.

Lookup definition 그룹 박스는 조회 필드를 만들기 위해서만 사용됩니다. 보다 자세한 설명은 23-8페이지의 "조회 필드 정의"를 참조하십시오.

데이터 필드 정의

데이터 필드는 데이터셋의 기존 필드를 바꿉니다. 예를 들어, 프로그래밍을 위해 *TSmallIntField*를 *TIntegerField*로 바꿔야 할 수도 있습니다. 필드의 데이터 타입을 직접 변경할 수 없기 때문에 이를 바꿀 새로운 필드를 정의해야 합니다.

중요 기존의 필드를 바꿀 새로운 필드를 정의하는 경우에도, 정의하는 필드는 데이터셋의 원본으로 사용하는 테이블에 있는 기존의 열에서 데이터 값을 파생시켜야 합니다.

데이터셋의 원본으로 사용하는 테이블에 있는 필드에 대한 교체 데이터 필드를 작성하려면, 다음 단계를 따르십시오.

1. 데이터셋에 할당된 영구적 필드 리스트에서 필드를 제거하고, 컨텍스트 메뉴에서 New Field를 선택합니다.
2. New Field 다이얼로그 박스에서 Name 에디트 박스의 데이터베이스 테이블에 있는 기존 필드의 이름을 입력합니다. 새로운 필드 이름을 입력하지 마십시오. 실제로는 새로운 필드가 데이터를 파생할 필드의 이름을 지정하는 것입니다.
3. Type 콤보 박스에서 필드의 데이터 타입을 선택합니다. 선택하는 데이터 타입은 바꾸는 필드의 데이터 타입과 달라야 합니다. 크기가 다른 문자열 필드는 서로 바꿀 수 없습니다. 데이터 타입은 달라야 하지만, 원본으로 사용하는 테이블에 있는 필드의 실제 데이터 타입과 호환되어야 합니다.

4 필요한 경우, Size 에디트 박스에 필드 크기를 입력합니다. 크기는 *TStringField*, *TBytesField* 및 *TVarBytesField*의 필드 타입에만 해당됩니다.

5 아직 선택되지 않았으면 Field type 라디오 그룹에서 Data를 선택합니다.

6 OK를 선택합니다. New Field 다이얼로그 박스가 닫히고, 새로 정의된 데이터 필드가 단계 1에서 지정한 기존 필드를 바꾸며, 데이터 모듈이나 폼의 헤더에 있는 컴포넌트 선언이 업데이트됩니다.

필드 컴포넌트와 연결된 속성 또는 이벤트를 편집하려면, Field Editor 리스트 박스에서 컴포넌트 이름을 선택한 다음 Object Inspector를 사용하여 속성 또는 이벤트를 편집합니다. 필드 컴포넌트 속성 및 이벤트의 편집에 대한 자세한 내용은 23-10페이지의 "영구적 필드 속성 및 이벤트 설정"을 참조하십시오.

계산된 필드 정의

계산된 필드는 런타임 시 데이터셋의 *OnCalcFields* 이벤트 핸들러로 계산된 값을 표시합니다. 예를 들어, 다른 필드에서 연결된 값을 표시하는 문자열 필드를 작성할 수 있습니다.

다음과 같은 방법으로 New Field 다이얼로그 박스에서 계산된 필드를 만듭니다.

1 Name 에디트 박스에 계산된 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.

2 Type 콤보 박스에서 필드의 데이터 타입을 선택합니다.

3 필요한 경우, Size 에디트 박스에 필드 크기를 입력합니다. 크기는 *TStringField*, *TBytesField* 및 *TVarBytesField*의 필드 타입에만 해당됩니다.

4 Field type 라디오 그룹에서 Calculated 또는 InternalCalc를 선택합니다. InternalCalc는 클라이언트 데이터셋에서 작업하는 경우에만 사용할 수 있습니다. 계산된 필드의 두 타입 간의 중요한 차이점은 InternalCalc 필드에 대해 계산된 값은 클라이언트 데이터셋의 데이터의 일부로서 저장 및 검색된다는 점입니다.

5 OK를 선택합니다. 새로 정의된 계산된 필드는 Field Editor 리스트 박스의 영구적 필드 리스트 끝에 자동으로 추가되며, 컴포넌트 선언은 폼 또는 데이터 모듈의 헤더에 자동으로 추가됩니다.

6 필드에 대해 값을 계산하는 코드를 데이터셋에 대한 *OnCalcFields* 이벤트 핸들러에 둡니다. 필드 값 계산을 위한 코드 작성에 대한 자세한 내용은 23-7페이지의 "계산된 필드 프로그래밍"을 참조하십시오.

참고 필드 컴포넌트와 연결된 속성 또는 이벤트를 편집하려면, Field Editor 리스트 박스에서 컴포넌트 이름을 선택한 다음 Object Inspector를 사용하여 속성 또는 이벤트를 편집합니다. 필드 컴포넌트 속성 및 이벤트 편집에 대한 자세한 내용은 23-10페이지의 "영구적 필드 속성 및 이벤트 설정"을 참조하십시오.

계산된 필드 프로그래밍

계산된 필드를 정의하고 나면 값을 계산하기 위한 코드를 작성해야 합니다. 그렇지 않으면 필드가 항상 Null 값을 가집니다. 계산된 필드의 코드는 데이터셋에 대한 *OnCalcFields* 이벤트에 둡니다.

다음과 같은 방법으로 계산된 필드에 대한 값을 프로그래밍합니다.

1 Object Inspector 드롭다운 리스트에서 데이터셋 컴포넌트를 선택합니다.

2 Object Inspector Events 페이지를 선택합니다.

3 *OnCalcFields* 속성을 더블 클릭하여 데이터셋 컴포넌트에 대한 *CalcFields* 프로시저를 불러오거나 만듭니다.

4 계산된 필드의 값과 속성을 설정하는 코드를 원하는 대로 작성합니다.

예를 들어, *CustomerData* 데이터 모듈의 *Customers* 테이블에 대한 *CityStateZip*이라는 계산된 필드를 만들었다고 가정합니다. *CityStateZip*은 데이터 인식 컨트롤에서 한 줄로 회사의 시/도와 우편 번호를 표시합니다.

Customers 테이블에 대한 *CalcFields* 프로시저에 코드를 추가하려면, Object Inspector 드롭다운 리스트에서 *Customers* 테이블을 선택하고, Events 페이지로 변환하여, *OnCalcFields* 속성을 더블 클릭합니다.

TCustomerData::CustomersCalcFields 프로시저는 유닛의 소스 코드 윈도우에 나타납니다. 다음 코드를 프로시저에 추가하여 필드를 계산합니다.

```
CustomersCityStateZip->Value = CustomersCity->Value + AnsiString(“, “) +
    CustomersState->Value + AnsiString(“ “) + CustomersZip->Value;
```

참고 내부적으로 계산된 필드에 대한 *OnCalcFields* 이벤트 핸들러를 작성할 때, 클라이언트 데이터셋의 *State* 속성을 확인하고 *State*가 *dsInternalCalc*인 경우에만 해당 값을 다시 계산하여 성능을 향상시킬 수 있습니다. 자세한 내용은 27-10페이지의 "클라이언트 데이터셋에서 내부적으로 계산된 필드 사용"을 참조하십시오.

조회 필드 정의

조회 필드는 지정한 검색 기준에 따라 런타임 시 값을 표시하는 읽기 전용 필드입니다. 가장 간단한 폼에서는, 조회 필드에 검색할 기존 필드의 이름과 검색할 필드 값 그리고 값을 표시해야 하는 조회 데이터셋의 다른 필드를 전달합니다.

예를 들어, 작업자가 조회 필드를 사용하여 고객이 제공하는 우편 번호와 일치하는 주소를 자동으로 결정할 수 있는 메일 주문 애플리케이션을 생각해 보십시오. 검색할 열은 *ZipTable->Zip*이고, 검색할 값은 *Order->CustZip*에 입력된 고객의 우편 번호이고, 반환할 값은 레코드의 *ZipTable->City* 및 *ZipTable->State* 열입니다. 여기서 *ZipTable->Zip*의 값은 *Order->CustZip* 필드의 현재 값과 일치합니다.

참고 단방향 데이터셋은 조회 필드를 지원하지 않습니다.

다음과 같은 방법으로 New Field 다이얼로그 박스에서 조회 필드를 만듭니다.

1 Name 에디트 박스에 조회 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.

2 Type 콤보 박스에서 필드의 데이터 타입을 선택합니다.

3 필요한 경우, Size 에디트 박스에 필드 크기를 입력합니다. 크기는 *TStringField*, *TBytesField* 및 *TVarBytesField*의 필드 타입에만 해당됩니다.

4 Field type 라디오 그룹에서 Lookup을 선택합니다. Lookup을 선택하면 Dataset 및 Key Fields 콤보 박스가 활성화됩니다.

5 Dataset 콤보 박스 드롭다운 리스트에서 필드 값을 알아낼 데이터셋을 선택합니다. 조회 데이터셋은 필드 컴포넌트 자체에 대한 데이터셋과는 달라야 합니다. 그렇지 않으면 런타임 시 순환 참조 예외가 발생합니다. 조회 데이터셋을 지정하면 Lookup Keys 및 Result Field 콤보 박스를 사용할 수 있습니다.

6 Key Fields 드롭다운 리스트에서 값을 비교할 현재 데이터셋의 필드를 선택합니다. 둘 이상의 필드를 비교하려면 드롭다운 리스트에서 선택하지 말고 필드 이름을 직접 입력합니다. 각 필드 이름은 세미콜론으로 구분하십시오. 둘 이상의 필드를 사용하는 경우에는 영구적 필드 컴포넌트를 사용해야 합니다.

7 단계 6 에서 지정한 Source Fields 필드와 비교할 조회 데이터셋의 필드를 Lookup Keys 드롭다운 리스트에서 선택합니다. 하나 이상의 키 필드를 지정한 경우에는 동일한 수의 조회 키를 지정해야 합니다. 하나 이상의 필드를 지정하려면, 필드 이름을 직접 입력하며 세미콜론으로 여러 필드 이름을 구분합니다.

8 Result Field 드롭다운 리스트에서 개발자가 만들고 있는 조회 필드의 값으로 반환할 조회 데이터셋의 필드를 선택합니다.

애플리케이션을 디자인하고 실행할 때, 조회 필드 값은 계산된 필드 값이 계산되기 전에 결정됩니다. 계산된 필드는 조회 필드를 기준으로 할 수 있지만 조회 필드는 계산된 필드를 기준으로 할 수 없습니다.

LookupCache 속성을 사용하여 조회 필드가 결정되는 방식을 조정할 수 있습니다. *LookupCache*는 데이터셋이 처음으로 열릴 때 조회 필드의 값이 메모리에 캐시로 저장되는지 또는 데이터셋의 현재 레코드가 변경될 때마다 조회 필드의 값을 동적으로 알아내는지의 여부를 결정합니다. *LookupCache*를 **true**로 설정하면 *LookupDataSet*이 변경되지 않으며 다른 조회 값의 수가 작은 경우 조회 필드 값을 캐시로 저장합니다. 조회 값을 캐시로 저장하면, *DataSet* 이 열릴 때 *LookupKeyFields* 값의 모든 집합에 대한 조회 값이 미리 로드되기 때문에 실행 속도가 빨라집니다. *DataSet*의 현재 레코드가 변경되면, 필드 객체는 *LookupDataSet*에 액세스하지 않고 캐시에서 *Value*를 찾을 수 있습니다. 이러한 성능 향상은 액세스가 느린 네트워크 상에 *LookupDataSet*이 있는 경우 특히 두드러집니다.

팁 조회 캐시를 사용하면 보조 데이터셋에서가 아니라 프로그램에서 조회 값을 제공할 수 있습니다. *LookupDataSet* 속성이 NULL인지 확인합니다. 그런 다음, *LookupList* 속성의 *Add* 메소드를 사용하여 조회 값으로 채웁니다. *LookupCache* 속성을 **true**로 설정합니다. 필드는 조회 데이터셋의 값으로 겹쳐쓰지 않고 제공된 조회 리스트를 사용합니다.

*DataSet*의 모든 레코드가 *KeyFields*에 대해 각기 다른 값을 가지면, 캐시의 값을 찾을 때의 오버헤드가 캐시에서 제공하는 성능상의 장점보다 더 클 수 있습니다. 캐시의 값을 찾는 오버헤드는 *KeyFields*에서 취할 수 있는 다른 값의 수만큼 증가합니다.

*LookupDataSet*이 일시적인 경우, 조회 값을 캐시로 저장하면 부정확한 결과를 초래할 수 있습니다. *RefreshLookupList*를 호출하여 조회 캐시의 값을 업데이트합니다. *RefreshLookupList*는 모든 *LookupKeyFields* 값의 집합에 대한 *LookupResultField*의 값을 포함하는 *LookupList* 속성을 다시 생성합니다.

런타임 시 *LookupCache*를 설정할 때, *RefreshLookupList*를 호출하여 캐시를 초기화합니다.

집계 필드 정의

집계 필드는 클라이언트 데이터셋의 유지 보수된 집계 값을 표시합니다. 집계는 레코드 집합의 데이터를 요약하는 계산입니다. 유지 보수된 집계에 대한 자세한 사항은 27-11페이지의 "유지 보수된 집계 사용"을 참조하십시오.

다음과 같은 방법으로 New Field 다이얼로그 박스에서 집계 필드를 만듭니다.

1. Name 에디트 박스에 집계 필드의 이름을 입력합니다. 기존 필드의 이름은 사용하지 마십시오.
2. Type 콤보 박스에서 필드의 집계 데이터 타입을 선택합니다.
3. Field 타입 라디오 그룹에서 Aggregate를 선택합니다.
4. OK를 선택합니다. 새로 정의된 집계 필드가 클라이언트 데이터셋에 자동으로 추가되고 *Aggregates* 속성이 적절한 집계 스펙을 포함하도록 자동으로 업데이트됩니다.
5. 새로 만들어진 집계 필드의 *ExprText* 속성에 집계에 대한 계산을 둡니다. 집계 정의에 대한 자세한 내용은 27-11페이지의 "집계 지정"을 참조하십시오.

일단 영구적 *TAggregateField*를 만들면 *TDBText* 컨트롤을 집계 필드에 연결할 수 있습니다. 그러면 *TDBText* 컨트롤이 원본으로 사용하는 클라이언트 데이터셋의 현재 레코드와 관련된 집계 필드의 값을 표시합니다.

영구적 필드(persistent field) 컴포넌트 삭제

영구적 필드 컴포넌트의 삭제는 테이블에서 사용 가능한 열의 부분 집합을 액세스하고, 테이블의 열을 바꿀 새로운 영구적 필드를 정의하는 경우에 유용합니다. 다음과 같은 방법으로 데이터셋에 대한 하나 이상의 영구적 필드 컴포넌트를 제거합니다.

1. Fields Editor 리스트 박스에서 제거할 필드를 선택합니다.
2. *Del*을 누릅니다.

참고 컨텍스트 메뉴를 호출하고 Delete를 선택하여 선택된 필드를 삭제할 수도 있습니다.

제거하는 필드는 더 이상 데이터셋에서 사용할 수 없으며 데이터 인식 컨트롤로 표시할 수도 없습니다. 실수로 삭제한 영구적 필드 컴포넌트는 항상 다시 만들 수 있지만 속성이나 이벤트에 대한 이전의 변경 내용은 없어집니다. 자세한 내용은 23-4페이지의 "영구적 필드(persistent field) 생성"을 참조하십시오.

참고 데이터셋의 모든 영구적 필드 컴포넌트를 제거하면 데이터셋은 원본으로 사용하는 데이터베이스 테이블에 있는 모든 열에 대해 동적 필드 컴포넌트를 사용하는 상태로 변환됩니다.

영구적 필드 속성 및 이벤트 설정

디자인 타임에 영구적 필드 컴포넌트에 대한 속성을 설정하고 이벤트를 사용자 정의할 수 있습니다. 속성은 데이터 인식 컴포넌트에서 예를 들어, 필드가 *TDBGrid*에 나타날 수 있는지의 여부나 값을 수정할 수 있는지의 여부와 같은, 필드 표시 방식을 제어합니다. 이벤트는 필드의 데이터를 가져오고, 변경, 설정 또는 검증할 때 발생하는 내용을 제어합니다.

필드 컴포넌트의 속성을 설정하거나 속성에 대한 사용자 정의 이벤트 핸들러를 작성하려면, Fields Editor 에서 컴포넌트를 선택하거나 Object Inspector 의 컴포넌트 리스트에서 컴포넌트를 선택합니다.

디자인 타임 시 표시 및 편집 속성 설정

선택된 필드 컴포넌트의 표시 속성을 편집하려면, Object Inspector 윈도우의 Properties 페이지로 변환합니다. 다음 표는 편집될 수 있는 표시 속성을 요약한 것입니다.

표 23.3 필드 컴포넌트 속성

속성	용도
Alignment	데이터 인식 컴포넌트 내에서 필드 내용을 왼쪽, 오른쪽 또는 가운데에 정렬합니다.
ConstraintErrorMessage	제약 조건으로 충돌을 편집할 때 표시할 텍스트를 지정합니다.
CustomConstraint	편집 중에 데이터에 적용할 로컬 제약 조건을 지정합니다.
Currency	Numeric 필드에만 해당됩니다. **true**: 통화 값을 표시합니다. **false**(기본값): 통화 값을 표시하지 않습니다.
DisplayFormat	데이터 인식 컴포넌트에 표시되는 데이터 형식을 지정합니다.
DisplayLabel	데이터 인식 그리드 컴포넌트의 필드에 대한 열 이름을 지정합니다.
DisplayWidth	해당 필드를 표시하는 그리드 열을 문자로 지정합니다.
EditFormat	데이터 인식 컴포넌트의 데이터의 편집 형식을 지정합니다.
EditMask	편집 가능한 필드의 데이터 입력을 문자의 지정된 타입 및 범위로 제한하고, 필드 내에 나타나는 편집 불가능한 모든 특수 문자(하이픈, 괄호 등)를 지정합니다.
FieldKind	만들 필드의 타입을 지정합니다.
FieldName	필드가 값과 데이터 타입을 파생한, 테이블의 실제 열 이름을 지정합니다.
HasConstraints	필드에 주어진 제약 조건이 있는지 나타냅니다.
ImportedConstraint	Data Dictionary 또는 SQL 서버에서 임포트된 SQL 제약 조건을 지정합니다.
Index	데이터셋의 필드 순서를 지정합니다.
LookupDataSet	*Lookup*이 **true**일 때 필드 값을 알아내는 데 사용되는 데이터셋을 지정합니다.
LookupKeyFields	조회할 때 비교할 조회 데이터셋의 필드를 지정합니다.
LookupResultField	해당 필드로 값을 복사할 조회 데이터셋의 필드를 지정합니다.
MaxValue	Numeric 필드에만 해당됩니다. 사용자가 필드에 입력할 수 있는 최대 값을 지정합니다.
MinValue	Numeric 필드에만 해당됩니다. 사용자가 필드에 입력할 수 있는 최소 값을 지정합니다.
Name	C++Builder 내의 필드 컴포넌트의 컴포넌트 이름을 지정합니다.
Origin	원본으로 사용하는 데이터베이스에 나타나는 것과 똑같이 필드의 이름을 지정합니다.
Precision	Numeric 필드에만 해당됩니다. 유효 자릿수를 지정합니다.

표 23.3 필드 컴포넌트 속성(계속)

속성	용도
ReadOnly	**true**: 데이터 인식 컨트롤에 필드 값을 표시하지만 편집할 수는 없습니다. **false**(기본값): 필드 값을 표시하고 편집할 수 있습니다.
Size	문자열 기반 필드에 표시하거나 입력할 수 있는 문자의 최대 수 또는 *TBytesField* 및 *TVarBytesField* 필드의 크기(바이트)를 지정합니다.
Tag	필요에 따라 모든 컴포넌트에서 프로그래머가 사용할 수 있는 정수(Long) 버킷
Transliterate	**true**(기본값): 데이터셋과 데이터베이스 간에 데이터가 전송될 때 각 로케일에 대한 변환이 일어나도록 지정합니다. **false**: 로케일 변환이 일어나지 않습니다.
Visible	**true**(기본값): 데이터 인식 그리드에 필드를 표시할 수 있습니다. **false**: 데이터 인식 그리드 컴포넌트의 필드를 표시할 수 없습니다. 사용자 정의 컴포넌트는 이 속성에 따라 표시 여부를 결정할 수 있습니다.

모든 필드 컴포넌트가 모든 속성을 다 사용할 수 있는 것은 아닙니다. 예를 들어, *TString Field* 타입의 필드 컴포넌트는 *Currency*, *MaxValue* 또는 *DisplayFormat* 속성이 없으며, *TFloatField* 타입의 컴포넌트는 *Size* 속성이 없습니다.

대부분의 속성의 용도는 간단하지만, *Calculated*와 같은 일부 속성은 추가적인 프로그래밍 단계를 거쳐야 사용할 수 있습니다. *DisplayFormat*, *EditFormat* 및 *Edit Mask*와 같은 기타 속성은 상호 관련되어 있으므로 서로의 설정이 통합되어 조정되어야 합니다. *Display Format*, *EditFormat* 및 *EditMask* 사용에 대한 자세한 내용은 23-14페이지의 "사용자 입력 제어 및 마스킹"을 참조하십시오.

런타임 시 필드 컴포넌트 속성 설정

런타임 시 필드 컴포넌트의 속성을 사용하고 처리할 수 있습니다. 필드 이름에 데이터셋 이름을 연결하여 얻을 수 있는 이름으로 영구적 필드 컴포넌트에 액세스합니다.

예를 들어, 다음 코드는 *Customers* 테이블의 *CityStateZip* 필드에 대한 *ReadOnly* 속성을 **true**로 설정합니다.

```
CustomersCityStateZip->ReadOnly = true;
```

그리고 다음 명령문은 *Customers* 테이블에 있는 *CityStateZip* 필드의 *Index* 속성을 3으로 설정하여 필드 순서를 변경합니다.

```
CustomersCityStateZip->Index = 3;
```

필드 컴포넌트에 대한 어트리뷰트(attribute) 집합 생성

애플리케이션에서 사용하는 데이터셋의 일부 필드가 일반적인 서식 속성(*Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue* 등)을 공유하는 경우 단일 필드에 대한 속성을 설정하고 이러한 속성을 Data Dictionary에 어트리뷰트 집합으로 저장하면 매우 편리합니다. Data Dictionary에 저장한 어트리뷰트 집합은 다른 필드에 쉽게 적용할 수 있습니다.

참고 어트리뷰트 집합과 Data Dictionary는 BDE 호환 데이터셋에서만 사용 가능합니다.

다음과 같은 방법으로 데이터셋의 필드 컴포넌트에 기반한 어트리뷰트(attribute) 집합을 만듭니다.

1. 데이터셋을 더블 클릭하여 Fields Editor를 호출합니다.
2. 속성을 설정할 필드를 선택합니다.
3. Object Inspector에 있는 필드에 대해 원하는 속성을 설정합니다.
4. 마우스 오른쪽 버튼으로 Fields Editor 리스트 박스를 클릭하여 컨텍스트 메뉴를 호출합니다.
5. Save Attributes를 선택하여 현재 필드의 속성 설정을 Data Dictionary의 어트리뷰트 집합으로 저장합니다.

어트리뷰트 집합에 대한 이름의 기본값으로 현재 필드의 이름을 사용합니다. 컨텍스트 메뉴에서 Save Attributes를 선택하는 대신 Save Attributes As를 선택하여 어트리뷰트 집합에 다른 이름을 지정할 수 있습니다.

일단 새로운 어트리뷰트 집합을 만들어 Data Dictionary 에 추가하면 이 어트리뷰트 집합을 다른 영구적 필드 컴포넌트에 연결할 수 있습니다. 나중에 이 연결을 끊더라도 어트리뷰트 집합은 Data Dictionary에 계속 정의되어 있습니다.

참고 SQL 탐색기에서 직접 어트리뷰트 집합을 만들 수도 있습니다. SQL 탐색기를 사용하여 어트리뷰트 집합을 만들 경우 어트리뷰트 집합은 Data Dictionary에 추가되지만 다른 필드에 적용되지는 않습니다. SQL 탐색기를 사용하여 다음의 두 가지의 추가 어트리뷰트를 지정할 수 있습니다. *TFloatField*, *TStringField* 등과 같은 필드 타입과 *TDBEdit*, *TDBCheckBox* 등과 같은 데이터 인식 컨트롤은 어트리뷰트 집합에 기반한 필드를 폼에 끌어 놓을 때 자동으로 폼에 들어갑니다. 자세한 내용은 SQL 탐색기에 대한 온라인 도움말을 참조하십시오.

어트리뷰트(attribute) 집합과 필드 컴포넌트 연결

애플리케이션에서 사용하는 데이터셋의 일부 필드가 같은 서식 속성을 사용하고 (*Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue* 등) 해당 속성을 Data Dictionary에 어트리뷰트 집합으로 저장한 경우, 각 필드에 대해 수동적으로 설정을 다시 만들 필요 없이 어트리뷰트 집합을 필드에 쉽게 적용할 수 있습니다. 또한 나중에 Data Dictionary에 있는 어트리뷰트 설정을 변경한 경우, 다음에 필드 컴포넌트가 데이터셋에 추가될 때, 어트리뷰트 집합과 연관된 모든 필드에 이러한 변경 내용이 자동으로 적용됩니다.

다음과 같은 방법으로 어트리뷰트 집합을 필드 컴포넌트에 추가합니다.

1. 데이터셋을 더블 클릭하여 Fields Editor를 호출합니다.
2. 어트리뷰트 집합을 적용할 필드를 선택합니다.
3. 컨텍스트 메뉴를 호출하고 Associate Attributes를 선택합니다.
4. Associate Attributes 다이얼로그 박스에서 적용할 어트리뷰트 집합을 선택하거나 입력합니다. 현재 필드와 같은 이름을 가진 Data Dictionary에 어트리뷰트 집합이 있으면 집합 이름이 에디트 박스에 나타납니다.

중요 Data Dictionary 에 있는 어트리뷰트 집합이 나중에 변경된 경우 어트리뷰트 집합을 사용하는 모든 필드 컴포넌트에 어트리뷰트 집합을 다시 적용해야 합니다. 어트리뷰트를 다시 적용할 때 Fields Editor와 데이터셋 안에 있는 다중 선택 필드 컴포넌트를 호출할 수 있습니다.

어트리뷰트(attribute) 연결 제거

어트리뷰트 집합과 필드 간의 연결이 더 이상 필요하지 않은 경우 다음 단계에 따라 어트리뷰트 연결을 끊을 수 있습니다.

1. 필드를 포함하는 데이터셋에 대한 Fields Editor를 호출합니다.
2. 어트리뷰트 연결을 끊을 하나 이상의 필드를 선택합니다.
3. Fields Editor에 대한 컨텍스트 메뉴를 호출하고 Unassociate Attributes를 선택합니다.

중요 어트리뷰트 집합의 연결을 끊어도 필드 속성은 변하지 않습니다. 필드는 어트리뷰트 집합이 필드에 적용되었을 때 가지게 된 설정을 유지합니다. 이러한 속성을 변경하려면 Fields Editor를 선택하고 Object Inspector에 있는 속성을 설정합니다.

사용자 입력 제어 및 마스킹

EditMask 속성은 사용자가 *TStringField*, *TDateField*, *TTimeField*, *TDateTime Field* 및 *TSQLTimeStampField* 컴포넌트와 연결된 데이터 인식 컴포넌트에 입력할 수 있는 값의 타입과 범위를 제어하는 방법을 제공합니다. 기존의 마스크를 사용하거나 새로운 마스크를 만들 수 있습니다. 가장 쉽게 편집 마스크를 사용하고 만드는 방법은 Input Mask Editor를 사용하는 것입니다. 그러나 Object Inspector의 *EditMask* 필드에 직접 마스크를 입력할 수도 있습니다.

참고 *TStringField* 컴포넌트에서, *EditMask* 속성 또한 표시 형식입니다.

다음과 같은 방법으로 필드 컴포넌트에 대한 Input Mask Editor를 호출합니다.

1. Fields Editor나 Object Inspector에서 컴포넌트를 선택합니다.
2. Object Inspector의 Properties 페이지를 클릭합니다.
3. Object Inspector의 EditMask 필드에 대한 값 열을 더블 클릭하거나 생략 부호 버튼을 클릭합니다. Input Mask Editor가 열립니다.

Input Mask 에디트 박스를 사용하면 마스크 형식을 만들고 편집할 수 있습니다. Sample Masks 그리드를 사용하면 미리 정의된 마스크에서 선택할 수 있습니다. 예제 마스크를 선택하면, 개발자가 수정하거나 있는 그대로 사용할 수 있는 Input Mask 에디트 박스에 마스크 형식이 나타납니다. Test Input 에디트 박스에서 마스크에 대해 허용할 수 있는 사용자 입력을 테스트할 수 있습니다.

개발자가 사용자 정의 마스크 집합을 만든 경우에는 Masks 버튼을 사용하면 선택을 쉽게 하기 위해 마스크 집합을 Sample Masks 그리드로 로드할 수 있습니다.

숫자, 날짜 및 시간 필드에 대한 디폴트 서식 사용

C++Builder는 *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, *TTimeField*, *TSQLTimeStampField* 컴포넌트에 대한 기본 표시, 편집 서식 루틴, 지능적 디폴트 서식을 제공합니다. 이 루틴들을 사용하기 위해 해야 할 작업은 없습니다.

디폴트 서식은 다음과 같은 루틴에서 수행됩니다.

표 23.4 필드 컴포넌트 서식 루틴

루틴	사용 위치
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*,
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField*, *TBCDField*
BcdToStrF	*TFMTBcdField*

필드 컴포넌트의 데이터 타입에 적합한 형식 속성만 지정된 컴포넌트에 대해 사용할 수 있습니다.

날짜, 시간, 통화, 숫자 값에 대한 디폴트 서식 규칙은 제어판의 국가별 설정 속성을 기반으로 합니다. 예를 들어, 미국의 디폴트 설정을 사용하면 *Currency* 속성이 **true**로 설정된 *TFloat Field* 열은 값 1234.56에 대한 *DisplayFormat* 속성을 $1234.56로 설정하는 반면에 *EditFormat*은 1234.56입니다.

디자인 타임이나 런타임 시, 필드 컴포넌트의 *DisplayFormat* 및 *EditFormat* 속성을 편집하면 필드에 대한 디폴트 표시 설정을 오버라이드할 수 있습니다. 또한 *OnGetText* 및 *OnSetText* 이벤트 핸들러를 작성하여 런타임 시 필드 컴포넌트에 대한 서식을 사용자 정의할 수 있습니다.

이벤트 처리

대부분의 컴포넌트처럼 필드 컴포넌트도 연결된 이벤트가 있습니다. 메소드는 이벤트에 대한 핸들러로 할당될 수 있습니다. 이들 핸들러를 작성하면, 데이터 인식 컨트롤을 통해 필드에 입력된 데이터에 영향을 주는 이벤트의 발생에 응답하고 개발자가 의도한대로 디자인 동작을 수행할 수 있습니다. 다음 표는 필드 컴포넌트에 연결된 이벤트를 나열한 것입니다.

표 23.5 필드 컴포넌트 이벤트

이벤트	용도
OnChange	필드 값이 변경될 때 호출됩니다.
OnGetText	표시 또는 검색을 위해 필드 컴포넌트 값을 검색할 때 호출됩니다.
OnSetText	필드 컴포넌트 값이 설정될 때 호출됩니다.
OnValidate	편집이나 삽입 작업으로 인해 값이 변경될 때마다 필드 컴포넌트 값을 확인하기 위해 호출됩니다.

OnGetText 및 *OnSetText* 이벤트는 주로 기본 서식 함수보다 정교한 사용자 정의 서식을 원하는 프로그래머에게 유용합니다. *OnChange*는 메뉴나 비주얼 컨트롤을 활성화하거나 비활성화하는 것과 같은 데이터 변경과 관련된 애플리케이션 특정 작업을 수행하는 경우에 유용합니다. *OnValidate*는 데이터베이스 서버에 값을 반환하기 전에 애플리케이션에서 데이터 입력 검증을 제어하고자 하는 경우에 유용합니다.

다음과 같은 방법으로 필드 컴포넌트에 대한 이벤트 핸들러를 작성합니다.

1 컴포넌트를 선택합니다.

2 Object Inspector에서 Events 페이지를 선택합니다.

3 이벤트 핸들러에 대한 Value 필드를 더블 클릭하여 소스 코드 윈도우를 표시합니다.

4 핸들러 코드를 작성하거나 편집합니다.

런타임 시 필드 컴포넌트 메소드 작업

런타임 시 사용 가능한 필드 컴포넌트 메소드를 사용하면 한 데이터 타입에서 다른 데이터 타입으로 필드 값을 변환할 수 있고, 필드 컴포넌트에 연결된 폼의 첫 번째 데이터 인식 컨트롤에 포커스를 설정할 수 있습니다.

필드에 연결된 데이터 인식 컴포넌트의 포커스를 제어하는 것은 애플리케이션이 *BeforePost*와 같은 데이터셋 이벤트 핸들러에서 레코드 지향 데이터 검증을 수행할 때 중요합니다. 검증은 연결된 데이터 인식 컨트롤이 포커스를 갖고 있는지 여부와 상관없이 레코드의 필드에서 수행될 수 있습니다. 레코드의 특정 필드에서 검증이 실패하면 잘못된 데이터를 포함하는 데이터 인식 컨트롤에 포커스를 두어 수정 사항을 입력할 수 있습니다.

필드의 *FocusControl* 메소드를 사용하여 필드의 데이터 인식 컴포넌트에 대한 포커스를 제어합니다. *FocusControl*은 포커스를 필드에 연결된 폼의 첫 번째 데이터 인식 컨트롤로 설정합니다. 이벤트 핸들러는 필드를 검증하기 전에 필드의 *FocusControl* 메소드를 호출해야 합니다. 다음 코드는 *Customers* 테이블의 *Company* 필드에 대한 *FocusControl* 메소드를 호출하는 방법을 설명합니다.

```
CustomersCompany->FocusControl();
```

다음 표는 다른 필드 컴포넌트 메소드와 그 용도를 나열한 것입니다. 각 메소드에 대한 전체 리스트와 자세한 내용은 온라인 *VCL Reference*에서 *TField*와 자손에 대한 항목을 참조하십시오.

표 23.6 선택된 필드 컴포넌트 메소드

메소드	용도
AssignValue	자동 변환 기능을 사용하여 필드의 타입에 따라 지정된 값으로 필드 값을 설정합니다.
Clear	필드를 해제하고 필드 값을 Null로 설정합니다.
GetData	필드에서 서식이 없는 데이터를 검색합니다.
IsValidChar	값을 설정하기 위해 데이터 인식 컨트롤에서 사용자가 입력한 문자가 이 필드에 허용되는지 결정합니다.
SetData	이 필드에 서식이 없는 데이터를 할당합니다.

필드 값 표시, 변환 및 액세스

TDBEdit 및 *TDBGrid*와 같은 데이터 인식 컨트롤은 필드 컴포넌트에 연결된 값을 자동으로 표시합니다. 데이터셋과 컨트롤에서 편집이 가능하면, 데이터 인식 컨트롤 또한 데이터베이스에 새로운 값과 변경된 값을 전송할 수 있습니다. 일반적으로 데이터 인식 컨트롤의 기본 속성과 메소드를 사용하면 개발자가 추가 프로그래밍 없이도 데이터셋에 연결하고 값을 표시하며 업데이트를 할 수 있습니다. 데이터베이스 애플리케이션에서 가능할 때마다 사용하십시오. 데이터 인식 컨트롤에 대한 자세한 내용은 19장의 "데이터 컨트롤 사용"을 참조하십시오.

표준 컨트롤 또한 필드 컴포넌트에 연결된 데이터베이스 값을 표시하고 편집할 수 있습니다. 그러나 표준 컨트롤을 사용하려면 개발자가 추가로 프로그래밍해야 합니다. 예를 들어, 애플리케이션에서는 표준 컨트롤을 사용할 때 필드 값이 변경되기 때문에 컨트롤을 업데이트할 시기를 추적해야 합니다. 데이터셋에 데이터 소스 컴포넌트가 있을 경우, 이벤트를 사용하여 이러한 작업을 할 수 있습니다. 특히, *OnDataChange* 이벤트를 사용하면 컨트롤의 값을 업데이트해야 할 시기를 알 수 있고, *OnStateChange* 이벤트를 사용하면 컨트롤을 활성화하거나 및 비활성화할 시기를 결정할 수 있습니다. 이들 이벤트에 대한 자세한 내용은 19-4페이지의 "데이터 소스에 따른 변경 내용에 대한 응답"을 참조하십시오.

다음 항목에서는 표준 컨트롤에 표시하기 위해 필드 값을 사용하는 방법을 설명합니다.

표준 컨트롤에 필드 컴포넌트 값 표시

애플리케이션은 필드 컴포넌트의 *Value* 속성을 통해 데이터셋 열의 값에 액세스할 수 있습니다. 예를 들어, *CustomersCompany* 필드의 값이 변경될 수도 있기 때문에 다음과 같은 *OnDataChange* 이벤트 핸들러는 *TEdit* 컨트롤의 텍스트를 업데이트합니다.

```
void __fastcall TForm1::Table1DataChange(TObject *Sender, TField *Field)
{
  Edit3->Text = CustomersCompany->Value;
}
```

이 메소드는 문자열 값에는 잘 수행되지만, 다른 데이터 타입에 대해서는 변환을 처리할 추가 프로그램이 필요할 수도 있습니다. 다행히도, 필드 컴포넌트에는 기본 변환 처리 속성이 있습니다.

참고 또한 가변 타입을 사용하여 필드 값을 액세스하고 설정할 수 있습니다. 필드 값 액세스 및 설정을 위한 가변 타입 사용에 대한 자세한 내용은 23-19페이지의 "디폴트 데이터셋 속성으로 필드 값 액세스"를 참조하십시오.

필드 값 변환

변환 속성은 한 데이터 타입에서 다른 데이터 타입으로 변환합니다. 예를 들어, *AsString* 속성은 숫자 및 부울 값을 문자열 표시로 변환합니다. 다음 표는 필드 컴포넌트 변환 속성을 나열하고, 필드 컴포넌트 클래스에 따라 필드 컴포넌트에 권장되는 속성을 설명한 것입니다.

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	예	해당 없음	예	예	예	예
TWideStringField	예	예	예	예	예	예
TIntegerField	예	예	해당 없음	예		
TSmallIntField	예	예	예	예		
TWordField	예	예	예	예		
TLargeintField	예	예	예	예		
TFloatField	예	예	예	예		
TCurrencyField	예	예	예	예		
TBCDField	예	예	예	예		
TFMTBCDField	예	예	예	예		
TDateTimeField	예	예		예	예	
TDateField	예	예		예	예	
TTimeField	예	예		예	예	
TSQLTimeStampField	예	예		예	예	
TBooleanField	예	예				
TBytesField	예	예				
TVarBytesField	예	예				
TBlobField	예	예				
TMemoField	예	예				
TGraphicField	예	예				
TVariantField	해당 없음	예	예	예	예	예
TAggregateField	예	예				

테이블의 일부 열들은 하나 이상의 변환 속성(예를 들어, *AsFloat*, *AsCurrency* 및 *AsBCD*)을 참조할 수도 있습니다. 왜냐하면 해당 속성 중 하나를 지원하는 모든 필드 데이터 타입은 항상 다른 속성도 함께 지원하기 때문입니다.

또한 *AsVariant* 속성은 모든 데이터 타입을 변환할 수 있습니다. 위에 나열되지 않은 다른 데이터 타입 중에서 실제로는 유일한 옵션이라고 할 수 있는 *AsVariant*도 사용할 수 있습니다. 확실하지 않을 경우에는 *AsVariant*를 사용합니다.

어떤 경우에는 변환이 항상 가능하지는 않습니다. 예를 들어, 문자열 값이 인식할 수 있는 datetime 형식으로 되어 있는 경우에만 *AsDateTime*이 문자열을 날짜, 시간 또는 datetime 형식으로 변환하는 데 사용될 수 있습니다. 변환 시도가 실패하면 예외가 발생합니다.

일부 경우에는, 변환은 가능하지만 변환의 결과가 항상 명확하지는 않습니다. 예를 들어, *TDateTimeField* 값을 부동 소수점 형식으로 변환하는 것은 무엇을 의미합니까? *AsFloat*는 필드의 날짜 부분을 12/31/1899 이후의 날짜 수로 변환하고 필드의 시간 부분은 24시간을 분모로 하는 분수로 변환합니다. 표 23.7은 특수한 결과를 생성하는 변환을 나열한 것입니다.

표 23.7 특수한 변환 결과

변환	결과
문자열에서 부울로 변환	"True", "False", "Yes" 및 "No"를 부울로 변환합니다. 다른 값은 예외를 발생시킵니다.
부동 소수점에서 정수로 변환	부동 소수점 값을 가장 가까운 정수 값으로 반올림하거나 우수리를 버립니다
DateTime 또는 SQLTimeStamp에서 부동 소수점으로 변환	날짜는 12/31/1899 이후의 일 수로 변환하고 시간은 24시간을 분모로 하는 분수로 변환합니다.
부울에서 문자열로 변환	모든 부울 값을 "True"나 "False"로 변환합니다.

그 외의 경우에는 변환을 할 수 없습니다. 이 경우 변환을 시도하면 예외가 발생됩니다.

변환은 항상 할당되기 전에 발생합니다. 예를 들어, 다음 명령문은 *Customers CustNo* 값을 문자열로 변환하고 문자열을 편집 컨트롤의 텍스트에 할당합니다.

```
Edit1->Text = CustomersCustNo->AsString;
```

이와 반대로 다음 명령문은 편집 컨트롤의 텍스트를 정수로 *CustomersCustNo* 필드에 할당합니다.

```
MyTableMyField->AsInteger = StrToInt(Edit1->Text);
```

디폴트 데이터셋 속성으로 필드 값 액세스

필드 값 액세스를 위한 가장 일반적인 메소드는 *FieldValues* 속성에 가변 타입을 사용하는 것입니다. 예를 들어, 다음 명령문은 에디트 박스의 값을 *Customers* 테이블의 *CustNo* 필드에 둡니다.

```
Customers->FieldValues["CustNo"] = Edit2->Text;
```

FieldValues 속성은 Variant 타입이므로, 다른 데이터 타입을 Variant 값으로 자동 변환합니다.

가변 타입에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

데이터셋의 Fields 속성으로 필드 값 액세스

필드가 속한 데이터셋 컴포넌트의 *Fields* 속성으로 필드 값을 액세스할 수 있습니다. *Fields*는 데이터셋의 모든 필드의 인덱스된 리스트를 유지합니다. 필드 값을 *Fields* 속성으로 액세스하면, 많은 열을 반복해야 할 경우나 애플리케이션에서 디자인 타임에는 사용할 수 없는 테이블 작업을 할 경우에 유용합니다.

Fields 속성을 사용하려면, 데이터셋에 있는 필드의 순서와 필드의 데이터 타입을 알아야 합니다. 순서 번호를 사용하여 액세스할 필드를 지정합니다. 데이터셋의 첫 번째 필드는 번호가 0입니다. 필드 값은 각 필드 컴포넌트의 변환 속성을 사용하여 적절하게 변환되어야 합니다. 필드 컴포넌트 변환 속성에 대한 자세한 내용은 23-18페이지의 "필드 값 변환"을 참조하십시오.

예를 들어, 다음 명령문은 *Customers* 테이블의 일곱 번째 열(Country)의 현재 값을 편집 컨트롤에 할당합니다.

```
Edit1->Text = CustTable->Fields->Fields[6]->AsString;
```

이와 반대로, 데이터셋의 *Fields* 속성을 원하는 필드로 설정하여 값을 필드에 할당할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Customers->Edit();
Customers->Insert();
Customers->Fields->Fields[6]->AsString = Edit1->Text;
Customers->Post();
```

데이터셋의 FieldByName 메소드로 필드 값 액세스

또한 데이터셋의 *FieldByName* 메소드로 필드 값을 액세스할 수 있습니다. 이 메소드는 액세스하려는 필드 이름은 알지만 디자인 타임 시 원본으로 사용하는 테이블에 액세스할 수 없을 경우에 유용합니다.

*FieldByName*을 사용하려면, 액세스하려는 데이터셋과 필드 이름을 알아야 합니다. 필드의 이름을 인수로서 메소드에 전달합니다. 필드 값을 액세스하거나 변경하려면, *AsString*이나 *AsInteger*와 같은 적절한 필드 컴포넌트 변환 속성을 사용하여 결과를 변환합니다. 예를 들어, 다음 명령문은 *Customers* 데이터셋의 *CustNo* 필드 값을 편집 컨트롤에 할당합니다.

```
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

이와 반대로, 필드에 값을 할당할 수 있습니다.

```
Customers->Edit();
Customers->FieldByName("CustNo")->AsString = Edit2->Text;
Customers->Post();
```

필드에 대한 기본값 설정

DefaultExpression 속성을 사용하여 런타임 시 클라이언트 데이터셋 또는 BDE 호환 데이터셋의 필드에 대한 기본값을 계산하는 방법을 지정할 수 있습니다. *DefaultExpression*은 필드 값을 참조하지 않는 유효한 SQL 값 표현식이 될 수 있습니다. 표현식에 숫자 값이 아닌 리터럴이 포함되는 경우에는 리터럴이 인용 부호로 나타납니다. 예를 들어, 시간 필드에 대한 정오의 기본값은 다음과 같이 표시됩니다.

```
'12:00:00'
```

리터럴 값 주위에 인용 부호가 포함됩니다.

참고 원본으로 사용하는 데이터베이스 테이블이 필드에 대한 기본값을 정의하는 경우, 개발자가 *DefaultExpression*에 지정하는 기본값이 우선권을 가집니다. 이는 데이터셋이 필드를 포함한 레코드를 포스트할 때 *DefaultExpression*이 적용되고 이것이 편집된 레코드가 데이터베이스 서버에 적용되기 전에 발생하기 때문입니다.

제약 조건 사용

클라이언트 데이터셋 또는 BDE 호환 데이터셋의 필드 컴포넌트는 SQL 서버 제약 조건을 사용할 수 있습니다. 또한 애플리케이션의 로컬 데이터셋에 대해 애플리케이션은 사용자 정의 제약 조건을 만들고 사용할 수 있습니다. 모든 제약 조건은 필드가 저장할 수 있는 값의 범위를 제한하는 규칙 또는 조건입니다.

사용자 정의 제약 조건 생성

사용자 정의 제약 조건은 다른 제약 조건과는 달리 서버로부터 임포트되지 않습니다. 사용자 정의 제약 조건은 개발자의 로컬 애플리케이션에서 선언, 구현, 적용됩니다. 그러므로 사용자 정의 제약 조건은 데이터 입력의 적용을 미리 확인할 때 유용하지만 서버 애플리케이션을 통해 보내고 받는 데이터에 대해서는 사용자 정의 제약 조건을 적용할 수 없습니다.

사용자 정의 제약 조건을 만들려면, *CustomConstraint* 속성을 설정하여 제약 조건을 지정하고, *ConstraintErrorMessage*를 사용자가 런타임 시 제약 조건을 위반할 때 표시하는 메시지로 설정합니다.

*CustomConstraint*는 필드 값에 부여된 애플리케이션 특정 제약 조건을 지정하는 SQL 문자열입니다. 사용자가 필드에 입력할 수 있는 값을 제한하도록 *CustomConstraint*를 설정합니다. *CustomConstraint*는 다음과 같은 유효한 SQL 검색 표현식이 될 수 있습니다.

```
x > 0and x < 100
```

필드 값을 참조하는 데 사용되는 이름은, 제약 조건 표현식에서 지속적으로 사용되는 한, 예약된 SQL 키워드가 아닌 모든 문자열이 될 수 있습니다.

참고 사용자 정의 제약 조건은 BDE 호환 데이터셋 또는 클라이언트 데이터셋에서만 사용할 수 있습니다.

사용자 정의 제약 조건은 서버로부터 받은 필드 값에 대한 제약 조건과 함께 부과됩니다. 서버에 의해 부과된 제약 조건을 보려면 *ImportedConstraint* 속성을 참조하십시오.

서버 제약 조건 사용

대부분의 SQL 데이터베이스는 제약 조건을 사용하여 필드에 대해 사용할 수 있는 값에 조건을 부여합니다. 예를 들어, 어떤 필드는 Null 값을 허용하지 않을 수도 있고, 값이 해당 열에 대해 고유해야 하거나 *0*보다는 크고 *150*보다 작아야 할 수도 있습니다. 이러한 조건을 클라이언트 애플리케이션에 복제할 수 있으므로 클라이언트 데이터셋과 BDE 호환 데이터셋은 *ImportedConstraint* 속성을 제공하여 서버의 제약 조건을 로컬로 전달합니다.

*ImportedConstraint*는 읽기 전용 속성으로서 일정 방법으로 필드 값을 제한하는 SQL 절을 지정합니다. 예를 들면, 다음과 같습니다.

```
Value > 0 and Value < 100
```

데이터베이스 엔진이 해석할 수 없어 주석으로 임포트된 비표준 또는 서버 관련 SQL을 편집하는 경우 외에는 *ImportedConstraint*의 값을 변경하지 마십시오.

필드 값에 제약 조건을 추가하려면 *CustomConstraint* 속성을 사용합니다. 사용자 정의 제약 조건이 임포트된 제약 조건에 추가로 부여됩니다. 서버 제약 조건이 변경되면 *ImportedConstraint* 값도 변경되지만 *CustomConstraint* 속성의 제약 조건은 계속 유지됩니다.

ImportedConstraint 속성으로부터 제약 조건을 제거해도 해당 제약 조건을 위반하는 필드 값의 유효성은 변경되지 않습니다. 제약 조건을 제거하면 로컬이 아니라 서버에서 제약 조건이 확인됩니다. 제약 조건이 로컬에서 확인되면 위반이 발견된 경우 서버로부터 오류 메시지를 표시하지 않고 *ConstraintErrorMessage* 속성으로 제공된 오류 메시지가 표시됩니다.

객체 필드 사용

객체 필드는 보다 간단한 데이터 타입과의 복합을 나타내는 필드입니다. 여기에는 ADT(Abstract Data Type) 필드, 배열 필드, 데이터셋 필드 및 참조 필드가 포함됩니다. 이러한 필드 타입은 모두 자식 필드 또는 다른 데이터셋을 포함하거나 참조합니다.

ADT 필드와 배열 필드는 자식 필드를 포함하는 필드입니다. ADT 필드의 자식 필드는 다른 필드 타입인 스칼라 또는 객체 타입이 될 수 있습니다. 이러한 자식 필드는 서로 타입이 다를 수 있습니다. 배열 필드에는 동일한 타입의 자식 필드 배열이 포함됩니다.

데이터셋 및 참조 필드는 다른 데이터셋을 액세스하는 필드입니다. 데이터셋 필드는 중첩된(디테일) 데이터셋에 액세스하고, 참조 필드는 포인터(참조)를 다른 영구적 객체(ADT)에 저장합니다.

표 23.8 객체 필드 컴포넌트의 타입

컴포넌트 이름	용도
TADTField	ADT(Abstract Data Type) 필드를 나타냅니다.
TArrayField	배열 필드를 나타냅니다.
TDataSetField	중첩된 데이터셋 참조를 포함하는 필드를 나타냅니다.
TReferenceField	ADT에 대한 포인터인 참조 필드를 나타냅니다.

Fields Editor로 객체 필드가 있는 데이터셋에 필드를 추가할 때, 올바른 타입의 영구적 객체 필드가 자동으로 만들어집니다. 데이터셋에 영구적 객체 필드를 추가하면 자동으로 데이터셋의 *ObjectView* 속성을 **true**로 설정합니다. 이렇게 하면 데이터셋이 구성 요소인 자식 필드를 각각의 독립적인 필드인 것처럼 단순화하지 않고 계층적으로 저장합니다.

다음 속성들은 모든 객체 필드에 공통적이며, 자식 필드와 데이터셋을 처리할 수 있는 기능을 제공합니다.

표 23.9 공통된 객체 필드 자손 속성

속성	용도
Fields	객체 필드에 속하는 자식 필드를 포함합니다.
ObjectType	객체 필드를 분류합니다.
FieldCount	객체 필드를 구성하는 자식 필드의 수를 포함합니다.
FieldValues	자식 필드 값에 대한 액세스를 제공합니다.

ADT 및 배열 필드 표시

ADT 및 배열 필드는 모두 데이터 인식 컨트롤을 통해 표시될 수 있는 자식 필드를 포함합니다. 하나의 필드 값을 나타내는 *TDBEdit*와 같은 데이터 인식 컨트롤은 쉼표로 구분된 편집할 수 없는 문자열에 자식 필드 값을 표시합니다. 또한 컨트롤의 *DataField* 속성을 객체 필드가 아니라 자식 필드로 설정하면, 자식 필드는 다른 일반적인 데이터 필드와 마찬가지로 편집된 상태로 표시될 수 있습니다.

TDBGrid 컨트롤은 데이터셋의 *ObjectView* 속성 값에 따라 ADT 필드와 배열 필드를 다르게 표시합니다. *ObjectView*가 **false**이면, 각 자식 필드가 하나의 열에 나타납니다. *ObjectView*가 **true**이면 , ADT 또는 배열 필드는 열의 제목 표시줄에 있는 화살표를 클릭하여 확장하거나 축소시킬 수 있습니다. 필드가 확장되면 각 자식 필드가 ADT 또는 배열 필드의 제목 표시줄 아래에 있는 각각의 열과 제목 표시줄에 나타납니다. ADT 또는 배열 필드를 축소하면, 단지 하나의 열만이 자식 필드가 포함된 쉼표로 구분된 편집할 수 없는 문자열로 나타납니다.

ADT 필드 작업

ADT는 서버에서 만들어진 사용자 정의 타입이며, 구조와 유사합니다. ADT는 대부분의 스칼라 필드 타입, 배열 필드, 참조 필드 및 중첩 ADT를 포함할 수 있습니다.

ADT 필드 타입의 데이터에 액세스하는 방법은 매우 다양합니다. 다음 예제에서 자식 필드 값을 *CityEdit*라는 에디트 박스에 할당하고 다음과 같은 ADT 구조를 사용하는 것에 대해 설명합니다.

```
Address
  Street
  City
  State
  Zip
```

영구적 필드(persistent field) 컴포넌트 사용

ADT 필드 값에 액세스하는 가장 쉬운 방법은 영구적 필드 컴포넌트를 사용하는 것입니다.위의 ADT 구조에서, 다음과 같은 영구적 필드는 Fields Editor를 사용하여 *Customer* 테이블에 추가될 수 있습니다.

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

이러한 영구적 필드의 경우에는 ADT 필드의 자식 필드에 이름으로 간단히 액세스할 수 있습니다.

```
CityEdit->Text = CustomerAddrCity->AsString;
```

영구적 필드는 ADT 자식 필드에 액세스하기 위한 가장 쉬운 방법이지만, 데이터셋 구조를 디자인 타임 시 알 수 없는 경우에는 영구적 필드를 사용할 수 없습니다. 영구적 필드를 사용하지 않고 ADT 자식 필드에 액세스할 때는 데이터셋의 *ObjectView* 속성을 **true**로 설정해야 합니다.

데이터셋의 FieldByName 메소드 사용

자식 필드의 이름을 ADT 필드의 이름으로 한정하면 데이터셋의 *FieldByName* 메소드를 사용하여 ADT 필드의 자식 필드를 액세스할 수 있습니다.

```
CityEdit->Text = Customer->FieldByName(“Address.City”)->AsString;
```

데이터셋의 FieldValues 속성 사용

또한 데이터셋의 *FieldValues* 속성으로 한정된 필드 이름을 사용할 수도 있습니다.

```
CityEdit->Text = Customer->FieldValues["Address.City"];
```

참고 ADT 자식 필드 값에 액세스하기 위한 다른 런타임 메소드와 달리, *FieldValues* 속성은 데이터셋의 *ObjectView* 속성이 **false**인 경우에도 작동합니다.

ADT 필드의 FieldValues 속성 사용

*TADTField*의 *FieldValues* 속성으로 자식 필드 값에 액세스할 수 있습니다. *Field Values*는 *Variant*를 받아들이고 반환하므로 모든 타입의 필드를 처리하고 변환할 수 있습니다. 인덱스 매개변수는 필드의 오프셋을 지정하는 정수 값입니다.

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))-
>FieldValues[1];
```

ADT 필드의 Fields 속성 사용

각 ADT 필드에는 데이터셋의 *Fields* 속성과 유사한 *Fields* 속성이 있습니다. 데이터셋의 *Fields* 속성처럼 위치별로 자식 필드에 액세스하는 데 사용할 수 있습니다.

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->Fields->
Fields[1]->AsString;
```

이름별로 자식 필드에 액세스할 수도 있습니다.

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->
Fields->FieldByName("City")->AsString;
```

배열 필드 작업

배열 필드는 동일한 타입의 필드 집합으로 구성됩니다. 필드 타입은 스칼라(예를 들어, 부동 소수점 및 문자열) 또는 스칼라가 아닐(ADT) 수 있지만 배열에서는 배열 필드를 사용할 수 없습니다. *TDataSet*의 SparseArrays 속성은 배열 필드의 각 요소에 대해 고유한 *TField* 객체가 만들어지는지 여부를 결정합니다.

배열 필드 타입의 데이터에 액세스하는 방법은 매우 다양합니다. 영구적 필드를 사용하지 않는 경우, 배열 필드의 요소에 액세스하기 전에 데이터셋의 *ObjectView* 속성을 **true**로 설정해야 합니다.

영구적 필드(persistent field) 사용

영구적 필드는 배열 필드의 각 배열 요소에 개별적으로 매핑할 수 있습니다. 예를 들어, 문자열의 6가지 요소 배열인 *TelNos_Array* 배열 필드를 생각해 보십시오. *Customer* 테이블 컴포넌트에 대해 만들어진 다음과 같은 영구적 필드는 *TelNos_Array* 필드와 이 필드의 6가지 요소를 나타냅니다.

```
CustomerTELNOS_ARRAY: TArrayField;
CustomerTELNOS_ARRAY0: TStringField;
CustomerTELNOS_ARRAY1: TStringField;
CustomerTELNOS_ARRAY2: TStringField;
CustomerTELNOS_ARRAY3: TStringField;
CustomerTELNOS_ARRAY4: TStringField;
CustomerTELNOS_ARRAY5: TStringField;
```

이들 영구적 필드의 경우, 다음 코드는 영구적 필드를 사용하여 배열 요소 값을 *TelEdit*라고 하는 에디트 박스에 할당합니다.

```
TelEdit->Text = CustomerTELNOS_ARRAY0->AsString;
```

배열 필드의 FieldValues 속성 사용

배열 필드의 *FieldValues* 속성으로 자식 필드 값에 액세스할 수 있습니다. *Field Values*는 *Variant*를 받아들이고 반환하므로 모든 타입의 자식 필드를 처리하고 변환할 수 있습니다. 예를 들면, 다음과 같습니다.

```
TelEdit->Text = ((TArrayField*)Customer->FieldByName("TelNos_Array"))-
>FieldValues[1];
```

배열 필드의 Fields 속성 사용

*TArrayField*에는 개별 하위 필드를 액세스하는 데 사용할 수 있는 *Fields* 속성이 있습니다. 다음 예제에서는 배열 필드(*OrderDates*)가 Null이 아닌 배열 요소로 리스트 박스를 채우는 데 사용됩니다.

```
for (int i = 0; i< OrderDates->Size; ++i)
  if (!OrderDates->Fields->Fields[i]->IsNull)
    OrderDateListBox->Items->Add(OrderDates->Fields->Fields[i]-
>AsString);
```

데이터셋 필드 작업

데이터셋 필드는 중첩된 데이터셋에 저장된 데이터에 대한 액세스를 제공합니다. NestedDataSet 속성은 중첩된 데이터셋을 참조합니다. 그런 다음, 중첩된 데이터셋의 데이터가 중첩된 데이터셋의 필드 객체를 통해 액세스됩니다.

데이터셋 필드 표시

TDBGrid 컨트롤을 사용하면 데이터셋 필드에 저장된 데이터를 표시할 수 있습니다. *TDBGrid* 컨트롤에서, 데이터셋 필드는 문자열 "(DataSet)"을 사용하여 데이터셋 열의 각 셀에 나타나고 런타임 시 생략 부호 버튼이 오른쪽에 표시됩니다. 생략 부호 버튼을 클릭하면 그리드가 들어 있는 새로운 폼이 나타나면서 현재 레코드의 데이터셋 필드에 연결된 데이터셋이 표시됩니다. 또한 이 폼은 프로그램에서 DB 그리드의 *ShowPopupEditor* 메소드를 사용하여 불러올 수 있습니다. 예를 들어, 그리드의 일곱 번째 열이 데이터셋 필드를 나타낸다면, 다음 코드는 현재 레코드의 필드에 연결된 데이터셋을 표시합니다.

```
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

중첩된 데이터셋의 데이터 액세스

일반적으로 데이터셋 필드는 데이터 인식 컨트롤에 직접 연결되지 않습니다. 오히려 중첩된 데이터셋은 하나의 데이터셋일 뿐이므로 *TDataSet* 자손을 통해 해당 데이터를 액세스합니다. 사용하는 데이터셋의 타입은 부모 데이터셋(데이터셋 필드가 있는 데이터셋)에 의해 결정됩니다. 예를 들어, BDE 호환 데이터셋은 *TNestedTable*을 사용하여 데이터셋 필드에 데이터를 나타내고 클라이언트 데이터셋은 다른 클라이언트 데이터셋을 사용하여 데이터를 나타냅니다.

다음과 같은 방법으로 데이터셋 필드의 데이터에 액세스합니다.

1. 부모 데이터셋용 Fields Editor를 호출하여 영구적 *TDataSetField* 객체를 만듭니다.
2. 데이터셋 필드의 값을 나타내는 데이터셋을 만듭니다. 이 데이터셋은 부모 데이터셋과 호환되는 타입이어야 합니다.
3. 단계 2에서 만든 데이터셋의 *DataSetField* 속성을 단계 1에서 만든 영구적 데이터셋 필드로 설정합니다.

현재 레코드의 중첩된 데이터셋 필드에 값이 있는 경우에는 디테일 데이터셋 컴포넌트가 중첩된 데이터를 가진 레코드를 포함하며, 그렇지 않을 경우에는 디테일 데이터셋이 비어 있을 것입니다.

중첩된 데이터셋에 레코드를 삽입하기 전에, 해당 레코드가 바로 전에 삽입된 경우 마스터 테이블에 있는 이 레코드를 반드시 포스트해야 합니다. 만약 삽입된 레코드가 포스트되지 않으면 중첩된 데이터셋이 포스트되기 전에 자동으로 포스트됩니다.

참조 필드 작업

참조 필드는 포인터나 참조를 다른 ADT 객체에 저장합니다. 이 ADT 객체는 다른 객체 테이블의 단일 레코드입니다. 참조 필드는 항상 데이터셋(객체 테이블)의 단일 레코드를 참조합니다. 참조되는 객체의 데이터는 실제로 중첩된 데이터셋으로 반환되지만, 또한 *TReferenceField*의 *Fields* 속성을 통해 액세스될 수도 있습니다.

참조 필드 표시

TDBGrid 컨트롤에서, 참조 필드는 데이터셋 열의 각 셀에서 평소에는 (Reference)로 런타임 시에는 오른쪽에 있는 생략 부호 버튼으로 데이터셋 열의 각 셀에 지정됩니다. 런타임 시 생략 부호 버튼을 클릭하면, 그리드가 들어 있는 새로운 폼이 나타나면서 현재 레코드의 참조 필드에 연결된 객체를 표시합니다.

또한 이 폼은 프로그램에서 DB 그리드의 *ShowPopupEditor* 메소드를 사용하여 불러올 수 있습니다. 예를 들어, 그리드의 일곱 번째 열이 참조 필드를 나타낸다면, 다음 코드는 현재 레코드의 필드에 연결된 객체를 표시합니다.

```
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

참조 필드의 데이터 액세스

중첩된 데이터셋에 액세스하는 것과 동일한 방법으로 참조 필드의 데이터에 액세스할 수 있습니다.

1. 부모 데이터셋에 대한 Fields Editor를 호출하여 영구적 *TDataSetField* 객체를 만듭니다.
2. 해당 데이터셋 필드의 값을 나타내는 데이터셋을 만듭니다.
3. 단계 2에서 만든 데이터셋의 *DataSetField* 속성을 단계 1에서 만든 영구적 데이터셋 필드로 설정합니다.

참조가 할당되면, 참조 데이터셋은 참조된 데이터를 가진 단일 레코드를 포함합니다. 참조가 Null이면, 참조 데이터셋은 비게 됩니다.

또한 참조 필드의 Fields 속성을 사용하여 참조 필드의 데이터에 액세스할 수도 있습니다. 예를 들어, 다음은 참조 필드 *CustomerRefCity*의 데이터를 *CityEdit*라고 하는 에디트 박스에 할당합니다.

```
CityEdit->Text = CustomerADDRESS_REF->NestedDataSet->Fields->Fields[1]-
>AsString;
```

참조 필드의 데이터가 편집된 경우 실제로 수정된 것은 참조된 데이터입니다.

참조 필드를 할당하려면 우선 SELECT 문을 사용하여 테이블에서 참조를 선택하고 나서 필드를 할당합니다. 예를 들면, 다음과 같습니다.

```
AddressQuery->SQL->Text = "SELECT REF(A)FROM AddressTable (A)WHERE
(A)City = 'San Francisco'";
AddressQuery->Open();
CustomerAddressRef->Assign(AddressQuery->Fields->Fields[0]);
```

장

24

Borland Database Engine 사용

BDE(Borland Database Engine)는 여러 애플리케이션에서 공유할 수 있는 데이터 액세스 메커니즘입니다. BDE는 강력한 API 호출 라이브러리를 정의하여 로컬 및 원격 데이터베이스 서버를 생성, 재구성, 업데이트, 처리하고 이들 서버에서 데이터를 패치(fetch)할 수 있습니다. BDE가 제공하는 일정한 인터페이스를 사용하면 다른 데이터베이스에 연결하는 드라이버를 사용하여 다양한 데이터베이스 서버를 액세스할 수 있습니다. C++Builder 버전에 따라 로컬 데이터베이스(Paradox, dBASE, FoxPro 및 Access)용 드라이버와 InterBase, Oracle, Sybase, Informix, Microsoft SQL server, DB2 등과 같은 원격 데이터 서버용 SQL Links 드라이버 및 고유한 ODBC 드라이버를 제공할 수 있게 하는 ODBC 어댑터 등을 사용할 수 있습니다.

BDE 기반 애플리케이션을 배포하는 경우 애플리케이션에 BDE를 포함시켜야 합니다. 이로 인해 애플리케이션 크기가 늘고 배포가 더 복잡해지지만 BDE를 다른 BDE 기반 애플리케이션과 공유할 수 있고 데이터베이스 조작에 대한 광범위한 지원을 할 수 있게 됩니다. 애플리케이션에서 BDE의 API를 직접 사용할 수도 있지만 컴포넌트 팔레트의 BDE 페이지에 있는 컴포넌트가 이 기능의 대부분을 랩핑합니다.

참고 BDE API에 대한 자세한 내용은 Borland Database Engine을 설치하는 디렉토리에 설치되는 온라인 도움말 파일 BDE32.hlp를 참조하십시오.

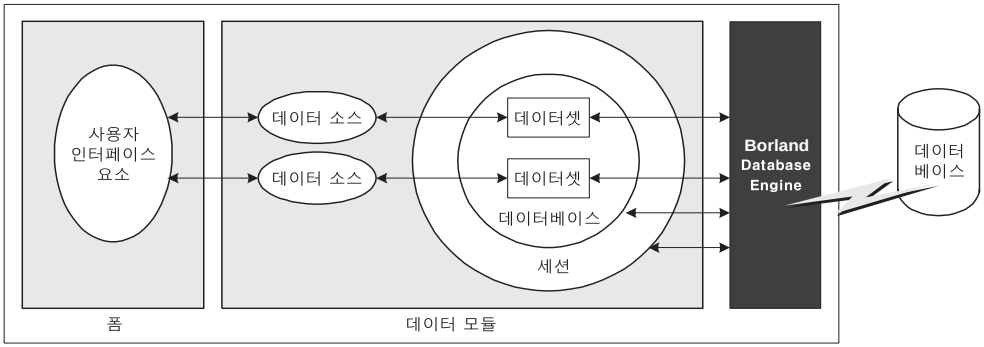
BDE 기반 아키텍처

BDE를 사용하는 경우 애플리케이션은 18-5페이지의 "데이터베이스 아키텍처"에서 설명된 일반적인 데이터베이스 아키텍처에 대한 변형을 사용합니다. BDE 기반 애플리케이션은 모든 C++Builder 애플리케이션에 공통적인 사용자 인터페이스 요소, 데이터 소스 및 데이터셋과 더불어 다음을 포함합니다.

- 트랜잭션을 제어하고 데이터베이스 연결을 관리하는 하나 이상의 데이터베이스 컴포넌트
- 데이터 액세스 작업(예: 데이터베이스 연결)을 분리하고 데이터베이스 그룹을 관리하는 하나 이상의 세션 컴포넌트

그림 24.1은 BDE 기반 애플리케이션의 컴포넌트 간 관계를 보여 줍니다.

그림 24.1 BDE 기반 애플리케이션의 컴포넌트



BDE 호환 데이터셋 사용

BDE 호환 데이터셋은 BDE(Borland Database Engine)를 사용하여 데이터를 액세스합니다. 이 데이터셋은 22장, "데이터셋 이해"에서 설명한 BDE를 사용하여 구현하는 공통 데이터셋 기능을 상속받습니다. 또한 모든 BDE 데이터셋은 다음 용도를 위한 속성, 이벤트 및 메소드를 추가합니다.

- 데이터베이스 및 세션 연결에 데이터셋 연결
- BLOB 캐싱
- BDE 핸들 가져오기

BDE 호환 데이터셋은 다음 세 가지입니다.

- *TTable*. 단일 데이터베이스 테이블의 모든 행과 열을 나타내는 테이블 타입 데이터셋입니다. 테이블 타입 데이터셋에 공통적인 기능에 대한 설명은 22-24페이지의 "테이블 타입 데이터셋 사용"을 참조하고 *TTable*에 고유한 기능에 대한 설명은 24-4페이지의 "TTable 사용"을 참조하십시오.
- *TQuery*. SQL 문을 캡슐화하고 애플리케이션이 결과 집합을 액세스할 수 있도록 하는 쿼리 타입 데이터셋입니다. 쿼리 타입 데이터셋에 공통적인 기능에 대한 설명은 22-40페이지의 "쿼리 타입 데이터셋 사용"을 참조하고 *TQuery*에 고유한 기능에 대한 설명은 24-8페이지의 "TQuery 사용"을 참조하십시오.
- *TStoredProc*. 데이터베이스 서버에 정의된 내장 프로시저를 실행하는 내장 프로시저 타입의 데이터셋입니다. 내장 프로시저 타입의 데이터셋에 공통적인 기능에 대한 설명은 22-48페이지의 "내장 프로시저 타입의 데이터셋 사용"을 참조하고 *TStoredProc*에 고유한 기능에 대한 설명은 24-11페이지의 "TStoredProc 사용"을 참조하십시오.

참고 세 개의 BDE 호환 데이터셋 이외에도 업데이트 캐싱에 사용할 수 있는 BDE 기반 클라이언트 데이터셋(*TBDEClientDataSet*)이 있습니다. 업데이트 캐싱에 대한 자세한 내용은 27-15페이지의 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용"을 참조하십시오.

데이터베이스 및 세션 연결에 데이터셋 연결

BDE 호환 데이터셋이 데이터베이스 서버에서 데이터를 패치하려면 데이터베이스와 세션을 모두 사용해야 합니다.

- 데이터베이스는 특정 데이터베이스 서버에 대한 연결을 나타냅니다. 데이터베이스는 BDE 드라이버와 이 드라이버를 사용하는 특정 데이터베이스 서버 및 이 데이터베이스 서버에 연결하기 위한 연결 매개변수 집합을 식별합니다. 각 데이터베이스는 *TDatabase* 컴포넌트로 나타냅니다. 사용자는 폼이나 데이터 모듈에 추가한 *TDatabase* 컴포넌트에 데이터셋을 연결할 수도 있고 간단히 이름으로 데이터베이스 서버를 식별하고 C++Builder가 암시적 데이터베이스 컴포넌트를 생성하도록 할 수도 있습니다. 대부분의 애플리케이션에 대해 명시적으로 생성한 *TDatabase* 컴포넌트를 사용하는 것이 좋은데 이 데이터베이스 컴포넌트가 로그인 프로세스를 포함하여 연결을 설정하는 방법을 더 잘 제어할 수 있게 하고 트랜잭션을 생성 및 사용할 수 있도록 하기 때문입니다.

  BDE 호환 데이터셋을 데이터베이스와 연결하려면 *DatabaseName* 속성을 사용하십시오. DatabaseName은 명시적 데이터베이스 컴포넌트를 사용하는지 여부에 따라 그리고 명시적 컴포넌트를 사용하지 않는 경우 사용하는 데이터베이스 타입에 따라 다음과 같이 다른 정보를 포함하는 문자열입니다.

  - 명시적 *TDatabase* 컴포넌트를 사용하는 경우 *DatabaseName*은 데이터베이스 컴포넌트의 *DatabaseName* 속성 값입니다.
  - 암시적 데이터베이스 컴포넌트를 사용하고 이 데이터베이스에 BDE 알리아스가 있는 경우 BDE 알리아스를 *DatabaseName* 값으로 지정할 수 있습니다. BDE 알리아스는 해당 데이터베이스의 구성 정보를 포함한 데이터베이스를 나타냅니다. 알리아스에 연결된 구성 정보는 데이터베이스 타입(Oracle, Sybase, InterBase, Paradox, dBASE 등)에 따라 다릅니다. BDE 알리아스를 만들거나 관리하려면 BDE Administration 도구나 SQL 탐색기를 사용하십시오.
  - Paradox 또는 dBASE 데이터베이스용 암시적 데이터베이스 컴포넌트를 사용하는 경우 *DatabaseName*을 사용하여 데이터베이스 테이블이 있는 디렉토리를 간단히 지정할 수 있습니다.

- 세션을 사용하면 애플리케이션의 데이터베이스 연결 그룹을 전역적으로 관리할 수 있습니다. BDE 호환 데이터셋을 애플리케이션에 추가하면 애플리케이션은 자동으로 *Session*이라는 이름의 세션 컴포넌트를 포함하게 됩니다. 애플리케이션에 데이터베이스와 데이터셋 컴포넌트를 추가하면 컴포넌트는 이 디폴트 세션에 자동으로 연결됩니다. 세션은 또한 암호 사용 Paradox 파일에 대한 액세스를 제어하며 네트워크를 통해 Paradox 파일을 공유하기 위한 디렉토리 위치를 지정합니다. 세션의 속성, 이벤트 및 메소드를 사용하여 데이터베이스 연결과 Paradox 파일에 대한 액세스를 제어할 수 있습니다.

  디폴트 세션을 사용하여 애플리케이션에 있는 모든 데이터베이스 연결을 제어할 수 있습니다. 또는 디자인 타임에 추가 세션 컴포넌트를 추가하거나 런타임에 동적으로 추가 세션 컴포넌트를 만들어서 애플리케이션에 있는 데이터베이스 연결 부분 집합을 제어할 수 있습니다. 데이터셋을 명시적으로 만든 세션 컴포넌트에 연결하려면 *SessionName* 속성을 사용하십시오. 애플리케이션에 명시적 세션 컴포넌트를 사용하지 않는다면 이 속성 값을 입력할 필요가 없습니다. 디폴트 세션을 사용하든 *SessionName* 속성을 사용하여 세션을 명시적으로 지정하든 *DBSession* 속성을 읽어서 데이터셋과 연결된 세션을 액세스할 수 있습니다.

참고 세션 컴포넌트를 사용하는 경우 데이터셋의 *SessionName* 속성이 이 데이터셋이 연결된 데이터 컴포넌트의 *SessionName* 속성과 일치해야 합니다.

TDatabase 및 *TSession*에 대한 자세한 내용은 24-12페이지의 "TDatabase를 사용한 데이터베이스 연결" 및 24-16페이지의 "데이터베이스 세션 관리"를 참조하십시오.

BLOB 캐싱

BDE 호환 데이터셋은 모두 *CacheBlobs* 속성을 가지는데 이 속성은 애플리케이션이 BLOB 레코드를 읽을 때 BDE가 로컬에서 BLOB 필드를 캐싱할지 여부를 제어합니다. 디폴트로, *CacheBlobs* 속성 값은 **true**이며 이것은 BDE가 BLOB 필드의 로컬 복사본을 캐싱한다는 의미입니다. BLOB을 캐싱하면 사용자가 레코드를 스크롤할 때 BDE가 데이터베이스 서버에서 BLOB을 반복해서 패치하지 않고 BLOB의 로컬 복사본을 저장할 수 있게 함으로써 애플리케이션 성능을 향상시킬 수 있습니다.

BLOB 을 자주 업데이트하거나 바꾸고 BLOB 데이터 새로 보기가 애플리케이션 성능보다 중요한 애플리케이션 및 환경에서는 *CacheBlobs*를 **false**로 설정하여 애플리케이션이 항상 BLOB 필드의 최신 버전을 확인하도록 할 수 있습니다.

BDE 핸들 가져오기

Borland Database Engine에 대해 직접 API 호출을 할 필요 없이 BDE 호환 데이터셋을 사용할 수 있습니다. BDE 호환 데이터셋은 데이터베이스 및 세션 컴포넌트와 함께 많은 BDE 기능을 캡슐화합니다. 그러나 BDE에 대해 직접 API 호출을 해야 한다면 BDE가 관리하는 리소스에 대한 핸들이 필요할 수 있습니다. 많은 BDE API는 이 핸들을 매개변수로 요청합니다.

모든 BDE 호환 데이터셋은 런타임에 다음과 같은 세 가지 읽기 전용 속성을 사용하여 BDE 핸들을 액세스합니다.

- *Handle*은 데이터셋의 레코드를 액세스하는 BDE 커서에 대한 핸들입니다.
- *DBHandle*은 원본으로 사용하는 테이블 또는 내장 프로시저를 포함하는 데이터베이스에 대한 핸들입니다.
- *DBLocale*은 데이터셋의 BDE 랭귀지 드라이버에 대한 핸들입니다. 로케일은 문자열 데이터에 사용되는 문자 집합과 정렬 순서를 제어합니다.

이 속성들은 데이터셋이 BDE를 통해 데이터베이스에 연결될 때 데이터셋에 자동으로 할당됩니다. BDE API에 대한 자세한 내용은 온라인 도움말 파일 BDE32.HLP를 참조하십시오.

TTable 사용

*TTable*은 원본으로 사용하는 데이터베이스 테이블의 전체 구조와 데이터를 캡슐화합니다. 또한 *TDataSet*에 도입된 모든 기본 기능을 구현하며 테이블 타입 데이터셋에 일반적인 특수 기능들도 모두 구현합니다. *TTable*에 도입된 고유한 기능을 살펴보기 전에 22-24페이지에 있는 테이블 타입 데이터셋에 관한 단원과 "데이터셋 이해"에 설명된 데이터셋의 일반적인 기능에 대해 잘 알아두는 것이 좋습니다.

*TTable*은 BDE 호환 데이터셋이므로 데이터베이스 및 세션에 연결해야 합니다. 24-3페이지의 "데이터베이스 및 세션 연결에 데이터셋 연결"에서 이 연결을 형성하는 방법에 대해 설명합니다. 데이터베이스와 세션에 데이터셋을 연결하면 데이터셋의 *TableName* 속성과 *TableType* 속성(Paradox, dBASE, FoxPro 또는 쉼표로 구분된 ASCII 텍스트 테이블을 사용하는 경우)을 설정하여 특정 데이터베이스 테이블에 데이터셋을 연결할 수 있습니다.

참고 데이터베이스, 세션 또는 데이터베이스 테이블에 대한 데이터셋 연결을 변경하거나 *TableType* 속성을 설정하는 경우에는 테이블을 닫아야 합니다. 그러나 테이블을 닫아 이 속성들을 변경하기 전에 우선 보류 중인 변경 내용을 포스트하거나 버려야 합니다. 캐싱된 업데이트를 사용하는 경우 *ApplyUpdates* 메소드를 호출하여 포스트된 변경 내용을 데이터베이스에 쓰십시오.

TTable 컴포넌트는 유일하게 로컬 데이터베이스 테이블(Paradox, dBASE, FoxPro 및 쉼표로 구분된 ASCII 텍스트 테이블)을 지원합니다. 이와 같은 지원을 구현하는 특수 속성과 메소드에 대해 다음 항목에서 설명합니다.

또한 *TTable* 컴포넌트는 BDE에서 지원하는 일괄 작업(레코드 그룹 전체를 추가, 업데이트, 삭제 또는 복사하기 위한 테이블 수준 작업)을 사용할 수 있습니다. 이 지원에 대한 자세한 내용은 24-7페이지의 "다른 테이블에서 데이터 임포트"를 참조하십시오.

로컬 테이블의 테이블 타입 지정

애플리케이션이 Paradox, dBASE, FoxPro 또는 쉼표로 구분된 ASCII 텍스트 테이블을 액세스하는 경우 BDE는 *TableType* 속성을 사용하여 테이블 타입(해당 예상 구조)을 결정합니다. *TTable*이 데이터베이스 서버의 SQL 기반 테이블을 나타내는 경우에는 *TableType*을 사용하지 않습니다.

디폴트로, *TableType*은 *ttDefault*로 설정됩니다. *TableType*이 *ttDefault*이면 BDE가 테이블 파일 이름 확장자를 통해 테이블 타입을 결정합니다. 표 24.1은 BDE에서 인식하는 파일 확장자 및 확장자에 따라 BDE가 가정하는 테이블 타입을 요약한 것입니다.

표 24.1 파일 확장자에 따라 BDE가 인식하는 테이블 타입

확장자	테이블 타입
파일 확장자 없음	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII 텍스트

로컬 Paradox, dBASE 및 ASCII 텍스트 테이블이 표 24.1에 설명된 파일 확장자를 사용하는 경우에는 *TableType*을 *ttDefault*로 설정해 두어도 됩니다. 그렇지 않은 경우에는 애플리케이션이 *TableType*을 설정하여 올바른 테이블 타입을 지정해야 합니다. 표 24.2는 *TableType*에 할당할 수 있는 값을 보여 줍니다.

표 24.2 테이블 타입 값

값	테이블 타입
ttDefault	BDE가 자동으로 결정하는 테이블 타입
ttParadox	Paradox
ttDBase	dBASE

표 24.2 테이블 타입 값

값	테이블 타입
ttFoxPro	FoxPro
ttASCII	쉼표로 구분된 ASCII 텍스트

로컬 테이블에 대한 읽기/쓰기 액세스 제어

다른 테이블 타입 데이터셋과 마찬가지로 *TTable*은 *ReadOnly* 속성을 사용하여 애플리케이션에서 읽기 및 쓰기 액세스를 제어할 수 있게 해 줍니다.

또한 Paradox, dBASE 및 FoxPro 테이블인 경우 *TTable*을 사용하면 테이블에 대한 다른 애플리케이션의 읽기 및 쓰기 액세스도 제어할 수 있습니다. *Exclusive* 속성은 애플리케이션이 Paradox, dBASE 또는 FoxPro 테이블을 단독 읽기/쓰기 액세스를 하는지 여부를 제어합니다. 이 테이블 타입에 단독 읽기/쓰기 액세스를 하려면 테이블 컴포넌트의 *Exclusive* 속성을 **true**로 설정하고 테이블을 열어야 합니다. 단독 사용으로 테이블을 열면 다른 애플리케이션이 테이블의 데이터를 읽거나 테이블에 데이터를 쓸 수 없습니다. 이미 사용 중인 테이블을 열려고 하면 단독 사용 요청이 받아들여지지 않습니다.

다음 명령문은 단독으로 사용하기 위한 테이블을 엽니다.

```
CustomersTable->Exclusive = true; // Set request for exclusive lock
CustomersTable->Active = true; // Now open the table
```

참고 SQL 테이블에 *Exclusive*를 설정하려 할 수 있지만 일부 서버는 단독 사용을 위한 테이블 잠금을 지원하지 않습니다. 또 일부 서버에서는 단독 사용을 위한 잠금을 허용하더라도 다른 애플리케이션이 해당 테이블의 데이터를 읽는 것도 허용합니다. 단독 사용을 위해 데이터베이스 테이블을 잠그는 방법에 대한 자세한 내용은 해당 서버 문서를 참조하십시오.

dBASE 인덱스 파일 지정

대부분의 서버에서 모든 테이블 타입 데이터셋에 공통적인 메소드를 사용하여 인덱스를 지정합니다. 이 방법에 대해서는 22-25페이지의 "인덱스를 사용하여 레코드 정렬"에서 설명합니다.

그러나 non-production 인덱스 파일이나 dBASE III PLUS 스타일 인덱스(*.NDX)를 사용하는 dBASE 테이블인 경우에는 *IndexFiles* 및 *IndexName* 속성을 사용해야 합니다. *IndexFiles* 속성을 non-production 인덱스 파일의 이름으로 설정하거나 .NDX 파일을 나열합니다. 그런 다음 *IndexName* 속성에서 데이터셋을 정렬하는 데 실제로 사용할 인덱스 하나를 지정합니다.

디자인 타임에 Object Inspector의 *IndexFiles* 속성 값에서 생략 부호 버튼을 클릭하여 Index Files Editor를 호출합니다. non-production 인덱스 파일이나 .NDX 파일을 추가하려면, Index Files 다이얼로그 박스에서 Add 버튼을 클릭하고 Open 다이얼로그 박스에서 파일을 선택합니다. 각 non-production 인덱스 파일 또는 .NDX 파일에 대해 한 번씩 이 과정을 반복합니다. 원하는 인덱스 파일을 모두 추가했으면 Index Files 다이얼로그 박스에서 OK 버튼을 클릭합니다.

런타임 시 프로그램에서 이와 같은 작업을 수행할 수도 있습니다. 이렇게 하려면 문자열 리스트의 속성과 메소드를 사용하여 *IndexFiles* 속성을 액세스합니다. 새로운 인덱스 집합을 추가하는 경우에는 아래 코드와 같이 우선 테이블의 *IndexFiles* 속성에 대해 *Clear* 메소드를 호출하여 기존 항목을 제거합니다 . 그리고 *Add* 메소드를 호출하여 각 non-production 인덱스 파일 또는 .NDX 파일을 추가합니다.

```
Table2->IndexFiles->Clear();
Table2->IndexFiles->Add("Bystate.ndx");
Table2->IndexFiles->Add("Byzip.ndx");
Table2->IndexFiles->Add("Fullname.ndx");
Table2->IndexFiles->Add("St_name.ndx");
```

원하는 non-production 또는 .NDX 인덱스 파일을 추가하면 인덱스 파일에 있는 개별 인덱스의 이름을 사용할 수 있게 되므로 그 이름을 *IndexName* 속성에 할당할 수 있습니다. *GetIndexNames* 메소드를 사용하는 경우와 *IndexDefs* 속성에서 *TIndexDef* 객체를 통해 인덱스 정의를 검사하는 경우에는 인덱스 태그도 나열됩니다. .NDX 파일이 제대로 나열되면 해당 인덱스를 *IndexName* 속성에서 사용하고 있는지 여부에 관계 없이, 데이터가 추가, 변경 또는 삭제되면 .NDX 파일도 자동으로 업데이트됩니다.

아래 예제에서는 *AnimalsTable* 테이블 컴포넌트의 *IndexFiles*을 ANIMALS.MDX라는 non-production 인덱스 파일로 설정하고 해당 *IndexName* 속성을 "NAME"이라는 인덱스 태그로 설정합니다.

```
AnimalsTable->IndexFiles->Add(“ANIMALS.MDX”);
AnimalsTable->IndexName = "NAME";
```

인덱스 파일을 지정한 후 non-production 또는 .NDX 인덱스를 사용하여 다른 인덱스와 같은 작업을 수행합니다. 인덱스 이름을 지정하여 테이블의 데이터를 정렬하고 인덱스 이름을 사용하여 인덱스 기반 검색, 범위 지정 및 master-detail 링크(non-production 인덱스인 경우) 등을 할 수 있습니다. 인덱스 사용에 대한 자세한 내용은 22-24페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.

TTable 컴포넌트와 함께 dBASE III PLUS 스타일 .NDX 인덱스를 사용하는 경우 두 가지의 특별한 고려 사항이 있습니다. 첫째 master-detail 링크의 기초로 .NDX 파일을 사용할 수 없습니다. 둘째 *IndexName* 속성으로 .NDX 인덱스를 활성화할 때 속성 값에 인덱스 이름의 일부로 .NDX 확장자를 포함시켜야 합니다.

```
Table1->IndexName = "ByState.NDX";
TVarRec vr = ("NE");
Table1->FindKey(&vr, 0);
```

로컬 테이블 이름 변경

디자인 타임에 Paradox 또는 dBASE 테이블의 이름을 바꾸려면 테이블 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 컨텍스트 메뉴에서 Rename Table을 선택합니다.

런타임에 Paradox 또는 dBASE 테이블의 이름을 바꾸려면 테이블의 *RenameTable* 메소드를 호출하십시오. 예를 들어 다음 명령문은 Customer 테이블의 이름을 CustInfo로 바꿉니다.

```
Customer->RenameTable(“CustInfo”);
```

다른 테이블에서 데이터 임포트

테이블 컴포넌트의 *BatchMove* 메소드를 사용하여 다른 테이블에서 데이터를 임포트할 수 있습니다. *BatchMove*는 다음을 수행할 수 있습니다.

- 다른 테이블의 레코드를 이 테이블로 복사합니다.
- 이 테이블의 레코드 중 다른 테이블에도 있는 레코드를 업데이트합니다.
- 다른 테이블의 레코드를 이 테이블의 끝에 추가합니다.
- 이 테이블의 레코드 중 다른 테이블에도 있는 레코드를 삭제합니다.

*BatchMove*는 데이터를 임포트할 테이블 이름 및 수행할 임포트 작업을 결정하는 모드 지정 등 두 개의 매개변수를 갖습니다. 표 24.3에서는 모드 지정에 사용할 수 있는 설정에 대해 설명합니다.

표 24.3 BatchMove 임포트 모드

값	의미
batAppend	소스 테이블의 모든 레코드를 이 테이블의 끝에 추가합니다.
batAppendUpdate	소스 테이블의 모든 레코드를 이 테이블의 끝에 추가하고 이 테이블의 기존 레코드 중 소스 테이블의 레코드와 일치하는 레코드를 업데이트합니다.
batCopy	소스 테이블의 모든 레코드를 이 테이블에 복사합니다.
batDelete	이 테이블의 레코드 중 소스 테이블에도 있는 모든 레코드를 삭제합니다.
batUpdate	이 테이블의 기존 레코드 중 소스 테이블의 레코드와 일치하는 레코드를 업데이트합니다.

예를 들어 다음 코드는 현재 테이블의 모든 레코드를 *Customer* 테이블의 레코드 중 현재 인덱스에 있는 필드의 값과 같은 레코드로 업데이트합니다.

```
Table1->BatchMove("CUSTOMER.DB", batUpdate);
```

*BatchMove*는 성공적으로 임포트한 레코드 수를 반환합니다.

주의 *batCopy* 모드를 사용하여 레코드를 임포트하면 이 레코드가 기존 레코드를 덮어씁니다. 기존 레코드를 유지하려면 *batAppend*를 사용하십시오.

*BatchMove*는 BDE에서 지원하는 일괄 작업 중 일부만을 수행합니다. *TBatchMove* 컴포넌트를 사용하면 추가 기능을 사용할 수 있습니다. 테이블 간에 많은 양의 데이터를 이동시켜야 하는 경우에는 테이블의 *BatchMove* 메소드를 호출하는 대신 *TBatchMove*를 사용하십시오. *TBatchMove* 사용에 대한 자세한 내용은 24-47페이지의 "TBatchMove 사용"을 참조하십시오.

TQuery 사용

*TQuery*는 SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE 명령 등과 같이 하나의 데이터 정의 언어(DDL) 또는 데이터 조작 언어(DML) 문을 나타냅니다. 명령에 사용되는 랭귀지는 서버에 따라 다르지만 일반적으로 SQL 랭귀지의 SQL-92 표준을 따릅니다. *TQuery* 는 *TDataSet* 에 도입된 모든 기본 기능과 쿼리 타입 데이터셋에 일반적인 특수 기능을 모두 지원합니다. *TQuery*에 도입된 고유한 기능을 살펴보기 전에 22-24페이지에 있는 쿼리 타입 데이터셋에 대한 단원과 "데이터셋 이해"에서 설명하는 일반적인 데이터베이스 기능에 대해 잘 알아두는 것이 좋습니다.

*TQuery*는 BDE 호환 데이터셋이므로 일반적으로 데이터베이스와 세션에 연결해야 합니다. *TQuery* 를 사용하여 이질적 쿼리를 하는 경우에는 예외입니다. 24-3 페이지의 " 데이터베이스 및 세션 연결에 데이터셋 연결"에서는 이러한 연결을 형성하는 방법에 대해 설명합니다. *SQL* 속성을 사용하여 쿼리에 SQL 문을 지정합니다.

TQuery 컴포넌트는 다음에 있는 데이터를 액세스할 수 있습니다.

- BDE의 일부인 Local SQL을 사용하는 Paradox 또는 dBASE 테이블. Local SQL은 SQL-92 스펙의 부분 집합으로 대부분의 DML과 이러한 타입의 테이블 작업을 위한 충분한 DDL 구문을 지원합니다. 지원되는 SQL 구문에 대한 자세한 내용은 로컬 SQL 도움말 파일인 LOCALSQL.HLP를 참조하십시오.
- InterBase 엔진을 사용하는 Local InterBase Server 데이터베이스. InterBase 의 SQL-92 표준 SQL 구문 지원 및 확장 구문 지원에 대한 자세한 내용은 InterBase *Language Reference*를 참조하십시오.
- Oracle, Sybase, MS-SQL Server, Informix, DB2, InterBase 등과 같은 원격 데이터베이스 서버의 데이터베이스. 원격 서버를 액세스할 데이터베이스 서버에 고유한 적절한 SQL Link 드라이버와 클라이언트 소프트웨어(업체에서 공급)를 설치해야 합니다. 이 서버들에서 지원하는 표준 SQL 구문은 모두 사용할 수 있습니다. SQL 구문, 제한 사항 및 확장에 대한 자세한 내용은 해당 서버의 문서를 참조하십시오.

이질적 쿼리 생성

*TQuery*는 둘 이상의 서버나 테이블 타입(예: Oracle 테이블과 Paradox 테이블의 데이터)에 대해 이질적 쿼리를 지원합니다. 이질적 쿼리를 실행하면 BDE에서는 Local SQL을 사용하여 이 쿼리를 분석하고 처리합니다. BDE가 Local SQL를 사용하므로 확장 및 서버 특정 SQL 구문은 지원되지 않습니다.

다음 단계에 따라 이질적 쿼리를 수행합니다.

1. BDE Administration 도구나 SQL 탐색기를 사용하여 쿼리에서 액세스하는 데이터베이스에 대해 각각의 BDE 알리아스를 정의합니다.
2. *TQuery*의 *DatabaseName* 속성은 비워둡니다. 사용되는 데이터베이스의 이름은 SQL 문에서 지정합니다.
3. SQL 속성에서 실행할 SQL 문을 지정합니다. SQL 문의 각 테이블 이름 앞에 테이블의 데이터베이스에 대한 BDE 알리아스를 콜론으로 묶어서 놓습니다. 그런 다음 이 전체 참조를 따옴표로 묶습니다.
4. *Params* 속성에 쿼리에 대한 매개변수를 설정합니다.
5. 처음으로 쿼리를 실행하기 전에 *Prepare*를 호출하여 쿼리 실행을 준비합니다.
6. 실행하려는 쿼리 타입에 따라 *Open* 또는 *ExecSQL*을 호출합니다.

예를 들어 CUSTOMER 테이블이 있는 Oracle 데이터베이스에 *Oracle1*이라는 알리아스를 정의하고 ORDERS 테이블이 있는 Sybase 데이터베이스에 *Sybase1*이라는 알리아스를 정의한다고 가정합니다. 아래와 같이 이 두 테이블에 대해 간단한 쿼리를 만들 수 있습니다.

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

이질적 쿼리에서 BDE 알리아스를 사용하여 데이터베이스를 지정하는 대신 *TDatabase* 컴포넌트를 사용할 수도 있습니다. *TDatabase*를 정상적으로 구성하여 데이터베이스를 가리키고 *TDatabase::DatabaseName*을 임의의 고유한 값으로 설정한 다음 SQL 문에 BDE 알리아스 이름 대신 이 값을 사용합니다.

편집할 수 있는 결과 집합 얻기

데이터 인식 컨트롤에서 편집할 수 있는 결과 집합을 요청하려면 쿼리 컴포넌트의 *RequestLive* 속성을 **true**로 설정하십시오. *RequestLive*를 **true**로 설정한다고 항상 라이브 결과 집합을 얻을 수 있는 것은 아니지만 BDE는 가능하면 그 요청을 받아들이려고 합니다. 쿼리가 로컬 SQL 파서를 사용하는지 서버의 SQL 파서를 사용하는지 여부에 따라 라이브 결과 집합 요청에 일부 제한이 따릅니다.

- 테이블 이름 앞에 BDE 데이터베이스 알리아스를 놓는 쿼리(예: 이질적 쿼리)와 Paradox 또는 dBASE에 대해 실행하는 쿼리는 BDE에서 Local SQL을 사용하여 분석합니다. 쿼리가 로컬 SQL 파서를 사용하는 경우 단일 테이블 쿼리와 다중 테이블 쿼리 모두에서 BDE는 업데이트할 수 있는 라이브 결과 집합에 대한 확장된 지원을 제공합니다. Local SQL를 사용하는 경우, 쿼리에 다음 중 아무것도 들어있지 않으면 단일 테이블이나 뷰에 대한 쿼리의 라이브 결과 집합이 반환됩니다.
  - SELECT 절의 DISTINCT
  - 조인(내부, 외부 또는 UNION)
  - GROUP BY 또는 HAVING 절을 포함하거나 포함하지 않는 집계 함수
  - 업데이트할 수 없는 기본 테이블 또는 뷰
  - 하위 쿼리
  - 인덱스를 기반으로 하지 않는 ORDER BY 절
- 원격 데이터베이스 서버에 대한 쿼리는 서버에서 분석합니다. *RequestLive* 속성을 **true**로 설정한 경우에는 SQL 문이 Local SQL 표준 및 서버에 부과된 제한 사항을 따라야 하는데 BDE에서 이 SQL 문을 사용하여 데이터 변경 내용을 테이블에 전달해야 하기 때문입니다. 쿼리에 다음 중 아무것도 들어있지 않으면 단일 테이블 쿼리에 대한 라이브 결과 집합 또는 뷰가 반환됩니다.
  - SELECT 문의 DISTINCT 절
  - GROUP BY 또는 HAVING 절을 포함하거나 포함하지 않는 집계 함수
  - 둘 이상의 기본 테이블 또는 업데이트 가능한 뷰에 대한 참조(조인)
  - FROM 절의 테이블이나 다른 테이블을 참조하는 하위 쿼리

애플리케이션이 라이브 결과 집합을 요청하고 받으면 쿼리 컴포넌트의 *CanModify* 속성이 **true**로 설정됩니다. 쿼리에서 라이브 결과 집합을 반환하더라도 결과 집합에 연결된 필드가 들어있거나 업데이트를 시도하기 전에 인덱스를 바꾼 경우에는 결과 집합을 직접 업데이트하지 못할 수도 있습니다. 이런 경우에는 결과 집합을 읽기 전용으로 처리하고 그에 맞게 업데이트해야 합니다.

애플리케이션이 라이브 결과 집합을 요청했는데 SELECT 문 구문에서 그 요청을 허용하지 않는 경우 BDE는 다음 중 하나를 반환합니다.

- Paradox 또는 dBASE에 대해 쿼리하는 경우에는 읽기 전용 결과 집합
- 원격 서버에 대한 SQL 쿼리인 경우에는 오류 코드

읽기 전용 결과 집합 업데이트

애플리케이션이 캐싱된 업데이트를 사용하는 경우에는 읽기 전용 결과 집합으로 반환된 데이터를 업데이트할 수 있습니다.

클라이언트 데이터셋을 사용하여 업데이트를 캐싱하는 경우에는, 쿼리가 여러 테이블을 나타내지 않는 한 클라이언트 데이터셋 또는 이와 관련된 프로바이더가 업데이트 적용을 위한 SQL를 자동으로 생성합니다. 쿼리가 여러 테이블을 나타내는 경우에는 다음과 같은 방법으로 업데이트 적용 방법을 직접 지정해야 합니다.

- 모든 업데이트가 단일 데이터베이스 테이블에 적용되는 경우 *OnGetTableName* 이벤트 핸들러에 원본으로 사용하는 테이블을 지정하여 업데이트할 수 있습니다.
- 업데이트 적용을 보다 잘 제어해야 한다면 다음과 같이 쿼리를 업데이트 객체(*TUpdateSQL*)에 연결할 수 있습니다. 그러면 프로바이더가 자동으로 이 업데이트 객체를 사용하여 업데이트를 적용합니다.
  1. 쿼리의 *UpdateObject* 속성을 사용하고 있는 *TUpdateSQL* 객체로 설정하여 쿼리와 업데이트 객체를 연결합니다.
  2. 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성을 쿼리 데이터에 대해 적절한 업데이트를 수행하는 SQL 문으로 설정합니다.

BDE를 사용하여 업데이트를 캐싱하고 있는 경우에는 업데이트 객체를 사용해야 합니다.

참고 업데이트 객체 사용에 대한 자세한 내용은 24-39페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

TStoredProc 사용

*TStoredProc*은 내장 프로시저를 나타내며, *TDataSet*에 도입된 모든 기본 기능을 구현하며 내장 프로시저 타입 데이터셋에 일반적인 특수 기능을 대부분 구현합니다. *TStoredProc*에 도입된 고유한 기능을 살펴보기 전에 22-48페이지에 있는 내장 프로시저 타입의 데이터셋에 관한 단원과 "데이터셋 이해"에서 설명하는 공통적인 데이터베이스 기능에 대해 잘 알아두는 것이 좋습니다.

*TStoredProc*는 BDE 호환 데이터셋이므로 데이터베이스와 세션에 연결해야 합니다. 24-3페이지의 " 데이터베이스 및 세션 연결에 데이터셋 연결 " 에서 이런 연결을 형성하는 방법에 대해 설명합니다. 데이터셋을 데이터베이스와 세션에 연결하면 *StoredProcName* 속성을 설정하여 데이터셋을 특정 내장 프로시저에 연결할 수 있습니다.

*TStoredProc*은 다음과 같은 점에서 다른 내장 프로시저 타입 데이터셋과 다릅니다.

- 매개변수 연결 방법에 대한 보다 큰 제어력을 제공합니다.
- 오버로드된 Oracle 내장 프로시저를 지원합니다.

매개변수 연결

내장 프로시저를 준비하고 실행하면 내장 프로시저의 입력 매개변수가 자동으로 서버의 매개변수에 연결됩니다.

*TStoredProc*의 *ParamBindMode* 속성을 사용하여 매개변수가 서버의 매개변수에 연결되는 방법을 지정할 수 있습니다. 디폴트로, *ParamBindMode*는 *pbByName*으로 설정되며 이는 내장 프로시저 컴포넌트의 매개변수와 서버의 매개변수를 이름으로 일치시킨다는 것을 의미합니다. 이 방법이 매개변수 연결 방법 중 가장 쉽습니다.

일부 서버에서는 매개변수가 내장 프로시저에 표시되는 순서 값에 따라 매개변수를 연결하는 기능도 지원합니다. 이 경우 매개변수 컬렉션 에디터에서 매개변수를 지정하는 순서가 중요합니다. 처음 지정한 매개변수는 서버의 첫 번째 입력 매개변수와 비교되고, 두 번째 매개변수는 서버의 두 번째 입력 매개변수와 비교됩니다. 서버가 순서 값에 따라 매개변수 연결을 지원하는 경우 *ParamBindMode*를 *pbByNumber*로 설정할 수 있습니다.

팁 *ParamBindMode*를 *pbByNumber*로 설정하려는 경우에는 올바른 순서로 정확한 매개변수 타입을 지정해야 합니다. SQL 탐색기에서 서버의 내장 프로시저 소스 코드를 보고 지정할 매개변수의 올바른 순서와 타입을 확인할 수 있습니다.

Oracle 오버로드된 내장 프로시저 작업

Oracle 서버에서는 내장 프로시저를 오버로드할 수 있습니다. 오버로드된 프로시저는 같은 이름의 다른 프로시저입니다. 내장 프로시저 컴포넌트의 *Overload* 속성을 사용하면 애플리케이션이 실행할 프로시저를 지정할 수 있습니다.

*Overload*가 0(기본값)이면 오버로드가 없는 것입니다. *Overload*가 1이면 내장 프로시저 컴포넌트는 Oracle 서버에 있는 오버로드된 이름을 가지는 내장 프로시저 중 첫째 내장 프로시저를 실행하고 값이 2이면 둘째 내장 프로시저를 실행하는 식입니다.

참고 오버로드된 내장 프로시저는 서로 다른 입력 및 출력 매개변수를 가질 수 있습니다. 자세한 내용은 Oracle 서버 문서를 참조하십시오.

TDatabase를 사용한 데이터베이스 연결

C++Builder 애플리케이션이 BDE(Borland Database Engine)를 사용하여 데이터베이스에 연결하면 해당 연결은 *TDatabase* 컴포넌트에 캡슐화됩니다. BDE 세션의 컨텍스트에서 데이터베이스 컴포넌트는 단일 데이터베이스에 대한 연결을 나타냅니다.

*TDatabase*는 다른 데이터베이스 연결 컴포넌트와 같은 많은 수의 작업을 수행하며 공통 속성, 메소드 및 이벤트를 공유합니다. 이런 공통성에 대해서는 21장의 "데이터베이스에 연결"에서 설명합니다.

공통 속성, 메소드 및 이벤트 이외에, *TDatabase*는 많은 BDE 특정 기능들을 도입합니다. 다음 항목에서 이 기능들에 대해 설명합니다.

데이터베이스 컴포넌트를 세션에 연결

모든 데이터베이스 컴포넌트는 BDE 세션에 연결해야 합니다. *SessionName*을 사용하여 이 연결을 설정합니다. 디자인 타임에 데이터베이스 컴포넌트를 처음으로 생성하면 *SessionName*은 "Default"로 설정되며, 이는 데이터베이스 컴포넌트가 전역 *Session* 변수에서 참조되는 디폴트 세션 컴포넌트에 연결되는 것을 의미합니다.

멀티 스레드 또는 재진입 BDE 애플리케이션에서는 둘 이상의 세션이 필요할 수 있습니다. 여러 세션을 사용해야 하는 경우에는 각 세션에 대해 *TSession* 컴포넌트를 추가해야 합니다. 그런 다음 데이터셋의 *SessionName* 속성을 세션 컴포넌트의 *SessionName* 속성으로 설정하여 데이터셋을 세션 컴포넌트에 연결합니다.

런타임에 *Session* 속성을 읽어서 데이터베이스가 연결되어 있는 세션 컴포넌트에 액세스할 수 있습니다. *SessionName*이 비어 있거나 "Default"인 경우에는 *Session* 속성이 전역 *Session* 변수에서 참조하는 것과 같은 *TSession* 인스턴스를 참조합니다. *Session*을 사용하면 세션의 실제 이름을 몰라도 애플리케이션이 데이터베이스의 상위 세션 컴포넌트의 속성, 메소드 및 이벤트를 액세스하게 할 수 있습니다.

BDE 세션에 대한 자세한 내용은 24-16페이지의 "데이터베이스 세션 관리"를 참조하십시오.

암시적 데이터베이스 컴포넌트를 사용하고 있으면 해당 데이터베이스의 세션은 데이터셋의 *SessionName* 속성에서 지정됩니다.

데이터베이스 및 세션 컴포넌트의 상호 작용 이해

일반적으로 런타임에 생성되는 모든 암시적 데이터베이스 컴포넌트에 적용하는 전역 및 디폴트 동작은 세션 컴포넌트 속성에서 지정합니다. 예를 들어 제어 세션의 *KeepConnections* 속성은 세션에 연결된 데이터셋이 닫히는 경우(기본값)나 모든 데이터셋이 닫혀서 연결이 끊어지는 경우에도 데이터베이스 연결을 유지할지 여부를 지정합니다. 마찬가지로 세션의 디폴트 *OnPassword* 이벤트는 애플리케이션에서 암호를 요구하는 서버의 데이터베이스에 연결하려 할 때 표준 암호 프롬프트 다이얼로그 박스를 표시하게 합니다.

세션 메소드는 약간 달리 적용됩니다. *TSession* 메소드는 컴포넌트가 데이터셋에서 암시적으로 인스턴스화한 것이든 명시적으로 생성한 것이든 모든 데이터베이스 컴포넌트에 영향을 미칩니다. 예를 들어 세션 메소드 *DropConnections*은 개별 데이터베이스 컴포넌트의 *KeepConnection* 속성이 **true**로 설정되어 있더라도 세션의 데이터베이스 컴포넌트에 속한 모든 데이터셋을 닫은 다음 모든 데이터베이스 연결을 끊습니다.

데이터베이스 컴포넌트 메소드는 해당 데이터베이스 컴포넌트에 연결된 데이터셋에만 적용됩니다. 예를 들어 *Database1*이라는 데이터베이스 컴포넌트가 디폴트 세션에 연결되어 있다고 가정합니다. *Database1->CloseDataSets()*는 *Database1*에 연결된 데이터셋만을 닫습니다. 디폴트 세션의 다른 데이터베이스 컴포넌트에 속한 데이터셋 중 열려 있는 데이터셋은 여전히 열린 채로 남습니다.

데이터베이스 식별

AliasName 및 *DriverName*은 상호 배타적인 속성으로 *TDatabase* 컴포넌트가 연결하는 데이터베이스 서버를 식별합니다.

- *AliasName*은 데이터베이스 컴포넌트에 사용할 기존 BDE 알리아스의 이름을 지정합니다. 데이터셋 컴포넌트에 대한 이후의 드롭다운 리스트에 알리아스가 표시되므로 이 이름을 특정 데이터베이스 컴포넌트에 연결할 수 있습니다. 데이터베이스 컴포넌트의 *AliasName*을 지정하면 *DriverName*에 미리 할당되어 있던 값은 지워지는데, 그것은 드라이버 이름은 언제나 BDE 알리아스의 일부이기 때문입니다.

Database Explorer나 BDE Administration 유틸리티를 사용하여 BDE 알리아스를 만들거나 편집할 수 있습니다. BDE 알리아스 생성 및 관리에 대한 자세한 내용은 이 유틸리티에 대한 온라인 문서를 참조합시오.

- *DriverName*은 BDE 드라이버의 이름입니다. 드라이버 이름은 BDE 알리아스의 매개변수 중 하나이지만 *DatabaseName* 속성을 사용하여 데이터베이스 컴포넌트의 로컬 BDE 알리아스를 만드는 경우에는 알리아스 대신 드라이버 이름을 지정할 수도 있습니다. *DriverName*을 지정하면 *AliasName*에 미리 할당된 값은 지워져서 지정한 드라이버 이름과 *AliasName*에 BDE 알리아스의 일부로 지정된 드라이버 이름 사이에 일어날 수 있는 충돌을 방지합니다.

*DatabaseName*을 사용하면 데이터베이스 연결에 고유한 이름을 지정할 수 있습니다. 지정하는 이름은 *AliasName* 또는 *DriverName*에 추가되며 애플리케이션 내에서만 사용됩니다. *DatabaseName*은 BDE 알리아스일 수도 있고 Paradox 및 dBASE 파일인 경우에는 전체 경로 이름일 수도 있습니다. *AliasName*과 마찬가지로 *DatabaseName*도 이후의 데이터셋 컴포넌트의 드롭다운 리스트에 표시되므로 이 이름을 데이터베이스 컴포넌트에 연결할 수 있습니다.

디자인 타임에 BDE 알리아스 지정, BDE 드라이버 할당 또는 로컬 BDE 알리아스 생성 등을 하려면 데이터베이스 컴포넌트를 더블 클릭하여 Database Properties Editor를 호출하십시오.

속성 에디터의 Name 에디트 박스에 *DatabaseName*을 입력할 수 있습니다. *Alias* 속성의 Alias Name 콤보 박스에 기존 BDE 알리아스를 입력하거나 드롭다운 리스트에 있는 기존 알리아스에서 선택할 수 있습니다. Driver 이름 콤보 박스를 사용하면 *DriverName* 속성의 기본 BDE 드라이버 이름을 입력하거나 드롭다운 리스트에 있는 기존 드라이버 이름에서 선택할 수 있습니다.

참고 Database Properties Editor에서는 또한 BDE 연결 매개변수를 보거나 편집할 수 있고 *LoginPrompt* 및 *KeepConnection* 속성의 상태를 설정할 수 있습니다. 연결 매개변수에 대해서는 아래의 "BDE 알리아스 매개변수 설정"을 참조하십시오. *LoginPrompt*에 대한 자세한 내용은 21-4페이지의 "서버 로그인 제어"를 참조하고, *KeepConnection*에 대한 자세한 내용은 24-15페이지의 "Tdatabase를 사용하여 연결 열기"를 참조하십시오.

BDE 알리아스 매개변수 설정

다음과 같은 세 가지 방법으로 디자인 타임에 연결 매개변수를 만들거나 편집할 수 있습니다.

- Database Explorer 또는 BDE Administration 유틸리티를 사용하여 BDE 알리아스 및 매개변수를 만들거나 편집합니다. 이 유틸리티에 대한 자세한 내용은 해당 온라인 도움말을 참조하십시오.
- Object Inspector에서 *Params* 속성을 더블 클릭하여 String List Editor를 호출합니다.
- 데이터 모듈이나 폼에 있는 데이터베이스 컴포넌트를 더블 클릭하여 Database Properties Editor를 호출합니다.

이 방법은 모두 데이터베이스 컴포넌트의 *Params* 속성을 편집합니다. *Params*는 데이터베이스 컴포넌트에 연결된 BDE 알리아스의 데이터베이스 연결 매개변수를 포함하는 문자열 리스트입니다. 몇 가지 일반적인 연결 매개변수는 경로 문, 서버 이름, 스키마 캐싱 크기, 랭귀지 드라이버, SQL 쿼리 모드 등입니다.

Database Properties Editor를 처음 호출하면 BDE 알리아스의 매개변수가 표시되어 있지 않습니다. 현재 매개변수 설정을 보려면 Defaults를 클릭하십시오. 현재 매개변수가 Parameter overrides memo 박스에 표시됩니다. 기존 항목을 편집하거나 새 항목을 추가할 수 있습니다. 기존 매개변수를 지우려면 Clear를 클릭하십시오. 변경한 내용을 적용하려면 OK를 클릭해야 합니다.

런타임에 애플리케이션이 알리아스를 설정하려면 *Params* 매개변수를 직접 편집해야만 합니다. BDE와 함께 SQL Links 드라이버 사용에 관련된 매개변수에 대한 자세한 내용은 SQL Links 도움말 파일을 참조하십시오.

Tdatabase를 사용하여 연결 열기

모든 데이터베이스 연결 컴포넌트와 마찬가지로 *TDatabase*를 사용하여 데이터베이스에 연결하려면 *Connected* 속성을 **true**로 설정하거나 *Open* 메소드를 호출합니다. 이 프로세스에 대해서는 21-3페이지의 "데이터베이스 서버에 연결"에서 설명합니다. 데이터베이스 연결이 설정된 후에는 활성 데이터셋이 적어도 하나라도 있는 한 연결은 유지됩니다. 활성 데이터셋이 더 이상 없는 경우에는 데이터베이스 컴포넌트의 *KeepConnection* 속성을 **true**로 설정하지 않았으면 연결이 끊어집니다.

애플리케이션에서 원격 데이터베이스 서버에 연결하는 경우에는 애플리케이션은 BDE 및 Borland SQL Links 드라이버를 사용하여 연결을 설정합니다. BDE는 사용자가 제공하는 ODBC 드라이버와도 통신할 수 있습니다. 연결하기에 앞서 애플리케이션을 위해 SQL Links 또는 ODBC 드라이버를 구성해야 합니다. SQL Links 및 ODBC 매개변수는 데이터베이스 컴포넌트의 *Params* 속성에 저장됩니다. SQL Links 매개변수에 대한 자세한 내용은 온라인 *SQL Links User's Guide*를 참조하십시오. *Params* 속성 편집에 대해서는 24-14페이지의 "BDE 알리아스 매개변수 설정"을 참조하십시오.

네트워크 프로토콜 작업

적절한 SQL Links 또는 ODBC 드라이버 구성 작업의 일부로 드라이버 구성 옵션에 따라 서버에서 사용하는 네트워크 프로토콜(예: SPX/IPX 또는 TCP/IP)을 지정해야 할 수 있습니다. 대부분의 경우 네트워크 프로토콜 구성은 서버의 클라이언트 설치 소프트웨어를 사용하여 처리됩니다. ODBC에서는 ODBC 드라이버 관리자를 사용하여 드라이버 설치도 확인해야 할 수 있습니다.

클라이언트와 서버 간에 초기 연결을 설정할 때 문제가 많이 발생할 수 있습니다. 문제가 발생하면 다음과 같은 문제 해결 체크 리스트를 사용해 보십시오.

- 서버의 클라이언트사이드 연결이 제대로 구성되어 있습니까?
- 검색 경로에 연결 및 데이터베이스 드라이버의 DLL이 있습니까?
- TCP/IP를 사용하는 경우,
  - TCP/IP 통신 소프트웨어가 설치되어 있습니까? 적절한 WINSOCK.DLL이 설치되어 있습니까?
  - 클라이언트의 HOSTS 파일에 서버의 IP 주소가 등록되어 있습니까?
  - DNS(Domain Name Services)가 제대로 구성되어 있습니까?
  - 서버를 ping할 수 있습니까?

문제 해결에 대한 자세한 내용은 온라인 *SQL Links User's Guide*와 서버 문서를 참조하십시오.

ODBC 사용

애플리케이션이 ODBC 데이터 소스(예: Btrieve)를 사용할 수 있습니다. ODBC 드라이버 연결에는 다음이 필요합니다.

- 업체에서 공급한 ODBC 드라이버
- Microsoft ODBC 드라이버 관리자
- BDE Administration 유틸리티

ODBC 드라이버 연결의 BDE 알리아스를 설정하려면 BDE Administration 유틸리티를 사용하십시오. 자세한 내용은 BDE Administration 유틸리티의 온라인 도움말 파일을 참조하십시오.

데이터 모듈의 데이터베이스 컴포넌트 사용

데이터베이스 컴포넌트를 데이터 모듈에 안전하게 넣을 수 있습니다. 그러나 데이터베이스 컴포넌트를 포함하는 데이터 모듈을 Object Repository에 넣고 다른 사용자가 데이터베이스 컴포넌트를 상속받을 수 있게 하려면 데이터베이스 컴포넌트의 *HandleShared* 속성을 **true**로 설정해야 전역 네임스페이스 충돌을 방지할 수 있습니다.

데이터베이스 세션 관리

BDE 기반 애플리케이션의 데이터베이스 연결, 드라이버, 커서, 쿼리 등은 하나 이상의 BDE 세션의 컨텐스트 내에서 관리됩니다. 세션이 데이터베이스 연결과 같은 일련의 데이터베이스 액세스 작업을 분리하므로 애플리케이션의 다른 인스턴스를 시작할 필요가 없습니다.

모든 BDE 기반 데이터베이스 애플리케이션은 디폴트 BDE 세션을 캡슐화하는 *Session* 이라는 디폴트 세션 컴포넌트를 자동으로 포함하게 됩니다. 애플리케이션에 추가되는 컴포넌트는 자동으로 디폴트 세션(이 세션의 *SessionName*은 "Default"임)에 연결됩니다. 디폴트 세션은 다른 세션에 연결되지 않은 모든 데이터베이스 컴포넌트를 전역적으로 관리합니다. 이런 점은 데이터베이스 컴포넌트가 런타임에 사용자가 만든 데이터베이스 컴포넌트에 연결되지 않은 데이터셋을 열 때 세션이 만드는 암시적 데이터베이스든 애플리케이션이 명시적으로 만드는 영구적 데이터베이스든 관계 없습니다. 디자인 타임에는 디폴트 세션이 데이터 모듈에 표시되지 않지만 런타임에 코드를 통해 디폴트 세션의 속성과 메소드를 액세스할 수 있습니다.

애플리케이션에서 다음과 같은 작업을 해야 하는 경우를 제외하고는 디폴트 세션을 사용하기 위해 코드를 작성할 필요는 없습니다.

- 명시적으로 세션을 활성 또는 비활성화하나 세션 데이터베이스의 열기 기능을 사용 또는 사용할 수 없게 하는 경우
- 암시적으로 생성한 데이터베이스 컴포넌트의 디폴트 속성을 지정하는 것과 같이 세션 속성을 수정하는 경우
- 데이터베이스 연결 관리(예: 사용자 작업에 따라 데이터베이스 연결 열기 및 닫기)와 같이 세션 메소드를 실행하는 경우
- 세션 이벤트(예: 애플리케이션이 암호 사용 Paradox 또는 dBASE 테이블을 액세스하려는 경우)에 응답하는 경우

- 네트워크에 있는 Paradox 테이블을 액세스하기 위한 *NetFileDir* 속성 및 *PrivateDir* 속성과 같은 Paradox 디렉토리 위치를 로컬 하드 드라이브에 설정하여 성능을 향상시켜야 하는 경우
- 세션을 사용하는 데이터베이스와 데이터셋에 대해 가능한 데이터베이스 연결 구성을 설명하는 BDE 알리아스를 관리하려는 경우

데이터베이스 컴포넌트를 디자인 타임에 애플리케이션에 추가하든 런타임에 동적으로 만들든 특별히 그 컴포넌트를 다른 세션에 할당하지 않는 한 자동으로 디폴트 세션에 연결합니다. 데이터베이스 컴포넌트에 연결되지 않는 데이터셋을 여는 경우 C++Builder는 자동으로 다음을 수행합니다.

- 런타임에 그 데이터셋의 데이터베이스 컴포넌트를 만듭니다.
- 이 데이터베이스 컴포넌트를 디폴트 세션에 연결합니다.
- 디폴트 세션의 속성을 기준으로 데이터베이스 컴포넌트의 일부 키 속성을 초기화합니다. 이런 속성 중에 가장 중요한 속성 하나는 *KeepConnections*인데, 이 속성은 애플리케이션에서 데이터베이스 연결을 유지할지 끊을지를 결정합니다.

디폴트 세션은 대부분의 애플리케이션에서 사용할 수 있는 널리 적용 가능한 기본값 집합을 제공합니다. 디폴트 세션에서 이미 열어 둔 데이터베이스에 대해 컴포넌트에서 동시 쿼리를 수행하는 경우에는 데이터베이스 컴포넌트를 명시적으로 명명된 세션에 연결하기만 하면 됩니다. 이 경우, 동시 쿼리는 각각의 세션에서 실행해야 합니다. 멀티 스레드 데이터베이스 애플리케이션에서도 스레드마다 고유한 세션을 가지므로 여러 세션이 필요합니다.

애플리케이션은 필요에 따라 추가 세션 컴포넌트를 만들 수 있습니다. BDE 기반 데이터베이스 애플리케이션은 자동으로 *Sessions*라는 세션 리스트 컴포넌트를 포함시키며 이 리스트를 사용하여 모든 세션 컴포넌트를 관리할 수 있습니다. 여러 세션 관리에 대한 자세한 내용은 24-28페이지의 "여러 세션 관리"를 참조하십시오.

세션 컴포넌트를 데이터 모듈에 안전하게 넣을 수 있습니다. 그러나 하나 이상의 세션 컴포넌트를 포함하는 데이터 모듈을 Object Repository에 넣는 경우에는 *AutoSessionName* 속성을 **true**로 설정해야 사용자가 세션 컴포넌트를 상속받을 때 네임스페이스 충돌을 방지할 수 있습니다.

세션 활성화

*Active*는 부울 속성으로 세션에 연결된 데이터베이스와 데이터셋이 열려 있는지 여부를 결정합니다. 이 속성을 사용하여 세션의 데이터베이스와 데이터셋 연결의 현재 상태를 읽거나 변경할 수 있습니다. *Active*가 **false**(기본값)이면 세션에 연결된 모든 데이터베이스와 데이터셋이 닫혀 있는 것이고 **true**이면 데이터베이스와 데이터셋이 열려 있는 것입니다.

세션은 처음 생성될 때 활성화하고 이후에 세션의 *Active* 속성이 **false**에서 **true**로 변경될 때(예: 세션에 연결된 데이터베이스 또는 데이터셋이 열려 있고 현재 열려 있는 다른 데이터베이스 또는 데이터셋이 없는 경우)마다 활성화합니다. *Active*를 **true**로 설정하면 세션의 *OnStartup* 이벤트를 실행하고, paradox 디렉토리 위치를 BDE에 등록하고, 세션에서 사용할 수 있는 BDE 알리아스를 지정하는 *ConfigMode* 속성을 등록합니다. *OnStartup* 이벤트 핸들러를 작성하여 *NetFileDir*, *PrivateDir* 및 *ConfigMode* 속성이 BDE에 등록되기 전에 이 속성들을 초기화하거나 다른 특정 세션 시작 작업을 수행할 수 있습니다. *NetFileDir* 및 *PrivateDir* 속성에 대한 자세한 내용은 24-23페이지의 "Paradox 디렉토리 위치 지정"을 참조하고, ConfigMode에 대한 자세한 내용은 24-24페이지의 "BDE 알리아스 작업"을 참조하십시오.

세션이 활성이면 *OpenDatabase* 메소드를 호출하여 세션의 데이터베이스 연결을 열 수 있습니다.

데이터 모듈이나 폼에 배치한 세션 컴포넌트에 대해 *Active*를 **false**로 설정하면 열려 있는 데이터베이스나 데이터셋이 있는 경우 이들을 닫습니다. 런타임에 데이터베이스와 데이터셋을 닫으면 이들과 연결된 이벤트가 실행될 수 있습니다.

참고 디자인 타임에 디폴트 세션에 대해 *Active*를 **false**로 설정할 수 없습니다. 런타임에 디폴트 세션을 닫을 수는 있지만 닫지 않는 것이 좋습니다.

런타임에 세션의 *Open* 및 *Close* 메소드를 사용하여 디폴트 세션 이외의 세션을 활성화 또는 비활성화할 수도 있습니다. 예를 들어 다음 한 줄의 코드는 세션에 열려 있는 모든 데이터베이스와 데이터셋을 닫습니다.

```
Session1->Close();
```

위의 코드는 Session1의 *Active* 속성을 **false**로 설정합니다. 세션의 *Active* 속성이 **false**일 때 이후에 애플리케이션이 데이터베이스나 데이터셋을 열려고 하면 *Active*를 **true**로 다시 설정하고 세션의 *OnStartup* 이벤트 핸들러가 있는 경우 이 핸들러를 호출합니다. 런타임에 세션 다시 활성화를 명시적으로 코딩할 수도 있습니다. 다음 코드는 *Session1*을 다시 활성화합니다.

```
Session1->Open();
```

참고 세션이 활성인 경우 개별 데이터베이스 연결도 열거나 닫을 수 있습니다. 자세한 내용은 24-19 페이지의 "데이터베이스 연결 닫기"를 참조하십시오.

디폴트 데이터베이스 연결 동작 지정

*KeepConnections*은 런타임에 생성한 명시적 데이터베이스 컴포넌트의 *KeepConnection* 속성의 기본값을 제공합니다. *KeepConnection*은 데이터베이스의 모든 데이터셋이 닫힐 때 데이터베이스 컴포넌트에 설정된 데이터베이스 연결이 어떻게 되는지를 지정합니다. **true**(기본값)이면 활성인 데이터셋이 없더라도 일정하거나 *영구적* 데이터베이스 연결이 유지됩니다. **false**이면 모든 데이터셋이 닫히면 곧바로 데이터베이스 연결이 끊깁니다.

참고 데이터 모듈이나 폼에 명시적으로 놓은 데이터베이스 컴포넌트의 연결 지속성은 데이터베이스 컴포넌트의 *KeepConnection* 속성으로 제어합니다. 다르게 설정된 경우, 데이터베이스 컴포넌트의 *KeepConnection* 속성이 항상 세션의 *KeepConnections* 속성을 오버라이드합니다. 세션에서 개별 데이터베이스 연결을 제어하는 것에 대한 자세한 내용은 24-19페이지의 "데이터베이스 연결 관리"를 참조하십시오.

원격 서버의 데이터베이스에 연결된 모든 데이터셋을 자주 열고 닫는 애플리케이션에 대해서는 *KeepConnections*을 **true**로 설정해야 합니다. 이렇게 설정하면 세션의 수명 동안 연결을 한 번만 열고 닫게 되므로 네트워크 트래픽을 줄이고 데이터 액세스 속도를 늘립니다. 반대의 경우에는 애플리케이션이 연결을 끊거나 다시 설정할 때마다 데이터베이스를 데이터베이스를 연결하고 분리해야 하는 오버헤드가 발생합니다.

참고 세션의 *KeepConnections*가 **true**인 경우에도 *DropConnections* 메소드를 호출하여 모든 암시적 데이터베이스 컴포넌트의 비활성 데이터베이스 연결을 닫거나 해제할 수 있습니다. *DropConnections*에 대한 자세한 내용은 24-20페이지의 "비활성 데이터베이스 연결 끊기"를 참조하십시오.

데이터베이스 연결 관리

세션 컴포넌트를 사용하여 세션 내의 데이터베이스 연결을 관리할 수 있습니다. 세션 컴포넌트는 다음을 하는 데 사용할 수 있는 속성과 메소드를 포함합니다.

- 데이터베이스 연결 열기
- 데이터베이스 연결 닫기
- 모든 비활성 임시 데이터베이스 연결 닫기 및 해제
- 특정 데이터베이스 연결 찾기
- 모든 열려 있는 데이터베이스 연결 내에서 반복

데이터베이스 연결 열기

세션 내에서 데이터베이스 연결을 열려면 *OpenDatabase* 메소드를 호출합니다. *OpenDatabase*는 열려는 데이터베이스 이름에 해당하는 하나의 매개변수를 가지니다. 이 이름은 BDE 알리아스이거나 데이터베이스 컴포넌트의 이름입니다. Paradox 또는 dBASE인 경우 이 이름은 전체 경로 이름이 될 수 있습니다. 예를 들어 다음 명령문은 디폴트 세션을 사용하여 BCDEMOS 알리아스가 가리키는 데이터베이스에 대한 데이터베이스 연결을 열려고 합니다.

```
TDatabase *BCDemosDatabase = Session->OpenDatabase("BCDEMOS");
```

*OpenDatabase*는 세션이 아직 활성이 아닌 경우 활성화하고 세션의 데이터베이스 컴포넌트 중에 *DatabaseName* 속성이 지정한 데이터베이스 이름과 일치하는 데이터베이스 컴포넌트가 있는지 확인합니다. 이름이 기존 데이터베이스 컴포넌트와 일치하지 않으면 *OpenDatabase*가 지정한 이름을 사용하여 임시 데이터베이스 컴포넌트를 만듭니다. 끝으로 *OpenDatabase*가 데이터베이스 컴포넌트의 *Open* 메소드를 호출하여 서버에 연결합니다. *OpenDatabase*를 호출할 때마다 데이터베이스의 참조 카운팅이 1씩 증가합니다. 이 참조 카운팅이 0보다 큰 채로 있는 한 데이터베이스는 열려 있습니다.

데이터베이스 연결 닫기

개별 데이터베이스 연결을 닫으려면 *CloseDatabase* 메소드를 호출합니다. *CloseDatabase*를 호출하면 *OpenDatabase*를 호출하면 증가하는 데이터베이스의 참조 카운팅이 1씩 감소합니다. 데이터베이스의 참조 카운팅이 0이 되면 데이터베이스가 닫힙니다. *CloseDatabase*는 닫을 데이터베이스에 해당하는 하나의 매개변수를 가집니다. *OpenDatabase* 메소드를 사용하여 데이터베이스를 열면 이 매개변수가 *OpenDatabase*의 반환 값으로 설정될 수 있습니다.

```
Session->CloseDatabase(BCDemosDatabase);
```

지정한 데이터베이스 이름이 임시(암시적) 데이터베이스 컴포넌트에 연결되어 있고 세션의 *KeepConnections* 속성이 **false**이면 데이터베이스 컴포넌트가 해제되어 결과적으로 연결이 닫힙니다.

참고 *KeepConnections*가 **false**인 경우 데이터베이스 컴포넌트에 연결된 마지막 데이터셋이 닫히면 임시 데이터베이스 컴포넌트는 자동으로 닫히고 해제됩니다. 그 전에 언제라도 애플리케이션이 *CloseDatabase*를 호출하여 강제로 닫을 수 있습니다. *KeepConnections*가 **true**인 경우에 임시 데이터베이스 컴포넌트를 해제하려면 데이터베이스 컴포넌트의 *Close* 메소드를 호출한 다음 세션의 *DropConnections* 메소드를 호출하십시오.

참고 영구적 데이터베이스 컴포넌트인 경우에는 *CloseDatabase*를 호출하더라도 연결이 실제로 닫히지 않습니다. 연결을 닫으려면 데이터베이스 컴포넌트의 *Close* 메소드를 직접 호출하십시오.

다음과 같은 두 가지 방법으로 세션 내의 모든 데이터베이스 연결을 닫을 수 있습니다.

- 세션의 *Active* 속성을 **false**로 설정합니다.
- 세션에 대해 *Close* 메소드를 호출합니다.

*Active*를 **false**로 설정하면 C++Builder는 자동으로 *Close* 메소드를 호출합니다. *Close*는 임시 데이터베이스 컴포넌트를 해제하고 영구적 데이터베이스 컴포넌트 각각에 대해 *Close* 메소드를 호출하여 모든 활성 데이터베이스의 연결을 끊습니다. 끝으로 *Close*는 세션의 BDE 핸들을 **NULL**로 설정합니다.

비활성 데이터베이스 연결 끊기

세션의 *KeepConnections* 속성이 **true**(기본값)이면 임시 데이터베이스 컴포넌트에서 사용하는 데이터셋이 모두 닫혀도 해당 컴포넌트의 데이터베이스 연결은 유지됩니다. 이런 연결을 제거하고 세션에서 비활성 임시 데이터베이스 컴포넌트를 모두 해제하려면 *DropConnections* 메소드를 호출하십시오. 예를 들어 다음 코드는 디폴트 세션의 모든 비활성 임시 데이터베이스 컴포넌트를 해제합니다.

```
Session->DropConnections();
```

하나 이상의 데이터셋이 활성인 임시 데이터베이스 컴포넌트는 이 호출로 연결이 끊기거나 해제되지 않습니다. 이런 컴포넌트를 해제하려면 *Close*를 호출하십시오.

데이터베이스 연결 검색

세션의 *FindDatabase* 메소드를 사용하여 지정한 데이터베이스 컴포넌트가 이미 세션에 연결되어 있는지 확인합니다. *FindDatabase*는 하나의 매개변수를 가지며 그것은 검색할 데이터베이스 이름입니다. 이 이름은 BDE 알리아스거나 데이터베이스 컴포넌트 이름입니다. Paradox 또는 dBASE인 경우 이 이름이 전체 경로 이름이 될 수도 있습니다.

*FindDatabase*는 일치하는 데이터베이스 컴포넌트를 찾으면 이것을 반환하고, 그렇지 않으면 **NULL**을 반환합니다.

다음 코드는 디폴트 세션에서 BCDEMOS 알리아스를 사용하는 데이터베이스 컴포넌트를 검색하고 없는 경우에는 새로 생성해서 엽니다.

```
TDatabase *DB = Session->FindDatabase("BCDEMOS");
if ( !DB )                                  // Database does not exist for
session so
  DB = Session->OpenDatabase("BCDEMOS"); // create and open it
if (DB && DB->Connected)
{
  if (!DB->InTransaction)
  {
    DB->StartTransaction();
    f
  }
}
```

세션의 데이터베이스 컴포넌트 내에서 반복

Databases 및 *DatabaseCount*이라는 두 가지 세션 컴포넌트 속성을 사용하여 세션에 연결된 모든 활성 데이터베이스 컴포넌트 내에서 반복하게 할 수 있습니다.

*Databases*는 세션에 연결된 현재 활성인 모든 데이터베이스 컴포넌트의 배열이고, *DatabaseCount*는 이 배열에 있는 데이터베이스의 수입니다. 세션의 수명 동안 연결이 열리거나 닫힐 때 *Databases* 및 *DatabaseCount*는 변합니다. 예를 들어 세션의 *KeepConnections* 속성이 **false**이고 런타임에 필요에 따라 모든 데이터베이스 컴포넌트를 만드는 경우, 고유한 데이터베이스를 열 때마다 *DatabaseCount*는 1씩 증가합니다. 고유한 데이터베이스가 닫힐 때마다, *DatabaseCount*는 1씩 감소합니다. *DatabaseCount*가 0이면 세션에 현재 활성인 데이터베이스 컴포넌트는 없습니다.

다음 예제 코드는 디폴트 세션에 있는 각 활성 데이터베이스의 *KeepConnection* 속성을 **true** 로 설정합니다.

```
if (Session->DatabaseCount > 0)
  for (int MaxDbCount = 0; MaxDbCount < Session->DatabaseCount;
MaxDbCount++)
    Session->Databases[MaxDbCount]->KeepConnection = true;
```

암호 사용 Paradox 및 dBASE 테이블

세션 컴포넌트는 암호 사용 Paradox 및 dBASE 테이블의 암호를 저장할 수 있습니다. 암호를 세션에 추가하면 애플리케이션이 해당 암호를 사용하는 테이블을 열 수 있습니다. 세션에서 암호를 제거하면 다시 추가하기 전까지는 애플리케이션이 그 암호를 사용하는 테이블을 열 수 없습니다.

AddPassword 메소드 사용

AddPassword 메소드를 사용하면 애플리케이션이 액세스 암호를 요구하는 암호화된 Paradox 또는 dBASE 테이블을 열기 전에 세션에 암호를 제공할 수 있습니다. 세션에 암호를 추가하지 않은 경우에는 애플리케이션이 암호 사용 테이블을 열려고 할 때 사용자에게 암호를 묻는 다이얼로그 박스가 표시됩니다.

*AddPassword*는 한 개의 매개변수 즉, 사용할 암호를 포함하는 문자열을 가집니다. 필요한 만큼 많이 *AddPassword*를 호출하여 다른 암호를 사용하는 테이블을 액세스하기 위한 암호를 한 번에 하나씩 추가할 수 있습니다.

```
AnsiString PassWrd;
PassWrd = InputBox(”Enter password”, ”Password:”, ””);
Session->AddPassword(PassWrd);
try
{
  Table1->Open();
}
catch(...)
{
  ShowMessage”Couldnotopentable
  Application->Terminate();
}
```

참고 위에서 *InputBox* 함수는 데모로 사용한 것입니다. 실제 애플리케이션에서는 입력하는 암호를 보이지 않게 하는 *PasswordDialog* 함수나 사용자 정의 폼과 같은 암호 입력 기능을 사용합니다.

PasswordDialog 함수 다이얼로그 박스의 Add 버튼은 *AddPassword* 메소드와 같은 결과를 만듭니다.

```
if (PasswordDlg(Session))
  Table1->Open();
else
  ShowMessage("No password given, could not open table!"};
```

RemovePassword 및 RemoveAllPasswords 메소드 사용

*RemovePassword*는 이전에 메모리에 추가한 암호를 삭제합니다. *RemovePassword*는 매개변수 하나 즉, 삭제할 암호를 포함하는 문자열을 가집니다.

```
Session->RemovePassword("secret");
```

*RemoveAllPasswords*는 이전에 메모리에 추가한 암호를 모두 삭제합니다.

```
Session->RemoveAllPasswords();
```

GetPassword 메소드 및 OnPassword 이벤트 사용

OnPassword 이벤트는 Paradox 및 dBASE 테이블에서 암호를 요구할 때 애플리케이션이 그 암호를 제공하는 방법을 제어할 수 있도록 합니다 . 디폴트 암호 처리 동작을 오버라이드하려는 경우에도 *OnPassword* 이벤트에 대한 핸들러를 제공하십시오. 핸들러를 제공하지 않으면 C++Builder는 암호 입력을 위한 디폴트 다이얼로그 박스를 표시하기만 하고 아무런 동작도 제공하지 않습니다. 즉, 테이블 열기 시도가 성공하거나 예외가 발생됩니다.

OnPassword 이벤트의 핸들러를 제공하는 경우 이벤트 핸들러에서 두 가지 작업을 합니다. 즉, *AddPassword* 메소드를 호출하고 이벤트 핸들러의 *Continue* 매개변수를 **true**로 설정합니다. *AddPassword* 메소드는 테이블의 암호로 사용될 문자열을 세션에 전달합니다. *Continue* 매개변수는 C++Builder 에게 이 테이블 열려고 할 때 더 이상 암호를 물을 필요가 없다는 것을 표시합니다. *Continue*의 기본값은 **false**이므로 명시적으로 이 값을 **true**로 설정해야 합니다. 이 이벤트 핸들러의 실행이 종료된 후 *Continue*가 **false**이면 *AddPassword* 메소드를 사용하여 올바른 암호가 전달된 경우에도 *OnPassword* 이벤트가 다시 발생합니다. 이 이벤트 핸들러가 실행된 후 *Continue*가 **true**이고 *AddPassword*로 전달된 문자열이 잘못된 암호인 경우에는 테이블 열기 시도가 실패하고 예외가 발생됩니다.

*OnPassword*는 두 가지 경우에 실행될 수 있습니다. 첫째는 세션에 아직 올바른 암호를 제공하지 않은 암호 사용 테이블(dBASE 또는 Paradox)을 열려고 하는 경우입니다. 해당 테이블의 암호를 이미 제공한 경우에는 *OnPassword* 이벤트가 발생하지 않습니다.

다른 경우는 *GetPassword* 메소드를 호출하는 경우입니다. *GetPassword*는 *OnPassword* 이벤트를 생성하지만 *OnPassword* 이벤트 핸들러가 없는 경우에는 디폴트 암호 다이얼로그 박스를 표시합니다. 이 메소드는 *OnPassword* 이벤트 핸들러나 디폴트 다이얼로그 박스에서 세션에 암호를 추가한 경우 **true**를 반환하고 입력된 사항이 없을 경우에는 **false**를 반환합니다.

다음 예제에서는 *Password* 메소드를 전역 *Session* 객체의 *OnPassword* 속성에 할당하여 이 메소드를 디폴트 세션의 *OnPassword* 이벤트로 지정합니다.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Session->OnPassword = Password;
}
```

Password 메소드에서 *InputBox* 함수는 사용자에게 암호를 묻는 다이얼로그 박스를 표시합니다. 그런 다음 *AddPassword* 메소드는 다이얼로그 박스에 입력한 암호를 프로그래밍 방식으로 세션에 제공합니다.

```
void __fastcall TForm1::Password(TObject *Sender, bool &Continue)
{
  AnsiString PassWrd = InputBox("Enter password", "Password:", "");
  Session->AddPassword(PassWrd);
  Continue = (PassWrd > "");
}
```

OnPassword 이벤트(즉, *Password* 이벤트 핸들러)는 아래에 나타난 것과 같이 암호 사용 테이블을 열려고 하면 실행됩니다. *OnPassword* 이벤트에서 사용자에게 암호를 묻는 다이얼로그 박스를 표시하더라도 잘못된 암호를 입력하거나 다른 잘못이 있으면 테이블 열기 시도는 여전히 실패합니다.

```
void __fastcall TForm1::OpenTableBtnClick(TObject *Sender)
{
  try
  {
    // this line triggers the OnPassword event
    Table1->Open();
  }
  // exception if cannot open table
  catch(...)
  {
    ShowMessage("Could not open table!");
    Application->Terminate();
  }
}
```

Paradox 디렉토리 위치 지정

NetFileDir 및 *PrivateDir* 등 두 가지 세션 컴포넌트 속성은 Paradox 테이블에서만 사용됩니다.

*NetFileDir*은 Paradox 네트워크 제어 파일인 PDOXUSRS.NET을 포함하는 디렉토리를 지정합니다. 이 파일은 네트워크 드라이브에 있는 Paradox 테이블의 공유를 제어합니다. Paradox 테이블을 공유해야 하는 모든 애플리케이션은 네트워크 제어 파일에 대해 같은 디렉토리 (일반적으로 네트워크 서버에 있는 디렉토리)를 지정해야 합니다. C++Builder는 *NetFileDir*의 값을 해당 데이터베이스 알리아스의 BDE(Borland Database Engine) 설정 파일에서 파생시킵니다. *NetFileDir*을 직접 설정하면 제공하는 값이 BDE 구성 설정을 오버라이드하므로 새 값이 올바른지 확인해야 합니다.

디자인 타임에 Object Inspector에서 *NetFileDir*의 값을 지정할 수 있으며 런타임에는 코드로 *NetFileDir*을 변경하거나 설정할 수 있습니다. 다음 코드는 디폴트 세션의 *NetFileDir*을 애플리케이션이 실행되는 디렉토리 위치로 설정합니다.

```
Session->NetFileDir = ExtractFilePath(ParamStr(0));
```

참고 *NetFileDir*을 변경하려면 애플리케이션에 열려 있는 Paradox 파일이 없어야 합니다. 런타임에 *NetFileDir*을 변경하는 경우 이 값이 네트워크 사용자들이 공유하는 올바른 네트워크 디렉토리를 가리키는지 확인하십시오.

*PrivateDir*은 로컬 SQL 문을 처리하기 위해 BDE에서 생성하는 것과 같은 임시 테이블 처리 파일을 저장하는 디렉토리를 지정합니다. *PrivateDir* 속성에 값을 지정하지 않으면 자동으로 BDE가 초기화될 때의 현재 디렉토리가 자동으로 사용됩니다. 애플리케이션이 네트워크 파일 서버에서 직접 실행되는 경우에는 런타임에 *PrivateDir*을 사용자의 로컬 하드 드라이브로 설정하고 데이터베이스를 열면 애플리케이션 성능을 향상시킬 수 있습니다.

참고 디자인 타임에 *PrivateDir*을 설정하고 IDE에서 세션을 열지 마십시오. 이렇게 하면 IDE에서 애플리케이션을 실행할 때 Directory is busy 오류가 생성됩니다.

다음 코드는 디폴트 세션의 *PrivateDir* 속성 설정을 사용자의 C:\TEMP 디렉토리로 변경합니다.

```
Session->PrivateDir = "C:\\TEMP";
```

중요 *PrivateDir*을 드라이브의 루트 디렉토리로 설정하지 마십시오. 항상 하위 디렉토리를 지정하십시오.

BDE 알리아스 작업

세션에 연결된 각 데이터베이스 컴포넌트는 BDE 알리아스(Paradox 및 dBASE 테이블을 액세스하는 경우에는 전체 경로 이름으로 대체할 수도 있음)를 가집니다. 세션은 수명 동안 알리아스를 생성, 수정 및 삭제할 수 있습니다.

AddAlias 메소드는 SQL 데이터베이스 서버의 새 BDE 알리아스를 만듭니다. *AddAlias*는 세 가지 매개변수 즉, 알리아스 이름을 포함하는 문자열, 사용할 SQL Links 드라이버를 지정하는 문자열 및 알리아스의 매개변수에 채울 문자열 리스트 등을 취합니다. 예를 들어 다음 구문은 *AddAlias*를 사용하여 디폴트 세션에 InterBase 서버를 액세스하기 위한 새 알리아스를 추가합니다.

```
TStringList *AliasParams = new TStringList();
try
{
  AliasParams->Add("OPEN MODE=READ");
  AliasParams->Add("USER NAME=TOMSTOPPARD");
  AliasParams->Add("SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB");
  Session->AddAlias("CATS", "INTRBASE", AliasParams);
  f
}
catch (...)
{
  delete AliasParams;
  throw;
}
delete AliasParams;
```

*AddStandardAlias*는 Paradox, dBASE 또는 ASCII 테이블에 대한 BDE 알리아스를 만듭니다. *AddStandardAlias*는 세 가지 매개변수 즉, 알리아스 이름, 액세스할 Paradox 또는 dBASE 테이블의 전체 경로 및 확장자가 없는 테이블을 열려고 할 때 사용할 디폴트 드라이버의 이름 등을 취합니다. 예를 들어 다음 구문은 *AddStandardAlias*를 사용하여 Paradox 테이블을 액세스하기 위한 새 알리아스를 만듭니다.

```
Session->AddStandardAlias("MYBCDEMOS= "C:\\TESTING\\DEMOS\\", "Paradox");
```

세션에 알리아스를 추가하면 BDE는 알리아스의 복사본을 메모리에 저장하여 이 세션과 *ConfigMode* 속성에 *cfmPersistent*를 포함하는 다른 세션에서 사용할 수 있게 합니다. *ConfigMode* 설정은 세션의 데이터베이스에서 사용될 수 있는 알리아스 타입을 설명합니다. 디폴트 설정은 *cmAll*이며 이것이 [*cfmVirtual*, *cfmPersistent*, *cfmSession*] 설정으로 바뀝니다. *ConfigMode*가 *cmAll*이면 세션은 세션 내에서 생성된 모든 알리아스(*cfmSession*), 사용자 시스템의 BDE 설정 파일에 있는 모든 알리아스(*cfmPersistent*) 및 BDE가 메모리에 보유하는 모든 알리아스(*cfmVirtual*) 등을 볼 수 있습니다. *ConfigMode*를 변경하여 세션의 데이터베이스가 사용할 수 있는 BDE 알리아스를 제한할 수 있습니다. 예를 들어 *ConfigMode*를 *cfmSession*으로 설정하면 세션이 세션 내에서 생성된 알리아스만 볼 수 있도록 제한합니다. BDE 설정 파일과 메모리에 있는 다른 모든 알리아스는 사용할 수 없습니다.

새로 생성한 알리아스를 모든 세션과 다른 애플리케이션에서 사용할 수 있게 하려면 세션의 *SaveConfigFile* 메소드를 사용하십시오. *SaveConfigFile*은 메모리의 알리아스를 BDE 설정 파일에 써서 다른 BDE 호환 애플리케이션이 읽거나 사용할 수 있게 합니다.

알리아스를 생성한 후 *ModifyAlias*를 호출하여 알리아스의 매개변수를 변경할 수 있습니다. *ModifyAlias* 는 변경할 알리아스 이름 및 변경할 매개변수와 그 값을 포함하는 문자열 리스트 등 두 개의 매개변수를 취합니다. 예를 들어 다음 명령문은 *ModifyAlias*를 사용하여 디폴트 세션에서 CATS 알리아스의 OPEN MODE 매개변수를 READ/WRITE로 변경합니다.

```
TStringList *List = new TStringList();
List->Clear();
List->Add("OPEN MODE=READ/WRITE");
Session->ModifyAlias("CATS", List);
delete List;
```

세션에서 이전에 생성한 알리아스를 삭제하려면 *DeleteAlias* 메소드를 호출하십시오. *DeleteAlias*는 삭제할 알리아스 이름을 매개변수로 취합니다. *DeleteAlias*는 알리아스를 세션에서 사용할 수 없게 만듭니다.

참고 이전에 *SaveConfigFile*을 호출하여 알리아스를 파일에 쓴 경우에는 *DeleteAlias*를 사용해도 BDE 설정 파일에서 알리아스를 제거하지는 않습니다. *DeleteAlias*를 호출한 후 설정 파일에서 알리아스를 제거하려면 *SaveConfigFile*을 다시 호출하십시오.

세션 컴포넌트는 매개변수 정보와 드라이버 정보 등 BDE 알리아스에 대한 정보를 검색하기 위한 다섯 개의 메소드를 제공합니다. 이 메소드는 다음과 같습니다.

- *GetAliasNames*는 세션이 액세스할 수 있는 알리아스를 나열합니다.
- *GetAliasParams*는 지정한 알리아스의 매개변수를 나열합니다.
- *GetAliasDriverName*은 알리아스에서 사용되는 BDE 드라이버 이름을 반환합니다.
- *GetDriverNames*는 세션에서 사용할 수 있는 모든 BDE 드라이버 리스트를 반환합니다.
- *GetDriverParams*는 지정한 드라이버의 드라이버 매개변수를 반환합니다.

세션의 정보 메소드 사용에 대한 자세한 내용은 아래의 "세션에 대한 정보 검색"을 참조하십시오. BDE 알리아스 및 이 알리아스가 함께 작동하는 SQL Links 드라이버에 대한 자세한 내용은 BDE 온라인 도움말인 BDE32.HLP를 참조하십시오.

세션에 대한 정보 검색

세션의 정보 메소드를 사용하여 세션 및 그 데이터베이스 컴포넌트에 대한 정보를 검색할 수 있습니다. 예를 들어 세션에 알려진 모든 알리아스의 이름을 검색하는 메소드, 세션에서 사용되는 특정 데이터베이스 컴포넌트와 연결된 테이블 이름을 검색하는 메소드 등이 있습니다. 표 24.4에서는 세션 컴포넌트의 정보 메소드를 요약합니다.

표 24.4 세션 컴포넌트의 데이터베이스 관련 정보 메소드

메소드	용도
GetAliasDriverName	특정 데이터베이스 알리아스의 BDE 드라이버를 검색합니다.
GetAliasNames	데이터베이스의 BDE 알리아스 리스트를 검색합니다.
GetAliasParams	데이터베이스의 지정한 BDE 알리아스의 매개변수 리스트를 검색합니다.
GetConfigParams	BDE 설정 파일에서 구성 정보를 검색합니다.
GetDatabaseNames	현재 사용 중인 *TDatabase* 컴포넌트의 이름과 BDE 알리아스의 리스트를 검색합니다.
GetDriverNames	현재 설치된 모든 BDE 드라이버의 이름을 검색합니다.
GetDriverParams	지정한 BDE 드라이버의 매개변수 리스트를 검색합니다.
GetStoredProcNames	지정한 데이터베이스의 모든 내장 프로시저의 이름을 검색합니다.
GetTableNames	지정한 데이터베이스에 대해 지정한 패턴과 일치하는 모든 테이블의 이름을 검색합니다.
GetFieldNames	지정한 데이터베이스의 지정한 테이블에 있는 모든 필드의 이름을 검색합니다.

GetAliasDriverName 이외의 모든 메소드는 이 값들의 집합을 문자열 리스트로 반환하며 이 리스트는 애플리케이션에서 선언되고 유지됩니다. *GetAliasDriverName*은 세션에서 사용되는 특정 데이터베이스 컴포넌트의 현재 BDE 드라이버의 이름인 단일 문자열을 반환합니다.

예를 들어 다음 코드는 디폴트 세션에 알려진 모든 데이터베이스 컴포넌트와 알리아스의 이름을 반환합니다.

```
TStringList *List = new TStringList();
try
{
  Session->GetDatabaseNames(List);
  f
}
catch (...)
{
  delete List;
  throw;
}
delete List;
```

추가 세션 생성

세션을 만들어서 디폴트 세션을 보충할 수 있습니다. 디자인 타임에 데이터 모듈이나 폼에 세션을 추가하고, Object Inspector에서 그 속성을 설정하고, 그에 대한 이벤트 핸들러를 작성하고, 그 메소드를 호출하는 코드를 작성할 수 있습니다. 런타임에도 세션을 만들고, 그 속성을 설정하고, 그 메소드를 호출할 수 있습니다.

참고 애플리케이션이 데이터베이스에 대해 동시 쿼리를 실행하거나 멀티 스레드인 경우를 제외하고는 추가 세션을 만드는 것은 옵션입니다.

런타임에 동적으로 세션 컴포넌트를 만들 수 있게 하려면 다음 단계를 따르십시오.

1. *TSession* 변수를 선언합니다.
2. **new** 연산자를 사용하여 새 세션을 인스턴스화합니다. 이 연산자는 *TSession* 생성자를 호출하여 새 세션을 만들고 초기화합니다. 이 생성자는 비어 있는 데이터베이스 컴포넌트 리스트를 세션에 설정하고, *KeepConnections* 속성을 **true**로 설정하며, 애플리케이션의 세션 리스트 컴포넌트가 유지 보수하는 세션 리스트에 세션을 추가합니다.
3. 새 세션의 *SessionName* 속성을 고유한 이름으로 설정합니다. 이 속성은 세션에 데이터베이스 컴포넌트를 연결하는 데 사용됩니다. *SessionName* 속성에 대한 자세한 내용은 24-27페이지의 "세션 이름 지정"을 참조하십시오.
4. 세션을 활성화하고 경우에 따라 그 속성을 조정합니다.

또한, *TSessionList*의 *OpenSession* 메소드를 사용하여 세션을 만들고 열 수 있습니다. *OpenSession*을 사용하는 것이 **new** 연산자를 사용하는 것보다 안전한데, 그 이유는 *OpenSession*은 세션이 아직 없는 경우에만 그 세션을 만들기 때문입니다. *OpenSession*에 대한 자세한 내용은 24-28페이지의 "여러 세션 관리"를 참조하십시오.

다음 코드는 새로운 세션 컴포넌트를 만들고 이름을 할당하여, 이후의 데이터베이스 작업(여기에는 표시되지 않음)을 위해 이 세션을 엽니다. 세션을 사용한 다음에는 *Free* 메소드를 호출하여 세션을 삭제합니다.

참고 디폴트 세션은 삭제하지 마십시오.

```
TSession *SecondSession = new TSession(Form1);
try
{
  SecondSession->SessionName = "SecondSession";
  SecondSession->KeepConnections = false;
  SecondSession->Open();
  f
}
__finally
{
  delete SecondSession;
};
```

세션 이름 지정

세션의 *SessionName* 속성을 사용하여 데이터베이스와 데이터셋을 세션에 연결할 수 있도록 세션의 이름을 지정합니다. 디폴트 세션인 경우 *SessionName*은 "Default"이며, 추가로 생성하는 각 세션에 대해서는 *SessionName* 속성을 고유한 값으로 설정해야 합니다.

데이터베이스와 데이터셋 컴포넌트는 세션 컴포넌트의 *SessionName* 속성과 일치하는 *SessionName* 속성을 가집니다. 데이터베이스나 데이터셋 컴포넌트의 *SessionName* 속성을 비워두면 이 컴포넌트는 자동으로 디폴트 세션에 연결됩니다. 데이터베이스나 데이터셋 컴포넌트의 *SessionName*을 생성하는 세션 컴포넌트의 *SessionName*과 일치하는 이름으로 설정할 수도 있습니다.

다음 코드는 디폴트 *TSessionList* 컴포넌트인 *Sessions*의 *OpenSession* 메소드를 사용하여 새 세션 컴포넌트를 열고, 그 *SessionName* 속성을 "InterBaseSession"으로 설정하고, 세션을 활성화하고, 해당 세션에 기존 데이터베이스 컴포넌트 *Database1*을 연결합니다.

```
TSession *IBSession = Sessions->OpenSession("InterBaseSession");
Database1->SessionName = "InterBaseSession";
```

여러 세션 관리

여러 스레드를 사용하여 데이터베이스 작업을 수행하는 단일 애플리케이션을 만드는 경우 각 스레드에 대해 추가 세션을 하나씩 만들어야 합니다. 디자인 타임에 Component 팔레트의 BDE 페이지에 있는 세션 컴포넌트를 데이터 모듈이나 폼에 넣을 수 있습니다.

중요 세션 컴포넌트를 넣을 때 세션의 *SessionName* 속성을 고유한 값으로 설정해야 디폴트 세션의 *SessionName* 속성과 충돌하지 않습니다.

런타임에 애플리케이션에서 필요로 하는 스레드 수(즉, 세션)가 고정되어 있다고 미리 가정한 경우에 디자인 타임에 세션 컴포넌트를 넣습니다. 그러나 애플리케이션이 세션을 동적으로 생성해야 하는 경우가 분명히 있을 수 있습니다. 세션을 동적으로 생성하려면 런타임에 전역 *Sessions* 객체의 *OpenSession* 메소드를 호출하십시오.

*OpenSession*은 애플리케이션의 모든 세션 이름에 대해 고유한 세션 이름을 단일 매개변수로 취합니다. 다음 코드는 고유하게 생성된 이름으로 새 세션을 동적으로 만들고 활성화합니다.

```
Sessions->OpenSession("RunTimeSession" + IntToStr(Sessions->Count + 1));
```

이 명령문은 현재 세션 수를 검색하고 이 값에 하나를 더해서 새 세션의 고유한 이름을 생성합니다. 런타임에 세션을 동적으로 생성 및 삭제하는 경우 이 예제 코드는 예상한 대로 작동하지 않습니다. 그렇지만 이 예제는 *Sessions*의 속성과 메소드를 사용하여 여러 세션을 관리하는 방법을 보여 줍니다.

*Sessions*는 BDE 기반 데이터베이스 애플리케이션을 위해 자동으로 인스턴스화되는 *TSessionList* 타입의 변수입니다. *Sessions*의 속성과 메소드를 사용하여 멀티 스레드 데이터베이스 애플리케이션에서 여러 세션을 추적할 수 있습니다. 표 24.5는 *TSessionList* 컴포넌트의 속성과 메소드를 요약합니다.

표 24.5 TSessionList 속성 및 메소드

속성 또는 메소드	용도
Count	세션 리스트에 있는 활성 및 비활성 세션의 수를 반환합니다.
FindSession	지정한 이름의 세션을 찾고 그 세션에 대한 포인터를 반환하거나 지정한 이름의 세션이 없는 경우 NULL을 반환합니다. 비어 있는 세션 이름을 전달하면 *FindSession*은 디폴트 세션인 *Session*에 대한 포인터를 반환합니다.
GetSessionNames	문자열 리스트를 현재 인스턴스화된 모든 세션 컴포넌트로 채웁니다. 이 과정에서 디폴트 세션에 대한 "Default"라는 문자열이 항상 한 개 이상 추가됩니다.
List	지정한 세션 이름의 세션 컴포넌트를 반환합니다. 지정한 이름의 세션이 없으면 예외가 발생합니다.
OpenSession	지정한 세션 이름으로 새 세션을 생성 및 활성화하거나 이 이름의 기존 세션을 다시 활성화합니다.
Sessions	순서 값으로 세션 리스트를 액세스합니다.

멀티 스레드 애플리케이션에서 *Sessions* 속성 및 메소드를 사용하는 예제로 데이터베이스 연결을 열려고 할 때 어떤 일이 발생하는지 고려해 보십시오. 연결이 이미 존재하는지 확인하려면 *Sessions* 속성을 사용하여 디폴트 세션부터 세션 리스트의 각 세션을 검사합니다. 각 세션 컴포넌트에 대해 *Databases* 속성을 검사하여 해당 데이터베이스가 열려 있는지 확인합니다. 다른 스레드가 이미 해당 데이터베이스를 사용하고 있다면 리스트의 다음 세션을 검사합니다.

기존 스레드가 해당 데이터베이스를 사용하고 있지 않으면 해당 세션 내에서 연결을 열 수 있습니다.

반면에 모든 기존 스레드가 해당 데이터베이스를 사용하고 있으면 새 세션을 열고 여기에서 다른 데이터베이스 연결을 열어야 합니다.

스레드마다 고유한 데이터 모듈 복사본을 가지는 멀티 스레드 애플리케이션에서 세션을 포함하는 데이터 모듈을 복제하려는 경우 *AutoSessionName* 속성을 사용하여 데이터 모듈의 모든 데이터셋이 올바른 세션을 사용하도록 할 수 있습니다. *AutoSessionName*을 **true**로 설정하면 세션이 런타임에 생성될 때 고유한 이름을 동적으로 생성하게 합니다. 그런 다음 이 이름이 데이터 모듈의 모든 데이터셋에 할당되어 명시적으로 설정된 세션 이름을 오버라이드합니다. 따라서 각 스레드는 고유한 세션을 가지게 되고, 각 데이터셋은 자체 모듈에서 이 세션을 사용하게 됩니다.

BDE에서 트랜잭션 사용

디폴트로, BDE에서는 애플리케이션 트랜잭션을 암시적으로 제어합니다. 애플리케이션이 암시적 트랜잭션 제어 상태인 경우 원본으로 사용하는 데이터베이스에 기록되는 데이터셋의 레코드에 각각의 트랜잭션이 사용됩니다. 암시적 트랜잭션을 사용하면 레코드 업데이트 충돌을 최소화하고 데이터베이스를 일관성 있게 볼 수 있습니다. 반면에 데이터베이스에 기록되는 데이터의 각 행이 각기 다른 트랜잭션에서 수행되므로 네트워크 트래픽이 과도해지고 애플리케이션 성능을 떨어뜨릴 수 있습니다. 또한 암시적 트랜잭션 제어는 두 개 이상의 레코드에 걸쳐 있는 논리 작업을 보호하지 않습니다.

트랜잭션을 명시적으로 제어하는 경우에는 가장 효율적인 시간을 선택하여 트랜잭션을 시작, 커밋 및 롤백할 수 있습니다. 다중 사용자 환경에서 애플리케이션을 개발하며, 특히 원격 SQL 서버에 대해 애플리케이션을 실행하는 경우에는 트랜잭션을 명시적으로 제어해야 합니다.

다음과 같은 두 가지 상호 배타적인 방법으로 BDE 기반 데이터베이스 애플리케이션에서 트랜잭션을 명시적으로 제어할 수 있습니다.

- 데이터베이스 컴포넌트를 사용하여 트랜잭션을 제어합니다. 데이터베이스 컴포넌트의 메소드와 속성을 사용하면 특정 데이터베이스나 서버에 종속되지 않는 명확하고 이식 가능한 애플리케이션을 제공할 수 있다는 기본 장점이 있습니다. 이러한 트랜잭션 제어는 모든 데이터베이스 연결 컴포넌트에서 지원합니다. 이에 대한 설명은 21-6페이지의 "트랜잭션 관리"를 참조하십시오.

- 쿼리 컴포넌트에 통과(passthrough) SQL을 사용하여 원격 SQL 또는 ODBC 서버에 SQL 문을 직접 전달합니다. 통과 SQL의 기본 장점은 특정 서버의 고급 트랜잭션 관리 기능(예: 스키마 캐싱)을 사용할 수 있다는 점입니다. 서버의 트랜잭션 관리 모델의 장점을 알아 보려면 해당 데이터베이스 서버 문서를 참조하십시오. 통과 SQL 사용에 대한 자세한 내용은 아래의 "통과(passthrough) SQL 사용"을 참조하십시오.

로컬 데이터베이스에서 작업하는 경우에는 데이터베이스 컴포넌트를 사용하여 명시적 트랜잭션만을 만들 수 있습니다. 로컬 데이터베이스는 통과 SQL을 지원하지 않습니다. 그러나 로컬 트랜잭션을 사용하는 데에는 제한 사항이 따릅니다. 로컬 트랜잭션 사용에 대한 자세한 내용은 24-31페이지의 "로컬 트랜잭션 사용"을 참조하십시오.

참고 업데이트 캐싱에 필요한 트랜잭션 수를 최소화할 수 있습니다. 캐싱된 업데이트에 대한 자세한 내용은 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용" 및 24-31페이지의 "BDE를 사용하여 업데이트 캐싱"을 참조하십시오.

통과(passthrough) SQL 사용

통과 SQL 에서 *TQuery*, *TStoredProc* 또는 *TUpdateSQL* 컴포넌트를 사용하여 SQL 트랜잭션 제어 문을 원격 데이터베이스 서버에 직접 전달할 수 있습니다. BDE는 SQL 문을 처리하지 않습니다. 통과 SQL을 사용하면 서버에서 제공하는 트랜잭션 컨트롤을 직접 사용할 수 있습니다. 특히, 이 컨트롤이 표준이 아닌 경우에 유용합니다.

통과 SQL을 사용하여 트랜잭션을 제어하려면 다음을 수행해야 합니다.

- 적당한 SQL Links 드라이버를 설치해야 합니다. C++Builder를 설치할 때 "Typical" 설치를 선택한 경우에는 모든 SQL Links 드라이버가 올바로 설치되어 있습니다.
- 네트워크 프로토콜을 구성해야 합니다. 자세한 내용은 네트워크 관리자에게 문의하십시오.
- 원격 서버의 데이터베이스에 액세스할 수 있어야 합니다.
- SQL 탐색기를 사용하여 SQLPASSTHRU MODE를 NOT SHARED로 설정해야 합니다. SQLPASSTHRU MODE는 BDE와 통과 SQL 문이 같은 데이터베이스 연결을 공유할 수 있는지 여부를 지정합니다. 대부분의 경우 SQLPASSTHRU MODE는 SHARED AUTOCOMMIT로 설정됩니다. 그러나 트랜잭션 제어 문을 사용하는 경우에는 데이터베이스 연결을 공유할 수 없습니다. SQLPASSTHRU 모드에 대한 자세한 내용은 BDE Administration 유틸리티의 도움말 파일을 참조하십시오.

참고 SQLPASSTHRU MODE가 NOT SHARED인 경우에는 서버에 SQL 트랜잭션 문을 전달하는 데이터셋과 그렇지 않은 데이터셋에 대해 각각의 데이터베이스 컴포넌트를 사용해야 합니다.

로컬 트랜잭션 사용

BDE에서는 Paradox, dBASE, Access 및 FoxPro 테이블에 대한 로컬 트랜잭션을 지원합니다. 코딩하는 측면에서는 로컬 트랜잭션과 원격 서버에 대한 트랜잭션 사이에 다른 점이 없습니다.

참고 로컬 Paradox, dBASE, Access 및 FoxPro 테이블에서 트랜잭션을 사용하는 경우에는 *TransIsolation*을 기본값인 *tiReadCommitted* 대신 *tiDirtyRead*로 설정하십시오. 로컬 테이블에 대해 *TransIsolation*을 *tiDirtyRead* 이외의 값으로 설정하면 BDE 오류가 반환됩니다.

로컬 테이블에 대해 트랜잭션이 시작하면 해당 테이블에 대해 수행되는 업데이트가 기록됩니다. 각 로그 레코드에는 레코드의 기존 레코드 버퍼가 포함됩니다. 트랜잭션이 활성 상태이면 업데이트되는 레코드는 트랜잭션이 커밋 또는 롤백될 때까지 잠깁니다. 롤백되는 경우에는 기존 레코드 버퍼가 업데이트된 레코드에 적용되어 업데이트되기 전의 상태로 복원합니다.

로컬 트랜잭션에는 SQL 서버나 ODBC 드라이버에 대한 트랜잭션보다 많은 제한이 따릅니다. 특히 다음 제한 사항이 로컬 트랜잭션에 적용됩니다.

- 자동 크래시 복구를 지원하지 않습니다.
- 데이터 정의 문을 지원하지 않습니다.
- 임시 테이블에 대해 트랜잭션을 실행할 수 없습니다.
- *TransIsolation* 레벨은 *tiDirtyRead*로만 설정해야 합니다.
- Paradox인 경우 올바른 인덱스가 있는 테이블에만 로컬 트랜잭션을 수행할 수 있습니다. 인덱스가 없는 Paradox 테이블의 데이터는 롤백할 수 없습니다.
- 제한된 수의 레코드만 잠그거나 수정할 수 있습니다. Paradox 테이블에서는 255개의 레코드로 제한되며, dBASE에서는 100개로 제한됩니다.
- BDE ASCII 드라이버에 대해 트랜잭션을 실행할 수 없습니다.
- 다음 경우를 제외하고는 트랜잭션 동안 테이블에서 커서를 닫으면 트랜잭션을 롤백합니다.
  - 여러 테이블이 열려 있는 경우
  - 변경 사항이 없는 테이블에서 커서를 닫는 경우

BDE를 사용하여 업데이트 캐싱

업데이트를 캐싱하는 방법으로는 클라이언트 데이터셋 (*TBDEClientDataSet*) 을 사용하는 방법이나 데이터셋 프로바이더를 사용하여 BDE 데이터셋을 클라이언트 데이터셋에 연결하는 방법이 좋습니다. 클라이언트 데이터셋을 사용하는 것과 관련된 장점에 대해서는 27-15페이지의 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용"에서 설명합니다.

그러나 간단한 작업인 경우에는 대신 BDE를 사용하여 업데이트를 캐싱할 수 있습니다. BDE 호환 데이터셋과 *TDatabase* 컴포넌트는 캐싱된 업데이트를 처리하기 위한 내장 속성, 메소드 및 이벤트를 제공합니다. 이들 대부분은 클라이언트 데이터셋을 사용하여 업데이트를 캐싱하는 경우에 클라이언트 데이터셋과 데이터셋 프로바이더에서 사용하는 속성, 메소드 및 이벤트와 곧바로 일치합니다. 다음 표에서는 이 속성, 메소드 및 이벤트와 함께 *TBDEClientDataSet* 에서 그에 해당하는 속성, 메소드 및 이벤트를 나열합니다.

표 24.6 캐싱된 업데이트를 위한 속성, 메소드 및 이벤트

BDE 호환 데이터셋 (또는 TDatabase)	TBDEClientDataSet	용도
CachedUpdates	클라이언트 데이터셋인 경우 항상 업데이트를 캐싱하므로 필요 없음	캐싱된 업데이트가 데이터셋에 적용되는지 여부를 결정합니다.
UpdateObject	*BeforeUpdateRecord* 이벤트 핸들러 사용 또는 *TClientDataSet*을 사용하는 경우 BDE 호환 소스 데이터셋에서 *UpdateObject* 속성 사용	읽기 전용 데이터셋을 업데이트하기 위한 업데이트 객체를 지정합니다.
UpdatesPending	*ChangeCount*	로컬 캐시에 데이터베이스에 적용해야 하는 업데이트된 레코드가 있는지 여부를 나타냅니다.
UpdateRecordTypes	*StatusFilter*	캐싱된 업데이트를 적용할 때 보이게 할 업데이트된 레코드의 타입을 나타냅니다.
UpdateStatus	*UpdateStatus*	레코드가 수정, 삽입, 삭제되었는지 또는 변경되지 않았는지를 나타냅니다.
OnUpdateError	*OnReconcileError*	레코드 단위로 업데이트 오류를 처리하기 위한 이벤트입니다.
OnUpdateRecord	*BeforeUpdateRecord*	레코드 단위로 업데이트를 처리하기 위한 이벤트입니다.
ApplyUpdates *ApplyUpdates* (데이터베이스)	*ApplyUpdates*	로컬 캐시의 레코드를 데이터베이스에 적용합니다.
CancelUpdates	*CancelUpdates*	로컬 캐시에서 보류 중인 모든 업데이트를 적용하지 않고 제거합니다.
CommitUpdates	*Reconcile*	성공적으로 업데이트를 적용한 후에 업데이트 캐시를 지웁니다.
FetchAll	*GetNextPacket* (및 *PacketRecords*)	데이터베이스 레코드를 편집 및 업데이트하기 위해 로컬 캐시에 복사합니다.
RevertRecord	*RevertRecord*	업데이트가 아직 적용되지 않았으면 현재 레코드에 대한 업데이트를 취소합니다.

캐싱된 업데이트 프로세스에 대한 개요는 27-16페이지의 "캐싱된 업데이트 사용의 개요"를 참조하십시오.

참고 클라이언트 데이터셋을 사용하여 업데이트를 캐싱하려는 경우에도 24-39페이지에 있는 업데이트 객체에 관한 단원을 읽어 보는 것이 좋습니다. *TBDEClientDataSet* 또는 *TDataSetProvider*의 *BeforeUpdateRecord* 이벤트 핸들러에 업데이트 객체를 사용하여 내장 프로시저나 다중 테이블 쿼리의 업데이트를 적용할 수 있습니다.

BDE 기반의 캐싱된 업데이트 사용

캐싱된 업데이트에 BDE 를 사용하려면 BDE 호환 데이터셋이 업데이트를 캐싱하도록 지정해야 합니다. 이렇게 지정하려면 *CachedUpdates* 속성을 **true**로 설정하십시오. 캐싱된 업데이트를 활성화하면 모든 레코드의 복사본이 로컬 메모리에 캐싱됩니다. 사용자가 이 로컬 데이터 복사본을 보거나 편집합니다. 변경, 삽입 및 삭제 내용도 메모리에 캐싱됩니다. 변경 내용은 애플리케이션이 이 내용을 데이터베이스 서버에 적용할 때까지 메모리에 축적됩니다. 변경된 레코드가 데이터베이스에 성공적으로 적용되면 해당 레코드는 캐시에서 해제됩니다.

데이터셋은 *CachedUpdates*를 **false**로 설정하기 전까지 모든 업데이트를 캐싱합니다. 캐싱된 업데이트 적용으로 그 이후의 캐싱된 업데이트가 비활성화되는 것은 아닙니다. 단지 현재의 변경 내용 집합을 데이터베이스에 쓰고 메모리에서 이 내용을 지울 뿐입니다. *CancelUpdates*를 호출하여 업데이트를 취소하면 현재 캐시에 있는 모든 변경 내용을 제거하지만 데이터셋이 이후의 변경 내용을 캐싱하지 못하도록 하지는 않습니다.

참고 *CachedUpdates*를 **false**로 설정하여 캐싱된 업데이트를 비활성화하면 아직 적용하지 않고 보류 중인 변경 사항은 경고 메시지 없이 삭제됩니다. 변경 내용을 잃지 않으려면 캐싱된 업데이트를 비활성화하기 전에 *UpdatesPending* 속성을 테스트하십시오.

BDE 기반의 캐싱된 업데이트 적용

업데이트 적용은 2단계 프로세스로 데이터베이스 컴포넌트 트랜잭션의 컨텍스트에서 수행되므로 애플리케이션에서 오류를 여유 있게 복구할 수 있습니다. 데이터베이스 컴포넌트에서 트랜잭션 처리에 대한 자세한 내용은 21-6페이지의 "트랜잭션 관리"를 참조하십시오.

데이터베이스 트랜잭션 제어 상태에서 업데이트를 적용하는 경우 다음과 같은 이벤트가 발생합니다.

1. 데이터베이스 트랜잭션이 시작합니다.
2. 캐싱된 업데이트를 데이터베이스에 씁니다(단계 1). 캐싱된 업데이트를 제공하면 데이터베이스에 레코드가 기록될 때마다 *OnUpdateRecord* 이벤트가 실행됩니다. 레코드가 데이터베이스에 적용될 때 오류가 발생하는 경우, 레코드를 제공하면 *OnUpdateError* 이벤트가 실행됩니다.
3. 쓰기를 성공하면 트랜잭션이 커밋되고, 그렇지 않으면 트랜잭션이 롤백됩니다.

   데이터베이스 쓰기를 성공한 경우,

   - 데이터베이스 변경 내용이 커밋되고 데이터베이스 트랜잭션이 끝납니다.
   - 캐싱된 업데이트가 커밋되고 내부 캐시 버퍼가 지워집니다(단계 2).

   데이터베이스 쓰기를 실패한 경우,

   - 데이터베이스 변경 내용이 롤백되고 데이터베이스 트랜잭션이 끝납니다.
   - 캐싱된 업데이트가 커밋되지 않고 내부 캐시에 비활성으로 남습니다.

OnUpdateRecord 이벤트 핸들러 생성 및 사용에 대한 자세한 내용은 24-35페이지의 "OnUpdateRecord 이벤트 핸들러 생성"을 참조하십시오. 캐싱된 업데이트를 적용할 때 발생하는 업데이트 오류 처리에 대한 자세한 내용은 24-37페이지의 "캐싱된 업데이트 오류 처리"를 참조하십시오.

참고 마스터/디테일 관계로 연결된 여러 데이터셋에서 작업하는 경우에는 각 데이터셋에 업데이트를 적용하는 순서가 중요하므로 캐싱된 업데이트를 적용하는 것이 특히 까다롭습니다. 일반적으로 디테일 테이블보다 마스터 테이블을 먼저 업데이트해야 합니다. 예외적으로 삭제된 레코드를 처리하는 경우에는 이 순서를 반대로 해야 합니다. 이러한 어려움 때문에 마스터/디테일 폼에서 업데이트를 캐싱하는 경우에는 클라이언트 데이터셋을 사용하는 것이 가장 좋습니다. 클라이언트 데이터셋은 마스터/디테일 관계와 관련된 모든 순서 문제를 자동으로 처리합니다.

다음과 같은 두 가지 방법으로 BDE 기반 업데이트를 적용할 수 있습니다.

- 데이터베이스 컴포넌트의 *ApplyUpdates* 메소드를 호출하여 업데이트를 적용할 수 있습니다. 이 방법을 사용하면 업데이트 프로세스를 위해 트랜잭션을 관리하고, 업데이트가 완료하는 경우 데이터셋 캐시를 지우는 것 등과 관련된 모든 상세한 부분을 데이터베이스가 처리하므로 가장 간단합니다.
- 데이터셋의 *ApplyUpdates* 및 *CommitUpdates* 메소드를 호출하여 단일 데이터셋에 업데이트를 적용할 수 있습니다. 데이터셋 수준에서 업데이트를 적용하는 경우 업데이트 프로세스를 랩핑하는 트랜잭션을 명시적으로 코딩해야 하고 *CommitUpdates*를 명시적으로 호출하여 캐시에 있는 업데이트를 커밋해야 합니다.

중요 라이브 결과 집합을 반환하지 않는 SQL 쿼리 또는 내장 프로시저에서 업데이트를 적용하려면 *TUpdateSQL*을 사용하여 업데이트를 수행하는 방법을 지정해야 합니다. 조인(두 개 이상의 테이블을 포함하는 쿼리)에 대한 업데이트인 경우 포함된 각 테이블에 대해 *TUpdateSQL* 객체를 하나씩 제공해야 하며 *OnUpdateRecord* 이벤트 핸들러에서 이 객체들을 호출하여 업데이트를 수행해야 합니다. 자세한 내용은 24-39페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

데이터베이스를 사용하여 캐싱된 업데이트 적용

데이터베이스 연결의 컨텍스트에 있는 하나 이상의 데이터셋에 캐싱된 업데이트를 적용하려면 데이터베이스 컴포넌트의 *ApplyUpdates* 메소드를 호출하십시오. 다음 코드는 버튼 클릭 이벤트에 응답하여 *CustomersQuery* 데이터셋에 업데이트를 적용합니다.

```
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
    Database1->TransIsolation = tiDirtyRead;
  Database1->ApplyUpdates(&CustomersQuery,0);
}
```

위의 시퀀스는 자동으로 생성된 트랜잭션의 컨텍스트에서 데이터베이스에 캐싱된 업데이트를 씁니다. 쓰기를 성공하면 트랜잭션을 커밋한 다음 캐싱된 업데이트를 커밋합니다. 쓰기를 성공하지 못하면 트랜잭션을 롤백하고 업데이트 캐시를 변경되지 않은 채로 둡니다. 후자의 경우에는 데이터셋의 *OnUpdateError* 이벤트를 통해 캐싱된 업데이트 오류를 처리해야 합니다. 업데이트 오류 처리에 대한 자세한 내용은 24-37페이지의 "캐싱된 업데이트 오류 처리"를 참조하십시오.

데이터베이스 컴포넌트의 *ApplyUpdates* 메소드를 사용하면 해당 데이터베이스에 연결된 데이터셋 컴포넌트를 얼마든지 업데이트할 수 있다는 기본적인 장점이 있습니다. 데이터베이스의 *ApplyUpdates* 메소드에 대한 인수는 *TDBDataSet* 배열과 이 배열에서 마지막 데이터셋의 인덱스 등 두 개입니다. 두 개 이상의 데이터셋에 업데이트를 적용하려면 데이터셋에 대한 포인터의 로컬 배열을 만드십시오. 예를 들어 다음 코드는 두 개의 쿼리에 대해 업데이트를 적용합니다.

```
TDBDataSet* ds[] = {CustomerQuery, OrdersQuery};
if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
  Database1->TransIsolation = tiDirtyRead;
Database1->ApplyUpdates(ds,1);
```

데이터셋 컴포넌트 메소드로 캐싱된 업데이트 적용

데이터셋의 *ApplyUpdates* 및 *CommitUpdates* 메소드를 사용하여 BDE 호환 개별 데이터셋에 대해 직접 업데이트를 적용할 수 있습니다. 이들 메소드는 각각 업데이트 프로세스의 단계를 각각 캡슐화합니다.

1. *ApplyUpdates*는 캐싱된 업데이트를 데이터베이스에 씁니다(단계 1).
2. *CommitUpdates*는 데이터베이스 쓰기가 성공하면 내부 캐시를 지웁니다(단계 2).

다음 코드는 *CustomerQuery* 데이터셋에 대해 트랜잭션의 업데이트를 적용하는 방법을 보여 줍니다.

```
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  Database1->StartTransaction();
  try
  {
    if (!Database1->IsSQLBased && Database1->TransIsolation !=
tiDirtyRead)
      Database1->TransIsolation = tiDirtyRead;
    CustomerQuery->ApplyUpdates(); // try to write the updates to the
database
    Database1->Commit(); // on success, commit the changes
  }
  catch (...)
  {
    Database1->Rollback(); // on failure, undo any changes
    throw; // throw the exception again to prevent a call to
CommitUpdates
  }
  CustomerQuery->CommitUpdates(); // on success, clear the internal cache
}
```

*ApplyUpdates*를 호출하는 동안 예외가 발생하면 데이터베이스 트랜잭션이 롤백됩니다. 트랜잭션을 롤백하면 원본으로 사용하는 데이터베이스 테이블이 변경되지 않도록 합니다. **try...catch** 블록에 있는 *throw* 문은 예외를 다시 *발생*시켜 *CommitUpdates*를 호출하지 않게 합니다. *CommitUpdates*가 호출되지 않아 업데이트의 내부 캐시가 지워지지 않으므로 오류 상태를 처리할 수 있고 업데이트를 다시 시도할 수도 있습니다.

OnUpdateRecord 이벤트 핸들러 생성

BDE 호환 데이터셋이 캐싱된 업데이트를 적용하는 경우 캐시에 기록된 변경 내용들 내에서 반복하여 그 내용을 기본 테이블의 해당 레코드에 적용하려고 합니다. 변경, 삭제 또는 새로 삽입된 각 레코드에 대한 업데이트를 적용하는 시점에 데이터셋 컴포넌트의 *OnUpdateRecord* 이벤트가 발생합니다.

OnUpdateRecord 이벤트의 핸들러를 제공하면 현재 레코드의 업데이트가 실제로 적용되기 바로 전에 작업을 수행할 수 있습니다. 이런 작업에는 특수 데이터 확인, 다른 테이블 업데이트, 특수 매개변수 대체 또는 여러 업데이트 객체 실행 등이 있습니다. *OnUpdateRecord* 이벤트 핸들러는 업데이트 프로세스를 더 잘 제어할 수 있게 합니다.

다음은 *OnUpdateRecord* 이벤트 핸들러를 위한 스켈레톤 코드입니다.

```
void __fastcall TForm1::DataSetUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // Perform updates here...
}
```

DataSet 매개변수는 업데이트가 캐싱된 데이터셋을 지정합니다.

UpdateKind 매개변수는 현재 레코드에 대해 수행되어야 하는 업데이트 타입을 나타냅니다. *UpdateKind*의 값은 *ukModify*, *ukInsert* 및 *ukDelete* 등입니다. 업데이트 객체를 사용하는 경우에는 업데이트를 적용할 때 이 매개변수를 업데이트 객체에 전달해야 합니다. 핸들러가 업데이트 타입에 따라 특별 처리를 수행하는 경우에도 이 매개변수를 검사해야 할 수 있습니다.

UpdateAction 매개변수는 업데이트를 적용했는지 여부를 나타냅니다. *UpdateAction*의 값은 *uaFail*(기본값), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied* 등입니다. 이벤트 핸들러가 업데이트를 성공적으로 적용하면 끝내기 전에 이 매개변수를 *uaApplied*로 변경합니다. 현재 레코드를 업데이트하지 않으려면 이 값을 *uaSkip*로 변경하여 적용되지 않은 변경 내용을 캐시에 보존합니다. *UpdateAction* 값을 변경하지 않으면 데이터셋에 대한 전체 업데이트 작업이 중지하고 예외가 발생합니다. *UpdateAction*을 *uaAbort*로 변경하면 오류 메시지가 표시되지 않게 할 수 있습니다(상태를 표시하지 않고 예외 발생).

이 매개변수들 이외에 일반적으로 현재 레코드에 연결된 필드 컴포넌트에 대해 *OldValue* 및 *NewValue* 속성을 사용할 수 있습니다 . *OldValue* 는 데이터베이스에서 가져온 원래 필드 값을 제공합니다. 이 값은 업데이트할 데이터베이스 레코드를 찾는 데 유용할 수 있습니다. *NewValue*는 적용하려는 업데이트에서 편집된 값입니다.

중요 *OnUpdateRecord* 이벤트 핸들러는 *OnUpdateError* 또는 *OnCalcFields* 이벤트 핸들러와 마찬가지로 데이터셋의 현재 레코드를 변경하는 메소드를 호출하면 안 됩니다.

다음 예제는 이 매개변수와 속성을 사용하는 방법을 보여 줍니다. 여기서는 *UpdateTable*이라는 *TTable* 컴포넌트를 사용하여 업데이트를 적용합니다. 실제로는 업데이트 객체를 사용하는 것이 더 편리하지만 가능성을 좀더 분명히 보여 주기 위해 테이블을 사용합니다.

```
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  if (UpdateKind == ukInsert)
  {
    TVarRec values[2];
    for (int i = 0; i < 2; i++)
      values[i] =DataSet->Fields->Fields[i]->NewValue;
    UpdateTable->AppendRecord(values, 1);
  }
  else
  {
    TLocateOptions lo;
    lo.Clear();
```

```
    if (UpdateTable->Locate("KeyField", DataSet->Fields->Fields[0]-
>OldValue, lo))
      switch (UpdateKind)
      {
        case ukModify:
          UpdateTable->Edit();
          UpdateTable->Fields->Fields[1]->Value = DataSet->Fields-
>Fields[1]->Value;
          UpdateTable->Post();
          break;
        case ukDelete:
          UpdateTable->Delete();
          break;
      }
  }
  UpdateAction = uaApplied;
}
```

캐싱된 업데이트 오류 처리

BDE(Borland Database Engine)는 업데이트를 적용하려 할 때 사용자의 업데이트 충돌 및 다른 상태를 특별히 검사하고 오류가 있으면 보고합니다. 데이터셋 컴포넌트의 *OnUpdateError* 이벤트를 사용하여 오류를 catch하고 이에 응답할 수 있습니다. 캐싱된 업데이트를 사용하는 경우 이 이벤트의 핸들러를 만들어야 합니다. 그렇지 않으면 오류가 발생하는 경우 전체 업데이트 작업이 실패합니다.

다음은 *OnUpdateError* 이벤트 핸들러를 위한 스켈레톤 코드입니다.

```
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // Respond to errors here...
}
```

*DataSet*은 업데이트가 적용될 데이터셋을 참조합니다. 오류를 처리하는 동안 이 데이터셋을 사용하여 새 값과 이전 값을 액세스할 수 있습니다. 각 레코드에 있는 필드의 원래 값은 *OldValue*라는 읽기 전용 *TField* 속성에 저장됩니다. 변경된 값은 *NewValue*라는 유사한 *TField* 속성에 저장됩니다. 이 값들을 사용해야만 이벤트 핸들러에서 업데이트 값을 검사하고 변경할 수 있습니다.

경고 현재 레코드를 변경하는 데이터셋 메소드(예: *Next* 및 *Prior*)를 호출하지 마십시오. 이렇게 하면 이벤트 핸들러가 무한 루프에 빠집니다.

E 매개변수는 일반적으로 *EDBEngineError* 타입입니다. 오류 핸들러에서 사용자에게 표시할 수 있는 오류 메시지를 이 예외 타입에서 추출할 수 있습니다. 예를 들어 다음 코드를 사용하여 다이얼로그 박스의 캡션에 오류 메시지를 표시할 수 있습니다.

```
ErrorLabel->Caption = E->Message;
```

이 매개변수는 업데이트 오류의 실제 원인을 결정하는 데도 유용합니다. *EDBEngineError*에서 특정 오류 코드를 추출하고, 이에 따라 적절한 작업을 수행할 수 있습니다.

UpdateKind 매개변수는 오류를 생성한 업데이트 타입을 설명합니다. 오류 핸들러가 수행되는 업데이트 타입에 따라 특수 작업을 수행하는 경우를 제외하고는 코드에서 이 매개변수를 사용하지 않습니다.

다음 표에서는 *UpdateKind*에 사용할 수 있는 값을 나열합니다.

표 24.7 UpdateKind 값

값	의미
ukModify	기존 레코드 편집으로 인한 오류 발생
ukInsert	새 레코드 삽입으로 인한 오류 발생
ukDelete	기존 레코드 삭제로 인한 오류 발생

*UpdateAction*은 BDE에게 이벤트 핸들러가 끝날 때 업데이트 프로세스를 계속하는 벙법에 대해 알려 줍니다. 업데이트 오류 핸들러를 처음으로 호출하면 이 값은 항상 *uaFail*로 설정됩니다. 오류를 일으킨 레코드에 대한 오류 상태와 오류 수정을 위해 수행한 작업에 따라 일반적으로 *UpdateAction*을 다른 값으로 설정하고 핸들러를 끝냅니다.

- 오류 핸들러가 자신을 호출하게 만든 오류 상태를 수정할 수 있는 경우에는 *UpdateAction*을 핸들러를 끝낼 때 수행할 적절한 작업으로 설정하십시오. 수정하는 오류 상태에 대해 *UpdateAction*을 *uaRetry*로 설정하여 해당 레코드에 대해 업데이트를 다시 적용하십시오.
- 이 값을 *uaSkip*으로 설정하면 오류를 일으킨 행에 대한 업데이트를 건너뛰고 다른 모든 업데이트가 완료된 후 해당 레코드의 업데이트는 캐시에 남습니다
- *uaFail* 및 *uaAbort* 모두 전체 업데이트 작업을 끝내지만, *uaFail*은 오류 메시지와 함께 예외를 발생시키는 반면 *uaAbort*는 오류 메시지 없이 예외를 발생시킵니다.

다음 코드는 업데이트 오류가 주요 위반과 관련되어 있는지 확인하고 그럴 경우 *UpdateAction* 매개변수를 *uaSkip*으로 설정하는 *OnUpdateError* 이벤트 핸들러를 보여 줍니다.

```
// include BDE.hpp in your unit file for this example
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  UpdateAction = uaFail // initialize to fail the update
  if *E,>ClassNameIs("EDBEngineError"))
  {
    EDBEngineError *pDBE = (EDBEngineError *E,
    if (pDBE->Errors[pDBE->ErrorCount - 1]->ErrorCode == DBIERR_KEYVIOL)
      UpdateAction = uaSkip; // Key violation, just skip this record
  }
}
```

참고 캐싱된 업데이트를 적용하는 동안 오류가 발생하면 예외가 *발생*되고 오류 메시지가 표시됩니다. **try...catch** 생성자에서 *ApplyUpdates*를 호출하지 않은 경우에는 *OnUpdateError* 이벤트 핸들러 내에서 사용자에게 표시되는 오류 메시지로 인해 애플리케이션이 같은 오류 메시지를 두 번 표시하게 됩니다. 오류 메시지가 반복되지 않게 하려면 *UpdateAction*을 *uaAbort*로 설정하여 시스템 생성 오류 메시지 표시를 해제하십시오.

업데이트 객체를 사용하여 데이터셋 업데이트

BDE 호환 데이터셋이 "라이브"가 아닌 쿼리나 내장 프로시저를 나타내는 경우 데이터셋에서 직접 업데이트를 적용할 수 없습니다. 이런 데이터셋은 클라이언트 데이터셋을 사용하여 업데이트를 캐싱하는 경우에도 문제를 일으킬 수 있습니다. 업데이트를 캐싱하기 위해 BDE를 사용하든 클라이언트를 사용하든 다음과 같이 업데이트 객체를 사용하여 이 문제의 데이터셋을 처리할 수 있습니다.

1. 클라이언트 데이터셋을 사용하는 경우 *TBDEClientDataSet* 대신 *TClientDataSet*과 함께 외부 프로바이더 컴포넌트를 사용하십시오. 이렇게 해야 BDE 호환 소스 데이터셋의 *UpdateObject* 속성을 설정할 수 있습니다(단계 3).
2. BDE 호환 데이터셋과 같은 데이터 모듈에 *TUpdateSQL* 컴포넌트를 추가합니다.
3. BDE 호환 데이터셋 컴포넌트의 *UpdateObject* 속성을 데이터 모듈에 있는 *TUpdateSQL* 컴포넌트로 설정합니다.
4. 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성을 사용하여 업데이트를 수행하는 데 필요한 SQL 문을 지정합니다. Update SQL Editor를 사용하여 이 명령문을 작성할 수 있습니다.
5. 데이터셋을 닫습니다.
6. 데이터셋 컴포넌트의 *CachedUpdates* 속성을 **true**로 설정하거나 데이터셋 프로바이더를 사용하여 데이터셋을 클라이언트 데이터셋에 연결합니다.
7. 데이터셋을 다시 엽니다.

참고 여러 업데이트 객체를 사용해야 하는 경우도 있습니다. 예를 들어 여러 데이터셋을 나타내는 내장 프로시저나 다중 테이블 조인을 업데이트하는 경우 업데이트할 테이블마다 *TUpdateSQL* 객체를 하나씩 제공해야 합니다. 여러 업데이트 객체를 사용하는 경우 간단히 데이터셋의 *UpdateObject* 속성을 설정하여 업데이트 객체를 데이터셋과 연결할 수 없습니다. 대신 *OnUpdateRecord* 이벤트 핸들러(BDE를 사용하여 업데이트를 캐싱하는 경우)나 *BeforeUpdateRecord* 이벤트 핸들러(클라이언트 데이터셋을 사용하는 경우)에서 업데이트 객체를 수동으로 호출해야 합니다.

업데이트 객체는 실제로 세 가지 *TQuery* 컴포넌트를 캡슐화합니다. 이 쿼리 컴포넌트 각각은 단일 업데이트 작업을 수행합니다. 첫째 쿼리 컴포넌트는 기존 레코드를 수정하기 위한 SQL UPDATE 문을, 둘째 쿼리 컴포넌트는 테이블에 새 레코드를 추가하기 위한 INSERT 문을, 셋째 컴포넌트는 테이블에서 레코드를 제거하기 위한 DELETE 문을 제공합니다.

업데이트 컴포넌트를 데이터 모듈에 넣어도 이 컴포넌트가 캡슐화하는 쿼리 컴포넌트는 표시되지 않습니다. 이 쿼리 컴포넌트는 SQL 문을 제공하는 세 가지 업데이트 속성을 기반으로 런타임에 업데이트 컴포넌트가 생성합니다.

- *ModifySQL*은 UPDATE 문을 지정합니다.
- *InsertSQL*은 INSERT 문을 지정합니다.
- *DeleteSQL*은 DELETE 문을 지정합니다.

런타임에 업데이트 컴포넌트를 사용하여 업데이트를 적용하면 업데이트 컴포넌트는 다음을 수행합니다.

1. 현재 레코드를 수정, 삽입 또는 삭제할지에 따라 실행할 SQL 문을 선택합니다.
2. SQL 문의 매개변수 값을 제공합니다.
3. SQL 문을 준비하고 실행하여 지정된 업데이트를 수행합니다.

업데이트 컴포넌트를 위한 SQL 문 작성

연결된 데이터셋의 레코드를 업데이트하기 위해 업데이트 객체는 세 가지 SQL 문 중 하나를 사용합니다. 각 업데이트 객체는 단일 테이블만 업데이트할 수 있으므로 해당 객체의 업데이트 문은 같은 기본 테이블을 참조해야 합니다.

세 가지 SQL 문은 업데이트를 위해 캐싱된 레코드를 삭제, 삽입 및 수정합니다. 이 명령문들을 업데이트 객체의 *DeleteSQL*, *InsertSQL* 및 *ModifySQL* 속성으로 제공해야 합니다. 이 값은 디자인 타임이나 런타임에 제공할 수 있습니다. 예를 들어 다음 코드는 런타임에 *DeleteSQL* 속성의 값을 지정합니다.

```
UpdateSQL->DeleteSQL->Clear();
UpdateSQL->DeleteSQL->Add("DELETE FROM Inventory I");
UpdateSQL->DeleteSQL->Add("WHERE (I.ItemNo = :OLD_ItemNo}");
```

디자인 타임에는 Update SQL Editor를 사용하여 업데이트를 적용하는 SQL 문을 작성할 수 있습니다.

업데이트 객체는 데이터셋의 원래 필드 및 업데이트된 필드 값을 참조하는 매개변수에 대해 자동 매개변수 연결을 제공합니다. 따라서 일반적으로 SQL 문을 작성하는 경우에는 특별히 서식화된 이름으로 매개변수를 삽입합니다. 이런 매개변수 사용에 대한 자세한 내용은 24-41 페이지의 "업데이트 SQL 문에서 매개변수 대체"를 참조하십시오.

Update SQL Editor 사용

다음과 같은 방법으로 업데이트 컴포넌트의 SQL 문을 작성할 수 있습니다.

1. Object Inspector를 사용하여 데이터셋의 *UpdateObject* 속성에 대한 드롭다운 리스트에서 업데이트 객체의 이름을 선택합니다. 이 단계를 통해 다음 단계에 호출하는 Update SQL Editor가 SQL 생성 옵션으로 사용할 적절한 기본값을 결정할 수 있게 됩니다.
2. 업데이트 객체를 마우스 오른쪽 버튼으로 클릭하고 컨텍스트 메뉴에서 UpdateSQL Editor를 선택합니다. 그러면 Update SQL Editor가 표시됩니다. 이 에디터를 사용하여 원본으로 사용하는 데이터셋 및 직접 제공하는 값을 기반으로 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성에 대한 SQL 문을 작성합니다.

Update SQL Editor에는 두 가지 페이지가 있습니다. 에디터를 처음으로 호출하면 Options 페이지가 표시됩니다. Table Name 콤보 박스를 사용하여 업데이트할 테이블을 선택합니다. 테이블 이름을 지정하면 Key Fields 및 Update Fields 리스트 박스가 사용 가능한 열로 채워집니다.

Update Fields 리스트 박스는 업데이트해야 할 열을 나타냅니다. 테이블을 처음으로 지정하면 Update Fields 리스트 박스의 모든 열이 포함되도록 선택됩니다. 원하는 만큼 여러 필드를 선택할 수 있습니다.

Key Fields 리스트 박스는 업데이트하는 동안 키로 사용할 열을 지정하는 데 사용됩니다. Paradox, dBASE 및 FoxPro인 경우 여기에 지정하는 열이 기존 인덱스와 일치해야 하지만 원격 SQL 데이터베이스인 경우에는 그럴 필요가 없습니다. Key Fields를 설정하지 않고 Primary Keys 버튼을 클릭하여 테이블의 기본 인덱스에 따라 업데이트를 위한 키 필드를 선택할 수도 있습니다. Dataset Defaults를 클릭하면 선택 리스트를 원래 상태(모든 필드를 키로 선택하고 모든 필드를 업데이트하도록 선택)로 되돌립니다.

필드 이름에 인용 부호를 붙여야 하는 서버인 경우 Quote Field Names 체크 박스를 선택하십시오.

테이블을 지정하고, 키 열 및 업데이트 열을 지정한 후 Generate SQL을 클릭하여 업데이트 객체의 *ModifySQL*, *InsertSQL* 및 *DeleteSQL* 속성에 연결할 예비 SQL 문을 생성합니다. 대부분의 경우 자동으로 생성된 SQL 문을 미세 조정해야 합니다.

생성된 SQL 문을 보거나 수정하려면 SQL 페이지를 선택하십시오. SQL 문을 생성한 다음에 이 페이지를 선택한 경우에는 *ModifySQL* 속성에 대한 SQL 문이 SQL Text 메모 박스에 미리 표시되어 있습니다. 박스에 있는 명령문을 필요에 따라 편집할 수 있습니다.

중요 생성된 SQL 문은 업데이트 문을 작성하기 위한 기준일 뿐이므로 올바로 실행되도록 이 명령문들을 수정해야 합니다. 예를 들어 NULL 값을 포함하는 데이터 작업을 하는 경우에는 생성된 필드 변수를 사용하지 않고 다음과 같이 WHERE 절을 수정해야 합니다.

```
WHERE field IS NULL
```

각 명령문을 적용하기 전에 직접 테스트하십시오.

Statement Type 라디오 버튼을 사용하여 생성된 SQL 문 간을 전환하고 원하는 대로 편집하십시오.

문을 적용하고 업데이트 컴포넌트의 SQL 속성에 연결하려면 OK를 클릭하십시오.

업데이트 SQL 문에서 매개변수 대체

Update SQL 문은 레코드 업데이트에 있는 이전 또는 새 필드 값을 대체할 수 있게 하는 특수한 형태의 매개변수 대체를 사용합니다. Update SQL Editor를 사용하여 SQL 문을 생성하는 경우 사용할 필드 값을 Update SQL Editor가 결정합니다. 업데이트 SQL를 직접 작성하는 경우에는 사용할 필드 값을 지정해야 합니다.

매개변수 이름이 테이블의 열 이름과 일치하면 해당 레코드의 캐싱된 업데이트에 있는 새 값이 자동으로 매개변수 값으로 사용됩니다. 매개변수 이름이 열 이름 앞에 "OLD_"라는 문자열이 붙은 이름과 일치하면 필드의 이전 값이 사용됩니다. 예를 들어 아래의 업데이트 SQL 문에서 매개변수 :LastName은 삽입된 레코드의 캐싱된 업데이트에 있는 새 필드 값으로 자동으로 채워집니다.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

새 필드 값은 일반적으로 *InsertSQL* 및 *ModifySQL* 문에서 사용됩니다. 수정된 레코드의 업데이트에서 UPDATE 문은 업데이트될 기본 테이블에 있는 이전 필드 값을 업데이트 캐시에 있는 새 필드 값으로 바꿉니다.

삭제된 레코드인 경우 새 값이 없으므로 *DeleteSQL* 속성은 ":OLD_FieldName" 구문을 사용합니다. 이전 필드 값은 업데이트 또는 삭제할 레코드를 결정하기 위해 수정 또는 삭제된 업데이트를 위한 SQL 문의 WHERE 절에서도 사용됩니다.

UPDATE 또는 DELETE 업데이트 SQL 문의 WHERE 절에는 캐싱된 업데이트로 업데이트할 기본 테이블의 레코드를 고유하게 식별하기 위한 최소 수의 매개변수를 제공합니다. 예를 들어 고객 리스트에서 고객의 성만을 사용하면 기본 테이블의 레코드를 고유하게 식별하기에 충분하지 않을 수 있습니다. 성이 "Smith"인 레코드가 많이 있을 수 있기 때문입니다. 그러나 성, 이름 및 전화 번호에 대해 매개변수를 사용하면 고유한 조합이 될 수 있습니다. 고객 번호와 같은 고유한 필드 값이면 훨씬 좋습니다.

참고 편집된 필드 값이나 원래 필드 값을 참조하지 않는 매개변수를 포함하는 SQL 문을 작성하면 업데이트 객체가 이 값들을 연결하지 못합니다. 그러나 업데이트 객체의 *Query* 속성을 사용하면 이 값들을 수동으로 연결할 수 있습니다. 자세한 내용은 24-46페이지의 "컴포넌트의 Query 속성 사용"을 참조하십시오.

업데이트 SQL 문 작성

디자인 타임에 Update SQL Editor를 사용하여 *DeleteSQL*, *InsertSQL* 및 *ModifySQL* 속성을 위한 SQL 문을 작성할 수 있습니다. Update SQL Editor를 사용하지 않거나 생성된 명령문을 수정하려는 경우, 기본 테이블의 레코드를 삭제, 삽입 및 수정하기 위한 SQL 문을 작성할 때 다음 지침을 기억하십시오.

DeleteSQL 속성은 DELETE 명령이 있는 SQL 문만을 포함해야 합니다. 업데이트될 기본 테이블은 FROM 절에 이름을 지정해야 합니다. SQL 문이 기본 테이블의 레코드 중 업데이트 캐시에서 삭제된 레코드에 해당하는 레코드만 삭제하도록 WHERE 절을 사용합니다. WHERE 절에는 캐싱된 업데이트 레코드에 해당하는 기본 테이블 레코드를 고유하게 식별하기 위해 하나 이상의 필드에 대해 매개변수를 사용합니다. 매개변수가 필드와 같은 이름 앞에 "OLD_"라는 접두사를 붙인 이름으로 지정된 경우 매개변수에는 캐싱된 업데이트 레코드의 해당 필드 값이 자동으로 지정됩니다. 매개변수 이름이 다른 방식으로 지정된 경우에는 매개변수 값을 직접 제공해야 합니다.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo}
```

일부 테이블 타입에서는 NULL 값을 포함하는 레코드를 식별하기 위해 필드를 사용하는 경우 기본 테이블의 레코드를 찾지 못할 수도 있습니다. 이런 경우 삭제 업데이트가 해당 레코드에 대해 실패합니다. 이런 조건에 맞추려면 NULL 값이 아닌 필드를 위한 조건과 더불어 NULL을 포함할 수 있는 필드를 위한 조건을 IS NULL 술부를 사용하여 추가하십시오. 예를 들어 FirstName 필드가 NULL 값을 포함할 수 있는 경우에 다음과 같이 합니다.

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

InsertSQL 문은 INSERT 명령이 있는 SQL 문만을 포함해야 합니다. 업데이트될 기본 테이블은 INTO 절에 이름을 지정해야 합니다. VALUES 절에는 쉼표로 구분된 매개변수 리스트를 제공합니다. 매개변수가 필드와 같은 이름으로 지정된 경우 매개변수에는 캐싱된 업데이트 레코드의 값이 자동으로 지정됩니다. 매개변수 이름이 다른 방식으로 지정된 경우에는 매개변수 값을 직접 제공해야 합니다. 매개변수 리스트가 새로 삽입된 레코드의 필드 값을 제공합니다. 명령문에 나열된 필드 수 만큼의 매개변수 값이 있어야 합니다.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

ModifySQL 문은 UPDATE 명령이 있는 SQL 문만을 포함해야 합니다. 업데이트될 기본 테이블은 FROM 절에 이름을 지정해야 합니다. SET 절에 하나 이상의 값 할당문을 포함시키십시오. SET 절 할당문의 값이 필드와 같은 이름으로 지정된 매개변수인 경우 이 매개변수에는 캐시에 있는 업데이트 레코드의 필드 중 이름이 같은 필드의 값이 자동으로 지정됩니다. 매개변수를 필드와 같은 이름으로 지정하지 않는 한 다른 매개변수를 사용하여 추가 필드 값을 할당할 수 있고 그 값을 수동으로 제공할 수 있습니다. *DeleteSQL* 문과 마찬가지로 WHERE 절에 필드와 같은 이름 앞에 "OLD_"를 붙인 이름으로 지정한 매개변수를 사용하여 업데이트될 기본 테이블의 레코드를 고유하게 식별합니다. 아래 업데이트 문에서 매개변수 :ItemNo에는 자동으로 값이 지정되고 :Price에는 그렇지 않습니다.

```
UPDATE Inventory I
SET (I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo}
```

위의 업데이트 SQL에 대해 애플리케이션 엔드 유저가 기존 레코드를 수정하는 경우를 예로 들어 봅니다. ItemNo 필드의 원래 값은 999입니다. 캐싱된 데이터셋에 연결된 그리드에서 엔드 유저가 ItemNo 필드 값은 123으로, Amount는 20으로 변경합니다. ApplyUpdates 메소드를 호출하면 이 SQL 문은 매개변수 :OLD_ItemNo의 이전 필드 값을 사용하여 기본 테이블의 레코드 중 ItemNo 필드가 999인 모든 레코드에 영향을 미칩니다. 그리드에서 제공되는 값인 매개변수 :ItemNo를 사용하여 이 레코드의 ItemNo 필드 값을 123으로 변경하고 Amount는 20으로 변경합니다.

여러 업데이트 객체 사용

업데이트 데이터셋이 참조하는 두 개 이상의 기본 테이블을 업데이트해야 하는 경우 여러 업데이트 객체를 사용해야 합니다. 업데이트되는 각 기본 테이블에 대해 업데이트 객체가 하나씩 필요합니다. 데이터셋 컴포넌트의 *UpdateObject*에서는 데이터셋에 하나의 업데이트 객체만 연결할 수 있으므로 업데이트 객체의 *DataSet* 속성을 데이터셋의 이름으로 설정하여 각 업데이트 객체를 해당 데이터셋에 연결해야 합니다.

팁 여러 업데이트 객체를 사용하는 경우 외부 프로바이더와 함께 *TClientDataSet* 대신 *TBDEClientDataSet*을 사용할 수 있습니다. 이는 소스 데이터셋의 *UpdateObject* 속성을 설정할 필요가 없기 때문입니다.

업데이트 객체의 *DataSet* 속성은 디자인 타임에 Object Inspector에서 사용할 수 없습니다. 런타임에만 이 속성을 설정할 수 있습니다.

```
UpdateSQL1->DataSet = Query1;
```

업데이트 객체는 이 데이터셋을 사용하여 매개변수 대체를 위해 원래 필드 및 업데이터된 필드 값을 가져오며, 이 데이터셋이 BDE 호환인 경우에는 업데이트를 적용할 때 사용할 세션 및 데이터베이스를 식별합니다. 매개변수 대체가 올바로 작동하기 위해서는 업데이트 객체의 *DataSet* 속성이 업데이트된 필드 값을 포함하는 데이터셋이 되어야 합니다. BDE 호환 데이터셋을 사용하여 업데이트를 캐싱하는 경우 이 데이터셋은 BDE 호환 데이터셋 자체입니다. 클라이언트 데이터셋을 사용하는 경우 이 데이터셋은 *BeforeUpdateRecord* 이벤트 핸들러에 매개변수로 제공되는 클라이언트 데이터셋입니다.

업데이트 객체를 데이터셋의 *UpdateObject* 속성에 할당하지 않은 경우에는 *ApplyUpdates*를 호출하더라도 업데이트 객체의 SQL 문이 자동으로 실행되지 않습니다. 레코드를 업데이트하려면 *OnUpdateRecord* 이벤트 핸들러(BDE를 사용하여 업데이트를 캐싱하는 경우) 또는 *BeforeUpdateRecord* 이벤트 핸들러(클라이언트 데이터셋을 사용하는 경우)에서 업데이트 객체를 수동으로 호출해야 합니다. 이벤트 핸들러에서 적어도 다음과 같은 작업을 수행해야 합니다.

- 클라이언트 데이터셋을 사용하여 업데이트를 캐싱하는 경우 업데이트 객체의 *DatabaseName* 및 *SessionName* 속성이 소스 데이터셋의 *DatabaseName* 및 *SessionName* 속성으로 설정되어 있는지 확인해야 합니다.
- 이벤트 핸들러가 업데이트 객체의 *ExecSQL* 또는 *Apply* 메소드를 호출해야 합니다. 이 이벤트 핸들러는 업데이트를 해야 하는 레코드마다 업데이트 객체를 호출합니다. 업데이트 문 실행에 대한 자세한 내용은 아래의 "SQL 문 실행"을 참조하십시오.
- 이벤트 핸들러의 *UpdateAction* 매개변수를 *uaApplied*로 설정하거나(*OnUpdateRecord*) *Applied* 매개변수를 **true**로 설정하십시오(*BeforeUpdateRecord*).

각 레코드 업데이트에 따라 데이터 확인, 데이터 수정 또는 기타 작업 등을 수행할 수도 있습니다.

경고 *OnUpdateRecord* 이벤트 핸들러에서 업데이트 객체의 *ExecSQL* 또는 *Apply* 메소드를 호출하는 경우 데이터셋의 *UpdateObject* 속성을 해당 업데이트 객체로 설정하지 마십시오. 그렇게 설정하면 각 레코드의 업데이트 적용을 두 번 시도하게 됩니다.

SQL 문 실행

여러 업데이트 객체를 사용하는 경우 데이터셋의 *UpdateObject* 속성을 설정하여 업데이트 객체를 데이터셋에 연결하지 않습니다. 결과적으로 업데이트를 적용할 때 해당 명령문이 자동으로 실행되지 않습니다. 대신 코드에서 업데이트 객체를 명시적으로 호출해야 합니다.

다음과 같은 두 가지 방법으로 업데이트 객체를 호출할 수 있습니다. 어떤 방법을 선택하는지는 SQL 문이 필드 값을 나타내는 매개변수를 사용하는지 여부에 따라 결정됩니다.

- 실행할 SQL 문이 매개변수를 사용하는 경우 *Apply* 메소드를 호출하십시오.
- 실행할 SQL 문이 매개변수를 사용하지 않는 경우 *ExecSQL* 메소드를 호출하는 것이 더 효율적입니다.

참고 SQL 문이 이전 필드 및 업데이트 필드 값에 대해 내장 타입 이외의 매개변수를 사용하는 경우 *Apply* 메소드에서 제공되는 매개변수 대체를 사용하는 대신 매개변수 값을 수동으로 제공해야 합니다. 매개변수 값을 수동으로 제공하는 것에 대한 자세한 내용은 24-46페이지의 "컴포넌트의 Query 속성 사용"을 참조하십시오.

업데이트 객체의 SQL 문의 매개변수를 위한 디폴트 매개변수 대체에 대한 자세한 내용은 24-41페이지의 "업데이트 SQL 문에서 매개변수 대체"를 참조하십시오.

Apply 메소드 호출

업데이트 컴포넌트의 *Apply* 메소드는 현재 레코드에 대해 업데이트를 수동으로 적용합니다. 이 프로세스는 다음과 같이 2단계로 구성됩니다.

1. 레코드의 초기 필드 값과 편집된 필드 값을 해당 SQL 문의 매개변수에 연결합니다.
2. SQL 문이 실행됩니다.

Apply 메소드를 호출하여 업데이트 캐시에 있는 현재 레코드의 업데이트를 적용합니다. *Apply* 메소드는 주로 데이터셋의 *OnUpdateRecord* 이벤트 핸들러나 프로바이더의 *BeforeUpdateRecord* 이벤트 핸들러에서 호출됩니다.

경고 데이터셋의 *UpdateObject* 속성을 사용하여 데이터셋과 업데이트 객체를 연결하는 경우에는 *Apply*가 자동으로 호출됩니다. 이런 경우에는 *OnUpdateRecord* 이벤트 핸들러에서 *Apply*를 호출하지 마십시오. 그러면 현재 레코드의 업데이트 적용을 두 번 시도하게 됩니다.

OnUpdateRecord 이벤트 핸들러에는 *TUpdateKind* 타입의 *UpdateKind* 매개변수를 사용하여 적용해야 하는 업데이트 타입을 지정합니다. 이 매개변수를 *Apply* 메소드에 전달하여 어떤 SQL 문을 사용할지 지정해야 합니다. 다음 코드는 *BeforeUpdateRecord* 이벤트 핸들러를 사용하여 이것을 보여 줍니다.

```
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject
*Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind
UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS;
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->Apply(UpdateKind);
  Applied = true;
}
```

ExecSQL 메소드 호출

업데이트 컴포넌트의 *ExecSQL* 메소드는 현재 레코드에 대해 업데이트를 수동으로 적용합니다. *Apply* 메소드와 달리 *ExecSQL*은 실행하기 전에 SQL 문의 매개변수를 연결하지 않습니다. *ExecSQL* 메소드는 주로 *OnUpdateRecord* 이벤트 핸들러(BDE를 사용하는 경우)나 *BeforeUpdateRecord* 이벤트 핸들러(클라이언트 데이터셋을 사용하는 경우)에서 호출됩니다.

*ExecSQL*은 매개변수 값을 연결하지 않으므로 기본적으로 업데이트 객체의 SQL 문에 매개변수가 없을 경우에 사용됩니다. 매개변수가 없는 경우에도 *Apply* 메소드를 사용할 수 있지만 매개변수를 확인하지 않는 *ExecSQL*이 더 효율적입니다.

SQL 문이 매개변수를 포함하는 경우에도 *ExecSQL*를 호출할 수 있지만 우선 매개변수를 명시적으로 연결해야 합니다. BDE를 사용하여 업데이트를 캐싱하는 경우 업데이트 객체의 *DataSet* 속성을 설정한 다음 *SetParams* 메소드를 호출하여 매개변수를 명시적으로 연결할 수 있습니다. 클라이언트 데이터셋을 사용하여 업데이트를 캐싱하는 경우에는 *TUpdateSQL*이 유지 보수하는, 원본으로 사용하는 쿼리 객체에 매개변수를 제공해야 합니다. 이 방법에 대한 자세한 내용은 24-46페이지의 "컴포넌트의 Query 속성 사용"을 참조하십시오.

경고 데이터셋의 *UpdateObject* 속성을 사용하여 데이터셋과 업데이트 객체를 연결하는 경우에는 *ExecSQL*이 자동으로 호출됩니다. 이런 경우에는 *OnUpdateRecord* 또는 *BeforeUpdateRecord* 이벤트 핸들러에서 *ExecSQL*을 호출하지 마십시오. 그러면 현재 레코드의 업데이트 적용을 두 번 시도하게 됩니다.

OnUpdateRecord 및 *BeforeUpdateRecord* 이벤트 핸들러에는 *TUpdateKind* 타입의 *UpdateKind* 매개변수를 사용하여 적용해야 하는 업데이트 타입을 지정합니다. 이 매개변수를 *ExecSQL* 메소드에 전달하여 어떤 SQL 문을 사용할지 지정해야 합니다. 다음 코드는 *BeforeUpdateRecord* 이벤트 핸들러를 사용하여 이것을 보여 줍니다.

```
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject
*Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind
UpdateKind, bool &Applied)
{
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->ExecSQL(UpdateKind);
  Applied = true;
}
```

업데이트 프로그램을 실행하는 동안 예외가 발생하면 *OnUpdateError* 이벤트가 정의된 경우 이 이벤트가 실행됩니다.

컴포넌트의 Query 속성 사용

업데이트 컴포넌트의 *Query* 속성을 사용하여 해당 *DeleteSQL, InsertSQL* 및 *ModifySQL* 문을 구현하는 쿼리 컴포넌트를 액세스할 수 있습니다. 대부분의 애플리케이션에서 이 쿼리 컴포넌트를 직접 액세스할 필요가 없습니다. *DeleteSQL, InsertSQL* 및 *ModifySQL* 속성을 사용하여 이 쿼리가 실행하는 SQL 문을 지정할 수 있고 업데이트 객체의 *Apply* 또는 *ExecSQL* 메소드를 호출하여 SQL 문을 실행할 수 있습니다. 그러나 쿼리 컴포넌트를 직접 조작해야 하는 경우도 있을 수 있습니다. 특히 업데이트 객체가 매개변수를 이전 또는 새 필드 값으로 자동으로 연결하게 하지 않고 SQL 문의 매개변수 값을 직접 제공하려는 경우 *Query* 속성이 유용합니다.

참고 *Query* 속성은 런타임에만 액세스할 수 있습니다.

Query 속성은 *TUpdateKind* 값으로 인덱싱되어 있습니다.

- *ukModify* 인덱스를 사용하면 기본 레코드를 업데이트하는 쿼리를 액세스합니다.
- *ukInsert* 인덱스를 사용하면 새 레코드를 삽입하는 쿼리를 액세스합니다.
- *ukDelete* 인덱스를 사용하면 레코드를 삭제하는 쿼리를 액세스합니다.

다음 코드는 *Query* 속성을 사용하여 자동으로 연결할 수 없는 매개변수 값을 제공하는 방법을 보여 줍니다.

```
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject
*Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind
UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS; // required for the automatic parameter
substitution
  TQuery *pQuery = UpdateSQL1->Query[UpdateKind]; // access the query
  // make sure the query has the correct DatabaseName and SessionName
```

```
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  pQuery->DatabaseName = pSrcDS->DatabaseName;
  pQuery->SessionName = pSrcDS->SessionName;
  // now substitute values for custom parameters
  pQuery->ParamByName("TimeOfLastUpdate")->Value = Now();
  UpdateSQL1->Apply(UpdateKind); // now do automatic substitution and
execute
  Applied = true;
}
```

TBatchMove 사용

*TBatchMove*는 데이터셋 복제, 한 데이터셋의 레코드를 다른 데이터셋에 추가, 한 데이터셋의 레코드를 다른 데이터셋의 레코드로 업데이트, 한 데이터셋의 레코드 중 다른 데이터셋의 레코드와 일치하는 레코드 삭제 등을 할 수 있게 하는 BDE(Borland Database Engine) 기능을 캡슐화합니다. *TBatchMove*는 다음을 하는 데 주로 사용됩니다.

- 분석이나 기타 작업을 위해 서버의 데이터를 로컬 데이터 소스로 다운로드
- 업사이징 작업의 일환으로 데이크탑 데이터베이스를 원격 서버의 테이블로 이동

일괄 이동 컴포넌트는 자동으로 열 이름과 데이터 타입을 적절하게 매핑하여 소스 테이블과 일치하는 테이블을 대상에 만들 수 있습니다.

일괄 이동 컴포넌트 생성

다음과 같은 방법으로 일괄 이동 컴포넌트를 만듭니다.

1. 레코드를 임포트할 데이터셋(*Source* 데이터셋이라고 함)의 테이블 또는 쿼리 컴포넌트를 폼이나 데이터 모듈에 배치합니다.
2. 레코드를 이동시킬 데이터셋(*Destination* 데이터셋이라고 함)을 폼이나 데이터 모듈에 배치합니다.
3. Component 팔레트의 BDE 페이지에서 *TBatchMove* 컴포넌트를 데이터 모듈이나 폼에 배치하고 그 *Name* 속성을 애플리케이션에 맞는 고유한 값으로 설정합니다.
4. 일괄 이동 컴포넌트의 *Source* 속성을 레코드를 복사, 추가 또는 업데이트할 테이블 이름으로 설정합니다. 사용할 수 있는 데이터셋 컴포넌트에 대한 드롭다운 리스트에서 테이블을 선택할 수 있습니다.
5. *Destination* 속성을 생성, 추가 또는 업데이트할 데이터셋으로 설정합니다. 사용할 수 있는 데이터셋 컴포넌트에 대한 드롭다운 리스트에서 대상 테이블을 선택할 수 있습니다.
   - 추가, 업데이트 또는 삭제하는 경우 *Destination*은 기존 테이블을 나타내야 합니다.
   - 테이블을 복사하는데 *Destination*이 기존 테이블을 나타내는 경우 일괄 이동을 실행하면 대상 테이블의 현재 데이터를 모두 덮어씁니다.
   - 기존 테이블을 복사하여 완전히 새 테이블을 생성하는 경우 결과 테이블은 복사하려는 테이블 컴포넌트의 *Name* 속성에 지정된 이름을 가집니다. 결과 테이블 타입은 *DatabaseName* 속성에 지정된 서버에 적당한 구조가 됩니다.

6 *Mode* 속성을 설정하여 수행할 작업의 유형을 나타냅니다. 유효한 작업은 *batAppend*(기본값), *batUpdate*, *batAppendUpdate*, *batCopy* 및 *batDelete* 등입니다. 이 모드에 대한 자세한 내용은 24-48페이지의 "일괄 이동 모드 지정"을 참조하십시오.

7 *Transliterate* 속성을 설정합니다(옵션). *Transliterate*가 **true**(기본값)이면 필요한 경우 문자 데이터가 *Source* 데이터셋의 문자 집합에서 *Destination* 데이터셋의 문자 집합으로 변환됩니다.

8 *Mappings* 속성을 사용하여 열 매핑을 설정합니다(옵션). 일괄 이동에서 소스 및 대상 테이블의 열 위치에 따라 열을 비교하게 하려면 이 속성을 설정할 필요가 없습니다. 열 매핑에 대한 자세한 내용은 24-49페이지의 "데이터 타입 매핑"을 참조하십시오.

9 *ChangedTableName*, *KeyViolTableName* 및 *ProblemTableName* 속성을 설정합니다(옵션). 일괄 이동은 일괄 작업 동안에 문제가 발생한 레코드를 *ProblemTableName*에 지정된 테이블에 저장합니다. 일괄 이동을 통해 Paradox 테이블을 업데이트하는 경우 *KeyViolTableName*에 지정한 테이블에 키 위반을 기록하게 할 수 있습니다. *ChangedTableName*은 일괄 이동 작업의 결과로 대상 테이블에서 변경된 모든 레코드를 나열합니다. 이 속성을 지정하지 않으면 이 오류 테이블은 생성되거나 사용되지 않습니다. 일괄 이동 오류 처리에 대한 자세한 내용은 24-50페이지의 "일괄 이동 오류 처리"를 참조하십시오.

일괄 이동 모드 지정

Mode 속성은 일괄 이동 컴포넌트가 수행하는 작업을 지정합니다.

표 24.8 일괄 이동 모드

속성	용도
batAppend	레코드를 대상 테이블에 추가합니다.
batUpdate	대상 테이블의 레코드를 소스 테이블에서 일치하는 테이블로 업데이트합니다. 대상 테이블의 현재 인덱스를 기준으로 업데이트합니다.
batAppendUpdate	일치하는 레코드가 대상 테이블에 있으면 이 레코드를 업데이트하고, 그렇지 않으면 대상 테이블에 레코드를 추가합니다.
batCopy	소스 테이블 구조를 기반으로 대상 테이블을 생성합니다. 이미 대상 테이블이 있는 경우에는 이 테이블을 삭제하고 다시 생성합니다.
batDelete	대상 테이블의 레코드 중 소스 테이블의 레코드와 일치하는 레코드를 삭제합니다.

레코드 추가

데이터를 추가하려면 대상 데이터셋은 기존 테이블을 나타내야 합니다. 추가 작업 동안 BDE는 필요한 경우 데이터를 대상 데이터셋에 적당한 데이터 타입 및 크기로 변환합니다. 변환할 수 없는 경우에는 예외가 발생되고 데이터가 추가되지 않습니다.

레코드 업데이트

데이터를 업데이트하려면 대상 데이터셋은 기존 테이블을 나타내야 하며, 레코드를 비교할 수 있는 인덱스가 정의되어 있어야 합니다. 기본 인덱스 필드를 사용하여 레코드를 비교하는 경우, 소스 데이터의 인덱스 필드 레코드와 일치하는 인덱스 필드를 가진 대상 데이터셋의 레코드는 소스 데이터가 덮어씁니다. 업데이트 작업 동안 BDE는 필요한 경우 데이터를 대상 데이터셋에 적당한 데이터 타입 및 크기로 변환합니다.

레코드 추가 및 업데이트

데이터를 추가하고 업데이트하려면 대상 데이터셋은 기존 테이블을 나타내야 하며, 레코드를 비교할 수 있는 인덱스가 정의되어 있어야 합니다. 기본 인덱스 필드를 사용하여 레코드를 비교하는 경우, 소스 데이터의 인덱스 필드 레코드와 일치하는 인덱스 필드를 가진 대상 데이터셋의 레코드는 소스 데이터가 덮어씁니다. 일치하는 레코드가 없으면 소스 데이터셋의 데이터가 대상 데이터셋에 추가됩니다. 추가 및 업데이트 작업 동안 BDE는 필요한 경우 데이터를 대상 데이터셋에 적당한 데이터 타입 및 크기로 변환합니다.

데이터셋 복사

소스 데이터셋을 복사하려면 대상 데이터셋이 기존 테이블을 나타내서는 안됩니다. 기존 테이블을 나타낼 경우 일괄 이동 작업은 기존 테이블을 소스 데이터셋으로 덮어씁니다.

소스 및 대상 데이터셋을 다른 타입의 데이터베이스 엔진(예: Paradox 및 InterBase)에서 유지보수하는 경우, BDE는 가능한 한 소스 데이터셋의 구조와 유사하게 대상 데이터셋을 생성하고 필요에 따라 데이터 타입 및 크기 변환을 자동으로 수행합니다.

참고 *TBatchMove*는 인덱스, 제약 조건, 내장 프로시저 등과 같은 메타데이터를 복사하지 않습니다. 경우에 따라 SQL 탐색기를 통해서나 서버에서 이런 메타데이터 객체를 다시 생성해야 합니다.

레코드 삭제

대상 데이터셋의 데이터를 삭제하려면 대상 데이터셋은 기존 테이블을 나타내야 하며 , 레코드를 비교할 수 있는 인덱스가 정의되어 있어야 합니다. 기본 인덱스 필드를 사용하여 레코드를 비교하는 경우 , 소스 데이터의 인덱스 필드 레코드와 일치하는 인덱스 필드를 가진 대상 데이터셋의 레코드가 대상 테이블에서 삭제됩니다.

데이터 타입 매핑

batAppend 모드에서 일괄 이동 컴포넌트는 소스 테이블의 열 데이터 타입에 따라 대상 테이블을 생성합니다. 열과 타입은 소스 및 대상 테이블에서의 각각의 위치에 따라 비교됩니다. 즉, 소스 테이블의 첫번 째 열은 대상 테이블의 첫번 째 열과 비교됩니다.

디폴트 열 매핑을 오버라이드하려면 *Mappings* 속성을 사용하십시오. *Mappings*은 열 매핑을 한 줄 당 하나씩 나열합니다. 이 리스트는 두 가지 폼 중 하나를 취할 수 있습니다. 소스 테이블의 열을 대상 테이블에서 같은 이름의 열로 매핑하려면 일치시킬 열 이름을 지정하는 간단한 리스트를 사용할 수 있습니다. 예를 들어 다음 매핑은 소스 테이블에서 *ColName*이라는 열을 대상 테이블에서 같은 이름의 열로 매핑하도록 지정합니다.

```
ColName
```

소스 테이블에서 *SourceColName*이라는 열을 대상 테이블에서 *DestColName*이라는 열로 매핑하려면 다음 구문을 사용합니다.

```
DestColName = SourceColName
```

소스 및 대상 열 데이터 타입이 같지 않으면 일괄 이동 작업은 "자동 맞춤"을 시도합니다. 필요한 경우 문자 데이터 타입을 잘라내고 가능하면 제한된 양의 변환을 수행하려 합니다. 예를 들어 CHAR(10) 열을 CHAR(5) 열로 매핑하면 소스 열의 마지막 다섯 문자를 잘라내게 됩니다.

변환하는 예를 들면 문자 데이터 타입의 소스 열을 정수 타입의 대상 열로 매핑하는 경우 일괄 이동 작업은 "5"라는 문자 값을 해당하는 정수 값으로 매핑합니다. 변환될 수 없는 값은 오류를 생성합니다. 오류에 대한 자세한 내용은 24-50페이지의 "일괄 이동 오류 처리"를 참조하십시오.

다른 테이블 타입 간에 데이터를 이동시키는 경우 일괄 이동 컴포넌트는 데이터셋의 서버 타입을 기준으로 적절하게 데이터를 변환합니다. 서버 타입 간의 최근 매핑 테이블을 보려면 BDE 온라인 도움말 파일을 참조하십시오.

참고 데이터를 SQL 서버 데이터베이스로 일괄 이동시키려면 해당 데이터베이스 서버와 적절한 SQL Link가 설치된 C++Builder 버전이 있어야 합니다. 적절한 서드파티 ODBC 드라이버가 설치되어 있는 경우에는 ODBC를 사용할 수도 있습니다.

일괄 이동 실행

Execute 메소드를 사용하여 런타임에 미리 준비한 일괄 작업을 실행할 수 있습니다. 예를 들어 *BatchMoveAdd*가 일괄 이동 컴포넌트 이름인 경우 다음 명령문으로 실행합니다.

```
BatchMoveAdd->Execute();
```

디자인 타임에 일괄 이동 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 컨텍스트 메뉴에서 Execute를 선택하여 일괄 이동을 실행할 수도 있습니다.

MovedCount 속성은 일괄 이동을 실행할 때 이동되는 레코드의 수를 추적합니다.

RecordCount 속성은 이동시킬 레코드의 최대 수를 지정합니다. *RecordCount*가 0이면 소스 데이터셋의 첫째 레코드에서 시작하여 모든 레코드가 이동한 것입니다. *RecordCount*가 양수이면 소스 데이터셋의 현재 레코드에서 시작하여 최대의 *RecordCount* 레코드가 이동한 것입니다. *RecordCount*가 소스 데이터셋의 현재 레코드와 마지막 레코드 사이에 있는 레코드 수보다 큰 경우 소스 데이터셋의 끝에 도달하면 일괄 이동이 종료합니다. *MoveCount*를 검사하여 실제 이동된 레코드 수를 확인할 수 있습니다.

일괄 이동 오류 처리

일괄 이동 작업에서는 데이터 타입 변환 오류 및 무결성 위반 등 두 가지 유형의 오류가 발생할 수 있습니다. *TBatchMove*에는 오류 처리에 대해 보고하고 제어하는 여러 속성이 있습니다.

AbortOnProblem 속성은 데이터 타입 변환 오류가 발생하는 경우 작업을 중단할지 여부를 지정합니다. *AbortOnProblem*이 **true**인 경우 오류가 발생하면 일괄 이동 작업이 취소됩니다. **false**인 경우에는 작업이 계속됩니다. *ProblemTableName*에 지정한 테이블을 검사하여 어떤 레코드가 문제를 일으켰는지 확인할 수 있습니다.

AbortOnKeyViol 속성은 Paradox 키 위반이 발생하는 경우 작업을 중단할지 여부를 지정합니다.

ProblemCount 속성은 대상 테이블에서 데이터 손상 없이 처리될 수 없는 레코드 수를 나타냅니다. *AbortOnProblem*이 **true**인 경우 오류가 발생하면 작업이 중단하므로 이 값은 1입니다.

다음 속성을 사용하여 일괄 이동 작업을 문서화하는 추가 테이블을 생성할 수 있습니다.

- *ChangedTableName*을 지정하는 경우 업데이트 또는 삭제 작업에 따라 변경되는 대상 테이블의 모든 레코드를 포함하는 로컬 Paradox 테이블을 생성합니다.
- *KeyViolTableName*을 지정하는 경우 Paradox 테이블 작업을 수행할 때 키 위반을 일으킨 소스 테이블의 모든 레코드를 포함하는 로컬 Paradox 테이블을 생성합니다. *AbortOnKeyViol*이 **true**인 경우 첫 문제가 발생하면 작업이 중단하므로 이 테이블은 하나의 항목만 포함합니다.
- *ProblemTableName*을 지정한 경우 데이터 타입 변환 오류에 따라 대상 테이블에 포스트할 수 없는 모든 레코드를 포함하는 Paradox 테이블을 생성합니다. 예를 들어 테이블에 소스 테이블의 레코드 중 데이터를 대상 테이블에 적절하게 잘라내야 하는 레코드가 포함될 수 있습니다. *AbortOnProblem*이 **true**인 경우 첫 문제가 발생하면 작업이 중단되므로 이 테이블에는 하나의 레코드만 있습니다.

참고 *ProblemTableName*을 지정하지 않는 경우에는 레코드의 데이터를 잘라내어 대상 테이블에 넣습니다.

Data Dictionary

BDE를 사용하여 데이터를 액세스하는 경우 애플리케이션이 Data Dictionary를 액세스할 수 있습니다. Data Dictionary는 애플리케이션에 독립적이고 사용자 정의할 수 있는 저장소 영역을 제공합니다. 여기에서 데이터의 내용과 모양을 설명하는 확장 필드 어트리뷰트(attribute) 집합을 만들 수 있습니다.

예를 들어 금융 애플리케이션을 빈번하게 개발하는 경우 다양한 통화 표시 형식을 설명하는 특수 필드 어트리뷰트 집합을 만들 수 있습니다. 애플리케이션의 데이터셋을 만들 때 Object Inspector를 사용하여 각 데이터셋의 통화 필드를 수동으로 설정하는 대신 이 필드를Data Dictionary에 있는 확장 필드 어트리뷰트 집합에 연결할 수 있습니다. Data Dictionary를 사용하면 생성하는 여러 애플리케이션 사이에서 일관성 있는 데이터 모양을 유지할 수 있습니다.

클라이언트/서버 환경인 경우 추가적인 정보 공유를 위해 Data Dictionary를 원격 서버에 상주시킬 수 있습니다.

디자인 타임에 Fields Editor에서 확장 필드 어트리뷰트 집합을 만드는 방법과 이 집합을 애플리케이션의 데이터셋에 있는 필드에 연결하는 방법에 대한 자세한 내용은 23-12페이지의 "필드 컴포넌트에 대한 어트리뷰트(attribute) 집합 생성"을 참조하십시오. SQL 및 Database 탐색기를 사용하여 Data Dictionary 및 확장 필드 어트리뷰트를 생성하는 방법에 대한 자세한 내용은 해당 온라인 도움말을 참조하십시오.

Data Dictionary에 대한 프로그래밍 인터페이스는 drintf 헤더 파일(include\VCL 디렉토리에 있음)에 있습니다. 이 인터페이스에서는 다음과 같은 메소드를 제공합니다.

표 24.9 Data Dictionary 인터페이스

루틴	용도
DictionaryActive	Data Dictionary가 활성인지 여부를 나타냅니다.
DictionaryDeactivate	Data Dictionary를 비활성화합니다.
IsNullID	지정한 ID가 Null ID인지 여부를 나타냅니다.
FindDatabaseID	지정한 알리아스에 대한 데이터베이스의 ID를 반환합니다.
FindTableID	지정한 데이터베이스에 있는 테이블의 ID를 반환합니다.
FindFieldID	지정한 테이블에 있는 필드의 ID를 반환합니다.
FindAttrID	이름이 지정된 어트리뷰트(attribute) 집합의 ID를 반환합니다.
GetAttrName	지정한 ID의 어트리뷰트 집합에 대한 이름을 반환합니다.
GetAttrNames	Data dictionary에 있는 각 어트리뷰트 집합에 대해 콜백을 실행합니다.
GetAttrID	지정된 필드에 대해 어트리뷰트 집합의 ID를 반환합니다.
NewAttr	필드 컴포넌트에서 새 어트리뷰트 집합을 만듭니다.
UpdateAttr	필드 속성과 일치하는 어트리뷰트 집합을 업데이트합니다.
CreateField	저장된 어트리뷰트에 따라 필드 컴포넌트를 생성합니다.
UpdateField	필드 속성을 지정된 어트리뷰트 집합과 일치하도록 변경합니다.
AssociateAttr	어트리뷰트 집합을 지정된 필드 ID와 연결합니다.
UnassociateAttr	필드 ID의 어트리뷰트 집합 연결을 제거합니다.
GetControlClass	지정된 어트리뷰트 ID의 컨트롤 클래스를 반환합니다.
QualifyTableName	사용자 이름으로 한정한 전체 테이블 이름을 반환합니다.
QualifyTableNameByName	사용자 이름으로 한정한 전체 테이블 이름을 반환합니다.
HasConstraints	데이터셋이 Data dictionary에 있는 제약 조건을 가지는지 여부를 나타냅니다.
UpdateConstraints	임포트된 데이터셋의 제약 조건을 업데이트합니다.
UpdateDataset	데이터셋을 Data dictionary의 현재 설정 및 제약 조건으로 업데이트합니다.

BDE 작업 도구

BDE를 데이터 액세스 매커니즘으로 사용하는 경우의 한 가지 장점은 C++Builder에 탑재된 풍부한 지원 유틸리티를 사용할 수 있다는 점입니다. 이 유틸리티는 다음과 같습니다.

- **SQL 탐색기** 및 **Database 탐색기:** C++Builder는 구입한 버전에 따라 두 가지 애플리케이션 중 하나를 탑재하고 있습니다. 두 탐색기를 사용하여 다음을 할 수 있습니다.
  - 기존 데이터베이스 테이블 및 구조를 검사합니다. SQL 탐색기를 사용하면 원격 SQL 데이터베이스를 검사 및 쿼리할 수 있습니다.
  - 테이블을 데이터로 채웁니다.
  - Data Dictionary에 확장 필드 어트리뷰트(attribute) 집합을 만들거나 이 집합을 애플리케이션의 필드에 연결합니다.
  - BDE 알리아스를 생성 및 관리합니다.

  SQL 탐색기를 사용하면 추가로 다음을 할 수 있습니다.
  - 원격 데이터베이스 서버에 내장 프로시저와 같은 SQL 객체를 생성합니다.
  - 원격 데이터베이스 서버에 있는 SQL 객체의 다시 구성한 텍스트를 봅니다.
  - SQL 스크립트를 실행합니다.
- **SQL Monitor:** SQL Monitor를 사용하면 원격 데이터베이스 서버와 BDE 간에 전송하는 모든 통신을 감시할 수 있습니다. 관심 범주만으로 한정하여 감시할 메시지를 필터링할 수 있습니다. SQL Monitor는 애플리케이션을 디버깅할 때 가장 유용합니다.
- **BDE Administration 유틸리티:** BDE Administration 유틸리티를 사용하면 새 데이터베이스 드라이버 추가, 기존 드라이버의 기본값 구성 및 새 BDE 알리아스 생성 등을 할 수 있습니다.
- **Database Desktop:** Paradox 또는 dBASE 테이블을 사용하는 경우 Database Desktop을 사용하면 데이터 보기 및 편집, 새 테이블 생성, 기존 테이블 다시 구성 등을 할 수 있습니다. Database Desktop을 사용하면 *TTable* 컴포넌트의 메소드를 사용하는 것보다 더 잘 제어할 수 있습니다. 예를 들어 유효성 검사와 랭귀지 드라이버를 지정할 수 있습니다. Database Desktop은 BDE의 API를 직접 호출하는 것 이외에 Paradox 및 dBASE 테이블을 다시 구성할 수 있는 유일한 메커니즘을 제공합니다.

장

25

ADO 컴포넌트 사용

dbGo 컴포넌트는 ADO 프레임워크를 통한 데이터 액세스를 제공합니다. ADO(Microsoft ActiveX Data Objects)는 OLE DB 프로바이더를 통해 데이터에 액세스하는 COM 객체 집합입니다. *dbGo* 컴포넌트는 C++Builder 데이터베이스 아키텍처에서 이 ADO 객체들을 캡슐화합니다.

ADO 기반 애플리케이션의 ADO 레이어는 Microsoft ADO 2.1, 데이터 저장소 액세스용 OLE DB 프로바이더 또는 ODBC 드라이버, 사용되는 특정 데이터베이스 시스템을 위한 클라이언트 소프트웨어(SQL 데이터베이스의 경우), 애플리케이션에 액세스할 수 있는 데이터베이스 백 엔드 시스템(SQL 데이터베이스 시스템용) 및 데이터베이스로 구성됩니다. ADO를 기반으로 하는 애플리케이션은 이러한 모든 구성 요소에 액세스할 수 있어야 제 기능을 수행할 수 있습니다.

가장 대표적인 ADO 객체는 Connection, Command 및 Recordset 객체입니다. 이러한 ADO 객체는 *TADOConnection*, *TADOCommand* 및 ADO 데이터셋 컴포넌트에 의해 랩핑됩니다. ADO 프레임워크에는 Field 및 Properties 객체 등의 다른 "helper" 객체가 포함되어 있지만 이러한 객체는 일반적으로 dbGo 애플리케이션에서 직접 사용되지 않으며 전용 컴포넌트에 의해 랩핑되지 않습니다.

이 장에서는 *dbGo* 컴포넌트를 소개하고 이 컴포넌트가 C++Builder 데이터베이스 아키텍처에 제공하는 고유한 기능에 대해 설명합니다. *dbGo* 컴포넌트의 기능에 대한 내용을 읽기 전에 먼저 21장의 "데이터베이스에 연결" 및 22장의 "데이터셋 이해"에 설명되어 있는 데이터베이스 연결 컴포넌트 및 데이터셋의 기능을 잘 알아야 합니다.

ADO 컴포넌트 개요

컴포넌트 팔레트의 ADO 페이지는 *dbGo* 컴포넌트를 호스팅합니다. 이 컴포넌트를 사용하여 ADO 데이터 저장소에 연결하고, 명령을 실행하며, ADO 프레임워크를 사용하여 데이터베이스 테이블에서 데이터를 검색할 수 있습니다. 이 컴포넌트를 사용하려면 ADO 2.1 이상을 호스트 컴퓨터에 설치해야 합니다. 또한 Microsoft SQL Server와 같은 대상 데이터베이스 시스템을 위한 클라이언트 소프트웨어와 특정 데이터베이스 시스템용 OLE DB 드라이버나 ODBC 드라이버도 설치해야 합니다.

대부분의 *dbGo* 컴포넌트에는 다른 데이터 액세스 방법을 위해 사용할 수 있는 직접적인 카운터파트인 데이터베이스 연결 컴포넌트 *TADOConnection*과 여러 타입의 데이터셋이 있습니다. 그 외에 *dbGo*에는 데이터셋은 아니지만 ADO 데이터 저장소에서 실행될 SQL 명령을 나타내는 간단한 컴포넌트인 *TADOCommand*가 포함됩니다.

다음 표는 ADO 컴포넌트를 나열한 것입니다.

표 25.1 ADO 컴포넌트

컴포넌트	용도
TADOConnection	ADO 데이터 저장소와의 연결을 수립하는 데이터베이스 연결 컴포넌트. 여러 ADO 데이터셋과 커맨드 컴포넌트에서 이 연결을 공유하여 명령을 실행하고 데이터를 검색하며 메타데이터 작업을 수행할 수 있습니다.
TADODataSet	데이터 검색 및 작업을 위한 기본 데이터셋. *TADODataSet*은 하나 또는 여러 테이블에서 데이터를 검색할 수 있으며 데이터 저장소에 직접 연결하거나 *TADOConnection* 컴포넌트를 사용할 수 있습니다.
TADOTable	단일 데이터베이스 테이블에 의해 생성되는 레코드셋 검색 및 작업을 위한 테이블 타입 데이터셋. *TADOTable*은 데이터 저장소에 직접 연결하거나 *TADOConnection* 컴포넌트를 사용할 수 있습니다.
TADOQuery	유효한 SQL문에 의해 생성된 레코드셋 검색 및 작업을 위한 쿼리 타입 데이터셋. *TADOQuery*는 데이터 정의 언어(DDL) SQL 문도 실행할 수 있으며 데이터 저장소에 직접 연결하거나 *TADOConnection* 컴포넌트를 사용할 수 있습니다.
TADOStoredProc	내장 프로시저를 실행하기 위한 내장 프로시저 타입 데이터셋. *TADOStoredProc*는 데이터를 검색하거나 또는 검색하지 않는 내장 프로시저를 실행하며 데이터 저장소에 직접 연결하거나 *TADOConnection* 컴포넌트를 사용할 수 있습니다.
TADOCommand	명령(결과 집합을 반환하지 않는 SQL 문) 실행을 위한 간단한 컴포넌트. *TADOCommand*는 지원 데이터셋 컴포넌트와 함께 사용하거나 테이블에서 데이터셋을 검색할 수 있으며, 데이터 저장소에 직접 연결하거나 *TADOConnection* 컴포넌트를 사용할 수 있습니다.

ADO 데이터 저장소에 연결

dbGo 애플리케이션에서는 Microsoft ActiveX Data Objects(ADO) 2.1을 사용하여 데이터 저장소에 연결하고 데이터에 액세스하는 OLE DB 프로바이더와 상호 작용합니다. 데이터 저장소가 나타낼 수 있는 항목 중 하나는 데이터베이스입니다. ADO 기반 애플리케이션은 클라이언트 컴퓨터에 ADO 2.1이 설치되어 있어야 합니다. ADO 및 OLE DB는 Microsoft에서 제공하며 Windows와 함께 설치됩니다.

ADO 프로바이더는 OLE DB 드라이버에서 ODBC 드라이버까지의 몇 가지 액세스 타입 중 하나를 나타냅니다. 이 드라이버들은 클라이언트 컴퓨터에 설치되어 있어야 합니다. 다양한 데이터베이스 시스템을 위한 OLE DB 드라이버가 데이터베이스 업체나 서드파티에 의해 공급되고 있습니다. 애플리케이션에서 Microsoft SQL Server나 Oracle과 같은 SQL 데이터베이스를 사용하는 경우 해당 데이터베이스 시스템을 위한 클라이언트 소프트웨어도 클라이언트 컴퓨터에 설치되어 있어야 합니다. 클라이언트 소프트웨어는 데이터베이스 업체에서 공급하며 데이터베이스 시스템 CD나 디스크에서 설치합니다.

애플리케이션을 데이터 저장소에 연결하려면 ADO 연결 컴포넌트인 *TADOConnection*을 사용하십시오. ADO 연결 컴포넌트는 ADO 프로바이더 중 하나를 사용하도록 구성합니다. *TADOConnection*을 반드시 사용해야 하는 것은 아니지만 ADO 명령과 데이터셋 컴포넌트는 *ConnectionString* 속성을 사용하여 직접 연결을 설정할 수 있기 때문에 *TADOConnection*을 사용하여 여러 ADO 컴포넌트 간에 하나의 연결을 공유할 수 있습니다. 이 방법을 사용하면 리소스 사용을 줄일 수 있고 다수의 데이터셋을 범위로 하는 트랜잭션을 만들 수 있습니다.

다른 데이터베이스 연결 컴포넌트와 달리 *TADOConnection*은 다음과 같은 지원을 제공합니다.

- 연결 제어
- 서버 로그인 제어
- 트랜잭션 관리
- 연결된 데이터셋을 사용한 작업
- 서버에 명령 보내기
- 메타데이터 얻기

모든 데이터베이스 연결 컴포넌트가 가지는 이러한 일반적인 기능 외에 *TADOConnection*은 다음과 같은 자체 지원을 제공합니다.

- 연결을 미세 조정하는데 사용할 수 있는 다양한 옵션
- 연결을 사용하는 명령 객체 리스트를 나열하는 능력
- 일반적인 작업 수행 시 추가 이벤트

TADOConnection을 사용하여 데이터 저장소에 연결

둘 이상의 ADO 데이터셋과 커맨드 컴포넌트가 *TADOConnection*을 사용하여 데이터 저장소에 대한 하나의 연결을 공유할 수 있습니다. 이렇게 하려면 *Connection* 속성을 통해 데이터셋과 커맨드 컴포넌트를 연결 컴포넌트와 연결하고, 디자인 타임에 Object Inspector의 *Connection* 속성 드롭다운 리스트에서 원하는 연결 컴포넌트를 선택한 다음, 런타임에 *Connection* 속성에 대한 참조를 지정합니다. 예를 들어, 다음 줄은 *TADODataSet* 컴포넌트를 *TADOConnection* 컴포넌트와 연결합니다.

```
ADODataSet1->Connection = ADOConnection1;
```

연결 컴포넌트는 ADO 연결 객체를 나타냅니다. 연결 객체를 사용하려면 연결할 데이터 저장소를 식별해야 합니다. 일반적으로 *ConnectionString* 속성을 사용하여 정보를 제공합니다. *ConnectionString*은 이름이 지정된 둘 이상의 연결 매개변수를 세미콜론으로 구분하여 나열한 문자열입니다. 이러한 매개변수는 연결 정보가 포함된 파일 이름이나 ADO 프로바이더 이름과 데이터 저장소를 식별하는 참조를 지정함으로써 데이터 저장소를 식별합니다. 다음과 같은 미리 정의된 매개변수 이름을 사용하여 이러한 정보를 제공합니다.

매개변수	설명
Provider	연결을 위해 사용하는 로컬 ADO 프로바이더의 이름
Data Source	데이터 저장소의 이름
File name	연결 정보가 포함된 파일의 이름

매개변수	설명
Remote Provider	원격 시스템에 상주하는 ADO 프로바이더의 이름
Remote Server	원격 프로바이더 사용 시 원격 서버의 이름

따라서 전형적인 *ConnectionString* 값은 다음과 같은 형태입니다.

```
Provider=MSDASQL.1;Data Source=MQIS
```

참고 *Provider* 속성을 사용하여 ADO 프로바이더를 지정하는 경우에는 *ConnectionString*의 연결 매개변수에 *Provider* 또는 *Remote Provider* 매개변수를 포함시킬 필요가 없습니다. 마찬가지로 *DefaultDatabase* 속성을 사용하는 경우에는 *Data Source* 매개변수를 지정할 필요가 없습니다.

*ConnectionString*에는 위에 언급한 매개변수 외에 사용 중인 특정 ADO 프로바이더 특유의 연결 매개변수가 포함될 수 있습니다. 로그인 정보를 하드코드화하는 경우에는 사용자 ID와 암호가 이러한 추가 연결 매개변수에 포함될 수 있습니다.

디자인 타임에 리스트에서 프로바이더 및 서버와 같은 연결 요소를 선택함으로써 Connection String Editor를 사용하여 연결 문자열을 생성할 수 있습니다. Object Inspector에서 *ConnectionString* 속성을 위한 생략 부호(...) 버튼을 클릭하여 ADO에 의해 제공되는 ActiveX 속성 에디터인 Connection String Editor를 시작합니다.

ConnectionString 속성과 옵션인 *Provider* 속성을 지정한 후에는 다른 속성을 사용하여 연결을 미세 조정할 수도 있지만 ADO 연결 컴포넌트를 사용하여 ADO 데이터 저장소에 연결하거나 또는 ADO 데이터 저장소와의 연결을 끊을 수 있습니다. 데이터 저장소와 연결하거나 연결을 끊을 때 *TADOConnection*을 사용하여 모든 데이터베이스 연결 컴포넌트에 일반적인 이벤트 외에 몇 가지 추가 이벤트에 응답할 수 있습니다. 이러한 추가 이벤트는 25-7페이지의 "연결을 설정할 때 발생하는 이벤트" 및 25-8페이지의 "연결을 끊을 때 발생하는 이벤트"에 설명되어 있습니다.

참고 연결 컴포넌트의 *Connected* 속성을 **true**로 설정하여 연결을 명시적으로 활성화하지 않으면 첫 번째 데이터셋 컴포넌트가 열리거나 ADO 커맨드 컴포넌트를 사용하여 처음으로 명령을 실행할 때 자동으로 연결됩니다.

연결 객체 액세스

원본으로 사용하는 ADO 연결 객체에 액세스하려면 *TADOConnection*의 *ConnectionObject* 속성을 사용하십시오. 이 참조를 사용하면 원본으로 사용하는 ADO 연결 객체의 속성에 액세스하고 메소드를 호출할 수 있습니다.

원본으로 사용하는 ADO 연결 객체를 사용하려면 ADO 객체와 특히 ADO 연결 객체에 대해 잘 알아야 합니다. 연결 객체 사용에 대해 잘 알고 있는 경우가 아니면 연결 객체를 사용하지 않는 것이 좋습니다. ADO 연결 객체 사용에 대한 구체적인 내용은 Microsoft Data Access SDK 도움말을 참조하십시오.

연결 미세 조정

ADO 명령과 데이터셋 컴포넌트에 대해 단순히 연결 문자열을 제공하는 대신 *TADOConnection*을 사용하여 데이터 저장소에 대한 연결을 설정할 경우 연결 상태와 어트리뷰트(attribute)를 강력히 제어할 수 있다는 장점이 있습니다.

비동기 연결 강제 적용

비동기 연결을 강제 적용하려면 *ConnectOptions* 속성을 사용하십시오. 비동기 연결을 사용하면 연결이 완전히 열리기를 기다리지 않고 애플리케이션을 계속 처리할 수 있습니다.

디폴트로, *ConnectionOptions*은 *coConnectUnspecified*로 설정되므로 서버에서 가장 좋은 연결 타입을 결정할 수 있습니다. 연결을 명시적으로 비동기로 만들려면 *ConnectOptions*를 *coAsyncConnect*로 설정합니다.

아래의 예제 루틴은 지정된 연결 컴포넌트에서 비동기 연결을 활성화하고 비활성화합니다.

```
void __fastcall TForm1::AsyncConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coAsyncConnect;
  ADOConnection1->Open();
}
void __fastcall TForm1::ServerChoiceConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coConnectUnspecified;
  ADOConnection1->Open();
}
```

시간 제한 제어

명령과 연결을 시도한 후 실패한 것으로 간주하여 중단하기까지 경과되는 시간을 *ConnectionTimeout* 및 *CommandTimeout* 속성을 사용하여 제어할 수 있습니다.

ConnectionTimeout 속성은 데이터 저장소에 대한 연결 시도 제한 시간을 초로 지정합니다. *ConnectionTimeout*에 지정된 시간이 만료되기 전에 연결이 성공적으로 컴파일되지 않으면 연결 시도는 취소됩니다.

```
  ADOConnection1->ConnectionTimeout = 10; // seconds
  ADOConnection1->Open();
```

CommandTimeout 속성은 시도한 명령의 제한 시간을 초로 지정합니다. *Execute* 메소드를 호출하여 시작된 명령이 *CommandTimeout*에 지정된 시간이 만료되기 전에 성공적으로 완료되지 않으면 명령은 취소되고 ADO에서는 예외를 생성합니다.

```
  ADOConnection1->ConnectionTimeout = 10;
  ADOConnection1->Execute("DROP TABLE Employee1997", cmdText,
TExecuteOptions());
```

연결에서 지원하는 작업 타입 표시

ADO 연결은 파일을 열 때 사용하는 모드와 비슷한 특정 모드를 사용하여 설정됩니다. 연결 모드는 연결에 사용할 수 있는 권한과 연결을 사용하여 수행할 수 있는 읽기, 쓰기 등 작업 타입을 결정합니다.

Mode 속성을 사용하여 연결 모드를 표시합니다. 아래의 표 25.2에 나열된 값을 사용할 수 있습니다.

표 25.2 ADO 연결 모드

연결 모드	의미
cmUnknown	연결 권한이 아직 설정되지 않았거나 알 수 없습니다.
cmRead	읽기 전용 권한을 사용할 수 있는 연결입니다.
cmWrite	쓰기 전용 권한을 사용할 수 있는 연결입니다.
cmReadWrite	읽기/쓰기 권한을 사용할 수 있는 연결입니다.
cmShareDenyRead	다른 사용자가 읽기 권한을 가지고 연결을 여는 것을 금지합니다.
cmShareDenyWrite	다른 사용자가 쓰기 권한을 가지고 연결을 여는 것을 금지합니다.
cmShareExclusive	다른 사용자가 연결을 여는 것을 금지합니다.
cmShareDenyNone	다른 사용자가 어떠한 권한으로도 연결을 여는 것을 금지합니다.

Mode 값으로 사용할 수 있는 값은 원본으로 사용하는 ADO 연결 객체에 대한 *Mode* 속성의 *ConnectModeEnum* 값과 일치합니다. 이에 대한 자세한 내용은 Microsoft Data Access SDK 도움말을 참조하십시오.

연결 시 트랜잭션 자동 시작 여부 지정

Attributes 속성을 사용하여 연결 컴포넌트의 retaining commits와 retaining aborts 사용을 제어합니다. 연결 컴포넌트에서 retaining commits을 사용하면 애플리케이션에서 트랜잭션을 커밋할 때마다 새로운 트랜잭션이 자동으로 시작되며, retaining aborts를 사용하면 애플리케이션에서 트랜잭션을 롤백할 때마다 새로운 트랜잭션이 자동으로 시작됩니다.

*Attributes*는 *xaCommitRetaining* 상수와 *xaAbortRetaining* 상수를 둘 다 포함하거나 둘 다 포함하지 않거나 또는 둘 중 하나를 포함할 수 있는 집합입니다. *Attributes*에 *xaCommitRetaining*이 포함된 경우에는 연결에서 retaining commits를 사용하며, *Attributes*에 *xaAbortRetaining*이 포함된 경우에는 연결에서 retaining aborts를 사용합니다.

Set *Contains* 메소드를 사용하여 retaining commits이나 retaining aborts 중 어느 하나가 활성화되어 있는지 여부를 확인하십시오. Attributes 속성에 적절한 값을 추가하여 retaining commits 또는 aborts를 활성화하고 그 값을 제거하여 비활성화합니다. 아래의 예제 루틴은 ADO 연결 컴포넌트에서 retaining commits를 활성화하고 비활성화합니다.

```
void __fastcall TForm1::RetainingCommitsOnButtonClick(TObject *Sender)
{
  ADOConnection1->Close()
  if (!ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() << xaCommitRetaining;
  ADOConnection1->Open()
}
```

```
void __fastcall TForm1::RetainingCommitsOffButtonClick(TObject *Sender)
{
  ADOConnection1->Close()
  if (ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() >> xaCommitRetaining;
  ADOConnection1->Open()
}
```

연결의 명령 액세스

다른 데이터베이스 연결 컴포넌트와 마찬가지로 *DataSets* 및 *DataSetCount* 속성을 사용하여 연결과 연관된 데이터셋에 액세스할 수 있습니다. 그러나, *dbGo*에는 *TADOCommand* 객체도 포함될 수 있습니다. 이 객체는 데이터셋은 아니지만 연결 컴포넌트와 비슷한 관계를 유지합니다.

*DataSets*와 *DataSetCount* 속성을 사용하여 연관된 데이터셋에 액세스하는 것과 같은 방법으로 *TADOConnection*의 *Commands* 및 *CommandCount* 속성을 사용하여 연관된 ADO 명령 객체에 액세스할 수 있습니다. *DataSets* 및 *DataSetCount* 속성은 활성 데이터넷만 나열하지만 *Commands* 및 *CommandCount* 속성은 연결 컴포넌트와 연관된 모든 *TADOCommand* 컴포넌트에 대한 참조를 제공합니다.

*Commands*는 인덱스가 0부터 시작하는 ADO 커맨드 컴포넌트 참조 배열입니다. *CommandCount*는 *Commands*에 나열된 모든 명령의 총 수를 제공합니다. 다음 코드에 나타난 것처럼 이 두 속성을 함께 사용하여 연결 컴포넌트를 사용하는 모든 명령을 통해 반복합니다.

```
for (int i = 0; i< ADOConnection2->CommandCount; i++)
  ADOConnection2->Commands[i]->Execute();
```

ADO 연결 이벤트

모든 데이터베이스 연결 컴포넌트에 대해 발생하는 일반적인 이벤트 외에, *TADOConnection*은 정상적으로 사용하는 동안 발생하는 몇 가지 추가 이벤트를 생성합니다.

연결을 설정할 때 발생하는 이벤트

*TADOConnection*은 연결을 설정할 때 모든 데이터베이스 연결 컴포넌트에 공통적인 *BeforeConnect* 이벤트와 *AfterConnect* 이벤트 외에도 *OnWillConnect* 이벤트와 *OnConnectComplete* 이벤트를 생성합니다. 이 두 이벤트는 *BeforeConnect* 이벤트 후에 발생합니다.

- *OnWillConnect* 이벤트는 ADO 프로바이더에서 연결을 설정하기 전에 발생합니다. 이 이벤트를 사용하여 연결 문자열 최종 변경, 자체 로그인 지원을 처리하는 경우 사용자 이름과 암호 제공, 비동기 연결 강제 적용 또는 연결이 열리기 전에 취소 등을 할 수 있습니다.
- *OnConnectComplete* 이벤트는 연결이 열린 후에 발생합니다. *TADOConnection*은 비동기 연결을 나타낼 수 있기 때문에 연결 컴포넌트에서 ADO 프로바이더에 연결을 열도록 지시한 후에 발생하기는 하지만 반드시 연결이 열린 후에 발생하는 것은 아닌 *AfterConnect* 이벤트를 사용하는 대신 연결이 열리거나 오류 상황으로 인해 실패한 후에 발생하는 *OnConnectComplete*를 사용해야 합니다.

연결을 끊을 때 발생하는 이벤트

*TADOConnection*는 연결을 끊은 후 모든 데이터베이스 연결 컴포넌트에 공통적인 *BeforeDisconnect* 이벤트와 *AfterDisconnect* 이벤트 외에도 *OnDisconnect* 이벤트를 생성합니다. *OnDisconnect* 이벤트는 연결이 끊긴 후이지만 연관된 데이터셋이 닫히기 전이고 *AfterDisconnect* 이벤트가 발생하기 전에 발생합니다.

트랜잭션을 관리할 때 발생하는 이벤트

ADO 연결 컴포넌트는 트랜잭션과 관련된 프로세스가 완료되었을 때를 감지하는 몇 가지 이벤트를 제공합니다. 이러한 이벤트는 *BeginTrans*, *CommitTrans*, 및 *RollbackTrans* 메소드가 데이터 저장소에서 성공적으로 완료되어 트랜잭션 프로세스가 시작된 때를 나타냅니다.

- *OnBeginTransComplete* 이벤트는 데이터 저장소에서 *BeginTrans* 메소드를 호출한 후 트랜잭션을 성공적으로 시작했을 때 발생합니다.
- *OnCommitTransComplete* 이벤트는 *CommitTrans* 메소드를 호출하여 트랜잭션이 성공적으로 커밋된 후 발생합니다.
- *OnRollbackTransComplete* 이벤트는 *RollbackTrans* 메소드를 호출하여 트랜잭션이 성공적으로 중단된 후 발생합니다.

기타 이벤트

ADO 연결 컴포넌트는 원본으로 사용하는 ADO 연결 객체의 통지에 응답하는 데 사용할 수 있는 두 가지 이벤트를 추가로 제공합니다.

- *OnExecuteComplete* 이벤트는 연결 컴포넌트가 데이터 저장소에서 명령을 실행한 후 발생합니다. 예를 들면, *Execute* 메소드를 호출한 후에 발생합니다. *OnExecuteComplete* 이벤트는 실행이 성공적이었는지 여부를 나타냅니다.
- *OnInfoMessage* 이벤트는 원본으로 사용하는 연결 객체에서 작업이 완료된 후 자세한 정보를 제공할 때 발생합니다. *OnInfoMessage* 이벤트 핸들러는 작업이 성공적이었는지 여부를 나타내는 자세한 정보와 상태 코드가 포함된 ADO Error 객체에 대한 인터페이스를 받습니다.

ADO 데이터셋 사용

ADO 데이터셋 컴포넌트는 ADO 레코드셋 객체를 캡슐화합니다 . ADO 데이터셋 컴포넌트는 ADO를 사용하여 22장의 "데이터셋 이해"에 설명된 일반적인 데이터셋 기능을 상속하여 구현을 제공합니다. ADO 데이터셋을 사용하려면 이러한 일반적인 기능에 대해 잘 알아야 합니다.

모든 ADO 데이터셋은 일반적인 데이터셋 기능 외에 속성, 이벤트 및 메소드를 추가하여 다음과 같은 기능들을 제공합니다.

- ADO 데이터 저장소에 연결
- 원본으로 사용하는 Recordset 객체 액세스
- 북마크를 사용한 레코드 필터링
- 레코드 비동기 페치
- 배치 업데이트(캐싱된 업데이트) 수행
- 디스크 상의 파일을 사용하여 데이터 저장

다음 네 가지의 ADO 데이터셋이 있습니다.

- *TADOTable*, 단일 데이터베이스 테이블의 모든 행과 열을 나타내는 테이블 타입 데이터셋. *TADOTable* 및 기타 테이블 타입 데이터셋 사용에 대한 내용은 22-24페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.
- *TADOQuery*, SQL 문을 캡슐화하고 애플리케이션이 그 결과 얻어지는 레코드에 액세스할 수 있도록 해주는 쿼리 타입 데이터셋. *TADOQuery* 및 기타 쿼리 타입 데이터셋 사용에 대한 내용은 22-40페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오.
- *TADOStoredProc*, 데이터베이스 서버에 정의된 내장 프로시저를 실행하는 내장 프로시저 타입 데이터셋. *TADOStoredProc* 및 기타 내장 프로시저 타입 데이터셋 사용에 대한 내용은 22-48페이지의 "내장 프로시저 타입의 데이터셋 사용"을 참조하십시오.
- *TADODataSet*, 다른 세 가지 타입의 기능을 포함하는 다용도 데이터셋. *TADODataSet*의 고유 기능에 대한 설명은 25-15페이지의 "TADODataSet 사용"을 참조하십시오.

참고 ADO를 사용하여 데이터베이스 정보에 액세스할 때는 커서를 반환하지 않는 SQL 명령을 나타내기 위해 *TADOQuery*와 같은 데이터셋을 사용할 필요가 없습니다. 그 대신 데이터셋이 아닌 간단한 컴포넌트인 *TADOCommand*를 사용할 수 있습니다. *TADOCommand*에 대한 자세한 내용은 25-16페이지의 "명령 객체 사용"을 참조하십시오.

데이터 저장소에 ADO 데이터셋 연결

ADO 데이터셋은 집합적 또는 개별적으로 ADO 데이터 저장소에 연결될 수 있습니다.

데이터셋을 집합적으로 연결할 때는 각 데이터셋의 *Connection* 속성을 *TADOConnection* 컴포넌트로 설정합니다. 그런 다음 각 데이터셋에서 ADO 연결 컴포넌트의 연결을 사용할 수 있습니다.

```
ADODataSet1->Connection = ADOConnection1;
ADODataSet2->Connection = ADOConnection1;
...
```

데이터셋을 집합적으로 연결하면 다음과 같은 이점이 있습니다.

- 데이터셋이 연결 객체의 어트리뷰트(attribute)를 공유합니다.
- 한 가지 연결, 즉 *TADOConnection*의 연결만 설정하면 됩니다.
- 데이터셋이 트랜잭션에 참여할 수 있습니다.

TADOConnection 사용에 대한 자세한 내용은 25-2페이지의 "ADO 데이터 저장소에 연결"을 참조하십시오.

데이터셋을 개별적으로 연결할 때는 각 데이터셋의 *ConnectionString* 속성을 설정합니다. *ConnectionString*을 사용하는 각 데이터셋은 애플리케이션에 있는 다른 데이터셋 연결과 관계없이 데이터 저장소에 대해 자체적으로 연결을 설정합니다.

ADO 데이터셋의 *ConnectionString* 속성은 *TADOConnection*의 *ConnectionString* 속성과 같은 방법으로 사용됩니다. 이 속성은 다음과 같이 세미콜론을 사용하여 구분한 연결 매개변수의 집합입니다.

```
ADODataSet1->ConnectionString =
"Provider=YourProvider;Password=SecretWord;";
ADODataSet1->ConnectionString += "User ID=JaneDoe;SERVER=PURGATORY";
ADODataSet1->ConnectionString += "UID=JaneDoe;PWD=SecretWord;"
ADODataSet1->ConnectionString += "Initial Catalog=Employee";
```

디자인 타임에 Connection String Editor를 사용하여 연결 문자열을 작성할 수 있습니다. 연결 문자열에 대한 자세한 내용은 25-3페이지의 "TADOConnection을 사용하여 데이터 저장소에 연결"을 참조하십시오.

레코드셋 작업

Recordset 속성은 데이터셋 컴포넌트의 원본으로 사용하는 ADO 레코드셋 객체를 직접 액세스할 수 있게 합니다. 이 객체를 사용하면 애플리케이션에서 레코드셋 객체의 속성과 호출 메소드에 액세스할 수 있습니다. 원본으로 사용하는 ADO 레코드셋 객체에 직접 액세스하기 위해 *Recordset*을 사용하려면 ADO 객체에 대한 일반적인 내용과 특히 ADO 레코드셋 객체에 대해 잘 알아야 합니다 . 레코드셋 객체 작업에 대해 잘 모르는 경우에는 레코드셋 객체를 직접 사용하지 않는 것이 좋습니다 . ADO 레코드셋 객체 사용에 대한 구체적인 내용은 Microsoft Data Access SDK 도움말을 참조하십시오.

RecordsetState 속성은 원본으로 사용하는 레코드셋 객체의 현재 상태를 나타냅니다. *RecordsetState*는 ADO 레코드셋 객체의 *State* 속성과 일치합니다. *RecordsetState*의 값은 *stOpen*, *stExecuting* 또는 *stFetching* 중 하나입니다. *RecordsetState* 속성의 타입인 *TObjectState*는 다른 값들을 정의하지만 *stOpen*, *stExecuting* 및 *stFetching*만 레코드셋과 관련됩니다. *stOpen* 값은 레코드셋이 현재 유휴 상태라는 것을 나타냅니다 . *stExecuting* 값은 명령을 실행하는 중이라는 것을 나타냅니다. *stFetching*은 연관된 테이블에서 행을 가져오는 중이라는 것을 나타냅니다.

RecordsetState 값을 사용하여 데이터셋의 현재 상태와 관계 없이 작업을 수행하십시오. 예를 들면, 데이터를 업데이트하는 루틴을 사용하여 *RecordsetState* 속성을 확인하여 데이터셋이 활성화되어 있고 데이터를 연결하거나 가져오는 등 다른 작업을 수행하는 중이 아닌지 알 수 있습니다.

북마크를 사용한 레코드 필터링

ADO 데이터셋은 북마크를 사용하여 특정 레코드를 표시하여 반환하는 일반적인 데이터셋 기능을 지원합니다. 또한 다른 데이터셋과 마찬가지로 필터를 사용하여 데이터셋에서 사용할 수 있는 레코드를 제한할 수 있도록 해줍니다. ADO 데이터셋은 이러한 두 가지 일반적인 데이터셋 기능을 결합하여 북마크로 표시된 레코드 집합을 필터링하는 기능을 추가 제공합니다.

다음과 같은 방법으로 북마크 집합에 대해 필터링을 수행합니다.

1. *Bookmark* 메소드를 사용하여 필터링된 데이터셋에 포함시키려는 레코드에 북마크를 표시합니다.
2. *FilterOnBookmarks* 메소드를 호출하여 북마크 표시된 레코드만 나타나도록 데이터셋을 필터링합니다.

다음 예제는 이 프로세스를 보여 줍니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TBookmarkStr BM1;
  TBookmarkStr BM2;
  BM1 = ADODataSet1->Bookmark;
  BMList->Add(BM1);
  ADODataSet1->MoveBy(3);
  BM2 = ADODataSet1->Bookmark;
  BMList->Add(BM2);
  ADODataSet1->FilterOnBookmarks(ARRAYOFCONST((BM1,BM2)));
}
```

위 예제에는 BMList라는 이름의 리스트 객체에 대한 북마크도 추가되어 있습니다. 이 북마크는 나중에 애플리케이션에서 북마크가 더 이상 필요하지 않을 때 사용할 수 있게 하기 위해 필요합니다.

북마크 사용에 대한 자세한 내용은 22-9페이지의 "레코드 표시 및 레코드로 돌아가기"를 참조하십시오. 그 밖의 필터 타입에 대한 자세한 내용은 22-12페이지의 "필터를 사용하여 데이터의 부분 집합 표시 및 편집"을 참조하십시오.

레코드 비동기 페치

다른 데이터셋과 달리 ADO 데이터셋은 데이터를 비동기로 가져올 수 있습니다. 따라서 데이터 저장소의 데이터를 사용하여 데이터셋을 채우는 동안 애플리케이션에서 다른 작업을 계속 수행할 수 있습니다.

데이터셋에서 데이터를 가져올 경우 비동기로 할 것인지 여부를 제어하려면 *ExecuteOptions* 속성을 사용하십시오. *ExecuteOptions* 속성은 *Open* 메소드를 호출하거나 *Active*를 **true**로 설정할 때 데이터셋에서 레코드를 가져오는 방법을 결정합니다. 데이터셋이 레코드를 반환하지 않는 쿼리나 내장 프로시저를 나타내는 경우에는 *ExecSQL*이나 *ExecProc*를 호출할 때 쿼리나 내장 프로시저가 실행되는 방법을 결정합니다.

*ExecuteOptions*는 다음 값을 하나도 포함하지 않거나 하나 이상 포함하는 집합입니다.

표 25.3 ADO 데이터셋의 실행 옵션

실행 옵션	의미
eoAsyncExecute	명령 또는 데이터 페치가 비동기로 실행됩니다.
eoAsyncFetch	데이터셋은 먼저 *CacheSize* 속성에 의해 지정된 레코드의 수를 동기적으로 가져온 다음 남은 행이 있으면 비동기로 가져옵니다.
eoAsyncFetchNonBlocking	비동기 데이터 페치 또는 명령 실행이 현재 실행 스레드를 차단하지 않습니다.
eoExecuteNoRecords	데이터를 반환하지 않는 명령 또는 내장 프로시저. 읽어들인 행이 있으면 폐기하고 반환하지 않습니다.

배치 업데이트 사용

캐싱된 업데이트 방법 중 하나는 데이터셋 프로바이더를 사용하여 ADO 데이터셋을 클라이언트 데이터셋에 연결하는 것입니다. 이 방법은 27-15페이지의 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용"에 설명되어 있습니다.

그러나, ADO 데이터셋 컴포넌트는 배치 업데이트라 불리는 캐싱된 업데이트를 지원합니다. 다음 표는 클라이언트 데이터셋을 사용한 캐싱된 업데이트와 배치 업데이트 기능을 사용한 캐싱된 업데이트를 비교한 것입니다.

표 25.4 ADO 및 클라이언트 데이터셋에 캐싱된 업데이트 비교

ADO 데이터셋	TClientDataSet	설명
LockType	사용 안함. 항상 클라이언트 데이터셋에 캐싱된 업데이트	데이터셋이 배치 업데이트 모드로 열리는 지 여부를 지정합니다.
CursorType	사용 안함. 클라이언트 데이터셋은 항상 데이터의 인메모리(in-memory) 스냅샷을 사용합니다.	ADO 데이터셋이 서버의 변경 사항과 분리된 정도를 지정합니다.
RecordStatus	UpdateStatus	현재 행에 대해 어떤 업데이트가 이루어졌는지 나타냅니다. *RecordStatus*는 *UpdateStatus*보다 더 많은 정보를 제공합니다.
FilterGroup	StatusFilter	어떤 타입의 레코드를 사용할 수 있는지 지정합니다. *FilterGroup*은 더 다양한 정보를 제공합니다.
UpdateBatch	ApplyUpdates	캐싱된 업데이트를 데이터베이스 서버에 다시 적용합니다. *ApplyUpdates*와 달리 *UpdateBatch*를 사용하면 적용될 업데이트 타입을 제한할 수 있습니다.
CancelBatch	CancelUpdates	보류 중인 업데이트를 취소하고 원래의 값으로 되돌립니다. *CancelUpdates*와 달리 *CancelBatch*를 사용하면 취소할 업데이트 타입을 제한할 수 있습니다.

ADO 데이터셋 컴포넌트의 배치 업데이트 기능 사용과 관련된 작업은 다음과 같습니다.

- 배치 업데이트 모드로 데이터셋 열기
- 각 행의 업데이트 상태 조사
- 업데이트 상태에 따라 여러 행 필터링
- 기본 테이블에 배치 업데이트 적용
- 배치 업데이트 취소

배치 업데이트 모드로 데이터셋 열기

ADO 데이터셋을 배치 업데이트 모드에서 열려면 다음 기준에 맞아야 합니다.

1. 컴포넌트의 *CursorType* 속성 값은 디폴트 속성 값인 *ctKeySet*이거나 *ctStatic*이어야 합니다.
2. *LockType* 속성은 *ltBatchOptimistic*이어야 합니다.
3. 명령은 SELECT 쿼리라야 합니다.

데이터셋 컴포넌트를 활성화하기 전에 *CursorType* 속성과 *LockType* 속성을 위와 같이 설정합니다. 컴포넌트의 *CommandText* 속성(*TADODataSet*인 경우) 또는 *SQL* 속성(*TADOQuery*인 경우)에 SELECT 문을 할당합니다. *TADOStoredProc* 컴포넌트인 경우에는 결과 집합을 반환하는 내장 프로시저의 이름에 *ProcedureName*을 설정합니다. 이 속성들은 디자인 타임에 Object Inspector를 통해 설정하거나 런타임에 프로그램을 사용하여 설정할 수 있습니다. 아래의 예제는 배치 업데이트 모드를 위해 *TADODataSet* 컴포넌트를 준비하는 것을 보여 줍니다.

```
ADODataSet1->CursorLocation = clUseClient;
ADODataSet1->CursorType = ctStatic;
ADODataSet1->LockType = ltBatchOptimistic;
ADODataSet1->CommandType = cmdText;
ADODataSet1->CommandText = "SELECT * FROM Employee";
```

배치 업데이트 모드에서 데이터셋이 열린 후 데이터에 대한 모든 변경 사항은 기준 테이블에 직접 적용되지 않고 캐시에 저장됩니다.

각 행의 업데이트 상태 조사

주어진 행을 현재 행으로 만든 다음 ADO 데이터 컴포넌트의 *RecordStatus* 속성을 조사하여 업데이트 상태를 확인합니다. *RecordStatus* 속성은 현재 행과 그 행만의 업데이트 상태를 나타냅니다.

```
switch (ADOQuery->RecordStatus)
{
  case rsUnmodified:
    StatusBar1->Panels->Items[0]->Text = "Unchanged record";
    break;
  case rsModified:
    StatusBar1->Panels->Items[0]->Text = "Changed record";
    break;
  case rsDeleted:
    StatusBar1->Panels->Items[0]->Text = "Deleted record";
    break;
  case rsNew:
    StatusBar1->Panels->Items[0]->Text = "New record";
    break;
}
```

업데이트 상태에 따라 여러 행 필터링

업데이트 상태가 같은 행 그룹에 속한 행들만 표시하려면 *FilterGroup* 속성을 사용하여 레코드셋을 필터링합니다. *FilterGroup* 속성을 표시할 행의 업데이트 상태를 나타내는 *TFilterGroup* 상수로 설정합니다. 이 속성의 기본값인 *fgNone* 값은 필터링이 적용되지 않고 삭제 표시가 있는 행 외에는 업데이트 상태에 관계없이 모든 행이 표시되도록 지정합니다. 아래 예제는 보류 중인 배치 업데이트 행만 표시합니다.

```
FilterGroup = fgPendingRecords;
Filtered = true;
```

참고 *FilterGroup* 속성이 유효성을 가지려면 ADO 데이터셋 컴포넌트의 *Filtered* 속성이 **true**로 설정되어 있어야 합니다.

기본 테이블에 배치 업데이트 적용

아직 적용되지 않았거나 취소되지 않았던 보류 중인 데이터 변경 사항을 적용하려면 *UpdateBatch* 메소드를 호출합니다. 변경되어 적용되는 행은 변경 사항이 레코드셋의 기준이 있는 기준 테이블에 적용됩니다. 삭제 표시가 있는 캐싱된 행은 기준 테이블의 해당 행이 삭제됩니다. 캐시에는 있지만 기준 테이블에는 없는 레코드 삽입은 기준 테이블에 추가됩니다. 수정된 행은 기준 테이블에서 해당 행의 열이 캐시에 있는 새 열 값으로 변경됩니다.

*UpdateBatch*는 *TAffectRecords* 값 하나만 매개변수로 받아들입니다. *arAll* 이외의 값이 전달되는 경우에는 보류 중인 변경 사항의 부분 집합만 적용됩니다. 아래의 예제는 현재 활성화되어 있는 적용할 행만 적용합니다.

```
ADODataSet1->UpdateBatch(arCurrent);
```

배치 업데이트 취소

아직 취소되지 않았거나 적용되지 않은 보류 중인 데이터 변경 사항을 취소하려면 *CancelBatch* 메소드를 호출합니다. 보류 중인 배치 업데이트를 취소하면 변경되었던 행의 필드 값이 *CancelBatch* 메소드나 *UpdateBatch* 메소드가 호출되었던 경우에는 최종 호출 이전에 존재했던 값으로 되돌아가고 그렇지 않으면 현재 보류 중인 변경 사항 배치 이전의 값으로 되돌아갑니다.

*CancelBatch*는 *TAffectRecords* 값 하나만 매개변수로 받아들입니다. *arAll* 이외의 값이 전달되는 경우에는 보류 중인 변경 사항의 부분 집합만 취소됩니다. 아래 예제는 모든 보류 중인 변경 사항을 취소합니다.

```
ADODataSet1->CancelBatch(arAll);
```

파일에서 데이터 로드 및 파일에 데이터 저장

ADO 데이터셋 컴포넌트를 통해 읽어들인 데이터를 파일에 저장하여 나중에 동일한 컴퓨터 또는 다른 컴퓨터에서 읽어들일 수 있습니다. 데이터는 ADTG 형식이나 XML 형식 중 하나로 저장됩니다. ADO에서는 이 두 가지 파일 형식만 지원됩니다. 그러나, 모든 ADO 버전에서 두 형식이 모두 지원되는 것은 아닙니다. 어떤 저장 파일 형식이 지원되는지 확인하려면 사용하고 있는 버전의 ADO 설명서를 참조하십시오.

데이터를 파일에 저장하려면 *SaveToFile* 메소드를 호출합니다. *SaveToFile* 메소드는 데이터가 저장되는 파일의 이름과 데이터 저장 형식(ADTG 또는 XML)을 매개변수로 사용합니다. *Format* 매개변수를 *pfADTG*이나 *pfXML*로 설정하여 저장된 파일의 형식을 나타냅니다. *FileName* 매개변수에 의해 지정된 파일이 이미 존재하면 *SaveToFile* 메소드는 *EOleException*을 발생시킵니다.

파일에서 데이터를 읽어들이려면 *LoadFromFile* 메소드를 사용합니다. *LoadFromFile* 메소드는 로드할 파일의 이름을 매개변수로 사용합니다. 지정된 파일이 존재하지 않으면 *LoadFromFile* 메소드는 *EOleException* 예외를 발생시킵니다. *LoadFromFile* 메소드를 호출하면 데이터셋 컴포넌트가 자동으로 활성화됩니다.

아래 예제에서 첫 번째 프로시저는 *TADODataSet* 컴포넌트 *ADODataSet1*에서 읽어들인 데이터셋을 파일에 저장합니다. 대상 파일은 로컬 드라이브에 저장된 SaveFile이라는 이름의 ADTG 파일입니다. 두 번째 프로시저는 이 저장된 파일을 *TADODataSet* 컴포넌트 *ADODataSet2*에 로드합니다.

```
void __fastcall TForm1::SaveBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
  {
    DeleteFile("c:\\SaveFile");
    Statusbar1->Panels->Items[0]->Text = "Save file deleted!";
  }
  ADODataSet1->SaveToFile("c:\\SaveFile");
}
```

```
void __fastcall TForm1::LoadBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
    ADODataSet1->LoadFromFile("c:\\SaveFile");
  else
    Statusbar1->Panels->Items[0]->Text = "Save file does not exist!";
}
```

데이터를 저장하고 로드하는 데이터셋들은 위와 같은 형태가 아니어도 되며 같은 애플리케이션에 없어도 되고 같은 컴퓨터에 있지 않아도 됩니다. 따라서 데이터를 서류 가방에 넣어 옮기는 것처럼 한 컴퓨터에서 다른 컴퓨터로 전송할 수 있습니다.

TADODataSet 사용

*TADODataSet*은 ADO 데이터 저장소의 데이터를 사용한 작업을 위한 범용 데이터셋입니다. 다른 ADO 데이터셋 컴포넌트와 달리, *TADODataSet*은 테이블 타입, 쿼리 타입 또는 내장 프로시저 타입 데이터셋이 아닙니다. 그러나 이 세 타입 모두의 기능을 모두 수행할 수 있습니다.

- *TADODataSet*을 테이블 타입 데이터셋처럼 사용하여 단일 데이터베이스 테이블의 모든 행과 열을 나타낼 수 있습니다. 이러한 방법으로 사용하려면 *CommandType* 속성을 *cmdTable*로 설정하고 *CommandText* 속성을 테이블 이름으로 설정하십시오. *TADODataSet*은 다음과 같은 테이블 타입 작업을 지원합니다.
  - 인덱스를 지정하여 레코드를 정렬하거나 레코드 기반 검색 기준 형성. *TADODataSet*를 사용하면 22-25페이지의 "인덱스를 사용하여 레코드 정렬"에 설명되어 있는 기본 인덱스 속성 및 메소드 외에 *Sort* 속성을 설정함으로써 임시 인덱스를 사용하여 정렬할 수있습니다. *Seek* 메소드를 사용하여 수행되는 인덱스 기반 검색에는 현재 인덱스가 사용됩니다.
  - 데이터셋 비우기. *DeleteRecords* 메소드를 사용하여 삭제할 레코드를 지정할 수 있기 때문에 다른 테이블 타입 데이터셋의 관련 메소드보다 .

  *TADODataSet*에 의해 지원되는 테이블 타입 작업은 *cmdTable*의 *CommandType*을 사용하지 않을 때도 수행할 수 있습니다.
- *TADODataSet*을 쿼리 타입 데이터셋처럼 사용하여 데이터셋을 열 때 실행되는 단일 SQL 명령을 지정할 수 있습니다. 이러한 방법으로 사용하려면 *CommandType* 속성을 *cmdText* 로 설정하고 *CommandText* 속성을 실행할 SQL 명령으로 설정하십시오. 디자인 타임에 Object Inspector 에서 *CommandText* 속성을 더블 클릭하고 Command Text Editor 를 사용하여 SQL 명령을 작성할 수 있습니다. *TADODataSet*은 다음과 같은 쿼리 타입 작업을 지원합니다.
  - 쿼리 텍스트에 매개변수 사용. 쿼리 매개변수에 대한 자세한 내용은 22-43페이지의 "쿼리 매개변수 사용"을 참조하십시오.
  - 매개변수를 사용하여 마스터/디테일 관계 설정. 자세한 내용은 22-45페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.
  - *Prepared* 속성을 **true**로 설정하여 성능 향상을 위해 쿼리 사전 준비

- *TADODataSet*을 내장 프로시저 타입 데이터셋처럼 사용하여 데이터셋을 열 때 실행되는 내장 프로시저를 지정할 수 있습니다. 이러한 방법으로 사용하려면 *CommandType* 속성을 *cmdStoredProc*로 설정하고 *CommandText* 속성을 내장 프로시저의 이름으로 설정하십시오. *TADODataSet*은 다음과 같은 내장 프로시저 타입 작업을 지원합니다.
  - 내장 프로시저 매개변수 작업. 내장 프로시저 매개변수에 대한 자세한 내용은 22-49페이지의 "내장 프로시저 매개변수 사용"을 참조하십시오.
  - 여러 개의 결과 집합 가져오기. 이에 대한 자세한 내용은 22-53페이지의 "여러 결과 집합 가져오기"를 참조하십시오.
  - *Prepared* 속성을 **true**로 설정하여 성능 향상을 위해 내장 프로시저 준비

그 외에 *TADODataSet*을 사용하면 *CommandType* 속성을 *cmdFile*로 설정하고 *CommandText* 속성을 파일 이름으로 설정하여 파일에 저장된 데이터를 사용하여 작업할 수 있습니다.

CommandText 및 *CommandType* 속성을 설정하기 전에 *Connection* 또는 *ConnectionString* 속성을 설정하여 *TADODataSet*을 데이터 저장소에 연결해야 합니다. 이 과정은 25-9페이지의 "데이터 저장소에 ADO 데이터셋 연결"에 설명되어 있습니다. RDS DataSpace 객체를 사용하여 *TADODataSet*을 ADO 기반 애플리케이션 서버에 연결하는 방법도 있습니다. RDS DataSpace 객체를 사용하려면 *RDSConnection* 속성을 *TRDSConnection* 객체로 설정하십시오.

명령 객체 사용

ADO 환경에서 명령은 프로바이더에 특정적인 작업 요청을 텍스트로 나타낸 것입니다. 일반적으로 명령은 데이터 정의 언어(DDL) 및 데이터 조작 언어(DML) SQL 문입니다. 명령에 사용되는 랭귀지는 프로바이더별로 다르지만 대개 SQL 랭귀지의 SQL-92 표준을 준수합니다.

*TADOQuery*를 사용하면 명령을 실행할 수 있습니다. 그러나 데이터셋 컴포넌트를 사용할 때 발생하는 오버헤드를 원하지 않는 경우, 특히 결과 집합을 반환하지 않는 명령의 경우에는 명령을 한 번에 하나씩 실행하도록 디자인한 *TADOCommand* 컴포넌트를 사용하는 방법이 있습니다. *TADOCommand*는 기본적으로 DDL SQL 문과 같이 결과 집합을 반환하지 않는 명령을 실행하기 위한 것입니다. 그러나 *Execute* 메소드의 오버로드된 버전을 통해 ADO 데이터셋 컴포넌트의 *RecordSet* 속성으로 지정될 수 있는 결과 집합을 반환할 수 있습니다.

일반적으로 *TADOCommand*를 사용하는 것은 *TADODataSet*을 사용하는 것과 비슷합니다. 다만 표준 데이터셋 메소드를 사용하여 데이터 페치, 레코드 탐색, 데이터 편집 등의 작업을 할 수 없습니다. *TADOCommand* 객체는 ADO 데이터셋과 같은 방법으로 데이터 저장소에 연결됩니다. 자세한 내용은 25-9페이지의 "데이터 저장소에 ADO 데이터셋 연결"을 참조하십시오.

다음 부분에서는 *TADOCommand*를 사용하여 명령을 지정하고 실행하는 방법을 자세히 설명합니다.

명령 지정

TADOCommand 컴포넌트를 위한 명령을 지정하려면 *CommandText* 속성을 사용합니다. *TADODataSet*과 마찬가지로, *TADOCommand*를 사용하여 *CommandType* 속성에 따라 여러 가지 방법으로 명령을 지정할 수 있습니다. *CommandType* 속성 값은 *cmdText*(명령이 SQL 문인 경우), *cmdTable*(테이블 이름인 경우), 및 *cmdStoredProc*(내장 프로시저 이름인 경우) 중 하나일 수 있습니다. 디자인 타임에 Object Inspector의 리스트에서 해당 명령 타입을 선택하고 런타임에 *TCommandType* 타입 값을 *CommandType* 속성으로 지정하십시오.

```
ADOCommand1->CommandText = "AddEmployee";
ADOCommand1->CommandType = cmdStoredProc;
...
```

특정 타입이 지정되어 있지 않은 경우에는 서버에서 *CommandText*의 명령을 기반으로 가장 적합한 것을 결정합니다.

*CommandText*는 매개변수를 포함하는 SQL 쿼리 텍스트 또는 매개변수를 사용하는 내장 프로시저의 이름을 포함할 수 있습니다. 그런 다음 명령을 실행하기 전에 매개변수 값을 제공해야 합니다. 자세한 내용은 25-18페이지의 "명령 매개변수 처리"를 참조하십시오.

Execute 메소드 사용

*TADOCommand*가 데이터 저장소에 연결되어 있어야 명령을 실행할 수 있습니다. 연결 방법은 ADO 데이터셋을 데이터 저장소에 연결하는 것과 같습니다. 자세한 내용은 25-9페이지의 "데이터 저장소에 ADO 데이터셋 연결"을 참조하십시오.

명령을 실행하려면 *Execute* 메소드를 호출하십시오. *Execute* 메소드는 오버로드된 메소드로서 이 메소드를 사용하면 명령을 실행하는 가장 적합한 방법을 선택할 수 있습니다.

매개변수가 필요 없고 영향받은 레코드 수를 알 필요가 없는 명령인 경우에는 *Execute*를 매개변수 없이 호출하십시오.

```
ADOCommand1->CommandText = "UpdateInventory";
ADOCommand1->CommandType = cmdStoredProc;
ADOCommand1->Execute();
```

다른 *Execute* 버전을 사용하면 Variant 배열을 사용하여 매개변수 값을 제공할 수 있고 명령에 의해 영향을 받는 레코드의 수를 알 수 있습니다.

결과 집합을 반환하는 명령 실행에 대한 내용은 25-18페이지의 "명령을 사용하여 결과 집합 검색"을 참조하십시오.

명령 취소

명령을 비동기로 실행하는 경우 *Execute*를 호출한 후 *Cancel* 메소드를 호출하면 명령 실행을 중단할 수 있습니다.

```
void __fastcall TDataForm::ExecuteButtonClick(TObject *Sender)
{
  ADOCommand1->Execute();
}
void __fastcall TDataForm::CancelButtonClick(TObject *Sender)
{
  ADOCommand1->Cancel();
}
```

Cancel 메소드는 보류 중인 명령이 있고 그 명령이 비동기로 실행되었던 경우에만 영향을 미칩니다(*eoAsynchExecute*는 *Execute* 메소드의 *ExecuteOptions* 매개변수에 있습니다). 명령을 보류 중이라는 말은 *Execute* 메소드가 호출되었지만 명령이 아직 완료되지 않았거나 시간 제한이 초과된 경우에 사용합니다.

CommandTimeout 속성에 지정된 시간(초)이 만료되기 전에 명령이 완료되거나 취소되지 않은 경우 그 명령은 시간 제한을 초과한 것입니다. 디폴트로 명령 시간 제한은 30초입니다.

명령을 사용하여 결과 집합 검색

결과 집합 반환 여부에 따라 다른 실행 메소드를 사용하는 *TADOQuery* 컴포넌트와 달리, *TADOCommand*는 결과 집합 반환 여부에 관계 없이 항상 *Execute* 명령을 사용하여 명령을 실행합니다. 명령이 결과 집합을 반환할 때 *Execute* 메소드는 인터페이스를 ADO _RecordSet 인터페이스로 반환합니다.

이 인터페이스를 사용하는 가장 편리한 방법은 인터페이스를 ADO 데이터셋의 *RecordSet* 속성으로 지정하는 것입니다.

예를 들어, 다음 코드는 *TADOCommand*인 *ADOCommand1*를 사용하여 결과 집합을 반환하는 SELECT 쿼리를 실행합니다. 이 결과 집합은 *TADODataSet* 컴포넌트인 *ADODataSet1*의 *RecordSet* 속성으로 지정됩니다.

```
ADOCommand1->CommandText = "SELECT Company, State ";
ADOCommand1->CommandText += "FROM customer ";
ADOCommand1->CommandText += "WHERE State = :StateParam";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("StateParam")->Value = "HI";
ADOCommand1->Recordset = ADOCommand1->Execute();
```

결과 집합이 ADO 데이터셋의 *Recordset* 속성으로 지정되면 곧 데이터셋이 자동으로 활성화되고 데이터를 사용할 수 있습니다.

명령 매개변수 처리

TADOCommand 객체에서 매개변수를 사용하는 방법은 다음 두 가지가 있습니다.

- *CommandText* 속성을 사용하여 매개변수를 포함하는 쿼리를 지정할 수 있습니다. *TADOCommand*에서 매개변수화된 쿼리를 사용하는 것은 ADO 데이터셋에서 매개변수화된 쿼리를 사용하는 것과 비슷합니다. 매개변수화된 쿼리에 대한 자세한 내용은 22-43페이지의 "쿼리 매개변수 사용"을 참조하십시오.

- *CommandText* 속성은 매개변수를 사용하는 내장 프로시저를 지정할 수 있습니다. 내장 프로시저 매개변수는 *TADOCommand*와 ADO 데이터셋에서 거의 같은 방식으로 사용됩니다. 내장 프로시저 매개변수에 대한 자세한 내용은 22-49페이지의 "내장 프로시저 매개변수 사용"을 참조하십시오.

*TADOCommand*를 사용할 때 매개변수 값을 제공하는 방법은 두 가지가 있습니다. *Execute* 메소드를 호출할 때 제공하거나 *Parameters* 속성을 사용하여 사전에 지정할 수 있습니다.

Execute 메소드는 일련의 매개변수 값을 Variant 배열로 취하는 버전을 포함하도록 오버로드됩니다. 이는 *Parameters* 속성을 설정하는 오버헤드 없이 매개변수 값을 빠르게 제공하고자 할 때 유용합니다.

```
Variant Values[2];
Values[0] = Edit1->Text;
Values[1] = Date();
ADOCommand1.Execute(VarArrayOf(Values,1));
```

출력 매개변수를 반환하는 내장 프로시저를 사용할 때는 *Parameters* 속성을 사용해야 합니다. 출력 매개변수를 읽을 필요가 없는 경우라도 *Parameters* 속성을 사용하면 디자인 타임에 매개변수를 제공할 수 있으며 데이터셋에서 매개변수를 사용하는 것과 같은 방법으로 *TADOCommand* 속성을 사용할 수 있습니다.

CommandText 속성을 설정하면 *Parameters* 속성이 자동으로 업데이트되어 쿼리에 있는 매개변수 또는 내장 프로시저에서 사용된 매개변수가 반영됩니다. 디자인 타임에 Object Inspector에서 *Parameters* 속성의 생략 부호 버튼을 클릭하면 Parameter Editor를 사용하여 매개변수에 액세스할 수 있습니다. 런타임에는 *TParameter*의 속성과 메소드를 사용하여 각 매개변수의 값을 설정하거나 가져올 수 있습니다.

```
ADOCommand1->CommandText = "INSERT INTO Talley ";
ADOCommand1->CommandText += "(Counter) ";
ADOCommand1->CommandText += "VALUES (:NewValueParam)";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("NewValueParam")->Value = 57;
ADOCommand1->Execute()
```

장

26

단방향 데이터셋 사용

*dbExpress*는 SQL 데이터베이스 서버에 대한 빠른 액세스를 제공하는 lightweight 데이터베이스 드라이버 집합입니다. 각 지원 데이터베이스에 대해 *dbExpress*는 일정한 *dbExpress* 클래스 집합에 적합한 서버 특정 소프트웨어를 사용하는 드라이버를 제공합니다. *dbExpress*를 사용하는 데이터베이스 애플리케이션을 배포할 때 직접 생성하는 애플리케이션과 dll(서버 특정 드라이버)만 포함해야 합니다.

*dbExpress*를 사용하면 단방향 데이터셋을 사용하여 데이터베이스에 액세스할 수 있습니다. 단방향 데이터셋은 최소한의 오버헤드를 가지고 데이터베이스 정보에 빠르게 lightweight 액세스를 하도록 디자인되었습니다. 다른 데이터셋과 달리 단방향 데이터셋은 SQL 명령을 데이터베이스 서버에 보낼 수 있으며, 명령이 레코드 집합을 반환하면 해당 레코드에 액세스하기 위한 단방향 커서가 생깁니다. 그러나 단방향 데이터셋은 단방향 커서만 가져옵니다. 이 데이터셋은 데이터를 메모리에 버퍼로 저장하지 않으며, 다른 타입의 데이터셋에 비해 실행 속도가 더 빠르고 리소스를 덜 차지합니다. 그러나 레코드가 버퍼로 저장되지 않기 때문에 단방향 데이터셋은 다른 데이터셋에 비해 유연하지 않습니다. *TDataSet* 에 도입된 대부분의 기능은 단방향 데이터셋에서 구현되지 않거나, 단방향 데이터셋에 예외를 발생시킵니다. 예를 들면, 다음과 같습니다.

- 유일하게 지원되는 탐색 메소드는 *First* 및*Next* 메소드입니다. 대부분의 다른 메소드는 예외를 발생시킵니다. 북마크 지원과 관련된 메소드 같은 일부 메소드는 아무런 기능도 하지 않습니다.
- 편집 기능은 편집 내용을 보유하는 버퍼가 필요하므로 여기에는 편집에 대한 기본 지원이 없습니다. *CanModify* 속성은 항상 **false**이므로 데이터셋을 편집 모드로 설정하려는 시도는 항상 실패합니다. 그러나 단방향 데이터셋을 사용하여 SQL UPDATE 명령을 통해 데이터를 업데이트하거나, dbExpress 사용 클라이언트 데이터셋을 사용하거나 데이터셋을 클라이언트 데이터셋에 연결하여 일반적인 편집 지원을 제공할 수 있습니다(18-10페이지의 "다른 데이터셋 연결" 참조).
- 필터는 버퍼링을 필요로 하는 여러 레코드와 함께 작동하기 때문에 여기에는 필터에 대한 지원이 없습니다. 단방향 데이터셋을 필터링하려고 시도하면 예외가 발생합니다. 그 대신 데이터 표시에 대한 모든 제한 사항은 데이터셋의 데이터를 정의하는 SQL 명령을 사용하여 부과되어야 합니다.

- 조회 필드는 조회 값이 들어 있는 여러 레코드를 보유하기 위해 버퍼링을 필요로 하기 때문에 조회 필드에 대한 지원이 없습니다. 단방향 데이터셋에서 조회 필드를 정의하는 경우 필드가 올바르게 작동하지 않습니다.

이러한 제한 사항에도 불구하고, 단방향 데이터셋은 데이터에 액세스하는 확실한 방법입니다. 단방향 데이터셋은 가장 빠른 데이터 액세스 메커니즘이며, 사용과 배포가 매우 간단합니다.

단방향 데이터셋의 타입

컴포넌트 팔레트의 *dbExpress* 페이지에는 네 가지 단방향 데이터셋 타입인 *TSQLDataSet*, *TSQLQuery*, *TSQLTable* 및 *TSQLStoredProc*이 포함되어 있습니다.

*TSQLDataSet*은 네 가지 타입 중에서 가장 일반적인 타입입니다. SQL 데이터셋을 사용하면 *dbExpress*를 통해 사용 가능한 데이터를 나타내거나, *dbExpress*를 통해 액세스된 데이터베이스에 명령을 보낼 수 있습니다. 데이터베이스 애플리케이션의 데이터베이스 테이블로 작업하는 경우에는 이 컴포넌트를 사용하는 것이 좋습니다.

*TSQLQuery*는 SQL 문을 캡슐화하고, 애플리케이션이 결과 레코드(있을 경우)에 액세스할 수 있게 하는 쿼리 타입 데이터셋입니다. 쿼리 타입 데이터셋 사용에 대한 자세한 내용은 22-40페이지의 "쿼리 타입 데이터셋 사용"을 참조하십시오.

*TSQLTable*은 단일 데이터베이스 테이블의 모든 행과 열을 나타내는 테이블 타입 데이터셋입니다. 테이블 타입 데이터셋 사용에 대한 자세한 내용은 22-24페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.

*TSQLStoredProc*는 데이터베이스 서버에서 정의된 내장 프로시저를 실행하는 내장 프로시저 타입의 데이터셋입니다. 내장 프로시저 타입의 데이터셋 사용에 대한 자세한 내용은 22-48페이지의 "내장 프로시저 타입의 데이터셋 사용"을 참조하십시오.

참고 또한 *dbExpress* 페이지에는 단방향 데이터셋이 아닌 *TSQLClientDataSet*도 포함되어 있습니다. 그러나 이 데이터셋은 내부적으로 단방향 데이터셋을 사용하여 해당 데이터에 액세스하는 클라이언트 데이터셋입니다.

데이터베이스 서버에 연결

단방향 데이터셋을 사용하는 첫 번째 단계는 단방향 데이터셋을 데이터베이스 서버에 연결하는 것입니다. 디자인 타임에 데이터셋에 데이터베이스 서버에 대한 활성화된 연결이 있으면 Object Inspector가 다른 속성에 대한 값의 드롭다운 리스트를 제공할 수 있습니다. 예를 들어, 내장 프로시저를 나타낼 때는 활성화된 연결이 있어야 Object Inspector 가 서버에서 사용할 수 있는 내장 프로시저 리스트를 나타낼 수 있습니다.

각각의 *TSQLConnection* 컴포넌트는 데이터베이스 서버에 대한 연결을 나타냅니다. 다른 데이터베이스 연결 컴포넌트와 마찬가지로 *TSQLConnection*을 사용할 수 있습니다. 데이터베이스 연결 컴포넌트에 대한 자세한 내용은 21장, "데이터베이스에 연결"을 참조하십시오.

*TSQLConnection*을 사용하여 단방향 데이터셋을 데이터베이스 서버에 연결하려면 *SQLConnection* 속성을 설정하십시오. 디자인 타임에 Object Inspector의 드롭다운 리스트에서 SQL 연결 컴포넌트를 선택할 수 있습니다. 런타임에 이러한 할당을 지정하는 경우에는 연결이 활성화되어 있는지 확인하십시오.

```
SQLDataSet1->SQLConnection = SQLConnection1;
SQLConnection1->Connected = true;
```

일반적으로 여러 데이터베이스 서버의 데이터를 사용하지 않으면 애플리케이션에 있는 모든 단방향 데이터셋이 같은 연결 컴포넌트를 공유합니다. 그러나 서버에서 연결마다 여러 명령문을 지원하지 않는 경우 데이터셋에 각각의 연결을 사용하려는 경우가 있습니다. 이런 경우, *MaxStmtsPerConn* 속성을 읽어서 데이터베이스 서버가 데이터셋에 대한 각각의 연결을 필요로 하는지 확인합니다. 디폴트로, *TSQLConnection*은 서버에서 연결을 통해 실행할 수 있는 명령문의 수를 제한하는 경우 필요에 따라 연결을 생성합니다. 사용 중인 연결을 더 자세히 추적하려면 *AutoClone* 속성을 **false**로 설정하십시오.

SQLConnection 속성을 지정하기 전에 *TSQLConnection* 컴포넌트를 설정하여 이 컴포넌트가 데이터베이스 서버 및 필요한 연결 매개변수(서버에서 사용할 데이터베이스, 서버에서 실행 중인 컴퓨터의 호스트 이름, 사용자 이름, 암호 등을 포함)를 식별하게 해야 합니다.

TSQLConnection 설정

*TSQLConnection*이 연결을 열 수 있도록 데이터베이스 연결에 대해 충분히 자세하게 설명하려면 사용할 드라이버와 해당 드라이버로 전달되는 연결 매개변수 집합을 둘 다 식별해야 합니다.

드라이버 식별

INTERBASE, ORACLE, MYSQL, DB2 등 설치된 *dbExpress* 드라이버의 이름인 *DriverName* 속성이 드라이버를 식별합니다. 드라이버 이름은 다음 두 파일과 관련이 있습니다.

- *dbExpress* 드라이버. 이 드라이버는 dbexpint.dll, dbexpora.dll, dbexpmys.dll 또는 dbexpdb2.dll과 같은 이름을 가진 동적 연결 라이브러리입니다.
- 데이터베이스 업체가 클라이언트사이드 지원을 위해 제공하는 동적 연결 라이브러리

이러한 두 파일 간의 관계와 데이터베이스 이름은 dbxdrivers.ini라는 파일에 저장됩니다. 이 파일은 *dbExpress* 드라이버를 설치할 때 업데이트됩니다. 일반적으로 SQL 연결 컴포넌트는 *DriverName*의 값이 지정될 때 dbxdrivers.ini에서 이러한 파일을 확인하므로 특별히 염려할 필요는 없습니다. *DriverName* 속성을 설정하면 *TSQLConnection*이 *LibraryName* 및 *VendorLib* 속성을 관련된 dll의 이름으로 자동으로 설정합니다. *LibraryName*과 *VendorLib*가 설정되면 애플리케이션이 dbxdrivers.ini에 의존할 필요가 없습니다. (즉, 런타임에 *DriverName* 속성을 설정하는 경우가 아니라면 애플리케이션과 함께 dbxdrivers.ini를 배포할 필요가 없습니다.)

연결 매개변수 지정

Params 속성은 이름/값 쌍을 나열하는 문자열 리스트입니다. 각 쌍에는 *Name*=*Value*과 같은 형태가 있습니다. 여기서 *Name*은 매개변수 이름이고, *Value*는 할당할 값입니다.

사용 중인 데이터베이스 서버에 따라 필요한 특정 매개변수가 다릅니다. 그러나 특정 매개변수 *Database*는 모든 서버에 필요합니다. 이 매개변수 값은 사용하는 서버에 따라 다릅니다. 예를 들어, InterBase의 경우 *Database*는 .gdb 파일의 이름이고, ORACLE에서는 TNSNames.ora에 있는 항목이고, DB2에서는 클라이언트사이드 노드 이름입니다.

다른 일반적인 매개변수에는 *User_Name*(로그인할 때 사용할 이름), *Password*(*User_Name*에 대한 암호), *HostName*(서버가 위치한 컴퓨터 이름 또는 IP 주소) 및 *TransIsolation*(사용할 트랜잭션이 다른 트랜잭션에 의한 변경을 인식하는 정도)이 있습니다. 드라이버 이름을 지정할 때 *Params* 속성이 해당 드라이버 타입에 필요한 모든 매개변수를 사용하여 기본값으로 초기화되어 미리 로드됩니다.

*Params*는 문자열 리스트이므로 디자인 타임에 Object Inspector에서 *Params* 속성을 더블 클릭하면 String List Editor 를 사용하여 매개변수를 편집할 수 있습니다 . 런타임에는 *Params*::*Values* 속성을 사용하여 개별 매개변수에 값을 할당합니다.

연결 이름 지정 설명

항상 *DatabaseName* 및 *Params* 속성만 사용하여 연결을 지정할 수 있지만, 특정 조합의 이름을 지정한 다음 이름으로 연결을 식별하는 것이 더 편리합니다. *dbExpress* 데이터베이스와 매개변수 조합의 이름을 지정할 수 있습니다. 그러면 dbxconnections.ini 파일에 이름이 저장됩니다. 각 조합의 이름은 연결 이름입니다.

연결 이름을 정의한 다음 *ConnectionName* 속성을 유효한 연결 이름으로 설정하기만 하면 데이터베이스 연결을 식별할 수 있습니다. *ConnectionName*을 설정하면 *DriverName* 및 *Params* 속성이 자동으로 설정됩니다. *ConnectionName*이 설정되면 *Params* 속성을 편집하여 저장된 매개변수 값 집합과 임시로 다르게 할 수 있습니다. 그러나 *DriverName* 속성을 변경하면 *Params*와 *ConnectionName*이 모두 지워집니다.

Local InterBase와 같은 하나의 데이터베이스를 사용하여 애플리케이션을 개발하는 경우 연결 이름을 사용할 수 있다는 장점이 있지만, ORACLE과 같은 다른 데이터베이스와 함께 사용할 수 있도록 애플리케이션을 배포합니다. 이 경우 애플리케이션을 배포하는 시스템에서 *DriverName* 및 *Params*가 개발 도중 사용하는 값과 다를 수 있습니다. dbxconnections.ini 파일 버전 두 개를 사용하여 두 연결 설명 사이에서 쉽게 전환할 수 있습니다. 디자인 타임에 애플리케이션이 dbxconnections.ini의 디자인 타임 버전으로부터 *DriverName*과 *Params*를 로드합니다. 그러면 애플리케이션을 배포할 때 애플리케이션이 "실제" 데이터베이스를 사용하는 각각의 dbxconnections.ini 버전으로부터 이 값을 로드합니다. 하지만 그러기 위해서는 런타임에 *DriverName*과 *Params* 속성을 다시 로드하도록 연결 컴포넌트에 지시해야 합니다. 이 작업을 수행하는 방법은 두 가지가 있습니다.

- *LoadParamsOnConnect* 속성을 **true**로 설정합니다. 그러면 연결이 열릴 때 *TSQLConnection*이 자동으로 *DriverName*과 *Params*를 dbxconnections.ini의 *ConnectionName*으로 설정합니다.
- *LoadParamsFromIniFile* 메소드를 호출합니다. 이 메소드는 *DriverName*과 *Params*를 dbxconnections.ini(또는 지정하는 다른 파일)의 *ConnectionName*으로 설정합니다. 연결을 열기 전에 특정 매개변수 값을 무시하려면 이 메소드를 사용하도록 선택할 수 있습니다.

Connection Editor 사용

연결 이름 및 연결된 드라이버와 연결 매개변수의 관계는 dbxconnections.ini 파일에 저장됩니다. Connection Editor를 사용하여 이러한 연결을 만들거나 수정할 수 있습니다.

Connection Editor를 표시하려면 *TSQLConnection* 컴포넌트를 더블 클릭하십시오. 사용 가능한 모든 드라이버를 포함하는 드롭다운 리스트, 현재 선택된 드라이버에 대한 연결 이름 리스트, 현재 선택된 연결 이름의 연결 매개변수를 나열하는 테이블 등이 있는 Connection Editor가 나타납니다.

이 다이얼로그 박스에서 드라이버와 연결 이름을 선택하여 사용할 연결을 지정할 수 있습니다. 원하는 구성을 선택한 다음 Test Connection 버튼을 클릭하여 선택한 구성이 올바른지 확인합니다.

또한 이 다이얼로그 박스를 사용하여 dbxconnections.ini에서 명명된 연결을 편집할 수 있습니다.

- 매개변수 테이블에 있는 매개변수 값을 편집하여 현재 선택된 명명된 연결을 변경합니다. OK를 클릭하여 다이얼로그 박스를 종료할 때 새 매개변수 값이 dbxconnections.ini에 저장됩니다.
- Add Connection 버튼을 클릭하여 새 명명된 연결을 정의합니다. 사용할 드라이버와 새 연결 이름을 지정하는 다이얼로그 박스가 나타납니다. 연결 이름이 지정되면 매개변수를 편집하여 필요한 연결을 지정하고, OK 버튼을 클릭하여 dbxconnections.ini에 새 연결을 저장합니다.
- Delete Connection 버튼을 클릭하여 현재 선택된 명명된 연결을 dbxconnections.ini에서 삭제합니다.
- Rename Connection 버튼을 클릭하여 현재 선택된 명명된 연결의 이름을 변경합니다. 매개변수의 편집 내용은 OK 버튼을 클릭할 때 저장됩니다.

표시할 데이터 지정

단방향 데이터셋이 나타낼 데이터를 지정하는 방법은 여러 가지입니다. 사용하고 있는 단방향 데이터셋의 타입에 따라 사용할 방법이 달라지며, 정보를 단일 데이터베이스 테이블, 쿼리의 결과 또는 내장 프로시저 중 어디에서 가져 오는지에 따라 방법이 다릅니다.

TSQLDataSet 컴포넌트를 사용할 때 *CommandType* 속성을 사용하여 데이터셋이 데이터를 가져오는 곳을 표시합니다. *CommandType*은 다음 값 중 하나를 가져올 수 있습니다.

- *ctQuery*:*CommandType*이 *ctQuery*이면, *TSQLDataSet*이 사용자가 지정하는 쿼리를 실행합니다. 쿼리가 SELECT 명령이면 데이터셋은 레코드의 결과 집합을 포함합니다.
- *ctTable*:*CommandType*이 *ctTable*이면 *TSQLDataSet*이 지정된 테이블의 모든 레코드를 검색합니다.
- *ctStoredProc*:*CommandType*이 *ctStoredProc*이면, *TSQLDataSet*이 내장 프로시저를 실행합니다. 내장 프로시저가 커서를 반환하면 데이터셋이 반환된 레코드를 포함합니다.

참고 서버에서 이용할 수 있는 메타데이터가 단방향 데이터셋에 있을 수도 있습니다. 이 방법에 대한 자세한 내용은 26-12페이지의 "메타데이터를 단방향 데이터셋으로 가져오기"를 참조하십시오.

쿼리의 결과 표시

쿼리 사용은 레코드 집합을 지정하는 가장 일반적인 방법입니다. 쿼리는 단순히 SQL로 작성된 명령입니다. *TSQLDataSet* 또는 *TSQLQuery*를 사용하여 쿼리 결과를 나타낼 수 있습니다.

*TSQLDataSet*을 사용할 때 *CommandType* 속성을 *ctQuery*로 설정하고, 쿼리 문의 텍스트를 *CommandText* 속성에 할당합니다. *TSQLQuery*를 사용할 때는 대신 쿼리를 *SQL* 속성에 할당합니다. 이러한 속성은 모든 일반적인 용도나 쿼리 타입 데이터셋에 대해 같은 방식으로 작용합니다. 22-41페이지의 "쿼리 지정"에서 이러한 속성을 보다 자세히 설명합니다.

쿼리를 지정할 때 쿼리에는 디자인 타임이나 런타임에 해당 값이 달라질 수 있는 매개변수나 변수가 포함될 수 있습니다. 매개변수는 SQL 문에 나타나는 데이터 값을 대체할 수 있습니다. 쿼리에 매개변수를 사용하고 해당 매개변수에 값을 제공하는 데 대한 내용은 22-43페이지의 "쿼리 매개변수 사용"에서 다룹니다.

SQL은 서버에서 동작을 수행하지만 레코드를 반환하지는 않는 UPDATE 쿼리 등의 쿼리를 정의합니다. 이러한 쿼리는 26-9페이지의 "레코드를 반환하지 않는 명령 실행"에서 다룹니다.

테이블의 레코드 표시

원본으로 사용하는 데이터베이스 테이블 하나에 모든 필드와 모든 레코드를 표시하려는 경우, 직접 SQL 문을 작성하는 것보다 *TSQLDataSet* 또는 *TSQLTable* 을 사용하여 쿼리를 생성할 수 있습니다.

참고 서버 성능이 문제가 될 경우에는, 자동 생성되는 쿼리에 의존하기보다 명시적으로 쿼리를 작성할 수도 있습니다. 자동으로 생성되는 쿼리는 테이블에 있는 필드를 명시적으로 나열하지 않고 와일드카드를 사용합니다. 그러면 서버 성능이 약간 저하될 수 있습니다. 자동으로 생성되는 쿼리의 와일드카드(*)는 서버의 필드 변경 사항에 대해 좀 더 강력합니다.

TSQLDataSet을 사용하여 테이블 표시

*TSQLDataSet*이 단일 데이터베이스 테이블의 모든 필드와 모든 레코드를 가져오는 쿼리를 생성하도록 하려면 *CommandType* 속성을 *ctTable*로 설정합니다.

*CommandType*이 *ctTable*이면, *TSQLDataSet*이 두 속성 값을 기반으로 쿼리를 생성합니다.

- *CommandText*는 *TSQLDataSet* 객체가 나타낼 데이터셋 테이블의 이름을 지정합니다.
- *SortFieldNames*는 데이터 정렬에 사용할 필드의 이름을 중요한 순서에 따라 나열합니다.

예를 들어, 다음과 같이 지정한다고 가정해 보십시오.

```
SQLDataSet1->CommandType = ctTable;
SQLDataSet1->CommandText = "Employee";
SQLDataSet1->SortFieldNames = "HireDate,Salary"
```

*TSQLDataSet*이 Employee 테이블에서 HireDate로 정렬하고 그 안에서 Salary로 정렬하여 모든 레코드를 나열하는 다음과 같은 쿼리를 생성합니다.

```
select * from Employee order by HireDate, Salary
```

TSQLTable을 사용하여 테이블 표시

*TSQLTable*을 사용할 때 *TableName* 속성을 사용하여 원하는 테이블을 지정합니다.

데이터셋의 필드 순서를 지정하려면 인덱스를 지정해야 합니다. 이를 수행하는 방법은 두 가지가 있습니다.

- 원하는 순서를 부여할 서버에 정의된 인덱스의 이름으로 *IndexName* 속성을 설정합니다.
- *IndexFieldNames* 속성을 정렬할 필드 이름이 세미콜론으로 구분된 리스트에 설정합니다. *IndexFieldNames*는 쉼표가 아닌 세미콜론을 구분자로 사용한다는 점만 제외하면, *TSQLDataSet*의 *SortFieldNames* 속성과 동일한 기능을 합니다.

내장 프로시저의 결과 표시

내장 프로시저는 SQL 서버에 이름이 지정되고 저장된 SQL 문의 집합입니다. 실행하려는 내장 프로시저를 나타내는 방법은 사용하고 있는 단방향 데이터셋의 타입에 따라 다릅니다.

*TSQLDataSet*을 사용할 때 내장 프로시저를 지정하려면 다음과 같이 합니다.

- *CommandType* 속성을 *ctStoredProc*로 설정합니다.
- 내장 프로시저 이름을 *CommandText* 속성의 값에 따라 지정합니다.

```
SQLDataSet1->CommandType = ctStoredProc;
SQLDataSet1->CommandText = "MyStoredProcName";
```

*TSQLStoredProc*를 사용할 때는 내장 프로시저 이름을 *StoredProcName* 속성의 값에 따라 지정하기만 하면 됩니다.

```
SQLStoredProc1->StoredProcName = "MyStoredProcName";
```

내장 프로시저를 식별한 다음 애플리케이션이 내장 프로시저의 입력 매개변수에 대한 값을 입력해야 하거나, 내장 프로시저를 실행한 다음 출력 매개변수의 값을 가져와야 할 수 있습니다. 내장 프로시저 매개변수 사용에 대한 자세한 내용은 22-49페이지의 "내장 프로시저 매개변수 사용"을 참조하십시오.

데이터 가져오기

데이터의 소스를 지정한 다음에는 애플리케이션이 액세스하기 전에 데이터를 가져와야 합니다. 데이터셋이 데이터를 가져오고 나면 데이터 소스를 통해 데이터셋에 연결된 데이터 인식 컨트롤이 자동으로 데이터 값을 표시하고, 프로바이더를 통해 데이터셋에 연결된 클라이언트 데이터셋을 레코드로 채울 수 있습니다.

다른 데이터셋에서와 같이, 단방향 데이터셋에 대해 데이터를 가져오는 방법에는 두 가지가 있습니다.

- 디자인 타임에 Object Inspector에서 또는 런타임에 코드에서 *Active* 속성을 **true**로 설정합니다.

```
CustQuery->Active = true;
```

- 런타임에 *Open* 메소드를 호출합니다.

```
CustQuery->Open();
```

서버에서 레코드를 얻는 단방향 데이터셋과 함께 *Active* 속성이나 *Open* 메소드를 사용합니다. 이러한 레코드가 SELECT 쿼리(*CommandType*이 *ctTable*일 때 자동으로 생성된 쿼리 포함) 또는 내장 프로시저 중 어디에서 비롯된 것인지는 중요하지 않습니다.

데이터셋 준비

서버에서 쿼리 또는 내장 프로시저를 실행하기 전에, 먼저 데이터셋을 "준비"해야 합니다. 데이터셋 준비는 *dbExpress*와 서버가 명령문 및 매개변수에 대한 리소스를 할당하는 것을 의미합니다. *CommandType*이 *ctTable*인 경우는 데이터셋이 SELECT 쿼리를 생성할 때입니다. 서버에 의해 연결되지 않은 모든 매개변수는 이 시점에서 쿼리에 포함됩니다.

단방향 데이터셋은 *Active*를 **true**로 설정하거나 *Open* 메소드를 호출할 때 자동으로 준비됩니다. 데이터셋을 닫으면 명령문 실행을 위해 할당된 리소스가 해제됩니다. 쿼리나 내장 프로시저를 두 번 이상 실행하려는 경우, 데이터셋을 처음 열기 전에 명시적으로 준비하면 성능을 향상시킬 수 있습니다 . 데이터셋을 명시적으로 준비하려면 데이터셋의 *Prepared* 속성을 **true** 로 설정하십시오.

```
CustQuery->Prepared = true;
```

데이터셋을 명시적으로 준비할 때 *Prepared*를 **false**로 설정하기 전에는 명령문 실행을 위해 할당된 리소스가 해제되지 않습니다.

데이터셋이 실행되기 전에 다시 준비되는지 확인하려면(예를 들어, 매개변수 값이나 *SortFieldNames* 속성을 변경하는 경우), *Prepared* 속성을 **false**로 설정하십시오.

여러 데이터셋 가져오기

일부 내장 프로시저는 여러 레코드 집합을 반환합니다. 사용자가 데이터셋을 열 때 데이터셋은 첫 번째 집합만 가져옵니다. 다른 레코드 집합에 액세스하려면 *NextRecordSet* 메소드를 호출하십시오.

```
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet(nRows);
```

*NextRecordSet*은 다음 레코드 집합에 대한 액세스를 제공하는 새로 생성된 *TCustomSQLDataSet* 컴포넌트를 반환합니다. 즉, *NextRecordSet*을 처음 호출할 때 두 번째 레코드 집합에 대한 데이터셋을 반환합니다. *NextRecordSet*을 호출하면 다시 세 번째 데이터셋을 반환하며, 이 과정은 더 이상 남은 레코드 집합이 없을 때까지 반복됩니다. 남은 데이터셋이 없으면 *NextRecordSet*이 Null을 반환합니다.

레코드를 반환하지 않는 명령 실행

단방향 데이터셋이 나타내는 쿼리 또는 내장 프로시저가 레코드를 반환하지 않는 경우에도 단방향 데이터셋을 사용할 수 있습니다. 이러한 명령은 SELECT 문이 아닌 데이터 정의 언어(DDL) 또는 데이터 조작 언어(DML) 문을 사용하는 명령문을 포함합니다. 예를 들어, INSERT, DELETE, UPDATE, CREATE INDEX 및 ALTER TABLE 명령은 레코드를 반환하지 않습니다. 명령에 사용되는 랭귀지는 서버 특정적이지만, 일반적으로 SQL 랭귀지의 SQL-92 표준과 호환됩니다.

실행하는 SQL 명령은 사용 중인 서버에서 수용 가능해야 합니다. 단방향 데이터셋은 SQL을 분석하거나 실행하지 않습니다. 단방향 데이터셋은 명령을 실행하도록 서버에 전달하기만 합니다.

참고 명령이 레코드를 반환하지 않을 경우 레코드 집합에 대한 액세스를 제공하는 데이터셋 메소드가 필요 없으므로 단방향 데이터셋을 사용할 필요가 전혀 없습니다. 데이터베이스 서버에 연결하는 SQL 연결 컴포넌트는 서버에서 명령을 실행하기 위해 직접 사용될 수 있습니다. 자세한 내용은 21-10페이지의 "서버에 명령 보내기"를 참조하십시오.

실행할 명령 지정

단방향 데이터셋에서 실행할 명령을 지정하는 방법은 명령이 데이터셋을 발생시키는지 여부에 관계 없이 동일합니다. 즉, 다음과 같습니다.

*TSQLDataSet*을 사용할 때 *CommandType*과 *CommandText* 속성을 사용하여 명령을 지정합니다.

- *CommandType*이 *ctQuery*이면, *CommandText*는 서버에 전달할 SQL 문입니다
- *CommandType*이 *ctStoredProc*이면, *CommandText*는 실행할 내장 프로시저의 이름입니다.

*TSQLQuery*를 사용할 때 *SQL* 속성을 사용하여 서버에 전달할 SQL 문을 지정합니다.

*TSQLStoredProc*를 사용할 때는 *StoredProcName* 속성을 사용하여 실행할 내장 프로시저 이름을 지정합니다.

레코드를 검색할 때 사용한 것과 같은 방법으로 명령을 지정하는 것처럼, 레코드를 반환하는 쿼리 및 내장 프로시저를 사용하는 것과 같은 방법으로 쿼리 매개변수나 내장 프로시저 매개변수를 사용합니다. 자세한 내용은 22-43페이지의 "쿼리 매개변수 사용" 및 22-49페이지의 "내장 프로시저 매개변수 사용"을 참조하십시오.

명령 실행

레코드를 반환하지 않는 쿼리나 내장 프로시저를 실행하려는 경우 *Active* 속성이나 *Open* 메소드를 사용하지 않습니다. 그 대신, 다음을 사용해야 합니다.

- 데이터셋이 *TSQLDataSet*이나 *TSQLQuery*의 인스턴스인 경우 *ExecSQL* 메소드를 사용합니다.

```
FixTicket->CommandText = "DELETE FROM TrafficViolations WHERE (TicketID =
1099)";
FixTicket->ExecSQL();
```

- 데이터셋이 *TSQLStoredProc*의 인스턴스인 경우 *ExecProc* 메소드를 사용합니다.

```
SQLStoredProc1->StoredProcName = "MyCommandWithNoResults";
```

팁 쿼리나 내장 프로시저를 여러 번 실행하는 경우 *Prepared* 속성을 **true**로 설정하는 것이 좋습니다.

SQLStoredProc1->ExecProc();

서버 메타데이터 작성 및 수정

데이터를 반환하지 않는 대부분의 명령은 두 범주로 나눌 수 있습니다.하나는 데이터를 편집하기 위해 사용하는 명령(예: INSERT, DELETE 및 UPDATE 명령)이고, 다른 하나는 테이블, 인덱스 및 내장 프로시저 같은 서버의 항목을 만들거나 수정하기 위해 사용하는 명령입니다.

편집을 위해 SQL 명령을 명시적으로 사용하지 않으려면 단방향 데이터셋을 클라이언트 데이터셋에 연결하고, 편집과 관련하여 생성된 모든 SQL 명령을 처리하게 할 수 있습니다(18-11페이지의 "클라이언트 데이터셋을 같은 애플리케이션의 다른 데이터셋에 연결" 참조). 사실 데이터 인식 컨트롤은 *TClientDataSet*과 같은 데이터셋에서 편집을 수행하도록 디자인되었으므로 이 방법을 사용하는 것이 좋습니다.

그러나 애플리케이션이 서버의 메타데이터를 작성 또는 수정할 수 있는 유일한 방법은 명령을 전송하는 것입니다. 모든 데이터베이스 드라이버가 동일한 SQL 구문을 지원하지는 않습니다. 각 데이터베이스 타입이 지원하는 SQL 구문과 데이터베이스 타입 사이의 차이점에 대해서는 여기서 설명하지 않습니다. 특정 데이터베이스 시스템의 SQL 구현에 대한 포괄적인 최신 정보는 해당 시스템과 함께 제공된 설명서를 참조하십시오.

일반적으로 CREATE TABLE 문을 사용하여 데이터베이스의 테이블을 만들고, CREATE INDEX 문을 사용하여 해당 테이블에 대한 새 인덱스를 만듭니다. 지원이 가능한 경우에는 여러 메타데이터 객체를 추가하기 위해 CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA 및 CREATE PROCEDURE 같은 다른 CREATE 문을 사용합니다.

각 CREATE 문마다 메타데이터 객체를 삭제하는 데 사용되는 해당 DROP 문이 있습니다. 이러한 명령문에는 DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA 및 DROP PROCEDURE가 있습니다.

테이블의 구조를 변경하려면 ALTER TABLE 문을 사용하십시오. ALTER TABLE은 테이블에 새로운 요소를 만들고 삭제할 수 있는 ADD 및 DROP 절이 있습니다. 예를 들어, ADD COLUMN 절을 사용하여 테이블에 새로운 열을 추가하고, DROP CONSTRAINT 절을 사용하여 테이블에서 기존의 제약 조건을 삭제합니다.

예를 들면, 다음 명령문은 InterBase 데이터베이스에 GET_EMP_PROJ라는 내장 프로시저를 만듭니다.

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

다음 코드는 *TSQLDataSet*을 사용하여 이 내장 프로시저를 만듭니다. 데이터셋이 내장 프로시저 정의(:EMP_NO 및 :PROJ_ID)의 매개변수를 내장 프로시저를 작성하는 쿼리의 매개변수와 혼동하지 않게 하기 위해 *ParamCheck* 속성을 사용한다는 점을 유의하십시오.

```
SQLDataSet1->ParamCheck = false;
SQLDataSet1->CommandType = ctQuery;
SQLDataSet1->CommandText = "CREATE PROCEDURE GET_EMP_PROJ (EMP_NO
SMALLINT) RETURNS (PROJ_ID CHAR(5)) AS BEGIN FOR SELECT PROJ_ID FROM
EMPLOYEE_PROJECT WHERE EMP_NO = :EMP_NO INTO :PROJ_ID DO SUSPEND; END";
SQLDataSet1->ExecSQL();
```

마스터/디테일 연결된 커서 설정

연결된 커서를 사용하여 디테일 셋에 따라 단방향 데이터셋으로 마스터/디테일 관계를 설정하는 두 가지 방법이 있습니다. 사용하고 있는 단방향 데이터셋의 타입에 따라 다른 방법을 사용합니다. 마스터/디테일 관계를 설정하면 단방향 데이터셋(일대다 관계의 "다")이 마스터 셋(일대다 관계의 "일")의 현재 레코드에 해당하는 레코드에 대해서만 액세스를 제공합니다.

*TSQLDataSet*과 *TSQLQuery*의 경우 매개변수화된 쿼리를 사용하여 마스터/디테일 관계를 설정해야 합니다. 이 기술은 모든 쿼리 타입 데이터셋에서 마스터/디테일 관계를 만드는 기술입니다. 쿼리 타입 데이터셋을 사용하여 마스터/디테일 관계를 만드는 데 대한 자세한 내용은 22-45페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.

디테일 셋이 *TSQLTable*의 인스턴스인 마스터/디테일 관계를 설정하려면 다른 테이블 타입 데이터셋을 사용하는 경우와 마찬가지로, *MasterSource* 및 *MasterFields* 속성을 사용하십시오. 테이블 타입 데이터셋을 사용하여 마스터/디테일 관계를 만드는 데 대한 자세한 내용은 22-45페이지의 "매개변수를 사용하여 마스터/디테일 관계 설정"을 참조하십시오.

스키마 정보 액세스

서버에서 사용 가능한 정보를 얻는 두 가지 방법이 있습니다. 스키마 정보 또는 메타데이터라고도 하는 이 정보에는 서버에서 사용 가능한 테이블 및 내장 프로시저에 대한 정보와 이러한 테이블 및 내장 프로시저에 대한 정보(예: 테이블에 있는 필드, 정의된 인덱스, 내장 프로시저가 사용하는 매개변수 등)가 포함되어 있습니다.

이 메타데이터를 사용하는 가장 간단한 방법은 *TSQLConnection*의 메소드를 사용하는 것입니다. 이 메소드는 기존의 문자열 리스트를 채우거나 테이블, 내장 프로시저, 필드 또는 인덱스의 이름을 사용하거나 매개변수 설명자를 사용하여 객체를 나열합니다. 이 기법은 다른 데이터베이스 연결 컴포넌트에 대한 메타데이터로 리스트를 채우는 방법과 동일합니다. 이 메소드는 21-12페이지의 "메타데이터 얻기"에 설명되어 있습니다.

보다 자세한 스키마 정보가 필요할 경우 메타데이터로 단방향 데이터셋을 채울 수 있습니다. 간단한 리스트 대신, 각 레코드가 단일 테이블, 내장 프로시저, 인덱스, 필드 또는 매개변수를 나타내는 스키마 정보로 단방향 데이터셋이 채워집니다.

메타데이터를 단방향 데이터셋으로 가져오기

데이터베이스 서버의 메타데이터로 단방향 데이터셋을 채우려면 먼저 *SetSchemaInfo* 메소드를 사용하여 나타낼 데이터를 표시합니다. *SetSchemaInfo*는 다음 세 가지 매개변수를 사용합니다.

- 가져올 스키마 정보(메타데이터)의 타입. 테이블 리스트(*stTables*), 시스템 테이블 리스트(*stSysTables*), 내장 프로시저 리스트(*stProcedures*), 테이블의 필드 리스트(*stColumns*), 인덱스 리스트(*stIndexes*) 또는 내장 프로시저에서 사용되는 매개변수 리스트(*stProcedureParams*)가 여기에 해당됩니다. 각각의 정보 타입은 다른 필드 집합을 사용하여 리스트의 항목을 설명합니다. 이 데이터셋의 구조에 대한 자세한 내용은 26-13페이지의 "메타데이터 데이터셋의 구조"를 참조하십시오.
- 필드, 인덱스 또는 내장 프로시저 매개변수에 대한 정보를 가져오는 경우 적용되는 테이블 또는 내장 프로시저의 이름. 다른 타입의 스키마 정보를 가져오는 경우 이 매개변수는 Null입니다.
- 반환되는 모든 이름에 대해 일치되어야 하는 패턴. 이 패턴은 와일드카드 '%'(모든 길이의 임의 문자의 문자열에 일치) 및 '_'(하나의 임의 문자에 일치)를 사용하는 'Cust%' 같은 SQL 패턴입니다. 패턴에서 퍼센트나 밑줄을 사용하려면 문자를 두 번 쓰십시오(%% 또는 __). 패턴을 사용하지 않으려면 매개변수가 Null이어도 됩니다.

참고 테이블(*stTables*)에 대한 스키마 정보를 가져오는 경우 결과 스키마 정보가 SQL 연결의 *TableScope* 속성 값에 따라 일반 테이블, 시스템 테이블, 뷰 및/또는 동의어를 기술할 수 있습니다.

다음의 호출은 모든 시스템 테이블(메타데이터를 포함하는 서버 테이블)을 나열하는 테이블을 요청합니다.

```
SQLDataSet1->SetSchemaInfo(stSysTable, "", "");
```

*SetSchemaInfo*에 대한 이 호출 다음에 데이터셋을 열면 결과 데이터셋에 테이블 이름, 타입, 스키마 이름 등을 제공하는 열이 있는 각 테이블에 대한 레코드가 포함됩니다. MySQL과 같이 서버가 메타데이터를 저장하기 위해 시스템 테이블을 사용하지 않는 경우에는, 데이터셋을 열 때 데이터셋에 레코드가 없습니다.

앞의 예제는 첫 번째 매개변수만 사용합니다. 그 대신 'MyProc'라는 내장 프로시저에 대한 입력 매개변수 리스트를 얻는 경우를 가정해 보십시오. 또한 이 내장 프로시저를 작성한 사람이 접두사를 사용해 모든 매개변수에 이름을 지정하여 입력 매개변수인지 또는 출력 매개변수인지 나타냅니다('inName', 'out Value' 등). 다음과 같이 *SetSchemaInfo*를 호출할 수 있습니다.

```
SQLDataSet1->SetSchemaInfo(stProcedureParams, "MyProc", "in%");
```

결과 데이터셋은 각 매개변수의 속성을 설명하는 열이 있는 입력 매개변수 테이블입니다.

메타데이터에 대한 데이터셋 사용 후 데이터 가져오기

SetSchemaInfo 호출 후에 데이터셋을 사용하여 쿼리나 내장 프로시저 실행을 반환하는 두 가지 방법이 있습니다.

- 데이터를 가져올 쿼리, 테이블 또는 내장 프로시저를 지정하여 *CommandText* 속성을 변경합니다.
- 첫 번째 매개변수를 *stNoSchema*로 설정하여 *SetSchemaInfo*를 호출합니다. 이 경우에 데이터셋은 *CommandText*의 현재 값에 의해 지정된 데이터 가져오기로 되돌아갑니다.

메타데이터 데이터셋의 구조

*TSQLDataSet*을 사용하여 액세스할 수 있는 각 메타데이터 타입에는, 요청된 타입의 항목에 대한 정보가 있는 미리 정의된 열(필드) 집합이 있습니다.

테이블에 대한 정보

테이블(*stTables* 또는 *stSysTables*)에 대한 정보를 요청하면 결과 데이터셋이 각 테이블에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 26.1 테이블을 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호
CATALOG_NAME	ftString	테이블이 들어 있는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 *Database* 매개변수와 동일합니다.
SCHEMA_NAME	ftString	테이블의 소유자를 식별하는 스키마 이름

표 26.1 테이블을 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
TABLE_NAME	ftString	테이블 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
TABLE_TYPE	ftInteger	테이블의 타입을 식별합니다. 이 값은 다음 값들 중에서 하나 이상의 합입니다. 1: 테이블 2: 뷰 4: 시스템 테이블 8: 동의어 16: 임시 테이블 32: 로컬 테이블

내장 프로시저에 대한 정보

내장 프로시저(*stProcedures*)에 대한 정보를 요청하면 결과 데이터셋이 각 내장 프로시저에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 26.2 내장 프로시저를 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호
CATALOG_NAME	ftString	내장 프로시저를 포함하는 카탈로그(데이터베이스) 이름. 이것은 SQL 연결 컴포넌트에 있는 *Database* 매개변수와 동일합니다.
SCHEMA_NAME	ftString	내장 프로시저의 소유자를 식별하는 스키마 이름
PROC_NAME	ftString	내장 프로시저 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
PROC_TYPE	ftInteger	내장 프로시저의 타입을 식별합니다. 이 값은 다음 값들 중에서 하나 이상의 합입니다. 1: 프로시저(반환 값 없음) 2: 함수(값 반환) 4: 패키지 8: 시스템 프로시저
IN_PARAMS	ftSmallint	입력 매개변수의 수
OUT_PARAMS	ftSmallint	출력 매개변수의 수

필드에 대한 정보

지정된 테이블의 필드(*stColumns*)에 대한 정보를 요청하면 결과 데이터셋이 각 필드에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 26.3 필드를 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호
CATALOG_NAME	ftString	나열하는 필드의 테이블을 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 *Database* 매개변수와 동일합니다.

표 26.3 필드를 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
SCHEMA_NAME	ftString	필드의 소유자를 식별하는 스키마 이름
TABLE_NAME	ftString	필드를 포함하는 테이블의 이름
COLUMN_NAME	ftString	속성 이름. 이 값은 데이터셋의 정렬 순서를 결정합니다.
COLUMN_POSITION	ftSmallint	테이블에서 열의 위치
COLUMN_TYPE	ftInteger	필드에 있는 값의 타입을 식별합니다. 이 값은 다음 중 하나 이상의 합입니다. 1: 행 ID 2: 행 버전 4: 자동 증가 필드 8: 기본값이 있는 필드
COLUMN_DATATYPE	ftSmallint	열의 데이터 타입. sqllinks.h에 정의된 논리 필드 타입 상수 중 하나입니다.
COLUMN_TYPENAME	ftString	데이터 타입을 설명하는 문자열. COLUMN_DATATYPE과 COLUMN_SUBTYPE에 포함되어 있지만 일부 DDL 문에 사용되는 형식으로 된 동일한 정보입니다.
COLUMN_SUBTYPE	ftSmallint	열의 데이터 타입에 대한 하위 타입. sqllinks.h에 정의된 논리 하위 타입 상수 중 하나입니다.
COLUMN_PRECISION	ftInteger	필드 타입의 크기(문자열의 문자 수, 바이트 필드의 바이트, BCD 값의 유효 숫자, ADT 필드의 멤버 등)
COLUMN_SCALE	ftSmallint	BCD 값의 소수점 오른쪽의 숫자의 수 또는 ADT 및 배열 필드의 자손 수
COLUMN_LENGTH	ftInteger	필드 값 저장에 필요한 바이트 수
COLUMN_NULLABLE	ftSmallint	필드를 비워 둘 수 있는지 여부를 표시하는 부울. 0은 필드에 값이 필요하다는 의미입니다.

인덱스에 대한 정보

테이블의 인덱스(stIndexes)에 대한 정보를 요청하면 결과 데이터셋이 각 레코드별로 각 필드 레코드를 포함합니다. 다중 레코드 인덱스는 여러 레코드를 사용하여 설명됩니다. 데이터셋은 다음과 같은 열을 포함합니다.

표 26.4 인덱스를 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호
CATALOG_NAME	ftString	인덱스를 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 *Database* 매개변수와 동일합니다.
SCHEMA_NAME	ftString	인덱스 소유자를 식별하는 스키마 이름
TABLE_NAME	ftString	인덱스가 정의되는 테이블의 이름
INDEX_NAME	ftString	인덱스 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.

표 26.4 인덱스를 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
PKEY_NAME	ftString	Primary Key의 이름을 나타냅니다.
COLUMN_NAME	ftString	인덱스의 필드(열) 이름
COLUMN_POSITION	ftSmallint	인덱스에서 이 필드의 위치
INDEX_TYPE	ftSmallint	인덱스의 타입을 식별합니다. 이 값은 다음 값들 중에서 하나 이상의 합입니다. 1: 고유하지 않은 값 2: 고유한 값 4: Primary Key
SORT_ORDER	ftString	인덱스가 오름차순(a)인지 내림차순(d)인지 나타냅니다.
FILTER	ftString	인덱싱된 레코드를 제한하는 필터 조건을 설명합니다.

내장 프로시저 매개변수에 대한 정보

내장 프로시저의 매개변수(*stProcedureParams*)에 대한 정보를 요청하면 결과 데이터셋이 각 매개변수에 대한 레코드 포함합니다. 다음과 같은 열이 포함됩니다.

표 26.5 매개변수를 나열하는 메타데이터 테이블의 열

열 이름	필드 타입	열 데이터 설명
RECNO	ftInteger	각 레코드를 고유하게 식별하는 레코드 번호
CATALOG_NAME	ftString	내장 프로시저를 포함하는 카탈로그(데이터베이스) 이름. 이것은 SQL 연결 컴포넌트에 있는 *Database* 매개변수와 동일합니다.
SCHEMA_NAME	ftString	내장 프로시저의 소유자를 식별하는 스키마 이름
PROC_NAME	ftString	매개변수를 포함하는 내장 프로시저의 이름
PARAM_NAME	ftString	매개변수 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.
PARAM_TYPE	ftSmallint	매개변수의 타입을 식별합니다. *TParam* 객체의 *ParamType* 속성과 동일합니다.
PARAM_DATATYPE	ftSmallint	매개변수의 데이터 타입. sqllinks.h에 정의된 논리 필드 타입 상수 중 하나입니다.
PARAM_SUBTYPE	ftSmallint	매개변수의 데이터 타입에 대한 하위 타입. sqllinks.h에 정의된 논리 하위 타입 상수 중 하나입니다.
PARAM_TYPENAME	ftString	데이터 타입을 설명하는 문자열. PARAM_DATATYPE과 PARAM_SUBTYPE에 포함되어 있지만 일부 DDL 문에 사용되는 형식으로 된 동일한 정보입니다.
PARAM_PRECISION	ftInteger	부동 소수점 값의 최대 자릿수 또는 최대 바이트 수(문자열 및 Bytes 필드)
PARAM_SCALE	ftSmallint	부동 소수점 값의 소수점 오른쪽의 숫자 수
PARAM_LENGTH	ftInteger	매개변수 값 저장에 필요한 바이트 수
PARAM_NULLABLE	ftSmallint	매개변수를 비워 둘 수 있는지 여부를 표시하는 부울 0은 매개변수에 값이 필요하다는 의미입니다.

dbExpress 애플리케이션 디버깅

데이터베이스 애플리케이션을 디버깅하는 동안, 이 애플리케이션을 사용하여 프로바이더 컴포넌트 또는 *dbExpress* 드라이버에 의해 자동으로 생성되는 메시지를 비롯하여, 연결 컴포넌트 간에 전달되는 SQS 메시지를 모니터할 수 있습니다.

TSQLMonitor를 사용하여 SQL 명령 모니터

*TSQLConnection*은 *TSQLMonitor* 컴포넌트를 함께 사용하여 이 메시지를 인터셉트하고, 메시지를 문자열 리스트에 저장합니다. *TSQLMonitor*는 *dbExpress*가 관리하는 모든 명령이 아니라 하나의 *TSQLConnection* 컴포넌트를 포함하는 명령만 모니터한다는 점을 제외하면, BDE와 함께 사용할 수 있는 SQL 모니터 유틸리티와 매우 비슷하게 작동합니다.

*TSQLMonitor*을 사용하려면 다음과 같이 합니다.

1. SQL 명령을 모니터할 *TSQLConnection* 컴포넌트가 있는 폼이나 데이터 모듈에 *TSQLMonitor*를 추가합니다.
2. *SQLConnection* 속성을 *TSQLConnection* 컴포넌트로 설정합니다.
3. SQL 모니터의 *Active* 속성을 **true**로 설정합니다.

SQL 명령이 서버로 보내지면 SQL 모니터의 *TraceList* 속성이 자동으로 업데이트되어 인터셉트된 모든 SQL 명령을 나열합니다.

FileName 속성의 값을 지정한 다음 *AutoSave* 속성을 **true**로 설정하여 이 리스트를 파일에 저장할 수 있습니다. *AutoSave*를 사용하면 새 메시지가 로그될 때마다 SQL 모니터가 *TraceList* 속성 내용을 파일에 저장합니다.

메시지가 로그될 때마다 파일을 저장하는 오버헤드를 피하려면 *OnLogTrace* 이벤트 핸들러를 사용하여 여러 메시지가 로그된 후에만 파일을 저장할 수 있습니다. 예를 들어, 다음 이벤트 핸들러는 리스트가 너무 길어지지 않도록 저장 후에 로그를 지우고, 10번째 메시지마다 *TraceList*의 내용을 저장합니다.

```
void __fastcall TForm1::SQLMonitor1LogTrace(TObject *Sender, void *CBInfo)
{
  TSQLMonitor *pMonitor = dynamic_cast<TSQLMonitor *>(Sender);
  if (pMonitor->TraceCount == 10)
  {
    // build unique file name
    AnsiString LogFileName = "C:\\log";
    LogFileName = LogFileName + IntToStr(pMonitor->Tag);
    LogFileName = LogFileName + ".txt"
    pMonitor->Tag = pMonitor->Tag + 1;
    // Save contents of log and clear the list
    pMonitor->SaveToFile(LogFileName);
    pMonitor->TraceList->Clear();

}
```

참고 이전 이벤트 핸들러를 사용할 경우 애플리케이션이 종료될 때 부분적인 리스트(항목 10개 미만)를 저장할 수도 있습니다.

콜백을 사용하여 SQL 명령 모니터

*TSQLMonitor*를 사용하는 대신, SQL 연결 컴포넌트의 *SetTraceCallbackEvent* 메소드를 사용하여 애플리케이션이 SQL 명령을 추적하는 방법을 사용자 정의할 수 있습니다. *SetTraceCallbackEvent* 는 *TSQLCallbackEvent* 타입의 콜백 및 콜백 함수로 전달되는 사용자 정의된 값을 매개변수로 사용합니다.

콜백 함수는 *CallType*과 *CBInfo*를 매개변수로 사용합니다.

- *CallType*은 나중에 사용할 수 있도록 예약됩니다.
- *CBInfo*는 범주(*CallType*과 동일), SQL 명령의 텍스트 및 *SetTraceCallbackEvent* 메소드로 전달되는 사용자 정의된 값을 포함하는 구조에 대한 포인터입니다.

콜백은 *CBRType* 타입의 값(대개 *cbrUSEDEF*)을 반환합니다.

dbExpress 드라이버는 SQL 연결 컴포넌트가 서버에 명령을 전달하거나, 서버가 오류 메시지를 반환할 때마다 콜백을 호출합니다.

경고 *TSQLConnection* 객체에 관련 *TSQLMonitor* 컴포넌트가 있을 경우 *SetTraceCallbackEvent*를 호출하지 마십시오. *TSQLMonitor*는 콜백 메커니즘을 사용하여 작동하고, *TSQLConnection*은 한 번에 하나의 콜백만 지원합니다.

장

27

클라이언트 데이터셋 사용

클라이언트 데이터셋은 메모리에 모든 데이터를 보관하는 특화된 데이터셋입니다. 클라이언트 데이터셋에서 메모리에 보관한 데이터를 조작할 수 있는 기능은 midas.dll에서 제공합니다. 클라이언트 데이터셋에서 데이터를 저장하는 데 사용하는 형식은 독립적이며 쉽게 옮길 수 있으므로 클라이언트 데이터셋에서 다음 작업을 할 수 있습니다.

- 파일 기반 데이터셋처럼 동작하면서 디스크상의 전용 파일을 읽거나 씁니다. 이러한 메커니즘을 지원하는 속성 및 메소드는 27-32페이지의 "파일 기반 데이터와 함께 클라이언트 데이터셋 사용"에서 설명합니다.
- 데이터베이스 서버 데이터에 대한 업데이트를 캐싱합니다. 캐싱된 업데이트를 지원하는 클라이언트 데이터셋 기능은 27-15페이지의 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용"에서 설명합니다.
- 멀티 티어 애플리케이션에서 클라이언트 부분의 데이터를 나타냅니다. 이러한 방식으로 작동하려면 27-24페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"에서 설명한 대로 클라이언트 데이터셋은 외부 프로바이더를 사용해야 합니다. 멀티 티어 데이터베이스 애플리케이션에 대한 자세한 내용은 29장, "멀티 티어 애플리케이션 생성"을 참조하십시오.
- 데이터셋이 아닌 소스의 데이터를 나타냅니다. 클라이언트 데이터셋이 외부 프로바이더의 데이터를 사용할 수 있으므로 특화된 프로바이더는 다양한 정보 소스를 사용하여 클라이언트 데이터셋 작업을 할 수 있습니다. 예를 들어, XML 프로바이더를 사용하여 클라이언트 데이터셋이 XML 문서의 정보를 나타내도록 할 수 있습니다.

파일 기반 데이터나 업데이트 캐싱, 외부 프로바이더의 데이터(예: XML 문서를 사용하거나 멀티 티어 애플리케이션의 경우), "briefcase model" 애플리케이션과 같은 여러 접근 방법의 조합에서 클라이언트 데이터셋을 사용하면 클라이언트 데이터셋에서 지원하는 광범위한 기능을 데이터 작업에 이용할 수 있습니다.

클라이언트 데이터셋을 사용하는 데이터 작업

모든 데이터셋의 경우처럼 클라이언트 데이터셋을 사용하면 데이터 소스 컴포넌트를 통해 데이터 인식 컨트롤에 데이터를 제공할 수 있습니다. 데이터 인식 컨트롤에서 데이터베이스 정보를 표시하는 방법에 대한 자세한 내용은 19장, "데이터 컨트롤 사용"을 참조하십시오.

클라이언트 데이터셋은 *TDataSet*로부터 상속받은 모든 속성과 메소드를 구현합니다. 이러한 일반적인 데이터셋 동작에 대한 자세한 내용은 22장, "데이터셋 이해"를 참조하십시오.

또한 클라이언트 데이터셋은 다음과 같은 테이블 타입 데이터셋의 공통 기능을 많이 구현합니다.

- 인덱스를 사용하여 레코드 정렬
- 인덱스를 사용하여 레코드 검색
- 범위를 사용하여 레코드 제한
- 마스터/디테일 관계 생성
- 읽기/쓰기 액세스 제어
- 원본으로 사용하는 데이터셋 생성
- 데이터셋 비우기
- 클라이언트 데이터셋 동기화

이러한 기능에 대한 자세한 내용은 22-24페이지의 "테이블 타입 데이터셋 사용"을 참조하십시오.

클라이언트 데이터셋은 모든 데이터를 메모리에 보관한다는 점에서 다른 데이터셋과 다릅니다. 이러한 이유에서 일부 데이터베이스 기능을 지원하기 위해 추가적인 기능이나 고려사항이 필요할 수도 있습니다. 이 장에서는 클라이언트 데이터셋의 이러한 몇 가지 공통 기능과 차이점을 설명합니다.

클라이언트 데이터셋에서 데이터 탐색

애플리케이션에서 표준 데이터 인식 컨트롤을 사용한다면 사용자는 이러한 컨트롤의 기본 동작을 사용하여 클라이언트 데이터셋의 레코드를 탐색할 수 있습니다. 또한 *First*, *Last*, *Next*, *Prior* 같은 표준 데이터셋 메소드를 사용하여 프로그래밍 방식으로 레코드를 탐색할 수 있습니다. 이러한 메소드에 대한 자세한 내용은 22-4페이지의 "데이터셋 탐색"을 참조하십시오.

대부분의 데이터셋과는 달리 클라이언트 데이터셋은 *RecNo* 속성을 사용하여 데이터셋의 특정 레코드로 커서를 갖다 놓을 수 있습니다. 일반적으로 애플리케이션에서는 *RecNo*를 사용하여 현재 레코드의 레코드 번호를 확인합니다. 하지만 클라이언트 데이터셋에서는 *RecNo*를 특정 레코드 번호로 설정하여 해당 레코드를 현재 레코드로 만들 수 있습니다.

레코드 표시 제한

사용 가능한 데이터의 부분 집합을 사용하도록 임시로 제한하기 위해 애플리케이션에서 범위와 필터를 사용할 수 있습니다. 범위나 필터를 적용하면 클라이언트 데이터셋에서 인메모리 캐시에 있는 데이터를 모두 표시하지는 않습니다. 대신 범위나 필터 조건에 맞는 데이터만 표시합니다. 필터 사용에 대한 자세한 내용은 22-12페이지의 "필터를 사용하여 데이터의 부분 집합 표시 및 편집"을 참조하십시오. 범위에 대한 자세한 내용은 22-30페이지의 "범위를 사용하여 레코드 제한"을 참조하십시오.

대부분의 데이터셋에서 필터 문자열은 SQL 명령으로 분석된 다음 데이터베이스 서버에서 구현됩니다. 따라서 서버의 SQL 언어로 인해 필터 문자열에서 사용될 수 있는 작업이 제한됩니다. 클라이언트 데이터셋은 자체 필터를 지원하므로 다른 데이터셋의 필터보다 더 많은 작업을 포함할 수 있습니다. 예를 들어, 클라이언트 데이터셋을 사용하면 필터 식에는 부분 문자열을 반환하는 문자열 연산자와 날짜/시간 값을 분석하는 연산자 등이 포함될 수 있습니다. 또한 클라이언트 데이터셋은 BLOB 필드에서 그리고 ADT 필드와 배열 필드와 같은 복잡한 필드에서 필터를 허용합니다.

클라이언트 데이터셋에서 필터에 사용할 수 있는 다양한 연산자와 함수가 필터를 지원하는 다른 데이터셋과 비교하여 아래에 나와 있습니다.

표 27.1 클라이언트 데이터셋의 필터 지원

연산자 또는 함수	예제	다른 데이터셋에서 지원 여부	설명
비교 연산자			
=	State = 'CA'	예	
<>	State <> 'CA'	예	
>=	DateEntered >= '1/1/1998'	예	
<=	Total <= 100,000	예	
>	Percentile > 50	예	
<	Field1 < Field2	예	
BLANK	State <> 'CA' or State = BLANK	예	필터에 명시적으로 포함되어 있어야만 비어 있는 레코드가 표시됨
IS NULL	Field1 IS NULL	아니오	
IS NOT NULL	Field1 IS NOT NULL	아니오	
논리 연산자			
and	State = 'CA' and Country = 'US'	예	
or	State = 'CA' or State = 'MA'	예	
not	not (State = 'CA')	예	
산술 연산자			
+	Total + 5 > 100	드라이버에 따라 다름	숫자, 문자열, 날짜(시간) + 숫자에 적용
-	Field1 - 7 <> 10	드라이버에 따라 다름	숫자, 문자열, 날짜(시간) - 숫자에 적용
*	Discount * 100 > 20	드라이버에 따라 다름	숫자에만 적용
/	Discount > Total / 5	드라이버에 따라 다름	숫자에만 적용

표 27.1 클라이언트 데이터셋의 필터 지원(계속)

연산자 또는 함수	예제	다른 데이터셋에서지원 여부	설명
문자열 함수			
Upper	Upper(Field1) = 'ALWAYS'	아니오	
Lower	Lower(Field1 + Field2) = 'josp'	아니오	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	아니오	값이 두 번째 인수의 위치에서 세 번째 인수의 끝이나 문자 번호로 이동. 첫 번째 문자는 위치 1
Trim	Trim(Field1 + Field2) Trim(Field1, '-')	아니오	세 번째 인수의 앞과 뒤를 제거. 세 번째 인수가 없으며 공백을 자름
TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '$') <> ''	아니오	Trim 참조
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> ''	아니오	Trim 참조
DateTime 함수			
Year	Year(DateField) = 2000	아니오	
Month	Month(DateField) <> 12	아니오	
Day	Day(DateField) = 1	아니오	
Hour	Hour(DateField) < 16	아니오	
Minute	Minute(DateField) = 0	아니오	
Second	Second(DateField) = 30	아니오	
GetDate	GetDate - DateField > 7	아니오	현재 날짜와 시간을 반환
Date	DateField = Date(GetDate)	아니오	datetime 값의 날짜 부분을 반환
Time	TimeField > Time(GetDate)	아니오	datetime 값의 시간 부분을 반환
기타			
Like	Memo LIKE '%filters%'	아니오	ESC 절이 없는 SQL-92처럼 작동. BLOB 필드에 적용되면 FilterOptions에서 대소문자 구분 여부를 결정
In	Day(DateField) in (1,7)	아니오	SQL-92처럼 작동. 두 번째 인수는 모두 같은 타입을 갖는 값의 리스트
*	State = 'M*'	예	부분 비교를 위한 와일드카드

범위나 필터를 적용할 때 클라이언트 데이터셋은 모든 레코드를 계속 메모리에 저장합니다. 범위나 필터는 클라이언트 데이터셋의 데이터를 탐색하거나 표시할 컨트롤에서 사용할 수 있는 레코드를 결정합니다.

참고 프로바이더에서 데이터를 가져올 때 매개변수를 프로바이더에 제공하여 클라이언트 데이터셋에서 저장하는 데이터를 제한할 수도 있습니다. 자세한 내용은 27-28페이지의 "매개변수로 레코드 제한"을 참조하십시오.

데이터 편집

클라이언트 데이터셋은 자체 데이터를 인메모리 데이터 패킷으로 나타냅니다. 이 패킷은 클라이언트 데이터셋의 *Data* 속성 값입니다. 그러나 디폴트로, 편집 내용은 *Data* 속성에 저장되지 않습니다. 대신 사용자나 프로그램에 의한 추가, 삭제, 수정 내용은 *Delta* 속성으로 표현되는 내부 변경 로그에 저장됩니다. 변경 로그는 다음과 같이 두 가지 용도로 사용됩니다.

- 데이터베이스 서버나 외부 프로바이더 컴포넌트에 업데이트 내용을 적용할 때 변경 로그가 필요합니다.
- 변경 로그는 변경 내용을 취소할 수 있는 고급 지원 기능을 제공합니다.

LogChanges 속성을 사용하면 로깅을 사용할 수 없게 만들 수 있습니다. *LogChanges*가 **true**인 경우, 로그에 변경 내용이 기록됩니다. *LogChanges*가 **false**인 경우는 *Data* 속성으로 직접 변경이 이루어집니다. 실행 취소 지원이 필요하지 않은 경우는 파일 기반 애플리케이션에서 변경 로그를 사용할 수 없게 만들 수 있습니다.

변경 로그의 편집 내용은 애플리케이션에서 삭제하기 전까지 계속 변경 로그에 남아 있습니다. 애플리케이션에서는 다음의 경우 편집 내용을 삭제합니다.

- 변경 취소
- 변경 내용 저장

참고 클라이언트 데이터셋을 파일에 저장하면 편집 내용은 변경 로그에서 삭제되지 않습니다. 데이터셋을 다시 로드할 때 *Data*와 *Delta* 속성은 데이터가 저장되었을 당시와 동일합니다.

변경 취소

레코드의 원래 버전이 *Data*에 바뀌지 않고 남아 있더라도 사용자가 레코드를 편집하거나, 다른 데이터로 이동했다가 다시 돌아올 때마다 사용자는 항상 최근에 변경될 버전의 레코드를 보게 됩니다. 사용자나 애플리케이션에서 레코드를 자주 편집하는 경우 변경된 레코드 버전은 각각의 항목으로 변경 로그에 저장됩니다.

레코드에 대한 각 변경 내용을을 저장하면 레코드의 이전 상태를 복구해야 할 경우 여러 단계의 실행 취소 작업을 지원할 수 있게 됩니다.

- 레코드에 대한 마지막 변경 내용을 제거하려면 *UndoLastChange*를 호출합니다. *UndoLastChange*는 부울 매개변수인 *FollowChange*를 취하는데 이 매개변수는 커서를 복구된 레코드로 옮길 것인지(**true**) 아니면 커서를 현재 레코드에 둘 것인지(**false**)를 표시합니다. 하나의 레코드를 여러 번 변경한 경우 *UndoLastChange*를 호출할 때마다 단계적으로 변경 내용이 제거됩니다. *UndoLastChange*는 성공이나 실패를 나타내는 부울 값을 반환합니다. 제거가 발생하면 *UndoLastChange*는 **true**를 반환합니다. *ChangeCount* 속성을 사용하여 취소할 변경 내용이 더 있는지 확인합니다. *ChangeCount*는 변경 로그에 저장된 변경 횟수를 의미합니다.

- 레코드별로 각 변경 내용을 제거하는 대신 한 번에 모두 제거할 수 있습니다. 레코드의 모든 변경 내용을 제거하려면 해당 레코드를 선택한 다음 *RevertRecord*를 호출합니다. *RevertRecord*는 변경 로그에서 현재 레코드에 대한 모든 변경을 제거합니다.
- 삭제된 레코드를 복구하려면 먼저 *StatusFilter* 속성을 [*usDeleted*]로 설정하여 삭제된 레코드를 "보이게" 만듭니다. 그리고 나서 복구할 레코드를 탐색하고 *RevertRecord*를 호출합니다. 마지막으로 이제 복구된 레코드가 포함된 데이터셋의 편집된 버전을 다시 볼 수 있게 *StatusFilter* 속성을 [*usModified*, *usInserted*, *usUnmodified*]로 설정합니다.
- 편집 작업 중 언제라도 *SavePoint* 속성을 사용하여 변경 로그의 현재 상태를 저장할 수 있습니다. *SavePoint*를 읽으면 변경 로그의 현재 위치로 특정 표시를 반환합니다. 나중에, 저장 위치(save point)를 읽은 다음에 발생한 모든 변경 내용을 취소하려면 *SavePoint*를 이전에 읽은 값으로 설정하십시오. 애플리케이션에서는 여러 저장 위치에 대한 값을 얻을 수 있습니다. 그러나 일단 변경 로그를 저장 위치로 백업하면 이후에 애플리케이션에서 읽은 모든 저장 위치의 값은 유효하지 않습니다.
- *CancelUpdates*를 호출하여 변경 로그에 기록된 모든 변경 내용을 버릴 수 있습니다. *CancelUpdates*는 모든 레코드에 대한 모든 편집 내용을 버림으로써 변경 로그를 지웁니다. *CancelUpdates*를 호출할 때는 주의를 기울여야 합니다. *CancelUpdates*를 호출하고 나면 로그에 있었던 어떠한 변경 내용도 복구할 수 없습니다.

변경 내용 저장

클라이언트 데이터셋이 데이터를 파일에 저장하는지 아니면 프로바이더를 통해 얻은 데이터를 나타내는지 여부에 따라 변경 로그의 변경 내용을 통합할 때 클라이언트 데이터셋은 다른 메커니즘을 사용합니다. 어떤 메커니즘을 사용하는지 상관없이 모든 업데이트 내용이 통합되면 변경 로그는 자동으로 비워집니다.

파일 기반 애플리케이션은 *Data* 속성에 의해 나타나는 로컬 캐시로 변경 내용을 쉽게 병합할 수 있습니다. 또한 다른 사용자에 의해 이루어진 변경 내용으로 로컬 편집을 해석하는데 신경 쓸 필요가 없습니다. 변경 로그를 *Data* 속성으로 병합하려면 *MergeChangeLog* 메소드를 호출하십시오. 27-33페이지의 "데이터에 변경 내용 병합"에서 이러한 과정을 설명합니다.

클라이언트 데이터셋을 사용하여 업데이트 내용을 캐싱하거나 외부 프로바이더 컴포넌트의 데이터를 나타내고 있는 경우 *MergeChangeLog*를 사용할 수 없습니다. 변경 로그의 정보는 데이터베이스나 소스 데이터셋에 저장된 데이터로 업데이트된 레코드를 해석할 때 필요합니다. 그 대신 변경 내용을 데이터베이스 서버나 소스 데이터셋에 전달하는 *ApplyUpdates*를 호출하고 수정 내용이 성공적으로 커밋된 경우에만 *Data* 속성을 업데이트합니다. 이 과정에 대한 자세한 내용은 27-20페이지의 "업데이트 적용"을 참조하십시오.

데이터 값 제약

클라이언트 데이터셋은 사용자의 데이터 편집 내용에 제약 조건을 적용할 수 있습니다. 이러한 제약 조건은 사용자가 변경 내용을 변경 로그에 포스트하려고 할 때 적용됩니다. 개발자 스스로 항상 사용자 정의 제약 조건을 제공할 수 있는데 이렇게 하면 클라이언트 데이터셋으로 포스트하는 값에 대해 개발자 고유의, 애플리케이션 정의 제한을 둘 수 있습니다.

그리고 클라이언트 데이터셋에서 BDE를 사용하여 액세스한 서버 데이터를 나타낼 때 데이터베이스 서버에서 임포트된 데이터 제약 조건을 적용할 수도 있습니다. 클라이언트 데이터셋에서 외부 프로바이더 컴포넌트를 사용하는 경우, 프로바이더에서는 제약 조건을 클라이언트 데이터셋으로 보낼 것인지 제어할 수 있고 클라이언트 데이터셋에서는 이러한 제약 조건을 사용할 것인지 제어할 수 있습니다. 프로바이더에서 데이터 패킷에 제약 조건을 포함할 것인지 여부를 제어하는 방법에 대한 자세한 내용은 28-12페이지의 "서버 제약 조건 처리"를 참조하십시오. 클라이언트 데이터셋에서 서버 제약 조건 적용을 해제하는 방법 및 이유에 대한 자세한 내용은 27-29페이지의 "서버로부터의 제약 조건 처리"를 참조하십시오.

사용자 정의 제약 조건 지정

클라이언트 데이터셋의 필드 컴포넌트 속성을 사용하여 사용자가 특정 데이터만 입력할 수 있도록 제약 조건을 부과할 수 있습니다. 각 필드 컴포넌트에는 다음과 같이 제약 조건을 지정하는 데 사용할 수 있는 두 가지 속성이 있습니다.

- *DefaultExpression* 속성은 사용자가 값을 입력하지 않을 경우 필드에 할당되는 기본값을 정의합니다. 데이터베이스 서버나 소스 데이터셋에서 필드에 대해 디폴트 표현식을 할당하는 경우에도, 데이터베이스 서버나 소스 데이터셋에 업데이트 내용이 적용되기 전에 클라이언트 데이터셋이 먼저 할당되기 때문에 클라이언트 데이터셋의 버전이 우선한다는 점에 유의하십시오.
- *CustomConstraint* 속성을 사용하면 필드 값을 포스트하기 전에 만족해야 하는 제약 조건을 할당할 수 있습니다. 이렇게 정의된 사용자 정의 제약 조건은 서버에서 임포트된 모든 제약 조건과 함께 적용됩니다. 필드 컴포넌트에서 사용자 정의 제약 조건을 사용하는 방법에 대한 자세한 내용은 23-21페이지의 "사용자 정의 제약 조건 생성"을 참조하십시오.

그리고 클라이언트 데이터셋의 *Constraints* 속성을 사용하여 레코드 수준 제약 조건을 만들 수 있습니다. *Constraints*는 *TCheckConstraint* 객체의 컬렉션으로 여기서 객체는 각각의 조건을 나타냅니다. *TCheckConstraint* 객체의 *CustomConstraint* 속성을 사용하여 레코드를 포스트할 때 검사되는 사용자 고유의 제약 조건을 추가할 수 있습니다.

정렬과 인덱싱

인덱스를 사용하면 애플리케이션에 여러 가지 이점을 제공할 수 있습니다.

- 클라이언트 데이터셋에서 데이터를 빨리 찾을 수 있습니다.
- 사용 가능 레코드를 제한하는 범위를 적용할 수 있습니다.
- 애플리케이션에서 조회 테이블이나 마스터/디테일 폼과 같은 다른 데이터셋과의 관계를 설정할 수 있습니다.
- 레코드가 나타나는 순서를 지정할 수 있습니다.

클라이언트 데이터셋에서 서버 데이터를 나타내거나 외부 프로바이더를 사용하면 받아들이는 데이터를 기반으로 디폴트 인덱스와 정렬 순서를 상속받습니다. 디폴트 인덱스를 DEFAULT_ORDER라고 합니다. 이 순서를 사용할 수는 있지만 인덱스를 변경하거나 삭제할 수는 없습니다.

디폴트 인덱스 이외에도 클라이언트 데이터셋은 변경 로그(*Delta* 속성)에 저장된 변경된 레코드에서 CHANGEINDEX라는 보조 인덱스를 유지합니다. CHANGEINDEX는 클라이언트 데이터셋의 모든 레코드를 *Delta*에 지정된 변경 내용이 적용되었을 경우 나타나는 대로 정렬합니다. CHANGEINDEX는 DEFAULT_ORDER로부터 상속받은 순서를 기반으로 합니다. DEFAULT_ORDER와 마찬가지로 CHANGEINDEX 인덱스를 변경하거나 삭제할 수는 없습니다.

기존의 다른 인덱스를 사용할 수 있으며 인덱스를 직접 만들 수도 있습니다. 다음 단원에서 클라이언트 데이터셋에서 인덱스를 만들고 사용하는 방법을 설명합니다.

참고 역시 클라이언트 데이터셋에 적용되는, 테이블 타입 데이터셋 인덱스에 대한 자료를 검토하려면 22-25페이지의 "인덱스를 사용하여 레코드 정렬"과 22-30페이지의 "범위를 사용하여 레코드 제한"을 참조하십시오.

새 인덱스 추가

클라이언트 데이터셋에 인덱스를 추가하는 방법에는 세 가지가 있습니다.

- 클라이언트 데이터셋의 레코드를 정렬하는 임시 인덱스를 런타임 시 만들려면 *IndexFieldNames* 속성을 사용할 수 있습니다. 세미콜론으로 구분하여 필드 이름을 지정합니다. 리스트에 있는 필드 이름 순서대로 인덱스에서의 순서가 결정됩니다.

  이 메소드는 인덱스를 추가하는 가장 약한 메소드입니다. 내림차순이나 대소문자를 구분하지 않는 인덱스를 지정할 수 없으며 결과 인덱스에서는 그룹화를 지원하지 않습니다. 이 인덱스들은 데이터셋을 닫으면 영구적으로 남지 않으며 클라이언트 데이터셋을 파일에 저장할 때 저장되지 않습니다.

- 그룹화에 사용할 수 있는 인덱스를 런타임 시 만들려면 *AddIndex*를 호출합니다. *AddIndex*를 사용하면 다음을 포함하여 인덱스의 속성을 지정할 수 있습니다.
  - 인덱스 이름. 런타임 시 인덱스 전환에 사용할 수 있습니다.
  - 인덱스를 구성하는 필드. 인덱스는 이 필드를 사용하여 레코드를 정렬하고 이 필드에서 특정 값을 가진 레코드를 찾습니다.
  - 인덱스에서 레코드를 정렬하는 방법. 디폴트로, 인덱스는 컴퓨터의 로케일에 따라 오름차순으로 정렬됩니다. 디폴트 정렬 순서는 대소문자를 구분합니다. 옵션을 설정하면 전체 인덱스에서 대소문자를 구분하지 않거나 내림차순으로 정렬하게 할 수 있습니다. 또는 대소문자를 구분하지 않고 정렬할 필드의 리스트와 내림차순으로 정렬할 필드의 리스트를 제공할 수 있습니다.
  - 인덱스에 대한 그룹화 지원의 디폴트 레벨

  *AddIndex*로 만든 인덱스는 클라이언트 데이터셋이 닫히면 영구적으로 남지 않습니다. 즉, 클라이언트 데이터셋을 다시 열 때 없어집니다. 데이터셋이 닫히면 *AddIndex*를 호출할 수 없습니다. *AddIndex*를 사용하여 추가하는 인덱스는 클라이언트 데이터셋을 파일에 저장할 때 저장되지 않습니다.

- 인덱스를 만드는 세 번째 방법은 클라이언트 데이터셋이 만들어질 때 만든 인덱스를 사용하는 것입니다. 클라이언트 데이터셋을 만들기 전에 *IndexDefs* 속성을 사용하여 원하는 인덱스를 지정합니다. 그러면 *CreateDataSet*을 호출할 때 원본으로 사용하는 데이터셋과 함께 인덱스가 만들어집니다. 클라이언트 데이터셋 생성에 대한 자세한 내용은 22-37페이지의 "테이블 생성 및 삭제"를 참조하십시오.

*AddIndex*에서 데이터셋 지원 그룹화로 만드는 인덱스는 일부 필드에서는 올림차순으로, 또 일부 필드에서는 내림차순으로 정렬이 가능하고, 일부 필드에서는 대소문자 구분이 불가능하며 또 일부 필드에서는 대소문자 구분이 가능합니다. 이러한 방법으로 만들어진 인덱스는 영구적으로 남으며 클라이언트 데이터셋을 파일에 저장할 때 저장됩니다.

팁 내부적으로 계산된 필드에서 클라이언트 데이터셋으로 인덱스를 만들고 정렬할 수 있습니다.

인덱스 삭제와 전환

클라이언트 데이터셋에서 사용하기 위한 만든 인덱스를 제거하려면 *DeleteIndex*를 호출한 다음 제거할 인덱스 이름을 지정합니다. DEFAULT_ORDER 인덱스와 CHANGEINDEX 인덱스는 제거할 수 없습니다.

두 개 이상의 인덱스를 사용할 수 있는 경우에 다른 인덱스를 사용하려면 *IndexName* 속성을 사용하여 사용할 인덱스를 선택합니다. 디자인 타임 시 Object Inspector의 *IndexName* 속성 드롭다운 박스에서 사용 가능한 인덱스 중 하나를 선택할 수 있습니다.

인덱스를 사용하여 데이터 그룹화

클라이언트 데이터셋에서 인덱스를 사용할 경우 레코드에 정렬 순서가 자동으로 적용됩니다. 이 순서로 인해 인접한 레코드는 인덱스를 구성하는 필드에 대개 중복 값을 포함합니다. 예를 들어, SalesRep 필드와 Customer 필드에 인덱스가 설정되어 있는 Orders 테이블에서 다음과 같은 데이터 부분을 살펴봅니다.

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

정렬 순서 때문에 SalesRep 열의 인접한 값이 중복됩니다. SalesRep 1의 레코드 내에서, Customer 열의 인접한 값이 중복됩니다. 즉, 데이터가 SalesRep에 의해 그룹화되며, SalesRep 그룹 내에서는 Customer에 의해 그룹화됩니다. 각 그룹화에는 관련 레벨이 있습니다. 이 경우 SalesRep 그룹은 다른 그룹에 의해 중첩되지 않았으므로 레벨 1이 되고, Customer 그룹은 레벨 1인 그룹에 중첩되었으므로 레벨 2가 됩니다. 그룹화 레벨은 인덱스에서 필드의 순서에 해당합니다.

클라이언트 데이터셋을 사용하면 현재 레코드가 특정 그룹화 레벨 내에서 어느 위치에 있는지를 확인할 수 있습니다. 이렇게 하면 그룹에서 첫 번째 레코드인지, 그룹의 중간에 있는지 또는 그룹의 마지막 레코드인지에 따라 애플리케이션에서 레코드를 다르게 표시할 수 있습니다. 예를 들어, 그룹의 첫 번째 레코드에 있는 필드 값만 표시하고 중복 값은 제거하려고 합니다. 위의 Orders 테이블에 이를 적용하면 다음과 같은 결과를 얻게 됩니다.

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200

SalesRep	Customer	OrderNo	Amount
		6	75
2	1	1	10
	3	4	200

그룹 내에서 현재 레코드의 위치를 확인하려면 *GetGroupState* 메소드를 사용합니다. *GetGroupState*는 그룹의 레벨을 부여하는 정수를 사용하며 그룹에서 현재 레코드가 그룹의 어느 곳에 있는지(첫 번째 레코드, 마지막 레코드 또는 둘 다 아님) 알려주는 값을 반환합니다.

인덱스를 만들 때 인덱스가 지원하는 그룹화 레벨을 인덱스에 있는 필드 수만큼 지정할 수 있습니다. *GetGroupState*는 인덱스에서 추가 필드의 레코드를 정렬하더라도 해당 레벨 이상으로 그룹에 대한 정보를 제공할 수는 없습니다.

계산된 값 표시

데이터셋의 경우처럼 계산된 필드를 클라이언트 데이터셋에 추가할 수 있습니다. 이 필드는 동일한 레코드에 있는 다른 필드의 값에 따라 동적으로 계산할 수 있습니다. 계산된 필드 사용에 대한 자세한 내용은 23-7페이지의 "계산된 필드 정의"를 참조하십시오.

하지만 클라이언트 데이터셋을 사용하면 내부적으로 계산된 필드를 사용하여 필드가 계산될 때 최적화가 가능합니다. 내부적으로 계산된 필드에 대한 자세한 내용은 다음에 나오는 "클라이언트 데이터셋에서 내부적으로 계산된 필드 사용"을 참조하십시오.

유지 보수된 집계를 사용하여 여러 레코드의 데이터를 요약하는 계산 값을 생성하도록 클라이언트 데이터셋에 명령할 수도 있습니다. 유지 보수된 집계에 대한 자세한 내용은 27-11페이지의 "유지 보수된 집계 사용"을 참조하십시오.

클라이언트 데이터셋에서 내부적으로 계산된 필드 사용

다른 데이터셋의 경우 애플리케이션은 레코드가 변경되거나 사용자가 현재 레코드의 필드를 편집할 때마다 계산된 필드의 값을 계산해야 합니다. 이러한 작업은 *OnCalcFields* 이벤트 핸들러에서 이루어집니다.

클라이언트 데이터셋을 사용하면 클라이언트 데이터셋의 데이터에 계산된 값을 저장하여 계산된 필드를 다시 계산해야 하는 횟수를 최소화할 수 있습니다. 계산된 값이 클라이언트 데이터셋으로 저장되어 있는 경우 사용자가 현재 레코드를 편집하면 값을 여전히 다시 계산해야 하지만 애플리케이션에서는 현재 레코드가 변경될 때마다 값을 다시 계산하지 않아도 됩니다. 클라이언트 데이터셋의 데이터에 계산된 값을 저장하려면 계산된 필드 대신에 내부적으로 계산된 필드를 사용하십시오.

계산된 필드와 마찬가지로 내부적으로 계산된 필드는 *OnCalcFields* 이벤트 핸들러에서 계산됩니다. 하지만 클라이언트 데이터셋의 *State* 속성을 검사하여 이벤트 핸들러를 최적화할 수 있습니다. *State*가 *dsInternalCalc*인 경우는 내부적으로 계산된 필드를 다시 계산해야 합니다. *State*가 *dsCalcFields*인 경우는 정기적으로 계산된 필드를 다시 계산하기만 하면 됩니다.

내부적으로 계산된 필드를 사용하려면 클라이언트 데이터셋을 만들기 전에 필드를 내부적으로 계산된 필드로 정의해야 합니다. 영구적 필드(persistent field)를 사용하는지 아니면 필드 정의를 사용하는지에 따라 다음과 같은 방법으로 내부적으로 계산된 필드를 사용할 수 있습니다.

- 영구적 필드를 사용하고 있다면 Fields Editor에서 InternalCalc를 선택하여 필드를 내부적 계산 형식으로 정의할 수 있습니다.
- 필드 정의를 사용하고 있다면 관련 필드 정의의 *InternalCalcField* 속성을 **true**로 설정합니다.

참고 다른 종류의 데이터셋은 내부적으로 계산된 필드를 사용합니다. 그러나 다른 데이터셋의 경우 *OnCalcFields* 이벤트 핸들러에서 이 값들을 계산하지 마십시오. 대신 BDE나 원격 데이터베이스 서버에서 이 값들이 자동으로 계산됩니다.

유지 보수된 집계 사용

클라이언트 데이터셋은 레코드 그룹에 대해서 데이터를 요약할 수 있는 기능을 제공합니다. 이 요약들은 데이터셋에서 데이터를 편집할 때 자동적으로 업데이트되기 때문에 이 요약된 데이터를 "유지 보수된 집계"라고 합니다.

가장 단순한 형태로는, 유지 보수된 집계를 사용하여 클라이언트 데이터셋의 열에 있는 모든 값의 합계와 같은 정보를 알아낼 수 있습니다. 하지만 유지 보수된 집계는 다양한 요약 계산을 지원하고, 그룹화를 지원하는 특정 인덱스의 한 필드에서 지정된 여러 레코드 그룹의 소계를 제공할 수 있을 정도로 충분한 유연성을 갖추고 있습니다.

집계 지정

클라이언트 데이터셋의 레코드에 대한 요약을 계산할 것을 지정하려면 *Aggregates* 속성을 사용합니다. *Aggregates*는 집계 규정의 컬렉션(*TAggregate*)입니다. 디자인 타임 시 Collection Editor를 사용하거나 런타임 시 *Aggregates*의 *Add* 메소드를 사용하여 집계 규정을 클라이언트 데이터셋에 추가할 수 있습니다. 집계에 대한 필드 컴포넌트를 만들려면 Fields Editor에서 집계된 값에 대한 영구적 필드를 만드십시오.

참고 집계된 필드를 만들 때 적절한 집계 객체가 클라이언트 데이터셋의 *Aggregates* 속성에 자동으로 추가됩니다. 집계된 영구적 필드를 만들 때에는 명시적으로 추가하지 마십시오. 집계된 영구적 필드 생성에 대한 자세한 내용은 23-10페이지의 "집계 필드 정의"를 참조하십시오.

각 집계에 대해 *Expression* 속성은 자신이 나타내는 요약 계산을 표시합니다. *Expression*에는 다음과 같은 단순한 요약 표현식을 포함할 수 있습니다.

```
Sum(Field1)
```

또는 다음과 같이 여러 필드의 정보를 결합하는 복잡한 표현식을 포함할 수도 있습니다.

```
Sum(Qty * Price) - Sum(AmountPaid)
```

집계 표현식에는 표 27.2에 있는 하나 이상의 요약 연산자가 포함되어 있습니다.

표 27.2 유지 보수된 집계의 요약 연산자

연산자	용도
Sum	숫자 필드 또는 표현식의 값의 합계
Avg	숫자나 날짜 시간 필드 또는 표현식의 평균값 계산
Count	필드 또는 표현식의 비어 있지 않은 값들의 수 지정
Min	문자열, 숫자, 날짜 시간 필드나 표현식의 최소값 표시
Max	문자열, 숫자, 날짜 시간 필드나 표현식의 최대값 표시

요약 연산자는 필터 작성에 사용하는 연산자와 같은 연산자로 필드 값 또는 필드 값으로부터 만들어진 표현식에 사용됩니다. 요약 연산자는 중첩할 수 없습니다. 요약된 값과 다른 요약된 값에 연산자를 사용하거나 요약된 값과 상수에 연산자를 사용하여 표현식을 만들 수 있습니다. 그러나 요약된 값과 필드 값을 결합할 수 없습니다. 왜냐하면 그러한 표현식은 어떤 레코드가 필드 값을 제공하는지에 대한 지시가 없기 때문에 모호합니다. 이러한 규칙은 다음 표현식에서 설명됩니다.

```
Sum(Qty * Price)              {legal -- summary of an expression on fields }
Max(Field1) - Max(Field2)     {legal -- expression on summaries }
Avg(DiscountRate) * 100       {legal -- expression of summary and constant }
Min(Sum(Field1))              {illegal -- nested summaries }
Count(Field1) - Field2        {illegal -- expression of summary and field }
```

여러 레코드 그룹의 집계

유지 보수된 집계는 디폴트로 클라이언트 데이터셋의 모든 레코드를 요약하도록 계산됩니다. 하지만 특정 그룹의 레코드만 요약하도록 지정할 수 있습니다. 이렇게 하면 공통 필드값을 공유하는 여러 레코드 그룹의 소계와 같은 중간 요약을 제공할 수 있습니다.

한 레코드 그룹의 유지 보수된 집계를 지정할 수 있으려면 적절한 그룹화를 지원하는 인덱스를 사용해야 합니다. 그룹화 지원에 대한 자세한 내용은 27-9페이지의 "인덱스를 사용하여 데이터 그룹화"를 참조하십시오.

사용자가 요약하기 원하는 방식으로 데이터를 그룹화하는 인덱스가 있으면, 집계의 *IndexName* 과 *GroupingLevel* 속성을 지정하여 사용할 인덱스를 표시하고 요약할 레코드를 정의하는 해당 인덱스의 그룹이나 하위 그룹을 표시합니다.

예를 들어, SalesRep로 그룹화되고, SalesRep 내에서는 Customer로 그룹화된 Orders 테이블의 다음 데이터 부분을 살펴봅니다.

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200

SalesRep	Customer	OrderNo	Amount
1	2	6	75
2	1	1	10
2	3	4	200

다음 코드는 각 판매 사원의 총 판매액을 나타내는 유지 보수된 집계를 설정합니다.

```
Agg->Expression = "Sum(Amount)";
Agg->IndexName = "SalesCust";
Agg->GroupingLevel = 1;
Agg->AggregateName = "Total for Rep";
```

특정 판매 사원 내의 각 고객에 대한 정보를 요약하는 집계를 추가하려면 레벨을 2로 설정하여 유지 보수된 집계를 만듭니다.

한 레코드 그룹을 요약하는 유지 보수된 집계는 지정된 인덱스에 연결됩니다. *Aggregates* 속성은 다양한 인덱스를 사용하는 집계를 포함할 수 있습니다. 그러나 전체 데이터셋을 요약하는 집계와 현재 인덱스를 사용하는 집계만이 유효합니다. 현재 인덱스를 변경하면 유효한 집계도 변경됩니다. 언제든지 유효한 집계를 확인하려면 *ActiveAggs* 속성을 사용합니다.

집계 값 얻기

유지 보수된 집계 값을 얻으려면 집계를 나타내는 *TAggregate* 객체의 *Value* 메소드를 호출합니다. *Value*는 클라이언트 데이터셋의 현재 레코드를 가지고 있는 그룹의 유지 보수된 집계를 반환합니다.

전체 클라이언트 데이터셋을 요약하는 경우, 유지 보수된 집계를 얻기 위해 언제라도 *Value*를 호출할 수 있습니다. 그러나 그룹화된 정보를 요약하는 경우 현재 레코드가 요약을 원하는 그룹 내에 있는지를 확인해야 합니다. 이 때문에 그룹의 첫 레코드로 이동할 때나 그룹의 마지막 레코드로 이동할 때처럼 명확하게 지정된 경우에만 집계 값을 구하는 것이 좋습니다. *GetGroupState* 메소드를 사용하여 현재 레코드의 그룹 내 위치를 확인해야 합니다.

유지 보수된 집계를 데이터 인식 컨트롤에 표시하려면 Fields Editor를 사용하여 영구적 집계 필드 컴포넌트를 만듭니다. Fields Editor의 집계 필드를 지정할 때 클라이언트 데이터셋의 *Aggregates*는 적절한 집계 규정을 포함하도록 자동으로 업데이트됩니다. *AggFields* 속성에는 새 집계 필드 컴포넌트가 포함되며 *FindField* 메소드에서 이 집계 필드 컴포넌트를 반환합니다.

다른 데이터셋에서 데이터 복사

디자인 타임 시 다른 데이터셋으로부터 데이터를 복사하려면 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Assign Local Data를 선택합니다. 프로젝트에서 사용 가능한 모든 데이터셋을 나열하는 다이얼로그 박스가 나타납니다. 복사할 데이터와 구조를 갖는 데이터셋을 선택한 다음 OK를 선택합니다. 소스 데이터셋을 복사하면 클라이언트 데이터셋은 자동으로 활성화됩니다.

런타임 시 다른 데이터셋으로부터 복사하려면 데이터셋의 데이터를 직접 할당하거나 소스가 다른 클라이언트 데이터셋인 경우, 커서를 복제할 수 있습니다.

데이터 직접 할당

클라이언트 데이터셋의 *Data* 속성을 사용하여 다른 데이터셋에서 클라이언트 데이터셋으로 데이터를 할당할 수 있습니다. *Data*는 OleVariant 형식의 데이터 패킷입니다. 데이터 패킷은 다른 클라이언트 데이터셋이나 프로바이더를 사용하여 임의의 다른 데이터셋에서 가져올 수 있습니다. 일단 데이터 패킷을 *Data*에 할당하면 데이터 소스 컴포넌트에 의해 클라이언트 데이터셋에 연결된 데이터 인식 컨트롤에 내용이 자동으로 표시됩니다.

서버 데이터를 나타내거나 외부 프로바이더 컴포넌트를 사용하는 클라이언트 데이터셋을 열면 데이터 패킷이 자동으로 *Data*에 할당됩니다.

클라이언트 데이터셋에서 프로바이더를 사용하고 있지 않으면 다음과 같이 다른 클라이언트 데이터셋에서 데이터를 복사해올 수 있습니다.

```
ClientDataSet1->Data = ClientDataSet2->Data;
```

참고 다른 클라이언트 데이터셋의 *Data* 속성을 복사하면 변경 로그는 복사되지만 적용된 필터나 범위는 반영하지 않습니다. 필터나 범위를 포함하려면 소스 데이터셋의 커서를 대신 하나 더 복제해야 합니다.

클라이언트 데이터셋 이외의 데이터셋에서 복사해 오는 경우 다음과 같이 데이터셋 프로바이더 컴포넌트를 만들어 소스 데이터셋에 연결한 다음 데이터를 복사합니다.

```
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = SourceDataSet;
ClientDataSet1->Data = TempProvider->Data;
delete TempProvider;
```

참고 사용자가 직접 *Data* 속성에 할당하면 새 데이터 패킷은 기존의 데이터에 병합되지 않습니다. 그 대신 모든 이전 데이터가 바뀝니다.

다른 데이터셋의 데이터를 복사하는 대신 변경 내용을 병합하는 경우 프로바이더 컴포넌트를 사용해야 합니다. 앞의 예제에서처럼 데이터셋 프로바이더를 만듭니다. 그러나 대상 데이터셋에 연결한 다음, 데이터 속성을 복사하는 대신에 *ApplyUpdates* 메소드를 사용합니다.

```
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = ClientDataSet1;
TempProvider->ApplyUpdates(SourceDataSet->Delta, -1, ErrCount);
delete TempProvider;
```

클라이언트 데이터셋 커서 복제

클라이언트 데이터셋은 사용자가 런타임 시 데이터의 두 번째 뷰로 작업할 수 있도록 *CloneCursor* 메소드를 사용합니다. *CloneCursor*를 사용하면 두 번째 클라이언트 데이터셋은 원래 클라이언트 데이터셋의 데이터를 공유할 수 있습니다. 이 방법은 원래 데이터를 모두 복사하는 것보다 비용이 적게 들지만 데이터를 공유하기 때문에 두 번째 클라이언트 데이터셋의 데이터를 수정하면 원래 클라이언트 데이터셋이 영향을 받습니다.

*CloneCursor*는 다음 세 가지 매개변수를 취합니다. *Source*는 복제할 클라이언트 데이터셋을 지정합니다. 나머지 두 매개변수(*Reset*과 *KeepSettings*)는 데이터 이외의 정보를 복사할지 여부를 나타냅니다. 모든 필터, 현재 인덱스, 마스터 테이블로의 연결(소스 데이터셋이 디테일 셋인 경우), *ReadOnly* 속성 및 연결 컴포넌트나 프로바이더 인터페이스로의 모든 연결 등이 이 정보에 해당합니다.

*Reset*과 *KeepSettings*가 **false**이면 복제된 클라이언트 데이터셋이 열리고 소스 클라이언트 데이터셋의 설정값이 대상의 속성 설정에 사용됩니다. *Reset*이 **true**이면 대상 데이터셋의 속성에 기본값(인덱스나 필터 없음, 마스터 테이블 없음, *ReadOnly*는 **false**, 연결 컴포넌트나 프로바이더는 지정되지 않음)이 설정됩니다. *KeepSettings*가 **true**이면 대상 데이터셋의 속성은 변경되지 않습니다.

데이터에 애플리케이션 특정 정보 추가

애플리케이션 개발자는 클라이언트 데이터셋의 *Data* 속성에 사용자 정의 정보를 추가할 수 있습니다. 이 정보는 데이터 패킷과 함께 제공되기 때문에 데이터를 파일이나 스트림에 저장할 때 포함됩니다. 데이터를 다른 데이터셋으로 복사할 때 애플리케이션 특정 정보가 복사됩니다. 옵션으로서 클라이언트 데이터셋에서 업데이트를 받을 때 이 정보를 프로바이더가 읽을 수 있도록 *Delta* 속성과 함께 포함될 수 있습니다.

애플리케이션 특정 정보를 *Data* 속성과 함께 저장하려면 *SetOptionalParam* 메소드를 사용합니다. 이 메소드를 통해 특정 이름의 데이터를 포함하는 OleVariant를 저장할 수 있습니다.

이러한 애플리케이션 특정 정보를 검색하려면 정보가 저장될 때 사용된 이름을 전달하는 *GetOptionalParam* 메소드를 사용합니다.

업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용

디폴트로, 대부분의 데이터셋에서 데이터를 편집하는 경우 레코드를 삭제하거나 포스트할 때마다 데이터셋에서 트랜잭션을 생성하고 해당 레코드를 데이터베이스 서버에서 삭제하거나 기록한 다음 트랜잭션을 커밋합니다. 데이터베이스에 변경 내용을 기록하는 데 문제가 발생하면 애플리케이션에서 즉시 통지를 받습니다. 레코드를 포스트할 때 데이터셋에서 예외를 발생시킵니다.

데이터셋에서 원격 데이터베이스 서버를 사용하는 경우 이러한 접근 방법은 현재 레코드를 편집한 다음 새 레코드로 이동할 때마다 개발자 애플리케이션과 서버간 네트워크 트래픽으로 인해 성능이 저하될 수 있습니다. 네트워크 트래픽을 최소화하기 위해 업데이트 내용을 로컬에 캐싱해야할 수도 있습니다. 업데이트 내용을 캐싱할 때 애플리케이션에서 데이터를 데이터베이스로부터 검색하여 로컬에 캐싱하고 편집한 다음 캐싱된 업데이트를 단일 트랜잭션으로 데이터베이스에 적용합니다. 업데이트 내용을 캐싱할 때 변경 내용 포스트나 레코드 삭제와 같은 데이터베이스에 대한 변경 내용은 데이터셋의 원본으로 사용하는 테이블에 직접 기록되지 않고 로컬에 저장됩니다. 변경이 완료되면 애플리케이션에서 캐싱된 변경 내용을 데이터베이스에 기록하는 메소드를 호출한 다음 캐시를 비웁니다.

업데이트 캐싱은 트랜잭션 시간을 최소화하고 네트워크 트래픽을 줄입니다. 하지만 캐싱된 데이터는 트랜잭션 제어를 받는 것이 아니라 애플리케이션에 속해 있습니다. 로컬, 인메모리에서 데이터 복사본을 작업 중인 동안에는 다른 애플리케이션에서 원본으로 사용하는 데이터베이스 테이블의 데이터를 변경할 수 있음을 의미합니다. 그리고 캐싱된 업데이트를 적용할 때까지 사용자의 변경 내용을 다른 애플리케이션에 볼 수도 있습니다. 이러한 이유로 volatile data를 작업하는 애플리케이션에서는 캐싱된 업데이트가 적절하지 않습니다. 변경 내용을 데이터베이스로 병합하려고할 때 충돌이 너무 많이 발생할 수 있습니다.

BDE와 ADO에서는 업데이트 캐싱에 대한 대체 메커니즘을 제공하고 있지만 다음과 같이 업데이트 캐싱에 클라이언트 데이터셋을 사용하는 것이 몇 가지 이점이 있습니다.

- 데이터셋이 마스터/디테일 관계에 연결되어 있을 때 업데이트 적용이 사용자에 맞게 처리됩니다. 따라서 다중 연결된 데이터셋에 대한 업데이트가 올바른 순서로 적용되게할 수 있습니다.
- 클라이언트 데이터셋은 업데이트 프로세스에 대한 최대 컨트롤 수를 제공합니다. 속성을 설정하면 레코드 업데이트를 위해 생성된 SQL에 영향을 미치거나 다중 테이블 조인의 레코드를 업데이트할 때 사용할 테이블을 지정할 수 있으며 *BeforeUpdateRecord* 이벤트 핸들러로부터 업데이트를 수동으로 적용할 수도 있습니다.
- 캐싱된 업데이트를 데이터베이스 서버에 적용할 때 오류가 발생하면 클라이언트 데이터셋과 데이터셋 프로바이더에서만 개발자 데이터셋의 원래(편집되지 않은) 값 및 실패한 새 업데이트(편집한) 값과 함께 데이터베이스 서버의 현재 레코드 값에 대한 정보를 제공합니다.
- 클라이언트 데이터셋에서는 전체 업데이트가 롤백되기 전에 허용할 수 있는 업데이트 오류의 수를 지정할 수 있습니다.

캐싱된 업데이트 사용의 개요

캐싱된 업데이트를 사용하려면 다음과 같은 순서로 프로세스가 응용 프로그램에서 발생해야 합니다.

1 **편집할 데이터를 표시합니다.** 표시하는 방법은 사용하고 있는 클라이언트 데이터셋의 타입에 따라 다음과 같이 달라집니다.

   - *TClientDataSet*를 사용하고 있으면 편집할 데이터를 나타내는 프로바이더 컴포넌트를 지정합니다. 이 내용은 27-24페이지의 "프로바이더 지정"에 설명되어 있습니다.
   - 특정 데이터 액세스 메커니즘과 연결된 클라이언트 데이터셋을 사용하고 있으면 다음 작업을 수행해야 합니다.
     - *DBConnection* 속성을 적절한 연결 컴포넌트로 설정하여 데이터베이스 서버를 식별합니다.
     - *CommandText*와 *CommandType* 속성을 지정하여 보려는 데이터를 표시합니다. *CommandType*은 *CommandText*가 실행할 SQL 문과 내장 프로시저 이름, 테이블 이름 중 어떤 것인지를 표시합니다. *CommandText*가 쿼리나 내장 프로시저인 경우 *Params* 속성을 사용하여 입력 매개변수를 제공해야 합니다.
     - 경우에 따라 *Options* 속성을 사용하여 중첩된 디테일 셋과 BLOB 데이터를 데이터 패킷에 포함할 것인지 아니면 개별적으로 가져와야 하는지, 특정 편집 타입(삽입, 수정, 삭제)을 사용할 수 없게 만들 것인지, 단일 업데이트로 여러 서버 레코드에 영향을 미칠 수 있는지 그리고 업데이트를 적용할 때 클라이언트 데이터셋 레코드를 새로 고칠 것인지를 표시해야 합니다. *Options*는 프로바이더의 *Options* 속성과 일치하므로 관련되지 않거나 적절하지 않은 옵션을 설정할 수 있게 해줍니다. 예를 들어, 클라이언트 데이터셋은 영구적 필드와 함께 데이터셋에서 데이터를 가져오지 않으므로 *poIncFieldProps*는 포함할 필요가 없습니다. 반대로 *poAllowCommandText*는 디폴트로 포함되는데 *CommandText* 속성을 사용할 수 없게 만들기 때문에 클라이언트 데이터셋에서 원하는 데이터를 지정할 때 poAllowCommandText를 포함할 수밖에 없습니다. 프로바이더의 *Options* 속성에 대한 자세한 내용은 28-5페이지의 "데이터 패킷에 영향을 주는 옵션 설정"을 참조하십시오.

2 **데이터를 표시하고 편집합니다.** 새 레코드 삽입을 허용하고 기존 레코드 삭제를 지원합니다. 각 레코드의 원본 복사본과 편집 내용이 모두 메모리에 저장됩니다. 이 프로세스는 27-5페이지의 "데이터 편집"에 설명되어 있습니다.

3 **필요하면 추가 레코드를 가져옵니다.** 디폴트로, 클라이언트 데이터셋은 레코드를 모두 가져와서 메모리에 저장합니다. 하지만 데이터셋에 많은 레코드가 있거나 대용량 BLOB 필드가 있는 레코드가 있으면 클라이언트 데이터셋에서 표시할 레코드만 가져온 다음 필요하면 다시 가져올 수 있도록 이러한 방식을 바꾸어야 할 수도 있습니다. 레코드 페칭 프로세스를 제어하는 방법에 대한 자세한 내용은 27-25페이지의 "소스 데이터셋이나 문서에서 데이터 요청"을 참조하십시오.

4 **경우에 따라 레코드를 새로 고칩니다.** 시간이 경과하면 다른 사용자도 데이터베이스 서버의 데이터를 수정할 수 있습니다. 따라서 클라이언트 데이터셋의 데이터가 서버 데이터와 점점 더 달라지므로 업데이트를 적용할 때 오류가 증가할 가능성이 커집니다. 이러한 문제점을 보완하기 위해 아직 편집되지 않은 레코드를 새로 고칠수 있습니다. 자세한 내용은 27-30페이지의 "레코드 새로 고침"을 참조하십시오.

5 **캐싱된 레코드를 데이터베이스에 로컬로 적용하거나 업데이트를 취소합니다.** 데이터베이스에 기록된 레코드마다 *BeforeUpdateRecord* 이벤트가 실행됩니다. 개별 레코드를 데이터베이스에 기록할 때 오류가 발생하면 *OnUpdateError* 이벤트는 가능하면 애플리케이션 자체에서 오류를 수정하게 한 다음 업데이트를 계속 수행합니다. 업데이트가 완료되면 성공적으로 적용된 모든 업데이트 내용은 로컬 캐시에서 지워집니다. 업데이트 내용을 데이터베이스에 적용하는 방법에 대한 자세한 내용은 27-19페이지의 "레코드 업데이트"를 참조하십시오.

   업데이트를 적용하는 대신 애플리케이션에서 변경 내용을 데이터베이스에 기록하지 않고 변경 로그를 비움으로써 업데이트를 취소할 수 있습니다. *CancelUpdates* 메소드를 호출하여 업데이트를 취소할 수 있습니다. 캐시의 삭제된 모든 레코드는 삭제되지 않은 상태로 되고 수정된 레코드는 원래 값으로 되돌아가며 새로 삽입된 레코드는 없어져 버립니다.

업데이트 캐싱을 위한 데이터셋 타입 선택

C++Builder에는 업데이트 캐싱을 위한 몇 가지 특화된 클라이언트 데이터셋 컴포넌트가 있습니다. 각 클라이언트 데이터셋은 특정 데이터 액세스 메커니즘과 연결되어 있습니다. 이 내용은 표 27.3에 나열되어 있습니다.

표 27.3 업데이트 캐싱을 위한 특화된 클라이언트 데이터셋

클라이언트 데이터셋	데이터 액세스 메커니즘
TBDEClientDataSet	Borland Database Engine
TSQLClientDataSet	dbExpress
TIBClientDataSet	InterBase Express

그리고 외부 프로바이더와 소스 데이터셋과 함께 일반적인 클라이언트 데이터셋(*TClientDataSet*)을 사용하여 업데이트를 캐싱할 수 있습니다. 외부 프로바이더와 함께 *TClientDataSet*을 사용하는 방법에 대한 자세한 내용은 27-24페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

참고 각 데이터 액세스 메커니즘과 연결된 특화된 클라이언트 데이터셋은 실제로 프로바이더와 소스 데이터셋도 사용합니다. 하지만 프로바이더와 소스 데이터셋은 모두 클라이언트 데이터셋의 내부에 있습니다.

업데이트 내용을 캐싱하는 가장 간단한 방법은 특화된 클라이언트 데이터셋 중 하나를 사용하는 것입니다. 하지만 다음과 같이 외부 프로바이더와 함께 *TClientDataSet*을 사용하는 것이 더 나은 경우도 있습니다.

- 특화된 클라이언트 데이터셋이 없는 데이터 액세스 메커니즘을 사용하는 경우 외부 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용해야 합니다. 예를 들어, 데이터가 XML 문서나 사용자 정의 데이터셋에서 온 경우입니다.
- 마스터/디테일 관계와 관련된 테이블 작업을 하고 있는 경우 *TClientDataSet*을 사용하여 프로바이더를 통해 마스터/디테일 관계에 연결된 두 개의 소스 데이터셋의 마스터 테이블에 연결해야 합니다. 클라이언트 데이터셋은 디테일 데이터셋을 중첩 데이터셋 필드로 봅니다. 마스터와 디테일 테이블에 대한 업데이트가 올바른 순서로 적용될 수 있으려면 이러한 접근 방법이 필요합니다.
- 클라이언트 데이터셋과 프로바이더 간 통신에 응답하는 이벤트 핸들러를 코딩하는 경우(예를 들어, 클라이언트 데이터셋에서 프로바이더로부터 레코드를 가져오기 전이나 직후) 외부 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용해야 합니다. 특화된 클라이언트 데이터셋은 업데이트를 적용하는 가장 중요한 이벤트(*OnReconcileError*, *BeforeUpdateRecord* 및 *OnGetTableName*)를 게시하지만 클라이언트 데이터셋과 프로바이더 사이의 통신에 관한 이벤트는 주로 멀티 티어 애플리케이션을 위한 것이므로 게시하지 않습니다.
- BDE를 사용할 때 업데이트 객체를 사용해야 하면 외부 프로바이더와 소스 데이터셋을 사용하고 싶을 것입니다. *TBDEClientDataSet*의 *BeforeUpdateRecord* 이벤트 핸들러에서 업데이트 객체를 코딩할 수 있지만 소스 데이터셋의 *UpdateObject* 속성을 할당하는 것이 더 간단할 수 있습니다. 업데이트 객체 사용에 대한 자세한 내용은 24-39페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

수정된 레코드 표시

사용자가 클라이언트 데이터셋을 편집할 때 처리된 편집 내용에 대해 피드백을 제공하는 것이 좋습니다. 특히 사용자가 편집된 내용을 찾아 "Undo" 버튼을 클릭하여 특정 편집 작업을 취소하는 것을 허용하고자 할 때 특히 유용합니다.

업데이트가 발생한 경우에 피드백을 제공할 때 *UpdateStatus* 메소드와 *StatusFilter* 속성을 사용하는 것이 다음과 같이 유용합니다.

- *UpdateStatus*는 업데이트가 있으면 현재 레코드에 대해 발생한 업데이트 타입을 표시합니다. 이 타입은 다음 값 중 하나가 될 수 있습니다.
  - *usUnmodified*는 현재 레코드가 변경되지 않았음을 표시합니다.
  - *usModified*는 현재 레코드가 편집되었음을 표시합니다.
  - *usInserted*는 사용자가 삽입한 레코드를 표시합니다.
  - *usDeleted*는 사용자가 삭제한 레코드를 표시합니다.

- *StatusFilter*는 변경 로그에서 볼 수 있는 업데이트 타입을 제어합니다. *StatusFilter*는 일반적인 데이터에서 필터가 작동하는 것과 같은 방식으로 캐싱된 레코드에서 작동합니다. *StatusFilter*는 집합이므로 다음과 같은 값의 어떠한 조합도 포함할 수 있습니다.
  - *usUnmodified*는 수정되지 않은 레코드를 표시합니다.
  - *usModified*는 수정된 레코드를 표시합니다.
  - *usInserted*는 삽입된 레코드를 표시합니다.
  - *usDeleted*는 삭제된 레코드를 표시합니다.

  디폴트로, *StatusFilter*는 [*usModified*, *usInserted*, *usUnmodified*] 집합입니다. *usDeleted*를 이 집합에 추가하여 삭제된 레코드에 대한 피드백도 제공할 수 있습니다.

참고 *UpdateStatus*와 *StatusFilter*도 *BeforeUpdateRecord*와 *OnReconcileError* 이벤트 핸들러에서 유용합니다. *BeforeUpdateRecord*에 대한 자세한 내용은 27-21페이지의 "업데이트 적용 시 간섭"을 참조하십시오. *OnReconcileError*에 대한 자세한 내용은 27-22페이지의 "업데이트 오류 조정"을 참조하십시오.

다음 예제에서는 *UpdateStatus* 메소드를 사용하여 레코드의 업데이트 상태에 관한 피드백을 제공하는 방법을 보여 줍니다. 삭제된 레코드가 데이터셋에 보이는 상태로 남아 있도록 *usDeleted*를 포함하게 *StatusFilter* 속성을 변경했다고 가정합니다. 그리고 계산된 필드를 "Status"라는 데이터셋에 추가했다고 가정합니다.

```
void __fastcall TForm1::ClientDataSet1CalcFields(TDataSet *DataSet)
{
  switch (DataSet->UpdateStatus())
  {
    case usUnmodified:
      ClientDataSet1Status->Value = NULL; break;
    case usModified:
      ClientDataSet1Status->Value = "M"; break;
    case usInserted:
      ClientDataSet1Status->Value = "I"; break;
    case usDeleted:
      ClientDataSet1Status->Value = "D"; break;
  }
}
```

레코드 업데이트

변경 로그의 내용은 클라이언트 데이터셋의 *Delta* 속성에 데이터 패킷으로 저장됩니다. *Delta*의 변경 내용을 영구적으로 만들려면 클라이언트 데이터셋에서 변경 내용을 데이터베이스나 소스 데이터셋 또는 XML 문서에 적용해야 합니다.

클라이언트는 다음과 같은 단계를 따라 서버에 업데이트를 적용합니다.

1 클라이언트 애플리케이션은 클라이언트 데이터셋 객체의 *ApplyUpdates* 메소드를 호출합니다. 이 메소드는 클라이언트 데이터셋의 *Delta* 속성 내용을 내부나 외부 프로바이더에 전달합니다. *Delta*는 클라이언트 데이터셋의 업데이트, 삽입 및 삭제된 레코드를 포함하는 데이터 패킷입니다.

2 프로바이더에서는 자체적으로 해결할 수 없는 문제 레코드는 캐싱하면서 업데이트를 적용합니다. 프로바이더에서 업데이트를 적용하는 방법에 대한 자세한 내용은 28-8페이지의 "클라이언트 업데이트 요청에 대한 응답"을 참조하십시오.

3 프로바이더는 해결하지 못한 모든 레코드를 *Result* 데이터 패킷의 클라이언트 데이터셋에 반환합니다. *Result* 데이터 패킷에는 업데이트되지 않은 레코드가 모두 있습니다. 또한 오류 메시지와 오류 코드 같은 오류 정보도 포함합니다.

4 클라이언트 데이터셋은 레코드별로 *Result* 데이터 패킷에 반환된 업데이트 오류의 해결을 시도합니다.

업데이트 적용

클라이언트 데이터셋의 데이터 로컬 복사본에 대한 변경 내용은 클라이언트 애플리케이션에서 *ApplyUpdates* 메소드를 호출할 때까지 데이터베이스 서버나 XML 문서로 보내지지 않습니다. *ApplyUpdates*는 변경 로그의 변경 내용을 *Delta*라는 데이터 패킷으로 프로바이더에게 보냅니다. 대부분의 클라이언트 데이터셋을 사용할 때 프로바이더는 클라이언트 데이터셋의 내부에 있습니다.

*ApplyUpdates*는 *MaxErrors*라는 단일 매개변수를 사용하는데 이는 업데이트 프로세스를 중단할 때까지 프로바이더에서 허용할 수 있는 최대 오류 수를 나타냅니다. *MaxErrors*가 *0*이면 업데이트 오류가 발생하는 즉시 전체 업데이트 프로세스가 종료됩니다. 데이터베이스에 아무런 변경 내용도 적용되지 않으며 클라이언트 데이터셋의 변경 로그도 전혀 바뀌지 않습니다. *MaxErrors*가 *-1*인 경우에는 오류가 무제한으로 허용되며 성공적으로 적용되지 않은 모든 레코드가 변경 로그에 포함됩니다. *MaxErrors*가 양수이고 *MaxErrors*에서 허용하는 수보다 오류가 많이 발생한 경우 모든 업데이트는 중지됩니다. *MaxErrors*에서 지정된 수보다 오류가 작게 발생한 경우 성공적으로 적용된 모든 레코드는 클라이언트 데이터셋의 변경 로그에서 자동으로 지워집니다.

*ApplyUpdates*는 실제로 발생한 오류의 수를 반환하는데 이 수는 항상 *MaxErrors* + 1보다 작거나 같습니다. 이 반환 값은 데이터베이스에 기록되지 못한 레코드의 수를 나타냅니다.

클라이언트 데이터셋의 *ApplyUpdates* 메소드는 다음 작업을 수행합니다.

1 프로바이더의 *ApplyUpdates* 메소드를 간접적으로 호출합니다. 프로바이더의 *ApplyUpdates* 메소드는 데이터베이스나 소스 데이터셋 또는 XML 문서에 업데이트를 기록하고 발생한 오류를 수정하려고 합니다. 오류 조건으로 인해 적용될 수 없는 레코드는 클라이언트 데이터셋으로 다시 보내집니다.

2 그리고 나서 클라이언트 데이터셋의 *ApplyUpdates* 메소드에서는 *Reconcile* 메소드를 호출하여 문제 레코드를 해결하려고 시도합니다. *Reconcile*은 *OnReconcileError* 이벤트 핸들러를 호출하는 오류 처리 루틴입니다. 오류를 수정하려면 *OnReconcileError* 이벤트 핸들러를 직접 코딩해야 합니다. *OnReconcileError* 사용에 대한 자세한 내용은 27-22페이지의 "업데이트 오류 조정"을 참조하십시오.

3 마지막으로 *Reconcile*은 성공적으로 적용된 변경 내용을 변경 로그에서 제거하고 새로 업데이트된 레코드를 반영하도록 *Data*를 업데이트합니다. *Reconcile*을 완료하면 *ApplyUpdates*는 발생한 오류 수를 보고합니다.

중요 경우에 따라 프로바이더에서는 업데이트를 적용하는 방법을 결정하지 못할 수가 있습니다. 내장 프로시저나 다중 테이블 조인의 업데이트를 적용하는 경우를 예로 들 수 있습니다. 클라이언트 데이터셋과 프로바이더 컴포넌트는 이러한 상황을 처리하는 이벤트를 생성합니다. 자세한 내용은 다음에 나오는 "업데이트 적용 시 간섭"을 참조하십시오.

팁 프로바이더가 stateless 애플리케이션 서버에 있는 경우 업데이트를 적용하기 전후에 영구적 상태 정보에 대해 프로바이더와 통신하기를 원할 수도 있습니다. *TClientDataSet*은 업데이트 내용을 전송하기 전에 *BeforeApplyUpdates* 이벤트를 받으므로 사용자가 서버로 영구적 상태 정보를 보낼 수 있게 해줍니다. 업데이트가 적용된 다음, 하지만 조정 프로세스(reconcile process) 전에 *TClientDataSet* 은 애플리케이션 서버에서 반환하는 영구적 상태 정보에 응답할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.

업데이트 적용 시 간섭

클라이언트 데이터셋에서 업데이트 내용을 적용할 때 프로바이더에서는 데이터베이스 서버나 소스 데이터셋에 삽입, 삭제, 수정 내용을 기록하는 방법을 결정합니다. 외부 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용하는 경우 해당 프로바이더의 속성과 이벤트를 사용하여 업데이트 내용을 적용하는 방법에 영향을 미칠 수 있습니다. 이 내용은 28-8페이지의 "클라이언트 업데이트 요청에 대한 응답"에 설명되어 있습니다.

프로바이더가 내부에 있는 경우 데이터 액세스 메커니즘과 연결된 클라이언트 데이터셋에 대한 것이므로 프로바이더의 속성을 설정하거나 이벤트 핸들러를 제공할 수 없습니다. 따라서 클라이언트 데이터셋은 내부 프로바이더의 업데이트 내용 적용 방법에 영향을 미칠 수 있게 해주는 한 개의 속성과 두 개의 이벤트를 게시합니다.

- *UpdateMode*는 업데이트 내용 적용을 위해 프로바이더에서 생성하는 SQL 문에서 레코드를 찾을 때 사용되는 필드를 제어합니다. *UpdateMode*는 프로바이더의 *UpdateMode* 속성과 동일합니다. 프로바이더의 *UpdateMode* 속성에 대한 자세한 내용은 28-9페이지의 "업데이트 적용 방법 변경"을 참조하십시오.
- *OnGetTableName*을 사용하면 업데이트를 적용해야 할 데이터베이스 테이블 이름을 프로바이더에 제공할 수 있습니다. 이를 통해 프로바이더는 *CommandText*에 의해 지정된 내장 프로시저나 쿼리에서 데이터베이스 테이블을 식별할 수 없을 때 업데이트에 대한 SQL 문을 생성합니다. 예를 들면, 하나의 테이블에만 업데이트가 필요한 다중 테이블 조인을 쿼리에서 실행하는 경우 *OnGetTableName* 이벤트 핸들러를 제공하면 내부 프로바이더에서 정확하게 업데이트를 적용할 수 있습니다.

  OnGetTableName 이벤트 핸들러에는 내부 프로바이더 컴포넌트, 서버에서 데이터를 가져올 수 있는 내부 데이터셋 그리고 생성된 SQL에서 사용할 테이블 이름을 반환하는 매개변수 등 세 개의 매개변수가 있습니다.
- *BeforeUpdateRecord*는 델타 패킷의 모든 레코드에 대해 발생합니다. 이 이벤트를 통해 레코드를 삽입, 삭제 또는 수정하기 전에 마지막 변경을 가능하게 합니다. 또한 프로바이더가 올바른 SQL을 생성할 수 없는 경우(예를 들어, 여러 테이블이 업데이트되어야 하는 다중 테이블 조인)에 업데이트를 적용할 사용자가 만든 SQL 문을 실행하는 방법을 제공합니다.

  BeforeUpdateRecord 이벤트 핸들러에는 내부 프로바이더 컴포넌트, 서버에서 데이터를 가져오는 내부 데이터셋, 바로 업데이트될 레코드에 위치한 델타 패킷, 업데이트가 삽입, 삭제, 수정 중 어느 것인지에 대한 표시 및 이벤트 핸들러에서 업데이트를 수행했는지 여부를 반환하는 매개변수 등 5개의 매개변수가 있습니다. 이러한 매개변수의 사용에 대해서는 다음 이벤트 핸들러에서 설명합니다. 간단히 하기 위해 다음 예제에서는 SQL 문을 필드 값만 필요로 하는 전역 변수로 사용할 수 있다고 가정합니다.

```
void __fastcall TForm1::SQLClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind
UpdateKind, bool &Applied)
{
  TSQLConnection *pConn := (dynamic_cast<TCustomSQLDataSet *>(SourceDS)-
>SQLConnection);
  char buffer[256];
  switch (UpdateKind)
  case ukModify:
    // 1st dataset: update Fields[1], use Fields[0] in where clause
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[1]->NewValue,
            DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: update Fields[2], use Fields[3] in where clause
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
  case ukDelete:
    // 1st dataset: use Fields[0] in where clause
    sprintf(buffer, DeleteStmt1, DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: use Fields[3] in where clause
    sprintf(buffer, DeleteStmt2, DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
  case ukInsert:
    // 1st dataset: values in Fields[0] and Fields[1]
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[0]->NewValue,
            DeltaDS->Fields->Fields[1]->NewValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: values in Fields[2] and Fields[3]
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->NewValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
}
```

업데이트 오류 조정

업데이트 프로세스 중에 발생하는 오류를 처리할 수 있는 이벤트는 다음과 같이 두 가지입니다.

- 업데이트 프로세스 중에 내부 프로바이더는 처리할 수 없는 업데이트가 발생할 때마다 *OnUpdateError* 이벤트를 생성합니다. *OnUpdateError* 이벤트 핸들러의 문제를 수정하고 나면 *ApplyUpdates* 메소드에 전달될 최대 오류 수에 해당 오류는 포함되지 않습니다. 이 이벤트는 내부 프로바이더를 사용하는 클라이언트 데이터셋에 대해서만 발생합니다. *TClientDataSet*을 사용하고 있으면 대신 프로바이더 컴포넌트의 *OnUpdateError* 이벤트를 사용할 수 있습니다.
- 전체 업데이트 작업이 끝나면 클라이언트 데이터셋에서는 프로바이더에서 데이터베이스 서버에 적용할 수 없는 각 레코드에 대해 *OnReconcileError* 이벤트를 생성합니다.

적용되지 못해 반환된 레코드를 버리게 되더라도 *OnReconcileError*나 *OnUpdateError* 이벤트 핸들러를 항상 코딩해야 합니다. 이러한 두 개의 이벤트에 대한 이벤트 핸들러는 같은 방식으로 작동합니다. 그리고 다음과 같은 매개변수가 포함됩니다.

- *DataSet:* 적용될 수 없는 업데이트된 레코드를 포함하는 클라이언트 데이터셋. 클라이언트 데이터셋 메소드를 사용하여 문제 레코드에 대한 정보를 얻고 문제를 해결하기 위해 레코드를 변경할 수 있습니다. 특히 현재 레코드에 있는 필드의 *CurValue*, *OldValue* 및 *NewValue* 속성을 사용하여 업데이트 문제의 원인을 확인할 수 있습니다. 그러나 이벤트 핸들러의 현재 레코드를 변경하는 클라이언트 데이터셋 메소드를 호출하면 안됩니다.
- *E:* 발생한 문제를 나타내는 객체. 이 예외를 사용하여 오류 메시지를 추출하거나 업데이트 오류의 원인을 알아낼 수 있습니다.
- *UpdateKind:* 오류를 생성한 업데이트의 종류. *UpdateKind*는 *ukModify*(수정된 기존 레코드를 업데이트할 때 발생하는 문제), *ukInsert*(새 레코드를 삽입할 때 발생하는 문제) 또는 *ukDelete*(기존 레코드를 삭제할 때 발행하는 문제) 중의 하나입니다.
- *Action:* 이벤트 핸들러가 존재할 때 취할 동작을 표시하는 참조 매개변수. 이벤트 핸들러에서 이 매개변수를 설정하여 다음 작업을 수행합니다.
  - 변경 로그에 남겨둔 채로 이 레코드를 건너뜁니다(rrSkip 또는 raSkip).
  - 모든 조정 작업을 중지합니다(rrAbort 또는 raAbort).
  - 서버에서 해당 레코드에 실패한 수정을 병합합니다(rrMerge 또는 raMerge). 이 작업은 서버 레코드가 클라이언트 데이터셋의 레코드에서 수정된 필드에 변경 내용이 없는 경우에만 가능합니다.
  - 변경 로그의 현재 업데이트를 이미 수정되었을 이벤트 핸들러의 레코드 값으로 바꿉니다(rrApply 또는 raCorrect).
  - 오류를 완전히 무시합니다(rrIgnore). 이러한 경우는 *OnUpdateError* 이벤트 핸들러에서만 발생하며 이벤트 핸들러에서 업데이트를 데이터베이스 서버로 다시 적용하려고 하는 경우입니다. 프로바이더에서 업데이트를 적용했던 것처럼 업데이트된 레코드를 변경 로그에서 제거하여 *Data*로 병합합니다.
  - 클라이언트 데이터셋에서 이 레코드의 변경 내용을 원래 제공된 값으로 복귀하여 되돌립니다(raCancel). 이러한 경우는 *OnReconcileError* 이벤트 핸들러에서만 발생합니다.
  - 현재 레코드 값을 업데이트하여 서버의 레코드와 일치하도록 합니다(raRefresh). 이러한 경우는 *OnReconcileError* 이벤트 핸들러에서만 발생합니다.

다음 코드는 objrepos 디렉토리에 있는 RecError 유닛에서 Reconcile Error 다이얼로그 박스를 사용하는 *OnReconcileError* 이벤트 핸들러를 보여 줍니다. 이 다이얼로그 박스를 사용하려면 개발자 소스 유닛에 RecError.hpp를 포함해야 합니다.

```
void __fastcall TForm1::ClientDataSetReconcileError(TCustomClientDataSet
*DataSet,
    EReconcileError *E, TUpdateKind UpdateKind, TReconcileAction &Action)
{
  Action = HandleReconcileError(this, DataSet, UpdateKind, E);
}
```

프로바이더와 함께 클라이언트 데이터셋 사용

클라이언트 데이터셋은 프로바이더를 사용하여 다음과 같은 경우에 프로바이더에 데이터를 제공하고 업데이트를 적용합니다.

- 데이터베이스 서버나 다른 데이터셋에서 업데이트를 캐싱한 경우
- XML 문서의 데이터를 나타내는 경우
- 멀티 티어 애플리케이션의 클라이언트 부분에 데이터를 저장하는 경우

TClientDataSet 이외의 클라이언트 데이터셋의 경우 이러한 프로바이더는 내부에 있으므로 애플리케이션에서 직접 액세스할 수 없습니다. *TClientDataSet*을 사용하면 프로바이더는 클라이언트 데이터셋을 데이터의 외부 소스에 연결하는 외부 컴포넌트가 됩니다.

외부 프로바이더 컴포넌트는 클라이언트 데이터셋과 동일한 애플리케이션에 있거나 다른 시스템에서 실행 중인 독립적인 애플리케이션의 일부일 수 있습니다. 프로바이더 컴포넌트에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오. 프로바이더가 다른 시스템의 독립적인 애플리케이션에 있는 애플리케이션에 대한 자세한 내용은 29장, "멀티 티어 애플리케이션 생성"을 참조하십시오.

내부 또는 외부 프로바이더를 사용할 때 클라이언트 데이터셋은 항상 업데이트를 캐싱합니다. 이러한 캐싱 방법에 대한 자세한 내용은 27-15페이지의 "업데이트 내용을 캐싱하기 위해 클라이언트 데이터셋 사용"을 사용하십시오.

다음 주제에서는 프로바이더와 함께 클라이언트 데이터셋이 작동할 수 있게 해주는 클라이언트 데이터셋의 추가 속성과 메소드를 설명합니다.

프로바이더 지정

데이터 액세스 메커니즘과 연관된 클라이언트 데이터셋과는 달리 *TClientDataSet*에는 데이터를 패키지하거나 업데이트를 적용하는 내부 프로바이더 컴포넌트가 없습니다. TClientDataSet에서 소스 데이터셋이나 XML 문서의 데이터를 나타내게 하려면 클라이언트 데이터셋을 외부 프로바이더 컴포넌트와 연결해야 합니다.

프로바이더를 *TClientDataSet*에 연결하는 방법은 프로바이더가 클라이언트 데이터셋과 동일한 애플리케이션에 있는지 또는 다른 시스템에서 실행 중인 원격 애플리케이션 서버에 있는지에 따라 다릅니다.

- 프로바이더가 클라이언트 데이터셋과 동일한 애플리케이션에 있으면 Object Inspector의 *ProviderName* 속성에 대한 드롭다운 리스트에서 프로바이더를 선택하여 TClientDataSet을 프로바이더와 연결할 수 있습니다. 프로바이더가 클라이언트 데이터셋과 같은 *Owner*를 가졌을 때만 연결할 수 있습니다. 클라이언트 데이터셋과 프로바이더는 동일한 폼이나 데이터 모듈에 있는 경우 동일한 *Owner*를 가집니다. 다른 *Owner*를 갖는 로컬 프로바이더를 사용하려면 클라이언트 데이터셋의 *SetProvider* 메소드를 사용하여 런타임 시 해당 프로바이더를 연결해야 합니다.

결국 원격 프로바이더로 범위를 확장하거나 *IAppServer* 인터페이스로 직접 호출하려면 *RemoteServer* 속성을 *TLocalConnection* 컴포넌트로 설정할 수도 있습니다. *TLocalConnection* 을 사용하면 *TLocalConnection* 인스턴스는 애플리케이션에 있는 모든 프로바이더 리스트를 관리하고 클라이언트 데이터셋의 *IAppServer* 호출을 처리합니다. *TLocalConnection*을 사용하지 않으면 애플리케이션에서 클라이언트 데이터셋의 *IAppServer* 호출을 처리하는 숨겨진 객체를 만듭니다.

- 프로바이더가 원격 애플리케이션 서버에 있으면 *ProviderName* 속성 이외에도 클라이언트 데이터셋을 애플리케이션 서버에 연결하는 컴포넌트를 지정해야 합니다. 이러한 작업을 처리할 수 있는 속성으로는 프로바이더 리스트를 얻을 수 있는 연결 컴포넌트의 이름을 지정하는 *RemoteServer*와 클라이언트 데이터셋과 연결 컴포넌트 사이의 추가 레벨의 간접 참조를 제공하는 중앙 브로커를 지정하는 *ConnectionBroker*, 두 가지가 있습니다. 연결 브로커가 사용된다면 연결 브로커와 연결 컴포넌트는 클라이언트 데이터셋과 같은 데이터 모듈에 있습니다. 연결 컴포넌트는 애플리케이션 서버에 대한 연결을 만들고 유지 보수하므로 "데이터 브로커"라고 부르기도 합니다. 자세한 내용은 29-4페이지의 "클라이언트 애플리케이션의 구조"를 참조하십시오.

  디자인 타임 시 *RemoteServer*나 *ConnectionBroker*를 지정한 다음 Object Inspector의 *ProviderName* 속성의 드롭다운 리스트에서 프로바이더를 선택할 수 있습니다. 이 리스트에는 연결 컴포넌트를 통해 액세스할 수 있는 원격 프로바이더와 동일한 폼 또는 데이터 모듈의 로컬 프로바이더가 모두 있습니다.

참고 연결 컴포넌트가 *TDCOMConnection*의 인스턴스인 경우 애플리케이션 서버는 클라이언트 컴퓨터에 등록되어 있어야 합니다.

런타임 시 코드에 *ProviderName*을 설정하여 사용 가능한 로컬 프로바이더와 원격 프로바이더 사이를 전환할 수 있습니다.

소스 데이터셋이나 문서에서 데이터 요청

클라이언트 데이터셋은 프로바이더에서 데이터 패킷을 가져오는 방법을 제어할 수 있습니다. 디폴트로, 클라이언트 데이터셋은 소스 데이터셋과 프로바이더가 내부 컴포넌트 (*TBDEClientDataSet*, *TSQLClientDataSet* 및 *TIBClientDataSet* 등)이든 *TClientDataSet*에 데이터를 제공하는 독립적인 컴포넌트이든 상관 없이 소스 데이터셋의 모든 레코드를 가져옵니다.

클라이언트 데이터셋에서 *PacketRecords* 와 *FetchOnDemand* 속성을 가져오는 방법을 변경할 수 있습니다.

증분 페치

PacketRecords 속성을 변경하여 클라이언트 데이터셋에서 데이터를 좀 더 작은 조각으로 가져오도록 지정할 수 있습니다. *PacketRecords* 는 한 번에 가져오는 레코드 수 또는 반환하는 레코드의 타입을 지정합니다. 디폴트로 *PacketRecords*는 *-1*로 설정되며, 이는 클라이언트 데이터셋이 처음 열릴 때나 애플리케이션이 명시적으로 *GetNextPacket* 을 호출할 때 사용 가능한 모든 레코드를 한꺼번에 가져온다는 것을 의미합니다. *PacketRecords*가 *-1*인 경우 클라이언트 데이터셋이 데이터를 처음 가져오고 나면 이미 사용 가능한 모든 레코드가 있기 때문에 더 이상 데이터를 가져올 필요가 없습니다.

작은 배치(batch) 단위로 레코드를 가져오려면 *PacketRecords*를 가져올 레코드 수로 설정합니다. 예를 들어, 다음 명령문은 각 데이터 패킷의 크기를 10개 레코드로 설정합니다.

```
ClientDataSet1->PacketRecords = 10;
```

배치로 레코드를 가져오는 이러한 프로세스를 "증분 페치(incremental fetch)"라고 합니다. 클라이언트 데이터셋은 *PacketRecords*가 0보다 크면 증분 페치를 사용합니다.

레코드의 각 배치를 가져오기 위해 클라이언트 데이터셋은 *GetNextPacket*을 호출합니다. 새로 가져온 패킷은 클라이언트 데이터셋에 이미 존재하는 데이터의 끝에 추가됩니다. *GetNextPacket*은 가져온 레코드 수를 반환합니다. 반환 값이 *PacketRecords*와 동일한 경우 사용 가능한 레코드의 끝에 도달하지 않습니다. 반환 값이 *0*보다 크고 *Packet Records*보다 작은 경우 마지막 레코드는 가져오기 작업 중에 도착합니다. *GetNextPacket*이 *0*을 반환하면 더 이상 가져올 레코드가 없습니다.

경고 증분 페치는 stateless 애플리케이션 서버의 원격 프로바이더로부터 데이터를 가져오는 경우 작동하지 않습니다. stateless 원격 데이터 모듈과 함께 증분 페치를 사용하는 방법에 대한 자세한 내용은 29-18페이지의 "원격 데이터 모듈에서 상태 정보 지원"을 참조하십시오.

참고 또한 *PacketRecords*를 사용하여 소스 데이터셋에 대한 메타데이터 정보를 가져올 수 있습니다. 메타데이터 정보를 검색하려면 *PacketRecords*를 *0*으로 설정합니다.

요청 시 가져오기

레코드의 자동 페치는 *FetchOnDemand* 속성에서 제어합니다. *FetchOnDemand*가 **true**(기본값)이면 클라이언트 데이터셋에서 자동으로 필요한 만큼 레코드를 가져옵니다. 레코드의 자동 페치를 막으려면 *FetchOnDemand*를 **false**로 설정합니다. *FetchOnDemand*가 **false**인 경우 애플리케이션은 레코드를 가져오기 위해 *GetNextPacket*을 명시적으로 호출해야 합니다.

예를 들면, 대량의 읽기 전용 데이터셋을 나타내야 하는 애플리케이션의 경우 *FetchOnDemand*를 해제하여 클라이언트 데이터셋에서 메모리를 초과하는 데이터를 로드하지 않게 할 수 있습니다. 가져오기 작업들 사이에 클라이언트 데이터셋은 *EmptyDataSet* 메소드를 사용하여 캐시를 해제합니다. 그러나 클라이언트가 서버에 업데이트를 포스트해야 하는 경우에는 이 방법이 잘 작동하지 않습니다.

프로바이더에서는 데이터 패킷의 레코드가 BLOB 데이터와 중첩 디테일 데이터셋을 포함할지 여부를 제어합니다. 프로바이더가 이 정보를 포함하지 않으면 *FetchOnDemand* 속성은 클라이언트 데이터셋이 필요에 따라 BLOB 데이터와 디테일 데이터셋을 자동으로 가져오게 합니다. *FetchOnDemand*가 **false**이고 프로바이더에서 레코드가 있는 디테일 데이터셋과 BLOB 데이터를 포함하고 있지 않으면 명시적으로 *FetchBlobs*나 *FetchDetails* 메소드를 호출하여 이 정보를 검색해야 합니다.

소스 데이터셋에서 매개변수 얻기

클라이언트 데이터셋에서 매개변수 값을 가져와야 하는 상황은 다음 두 가지입니다.

- 애플리케이션에서 내장 프로시저의 출력 매개변수 값이 필요한 경우입니다.
- 애플리케이션에서 쿼리나 내장 프로시저의 입력 매개변수를 소스 데이터셋의 현재 값으로 초기화하려고 합니다.

클라이언트 데이터셋은 매개변수 값을 *Params* 속성에 저장합니다. 클라이언트 데이터셋이 소스 데이터셋에서 데이터를 가져올 때 이러한 값은 출력 매개변수와 함께 새로 고쳐집니다. 그러나 클라이언트 애플리케이션의 *TClientDataSet* 컴포넌트가 데이터를 가져오지 않는 경우에도 출력 매개변수가 필요할 때가 있습니다.

레코드를 가져오지 않을 때 출력 매개변수를 가져오거나 입력 매개변수를 초기화하기 위해 클라이언트 데이터셋은 *FetchParams* 메소드를 호출하여 소스 데이터셋의 매개변수 값을 요청할 수 있습니다. 매개변수는 프로바이더로부터 데이터 패킷에 반환되고 클라이언트 데이터셋의 *Params* 속성에 할당됩니다.

디자인 타임 시 *Params* 속성은 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Fetch Params를 선택하여 초기화할 수 있습니다.

참고 *Params* 속성은 항상 내부 소스 데이터셋의 매개변수를 반영하므로 클라이언트 데이터셋에서 내부 프로바이더와 소스 데이터셋을 사용할 때 *FetchParams*를 호출할 필요가 없습니다. *TClientDataSet*과 함께 *FetchParams* 메소드나 Fetch Params 명령은 연결된 데이터셋이 매개변수를 제공할 수 있는 프로바이더에 클라이언트 데이터셋이 연결된 경우에만 작동합니다. 예를 들어, 소스 데이터셋이 테이블 타입 데이터셋이면 가져올 매개변수가 없습니다.

프로바이더가 stateless 애플리케이션 서버의 일부로서 독립적인 시스템에 있는 경우 *FetchParams*를 사용하여 출력 매개변수를 검색할 수 없습니다. stateless 애플리케이션 서버에서 다른 클라이언트는 *FetchParams*로의 호출 전에 출력 매개변수를 변경하여 쿼리나 내장 프로시저를 변경하고 재실행할 수 있습니다. stateless 애플리케이션 서버에서 출력 매개변수를 검색하려면 *Execute* 메소드를 사용합니다. 프로바이더가 쿼리나 내장 프로시저에 연결되어 있다면 *Execute*는 프로바이더가 쿼리나 내장 프로시저를 실행하고 출력 매개변수를 반환하게 합니다. 이렇게 반환되는 매개변수는 자동으로 *Params* 속성을 업데이트하는 데 사용됩니다.

소스 데이터셋에 매개변수 전달

클라이언트 데이터셋은 데이터 패킷으로 받기 원하는 데이터를 지정하기 위해서 소스 데이터셋에 매개변수를 전달할 수 있습니다. 이런 매개변수는 다음 내용을 지정할 수 있습니다.

- 애플리케이션 서버에서 실행되는 쿼리나 내장 프로시저의 입력 매개변수 값
- 데이터 패킷에 보낼 레코드를 제한하는 필드 값

디자인 타임이나 런타임 시 클라이언트 데이터셋에서 소스 데이터셋에 보내는 매개변수 값을 지정할 수 있습니다. 디자인 타임시 클라이언트 데이터셋을 선택하고 Object Inspector의 *Params* 속성을 두 번 클릭합니다. 이렇게 하면 매개변수를 추가, 삭제 또는 재배치할 수 있는 Collection Editor가 나타납니다. Collection Editor에서 매개변수를 선택하여 Object Inspector를 이용해서 해당 매개변수의 속성을 편집할 수 있습니다.

런타임 시 *Params* 속성의 *CreateParam* 메소드를 사용하면 매개변수를 클라이언트 데이터셋에 추가할 수 있습니다. *CreateParam*은 매개변수 객체, 지정된 이름, 매개변수 타입 및 데이터 타입을 반환합니다. 그런 다음 해당 매개변수 객체의 속성을 사용하여 매개변수에 값을 할당합니다.

예를 들어, 다음 코드에서는 값이 605인 CustNo라는 입력 매개변수를 추가합니다.

```
TParam *pParam = ClientDataSet1->Params->CreateParam(ftInteger, "CustNo",
ptInput);
pParam->AsInteger = 605;
```

클라이언트 데이터셋이 활성화되어 있지 않으면 *Active* 속성을 **true**로 설정하여 매개변수를 애플리케이션 서버로 보낸 다음 매개변수 값을 반영하는 데이터 패킷을 검색할 수 있습니다.

쿼리나 내장 프로시저 매개변수 보내기

클라이언트 데이터셋의 *CommandType* 속성이 *ctQuery* 또는 *ctStoredProc*이거나 클라이언트 데이터셋이 *TClientDataSet* 인스턴스인 경우 연결된 프로바이더에서 쿼리나 내장 프로시저의 결과를 나타내고 있으면 *Params* 속성을 사용하여 매개변수 값을 지정할 수 있습니다. 클라이언트 데이터셋에서 소스 데이터셋의 데이터를 요청 하거나 데이터셋을 반환하지 않는 쿼리나 내장 프로시저를 실행하는 *Execute* 메소드를 사용하는 경우, 데이터 요청과 함께 이러한 매개변수 값이 전달되거나 실행 명령이 전달됩니다. 프로바이더는 이 매개변수 값을 받으면 연결된 데이터셋에 할당합니다. 그런 다음 매개변수 값을 사용하여 쿼리나 내장 프로시저를 실행하도록 데이터셋에 지시하고 클라이언트 데이터셋이 데이터를 요청한 경우 결과 셋의 첫 번째 레코드로 시작하는 데이터를 제공하기 시작합니다.

참고 매개변수 이름은 소스 데이터셋의 해당 매개변수 이름과 일치해야 합니다.

매개변수로 레코드 제한

클라이언트 데이터셋이 다음과 같은 경우

- 연결 프로바이더에서 *TTable*이나 *TSQLTable* 컴포넌트를 나타내는 *TClientDataSet* 인스턴스
- *CommandType* 속성이 *ctTable*인 *TSQLClientDataSet*나 *TBDEClientDataSet* 인스턴스

클라이언트 데이터셋은 *Params* 속성을 사용하여 메모리에서 캐싱하는 레코드를 제한할 수 있습니다. 각 매개변수는 클라이언트 데이터셋의 데이터에 레코드가 포함되기 전에 일치해야 하는 필드 값을 나타냅니다. 이러한 매개변수는 필터처럼 작동합니다. 하지만 필터의 경우 메모리에서 레코드를 캐싱할 수 있지만 이들 매개변수에서는 캐싱할 수 없습니다.

각 매개변수 이름은 필드 이름과 일치해야 합니다. *TClientDataSet*을 사용하는 경우 매개변수는 프로바이더와 연결된 *TTable*이나 *TSQLTable* 컴포넌트의 필드 이름이 되어야 합니다. *TSQLClientDataSet*이나 *TBDEClientDataSet*을 사용하는 경우 매개변수는 데이터베이스 서버 테이블의 필드 이름이 되어야 합니다. 그러면 클라이언트 데이터셋의 데이터는 해당 필드의 값이 매개변수에 할당된 값과 같은 레코드만 포함합니다.

예를 들어, 고객 한 명의 주문 내용을 보여주는 애플리케이션을 생각해 봅니다. 사용자가 고객 한 명을 식별할 때 클라이언트 데이터셋은 그 고객의 주문 정보만 보이도록 식별하는 값인 CustID라는 단일 매개변수를 포함하거나 소스 테이블의 한 필드가 호출되도록 *Params* 속성을 설정합니다. 클라이언트 데이터셋에서 소스 데이터셋의 데이터를 요청하면 이 매개변수 값이 전달됩니다. 그러면 프로바이더는 식별된 고객에 대한 레코드만 보냅니다. 이것은 프로바이더가 모든 주문 레코드를 클라이언트 애플리케이션에 보낸 다음 클라이언트 데이터셋을 사용하여 레코드를 필터링하는 것보다 훨씬 효율적입니다.

서버로부터의 제약 조건 처리

데이터베이스 서버에서 유효한 데이터를 규정하는 제약 조건을 정의하는 경우 클라이언트 데이터셋에서 이러한 제약 조건을 알고 있어야 유용합니다. 클라이언트 데이터셋에서 사용자 편집 내용이 이러한 서버 제약 조건을 위반하지 않도록 할 수 있기 때문입니다. 결과적으로 거부될 위반 내용은 데이터베이스 서버로 전달되지 않게 됩니다. 이는 업데이트가 뜸하면 업데이트 프로세스 동안 오류 조건이 발생한다는 의미입니다.

데이터 소스에 상관없이 서버 제약 조건을 클라이언트 데이터셋에 명시적으로 추가하여 해당 서버 제약 조건을 복제할 수 있습니다. 이 프로세스는 27-7페이지의 "사용자 정의 제약 조건 지정"에 설명되어 있습니다.

하지만 서버 제약 조건이 자동으로 데이터 패킷에 포함되면 더욱 편리합니다. 디폴트 표현식과 제약 조건을 명시적으로 지정하지 않아도 되고 서버 제약 조건이 변경될 때 강제로 적용되는 값을 클라이언트 데이터셋이 변경합니다. 디폴트로 다음과 같은 작업이 수행됩니다. 소스 데이터셋에서 서버 제약 조건을 인식하고 있으면 프로바이더는 자동으로 데이터 패킷에 제약 조건을 포함하고 사용자가 편집 내용을 변경 로그에 포스트하면 클라이언트 데이터셋에서는 제약 조건을 적용합니다.

참고 BDE를 사용하는 데이터셋만 서버에서 제약 조건을 임포트할 수 있습니다. 즉 BDE 기반 데이터셋을 나타내는 프로바이더와 함께 *TBDEClientDataSet*이나 *TClientDataSet*을 사용하는 경우 서버 제약 조건이 데이터 패킷에만 포함됩니다. 서버 제약 조건을 임포트하는 방법과 프로바이더에서 제약 조건을 데이터 패킷에 포함하지 못하게 하는 방법에 대한 자세한 내용은 28-12페이지의 "서버 제약 조건 처리"를 참조하십시오.

참고 임포트된 제약 조건을 사용하는 방법에 대한 자세한 내용은 23-21페이지의 "서버 제약 조건 사용"을 참조하십시오.

서버 제약 조건과 표현식을 임포트하는 것은 애플리케이션에서 데이터 무결성을 유지하도록 도와주는 매우 중요한 기능이지만 일시적으로 제약 조건을 사용할 수 없게 만들 수 있습니다. 예를 들어, 서버 제약 조건이 필드의 현재 최대 값에 기반을 둔 것이지만 클라이언트 데이터셋에서 증분 페치(incremental fetch)를 사용하고 클라이언트 데이터셋에 있는 필드의 현재 최대 값이 데이터베이스 서버의 최대 값과 달라 제약 조건이 다르게 실행되는 경우입니다. 다른 경우를 예를 들면, 제약 조건이 사용 가능할 때 클라이언트 데이터셋에서 필터를 레코드에 적용하면 필터는 의도하지 않게 제약 조건에 방해가 될 수 있습니다. 이러한 경우 애플리케이션에서 제약 조건 검사를 사용할 수 없게 만들 수 있습니다.

일시적으로 제약 조건을 사용할 수 없게 만들려면 *DisableConstraints* 메소드를 호출합니다. *DisableConstraints*가 호출될 때마다 참조 카운트가 증가됩니다. 참조 카운트가 0보다 크면 클라이언트 데이터셋에서 제약 조건이 실행되지 않습니다.

클라이언트 데이터셋의 제약 조건을 다시 사용 가능하게 만들려면 *EnableConstraints* 메소드를 호출합니다. *EnableConstraints*를 호출할 때마다 참조 카운트가 감소됩니다. 참조 카운트가 0이면 제약 조건이 다시 사용 가능하게 됩니다.

팁 의도한대로 제약 조건이 사용 가능하게 되었는지를 확인하려면 항상 *DisableConstraints*와 *EnableConstraints*를 쌍을 이루는 블록으로 호출하십시오.

레코드 새로 고침

클라이언트 데이터셋은 소스 데이터셋 데이터의 인메모리 스냅샷으로 작업합니다. 소스 데이터셋에서 서버 데이터를 나타내는 경우 시간이 경과되면 다른 사용자가 해당 데이터를 수정할 수도 있습니다. 그러면 클라이언트 데이터셋의 데이터는 원본으로 사용하는 데이터의 정확한 모습과는 점점 더 멀어집니다.

다른 데이터셋 같이 클라이언트 데이터셋은 서버의 현재 값과 일치하도록 레코드를 업데이트하는 *Refresh* 메소드를 가지고 있습니다. 그러나 *Refresh* 호출은 변경 로그에 편집 내용이 없는 경우에만 작동합니다. 적용되지 않은 편집 내용이 있을 때 *Refresh*를 호출하면 예외가 발생합니다.

클라이언트 데이터셋은 변경 로그는 그대로 남겨 둔 채로 데이터를 업데이트할 수도 있습니다. 이렇게 하려면 *RefreshRecord* 메소드를 호출하십시오. *Refresh* 메소드와 달리 *RefreshRecord*는 클라이언트 데이터셋의 현재 레코드만 업데이트합니다. *RefreshRecord*는 프로바이더에서 원래 얻은 레코드 값만 변경하고 변경 로그의 모든 변경 내용은 그냥 둡니다.

경고 *RefreshRecord*를 호출하는 것이 항상 적절한 것은 아닙니다. 사용자의 편집 내용과 원본으로 사용하는 데이터셋에 대한 다른 사용자의 변경 내용이 서로 충돌하는 경우 *RefreshRecord*를 호출하면 이러한 충돌 상황이 감춰집니다. 클라이언트 데이터셋에서 업데이트 내용을 적용하는 경우 조정 오류가 발생하지 않으므로 애플리케이션에서 충돌 문제를 해결할 수 없습니다.

업데이트 오류가 감춰지는 것을 방지하려면 *RefreshRecord*를 호출하기 전에 보류 중인 업데이트가 없는지 확인해야 합니다. 예를 들어, 다음 *AfterScroll*에서는 사용자가 새 레코드로 이동할 때마다 가장 최신 값이 되게 하면서 현재 레코드를 새로 고칩니다. 하지만 이러한 작업이 안전할 때만 합니다.

```
void __fastcall TForm1::ClientDataSet1AfterScroll(TDataSet *DataSet)
{
 if (ClientDataSet1->UpdateStatus == usUnModified)
    ClientDataSet1->RefreshRecord();
}
```

사용자 정의 이벤트를 사용하여 프로바이더와 통신

클라이언트 데이터셋은 *IAppServer*라는 특수 인터페이스를 통해 프로바이더 컴포넌트와 통신합니다. 로컬 프로바이더의 경우, *IAppServer*는 클라이언트 데이터셋과 프로바이더 사이의 모든 통신을 처리하는 자동으로 생성된 객체에 대한 인터페이스입니다. 원격 프로바이더의 경우, *IAppServer* 는 애플리케이션 서버에 있는 원격 데이터 모듈의 연결 COM 객체에 대한 인터페이스이거나 연결 컴포넌트에서 생성한 인터페이스(SOAP 서버의 경우)입니다.

*TClientDataSet*은 *IAppServer* 인터페이스를 사용하는 통신을 사용자 정의할 수 있는 기회를 많이 제공합니다. 클라이언트 데이터셋의 프로바이더에서 다이렉트되는 모든 *IAppServer* 메소드 호출 전후에 *TClientDataSet*은 프로바이더와 정보를 교환할 수 있게 하는 특별한 이벤트를 받습니다. 이러한 이벤트는 프로바이더의 유사한 이벤트와 일치합니다. 예를 들어, 클라이언트 데이터셋이 *ApplyUpdates* 메소드를 호출하면 다음 이벤트가 발생합니다.

1. 클라이언트 데이터셋은 *OwnerData*라는 OleVariant에서 임의의 사용자 정의 정보를 지정하는 *BeforeApplyUpdates* 이벤트를 받습니다.

2 프로바이더는 클라이언트 데이터셋의 *OwnerData*에 응답하고 *OwnerData*의 값을 새 정보로 업데이트할 수 있는 *BeforeApplyUpdates* 이벤트를 받습니다.

3 프로바이더는 데이터 패킷을 조합하는 정상 프로세스 및 이에 수반하는 이벤트를 모두 수행합니다.

4 프로바이더가 *OwnerData*의 현재 값에 응답하고 클라이언트 데이터셋의 값으로 업데이트할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.

5 클라이언트 데이터셋은 *OwnerData*의 반환된 값에 응답할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.

그 밖의 다른 *IAppServer* 메소드를 호출해도 클라이언트 데이터셋과 프로바이더 사이의 통신을 사용자 정의할 수 있게 하는 유사한 일련의 *BeforeXXX* 및 *AfterXXX* 이벤트가 발생합니다.

또한 클라이언트 데이터셋에는 프로바이더와 애플리케이션의 특정 통신만 허용하는 특수 메소드인 *DataRequest*가 있습니다. 클라이언트 데이터셋이 *DataRequest*를 호출하는 경우 원하는 모든 정보를 포함할 수 있는 OleVariant를 매개변수로 전달합니다. 이렇게 하면 프로바이더에 *OnDataRequest* 이벤트가 생성되므로 애플리케이션에서 정의한 방식으로 응답할 수 있고 값을 클라이언트 데이터셋에 반환할 수 있습니다.

소스 데이터셋 오버라이드

특정 데이터 액세스 메커니즘과 연결된 클라이언트 데이터셋에서는 *CommandText*와 *CommandType* 속성을 사용하여 자신이 나타내는 데이터를 지정합니다. 하지만 *TClientDataSet*을 사용하는 경우 데이터는 클라이언트 데이터셋이 아닌 소스 데이터셋에 의해 지정됩니다. 일반적으로 이 소스 데이터셋에는 데이터베이스 테이블이나 내장 프로시저의 이름 또는 데이터를 생성하는 SQL 문을 지정하는 속성이 있습니다.

프로바이더가 허용하면 *TClientDataSet*은 데이터가 나타내는 내용을 표시하는 소스 테이터셋의 속성을 오버라이드할 수 있습니다. 즉, 프로바이더에서 허용하면 클라이언트 데이터셋의 *CommandText* 속성은 자신이 나타내는 데이터를 지정하는 프로바이더의 데이터셋 속성을 대체합니다. 따라서 *TClientDataSet*에서는 보고자 하는 데이터를 동적으로 지정할 수 있게 됩니다.

디폴트로, 외부 프로바이더 컴포넌트는 클라이언트 데이터셋에서 *CommandText* 값을 이런 식으로 사용하지 못하게 합니다. *TClientDataSet*에서 *CommandText* 속성을 사용할 수 있게 하려면 *poAllowCommandText* 속성을 프로바이더의 *Options* 속성에 추가해야 합니다. 그렇지 않으면 *CommandText*의 값이 무시됩니다.

참고 *TSQLClientDataSet, TBDEClientDataSet* 또는 *TIBClientDataSet*의 *Options* 속성에서 *poAllowCommandText*를 제거하지 마십시오. 클라이언트 데이터셋의 *Options* 속성은 내부 프로바이더로 전달되므로 *poAllowCommandText*를 제거하면 클라이언트 데이터셋에서 액세스할 데이터를 지정하지 못합니다.

다음의 두 경우에 클라이언트 데이터셋은 *CommandText* 문자열을 프로바이더로 보냅니다.

- 클라이언트 데이터셋이 처음으로 열리는 경우. 클라이언트 데이터셋은 프로바이더로부터 첫 번째 데이터 패킷을 받은 이후에 다시 데이터 패킷을 가져올 때에는 *CommandText*를 보내지 않습니다.
- 클라이언트 데이터셋이 *Execute* 명령을 프로바이더로 보내는 경우

나중에 언제든 SQL 명령을 보내거나 테이블 또는 내장 프로시저 이름을 변경하려면 *AppServer* 속성으로 사용 가능한 *IAppServer* 인터페이스를 명시적으로 사용해야 합니다. 이 속성은 클라이언트 데이터셋이 프로바이더와 통신하는 인터페이스를 나타냅니다.

파일 기반 데이터와 함께 클라이언트 데이터셋 사용

클라이언트 데이터셋은 서버 데이터뿐만 아니라 디스크상의 전용 파일과도 함께 사용할 수 있습니다. 따라서 클라이언트 데이터셋을 파일 기반 데이터베이스 애플리케이션과 "briefcase model" 애플리케이션에서 사용할 수 있습니다. 클라이언트 데이터셋이 데이터에 대해 사용하는 특수 파일을 MyBase라고 합니다.

팁 모든 클라이언트 데이터셋은 briefcase model 애플리케이션에 적합하지만 프로바이더를 사용하지 않는 순수 MyBase 애플리케이션의 경우에는 오버헤드가 적은 *TClientDataSet*을 사용하는 것이 더 좋습니다.

순수 MyBase 애플리케이션에서 클라이언트 애플리케이션은 서버로부터 테이블 정의와 데이터를 가져올 수 없으므로 업데이트를 적용할 서버가 없게 됩니다. 그 대신 클라이언트 데이터셋은 다음 작업을 독립적으로 수행해야 합니다.

- 테이블 정의 및 생성
- 저장된 데이터 로드
- 편집 내용을 데이터에 병합
- 데이터 저장

새로운 데이터셋 생성

다음과 같은 세 가지 방법으로 서버 데이터를 나타내지 않는 클라이언트 데이터셋을 정의하고 만듭니다.

- 영구적 필드나 필드 및 인덱스 정의를 사용하여 새 클라이언트 데이터셋을 정의하고 만들 수 있습니다. 이러한 방법은 테이블 타입 데이터셋을 만드는 스키마와 같습니다. 자세한 내용은 22-37페이지의 "테이블 생성 및 삭제"를 참조하십시오.
- 디자인이나 런타임 시 기존 데이터셋을 복사할 수 있습니다. 기존 데이터셋 복사에 대한 자세한 내용은 27-13페이지의 "다른 데이터셋에서 데이터 복사"를 참조하십시오.
- 임의의 XML 문서에서 클라이언트 데이터셋을 만들 수 있습니다. 자세한 내용은 30-6페이지의 "XML 문서를 데이터 패킷으로 변환"을 참조하십시오.

데이터셋을 만든 다음에는 파일에 이 데이터셋을 저장할 수 있습니다. 이제 테이블을 다시 만들지 않아도 되며 저장한 파일에서 테이블을 읽어오기만 하면 됩니다. 파일 기반의 데이터베이스 애플리케이션을 시작할 때 애플리케이션을 작성하기에 앞서 먼저 데이터베이스를 위한 비어 있는 파일을 만들고 저장해야 하는 경우도 있습니다. 이런 방법으로, 미리 정의되어 있는 클라이언트 데이터셋의 메타데이터로 시작하면 사용자 인터페이스를 쉽게 설정할 수 있습니다.

파일이나 스트림에서 데이터 로드

파일에서 데이터를 로드하려면 클라이언트 데이터셋의 *LoadFromFile* 메소드를 호출합니다. *LoadFromFile*은 데이터를 읽어올 파일을 지정하는 문자열인 하나의 매개변수를 가집니다. 파일 이름은 전체 경로 이름이 될 수도 있습니다. 같은 파일에서 클라이언트 데이터셋의 데이터를 항상 로드하면 *FileName* 속성을 대신 사용할 수 있습니다. *FileName*이 기존 파일을 명명하면 데이터는 클라이언트 데이터셋이 열릴 때 자동으로 로드됩니다.

스트림에서 데이터를 로드하려면 클라이언트 데이터셋의 *LoadFromStream* 메소드를 호출합니다. *LoadFromStream*은 데이터를 제공하는 스트림 객체인 하나의 매개변수를 가집니다.

*LoadFromFile(LoadFromStream)*에 의해 로드되는 데이터는 이 클라이언트 데이터셋이나 다른 클라이언트 데이터셋에서 *SaveToFile(SaveToStream)* 메소드를 사용하여 클라이언트 데이터셋의 데이터 형식으로 이전에 저장하였거나 XML 문서에서 생성된 것입니다. 파일이나 스트림에 데이터를 저장하는 방법에 대한 자세한 내용은 27-34페이지의 "파일이나 스트림에 데이터 저장"을 참조하십시오. XML 문서에서 클라이언트 데이터셋 데이터를 만드는 방법에 대한 자세한 내용은 30장, "데이터베이스 애플리케이션에서 XML 사용"을 참조하십시오.

LoadFromFile 또는 *LoadFromStream*을 호출할 때, 파일의 모든 데이터를 *Data* 속성으로 읽어 들입니다. 즉, 데이터가 저장될 때 변경 로그에 있던 모든 편집 내용을 *Delta* 속성으로 읽어 들입니다. 하지만 파일로 읽어 들인 인덱스만 데이터셋으로 만든 인덱스입니다.

데이터에 변경 내용 병합

클라이언트 데이터셋의 데이터를 편집할 때 데이터의 모든 편집 내용은 인메모리 변경 로그에만 존재합니다. 이 로그는 클라이언트 데이터셋을 사용하는 객체에 완전히 투명하지만 데이터 자체에서 따로 유지될 수 있습니다. 즉, 클라이언트 데이터셋을 탐색하거나 데이터를 표시하는 컨트롤은 변경 내용을 포함하는 데이터 뷰를 봅니다. 하지만 변경 내용을 되돌리지 않으려면 *MergeChangeLog* 메소드를 호출하여 변경 로그를 클라이언트 데이터셋 데이터로 병합해야 합니다. *MergeChangeLog*는 변경 로그의 변경된 필드 값으로 *Data*의 레코드를 오버라이드합니다.

*MergeChangeLog*가 실행된 다음 *Data*에는 기존 데이터와 변경 로그에 있던 변경 내용이 혼합되어 있습니다. 이 혼합은 향후 변경이 수행될 때 새로운 *Data* 기준이 됩니다. *MergeChangeLog*에서는 모든 레코드의 변경 로그를 지운 다음 *ChangeCount* 속성을 *0*으로 다시 설정합니다.

경고 프로바이더를 사용하는 클라이언트 데이터셋의 *MergeChangeLog*는 호출하지 마십시오. 이 경우에는 *ApplyUpdates*를 호출하여 변경 내용을 데이터베이스에 기록해야 합니다. 자세한 내용은 27-20페이지의 "업데이트 적용"을 참조하십시오.

참고 데이터셋이 원래 *Data* 속성에 데이터를 제공한 경우 각각의 클라이언트 데이터셋의 데이터로 변경 내용을 병합할 수도 있습니다. 이렇게 하려면 데이터셋 프로바이더를 사용해야 합니다. 이러한 방법에 대한 예제는 27-14페이지의 "데이터 직접 할당"을 참조하십시오.

변경 로그의 확장된 취소 기능을 사용하지 않으려면 클라이언트 데이터셋의 *LogChanges* 속성을 **false**로 설정할 수 있습니다. *LogChanges*가 **false**인 경우 레코드를 포스트할 때 편집 내용이 자동으로 병합되므로 *MergeChangeLog*를 호출하지 않아도 됩니다.

파일이나 스트림에 데이터 저장

클라이언트 데이터셋의 데이터에 변경 내용을 병합할 때도 이 데이터는 여전히 메모리에만 존재합니다. 클라이언트 데이터셋을 닫고 애플리케이션에서 다시 여는 경우에도 데이터는 여전히 메모리에 존재하지만 애플리케이션을 종료하면 사라집니다. 데이터를 영구적으로 만들려면 데이터를 디스크에 써야 합니다. *SaveToFile* 메소드를 사용하여 변경 내용을 디스크에 기록합니다.

*SaveToFile*은 데이터를 기록할 파일을 지정하는 문자열인 하나의 매개변수를 가집니다. 파일 이름은 전체 경로 이름이 될 수도 있습니다. 해당 파일이 이미 존재하면 파일의 현재 내용이 완전히 덮어써집니다.

참고 *SaveToFile*은 런타임 시 클라이언트 데이터셋에 추가한 인덱스는 유지하지 않고 클라이언트 데이터셋을 만들때 추가한 인덱스만 유지합니다.

데이터를 같은 파일에 항상 저장하는 경우 *FileName* 속성을 대신 사용할 수 있습니다. *FileName*을 설정하면 클라이언트 데이터셋이 닫힐 때 데이터를 자동으로 명명된 파일에 저장합니다.

SaveToStream 메소드를 사용하여 데이터를 스트림에 저장할 수도 있습니다. *SaveToStream*은 데이터를 받는 스트림 객체인 하나의 매개변수를 가집니다.

참고 편집 내용이 여전히 변경 로그에 있는 경우 클라이언트 데이터셋을 저장하면 데이터에 병합되지 않습니다. *LoadFromFile* 또는 *LoadFromStream* 메소드를 사용하여 데이터를 다시 로드하면 변경 로그에는 여전히 병합되지 않은 편집 내용이 있습니다. 변경 내용이 결국엔 애플리케이션 서버의 프로바이더 컴포넌트에 적용되는, briefcase model을 지원하는 애플리케이션에서 이러한 내용은 중요합니다.

장

28

프로바이더 컴포넌트 사용

프로바이더 컴포넌트(*TDataSetProvider* 및 *TXMLTransformProvider*)는 클라이언트 데이터셋이 데이터를 가져오는 가장 일반적인 메커니즘을 제공합니다. 프로바이더는 다음과 같은 작업을 수행합니다.

- 클라이언트 데이터셋 또는 XML 브로커의 데이터 요청을 받고, 요청된 데이터를 가져오고, 전송 가능한 데이터 패킷으로 데이터를 패키지화하고, 클라이언트 데이터셋 또는 XML 브로커에 데이터를 반환합니다. 이러한 작업을 "Providing"이라고 합니다.
- 클라이언트 데이터셋 또는 XML 브로커에서 업데이트된 데이터를 받고, 데이터베이스 서버, 소스 데이터셋 또는 소스 XML 문서에 업데이트를 적용하고, 적용할 수 없는 모든 업데이트를 기록하고, 해결되지 않은 업데이트를 나중에 조정하기 위해 클라이언트 데이터셋에 반환합니다. 이러한 작업을 "Resolving"이라고 합니다.

프로바이더 컴포넌트의 대부분의 작업은 자동으로 이루어집니다. 개발자가 데이터셋이나 XML 문서의 데이터로부터 데이터 패킷을 생성하거나 업데이트를 적용하기 위해 프로바이더에서 코드를 작성할 필요가 없습니다. 또한 프로바이더 컴포넌트에는 다양한 이벤트 및 속성이 포함되어 있어서 애플리케이션이 보다 직접적으로 클라이언트용으로 패키지화되는 정보 및 클라이언트 요청에 반응하는 방법을 제어할 수 있습니다.

TBDEClientDataSet, *TSQLClientDataSet* 또는 *TIBClientDataSet*을 사용하는 경우 프로바이더가 클라이언트 데이터셋 내부에 있으므로 애플리케이션이 프로바이더를 직접 액세스할 수 없습니다. 그러나 *TClientDataSet* 또는 *TXMLBroker*를 사용하는 경우 프로바이더는 클라이언트를 위해, 그리고 Providing 및 Resolving 과정에서 발생하는 이벤트에 응답하기 위해 패키지화되는 정보를 제어하는 데 사용할 수 있는 독립적인 컴포넌트입니다. 내부 프로바이더가 있는 클라이언트 데이터셋은 내부 프로바이더의 일부 속성과 이벤트를 자체 속성과 이벤트로 사용하지만 제어 기능을 향상시키기 위해 각각의 프로바이더 컴포넌트가 있는 *TClientDataSet*을 사용할 수도 있습니다.

각각의 프로바이더 컴포넌트를 사용하는 경우 프로바이더는 클라이언트 데이터셋 또는 XML 브로커와 동일한 애플리케이션에 상주하거나 멀티 티어 애플리케이션의 일부로서 애플리케이션 서버에 상주할 수 있습니다.

이 장에서는 프로바이더 컴포넌트를 사용하여 클라이언트 데이터셋 또는 XML 브로커의 상호 작용을 제어하는 방법에 대해 설명합니다.

데이터의 소스 정하기

프로바이더 컴포넌트를 사용하려면 데이터 패킷으로 만들려는 데이터를 가져올 소스를 지정해야 합니다. 사용자는 C++Builder의 버전에 따라 다음 중 하나를 소스로 지정할 수 있습니다.

- 데이터셋으로부터 데이터를 공급하려면 *TDataSetProvider*를 사용합니다.
- XML 문서로부터 데이터를 공급하려면 *TXMLTransformProvider*를 사용합니다.

데이터 소스로 데이터셋 사용

프로바이더가 데이터셋 프로바이더(*TDataSetProvider*)이면 프로바이더의 *DataSet* 속성이 소스 데이터셋을 나타내도록 설정합니다. 디자인 타임 시 Object Inspector의 *DataSet* 속성 드롭다운 리스트에 있는 사용 가능한 데이터셋 중에서 선택합니다.

*TDataSetProvider*는 *IProviderSupport* 인터페이스를 사용하여 소스 데이터셋과 상호 작용합니다. 이 인터페이스는 *TDataSet*에 포함되어 있으므로 모든 데이터셋에서 사용할 수 있습니다. 그러나, *TDataSet*에서 구현된 *IProviderSupport* 메소드는 대개 아무 작업도 하지 않거나 예외를 발생시키는 스텁입니다.

C++Builder에서 제공하는 데이터셋 클래스(BDE 호환 데이터셋, ADO를 사용할 수 있는 데이터셋, *dbExpress* 데이터셋 및 InterBase Express 데이터셋)는 이러한 메소드를 오버라이드하여 *IProviderSupport* 인터페이스를 보다 유용한 방식으로 구현합니다. 클라이언트 데이터셋은 상속된 *IProviderSupport* 구현에 아무것도 추가하지 않지만 프로바이더의 *ResolveToDataSet* 속성이 **true**이면 여전히 소스 데이터셋으로 사용할 수 있습니다.

*TDataSet*으로부터 고유한 사용자 정의 자손을 만드는 컴포넌트 작성자는 데이터셋이 프로바이더에 데이터를 공급하는 경우 모든 해당 *IProviderSupport* 메소드를 오버라이드해야 합니다. 프로바이더가 단지 읽기 전용으로 데이터 패킷을 공급하는 경우, 즉 업데이트를 적용하지 않는 경우 *TDataSet*에서 구현된 *IProviderSupport* 메소드로 충분할 수 있습니다.

데이터 소스로 XML 문서 사용

프로바이더가 XML 프로바이더이면 프로바이더의 *XMLDataFile* 속성이 소스 문서를 나타내도록 설정합니다.

XML 프로바이더는 소스 문서를 데이터 패킷으로 변환해야 하므로 소스 문서를 지정하는 것 외에 문서를 데이터 패킷으로 변환하는 방법도 지정해야 합니다. 이러한 변환은 프로바이더의 *TransformRead* 속성에 의해 처리됩니다. *TransformRead*는 *TXMLTransform* 객체를 나타냅니다. 해당 속성을 설정하여 사용할 변환 방법을 지정하고 해당 이벤트를 사용하여 변환에 원하는 입력 정보를 제공할 수 있습니다. XML 프로바이더 사용에 대한 자세한 내용은 30-7페이지의 "XML 문서를 프로바이더의 소스로 사용"을 참조하십시오.

클라이언트 데이터셋과의 통신

프로바이더와 클라이언트 데이터셋 또는 XML 브로커 간의 모든 통신은 *IAppServer* 인터페이스를 통해 이루어집니다. 프로바이더가 클라이언트와 동일한 애플리케이션에 있는 경우 이 인터페이스는 자동으로 생성된 숨겨진 객체 또는 *TLocalConnection* 컴포넌트에 의해 구현됩니다. 프로바이더가 멀티 티어 애플리케이션의 일부이면 애플리케이션 서버에 대한 인터페이스가 되며 SOAP 서버의 경우에는 연결 컴포넌트에 의해 생성된 인터페이스가 됩니다.

대부분의 애플리케이션은 *IAppServer*를 직접 사용하지 않고 클라이언트 데이터셋 또는 XML 브로커의 속성 및 메소드를 통해 간접적으로 호출합니다. 그러나 필요에 따라 클라이언트 데이터셋의 *AppServer* 속성을 사용하여 *IAppServer* 인터페이스를 직접 호출할 수 있습니다.

표 28.1에는 *IAppServer* 인터페이스의 메소드와 함께 프로바이더 컴포넌트와 클라이언트 데이터셋의 해당 메소드 및 이벤트가 나열되어 있습니다. 이러한 *IAppServer* 메소드에는 *Provider* 매개변수가 포함됩니다. 멀티 티어 애플리케이션에서 이 매개변수는 클라이언트 데이터셋과 서로 통신하는 애플리케이션 서버 상의 프로바이더를 나타냅니다. 또한 대부분의 메소드에는 *OwnerData*라 불리는 OleVariant 매개변수가 포함됩니다. 이 매개변수를 통해 클라이언트 데이터셋과 프로바이더가 사용자 정의 정보를 앞뒤로 전달할 수 있습니다. *OwnerData*는 디폴트로 사용되지 않지만 모든 이벤트 핸들러에 전달되므로 프로바이더가 클라이언트 데이터셋에서 호출을 받기 전후에 애플리케이션 정의 정보에 따라 조정할 수 있게 하는 코드를 작성할 수 있습니다.

표 28.1 AppServer 인터페이스 멤버

IAppServer	프로바이더 컴포넌트	TClientDataSet
AS_ApplyUpdates 메소드	ApplyUpdates 메소드, BeforeApplyUpdates 이벤트, AfterApplyUpdates 이벤트	ApplyUpdates 메소드, BeforeApplyUpdates 이벤트, AfterApplyUpdates 이벤트
AS_DataRequest 메소드	DataRequest 메소드, OnDataRequest 이벤트	DataRequest 메소드
AS_Execute 메소드	Execute 메소드, BeforeExecute 이벤트, AfterExecute 이벤트	Execute 메소드, BeforeExecute 이벤트, AfterExecute 이벤트
AS_GetParams 메소드	GetParams 메소드, BeforeGetParams 이벤트, AfterGetParams 이벤트	FetchParams 메소드, BeforeGetParams 이벤트, AfterGetParams 이벤트
AS_GetProviderNames 메소드	사용 가능한 모든 프로바이더를 식별하는 데 사용	ProviderName 속성에 대한 디자인 타임 리스트 작성에 사용
AS_GetRecords 메소드	GetRecords 메소드, BeforeGetRecords 이벤트, AfterGetRecords 이벤트	GetNextPacket 메소드 Data 속성 BeforeGetRecords 이벤트, AfterGetRecords 이벤트
AS_RowRequest 메소드	RowRequest 메소드, BeforeRowRequest 이벤트, AfterRowRequest 이벤트	FetchBlobs 메소드, FetchDetails 메소드, RefreshRecord 메소드, BeforeRowRequest 이벤트, AfterRowRequest 이벤트

데이터셋 프로바이더를 사용하여 업데이트를 적용하는 방법 선택

TXMLTransformProvider 컴포넌트는 항상 연결된 XML 문서에 업데이트를 적용합니다. 그러나 *TDataSetProvider*를 사용하는 경우에는 업데이트 적용 방법을 선택할 수 있습니다. 디폴트로, *TDataSetProvider* 컴포넌트는 업데이트를 적용하고 업데이트 오류를 해결할 때 동적으로 생성되는 SQL 문을 사용하여 데이터베이스 서버와 직접 통신합니다. 이러한 접근 방법은 서버 애플리케이션이 업데이트를 두 번(데이터셋에 한 번, 원격 서버에 한 번) 병합하지 않아도 된다는 장점이 있습니다.

그러나 이 접근 방법을 사용하지 않을 수도 있습니다. 예를 들면, 데이터셋 컴포넌트의 이벤트 중 일부만을 사용하고자 하는 경우도 있을 것입니다. 또는 *TClientDataSet* 컴포넌트에서 공급할 경우처럼 사용 중인 데이터셋이 SQL 문 사용을 지원하지 않는 경우도 있을 것입니다.

*TDataSetProvider*는 SQL을 사용하여 데이터베이스 서버에 업데이트를 적용할 것인지, 아니면 *ResolveToDataSet* 속성을 설정하여 소스 데이터셋에 업데이트를 적용할 것인지 결정할 수 있게 해줍니다. 이 속성이 **true**이면 업데이트가 데이터셋에 적용됩니다. 반대로 **false**이면 업데이트가 원본으로 사용하는 데이터베이스 서버에 직접 적용됩니다.

데이터 패킷에 포함될 정보 제어

데이터셋 프로바이더를 사용하여 작업하는 경우 다양한 방법으로 클라이언트와 주고 받는 데이터 패킷에 포함될 정보를 제어할 수 있습니다. 다음과 같은 방법이 있습니다.

- 데이터 패킷에 표시되는 필드 지정
- 데이터 패킷에 영향을 미치는 옵션 설정
- 데이터 패킷에 사용자 정의 정보 추가

참고 데이터 패킷의 내용을 제어하는 이러한 기술은 데이터셋 프로바이더에 대해서만 사용할 수 있습니다. *TXMLTransformProvider*를 사용하는 경우에는 프로바이더가 사용하는 변환 파일을 제어하는 방법을 통해서만 데이터 패킷의 내용을 제어할 수 있습니다.

데이터 패킷에 표시되는 필드 지정

데이터셋 프로바이더를 사용하는 경우에는 프로바이더가 데이터 패킷을 구축하기 위해 사용하는 데이터셋에 영구적 필드(persistent field)를 작성하여 데이터 패킷에 포함되는 필드를 제어할 수 있습니다. 그러면 프로바이더는 이러한 영구적 필드만 포함합니다. 계산된 필드 또는 조회 필드와 같은 소스 데이터셋에 의해 그 값이 동적으로 생성되는 필드도 데이터 패킷에 포함될 수 있지만 맨 마지막으로 받는 클라이언트 데이터셋에 정적 읽기 전용 필드로 표시됩니다. 영구적 필드에 대한 자세한 내용은 23-3페이지의 "영구적 필드(persistent field) 컴포넌트"를 참조하십시오.

클라이언트 데이터셋이 데이터를 편집하고 업데이트를 적용하는 경우에는 데이터 패킷에 중복된 레코드가 존재하지 않도록 충분한 필드를 포함해야 합니다. 그렇지 않으면 업데이트가 적용될 때 업데이트할 레코드를 결정할 수 없습니다. 클라이언트 데이터셋이 고유성을 보장하기 위해 제공되는 여분의 필드를 보거나 사용할 수 없도록 하려면 해당 필드의 *ProviderFlags* 속성을 *pfHidden*으로 설정합니다.

참고 프로바이더의 소스 데이터셋이 쿼리를 나타낼 때도 레코드 중복을 피하기 위해 충분한 필드를 포함해야 합니다. 즉, 애플리케이션이 모든 필드를 사용하지 않더라도 쿼리가 충분한 필드를 포함하도록 지정하여 모든 레코드가 고유하도록 보장해야 합니다.

데이터 패킷에 영향을 주는 옵션 설정

데이터셋 프로바이더의 *Options* 속성을 통해 BLOB 또는 중첩된 디테일 테이블을 보낼 시기, 필드 표시 속성이 포함되는지 여부, 어떤 타입의 업데이트가 허용되는지 등을 지정할 수 있습니다. 다음 표는 *Options*에 포함할 수 있는 값을 나열한 것입니다.

표 28.2 프로바이더 옵션

값	의미
poAutoRefresh	프로바이더는 업데이트를 적용할 때마다 클라이언트 데이터셋을 현재 레코드 값으로 새로 고칩니다.
poReadOnly	클라이언트 데이터셋은 프로바이더에 업데이트를 적용할 수 없습니다.
poDisableEdits	클라이언트 데이터셋은 기존 데이터 값을 수정할 수 없습니다. 사용자가 필드를 편집하려고 하면 클라이언트 데이터셋에 예외가 발생합니다. 그러나 레코드를 삽입하거나 삭제하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.
poDisableInserts	클라이언트 데이터셋은 새 레코드를 삽입할 수 없습니다. 사용자가 새 레코드를 삽입하려고 하면 클라이언트 데이터셋에 예외가 발생합니다. 그러나 레코드를 삭제하거나 기존 데이터를 수정하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.
poDisableDeletes	클라이언트 데이터셋은 레코드를 삭제할 수 없습니다. 사용자가 레코드를 삭제하려고 하면 클라이언트 데이터셋에 예외가 발생합니다. 그러나 레코드를 삽입하거나 수정하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.
poFetchBlobsOnDemand	BLOB 필드 값이 데이터 패킷에 포함되지 않습니다. 그 대신 클라이언트 데이터셋은 이러한 값을 필요에 따라 요청해야 합니다 . 클라이언트 데이터셋의 *FetchOnDemand* 속성이 **true**이면 이러한 값을 자동으로 요청합니다. 그렇지 않으면 애플리케이션이 클라이언트 데이터셋의 *FetchBlobs* 메소드를 호출하여 BLOB 데이터를 검색해야 합니다.
poFetchDetailsOnDemand	프로바이더의 데이터셋이 마스터/디테일 관계의 마스터를 나타내면 중첩 디테일 값이 데이터 패킷에 포함되지 않습니다. 그 대신 클라이언트 데이터셋은 필요에 따라 이러한 값을 요청합니다. 클라이언트 데이터셋의 *FetchOnDemand* 속성이 **true**이면 이러한 값을 자동으로 요청합니다 . 그렇지 않으면 애플리케이션이 클라이언트 데이터셋의 *FetchDetails* 메소드를 호출하여 중첩 디테일을 검색해야 합니다.

표 28.2 프로바이더 옵션

값	의미
poIncFieldProps	데이터 패킷은 해당 사항이 있는 경우 다음과 같은 필드 속성을 포함합니다. *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues*.
poCascadeDeletes	프로바이더의 데이터셋이 마스터/디테일 관계의 마스터를 나타내면 서버는 마스터 레코드가 삭제될 때 자동으로 디테일 레코드를 삭제합니다. 이 옵션을 사용하려면 참조 무결성의 일부로 연결 삭제를 수행하도록 데이터베이스 서버를 설정해야 합니다.
poCascadeUpdates	프로바이더의 데이터셋이 마스터/디테일 관계의 마스터를 나타내면 마스터 레코드에서 해당 값이 바뀔 때 디테일 테이블의 키 값이 자동으로 업데이트됩니다. 이 옵션을 사용하려면 참조 무결성의 일부로 연결 업데이트를 수행하도록 데이터베이스 서버를 설정해야 합니다.
poAllowMultiRecordUpdates	단일 업데이트로 인해 원본으로 사용하는 데이터베이스 테이블에서 둘 이상의 레코드가 변경될 수 있습니다. 이는 트리거, 참조 무결성, 소스 데이터셋의 SQL 문 등의 결과일 수 있습니다. 오류가 발생하는 경우 이벤트 핸들러는 결과적으로 변경되는 다른 레코드가 아니라 업데이트된 레코드에 대한 액세스를 제공한다는 점에 유의하십시오.
poNoReset	클라이언트 데이터셋은 데이터를 공급하기 전에 프로바이더가 첫 번째 레코드에 커서를 다시 배치하도록 지정할 수 없습니다.
poPropogateChanges	서버가 업데이트 프로세스의 일부로 업데이트된 레코드를 변경한 내용은 클라이언트로 다시 보내져 클라이언트 데이터셋에 병합됩니다.
poAllowCommandText	클라이언트는 연결된 데이터셋의 SQL 텍스트나 자신이 나타내는 테이블 또는 내장 프로시저의 이름을 오버라이드할 수 있습니다.
poRetainServerOrder	클라이언트 데이터셋은 디폴트 순서를 적용하기 위해 데이터셋의 레코드를 다시 정렬해서는 안됩니다.

데이터 패킷에 사용자 정의 정보 추가

데이터셋 프로바이더는 *OnGetDataSetProperties* 이벤트를 사용하여 데이터 패킷에 애플리케이션 정의 정보를 추가할 수 있습니다. 이 정보는 OleVariant로 인코드되고 사용자가 지정한 이름으로 저장됩니다. 그러면 클라이언트 데이터셋이 *GetOptionalParam* 메소드를 사용하여 정보를 검색할 수 있습니다. 또한 사용자는 레코드를 업데이트할 때 클라이언트 데이터셋이 보내는 델타 패킷에 정보가 포함되도록 지정할 수 있습니다. 이 경우 클라이언트 데이터셋은 정보를 인식하지 못하지만 프로바이더는 자신에게 왕복 메시지를 보낼 수 있습니다.

OnGetDataSetProperties 이벤트에서 사용자 정의 정보를 추가할 때 세 개의 요소를 포함한 가변 타입 배열을 사용하여 "선택 매개변수"라고도 불리는 각각의 개별 어트리뷰트(attribute)를 지정합니다. 여기서 세 개의 요소는 이름(문자열), 값(가변), 그리고 클라이언트가 업데이트를 적용할 때 델타 패킷에 정보가 포함되는지 여부를 나타내는 부울 플래그를 말합니다. 가변 타입 배열의 가변 타입 배열을 작성하여 여러 어트리뷰트를 추가합니다. 예를 들어, 다음 *OnGetDataSetProperties* 이벤트 핸들러는 두 개의 값, 즉 데이터가 공급된 시간과 소스 데이터 셋의 총 레코드 수를 보냅니다. 클라이언트 데이터셋이 업데이트를 적용하는 경우 데이터가 공급된 시간만 반환됩니다.

```
void __fastcall TMyDataModule1::Provider1GetDataSetProperties(TObject
*Sender, TDataSet *DataSet, out OleVariant Properties)
{
  int ArrayBounds[2];
  ArrayBounds[0] = 0;
  ArrayBounds[1] = 1;
  Properties = VarArrayCreate(ArrayBounds, 1, varVariant);
  Variant values[3];
  values[0] = Variant("TimeProvided");
  values[1] = Variant(Now());
  values[2] = Variant(true);
  Properties[0] = VarArrayOf(values,2);
  values[0] = Variant("TableSize");
  values[1] = Variant(DataSet->RecordCount);
  values[2] = Variant(false);
  Properties[1] = VarArrayOf(values,2);
}
```

클라이언트 데이터셋이 업데이트를 적용하는 경우 프로바이더의 *OnUpdateData* 이벤트에서 원래 레코드가 공급된 시간을 읽을 수 있습니다.

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender,
TCustomClientDataSet *DataSet)
{
  Variant WhenProvided = DataSet->GetOptionalParam("TimeProvided");
  …
}
```

클라이언트 데이터 요청에 대한 응답

일반적으로 데이터에 대한 클라이언트 요청은 자동으로 처리됩니다. 클라이언트 데이터셋 또는 XML 브로커는 *IAppServer* 인터페이스를 통해 간접적으로 *GetRecords*를 호출하여 데이터 패킷을 요청합니다. 프로바이더는 연결된 데이터셋 또는 XML 문서로부터 데이터를 가져오거나 데이터 패킷을 작성하거나 패킷을 클라이언트에 보냄으로써 자동으로 응답합니다.

프로바이더는 데이터 패킷을 만든 후 클라이언트에게 아직 보내지 않은 상태에서 데이터를 편집할 수 있습니다. 예를 들어, 사용자의 액세스 레벨과 같은 몇 가지 기준에 기초하여 패킷에서 레코드를 제거하거나 멀티 티어 애플리케이션의 경우 클라이언트에 보내기 전에 중요한 데이터를 암호화할 수 있습니다.

데이터 패킷을 클라이언트에 보내기 전에 편집하려면 *OnGetData* 이벤트 핸들러를 작성합니다. *OnGetData* 이벤트 핸들러는 클라이언트 데이터셋 형식의 매개변수로 데이터 패킷을 제공합니다. 이 클라이언트 데이터셋의 메소드를 사용하면 클라이언트에 데이터를 보내기 전에 편집할 수 있습니다.

IAppServer 인터페이스를 통해 이루어지는 모든 메소드 호출과 마찬가지로 프로바이더는 *GetRecords*를 호출하기 전후에 클라이언트 데이터셋과 영구적 상태 정보를 통신할 수 있습니다. 이 통신은 *BeforeGetRecords* 및 *AfterGetRecords* 이벤트 핸들러를 사용하여 수행됩니다. 애플리케이션 서버에서의 영구적 상태 정보에 대한 자세한 내용은 29-18페이지의 "원격 데이터 모듈에서 상태 정보 지원"을 참조하십시오.

클라이언트 업데이트 요청에 대한 응답

프로바이더는 클라이언트 데이터셋 또는 XML 브로커에서 받은 *델타* 데이터 패킷에 기초하여 데이터베이스 레코드에 업데이트를 적용합니다. 클라이언트는 *IAppServer* 인터페이스를 통해 간접적으로 *ApplyUpdates* 메소드를 호출하여 업데이트를 요청합니다.

IAppServer 인터페이스를 통해 이루어지는 모든 메소드 호출과 마찬가지로 프로바이더는 *ApplyUpdates*를 호출하기 전후에 클라이언트 데이터셋과 영구적 상태 정보를 통신할 수 있습니다. 이 통신은 *BeforeApplyUpdates* 및 *AfterApplyUpdates* 이벤트 핸들러를 통해 수행됩니다. 애플리케이션 서버에서의 영구적 상태 정보에 대한 자세한 내용은 29-18페이지의 "원격 데이터 모듈에서 상태 정보 지원"을 참조하십시오.

데이터셋 프로바이더를 사용하고 있는 경우에는 수많은 추가 이벤트를 통해 보다 많은 요소를 제어할 수 있습니다.

데이터셋 프로바이더가 업데이트 요청을 받는 경우 *OnUpdateData* 이벤트가 생성되고, 이 이벤트를 통해 델타 패킷을 데이터셋에 기록하기 전에 편집하거나 업데이트가 적용되는 방법에 영향을 줄 수 있습니다. *OnUpdateData* 이벤트 이후 프로바이더는 변경된 내용을 데이터베이스 또는 소스 데이터셋에 기록합니다.

프로바이더는 레코드 단위로 업데이트를 수행합니다. 데이터셋 프로바이더는 각 레코드를 적용하기 전에 *BeforeUpdateRecord* 이벤트를 생성합니다. 이 이벤트를 사용하여 업데이트를 적용하기 전에 미리 검열할 수 있습니다. 레코드를 업데이트할 때 오류가 발생하면 프로바이더는 오류를 해결할 수 있는 *OnUpdateError* 이벤트를 받습니다. 대체로 변경 내용이 서버 제약 조건을 위반하거나 프로바이더가 검색한 이후 클라이언트 데이터셋이 업데이트 적용을 요청하기 전에 데이터베이스 레코드가 다른 애플리케이션에 의해 변경된 경우에 오류가 발생합니다.

업데이트 오류는 데이터셋 프로바이더 또는 클라이언트 데이터셋에 의해 처리될 수 있습니다. 멀티 티어 애플리케이션의 일부인 경우 프로바이더는 사용자 상호 작용 없이 해결해야 하는 모든 업데이트 오류를 처리해야 합니다. 프로바이더가 오류 조건을 해결할 수 없으면 오류를 유발한 레코드의 복사본을 임시로 저장합니다. 레코드 처리가 완료되면 프로바이더는 발생한 오류 개수를 클라이언트 데이터셋에 반환하고 해결되지 않은 레코드를 결과 데이터 패킷에 복사하여 나중에 조정할 수 있도록 클라이언트 데이터셋에 반환합니다.

모든 프로바이더 이벤트에 대한 이벤트 핸들러는 클라이언트 데이터셋으로 업데이트 집합에 전달됩니다. 이벤트 핸들러가 특정 유형의 업데이트만 처리하는 경우 레코드의 업데이트 상태에 기초하여 데이터셋을 필터링할 수 있습니다. 레코드를 필터링하면 이벤트 핸들러가 사용하지 않는 레코드를 정렬할 필요가 없어집니다. 해당 레코드의 업데이트 상태에 관한 클라이언트 데이터셋을 필터링하려면 *StatusFilter* 속성을 설정합니다.

참고 단일 테이블을 나타내지 않는 데이터셋에 업데이트가 지시되는 경우 애플리케이션에서 추가 지원을 제공해야 합니다. 이에 대한 자세한 내용은 28-11페이지의 "단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용"을 참조하십시오.

데이터베이스 업데이트 이전의 델타 패킷 편집

데이터셋 프로바이더는 데이터베이스에 업데이트를 적용하기 전에 *OnUpdateData* 이벤트를 생성합니다. *OnUpdateData* 이벤트 핸들러는 매개변수로 *델타* 패킷의 복사본을 받습니다. 이것은 클라이언트 데이터셋입니다.

OnUpdateData 이벤트 핸들러에서는 클라이언트 데이터셋의 모든 속성 및 메소드를 사용하여 데이터셋에 기록되기 전에 *델타* 패킷을 편집할 수 있습니다. 특히 유용한 속성은 *UpdateStatus* 속성입니다. *UpdateStatus*는 델타 패킷의 현재 레코드가 어떤 타입의 수정을 나타내는지 표시합니다. 또한 표 28.3에 나열된 값 중 하나를 사용할 수 있습니다.

표 28.3 UpdateStatus 값

값	설명
usUnmodified	레코드 내용이 변경되지 않았습니다.
usModified	레코드 내용이 변경되었습니다.
usInserted	레코드가 삽입되었습니다.
usDeleted	레코드가 삭제되었습니다.

예를 들어, 다음과 같은 *OnUpdateData* 이벤트 핸들러는 데이터베이스에 삽입되는 모든 새 레코드에 현재 날짜를 삽입합니다.

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender,
TCustomClientDataSet *DataSet)
{
  DataSet->First();
  while (!DataSet->Eof)
  {
    if (DataSet->UpdateStatus == usInserted)
    {
      DataSet->Edit();
      DataSet->FieldByName("DateCreated")->AsDateTime = Date();
      DataSet->Post();
    }
    DataSet->Next();
  }
}
```

업데이트 적용 방법 변경

데이터셋 프로바이더는 *OnUpdateData* 이벤트를 사용하여 델타 패킷 내의 레코드가 데이터베이스에 적용되는 방법을 표시할 수도 있습니다.

디폴트로, 델타 패킷의 변경 내용은 다음과 같이 자동으로 생성된 SQL UPDATE, INSERT 또는 DELETE 문을 통해 데이터베이스에 기록됩니다.

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

따로 지정하지 않으면 델타 패킷 레코드의 모든 필드가 UPDATE 절과 WHERE 절에 포함됩니다. 그러나 이러한 필드의 일부를 제외할 수 있습니다. 이러한 작업을 수행할 수 있는 한 가지 방법은 프로바이더의 *UpdateMode* 속성을 설정하는 것입니다. *UpdateMode*에는 다음 값이 지정될 수 있습니다.

표 28.4 UpdateMode 값

값	의미
upWhereAll	모든 필드를 사용하여 레코드를 찾습니다(WHERE 절).
upWhereChanged	키 필드와 변경된 필드만 사용하여 레코드를 찾습니다.
upWhereKeyOnly	키 필드만 사용하여 레코드를 찾습니다.

단지 일부 필드를 제외하는 것 말고 더 세부적으로 제어할 수도 있습니다. 예를 들어, 앞의 명령문에서 EMPNO 필드를 UPDATE 절에서 제외하여 수정을 방지하고 TITLE 및 DEPT 필드를 WHERE 절에서 제외하여 다른 애플리케이션이 데이터를 수정하는 경우 업데이트 충돌을 피할 수 있습니다. 특정 필드가 표시되는 절을 지정하려면 *ProviderFlags* 속성을 사용합니다. *ProviderFlags*는 표 28.5에 나열된 값 중 하나를 포함할 수 있는 집합입니다.

표 28.5 ProviderFlags 값

값	설명
pfInWhere	*UpdateMode* 가 *upWhereAll* 또는 *upWhereChanged* 일 경우 생성된 INSERT, DELETE 및 UPDATE 문의 WHERE 절에 필드가 표시됩니다.
pfInUpdate	생성된 UPDATE 문의 UPDATE 절에 필드가 표시됩니다.
pfInKey	*UpdateMode*가 *upWhereKeyOnly*일 경우 생성된 문의 WHERE 절에 필드가 사용됩니다.
pfHidden	고유성을 보장하기 위해 레코드에 필드가 포함되지만 클라이언트측에서는 필드를 보거나 사용할 수 없습니다.

따라서, 다음과 같은 *OnUpdateData* 이벤트 핸들러에서는 TITLE 필드의 업데이트가 허용되며 EMPNO 및 DEPT 필드를 사용하여 원하는 레코드를 찾습니다. 오류가 발생하고 키 필드만 사용하여 레코드를 찾는 두 번째 시도가 이루어질 경우 생성된 SQL은 EMPNO 필드만 검색합니다.

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender,
TCustomClientDataSet *DataSet)
{
  DataSet->FieldByName("EMPNO")->ProviderFlags.Clear();
  DataSet->FieldByName("EMPNO")->ProviderFlags << pfInWHere << pfInKey;
  DataSet->FieldByName("TITLE")->ProviderFlags.Clear();
  DataSet->FieldByName("TITLE")->ProviderFlags << pfInUpdate;
  DataSet->FieldByName("DEPT")->ProviderFlags.Clear();
  DataSet->FieldByName("DEPT")->ProviderFlags << pfInWhere;
}
```

참고 동적으로 생성된 SQL을 사용하지 않고 데이터셋을 업데이트하는 경우에도 *ProviderFlags* 속성을 사용하여 업데이트가 적용되는 방법에 영향을 줄 수 있습니다. 이러한 플래그는 레코드를 찾기 위해 사용되는 필드와 업데이트할 필드를 결정합니다.

개별 업데이트 검열

각 업데이트가 적용되기 바로 전에 데이터셋 프로바이더가 *BeforeUpdateRecord* 이벤트를 받습니다. 이 이벤트를 사용하면 *OnUpdateData* 이벤트를 사용하여 전체 델타 패킷을 편집하는 것과 비슷한 방법으로 레코드를 적용하기 전에 편집할 수 있습니다. 예를 들어, 프로바이더는 업데이트 충돌을 검사할 때 메모와 같은 BLOB 필드를 비교하지 않습니다. BLOB 필드를 포함한 업데이트 오류를 검사하기 위해 *BeforeUpdateRecord* 이벤트를 사용할 수 있습니다.

또한 이 이벤트를 사용하면 업데이트를 적용하거나 검열하거나 거부할 수 있습니다. *BeforeUpdateRecord* 이벤트 핸들러는 업데이트가 이미 처리되었으므로 적용해서는 안된다는 것을 알려줄 수 있게 합니다. 이 경우 프로바이더는 해당 레코드를 건너뛰지만 업데이트 오류로 간주하지는 않습니다. 예를 들어, 이 이벤트는 이벤트 핸들러에서 일단 레코드가 업데이트된 경우 프로바이더가 자동 처리를 건너뛰도록 하면서 자동으로 업데이트할 수 없는 내장 프로시저에 업데이트를 적용할 수 있는 메커니즘을 제공합니다.

프로바이더에서 업데이트 오류 해결

데이터셋 프로바이더가 델타 패킷에 레코드를 포스트하려고 시도할 때 중대한 오류가 발생하면 *OnUpdateError* 이벤트가 발생합니다. 업데이트 오류를 해결할 수 없는 경우 프로바이더는 오류를 유발한 레코드의 복사본을 임시로 저장합니다. 레코드 처리가 완료되면 프로바이더는 발생한 오류 개수를 반환하고 해결되지 않는 레코드를 결과 데이터 패킷에 복사하여 나중에 조정할 수 있도록 클라이언트에 다시 전달합니다.

멀티 티어 애플리케이션에서 이 메커니즘을 통해 오류 조건 수정을 위한 클라이언트 애플리케이션의 사용자 상호 작용을 계속 허용하면서 애플리케이션 서버에서 메커니즘적으로 해결할 수 있는 모든 업데이트 오류를 처리할 수 있습니다.

OnUpdateError 핸들러는 변경할 수 없는 레코드 복사본과 데이터베이스의 오류 코드를 가지며 해결 프로그램이 레코드를 삽입, 삭제 또는 업데이트하려고 했는지 여부를 표시합니다. 문제 레코드는 클라이언트 데이터셋에 다시 전달됩니다. 결코 이 데이터셋에서 데이터 탐색 메소드를 사용해서는 안됩니다. 단, 데이터셋의 각 필드에 대해 *NewValue*, *OldValue* 및 *CurValue* 속성을 사용하여 문제 발생 이유를 판별하고 업데이트 오류를 해결하기 위해 필요한 사항을 수정할 수 있습니다. *OnUpdateError* 이벤트 핸들러로 문제를 수정할 수 있는 경우 *Response* 매개변수를 설정하여 수정된 레코드를 적용합니다.

단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용

데이터셋 프로바이더가 데이터베이스 서버에 직접 업데이트를 적용하는 SQL 문을 생성할 때 레코드를 포함하는 데이터베이스 테이블의 이름이 필요합니다. 테이블 타입 데이터셋 또는 "라이브" *TQuery* 컴포넌트 등 많은 데이터셋에서 이 작업은 자동으로 처리됩니다. 그러나 프로바이더가 결과 집합 또는 다중 테이블 쿼리가 있는 내장 프로시저의 원본으로 사용하는 데이터에 반드시 업데이트를 적용해야 하는 경우에는 자동 업데이트가 문제가 됩니다. 즉, 이 경우 업데이트를 적용해야 할 테이블의 이름을 쉽게 확인할 수 없습니다.

쿼리 또는 내장 프로시저가 BDE 호환 데이터셋(*TQuery* 또는 *TStoredProc*)이며 연결된 업데이트 객체가 있으면 프로바이더가 업데이트 객체를 사용합니다. 그러나 업데이트 객체가 없으면 프로그램에서 *OnGetTableName* 이벤트 핸들러에 테이블 이름을 제공할 수 있습니다. 일단 이벤트 핸들러가 테이블 이름을 제공하면 프로바이더는 적절한 SQL 문을 생성하여 업데이트를 적용할 수 있습니다.

업데이트 대상이 단일 데이터베이스 테이블인 경우에만, 즉 단일 테이블에 있는 레코드만 업데이트해야 하는 경우에 테이블 이름이 제공됩니다. 원본으로 사용하는 여러 데이터베이스 테이블에서 변경이 필요한 업데이트의 경우에는 프로바이더의 *BeforeUpdateRecord* 이벤트를 사용하여 코드에서 업데이트를 명시적으로 적용해야 합니다. 일단 이 이벤트 핸들러가 업데이트를 적용한 후에는 프로바이더가 오류를 생성하지 않도록 이벤트 핸들러의 *Applied* 매개변수를 **true**로 설정할 수 있습니다.

참고 프로바이더가 BDE 호환 데이터셋과 연결되어 있는 경우 *BeforeUpdateRecord* 이벤트 핸들러의 업데이트 객체를 통해 사용자 정의 SQL 문을 사용하여 업데이트를 적용할 수 있습니다. 자세한 내용은 24-39페이지의 "업데이트 객체를 사용하여 데이터셋 업데이트"를 참조하십시오.

클라이언트 생성 이벤트에 응답

프로바이더 컴포넌트는 일반적인 용도의 이벤트를 구현하여 사용자가 클라이언트 데이터셋에서 프로바이더를 직접 호출할 수 있도록 해줍니다. 이러한 이벤트가 바로 *OnDataRequest* 이벤트입니다.

*OnDataRequest*는 프로바이더의 일반적인 기능이 아닙니다. 단지 클라이언트 데이터셋이 프로바이더와 직접 통신할 수 있게 하는 후크(hook)입니다. 이벤트 핸들러는 OleVariant를 입력 매개변수로 가져와 OleVariant를 반환합니다. OleVariant를 사용하므로 인터페이스는 매우 일반적이어서 사용자가 프로바이더와 주고 받고자 하는 거의 모든 정보를 수용할 수 있습니다.

클라이언트 애플리케이션은 *OnDataRequest* 이벤트를 생성하기 위해 클라이언트 데이터셋의 *DataRequest* 메소드를 호출합니다.

서버 제약 조건 처리

대부분의 관계형 데이터베이스 관리 시스템은 테이블에 대한 제약 조건을 구현하여 데이터 무결성을 유지합니다. 제약 조건이란 테이블과 열의 데이터 값에 적용되거나 여러 테이블에 있는 열 간의 데이터 관계에 적용되는 규칙입니다. 예를 들어, SQL-92를 따르는 대부분의 관계형 데이터베이스는 다음 제약 조건을 지원합니다.

- NOT NULL, 열에 반드시 값이 있어야 합니다.
- NOT NULL UNIQUE, 열에 반드시 값이 있고 다른 레코드의 해당 열에 있는 다른 값과 중복되지 않아야 합니다.
- CHECK, 열 값이 특정 범위에 속하거나 특정 값 중 하나여야 합니다.
- CONSTRAINT, 여러 열에 적용되는 테이블 범위의 CHECK 제약 조건입니다.

- PRIMARY KEY, 인덱싱 목적으로 하나 이상의 열을 테이블의 Primary Key로 지정합니다.
- FOREIGN KEY, 다른 테이블을 참조하는 테이블에서 하나 이상의 열을 지정합니다.

참고 이 리스트에 모든 제약 조건이 포함된 것은 아닙니다. 사용하는 데이터베이스 서버에서 이러한 제약 조건의 일부 또는 전부를 지원할 수 있고 추가 제약 조건을 지원할 수도 있습니다. 지원되는 제약 조건에 대한 자세한 내용은 서버 설명서를 참조하십시오.

데이터베이스 서버 제약 조건은 데스크탑 데이터베이스 애플리케이션이 관리하는 수많은 종류의 데이터 검사와 명확히 중복됩니다. 따라서 멀티 티어 데이터베이스 애플리케이션에서 애플리케이션 서버 또는 클라이언트 애플리케이션 코드의 제약 조건을 복제할 필요 없이 서버 제약 조건을 사용할 수 있습니다.

프로바이더가 BDE 호환 데이터셋을 사용하여 작업하는 경우 *Constraints* 속성을 통해 클라이언트 데이터셋과 주고 받는 데이터에 서버 제약 조건을 복제하고 적용할 수 있습니다. *Constraints*가 **true**(기본값)인 경우 소스 데이터셋에 저장된 서버 제약 조건은 데이터 패킷에 포함되며 클라이언트가 데이터를 업데이트하려고 시도할 때 영향을 미칩니다.

중요 프로바이더가 클라이언트 데이터셋에 제약 조건 정보를 전달하려면 반드시 먼저 데이터베이스 서버에서 제약 조건을 검색해야 합니다. 서버에서 데이터베이스 제약 조건을 임포트하려면 SQL 탐색기를 사용하여 데이터베이스 서버의 제약 조건과 디폴트 표현식을 Data Dictionary로 임포트하십시오. BDE 호환 데이터셋에서는 자동으로 Data Dictionary의 제약 조건과 디폴트 표현식을 사용할 수 있습니다.

클라이언트 데이터셋으로 보낸 데이터에 서버 제약 조건을 적용하지 않는 경우도 있습니다. 예를 들어, 패킷 단위로 데이터를 받으면서 추가로 레코드를 가져오기 전에 레코드의 로컬 업데이트를 허용하는 클라이언트 데이터셋은 일시적으로 완전하지 못한 데이터 집합 때문에 일부 서버 제약 조건을 비활성화해야 하는 경우가 있습니다. 프로바이더에서 클라이언트 데이터셋으로 제약 조건이 복제되는 것을 방지하려면 *Constraints*를 **false**로 설정합니다. 클라이언트 데이터셋은 *DisableConstraints* 및 *EnableConstraints* 메소드를 사용하여 제약 조건의 사용 여부를 결정할 수 있습니다. 클라이언트 데이터셋에서 제약 조건을 활성화 및 비활성화하는 방법에 대한 자세한 내용은 27-29페이지의 "서버로부터의 제약 조건 처리"를 참조하십시오.

장

29

멀티 티어 애플리케이션 생성

이 장에서는 멀티 티어 클라이언트/서버 데이터베이스 애플리케이션을 만드는 방법을 설명합니다. 멀티 티어 클라이언트/서버 애플리케이션은 티어라고 하는 논리 유닛으로 분할되는데, 논리 유닛은 각각의 시스템에서 함께 실행됩니다. 멀티 티어 애플리케이션에서는 데이터를 공유하고 LAN이나 인터넷을 통해 서로 통신합니다. 멀티 티어 애플리케이션에서는 중앙에 집중된 비즈니스 로직과 씬 클라이언트 애플리케이션과 같은 많은 장점을 제공합니다.

멀티 티어 애플리케이션의 가장 간단한 형태인 "3티어 모델"은 다음과 같은 세 가지 영역으로 이루어집니다.

- **클라이언트 애플리케이션:** 사용자의 시스템에서 사용자 인터페이스를 제공합니다.
- **애플리케이션 서버:** 모든 클라이언트가 액세스할 수 있는 중앙 네트워킹 위치에 상주하며 일반적인 데이터 서비스를 제공합니다.
- **원격 데이터베이스 서버:** 관계형 데이터베이스 관리 시스템(RDBMS)을 제공합니다.

이 3티어 모델에서 애플리케이션 서버는 클라이언트와 원격 데이터베이스 서버 간의 데이터 흐름을 관리하기 때문에 때로 "데이터 브로커"라고 합니다. 대개는 애플리케이션 서버와 해당 클라이언트만 만듭니다. 그러나 원하는 경우 고유의 데이터베이스 백 엔드도 만들 수 있습니다.

더 복잡한 멀티 티어 애플리케이션에서는 클라이언트와 원격 데이터베이스 서버 사이에 다른 서비스가 상주합니다. 예를 들어, 안전한 인터넷 트랜잭션을 처리하는 보안 서비스 브로커나 다른 플랫폼에 있는 데이터베이스와의 데이터 공유를 처리하는 브리지 서비스가 있을 수 있습니다.

멀티 티어 애플리케이션을 개발하기 위한 VCL 및 CLX 지원은 클라이언트 데이터셋에서 전송 가능한 데이터 패킷을 사용하여 프로바이더 컴포넌트와 통신하는 방법의 확장입니다. 이 장에서는 3티어 데이터베이스 애플리케이션 생성에 대해 중점적으로 설명합니다. 3티어 애플리케이션을 생성하고 관리하는 방법을 이해하게 되면 필요에 따라 새로운 서비스 레이어를 생성하고 추가할 수 있습니다.

멀티 티어 데이터베이스 모델의 장점

멀티 티어 데이터베이스 모델에서는 데이터베이스 애플리케이션을 논리적인 부분들로 분할합니다. 클라이언트 애플리케이션은 데이터 표시와 사용자 상호 작용에 중점을 둘 수 있습니다. 클라이언트 애플리케이션이 데이터 저장 방법이나 유지 보수 방법에 대해 전혀 모르는 것이 이상적입니다. 애플리케이션 서버(미들 티어)는 여러 클라이언트의 요청과 업데이트를 조정하고 처리합니다. 또한, 데이터셋 정의 및 데이터베이스 서버와의 상호 작용에 대한 모든 세부적인 작업을 처리합니다.

멀티 티어 모델에는 다음과 같은 장점이 있습니다.

- **공유 미들 티어에 비즈니스 로직 캡슐화.** 서로 다른 모든 클라이언트 애플리케이션들이 동일한 미들 티어를 액세스합니다. 이렇게 하면 각각의 클라이언트 애플리케이션에 대한 비즈니스 룰을 복제하는 중복과 유지 보수 비용을 피할 수 있습니다.
- **씬 클라이언트 애플리케이션.** 미들 티어에 보다 많은 프로세스를 위임하여 클라이언트 애플리케이션이 처리할 작업을 줄일 수 있습니다. 클라이언트 애플리케이션이 더 작아질 뿐만 아니라, Borland Database Engine과 데이터베이스 서버의 클라이언트사이드 소프트웨어와 같은 데이터베이스 연결 소프트웨어의 설치, 구성 및 유지 보수에 신경쓸 필요가 없으므로 보다 쉽게 배포할 수 있습니다. 인터넷을 통해 씬 클라이언트 애플리케이션을 배포할 수 있다는 사실도 이러한 유연성을 더욱 증가시킵니다.
- **분산된 데이터 처리.** 여러 시스템에서 애플리케이션 작업이 분산되어 처리되면 로드 밸런싱으로 인해 성능이 개선되고 서버가 다운될 경우 다른 시스템에서 작업을 계속 수행할 수 있습니다.
- **향상된 보안성.** 다른 액세스 제한이 있는 티어로 중요한 기능을 분리할 수 있습니다. 그러면 유연하고 구성 가능한 수준의 보안이 제공됩니다. 미들 티어에서는 중요한 자료에 대한 엔트리 포인트를 제한할 수 있기 때문에 액세스를 더 쉽게 제어할 수 있습니다. HTTP나 COM+를 사용할 경우 HTTP나 COM+에서 지원하는 보안 모델의 장점을 활용할 수 있습니다.

프로바이더 기반 멀티 티어 애플리케이션 이해

멀티 티어 애플리케이션에서는 컴포넌트 팔레트의 DataSnap 페이지, Data Access 페이지, WebServices 페이지에 있는 컴포넌트와 New Items 다이얼로그 박스의 Multitier 또는 WebServices 페이지에서 마법사가 만든 원격 데이터 모듈을 사용합니다. 멀티 티어 애플리케이션은 데이터를 전송 가능한 데이터 패킷으로 패키지화하고 전송 가능한 델타 패킷으로 받은 업데이트를 처리하는 프로바이더 컴포넌트의 기능을 기반으로 합니다.

멀티 티어 애플리케이션에 필요한 컴포넌트는 표 29.1에서 설명합니다.

표 29.1 멀티 티어 애플리케이션에서 사용하는 컴포넌트

컴포넌트	설명
원격 데이터 모듈	COM Automation 서버나 웹 서비스 애플리케이션을 사용하여 클라이언트 애플리케이션에게 포함하는 프로바이더에 대한 액세스를 제공하는 특화된 데이터 모듈입니다. 애플리케이션 서버에서 사용합니다.
프로바이더 컴포넌트	데이터 패킷을 만들어서 데이터를 제공하고 클라이언트 업데이트를 해결하는 데이터 브로커입니다. 애플리케이션 서버에서 사용합니다.
클라이언트 데이터셋 컴포넌트	Midas.dll이나 midaslib.dcu를 사용하여 데이터 패킷에 저장된 데이터를 관리하는 특화된 데이터셋입니다. 클라이언트 데이터셋은 클라이언트 애플리케이션에서 사용합니다. 클라이언트 데이터셋은 로컬에서 업데이트를 캐싱하고 델타 패킷의 업데이트를 애플리케이션 서버에 적용합니다.
연결 컴포넌트	서버를 찾고, 연결을 만들고, *IAppServer* 인터페이스를 클라이언트 데이터셋에서 사용할 수 있게 하는 컴포넌트 패밀리입니다. 모든 연결 컴포넌트는 특정한 통신 프로토콜을 사용하도록 특화되었습니다.

프로바이더 컴포넌트와 클라이언트 데이터셋 컴포넌트에서는 데이터 패킷으로 저장된 데이터셋을 관리하는 midas.dll이나 midaslib.dcu가 필요합니다. 프로바이더는 애플리케이션 서버에서 사용하고 클라이언트 데이터셋은 클라이언트 애플리케이션에서 사용하기 때문에 midas.dll을 사용할 경우 애플리케이션 서버와 클라이언트 애플리케이션 모두에 midas.dll을 배포해야 합니다.

BDE 호환 데이터셋을 사용할 경우 애플리케이션 서버에서도 SQL 탐색기가 데이터베이스 관리를 도와주고 서버 제약 조건을 Data Dictionary로 임포트하여 멀티 티어 애플리케이션의 모든 레벨에서 서버 제약 조건을 확인할 수 있게 해야 합니다.

참고 애플리케이션 서버를 배포하기 위한 서버 라이센스를 구매해야 합니다.

이 컴포넌트가 맞는 아키텍처의 개요는 18-12페이지의 "멀티 티어 아키텍처 사용"에서 설명합니다.

3티어 애플리케이션의 개요

번호가 매겨진 다음 단계는 프로바이더 기반 3티어 애플리케이션에 대한 일반적인 이벤트 순서를 나타냅니다.

1. 사용자가 클라이언트 애플리케이션을 시작합니다. 클라이언트는 디자인 타임이나 런타임 시 지정할 수 있는 애플리케이션 서버에 연결됩니다. 애플리케이션 서버를 실행하지 않고 있으면 서버가 시작됩니다. 클라이언트는 애플리케이션 서버와 통신하기 위해 *IAppServer* 인터페이스를 받습니다.
2. 클라이언트가 애플리케이션 서버에서 데이터를 요청합니다. 클라이언트는 한 번에 모든 데이터를 요청하거나, 세션 중에 데이터 청크를 요청할 수 있습니다(요청 시 가져오기).
3. 애플리케이션 서버는 데이터를 검색하고(필요하면 먼저 데이터베이스 연결을 설정), 클라이언트를 위해 데이터를 패키지화하며, 데이터 패킷을 클라이언트에게 반환합니다. 데이터 패킷의 메타데이터에 다른 정보(예: 필드 표시 특징)를 포함시킬 수 있습니다. 데이터를 데이터 패킷으로 패키지화하는 이 프로세스를 "Providing"이라고 합니다.

4. 클라이언트에서 데이터 패킷을 디코딩하고 데이터를 사용자에게 표시합니다.
5. 사용자가 클라이언트 애플리케이션과 상호 작용하면서 데이터가 업데이트됩니다. 즉 레코드가 추가, 삭제 또는 수정됩니다. 이 수정 사항은 클라이언트의 변경 로그에 저장됩니다.
6. 결국 클라이언트는 대개 사용자 액션에 대한 응답으로 애플리케이션 서버에 업데이트를 적용합니다. 업데이트를 적용하기 위해 클라이언트는 변경 로그를 패키지화하여 이를 서버에 데이터 패킷으로 보냅니다.
7. 애플리케이션 서버에서는 패키지를 디코딩하고 해당하는 트랜잭션의 컨텍스트에서 업데이트를 포스트합니다. 예를 들어, 클라이언트에서 레코드를 요청한 후 그리고 클라이언트에서 업데이트를 적용하기 전에 다른 애플리케이션에서 레코드를 변경했기 때문에 레코드를 포스트할 수 없을 경우 애플리케이션 서버는 클라이언트의 변경 사항과 현재 데이터를 같이 조정하거나 포스트할 수 없는 레코드를 저장합니다. 레코드를 포스트하고 문제 레코드를 캐싱하는 이 과정을 "Resolving"이라고 합니다.
8. 애플리케이션 서버에서 Resolving 프로세스를 완료하면 포스트되지 않은 레코드를 앞으로 해결하기 위해 클라이언트에 반환합니다.
9. 클라이언트에서는 해결되지 않은 레코드를 조정합니다. 클라이언트에서는 해결되지 않은 레코드를 여러 가지 방법으로 조정할 수 있습니다. 일반적으로 클라이언트는 레코드의 포스트를 방해하는 상황을 수정하거나 변경 사항을 버립니다. 오류 상황을 수정할 수 있으면 클라이언트에서는 업데이트를 다시 적용합니다.
10. 클라이언트는 서버의 해당 데이터를 새로 고칩니다.

클라이언트 애플리케이션의 구조

엔드 유저에게 있어서 멀티 티어 애플리케이션의 클라이언트 애플리케이션은 캐싱된 업데이트를 사용하는 2티어 애플리케이션과 모양이나 동작면에서 유사합니다. *TClientDataSet* 컴포넌트의 데이터를 표시하는 표준 데이터 인식 컨트롤을 통해 사용자 상호 작용이 발생합니다. 속성, 이벤트 및 클라이언트 데이터셋의 메소드 사용에 대한 자세한 내용은 27장, "클라이언트 데이터셋 사용"을 참조하십시오.

외부 프로바이더와 함께 클라이언트 데이터셋을 사용하는 2티어 애플리케이션에서처럼 *TClientDataSet* 은 프로바이더 컴포넌트에서 데이터를 가져오고 업데이트를 프로바이더 컴포넌트에 적용합니다. 프로바이더에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오. 프로바이더와 통신을 용이하게 하는 클라이언트 데이터셋 기능에 대한 자세한 내용은 27-24페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

클라이언트 데이터셋은 *IAppServer* 인터페이스를 통해 프로바이더와 통신합니다. 클라이언트 데이터셋은 연결 컴포넌트에서 이 인터페이스를 가져옵니다. 연결 컴포넌트는 애플리케이션 서버에 연결합니다. 다른 통신 프로토콜을 사용하기 위해 다른 연결 컴포넌트를 사용할 수 있습니다. 다음 표는 연결 컴포넌트를 요약한 것입니다.

표 29.2 연결 컴포넌트

컴포넌트	프로토콜
TDCOMConnection	DCOM
TSocketConnection	Windows 소켓(TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP(HTTP 및 XML)

참고 컴포넌트 팔레트의 DataSnap 페이지에는 애플리케이션 서버에 전혀 연결되지 않은 연결 컴포넌트도 포함하지만 대신 클라이언트 데이터셋이 같은 애플리케이션의 프로바이더와의 통신 중에 사용하는 *IAppServer* 인터페이스를 제공합니다. *TLocalConnection* 컴포넌트는 필요하지 않지만 나중에 멀티 티어 애플리케이션으로 더 쉽게 확장할 수 있습니다.

연결 컴포넌트 사용에 대한 자세한 내용은 29-22페이지의 "애플리케이션 서버에 연결"을 참조하십시오.

애플리케이션 서버의 구조

애플리케이션 서버를 설정하고 실행할 경우 애플리케이션 서버가 클라이언트 애플리케이션과 연결되지 않습니다. 대신 클라이언트 애플리케이션에서 연결을 초기화하고 유지 보수합니다. 클라이언트 애플리케이션에서는 연결 컴포넌트를 사용하여 애플리케이션 서버에 연결하고, 애플리케이션 서버의 인터페이스를 사용하여 선택한 프로바이더와 통신합니다. 개발자가 들어온 요청을 관리하거나 인터페이스를 제공하기 위한 코드를 작성하지 않아도 이런 작업은 모두 자동으로 발생합니다.

애플리케이션 서버의 기본은 원격 데이터 모듈입니다. 원격 데이터 모듈은 *IAppServer* 인터페이스를 지원하는 특화된 데이터 모듈입니다. 웹 서비스 역할도 하는 애플리케이션 서버용의 경우 원격 데이터 모듈에서는 *IAppServerSOAP* 인터페이스도 지원하며 *IAppServer*보다는 이 인터페이스를 사용합니다. 클라이언트 애플리케이션에서는 원격 데이터 모듈의 인터페이스를 사용하여 애플리케이션 서버의 프로바이더와 통신합니다. 원격 데이터 모듈에서 *IAppServerSOAP*를 사용할 경우 연결 컴포넌트에서는 클라이언트 데이터셋에서 사용할 수 있는 *IAppServer* 인터페이스에 이를 사용합니다.

다음과 같은 두 종류의 원격 데이터 모듈이 있습니다.

- COM 기반 원격 데이터 모듈은 REMOTEDATAMODULE_IMPL()의 자손인 구현 객체라고 하는 연결된 객체를 사용하는 데이터 모듈입니다. REMOTEDATAMODULE_IMPL은 구현 객체에 대한 조상을 나열하는 Atlvcl.h에 정의된 매크로입니다. 여기에는 *IAppServer* 인터페이스뿐만 아니라 ATL 클래스 *CComObjectRootEx* 및 *CComCoClass*도 포함됩니다. MTS나 COM+ 에서 제공하는 분산된 애플리케이션 서비스를 이용할 수 있는 애플리케이션 서버를 만들 경우 REMOTEDATAMODULE_IMPL에는 모든 트랜잭션 객체에 필요한 *IObjectControl* 인터페이스도 포함됩니다. 마법사에서 대신 만들어준 구현 클래스에는 *IObjectContext* 인터페이스에 대한 액세스가 있습니다. 이 인터페이스는 시스템에서 대신 제공하며 트랜잭션을 관리하고, 리소스를 확보하며, 보안 지원을 활용하기 위해 시스템에서 사용합니다.

- SOAP 데이터 모듈은 *IAppServerSOAP* 인터페이스를 인보커블 인터페이스로 구현하는 데이터 모듈입니다. 이 데이터 모듈은 웹 서비스 애플리케이션에 추가되고, 이 데이터 모듈을 사용하면 클라이언트에서 웹 서비스처럼 데이터를 액세스할 수 있습니다. 웹 서비스 애플리케이션에 대한 자세한 내용은 36장, "웹 서비스 사용"을 참조하십시오.

참고 애플리케이션 서버를 MTS나 COM+에서 배포할 경우 원격 데이터 모듈에는 애플리케이션 서버가 활성화하거나 비활성화되었을 경우의 이벤트가 포함됩니다. 그러면 활성화되었을 때는 데이터베이스 연결을 취득해 오고 비활성화되었을 때는 연결을 릴리스할 수 있습니다.

원격 데이터 모듈의 내용

다른 데이터 모듈을 사용할 때처럼 넌비주얼(nonvisual) 컴포넌트를 원격 데이터 모듈에 포함시킬 수 있습니다. 그러나 다음과 같이 반드시 포함시켜야 하는 특정 컴포넌트도 있습니다.

- 원격 데이터 모듈에서 데이터베이스 서버의 정보를 제공할 경우 해당 데이터베이스 서버의 레코드를 나타내기 위해 데이터셋 컴포넌트도 포함해야 합니다. 일부 타입의 데이터베이스 연결 컴포넌트와 같은 다른 컴포넌트에서는 데이터셋이 데이터베이스 서버와 상호 작용하도록 허용해야 합니다. 데이터셋에 대한 자세한 내용은 22장, "데이터셋 이해"를 참조하십시오. 데이터베이스 연결 컴포넌트에 대한 자세한 내용은 21장, "데이터베이스에 연결"을 참조하십시오.

  원격 데이터 모듈이 클라이언트에 노출하는 모든 데이터셋은 데이터셋 프로바이더를 가져야 합니다. 데이터셋 프로바이더는 클라이언트 데이터셋으로 보내고 클라이언트 데이터셋에서 받은 업데이트를 다시 소스 데이터셋이나 데이터베이스 서버에 적용하는 데이터 패킷으로 데이터를 패키지화합니다. 데이터셋 프로바이더에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오.

- 원격 데이터 모듈이 클라이언트에 노출하는 모든 XML 문서는 XML 프로바이더를 가져야 합니다. XML 프로바이더는 데이터베이스 서버보다는 XML 문서에 업데이트를 적용하고 XML 문서에서 데이터를 가져온다는 점만 제외하고 데이터셋 프로바이더와 같이 작동합니다. XML 프로바이더에 대한 자세한 내용은 30-7 페이지의 "XML 문서를 프로바이더의 소스로 사용"을 참조하십시오.

참고 데이터셋을 데이터베이스 서버에 연결하는 데이터베이스 연결 컴포넌트를 멀티 티어 애플리케이션의 클라이언트 애플리케이션에서 사용하는 연결 컴포넌트와 혼동하지 마십시오. 멀티 티어 애플리케이션의 연결 컴포넌트는 컴포넌트 팔레트의 DataSnap 페이지나 WebServices 페이지에 있습니다.

트랜잭션 데이터 모듈 사용

MTS(Windows 2000 이전) 또는 COM+(Windows 2000 이후)에서 제공하는 분산 애플리케이션의 특수 서비스를 이용하는 애플리케이션 서버를 작성할 수 있습니다. 그렇게 하려면 일반적인 원격 데이터 모듈 대신 트랜잭션 데이터 모듈을 만드십시오.

트랜잭션 데이터 모듈을 사용할 경우 애플리케이션에서 다음 특수 서비스를 이용할 수 있습니다.

- **보안.** MTS나 COM+에서는 애플리케이션 서버에 대한 역할 기반 보안을 제공합니다. 클라이언트에는 역할이 할당되는데, 역할은 클라이언트에서 애플리케이션 서버의 인터페이스를 액세스할 수 있는 방법을 결정합니다. 구현 클래스를 통해 액세스하는 *IObjectContext* 인터페이스를 사용하여 이러한 보안 서비스를 액세스할 수 있습니다. MTS 및 COM+ 보안에 대한 자세한 내용은 44-17페이지의 "역할 기반 보안"을 참조하십시오.
- **데이터베이스 핸들 풀링(pooling).** 트랜잭션 데이터 모듈에서는 ADO(MTS를 사용하고 MTS POOLING을 선택한 경우)나 BDE를 통해 만든 데이터베이스 연결을 자동으로 풀링(pooling)합니다. 한 클라이언트에서 데이터베이스 연결을 종료한 경우 다른 클라이언트에서 이를 다시 사용할 수 있습니다. 그러면 네트워크 트래픽이 줄어듭니다. 미들 티어에서 원격 데이터베이스 서버를 로그오프한 다음 다시 로그온하지 않아도 되기 때문입니다. 데이터베이스 핸들을 풀링(pooling)할 경우 데이터베이스 연결 컴포넌트에서는 애플리케이션에서 연결 공유를 최대화하도록 *KeepConnection* 속성을 **false**로 설정해야 합니다. 데이터베이스 핸들 풀링(pooling)에 대한 자세한 내용은 44-6 페이지의 "데이터베이스 리소스 디스펜서"를 참조하십시오.
- **트랜잭션.** 트랜잭션 데이터 모듈을 사용할 경우 단일 데이터베이스 연결로 사용할 수 있는 트랜잭션 지원 이상의 향상된 트랜잭션 지원을 제공할 수 있습니다. 트랜잭션 데이터 모듈은 여러 데이터베이스에 걸친 트랜잭션에 참여하거나 데이터베이스에 전혀 관련되지 않은 함수를 포함할 수 있습니다. 트랜잭션 데이터 모듈과 같은 트랜잭션 객체에서 제공하는 트랜잭션 지원에 대한 자세한 내용은 29-17페이지의 "멀티 티어 애플리케이션의 트랜잭션 관리"를 참조하십시오.
- **just-in-time 활성화 및 as-soon-as-possible 비활성화.** 필요에 따라 인스턴스를 활성화거나 비활성화하도록 서버를 작성할 수 있습니다. just-in-time 활성화와 as-soon-as-possible 비활성화를 사용할 경우 클라이언트 요청을 처리하기 위해 필요한 경우에만 애플리케이션 서버가 인스턴스화됩니다. 그러면 데이터베이스 핸들과 같은 리소스를 사용하지 않을 경우 리소스에 연결되지 않아도 됩니다.

  just-in-time 활성화와 as-soon-as-possible 비활성화를 사용하면 단일 원격 데이터 모듈 인스턴스를 통해 모든 클라이언트를 라우트하는 것과 클라이언트 연결에 대한 각각의 인스턴스를 만드는 것 사이의 중간 지점이 제공됩니다. 단일 원격 데이터 모듈 인스턴스를 사용할 경우 애플리케이션 서버는 단일 데이터베이스 연결을 통해 모든 데이터베이스 호출을 처리해야 합니다. 그러면 병목 역할을 하며 클라이언트가 많을 경우 성능에 영향을 미칠 수 있습니다. 원격 데이터 모듈의 복수 인스턴스를 사용할 경우 모든 인스턴스에서 각각의 데이터베이스 연결을 유지할 수 있습니다. 그러면 데이터베이스 액세스를 serialize하지 않아도 됩니다. 그러나 이러면 다른 클라이언트의 원격 데이터 모듈과 데이터베이스가 연결되어 있는 동안 또다른 클라이언트에서 데이터베이스 연결을 사용할 수 없기 때문에 리소스를 독점하게 됩니다.

트랜잭션, just-in-time 활성화 및 as-soon-as-possible 비활성화를 이용하려면 원격 데이터 모듈 인스턴스가 stateless여야 합니다. 이것은 클라이언트가 상태 정보에 의존할 경우 다른 지원을 제공해야 함을 의미합니다. 예를 들어, 증분 페치(fetch)를 수행할 경우 클라이언트에서 현재 레코드에 대한 정보를 전달해야 합니다. 상태 정보 및 멀티 티어 애플리케이션의 원격 데이터 모듈에 대한 자세한 내용은 29-18페이지의 "원격 데이터 모듈에서 상태 정보 지원"을 참조하십시오.

디폴트로, 트랜잭션 데이터 모듈에 대해 자동으로 만들어진 모든 호출은 트랜잭션입니다. 즉, 호출이 있으면 데이터 모듈을 비활성화하고 현재 트랜잭션을 커밋하거나 롤백할 수 있음을 가정합니다. *AutoComplete* 속성을 **false**로 설정하여 영구적 상태 정보에 의존하는 트랜잭션 데이터 모듈을 작성할 수 있지만, 사용자 정의 인터페이스를 사용하지 않으면 트랜잭션, just-in-time 활성화 또는 as-soon-as-possible 비활성화를 지원하지 않습니다.

경고 트랜잭션 데이터 모듈을 포함하는 애플리케이션 서버에서는 데이터 모듈을 활성화할 때까지 데이터베이스 연결을 열지 말아야 합니다. 애플리케이션을 개발하는 중에 모든 데이터셋을 비활성화해야 하고 애플리케이션을 실행하기 전에 데이터베이스를 연결하지 말아야 합니다. 데이터 모듈이 활성화되어 있으면 애플리케이션에서 데이터베이스 연결을 여는 코드를 추가하고 데이터 모듈이 비활성화되어 있으면 데이터베이스 연결을 닫습니다.

원격 데이터 모듈 풀링(pooling)

객체 풀링을 사용하면 클라이언트에서 공유하는 애플리케이션 서버의 캐시를 만들어 리소스를 절약할 수 있습니다. 이러한 방법은 원격 데이터 모듈 타입과 연결 프로토콜에 따라 다릅니다.

COM+에 설치될 트랜잭션 데이터 모듈을 만들 경우 COM+ Component Manager를 사용하여 애플리케이션 서버를 풀링된 객체로 설치할 수 있습니다. 자세한 내용은 44-9페이지의 "객체 풀링"을 참조하십시오.

트랜잭션 데이터 모듈을 사용하지 않더라도 *TWebConnection*을 사용하여 연결을 만든 경우 객체 풀링의 장점을 활용할 수 있습니다. 두 번째 타입의 객체 풀링에서는 만든 애플리케이션 서버의 인스턴스 수를 제한하십시오. 그러면 애플리케이션 서버에서 사용한 다른 리소스 뿐만 아니라 유지해야 할 데이터베이스 연결 수가 제한됩니다.

사용자의 애플리케이션 서버에 호출을 전달하는 웹 서버 애플리케이션에서 클라이언트 요청을 받은 경우 풀에서 사용 가능한 첫 번째 애플리케이션 서버에 이 요청을 전달합니다. 사용 가능한 애플리케이션 서버가 없으면 새로운 애플리케이션 서버를 만드는데, 지정한 최대 개수까지 가능합니다. 그러면 병목 역할을 할 수 있는 단일 애플리케이션 서버 인스턴스를 통해 모든 클라이언트를 라우트는 것과 많은 리소스를 사용할 수 있는 각 클라이언트 연결에 대한 각각의 인스턴스를 만드는 것 사이의 중간 지점이 제공됩니다.

풀의 애플리케이션 서버 인스턴스에서 오랫동안 클라이언트 요청을 받지 않으면 자동으로 해제됩니다. 그러면 리소스를 사용하지 않을 경우 풀에서 리소스를 독점하지 못하게 합니다.

웹 연결(HTTP)을 사용할 때 객체 풀링을 설정하려면 다음을 수행하십시오.

1 구현 클래스의 *UpdateRegistry* 메소드를 찾습니다. 이 메소드는 구현 유닛의 헤더 파일에 나타납니다.

```
static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
  TRemoteDataModuleRegistrar regObj(GetObjectCLSID(), GetProgID(),
GetDescription());
  return regObj.UpdateRegistry(bRegister);
}
```

2 *TRemoteDataModuleRegistrar*의 인스턴스인 *regObj* 변수의 *RegisterPooled* 플래그를 **true**로 설정합니다. 원격 데이터 모듈의 캐싱 관리 방법을 나타내기 위해 *regObj*의 다른 속성을 설정하고자 할 수도 있습니다. 예를 들어, 다음 코드에서는 최대 10개의 원격 데이터 모듈 인스턴스를 허용하며 15분 이상 유휴 상태일 경우 이를 캐시에서 해제합니다.

```
static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
  TRemoteDataModuleRegistrar regObj(GetObjectCLSID(), GetProgID(),
GetDescription());
  regObj.RegisterPooled = true;
  regObj.Timeout = 15;
  regObj.Max = 10;
  return regObj.UpdateRegistry(bRegister);
}
```

객체 풀링의 메소드를 사용할 경우 애플리케이션 서버는 stateless여야 합니다. 단일 인스턴스에서 잠재적으로 여러 클라이언트의 요청을 처리하기 때문에 그렇습니다. 영구적 상태 정보에 의존할 경우 클라이언트는 서로 간섭할 수 있습니다. 원격 데이터 모듈이 stateless인지 확인하는 방법에 대한 자세한 내용은 29-18페이지의 "원격 데이터 모듈에서 상태 정보 지원"을 참조하십시오.

연결 프로토콜 선택

클라이언트 애플리케이션을 애플리케이션 서버에 연결하기 위해 사용할 수 있는 모든 통신 프로토콜에는 각각 고유한 장점이 있습니다. 프로토콜을 선택하기 전에 예상하는 클라이언트 수, 애플리케이션을 배포하는 방법, 앞으로의 개발 계획 등을 고려하십시오.

DCOM 연결 사용

DCOM에서는 통신에 대한 가장 직접적인 접근 방법을 제공하며 서버에 다른 런타임 애플리케이션이 필요하지 않습니다. 그러나 DCOM은 Windows 95에 포함되지 않기 때문에 이전의 일부 클라이언트 시스템에 DCOM이 설치되어 있지 않을 수 있습니다.

DCOM에서는 트랜잭션 데이터 모듈을 작성할 때 보안 서비스를 사용할 수 있게 해주는 유일한 접근 방법을 제공합니다. 이 보안 서비스는 트랜잭션 객체의 호출자에게 역할을 할당하는 것을 바탕으로 합니다. DCOM을 사용할 경우, DCOM은 개발자의 애플리케이션 서버(MTS 또는 COM+)를 호출하는 시스템 호출자를 식별합니다. 그러므로 호출자의 역할을 정확하게 판별할 수 있습니다. 그러나 다른 프로토콜을 사용할 경우 애플리케이션 서버와는 별개이며 클라이언트 호출을 받는 런타임 실행 파일이 있습니다. 이 런타임 실행 파일이 클라이언트를 대신하여 애플리케이션 서버로 COM 호출을 합니다. 이 때문에 각각의 클라이언트에 역할을 할당할 수 없습니다. 실제로 런타임 실행 파일이 유일한 클라이언트입니다. 보안 및 트랜잭션 객체에 대한 자세한 내용은 44-17페이지의 "역할 기반 보안"을 참조하십시오.

소켓 연결 사용

TCP/IP 소켓을 사용하면 lightweight 클라이언트를 만들 수 있습니다. 예를 들어, 웹 기반 클라이언트 애플리케이션을 작성할 경우 클라이언트 시스템에서 DCOM을 지원하는지 확인할 수 없습니다. 소켓에서는 애플리케이션 서버에 연결하기 위해 사용할 수 있는 가장 낮은 공통 분모를 제공합니다. 소켓에 대한 자세한 내용은 37장, "소켓 작업"을 참조하십시오.

DCOM에서처럼 클라이언트에서 직접 원격 데이터 모듈을 인스턴스화하는 대신 소켓에서는 COM을 사용하여 클라이언트 요청을 승인하고 애플리케이션 서버를 인스턴스화하는 독립적인 서버 애플리케이션(ScktSrvr.exe)을 사용합니다. 클라이언트의 연결 컴포넌트와 서버의 ScktSrvr.exe에서 *IAppServer* 호출을 마샬링합니다.

참고 ScktSrvr.exe를 NT 서비스 애플리케이션처럼 실행할 수 있습니다. -install 명령줄 옵션을 사용하여 ScktSrve.exe를 시작하여 서비스 관리자에 등록하십시오. -uninstall 명령줄 옵션을 사용하여 등록을 해제할 수 있습니다.

소켓 연결을 사용하려면 애플리케이션 서버에서 소켓 연결을 사용하여 클라이언트에 사용 가능성을 등록해야 합니다. 디폴트로, 모든 새로운 원격 데이터 모듈은 구현 객체의 *UpdateRegistry* 메소드에 있는 *TRemoteDataModuleRegistrar* 객체를 통해 자동으로 자신을 등록합니다. 객체의 *EnableSocket* 속성을 **false**로 설정하여 이러한 등록을 못하게 할 수 있습니다.

참고 이전 서버에서는 등록을 추가하지 않았기 때문에 ScktSrvr.exe에서 Connections|Registered Objects Only 메뉴 항목의 선택을 해제하여 애플리케이션 서버가 등록되었는지 여부를 확인하지 않을 수 있습니다.

소켓을 사용하는 경우 클라이언트 시스템이 애플리케이션 서버의 인터페이스에 대한 참조를 해제할 때까지 서버에 클라이언트 시스템 오류에 대한 보호 수단이 전혀 없습니다. 이 때문에 정기적으로 keep-alive 메시지를 보내는 DCOM 을 사용할 때보다 메시지 트래픽이 감소되긴 하지만, 애플리케이션 서버가 클라이언트 오류를 인식하지 못하므로 리소스를 해제하지 못할 수 있습니다.

웹 연결 사용

HTTP를 사용하면 방화벽으로 보호된 애플리케이션 서버와 통신할 수 있는 클라이언트를 만들 수 있습니다. HTTP 메시지는 내부 애플리케이션에 제어된 액세스를 제공하므로 클라이언트 애플리케이션을 안전하고 폭넓게 분산시킬 수 있습니다. 소켓 연결처럼 HTTP 메시지는 애플리케이션 서버에 연결하기 위해 사용할 수 있는 가장 낮은 공통 분모를 제공합니다. HTTP 메시지에 대한 자세한 내용은 32장, "인터넷 서버 애플리케이션 생성"을 참조하십시오.

DCOM 에서처럼 클라이언트에서 직접 원격 데이터 모듈을 인스턴스화하는 대신 HTTP 기반 연결에서는 COM을 사용하여 클라이언트 요청을 승인하고 애플리케이션 서버를 인스턴스화하는 서버의 웹 서버 애플리케이션(httpsrvr.dll)을 사용합니다. 이 때문에 웹 연결이라고도 합니다. 클라이언트의 연결 컴포넌트와 서버의 httpsrvr.dll에서 *IAppServer* 호출을 마샬링합니다.

웹 연결에서는 wininet.dll에서 제공하는 SSL 보안을 이용할 수 있습니다. Wininet.dll은 클라이언트 시스템에서 실행되는 인터넷 유틸리티의 라이브러리입니다. 인증을 요구하기 위해 서버 시스템에 웹 서버를 구성한 경우 웹 연결 컴포넌트의 속성을 사용하여 사용자 이름과 암호를 지정할 수 있습니다.

추가 보안 조치로서 애플리케이션 서버는 웹 연결을 사용하여 클라이언트에 대한 사용 가능성을 등록해야 합니다. 디폴트로, 모든 새로운 원격 데이터 모듈은 구현 객체의 *UpdateRegistry* 메소드에 있는 *TRemoteDataModuleRegistrar* 객체를 통해 자동으로 자신을 등록합니다. 해당 객체의 *EnableWeb* 속성을 **false**로 설정하여 등록을 못하게 할 수 있습니다.

웹 연결에서는 객체 풀링의 장점을 활용할 수 있습니다. 그러면 서버가 클라이언트 요청에 사용할 수 있는 제한된 풀의 애플리케이션 서버 인스턴스를 만들 수 있습니다. 애플리케이션 서버를 풀링하면 서버에서 필요한 경우는 제외하고 데이터 모듈과 해당 데이터베이스 연결에 대한 리소스를 사용하지 않습니다. 객체 풀링에 대한 자세한 내용은 29-8페이지의 "원격 데이터 모듈 풀링(pooling)"을 참조하십시오.

대부분의 다른 연결 컴포넌트와는 달리 HTTP를 통해 연결을 만든 경우 콜백을 사용할 수 없습니다.

SOAP 연결 사용

SOAP는 웹 서비스 애플리케이션에 대한 VCL 또는 CLX 지원의 기본이 되는 프로토콜입니다. SOAP에서는 XML 인코딩을 사용하여 메소드 호출을 마샬링합니다. SOAP 연결에서는 HTTP를 전송 프로토콜로 사용합니다.

SOAP 연결은 Windows와 Linus 모두에서 지원되므로 크로스 플랫폼 애플리케이션에서 작동하는 장점이 있습니다. SOAP 연결에서는 HTTP를 사용하므로 웹 연결과 같은 장점이 있습니다. HTTP에서는 모든 클라이언트에서 사용할 수 있고 클라이언트가 "방화벽"으로 보호된 애플리케이션 서버와 통신할 수 있는 가장 낮은 공통 분모를 제공합니다. SOAP를 사용하여 애플리케이션을 분산시키는 것에 대한 자세한 내용은 36장, "웹 서비스 사용"을 참조하십시오.

HTTP 연결을 사용할 때처럼 SOAP를 통해 연결을 만든 경우 콜백을 사용할 수 없습니다. SOAP 연결에서는 애플리케이션 서버의 단일 원격 데이터 모듈로 제한하기도 합니다.

멀티 티어 애플리케이션 생성

멀티 티어 데이터베이스 애플리케이션을 만들기 위한 일반적인 단계는 다음과 같습니다.

1. 애플리케이션 서버를 만듭니다.
2. 애플리케이션 서버를 등록하거나 설치합니다.
3. 클라이언트 애플리케이션을 만듭니다.

만드는 순서가 중요합니다. 클라이언트를 만들기 전에 애플리케이션을 만들어 실행해야 합니다. 그런 다음 디자인 타임 시 애플리케이션 서버에 연결하여 클라이언트를 테스트할 수 있습니다. 물론 디자인 타임 시 애플리케이션 서버를 지정하지 않고 클라이언트를 만들 수 있으며 런타임 시 서버 이름만 제공합니다. 그러나 그렇게 하면 디자인 타임 시 코딩한 경우 애플리케이션이 예상한 대로 작동하는지 확인할 수 없습니다. 그리고 Object Inspector를 사용하여 서버와 프로바이더를 선택할 수 없습니다.

참고 서버와 같은 시스템에 클라이언트 애플리케이션을 만들지 않고 COM 연결을 사용할 경우 클라이언트 시스템에서 애플리케이션 서버를 등록하고자 할 수 있습니다. 그러면 연결 컴포넌트에서 디자인 타임 시 애플리케이션 서버를 알 수 있으므로 Object Inspector의 드롭다운 리스트에서 서버 이름과 프로바이더 이름을 선택할 수 있습니다. 웹 연결, SOAP 연결 또는 소켓 연결을 사용할 경우 연결 컴포넌트는 서버 시스템에서 등록된 프로바이더 이름을 가져옵니다.

애플리케이션 서버 생성

대부분의 데이터베이스 애플리케이션을 만들 때처럼 애플리케이션 서버를 만듭니다. 가장 중요한 차이점은 애플리케이션 서버에서는 원격 데이터 모듈을 사용한다는 것입니다.

애플리케이션 서버를 만들려면 다음 단계를 따르십시오.

1 새 프로젝트를 시작합니다.

- 전송 프로토콜로 SOAP를 사용할 경우에는 새로운 웹 서비스 애플리케이션을 작성해야 합니다. File|New|Other를 선택하고 New Items 다이얼로그 박스의 WebServices 페이지에서 SOAP Server 애플리케이션을 선택합니다.
- 다른 전송 프로토콜의 경우에는 File|New|Application를 선택하면 됩니다.

새 프로젝트를 저장합니다.

2 프로젝트에 새 원격 데이터 모듈을 추가합니다. 메인 메뉴에서 File|New |Other를 선택하고 New Items 다이얼로그 박스의 MultiTier 페이지나 WebServices 페이지에서 다음을 선택합니다.

- **Remote Data Module.** 클라이언트에서 DCOM, HTTP 또는 소켓을 사용하여 액세스하는 COM Automation 서버를 만들 경우
- **Transactional Data Module.** MTS나 COM+에서 실행되는 원격 데이터 모듈을 만들 경우. DCOM, HTTP 또는 소켓을 사용하여 연결을 만들 수 있습니다. 그러나 DCOM에서만 보안 서비스를 지원합니다.
- **SOAP Server Data Module.** 웹 서비스 애플리케이션에서 SOAP 서버를 만들 경우

원격 데이터 모듈 설정에 대한 자세한 내용은 29-13페이지의 "원격 데이터 모듈 설정"을 참조하십시오.

참고 Remote Data Module이나 Transactional Data Module을 선택한 경우 마법사에서도 원격 데이터 모듈에 대한 참조가 포함된 특수한 COM Automation 객체를 만들고 이 객체를 사용하여 프로바이더를 찾습니다. 이 객체를 구현 객체라고 합니다. SOAP 데이터 모듈의 경우 데이터 모듈에서 *IAppServerSOAP* 인터페이스를 스스로 구현하기 때문에 구현 객체가 따로 필요하지 않습니다.

3 해당하는 데이터셋 컴포넌트를 데이터 모듈에 배치하고 데이터베이스 서버를 액세스하도록 설정합니다.

4 모든 데이터셋의 데이터 모듈에 *TDataSetProvider* 컴포넌트를 배치합니다. 클라이언트 요청을 중개하고 데이터를 패키지화하는 데 이 프로바이더가 필요합니다. 모든 프로바이더 컴포넌트의 *DataSet* 속성을 액세스할 데이터셋의 이름으로 설정합니다. 프로바이더의 다른 속성을 설정할 수 있습니다. 프로바이더 설정에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오.

XML 문서의 데이터를 사용할 경우 데이터셋과 *TDataSetProvider* 컴포넌트 대신 *TXMLTransformProvider* 컴포넌트를 사용할 수 있습니다. *TXMLTransformProvider*를 사용할 경우 *XMLDataFile* 속성을 설정하여 데이터가 제공되고 업데이트를 적용할 XML 문서를 지정합니다.

5 이벤트, 공유 비즈니스 룰, 공유 데이터 검증 및 공유 보안을 구현하기 위한 애플리케이션 서버 코드를 작성합니다. 이 코드를 작성할 때 다음을 원할 수 있습니다.

- 애플리케이션 서버의 인터페이스를 확장하여 클라이언트 애플리케이션에서 서버를 호출할 수 있는 다른 방법을 제공하고자 할 수 있습니다. 애플리케이션 서버의 인터페이스 확장은 29-16페이지의 "애플리케이션 서버의 인터페이스 확장"에서 설명합니다.
- 업데이트를 적용할 때 자동으로 만들어진 트랜잭션 이상의 트랜잭션 지원을 제공하고자 할 수 있습니다. 멀티 티어 데이터베이스 애플리케이션의 트랜잭션 지원은 29-17페이지의 "멀티 티어 애플리케이션의 트랜잭션 관리"에서 설명합니다.
- 애플리케이션 서버의 데이터셋 사이에 마스터/디테일 관계를 만들고자 할 수 있습니다. 마스터/디테일 관계는 29-18페이지의 "마스터/디테일 관계 지원"에서 설명합니다.
- 애플리케이션 서버가 stateless인지 확인하고자 할 수 있습니다. 상태 정보 처리는 29-18페이지의 "원격 데이터 모듈에서 상태 정보 지원"에서 설명합니다.
- 애플리케이션 서버를 여러 개의 원격 데이터 모듈로 나누고자 할 수 있습니다. 여러 원격 데이터 모듈 사용은 29-20페이지의 "여러 원격 데이터 모듈 사용"에서 설명합니다.

6 애플리케이션 서버를 저장, 컴파일, 등록 또는 설치합니다. 애플리케이션 서버 등록은 29-21페이지의 "애플리케이션 서버 등록"에서 설명합니다.

7 서버 애플리케이션에서 DCOM이나 SOAP를 사용하지 않을 경우 클라이언트 메시지를 받고, 원격 데이터 모듈을 인스턴스화하며, 인터페이스 호출을 마샬링하는 런타임 소프트웨어를 설치해야 합니다.

- TCP/IP 소켓의 경우에는 소켓 디스패처 애플리케이션인 Scktsrvr.exe입니다.
- HTTP 연결의 경우에는 웹 서버에서 설치한 ISAPI/NSAPI DLL인 httpsrvr.dll입니다.

원격 데이터 모듈 설정

원격 데이터 모듈을 만들 경우 클라이언트 요청에 응답하는 방법을 나타내는 특정 정보를 제공해야 합니다. 원격 데이터 모듈의 타입에 따라 이 정보가 다릅니다.

트랜잭션이 아닐 경우 원격 데이터 모듈 구성

트랜잭션 어트리뷰트(attribute)를 포함시키지 않고 COM 기반 원격 데이터 모듈을 애플리케이션에 추가하려면 File|New|Other를 선택한 다음 New Items 다이얼로그 박스의 Multitier 페이지에서 Remote Data Module을 선택합니다. 그러면 Remote Data Module 마법사가 나타납니다.

원격 데이터 모듈에 대한 클래스 이름을 제공해야 합니다. 이 이름이 애플리케이션에서 만든 *TCRemoteDataModule* 자손의 기본 이름입니다. 또한 애플리케이션 서버의 인스턴스의 기본 이름이기도 합니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정한 경우 마법사에서는 *TCRemoteDataModule*의 자손인 *TMyDataServer*를 선언하는 새로운 유닛을 만듭니다. 유닛 헤더에서 마법사는 *IAppServer*의 자손인 *IMyDataServer*를 구현하는 구현 클래스 (*TMyDataServerImpl*)도 선언합니다.

참고 새 인터페이스에 고유한 속성과 메소드를 추가할 수 있습니다. 자세한 내용은 29-16페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

Remote Data Module 마법사에서 스레드 모델을 지정해야 합니다. Single-threaded, Apartment-threaded, Free-threaded 또는 Both를 선택할 수 있습니다.

- Single-threaded를 선택한 경우 COM에서는 한 번에 하나의 클라이언트 요청만 처리하도록 합니다. 클라이언트 요청이 다른 클라이언트 요청을 방해하는지 걱정할 필요가 없습니다.
- Apartment-threaded를 선택한 경우 COM에서는 원격 데이터 모듈의 모든 인스턴스에서 한 번에 하나의 요청을 처리하도록 합니다. Apartment-threaded 라이브러리에 코드를 작성할 경우 전역 변수나 원격 데이터 모듈에 포함되지 않은 객체를 사용하면 스레드 충돌을 방지해야 합니다. BDE 호환 데이터셋을 사용할 경우 권장하는 모델입니다. BDE 호환 데이터셋의 스레드 문제를 처리하기 위해 *AutoSessionName* 속성을 **true**로 설정한 세션 컴포넌트가 필요할 것입니다.
- Free-threaded를 선택한 경우 애플리케이션은 일부 스레드에서 동시에 클라이언트 요청을 받을 수 있습니다. 애플리케이션에서 스레드가 안전한지 확인해야 합니다. 여러 클라이언트에서 원격 데이터 모듈을 동시에 액세스할 수 있기 때문에 전역 변수뿐만 아니라 인스턴스 데이터(속성, 포함된 객체 등)도 보호해야 합니다. ADO 데이터셋을 사용할 경우 권장하는 모델입니다.
- Both를 선택한 경우 라이브러리는 한 가지 점만 제외하고 Free-threaded를 선택했을 때와 같이 작동합니다. 즉, 모든 콜백(클라이언트 인터페이스 호출)이 자동으로 serialize됩니다.
- Neutral을 선택한 경우 Free-threaded 모델에서처럼 원격 데이터 모듈은 각각의 스레드에서 동시에 호출을 받을 수 있지만 COM에서는 두 개의 스레드가 동시에 같은 메소드를 액세스하지 못하게 보장합니다.

트랜잭션 원격 데이터 모듈 구성

MTS나 COM+를 사용할 때 애플리케이션에 원격 데이터 모듈을 추가하려면 File|New|Other를 선택한 다음 New Items 다이얼로그 박스의 Multitier 페이지에서 Transactional Data Module을 선택합니다. Transactional Data Module 마법사가 나타납니다.

원격 데이터 모듈의 클래스 이름을 제공해야 합니다. 이 이름이 애플리케이션에서 만든 *TCRemoteDataModule* 자손의 기본 이름입니다. 애플리케이션 서버의 인터페이스 기본 이름이기도 합니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정하면 마법사에서는 *TCRemoteDataModule*의 자손인 *TMyDataServer*를 선언하는 새로운 유닛을 만듭니다. 유닛 헤더에서 마법사는 *IMyDataServer*(*IAppServer*의 자손)와 *IObjectControl*(모든 트랜잭션 객체에 필요)을 모두 구현하는 구현 클래스(*TMyDataServerImpl*)도 선언합니다. *TMyDataServerImpl*에는 *IObjectContext* 인터페이스의 데이터 멤버가 포함됩니다. 이 인터페이스를 사용하여 트랜잭션을 관리하고 보안 등을 확인할 수 있습니다.

참고 새 인터페이스에 고유한 속성 및 메소드를 추가할 수 있습니다. 자세한 내용은 29-16페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

Transactional Data Module 마법사에서 스레드 모델을 지정해야 합니다. Single, Apartment 또는 Both를 선택합니다.

- Single을 선택하면 클라이언트 요청이 serialize되기 때문에 애플리케이션에서 한 번에 하나의 클라이언트 요청만 서비스합니다. 클라이언트 요청이 다른 클라이언트 요청을 방해하는지 걱정할 필요가 없습니다.
- Apartment를 선택한 경우 시스템에서는 원격 데이터 모듈의 모든 인스턴스에서 한 번에 하나의 요청을 처리하고 호출에서 항상 같은 스레드를 사용하도록 합니다. 전역 변수나 원격 데이터 모듈에 포함되지 않은 객체를 사용할 경우 스레드 충돌을 방지해야 합니다. 전역 변수를 사용하는 대신 공유 속성 관리자를 사용할 수 있습니다. 공유 속성 관리자에 대한 자세한 내용은 44-6페이지의 "Shared Property Manager"를 참조하십시오.
- Both를 선택하면 MTS에서는 Apartment를 선택했을 때와 같은 방법으로 애플리케이션 서버의 인터페이스로 호출합니다. 그러나 클라이언트 애플리케이션에 대한 콜백은 serialize되므로 서로 방해하는지 걱정할 필요는 없습니다.

참고 MTS나 COM+에서의 Apartment 모델은 DCOM에서의 Apartment 모델과 다릅니다.

원격 데이터 모듈의 트랜잭션 어트리뷰트(attribute)도 지정해야 합니다. 다음 옵션 중에서 선택할 수 있습니다.

- Requires a transaction. 이 옵션을 선택하면 클라이언트에서 개발자의 애플리케이션 서버의 인터페이스를 사용할 때마다 해당 호출은 트랜잭션의 컨텍스트에서 실행됩니다. 호출자가 트랜잭션을 제공하면 새로운 트랜잭션을 만들지 않아도 됩니다.
- Requires a new transaction. 이 옵션을 선택하면 클라이언트에서 개발자의 애플리케이션 서버의 인터페이스를 사용할 때마다 해당 호출에 대해 새로운 트랜잭션이 자동으로 만들어집니다.
- Supports transactions. 이 옵션을 선택한 경우 애플리케이션 서버를 트랜잭션의 컨텍스트에서 사용할 수 있지만 호출자는 인터페이스를 호출할 때 트랜잭션을 제공해야 합니다.
- Does not support transactions. 이 옵션을 선택한 경우 애플리케이션 서버를 트랜잭션의 컨텍스트에서 사용할 수 없습니다.

TSoapDataModule 구성

TSoapDataModule 컴포넌트를 애플리케이션에 추가하려면 File|New|Other를 선택한 다음 New Items 다이얼로그 박스의 WebServices 페이지에서 SOAP Server Data Module을 선택합니다. SOAP Data Module 마법사가 나타납니다.

SOAP 데이터 모듈에 대한 클래스 이름을 제공해야 합니다. 이 이름이 애플리케이션에서 만든 *TSoapDataModule* 자손의 기본 이름입니다. 예를 들어, 클래스 이름 *MyDataServer*를 지정할 경우 마법사에서 *TSoapDataModule*의 자손인 *TMyDataServer*를 선언하는 새 유닛을 만듭니다. 새로운 이 클래스는 *TSoapDataModule*에서 *IAppServer*와 *IAppServerSOAP* 구현을 상속받습니다.

다른 원격 데이터 모듈과는 달리 SOAP 데이터 모듈에서는 *IAppServer* 또는 *IAppServerSOAP* 의 자손인 사용자 정의 인터페이스를 구현하지 않습니다. 웹 서비스 애플리케이션에서 들어오는 인터페이스 호출을 디스패치하는 방법의 차이점 때문입니다. 대신 Add Web Service 마법사를 사용하여 애플리케이션에 새로운 인터페이스를 추가할 수 있습니다.

참고 *TSoapDataModule*을 사용하려면 새로운 데이터 모듈을 웹 서비스 애플리케이션에 추가해야 합니다. *IAppServerSOAP* 인터페이스는 인보커블 인터페이스로 새 유닛의 시작 코드에 등록합니다. 그러면 메인 웹 모듈의 인보커 컴포넌트에서 들어오는 모든 호출을 개발자의 데이터 모듈로 전달할 수 있습니다.

애플리케이션 서버의 인터페이스 확장

COM 기반 서버의 경우 클라이언트 애플리케이션은 구현 클래스를 만들거나 데이터 모듈 마법사에서 만든 구현 클래스에 연결하여 애플리케이션 서버와 상호 작용합니다 . 클라이언트 애플리케이션은 애플리케이션 서버의 인터페이스를 애플리케이션 서버와의 모든 통신의 기초로 사용합니다.

COM 기반 애플리케이션 서버를 사용할 경우 구현 클래스의 인터페이스에 추가하여 클라이언트 애플리케이션에 대한 추가 지원을 제공할 수 있습니다. 이 인터페이스는 *IAppServer*의 자손이며 원격 데이터 모듈을 만들 경우 마법사에서 자동으로 만듭니다.

구현 클래스의 인터페이스에 추가하려면 Type Library Editor를 사용합니다. Type Library Editor 사용에 대한 자세한 내용은 39장, "타입 라이브러리 사용"을 참조하십시오.

COM 인터페이스에 추가할 경우 변경 사항은 유닛 소스 코드와 타입 라이브러리 파일(.TLB)에 추가됩니다.

참고 Type Library Editor에서 Refresh를 선택한 다음 IDE에서 변경 사항을 저장하여 TLB 파일을 명시적으로 저장해야 합니다.

구현 클래스의 인터페이스에 추가했으면 구현 클래스에 추가한 속성과 메소드를 찾습니다. 새 메소드의 바디를 채워서 이 구현을 완료하기 위한 코드를 추가합니다.

클라이언트 애플리케이션은 연결 컴포넌트의 *AppServer* 속성을 사용하여 인터페이스 확장을 호출합니다. 이에 대한 자세한 내용은 29-27페이지의 "서버 인터페이스 호출"을 참조하십시오.

애플리케이션 서버의 인터페이스에 콜백 추가

콜백을 도입하여 애플리케이션 서버에서 클라이언트 애플리케이션을 호출하도록 할 수 있습니다. 이렇게 하려면 클라이언트 애플리케이션에서 애플리케이션 서버의 메소드 중 하나에 인터페이스를 전달하고 나중에 필요할 때 애플리케이션 서버에서 이 메소드를 호출합니다. 그러나 구현 클래스의 인터페이스에 대한 확장에 콜백이 포함될 경우 HTTP 또는 SOAP 기반 연결을 사용할 수 없습니다. *TWebConnection* 및 *TSoapConnection*에서는 콜백을 지원하지 않습니다. 소켓 기반 연결을 사용할 경우 클라이언트 애플리케이션에서는 *SupportCallbacks* 속성을 설정하여 콜백을 사용하는지 여부를 표시해야 합니다. 다른 모든 타입의 연결에서는 자동으로 콜백을 지원합니다.

트랜잭션 애플리케이션 서버의 인터페이스 확장

트랜잭션이나 just-in-time 활성화를 사용할 경우 새로운 모든 메소드에서 IObjectContext의 *SetComplete* 메소드를 호출하여 종료된 시기를 나타내도록 합니다. 그러면 트랜잭션을 완료하고 애플리케이션 서버를 비활성화할 수 있습니다.

그리고 안전한 참조를 제공하지 않으면 클라이언트가 애플리케이션 서버의 객체나 인터페이스와 직접 통신할 수 있게 해주는 새로운 메소드에서 값을 반환할 수 없습니다. stateless MTS 데이터 모듈을 사용할 경우 안전한 참조 사용을 무시하면 원격 데이터 모듈이 활성화되었는지 보장할 수 없기 때문에 중지될 수 있습니다. 안전한 참조에 대한 자세한 내용은 44-25페이지의 "객체 참조 전달"을 참조하십시오.

멀티 티어 애플리케이션의 트랜잭션 관리

클라이언트 애플리케이션에서 애플리케이션 서버에 업데이트를 적용할 경우 프로바이더 컴포넌트에서 업데이트 적용 및 트랜잭션의 오류 해결 프로세스를 자동으로 랩핑합니다. 문제 레코드의 수가 *ApplyUpdates* 메소드에 대한 인수로 지정된 *MaxErrors* 값을 초과하지 않을 경우 이 트랜잭션이 커밋됩니다. 그렇지 않으면 롤백됩니다.

그리고 데이터베이스 연결 컴포넌트를 추가하거나 SQL을 데이터베이스 서버에 보내 직접 트랜잭션을 관리하여 서버 애플리케이션에 트랜잭션 지원을 추가할 수 있습니다. 이것은 2티어 애플리케이션에서 트랜잭션을 관리하는 것과 똑같은 방법으로 작동합니다. 이런 종류의 트랜잭션 제어에 대한 자세한 내용은 21-6페이지의 "트랜잭션 관리"를 참조하십시오.

트랜잭션 데이터 모듈이 있으면 MTS나 COM+ 트랜잭션을 사용하여 트랜잭션 지원을 확대할 수 있습니다. 이러한 트랜잭션에는 애플리케이션 서버의 모든 비즈니스 로직이 포함될 수 있지만 데이터베이스 액세스는 포함되지 않습니다. 그리고 이러한 트랜잭션에서는 2단계 커밋을 지원하기 때문에 여러 데이터베이스에 걸칠 수 없습니다.

BDE 및 ADO 기반 데이터 액세스 컴포넌트에서는 2단계 커밋을 지원합니다. 여러 데이터베이스에 걸친 트랜잭션을 사용하려면 InterbaseExpress or dbExpress 컴포넌트를 사용하지 마십시오.

경고 BDE를 사용할 경우 Oracle7과 MS-SQL 데이터베이스에서만 2단계 커밋이 완전히 구현됩니다. 트랜잭션에 여러 데이터베이스가 포함되고 그 중 일부가 Oracle7이나 MS-SQL 외의 다른 원격 서버일 경우 트랜잭션에는 부분적으로만 구현되는 위험이 있습니다. 그러나 하나의 데이터베이스 내에서는 항상 트랜잭션 지원이 있습니다.

디폴트로, 트랜잭션 데이터 모듈의 모든 *IAppServer* 호출은 트랜잭션입니다. 트랜잭션에 참여하도록 표시하려면 데이터 모듈의 트랜잭션 어트리뷰트(attribute)를 설정하기만 하면 됩니다. 그리고 애플리케이션 서버의 인터페이스를 확장하여 개발자가 정의한 트랜잭션을 캡슐화하는 메소드 호출을 포함시킬 수 있습니다.

트랜잭션 어트리뷰트에서 애플리케이션 서버가 트랜잭션을 요구하도록 표시할 경우 클라이언트가 해당 인터페이스에서 메소드를 호출할 때마다 자동으로 트랜잭션을 랩핑합니다. 그러면 트랜잭션이 완료되었음을 나타낼 때까지 애플리케이션 서버에 대한 모든 클라이언트 호출이 해당 트랜잭션에 참여합니다. 이 호출은 전체적으로 성공하거나 롤백됩니다.

참고 MTS나 COM+ 트랜잭션을 데이터베이스 연결 컴포넌트에서 만들거나 명시적인 SQL 명령을 사용하여 만든 명시적인 트랜잭션과 결합하지 마십시오. 트랜잭션 데이터 모듈이 트랜잭션에 참여한 경우 데이터베이스 호출을 모두 트랜잭션에 자동으로 참여시킵니다.

MTS 및 COM+ 트랜잭션 사용에 대한 자세한 내용은 44-10페이지의 "MTS 및 COM+ 트랜잭션 지원"을 참조하십시오.

마스터/디테일 관계 지원

테이블 타입 데이터셋을 사용하여 설정하는 것과 같은 방법으로 클라이언트 애플리케이션의 클라이언트 데이터셋 사이에 마스터/디테일 관계를 만들 수 있습니다. 이런 방법으로 마스터/디테일 관계를 설정하는 것에 대한 자세한 내용은 22-34페이지의 "마스터/디테일 관계 생성"을 참조하십시오.

그러나 이 방법에는 다음과 같은 두 가지 중요한 단점이 있습니다.

- 한 번에 하나의 디테일 집합만 사용하더라도 디테일 테이블에서는 애플리케이션 서버의 모든 레코드를 가져오고 저장해야 합니다. 매개변수를 사용하여 이 문제를 줄일 수 있습니다. 자세한 내용은 27-28페이지의 "매개변수로 레코드 제한"을 참조하십시오.
- 클라이언트 데이터셋은 데이터셋 레벨에서 업데이트를 적용하고 마스터/디테일 업데이트는 여러 데이터셋에 걸치므로 업데이트를 적용하는 것이 매우 어렵습니다. 데이터베이스 연결 컴포넌트를 사용하여 여러 테이블에 대한 업데이트를 단일 트랜잭션으로 적용할 수 있는 2티어 환경일지라도 마스터/디테일 형태의 업데이트를 적용하는 것은 까다로운 일입니다.

멀티 티어 애플리케이션에서는 마스터/디테일 관계를 나타내는 중첩된 테이블을 사용하여 이런 문제를 방지할 수 있습니다. 데이터셋에서 제공할 때 이렇게 하려면 애플리케이션 서버의 데이터셋 사이에 마스터 / 디테일 관계를 설정합니다. 그런 다음 프로바이더 컴포넌트의 *DataSet* 속성을 마스터 테이블로 설정합니다. XML 문서에서 제공할 경우 중첩된 테이블을 사용하여 마스터/디테일 관계를 나타내려면 중첩된 디테일 집합을 정의하는 변환 파일을 사용합니다.

클라이언트에서 프로바이더의 *GetRecords* 메소드를 호출할 경우 디테일 데이터셋을 데이터 패킷의 레코드에 DataSet 필드로 자동으로 포함시킵니다. 클라이언트에서 프로바이더의 *ApplyUpdates* 메소드를 호출할 경우 해당하는 순서로 업데이트를 적용하는 것을 자동으로 처리합니다.

원격 데이터 모듈에서 상태 정보 지원

클라이언트 데이터셋에서 애플리케이션 서버의 프로바이더와 통신하기 위해 사용하는 *IAppServer* 인터페이스는 대부분 stateless입니다. 애플리케이션이 stateless일 경우 클라이언트의 이전 호출에서 발생한 사항을 전혀 "기억"하지 못합니다. 트랜잭션 데이터 모듈에서 데이터베이스 연결을 풀링할 경우 이 stateless 상태가 도움이 됩니다. 애플리케이션 서버에서는 레코드 통화와 같은 영구적 정보에 대해 데이터베이스 연결 사이를 구별하지 않아도 되기 때문입니다. 마찬가지로 여러 클라이언트 사이에서 원격 데이터 모듈 인스턴스를 공유할 경우에 just-in-time 활성화 또는 객체 풀링과 함께 발생하기 때문에 이 stateless 상태가 중요합니다. SOAP 데이터 모듈은 stateless여야 합니다.

그러나 애플리케이션 서버 호출 사이에서 상태 정보를 유지하고자 할 경우가 있습니다. 예를 들어, 증분 페치(fetch)를 사용하여 데이터를 요청할 경우 애플리케이션 서버의 프로바이더는 이전 호출의 정보(현재 레코드)를 기억해야 합니다.

클라이언트 데이터셋에서 만든 *IAppServer* 인터페이스 호출(AS_*ApplyUpdates*, AS_*Execute*, AS_*GetParams*, AS_*GetRecords* 또는 AS_*RowRequest*) 전후에 사용자 정의 상태 정보를 보내거나 받을 수 있는 이벤트를 받습니다. 마찬가지로 프로바이더에서 클라이언트가 만든 이러한 호출에 응답한 전후 프로바이더는 사용자 정의 상태 정보를 보내거나 받을 수 있는 이벤트를 받습니다. 이러한 방법을 사용하면 애플리케이션 서버가 stateless라 하더라도 클라이언트 애플리케이션과 애플리케이션 서버 사이에서 영구적 상태 정보를 통신할 수 있습니다.

예를 들어, 다음과 같은 매개변수화된 쿼리를 나타내는 데이터셋을 고려해보십시오.

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

stateless 애플리케이션 서버에서 증분 페치(fetch)를 사용하려면 다음을 수행하십시오.

- 프로바이더에서 레코드 집합을 데이터 패킷으로 패키지화할 경우 패킷에 있는 마지막 레코드의 CUST_NO 값에 주의합니다.

```
TRemoteDataModule1::DataSetProvider1GetData(TObject *Sender,
TCustomClientDataSet *DataSet)
{
  DataSet->Last(); // move to the last record
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  pProvider->Tag = DataSet->FieldValues["CUST_NO"];
}
```

- 데이터 패킷을 보낸 후 프로바이더는 이 마지막 CUST_NO 값을 클라이언트에 보냅니다.

```
TRemoteDataModule1::DataSetProvider1AfterGetRecords(TObject *Sender,
OleVariant &OwnerData)
{
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  OwnerData = pProvider->Tag;
}
```

- 클라이언트에서 클라이언트 데이터셋은 CUST_NO의 이 마지막 값을 저장합니다.

```
TDataModule1::ClientDataSet1AfterGetRecords(TObject *Sender, OleVariant
&OwnerData)
{
  TComponent *pDS = dynamic_cast<TComponent *>(Sender);
  pDS->Tag = OwnerData;
}
```

- 데이터 패킷을 가져오기 전에 클라이언트는 자신이 받은 CUST_NO의 마지막 값을 보냅니다.

```
TDataModule1::ClientDataSet1BeforeGetRecords(TObject *Sender, OleVariant
&OwnerData)
{
```

```
  TClientDataSet *pDS = dynamic_cast<TClientDataSet *>(Sender);
  if (!pDS->Active)
    return;
  OwnerData = pDS->Tag;
}
```

- 마지막으로 서버에서 프로바이더는 쿼리에서 최소 값으로 보낸 마지막 CUST_NO를 사용합니다.

```
TRemoteDataModule1::DataSetProvider1BeforeGetRecords(TObject *Sender,
OleVariant &OwnerData)
{
  if (!VarIsEmpty(OwnerData))
  {
    TDataSetProvider *pProv = dynamic_cast<TDataSetProvider *>(Sender);
    TSQLDataSet *pDS = (dynamic_cast<TSQLDataSet *>(pProv->DataSet);
    pDS->Params->ParamValues["MinVal"] = OwnerData;
    pDS->Refresh(); // force the query to reexecute
  }
}
```

여러 원격 데이터 모듈 사용

애플리케이션 서버에서 여러 원격 데이터 모듈을 사용하도록 구성할 수 있습니다. 여러 원격 데이터 모듈을 사용하면 코드를 분할할 수 있고 큰 애플리케이션 서버를 여러 유닛으로 구성할 수 있습니다. 여기서 각 유닛은 상대적으로 독립되어 있습니다.

독립적으로 작동하는 애플리케이션 서버에서는 항상 여러 원격 데이터 모듈을 만들 수 있지만, 컴포넌트 팔레트의 DataSnap 페이지에 있는 특수한 연결 컴포넌트에서는 다른 "자식" 원격 데이터 모듈로 클라이언트의 연결을 디스패치하는 메인 "부모" 원격 데이터 모듈을 하나 갖고 있는 모델에 대한 지원을 제공합니다. 이 모델에서는 COM 기반 애플리케이션 서버를 사용해야 합니다.

부모 원격 데이터 모듈을 만들려면 해당하는 *IAppServer* 인터페이스를 확장하고 자식 원격 데이터 모듈의 인터페이스를 노출하는 속성을 추가합니다. 즉 모든 자식 원격 데이터 모듈의 경우 값이 자식 데이터 모듈의 *IAppServer* 인터페이스인 부모 데이터 모듈에 속성을 추가합니다.

부모 원격 데이터 모듈의 인터페이스 확장에 대한 자세한 내용은 29-16페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

팁 자식 데이터 모듈의 인터페이스를 확장하고 부모 데이터 모듈의 인터페이스나 다른 자식 데이터 모듈의 인터페이스를 제공할 수도 있습니다. 그러면 애플리케이션 서버의 여러 데이터 모듈이 더 자유롭게 서로 통신할 수 있습니다.

자식 원격 데이터 모듈을 나타내는 속성을 메인 원격 데이터 모듈에 추가하면 클라이언트 애플리케이션이 애플리케이션 서버에 원격 데이터 모듈에 대한 각각의 연결을 만들지 않아도 됩니다. 대신 부모 원격 데이터 모듈에 대한 단일 연결을 공유하고 메시지를 "자식" 데이터 모듈에 디스패치합니다. 클라이언트 애플리케이션에서는 모든 원격 데이터 모듈에 대해 같은 연결을 사용하기 때문에 원격 데이터 모듈에서 단일 데이터베이스 연결을 공유하여 리소스를 절약할 수 있습니다 . 자식 애플리케이션에서 단일 연결을 공유하는 방법에 대한 자세한 내용은 29-28페이지의 "여러 데이터 모듈을 사용하는 애플리케이션 서버에 연결"을 참조하십시오.

애플리케이션 서버 등록

애플리케이션 서버를 등록하거나 설치해야 클라이언트 애플리케이션에서 애플리케이션 서버를 찾거나 사용할 수 있습니다.

- 애플리케이션 서버에서 통신 프로토콜로 DCOM, HTTP 또는 소켓을 사용할 경우 애플리케이션 서버는 Automation 서버의 역할을 하며 다른 COM 서버와 같이 등록해야 합니다. COM 서버 등록에 대한 자세한 내용은 41-16페이지의 "COM 객체 등록"을 참조하십시오.
- 트랜잭션 데이터 모듈을 사용할 경우 애플리케이션 서버를 등록하지 않습니다. 대신 MTS나 COM+로 애플리케이션 서버를 설치합니다. 트랜잭션 객체 설치에 대한 자세한 내용은 44-27페이지의 "트랜잭션 객체 설치"를 참조하십시오.
- 애플리케이션 서버에서 SOAP 를 사용할 경우 애플리케이션은 웹 서비스 애플리케이션이어야 합니다. 또한 웹 서버에 등록해야만 들어오는 HTTP 메시지를 받을 수 있습니다. 애플리케이션의 인보커블 인터페이스를 설명하는 WSDL 문서를 게시해야 합니다. 웹 서비스 애플리케이션을 위한 WSDL 문서 엑스포트에 대한 자세한 내용은 36-15 페이지의 " 웹 서비스 애플리케이션을 위한 WSDL 문서 생성"을 참조하십시오.

클라이언트 애플리케이션 생성

멀티 티어 클라이언트 애플리케이션의 생성 방법은 클라이언트 데이터셋을 사용하여 업데이트를 캐싱하는 2티어 클라이언트 생성 방법과 거의 비슷합니다. 중요한 차이점은 멀티 티어 클라이언트에서는 연결 컴포넌트를 사용하여 애플리케이션 서버에 대한 매개체를 만든다는 것입니다.

멀티 티어 클라이언트 애플리케이션을 만들려면 새 프로젝트를 시작하고 다음 단계를 따르십시오.

1. 새로운 데이터 모듈을 프로젝트에 추가합니다.
2. 데이터 모듈에 연결 컴포넌트를 배치합니다. 사용할 통신 프로토콜에 따라 추가한 연결 컴포넌트 타입이 다릅니다. 자세한 내용은 29-4페이지의 "클라이언트 애플리케이션의 구조"를 참조하십시오.
3. 연결 컴포넌트의 속성을 설정하여 연결을 만들 애플리케이션 서버를 지정합니다. 연결 컴포넌트 설정에 대한 자세한 내용은 29-22페이지의 "애플리케이션 서버에 연결"을 참조하십시오.
4. 애플리케이션에서 필요한 대로 다른 연결 컴포넌트 속성을 설정합니다. 예를 들어, 연결 컴포넌트가 여러 서버에서 동적으로 선택할 수 있도록 *ObjectBroker* 속성을 설정할 수 있습니다. 연결 컴포넌트 사용에 대한 자세한 내용은 29-26페이지의 "서버 연결 관리"를 참조하십시오.

5 데이터 모듈에 필요한 대로 *TClientDataSet* 컴포넌트를 배치하고, 각 컴포넌트의 *RemoteServer* 속성을 단계 2에서 배치한 연결 컴포넌트의 이름으로 설정합니다. 클라이언트 데이터셋에 대한 자세한 소개는 27장, "클라이언트 데이터셋 사용"을 참조하십시오.

6 *TClientDataSet* 컴포넌트의 *ProviderName* 속성을 설정합니다. 디자인 타임 시 연결 컴포넌트를 애플리케이션 서버에 연결한 경우 *ProviderName* 속성의 드롭다운 리스트에서 사용 가능한 애플리케이션 서버 프로바이더를 선택할 수 있습니다.

7 다른 데이터베이스 애플리케이션을 만들 때와 같은 방법으로 계속 진행합니다. 멀티 티어 애플리케이션의 클라이언트에서 사용할 수 있는 다른 몇 가지 기능이 있습니다.

- 애플리케이션에서 애플리케이션 서버를 직접 호출할 수 있습니다. 29-27페이지의 "서버 인터페이스 호출"에서 이 방법을 설명합니다.
- 프로바이더 컴포넌트와의 상호 작용을 지원하는 클라이언트 데이터셋의 특수 기능을 사용할 수 있습니다. 이 내용은 27-24페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"에 설명되어 있습니다.

애플리케이션 서버에 연결

애플리케이션 서버에 연결하고 연결을 유지하려면 클라이언트 애플리케이션에서 하나 이상의 연결 컴포넌트를 사용해야 합니다. 컴포넌트 팔레트의 DataSnap 페이지나 WebServices 페이지에 이러한 컴포넌트가 있습니다.

연결 컴포넌트를 사용하면 다음과 같은 작업을 수행할 수 있습니다.

- 애플리케이션 서버와 통신하기 위한 프로토콜을 식별할 수 있습니다. 각각의 연결 컴포넌트 타입은 서로 다른 통신 프로토콜을 나타냅니다. 사용 가능한 프로토콜의 장점과 제한에 대한 자세한 내용은 29-9페이지의 "연결 프로토콜 선택"을 참조하십시오.
- 서버 시스템을 찾는 방법을 나타낼 수 있습니다. 서버 시스템을 식별하는 자세한 내용은 프로토콜에 따라 다릅니다.
- 서버 시스템의 애플리케이션 서버를 식별할 수 있습니다.
- SOAP를 사용하지 않을 경우 *ServerName* 속성이나 *ServerGUID* 속성을 사용하여 서버를 식별할 수 있습니다. *ServerName*은 애플리케이션 서버에 원격 데이터 모듈을 만들 때 지정한 클래스의 기본 이름을 식별합니다. 서버에 이러한 값을 지정하는 방법에 대한 자세한 내용은 29-13페이지의 "원격 데이터 모듈 설정"을 참조하십시오. 클라이언트 시스템에 서버를 등록 또는 설치하거나 연결 컴포넌트를 서버 시스템에 연결한 경우 Object Inspector의 드롭다운 리스트에서 선택하여 디자인 타임 시 *ServerName* 속성을 설정할 수 있습니다. *ServerGUID*는 원격 데이터 모듈의 인터페이스의 GUID를 지정합니다. Type Library Editor를 사용하면 이 값을 알아낼 수 있습니다.

  SOAP를 사용할 경우 서버 시스템을 찾기 위해 사용한 URL로 서버가 식별됩니다. 29-25페이지의 "SOAP를 사용하여 연결 지정" 단계를 수행합니다.
- 서버 연결을 관리할 수 있습니다. 연결 컴포넌트를 사용하여 연결을 만들거나 끊고, 애플리케이션 서버 인터페이스를 호출할 수 있습니다.

대개 애플리케이션 서버는 클라이언트 애플리케이션과는 다른 시스템에 있습니다. 그러나 서버가 클라이언트 애플리케이션과 같은 시스템에 상주하더라도 연결 컴포넌트를 사용하여 이름으로 애플리케이션 서버를 식별할 수 있고, 서버 시스템을 지정할 수 있으며, 애플리케이션 서버 인터페이스를 사용할 수 있습니다.

DCOM을 사용하여 연결 지정

DCOM을 사용하여 애플리케이션 서버와 통신할 경우 클라이언트 애플리케이션에는 애플리케이션 서버에 연결하기 위한 *TDCOMConnection* 컴포넌트가 포함됩니다. *TDCOMConnection*에서는 *ComputerName* 속성을 사용하여 서버가 상주하는 시스템을 식별합니다.

*ComputerName*이 공백일 경우 DCOM 연결 컴포넌트에서는 애플리케이션 서버가 클라이언트 시스템에 상주하거나 애플리케이션 서버에 시스템 레지스트리 항목이 있음을 가정합니다. DCOM을 사용할 때 클라이언트의 애플리케이션 서버에 대한 시스템 레지스트리 항목을 제공하지 않고 서버가 클라이언트와 다른 시스템에 상주할 경우 *ComputerName*을 제공해야 합니다.

참고 애플리케이션 서버에 대한 시스템 레지스트리 항목이 있더라도 *ComputerName*을 지정하여 이 항목을 오버라이드할 수 있습니다. 개발, 테스트 및 디버깅 중에 특히 유용합니다.

클라이언트 애플리케이션에서 선택할 수 있는 서버가 여러 개일 경우 *ComputerName*에 대한 값을 지정하는 대신 *ObjectBroker* 속성을 사용할 수 있습니다. 자세한 내용은 29-25페이지의 "연결 브로커"를 참조하십시오.

찾을 수 없는 서버나 호스트 컴퓨터의 이름을 제공한 경우 연결을 열려고 하면 DCOM 연결 컴포넌트에서 예외를 발생시킵니다.

소켓을 사용하여 연결 지정

TCP/IP 주소를 가진 시스템의 소켓을 사용하여 애플리케이션 서버에 연결할 수 있습니다. 이 메소드는 더 많은 시스템에 적용할 수 있는 장점이 있지만 보안 프로토콜 사용은 허용하지 않습니다. 소켓을 사용할 경우 애플리케이션 서버에 연결하려면 *TSocketConnection* 컴포넌트를 포함시키십시오.

*TSocketConnection*에서는 IP 주소나 서버 시스템의 호스트 이름을 사용하는 서버 시스템과 서버 시스템에서 실행되는 소켓 디스패처 프로그램(Scktsrvr.exe)의 포트 번호를 식별합니다. IP 주소와 포트 값에 대한 자세한 내용은 37-3페이지의 "소켓 설명"을 참조하십시오.

*TSocketConnection*의 세 속성에서 다음 정보를 지정합니다.

- *Address*는 서버의 IP 주소를 지정합니다.
- *Host*에서는 서버의 호스트 이름을 지정합니다.
- *Port*는 애플리케이션 서버에 있는 소켓 디스패처 프로그램의 포트 번호를 지정합니다.

*Address*와 *Host*는 같이 사용할 수 없습니다. 하나를 설정하면 다른 속성의 값 설정이 해제됩니다. 사용할 속성에 대한 자세한 내용은 37-4페이지의 "호스트 설명"을 참조하십시오.

클라이언트 애플리케이션에서 선택할 수 있는 서버가 여러 개일 경우 *Address*나 *Host*에 대한 값을 지정하는 대신 *ObjectBroker* 속성을 사용할 수 있습니다. 자세한 내용은 29-25페이지의 "연결 브로커"를 참조하십시오.

*Port*의 기본값은 211입니다. 이것은 들어오는 메시지를 애플리케이션 서버에 전달하는 소켓 디스패처 프로그램의 디폴트 포트 번호입니다. 소켓 디스패처에서 다른 포트를 사용하도록 구성한 경우 해당 값에 일치하도록 *Port* 속성을 설정합니다.

참고 Borland Socket Server 트레이 아이콘을 마우스 오른쪽 버튼으로 클릭하고 Properties를 선택하여 실행 중인 소켓 디스패처의 포트를 구성할 수 있습니다.

소켓 연결에서 보안 프로토콜 사용을 허용하지 않더라도 소켓 연결을 사용자 정의하여 고유의 암호화를 추가할 수 있습니다. 다음 단계를 수행하십시오.

1. *IDataIntercept* 인터페이스를 지원하는 COM 객체를 만듭니다. 이것이 데이터를 암호화 및 해독하기 위한 인터페이스입니다.
2. 새로운 COM 서버를 클라이언트 시스템에 등록합니다.
3. 소켓 연결 컴포넌트의 *InterceptName* 또는 *InterceptGUID* 속성을 설정하여 이 COM 객체를 지정합니다.
4. 마지막으로 Borland Socket Server 트레이 아이콘을 마우스 오른쪽 버튼으로 클릭하고, Properties를 선택한 다음, Properties 탭에서 Intercept Name이나 Intercept GUID를 인터셉터의 ProgId 또는 GUID로 설정합니다.

데이터 압축과 압축 해제에 이 방법을 사용할 수도 있습니다.

HTTP를 사용하여 연결 지정

TCP/IP 주소를 가진 시스템에서 HTTP를 사용하여 애플리케이션 서버에 연결할 수 있습니다. 그러나 소켓과는 달리 HTTP를 사용하면 SSL 보안의 장점을 활용할 수 있고 방화벽으로 보호된 서버와 통신할 수 있습니다. HTTP를 사용할 경우 애플리케이션 서버에 연결하려면 *TWebConnection* 컴포넌트를 포함시킵니다.

웹 연결 컴포넌트는 차례로 애플리케이션 서버와 통신하는 웹 서버 애플리케이션(httpsrvr.dll)에 연결합니다. *TWebConnection*에서는 URL(Uniform Resource Locator)을 사용하여 httpsrvr.dll을 찾습니다. URL에서는 프로토콜(http 또는 SSL 보안을 사용할 경우 https), 웹 서버와 httpsrvr.dll을 실행하는 시스템의 호스트 이름, 웹 서버 애플리케이션(httpsrvr.dll)의 경로를 지정합니다. *URL* 속성을 사용하여 이 값을 지정합니다.

참고 *TWebConnection*을 사용할 경우 클라이언트 시스템에 wininet.dll을 설치해야 합니다. IE3 이상을 설치한 경우 wininet.dll은 Windows 시스템 디렉토리에 있습니다.

웹 서버에서 인증을 요구하는 경우 또는 인증을 요구하는 프록시 서버를 사용할 경우 연결 컴포넌트에서 로그온할 수 있도록 *UserName* 속성과 *Password* 속성의 값을 설정해야 합니다.

클라이언트 애플리케이션에서 선택할 수 있는 서버가 여러 개일 경우 *URL* 의 값을 지정하는 대신 *ObjectBroker* 속성을 사용할 수 있습니다. 자세한 내용은 29-25페이지의 "연결 브로커"를 참조하십시오.

SOAP를 사용하여 연결 지정

TSoapConnection 컴포넌트를 사용하여 SOAP 애플리케이션 서버에 연결할 수 있습니다. *TSoapConnection*에서도 전송 프로토콜로 HTTP를 사용하기 때문에 *TWebConnection*과 매우 유사합니다. 따라서 TCP/IP 주소를 가진 시스템에서 *TSoapConnection*을 사용할 수 있고, SSL 보안의 장점을 활용하여 방화벽으로 보호된 서버와 통신할 수 있습니다.

SOAP 연결 컴포넌트는 *IAppServerSOAP* 인터페이스나 *IAppServer* 인터페이스를 구현하는 웹 서비스 프로바이더에 연결합니다. *UseSOAPAdapter* 속성은 서버에서 지원할 것으로 예상되는 인터페이스를 지정합니다. 서버에서 *IAppServerSOAP* 인터페이스를 구현할 경우 *TSoapConnection*에서 해당 인터페이스를 클라이언트 데이터셋의 *IAppServer* 인터페이스로 변환합니다. *TSoapConnection*에서는 URL(Uniform Resource Locator)을 사용하여 웹 서버 애플리케이션을 찾습니다. URL에서는 프로토콜(http 또는 SSL 보안을 사용할 경우 https), 웹 서버를 실행하는 시스템의 호스트 이름, 웹 서비스 애플리케이션의 이름, 애플리케이션 서버에 있는 *THTTPSoapDispatcher*의 경로 이름과 일치하는 경로를 지정합니다. *URL* 속성을 사용하여 이 값을 지정합니다.

참고 *TSoapConnection*을 사용할 경우 클라이언트 시스템에 wininet.dll을 설치해야 합니다. IE3 이상을 설치한 경우 wininet.dll은 Windows 시스템 디렉토리에 있습니다.

웹 서버에서 인증을 요구하는 경우 또는 인증을 요구하는 프록시 서버를 사용할 경우 연결 컴포넌트에서 로그온할 수 있도록 <~JMP *UserName* 속성과 *Password* 속성의 값을 설정해야 합니다.

연결 브로커

클라이언트 애플리케이션에서 선택할 수 있는 COM 기반 서버가 여러 개일 경우 Object Broker를 사용하여 사용할 수 있는 서버 시스템을 찾을 수 있습니다. Object Broker는 연결 컴포넌트에서 선택할 수 있는 서버 리스트를 유지합니다. 연결 컴포넌트가 애플리케이션 서버에 연결해야 할 경우 Object Broker에서 컴퓨터 이름이나 IP 주소, 호스트 이름 또는 URL을 묻습니다. 브로커에서 이름을 제공하고, 연결 컴포넌트에서 연결합니다. 제공한 이름이 작동하지 않으면(예를 들어, 서버가 다운된 경우) 연결이 될 때까지 브로커에서 다른 이름을 제공합니다.

연결 컴포넌트가 브로커에서 제공한 이름으로 연결을 하면 연결 컴포넌트는 해당하는 속성(*ComputerName*, *Address*, *Host*, *RemoteHost* 또는 *URL*)의 값으로 해당 이름을 저장합니다. 나중에 연결 컴포넌트가 연결을 닫고 다시 연결을 열어야 할 경우 이 속성 값을 사용하여 열고, 연결이 실패한 경우에만 브로커에게 새로운 이름을 요청합니다.

연결 컴포넌트의 *ObjectBroker* 속성을 지정하여 Object Broker를 사용합니다. *ObjectBroker* 속성을 설정하면 연결 컴포넌트에서는 *ComputerName*, *Address*, *Host*, *RemoteHost* 또는 *URL*의 값을 디스크에 저장하지 않습니다.

서버 연결 관리

연결 컴포넌트의 기본 목적은 애플리케이션 서버를 찾아 연결하는 것입니다. 연결 컴포넌트에서 서버 연결을 관리하기 때문에 연결 컴포넌트를 사용하여 애플리케이션 서버의 인터페이스의 메소드를 호출할 수 있습니다.

서버에 연결

애플리케이션 서버를 찾아 연결하려면 먼저 연결 컴포넌트의 속성을 설정하여 애플리케이션 서버를 식별해야 합니다.이 프로세스는 29-22페이지의 "애플리케이션 서버에 연결"에 설명되어 있습니다. 연결 컴포넌트를 사용하여 애플리케이션 서버와 통신하는 모든 클라이언트 데이터셋은 연결을 열기 전에 *RemoteServer* 속성에서 연결 컴포넌트를 설정해야 합니다.

클라이언트 데이터셋이 애플리케이션 서버를 액세스하려고 시도하면 자동으로 연결이 열립니다. 예를 들어, 클라이언트 데이터셋의 *Active* 속성을 **true**로 설정하면 *RemoteServer* 속성이 설정되어 있기만 하면 연결이 열립니다.

클라이언트 데이터셋을 연결 컴포넌트에 연결하지 않은 경우 연결 컴포넌트의 *Connected* 속성을 **true**로 설정하여 연결을 열 수 있습니다.

연결 컴포넌트는 애플리케이션 서버에 연결하기 전에 *BeforeConnect* 이벤트를 생성합니다. 개발자는 자신이 코딩한 *BeforeConnect* 핸들러에서 연결하기 전의 특수한 작업을 수행할 수 있습니다. 연결한 후에도 연결 컴포넌트는 특수한 작업을 위해 *AfterConnect* 이벤트를 생성합니다.

서버 연결 끊기 또는 변경

다음과 같은 경우 연결 컴포넌트는 애플리케이션 서버 연결을 끊습니다.

- *Connected* 속성을 **false**로 설정한 경우
- 연결 컴포넌트를 해제한 경우. 클라이언트 애플리케이션을 닫을 때 연결 객체가 자동으로 해제된 경우
- 애플리케이션 서버를 식별하는 속성(*ServerName*, *ServerGUID*, *ComputerName* 등)을 변경한 경우. 이 속성을 변경하면 런타임 시 사용할 수 있는 애플리케이션 서버 사이에서 전환할 수 있습니다. 연결 컴포넌트에서 현재 연결을 끊고 새로 연결합니다.

참고 하나의 연결 컴포넌트를 사용하여 사용 가능한 애플리케이션 서버 사이에서 전환하는 대신 클라이언트 애플리케이션에 둘 이상의 연결 컴포넌트가 있을 수 있습니다. 각각의 연결 컴포넌트는 서로 다른 애플리케이션 서버에 연결됩니다.

연결 컴포넌트에서 연결을 끊기 전에 자동으로 해당하는 *BeforeDisconnect* 이벤트 핸들러를 호출합니다. 연결을 끊기 전에 특정 액션을 수행하려면 *BeforeDisconnect* 핸들러를 작성합니다. 마찬가지로 연결을 끊은 후에 *AfterDisconnect* 이벤트 핸들러를 호출합니다. 연결을 끊은 후에 특정 액션을 수행하려면 *AfterDisconnect* 핸들러를 작성합니다.

서버 인터페이스 호출

클라이언트 데이터셋의 속성과 메소드를 사용할 경우 *IAppServer* 인터페이스를 자동으로 호출하기 때문에 애플리케이션에서 직접 호출하지 않아도 됩니다. 그러나 *IAppServer* 인터페이스를 직접 작업할 필요는 없지만, SOAP를 사용하지 않을 경우고유의 확장자를 애플리케이션 서버의 인터페이스에 추가했을 수 있습니다. 애플리케이션 서버의 인터페이스를 확장할 경우 연결 컴포넌트에서 만든 연결을 사용하여 확장을 호출할 수 있는 방법이 필요합니다. 연결 컴포넌트의 *AppServer* 속성을 사용하여 이렇게 할 수 있습니다. 애플리케이션 서버의 인터페이스 확장에 대한 자세한 내용은 29-16페이지의 "애플리케이션 서버의 인터페이스 확장"을 참조하십시오.

*AppServer*는 애플리케이션 서버의 인터페이스를 나타내는 Variant입니다. 이 인터페이스를 호출하려면 이 Variant에서 디스패치 인터페이스를 가져와야 합니다. 디스패치 인터페이스는 원격 데이터 모듈을 만들 때 만들어진 인터페이스와 같은 이름이지만 "Disp" 문자열이 추가된 이름을 갖습니다. 따라서 원격 데이터 모듈을 MyAppServer라고 할 경우 다음과 같이 *AppServer*를 사용하여 해당 인터페이스를 호출할 수 있습니다.

```
IDispatch* disp = (IDispatch*)(MyConnection->AppServer)
IMyAppServerDisp TempInterface( (IMyAppServer*)disp);
TempInterface.SpecialMethod(x,y);
```

참고 디스패치 인터페이스는 Type Library Editor에서 만든 _TLB.h 파일에 선언되어 있습니다.

SOAP를 사용할 경우 *AppServer* 속성을 사용할 수 없습니다. 대신 원격 인터페이스 객체(*THTTPRio*)를 사용하고 초기 연결 호출을 해야 합니다. 모든 초기 연결 호출에서처럼 클라이언트 애플리케이션은 컴파일 시 애플리케이션 서버의 인터페이스 선언을 알고 있어야 합니다. 호출할 인터페이스를 설명하는 WSDL 문서를 참조하여 클라이언트 애플리케이션에 이를 추가할 수 있습니다. SOAP 서버의 경우 이 인터페이스는 SOAP 데이터 모듈의 인터페이스와 완전히 별개입니다. 인터페이스를 설명하는 WSDL 문서 임포트에 대한 자세한 내용은 36-16페이지의 "WSDL 문서 임포트"를 참조하십시오.

참고 서버 인터페이스를 선언하는 유닛도 인보케이션 레지스트리에 자신을 등록해야 합니다. 인보커블 인터페이스를 등록하는 방법에 대한 자세한 내용은 36-2 페이지의 " 인보커블 (invocable) 인터페이스 이해"를 참조하십시오.

WSDL 문서를 임포트하여 인터페이스를 선언하고 등록하는 유닛을 생성했으면 원하는 인터페이스에 대해 *THTTPRio*의 인스턴스를 만듭니다.

```
THTTPRio *X = new THTTPRio(NULL);
```

그 다음, 연결 컴포넌트에서 사용하는 URL을 원격 인터페이스 객체에 할당하고 호출할 인터페이스 이름을 추가합니다.

```
X,>URL = SoapConnection1.URL + "IMyInterface";
```

이제 QueryInterface 메소드를 사용하여 서버의 메소드를 호출할 인터페이스를 가져올 수 있습니다.

```
InterfaceVariable = X->QueryInterface(IMyInterfaceIntf);
if (InterfaceVariable)
{
  InterfaceVariable->SpecialMethod(a,b);
}
```

QueryInterface 호출에서는 인보커블 인터페이스 자체보다는 인보커블 인터페이스의 *DelphiInterface* 랩퍼를 인수로 갖습니다.

여러 데이터 모듈을 사용하는 애플리케이션 서버에 연결

29-20페이지의 "여러 원격 데이터 모듈 사용"에서 설명한 대로 COM 기반 애플리케이션 서버에서 메인 "부모" 원격 데이터 모듈과 여러 개의 자식 원격 데이터 모듈을 사용할 경우 애플리케이션 서버의 모든 원격 데이터 모듈에 대한 각각의 연결 컴포넌트가 필요합니다. 각각의 연결 컴포넌트는 단일 원격 데이터 모듈에 대한 연결을 나타냅니다.

클라이언트 애플리케이션에서 애플리케이션 서버의 원격 데이터 모듈에 독립적으로 연결할 수 있지만 모든 연결 컴포넌트에서 공유하는 애플리케이션 서버에 단일 연결을 사용하는 것이 더 효율적입니다. 즉, 애플리케이션 서버의 "메인" 원격 데이터 모듈에 연결하는 단일 연결 컴포넌트를 추가한 다음, 각각의 "자식" 원격 데이터 모듈의 경우 메인 원격 데이터 모듈에 대한 연결을 공유하는 다른 컴포넌트를 추가합니다.

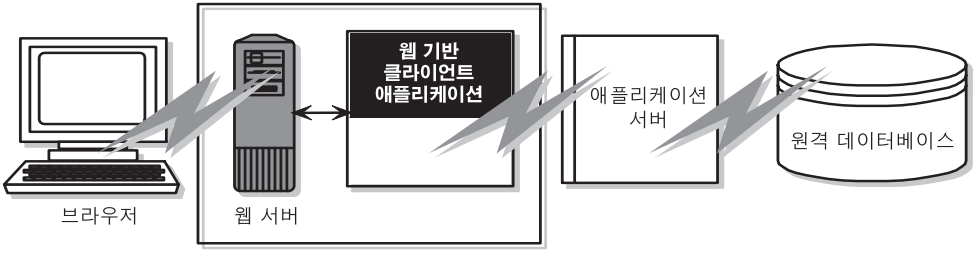
1. 메인 원격 데이터 모듈에 대한 연결의 경우 29-22페이지의 "애플리케이션 서버에 연결"에서 설명한 대로 연결 컴포넌트를 추가하고 설정합니다. 유일한 제한은 SOAP 연결을 사용할 수 없다는 것입니다.
2. 자식 원격 데이터 모듈의 경우 *TSharedConnection* 컴포넌트를 사용합니다.
   - *ParentConnection* 속성을 단계 1에서 추가한 연결 컴포넌트로 설정합니다. *TSharedConnection* 컴포넌트는 이 메인 연결에서 설정한 연결을 공유합니다.
   - *ChildName* 속성을 원하는 자식 원격 데이터 모듈의 인터페이스를 노출하는 메인 원격 데이터 모듈의 인터페이스에 있는 속성 이름으로 설정합니다.

단계 2에서 배치한 *TSharedConnection* 컴포넌트를 클라이언트 데이터셋의 *RemoteServer* 속성 값으로 할당한 경우 자식 원격 데이터 모듈에 대해 완전히 독립적인 연결을 사용하는 것처럼 작동합니다. 그러나 *TSharedConnection* 컴포넌트에서는 단계 1에서 배치한 컴포넌트에서 설정한 연결을 사용합니다.

웹 기반 클라이언트 애플리케이션 생성

멀티 티어 데이터베이스 애플리케이션에 대한 웹 기반 클라이언트를 만들 경우 클라이언트 티어를 특수한 웹 애플리케이션으로 바꿔야 합니다. 이 웹 애플리케이션은 애플리케이션 서버에 대해 클라이언트 역할을 함과 동시에 동일한 시스템에 웹 서버와 함께 설치된 웹 서버 애플리케이션의 역할을 합니다. 이 아키텍처에 대해서는 그림 29.1에서 설명합니다.

그림 29.1 웹 기반 멀티 티어 데이터베이스 애플리케이션



웹 애플리케이션을 생성하기 위해 다음 두 가지 방법을 사용할 수 있습니다.

- 멀티 티어 데이터베이스 아키텍처를 ActiveX 폼과 결합하여 클라이언트 애플리케이션을 ActiveX 컨트롤로 분산시킬 수 있습니다. 그러면 ActiveX를 지원하는 모든 브라우저에서 클라이언트 애플리케이션을 in-process 서버로 실행할 수 있습니다.
- XML 데이터 패킷을 사용하여 InternetExpress 애플리케이션을 생성할 수 있습니다. 그러면 Javascript를 지원하는 브라우저에서 html 페이지를 통해 클라이언트 애플리케이션과 상호작용할 수 있습니다.

이 두 가지 방법은 매우 다릅니다. 다음 사항에 따라 선택하는 방법이 다릅니다.

- 각 방법에서 서로 다른 기술(ActiveX 대 Javascript 및 XML)을 사용합니다. 엔드 유저가 사용할 시스템을 고려하십시오. 첫 번째 방법에서는 ActiveX를 지원하는 브라우저가 필요합니다. ActiveX는 클라이언트를 Windows 플랫폼으로 제한합니다. 두 번째 방법에서는 javascript와 Netscape 4 및 Internet Explorer 4에서 소개하는 DHTML 기능을 지원하는 브라우저가 필요합니다.
- ActiveX 컨트롤을 in-process 서버 역할을 하는 브라우저로 다운로드해야 합니다. 그 결과, ActiveX 방법을 사용하는 클라이언트에는 HTML 기반 애플리케이션의 클라이언트보다 더 많은 메모리가 필요합니다.
- InternetExpress 방법을 다른 HTML 페이지와 통합할 수 있습니다. ActiveX 클라이언트는 각각의 윈도우에서 실행해야 합니다.
- InternetExpress 방법에서는 표준 HTTP를 사용하여 ActiveX 애플리케이션에서 부딪히는 방화벽 문제를 방지합니다.
- ActiveX 방법에서는 애플리케이션을 프로그래밍하는 방법에 있어서 더 많은 융통성을 제공합니다. Javascript 라이브러리의 기능에 의해 제한을 받지 않습니다. ActiveX 방법에서 사용한 클라이언트 데이터셋에서는 InternetExpress 방법에서 사용한 XML 브로커보다 더 많은 기능(예: 필터, 범위, 집계, 옵션 매개변수, BLOB 또는 중첩된 디테일의 가져오기 지연 등)을 제공합니다.

주의 웹 클라이언트 애플리케이션이 다른 브라우저에서 볼 때와는 다르게 보이거나 다르게 작동할 수 있습니다. 엔드 유저가 사용할 브라우저로 애플리케이션을 테스트합니다.

클라이언트 애플리케이션을 ActiveX 컨트롤로 분산

멀티 티어 데이터베이스 아키텍처를 ActiveX 기능과 결합하여 클라이언트 애플리케이션을 ActiveX 컨트롤로 분산시킬 수 있습니다.

클라이언트 애플리케이션을 ActiveX 컨트롤로 분산시킬 경우 다른 멀티 티어 애플리케이션에서처럼 애플리케이션 서버를 만듭니다. 애플리케이션 서버 생성에 대한 자세한 내용은 29-12페이지의 "애플리케이션 서버 생성"을 참조하십시오.

클라이언트 애플리케이션을 만들 경우 일반 폼 대신 Active Form을 기본으로 사용해야 합니다. 자세한 내용은 "클라이언트 애플리케이션용 Active Form 생성"을 참조하십시오.

클라이언트 애플리케이션을 생성 및 배포했으면 다른 시스템에 있는 ActiveX를 사용할 수 있는 웹 브라우저에서 액세스할 수 있습니다. 웹 브라우저에서 클라이언트 애플리케이션을 성공적으로 시작하려면 웹 서버를 클라이언트 애플리케이션을 갖고 있는 시스템에서 실행해야 합니다.

클라이언트 애플리케이션이 DCOM을 사용하여 클라이언트 애플리케이션과 애플리케이션 서버 사이에서 통신할 경우 웹 브라우저가 있는 시스템을 DCOM과 같이 사용할 수 있어야 합니다. 웹 브라우저가 있는 시스템이 Windows 95 시스템일 경우 Microsoft의 DCOM95를 설치해야 합니다.

클라이언트 애플리케이션용 Active Form 생성

1. 클라이언트 애플리케이션은 ActiveX 컨트롤로 배포되기 때문에 같은 시스템에서 클라이언트 애플리케이션으로 실행되는 서버가 있어야 합니다. Microsoft Personal Web Server와 같이 미리 만들어진 서버를 사용하거나 37장, "소켓 작업"에서 설명한 소켓 컴포넌트를 사용하여 고유한 서버를 작성할 수 있습니다.
2. 일반적인 클라이언트 프로젝트를 시작하는 대신 File|New|ActiveX|Active Form을 선택하여 시작한다는 점만 제외하면 29-21페이지의 "클라이언트 애플리케이션 생성"에서 설명한 단계를 수행하는 클라이언트 애플리케이션을 만듭니다.
3. 클라이언트 애플리케이션에서 데이터 모듈을 사용할 경우 호출을 추가하여 활성 폼 초기화에서 데이터 모듈을 명시적으로 만듭니다.
4. 클라이언트 애플리케이션이 끝나면 프로젝트를 컴파일하고, Project | Web Deployment Options를 선택합니다. Web Deployment Options 다이얼로그 박스에서 다음을 수행합니다.
   1. Project 페이지에서 Target 디렉토리, 대상 디렉토리의 URL 및 HTML 디렉토리를 지정합니다. 일반적으로 Target 디렉토리와 HTML 디렉토리는 웹 서버의 프로젝트 디렉토리와 똑같습니다. 대상 URL은 대개 서버 시스템의 이름입니다.
   2. Additional Files 페이지에서 midas.dll을 클라이언트 애플리케이션에 포함시킵니다.
5. 마지막으로 Project|WebDeploy를 선택하여 클라이언트 애플리케이션을 활성 폼으로 배포합니다.

Active Form을 실행할 수 있는 웹 브라우저에서는 클라이언트 애플리케이션을 배포할 때 만든 HTM 파일을 지정하여 클라이언트 애플리케이션을 실행할 수 있습니다. 이 .HTM 파일은 클라이언트 애플리케이션 프로젝트와 같은 이름을 갖고 있으며 Target 디렉토리로 지정한 디렉토리에 나타납니다.

InternetExpress를 이용한 웹 애플리케이션 개발

클라이언트 애플리케이션에서는 애플리케이션 서버에서 OleVariants 대신 XML로 코딩된 데이터 패킷을 제공하도록 요청할 수 있습니다. XML 코딩 데이터 패킷, 데이터베이스 기능의 특수한 javascript 라이브러리 및 웹 서버 애플리케이션 지원을 결합하면 javascript를 지원하는 웹 브라우저를 사용하여 액세스할 수 있는 씬 클라이언트 애플리케이션을 만들 수 있습니다. 이러한 기능의 결합을 InternetExpress라고 합니다.

InternetExpress 애플리케이션을 생성하기 전에 웹 서버 애플리케이션 아키텍처를 알고 있어야 합니다. 이 내용은 32장, "인터넷 서버 애플리케이션 생성"에서 설명합니다.

InternetExpress 애플리케이션은 기본 웹 서버 애플리케이션 아키텍처를 확장하여 애플리케이션 서버의 클라이언트 역할을 합니다. InternetExpress 애플리케이션에서는 HTML, XML 및 javascript가 혼합된 HTML 페이지를 만듭니다. HTML은 엔드 유저가 자신의 브라우저로 볼 수 있는 페이지의 레이아웃과 모양을 관리합니다. XML은 데이터베이스 정보를 나타내는 데이터 패킷과 델타 패킷을 암호화합니다. Javascript를 사용하면 HTML 컨트롤이 클라이언트 시스템에서 XML 데이터 패킷의 데이터를 해석하고 처리할 수 있습니다.

InternetExpress 애플리케이션에서 DCOM을 사용하여 애플리케이션 서버에 연결할 경우 애플리케이션 서버에서 해당 클라이언트에게 액세스 및 시작 권한을 부여하도록 추가 단계를 수행해야 합니다. 자세한 내용은 29-33페이지의 "애플리케이션 서버를 액세스 및 시작할 수 있는 권한 부여"를 참조하십시오

팁 애플리케이션 서버가 없더라도 InternetExpress 애플리케이션을 만들어 웹 브라우저에 "라이브" 데이터를 제공할 수 있습니다. 프로바이더와 해당 데이터셋을 웹 모듈에 추가하기만 하면 됩니다.

InternetExpress 애플리케이션 생성

다음 단계에서는 InternetExpress를 사용하여 웹 애플리케이션을 생성하는 한 가지 방법을 설명합니다. 이 방법을 사용하면 사용자가 javascript 사용 웹 브라우저를 통해 애플리케이션 서버의 데이터를 액세스할 수 있는 HTML 페이지를 만드는 애플리케이션이 생성됩니다.
InternetExpress 페이지 프로듀서(*TInetXPageProducer*)를 사용하여 Site Express 아키텍처를 사용하는 InternetExpress 애플리케이션을 생성할 수도 있습니다.

1. File|New|Other 를 선택하여 New Items 다이얼로그 박스를 표시한 다음, New 페이지에서 Web Server 애플리케이션을 선택합니다. 이 프로세스는 33-1페이지의 "Web Broker를 사용하여 웹 서버 애플리케이션 생성"에 설명되어 있습니다.
2. 컴포넌트 팔레트의 DataSnap 페이지에서 새로운 웹 서버 애플리케이션을 만들 때 나타나는 웹 모듈에 연결 컴포넌트를 추가합니다. 사용할 통신 프로토콜에 따라 추가한 연결 컴포넌트의 타입이 다릅니다. 자세한 내용은 29-9페이지의 "연결 프로토콜 선택"을 참조하십시오.

3. 연결 컴포넌트에서 연결할 애플리케이션 서버를 지정하도록 연결 컴포넌트의 속성을 설정합니다. 연결 컴포넌트 설정에 대한 자세한 내용을 배우려면 29-22페이지의 "애플리케이션 서버에 연결"을 참조하십시오.
4. 클라이언트 데이터셋 대신 컴포넌트 팔레트의 InternetExpress 페이지에서 XML 브로커를 웹 모듈에 추가합니다. *TClientDataSet*처럼 *TXMLBroker*는 애플리케이션 서버의 프로바이더의 데이터를 나타내며 *IAppServer* 인터페이스를 통해 애플리케이션 서버를 사용합니다. 그러나 클라이언트 데이터셋과는 달리 XML 브로커에서는 데이터 패킷을 OleVariants 대신 XML로 요청하고, 데이터 컨트롤 대신 InternetExpress 컴포넌트를 사용합니다.
5. XML 브로커의 *RemoteServer* 속성을 단계 2에서 추가한 연결 컴포넌트를 가리키도록 설정합니다. *ProviderName* 속성을 데이터를 제공하고 업데이트를 적용하는 애플리케이션 서버의 프로바이더를 가리키도록 설정합니다. XML 브로커 설정에 대한 자세한 내용은 29-34페이지의 "XML 브로커 사용"을 참조하십시오.
6. 사용자의 브라우저로 표시되는 각 페이지의 웹 모듈에 InternetExpress 페이지 프로듀서(*TInetXPageProducer*)를 추가합니다. 각 페이지 프로듀서마다 페이지 프로듀서에서 만든 HTML 컨트롤을 데이터 관리 기능으로 증대시키는 javascript 라이브러리가 있는 곳을 가리키도록 *IncludePathURL* 속성을 설정합니다.
7. Web 페이지를 마우스 오른쪽 버튼으로 클릭하고 액션 에디터를 선택하여 표시합니다. 브라우저에서 처리할 모든 메시지에 대한 액션 항목을 추가합니다. 페이지 프로듀서의 *Producer* 속성을 설정하거나 *OnAction* 이벤트 핸들러에서 코드를 작성하여 단계 6에서 추가한 페이지 프로듀서를 이 액션과 연결시킵니다. 액션 에디터를 사용한 액션 항목 추가에 대한 자세한 내용은 33-5페이지의 "디스패처에 액션 추가"를 참조하십시오.
8. Web 페이지를 더블 클릭하여 웹 페이지 에디터를 표시합니다. *WebPageItems* 속성 옆에 있는 Object Inspector의 생략 부호 버튼을 클릭하여 웹 페이지 에디터를 표시할 수도 있습니다. 웹 페이지 에디터에서 웹 항목을 추가하여 사용자의 브라우저로 표시되는 페이지를 디자인할 수 있습니다. InternetExpress 애플리케이션의 웹 페이지 디자인에 대한 자세한 내용은 29-36페이지의 "InternetExpress 페이지 프로듀서로 웹 페이지 생성"을 참조하십시오.
9. 웹 애플리케이션을 생성합니다. 웹 서버로 웹 애플리케이션을 설치하면 브라우저에서 애플리케이션 이름을 URL의 스크립트 이름 부분으로 지정하고 웹 페이지 컴포넌트 이름을 경로 부분으로 지정하여 이 애플리케이션을 호출할 수 있습니다.

Javascript 라이브러리 사용

InternetExpress 컴포넌트에서 만든 HTML 페이지와 InternetExpress에 포함되어 있는 웹 항목에서는 source/webmidas 디렉토리에 제공된 다음과 같은 여러 javascript 라이브러리를 사용합니다.

표 29.3 Javascript 라이브러리

라이브러리	설명
xmldom.js	Javascript로 작성된 DOM 호환 XML 파서입니다. 이 라이브러리를 사용하면 XML을 지원하지 않는 파서에서 XML 데이터 패킷을 사용할 수 있습니다. IE5 이상에서 지원되는 XML Islands에 대한 지원은 포함되지 않습니다.
xmldb.js	XML 데이터 패킷과 XML 델타 패킷을 관리하는 데이터 액세스 클래스를 정의합니다.
xmldisp.js	Xmldb의 데이터 액세스 클래스와 HTML 페이지의 HTML 컨트롤을 연결하는 클래스를 정의합니다.
xmlerrdisp.js	업데이트를 오류를 조정할 때 사용할 수 있는 클래스를 정의합니다. 기본 InternetExpress 컴포넌트에서는 이 클래스를 사용하지 않지만 조정 프로듀서를 작성할 때 유용합니다.
xmlshow.js	서식화된 XML 데이터 패킷과 XML 델타 패킷을 표시하는 함수를 포함합니다. InternetExpress 컴포넌트에서는 사용하지 않지만 디버깅할 때 유용합니다.

이러한 라이브러리를 설치했으면 모든 InternetExpress 페이지 프로듀서의 *IncludePathURL* 속성을 라이브러리가 있는 곳을 가리키도록 설정해야 합니다.

웹 항목을 사용하여 웹 페이지를 만드는 대신 이 라이브러리에서 제공하는 javascript 클래스를 사용하여 고유한 HTML 페이지를 작성할 수 있습니다. 그러나 코드에서 오류를 수행하지 않게 확인해야 합니다. 생성된 웹 페이지의 크기를 최소화하기 위해 이 클래스에는 최소한의 오류 검사만 포함되어 있기 때문입니다.

애플리케이션 서버를 액세스 및 시작할 수 있는 권한 부여

애플리케이션 서버에게는 InternetExpress 애플리케이션의 요청이 IUSR_computername 이름을 가진 게스트 계정에서 시작된 것처럼 보입니다. 여기서 computername은 웹 애플리케이션을 실행하는 시스템의 이름입니다. 디폴트로, 이 계정에는 애플리케이션 서버에 대한 액세스 권한이나 시작 권한이 없습니다. 이런 권한을 부여하지 않고 웹 애플리케이션을 사용할 경우 웹 브라우저에서 요청한 페이지를 로드하려고 하면 EOLE_ACCESS_ERROR으로 시간이 종료됩니다.

참고 애플리케이션 서버가 이 게스트 계정으로 실행되기 때문에 다른 계정에서 애플리케이션 서버를 종료시킬 수 없습니다.

웹 애플리케이션 액세스 및 시작 권한을 부여하려면 애플리케이션 서버를 실행하는 시스템의 System32 디렉토리에 있는 DCOMCnfg.exe를 실행합니다. 다음 단계에서 애플리케이션 서버 구성 방법을 설명합니다.

1. DCOMCnfg 를 실행할 경우 Applications 페이지의 애플리케이션 리스트에서 애플리케이션 서버를 선택합니다.
2. Properties 버튼을 클릭합니다. 다이얼로그 박스가 바뀌면 Security 페이지를 선택합니다.

3 Use Custom Access Permissions를 선택하고 Edit 버튼을 누릅니다. 액세스 권한이 있는 계정 리스트에 IUSR_computername 이름을 추가합니다. 여기서 computername은 웹 애플리케이션을 실행하는 시스템 이름입니다.

4 Use Custom Launch Permissions를 선택하고 Edit 버튼을 누릅니다. 이 리스트에도 IUSR_computername을 추가합니다.

5 Apply 버튼을 클릭합니다.

XML 브로커 사용

XML 브로커는 다음과 같은 두 가지 중요한 기능을 제공합니다.

- 애플리케이션 서버에서 XML 데이터 패킷을 가져와서 InternetExpress 애플리케이션용 HTML을 만드는 웹 항목에서 사용할 수 있게 합니다.
- 브라우저에서 XML 델타 패킷 형태로 업데이트를 받아 애플리케이션 서버에 적용합니다.

XML 데이터 패킷 가져오기

XML 브로커가 HTML 페이지를 생성하는 컴포넌트에 XML 데이터 패킷을 제공하려면 애플리케이션 서버에서 XML 데이터 패킷을 가져와야 합니다. 그렇게 하려면 *IAppServer* 인터페이스를 사용합니다. 이 인터페이스는 연결 컴포넌트에서 가져옵니다.

참고 애플리케이션 서버에서 *IAppServerSOAP*을 지원하는 SOAP를 사용할 경우에도 XML 브로커에서는 *IAppServer*를 사용합니다. 연결 컴포넌트가 두 인터페이스 사이에서 어댑터 역할을 하기 때문입니다.

다음 속성을 설정해야 XML 프로듀서에서 *IAppServer* 인터페이스를 사용할 수 있습니다.

- *RemoteServer* 속성을 애플리케이션 서버에 연결하고 *IAppServer* 인터페이스를 가져오는 연결 컴포넌트로 설정합니다. 디자인 타임 시 Object Inspector의 드롭다운 리스트에서 이 값을 선택할 수 있습니다.
- *ProviderName* 속성을 원하는 XML 데이터 패킷의 데이터셋을 나타내는 애플리케이션 서버의 프로바이더 컴포넌트 이름으로 설정합니다. 이 프로바이더에서 XML 데이터 패킷을 제공하고 XML 데이터 패킷의 업데이트를 적용합니다. 디자인 타임 시 *RemoteServer* 속성을 설정하고 연결 컴포넌트에 활성 연결이 있으면 Object Inspector는 사용 가능한 프로바이더 리스트를 표시합니다. DCOM 연결을 사용할 경우 클라이언트 시스템에도 애플리케이션 서버를 등록해야 합니다.

다음 두 속성을 사용하여 데이터 패킷에 포함시킬 내용을 지정할 수 있습니다.

- *MaxRecords* 속성을 설정하여 데이터 패킷에 추가한 레코드의 수를 제한할 수 있습니다. InternetExpress 애플리케이션에서 전체 데이터 패킷을 클라이언트 웹 브라우저로 보내기 때문에 큰 데이터셋인 경우에 특히 중요합니다. 데이터 패킷이 너무 크면 다운로드 시간이 너무 길어질 수 있습니다.
- 애플리케이션 서버의 프로바이더가 쿼리나 내장 프로시저를 나타낼 경우 XML 데이터 패킷을 가져오기 전에 매개변수 값을 제공할 수 있습니다. *Params* 속성을 사용하여 이러한 매개변수 값을 제공할 수 있습니다.

XMLBroker 속성이 설정되어 있으면 InternetExpress 애플리케이션용 HTML과 javascript를 만드는 컴포넌트에서는 자동으로 XML 브로커의 XML 데이터 패킷을 사용합니다. 코드에서 직접 XML 데이터 패킷을 가져오려면 *RequestRecords* 메소드를 사용합니다.

참고 XML 브로커에서 데이터 패킷을 다른 컴포넌트에게 제공하거나 *RequestRecords*를 호출할 경우 *OnRequestRecords* 이벤트를 받게 됩니다. 이 이벤트를 사용하여 애플리케이션 서버의 데이터 패킷 대신 고유한 XML 문자열을 제공할 수 있습니다. 예를 들어, *GetXMLRecords*를 사용하여 애플리케이션 서버에서 데이터 패킷을 가져온 다음 표시된 웹 페이지에 제공하기 전에 편집할 수 있습니다.

XML 델타 패킷의 업데이트 적용

XML 브로커를 웹 모듈이나 *TWebDispatcher*가 포함된 데이터 모듈에 추가한 경우 웹 디스패처에 자동 디스패칭 객체로 자동으로 등록됩니다. 이것은 다른 컴포넌트와는 달리 XML 브로커가 웹 브라우저의 업데이트 메시지에 응답하기 위해 액션 항목을 만들지 않아도 된다는 것을 의미합니다. 이 메시지에는 애플리케이션 서버에 적용해야 할 XML 델타 패킷이 포함되어 있습니다. 대개는 웹 클라이언트 애플리케이션에서 만든 HTML 페이지 중 하나에 있는 버튼으로 시작됩니다.

디스패처에서 XML 브로커의 메시지를 알 수 있게 *WebDispatch* 속성을 사용하여 메시지를 설명해야 합니다. *PathInfo* 속성을 XML 브로커 메시지를 보낼 URL의 경로 부분으로 설정합니다. *MethodType*을 URL에 전달한 업데이트 메시지의 메소드 헤더 값(대개는 *mtPost*)으로 설정합니다. 지정된 경로를 가진 모든 메시지에 응답하려면 *MethodType*을 *mtAny*로 설정합니다.
XML 브로커에서 직접 업데이트 메시지에 응답하게 하려면(예를 들어, 액션 항목을 사용하여 명시적으로 처리하게 할 경우) *Enabled* 속성을 **false**로 설정합니다. 웹 디스패처에서 웹 브라우저의 메시지를 처리할 컴포넌트를 결정하는 방법에 대한 자세한 내용은 33-5페이지의 "요청 메시지 디스패칭"을 참조하십시오.

디스패처에서 업데이트 메시지를 XML 브로커에 전달할 경우 애플리케이션 서버에 업데이트를 전달합니다. 업데이트 오류가 있으면 모든 업데이트 오류를 설명하는 XML 델타 패킷을 받습니다. 마지막으로 응답 메시지를 다시 브라우저로 보냅니다. 브라우저를 XML 델타 패킷을 만든 같은 페이지로 리디렉션하거나 약간 새로운 내용을 보냅니다.

다음과 같은 이벤트를 사용하면 업데이트 프로세스의 모든 단계에서 사용자 정의 처리를 삽입할 수 있습니다.

1. 디스패처에서 먼저 업데이트 메시지를 XML 브로커로 보낸 경우 *BeforeDispatch* 이벤트를 받습니다. 이 이벤트에서는 요청을 사전 처리하거나 완전히 처리할 수 있습니다. 이 이벤트를 사용하면 XML 브로커는 업데이트 메시지 외의 메시지를 처리할 수 있습니다.
2. *BeforeDispatch* 이벤트 핸들러에서 메시지를 처리하지 않을 경우 XML 브로커는 *OnRequestUpdate* 이벤트를 받습니다. 이 이벤트에서는 디폴트 프로세싱을 사용하는 대신 스스로 업데이트를 적용할 수 있습니다.
3. *OnRequestUpdate* 이벤트 핸들러에서 요청을 처리하지 않을 경우 XML 브로커는 업데이트를 적용하고 업데이트 오류가 포함된 델타 패킷을 받습니다.

4 업데이트 오류가 없을 경우 XML 브로커는 *OnGetResponse* 이벤트를 받습니다. 이 이벤트에서는 업데이트가 성공적으로 적용되었음을 나타내거나 새로 고친 데이터를 브라우저에 보내는 응답 메시지를 만들 수 있습니다. *OnGetResponse* 이벤트 핸들러에서 응답을 완료하지 않을 경우(*Handled* 매개변수를 **true**로 설정하지 않은 경우) XML 브로커는 브라우저를 델타 패킷을 만든 문서로 다시 리디렉션하는 응답을 보냅니다.

5 업데이트 오류가 있으면 XML 브로커는 대신 *OnGetErrorResponse* 이벤트를 받습니다. 이 이벤트를 사용하여 업데이트 오류를 해결하거나 엔드 유저에게 업데이트 오류를 설명하는 웹 페이지를 만들 수 있습니다. *OnGetErrorResponse* 이벤트 핸들러에서 응답을 완료하지 않은 경우(*Handled* 매개변수를 **true**로 설정하지 않은 경우) XML 브로커는 *ReconcileProducer*라고 하는 특수 컨텐트 프로듀서에서 호출하여 응답 메시지의 내용을 만듭니다.

6 마지막으로 XML 브로커는 *AfterDispatch* 이벤트를 받습니다. 이 이벤트에서는 응답을 웹 브라우저로 다시 보내기 전에 마지막 액션을 수행할 수 있습니다.

InternetExpress 페이지 프로듀서로 웹 페이지 생성

모든 InternetExpress 페이지 프로듀서는 애플리케이션의 클라이언트 브라우저로 표시되는 HTML 문서를 만듭니다. 애플리케이션에 여러 개의 웹 문서가 포함된 경우 각각의 페이지 프로듀서를 사용합니다.

InternetExpress 페이지 프로듀서(*TInetXPageProducer*)는 특수한 페이지 프로듀서 컴포넌트입니다. 다른 페이지 프로듀서에서처럼 이 프로듀서를 액션 항목의 *Producer* 속성으로 할당하거나 *OnAction* 이벤트 핸들러에서 명시적으로 호출할 수 있습니다. 액션 항목을 이용한 컨텐트 프로듀서 사용에 대한 자세한 내용은 33-8페이지의 "액션 항목으로 요청 메시지에 응답"을 참조하십시오. 페이지 프로듀서에 대한 자세한 내용은 33-14페이지의 "페이지 프로듀서 컴포넌트 사용"을 참조하십시오.

InternetExpress 페이지 프로듀서에는 *HTMLDoc* 속성의 값인 디폴트 템플릿이 있습니다. 이 템플릿에는 InternetExpress 페이지 프로듀서에서 다른 컴포넌트에서 만든 컨텐트를 포함한 HTML 문서(포함된 javascript와 XML이 있는)를 어셈블링하는 데 사용하는 HTML 투명 태그 집합이 포함되어 있습니다. InternetExpress 페이지 프로듀서에서 모든 HTML 투명 태그를 변환하고 이 문서를 어셈블링하려면 페이지의 포함된 javascript에 사용한 javascript 라이브러리 위치를 표시해야 합니다. *IncludePathURL* 속성을 설정하여 이 위치를 지정합니다.

웹 페이지 에디터를 사용하여 웹 페이지 일부를 만드는 컴포넌트를 지정할 수 있습니다. 웹 페이지 컴포넌트를 더블 클릭하거나 Object Inspector의 *WebPageItems* 속성 옆에 있는 생략 부호 버튼을 클릭하여 웹 페이지 에디터를 표시합니다.

웹 페이지 에디터에서 추가한 컴포넌트에서 InternetExpress 페이지 프로듀서의 디폴트 템플릿에 있는 HTML 투명 태그를 대체하는 HTML을 만듭니다. 이 컴포넌트가 *WebPageItems* 속성의 값이 됩니다. 원하는 순서대로 컴포넌트를 추가한 후 템플릿을 사용자 정의하여 고유한 HTML을 추가하거나 디폴트 태그를 변경할 수 있습니다.

웹 페이지 에디터 사용

웹 페이지 에디터를 사용하면 InternetExpress 페이지 프로듀서에 웹 항목을 추가하고 만들어진 HTML 페이지를 볼 수 있습니다. InternetExpress 페이지 프로듀서 컴포넌트에서 더블 클릭하여 웹 페이지 에디터를 표시합니다.

참고 웹 페이지 에디터를 사용하려면 Internet Explorer 4 이상을 설치해야 합니다.

웹 페이지 에디터의 맨 위에는 HTML 문서를 생성하는 웹 항목들이 표시됩니다. 이 웹 항목들은 중첩되어 있으며, 여기서 각 타입의 웹 항목은 각각의 하위 항목에서 생성된 HTML로 구성됩니다. 서로 다른 타입의 항목에는 다른 하위 항목들이 들어 있을 수 있습니다. 왼쪽의 트리 뷰는 모든 웹 항목을 표시하고 중첩된 모양을 보여 줍니다. 오른쪽에는 현재 선택된 항목에 포함되어 있는 웹 항목이 표시됩니다. 웹 페이지 에디터의 맨 위에서 컴포넌트를 선택하면 Object Inspector를 사용하여 해당 속성을 설정할 수 있습니다.

New Item 버튼을 클릭하여 하위 항목을 현재 선택된 항목에 추가합니다. Add Web Component 다이얼로그 박스는 현재 선택된 항목에 추가할 수 있는 항목만 나열합니다.

InternetExpress 페이지 프로듀서에는 두 가지 타입의 항목 중 하나가 포함될 수 있는데, 각각 다음과 같은 HTML 폼을 만듭니다.

- *TDataForm*. 데이터를 표시하기 위한 HTML 폼과 데이터를 처리하거나 업데이트를 제출하는 컨트롤을 만듭니다.

  *TDataForm*에 추가한 항목은 복수 레코드 그리드(*TDataGrid*)나 단일 레코드의 단일 필드를 나타내는 컨트롤 집합(*TFieldGroup*)으로 데이터를 표시합니다. 그리고 데이터를 탐색하거나 업데이트를 포스트(*TDataNavigator*)하기 위해 버튼 집합을 추가하거나, 웹 클라이언트에 업데이트를 다시 적용하기 위한 버튼(*TApplyUpdatesButton*)을 추가할 수 있습니다. 각 항목에는 개별 필드나 버튼을 나타내기 위한 하위 항목이 포함되어 있습니다. 마지막으로 대부분의 웹 항목에서처럼 포함된 항목의 레이아웃을 사용자 정의할 수 있게 해주는 레이아웃 그리드(*TLayoutGroup*)를 추가할 수 있습니다.

- *TQueryForm*. 애플리케이션에서 정의한 값을 표시하거나 읽기 위한 HTML 폼을 만듭니다. 예를 들어, 이 폼을 사용하여 매개변수 값을 표시하거나 제출할 수 있습니다.

  *TQueryForm*에 추가한 항목은 애플리케이션에서 정의한 값(*TQueryFieldGroup*)이나 해당 값을 제출하거나 다시 설정하기 위한 버튼 집합(*TQueryButtons*)을 표시합니다. 각 항목에는 개별 값이나 버튼을 나타내는 하위 항목이 포함되어 있습니다. 데이터 폼에 할 수 있는 것처럼 쿼리 폼에도 레이아웃 그리드를 추가할 수 있습니다.

웹 페이지 에디터 아래쪽에는 생성된 HTML 코드가 표시되고 브라우저(IE4)로 표시되는 모양을 볼 수 있습니다.

웹 항목 속성 설정

웹 페이지 에디터를 사용하여 추가한 웹 항목은 HTML을 생성하는 특화된 컴포넌트입니다. 모든 웹 항목 클래스는 최종 HTML 문서의 특정 컨트롤이나 섹션을 만들도록 설계되었지만, 일반적인 속성 집합은 최종 HTML의 모양에 영향을 줍니다.

웹 항목에서 XML 데이터 패킷의 정보를 나타낼 경우(예를 들어, 필드나 매개변수 표시 컨트롤 집합이나 데이터를 처리하는 버튼을 만들 경우) *XMLBroker* 속성은 웹 항목을 데이터 패킷을 관리하는 XML 브로커와 연결시킵니다. *XMLDataSetField* 속성을 사용하여 해당 데이터 패킷의 데이터셋 필드에 포함된 데이터셋을 지정할 수도 있습니다. 웹 항목이 특정 필드나 매개변수 값을 나타낼 경우 웹 항목에는 *FieldName* 속성이나 *ParamName* 속성이 있습니다.

웹 항목에 스타일 어트리뷰트(attribute)를 적용하여 웹 항목에서 만든 모든 HTML의 전체적인 모양에 영향을 미칠 수 있습니다. 스타일과 스타일 시트는 HTML 4 표준의 일부입니다. 이를 사용하면 HTML 문서에서 태그에 적용되는 표시 어트리뷰트 집합과 유효 범위에 있는 모든 것을 정의할 수 있습니다. 웹 항목을 사용할 방법을 융통성있게 선택할 수 있습니다.

- 스타일을 사용할 수 있는 가장 간단한 방법은 웹 항목에 직접 스타일 어트리뷰트를 정의하는 것입니다. 이렇게 하려면 *Style* 어트리뷰트를 사용합니다. 다음과 같이 *Style*의 값은 표준 HTML 스타일 정의의 속성 정의 부분입니다.
  color: red.
- 스타일 정의 집합을 정의하는 스타일 시트를 정의할 수도 있습니다. 각 정의에는 스타일 선택자(항상 스타일이 적용되는 태그 이름이나 사용자가 정의한 스타일 이름)와 중괄호로 묶인 어트리뷰트 정의가 포함됩니다.
  ```
  H2 B  {color: red}
  .MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
  ```
  정의의 전체 집합은 InternetExpress 페이지 프로듀서에서 *Styles* 속성으로 보관합니다. 웹 항목에서 *StyleRule* 속성을 설정하여 사용자 정의한 이름으로 스타일을 참조할 수 있습니다.
- 스타일 시트를 다른 애플리케이션과 공유할 경우 *Styles* 속성 대신 InternetExpress 페이지 프로듀서의 *StylesFile* 속성 값으로 스타일 정의를 제공할 수 있습니다. 개별 웹 항목에서는 *StyleRule* 속성을 사용하여 계속 스타일을 참조합니다.

웹 항목의 또 다른 일반적인 속성이 *Custom* 속성입니다. *Custom*에는 생성된 HTML 태그에 추가한 옵션 집합이 포함되어 있습니다. HTML에서는 각 타입의 태그에 대해 서로 다른 옵션 집합을 정의합니다. 대부분 웹 항목의 *Custom* 속성에 대한 VCL 참조에서는 사용 가능한 옵션 예제를 제공합니다. 사용 가능한 옵션에 대한 자세한 내용은 HTML 참조를 사용하십시오.

InternetExpress 페이지 프로듀서 템플릿 사용자 정의

InternetExpress 페이지 프로듀서의 템플릿은 애플리케이션에서 동적으로 변환하는 포함된 태그가 있는 HTML 문서입니다. 처음에 페이지 프로듀서는 디폴트 템플릿을 *HTMLDoc* 속성의 값으로 만듭니다. 이 디폴트 템플릿의 형식은 다음과 같습니다.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

디폴트 템플릿의 HTML 투명 태그는 다음과 같이 변환됩니다.

<#INCLUDES> Javascript 라이브러리를 포함하는 명령문을 만듭니다. 이 명령문의 형식은 다음과 같습니다.

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/
xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/
xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/
xmlbind.js"> </SCRIPT>
```

<#STYLES> InternetExpress 페이지 프로듀서의 *Styles* 속성이나 *StylesFile* 속성에 나열된 정의로부터 스타일 시트를 정의하는 명령문을 만듭니다.

<#WARNINGS> 런타임 시 아무 것도 생성하지 않습니다. 디자인 타임 시 HTML 문서를 생성하는 중에 발생한 문제에 대한 경고 메시지를 추가합니다. 웹 페이지 에디터에서 이 메시지를 볼 수 있습니다.

<#FORMS> 웹 페이지 에디터에서 추가한 컴포넌트에서 만든 HTML을 생성합니다. 각 컴포넌트의 HTML은 *WebPageItems*에 나타나는 순서대로 만들어집니다.

<#SCRIPT> 웹 페이지 에디터에서 추가한 컴포넌트에서 만든 HTML에서 사용하는 javascript 선언 블록을 만듭니다.

*HTMLDoc*의 값을 변경하거나 *HTMLFile* 속성을 설정하여 디폴트 템플릿을 대체할 수 있습니다. 사용자 정의한 HTML 템플릿에는 디폴트 템플릿을 구성하는 모든 HTML 투명 태그가 포함될 수 있습니다. InternetExpress 페이지 프로듀서에서는 *Content* 메소드를 호출할 때 이 태그를 자동으로 변환합니다. 그리고 InternetExpress 페이지 프로듀서에서는 다음 세 가지 태그도 자동으로 변환합니다.

<#BODYELEMENTS>는 디폴트 템플릿의 다섯 개 태그 결과와 같은 HTML로 대체됩니다. 디폴트 레이아웃을 사용하고 싶지만 에디터를 사용하여 다른 요소를 추가할 경우 HTML 에디터로 템플릿을 만들 때 유용합니다.

<#COMPONENT Name=WebComponentName> 은 *WebComponentName* 이라는 컴포넌트에서 만든 HTML로 대체됩니다. 이 컴포넌트는 웹 페이지 에디터에서 추가한 컴포넌트 중의 하나이거나 *IWebContent* 인터페이스를 지원하고 InternetExpress 페이지 프로듀서와 같은 소유자를 갖고 있는 컴포넌트일 수 있습니다.

<#DATAPACKET XMLBroker=BrokerName> 은 *BrokerName* 에서 지정한 XML 브로커에서 가져온 XML 데이터 패킷과 대체됩니다. 웹 페이지 에디터로 InternetExpress 페이지 프로듀서에서 만든 HTML을 표시할 때 실제 XML 데이터 패킷 대신 이 태그가 표시됩니다.

더욱이, 사용자 정의된 템플릿에는 개발자가 정의한 다른 모든 HTML 투명 태그가 포함될 수 있습니다. InternetExpress 페이지 프로듀서는 자동으로 변환하는 7가지 타입에 속하지 않는 태그를 발견하면 *OnHTMLTag* 이벤트를 생성합니다. 개발자는 이 이벤트에서 원하는 변환을 수행할 코드를 작성할 수 있습니다. 일반적인 HTML 템플릿에 대한 자세한 내용은 33-14페이지의 "HTML 템플릿"을 참조하십시오.

팁 웹 페이지 에디터에 나타나는 컴포넌트에서 정적 코드를 생성합니다. 즉, 애플리케이션 서버에서 데이터 패킷에 나타나는 메타데이터를 변경하지 않을 경우 HTML이 생성된 시점과는 상관없이 HTML은 항상 똑같습니다. 웹 페이지 에디터에서 생성된 HTML을 복사하고 이를 템플릿으로 사용하여 모든 요청 메시지에 대한 응답으로 런타임 시 동적으로 이 코드를 생성하는 오버헤드를 피할 수 있습니다. 웹 페이지 에디터에서는 실제 XML 대신 <#DATAPACKET> 태그를 표시하기 때문에 이를 템플릿으로 사용하면 애플리케이션이 애플리케이션 서버에서 동적으로 데이터 패킷을 가져올 수 있습니다.

장

30

데이터베이스 애플리케이션에서 XML 사용

C++Builder는 데이터베이스 서버 연결을 지원하는 외에도 XML 문서가 데이터베이스 서버에 있는 것처럼 작업할 수 있게 해줍니다. XML(Extensible Markup Language)은 구조화된 데이터를 설명하는 표시 랭귀지입니다. XML 문서는 웹 애플리케이션, B2B 거래 등에 사용되는 전송 가능한 표준 데이터 형식을 제공합니다. C++Builder의 XML 문서 작업 지원에 대한 자세한 내용은 35장, "XML 문서 작업"을 참조하십시오.

데이터베이스 애플리케이션에서 C++Builder는 데이터 패킷(클라이언트 데이터셋의 *Data* 속성)을 XML 문서로 변환하고 XML 문서를 데이터 패킷으로 변환할 수 있는 컴포넌트 집합을 기초로 XML 문서 작업을 지원합니다. 이 컴포넌트를 사용하려면 먼저 XML 문서와 데이터 패킷 사이의 변환을 정의해야 합니다. 변환을 정의한 경우 특수 컴포넌트를 사용하여 다음을 수행할 수 있습니다.

- XML 문서를 데이터 패킷으로 변환합니다.
- XML 문서에서 데이터를 제공하고 XML 문서를 업데이트합니다.
- XML 문서를 프로바이더의 클라이언트로 사용합니다.

변환 정의

데이터 패킷과 XML 문서를 서로 변환하기 전에 데이터 패킷의 메타데이터와 해당 XML 문서의 노드 사이의 관계를 정의해야 합니다. 이 관계에 대한 설명은 변환이라는 특수 XML 문서에 저장됩니다.

각 변환 파일에는 XML 스키마의 노드와 데이터 패킷의 필드 간의 매핑 및 변환 결과의 구조를 나타내는 스켈레톤 XML 문서의 두 가지가 포함됩니다. 변환은 XML 스키마 또는 문서에서 데이터 패킷으로 매핑하거나 데이터 패킷의 메타데이터에서 XML 스키마로 매핑하는 단방향 매핑입니다. 경우에 따라 변환 파일을 XML에서 데이터 패킷으로 매핑하는 파일 하나와 데이터 패킷에서 XML로 매핑하는 파일 하나로 쌍으로 만들 수도 있습니다.

매핑할 변환 파일을 만들려면 bin 디렉토리에 있는 XMLMapper 유틸리티를 사용하십시오.

XML 노드와 데이터 패킷 필드 사이의 매핑

XML에서는 구성된 데이터를 텍스트 방식으로 저장하거나 설명하지만 데이터셋에서는 구성된 데이터를 다른 방식으로 저장하고 설명합니다. 따라서, XML 문서를 데이터셋으로 변환하려면 XML 문서의 노드와 데이터셋의 필드 사이의 일치를 식별해야 합니다.

예를 들어, 전자 메일 메시지 집합을 나타내는 XML 문서를 가정해 보십시오. 이 XML 문서는 다음과 같이 나타납니다(단일 메시지 포함).

```
<?xml version="1.0" standalone='yes' ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the new XML component support
in C++Builder.
        This looks like a good solution to our buisness-to-buisness
application!
        Also attached, please find the project schedule. Do you think its
reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

이 문서와 데이터셋 사이의 자연 매핑 중 하나는 각 전자 메일 메시지를 단일 레코드로 매핑하는 것입니다. 레코드에는 보낸 사람의 이름 및 주소 필드가 있습니다. 전자 메일 메시지에는 수신자가 여러 명 있을 수 있기 때문에 수신자(<to>)는 중첩 데이터셋에 매핑됩니다. 마찬가지로 cc 리스트도 중첩 데이터셋에 매핑됩니다. 제목 줄은 메시지 자체 (<content>) 가 메모 필드가 될 수 있는 문자열 필드에 매핑됩니다. 메시지 하나에 첨부 파일이 여러 개 있을 수 있기 때문에 첨부 파일의 이름은 중첩 데이터셋에 매핑됩니다. 따라서, 위의 전자 메일은 다음과 같이 나타나는 데이터셋에 매핑됩니다.

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(데이터셋)	(데이터셋)	XML 컴포넌트	(메모)	(데이터셋)

"To" 필드의 중첩 데이터셋 위치

Name	Address
Joe Engineer	jengineer@MyCo.Com

"CC" 필드의 중첩 데이터셋

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

"Attach" 필드의 중첩 데이터셋

Attachfile
XMLSpec.txt
Schedule.txt

그런 매핑 정의에는 반복될 수 있는 XML 문서의 노드를 식별하여 중첩 데이터셋에 매핑하는 작업이 포함됩니다. 값을 가지고 있고 한 번만 표시되는 태크로 지정된 요소(예: <content>...</content>)들은 데이터 타입이 값으로 표시될 수 있는 데이터의 타입을 반영하는 필드에 매핑됩니다. 첨부 태그의 AttachFile 어트리뷰트(attribute)처럼 태그의 어트리뷰트 역시 필드에 매핑됩니다.

XML 문서에 있는 모든 태그가 해당 데이터셋에 표시되는 것은 아닙니다. 예를 들어, <head>...<head/> 요소는 결과 데이터셋에 일치하는 요소가 없습니다. 일반적으로 값이 있는 요소, 반복될 수 있는 요소, 태그의 어트리뷰트 등만 데이터셋 필드(중첩 데이터셋 필드 포함)에 매핑됩니다. XML 문서에 있는 부모 노드가 필드 값이 하위 노드의 값으로부터 만들어진 필드에 매핑되는 경우에는 이 규칙이 적용되지 않습니다. 예를 들어, XML 문서에 다음과 같은 태그 집합이 들어 있을 수 있습니다.

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
</FullName>
```

이러한 태그 집합은 다음과 같은 값을 지닌 단일 데이터셋 필드에 매핑될 수 있습니다.

```
Mr. John Smith
```

XMLMapper 사용

XML 매퍼 유틸리티인 xmlmapper.exe를 사용하여 다음 세 가지 방법으로 매핑을 정의할 수 있습니다.

- 기존 XML 스키마 또는 문서에서 사용자가 정의하는 클라이언트 데이터셋으로. 이 매핑은 XML 스키마가 이미 존재하는 데이터로 작업하기 위해 데이터베이스 애플리케이션을 만들 경우에 유용합니다.
- 기존 데이터 패킷에서 사용자가 정의하는 새 XML 스키마로. 이 매핑은 예를 들어 새 기업 간 통신 시스템을 만들기 위해 기존 데이터베이스 정보를 XML로 노출하고자 할 경우에 유용합니다.
- 기존 XML 스키마와 기존 데이터 패킷 사이. 이 매핑은 모두 같은 정보를 설명하고 함께 작업되기를 바라는 데이터베이스와 XML 스키마가 있을 경우에 유용합니다.

매핑을 정의한 경우 XML 문서를 데이터 패킷으로, 데이터 패킷을 XML 문서로 서로 변환하는 데 사용되는 변환 파일을 생성할 수 있습니다. 변환 파일은 방향성이 있기 때문에 단일 매핑을 사용하여 XML을 데이터 패킷으로 변환하고 데이터 패킷을 XML로 변환하는 두 가지 변환을 모두 생성할 수 있습니다.

참고 XML 매퍼는 midas.dll과 msxml.dll의 두 .DLL 파일을 사용하여 올바르게 작동합니다. xmlmapper.exe를 사용하려면 이 두 .DLL을 먼저 설치해야 합니다. 또한 msxml.dll을 COM 서버로 등록해야 합니다. Regsvr32.exe를 사용하여 이 파일을 등록할 수 있습니다.

XML 스키마 또는 데이터 패킷 로드

매핑을 정의하고 변환 파일을 생성하기 전에 먼저 매핑할 XML 문서와 데이터 패킷에 대한 설명을 로드해야 합니다.

XML 문서 또는 스키마를 로드하려면 File|Open 을 선택한 후 나타나는 다이얼로그 박스에서 해당 문서 또는 스키마를 선택합니다.

데이터 패킷을 로드하려면 File|Open을 선택한 후 나타나는 다이얼로그 박스에서 해당 데이터 패킷 파일을 선택합니다. 데이터 패킷은 클라이언트 데이터셋의 *SaveToFile* 메소드를 호출할 때 단순히 생성되는 파일입니다. 데이터 패킷을 디스크에 저장하지 않은 경우 Datapacket 창의 내부를 마우스 오른쪽 버튼으로 클릭하고 Connect To Remote Server를 선택하여 멀티 티어 애플리케이션의 애플리케이션 서버에서 해당 데이터 패킷을 직접 가져올 수 있습니다.

XML 문서 또는 스키마, 데이터 패킷을 하나만 로드하거나 모두 로드할 수 있습니다. 사용자가 매핑의 한 쪽만 로드하면 XML 매퍼가 다른 쪽에 대한 자연 매핑을 생성할 수 있습니다.

매핑 정의

XML 문서와 데이터 패킷 간의 매핑에 데이터 패킷의 필드 또는 XML 문서의 태그로 지정된 요소들이 모두 포함될 필요는 없습니다. 따라서, 매핑되는 요소를 먼저 지정해야 합니다. 이 요소들을 지정하려면 먼저 다이얼로그 박스의 가운데 창에서 Mapping 페이지를 선택하십시오.

데이터 패킷의 필드에 매핑되는 XML 문서 또는 스키마의 요소를 지정하려면 XML 문서 창에서 Sample 또는 Structure 탭을 선택하고 데이터 패킷 필드에 매핑되는 요소에 대한 노드를 더블 클릭하십시오.

XML 문서의 어트리뷰트 또는 태그 요소에 매핑되는 데이터 패킷 필드를 지정하려면 Datapacket 창에서 해당 필드에 대한 노드를 더블 클릭하십시오.

매핑의 한 쪽만(XML 문서 또는 데이터 패킷) 로드한 경우 매핑되는 노드를 선택한 다음 다른 쪽을 생성할 수 있습니다.

- XML 문서에서 데이터 패킷을 생성할 경우 먼저 데이터 패킷에서 일치하는 필드 타입을 결정하는 선택된 노드에 대한 어트리뷰트(attribute)를 정의합니다. 가운데 창에서 Node Repository 페이지를 선택합니다. 매핑에 관여하는 각 노드를 선택하고 해당 필드의 어트리뷰트를 지정합니다. 매핑이 단순하지 않을 경우(예를 들어, 노드에 하위 노드가 있고 필드 값이 이러한 하위 노드로부터 만들어진 필드에 노드가 대응하는 경우) User Defined Translation 체크 박스를 선택하십시오. 이벤트 핸들러를 작성하여 나중에 사용자 정의 노드를 변환해야 합니다.

  노드가 매핑되는 방향을 지정했으면 Create|Datapacket from XML을 선택합니다. 해당 데이터 패킷이 자동으로 생성되어 Datapacket 창에 표시됩니다.

- 데이터 패킷에서 XML 문서를 생성할 경우 Create|XML from Datapacket을 선택합니다. 데이터 패킷의 필드, 레코드 및 데이터셋과 일치하는 XML 문서의 어트리뷰트 및 태그의 이름을 지정할 수 있는 다이얼로그 박스가 나타납니다. 필드 값이 사용자가 이름을 지정한 방법으로 어트리뷰트에 매핑되는지 값이 있는 태그 요소에 매핑되는지 여부를 지정합니다. @ 기호로 시작되는 이름은 레코드에 일치하는 태그의 어트리뷰트에 매핑되는 반면 @ 기호로 시작되지 않는 이름은 값이 있고 레코드의 요소 내에서 중첩되는 태그 요소에 매핑됩니다.

- XML 문서와 데이터 패킷(클라이언트 데이터셋 파일)을 모두 로드한 경우 일치하는 노드를 동일한 순서로 선택해야 합니다. 일치하는 노드들은 Mapping 페이지 위쪽의 테이블에 서로 이웃하게 표시되어야 합니다.

XML 문서와 데이터 패킷을 모두 로드하거나 생성하고 매핑 시에 나타나는 노드를 선택하면 Mapping 페이지의 위쪽에 있는 테이블에 사용자가 정의한 매핑이 반영되어야 합니다.

변환 파일 생성

변환 파일을 생성하려면 다음 단계를 따르십시오.

1. 먼저 다음과 같은 방법으로 변환이 만드는 내용을 나타내는 라디오 버튼을 선택합니다.
   - 데이터 패킷을 XML 문서로 매핑할 경우 Datapacket to XML 버튼을 선택합니다.
   - XML 문서를 데이터 패킷으로 매핑할 경우 XML to Datapacket 버튼을 선택합니다.
2. 데이터 패킷을 생성할 경우 Create Datapacket As 섹션에서 라디오 버튼을 사용할 수 있습니다. 이 버튼을 클릭하여 다음 중에서 데이터 패킷이 사용되는 방법을 지정할 수 있습니다(데이터셋으로, 업데이트 적용을 위한 델타 패킷으로, 데이터를 가져오기 전에 프로바이더에게 공급할 매개변수로).
3. Create 및 Test Transformation을 클릭하여 변환의 인메모리 버전을 생성합니다. XML 매퍼는 데이터 패킷에 대해 생성되는 XML 문서를 Datapacket 창에 표시하거나 XML 문서에 대해 생성되는 데이터 패킷을 XML Document 창에 표시합니다.
4. 마지막으로 File|Save|Transformation을 선택하여 변환 파일을 저장합니다. 변환 파일은 사용자가 정의한 변환을 설명하고 확장명이 .xtr인 특수 XML 파일입니다.

XML 문서를 데이터 패킷으로 변환

XML 문서를 데이터 패킷으로 변환하는 방법을 나타내는 변환 파일을 만든 경우 변환에 사용되는 스키마와 일치하는 XML 문서에 대한 데이터 패킷을 만들 수 있습니다. 그런 다음 이 데이터 패킷을 클라이언트 데이터셋에 할당하고 파일로 저장하여 파일 기반 데이터베이스 애플리케이션의 기본으로 사용할 수 있습니다.

TXMLTransform 컴포넌트는 변환 파일에 있는 매핑에 따라 XML 문서를 데이터 패킷으로 변환합니다.

참고 또한 *TXMLTransform*을 사용하여 XML 형식으로 표시되는 데이터 패킷을 임의의 XML 문서로 변환할 수도 있습니다.

소스 XML 문서 지정

다음과 같은 세 가지 방법으로 소스 XML 문서를 지정할 수 있습니다.

- 소스 문서가 디스크 상의 .xml 파일일 경우 *SourceXmlFile* 속성을 사용할 수 있습니다.
- 소스 문서가 메모리 내의 XML 문자열일 경우 *SourceXml* 속성을 사용할 수 있습니다.
- 소스 문서에 대한 IDOMDocument 인터페이스가 있을 경우 *SourceXmlDocument* 속성을 사용할 수 있습니다.

*TXMLTransform*은 위에 나열된 순서대로 이 속성들을 확인합니다. 즉, *SourceXmlFile* 속성에서 파일 이름을 먼저 확인하고 *SourceXmlFile*이 빈 문자열인 경우에만 *SourceXml* 속성을 확인합니다. 또한 *SourceXml*이 빈 문자열인 경우에만 *SourceXmlDocument* 속성을 확인합니다.

변환 지정

다음과 같은 두 가지 방식으로 XML 문서를 데이터 패킷으로 변환하는 변환을 지정할 수 있습니다.

- *TransformationFile* 속성을 설정하여 xmlmapper.exe를 사용하여 만든 변환 파일을 나타냅니다.
- 변환을 위한 *IDOMDocument* 인터페이스가 있을 경우 *TransformationDocument* 속성을 설정합니다.

*TXMLTransform*은 이들 속성을 위에 나열된 순서대로 확인합니다. 즉, 먼저 *TransformationFile* 속성에서 파일 이름을 확인하고 *TransformationFile*이 빈 문자열인 경우에만 *TransformationDocument* 속성을 확인합니다.

결과 데이터 패킷 가져오기

*TXMLTransform*이 변환을 수행하고 데이터 패킷을 생성하게 하려면 *Data* 속성만 읽어야 합니다. 예를 들어, 다음 코드는 XML 문서와 변환 파일을 사용하여 데이터 패킷을 생성한 다음 클라이언트 데이터셋에 할당합니다.

```
XMLTransform1->SourceXMLFile = "CustomerDocument.xml";
XMLTransform1->TransformationFile = "CustXMLToCustTable.xtr";
ClientDataSet1->XMLData = XMLTransform1->Data;
```

사용자 정의 노드 변환

xmlmapper.exe를 사용하여 변환을 정의할 경우 XML 문서에 있는 일부 노드가 "사용자 정의" 노드인지 지정할 수 있습니다. 사용자 정의 노드는 노드 값을 필드 값으로 직접 변환하지 않고 코드로 변환하는 노드입니다.

OnTranslate 이벤트를 이용하면 사용자 정의 노드를 변환할 코드를 제공할 수 있습니다. *OnTranslate*은 *TXMLTransform* 컴포넌트가 XML 문서에서 사용자 정의 노드를 발견할 때마다 호출됩니다. OnTranslate 이벤트 핸들러에서 소스 문서를 읽고 데이터 패킷의 필드에 대한 결과 값을 지정합니다.

예를 들어, 다음 *OnTranslate* 이벤트 핸들러는 다음과 같은 형태로 된 XML 문서의 노드를

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

단일 필드 값으로 변환합니다.

```
void __fastcall TForm1::XMLTransform1Translate(TObject *Sender, AnsiString
Id,
  _di_IDOMNode SrcNode, AnsiString &Value, _di_IDOMNode DestNode)
{
  if (Id == "FullName")
  {
    Value = "";
    if (SrcNode.hasChildNodes)
    {
      _di_IXMLDOMNode CurNode = SrcNode.firstChild;
      Value = SrcValue + AnsiString(CurNode.nodeValue);
      while (CurNode != SrcNode.lastChild)
      {
        CurNode = CurNode.nextSibling;
        Value = Value + AnsiString(" ");
        Value = Value + AnsiString(CurNode.nodeValue);
      }
    }
  }
}
```

XML 문서를 프로바이더의 소스로 사용

TXMLTransformProvider 컴포넌트를 통해 XML 문서를 마치 데이터베이스 테이블처럼 사용할 수 있습니다. *TXMLTransformProvider*는 XML 문서에서 데이터를 패키지화하여 업데이트를 클라이언트에서 해당 XML 문서로 다시 적용합니다. 또한 다른 프로바이더 컴포넌트처럼 클라이언트 데이터셋 또는 XML 브로커와 같은 클라이언트에게 표시됩니다. 프로바이더 컴포넌트에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오. 클라이언트 데이터셋에서 프로바이더 컴포넌트를 사용하는 것에 대한 자세한 내용은 27-24페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

XML 프로바이더가 제공할 데이터를 가져오고 *XMLDataFile* 속성을 사용하여 업데이트를 적용하는 XML 문서를 지정할 수 있습니다.

TXMLTransformProvider 컴포넌트는 내부 *TXMLTransform* 컴포넌트를 사용하여 데이터 패킷과 소스 XML 문서 사이에서 XML 문서를 데이터 패킷으로 변환하거나 업데이트를 적용한 후 데

이터 패킷을 소스 문서의 XML 형식으로 변환합니다. 이 두 *TXMLTransform* 컴포넌트는 각각 *TransformRead* 속성과 *TransformWrite* 속성을 사용하여 액세스할 수 있습니다.

*TXMLTransformProvider*를 사용할 경우 이 두 *TXMLTransform* 컴포넌트가 데이터 패킷과 소스 XML 문서 사이에서 변환하는 데 사용할 변환을 지정해야 합니다. 독립 실행형 *TXMLTransform* 컴포넌트를 사용할 때처럼 *TXMLTransform* 컴포넌트의 *TransformationFile* 또는 *TransformationDocument* 속성을 설정하여 이 작업을 수행합니다.

또한 변환에 사용자 정의 노드가 포함될 경우 내부 *TXMLTransform* 컴포넌트에 *OnTranslate* 이벤트 핸들러를 제공해야 합니다.

TXMLTransform 컴포넌트에 *TransformRead* 및 *TransformWrite*의 값인 소스 문서를 지정할 필요는 없습니다. *TransformRead*의 경우 소스는 프로바이더의 *XMLDataFile* 속성에 의해 지정되는 파일입니다. 그렇지만 *XMLDataFile*을 빈 문자열로 설정할 경우 *TransformRead->XmlSource* 또는 *TransformRead->XmlSourceDocument*를 사용하여 소스 문서를 제공할 수 있습니다. *TransformWrite*의 경우 업데이트가 적용되면 프로바이더에 의해 소스가 내부적으로 생성됩니다.

XML 문서를 프로바이더의 클라이언트로 사용

TXMLTransformClient 컴포넌트는 XML 문서 또는 문서 집합을 애플리케이션 서버의 클라이언트로 사용할 수 있게 해주거나, 단순히 *TDataSetProvider* 컴포넌트를 통해 연결할 데이터셋의 클라이언트로 사용할 수 있게 해주는 어댑터 역할을 합니다. 즉, *TXMLTransform* 클라이언트를 사용하여 데이터베이스 데이터를 XML 문서로 게시할 수 있으며, XML 문서 형식으로 데이터를 제공하는 외부 애플리케이션으로부터 업데이트 요청(삽입 또는 삭제)을 이용할 수 있습니다.

TXMLTransformClient 객체가 데이터를 가져올 프로바이더를 지정하려면 *ProviderName* 속성을 설정하십시오. 클라이언트 데이터셋의 *ProviderName* 속성에서처럼 *ProviderName*은 원격 애플리케이션 서버에 있는 프로바이더의 이름이거나 *TXMLTransformClient* 객체와 동일한 폼 또는 데이터 모듈에 있는 로컬 프로바이더일 수 있습니다. 프로바이더에 대한 자세한 내용은 28장, "프로바이더 컴포넌트 사용"을 참조하십시오.

프로바이더가 원격 애플리케이션 서버에 있을 경우 DataSnap 연결 컴포넌트를 사용하여 해당 애플리케이션 서버에 연결해야 합니다. *RemoteServer* 속성을 사용하여 연결 컴포넌트를 지정합니다. DataSnap 연결 컴포넌트에 대한 자세한 내용은 29-22페이지의 "애플리케이션 서버에 연결"을 참조하십시오.

프로바이더로부터 XML 문서 가져오기

*TXMLTransformClient*는 내부 *TXMLTransform* 컴포넌트를 사용하여 프로바이더로부터 가져온 데이터 패킷을 XML 문서로 변환합니다. 이 TXMLTransform 컴포넌트를 *TransformGetData* 속성의 값으로 액세스할 수 있습니다.

프로바이더로부터 가져온 데이터를 표시하는 XML 문서를 만들기 전에, 데이터 패킷을 해당 XML 형식으로 변환할 때 *TransformGetData*가 사용할 변환 파일을 지정해야 합니다. 독립형 *TXMLTransform* 컴포넌트를 사용할 때처럼 *TXMLTransform* 컴포넌트의 *TransformationFile* 또는 *TransformationDocument* 속성을 설정하여 이 작업을 수행합니다. 이 변환에 사용자 정의 노드가 포함될 경우 *TransformGetData*에 *OnTranslate* 이벤트 핸들러를 함께 제공합니다.

*TXMLTransformClient*가 프로바이더로부터 소스 문서를 가져오기 때문에 *TransformGetData*에 대한 소스 문서를 지정할 필요가 없습니다. 그러나, 프로바이더가 입력 매개변수를 원할 경우 데이터를 가져오기 전에 해당 매개변수를 설정할 수 있습니다. *SetParams* 메소드를 사용하여 프로바이더로부터 데이터를 가져오기 전에 입력 매개변수를 제공합니다. *SetParams*는 매개변수 값을 추출해 올 XML 문자열과 해당 XML을 데이터 패킷으로 변환할 변환 파일 이름의 두 요소를 갖습니다. *SetParams*는 변환 파일을 사용하여 XML 문자열을 데이터 패킷으로 변환한 다음 해당 패킷으로부터 매개변수 값을 추출합니다.

참고 매개변수 문서 또는 변환을 다른 방법으로 지정하려면 이 요소들 중 어느 것을 오버라이드할 수 있습니다. *TransformSetParams* 속성에 있는 속성 중 하나를 설정하여 변환할 때 사용할 매개변수나 변환이 포함된 문서를 가리킨 다음 *SetParams*가 호출될 때 빈 문자열을 오버라이드할 인수를 설정합니다. 사용 가능한 속성에 대한 자세한 내용은 30-6페이지의 "XML 문서를 데이터 패킷으로 변환"을 참조하십시오.

TransformGetData 를 구성하고 입력 매개변수를 제공했으면 *GetDataAsXml* 메소드를 호출하여 XML을 가져올 수 있습니다. *GetDataAsXml*는 프로바이더에게 현재 매개변수 값을 보내고 데이터 패킷을 가져와서 XML 문서로 변환한 다음 해당 문서를 문자열로 반환합니다. 이 문자열을 파일로 저장할 수 있습니다.

```
XMLTransformClient1->ProviderName = "Provider1";
XMLTransformClient1->TransformGetData->TransformationFile =
"CustTableToCustXML.xtr";
XMLTransformClient1->TransFormSetParams->SourceXmlFile =
"InputParams.xml";
XMLTransformClient1->SetParams("", "InputParamsToDP.xtr");
AnsiString XML = XMLTransformClient1->GetDataAsXml();
TFileStream pXMLDoc = new TFileStream("Customers.xml", fmCreate ||
fmOpenWrite);
__try
{
  pXMLDoc->Write(XML.c_str(), XML.Length());
}
__finally
{
  delete pXMLDoc;
}
```

XML 문서의 업데이트 내용을 프로바이더에 적용

*TXMLTransformClient*를 사용하면 XML 문서의 모든 데이터를 프로바이더의 데이터셋에 삽입하거나, XML 문서의 모든 레코드를 프로바이더의 데이터셋에서 삭제할 수 있습니다. 업데이트를 수행하려면 *ApplyUpdates* 메소드를 호출하여 다음 사항을 전달하십시오.

- 값이 삽입 또는 삭제할 데이터가 있는 XML 문서의 컨텐트인 문자열
- 해당 XML 데이터를 삽입 또는 삭제 델타 패킷으로 변환할 수 있는 변환 파일의 이름. XML 매퍼 유틸리티를 사용하여 변환 파일을 정의할 경우 변환이 삽입 델타 패킷을 위한 것인지 삭제 델타 패킷을 위한 것인지를 지정합니다.
- 업데이트 작업에 허용될 수 있는 업데이트 오류의 수. 지정된 수보다 적은 레코드를 삽입하거나 삭제할 수 없을 경우 *ApplyUpdates*는 실제 오류의 수를 반환합니다. 지정된 수보다 많은 레코드를 삽입하거나 삭제할 수 없을 경우 전체 업데이트 작업이 롤백되고 업데이트가 수행되지 않습니다.

다음 호출은 XML 문서 Customers.xml을 델타 패팃으로 변환하고 오류 수에 관계 없이 모든 업데이트를 적용합니다.

```
StringList1->LoadFromFile("Customers.xml");
nErrors = ApplyUpdates(StringList1->Text, "CustXMLToInsert.xtr", -1);
```

부

III

인터넷 애플리케이션 작성

"인터넷 애플리케이션 작성"에 포함된 장에서는 인터넷으로 배포되는 애플리케이션을 작성하는 데 필요한 개념과 기술을 설명합니다.

참고 이 단원에서 설명하는 컴포넌트들을 모든 C++Builder 에디션에서 사용할 수 있는 것은 아닙니다.

장

31

CORBA 애플리케이션 작성

CORBA(Common Object Request Broker Architecture)는 복잡한 분산 객체 애플리케이션 개발 과정을 단순화하기 위해 OMG(Object Management Group)에서 채택한 표준입니다.

이름에서 알 수 있듯이 CORBA는 분산 애플리케이션을 개발하는 데 있어 객체 지향적인 접근 방법을 제공합니다. 이 방법은 32장, "인터넷 서버 애플리케이션 생성"에서 HTTP 애플리케이션에 대해 설명되는 메시지 지향 접근 방법과 대조됩니다. CORBA에서 서버 애플리케이션은 잘 정의된 인터페이스를 통해 클라이언트 애플리케이션이 원격으로 사용할 수 있는 객체를 구현합니다.

참고 COM은 분산 애플리케이션에 대한 또 다른 객체 지향적인 접근 방법을 제공합니다. COM에 대한 자세한 내용은 38장, "COM 기술 개요"를 참조하십시오. 그러나 CORBA는 COM과 달리 Windows 이외의 플랫폼에 적용되는 표준입니다. 이것은 C++Builder로, 다른 플랫폼에서 실행되는 CORBA 사용 가능 애플리케이션과 통신할 수 있는 CORBA 클라이언트 또는 서버를 작성할 수 있다는 것을 의미합니다.

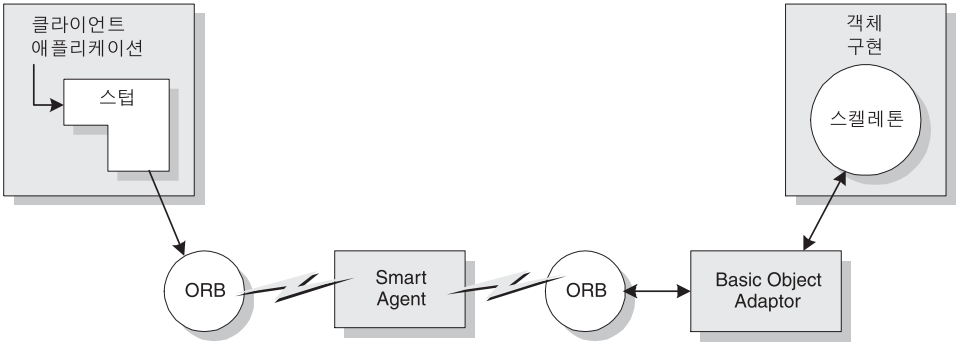
CORBA 표준은 클라이언트 애플리케이션이 서버에서 구현되는 객체와 통신하는 방법을 정의합니다. 이 통신은 ORB(Object Request Broker)에 의해 처리됩니다. C++Builder는 Borland의 VisiBroker ORB(버전 4.5)와 밀접하게 통합되어 있으므로 CORBA 개발 작업을 보다 쉽게 수행할 수 있습니다.

클라이언트가 서버 시스템의 객체와 통신할 수 있게 해주는 기본 ORB 기술 외에도, CORBA 표준은 많은 표준 서비스를 정의합니다. 이러한 서비스는 잘 정의된 인터페이스를 사용하기 때문에 다른 업체에서 서버를 개발한 경우에도 이러한 서비스를 사용하는 클라이언트를 개발할 수 있습니다.

CORBA 애플리케이션 개요

객체 지향 프로그래밍을 수행하는 도중에 CORBA는 개발자가 원격 객체를 마치 로컬 객체인 것처럼 사용할 수 있게 하므로 분산 애플리케이션을 쉽게 작성할 수 있습니다. 이것은 객체가 다른 시스템에 존재할 경우의 네트워크 통신 처리를 위해 추가 레이어가 포함되었다는 점만 제외하면 CORBA 애플리케이션의 디자인이 다른 객체 지향 애플리케이션과 동일하기 때문입니다. 이 추가 레이어는 스텁 및 스켈레톤이라고 부르는 특수한 객체에 의해 처리됩니다.

그림 31.1 CORBA 애플리케이션 구조



CORBA 클라이언트에서 스텁은 동일 프로세스, 다른 프로세스 또는 다른 서버 시스템에 의해 구현될 객체의 대리자로서 작동합니다. 클라이언트는 다른 객체와 상호 작용하는 것처럼 스텁과 상호 작용합니다.

그러나 대부분의 다른 객체들과 달리 스텁은 클라이언트 시스템에 설치된 ORB 소프트웨어를 호출하여 인터페이스 호출을 처리합니다. VisiBroker ORB는 LAN 상의 임의 위치에서 실행 중인 Smart Agent(osagent)를 사용합니다. 이 Smart Agent는 실제 객체 구현을 제공하는, 사용 가능한 서버를 찾는 동적인 분산 디렉토리 서비스입니다.

CORBA 서버사이드의 ORB 소프트웨어는 인터페이스 호출을 자동 생성된 스켈레톤에 전달합니다. 이 스켈레톤은 BOA(Basic Object Adaptor)를 통해 ORB 소프트웨어와 통신합니다. 이 스켈레톤은 BOA를 사용하여 객체를 Smart Agent에 등록하고, 객체의 범위(객체가 원격 시스템에서 사용될 수 있는지 여부)를 나타내고, 객체가 인스턴스화되어 클라이언트에 응답할 준비가 된 시점을 나타냅니다.

스텁 및 스켈레톤 이해

CORBA에서 분산 객체의 기본은 인터페이스입니다. 인터페이스는 구현 정보가 포함되지 않았다는 점을 제외하면 클래스 정의와 동일합니다. 인터페이스는 CORBA IDL(Interface Definition Language)을 사용하여 정의합니다. 그런 다음 인터페이스 정의를 프로젝트에 IDL 파일로 추가할 수 있습니다. IDL 파일, 스텁 및 스켈레톤에 대한 자세한 내용은 *VisiBroker Programmer's Guide*를 참조하십시오. C++Builder에서 IDL 파일을 사용하는 것에 대한 자세한 내용은 31-5페이지의 "객체 인터페이스 정의"를 참조하십시오.

IDL 파일을 컴파일할 때, C++Builder는 .cpp 파일 두 개를 생성합니다. 이 파일 중 하나는 스텁 클래스의 구현을 포함하고 있는 클라이언트 파일이고, 다른 하나는 스켈레톤 클래스의 구현을 포함하는 서버 파일입니다. 스텁과 스켈레톤은 CORBA 애플리케이션에서 인터페이스 호출을 마샬링할 수 있는 메커니즘을 제공합니다. 마샬링은 다음을 수행합니다.

- 서버 프로세스에서 객체 인스턴스를 가져와 클라이언트 프로세스에서 이 인스턴스를 코드에 사용할 수 있게 합니다.
- 인터페이스 호출의 인수를 클라이언트에서 전달된 것처럼 전송하고 원격 객체의 프로세스 공간에 둡니다.

CORBA 객체의 메소드를 호출할 때, 클라이언트 애플리케이션은 인수를 스택에 푸시하고 스텁 객체를 호출합니다. 스텁은 마샬링 버퍼에서 인수를 작성하고 구조화된 호출을 원격 객체에 전송합니다. 서버 스켈레톤은 이 구조의 압축을 풀어 인수를 스택에 푸시하고 객체의 구현을 호출합니다. 본질적으로 이 스켈레톤은 자신의 주소 공간에서 클라이언트의 호출을 다시 만듭니다.

Smart Agent 사용

Smart Agent(osagent)는 객체를 구현하는 사용 가능한 서버를 찾는 동적인 분산 디렉토리 서비스입니다. 선택할 수 있는 서버가 여러 개인 경우, Smart Agent는 로드 밸런싱을 제공합니다. 또한 연결 실패 시 서버를 다시 시작하거나, 필요한 경우 다른 호스트에 있는 서버를 찾는 방법으로 서버 실패를 방지합니다.

로컬 네트워크에 있는 최소한 한 개 이상의 호스트에서 Smart Agent를 시작해야 합니다. 여기서 로컬 네트워크는 그 안에서 브로드캐스트 메시지를 보낼 수 있는 네트워크를 말합니다. ORB는 브로드캐스트 메시지를 사용하여 Smart Agent를 찾습니다. 네트워크에 여러 개의 Smart Agent가 있는 경우, ORB는 처음 응답하는 Smart Agent를 사용합니다. Smart Agent를 찾은 후, ORB는 지점간 UDP 프로토콜을 사용하여 Smart Agent와 통신합니다. UDP 프로토콜은 TCP 연결보다 적은 네트워크 리소스를 사용합니다.

네트워크에 여러 Smart Agent가 있는 경우, 각 Smart Agent는 사용할 수 있는 객체의 부분 집합을 인식하고 다른 Smart Agent와 통신하여 직접 인식할 수 없는 객체를 찾습니다. 특정 Smart Agent가 예기치 않게 종료되면 이 Smart Agent가 인식하고 있던 객체는 사용 가능한 다른 Smart Agent에 자동으로 등록됩니다.

로컬 네트워크에서 Smart Agent 를 구성하고 사용하는 방법에 대한 자세한 내용은 *VisiBroker Installation and Administration Guide*를 참조하십시오.

서버 애플리케이션 활성화

서버 애플리케이션은 BOA를 통해 시작과 함께 클라이언트 호출을 승인할 수 있는 객체를 ORB에 알립니다. ORB를 초기화하고 서버가 준비되었다는 사실을 ORB에 알리는 이 코드는 CORBA 서버 애플리케이션을 시작하는 데 사용하는 마법사에 의해 자동으로 애플리케이션에 추가됩니다.

일반적으로 CORBA 서버 애플리케이션은 수동으로 시작됩니다. 그러나 OAD(Object Activation Daemon)를 사용하여 서버를 시작하거나 클라이언트에서 서버의 객체를 사용해야 하는 경우에 한하여 이러한 객체를 인스턴스화할 수 있습니다.

OAD를 사용하려면 객체를 OAD에 등록해야 합니다. 객체를 OAD에 등록하면 Implementation Repository라 불리는 데이터베이스에서 객체를 구현하는 서버 애플리케이션과 객체 간의 연결이 저장됩니다.

Implementation Repository에 객체에 대한 엔트리가 존재하면 OAD는 애플리케이션을 ORB로 시뮬레이트합니다. 클라이언트가 객체를 요청하면 ORB는 마치 서버 애플리케이션인 것처럼 OAD에 연결됩니다. 그런 다음 OAD는 필요한 경우 애플리케이션을 시작하여 클라이언트 요청을 실제 서버에 전달합니다.

객체를 OAD에 등록하는 방법에 대한 자세한 내용은 *VisiBroker Programmer's Guide*를 참조하십시오.

인터페이스 호출의 동적 연결

일반적으로 CORBA 클라이언트는 서버에서 객체의 인터페이스를 호출할 때 정적 연결을 사용합니다. 이 접근 방법은 성능 향상 및 컴파일 타임 타입 검사를 비롯한 많은 장점을 갖고 있습니다. 그러나 사용할 인터페이스를 런타임이 되기 전까지 확인하지 못할 수도 있습니다. 이러한 경우, C++Builder를 사용하여 인터페이스를 런타임에 동적으로 연결할 수 있습니다.

동적 연결을 사용할 때는 인터페이스를 Interface Repository에 등록하는 것이 도움이 됩니다.

CORBA 클라이언트 애플리케이션에서 동적 연결을 사용하는 방법에 대한 자세한 내용은 *VisiBroker Programmer's Guide*에서 DII(Dynamic Invocation Interface)에 대한 정보를 참조하십시오.

CORBA 서버 작성

C++Builder에는 CORBA 서버 개발 과정을 용이하게 하는 여러 마법사가 포함되어 있습니다. 다음 단계는 C++Builder를 사용하여 CORBA 서버를 만드는 방법에 대해 설명합니다.

1. 객체 인터페이스를 정의합니다. 이러한 인터페이스는 클라이언트 애플리케이션이 서버와 상호 작용하는 방법을 정의합니다. 또한 C++Builder는 이러한 인터페이스에 기초하여 스텁 및 스켈레톤 구현을 만듭니다.
2. CORBA 서버 마법사를 사용하여 시작 시 CORBA BOA 및 ORB를 초기화하는 코드가 포함된 새 CORBA 서버 애플리케이션을 만듭니다.
3. 인터페이스 정의가 포함된 IDL 파일을 스켈레톤 클래스 및 스텁 클래스로 컴파일합니다.
4. CORBA Object 마법사를 사용하여 구현 클래스를 정의 및 인스턴스화합니다.
5. 단계 4에서 만든 클래스를 완성하여 CORBA 객체를 구현합니다.
6. 필요한 경우 CORBA 인터페이스를 변경하고 변경 내용을 구현 클래스에 포함시킵니다.

위에 나열된 단계를 수행하는 것 이외에도, IDL 파일을 Interface Repository 및 Object Activation Daemon에 등록할 수도 있습니다.

객체 인터페이스 정의

CORBA 객체 인터페이스는 CORBA IDL(Interface Definition Language)을 사용하여 정의합니다. IDL이 C++과 유사한 구문을 갖기 때문에 IDL 파일은 C++ 헤더 파일과 그 모양이 비슷합니다. 또한 IDL 파일은 헤더 파일과 비슷하게 작동합니다. 즉, 헤더 파일이 공유할 수 있는 클래스를 선언하는 것처럼 IDL 파일은 공유할 수 있는 인터페이스를 선언합니다. 그러나 CORBA에서 인터페이스(클래스)는 동일한 애플리케이션에 있는 다른 모듈뿐 아니라 다른 애플리케이션과도 공유할 수 있습니다.

참고 IDL이라는 용어는 유사한 인터페이스 정의 언어를 나타내는 다른 의미로 사용됩니다. 즉, OMG에서 정의된 CORBA IDL, COM에 사용되는 Microsoft IDL, DCE IDL 등이 존재합니다. CORBA IDL에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

C++Builder 내에서 새 IDL 파일을 정의하려면 File|New|Other를 선택하고 New Items 다이얼로그 박스의 Multitier 페이지에서 CORBA IDL File을 선택하십시오. 그러면 비어 있는 IDL 파일이 표시된 코드 에디터가 열리고 현재 프로젝트에 이 IDL 파일이 추가됩니다.

객체를 정의하는 IDL 파일이 이미 있으면, Project|Add to Project를 선택하고 IDL 파일을 파일 타입으로 선택한 다음 기존 IDL 파일을 선택하여 IDL 파일을 간단하게 프로젝트에 추가할 수 있습니다.

CORBA 서버 마법사 사용

새 CORBA 서버 프로젝트를 시작하려면 File|New|Other를 선택하고 New Items 다이얼로그 박스의 Multitier 페이지에서 CORBA 서버 아이콘을 선택하십시오. CORBA 서버 마법사를 사용하면 콘솔 애플리케이션을 만들 것인지, 아니면 Windows 애플리케이션을 만들 것인지를 지정할 수 있습니다.

콘솔 애플리케이션을 만드는 중이면 서버에서 VCL 클래스를 사용할 것인지 여부를 지정할 수 있습니다. VCL 체크 박스를 선택하지 않으면 생성된 모든 코드를 다른 플랫폼으로 이식할 수 있습니다.

애플리케이션 타입을 선택하는 것 외에도 모든 기존 IDL 파일을 프로젝트에 포함하거나, 비어 있는 새 IDL 파일을 포함하도록 지정할 수 있습니다. 결국 서버 프로젝트는 클라이언트가 서버와 통신하기 위해 사용하는 인터페이스를 정의하는 하나 이상의 IDL 파일을 포함해야 합니다.

참고 CORBA 서버 프로젝트를 시작할 때 IDL 파일을 추가하지 않은 경우, 나중에 Project|Add to Project를 선택하거나(기존 IDL 파일의 경우) New Items 다이얼로그 박스의 Multitier 페이지에서 CORBA IDL 파일을 선택하여(새 IDL 파일을 정의하는 경우) 언제든지 IDL 파일을 추가할 수 있습니다.

원하는 서버 타입을 지정하고 OK를 클릭하면 CORBA 서버 마법사는 지정된 타입의 새 서버 프로젝트를 만들고 프로젝트 파일 및 시작 코드에 CORBA 라이브러리를 추가하여 ORB(Object Request Broker) 및 BOA(Basic Object Adaptor)를 초기화합니다.

자동으로 생성된 코드는 다음과 같이 ORB 및 BOA와 통신하는 데 사용할 수 있는 두 변수 orb와 boa를 선언합니다.

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::BOA_var boa = orb->BOA_init(argc, argv);
```

콘솔 애플리케이션을 지정한 경우에는 다음 줄도 표시됩니다.

```
boa->impl_is_ready();
```

Windows 애플리케이션이 Windows 메시지 루프를 통해 Windows에서 메시지를 받을 수 있는 것처럼 이 호출은 애플리케이션이 BOA에서 메시지를 받을 수 있게 합니다. Windows 애플리케이션은 CORBA 메시지에도 Windows 메시지 루프를 사용합니다.

ORB와 BOA에 대한 자세한 내용은 VisiBroker 설명서를 참조하십시오.

IDL 파일에서 스텁 및 스켈레톤 생성

CORBA 인터페이스를 설명하는 IDL 파일을 CORBA 프로젝트에 추가했다면 이제 서버와 서버 클라이언트 애플리케이션 간의 통신을 처리하는 스텁 및 스켈레톤 클래스를 생성할 수 있습니다.

스텁 및 스켈레톤 클래스를 생성하려면 IDL 파일을 컴파일하기만 하면 됩니다. IDL 파일이 코드 에디터에 표시되었을 때 Project|Compile Unit을 선택하거나 Project Manager에서 IDL 파일을 마우스 오른쪽 버튼으로 클릭하고 Compile을 선택하여 이 파일을 컴파일할 수 있습니다. IDL 컴파일러는 Code Editor와 Project Manager에 표시되는 다음과 같은 파일 두 개를 새로 생성합니다.

- 서버 파일(xxx_s.cpp). 이 파일에는 스켈레톤 클래스의 구현이 포함됩니다.
- 클라이언트 파일(xxx_c.cpp). 이 파일에는 스텁 클래스의 구현이 포함됩니다.

Project Options 다이얼로그 박스의 CORBA 페이지를 사용하면 스텁 및 스켈레톤 클래스가 IDL 파일에서 생성되는 방법에 영향을 줄 수 있습니다. 예를 들어, 생성된 서버 유닛만 프로젝트에 포함하도록 지정하거나 서버에서 위임 모델(31-8페이지의 "위임 모델 사용"에서 설명)을 사용하는 경우 타이(tie) 클래스를 포함하도록 지정할 수 있습니다. 이러한 옵션은 현재 프로젝트에 포함된 모든 IDL 파일의 컴파일에 영향을 미칩니다.

CORBA Object Implementation 마법사 사용

IDL 파일에 정의된 CORBA 객체를 구현하려면 모든 스켈레톤 클래스에 해당하는 구현 클래스를 만들어야 합니다. 이렇게 하려면 File|New|Other를 선택하고 New Items 다이얼로그 박스의 Multitier 페이지에서 CORBA Object Implementation을 선택하십시오. 그러면 CORBA Object Implementation 마법사가 나타납니다.

CORBA Object Implementation 마법사는 프로젝트에 포함된 IDL 파일에 정의되어 있는 모든 인터페이스를 나열합니다. IDL 파일을 선택하고 이 IDL 파일에서 구현하고자 하는 인터페이스를 선택합니다. 구현 클래스의 이름을 제공하고 객체 정의와 구현을 포함하게 될 .h 및 .cpp 파일의 유닛 이름을 지정합니다.

이 마법사에서는 구현 클래스가 자동으로 생성된 스켈레톤 클래스의 자손이 되어야 하는지, 아니면 위임 모델(타이(tie) 클래스)을 사용하고 있는지 여부를 지정합니다. 위임 모델을 사용하고 있고 서버 애플리케이션에서 VCL이 가능하다면 이 마법사에서 구현 클래스를 데이터 모듈로 만들도록 지정할 수도 있습니다. 이렇게 하면 구현을 작성할 때 사용할 구현 데이터 모듈에 컴포넌트를 추가할 수 있습니다.

애플리케이션이 시작될 때 이 마법사에서 CORBA 객체를 인스턴스화하는 코드를 추가하도록 지정하여 CORBA 객체를 클라이언트 요청을 받는 데 사용할 수 있습니다. 시작 시 객체를 인스턴스화하는 경우 클라이언트가 객체를 찾을 수 있도록 이러한 객체에 이름을 제공해야 합니다. 이 이름은 생성자에 매개변수로 전달되며 인스턴스를 참조하는 변수의 이름이 아닙니다. CORBA 객체에 연결된 클라이언트 애플리케이션은 이 이름을 사용하여 원하는 객체를 나타냅니다.

OK를 클릭하면 CORBA Object Implementation 마법사가 구현 클래스에 대한 정의를 생성하고 모든 메소드에 대해 비어 있는 바디가 있는 구현을 생성합니다. 또한 이 마법사는 개발자가 그렇게 하도록 지정한 경우 객체를 인스턴스화하는 코드를 추가할 수도 있습니다.

이러한 변경 내용은 프로젝트에 추가하기 전에 확인 및 편집할 수 있습니다. 이렇게 하려면 OK를 클릭하기 전에 Show Updates 박스를 선택하십시오. 변경 내용을 확인 및 편집하고 나면 프로젝트에 이러한 변경 내용이 추가됩니다. 변경 내용을 확인 및 편집하지 않고 마법사를 종료한 경우, 나중에 코드 에디터에서 변경 내용을 편집할 수 있습니다.

여러 인터페이스를 구현하려면 각 인터페이스에 대해 한 번씩, CORBA Object Implementation 마법사를 여러 번 호출해야 합니다. 단, 이 마법사에서 여러 이름을 제공하여 단일 구현 클래스의 인스턴스를 여러 개 생성하도록 지정할 수 있습니다.

CORBA 객체 인스턴스화

모든 CORBA 객체에 대해 인스턴스화하는 방법을 결정해야 합니다. 다음과 같은 두 가지 방법이 있습니다.

- 서버 애플리케이션이 시작될 때 객체를 인스턴스화합니다. 이 경우, 서버가 시작될 때마다 클라이언트 애플리케이션에서 객체를 사용할 수 있습니다.
- 클라이언트 요청(기존 객체에서의 메소드 호출)에 응답하여 객체를 인스턴스화합니다. 이 경우, 서버 애플리케이션은 인터페이스 메소드의 구현 내에서 객체를 인스턴스화하고 이 메소드에서 객체에 대한 참조를 반환합니다.

애플리케이션이 시작될 때 객체를 인스턴스화하는 경우에는 CORBA Object 마법사의 메인 윈도우에서 Instantiate 박스를 선택합니다. 이 마법사는 구현 클래스를 인스턴스화하는 코드를 프로젝트의 소스 파일에 직접 추가합니다. 이 코드는 Project|View Source를 선택하여 확인할 수 있습니다. 다음과 같이 ORB_init 및 BOA_init를 호출하기 위해 CORBA 서버 마법사가 삽입한 줄과 impl_is_ready() 호출 사이에서 구현 클래스를 인스턴스화하고 객체가 요청을 받을 수 있다는 것을 boa에 알리는 코드를 볼 수 있습니다.

```
MyObjImpl MyObject_TheObject("TheObject"); // create the object instance
boa->obj_is_ready(&MyObject_TheObject); // inform the boa it can take
requests
```

다음과 같이 생성자와 obj_is_ready 호출 사이에 초기화 코드를 추가할 수 있습니다.

```
MyObjImpl MyObject_TheObject("TheObject"); // create the object instance
MyObject_TheObject.Initialize(50000);
boa->obj_is_ready(&MyObject_TheObject); // inform the boa it can take
requests
```

참고 타이(tie) 클래스(아래 참조)를 사용하는 경우 타이(tie) 클래스를 인스턴스화하는 호출도 있습니다.

시작 시 객체를 인스턴스화하지 않는 경우, 메소드에서 생성자(필요에 따라 타이(tie) 클래스에 대한 생성자 포함)를 호출하여 객체 인스턴스에 대한 참조를 반환할 수 있습니다. BOA의 obj_is_ready 메소드에도 호출을 추가하는 것이 좋습니다.

CORBA 클라이언트의 호출을 승인하는 객체에는 이름을 제공해야 하며, 그렇지 않으면 클라이언트가 이러한 객체를 찾지 못합니다. 일시적인 객체(클라이언트 호출에 응답하여 만들어지고 참조로 반환된 객체)에는 이름을 제공할 필요가 없습니다.

위임 모델 사용

디폴트로, CORBA 객체는 IDL 파일을 컴파일할 때 만들어진 스켈레톤 클래스의 자손입니다. 이것은 객체 클래스가 인터페이스 호출을 마샬링하고 BOA 와 상호 작용하는 모든 코드를 상속한다는 것을 의미합니다. 그러나 이미 작성되었거나, 다중 상속을 허용하지 않는 VCL 클래스의 자손이거나, CORBA 이외의 다른 애플리케이션과 공유하고자 하는 객체를 CORBA 서버에서 노출할 경우에는 CORBA나 생성된 서버 파일에 종속하지 않는 서버 클래스를 사용할 수 있습니다.

스켈레톤 클래스의 자손이 아닌 객체를 노출하려면 애플리케이션에서 위임 모델을 사용해야 합니다. 위임 모델에서는 스켈레톤 클래스의 인스턴스가 CORBA 객체를 직접 구현하는 대신 인터페이스 클래스를 완전히 독립적인 구현 클래스에 전달합니다. 즉, 구현 클래스는 스켈레톤의 자손이 아니라 스켈레톤에 의해 호출됩니다.

위임 모델을 사용하려면 IDL 파일을 컴파일할 때 타이(tie) 클래스를 생성해야 합니다. IDL 파일을 컴파일하기 전에 Project|Options를 선택하고 Project Options 다이얼로그 박스의 IDL 페이지에서 타이 클래스 생성 체크 박스를 선택합니다.

이렇게 하면 IDL 파일이 컴파일될 때, 스켈레톤 클래스 외에 특수한 타이 클래스가 서버 파일에 포함됩니다. 이 타이 클래스는 구현 클래스를 위한 대리자로서 작동하며, 메소드 구현을 위하여 구현 클래스에 위임됩니다(즉, 구현 클래스에 호출 전달).

CORBA Object 마법사를 사용하여 구현 클래스를 정의할 때, 구현 클래스가 스켈레톤 클래스의 자손으로 만들어지지 않도록 Delegation (Tie) 라디오 버튼을 선택합니다. 그러면 CORBA Object 마법사는 클래스 선언 옆에 주석을 추가합니다. C++Builder가 이 주석을 사용하여 클래스를 CORBA 구현 클래스로 식별하므로 이 주석은 제거하지 마십시오.

Tie를 선택하면 CORBA Object 마법사에서 구현 클래스를 생성하는 방법이 바뀌는 것 외에도, 객체를 인스턴스화하는 자동으로 생성된 코드에 영향을 줍니다.

구현 클래스를 인스턴스화한 직후 서버 애플리케이션은 구현 클래스 인스턴스를 생성자에 인수로 전달하여 연관된 타이 클래스를 인스턴스화해야 합니다. 구현 클래스가 시작 시 인스턴스화된 경우, 이 작업을 수행하는 코드가 CORBA Object 마법사에 의해 자동으로 생성됩니다. 그러나 런타임에서 CORBA 객체를 동적으로 인스턴스화할 경우에는 이 코드를 직접 추가해야 합니다. 예를 들면, 다음과 같습니다.

```
MyObjImpl myobj(); // instantiate implementation object
_tie_MyObj<MyObjImpl> tieobj(myobj, "InstanceName");
```

변경 내용 확인 및 편집

변경 내용을 프로젝트에 추가하기 전에 미리 확인하고 편집하는 것이 좋습니다. 이렇게 하면 다음 두 가지 목적을 달성할 수 있습니다.

- 프로젝트의 모든 변경 내용을 알 수 있습니다. 따라서 자동으로 생성된 코드에서 알 수 없는 동작이 일어나는 문제를 피할 수 있습니다.
- 구현 파일을 검색할 필요 없이 변경 내용을 사용자 정의할 수 있습니다. 이것은 특히 많은 코드가 포함된 파일에 변경 내용을 추가하는 경우 아주 유용합니다.

변경 내용을 확인 및 편집하려면 Project Updates 다이얼로그 박스를 사용하십시오. CORBA 마법사에서 Show Updates를 선택하거나 Edit|CORBA Refresh를 선택하면 이 다이얼로그 박스가 나타납니다.

Project Updates 다이얼로그 박스는 변경 내용을 왼쪽 창에 나열합니다. 변경 내용의 왼쪽에 있는 체크 박스를 선택 해제하여 해당 변경 내용을 취소할 수 있습니다. 취소한 변경 내용은 프로젝트에 적용되지 않으므로 다시 추가하려면 그러한 코드를 직접 작성해야 합니다.

참고 변경 내용은 작성된 순서로 나열됩니다. 특정 변경 내용을 제거하면 종속된 다른 변경 내용도 제거됩니다. 예를 들어, 새 파일 작성을 선택 해제하면 해당 파일에 추가되는 코드를 나타내는 변경 내용도 선택 해제됩니다.

변경 내용 리스트를 스크롤하면 추가되는 코드를 코드 창에서 볼 수 있습니다. 해당 코드를 포함하도록 변경되는 파일은 코드 창의 맨 아래에 나열됩니다.

변경 내용을 사용자 정의해야 할 경우, 코드 창을 편집할 수 있습니다. 예를 들어, 또 다른 조상 클래스를 구현 클래스에 추가하여 기존 클래스 중 하나의 동작을 상속할 수 있습니다. 또한 자동으로 생성된, 비어 있는 메소드 바디를 채우는 코드를 추가하여 메소드를 구현할 수 있습니다.

자동으로 생성된 변경 내용의 확인 및 편집이 끝나면 OK를 클릭합니다. 그러면 편집한 내용을 비롯하여 모든 변경 내용이 프로젝트에 추가됩니다. 변경 내용을 적용하지 않으려면 다이얼로그 박스를 취소합니다.

CORBA 객체 구현

위임 모델을 사용하지 않는 경우, 구현 클래스는 IDL 파일을 컴파일할 때 생성된 스켈레톤 클래스의 자손입니다. 위임 모델을 사용하는 경우에는 구현 클래스에 대한 명시적 조상이 존재하지 않습니다. 서버 애플리케이션의 다른 클래스에서 상속되도록 이 클래스 정의를 변경할 수 있지만, 스켈레톤 클래스에서 기존 상속을 제거하지는 마십시오.

IDL 파일인 account.idl에서 다음 선언을 한다고 가정합니다.

```
interface Account {
  float balance();
};
```

IDL 파일이 타이(tie) 클래스 없이 컴파일되면 생성된 서버 헤더(account_s.hh)에는 인터페이스의 모든 메소드에 대해 순수 가상 메소드를 갖는 스켈레톤 클래스 정의가 포함됩니다.

CORBA Object 마법사는 스켈레톤 클래스(_sk_Account)에서 구현 클래스를 파생시키는 구현 유닛을 만듭니다. 이 유닛의 헤더 파일은 다음과 같이 나타납니다.

```
#ifndef Interface1ServerH
```

```
#define Interface1ServerH
#include "account_s.hh"
//------------------------------------------------------------------------
----
class AccountImpl: public _sk_Account
{
  protected:
  public:
    AccountImpl(const char *object_name=NULL);
    CORBA::float balance();
};

#endif
```

구현에 필요한 다른 데이터 멤버 또는 메소드를 이 클래스 정의에 추가할 수 있습니다. 예를 들면, 다음과 같습니다.

```
#ifndef Interface1ServerH
#define Interface1ServerH
#include "account_s.hh"
//------------------------------------------------------------------------
----
class AccountImpl: public _sk_Account
{
  protected:
    CORBA::float _bal;
  public:
    void Initialize(CORBA::float startbal); // not available to clients
    AccountImpl(const char *object_name=NULL);
    CORBA::float balance();
};

#endif
```

서버 애플리케이션에서는 이러한 추가 메소드와 속성을 사용할 수 있지만 클라이언트에 노출되지는 않습니다.

생성된 .cpp 파일에서 다음과 같이 구현 클래스의 바디를 채워 작업 CORBA 객체를 만듭니다.

```
AccountImpl::AccountImpl(const char *object_name):
    _sk_Account(object_name)
{
}

CORBA::float AccountImpl::balance()
{
  return _bal;
};

void Initialize(CORBA::float startbal) // not available to clients
{
  _bal = startbal;
}
```

참고 CORBA 서버에서 BOA와 상호 작용하는 코드를 작성할 수 있습니다. 예를 들어, BOA를 사용하여 서버 객체를 임시로 숨기거나 비활성화했다가 나중에 다시 활성화할 수 있습니다. BOA와 상호 작용하는 코드를 작성하는 방법에 대한 자세한 내용은 *VisiBroker Programmer's Guide*를 참조하십시오.

스레드 충돌 방지

디폴트로, CORBA 애플리케이션은 멀티 스레드로 작동합니다. 이것은 CORBA 객체를 구현할 때 스레드 충돌을 방지해야 한다는 것을 의미합니다.

따로 지정하지 않으면 BOA는 클라이언트 스레드를 풀링하므로 클라이언트 요청은 준비된 스레드들 중 어느 것이라도 사용할 수 있습니다. 그러나 세션 당 스레드 규칙을 가진 BOA를 시작하여 각 클라이언트가 항상 동일한 스레드를 사용하게 할 수 있습니다. 각 클라이언트가 항상 동일한 스레드를 사용하면 스레드 변수에 영구적 클라이언트 정보를 저장할 수 있습니다. BOA를 시작할 때 스레드 규칙을 설정하는 방법에 대한 자세한 내용은 *VisiBroker Programmer's Guide*를 참조하십시오.

VisiBroker 라이브러리에는 스레드 충돌로부터 코드를 보호하는 데 도움이 되는 클래스가 포함되어 있습니다. 이러한 클래스는 VisiBroker include 디렉토리에 설치된 헤더 파일인 vthread.h에서 정의됩니다. VisiBroker 스레드 지원 클래스에는 뮤텍스(VisMutex), 조건 변수(VISCondition) 및 동시 읽기는 허용하고 쓰기는 배타적인 VCL의 잠금 동기화 장치와 비슷하게 작동하는 읽기/쓰기 잠금(VISRWLock)이 포함됩니다. 이러한 클래스를 사용하면 VisiBroker가 지원하는 모든 플랫폼에서 클래스를 이식할 수 있다는 장점이 있습니다.

예를 들어, 구현 클래스에 VisMutex 필드를 추가하여 인스턴스 데이터를 보호할 수 있습니다.

```
class A {
  VISMutex _mtx;
  ...
}
```

그런 다음 인스턴스 데이터에 대한 액세스를 동기화해야 할 경우 인스턴스 데이터를 액세스하는 임의의 메소드 내에서 뮤텍스를 잠글 수 있습니다.

```
void A::AccessSharedMemory(...)
{
  VISMutex_var lock(_mtx); // acquires a lock that is released on exit
// add code here to access the instance data
}
```

AccessSharedMemory가 실행을 완료하면 메소드 바디 내에 예외가 있더라도 뮤텍스가 해제된다는 점에 주의합니다.

서버 애플리케이션에서 VCL을 사용하는 경우 C++Builder는 스레드 충돌을 방지하는 데 도움이 되는 여러 클래스를 포함시켜 줍니다. 자세한 내용은 11장, "멀티 스레드 애플리케이션 개발"을 참조하십시오.

CORBA 인터페이스 변경

CORBA Object 마법사를 사용하여 IDL 파일의 구현 클래스를 생성한 후에 IDL 파일에서 인터페이스를 변경하는 경우, C++Builder를 사용하면 구현 클래스 작성에 들어간 노력이 수포로 돌아가는 일 없이 서버 프로젝트를 업데이트하여 변경 내용을 반영할 수 있습니다.

IDL 파일을 변경한 후, Edit|CORBA Refresh를 선택합니다. 이 명령은 자동으로 생성된 클라이언트 및 서버 파일에서 새 인터페이스 정의와 현재 컴파일러 옵션을 반영하도록 IDL 파일을 다시 컴파일합니다. IDL 파일이 성공적으로 컴파일되면 Project Updates 다이얼로그 박스를 사용하여 C++Builder에서 수행된 변경 내용을 미리 보고 편집할 수 있습니다.

참고 C++Builder는 기존 구현 클래스에 대응하는 인터페이스를 찾지 못하면 인터페이스 이름을 변경했는지 묻는 메시지를 표시합니다. 여기서 Yes를 선택하면, 구현 클래스를 인터페이스와 올바로 연결시킬 수 있도록 인터페이스 이름을 어떻게 변경했는지 묻는 메시지를 표시합니다.

CORBA Refresh 명령을 사용하면 새 메소드가 구현 클래스에 추가되고 선언이 업데이트되어 기존의 메소드 및 어트리뷰트(attribute)의 변경 내용을 반영합니다. 그러나 특정 변경 내용은 업데이트되지 않습니다. 특히 메소드나 어트리뷰트를 삭제한 경우 해당 코드는 삭제되지 않습니다. 대신 삭제된 메소드는 클래스의 일부로 계속 남아 있지만, CORBA 클라이언트에서 더 이상 사용할 수 없습니다. 마찬가지로 메소드나 어트리뷰트의 이름을 변경하면 삭제 및 추가와 동일하게 간주되어 이전 메소드나 어트리뷰트는 계속 남아 있고 새 메소드나 어트리뷰트에 대한 코드가 생성됩니다.

서버 인터페이스 등록

서버 객체의 클라이언트 호출에 대해 정적 연결만 사용할 경우에는 서버 인터페이스를 등록할 필요가 없지만, 일반적으로 인터페이스를 등록하는 것이 좋습니다. 다음 두 가지 유틸리티를 사용하여 인터페이스를 등록할 수 있습니다.

- **Interface Repository.** Interface Repository에 등록하면 클라이언트는 동적 호출 인터페이스(DII)를 사용할 때 인터페이스에 대한 정보를 프로그램으로 얻을 수 있습니다. DII 사용에 대한 자세한 내용은 31-15페이지의 "동적 호출 인터페이스 사용"을 참조하십시오. 또한 Interface Repository에 등록하면 다른 개발자가 클라이언트 애플리케이션 작성 시 등록된 인터페이스를 볼 수 있습니다.

  Tools|IDL Repository를 선택하면 Interface Repository에 인터페이스를 등록할 수 있습니다.

- **Object Activation Daemon.** OAD(Object Activation Daemon)에 등록하면 클라이언트에서 필요할 때까지 서버를 시작하거나 객체를 인스턴스화할 필요가 없습니다. 이렇게 하면 서버 시스템의 리소스가 절약됩니다.

CORBA 클라이언트 작성

CORBA 클라이언트 작성 시의 첫 번째 단계는 클라이언트 애플리케이션이 클라이언트 시스템 상의 ORB 소프트웨어와 대화할 수 있는지 확인하는 것입니다. 이렇게 하려면 CORBA Client 마법사를 사용하십시오. File|New|Other를 선택하고 New Items 다이얼로그 박스의 Multitier 페이지에서 CORBA Client 아이콘을 선택합니다. CORBA Client 마법사를 사용하면 콘솔 애플리케이션을 만들 것인지, 아니면 Windows 애플리케이션을 만들 것인지 지정할 수 있습니다.

CORBA 서버 애플리케이션에서와 마찬가지로 CORBA 클라이언트 애플리케이션의 VCL 클래스 사용 여부를 지정할 수 있습니다. VCL 체크 박스를 선택하지 않으면 생성된 모든 코드를 다른 플랫폼으로 이식할 수 있습니다.

CORBA Client 마법사에서 사용할 서버 객체의 인터페이스를 정의하는 모든 기존 IDL 파일을 포함합니다. 생성된 클라이언트 유닛을 프로젝트에 명시적으로 추가하여 IDL 파일을 포함하지 않고서도 CORBA 클라이언트 애플리케이션을 만들 수 있지만, 이 방법은 자주 사용되지 않습니다. 프로젝트에 서버 인터페이스에 대한 IDL 파일이 포함되면 Use CORBA Object 마법사를 사용하여 서버 상의 객체를 연결할 수 있습니다.

참고 CORBA 클라이언트 프로젝트를 시작할 때 IDL 파일을 추가하지 않은 경우, 나중에 Project|Add to Project를 선택하여 언제든지 IDL 파일을 추가할 수 있습니다.

CORBA Client 마법사는 항상 CORBA 라이브러리를 프로젝트 파일에 추가하는 지정된 타입의 새 클라이언트 프로젝트를 만들고 다음 시작 코드를 추가하여 ORB(Object Request Broker)를 초기화합니다.

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

콜백 인터페이스를 CORBA 서버에 전달하려면 클라이언트 애플리케이션에서 BOA(Basic Object Adaptor)도 초기화해야 합니다. 이렇게 하려면 이 마법사에서 해당 박스를 선택하기만 하면 됩니다.

이제 C++Builder에서 다른 애플리케이션을 작성하는 것과 동일한 방법으로 애플리케이션 작성을 시작합니다. 그러나 서버 애플리케이션에서 정의된 객체를 사용하려는 경우, 객체 인스턴스를 직접 작업하는 대신 CORBA 객체 참조를 사용합니다. CORBA 객체 참조는 정적 연결을 사용할 것인지, 아니면 동적 연결을 사용할 것인지에 따라 두 방법 중 하나로 얻을 수 있습니다.

정적 연결을 사용하려면 Edit|Use CORBA Object를 선택하여 Use CORBA Object 마법사를 호출합니다. 정적 연결을 사용하면 동적 연결을 사용할 때보다 속도가 향상되고 컴파일 타임 타입 검사 및 코드 완성 등의 추가 이점을 얻을 수 있습니다.

그러나 사용할 객체 또는 인터페이스를 런타임이 될 때까지 모를 수 있습니다. 이러한 경우에는 동적 연결을 사용할 수 있습니다. 동적 연결은 Any라 불리는 특수한 CORBA 타입을 사용하여 요청을 서버에 전달하는 일반적인 CORBA 객체를 사용합니다.

스텁 사용

스텁 클래스는 IDL 파일을 컴파일할 때 자동으로 생성됩니다. 스텁 클래스는 BaseName_c.cpp 및 BaseName_c.hh 형태의 이름을 갖는 생성된 클라이언트 파일에서 정의됩니다.

참고 Project Options 다이얼로그 박스의 CORBA 페이지를 사용하여 C++Builder에서 서버 파일을 제외하고 클라이언트(스텁) 파일만 생성하도록 지정할 수 있습니다.

CORBA 클라이언트 작성 시에는 생성된 클라이언트 파일에서 코드를 편집하지 않습니다. 대신 클라이언트 파일이 사용되는 스텁 클래스를 인스턴스화합니다. 이렇게 하려면 Edit|Use CORBA Object를 선택하여 Use CORBA Object 마법사를 호출하십시오.

Use CORBA Object 마법사에서 액세스할 인터페이스가 포함된 IDL 파일을 지정하고 사용할 인터페이스를 선택합니다. CORBA 객체의 명명된 특정 인스턴스에만 연결할 경우에는 CORBA 객체 이름을 제공할 수 있습니다.

Use CORBA Object 마법사를 사용하면 서버 객체에 연결하는 것에 대한 여러 메커니즘을 선택할 수 있습니다.

- 클라이언트 애플리케이션이 VCL 가능 Windows 애플리케이션인 경우, CORBA 객체 스텁 클래스의 인스턴스를 보유하는 애플리케이션의 폼 위에서 속성을 만들 수 있습니다. 그런 다음 이 속성을 CORBA 서버 객체의 인스턴스처럼 사용할 수 있습니다.
- 콘솔 애플리케이션을 만드는 중이면 이 마법사는 main() 함수에서 스텁 클래스를 변수로 인스턴스화할 수 있습니다. 마찬가지로 Windows 애플리케이션을 만드는 중이면 이 마법사는 WinMain() 함수에서 스텁 클래스를 변수로 인스턴스화할 수 있습니다.
- 만들려는 애플리케이션이 Windows 애플리케이션인지, 아니면 콘솔 애플리케이션인지에 관계 없이 이 마법사는 지정된 임의 유닛의 기존 클래스에 속성을 추가하거나, 스텁 인스턴스에 대한 속성을 포함하는 새 클래스를 시작할 수 있습니다.

선택한 메커니즘에 상관 없이 이 마법사는 필요한 헤더 파일을 추가하고 스텁 변수 또는 속성을 CORBA 서버 객체에 연결하는 코드를 생성합니다. 예를 들어, 다음 코드는 콘솔 애플리케이션의 main() 함수에서 MyServerObj라는 서버 인터페이스에 대한 스텁을 인스턴스화합니다.

```
#include <corba.h>
#include <condefs.h>
#include "MyServerObj_c.hh"
#pragma argsused
int main(int argc, char* argv[])
{
  try
  {
    // Initialize the ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    MyServerObj_var TheObject = MyServerObj::_bind("InstanceName");
  }
  catch(const CORBA::Exception& e)
  {
    cerr << e << endl;
    return(1);
  }
  return 0;
}
```

경고 서버 객체에 연결하는 코드를 애플리케이션의 다른 부분으로 복사하는 경우, 스텁 변수가 로컬 변수 또는 클래스의 데이터 멤버인지 확인합니다. 전역 스텁 변수는 orb 변수가 해제되기 전까지 CORBA 참조 카운트를 해제하지 않으므로 사용이 쉽지 않습니다. 전역 스텁 변수를 사용해야 하는 경우 orb가 해제되기 전에 전역 스텁 변수를 NULL로 설정하여 연결을 해제하십시오.

클라이언트 애플리케이션에 생성된 클라이언트 유닛(BaseName_c)만 포함되어 있는 경우에는 CORBA Object 마법사를 사용하여 서버 객체를 연결할 수 없고, 대신 이 코드를 직접 생성해야 합니다. 또한 고유한 바인드 코드를 작성하여 VisiBroker의 바인드 옵션을 사용할 수도 있습니다. 예를 들어, 특정 서버를 지정하거나 CORBA 연결이 손실된 경우 서버 연결을 자동으로 다시 시도하는 기능을 해제하는 등의 작업을 할 수 있습니다. 서버 객체에 연결하는 코드를 작성하는 방법에 대한 자세한 내용은 VisiBroker 설명서를 참조하십시오.

동적 호출 인터페이스 사용

동적 호출 인터페이스(DII)를 사용하면 클라이언트 애플리케이션은 인터페이스 호출을 명시적으로 마샬링하는 스텁 클래스를 사용하지 않고 서버 객체를 호출할 수 있습니다. 클라이언트가 요청을 보내기 전에 모든 타입의 정보를 인코딩한 다음 서버에서 이러한 정보를 디코딩해야 하기 때문에, DII를 사용하면 스텁 클래스를 사용할 때보다 속도가 느려집니다.

클라이언트 애플리케이션에서 DII를 사용하려면 다음을 수행해야 합니다.

1 요청 마샬링을 위한 코드가 구현에 포함되지 않는다는 점만 제외하면 스텁 객체와 동일하게 작동하는 일반적인 CORBA 객체를 만듭니다. 다음과 같이 DII 요청을 받아야 하는 객체를 나타내기 위해 객체의 리포지토리 ID를 제공해야 합니다.

```
CORBA::Object_var diiObj;
try
{
  diiObj = orb->bind("IDL:ServerObj:1.0");
} catch (const CORBA::Exception& E)
{
  // handle the bind exception.
}
```

2 일반적인 CORBA 객체를 사용하여 일반적인 CORBA 객체가 나타내는 객체의 특정 메소드에 대한 요청 객체를 만듭니다.

```
CORBA::Request_var req = diiObj->_request("methodName");
```

3 메소드가 사용하는 모든 인수를 추가합니다. 이러한 인수는 CORBA::NVList_ptr 타입의 요청이 있을 때 리스트에 추가됩니다. 각 인수는 가변 타입과 유사한 CORBA::Any 타입입니다. 명명된 값 리스트(CORBA::NVList)에서 임의 타입의 값을 포함하는 모든 인수 리스트를 처리해야 하므로 CORBA::Any 타입이 필요합니다.

```
CORBA::Any arg;
arg <<= (const char *)"argvalue";
CORBA::NVList_ptr arguments = req->arguments();
arguments->add_value("arg1", arg, CORBA::ARG_IN);
```

4 이제 다음과 같이 요청을 호출할 수 있습니다.

```
req->invoke();
```

5 마지막으로 오류를 검사하고 결과를 검색합니다.

```
if (req->env()->exception())
  // handle exception
else
{
  CORBA::Any_ptr pRetVal = req->result()->value();
  CORBA::float val;
  Any_ptr >>= val;
}
```

동적 호출 인터페이스를 사용하는 방법에 대한 자세한 내용은 *VisiBroker Programmer's Guide*를 참조하십시오.

CORBA 서버 테스트

C++Builder에는 서버 인터페이스를 테스트할 수 있는 범용 클라이언트를 생성하는 CORBATest 데모가 포함되어 있습니다. CORBATest는 CORBA 서버 애플리케이션에서 객체를 사용해 보는 스크립트를 기록하고 실행합니다.

참고 CORBATest를 사용하여 여러 서버를 테스트하려는 경우, 이 애플리케이션을 Tools 메뉴에 설치하는 것이 간편합니다. 이렇게 하려면 Tools|Configure Tools를 선택하고 Tools Options 다이얼로그 박스를 사용하여 CorbaTest.exe를 추가하십시오.

테스트 도구 설정

CORBATest를 사용하여 CORBA 애플리케이션을 테스트할 수 있으려면 다음과 같은 방법으로 서버 인터페이스를 실행 중인 Interface Repository에 등록해야 합니다.

1 메뉴 항목이 선택되어 있지 않으면, Tools|VisiBroker SmartAgent를 선택하여 VisiBroker SmartAgent가 실행 중인지 확인합니다.

2 테스트할 서버 애플리케이션을 실행합니다.

3 Tools|IDL Repository를 선택하여 서버 애플리케이션의 .idl 파일을 Interface Repository에 추가합니다. 실행 중인 임의의 Interface Repository를 사용할 수 있지만, Interface Repository가 다른 목적에 사용되는 것을 방해하지 않으면서 테스트를 수행할 수 있도록 테스트 전용 리포지토리를 사용하는 것이 일반적입니다. 테스트 전용 리포지토리를 시작하려면 Tools|IDL Repository를 선택한 다음 Update IDL Repository 다이얼로그 박스에서 Add IREP 버튼을 클릭하십시오.

4 corbatest 디렉토리(..\examples\corba\corbatest)의 데모 코드에서 컴파일하여 생성할 수 있는 CORBATest.exe를 실행하여 테스트 도구를 시작합니다.

테스트 스크립트 기록 및 실행

테스트 스크립트는 설정된 CORBA 객체 메소드로서 이러한 메소드에 전달되는 모든 매개변수와 함께 호출됩니다. 단일 스크립트는 동일한 메소드에 대한 여러 호출(예를 들어, 다른 매개변수 값을 가진 메소드를 테스트하는 경우)을 포함할 수 있습니다.

테스트 도구 상단의 가운데 창에 있는 명령 창에 명령을 추가하여 스크립트를 작성합니다. 다음과 같은 방법으로 명령을 추가합니다.

1. 왼쪽 상단의 객체 창에 호출할 메소드를 가진 객체의 탭 모양 페이지가 없는 경우, Edit|Add Object를 선택합니다. 리포지토리 ID로 객체를 선택할 수 있는 New Objects 다이얼로그 박스가 나타납니다. 객체의 리포지토리 ID는 "IDL:MyInterface1:1.0" 형태의 문자열이며, 여기서 "MyInterface"는 .idl 파일에 선언된 인터페이스 이름입니다. 객체를 선택하고 이름을 할당합니다.
2. 왼쪽 상단 창에서 추가할 메소드를 가진 객체의 탭을 선택합니다. 각 탭에는 단계 1에서 객체를 추가할 때 할당한 객체 이름이 표시됩니다. 탭 모양의 각 페이지는 테스트할 수 있는 객체의 작업(메소드)을 나열합니다.
3. 객체 창에서 테스트 도구 상단의 중앙에 있는 명령 창으로 작업을 끌어 명령을 스크립트에 추가합니다.
4. 명령 창에서 메소드를 선택하면 오른쪽 상단의 세부 정보 창에 메소드의 입력 매개변수가 나타납니다. 입력 매개변수에 값을 제공합니다. 상태 표시줄에는 현재 입력 매개변수에 필요한 값 타입이 나타납니다.

이러한 단계를 반복하여 원하는 객체 수만큼 명령을 추가합니다. 스크립트에 원하는 모든 명령이 포함되면, Edit|Save Script As를 선택하거나 Save Script 버튼을 클릭하여 스크립트를 저장하고 이름을 제공합니다. 그런 다음 File|New Script를 선택하거나 New Script 버튼을 클릭하여 새 스크립트를 시작할 수 있습니다.

참고 스크립트에 추가한 객체나 명령을 선택하고 Edit|Remove Object 또는 Edit|Remove Command를 선택하여 해당 객체나 명령을 제거할 수 있습니다.

Run|Run을 선택하거나 Run Script 버튼을 클릭하여 스크립트를 실행할 수 있습니다. 스크립트가 실행되면 테스트 도구는 동적 호출 인터페이스를 사용하여 CORBA 서버를 호출하고 지정된 매개변수 값을 전달합니다. 그런 다음 모든 메소드에 대한 반환 값(있는 경우)과 반환 매개변수가 테스트 도구의 결과 섹션이나 결과 파일(자동 테스트의 경우)에 표시됩니다.

결과 섹션은 스크립트에 있는 모든 명령의 결과를 결과 탭에 표시합니다. 또한 이러한 결과는 결과 섹션의 다른 페이지에도 표시되고, 원하는 정보를 쉽게 찾을 수 있도록 범주로 분류됩니다. 결과 섹션의 다른 페이지에는 반환 값, 입력 매개변수 값, 출력 매개변수 값, 스크립트 실행 중 발생한 오류 등이 포함되어 있습니다.

장

32

인터넷 서버 애플리케이션 생성

웹 서버 애플리케이션은 기존 웹 서버의 기능을 확장합니다. 웹 서버 애플리케이션은 웹 서버로부터 HTTP 요청 메시지를 받아 이 메시지에서 요청된 모든 액션을 수행하고 웹 서버로 다시 보낼 응답을 작성합니다. C++Builder 애플리케이션으로 수행할 수 있는 많은 작업은 웹 서버 애플리케이션에 통합될 수 있습니다.

C++Builder는 웹 서버 애플리케이션 개발을 위한 서로 다른 두 아키텍처인 Web Broker와 WebSnap을 제공합니다. WebSnap과 Web Broker 아키텍처는 서로 다르긴 하지만 많은 공통 요소들을 가지고 있습니다. WebSnap 아키텍처는 Web Broker의 수퍼셋 역할을 합니다. 또한 개발자가 애플리케이션을 실행하지 않고도 페이지의 컨텐트를 표시할 수 있는 Preview 탭과 같은 새로운 기능과 추가 컴포넌트를 제공합니다. WebSnap에서 개발된 애플리케이션은 Web Broker 컴포넌트를 포함할 수 있지만 Web Broker에서 개발된 애플리케이션은 WebSnap 컴포넌트를 포함할 수 없습니다.

이 장에서는 Web Broker 및 WebSnap 기술의 기능을 설명하고 인터넷 기반 클라이언트/서버 애플리케이션에 대한 일반적인 정보를 제공합니다.

Web Broker 및 WebSnap

모든 애플리케이션에는 사용자와 정보를 주고 받을 수 있는 기능이 포함되어 있습니다. 표준 C++Builder 애플리케이션에서는 다이얼로그 박스 및 스크롤 윈도우와 같은 일반적인 프런트 엔드 요소를 만들어 데이터를 액세스할 수 있게 해줍니다. 개발자들은 익숙한 C++Builder 폼 디자인 도구를 사용하여 이들 객체의 정확한 레이아웃을 지정할 수 있습니다. 그러나 웹 서버 애플리케이션은 서로 다르게 디자인되어야 합니다. 사용자에게 전달되는 모든 정보는 HTTP를 통해 전송되는 HTML 페이지 형식으로 작성되어야 합니다. 페이지는 클라이언트 시스템에서 웹 브라우저 애플리케이션에 의해 해석되며, 현재 상태에서 사용자의 특정 시스템에 적절한 형식으로 표시됩니다.

웹 서버 애플리케이션을 구축하는 첫 번째 단계는 Web Broker 또는 WebSnap 중에서 사용할 아키텍처를 선택하는 것입니다. 두 접근 방법은 다음과 같은 많은 동일한 기능을 제공합니다.

- CGI 및 Apache DSO 웹 서버 애플리케이션 타입 지원. 이 내용은 32-6페이지의 "웹 서버 애플리케이션 타입"에 설명되어 있습니다.
- 들어오는 클라이언트 요청이 각각의 스레드에서 처리되도록 멀티 스레드 지원
- 신속한 응답을 위한 웹 모듈 캐싱
- 크로스 플랫폼 개발. Windows와 Linux 운영 체제 사이에 웹 서버 애플리케이션을 쉽게 이식할 수 있습니다. 동일한 소스 코드를 양쪽 플랫폼 모두에서 컴파일할 수 있습니다.

Web Broker 컴포넌트와 WebSnap 컴포넌트는 모든 페이지 전송 기술을 처리합니다. WebSnap은 Web Broker를 기초로 사용하여 Web Broker 아키텍처의 모든 기능을 통합합니다. 그러나 WebSnap이 훨씬 강력한 페이지 생성 도구 집합을 제공합니다. 또한, WebSnap 애플리케이션에서는 서버사이드 스크립트를 사용하여 런타임 시 페이지 생성을 도울 수 있습니다. Web Broker에는 이 스크립트 기능이 없습니다. Web Broker에 제공되는 도구는 WebSnap에 제공되는 도구만큼 완벽하지 않으며 직관적이지도 않습니다. 새 웹 서버 애플리케이션을 개발할 경우 Web Broker보다는 WebSnap을 아키텍처로 선택하는 것이 더 좋습니다.

두 접근 방법 간의 주요한 차이점은 다음 표에 설명되어 있습니다.

표 32.1 Web Broker와 WebSnap 비교

Web Broker	WebSnap
역 호환 가능	WebSnap 애플리케이션은 컨텐트를 생성하는 모든 Web Broker 컴포넌트를 사용할 수 있지만 이들을 포함하는 웹 모듈과 디스패처는 새로운 기능입니다.
각 애플리케이션에는 웹 모듈이 하나만 허용됩니다.	다수의 웹 모듈은 애플리케이션을 여러 유닛으로 분할하여 여러 개발자들이 동일한 프로젝트에서 거의 충돌 없이 작업할 수 있게 합니다.
애플리케이션에는 웹 디스패처가 하나만 허용됩니다.	여러 개의 전용 디스패처가 서로 다른 타입의 요청을 처리합니다.
컨텐트를 만들기 위한 특화된 컴포넌트로는 페이지 프로듀서, InternetExpress 컴포넌트, 웹 서비스 컴포넌트 등이 있습니다.	Web Broker 애플리케이션에 나타날 수 있는 모든 컨텐트 프로듀서에 더하여 복잡한 데이터 방식 웹 페이지를 신속하게 생성할 수 있도록 디자인된 많은 다른 컨텐트 프로듀서를 지원합니다.
스크립트 지원 안함	서버사이드 스크립트 지원을 통해 HTML 생성 로직을 비즈니스 로직과 분리할 수 있습니다.
기본적으로 명명된 페이지 지원 안함	페이지 디스패처가 자동으로 명명된 페이지를 검색하고 서버사이드 스크립트에서 주소를 지정할 수 있습니다.

표 32.1 Web Broker와 WebSnap 비교(계속)

Web Broker	WebSnap
세션 지원 안함	세션은 단기간 동안 필요한 엔드 유저 정보를 저장합니다. 이 정보는 로그인/로그아웃 지원과 같은 작업에 사용될 수 있습니다.
모든 요청은 액션 항목이나 자동 디스패칭 컴포넌트를 사용하여 명시적으로 처리되어야 합니다.	디스패치 컴포넌트는 다양한 요청에 자동으로 응답합니다.
일부 특화된 컴포넌트만 생성될 컨텐트의 미리 보기를 제공합니다. 대부분의 개발은 비주얼하지 않습니다.	WebSnap은 웹 페이지를 보다 시각적으로 생성하고 디자인 타임에 결과를 보여 줍니다. 모든 컴포넌트에서 미리 보기를 사용할 수 있습니다.

Web Broker에 대한 자세한 내용은 33장, "Web Broker 사용"을 참조하십시오. WebSnap에 대한 자세한 내용은 34장, "WebSnap을 사용하여 웹 서버 애플리케이션 생성"을 참조하십시오.

용어 및 표준

인터넷 상의 활동을 제어하는 많은 프로토콜은 RFC(Request for Comment) 문서에서 정의되며, 이러한 문서는 인터넷 프로토콜 엔지니어링 및 개발 조직인 IETF(Internet Engineering Task Force)에 의해 작성, 업데이트 및 유지 보수됩니다. 인터넷 애플리케이션을 작성하는 데 유용한 RFC는 다음과 같습니다.

- RFC822, "Standard for the format of ARPA Internet text messages"에서는 메시지 헤더의 구조 및 내용에 대해 설명합니다.
- RFC1521, "MIME(Multipurpose Internet Mail Extensions) 1부: Mechanixms for Specifying and Describing the Format of Internet Message Bodies"에서는 multipart/multiformat 메시지를 캡슐화하고 전송하는 데 사용되는 방법에 대해 설명합니다.
- RFC1945, "Hypertext Transfer Protocol - HTTP/1.0"에서는 공동 하이퍼미디어 문서를 배포하는 데 사용되는 전송 메커니즘에 대해 설명합니다.

IETF는 자체 웹 사이트인 `www.ietf.cnri.reston.va.us`에서 RFC 라이브러리를 유지 보수합니다.

URL(Uniform Resource Locator)의 각 부분

URL(Uniform Resource Locator)은 네트워크상에서 사용할 수 있는 리소스의 위치에 대한 전체 설명입니다. URL은 애플리케이션에서 액세스할 수 있는 여러 부분들로 구성됩니다. 이 부분들에 대한 자세한 내용은 그림 32.1에 설명되어 있습니다.

그림 32.1 URL(Uniform Resource Locator)의 각 부분



첫 번째 부분은 엄밀하게 말해 URL의 일부는 아니지만 프로토콜(http)을 식별합니다. 이 부분에는 https(보안 http), ftp 등의 다른 프로토콜을 지정할 수 있습니다.

호스트 부분은 웹 서버와 웹 서버 애플리케이션을 실행하는 시스템을 식별합니다. 위의 그림에는 나와 있지 않지만 이 부분은 메시지를 받는 포트를 오버라이드할 수 있습니다. 일반적으로 각 프로토콜은 기본 포트를 가지므로 포트를 지정할 필요가 없습니다.

스크립트 이름 부분은 웹 서버 애플리케이션의 이름을 지정합니다. 이 웹 서버 애플리케이션은 웹 서버가 메시지를 전달하는 애플리케이션입니다.

스크립트 이름 다음에는 Path Info가 옵니다. 이 부분은 웹 서버 애플리케이션 내에서 메시지의 대상을 식별합니다. Path Info 값은 호스트 시스템의 디렉토리, 특정 메시지에 응답하는 컴포넌트의 이름, 들어오는 메시지의 처리를 분할하기 위해 웹 서버 애플리케이션이 사용하는 다른 메커니즘 등이 될 수 있습니다.

쿼리 부분은 이름을 가진 값들의 집합을 가지고 있습니다. 이러한 값과 그 이름은 웹 서버 애플리케이션에 의해 정의됩니다.

URI와 URL 비교

URL은 HTTP 표준인 RFC1945에 정의된 URI(Uniform Resource Identifier)의 부분 집합입니다. 웹 서버 애플리케이션은 최종 결과가 특정 위치에 있지는 않지만 필요에 따라 만들어지는 많은 소스로부터 컨텐트를 만드는 경우가 많습니다. URI는 특정 위치와 관련되지 않은 리소스를 설명할 수 있습니다.

HTTP 요청 헤더 정보

HTTP 요청 메시지에는 클라이언트, 요청 대상, 요청 처리 방법, 요청과 함께 보내진 컨텐트 등에 대한 정보를 설명하는 여러 개의 헤더가 들어 있습니다. 각 헤더는 뒤에 문자열 값이 따라오는 "Host" 등의 이름으로 식별됩니다. 예를 들어, 다음과 같은 HTTP 요청이 있다고 가정해 보십시오.

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

첫 번째 줄은 요청을 GET으로 식별합니다. GET 요청 메시지는 GET 다음의 URI(여기서는 /art/gallery.dll/animals?animal=doc&color=black)와 연결된 컨텐트를 반환하도록 웹 서버 애플리케이션에 요청합니다. 첫 번째 줄의 마지막 부분은 클라이언트가 HTTP 1.0 표준을 사용하고 있음을 나타냅니다.

두 번째 줄은 Connection 헤더이며 요청이 처리된 후에도 연결을 끊어서는 안된다는 것을 나타냅니다. 세 번째 줄은 User-Agent 헤더이며 요청을 생성한 프로그램에 대한 정보를 제공합니다. 네 번째 줄은 Host 헤더이며 연결을 설정하기 위해 사용한 호스트 이름과 서버의 포트에 대한 정보를 제공합니다. 마지막 줄은 Accept 헤더로서, 클라이언트가 적절한 응답으로 받아들일 수 있는 미디어 타입을 나열합니다.

HTTP 서버의 동작

웹 브라우저의 클라이언트/서버 특성은 언뜻 보기엔 간단합니다. 대부분의 사용자에게 World Wide Web에서 정보를 검색하는 것은 링크를 클릭하면 해당 정보가 화면에 나타나는 것과 같이 간단한 프로시저입니다. 반면, 좀 더 경험있는 사용자들은 HTML 구문의 특성과 사용된 프로토콜의 클라이언트/서버 특성을 어느 정도는 이해하고 있습니다. 이러한 정도면 일반적으로 간단한 웹 사이트 컨텐트를 충분히 만들 수 있습니다. 그러나 더 복잡한 웹 페이지를 만드는 사람은 HTML을 이용해서 자동으로 정보를 수집하고 표시하는 다양한 방법을 사용합니다.

웹 서버 애플리케이션을 개발하기 전에 클라이언트가 요청을 하는 방법과 서버가 클라이언트 요청에 응답하는 방법을 이해하는 것이 좋습니다.

클라이언트 요청 작성

사용자가 HTML 하이퍼텍스트 링크를 선택하거나 URL을 지정하면 브라우저는 일단 프로토콜, 지정된 도메인, 정보 경로, 날짜 및 시간, 운영 환경, 브라우저 자체 등에 대한 정보와 기타 컨텐트 정보를 수집합니다. 그런 후에 요청을 작성합니다.

예를 들어, 한 폼에서 버튼을 클릭하여 선택된 기준에 따라 이미지 페이지를 표시하기 위해 클라이언트는 다음과 같은 URL을 만듭니다.

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

이 URL은 www.TSite.com 도메인에 있는 HTTP 서버를 지정합니다. 클라이언트는 www.TSite.com에 접속하고 HTTP 서버에 연결한 다음 요청을 전달합니다. 이 요청은 다음과 같이 나타납니다.

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

클라이언트 요청 처리

웹 서버는 클라이언트로부터 요청을 받은 다음 설정에 따라 여러 액션을 수행할 수 있습니다. 요청의 /gallery.dll 부분을 프로그램으로 인식하도록 설정된 경우 서버는 요청에 대한 정보를 해당 프로그램에 전달합니다. 요청에 대한 정보를 프로그램에 전달하는 방법은 웹 서버 애플리케이션의 타입에 따라 다릅니다.

- 프로그램이 CGI(Common Gateway Interface) 프로그램일 경우 서버는 요청에 포함된 정보를 CGI 프로그램에 직접 전달합니다. 서버는 프로그램이 실행되는 동안 기다립니다. CGI 프로그램은 종료될 때 컨텐트를 다시 서버로 직접 전달합니다.
- 프로그램이 WinCGI일 경우 서버는 파일을 열고 요청 정보를 작성합니다. 그런 다음 클라이언트 정보가 포함된 파일의 위치와 Win-CGI 프로그램이 컨텐트를 만들기 위해 사용할 파일의 위치를 전달하여 Win-CGI 프로그램을 실행합니다. 서버는 프로그램이 실행되는 동안 기다립니다. 프로그램이 종료되면 서버는 Win-CGI 프로그램이 작성한 컨텐트 파일에서 데이터를 읽습니다.

- 프로그램이 DLL(Dynamic-Link Library)이면 서버는 필요한 경우 DLL을 로드하고 요청에 포함된 정보를 DLL에 구조체로 전달합니다. 서버는 프로그램이 실행되는 동안 기다립니다. DLL은 종료될 때 컨텐트를 다시 서버로 직접 전달합니다.

어떤 경우라도 프로그램은 요청에 따라 동작하고 데이터베이스 액세스, 간단한 테이블 조회나 계산, HTML 문서 생성이나 선택 등 프로그래머가 지정한 작업을 수행합니다.

클라이언트 요청에 응답하기

웹 서버 애플리케이션은 클라이언트 요청을 모두 처리한 후, HTML 페이지 또는 기타 MIME 컨텐트를 생성하고 클라이언트가 이것을 표시할 수 있도록 서버를 통해 클라이언트에 다시 전달합니다. 응답을 보내는 방법 역시 프로그램의 타입에 따라 달라집니다.

- Win-CGI 스크립트는 작업이 완료되면 HTML 페이지를 작성하여 파일에 기록하고, 다른 파일에 응답 정보를 기록한 다음 두 파일의 위치를 서버에 다시 전달합니다. 서버는 두 파일을 모두 열고 HTML 페이지를 클라이언트에 다시 전달합니다.
- DLL은 작업이 완료되면 HTML 페이지와 응답 정보를 직접 서버에 다시 전달하고, 서버가 이를 다시 클라이언트에 전달합니다.

웹 서버 애플리케이션을 DLL 로 만들면 개별 요청을 처리하는 데 필요한 프로세스 및 디스크 액세스 수가 감소하여 시스템 로드 및 리소스 사용이 줄어듭니다.

웹 서버 애플리케이션 타입

Web Broker를 사용할지 WebSnap을 사용할지 여부에 관계 없이 다섯 가지 표준 타입의 웹 서버 애플리케이션을 만들 수 있습니다. 또한 애플리케이션 로직을 디버깅할 수 있도록 웹 서버를 애플리케이션으로 통합하는 Web Application Debugger 실행 파일을 만들 수 있습니다. Web Application Debugger 실행 파일은 디버깅 전용입니다. 애플리케이션을 배포할 경우 다른 다섯 가지 타입 중 하나로 변환해야 합니다.

ISAPI 및 NSAPI

ISAPI 또는 NSAPI 웹 서버 애플리케이션은 웹 서버에 의해 로드되는 DLL입니다. 클라이언트 요청 정보는 구조체로 DLL에 전달되며, ISAPI/NSAPI 애플리케이션이 이를 분석하여 적절한 요청 및 응답 객체를 만듭니다. 각 요청 메시지는 각각의 실행 스레드에서 자동으로 처리됩니다.

CGI 독립 실행형

CGI 독립 실행형 웹 서버 애플리케이션은 표준 입력으로 클라이언트 요청 정보를 받아 표준 출력으로 결과를 서버에 다시 전달하는 콘솔 애플리케이션입니다. CGI 애플리케이션은 이 데이터를 분석하여 적절한 요청 및 응답 객체를 만듭니다. 각 요청 메시지들은 각각의 애플리케이션 인스턴스에 의해 처리됩니다.

Win-CGI 독립 실행형

Win-CGI 독립 실행형 웹 서버 애플리케이션은 서버에서 작성된 구성 설정(INI) 파일로부터 클라이언트 요청 정보를 받아 그 결과를 파일로 쓰면 서버가 해당 파일을 클라이언트에 다시 전달하는 Windows 애플리케이션입니다. 웹 서버 애플리케이션은 INI 파일을 분석하여 적절한 요청 및 응답 객체를 만듭니다. 요청 메시지는 각각의 애플리케이션 인스턴스에 의해 처리됩니다.

Apache

Apache 웹 서버 애플리케이션은 웹 서버에 의해 로드되는 DLL입니다. 클라이언트 요청 정보는 구조체로 DLL에 전달되며, Apache 웹 서버 애플리케이션은 이를 분석하여 적절한 요청 및 응답 객체를 만듭니다. 각 요청 메시지는 자동으로 각각의 실행 스레드에서 처리됩니다.

Apache 웹 서버 애플리케이션을 배포할 경우 Apache 설정 파일에 몇 가지 애플리케이션에 대한 정보를 지정해야 합니다. 디폴트 모듈 이름은 프로젝트 이름 끝에 _module이 붙습니다. 예를 들어, 프로젝트 이름이 Project1일 경우 모듈 이름은 Project1_module입니다. 마찬가지로, 디폴트 컨텐트 타입은 프로젝트 이름에 -content가 붙고 디폴트 핸들러 타입은 프로젝트 이름에 -handler가 붙습니다.

필요할 경우 이 지정을 프로젝트(.bpr) 파일에서 변경할 수 있습니다. 예를 들어, 프로젝트를 만들 경우 디폴트 모듈 이름은 프로젝트 파일에 저장됩니다. 예를 들면, 다음과 같습니다.

```
extern "C"
{
  Httpd::module __declspec(dllexport) Project1_module;
}
```

참고 프로젝트 이름을 바꾸어서 저장해도 해당 이름은 자동으로 바뀌지 않습니다. 프로젝트의 이름을 바꿀 때마다 프로젝트 파일의 모듈 이름을 프로젝트 이름과 일치하도록 변경해야 합니다. 모듈 이름이 변경되면 컨텐트 및 핸들러 정의가 자동으로 변경됩니다.

Apache 설정 파일의 모듈, 컨텐트, 핸들러 정의 사용에 대한 자세한 내용은 Apache 웹 사이트 httpd.apache.org에 있는 설명서를 참조하십시오.

Web App Debugger

위에서 설명한 서버 타입들은 작업 환경에 따라 장단점이 있지만 어느 타입도 디버깅에는 적합하지 않습니다. 애플리케이션을 배치하고 디버거를 동작시키면 웹 서버 애플리케이션을 디버깅하는 것은 다른 애플리케이션을 디버깅할 때보다 훨씬 더 복잡해질 수 있습니다.

다행히도, 웹 서버 애플리케이션 디버깅을 그렇게 복잡하게 할 필요는 없습니다. C++Builder에는 디버깅을 간단하게 해주는 Web App Debugger가 포함되어 있습니다. Web App Debugger는 개발 시스템에서 웹 서버 역할을 합니다. 웹 서버 애플리케이션을 Web App Debugger 실행 파일로 작성하면 생성 과정 중에 애플리케이션이 자동으로 배포됩니다. 애플리케이션을 디버깅하려면 Run|Run을 사용하여 디버깅을 시작하십시오. Tools|Web App Debugger를 선택하고 디폴트 URL을 클릭한 다음 표시되는 웹 브라우저에서 애플리케이션을 선택합니다. 애플리케이션이 브라우저 윈도우에서 시작되면 IDE를 사용하여 브레이크포인트를 설정하고 디버깅 정보를 가져올 수 있습니다.

작업 환경에서 애플리케이션을 테스트하거나 배포할 준비가 되면 아래와 같은 절차에 따라 Web App Debugger 프로젝트를 다른 대상 타입 중 하나로 변환할 수 있습니다.

참고 Web App Debugger 프로젝트를 만들 경우 프로젝트에 대한 CoClass Name을 제공해야 합니다. 이 이름은 단순히 Web App Debugger가 애플리케이션을 참조하는 데 사용되는 이름입니다. 대부분의 개발자들은 애플리케이션의 이름을 CoClass Name으로 사용합니다.

웹 서버 애플리케이션 대상 타입 변환

Web Broker와 WebSnap의 강력한 기능 중 하나는 서로 다른 여러 가지 대상 서버 타입을 제공한다는 것입니다. C++Builder를 사용하여 대상 타입을 쉽게 변환할 수 있습니다.

Web Broker와 WebSnap은 약간 다른 디자인 철학을 가지고 있기 때문에 아키텍처마다 다른 변환 방법을 사용해야 합니다. 다음과 같은 방법으로 Web Broker 애플리케이션 대상 타입을 변환합니다.

1. 웹 모듈을 마우스 오른쪽 버튼으로 클릭하고 Add To Repository를 선택합니다.
2. Add To Repository 다이얼로그 박스에서 웹 모듈에 제목, 텍스트 설명, Repository 페이지(대체로 데이터 모듈), 작성자 이름, 아이콘 등을 지정합니다.
3. OK를 선택하여 웹 모듈을 템플릿으로 저장합니다.
4. 메인 메뉴에서 File|New를 선택한 다음 Web Server Application을 선택합니다. New Web Server Application 다이얼로그 박스에서 해당 대상 타입을 선택합니다.
5. 자동으로 생성된 웹 모듈을 삭제합니다.
6. 메인 메뉴에서 File|New를 선택한 다음 단계 3에서 저장한 템플릿을 선택합니다. 이 템플릿은 단계 2에서 지정한 페이지에 놓여집니다.

다음과 같은 방법으로 WebSnap 애플리케이션의 대상 타입을 변환합니다.

1. IDE에서 프로젝트를 엽니다.
2. View|Project Manager를 사용하여 Project Manager를 표시합니다. 모든 유닛이 표시되도록 프로젝트를 확장합니다.
3. Project Manager에서 New 버튼을 클릭하여 새 웹 서버 애플리케이션 프로젝트를 만듭니다. WebSnap 탭에서 WebSnap Application 항목을 더블 클릭합니다. 사용할 서버 타입을 포함하여 프로젝트에 대한 해당 옵션을 선택한 다음 OK를 클릭합니다.
4. Project Manager에서 새 프로젝트를 확장합니다. 여기에 표시되는 파일을 선택하여 삭제합니다.
5. Web App Debugger 프로젝트에 있는 폼 파일은 제외하고 프로젝트에 있는 각 파일을 한 번에 하나씩 선택하여 새 프로젝트로 끌어 놓습니다. 해당 파일을 새 프로젝트에 추가할지 묻는 다이얼로그 박스가 나타나면 Yes를 클릭합니다.

서버 애플리케이션 디버깅

웹 서버 애플리케이션은 웹 서버에서 보낸 메시지에 응답하여 실행되므로 이 애플리케이션에 대한 디버깅은 몇 가지 고유한 문제를 야기합니다. 또한 웹 서버가 루프를 벗어나게 되고 애플리케이션이 예상 요청 메시지를 찾지 않기 때문에 간단하게 IDE에서 애플리케이션을 시작할 수 없습니다.

다음 항목에서는 웹 서버 애플리케이션을 디버깅하는 데 사용할 수 있는 기술을 설명합니다.

Web Application Debugger 사용

Web Application Debugger를 사용하면 HTTP 요청, 응답, 응답 시간 등을 쉽게 모니터할 수 있습니다. Web Application Debugger는 웹 서버를 대신합니다. 일단 애플리케이션 디버깅이 끝나면 애플리케이션을 지원되는 웹 애플리케이션 타입 중 하나로 변환하고 상업용 웹 서버에 설치할 수 있습니다.

Web Application Debugger를 사용하려면 먼저 웹 애플리케이션을 Web Application Debugger 실행 파일로 만들어야 합니다. Web Broker를 사용하든 WebSnap을 사용하든 간에 웹 서버 애플리케이션을 만드는 마법사에는 프로젝트를 처음 생성할 때 이 실행 파일이 옵션으로 제공됩니다. 이 옵션은 COM 서버이기도 한 웹 서버 애플리케이션을 만듭니다.

Web Broker를 사용하여 웹 서버 애플리케이션을 작성하는 방법에 대한 자세한 내용은 33장, "Web Broker 사용"을 참조하십시오. WebSnap 사용에 대한 자세한 내용은 34장, "WebSnap을 사용하여 웹 서버 애플리케이션 생성"을 참조하십시오. Web Application Debugger를 사용하여 애플리케이션 시작 웹 서버 애플리케이션을 개발했으면 다음과 같은 방법으로 실행하고 디버깅할 수 있습니다.

1. IDE에 로드된 프로젝트에서 다른 실행 파일들처럼 애플리케이션을 디버깅할 수 있도록 브레이크포인트를 설정합니다.
2. Run|Run을 선택합니다. 이렇게 하면 웹 서버 애플리케이션인 COM 서버의 콘솔 윈도우가 나타납니다. 애플리케이션을 처음으로 실행할 때 Web App debugger에서 액세스할 수 있도록 COM 서버를 등록합니다.
3. Tools|Web App Debugger를 선택합니다.
4. Start 버튼을 클릭합니다. 이렇게 하면 디폴트 브라우저에 ServerInfo 페이지가 나타납니다.
5. ServerInfo 페이지에는 등록된 모든 Web Application Debugger 실행 파일의 드롭다운 리스트가 제공됩니다. 드롭다운 리스트에서 애플리케이션을 선택합니다. 이 드롭다운 리스트에 자신의 애플리케이션이 없을 경우 해당 애플리케이션을 실행 파일로 실행합니다. 애플리케이션이 실행되어야 등록할 수 있습니다. 드롭다운 리스트에 아직 애플리케이션이 없으면 웹 페이지를 새로 고쳐 봅니다. 웹 브라우저에서 이 페이지를 캐싱해도 최신 변경 내용을 볼 수 없는 경우도 있습니다.
6. 드롭다운 리스트에서 애플리케이션을 선택했으면 Go 버튼을 누릅니다. 이렇게 하면 Web Application Debugger에서 애플리케이션이 시작되어 애플리케이션과 Web Application Debugger가 주고 받는 요청 및 응답 메시지에 대한 자세한 내용을 볼 수 있습니다.

애플리케이션을 다른 타입의 웹 서버 애플리케이션으로 변환

Web Application Debugger에서 웹 서버 애플리케이션 디버깅이 완료되면 애플리케이션을 상업용 웹 서버에 설치될 수 있는 다른 타입으로 변환해야 합니다. 애플리케이션 변환에 대한 자세한 내용은 32-8페이지의 "웹 서버 애플리케이션 대상 타입 변환"을 참조하십시오.

DLL 웹 애플리케이션 디버깅

ISAPI, NSAPI 및 Apache 애플리케이션은 실질적으로는 미리 정의된 엔트리 포인트를 가지고 있는 DLL입니다. 웹 서버는 이 엔트리 포인트를 호출하여 요청 메시지를 애플리케이션에 전달합니다. 이 애플리케이션들은 DLL이기 때문에 서버를 시작할 애플리케이션 실행 매개변수를 설정하여 디버깅할 수 있습니다.

애플리케이션의 실행 매개변수를 설정하려면 Run|Parameters를 선택하고 Host Application 및 Run Parameters를 설정하여 웹 서버에 대한 실행 파일과 해당 파일을 시작하는 데 필요한 매개변수를 지정합니다. 웹 서버상의 이들 값에 대한 자세한 내용은 웹 서버 업체에서 제공하는 설명서를 참조하십시오.

참고 일부 웹 서버에서 이 방법으로 Host Application을 시작할 수 있으려면 몇 가지 사항을 추가로 변경해야 합니다. 자세한 내용은 웹 서버 업체에 문의하십시오.

팁 IIS 5가 있는 Windows 2000을 사용 중인 경우 권한을 올바르게 설정하는 데 필요한 모든 변경에 대한 자세한 내용은 다음 웹 사이트에 설명되어 있습니다.

```
http://community.borland.com/article/0,1410,23024,00.html
```

Host Application과 Run Parameters를 설정한 경우 서버가 요청 메시지를 DLL에 전달할 때 브레이크포인트 중 하나를 눌러 정상적으로 디버깅할 수 있도록 브레이크포인트를 설정할 수 있습니다.

참고 애플리케이션의 실행 매개변수를 사용하여 웹 서버를 시작하기 전에 서버가 이미 실행 중이 아닌지 확인합니다.

DLL 디버깅에 필요한 사용자 권한

Windows에서 DLL을 디버깅하려면 올바른 사용자 권한이 있어야 합니다. 다음과 같은 방법으로 권한을 획득할 수 있습니다.

1. 제어판의 **관리 도구**에서 **로컬 보안 정책**을 클릭합니다. **로컬 정책**을 확장하고 **사용자 권한 할당**을 더블 클릭합니다. 오른쪽 창에서 **운영 체제의 일부로 활동**을 더블 클릭합니다.
2. **추가**를 선택하여 사용자를 리스트에 추가합니다. 현재 사용자를 추가합니다.
3. 재부팅하여 변경 내용을 적용합니다.

장

33

Web Broker 사용

컴포넌트 팔레트의 Internet 탭에 있는 Web Broker 컴포넌트를 사용하면 특정 URI (Uniform Resource Identifier)와 연결되는 이벤트 핸들러를 만들 수 있습니다. 처리가 완료되면 프로그램에서는 HTML 또는 XML 문서를 생성하여 클라이언트에 전송할 수 있습니다. Web Broker 컴포넌트를 사용하면 크로스 플랫폼 애플리케이션을 개발할 수 있습니다.

웹 페이지의 컨텐트는 데이터베이스에서 가져오는 경우가 많습니다. 인터넷 컴포넌트를 사용하면 단일 DLL이 무수히 많고 스레드에 대해 안전한 데이터베이스 연결을 동시에 처리하면서 데이터베이스에 대한 연결을 자동으로 관리할 수 있습니다.

이 장의 다음 단원에서는 Web Broker 컴포넌트를 사용하여 웹 서버 애플리케이션을 만드는 방법에 대해 설명합니다.

Web Broker를 사용하여 웹 서버 애플리케이션 생성

다음과 같은 방법으로 Web Broker 아키텍처를 사용하여 새로운 웹 서버 애플리케이션을 만듭니다.

1 File | New | Other를 선택합니다.

2 New Items 다이얼로그 박스에서 New 탭을 선택하고 Web Server Application을 선택합니다.

3 그러면 다음 웹 서버 애플리케이션 타입 중 하나를 선택할 수 있는 다이얼로그 박스가 나타납니다.

- ISAPI 및 NSAPI: 이 애플리케이션 타입을 선택하면 프로젝트가 웹 서버에서 요구하는 엑스포트된 메소드가 있는 DLL로 설정됩니다. 또한 애플리케이션 작성을 위한 해당 헤더 파일도 포함됩니다.
- Apache: 이 애플리케이션 타입을 선택하면 프로젝트가 Apache 웹 서버에서 요구하는 엑스포트된 메소드가 있는 DLL로 설정됩니다. 또한 애플리케이션 작성을 위한 해당 헤더 파일도 포함됩니다.

- CGI 독립 실행형: 이 애플리케이션 타입을 선택하면 프로젝트가 콘솔 애플리케이션으로 설정되고 해당 헤더 파일이 포함됩니다.
- Win-CGI 독립 실행형: 이 애플리케이션 타입을 선택하면 프로젝트가 Windows 애플리케이션으로 설정되고 해당 헤더 파일이 포함됩니다.
- Web Application Debugger 독립 실행형: 이 애플리케이션 타입을 선택하면 웹 서버 애플리케이션을 위한 개발 및 테스트 환경이 설정됩니다. 이 애플리케이션 타입은 배포용이 아닙니다.

작성할 애플리케이션이 사용하게 될 웹 서버 타입과 통신할 웹 서버 애플리케이션의 타입을 선택합니다. 그러면 인터넷 컴포넌트를 사용하도록 구성되고 비어 있는 웹 모듈이 포함된 새 프로젝트가 만들어집니다.

웹 모듈

웹 모듈 (*TWebModule*) 은 *TDataModule* 에서 상속된 클래스이며 다음과 같은 방법으로 사용됩니다. 웹 애플리케이션의 비즈니스 룰과 넌비주얼(nonvisual) 컴포넌트를 중앙에서 관리합니다.

애플리케이션이 응답 메시지를 생성하는 데 사용하는 컨텐트 프로듀서를 추가합니다. 이 경우 *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer*, *TInetXPageProducer* 등과 같은 기본 컨텐트 프로듀서 또는 사용자가 작성한 *TCustomContentProducer*의 자손들 중 하나가 될 수 있습니다. 애플리케이션이 데이터베이스에서 가져온 자료를 포함한 응답 메시지를 생성한다면, 멀티 티어 데이터베이스 애플리케이션에서 클라이언트 역할을 하는 특수 컴포넌트나 데이터 액세스 컴포넌트를 추가할 수 있습니다.

웹 모듈은 넌비주얼 컴포넌트와 비즈니스 룰을 가지고 있는 역할뿐만 아니라, 들어오는 HTTP 요청 메시지를 응답을 생성할 액션 항목과 비교하는 디스패처 역할도 수행합니다.

개발자는 이미 웹 애플리케이션에서 사용할 많은 넌비주얼 컴포넌트와 비즈니스 룰이 설정된 데이터 모듈을 갖고 있을 수도 있습니다. 이럴 경우 기존 데이터 모듈을 웹 모듈로 바꿀 수 있습니다. 단순히 자동으로 생성된 웹 모듈을 삭제하고 기존의 데이터 모듈을 대신 추가하면 됩니다. 그런 다음, 웹 모듈의 경우처럼 데이터 모듈에 *TWebDispatcher* 컴포넌트를 추가하여 요청 메시지를 액션 항목에 디스패치할 수 있도록 합니다. 들어오는 HTTP 요청 메시지에 응답하기 위해 액션 항목이 선택되는 방법을 변경하려면 *TCustomWebDispatcher*에서 새 디스패처 컴포넌트를 파생시켜 데이터 모듈에 추가합니다.

프로젝트에는 디스패처가 하나만 포함될 수 있습니다. 디스패처는 프로젝트를 만들 때 자동으로 생성되는 웹 모듈이거나 웹 모듈을 대체하는 데이터 모듈에 추가된 *TWebDispatcher* 컴포넌트일 수 있습니다. 실행 중에 디스패처가 포함된 데이터 모듈이 추가적으로 만들어지면 웹 서버 애플리케이션은 런타임 오류를 생성합니다.

참고 디자인 타임에 설정한 웹 모듈은 실제로 템플릿입니다. ISAPI 및 NSAPI 애플리케이션에서는 요청 메시지마다 각각의 스레드를 생성하며, 스레드에 대한 각각의 웹 모듈 인스턴스와 컨텐트가 동적으로 만들어집니다.

경고 DLL 기반 웹 서버 애플리케이션의 웹 모듈은 나중에 재사용할 수 있도록 캐싱되어 응답 속도를 향상시킵니다. 디스패처의 상태와 액션 리스트는 뒤이어 요청이 들어오더라도 다시 초기화되지 않습니다. 요청을 실행하는 동안 액션 항목을 활성화하거나 비활성화하면 해당 모듈이 이후의 클라이언트 요청에 사용될 때 예기치 않은 결과가 발생될 수 있습니다.

웹 애플리케이션 객체

웹 애플리케이션을 위해 설정되는 프로젝트에는 *Application*이라는 전역 변수가 포함됩니다. *Application*은 작성 중인 애플리케이션의 타입에 적당한 *TWebApplication*의 자손(*TISAPIApplication*, *TApacheApplication* 또는 *TCGIApplication*)입니다. Application은 웹 서버가 받은 HTTP 요청 메시지에 의해 실행됩니다.

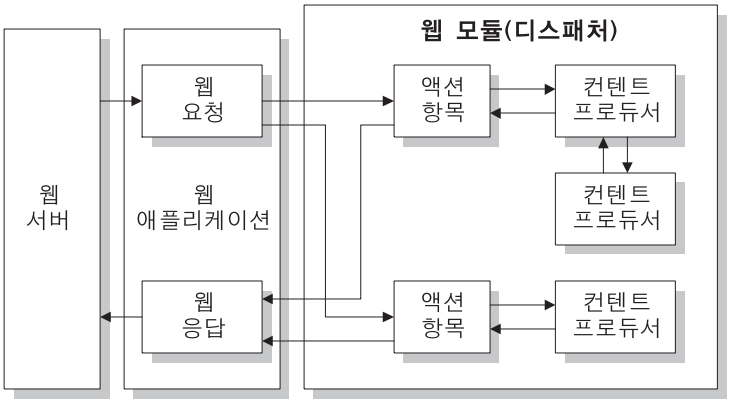
경고 프로젝트 파일에서 CGIApp.hpp 또는 ApacheApp.hpp 유닛의 include 문 뒤에 Forms.hpp를 포함시키지 마십시오. Forms.hpp 역시 *Application*이라는 전역 변수를 가지고 있기 때문에 CGIApp.hpp 또는 ISAPIApp.hpp 뒤에 Forms.hpp를 표시하면 *Application*이 잘못된 타입의 객체로 초기화됩니다.

Web Broker 애플리케이션의 구조

HTTP 요청 메시지를 받으면 웹 애플리케이션은 해당 메시지를 표시할 *TWebRequest* 객체와 돌려줄 응답을 표시할 *TWebResponse* 객체를 만듭니다. 그런 다음, 이 객체들을 웹 디스패처(웹 모듈 또는 *TWebDispatcher* 컴포넌트)에 전달합니다.

웹 디스패처는 웹 서버 애플리케이션의 흐름을 제어합니다. 디스패처는 특정 타입의 HTTP 요청 메시지에 대한 처리 방법을 알고 있는 액션 항목(*TWebActionItem*)의 컬렉션을 가지고 있습니다. 디스패처는 적절한 액션 항목 또는 자동 디스패칭 컴포넌트를 인식하여 HTTP 요청 메시지를 처리하며, 요청된 모든 작업을 수행하거나 응답 메시지를 생성할 수 있도록 요청 및 응답 객체를 식별된 핸들러에 전달합니다. 자세한 내용은 33-4페이지의 "웹 디스패처"에 설명되어 있습니다.

그림 33.1 서버 애플리케이션의 구조



액션 항목은 요청을 읽고 응답 메시지를 구성하는 작업을 담당합니다. 특화된 컨텐트 프로듀서 컴포넌트는 액션 항목이 동적으로 사용자 정의 HTML 코드나 다른 MIME 컨텐트를 포함할 수 있는 응답 메시지를 동적으로 생성하는 것을 도와 줍니다. 컨텐트 프로듀서는 다른 컨텐트 프로듀서나 *THTMLTagAttributes*의 자손을 사용하여 액션 항목이 응답 메시지의 컨텐트를 만들 수 있게 합니다. 컨텐트 프로듀서에 대한 자세한 내용은 33-13페이지의 "응답 메시지 컨텐트 생성"을 참조하십시오.

멀티 티어 데이터베이스 애플리케이션에서 웹 클라이언트를 만들 경우, 웹 서버 애플리케이션에 XML로 인코드된 데이터베이스 정보를 나타내는 추가적인 자동 디스패칭 컴포넌트와 javascript로 인코드된 데이터베이스 조작 클래스가 포함될 수 있습니다. 웹 서비스를 구현하는 서버를 만들 경우 웹 서버 애플리케이션에 SOAP 기반 메시지를 인보커에게 전달하여 해석 및 실행하게 하는 자동 디스패칭 컴포넌트가 포함될 수 있습니다. 디스패처는 모든 액션 항목을 수행한 다음 이 자동 디스패칭 컴포넌트를 호출하여 요청 메시지를 처리합니다.

모든 액션 항목 또는 자동 디스패칭 컴포넌트가 *TWebResponse* 객체를 채워 응답 생성을 끝내면 디스패처는 결과를 웹 애플리케이션에 돌려줍니다. 그러면 웹 애플리케이션은 웹 서버를 통해 클라이언트에 응답을 보냅니다.

웹 디스패처

웹 모듈을 사용할 경우 웹 모듈이 웹 디스패처 역할을 합니다. 기존 데이터 모듈을 사용할 경우 해당 데이터 모듈에 디스패처 컴포넌트(*TWebDispatcher*)를 하나 추가해야 합니다. 디스패처는 특정 유형의 요청 메시지에 대한 처리 방법을 알고 있는 액션 항목의 컬렉션을 가지고 있습니다. 웹 애플리케이션은 요청 객체와 응답 객체를 디스패처에 전달할 때 요청에 응답할 하나 이상의 액션 항목을 선택합니다.

디스패처에 액션 추가

디스패처의 *Actions* 속성에 있는 생략 부호 (...)를 클릭하여 Object Inspector 에서 액션 에디터를 엽니다. 액션 에디터에서 Add 버튼을 클릭하여 디스패처에 액션 항목을 추가할 수 있습니다.

다른 요청 메소드나 대상 URI에 응답하려면 디스패처에 액션을 추가합니다. 액션 항목은 다양한 방법으로 설정할 수 있습니다. 요청을 미리 처리하는 액션 항목에서 시작하여 응답을 보냈는지 아니면 오류 코드를 반환했는지의 응답 완료 여부를 검사하는 디폴트 액션으로 마무리할 수 있습니다. 또는 각 액션 항목이 요청을 완벽하게 처리할 수 있도록 모든 타입의 요청에 대해 각각의 액션 항목을 추가할 수 있습니다.

액션 항목은 33-6페이지의 "액션 항목"에 더 자세히 설명되어 있습니다.

요청 메시지 디스패칭

디스패처는 클라이언트 요청을 받으면 *BeforeDispatch* 이벤트를 생성합니다. 이렇게 하면 요청 메시지가 액션 항목에 전달되기 전에 요청 메시지를 미리 처리할 수 있습니다.

그런 다음 디스패처는 요청 메시지에 있는 대상 URL의 PathInfo 부분과 일치하면서 요청 메시지의 메소드로 지정된 서비스를 제공하는 항목을 액션 항목 리스트에서 검색합니다. 이 작업은 *TWebRequest* 객체의 *PathInfo* 및 *MethodType* 속성을 액션 항목에 있는 같은 이름의 속성과 비교하여 수행합니다.

디스패처가 적절한 액션 항목을 찾아내면 해당 액션 항목을 실행시킵니다. 실행된 액션 항목은 다음 중 하나를 수행합니다.

- 응답 컨텐트를 완성하고 응답 또는 요청이 완전히 처리되었다는 신호를 보냅니다.
- 응답을 추가하고 다른 액션 항목이 작업을 완료할 수 있게 해 줍니다.
- 다른 액션 항목에 대한 요청을 지연시킵니다.

모든 액션 항목을 확인한 후 메시지가 처리되지 않으면 디스패처는 액션 항목을 사용하지 않는 특수 등록 자동 디스패칭 컴포넌트를 확인합니다. 이 컴포넌트는 멀티 티어 데이터베이스 애플리케이션 전용이며 자세한 내용은 29-31페이지의 "InternetExpress를 이용한 웹 애플리케이션 개발"에 설명되어 있습니다.

모든 액션 항목과 자동 디스패칭 컴포넌트를 확인한 후에도 요청 메시지가 완벽하게 처리되지 않으면 디스패처는 디폴트 액션 항목을 호출합니다. 디폴트 액션 항목은 대상 URL이나 요청 메소드와 일치하지 않을 수도 있습니다.

디스패처가 액션 리스트(디폴트 액션이 있을 경우 디폴트 액션 포함)의 끝에 도달해도 실행된 액션이 없을 경우 서버에는 아무 것도 전달되지 않습니다. 이 경우 서버는 단순히 클라이언트와의 연결을 끊어버립니다.

액션 항목이 요청을 처리하면 디스패처는 *AfterDispatch* 이벤트를 실행합니다. 이 때 애플리케이션은 마지막으로 생성된 응답을 체크하고 내용을 변경할 수 있습니다.

액션 항목

각 액션 항목(*TWebActionItem*)은 주어진 요청 메시지의 타입에 따라 특정 작업을 수행합니다.

액션 항목은 요청에 완전하게 응답하거나, 부분적으로 응답하고 다른 액션 항목이 작업을 완료할 수 있게 합니다. 액션 항목은 요청에 대해 HTTP 응답 메시지를 보내거나 다른 액션 항목이 완료할 수 있도록 응답의 일부를 설정할 수 있습니다. 액션 항목이 응답을 완료했지만 보내지 않았을 경우 웹 서버 애플리케이션이 응답 메시지를 보냅니다.

액션 항목이 실행되는 조건

대부분의 액션 항목 속성은 HTTP 요청 메시지를 처리하기 위해 디스패처가 액션 항목을 선택하는 조건을 결정합니다. 액션 항목의 속성을 설정하려면 먼저 다음을 수행하여 액션 에디터를 불러와야 합니다. Object Inspector에서 디스패처의 *Actions* 속성을 선택하고 생략 부호(...)를 클릭합니다. 액션 에디터에서 액션을 선택하면 Object Inspector에서 해당 액션의 속성을 수정할 수 있습니다.

대상 URL

디스패처는 액션 항목의 *PathInfo* 속성과 요청 메시지의 *PathInfo* 속성을 비교합니다. 이 속성 값은 액션 항목이 처리할 요청의 URL의 Path Info 부분이어야 합니다. 예를 들어, 다음과 같은 URL이 있다고 가정합니다.

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

여기서 `/gallery.dll` 부분이 웹 서버 애플리케이션을 나타낸다고 가정하면 다음 부분이 Path Info입니다.

```
/mammals
```

요청을 처리할 때 웹 애플리케이션이 정보를 찾아야 할 위치를 나타내거나 웹 애플리케이션을 논리적인 하위 서비스로 나누기 위해 Path Info를 사용합니다.

요청 메소드 타입

액션 항목의 *MethodType* 속성은 해당 액션 항목이 처리할 수 있는 요청 메시지의 타입을 나타냅니다. 디스패처는 액션 항목의 *MethodType* 속성과 요청 메시지의 *MethodType* 속성을 비교합니다. *MethodType*은 다음 값 중 하나를 가질 수 있습니다.

표 33.1 MethodType 값

값	의미
mtGet	요청은 응답 메시지로 반환될 대상 URI와 연결된 정보를 요구합니다.
mtHead	요청은 *mtGet* 요청을 처리하는 경우처럼 응답의 헤더 속성을 요구하지만 응답의 컨텐트는 생략합니다.
mtPost	요청은 웹 애플리케이션에 포스트될 정보를 보냅니다.
mtPut	요청은 대상 URI와 연결된 리소스를 요청 메시지의 컨텐트로 바꾸도록 요구합니다.
mtAny	*mtGet*, *mtHead*, *mtPut* 및 *mtPost*를 포함하여 모든 요청 메소드 타입을 받아들입니다.

액션 항목 활성화 및 비활성화

각 액션 항목에는 해당 액션 항목을 활성화하거나 비활성화하는 데 사용할 수 있는 *Enabled* 속성이 있습니다. *Enabled*를 **false**로 설정하면 액션 항목이 비활성화되므로 디스패처가 요청을 처리하기 위해 액션 항목을 검색할 때 해당 액션 항목은 고려하지 않습니다.

BeforeDispatch 이벤트 핸들러는 디스패처가 액션 항목을 요청 메시지와 비교하기 전에 액션 항목의 *Enabled* 속성을 변경하여 요청을 처리할 액션 항목을 제어할 수 있습니다.

주의 실행 도중에 액션의 *Enabled* 속성을 변경하면 이후의 요청에 대해 예기치 않은 결과가 발생할 수 있습니다. 웹 서버 애플리케이션이 웹 모듈을 캐싱하는 DLL일 경우 다음 요청에 대해 초기 상태가 재초기화되지 않습니다. 모든 액션 항목이 해당 시작 상태로 올바르게 초기화되게 하려면 *BeforeDispatch* 이벤트를 사용하십시오.

디폴트 액션 항목 선택

액션 항목 중 하나만 디폴트 액션 항목이 될 수 있습니다. 액션 항목의 *Default* 속성을 **true**로 설정하면 디폴트 액션 항목이 선택됩니다. 액션 항목의 *Default* 속성이 **true**로 설정되면 이전 디폴트 액션 항목(있을 경우)의 *Default* 속성은 **false**로 설정됩니다.

디스패처는 요청 처리를 위해 액션 항목 리스트를 검색하여 특정 액션 항목을 선택할 때 디폴트 액션 항목의 이름을 저장합니다. 액션 항목 리스트의 끝에 도달했지만 요청이 완전히 처리되지 않은 경우 디스패처는 디폴트 액션 항목을 실행합니다.

디스패처는 디폴트 액션 항목의 *PathInfo* 또는 *MethodType* 속성을 검사하지 않습니다. 또한 디스패처는 디폴트 액션 항목의 *Enabled* 속성도 검사하지 않습니다. 따라서, *Enabled* 속성을 **false**로 설정하여 대안이 없을 경우에만 디폴트 액션 항목이 호출되도록 해야 합니다.

잘못된 URI 또는 *MethodType*을 나타내는 오류 코드만 반환될 경우에도 디폴트 액션 항목은 발생하는 모든 요청을 처리하도록 준비되어야 합니다. 디폴트 액션 항목이 요청을 처리하지 않을 경우 웹 클라이언트에 응답이 보내지지 않습니다.

주의 실행 도중에 액션의 *Default* 속성을 변경하면 현재 요청에 대해 예기치 않은 결과가 발생할 수 있습니다. 이미 실행된 액션의 *Default* 속성을 **true**로 설정할 경우 해당 액션이 다시 분석되지 않으며 디스패처는 액션 리스트의 끝에 도달했을 때 해당 액션을 실행하지 않습니다.

액션 항목으로 요청 메시지에 응답

액션 항목은 실행 시 웹 서버 애플리케이션의 실제 작업을 수행합니다. 웹 디스패처가 액션 항목을 실행시키면 해당 액션 항목은 다음 두 가지 방법으로 현재 요청 메시지에 응답할 수 있습니다.

- 액션 항목에 *Producer* 속성 값으로 연결된 프로듀서 컴포넌트가 있을 경우 해당 프로듀서는 *Content* 메소드를 사용하여 응답 메시지의 *Content* 속성을 자동으로 채워넣습니다. 컴포넌트 팔레트의 인터넷 페이지에는 응답 메시지 컨텐트의 HTML 페이지의 생성을 도와주는 여러 컨텐트 프로듀서 컴포넌트가 포함되어 있습니다.
- 연결된 프로듀서가 있을 경우 프로듀서가 응답 컨텐트를 할당한 다음, 액션 항목은 *OnAction* 이벤트를 받습니다. *OnAction* 이벤트 핸들러는 HTTP 요청 메시지를 표시하는 *TWebRequest* 객체와 응답 정보를 채우는 *TWebResponse* 객체를 전달 받습니다.

단일 컨텐트 프로듀서가 액션 항목의 컨텐트를 생성할 수 있을 경우 가장 간단한 방법은 컨텐트 프로듀서를 액션 항목의 *Producer* 속성으로 지정하는 것입니다. 그러나 *OnAction* 이벤트 핸들러에서 언제든지 모든 컨텐트 프로듀서에 액세스할 수 있습니다. *OnAction* 이벤트 핸들러는 여러 컨텐트 프로듀서를 사용하고 응답 메시지의 속성을 할당하는 등의 작업을 수행할 수 있도록 더 많은 유연성을 부여합니다.

컨텐트 프로듀서 컴포넌트와 *OnAction* 이벤트 핸들러는 모두 객체 또는 런타임 라이브러리 함수를 사용하여 요청 메시지에 응답할 수 있습니다 . 또한 데이터베이스 액세스 , 계산 수행 , HTML 문서 생성 혹은 선택 등의 작업을 수행할 수 있습니다. 컨텐트 프로듀서 컴포넌트를 사용하여 응답 컨텐트를 생성하는 방법에 대한 자세한 내용은 33-13페이지의 "응답 메시지 컨텐트 생성"을 참조하십시오.

응답 보내기

OnAction 이벤트 핸들러는 *TWebResponse* 객체의 메소드를 사용하여 웹 클라이언트에 응답을 돌려 보낼 수 있습니다. 그러나 클라이언트에 응답을 보내는 액션 항목이 없을 경우 요청에 대한 마지막 액션 항목이 요청이 처리되었다고 알려 주면 웹 서버 애플리케이션은 응답을 보낸 것으로 인식합니다.

여러 액션 항목 사용

단일 액션 항목에서 요청에 응답하거나 여러 액션 항목 간에 작업을 분할할 수 있습니다. 액션 항목이 응답 메시지 작성을 완전히 끝내지 않았을 경우 액션 항목은 *Handled* 매개변수를 **false** 로 설정하여 *OnAction* 이벤트 핸들러에서 이러한 상태를 표시해 주어야 합니다.

요청 메시지에 응답하는 작업이 여러 액션 항목 간에 분할되면 각 액션 항목은 다른 액션 항목이 작업을 계속할 수 있도록 *Handled* 매개변수를 **false**로 설정합니다. 이 때 디폴트 액션 항목에서는 *Handled* 매개변수를 **true**로 설정된 대로 두어야 합니다. 그렇지 않으면 웹 클라이언트에 응답이 보내지지 않습니다.

여러 액션 항목 간에 작업을 분할할 때 디폴트 액션 항목의 *OnAction* 이벤트 핸들러 또는 디스패처의 *AfterDispatch* 이벤트 핸들러는 모든 작업이 수행되었는지 검사하고 그렇지 않을 경우 적절한 오류 코드를 설정해야 합니다.

클라이언트 요청 정보 액세스하기

웹 서버 애플리케이션이 HTTP 요청 메시지를 받으면 클라이언트 요청의 헤더가 *TWebRequest* 객체의 속성들로 읽어들여집니다. NSAPI 및 ISAPI 애플리케이션에서는 *TISAPIRequest* 객체가 요청 메시지를 캡슐화합니다. Console CGI 애플리케이션은 *TCGIRequest* 객체를 사용하고 Windows CGI 애플리케이션은 *TWinCGIRequest* 객체를 사용합니다.

요청 객체의 속성은 읽기 전용입니다. 이러한 속성을 사용하면 클라이언트 요청에서 알아낼 수 있는 모든 정보를 수집할 수 있습니다.

요청 헤더 정보를 포함하는 속성

요청 객체의 속성에는 대부분 HTTP 요청 헤더에서 가져온 요청에 대한 정보가 포함되어 있습니다. 그러나 모든 요청이 이러한 속성 모두에 대해 값을 제공하는 것은 아닙니다. 특히 HTTP 표준이 계속 발전하면서 일부 요청은 요청 객체의 속성으로 표시되지 않는 헤더 필드를 포함할 수도 있습니다 . 요청 객체의 속성으로 표시되지 않는 요청 헤더 필드의 값을 가져오려면 *GetFieldByName* 메소드를 사용하십시오.

대상을 식별하는 속성

요청 메시지의 전체 대상은 *URL* 속성에 의해 지정되며, 일반적으로 프로토콜(HTTP), *Host*(서버 시스템), *ScriptName*(서버 애플리케이션), *PathInfo*(호스트 상의 위치), *Query* 등으로 나뉘어질 수 있는 URL입니다.

이러한 각 부분은 각각의 속성으로 표시됩니다. 프로토콜은 항상 HTTP이고 *Host* 및 *ScriptName*은 웹 서버 애플리케이션을 식별합니다. 디스패처는 액션 항목을 요청 메시지와 비교할 때 *PathInfo* 부분을 사용합니다. 어떤 요청에서는 *Query*를 사용하여 요청된 정보에 대한 세부 사항을 지정합니다. 또한 Query의 값은 *QueryFields* 속성으로 분석되어 저장됩니다.

웹 클라이언트를 설명하는 속성

또한 요청에는 요청이 시작된 장소에 대한 정보를 제공하는 여러 속성이 포함되어 있습니다. 여기에는 보낸 사람의 전자 메일 주소(*From* 속성)에서 메시지가 시작된 URI(*Referer* 또는 *RemoteHost* 속성)에 이르기까지 모든 것이 포함됩니다. 요청에 컨텐트가 포함되어 있지만 해당 컨텐트가 요청과 동일한 URI에서 가져온 것이 아닐 경우 컨텐트의 소스는 *DerivedFrom* 속성에 의해 지정됩니다. 또한 클라이언트의 IP 주소(*RemoteAddr* 속성) 및 요청을 보낸 애플리케이션의 이름과 버전(*UserAgent* 속성)을 알아낼 수 있습니다.

요청 목적을 식별하는 속성

Method 속성은 요청 메시지가 서버 애플리케이션에 요구하는 작업을 설명하는 문자열입니다. HTTP 1.1 표준은 다음 메소드를 정의합니다.

값	메시지가 요청하는 내용
OPTIONS	사용할 수 있는 통신 옵션에 대한 정보
GET	*URL* 속성에 의해 식별되는 정보
HEAD	GET 메시지에서 가져온 헤더 정보, 이 때 응답 컨텐트는 생략됨
POST	*Content* 속성에 포함된 데이터를 포스트할 서버 애플리케이션
PUT	*URL* 속성이 나타내는 리소스를 *Content* 속성에 포함된 데이터로 바꿀 서버 애플리케이션
DELETE	*URL* 속성에 의해 식별되는 리소스를 삭제하거나 숨길 서버 애플리케이션
TRACE	요청 수신을 확인하는 루프백을 보낼 서버 애플리케이션

Method 속성은 웹 클라이언트가 요청하는 서버의 다른 메소드를 나타낼 수 있습니다.

웹 서버 애플리케이션이 *Method*의 모든 가능한 값에 대해 응답을 제공할 필요는 없습니다. 단, HTTP 표준에서는 GET 및 HEAD 요청에 대해 처리할 것을 요구합니다.

MethodType 속성은 *Method*의 값이 GET(mtGet), HEAD(mtHead), POST(mtPost), PUT(mtPut) 또는 다른 문자열(mtAny) 중 어떤 것에 속하는지를 나타냅니다. 디스패처는 *MethodType* 속성의 값을 각 액션 항목의 *MethodType*과 비교합니다.

기대 응답을 설명하는 속성

Accept 속성은 웹 클라이언트가 응답 메시지의 컨텐트로 받아들일 미디어 타입을 나타냅니다. *IfModifiedSince* 속성은 클라이언트가 최근에 변경된 정보만 원하는지 여부를 지정합니다. *Cookie* 속성은 일반적으로 애플리케이션에 의해 미리 추가되며, 응답을 변경시킬 수 있는 상태 정보가 포함되어 있습니다.

컨텐트를 설명하는 속성

대부분의 요청은 정보에 대한 요청이기 때문에 컨텐트를 포함하지 않습니다. 그러나 POST와 같은 일부 요청은 웹 서버 애플리케이션이 사용할 컨텐트를 포함합니다. 컨텐트의 미디어 타입은 *ContentType* 속성으로 지정되고, 컨텐트의 길이는 *ContentLength* 속성으로 지정됩니다. 데이터 압축의 경우처럼 메시지의 컨텐트가 인코딩된 경우에는 이 정보가 *ContentEncoding* 속성에 들어 있습니다. 컨텐트를 생성한 애플리케이션의 이름과 버전 번호는 *ContentVersion* 속성으로 지정됩니다. 또한 *Title* 속성도 컨텐트에 대한 정보를 제공할 수 있습니다.

HTTP 요청 메시지의 컨텐트

헤더 필드 이외에 일부 요청 메시지에는 헤더 필드 이외에도 웹 서버 애플리케이션에서 처리해야 하는 컨텐트 부분이 포함되어 있습니다. 예를 들어, POST 요청은 웹 서버 애플리케이션이 액세스하는 데이터베이스에 추가해야 할 정보를 포함할 수 있습니다.

사전 처리되지 않은 컨텐트 값은 *Content* 속성에 의해 지정됩니다. 컨텐트가 앰퍼샌드(&)로 분리된 필드들로 분석될 수 있을 경우 분석된 버전은 *ContentFields* 속성에 저장됩니다.

HTTP 응답 메시지 생성

웹 서버 애플리케이션은 들어오는 HTTP 요청 메시지에 대한 *TWebRequest* 객체를 만들 때 반환시 보내질 응답 메시지를 나타내는 해당 *TWebResponse* 객체도 함께 만듭니다. NSAPI 및 ISAPI 애플리케이션에서는 *TISAPIResponse* 객체가 응답 메시지를 캡슐화합니다. Console CGI 애플리케이션은 *TCGIResponse* 객체를 사용하고 Windows CGI 애플리케이션은 *TWinCGIResponse* 객체를 사용합니다.

웹 클라이언트 요청에 대한 응답을 생성하는 액션 항목은 응답 객체의 속성을 채웁니다. 이 작업이 오류 코드를 반환하거나 다른 URI에 요청을 리디렉션하는 것만큼 간단한 경우도 있고, 다른 소스에서 정보를 가져와서 완전한 형태로 구성하기 위해 액션 항목의 복잡한 계산이 필요한 경우도 있습니다. 응답 메시지가 단지 요청된 작업이 수행되었다는 사실을 알리는 것에 불과할지라도 대부분의 요청 메시지는 응답을 필요로 합니다.

응답 헤더 채우기

TWebResponse 객체의 대부분의 속성은 웹 클라이언트로 다시 보내지는 HTTP 응답 메시지의 헤더 정보를 나타냅니다. 액션 항목은 *OnAction* 이벤트 핸들러에서 이러한 속성을 설정합니다.

모든 응답 메시지가 헤더 속성 모두에 대해 값을 필요로 하지는 않습니다. 설정해야 할 속성은 요청의 특성과 응답의 상태에 따라 달라집니다.

응답 상태 표시

모든 응답 메시지는 응답 상태를 나타내는 상태 코드를 포함해야 합니다. *StatusCode* 속성을 설정하여 상태 코드를 지정할 수 있습니다. HTTP 표준은 미리 정의된 다수의 표준 상태 코드를 정의합니다. 이외에도 사용되지 않은 값을 사용하여 새로운 상태 코드를 정의할 수 있습니다.

각 상태 코드는 다음과 같이 최상위 자리가 응답의 종류를 나타내는 세 자리 숫자입니다.

- 1xx: 정보(요청을 받았지만 완전히 처리되지 않았음)
- 2xx: 성공(요청을 받아 이해하고 받아들여졌음)
- 3xx: 리디렉션(요청을 완료하기 위해 클라이언트가 추가적인 처리를 해야 함)
- 4xx: 클라이언트 오류(요청을 이해하거나 처리할 수 없음)
- 5xx: 서버 오류(요청은 유효하지만 서버가 요청을 처리할 수 없음)

각 상태 코드에 연결된 문자열은 상태 코드의 의미를 설명합니다. 이 문자열은 *ReasonString* 속성에 지정됩니다. 미리 정의된 상태 코드의 경우 *ReasonString* 속성을 설정할 필요가 없습니다. 새로운 상태 코드를 정의할 경우에는 *ReasonString* 속성도 함께 설정해야 합니다.

클라이언트 액션에 대한 요구 표시

상태 코드의 범위가 300-399 사이인 경우 웹 서버 애플리케이션이 요청을 완료할 수 있으려면 클라이언트가 추가적인 처리를 해야 합니다. 클라이언트를 다른 URI로 리디렉션해야 하거나 요청 처리를 위해 새 URI를 만들었다는 것을 나타내야 한다면 *Location* 속성을 설정합니다. 계속하려면 클라이언트가 암호를 제공해야 할 경우 *WWWAuthenticate* 속성을 설정합니다.

서버 애플리케이션 설명

일부 응답 헤더 속성은 웹 서버 애플리케이션의 기능을 설명합니다. *Allow* 속성은 애플리케이션이 응답할 수 있는 메소드를 나타냅니다. *Server* 속성은 응답을 생성하는 데 사용된 애플리케이션의 이름과 버전 번호를 제공합니다. *Cookies* 속성은 이후의 요청 메시지에 포함되는 서버 애플리케이션에 대한 클라이언트의 사용 상태 정보를 가질 수 있습니다.

컨텐트 설명

몇 가지 속성들은 응답 컨텐트를 설명합니다. *ContentType*은 응답의 미디어 타입을 제공하고 *ContentVersion*은 해당 미디어 타입의 버전 번호입니다. *ContentLength*는 응답의 길이를 제공합니다. 데이터 압축의 경우처럼 컨텐트가 인코드되는 경우 *ContentEncoding* 속성으로 그 사실을 나타냅니다. 컨텐트가 다른 URI에서 가져온 것일 경우 *DerivedFrom* 속성에 표시되어야 합니다. 컨텐트 값에 시간 제한이 있을 경우 *LastModified* 속성과 *Expires* 속성은 해당 값이 아직 유효한지 여부를 나타냅니다. *Title* 속성은 컨텐트에 대한 설명 정보를 제공할 수 있습니다.

응답 컨텐트 설정

일부 요청의 경우 요청 메시지에 대한 응답이 응답의 헤더 속성에 전부 포함됩니다. 그러나 대부분의 경우 액션 항목은 응답 메시지에 컨텐트를 지정합니다. 이런 컨텐트는 파일에 저장된 정적 정보나, 액션 항목 또는 컨텐트 프로듀서에 의해 동적으로 만들어진 정보일 수 있습니다.

Content 속성 또는 *ContentStream* 속성을 사용하여 응답 메시지의 컨텐트를 지정할 수 있습니다.

Content 속성은 AnsiString입니다. AnsiStrings은 텍스트 값에만 한정되는 것이 아니므로 *Content* 속성은 HTML 명령의 문자열, 비트 스트림과 같은 그래픽 컨텐트 또는 다른 MIME 컨텐트 타입이 될 수 있습니다.

응답 메시지의 컨텐트를 스트림에서 읽을 수 있는 경우 *ContentStream* 속성을 사용합니다. 예를 들어, 응답 메시지가 파일의 컨텐트를 보내야 할 경우 *ContentStream* 속성에 대한 *TFileStream* 객체를 사용합니다. *Content* 속성과 마찬가지로 *ContentStream*은 HTML 명령의 문자열이나 다른 MIME 컨텐트 타입을 제공할 수 있습니다. *ContentStream* 속성을 사용할 경우 스트림을 직접 해제하지 마십시오. 웹 응답 객체가 스트림을 자동으로 해제합니다.

참고 *ContentStream* 속성 값이 NULL이 아닌 경우 *Content* 속성은 무시됩니다.

응답 보내기

요청 메시지에 대한 응답에서 더 이상 수행할 작업이 없다는 것이 확실하면 *OnAction* 이벤트 핸들러에서 응답을 직접 보낼 수 있습니다. 응답 객체는 응답을 보내기 위해 *SendResponse* 및 *SendRedirect*의 두 메소드를 제공합니다. *TWebResponse* 객체의 모든 헤더 속성과 지정된 컨텐트를 사용하여 응답을 보내려면 *SendResponse*를 호출합니다. 웹 클라이언트를 다른 URI로 리디렉션하기만 하면 되는 경우에는 *SendRedirect* 메소드가 더 효율적입니다.

이벤트 핸들러가 응답을 보내지 않을 경우 웹 애플리케이션 객체는 디스패처가 종료된 후 응답을 보냅니다. 그러나 응답을 처리한 것으로 알려주는 액션 항목이 없으면 웹 애플리케이션은 응답을 보내지 않고 웹 클라이언트와의 연결을 끊습니다.

응답 메시지 컨텐트 생성

C++Builder에서는 액션 항목이 HTTP 응답 메시지의 컨텐트를 만들 수 있도록 도와줄 여러 가지 객체를 제공합니다. 이 객체들을 사용하면 파일에 저장되거나 직접 웹 클라이언트에 다시 보내지는 HTML 명령의 문자열을 생성할 수 있습니다. *TCustomContentProducer* 또는 TCustomContentProducer의 자손 중 하나로부터 파생하여 새로운 컨텐트 프로듀서를 작성할 수 있습니다.

*TCustomContentProducer*는 특정 MIME 타입의 HTTP 응답 메시지의 컨텐트를 만들기 위한 일반적인 인터페이스를 제공합니다. 그 자손에는 페이지 프로듀서와 테이블 프로듀서가 있습니다.

- 페이지 프로듀서는 사용자 정의 HTML 코드로 바꿀 특수 태그를 HTML 문서에서 검색합니다. 이 내용은 다음 단원에 설명되어 있습니다.
- 테이블 프로듀서는 데이터셋의 데이터에 기초하여 HTML 명령을 만듭니다 . 이 내용은 33-17페이지의 "응답에서 데이터베이스 정보 사용"에 설명되어 있습니다.

페이지 프로듀서 컴포넌트 사용

페이지 프로듀서(*TPageProducer*와 자손)는 HTML 템플릿을 가져와서 특수한 HTML 투명 태그를 사용자 정의 HTML 코드로 대체하여 해당 HTML 템플릿을 변환합니다. HTTP 요청 메시지에 대한 응답을 생성할 때 페이지 프로듀서에 사용될 표준 응답 템플릿들을 저장해둘 수 있습니다. 또한 페이지 프로듀서를 서로 연결하면 HTML 투명 태그를 계속 다듬어서 HTML 문서를 반복적으로 작성할 수 있습니다.

HTML 템플릿

HTML 템플릿은 연속적인 HTML 명령과 HTML 투명 태그들입니다. HTML 투명 태그는 다음과 같은 형태를 갖습니다.

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

꺽쇠 괄호(< 및 >)는 태그의 전체 유효 범위(scope)를 정의합니다. 열린 꺽쇠 괄호(<) 바로 뒤에는 공백 없이 파운드 기호(#)가 오며 꺽쇠 괄호와 구분됩니다. 파운드 기호는 페이지 프로듀서에 대해 문자열을 HTML 투명 태그로 식별합니다. 파운드 기호 바로 뒤에는 공백없이 태그 이름이 오며 파운드 기호와 구분됩니다. 태그 이름은 어떠한 유효한 식별자라도 될 수 있으며 태그가 나타내는 변환 타입을 식별합니다.

HTML 투명 태그는 태그 이름 다음에 선택적으로 수행할 변환에 대한 세부 정보를 지정하는 매개변수를 포함할 수 있습니다. 각 매개변수는 *ParamName*=*Value*의 형태를 가지며 매개변수 이름, 등호 기호(=) 및 값 사이에는 공백이 없습니다. 각 매개변수들은 공백으로 구분합니다.

꺽쇠 괄호(< 및 >)는 #TagName 구조를 인식하지 않는 HTML 브라우저에게 태그가 투명하도록 해줍니다.

새로운 HTML 투명 태그를 만들어 페이지 프로듀서가 처리하는 모든 종류의 정보를 나타낼 수도 있지만, *TTag* 데이터 타입 값에 연결된 여러 가지 태그 이름들이 미리 정의되어 있습니다. 이와 같이 미리 정의된 태그 이름은 응답 메시지에 따라 달라질 수 있는 HTML 명령에 해당합니다. 다음 표는 이러한 태그 이름을 나열한 것입니다.

태그 이름	TTag 값	태그 변환
Link	*tgLink*	하이퍼텍스트 링크. 결과는 <A> 태그로 시작하여 </A> 태그로 끝나는 HTML 시퀀스입니다.
Image	*tgImage*	그래픽 이미지. 결과는 HTML <IMG> 태그입니다.
Table	*tgTable*	HTML 테이블. 결과는 <TABLE> 태그로 시작하여 </TABLE> 태그로 끝나는 HTML 시퀀스입니다.
ImageMap	*tgImageMap*	연결된 핫 존(hot zone)이 있는 그래픽 이미지. 결과는 <MAP> 태그로 시작하여 </MAP> 태그로 끝나는 HTML 시퀀스입니다.
Object	*tgObject*	포함된 ActiveX 객체. 결과는 <OBJECT> 태그로 시작하여 </OBJECT> 태그로 끝나는 HTML 시퀀스입니다.
Embed	*tgEmbed*	Netscape 호환 애드 인 DLL. 결과는 <EMBED> 태그로 시작하여 </EMBED> 태그로 끝나는 HTML 시퀀스입니다.

다른 모든 태그 이름은 *tgCustom*에 연결됩니다. 페이지 프로듀서는 미리 정의된 태그 이름에 대해 기본적인 처리 기능을 제공하지는 않습니다. 미리 정의된 태그 이름은 애플리케이션이 변환 프로세스를 더 일반적인 여러 작업으로 구성할 수 있도록 돕기 위해 제공되는 것입니다.

참고 미리 정의된 태그 이름은 대소문자를 구분하지 않습니다.

HTML 템플릿 지정

페이지 프로듀서에서는 여러 가지 방법으로 HTML 템플릿을 지정할 수 있습니다. *HTMLFile* 속성을 HTML 템플릿이 포함된 파일의 이름으로 설정할 수 있습니다. *HTMLDoc* 속성을 HTML 템플릿이 포함된 *TStrings* 객체로 설정할 수 있습니다. *HTMLFile* 속성 또는 *HTMLDoc* 속성을 사용하여 템플릿을 지정하는 경우, *Content* 메소드를 호출하면 변환된 HTML 명령들을 생성할 수 있습니다.

또한 *ContentFromString* 메소드를 호출하면 매개변수로 전달된 단일 AnsiString인 HTML 템플릿을 직접 변환할 수 있습니다. *ContentFromStream* 메소드를 호출하여 스트림에서 HTML 템플릿을 읽을 수도 있습니다. 예를 들어, 모든 HTML 템플릿을 데이터베이스의 메모 필드에 저장하고 *ContentFromStream* 메소드를 사용하여 변환된 HTML 명령을 얻을 수 있으며, 이 때 *TBlobStream* 객체에서 템플릿을 직접 읽을 수 있습니다.

HTML 투명 태그 변환

페이지 프로듀서는 *Content* 메소드 중 하나가 호출되면 HTML 템플릿을 변환합니다. *Content* 메소드는 HTML 투명 태그를 발견하면 *OnHTMLTag* 이벤트를 발생시킵니다. 발견된 태그의 타입을 결정하고 이를 사용자 정의 컨텐트로 바꾸려면 이벤트 핸들러를 작성해야 합니다.

페이지 프로듀서에 대한 *OnHTMLTag* 이벤트 핸들러를 만들지 않으면 HTML 투명 태그는 빈 문자열로 바뀝니다.

액션 항목에서 페이지 프로듀서 사용

페이지 프로듀서 컴포넌트는 일반적으로 HTML 템플릿이 포함된 파일을 지정하기 위해 *HTMLFile* 속성을 사용합니다. *OnAction* 이벤트 핸들러는 다음과 같이 *Content* 메소드를 호출하여 템플릿을 최종적인 HTML 시퀀스로 변환합니다.

```
void __fastcall WebModule1::MyActionEventHandler(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  PageProducer1->HTMLFile = “Greeting.html”;
  Response->Content = PageProducer1->Content();
}
```

Greeting.html은 다음과 같은 HTML 템플릿이 포함된 파일입니다.

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our Web site.
</BODY>
</HTML>
```

OnHTMLTag 이벤트 핸들러는 다음과 같은 방법으로 실행 도중에 HTML에 있는 사용자 정의 태그(<#UserName>)를 바꿉니다.

```
void __fastcall WebModule1::HTMLTagHandler(TObject *Sender, TTag Tag,
   const AnsiString TagString, TStrings *TagParams, AnsiString
&ReplaceText)
{
  if (CompareText(TagString,"UserName") == 0)
    ReplaceText = ((TPageProducer *)Sender)->Dispatcher->Request-
>Content;
}
```

요청 메시지의 컨텐트가 문자열 *Mr.Ed*일 경우 *Response->Content*의 값은 다음과 같습니다.

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our Web site.
</BODY>
</HTML>
```

참고 이 예제에서는 *OnAction* 이벤트 핸들러를 사용하여 컨텐트 프로듀서를 호출하고 응답 메시지의 컨텐트를 할당합니다. 디자인 타임에 페이지 프로듀서의 *HTMLFile* 속성을 지정하는 경우 *OnAction* 이벤트 핸들러를 작성할 필요가 없습니다. 그럴 경우 *PageProducer1*을 액션 항목의 *Producer* 속성 값으로 지정하기만 하면 위의 *OnAction* 이벤트 핸들러와 동일한 효과를 얻을 수 있습니다.

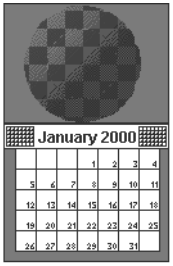
페이지 프로듀서의 연결

OnHTMLTag 이벤트 핸들러에서 가져온 대체 텍스트는 HTTP 응답 메시지에서 사용할 최종적인 HTML 시퀀스가 아니어도 됩니다. 특정 페이지 프로듀서의 출력이 다음 페이지의 입력이 되는 페이지 프로듀서를 여러 개 사용할 수도 있습니다.

페이지 프로듀서를 서로 연결하는 가장 간단한 방법은 각 페이지 프로듀서를 각각의 액션 항목에 연결하는 것이며, 이 때 모든 액션 항목은 동일한 *PathInfo*와 *MethodType*을 가집니다. 첫 번째 액션 항목은 컨텐트 프로듀서에 가져온 웹 응답 메시지의 컨텐트를 설정하지만 이 액션 항목의 *OnAction* 이벤트 핸들러는 메시지를 처리되지 않은 것으로 간주합니다. 다음 액션 항목은 해당 액션 항목에 연결된 프로듀서의 *ContentFromString* 메소드를 사용하여 웹 응답 메시지의 컨텐트 등을 조작합니다. 첫 번째 이후의 액션 항목은 다음과 같이 *OnAction* 이벤트 핸들러를 사용합니다.

```
void __fastcall WebModule1::Action2Action(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  Response->Content = PageProducer2->ContentFromString(Response-
>Content);
}
```

예를 들어, 원하는 페이지의 월과 연도를 지정하는 요청 메시지에 대한 응답으로 달력 페이지를 반환하는 애플리케이션을 가정해 보십시오. 각 달력 페이지에는 우선 그림이 있고 이전 월과 다음 월의 이미지 사이에 월 이름과 연도가 있으며 마지막으로 실제 달력이 나옵니다. 결과 이미지는 다음과 같이 나타납니다.



일반적인 달력 형식은 템플릿 파일에 저장됩니다. 다음과 같이 나타납니다.

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

첫 번째 페이지 프로듀서의 *OnHTMLTag* 이벤트 핸들러는 요청 메시지에서 연도와 월을 알아냅니다. 이벤트 핸들러는 알아낸 정보와 템플릿 파일을 사용하여 다음을 수행합니다.

- <#MonthlyImage>를 <#Image Month=January Year=2000>으로 바꿉니다.
- <#TitleLine>을 <#Calendar Month=December Year=1999 Size=Small> January 2000 <#Calendar Month=February Year=2000 Size=Small>로 바꿉니다.
- <#MainBody>를 <#Calendar Month=January Year=2000 Size=Large>로 바꿉니다.

다음 페이지 프로듀서의 *OnHTMLTag* 이벤트 핸들러는 첫 번째 페이지 프로듀서가 만든 컨텐트를 사용하고 <#Image Month=January Year=2000> 태그를 해당 HTML <IMG> 태그로 바꿉니다. 다른 페이지 프로듀서는 적절한 HTML 테이블을 사용하여 #Calendar 태그를 해결합니다.

응답에서 데이터베이스 정보 사용

HTTP 요청 메시지에 대한 응답에는 데이터베이스에서 가져온 정보가 포함될 수 있습니다. 인터넷 팔레트 페이지에 있는 각각의 특화된 컨텐트 프로듀서는 HTML을 생성하여 데이터베이스의 레코드를 HTML 테이블에 나타낼 수 있습니다.

대체 방법으로 컴포넌트 팔레트의 InternetExpress 페이지에 있는 특수 컴포넌트를 사용하여 멀티 티어 데이터베이스 웹 애플리케이션을 작성할 수 있습니다. 자세한 내용은 29-31페이지의 "InternetExpress를 이용한 웹 애플리케이션 개발"을 참조하십시오.

웹 모듈에 세션 추가

콘솔 CGI 애플리케이션과 Win-CGI 애플리케이션은 모두 HTTP 요청 메시지에 의해 시작됩니다. 이러한 타입의 애플리케이션에서 데 이터베이스 작업을 수행할 경우 각각의 요청 메시지가 자체 애플리케이션 인스턴스를 갖기 때문에 사용자는 디폴트 세션을 사용하여 데이터베이스 연결을 관리할 수 있습니다. 각 애플리케이션 인스턴스는 서로 다른 자체 디폴트 세션을 갖습니다.

그러나 ISAPI 애플리케이션 또는 NSAPI 애플리케이션을 작성할 경우 요청 메시지는 애플리케이션 인스턴스 한 개의 각각의 스레드에서 처리됩니다. 서로 다른 스레드에서 가져온 데이터베이스 연결이 서로 방해하지 못하게 하려면 스레드마다 고유한 세션을 제공해야 합니다.

ISAPI 또는 NSAPI 애플리케이션의 각 요청 메시지는 새 스레드를 생성합니다. 해당 스레드에 대한 웹 모듈은 런타임에 동적으로 생성됩니다. 웹 모듈에 *TSession* 객체를 추가하여 웹 모듈이 포함된 스레드에 대한 데이터베이스 연결을 처리합니다.

런타임에는 스레드마다 각각의 웹 모듈 인스턴스가 생성됩니다. 각 모듈에는 세션 객체가 포함됩니다. 각 요청 메시지를 처리하는 스레드가 다른 스레드의 데이터베이스 연결을 방해하지 않도록 세션마다 각각의 이름이 있어야 합니다. 각 모듈의 세션 객체가 동적으로 고유한 이름을 생성하게 하려면 해당 세션 객체의 *AutoSessionName* 속성을 설정합니다. 각 세션 객체는 고유 이름을 동적으로 생성하고 해당 이름을 참조하도록 모듈에 있는 모든 데이터셋의 *SessionName* 속성을 설정합니다. 이렇게 하면 각 요청 스레드가 서로 다른 요청 메시지를 방해하지 않고 모든 데이터베이스 작업을 처리할 수 있습니다. 세션에 대한 자세한 내용은 24-16페이지의 "데이터베이스 세션 관리"를 참조하십시오.

HTML에 데이터베이스 정보 표시

인터넷 팔레트 페이지의 특화된 컨텐트 프로듀서 컴포넌트는 데이터셋의 레코드에 기초하여 HTML 명령을 제공합니다. 다음과 같은 두 가지 타입의 데이터 인식 컨텐트 프로듀서가 있습니다.

- 데이터셋의 필드 서식을 HTML 문서의 텍스트로 설정하는 데이터셋 페이지 프로듀서
- 데이터셋의 레코드 서식을 HTML 테이블로 설정하는 테이블 프로듀서

데이터셋 페이지 프로듀서 사용

데이터셋 페이지 프로듀서는 다른 페이지 프로듀서 컴포넌트와 같은 방식으로 사용됩니다. 즉, HTML 투명 태그를 포함하는 템플릿을 최종 HTML 표현으로 변환합니다. 그러나 데이터셋 페이지 프로듀서에는 데이터셋에 있는 필드 이름과 일치하는 태그 이름을 가진 태그를 해당 필드의 현재 값으로 변환하는 특수 기능이 있습니다. 페이지 프로듀서의 일반적인 사용 방법에 대한 자세한 내용은 33-14페이지의 "페이지 프로듀서 컴포넌트 사용"을 참조하십시오.

데이터셋 페이지 프로듀서를 사용하려면 웹 모듈에 *TDataSetPageProducer* 컴포넌트를 추가하고 *DataSet* 속성을 HTML 컨텐트에 필드 값이 표시되어야 하는 데이터셋으로 설정합니다. 데이터셋 페이지 프로듀서의 출력을 설명하는 HTML 템플릿을 만듭니다. 그리고 표시하려는 모든 필드 값에 대해 HTML 템플릿에 다음과 같은 형태의 태그를 포함시킵니다.

```
<#FieldName>
```

HTML 템플릿에서 *FieldName*은 값을 표시해야 하는 데이터셋의 필드 이름을 지정합니다.

애플리케이션이 *Content*, *ContentFromString* 또는 *ContentFromStream* 메소드를 호출하면 데이터셋 페이지 프로듀서는 필드를 나타내는 태그를 현재 필드 값으로 대체합니다.

테이블 프로듀서 사용

인터넷 팔레트 페이지에는 HTML 테이블을 만들어 데이터셋 레코드를 나타내는 다음과 같은 두 개의 컴포넌트가 들어 있습니다.

- 데이터셋 필드 서식을 HTML 문서의 텍스트로 설정하는 데이터셋 테이블 프로듀서
- 요청 메시지로부터 매개변수를 설정한 쿼리를 실행하고 결과 데이터셋의 서식을 HTML 테이블로 설정하는 쿼리 테이블 프로듀서

두 테이블 프로듀서 중 하나를 사용하면 테이블 색, 테두리, 구분자 두께 등의 속성을 지정하여 결과 HTML 테이블의 모양을 지정할 수 있습니다. 디자인 타임에 테이블 프로듀서의 속성을 설정하려면 테이블 프로듀서 컴포넌트를 더블 클릭하여 Response Editor 다이얼로그 박스를 표시하십시오.

테이블 어트리뷰트(attribute) 지정

테이블 프로듀서는 *THTMLTableAttributes* 객체를 사용하여 데이터셋의 레코드를 표시하는 HTML 테이블의 시각적 모양을 설명합니다. *THTMLTableAttributes* 객체에는 HTML 문서 내의 테이블 너비와 간격, 배경색, 테두리 두께, 셀 안쪽 여백, 셀 간격 등에 대한 속성이 포함되어 있습니다. 이러한 모든 속성은 테이블 프로듀서가 만든 HTML <TABLE> 태그의 옵션으로 전환됩니다.

디자인 타임에 Object Inspector를 사용하여 이러한 속성을 지정합니다. Object Inspector에서 테이블 프로듀서 객체를 선택하고 *TableAttributes* 속성을 확장하여 *THTMLTableAttributes* 객체의 디스플레이 속성을 액세스합니다.

또한 이러한 속성을 런타임에 프로그램에서 지정할 수도 있습니다.

행 어트리뷰트 지정

테이블 어트리뷰트처럼 데이터를 표시하는 테이블의 행에서 셀의 배경색과 정렬을 지정할 수 있습니다. *RowAttributes* 속성은 *THTMLTableRowAttributes* 객체입니다.

디자인 타임에 Object Inspector 에서 *RowAttributes* 속성을 확장하여 이러한 속성을 지정합니다. 또한 이러한 속성을 런타임에 프로그램에서 지정할 수도 있습니다.

또한 *MaxRows* 속성을 설정하여 HTML 테이블에 표시되는 행의 수를 조절할 수도 있습니다.

열 지정

디자인 타임에 테이블의 데이터셋을 알고 있다면 Columns Editor를 사용하여 열의 필드 연결을 지정하고 어트리뷰트(attribute)를 표시할 수 있습니다. 테이블 프로듀서 컴포넌트를 선택하고 마우스 오른쪽 버튼을 클릭합니다. 컨텍스트 메뉴에서 Columns Editor를 선택합니다. 이 에디터를 사용하면 테이블에서 열을 추가, 삭제 또는 재정렬할 수 있습니다. Columns Editor에서 열을 선택한 후 Object Inspector에서 개별 열의 필드 연결을 설정하고 속성을 표시할 수 있습니다.

HTTP 요청 메시지에서 데이터셋의 이름을 가져오는 경우에는 디자인 타임에 Columns Editor에서 필드를 연결할 수 없습니다. 그러나, 런타임에서는 여전히 해당 *THTMLTableColumn* 객체를 설정하고 *Columns* 속성의 메소드를 사용하여 객체를 테이블에 추가하면 프로그램에서 열을 사용자 정의할 수 있습니다. *Columns* 속성을 설정하지 않으면 테이블 프로듀서는 데이터셋 필드와 일치하는 디폴트 컬럼 셋을 만들고 특별한 표시 특성을 지정하지 않습니다.

HTML 문서에 테이블 포함

테이블 프로듀서의 *Header* 및 *Footer* 속성을 사용하면 데이터셋을 나타내는 HTML 테이블을 더 큰 문서에 포함시킬 수 있습니다. *Header*는 테이블 앞에 오는 모든 내용을 지정하기 위하여, *Footer*는 테이블 뒤에 오는 모든 내용을 지정하기 위하여 사용합니다.

페이지 프로듀서와 같은 다른 컨텐트 프로듀서를 사용하여 *Header* 및 *Footer* 속성의 값을 만들 수도 있습니다.

더 큰 문서에 테이블을 포함할 경우 테이블에 캡션을 추가할 수 있습니다. *Caption* 및 *CaptionAlignment* 속성을 사용하여 테이블에 캡션을 제공합니다.

데이터셋 테이블 프로듀서 설정

*TDataSetTableProducer*는 데이터셋에 대한 HTML 테이블을 만드는 테이블 프로듀서입니다. *TDataSetTableProducer*의 *DataSet* 속성을 설정하여 표시할 레코드가 들어 있는 데이터셋을 지정합니다. 일반적인 데이터베이스 애플리케이션에 있는 대부분의 데이터 인식 객체의 경우와 달리 *DataSource* 속성은 설정하지 않습니다. 이는 *TDataSetTableProducer*가 내부적으로 자신만의 데이터 소스를 생성하기 때문입니다.

웹 애플리케이션이 항상 같은 데이터셋에서 가져온 레코드를 표시하는 경우 디자인 타임에 *DataSet*의 값을 설정할 수 있습니다. 데이터셋이 HTTP 요청 메시지에 있는 정보에 따라 결정되는 경우 런타임에 *DataSet* 속성을 설정해야 합니다.

쿼리 테이블 프로듀서 설정

HTTP 요청 메시지에서 매개변수를 가져온 쿼리의 결과를 표시하기 위해 HTML 테이블을 만들 수 있습니다. 매개변수를 *TQueryTableProducer* 컴포넌트의 *Query* 속성으로 사용하는 *TQuery* 객체를 지정합니다.

요청 메시지가 GET 요청인 경우 HTTP 요청 메시지의 *Query* 필드에서 쿼리의 매개변수를 가져옵니다. 요청 메시지가 POST 요청인 경우에는 요청 메시지의 컨텐트에서 쿼리의 매개변수를 가져옵니다.

*TQueryTableProducer*의 *Content* 메소드를 호출하면 해당 메소드가 요청 객체에서 알아낸 매개변수를 사용하여 쿼리를 실행합니다. 그런 다음 HTML 테이블의 서식을 설정하여 결과 데이터셋에 레코드를 표시합니다.

테이블 프로듀서에서처럼 HTML 테이블의 표시 속성이나 열 연결을 사용자 정의하거나 더 큰 HTML 문서에 테이블을 포함시킬 수 있습니다.

장

34

WebSnap을 사용하여 웹 서버 애플리케이션 생성

WebSnap은 추가적인 컴포넌트, 마법사, 뷰 등으로 Web Broker를 보강하여 복잡한 데이터 웹 페이지를 표시하는 웹 서버 애플리케이션을 쉽게 개발할 수 있게 합니다. WebSnap은 다중 모듈과 서버사이드 스크립트를 지원하여 C++Builder 개발자 및 웹 디자이너 팀이 개발 및 유지 보수 작업을 보다 쉽게 수행할 수 있게 해줍니다.

WebSnap을 사용하면 팀 내의 HTML 디자인 전문가들이 웹 서버를 보다 효과적으로 개발하고 유지 보수할 수 있습니다. WebSnap 개발 프로세스의 최종 제품에는 일련의 스크립트 가능한 HTML 페이지 템플릿들이 포함되어 있습니다. Microsoft FrontPage처럼 포함된 스크립트 태그를 지원하는 HTML 에디터나 간단한 텍스트 에디터를 사용하여 이들 페이지를 변경할 수 있습니다. 필요하면 애플리케이션이 배포된 후에도 템플릿을 변경할 수 있습니다. 프로젝트 소스 코드는 전혀 수정할 필요가 없기 때문에 개발 기간이 단축됩니다. 또한 WebSnap의 다중 모듈 지원 기능을 사용하면 프로젝트의 코딩 단계 동안 애플리케이션을 더 작은 부분들로 분할할 수 있습니다. C++Builder 개발자들은 더욱 독립적으로 작업할 수 있습니다.

디스패처 컴포넌트는 페이지 컨텐트에 대한 요청, HTML 폼 전송 및 동적 이미지 요청을 자동으로 처리합니다. 어댑터라는 WebSnap 컴포넌트는 애플리케이션의 비즈니스 룰에 스크립트 가능한 인터페이스를 정의하는 수단을 제공합니다. 예를 들어, *TDataSetAdapter* 객체를 사용하면 데이터셋 컴포넌트를 스크립트할 수 있습니다. WebSnap 프로듀서 컴포넌트를 사용하여 복잡한 데이터 폼과 테이블을 빠르게 생성하거나 XSL을 사용하여 페이지를 생성할 수 있습니다. 또한, 세션 컴포넌트를 사용하여 엔드 유저를 추적하고 유저 리스트 컴포넌트를 사용하여 사용자 이름, 암호 및 액세스 권한을 액세스할 수 있습니다.

Web Application 마법사를 사용하면 필요한 컴포넌트로 사용자 정의된 애플리케이션을 신속하게 생성할 수 있습니다. Web Page Module 마법사를 사용하여 애플리케이션에서 새 페이지를 정의하는 모듈을 만들 수 있습니다. 또는 Web Data Module 마법사를 사용하여 웹 애플리케이션을 통해 공유되는 컴포넌트의 컨테이너를 만들 수도 있습니다.

페이지 모듈 뷰는 애플리케이션을 실행하지 않고 서버사이드 스크립트 결과를 표시합니다. 생성된 HTML은 Preview 탭을 사용하여 포함된 브라우저에서 보거나 HTML Result 탭을 사용하여 텍스트 형식으로 볼 수 있습니다. HTML Script 탭에는 페이지에 대한 HTML을 생성하는 데 사용되는 서버사이드 스크립트 페이지가 표시됩니다.

이 장의 다음 단원에서는 WebSnap 컴포넌트를 사용하여 웹 서버 애플리케이션을 만드는 방법에 대해 설명합니다.

기본 WebSnap 컴포넌트

WebSnap을 사용하여 웹 서버 애플리케이션을 개발하려면 WebSnap 개발에 사용되는 기본 컴포넌트들을 잘 알고 있어야 합니다. 기본 컴포넌트는 다음과 같은 세 가지 범주로 분류됩니다.

- 애플리케이션을 구성하고 페이지를 정의하는 컴포넌트들이 포함된 웹 모듈
- HTML 페이지와 웹 서버 애플리케이션 사이에 인터페이스를 제공하는 어댑터
- 엔드 유저에게 제공될 HTML 페이지를 만드는 루틴이 포함된 페이지 프로듀서

다음 단원에서는 각 컴포넌트 타입을 보다 자세히 설명합니다.

웹 모듈

웹 모듈은 WebSnap 애플리케이션의 기본적인 구성 단위입니다. 모든 WebSnap 서버 애플리케이션에는 웹 모듈이 하나 이상 있어야 합니다. 필요한 경우 더 많은 웹 모듈을 추가할 수 있습니다. 웹 모듈은 다음과 같은 네 가지 타입이 있습니다.

- 웹 애플리케이션 페이지 모듈(*TWebAppPageModule* 객체)
- 웹 애플리케이션 데이터 모듈(*TWebAppDataModule* 객체)
- 웹 페이지 모듈(*TWebPageModule* 객체)
- 웹 데이터 모듈(*TWebDataModule* 객체)

웹 페이지 모듈과 웹 애플리케이션 페이지 모듈은 웹 페이지에 컨텐트를 제공합니다. 웹 데이터 모듈과 웹 애플리케이션 데이터 모듈은 애플리케이션을 통해 공유되는 컴포넌트의 컨테이너 역할을 하며, 일반적인 데이터 모듈이 보통의 C++ Builder 애플리케이션에서 수행하는 것과 같은 용도로 WebSnap 애플리케이션에서 사용됩니다. 서버 애플리케이션에 원하는 수만큼 웹 페이지 모듈 또는 웹 데이터 모듈을 포함시킬 수 있습니다.

애플리케이션에 필요한 웹 모듈의 수가 궁금할 수도 있습니다. 일부 웹 애플리케이션 모듈 타입은 모든 WebSnap 애플리케이션에 하나만 있으면 됩니다. 그 외의 타입에서는 필요한 만큼 웹 페이지 모듈 또는 웹 데이터 모듈을 추가할 수 있습니다.

웹 페이지 모듈의 경우 페이지 스타일마다 하나씩 사용하는 것이 좋습니다. 기존 페이지 형식을 사용할 수 있는 페이지를 구현하려면 새 웹 페이지 모듈이 필요하지 않을 수도 있습니다. 이 경우 기존 페이지 모듈을 수정하기만 하면 됩니다. 페이지가 기존 모듈과 많이 다를 경우 새 모듈을 만들 수 있습니다. 예를 들어, 온라인 카탈로그 판매를 처리할 서버를 개발한다고 가정합니다. 사용 가능한 제품을 설명하는 모든 페이지는 같은 레이아웃을 사용하는 동일한 기본 정보 타입을 포함할 수 있기 때문에 동일한 웹 페이지 모듈을 공유할 수 있습니다. 그러나, 주문 양식의 형식과 기능은 항목 설명 페이지와 다르기 때문에 주문 양식에는 다른 웹 페이지 모듈이 필요합니다.

규칙은 웹 데이터 모듈마다 다릅니다. 공유 액세스를 단순화하려면 여러 웹 모듈이 공유할 수 있는 컴포넌트를 웹 데이터 모듈에 배치해야 합니다. 또한, 여러 웹 애플리케이션이 사용할 수 있는 컴포넌트를 각각의 웹 데이터 모듈에 배치할 수 있습니다. 이렇게 하면 애플리케이션 사이에서 이러한 항목들을 쉽게 공유할 수 있습니다. 물론, 위의 두 가지 경우가 모두 아니라면 웹 데이터 모듈을 전혀 사용하지 않을 수도 있습니다. 웹 데이터 모듈은 일반적인 데이터 모듈과 똑같은 방법으로 사용하며, 개발자 자신의 판단과 경험에 따라 사용하면 됩니다.

웹 애플리케이션 모듈 타입

웹 애플리케이션 모듈은 웹 애플리케이션에서 비즈니스 룰과 넌비주얼 (nonvisual) 컴포넌트를 중앙 집중식으로 제어합니다. 다음 표에서는 두 가지 타입의 웹 애플리케이션 모듈에 대해 설명합니다.

표 34.1 웹 애플리케이션 모듈 타입

웹애플리케이션 모듈 타입	설명
페이지	컨텐트 페이지를 만듭니다. 페이지 모듈에는 페이지의 컨텐트 생성을 담당하는 페이지 프로듀서가 포함되어 있습니다. 페이지 프로듀서는 HTTP 요청의 pathinfo가 페이지 이름과 일치할 경우 연결된 페이지를 표시합니다. 페이지는 pathinfo가 비어 있을 경우 디폴트 페이지 역할을 할 수 있습니다.
데이터	다중 웹 페이지 모듈에서 사용되는 데이터베이스 컴포넌트처럼 서로 다른 여러 모듈이 공유하는 컴포넌트의 컨테이너로 사용됩니다.

웹 애플리케이션 모듈은 대체적으로 요청 디스패칭, 세션 관리, 사용자 리스트 유지 보수 등과 같은 애플리케이션 기능을 수행하는 컴포넌트의 컨테이너 역할을 합니다. 이미 Web Broker 아키텍처를 잘 알고 있는 사용자는 웹 애플리케이션 모듈이 *TWebApplication* 객체와 비슷하다고 생각하실 것입니다. 또한 웹 애플리케이션 모듈에는 웹 애플리케이션 모듈의 타입에 따라 보통의 웹 페이지 모듈 또는 웹 데이터 모듈의 기능이 있습니다. 사용자의 프로젝트에는 웹 애플리케이션 모듈이 하나만 있을 수 있습니다. 두 개 이상의 웹 애플리케이션 모듈은 결코 필요하지 않습니다. 추가 기능을 제공하려면 보통의 웹 모듈을 서버에 추가하면 됩니다.

서버 애플리케이션의 가장 기본적인 기능을 포함시키려면 웹 애플리케이션 모듈을 사용합니다. 서버에서 홈 페이지를 유지 보수하는 경우 해당 페이지에 대해 추가 웹 페이지 모듈을 만들 필요가 없도록 웹 애플리케이션 모듈을 *TWebAppDataModule* 대신 *TWebAppPageModule* 로 만들 수도 있습니다.

웹 페이지 모듈

각 웹 페이지 모듈에는 연결된 페이지 프로듀서가 하나씩 있습니다. 요청을 받으면 페이지 디스패처는 요청을 분석하고, 해당 페이지 모듈을 호출하여 요청을 처리하고 페이지 컨텐트를 반환합니다.

웹 데이터 모듈처럼 웹 페이지 모듈도 컴포넌트의 컨테이너입니다. 그러나 웹 페이지 모듈은 컨테이너 그 이상입니다. 웹 페이지 모듈은 특별히 웹 페이지를 생성하는 데 사용됩니다.

모든 웹 페이지 모듈에는 작성 중인 페이지를 미리볼 수 있는 Preview라는 에디터 보기가 있습니다. C++Builder에 제공되는 시각적 애플리케이션 개발 환경을 모두 이용할 수 있습니다.

페이지 프로듀서 컴포넌트

웹 페이지 모듈에는 페이지의 컨텐트 생성을 담당하는 페이지 프로듀서 컴포넌트를 식별하는 속성이 있습니다. 페이지 프로듀서에 대한 자세한 내용은 34-6페이지의 "페이지 프로듀서"를 참조하십시오. WebSnap Page Module 마법사는 웹 페이지 모듈을 만들 때 프로듀서를 자동으로 추가합니다. WebSnap 팔레트의 다른 프로듀서에 놓은 다음 나중에 페이지 프로듀서 컴포넌트를 변경할 수 있습니다. 그러나 페이지 모듈에 템플릿 파일이 있을 경우 이 파일의 컨텐트가 대체 프로듀서 컴포넌트와 호환되어야 합니다.

페이지 이름

웹 페이지 모듈에는 HTTP 요청이나 애플리케이션 로직에서 페이지를 참조하는 데 사용될 수 있는 페이지 이름이 있습니다. 웹 페이지 모듈 유닛의 팩터리는 웹 페이지 모듈의 페이지 이름을 지정합니다.

프로듀서 템플릿

대부분의 페이지 프로듀서는 템플릿을 사용합니다. HTML 템플릿에는 일반적으로 투명 태그 또는 서버사이드 스크립트와 혼합되는 일부 정적 HTML이 포함되어 있습니다. 페이지 프로듀서는 컨텐트를 만들 때 투명 태그를 해당 값으로 바꾸고 서버사이드 스크립트를 실행하여 클라이언트 브라우저에 표시되는 HTML을 생성합니다. XSLPageProducer는 예외입니다. XLSPageProducer는 HTML 대신 XSL이 포함되어 있는 XSL 템플릿을 사용합니다. XSL 템플릿은 투명 태그 또는 서버사이드 스크립트를 지원하지 않습니다.

웹 페이지 모듈에는 유닛의 일부로 관리되는 연결된 템플릿 파일이 있을 수 있습니다. 관리되는 템플릿 파일은 Project Manager 에 표시되며 유닛 서비스 파일과 동일한 기본 파일 이름과 위치를 가집니다. 웹 페이지 모듈에 연결된 템플릿 파일이 없을 경우 페이지 프로듀서 컴포넌트의 속성이 템플릿을 지정합니다.

웹 데이터 모듈

표준 데이터 모듈과 마찬가지로 웹 데이터 모듈도 팔레트의 컴포넌트에 대한 컨테이너입니다. 데이터 모듈은 컴포넌트 추가, 제거 및 선택을 위한 디자인 표면을 제공합니다. 웹 데이터 모듈은 웹 데이터 모듈이 구현하는 인터페이스와 유닛의 구조가 표준 데이터 모듈과 다릅니다.

웹 데이터 모듈은 애플리케이션에서 공유되는 컴포넌트의 컨테이너로 사용됩니다. 예를 들어, 데이터 모듈에 데이터셋 컴포넌트를 가져다 놓고 다음과 같은 페이지 모듈에서 데이터셋을 액세스할 수 있습니다.

- 그리드를 표시하는 페이지 모듈
- 입력 형식을 표시하는 페이지 모듈

또한 서로 다른 여러 웹 서버 애플리케이션에서 사용할 수 있는 컴포넌트 집합을 포함시키기 위해 웹 데이터 모듈을 사용할 수 있습니다.

웹 데이터 모듈 유닛의 구조

표준 데이터 모듈에는 데이터 모듈 객체를 액세스하는 데 사용되는 폼 변수가 있습니다. 웹 데이터 모듈은 이 변수를 웹 데이터 모듈 유닛에 정의되어 있으며 웹 데이터 모듈과 이름이 같은 함수로 바꿉니다. 함수의 용도는 대체하는 변수의 용도와 같습니다. WebSnap 애플리케이션은 멀티 스레드가 될 수 있으며 특정 모듈에 대한 다중 인스턴스를 가지고 여러 요청에 동시에 처리할 수 있습니다. 따라서 올바른 인스턴스를 반환하기 위해 함수를 사용합니다.

또한 웹 데이터 모듈 유닛은 팩토리를 등록하여 WebSnap 애플리케이션이 모듈을 관리하는 방법을 지정합니다. 예를 들어, 플래그들은 모듈을 캐싱하여 다른 요청에 재사용할지, 아니면 요청을 서비스한 후 모듈을 소멸시킬지를 나타냅니다.

어댑터

어댑터는 서버 애플리케이션에 대한 스크립트 인터페이스를 정의합니다. 어댑터를 통해 페이지에 스크립트 언어를 삽입할 수 있으며 스크립트 코드에서 어댑터를 호출하여 정보를 알아낼 수 있습니다. 예를 들어, 어댑터를 사용하여 HTML 페이지에 표시할 데이터 필드를 정의할 수 있습니다. 그런 다음 스크립트된 HTML 페이지에 해당 데이터 필드의 값을 검색하는 스크립트 문과 HTML 컨텐트를 포함시킬 수 있습니다. 이것은 Web Broker 애플리케이션에 사용되는 투명 태그와 비슷합니다. 또한 어댑터는 명령을 실행하는 액션을 지원합니다. 예를 들어, 하이퍼링크를 클릭하거나 HTML 폼을 전송하여 어댑터 액션을 실행시킬 수 있습니다.

어댑터는 HTML 페이지를 동적으로 만드는 작업을 쉽게 해줍니다. 애플리케이션에서 어댑터를 사용하여 조건부 로직 및 루핑을 지원하는 객체 지향 스크립트를 포함시킬 수 있습니다. 어댑터와 서버사이드 스크립트가 없다면 C++ 이벤트 핸들러에서 HTML 생성 로직을 코딩해야 합니다. 어댑터를 사용하면 개발 기간을 상당히 단축할 수 있습니다.

스크립트에 대한 자세한 내용은 34-31페이지의 "WebSnap의 서버사이드 스크립트" 및 34-34 페이지의 "요청 및 응답 디스패칭"을 참조하십시오.

필드, 액션, 오류 및 레코드의 네 가지 타입의 어댑터 컴포넌트들이 페이지 컨텐트를 생성하기 위해 사용될 수 있습니다.

필드

필드는 애플리케이션에서 데이터를 알아내고 웹 페이지에 컨텐트를 표시하기 위해 페이지 프로듀서가 사용하는 컴포넌트입니다. 또한 필드를 사용하여 이미지를 얻어낼 수도 있습니다. 이 경우 필드에는 웹 페이지에 기록된 이미지의 주소가 반환됩니다. 페이지에 컨텐트가 표시되면 어댑터를 호출하여 필드 컴포넌트에서 실제 이미지를 얻어오라는 요청을 웹 서버 애플리케이션에 보냅니다.

액션

액션은 어댑터를 대신하여 명령을 실행하는 컴포넌트입니다. 페이지 프로듀서가 페이지를 생성하면 스크립트 언어가 어댑터 액션 컴포넌트를 호출하여 명령을 실행하는 데 필요한 매개변수와 함께 액션 이름을 반환합니다. 예를 들어, HTML 폼에서 버튼을 클릭하여 테이블의 한 행을 삭제한다고 가정합니다. 이렇게 하면 HTTP 요청에 해당 버튼과 연결된 액션 이름과 행 번호를 나타내는 매개변수가 반환됩니다. 어댑터 디스패처는 명명된 액션 컴포넌트를 찾아서 행 번호를 액션에 매개변수로 전달합니다.

오류

어댑터는 액션을 실행하는 동안 발생한 오류 리스트를 유지합니다. 페이지 프로듀서는 이 오류 리스트를 읽어 애플리케이션이 엔드 유저에게 반환하는 웹 페이지에 표시할 수 있습니다.

레코드

*TDataSetAdapter*처럼 다중 레코드를 표시하는 어댑터 컴포넌트도 있습니다. 이 어댑터는 여러 레코드에서 반복할 수 있는 스크립트 인터페이스를 제공합니다. 일부 어댑터는 페이징을 지원하고 현재 페이지의 레코드에서만 반복됩니다.

페이지 프로듀서

페이지 프로듀서는 웹 페이지 모듈 대신 컨텐트를 생성합니다. 페이지 프로듀서에는 다음과 같은 기능이 제공됩니다.

- HTML 컨텐트를 생성합니다.
- HTMLFile 속성을 사용하여 외부 파일을 참조하거나 HTMLDoc 속성을 사용하여 내부 문자열을 참조할 수 있습니다.
- 프로듀서를 웹 페이지 모듈과 함께 사용할 경우 템플릿은 유닛과 연결된 파일이 될 수 있습니다.
- 프로듀서는 투명 태그 또는 액티브 스크립트를 사용하여 템플릿에 삽입할 수 있는 HTML을 동적으로 생성합니다. 투명 태그는 WebBroker 애플리케이션과 같은 방법으로 사용될 수 있습니다. 투명 태그 사용에 대한 자세한 내용은 33-15페이지의 "HTML 투명 태그 변환"을 참조하십시오. 액티브 스크립트 지원을 통해 HTML 페이지에 JScript 또는 VBScript를 포함시킬 수 있습니다.

페이지 프로듀서 사용을 위한 일반적인 WebSnap 방법은 다음과 같습니다. 웹 페이지 모듈을 만들려면 Web Page Module 마법사에서 페이지 프로듀서 타입을 선택해야 합니다. 여러 가지 방법이 있지만 대부분의 WebSnap 개발자들은 어댑터 페이지 프로듀서인 *TAdapterPageProducer*를 사용하여 새로운 페이지 프로토타입을 만듭니다. 어댑터 페이지 프로듀서에서 표준 컴포넌트 모델과 비슷한 과정으로 프로토타입 웹 페이지를 만들 수 있습니다. 다음으로 어댑터 페이지 프로듀서에 폼(어댑터 폼) 타입을 추가합니다. 필요한 경우 어댑터 폼에 어댑터 그리드와 같은 어댑터 컴포넌트를 추가할 수 있습니다. 어댑터 페이지 프로듀서를 사용하면 사용자 인터페이스를 구성하는 일반적인 C++Builder 방법과 비슷한 방법으로 웹 페이지를 만들 수 있습니다.

어댑터 페이지 프로듀서에서 일반적인 페이지 프로듀서로 전환하는 것이 더 좋은 경우도 있습니다. 예를 들어, 어댑터 페이지 프로듀서의 기능 중 일부는 런타임에 페이지 템플릿에 스크립트를 동적으로 생성하는 것입니다. 서버 최적화를 위해 정적인 스크립트를 사용할 수 있습니다. 또한 스크립트 사용 경험이 있는 사용자는 스크립트를 직접 변경할 수 있습니다. 이 경우 동적 스크립트와 정적 스크립트 사이의 충돌을 방지하려면 일반적인 페이지 프로듀서를 사용해야 합니다. 일반적인 페이지 프로듀서로 변경하는 방법에 대한 자세한 내용은 34-24페이지의 "고급 HTML 디자인"을 참조하십시오.

또한 프로듀서를 웹 디스패처 액션 항목과 연결하여 페이지 프로듀서를 Web Broker 애플리케이션에서와 같은 방법으로 사용할 수 있습니다. 웹 페이지 모듈을 사용하면 다음과 같은 장점이 있습니다.

- 애플리케이션을 실행하지 않고도 페이지의 레이아웃을 미리 볼 수 있습니다.
- 페이지 디스패처가 페이지 프로듀서를 자동으로 호출할 수 있도록 페이지 이름을 모듈에 연결할 수 있습니다.

WebSnap을 사용하여 웹 서버 애플리케이션 생성

WebSnap의 소스 코드를 살펴보면 WebSnap이 수 백개의 객체로 구성된다는 사실을 알게 될 것입니다. 실제로 WebSnap에는 많은 시간을 들여 학습해야 아키텍처를 완벽하게 이해할 수 있을 만큼의 풍부한 객체와 기능이 있습니다. 다행히도 WebSnap 시스템을 완벽하게 알고 있지 않더라도 서버 애플리케이션을 개발할 수 있습니다.

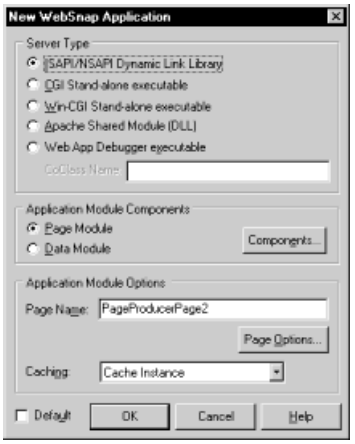
이 단원에서는 새 웹 서버 애플리케이션을 만들어 WebSnap에서 작업하는 방법에 대해 자세히 배웁니다.

다음과 같은 방법으로 WebSnap 아키텍처를 사용하여 새 웹 서버 애플리케이션을 만듭니다.

1. File|New|Other를 선택합니다.
2. New Items 다이얼로그 박스에서 WebSnap 탭을 선택하고 WebSnap Application을 선택합니다.

   그림 34.1에 표시된 것처럼 다이얼로그 박스가 나타납니다.
3. 올바른 서버 타입을 지정합니다.
4. Components 버튼을 사용하여 애플리케이션 모듈 컴포넌트를 지정합니다.
5. Page Options 버튼을 사용하여 애플리케이션 모듈 옵션을 선택합니다.

그림 34.1 New WebSnap 애플리케이션 다이얼로그 박스



서버 타입 선택

웹 서버의 애플리케이션 타입에 따라 다음 웹 서버 애플리케이션 타입 중 하나를 선택합니다.

표 34.2 웹 서버 애플리케이션 타입

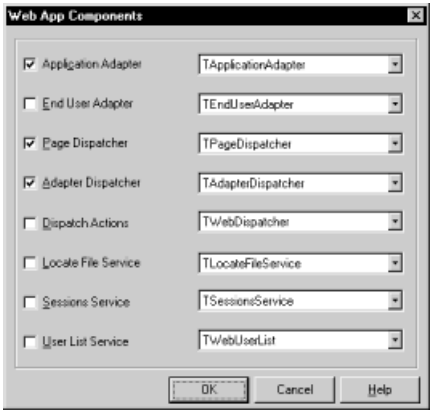
서버 타입	설명
ISAPI 및 NSAPI	프로젝트를 해당 웹 서버에서 요구하는 엑스포트된 메소드가 있는 DLL로 설정합니다.
Apache	프로젝트를 Apache 웹 서버에서 요구하는 엑스포트된 메소드가 있는 DLL로 설정합니다.
CGI 독립 실행형	프로젝트를 CGI(Common Gateway Interface) 표준에 따르는 콘솔 애플리케이션으로 설정합니다.
Win-CGI 독립 실행형	프로젝트를 Windows에 채택된 CGI 표준의 변형인 Windows CGI 애플리케이션으로 설정합니다.
웹 애플리케이션 디버거 실행 파일	웹 서버 애플리케이션 개발 및 테스트 환경을 만듭니다. 이 애플리케이션 타입은 배포용이 아닙니다.

애플리케이션 모듈 컴포넌트 지정

애플리케이션 컴포넌트는 웹 애플리케이션의 기능을 제공합니다. 예를 들어, 어댑터 디스패처 컴포넌트를 포함시키면 HTML 폼 전송이 자동으로 처리되고 동적으로 생성된 이미지가 자동으로 반환됩니다. 페이지 디스패처를 포함시키면 HTTP 요청 Path Info에 페이지 이름이 있을 경우 해당 페이지의 컨텐트가 자동으로 표시됩니다.

새 WebSnap 애플리케이션 다이얼로그 박스(그림 34.1 참조)에서 Components 버튼을 선택하면 하나 이상의 웹 애플리케이션 모듈 컴포넌트를 선택할 수 있는 다른 다이얼로그 박스가 표시됩니다 . 이 다이얼로그 박스는 Web App Components 다이얼로그 박스이며 그 모양은 그림 34.2에 표시된 것과 같습니다.

그림 34.2 Web App Components 다이얼로그 박스



다음 표에는 사용 가능한 컴포넌트에 대한 간략한 설명이 있습니다.

표 34.3 웹 애플리케이션 컴포넌트

컴포넌트 타입	설명
Application Adapter	타이틀 등의 애플리케이션 정보가 들어 있습니다. 디폴트 타입은 *TApplicationAdapter*입니다.
End User Adapter	사용자 이름, 액세스 권한, 로그인 여부 등과 같은 사용자 정보가 들어 있습니다. 디폴트 타입은 *TEndUserAdapter*입니다. *TEndUserSessionAdapter*를 선택할 수도 있습니다.
Page Dispatcher	HTTP 요청 Path Info를 조사하고 해당 페이지 모듈을 호출하여 페이지의 컨텐트를 반환합니다. 디폴트 타입은 *TPageDispatcher*입니다.
Adapter Dispatcher	HTML 폼 전송을 자동으로 처리하고 어댑터 액션과 필드 컴포넌트를 호출하여 동적 이미지를 요청합니다. 디폴트 타입은 *TAdapterDispatcher*입니다.
Dispatch Actions	pathinfo와 메소드 타입을 기초로 요청을 처리할 액션 항목의 컬렉션을 정의할 수 있게 해줍니다. 액션 항목은 사용자 정의 이벤트를 호출하거나 페이지 프로듀서 컴포넌트의 컨텐트를 요청합니다. 디폴트 타입은 *TWebDispatcher*입니다.
Locate File Service	웹 애플리케이션이 실행 중인 동안 템플릿 파일과 스크립트 포함 파일의 로딩을 제어합니다. 디폴트 타입은 *TLocateFileService*입니다.
Sessions Service	짧은 기간 동안 필요한 엔드 유저 정보를 저장합니다. 예를 들어, 세션을 사용하여 로그인한 사용자를 추적하고 일정 기간 동안 비활성 상태인 사용자를 자동으로 로그아웃시킬 수 있습니다. 디폴트 타입은 *TSessionsService*입니다.
User List Service	승인된 사용자, 사용자 암호 및 액세스 권한을 추적합니다. 디폴트 타입은 *TWebUserList*입니다.

위에 있는 각각의 컴포넌트에 대해 나열된 컴포넌트 타입은 C++Builder에 포함된 디폴트 타입입니다. 사용자는 새 컴포넌트 타입을 만들거나 서드파티의 컴포넌트 타입을 사용할 수 있습니다.

웹 애플리케이션 모듈 옵션 선택

선택된 애플리케이션 모듈 타입이 페이지 모듈일 경우 New WebSnap Application 다이얼로그 박스의 Page Name 필드에 이름을 입력하여 페이지와 이름을 연결할 수 있습니다. 이 모듈 인스턴스는 런타임 시 캐시에 저장되거나 요청이 서비스된 후 메모리에서 제거될 수 있습니다. Caching 필드에서 옵션 중 하나를 선택합니다. Page Options 버튼을 선택하여 페이지 모듈 옵션을 추가로 선택할 수 있습니다. 다음과 같은 범주가 제공되는 Application Module Page Options 다이얼로그 박스가 표시됩니다.

- Producer: 페이지의 프로듀서 타입을 *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, *XSLPageProducer* 중 하나로 설정할 수 있습니다. 선택한 페이지 프로듀서가 스크립트를 지원하면 Script Engine 드롭다운 리스트를 사용하여 페이지 스크립트에 사용되는 언어를 선택합니다.

참고 *AdapterPageProducer*는 JScript만 지원합니다.

- HTML: 선택된 프로듀서가 HTML 템플릿을 사용할 경우 이 그룹이 표시됩니다.
- XSL: 선택된 프로듀서가 *TXSLPageProducer*와 같은 XSL 템플릿을 사용할 경우 이 그룹이 표시됩니다.
- New File: 템플릿 파일을 만들어 유닛의 일부로 관리하려면 New File 에 선택 표시를 합니다. 관리되는 템플릿 파일은 Project Manager에 표시되고 유닛 소스 파일과 동일한 파일 이름과 위치를 가집니다. 프로듀서 컴포넌트의 속성 (일반적으로 *HTMLDoc* 또는 *HTMLFile* 속성)을 사용하려면 New File을 선택 해제합니다.
- Template: New File을 선택한 경우 Template 드롭다운 리스트에서 템플릿 파일에 대한 디폴트 컨텐트를 선택합니다. 표준 템플릿은 애플리케이션 제목, 페이지 제목, published로 선언된 페이지에 대한 하이퍼링크 등을 표시합니다. 템플릿이 비어 있으면 빈 페이지가 만들어집니다.
- Page: 페이지 모듈의 페이지 이름과 제목을 입력합니다. 페이지 이름은 HTTP 요청이나 애플리케이션의 로직에서 페이지를 참조하는 데 사용되는 이름이고, 제목은 페이지가 브라우저에 표시될 때 엔드 유저에게 표시되는 이름입니다.
- Published: 페이지 이름이 요청 메시지에 있는 Path Info와 일치할 경우 페이지가 HTTP 요청에 자동으로 응답하게 하려면 Published에 선택 표시를 합니다.
- Login Required: 사용자가 로그인한 후 페이지를 액세스할 수 있게 하려면 Login Required 에 선택 표시를 합니다.

지금까지는 WebSnap 서버 애플리케이션 생성을 시작하는 방법을 배웠습니다. 다음 단원의 WebSnap 자습서에는 보다 완벽한 애플리케이션을 개발하는 방법이 설명되어 있습니다.

WebSnap 자습서

이 자습서에는 WebSnap 애플리케이션을 구축하는 단계별 지침이 들어 있습니다. WebSnap 애플리케이션은 WebSnap HTML 컴포넌트를 사용하여 테이블의 컨텐트를 편집하는 애플리케이션을 구축하는 방법을 보여 줍니다. 처음부터 끝까지 지침을 따르십시오. 잠시 중단하기를 원한다면 언제든지 File|Save All을 사용하여 프로젝트를 저장할 수 있습니다.

새 애플리케이션 생성

이 항목에는 Country Tutorial이라는 새 WebSnap 애플리케이션을 만드는 방법이 설명되어 있습니다. 또한 웹 상의 사용자에게 여러 국가에 대한 정보를 보여주는 표가 들어 있습니다. 사용자는 국가를 추가 및 삭제하거나 기존 국가에 대한 정보를 편집할 수 있습니다.

단계 1. WebSnap 애플리케이션 마법사 시작

1 C++Builder를 실행하고 File|New|Other를 선택합니다.

2 New Items 다이얼로그 박스에서 WebSnap 탭을 선택하고 WebSnap Application을 선택합니다.

3 New WebSnap Application 다이얼로그 박스에서 다음을 수행합니다.

- Web App Debugger 실행 파일을 선택합니다.
- CoClass Name 편집 컨트롤에 CountryTutorial을 입력합니다.
- Page Module을 컴포넌트 타입으로 선택합니다.
- Page Name 필드에 Home을 입력합니다.

4 OK를 클릭합니다.

단계 2. 생성된 파일 및 프로젝트 저장

다음과 같은 방법으로 유닛 파일 및 프로젝트를 저장합니다.

1 File|Save All을 선택합니다.

2 Save 다이얼로그 박스에서 Country Tutorial 프로젝트의 모든 파일을 저장할 수 있는 적절한 디렉토리로 변경합니다.

3 File name 필드에 HomeU.cpp를 입력하고 Save를 클릭합니다.

4 File name 필드에 CountryU.cpp를 입력하고 Save를 클릭합니다.

5 File name 필드에 CountryTutorial.bpr을 입력하고 Save를 클릭합니다.

참고 애플리케이션은 실행 파일과 같은 디렉토리에서 HomeU.html 파일을 찾기 때문에 유닛과 프로젝트를 같은 디렉토리에 저장하십시오.

단계 3. 애플리케이션 제목 지정

애플리케이션 제목은 엔드 유저에게 표시되는 이름입니다. 다음과 같은 방법으로 애플리케이션 제목을 지정합니다.

1 View | Project Manager를 선택합니다.

2 Project Manager에서 CountryTutorial.exe를 확장하고 HomeU 엔트리를 더블 클릭합니다.

3 Object Inspector 윈도우의 맨 위쪽 줄에 있는 풀다운 리스트에서 ApplicationAdapter를 선택합니다.

4 Properties 탭에서 ApplicationTitle 속성에 대해 Country Tutorial을 입력합니다.

5 에디터 윈도우에서 Preview 탭을 클릭합니다. 페이지 이름이 Home인 페이지의 맨 위쪽에 애플리케이션 제목이 표시됩니다.

이 페이지는 매우 기본적인 페이지입니다. HomeU.html 파일을 편집하거나 HomeU.html 에디터 탭이나 외부 에디터를 사용하여 페이지를 향상시킬 수 있습니다. 페이지 템플릿 편집 방법에 대한 자세한 내용은 34-24페이지의 "고급 HTML 디자인"을 참조하십시오.

CountryTable 페이지 생성

웹 페이지 모듈은 published로 선언된 페이지를 정의하고 데이터 컴포넌트의 컨테이너 역할을 합니다. 웹 페이지 모듈은 엔드 유저에게 웹 페이지를 반환해야 할 때마다 포함된 데이터 컴포넌트에서 필요한 정보를 추출하고 해당 정보를 사용하여 페이지 작성을 도와 줍니다.

이제 WebSnap 애플리케이션에 새 페이지 모듈을 추가합니다. 페이지 모듈은 Country Tutorial 애플리케이션에 표시할 수 있는 두 번째 페이지를 추가합니다. 첫 번째 페이지인 Home은 애플리케이션을 만들 때 정의되었습니다. 두 번째 페이지인 CountryTable에는 국가 정보에 대한 테이블이 표시됩니다.

단계 1. 새 웹 페이지 모듈 추가

다음과 같은 방법으로 새 페이지 모듈을 추가합니다.

1 File | New | Other를 선택합니다.

2 New Items 다이얼로그 박스에서 WebSnap 탭을 선택하고 WebSnap Page Module을 선택한 다음 OK를 클릭합니다.

3 다이얼로그 박스에서 Producer Type을 AdapterPageProducer로 설정합니다.

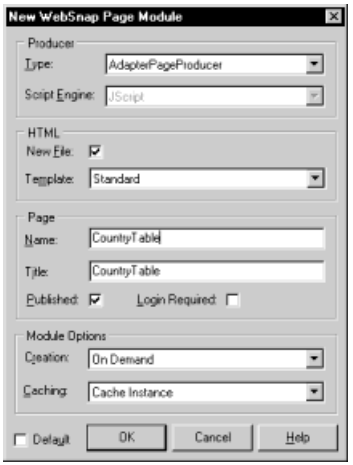
4 Page Name 필드에 CountryTable을 입력합니다. 이 때 Title도 함께 변경됩니다.

5 나머지 필드는 기본값을 그대로 둡니다.

그림 34.3에 표시된 것과 같은 다이얼로그 박스가 나타나야 합니다.

6 OK를 클릭합니다.

그림 34.3 CountryTable 페이지의 New WebSnap Page Module 다이얼로그 박스



이제 IDE에 CountryTable 모듈이 나타날 것입니다. 모듈을 저장한 후 CountryTable 모듈에 새 컴포넌트를 추가합니다.

단계 2. 새 웹 페이지 모듈 저장

프로젝트 파일의 디렉토리에 유닛을 저장합니다. 애플리케이션은 실행된 후 실행 파일과 같은 디렉토리에서 CountryTableU.html 파일을 찾습니다.

1 File|Save를 선택합니다.

2 File name 필드에 CountryTableU.cpp를 입력하고 Save를 클릭합니다.

CountryTable 모듈에 데이터 컴포넌트 추가

TTable 및 *TDataSetAdapter*는 데이터 인식 컴포넌트이며 데이터를 액세스합니다. *TTable*은 HTML 테이블에 데이터를 제공합니다. *TDataSetAdapter*를 사용하여 서버사이드 스크립트에서 *TTable* 컴포넌트를 액세스할 수 있습니다. 여기서 애플리케이션에 이 데이터 인식 컴포넌트를 추가합니다.

아래 단계 1과 단계 2에서는 데이터베이스 프로그래밍에 대한 기본적인 지식이 있는 것으로 가정하지만 이 자습서를 따라오는 데 이러한 지식이 필요하지는 않습니다. WebSnap은 어댑터 컴포넌트를 통해 데이터베이스 컴포넌트에 대한 인터페이스 역할만 합니다. 데이터베이스 프로그래밍에 대한 자세한 내용은 이 안내서의 2부를 참조하십시오.

단계 1. 데이터 인식 컴포넌트 추가

1 View|Project Manager를 선택합니다.

2 Project Manager에서 CountryTutorial.exe를 확장하고 CountryTableU 엔트리를 더블 클릭합니다.

3 View|Object TreeView를 선택합니다. Object TreeView 윈도우가 활성화됩니다.

4 컴포넌트 팔레트에서 BDE 탭을 선택합니다.

5 Table 컴포넌트를 선택하여 CountryTable 웹 페이지 모듈에 추가합니다.

6 Session 컴포넌트를 선택하여 CountryTable 웹 페이지 모듈에 추가합니다. 멀티 스레드 애플리케이션에서 BDE 컴포넌트(*TTable*)를 사용하기 때문에 Session 컴포넌트가 필요합니다.

7 웹 페이지 모듈 또는 Object TreeView에서 디폴트로 Session1로 이름이 붙여진 Session 컴포넌트를 선택합니다. 이렇게 하면 Object Inspector에 Session 컴포넌트 값이 표시됩니다.

8 Object Inspector에서 *AutoSessionName* 속성을 *true*로 설정합니다.

9 웹 페이지 모듈 또는 Object TreeView에서 Table 컴포넌트를 선택합니다. 이렇게 하면 Object Inspector에 Table 컴포넌트 값이 표시됩니다.

10 Object Inspector에서 다음 속성을 변경합니다.

- *DatabaseName* 속성을 DBDEMOS로 설정합니다.
- *Name* 속성에 Country를 입력합니다.
- *TableName* 속성을 country.db로 설정합니다.
- *Active* 속성을 *true*로 설정합니다.

단계 2. 키 필드 지정

키 필드는 테이블에서 레코드를 식별하는 데 사용됩니다. 또한 애플리케이션에 편집 페이지를 추가할 때 중요합니다. 다음과 같은 방법으로 키 필드를 지정합니다.

1 Object TreeView에서 Session 및 DBDemos 노드를 확장한 다음 country.db 노드를 선택합니다. 이 노드는 Country Table 컴포넌트입니다.

2 country.db 노드를 마우스 오른쪽 버튼으로 클릭하고 Fields Editor를 선택합니다.

3 CountryTable->Country Editor 윈도우를 마우스 오른쪽 버튼으로 클릭하고 Add All Fields를 선택합니다.

4 추가된 필드 리스트에서 Name 필드를 선택합니다.

5 Object Inspector에서 *ProviderFlags* 속성을 확장합니다.

6 *pfInKey* 속성을 *true*로 설정합니다.

단계 3. 어댑터 컴포넌트 추가

데이터베이스 컴포넌트 추가 작업을 완료했습니다. 이제 서버사이드 스크립트를 통해 *TTable*에 데이터를 노출하려면 데이터셋 어댑터(*TDataSetAdapter*) 컴포넌트를 포함시켜야 합니다. 다음과 같은 방법으로 데이터셋 어댑터를 추가합니다.

1 컴포넌트 팔레트의 WebSnap 탭에서 DataSetAdapter 컴포넌트를 선택합니다. 선택한 DataSetAdapter 컴포넌트를 CountryTable Web 모듈에 추가합니다.

2 Object Inspector에서 다음 속성을 변경합니다.

- *DataSet* 속성을 Country로 설정합니다.
- *Name* 속성을 Adapter로 설정합니다.

작업이 완료되면 CountryTable 웹 페이지 모듈이 그림 34.4와 비슷한 모양으로 표시될 것입니다.

그림 34.4 CountryTable 웹 페이지 모듈



모듈에 있는 요소들은 보이지 않으므로 모듈에서 요소의 위치는 중요하지 않습니다. 중요한 것은 그림에 표시된 것과 같은 컴포넌트들이 모듈에 들어 있는가 하는 것입니다.

데이터를 표시할 그리드 생성

단계 1. 그리드 추가

이제 다음과 같은 방법으로 그리드를 추가하여 국가 테이블 데이터베이스에서 데이터를 표시합니다.

1. View | Project Manager를 선택합니다.
2. Project Manager에서 CountryTutorial.exe를 확장하고 CountryTableU 엔트리를 더블 클릭합니다.
3. View | Object TreeView를 선택하거나 Object TreeView를 클릭합니다.
4. AdapterPageProducer 컴포넌트를 확장합니다. 이 컴포넌트는 서버사이드 스크립트를 생성하여 HTML 테이블을 빠르게 생성합니다.
5. WebPageItems 엔트리를 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
6. Add Web Component 다이얼로그 박스에서 AdapterForm을 선택한 다음 OK를 클릭합니다. Object TreeView에 AdapterForm1 컴포넌트가 나타납니다.
7. AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
8. Add Web Component 윈도우에서 AdapterGrid를 선택한 다음 OK를 클릭합니다. Object TreeView에 AdapterGrid1 컴포넌트가 나타납니다.
9. Object Inspector 윈도우에서 Adapter 속성을 Adapter로 설정합니다.

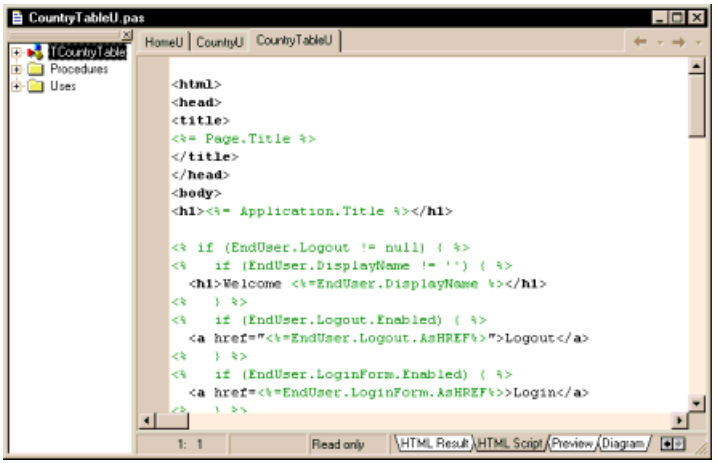
그림 34.5 CountryTable Preview 탭



페이지를 미리 보려면 코드 에디터의 맨 위에 있는 CountryTableU 탭을 선택한 다음 하단에 있는 Preview 탭을 선택합니다. Preview 탭이 표시되지 않으면 하단에 있는 오른쪽 화살표를 사용하여 탭을 스크롤합니다. 그림 34.5와 비슷한 미리 보기가 나타날 것입니다.

Preview 탭에는 웹 브라우저에 표시되는 것과 같은 모양의 최종 정적 HTML 페이지가 표시됩니다. 이 페이지는 스크립트를 포함하는 동적 HTML 페이지로부터 생성됩니다. 스크립트 명령의 텍스트를 보려면 에디터 윈도우의 하단에 있는 HTML Script 탭을 선택합니다(그림 34.6 참조).

그림 34.6 CountryTable HTML Script 탭



HTML Script 탭에는 HTML과 스크립트가 함께 표시됩니다. 에디터에서 HTML과 스크립트의 차이는 글꼴 색과 어트리뷰트(attribute)로 구분됩니다. 디폴트로, HTML 태그는 굵은 검정 텍스트로 표시되고 HTML 어트리뷰트 이름은 검정색으로 표시되며 HTML 어트리뷰트 값은 파란색으로 표시됩니다. 스크립트는 스크립트 괄호 <% %> 사이에 녹색으로 표시됩니다. Editor Properties 다이얼로그 박스의 Color 탭에서 이들 항목에 대한 디폴트 글꼴 색과 어트리뷰트를 변경할 수 있습니다. Editor Properties 다이얼로그 박스는 에디터를 마우스 오른쪽 버튼으로 클릭하고 Properties를 선택하면 표시됩니다.

서로 다른 두 개의 HTML 관련 에디터 탭이 있습니다. HTML Result 탭에는 미리 보기의 원시 HTML 컨텐트가 표시됩니다. HTML Result, HTML Script 및 Preview 보기는 모두 읽기 전용입니다. 마지막 HTML 관련 에디터 탭인 CountryTable.html을 사용하여 편집할 수 있습니다.

이 페이지의 모양을 개선하려면 언제든지 CountryTable.html 탭이나 외부 에디터를 사용하여 HTML을 추가할 수 있습니다. 페이지 템플릿 편집 방법에 대한 자세한 내용은 34-24페이지의 "고급 HTML 디자인"을 참조하십시오.

단계 2. 그리드에 편집 커맨드 추가

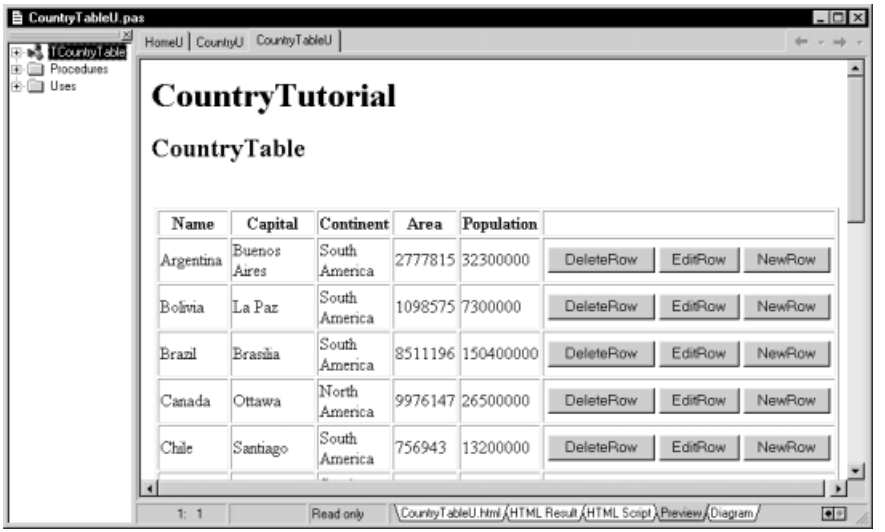
사용자가 행을 삭제, 삽입 또는 편집하여 테이블의 컨텐트를 업데이트해야 하는 경우가 있을 수 있습니다. 사용자들이 업데이트할 수 있게 하려면 커맨드 컴포넌트를 추가하십시오.

다음과 같은 방법으로 커맨드 컴포넌트를 추가합니다.

1. CountryTable의 Object TreeView에서 AdapterPageProducer 컴포넌트와 그 분기를 모두 확장합니다.
2. AdapterGrid1 컴포넌트를 마우스 오른쪽 버튼으로 클릭하고 Add All Columns를 선택합니다. 어댑터 그룹에 다섯 개의 열이 추가됩니다.
3. AdapterGrid1 컴포넌트를 마우스 오른쪽 버튼으로 다시 클릭하고 New Component를 선택합니다.
4. AdapterCommandColumn을 선택한 다음 OK를 클릭합니다. AdapterCommandColumn1 항목이 AdapterGrid1 컴포넌트에 추가됩니다.
5. AdapterCommandColumn1을 마우스 오른쪽 버튼으로 클릭하고 Add Commands를 선택합니다.
6. DeleteRow, EditRow 및 NewRow 명령을 동시에 선택한 다음 OK를 클릭합니다.
7. 페이지를 미리 보려면 코드 에디터의 하단에 있는 Preview 탭을 클릭합니다. 그림 34.7에 표시된 대로 테이블에 있는 각 행의 끝에 DeleteRow, EditRow, NewRow의 세 버튼이 새로 생깁니다. 애플리케이션이 실행되는 동안 이 버튼 중 하나를 누르면 연결된 액션이 수행됩니다.

계속하기 전에 Save All 버튼을 클릭하여 애플리케이션을 저장합니다.

그림 34.7 편집 커맨드를 추가한 이후의 CountryTable 미리 보기



Edit 폼 추가

이제 국가 테이블에 대한 Edit 폼을 처리할 웹 페이지 모듈을 만들 수 있습니다. 사용자는 Edit 폼을 통해 CountryTutorial 애플리케이션의 데이터를 변경할 수 있습니다. EditRow 또는 NewRow 버튼을 누르면 Edit 폼이 나타납니다. Edit 폼 작업이 완료되면 테이블에 수정된 정보가 자동으로 나타납니다.

단계 1. 새 웹 페이지 모듈 추가

다음과 같은 방법으로 새 웹 페이지 모듈을 추가합니다.

1 File|New|Other를 선택합니다.

2 New Items 다이얼로그 박스에서 WebSnap 탭을 선택하고 WebSnap Page Module을 선택합니다.

3 다이얼로그 박스에서 Producer Type을 AdapterPageProducer로 설정합니다.

4 Page Name 필드를 CountryForm으로 설정합니다.

5 Published 체크 박스를 선택 해제하여 해당 페이지가 이 사이트의 사용 가능한 페이지 리스트에 나타나지 않게 합니다. Edit 폼은 Edit 버튼을 클릭하여 액세스할 수 있으며 Edit 폼의 컨텐트는 수정할 국가 테이블 행에 따라 달라집니다.

6 나머지 필드와 선택 사항은 기본값으로 그대로 둡니다. OK를 클릭합니다.

단계 2. 새 모듈 저장

프로젝트 파일과 같은 디렉토리에 모듈을 저장합니다. 애플리케이션은 실행된 후 실행 파일과 같은 디렉토리에서 CountryFormU.html 파일을 찾습니다.

1 File|Save를 선택합니다.

2 File name 필드에 CountryFormU.cpp를 입력하고 OK를 클릭합니다.

단계 3. 새 모듈에서 CountryTableU를 액세스할 수 있게 만들기

모듈이 Adapter 컴포넌트를 액세스할 수 있게 하려면 include에 CountryTableU 유닛을 추가합니다.

1 File|Include Unit Hdr을 선택합니다.

2 리스트에서 CountryTableU를 선택한 다음 OK를 클릭합니다.

3 File|Save를 선택합니다.

단계 4. 입력 필드 추가

HTML 폼에 데이터 입력 필드를 생성하려면 AdapterPageProducer 컴포넌트에 컴포넌트를 추가합니다.

다음과 같은 방법으로 입력 필드를 추가합니다.

1 View|Project Manager를 선택합니다.

2 Project Manager 윈도우에서 CountryTutorial.exe를 확장하고 CountryFormU 항목을 더블 클릭합니다.

3 Object TreeView에서 AdapterPageProducer 컴포넌트를 확장하고 WebPageItems를 마우스 오른쪽 버튼으로 클릭한 다음 New Component를 선택합니다.

4 AdapterForm을 선택한 다음 OK를 클릭합니다. AdapterForm1 엔트리가 Object TreeView에 나타납니다.

5 AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.

6 AdapterFieldGroup을 선택한 다음 OK를 클릭합니다. AdapterFieldGroup1 엔트리가 Object TreeView에 나타납니다.

7 Object Inspector 윈도우에서 *Adapter* 속성을 CountryTable->Adapter로 설정합니다. *AdapterMode* 속성을 Edit로 설정합니다.

8 페이지를 미리 보려면 코드 에디터 하단에 있는 Preview 탭을 클릭합니다. 미리 보기는 그림 34.8에 표시된 것과 비슷할 것입니다.

그림 34.8 CountryForm 미리 보기



단계 5. 버튼 추가

HTML 폼에 전송 버튼을 생성하려면 AdapterPageProducer 컴포넌트에 컴포넌트들을 추가합니다. 다음과 같은 방법으로 컴포넌트들을 추가합니다.

1. Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장합니다.
2. AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
3. AdapterCommandGroup을 선택한 다음 OK를 클릭합니다. AdapterCommandGroup1 엔트리가 Object TreeView에 나타납니다.
4. Object Inspector에서 *DisplayComponent* 속성을 AdapterFieldGroup1로 설정합니다.
5. AdapterCommandGroup1 엔트리를 마우스 오른쪽 버튼으로 클릭하고 Add Commands를 선택합니다.
6. Cancel, Apply 및 Refresh Row 명령을 다중 선택한 다음 OK를 클릭합니다.
7. 페이지를 미리 보려면 코드 에디터 윈도우의 하단에 있는 Preview 탭을 클릭합니다. 미리 보기에 국가 폼이 표시되지 않으면 Code 탭을 클릭한 다음 Preview 탭을 다시 클릭합니다. 미리 보기는 그림 34.9에 표시된 것과 비슷할 것입니다.

그림 34.9 폼 전송 버튼이 있는 CountryForm



단계 6. 그리드 페이지에 폼 액션 연결

사용자가 버튼을 클릭하면 설명된 액션을 차례로 수행하는 어댑터 액션이 실행됩니다. 다음과 같은 방법으로 어댑터 액션이 실행된 후 표시할 페이지를 지정합니다.

1 Object TreeView에서 AdapterCommandGroup1을 확장하여 CmdCancel, CmdApply 및 CmdRefreshRow 엔트리를 표시합니다.

2 CmdCancel을 선택합니다. Object Inspector 윈도우의 *PageName* 속성에 CountryTable을 입력합니다.

3 CmdApply를 선택합니다. Object Inspector 윈도우의 *PageName* 속성에 CountryTable을 입력합니다.

단계 7. 폼 페이지에 그리드 액션 연결

그리드에 있는 버튼을 누르면 어댑터 액션이 실행됩니다. 다음과 같은 방법으로 어댑터 액션에 응답하여 표시되는 페이지를 지정합니다.

1 View | Project Manager를 선택합니다.

2 Project Manager에서 CountryTutorial.exe를 확장하고 CountryTableU 엔트리를 더블 클릭합니다.

3 Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장하여 CmdNewRow, CmdEditRow 및 CmdDeleteRow 엔트리를 표시합니다. 이 엔트리들은 AdapterCommandColumn1 엔트리 아래 표시됩니다.

4 CmdNewRow를 선택합니다. Object Inspector의 *PageName* 속성에 CountryForm을 입력합니다.

5 CmdEditRow를 선택합니다. Object Inspector의 *PageName* 속성에 CountryForm을 입력합니다.

애플리케이션이 제대로 작동하며 모든 버튼이 액션을 수행하는지 확인하려면 애플리케이션을 실행합니다. 애플리케이션 실행 방법에 대한 자세한 내용은 34-22페이지의 "완성된 애플리케이션 실행"을 참조하십시오. 애플리케이션을 실행하면 서버가 실행됩니다. 애플리케이션이 작동하는지 확인하려면 웹 브라우저에서 애플리케이션을 표시해야 합니다. 이 작업은 Web Application 디버거에서 애플리케이션을 시작하여 수행할 수 있습니다.

참고 잘못된 타입과 같은 데이터베이스 오류는 표시되지 않습니다. 예를 들어, Area 필드에 잘못된 값(예: 'abc')을 입력하여 새 국가를 추가하려고 시도해 보십시오.

완성된 애플리케이션 실행

다음과 같은 방법으로 완성된 애플리케이션을 실행합니다.

1. Run|Run을 선택합니다. 폼이 표시됩니다. Web App Debugger 실행 파일 웹 애플리케이션은 COM 서버이고 사용자에게 보이는 폼은 COM 서버용 콘솔 윈도우입니다. 프로젝트를 처음으로 실행하면 Web App Debugger가 직접 액세스할 수 있는 COM 객체가 등록됩니다.
2. Tools|Web App Debugger를 선택합니다.
3. 디폴트 URL 링크를 클릭하여 ServerInfo 페이지를 표시합니다. ServerInfo 페이지에는 등록된 모든 Web Application Debugger 실행 파일의 이름이 표시됩니다.
4. 드롭다운 리스트에서 CountryTutorial을 선택하고 Go 버튼을 클릭합니다.

이제 브라우저에 Country Tutorial 애플리케이션이 나타납니다. CountryTable 링크를 클릭하면 CountryTable 페이지가 나타납니다.

오류 리포트 추가

지금은 애플리케이션이 사용자에게 오류를 보여주지 않습니다. 예를 들어, 사용자가 국가 레코드의 Area 필드에 문자를 입력해도 오류가 발생했다는 통지를 받지 못합니다. 이제 국가 테이블을 편집하는 어댑터 액션을 실행하는 동안 발생하는 오류를 표시하기 위해 *AdapterErrorList* 컴포넌트를 추가합니다.

단계 1. 그리드에 오류 지원 추가

다음과 같은 방법으로 그리드에 오류 지원을 추가합니다.

1. CountryTable의 Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장하여 AdapterForm1을 표시합니다.
2. AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
3. 리스트에서 AdapterErrorList를 선택한 다음 OK를 클릭합니다. AdapterErrorList1 엔트리가 Object TreeView에 나타납니다.
4. AdapterErrorList1을 끌어 놓거나 Object TreeView 툴바에서 위쪽 화살표를 사용하여 AdapterErrorList1을 AdapterGrid1 위로 이동합니다.
5. Object Inspector에서 *Adapter* 속성을 Adapter로 설정합니다.

단계 2. 폼에 오류 지원 추가

다음과 같은 방법으로 폼에 오류 지원을 추가합니다.

1. CountryForm의 Object TreeView에서 *AdapterPageProducer* 컴포넌트와 모든 분기를 확장하여 AdapterForm1을 표시합니다.
2. AdapterForm1을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다.
3. 리스트에서 AdapterErrorList를 선택한 다음 OK를 클릭합니다. AdapterErrorList1 엔트리가 Object TreeView에 나타납니다.
4. AdapterErrorList1을 끌어 놓거나 Object TreeView 툴바에서 위쪽 화살표를 사용하여 AdapterErrorList1을 AdapterFieldGroup1 위로 이동합니다.
5. Object Inspector에서 *Adapter* 속성을 CountryTable->Adapter로 설정합니다.

단계 3. 오류 리포트 메커니즘 테스트

오류 리포트 메커니즘을 관찰하려면 먼저 C++Builder IDE를 조금 변경해야 합니다. Tools|Debbugger Options를 선택합니다. Language Exceptions 탭에서 Stop on Delphi Exceptions 체크 박스를 선택 해제하여 예외가 발견되어도 애플리케이션이 계속 실행되게 합니다. 다음과 같은 방법으로 그리드 오류를 테스트합니다.

1. 애플리케이션을 실행하고 Web Application Debugger를 사용하여 CountryTable 페이지를 찾습니다. 이 작업을 수행하는 방법에 대한 자세한 내용은 34-22페이지의 "완성된 애플리케이션 실행"을 참조하십시오.
2. 다른 브라우저 윈도우를 열고 CountryTable 페이지를 찾습니다.
3. 그리드의 첫 번째 행에서 DeleteRow 버튼을 클릭합니다.
4. 두 번째 브라우저를 새로 고치지 않고 그리드의 첫 번째 행에서 DeleteRow 버튼을 클릭합니다.

"Row not found in Country"라는 오류 메시지가 그리드 위에 표시됩니다.

다음과 같은 방법으로 폼 오류를 테스트합니다.

1. 애플리케이션을 실행하고 Web Application 디버거를 사용하여 CountryTable 페이지를 찾습니다.
2. EditRow 버튼을 클릭합니다. CountryForm 페이지가 표시됩니다.
3. Area 필드를 'abc'로 변경하고 Apply 버튼을 클릭합니다.

"Invalid value for field 'Area'"라는 오류 메시지가 첫 번째 필드 위에 표시됩니다.

이제 WebSnap 자습서를 완료했습니다. Stop on Delphi Exceptions 체크 박스를 다시 선택할 수 있습니다. 또한 완료된 애플리케이션을 다음에 참조할 수 있도록 File|Save All을 선택하여 애플리케이션을 저장합니다.

고급 HTML 디자인

WebSnap에서 어댑터 및 어댑터 페이지 프로듀서를 사용하면 웹 서버 애플리케이션에서 스크립트된 HTML 페이지를 쉽게 만들 수 있습니다. 필요한 모든 WebSnap 도구를 사용하여 애플리케이션 데이터를 위한 웹 페이지 프로토타입을 만들 수 있습니다. 그러나 WebSnap의 가장 강력한 기능 중 하나는 외부의 웹 디자인 전문 기술을 사용자의 애플리케이션으로 통합하는 기술입니다. 이 단원에서는 웹 서버 디자인 및 유지 보수 과정을 확장하여 다른 도구와 프로그래머가 아닌 팀 멤버를 포함시키는 몇 가지 방법에 대해 설명합니다.

WebSnap 개발의 최종 산출물은 서버가 생성하는 페이지에 대한 HTML 템플릿과 서버 애플리케이션입니다. 템플릿에는 스크립트와 HTML이 함께 포함됩니다. 템플릿을 생성한 후 원하는 HTML 도구를 사용하여 언제든지 편집할 수 있습니다. 에디터에서 스크립트가 실수로 손상되지 않게 하려면 Microsoft FrontPage처럼 스크립트 태그가 포함되도록 지원하는 도구를 사용하는 것이 가장 좋습니다. IDE 외부에서 템플릿 페이지를 편집할 수 있는 능력은 여러 가지로 활용될 수 있습니다.

예를 들어, C++ Builder 개발자는 디자인 타임에 자신이 원하는 외부 에디터를 사용하여 HTML 템플릿을 편집할 수 있습니다. 이렇게 하면 개발자들이 C++ Builder에는 없지만 외부 HTML 에디터에는 있을 수 있는 고급 서식 및 스크립트 기능을 사용할 수 있습니다. 다음과 같은 방법으로 IDE에서 외부 HTML 에디터를 활성화합니다.

1. 메인 메뉴에서 Tools|Environment Options를 선택합니다. Environment Options 다이얼로그 박스에서 Internet 탭을 클릭합니다.
2. Internet File Types 박스에서 HTML을 선택하고 Edit 버튼을 클릭하여 Edit Type 다이얼로그 박스를 표시합니다.
3. Edit Action 박스에서 HTML 에디터와 연결된 액션을 선택합니다. 예를 들어, 시스템에 있는 디폴트 HTML 에디터를 선택하려면 드롭다운 리스트에서 Edit를 선택합니다. OK를 더블 클릭하여 Edit Type 다이얼로그 박스와 Environment Options 다이얼로그 박스를 닫습니다.

HTML 템플릿을 편집하려면 해당 템플릿이 들어 있는 유닛을 엽니다. Edit 윈도우에서 마우스 오른쪽 버튼을 클릭하고 컨텍스트 메뉴에서 HTML 에디터를 선택합니다. 편집할 템플릿이 HTML 에디터에 독립적인 윈도우로 표시됩니다. 에디터는 IDE와 독립적으로 실행되므로 작업이 완료되면 템플릿을 저장하고 에디터를 닫습니다.

제품이 배포된 후 최종 HTML 페이지의 모양을 변경하고자 할 수 있습니다. 심지어 소프트웨어 개발 팀이 최종 페이지 레이아웃을 담당하지 않을 수도 있습니다. 이 작업은 조직 내의 전문 웹 페이지 디자이너가 담당하게 될 것입니다. 페이지 디자이너는 C++ Builder 개발 경험이 없을 수도 있지만, 다행히 이러한 경험은 필요하지 않습니다. 소스 코드를 변경하지 않고 제품 개발 및 유지 보수 기간 동안 언제라도 페이지 템플릿을 편집할 수 있습니다. 따라서, WebSnap HTML 템플릿을 사용하면 서버 개발과 유지 보수를 더욱 효과적으로 수행할 수 있습니다.

HTML 파일에서 서버사이드 스크립트 조작

개발 기간 동안 언제든지 페이지 템플릿의 HTML을 수정할 수 있습니다. 그러나 서버사이드 스크립트는 다른 문제일 수 있습니다. C++ Builder 외부의 템플릿에서도 서버사이드 스크립트를 항상 조작할 수 있지만 어댑터 페이지 프로듀서에서 생성된 페이지는 수정하지 않는 것이 좋습니다. 어댑터 페이지 프로듀서는 런타임 시 페이지 템플릿에서 서버사이드 스크립트를 변경할 수 있다는 점에서 일반적인 페이지 프로듀서와 다릅니다. 다른 스크립트가 동적으로 추가될 경우 스크립트의 동작 방법을 예상하기 어렵습니다. 스크립트를 직접 조작하려면 웹 페이지 모듈에 어댑터 페이지 프로듀서 대신 페이지 프로듀서를 추가해야 합니다.

어댑터 페이지 프로듀서를 사용하는 웹 페이지 모듈이 있을 경우 다음 단계를 사용하여 일반적인 페이지 프로듀서를 사용하도록 변환할 수 있습니다.

1. 변환할 모듈(ModuleName이라고 가정)에서 HTML Script 탭의 모든 정보를 ModuleName.html 탭에 복사하여 이전에 들어 있던 모든 정보를 바꿉니다.
2. 컴포넌트 팔레트의 Internet 탭에 있는 페이지 프로듀서를 웹 페이지 모듈로 가져다 놓습니다.
3. 페이지 프로듀서의 *ScriptEngine* 속성을 대체 어댑터 페이지 프로듀서의 해당 속성과 일치되게 설정합니다.
4. 웹 페이지 모듈의 페이지 프로듀서를 어댑터 페이지 프로듀서에서 새 페이지 프로듀서로 변경합니다. Preview 탭을 클릭하여 페이지 컨텐트가 변경되지 않았는지 확인합니다.
5. 이제 어댑터 페이지 프로듀서는 무시되므로 웹 페이지 모듈에서 삭제할 수 있습니다.

로그인 지원

많은 웹 서버 애플리케이션은 로그인 지원을 필요로 합니다. 예를 들어, 서버 애플리케이션에서 웹 사이트의 특정 부분을 액세스하려면 사용자가 로그인해야 할 수 있습니다. 페이지 모양은 사용자마다 다를 수 있으며, 로그인해야 웹 서버가 정확한 페이지를 보낼 수 있습니다. 또한 서버에는 메모리와 프로세서 시간에 대한 물리적인 제한이 있기 때문에 때로는 서버 애플리케이션이 지정된 시간에 사용자 수를 제한할 수 있어야 합니다.

WebSnap을 사용하여 웹 서버 애플리케이션에 로그인 지원을 통합하는 것은 매우 간단합니다. 이 단원에서는 개발 프로세스를 시작할 때부터 설계하거나 기존 애플리케이션에 로그인 지원을 추가하는 방법을 배우게 될 것입니다.

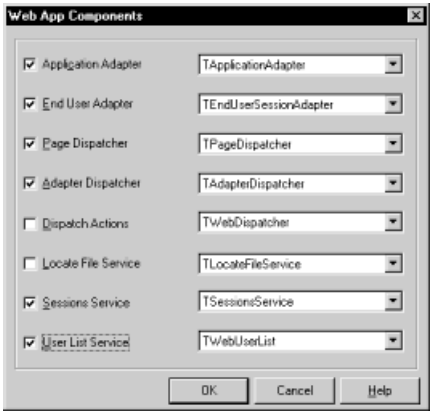
로그인 지원 추가

로그인 지원을 구현하려면 웹 애플리케이션 모듈에는 다음과 같은 컴포넌트가 포함되어 있어야 합니다.

- 서버 사용자의 사용자 이름, 암호 및 사용 권한이 들어 있는 유저 리스트 서비스 (*TWebUserList* 타입의 객체)
- 현재 서버에 로그인한 사용자에 대한 정보를 저장하는 세션 서비스(*TSessionsService*)
- 로그인과 연결된 액션을 처리하는 엔드 유저 어댑터(*TEndUserSessionAdapter*)

웹 서버 애플리케이션을 처음 만들 때 New WebSnap Application 다이얼로그 박스를 사용하여 이 컴포넌트들을 추가할 수 있습니다. 이 다이얼로그 박스에서 Components 버튼을 클릭하여 New Web App Components 다이얼로그 박스를 표시합니다. End User Adapter, Sessions Service 및 Web User List 박스에 선택 표시를 합니다. End User Adapter 박스 옆에 있는 드롭다운 메뉴에서 *TEndUserSessionAdapter*를 선택하여 엔드 유저 어댑터 타입을 선택합니다. 디폴트로 선택되는 *TEndUserAdapter*는 현재 사용자를 추적할 수 없기 때문에 로그인 지원에는 알맞지 않습니다. 작업이 완료되면 다이얼로그 박스가 아래 표시와 같이 나타납니다. OK를 더블 클릭하여 다이얼로그 박스를 닫습니다. 이제 웹 애플리케이션 모듈이 로그인 지원에 필요한 컴포넌트들을 가지고 있습니다.

그림 34.10 로그인 지원 옵션이 선택되어 있는 Web App Components 다이얼로그 박스



기존 웹 애플리케이션 모듈에 로그인 지원을 추가할 경우 이 컴포넌트들을 컴포넌트 팔레트의 WebSnap 탭에서 개발자의 모듈로 직접 놓을 수 있습니다. 웹 애플리케이션 모듈은 자동으로 구성됩니다.

디자인 단계 동안 세션 서비스 및 엔드 유저 어댑터에 주의를 기울일 필요는 없지만 웹 유저 리스트에는 주의를 기울여야 합니다. WebUserList 컴포넌트 에디터를 통해 디폴트 사용자를 추가하고 이들의 읽기/수정 사용 권한을 설정할 수 있습니다. 컴포넌트를 더블 클릭하여 사용자 이름, 암호 및 액세스 권한을 설정할 수 있는 에디터를 표시합니다. 액세스 권한 설정 방법에 대한 자세한 내용은 34-29페이지의 "사용자 액세스 권한"을 참조하십시오.

세션 서비스 사용

세션 서비스는 *TSessionsService* 타입의 객체로 웹 서버 애플리케이션에 로그인하는 사용자를 추적합니다. 세션 서비스는 사용자마다 다른 세션을 할당하고 이름/값 쌍(예: 사용자 이름)을 사용자와 연결합니다.

세션 서비스에 포함된 정보는 애플리케이션의 메모리에 저장됩니다. 따라서 세션 서비스가 동작하기 위해서는 웹 서버 애플리케이션이 여러 요청들 사이에 계속 실행되어야 합니다. CGI와 같이 여러 요청들 사이에 종료되는 서버 애플리케이션 타입도 있습니다.

참고 애플리케이션이 로그인을 지원하게 하려면 여러 요청들 사이에 종료되지 않는 서버 타입을 사용해야 합니다. 프로젝트에서 Web App 디버거 실행 파일을 생성할 경우 백그라운드에서 실행 중인 애플리케이션이 있어야 페이지 요청을 받을 수 있습니다. 그렇지 않으면 페이지 요청이 있은 다음 애플리케이션이 종료되기 때문에 사용자가 로그인 페이지를 넘어갈 수 없습니다.

세션 서비스에는 디폴트 서버 동작을 변경하는 데 사용할 수 있는 두 가지 중요한 속성이 있습니다. *MaxSessions* 속성은 지정된 시간에 시스템에 로그인할 수 있는 사용자 수를 지정합니다. *MaxSessions*에 대한 기본값은 -1이며 이 경우 허용되는 사용자 수에 대한 소프트웨어 제한이 없습니다. 물론, 서버 하드웨어에 새 사용자를 위한 메모리 또는 프로세서 주기가 부족하여 시스템 성능이 저하될 수도 있습니다. 과도한 수의 사용자로 인해 서버가 마비될 것을 염려한다면 *MaxSessions*를 적절한 값으로 설정해야 합니다.

DefaultTimeout 속성은 기본 시간 제한(분)을 지정합니다. *DefaultTimeout* 시간(분)이 경과할 때까지 사용자의 활동이 없으면 세션이 자동으로 종료됩니다. 사용자가 로그인한 경우 모든 로그인 정보가 소실됩니다. 기본값은 20입니다. 특정 세션에서 *TimeoutMinutes* 속성을 변경하여 기본값을 오버라이드할 수 있습니다.

로그인 페이지

물론, 애플리케이션에 로그인 페이지도 필요합니다. 사용자는 제한된 페이지를 액세스하려고 시도하거나 그런 시도 이전에 인증을 위한 사용자 이름과 암호를 입력합니다. 또한 인증이 완료될 때 표시되는 페이지를 지정할 수 있습니다. 사용자 이름과 암호가 웹 유저 리스트에 있는 사용자와 일치하면 사용자는 해당 액세스 권한을 취득하고 로그인 페이지에 지정된 페이지로 이동됩니다. 사용자가 인증을 받는 데 실패하면 로그인 페이지가 다시 표시(디폴트 액션)되거나 다른 액션이 발생될 수 있습니다.

다행히도, WebSnap은 웹 페이지 모듈 및 어댑터 페이지 프로듀서를 사용하여 간단한 로그인 페이지를 쉽게 만들 수 있습니다. 로그인 페이지를 만들려면 새 웹 페이지 모듈을 먼저 만듭니다. File|New|Other를 선택하여 New Items 다이얼로그 박스를 표시한 다음 WebSnap 탭에서 WebSnap Page Module을 선택합니다. AdapterPageProducer를 페이지 프로듀서 타입으로 선택합니다. 다른 옵션을 원하는대로 채웁니다. Login은 로그인 페이지에 대한 좋은 이름이 될 수 있습니다.

이제 사용자 이름 필드, 암호 필드, 사용자가 로그인한 후 표시되는 페이지를 선택하는 선택 박스, 페이지 폼을 전송하고 사용자를 인증하는 Login 버튼 등 가장 기본적인 로그인 페이지 필드를 추가합니다. 다음과 같은 방법으로 이들 필드를 추가합니다.

1. 컴포넌트 팔레트의 WebSnap 탭에 있는 LoginFormAdapter 컴포넌트를 방금 만든 웹 페이지 모듈에 추가합니다.
2. *AdapterPageProducer* 컴포넌트를 더블 클릭하여 웹 페이지 에디터 윈도우를 표시합니다.
3. 왼쪽 위에 있는 창에서 *AdapterPageProducer*를 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다. Add Web Component 다이얼로그 박스에서 *AdapterForm*을 선택하고 OK를 클릭합니다.

4 AdapterForm에 *AdapterFieldGroup*을 추가합니다. 왼쪽 위에 있는 창에서 *AdapterForm*을 마우스 오른쪽 버튼으로 클릭하고 New Component를 선택합니다. Add Web Component 다이얼로그 박스에서 *AdapterFieldGroup*을 선택하고 OK를 클릭합니다.

5 이제 Object Inspector로 이동하고 *AdapterFieldGroup*의 *Adapter* 속성을 *LoginFormAdapter*로 설정합니다. 웹 페이지 에디터의 Browser 탭에 UserName, Password 및 NextPage 필드가 자동으로 나타날 것입니다.

이렇게 WebSnap은 대부분의 작업을 몇 가지 간단한 단계로 수행합니다. 로그인 페이지에는 아직 인증을 위한 정보를 폼으로 전송하는 Login 버튼이 없습니다. 다음과 같은 방법으로 Login 버튼을 추가합니다.

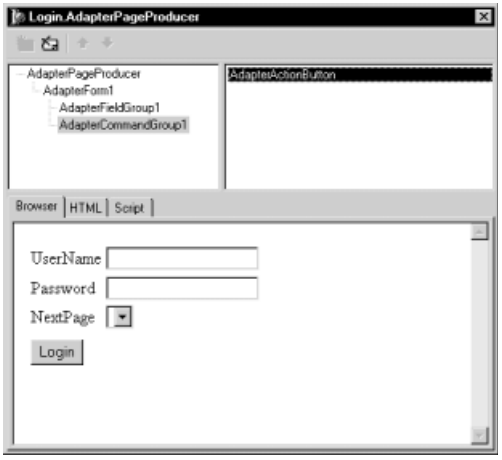
1 *AdapterForm*에 *AdapterCommandGroup*을 추가합니다.

2 *AdapterCommandGroup*에 *AdapterActionButton*을 추가합니다.

3 웹 페이지 에디터의 오른쪽 위에 있는 창에 나열된 *AdapterActionButton*을 클릭하고 Object Inspector를 사용하여 *ActionName* 속성을 *Login*으로 변경합니다. 웹 페이지 에디터의 Browser 탭에서 로그인 페이지를 미리 볼 수 있습니다.

웹 페이지 에디터는 아래 표시된 것과 비슷한 모양일 것입니다.

그림 34.11 웹 페이지 에디터에 표시되는 로그인 페이지의 예제



AdapterFieldGroup 아래에 버튼이 나타나지 않으면 웹 페이지 에디터의 *AdapterFieldGroup* 아래에 있는 리스트에 *AdapterCommandGroup*이 있는지 확인합니다. *AdapterCommandGroup*이 있으면 선택하고 웹 페이지 에디터의 아래쪽 화살표를 클릭합니다. 일반적으로 웹 페이지 요소는 웹 페이지 에디터에 나타나는 순서대로 세로로 나타납니다.

로그인 페이지가 제대로 작동하려면 필요한 단계가 하나 더 있습니다. 엔드 유저 세션 어댑터의 어느 페이지가 로그인 페이지인지 지정해야 합니다. 그렇게 하려면 웹 애플리케이션 모듈에서 *EndUserSessionAdapter* 컴포넌트를 선택합니다. Object Inspector에서 *LoginPage* 속성을 로그인 페이지 이름으로 변경합니다. 이제 웹 서버 애플리케이션의 모든 페이지에서 로그인 페이지를 사용할 수 있습니다.

로그인이 필요한 페이지 설정

작업 중인 로그인 페이지가 있을 경우 액세스를 제어해야 하는 페이지에 대해서는 로그인을 하도록 설정합니다. 페이지에서 로그인을 하도록 요청하는 가장 쉬운 방법은 페이지에 해당 요구 사항을 디자인하는 것입니다. 웹 페이지 모듈을 처음 만들 때 New WebSnap Page Module 다이얼로그 박스의 Page 섹션에 있는 Login Required 박스에 선택 표시를 합니다.

로그인이 필요하지 않은 페이지를 만들더라도 나중에 로그인이 필요한 페이지로 만들 수 있습니다. 웹 페이지 모듈을 만든 후 다음과 같은 방법으로 로그인을 요청합니다.

1 에디터에서 웹 페이지 모듈과 연결된 소스 코드 파일을 엽니다.

2 정적 WebInit 함수의 선언이 나타나도록 아래로 스크롤합니다.

3 둘러싸고 있는 /* 및 */ 기호를 제거하여 매개변수 리스트에서 *wpLoginRequired* 부분의 주석 처리를 제거합니다. WebInit 함수는 아래 표시된 함수와 비슷할 것입니다.

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3),
crOnDemand, caCache, PageAccess << wpPublished << wpLoginRequired,
".html", "", "", "", "");
```

페이지에서 로그인 요구 사항을 제거하려면 프로세스를 역으로 수행하고 선언의 *wpLoginRequired* 부분에 다시 주석 처리를 합니다.

참고 동일한 과정으로 페이지를 published로 선언하거나 해제할 수 있습니다. 필요에 따라 *wpPublished* 부분의 시작과 끝에 주석 표시를 추가하거나 제거하면 됩니다.

사용자 액세스 권한

사용자 액세스 권한은 웹 서버 애플리케이션의 중요한 부분입니다. 서버에서 제공하는 정보를 보거나 수정할 수 있는 사용자를 제어할 수 있어야 합니다. 예를 들어, 온라인 소매 판매를 처리할 서버 애플리케이션을 구축한다고 가정합니다. 사용자가 카탈로그에 있는 항목을 볼 수는 있지만 가격을 변경하지는 못하게 하려 합니다. 이 경우 액세스 권한이 중요한 문제가 됩니다.

WebSnap은 페이지 및 서버 컨텐트에 대한 액세스를 제어할 수 있는 여러 가지 방법을 제공합니다. 이전 단원에서는 로그인을 요청하여 페이지 액세스를 제어하는 방법을 배웠습니다. 이 외에도 다른 옵션들이 있습니다. 예를 들면, 다음과 같습니다.

- 적절한 수정 액세스 권한이 있는 사용자는 에디트 박스의 데이터 필드를 볼 수 있지만, 그렇지 않은 사용자는 필드 컨텐트를 볼 수는 있지만 편집할 수 없습니다.
- 올바른 보기 액세스 권한이 없는 사용자에게 특정 필드를 숨길 수 있습니다.
- 승인되지 않은 사용자가 특정 페이지를 받지 못하게 할 수 있습니다.

이러한 동작을 구현하는 방법에 대한 자세한 내용은 이 단원에 설명되어 있습니다.

필드를 동적으로 에디트 박스 또는 텍스트로 표시

어댑터 페이지 프로듀서를 사용하는 경우 다른 액세스 권한을 갖는 사용자에 대해 페이지 요소 모양을 변경할 수 있습니다. 예를 들어, Demos 디렉토리의 WebSnap 하위 디렉토리에 있는 Biolife 데모에는 지정된 형식에 대한 모든 정보를 보여 주는 폼 페이지가 들어 있습니다. 폼은 사용자가 그리드에서 Details 버튼을 클릭하면 나타납니다. Will로 로그인한 사용자는 일반 텍스트로 표시된 데이터를 보게 됩니다. Will에게는 데이터 수정이 허용되지 않기 때문에 폼에 데이터 수정 메커니즘이 제공되지 않습니다. 사용자 Ellen은 수정 권한이 있기 때문에 Ellen이 폼 페이지를 표시하면 필드 컨텐트를 변경할 수 있는 일련의 에디트 박스들이 표시됩니다. 이런 방법으로 액세스 권한을 사용하면 추가 페이지를 만드는 시간을 절약할 수 있습니다.

TAdapterDisplayField 및 *TAdapterDisplayColumn*과 같은 일부 페이지 요소의 모양은 *ViewMode* 속성에 의해 결정됩니다. *ViewMode*를 *vmToggleOnAccess*로 설정하면 수정 액세스 권한이 있는 사용자에게는 페이지 요소가 에디트 박스로 나타나고 수정 액세스 권한이 없는 사용자에게는 일반 텍스트로 표시됩니다. *ViewMode* 속성을 *vmToggleOnAccess*로 설정하면 페이지 요소의 모양과 함수를 동적으로 결정할 수 있습니다.

웹 유저 리스트는 시스템에 로그인할 수 있는 사용자에게 하나씩 할당되는 *TWebUserListItem* 객체의 리스트입니다. 사용자의 사용 권한은 웹 사용자 리스트 항목의 *AccessRights* 속성에 저장됩니다. *AccessRights*는 텍스트 문자열이기 때문에 원하는 사용 권한을 자유롭게 지정할 수 있습니다. 서버 애플리케이션에서 사용하려고 하는 모든 종류의 액세스 권한에 대해 이름을 만듭니다. 한 사용자가 여러 액세스 권한을 갖게 하려면 리스트에 있는 항목을 공백, 세미콜론 또는 쉼표로 분리하십시오.

필드에 대한 액세스 권한은 *ViewAccess* 및 *ModifyAccess* 속성에 의해 결정됩니다. *ViewAccess*는 주어진 필드를 보는 데 필요한 액세스 권한의 이름을 저장합니다. *ModifyAccess*는 필드 데이터를 수정하는 데 필요한 액세스 권한을 지시합니다. 이 속성들은 해당 속성이 들어 있는 어댑터 객체와 각 필드의 두 위치에 나타납니다.

액세스 권한 검사는 2단계 과정입니다. 페이지에서 필드의 모양을 결정할 때 애플리케이션은 먼저 필드 자체의 액세스 권한을 검사합니다. 값이 비어 있는 문자열이면 애플리케이션은 필드가 들어 있는 어댑터의 액세스 권한을 검사합니다. 어댑터 속성 역시 빈 문자열이면 애플리케이션은 디폴트 동작을 따릅니다. 수정 액세스에 대한 디폴트 동작은 웹 사용자 리스트에서 비어 있지 않은 *AccessRights* 속성을 갖는 모든 사용자에게 수정을 허용하는 것입니다. 보기 액세스에 대한 사용 권한은 보기 액세스 권한을 지정하지 않으면 자동으로 허용됩니다.

필드 및 컨텐트 숨기기

적절한 보기 권한이 없는 사용자에게 필드의 컨텐트를 숨길 수 있습니다. 먼저 필드에 대한 *ViewAccess* 속성을 사용자가 갖기를 원하는 사용 권한과 일치하도록 설정합니다. 그런 다음 필드의 페이지 요소에 대한 *ViewAccess*를 *vmToggleOnAccess*로 설정해야 합니다. 필드 캡션은 나타나지만 필드 값은 나타나지 않습니다.

물론, 사용자가 보기 권한이 없을 경우에는 필드에 대한 모든 참조를 숨기는 것이 가장 좋습니다. 그렇게 하려면 필드의 페이지 요소에 대한 *HideOptions*가 *hoHideOnNoDisplayAccess*를 포함하도록 설정합니다. 캡션과 필드 컨텐트가 모두 표시되지 않습니다.

페이지 액세스 금지

어떤 페이지는 인증되지 않은 사용자가 액세스해서는 안된다고 결정할 수도 있습니다. 페이지를 표시하기 전에 액세스 권한을 검사하게 하려면 모듈의 WebInit 함수 선언을 변경합니다. 이 함수는 모듈의 소스 코드에 있습니다.

WebInit 함수는 최대 9개의 요소를 가집니다. WebSnap은 일반적으로 9개의 요소 중 네 개를 기본값(빈 문자열)으로 그대로 두기 때문에 호출은 대체로 다음과 같이 표시됩니다.

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3),
crOnDemand, caCache, PageAccess << wpPublished /* << wpLoginRequired */,
".html", "", "", "", "");
```

액세스를 허용하기 전에 사용 권한을 검사하려면 아홉 번째 매개변수에 필요한 권한에 대한 문자열을 제공해야 합니다. 예를 들어, 사용 권한이 Access라고 가정하면 다음과 같은 방법으로 WebInit 함수를 수정할 수 있습니다.

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3),
crOnDemand, caCache, PageAccess << wpPublished /* << wpLoginRequired */,
".html", "", "", "", "Access");
```

이제 Access 권한이 없는 사용자에 대해서는 페이지 액세스가 거부됩니다.

WebSnap의 서버사이드 스크립트

페이지 프로듀서 템플릿은 JScript 또는 VBScript와 같은 스크립트 언어를 포함할 수 있습니다. 페이지 프로듀서는 프로듀서의 컨텐트에 대한 요청에 응답하여 스크립트를 실행합니다. 웹 서버 애플리케이션이 스크립트를 분석하기 때문에 브라우저에 의해 분석되는 클라이언트사이드 스크립트와 반대로 서버사이드 스크립트라 합니다.

이 단원에서는 서버사이드 스크립트의 개념적 개요와 서버사이드 스크립트가 WebSnap 애플리케이션에서 사용되는 방법에 대해 설명합니다. 스크립트 객체와 스크립트 객체의 속성 및 메소드에 대한 자세한 내용은 부록의 "WebSnap 서버사이드 스크립트 참조"를 참조하십시오. 도움말 파일에 있는 C++Builder에 대한 객체 설명과 마찬가지로 서버사이드 스크립트에 대한 API 레퍼런스라고 생각하면 됩니다. 부록에는 스크립트를 사용하여 HTML 페이지를 생성하는 정확한 방법을 보여 주는 자세한 스크립트 예제도 들어 있습니다.

서버사이드 스크립트가 WebSnap 의 중요한 부분이기는 하지만 WebSnap 애플리케이션에서 반드시 스크립트를 사용해야 하는 것은 아닙니다. 스크립트는 HTML 생성에만 사용됩니다. 스크립트를 사용하여 HTML 페이지에 애플리케이션 데이터를 삽입할 수 있습니다. 사실 어댑터 및 다른 스크립트 사용 가능 객체에서 노출하는 거의 모든 속성은 읽기 전용입니다. 애플리케이션 데이터를 변경할 때는 서버사이드 스크립트를 사용하지 않습니다. 애플리케이션 데이터는 여전히 Pascal 또는 C++로 작성된 이벤트 핸들러 및 컴포넌트에서 관리합니다.

다른 방법으로 애플리케이션 데이터를 HTML 페이지에 삽입할 수도 있습니다. 원하는 경우 Web Broker의 투명 태그나 일부 다른 태그 기반 솔루션을 사용할 수 있습니다. 예를 들어, C++Builder Examples\WebSnap 폴더에 있는 여러 프로젝트는 스크립트 대신 XML 및 XSL을 사용합니다. 그러나 스크립트가 없다면 대부분의 HTML 생성 로직을 C++로 작성해야 하기 때문에 개발 기간이 늘어납니다.

WebSnap에서 사용되는 스크립트는 객체 지향적이고 조건부 로직 및 루핑을 지원하므로 페이지 생성 작업을 상당 부분 단순화할 수 있습니다. 예를 들어, 페이지에 일부 사용자만 편집할 수 있는 데이터 필드가 포함되어 있을 수 있습니다. 템플릿 페이지에서 스크립트를 사용하여 조건부 로직을 추가하면 인증된 사용자에게만 에디트 박스를 표시하고 인증되지 않은 사용자에게는 단순 텍스트만 표시할 수 있습니다. 태그 기반 접근 방법을 사용하는 경우에는 이러한 의사 결정을 HTML 생성 소스 코드에 프로그래밍해야 합니다.

액티브 스크립트

WebSnap은 *액티브 스크립트*에 의존하여 서버사이드 스크립트를 구현합니다. 액티브 스크립트는 COM 인터페이스를 통해 스크립트 언어를 애플리케이션 객체와 함께 사용할 수 있도록 Microsoft에서 만든 기술입니다. Microsoft는 VBScript와 JScript의 두 액티브 스크립트 언어를 제공합니다. 다른 언어 지원은 서드파티를 통해 사용할 수 있습니다.

스크립트 엔진

페이지 프로듀서의 *ScriptEngine* 속성은 템플릿의 스크립트를 분석하는 액티브 스크립트 엔진을 식별합니다. 디폴트로 JScript를 지원하도록 설정되지만 VBScript와 같은 다른 스크립트 언어를 지원할 수도 있습니다.

참고 WebSnap 어댑터는 JScript를 생성하도록 설계되어 있습니다. 다른 스크립트 언어의 경우 자체 스크립트 생성 로직을 제공해야 합니다.

스크립트 블록

스크립트 블록은 HTML 템플릿에 나타나며 <% 및 %>에 의해 구분됩니다. 스크립트 엔진은 스크립트 블록 내부에 있는 모든 텍스트를 분석합니다. 결과는 페이지 프로듀서 컨텐트의 일부가 됩니다. 페이지 프로듀서는 포함된 투명 태그를 번역한 후 스크립트 블록 외부에 텍스트를 씁니다. 또한 스크립트 블록은 텍스트에 괄호를 사용하여 조건부 로직과 루프로 텍스트 출력을 지시할 수 있습니다. 예를 들어, 다음 JScript 블록은 번호가 매겨진 다섯 줄의 리스트를 생성합니다.

```
<ul>
<% for (i=0;i<5;i++) { %>
     <li>Item <%=i %></li>
<% } %>
</ul>
```

(<%= 구분 기호는 *Response.Write*에 대한 약어입니다.)

스크립트 생성

개발자는 WebSnap 기능을 이용하여 스크립트를 자동으로 생성할 수 있습니다.

마법사 템플릿

새 WebSnap 애플리케이션 또는 페이지 모듈을 만들 경우 WebSnap 마법사가 페이지 모듈 템플릿의 초기 컨텐트를 선택하는 데 사용되는 템플릿 필드를 제공합니다. 예를 들어, Default 템플릿은 애플리케이션 제목, 페이지 이름, published로 선언된 페이지에 대한 링크 등을 차례로 표시하는 JScript를 생성합니다.

TAdapterPageProducer

*TAdapterPageProducer*는 HTML 및 JScript를 생성하여 폼과 테이블을 생성합니다. 생성된 JScript는 어댑터 객체를 호출하여 필드 값, 필드 이미지 매개변수, 액션 매개변수 등을 얻어냅니다.

스크립트 편집 및 보기

HTML Result 탭을 사용하여 실행된 스크립트의 HTML 결과를 표시합니다. 브라우저에 결과를 표시하려면 Preview 탭을 사용합니다. HTML Script 탭은 웹 페이지 모듈이 *TAdapterPageProducer*를 사용하는 경우에 이용할 수 있습니다. HTML Script 탭은 *TAdapterPageProducer* 객체에 의해 생성된 HTML 및 JScript를 표시합니다. HTML 폼을 생성하여 어댑터 필드를 표시하고 어댑터 액션을 실행하는 스크립트를 작성하는 방법에 대한 자세한 내용은 이 탭을 참조하십시오.

페이지에 스크립트 포함

템플릿은 파일이나 다른 페이지에서 스크립트를 포함시킬 수 있습니다. 파일에서 스크립트를 포함시키려면 다음 코드 문을 사용합니다.

```
<!-- #include file="filename.html" -->
```

템플릿이 다른 페이지의 스크립트를 포함하는 경우 해당 스크립트는 포함하는 페이지에 의해 분석됩니다. 다음 코드 문을 사용하여 분석되지 않은 page1 내용을 포함시킵니다.

```
<!-- #include page="page1" -- >
```

스크립트 객체

스크립트 객체는 스크립트 명령이 참조할 수 있는 객체입니다. 액티브 스크립트 엔진을 사용하여 객체에 *IDispatch* 인터페이스를 등록하면 객체를 스크립트에 사용할 수 있습니다. 스크립트에 사용할 수 있는 객체는 다음과 같습니다.

표 34.4 스크립트 객체

스크립트 객체	설명
Application	웹 애플리케이션 모듈의 애플리케이션 어댑터를 액세스합니다.
EndUser	웹 애플리케이션 모듈의 엔드 유저 어댑터를 액세스합니다.
Session	웹 애플리케이션 모듈의 세션 객체를 액세스합니다.
Pages	애플리케이션 페이지를 액세스합니다.
Modules	애플리케이션 모듈을 액세스합니다.
Page	현재 페이지를 액세스합니다.
Producer	웹 페이지 모듈의 페이지 프로듀서를 액세스합니다.
Response	WebResponse를 액세스합니다. 태그 변환을 원하지 않을 경우 이 객체를 사용합니다.

표 34.4 스크립트 객체

스크립트 객체	설명
Request	WebRequest를 액세스합니다.
Adapter 객체	현재 페이지에 있는 모든 어댑터 컴포넌트를 제한 없이 참조할 수 있습니다. 다른 모듈에 있는 어댑터는 Modules 객체를 사용하여 참조되어야 합니다.

현재 페이지에 있는 스크립트 객체는 모두 동일한 어댑터를 사용하며 제한 없이 참조될 수 있습니다. 다른 페이지에 있는 스크립트 객체는 다른 페이지 모듈의 일부이며 다른 어댑터 객체를 가집니다. 또한 어댑터 객체의 이름을 앞에 붙여 스크립트 객체 참조를 액세스할 수 있습니다. 예를 들면, 다음과 같습니다.

```
<%= FirstName %>
```

현재 페이지 어댑터의 *FirstName* 속성 컨텐트를 표시합니다. 다음 스크립트 줄은 다른 페이지 모듈에 있는 Adapter1의 *FirstName* 속성을 표시합니다.

```
<%= Adapter1.FirstName %>
```

스크립트 객체에 대한 자세한 내용은 부록의 "WebSnap 서버사이드 스크립트 참조"를 참조하십시오.

요청 및 응답 디스패칭

웹 서버 애플리케이션 개발에 WebSnap을 사용하는 한 가지 이유는 WebSnap 컴포넌트가 HTML 요청과 응답을 자동으로 처리하기 때문입니다. 일반적인 페이지 전송 작업을 위해 이벤트 핸들러를 작성하는 대신 비즈니스 로직과 서버 디자인에 집중할 수 있습니다. 그러나 WebSnap 애플리케이션에서 HTML 요청과 응답을 처리하는 방법을 이해하는 것은 유용할 수 있습니다. 이 단원에서는 이러한 과정에 대한 개요를 설명합니다.

요청을 처리하기 전에 웹 애플리케이션 모듈은 *TWebContext* 타입의 웹 컨텍스트 객체를 초기화합니다. 전역 WebContext 함수를 호출하여 액세스되는 웹 컨텍스트 객체는 요청을 처리하는 컴포넌트에서 사용하는 변수에 대한 전역 액세스 권한을 제공합니다. 예를 들어, 웹 컨텍스트에는 HTTP 요청 메시지와 반환되어야 하는 응답을 나타내는 *TWebRequest* 및 *TWebResponse* 객체가 들어 있습니다.

디스패처 컴포넌트

웹 애플리케이션 모듈의 디스패처 컴포넌트는 애플리케이션의 흐름을 제어합니다. 디스패처는 HTTP 요청을 조사하여 특정 타입의 HTTP 요청 메시지를 처리하는 방법을 결정합니다.

어댑터 디스패처 컴포넌트(*TAdapterDispatcher)*는 컨텐트 필드나 쿼리 필드를 조사하여 어댑터 액션 컴포넌트 또는 어댑터 이미지 필드 컴포넌트를 식별합니다. 어댑터 디스패처는 컴포넌트를 찾아서 해당 컴포넌트에 제어권을 전달합니다.

웹 디스패처 컴포넌트(*TWebDispatcher*)는 특정 타입의 HTTP 요청 메시지의 처리 방법을 알고 있는 액션 항목의 컬렉션(*TWebActionItem* 타입)을 가지고 있습니다. 웹 디스패처는 요청과 일치하는 액션 항목을 찾습니다. 일치하는 액션 항목이 있으면 해당 액션 항목에 제어권을 전달합니다. 또한 웹 디스패처는 요청을 처리할 수 있는 자동 디스패칭 컴포넌트도 조사합니다.

페이지 디스패처 컴포넌트(*TPageDispatcher*)는 *TWebRequest* 객체의 *PathInfo* 속성을 조사하여 등록된 웹 페이지 모듈의 이름을 찾습니다. 디스패처는 웹 페이지 모듈 이름을 찾으면 해당 모듈에 컨트롤을 전달합니다.

어댑터 디스패처 작업

어댑터 디스패처 컴포넌트 (*TAdapterDispatcher*) 는 어댑터 액션 및 필드 컴포넌트를 호출하여 HTML 폼 전송과 동적 이미지 요청을 자동으로 처리합니다.

어댑터 컴포넌트를 사용하여 컨텐트 생성

WebSnap 애플리케이션이 어댑터 액션을 자동으로 실행하고 어댑터 필드에서 동적 이미지를 검색하게 하려면 HTML 컨텐트가 올바르게 구성되어야 합니다. HTML 컨텐트가 올바르게 구성되지 않으면 어댑터 디스패처가 어댑터 액션 및 필드 컴포넌트를 호출하는 데 필요한 정보가 결과 HTTP 요청에 포함되지 않습니다.

HTML 페이지 구성 오류를 줄이기 위해 어댑터 컴포넌트는 HTML 요소의 이름과 값을 나타냅니다. 어댑터 컴포넌트에는 어댑터 필드를 업데이트하도록 디자인된 HTML 폼에 나타내야 하는 히든 필드의 이름과 값을 검색하는 메소드가 있습니다. 일반적으로 페이지 프로듀서는 서버사이드 스크립트를 사용하여 어댑터 컴포넌트에서 이름과 값을 검색한 다음 이 정보를 사용하여 HTML을 생성합니다. 예를 들어, 다음 스크립트는 Adapter1에서 Graphic이라는 필드를 참조하는 <IMG> 요소를 구성합니다.

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>"
alt="<%=Adapter1.Graphic.DisplayText%>">
```

웹 애플리케이션이 스크립트를 분석할 때 HTML src 어트리뷰트(attribute)에는 필드 컴포넌트가 이미지를 검색하는 데 필요한 매개변수와 필드를 식별하는 데 필요한 정보가 들어 있습니다. 따라서 결과물로 작성되는 HTML은 다음과 같을 수 있습니다.

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

브라우저가 웹 애플리케이션에 이 이미지를 검색하라는 HTTP 요청을 보내면 어댑터 디스패처는 모듈 DM에서 "Species No=90090" 매개변수를 사용하여 Adapter1의 Graphic 필드를 호출해야 한다는 것을 알아낼 수 있습니다. 어댑터 디스패처는 Graphic 필드를 호출하여 해당 HTTP 응답을 작성합니다.

다음 스크립트는 Adapter1의 EditRow 액션을 참조하는 <A> 요소를 구성하고 Details라는 페이지에 대한 하이퍼링크를 만듭니다.

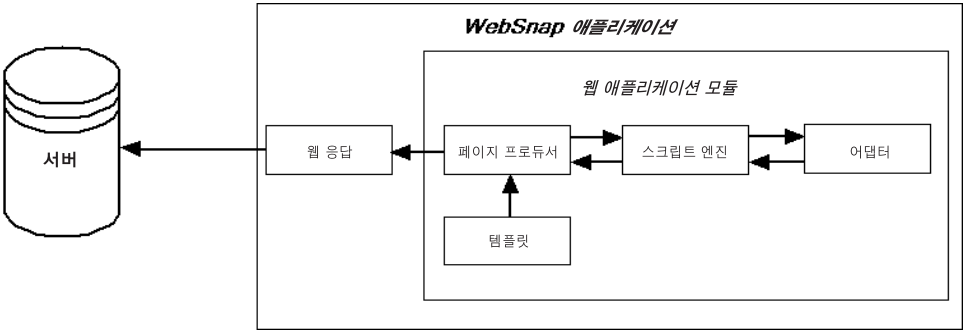
```
<ahref="<%=Adapter1.EditRow.LinkToPage("Details",
Page.Name).AsHREF%>">Edit...</a>
```

결과 HTML은 다음과 같습니다.

```
<ahref="?&_lSpecies
No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

엔드 유저가 이 하이퍼링크를 클릭하면 브라우저가 HTTP 요청을 보냅니다. 어댑터 디스패처는 모듈 DM에서 매개변수 Species No=903010을 사용하여 Adapter1의 EditRow 액션을 호출해야 한다는 것을 알아낼 수 있습니다. 또한 어댑터 디스패처는 액션이 성공적으로 실행되면 Edit 페이지를 표시하고 액션이 실패하면 Grid 페이지를 표시합니다. 그런 다음 EditRow 액션을 호출하여 편집할 행을 찾아서 이름이 Edit인 페이지를 호출하여 HTTP 응답을 생성합니다.
그림 34.12는 어댑터 컴포넌트를 사용하여 컨텐트를 생성하는 방법을 보여 줍니다.

그림 34.12 컨텐트 흐름 생성



어댑터 요청 받기 및 응답 생성

어댑터 디스패처는 클라이언트 요청을 받으면 해당 HTTP 요청에 대한 정보를 보관하기 위해 어댑터 요청 및 어댑터 응답 객체를 생성합니다. 어댑터 요청 및 어댑터 응답 객체는 웹 컨텍스트에 저장되어 요청을 처리하는 동안 액세스할 수 있습니다.

어댑터 디스패처는 액션과 이미지라는 두 가지 타입의 어댑터 요청 객체를 만듭니다. 어댑터 액션을 실행할 경우 액션 요청 객체를 만들고 어댑터 필드에서 이미지를 검색할 경우 이미지 요청 객체를 만듭니다.

어댑터 응답 객체는 어댑터 컴포넌트가 어댑터 액션 또는 어댑터 이미지 요청에 대한 응답을 나타내는 데 사용됩니다. 액션과 이미지 등 두 가지 타입의 어댑터 응답 객체가 있습니다.

액션 요청

액션 요청 객체는 HTTP 요청에서 어댑터 액션을 실행하는 데 필요한 정보를 추출해냅니다. 어댑터 액션을 실행하는 데 필요한 정보 타입에는 다음과 같은 요청 정보가 포함될 수 있습니다.

표 34.5 액션 요청에 있는 요청 정보

요청 정보	설명
컴포넌트 이름	어댑터 액션 컴포넌트를 식별합니다.
어댑터 모드	모드를 정의합니다. 예를 들어, *TDataSetAdapter*는 Edit, Insert 및 Browse 모드를 지원합니다. 어댑터 액션은 모드에 따라 다르게 실행될 수 있습니다.
성공 페이지	액션이 성공적으로 실행될 경우 표시되는 페이지를 식별합니다.
실패 페이지	액션을 실행하는 동안 오류가 발생할 경우 표시되는 페이지를 식별합니다.

표 34.5 액션 요청에 있는 요청 정보

요청 정보	설명
액션 요청 매개변수	어댑터 액션이 필요로 하는 매개변수를 식별합니다. 예를 들어, *TDataSetAdapter* Apply 액션은 업데이트할 레코드를 식별하는 키 값을 포함하고 있습니다.
어댑터 필드 값	HTML 폼이 전송될 때 HTTP 요청에 전달되는 어댑터 필드의 값을 지정합니다. 필드 값에는 엔드 유저가 입력한 새 값, 어댑터 필드의 원래의 값, 업로드된 파일 등이 포함될 수 있습니다.
레코드 키	각 레코드를 고유하게 식별하는 키를 지정합니다.

액션 응답 생성

액션 응답 객체는 어댑터 액션 컴포넌트를 대신하여 HTTP 응답을 생성합니다. 어댑터 액션은 객체 내부의 속성을 설정하거나 액션 응답 객체에 있는 메소드를 호출하여 응답 타입을 나타냅니다. 다음과 같은 속성이 포함됩니다.

- *RedirectOptions* - 리디렉션 옵션은 HTML 컨텐트를 반환하는 대신 HTTP 리디렉션을 수행할지 여부를 나타냅니다.
- *ExecutionStatus* - 상태를 success로 설정하면 디폴트 액션 응답이 Action Request에서 지정된 성공 페이지의 컨텐트가 됩니다.

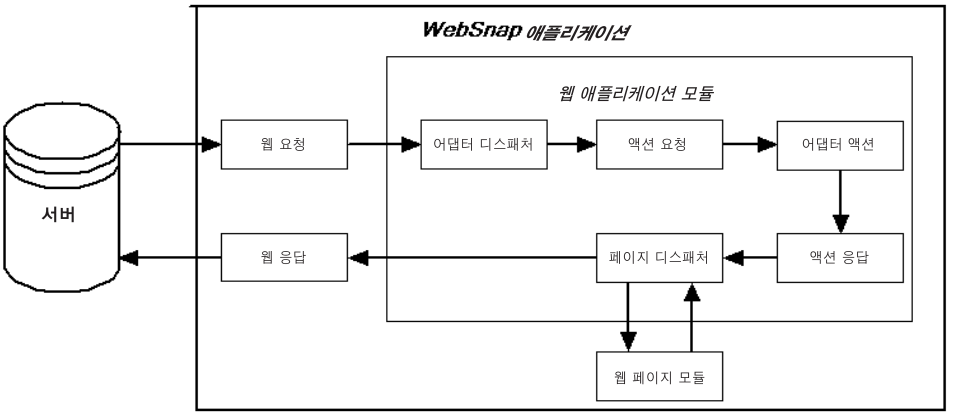
액션 응답 메소드는 다음을 포함합니다.

- *RespondWithPage* - 어댑터 액션은 특정 웹 페이지 모듈이 응답을 생성해야 할 경우에 이 메소드를 호출합니다.
- *RespondWithComponent* - 어댑터 액션은 이 컴포넌트가 들어 있는 웹 페이지 모듈에서 응답을 가져와야 할 경우에 이 메소드를 호출합니다.
- *RespondWithURL* - 어댑터 액션은 응답이 특정 URL에 대한 리디렉션일 경우에 이 메소드를 호출합니다.

페이지로 응답할 때 액션 응답 객체는 페이지 디스패처를 사용하여 페이지 컨텐트를 생성하려고 시도합니다. 페이지 디스패처가 없을 경우 응답 객체는 웹 페이지 모듈을 직접 호출합니다.

그림 34.13은 액션 요청 및 액션 응답 객체가 요청을 처리하는 방법을 보여 줍니다.

그림 34.13 액션 요청 및 응답



이미지 요청

이미지 요청 객체는 HTTP 요청에서 어댑터 이미지 필드가 이미지를 생성하는 데 필요한 정보를 추출해냅니다. Image Request에 의해 표시되는 정보 타입은 다음과 같습니다.

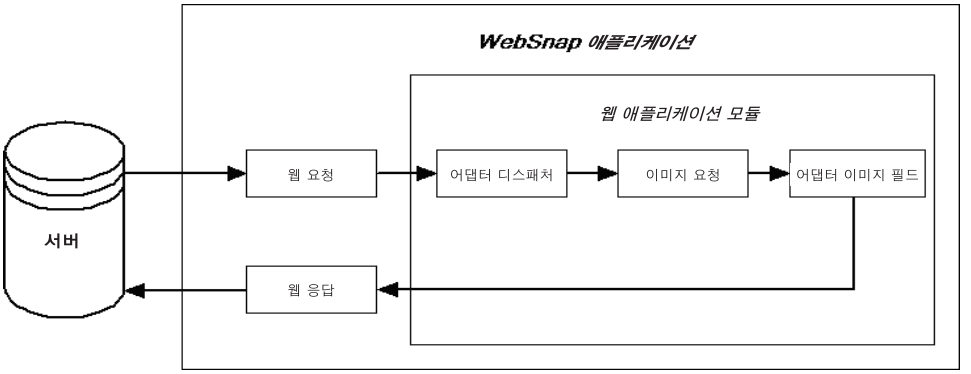
- 컴포넌트 이름 - 어댑터 필드 컴포넌트를 식별합니다.
- 이미지 요청 매개변수 - 어댑터 이미지에 필요한 매개변수를 식별합니다. 예를 들어, *TDataSetAdapterImageField* 객체는 키 값이 있어야 이미지가 포함된 레코드를 식별할 수 있습니다.

이미지 응답

이미지 응답 객체에는 *TWebResponse* 객체가 포함됩니다. 어댑터 필드는 웹 응답 객체에 이미지를 작성하여 어댑터 요청에 응답합니다.

그림 34.14는 어댑터 이미지 필드가 요청에 응답하는 방법을 보여 줍니다.

그림 34.14 이미지 요청에 대한 응답



액션 항목 디스패칭

요청에 응답할 때 웹 디스패처(*TWebDispatcher*)는 액션 항목의 리스트에서 다음 조건을 만족시키는 항목을 검색합니다.

- 대상 URL 요청 메시지의 PathInfo 부분에 일치
- 요청 메시지의 메소드로 지정된 서비스 제공 가능

이 작업은 *TWebRequest* 객체의 *PathInfo* 및 *MethodType* 속성을 액션 항목에 있는 동일한 이름의 속성과 비교하여 수행됩니다.

디스패처는 적절한 액션 항목이 발견되면 해당 액션 항목을 실행시킵니다. 액션 항목이 실행되면 다음 중 하나를 수행합니다.

- 응답 컨텐트를 완성하고 요청이 완전히 처리된 응답 또는 신호를 보냅니다.
- 응답을 추가하고 다른 액션 항목이 작업을 완료할 수 있게 합니다.
- 요청을 다른 액션 항목에 전달합니다.

디스패처가 모든 액션 항목을 검사한 후에도 메시지가 올바르게 처리되지 않으면 디스패처는 액션 항목을 사용하지 않는 특수 등록 자동 디스패칭 컴포넌트를 검사합니다. (이 컴포넌트는 멀티 티어 데이터베이스 애플리케이션 전용입니다.) 요청 메시지가 여전히 완벽하게 처리되지 않을 경우 디스패처는 디폴트 액션 항목을 호출합니다. 디폴트 액션 항목은 대상 URL이나 요청 메소드와 반드시 일치할 필요는 없습니다.

페이지 디스패처 작업

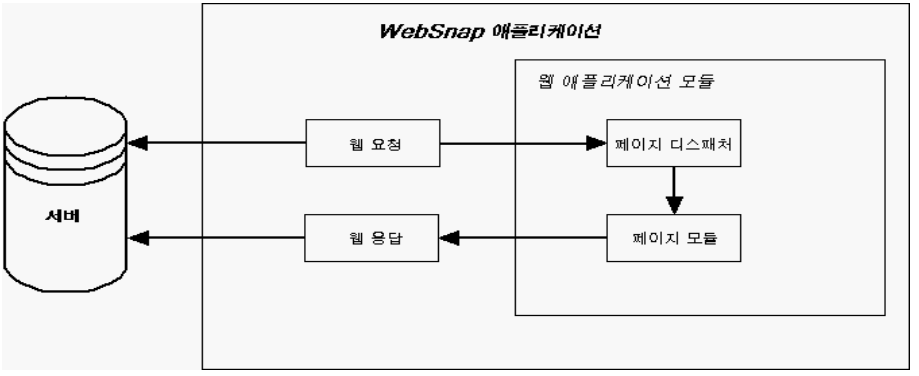
페이지 디스패처는 클라이언트 요청을 받을 경우 대상 URL 요청 메시지의 PathInfo 부분을 검사하여 페이지 이름을 결정합니다. PathInfo 부분이 비어 있지 않을 경우 페이지 디스패처는 PathInfo의 끝 단어를 페이지 이름으로 사용합니다. PathInfo 부분이 비어 있을 경우 페이지 디스패처는 디폴트 페이지 이름을 확인하려고 시도합니다.

페이지 디스패처의 *DefaultPage* 속성에 페이지 이름이 포함되어 있을 경우 페이지 디스패처는 이 이름을 디폴트 페이지 이름으로 사용합니다. *DefaultPage* 속성이 비어 있고 웹 애플리케이션 모듈이 페이지 모듈일 경우 페이지 디스패처는 웹 애플리케이션의 이름을 디폴트 페이지 이름으로 사용합니다.

페이지 이름이 비어 있지 않을 경우 페이지 디스패처는 일치하는 이름이 있는 웹 페이지 모듈을 검색합니다. 페이지 디스패처가 웹 페이지 모듈을 발견하면 해당 모듈을 호출하여 응답을 생성합니다. 페이지 이름이 비어 있거나 페이지 디스패처가 웹 페이지 모듈을 찾지 못할 경우 페이지 디스패처는 예외를 발생합니다.

그림 34.15는 페이지 디스패처가 요청에 응답하는 방법을 보여 줍니다.

그림 34.15 페이지 디스패칭



장

35

XML 문서 작업

XML(Extensible Markup Language)은 구조화된 데이터를 기술하는 마크업 랭귀지입니다. 또한 태그가 표시 특성 대신 정보의 구조를 설명한다는 점을 제외하면 HTML과 비슷합니다. XML 문서에서는 정보를 간단히 텍스트 기반으로 저장하여 쉽게 검색하거나 편집할 수 있습니다. XML 문서는 웹 애플리케이션, 기업 간 통신 등에서 전송 가능한 데이터 표준 형식으로 사용되는 경우도 많습니다.

XML 문서는 데이터 바디에 대한 계층적 뷰를 제공합니다. XML 문서에서 태그는 다음 문서에 보여진 것처럼 각 데이터 요소의 역할 또는 의미를 설명합니다. 이 예제 문서에서는 보유 주식 컬렉션에 대해 설명합니다.

```
<?xml version="1.0" encoding=UTF-8 standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

이 예제에서는 XML 문서에 있는 여러 개의 일반적인 요소들에 대해 설명합니다. 첫 번째 줄은 XML 선언이라는 처리 명령입니다. XML 선언은 옵션이지만 문서에 대한 유용한 정보를 제공하기 때문에 포함시켜야 합니다. 이 예제에서 XML 선언은 해당 문서가 XML 스펙의 버전 1.0을 준수하고, UTF-8 문자 인코딩을 사용하며, 외부 파일을 DTD(Document Type Declaration)로 이용한다는 사실을 알려 줍니다.

<!DOCType> 태그로 시작하는 두 번째 줄은 문서 타입 선언입니다. DTD(Document Type Declaration)는 XML이 문서의 구조를 정의하는 방법이며 문서에 포함된 요소(태그)에 구문 규칙을 부과합니다. 이 예제에서 DTD는 다른 파일(sth.dtd)을 참조합니다. 이 경우 구조는 XML 문서 자체 내에 정의되지 않고 외부 파일에 정의됩니다. XML 문서의 구조를 설명하는 다른 타입의 파일들로는 XDR(Reduced XML Data)과 XSD(XML schema)가 있습니다.

나머지 줄은 단일 루트 노드(<StockHoldings> 태그)를 갖는 계층으로 구성됩니다. 이 계층의 각 노드에는 자식 노드 집합이나 텍스트 값이 들어 있습니다. <Stock> 및 <shares>와 같은 일부 태그에는 어트리뷰트(attribute)가 포함되어 있습니다. 여기서는 태그를 해석하는 방법에 대한 정보를 제공하는 Name=Value 쌍입니다.

XML 문서에서 직접 텍스트 작업을 할 수도 있지만 일반적으로 애플리케이션은 데이터를 분석하고 편집하는 추가 도구를 사용합니다. W3C는 DOM(Document Object Model)이라는 분석된 XML 문서를 나타내는 표준 인터페이스 집합을 정의합니다. 많은 업체들이 DOM 인터페이스를 구현하여 XML 문서를 보다 쉽게 해석하고 편집할 수 있게 해주는 XML 파서를 제공합니다.

C++Builder는 XML 문서 작업을 위한 많은 추가 도구를 제공합니다. 이 도구들은 다른 업체에서 제공하는 DOM 파서를 사용하여 XML 문서를 훨씬 쉽게 작업할 수 있게 해줍니다. 이 장에서는 이러한 도구들에 대해 설명합니다.

참고 이 장에서 설명하는 도구 외에도 C++Builder는 XML 문서를 데이터 패킷으로 변환하여 C++Builder 데이터베이스 아키텍처로 통합하기 위한 도구 및 컴포넌트도 함께 제공합니다. XML 문서를 데이터베이스 애플리케이션으로 통합하는 도구에 대한 자세한 내용은 30장, "데이터베이스 애플리케이션에서 XML 사용"을 참조하십시오.

DOM(Document Object Model) 사용

DOM(Document Object Model)은 분석된 XML 문서를 나타내는 표준 인터페이스 집합입니다. 디폴트로, Microsoft DOM 구현이 제공됩니다. 이 외에도 다른 업체에 의한 추가적인 DOM 구현을 XML 프레임워크로 통합할 수 있는 등록 메커니즘이 있습니다.

XMLDOM 유닛에는 W3C XML DOM level 2 스펙에 정의된 모든 DOM 인터페이스에 대한 선언이 들어 있습니다. 각 DOM 업체는 이들 인터페이스에 대한 구현을 제공합니다.

- Microsoft 구현을 사용하려면 소스 파일에 MSXMLDOM 유닛 헤더를 포함시킵니다. Microsoft 구현이 COM 기반이므로 아직 등록하지 않은 경우에는 msxml.dll 라이브러리를 COM 서버로 등록해야 할 수도 있습니다. Regsvr32.exe를 사용하여 이 DLL을 등록할 수 있습니다.
- 다른 DOM 구현을 사용하려면 *TDOMVendor* 클래스의 자손을 정의하는 유닛을 만들어야 합니다. 자손 클래스에서 업체를 식별하는 문자열을 반환하는 *Description* 메소드와 상위 인터페이스(*IDOMImplementation*)를 반환하는 *DOMImplementation* 메소드의 두 메소드를 오버라이드해야 합니다. 전역 *RegisterDOMVendor* 프로시저를 호출하여 이 업체를 등록해야 합니다. 전역 *UnRegisterDOMVendor* 프로시저를 호출하여 업체 등록을 취소할 수 있습니다. 새 DOMVendor 를 등록하면 애플리케이션은 업체의 등록을 취소하지 않는 한 해당 DOM 구현을 액세스합니다.

일부 업체는 표준 DOM 인터페이스에 대한 확장을 제공합니다. 이러한 확장을 사용할 수 있도록 XMLDOM 유닛은 *IDOMNodeEx* 인터페이스를 정의합니다. *IDOMNodeEx*는 표준 *IDOMNode*에서 상속된 인터페이스이며, 이러한 확장들 중에서 가장 유용한 확장을 포함하고 있습니다.

DOM 인터페이스로 직접 XML 문서를 분석하고 편집할 수 있습니다. *GetDOM* 함수를 호출하여 *IDOMImplementation* 인터페이스를 가져오기만 하면 여기서부터 시작할 수 있습니다.

참고 DOM 인터페이스에 대한 자세한 내용은 XMLDOM 유닛 헤더의 선언부, DOM 업체에서 공급한 설명서 또는 W3C 웹 사이트(www.w3.org)에 있는 스펙을 참조하십시오.

DOM 인터페이스로 직접 작업하는 것보다 특수 XML 클래스를 사용하는 것이 더 편리할 수도 있습니다. 이에 대한 자세한 내용은 아래에 설명되어 있습니다.

XML 컴포넌트 작업

VCL 또는 CLX는 XML 문서 작업을 위한 여러 클래스와 인터페이스를 정의합니다. 또한 XML 문서 로딩, 편집, 저장 등의 프로세스를 단순화합니다.

TXMLDocument 사용

XML 문서 작업을 위한 시작점은 *TXMLDocument* 컴포넌트입니다. 다음 단계에서는 *TXMLDocument*를 사용하여 XML 문서에서 직접 작업하는 방법에 대해 설명합니다.

1. *TXMLDocument* 컴포넌트를 폼 또는 데이터 모듈에 추가합니다. *TXMLDocument*가 컴포넌트 팔레트의 Internet 페이지에 있습니다.
2. *DOMVendor* 속성을 설정하여 XML 문서를 분석하고 편집할 때 컴포넌트가 사용할 DOM 구현을 지정합니다. Object Inspector에 현재 등록된 DOM 업체가 모두 나열됩니다. DOM 구현에 대한 자세한 내용은 35-2페이지의 "DOM(Document Object Model) 사용"을 참조하십시오.
3. 구현에 따라 *ParseOptions* 속성을 설정하여 원본으로 사용하는 DOM 구현이 XML 문서를 분석하는 방법을 구성할 수 있습니다.
4. 기존 XML 문서로 작업할 경우 다음과 같은 방법으로 해당 문서를 지정합니다.
   - XML 문서가 파일로 저장되는 경우 *FileName* 속성을 해당 파일의 이름으로 설정합니다.
   - *XML* 속성을 사용하는 대신 XML 문서를 문자열로 지정할 수 있습니다.
5. *Active* 속성을 **true**로 설정합니다.

활성 *TXMLDocument* 객체가 있으면 그 값을 읽거나 설정하여 노드의 계층을 살펴볼 수 있습니다. 이 계층의 루트 노드는 *DocumentElement* 속성으로 사용할 수 있습니다.

XML 노드 작업

XML 문서가 DOM 구현에 의해 분석된 경우 XML 문서에 표시되는 데이터를 노드의 계층으로 사용할 수 있습니다. 각 노드는 문서에서 태그 요소에 해당합니다. 예를 들어, 다음과 같은 XML이 있다고 가정합니다.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

*TXMLDocument*는 다음과 같은 방법으로 노드의 계층을 생성합니다. 노드 계층의 루트는 *StockHoldings* 노드가 됩니다. *StockHoldings*는 두 *Stock* 태그에 해당하는 두 개의 자식 노드를 가집니다. 다시 두 자식은 각각 *name*, *price*, *symbol* 및 *shares* 등 네 개의 자식 노드를 가집니다. 네 개의 자식 노드는 리프(leaf) 노드 역할을 합니다. 네 개의 자식 노드에 포함되는 텍스트는 각 리프 노드의 값으로 표시됩니다.

참고 이러한 노드 분류는 DOM 구현이 XML 문서에 대한 노드를 생성하는 방법과 약간 다릅니다. 특히, DOM 파서는 모든 태그 요소를 내부 노드로 취급합니다. *name*, *price*, *symbol* 및 *shares* 노드 값에 대해 텍스트 노드 타입의 추가 노드가 만들어집니다. 그러면 이 텍스트 노드가 *name*, *price*, *symbol* 및 *shares* 노드의 자식으로 표시됩니다.

각 노드는 *IXMLNode* 인터페이스를 통해 액세스되며 XML 문서 컴포넌트의 *DocumentElement* 속성 값인 루트 노드로 시작합니다.

노드 값 작업

IXMLNode 인터페이스를 가정할 경우 *IsTextElement* 속성을 선택 표시하여 해당 인터페이스가 내부 노드를 나타내는지 리프 노드를 나타내는지를 확인할 수 있습니다.

- 리프 노드를 나타낼 경우 *Text* 속성을 사용하여 값을 읽거나 설정할 수 있습니다.
- 내부 노드를 나타낼 경우 *ChildNodes* 속성을 사용하여 그 자식 노드를 액세스할 수 있습니다.

예를 들어, 위의 XML 문서를 사용하여 다음과 같은 방법으로 Borland의 주가를 읽을 수 있습니다.

```
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes[0];
AnsiString Price = BorlandStock->ChildNodes->Nodes["price"]->Text;
```

노드 어트리뷰트(attribute) 작업

노드에 어트리뷰트가 포함되어 있는 경우 *Attributes* 속성을 사용하여 해당 속성 작업을 할 수 있습니다. 기존의 어트리뷰트 이름을 지정하여 어트리뷰트 값을 읽거나 변경할 수 있습니다. *Attributes* 속성을 설정할 때 다음과 같은 방법으로 새 어트리뷰트 이름을 지정하여 새 어트리뷰트를 추가할 수 있습니다.

```
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes[0];
BorlandStock->ChildNodes->Nodes["shares"]->Attributes["type"] = "common";
```

자식 노드 추가 및 삭제

AddChild 메소드를 사용하여 자식 노드를 추가할 수 있습니다. *AddChild*는 XML 문서에서 태그 요소에 해당하는 새 노드를 만듭니다. 이러한 노드를 요소 노드라고 합니다.

새 요소 노드를 만들려면 새 태그에 나타나는 이름과 새 노드가 표시되어야 하는 위치(옵션)를 지정하십시오. 예를 들어, 다음 코드는 위의 문서에 새 주식 리스트를 추가합니다.

```
_di_IXMLNode NewStock = XMLDocument1->DocumentElement->AddChild("stock");
NewStock->Attributes["exchange"] = "NASDAQ";
_di_IXMLNode ValueNode = NewStock->AddChild("name");
ValueNode->Text = "Cisco Systems";
ValueNode = NewStock->AddChild("price");
ValueNode->Text = "62.375";
ValueNode = NewStock->AddChild("symbol");
ValueNode->Text = "CSCO";
ValueNode = NewStock->AddChild("shares");
ValueNode->Text = "25";
```

오버로드된 *AddChild* 버전을 사용하여 태그 이름이 정의되는 네임스페이스 URI를 지정할 수 있습니다.

ChildNodes 속성의 메소드를 사용하여 자식 노드를 삭제할 수 있습니다. *ChildNodes*는 노드의 자식을 관리하는 *IXMLNodeList* 인터페이스입니다. *Delete* 메소드를 사용하여 위치 또는 이름에 의해 식별되는 단일 자식 노드를 삭제할 수 있습니다. 예를 들어, 다음 코드는 위 문서에 나열된 마지막 주식을 삭제합니다.

```
_di_IXMLNode StockList = XMLDocument1->DocumentElement;
StockList->ChildNodes->Delete(StockList->ChildNodes->Count - 1);
```

Data Binding 마법사로 XML 문서 고유화

TXMLDocument 컴포넌트 및 이 컴포넌트가 XML 문서의 노드에 사용하는 *IXMLNode* 인터페이스만 사용하여 XML 문서 작업을 하거나 *TXMLDocument*도 사용하지 않고 DOM 인터페이스만 사용하여 작업할 수도 있지만 XML Data Binding 마법사를 사용하면 훨씬 간단하고 읽기 쉬운 코드를 작성할 수 있습니다.

Data Binding 마법사는 XML 스키마 또는 데이터 파일을 가져와서 그 위에 매핑하는 인터페이스 집합을 생성합니다. 인터페이스는 순수 가상 멤버만 포함하는 클래스입니다. 예를 들어, 다음과 같이 나타나는 XML 데이터를 가정해 보십시오.

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

Data Binding 마법사는 다음과 같은 인터페이스를 구현 클래스와 함께 생성합니다.

```
__interface INTERFACE_UUID("{F3729105-3DD0-1234-80e0-22A04FE7B451}")
ICustomer :
    public IXMLNode
{
public:
  virtual int __fastcall Getid(void) = 0 ;
  virtual DOMString __fastcall Getname(void) = 0 ;
  virtual DOMString __fastcall Getphone(void) = 0 ;
  virtual void __fastcall Setid(int Value)= 0 ;
  virtual void __fastcall Setname(DOMString Value)= 0 ;
  virtual void __fastcall Setphone(DOMString Value)= 0 ;
  __property int id = {read=Getid, write=Setid};
  __property DOMString name = {read=Getname, write=Setname};
  __property DOMString phone = {read=Getphone, write=Setphone};
};
```

모든 자식 노드는 이름이 자식 노드의 태그 이름과 일치하고 그 값이 자식 노드의 인터페이스(자식이 내부 노드일 경우) 또는 자식 노드의 값(리프 노드일 경우)과 일치하는 속성으로 매핑됩니다. 모든 노드 어트리뷰트(attribute)도 속성으로 매핑됩니다. 여기서 속성 이름은 어트리뷰트 이름이고 속성 값은 어트리뷰트 값입니다.

마법사는 XML 문서에서 각 태그 요소에 대해 인터페이스 및 자손 구현 클래스를 만들 뿐 아니라 루트 노드에 인터페이스를 가져오는 전역 함수도 만듭니다. 예를 들어, 위의 XML을 루트 노드에 <Customers> 태그가 있는 문서에서 가져온 것일 경우 Data Binding 마법사는 다음과 같은 전역 루틴을 만듭니다.

```
extern PACKAGE _di_ICustomers __fastcall GetCustomers(TXMLDocument
*XMLDoc);

extern PACKAGE _di_ICustomers __fastcall GetCustomers(_di_IXMLDocument
XMLDoc);

extern PACKAGE _di_ICustomers __fastcall LoadCustomers(const WideString
FileName);

extern PACKAGE _di_ICustomers __fastcall NewCustomers(void);
```

Get... 함수는 *TXMLDocument* 인스턴스에 대한 인터페이스 랩퍼 또는 해당 *TXMLDocument* 인스턴스에 대한 포인터를 가져옵니다. Load... 함수는 *TXMLDocument* 인스턴스를 동적으로 만들고 인터페이스 포인터를 반환하기 전에 지정된 XML 파일을 해당 값으로 로드합니다. New... 함수는 새(빈) *TXMLDocument* 인스턴스를 만들고 인터페이스를 루트 노드에 반환합니다.

생성된 인터페이스는 XML 문서의 구조를 보다 직접적으로 반영하기 때문에 코드를 단순화합니다. 예를 들어 다음과 같은 코드를 작성하는 대신,

```
_di_IXMLNode CustIntf = XMLDocument1->DocumentElement;
CustName = CustIntf->ChildNodes->Nodes[0]->ChildNodes->Nodes["name"]-
>Value;
```

코드가 다음과 같이 나타납니다.

```
_di_ICustomers CustIntf = GetCustomers(XMLDocument1);
CustName = CustIntf->Nodes[0]->Name;
```

Data Binding 마법사에서 생성한 인터페이스는 모두 *IXMLNode*의 자손입니다. 따라서 Data Binding 마법사를 사용하지 않았을 경우와 마찬가지로 자식 노드를 추가 및 삭제할 수 있습니다(35-5페이지의 "자식 노드 추가 및 삭제" 참조). 또한 자식 노드가 반복 요소를 나타내고 노드의 모든 자식이 같은 타입일 경우 다른 반복을 추가하기 위한 *Add*와 *Insert*의 두 메소드가 부모 노드에 제공됩니다. 이 두 메소드는 생성할 노드 타입을 지정할 필요가 없기 때문에 *AddChild*를 사용하는 것보다 더 간단합니다.

XML Data Binding 마법사 사용

다음과 같은 방법으로 Data Binding 마법사를 사용합니다.

1. File|New|Other를 선택하고 New 페이지의 하단에서 XML Data Binding 아이콘을 선택합니다.
2. XML Data Binding 마법사가 나타납니다.
3. 마법사의 첫 번째 페이지에서 인터페이스를 생성하려는 XML 문서 또는 스키마를 지정합니다. 이것은 예제 XML 문서, Document Type Definition(.dtd) 파일, Reduced XML Data(.xdr) 파일 또는 XML 스키마(.xsd) 파일이 될 수 있습니다.
4. Options 버튼을 클릭하여 인터페이스 및 구현 클래스를 생성할 때 마법사가 사용할 이름 지정 전략 및 스키마에 정의된 타입과 native 데이터 타입의 디폴트 매핑을 지정합니다.
5. 마법사의 두 번째 페이지로 이동합니다. 이 페이지에서는 문서 또는 스키마에 있는 모든 노드 타입에 대한 자세한 정보를 제공할 수 있습니다. 왼쪽에는 문서에 있는 모든 노드 타입이 표시되는 트리 뷰가 있습니다. 자식이 있는 복잡한 노드의 경우 트리 뷰를 확장하여 자식 요소를 표시할 수 있습니다. 이 트리 뷰에서 노드 하나를 선택하면 다이얼로그 박스의 오른쪽에 해당 노드에 대한 정보가 표시되고 마법사가 해당 노드를 처리하는 방법을 지정할 수 있습니다.
   - Source Name 컨트롤에는 XML 스키마의 노드 타입 이름이 표시됩니다.
   - Source Datatype 컨트롤에는 XML 스키마에 지정된 대로 노드 값의 타입이 표시됩니다.
   - Documentation 컨트롤을 사용하여 노드 사용 또는 목적을 설명하는 주석을 스키마에 추가할 수 있습니다.

- 마법사가 선택된 노드에 대한 코드를 생성하는 경우(즉, 마법사가 인터페이스 및 구현 클래스를 생성하는 복잡한 타입인 경우 또는 마법사가 복잡한 타입 인터페이스에 속성을 생성하는 복잡한 타입의 자식 요소 중 하나인 경우), Generate Binding 체크 박스를 사용하여 마법사가 노드에 대한 코드를 생성할지 여부를 지정할 수 있습니다. Generate Binding을 선택 취소하면 마법사는 복잡한 타입에 대한 인터페이스나 구현 클래스를 생성하지 않거나, 자식 요소 또는 어트리뷰트(attribute)에 대한 속성을 부모 인터페이스에 만들지 않습니다.
- Binding Options 섹션에서는 마법사가 선택된 요소에 대해 생성하는 코드에 영향을 줄 수 있습니다. 모든 노드의 경우 생성된 인터페이스 또는 속성의 이름인 Identifier Name을 지정할 수 있습니다. 또한 인터페이스의 경우 문서의 루트 노드를 나타내는 노드를 지정해야 합니다. 속성을 나타내는 노드의 경우 속성 타입을 지정할 수 있으며, 속성이 인터페이스가 아닐 경우 읽기 전용 속성인지 여부를 지정할 수 있습니다.

6 마법사가 각 노드에 대해 생성할 코드를 지정했으면 세 번째 페이지로 이동합니다. 이 페이지에서 마법사가 코드를 생성하는 방법에 대한 일부 전역 옵션을 선택하고, 생성되는 코드를 미리 보고, 나중에 다시 사용할 수 있도록 선택 사항을 저장하는 방법을 마법사에 지시할 수 있습니다.

- 마법사가 생성하는 코드를 미리 보려면 Binding Summary 리스트에서 인터페이스를 선택하고 Code Preview 컨트롤에서 결과 인터페이스 정의를 봅니다.
- Data Binding Settings를 사용하여 마법사가 선택 내용을 저장하는 방법을 지정합니다. 문서에 연결되는 스키마 파일(다이얼로그 박스의 첫 번째 페이지에 지정된 스키마 파일)에 설정을 주석으로 저장하거나 마법사만 사용하는 독립 스키마 파일을 지정할 수 있습니다.

7 Finish를 클릭하면 Data Binding 마법사가 XML 문서에 있는 모든 노드 타입에 대한 인터페이스 및 구현 클래스를 정의하는 새 유닛을 생성합니다. 또한 *TXMLDocument* 객체를 가져오고 데이터 계층의 루트 노드에 대한 인터페이스를 반환하는 전역 함수를 만듭니다.

XML Data Binding 마법사가 생성하는 코드 사용

마법사가 인터페이스 및 구현 클래스 집합을 생성하면 해당 집합을 사용하여 자신이 마법사에 제공한 문서 또는 스키마의 구조와 일치하는 XML 문서에서 작업할 수 있습니다. 기본 XML 컴포넌트만 사용할 경우 시작점은 컴포넌트 팔레트의 Internet 페이지에 나타나는 *TXMLDocument* 컴포넌트입니다.

XML 문서 작업을 하려면 다음 단계를 따르십시오.

1 XML 문서의 루트 노드에 대한 인터페이스를 가져옵니다. 다음 세 가지 방법 중 하나로 이 작업을 수행할 수 있습니다.

- 폼 또는 데이터 모듈에 *TXMLDocument* 컴포넌트를 놓습니다. *FileName* 속성을 설정하여 *TXMLDocument*를 XML 문서에 연결합니다. 대체 방식으로 런타임 시 *XML* 속성을 설정하여 XML의 문자열을 사용할 수도 있습니다. 그런 다음 코드에서 마법사가 만든 전역 함수를 호출하여 XML 문서의 루트 노드에 대한 인터페이스를 가져옵니다. 예를 들어, XML 문서의 루트 요소가 <StockList> 태그일 경우 디폴트로, 마법사는 *IStockListType* 인터페이스를 반환하는 *GetStockListType* 함수를 생성합니다.

```
XMLDocument1->FileName := "Stocks.xml";
_di_IStockListType StockList = GetStockListType(XMLDocument1);
```

- 생성된 Load... 함수를 호출하여 *TXMLDocument* 인스턴스를 만들고 연결하여 모든 인터페이스를 한 번에 가져옵니다. 예를 들어, 다음과 같은 방법으로 위에 설명된 XML 문서를 사용합니다.

```
_di_IStockListType StockList = LoadStockListType("Stocks.xml");
```

- 애플리케이션에서 모든 데이터를 만들려면 생성된 New... 함수를 호출하여 비어 있는 문서에 대한 *TXMLDocument* 인스턴스를 만듭니다.

```
_di_IStockListType StockList = NewStockListType();
```

2 이 인스턴스에는 해당 루트 요소의 어트리뷰트(attribute)에 해당하는 속성뿐만 아니라 문서 루트 요소의 하위 노드에 해당하는 속성도 있습니다. 이 속성을 사용하여 XML 문서의 계층을 살펴보고, 문서 데이터 수정 등을 수행할 수 있습니다.

3 마법사에서 생성한 인터페이스를 사용하여 변경 내용을 저장하려면 *TXMLDocument* 컴포넌트의 *SaveToFile* 메소드를 호출하거나 *XML* 속성을 읽으십시오.

팁 *TXMLDocument* 객체의 *Options* 속성이 *doAutoSave*를 포함하도록 설정하면 *SaveToFile* 메소드를 명시적으로 호출할 필요가 없습니다.

장

36

웹 서비스 사용

웹 서비스는 인터넷을 통해 게시 및 호출할 수 있는 독립적인 모듈 애플리케이션입니다. 웹 서비스는 제공된 서비스를 설명하는 잘 정의된 인터페이스를 제공합니다. 클라이언트 브라우저용 웹 페이지를 생성하는 웹 서버 애플리케이션과 달리, 웹 서비스는 사용자와 직접 상호 작용하도록 설계된 것이 아닙니다. 대신 클라이언트 애플리케이션을 통해 프로그램에서 웹 서비스에 액세스합니다.

웹 서비스는 클라이언트와 서버를 느슨하게 결합할 수 있도록 설계되었습니다. 즉, 서버를 구현하기 위해 클라이언트에서 특정 플랫폼이나 프로그래밍 랭귀지를 사용할 필요가 없습니다. 웹 서비스는 랭귀지 중립적으로 인터페이스를 정의할 뿐 아니라, 여러 통신 메커니즘을 허용하도록 설계되었습니다.

C++Builder는 SOAP(Simple Object Access Protocol)를 사용하여 웹 서비스를 지원하도록 설계되었습니다. SOAP는 분산 환경에서 정보를 교환하기 위한 표준 lightweight 프로토콜입니다. SOAP는 XML을 사용하여 원격 프로시저 호출을 인코드하며, 일반적으로 HTTP를 통신 프로토콜로 사용합니다. SOAP에 대한 자세한 내용은 다음 주소에서 SOAP 스펙을 참조하십시오.

```
http://www.w3.org/TR/SOAP/
```

참고 웹 서비스를 지원하는 컴포넌트가 SOAP와 HTTP를 사용하도록 만들어져 있긴 하지만 프레임워크는 충분히 일반적이므로 다른 인코딩 및 통신 프로토콜을 사용하기 위해 확장할 수 있습니다.

C++Builder를 사용하면 SOAP 기반 웹 서비스 애플리케이션(서버)을 구축할 수 있을 뿐 아니라, SOAP 인코딩 또는 문서 리터럴 스타일을 사용하는 웹 서비스의 클라이언트를 지원할 수 있습니다. 문서 리터럴 스타일은 .Net 웹서비스에서 사용됩니다.

웹 서비스를 지원하는 컴포넌트는 Windows와 Linux 모두에서 사용 가능하므로 이러한 컴포넌트를 크로스 플랫폼 분산 애플리케이션의 기반으로 사용할 수 있습니다. CORBA를 사용하여 애플리케이션을 배포하는 경우와 같이 특별한 클라이언트 런타임 소프트웨어를 설치할 필요는 없습니다. 이 기술은 HTTP 메시지 기반이므로 다양한 시스템에서 폭넓게 사용할 수 있다는 장점이 있습니다. 웹 서비스 지원은 웹 서버 애플리케이션 아키텍처(Web Broker)를 기반으로 하고 있습니다.

웹 서비스 애플리케이션은 사용 가능한 인터페이스 및 WSDL(Web Service Definition Language) 문서를 사용하여 이러한 인터페이스를 호출하는 방법에 대한 정보를 게시합니다. 서버사이드에서는 애플리케이션이 웹 서비스를 설명하는 WSDL 문서를 게시할 수 있으며 클라이언트사이드에서는 마법사나 명령줄 유틸리티가 게시된 WSDL 문서를 임포트하여 필요한 인터페이스 정의와 연결 정보를 제공할 수 있습니다. 구현할 웹 서비스를 설명하는 WSDL 문서가 이미 있을 경우에는 WSDL 문서를 임포트할 때 서버사이드 코드도 생성할 수 있습니다.

인보커블(invocable) 인터페이스 이해

웹 서비스를 지원하는 서버는 인보커블 인터페이스를 사용하여 생성됩니다. 인보커블 인터페이스는 런타임 타입 정보(RTTI)를 포함하도록 컴파일되는 순수 가상 클래스(순수 가상 메소드만 포함하는 클래스)입니다. 서버에서는 클라이언트에서 들어오는 메소드 호출을 올바르게 마샬링할 수 있도록 이 RTTI를 사용하여 해당 호출을 해석합니다. 클라이언트에서는 인터페이스의 메소드를 호출하기 위해 이 RTTI를 사용하여 메소드 테이블을 동적으로 생성합니다.

인보커블 인터페이스를 만들려면 *delphirtti* 변경자와 함께 **_declspec** 키워드를 사용하여 인터페이스 클래스를 선언해야 합니다. 인보커블 인터페이스의 자손도 인보커블하게 됩니다. 그러나 인보커블 인터페이스가 인보커블하지 않은 인터페이스의 자손일 경우 웹 서비스에서는 인보커블 인터페이스 및 그 자손에 정의된 메소드만 사용할 수 있습니다. 인보커블하지 않은 조상에서 상속된 메소드는 타입 정보를 사용하여 컴파일되지 않으므로 웹 서비스의 일부로 사용될 수 없습니다.

참고 C++Builder의 인터페이스 클래스에 대한 자세한 내용은 13-2페이지의 "상속 및 인터페이스"를 참조하십시오.

sysmac.h 헤더 파일은 인보커블 기본 인터페이스인 *IInvokable*을 정의합니다. 이 인터페이스에서 웹 서비스 서버가 클라이언트에 노출하는 인터페이스를 파생시킬 수 있습니다. *IInvokable*은 자신과 모든 자손이 RTTI를 포함하도록 **_declspec**(delphirtti*)* 옵션을 사용하여 컴파일된다는 점을 제외하고 기본 인터페이스(*IInterface*)와 동일합니다.

예를 들면, 다음 코드에서는 숫자 값 인코딩 및 디코딩을 위한 두 개의 메소드를 포함하는 인보커블 인터페이스를 정의합니다.

```
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}")
IEncodeDecode :
    public IInvokable
{
public:
  virtual double __stdcall EncodeValue(int Value) = 0 ;
  virtual int __stdcall DecodeValue(double Value) = 0 ;
};
```

참고 이 예제에서 선언에 **__interface**가 사용되는 것에 주의하십시오. __interface는 키워드가 아니라 인터페이스 규칙에 사용되는 매크로이며, **class** 키워드에 매핑됩니다. INTERFACE_UUID 매크로는 인터페이스에 GUID(globally unique identifier)를 할당합니다. 인터페이스 클래스는 순수 가상 메소드만 포함합니다.

인보케이션 레지스트리를 사용하여 이 인보커블 인터페이스를 등록해야 웹 서비스 애플리케이션에서 사용할 수 있습니다. 서버에서 인보케이션 레지스트리 항목을 사용하면 인보커 컴포넌트(*THTTPSOAPCppInvoker*)가 인터페이스 인보케이션을 실행하는 데 사용할 구현 클래스를 식별하게 해줍니다. 클라이언트 애플리케이션에서 인보케이션 레지스트리 항목은 원격 인터페이스 객체(*THTTPRio*)가 인보커블 인터페이스를 식별하고 이 인터페이스 호출 방법을 알려 주는 정보를 찾을 수 있게 해줍니다.

일반적으로 웹 서비스 클라이언트나 서버에서는 인보커블 인터페이스를 정의하기 위해 WSDL 문서를 임포트하거나 Web Service 마법사를 사용하여 코드를 생성합니다. 디폴트로, WSDL 임포터 또는 Web Service 마법사가 인터페이스를 생성할 때 그 정의가 웹 서비스와 같은 이름으로 헤더 파일에 추가됩니다. 해당 .cpp 파일에는 인터페이스를 등록할 코드가 포함됩니다. 서버를 작성하는 경우에는 구현 클래스도 포함됩니다. 위에서 설명한 인터페이스에 대한 등록 코드는 다음과 같습니다.

```
static void RegTypes()
{
  InvRegistry()->RegisterInterface(__delphirtti(IEncodeDecode), "", "");
}
#pragma startup RegTypes 32
```

웹 서비스의 인터페이스에는 가능한 모든 웹 서비스의 모든 인터페이스들 사이에서 식별되는 네임스페이스가 있어야 합니다. 이전 예제에서는 인터페이스의 네임스페이스를 제공하지 않습니다. 네임스페이스를 명시적으로 제공하지 않으면 인보케이션 레지스트리가 자동으로 네임스페이스를 생성합니다. 이 네임스페이스는 애플리케이션을 고유하게 식별하는 문자열(*AppNamespacePrefix* 변수), 인터페이스 이름 및 네임스페이스가 정의된 유닛의 이름으로 만들어집니다. 자동으로 생성된 네임스페이스를 사용하지 않으려면 *RegisterInterface* 호출에 두 번째 매개변수를 사용하여 명시적으로 네임스페이스를 지정할 수 있습니다.

참고 인보커블 인터페이스를 정의하는 동일한 헤더 파일을 클라이언트 애플리케이션과 서버 애플리케이션 모두에서 사용하지는 마십시오. 동일한 헤더 파일을 사용할 경우 헤더가 다른 소스 파일에 포함되면 유닛 이름이 해당 소스 파일로 변경되고, 생성된 네임스페이스 이름이 변경되므로 네임스페이스가 일치하지 않을 수 있습니다. 따라서 클라이언트 애플리케이션은 WSDL 문서를 사용하여 웹 서비스를 임포트해야 합니다.

인보커블(invocable) 인터페이스에 넌스칼라(nonscalar) 타입 사용

웹 서비스 아키텍처는 다음과 같은 스칼라 타입의 마샬링에 대한 지원을 자동으로 포함합니다.

- bool
- char
- signed char
- unsigned char
- short
- unsigned short
- int
- unsigned int

- long
- unsigned long
- __int64
- unsigned __int64
- float
- double
- long double
- AnsiString
- WideString
- Currency
- Variant

인보커블 인터페이스에서 이러한 타입을 사용할 때는 특별한 작업이 필요 없습니다. 그러나 다른 타입을 사용하는 속성이나 메소드가 인터페이스에 포함되어 있을 경우 애플리케이션은 리모터블(remotable) 타입 레지스트리를 사용하여 해당 타입을 등록해야 합니다. 리모터블 타입 레지스트리에 대한 자세한 내용은 36-4페이지의 "넌스칼라(nonscalar) 타입 등록"을 참조하십시오.

열거 타입 및 typedef 문을 사용하여 선언된 타입은 리모터블 타입 레지스트리가 필요한 타입 정보를 타입 정의에서 추출할 수 있도록 하는 추가 작업을 필요로 합니다. 이 내용은 36-6페이지의 "typedef로 선언된 타입 및 열거 타입 등록"에 설명되어 있습니다.

sysdyn.h에 정의된 동적 배열 타입은 자동으로 등록되므로 애플리케이션이 이러한 타입에 대한 특별한 등록 코드를 추가할 필요가 없습니다. 동적 배열 타입 중 하나인 *TByteDynArray*는 다른 동적 배열 타입과 같이 배열 요소를 각각 매핑하지 않고, 바이너리 데이터의 'base64' 블록에 매핑하므로 특히 주의해야 합니다.

정적 배열, 구조체 또는 클래스와 같은 다른 타입에 대해서는 해당 타입을 리모터블 클래스에 매핑해야 합니다. 리모터블 클래스는 런타임 타입 정보(RTTI)를 포함하는 클래스입니다. 그러므로 인터페이스가 해당 정적 배열, 구조체 또는 클래스 대신 리모터블 클래스를 사용해야 합니다. 만든 리모터블 클래스는 리모터블 타입 레지스트리를 사용하여 등록해야 합니다.

중요 모든 타입은 인라인이 아닌 **typedef** 문을 사용하여 명시적으로 선언되어야 합니다. 이것은 리모터블 타입 레지스트리가 원시 C++ 타입 이름을 확인하는 데 필요합니다.

넌스칼라(nonscalar) 타입 등록

인보커블 인터페이스가 36-3페이지의 "인보커블(invocable) 인터페이스에 넌스칼라(nonscalar) 타입 사용"에 나열된 기본 스칼라 타입 이외에 다른 타입을 사용하려면 먼저 애플리케이션이 리모터블 타입 레지스트리를 사용하여 타입을 등록해야 합니다. 리모터블 타입 레지스트리에 액세스하려면 소스 파일에 InvokeRegistry.hpp를 포함해야 합니다. 이 헤더가 리모터블 타입 레지스트리에 참조를 반환하는 전역 함수 *RemTypeRegistry*()를 선언합니다.

참고 클라이언트에서는 WSDL 문서를 임포트할 때 리모터블 타입 레지스트리를 사용하여 타입을 등록할 코드가 자동으로 생성됩니다. 서버에서는 리모터블 타입을 사용하는 인터페이스를 등록할 때 이러한 타입이 자동으로 등록됩니다. 자동으로 생성된 값을 사용하지 않고 네임스페이스나 타입 이름을 지정하려면 타입을 등록할 코드를 명시적으로 추가하면 됩니다.

리모터블 타입 레지스트리에는 타입을 등록하는 데 사용할 수 있는 두 개의 메소드인 *RegisterXSInfo*와 *RegisterXSClass*가 있습니다. 첫 번째 메소드(*RegisterXSInfo*)를 사용하여 동적 배열을 등록할 수 있습니다. 두 번째 메소드(*RegisterXSClass*)를 사용하면 다른 타입을 나타내도록 정의되는 리모터블 클래스를 등록할 수 있습니다.

동적 배열을 사용하는 경우 인보케이션 레지스트리는 컴파일러가 생성한 타입 정보에서 필요한 정보를 얻을 수 있습니다. 예를 들면, 인터페이스가 사용할 수 있는 타입은 다음과 같습니다.

```
typedef DynamicArray<TXSDateTime> TDateTimeArray;
```

인보커블 인터페이스를 등록할 때 이 타입이 자동으로 등록됩니다. 그러나 타입이 정의되는 네임스페이스나 타입 이름을 지정하려면 이 동적 배열을 사용하는 인보커블 인터페이스가 등록되는 RegTypes 함수에 다음과 같은 등록 사항을 추가해야 합니다.

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(__arraytypeinfo(TDateTimeArray),
        MyNameSpace, "DTarray", "DTarray");
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

참고 원본으로 사용하는 요소 타입이 모두 동일하게 매핑되는 여러 동적 배열 타입은 사용하지 마십시오. 컴파일러는 이러한 타입을 암시적으로 다른 타입으로 변환될 수 있는 투명한 타입으로 처리하므로 해당 런타임 타입 정보를 구별하지 않습니다.

*RegisterXSInfo*의 첫 번째 매개변수는 등록할 타입의 타입 정보입니다. 두 번째 매개변수는 타입이 정의되는 네임스페이스의 네임스페이스 URI입니다. 이 매개변수를 생략하거나 빈 문자열을 제공하면 레지스트리가 네임스페이스를 생성합니다. 세 번째 매개변수는 순수 C++ 코드에 표시되는 타입 이름입니다. 이 매개변수에 대한 값을 제공해야 합니다. 마지막 매개변수는 WSDL 문서에 표시되는 타입 이름입니다. 이 매개변수를 생략하거나 빈 문자열을 제공하면 레지스트리는 원시 타입 이름(세 번째 매개변수)을 사용합니다.

타입 정보 포인터 대신 클래스 참조를 제공하는 점을 제외하면 리모터블 클래스를 등록하는 것도 유사합니다. 예를 들면, *TXSRecord*라는 리모터블 클래스를 등록하는 코드는 다음과 같습니다.

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TXSRecord), MyNameSpace,
"record", "", false);
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

첫 번째 매개변수는 타입을 나타내는 리모터블 클래스에 대한 클래스 참조입니다. 두 번째 매개변수는 새 클래스의 네임스페이스를 고유하게 식별하는 URI(Uniform Resource Identifier)입니다. 빈 문자열을 제공하면 레지스트리가 URI를 생성합니다. 세 번째 매개변수와 네 번째 매개변수는 클래스가 나타내는 데이터 타입의 원시 및 외부 이름을 지정합니다. 두 번째 매개변수와 네 번째 매개변수를 생략하면 타입 레지스트리가 세 번째 매개변수를 두 값에 사용합니다. 두 매개변수에 빈 문자열을 제공하면 레지스트리가 클래스 이름을 사용합니다. 다섯 번째 매개변수는 클래스 인스턴스 값이 문자열로 전송 가능한지 여부를 나타냅니다. 선택적으로 여기에 표시되지 않은 여섯 번째 매개변수를 추가하여 같은 객체 인스턴스에 대한 여러 개의 참조가 SOAP 패킷에 표시되는 방법을 제어할 수 있습니다.

typedef로 선언된 타입 및 열거 타입 등록

인보커블 인터페이스가 열거 타입을 사용할 경우 **__delphirtti** 함수가 타입 정보를 직접 추출할 수 없습니다. 이 문제를 해결하려면 열거 타입의 타입 정보를 추출하는 데 사용할 수 있는 VCL 스타일의 소유자(holder) 클래스를 생성해야 합니다.

```
class MyEnumType_TypeInfoHolder : public TObject {
  MyEnumType __instanceType;
public:
__published:
  __property MyEnumType __propType = {read=__instanceType };
};
```

이 경우 호출된 *MyEnumType* 열거 타입에서 타입 정보를 추출하기 위해 *MyEnumType_TypeInfoHolder* 클래스가 선언되었습니다. 이 클래스에는 등록하려는 열거 타입의 단일 published 속성인 *__propType*이 있습니다. 소유자(holder) 클래스를 정의하면 전역 *GetClsMemberTypeInfo* 함수를 호출하여 열거 타입의 타입 정보를 얻을 수 있습니다. 다음 코드는 소유자(holder) 클래스에 제공된 열거 타입 등록 방법을 나타낸 것입니다.

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(
          GetClsMemberTypeInfo(__classid(MyEnumType_TypeInfoHolder),
"__propType"),
          MyNameSpace, "MyEnumType");
}
```

*RegisterClsMemberTypeInfo*는 소유자(holder) 클래스의 타입 정보 및 타입 정보를 추출할 해당 타입을 가진 published 속성 이름, 이 두 가지를 매개변수로 사용합니다. 이전 예제에서와 같이 소유자(holder) 클래스에 published 속성이 하나만 있을 경우 두 번째 매개변수를 생략할 수 있습니다.

타입을 기본 C++ 스칼라 타입에 매핑하는 typedef 문을 사용하여 타입을 선언한 경우에도 이와 같이 소유자(holder) 클래스를 사용해야 합니다. (오브젝트 파스칼 타입을 에뮬레이트하는 클래스 타입을 제외하고, 기본 C++ 스칼라 타입은 36-3페이지의 "인보커블(invocable) 인터페이스에 넌스칼라(nonscalar) 타입 사용"에 나열된 타입 중 하나입니다.) 예를 들어, 다음과 같은 타입인 경우,

```
typedef int CardNumber;
```

다음과 같이 소유자(holder) 클래스를 만듭니다.

```
class CardNumberType_TypeInfoHolder : public TObject {
  CardNumber __instanceType;
public:
__published:
  __property CardNumber __propType = {read=__instanceType };
};
```

그런 다음 다음과 같이 등록합니다.

```
RemTypeRegistry()->RegisterXSInfo(GetClsMemberTypeInfo(
          __classid(CardNumber_TypeInfoHolder), "__propType"),
          MyNameSpace, "CardNumber");
```

typedef 문을 사용하여 선언된 다른 타입에 대해 동적 배열 클래스에 매핑하는 경우 *RegisterXSInfo*를 호출하고, 클래스에 매핑하는 경우 *RegisterXSClass*를 호출합니다. 동적 배열 및 클래스 등록에 대한 자세한 내용은 36-4페이지의 "넌스칼라(nonscalar) 타입 등록"을 참조하십시오.

참고 기초가 되는 하나의 타입에 여러 타입을 동시에 매핑하는 typedef 문은 사용하지 마십시오. 이러한 명령문의 런타임 타입 정보는 구별되지 않습니다.

리모터블(remotable) 객체 사용

인보커블 인터페이스에서 복잡한 데이터 타입을 나타내기 위해 클래스를 정의할 때는 *TRemotable*을 기본 클래스로 사용하십시오. 예를 들면, 일반적으로 구조체를 매개변수로 전달할 경우 새 클래스에 있는 구조체의 모든 멤버가 published 속성인 *TRemotable* 자손을 정의하면 됩니다.

타입의 해당 SOAP 인코딩에서 *TRemotable* 자손의 published 속성이 요소 노드로 표시될지 또는 어트리뷰트(attribute)로 표시될지 제어할 수 있습니다. 속성을 어트리뷰트로 만들려면 속성을 정의하는 곳에 AS_ATTRIBUTE의 값을 할당한 stored 지시어를 사용하십시오.

```
__property bool MyAttribute =
                          {read=FMyAttribute, write=FMyAttribute, stored=
AS_ATTRIBUTE;
```

참고 stored 지시어를 포함하지 않을 경우 또는 stored 지시어(AS_ATTRIBUTE를 반환하는 함수일 경우에도)에 다른 값을 할당할 경우 속성이 어트리뷰트가 아니라 노드로 인코딩됩니다.

새 *TRemotable* 자손의 값이 WSDL 문서에서 스칼라 타입을 표시할 경우 대신 *TRemotableXS*를 기본 클래스로 사용해야 합니다. *TRemotableXS*는 새 클래스와 해당 문자열 표현을 서로 변환하기 위해 두 메소드를 추가한 *TRemotable* 자손입니다. *XSToNative* 메소드와 *NativeToXS* 메소드를 오버라이드하여 이 메소드를 구현하십시오.

일반적으로 사용되는 특정 XML 스칼라 타입의 경우 XSBuiltIns 유닛에 이미 리모터블 클래스가 정의되고 등록되어 있습니다. 다음 표는 이러한 리모터블 클래스를 나열한 것입니다.

표 36.1 리모터블 클래스

XML 타입	리모터블 클래스
dateTime timeInstant	TXSDateTime
date	TXSDate
time	TXSTime
duration timeDuration	TXSDuration
decimal	TXSDecimal
hexBinary	TXSHexBinary

리모터블 클래스를 정의한 다음 36-4페이지의 "넌스칼라(nonscalar) 타입 등록"에서 설명한 대로 리모터블 타입 레지스트리를 사용하여 리모터블 클래스가 등록되어야 합니다. 이 등록은 클래스를 사용하는 인터페이스를 등록할 때 서버에서 자동으로 일어납니다. 클라이언트에서는 타입을 정의하는 WSDL 문서를 임포트할 때 클래스를 등록할 코드가 자동으로 생성됩니다.

팁 인보커블 인터페이스를 선언하고 등록하는 유닛을 비롯한 나머지 서버 애플리케이션과 독립적인 다른 유닛에서 *TRemotable* 자손을 구현 및 등록하는 것이 좋습니다. 이렇게 하면 두 개 이상의 인터페이스에 타입을 사용할 수 있습니다.

TRemotable 자손을 사용할 때의 한 가지 문제는 TRemotable 자손이 생성 및 소멸되는 시기입니다. 호출자의 인스턴스가 독립적인 프로세스 공간에 있으므로 서버 애플리케이션은 분명히 이러한 객체의 자체 로컬 인스턴스를 만들어야 합니다. 이 문제를 처리하기 위해 웹 서비스 애플리케이션이 들어오는 요청에 대한 데이터 컨텍스트를 만듭니다. 데이터 컨텍스트는 서버에서 요청을 처리하는 동안 유지되고, 출력 매개변수가 반환 메시지로 마샬링된 후에 해제됩니다. 서버에서 리모터블 객체의 로컬 인스턴스를 만들 때 로컬 인스턴스가 데이터 컨텍스트에 추가되고, 이러한 인스턴스가 데이터 컨텍스트와 함께 해제됩니다. 메소드 호출 후에 리모터블 객체의 인스턴스가 해제되지 않게 할 수도 있습니다. 예를 들면, 객체에 상태 정보가 있을 경우 모든 메시지 호출에 사용되는 단일 인스턴스를 보유하는 것이 더 효과적일 수 있습니다. 리모터블 객체가 데이터 컨텍스트와 함께 해제되지 않게 하려면 *DataContext* 속성을 변경하십시오.

리모터블 객체 예제

이 예제는 다른 경우에 기존 클래스를 사용하는 인보커블 인스턴스에서 매개변수에 대한 리모터블 객체를 만드는 방법을 나타낸 것입니다. 이 예제에서 기존 클래스는 문자열 리스트(*TStringList*)입니다. 예제를 단순화하기 위해 문자열 리스트의 *Objects* 속성을 구현하지 않습니다.

새 클래스는 스칼라가 아니므로 *TRemotableXS*가 아닌 *TRemotable*의 자손입니다. 이 클래스는 클라이언트와 서버 사이에 통신할 문자열 리스트의 모든 속성에 대한 published 속성을 포함합니다. 이들 리모터블 속성은 각각의 리모터블 타입에 해당합니다. 또한 새 리모터블 클래스는 문자열 리스트와의 상호 변환 기능을 제공하는 메소드를 포함합니다.

```
class TRemotableStringList: public TRemotable
{
  private:
    bool FCaseSensitive;
    bool FSorted;
    Classes::TDuplicates FDuplicates;
    System::TStringDynArray FStrings;
  public:
    void __fastcall Assign(Classes::TStringList *SourceList);
    void __fastcall AssignTo(Classes::TStringList *DestList);
__published:
    __property bool CaseSensitive = {read=FCaseSensitive,
write=FCaseSensitive};
    __property bool Sorted = {read=FSorted, write=FSorted};
    __property Classes::TDuplicates Duplicates = {read=FDuplicates,
write=FDuplicates};
    __property System::TStringDynArray Strings = {read=FStrings,
write=FStrings};
}
```

*TRemotableStringList*는 전송 클래스로만 존재한다는 사실에 주의하십시오. 따라서 문자열 리스트의 *Sorted* 속성 값을 전송하기 위해 *Sorted* 속성을 가진 경우에도 저장하는 문자열을 정렬할 필요 없이 문자열 정렬 여부만 기록하면 됩니다. 그러면 구현이 단순해집니다. 문자열 리스트와의 상호 변환 기능을 제공하는 *Assign* 메소드와 *AssignTo* 메소드를 다음과 같이 구현해야 합니다.

```
void __fastcall TRemotableStringList::Assign(Classes::TStringList
*SourceList)
{
  SetLength(Strings, SourceList->Count);
  for (int i = 0; i< SourceList->Count; i++)
    Strings[i] =SourceList->Strings[i];
  CaseSensitive = SourceList->CaseSensitive;
  Sorted = SourceList->Sorted;
  Duplicates = SourceList->Duplicates;
}
void __fastcall TRemotableStringList::AssignTo(Classes::TStringList
*DestList)
{
  DestList->Clear();
  DestList->Capacity = Length(Strings);
  DestList->CaseSensitive = CaseSensitive;
  DestList->Sorted = Sorted;
  DestList->Duplicates = Duplicates;
  for (int i = 0; i< Length(Strings); i++)
    DestList->Add(Strings[i]);
}
```

해당 클래스 이름을 지정하기 위해 새로운 리모터블 클래스를 선택적으로 등록할 수 있습니다. 클래스를 등록하지 않을 경우 클래스를 사용하는 인터페이스를 등록할 때 자동으로 등록됩니다. 마찬가지로 클래스가 사용하는 *TDuplicates* 및 *TStringDynArray* 타입을 등록하지 않고 클래스를 등록할 경우에도 타입이 자동으로 등록됩니다. 이 코드는 *TRemotableStringList* 클래스 등록 방법을 나타낸 것입니다. *TStringDynArray*는 sysdyn.h에 선언된 기본 동적 배열 타입 중 하나이므로 자동으로 등록됩니다. *TDuplicates*와 같은 열거 타입의 명시적 등록에 대한 자세한 내용은 36-6페이지의 "typedef로 선언된 타입 및 열거 타입 등록"을 참조하십시오.

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TRemotableStringList),
MyNameSpace, "stringList", "", false);
}
#pragma startup initServices 32
```

웹 서비스를 지원하는 서버 작성

서버는 인보커블 인터페이스 및 이 인터페이스의 순수 가상 메소드를 구현하는 자손 클래스 외에도 두 개의 컴포넌트인 디스패처와 인보커가 필요합니다. 디스패처(*THTTPSoapDispatcher*)는 들어오는 SOAP 메시지를 수신하여 인보커에 전달합니다. 인보커(*THTTPSoapCppInvoker*)는 SOAP 메시지를 해석하여 자신이 호출하는 인보커블 인터페이스를 식별하고 호출을 실행하며 응답 메시지를 구성합니다.

참고 *THTTPSoapDispatcher* 및 *THTTPSoapCppInvoker*는 SOAP 요청을 가진 HTTP 메시지에 응답하도록 설계되었습니다. 그러나 기본 아키텍처는 충분히 일반적이므로 여러 디스패처와 인보커 컴포넌트를 대신 사용하여 다른 프로토콜을 지원할 수 있습니다.

인보커블 인터페이스와 해당 구현 클래스를 등록하면 디스패처와 인보커가 HTTP 요청 메시지의 SOAP Action 헤더에 있는 인터페이스를 식별하는 메시지를 자동으로 처리합니다.

또한 웹 서비스는 퍼블리셔(*TWSDLHTMLPublish*)를 포함합니다. 퍼블리셔는 애플리케이션에서 웹 서비스를 호출하는 방법을 설명하는 WSDL 문서를 만들어, 들어오는 클라이언트 요청에 응답합니다.

웹 서비스 서버 생성

다음 단계를 사용하여 웹 서비스를 구현하는 서버 애플리케이션을 생성합니다.

1. File|New|Other를 선택하고 WebServices 탭에서 Soap Server Application 아이콘을 더블 클릭하여 SOAP Server Application 마법사를 시작합니다. 마법사는 SOAP 요청에 응답하는 데 필요한 컴포넌트를 포함하는 새 웹 서버 애플리케이션을 만듭니다. SOAP 애플리케이션 마법사 및 마법사가 생성하는 코드에 대한 자세한 내용은 36-11 페이지의 "SOAP 애플리케이션 마법사 사용"을 참조하십시오.
2. SOAP Server Application 마법사를 종료할 때 웹 서비스에 대한 인터페이스를 정의할 것인지 묻습니다. 처음 웹 서비스를 새로 만들 경우 yes를 클릭하면 Add New Web Service 마법사가 나타납니다. 마법사는 웹 서비스에 대한 새 인보커블 인터페이스를 선언하고 등록할 코드를 추가합니다. 생성된 코드를 편집하여 웹 서비스를 정의하고 구현합니다. 다른 인터페이스를 추가하거나 나중에 인터페이스를 정의하려면 File|New|Other를 선택하고 WebServices 탭에서 SOAP Web Service 인터페이스 아이콘을 더블 클릭합니다. Add New Web Service 마법사 사용과 마법사가 생성하는 코드 완료에 대한 자세한 내용은 36-12 페이지의 "새 웹 서비스 추가"를 참조하십시오.
3. WSDL 문서에 이미 정의되어 있는 웹 서비스를 구현하는 경우 Web Services Importer를 사용하여 인터페이스, 구현 클래스 및 애플리케이션이 필요로 하는 등록 코드를 생성할 수 있습니다. 임포터가 구현 클래스에 대해 생성하는 메소드 바디를 채워 넣기만 하면 됩니다. Web Services Importer 사용에 대한 자세한 내용은 36-13페이지의 "Web Services Importer 사용"을 참조하십시오.
4. SOAP 요청 실행을 시도할 때 애플리케이션에 예외가 발생할 경우 예외가 SOAP 오류 패킷에서 자동으로 인코드되어 메소드 호출 결과 대신 반환됩니다. 간단한 오류 메시지 이상의 정보를 전달하려면 인코드되어 클라이언트로 전달되는 예외 클래스를 직접 만들 수 있습니다. 이 내용은 36-14페이지의 "웹 서비스를 위한 사용자 정의 예외 클래스 생성"에 설명되어 있습니다.
5. SOAP Server Application 마법사는 새로운 웹 서비스 애플리케이션에 퍼블리셔 컴포넌트(*TWSDLHTMLPublish)*를 추가합니다. 그러면 애플리케이션이 웹 서비스를 설명하는 WSDL 문서를 클라이언트에 게시할 수 있습니다. WSDL 퍼블리셔에 대한 자세한 내용은 36-15 페이지의 "웹 서비스 애플리케이션을 위한 WSDL 문서 생성"을 참조하십시오.

SOAP 애플리케이션 마법사 사용

웹 서비스 애플리케이션은 특수한 웹 서버 애플리케이션 형식입니다. 그래서 웹 서비스 지원은 Web Broker 아키텍처를 그 기반으로 하고 있습니다. 따라서 Web Broker 아키텍처를 이해하면 SOAP Application 마법사가 생성하는 코드를 이해하는 데 도움이 됩니다. 일반적으로 웹 서버 애플리케이션, 특히 Web Broker에 대한 정보는 32장, "인터넷 서버 애플리케이션 생성" 및 33장, "Web Broker 사용"에서 볼 수 있습니다.

SOAP Application 마법사를 시작하려면 File|New|Other를 선택하고 WebServices 페이지에서 Soap Server Application 아이콘을 더블 클릭하십시오. 웹 서비스에 사용할 웹 서버 애플리케이션 타입을 선택하십시오. 여러 타입의 웹 서버 애플리케이션에 대한 자세한 내용은 32-6 페이지의 "웹 서버 애플리케이션 타입"을 참조하십시오.

마법사는 세 가지 컴포넌트가 들어 있는 웹 모듈을 포함하는 새로운 웹 서버 애플리케이션을 생성합니다.

- 인보커 컴포넌트(*THTTPSOAPCppInvoker*). 인보커는 웹 서비스 애플리케이션에서 SOAP 메시지 및 등록된 인보커블 인터페이스의 메소드를 서로 변환합니다.
- 디스패처 컴포넌트(*THTTPSoapDispatcher*). 디스패처는 들어오는 SOAP 메시지에 자동으로 응답하고 이 메시지를 인보커에게 전달합니다. *WebDispatch* 속성을 사용하여 애플리케이션이 응답하는 HTTP 요청 메시지를 식별할 수 있습니다. 디스패처는 애플리케이션으로 향하는 URL의 경로 부분을 표시하기 위한 *PathInfo* 속성 설정 및 요청 메시지에 대한 메소드 헤더를 표시하기 위한 *MethodType* 속성 설정을 포함합니다.
- WSDL 퍼블리셔(*TWSDLHTMLPublish*). WSDL 퍼블리셔는 인터페이스 및 인터페이스 호출 방법을 설명하는 WSDL 문서를 게시합니다. 클라이언트는 WSDL 문서를 통해 웹 서비스 애플리케이션에서 호출하는 방법을 알 수 있습니다. WSDL 퍼블리셔 사용에 대한 자세한 내용은 36-15페이지의 "웹 서비스 애플리케이션을 위한 WSDL 문서 생성"을 참조하십시오.

SOAP 디스패처와 WSDL 퍼블리셔는 자동 디스패칭 컴포넌트입니다. 즉, 이 컴포넌트들은 *WebDispatch* 속성의 Path Info대로 들어오는 요청을 전달하기 위해 웹 모듈에 자동으로 등록됩니다. 마우스 오른쪽 버튼으로 웹 모듈을 클릭하면 이러한 자동 디스패칭 컴포넌트 외에도 *DefaultHandler*라는 웹 액션 항목 하나가 있는 것을 볼 수 있습니다.

*DefaultHandler*가 디폴트 액션 항목입니다. 즉, 웹 모듈이 핸들러를 찾을 수 없는(Path Info를 연결할 수 없는) 요청을 수신하면 그러한 메시지를 디폴트 액션 항목에 전달합니다.
*DefaultHandler*는 웹 서비스를 설명하는 웹 페이지를 생성합니다. 디폴트 액션을 변경하려면 이 액션 항목의 *OnAction* 이벤트 핸들러를 편집하십시오.

새 웹 서비스 추가

서버 애플리케이션에 새로운 웹 서비스 인터페이스를 추가하려면 File|New|Other를 선택하고 WebServices 탭에서 SOAP Server Interface라는 아이콘을 더블 클릭하십시오.

Add New Web Service 마법사를 사용하여 클라이언트에 노출할 인보커블 인터페이스 이름을 지정할 수 있으며 인터페이스와 인터페이스의 자손 구현 클래스를 선언하고 등록할 코드를 생성합니다. 또한 디폴트로, 마법사는 예제 메소드와 추가 타입 정의를 보여 주는 주석을 생성하며 이 주석은 생성된 파일을 수정하기 시작할 때 유용합니다.

생성된 코드 수정

생성된 유닛의 핸들러 파일에 인터페이스 정의가 표시되며 핸들러 파일에는 마법사를 사용하여 지정한 이름이 있습니다. 예제 메소드를 클라이언트에서 사용할 수 있는 메소드로 바꾸어 인터페이스 선언을 변경할 수 있습니다.

마법사는 *TInvokableClass* 및 인보커블 인터페이스의 자손인 구현 클래스를 생성합니다. 처음 인보커블 인터페이스를 새로 정의하는 경우 구현 클래스의 선언을 수정하여 수정한 내용을 생성된 인보커블 인터페이스에 일치시켜야 합니다.

인보커블 인터페이스와 구현 클래스에 메소드를 추가할 때 메소드가 리모터블 타입만 사용해야 합니다. 리모터블 타입과 인보커블 인터페이스에 대한 자세한 내용은 36-3페이지의 "인보커블(invocable) 인터페이스에 넌스칼라(nonscalar) 타입 사용"을 참조하십시오.

다른 기본 클래스 사용

Add New Web Service 마법사는 *TInvokableClass*의 자손인 구현 클래스를 생성합니다. 이것은 웹 서비스를 구현할 새 클래스를 만드는 가장 쉬운 방법입니다. 그러나 이 생성된 클래스를 다른 기본 클래스를 갖는 구현 클래스로 바꿀 수 있습니다. 예를 들면, 기존 클래스를 기본 클래스로 사용할 수 있습니다. 생성된 구현 클래스를 바꿀 때는 다음과 같은 여러 가지 사항을 고려해야 합니다.

- 새 구현 클래스는 인보커블 클래스의 직접적인 자손이어야 합니다. 인보커블 인터페이스와 해당 구현 클래스 등록에 사용되는 인보케이션 레지스트리는 등록된 각 인터페이스를 구현하는 클래스를 추적하여, 인보커가 인터페이스를 호출해야 할 때 인보커 컴포넌트에서 이 클래스를 사용할 수 있게 합니다. 인터페이스가 클래스 선언에 직접 포함될 경우 인보케이션 레지스트리는 클래스가 인터페이스를 구현하는 것만 감지할 수 있습니다. 기본 클래스와 함께 상속될 경우에는 인터페이스 지원을 감지하지 않습니다.
- 새 구현 클래스는 인터페이스의 일부인 IUnknown 메소드에 대한 지원을 포함해야 합니다. 이것은 당연한 것 같지만 간과되기 쉽습니다. IUnknown에 대한 자세한 내용은 13-4페이지의 "IUnknown을 지원하는 클래스 생성"을 참조하십시오.
- 구현 클래스를 등록하는 생성된 코드가 팩토리 메소드를 포함하도록 변경하여 구현 클래스의 인스턴스를 만들어야 합니다.

마지막 사항에 대한 설명은 간단합니다. 구현 클래스가 *TInvokableClass*의 자손이고, 상속된 생성자를 하나 이상의 매개변수를 포함하는 새 생성자로 바꾸지 않을 경우 인보케이션 레지스트리는 필요할 때 클래스의 인스턴스를 만드는 방법을 알고 있습니다. *TInvokableClass*의 자손이 아닌 구현 클래스를 쓰거나 생성자를 변경할 때는 구현 클래스의 인스턴스를 얻는 방법을 인보케이션 레지스트리에 알려야 합니다.

인보케이션 레지스트리에 팩토리 프로시저를 제공하여 구현 클래스의 인스턴스를 얻는 방법을 알릴 수 있습니다. 구현 클래스가 *TInvokableClass*의 자손이고 상속된 생성자를 사용하더라도 팩토리 프로시저를 제공할 수 있습니다. 예를 들면, 애플리케이션이 인보커블 인터페이스 호출을 수신할 때마다 인보케이션 레지스트리를 요청하여 새 인스턴스를 만드는 대신, 구현 클래스의 단일한 전역 인스턴스를 사용할 수 있습니다.

팩토리 프로시저는 *TCreateInstanceProc* 타입이어야 합니다. 팩토리 프로시저는 구현 클래스의 인스턴스를 반환합니다. 이 프로시저에서 새 인스턴스가 만들어질 경우 인터페이스의 참조 카운트가 0이 될 때 인보케이션 레지스트리가 명시적으로 객체 인스턴스를 해제하지 않으므로 구현 클래스가 자체 해제되어야 합니다. 다음 코드는 팩토리 프로시저가 구현 클래스의 단일 전역 인스턴스를 반환하는 다른 방법을 나타낸 것입니다.

```
void __fastcall CreateEncodeDecode(System::TObject* &obj)
{
  if (!FEncodeDecode)
  {
    FEncodeDecode = new TEncodeDecodeImpl();
    // save a reference to the interface so that the global instance
doesn't free itself
    TEncodeDecodeImpl->QueryInterface(FEncodeDecodeInterface);
  }
  obj = FEncodeDecode;
}
```

참고 이전 예제에서는 *FEncodeDecodeInterface*가 *_di_IEncodeDecode*의 변수입니다.

인보케이션 레지스트리에 클래스를 등록하는 호출에 구현 클래스를 두 번째 인자로 넘겨서, 구현 클래스와 함께 팩토리 프로시저를 등록합니다. 먼저 마법사가 생성한 호출을 찾아 구현 클래스를 등록합니다. 호출을 정의하는 유닛 아래쪽의 RegTypes 메소드에 이 호출이 표시됩니다. 호출은 다음과 같이 나타납니다.

```
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl));
```

팩토리 프로시저를 지정하는 두 번째 매개변수를 이 호출에 추가하십시오.

```
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl),
&CreateEncodeDecode);
```

Web Services Importer 사용

Web Services Importer를 사용하려면 File|New|Other를 선택하고 WebServices 페이지에서 Web Services Importer 아이콘을 더블 클릭합니다. 다이얼로그 박스가 나타나면 WSDL 문서나 XML 파일의 파일 이름을 지정하거나 문서가 게시되는 URL을 입력하십시오.

WSDL 문서가 인증이 필요한 서버 또는 인증이 필요한 프록시 서버를 사용하여 액세스해야 하는 서버에 있을 경우 마법사가 WSDL 문서를 검색할 수 있도록 먼저 사용자 이름과 암호를 제공해야 합니다. 이 정보를 제공하려면 Options 버튼을 클릭하고 알맞은 연결 정보를 제공하십시오.

Next 버튼을 클릭하면 Web Services Importer가 웹 서비스 프레임워크와 호환되는 WSDL 문서에 있는 모든 정의에 대해 생성하는 코드를 보여 줍니다. 즉, Web Services Importer는 SOAP 연결이 있는 포트 타입만 사용합니다. Options 버튼을 클릭하고 원하는 옵션을 선택하여 Importer가 코드를 생성하는 방법을 변경할 수 있습니다.

서버 또는 클라이언트 애플리케이션을 작성할 때 Web Services Importer를 사용할 수 있습니다. 서버를 작성할 때는 Options 버튼을 클릭하고, 해당 다이얼로그 박스에서 Web Services Importer에 서버 코드를 생성하도록 지시하는 옵션에 선택 표시를 합니다. 이 옵션을 선택하면 Web Services Importer가 인보커블 인터페이스에 대한 구현 클래스를 생성하므로 메소드 바디만 채우면 됩니다.

경고 WSDL 문서를 임포트하여 이미 정의되어 있는 웹 서비스를 구현하는 서버를 만드는 경우에도 해당 서비스에 대한 새로운 WSDL 문서를 게시해야 합니다. 임포트된 WSDL 문서와 생성된 구현 사이에는 사소한 차이점이 있을 수 있습니다. 예를 들면, WSDL 문서나 XML 스키마 파일이 C++ 키워드를 식별자로 사용할 경우 생성된 코드가 컴파일될 수 있도록 Web Services Importer가 이름을 자동으로 조정합니다.

Finish를 클릭하면 임포터가 문서에 정의된 작업에 대한 인보커블 인터페이스를 정의 및 등록하고, 문서에 정의된 타입에 대한 리모터블 클래스를 정의 및 등록하는 새 유닛을 만듭니다.

이 방법 대신 명령줄 WSDL 임포터를 사용할 수도 있습니다. 서버에서는 다음과 같이 -S 옵션을 사용하여 명령줄 임포터를 호출합니다.

```
WSDLIMP -S -C -V MyWSDLDoc.wsdl
```

클라이언트 애플리케이션에서는 -S 옵션을 사용하지 않고 명령줄 임포터를 호출합니다.

```
WSDLIMP -C -V MyWSDLDoc.wsdl
```

웹 서비스를 위한 사용자 정의 예외 클래스 생성

SOAP 요청 실행 시도 중에 웹 서비스 애플리케이션에서 예외가 발생하면 SOAP 오류 패킷에서 해당 예외에 대한 정보가 자동으로 인코드되어 메소드 호출 결과 대신 반환됩니다. 그러면 클라이언트 애플리케이션에 예외가 발생합니다.

디폴트로, 클라이언트 애플리케이션에서는 SOAP 오류 패킷의 정보와 함께 *ERemotableException* 타입의 일반 예외를 발생시킵니다. *ERemotableException* 자손을 파생하여 애플리케이션 특정 정보를 추가적으로 전송할 수 있습니다. 예외 클래스에 추가하는 published 속성 값은 SOAP 오류 패킷에 포함되어 클라이언트에서 동일한 예외를 발생할 수 있게 합니다.

ERemotableException 자손을 사용하려면 리모터블 타입 레지스트리를 사용하여 이 자손을 등록해야 합니다. 따라서 *ERemotableException* 자손을 정의하는 유닛에서 InvokeRegistry.hpp를 포함하고, 전역 *RemTypeRegistry* 함수가 반환되는 객체의 *RegisterXSClass* 메소드에 호출을 추가해야 합니다.

또한 클라이언트가 *ERemotableException* 자손을 정의하고 등록할 경우 SOAP 오류 패킷을 수신할 때 모든 속성이 SOAP 오류 패킷에 있는 값으로 설정되어 알맞은 예외 클래스의 인스턴스가 자동으로 발생합니다.

웹 서비스 애플리케이션을 위한 WSDL 문서 생성

애플리케이션에 준비된 웹 서비스를 클라이언트 애플리케이션에 알리기 위해 인보커블 인터페이스를 설명하고 이 인터페이스를 호출하는 방법을 표시하는 WSDL 문서를 게시할 수 있습니다.

참고 임포트된 서비스를 구현한 경우나 C++Builder를 사용하여 클라이언트를 개발한 경우에도 웹 서비스 애플리케이션을 위한 WSDL 문서를 항상 게시해야 합니다.

웹 서비스를 설명하는 WSDL 문서를 게시하려면 웹 모듈에 *TWSDLHTMLPublish* 컴포넌트를 포함하십시오. (SOAP Server Application 마법사는 이 컴포넌트를 디폴트로 추가합니다.) *TWSDLHTMLPublish*는 자동 디스패칭 컴포넌트입니다. 즉, 웹 서비스의 WSDL 문서 리스트를 요청하는 들어오는 메시지에 자동으로 응답합니다. *WebDispatch* 속성을 사용하여 클라이언트가 WSDL 문서 리스트에 액세스하는 데 사용할 URL의 Path Info를 지정합니다. 그러면 웹 브라우저가 *WebDispatch* 속성에서 서버 애플리케이션 위치 뒤에 경로가 덧붙여진 URL을 지정하여, WSDL 문서 리스트를 요청할 수 있습니다. 이 URL은 다음과 비슷하게 나타납니다.

```
http://www.myco.com/MyService.dll/WSDL
```

팁 대신에 물리적인 WSDL 파일을 사용하려면 웹 브라우저에 WSDL 문서를 표시한 다음 이 문서를 저장하여 WSDL 문서 파일을 생성할 수 있습니다.

반드시 웹 서비스를 구현하는 애플리케이션과 동일한 애플리케이션에서 WSDL 문서를 게시할 필요는 없습니다. 단순히 WSDL 문서만 게시하는 애플리케이션을 만들려면 구현 객체를 구현 및 등록하는 코드를 생략하고, 인보커블 인터페이스, 복잡한 타입을 나타내는 리모터블 클래스 및 리모터블 예외를 정의하고 등록하는 코드를 포함하십시오.

디폴트로, WSDL 문서를 게시하면 이 문서는 WSDL 문서를 게시한 URL과 동일한 URL(경로는 다름)에서 서비스를 사용할 수 있음을 나타냅니다. 여러 버전의 웹 서비스 애플리케이션을 배포하는 경우 또는 웹 서비스를 구현하는 애플리케이션이 아닌 다른 애플리케이션에서 WSDL 문서를 게시하는 경우 웹 서비스 위치에 대한 업데이트된 정보를 포함하도록 WSDL 문서를 변경해야 합니다.

URL을 변경하려면 WSDL 관리자를 사용하십시오. 먼저 관리자를 활성화합니다. *TWSDLHTMLPublish* 컴포넌트의 *AdminEnabled* 속성을 **true**로 설정하여 관리자를 활성화합니다. 그러면 브라우저를 사용하여 WSDL 문서 리스트를 표시할 때 문서를 관리할 버튼이 함께 포함됩니다. WSDL 관리자를 사용하여 웹 서비스 애플리케이션을 배포한 위치(URL)를 지정합니다.

웹 서비스용 클라이언트 작성

C++Builder는 SOAP 기반 연결을 사용하는 웹 서비스 호출에 대한 클라이언트사이드 지원을 제공합니다. C++Builder로 작성된 서버나 WSDL 문서로 웹 서비스를 정의하는 다른 모든 서버가 웹 서비스를 제공할 수 있습니다.

WSDL 문서 임포트

웹 서비스를 사용하려면 먼저 애플리케이션이 웹 서비스 애플리케이션에 포함된 인보커블 인터페이스와 타입을 정의하고 등록해야 합니다. 서비스를 정의하는 WSDL 문서나 XML 파일을 임포트해서 이러한 정의를 얻을 수 있습니다. Web Services Importer는 필요한 인터페이스와 타입을 정의하고 등록하는 유닛을 만듭니다. Web Services Importer 사용에 대한 자세한 내용은 36-13페이지의 "Web Services Importer 사용"을 참조하십시오.

인보커블(invocable) 인터페이스 호출

인보커블 인터페이스를 호출하려면 클라이언트 애플리케이션은 인보커블 인터페이스를 정의하는 헤더와 복잡한 타입을 구현하는 리모터블 클래스를 포함해야 합니다.

클라이언트 애플리케이션에서 인보커블 인터페이스의 선언을 포함시킨 후에 해당 인터페이스에 대한 *THTTPRio* 인스턴스를 만드십시오.

```
X = new THTTPRio(NULL);
```

참고 *THTTPRio* 인스턴스를 명시적으로 소멸시키지 않아야 합니다. 위 코드와 같이 *Owner*를 사용하지 않고 이 인스턴스를 만들 경우 인터페이스가 해제될 때 이 인스턴스가 자동으로 해제됩니다. *Owner*를 사용하여 *THTTPRio* 인스턴스를 만들 경우 *Owner*가 이 인스턴스를 해제합니다.

그런 다음 *THTTPRio* 객체에 서버 인터페이스를 식별하고 서버를 찾는 데 필요한 정보를 제공합니다. 다음 두 가지 방법으로 이 정보를 제공할 수 있습니다.

- 웹 서비스의 URL이나 변경할 네임스페이스 및 soap Action 헤더를 표시하지 않으려면 액세스할 웹 서비스 URL만 지정하면 됩니다. *THTTPRio*는 이 URL을 사용하여 인보케이션 레지스트리에 있는 정보를 기준으로 인터페이스 정의와 네임스페이스 및 헤더 정보를 찾습니다. *URL* 속성을 서버 주소로 설정하여 URL을 지정하십시오.

  ```
  X->URL = "http://www.myco.com/MyService.dll/SOAP/IServerInterface";
  ```

- 런타임 시 WSDL 문서에서 동적으로 URL, 네임스페이스 또는 Soap Action 헤더를 찾으려면 *WSDLLocation*, *Service* 속성과 *Port* 속성을 사용할 수 있습니다. 그러면 WSDL 문서에서 필요한 정보가 추출됩니다.

  ```
  X.WSDLLocation = "Cryptography.wsdl";
  X.Service = "Cryptography";
  X.Port = "SoapEncodeDecode";
  ```

서버를 찾고 인터페이스를 식별하는 방법을 지정하면 *QueryInterface* 메소드를 사용하여 인보커블 인터페이스에 대한 인터페이스 포인터를 얻을 수 있습니다. 이 작업을 수행할 때 이 메소드가 메모리에서 관련 인터페이스에 대한 vtable을 동적으로 만들어 인터페이스를 호출할 수 있게 합니다.

```
_di_IEncodeDecode InterfaceVariable;
X->QueryInterface(InterfaceVariable);
if (InterfaceVariable)
{
  Code = InterfaceVariable->EncodeValue(5);
}
```

QueryInterface 호출은 인보커블 인터페이스 자체가 아니라, 인보커블 인터페이스의 *DelphiInterface* 랩퍼를 인수로 사용한다는 점에 유의하십시오.

*THTTPRio*는 인보케이션 레지스트리를 기반으로 인보커블 인터페이스에 대한 정보를 얻습니다. 클라이언트 애플리케이션에 인보케이션 레지스트리가 없거나, 인보커블 인터페이스가 등록되지 않은 경우 *THTTPRio*가 메모리 내 vtable을 생성할 수 없습니다.

경고 *THTTPRio* 에서 얻은 인터페이스를 전역 변수에 할당할 경우 애플리케이션을 종료하기 전에 할당을 NULL로 변경해야 합니다. 예를 들면, 이전 코드 예제의 *InterfaceVariable*이 스택 변수가 아니라 전역 변수일 경우 *THTTPRio* 객체가 해제되기 전에 인터페이스를 해제해야 합니다. 대개 이 코드는 폼이나 데이터 모듈의 *OnDestroy* 이벤트 핸들러로 전달됩니다.

```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  InterfaceVariable = NULL;
}
```

전역 인터페이스 변수를 NULL로 재할당해야 하는 이유는 *THTTPRio*가 vtable을 메모리 내에서 동적으로 생성하기 때문입니다. 인터페이스가 해제될 때 vtable은 계속 존재해야 합니다. 폼이나 데이터 모듈과 함께 인터페이스를 해제하지 않을 경우 종료 시에 전역 변수가 해제될 때 인터페이스가 해제됩니다. *THTTPRio* 객체가 들어 있는 폼이나 모듈 다음에 전역 변수의 메모리가 해제될 수 있습니다. 이 경우 인터페이스가 해제될 때 vtable은 사용할 수 없게 됩니다.

장

37

소켓 작업

이 장에서는 TCP/IP 및 관련 프로토콜을 사용하여 다른 시스템과 통신할 수 있는 애플리케이션을 만들 수 있게 하는 소켓 컴포넌트에 대해 설명합니다. 소켓을 사용하면 기초가 되는 네트워킹 소프트웨어의 세부 사항에 대해 걱정할 필요 없이 다른 컴퓨터에 대한 연결을 통해 읽기와 쓰기를 수행할 수 있습니다. 소켓은 TCP/IP 프로토콜에 기반하여 연결을 제공하지만 UDP(User Datagram Protocol), XNS(Xerox Network System), Digital의 DECnet 또는 Novell의 IPX/SPX 제품군 같은 관련 프로토콜에서도 광범위하게 사용됩니다.

소켓을 사용하면 다른 시스템에 대해 읽기와 쓰기를 수행하는 네트워크 서버 또는 클라이언트 애플리케이션을 작성할 수 있습니다. 서버 또는 클라이언트 애플리케이션은 일반적으로 HTTP(Hypertext Transfer Protocol)나 FTP(File Transfer Protocol)와 같은 단일 서비스 전용으로 개발됩니다. 서버 소켓을 사용하면 이러한 서비스 중 하나를 제공하는 애플리케이션과 해당 서비스를 사용하려는 클라이언트 애플리케이션을 연결할 수 있습니다. 클라이언트 소켓을 사용하면 이러한 서비스 중 하나를 사용하는 애플리케이션과 해당 서비스를 제공하는 서버 애플리케이션을 연결할 수 있습니다.

서비스 구현

소켓은 네트워크 서버 또는 클라이언트 애플리케이션을 작성하는 데 필요한 서비스 중 하나를 제공합니다. HTTP 또는 FTP와 같은 대부분의 서비스에서는 서드파티의 서버를 쉽게 사용할 수 있습니다. 이러한 서비스 중 일부는 운영 체제와 함께 제공되기 때문에 개발자가 직접 서버나 클라이언트 애플리케이션을 작성할 필요가 없습니다. 그러나 서비스가 구현되는 방법을 더 자세히 제어해야 하거나, 애플리케이션과 네트워크 통신 간에 보다 긴밀한 통합이 필요하거나, 또는 특정 서비스에 사용할 수 있는 서버가 없는 경우에는 사용자 정의 서버 또는 클라이언트 애플리케이션을 만들 수 있습니다. 예를 들어, 분산 데이터 집합에서 작업하는 경우 다른 시스템의 데이터베이스와 통신하기 위한 레이어를 만들기를 원할 수 있습니다.

서비스 프로토콜 이해

네트워크 서버 또는 클라이언트를 작성하기 전에 애플리케이션에서 제공하거나 사용 중인 서비스에 대해 이해해야 합니다. 대부분의 서비스에는 네트워크 애플리케이션이 지원하는 표준 프로토콜이 있습니다. HTTP나 FTP는 물론 핑거(finger)나 타임(time)과 같은 표준 서비스를 위한 네트워크 애플리케이션을 작성하는 경우, 우선 다른 시스템과 통신하는 데 사용하는 프로토콜을 이해해야 합니다. 이를 위해 현재 제공 중이거나 사용 중인 특정 서비스에 대한 설명서를 참조하십시오.

다른 시스템과 통신하는 애플리케이션에 대해 새로운 서비스를 제공하려면 먼저, 어떤 메시지를 보내는가? 이러한 메시지들이 어떻게 통합되는가? 정보를 인코드하는 방법은 무엇인가? 등을 고려하여 해당 서비스를 사용할 서버와 클라이언트에 맞는 통신 프로토콜을 디자인해야 합니다.

애플리케이션과 통신

종종 네트워크 서버 또는 클라이언트 애플리케이션은 네트워킹 소프트웨어와 서비스를 사용하는 애플리케이션 간의 레이어를 제공합니다. 예를 들어, HTTP 서버는 컨텐트를 제공하고 HTTP 요청 메시지에 응답하는 웹 서버 애플리케이션과 인터넷 사이에 있습니다.

소켓은 네트워크 서버나 클라이언트 애플리케이션과 네트워킹 소프트웨어 간의 인터페이스를 제공합니다. 개발자는 자신의 애플리케이션과 이를 사용하는 클라이언트 간의 인터페이스를 제공해야 하며, Apache와 같은 표준 서드파티 서버의 API를 복사하거나 새로운 API를 디자인하여 공개할 수 있습니다.

서비스와 포트

대부분의 표준 서비스는 규칙에 따라 특정 포트 번호와 연결됩니다. 포트 번호에 대해서는 나중에 자세히 설명합니다. 여기서는 포트 번호를 해당 서비스에 대한 숫자 코드로 생각합니다.

크로스 플랫폼 애플리케이션에서 사용하는 표준 서비스를 구현하는 경우, Linux 소켓 객체는 해당 서비스에 대한 포트 번호를 찾을 수 있는 메소드를 제공합니다. 새로운 서비스를 제공하려는 경우 /etc/services 파일에서 연결할 포트 번호를 지정할 수 있습니다. services 파일에 대한 자세한 내용은 Linux 설명서를 참조하십시오.

소켓 연결의 타입

소켓 연결을 다음과 같이 세 가지 기본 타입으로 구분하며 세 타입은 소켓의 연결이 시작되는 방법과 로컬 소켓이 연결되는 대상을 반영합니다. 소켓 연결의 세 가지 기본 타입은 다음과 같습니다.

- 클라이언트 연결
- 수신 대기 연결
- 서버 연결

일단 클라이언트 소켓과의 연결이 완료되면 서버 연결과 클라이언트 연결을 구분할 수 없습니다. 서버 연결과 클라이언트 연결의 두 끝점은 기능이 동일하고 같은 타입의 이벤트를 받습니다. 수신 대기 연결은 끝점을 하나만 가지므로 서버 연결 또는 클라이언트 연결과 본질적으로 다릅니다.

클라이언트 연결

클라이언트 연결은 로컬 시스템의 클라이언트 소켓을 원격 시스템의 서버 소켓에 연결합니다. 클라이언트 연결은 클라이언트 소켓에 의해 시작됩니다. 우선 클라이언트 소켓은 연결하고자 하는 서버 소켓에 대한 정보를 제공해야 합니다. 그 다음, 클라이언트 소켓은 서버 소켓을 찾고, 해당 서버를 찾고 나면 연결을 요청합니다. 서버 소켓은 연결을 즉시 완료하지 않을 수도 있습니다. 서버 소켓은 클라이언트 요청의 대기열을 유지하다가 순서가 되면 연결을 완료합니다. 서버 소켓이 클라이언트 연결을 승인하면 서버 소켓은 연결하려는 서버 소켓에 대한 자세한 설명을 클라이언트 소켓으로 보내고 클라이언트가 연결을 완료합니다.

수신 대기 연결

서버 소켓은 클라이언트를 찾지 않습니다. 대신, 서버 소켓은 클라이언트 요청을 수신 대기하는 수동적인 "반 연결"을 구성합니다. 서버 소켓은 수신 대기 연결에 대기열을 연결하며, 대기열은 클라이언트 연결 요청이 들어올 때 요청을 기록합니다. 서버 소켓이 클라이언트 연결 요청을 승인하면 서버 소켓은 새 소켓을 만들어 클라이언트에 연결함으로써 수신 대기 연결을 계속 열어 놓고 다른 클라이언트 요청을 받을 수 있도록 합니다.

서버 연결

서버 연결은 수신 대기 소켓이 클라이언트 요청을 승인할 때 서버 소켓에 의해 구성됩니다. 서버가 연결을 승인하면 클라이언트에 대한 연결을 완료하는 서버 소켓의 정보가 클라이언트로 보내집니다. 클라이언트 소켓이 이 정보를 받아 연결을 완료하면 연결이 설정된 것입니다.

소켓 설명

네트워크 애플리케이션은 소켓을 사용하여 네트워크상의 다른 시스템과 통신할 수 있습니다. 각 소켓은 네트워크 연결의 끝점이라고 할 수 있습니다. 소켓은 다음 항목을 지정하는 주소를 가집니다.

- 소켓이 실행되고 있는 시스템
- 소켓이 이해하는 인터페이스의 타입
- 연결에 대해 사용하는 포트

소켓 연결에 대한 전체 정보에는 연결의 두 끝점에 있는 소켓 주소가 포함됩니다. 두 끝점의 IP 주소나 호스트 주소와 포트 번호를 제공하여 각 소켓 끝점의 주소를 나타낼 수 있습니다.

소켓 연결을 하기 전에 우선 끝점을 구성하는 소켓에 대해 자세한 정보를 제공해야 합니다. 일부 정보는 애플리케이션을 실행하는 시스템에서 얻을 수 있습니다. 예를 들어, 클라이언트 소켓의 로컬 IP 주소는 운영 체제에서 얻을 수 있으므로 설명할 필요가 없습니다.

개발자가 제공해야 할 정보는 사용 중인 소켓 타입에 따라 달라집니다. 클라이언트 소켓은 연결하려는 서버에 대한 정보를 제공해야 합니다. 수신 대기 서버 소켓은 제공하는 서비스를 나타내는 포트 정보를 제공해야 합니다.

호스트 설명

호스트는 소켓이 포함된 애플리케이션을 실행하는 시스템입니다. 다음과 같이 표준 인터넷 도트 표시법으로 된 4개의 숫자(바이트) 값의 문자열인 IP 주소를 제공하여 소켓의 호스트 정보를 나타낼 수 있습니다.

```
123.197.1.2
```

하나의 시스템이 둘 이상의 IP 주소를 지원할 수도 있습니다.

종종 IP 주소를 기억하지 못해 잘못 입력하는 경우가 있습니다. 이 경우에 대한 대안은 호스트 이름을 사용하는 것입니다. 호스트 이름은 URL(Uniform Resource Locator)에서 자주 볼 수 있는 IP 주소의 별칭으로서, 다음과 같이 도메인 이름과 서비스를 포함하는 문자열입니다.

```
http://www.ASite.com
```

대부분의 인트라넷은 인터넷에 있는 시스템의 IP 주소에 호스트 이름을 제공합니다. 명령 프롬프트에서 다음의 명령을 실행하면 IP 주소와 연결된 호스트 이름이 있는 경우 해당 이름을 알 수 있습니다.

```
nslookup IPADDRESS
```

여기서 *IPADDRESS*는 알고자 하는 IP 주소입니다. 로컬 IP 주소와 연결된 호스트 이름이 없어 호스트 이름을 만들려는 경우 네트워크 관리자에게 문의하십시오. 일반적으로 컴퓨터는 *localhost*라는 이름과 IP 번호 127.0.0.1로 자신을 나타냅니다.

서버 소켓은 호스트를 지정할 필요가 없으며, 시스템에서 로컬 IP 주소를 읽을 수 있습니다. 로컬 시스템이 둘 이상의 IP 주소를 지원하는 경우 서버 소켓은 모든 IP 주소에서 클라이언트 요청을 동시에 수신 대기합니다. 서버 소켓이 연결을 승인하면 클라이언트 소켓은 원격 IP 주소를 제공합니다.

클라이언트 소켓은 호스트 이름 또는 IP 주소를 제공하여 원격 호스트를 지정해야 합니다.

호스트 이름과 IP 주소 중에서 선택

대부분의 애플리케이션에서는 호스트 이름을 사용하여 시스템을 지정합니다. 호스트 이름은 기억하기가 쉽고 입력 오류를 쉽게 확인할 수 있습니다. 또한 서버가 특정 호스트 이름과 연결된 시스템 또는 IP 주소를 변경할 수도 있습니다. 클라이언트 소켓은 호스트 이름을 사용하여 호스트 이름이 나타내는 유일한 사이트를 찾을 수 있으며, 이 사이트가 새 IP 주소로 이동한 경우에도 해당 사이트를 찾을 수 있습니다.

호스트 이름을 알 수 없는 경우, 클라이언트 소켓은 IP 주소를 사용하여 서버 시스템을 지정해야 합니다. IP 주소를 제공하면 서버 시스템을 더 빠르게 찾을 수 있습니다. 개발자가 호스트 이름을 제공한 경우에는 소켓이 호스트 이름과 연결된 IP 주소를 검색해야만 서버 시스템을 찾을 수 있습니다.

포트 사용

IP 주소가 소켓 연결의 반대쪽 끝에 있는 시스템을 찾는 데 필요한 정보를 충분히 제공하긴 하지만, 해당 시스템의 포트 번호도 필요합니다. 포트 번호가 없으면 시스템은 한 번에 하나의 연결만 구성할 수 있습니다. 포트 번호는 연결에 각각의 포트 번호를 제공하여 단일 시스템이 여러 연결을 동시에 호스트할 수 있도록 해 주는 고유 식별자입니다.

이 설명서의 앞 부분에서는 포트 번호를 네트워크 애플리케이션으로 구현한 서비스에 대한 숫자 코드라고 설명했습니다. 포트 번호는 사실상 수신 대기 서버 연결을 고정 포트 번호에서 사용할 수 있도록 하여 클라이언트 소켓이 수신 대기 서버를 찾을 수 있도록 하는 규칙일 뿐입니다. 서버 소켓은 자신이 제공하는 서비스와 연결된 포트 번호를 수신 대기합니다. 서버 소켓이 클라이언트 소켓에 대한 연결을 받아들이면 임의의 다른 포트 번호를 사용하는 독립적인 소켓 연결이 만들어집니다. 이렇게 하면 수신 대기 연결이 서비스와 연결된 포트 번호를 계속 수신 대기할 수 있습니다.

다른 소켓이 클라이언트 소켓을 찾을 필요가 없기 때문에 클라이언트 소켓은 임의의 로컬 포트 번호를 사용합니다. 클라이언트 소켓은 연결하려는 서버 소켓의 포트 번호를 지정하여 서버 애플리케이션을 찾을 수 있도록 합니다. 경우에 따라 원하는 서비스 이름을 지정하면 서버 소켓의 포트 번호가 간접적으로 지정됩니다.

소켓 컴포넌트 사용

인터넷 팔레트 페이지에 포함된 세 가지 소켓 컴포넌트를 사용하면 네트워크 애플리케이션에서 다른 컴퓨터에 대한 연결을 구성할 수 있고, 개발자는 이러한 연결을 통해 정보를 읽고 쓸 수 있습니다. 세 가지 소켓 컴포넌트는 다음과 같습니다.

- *TcpServer*
- *TcpClient*
- *UdpSocket*

각 소켓 컴포넌트에는 실제 소켓 연결의 끝점을 나타내는 소켓 객체가 연결됩니다. 소켓 컴포넌트는 소켓 객체를 사용하여 소켓 서버 호출을 캡슐화하므로 애플리케이션에서 연결을 구성하거나 소켓 메시지를 관리하는 세부 사항에 대해 고려하지 않아도 됩니다.

소켓 컴포넌트가 만든 연결의 세부 항목을 변경하려면 소켓 객체의 속성, 이벤트 및 메소드를 사용할 수 있습니다.

연결에 대한 정보 얻기

클라이언트 소켓 또는 서버 소켓에 대한 연결을 완료한 후, 소켓 컴포넌트와 연결된 클라이언트 소켓 또는 서버 소켓 객체를 사용하여 연결에 대한 정보를 얻을 수 있습니다. *LocalHost*와 *LocalPort* 속성을 사용하여 로컬 클라이언트 또는 서버 소켓이 사용하고 있는 주소 및 포트 번호를 확인하거나 *RemoteHost*와 *RemotePort* 속성을 사용하여 원격 클라이언트 또는 서버 소켓이 사용하고 있는 주소 및 포트 번호를 확인합니다. *GetSocketAddr* 메소드를 사용하여 호스트 이름과 포트 번호로부터 유효한 소켓 주소를 만들어냅니다. *LookupPort* 메소드를 사용하면 포트 번호를 찾을 수 있습니다. *LookupProtocol* 메소드를 사용하면 프로토콜 번호를 찾을 수 있습니다. *LookupHostName* 메소드를 사용하면 호스트 컴퓨터의 IP 주소로부터 호스트 이름을 찾을 수 있습니다.

소켓을 드나드는 네트워크 트래픽을 보려면 *BytesSent*와 *BytesReceived* 속성을 사용하십시오.

클라이언트 소켓 사용

애플리케이션을 TCP/IP 또는 UDP 클라이언트로 만들려면 폼이나 데이터 모듈에 *TcpClient* 또는 *UdpSocket* 컴포넌트를 추가합니다. 클라이언트 소켓은 연결하려는 서버 소켓과 서버 서비스를 지정할 수 있게 해줍니다. 원하는 연결에 대한 정보를 제공했다면 클라이언트 소켓 컴포넌트를 사용하여 서버에 대한 연결을 완료할 수 있습니다.

클라이언트 소켓 컴포넌트는 하나의 클라이언트 소켓 객체를 사용하여 연결의 클라이언트 끝점을 나타냅니다.

대상 서버 지정

클라이언트 소켓 컴포넌트에는 연결하고자 하는 서버 시스템과 포트를 지정할 수 있게 해 주는 여러 속성이 있습니다. *RemoteHost* 속성을 사용하여 호스트 이름 또는 IP 주소로 원격 호스트 서버를 지정합니다.

서버 시스템 이외에도 클라이언트 소켓이 연결될 서버 시스템의 포트를 지정해야 합니다. *RemotePort* 속성을 사용하면 대상 서비스의 이름을 지정하여 서버 포트 번호를 직접 또는 간접적으로 지정할 수 있습니다.

연결 구성

일단 클라이언트 소켓 컴포넌트의 속성을 설정하여 연결하고자 하는 서버에 대한 정보를 제공했으면 *Open* 메소드를 호출하여 런타임 시 연결을 구성할 수 있습니다. 애플리케이션을 시작할 때 자동으로 연결을 구성하도록 하려면 디자인 타임에 Object Inspector를 사용하여 *Active* 속성을 **true**로 설정합니다.

연결에 대한 정보 얻기

서버 소켓에 대한 연결을 완료한 후 클라이언트 소켓 컴포넌트와 연결된 클라이언트 소켓 객체를 사용하여 연결에 대한 정보를 얻을 수 있습니다. *LocalHost*와 *LocalPort* 속성으로부터 연결의 끝점을 구성하기 위해 클라이언트와 서버 소켓이 사용한 주소와 포트 번호를 알아냅니다. *Handle* 속성으로부터 소켓 호출 시 사용할 소켓 연결에 대한 핸들을 알아낼 수 있습니다.

연결 끊기

소켓 연결을 통한 서버 애플리케이션과의 통신이 끝나면 *Close* 메소드를 호출하여 연결을 종료할 수 있습니다. 서버에서 연결이 끊길 수도 있는데, 이 경우 개발자는 *OnDisconnect* 이벤트에서 통지를 받습니다.

서버 소켓 사용

애플리케이션을 IP 서버로 만들려면 폼이나 데이터 모듈에 *TcpServer* 또는 *UdpSocket*과 같은 서버 소켓 컴포넌트를 추가합니다. 서버 소켓은 자신이 제공할 서비스를 지정하거나 클라이언트 요청을 수신 대기하는 데 사용할 포트를 지정할 수 있게 해줍니다. 서버 소켓 컴포넌트를 사용하면 클라이언트 연결 요청을 수신 대기하고 승인할 수 있습니다.

서버 소켓 컴포넌트는 하나의 서버 소켓 객체를 사용하여 수신 대기 연결의 서버 끝점을 나타냅니다. 또한 서버 소켓 컴포넌트는 서버가 승인한 클라이언트 소켓에 대한 각 활성 연결의 서버 끝점에도 서버 클라이언트 소켓 객체를 사용합니다.

포트 지정

서버 소켓이 클라이언트 요청을 수신 대기하도록 하려면 우선 서버가 수신 대기할 포트를 지정해야 합니다. *LocalPort* 속성을 사용하여 해당 포트를 지정할 수 있습니다. 서버 애플리케이션이 규칙에 의해 특정 포트 번호와 연결된 표준 서비스를 제공하는 경우, *LocalPort* 속성을 사용하여 서비스 이름을 지정할 수도 있습니다. 포트 번호를 지정할 때 입력 오류가 발생하기 쉽기 때문에 포트 번호 대신 서비스 이름을 사용하는 것이 좋습니다.

클라이언트 요청 수신 대기

일단 서버 소켓 컴포넌트의 포트 번호를 설정했다면 *Open* 메소드를 호출하여 런타임 시 수신 대기 연결을 구성할 수 있습니다. 애플리케이션을 시작할 때 자동으로 수신 대기 연결을 구성하도록 하려면 디자인 타임에 Object Inspector를 사용하여 *Active* 속성을 **true**로 설정합니다.

클라이언트에 연결

수신 대기 서버 소켓 컴포넌트는 클라이언트 연결 요청을 받을 경우 자동으로 연결을 승인합니다. 그 때마다 *OnAccept* 이벤트로 통지를 받습니다.

서버 연결 끊기

수신 대기 연결을 종료하려면 *Close* 메소드를 호출하거나 *Active* 속성을 **false**로 설정합니다. 그러면 클라이언트 애플리케이션에 대한 모든 연결이 종료되고 아직 승인되지 않고 보류 중인 연결이 취소된 다음 수신 대기 연결이 종료됩니다. 따라서 서버 소켓 컴포넌트는 더 이상 새 연결을 승인하지 않습니다.

TCP 클라이언트가 서버 소켓에 대한 각각의 연결을 종료하면 *OnDisconnect* 이벤트는 개발자에게 이를 알립니다.

소켓 이벤트에 응답

소켓을 사용하는 애플리케이션을 작성할 때 프로그램의 어느 곳에서나 소켓에 쓰거나 소켓에서 읽을 수 있습니다. 소켓을 연 다음 프로그램에 있는 *SendBuf, SendStream* 또는 *Sendln* 메소드를 사용하여 소켓에 쓸 수 있습니다. 비슷한 이름을 가진 *ReceiveBuf* 및 *Receiveln* 메소드를 사용하여 소켓에서 읽을 수도 있습니다. *OnSend*와 *OnReceive* 이벤트는 소켓에 쓰거나 소켓에서 읽을 때마다 발생됩니다. 필터링을 위해 OnSend 와 OnReceive 이벤트를 사용할 수 있습니다. 개발자가 읽기 또는 쓰기 작업을 할 때마다 읽기 또는 쓰기 이벤트가 발생됩니다.

연결로부터 오류 메시지를 받으면 클라이언트 소켓과 서버 소켓에서 모두 오류 이벤트를 생성합니다.

또한 소켓 컴포넌트는 연결을 열고 완료하는 과정에서 두 개의 이벤트를 받습니다. 애플리케이션에서 소켓을 여는 방법에 영향을 주려면 *SendBuf* 와 *ReceiveBuf* 메소드를 사용하여 클라이언트 이벤트 또는 서버 이벤트에 응답해야 합니다.

오류 이벤트

연결로부터 오류 메시지를 받으면 클라이언트 소켓과 서버 소켓은 *OnError* 이벤트를 생성합니다. *OnError* 이벤트 핸들러를 작성하면 이러한 오류 메시지에 응답할 수 있습니다.이 이벤트 핸들러에는 다음 사항에 대한 정보가 전달됩니다.

- 오류 통지를 받은 소켓 객체
- 오류 발생 시 소켓이 시도하던 작업
- 오류 메시지에서 제공한 오류 코드

이벤트 핸들러에서 오류에 응답할 수 있고 오류 코드를 0으로 변경하여 소켓이 예외를 발생시키는 것을 방지할 수 있습니다.

클라이언트 이벤트

클라이언트 소켓이 연결되면 다음 이벤트가 발생합니다.

- 이벤트 통지를 위해 소켓이 설정되고 초기화됩니다.
- 서버와 서버 소켓이 만들어지고 나면 *OnCreateHandle* 이벤트가 발생합니다. 이 경우 *Handle* 속성을 통해 사용할 수 있는 소켓 객체는 연결의 반대쪽 끝을 구성할 서버나 클라이언트 소켓에 대한 정보를 제공할 수 있습니다. 이 때 연결에 사용되는 실제 포트를 처음으로 가져올 수 있으며, 실제 포트는 연결을 승인한 수신 대기 소켓의 포트와 다를 수도 있습니다.
- 연결 요청이 서버에 의해 승인되고 클라이언트 소켓에 의해 완료됩니다.
- 연결되면 *OnConnect* 통지 이벤트가 발생합니다.

서버 이벤트

서버 소켓 컴포넌트는 두 가지 타입의 연결, 즉 수신 대기 연결 및 클라이언트 애플리케이션에 대한 연결을 구성합니다. 서버 소켓은 이 두 가지 타입의 연결이 구성되는 동안 이벤트를 받습니다.

수신 대기 시의 이벤트

수신 대기 연결이 구성되기 바로 전에 *OnListening* 이벤트가 발생합니다. *Handle* 속성을 사용하여 소켓이 수신 대기를 위해 열리기 전에 소켓을 변경할 수 있습니다. 예를 들어, 수신 대기를 위해 서버가 사용하는 IP 주소를 제한하려면 *OnListening* 이벤트 핸들러에서 작업을 수행합니다.

클라이언트 연결 시의 이벤트

서버 소켓이 클라이언트 연결 요청을 승인하면 다음 이벤트가 발생합니다.

- *OnAccept* 이벤트가 발생하여 새로운 *TTcpClient* 객체를 이벤트 핸들러로 전달합니다. 이것은 *TTcpClient*의 속성을 사용하여 클라이언트에 대한 연결의 서버 끝점에 대한 정보를 얻을 수 있는 첫 번째 시점입니다.
- 만약 *BlockMode*가 *bmThreadBlocking*이면 *OnGetThread* 이벤트가 발생합니다. 개발자가 작성한 *TServerSocketThread*의 자손을 이용하려는 경우 *OnGetThread* 이벤트 핸들러에서 그 스레드 객체를 생성하고, *TServerSocketThread* 대신 사용할 수 있습니다. 스레드를 초기화하려는 경우 또는 스레드가 연결을 통한 읽기 또는 쓰기를 시작하기 전에 소켓 API를 호출하려는 경우에도 *OnGetThread* 이벤트 핸들러를 사용해야 합니다.
- 클라이언트는 연결을 완료하면 *OnAccept* 이벤트가 발생합니다. 넌블로킹 서버의 경우에는, 이 시점에서 소켓 연결을 통해 읽기 또는 쓰기를 시작할 수도 있습니다.

소켓 연결을 통한 읽기 및 쓰기

다른 시스템에 대한 소켓 연결을 구성하는 이유는 소켓 연결을 통해 다른 시스템의 정보를 읽거나 다른 시스템에 정보를 쓰기 위해서입니다. 읽거나 쓰는 대상이 되는 정보 또는 정보를 읽거나 쓰는 시기는 소켓 연결과 연결된 서비스에 따라 달라집니다.

소켓을 통한 읽기 및 쓰기는 비동기적으로 발생할 수 있기 때문에 네트워크 애플리케이션에서 다른 코드의 실행이 중지되지 않습니다. 이를 넌블로킹 연결이라고 합니다. 또한 블로킹 연결을 구성할 수도 있는데 블로킹 연결에서 애플리케이션은 다음 행의 코드를 실행하기 전에 읽기 또는 쓰기가 완료되기를 기다립니다.

넌블로킹 연결

넌블로킹 연결은 비동기적으로 읽고 쓰기 때문에 데이터 전송이 네트워크 애플리케이션에서 다른 코드의 실행을 중지시키지 않습니다. 클라이언트 또는 서버 소켓에 대한 넌블로킹 연결을 구성하려면 *BlockMode* 속성을 *bmNonBlocking*으로 설정하십시오.

넌블로킹 연결인 경우, 소켓은 읽기 및 쓰기 이벤트를 통해 연결의 다른 끝에 있는 소켓이 정보를 읽거나 쓰려고 할 때 이 사실을 개발자의 소켓에 알립니다.

읽기 및 쓰기 이벤트

넌블로킹 소켓은 연결을 통해 읽거나 써야 할 경우 읽기 및 쓰기 이벤트를 생성합니다. 개발자는 *OnReceive* 또는 *OnSend* 이벤트 핸들러에서 이벤트 통지에 응답할 수 있습니다.

소켓 연결과 연결된 소켓 객체는 이벤트 핸들러를 읽거나 쓰기 위한 매개변수로 제공됩니다. 소켓 객체는 연결을 통해 정보를 읽거나 쓸 수 있도록 여러 메소드를 제공합니다.

소켓 연결에서 정보를 읽으려면 *ReceiveBuf* 또는 *Receiveln* 메소드를 사용합니다. 소켓 연결에 정보를 쓰려면 *SendBuf*, *SendStream* 또는 *Sendln* 메소드를 사용합니다.

블로킹 연결

블로킹 연결인 경우 개발자의 소켓은 연결을 통해 읽기 또는 쓰기를 시작해야 합니다. 소켓은 소켓 연결에서 보내는 통지를 수동적으로 기다릴 수 없습니다. 읽기 및 쓰기가 발생하는 시기를 연결의 끝에서 결정할 경우 블로킹 소켓을 사용합니다.

클라이언트 또는 서버 소켓의 경우 *BlockMode* 속성을 *bmBlocking*으로 설정하여 블로킹 연결을 구성합니다. 클라이언트 애플리케이션이 수행하는 기타 작업에 따라서 개발자는 애플리케이션이 연결을 통한 읽기 또는 쓰기 작업이 완료되기를 기다리는 동안 다른 스레드에서 코드를 계속 실행할 수 있도록, 읽기 또는 쓰기를 위한 새 실행 스레드를 만들 수 있습니다.

서버 소켓의 경우에는 *BlockMode* 속성을 *bmBlocking* 또는 *bmThreadBlocking*으로 설정하여 블로킹 연결을 구성합니다. 블로킹 연결에서는 소켓이 연결을 통한 정보의 읽기 및 쓰기가 완료될 때까지 기다리는 동안 다른 코드의 실행을 모두 보류하기 때문에 *BlockMode*가 *bmThreadBlocking*일 경우에는 항상 서버 소켓 컴포넌트가 모든 클라이언트 연결에 대해 새 실행 스레드를 생성합니다. *BlockMode*가 *bmBlocking*이면 새 연결이 설정될 때까지 프로그램 실행이 중지됩니다.

부

IV

COM 기반 애플리케이션 개발

"COM 기반 애플리케이션 개발"에 포함된 장에서는 Automation 컨트롤러, Automation 서버, ActiveX 컨트롤 및 COM+ 애플리케이션 등의 COM 기반 애플리케이션을 작성하는 데 필요한 개념을 설명합니다.

참고 COM 클라이언트에 대한 지원은 모든 C++Builder 에디션에서 사용할 수 있습니다. 하지만 서버를 만들려면 전문가용 또는 기업용 에디션이 필요합니다.

장

38

COM 기술 개요

C++Builder는 Microsoft의 COM(Component Object Model)에 기반하는 애플리케이션을 쉽게 구현할 수 있는 마법사와 클래스를 제공합니다. 이러한 마법사를 사용하여 애플리케이션 내에서 사용할 수 있는 COM 기반 클래스와 컴포넌트를 만들거나 COM 객체, Automation 서버(Active Server 객체 포함), ActiveX 컨트롤 또는 ActiveForms를 구현하는 완전한 기능의 COM 클라이언트나 서버를 만들 수 있습니다.

참고 ActiveX, COM+ 및 컴포넌트 팔레트의 Servers 탭에 있는 COM 컴포넌트는 CLX 애플리케이션에서 사용할 수 없습니다. 이 기술은 Windows에서만 사용할 수 있으며 다른 플랫폼에서는 사용할 수 없습니다.

COM은 랭귀지에 종속되지 않는 소프트웨어 컴포넌트 모델로서, Windows 플랫폼에서 실행되는 소프트웨어 컴포넌트와 애플리케이션 간의 상호 작용을 가능하게 합니다. COM의 핵심적인 요소는 명확히 정의된 인터페이스를 통해 컴포넌트 간, 애플리케이션 간 및 클라이언트와 서버간의 통신이 가능하다는 점입니다. 인터페이스는 런타임 시 클라이언트가 지원하는 COM 컴포넌트 기능을 요청하는 방법을 제공합니다. 컴포넌트에 추가 기능을 제공하려면 해당 기능에 대한 추가 인터페이스를 제공하기만 하면 됩니다.

애플리케이션은 해당 애플리케이션과 같은 컴퓨터에 있는 COM 컴포넌트 인터페이스에 액세스할 수도 있고 DCOM(Distributed COM)이라는 메커니즘을 사용하여 네트워크상의 다른 컴퓨터에 있는 COM 컴포넌트 인터페이스에 액세스할 수도 있습니다. 클라이언트, 서버 및 인터페이스에 대한 자세한 내용은 38-2페이지의 "COM 애플리케이션 요소"를 참조하십시오.

이 장에서는 Automation 및 ActiveX 컨트롤 구축의 기초가 되는 기술에 대한 개념적인 내용을 설명합니다. 다음 장에서는 C++Builder에서의 Automation 객체 및 ActiveX 컨트롤 작성에 대한 상세한 정보를 제공합니다.

스펙 및 구현에 대한 COM

COM은 스펙이면서 동시에 구현입니다. COM 스펙은 객체가 만들어지는 방법 및 다른 객체와 통신하는 방법을 정의합니다. COM 객체는 이 스펙에 따라 다양한 랭귀지로 작성될 수 있으며 다양한 프로세스 공간과 플랫폼에서 실행될 수 있습니다. 객체는 지정된 스펙을 준수하는 한 서로 통신할 수 있습니다. 이를 통해 개발자는 한 컴포넌트로서의 레거시 코드를 객체 지향 랭귀지로 구현된 새 컴포넌트와 통합할 수 있습니다.

지정된 스펙을 지원하는 다양한 핵심 서비스를 제공하는 Win32 하위 시스템에서는 COM 구현이 기본으로 제공됩니다. COM 라이브러리에는 COM 객체의 핵심 기능을 정의하는 일련의 표준 인터페이스 및 COM 객체를 만들고 관리하기 위한 용도로 설계된 소규모 집합의 API 함수가 포함되어 있습니다.

애플리케이션에서 C++Builder 마법사와 VCL 객체를 사용하는 것은 C++Builder를 COM 스펙의 구현으로 사용하는 것입니다. 또한 C++Builder는 Active Document처럼 직접 구현되지 않는 기능에 대해 COM 서비스에 대한 일부 랩퍼를 제공하기도 합니다.

참고 C++Builder는 클래스와 매크로에 의해 수정된 Microsoft ATL(Active Template Library)을 사용하여 COM 스펙을 준수하는 객체를 구현합니다.

COM 확장

COM은 지속적인 발전을 통해 기본 COM 서비스 이상으로 확장되었습니다. COM은 Automation, ActiveX 컨트롤, Active Document 및 Active Directory와 같은 다른 기술에 대한 기초가 됩니다. COM 확장에 대한 자세한 내용은 38-10페이지의 "COM 확장"을 참조하십시오.

또한 대규모의 분산 환경에서 작업하는 경우에는 트랜잭션 COM 객체를 만들 수 있습니다. Windows 2000 이전에는 이러한 객체가 아키텍처 측면에서 COM의 일부가 아니라 MTS(Microsoft Transaction Server) 환경에서 실행되었습니다. Windows 2000이 출시되면서 이러한 지원이 COM+에 통합되었습니다. 트랜잭션 객체에 대한 자세한 내용은 44장, "MTS 객체 또는 COM+ 객체 생성"을 참조하십시오.

C++Builder에는 앞에서 설명한 기술을 C++Builder 환경에 통합하는 애플리케이션을 쉽게 구현할 수 있는 마법사가 포함되어 있습니다. 자세한 내용은 38-19페이지의 "마법사로 COM 객체 구현"을 참조하십시오.

COM 애플리케이션 요소

COM 애플리케이션을 구축하는 경우에는 다음을 제공해야 합니다.

COM 인터페이스 객체가 클라이언트에 외부적으로 서비스를 노출하기 위해 사용하는 방법입니다. COM 객체는 연관된 각 메소드와 속성 집합에 대한 인터페이스를 제공합니다. 단, COM 속성은 VCL 객체의 속성과 동일하지 않습니다. COM 속성은 항상 읽기 및 쓰기 액세스 메소드를 사용하며 **__property** 키워드를 사용하여 선언되지 않습니다.

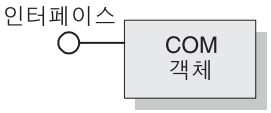
COM 서버 COM 객체에 대한 코드를 포함하는 EXE, DLL 또는 OCX 등의 모듈입니다. 객체 구현은 서버에 있으며 COM 객체는 하나 이상의 인터페이스를 구현합니다.

COM 클라이언트 요청된 서비스를 서버에서 가져오기 위해 인터페이스를 호출하는 코드입니다. 클라이언트는 인터페이스를 통해 서버에서 어떤 서비스를 가져올지는 알고 있으나 서버가 서비스를 제공하는 구체적인 방법에 대해서는 모릅니다. C++Builder를 사용하면 Word 문서나 PowerPoint 슬라이드처럼 COM 서버를 컴포넌트 팔레트의 컴포넌트로 설치할 수 있으므로 클라이언트 생성 과정이 수월해집니다. 이렇게 하면 서버에 접속하여 Object Inspector를 통해 해당 이벤트를 연결할 수 있습니다.

COM 인터페이스

COM 클라이언트는 COM 인터페이스를 통해 객체와 통신합니다. 인터페이스는 서비스 프로바이더(서버 객체)와 그 클라이언트 간의 통신을 제공하는, 논리적 또는 의미상으로 관련된 루틴 그룹입니다. 그림 38.1은 COM 인터페이스를 설명하는 가장 일반적인 방법입니다.

그림 38.1 COM 인터페이스



예를 들어, 모든 COM 객체는 *IUnknown*이라는 기본 인터페이스를 구현해야 합니다. 클라이언트는 *IUnknown* 내의 *QueryInterface*라는 루틴을 통해 서버가 구현한 다른 인터페이스를 요청할 수 있습니다.

객체는 인터페이스마다 서로 다른 기능을 구현하는 여러 인터페이스를 가질 수 있습니다. 인터페이스는 객체가 서비스를 제공하는 방법과 위치에 대한 상세한 구현 정보를 제공하지 않고도 클라이언트에 서비스를 전달하는 방법을 제공합니다.

COM 인터페이스의 핵심적인 특징은 다음과 같습니다.

- 인터페이스는 게시하고 나면 변경되지 않습니다. 인터페이스에 의존하여 특정 함수 집합을 제공할 수 있으며 추가 인터페이스를 통해 추가 기능을 제공할 수 있습니다.
- 일반적으로 COM 인터페이스 식별자는 *IMalloc* 또는 *IPersist*처럼 대문자 I와 인터페이스를 정의하는 상징적 이름으로 시작합니다.
- 인터페이스에는 128비트의 임의로 생성된 숫자인 **GUID(Globally Unique Identifier)**라는 고유한 ID가 있어야 합니다. 인터페이스 GUID를 **인터페이스 식별자(IID)**라고 하며 이를 통해 한 제품 또는 여러 제품의 서로 다른 버전 간에 이름 충돌을 방지할 수 있습니다.
- 인터페이스는 랭귀지와 무관합니다. 즉 포인터 구조체를 지원하는 모든 랭귀지를 사용하여 COM 인터페이스를 구현할 수 있으며 명시적 또는 암시적으로 포인터를 통해 함수를 호출할 수 있습니다.
- 인터페이스는 객체가 아니라 객체에 액세스할 수 있는 방법을 제공할 뿐입니다. 따라서 클라이언트는 데이터에 직접 액세스하는 것이 아니라 인터페이스 포인터를 통해 액세스합니다 . Windows 2000 은 인터셉터라는 간접 참조 레이어를 추가적으로 제공하며 이를 통해 just-in-time 활성화 및 객체 풀링과 같은 COM+ 기능을 제공합니다.
- 인터페이스는 항상 *IUnknown*이라는 기본 인터페이스로부터 상속됩니다.
- COM은 대리자를 통해 인터페이스를 리디렉션하여 인터페이스 메소드 호출이 스레드, 프로세스, 네트워크 시스템 간에 호출되도록 할 수 있으며 이 때 클라이언트나 서버 객체가 리디렉션에 대해 인식하지 않고도 모든 작업이 수행됩니다. 자세한 내용은 38-6페이지의 "in-process, out-of-process 및 원격 서버"를 참조하십시오.

기본 COM 인터페이스, IUnknown

모든 COM 객체는 기본 인터페이스 타입, *IInterface*에 대한 **typedef**인 *IUnknown*이라는 기본 인터페이스를 지원해야 합니다. *IUnknown*에 포함되어 있는 루틴은 다음과 같습니다.

QueryInterface	객체가 지원하는 다른 인터페이스에 대한 포인터를 제공합니다.
AddRef 및 Release	객체의 수명을 추적하여 클라이언트가 더 이상 해당 서비스를 원하지 않으면 객체가 스스로 삭제되도록 하는 간단한 참조 카운팅 메소드입니다.

클라이언트는 *IUnknown* 메소드의 *QueryInterface*를 통해 다른 인터페이스에 대한 포인터를 얻습니다. *QueryInterface*는 서버 객체의 모든 인터페이스에 대해 알고 있으므로 요청된 인터페이스에 대한 포인터를 클라이언트에 제공할 수 있습니다. 인터페이스에 대한 포인터를 받은 클라이언트는 인터페이스의 모든 메소드를 호출할 수 있습니다.

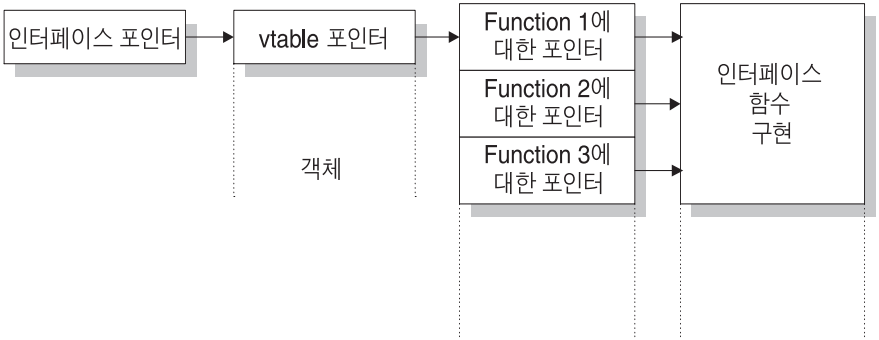
객체는 *IUnknown* 메소드 내의 간단한 참조 카운팅 메소드인 *AddRef* 및 *Release*를 통해 자체 수명을 추적합니다. 객체의 참조 카운트가 0이 아닌 객체는 메모리 내에 남게 됩니다. 참조 카운트가 0 이 되면 인터페이스 구현을 통해 원본으로 사용하는 객체를 안전하게 삭제할 수 있습니다.

COM 인터페이스 포인터

인터페이스 포인터는 인터페이스 내의 각 메소드 구현을 가리키는 객체 인스턴스에 대한 포인터이며, 이러한 메소드에 대한 포인터 배열인 **vtable**을 통해 이 구현에 액세스합니다. Vtable은 C++의 가상 함수를 지원하는 데 사용되는 메커니즘과 유사합니다. 이러한 유사성으로 인해 컴파일러가 C++ 클래스의 메소드에 대한 호출을 해결하는 것과 동일한 방법으로 인터페이스의 메소드에 대한 호출을 해결할 수 있습니다.

vtable은 객체 클래스의 모든 인스턴스에서 공유되므로 객체 코드는 각 객체 인스턴스에 대해 private 데이터를 포함하는 두 번째 구조체를 할당합니다. 그러면 다음 다이어그램에서 볼 수 있듯이 클라이언트의 인터페이스 포인터는 vtable에 대한 *포인터의* 포인터가 됩니다.

그림 38.2 인터페이스 vtable



Windows 2000 이상 버전에서는 객체가 COM+에서 실행되는 경우 인터페이스 포인터와 vtable 포인터 간에 간접 참조의 추가 레벨이 제공됩니다. 클라이언트에서 사용할 수 있는 인터페이스 포인터는 인터셉터를 가리키고 이 인터셉터가 다시 vtable을 가리킵니다. 이렇게 하면 COM+가 just-in-time 활성화같은 서비스를 제공할 수 있으므로 클라이언트와 관계 없이 동적으로 서버를 비활성화시키고 다시 활성화시킬 수 있습니다. 이를 수행하려면 COM+에서 인터셉터가 일반적인 vtable 포인터처럼 작동해야 합니다.

COM 서버

COM 서버는 클라이언트 애플리케이션 또는 라이브러리에 서비스를 제공하는 애플리케이션 또는 라이브러리입니다. COM 서버는 하나 이상의 COM 객체로 구성되며 COM 객체는 속성(데이터 멤버 또는 내용) 및 메소드(또는 멤버 함수)의 집합입니다.

클라이언트는 COM 객체가 서비스를 수행하는 *방법*을 모르며 객체의 구현은 캡슐화되어 있습니다. 객체는 앞에서 설명한 대로 인터페이스를 통해 서비스를 사용할 수 있게 합니다.

또한 클라이언트는 COM 객체가 있는 *위치*를 알 필요가 없습니다. COM은 객체의 위치에 상관없이 투명한 액세스를 제공합니다.

클라이언트가 COM 객체로부터 서비스를 요청할 때 클라이언트는 클래스 식별자(CLSID)를 COM으로 전달합니다. CLSID는 단순히 COM 객체를 식별하는 GUID입니다. COM은 시스템 레지스트리에 등록된 이 CLSID를 사용하여 해당되는 서버 구현을 찾습니다. 서버를 찾으면 COM은 코드를 메모리로 가져오고 서버가 클라이언트에 대한 객체 인스턴스를 인스턴스화하도록 합니다. 이러한 과정은 요구에 따라 객체의 인스턴스를 만드는, 인터페이스 기반의 클래스 팩토리라는 특수 객체를 통해 간접적으로 처리됩니다.

COM 서버는 최소한 다음을 수행해야 합니다.

- 서버 모듈과 클래스 식별자(CLSID)를 연결하는 시스템 레지스트리에 항목 등록
- 특정 CLSID의 또 다른 객체를 만드는 클래스 팩토리 객체 구현
- COM에 클래스 팩토리 노출
- 클라이언트 서비스를 제공하지 않는 서버를 메모리에서 제거할 수 있는 언로드 메커니즘 제공

참고 C++Builder 마법사는 38-19페이지의 "마법사로 COM 객체 구현"에서 설명한 대로 COM 객체 및 서버 생성을 자동화합니다.

CoClasses 및 클래스 팩토리

COM 객체는 하나 이상의 COM 인터페이스를 구현하는 **CoClass**의 인스턴스입니다. COM 객체는 해당 인터페이스가 정의하는 대로 서비스를 제공합니다.

CoClasses는 *클래스 팩토리*라는 특수한 타입의 객체에 의해 인스턴스화됩니다. 클라이언트가 객체의 서비스를 요청할 때마다 클래스 팩토리는 해당되는 특정 클라이언트에 대한 객체 인스턴스를 만듭니다. 일반적으로 다른 클라이언트가 객체의 서비스를 요청하는 경우 클래스 팩토리는 다른 객체 인스턴스를 만들어 두 번째 클라이언트에 서비스를 제공합니다. 클라이언트는 또한 클라이언트 지원을 위해 자신을 등록하는, 실행 중인 COM 객체에 연결할 수 있습니다.

CoClass에는 반드시 클래스 팩토리와 클래스 식별자(CLSID)가 있어야만 외부적으로 즉, 다른 모듈로부터 인스턴스화될 수 있습니다. CoClasses에 대해 이러한 고유한 식별자를 사용한다는 것은 새 인터페이스가 해당 클래스에 구현될 때마다 업데이트될 수 있음을 의미합니다. 새 인터페이스는 DLL을 사용하는 경우와 달리 이전 버전에 영향을 미치지 않고 메소드를 수정하거나 추가할 수 있습니다.

C++Builder 마법사는 클래스 식별자 할당 및 클래스 팩토리 구현과 인스턴스화를 담당합니다.

in-process, out-of-process 및 원격 서버

COM을 사용하는 클라이언트는 객체가 있는 위치를 알 필요 없이 객체의 인터페이스를 호출하기만 하면 됩니다. COM이 호출을 수행하는 데 필요한 단계를 수행합니다. 이러한 단계는 객체가 클라이언트와 동일한 프로세스에 있는지, 같은 클라이언트 컴퓨터의 다른 프로세스에 있는지, 또는 네트워크상의 다른 컴퓨터에 있는지에 따라 다릅니다. 알려진 서버 유형은 다음과 같습니다.

In-process 서버

클라이언트와 *동일한 프로세스 공간*에서 실행되는 라이브러리(DLL)로, Internet Explorer 또는 Netscape에서 보는 웹 페이지에 포함된 ActiveX 컨트롤을 예로 들 수 있습니다. 여기서 ActiveX 컨트롤은 클라이언트 컴퓨터에 다운로드되며 웹 브라우저와 동일한 프로세스에서 호출됩니다.

클라이언트는 COM 인터페이스에 대한 직접 호출을 사용하여 in-process 서버와 통신합니다.

Out-of-process 서버 (또는 로컬 서버)

*프로세스 공간*은 다르지만 클라이언트와 *동일한 컴퓨터*에서 실행되는 다른 애플리케이션(EXE)입니다. 예를 들어, Word 문서에 포함된 Excel 스프레드시트는 동일한 컴퓨터에서 실행되는 두 개의 독립적인 애플리케이션입니다.

로컬 서버는 COM을 사용하여 클라이언트와 통신합니다.

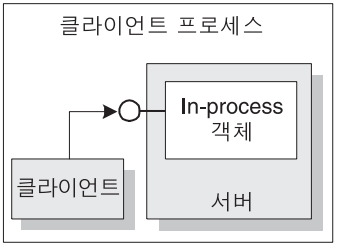
원격 서버

클라이언트와 *다른 컴퓨터*에서 실행되는 DLL 또는 다른 애플리케이션입니다. 예를 들어, C++Builder 데이터베이스 애플리케이션은 네트워크의 다른 컴퓨터에 있는 애플리케이션 서버에 연결됩니다.

원격 서버는 DCOM을 사용하여 인터페이스에 액세스하고 애플리케이션 서버와 통신합니다.

그림 38.3에서 볼 수 있듯이 in-process 서버의 경우, 객체 인터페이스에 대한 포인터가 클라이언트와 동일한 프로세스 공간에 있으므로 COM은 객체 구현을 직접 호출할 수 있습니다.

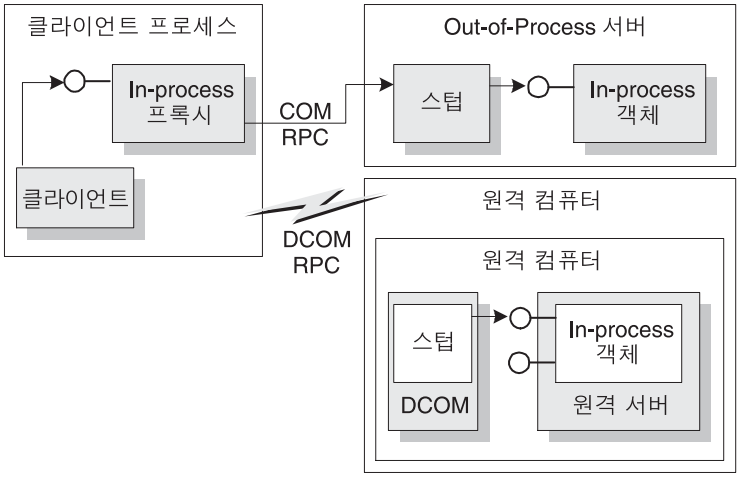
그림 38.3 In-process 서버



참고 COM+에서는 조금 다릅니다. 클라이언트가 다른 컨텍스트에 있는 객체를 호출하는 경우 COM+는 이 호출을 인터셉트하여 서버가 in-process임에도 불구하고 out-of-process 서버(아래 참조)인 것처럼 작동합니다. COM+ 작업에 대한 자세한 내용은 44장, "MTS 객체 또는 COM+ 객체 생성"을 참조하십시오.

그림 38.4에서 볼 수 있듯이 프로세스가 다른 프로세스에 있거나 다른 컴퓨터에 있는 경우, COM은 대리자를 사용하여 원격 프로시저 호출을 시작합니다. **대리자**는 클라이언트와 동일한 프로세스에 있으므로 클라이언트의 관점에서 보면 모든 인터페이스 호출이 동일하게 보입니다. 대리자는 클라이언트의 호출을 인터셉트하여 실제 객체가 실행 중인 곳으로 전달합니다. 클라이언트가 다른 프로세스 공간 또는 다른 컴퓨터에 있는 객체를 마치 자체 프로세스에 있는 것처럼 액세스할 수 있는 메커니즘을 **마샬링**이라고 합니다.

그림 38.4 Out-of-process 및 원격 서버



out-of-process 서버와 원격 서버의 차이점은 사용하는 프로세스 간 통신 유형이 다르다는 것입니다. 대리자는 COM을 사용하여 out-of-process 서버와 통신하고 DCOM을 사용하여 원격 컴퓨터와 통신합니다. DCOM은 다른 컴퓨터에서 실행 중인 원격 객체에 로컬 객체 요청을 백그라운드로 전송합니다.

참고 원격 프로시저 호출의 경우, DCOM은 Open Group의 DCE(Distributed Computing Environment)에서 제공하는 RPC 프로토콜을 사용합니다. 또한 분산 보안의 경우에는 NTLM(NT LAN Manager) 보안 프로토콜을 사용하고 디렉토리 서비스의 경우에는 DNS(Domain Name System)를 사용합니다.

마샬링 메커니즘

마샬링은 클라이언트가 다른 프로세스 또는 다른 컴퓨터에 있는 원격 객체에 대해 인터페이스 함수 호출을 수행할 수 있도록 해주는 메커니즘입니다. 마샬링은 다음과 같은 작업을 수행합니다.

- 서버 프로세스에서 인터페이스 포인터를 받아 클라이언트 프로세스의 코드에 대리자 포인터를 사용할 수 있게 합니다.
- 클라이언트로부터 전달된 인터페이스 호출의 인수를 전송하여 원격 객체의 프로세스 공간에 둡니다.

클라이언트는 모든 인터페이스 호출에서 인수를 스택에 푸시하고 인터페이스 포인터를 사용하여 함수를 호출합니다. 객체에 대한 호출이 in-process가 아닌 경우에는 호출이 대리자에게 전달됩니다. 대리자는 인수를 마샬링 패킷에 압축하고 이 구조체를 원격 객체로 전송합니다. 객체의 스텁은 패킷의 압축을 풀고 인수를 스택에 푸시한 다음 객체 구현을 호출합니다. 기본적으로 객체는 자체 주소 공간에서 클라이언트를 다시 호출합니다.

적용되는 마샬링 유형은 COM 객체가 구현하는 인터페이스에 따라 다릅니다. 객체는 *IDispatch* 인터페이스가 제공하는 표준 마샬링 메커니즘을 사용할 수 있는데, 이는 시스템 표준의 원격 프로시저 호출(RPC)을 통해 통신할 수 있도록 하는 일반적인 마샬링 메커니즘입니다. *IDispatch* 인터페이스에 대한 자세한 내용은 41-12페이지의 "Automation 인터페이스"를 참조하십시오.

객체가 *IDispatch*를 구현하지 않더라도 Automation과 호환할 수 있는 타입으로 자신을 제한하고 등록된 타입 라이브러리가 있으면 COM이 자동으로 마샬링 지원을 제공합니다.

Automation과 호환될 수 있는 타입으로 자신을 제한하지 않거나 타입 라이브러리를 등록하지 않는 애플리케이션은 반드시 자체 마샬링을 제공해야 합니다. 마샬링은 *IMarshal* 인터페이스 구현을 통해서 제공되거나 각각 생성된 대리자/스텁 DLL을 사용하여 제공됩니다. C++Builder는 대리자/스텁 DLL의 자동 생성을 지원하지 않습니다.

집합체(aggregation)

때때로 서버 객체는 다른 COM 객체를 사용하여 일부 기능을 수행하기도 합니다. 예를 들어, 재고 관리 객체는 각각의 송장 객체를 사용하여 고객 송장을 처리할 수 있습니다. 그러나 재고 관리 객체가 클라이언트에 송장 인터페이스를 제시하고자 하는 경우에는 문제가 있습니다. 재고 인터페이스를 가진 클라이언트는 *QueryInterface*를 호출하여 송장 인터페이스를 가져올 수 있지만 송장 객체가 만들어질 때는 재고 관리 객체에 대해 모르므로 *QueryInterface* 호출에 대한 응답으로 재고 인터페이스를 반환할 수 없습니다. 따라서 송장 인터페이스를 가진 클라이언트는 재고 인터페이스로 돌아갈 수 없습니다.

이러한 문제를 피하기 위해 일부 COM 객체는 **집합체(aggregation)**를 지원합니다. 즉, 재고 관리 객체가 송장 객체의 인스턴스를 만드는 경우, 송장 객체의 인스턴스에 자체 *IUnknown* 인터페이스의 복사본을 전달합니다. 그러면 송장 객체는 이 *IUnknown* 인터페이스를 사용하여 자신이 지원하지 않는 재고 인터페이스 등의 인터페이스를 요청하는 모든 *QueryInterface* 호출을 처리할 수 있습니다. 이런 경우, 두 객체를 합쳐 집합체라고 합니다. 송장 객체는 집합체의 내부 객체 또는 포함된 객체라 하고 재고 객체는 외부 객체라 합니다.

참고 집합체의 외부 객체 역할을 하기 위해 COM 객체는 Windows API인 *CoCreateInstance* 또는 *CoCreateInstanceEx*를 사용하여 *IUnknown* 포인터를 내부 객체가 *QueryInterface* 호출에 사용할 수 있는 매개변수로 전달하는 내부 객체를 만들어야 합니다. *TComInterface* 객체를 인스턴스화하여 *CreateInstance* 메소드를 사용하는 것도 같은 목적입니다.

집합체의 내부 객체 역할을 하는 객체는 *IUnknown* 인터페이스의 두 가지 구현을 제공합니다. 하나는 처리할 수 없는 *QueryInterface* 호출을 외부 객체가 *IUnknown*을 제어하는 시점까지 연기하는 것이고, 다른 하나는 처리할 수 없는 *QueryInterface* 호출을 받을 때 오류를 반환하는 것입니다. C++Builder 마법사는 내부 객체로서의 역할에 대해 이러한 지원을 포함하는 객체를 자동으로 만듭니다.

COM 클라이언트

클라이언트는 항상 COM 객체의 인터페이스를 통해 해당 COM 객체가 제공하는 기능을 확인할 수 있습니다. 모든 COM 객체는 클라이언트가 알려진 인터페이스를 요청할 수 있게 합니다. 또한 서버가 *IDispatch* 인터페이스를 지원하는 경우 클라이언트는 서버로부터 인터페이스가 지원하는 메소드에 대한 정보를 얻을 수 있습니다. 서버 객체는 클라이언트의 객체 사용에 대해 어떤 요구도 하지 않습니다. 마찬가지로, 클라이언트는 객체가 서비스를 제공하는 방법이나 위치를 알 필요가 없으며, 인터페이스를 통해 서비스를 제공하는 서버 객체에 의존하기만 하면 됩니다.

COM 클라이언트에는 컨트롤러와 컨테이너라는 두 가지 유형이 있습니다. 컨트롤러는 서버를 시작하고 인터페이스를 사용하여 서버와 상호 작용합니다. 컨트롤러는 COM 객체의 서비스를 요청하거나 COM 객체를 독립적인 프로세스로 실행합니다. 컨테이너는 컨테이너의 사용자 인터페이스에 표시되는 보이는 컨트롤 또는 객체를 호스트합니다. 또한 미리 정의된 인터페이스를 사용하여 표시 문제를 서버 객체와 조정합니다. 컨테이너 관계는 DCOM을 통해 표시할 수 없습니다. 예를 들어, 컨트롤이 자신을 그려야 할 때는 로컬 GDI 리소스에 액세스해야 하기 때문에 컨테이너의 사용자 인터페이스에 표시되는 컨트롤은 반드시 로컬에 있어야 합니다.

C++Builder를 사용하면 서버 객체가 다른 VCL 컴포넌트처럼 보이도록 타입 라이브러리나 ActiveX 컨트롤을 컴포넌트 랩퍼로 임포트할 수 있으므로 보다 쉽게 COM 클라이언트를 개발할 수 있습니다. 이 프로세스에 대한 자세한 내용은 40장, "COM 클라이언트 생성"을 참조하십시오.

COM 확장

COM은 원래 핵심적인 통신 기능을 제공하고 확장을 통해 이러한 기능을 더욱 광범위하게 사용할 수 있도록 설계되었습니다. COM 자체는 특정 목적에 대해 특화된 인터페이스 집합을 정의함으로써 그 핵심 기능을 확장해 왔습니다.

다음은 현재 COM 확장이 제공하는 서비스의 일부를 나열한 것입니다. 이후 단원에서는 이러한 서비스에 대해 상세하게 설명합니다.

Automation 서버 Automation은 애플리케이션이 다른 애플리케이션의 객체를 프로그래밍 방식으로 제어할 수 있는 기능을 뜻하고, Automation 서버는 런타임 시 다른 실행 파일에 의해 제어될 수 있는 객체를 뜻합니다.

ActiveX 컨트롤 ActiveX 컨트롤은 일반적으로 클라이언트 애플리케이션에 포함하기 위해 만들어진 특화된 in-process 서버입니다. 이 컨트롤은 이벤트뿐만 아니라 디자인과 런타임 동작까지 제공합니다.

Active Server Pages Active Server Pages는 HTML 페이지를 만드는 스크립트입니다. 스크립트 언어에는 Automation 객체를 만들고 실행하는 데 필요한 구조체가 포함됩니다. 즉, Active Server Page가 Automation 컨트롤러 역할을 합니다.

Active Document 연결 및 포함, 드래그 앤 드롭, 시각적 편집, 현재 위치에서 활성화를 지원하는 객체입니다. Active Document의 예로 Word 문서와 Excel 스프레드시트를 들 수 있습니다.

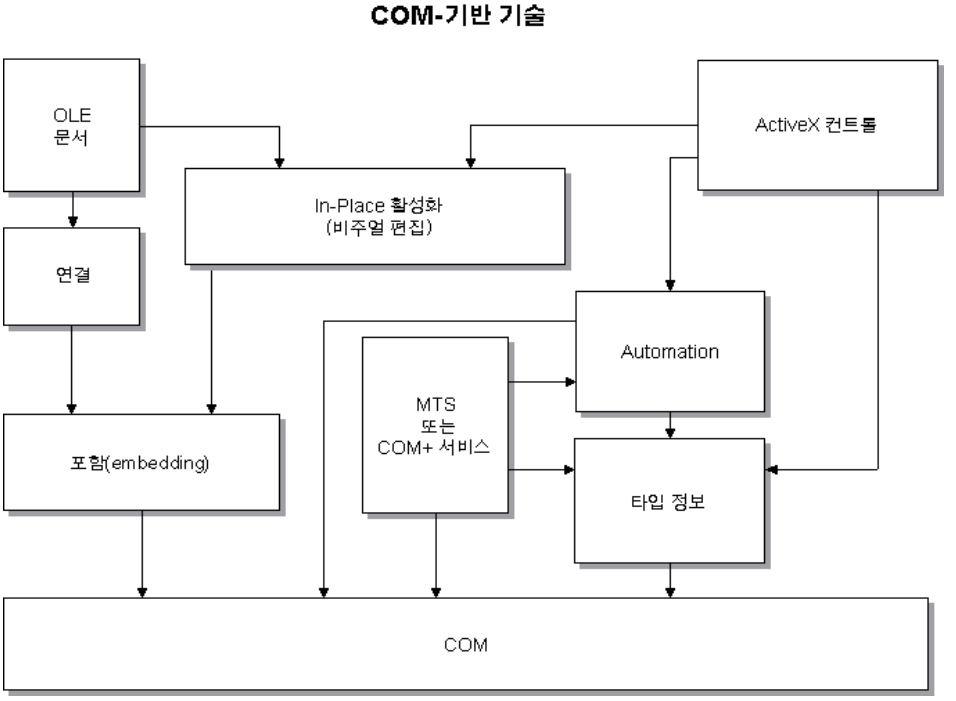
트랜잭션 객체 많은 수의 클라이언트에 응답하기 위한 추가 지원을 포함하는 객체입니다. 여기에는 just-in-time 활성화, 트랜잭션, 리소스 풀링 및 보안 서비스가 포함됩니다. 이러한 기능은 원래 MTS에 의해 처리되었으나 COM+ 출시와 함께 COM에 내장되었습니다.

COM+ 이벤트 및 이벤트 구독 객체 느슨하게 연결된 COM+ Event 모델을 지원하는 객체입니다. ActiveX 컨트롤에서 사용하는, 밀접하게 연결된 모델과는 달리 COM+ Event 모델에서는 이벤트 구독자와 관계 없이 이벤트 퍼블리셔를 개발할 수 있습니다.

타입 라이브러리 종종 리소스로 저장되는 정적 데이터 구조의 컬렉션이며 객체 및 그 인터페이스에 대한 자세한 타입 정보를 제공합니다. Automation 서버, ActiveX 컨트롤 및 트랜잭션 객체의 클라이언트는 타입 정보를 사용할 수 있을 것으로 예상합니다.

다음 다이어그램은 COM 확장 간의 관계 및 COM 확장이 COM을 기반으로 구축되는 방법을 보여 줍니다.

그림 38.5 COM 기반 기술



COM 객체는 보일 수도 있고 보이지 않을 수도 있습니다. 일부는 클라이언트와 동일한 프로세스 공간에서 실행되어야 하고 일부는 객체가 마샬링 지원을 제공하는 한 다른 프로세스 또는 원격 컴퓨터에서 실행될 수 있습니다. 표 38.1에는 생성 가능한 COM 객체 타입, 보이는지 여부, 실행 가능한 프로세스 공간, 마샬링 제공 방법 및 타입 라이브러리가 필요한지 여부가 요약되어 있습니다.

표 38.1 COM 객체 요구 사항

객체	객체가 보이는지 여부	프로세스 공간	통신	타입 라이브러리
Active Document	대부분	in-process 또는 out-of-process	OLE Verb	필요 없음
Automation 서버	종종	in-process, out-of-process 또는 원격	*IDispatch* 인터페이스를 사용하여 자동으로 마샬링됨 (out-of process 및 원격 서버의 경우)	자동 마샬링에 필요함
ActiveX 컨트롤	대부분	in-process	*IDispatch* 인터페이스를 사용하여 자동으로 마샬링됨	필요함
MTS 또는 COM+	종종	MTS의 경우 in-process, COM+의 경우 모든 프로세스 공간	타입 라이브러리를 통해 자동으로 마샬링됨	필요함
In-process 사용자 정의 인터페이스 객체	선택적	in-process	in-process 서버에는 마샬링이 필요하지 않음	권장됨
기타 사용자 정의 인터페이스 객체	선택적	in-process, out-of-process 또는 원격	타입 라이브러리를 통해 자동으로 마샬링되거나 사용자 정의 인터페이스를 사용하여 수동으로 마샬링됨	권장됨

Automation 서버

Automation은 동시에 하나 이상의 애플리케이션을 처리하는 매크로처럼 애플리케이션이 다른 애플리케이션의 객체를 프로그래밍 방식으로 제어할 수 있는 기능을 말합니다. 처리되는 서버 객체를 Automation 객체라고 하고 Automation 객체의 클라이언트를 Automation 컨트롤러라고 합니다.

Automation은 in-process, 로컬 및 원격 서버에서 사용할 수 있습니다.

Automation에는 두 가지 핵심적인 특징이 있습니다.

- Automation 객체는 일련의 속성과 명령을 정의하고 타입 설명을 통해 기능을 설명합니다. 이러한 작업을 하려면 인터페이스, 인터페이스 메소드 및 해당 메소드의 인수에 대한 정보를 제공할 수 있는 방법이 있어야 합니다. 일반적으로 이러한 정보는 타입 라이브러리에 있습니다. Automation 서버는 *IDispatch* 인터페이스(다음 참조)를 통해 쿼리될 때 동적으로 타입 정보를 만들 수도 있습니다.
- Automation 객체는 다른 애플리케이션에서 사용할 수 있도록 메소드를 액세스 가능하게 만들며, 이를 위해 *IDispatch* 인터페이스를 구현합니다. 이 인터페이스를 통해 객체는 각각의 메소드와 속성을 모두 노출할 수 있습니다. 타입 정보를 통해 확인되었을 경우 객체의 메소드는 이 인터페이스의 기본 메소드를 통해 호출될 수 있습니다.

Automation *IDispatch* 인터페이스가 마샬링 프로세스를 자동화하므로 개발자들은 Automation을 사용하여 임의의 프로세스 공간에서 실행되는 보이지 않는 OLE 객체를 만들고 사용하는 경우가 많습니다. 그러나 Automation에 사용할 수 있는 타입은 제한되어 있습니다.

일반적으로 타입 라이브러리에 사용할 수 있는 타입과 특히 Automation 인터페이스에 사용할 수 있는 타입 리스트를 보려면 39-11페이지의 "유효한 타입"을 참조하십시오.

Automation 서버 작성에 대한 자세한 내용은 41장, "간단한 COM 서버 생성"을 참조하십시오.

Active Server Pages

ASP(Active Server Page) 기술을 사용하면 웹 서버를 통해 클라이언트에서 실행할 수 있는 Active Server Pages라는 간단한 스크립트를 작성할 수 있습니다. 클라이언트에서 실행되는 ActiveX 컨트롤과 달리 Active Server Pages는 서버에서 실행되며 결과 HTML 페이지를 클라이언트에 반환합니다.

Active Server Pages는 Jscript 또는 VB 스크립트로 작성되며 서버가 웹 페이지를 로드할 때마다 실행됩니다. 그런 다음 스크립트는 포함된 Automation 서버 또는 Enterprise Java Bean을 시작할 수 있습니다. 예를 들어, 비트맵을 만들거나 데이터베이스에 연결하기 위해 Automation 서버를 작성할 수 있으며 이 서버는 클라이언트가 웹 페이지를 로드할 때마다 업데이트된 데이터에 액세스할 수 있습니다.

Active Server Pages는 Microsoft IIS(Internet Information Server) 환경에 의존하여 웹 페이지에 서비스를 제공합니다.

C++Builder 마법사를 사용하면 Active Server Page와 함께 사용하기 위해 특별히 개발된 Automation 객체인 Active Server 객체를 만들 수 있습니다. 이러한 타입의 객체를 만들고 사용하는 방법에 대한 자세한 내용은 42장, "Active Server Page 생성"을 참조하십시오.

ActiveX 컨트롤

ActiveX는 COM 컴포넌트, 특히 컨트롤을 보다 작고 효율적으로 만들 수 있는 기술입니다. 특히 클라이언트가 사용하기 전에 다운로드해야 하는 인트라넷 애플리케이션용으로 개발된 컨트롤에는 필수적입니다.

ActiveX 컨트롤은 in-process 서버로만 실행되는 비주얼(visual) 컨트롤이며 ActiveX 컨테이너 애플리케이션에 연결될 수 있습니다. 그 자체로 완벽한 애플리케이션은 아니지만 다양한 애플리케이션에서 다시 사용할 수 있는 조립식 OLE 컨트롤이라고 생각할 수 있습니다. ActiveX 컨트롤에는 비주얼 사용자 인터페이스가 있으며 미리 정의된 인터페이스에 기초하여 I/O 및 표시 문제를 호스트 컨테이너와 조정합니다.

ActiveX 컨트롤은 Automation을 사용하여 각각의 속성, 메소드 및 이벤트를 노출하며, 이벤트 발생, 데이터 소스에 연결, 라이센스 지원 등의 기능을 수행합니다.

웹 사이트에서 웹 페이지의 대화형 객체로 사용하는 것도 ActiveX 컨트롤을 사용하는 방법 중 하나입니다. ActiveX는 웹 브라우저를 통해 HTML이 아닌 문서를 보기 위한 ActiveX Document의 사용을 비롯하여 World Wide Web에 대한 대화형 컨텐트를 대상으로 하는 표준입니다. ActiveX 기술에 대한 자세한 내용은 Microsoft ActiveX 웹 사이트를 참조하십시오.

C++Builder 마법사를 사용하면 보다 쉽게 ActiveX 컨트롤을 만들 수 있습니다. 이러한 유형의 객체를 만들고 사용하는 방법에 대한 자세한 내용은 43장, "ActiveX 컨트롤 생성"을 참조하십시오.

Active Document

Active Document(이전에는 OLE 문서로 지칭)는 연결 및 포함, 드래그 앤 드롭, 시각적 편집을 지원하는 일련의 COM 서비스입니다. Active Document는 사운드 클립, 스프레드시트, 텍스트, 비트맵 등 형식이 다른 데이터나 객체를 완벽하게 통합할 수 있습니다.

Active Document는 ActiveX 컨트롤과 달리 in-process 서버에 제한되지 않으며 cross-process 애플리케이션에서 사용할 수 있습니다.

거의 보이는 경우가 없는 Automation 객체와 달리, Active Document 객체는 다른 애플리케이션에서 시각적으로 활성화될 수 있습니다. 따라서 Active Document 객체는 디스플레이 또는 출력 장치에서 객체를 시각적으로 표시하는 데 사용되는 표시 데이터와 객체를 편집하는 데 사용되는 원시 데이터 등 두 가지 유형의 데이터와 연관됩니다.

Active Document 객체는 문서 컨테이너 또는 문서 서버일 수 있습니다. C++Builder는 Active Document를 만드는 데 필요한 자동 마법사를 제공하지 않지만 개발자가 VCL 클래스인 *TOleContainer*를 사용하여 기존 Active Document의 연결 및 포함을 지원할 수 있습니다.

또한 *TOleContainer*를 Active Document 컨테이너에 대한 기초로 사용할 수 있습니다. Active Document 서버에 대한 객체를 만들려면 객체가 지원해야 하는 서비스에 따라 COM Object 마법사를 사용하여 적절한 인터페이스를 추가합니다. Active Document 서버 생성과 사용에 대한 자세한 내용은 Microsoft ActiveX 웹 사이트를 참조하십시오.

참고 Active Document 스펙에서는 cross-process 애플리케이션에서의 마샬링 기능을 기본으로 지원하지만, Active Document는 윈도우 핸들, 메뉴 핸들 등 해당 컴퓨터의 특정 시스템에 한정되는 타입을 사용하므로 원격 서버에서 실행되지 않습니다.

트랜잭션 객체

C++Builder는 Windows 2000 이전의 Windows 버전에서는 MTS(Microsoft Transaction Server), Windows 2000 이상 버전에서는 COM+에 의해 제공되는 트랜잭션 서비스, 보안 및 리소스 관리를 사용하는 객체를 지칭하기 위해 "트랜잭션 객체"라는 용어를 사용합니다. 이러한 객체는 대규모의 분산 환경에서 사용하기 위해 개발되었습니다.

트랜잭션 서비스는 작업이 항상 완료되거나, 완료하지 못한 경우에는 롤백되도록 하는 견고성을 제공합니다. 서버에서 작업을 부분적으로 완료하는 경우는 없습니다. 보안 서비스를 사용하면 다양한 클래스의 클라이언트에 다양한 레벨의 지원을 노출할 수 있습니다. 리소스 관리는 리소스를 풀링하거나 객체가 사용 중인 경우에만 활성 상태를 유지하도록 함으로써 객체가 보다 많은 클라이언트를 처리할 수 있게 해줍니다. 시스템이 이러한 서비스를 제공하려면 객체가 *IObjectControl* 인터페이스를 구현해야 합니다. 트랜잭션 객체는 MTS 또는 COM+에서 만든 *IObjectContext* 인터페이스를 사용하여 서비스에 액세스합니다.

MTS에서 서버 객체는 라이브러리(DLL)에 내장되어야 하며 그런 다음 MTS 런타임 환경에 설치되어야 합니다. 즉, 서버 객체는 MTS 런타임 프로세스 공간에서 실행되는 in-process 서버입니다. COM+에서는 모든 COM 호출이 인터셉터를 통해 라우트되므로 이러한 제한이 적용되지 않습니다. 클라이언트에서 MTS와 COM+는 전혀 차이가 없습니다.

MTS 또는 COM+ 서버는 동일한 프로세스 공간에서 실행되는 트랜잭션 객체를 그룹화합니다. MTS에서는 이 그룹을 MTS 패키지라고 하며 COM+에서는 COM+ 애플리케이션이라고 합니다. 한 대의 컴퓨터에서 여러 개의 MTS 패키지 또는 COM+ 애플리케이션을 실행할 수 있으며 이들은 각각의 프로세스 공간에서 실행됩니다.

클라이언트에게 트랜잭션 객체는 다른 COM 서버 객체와 동일하게 보입니다. 클라이언트는 트랜잭션 자체를 시작하지 않는 한 트랜잭션, 보안 또는 just-in-time 활성화에 대해 알 필요가 없습니다.

MTS와 COM+ 모두 트랜잭션 객체를 관리하기 위한 각각의 도구를 제공합니다. 이 도구를 사용하면 객체를 패키지나 COM+ 애플리케이션으로 구성하고, 컴퓨터에 설치된 패키지나 COM+ 애플리케이션을 보고, 포함된 객체의 어트리뷰트(attribute)를 보거나 변경하고, 트랜잭션을 모니터하거나 관리하고, 클라이언트가 객체를 사용할 수 있게 하는 등의 작업을 할 수 있습니다. MTS에서는 이 도구가 MTS Explorer이고 COM+에서는 COM+ Component Manager입니다.

COM+ 이벤트 및 event subscriber object

COM+ Event 시스템에서는 이벤트를 만드는 애플리케이션(퍼블리셔)을 이벤트에 응답하는 애플리케이션(구독자)으로부터 분리하는 중간 레이어의 소프트웨어를 도입하였습니다. 퍼블리셔와 구독자를 엄격하게 연결하지 않고 서로 독립적으로 개발하고 배포하고 활성화할 수 있습니다.

COM+ Event 모델에서는 이벤트 인터페이스가 C++Builder의 COM+ Event Object 마법사를 사용하여 먼저 만들어집니다. 이벤트 인터페이스에는 구현이 없으며 단순히 퍼블리셔가 만들고 구독자가 응답하는 이벤트 메소드를 정의하기만 합니다. 그러면 COM+ 이벤트 객체가 COM+ Catalog 내의 COM+ 애플리케이션으로 설치됩니다. 이러한 작업은 프로그래밍 방식으로 *TComAdminCatalog* 객체를 사용하는 C++Builder IDE를 사용하거나, 시스템 관리자가 Component Services 도구를 사용하여 수행할 수 있습니다.

이벤트 구독자는 이벤트 인터페이스에 대한 구현을 제공합니다. C++Builder의 COM+ Event Subscription 마법사를 사용하여 이벤트 구독자 컴포넌트를 만들 수 있습니다. 마법사를 사용하면 구현하고자 하는 이벤트 객체를 선택할 수 있으며 C++Builder가 인터페이스의 각 메소드를 스텁으로 전달합니다. 또한 이벤트 객체가 COM+ Catalog에 아직 설치되지 않은 경우에는 타입 라이브러리를 선택할 수도 있습니다.

결론적으로 구독자 컴포넌트가 구현되면 COM+ Catalog에도 설치되어야 합니다. 다시 말해 이 작업은 C++Builder IDE, *TComAdminCatalog* 객체 또는 Component Services 관리 도구를 사용하여 수행될 수 있습니다.

퍼블리셔가 이벤트를 만들고자 하는 경우에는 구독자 컴포넌트가 아니라 이벤트 객체의 인스턴스를 만들고 이벤트 인터페이스의 적절한 메소드를 호출하기만 하면 됩니다. 그러면 COM+가 시작되어 해당 이벤트 객체를 구독하는 모든 애플리케이션을 한 번에 하나씩 동기적으로 호출하여 이를 알립니다. 이런 식으로 퍼블리셔는 이벤트를 구독하는 애플리케이션에 대해 알 필요가 없습니다. 구독자 또한 구독하고자 하는 퍼블리셔를 선택하기 위해 이벤트 인터페이스의 구현 이상의 것을 알 필요가 없습니다. COM+가 나머지 부분을 모두 처리합니다.

COM+ Event 시스템에 대한 자세한 내용은 44-21페이지의 "COM+에서 이벤트 생성"을 참조하십시오.

타입 라이브러리

타입 라이브러리는 객체의 인터페이스에서 확인할 수 있는 것보다 객체에 관해 더 많은 타입 정보를 얻을 수 있는 방법을 제공합니다. 타입 라이브러리에 포함된 타입 정보는 CLSID로 지정된 어떤 객체에 어떤 인터페이스가 존재하는지, 각 인터페이스에는 어떤 멤버 함수가 있는지, 이러한 함수에는 어떤 인수가 필요한지 등 객체와 그 인터페이스에 대해 필요한 정보를 제공합니다.

실행 중인 객체 인스턴스를 쿼리하거나 타입 라이브러리를 로드하여 읽는 방법으로 타입 정보를 얻을 수 있습니다. 이렇게 얻은 정보를 사용하여 원하는 객체를 사용하는 클라이언트를 구현할 수 있으며 특히 어떤 멤버 함수가 필요한지, 이러한 멤버 함수에 무엇을 전달할지 알 수 있습니다.

Automation 서버의 클라이언트, ActiveX 컨트롤 및 트랜잭션 객체는 타입 정보를 사용해야 합니다. 모든 C++Builder 마법사는 자동으로 타입 라이브러리를 만듭니다. 39장, "타입 라이브러리 사용"의 설명대로 Type Library Editor를 사용하여 이러한 타입 정보를 보거나 편집할 수 있습니다.

이 절에서는 타입 라이브러리에 포함된 내용, 생성 방법, 사용되는 시기 및 액세스 방법에 대해 설명합니다. 여러 랭귀지 간에 인터페이스를 공유하고자 하는 개발자들을 위해 이 단원의 끝부분에서는 타입 라이브러리 도구 사용에 대한 제안 사항을 제공합니다.

타입 라이브러리의 내용

타입 라이브러리에는 *타입 정보*가 포함되어 있으며 이 정보는 어떤 COM 객체에 어떤 인터페이스가 있는지를 비롯하여 인터페이스 메소드에 대한 인수의 타입과 수를 표시합니다. 이러한 설명에는 CoClasses(CLSID) 및 인터페이스(IID)에 대한 고유한 식별자가 포함되어 있으므로 Automation 인터페이스 메소드 및 속성에 대한 dispatch 식별자(dispID)처럼 제대로 액세스할 수 있습니다.

또한 타입 라이브러리에는 다음과 같은 정보가 포함될 수 있습니다.

- 사용자 정의 인터페이스와 연결된 사용자 정의 타입 정보에 대한 설명
- Automation 또는 ActiveX 서버에서 엑스포트되지만 인터페이스 메소드가 아닌 루틴
- 열거, 레코드(구조), 합집합, 알리아스 및 모듈 데이터 타입에 관한 정보
- 다른 타입 라이브러리에서의 타입 설명에 대한 참조

타입 라이브러리 생성

일반적인 개발 도구를 사용하면 IDL(Interface Definition Language) 또는 ODL(Object Description Language)에서 스크립트를 작성한 다음 컴파일러를 통해 이러한 스크립트를 실행하는 방법으로 타입 라이브러리를 만들 수 있습니다. 그러나 C++Builder는 New Items 다이얼로그 박스의 ActiveX 또는 Multitier 페이지에 있는 마법사 중 하나를 사용하여 COM 객체(ActiveX 컨트롤, Automation 객체, 원격 데이터 모듈 등)를 만들 때 자동으로 타입 라이브러리를 만듭니다. 또한 메인 메뉴에서 File|New|Other를 선택하고 ActiveX 탭을 선택한 다음 Type Library를 선택하여 타입 라이브러리를 만들 수도 있습니다.

C++Builder의 Type Library Editor를 사용하면 타입 라이브러리를 볼 수 있습니다. 또한 Type Library Editor 를 사용하면 쉽게 타입 라이브러리를 편집할 수 있으며 타입 라이브러리가 저장될 때 C++Builder가 자동으로 해당되는 .tlb 파일(바이너리 타입 라이브러리 파일)을 업데이트합니다. 마법사에 의해 만들어진 인터페이스 및 CoClasses가 변경되면 Type Library Editor 또한 개발자의 구현 파일을 업데이트합니다. Type Library Editor를 사용하여 인터페이스 및 CoClasses를 작성하는 방법에 대한 자세한 내용은 39장, "타입 라이브러리 사용"을 참조하십시오.

타입 라이브러리 사용 시기

외부 사용자에게 노출되는 각 객체 집합에 대해 타입 라이브러리를 만들어야 합니다. 예를 들어, 다음과 같습니다.

- ActiveX 컨트롤에는 타입 라이브러리가 필요하며 ActiveX 컨트롤을 포함하는 DLL에 리소스로 포함되어야 합니다.
- 사용자 정의 인터페이스의 vtable 연결을 지원하는 노출된 객체는 vtable 참조가 컴파일 시에 연결되므로 타입 라이브러리에서 설명되어야 합니다. 클라이언트는 타입 라이브러리에서 인터페이스에 대한 정보를 임포트하여 컴파일하는 데 사용합니다. vtable 및 컴파일 시 연결에 대한 자세한 내용은 41-12페이지의 "Automation 인터페이스"를 참조하십시오.
- Automation 서버를 구현하는 애플리케이션은 클라이언트가 초기 연결할 수 있도록 타입 라이브러리를 제공해야 합니다.
- *IProvideClassInfo* 인터페이스를 지원하는 클래스로부터 인스턴스화된 객체에는 타입 라이브러리가 있어야 합니다.
- 타입 라이브러리가 반드시 필요하지는 않으나 OLE 드래그 앤 드롭과 함께 사용되는 객체를 식별하는 데 유용합니다.

타입 라이브러리 액세스

바이너리 타입 라이브러리는 일반적으로 리소스 파일(.res) 또는 확장자가 .tlb인 독립형 파일의 일부입니다. 타입 라이브러리가 리소스 파일에 포함되어 있는 경우에는 서버(.dll, .ocx 또는 .exe)에 연결될 수 있습니다.

타입 라이브러리가 만들어지면 객체 브라우저, 컴파일러 및 유사한 도구의 특수한 타입 인터페이스를 통해 타입 라이브러리에 액세스할 수 있습니다.

인터페이스	설명
ITypeLib	타입 라이브러리 설명에 액세스하는 데 필요한 메소드를 제공합니다.
ITypeLib2	설명서 문자열, 사용자 정의 데이터 및 타입 라이브러리 통계에 대한 지원을 포함시키기 위해 *ITypeLib*를 늘립니다.
ITypeInfo	타입 라이브러리에 포함된 개별적인 객체에 대한 설명을 제공합니다. 예를 들어, 브라우저는 이 인터페이스를 사용하여 타입 라이브러리로부터 객체에 대한 정보를 추출합니다.
ITypeInfo2	사용자 정의 데이터 요소를 액세스하는 메소드를 포함하여, 추가적인 타입 라이브러리 정보에 액세스하기 위해 *ITypeInfo*를 늘립니다.
ITypeComp	인터페이스에 연결할 때 컴파일러가 필요한 정보에 신속하게 액세스할 수 있는 방법을 제공합니다.

C++Builder에서는 Project|Import Type Library를 선택하여 다른 애플리케이션에서 타입 라이브러리를 임포트하여 사용할 수 있습니다.

타입 라이브러리 사용의 이점

애플리케이션에 타입 라이브러리가 필요 없더라도 타입 라이브러리를 사용하면 다음과 같은 이점이 있음을 고려해 볼 수 있습니다.

- Variants를 통해 호출하는 것과 반대로 Automation과 초기 연결할 수 있으며 vtables 또는 이중 인터페이스를 지원하지 않는 컨트롤러가 컴파일 시 dispID를 인코드하여 런타임 성능을 향상시킬 수 있습니다.
- 타입 브라우저는 라이브러리를 검색하므로 클라이언트가 객체의 특성을 확인할 수 있습니다.
- *RegisterTypeLib* 함수를 사용하여 노출된 객체를 등록 데이터베이스에 등록할 수 있습니다.
- *UnRegisterTypeLib* 함수를 사용하여 시스템 레지스트리에서 애플리케이션의 타입 라이브러리를 완전히 제거할 수 있습니다.
- Automation이 타입 라이브러리의 정보를 사용하여 다른 프로세스의 객체에 전달되는 매개변수를 패키지화하므로 로컬 서버 액세스가 향상됩니다.

타입 라이브러리 도구 사용

다음은 라이브러리와 함께 사용하는 도구를 나열한 것입니다.

- 기존의 타입 라이브러리를 가져와서 C++Builder 인터페이스 파일(_TLB.cpp 및 _TLB.h 파일)을 만드는 TLIBIMP(Type Library Import) 도구가 Type Library Editor에 통합되었습니다. TLIBIMP는 Type Library Editor에서 사용할 수 없는 구성 옵션을 추가로 제공합니다.
- TRegSvr은 C++Builder에서 제공하는 도구로, 서버와 타입 라이브러리의 등록 및 등록 취소에 사용됩니다. TRegSvr에 대한 소스는 Examples 디렉토리에 예제로 제공됩니다.
- MIDL(Microsoft IDL) 컴파일러는 IDL 스크립트를 컴파일하여 타입 라이브러리를 만듭니다. MIDL에는 MS Win32 SDK에 있는 헤더 파일 작성에 필요한 선택 스위치가 있습니다.
- RegSvr32.exe는 서버와 타입 라이브러리의 등록 및 등록 취소에 사용되는 표준 Windows 유틸리티입니다.
- OLEView는 타입 라이브러리 브라우저 도구이며 Microsoft 웹 사이트에서 찾을 수 있습니다.

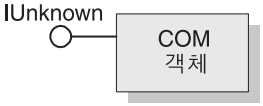
마법사로 COM 객체 구현

C++Builder는 여러 가지 세부 사항을 처리하는 마법사를 제공함으로써 보다 쉽게 COM 서버를 작성할 수 있도록 해줍니다. C++Builder는 다음을 만들기 위한 각각의 마법사를 제공합니다.

- 간단한 COM 객체
- Automation 객체
- Active Server 객체(Active Server Page 포함용)
- ActiveX 컨트롤
- ActiveX 폼
- 트랜잭션 객체
- COM+ 이벤트 객체
- COM+ 구독 객체
- 속성 페이지
- 타입 라이브러리
- ActiveX 라이브러리

마법사는 각 타입의 COM 객체 생성과 관련된 많은 작업을 처리하며 각 타입의 객체에 필요한 필수 COM 인터페이스를 제공합니다. 그림 38.6에서 볼 수 있듯이 마법사는 간단한 COM 객체를 사용하여 객체에 인터페이스 포인터를 제공하는 필수 COM 인터페이스인 *IUnknown* 을 구현합니다.

그림 38.6 간단한 COM 객체 인터페이스



또한 개발자가 *IDispatch* 자손을 지원하는 객체를 만들 것을 지정하면 COM Object 마법사는 *IDispatch*에 대한 구현을 제공합니다.

그림 38.7에서 볼 수 있듯이 Automation 및 Active Server 객체의 경우, 마법사는 자동 마샬링을 제공하는 *IUnknown* 및 *IDispatch*를 구현합니다.

그림 38.7 Automation 객체 인터페이스

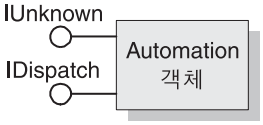


그림 38.8에서 볼 수 있듯이 ActiveX 컨트롤 객체 및 ActiveX 폼의 경우, 마법사는 *IUnknown*, *IDispatch*, *IOleObject*, *IOleControl* 등을 비롯하여 필수적인 모든 ActiveX 컨트롤 인터페이스를 구현합니다.

그림 38.8 ActiveX 객체 인터페이스

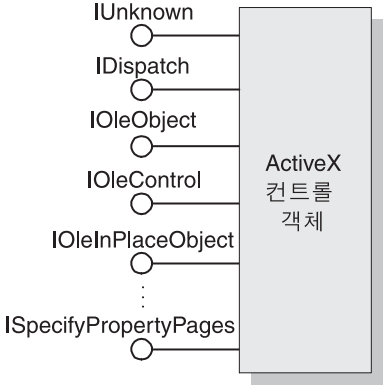


표 38.2는 다양한 마법사 및 해당 마법사가 구현하는 인터페이스를 보여 줍니다.

표 38.2 COM, Automation 및 ActiveX 객체 구현을 위한 C++Builder 마법사

마법사	구현되는 인터페이스	마법사가 수행하는 작업
COM Server	IUnknown (개발자가 *IDispatch*의 자손인 디폴트 인터페이스를 선택한 경우, *IDispatch* 포함)	서버 등록, 클래스 등록, 서버 로드 및 언로드, 객체 인스턴스화를 처리하는 루틴을 엑스포트합니다. 서버상에서 구현되는 객체에 대한 클래스 팩토리를 만들고 관리합니다. 선택된 스레딩 모델을 지정하는 객체에 대한 레지스트리 엔트리를 제공합니다. 필요할 경우 이벤트 생성에 필요한 서버사이드 지원을 제공합니다. 선택된 인터페이스를 구현하는 메소드를 선언하고 구현의 틀을 제공하여 개발자가 완료할 수 있도록 합니다. 타입 라이브러리를 제공합니다. 타입 라이브러리에 등록된 임의의 인터페이스를 선택하여 구현할 수 있도록 합니다. 이 작업을 하려면 타입 라이브러리를 사용해야 합니다.
Automation Server	*IUnknown*, *IDispatch*	위에서 설명한 COM 서버 마법사의 작업을 수행합니다. 개발자가 지정하는 이중 또는 dispatch 인터페이스를 구현합니다.
Active Server Object	*IUnknown*, *IDispatch*, *(IASPObject)*	위에서 설명한 Automation Object 마법사의 작업을 수행하고 웹 브라우저에 로드할 수 있는 ASP 페이지를 선택적으로 작성합니다. 이 마법사를 사용하면 Type Library Editor 가 표시되므로 필요에 따라 객체의 속성과 메소드를 수정할 수 있습니다. ASP 내장 함수를 속성으로 처리하여 ASP 애플리케이션 및 이를 실행한 HTTP 메시지에 대한 정보를 쉽게 얻을 수 있게 해줍니다.
ActiveX Control	*IUnknown*, *IDispatch*, *IPersistStreamInit*, *IOleInPlaceActiveObject*, *IPersistStorage*, *IViewObject*, *IOleObject*, *IViewObject2*, *IOleControl*, *IPerPropertyBrowsing*, *IOleInPlaceObject*, *ISpecifyPropertyPages*	위에서 설명한 Automation Server 마법사의 작업을 수행합니다. ActiveX 컨트롤의 기초가 되며 모든 ActiveX 인터페이스를 구현하는 VCL 컨트롤에 해당하는 CoClass 구현을 만듭니다. 이 마법사를 사용하면 소스 코드 에디터가 표시되므로 구현 클래스를 수정할 수 있습니다.

표 38.2 COM, Automation 및 ActiveX 객체 구현을 위한 C++Builder 마법사(계속)

마법사	구현되는 인터페이스	마법사가 수행하는 작업
ActiveForm	ActiveX 컨트롤과 동일한 인터페이스	ActiveX 컨트롤 마법사의 작업을 수행합니다. ActiveX 컨트롤 마법사의 기존 VCL 클래스를 대신하는 *TActiveForm* 자손을 만듭니다. 이 새 클래스를 사용하면 Windows 애플리케이션에서 폼을 설계하는 것과 같은 방식으로 Active Form 을 설계할 수 있습니다.
Transactional Object	*IUnknown*, *IDispatch*, *IObjectControl*	MTS 또는 COM+ 객체 정의를 포함하는 현재 프로젝트에 새 유닛을 추가합니다. 타입 라이브러리에 소유 GUID를 삽입하여 C++Builder가 객체를 제대로 설치할 수 있도록 하고 개발자에게 Type Library Editor를 표시하여 객체가 클라이언트에 노출하는 인터페이스를 정의할수 있도록 해 줍니다. 객체를 생성한 다음에는 각각 설치해야 합니다.
Property Page	*IUnknown*, *IPropertyPage*	폼 디자이너에서 설계할 수 있는 새 Attributes 페이지를 만듭니다.
COM+ Event Object	없음(기본값)	Type Library Editor를 사용하여 정의할 수 있는 COM+ 이벤트 객체를 만듭니다. COM+ Event Object 마법사는 다른 객체 마법사와 달리 이벤트 객체에 구현이 없으므로 구현 유닛을 만들지 않습니다. 구현은 event subscriber object에서 제공됩니다.
COM+ Subscription Object	없음(기본값)	선택된 이벤트 인터페이스를 구현하는 COM+ 구독 객체를 만듭니다.
Type Library	없음(기본값)	새 타입 라이브러리를 만들어 활성 프로젝트와 연결합니다.
ActiveX Library	없음(기본값)	새 ActiveX 또는 Com 서버 DLL을 만들고 필수적인 export 함수를 노출합니다.

추가 COM 객체를 추가하거나 기존 구현을 다시 구현할 수 있습니다. 새 객체를 추가하는 가장 쉬운 방법은 마법사를 두 번 사용하는 것입니다. 그 이유는 마법사가 타입 라이브러리와 구현 클래스 간의 연결을 설정하여 개발자가 Type Library Editor에서 변경한 내용을 구현 객체에 자동으로 적용할 수 있도록 하기 때문입니다.

마법사에 의해 작성된 코드

C++Builder의 마법사는 Microsoft ATL(Active Template Library)을 COM 지원의 기본으로 사용하는 코드를 작성합니다. ATL은 COM 애플리케이션 개발의 여러 가지 세부 구현을 처리하는 템플릿 클래스의 프레임워크입니다. ATL이 템플릿 기반이므로 DLL에 연결하지 않아도 됩니다. 대신, 객체 코드로 컴파일되는 ATL 헤더 파일을 프로젝트에 포함시켜야 합니다. C++Builder 마법사는 _ATL 접미사(Project1_ATL.cpp 및 Project1_ATL.h 등)가 있는 유닛 내에서 이러한 헤더 파일에 대한 include 문을 작성합니다.

참고 C++Builder의 Include/ATL 디렉토리에 있는 ATL 헤더 파일은 Microsoft에서 제공하는 ATL 헤더 파일과 약간 다릅니다. C++Builder의 컴파일러가 헤더를 컴파일할 수 있기 때문에 이러한 차이점이 있어야 합니다. 이러한 헤더를 다른 버전의 ATL로 대체하면 제대로 컴파일되지 않으므로 대체할 수 없습니다.

작성된 _ATL 유닛 헤더에는 ATL 파일 및 ATL 클래스가 VCL 클래스와 작동할 수 있도록 하는 추가 파일에 대한 include 문 외에 _Module이라는 전역 변수 선언도 포함되어 있습니다. _Module은 ATL 클래스인 *CComModule*의 인스턴스이며, CComModule은 스레딩과 등록을 처리하는 방법에 있어서 DLL과 EXE의 차이점을 애플리케이션의 나머지 부분이 인식하지 못하도록 합니다. 프로젝트 파일(Project1.cpp)에서 _Module은 C++Builder의 COM 등록 스타일을 지원하는 *CComModule*의 자손인 *TComModule*의 인스턴스에 할당됩니다. 일반적으로 이 객체는 직접 사용할 필요가 없습니다.

마법사는 또한 프로젝트 파일에 객체 맵을 추가합니다. 이러한 객체 맵은 다음과 유사한 일련의 ATL 매크로입니다.

```
BEGIN_OBJECT_MAP(ObjectMap)
  OBJECT_ENTRY(CLSID_MyObj, TMyObjImpl)
END_OBJECT_MAP()
```

BEGIN_OBJECT_MAP 줄 및 END_OBJECT_MAP 줄 사이의 각 엔트리는 클래스 ID 및 그 ATL 구현 간의 연결을 정의합니다. _Module 객체는 이 맵을 사용하여 컴포넌트를 등록합니다. 객체 맵은 ATL 객체 작성자 클래스에 의해서도 사용됩니다. 마법사를 사용하지 않고 애플리케이션에 COM 객체를 추가하는 경우에는 객체 맵을 업데이트하여 객체 맵이 제대로 등록되고 작성되도록 해야 합니다. 그러기 위해서는 OBJECT_ENTRY 매크로를 사용하는 또 다른 줄을 추가하고 클래스 ID와 구현 클래스 이름을 매개변수로 할당해야 합니다.

마법사는 작성 중인 특정한 타입의 COM 객체에 대해 구현 유닛을 만듭니다. 구현 유닛에는 COM 객체를 구현하는 클래스의 선언이 포함됩니다. 이 클래스는 작성 중인 객체의 타입에 따라 ATL 클래스 *CComObjectRootEx*, ATL 클래스 *CComCoClass* 및 기타 클래스의 직접적 또는 간접적인 자손입니다.

*CComCoClass*는 클래스 생성에 필요한 클래스 팩토리를 제공하며 개발자의 프로젝트 파일에 추가된 객체 맵을 사용합니다.

*CComObjectRootEx*는 *IUnknown*에 대한 기본 지원을 제공하는 템플릿 클래스이며 구현 유닛 헤더에서 찾을 수 있는 인터페이스 맵을 사용하여 *QueryInterface* 메소드를 구현합니다. 인터페이스 맵은 다음과 같습니다.

```
BEGIN_COM_MAP(TMyObjImpl)
  COM_INTERFACE_ENTRY(IMyObj)
END_COM_MAP()
```

인터페이스 맵의 각 항목은 *QueryInterface* 메소드에 의해 노출되는 인터페이스입니다. 구현 클래스에 인터페이스를 추가하는 경우에는 COM_INTERFACE_ENTRY 매크로를 사용하여 이를 인터페이스 맵에 추가하고 구현 클래스의 추가 조상으로 추가해야 합니다.

*CComObjectRootEx*는 참조 카운팅에 대한 기본 지원을 제공하지만 *AddRef* 및 *Release* 메소드를 선언하지는 않습니다. 이러한 메소드는 인터페이스 맵의 끝에 END_COM_MAP() 매크로를 통해 구현 클래스에 추가됩니다.

참고 ATL에 대한 자세한 내용은 Microsoft 설명서를 참조하십시오. 단, C++Builder의 COM 지원의 경우에는 등록, ActiveX 컨트롤(VCL 객체 기반) 또는 속성 페이지 지원을 위해 ATL을 사용하지 않습니다.

마법사는 또한 Project1_TLB 형식의 이름을 가진 타입 라이브러리와 그 연결 유닛을 만듭니다. Project1_TLB 유닛에는 타입 라이브러리에 정의된 타입 정의와 인터페이스를 사용하기 위해 애플리케이션이 필요로 하는 정의가 포함됩니다. 이 파일의 내용에 대한 자세한 내용은 40-5페이지의 "타입 라이브러리 정보를 임포트할 때 생성되는 코드"를 참조하십시오.

Type Library Editor를 사용하면 마법사에 의해 만들어진 인터페이스를 수정할 수 있습니다. 이 작업을 수행하면 구현 클래스가 자동으로 업데이트되어 해당되는 변경 사항을 반영합니다. 따라서 개발자는 만들어진 메소드의 바디를 채워 구현을 완성하기만 하면 됩니다.

장

39

타입 라이브러리 사용

이 장에서는 C++Builder의 Type Library Editor를 사용하여 타입 라이브러리를 만들고 편집하는 방법에 대해 설명합니다. 타입 라이브러리는 COM 객체가 노출하는 데이터 타입, 인터페이스, 멤버 함수 및 객체 클래스에 대한 정보가 들어 있는 파일입니다. 타입 라이브러리는 서버에서 사용 가능한 객체 및 인터페이스 타입을 식별하는 방법을 제공합니다. 타입 라이브러리를 사용하는 이유와 시기에 대한 자세한 내용은 38-16페이지의 "타입 라이브러리"를 참조하십시오.

타입 라이브러리는 다음 중 하나 이상을 포함할 수 있습니다.

- 알리아스, 열거, 구조체 및 합집합과 같은 사용자 지정 데이터 타입에 대한 정보
- 인터페이스, dispinterface 또는 CoClass와 같은 하나 이상의 COM 요소에 대한 설명. 이러한 각 설명을 일반적으로 *타입 정보*라고 합니다.
- 외부 모듈에 정의된 상수 및 메소드 설명
- 다른 타입 라이브러리의 타입 설명에 대한 참조

COM 애플리케이션이나 ActiveX 라이브러리로 타입 라이브러리를 포함하면 COM의 타입 라이브러리 도구와 인터페이스를 통해 애플리케이션에 있는 객체에 대한 정보를 다른 애플리케이션이나 프로그래밍 도구에서 사용하게 할 수 있습니다.

기존의 개발 도구를 사용할 경우 IDL(Interface Definition Language)로 스크립트를 작성하여 타입 라이브러리를 만든 다음 컴파일러를 통해 이 스크립트를 실행합니다. 그러나 Type Library Editor를 사용하면 이러한 프로세스 중 일부가 자동화되므로 타입 라이브러리를 쉽게 만들고 수정할 수 있습니다.

C++Builder의 마법사를 사용하여 타입(ActiveX 컨트롤, Automation 객체, 원격 데이터 모듈 등)에 관계 없이 COM 서버를 만들 때 마법사가 자동으로 타입 라이브러리를 생성합니다. 클라이언트에 노출하는 속성과 메소드를 타입 라이브러리에서 정의하므로 생성된 객체를 사용자 정의할 때 수행하는 대부분의 작업은 타입 라이브러리에서 시작됩니다. Type Library Editor를 사용하여 마법사가 생성한 CoClass의 인터페이스를 변경합니다. Type Library Editor가 객체의 구현 유닛을 자동으로 업데이트하므로 생성된 메소드의 바디를 채우기만 하면 됩니다.

Type Library Editor

개발자는 Type Library Editor를 사용하여 COM 객체에 대한 타입 정보를 조사하고 만들 수 있습니다. Type Library Editor를 사용하면 인터페이스, CoClass 및 타입을 정의하는 작업을 중앙 집중화하여 COM 객체를 개발하고, 새 인터페이스에 대한 GUID를 획득하고, 인터페이스를 CoClass와 연결하고, 구현 유닛을 업데이트하는 등의 작업을 매우 쉽게 수행할 수 있습니다.

Type Library Editor는 타입 라이브러리 내용을 나타내는 두 가지 유형의 파일을 출력합니다.

표 39.1 Type Library Editor 파일

File	설명
.TLB 파일	바이너리 타입 라이브러리 파일. 디폴트로, 타입 라이브러리는 자동으로 애플리케이션에 리소스로 컴파일되므로 이 파일은 사용할 필요가 없습니다. 그러나 이 파일을 사용하여 명시적으로 타입 라이브러리를 다른 프로젝트로 컴파일하거나, .exe 또는 .ocx에서 타입 라이브러리를 각각 배포할 수 있습니다. 자세한 내용은 39-13 페이지의 "기존의 타입 라이브러리 열기" 및 39-19페이지의 "타입 라이브러리 배포"를 참조하십시오.
_TLB 유닛	이 유닛(.cpp 및 .h 파일)은 애플리케이션에서 사용할 수 있도록 타입 라이브러리의 내용을 반영합니다. 또한 애플리케이션이 타입 라이브러리에 정의된 요소를 사용하기 위해 필요로 하는 모든 선언이 이 유닛에 들어 있습니다. 코드 에디터에서 이 파일을 열 수는 있지만 편집해서는 안됩니다. 이 파일은 Type Library Editor에 의해 유지 보수되므로 파일 내용을 변경하면 Type Library Editor가 변경 내용을 겹쳐 씁니다. 이 파일의 내용에 대한 자세한 내용은 40-5페이지의 "타입 라이브러리 정보를 임포트할 때 생성되는 코드"를 참조하십시오.

Type Library Editor의 구성 요소

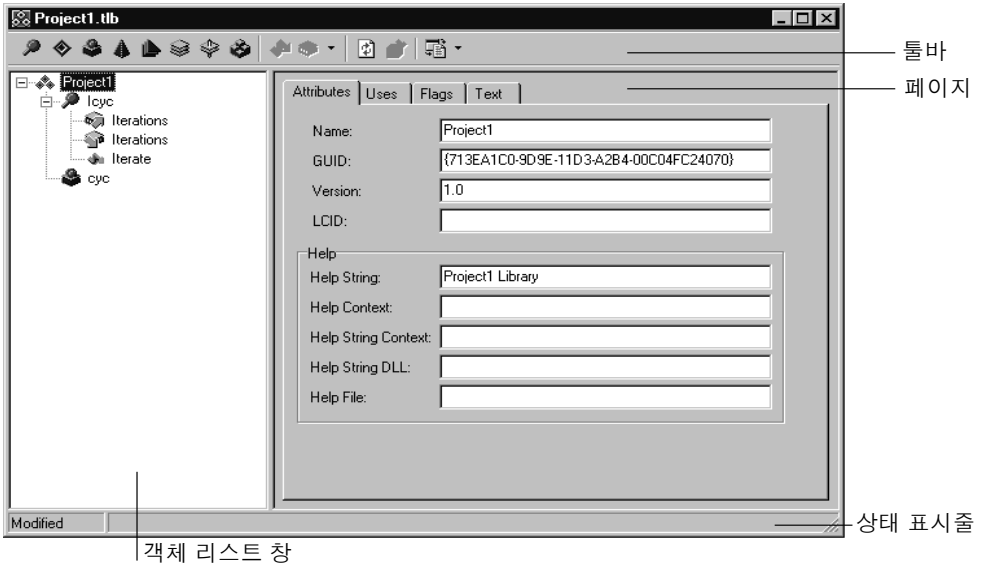
Type Library Editor의 주요 구성 요소는 표 39.2에 설명되어 있습니다.

표 39.2 Type Library Editor의 구성 요소

구성 요소	설명
툴바	타입 라이브러리에 새 타입, CoClass, 인터페이스 및 인터페이스 멤버를 추가하는 버튼이 있습니다. 또한 툴바에는 타입 라이브러리에 있는 정보를 사용하여 구현 유닛을 새로 고치고, 타입 라이브러리를 등록하고, IDL 파일을 저장하는 데 사용하는 버튼도 있습니다.
객체 리스트 창	타입 라이브러리에 있는 모든 기존 요소를 표시합니다. 객체 리스트 창에 있는 항목을 클릭하면 해당 객체에 유효한 페이지가 표시됩니다.
상태 표시줄	타입 라이브러리에 잘못된 타입을 추가하려고 하면 구문 오류를 표시합니다.
페이지	선택한 객체에 대한 정보를 표시합니다. 선택한 객체의 타입에 따라 다른 페이지가 표시됩니다.

이러한 구성 요소는 그림 39.1에 표시되어 있으며, 이 그림은 cyc라는 COM 객체에 대한 타입 정보를 표시하는 Type Library Editor를 나타낸 것입니다.

그림 39.1 Type Library Editor



툴바

Type Library Editor 맨 위에 있는 Type Library Editor 툴바에는 클릭하면 타입 라이브러리에 새 객체를 추가할 수 있는 버튼이 있습니다.

첫 번째 그룹의 버튼을 사용하여 타입 라이브러리에 요소를 추가할 수 있습니다. 툴바 버튼을 클릭하면 객체 리스트 창에 해당 요소의 아이콘이 표시됩니다. 그러면 오른쪽 창에서 해당 어

트리뷰트(attribute)를 사용자 정의할 수 있습니다. 선택하는 아이콘 유형에 따라 다른 정보 페이지가 오른쪽에 나타납니다.

다음 표는 타입 라이브러리에 추가할 수 있는 요소를 나열한 것입니다.

아이콘	의미
	인터페이스 설명
	dispinterface 설명
	CoClass
	열거
	알리아스
	레코드
	합집합
	모듈

객체 리스트 창에서 위에 나열된 요소 중 하나를 선택하면 두 번째 그룹의 버튼이 해당 요소에 유효한 멤버를 표시합니다. 예를 들면, Interface를 선택할 경우 인터페이스 정의에 메소드와 속성을 추가할 수 있으므로 두 번째 박스의 Method 아이콘과 Property 아이콘이 활성화됩니다. Enum을 선택하면 두 번째 그룹의 버튼이 변경되어 Const 멤버를 표시합니다. Const 멤버는 Enum 타입 정보에 대해서만 유효합니다.

다음 표는 객체 리스트 창에 있는 요소에 추가할 수 있는 멤버를 나열한 것입니다.

아이콘	의미
	인터페이스, dispinterface, 또는 모듈에 있는 엔트리 포인트의 메소드
	인터페이스 또는 dispinterface의 속성
	쓰기 전용 속성(속성 버튼의 드롭다운 리스트에서 사용 가능)
	읽기/쓰기 속성(속성 버튼의 드롭다운 리스트에서 사용 가능)
	읽기 전용 속성(속성 버튼의 드롭다운 리스트에서 사용 가능)
	레코드 또는 합집합의 필드
	열거 또는 모듈의 상수

세 번째 그룹의 버튼을 사용하면 39-17페이지의 "타입 라이브러리 정보 저장 및 등록"에서 설명한 대로 타입 라이브러리를 새로 고치거나, 등록하거나, 엑스포트할 수 있습니다(타입 라이브러리를 IDL 파일로 저장).

객체 리스트 창

객체 리스트 창의 트리 뷰에 현재 타입 라이브러리의 모든 요소가 표시됩니다. 트리의 루트는 타입 라이브러리 자체를 나타내며, 다음 아이콘으로 표시됩니다.



타입 라이브러리에 있는 요소는 타입 라이브러리 노드의 자손입니다.

그림 39.2 객체 리스트 창



타입 라이브러리 자체를 비롯하여 이러한 요소 중 하나를 선택하면 오른쪽에 있는 타입 정보 페이지가 변경되어 해당 요소에 대한 관련 정보만 표시합니다. 이 페이지를 사용하여 선택한 요소의 정의와 속성을 편집할 수 있습니다.

마우스 오른쪽 버튼을 클릭해 객체 리스트 창 컨텍스트 메뉴를 표시하여 객체 리스트 창에서 요소를 처리할 수 있습니다. 이 메뉴에는 새 요소를 추가하거나 Type Library Editor 모양을 사용자 정의하는 명령뿐 아니라, Windows 클립보드를 사용하여 기존의 요소를 이동하거나 복사할 수 있는 명령도 포함되어 있습니다.

상태 표시줄

타입 라이브러리를 편집하거나 저장할 때 상태 표시줄 창에 구문, 변환 오류 및 경고가 나열됩니다.

예를 들면, Type Library Editor에서 지원하지 않는 타입을 지정하면 구문 오류가 표시됩니다. Type Library Editor가 지원하는 전체 타입 리스트를 보려면 39-11 페이지의 "유효한 타입"을 참조하십시오.

타입 정보 페이지

객체 리스트 창에서 요소를 선택하면 선택한 요소에 유효한 타입 정보 페이지가 Type Library Editor에 표시됩니다. 객체 리스트 창에서 선택한 요소에 따라 다음과 같이 다른 페이지가 표시됩니다.

표 39.3 타입 라이브러리 페이지

타입 정보 요소	타입 정보 페이지	페이지 내용
Type Library	Attributes	타입 라이브러리를 도움말에 연결하는 정보, 타입 라이브러리의 이름, 버전 및 GUID
	Uses	이 요소에 대한 정의가 포함된 다른 타입 라이브러리 리스트
	Flags	다른 애플리케이션이 타입 라이브러리를 사용하는 방법을 결정하는 플래그
	Text	타입 라이브러리 자체를 정의하는 모든 정의 및 선언(아래 내용 참조)
Interface	Attributes	인터페이스의 이름, 버전 및 GUID, 조상 인터페이스 이름, 인터페이스를 도움말에 연결하는 정보
	Flags	인터페이스가 숨김, 이중, Automation 호환 및/또는 확장 가능인지 표시하는 플래그
	Text	인터페이스에 대한 정의 및 선언(아래 내용 참조)
Dispinterface	Attributes	인터페이스의 이름, 버전 및 GUID, 인터페이스를 도움말에 연결하는 정보
	Flags	Dispinterface가 숨김, 이중 및/또는 확장 가능인지 표시하는 플래그
	Text	Dispinterface에 대한 정의 및 선언 (아래 내용 참조)
CoClass	Attributes	CoClass의 이름, 버전 및 GUID, CoClass를 도움말에 연결하는 정보
	Implements	CoClass의 어트리뷰트(attribute) 및 CoClass가 구현하는 인터페이스 리스트
	COM+	트랜잭션 모델, 호출 동기화, just-in-time 활성화, 객체 풀링 등과 같은 트랜잭션 객체의 어트리뷰트. COM+ 이벤트 객체의 어트리뷰트도 포함
	Flags	클라이언트가 인스턴스를 만들고 사용하는 방법, 브라우저에서 CoClass가 사용자에게 표시되는지 여부, CoClass가 ActiveX 컨트롤인지 여부, CoClass가 집계될 수 있는지 여부(합성의 일부로 동작) 등을 포함하여 CoClass의 다양한 어트리뷰트를 표시하는 플래그
	Text	CoClass에 대한 정의 및 선언(아래 내용 참조)
Enumeration	Attributes	열거의 이름, 버전 및 GUID, 열거를 도움말에 연결하는 정보
	Text	열거 타입에 대한 정의 및 선언(아래 내용 참조)
Alias	Attributes	열거의 이름, 버전 및 GUID, 알리아스가 나타내는 타입, 알리아스를 도움말에 연결하는 정보
	Text	알리아스에 대한 정의 및 선언(아래 내용 참조)

표 39.3 타입 라이브러리 페이지(계속)

타입 정보 요소	타입 정보 페이지	페이지 내용
Record	Attributes	레코드의 이름, 버전 및 GUID, 레코드를 도움말에 연결하는 정보
	Text	레코드에 대한 정의 및 선언(아래 내용 참조)
Union	Attributes	합집합의 이름, 버전 및 GUID, 합집합을 도움말에 연결하는 정보
	Text	합집합에 대한 정의 및 선언(아래 내용 참조)
Module	Attributes	모듈의 이름, 버전, GUID 및 관련 DLL, 모듈을 도움말에 연결하는 정보
	Text	모듈에 대한 정의 및 선언(아래 내용 참조)
Method	Attributes	이름, 디스패치 ID 또는 DLL 엔트리 포인트, 메소드를 도움말에 연결하는 정보
	Parameters	메소드 반환 타입 및 해당 타입을 가진 모든 매개변수와 변경자 리스트
	Flags	클라이언트가 메소드를 보고 사용하는 방법, 이 메소드가 인터페이스의 디폴트 메소드인지 여부, 이 메소드를 바꿀 수 있는지 여부 등을 표시하는 플래그
	Text	메소드에 대한 정의 및 선언(아래 내용 참조)
Property	Attributes	속성 액세스 메소드(getter vs. setter)의 이름, 디스패치 ID, 타입 및 속성을 도움말에 연결하는 정보
	Parameters	속성 액세스 메소드 반환 타입 및 해당 타입을 가진 모든 매개변수와 변경자 리스트
	Flags	클라이언트가 속성을 보고 사용하는 방법, 이 속성이 인터페이스의 기본 속성인지 여부, 속성을 바꾸거나 연결할 수 있는지 여부 등을 표시하는 플래그
	Text	속성 액세스 메소드에 대한 정의 및 선언(아래 내용 참조)
Const	Attributes	모듈 const의 이름, 값, 타입 및 const를 도움말에 연결하는 정보
	Flags	클라이언트가 상수를 보고 사용하는 방법, 이 상수가 기본값을 나타내는지 여부, 상수가 연결 가능한지 여부 등을 표시하는 플래그
	Text	상수에 대한 정의 및 선언(아래 내용 참조)
Field	Attributes	이름, 타입 및 필드를 도움말에 연결하는 정보
	Flags	클라이언트가 필드를 보고 사용하는 방법, 이 필드가 기본값을 나타내는지 여부, 필드가 연결 가능한지 여부 등을 표시하는 플래그
	Text	필드에 대한 정의 및 선언(아래 내용 참조)

참고 타입 정보 페이지에서 설정할 수 있는 다양한 옵션에 대한 자세한 내용은 Type Library Editor의 온라인 도움말을 참조하십시오.

타입 정보의 각 페이지를 사용하여 표시되는 값을 보거나 편집할 수 있습니다. 대부분의 페이지는 정보를 컨트롤 집합으로 구성하므로 해당 선언의 구문을 모를 경우에도 값을 입력하거나 리스트에서 값을 선택할 수 있습니다. 따라서 제한된 집합에서 값을 지정할 때 표기 오류와 같은 여러 가지 사소한 실수를 방지할 수 있습니다. 그러나 선언에 직접 입력하는 것이 더 빠릅니다. 선언에 직접 입력하려면 Text 페이지를 사용하십시오.

모든 타입 라이브러리 요소에는 요소의 구문을 표시하는 Text 페이지가 있습니다. 이 구문은 Microsoft Interface Definition Language의 IDL 부분 집합입니다. 요소의 다른 페이지에서 변경한 내용은 Text 페이지에 반영됩니다. Text 페이지에서 직접 코드를 추가하면 변경 내용이 Type Library Editor의 다른 페이지에 반영됩니다.

현재 Type Library Editor에서 지원하지 않는 식별자를 추가할 경우 에디터가 구문 오류를 생성합니다. 현재 에디터는 타입 라이브러리 지원에 관련된 식별자만 지원합니다(Microsoft IDL 컴파일러가 C++ 코드 생성을 위해 사용하는 RPC 지원이나 생성 또는 마샬링 지원은 제외).

타입 라이브러리 요소

타입 라이브러리 인터페이스는 처음에 매우 복잡해 보일 수 있습니다. 타입 라이브러리 인터페이스가 각각의 특성을 가진 많은 요소에 대한 정보를 표시하기 때문입니다. 그러나 이러한 특성 중 대부분은 모든 요소에 대해 공통적입니다. 예를 들면, 타입 라이브러리를 비롯한 모든 요소는 다음을 포함합니다.

- 요소를 설명하고, 코드에서 요소를 참조하는 데 사용되는 이름
- COM이 요소를 식별하는 데 사용하는, 고유한 128비트 값인 GUID(Globally Unique Identifier). 타입 라이브러리 자체, CoClass 및 인터페이스에 대해서는 항상 GUID가 제공되어야 합니다. 그 외에는 옵션입니다.
- 요소의 여러 버전을 구별하는 버전 번호. 버전 번호는 항상 옵션이지만, 일부 도구에서는 버전 번호가 없으면 CoClass 와 인터페이스를 사용할 수 없으므로, CoClass 와 인터페이스에 대해서는 버전 번호가 제공되어야 합니다.
- 요소를 Help 항목에 연결하는 정보. Help String 및 Help Context나 Help String Context 값이 여기에 포함됩니다. Help Context는 타입 라이브러리가 독립 실행형 도움말 파일을 가지고 있는 일반적인 Windows 도움말 시스템에 사용됩니다. 각각의 DLL에서 도움말을 제공하는 경우에는 Help String Context가 사용됩니다. Help Context나 Help String Context는 타입 라이브러리의 Attributes 페이지에서 지정된 도움말 파일이나 DLL을 참조합니다. 이 정보는 항상 옵션입니다.

인터페이스

인터페이스는 가상 함수 테이블(vtable)을 통해 액세스해야 하는 객체에 대한 메소드 및 'get' 함수와 'set' 함수로 표현되는 모든 속성을 설명합니다. 이중으로 플래그 지정될 경우 인터페이스는 *IDispatch*에서 상속되며, 객체가 초기 연결되는 vtable 액세스와 OLE 자동화를 통한 런타임 연결을 모두 제공합니다. 디폴트로, 타입 라이브러리는 추가하는 모든 인터페이스를 이중으로 플래그 지정합니다.

인터페이스는 할당된 멤버 즉, 메소드와 속성일 수 있습니다. 이러한 인터페이스는 객체 리스트 창에 인터페이스 노드의 자식으로 표시됩니다. 속성의 원본으로 사용하는 데이터를 읽고 쓰는 데 사용되는 'get' 메소드와 'set' 메소드가 인터페이스에 대한 속성을 표시합니다. 이러한 속성은 해당 용도를 나타내는 특수한 아이콘을 사용하여 트리 뷰에 표시됩니다.

참고 속성이 Write By Reference로 지정되면 해당 속성이 값에 의해서가 아니라 포인터로 전달된다는 의미입니다. Visual Basic 등 일부 애플리케이션에서는 Write By Reference(있을 경우)를 사용하여 성능을 최적화합니다. 값이 아니라 참조를 사용하여 속성을 전달하려면 *By Reference Only* 속성 타입을 사용하십시오. 값과 참조를 사용하여 속성을 전달하려면 Read|Write|Write By Ref를 선택합니다. 이 메뉴를 호출하려면 툴바에서 속성 아이콘 옆에 있는 화살표를 선택하십시오.

툴바 버튼이나 객체 리스트 창 컨텍스트 메뉴를 사용하여 속성이나 메소드를 추가한 다음에는 속성이나 메소드를 선택하고 타입 정보 페이지를 사용하여 해당 구문과 어트리뷰트(attribute)를 설명합니다.

Attributes 페이지를 사용하여 속성이나 메소드에 이름과 디스패치 ID를 부여할 수 있습니다. 그래서 이 작업을 IDispatch 사용이라고도 합니다. 속성의 경우에는 타입도 할당합니다. 함수 시그니처는 Parameters 페이지를 사용하여 만듭니다. 이 페이지에서 매개변수를 추가, 제거, 재배열하고, 타입과 변경자를 설정하고, 함수 반환 타입을 지정할 수 있습니다.

인터페이스에 속성과 메소드를 할당하면 관련 CoClass에 속성과 메소드가 암시적으로 할당됩니다. 이 때문에 개발자는 Type Library Editor를 사용하여 CoClass에 속성과 메소드를 직접 추가할 수 없습니다.

Dispinterface

객체의 속성과 메소드를 설명할 때는 dispinterface 보다 인터페이스가 더 일반적으로 사용됩니다. Dispinterface는 동적 연결을 통해서만 액세스할 수 있지만, 인터페이스는 vtable을 통한 정적 연결을 지원합니다.

인터페이스에 메소드와 속성을 추가하는 것과 동일한 방법으로 dispinterface에 메소드와 속성을 추가할 수 있습니다. 그러나 dispinterface에 대한 속성을 만들 때는 함수 종류나 매개변수 타입을 지정할 수 없습니다.

CoClass

CoClass는 하나 이상의 인터페이스를 구현하는 고유한 COM 객체를 설명합니다. CoClass를 정의할 때 객체에 대해 기본적으로 구현된 인터페이스를 지정해야 하며, 이벤트에 대한 디폴트 소스인 dispinterface를 선택적으로 지정합니다. Type Library Editor에서는 CoClass에 속성이나 메소드를 추가하지 않는다는 점에 주의하십시오. 속성과 메소드는 Implements 페이지를 사용하여 CoClass와 연결되는 인터페이스를 통해 클라이언트에 노출됩니다.

타입 정의

열거, 알리아스, 레코드 및 합집합은 모두 타입 라이브러리 어디서나 사용할 수 있는 타입을 선언합니다.

열거는 상수 리스트로 구성되며, 각 상수는 숫자여야 합니다. 대개 숫자 입력은 10진수나 16진수 형식의 정수이며 기본값은 0입니다. 객체 리스트 창에서 열거를 선택하고 툴바에서 Const 버튼을 클릭하거나, 객체 리스트 창 컨텍스트 메뉴에서 New|Const 명령을 선택하여 열거에 상수를 추가할 수 있습니다.

참고 열거에 대한 도움말 문자열을 제공하여 열거의 의미를 더 명확하게 하는 것이 좋습니다. 다음은 마우스 버튼에 대한 열거 타입의 예제 항목이며, 각 열거 요소에 대한 도움말 문자열이 포함되어 있습니다.

```
typedef enum TxMouseButton
```

```
{
   [helpstring("mbLeft”)]
   mbLeft = 0,
   [helpstring("mbRight)]
   mbRight = 1.
   [helpstring("mbMiddle)]
   mbMiddle = 2
} TxMouseButton;
```

알리아스는 타입에 대한 알리아스(typedef)를 만듭니다. 알리아스를 사용하여 레코드나 합집합과 같은 다른 타입 정보에 사용할 타입을 정의할 수 있습니다. Attributes 페이지에서 Type 어트리뷰트(attribute)를 설정하여 원본으로 사용하는 타입 정의와 알리아스를 연결합니다.

레코드는 C 스타일의 구조체입니다. 레코드는 구조체 멤버나 필드 리스트로 구성됩니다. 합집합은 C 스타일의 *합집합*을 정의합니다. 레코드와 마찬가지로, 합집합은 구조체 멤버나 필드 리스트로 구성됩니다. 그러나 레코드의 멤버와 달리, 합집합의 각 멤버는 하나의 논리 값만 저장되도록 동일한 실제 주소를 사용합니다.

객체 리스트 창에서 레코드나 합집합을 선택하고 툴바에서 필드 버튼을 클릭하거나 마우스 오른쪽 버튼을 클릭하고 객체 리스트 창 컨텍스트 메뉴에서 필드를 선택하여 레코드나 합집합에 필드를 추가합니다. 각 필드에는 이름과 타입이 있습니다. Attributes 페이지를 사용하여 필드를 선택하고 값을 할당하여 이름과 타입을 할당합니다. C *struct*를 사용하는 것과 같이 옵션 태그를 사용하여 레코드와 합집합을 정의할 수 있습니다.

멤버는 기본 타입이거나 레코드를 정의하기 전에 알리아스를 사용하여 타입을 지정할 수 있습니다.

참고 C++Builder는 switch_type, first_is, last_is 등 구조체 및 합집합에 대한 키워드와 관련된 마샬링을 지원하지 않습니다.

모듈

모듈은 일반적으로 DLL 엔트리 포인트 집합인 함수 그룹을 정의합니다. 다음 방법으로 모듈을 정의합니다.

- Attributes 페이지에 표시되는 DLL을 지정합니다.
- 툴바나 객체 리스트 창 컨텍스트 메뉴를 사용하여 메소드와 상수를 추가합니다. 그런 다음 메소드나 상수의 경우 객체 리스트 창에서 메소드나 상수를 선택하고 Attributes 페이지에서 값을 설정하여 메소드나 상수의 어트리뷰트를 정의해야 합니다.

모듈 메소드의 경우 Attributes 페이지를 사용하여 이름과 DLL 엔트리 포인트를 할당해야 합니다. Parameters 페이지를 사용하여 함수의 매개변수와 반환 타입을 선언합니다.

모듈 상수의 경우 Attributes 페이지를 사용하여 이름, 타입 및 값을 지정합니다.

참고 Type Library Editor는 모듈 관련 선언이나 구현을 생성하지 않습니다. 지정된 DLL은 각각의 프로젝트로 만들어져야 합니다.

Type Library Editor 사용

Type Library Editor를 사용하여 새 타입 라이브러리를 만들거나, 기존의 타입 라이브러리를 편집할 수 있습니다. 대개 애플리케이션 개발자는 마법사를 사용하여 타입 라이브러리에서 노출되는 객체를 만들며, 이를 통해 C++Builder에서는 자동으로 타입 라이브러리를 생성합니

다. 그러면 자동 생성된 타입 라이브러리가 Type Library Editor에서 열리고 인터페이스를 정의 또는 수정하거나, 타입 정의를 추가하는 등의 작업을 수행할 수 있습니다.

그러나 마법사를 사용하여 객체를 정의하지 않는 경우에도 Type Library Editor를 사용하여 새 타입 라이브러리를 정의할 수 있습니다. 이 경우에 Type Library Editor는 마법사가 타입 라이브러리와 연결하지 않은 CoClass에 대한 코드를 생성하지 않으므로 직접 구현 클래스를 만들어야 합니다.

아래에 설명한 대로 에디터는 타입 라이브러리에 있는 유효한 타입의 부분 집합을 지원합니다.

이 단원의 마지막 항목에서는 다음 작업을 수행하는 방법을 설명합니다.

- 새 타입 라이브러리 생성
- 기존의 타입 라이브러리 열기
- 타입 라이브러리에 인터페이스 추가
- 인터페이스 수정
- 타입 라이브러리에 속성 및 메소드 추가
- 타입 라이브러리에 CoClass 추가
- CoClass에 인터페이스 추가
- 타입 라이브러리에 열거 추가
- 타입 라이브러리에 알리아스 추가
- 타입 라이브러리에 레코드 또는 합집합 추가
- 타입 라이브러리에 모듈 추가
- 타입 라이브러리 정보 저장 및 등록

유효한 타입

Type Library Editor가 타입 라이브러리에서 지원하는 IDL 타입은 다음과 같습니다. Automation 호환 열은 Automation 또는 Dispinterface 플래그가 선택된 인터페이스가 타입을 사용할 수 있는지 여부를 지정합니다. 이러한 타입은 COM이 타입 라이브러리를 통해 자동으로 마샬링하는 타입입니다.

IDL 타입	가변 타입	Automation 호환	설명
short	VT_I2	예	부호 있는 2바이트 정수
long	VT_I4	예	부호 있는 4바이트 정수
single	VT_R4	예	4바이트 실수
double	VT_R8	예	8바이트 실수
CURRENCY	VT_CY	예	통화

IDL 타입	가변 타입	Automation 호환	설명
DATE	VT_DATE	예	날짜
BSTR	VT_BSTR	예	바이너리 문자열
IDispatch	VT_DISPATCH	예	IDispatch 인터페이스에 대한 포인터
SCODE	VT_ERROR	예	OLE 오류 코드
VARIANT_BOOL	VT_BOOL	예	true = –1, false = 0
VARIANT	VT_VARIANT	예	OLE Variant에 대한 포인터
IUnknown	VT_UNKNOWN	예	IUnknown 인터페이스에 대한 포인터
DECIMAL	VT_DECIMAL	예	16바이트 고정 소수점
byte	VT_I1	아니오*	부호 있는 1바이트 정수
unsigned char	VT_UI1	예	부호 없는 1바이트 정수
unsigned short	VT_UI2	아니오*	부호 없는 2바이트 정수
unsigned long	VT_UI4	아니오*	부호 없는 4바이트 정수
__int64	VT_I8	아니오	부호 있는 8바이트 실수
uint64	VT_UI8	아니오	부호 없는 8바이트 실수
int	VT_INT	아니오*	시스템에 종속적인 정수 (Win32=Integer)
unsigned int	VT_UINT	아니오*	시스템에 종속적인 부호 없는 정수
void	VT_VOID	예	C 스타일 VOID
HRESULT	VT_HRESULT	아니오	32비트 오류 코드
SAFEARRAY	VT_SAFEARRAY	예	OLE Safe Array
LPSTR	VT_LPSTR	아니오	Null 종료 문자열
LPWSTR	VT_LPWSTR	아니오	와이드 Null 종료 문자열

* 일부 애플리케이션과 Automation이 호환될 수도 있습니다.

참고 unsigned char 타입(VT_UI1)은 Automation과 호환되지만, 많은 Automation 서버가 이 값을 제대로 처리하지 못하므로 Variant나 OleVariant에서는 허용되지 않습니다.

참고 CORBA 개발에 대해 유효한 타입에 대해서는 31-5페이지의 "객체 인터페이스 정의"를 참조하십시오.

이러한 IDL 타입 외에도 라이브러리 또는 참조된 라이브러리에서 정의된 인터페이스와 타입이 타입 라이브러리에서 사용될 수 있습니다.

Type Library Editor는 타입 정보를 생성된 타입 라이브러리 파일(.TLB)에 바이너리 형식으로 저장합니다.

SafeArray

COM에서는 배열이 SafeArray라는 특수한 데이터 타입을 통해 전달되어야 합니다. 이 작업을 위해 특수한 COM 함수를 호출하여 SafeArray를 만들고 삭제할 수 있으며, SafeArray의 모든 요소가 유효한 Automation 호환 타입이어야 합니다.

Type Library Editor에서 *SafeArray*가 해당 요소의 타입을 지정해야 합니다. 예를 들면, Text 페이지의 다음 줄에서 요소 타입이 long인 *SafeArray* 매개변수를 사용하여 메소드를 선언합니다.

```
HRESULT _stdcall HighlightLines(SAFEARRAY(long) Lines);
```

참고 Type Library Editor에서 *SafeArray* 타입을 선언할 때 요소 타입을 지정해야 하지만 생성된 _TLB 유닛에 있는 선언이 요소 타입을 표시하지 않습니다.

새 타입 라이브러리 생성

특정 COM 객체에 대해 독립적인 타입 라이브러리를 만들 수 있습니다. 예를 들면, 여러 다른 타입 라이브러리에서 사용하는 타입 정의가 들어 있는 타입 라이브러리를 정의할 수 있습니다. 그러면 기본적인 정의의 타입 라이브러리를 만들어 다른 타입 라이브러리의 Uses 페이지에 추가할 수 있습니다.

또한 아직 구현되지 않는 객체에 대한 타입 라이브러리를 만들 수도 있습니다. 타입 라이브러리에 인터페이스 정의가 포함되면 COM Object 마법사를 사용하여 CoClass와 구현을 생성할 수 있습니다.

다음과 같은 방법으로 새 타입 라이브러리를 만듭니다.

1. File|New|Other를 선택하여 New Items 다이얼로그 박스를 엽니다.
2. ActiveX 페이지를 선택합니다.
3. Type Library 아이콘을 선택합니다.
4. OK를 선택합니다.

   타입 라이브러리 이름을 입력하라는 메시지와 함께 Type Library Editor가 열립니다.
5. 타입 라이브러리 이름을 입력합니다. 타입 라이브러리에 요소를 추가하여 계속합니다.

기존의 타입 라이브러리 열기

마법사를 사용하여 ActiveX 컨트롤, Automation 객체, Active 폼, Active Server Page 객체, COM 객체, 트랜잭션 객체, 원격 데이터 모듈 또는 트랜잭션 데이터 모듈을 만들 때 구현 유닛을 사용하여 자동으로 타입 라이브러리가 만들어집니다. 또한 시스템에서 사용할 수 있는 다른 제품(서버)과 연결된 타입 라이브러리가 있을 수 있습니다.

다음과 같은 방법으로 현재 프로젝트의 일부가 아닌 타입 라이브러리를 엽니다.

1. IDE의 메인 메뉴에서 File|Open을 선택합니다.
2. Open 다이얼로그 박스에서 File Type을 타입 라이브러리로 설정합니다.
3. 원하는 타입 라이브러리 파일로 이동한 다음 Open을 선택합니다.

다음과 같은 방법으로 현재 프로젝트에 연결된 타입 라이브러리를 엽니다.

1. View|Type Library를 선택합니다.

인터페이스, CoClass 및 타입 라이브러리의 다른 요소(열거, 속성, 메소드 등)를 추가할 수 있습니다.

팁 클라이언트 애플리케이션을 작성할 때는 타입 라이브러리를 열 필요가 없습니다. 타입 라이브러리 자체가 아니라, Type Library Editor가 타입 라이브러리에서 만드는 *Project*_TLB 유닛만 있으면 됩니다. 이 파일을 클라이언트 프로젝트에 직접 추가하거나, 시스템에 타입 라이브러리가 등록된 경우에는 Import Type Library 다이얼로그 박스(Project|Import Type Library)를 사용할 수 있습니다.

타입 라이브러리에 인터페이스 추가

다음과 같은 방법으로 인터페이스를 추가합니다.

1 툴바에서 인터페이스 아이콘을 클릭합니다.

객체 리스트 창에 인터페이스가 추가되고 이름을 추가하라는 메시지가 표시됩니다.

2 인터페이스 이름을 입력합니다.

새 인터페이스에는 필요할 경우 수정할 수 있는 디폴트 어트리뷰트(attribute)가 있습니다.

인터페이스의 용도에 맞게 속성(getter/setter 함수로 표시)과 메소드를 추가할 수 있습니다.

타입 라이브러리를 사용하여 인터페이스 수정

인터페이스나 dispinterface를 만든 다음 수정할 수 있는 여러 가지 방법이 있습니다.

- 변경할 정보가 포함된 타입 정보 페이지를 사용하여 인터페이스 어트리뷰트를 변경할 수 있습니다. 객체 리스트 창에서 인터페이스를 선택한 다음 적절한 타입 정보 페이지에 있는 컨트롤을 사용합니다. 예를 들면, Attributes 페이지를 사용하여 부모 인터페이스를 변경하거나, Flags 페이지를 사용하여 이중 인터페이스로 표시할지 여부를 변경할 수 있습니다.
- 객체 리스트 창에서 인터페이스를 선택한 다음 Text 페이지에서 선언을 편집하여 인터페이스 선언을 직접 편집할 수 있습니다.
- 인터페이스에 속성과 메소드를 추가할 수 있습니다(다음 단원 참조).
- 해당 타입 정보를 변경하여 이미 인터페이스에 지정된 속성과 메소드를 수정할 수 있습니다.
- 객체 리스트 창에서 CoClass를 선택하고 Implements 페이지에서 마우스 오른쪽 버튼을 클릭한 다음 Insert Interface를 선택하여 인터페이스를 CoClass에 연결할 수 있습니다.

인터페이스가 마법사에 의해 생성된 CoClass에 연결될 경우 툴바의 Refresh 버튼을 클릭하여 Type Library Editor를 통해 구현 파일에 변경 내용을 적용할 수 있습니다.

인터페이스 또는 dispinterface에 속성 및 메소드 추가

다음과 같은 방법으로 인터페이스나 dispinterface에 속성이나 메소드를 추가합니다.

1 인터페이스를 선택하고 툴바에서 속성 또는 메소드 아이콘을 선택합니다. 속성을 추가할 경우 속성 아이콘을 직접 클릭하여 읽기/쓰기 속성을 만들거나(getter 및 setter 모두 사용) 아래쪽 화살표를 클릭하여 속성 타입의 메뉴를 표시합니다.

객체 리스트 창에 속성 액세스 메소드 멤버나 메소드 멤버가 추가되고, 이름을 추가하라는 메시지가 표시됩니다.

2 멤버 이름을 입력합니다.

새 멤버는 Attributes, Parameters 및 Flags 페이지에 멤버에 맞게 수정할 수 있는 디폴트 설정을 포함합니다. 예를 들면, Attributes 페이지에서 속성에 타입을 할당할 수 있습니다. 메소드를 추가하는 경우에는 Parameters 페이지에서 매개변수를 지정할 수도 있습니다.

다른 방법으로, IDL 구문을 사용해 Text 페이지에 직접 입력하여 속성과 메소드를 추가할 수도 있습니다. 예를 들면, 인터페이스의 Text 페이지에 다음과 같은 속성 선언을 입력할 수 있습니다.

```
[
  uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
  version(1.0),
  dual,
  oleautomation
]
interface Interface1: IDispatch
{ // Add everything between the curly braces
[propget, id(0x00000002)]
  HRESULT _stdcall AutoSelect([out, retval] long Value );
  [propget, id(0x00000003)]
  HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
  [propput, id(0x00000003)]
  HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

인터페이스 텍스트 페이지를 사용하여 인터페이스에 멤버를 추가하면 객체 리스트 창에 멤버가 각각의 항목으로 나타나며 각 멤버마다 Attributes, Flags 및 Parameters 페이지를 가집니다. 객체 리스트 창에서 속성이나 메소드를 선택하고, 이러한 페이지를 사용하거나 Text 페이지에서 직접 편집하여 새 속성이나 메소드를 각각 수정할 수 있습니다.

인터페이스가 마법사에 의해 생성된 CoClass에 연결될 경우 툴바의 Refresh 버튼을 클릭하여 Type Library Editor를 통해 구현 파일에 변경 내용을 적용할 수 있습니다. Type Library Editor가 구현 클래스에 새 메소드를 추가하여 새 멤버를 반영합니다. 그런 다음 구현 유닛의 소스 코드에서 새 메소드를 찾아 바디를 입력하여 구현을 완성할 수 있습니다.

타입 라이브러리에 CoClass 추가

프로젝트에 CoClass를 추가하는 가장 쉬운 방법은 IDE의 메인 메뉴에서 File|New|Other를 선택하고, New Items 다이얼로그 박스의 ActiveX나 Multitier 페이지에서 적절한 마법사를 선택하는 것입니다. 이 방법의 장점은 타입 라이브러리에 CoClass와 해당 인터페이스를 추가하는 것 외에도, 마법사가 구현 유닛을 추가하고, 프로젝트 파일을 업데이트하여 새 구현 유닛을 포함하고, 프로젝트 파일에서 객체 맵에 새 CoClass를 추가하는 것입니다.

그러나 마법사를 사용하지 않는 경우 툴바에서 CoClass 아이콘을 클릭한 다음 해당 어트리뷰트(attribute)를 지정하여 CoClass를 만들 수 있습니다. Attributes 페이지에서 새 CoClass에 이름을 부여할 수 있으며, Flags 페이지를 사용하여 CoClass가 애플리케이션 객체인지, ActiveX 컨트롤을 나타내는지 등의 정보를 표시할 수 있습니다.

참고 마법사 대신 툴바를 사용하여 타입 라이브러리에 CoClass를 추가할 때는 직접 CoClass에 대한 구현을 생성하여 CoClass 인터페이스 하나에서 요소를 변경할 때마다 수동으로 이 구현을 업데이트해야 합니다. CoClass 구현을 수동으로 추가할 때는 프로젝트 파일의 객체 맵에 CoClass를 추가해야 합니다.

CoClass에 직접 멤버를 추가할 수 없습니다. 대신 CoClass에 인터페이스를 추가할 때 암시적으로 멤버를 추가합니다.

CoClass에 인터페이스 추가

CoClass는 클라이언트에 제공하는 인터페이스에 의해 정의됩니다. 개발자는 CoClass의 구현 클래스에 원하는 수만큼 속성과 메소드를 추가할 수 있지만, 클라이언트는 CoClass에 연결된 인터페이스가 노출하는 속성과 메소드만 볼 수 있습니다.

인터페이스를 CoClass에 연결하려면 클래스에 대한 Implements 페이지에서 마우스 오른쪽 버튼을 클릭하고 Insert Interface를 선택하여 선택할 수 있는 인터페이스 리스트를 표시합니다. 리스트에는 현재 타입 라이브러리에 정의된 인터페이스 및 현재 타입 라이브러리가 참조하는 타입 라이브러리에 정의된 인터페이스가 들어 있습니다. 클래스가 구현할 인터페이스를 선택합니다. GUID 및 기타 어트리뷰트(attribute)가 있는 페이지에 인터페이스가 추가됩니다.

마법사를 사용하여 CoClass를 생성한 경우 Type Library Editor가 구현 클래스를 자동으로 업데이트하여 이런 방법으로 추가하는 인터페이스의 메소드(속성 액세스 메소드 포함)에 대한 스켈레톤 메소드를 포함합니다.

타입 라이브러리에 열거 추가

다음과 같은 방법으로 타입 라이브러리에 열거를 추가합니다.

1 툴바에서 열거 아이콘을 클릭합니다.

객체 리스트 창에 열거 타입이 추가되고, 이름을 추가하라는 메시지가 표시됩니다.

2 열거 이름을 입력합니다.

새 열거는 비어 있으며, Attributes 페이지에 수정할 수 있는 디폴트 어트리뷰트가 포함되어 있습니다.

New Const 버튼을 클릭하여 열거에 값을 추가합니다. 그런 다음 각 열거 값을 선택하고 속성 페이지를 사용하여 열거 값에 이름을 할당합니다(값을 할당할 수도 있음).

열거를 추가한 후에는 타입 라이브러리 또는 해당 사용 페이지에서 이 타입 라이브러리를 참조하는 다른 타입 라이브러리가 새 타입을 사용할 수 있습니다. 예를 들면, 열거를 속성이나 매개변수에 대한 타입으로 사용할 수 있습니다.

타입 라이브러리에 알리아스 추가

다음과 같은 방법으로 타입 라이브러리에 알리아스를 추가합니다.

1 툴바에서 알리아스 아이콘을 클릭합니다.

객체 리스트 창에 알리아스 타입이 추가되고 이름을 추가하라는 메시지가 표시됩니다.

2 알리아스 이름을 입력합니다.

디폴트로, 새 알리아스는 long 타입을 나타냅니다. Attributes 페이지를 사용하여 이 타입을 알리아스가 나타내게 할 타입으로 변경합니다.

알리아스를 추가한 후에는 타입 라이브러리 또는 해당 사용 페이지에서 이 타입 라이브러리를 참조하는 다른 타입 라이브러리가 새 타입을 사용할 수 있습니다. 예를 들면, 알리아스를 속성이나 매개변수에 대한 타입으로 사용할 수 있습니다.

타입 라이브러리에 레코드 또는 합집합 추가

다음과 같은 방법으로 타입 라이브러리에 레코드나 합집합을 추가합니다.

1 툴바에서 레코드 아이콘이나 합집합 아이콘을 클릭합니다.

객체 리스트 창에 선택한 타입 요소가 추가되고 이름을 추가하라는 메시지가 표시됩니다.

2 레코드나 합집합 이름을 입력합니다.

이때 새 레코드나 합집합에 필드가 포함되지 않습니다.

3 객체 리스트 창에서 레코드나 합집합을 선택하고 툴바에서 필드 아이콘을 클릭합니다. Attributes 페이지를 사용하여 필드 이름과 타입을 지정합니다.

4 필요한 수의 필드에 대해 단계 3을 반복합니다.

레코드나 합집합을 정의한 후에는 타입 라이브러리 또는 해당 사용 페이지에서 이 타입 라이브러리를 참조하는 다른 타입 라이브러리가 새 타입을 사용할 수 있습니다. 예를 들면, 레코드나 합집합을 속성이나 매개변수에 대한 타입으로 사용할 수 있습니다.

타입 라이브러리에 모듈 추가

다음과 같은 방법으로 타입 라이브러리에 모듈을 추가합니다.

1 툴바에서 모듈 아이콘을 클릭합니다.

객체 리스트 창에 선택한 모듈이 추가되고 이름을 추가하라는 메시지가 표시됩니다.

2 모듈 이름을 입력합니다.

3 Attributes 페이지에서 Module이 해당 엔트리 포인트를 나타내는 DLL의 이름을 지정합니다.

4 툴바의 Method 아이콘을 클릭한 다음 속성 페이지를 사용해 메소드를 설명함으로써 단계 3에서 지정한 DLL에서 메소드를 추가합니다.

5 툴바에서 Const 아이콘을 클릭하여 모듈이 정의하게 할 상수를 추가합니다. 각 상수의 이름, 타입 및 값을 지정합니다.

타입 라이브러리 정보 저장 및 등록

타입 라이브러리를 수정한 다음 타입 라이브러리 정보를 저장 및 등록할 수 있습니다.

타입 라이브러리를 저장하면 다음 항목이 자동으로 업데이트됩니다.

- 바이너리 타입 라이브러리 파일(.tlb 확장자)
- 타입 라이브러리의 내용을 나타내는 *Project*_TLB 유닛
- 마법사가 생성한 CoClass에 대한 구현 코드

참고 타입 라이브러리는 각각의 바이너리 파일(.TLB)로 저장되지만 서버(.EXE, DLL 또는 .OCX)로 연결되기도 합니다.

Type Library Editor에는 타입 라이브러리 정보를 저장할 수 있는 옵션이 있습니다. 타입 라이브러리를 구현하는 과정에서 어떤 단계에 와 있느냐에 따라 선택하는 옵션이 달라집니다.

- .TLB와 *Project*_TLB 유닛 모두를 디스크에 저장하려면 Save를 선택하십시오.

- 타입 라이브러리 유닛을 메모리에서만 업데이트하려면 Refresh를 선택하십시오.
- 시스템의 Windows 레지스트리에서 타입 라이브러리에 대한 항목을 추가하려면 Register를 선택하십시오. 이 작업은 .TLB와 연결된 서버가 자체적으로 등록될 때 자동으로 수행됩니다.
- IDL 구문에 타입 및 인터페이스 정의를 포함하는 IDL 파일을 저장하려면 Export를 선택하십시오.

위의 모든 메소드는 구문 검사를 수행합니다. 타입 라이브러리를 새로 고치거나, 등록하거나, 저장할 저장하는 경우 마법사를 사용하여 만든 CoClass의 구현 유닛을 C++Builder에서 자동으로 업데이트합니다.

타입 라이브러리 저장

타입 라이브러리 저장

- 구문 및 유효성 검사를 수행합니다.
- 정보를 .TLB 파일에 저장합니다.
- 정보를 *Project*_TLB 유닛에 저장합니다.
- 마법사를 사용하여 생성된 CoClass에 타입 라이브러리가 연결된 경우 IDE의 모듈 관리자에게 구현을 업데이트하도록 통지합니다.

타입 라이브러리를 저장하려면 C++Builder 메인 메뉴에서 File|Save를 선택하십시오.

타입 라이브러리 새로 고침

타입 라이브러리 새로 고침

- 구문 검사를 수행합니다.
- 메모리에서만 C++Builder 타입 라이브러리 유닛을 다시 생성합니다. 디스크에 파일을 저장하지 않습니다.
- 마법사를 사용하여 생성된 CoClass에 타입 라이브러리가 연결된 경우 IDE의 모듈 관리자에게 구현을 업데이트하도록 통지합니다.

타입 라이브러리를 새로 고치려면 Type Library Editor 툴바에서 Refresh 아이콘을 클릭하십시오.

참고 타입 라이브러리에 이름이 변경된 항목이 있을 경우 구현을 새로 고치면 중복 항목이 만들어질 수도 있습니다. 이 경우에 코드를 올바른 엔트리로 이동하고 중복된 항목을 삭제해야 합니다. 마찬가지로 타입 라이브러리에서 항목을 삭제할 경우 구현을 새로 고쳐도 CoClass에서 해당 항목이 제거되는 것은 아니며 클라이언트에게 표시되지 않게 할 뿐입니다. 이러한 항목이 필요 없을 경우 구현 유닛에서 수동으로 삭제해야 합니다.

타입 라이브러리 등록

일반적으로 COM 서버 애플리케이션을 등록할 때 타입 라이브러리가 자동으로 등록되므로 타입 라이브러리를 명시적으로 등록할 필요는 없습니다(41-16페이지의 "COM 객체 등록" 참조). 그러나 Type Library 마법사를 사용하여 타입 라이브러리를 만들 경우 타입 라이브러리가 서버 객체에 연결되지 않습니다. 이런 경우 툴바를 사용하여 타입 라이브러리를 직접 등록할 수 있습니다.

타입 라이브러리 등록

- 구문 검사를 수행합니다.
- 타입 라이브러리에 대한 Windows 레지스트리에 항목을 추가합니다.

타입 라이브러리를 등록하려면 Type Library Editor 툴바에서 Register 아이콘을 선택하십시오.

IDL 파일 엑스포트

타입 라이브러리를 엑스포트하려면 다음을 수행합니다.

- 구문 체크를 수행합니다.
- 타입 정보 선언이 들어 있는 IDL 파일을 만듭니다. 이 파일은 Microsoft IDL에 있는 타입 정보에 대해 설명합니다.

타입 라이브러리를 엑스포트하려면 Type Library Editor 툴바에서 Export 아이콘을 선택하십시오.

타입 라이브러리 배포

디폴트로, ActiveX 또는 Automation 서버 프로젝트의 일부로 만들어진 타입 라이브러리가 있을 경우 타입 라이브러리는 .DLL, .OCX 또는 EXE에 자동으로 리소스로 연결됩니다.

그러나 C++Builder에서 타입 라이브러리를 유지 보수하므로 필요에 따라 애플리케이션을 타입 라이브러리와 함께 각각의 .TLB로 배포할 수 있습니다.

기존에는 Automation 애플리케이션에 대한 타입 라이브러리가 .TLB 확장자를 사용하는 각각의 파일로 저장되었습니다. 이제 대부분의 Automation 애플리케이션이 타입 라이브러리를 .OCX 또는 .EXE 파일로 직접 컴파일합니다. 운영 체제는 타입 라이브러리가 실행 파일(.DLL, .OCX 또는 .EXE)의 첫 번째 리소스가 될 것으로 예상합니다.

기본 프로젝트 타입 라이브러리가 아닌 타입 라이브러리를 애플리케이션 개발자가 사용할 수 있게 하면 타입 라이브러리는 다음 형식 중 하나일 수 있습니다.

- 리소스. 이 리소스에는 타입 TYPELIB와 정수 ID가 있어야 합니다. 리소스 컴파일러를 사용하여 타입 라이브러리를 생성하도록 선택할 경우 타입 라이브러리가 리소스 파일(.RC)에 다음과 같이 선언되어야 합니다.

  ```
  1 typelib mylib1.tlb
  2 typelib mylib2.tlb
  ```

  ActiveX 라이브러리에 여러 개의 타입 라이브러리 리소스가 있을 수 있습니다. 애플리케이션 개발자는 리소스 컴파일러를 사용하여 자신의 ActiveX 라이브러리에 .TLB 파일을 추가합니다.
- 독립 실행형 바이너리 파일 . Type Library Editor 에서 출력되는 .TLB 파일은 바이너리 파일입니다.

장

40

COM 클라이언트 생성

COM 클라이언트는 다른 애플리케이션 또는 라이브러리에 의해 구현된 COM 객체를 사용하는 애플리케이션입니다. 가장 일반적인 타입은 Automation 서버(Automation 컨트롤러)를 컨트롤하는 애플리케이션과 ActiveX 컨트롤(ActiveX 컨테이너)을 호스트하는 애플리케이션입니다.

이 두 타입의 COM 클라이언트는 매우 달라 보입니다. 일반적인 Automation 컨트롤러는 외부 서버 EXE를 시작하고 그 서버에서 작업을 수행하도록 하는 명령을 실행합니다. Automation 서버는 넌비주얼(nonvisual)이며 out-of-process 서버입니다. 반면에 전형적인 ActiveX 클라이언트는 보이는 컨트롤을 호스트하며 컴포넌트 팔레트의 컨트롤을 사용하는 것과 비슷한 방법으로 사용합니다. ActiveX 서버는 항상 in-process 서버입니다.

그러나 이 두 가지 타입의 COM 클라이언트를 작성하는 방법은 놀랄 만큼 비슷합니다. 클라이언트 애플리케이션은 서버 객체를 위한 인터페이스를 확보하여 속성과 메소드를 사용합니다. C++Builder를 사용하면 클라이언트상의 컴포넌트에 있는 서버 CoClass를 랩핑할 수 있기 때문에 이 작업을 더 쉽게 할 수 있습니다. 그러한 컴포넌트 랩퍼 예제는 컴포넌트 팔레트의 두 페이지에 표시됩니다. 예제 ActiveX 랩퍼(ActiveX wrapper)는 ActiveX 페이지에 표시되고 Automation 객체는 Servers 페이지에 표시됩니다.

컴포넌트 팔레트에 있는 컴포넌트의 속성과 메소드를 이해해야 애플리케이션에서 사용할 수 있는 것과 마찬가지로 COM 클라이언트를 작성할 때는 서버가 클라이언트에 노출하는 인터페이스를 이해해야 합니다. 이 인터페이스는 서버 애플리케이션에 의해 결정되며 일반적으로 타입 라이브러리에 published로 선언됩니다. 특정 서버 애플리케이션의 published로 선언된 인터페이스에 대한 자세한 내용은 해당 애플리케이션 설명서를 참조해야 합니다.

컴포넌트 랩퍼에 서버 객체를 랩핑하도록 선택하지 않고 컴포넌트 팔레트에 설치하는 경우에도 애플리케이션에서 인터페이스 정의를 사용할 수 있도록 해야 합니다. 이를 위해 서버의 타입 정보를 임포트할 수 있습니다.

참고 COM API를 사용하여 타입 정보를 직접 쿼리할 수도 있지만, C++Builder에서 이를 위한 특별한 지원을 제공하지는 않습니다.

OLE와 같은 이전의 일부 COM 기술은 타입 라이브러리에 타입 정보를 제공하지 않고 일련의 사전 정의된 인터페이스에 의존합니다. 이 내용은 40-16페이지의 "타입 라이브러리가 없는 서버를 위한 클라이언트 생성"에 설명되어 있습니다.

타입 라이브러리 정보 임포트

COM 서버에 대한 정보를 클라이언트 애플리케이션에서 사용할 수 있도록 하려면 서버의 타입 라이브러리에 저장되어 있는 서버 정보를 임포트해야 합니다. 그러면 애플리케이션이 생성된 클래스를 사용하여 서버 객체를 제어할 수 있습니다.

타입 라이브러리 정보를 임포트하는 방법에는 다음 두 가지가 있습니다.

- Import Type Library 다이얼로그 박스를 사용하여 서버 타입, 객체 및 인터페이스에 대한 모든 정보를 임포트할 수 있습니다. 이것은 모든 타입 라이브러리에서 정보를 임포트할 수 있고 타입 라이브러리에서 Hidden, Restricted 또는 PreDeclID 플래그 표시가 되어 있지 않은, 생성 가능한 모든 CoClass에 대해 컴포넌트 랩퍼를 선택적으로 생성할 수 있기 때문에 가장 일반적인 방법입니다.
- ActiveX 컨트롤의 타입 라이브러리에서 임포트하는 경우에는 Import ActiveX 다이얼로그 박스를 사용할 수 있습니다. 이 방법도 동일한 타입 정보를 임포트하지만 ActiveX 컨트롤을 나타내는 CoClass에 대한 컴포넌트 랩퍼만 생성합니다.
- IDE 내에서는 사용할 수 없는 추가 구성 옵션을 제공하는 명령줄 유틸리티인 tlibimp.exe를 사용할 수 있습니다.
- 마법사를 사용하여 생성된 타입 라이브러리는 Import Type Library 메뉴 항목과 동일한 메커니즘을 사용하여 자동으로 임포트됩니다.

어떤 방법으로 타입 라이브러리 정보를 임포트하든 관계 없이 결과물인 다이얼로그 박스에서는 *TypeLibName*_TLB라는 이름의 유닛을 만듭니다. 이 때 *TypeLibName*은 타입 라이브러리의 이름입니다. 이 파일에는 타입 라이브러리에 정의된 클래스, 타입 및 인터페이스에 대한 선언이 들어 있습니다. 이 파일을 프로젝트에 포함시키면 애플리케이션에서 정의를 사용할 수 있으므로 객체를 만들 수 있으며 인터페이스를 호출할 수 있습니다. 이 파일은 때때로 IDE에 의해 다시 생성될 수 있으므로 파일을 수동으로 변경하지 않는 것이 좋습니다.

타입 정의를 *TypeLibName*_TLB 유닛에 추가하는 것 외에 타입 라이브러리에 정의된 CoClass에 대한 VCL 클래스 랩퍼를 만들고 *TypeLibName*_OCX 라는 이름의 독립적인 유닛에 넣을 수 있습니다. Import Type Library 다이얼로그 박스를 사용하는 경우에는 이 랩퍼가 옵션으로 제공되고 Import ActiveX 다이얼로그 박스를 사용하는 경우에는 컨트롤을 나타내는 모든 CoClass에 대해 생성됩니다.

참고 컴포넌트 랩퍼를 생성하는 경우, Import 다이얼로그 박스는 *TypeLibName*_OCX 유닛을 생성하지만 프로젝트에 이를 추가하지 않습니다. *TypeLibName*_TLB 유닛만 프로젝트에 추가합니다. Project|Add to Project를 선택하여 *TypeLibName*_OCX 유닛을 명시적으로 프로젝트에 추가할 수 있습니다.

생성된 클래스 랩퍼는 애플리케이션에 대한 CoClass를 나타내며 인터페이스의 속성과 메소드를 노출합니다. CoClass가 이벤트 생성을 위한 *IConnectionPointContainer* 및 *IConnectionPoint* 인터페이스를 지원하는 경우 VCL 클래스 랩퍼는 이벤트 싱크를 만들어 다른 컴포넌트와 마찬가지로 이벤트를 위한 이벤트 핸들러를 할당할 수 있습니다. 다이얼로그 박스에서 생성된 VCL 클래스를 컴포넌트 팔레트에 설치하도록 지정한 경우에는 Object Inspector를 사용하여 속성 값과 이벤트 핸들러를 지정할 수 있습니다.

참고 Import Type Library 다이얼로그 박스는 COM+ 이벤트 객체를 위한 클래스 랩퍼는 만들지 않습니다. COM+ 이벤트 객체에 의해 생성된 이벤트에 응답하는 클라이언트를 작성하려면 프로그램에서 이벤트 싱크를 만들어야 합니다. 이 과정은 40-15 페이지의 "COM+ 이벤트 처리"에 설명되어 있습니다.

타입 라이브러리를 임포트할 때 생성되는 코드에 대한 자세한 내용은 40-5페이지의 "타입 라이브러리 정보를 임포트할 때 생성되는 코드"를 참조하십시오.

Import Type Library 다이얼로그 박스 사용

다음과 같은 방법으로 타입 라이브러리를 임포트합니다.

1. Project | Import Type Library를 선택합니다.
2. 리스트에서 타입 라이브러리를 선택합니다.

   다이얼로그 박스는 이 시스템에 등록된 모든 라이브러리를 나열합니다. 타입 라이브러리가 리스트에 없는 경우에는 Add 버튼을 선택하고 타입 라이브러리 파일을 찾아 선택한 다음 OK를 선택합니다. 이렇게 하면 타입 라이브러리가 등록되어 사용할 수 있습니다. 그 다음에는 단계 2를 반복합니다. 타입 라이브러리는 독립 실행형 타입 라이브러리 파일(.tlb, .olb) 이거나 타입 라이브러리를 제공하는 서버(.dll, .ocx, .exe)입니다.
3. 타입 라이브러리에서 CoClass를 랩핑하는 VCL 컴포넌트를 생성하기 원하는 경우에는 Generate Component Wrapper를 선택합니다. 컴포넌트를 생성하지 않는 경우에는 *TypeLibName*_TLB 유닛에 있는 정의를 사용하여 CoClass를 사용할 수 있습니다. 그러나, 서버 객체를 만드는 호출을 자체적으로 작성하고 필요하면 이벤트 싱크를 설정해야 합니다.

   Import Type Library 다이얼로그 박스는 CanCreate 플래그가 설정되어 있고 Hidden, Restricted, 또는 PreDeclID 플래그는 설정되어 있지 않은 CoClass만 임포트합니다. 이러한 플래그는 명령줄 유틸리티인 tlibimp.exe을 사용하여 오버라이드될 수 있습니다.
4. 생성된 컴포넌트 랩퍼를 컴포넌트 팔레트에 설치하지 않으려면 Create Unit을 선택합니다. 이렇게 하면 *TypeLibName*_TLB 유닛이 생성되고 단계 3에서 Generate Component Wrapper를 선택한 경우에는 *TypeLibName*_OCX 유닛이 생성됩니다. Import Type Library 다이얼로그 박스가 종료됩니다.
5. 생성된 컴포넌트 랩퍼를 컴포넌트 팔레트에 설치하고자 하는 경우에는 이 컴포넌트가 상주할 Palette 페이지를 선택한 다음 Install을 선택합니다. 이렇게 하면 Create Unit 버튼을 선택한 것과 마찬가지로 *TypeLibName*_TLB 및 *TypeLibName*_OCX 유닛이 생성되고 Install component 다이얼로그 박스가 표시되어 컴포넌트가 상주해야 하는 기존 패키지 또는 새 패키지를 지정할 수 있습니다. 타입 라이브러리를 위한 컴포넌트를 만들 수 없는 경우에는 이 버튼이 회색으로 표시됩니다.

Import Type Library 다이얼로그 박스를 종료하면 새로운 *TypeLibName*_TLB 및 *TypeLibName*_OCX 유닛이 Unit dir 이름 컨트롤에 의해 지정된 디렉토리에 나타납니다. *TypeLibName*_TLB 유닛은 타입 라이브러리에 정의된 요소를 위한 선언을 포함하며, *TypeLibName*_OCX 유닛은 Generate Component Wrapper를 선택한 경우 생성된 컴포넌트 랩퍼를 포함합니다.

또한 생성된 컴포넌트 랩퍼를 설치한 경우에는 타입 라이브러리에서 설명한 서버 객체가 컴포넌트 팔레트에 상주하게 됩니다. Object Inspector를 사용하여 속성을 설정하거나 서버를 위한 이벤트 핸들러를 작성할 수 있습니다. 컴포넌트를 폼 또는 데이터 모듈에 추가하면 디자인 타임에 마우스 오른쪽 버튼을 클릭하여 지원되는 경우 속성 페이지를 볼 수 있습니다.

참고 컴포넌트 팔레트의 Servers 페이지에는 이러한 방법으로 임포트된 많은 Automation 서버 예제들이 포함되어 있습니다.

Import ActiveX 다이얼로그 박스 사용

다음과 같은 방법으로 ActiveX 컨트롤을 임포트합니다.

1. Component | Import ActiveX Control을 선택합니다.
2. 리스트에서 타입 라이브러리를 선택합니다.

   다이얼로그 박스에는 ActiveX 컨트롤을 정의하는 모든 등록된 라이브러리를 나열합니다. 이 리스트는 Import Type Library 다이얼로그 박스에 나열된 라이브러리의 부분 집합입니다. 타입 라이브러리가 리스트에 없으면 Add 버튼을 선택하고 타입 라이브러리 파일을 찾아 선택한 다음 OK를 선택합니다. 이렇게 하면 타입 라이브러리가 등록되어 사용할 수 있습니다. 그 다음에는 단계 2를 반복합니다. 타입 라이브러리는 독립 실행형 타입 라이브러리 파일(.tlb, .olb)이거나 ActiveX 서버(.dll, .ocx)입니다.
3. 컴포넌트 팔레트에 ActiveX 컨트롤을 설치하지 않으려는 경우에는 Create Unit을 선택합니다. 이렇게 하면 *TypeLibName*_TLB 유닛과 *TypeLibName*_OCX 유닛이 생성되고, Import ActiveX 다이얼로그 박스가 종료됩니다.
4. 컴포넌트 팔레트에 ActiveX 컨트롤을 설치하려는 경우에는 이 컴포넌트가 상주할 Palette 페이지를 선택한 다음 Install을 선택합니다. 이렇게 하면 Create Unit 버튼을 선택하는 것과 마찬가지로 *TypeLibName*_TLB 및 *TypeLibName*_OCX 유닛이 생성되고 Install component 다이얼로그 박스가 표시되어 컴포넌트가 상주해야 하는 기존 패키지 또는 새 패키지를 지정할 수 있습니다.

Import ActiveX 다이얼로그 박스를 종료하면 새로운 *TypeLibName*_TLB 및 *TypeLibName*_OCX 유닛이 Unit dir 이름 컨트롤에 의해 지정된 디렉토리에 나타납니다. *TypeLibName*_TLB 유닛은 타입 라이브러리에 정의된 요소를 위한 선언을 포함하며, *TypeLibName*_OCX 유닛은 ActiveX 컨트롤을 위해 생성된 컴포넌트 랩퍼를 포함합니다.

참고 옵션 사항인 Import Type Library 다이얼로그 박스와 달리 Import ActiveX 다이얼로그 박스는 항상 컴포넌트 랩퍼를 생성합니다. 이는 비주얼 컨트롤인 ActiveX 컨트롤이 VCL 폼과 함께 들어갈 수 있도록 컴포넌트 랩퍼에 대한 추가 지원이 필요하기 때문입니다.

생성된 컴포넌트 랩퍼를 설치한 경우 ActiveX 컨트롤은 컴포넌트 팔레트에 상주합니다. Object Inspector를 사용하여 속성을 설정하거나 이 컨트롤을 위한 이벤트 핸들러를 작성할 수 있습니다 . 컨트롤을 폼 또는 데이터 모듈에 추가하면 디자인 타임에 마우스 오른쪽 버튼을 클릭하여 지원되는 경우 속성 페이지를 볼 수 있습니다.

참고 컴포넌트 팔레트의 ActiveX 페이지에는 이러한 방법으로 임포트된 많은 ActiveX 컨트롤 예제들이 포함되어 있습니다.

타입 라이브러리 정보를 임포트할 때 생성되는 코드

타입 라이브러리를 임포트하면 생성된 *TypeLibName*_TLB 유닛을 볼 수 있습니다. 그 유닛의 소스 파일은 타입 라이브러리와 인터페이스 및 CoClass의 GUIDS에 상징적인 이름을 부여하는 상수를 정의합니다. 이러한 상수의 이름은 다음과 같이 생성됩니다.

- 타입 라이브러리를 위한 GUID는 LIBID_*TypeLibName*의 형태를 가집니다. 여기서 *TypeLibName*은 타입 라이브러리의 이름입니다.
- 인터페이스를 위한 GUID는 IID_*InterfaceName*의 형태를 가집니다. 여기서 *InterfaceName*은 인터페이스의 이름입니다.
- dispinterface를 위한 GUID는 DIID_*InterfaceName*의 형태를 가집니다. 여기서 *InterfaceName*은 dispinterface의 이름입니다.
- CoClass를 위한 GUID는 CLSID_*ClassName*의 형태를 가집니다. 여기서 *ClassName*은 CoClass의 이름입니다.

소스 파일을 마우스 오른쪽 버튼으로 클릭하고 Open Source/Header file을 선택하면 다음 정의를 찾을 것입니다.

- 타입 라이브러리에 있는 CoClass를 위한 선언. 이 선언은 각 CoClass를 디폴트 인터페이스와 매핑합니다. 그 외에 *TComInterface* 템플릿을 사용하여 각 CoClass를 위한 랩퍼가 선언됩니다. 이 랩퍼의 이름은 CoClassNamePtr의 형태를 가집니다.
- 타입 라이브러리에 있는 인터페이스와 dispinterface를 위한 선언
- 인터페이스 및 dispinterface를 위한 클래스 랩퍼 선언. 인터페이스의 경우 클래스 랩퍼는 *TComInterface* 템플릿을 사용하며 TCom*InterfaceName* 형태의 이름을 가집니다. dispinterface의 경우 클래스 랩퍼는 *TAutoDriver* 템플릿을 사용하며 *InterfaceName*Disp 형태의 이름을 가집니다.
- 디폴트 인터페이스가 VTable 연결을 지원하는 각 CoClass의 작성자 클래스를 위한 선언. 작성자 클래스는 *Create*와 *CreateRemote*의 두 가지 정적(static) 메소드를 가집니다. *Create* 메소드는 로컬에서, (*CreateRemote*는 원격으로 CoClass를 인스턴스화하는 데 사용될 수 있습니다. 이 두 메소드는 *TComInterface* 템플릿을 사용하여 *TypeLibName*_TLB.h에 정의된 CoClass의 디폴트 인터페이스를 위한 클래스 랩퍼를 반환합니다.
- 모든 이벤트 인터페이스를 위한 대리자 클래스 랩퍼. 이 클래스 랩퍼는 TEvents_*CoClassName* 형태의 이름을 가집니다. 여기서 *CoClassName*은 이벤트를 생성하는 CoClass의 이름입니다. 대리자 클래스는 모든 클라이언트 이벤트 싱크 리스트를 가지고 있으며 서버 객체에서 이벤트가 발생할 때 모든 해당 이벤트 싱크에서 해당되는 메소드를 호출합니다.

이러한 선언들은 CoClass 인스턴스를 생성하고 인터페이스에 액세스하는 데 필요한 것을 제공합니다. 생성된 *TypeLibName*_TLB.h 파일을 CoClass에 연결하고 인터페이스를 호출하려는 유닛에 포함시키기만 하면 됩니다.

경고 생성된 랩퍼 클래스는 정적 객체인 *TInitOleT*를 사용하여 COM 초기화를 처리합니다. 이는 서버에 연결되는 첫 번째 스레드에만 COM이 초기화되므로 여러 스레드에서 객체를 만드는 경우 문제가 될 수 있습니다. COM 서버에 연결되는 모든 다른 스레드에서 *TInitOleT* 또는 *TInitOle*의 각 인스턴스를 명시적으로 만들어야 합니다.

참고 Type Library Editor 또는 명령줄 유틸리티인 TLIBIMP를 사용할 때는 *TypeLibName*_TLB 유닛도 생성됩니다

ActiveX 컨트롤을 사용하기 원하는 경우에는 위에서 설명한 선언 외에 생성된 VCL 랩퍼도 필요합니다. VCL 랩퍼는 컨트롤을 위한 윈도우 관리 문제를 처리합니다. 또한 Import Type Library 다이얼로그 박스에서 다른 CoClass를 위한 VCL 랩퍼를 생성했을 수도 있습니다. 이러한 VCL 랩퍼는 서버 객체를 만들고 메소드를 호출하는 작업을 단순화합니다. 클라이언트 애플리케이션이 이벤트에 응답하도록 하려는 경우 특히 이 랩퍼를 권장합니다.

생성된 VCL 랩퍼를 위한 선언은 *TypeLibName*_OCX 유닛에 포함됩니다. ActiveX 컨트롤을 위한 컴포넌트 랩퍼는 *TOleControl*의 자손입니다. Automation 객체를 위한 컴포넌트 랩퍼는 *TOleServer*의 자손입니다. 생성된 컴포넌트 랩퍼는 CoClass의 인터페이스에 의해 노출되는 속성, 이벤트 및 메소드를 추가합니다. 이 컴포넌트는 다른 VCL 컴포넌트처럼 사용할 수 있습니다.

경고 생성된 *TypeLibName*_TLB 또는 *TypeLibName*_OCX 유닛은 편집할 수 없습니다. 이 두 유닛은 타입 라이브러리를 새로 고칠 때마다 다시 생성되므로 변경 사항이 있으면 덮어 씁니다.

참고 생성된 코드에 대한 최신 정보는 자동으로 생성된 *TypeLibName*_TLB 유닛 및 utilcls.h에 있는 주석을 참조하십시오.

임포트된 객체 제어

타입 라이브러리 정보를 임포트한 다음에는 임포트된 객체를 사용하여 프로그래밍을 시작할 수 있습니다. 진행 방법은 객체 및 컴포넌트 랩퍼의 생성 여부에 따라 달라집니다.

컴포넌트 랩퍼 사용

서버 객체를 위한 컴포넌트 랩퍼를 생성한 경우에는 COM 클라이언트 애플리케이션을 작성하는 것이 VCL 컴포넌트를 포함한 다른 애플리케이션을 작성하는 것과 크게 다르지 않습니다. 서버 객체의 속성, 메소드 및 이벤트는 VCL 컴포넌트에 이미 캡슐화되어 있습니다. 이벤트 핸들러를 지정하고 속성 값을 설정한 다음 메소드를 호출하면 됩니다.

서버 객체의 속성, 메소드 및 이벤트를 사용하려면 서버의 설명서를 참조하십시오. 컴포넌트 랩퍼는 가능한 곳에서 이중 인터페이스를 자동으로 제공합니다. C++Builder는 타입 라이브러리에 있는 정보에서 VTable 레이아웃을 결정합니다.

그 외에 새 컴포넌트는 특정한 중요 속성과 메소드를 기본 클래스로부터 상속합니다.

ActiveX 랩퍼(ActiveX wrapper)

ActiveX 컨트롤을 호스팅할 때는 컴포넌트 랩퍼가 컨트롤의 윈도우를 VCL 프레임워크와 통합하기 때문에 항상 컴포넌트 랩퍼를 사용해야 합니다.

ActiveX 컨트롤이 *TOleControl*에서 상속한 속성과 메소드를 사용하면 기초가 되는 인터페이스를 액세스하거나 컨트롤에 대한 정보를 얻을 수 있습니다. 그러나 대부분의 애플리케이션에서는 이들을 사용할 필요가 없으며, 임포트된 컨트롤을 다른 VCL 컨트롤과 같은 방법으로 사용합니다.

일반적으로 ActiveX 컨트롤은 속성을 설정할 수 있는 속성 페이지를 제공합니다. 속성 페이지는 일부 컴포넌트의 경우 폼 디자이너에서 더블 클릭하면 나타나는 컴포넌트 에디터와 비슷합니다. ActiveX 컨트롤 속성 페이지를 표시하려면 Properties를 마우스 오른쪽 버튼으로 클릭하여 선택합니다.

임포트된 대부분의 ActiveX 컨트롤의 사용 방법은 서버 애플리케이션에 의해 결정됩니다. 그러나, ActiveX 컨트롤은 데이터베이스 필드의 데이터를 나타낼 때 일련의 표준 통지를 사용합니다. 그러한 ActiveX 컨트롤을 호스팅하는 방법에 대한 내용은 40-8페이지의 "데이터 인식 ActiveX 컨트롤 사용"을 참조하십시오.

Automation 객체 랩퍼

Automation 객체를 위한 랩퍼를 사용하여 서버 객체에 대한 연결 방법을 제어할 수 있습니다.

- *ConnectKind* 속성은 로컬 서버인지 원격 서버인지 여부와 이미 실행 중인 서버에 대한 연결 여부 또는 새 인스턴스 시작 여부를 나타냅니다. 원격 서버에 연결할 때는 *RemoteMachineName* 속성을 사용하여 시스템 이름을 지정해야 합니다.
- 일단 *ConnectKind*를 지정한 후 컴포넌트를 서버에 연결하는 방법은 다음 세 가지입니다.
  - 컴포넌트의 *Connect* 메소드를 호출하여 서버에 명시적으로 연결할 수 있습니다.
  - *AutoConnect* 속성을 **true**로 설정하여 애플리케이션이 시작될 때 컴포넌트에서 자동으로 연결하도록 할 수 있습니다.
  - 서버에 명시적으로 연결할 필요는 없습니다. 컴포넌트는 서버의 속성이나 메소드 중 하나를 사용할 때 자동으로 연결됩니다.

메소드 호출이나 속성에 액세스하는 것은 다른 컴포넌트를 사용하는 것과 마찬가지입니다.

```
TServerComponent1->DoSomething();
```

Object Inspector 를 사용하여 이벤트 핸들러를 작성할 수 있기 때문에 이벤트를 쉽게 처리할 수 있습니다. 그러나, 컴포넌트의 이벤트 핸들러는 타입 라이브러리에서 이벤트를 위해 정의된 이벤트 핸들러와 매개변수가 약간 다를 수 있습니다. 인터페이스 포인터인 매개변수는 인터페이스 포인터 그대로 표시되지 않고 대개 *TComInterface* 템플릿을 사용하여 랩핑됩니다. 결과 인터페이스 랩퍼는 *InterfaceName*Ptr 형태의 이름을 가집니다.

예를 들어, 다음 코드는 ExcelApplication 이벤트를 위한 이벤트 핸들러인 OnNewWorkBook을 보여 줍니다. 이 이벤트 핸들러는 다른 CoClass(ExcelWorkbook)의 인터페이스를 제공하는 매개변수를 가집니다. 그러나, 인터페이스는 ExcelWorkbook 인터페이스 포인터가 아니라 *ExcelWorkbookPtr* 객체로서 전달됩니다.

```
void _fastcall TForm1::XLappNewWorkbook(TObject *Sender, ExcelWorkbookPtr
Wb)
{
  ExcelWorkbook1->ConnectTo(Wb);
}
```

이 예제에서 이벤트 핸들러는 ExcelWorkbook 컴포넌트에 통합 문서(ExcelWorkbook1)를 할당합니다. 이 예제는 *ConnectTo* 메소드를 사용하여 컴포넌트 랩퍼를 기존 인터페이스에 연결하는 방법을 보여 줍니다. *ConnectTo* 메소드는 컴포넌트 랩퍼를 위해 생성된 코드에 추가됩니다.

애플리케이션 객체를 가지는 서버는 해당 객체에 Quit 메소드를 노출하여 클라이언트에서 연결을 끊을 수 있도록 합니다. Quit 메소드는 File 메뉴를 사용하여 애플리케이션을 종료하는 것과 같은 기능을 노출합니다. Quit 메소드를 호출하는 코드는 컴포넌트의 *Disconnect* 메소드에서 생성됩니다. 매개변수 없이 Quit 메소드를 호출할 수 있는 경우 컴포넌트 랩퍼도 *AutoQuit* 속성을 가집니다. *AutoQuit* 속성은 컴포넌트가 해제되면 컨트롤러가 Quit 메소드를 호출하도록 합니다. 다른 때 연결을 끊기 원하거나 Quit 메소드가 매개변수를 필요로 하는 경우에는 명시적으로 호출해야 합니다. Quit 메소드는 생성된 컴포넌트에 public 메소드로 나타납니다.

데이터 인식 ActiveX 컨트롤 사용

C++Builder 애플리케이션에서 데이터 인식 ActiveX 컨트롤을 사용할 때는 해당 컨트롤이 나타내는 데이터베이스와 컨트롤을 연결해야 합니다. 이렇게 하려면 데이터 인식 VCL 컨트롤을 위한 데이터 소스가 필요한 것과 마찬가지로 데이터 소스 컴포넌트가 필요합니다.

데이터 인식 ActiveX 컨트롤을 폼 디자이너에 놓은 후 원하는 데이터셋을 나타내는 데이터 소스에 *DataSource* 속성을 할당합니다. 데이터 소스를 지정하고 나면 Data Bindings Editor를 사용하여 컨트롤의 data-bound 속성을 데이터셋의 필드에 연결할 수 있습니다.

Data Bindings Editor를 표시하려면 데이터 인식 ActiveX 컨트롤을 마우스 오른쪽 버튼으로 클릭하여 옵션 리스트를 표시합니다. 기본적인 옵션 외에 추가 Data Bindings 항목이 표시됩니다. 이 항목을 선택하여 Data Bindings Editor를 표시하면 데이터셋의 필드 이름과 ActiveX 컨트롤의 연결할 수 있는 속성이 나열됩니다.

다음과 같은 방법으로 필드를 속성에 연결합니다.

1 ActiveX Data Bindings Editor 다이얼로그 박스에서 필드와 속성 이름을 선택합니다.

Field Name에는 데이터베이스 필드가 나열되고 Property Name에는 데이터베이스 필드에 연결될 수 있는 ActiveX 컨트롤 속성이 나열됩니다. 속성의 dispID는 Value(12) 처럼 괄호 안에 표시됩니다.

2 Bind와 OK를 클릭합니다.

참고 다이얼로그 박스에 속성이 표시되지 않으면 ActiveX 컨트롤에 데이터 인식 속성이 포함되어 있지 않은 것입니다. ActiveX 컨트롤의 속성을 위한 간단한 데이터 연결을 할 수 있도록 하려면 43-11페이지의 "타입 라이브러리를 사용하여 간단한 데이터 연결 사용"에 설명되어 있는 대로 타입 라이브러리를 사용합니다.

다음 예제에서는 C++Builder 컨테이너에서 데이터 인식 ActiveX 컨트롤을 사용하는 방법을 단계별로 보여 줍니다. 이 예제에서는 시스템에 Microsoft Office 97이 설치되어 있는 경우 사용할 수 있는 Microsoft Calendar Control을 사용합니다.

1 C++Builder 메인 메뉴에서 Component | Import ActiveX Control을 선택합니다.

2 Microsoft Calendar 컨트롤 8.0과 같은 데이터 인식 ActiveX 컨트롤을 선택하고 그 클래스 이름을 *TCalendarAXControl*로 변경한 다음 Install을 클릭합니다.

3 Install 다이얼로그 박스에서 OK를 클릭하여 Palette에서 컨트롤을 사용할 수 있도록 해주는 디폴트 사용자 패키지에 컨트롤을 추가합니다.

4 Close All과 File | New | Application을 선택하여 새 애플리케이션을 시작합니다.

5 ActiveX 탭에서 Palette에 방금 추가한 *TCalendarAXControl* 객체를 폼에 가져다 놓습니다.

6 Data Access 탭에서 *DataSource* 객체를, 그리고 BDE 탭에서 *Table* 객체를 폼에 가져다 놓습니다.

7 *DataSource* 객체를 선택하고 *DataSet* 속성을 *Table1*로 설정합니다.

8 *Table* 객체를 선택하고 다음 작업을 수행합니다.

- *DatabaseName* 속성을 BCDEMOS로 설정합니다.
- *TableName* 속성을 EMPLOYEE.DB로 설정합니다.
- *Active* 속성을 **true**로 설정합니다.

9 *TCalendarAXControl* 객체를 선택하고 *DataSource* 속성을 *DataSource1*로 설정합니다.

10 *TCalendarAXControl* 객체를 선택한 다음 마우스 오른쪽 버튼을 클릭하고 Data Bindings를 선택하여 ActiveX Control Data Bindings Editor를 호출합니다.

Field Name에는 현재 데이터베이스의 모든 필드가 나열됩니다. Property Name에는 데이터베이스 필드에 연결될 수 있는 ActiveX Control의 속성이 나열됩니다. 속성의 dispID는 괄호 안에 표시됩니다.

11 *HireDate* 필드와 *Value* 속성 이름을 선택하고 Bind와 OK를 선택합니다.

필드 이름과 속성은 이제 연결되었습니다.

12 Data Controls 탭에서 *DBGrid* 객체를 폼에 가져다 놓고 *DataSource* 속성을 *DataSource1*로 설정합니다.

13 Data Controls 탭에서 *DBNavigator* 객체를 폼에 가져다 놓고 *DataSource* 속성을 *DataSource1*로 설정합니다.

14 애플리케이션을 실행합니다.

15 다음과 같은 방법으로 애플리케이션을 테스트합니다.

DBGrid 객체에 *HireDate* 필드가 표시되면 Navigator 객체를 사용하여 데이터베이스를 검색합니다. 데이터베이스를 검색하여 이동하는 데 따라 ActiveX 컨트롤의 날짜가 달라집니다.

예제: Microsoft Word를 사용한 문서 인쇄

다음 단계는 Office 97 에서 Microsoft Word 8 을 사용하여 문서를 인쇄하는 Automation 컨트롤러를 만드는 방법을 보여 줍니다.

시작하기 전에 다음 작업을 수행합니다. 폼, 버튼 및 열기 다이얼로그 박스인 *TOpenDialog*로 구성된 새 프로젝트를 만듭니다. 이러한 컨트롤들은 Automation 컨트롤러를 구성합니다.

단계 1: 이 예제를 위해 C++Builder 준비

사용자의 편의를 위해 C++Builder는 Word, Excel 및 PowerPoint 등 많은 일반적인 서버를 컴포넌트 팔레트에 제공합니다. 여기서는 서버를 임포트하는 방법을 설명하기 위해 Word를 사용합니다. Word는 이미 컴포넌트 팔레트에 있기 때문에 이 첫 번째 단계에서는 팔레트에 설치하는 방법을 직접 확인할 수 있도록 Word를 포함한 패키지를 제거하도록 요청합니다. 단계 4는 컴포넌트 팔레트를 정상적인 상태로 되돌리는 방법에 대해 설명합니다.

다음과 같은 방법으로 컴포넌트 팔레트에서 Word를 제거합니다.

1 Component|Install 패키지를 선택합니다.

2 Borland C++Builder COM Server Components Sample Package를 클릭하고 Remove를 선택합니다.

C++Builder와 함께 제공되는 서버가 컴포넌트 팔레트의 Servers 페이지에 더 이상 표시되지 않습니다. 다른 서버를 임포트하지 않은 경우에는 Servers 페이지도 표시되지 않습니다.

단계 2: Word 타입 라이브러리 임포트

다음과 같은 방법으로 Word 타입 라이브러리를 임포트합니다.

1 Project|Import Type Library를 선택합니다.

2 Import Type Library 다이얼로그 박스에서 다음 작업을 수행합니다.

   1 Microsoft Office 8.0 Object Library를 선택합니다.

     Word(Version 8)가 리스트에 없는 경우에는 Add 버튼을 선택하고 Program Files\Microsoft Office\Office로 이동합니다. Word 타입 라이브러리 파일인 MSWord8.olb를 선택하고 Add를 선택한 다음 리스트에서 Word(Version 8)를 선택합니다.

   2 Palette Page에서 Servers를 선택합니다.

   3 Install을 선택합니다.

     Install 다이얼로그 박스가 나타납니다. Into New Packages 탭을 선택하고 WordExample을 입력하여 이 타입 라이브러리를 포함하는 새 패키지를 만듭니다.

3 Servers Palette Page로 이동하고 WordApplication을 선택하여 폼에 가져다 놓습니다.

4 다음 단계에 설명되어 있는 것처럼 버튼 객체에 대한 이벤트 핸들러를 작성합니다.

단계 3: Vtable 또는 dispatch 인터페이스 객체를 사용하여 Microsoft Word 제어

VTable 또는 dispatch 객체 중 하나를 사용하여 Microsoft Word를 제어할 수 있습니다.

VTable 인터페이스 객체 사용

WordApplication 객체의 인스턴스를 폼에 가져다 놓으면 Vtable 인터페이스 객체를 사용하여 컨트롤에 쉽게 액세스할 수 있습니다. 방금 만든 클래스의 메소드를 호출하기만 하면 됩니다. Word의 경우에는 *TWordApplication* 클래스입니다.

1 버튼을 선택하고 *OnClick* 이벤트 핸들러를 더블 클릭한 후 다음과 같은 이벤트 처리 코드를 제공합니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  if (OpenDialog1->Execute())
  {
    TVariant FileName = OpenDialog1->FileName.c_str();
    WordApplication1->Documents->Open(&FileName);

    WordApplication1->ActiveDocument->PrintOut();
  }
}
```

2 프로그램을 작성하고 실행합니다. 버튼을 클릭하면 Word 에서 인쇄할 파일을 묻는 메시지가 나타납니다.

dispatch 인터페이스 객체 사용

다른 방법으로는 dispatch 인터페이스를 사용하여 후기 연결하는 방법이 있습니다. dispatch 인터페이스 객체를 사용하려면 다음과 같이 _ApplicationDisp dispatch 랩퍼 클래스를 사용하여 Application 객체를 만들고 초기화합니다. dispinterface 메소드는 소스에 의해 VTable 인터페이스를 반환하는 것으로 문서화되지만 실제로는 dispatch 인터페이스에 캐스트해야 합니다.

1 버튼을 선택하고 *OnClick* 이벤트 핸들러를 더블 클릭한 후 다음과 같은 이벤트 처리 코드를 제공합니다.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
if (OpenDialog1->Execute())
  {
    TVariant FileName = OpenDialog1->FileName.c_str();

    _ApplicationDisp MyWord;
    MyWord.Bind(CLSID_WordApplication);
    DocumentsDisp MyDocs = MyWord->Documents;
    MyDocs->Open(&FileName);

    _DocumentDisp MyActiveDoc = MyWord->ActiveDocument;
    MyActiveDoc->PrintOut();
    MyWord->Quit();
}
```

2 프로그램을 작성하고 실행합니다. 버튼을 클릭하면 Word 에서 인쇄할 파일을 묻는 메시지가 나타납니다.

단계 4: 예제 삭제

이 예제를 완료한 후에는 C++Builder를 원래 형태로 복원하고 싶을 것입니다.

1 이 Servers 페이지의 객체를 삭제합니다.

- Component | Install Packages를 선택합니다.
- 리스트에서 WordExample 패키지를 선택하고 Remove를 클릭합니다.
- 확인을 요청하는 메시지 박스에서 Yes를 클릭합니다.
- OK를 클릭하여 Install Packages 다이얼로그 박스를 종료합니다.

2 Borland C++Builder COM Server Components Sample Package 패키지로 돌아갑니다.

- Component | Install Packages를 선택합니다.
- Add 버튼을 클릭합니다.
- 다이얼로그 박스에서 Office 97 컴포넌트에 대한 bcb97axserver60.bpl을 선택하거나 Office 2000 컴포넌트에 대한 bcb2kaxserver60.bpl을 선택합니다.
- OK를 클릭하여 Install Packages를 종료합니다.

타입 라이브러리 정의를 기반으로 한 클라이언트 코드 작성

ActiveX 컨트롤러를 호스팅하려면 컴포넌트 랩퍼를 사용해야 하지만, *TypeLibName*_TLB 유닛에 나타나는 타입 라이브러리의 정의만 사용하여 Automation 컨트롤러를 작성할 수 있습니다. 이 프로세스는 컴포넌트가 작업을 수행하도록 두는 것보다 약간 복잡합니다. 특히 이벤트에 응답해야 하는 경우가 이에 해당됩니다.

서버에 연결

컨트롤러 애플리케이션에서 Automation 서버를 작동하려면 지원하는 인터페이스에 대한 참조를 확보해야 합니다. 일반적으로 메인 인터페이스를 통해 서버에 연결합니다. 예를 들면, WordApplication 컴포넌트를 통해 Microsoft Word에 연결합니다.

메인 인터페이스가 이중 인터페이스인 경우에는 *TypeLibName*_TLB.h 파일에 있는 작성자 객체를 사용할 수 있습니다. 작성자 클래스는 CoClass 이름과 같으며 앞에 "co"가 추가됩니다. *Create* 메소드를 호출하여 같은 시스템상의 서버에 연결하거나 *CreateRemote* 메소드를 사용하여 원격 시스템상의 서버에 연결할 수 있습니다. *Create* 메소드와 *CreateRemote* 메소드는 정적 메소드이기 때문에 호출하는 데 작성자 클래스의 인스턴스가 필요하지 않습니다.

```
pInterface = CoServerClassName.Create();
pInterface = CoServerClassName.CreateRemote("Machine1");
```

Create 및 *CreateRemote* 메소드는 TComInterface 템플릿을 사용하여 *TypeLibName*_TLB.h 에서 정의한 CoClass의 디폴트 인터페이스를 위한 클래스 랩퍼를 반환합니다.

디폴트 인터페이스가 dispatch 인터페이스인 경우에는 CoClass를 위한 Creator 클래스가 생성되지 않습니다. 그 대신 디폴트 인터페이스를 위해 자동으로 생성된 랩퍼 클래스의 인스턴스를 생성해야 합니다. 이 클래스는 *TAutoDriver* 템플릿을 사용하여 정의되며 *InterfaceName*Disp와 같은 형태의 이름을 가집니다. 다음에는 *Bind* 메소드를 호출하여 CoClass를 위한 GUID를 전달합니다. _TLB 유닛의 맨 앞에 이 GUID를 위한 상수가 정의되어 있습니다.

이중 인터페이스를 사용한 Automation 서버 제어

자동으로 생성된 작성자 클래스를 사용하여 서버에 연결한 후 "->" 연산자를 사용하여 인터페이스 랩퍼 객체의 메소드를 호출합니다. 예를 들면, 다음과 같습니다.

```
TComApplication AppPtr = CoWordApplication_.Create();
AppPtr->DoSomething;
```

인터페이스 랩퍼와 작성자 클래스는 타입 라이브러리를 임포트할 때 자동으로 생성된 *TypeLibName*_TLB 유닛에 정의되어 있습니다. 이 인터페이스 랩퍼 클래스를 사용하면 랩퍼 클래스가 삭제될 때 원본으로 사용하는 인터페이스가 자동으로 해제되는 이점이 있습니다.

이중 인터페이스에 대한 내용은 41-12페이지의 "이중 인터페이스"를 참조하십시오.

dispatch 인터페이스를 사용하여 Automation 서버 제어

일반적으로 위에 설명한 것처럼 이중 인터페이스를 사용하여 Automation 서버를 제어합니다. 그러나, 이중 인터페이스를 사용할 수 없기 때문에 dispatch 인터페이스를 사용하여 Automation 서버를 제어해야 하는 경우도 있습니다.

다음과 같은 방법으로 dispatch 인터페이스의 메소드를 호출합니다.

1 dispatch 인터페이스를 위한 랩퍼 클래스의 *Bind* 메소드를 사용하여 서버에 연결합니다. 서버 연결에 대한 자세한 내용은 40-13페이지의 "서버에 연결"을 참조하십시오.

2 dispatch 인터페이스 랩퍼 객체의 메소드를 호출하여 Automation 서버를 제어합니다.

랩퍼 클래스는 dispatch 인터페이스의 속성과 메소드를 자체 속성 및 메소드로 표면화합니다. 또한 이 랩퍼 클래스는 *TAutoDriver*의 자손이기 때문에 *IDispatch* 메커니즘을 사용하여 서버 객체의 속성과 메소드를 호출할 수 있습니다. 속성이나 메소드에 액세스하려면 dispatch ID를 확보해야 합니다. 이를 위해서는 *GetIDsOfNames* 메소드를 사용하여 이름과 그 이름을 위한 dispatch ID를 받아들이는 DISPID 변수에 대한 참조를 전달합니다. 일단 dispatch ID를 확보한 후에는 이를 사용하여 *OlePropertyGet*, *OlePropertyPut* 또는 *OleFunction* 을 호출하여 서버 객체 속성과 메소드에 액세스할 수 있습니다.

dispatch 인터페이스를 사용하는 다른 방법은 *Variant*에 할당하는 것입니다. 일부 VCL 객체는 해당 값이 인터페이스인 속성이나 매개변수에 이 방법을 사용합니다. Variant 타입에는 dispatch 인터페이스 랩퍼 클래스의 메소드에 해당하는 *OlePropertyGet*, *OlePropertySet*, *OleFunction* 및 *OleProcedure* 메소드를 통해 dispatch 인터페이스를 호출하기 위한 지원이 내장되어 있습니다. *Variant*를 사용하면 *Variant* 메소드가 호출될 때마다 dispatch ID를 동적으로 검색하기 때문에 dispatch 인터페이스 랩퍼 클래스를 사용하는 것보다 속도가 다소 느릴 수 있습니다. 그러나 타입 라이브러리를 임포트할 필요가 없다는 이점이 있습니다.

경고 인터페이스를 dispatch 인터페이스 랩퍼 대신 *Variant*에 할당할 때는 주의해야 합니다. 랩퍼는 *AddRef* 및 *Release*에 대한 호출을 자동으로 처리하지만 *Variant*는 그렇지 않습니다. 따라서, 예를 들어 해당 값이 인터페이스인 *Variant*를 인터페이스 랩퍼에 할당하는 경우에는 인터페이스 랩퍼가 *AddRef*를 호출하지 않고 소멸자가 *Release*를 호출합니다. 이 때문에 *Variant*는 이미 VCL 객체의 속성으로 나타나는 경우가 아니면 사용하지 않는 것이 좋습니다.

dispatch 인터페이스에 대한 자세한 내용은 41-12페이지의 "Automation 인터페이스"를 참조하십시오.

Automation 컨트롤러에서 이벤트 처리

해당 타입 라이브러리를 임포트하는 객체에 대한 컴포넌트 랩퍼를 생성하는 경우 생성된 컴포넌트에 추가된 이벤트를 사용하여 이벤트에 응답할 수 있습니다. 그러나 컴포넌트 랩퍼를 사용하지 않거나 서버가 COM+ 이벤트를 사용하는 경우에는 이벤트 싱크(sink) 코드를 직접 작성해야 합니다.

프로그램으로 Automation 이벤트 처리

이벤트를 처리하려면 먼저 이벤트 싱크를 정의해야 합니다. 이벤트 싱크는 서버의 타입 라이브러리에 정의된 이벤트 dispatch 인터페이스를 구현하는 클래스입니다.

이벤트 싱크는 *TEventDispatcher*의 자손이며, 이벤트 싱크 클래스 및 이벤트 싱크에서 처리하는 이벤트 인터페이스를 위한 GUID의 두 가지 매개변수가 필요한 템플릿화된 클래스입니다.

```
class MyEventSinkClass: TEventDispatcher<MyEventSinkClass,
DIID_TheServerEvents>
{
...// declare the methods of DIID_TheServerEvents here
}
```

일단 이벤트 싱크 클래스의 인스턴스를 가지게 되면 *ConnectEvents* 메소드를 호출하여 서버에 이벤트 싱크에 대해 알립니다. 이 메소드는 서버의 *IConnectionPointContainer* 및 *IConnectionPoint* 인터페이스를 사용하여 객체를 이벤트 싱크로 등록합니다. 이제 객체는 이벤트가 발생할 때 서버에서 호출됩니다:

```
pInterface = CoServerClassName.CreateRemote("Machine1");
MyEventSinkClass ES;
ES.ConnectEvents(pInterface);
```

이벤트 싱크를 해제하려면 먼저 연결을 종료해야 합니다 . 연결을 종료하려면 이벤트 싱크의 *DisconnectEvents* 메소드를 호출합니다.

```
ES.DisconnectEvents(pInterface);
```

참고 이벤트 싱크를 해제하기 전에 서버에서 이벤트 싱크에 대한 연결을 해제했는지 확인해야 합니다. *DisconnectEvents*에 의한 연결 끊기 통지에 서버가 어떻게 응답할지 모르기 때문에 호출 후 즉시 이벤트 싱크를 해제하면 경쟁적인 상황이 될 수 있습니다. *TEventDispatcher*는 서버가 이벤트 싱크의 인터페이스를 해제할 때까지 감소되지 않는 자체 참조 카운트를 사용하여 이러한 상황이 일어나는 것을 방지합니다.

COM+ 이벤트 처리

COM+에서 서버는 일련의 특별한 인터페이스(*IConnectionPointContainer* 및 *IConnectionPoint*) 대신 특별한 helper 객체를 사용합니다. 이 때문에 *TeventDispatcher*의 자손인 이벤트 싱크를 사용할 수 없습니다. *TEventDispatcher*는 COM+ 이벤트 객체가 아니라 그러한 인터페이스를 사용하도록 디자인된 것입니다.

이벤트 싱크를 정의하는 대신 클라이언트 애플리케이션에서는 구독자 객체를 정의합니다. 구독자 객체는 이벤트 싱크와 마찬가지로 이벤트 인터페이스의 구현을 제공합니다. 구독자 객체는 서버의 연결 포인트에 연결하지 않고 오히려 특정 이벤트 객체를 구독한다는 점에서 이벤트 싱크와 다릅니다.

구독자 객체를 정의하려면 COM Object 마법사를 사용하여 이벤트 객체의 인터페이스를 구현하려는 인터페이스로 선택합니다. 마법사에서는 이벤트 핸들러를 만들 수 있는 기본 메소드를 포함한 구현 유닛을 생성합니다. COM Object 마법사를 사용한 기존 인터페이스 구현에 대한 자세한 내용은 41-2페이지의 "COM Object 마법사 사용"을 참조하십시오.

참고 이벤트 객체의 인터페이스가 구현할 수 있는 인터페이스 리스트에 없으면 마법사를 사용하여 레지스트리에 추가해야 합니다.

구독자 객체를 만든 후에는 이벤트 객체의 인터페이스나 그 인터페이스의 각 메소드(이벤트)를 구독해야 합니다. 선택할 수 있는 구독의 종류는 다음 세 가지입니다.

- **일시 구독 (Transient subscription).** 일시 구독은 종래의 이벤트 싱크와 마찬가지로 객체 인스턴스의 수명과 연관됩니다. 구독자 객체를 사용할 수 있도록 해제하면 구독이 해제되고 COM+에서 더 이상 이벤트를 전송하지 않습니다.

- **영구 구독 (Persistent subscription).** 영구 구독은 특정 객체 인스턴스보다는 객체 클래스와 연관됩니다. 이벤트가 발생하면 COM 에서 구독자 객체의 인스턴스를 찾거나 시작하고 이벤트 핸들러를 호출합니다. in-process 객체(DLL)는 영구 구독을 사용합니다.
- **사용자별 구독(Per-user subscription).** 이러한 구독은 일시 구독보다 안전한 버전을 제공합니다. 이벤트를 발생시키는 구독자 객체와 서버 객체는 모두 같은 시스템에서 같은 사용자 계정을 사용하여 실행되어야 합니다.

이벤트 객체를 구독하려면 전역 *RegisterComPlusEventSubscription* 함수를 사용합니다.

참고 COM+ 이벤트를 구독하는 객체는 COM+ 애플리케이션에 설치되어야 합니다.

타입 라이브러리가 없는 서버를 위한 클라이언트 생성

OLE와 같은 이전의 일부 COM 기술은 타입 라이브러리에 타입 정보를 제공하지 않습니다. 그 대신 미리 정의된 표준 인터페이스 집합에 의존합니다. 그러한 객체를 호스팅하는 클라이언트를 작성하려면 *TOleContainer* 컴포넌트를 사용합니다. 이 컴포넌트는 컴포넌트 팔레트의 System 페이지에 표시됩니다.

*TOleContainer*는 Ole2 객체를 위한 호스트 사이트 역할을 하며, *IOleClientSite* 인터페이스와 옵션인 *IOleDocumentSite*를 구현합니다. 통신은 OLE 동사를 사용하여 처리됩니다.

다음과 같은 방법으로 *TOleContainer*를 사용합니다.

1. *TOleContainer* 컴포넌트를 폼에 가져다 놓습니다.
2. Active Document를 호스팅하려는 경우에는 *AllowActiveDoc* 속성을 **true**로 설정합니다.
3. *AllowInPlace* 속성을 설정하여 호스팅된 객체를 *TOleContainer*에 표시해야 할지 아니면 독립적인 윈도우에 표시해야 할지 나타냅니다.
4. 객체가 활성화, 비활성화, 이동 또는 크기 변경될 때 응답하는 이벤트 핸들러를 작성합니다.
5. 디자인 타임에 *TOleContainer* 객체를 연결하려면 마우스 오른쪽 버튼으로 Insert Object를 클릭하여 선택합니다. Insert Object 다이얼로그 박스에서 호스팅할 서버 객체를 선택합니다.
6. 런타임에 *TOleContainer* 객체를 연결하려면 서버 객체 식별 방법에 따라 몇 가지 메소드 중에서 선택할 수 있습니다. 선택할 수 있는 메소드로는 프로그램 ID를 취하는 *CreateObject*, 객체가 저장될 파일의 이름을 취하는 *CreateObjectFromFile*, 객체를 만드는 방법에 대한 정보를 포함하는 구조체를 취하는 *CreateObjectFromInfo* 또는 객체가 저장되어 있는 파일 이름을 취해서 포함하는 것이 아니라 연결하는 *CreateLinkToFile*이 있습니다.
7. 일단 객체가 연결된 후에는 *OleObjectInterface* 속성을 사용하여 인터페이스에 액세스할 수 있습니다. 그러나, Ole2 객체와의 통신은 OLE 동사를 기반으로 하기 때문에 *DoVerb* 메소드를 사용하여 명령을 서버에 보내려고 할 가능성이 큽니다.
8. 서버 객체를 해제하려면 *DestroyObject* 메소드를 호출합니다.

장

41

간단한 COM 서버 생성

C++Builder는 다양한 COM 객체를 만들 수 있도록 도와주는 마법사를 제공합니다. 가장 단순한 COM 객체는 클라이언트에서 호출할 수 있는 디폴트 인터페이스를 통해 속성과 메소드 또는 이벤트를 노출하는 서버입니다.

참고 COM 서버 및 Automation은 CLX 애플리케이션에서 사용할 수 없습니다. COM 서버 및 Automation 기술은 Windows 에서만 사용할 수 있으며 크로스 플랫폼을 지원하지는 않습니다.

특히 다음 두 마법사를 사용하면 간단한 COM 객체를 쉽게 만들 수 있습니다.

- COM Object 마법사는 디폴트 인터페이스가 *IUnknown* 의 자손이거나 시스템에 이미 등록된 인터페이스를 구현하는 lightweight COM 객체를 만듭니다. 이 마법사를 사용하면 다양한 타입의 COM 객체를 만들 수 있습니다.
- Automation Object 마법사는 디폴트 인터페이스가 *IDispatch*의 자손인 간단한 Automation 객체를 만듭니다. *IDispatch*는 표준 마샬링 메커니즘과 인터페이스 호출에 대한 후기 연결 지원을 도입합니다.

참고 COM은 특정 상황을 처리하기 위한 여러 표준 인터페이스와 메커니즘을 정의합니다. C++Builder 마법사는 가장 많이 수행하는 작업을 자동화합니다. 그러나 사용자 정의 마샬링과 같은 일부 작업은 모든 C++Builder 마법사에서 지원되지 않습니다. 사용자 정의 마샬링 및 C++Builder에서 명시적으로 지원하지 않는 다른 기술에 대한 자세한 내용은 MSDN (Microsoft Developerís Network) 문서를 참조하십시오. Microsoft 웹 사이트에서는 COM 지원에 대한 현재 정보도 제공합니다.

COM 객체 생성 개요

Automation Object 마법사를 사용하여 새 Automation 서버를 만들거나 아니면 COM Object 마법사를 사용하여 다른 COM 객체 타입을 만들거나 수행하는 과정은 동일하며, 다음 단계를 포함합니다.

1. COM 객체 생성
2. COM Object 마법사 또는 Automation Object 마법사를 사용하여 서버 객체 생성
3. Project Options 다이얼로그 박스의 ATL 페이지에서 COM이 객체를 수용하는 애플리케이션을 호출하는 방법과 원하는 디버깅 지원 타입을 지정하기 위해 옵션 지정
4. 객체가 클라이언트에 노출하는 인터페이스 정의
5. COM 객체 등록
6. 애플리케이션 테스트 및 디버깅

COM 객체 디자인

COM 객체를 디자인하는 경우 구현하려는 COM 인터페이스를 결정해야 합니다. COM 객체를 작성하여 미리 정의된 인터페이스를 구현하거나 새 인터페이스를 정의하여 객체에서 구현할 수도 있습니다. 또한 객체에서 두 개 이상의 인터페이스를 지원하게 할 수도 있습니다. 지원하려는 표준 COM 인터페이스에 대한 자세한 내용은 MSDN 문서를 참조하십시오.

- 기존 인터페이스를 구현하는 COM 객체를 만들려면 COM Object 마법사를 사용하십시오.
- 새로 정의한 인터페이스를 구현하는 COM 객체를 만들려면 COM Object 마법사나 Automation Object 마법사를 사용하십시오. COM Object 마법사에서는 *IUnknown*의 자손인 새로운 디폴트 인터페이스를 생성할 수 있고 Automation Object 마법사에서는 *IDispatch*의 자손인 디폴트 인터페이스를 객체에 제공할 수 있습니다. 어떤 마법사를 사용하든 나중에 Type Library Editor를 사용하여 마법사에서 생성한 디폴트 인터페이스의 상위 인터페이스를 변경할 수 있습니다.

지원할 인터페이스를 결정하고 in-process 서버나 out-of-process 서버 또는 원격 서버 중 COM 객체에 해당하는 서버도 결정해야 합니다. in-process 서버인 경우와 타입 라이브러리를 사용하는 out-of-process 서버 및 원격 서버인 경우에는 COM이 데이터를 마샬링합니다. 그렇지 않은 경우에는, out-of-process 서버로 데이터를 마샬링하는 방법에 대해 고려해 보아야 합니다. 서버 타입에 대한 자세한 내용은 38-6페이지의 "in-process, out-of-process 및 원격 서버"를 참조하십시오.

COM Object 마법사 사용

COM Object 마법사는 다음과 같은 작업을 수행합니다.

- 새 유닛을 만듭니다.
- ATL 클래스 CComObjectRootEx 및 CComCoClass의 자손인 새 클래스를 정의합니다. 기본 클래스에 대한 자세한 내용은 38-22페이지의 "마법사에 의해 작성된 코드"를 참조하십시오.
- 타입 라이브러리를 프로젝트에 추가하고 객체 및 해당 인터페이스를 타입 라이브러리에 추가합니다.

COM 객체를 만들기 전에 구현할 기능을 포함하는 애플리케이션용 프로젝트를 만들거나 엽니다. 프로젝트는 필요에 따라 애플리케이션이나 ActiveX 라이브러리가 될 수 있습니다.

다음과 같은 방법으로 COM Object 마법사를 표시합니다.

1 File|New|Other를 선택하여 New Items 다이얼로그 박스를 엽니다.

2 ActiveX 탭을 선택합니다.

3 COM 객체 아이콘을 더블 클릭합니다.

마법사에서 다음을 지정해야 합니다.

- **CoClass name:** 클라이언트에 표시되는 객체 이름입니다. 객체를 구현하기 위해 만들어지는 클래스 이름은 이 이름 앞에 'T'가 붙습니다. 기존 인터페이스를 구현하도록 선택하지 않으면 마법사는 이 이름 앞에 'I'가 있는 디폴트 인터페이스를 CoClass에 제공합니다.
- **Implemented Interface:** 디폴트로, 마법사는 객체에 *IUnknown*의 자손인 디폴트 인터페이스를 제공합니다. 마법사를 종료한 후 Type Library Editor를 사용하여 이 인터페이스에 속성과 메소드를 추가할 수 있습니다. 또한 객체가 구현할 미리 정의된 인터페이스를 선택할 수도 있습니다. COM Object 마법사에서 List 버튼을 클릭하면 Interface Selection 마법사가 나타납니다. 이 마법사를 사용하여 시스템에 등록된 타입 라이브러리에서 정의된 이중 또는 사용자 정의 인터페이스를 선택할 수 있으며, 여기서 선택한 인터페이스가 새 CoClass의 디폴트 인터페이스가 됩니다. Interface Selection 마법사가 이 인터페이스의 모든 메소드를 생성된 구현 클래스에 추가하므로 개발자는 구현 유닛에 메소드 바디를 채우기만 하면 됩니다. 기존 인터페이스를 선택한 경우에는 인터페이스가 프로젝트의 타입 라이브러리에 추가되지 않습니다. 따라서 객체를 배포할 때 인터페이스를 정의하는 타입 라이브러리도 배포해야 합니다.
- **Threading Model:** 일반적으로 객체에 대한 클라이언트 요청은 각각의 실행 스레드에 입력됩니다. 클라이언트가 객체를 호출할 때 COM이 스레드를 serialize하는 방법을 지정할 수 있습니다. 스레드 모델에 대한 선택에 따라 객체가 등록되는 방법이 결정됩니다. 개발자는 선택한 모델에 포함된 스레드를 지원해야 합니다. 선택할 수 있는 다른 스레드 모델에 대한 자세한 내용은 41-5페이지의 "스레드 모델 선택"을 참조하십시오. 애플리케이션에 스레드 지원을 제공하는 방법에 대한 자세한 내용은 11장, "멀티 스레드 애플리케이션 개발"을 참조하십시오.
- **Event support:** 클라이언트가 응답할 수 있는 이벤트를 객체에서 생성하도록 할지 여부를 지정해야 합니다. COM Object 마법사는 클라이언트 이벤트 핸들러에 대한 호출 디스패칭과 이벤트 생성에 필요한 인터페이스를 지원합니다. 이벤트가 작동하는 방법과 이벤트를 구현할 때 수행해야 하는 작업에 대한 자세한 내용은 41-10페이지의 "클라이언트에 이벤트 제공"을 참조하십시오.
- **Oleautomation:** Automation과 호환 가능한 타입으로 한정하는 경우 in-process 서버를 생성하지 않는다면 COM이 마샬링을 처리하도록 할 수 있습니다. 타입 라이브러리에서 객체의 인터페이스를 OleAutomation으로 표시하면 COM이 대리자와 스텁을 설정하고 프로세스 경계를 통해 매개변수를 전달할 수 있습니다. 이 프로세스에 대한 자세한 내용은 38-8페이지의 "마샬링 메커니즘"을 참조하십시오. 새 인터페이스를 생성하는 경우에는 인터페이스가 Automation과 호환 가능한지 여부만 지정할 수 있습니다. 기존 인터페이스를 선택하는 경우에는 인터페이스 어트리뷰트(attribute)가 타입 라이브러리에 미리 지정되어 있습니다. 객체의 인터페이스를 OleAutomation으로 표시하지 않으면 in-process 서버를 만들거나 마샬링 코드를 직접 작성해야 합니다.

- **Implement Ancestor Interfaces:** 마법사가 상속된 인터페이스에 대해 스텁 루틴을 제공하게 하려면 이 옵션을 선택합니다. 마법사에서 구현하지 않는 상속된 인터페이스로는 *IUnknown*, *IDispatch* 및 *IAppServer*의 세 가지 인터페이스가 있습니다. *IUnknown* 및 *IDispatch*가 구현되지 않는 이유는 ATL에서 두 인터페이스를 자체적으로 구현하기 때문입니다. *IAppServer*는 클라이언트 데이터셋과 데이터셋 프로바이더로 작업할 때 자동으로 구현되므로 따로 구현되지 않습니다.

COM 객체에 대한 설명을 옵션으로 추가할 수도 있습니다. 이 설명은 객체의 타입 라이브러리에 표시됩니다.

Automation Object 마법사 사용

Automation Object 마법사는 다음과 같은 작업을 수행합니다.

- 새 유닛을 만듭니다.
- ATL 클래스 *CComObjectRootEx* 및 *CComCoClass*의 자손인 새 클래스를 정의합니다. 기본 클래스에 대한 자세한 내용은 38-22페이지의 "마법사에 의해 작성된 코드"를 참조하십시오.
- 타입 라이브러리를 프로젝트에 추가하고 객체와 객체의 인터페이스를 타입 라이브러리에 추가합니다.

Automation 객체를 만들기 전에 노출할 기능을 포함하는 애플리케이션용 프로젝트를 만들거나 엽니다. 프로젝트는 필요에 따라 애플리케이션이나 ActiveX 라이브러리가 될 수 있습니다.

다음과 같은 방법으로 Automation 마법사를 표시합니다.

1 File | New | Other를 선택합니다.

2 ActiveX 탭을 선택합니다.

3 Automation 객체 아이콘을 더블 클릭합니다.

마법사 다이얼로그 박스에서 다음을 지정합니다.

- **CoClass name:** 클라이언트에 표시되는 객체 이름입니다. 객체의 디폴트 인터페이스는 이 이름 앞에 'I'가 붙은 이름으로 만들어지고 객체를 구현하기 위해 만들어지는 클래스의 이름은 이 이름 앞에 'T'가 붙습니다.
- **Threading Model**: 일반적으로 객체에 대한 클라인언트 요청은 각각의 실행 스레드에 입력됩니다. 클라이언트가 객체를 호출할 때 COM이 스레드를 serialize하는 방법을 지정할 수 있습니다. 스레드 모델에 대한 선택에 따라 객체가 등록되는 방법이 결정됩니다. 개발자가 선택한 모델에 포함된 스레드 지원이 제공됩니다. 선택할 수 있는 다른 스레드 모델에 대한 자세한 내용은 41-5페이지의 "스레드 모델 선택"을 참조하십시오. 애플리케이션에 스레드 지원을 제공하는 방법에 대한 자세한 내용은 11장, "멀티 스레드 애플리케이션 개발"을 참조하십시오.
- **Event support**: 클라이언트가 응답할 수 있는 이벤트를 객체에서 생성하도록 할지 여부를 지정해야 합니다. Automation 마법사는 클라이언트 이벤트 핸들러에 대한 호출 디스패칭과 이벤트 생성에 필요한 인터페이스를 지원합니다. 이벤트가 작동하는 방법과 이벤트를 구현할 때 수행해야 하는 작업에 대한 자세한 내용은 41-10페이지의 "클라이언트에 이벤트 제공"을 참조하십시오.

COM 객체에 대한 설명을 옵션으로 추가할 수도 있습니다. 이 설명은 객체의 타입 라이브러리에 표시됩니다.

Automation 객체는 Vtable을 통한 초기(컴파일 타임) 연결과 *IDispatch* 인터페이스를 통한 후기(런타임) 연결 모두를 지원하는 **이중 인터페이스**를 구현합니다. 자세한 내용은 41-12페이지의 "이중 인터페이스"를 참조하십시오.

스레드 모델 선택

마법사를 사용하여 객체를 만들 때 객체가 지원할 스레드 모델을 선택합니다. COM 객체에 스레드 지원을 추가하면 여러 클라이언트에서 애플리케이션을 동시에 액세스할 수 있으므로 객체 성능을 개선할 수 있습니다.

표 41.1은 지정할 수 있는 여러 스레드 모델을 나열한 것입니다.

표 41.1 COM 객체에 대한 스레드 모델

스레드 모델	설명	구현의 장단점
Single	서버에서 스레드 지원을 제공하지 않습니다. COM이 클라이언트 요청을 serialize 하여 애플리케이션이 요청을 한 번에 하나씩 받도록 합니다.	한 번에 하나의 클라이언트를 처리하므로 스레드 지원이 필요하지 않습니다. 성능상의 이점은 없습니다.
Apartment 또는 Single-threaded Apartment	COM이 한 번에 하나의 클라이언트 스레드에서만 객체를 호출할 수 있도록 합니다. 모든 클라이언트 호출은 객체가 만들어진 스레드를 사용합니다.	객체는 각각의 인스턴스 데이터에 안전하게 액세스할 수 있지만, 글로벌 데이터는 임계 구역이나 다른 형태의 serialization을 사용하여 보호해야 합니다. 여러 호출에 대해 스레드 로컬 변수를 신뢰할 수 있습니다. 성능에 약간의 이점이 있습니다.
Free(Multi-threaded Apartment라고도 함)	객체가 한 번에 여러 스레드의 호출을 받을 수 있습니다.	객체는 모든 인스턴스 및 글로벌 데이터를 임계 구역이나 다른 형태의 serializtion을 사용하여 보호해야 합니다. 여러 호출에 대해 스레드 로컬 변수를 신뢰할 수 *없습니다.*

표 41.1 COM 객체에 대한 스레드 모델(계속)

스레드 모델	설명	구현의 장단점
Both	이 모델은 나가는 호출(예: 콜백)이 같은 스레드에서 실행된다는 점을 제외하면 Free-threaded 모델과 동일합니다.	최적의 성능과 뛰어난 유연성을 가지고 있습니다. 애플리케이션이 나가는 호출에 제공되는 매개변수에 대해 스레드 지원을 제공할 필요가 없습니다.
Neutral	여러 클라이언트가 다른 스레드에서 객체를 동시에 호출할 수 있으며 COM이 두 호출이 충돌하지 않도록 보장합니다.	여러 메소드에서 액세스하는 인스턴스 데이터와 글로벌 데이터를 포함하는 스레드 충돌을 방지해야 합니다. 사용자 인터페이스(보이는 컨트롤)가 있는 객체에 이 모델을 사용할 수 없습니다. 이 모델은 COM+에서만 사용할 수 있습니다. COM에서 이 모델은 Apartment 모델로 매핑됩니다.

참고 콜백에 있는 로컬 변수 이외의 로컬 변수는 스레드 모델에 관계 없이 항상 안전합니다. 그 이유는 로컬 변수가 스택에 저장되고 각 스레드는 고유한 스택을 가지기 때문입니다. Free 스레드를 사용하는 경우 콜백의 로컬 변수가 안전하지 않을 수 있습니다.

마법사에서 선택하는 스레드 모델에 따라 객체가 시스템 레지스트리에 등록되는 방법이 결정됩니다. 선택한 스레드 모델을 객체 구현에서 유지하도록 해야 합니다. 스레드에 안전한 코드 작성에 대한 일반적인 내용은 11장, "멀티 스레드 애플리케이션 개발"을 참조하십시오.

참고 COM이 지정된 스레드 모델에 따라 객체를 호출하려면 해당 모델을 지원하도록 COM을 초기화해야 합니다. Project Options 다이얼로그 박스의 ATL 페이지를 사용하여 COM을 초기화하는 방법을 지정할 수 있습니다.

in-process 서버의 경우 마법사에서 스레드 모델을 설정하면 CLSID 레지스트리 항목에 스레드 모델 키가 설정됩니다.

Out-of-process 서버는 EXE로 등록되며 COM은 필요한 최상위 스레드 모델로 초기화되어야 합니다. 예를 들어 EXE에 Free-threaded 객체가 들어 있으면 COM은 Free 스레드 모델로 초기화되므로 EXE에 포함된 Free-threaded나 Apartment-threaded 객체에 대해 필요한 지원을 제공할 수 있습니다. COM을 초기화하는 방법을 지정하려면 Project Options 다이얼로그 박스의 ATL 페이지를 사용하십시오.

Free 스레드 모델을 지원하는 객체 작성

둘 이상의 스레드에서 객체를 액세스해야 하는 경우에는 항상 Apartment 스레드가 아니라 Free 스레드 또는 Both 스레드 모델을 사용합니다. 일반적인 예로는 원격 컴퓨터의 객체에 연결된 클라이언트 애플리케이션이 있습니다. 원격 클라이언트가 해당 객체의 메소드를 호출하면 서버는 서버 컴퓨터에 있는 스레드 풀의 스레드에서 이 호출을 받습니다. 이러한 받는 스레드는 로컬에서 실제 객체를 호출합니다. 그리고 객체가 Free 스레드 모델을 지원하므로 스레드는 객체를 직접 호출할 수 있습니다.

객체가 Apartment 스레드 모델을 지원하는 경우에는 호출을 객체가 만들어진 스레드에 전송해야 하고 그 결과를 클라이언트에 반환하기 전에 받는 스레드로 다시 전송해야 합니다. 이렇게 하려면 추가 마샬링이 필요합니다.

Free 스레드를 지원하려면 . *각*메소드에 대해 인스턴스 데이터를 액세스하는 방법을 고려해야 합니다. 메소드에서 인스턴스 데이터에 쓰고 있다면 임계 구역이나 다른 serialization 형태를 사용하여 인스턴스 데이터를 보호해야 합니다. 임계 호출을 serialize하는 것이 COM 마샬링 코드를 실행하는 것보다 오버헤드가 적을 것입니다.

인스턴스 데이터가 읽기 전용이면 serialization이 필요하지 않습니다.

Free-threaded in-process 서버는 Free-threaded 마샬러와 함께 외부 객체로 작동하여 성능을 개선할 수 있습니다. Free-threaded 마샬러는 Free-threaded가 아닌 호스트(클라이언트)에서 Free-threaded DLL을 호출할 때 COM 표준 스레드 처리에 대한 바로 가기를 제공합니다.

다음과 같은 방법으로 Free-threaded 마샬러를 결합합니다.

- *CoCreateFreeThreadedMarshaler*를 호출하여 결과물인 Free-threaded 마샬러가 사용할 객체의 *IUnknown* 인터페이스를 전달합니다.

  ```
  CoCreateFreeThreadedMarshaler(static_cast<IUnknown *>(this),
  &FMarshaler);
  ```

  이 줄은 Free-threaded 마샬러의 인터페이스를 클래스 멤버인 *FMarshaler*에 할당합니다.
- Type Library Editor를 사용하여 *IMarshal* 인터페이스를 CoClass가 구현하는 인터페이스 집합에 추가합니다.
- 객체의 *QueryInterface* 메소드에서 IDD_IMarshal에 대한 호출을 위에서 *FMarshaler*로 저장한 Free-threaded 마샬러에 위임합니다.

경고 Free-threaded 마샬러는 효율을 높이기 위해 COM 마샬링의 일반적인 규칙을 위반하므로 주의해서 사용해야 합니다. 특히 Free-threaded 마샬러는 in-process 서버에 있는 Free-threaded 객체와만 결합되고, 다른 스레드가 아닌 Free 스레드를 사용하는 객체로만 인스턴스화되어야 합니다.

Apartment 스레드 모델을 지원하는 객체 작성

다음과 같은 몇 가지 규칙에 따라 Single-threaded Apartment 스레드 모델을 구현합니다.

- 애플리케이션에서 만들어지는 첫째 스레드가 COM의 메인 스레드가 됩니다. 일반적으로 이 스레드에서 WinMain이 호출됩니다. 또한 이 스레드는 COM 초기화를 해제하는 마지막 스레드가 됩니다.
- Apartment 스레드 모델의 각 스레드에는 메시지 루프가 있어야 하며 메시지 대기열을 자주 확인해야 합니다.
- 어떤 스레드에 COM 인터페이스에 대한 포인터가 있으면 그 포인터는 해당 스레드에서만 사용할 수 있습니다.

Single-threaded Apartment 모델은 스레드 지원을 제공하지 않는 모델과 완전한 Multi 스레드를 지원하는 Free 스레드 모델의 중간 형태입니다. Apartment 모델을 적용하는 서버에서는 서버가 서버의 모든 글로벌 데이터(예: 객체 카운트)에 대한 액세스를 serialize합니다. 그 이유는 다른 객체가 다른 스레드에서 글로벌 데이터를 액세스하려고 할 수 있기 때문입니다. 그러나 메소드가 항상 같은 스레드에서 호출되므로 객체 인스턴스 데이터는 안전합니다.

일반적으로 웹 브라우저에서 사용하는 컨트롤은 브라우저 애플리케이션이 스레드를 항상 Apartment로 초기화하기 때문에 Apartment 스레드 모델을 사용합니다.

Neutral 스레드 모델을 지원하는 객체 작성

COM+에서는 Free 스레드와 Apartment 스레드 사이에 있는 또 다른 스레드 모델인 Neutral 모델을 사용할 수 있습니다. Free 스레드 모델과 마찬가지로 이 모델에서는 동시에 여러 스레드가 객체를 액세스할 수 있습니다. 그리고 객체가 만들어진 스레드로 전송하기 위한 추가 마샬링도 없습니다. 그럼에도 불구하고 객체는 서로 충돌하는 호출을 받지 않습니다.

Neutral 스레드 모델을 사용하는 객체를 작성하려면 Apartment-threaded 스레드 객체를 작성할 때와 거의 같은 규칙을 따릅니다. 다만, Neutral 스레드 모델에서는 객체 인터페이스의 다른 메소드가 인스턴스 데이트를 액세스할 수 있는 경우 스레드 충돌에 대해 인스턴스 데이터를 보호할 필요가 있습니다. 단일 인터페이스 메소드에서만 액세스하는 인스턴스 데이터는 자동으로 스레드에 안전하게 됩니다.

ATL 옵션 지정

마법사를 사용하여 COM 객체를 만들면 Project Options 다이얼로그 박스에 'ATL'이라는 이름의 추가 페이지가 생깁니다. 이 페이지에서는 COM을 초기화하거나 애플리케이션을 등록하는 ATL 호출에 대해 생성되는 매개변수를 제어하는 여러 개의 애플리케이션 레벨 플래그와 ATL을 사용하는 코드에서 디버그 추적을 활성화할지 여부를 제어하는 플래그를 설정할 수 있습니다.

ATL 페이지를 사용하여 다음 옵션을 설정합니다.

- **인스턴싱:** 애플리케이션이 in-process 서버가 아니면 인스턴싱은 클라이언트가 단일 프로세스 공간에서 만들 수 있는 객체의 인스턴스 수를 결정합니다. 단일 사용을 지정한 경우에는 클라이언트가 객체를 인스턴스화하면 COM이 애플리케이션을 뷰에서 제거하므로 다른 클라이언트는 자체 애플리케이션 인스턴스를 시작해야 합니다. 다중 사용을 지정한 경우에는 여러 클라이언트가 각각 고유한 객체 인스턴스를 만들고 모든 인스턴스가 같은 프로세스 공간에서 실행됩니다.
- **OLE 초기화 COINIT_XXX 플래그:** 이 플래그는 애플리케이션에서 COM을 인스턴스하는 방법을 결정합니다. 각 객체가 항상 자신이 만들어진 스레드에서 호출되는 경우에는 Apartment-threaded를 지정하고, 각 객체가 여러 스레드에서 호출될 수 있는 경우에는 Multi-threaded를 지정할 수 있습니다. COINIT 플래그는 애플리케이션의 객체가 사용할 수 있는 스레드 모델을 결정합니다. COINIT 플레그를 Apartment-threaded로 설정하면 애플리케이션에 Single-threaded 객체나 Apartment-threaded 객체만 가질 수 있습니다. 다른 스레드 모델로 등록된 객체는 디폴트로 Apartment-threaded 객체가 됩니다.
- **디버깅 플래그:** 애플리케이션이 IDE에서 실행될 때 ATL 호출이 이벤트 로그에 기록되도록 지정하기 위해 플래그를 설정할 수도 있습니다. *IUnknown* 메소드(QueryInterface 및 참조 카운팅 호출)에 대한 호출을 추적하거나 모든 ATL 호출을 추적할 수 있습니다.

COM 객체 인터페이스 정의

마법사를 사용하여 COM 객체를 만들면 마법사는 자동으로 타입 라이브러리를 생성합니다. 이 타입 라이브러리를 통해 호스트 애플리케이션은 객체가 수행할 수 있는 작업을 찾을 수 있습니다. 또한 Type Library Editor를 사용하여 객체 인터페이스를 정의할 수도 있습니다. Type Library Editor에 정의하는 인터페이스에서 객체가 클라이언트에 노출하는 속성, 메소드 및 이벤트를 정의합니다.

참고 COM Object 마법사에서 기존 인터페이스를 선택하면 속성과 메소드를 추가할 필요가 없습니다. 인터페이스 정의는 인터페이스가 정의된 타입 라이브러리에서 임포트됩니다. 구현 유닛에서 임포트된 인터페이스 메소드를 찾아 바디를 채우기만 하면 됩니다.

객체 인터페이스에 속성 추가

Type Library Editor를 사용하여 객체 인터페이스에 속성을 추가하면 속성 값을 읽는 메소드 또는 속성 값을 설정하는 메소드가 자동으로 추가됩니다. 그런 다음 Type Library Editor는 이 메소드를 구현 클래스에 추가하고 구현 유닛에서 비어 있는 메소드 구현을 만들어 개발자가 이를 완성하게 합니다.

다음과 같은 방법으로 객체 인터페이스에 속성을 추가합니다.

1 Type library Editor에서 객체의 디폴트 인터페이스를 선택합니다.

  디폴트 인터페이스의 이름은 객체 이름 앞에 'I'를 붙입니다. 디폴트 인터페이스를 결정하려면 Type Library Editor 에서 CoClass 및 Implements 탭을 선택하고 구현 인터페이스 리스트에서 "Default"로 표시된 인터페이스를 찾습니다.

2 읽기 / 쓰기로 하려고 하면 툴바에서 Property 버튼을 클릭하거나 툴바에서 Property 버튼 옆에 있는 화살표를 클릭한 다음 노출할 속성 타입을 클릭합니다.

3 Attributes 창에서 속성 이름과 타입을 지정합니다.

4 툴바에서 Refresh 버튼을 클릭합니다.

  속성 액세스 메소드에 대한 정의 및 스켈레톤 구현이 객체 구현 유닛에 추가됩니다.

5 구현 유닛에 속성에 대한 액세스 메소드를 찾습니다. 이 메소드는 get_PropertyName 및 set_PropertyName이라는 형태의 이름을 가지며, 예외를 catch하고 HRESULT를 반환하는 코드만을 포함합니다. 객체의 속성 값을 가져오거나 설정하는 코드를 **try**와 **catch** 문 사이에 추가합니다. 이 코드는 단순히 애플리케이션에 있는 기존 함수를 호출하거나 객체 정의에 추가한 데이터 멤버를 액세스할 수 있으며, 또는 속성을 구현할 수도 있습니다.

객체 인터페이스에 메소드 추가

Type Library Editor를 사용하여 객체 인터페이스에 메소드를 추가하면 Type Library Editor는 이 메소드를 구현 클래스에 추가하고 구현 유닛에 비어 있는 구현을 만들어 개발자가 이를 완성하게 할 수 있습니다.

다음과 같은 방법으로 객체 인터페이스를 통해 메소드를 노출합니다.

1 Type Library Editor에서 객체에 대한 디폴트 인터페이스를 선택합니다.

디폴트 인터페이스의 이름은 객체 이름 앞에 'I'를 붙입니다. 디폴트 인터페이스를 결정하려면 Type Library Editor에서 CoClass 및 Implements 탭을 선택하고 구현 인터페이스 리스트에서 "Default"로 표시된 인터페이스를 찾습니다.

2 Method 버튼을 클릭합니다.

3 Attributes 창에서 메소드 이름을 지정합니다.

4 Parameters 창에서 메소드 반환 타입과 적절한 매개변수를 지정합니다.

5 툴바에서 Refresh 버튼을 클릭합니다.

메소드에 대한 정의 및 스켈레톤 구현이 객체 구현 유닛에 추가됩니다.

6 구현 유닛에서 새로 삽입된 메소드 구현을 찾습니다. 이 메소드는 완전히 비어 있습니다. 메소드가 나타내는 작업을 수행하도록 메소드 바디를 채웁니다.

클라이언트에 이벤트 제공

COM 객체가 생성할 수 있는 이벤트 타입은 일반적인 이벤트 및 COM+ 이벤트의 두 가지 타입입니다.

- COM+ 이벤트를 생성하려면 Event Object 마법사를 사용하여 각각의 이벤트 객체를 만들고 서버 객체에서 이 이벤트 객체를 호출하는 코드를 추가해야 합니다. COM+ 이벤트 생성에 대한 자세한 내용은 44-21페이지의 "COM+에서 이벤트 생성"을 참조하십시오.
- Event Object 마법사를 사용하여 일반적인 이벤트 생성에 필요한 많은 작업을 처리할 수 있습니다. 이 프로세스는 아래에 설명되어 있습니다.

다음과 같은 작업을 수행하여 객체가 이벤트를 생성하게 할 수 있습니다.

1 마법사에서 Generate Event Support Code 박스를 선택 표시합니다.

마법사는 디폴트 인터페이스와 함께 Events 인터페이스를 포함하는 객체를 만듭니다. 이 Events 인터페이스는 I*CoClassname*Events라는 형태의 이름을 가집니다. 이 이벤트는 나가는(소스) 인터페이스가 됩니다. 나가는 인터페이스는 객체에서 구현하는 인터페이스가 아니라 클라이언트에서 구현하며 객체에서 호출하는 인터페이스를 가리킵니다. 이 인터페이스를 확인하려면 CoClass를 선택하고 Implements 페이지로 이동하여 Events 인터페이스의 Source 열이 **true**로 표시되어 있는지 확인하면 됩니다. Events 인터페이스의 GUID가 객체 선언의 인터페이스 맵 아래에 표시되는 Connection Point 맵에 추가됩니다. Connection Points 맵에 대한 자세한 내용은 ATL 문서를 참조하십시오.

마법사는 Events 인터페이스와 함께 다른 기본 클래스를 객체에 추가합니다. 여기에는 *IConnectionPointContainer* 인터페이스(*IConnectionPointContainerImpl*) 구현 및 모든 클라이언트에 대한 이벤트 발생을 관리하는 템플릿화된 기본 클래스(TEvents_*CoClassName*)가 포함됩니다. TEvents_CoClassName 클래스에 대한 템플릿은 _TLB 유닛 헤더에 있습니다.

2 Type Library Editor에서 객체의 나가는 Events 인터페이스를 선택합니다. 이 인터페이스는 I*CoClassName*Events라는 형태의 이름을 가집니다.

3 Type Library 툴바에서 Method 버튼을 클릭합니다. Events 인터페이스에 추가하는 각 메소드는 클라이언트에서 구현해야 하는 이벤트 핸들러를 나타냅니다.

4 Attributes 창에서 이벤트 핸들러 이름(예: MyEvent)을 지정합니다.

5 툴바에서 Refresh 버튼을 클릭합니다.

이제 객체 구현에 클라이언트 이벤트 싱크를 승인하고 이벤트 발생 시 호출할 인터페이스 리스트를 관리하는 데 필요한 모든 사항을 갖추게 되었습니다. 이 인터페이스를 호출하면 TEvents_*CoClassName*에서 구현된 이벤트를 시작하는 메소드를 호출하여 클라이언트에 이벤트를 생성할 수 있습니다. 각 이벤트 핸들러에 따라 이 메소드는 Fire_*EventHandlerName*이라는 형태의 이름을 가집니다.

6 이벤트를 발생시켜 클라이언트에 이벤트 발생을 알려야 하는 경우에는 다음과 같이 이벤트를 모든 이벤트 싱크로 디스패치하는 메소드를 호출합니다.

```
if (EventOccurs) Fire_MyEvent; // Call method you created to fire events.
```

Automation 객체의 이벤트 관리

일반적인 COM 이벤트를 지원하는 서버는 클라이언트에서 구현하는 나가는 인터페이스에 대한 정의를 제공해야 합니다. 이 나가는 인터페이스는 클라이언트가 서버 이벤트에 응답하기 위해 구현해야 하는 모든 이벤트 핸들러를 포함합니다.

클라이언트는 나가는 이벤트 인터페이스를 구현한 경우 서버의 *IConnectionPointContainer* 인터페이스를 쿼리하여 이벤트 통지를 받으려는 의사를 표시합니다. *ConnectionPointContainer* 인터페이스는 서버의 *IConnectionPoint* 인터페이스를 반환합니다. 그러면 클라이언트가 이 인터페이스를 사용하여 이벤트 핸들러 구현(싱크라고 함)에 대한 포인터를 서버에 전달합니다.

서버는 모든 클라이언트 싱크 리스트를 가지고 있으며 위에서 설명한 대로 이벤트가 발생할 때 이 리스트에 있는 메소드를 호출합니다.

Automation 인터페이스

Automation Object 마법사는 디폴트로 이중 인터페이스를 구현하며, 이것은 Automation 객체가 다음 두 가지 기능을 모두 지원한다는 것을 의미합니다.

- 런타임 시 후기 연결은 *IDispatch* 인터페이스를 통해 수행됩니다. 이 기능은 dispatch 인터페이스, 즉 **dispinterface**로 구현됩니다.
- 컴파일 타임 시 초기 연결은 객체의 Vtable(가상 함수 테이블)에 있는 멤버 함수 중 하나를 직접 호출하여 수행됩니다. 이것을 **사용자 정의 인터페이스**라고 합니다.

참고 COM Object 마법사에서 생성한 인터페이스 중 *IDispatch* 의 자손이 아닌 것은 VTable 호출만을 지원합니다.

이중 인터페이스

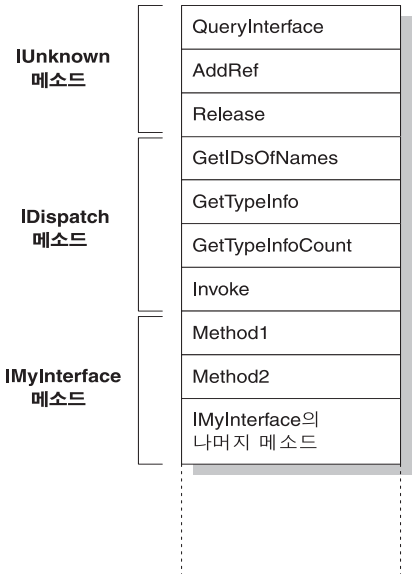
이중 인터페이스는 사용자 정의 인터페이스이며 동시에 dispinterface 입니다. 이중 인터페이스는 *IDispatch*의 자손인 COM Vtable 인터페이스로 구현됩니다. 런타임에만 액세스할 수 있는 컨트롤러에는 dispinterface를 사용할 수 있습니다. 컴파일 타임 연결을 사용할 수 있는 객체에는 좀 더 효율적인 VTable 인터페이스가 사용됩니다.

이중 인터페이스는 VTable 인터페이스와 dispinterface가 결합된 데에서 오는 다음과 같은 이점을 제공합니다.

- 타입 정보를 가져올 수 없는 Automation 컨트롤러에 대해 dispinterface가 객체에 런타임 액세스를 제공합니다.
- in-process 서버의 경우 VTable 인터페이스를 통해 더 빠르게 액세스할 수 있습니다.
- out-of-process 서버의 경우 COM이 Vtable 인터페이스와 dispinterface를 위해 데이터를 마샬링합니다. COM은 일반적인 대리자/스텁 구현을 제공하여 타입 라이브러리에 포함된 정보를 기반으로 인터페이스를 마샬링할 수 있습니다. 마샬링에 대한 자세한 내용은 41-14페이지의 "데이터 마샬링"을 참조하십시오.

다음 다이어그램은 *IMyInterface*라는 이중 인터페이스를 지원하는 객체의 *IMyInterface* 인터페이스를 보여 줍니다. 이중 인터페이스에 대한 처음 세 개의 Vtable 항목은 *IUnknown* 인터페이스를 나타내고 다음 네 항목은 *IDispatch* 인터페이스를 나타내며 나머지 항목은 사용자 정의 인터페이스 멤버를 직접 액세스하기 위한 COM 항목을 나타냅니다.

그림 41.1 이중 인터페이스 VTable



dispatch 인터페이스

Automation 컨트롤러는 COM *IDispatch* 인터페이스를 사용하여 COM 서버 객체를 액세스하는 클라이언트입니다. Automation 컨트롤러는 우선 객체를 만든 다음 객체의 *IUnknown* 인터페이스를 쿼리하여 *IDispatch* 인터페이스 포인터에 대한 인터페이스를 찾습니다. *IDispatch*는 인터페이스 멤버에 대한 고유한 식별 번호인 디스패치 식별자(dispID)로 메소드와 속성을 내부적으로 추적합니다. 컨트롤러는 *IDispatch*를 통해 객체의 dispatch 인터페이스에 대한 타입 정보를 가져온 다음 인터페이스 멤버 이름을 특정 dispID에 매핑합니다. 이 dispID는 런타임에 사용할 수 있으며 컨트롤러는 *IDispatch* 메소드인 *GetIDsOfNames*를 호출하여 dispID를 가져옵니다.

컨트롤러는 dispID를 가져온 다음에 해당 코드(속성 또는 메소드)를 실행하기 위한 *IDispatch* 메소드인 *Invoke*를 호출하여 속성 또는 메소드의 매개변수를 *Invoke* 매개변수 중 하나에 패키지화할 수 있습니다. *Invoke*에는 인터페이스 메소드를 호출할 때 여러 인수를 받아 들일 수 있게 하는 고정된 컴파일 타임 시그너처가 있습니다.

그런 다음 Automation 객체의 *Invoke* 메소드 구현으로 매개변수 패키지를 풀고 속성 또는 메소드를 호출하여 발생하는 오류를 처리할 준비를 합니다. 속성 또는 메소드가 반환되면 객체는 반환 값을 다시 컨트롤러에 전달합니다.

이런 연결은 컨트롤러가 컴파일 타임이 아닌 런타임에 속성 또는 메소드에 연결하므로 후기 연결이라고 합니다.

참고 C++Builder에서는 이중 인터페이스가 아닌 dispatch 인터페이스로 CoClass를 만들 수 없습니다. 그 이유는 C++Builder의 COM 지원이 ATL에 기반하여 이중이 아닌 dispatch 인터페이스를 지원하지 않기 때문입니다.

참고 타입 라이브러리를 임포트하면 C++Builder는 코드를 생성할 때 dispID를 쿼리합니다. 따라서 생성된 랩퍼 클래스가 *GetIDsOfNames*를 호출하지 않고도 *Invoke*를 호출할 수 있습니다. 이 때문에 컨트롤러의 런타임 성능이 상당히 증가할 수 있습니다.

사용자 정의 인터페이스

사용자 정의 인터페이스는 클라이언트가 Vtable에서의 인터페이스 메소드 순서와 인수 타입에 대한 지식을 기반으로 인터페이스 메소드를 호출할 수 있게 하는 사용자 정의 인터페이스입니다. Vtable은 객체가 지원하는 인터페이스의 멤버 함수를 포함하여 객체의 멤버인 모든 속성 및 메소드의 주소를 나열합니다. 객체가 *IDispatch*를 지원하지 않으면 객체의 사용자 정의 인터페이스 멤버 항목은 곧바로 *IUnknown*의 멤버와 같게 됩니다.

객체에 타입 라이브러리가 있으면 Type Library Editor를 사용하여 가져올 수 있는 VTable 레이아웃을 통해 사용자 정의 인터페이스를 액세스할 수 있습니다. 객체가 타입 라이브러리를 가지고 *IDispatch*도 지원하면 클라이언트가 *IDispatch* 인터페이스의 dispID를 가져와서 VTable 오프셋에 직접 연결할 수도 있습니다. C++Builder의 타입 라이브러리 임포터(TLIBIMP)는 임포트할 때 dispID를 검색하므로 dispinterface 랩퍼를 사용하는 클라이언트가 *GetIDsOfNames*를 호출하지 않아도 됩니다. 이 정보는 이미 _TLB 유닛에 들어 있습니다. 그러나 클라이언트는 *Invoke*를 호출해야 합니다.

데이터 마샬링

Out-of-process 및 원격 서버에서는 COM이 현재 프로세스 바깥에 있는 데이터를 마샬링하는 방법에 대해 고려해야 합니다. 다음과 같은 마샬링 방법을 제공할 수 있습니다.

- *IDispatch* 인터페이스를 사용하여 자동으로 마샬링합니다.
- 서버에 타입 라이브러리를 만들고 인터페이스를 OLE Automation 플래그로 표시하여 자동으로 마샬링합니다. COM은 타입 라이브러리에 있는 모든 **Automation과 호환 가능한** 타입을 마샬링하는 방법에 대해 알고 있으므로 대리자와 스텁을 설정할 수 있습니다. 일부 타입은 자동 마샬링을 사용하는 데 제한이 따릅니다.
- 수동으로 *IMarshal* 인터페이스의 모든 메소드를 구현합니다. 이것을 **사용자 정의 마샬링**이라고 합니다.

참고 첫 번째 방법(*IDispatch* 사용)은 Automation 서버에서만 사용할 수 있습니다. 두 번째 방법은 마법사에서 만들어지고 타입 라이브러리를 사용하는 모든 객체에서 자동으로 사용할 수 있습니다.

Automation과 호환 가능한 타입

이중 및 dispatch 인터페이스에서 선언된 메소드의 매개변수 타입과 함수 결과 및 OLE Automation으로 표시한 인터페이스는 *Automation과 호환 가능한* 타입이어야 합니다. OLE Automation과 호환 가능한 타입은 다음과 같습니다.

- 미리 정의된 유효한 타입(예: *short*, *int*, *single*, *double*, *WideString*). 전체 리스트를 보려면 39-11페이지의 "유효한 타입"을 참조하십시오.

- 타입 라이브러리에 정의된 열거 타입. OLE Automation과 호환 가능한 열거 타입은 32비트 값으로 저장되며 매개변수 전달을 위해 *Integer* 타입 값으로 처리됩니다.
- 타입 라이브러리에 정의된 인터페이스 타입으로 OLE Automation에서 안전한 즉, *IDispatch*에서 파생되고 OLE Automation과 호환 가능한 타입만 포함하는 인터페이스 타입
- 타입 라이브러리에 정의된 dispinterface 타입
- 타입 라이브러리 내에 정의된 사용자 정의 레코드 타입
- *IFont*, *IStrings* 및 *IPicture*. Helper 객체는 다음과 같이 매핑되도록 인스턴스화해야 합니다.
  - *Ifont*를 *TFont*로 매핑
  - *IStrings*를 *TStrings*로 매핑
  - *IPicture*를 *TPicture*로 매핑

  ActiveX 컨트롤과 ActiveForm 마법사는 필요한 경우 이 helper 객체를 자동으로 만듭니다. helper 객체를 사용하려면 *GetOleFont*, *GetOleStrings*, *GetOlePicture* 등의 전역 루틴을 각각 호출합니다.

자동 마샬링에 대한 타입 제한

자동 마샬링(Automation 마샬링 또는 타입 라이브러리 마샬링이라고도 함)을 지원하는 인터페이스에 대해 다음 제한 사항이 적용됩니다. Type library Editor를 사용하여 객체를 편집할 때 에디터는 다음과 같은 제한 사항을 강제로 적용합니다.

- 타입이 크로스 플랫폼 통신과 호환되어야 합니다. 예를 들어 데이터 구조(다른 속성 객체를 구현하는 경우 제외), 부호 없는 인수, AnsiString 등을 사용할 수 없습니다.
- 문자열 데이터 타입은 BSTR로 전송해야 합니다. PChar 및 AnsiString은 안전하게 마샬링될 수 없습니다.
- 이중 인터페이스의 모든 멤버는 HRESULT를 함수의 반환 값으로 전달해야 합니다.
- 다른 값을 반환해야 하는 이중 인터페이스 멤버는 이 매개변수를 **var** 또는 **out**으로 지정하여 함수 값을 반환하는 출력 매개변수임을 표시해야 합니다.

참고 Automation 타입 제한 사항을 무시하는 한 가지 방법은 각각의 *IDispatch* 인터페이스와 사용자 정의 인터페이스를 구현하는 것입니다. 이렇게 하면 모든 범위의 사용 가능한 인수 타입을 사용할 수 있습니다. 이것은 Automation 컨트롤러가 여전히 액세스할 수 있는 사용자 정의 인터페이스를 COM 클라이언트가 사용할지를 선택할 수 있다는 것을 의미합니다. 그렇지만 이 경우에는 마샬링 코드를 수동으로 구현해야 합니다.

사용자 정의 마샬링

일반적으로 out-of-process 서버 및 원격 서버에서는 자동 마샬링을 사용하는데, 이는 COM에서 작업을 수행하는 것이 더 쉽기 때문입니다. 그러나 마샬링 성능을 향상시킬 수 있다고 생각한다면 사용자 정의 마샬링을 제공할 수도 있습니다. 새로운 사용자 정의 마샬링을 구현하는 경우에는 *IMarshal* 인터페이스를 지원해야 합니다. 이 방법에 대한 자세한 내용은 Microsoft 문서를 참조하십시오.

COM 객체 등록

서버 객체를 in-process 또는 out-of-process 서버로 등록할 수 있습니다. 서버 타입에 대한 자세한 내용은 38-6페이지의 "in-process, out-of-process 및 원격 서버"를 참조하십시오.

참고 시스템에서 COM 객체를 제거하기 전에 그 객체의 등록을 취소해야 합니다.

in-process 서버 등록

다음과 같은 방법으로 in-process 서버(DLL 또는 OCX)를 등록합니다.

- Run|Register ActiveX Server를 선택합니다.

다음과 같은 방법으로 in-process 서버 등록을 취소합니다.

- Run|Unregister ActiveX Server를 선택합니다.

out-of-process 서버 등록

다음과 같은 방법으로 out-of-process 서버를 등록합니다.

- **/regserver** 명령줄 옵션과 함께 서버를 실행합니다.

  명령줄 옵션은 Run|Parameters 다이얼로그 박스에서 설정할 수 있습니다.

  이 다이얼로그 박스를 실행하여 서버를 등록할 수도 있습니다.

다음과 같은 방법으로 out-of-process 서버 등록을 취소합니다.

- **/unregserver** 명령줄 옵션과 함께 서버를 실행합니다.

다른 방법으로 명령줄에서 **tregsvr** 명령을 사용하거나 운영 체제에서 regsvr32.exe를 실행할 수도 있습니다.

참고 COM 서버를 COM+에서 사용할 계획이라면 COM 서버를 등록하는 대신 COM+ 애플리케이션에 설치해야 합니다. COM+ 애플리케이션에 객체를 설치하면 등록은 자동으로 처리됩니다. COM+ 애플리케이션에 객체를 설치하는 방법에 대한 자세한 내용은 44-27페이지의 "트랜잭션 객체 설치"를 참조하십시오.

애플리케이션 테스트 및 디버깅

다음과 같은 방법으로 COM 서버 애플리케이션을 테스트하고 디버깅합니다.

1. 필요한 경우 Project|Options 다이얼로그 박스의 Compiler 탭을 사용하여 디버깅 정보를 선택합니다. 또한 Tools|Debugger Options 다이얼로그 박스에서 Integrated Debugging을 선택합니다.
2. in-process 서버인 경우 Run|Parameters를 선택하고 Host Application 박스에 Automation 컨트롤러 이름을 입력하고 OK를 선택합니다.
3. Run|Run을 선택합니다.
4. Automation 서버에서 브레이크포인트를 설정합니다.
5. Automation 컨트롤러를 사용하여 Automation 서버와 상호 작용합니다.

브레이크포인트에 도달하면 Automation 서버가 일시 정지합니다.

애플리케이션이 인터페이스에 하는 호출을 추적하는 것도 유용할 수 있습니다. COM 호출의 흐름을 검사함으로써 애플리케이션이 제대로 동작하는지 여부를 결정할 수 있습니다. COM 인터페이스를 호출할 때마다 C++Builder가 메시지를 이벤트 로그에 추가하도록 하려면 Project Options 다이얼로그 박스의 ATL 페이지를 사용하여 디버깅 옵션을 지정합니다.

참고 Automation 컨트롤러를 작성하는 경우 COM cross-process 지원을 활성화하여 in-process 서버를 디버깅할 수도 있습니다. Tools|Debugger Options 다이얼로그 박스의 General 페이지를 사용하여 cross-process 지원을 활성화합니다.

장

42

Active Server Page 생성

Microsoft Internet Information Server(IIS) 환경을 사용하여 웹 페이지를 제공하는 경우에는 Active Server Page(ASP)를 사용하여 동적인 웹 기반 클라이언트/서버 애플리케이션을 만들 수 있습니다. Active Server Page를 사용하면 서버가 웹 페이지를 로드할 때마다 호출되는 스크립트를 작성할 수 있습니다. 이 스크립트는 Automation 객체를 호출하여 HTML 페이지에 포함된 정보를 얻을 수 있습니다. 예를 들어, 비트맵을 만들거나 데이터베이스에 연결하는 C++Builder Automation 서버를 작성하고 이 컨트롤을 사용하여 서버가 웹 페이지를 로드할 때마다 업데이트되는 데이터에 액세스할 수 있습니다.

클라이언트사이드에서 ASP는 표준 HTML 문서처럼 사용할 수 있으며 사용자가 어떤 플랫폼에서나 웹 브라우저를 사용하여 볼 수 있습니다.

ASP 애플리케이션은 C++Builder의 Web Broker 기술을 사용하여 작성하는 애플리케이션과 비슷합니다. Web Broker 기술에 대한 자세한 내용은 32장, "인터넷 서버 애플리케이션 생성"을 참조하십시오. 그러나, ASP는 UI 디자인을 비즈니스 룰 구현이나 복잡한 애플리케이션 로직과 구분한다는 점이 다릅니다.

- UI 디자인은 Active Server Page에 의해 관리됩니다. ASP는 기본적으로 HTML 문서지만 Active Server 객체를 호출하는 스크립트를 포함시켜 비즈니스 룰이나 애플리케이션 로직을 반영하는 컨텐트를 제공할 수 있습니다.
- 애플리케이션 로직은 간단한 메소드를 Active Server Page에 노출하는 Active Server 객체에 의해 캡슐화되어 필요한 컨텐트를 제공합니다.

참고 ASP는 UI 디자인을 애플리케이션 로직과 구분하는 이점을 제공하지만 성능은 규모에 따라 제한됩니다. 매우 많은 수의 클라이언트에 응답하는 웹 사이트인 경우에는 Web Broker 기술을 사용한 방식을 권장합니다.

Active Server Page에 있는 스크립트와 Active Server Page에 포함된 Automation 객체는 현재 애플리케이션에 대한 정보, 브라우저의 HTTP 메시지 등을 제공하는 기본 객체인 ASP 내장 함수를 사용할 수 있습니다.

이 장에서는 C++Builder Active Server Object 마법사를 사용하여 Active Server Object를 만드는 방법을 설명합니다. 그런 다음 Active Server Page에서 이 특별한 Automation 컨트롤을 호출할 수 있으며 내용을 제공할 수 있습니다.

다음과 같은 방법으로 Active Server Object를 만듭니다.

- 애플리케이션을 위한 Active Server Object를 만듭니다.
- Active Server Object의 인터페이스를 정의합니다.
- Active Server Object를 등록합니다.
- 애플리케이션을 테스트하고 디버깅합니다.

Active Server Object 생성

Active Server Object는 전체 ASP 애플리케이션에 대한 정보와 브라우저와의 통신에 사용하는 HTTP 메시지에 대한 액세스 권한을 가지는 Automation 객체입니다. Active Server Object는 ATL 기본 클래스인 *CComObjectRootEx* 및 *CComCoClass*와 *TASPObject* 또는 *TMTSASPObject*의 자손이며 Automation 프로토콜을 지원하여 다른 애플리케이션이나 Active Server Page의 스크립트에서 사용할 수 있도록 합니다. Active Server Object 마법사를 사용하여 Active Server Object를 만듭니다.

Active Server Object 프로젝트는 필요에 따라 실행 파일(exe) 또는 라이브러리 파일(dll) 중 한 가지일 수 있습니다. 그러나 out-of-process 서버 사용의 단점을 알아야 합니다. 이러한 단점은 42-7페이지의 "in-process 또는 out-of-process 서버를 위한 ASP 생성"에 설명되어 있습니다.

다음과 같은 방법으로 Active Server Object 마법사를 표시합니다.

1. File | New | Other를 선택합니다.
2. ActiveX 탭을 선택합니다.
3. Active Server Object 아이콘을 더블 클릭합니다.

마법사에서 새 Active Server Object에 이름을 지정하고 지원할 스레드 모델을 지정합니다. 지정한 모델에 맞도록(예를 들면 스레드 충돌을 피하도록) 구현을 작성합니다. 스레드 모델과 관련된 선택 사항은 다른 COM 객체의 경우와 같습니다. 자세한 내용은 41-5페이지의 "스레드 모델 선택"을 참조하십시오.

Active Server Object의 고유한 특성은 Active Server 페이지와 클라이언트 웹 브라우저 사이에 전달되는 ASP 애플리케이션과 HTTP 메시지에 대한 정보에 액세스하는 능력입니다. 이 정보는 ASP 내장 함수를 사용하여 액세스합니다. 마법사에서 Active Server Type을 설정하여 객체의 액세스 방법을 지정할 수 있습니다.

- IIS 3 또는 IIS 4를 사용하는 경우에는 Page Level Event Method를 사용합니다. 이 모델에서는 객체가 Active Server 페이지를 로드하고 언로드할 때 호출되는 *OnStartPage* 메소드와 *OnEndPage* 메소드를 구현합니다. 객체가 로드될 때 ASP 내장 함수에 액세스하는 데 사용하는 *IScriptingContext* 인터페이스를 자동으로 얻게 됩니다. 이러한 인터페이스는 기본 클래스에서 상속한 속성으로 나타납니다(*TASPObject*).

- IIS5 이상을 사용하는 경우에는 Object Context 타입을 사용합니다. 이 모델에서는 객체가 ASP 내장 함수에 액세스하는 데 사용하는 *IObjectContext* 인터페이스를 가져옵니다. 역시 이러한 인터페이스는 상속한 기본 클래스에서 속성으로 나타납니다(*TMTSASPObject*). 후자의 방법이 가지는 이점 중 한 가지는 객체가 *IObjectContext*를 통해 사용할 수 있는 다른 모든 서비스에 대해 액세스 권한을 가진다는 점입니다. *IObjectContext* 인터페이스에 액세스하려면 Active Server Object의 *ObjectContext* 속성을 사용하십시오. *IObjectContext*를 통해 사용할 수 있는 서비스에 대한 자세한 정보는 44장, "MTS 객체 또는 COM+ 객체 생성"을 참조하십시오.

마법사에서 간단한 ASP 페이지를 작성하여 새로운 Active Server Object를 호스팅하도록 할 수 있습니다. 작성된 페이지는 ProgID를 사용하여 Active Server Object를 만드는 최소한의 VBScript로 작성된 스크립트를 제공하며 메소드를 호출할 수 있는 곳을 나타냅니다. 이 스크립트는 **Server.CreateObject**를 호출하여 Active Server Object를 시작합니다.

참고 작성된 테스트 스크립트는 VBScript를 사용하지만 Jscript를 사용하여 Active Server Page를 작성할 수도 있습니다.

마법사를 종료하면 Active Server Object를 위한 정의를 포함하는 현재 프로젝트에 새 유닛이 추가됩니다. 그 외에 마법사는 타입 라이브러리 프로젝트를 추가하고 Type Library Editor를 엽니다. 이제 41-9페이지의 "COM 객체 인터페이스 정의"에 설명되어 있는 타입 라이브러리를 통해 인터페이스의 속성과 메소드를 노출할 수 있습니다. 객체의 속성과 메소드의 구현을 작성할 때 아래에서 설명하는 ASP 내장 함수를 이용하여 브라우저와 통신하는 데 사용하는 ASP 애플리케이션과 HTTP 메시지에 대한 정보를 얻을 수 있습니다.

Active Server Object는 다른 Automation 객체와 마찬가지로 Vtable 인터페이스를 통한 컴파일 타임 연결과 *IDispatch* 인터페이스를 통한 런타임 연결을 모두 지원하는 **이중 인터페이스**를 구현합니다. 이중 인터페이스에 대한 자세한 내용은 41-12페이지의 "이중 인터페이스"를 참조하십시오.

ASP 내장 함수 사용

ASP 내장 함수는 ASP가 Active Server Page에서 실행되는 객체에 제공하는 일련의 COM 객체입니다. Active Server Object는 ASP 내장 함수를 사용하여 동일한 ASP 애플리케이션에 속하는 Active Server Object 간의 공유 정보를 저장하는 장소 뿐 아니라 애플리케이션과 웹 브라우저 사이에 전달되는 메시지를 반영하는 정보에 액세스할 수 있습니다.

이러한 객체를 액세스하기 쉽게 만들려면 Active Server Object를 위한 기본 클래스에서 객체를 속성으로 표시하십시오. 이 객체에 대한 자세한 내용은 Microsoft 문서를 참조하십시오. 다음 부분에서는 간략한 개요를 제공합니다.

Application 객체

Application 객체에는 *IApplicationObject* 인터페이스를 통해 액세스합니다. 이 객체는 전체 ASP 애플리케이션을 나타내며 가상 디렉토리와 그 하위 디렉토리에 있는 모든 .asp 파일의 집합으로 정의됩니다. Application 객체는 여러 클라이언트에서 공유할 수 있으므로 스레드 충돌을 방지하는 데 사용되는 잠금 지원을 포함합니다.

*IApplicationObject*는 다음을 포함합니다.

표 42.1 IApplicationObject 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
Contents 속성	스크립트 명령을 사용하여 애플리케이션에 추가된 모든 객체를 나열합니다. 이 인터페이스는 리스트에서 객체를 한 개 또는 모두 삭제하는 데 사용할 수 있는 *Remove*와 *RemoveAll*의 두 가지 메소드를 가집니다.
StaticObjects 속성	<OBJECT> 태그를 사용하여 애플리케이션에 추가된 모든 객체를 나열합니다.
Lock 메소드	Unlock 메소드를 호출할 때까지 다른 클라이언트가 Application 객체를 잠그는 것을 방지합니다. 모든 클라이언트는 속성과 같은 공유 메모리에 액세스하기 전에 Lock 메소드를 호출해야 합니다.
Unlock 메소드	Lock 메소드를 사용하여 설정한 잠금을 해제합니다.
Application_OnEnd 이벤트	Session_OnEnd 이벤트 후에 애플리케이션을 종료할 때 발생합니다. 사용할 수 있는 내장 함수는 Application과 Server입니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성해야 합니다.
Application_OnStart 이벤트	Session_OnStart 이벤트 전, 새 세션이 만들어지기 전에 발생합니다. 사용할 수 있는 내장 함수는 Application과 Server입니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성해야 합니다.

Request

Request 객체는 *IRequest* 인터페이스를 통해 액세스합니다. 이 객체는 Active Server Page를 여는 HTTP 요청 메시지에 대한 정보를 제공합니다.

*IRequest*는 다음을 포함합니다.

표 42.2 IRequest 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
ClientCertificate 속성	HTTP 메시지와 함께 전송되는 클라이언트 인증서에 있는 모든 필드의 값을 나타냅니다.
Cookies 속성	HTTP 메시지의 모든 Cookie 헤더의 값을 나타냅니다.
Form 속성	HTTP 바디에 있는 폼 요소의 값을 나타냅니다. 이름으로 액세스할 수 있습니다.
QueryString 속성	HTTP 헤더의 쿼리 문자열에 있는 모든 변수의 값을 나타냅니다.
ServerVariables 속성	여러 환경 변수의 값을 나타냅니다. 이 변수들은 대부분의 일반적인 HTTP 헤더 변수를 나타냅니다.
TotalBytes 속성	요청 바디의 바이트 수를 나타냅니다. 이 속성은 BinaryRead 메소드에 의해 반환되는 바이트 수의 상한값입니다.
BinaryRead 메소드	Post 메시지의 내용을 읽어들입니다. 이 메소드를 호출하여 읽어들일 최대 바이트 수를 지정합니다. 결과 내용은 바이트의 Variant 배열로 반환됩니다. BinaryRead 메소드를 호출한 후에는 Form 속성을 사용할 수 없습니다.

Response

Response 객체는 *IResponse* 인터페이스를 통해 액세스합니다. 이 객체를 사용하여 클라이언트 브라우저에 반환되는 HTTP 응답 메시지에 대한 정보를 지정합니다.

*IResponse*는 다음을 포함합니다.

표 42.3 IResponse 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
Cookies 속성	HTTP 메시지의 모든 Cookie 헤더 값을 결정합니다.
Buffer 속성	페이지 출력이 버퍼링되는지 여부를 나타냅니다. 페이지 출력이 버퍼링되면 서버에서는 현재 페이지의 모든 서버 스크립트가 처리될 때까지 클라이언트에 응답을 보내지 않습니다.
CacheControl 속성	프록시 서버에서 출력을 응답에 캐싱할 수 있는지 여부를 결정합니다.
Charset 속성	문자 집합의 이름을 컨텐트 타입 헤더에 추가합니다.
ContentType 속성	응답 메시지 바디의 HTTP 컨텐트 타입을 지정합니다.
Expires 속성	만료되기까지 캐싱될 수 있는 응답의 길이를 지정합니다.
ExpiresAbsolute 속성	응답이 만료되는 날짜와 시간을 지정합니다.
IsClientConnected 속성	클라이언트에서 서버와의 연결을 끊었는지 여부를 나타냅니다.
Pics 속성	응답 헤더의 pics 필드 값을 설정합니다.
Status 속성	응답의 상태를 나타냅니다. 이 속성은 HTTP 상태 헤더의 값입니다.
AddHeader 메소드	HTTP 헤더를 지정된 이름과 값을 사용하여 추가합니다.
AppendToLog 메소드	이 요청을 위한 웹 서버 로그 엔트리 끝에 문자열을 추가합니다.
BinaryWrite 메소드	응답 메시지 바디에 해석되지 않은 정보를 기록합니다.
Clear 메소드	버퍼링된 HTML 출력을 삭제합니다.
End 메소드	.asp 파일 처리를 중단하고 현재의 결과를 반환합니다.
Flush 메소드	버퍼링된 출력을 즉시 보냅니다.
Redirect 메소드	리디렉션 응답 메시지를 보내고 클라이언트 브라우저를 다른 URL로 리디렉션합니다.
Write 메소드	현재 HTTP 출력에 변수를 문자열로 기록합니다.

Session

Session 객체는 *ISessionObject* 인터페이스를 통해 액세스합니다. 이 객체를 사용하여 클라이언트와 ASP 애플리케이션의 상호 작용이 지속되는 동안 유지되는 변수를 저장합니다. 즉, 이 변수들은 클라이언트가 ASP 애플리케이션 내의 페이지 사이를 이동할 때 해제되지 않고 클라이언트에서 애플리케이션을 완전히 종료할 때만 해제됩니다.

*ISessionObject*는 다음을 포함합니다.

표 42.4 ISessionObject 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
Contents 속성	<OBJECT> 태그를 사용하여 세션에 추가되었던 모든 객체를 나열합니다. 리스트에 있는 변수는 이름을 사용하여 액세스하거나 Contents 객체의 *Remove* 또는 *RemoveAll* 메소드를 호출하여 값을 삭제할 수 있습니다.
StaticObjects 속성	<OBJECT> 태그를 사용하여 세션에 추가되었던 모든 객체를 나열합니다.
CodePage 속성	기호 매핑을 위해 사용하는 코드 페이지를 지정합니다. 로케일에 따라 서로 다른 코드 페이지를 사용할 수 있습니다.
LCID 속성	문자열 컨텐트를 해석하는 데 사용하는 로케일 식별자를 지정합니다.
SessionID 속성	현재 클라이언트를 위한 세션 식별자를 나타냅니다.
TimeOut 속성	애플리케이션이 종료될 때까지 클라이언트의 요청 또는 새로 고침 없이 세션이 지속되는 시간을 분으로 지정합니다.
Abandon 메소드	세션을 없애고 리소스를 릴리스합니다.
Session_OnEnd 이벤트	세션이 중단되거나 시간이 초과될 때 발생합니다. 사용할 수 있는 내장 함수는 Application, Server 및 Session입니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성되어야 합니다.
Session_OnStart 이벤트	Application_OnStart 이벤트 후이지만 Active Server Page에서 스크립트를 실행하기 전, 서버에서 새로운 세션을 만들 때 발생합니다. 모든 내장 함수를 사용할 수 있습니다. 이벤트 핸들러는 VBScript 또는 JScript로 작성되어야 합니다.

Server

Server 객체는 *IServer* 인터페이스를 통해 액세스합니다. 이 객체는 ASP 애플리케이션을 작성하기 위한 여러 유틸리티를 제공합니다.

*IServer*는 다음을 포함합니다.

표 42.5 IServer 인터페이스 멤버

속성, 메소드 또는 이벤트	의미
ScriptTimeOut 속성	Session 객체의 TimeOut 속성과 같습니다.
CreateObject 메소드	지정된 Active Server Object를 인스턴스화합니다.
Execute 메소드	지정된 .asp 파일에 있는 스크립트를 실행합니다.
GetLastError 메소드	오류 상황을 설명하는 ASPError 객체를 반환합니다.
HTMLEncode 메소드	HTML 헤더에 사용하기 위한 문자열을 인코딩하여 예약 문자를 해당 기호 상수로 대체합니다.
MapPath 메소드	지정된 가상 경로(현재 서버에 대한 절대 경로 또는 현재 페이지에 대한 상대적 경로)를 실제 경로에 매핑합니다.
Transfer 메소드	모든 현재 상태 정보를 처리를 위해 다른 Active Server Page에 보냅니다.
URLEncode 메소드	이스케이프 문자를 비롯한 URL 인코딩 규칙을 지정된 문자열에 적용합니다.

in-process 또는 out-of-process 서버를 위한 ASP 생성

ASP 페이지의 **Server.CreateObject**를 사용하여 필요에 따라 in-process 또는 out-of-process 서버 중 하나를 시작할 수 있습니다. 그러나, in-process 서버를 시작하는 것이 더 일반적입니다.

대부분의 in-process 서버와 달리 in-process 서버의 Active Server Object는 클라이언트의 프로세스 공간에서 실행되지 않습니다. 그 대신 IIS 프로세스 공간에서 실행됩니다. 이는 클라이언트에서 애플리케이션을 다운로드할 필요가 없다는 것을 의미합니다(예를 들어, ActiveX 객체를 사용할 때). in-process 컴포넌트 DLL은 out-of-process 서버보다 더 빠르고 안전하므로 서버 사이드에서 사용하기에 더 적합합니다.

out-of-process 서버는 안전성이 떨어지기 때문에 구성될 IIS에서는 out-of-process 실행 파일을 허용하지 않는 것이 일반적입니다. 이 경우 Active Server Object를 위한 out-of-process 서버를 만들면 다음과 유사한 오류가 발생할 것입니다.

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

또한 out-of-process 컴포넌트는 각 객체 인스턴스에 대해 개별적인 서버 프로세스를 만드는 경우가 많으므로 CGI 애플리케이션보다 느리며, 컴포넌트 DLL만큼 확장되지 않습니다.

성능과 확장성을 우선으로 하는 사이트인 경우에는 in-process 서버를 사용할 것을 권장합니다. 그러나, 통신량이 중간 이하인 인트라넷 사이트에서는 사이트 전체의 성능에 영향을 미치지 않고 out-of-process 컴포넌트를 사용할 수 있습니다.

in-process 및 out-of-process 서버에 대한 일반적인 내용은 38-6페이지의 "in-process, out-of-process 및 원격 서버"를 참조하십시오.

Active Server Object 등록

Active Server Page를 in-process 또는 out-of-process 서버로 등록할 수 있습니다. 그러나, in-process 서버가 더 일반적입니다.

참고 Active Server Page 객체를 시스템에서 제거하려면 먼저 등록을 해제하고 Windows 레지스트리에서 엔트리를 제거합니다.

in-process 서버 등록

다음과 같은 방법으로 in-process 서버(DLL 또는 OCX)를 등록합니다.

- Run|Register ActiveX Server를 선택합니다.

다음과 같은 방법으로 in-process 서버 등록을 해제합니다.

- Run|Unregister ActiveX Server를 선택합니다.

out-of-process 서버 등록

다음과 같은 방법으로 out-of-process 서버를 등록합니다.

- /regserver 명령줄 옵션을 사용하여 서버를 실행합니다 . Run|Parameters 다이얼로그 박스를 사용하여 명령줄 옵션을 설정할 수 있습니다.

  또한 서버를 실행하여 등록할 수도 있습니다.

다음과 같은 방법으로 out-of-process 서버의 등록을 해제합니다.

- /unregserver 명령줄 옵션을 사용하여 서버를 실행합니다.

Active Server Page 애플리케이션 테스트 및 디버깅

Active Server Object와 같은 in-process 서버를 디버깅하는 것은 DLL을 디버깅하는 것과 비슷합니다. DLL을 로드하는 호스트 애플리케이션을 선택하고 디버깅합니다. 다음과 같은 방법으로 Active Server Object를 테스트하고 디버깅합니다.

1. 필요한 경우 Project|Options 다이얼로그 박스의 Compiler 탭을 사용하여 디버깅 정보를 표시합니다. 또한 Tools|Debugger Options 다이얼로그 박스에서 Integrated Debugging을 표시합니다.
2. Run|Parameters를 선택하고 Host Application 박스에 웹 서버 이름을 입력한 다음 OK를 선택합니다.
3. Run|Run을 선택합니다.
4. Active Server Object 구현에 브레이크포인트를 설정합니다.
5. 웹 브라우저를 사용하여 Active Server Page와 상호 작용을 합니다.

디버거는 브레이크포인트에 도달하면 일시 중지됩니다.

장

43

ActiveX 컨트롤 생성

ActiveX 컨트롤은 C++Builder, Delphi, Visual Basic, Internet Explorer, Netscape Navigator 같은 호스트 애플리케이션의 기능을 통합하고 확장하는 소프트웨어 컴포넌트입니다. ActiveX 컨트롤은 이러한 통합을 위한 특정한 인터페이스 집합을 구현합니다.

예를 들면, C++Builder에는 차트 컨트롤, 스프레드시트 컨트롤, 그래픽 컨트롤 등의 몇 가지 ActiveX 컨트롤이 포함되어 있습니다. 이 컨트롤들은 IDE의 컴포넌트 팔레트에 추가한 다음 폼에 가져다 놓고 Object Inspector를 사용하여 속성을 설정하면 기본 VCL 컴포넌트처럼 사용할 수 있습니다.

또한 ActiveX 컨트롤을 웹에 배포하여 HTML 문서에서 참조하고 ActiveX 사용 가능 웹 브라우저를 사용하여 볼 수 있습니다.

C++Builder에는 다음 두 가지 유형의 ActiveX 컨트롤을 만드는 마법사가 포함되어 있습니다.

- **VCL 클래스를 랩핑하는 ActiveX 컨트롤.** VCL 클래스를 랩핑함으로써 기존 컴포넌트를 ActiveX 컨트롤로 변환하거나 새 컴포넌트를 만들어 로컬에서 테스트한 다음 ActiveX 컨트롤로 변환할 수 있습니다. ActiveX 컨트롤은 일반적으로 크기가 큰 편인 호스트 애플리케이션에 포함시키기 위한 것입니다.
- **Active 폼.** Active 폼은 폼 디자이너를 사용하여 다이얼로그 박스나 완전한 애플리케이션처럼 작동하는 보다 정교한 컨트롤을 만들 수 있도록 해줍니다. Active 폼은 일반적인 C++Builder 애플리케이션을 만드는 것과 거의 동일한 방법으로 만들며, 주로 웹에 배포하기 위한 것입니다.

이 장에서는 C++Builder 환경에서 ActiveX 컨트롤을 만드는 방법을 개략적으로 설명합니다. 마법사를 사용하지 않고 ActiveX 컨트롤을 만드는 자세한 구현 방법은 MSDN(Microsoft Developer's Network) 설명서를 참조하거나 Microsoft 웹 사이트의 ActiveX 정보를 참조하십시오.

ActiveX 컨트롤 생성 개요

C++Builder 를 사용하여 ActiveX 컨트롤을 만드는 방법은 일반적인 컨트롤이나 폼을 만드는 것과 매우 비슷하며, 객체의 인터페이스를 정의한 다음 구현을 완료하는 다른 COM 객체 생성 방법과 매우 다릅니다. Active 폼이 아닌 ActiveX 컨트롤을 만들려면 이 과정을 역으로 수행하여 VCL 컨트롤을 구현하는 것에서 시작하여 일단 컨트롤을 만든 후 인터페이스와 타입 라이브러리를 만듭니다. Active 폼을 만들 때는 인터페이스와 타입 라이브러리가 폼과 동시에 만들어지며 그 다음에 폼 디자이너를 사용하여 폼을 구현합니다.

완성된 ActiveX 컨트롤은 원본으로 사용하는 구현을 제공하는 VCL 컨트롤, VCL 컨트롤을 랩핑하는 COM 객체, COM 객체의 속성, 메소드 및 이벤트 리스트를 포함한 타입 라이브러리로 구성됩니다.

Active 폼 이외의 새 ActiveX 컨트롤을 만들려면 다음 단계를 따르십시오.

1. ActiveX 컨트롤의 기반을 형성하는 사용자 정의 VCL 컨트롤을 디자인하고 만듭니다.
2. ActiveX 컨트롤 마법사를 사용하여 단계 1에서 만든 VCL 컨트롤에서 ActiveX 컨트롤을 만듭니다.
3. ActiveX Property Page 마법사를 사용하여 컨트롤을 위한 한 개 이상의 페이지를 만듭니다(선택 사항).
4. 속성 페이지를 ActiveX 컨트롤과 연관시킵니다(선택 사항).
5. 컨트롤을 등록합니다.
6. 모든 잠재적인 대상 애플리케이션을 사용하여 컨트롤을 테스트합니다.
7. 웹에 ActiveX 컨트롤을 배포합니다(선택 사항).

새 Active 폼을 만들려면, 다음 단계를 따르십시오.

1. ActiveForm 마법사를 사용하여 IDE에 비어 있는 폼으로 표시되는 Active 폼과 폼에 연결되는 ActiveX 랩퍼를 만듭니다.
2. 폼 디자이너를 사용하여 Active 폼에 컴포넌트를 추가하고 폼 디자이너를 사용하여 일반적인 폼을 만들고 구현하는 것과 같은 방법으로 구현합니다.
3. 위의 단계 3부터 단계 7까지 수행하여 Active 폼을 위한 속성 페이지를 만든 다음 Active 폼을 등록하고 웹에 배포합니다.

ActiveX 컨트롤의 요소

ActiveX 컨트롤에는 특정한 기능을 수행하는 많은 요소들이 포함되어 있습니다. ActiveX 컨트롤 요소로는 속성, 메소드 및 이벤트 리스트를 노출하는 COM 객체 랩퍼, VCL 컨트롤 및 하나 이상의 연결된 타입 라이브러리가 있습니다.

VCL 컨트롤

C++Builder에서 ActiveX 컨트롤의 원본으로 사용하는 구현은 VCL 컨트롤입니다. ActiveX 컨트롤을 만들 때는 먼저 ActiveX 컨트롤을 만들 VCL 컨트롤을 디자인하거나 선택해야 합니다.

원본으로 사용하는 VCL 컨트롤은 호스트 애플리케이션이 부모일 수 있는 윈도우가 있어야 하기 때문에 *TWinControl*의 자손이라야 합니다. Active 폼을 만들 때 이 객체는 *TActiveForm*의 자손입니다.

참고 ActiveX 컨트롤 마법사는 ActiveX 컨트롤을 만들도록 선택할 수 있는 *TWinControl* 자손 리스트를 제공합니다. 그러나, 모든 *TWinControl* 자손이 이 리스트에 포함되는 것은 아닙니다. *THeaderControl*와 같은 일부 컨트롤은 *RegisterNonActiveX* 프로시저를 사용하여 ActiveX와 호환되지 않는 것으로 등록되고 리스트에 포함하지 않습니다.

ActiveX 랩퍼(ActiveX wrapper)

실제 COM 객체는 VCL 컨트롤을 위한 ActiveX 랩퍼 객체입니다. 이 객체의 이름은 *TVCLClass*XImpl의 형태를 가집니다. 여기서 *TVCLClass*는 VCL 컨트롤 클래스의 이름입니다. 예를 들면, *TButton*을 위한 ActiveX 랩퍼의 이름은 *TButtonXImpl*일 것입니다.

랩퍼 클래스는 ActiveX 인터페이스에 대한 지원을 제공하는 VCLCONTROL_IMPL 매크로에 의해 선언된 클래스의 자손입니다. 이러한 지원은 ActiveX 랩퍼에 상속되므로 ActiveX 랩퍼는 Windows 메시지를 VCL 컨트롤에게 전송하고 호스트 애플리케이션에 있는 윈도우의 부모가 될 수 있습니다.

ActiveX 랩퍼는 디폴트 인터페이스를 통해 VCL 컨트롤 속성과 메소드를 클라이언트에 노출합니다. 마법사는 원본으로 사용하는 VCL 컨트롤에 메소드 호출을 위임하여 대부분의 랩퍼 클래스 속성 및 메소드를 자동으로 구현합니다. 또한 마법사는 클라이언트에서 VCL 컨트롤의 이벤트를 발생시키는 메소드를 랩퍼 클래스에 제공하며 이 메소드를 VCL 컨트롤 상의 이벤트 핸들러로 할당합니다.

타입 라이브러리

ActiveX 컨트롤 마법사는 랩퍼 클래스의 타입 정의, 디폴트 인터페이스 및 필요한 타입 정의를 포함하는 타입 라이브러리를 자동으로 생성합니다. 이 타입 정보는 컨트롤의 서비스 내용을 호스트 애플리케이션에 알리는 방법을 제공합니다. 이 정보는 Type Library Editor를 사용하여 보고 편집할 수 있습니다. 이 정보는 확장자가 .TLB인 독립적인 바이너리 타입 라이브러리 파일에 저장되지만, 또한 리소스로서 ActiveX 컨트롤 DLL에 자동으로 컴파일됩니다.

속성 페이지

ActiveX 컨트롤에 속성 페이지를 옵션으로 부여할 수 있습니다. 호스트(클라이언트) 애플리케이션 사용자는 속성 페이지를 사용하여 컨트롤의 속성을 보고 편집할 수 있습니다. 한 페이지에 몇 개의 속성을 그룹화하거나, 한 페이지를 사용하여 한 가지 속성을 위한 다이얼로그 박스와 유사한 인터페이스를 제공할 수 있습니다. 속성 페이지를 만드는 방법에 대한 자세한 내용은 43-13페이지의 "ActiveX 컨트롤을 위한 속성 페이지 생성"을 참조하십시오.

ActiveX 컨트롤 디자인

ActiveX 컨트롤을 디자인할 때는 사용자 정의 VCL 컨트롤을 만드는 작업부터 시작합니다. 사용자 정의 VCL 컨트롤은 ActiveX 컨트롤의 기반이 됩니다. 사용자 정의 컨트롤 생성에 대한 자세한 내용은 V부, "사용자 정의 컴포넌트 생성"을 참조하십시오.

VCL 컨트롤을 디자인할 때는 VCL 컨트롤 자체가 애플리케이션이 아니라 다른 애플리케이션에 포함될 것이라는 점을 염두에 어야 합니다. 따라서 정교한 다이얼로그 박스나 그 밖의 주요 사용자 인터페이스 컴포넌트를 사용하기보다는 주 애플리케이션의 규칙을 따르며 그 안에서 사용할 간단한 컨트롤을 만드는 것이 목표입니다.

또한 ActiveX 컨트롤의 인터페이스는 *IDispatch*를 지원해야 하므로 객체가 클라이언트에 노출하는 모든 속성과 메소드의 타입은 Automation과 호환되어야 합니다. 마법사는 Automation 호환 타입이 아닌 매개변수를 가진 랩퍼 클래스의 인터페이스에는 메소드를 추가하지 않습니다. Automation 호환 타입 리스트는 39-11페이지의 "유효한 타입"을 참조하십시오.

마법사에서는 COM 랩퍼 클래스를 사용하여 필요한 모든 ActiveX 인터페이스를 구현합니다. 또한 모든 Automation 호환 속성, 메소드 및 이벤트를 랩퍼 클래스의 디폴트 인터페이스를 통해 표면화합니다. 일단 마법사에서 COM 랩퍼 클래스와 인터페이스가 만들어진 다음에는 Type Library Editor를 사용하여 디폴트 인터페이스를 수정하거나 추가 인터페이스를 구현하여 랩퍼 클래스를 증가시킬 수 있습니다.

VCL 컨트롤에서 ActiveX 컨트롤 생성

VCL 컨트롤에서 ActiveX 컨트롤을 생성하려면 ActiveX 컨트롤 마법사를 사용합니다. VCL 컨트롤의 속성, 메소드 및 이벤트는 ActiveX 컨트롤의 속성, 메소드 및 이벤트가 됩니다.

ActiveX 컨트롤 마법사를 사용하기 전에 어떤 VCL 컨트롤이 생성된 ActiveX 컨트롤의 원본으로 사용하는 구현을 제공할 것인지 결정해야 합니다.

다음과 같은 방법으로 ActiveX 컨트롤 마법사를 시작합니다.

1. File|New|Other를 선택하여 New Items 다이얼로그 박스를 엽니다.
2. ActiveX 탭을 선택합니다.
3. ActiveX Control 아이콘을 더블 클릭합니다.

마법사에서 새 ActiveX 컨트롤에 의해 랩핑될 VCL 컨트롤의 이름을 선택합니다. *RegisterNonActiveX* 프로시저를 사용하여 ActiveX와 비호환으로 등록되지 않은 *TWinControl*의 자손인 사용 가능한 컨트롤 리스트가 다이얼로그 박스에 표시됩니다.

팁 원하는 컨트롤이 드롭다운 리스트에 없으면 IDE에 설치되었거나 프로젝트에 추가되었는지 확인합니다.

VCL 컨트롤을 선택하면 CoClass 이름, ActiveX 랩퍼 구현 유닛 및 ActiveX 라이브러리 프로젝트가 마법사에서 자동으로 생성됩니다. ActiveX 라이브러리 프로젝트가 열려 있고 COM+ 이벤트 객체가 포함되어 있지 않은 경우에는 현재 프로젝트가 자동으로 사용됩니다. ActiveX 라이브러리 프로젝트가 이미 열려 있어 프로젝트 이름을 수정할 수 없는 경우가 아니라면 마법사에서 이러한 사항을 모두 변경할 수 있습니다.

ActiveX 컨트롤 마법사는 항상 Apartment를 스레드 모델로 지정합니다. ActiveX 프로젝트에 하나의 컨트롤만 포함된 경우에는 문제가 되지 않지만 프로젝트에 객체를 추가하는 경우에는 스레드 지원을 제공해야 합니다.

또한 ActiveX 컨트롤 마법사를 사용하여 다음과 같은 다양한 ActiveX 컨트롤 옵션을 구성할 수 있습니다.

- **라이센스 사용:** 컨트롤 사용자가 라이센스 키를 소유하지 않으면 디자인 목적으로 또는 런타임에 컨트롤을 열 수 없도록 라이센스를 설정할 수 있습니다.
- **버전 정보 포함:** 저작권이나 파일 설명과 같은 버전 정보를 ActiveX 컨트롤에 포함시킬 수 있습니다. 이 정보는 브라우저에서 볼 수 있습니다. Visual Basic 4.0 등 일부 호스트 클라이언트의 경우에는 버전 정보가 있어야 ActiveX 컨트롤을 호스팅합니다. Project|Options를 선택하고 Version Info 페이지를 선택하여 버전 정보를 지정합니다.
- **About 박스 포함:** 마법사가 컨트롤에 대한 About 박스를 구현하는 독립적인 폼을 생성하도록 지시할 수 있습니다. 호스트 애플리케이션 사용자는 개발 환경에서 이 About 박스를 표시할 수 있습니다. 디폴트로, About 박스에는 ActiveX 컨트롤 이름, 이미지, 저작권 정보 및 OK 버튼이 포함됩니다. 마법사에 의해 프로젝트에 추가되는 이 디폴트 폼은 수정이 가능합니다.

ActiveX 컨트롤 마법사를 종료하면 다음이 생성됩니다.

- ActiveX 컨트롤을 시작하는 데 필요한 코드가 포함된 ActiveX Library 프로젝트 파일. 이 파일은 대개 변경하지 않습니다.
- 컨트롤을 위한 CoClass, 클라이언트에 노출하는 인터페이스 및 필요한 타입 정의를 정의하는 타입 라이브러리. 타입 라이브러리에 대한 자세한 내용은 39장, "타입 라이브러리 사용"을 참조하십시오.
- ActiveX 컨트롤을 정의하고 구현하는 ActiveX 구현 유닛. 이 유닛은 VCLCONTROL_IMPL 매크로를 사용하여 Microsoft Active Template Library(ATL)에 맞도록 조정됩니다. 이 ActiveX 컨트롤은 완전한 기능을 수행하는 구현이므로 추가 작업이 필요하지 않습니다. 그러나, ActiveX 컨트롤이 클라이언트에 노출하는 속성, 메소드 및 이벤트를 사용자 정의하기를 원하는 경우에는 이 클래스를 수정할 수 있습니다.
- *ActiveXControlProj*_ATL.cpp (.h) 형식의 이름을 가진 ATL 유닛. 여기서 *ActiveXControlProj*는 프로젝트 이름입니다. 이 유닛은 대부분 ATL 템플릿 클래스를 프로젝트에서 사용할 수 있도록 해주고 생성된 ActiveX 랩퍼가 VCL 객체와 작동할 수 있도록 해주는 클래스를 정의하는 include문으로 이루어집니다. 또한 ATL 클래스에 ActiveX 라이브러리를 나타내는 _Module이라 불리는 전역 변수를 선언합니다.
- 요청한 경우 About 박스 폼과 유닛
- 라이센스 사용을 지정한 경우 .LIC 파일

VCL 폼을 기반으로 ActiveX 컨트롤 생성

ActiveForm은 다른 ActiveX 컨트롤과 달리 ActiveX 랩퍼 클래스에 의해 디자인된 다음 랩핑되지 않습니다. 그 대신 ActiveForm 마법사에서 비어 있는 폼을 만든 다음 나중에 마법사를 종료할 때 폼 디자이너에서 디자인합니다.

ActiveForm을 웹에 배포하는 경우 C++Builder에서는 HTML 페이지를 만들어 ActiveForm에 대한 참조를 포함시키고 페이지 상의 위치를 지정합니다. 그런 다음 웹 브라우저에서 ActiveForm을 표시하고 실행할 수 있습니다. ActiveForm은 브라우저 내에서 독립형 C++Builder 폼과 마찬가지로 사용됩니다. ActiveForm에는 사용자 정의 VCL 컨트롤을 비롯하여 모든 VCL 컴포넌트 또는 ActiveX 컨트롤이 포함될 수 있습니다.

다음과 같은 방법으로 ActiveForm 마법사를 시작합니다.

1 File|New|Other를 선택하여 New Items 다이얼로그 박스를 엽니다.

2 ActiveX 탭을 선택합니다.

3 ActiveForm 아이콘을 더블 클릭합니다.

ActiveForm 마법사는 랩핑할 VCL 클래스 이름을 지정할 수 없다는 점 외에는 ActiveX 컨트롤 마법사와 같습니다. 이는 Active 폼이 항상 *TActiveForm*을 기반으로 하기 때문입니다.

ActiveX 컨트롤 마법사에서와 같이 CoClass, 구현 유닛 및 ActiveX 라이브러리 프로젝트의 디폴트 이름을 변경할 수 있습니다. 또한 이 마법사를 사용하여 Active Form에 대한 라이센스 사용 여부, 버전 정보 포함 여부 및 About 박스 폼 사용 여부를 지정할 수 있습니다.

ActiveForm 마법사를 종료하면 다음과 같은 객체가 생성됩니다:

- ActiveX 컨트롤을 시작하는 데 필요한 코드가 포함된 ActiveX Library 프로젝트 파일. 이 파일은 대개 변경하지 않습니다.
- 컨트롤을 위한 CoClass, 클라이언트에 노출하는 인터페이스 및 필요한 타입 정의를 정의하는 타입 라이브러리. 타입 라이브러리에 대한 자세한 내용은 39장, "타입 라이브러리 사용"을 참조하십시오.
- *TActiveForm*의 자손인 폼. 이 폼은 폼 디자이너에서 표시됩니다. 폼 디자이너에서 이 폼을 사용하여 클라이언트에게 표시하는 Active 폼을 디자인할 수 있습니다. 이 폼의 구현은 생성된 구현 유닛에 표시됩니다.
- 폼을 위한 ActiveX 랩퍼의 선언. 이 랩퍼 클래스는 구현 유닛에도 정의되며 *TActiveFormX*Impl 형식의 이름을 가집니다. 여기서 *TActiveFormX*는 폼 클래스의 이름입니다. 이 ActiveX 랩퍼는 완전한 기능을 수행하는 구현이며 추가 작업이 필요하지 않습니다. 그러나, Active Form이 클라이언트에 노출하는 속성, 메소드 및 이벤트를 사용자 정의하기를 원하는 경우에는 이 클래스를 수정할 수 있습니다.
- *ActiveXControlProj*_ATL.cpp (.h) 형식의 이름을 가지는 ATL 유닛. 여기서 *ActiveXControlProj*는 프로젝트 이름입니다. 이 유닛은 대부분 ATL 템플릿 클래스를 프로젝트에서 사용할 수 있도록 해주고 생성된 ActiveX 랩퍼가 VCL 폼과 함께 작동하도록 하는 클래스를 정의하는 include문으로 이루어집니다. 또한 ATL 클래스에 ActiveX 라이브러리를 나타내는 _Module이라 불리는 전역 변수를 선언합니다.

- 요청한 경우 About 박스 폼 및 유닛
- 라이센스 사용을 지정한 경우 .LIC 파일

이 시점에서 원하는 경우 컨트롤을 추가하고 폼을 디자인할 수 있습니다.

ActiveForm 프로젝트를 디자인하여 확장명이 OCX인 ActiveX 라이브러리로 컴파일한 후 프로젝트를 웹 서버에 배포할 수 있으며 C++Builder 에서는 ActiveForm 에 대한 참조를 포함하는 테스트 HTML 페이지를 만듭니다.

ActiveX 컨트롤 라이센스

ActiveX 컨트롤 라이센스는 디자인타임에 라이센스 키를 제공하고 런타임에 생성된 컨트롤에 대한 동적 라이센스 생성을 지원하는 것으로 이루어집니다.

디자인타임 라이센스를 제공하기 위해 ActiveX 마법사에서는 컨트롤을 위한 키를 만들고 확장명이 LIC인 프로젝트와 동일한 이름의 파일에 저장합니다. 이 .LIC 파일은 프로젝트에 추가됩니다. 컨트롤 사용자는 .LIC 파일 복사본을 가지고 개발 환경에서 컨트롤을 열어야 합니다. Make Control Licensed 체크 박스가 선택된 프로젝트에 있는 컨트롤은 LIC 파일에 각각의 키 항목이 있습니다.

런타임 라이센스를 지원하기 위해 랩퍼 클래스는 *GetLicenseString* 및 *GetLicenseFilename* 두 메소드를 구현합니다. 이 두 메소드는 각각 컨트롤과 .LIC 파일 이름의 라이센스 문자열을 반환합니다. 호스트 애플리케이션에서 ActiveX 컨트롤을 만들 때, 컨트롤을 위한 클래스 팩토리에서 이 두 메소드를 호출하여 *GetLicenseString*에 의해 반환된 문자열을 .LIC 파일에 저장된 문자열과 비교합니다.

Internet Explorer 를 위한 런타임 라이센스인 경우에는 사용자가 웹 페이지의 HTML 소스 코드를 볼 수 있고 ActiveX 컨트롤이 표시되기 전에 사용자의 컴퓨터에 복사되기 때문에 한 층. Internet Explorer에서 사용되는 컨트롤을 위한 런타임 라이센스를 만들려면 먼저 라이센스 패키지 파일(LPK 파일)을 만들고 이 파일을 해당 컨트롤이 있는 HTML 페이지에 포함시켜야 합니다. LPK 파일은 기본적으로 ActiveX 컨트롤 CLSID와 라이센스 키로 이루어진 배열입니다.

참고 LPK 파일은 만들려면 LPK_TOOL.EXE 유틸리티를 사용하십시오. 이 유틸리티는 Microsoft 웹 사이트(www.microsoft.com)에서 다운로드할 수 있습니다.

LPK 파일을 웹 페이지에 포함시키려면 HTML 객체인 <OBJECT>와 <PARAM>을 다음과 같이 사용하십시오.

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
    <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

CLSID는 객체를 라이센스 패키지로 식별하고 PARAM은 HTML 페이지와 관련된 라이센스 패키지 파일의 상대적 위치를 지정합니다.

Internet Explorer에서 컨트롤을 포함한 웹 페이지를 표시할 때 LPK 파일을 분석하여 라이센스 키를 추출하고 라이센스 키가 *GetLicenseString*에 의해 반환된 컨트롤의 라이센스와 일치하면 그 페이지의 컨트롤을 렌더링합니다. 한 웹 페이지에 둘 이상의 LPK가 포함된 경우 Internet Explorer에서는 첫 번째를 제외한 나머지 파일을 무시합니다.

자세한 내용은 Microsoft 웹 사이트의 Licensing ActiveX Controls를 참조하십시오.

ActiveX 컨트롤 인터페이스 사용자 정의

ActiveX 컨트롤 마법사와 ActiveForm 마법사는 ActiveX 랩퍼 클래스를 위한 디폴트 인터페이스를 생성합니다. 이 디폴트 인터페이스는 다음과 같은 예외를 제외하면 단순히 원래 VCL 컨트롤 또는 폼의 속성, 메소드 및 이벤트를 노출합니다.

- 데이터 인식 속성은 표시되지 않습니다. ActiveX 컨트롤을 데이터 인식 컨트롤로 만드는 방법은 VCL 컨트롤과 다르기 때문에 마법사에서는 데이터와 관련된 속성을 변환하지 않습니다. ActiveX 컨트롤을 데이터 인식 컨트롤로 만드는 방법은 43-11페이지의 "타입 라이브러리를 사용하여 간단한 데이터 연결 사용"을 참조하십시오.
- Automation 호환 타입이 아닌 속성, 메소드 또는 이벤트는 표시되지 않습니다. 마법사가 종료된 후 이러한 속성, 메소드 또는 이벤트를 ActiveX 컨트롤의 인터페이스에 추가할 수 있습니다.

39장, "타입 라이브러리 사용"에서 설명한 Type Library Editor를 사용하여 타입 라이브러리를 편집함으로써 ActiveX 컨트롤에서 속성, 메소드 및 이벤트를 추가, 편집 및 제거할 수 있습니다.

참고 ActiveX 컨트롤의 인터페이스에 published가 아닌 속성을 추가할 수 있습니다. 그러한 속성은 런타임에 설정될 수 있으며 개발 환경에 표시되지만 변경 사항은 그대로 유지되지 않습니다. 즉, 컨트롤 사용자가 디자인 타임에 속성의 값을 변경할 경우 그 변경 사항은 컨트롤이 실행될 때 반영되지 않습니다. 이는 ActiveX 컨트롤이 ATL 스트리밍 시스템이 아닌 VCL 스트리밍 시스템을 사용하기 때문입니다. 소스가 VCL 객체이고 속성이 published가 아니면 VCL 객체의 자손을 만들고 그 자손에 속성을 게시하여 속성을 영구적으로 만들 수 있습니다.

또한 VCL 컨트롤의 모든 속성, 메소드 및 이벤트를 호스트 애플리케이션에 노출하지 않도록 선택할 수 있습니다. Type Library Editor를 사용하여 마법사에서 생성된 인터페이스에서 VCL 컨트롤의 속성, 메소드 및 이벤트를 제거할 수 있습니다. Type Library Editor를 사용하여 인터페이스에서 속성과 메소드를 제거할 때 Type Library Editor는 해당 구현 클래스에서는 속성과 메소드를 제거하지 않습니다. 제거하려면 Type Library Editor에서 인터페이스를 변경한 후 구현 유닛에서 ActiveX 랩퍼 클래스를 편집합니다.

경고 VCL 컨트롤 또는 폼에서 ActiveX 컨트롤을 다시 만드는 경우에는 타입 라이브러리에 대한 변경 사항이 손실됩니다.

팁 마법사에서 ActiveX 랩퍼 클래스에 추가하는 메소드를 확인하는 것이 좋습니다. 이렇게 하면 마법사에서 Automation 호환 타입이 아닌 데이터 인식 속성이나 메소드를 빠뜨린 곳을 알 수 있을 뿐 아니라 마법사에서 구현을 생성할 수 없었던 메소드를 감지할 수 있습니다. 그러한 메소드는 구현에서 문제를 나타내는 주석과 함께 표시됩니다.

속성, 메소드 및 이벤트 추가

ActiveX 컨트롤 인터페이스에 속성과 메소드를 추가하는 방법은 그 밖의 다른 COM 인터페이스에 속성, 메소드 및 이벤트를 추가하는 방법과 같습니다. 이벤트를 추가하는 방법은 객체의 디폴트 인터페이스가 아니라 이벤트 인터페이스에 추가한다는 점 외에는 메소드(이벤트 핸들러)를 추가하는 방법과 같습니다. 타입 라이브러리를 사용하여 속성과 이벤트를 추가하는 방법은 41-9페이지의 "COM 객체 인터페이스 정의"에서 설명합니다.

ActiveX 컨트롤의 인터페이스에 추가하는 경우 대개 원본으로 사용하는 VCL 컨트롤에 있는 속성, 메소드 또는 이벤트를 표시하는 것뿐입니다. 다음 항목들은 이러한 작업을 수행하는 방법에 대해 설명합니다.

속성 및 메소드 추가

ActiveX 랩퍼 클래스는 읽기 및 쓰기 액세스 메소드를 사용하여 인터페이스에 속성을 구현합니다. 즉, 랩퍼 클래스는 인터페이스 상에 getter 또는 setter 메소드로 나타나는 COM 속성을 가집니다. COM 속성의 경우 VCL 속성과는 달리 인터페이스 상에 "property" 선언은 없고 속성 액세스 메소드 플래그가 있는 메소드가 포함되어 있습니다. ActiveX 컨트롤의 디폴트 인터페이스에 속성을 추가하면 한 개 또는 두 개의 새 메소드 (getter 또는 setter) 를 Type Library Editor 에 의해 업데이트되는 _TLB 유닛에 표시되는 랩퍼 클래스 정의에 구현해야 합니다. 이는 인터페이스에 메소드를 추가할 때 랩퍼 클래스에 상응하는 메소드를 구현해야 하는 것과 마찬가지입니다. 따라서 랩퍼 클래스의 인터페이스에 속성을 추가하면 메소드를 추가하는 것과 같이, 개발자가 완성할 수 있는 새 스켈레톤 메소드 구현이 랩퍼 클래스 정의에 추가됩니다.

참고 생성된 _TLB 유닛에 포함되는 자세한 내용은 40-5페이지의 "타입 라이브러리 정보를 임포트할 때 생성되는 코드"를 참조하십시오.

예를 들면, 원본으로 사용하는 VCL 객체에 있는 AnsiString 타입의 *Caption* 속성을 Type Library Editor에서 추가하면 C++Builder는 랩퍼 클래스에 다음과 같은 선언을 추가합니다.

```
STDMETHOD(get_Caption(BSTR* Value));
STDMETHOD(set_Caption(BSTR Value));
```

또한 개발자가 완성할 수 있는 스켈레톤 메소드 구현을 추가합니다.

```
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error (e.Message.cstr(), IID_IButtonX);
  }
  return S_OK;
};
```

```
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error (e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

일반적으로 이러한 메소드는 랩퍼 클래스의 *m_VclCtl* 멤버를 사용하여 액세스할 수 있는 연결된 VCL 컨트롤에 위임함으로써 구현할 수 있습니다.

```
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
  try
  {
    *Value = WideString (m_VclCtl->Caption).Copy();
  }
  catch(Exception &e)
  {
    return Error (e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
    m_VclCtl->Caption = AnsiString(Value);
  }
  catch(Exception &e)
  {
    return Error (e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

COM 데이터 타입을 원시 C++ 타입으로 변환하는 코드를 추가해야 하는 경우도 있습니다. 앞의 예제에서는 타입 변환이 사용되었습니다.

이벤트 추가

ActiveX 컨트롤은 Automation 객체가 클라이언트에 이벤트를 발생시키는 것과 같은 방법으로 컨테이너에 이벤트를 발생시킬 수 있습니다. 이 방법은 41-10페이지의 "클라이언트에 이벤트 제공"에 설명되어 있습니다.

ActiveX 컨트롤의 기반으로 사용하고 있는 VCL 컨트롤에 published 이벤트가 있는 경우 마법사에서는 ActiveX 랩퍼 클래스에 대한 클라이언트 이벤트 싱크 리스트를 관리하는 데 필요한 지원을 자동으로 추가하고 이벤트에 응답하기 위해 클라이언트가 구현해야 하는 나가는 dispinterface를 정의합니다.

컨테이너에 대한 이벤트를 발생시키려면 ActiveX 랩퍼 클래스에서 해당 이벤트를 위한 이벤트 핸들러를 VCL 객체에 구현해야 합니다. 이 이벤트 핸들러는 _TLB 유닛에 정의된 TEvents_*CoClassName* 클래스에 의해 구현되는*Fire_EventName* 메소드를 호출합니다.

```
void __fastcall TButtonXImpl::KeyPressEvent(TObject *Sender, char &Key)
{
  short TempKey;
  TempKey = (short)Key;
  Fire_OnKeyPress(&TempKey);
  Key = (short)TempKey;
};
```

그 다음에는 이 이벤트 핸들러를 VCL 컨트롤에 할당하여 이벤트가 발생할 때 호출되도록 해야 합니다. 이렇게 하려면 구현 유닛 헤더의 랩퍼 클래스 선언에 표시되는 *InitializeControl* 메소드에 이벤트를 추가합니다.

```
void InitializeControl()
{
  m_VclCtl->OnClick = ClickEvent;
  m_VclCtl->OnKeyPress = KeyPressEvent;
}
```

타입 라이브러리를 사용하여 간단한 데이터 연결 사용

간단한 데이터 연결을 사용하여 ActiveX 컨트롤의 속성을 데이터베이스의 필드에 연결할 수 있습니다. 이를 위해서는 ActiveX 컨트롤에서 필드 데이터를 나타내는 값과 그 값이 변경될 때를 호스트 애플리케이션에 전달해야 합니다. 이러한 통신은 Type Library Editor를 사용하여 속성의 연결 플래그를 설정하면 가능합니다.

속성에 연결 가능 표시를 하면 사용자가 그 속성(데이터베이스 필드 등)을 수정할 때 컨트롤에서는 값이 변경되었다는 것을 컨테이너인 클라이언트 호스트 애플리케이션에 통보하고 데이터베이스 레코드 업데이트를 요청합니다. 컨테이너에서는 데이터베이스와 상호 작용을 한 다음 레코드 업데이트 성공 또는 실패 여부를 컨트롤에 알립니다.

참고 타입 라이브러리에서 활성화한 데이터 인식 속성을 데이터베이스에 연결하는 것은 ActiveX 컨트롤을 호스팅하는 컨테이너 애플리케이션에서 담당합니다. C++Builder를 사용하여 그러한 컨테이너를 작성하는 방법에 대한 자세한 내용은 40-8페이지의 "데이터 인식 ActiveX 컨트롤 사용"을 참조하십시오.

간단한 데이터 연결을 사용하려면 타입 라이브러리를 사용하십시오.

1. 툴바에서 연결하려는 속성을 클릭합니다.
2. Flags 페이지를 선택합니다.

3 다음 연결 어트리뷰트(attribute)를 선택합니다.

연결 어트리뷰트	설명
Bindable	속성이 데이터 연결을 지원한다는 것을 나타냅니다. 속성이 bindable로 표시되면 속성 값이 변경될 때 컨테이너에 알립니다.
Request Edit	속성이 OnRequestEdit 통보를 지원한다는 것을 나타냅니다. 사용자가 컨트롤 값을 편집할 수 있는지 컨테이너에게 문의할 수 있습니다.
Display Bindable	이 속성이 bindable 속성이라는 것을 컨테이너에서 사용자에게 보여줄 수 있음을 나타냅니다.
Default Bindable	객체를 가장 잘 나타내는 하나의 bindable 속성을 나타냅니다. 이 어트리뷰트를 가지는 속성은 bindable 어트리뷰트도 가져야 합니다. 한 dispinterface에서 하나의 속성에 대해서만 지정할 수 있습니다.
Immediate Bindable	폼의 각 bindable 속성에서 이러한 동작을 지정할 수 있도록 합니다. 이 비트가 설정되어 있으면 모든 변경 사항이 통보됩니다. 이 새 비트는 bindable 및 request edit 어트리뷰트 비트가 설정되어 있어야 효력을 발생합니다.

4 툴바에서 Refresh 버튼을 클릭하여 타입 라이브러리를 업데이트합니다.

데이터 연결 컨트롤을 테스트하려면 먼저 등록해야 합니다.

예를 들어, *TEdit* 컨트롤을 데이터 연결된 ActiveX 컨트롤로 변환하려면 *TEdit*에서 ActiveX 컨트롤을 만든 다음 Text 속성 플래그를 Bindable, Display Bindable, Default Bindable 및 Immediate Bindable로 변경합니다.

ActiveX 랩퍼에 있는 데이터를 편집할 수 있도록 하려면 편집을 허용하기 전에 값을 변경할 수 있는지 ActiveX 랩퍼에서 컨테이너에 문의해야 합니다. 이를 위해 VCL 컨트롤에서 사용자로부터 키 입력 메시지를 받으면 *TVclComControl* 기본 클래스로부터 상속한 *FireOnRequestEdit*를 호출합니다. ActiveX 랩퍼에서는 이미 *OnKeyPress* 이벤트 핸들러를 VCL 컨트롤에 할당한 상태입니다. 이 이벤트는 다음과 같이 수정됩니다.

```
void __fastcall TMyAwareEditImpl::KeyPressEvent(TObject *Sender, char &Key)
{
   signed_char TempKey;
   const DISPID dispid = -517; // this is the dispatch id of the data-
bound property

   if (FireOnRequestEdit(dispid) == S_FALSE) // ask container if edits ok?
     {
      Key = 0; // if edits not ok, cancel the keypress message
      return;
     }
  TempKey = (signed_char)Key;
  Fire_OnKeyPress(&TempKey);// this forwards the OnKeyPress event to fire
in the container
  Key = (signed_char)TempKey;
};
```

또한 ActiveX 랩퍼에서는 데이터가 변경될 때 이를 컨테이너에 통보해야 합니다. 데이터 변경을 통보하려면 편집 컨트롤의 값이 변경될 때 *TVclComControl*에서 상속한 *FireOnChanged* 메소드를 호출합니다. *FireOnChanged* 메소드는 변경된 내용이 있다는 것을 컨트롤에 통보합니다. 컨트롤의 데이터가 편집되면 컨테이너에서 연결된 데이터셋을 편집 모드에 놓을 수 있습니다. *FireOnChanged*는 또한 필드 데이터를 실시간으로 수정할 수 있도록 합니다. 다음 코드는 수정된 *OnChange* 이벤트 핸들러입니다.

```
void __fastcall TMyAwareEditImpl::ChangeEvent(TObject *Sender)
{
  const DISPID dispid = -517; // the dispath id of the data-bound property
  FireOnChanged(dispid); // add this line to inform the container of
changes
  Fire_OnChange(); // this was the original call to fire the event in the
container
};
```

컨트롤은 등록하고 임포트한 후 데이터를 표시하는 데 사용할 수 있습니다.

ActiveX 컨트롤을 위한 속성 페이지 생성

속성 페이지는 사용자가 ActiveX 컨트롤의 속성을 변경할 수 있는 C++Builder Object Inspector와 비슷한 다이얼로그 박스입니다. 속성 페이지 다이얼로그 박스를 사용하여 컨트롤의 여러 속성을 그룹화하여 한 번에 편집하도록 할 수 있습니다. 더 복잡한 속성을 위해서는 다이얼로그 박스를 제공할 수 있습니다.

일반적으로 사용자는 ActiveX 컨트롤을 오른쪽 버튼으로 클릭하고 Properties를 선택하여 속성 페이지에 액세스합니다.

속성 페이지를 만드는 프로세스는 폼을 만드는 것과 비슷합니다.

1. 새 속성 페이지를 만듭니다.
2. 속성 페이지에 컨트롤을 추가합니다.
3. 속성 페이지의 컨트롤을 ActiveX 컨트롤의 속성과 연결합니다.
4. 속성 페이지를 ActiveX 컨트롤에 연결합니다.

참고 ActiveX 컨트롤 또는 ActiveForm에 속성을 추가하는 경우 영구적으로 유지하려는 속성은 게시해야 합니다. 원본으로 사용하는 VCL 컨트롤에서 속성이 published되지 않은 경우 이 속성을 published로 다시 선언하는 VCL 컨트롤의 사용자 정의 자손을 만든 다음, ActiveX 컨트롤 마법사를 사용하여 자손 클래스에서 ActiveX 컨트롤을 생성해야 합니다.

새 속성 페이지 생성

새 속성 페이지를 만들려면 Property Page 마법사를 사용합니다.

다음과 같은 방법으로 새 속성 페이지를 만듭니다.

1. File | New | Other를 선택합니다.
2. ActiveX 탭을 선택합니다.
3. Property Page 아이콘을 더블 클릭합니다.

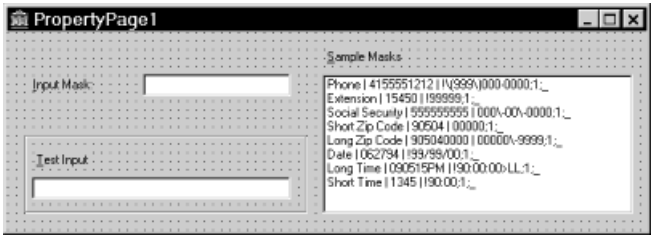
Property Page 마법사에서는 속성 페이지를 위한 새 폼과 구현 유닛을 만듭니다. 폼은 편집하는 ActiveX 컨트롤과 폼을 연결하는 *TPropertyPage*의 자손입니다. 또한 구현 유닛은 PROPERTYPAGE_IMPL 매크로를 사용하여 구현 객체를 선언합니다. 이 구현 객체는 속성 페이지 인터페이스를 구현하여 적절한 호출을 폼에 전달합니다.

속성 페이지에 컨트롤 추가

사용자가 액세스하도록 하려는 ActiveX 컨트롤의 각 속성에 대한 속성 페이지에 컨트롤을 추가해야 합니다.

다음 예에서는 ActiveX 컨트롤의 MaskEdit 속성을 설정하는 속성 페이지를 보여 줍니다.

그림 43.1 디자인 모드의 Mask Edit 속성 페이지



리스트 박스를 사용하여 사용자가 예제 마스크 리스트에서 선택할 수 있습니다. 편집 컨트롤을 사용하여 사용자가 마스크를 ActiveX 컨트롤에 적용하기 전에 테스트할 수 있습니다. 폼에 컨트롤을 추가하는 것과 같은 방법으로 속성 페이지에 컨트롤을 추가합니다.

속성 페이지 컨트롤을 ActiveX 컨트롤 속성에 연결

필요한 컨트롤을 속성 페이지에 추가한 후 각 컨트롤을 해당 속성에 연결해야 합니다. 코드를 속성 페이지의 *UpdatePropertyPage* 메소드와 *UpdateObject* 메소드에 추가하면 연결됩니다.

속성 페이지 업데이트

ActiveX 컨트롤의 속성이 변경될 때 속성 페이지의 컨트롤을 업데이트하려면 *UpdatePropertyPage* 메소드에 코드를 추가합니다. ActiveX 컨트롤 속성의 현재 값을 사용하여 속성 페이지를 업데이트하는 코드를 *UpdatePropertyPage* 메소드에 추가해야 합니다.

ActiveX 컨트롤의 인터페이스를 포함하는 *OleVariant*인 속성 페이지의 *OleObject* 속성을 사용하여 ActiveX 컨트롤에 액세스할 수 있습니다.

예를 들어, 다음 코드는 ActiveX 컨트롤의 *EditMask* 속성의 현재 값을 사용하여 속성 페이지의 편집 컨트롤인 InputMask를 업데이트합니다.

```
void __fastcall TPropertyPage1::UpdatePropertyPage(void)
{
  InputMask->Text = OleObject.OlePropertyGet("EditMask");
}
```

참고 둘 이상의 ActiveX 컨트롤을 나타내는 속성 페이지를 작성할 수도 있습니다. 이 경우에는 *OleObject* 속성을 사용하지 않고 그 대신 *OleObjects* 속성에 의해 유지 보수되는 인터페이스 리스트를 반복해서 사용합니다.

객체 업데이트

사용자가 속성 페이지의 컨트롤을 변경할 때 속성을 업데이트하려면 *UpdateObject* 메소드에 코드를 추가합니다. ActiveX 컨트롤 속성을 새로운 값으로 설정하기 위해서는 *UpdateObject* 메소드에 코드를 추가해야 합니다.

다시 한번*OleObject* 속성을 사용하여 ActiveX 컨트롤에 액세스합니다.

예를 들어, 다음 코드에서는 속성 페이지의 에디트 박스 컨트롤인 InputMask를 사용하여 ActiveX 컨트롤의 *EditMask* 속성을 설정합니다.

```
void __fastcall TPropertyPage1::UpdateObject(void)
{
  // Update OleObject from your control
  OleObject.OlePropertySet<WideString>("EditMask", WideString(InputMast-
>Text).Copy());
}
```

속성 페이지를 ActiveX 컨트롤에 연결

다음과 같은 방법으로 속성 페이지를 ActiveX 컨트롤에 연결합니다.

1 ActiveX 컨트롤의 구현 유닛의 BEGIN_PROPERTY_MAP 문과 END_PROPERTY_MAP 문 사이에 속성 페이지의 GUID를 전달하는 PROP_PAGE 매크로 호출을 추가합니다. 속성 페이지의 GUID는 속성 페이지의 구현 유닛에 정의되며 Property Page 마법사에 의해 자동으로 생성됩니다.

   예를 들어, 속성 페이지의 GUID가 기본값인 CLSID_PropertyPage1로 정의된 경우 VCL 폼을 기반으로 한 ActiveX 컨트롤의 속성 맵 섹션은 다음과 같습니다.

```
BEGIN_PROPERTY_MAP(TActiveFormXImpl)
  // Define property pages here.  Property pages are defined using
  // the PROP_PAGE macro with the class id of the page.  For example,
  // PROP_PAGE(CLSID_ActiveFormXPage)
  PROP_PAGE(CLSID_PropertyPage1)
END_PROPERTY_MAP()
```

2 ActiveX 컨트롤 유닛에 속성 페이지 유닛을 포함시킵니다.

ActiveX 컨트롤 등록

ActiveX 컨트롤을 만든 후에는 다른 애플리케이션에서 찾아 사용할 수 있도록 등록해야 합니다.

다음과 같은 방법으로 ActiveX 컨트롤을 등록합니다.

- Run|Register ActiveX Server를 선택합니다.

참고 ActiveX 컨트롤을 시스템에서 제거하기 전에 등록을 취소해야 합니다.

다음과 같은 방법으로 ActiveX 컨트롤을 등록 취소합니다.

- Run|Unregister ActiveX Server를 선택합니다.

아니면 명령줄에서 **tregsvr** 명령을 사용하거나 운영 체제에서 regsvr32.exe를 실행하는 방법도 있습니다.

ActiveX 컨트롤 테스트

컨트롤을 테스트하려면 컨트롤을 패키지에 추가하고 ActiveX 컨트롤로서 임포트합니다. 이 프로시저는 ActiveX 컨트롤을 C++Builder 컴포넌트 팔레트에 추가합니다. 컨트롤을 폼에 넣고 필요에 따라 테스트합니다.

컨트롤을 사용할 모든 대상 애플리케이션에서 테스트를 실행해야 합니다.

ActiveX 컨트롤을 디버깅하려면 Run|Parameters를 선택하고 Host Application 에디트 박스에 클라이언트 이름을 입력하십시오.

매개변수가 호스트 애플리케이션에 적용됩니다. Run|Run을 선택하면 호스트 또는 클라이언트 애플리케이션이 실행되고 컨트롤에 브레이크포인트를 설정할 수 있습니다.

ActiveX 컨트롤을 웹에 배포

만들어진 ActiveX 컨트롤을 웹 클라이언트에서 사용하려면 웹 서버에 배포해야 합니다. ActiveX 컨트롤을 변경할 때마다 다시 컴파일하고 배포해야만 클라이언트 애플리케이션에 변경 사항이 반영됩니다.

ActiveX 컨트롤을 배포하려면 클라이언트 메시지에 응답할 웹 서버가 있어야 합니다.

ActiveX 컨트롤을 배포하려면 다음 단계를 따르십시오.

1. Project|Web Deployment Options를 선택합니다.
2. Project 페이지에서 Target Dir을 웹 서버 상의 경로로서 ActiveX 컨트롤 DLL의 위치로 설정합니다. 이 경로는 C:\INETPUB\wwwroot와 같은 로컬 경로이거나 UNC 경로일 수 있습니다.
3. Target URL을 웹 서버에서 ActiveX 컨트롤 DLL의 위치로 웹 서버 상의 URL로서 파일 이름 없이 설정합니다(예 : http://mymachine.inprise.com/). 이 방법에 대한 자세한 내용은 웹 서버 설명서를 참조하십시오.
4. HTML Dir 을 ActiveX 컨트롤에 대한 참조를 포함하는 HTML 파일이 있는 위치로 설정합니다(예 : C:\INETPUB\wwwroot). 이 경로는 표준 경로 이름이거나 UNC 경로일 수 있습니다.
5. 43-17페이지의 "옵션 설정"에서 설명한 웹 배포 옵션 중 원하는 옵션을 설정합니다.
6. OK를 선택합니다.
7. Project|Web Deploy를 선택합니다.

ActiveX 라이브러리(확장명이 OCX)에 ActiveX 컨트롤을 포함하는 배포 코드 베이스가 생성됩니다. 지정한 옵션에 따라 이 배포 코드 베이스에는 확장명이 CAB인 캐비닛이나 확장명이 INF인 정보가 포함될 수 있습니다.

ActiveX 라이브러리는 단계 2에서 지정한 Target Directory에 배치됩니다. HTML 파일의 이름은 프로젝트 파일 이름과 동일하지만 확장명은 HTM입니다. 이 파일은 단계 4에서 지정한 HTML Directory에 생성됩니다. HTML 파일에는 ActiveX 라이브러리에 대한 URL 참조가 단계 3에서 지정한 위치에 포함됩니다.

참고 이러한 파일들을 웹 서버에 배포하려면 ftp와 같은 외부 유틸리티를 사용하십시오.

8 ActiveX 사용 가능 웹 브라우저를 호출하여 생성된 HTML 페이지를 봅니다.

웹 브라우저에서 이 HTML 페이지를 보면 폼 또는 컨트롤이 표시되고 브라우저 내에 포함된 애플리케이션으로 실행됩니다. 즉, 라이브러리는 브라우저 애플리케이션과 같은 프로세스에서 실행됩니다.

옵션 설정

ActiveX 컨트롤을 배포하기 전에 ActiveX 라이브러리를 만들 때 따라야 하는 웹 배포 옵션을 지정합니다.

웹 배포 옵션에는 다음 사항을 설정할 수 있도록 해주는 설정이 포함됩니다.

- **추가 파일 포함:** ActiveX 컨트롤이 패키지 또는 기타 추가 파일에 의존하는 경우 이러한 것들도 프로젝트와 함께 배포되어야 한다는 것을 나타낼 수 있습니다. 디폴트로, 이러한 파일은 전체 프로젝트를 위해 지정한 옵션과 같은 옵션을 사용하지만 Packages 또는 Additional files 탭을 사용하여 이 설정 내용을 오버라이드할 수 있습니다. 패키지 또는 추가 파일을 포함시키면 C++Builder에서 확장명이 .INF인 파일을 만듭니다. 이 파일은 ActiveX 라이브러리를 실행하기 위해 다운로드하여 설치해야 하는 여러 파일을 지정합니다. INF 파일의 구문은 URL에 다운로드하는 패키지 또는 파일을 지정할 수 있도록 되어 있습니다.
- **CAB 파일 압축:** 캐비닛(cabinet)은 파일 라이브러리에서 압축된 파일을 저장하는 단일 파일로서 대개 확장명이 **CAB**입니다. 캐비닛 압축을 사용하면 파일 다운로드 시간을 최대 70%까지 크게 줄일 수 있습니다. 설치하는 동안 브라우저에서는 캐비닛에 저장된 파일의 압축을 풀고 사용자 시스템에 복사합니다. 배포하는 각 파일은 압축된 CAB 파일일 수 있습니다. ActiveX 라이브러리가 Web Deployment Options 다이얼로그 박스의 Project 탭에서 CAB 파일 압축을 사용한다는 것을 지정할 수 있습니다.
- **버전 정보:** ActiveX 컨트롤과 함께 버전 정보를 포함시킨다는 것을 지정할 수 있습니다. 이 정보는 Project Options 다이얼로그 박스의 VersionInfo 페이지에서 설정됩니다. 이 정보의 일부인 릴리스 번호를 ActiveX 컨트롤을 배포할 때마다 매번 자동으로 업데이트되도록 할 수 있습니다. 추가 패키지나 파일을 포함시키는 경우에도 버전 정보 리소스를 INF 파일에 추가할 수 있습니다.

추가 파일 포함 여부와 CAB 파일 압축 사용 여부에 따라 ActiveX 라이브러리는 OCX 파일, OCX 파일을 포함하는 CAB 파일 또는 INF 파일이 될 수 있습니다. 다음 표는 여러 조합으로 선택했을 경우의 결과를 요약한 것입니다.

패키지 또는 추가 파일	CAB 파일 압축	결과
아니오	아니오	ActiveX 라이브러리 (OCX) 파일
아니오	예	ActiveX 라이브러리 파일을 포함하는 CAB 파일
예	아니오	INF 파일, ActiveX 라이브러리 파일 및 추가 파일과 패키지
예	예	INF 파일, ActiveX 라이브러리를 포함하는 CAB 파일 및 추가 파일과 패키지를 위한 CAB 파일

장

44

MTS 객체 또는 COM+ 객체 생성

C++Builder에서는 Microsoft Transaction Server(MTS)(Windows 2000 이전 버전용) 또는 COM+(Windows 2000 이후 버전용)에서 제공하는 트랜잭션 서비스, 보안 및 리소스 관리를 이용하는 객체를 가리키는 트랜잭션 객체라는 용어를 사용합니다. 이 객체는 대규모의 분산 환경에서 사용하기 위해 설계된 것입니다. 이 객체는 Windows 관련 기술에 종속되어 있기 때문에 크로스 플랫폼 애플리케이션에서는 사용할 수 없습니다.

C++Builder에서는 트랜잭션 객체를 만드는 마법사를 제공하므로 개발자는 COM+ 어트리뷰트(attribute)나 MTS 환경의 이점을 활용할 수 있습니다. 이 기능을 사용하면 COM 클라이언트와 서버, 특히 원격 서버 작성을 더 쉽게 구현할 수 있습니다.

참고 데이터베이스 애플리케이션의 경우에는 C++Builder가 트랜잭션 데이터 모듈도 제공합니다. 자세한 내용은 29장, "멀티 티어 애플리케이션 생성"을 참조하십시오.

트랜잭션 객체에서는 다음과 같은 여러 가지 저수준 서비스를 사용합니다.

- 서버 애플리케이션에서 동시에 여러 사용자를 처리할 수 있게 프로세스, 스레드 및 데이터베이스 연결을 포함한 시스템 리소스 관리
- 애플리케이션을 신뢰할 수 있게 자동으로 트랜잭션을 초기화 및 제어
- 필요한 경우 서버 컴포넌트 작성, 실행 및 삭제
- 승인받은 사용자만 애플리케이션을 액세스할 수 있도록 역할 기반 보안 제공
- 클라이언트가 서버에서 발생하는 조건에 응답할 수 있도록 이벤트 관리(COM+ 전용)

MTS나 COM+에서 이러한 기본 서비스를 제공하게 함으로써 특정한 분산 애플리케이션의 특성을 개발하는 데 집중할 수 있습니다. MTS 또는 COM+ 중 어떤 기술을 선택할지 여부는 애플리케이션을 실행하기로 선택한 서버에 따라 달라집니다. 클라이언트에서 특수한 인터페이스를 통해 트랜잭션 서비스를 명시적으로 조작하지 않는 한 서버 객체가 MTS와 COM+ 중 어떤 서비스를 사용하더라도 클라이언트에게는 전혀 차이가 없습니다.

트랜잭션 객체 이해

대개 트랜잭션 객체는 크기가 작고 불연속적인 비즈니스 함수에 사용합니다. 트랜잭션 객체는 애플리케이션의 비즈니스 룰을 구현하면서 애플리케이션 상태의 보기와 변환을 제공합니다. 예를 들어, 의료 애플리케이션의 경우 여러 데이터베이스에 저장된 의료 기록은 환자의 건강 기록과 같은 애플리케이션의 지속적인 상태를 나타냅니다. 트랜잭션 객체는 해당 상태를 업데이트하여 새로운 환자, 검사 결과 및 X레이 파일 등과 같은 변경 사항을 반영합니다.

트랜잭션 객체는 분산 컴퓨팅 환경에서 발생하는 문제를 처리하기 위해 MTS나 COM+에서 제공하는 어트리뷰트(attribute) 집합을 사용한다는 점에서 다른 COM 객체와 구분됩니다. 이런 어트리뷰트 중 일부는 트랜잭션 객체가 *IObjectControl* 인터페이스를 구현해야 사용할 수 있습니다. *IObjectControl*에서는 객체가 활성화되거나 비활성화될 때 호출되고 데이터베이스 연결과 같은 리소스를 관리할 수 있는 메소드를 정의합니다. 44-9페이지의 "객체 풀링"에서 설명하는 객체 풀링에도 이 인터페이스가 필요합니다.

참고 MTS를 사용할 경우 트랜잭션 객체가 *IObjectControl*을 구현해야 합니다. COM+에서는 *IObjectControl*이 반드시 필요하지는 않지만 강력하게 권장합니다. Transactional Object 마법사에서는 *IObjectControl*에서 파생되는 객체를 제공합니다.

트랜잭션 객체의 클라이언트를 **기본 클라이언트**라고 합니다. 기본 클라이언트 관점에서 트랜잭션 객체는 다른 COM 객체와 같습니다.

MTS에서는 트랜잭션 객체가 라이브러리(DLL)에 내장되어야 하며, 이 라이브러리는 다시 MTS 런타임 환경에 MTS 실행 파일인 mtxex.exe로 설치됩니다. 즉 서버 객체가 MTS 런타임 프로세스 공간에서 실행됩니다. MTS 실행 파일은 기본 클라이언트와 같은 프로세스에서 실행될 수도 있고, 기본 클라이언트와 같은 컴퓨터의 독립적인 프로세스로 실행될 수도 있으며, 독립적인 컴퓨터의 원격 서버 프로세스로 실행될 수도 있습니다.

COM+에서는 서버 애플리케이션이 in-process 서버일 필요는 없습니다. 여러 서비스가 COM 라이브러리에 통합되어 있기 때문에 각각의 MTS 프로세스에서 서버에 대한 호출을 인터셉트할 필요가 없습니다. 대신 COM이나 COM+에서 리소스 관리, 트랜잭션 지원 등을 제공합니다. 그러나 이번에는 서버 애플리케이션을 COM+ 애플리케이션에 설치해야 합니다.

기본 클라이언트와 트랜잭션 객체 사이의 연결은 out-of-process 서버에서처럼 클라이언트 대리자와 서버의 스텁에서 처리합니다. 연결 정보는 대리자가 가지고 있습니다. 클라이언트에서 서버에 대한 연결이 필요한 동안에는 기본 클라이언트와 대리자 사이의 연결이 열려 있습니다. 그러므로 클라이언트에게는 서버에 계속 액세스하고 있는 것처럼 보입니다. 그렇지만 실제로는 다른 클라이언트에서 연결을 사용할 수 있게 리소스를 보존하면서 대리자가 객체를 비활성화하거나 재활성화할 수 있습니다. 활성 및 비활성에 대한 자세한 내용은 44-4페이지의 "Just-in-time 활성화"를 참조하십시오.

트랜잭션 객체의 요구 사항

COM 요구 사항 외에도 트랜잭션 객체는 다음 요구 사항을 만족해야 합니다.

- 객체에 표준 클래스 팩토리가 있어야 합니다. 이것은 객체를 만들 때 마법사에서 자동으로 제공합니다.
- 서버는 표준 *DllGetClassObject* 메소드를 엑스포트하여 클래스 객체를 노출해야 합니다. 이를 위한 코드는 마법사에서 제공합니다.
- 모든 객체 인터페이스와 CoClass는 마법사에서 자동으로 만드는 타입 라이브러리로 나타내야 합니다. Type Library Editor를 사용하여 타입 라이브러리에 인터페이스에 대한 메소드와 속성을 추가할 수 있습니다. MTS Explorer나 COM+ Component Manager에서 타입 라이브러리의 정보를 사용하여 런타임 시 설치된 컴포넌트에 대한 정보를 추출합니다.
- 서버는 표준 COM 마샬링을 사용하는 인터페이스만 엑스포트해야 합니다. Transactional Object 마법사에서 자동으로 이를 제공합니다. C++Builder의 트랜잭션 객체 지원에서는 사용자 정의 인터페이스에 대한 수동 마샬링을 허용하지 않습니다. 모든 인터페이스는 COM의 자동 마샬링 지원을 사용하는 이중 인터페이스로 구현해야 합니다.
- 서버는 *DllRegisterServer* 함수를 엑스포트하고 이 루틴에 CLSID, ProgID, 인터페이스 및 타입 라이브러리를 자가 등록해야 합니다. 이 기능은 Transactional Object 마법사에서 제공합니다.

COM+ 대신 MTS를 사용하면 다음 조건이 적용됩니다.

- MTS에서는 서버가 동적 연결 라이브러리(DLL)여야 합니다. 실행 파일(.EXE 파일)로 구현된 서버는 MTS 런타임 환경에서 실행할 수 없습니다.
- 객체에서 *IObjectControl* 인터페이스를 구현해야 합니다. Transactional Object 마법사에서 이 인터페이스에 대한 지원을 자동으로 추가합니다.
- MTS 프로세스 공간에서 실행하는 서버는 MTS에서 실행되지 않는 COM 객체와 결합할 수 없습니다.

리소스 관리

트랜잭션 객체를 관리하여 애플리케이션에서 사용하는 리소스를 더 잘 관리할 수 있습니다. 이러한 리소스에는 객체 인스턴스 자체에 대한 메모리로부터 데이터베이스 연결처럼 이들이 사용하는 리소스에 이르기까지 모든 것이 포함됩니다.

일반적으로 객체를 설치하고 구성하는 방법으로 애플리케이션에서 리소스를 관리하는 방법을 구성합니다. 트랜잭션 객체에서 다음을 이용할 수 있게 설정합니다.

- Just-in-time 활성화
- 리소스 풀링
- 객체 풀링(COM+ 전용)

그러나 객체에서 이러한 서비스를 최대한 이용하게 하려면 객체에서 *IObjectContext* 인터페이스를 사용하여 리소스를 안전하게 릴리스할 수 있는 시점을 나타내야 합니다.

객체 컨텍스트 액세스

COM 객체에서처럼 트랜잭션 객체를 사용하려면 먼저 만들어야 합니다. COM 클라이언트에서는 COM 라이브러리 함수 *CoCreateInstance*를 호출하여 객체를 만듭니다.

모든 트랜잭션 객체에는 해당하는 컨텍스트 객체가 있어야 합니다. MTS나 COM+에서 이 컨텍스트 객체를 자동으로 구현하며 이 컨텍스트 객체를 사용하여 트랜잭션 객체를 관리합니다. 컨텍스트 객체의 인터페이스는 *IObjectContext*입니다. Transactional Object 마법사에서 만든 트랜잭션 객체를 활성화하면 자동으로 객체 컨텍스트를 가져옵니다. 객체 컨텍스트는 m_spObjectContext라고 하는 멤버 변수에 저장됩니다. 예를 들어, 다음과 같은 객체 컨텍스트 포인터를 사용할 수 있습니다.

```
BOOL flags;
m_spObjectContext->IsCallerInRole(OLESTR("Manager"), &flags);
```

또한 *TMtsDll*의 *Get_ObjectContext* 메소드를 호출하여 객체 컨텍스트에 대한 포인터를 가져올 수도 있습니다. *TMtsDll* 객체에서는 MTS나 COM+에서 애플리케이션을 실행하는지 여부와 상관 없이 사용할 수 있게 사용자의 호출을 사용하여 객체 컨텍스트를 가져옵니다.

```
IObjectContext* IAmWatchingYou = NULL;
TMtsDll MTSDLL;
HRESULT hr = MTSDLL.Get_ObjectContext(&IAmWatchingYou);
if (! (SUCCEEDED(hr)) )
{
  // do something useful with the error
}
BOOL flags;
IAmWatchingYou->IsCallerInRole(OLESTR("Manager"), &flags);
```

경고 Comsvcs.h에 정의된 *GetObjectContext* 매크로 대신 *TMtsDll*의 *Get_ObjectContext* 메소드를 사용하십시오. *TDatabase*의 *GetObjectContext* 메소드 재정의를 피하기 위해 VCL 헤더에서는 이를 정의하지 않습니다.

참고 m_spObjectContext 멤버는 *TMtsDll* 메소드를 사용하여 할당하므로 MTS와 COM+ 모두에서 작동합니다.

Just-in-time 활성화

클라이언트가 객체에 대한 참조를 갖고 있는 동안 객체를 비활성화하거나 재활성화할 수 있는 기능을 **just-in-time 활성화**라고 합니다. 클라이언트의 관점에서는 클라이언트에서 객체를 만들고 최종 릴리스할 때까지 오직 단일 인스턴스의 객체만 존재하지만 실제로는 객체가 여러 번 비활성화되고 재활성화됩니다. 객체를 비활성화하면 클라이언트에서는 시스템 리소스에 영향을 미치지 않고도 확장된 시간 동안 객체에 대한 참조를 보유할 수 있습니다. 객체를 비활성화하면 해당하는 모든 리소스를 릴리스할 수 있습니다. 예를 들어, 객체를 비활성화하면 해당하는 데이터베이스 연결을 릴리스하여 다른 객체에서 이를 사용할 수 있습니다.

트랜잭션 객체는 비활성화된 상태로 만들어져서 클라이언트 요청을 받으면 활성화됩니다. 트랜잭션 객체를 만들면 해당하는 컨텍스트 객체도 만들어집니다. 이 컨텍스트 객체는 한 번 이상의 재활성화 주기를 통해 트랜잭션 객체의 전체 수명 동안 존재합니다. *IObjectContext* 인터페이스에서 액세스한 컨텍스트 객체는 비활성화 기간 중의 객체 추적 내용을 보관하고 트랜잭션을 조정합니다.

트랜잭션 객체를 비활성화하는 것이 안전하면 바로 비활성화됩니다. 이를 **as-soon-as-possible 비활성화**라고 합니다. 다음과 같은 경우가 발생하면 트랜잭션 객체가 비활성화됩니다.

- **객체에서 SetComplete 또는 SetAbort**로 비활성화를 요청한 경우: 객체에서 작업을 성공적으로 완료하고 클라이언트의 다음 호출을 위해 내부 객체 상태를 저장하지 않아도 될 경우 객체에서는 *IObjectContext SetComplete* 메소드를 호출합니다. 객체에서 작업을 성공적으로 완료할 수 없고 객체 상태를 저장할 필요가 없음을 표시하려면 객체에서 *SetAbort*를 호출합니다. 즉, 객체의 상태가 현재 트랜잭션 이전의 상태로 롤백합니다. 때로 객체를 **stateless**로 설계할 수 있는데, stateless는 모든 메소드에서 반환 시 객체를 비활성화함을 의미합니다.
- **트랜잭션이 커밋되거나 중지된 경우:** 객체의 트랜잭션이 커밋되거나 중지된 경우 객체가 비활성화됩니다. 비활성화된 객체 중에 계속 존재하는 객체는 트랜잭션 외부의 클라이언트가 참조하는 객체뿐입니다. 나중에 이러한 객체를 호출하면 객체가 다시 활성화되어 새 트랜잭션에서 실행됩니다.
- **마지막 클라이언트에서 객체를 릴리스할 경우:** 물론 클라이언트에서 객체를 릴리스하면 객체가 비활성화되고 객체 컨텍스트도 릴리스됩니다.

참고 IDE에서 COM+에 트랜잭션 객체를 설치할 경우 Type Library Editor의 COM+ 페이지를 사용하여 객체에서 just-in-time 활성화를 지원하는지 여부를 지정할 수 있습니다. Type Library Editor에서 객체(CoClass)를 선택하고, COM+ 페이지로 이동한 다음 Just In Time Activation 박스를 선택하거나 선택을 해제하면 됩니다. 그렇지 않으면 시스템 관리자가 COM+ Component Manager나 MTS Explorer를 사용하여 이 어트리뷰트(attribute)를 지정합니다. 시스템 관리자가 Type Library Editor를 사용하여 사용자가 지정한 설정을 오버라이드할 수도 있습니다.

리소스 풀링

비활성화 상태에 있는 유휴 시스템 리소스는 해제되므로 이 해제된 리소스를 다른 서버 객체에서 사용할 수 있습니다. 예를 들어, 서버 객체에서 더 이상 사용하지 않는 데이터베이스 연결을 다른 클라이언트에서 다시 사용할 수 있습니다. 이를 **리소스 풀링**이라고 합니다. 풀링된 리소스는 리소스 디스펜서에서 관리합니다.

리소스 디스펜서에서 리소스를 캐싱하므로 함께 설치된 트랜잭션 객체에서 리소스를 공유할 수 있습니다. 리소스 디스펜서에서는 비지속적인 공유 상태 정보도 관리합니다. 이런 방식으로 리소스 디스펜서는 SQL 서버같은 리소스 관리자와 유사하지만 지속성을 보장하지는 않습니다.

트랜잭션 객체를 작성할 경우 개발자에게 제공되는 다음 두 가지 유형의 리소스 디스펜서를 이용할 수 있습니다.

- 데이터베이스 리소스 디스펜서
- Shared Property Manager

다른 객체에서 풀링된 리소스를 사용하려면 이를 명시적으로 릴리스해야 합니다.

데이터베이스 리소스 디스펜서

데이터베이스 연결을 열고 닫는 작업은 시간이 많이 걸릴 수 있습니다. 리소스 디스펜서를 사용하여 데이터베이스 연결을 풀링하면 객체에서 새로운 데이터베이스 연결을 만드는 대신 기존의 데이터베이스 연결을 다시 사용할 수 있습니다. 예를 들어, 고객 유지 보수 애플리케이션에서 데이터베이스 조회 및 데이터베이스 업데이트 컴포넌트를 실행할 경우 해당 컴포넌트를 함께 설치한 다음 데이터베이스 연결을 공유하도록 할 수 있습니다. 이렇게 하면, 애플리케이션이 많은 연결을 필요로 하지 않으며 새로운 객체 인스턴스가 이미 열려 있지만 사용하지 않는 연결을 사용하므로 데이터를 더 빨리 액세스할 수 있습니다.

- BDE 컴포넌트를 사용하여 데이터에 연결할 경우 리소스 디스펜서는 BDE(Borland Database Engine)입니다. MTS를 사용하여 트랜잭션 객체를 설치한 경우에만 이 리소스 디스펜서를 사용할 수 있습니다. 리소스 디스펜서를 사용하려면 BDE 관리자를 사용하여 구성의 System/Init 영역에서 MTS POOLING을 선택하십시오.
- ADO 데이터베이스 컴포넌트를 사용하여 데이터에 연결할 경우 리소스 디스펜서는 ADO에서 제공합니다.

참고 InterbaseExpress 컴포넌트를 사용하여 데이터베이스를 액세스할 경우에는 기본 리소스 풀링이 없습니다.

원격 트랜잭션 데이터 모듈의 경우 객체의 트랜잭션에 연결이 자동으로 참여하고 리소스 디스펜서에서 자동으로 연결을 다시 요구하여 사용합니다.

Shared Property Manager

Shared Property Manager는 서버 프로세스 내의 여러 객체 사이에서 상태를 공유하기 위해 사용할 수 있는 리소스 디스펜서입니다. Shared Property Manager를 사용하면 공유 데이터를 관리하기 위해 애플리케이션에 많은 코드를 추가할 필요가 없습니다. Shared Property Manager는 동시 액세스로부터 공유 속성을 보호하기 위해 잠금과 세마포를 구현하여 대신 이를 처리합니다. Shared Property Manager에서는 포함하고 있는 공유 속성에 대한 고유한 이름 공간을 설정하는 **공유 속성 그룹**을 제공하여 이름 충돌을 제거합니다.

Shared Property Manager 리소스를 사용하려면 먼저 *CreateSharedPropertyGroup* helper 함수를 사용하여 공유 속성 그룹을 만들어야 합니다. 그러면 해당 그룹에 모든 속성을 작성하고 해당 그룹의 모든 속성을 읽을 수 있습니다. 공유 속성 그룹을 사용하면 트랜잭션 객체가 비활성화될 때마다 상태 정보가 저장됩니다. 그리고 같은 MTS 패키지나 COM+ 애플리케이션에 설치된 모든 트랜잭션 객체에서 상태 정보를 공유할 수 있습니다. 44-27페이지의 "트랜잭션 객체 설치"에서 설명한 대로 트랜잭션 객체를 패키지에 설치할 수 있습니다.

객체에서 상태를 공유하려면 모든 객체가 같은 프로세스 내에서 실행되어야 합니다. 다른 컴포넌트의 인스턴스에서 속성을 공유하게 하려면 같은 MTS 패키지나 COM+ 애플리케이션에 설치해야 합니다. 관리자가 패키지 간에 컴포넌트를 이동할 수 있는 위험이 있기 때문에 공유 속성 그룹 사용을 같은 DLL이나 EXE에서 정의한 객체의 인스턴스로 제한하는 것이 가장 안전합니다.

속성을 공유하는 객체는 같은 활성화 어트리뷰트(attribute)를 갖고 있어야 합니다. 같은 패키지의 두 컴포넌트가 서로 다른 활성화 어트리뷰트를 갖고 있는 경우 일반적으로 속성을 공유할 수 없습니다. 예를 들어, 한 컴포넌트는 클라이언트 프로세스에서 실행되도록 구성하고 다른 컴포넌트는 서버 프로세스에서 실행되도록 구성한 경우 이 컴포넌트들이 같은 MTS 패키지나 COM+ 애플리케이션에 있더라도 해당 객체는 대개 다른 프로세스에서 실행됩니다.

다음 예제는 트랜잭션 객체에서 Shared Property Manager 를 지원하기 위해 코드를 추가하는 방법을 보여 줍니다.

예제: 트랜잭션 객체 인스턴스에서 속성 공유

이 예제에서는 MyGroup이라는 속성 그룹을 만들어 객체와 객체 인스턴스 간에 공유할 속성을 포함시킵니다. 이 예제에는 Counter라는 공유 속성이 있으며, *CreateSharedPropertyGroup* helper 함수를 사용하여 속성 그룹 관리자와 속성 그룹을 만든 다음 Group 객체의 *CreateProperty* 메소드를 사용하여 Counter라는 속성을 만듭니다.

속성의 값을 가져오려면 아래 표시된 것처럼 Group 객체의 *PropertyByName* 메소드를 사용합니다. *PropertyByPosition* 메소드를 사용할 수도 있습니다.

```
#include "Project1_TLB.H"
#define _MTX_NOFORCE_LIBS
#include <vcl\mtshlpr.h>

class ATL_NO_VTABLE TSharedPropertyExampleImpl :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<TSharedPropertyExampleImpl,
&CLSID_SharedPropertyExample>,
  public IObjectControl,
  public IDispatchImpl<ISharedPropertyExample, &IID_ISharedPropertyExample,
&LIBID_Project1>
{
private:
  ISharedPropertyGroupManager* manager;
  ISharedPropertyGroup* PG13;
  ISharedProperty* Counter;
public:
  TSharedPropertyExampleImpl()
  {
  }
  DECLARE_THREADING_MODEL(otApartment);
  DECLARE_PROGID("Project1.SharedPropertyExample");
  DECLARE_DESCRIPTION("");

static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
  TTypedComServerRegistrarT<TSharedPropertyExampleImpl>
  regObj(GetObjectCLSID(), GetProgID(), GetDescription());
  return regObj.UpdateRegistry(bRegister);
}

DECLARE_NOT_AGGREGATABLE(TSharedPropertyExampleImpl)

BEGIN_COM_MAP(TSharedPropertyExampleImpl)
  COM_INTERFACE_ENTRY(ISharedPropertyExample)
  COM_INTERFACE_ENTRY(IObjectControl)
```

```
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

public:
  STDMETHOD(Activate)();
  STDMETHOD(IncrementCounter());
  STDMETHOD_(BOOL, CanBePooled)();
  STDMETHOD_(void, Deactivate)();

CComPtr<IObjectContext>m_spObjectContext;

public:

};

// SHAREDPROPERTYEXAMPLEIMPL : Implementation of
TSharedPropertyExampleImpl

#include <vcl.h>
#pragma hdrstop

#include "SHAREDPROPERTYEXAMPLEIMPL.H"

STDMETHODIMP TSharedPropertyExampleImpl::Activate()
{
  static TMtsDll Mts;
  HRESULT hr = E_FAIL;
  hr = Mts.Get_ObjectContext (&m_spObjectContext);
  if (SUCCEEDED(hr))
  {
    VARIANT_BOOL _false = VARIANT_FALSE;
    VARIANT_BOOL _true = VARIANT_TRUE;
    CoCreateInstance( CLSID_SharedPropertyGroupManager, NULL,
CLSCTX_INPROC_SERVER,
                      IID_ISharedPropertyGroupManager, (void**)manager);
    manager->CreatePropertyGroup (L"Test Group", LockSetGet, Standard,
&_false, &PG13);
    if ( (PG13->CreateProperty (L"Counter", &_true, &Counter)) == S_OK)
    {
      Counter->put_Value(TVariant(0));
    }
    return S_OK;
  }
  return hr;
}

STDMETHODIMP_(BOOL) TSharedPropertyExampleImpl::CanBePooled()
{
  return FALSE;
}

STDMETHODIMP_(void) TSharedPropertyExampleImpl::Deactivate()
{
  PG13->Release();
  manager->Release();
  m_spObjectContext.Release();
}
```

```
STDMETHODIMP TSharedPropertyExampleImpl::IncrementCounter()
{
  try
  {
    TVariant temp;
    Counter->get_Value(&temp);
    temp=(int)temp+1;;
    Counter->put_Value(temp);
  }
  catch(Exception &e)
  {
    return Error (e.Message.c_str(), IID_ISharedPropertyExample);
  }
  return S_OK;
};
```

리소스 릴리스

객체의 리소스를 릴리스해야 합니다. 대개는 클라이언트 요청을 처리한 후에 *IObjectContext* 메소드인 *SetComplete* 및 *SetAbort*를 호출하여 리소스를 릴리스합니다. 이러한 메소드는 리소스 디스펜서에서 할당한 리소스를 릴리스합니다.

이 때 트랜잭션 객체와 컨텍스트 객체를 포함한 다른 객체에 대한 참조와 컴포넌트의 인스턴스에서 차지하는 메모리를 포함하여 다른 모든 리소스에 대한 참조를 릴리스해야 합니다.

클라이언트 호출 사이에서 상태를 유지할 경우에만 이러한 호출을 포함시키지 않습니다. 자세한 내용은 44-12페이지의 "stateful 객체와 stateless 객체"를 참조하십시오.

객체 풀링

리소스를 풀링할 수 있는 것처럼 COM+에서 객체도 풀링할 수 있습니다. 객체를 비활성화하면 COM+에서는 *IObjectControl* 인터페이스 메소드 *CanBePooled*를 호출합니다. 이 메소드는 객체를 풀링하여 다시 사용할 수 있음을 나타냅니다. *CanBePooled*에서 **true**를 반환할 경우 비활성화 시 소멸되는 대신 객체가 객체 풀로 이동됩니다. 객체를 요청한 클라이언트에서 객체를 사용할 수 있는 기간 동안 객체는 지정한 제한 시간 중에 객체 풀에 남아 있습니다. 객체 풀이 비어 있을 경우에만 객체의 새 인스턴스가 만들어집니다. **false**를 반환하거나 *IObjectControl* 인터페이스를 지원하지 않는 객체는 비활성화되면 소멸됩니다.

MTS에서는 객체 풀링을 사용할 수 없습니다. MTS에서도 이미 설명한 *CanBePooled*를 호출하지만 풀링은 발생하지 않습니다. 이 때문에 마법사에서 트랜잭션 객체를 만들 경우 생성된 *CanBePooled* 메소드에서는 항상 **false**를 반환합니다. COM+에서만 객체를 실행하고 객체 풀링을 허용하려면 구현 유닛에서 이 메소드를 찾아 **true**를 반환하도록 편집하십시오.

객체의 *CanBePooled* 메소드에서 **true**를 반환하더라도 COM+에서 객체를 객체 풀로 이동하지 못하도록 구성할 수 있습니다. IDE에서 COM+에 트랜잭션 객체를 설치한 경우 COM+에서 Type Library Editor의 COM+ 페이지를 사용하여 객체를 풀링할지 여부를 지정할 수 있습니다. Type Library Editor에서 객체(CoClass)를 선택하고 COM+ 페이지로 이동한 다음 객체 풀링 박스를 선택하거나 선택을 해제하십시오. 또는 시스템 관리자가 COM+ Component Manager나 MTS Explorer를 사용하여 이 어트리뷰트(attribute)를 지정할 수도 있습니다.

마찬가지로 객체를 해제하기 전에 비활성화된 객체가 객체 풀에 남아 있는 시간을 구성할 수 있습니다. IDE에서 설치할 경우 Type Library Editor의 COM+ 페이지에 있는 Creation TimeOut 설정을 사용하여 이 기간을 지정할 수 있습니다. 또는 시스템 관리자가 COM+ Component Manager를 사용하여 이 어트리뷰트를 지정할 수도 있습니다.

MTS 및 COM+ 트랜잭션 지원

트랜잭션 객체에 이름을 제공하는 트랜잭션 지원을 사용하면 액션을 트랜잭션으로 그룹화할 수 있습니다. 예를 들어, 의료 기록 애플리케이션에서 Transfer 컴포넌트가 의사 사이의 레코드를 전송하도록 한 경우 동일한 트랜잭션에 Add 및 Delete 메소드를 추가시킬 수 있습니다. 그런 방식으로 전체 Transfer가 작동하거나 이전 상태로 Transfer를 롤백할 수 있습니다. 트랜잭션은 *여러* 데이터베이스를 액세스해야 하는 애플리케이션의 오류 복구를 간소화합니다.

트랜잭션은 다음 사항을 보증합니다.

- 단일 트랜잭션의 모든 업데이트가 커밋되거나 또는 중지되어 이전 상태로 롤백됩니다. 이것을 **원자성**이라고 합니다.
- 트랜잭션은 상태 불변값을 보존하는 시스템 상태의 합당한 변환입니다. 이것을 **일관성**이라고 합니다.
- 동시 트랜잭션은 서로의 부분적인 결과나 커밋되지 않은 결과를 알지 못하므로 애플리케이션 상태에서 비일관성을 만들 수 있습니다. 이것을 **분리**라고 합니다. 리소스 관리자는 트랜잭션 기반 동기화 프로토콜을 사용하여 활성 트랜잭션의 커밋되지 않은 작업을 분리합니다.
- 데이터베이스 레코드 같이 관리된 리소스에 커밋된 업데이트는 통신 오류, 프로세스 오류, 서버 시스템 오류 등의 오류를 복구할 수 있습니다. 이것을 **지속성**이라고 합니다. 트랜잭션 로깅을 사용하면 디스크 매체 오류 후에 지속 상태를 복구할 수 있습니다.

객체와 관련된 컨텍스트 객체는 트랜잭션 내에서 객체가 실행되는지 여부와 그럴 경우 트랜잭션 ID를 나타냅니다. 객체가 트랜잭션의 일부일 경우 리소스 관리자와 리소스 디스펜서에서 대신 수행하는 서비스도 트랜잭션에서 실행됩니다. 리소스 디스펜서에서는 컨텍스트 객체를 사용하여 트랜잭션 기반 서비스를 제공합니다. 예를 들어, 트랜잭션 내에서 실행되는 객체에서 ADO나 BDE 리소스 디스펜서를 사용하여 데이터베이스 연결을 할당할 경우 연결이 자동으로 트랜잭션에 참여합니다. 이 연결을 사용하는 모든 데이터베이스 업데이트가 트랜잭션의 일부가 되며 커밋되거나 중지됩니다.

여러 객체의 작업이 트랜잭션 하나로 구성될 수 있습니다. MTS와 COM+의 가장 중요한 장점은 객체를 각각의 트랜잭션에 유지하거나 하나의 트랜잭션에 속하는 보다 큰 객체 그룹의 일부가 될 수 있게 하는 것입니다. 그러면 객체를 다양한 방법으로 사용할 수 있기 때문에 애플리케이션 개발자는 애플리케이션 로직을 재작성할 필요 없이 다른 애플리케이션에서 애플리케이션 코드를 다시 사용할 수 있습니다. 실제로 개발자는 트랜잭션 객체를 설치할 때 트랜잭션에서 객체가 사용되는 방법을 결정할 수 있습니다. 다른 MTS 패키지나 COM+ 애플리케이션에 객체를 추가하기만 하면 트랜잭션 동작을 변경할 수 있습니다. 트랜잭션 객체 설치에 대한 자세한 내용은 44-27페이지의 "트랜잭션 객체 설치"를 참조하십시오.

트랜잭션 어트리뷰트(attribute)

모든 트랜잭션 객체에는 MTS 카탈로그에 기록되거나 COM+ 에 등록된 트랜잭션 어트리뷰트가 있습니다.

C++Builder를 사용하면 Transactional Object 마법사나 Type Library Editor를 사용하여 디자인 타임 시 트랜잭션 어트리뷰트를 설정할 수 있습니다.

각 트랜잭션 어트리뷰트는 다음과 같이 설정될 수 있습니다.

Requires a transaction 객체는 *트랜잭션 범위 내*에서 실행되어야 합니다. 새로운 객체를 만들면 해당 객체 컨텍스트는 클라이언트의 컨텍스트에서 트랜잭션을 상속받습니다. 클라이언트에 트랜잭션 컨텍스트가 없으면 자동으로 새로운 트랜잭션 컨텍스트가 만들어집니다.

Requires a new transaction 객체는 *각각의 트랜잭션 내에서* 실행되어야 합니다. 새로운 객체를 만들면 객체의 클라이언트에 트랜잭션이 있는지 여부와 상관 없이 객체에 대한 새로운 트랜잭션이 자동으로 만들어집니다. 객체 클라이언트의 트랜잭션 범위 내에서는 객체가 실행되지 않습니다. 시스템은 항상 새로운 객체에 대해 독립 트랜잭션을 만듭니다.

Supports transaction 객체는 *객체 클라이언트의 트랜잭션 범위 내*에서 실행될 수 있습니다. 새로운 객체를 만들면 해당 객체 컨텍스트는 클라이언트의 컨텍스트에서 트랜잭션을 상속 받습니다. 그러면 여러 객체를 하나의 트랜잭션으로 구성할 수 있습니다. 클라이언트에 트랜잭션이 없으면 새로운 컨텍스트도 트랜잭션 없이 만들어집니다.

Transactions Ignored *트랜잭션의 범위 내*에서는 객체가 실행되지 않습니다. 새로운 객체를 만들면 클라이언트에 트랜잭션이 있는지 여부와 상관 없이 객체 컨텍스트는 트랜잭션 없이 만들어집니다. 이 설정은 COM+에서만 사용할 수 있습니다.

Does not support transactions 이 설정의 의미는 MTS 또는 COM+ 에서 객체를 설치했는지 여부에 따라 달라집니다. MTS에서 이 설정은 COM+의 Transactions Ignored와 의미가 같습니다. COM+에서는 트랜잭션 없이 객체 컨텍스트가 만들어질 뿐만 아니라 클라이언트에 트랜잭션이 있는 경우 객체가 활성화되지 않게 합니다.

트랜잭션 어트리뷰트 설정

Transactional Object 마법사를 사용하여 처음 트랜잭션 객체를 만들 때 트랜잭션 어트리뷰트를 설정할 수 있습니다.

Type Library Editor를 사용하여 트랜잭션 어트리뷰트를 설정하거나 변경할 수도 있습니다. 다음과 같은 방법으로 Type Library Editor에서 트랜잭션 어트리뷰트를 변경합니다.

1 View | Type Library를 선택하여 Type Library Editor를 엽니다.

2 트랜잭션 객체에 해당하는 클래스를 선택합니다.

3 COM+ 탭을 클릭하고 원하는 트랜잭션 어트리뷰트를을 선택합니다.

경고 트랜잭션 어트리뷰트(attribute)를 설정하면 C++Builder에서 타입 라이브러리에 사용자 정의 데이터와 같은 지정된 어트리뷰트에 대해 특수한 GUID를 삽입합니다. C++Builder 외부에서는 이 값을 인식하지 못합니다. 그러므로 IDE에서 트랜잭션 객체를 설치한 경우에만 영향을 미칩니다. 그렇지 않으면 시스템 관리자가 MTS Explorer나 COM+ Component Manager를 사용하여 이 값을 설정해야 합니다.

참고 트랜잭션 객체를 이미 설치했으면 트랜잭션 어트리뷰트를 변경할 경우 먼저 객체를 제거한 다음 다시 설치해야 합니다. 그렇게 하려면 Run | Install MTS objects 또는 Run | Install COM+ objects를 사용하십시오.

stateful 객체와 stateless 객체

다른 COM 객체처럼 트랜잭션 객체는 클라이언트와 여러 번 상호 작용하는 동안 내부 상태를 유지할 수 있습니다. 예를 들어, 클라이언트가 한 번의 호출에서 속성 값을 설정하면 다음 호출 수행 시에도 해당 속성 값이 변경되지 않고 남아있어야 합니다. 이런 객체를 **stateful**이라고 합니다. 트랜잭션 객체는 **stateless**일 수도 있는데, 이는 클라이언트의 다음 호출을 기다리는 동안 객체에서 중간 상태를 보관하지 않음을 의미합니다.

트랜잭션을 커밋하거나 중지한 경우 트랜잭션에 관련된 모든 객체가 비활성화되므로 트랜잭션 과정 중에 취득한 상태를 잃어버리게 됩니다. 그러면 트랜잭션 분리와 데이터베이스 일관성이 보장되고 다른 트랜잭션에서 사용할 수 있도록 서버 리소스도 해제됩니다. 트랜잭션을 완료하면 객체에서 가지고 있는 리소스를 객체가 비활성화되었을 때 다시 요구할 수 있습니다. 객체의 상태가 릴리스되는 시점을 제어하는 방법에 대한 자세한 내용은 다음 절을 참조하십시오.

객체의 상태를 유지하려면 객체가 활성화 상태여야 하고, 데이터베이스 연결같이 잠재적으로 가치있는 자원을 가지고 있어야 합니다.

트랜잭션 종료 방법 변경

트랜잭션 객체에서는 다음 표에 표시된대로 *IObjectContext* 메소드를 사용하여 트랜잭션 완료 방법을 변경합니다. 객체의 트랜잭션 어트리뷰트(attribute)와 함께 이 메소드를 사용하면 단일 트랜잭션에 하나 이상의 객체를 참여시킬 수 있습니다.

표 44.1 트랜잭션 지원을 위한 IObjectContext 메소드

메소드	설명
SetComplete	객체가 트랜잭션에 대한 해당 작업을 성공적으로 완료했음을 나타냅니다. 컨텍스트에 처음 진입한 메소드에서 반환되는 즉시 객체가 비활성화됩니다. 객체를 실행해야 하는 다음 호출에서 객체가 다시 활성화됩니다.
SetAbort	객체의 작업을 커밋할 수 없으며 트랜잭션을 롤백해야 함을 나타냅니다. 컨텍스트에 처음 진입한 메소드에서 반환되는 즉시 객체가 비활성화됩니다. 객체를 실행해야 하는 다음 호출에서 객체가 다시 활성화됩니다.
EnableCommit	객체의 작업이 반드시 수행될 필요는 없지만 객체의 트랜잭션 업데이트를 객체의 현재 폼으로 커밋할 수 있음을 나타냅니다. 트랜잭션을 완료하도록 허용하면서 클라이언트에서 여러 번 호출하더라도 상태를 유지하도록 할 때 이 메소드를 사용합니다. 객체에서 SetComplete나 SetAbort를 호출할 때까지 객체가 활성화되지 않습니다. 객체를 활성화한 경우 EnableCommit이 디폴트 상태입니다. 이 때문에 객체에서 클라이언트의 다음 호출에 대한 내부 상태를 유지하지 않으면 *메소드에서 반환되기 전에 객체가 항상 SetComplete나 SetAbort를 호출*해야 합니다.
DisableCommit	객체의 작업이 일관성이 없고 객체가 클라이언트에서 다른 메소드 호출을 받을 때까지 작업을 완료할 수 없음을 나타냅니다. 현재 트랜잭션을 활성 상태로 유지하면서 여러 클라이언트 호출에서 상태를 가지고 있으려면 컨트롤을 클라이언트에 돌려주기 전에 이 메소드를 호출합니다. DisableCommit은 메소드 호출에서 반환 시 객체를 비활성화하거나 객체의 리소스를 릴리스하지 못하게 합니다. 객체에서 DisableCommit을 호출한 경우 객체에서 EnableCommit이나 SetComplete를 호출하기 전에 클라이언트에서 트랜잭션을 커밋하면 트랜잭션이 중지됩니다.

트랜잭션 초기화

트랜잭션은 다음 세 가지 방법으로 제어할 수 있습니다.

- 클라이언트에서 트랜잭션을 제어할 수 있습니다.

  클라이언트는 트랜잭션 컨텍스트 객체를 사용하여(*ITransactionContext* 인터페이스 사용) 트랜잭션을 직접 제어할 수 있습니다.

- 서버에서 트랜잭션을 제어할 수 있습니다.

  서버에서 트랜잭션에 대한 객체 컨텍스트를 명시적으로 작성하여 트랜잭션을 제어할 수 있습니다. 서버에서 이런 식으로 객체를 만들면 만들어진 객체는 자동으로 현재 트랜잭션에 참여합니다.

- 객체의 트랜잭션 어트리뷰트(attribute) 결과로 트랜잭션이 자동으로 발생할 수 있습니다.

  객체를 만든 방법과 상관없이 객체가 항상 트랜잭션 내에서 실행되도록 트랜잭션 객체를 선언할 수 있습니다. 즉, 객체는 트랜잭션을 처리할 로직을 포함할 필요가 없으며 이로 인해 클라이언트 애플리케이션의 부담도 줄어듭니다. 클라이언트가 사용하는 컴포넌트에서 요구한다는 이유만으로 클라이언트가 트랜잭션을 초기화할 필요는 없습니다.

클라이언트에 트랜잭션 객체 설정

클라이언트 기반 애플리케이션에서 *ITransactionContextEx* 인터페이스를 통해 트랜잭션 컨텍스트를 제어할 수 있습니다. 다음 코드 예제에서는 클라이언트 애플리케이션에서 *CreateTransactionContextEx*를 사용하여 트랜잭션 컨텍스트를 만드는 방법을 보여 줍니다. 이 메소드는 이 객체로 인터페이스를 반환합니다.

```
#include <vcl\mtshlpr.h>

int main(int argc, char* argv[])
{
  // first, create a transactional object. [requires COM+ or local
installation of MTS]
  TCOMITransactionClientExample first_client =
CoTransactionClientExample::Create();
  //then, check to see if it's in a transaction.
  ITransactionContext* TCTX;
  HRESULT happily_transacting = first_client-
>QueryInterface(IID_ITransactionContext,
    (void**)&TCTX);
  // if it is in a transaction, you can do your data access calls from
here
  // and then call Commit or Abort yourself instead of waiting for the
  // transactional object to do so.
  if (happily_transacting)
  {
    TVariant database = "name";
    TVariant record = "data";
    TVariant flag;
    first_client.UpdateData(&database, &record, &flag);
    flag ? TCTX->Commit() : TCTX->Abort();
  }
  else
  {
    // otherwise, you can create a transaction:
    ITransactionContextEx* TXCTX  = CreateTransactionContextEx();
    // and an object. any objects you create in this fashion
    // will be enlisted in the transaction represented by this object.
    TCOMITransactionClientExample* second_client;
    TXCTX->CreateInstance(CLSID_TransactionClientExample,
IID_ITransactionClientExample,
      (void**)&second_client);
    // and then perform your data access and commit or abort.
    TVariant database = "name";
    TVariant record = "data";
    TVariant flag;
    second_client->UpdateData(&database, &record, &flag);
    flag ? TXCTX->Commit() : TXCTX->Abort();
  }
  return 0;
}
```

서버에 트랜잭션 객체 설정

서버에서 트랜잭션 컨텍스트를 제어하려면 *ObjectContext*의 인스턴스를 만듭니다. 다음 예제에서 Transfer 메소드는 트랜잭션 객체 안에 있습니다. 이런 방식으로 ObjectContext를 사용할 경우 만든 객체의 인스턴스가 ObjectContext를 만든 객체의 모든 트랜잭션 어트리뷰트(attribute)를 상속받습니다.

```
#include <vcl\mtshlpr.h>

STDMETHODIMP TTransactionServerExampleImpl::DoTransactionContext(long
execflag)
{
  if (m_spObjectContext->IsInTransaction())
  {
    // this means the current object has a transaction, and can pass
    // its transaction information to its children.
    // for simplicity, this object simply creates another object of its
    // own type, within the same transaction.
    // NOTE: you are still responsible for aggregating, if appropriate;
    if (execflag)
    {
      TCOMITransactionServerExample* inner;
      m_spObjectContext->CreateInstance(CLSID_TransactionServerExample,
                                        IID_ITransactionServerExample,
(void**)&inner);
      inner->DoTransactionContext(false);
      // add data access code here. data_access_succeeded() below is
      // an unimplemented placeholder.
      data_access_succeeded() ? m_spObjectContext->EnableCommit()
                              : m_spObjectContext->DisableCommit();
    }
  }
 else
  {
    //this means the current object has no transaction, and must
    //create one the way a client would.
    ITransactionContextEx* TCTX = CreateTransactionContextEx();
    TCOMITransactionServerExample* inner;
    // afterwards, follow the same steps.
    TCTX->CreateInstance(CLSID_TransactionServerExample,
                         IID_ITransactionServerExample, (void**)&inner);
    inner->DoTransactionContext(true);
    // add data access code here. data_access_succeeded() below is
    // an unimplemented placeholder.
    data_access_succeeded() ? TCTX->Commit() : TCTX->Abort();
  }
}
```

트랜잭션 제한 시간

트랜잭션 제한 시간은 트랜잭션을 활성화할 수 있는 시간(초)을 설정합니다. 제한 시간 후에 활성화되어 있는 트랜잭션은 시스템에서 자동으로 중지합니다. 디폴트로, 제한 시간은 60초입니다. 0값을 지정하여 트랜잭션 제한 시간을 사용하지 않을 수 있으며, 이는 트랜잭션 객체를 디버깅할 경우 유용합니다.

다음과 같은 방법으로 시스템에 제한 시간 값을 설정합니다.

1 MTS Explorer나 COM+ Component Manager에서 Computer, My Computer를 선택합니다.

디폴트로, My Computer는 로컬 컴퓨터에 해당합니다.

2 마우스 오른쪽 버튼을 클릭하고 Properties를 선택한 다음 Options 탭을 선택합니다.

Options 탭을 사용하여 시스템의 트랜잭션 제한 시간 속성을 설정합니다.

3 트랜잭션 제한 시간을 사용하지 않으려면 제한 시간 값을 0으로 변경합니다.

4 OK를 클릭하여 설정을 저장합니다.

MTS 애플리케이션 디버깅에 대한 자세한 내용은 44-27페이지의 "트랜잭션 객체 디버깅 및 테스트"를 참조하십시오.

역할 기반 보안

MTS와 COM+에서는 사용자의 논리적 그룹에 역할을 할당하는 역할 기반 보안을 제공합니다. 예를 들어, 의료 정보 애플리케이션에서는 의사, 방사선 기사, 환자에 대한 역할을 정의할 수 있습니다.

역할을 할당하여 각 객체와 인터페이스에 대한 권한을 정의합니다. 예를 들어, 의사의 의료 애플리케이션에서는 의사만 모든 의료 기록을 볼 수 있는 권한이 있고, 방사선 기사는 X레이만 볼 수 있고, 환자는 자신의 의료 기록만 볼 수 있습니다.

일반적으로 애플리케이션 개발 중에 역할을 정의하고 각 MTS 패키지나 COM+ 애플리케이션에 대한 역할을 할당합니다. 그런 다음 애플리케이션을 배포할 때 특정 사용자에게 이러한 역할을 할당합니다. 관리자는 MTS Explorer나 COM+ Component Manager를 사용하여 역할을 구성할 수 있습니다.

전체 객체가 아니라 코드 블록에 대한 액세스를 제어하려면 *IObjectContext* 메소드인 *IsCallerInRole*을 사용하여 더 세부적으로 보안을 제공할 수 있습니다. 이 메소드는 보안이 활성화되어 있을 경우에만 작동하며, 보안 활성화 여부는 *IObjectContext* 메소드인 *IsSecurityEnabled*를 호출하여 확인할 수 있습니다. 예를 들면, 다음과 같습니다.

```
if (m_spObjectContext.IsSecurityEnabled()) // check if security is enabled
{
  if (!m_spObjectContext.IsCallerInRole("Physician")) // check caller's role
  { // If not a physician, do something appropriate here.
  }
  else
  { // execute the call normally
  }
}
```

```
else // no security enabled
{ // do something appropriate
}
```

참고 더 강력한 보안이 필요한 애플리케이션의 경우 컨텍스트 객체는 *ISecurityProperty* 인터페이스를 구현합니다. 이 인터페이스의 메소드를 사용하면 객체를 사용하는 클라이언트의 보안 식별자(SID)뿐만 아니라 직접 호출자와 객체 작성자의 윈도우 보안 식별자(SID)를 검색할 수 있습니다.

트랜잭션 객체 작성 개요

트랜잭션 객체를 만드는 과정은 다음과 같습니다.

1 Transactional Object 마법사를 사용하여 트랜잭션 객체를 만듭니다.

2 Type Library Editor를 사용하여 객체의 인터페이스에 메소드와 속성을 추가합니다. Type Library Editor를 사용하여 메소드와 속성을 추가하는 것에 대한 자세한 내용은 39장, "타입 라이브러리 사용"을 참조하십시오.

3 객체의 메소드를 구현할 때 *IObjectContext* 인터페이스를 사용하여 트랜잭션, 영구적 상태 및 보안을 관리할 수 있습니다. 그리고 객체 참조를 전달할 경우 제대로 처리되도록 특별한 주의를 기울여야 합니다. 25페이지의 "객체 참조 전달"을 참조하십시오.

4 트랜잭션 객체를 디버그하고 테스트합니다.

5 트랜잭션 객체를 MTS 패키지나 COM+ 애플리케이션에 설치합니다.

6 MTS Explorer나 COM+ Component Manager를 사용하여 객체를 관리합니다.

Transactional Object 마법사 사용

MTS나 COM+에서 제공하는 리소스 관리, 트랜잭션 처리 및 역할 기반 보안을 이용할 수 있는 COM 객체를 만들려면 Transactional Object 마법사를 사용합니다.

다음과 같은 방법으로 Transactional Object 마법사를 표시합니다.

1 File | New | Other를 선택합니다.

2 ActiveX 탭을 선택합니다.

3 Transactional Object 아이콘을 더블 클릭합니다.

마법사에서 다음을 지정해야 합니다.

- 클라이언트 애플리케이션에서 객체의 인터페이스를 호출할 수 있는 방법을 나타내는 스레드 모델. 스레드 모델은 객체를 등록하는 방법을 결정합니다. 객체의 구현이 선택한 모델을 따르는지 확인해야 합니다. 스레드 모델에 대한 자세한 내용은 44-19페이지의 "트랜잭션 객체에 대한 스레드 모델 선택"을 참조하십시오.
- 트랜잭션 모델
- 객체가 클라이언트에게 이벤트를 통지하는지 여부 표시. 일반적인 이벤트에 대해서만 이벤트 지원이 제공되고 COM+ 이벤트에 대해서는 제공되지 않습니다.

이 과정을 마치면 트랜잭션 객체에 대한 정의가 포함된 현재 프로젝트에 새로운 유닛이 추가됩니다. 그리고 마법사에서 프로젝트에 타입 라이브러리를 추가하고 Type Library Editor에서 이를 엽니다. 이제 타입 라이브러리를 통해 인터페이스의 속성과 메소드를 노출할 수 있습니다. 41-9페이지의 "COM 객체 인터페이스 정의"에서 설명한 대로 COM 객체를 정의하는 것처럼 인터페이스를 정의합니다.

트랜잭션 객체에서는 **이중 인터페이스**를 구현합니다. 이중 인터페이스는 vtable을 통한 초기 연결(컴파일 타임)과 *IDispatch* 인터페이스를 통한 후기 연결(런타임)을 모두 지원합니다.

생성된 트랜잭션 객체에서는 *IObjectControl* 인터페이스 메소드 *Activate*, *Deactivate* 및 *CanBePooled*를 구현합니다.

반드시 Transactional Object 마법사를 사용할 필요는 없습니다. Type Library Editor의 COM+ 페이지를 사용한 다음 객체를 MTS 패키지나 COM+ 애플리케이션에 설치하여 Automation 객체를 COM+ 트랜잭션 객체로 변환하고 in-process Automation 객체를 MTS 트랜잭션 객체로 변환할 수 있습니다. 그러나 Transactional Object 마법사에서는 다음과 같은 장점을 제공합니다.

- 마법사는 자동으로 *IObjectControl* 인터페이스를 구현하고 *OnActivate* 및 *OnDeactivate* 이벤트를 객체에 추가하여 객체를 활성화하거나 비활성화할 때 응답하는 이벤트 핸들러를 만들 수 있습니다.
- 마법사는 자동으로 m_spObjectContext 멤버를 생성하므로 객체에서 *IObjectContext* 메소드를 쉽게 액세스하여 활성화와 트랜잭션을 제어할 수 있습니다.

트랜잭션 객체에 대한 스레드 모델 선택

MTS 런타임 환경이나 COM+에서 스레드를 관리합니다. 트랜잭션 객체에서는 스레드를 만들지 말아야 하고 DLL로 호출하는 스레드도 종료하지 말아야 합니다.

Transactional Object 마법사를 사용하여 스레드 모델을 지정한 경우 메소드 실행을 위해 스레드에 객체를 할당하는 방법을 지정해야 합니다.

표 44.2 트랜잭션 객체의 스레드 모델

스레드 모델	설명	구현의 장단점
Single	스레드가 지원되지 않습니다. 클라이언트 요청이 호출 메커니즘별로 serialize됩니다. 단일 스레드 컴포넌트의 모든 객체는 메인 스레드에서 실행됩니다. Threading Model Registry 어트리뷰트(attribute)가 없는 컴포넌트나 reentrant가 아닌 COM 컴포넌트에 사용하는 디폴트 COM 스레드 모델과 호환됩니다. 컴포넌트의 모든 객체와 프로세스의 모든 컴포넌트에서 메소드 실행이 serialize됩니다.	컴포넌트에서 reentrant가 아닌 라이브러리를 사용할 수 있습니다. 확장성이 매우 제한됩니다. 단일 스레드의 stateful 컴포넌트는 교착 상태에 빠지기 쉽습니다. stateless 객체를 사용하고 메소드에서 반환되기 전에 SetComplete를 호출하여 이 문제를 제거할 수 있습니다.

표 44.2 트랜잭션 객체의 스레드 모델(계속)

스레드 모델	설명	구현의 장단점
Apartment (또는 Single-threaded Apartment)	각각의 객체는 객체 수명동안 지속되는 스레드 Apartment에 할당되지만 여러 객체에 여러 스레드를 사용할 수 있습니다. 이것이 바로 표준 COM 동시성 모델입니다. 각 Apartment는 특정 스레드에 연결되어 있고 Windows 메시지 펌프를 갖고 있습니다.	Single 스레드 모델에 비해 동시성이 크게 향상됩니다. 같은 작업에 포함되어 있지 않으면 두 객체를 동시에 실행할 수 있습니다. 여러 프로세스에 객체를 분산시킬 수 있다는 점만 제외하면 COM Apartment와 유사합니다.
Both	클라이언트에 대한 콜백을 serialize 하는 점만 제외하고 Apartment와 동일합니다.	Apartment와 똑같은 장점을 갖고 있습니다. 객체 풀링을 사용할 경우에는 이 모델이 필요합니다.

참고 이 스레드 모델은 COM 객체에서 정의한 스레드 모델과 유사합니다. 그러나 MTS와 COM+는 스레드에 대한 기본 지원을 추가로 제공하기 때문에 여기서 각 스레드 모델의 의미에는 차이가 있습니다. 또한, 작업에 대한 기본 지원 때문에 Free 스레드 모델은 트랜잭션 객체에 적용되지 않습니다.

작업

스레드 모델 외에도 트랜잭션 객체는 **작업**을 통해 동시성을 확보합니다. 작업은 객체의 컨텍스트에 기록되고, 객체와 작업 사이의 연결은 변경할 수 없습니다. 작업에는 기본 클라이언트에서 생성한 트랜잭션 객체와 해당 객체 및 자손에서 만든 트랜잭션 객체도 포함됩니다. 이 객체를 하나 이상의 컴퓨터에서 실행하는 하나 이상의 프로세스에 분산시킬 수 있습니다.

예를 들어, 의사의 의료 애플리케이션에는 다른 객체로 표시되는 다양한 의료 데이터베이스에 업데이트를 추가하고 레코드를 제거하기 위한 트랜잭션 객체가 있을 수 있습니다. 이 레코드 객체에서는 트랜잭션을 기록하기 위한 영수증 객체와 같은 다른 객체도 사용할 수 있습니다. 그러면 기본 클라이언트가 직접 또는 간접적으로 제어하는 여러 트랜잭션 객체가 만들어집니다. 이 객체는 모두 같은 작업에 속합니다.

MTS나 COM+는 각 작업을 통한 실행 흐름을 추적하여 부주의한 병렬 처리로 인해 애플리케이션 상태가 손상되는 것을 방지합니다. 이 기능은 잠재적으로 분산된 객체의 컬렉션에 걸쳐 하나의 논리 스레드를 만듭니다. 하나의 논리 스레드를 가질 경우 애플리케이션을 훨씬 더 쉽게 작성할 수 있습니다.

트랜잭션 컨텍스트 객체나 객체 컨텍스트를 사용하여 기존 컨텍스트에서 트랜잭션 객체를 만들 경우 새로운 객체는 동일한 작업의 멤버가 됩니다. 즉, 새로운 컨텍스트는 컨텍스트를 만들기 위해 사용한 컨텍스트의 작업 식별자를 상속받습니다.

하나의 작업 내에서는 하나의 논리적 실행 스레드만 허용됩니다. 여러 프로세스에 객체를 분산시킬 수 있다는 점만 제외하면 COM Apartment 스레드 모델과 작동 방법이 유사합니다. 기본 클라이언트에서 작업을 호출하면 작업 내에서의 다른 모든 작업 요청(예: 다른 클라이언트 스레드로부터의 요청)은 초기 실행 스레드가 다시 클라이언트로 돌아올 때까지 차단됩니다.

MTS에서 모든 트랜잭션 객체는 하나의 작업에 속합니다. COM+에서는 **호출 동기화**를 설정하여 객체가 작업에 참여하는 방법을 구성할 수 있습니다. 다음과 같은 옵션을 사용할 수 있습니다.

표 44.3 호출 동기화 옵션

옵션	의미
Disabled	COM+는 객체에 작업을 할당하지 않지만 호출자의 컨텍스트를 사용하여 작업을 상속받을 수 있습니다. 호출자에게 트랜잭션이나 객체 컨텍스트가 없으면 객체가 작업에 할당되지 않으므로 객체를 COM+ 애플리케이션에 설치하지 않은 것과 동일한 결과가 됩니다. 애플리케이션에 리소스 관리자를 사용하는 객체가 있거나 객체가 트랜잭션 또는 just-in-time 활성화를 지원하는 경우에는 이 옵션을 사용하지 마십시오.
Not Supported	COM+에서는 호출자의 상태와 상관없이 객체를 작업에 할당하지 않습니다. 애플리케이션에 리소스 관리자를 사용하는 객체가 있거나 객체가 트랜잭션 또는 just-in-time 활성화를 지원하는 경우에는 이 옵션을 사용하지 마십시오.
Supported	COM+에서는 호출자와 같은 작업에 객체를 할당합니다. 호출자가 작업에 속하지 않으면 객체도 작업에 속하지 않습니다. 애플리케이션에 리소스 관리자를 사용하는 객체가 있거나 객체가 트랜잭션 또는 just-in-time 활성화를 지원하는 경우에는 이 옵션을 사용하지 마십시오.
Required	COM+에서는 항상 작업에 객체를 할당하고 필요하면 객체를 만듭니다. 트랜잭션 어트리뷰트 (attribute) 가 Supported 또는 Required 일 경우에는 이 옵션을 사용해야 합니다.
Requires New	COM+에서는 항상 새 작업에 객체를 할당하고 새 작업은 호출자의 작업과 다릅니다.

COM+에서 이벤트 생성

ActiveX 스크립팅 엔진이나 ActiveX 컨트롤과 같은 대부분의 COM 기반 기술에서는 이벤트 싱크와 COM의 연결 지점 인터페이스를 사용하여 이벤트를 생성합니다. 이벤트 싱크와 연결 지점은 밀접하게 연결된 이벤트 모델의 예입니다. 그런 모델에서 이벤트를 생성하는 애플리케이션(COM+ 용어로는 퍼블리셔, 이전 COM 용어로는 싱크)은 이벤트에 응답하는 애플리케이션(구독자)을 알고 있고, 그 반대도 마찬가지입니다. 퍼블리셔와 구독자의 수명은 일치하고 동시에 활성화되어야 합니다. 구독자의 컬렉션과 구독자에게 이벤트 발생 시기를 알려주는 메커니즘을 퍼블리셔에 유지하고 구현해야 합니다.

COM+에서는 이벤트를 관리하기 위한 새로운 시스템을 소개합니다. 각 구독자에 대한 관리 및 통지를 퍼블리셔가 처리하게 하지 않고 기초가 되는 시스템(COM+)에서 이 프로세스를 맡습니다. COM+ Event 모델은 느슨하게 연결되어 있기 때문에 퍼블리셔와 구독자는 서로 독립적으로 개발, 배포 및 활성화될 수 있습니다.

COM+ 이벤트 모델에서 퍼블리셔와 구독자 간의 통신을 크게 단순화했지만 퍼블리셔와 구독자 사이에 있는 새로운 레이어의 소프트웨어를 관리해야 하는 추가적인 작업도 생겼습니다. 이벤트와 구독자에 대한 정보는 이벤트 저장소라고 하는 COM+ Catalog 의 한 부분에 보관됩니다. Component Services 관리자같은 도구를 사용하여 이러한 관리 작업을 수행합니다. Component Services 도구는 스크립팅이 가능하므로 대부분의 관리 작업을 자동화할 수 있습니다. 예를 들어, 설치 스크립트에서 실행 중에 이 작업을 수행할 수 있습니다. 그리고 이벤트 저장소는 *TComAdminCatalog* 객체를 사용하여 프로그램 방식으로 관리할 수 있습니다.

Run | Install COM+ objects를 선택하여 C++Builder에서 직접 COM+ 컴포넌트를 설치할 수도 있습니다.

밀접하게 연결된 이벤트 모델을 사용하면 이벤트는 단순히 인터페이스의 메소드일 뿐입니다. 그러므로 먼저 이벤트 메소드에 대한 인터페이스를 만들어야 합니다. C++ Builder의 COM+ Event Object 마법사를 사용하여 COM+ 이벤트 객체가 포함된 프로젝트를 만들 수 있습니다. 그런 다음 Component Services 관리 도구(또는 *TComAdminCatalog*나 IDE)를 사용하여 이벤트 클래스 컴포넌트를 갖고 있는 COM+ 애플리케이션을 만듭니다. COM+ 애플리케이션에 이벤트 클래스 컴포넌트를 만들 경우 이벤트 객체를 선택합니다. 이벤트 클래스는 COM+에서 퍼블리셔를 구독자 리스트에 연결하기 위해 사용하는 연결점입니다.

COM+ 이벤트 객체에 관한 흥미로운 점은 이벤트 인터페이스의 구현을 포함하지 않는다는 것입니다. COM+ 이벤트 객체에서는 퍼블리셔와 구독자가 통신하기 위해 사용할 인터페이스를 정의합니다. C++Builder를 사용하여 COM+ 이벤트 객체를 만들 경우 Type Library Editor를 사용하여 인터페이스를 정의합니다. COM+ 애플리케이션과 해당하는 이벤트 클래스 컴포넌트를 만들 때 인터페이스가 구현됩니다. 그러면 이벤트 클래스에는 참조가 포함되고 COM+에서 제공하는 구현에 대한 액세스를 제공합니다. 런타임 시 퍼블리셔는 일반적인 COM 메커니즘(예: CoCreateInstance)을 사용하여 이벤트 클래스의 인스턴스를 작성합니다. 개발자 인터페이스의 COM+ 구현에서 퍼블리셔가 할 일은 이벤트 클래스의 인스턴스를 통해 인터페이스에서 메소드를 호출하여 모든 구독자에게 통지할 이벤트를 생성하는 것뿐입니다.

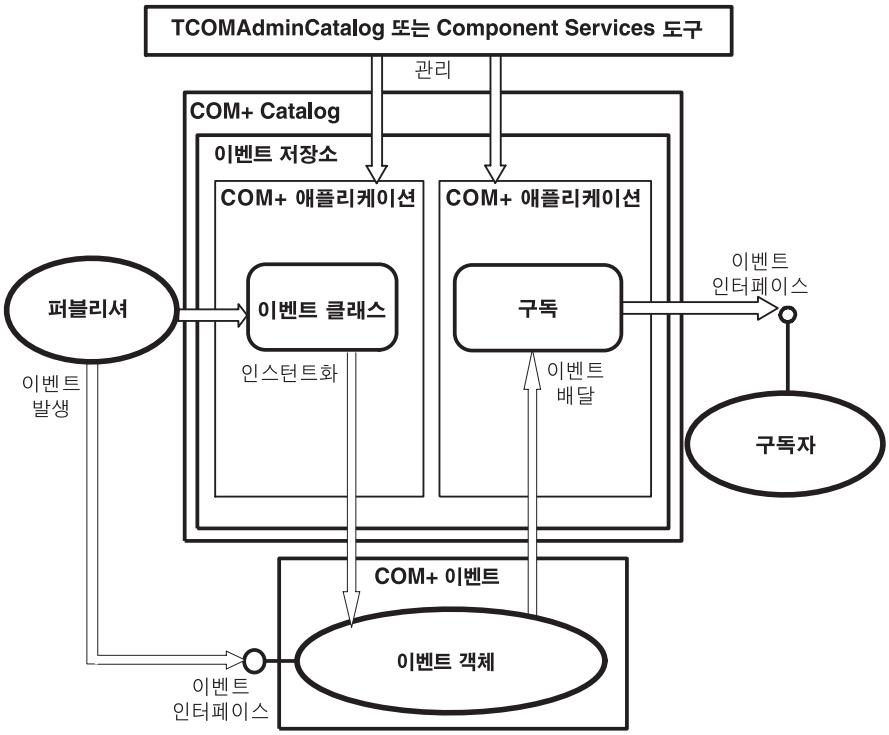
참고 퍼블리셔가 COM 컴포넌트일 필요는 없습니다. 퍼블리셔는 이벤트 클래스의 인스턴스를 작성하고 이벤트 인터페이스에서 메소드를 호출하여 이벤트를 작성하는 애플리케이션입니다.

구독자 컴포넌트도 COM+ Catalog에 설치해야 합니다. IDE인 *TComAdminCatalog*를 사용하거나 Component Services 관리 도구를 사용하여 프로그램 방식으로 이 작업을 수행할 수도 있습니다. 구독자 컴포넌트는 독립적인 COM+ 애플리케이션에 설치하거나 이벤트 클래스 컴포넌트를 포함시키기 위해 사용한 같은 애플리케이션에 설치할 수 있습니다. 컴포넌트를 설치한 후 컴포넌트에서 지원하는 모든 이벤트 인터페이스에 대한 구독을 작성해야 합니다. 구독을 작성한 후 컴포넌트에서 수신 대기할 이벤트 클래스, 즉 퍼블리셔를 선택합니다. 구독자 컴포넌트에서 개별 이벤트 클래스나 모든 이벤트 클래스를 선택할 수 있습니다.

COM+ 이벤트 객체와는 달리 COM+ 구독 객체에는 자신만의 이벤트 인터페이스 구현이 포함되어 있으며 이벤트가 생성되었을 때 이벤트에 응답하기 위한 실제 작업이 여기에서 수행됩니다. C++ Builder의 COM+ Event Subscription Object 마법사를 사용하여 구독자 컴포넌트가 포함된 프로젝트를 만들 수 있습니다.

다음 그림에서는 퍼블리셔, 구독자, COM+ Catalog 사이의 상호 작용을 보여 줍니다.

그림 44.1 COM+ Event 시스템



Event Object 마법사 사용

Event Object 마법사를 사용하여 이벤트 객체를 만들 수 있습니다. 이 마법사에서는 먼저 현재 프로젝트에 구현 코드가 포함되어 있는지 확인합니다. COM+ 이벤트 객체를 포함하는 프로젝트에는 구현이 포함되지 않기 때문입니다. 이벤트 객체 정의만 포함할 수 있습니다. 그러나 단일 프로젝트에 여러 개의 COM+ 이벤트 객체를 포함시킬 수 있습니다.

다음과 같은 방법으로 Event Object 마법사를 표시합니다.

1 File|New를 선택합니다.

2 ActiveX 탭을 선택합니다.

3 COM+ Event Object 아이콘을 더블 클릭합니다.

Event Object 마법사에서 이벤트 객체 이름, 이벤트 핸들러를 정의하는 인터페이스 이름 및 이벤트에 대한 간단한 설명(옵션)을 지정합니다.

마법사를 끝내면 이벤트 객체와 해당 인터페이스를 정의하는 타입 라이브러리가 포함된 프로젝트가 만들어 집니다. Type Library Editor를 사용하여 해당 인터페이스의 메소드를 정의합니다. 이 메소드는 이벤트에 응답하기 위해 클라이언트에서 구현한 이벤트 핸들러입니다.

이벤트 객체 프로젝트에는 프로젝트 파일, ATL 템플릿 클래스를 임포트하기 위한 _ATL 유닛, 타입 라이브러리 정보를 정의하기 위한 _TLB 유닛이 포함되어 있습니다. 그러나 COM+ 이벤트 객체에 구현이 없기 때문에 구현 유닛은 포함되지 않습니다. 인터페이스 구현은 클라이언트의 책임입니다. 서버 객체에서 COM+ 이벤트 객체를 호출할 경우 COM+에서 호출을 인터셉트하여 등록된 클라이언트로 디스패치합니다. COM+ 이벤트 객체에는 구현 객체가 필요하지 않기 때문에 Type Library Editor로 객체의 인터페이스를 정의한 후 프로젝트를 컴파일하고 COM+를 사용하여 프로젝트를 설치하기만 하면 됩니다.

COM+에는 이벤트 객체의 인스턴스에 대한 특정한 제한 사항이 있습니다. 이벤트 객체에 대해 Type Library Editor에서 정의한 인터페이스는 다음 규칙을 따라야 합니다.

- 이벤트 객체의 인터페이스는 IDispatch에서 파생되어야 합니다.
- 이벤트 객체의 모든 인터페이스에서 모든 메소드 이름이 고유해야 합니다.
- 이벤트 객체의 인터페이스에 있는 모든 메소드는 HRESULT 값을 반환해야 합니다.
- 메소드의 모든 매개변수에 대한 변경자가 [in]이어야 합니다.

COM+ Event Subscription Object 마법사

C++Builder의 COM+ Subscription Object 마법사를 사용하여 구독자 컴포넌트를 만들 수 있습니다. 다음과 같은 방법으로 마법사를 표시합니다.

1 File | New를 선택합니다.

2 ActiveX 탭을 선택합니다.

3 COM+ Subscription Object 아이콘을 더블 클릭합니다.

마법사 다이얼로그 박스에서 이벤트 인터페이스를 구현할 클래스 이름을 입력합니다. 콤보 박스에서 스레드 모델을 선택합니다. Interface 필드에서 이벤트 인터페이스 이름을 입력하거나 Browse 버튼을 클릭하여 현재 COM+ Catalog에 설치된 모든 이벤트 클래스 리스트를 표시할 수 있습니다. COM+ Event Interface Selection 다이얼로그 박스에도 Browse 버튼이 있습니다. 이 버튼을 사용하면 이벤트 인터페이스가 포함된 타입 라이브러리를 찾아서 선택할 수 있습니다. 기존의 이벤트 클래스나 타입 라이브러리를 선택하면 마법사는 기존의 이벤트 클래스에서 지원하는 인터페이스를 자동으로 구현할 수 있는 옵션을 제공합니다. Implement Existing Interface 체크 박스를 선택하면 마법사에서는 해당 인터페이스의 메소드를 자동으로 구현합니다. Implement Ancestor Interfaces 체크 박스를 선택하여 상속받은 인터페이스를 마법사에서 구현하도록 선택할 수 있습니다. 마법사에서는 *IUnknown*, *IDispatch* 및 *IAppServer* 같은 세 개의 조상 인터페이스는 구현하지 않습니다. 마법사를 완료하려면 이벤트 구독자 컴포넌트에 대한 간단한 설명을 입력하고 OK를 클릭합니다.

COM+ 이벤트 객체를 사용하여 이벤트 발생

이벤트를 발생시키려면 퍼블리셔는 일반적인 COM 메커니즘(예: CoCreateInstance)을 사용하여 이벤트 클래스의 인스턴스를 먼저 만듭니다. 이벤트 클래스에는 이벤트 인터페이스의 구현이 포함되어 있으므로 이벤트를 생성하는 것은 인터페이스에서 해당하는 메소드를 호출하는 것과 같습니다.

COM+ Event 시스템이 이후 작업을 수행합니다. 이벤트 메소드를 호출하면 시스템에서 COM+ Catalog의 모든 구독자를 알아내고 각 구독자에게 통지합니다. 구독자 쪽에서는 이벤트가 이벤트 메소드에 대한 호출로 나타납니다.

퍼블리셔가 이벤트를 생성하면 구독자에게 한 번에 하나씩 동시에 통지됩니다. 통지 순서를 지정할 수 있는 방법은 없으며 이벤트가 발생할 때마다 순서가 같도록 지정할 수도 없습니다. 이벤트 클래스를 COM+ Catalog에 설치하면 관리자는 *FireInParallel* 옵션을 선택하여 이벤트를 여러 스레드를 사용하여 전달하도록 요청할 수 있습니다. 동시에 전달되는지는 보장할 수 없으며 이는 이벤트를 전달할 수 있도록 시스템에게 허가를 요청하는 것에 불과합니다.

퍼블리셔에게 반환된 값은 각 구독자의 모든 반환 코드를 집계한 것입니다. 퍼블리셔가 오류가 발생한 구독자를 직접 찾을 수 있는 방법은 없습니다. 그렇게 하려면 퍼블리셔가 퍼블리셔 필터를 구현해야 합니다. 이 항목에 대한 자세한 내용은 Microsoft MSDN 설명서를 참조하십시오. 다음 표에서는 가능한 반환 코드를 요약합니다.

표 44.4 이벤트 퍼블리셔 반환 코드

반환 코드	의미
S_OK	모든 구독자가 성공했습니다.
EVENT_S_SOME_SUBSCRIBERS_FAILED	일부 구독자를 호출할 수 없거나 일부 구독자가 오류 코드를 반환했습니다. 오류 조건은 아닙니다.
EVENT_E_ALL_SUBSCRIBERS_FAILED	구독자를 호출할 수 없거나 모든 구독자가 오류 코드를 반환했습니다.
EVENT_S_NOSUBSCRIBERS	COM+ Catalog 에 구독이 없습니다 . 오류 조건은 아닙니다.

객체 참조 전달

참고 객체 참조 전달에 대한 내용은 MTS에만 적용되고 COM+에는 적용되지 않습니다. MTS에서 실행되는 객체의 모든 포인터가 인터셉터를 통해 라우트되야 하기 때문에 MTS에서는 이 메커니즘이 필요합니다. COM+의 경우에는 인터셉터가 내장되기 때문에 객체 참조를 전달할 필요가 없습니다.

MTS에서는 다음과 같은 방법으로만 객체 참조를 전달(예: 콜백으로 사용하기 위해)할 수 있습니다.

- *CoCreateInstance*(또는 이와 동등한 것), *ITransactionContext::CreateInstance* 또는 *IObjectContext::CreateInstance*와 같은 객체 생성 인터페이스에서 반환하여 전달
- *QueryInterface*에 대한 호출을 통해 전달
- 객체 참조를 가져오기 위해 *SafeRef*를 호출한 메소드를 통해 전달

위와 같은 방법으로 획득한 객체 참조를 **안전한 참조**라고 합니다. 안전한 참조를 사용하여 호출한 메소드는 해당 컨텍스트 내에서 실행됩니다.

MTS 런타임 환경에서는 호출에서 안전한 참조를 사용해야 합니다. 그러면 컨텍스트 스위치를 관리할 수 있고 트랜잭션 객체가 클라이언트 참조와는 별개인 수명을 가질 수 있습니다. COM+에서는 반드시 안전한 참조를 사용할 필요는 없습니다.

SafeRef 메소드 사용

객체는 *SafeRef* 함수를 사용하여 컨텍스트 외부에서 전달하기에 안전한 객체 자신에 대한 참조를 가져올 수 있습니다. 이 함수를 *TMtsDll* 객체의 메소드로 사용할 수 있으며, 이 메소드에서는 서버가 MTS나 COM+에서 실행되는지 여부와 해당하는 포인터를 제대로 반환하는지 여부를 확인합니다.

*SafeRef*는 다음을 입력으로 받아들입니다.

- 현재 객체가 다른 객체나 클라이언트에 전달할 인터페이스의 인터페이스 ID(RIID)에 대한 참조
- 현재 객체의 IUnknown 인터페이스에 대한 참조

*SafeRef*는 현재 객체의 컨텍스트 외부에서 전달하기에 안전한 RIID 매개변수로 지정한 인터페이스의 포인터를 반환합니다. 객체가 자신이 아닌 다른 객체에 대한 안전한 참조를 요구하거나 RIID 매개변수에서 요청한 인터페이스를 구현하지 않은 경우 NULL을 반환합니다.

MTS 객체에서 자체 참조를 클라이언트나 다른 객체에 전달할 경우(예를 들어, 콜백으로 사용하기 위해) 항상 *SafeRef*를 먼저 호출한 다음 이 호출에서 반환한 참조를 전달합니다. 객체는 **self** 포인터나 QueryInterface에 대한 내부 호출을 통해 확보한 자체 참조를 클라이언트나 다른 객체에 전달하지 말아야 합니다. 이런 참조를 객체 컨텍스트 외부에 전달하면 더 이상 유효한 참조가 아닙니다.

이미 안전한 참조에서 *SafeRef*를 호출하면 바뀌지 않은 안전한 참조가 반환됩니다. 그러나 인터페이스의 참조 개수는 증가합니다.

클라이언트가 안전한 참조에서 QueryInterface를 호출하면 클라이언트에 반환된 참조도 안전한 참조입니다.

안전한 참조를 가져온 객체는 사용이 끝났을 경우 안전한 참조를 릴리스해야 합니다.

*SafeRef*에 대한 자세한 내용은 Microsoft 설명서의 SafeRef 항목을 참조하십시오.

콜백

객체는 클라이언트와 다른 트랜잭션 객체에 대해 콜백을 수행할 수 있습니다. 예를 들어, 다른 객체를 만드는 객체가 있을 수 있습니다. 만든 객체는 자신에 대한 참조를 만들어진 객체에 전달할 수 있고, 만들어진 객체는 이 참조를 사용하여 만든 객체를 호출할 수 있습니다.

콜백을 사용하기로 선택한 경우 다음 제한 사항을 주의하십시오.

- 기본 클라이언트나 다른 패키지에 콜백하려면 클라이언트에 액세스 수준의 보안이 필요합니다. 그리고 클라이언트는 DCOM 서버여야 합니다.
- 방화벽을 끼워 넣으면 클라이언트에 대한 콜백이 차단될 수 있습니다.

- 콜백에서 수행한 작업은 호출된 객체의 환경에서 실행됩니다. 같은 트랜잭션이나 다른 트랜잭션의 일부일 수도 있고 어떤 트랜잭션에도 속하지 않을 수도 있습니다.
- MTS에서 객체를 만들려면 *SafeRef*를 호출한 다음 반환된 참조를 만들어진 객체로 전달하여 자신에게 콜백되게 합니다.

트랜잭션 객체 디버깅 및 테스트

로컬 및 원격 트랜잭션 객체를 디버깅할 수 있습니다. 트랜잭션 객체를 디버깅할 경우 트랜잭션 제한 시간을 해제할 수 있습니다.

트랜잭션 제한 시간은 트랜잭션을 활성화할 수 있는 기간(초)을 설정합니다. 제한 시간 후에도 여전히 활성화되어 있는 트랜잭션은 시스템에서 자동으로 중지시킵니다. 디폴트로, 제한 시간은 60초입니다. 0값을 지정하여 트랜잭션 제한 시간을 사용하지 않을 수있으며, 디버깅할 때 유용합니다.

원격 디버깅에 대한 자세한 내용은 온라인 도움말의 원격 디버깅 항목을 참조하십시오.

MTS 에서 실행할 트랜잭션 객체를 테스트할 경우 MTS 환경 외부에서 객체를 먼저 테스트하여 테스트 환경을 간단히 할 수 있습니다.

서버를 개발하는 동안 메모리에 있는 서버를 다시 생성할 수 없습니다. "Cannot write to DLL while executable is loaded"같은 컴파일러 오류가 발생할 수 있습니다. 이를 방지하려면 서버가 유휴 상태일 때 서버를 종료하도록 MTS 패키지나 COM+ 애플리케이션 속성을 설정할 수 있습니다.

다음과 같은 방법으로 서버가 유휴 상태일 때 서버를 종료하십시오.

1 MTS Explorer나 COM+ Component Manager에서 트랜잭션 객체가 설치된 MTS 패키지나 COM+ 애플리케이션을 마우스 오른쪽 버튼으로 클릭한 다음 Properties를 선택합니다.

2 Advanced 탭을 선택합니다.

  Advanced 탭에서는 패키지와 연관된 서버 프로세스가 항상 실행되는지 또는 특정 기간 후에 서버가 종료되는지 여부를 결정합니다.

3 제한 시간 값을 0으로 변경합니다. 그러면 처리할 클라이언트가 더 이상 없는 경우 바로 서버를 종료합니다.

4 OK를 클릭하여 설정을 저장합니다.

트랜잭션 객체 설치

MTS 애플리케이션은 MTS 실행 파일(EXE)의 단일 인스턴스에서 실행하는 in-process MTS 객체의 그룹으로 구성됩니다. 모두 같은 프로세스에서 실행되는 COM 객체 그룹을 **패키지**라고 합니다. 한 시스템에서 여러 개의 다른 패키지를 실행할 수 있습니다. 이 때, 패키지는 각각의 MTS EXE 내에서 실행됩니다.

COM+에서는 COM+ 애플리케이션이라는 유사한 그룹으로 작업합니다. **COM+ 애플리케이션**에서 객체는 in-process일 필요가 없으며 독립적인 런타임 환경이 없습니다.

애플리케이션 컴포넌트를 단일 프로세스에서 관리하는 단일 MTS 패키지나 COM+ 애플리케이션으로 그룹화할 수 있습니다. 여러 프로세스나 시스템에서 애플리케이션을 분할 작업하기 위해 컴포넌트를 다른 MTS 패키지나 COM+ 애플리케이션으로 분산할 수 있습니다.

다음과 같은 방법으로 트랜잭션 객체를 MTS 패키지나 COM+ 애플리케이션에 설치합니다.

1. 시스템에서 COM+를 지원할 경우 Run|Install COM+ objects를 선택합니다. 시스템에서 COM+를 지원하지 않지만 시스템에 MTS를 설치한 경우 Run|Install MTS objects를 선택합니다. 시스템에서 MTS나 COM+를 지원하지 않을 경우 트랜잭션 객체 설치를 위한 메뉴 항목이 표시되지 않습니다.
2. Install Object 다이얼로그 박스에서 설치할 객체를 선택합니다.
3. MTS 객체를 설치할 경우 Package 버튼을 클릭하여 시스템의 MTS 패키지 리스트를 가져옵니다. COM+ 객체를 설치할 경우 Application 버튼을 클릭합니다. 객체를 설치하려는 MTS 패키지나 COM+ 애플리케이션을 표시합니다. 객체를 설치할 새로운 MTS 패키지나 COM+ 애플리케이션을 만들려면 Into New Package나 Into New Application을 선택합니다. 기존의 나열된 MTS 패키지나 COM+ 애플리케이션에 객체를 설치하려면 Into Existing Package나 Into Existing Application을 선택합니다.
4. OK를 클릭하여 카탈로그를 새로 고칩니다. 그러면 런타임 시 객체를 사용할 수 있습니다.

MTS 패키지에는 여러 DLL의 컴포넌트가 포함될 수 있고 단일 DLL의 컴포넌트를 다른 패키지에 설치할 수 있습니다. 그러나 여러 패키지 사이에 단일 컴포넌트를 분산시킬 수 없습니다.

마찬가지로 COM+ 애플리케이션에는 여러 실행 파일의 컴포넌트가 포함될 수 있고, 단일 실행 파일의 다른 컴포넌트들을 다른 COM+ 애플리케이션에 설치할 수 있습니다.

참고 COM+ Component Manager나 MTS Explorer를 사용하여 트랜잭션 객체를 설치할 수도 있습니다. 이 도구 중 하나를 사용하여 객체를 설치할 경우 Type Library Editor의 COM+ 페이지에 표시된 객체에 대해 설정을 적용하십시오. IDE에서 설치하지 않을 경우에는 이 설정이 자동으로 적용되지 않습니다.

트랜잭션 객체 관리

트랜잭션 객체를 설치했으면 MTS Explorer(MTS 패키지에 객체를 설치한 경우) 또는 COM+ Component Manager(COM+ 애플리케이션에 객체를 설치한 경우)를 사용하여 런타임 객체를 관리할 수 있습니다. MTS Explorer는 MTS 런타임 환경에서 작동하고 COM+ Component Manager는 COM+ 객체에서 작동한다는 점만 제외하고 이 두 도구는 동일합니다.

COM+ Component Manager와 MTS Explorer에는 트랜잭션 객체를 관리하고 배포하기 위한 그래픽 사용자 인터페이스가 있습니다. 이 도구 중 하나를 사용하여 다음을 할 수 있습니다.

- 트랜잭션 객체, MTS 패키지 또는 COM+ 애플리케이션, 역할을 구성합니다.
- 패키지나 COM+ 애플리케이션의 컴포넌트 속성을 보고 컴퓨터에 설치된 MTS 패키지나 COM+ 애플리케이션을 봅니다.
- 트랜잭션을 구성하는 객체의 트랜잭션을 모니터하고 관리합니다.
- 컴퓨터 간에 MTS 패키지나 COM+ 애플리케이션을 이동합니다.
- 원격 트랜잭션 객체를 로컬 클라이언트에서 사용 가능하게 합니다.

이 도구에 대한 자세한 내용은 Microsoft의 해당 *관리자 안내서*를 참조하십시오.

부

V

사용자 정의 컴포넌트 생성

"사용자 정의 컴포넌트 생성"에 포함된 장에서는 C++Builder 사용자 정의 컴포넌트를 디자인하고 구현하는 데 필요한 개념을 설명합니다.

장

45

컴포넌트 생성 개요

이 장에서는 C++Builder 애플리케이션용 컴포넌트 디자인 및 컴포넌트 작성 절차에 대한 개요를 제공합니다. 이 장에서는 여러분이 C++Builder 및 해당 표준 컴포넌트에 대해 잘 알고 있다고 가정합니다.

- 클래스 라이브러리
- 컴포넌트 및 클래스
- 컴포넌트 생성 방법
- 컴포넌트 생성 시 고려 사항
- 새 컴포넌트 생성
- 설치되지 않은 컴포넌트 테스트
- 설치된 컴포넌트 테스트
- 컴포넌트 팔레트에 컴포넌트 설치

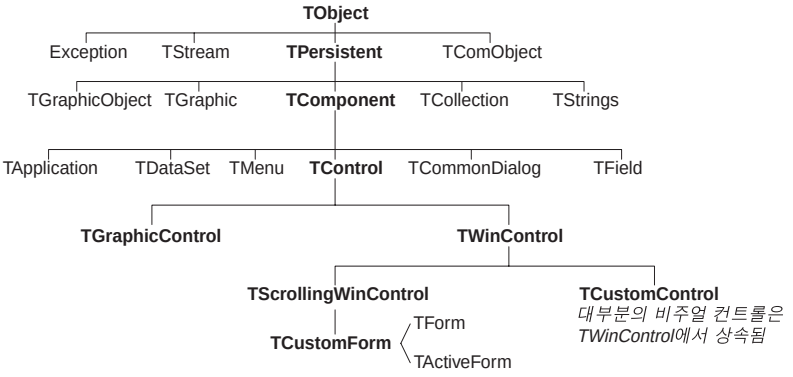
새 컴포넌트 설치에 대한 자세한 내용은 15-5페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

클래스 라이브러리

C++Builder의 컴포넌트는 비주얼 컴포넌트 라이브러리(VCL) 및 크로스 플랫폼 컴포넌트 라이브러리(CLX)라는 두 클래스 계층에 상주합니다. 그림 45.1은 VCL을 구성하는 선택된 클래스의 관계를 보여 줍니다. CLX 계층은 VCL과 유사하지만 Windows 컨트롤을 widget이라고 하고, *TWinControl*을 *TWidgetControl*이라고 하는 것을 비롯하여 그 밖에도 여러 가지 차이점이 있습니다. 클래스 계층에 대한 세부 사항 및 클래스 간의 상속 관계에 대한 자세한 내용은 46장, "컴포넌트 작성자를 위한 객체 지향 프로그래밍"을 참조하십시오. CLX와 VCL의 차이점에 대한 개요는 14-5페이지의 "CLX와 VCL 비교"를 참조하고 컴포넌트에 대한 자세한 내용은 CLX 온라인 참조를 사용하십시오.

TComponent 클래스는 VCL 및 CLX 내의 모든 컴포넌트의 공유 조상입니다. *TComponent*는 컴포넌트가 C++Builder에서 작업하는 데 필요한 최소한의 속성과 이벤트를 제공합니다. 다양한 라이브러리 분기는 특화된 다른 기능을 추가로 제공합니다.

그림 45.1 비주얼 컴포넌트 라이브러리 클래스 계층



컴포넌트를 만들 때는 계층 구조의 기존 클래스 타입 중 하나에서 새 클래스를 파생시켜 VCL 또는 CLX에 추가합니다.

컴포넌트 및 클래스

컴포넌트는 클래스이므로 컴포넌트 작성자는 애플리케이션 개발자와 다른 레벨에서 객체를 사용합니다. 새 컴포넌트를 생성하려면 새 클래스를 파생시켜야 합니다.

간단히 말해서, 컴포넌트를 생성하는 것과 애플리케이션에서 컴포넌트를 사용하는 것 사이에는 두 가지 중요한 차이점이 있습니다. 컴포넌트를 생성할 때는 다음을 수행합니다.

- 애플리케이션 프로그래머가 액세스할 수 없는 클래스의 부분에 액세스합니다.
- 속성과 같은 새로운 부분을 컴포넌트에 추가합니다.

이러한 차이점 때문에 더 많은 규칙을 알아야 하고 작성한 컴포넌트를 애플리케이션 개발자가 사용하는 방법에 대해 고려해야 합니다.

컴포넌트 생성 방법

컴포넌트는 디자인 타임에 처리하려는 거의 모든 프로그램 요소를 나타낼 수 있습니다. 컴포넌트 생성이란 기존 클래스에서 새 클래스를 파생시키는 것을 의미합니다. 다음과 같이 여러 가지 방법으로 새 컴포넌트를 파생시킬 수 있습니다.

- 기존 컨트롤 수정
- 윈도우 컨트롤 생성
- 그래픽 컨트롤 생성
- Windows 컨트롤 서브클래싱
- 넌비주얼(nonvisual) 컴포넌트 생성

표 45.1에는 각 작업에 대해 기본적으로 사용할 각기 다른 종류의 컴포넌트 및 클래스가 요약되어 있습니다.

표 45.1 컴포넌트 생성 방법

수행할 작업	사용할 타입
기존 컴포넌트 수정	*TButton* 또는 *TListBox* 같은 모든 기존 컴포넌트나 *TCustomListBox* 같은 추상 컴포넌트 타입
윈도우 컨트롤(CLX의 경우 widget 기반 컨트롤) 생성	*TWinControl* (CLX의 경우 *TWidgetControl*)
그래픽 컨트롤 생성	*TGraphicControl*
컨트롤 서브클래싱	모든 Windows(VCL) 또는 widget 기반(CLX) 컨트롤
넌비주얼 컴포넌트 생성	*TComponent*

또한 컴포넌트가 아니며 폼에서 처리할 수 없는 *TRegIniFile* 및 *TFont* 등의 클래스를 파생시킬 수 있습니다.

기존 컨트롤 수정

컴포넌트를 생성하는 가장 간단한 방법은 기존 컴포넌트를 사용자 정의하는 것입니다. C++Builder와 함께 제공되는 모든 컴포넌트에서 새 컴포넌트를 파생시킬 수 있습니다.

리스트 박스 및 그리드 등의 일부 컨트롤은 기본 테마를 여러 가지로 변형합니다. 이러한 경우, VCL 및 CLX에는 사용자 정의 버전을 파생시킨 추상 클래스(*TCustomGrid*와 같이 해당 이름에 "custom"이 포함됨)가 포함됩니다.

예를 들어, 표준 *TListBox* 클래스의 속성 중 일부를 갖지 않는 특수한 리스트 박스를 만들고자 하는 경우가 있을 것입니다. 그러나 조상 클래스로부터 상속받은 속성을 제거하거나 숨길 수 없으므로 계층에서 *TListBox* 보다 더 위에 속하는 클래스로부터 컴포넌트를 파생시켜야 합니다. VCL 또는 CLX에서 리스트 박스의 속성을 구현하지만 모든 속성을 게시하지는 않는 *TCustomListBox*가 제공되므로 *TWinControl*(또는 CLX의 *TWidgetControl*) 추상 클래스에서 시작하여 모든 리스트 박스 기능을 다시 구현하지 않아도 됩니다. *TCustomListBox*와 같은 추상 클래스로부터 컴포넌트를 파생시키는 경우 컴포넌트에서 사용하고자 하는 속성만 게시하고 나머지는 보호된 속성으로 두십시오.

47장, "속성 생성"에서는 상속된 속성 게시에 대해 설명합니다. 53장, "기존 컴포넌트 수정" 및 55장, "그리드 사용자 정의"에서는 기존 컨트롤을 수정하는 예를 보여 줍니다.

윈도우 컨트롤 생성

VCL 및 CLX에 있는 윈도우 컨트롤은 런타임 시 나타나며 사용자가 상호 작용할 수 있는 객체입니다. 모든 윈도우 컨트롤에는 *Handle* 속성을 통해 액세스할 수 있는 윈도우 핸들이 있으며 이를 통해 운영 체제는 컨트롤을 식별하고 조작할 수 있습니다. VCL 컨트롤을 사용하는 경우 핸들을 통해 컨트롤이 입력 포커스를 받고 Windows API 함수에 전달될 수 있습니다. CLX에서 이러한 컨트롤은 widget 기반의 컨트롤입니다. 모든 widget 기반 컨트롤에는 *Handle* 속성을 통해 액세스할 수 있는 핸들이 있으며 이를 통해 기초가 되는 widget을 식별할 수 있습니다.

모든 윈도우 컨트롤은 *TWinControl*(CLX의 경우 *TWidgetControl*) 클래스의 자손입니다. 여기에는 가장 표준적인 윈도우 컨트롤, 예를 들어, 누름 버튼, 리스트 박스, 에디트 박스 등이 포함됩니다. *TWinControl*(CLX의 경우 *TWidgetControl*)로부터 직접 기존 컨트롤과 연결되지 않은 원래 컨트롤을 파생시킬 수 있는 반면 C++Builder는 이 목적으로 *TCustomControl* 컴포넌트를 제공합니다. *TCustomControl*은 복잡한 비주얼 이미지를 보다 쉽게 그릴 수 있도록 해주는 특화된 윈도우 컨트롤입니다.

55장, "그리드 사용자 정의"에는 윈도우 컨트롤을 만드는 예제가 나와 있습니다.

그래픽 컨트롤 생성

컨트롤이 입력 포커스를 받을 필요가 없는 경우에는 컨트롤을 그래픽 컨트롤로 만들 수 있습니다. 그래픽 컨트롤은 윈도우 컨트롤과 유사하나 윈도우 핸들이 없으며 이로 인해 시스템 리소스를 거의 사용하지 않습니다. 입력 포커스를 받지 않는 *TLabel*과 같은 컴포넌트는 그래픽 컨트롤입니다. 이러한 컨트롤은 포커스를 받지 못하지만 마우스 메시지에 반응하도록 디자인할 수 있습니다.

C++Builder는 *TGraphicControl* 컴포넌트를 통해 사용자 정의 컨트롤의 생성을 지원합니다. *TGraphicControl*은 *TControl*에서 파생된 추상 클래스입니다. *TControl*에서 직접 컨트롤을 파생시킬 수도 있지만 그림을 그릴 캔버스를 제공하며 Windows의 경우는 *WM_PAINT* 메시지를 처리하는 *TGraphicControl*에서부터 시작하는 것이 더 좋습니다. 개발자는 *Paint* 메소드를 오버라이드하기만 하면 됩니다.

54장, "그래픽 컴포넌트 생성"에는 그래픽 컨트롤을 만드는 예제가 나와 있습니다.

Windows 컨트롤 서브클래싱

일반적인 Windows 프로그래밍에서는 새 *window 클래스*를 정의하고 Windows 에 등록하여 사용자 정의 컨트롤을 만듭니다. 객체 지향 프로그래밍의 *객체* 또는 *클래스*와 유사한 window 클래스에는 같은 종류의 컨트롤의 인스턴스와 공유하는 정보가 포함되며 개발자는 기존 클래스를 기반으로 새 window 클래스를 만들 수 있으며 이를 *서브클래싱 (subclassing)* 이라고 합니다. 그런 다음 표준 Windows 컨트롤과 같이 컨트롤을 동적 연결 라이브러리(DLL)에 두고 이에 대한 인터페이스를 제공합니다.

C++Builder를 사용하면 임의의 기존 window 클래스에 대해 컴포넌트 "랩퍼"를 생성할 수 있습니다. 따라서 C++Builder 애플리케이션에서 사용할 사용자 정의 컨트롤 라이브러리가 이미 있을 경우에는 컨트롤과 유사하게 작동하는 C++Builder 컴포넌트를 만들 수 있으며 기타 다른 컴포넌트의 경우와 마찬가지로 이로부터 새 컨트롤을 파생시킬 수 있습니다.

Windows 컨트롤 서브클래싱에 사용되는 기술에 대한 예제를 보려면 *TEdit*와 같은 표준 Windows 컨트롤을 나타내는 StdCtls 헤더 파일의 컴포넌트를 참조하십시오. CLX 예제의 경우에는 QStdCtls를 참조하십시오.

넌비주얼(nonvisual) 컴포넌트 생성

넌비주얼(nonvisual) 컴포넌트는 데이터베이스(*TDataSet* 또는 *TSQLConnection*) 및 시스템 클럭(*TTimer*)과 같은 요소의 인터페이스로 사용되며 다이얼로그 박스(*TCommonDialog*(VCL) 또는 *TDialog*(CLX) 및 그 자손)에 대한 위치 표시자 역할을 합니다. 개발자가 만드는 대부분의

컴포넌트는 비주얼(visual) 컨트롤입니다. 넌비주얼(nonvisual) 컴포넌트는 모든 컴포넌트에 대한 기본 추상 클래스인 *TComponent*로부터 직접 파생될 수 있습니다.

컴포넌트 생성 시 고려 사항

컴포넌트를 C++Builder 환경의 신뢰할 수 있는 부분으로 만들기 위해서는 디자인할 때 몇 가지 규칙을 따라야 합니다. 이 단원에서 다루는 주제는 다음과 같습니다.

- 종속성 제거
- 속성, 메소드 및 이벤트 설정
- 그래픽 캡슐화
- 컴포넌트 등록

종속성 제거

컴포넌트를 유용하게 만드는 한 가지 방법은 코드의 어떤 부분에서도 컴포넌트가 수행할 수 있는 작업에 대해 제한을 두지 않는 것입니다. 컴포넌트는 본질적으로 조합, 순서 및 컨텍스트를 달리 하여 애플리케이션에 통합됩니다. 개발자는 반드시 미리 정해진 조건 없이 어떠한 상황에서도 기능을 발휘할 수 있는 컴포넌트를 디자인해야 합니다.

종속성을 제거하는 훌륭한 예는 *TWinControl*의 *Handle* 속성입니다. Windows 애플리케이션을 작성해 본 경험이 있는 개발자라면 프로그램이 실행되도록 하는 데 있어 가장 어려우면서도 오류가 발생하기 쉬운 작업 중 하나는 *CreateWindow* API 함수를 호출하여 윈도우 컨트롤을 만들 때까지는 이 컨트롤에 액세스를 하지 않도록 주의하는 것임을 알 것입니다. C++Builder의 윈도우 컨트롤의 경우, 적절한 윈도우 핸들이 필요할 때 항상 사용할 수 있으므로 이런 걱정을 할 필요가 없습니다. 윈도우 컨트롤은 윈도우 핸들을 나타내는 속성을 사용하여 윈도우가 생성되었는지, 핸들이 유효하지 않은지, 컨트롤이 윈도우를 생성하고 핸들을 반환하는지를 검사할 수 있습니다. 따라서 애플리케이션의 코드가 *Handle* 속성에 액세스할 때마다 유효한 핸들을 가지도록 보장해 줍니다.

윈도우 생성 등의 백그라운드 작업을 제거하면 C++Builder 컴포넌트를 사용하여 실제로 원하는 작업에 집중할 수 있습니다. 윈도우 핸들을 API 함수에 전달하기 전에 핸들이 존재하는지 확인하거나 윈도우를 만들 필요가 없습니다. 애플리케이션 개발자는 오류가 없는지 계속 확인할 필요 없이 컴포넌트가 제대로 실행되리라고 안심할 수 있습니다.

종속성이 없는 컴포넌트를 작성하는 데 다소 시간이 걸리더라도 일반적으로 충분히 그럴만한 가치가 있는 일입니다. 즉 반복적이고 지루한 작업을 하지 않아도 될 뿐만 아니라 문서화와 지원에 필요한 작업도 줄여줍니다.

속성, 메소드 및 이벤트 설정

폼 디자이너에서 처리되는 시각적 이미지 외에 컴포넌트의 가장 확실한 어트리뷰트(attribute)는 해당 속성, 이벤트 및 메소드입니다. 이 설명서의 각 장에서 이러한 각 요소에 대해 자세히 다루고 있으며, 여기서는 이러한 요소들을 사용하는 이유에 대해 설명합니다.

속성

속성은 애플리케이션 개발자에게 변수 값을 설정하거나 읽는다는 착각을 주는 것과 동시에, 컴포넌트 작성자가 원본으로 사용하는 데이터 구조를 숨기고, 값이 액세스될 때 특수 처리 작업을 구현할 수 있게 합니다.

속성을 사용하면 다음과 같은 여러 가지 이점이 있습니다.

- 속성은 디자인 타임에서 사용할 수 있습니다. 애플리케이션 개발자가 코드를 작성할 필요 없이 속성의 초기 값을 설정하거나 변경할 수 있습니다.
- 속성은 애플리케이션 개발자가 할당한 값이나 형식을 검사할 수 있습니다. 디자인 타임에 입력을 검증하여 오류를 방지합니다.
- 컴포넌트는 요구 즉시 적절한 값을 생성할 수 있습니다. 프로그래머들이 가장 많이 실수하는 부분은 아마도 초기화되지 않은 변수를 참조하는 것입니다. 속성을 사용하여 데이터를 나타내면 값을 요구 즉시 항상 사용할 수 있도록 할 수 있습니다.
- 속성을 사용하면 간단하고 일관적인 인터페이스를 통해 데이터를 숨길 수 있습니다. 또한 애플리케이션 개발자들이 인식하지 못하는 상태에서 속성에 정보가 구성되는 방법을 변경할 수 있습니다.

47장, "속성 생성"에는 컴포넌트에 속성을 추가하는 방법이 설명되어 있습니다.

이벤트

이벤트란 런타임 시 입력 또는 다른 작업에 대한 응답으로 코드를 호출하는 특수 속성입니다. 애플리케이션 개발자는 이벤트를 사용하여 마우스 동작 및 키스트로크 등의 특정 런타임 작업에 대해 특정 코드 블록을 추가할 수 있습니다. 이벤트가 발생할 때 실행되는 코드를 *이벤트 핸들러*라고 합니다.

이벤트를 사용하면 애플리케이션 개발자는 새 컴포넌트를 정의하지 않아도 각기 다른 종류의 입력에 대한 응답을 지정할 수 있습니다.

48장, "이벤트 생성"에는 표준 이벤트를 구현하는 방법과 새 이벤트를 정의하는 방법이 설명되어 있습니다.

메소드

클래스 메소드는 클래스의 특정 인스턴스가 아니라 하나의 클래스에서 작동하는 함수입니다. 예를 들어, 모든 컴포넌트의 생성자 메소드는 클래스 메소드입니다. 컴포넌트 메소드는 컴포넌트 인스턴스 자체에서 작동하는 함수입니다. 애플리케이션 개발자는 메소드를 사용하여 컴포넌트가 특정 작업을 수행하도록 하거나 속성에 포함되지 않은 값을 반환하도록 합니다.

메소드를 호출하려면 코드를 실행해야 하므로 메소드는 런타임 시에만 호출할 수 있습니다. 메소드는 여러 가지 이유로 인해 유용합니다.

- 메소드는 데이터가 상주하는 동일한 객체 내에서 컴포넌트 기능을 캡슐화합니다.
- 메소드는 간단하고 일관적인 인터페이스 하에 복잡한 프로시저를 숨길 수 있습니다. 애플리케이션 개발자는 메소드가 작동하는 방법 또는 다른 컴포넌트에 있는 *AlignControls* 메소드와의 차이점을 모르더라도 컴포넌트의 *AlignControls* 메소드를 호출할 수 있습니다.
- 메소드를 사용하면 단일 호출로 여러 속성을 업데이트할 수 있습니다.

49장, "메소드 생성"에는 컴포넌트에 메소드를 추가하는 방법이 설명되어 있습니다.

그래픽 캡슐화

C++Builder는 다양한 그래픽 도구를 캔버스에 캡슐화하여 Windows 그래픽을 단순화합니다. 캔버스는 윈도우나 컨트롤의 그리기 표면을 나타내며 펜, 브러시 및 글꼴 등의 다른 클래스를 포함합니다. 캔버스는 Windows 장치 컨텍스트와 유사하나 개발자를 대신하여 모든 정리 작업을 관리한다는 점이 다릅니다.

그래픽 환경의 Windows 애플리케이션을 작성한 경험이 있으면 Windows의 GDI(Graphics Device Interface)에 필요한 요구 사항에 대해 잘 알 것입니다. 예를 들어, GDI는 사용할 수 있는 장치 컨텍스트의 수를 제한하며 그래픽 객체를 소멸시키지 전에 초기 상태로 복구하여야 합니다.

C++Builder를 사용하면 이러한 점에 대해 걱정할 필요가 없습니다. 폼이나 다른 컴포넌트에서 그림을 그리려면 컴포넌트의 *Canvas* 속성에 액세스하면 됩니다. 펜이나 브러시를 사용자 정의하려면 색상이나 스타일을 설정하면 됩니다. 설정이 끝나면 C++Builder가 리소스를 해제합니다. C++Builder는 애플리케이션이 동일한 리소스를 자주 사용하는 경우, 사용할 때마다 다시 작성하지 않아도 되도록 리소스를 캐싱합니다.

개발자는 모든 Windows GDI를 계속 액세스할 수 있으나 C++Builder 컴포넌트에 내장된 캔버스를 사용하면 코드가 보다 단순해지고 실행 속도도 빨라집니다. 그래픽 기능에 대한 자세한 내용은 50장, "컴포넌트에서 그래픽 사용"을 참조하십시오.

CLX 그래픽 캡슐화는 다르게 작동합니다. 페인터 대신 캔버스가 있습니다. 폼이나 다른 컴포넌트에서 그림을 그리려면 컴포넌트의 *Canvas* 속성에 액세스하십시오. *Canvas*는 속성이면서 또한 *TCanvas*라는 객체이기도 합니다. *TCanvas*는 *Handle* 속성을 통해 액세스할 수 있는 Qt 페인터의 랩퍼로 사용됩니다. 핸들을 사용하여 낮은 레벨의 Qt 그래픽 라이브러리 함수에 액세스할 수 있습니다.

펜이나 브러시를 사용자 정의하려면 색상이나 스타일을 설정하면 됩니다. 설정이 끝나면 C++Builder가 리소스를 해제합니다. CLX에서도 리소스를 캐싱합니다.

CLX 컴포넌트의 자손을 만드는 방법을 사용하여 CLX 컴포넌트에 내장된 캔버스를 사용할 수 있습니다. 컴포넌트에서 그래픽 이미지를 사용하는 방법은 컴포넌트가 파생된 객체의 캔버스에 따라 달라집니다.

컴포넌트 등록

C++Builder IDE에 컴포넌트를 설치하기 전에 반드시 등록을 해야 합니다. 등록이란 컴포넌트 팔레트의 어떤 위치에 컴포넌트를 둘 지 C++Builder에게 알려주는 작업을 의미합니다. 또한 C++Builder가 폼 파일에 컴포넌트를 저장하는 방법을 사용자 정의할 수 있습니다. 컴포넌트 등록에 대한 자세한 내용은 52장, "디자인 타임 시 컴포넌트 사용"을 참조하십시오.

새 컴포넌트 생성

다음 두 가지 방법으로 새 컴포넌트를 만들 수 있습니다.

- 컴포넌트 마법사로 컴포넌트 생성
- 수동으로 컴포넌트 생성

위에서 설명한 방법 중 하나를 사용하여 컴포넌트 팔레트에 설치할 준비가 된 최소한의 기능을 가진 컴포넌트를 만들 수 있습니다. 설치한 다음 폼에 새 컴포넌트를 추가하고 디자인 타임과 런타임에서 모두 테스트할 수 있습니다. 그런 다음 컴포넌트에 보다 많은 기능을 추가하고 컴포넌트 팔레트를 업데이트하여 계속 테스트할 수 있습니다.

새 컴포넌트를 생성할 때마다 수행해야 하는 몇 가지 기본적인 단계가 있습니다. 이러한 단계는 아래에 설명되어 있으며 이 문서의 기타 예제에서는 개발자가 이러한 단계 수행에 대해 알고 있다고 가정하고 설명합니다.

1 새 컴포넌트의 유닛을 만듭니다.

2 기존 컴포넌트 타입에서 컴포넌트를 파생시킵니다.

3 속성, 메소드 및 이벤트를 추가합니다.

4 C++Builder에 컴포넌트를 등록합니다.

5 컴포넌트와 그 속성, 메소드 및 이벤트에 대한 도움말 파일을 만듭니다.

참고 컴포넌트 사용자에게 컴포넌트 사용법을 알려 주는 도움말 파일을 만드는 것은 옵션입니다.

6 C++Builder IDE에 컴포넌트를 설치할 수 있도록 패키지(특수 동적 연결 라이브러리)를 만듭니다.

이를 완료하면 완성된 컴포넌트에 다음과 같은 파일이 포함됩니다.

- 패키지(.BPL) 또는 패키지 컬렉션(.PCE) 파일
- 패키지용 라이브러리(.LIB)
- 패키지용 Borland 임포트 라이브러리(.BPI) 파일
- 컴파일된 유닛 (.OBJ) 파일
- 팔레트 맵용 컴파일된 리소스(.RES) 파일
- 도움말(.HLP) 파일

또한 새 컴포넌트를 나타내기 위해 비트맵을 생성할 수 있습니다. 45-14페이지의 "컴포넌트용 비트맵 생성"을 참조하십시오.

5부의 나머지 장에서는 컴포넌트 생성에 대한 모든 측면을 설명하고 다른 종류의 컴포넌트를 작성하는 몇 가지 예제를 제공합니다.

컴포넌트 마법사로 컴포넌트 생성

컴포넌트 마법사는 컴포넌트 생성의 초기 단계를 단순화합니다. 컴포넌트 마법사를 사용하는 경우 다음을 지정해야 합니다.

- 컴포넌트를 파생시킬 대상 클래스
- 새 컴포넌트의 클래스 이름
- 새 컴포넌트를 나타내고자 하는 컴포넌트 팔레트 페이지
- 컴포넌트가 만들어지는 유닛의 이름
- 유닛이 검색되는 검색 경로
- 컴포넌트를 두고자 하는 패키지의 이름

다음과 같이 Component 마법사는 컴포넌트를 수동으로 만들 때와 동일한 작업을 수행합니다.

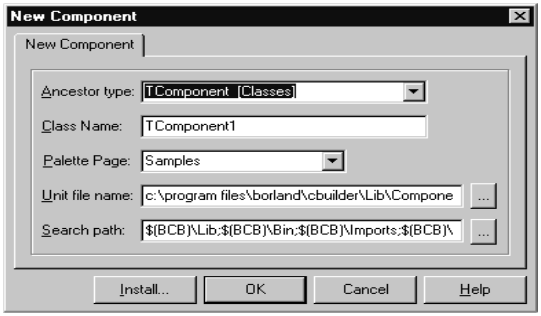
- 유닛 생성(.CPP 파일 및 연결 헤더)
- 컴포넌트 파생
- 새 생성자 선언
- 컴포넌트 등록

Component 마법사는 .CPP 파일과 연결 헤더 파일로 구성된 기존 유닛에 새 컴포넌트를 추가할 수 없습니다. 새 컴포넌트를 추가하려면 반드시 유닛에 수동으로 추가해야 합니다.

1 Component 마법사를 시작하려면 다음 두 가지 방법 중 하나를 선택합니다.

- Component|New Component를 선택합니다
- File|New|Other를 선택하고 Component를 더블 클릭합니다

그림 45.2 Component 마법사



다음과 같이 컴포넌트 마법사의 필드에 입력합니다.

2 Ancestor Type 필드에서 새 컴포넌트를 파생시킬 클래스를 지정합니다.

참고 드롭다운 리스트에서 대부분의 컴포넌트는 VCL 유닛 이름과 CLX 유닛 이름의 두 가지로 나열되어 있습니다. CLX 유닛 이름은 Q로 시작합니다. 예를 들어, Graphics 대신 QGraphics가 사용됩니다. 새 컴포넌트가 올바른 컴포넌트에서 파생되도록 합니다.

3 Class Name 필드에서 새 컴포넌트 클래스의 이름을 지정합니다.

4 Palette Page 필드에서 새 컴포넌트를 설치하고자 하는 컴포넌트 팔레트의 페이지를 지정합니다.

5 Unit file name 필드에서 컴포넌트 클래스를 선언하고자 하는 유닛의 이름을 지정합니다.

6 유닛이 검색 경로에 없는 경우 필요에 따라 Search Path 필드에서 검색 경로를 편집합니다.

7 컴포넌트를 새 패키지나 기존 패키지에 두려면 Component | Install 을 클릭한 다음 나타나는 다이얼로그 박스를 사용하여 패키지를 지정합니다.

경고 이름이 "custom"으로 시작하는 VCL 또는 CLX 클래스(예: *TCustomControl*)에서 컴포넌트를 파생시킨 경우 원래 컴포넌트에서 모든 추상 메소드를 오버라이드하기 전까지는 폼에 새 컴포넌트를 두려고 시도하지 마십시오. C++Builder는 추상 속성이나 메소드가 있는 클래스의 인스턴스 객체를 생성할 수 없습니다.

8 Component 마법사의 필드를 채운 다음 OK를 선택하십시오. 그러면 C++Builder가 .cpp 파일과 연결 헤더로 구성된 새 유닛을 생성합니다.

.cpp 파일은 코드 에디터에 나타납니다. 여기에는 컴포넌트를 등록하고 컴포넌트 라이브러리에 추가할 컴포넌트와 이를 표시할 컴포넌트 팔레트의 페이지를 C++Builder에 알려주는 *Register* 함수 및 컴포넌트용 생성자가 포함됩니다. 이 파일에는 생성된 헤더 파일을 지정하는 include 문도 포함됩니다. 예를 들면, 다음과 같습니다.

```
#include <vcl.h>
#pragma hdrstop
#include "NewComponent.h"
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not
have
// any pure virtual functions.
//

static inline void ValidCtrCheck(TNewComponent *)
{
        new TNewComponent(NULL);
}
//---------------------------------------------------------------------------
__fastcall TNewComponent::TNewComponent(TComponent* Owner)
   : TComponent(Owner)
{
}
//---------------------------------------------------------------------------
namespace Newcomponent
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] = {__classid(TNewComponent)};
```

```
    RegisterComponents("Samples", classes, 0); //In CLX use a different
page than Samples
  }
}
```

CLX CLX 애플리케이션은 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX에서 `<clx\qcontrols.hpp>`가 됩니다.

코드 에디터에서 헤더 파일을 열려면 커서를 헤더 파일 이름 위에 놓고 마우스 오른쪽 버튼을 클릭하여 컨텍스트 메뉴를 표시한 다음 Open File at Cursor를 선택하면 됩니다.

헤더 파일은 생성자 선언 및 새 클래스를 지원하는 데 필요한 **#include** 문을 비롯하여 새 클래스 선언을 포함합니다. 예를 들면, 다음과 같습니다.

```
#ifndef NewComponentH
#define NewComponentH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TNewComponent : public TComponent
{
private:
protected:
public:
  __fastcall TNewComponent(TComponent* Owner);
__published:
};
//---------------------------------------------------------------------------
#endif
```

계속 진행하기에 앞서 의미 있는 이름으로 .cpp 파일을 저장하십시오.

수동으로 컴포넌트 생성

새 컴포넌트를 만드는 가장 쉬운 방법은 컴포넌트 마법사를 사용하는 것입니다. 그러나 수동으로도 동일한 단계를 수행할 수 있습니다.

수동으로 컴포넌트를 만들려면 다음 단계를 따르십시오.

1. 유닛 파일 생성
2. 컴포넌트 파생
3. 새 생성자 선언
4. 컴포넌트 등록

유닛 파일 생성

C++Builder 유닛은 .OBJ 파일로 컴파일되는 .CPP 파일 및 .H 파일로 구성됩니다. C++Builder가 유닛을 사용하는 목적은 매우 다양합니다. 모든 폼에는 자체 유닛이 있으며 대부분의 컴포넌트 또는 컴포넌트의 논리적 그룹 또한 자체 유닛이 있습니다.

컴포넌트를 만들 때 컴포넌트의 새 유닛을 만들거나 기존 유닛에 새 컴포넌트를 추가합니다.

1 컴포넌트에 대해 유닛을 만들려면 다음 방법 중 하나를 선택하십시오.

- File|New|Unit을 선택합니다.
- File|New|Other를 선택하여 New Items 다이얼로그 박스를 표시하고 Unit을 선택한 다음 OK를 선택합니다.

  C++Builder는 .CPP 파일 및 헤더 파일을 생성하고 코드 에디터에 .CPP 파일을 표시합니다. 의미 있는 이름을 사용하여 파일을 저장합니다.

2 헤더 파일을 열려면 코드 에디터에서 헤더 파일 이름 위에 커서를 놓고 마우스 오른쪽 버튼을 클릭한 다음 팝업 메뉴에서 Open File at Cursor를 선택합니다.

3 기존 유닛을 열려면 File|Open을 선택한 다음 컴포넌트를 추가할 소스 코드 유닛을 선택합니다.

참고 기존 유닛에 컴포넌트를 추가하는 경우 유닛에 컴포넌트 코드만 포함되었는지 확인하십시오. 예를 들어, 폼을 포함하는 유닛에 컴포넌트 코드를 추가하면 컴포넌트 팔레트에서 오류가 발생할 수 있습니다.

4 컴포넌트에 대한 새 유닛 또는 기존 유닛을 만들면, 컴포넌트 클래스를 파생시킬 수 있습니다.

컴포넌트 파생

모든 컴포넌트는 *TComponent*나 이 컴포넌트의 특화된 자손(*TControl* 또는 *TGraphicControl*), 또는 기존 컴포넌트 클래스로부터 파생되는 클래스입니다. 45-2페이지의 "컴포넌트 생성 방법"에서는 어떤 클래스로부터 다른 종류의 컴포넌트가 파생되는지에 대해 설명합니다.

클래스 파생에 대해서는 46-1페이지의 "새 클래스 정의" 단원에서 보다 자세히 설명합니다.

컴포넌트 클래스를 파생시키려면 클래스 선언을 헤더 파일에 추가하십시오.

간단한 컴포넌트 클래스는 *TComponent*에서 직접 파생된 넌비주얼(nonvisual) 컴포넌트입니다.

간단한 컴포넌트 클래스를 만들려면 다음 클래스 선언을 개발자의 헤더 파일에 추가하십시오.

```
class PACKAGE TNewComponent : public TComponent
{
};
```

PACKAGE 매크로는 클래스를 임포트하고 엑스포트할 수 있는 명령문으로 확장됩니다. 또한 새 컴포넌트에 필요한 .HPP 파일을 지정하는 필수 include 문을 추가해야 합니다. 다음은 가장 일반적인 필수 include 문입니다.

```
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
```

지금까지는 새 컴포넌트가 *TComponent*와 다른 점이 없습니다. 즉, 새 컴포넌트를 생성할 프레임워크를 작성한 것입니다.

새 생성자 선언

각 새 컴포넌트는 반드시 컴포넌트가 파생된 클래스의 생성자를 오버라이드하는 생성자가 있어야 합니다. 새 컴포넌트에 대해 생성자를 작성하는 경우 *항상* 상속된 생성자를 호출해야 합니다.

클래스 선언 내에서 클래스의 public 섹션에서 가상 생성자를 선언합니다. public 섹션에 대한 자세한 내용은 46-4페이지의 "액세스 제어"를 참조하십시오. 예를 들면, 다음과 같습니다.

```
class PACKAGE TNewComponent : public TComponent
{
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
};
```

.CPP 파일에서 생성자를 구현합니다.

```
__fastcall TNewComponent::TNewComponent(TComponent* AOwner):
TComponent(AOwner)
{
}
```

컴포넌트가 생성될 때 실행될 코드를 생성자에 추가합니다.

컴포넌트 등록

등록이란 컴포넌트 라이브러리에 추가할 컴포넌트와 이를 컴포넌트 팔레트의 어떤 페이지에 둘 지를 C++Builder에게 알려주는 작업을 의미합니다. 등록 절차에 대한 자세한 내용은 52장, "디자인 타임 시 컴포넌트 사용"을 참조하십시오.

다음과 같은 방법으로 컴포넌트를 등록합니다.

1 *Register*라는 함수를 유닛의 .CPP 파일에 추가하고 네임스페이스 내에 놓습니다. 네임스페이스는 컴포넌트가 속한 파일의 이름에서 파일 확장자를 빼고 첫문자를 제외하고는 모두 소문자로 만든 이름입니다.

   예를 들어, 다음 코드는 *Newcomp* 네임스페이스에 있으며 여기서 *Newcomp*는 .CPP 파일의 이름입니다.

   ```
   namespace Newcomp
   {
     void __fastcall PACKAGE Register()
     {
     }
   }
   ```

2 *Register* 함수 내에서 등록하고 있는 컴포넌트의 배열을 포함한 *TComponentClass* 타입의 개방형 배열을 선언합니다. 구문은 다음과 같이 나타납니다.

   ```
   TComponentClass classes[1] = {__classid(TNewComponent)};
   ```

   이런 경우, 클래스의 배열은 단 하나의 컴포넌트만을 포함하지만 등록하려는 모든 컴포넌트를 배열에 추가할 수 있습니다.

3 *Register* 함수에서 등록하고자 하는 각 컴포넌트에 대해 *RegisterComponents*를 호출합니다.

*RegisterComponents*는 세 개의 매개변수를 사용하는 함수입니다. 해당 매개변수는 컴포넌트 팔레트 페이지의 이름, 컴포넌트 클래스의 배열 및 컴포넌트 클래스의 크기에서 1을 뺀 값입니다. 기존 등록에 컴포넌트를 추가하는 경우 기존 명령문에 있는 집합에 새 컴포넌트를 추가하거나 *RegisterComponents*를 호출하는 새 명령문을 추가할 수 있습니다.

모든 컴포넌트가 컴포넌트 팔레트의 동일한 페이지로 가는 경우에는 *RegisterComponents*를 한 번만 호출하여도 여러 컴포넌트를 등록할 수 있습니다.

*TNewComponent*라는 컴포넌트를 등록하여 컴포넌트 팔레트의 Samples 페이지에 놓으려면 *TNewComponent*를 선언하는 유닛의 .CPP 파일에 다음과 같은 *Register* 함수를 추가하십시오.

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TNewComponent)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

이 *Register* 호출은 *TNewComponent*를 컴포넌트 팔레트의 Samples 페이지에 둡니다.

일단 컴포넌트를 등록하면 컴포넌트를 테스트할 수 있으며 마침내 컴포넌트 팔레트에 컴포넌트를 설치할 수 있습니다. 자세한 내용은 45-18 페이지의 "컴포넌트 팔레트에 컴포넌트 설치" 단원을 참조하십시오.

컴포넌트용 비트맵 생성

새 컴포넌트를 만드는 경우 사용자 정의 컴포넌트용 자체 비트맵을 정의할 수 있습니다.

1 Tools | Image Editor를 선택합니다.

2 Image Editor 다이얼로그 박스에서 File | New | Component Resource File (.dcr)을 선택합니다.

3 untitled1.dcr 다이얼로그 박스에서 마우스 오른쪽 버튼으로 Contents를 클릭합니다. New | Bitmap을 선택합니다.

4 Bitmaps Properties 다이얼로그 박스에서 Width 및 Height를 모두 24 픽셀로 변경합니다. VGA (16 colors)가 선택되었는지 확인합니다. OK를 클릭합니다.

5 그러면 Contents 아래에 Bitmap과 Bitmap1이 표시됩니다. Bitmap1을 선택하고 마우스 오른쪽 버튼으로 클릭한 다음, Rename을 선택합니다. 비트맵에 T를 포함하여 새 컴포넌트용 클래스 이름과 동일한 이름을 지정합니다. 이 때 모두 대문자를 사용합니다. 예를 들어, 새 클래스 이름이 *TMyNewButton*이면 비트맵 이름은 TMYNEWBUTTON이 됩니다.

참고 New Component 다이얼로그 박스의 클래스 이름과 상관 없이 반드시 대문자를 사용해야 합니다.



6. TMYNEWBUTTON을 더블 클릭하여 비어 있는 비트맵이 있는 다이얼로그 박스를 표시합니다.
7. Image Editor 밑에 있는 색상 팔레트를 사용하여 아이콘을 디자인합니다.
8. File|Save As를 선택하고 컴포넌트 클래스를 선언하고자 하는 대상 유닛과 기본 이름이 동일한 리소스 파일의 이름(.dcr 또는 .res)을 지정합니다. 예를 들어, 리소스 파일의 이름을 MyNewButton.dcr로 지정합니다.
9. Component|New Component를 선택합니다. 45-9페이지에서 컴포넌트 마법사로 새 컴포넌트를 만드는 방법에 대한 지침을 따릅니다. 컴포넌트 소스인 MyNewButton.cpp가 MyNewButton.dcr과 동일한 디렉토리에 있는지 확인합니다.

   TMyNewButton 클래스의 경우, 컴포넌트 마법사는 컴포넌트 소스 또는 유닛의 이름을 MyNewButton.cpp로 지정하고 디폴트로 LIB 디렉토리에 배치합니다. Browse 버튼을 클릭하여 생성된 컴포넌트 유닛의 새 위치를 찾습니다.

참고 비트맵에 .dcr이 아니라 .res 파일을 사용하는 경우 컴포넌트 소스에 참조를 추가하여 리소스를 연결하십시오. 예를 들어, .res 파일의 이름이 MyNewButton.res이면 .cpp 및 .res가 동일한 디렉토리에 있는지 확인한 다음, MyNewButton.cpp에 다음을 추가하십시오.

```
#pragma resource "*.res"
```

10. Component|Install Component를 선택하여 새 패키지 또는 기존 패키지에 컴포넌트를 설치합니다. OK를 클릭합니다.

    새 패키지가 생성되고 설치됩니다. 컴포넌트 마법사에서 지정한 컴포넌트 팔레트 페이지에 새 컴포넌트를 나타내는 비트맵이 표시됩니다.

설치되지 않은 컴포넌트 테스트

컴포넌트 팔레트에 컴포넌트를 설치하기 전에 컴포넌트의 런타임 동작을 테스트할 수 있습니다. 이러한 테스트는 새로 생성된 컴포넌트 디버깅에 특히 유용하지만 컴포넌트가 컴포넌트 팔레트에 있는지에 관계 없이 모든 컴포넌트에 이 기술을 사용할 수 있습니다. 이미 설치된 컴포넌트를 테스트하는 방법에 대한 자세한 내용은 45-18페이지의 "설치된 컴포넌트 테스트"를 참조하십시오.

컴포넌트를 설치하지 않고 테스트하는 경우 클래스를 인스턴스화할 때만 볼 수 있는 컴파일 타임 오류를 생성할 수 있다는 장점이 있습니다. 예를 들어, 추상 클래스의 인스턴스를 생성하려고 하면 오버로드되어야 하는 순수 가상을 지시하는 오류가 발생합니다.

팔레트에서 컴포넌트를 선택하여 폼에 놓을 때 C++Builder가 수행하는 작업을 에뮬레이트하여 설치되지 않은 컴포넌트를 테스트합니다.

설치되지 않은 컴포넌트를 테스트하려면 다음과 같이 하십시오.

1. 새 애플리케이션을 생성하거나 기존 애플리케이션을 엽니다.
2. Project|Add to Project를 선택하여 개발자의 프로젝트에 컴포넌트 유닛을 추가합니다.
3. 폼 유닛의 헤더 파일에 컴포넌트 유닛의 .H 파일을 포함시킵니다.
4. 컴포넌트를 나타내는 폼에 데이터 멤버를 추가합니다.

   이러한 점이 개발자가 컴포넌트를 추가하는 방식과 C++Builder가 추가하는 방식의 가장 큰 차이점 중 하나입니다. 개발자는 폼의 클래스 선언의 아래쪽에 있는 public 부분에 데이터 멤버를 추가하며 C++Builder는 자신이 관리하는 클래스 선언의 위에 있는 published 부분에 추가합니다.

   이 때 절대로 폼의 클래스 선언 중 C++Builder가 관리하는 부분에 데이터 멤버를 추가해서는 안됩니다. 클래스 선언 중 해당 부분에 있는 항목은 폼 파일에 저장되는 항목에 해당됩니다. 폼에 존재하지 않는 컴포넌트의 이름을 추가하면 폼 파일이 무효화될 수 있습니다.
5. 폼의 생성자에서 컴포넌트를 만듭니다.

   컴포넌트의 생성자를 호출하는 경우 반드시 컴포넌트의 소유자(적절한 때에 컴포넌트를 소멸시킬 책임이 있는 컴포넌트)를 지정하는 매개변수를 전달해야 합니다. 거의 대부분 소유자로 **this**를 전달합니다. 메소드에서 **this**는 메소드를 포함한 클래스에 대한 참조입니다. 이런 경우, 폼의 *OnCreate* 핸들러에서 **this**는 폼을 참조합니다.
6. *Parent* 속성을 할당합니다.

   Parent 속성을 설정하는 것은 컨트롤을 생성한 다음 항상 제일 먼저 해야 할 일입니다. *Parent*는 시각적으로 컨트롤을 포함하는 컴포넌트이며 대부분의 경우 폼이지만 그룹 박스나 패널일 경우도 있습니다. 일반적으로 *Parent*를 **this**, 즉 폼으로 설정하게 됩니다. 컨트롤의 기타 속성을 설정하기 전에 항상 *Parent*를 설정하십시오.
7. 다른 모든 컴포넌트 속성을 원하는대로 설정합니다.

*NewCtrl*이라는 유닛에 있는 *TNewControl* 클래스의 새 컴포넌트를 테스트한다고 가정합니다. 새 프로젝트를 만들고 위의 단계를 따라 폼 유닛의 헤더 파일을 다음과 같이 만듭니다.

```
//---------------------------------------------------------------------------
#ifndef TestFormH
#define TestFormH
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
#include "NewCtrl.h"
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:          // IDE-managed Components
private:              // User declarations
public:               // User declarations
    TNewControl* NewControl1;
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

NEWCTRL.H 파일을 포함하는 **#include** 문은 컴포넌트가 현재 프로젝트의 디렉토리 또는 프로젝트의 include 경로에 있는 디렉토리에 상주한다고 가정합니다.

다음은 폼 유닛의 .CPP 파일입니다.

```
#include <vcl\vcl.h>
#pragma hdrstop
#include "TestForm.h"
#include "NewCtrl.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
static inline TNewControl *ValidCtrCheck()
{
    return new TNewControl(NULL);
}
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    NewControl1 = new TNewControl(this);
    NewControl1->Parent = this;
    NewControl1->Left = 12;
}
//---------------------------------------------------------------------------
```

```
namespace Newctrl
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TNewControl)};
        RegisterComponents("Samples", classes, 0);
    }
}
```

설치된 컴포넌트 테스트

컴포넌트 팔레트에 컴포넌트를 설치한 후에 컴포넌트의 디자인 타임 동작을 테스트할 수 있습니다. 이런 테스트는 새로 생성된 컴포넌트의 디버깅에 특히 유용하며 컴포넌트가 컴포넌트 팔레트에 있는지에 관계 없이 모든 컴포넌트에 적용할 수 있습니다. 아직 설치하지 않은 컴포넌트 테스트에 대한 내용은 45-16페이지의 "설치되지 않은 컴포넌트 테스트"를 참조하십시오.

설치 후 컴포넌트를 테스트하면 폼에 가져다 놓을 때 디자인 타임 예외만을 생성하는 컴포넌트를 디버깅할 수 있습니다.

C++Builder의 두 번째 실행 인스턴스를 사용하여 설치된 컴포넌트를 테스트합니다.

1. C++Builder IDE 메뉴에서 Project|Options를 선택하고 Directories/Conditional 페이지에서 Debug Source Path를 컴포넌트 소스 파일로 설정합니다.
2. 그런 다음 Tools|Debugger 옵션을 선택합니다. Language Exceptions 페이지에서 추적할 예외를 활성화합니다.
3. 컴포넌트 소스 파일을 열고 브레이크포인트를 설정합니다.
4. Run|Parameters를 선택하고 Host Application 필드를 C++Builder 실행 파일의 이름과 위치로 설정합니다.
5. Run Parameters 다이얼로그 박스에서 Load 버튼을 클릭하여 C++Builder의 두 번째 인스턴스를 시작합니다.
6. 그리고 나서, 테스트할 컴포넌트를 폼에 가져다 놓습니다. 이 작업은 소스의 브레이크포인트에서 중단되어야 합니다.

컴포넌트 팔레트에 컴포넌트 설치

컴포넌트 팔레트에 컴포넌트를 추가하는 과정은 다음과 같은 두 부분으로 이루어집니다.

- 소스 파일을 사용할 수 있게 만들기
- 컴포넌트 추가

소스 파일을 사용할 수 있게 만들기

컴포넌트가 사용하는 모든 소스 파일은 같은 디렉토리에 있어야 합니다. 여기에는 소스 코드 파일(.CPP 및 .PAS)과 바이너리 파일(.DFM, .RES, .RC, .DCR)이 포함됩니다. 헤더 파일(.H 및 .HPP)은 Include 디렉토리 또는 IDE나 프로젝트용 검색 경로 상의 위치에 있어야 합니다.

컴포넌트를 추가하는 과정을 통해 많은 파일이 생성됩니다. 생성된 파일은 자동으로 IDE 환경 옵션(메뉴 명령 Tools|Environment Option 사용, Library 탭 페이지로 이동)에서 지정한 디렉토리에 저장됩니다. .LIB 파일은 BPI/LIB 출력 디렉토리에 저장됩니다. 컴포넌트를 추가할 때 기존 패키지에 설치하지 않고 새 패키지를 만들 경우 .BPL 파일은 BPL 출력 디렉토리에 저장되며 .BPI 파일들은 BPI/LIB 출력 디렉토리에 저장됩니다.

컴포넌트 추가

컴포넌트를 컴포넌트 라이브러리에 추가하려면 다음과 같이 하십시오.

1. Component|Install Component를 선택합니다.

   Install Component 다이얼로그 박스가 나타납니다.
2. 해당 페이지를 선택하여 새 컴포넌트를 기존 패키지에 설치할 것인지 또는 새 패키지를 만들 것인지를 선택합니다.
3. 새 컴포넌트를 포함하는 .CPP 파일의 이름을 입력하거나 Browse를 선택하여 유닛을 검색합니다.
4. 새 컴포넌트용 .CPP 파일이 디폴트 위치에 없으면 검색 경로를 조정합니다.
5. 컴포넌트를 설치할 패키지의 이름을 입력하거나 Browse를 선택하여 패키지를 검색합니다.
6. 컴포넌트를 새 패키지에 설치하는 경우, 패키지 설명을 입력할 수 있는 옵션이 있습니다.
7. OK를 선택하여 Install Component 다이얼로그 박스를 닫습니다. 패키지가 컴파일 또는 재생성되고 컴포넌트 팔레트에 컴포넌트가 설치됩니다.

참고 새로 설치된 컴포넌트는 처음에 컴포넌트 작성자가 지정한 컴포넌트 팔레트 페이지에 나타납니다. 그러나 Component|Configure Palette 다이얼로그 박스를 사용하면 팔레트에 컴포넌트를 설치한 후 다른 페이지로 이동시킬 수 있습니다.

장

46

컴포넌트 작성자를 위한 객체 지향 프로그래밍

C++Builder를 사용하여 애플리케이션을 작성해 본 적이 있다면 클래스에 데이터와 코드 두가지가 모두 포함되어 있으며 디자인 타임 및 런타임 시에 클래스를 처리할 수 있다는 것을 알고 있을 것입니다. 이 사실을 알고 있다는 것은 이미 컴포넌트 사용자임을 나타냅니다.

컴포넌트 작성자는 애플리케이션 개발자와는 전혀 다른 방법으로 클래스를 처리합니다. 또한 컴포넌트를 사용할 개발자에게 컴포넌트의 내부 작업을 알리지 않으려고 시도합니다. 컴포넌트의 적절한 조상을 선택하고 개발자가 필요로 하는 속성과 메소드만을 노출하는 인터페이스를 디자인하고 이 장의 기타 지침에 따라서 재사용 가능한 다용도의 컴포넌트를 만들 수 있습니다.

컴포넌트를 만들기 전에 객체 지향 프로그래밍(OOP)과 관련된 다음 주제에 익숙해져야 합니다.

- 새 클래스 정의
- 조상, 자손 및 클래스 계층 구조
- 액세스 제어
- 메소드 디스패칭
- 추상 클래스 멤버
- 클래스 및 포인터

새 클래스 정의

컴포넌트 작성자와 애플리케이션 개발자 간의 차이는 컴포넌트 작성자가 새 클래스를 만드는 반면에 애플리케이션 개발자는 클래스의 인스턴스를 처리한다는 점입니다.

클래스는 본질적으로 타입입니다. 클래스라는 용어를 사용하지 않는 경우에도 프로그래머는 타입과 인스턴스를 항상 사용하고 있습니다. *int*와 같은 타입의 변수를 생성하는 경우가 그 예입니다. 클래스는 일반적으로 간단한 데이터 타입보다 복잡하지만 동일한 방법으로 작동됩니다. 동일한 타입의 인스턴스에 다른 값을 할당하여 다른 작업을 수행할 수 있습니다.

예를 들어, '확인'이라는 레이블과 '취소'라는 레이블이 붙은 두 개의 버튼이 있는 폼을 생성하는 것은 매우 일반적인 작업입니다. 이 두 버튼은 모두 *TButton* 클래스의 인스턴스이지만 *Caption* 속성에 다른 값을 할당하고 *OnClick* 이벤트에 다른 핸들러를 할당하여 다르게 작동되는 두 개의 인스턴스를 만들 수 있습니다.

새 클래스 파생

새 클래스를 파생시키는 이유는 다음 두 가지입니다.

- 반복을 피하기 위한 클래스 기본값 변경
- 클래스에 새 기능 추가

두 가지 경우 모두 목적은 재사용 가능한 객체를 만드는 것입니다. 재사용할 수 있는 컴포넌트를 설계하려는 경우 나중에 사용하기 위해 컴포넌트를 저장할 수 있습니다. 클래스에 사용 가능한 기본값을 주고 사용자 정의할 수 있도록 허용하십시오.

반복을 피하기 위한 클래스 기본값 변경

대부분의 프로그래머는 반복을 피하려고 합니다. 따라서 동일한 코드 라인을 다시 작성하고 있는 경우에는 코드를 함수에 두거나 많은 프로그램에서 사용할 수 있는 루틴 라이브러리를 구축합니다. 컴포넌트에 대해서도 같은 이론이 적용됩니다. 동일한 속성을 변경하거나 동일한 메소드 호출을 작성하는 경우에는 이러한 작업을 디폴트로 수행하는 새 컴포넌트를 만들 수 있습니다.

예를 들어, 애플리케이션을 만들 때마다 다이얼로그 박스를 추가하여 특정 작업을 수행합니다. 그럴 때마다 다이얼로그 박스를 다시 작성하는 것은 어렵지 않지만 반드시 그럴 필요는 없습니다. 다이얼로그 박스를 한 번 디자인하여 속성을 설정하고 연결된 랩퍼 컴포넌트를 컴포넌트 팔레트에 설치할 수 있습니다. 다이얼로그 박스를 재사용할 수 있는 컴포넌트로 만들면 반복적인 작업을 피할 수 있을 뿐만 아니라 표준화가 가능하고 다이얼로그 박스가 다시 작성될 때마다 오류 가능성을 줄일 수 있습니다.

53장, "기존 컴포넌트 수정"에는 컴포넌트의 디폴트 속성을 변경하는 예제가 나와 있습니다.

참고 기존 컴포넌트의 published 속성만 수정하거나 컴포넌트 또는 컴포넌트 그룹의 특정 이벤트 핸들러를 저장하려는 경우 *컴포넌트 템플릿*을 만들면 필요한 작업을 더 쉽게 수행할 수 있습니다.

클래스에 새 기능 추가

새 컴포넌트를 생성하는 일반적인 이유는 기존 컴포넌트에 없는 기능을 추가하기 위해서입니다. 그러기 위해서는 *TComponent* 또는 *TControl* 등의 기본 추상 클래스나 기존 클래스에서 새 컴포넌트를 파생시켜야 합니다.

원하는 기능의 부분 집합에 가장 가까운 내용을 포함한 클래스로부터 새 컴포넌트를 파생시키십시오 . 클래스에 기능을 추가할 수는 있지만 제거할 수는 없으므로 기존 컴포넌트 클래스에 원하지 *않는* 기능이 있으면 컴포넌트의 조상으로부터 파생시켜야 합니다.

예를 들어, 리스트 박스에 기능을 추가하고자 하는 경우, *TListBox*에서 컴포넌트를 파생시킬 수 있습니다. 그러나 표준 리스트 박스에 새 기능을 추가하고 몇몇 기능을 제외하려면 *TListBox*의

조상인 *TCustomListBox*에서 컴포넌트를 파생시켜야 합니다. 그런 다음 원하는 리스트 박스 기능만 다시 작성하거나 보이게 하고 새 기능을 추가하면 됩니다.

55장, "그리드 사용자 정의"에는 추상 컴포넌트 클래스를 사용자 정의하는 예제가 나와 있습니다.

새 컴포넌트 클래스 선언

C++Builder는 표준 컴포넌트 외에, 새 컴포넌트를 파생시키기 위한 기본으로서 여러 개의 기본 추상 클래스를 제공합니다. 45-3페이지의 표 45.1은 새로운 컴포넌트를 만들 때부터 시작할 수 있는 클래스를 보여 줍니다.

새 컴포넌트 클래스를 선언하려면, 컴포넌트의 헤더 파일에 클래스 선언을 추가합니다.

다음은 간단한 그래픽 컴포넌트를 선언한 것입니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent* Owner);
};
```

클래스를 임포트하고 엑스포트할 수 있는 PACKAGE 매크로(Sysmac.h에서 정의됨)를 포함시키는 것을 잊지 마십시오.

완성된 컴포넌트 선언은 마지막 중괄호 앞에 속성, 데이터 멤버 및 메소드 선언을 포함하지만 비어 있는 선언도 사용할 수 있으며 여기서 컴포넌트 기능을 추가할 수 있습니다.

조상, 자손 및 클래스 계층 구조

모든 컨트롤에 폼에서 컨트롤의 위치를 결정하는 *Top* 및 *Left* 속성이 있는 것을 당연하게 생각합니다. 즉, 모든 컨트롤이 공통된 조상인 *TControl*로부터 이러한 속성을 상속했다는 것에 관심을 갖지 않습니다. 그러나 개발자가 컴포넌트를 생성할 때에는 반드시 이로부터 어떤 클래스가 파생되었는지를 알아야만 적절한 기능을 상속할 수 있습니다. 또한 컨트롤 상속에 대한 것도 모두 알고 있어야만 컨트롤을 다시 생성하지 않고 상속된 기능을 사용할 수 있습니다.

컴포넌트를 파생시킨 클래스를 *직계 조상*이라고 합니다. 각 컴포넌트는 직계 조상 및 직계 조상의 직계 조상 등으로부터 상속됩니다. 컴포넌트가 상속한 모든 대상 클래스를 *조상*이라고 하며 컴포넌트는 그 조상의 *자손*이 됩니다.

애플리케이션 내의 모든 조상, 자손 관계가 함께 클래스 계층 구조를 구성합니다. 클래스가 조상으로부터 모든 것을 상속한 다음 새 속성과 메소드를 추가하거나 기존 내용을 다시 정의하므로 계층 구조 내의 각 생성은 그 조상 이상의 것을 포함하게 됩니다.

직계 조상을 지정하지 않는 경우 C++Builder는 디폴트 조상인 *TObject*로부터 컴포넌트를 파생시킵니다. *TObject*는 객체 계층 구조 내의 모든 클래스에 대한 궁극적인 조상입니다.

어떤 객체로부터 파생시킬지 선택하는 일반적인 규칙은 간단합니다. 새 객체에 포함시키고자 하는 기능이 가능한 한 많으면서 원하지 않는 기능이 없는 객체를 선택합니다. 개발자는 항상 자신의 객체에 기능을 추가할 수 있으나 기능을 삭제할 수는 없습니다.

액세스 제어

속성, 메소드 및 데이터 멤버에 관한 다섯 가지 레벨의 *액세스 제어*, 일명 *가시성*이 있습니다. 가시성은 어떤 코드가 클래스의 어떤 부분에 액세스할 수 있는지를 결정합니다. 가시성을 지정하면 컴포넌트에 대한 *인터페이스*를 정의할 수 있습니다.

표 46.1에는 액세스가 가장 제한적인 것부터 가장 허용적인 것까지 가시성의 레벨이 나와 있습니다.

표 46.1 객체 내의 가시성 레벨

가시성	의미	용도
private	객체가 정의된 클래스에서만 액세스 가능	구현 세부 사항 숨기기
protected	객체가 정의되었으며 자손이 있는 클래스에서만 액세스 가능	컴포넌트 작성자의 인터페이스 정의
public	모든 코드에서 액세스 가능	런타임 인터페이스 정의
__automated	모든 코드에서 액세스 가능하며 Automation 타입 정보가 생성됨	OLE Automation 전용
__published	모든 코드와 Object Inspector에서 액세스 가능하며 폼 파일로 저장됨	디자인 타임 인터페이스 정의

구현 세부 사항 숨기기

클래스의 일부를 **private**으로 선언하면 함수가 클래스의 friend가 아닌 한 클래스 외부에 있는 코드에서 해당 부분을 볼 수 없습니다. 클래스의 private 부분은 대부분 클래스 사용자로부터 구현 세부 사항을 숨기는 데 유용합니다. 클래스 사용자는 private 부분에 액세스할 수 없으므로 개발자가 사용자 코드에 영향을 미치지 않고 클래스의 내부 구현을 변경할 수 있습니다.

데이터 멤버, 메소드 또는 속성에 대한 액세스 제어를 지정하지 않으면 해당 부분이 **private**이 됩니다.

다음은 데이터 멤버를 **private**으로 선언하여 사용자가 정보에 액세스하지 못하도록 하는 방법을 설명하는 두 부분을 보여주는 예제입니다.

첫 번째 부분은 헤더 파일과 폼의 *OnCreate* 이벤트 핸들러의 private 데이터 멤버에 값을 할당하는 .CPP 파일로 구성된 폼 유닛입니다. 이벤트 핸들러가 *TSecretForm* 클래스 내에서 선언되므로 유닛이 오류 없이 컴파일됩니다.

```
#ifndef HideInfoH
#define HideInfoH
//---------------------------------------------------------------------------
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSecretForm : public TForm
{
__published:    // IDE-managed Components
    void __fastcall FormCreate(TObject *Sender);
private:
    int FSecretCode;                                    // declare a
private data member
public:         // User declarations
  __fastcall TSecretForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TSecretForm *SecretForm;
//---------------------------------------------------------------------------
#endif
```

다음은 이와 연관된 .CPP 파일입니다.

```
#include <vcl.h>
#pragma hdrstop
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.dfm"
TSecretForm *SecretForm;
//---------------------------------------------------------------------------
__fastcall TSecretForm::TSecretForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TSecretForm::FormCreate(TObject *Sender)
{
  FSecretCode = 42;                             // this compiles
correctly
}
```

이 예제의 두 번째 부분은 *SecretForm* 폼의 *FSecretCode* 데이터 멤버에 값을 할당하려고 시도하는 또 다른 폼 유닛입니다. 다음은 유닛에 대한 헤더 파일입니다.

```
#ifndef TestHideH
#define TestHideH
//---------------------------------------------------------------------------
#include <vcl\SysUtils.hpp>
#include <vcl\Controls.hpp>
```

```
#include <vcl\Classes.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TTestForm : public TForm
{
__published:    // IDE-managed Components
    void __fastcall FormCreate(TObject *Sender);
public:         // User declarations
    __fastcall TTestForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TTestForm *TestForm;
//---------------------------------------------------------------------------
#endif
```

다음은 이와 연관된 .CPP 파일입니다. *OnCreate* 이벤트 핸들러가 *SecretForm* 폼의 private 데이터 멤버에 값을 할당하려고 시도하였으므로 `'TSecretForm::FSecretCode' is not accessible`이라는 오류 메시지가 표시되면서 컴파일이 실패합니다.

```
#include <vcl.h>
#pragma hdrstop
#include "testHide.h"
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.dfm"
TTestForm *TestForm;
//---------------------------------------------------------------------------
__fastcall TTestForm::TTestForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TTestForm::FormCreate(TObject *Sender)
{
  SecretForm->FSecretCode = 13;                //compiler stops here
with error message
}
```

HideInfo 유닛을 사용하는 프로그램은 *TSecretForm* 타입의 클래스를 사용할 수 있지만 이러한 클래스에 있는 *FSecretCode* 데이터 멤버에 액세스할 수 없습니다.

CLX CLX 애플리케이션은 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX에서 `<clx\qcontrols.hpp>`가 됩니다.

컴포넌트 작성자의 인터페이스 정의

클래스의 일부를 **protected**로 선언하면 이 부분이 해당 클래스와 그 자손에게만 보입니다.

개발자는 **protected** 선언을 사용하여 *컴포넌트 작성자*의 클래스에 대한 *인터페이스*를 정의할 수 있습니다. 애플리케이션 유닛에서는 protected 부분을 액세스할 수 없지만 파생된 클래스에서는 액세스할 수 있습니다. 즉, 컴포넌트 작성자는 애플리케이션 개발자에게 세부 사항을 공개하지 않고 클래스의 작업 방식을 변경할 수 있습니다.

참고 이벤트 핸들러에서 protected 메소드에 액세스하려고 시도하는 실수가 자주 발생합니다. 이벤트 핸들러가 일반적으로 폼의 메소드이지만 이벤트를 받을 수 있는 컴포넌트는 아닙니다. 결과적으로 컴포넌트가 폼과 동일한 유닛에서 선언되지 않는 한 컴포넌트의 protected 메소드에 액세스할 수 없습니다.

런타임 인터페이스 정의

클래스의 일부를 **public** 으로 선언하면 해당 클래스 전체를 액세스할 수 있는 모든 코드에서 해당 부분을 볼 수 있습니다.

public 부분은 런타임 시에 모든 코드에서 볼 수 있으므로 클래스의 public 부분은 *런타임 인터페이스*를 정의합니다. 이러한 런타임 인터페이스는 런타임 입력에 의존하거나 읽기 전용인 속성처럼 디자인 타임에서 의미가 없거나 적절하지 않은 항목에 대해 유용합니다. 애플리케이션 개발자가 메서드를 호출할 수 있게 하려면 public으로 선언해야 합니다.

다음은 컴포넌트의 런타임 인터페이스 부분으로 선언된 두 개의 읽기 전용 속성을 보여 주는 예제입니다.

```
class PACKAGE TSampleComponent : public TComponent
{
private:
    int FTempCelsius;                                    // implementation
details are private
    int GetTempFahrenheit();
public:
    f
    __property int TempCelsius = {read=FTempCelsius};          //
properties are public
    __property int TempFahrenheit = {read=GetTempFahrenheit};
};
```

다음은 .CPP 파일 내의 *GetTempFahrenheit* 메소드입니다.

```
int TSampleComponent::GetTempFahrenheit()
{
  return FTempCelsius * (9 / 5) + 32;
}
```

디자인 타임 인터페이스 정의

클래스의 부분을 **published**로 선언하면 해당 부분이 public이 되고 런타임 타입 정보가 생성됩니다. 특히 런타임 타입 정보를 통해 Object Inspector가 속성과 이벤트에 액세스할 수 있습니다.

클래스의 published 부분은 Object Inspector에 표시되므로 이 부분이 클래스의 *디자인 타임 인터페이스*를 정의합니다. 디자인 타임 인터페이스에는 애플리케이션 개발자가 디자인 타임에 사용자 정의하고자 하는 클래스의 모든 측면이 포함되어야 하지만 런타임 환경에 관한 특정 정보에 의존하는 속성은 포함되서는 안됩니다.

읽기 전용 속성은 애플리케이션 개발자가 직접적으로 값을 할당할 수 없으므로 디자인 타임 인터페이스의 부분이 될 수 없습니다. 따라서 읽기 전용 속성은 published 보다는 public 이어야 합니다.

다음은 *Temperature*라는 published 속성의 예제입니다. 이 속성은 published 속성이므로 디자인 타임에 Object Inspector에 표시됩니다.

```
class PACKAGE TSampleComponent : public TComponent
{
private:
    int FTemperature;
  f
__published:
    __property int Temperature = {read=FTemperature, write=FTemperature};
};
```

메소드 디스패칭

*디스패치*는 애플리케이션에 클래스 메소드 호출이 발생할 때 어떤 클래스 메소드가 호출되어야 하는지 결정하는 방법을 설명하는 데 사용되는 용어입니다. 클래스 메소드를 호출하는 코드를 작성할 때 마치 기타 다른 함수 호출처럼 보입니다. 그러나 클래스에는 메소드를 디스패치하는 두 가지 다른 방법이 있습니다.

두 가지 타입의 메소드 디스패치는 다음과 같습니다.

- 가상 메소드가 아닌 일반적인 메소드
- 가상 메소드

일반적인 메소드

개발자가 특별히 클래스 메소드를 가상으로 지정하거나 기본 클래스에서 가상 메소드를 오버라이드하지 않는 한 클래스 메소드는 일반적인(또는 비가상) 메소드가 됩니다. 컴파일러는 컴파일 시에 일반적인 클래스 멤버의 정확한 주소를 판별할 수 있습니다. 이것을 컴파일 시 연결이라고 합니다.

기본 클래스의 일반적인 메소드는 파생된 클래스에 의해 상속됩니다. 다음 예제에서 *Derived* 타입의 객체는 *Regular()* 메소드를 자체 메소드처럼 호출할 수 있습니다. 클래스의 조상에 있는 일반적인 메소드와 이름 및 매개변수가 동일한 메소드를 파생된 클래스에서 선언하면 조상의 메소드가 *대체*됩니다. 다음 예제에서 *d->AnotherRegular()*가 호출되면 *AnotherRegular()*가 *Derived* 클래스로 대체되도록 디스패치됩니다.

```
class Base
{
public:
    void Regular();
    void AnotherRegular();
    virtual void Virtual();
};
class Derived : public Base
{
public:
    void AnotherRegular();          // replaces Base::AnotherRegular()
    void Virtual();                 // overrides Base::Virtual()
```

```
};
void FunctionOne()
{
  Derived *d;
  d = new Derived;
  d->Regular();                    // Calling Regular() as it were a
member of Derived
                                   // The same as calling d-
>Base::Regular()
  d->AnotherRegular();             // Calling the redefined
AnotherRegular(), ...
                                   // ... the replacement for
Base::AnotherRegular()
  delete d;
}
void FunctionTwo(Base *b)
{
  b->Virtual();
  b->AnotherRegular();
}
```

가상 메소드

컴파일 시에 연결되는 일반적인 메소드와 달리 가상 메소드는 *런타임*에 연결됩니다. C++의 가상 메커니즘을 사용하면 메소드를 호출하기 위해 사용하는 클래스 타입에 따라 메소드를 호출할 수 있습니다.

앞의 예에서 *Derived* 객체에 대한 포인터를 사용하여 *FunctionTwo()*를 호출하였으면 *Derived::Virtual()* 함수가 호출되었을 것입니다. 가상 메커니즘은 사용자가 런타임에 전달한 객체의 클래스 타입을 동적으로 확인하고 적절한 메소드를 디스패치합니다. 그러나 *AnotherRegular()*의 주소는 컴파일 시에 결정되므로 일반적인 함수 *b->AnotherRegular()*에 대한 호출은 항상 *Base::AnotherRegular()*를 호출합니다.

새 가상 메소드를 선언하려면 **virtual** 키워드로 메소드 선언을 시작해야 합니다.

컴파일러가 **virtual** 키워드를 발견하면 클래스의 가상 메소드 테이블(VMT)에 엔트리를 생성합니다. VMT는 클래스 내의 모든 가상 메소드의 주소를 보관합니다. 이 조회 테이블은 런타임에 사용되어 *b->Virtual*이 *Base::Virtual()*이 아니라 *Derived::Virtual()*을 호출해야 할지를 결정합니다.

기존 클래스에서 새 클래스를 파생시키면 새 클래스는 조상의 VMT에 있는 모든 엔트리와 새 클래스에서 추가로 선언된 가상 메소드를 포함하는 고유한 VMT를 갖게 됩니다. 또한 자손 클래스는 상속받은 모든 가상 메소드를 *오버라이드*할 수 있습니다.

메소드 오버라이드

메소드 오버라이드란 대체와는 달리 조상의 메소드를 확장하거나 조정하는 것을 의미합니다. 조상 클래스에 있는 메소드를 오버라이드하려면 인수의 수와 타입이 동일하도록 파생된 클래스에서 메소드를 다시 선언해야 합니다.

다음은 두 가지 간단한 컴포넌트의 선언을 보여 주는 코드입니다. 첫 번째 컴포넌트는 각각 다른 종류의 디스패치가 있는 두 개의 메소드를 선언하며 첫 번째 컴포넌트에서 파생된 두 번째 컴포넌트는 비가상 메소드를 대체하고 가상 메소드를 오버라이드합니다.

```
class PACKAGE TFirstComponent : public TComponent
{
public:
  void Move();                                  // regular method
  virtual void Flash();                         // virtual method
};
class PACKAGE TSecondComponent : public TFirstComponent
{
public:
  void Move();                // declares new method "hiding"
TFirstComponent::Move()
  void Flash();               // overrides virtual TFirstComponent::Flash
in TFirstComponent
};
```

추상 클래스 멤버

메소드가 조상 클래스에서 **abstract**로 선언된 경우에는 애플리케이션에서 새 컴포넌트를 사용하기 전에 반드시 자손 컴포넌트에서 다시 선언하고 구현해야 합니다. C++Builder는 추상 멤버를 포함하는 클래스의 인스턴스를 생성할 수 없습니다. 클래스의 상속된 부분을 제공하는 것에 대한 자세한 내용은 47장, "속성 생성" 및 49장, "메소드 생성"을 참조하십시오.

클래스 및 포인터

모든 클래스 및 그에 따른 모든 컴포넌트는 실제로 포인터입니다. 포인터로서의 클래스의 상태는 개발자가 클래스를 매개변수로 전달할 때 중요성을 갖습니다. 일반적으로 개발자는 참조보다는 값을 사용하여 클래스를 전달해야 합니다. 그 이유는 클래스가 이미 포인터이며 포인터는 또 참조이므로 클래스를 참조로 전달하는 것은 참조를 참조에 전달하는 결과가 되기 때문입니다.

장

47

속성 생성

속성은 컴포넌트에서 가장 시각적인 부분입니다. 애플리케이션 개발자는 디자인 타임에 속성을 조작하고, 폼 디자이너에서 컴포넌트를 확인하여 즉각적인 피드백을 얻을 수 있습니다. 속성을 잘 디자인하면 컴포넌트를 유지 보수하기가 쉽고, 다른 사용자도 쉽게 사용할 수 있습니다.

컴포넌트의 속성을 가장 적절하게 사용하려면 다음 사항을 이해해야 합니다.

- 속성을 만드는 이유
- 속성 타입
- 상속된 속성 게시
- 속성 정의
- 배열 속성 생성
- 속성 저장 및 로드

속성을 만드는 이유

애플리케이션 개발자의 관점에서 보면 속성은 변수와 비슷합니다. 개발자는 속성 값을 데이터 멤버처럼 설정하거나 읽을 수 있습니다. 변수로는 가능하지만 속성으로는 가능하지 않은 유일한 작업은 속성을 인수로 사용하여 참조에 의해 메소드로 전달하는 것입니다.

다음과 같은 이유에서 속성은 간단한 데이터 멤버보다 강력한 기능을 제공합니다.

- 애플리케이션 개발자는 디자인 타임에 속성을 설정할 수 있습니다. 런타임에만 사용할 수 있는 메소드와는 달리, 속성은 애플리케이션을 실행하기 전에 개발자가 컴포넌트를 사용자 정의할 수 있도록 합니다. 속성은 프로그래머의 작업을 단순화하는 Object Inspector에 표시될 수 있습니다. 즉, 객체를 생성하기 위해 여러 매개변수를 처리하는 대신 C++ Builder에서 Object Inspector의 값을 읽어올 수 있습니다. 또한 Object Inspector는 속성 할당이 이루어진 후 즉시 유효성을 검사합니다.

- 속성은 구현 세부 사항을 숨길 수 있습니다. 예를 들면, 내부적으로 암호화된 형식으로 저장된 데이터가 암호화되지 않은 속성 값으로 나타날 수 있습니다. 즉, 값이 간단한 숫자이더라도 컴포넌트는 데이터베이스에서 해당 값을 알아내거나 복잡한 계산을 수행하여 값에 도달합니다. 속성을 할당하면 겉으로 보기에는 간단하지만 복잡한 효과를 추가할 수 있습니다. 즉, 데이터 멤버에 대한 속성 할당이 복잡한 처리를 구현하는 메소드에 대한 호출일 수도 있습니다.
- 속성은 가상(virtual)일 수 있습니다. 따라서 애플리케이션 개발자에게 하나의 속성처럼 보이는 것이 다른 컴포넌트에서 다르게 구현될 수도 있습니다.

간단한 예로 모든 컨트롤에 나타나는 *Top* 속성을 들 수 있습니다. *Top*에 새 값을 할당하면 단순히 저장된 값을 변경하는 것이 아니라, 컨트롤이 재배치되고 다시 그려집니다. 또한 속성 설정의 효과를 개별 컴포넌트로 제한할 필요가 없습니다. 예를 들어, 스피드 버튼의 *Down* 속성을 **true**로 설정하면 해당 그룹에 있는 다른 모든 스피드 버튼의 *Down* 속성은 **false**로 설정됩니다.

속성 타입

속성은 어떤 타입이라도 될 수 있습니다. 타입이 다르면 Object Inspector에서 각기 다르게 표시되는데, Object Inspector에서는 디자인 타임에 속성이 할당될 때 그 유효성을 검사합니다.

표 47.1 Object Inspector에 속성이 표시되는 방법

속성 타입	Object Inspector 처리
일반	숫자, 문자 및 문자열 속성은 숫자, 문자 및 문자열로 표시됩니다. 애플리케이션 개발자는 속성 값을 직접 편집할 수 있습니다.
열거	부울을 포함한 열거 타입의 속성은 편집 가능한 문자열로 표시됩니다. 또한 개발자는 값 열을 더블 클릭하여 가능한 값을 볼 수 있습니다. 가능한 모든 값을 보여 주는 드롭다운 리스트가 있습니다.
집합	집합 타입의 속성은 집합으로 표시됩니다. 개발자는 속성을 더블 클릭하여 집합을 확장하고 각 요소를 부울 값(집합에 포함된 경우 **true**)으로 처리할 수 있습니다.
객체	자신이 클래스인 속성은 컴포넌트의 등록 프로시저에서 지정된 각각의 속성 에디터를 가집니다. 속성이 갖는 클래스에 고유한 published 속성이 있는 경우, 개발자는 Object Inspector에서 리스트를 더블 클릭하여 확장한 다음 이러한 속성을 포함시켜 개별적으로 편집할 수 있습니다. 객체 속성은 *TPersistent*의 자손이어야 합니다.
인터페이스	해당 값이 *TComponent*의 자손인 컴포넌트에 의해 구현된 인터페이스이면 인터페이스 속성은 Object Inspector에 표시될 수 있습니다. 인터페이스 속성은 각각의 속성 에디터를 가집니다.
배열	Object Inspector는 기본적으로 배열 속성의 편집을 지원하지 않으므로 배열 속성에는 각각의 속성 에디터가 있어야 합니다. 컴포넌트를 등록할 때 속성 에디터를 지정할 수 있습니다.

상속된 속성 게시

모든 컴포넌트는 조상 클래스로부터 속성을 상속합니다. 기존 컴포넌트에서 새 컴포넌트를 파생시키면 새 컴포넌트는 직계 조상의 모든 속성을 상속합니다. 추상 클래스에서 컴포넌트를 파생시킬 경우, 대부분의 상속된 속성은 published가 아니라 protected 또는 public입니다.

Object Inspector에서 디자인 타임에 protected 또는 public 속성을 사용할 수 있도록 하려면 해당 속성을 published로 재선언해야 합니다. 재선언은 상속된 속성에 대한 선언을 자손 클래스의 선언에 추가하는 것을 의미합니다.

예를 들어 *TWinControl*에서 VCL 컴포넌트를 파생시키면 protected *DockSite* 속성을 상속합니다. 이 때 새 컴포넌트에서 *DockSite*를 재선언하여 public 또는 published로 보호 수준을 변경할 수 있습니다.

다음 코드는 *DockSite*를 published로 재선언하여 디자인 타임에 사용할 수 있도록 만듭니다.

```
class PACKAGE TSampleComponent : public TWinControl
{
__published:
    __property DockSite;
};
```

속성을 재선언할 때는 속성 이름만 지정하고 "속성 정의"에서 설명하는 타입 및 기타 다른 정보는 지정하지 않습니다. 또한 새 기본값을 선언하고 속성 저장 여부를 지정할 수 있습니다.

재선언은 속성을 덜 제한할 수는 있지만 더 많이 제한할 수는 없습니다. 따라서 protected 속성을 public으로 만들 수는 있지만 public 속성을 protected로 재선언하여 이를 숨길 수는 없습니다.

속성 정의

이 단원에서는 새 속성을 선언하는 방법과 표준 컴포넌트에서 따라야 할 몇 가지 규칙을 설명합니다. 다음과 같은 사항에 대해 설명합니다.

- 속성 선언
- 내부 데이터 저장소
- 직접 액세스
- 액세스 메소드
- 디폴트 속성 값

속성 선언

속성은 컴포넌트 클래스의 선언에서 선언됩니다. 속성을 선언하기 위해서는 다음 세 가지 사항을 지정해야 합니다.

- 속성의 이름
- 속성의 타입
- 속성의 값을 읽고 쓰는 데 사용되는 메소드. 쓰기 메소드가 선언되어 있지 않은 경우 그 속성은 읽기 전용입니다.

컴포넌트 클래스 선언의 **__published** 섹션에서 선언된 속성은 디자인 타임 시 Object Inspector에서 편집할 수 있습니다. published 속성 값은 폼 파일에 컴포넌트와 함께 저장됩니다. **public** 섹션에서 선언된 속성은 런타임 시 사용할 수 있고 프로그램 코드에서 읽거나 설정할 수 있습니다.

다음 예제는 *Count*라는 속성에 대한 전형적인 선언의 예입니다.

```
class PACKAGE TYourComponent : public TComponent
{
private:
    int FCount;                                                  // data
member for storage
    int __fastcall GetCount();                                   // read
method
    void __fastcall SetCount( int ACount );                      // write
method
public:
    __property int Count = {read=GetCount, write=SetCount};   // property
declaration
  f
};
```

내부 데이터 저장소

속성 데이터를 저장하는 방법에는 제한이 없지만 일반적으로 C++Builder 컴포넌트에는 다음 규칙이 적용됩니다.

- 속성 데이터는 클래스 데이터 멤버에 저장됩니다.
- 속성 데이터를 저장하는 데 사용되는 데이터 멤버는 private이므로 컴포넌트 내부에서만 액세스해야 합니다. 파생된 컴포넌트는 상속된 속성을 사용해야 하기 때문에 속성의 내부 데이터 저장소에 직접 액세스할 필요가 없습니다.
- 이러한 데이터 멤버에 대한 식별자는 *F*로 시작하고 속성 이름이 뒤에 붙습니다. 예를 들어, *TControl*에 정의된 *Width* 속성의 원시 데이터는 *FWidth*라고 하는 데이터 멤버에 저장됩니다.

이러한 규칙의 토대가 되는 원칙은 속성에 대한 구현 메소드만이 관련 데이터에 액세스할 수 있다는 것입니다. 만일 메소드나 다른 속성이 특정 속성과 관련된 데이터를 변경할 필요가 있는경우, 해당 속성을 사용해야 하며 저장된 데이터에 직접 액세스하면 안됩니다. 이렇게 하면 파생된 컴포넌트를 무효로 만들지 않고 상속된 속성의 구현을 변경할 수 있습니다.

직접 액세스

*직접 액세스*는 속성 데이터를 사용할 수 있게 하는 가장 간단한 방법입니다. 즉, 속성 선언의 **read** 및 **write** 부분은 액세스 메소드를 호출하지 않고 속성 값의 할당 또는 읽기가 내부 저장소 데이터 멤버에서 직접 이루어지도록 지정합니다. 직접 액세스는 Object Inspector에서 속성을 사용할 수 있게 만들 때 유용하지만 속성 값에 대한 변경 내용은 즉시 처리되지 않습니다.

일반적으로 속성 선언의 **read** 부분에서는 직접 액세스를, **write** 부분에서는 액세스 메소드를 사용합니다. 이렇게 하면 속성 값이 바뀔 때 컴포넌트 상태가 업데이트됩니다.

다음과 같은 컴포넌트 타입의 선언은 **read** 및 **write** 부분 모두에서 직접 액세스를 사용하는 속성을 보여 줍니다.

```
class PACKAGE TSampleComponent : public TComponent
{
private:                                          // internal storage is
private
    bool FReadOnly;                               // declare data member to
hold value
  f
__published:                                      // make property
available at design time
    __property bool ReadOnly = {read=FReadOnly, write=FReadOnly};
};
```

액세스 메소드

속성 선언의 **read** 및 **write** 부분에서 데이터 멤버 대신 액세스 메소드를 지정할 수 있습니다. 액세스 메소드는 protected여야 하고 대개 **virtual**로 선언되기 때문에, 자손 컴포넌트는 속성의 구현을 오버라이드할 수 있습니다.

액세스 메소드를 public으로 만들어서는 안됩니다. 액세스 메소드를 protected로 유지하면 애플리케이션 개발자가 실수로 이러한 메소드 중 하나를 호출하여 속성을 수정하는 것을 방지할 수 있습니다.

다음 클래스는 모든 속성이 동일한 read 및 write 액세스 메소드를 갖도록 하는 인덱스 지정자를 사용하여 세 개의 속성을 선언합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
  int __fastcall GetDateElement(int Index);    // note Index parameter
  void __fastcall SetDateElement(int Index, int Value);
public:
  __property int Day = {read=GetDateElement, write=SetDateElement,
index=3, nodefault};
  __property int Month = {read=GetDateElement, write=SetDateElement,
index=2, nodefault};
  __property int Year = {read=GetDateElement, write=SetDateElement,
index=1, nodefault};
};
```

날짜의 각 요소(연/월/일)가 정 수이고 각 설정에 날 짜 인 코딩이 필요하기 때문에, 이 코드는 세 개의 속성 모두에 대해 읽기 및 쓰기 메소드를 공유하여 중복을 피합니다. 날짜 요소를 읽거나 쓸 때는 각기 하나의 메소드만 있으면 됩니다.

다음은 날짜 요소를 가져오는 read 메소드입니다.

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &Aday); // break date into elements
  switch (Index)
  {
```

```
      case 1: result = AYear;  break;
      case 2: result = AMonth; break;
      case 3: result = ADay;   break;
      default: result = -1;
    }
    return result;
  }
```

다음은 해당 날짜 요소를 설정하는 write 메소드입니다.

```
  void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
  {
    unsigned short AYear, AMonth, ADay;
    if (Value > 0) // all elements must be positive
    {
      FDate.DecodeDate(&AYear, &AMonth, &ADay); // get date elements
      switch (Index)
      {
        case 1: AYear = Value;  break;
        case 2: AMonth = Value; break;
        case 3: ADay = Value;   break;
        default: return;
      }
   }
   FDate = TDateTime(AYear, AMonth, ADay); // encode the modified date
   Refresh();// update the visible calendar
  }
```

read 메소드

속성에 대한 read 메소드는 아래에서 언급한 경우를 제외하면 매개변수를 사용하지 않고 속성과 동일한 타입의 값을 반환하는 함수입니다. 규칙에 따라 이 함수의 이름은 *Get*으로 시작하여 속성의 이름이 뒤에 붙습니다. 예를 들어, *Count*라는 속성에 대한 read 메소드는 *GetCount*입니다. read 메소드는 적절한 타입으로 속성 값을 만들기 위해 내부 저장소 데이터를 처리합니다.

예외적으로 배열 속성과 인덱스 지정자를 사용하는 속성(47-8페이지의 "배열 속성 생성" 참조)에는 매개변수를 사용하며, 두 속성 모두 인덱스 값을 매개변수로 전달합니다. 인덱스 지정자를 사용하여 여러 속성이 공유하는 단일 read 메소드를 만들 수도 있습니다.

read 메소드를 선언하지 않으면 속성은 쓰기 전용이 됩니다. 일반적으로 쓰기 전용 속성은 거의 사용되지 않습니다.

write 메소드

속성에 대한 write 메소드는 아래에서 언급한 경우를 제외하면 속성과 동일한 타입의 단일 매개변수를 취하는 멤버 함수입니다. 매개변수는 참조에 의해서 또는 값에 의해서 전달될 수 있고, 사용자가 선택한 어떤 이름도 가질 수 있습니다. 규칙에 따라 write 메소드의 이름은 *Set*으로 시작하여 속성의 이름이 뒤에 붙습니다. 예를 들어, *Count*라는 속성에 대한 write 메소드는 *SetCount*입니다. 매개변수로 전달된 값은 속성의 새 값이 되므로 write 메소드는 속성의 내부 저장소에 적절한 데이터를 두는 데 필요한 모든 조작을 수행해야 합니다.

예외적으로 배열 속성 및 인덱스 지정자를 사용하는 속성에는 단일 매개변수 규칙이 적용되지 않으며, 두 속성 모두 인덱스 값을 두 번째 매개변수로 전달합니다. 인덱스 지정자를 사용하여 여러 속성이 공유하는 단일 write 메소드를 만듭니다.

write 메소드를 선언하지 않으면 속성은 읽기 전용이 됩니다.

일반적으로 write 메소드는 속성을 변경하기 전에 새 값이 현재 값과 다른지 여부를 테스트합니다. 예를 들어, 다음은 *FCount*라고 하는 데이터 멤버에 현재 값을 저장하는 *Count*라는 정수 속성에 대한 간단한 write 메소드입니다.

```
void __fastcall TMyComponent::SetCount( int Value )
{
  if ( Value != FCount )
  {
    FCount = Value;
    Update();
  }
```

디폴트 속성 값

속성을 선언할 때 *기본값*을 지정할 수 있습니다. C++Builder는 기본값을 사용하여 폼 파일에 속성을 저장할지 여부를 결정합니다. 사용자가 속성에 대한 기본값을 지정하지 않으면 C++Builder는 항상 속성을 저장합니다.

속성에 대한 기본값을 선언하려면, 속성 이름 뒤에 등호를 붙이고 중괄호 안에 **default** 키워드와 기본값을 입력합니다. 예를 들면, 다음과 같습니다.

```
__property bool IsTrue = {default=true};
```

참고 기본값을 선언해도 속성이 기본값으로 설정되지는 않습니다. 컴포넌트의 생성자 메소드는 적절한 시기에 속성 값을 초기화해야 합니다. 그러나 객체는 항상 데이터 멤버를 0으로 초기화하기 때문에 생성자가 반드시 정수 속성을 0으로, 문자열 속성을 Null로, 부울 속성을 **false**로 설정할 필요는 없습니다.

기본값을 갖지 않도록 지정

상속된 속성이 기본값을 지정하더라도 속성을 재선언할 때 속성이 기본값을 갖지 않도록 지정할 수 있습니다.

속성에 기본값이 없도록 지정하려면 속성 이름 뒤에 등호를 붙이고 중괄호 안에 **nodefault** 키워드를 입력합니다. 예를 들면, 다음과 같습니다.

```
__property int NewInteger = {nodefault};
```

속성을 처음 선언할 때 **nodefault**를 포함할 필요는 없습니다. 선언된 기본값이 없다는 것은 기본값이 없음을 의미합니다.

다음에서는 기본값을 설정하는 생성자를 비롯하여 기본값이 **true** 인 *IsTrue* 라는 단일 부울 속성을 포함하는 컴포넌트를 선언합니다.

```
class PACKAGE TSampleComponent : public TComponent
{
private:
     bool FIsTrue;
```

```
public:
    virtual __fastcall TSampleComponent( TComponent* Owner );
__published:
    __property bool IsTrue = {read=FIsTrue, write=FIsTrue, default=true};
};
__fastcall TSampleComponent::TSampleComponent ( TComponent* Owner )
  : TComponent ( Owner )
{
  FIsTrue = true;
}
```

배열 속성 생성

몇몇 속성들은 배열처럼 인덱스를 붙일 수 있습니다. 예를 들어, *TMemo*의 *Lines* 속성은 메모의 텍스트를 구성하는 문자열의 인덱싱된 리스트며 문자열 배열로 처리할 수 있습니다. *Lines*는 더 큰 데이터(메모 텍스트) 집합에 있는 특정 요소(문자열)에 대한 일반적인 액세스를 제공합니다.

배열 속성은 다음 사항을 제외하고는 다른 속성과 똑같이 선언됩니다.

- 선언에는 지정된 타입의 인덱스가 하나 이상 포함됩니다. 인덱스는 모든 타입이 될 수 있습니다.
- 속성 선언의 **read** 및 **write** 부분을 지정할 경우 메소드여야 합니다. read 및 write는 데이터 멤버가 될 수 없습니다.

배열 속성에 대한 read 및 write 메소드는 인덱스에 해당하는 추가적인 매개변수를 갖습니다. 이 매개변수는 선언에서 지정한 인덱스와 동일한 순서 및 타입을 가져야 합니다.

배열 속성과 배열 사이에는 몇 가지 중요한 차이점이 있습니다. 배열의 인덱스와는 달리, 배열 속성의 인덱스는 정수 타입일 필요가 없습니다. 예를 들어, 문자열로 속성을 인덱싱할 수 있습니다. 또한 배열 속성에 대해서는 속성의 전체 범위가 아니라 개별 요소만 참조할 수 있습니다.

다음 예제는 정수 인덱스에 기초하여 문자열을 반환하는 속성 선언을 보여 줍니다.

```
class PACKAGE TDemoComponent : public TComponent
{
private:
    System::AnsiString __fastcall GetNumberSize(int Index);
public:
    __property System::AnsiString NumberSize[int Index] =
{read=GetNumberSize};
  f
};
```

.CPP 파일에 있는 *GetNumberSize* 메소드는 다음과 같습니다.

```
System::AnsiString __fastcall TDemoComponent::GetNumberSize(int Index)
{
  System::AnsiString Result;
```

```
  switch (Index)
  {
    case 0:
      Result = "Zero";
      break;
    case 100:
      Result = "Medium";
      break;
    case 1000:
      Result = "Large";
      break;
    default: Result = "Unknown size";
  }
  return Result;
}
```

하위 컴포넌트의 속성 생성

디폴트로, 속성 값이 또 다른 컴포넌트인 경우, 다른 컴포넌트의 인스턴스를 폼이나 데이터 모듈에 추가한 다음 해당 컴포넌트를 속성 값으로 할당하여 속성에 값을 할당합니다. 그러나 컴포넌트가 속성 값을 구현하는 객체의 고유한 인스턴스를 만들 수도 있습니다. 이러한 전용 컴포넌트를 하위 컴포넌트라고 합니다.

하위 컴포넌트는 영구적 객체, 즉 *TPersistent*의 자손이 될 수 있습니다. 속성 값으로 할당된 독립적인 컴포넌트와 달리, 하위 컴포넌트의 published 속성은 하위 컴포넌트를 만드는 컴포넌트와 함께 저장됩니다. 그러나 이러한 작업을 하기 위해서는 다음 조건이 충족되어야 합니다.

- 하위 컴포넌트의 *Owner*는 하위 컴포넌트를 만들어 published 속성 값으로 사용하는 컴포넌트여야 합니다. *TComponent*의 자손인 하위 컴포넌트의 경우, 하위 컴포넌트의 *Owner* 속성을 설정하여 이 작업을 수행할 수 있습니다. 다른 하위 컴포넌트의 경우에는 생성하는 컴포넌트를 반환하도록 영구적 객체의 *GetOwner* 메소드를 오버라이드해야 합니다.
- 하위 컴포넌트가 *TComponent*의 자손인 경우 *SetSubComponent* 메소드를 호출하여 하위 컴포넌트라는 사실을 나타내야 합니다. 일반적으로 이 메소드는 하위 컴포넌트를 만들 때 소유자가 호출하거나 하위 컴포넌트의 생성자가 호출합니다.

일반적으로 값이 하위 컴포넌트인 속성은 대개 읽기 전용입니다. 값이 하위 컴포넌트인 속성을 변경하도록 하는 경우, 속성 설정자는 다른 컴포넌트가 속성 값으로 할당될 때 하위 컴포넌트를 해제해야 합니다. 또한 속성이 NULL로 설정되면 컴포넌트는 종종 하위 컴포넌트를 다시 인스턴스화합니다. 그렇지 않으면 속성이 다른 컴포넌트로 바뀔 경우 디자인 타임에 하위 컴포넌트를 복원할 수 없습니다. 다음 예제는 *TTimer* 값을 가지는 속성에 대한 이러한 속성 설정자를 보여 줍니다.

```
void __fastcall TDemoComponent::SetTimerProp(ExtCtrls::TTimer *Value)
{
  if (Value != FTimer)
  {
```

```
    if (Value)
    {
      if (FTimer && FTimer->Owner == this)
        delete FTimer;
      FTimer = Value;
      FTimer->FreeNotification(this);
    }
    else // NULL value
    {
      if (FTimer && FTimer->Owner != this)
      {
        FTimer = new ExtCtrls::TTimer(this);
        FTimer.SetSubComponent(true);
        FTimer->FreeNotification(this);
      }
    }
  }
}
```

위의 속성 설정자가 속성 값으로 설정된 컴포넌트의 *FreeNotification* 메소드를 호출했다는 점에 유의합니다. 이 호출은 속성 값인 컴포넌트가 소멸되려는 순간에 통지를 보내도록 합니다. *Notification* 메소드를 호출하여 이러한 통지를 보냅니다. 다음과 같이 *Notification* 메소드를 오버라이드하여 이 호출을 처리합니다.

```
void __fastcall TDemoComponent::Notification(Classes::TComponent
*AComponent, Classes::TOperation Operation)
{
  TComponent::Notification(AComponent, Operation); { call inherited
method }
  if ((Operation == opRemove) && (AComponent == (TComponent *)FTimer))
    FTimer = NULL;
}
```

속성 저장 및 로드

C++Builder는 폼 파일(VCL의 .dfm 및 CLX의 .xfm)에 폼 및 해당 컴포넌트를 저장합니다. 폼 파일은 폼과 컴포넌트의 속성을 저장합니다. C++Builder 개발자가 여러분이 작성한 컴포넌트를 폼에 추가할 경우, 이 컴포넌트는 저장할 때 컴포넌트의 속성을 폼 파일에 기록할 수 있어야 합니다. 마찬가지로 컴포넌트가 C++Builder로 로드되거나 애플리케이션의 일부로 실행될 때 그 컴포넌트는 폼 파일로부터 자신을 복원할 수 있어야만 합니다.

속성을 저장하고 폼 파일에서 로드하는 기능은 컴포넌트의 상속된 동작의 일부이기 때문에 대개의 경우 컴포넌트를 폼 파일에서 작동시키기 위해 추가로 할 일은 없습니다. 하지만 가끔 컴포넌트가 자신을 저장하는 방법이나 로드 시 초기화하는 방법을 변경할 수도 있기 때문에 기초가 되는 메커니즘을 이해해야 합니다.

속성 저장에 대해 이해가 필요한 측면은 다음과 같습니다.

- 저장 및 로드 메커니즘 사용
- 기본값 지정

- 저장할 대상 결정
- 로드 후 초기화
- published가 아닌 속성 저장 및 로드

저장 및 로드 메커니즘 사용

폼에 대한 설명은 폼에 있는 각 컴포넌트에 대한 유사한 설명과 폼 속성 리스트로 구성됩니다. 폼을 비롯한 모든 컴포넌트는 각각의 설명을 저장하고 로드합니다.

디폴트로, 컴포넌트가 저장될 때는 기본값과 다른 모든 published 속성의 값이 선언된 순서대로 기록됩니다. 컴포넌트가 로드될 때는 우선 모든 속성을 기본값으로 설정하면서 컴포넌트가 생성된 다음 기본값이 아닌 저장된 속성 값을 읽어옵니다.

이 디폴트 메커니즘에서는 거의 모든 컴포넌트의 요구가 충족되고, 컴포넌트 작성자가 작업을 수행할 필요가 전혀 없습니다. 그러나 특정 컴포넌트의 요구를 충족시키기 위해서 여러 가지 방법으로 저장 및 로드 프로세스를 사용자 정의할 수 있습니다.

기본값 지정

C++Builder 컴포넌트는 속성 값이 기본값과 다를 경우에만 저장합니다. 기본값이 지정되지 않으면, C++Builder는 속성에 기본값이 없다고 가정합니다. 따라서 컴포넌트는 그 값에 상관없이 항상 속성을 저장합니다.

다음과 같은 방법으로 속성에 대한 기본값을 지정합니다.

1 속성 이름 뒤에 등호(=)를 추가합니다.

2 등호 뒤에는 중괄호({})를 추가합니다.

3 중괄호 안에 키워드 **default**와 그 뒤에 등호를 입력합니다.

4 새 기본값을 지정합니다.

예를 들면, 다음과 같습니다.

```
__property Alignment = {default=taCenter};
```

또한 속성을 재선언할 때 기본값을 지정할 수도 있습니다. 실제로 속성을 재선언하는 이유 중 하나가 다른 기본값을 지정하기 위해서 입니다.

참고 기본값을 지정한다고 해서 기본값이 자동으로 속성에 할당되는 것은 아닙니다. 따라서 컴포넌트 생성자가 필요한 값을 할당하도록 해야 합니다. 컴포넌트 생성자가 값을 설정하지 않은 속성은 0 값, 즉 저장소 메모리가 0으로 설정될 때 가질 수 있는 값을 갖습니다. 따라서 숫자 값에서는 0, 부울 값에서는 **false**, 그리고 포인터에서는 **NULL**이 기본값이 됩니다. 확실하지 않은 경우에는, 생성자 메소드에 있는 값을 할당합니다.

다음 코드는 *Align* 속성의 기본값을 지정하는 컴포넌트 선언과 기본값을 설정하는 컴포넌트 생성자의 구현을 보여 줍니다. 여기서 새 컴포넌트는 윈도우에서 상태 표시줄에 사용되는 표준 패널 컴포넌트의 특별한 경우이기 때문에 디폴트 정렬은 소유자의 맨 아래쪽이어야 합니다.

```
class PACKAGE TMyStatusBar : public TPanel
{
public:
  virtual __fastcall TMyStatusBar(TComponent* AOwner);
__published:
  __property Align = {default=alBottom};
};
```

TMyStatusBar 컴포넌트의 생성자는 .CPP 파일에 있습니다.

```
__fastcall TMyStatusBar::TMyStatusBar (TComponent* AOwner)
  : TPanel(AOwner)
{
   Align = alBottom;
}
```

저장할 대상 결정

C++Builder가 각 컴포넌트의 속성을 저장하는지 여부를 제어할 수 있습니다. 디폴트로, 클래스 선언의 published 부분에 있는 모든 속성이 저장됩니다. 지정한 속성을 저장하지 않도록 선택하거나 속성 저장 여부를 동적으로 결정하는 함수를 지정할 수 있습니다.

다음과 같은 방법으로 C++Builder에서 속성 저장 여부를 제어합니다.

1 속성 이름 뒤에 등호(=)를 추가합니다.

2 등호 뒤에는 중괄호({})를 추가합니다.

3 중괄호 안에 키워드 **stored**와 그 뒤에 **true**, **false** 또는 부울 함수의 이름을 입력합니다.

다음 코드는 세 개의 새로운 속성을 선언하는 컴포넌트를 보여 줍니다. 이 중 하나는 항상 저장되고, 다른 하나는 절대 저장되지 않으며, 마지막 하나는 부울 함수 값에 따라 저장 여부가 결정됩니다.

```
class PACKAGE TSampleComponent : public TComponent
{
protected:
  bool __fastcall StoreIt();
public:
   f
__published:
  __property int Important = {stored=true};         // always stored
  __property int Unimportant = {stored=false};      // never stored
  __property int Sometimes = {stored=StoreIt};      // storage depends on
function value
};
```

로드 후 초기화

컴포넌트는 저장된 설명에서 모든 속성 값을 읽은 후 필요한 모든 초기화를 수행하는 *Loaded* 라고 하는 가상 메소드를 호출합니다. *Loaded*는 폼과 컨트롤이 나타나기 전에 호출되므로 초기화 시 화면이 깜빡일 염려가 없습니다.

속성 값이 로드된 후 컴포넌트를 초기화하기 위해서는 *Loaded* 메소드를 오버라이드합니다.

참고 *Loaded* 메소드에서 가장 먼저 해야 할 일은 상속된 *Loaded* 메소드를 호출하는 것입니다. 이렇게 하면 자신의 컴포넌트를 초기화하기 전에 상속된 모든 속성을 올바르게 초기화할 수 있습니다.

published가 아닌 속성 저장 및 로드

디폴트로, published 속성들만 컴포넌트와 함께 로드되고 저장됩니다. 하지만 published가 아닌 속성을 로드하고 저장하는 것도 가능합니다. 이는 Object Inspector에 나타나지 않는 영구적 속성을 가질 수 있도록 합니다. 또한 속성 값이 너무 복잡하기 때문에 컴포넌트가 C++Builder에서 읽거나 쓰는 방법을 알지 못하는 속성 값을 저장하고 로드할 수 있게 합니다. 예를 들어, *TStrings* 객체는 C++Builder의 자동 동작에 의존하여 자신이 나타내는 문자열을 저장 및 로드할 수 없고 다음과 같은 메커니즘을 사용해야만 합니다.

속성 값의 로드 및 저장 방법을 C++Builder에게 알려주는 코드를 추가하여 published가 아닌 속성을 저장할 수 있습니다.

속성을 로드하고 저장하는 코드를 작성하려면 다음 단계를 따르십시오.

1 속성 값을 저장하고 로드하는 메소드들을 만듭니다.

2 *DefineProperties* 메소드를 파일러 객체에 전달하여 오버라이드합니다.

속성 값을 저장 및 로드하는 메소드 생성

published가 아닌 속성을 저장하고 로드하려면 우선 속성 값을 저장하고 로드하기 위한 메소드를 각각 만들어야 합니다. 다음 두 가지 방법 중 하나를 선택할 수 있습니다.

- *TWriterProc* 타입의 메소드를 만들어 속성 값을 저장하고, *TReaderProc* 타입의 메소드를 만들어 속성 값을 로드합니다. 이 방법은 일반 타입을 저장하고 로드할 수 있는 C++Builder의 기본 기능을 이용할 수 있습니다. C++Builder가 저장하고 로드하는 방법을 아는 타입으로 속성 값을 만드는 경우 이 방법을 사용합니다.
- *TStreamProc* 타입의 메소드를 두 개 만들어 속성 값을 저장하고 로드하는 데 각각 하나씩 사용합니다. *TStreamProc*는 스트림을 인수로 가지며, 이 스트림의 메소드를 사용하여 속성 값을 읽고 쓸 수 있습니다.

예를 들어, 런타임 시 생성되는 컴포넌트를 나타내는 속성을 고려해 봅니다. C++Builder는 이 값을 기록하는 방법을 알고 있지만 폼 디자이너에서 컴포넌트를 만들었기 때문에 자동으로 기록하지는 않습니다. 스트리밍 시스템은 이미 컴포넌트를 로드하고 저장할 수 있으므로 첫 번째 방법을 사용할 수 있습니다. 다음 메소드는 *MyCompProperty*라고 하는 속성의 값인 동적으로 생성된 컴포넌트를 로드하고 저장합니다.

```
void __fastcall TSampleComponent::LoadCompProperty(TReader *Reader)
{
  if (Reader->ReadBoolean())
    MyCompProperty = Reader->ReadComponent(NULL);
}
void __fastcall TSampleComponent::StoreCompProperty(TWriter *Writer)
{
  if (MyCompProperty)
  {
    Writer->WriteBoolean(true);
    Writer->WriteComponent(MyCompProperty);
  }
  else
    Writer->WriteBoolean(false);
}
```

DefineProperties 메소드 오버라이드

속성 값을 저장하고 로드하기 위한 메소드를 만들었다면 컴포넌트의 *DefineProperties* 메소드를 오버라이드할 수 있습니다. C++Builder는 컴포넌트를 저장하거나 로드할 때 이 메소드를 호출합니다. *DefineProperties* 메소드에서는 현재 파일러의 *DefineProperty* 또는 *DefineBinaryProperty* 메소드를 호출하여, 속성 값을 저장하거나 로드하는 데 사용할 수 있도록 파일러에 이러한 메소드를 전달해야 합니다. 저장 및 로드 메소드가 각각 *TWriterProc* 및 *TReaderProc*인 경우, *DefineProperty* 메소드를 호출합니다. *TStreamProc* 타입의 메소드를 만든 경우, *DefineBinaryProperty*를 호출합니다.

속성을 정의하는 데 사용하는 메소드에 상관 없이 속성 값을 저장하고 로드하는 메소드와 속성 값의 기록 여부를 나타내는 부울 값을 파일러에 전달합니다. 값을 상속할 수 있거나 기본값이 지정된 경우에는 속성 값을 기록할 필요가 없습니다.

예를 들어, *TReaderProc* 타입의 *LoadCompProperty* 메소드와 *TWriterProc* 타입의 *StoreCompProperty* 메소드가 있는 경우, 다음과 같이 *DefineProperties*를 오버라이드합니다.

```
void __fastcall TSampleComponent::DefineProperties(TFiler *Filer)
{
  // before we do anything, let the base class define its properties.
  // Note that this example assumes that TSampleComponent derives
directly from TComponent
  TComponent::DefineProperties(Filer);
  bool WriteValue;
  if (Filer->Ancestor) // check for inherited value
  {
    if ((TSampleComponent *)Filer->Ancestor)->MyCompProperty == NULL)
      WriteValue = (MyCompProperty != NULL);
    else if ((MyCompProperty == NULL) ||
             (((TSampleComponent *)Filer->Ancestor)->MyCompProperty->Name
!=
                                                    MyCompProperty-
>Name))
      WriteValue = true;
    else WriteValue = false;
  }
```

```
  else // no inherited value, write property if not null
    WriteValue = (MyCompProperty != NULL);
  Filer->DefineProperty("MyCompProperty
",LoadCompProperty,StoreCompProperty, WriteValue);
end;
```

장

48

이벤트 생성

이벤트는 사용자의 동작이나 포커스 변경 등과 같은 시스템에서 발생하는 사건과 이 사건에 응답하는 코드 부분 간의 연결입니다. 응답 코드는 *이벤트 핸들러*이고, 거의 대부분 애플리케이션 개발자에 의해 작성됩니다. 이벤트를 통해 애플리케이션 개발자는 클래스 자체를 변경하지 않고도 컴포넌트의 동작을 사용자 정의할 수 있습니다. 이것을 *위임(delegation)*이라고 합니다.

마우스 동작과 같은 가장 일반적인 사용자 동작에 대한 이벤트는 모든 표준 컴포넌트에 내장되어 있지만 사용자가 새로운 이벤트를 정의할 수도 있습니다. 컴포넌트에서 이벤트를 만들려면 다음의 사항들을 이해할 필요가 있습니다.

- 이벤트 정의
- 표준 이벤트 구현
- 새로운 이벤트 정의

이벤트는 속성으로 구현되므로 컴포넌트의 이벤트를 만들거나 변경하기 전에 47장, "속성 생성"의 내용을 먼저 이해해야 합니다.

이벤트 정의

이벤트는 사건을 어떤 코드에 연결하는 메커니즘입니다. 보다 자세히 말하면, 이벤트는 특정 클래스 인스턴스의 메소드를 가리키는 클로저(closure)입니다.

애플리케이션 개발자의 관점에서 보면 이벤트는 특정 코드를 추가할 수 있는 *OnClick*과 같은 시스템의 사건에 관련된 이름일 뿐입니다. 예를 들어, *Button1*이라고 하는 푸시 버튼은 *OnClick* 메소드를 가집니다. 디폴트로, C++Builder는 버튼을 포함하는 폼에 *Button1Click*이라는 이벤트 핸들러를 만들고 이를 *OnClick*에 할당합니다. 클릭 이벤트가 버튼에서 발생하면 버튼은 *OnClick*에 할당된 메소드, 즉 이 경우 *Button1Click*을 호출합니다.

이벤트를 작성하려면 다음을 이해해야 합니다.

- 이벤트는 클로저(closure)
- 이벤트는 속성
- 이벤트 타입은 클로저(closure) 타입
- 이벤트 핸들러는 void의 반환 타입을 가짐
- 이벤트 핸들러는 옵션

이벤트는 클로저(closure)

C++Builder는 클로저(closure)를 사용하여 이벤트를 구현합니다. 클로저(closure)는 특정 클래스 인스턴스의 메소드를 가리키는 특수 포인터 타입입니다. 컴포넌트 작성자는 클로저(closure)를 위치 표시자로 취급할 수 있습니다. 코드에서 이벤트가 발생했음을 탐지하면 해당 이벤트에 대해 사용자가 지정한 메소드가 있는 경우, 이를 호출합니다.

클로저(closure)는 클래스 인스턴스에 대한 숨겨진 포인터를 가지고 있습니다. 사용자가 핸들러를 컴포넌트 이벤트에 할당하면 특정 이름을 갖는 메소드뿐만 아니라 특정 클래스 인스턴스의 특정 메소드에 대해서도 할당이 적용됩니다. 이러한 인스턴스는 일반적으로 컴포넌트를 포함하는 폼이지만 반드시 그럴 필요는 없습니다.

예를 들어, 모든 컨트롤은 클릭 이벤트를 처리하기 위해 *Click*이라는 가상 메소드를 상속합니다.

```
virtual void __fastcall Click(void);
```

*Click*의 구현은 사용자의 클릭 이벤트 핸들러가 있을 경우 이를 호출합니다. 사용자가 컨트롤의 *OnClick* 이벤트에 핸들러를 할당한 경우, 컨트롤을 클릭하면 해당 메소드가 호출됩니다. 핸들러를 할당하지 않은 경우에는 아무 일도 일어나지 않습니다.

이벤트는 속성

컴포넌트는 속성을 사용하여 이벤트를 구현합니다. 대부분의 다른 속성과 달리, 이벤트는 일반적으로 읽기 및 쓰기 부분을 구현하는 데 메소드를 사용하지 않습니다. 그 대신, 이벤트 속성은 속성과 동일한 타입의 private 데이터 멤버를 사용합니다.

규칙에 따라 데이터 멤버의 이름은 속성의 이름과 동일하지만 문자 *F*가 이름 앞에 붙습니다. 예를 들면, *OnClick* 클로저(closure)는 *TNotifyEvent* 타입의 *FOnClick*이라고 하는 데이터 멤버에 저장되고, *OnClick* 이벤트 속성의 선언은 다음과 같습니다.

```
class PACKAGE TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
  f
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
  f
};
```

TNotifyEvent 및 다른 이벤트 타입에 대한 내용은 다음의 "이벤트 타입은 클로저(closure) 타입" 단원을 참조하십시오.

다른 속성과 마찬가지로, 런타임 시 이벤트 값을 설정하거나 변경할 수 있습니다. 그러나 이벤트를 속성으로 만드는 가장 큰 이점은 컴포넌트 사용자가 Object Inspector를 사용하여 디자인 타임에 핸들러를 이벤트에 할당할 수 있다는 것입니다.

이벤트 타입은 클로저(closure) 타입

이벤트는 이벤트 핸들러에 대한 포인터이기 때문에 이벤트 속성의 타입은 클로저(closure) 타입이어야 합니다. 마찬가지로, 이벤트 핸들러로 사용된 모든 코드는 적절하게 타입이 지정된 클래스 메소드여야 합니다.

주어진 타입의 이벤트와 호환 가능하려면 이벤트 핸들러 메소드는 매개변수의 수와 타입이 동일하고 동일한 순서와 방식으로 전달되어야 합니다.

C++Builder는 모든 표준 이벤트의 클로저(closure)를 정의합니다. 사용자 고유의 이벤트를 만들 경우, 적절한 기존 클로저(closure)를 사용하거나 사용자 고유 클로저(closure)를 정의할 수 있습니다.

이벤트 핸들러의 반환 타입은 void

이벤트 핸들러의 반환 타입은 void만 될 수 있습니다. 핸들러에서 void만 반환하더라도 참조로 인수를 전달하여 사용자 코드에서 정보를 다시 가져올 수 있습니다. 이런 작업을 할 때 사용자 코드에서 값을 변경할 필요가 없도록 핸들러를 호출하기 전에 인수에 유효한 값을 할당합니다.

TKeyPressEvent 타입의 키 누름(key-press) 이벤트는 참조로 인수를 이벤트 핸들러에 전달하는 예입니다. *TKeyPressEvent*는 두 개의 인수를 정의합니다. 한 인수는 이벤트를 생성한 객체를 나타내고, 다른 인수는 눌러진 키를 나타냅니다.

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender, Char
&Key);
```

보통 *Key* 매개변수는 사용자가 누른 문자를 포함합니다. 그러나 어떤 상황에서는 컴포넌트 사용자가 문자를 변경하고자 할 수도 있습니다. 편집 컨트롤에서 모든 문자를 대문자로 바꾸려 할 때가 이러한 경우입니다. 이 경우, 사용자는 키스트로크에 대한 다음과 같은 핸들러를 정의할 수 있습니다.

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
  Key = UpCase(Key);
}
```

사용자가 디폴트 처리를 오버라이드할 수 있도록 참조에 의해 전달된 인수를 사용할 수 있습니다.

이벤트 핸들러는 옵션

이벤트를 만들 때, 컴포넌트를 사용하는 개발자가 핸들러를 추가하지 않을 수도 있다는 것을 고려해야 합니다. 이는 특정 이벤트에 연결된 핸들러가 없으므로 컴포넌트가 실패하거나 오류를 발생시키지 않아야 한다는 것을 의미합니다. 핸들러를 호출하고 연결된 핸들러가 없는 이벤트를 처리하는 메커니즘에 대해서는 48-8페이지의 "이벤트 호출"을 참조합니다.

이벤트는 GUI 애플리케이션에서 거의 항상 발생합니다. 비주얼 컴포넌트 위에서 마우스 포인터를 이동하기만 해도 수많은 마우스 이동 메시지를 보내며, 컴포넌트는 이 메시지를 *OnMouseMove* 이벤트로 번역합니다. 대부분의 경우 개발자는 마우스 이동 이벤트를 처리하지

않으며, 이것이 문제가 되어서는 안됩니다. 따라서 생성된 컴포넌트는 각각의 이벤트에 대한 핸들러를 요구해서는 안됩니다.

또한 애플리케이션 개발자는 이벤트 핸들러에서 원하는 모든 코드를 작성할 수 있습니다. VCL 및 CLX 의 컴포넌트에서는 이벤트 핸들러가 오류를 발생하는 경우를 최소화하도록 이벤트가 작성됩니다. 애플리케이션 코드에서 로직 오류는 막을 수 없지만, 애플리케이션 개발자가 유효하지 않은 데이터를 액세스하지 않도록 이벤트를 호출하기 전에 데이터 구조를 초기화할 수 있습니다.

표준 이벤트 구현

C++Builder에서 기본적으로 제공하는 컨트롤은 가장 일반적인 사건에 대한 이벤트를 상속합니다. 이러한 이벤트를 *표준 이벤트*라고 합니다. 이러한 이벤트가 모두 컨트롤에 내장되어 있더라도 **protected**로 설정된 경우가 많으므로 개발자가 핸들러를 추가할 수 없습니다. 컨트롤을 만들 때 컨트롤 사용자가 이벤트를 볼 수 있도록 선택할 수 있습니다.

표준 이벤트를 컨트롤에 통합시킬 때 고려해야 할 사항은 다음 세 가지입니다.

- 표준 이벤트 식별
- 이벤트 표시
- 표준 이벤트 처리 변경

표준 이벤트 식별

표준 이벤트의 두 가지 범주에는 모든 컨트롤에 대해 정의된 이벤트와 표준 윈도우 컨트롤에 대해 정의된 이벤트가 있습니다.

모든 컨트롤에 대한 표준 이벤트

대부분의 기본 이벤트는 *TControl* 클래스에서 정의됩니다. 윈도우 컨트롤, 그래픽 컨트롤 또는 사용자 정의 컨트롤에 상관 없이 모든 컨트롤이 이러한 이벤트를 상속받습니다. 다음 이벤트는 모든 컨트롤에서 사용 가능합니다.

OnClick	*OnDragDrop*	*OnEndDrag*	*OnMouseMove*
OnDblClick	*OnDragOver*	*OnMouseDown*	*OnMouseUp*

표준 이벤트에는 이벤트 이름에 해당하는 이름을 갖고 *TControl*에서 선언되는 해당 protected 가상 메소드가 있습니다. 예를 들어, *OnClick* 이벤트는 *Click*이라는 메소드를 호출하고, *OnEndDrag* 이벤트는 *DoEndDrag*라는 메소드를 호출합니다.

표준 컨트롤에 대한 표준 이벤트

모든 컨트롤에 대한 일반적인 이벤트를 비롯하여 표준 윈도우 컨트롤(VCL의 *TWinControl* 및 CLX의 *TWidgetControl*의 자손)은 다음 이벤트를 가집니다.

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

*TControl*의 표준 이벤트와 마찬가지로, 윈도우 컨트롤 이벤트는 해당 메소드를 가집니다. 위에 나열된 표준 키 이벤트는 일반적인 모든 키스트로크에 응답합니다.

VCL 그러나 Alt 키와 같은 특수 키스트로크에 응답하게 하려면 Windows에서 WM_GETDLGCODE 또는 CM_WANTSPECIALKEYS 메시지에 응답해야 합니다. 메시지 핸들러 작성에 대한 내용은 51장, "메시지 및 시스템 통지 처리"를 참조하십시오.

이벤트 표시

TControl 및 *TWinControl*(CLX의 *TWidgetControl*)의 표준 이벤트 선언은 protected이고, 그에 해당하는 메소드 역시 protected입니다. 이러한 추상 클래스 중 하나에서 상속하고 런타임 또는 디자인 타임에 이벤트를 액세스할 수 있도록 하려면, 이벤트를 public 또는 published로 재선언해야 합니다.

구현을 지정하지 않고 속성을 재선언하면 동일한 구현 메소드가 유지되지만 보호 레벨이 변경됩니다. 그러므로 *TControl*에서 정의된 보이지 않는 이벤트를 가져와서 public 또는 published로 선언하여 이벤트를 제공할 수 있습니다.

예를 들어, 디자인 타임에 *OnClick* 이벤트를 제공하는 컴포넌트를 만들려면 컴포넌트의 클래스 선언에 다음을 추가합니다.

```
class PACKAGE TMyControl : public TCustomControl
{
  f
__published:
    __property OnClick;             // Makes OnClick available in the
Object Inspector
};
```

표준 이벤트 처리 변경

컴포넌트가 특정 유형의 이벤트에 응답하는 방식을 변경할 경우, 특정 코드를 작성하여 이벤트에 할당하고자 할 수 있습니다. 이것이 바로 애플리케이션 개발자가 하는 일입니다. 그러나 컴포넌트를 만들 때 컴포넌트를 사용하는 다른 개발자들이 이벤트를 사용할 수 있도록 해야 합니다.

이것이 각 표준 이벤트에 연결된 protected 구현 메소드가 있는 이유입니다. 구현 메소드를 오버라이드하면 내부 이벤트 처리를 수정할 수 있으며, 상속된 메소드를 호출하면 애플리케이션 개발자 코드의 이벤트를 포함하여 표준 처리를 유지할 수 있습니다.

메소드를 호출하는 순서는 중요합니다. 일반적으로 상속된 메소드를 먼저 호출하여 애플리케이션 개발자의 이벤트 핸들러가 자신의 사용자 정의 이전에 실행될 수 있게 하거나 사용자 정의가 아예 실행되지 않도록 합니다. 그러나 상속된 메소드를 호출하기 전에 자신의 코드를 먼저 실행하려는 경우도 있습니다. 예를 들어, 상속된 코드가 컴포넌트의 상태에 종속적이고 사용자 코드로 이 상태를 변경하는 경우에는, 상태를 변경한 다음 사용자 코드가 변경 내용에 응답하도록 해야 합니다.

컴포넌트를 작성하고 마우스 클릭에 응답하는 방식을 수정하려 한다고 가정합니다. 애플리케이션 개발자가 하는 것과 같이 *OnClick* 이벤트에 핸들러를 할당하는 대신 protected 메소드 *Click*을 오버라이드합니다.

```
void __fastcall TMyControl::Click()
{
  TWinControl::Click();        // perform standard handling, including
calling handler
  // your customizations go here
}
```

새로운 이벤트 정의

일반적으로 완전히 새로운 이벤트를 정의하는 경우는 드뭅니다. 그러나 컴포넌트가 다른 컴포넌트의 동작과 완전히 다른 동작을 수행할 때에는 그에 대한 이벤트를 정의해야 할 필요가 있습니다.

이벤트를 정의할 때 고려해야 할 사항은 다음과 같습니다.

- 이벤트 트리거
- 핸들러 타입 정의
- 이벤트 선언
- 이벤트 호출

이벤트 트리거

이벤트를 실행하는 것이 무엇인지 알아야 합니다. 일부 이벤트의 경우에는 그 대답이 명백합니다. 예를 들어, 마우스 다운(mouse-down) 이벤트는 마우스의 왼쪽 버튼을 누를 때 일어나고 Windows는 *WM_LBUTTONDOWN* 메시지를 애플리케이션에 보냅니다. 메시지를 받으면 컴포넌트는 *MouseDown* 메소드를 호출한 다음 *OnMouseDown* 이벤트에 연결된 모든 코드를 호출합니다.

하지만 몇몇 이벤트는 특정 외부 사건에 대해 명확하게 연결되지 않습니다. 예를 들어 스크롤 막대는 *OnChange* 이벤트를 가지는데, 이 이벤트는 키스트로크, 마우스 클릭 및 다른 컨트롤에서의 변경 사항 등을 포함한 여러 가지 종류의 사건에 의해 실행됩니다. 이벤트를 정의할 경우 발생한 모든 사건이 적절한 해당 이벤트를 호출하는지 확인해야 합니다.

CLX CLX 애플리케이션에 대한 내용은 51-10페이지의 "CLX를 사용하여 시스템 통지에 응답"을 참조하십시오.

두 종류의 이벤트

이벤트를 제공해야 하는 두 가지 경우로, 사용자 조작과 상태 변경이 있습니다. 사용자 조작 이벤트는 거의 항상 Windows의 메시지에 의해 실행되며, 사용자가 컴포넌트의 응답이 필요한 작업을 수행했음을 나타냅니다. 상태 변경 이벤트도 포커스 변경이나 활성화처럼 Windows의 메시지와 관련될 수 있지만 속성이나 다른 코드의 변경에 의해 발생할 수도 있습니다.

자신이 정의하는 이벤트의 실행을 완전히 제어할 수 있습니다. 개발자가 이해하고 사용할 수 있도록 이벤트를 신중하게 정의해야 합니다.

핸들러 타입 정의

이벤트 발생 시기를 결정한 후 이벤트 처리 방법을 정의해야 합니다. 즉, 이벤트 핸들러의 타입을 결정해야 합니다. 대부분의 경우 자신이 직접 정의한 이벤트의 핸들러는 일반적인 통지 타입 또는 이벤트별 타입입니다. 핸들러에서 거꾸로 정보를 얻는 것도 가능합니다.

일반적인 통지

통지 이벤트는 이벤트 발생 시기나 위치에 대한 자세한 정보 없이 특정 이벤트가 발생했다는 것만을 알려주는 이벤트입니다. 통지 이벤트는 단 하나의 매개변수, 이벤트를 보내는 사람만을 알리는 *TNotifyEvent* 타입을 사용합니다. 통지를 위한 모든 핸들러는 이벤트가 어떤 타입인지, 어떤 컴포넌트에서 이벤트가 발생했는지에 대해 "알고" 있습니다. 예를 들어, 클릭 이벤트는 통지에 해당합니다. 클릭 이벤트의 핸들러를 작성할 때는 클릭이 발생했으며, 어떤 컴포넌트를 클릭했는지만 알 수 있습니다.

통지는 단방향 프로세스입니다. 피드백을 제공하거나 통지가 더 이상 처리되지 않도록 하는 메커니즘은 없습니다.

이벤트별 핸들러

어떤 경우에는 발생한 이벤트와 이벤트가 발생한 컴포넌트를 아는 것만으로 충분하지 않습니다. 예를 들어, 이벤트가 키 누름 이벤트인 경우 핸들러에서는 사용자가 누른 키를 알아야 합니다. 이러한 경우, 추가 정보를 위한 매개변수를 포함하는 핸들러 타입이 있어야 합니다.

메시지에 응답하여 이벤트가 생성된 경우 이벤트 핸들러에 전달하는 매개변수는 메시지 매개변수에서 직접 가져올 수 있습니다.

핸들러의 정보 반환

모든 이벤트 핸들러가 void만 반환하므로 핸들러에서 정보를 반환하는 유일한 방법은 참조에 의해 전달된 매개변수를 사용하는 것입니다. 컴포넌트는 이 정보를 사용하여 사용자의 핸들러가 실행된 후 이벤트를 처리할지 여부 및 그 방법을 결정합니다.

예를 들어, 모든 키 이벤트(*OnKeyDown*, *OnKeyUp* 및 *OnKeyPress*)는 눌러진 키 값을 *Key*라는 매개변수에 넣어 참조에 의해 전달합니다. 이벤트 핸들러는 애플리케이션에서 이벤트에 관련된 서로 다른 키를 볼 수 있도록 *Key*를 변경할 수 있습니다. 예를 들면, 이렇게 함으로써 입력된 문자를 대문자로 바꿀 수 있습니다.

이벤트 선언

이벤트 핸들러의 타입을 결정하면 클로저(closure) 및 이벤트에 대한 속성을 선언할 수 있습니다. 사용자가 이벤트가 수행하는 작업을 잘 알 수 있도록 이벤트에 의미있고 설명적인 이름을 제공하도록 합니다. 다른 컴포넌트의 비슷한 속성들의 이름과 일관성을 유지하도록 합니다.

"On"으로 시작하는 이벤트 이름

C++Builder에 있는 대부분의 이벤트 이름은 "On"으로 시작합니다. 이것은 단지 규칙일 뿐이며, 컴파일러에서 이것을 요구하는 것은 아닙니다. Object Inspector는 속성 타입을 보고 속성이 이벤트인지 알아냅니다. 모든 클로저(closure) 속성은 이벤트로 간주되어 Events 페이지에 나타납니다.

개발자는 "On"으로 시작하는 이름의 알파벳 순서 리스트에서 이벤트를 찾고자 합니다. 다른 종류의 이름을 사용하면 혼동을 일으킬 수 있습니다.

참고 이 규칙에 대한 전형적인 예외로 특정 작업의 전후에 발생하는 많은 이벤트를 들 수 있는데, 이러한 이벤트의 이름은 "Before" 및 "After"로 시작합니다.

이벤트 호출

이벤트에 대한 호출을 중앙 집중화해야 합니다. 즉, 애플리케이션의 이벤트 핸들러(지정된 경우)를 호출하고 디폴트 처리를 수행하는 가상 메소드를 컴포넌트 내에 만들어야 합니다.

모든 이벤트 호출을 한 곳에 두면 사용자의 컴포넌트에서 새 컴포넌트를 파생시키는 누군가가 이벤트가 호출되는 장소를 찾기 위해 코드를 검색하지 않고도, 하나의 메소드를 오버라이드함으로써 이벤트 처리를 사용자 정의할 수 있습니다.

이벤트를 호출할 때 더 고려해야 할 두 가지 사항은 다음과 같습니다.

- 비어 있는 핸들러도 유효해야 함
- 사용자가 디폴트 처리를 오버라이드할 수 있음

비어 있는 핸들러도 유효해야 함

이벤트 핸들러가 비어 있다는 이유로 오류를 발생하는 상황을 만들어서는 안되며, 애플리케이션의 이벤트 처리 코드로부터 특정한 응답을 받은 경우에만 컴포넌트가 제대로 작동하게 해서도 안됩니다.

비어 있는 핸들러는 핸들러가 전혀 없는 것과 동일한 결과를 만들어야 합니다. 따라서 애플리케이션의 이벤트 핸들러를 호출하기 위한 코드는 다음과 같아야 합니다.

```
if (OnClick)
  OnClick(this);
// perform default handling }
```

다음과 같이 코드를 작성해서는 *안됩니다*.

```
if (OnClick)
  OnClick(this);
else
  // perform default handling
```

사용자가 디폴트 처리를 오버라이드할 수 있음

어떤 종류의 이벤트에 대해서는 개발자가 디폴트 처리를 대체하거나 모든 응답을 막습니다. 이렇게 할 수 있으려면 핸들러가 반환할 때 핸들러에 참조에 의한 인수를 전달하고 특정 값을 확인해야 합니다.

이는 비어 있는 핸들러가 핸들러가 없는 경우와 동일한 결과를 내야 한다는 규칙을 지키려는 것입니다. 비어 있는 핸들러는 참조에 의해 전달된 인수 값을 변경하지 않기 때문에 비어 있는 핸들러를 호출한 후에는 항상 디폴트 처리가 수행됩니다.

예를 들어 키 누름(key-press) 이벤트를 처리할 때 *Key* 매개변수를 Null 문자로 설정하여 키스트로크에 대한 컴포넌트의 디폴트 처리를 무시할 수 있습니다. 이를 지원하는 로직은 다음과 같습니다.

```
if (OnKeyPress)
  OnKeyPress(this, &Key);
if (Key != NULL)
  //perform default handling
```

실제 코드는 Windows 메시지를 다루기 때문에 위와 약간 다르지만 로직은 같습니다. 디폴트로, 컴포넌트는 사용자가 지정한 핸들러를 호출한 다음 디폴트 처리를 수행합니다. 사용자 핸들러가 *Key*를 Null 문자로 설정하면 컴포넌트는 디폴트 처리를 건너뜁니다.

장

49

메소드 생성

컴포넌트 메소드는 다른 클래스의 메소드와 다르지 않습니다. 즉, 컴포넌트 메소드는 컴포넌트 클래스 구조에 내장된 멤버 함수입니다. 본질적으로 컴포넌트의 메소드로 할 수 있는 작업에 제한은 없지만 C++Builder에서 따라야 하는 몇 가지 표준이 있습니다. 여기에는 다음 사항이 포함됩니다.

- 종속성 피하기
- 메소드 이름 지정
- 메소드 보호
- 가상 메소드 만들기
- 메소드 선언

일반적으로 컴포넌트는 많은 메소드를 포함하지 말아야 하고, 애플리케이션에서 호출해야 할 메소드 수를 최소화해야 합니다. 기능을 메소드로 구현하는 대신 속성으로 캡슐화하는 것이 더 적합한 경우가 많습니다. 속성은 C++Builder 환경에 적합하고 디자인 타임에 액세스할 수 있는 인터페이스를 제공합니다.

종속성 피하기

컴포넌트를 작성할 때는 개발자가 따라야 하는 전제 조건을 최소화합니다. 개발자는 가능한 최대한의 범위까지 컴포넌트에 수행하고 싶은 작업을 언제라도 수행할 수 있어야 합니다. 불가능할 경우도 있겠지만 가능한 한 이 목표를 달성해야 합니다.

다음 리스트는 피해야 할 종속성의 종류입니다.

- 컴포넌트를 사용하기 위해 사용자가 *반드시* 호출해야 하는 메소드
- 특정 순서로 실행해야 할 메소드
- 컴포넌트를 특정 이벤트나 메소드를 사용할 수 없는 상태나 모드로 가져가는 메소드

이런 상황을 처리할 수 있는 가장 좋은 방법은 이를 해결할 수 있는 방법을 제공하는 것입니다. 예를 들어, 특정 메소드의 호출로 인해 컴포넌트에서 다른 메소드를 호출할 수 없는 상태가 될 경우, 컴포넌트가 오류 상태일 때 애플리케이션이 메소드를 호출하면 해당 메인 코드를 실행하기 전에 상태를 수정하도록 두 번째 메소드를 작성하면 됩니다. 최소한 사용자가 잘못된 메소드를 호출한 경우 예외를 발생시켜야 합니다.

즉 코드 일부가 서로에게 종속되는 상황을 만든 경우 잘못된 방법으로 코드를 사용해도 문제가 생기지 않도록 할 책임이 *개발자*에게 있습니다. 예를 들어, 사용자가 종속성을 수용하지 않을 경우 시스템 오류에 대해서는 경고 메시지를 사용하는 것이 좋습니다.

메소드 이름 지정

C++Builder에서는 메소드나 매개변수의 이름을 지정하는데 아무런 제한이 없습니다. 그러나 애플리케이션 개발자를 위해 메소드를 더 쉽게 만들 수 있는 몇 가지 규칙이 있습니다. 컴포넌트 아키텍처의 특성으로 인해 다양한 사람들이 컴포넌트를 사용할 수 있습니다.

자기 자신이나 소수의 프로그래머만을 대상으로 하는 코드 작성에 익숙해 있다면 이름을 지정하는 방법에 대해 신중하게 생각하지 않을 수 있습니다. 메소드 이름을 명확하게 지정하는 것이 좋습니다. 코드나 코드 작성에 익숙하지 않은 사람들도 개발자의 컴포넌트를 사용해야 할 수 있기 때문입니다.

다음은 메소드의 이름을 명확하게 지정하기 위한 몇 가지 제안입니다.

- 설명이 포함된 이름을 사용합니다. 의미가 있는 동사를 사용합니다.

  *PasteFromClipboard*와 같은 이름은 *Paste* 또는 *PFC*보다 더 많은 정보를 알려 줍니다.

- 함수 이름은 반환하는 값의 성격을 반영해야 합니다.

  프로그래머들은 *X*라는 함수가 어떤 것의 수평 좌표를 반환한다는 것을 분명히 알 수 있지만, *GetHorizontalPosition*이라는 이름이 보다 보편적으로 이해하기 쉽습니다.

- 함수 반환 타입이 void일 경우 함수 이름이 구체적이어야 합니다.

  함수 이름에 구체적인 동사를 사용합니다. 예를 들어, *ReadFileNames*가 *DoFiles*보다 훨씬 더 유용합니다.

마지막으로 고려할 것은 메소드가 실제로 메소드여야 하는지 확인하는 것입니다. 이를 확인하는 좋은 방법은 메소드 이름에는 동사가 포함된다는 것입니다. 이름에 동사가 없는 메소드가 많이 만들어졌다면 메소드 대신 속성으로 만들어도 가능한지 여부를 고려합니다.

메소드 보호

데이터 멤버, 메소드 및 속성을 포함하여 클래스의 모든 부분에는 46-4페이지의 "액세스 제어"에서 설명한 대로 보호 레벨이나 "가시성"이 있습니다. 메소드에 대한 적절한 가시성을 선택하는 것은 간단합니다.

컴포넌트에서 작성하는 대부분의 메소드는 **public** 또는 **protected** 메소드입니다. 파생된 컴포넌트조차 그 메소드에 액세스해서는 안될 정도로 엄격하게 해당 컴포넌트 타입에만 한정된 것이 아니라면 메소드를 **private**로 만들 필요는 거의 없습니다.

참고 대체로 이벤트 핸들러 외의 메소드를 **__published**로 선언할 이유가 없습니다. 그렇게 하면 엔드 유저에게는 메소드가 **public**인 것처럼 보입니다.

public이어야 하는 메소드

애플리케이션 개발자가 호출해야 할 메소드는 **public**으로 선언해야 합니다. 대부분의 메소드 호출은 이벤트 핸들러에서 발생하므로 메소드가 시스템 리소스를 모두 사용하거나 운영 체제를 사용자에게 응답할 수 없는 상태로 만들지 말아야 합니다.

참고 생성자와 소멸자는 항상 **public**이어야 합니다.

protected여야 하는 메소드

컴포넌트의 모든 구현 메소드는 애플리케이션이 잘못된 시점에 해당 메소드를 호출할 수 없도록 **protected**여야 합니다. 애플리케이션 코드에서는 호출하지 말아야 하지만 파생된 클래스에서는 호출하는 메소드가 있을 경우 **protected**로 선언합니다.

예를 들어, 특정 데이터가 미리 설정되어 있는지에 의존하는 메소드가 있다고 가정합니다. 이 메소드를 **public**으로 만들면 데이터를 설정하기 전에 애플리케이션에서 이 메소드를 호출할 수 있습니다. 반대로 **protected**로 만들면 애플리케이션에서 이 메소드를 직접 호출할 수 없습니다. 그런 다음 **protected** 메소드를 호출하기 전에 데이터를 설정하는 다른 **public** 메소드를 설정할 수 있습니다.

속성 구현 메소드는 가상 **protected** 메소드로 선언해야 합니다. 가상 메소드로 선언한 메소드를 사용하면 애플리케이션 개발자는 속성 구현을 오버라이드하여 메소드의 기능을 증대시키거나 완전히 대체할 수 있습니다. 이런 속성은 완전히 다형적입니다. 액세스 메소드를 **protected**로 유지하면 개발자가 실수로 메소드를 호출하거나 부주의하게 속성을 수정하지 않도록 할 수 있습니다.

가상 메소드 만들기

동일한 메소드 호출에 대한 응답으로 타입이 다르면 실행되는 코드도 다르게 하려는 경우에는 메소드를 **가상**으로 만듭니다.

애플리케이션 개발자가 직접 사용할 컴포넌트를 만들 경우 모든 메소드를 비가상으로 만들 수 있습니다. 한편 다른 컴포넌트를 파생시키는 추상 컴포넌트를 만들 경우 추가된 메소드를 **가상**으로 만드십시오. 이런 식으로 파생된 컴포넌트는 상속받은 **가상** 메소드를 오버라이드할 수 있습니다.

메소드 선언

컴포넌트에서 메소드를 선언하는 것은 클래스 메소드를 선언하는 것과 동일합니다.

컴포넌트에서 새 메소드를 선언하려면 다음 작업을 수행합니다.

- 컴포넌트의 헤더 파일에 있는 컴포넌트의 클래스 선언에 선언을 추가합니다.
- 유닛의 .CPP 파일에 메소드를 구현하는 코드를 작성합니다.

다음 코드는 두 개의 새 메소드 즉, **protected** 메소드와 **public** 가상 메소드를 정의하는 컴포넌트를 보여 줍니다. 다음은 .H 파일의 인터페이스 정의입니다.

```
class PACKAGE TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
  f
};
```

다음은 메소드를 구현하는 유닛의 .CPP 파일에 있는 코드입니다.

```
void __fastcall TSampleComponent::MakeBigger()
{
  Height = Height + 5;
  Width = Width + 5;
}
int __fastcall TSampleComponent::CalculateArea()
{
  return Width * Height;
}
```

장

50

컴포넌트에서 그래픽 사용

Windows에서는 장치에 독립적인 그래픽을 그릴 수 있는 강력한 그래픽 장치 인터페이스(GDI)를 제공합니다. 그러나 GDI는 그래픽 리소스 관리와 같은 또 다른 요구 사항을 프로그래머에게 부담시킵니다. C++Builder에서는 단조롭고 힘든 GDI 작업을 모두 처리해 주기 때문에 손실된 핸들이나 해제되지 않은 리소스를 찾는 대신 생산적인 작업에 집중할 수 있습니다.

Windows API의 다른 부분을 사용할 때처럼 C++Builder 애플리케이션에서 직접 GDI 함수를 호출할 수 있습니다. 그러나 C++Builder의 그래픽 함수 캡슐화를 사용하는 것이 더 빠르고 쉽다는 것을 알 수 있습니다.

CLX GDI 함수는 Windows에 한정된 것이므로 CLX나 크로스 플랫폼 애플리케이션에는 적용되지 않습니다. 대신 CLX 컴포넌트에서는 Qt 라이브러리를 사용합니다.

이 단원에서는 다음 사항을 설명합니다.

- 그래픽 개요
- 캔버스 사용
- 그림 작업
- 오프스크린 비트맵
- 변경에 대한 응답

그래픽 개요

C++Builder에서는 Windows GDI(CLX의 Qt)를 여러 레벨에서 캡슐화합니다. 컴포넌트 작성자에게 가장 중요한 사항은 컴포넌트가 화면에 이미지를 표시하는 방법입니다. 직접 GDI 함수를 호출하는 경우 펜, 브러시 및 글꼴과 같은 다양한 드로잉 툴을 선택해 둔 장치 컨텍스트에 대한 핸들이 필요합니다. 그래픽 이미지를 렌더링한 후에는 장치 컨텍스트를 폐기하기 전에 원래 상태로 복구해야 합니다.

상세한 레벨에서 그래픽을 작업해야 하는 대신 C++Builder에서는 간단하고 완벽한 인터페이스, 즉 컴포넌트의 *Canvas* 속성을 제공합니다. 캔버스는 유효한 장치 컨텍스트를 사용하도록 하고 이를 사용하지 않을 경우에는 컨텍스트를 해제합니다. 마찬가지로 캔버스에는 현재의 펜, 브러시 및 글꼴을 나타내는 고유한 속성이 있습니다.

캔버스에서 이러한 모든 리소스를 대신 관리해주기 때문에 펜 핸들 등을 만들고, 선택하고, 릴리스하는 것에 신경쓸 필요가 없습니다. 캔버스에게 사용할 펜 종류를 알려주면 캔버스에서 나머지를 처리합니다.

C++Builder에서 그래픽 리소스를 관리하게 하는 장점 중 하나는 리소스를 캐싱하여 나중에 사용할 수 있다는 점입니다. 그러면 반복되는 작업의 속도를 빠르게 할 수 있습니다. 예를 들어, 프로그램에서 특정한 종류의 펜 도구를 반복해서 만들고, 사용하고, 해제할 경우 펜을 사용할 때마다 해당 단계를 반복해야 합니다. C++Builder에서 그래픽 리소스를 캐싱하기 때문에 반복해서 사용하는 도구가 캐시에 계속 저장되어 있으므로 C++Builder 에서 도구를 다시 만들 필요 없이 기존 도구를 사용합니다.

수십 개의 폼이 열려 있고 수백 개의 컨트롤이 있는 애플리케이션이 바로 이런 예입니다. 각 컨트롤마다 하나 이상의 *TFont* 속성이 있을 수 있습니다. 그 결과 수백 또는 수천 개의 *TFont* 객체 인스턴스가 생길 수 있지만, 대부분의 애플리케이션에서는 글꼴 캐시 덕분에 두세 개의 글꼴 핸들만 사용합니다.

다음은 C++Builder의 그래픽 코드가 얼마나 간단해질 수 있는지 보여주는 두 가지 예제입니다. 첫 번째 예제에서는 표준 GDI 함수를 사용하여 다른 개발 도구를 사용할 때와 같은 방법으로 윈도우에 파란색 윤곽의 노란색 타원을 그립니다. 두 번째 예제에서는 캔버스를 사용하여 C++Builder로 작성된 애플리케이션에서 같은 타원을 그립니다.

다음은 ObjectWindows 코드입니다.

```
void TMyWindow::Paint(TDC& PaintDC, bool erase, TRect& rect)
{
  HPEN PenHandle, OldPenHandle;
  HBRUSH BrushHandle, OldBrushHandle;
  PenHandle = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
  OldPenHandle = SelectObject(PaintDC, PenHandle);
  BrushHandle = CreateSolidBrush(RGB(255, 255, 0));
  OldBrushHandle = SelectObject(PaintDC, BrushHandle);
  Ellipse(10, 20, 50, 50);
  SelectObject(OldBrushHandle);
  DeleteObject(BrushHandle);
  SelectObject(OldPenHandle);
  DeleteObject(PenHandle);
)
```

다음 C++Builder 코드는 같은 작업을 수행합니다.

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Pen->Color = clBlue;
  Canvas->Brush->Color = clYellow;
  Canvas->Ellipse(10, 20, 50, 50);
}
```

캔버스 사용

캔버스 클래스는 개별 선, 도형 및 텍스트를 그리기 위한 고수준 함수 및 캔버스의 그리기 기능을 조작하기 위한 중간 속성을 포함하여 여러 레벨에서 그래픽 컨트롤을 캡슐화합니다. 그리고 VCL에서는 Windows GDI에 대한 저수준 액세스를 제공합니다.

표 50.1에는 캔버스의 기능이 요약되어 있습니다.

표 50.1 캔버스 기능 요약

레벨	작업	도구
높음	선과 도형 그리기	*MoveTo*, *LineTo*, *Rectangle* 및 *Ellipse* 같은 메소드
	텍스트 표시 및 측정	*TextOut*, *TextHeight*, *TextWidth* 및 *TextRect* 메소드
	영역 채우기	*FillRect* 및 *FloodFill* 메소드
중간	텍스트와 그래픽 사용자 정의	*Pen*, *Brush* 및 *Font* 속성
	픽셀 처리	*Pixels* 속성
	이미지 복사 및 병합	*Draw*, *StretchDraw*, *BrushCopy* 및 *CopyRect* 메소드, *CopyMode* 속성
낮음	Windows GDI 함수 호출	*Handle* 속성

캔버스 클래스 및 해당 메소드와 속성에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

그림 작업

C++Builder 에서 사용자가 수행하는 대부분의 그래픽 작업은 컴포넌트와 폼의 캔버스에서 직접 그리는 것으로 제한됩니다. C++Builder에서는 팔레트의 자동 관리를 포함하여 비트맵, 메타파일 및 아이콘 같은 독립적인 그래픽 이미지 처리도 제공합니다.

C++Builder에서 그림 작업을 할 때는 다음 세 가지 사항이 중요합니다.

- 그림, 그래픽 또는 캔버스 사용
- 그래픽 로드 및 저장
- 팔레트 처리

그림, 그래픽 또는 캔버스 사용

C++Builder에는 그래픽을 다루는 세 가지 종류의 클래스가 있습니다.

- *캔버스*는 폼, 그래픽 컨트롤, 프린터 또는 비트맵에서 비트맵 형식의 그리기 화면을 나타냅니다. 캔버스는 항상 어떤 것의 속성으로 사용되며 독립적인 클래스로 사용되지 않습니다.
- *그래픽*은 비트맵, 아이콘 또는 메타파일 같이 대개 파일이나 리소스에 있는 종류의 그래픽 이미지를 나타냅니다. C++Builder에서는 *TBitmap*, *TIcon* 및 *TMetafile*(VCL 전용) 클래스를 정의하는데, 이 클래스는 모두 일반적인 *TGraphic*의 자손입니다. 고유한 그래픽 클래스를 정의할 수도 있습니다. *TGraphic*은 모든 그래픽에 대해 최소한의 표준 인터페이스를 정의함으로써 애플리케이션에서 다른 종류의 그래픽을 쉽게 사용할 수 있는 간단한 메커니즘을 제공합니다.

- *그림*은 그래픽의 컨테이너입니다. 이것은 모든 그래픽 클래스를 포함할 수 있다는 것을 의미합니다. 즉, *TPicture* 타입의 항목에는 비트맵, 아이콘, 메타파일 또는 사용자 정의 그래픽 타입이 포함될 수 있고, 애플리케이션에서는 그림 클래스를 통해 같은 방법으로 모두를 액세스할 수 있습니다. 예를 들어, 이미지 컨트롤에는*Picture*라고 하는 *TPicture* 타입의 속성이 있어서 컨트롤에 여러 종류의 그래픽 이미지를 표시할 수 있습니다.

그림 클래스에는 항상 그래픽이 있고 그래픽에는 캔버스가 있을 수 있다는 것을 명심하십시오. 캔버스가 있는 유일한 표준 그래픽은 *TBitmap*입니다. 대개 그림을 작업할 때 *TPicture*를 통해 노출된 그래픽 클래스의 일부만 사용합니다. 그래픽 클래스의 특성을 액세스해야 할 경우 그림의 *Graphic* 속성을 참조할 수 있습니다.

그래픽 로드 및 저장

C++Builder의 모든 그림과 그래픽은 파일에서 이미지를 로드하고 다시 저장하거나 다른 파일로 저장할 수 있습니다. 언제라도 그림의 이미지를 로드하거나 저장할 수 있습니다.

CLX CLX 컴포넌트를 만들 경우Qt MIME 소스나 스트림 객체에서 이미지를 로드하고 저장할 수도 있습니다.

이미지를 파일에서 그림으로 로드하려면 그림의 *LoadFromFile* 메소드를 호출합니다. 그림의 이미지를 파일에 저장하려면 그림의 *SaveToFile* 메소드를 호출합니다.

LoadFromFile 및 *SaveToFile*은 파일 이름을 유일한 매개변수로 취합니다. *LoadFromFile*은 파일 이름의 확장자를 사용하여 만들거나 로드하는 그래픽 객체의 종류를 결정합니다. *SaveToFile*은 저장되는 그래픽 객체 타입에 적합한 파일 타입을 저장합니다.

예를 들어, 비트맵을 이미지 컨트롤의 그림으로 로드하려면 비트맵 파일의 이름을 그림의 *LoadFromFile* 메소드에 전달합니다.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Image1->Picture->LoadFromFile("c:\\windows\\athena.bmp");
}
```

그림은 .bmp를 비트맵 파일의 표준 확장자로 인식하기 때문에 그래픽을 *TBitmap*으로 만든 다음 해당 그래픽의 *LoadFromFile* 메소드를 호출합니다. 그래픽이 비트맵이기 때문에 파일에서 이미지를 비트맵으로 로드합니다.

팔레트 처리

VCL 및 CLX 컴포넌트의 경우 팔레트 기반 장치(대개 256 색상 비디오 모드)에서 실행하면 C++Builder 컨트롤에서 자동으로 팔레트 실현을 지원합니다. 즉, 팔레트를 갖고 있는 컨트롤이 있으면 *TControl*에서 상속받은 두 개의 메소드를 사용하여 Windows에서 팔레트를 처리하는 방법을 제어할 수 있습니다.

컨트롤에 대한 팔레트 지원에는 다음 두 가지 사항이 있습니다.

- 컨트롤의 팔레트 지정
- 팔레트 변경 사항에 응답

대부분의 컨트롤에는 팔레트가 필요하지 않지만 이미지 컨트롤 같이 "다양한 색상"의 그래픽 이미지를 포함하고 있는 컨트롤은 Windows 및 화면 장치 드라이버와 상호 작용하여 컨트롤이 제대로 표시되도록 해야 합니다. Windows에서는 이러한 과정을 팔레트 *실현*이라고 합니다.

팔레트 실현은 맨 앞의 윈도우에서 전체 팔레트를 사용하고 백그라운드 윈도우는 가능한 한 많은 팔레트를 사용하도록 한 다음 다른 모든 색상을 "실제" 팔레트에서 사용할 수 있는 가장 가까운 색상으로 매핑하는 과정입니다. 윈도우가 다른 윈도우 앞으로 이동할 때마다 Windows에서는 계속 팔레트를 실현하게 됩니다.

참고 C++Builder는 비트맵 이외에서는 팔레트 생성이나 유지 보수를 위한 특별한 지원을 제공하지 않습니다. 그러나 팔레트 핸들이 있으면 C++Builder 컨트롤에서 대신 관리할 수 있습니다.

컨트롤의 팔레트 지정

VCL 및 CLX 컨트롤을 위한 팔레트를 지정하려면 컨트롤의 *GetPalette* 메소드를 오버라이드하여 팔레트의 핸들을 반환합니다.

컨트롤에 팔레트를 지정하면 애플리케이션에 대해 다음 작업이 수행됩니다.

- 애플리케이션에게 컨트롤의 팔레트를 실현해야 함을 알려줍니다.
- 실현을 위해 사용할 팔레트를 지정합니다.

팔레트 변경 사항에 응답

VCL 및 CLX 컨트롤에서 *GetPalette* 메소드를 오버라이드하여 팔레트를 지정하면 C++Builder에서 자동으로 Windows의 팔레트 메시지에 응답합니다. 팔레트 메시지를 처리하는 메소드는 *PaletteChanged*입니다.

*PaletteChanged*의 기본적인 역할은 컨트롤의 팔레트를 전경 또는 배경에서 실현할 것인지 결정하는 것입니다. Windows에서는 가장 위쪽에 있는 윈도우가 전경 팔레트를 갖고 다른 윈도우는 백그라운드 팔레트를 갖도록 하여 팔레트를 실현합니다. C++Builder는 한 윈도우에서 탭 순서대로 컨트롤에 대한 팔레트를 실현하는 한 가지 단계를 더 처리합니다. 탭 순서에서 첫 번째가 아닌 컨트롤이 전경 팔레트를 갖게 할 경우에는 이 디폴트 동작을 오버라이드해야 합니다.

오프스크린 비트맵

복잡한 그래픽 이미지를 그릴 경우 그래픽 프로그래밍의 일반적인 기술은 오프스크린 비트맵을 만들고, 비트맵에 이미지를 그린 다음 비트맵의 전체 이미지를 최종 대상 화면으로 복사하는 것입니다. 오프스크린 이미지를 사용하면 화면에 직접 반복해서 그릴 경우 발생하는 화면 떨림을 줄일 수 있습니다.

리소스와 파일의 비트맵 형식의 이미지를 나타내는 C++Builder의 비트맵 클래스는 오프스크린 이미지로 사용할 수도 있습니다.

오프스크린 비트맵을 사용하는 데에는 두 가지 용도가 있습니다.

- 오프스크린 비트맵 생성 및 관리
- 비트맵 이미지 복사

오프스크린 비트맵 생성 및 관리

복잡한 그래픽 이미지를 만들 경우 화면에 나타나는 캔버스에 직접 그리지 마십시오. 폼이나 컨트롤의 캔버스에 그리는 대신 비트맵 객체를 만들고, 해당 캔버스에 그린 다음 전체 이미지를 화면상의 캔버스로 복사할 수 있습니다. 오프스크린 비트맵을 가장 일반적으로 사용하는 경우는 그래픽 컨트롤의 *Paint* 메소드입니다.

오프스크린 비트맵에 복잡한 이미지를 그리는 예제를 보려면 컴포넌트 팔레트의 Samples 페이지에서 Gauge 컨트롤의 소스 코드를 참조하십시오. Gauge 컨트롤에서는 화면으로 복사하기 전에 오프스크린 비트맵에 다른 도형과 텍스트를 그립니다. Gauge 컨트롤의 소스 코드는 Examples\Controls\Source 하위 디렉토리의 Cgauges.cpp 파일에 있습니다.

비트맵 이미지 복사

C++Builder에서는 캔버스 간에 이미지를 복사할 수 있는 네 가지 다른 방법을 제공합니다. 만들려고 하는 효과에 따라 다른 메소드를 호출합니다.

표 50.2에는 캔버스 객체의 이미지 복사 메소드가 요약되어 있습니다.

표 50.2 이미지 복사 메소드

효과	호출하는 메소드
전체 그래픽 복사	Draw
그래픽 복사 및 크기 조정	StretchDraw
캔버스 일부분 복사	CopyRect
래스터 작업으로 비트맵 복사	BrushCopy(VCL)
그래픽을 반복적으로 복사해서 영역에 바둑판식으로 배열	TiledDraw(CLX)

변경에 대한 응답

캔버스와 캔버스에서 소유한 객체(펜, 브러시 및 글꼴)를 포함하여 모든 그래픽 객체에는 객체의 변경에 응답하기 위해 만든 이벤트가 있습니다. 이 이벤트를 사용하면 이미지를 다시 그려서 컴포넌트나 컴포넌트를 사용하는 애플리케이션에서 변경에 응답하게 할 수 있습니다.

컴포넌트의 디자인 타임 인터페이스 일부로 그래픽 객체를 게시한 경우에 그래픽 객체의 변화에 대한 응답이 특히 중요합니다. 컴포넌트의 디자인 타임 모양이 Object Inspector에서 설정한 속성과 일치하게 할 수 있는 유일한 방법은 객체의 변화에 반응하는 것입니다.

그래픽 객체의 변화에 반응하려면 클래스의 *OnChange* 이벤트에 메소드를 할당합니다.

도형 컴포넌트에서는 모양을 그리기 위해 사용하는 펜과 브러시를 나타내는 속성을 게시합니다. 컴포넌트의 생성자는 각각의 *OnChange* 이벤트에 메소드를 할당하여 펜이나 브러시가 바뀔 경우 컴포넌트에서 이미지를 새로 고치도록 합니다. 도형 컴포넌트는 오브젝트 파스칼로 작성되어 있지만 다음은 도형 컴포넌트를 *TMyShape*라는 새로운 이름을 사용하여 C++로 작성한 것입니다.

다음은 헤더 파일의 클래스 선언입니다.

```
class PACKAGE TMyShape : public TGraphicControl
{
private:
protected:
public:
    virtual __fastcall TMyShape(TComponent* Owner);
__published:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall StyleChanged(TObject *Sender);
};
```

다음은 .CPP 파일의 코드입니다.

```
__fastcall TMyShape::TMyShape(TComponent* Owner)
  : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FPen = new TPen;
  FPen->OnChange = StyleChanged;
  FBrush = new TBrush;
  FBrush->OnChange = StyleChanged;
}
void __fastcall TMyShape::StyleChanged(TObject *Sender)
{
  Invalidate();
}
```

장

51

메시지 및 시스템 통지 처리

컴포넌트는 기초가 되는 운영 체체에서 보내는 통지에 응답해야 하는 경우가 종종 있습니다. 운영 체제는 사용자가 마우스나 키보드로 수행한 작업과 같은 사건을 애플리케이션에게 알려 줍니다. 일부 컨트롤에서는 리스트 박스에서 항목을 선택하는 등의 사용자 작업으로 인한 결과와 같은 통지를 생성하기도 합니다. VCL 및 CLX에서 일반적인 통지를 대부분 처리합니다. 그러나 이러한 통지를 처리하기 위한 고유한 코드를 작성해야 할 수도 있습니다.

VCL에서는 통지가 *메시지* 형태로 도착합니다. 이 메시지는 Windows, VCL 컴포넌트 및 개발자가 정의한 컴포넌트를 포함한 모든 소스에서 올 수 있습니다. 메시지 작업과 관련된 다음과 같은 세 가지 사항이 있습니다.

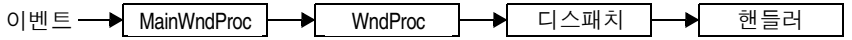
- 메시지 처리 시스템 이해
- 메시지 처리 변경
- 새 메시지 핸들러 생성

CLX에서는 통지가 Windows 메시지 대신 *신호*와 *시스템 이벤트* 형태로 도착합니다. CLX에서의 시스템 통지 작업 방법에 대한 자세한 내용은 51-10페이지의 "CLX를 사용하여 시스템 통지에 응답"을 참조하십시오.

메시지 처리 시스템 이해

모든 VCL 클래스에는 *메시지 처리 메소드* 또는 *메시지 핸들러*라고 하는, 메시지 처리를 위한 기본 메커니즘이 있습니다. 메시지 핸들러의 기본 개념은 클래스에서 특정 종류의 메시지를 받아 디스패치하고, 받은 메시지에 따라 지정한 메소드 집합 중 하나를 호출하는 것입니다. 특정 메시지에 대한 지정된 메소드가 없으면 디폴트 핸들러가 제공됩니다.

다음 다이어그램에서는 메시지 디스패치 시스템을 보여 줍니다.



비주얼 컴포넌트 라이브러리에서는 특정 클래스에 전달된 사용자 정의 메시지를 포함한 모든 Windows 메시지를 메소드 호출로 변환하는 메시지 디스패칭 시스템을 정의합니다. 이 메시지 디스패치 메커니즘은 변경할 필요가 없습니다. 메시지 처리 메소드를 만들기만 하면 됩니다. 이에 대한 자세한 내용은 51-7페이지의 "새 메시지 처리 메소드 선언"을 참조하십시오.

Windows 메시지의 내용

Windows 메시지는 유용한 여러 데이터 멤버가 포함된 데이터 구조라고 할 수 있습니다. 이러한 데이터 멤버 중에서 가장 중요한 것은 메시지를 식별하는 정수 크기의 값입니다. Windows에서 많은 메시지를 정의하고 MESSAGES.HPP 파일에서는 각 메시지에 대한 식별자를 선언합니다.

Windows 프로그래머는 *WM_COMMAND* 또는 *WM_PAINT*같이 메시지를 식별하는 Windows 정의를 사용하는 데 익숙합니다. 일반적인 Windows 프로그램에는 시스템에서 생성된 메시지의 콜백 역할을 하는 윈도우 프로시저가 있습니다. 이 윈도우 프로시저에는 대개 이 윈도우에서 처리할 각각의 메시지에 대한 case 레이블이 있는 큰 switch 문이 있습니다.

그 밖의 유용한 정보는 *word parameter*와 *long parameter*에 대한 각각의 두 매개변수인 *wParam*과 *lParam*을 통해 윈도우 프로시저에 전달됩니다. 때로 각 매개변수에 둘 이상의 정보가 포함될 수 있습니다. 그럴 경우 *LOWORD* 및 *HIWORD* 같은 Windows 매크로를 사용하여 해당하는 부분을 끌어내야 합니다. 예를 들어, HIWORD(lParam)를 호출하면 이 매개변수의 상위 워드를 가져옵니다.

원래 Windows 프로그래머는 각 매개변수에 포함된 내용을 기억하거나 Windows API로 찾아보아야 했습니다. 그러나 이제 Windows에서는 *메시지 크래커*를 사용하여 Windows 메시지 및 연관된 매개변수 처리와 관련된 구문을 단순화시켰습니다. 메시지 크래커를 사용하면 긴 switch 문을 사용하여 모든 정보의 압축을 풀어 매개변수로 나타내는 대신, 핸들러 함수를 메시지와 연관시킬 수 있습니다. WINDOWSX.H를 표준 Windows 프로그램에 포함시키면 해당 프로그램에서 *HANDLE_MSG* 매크로를 사용할 수 있게 되므로 다음과 같은 코드를 작성할 수 있습니다.

```
void MyKeyDownHandler(  HWND hwnd, UINT nVirtKey, BOOL fDown, int
CRepeat, UINT flags )
{
  f
}

LRESULT MyWndProc( HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam
)
{
  switch(  Message )
  {
    HANDLE_MSG( hwnd, WM_KEYDOWN, MyKeyDownHandler );
    f
}
```

이런 스타일의 메시지 크래킹을 사용하면 더 확실하게 메시지를 특정 핸들러에 디스패치할 수 있습니다. 그리고 핸들러 함수의 매개변수 리스트에 의미가 있는 이름을 제공할 수 있습니다. *WM_KEYDOWN* 메시지의 *wParam*에 대한 값인 *nVirtKey*라는 매개변수를 갖고 있는 함수를 이해하기가 더 쉽습니다.

메시지 디스패칭

애플리케이션에서 윈도우를 만들 경우 Windows 커널에 *윈도우 프로시저*를 등록합니다. 윈도우 프로시저는 윈도우의 메시지를 처리하는 루틴입니다. 대개 윈도우 프로시저에는 윈도우에서 처리해야 할 각각의 메시지에 대한 항목이 있는 큰 **switch** 문이 있습니다. 여기서 "윈도우"란 화면 상의 모든 것, 즉 윈도우, 컨트롤 등을 의미합니다. 새로운 타입의 윈도우를 만들 때마다 완전한 윈도우 프로시저를 만들어야 합니다.

VCL에서는 다음과 같은 방법으로 메시지 디스패칭을 단순화합니다.

- 각 컴포넌트에서 완전한 메시지 디스패칭 시스템을 상속받습니다.
- 디스패치 시스템에서 디폴트 처리를 합니다. 특별히 응답해야 하는 메시지에 대해서만 핸들러를 정의합니다.
- 메시지 처리의 사소한 부분은 수정할 수 있고 대부분의 처리에 있어서 상속받은 메소드를 사용할 수 있습니다.

이 메시지 디스패치 시스템의 가장 큰 장점은 언제라도 컴포넌트에 메시지를 안전하게 보낼 수 있다는 점입니다. 컴포넌트에 메시지를 위해 정의한 핸들러가 없으면 대개 메시지를 무시하여 디폴트 처리에서 메시지를 처리합니다.

메시지 흐름 추적

VCL에서는 애플리케이션의 모든 타입의 컴포넌트에 대해 *MainWndProc*라고 하는 메소드를 윈도우 프로시저로 등록합니다. *MainWndProc*에는 Windows의 메시지 구조체를 *WndProc*라고 하는 가상 메소드로 전달하고 애플리케이션 클래스의 *HandleException* 메소드를 호출하여 예외를 처리하는 예외 처리 블록이 포함되어 있습니다.

*MainWndProc*는 특정 메시지에 대한 특수 처리가 포함되지 않은 비가상 메소드입니다. 컴포넌트 타입은 각각의 특정 요구에 맞게 메소드를 오버라이드할 수 있기 때문에 *WndProc*에서 사용자 정의가 수행됩니다.

WndProc 메소드에서는 메소드 처리에 영향을 미치는 특수한 조건을 확인하여 불필요한 메시지를 "트랩"할 수 있습니다. 예를 들어, 컴포넌트를 끌어 놓는 동안에는 컴포넌트에서 키보드 이벤트를 무시하므로 컴포넌트를 끌어 놓지 않는 동안에만 *TWinControl*의 *WndProc* 메소드가 키보드 이벤트를 전달합니다. 결국 *WndProc*에서는 메시지를 처리하기 위해 호출할 메소드를 결정하는 *TObject*에서 상속받은 비가상 메소드인 *Dispatch*를 호출합니다.

*Dispatch*에서는 메시지 구조체의 *Msg* 데이터 멤버를 사용하여 특정 메시지를 디스패치하는 방법을 결정합니다. 컴포넌트에서 특정 메시지의 핸들러를 정의한 경우 *Dispatch*에서 메소드를 호출합니다. 컴포넌트에서 특정 메시지의 핸들러를 정의하지 않은 경우 *Dispatch*에서 *DefaultHandler*를 호출합니다.

메시지 처리 변경

컴포넌트의 메시지 처리를 변경하기 전에 원하는 작업이 정확히 무엇인지를 확인해야 합니다. VCL에서는 대부분의 Windows 메시지를 컴포넌트 작성자와 컴포넌트 사용자가 모두 처리할 수 있는 이벤트로 변환합니다. 메시지 처리 동작을 변경하는 것이 아니라 이벤트 처리 동작을 변경해야 할 수도 있습니다.

VCL 컴포넌트의 메시지 처리를 변경하려면 메시지 처리 메소드를 오버라이드합니다. 메시지를 트랩하여 특정 상황에서 컴포넌트가 메시지를 처리하지 못하게 할 수도 있습니다.

핸들러 메소드 오버라이드

컴포넌트에서 특정 메시지를 처리하는 방법을 변경하려면 해당 메시지의 메시지 처리 메소드를 오버라이드합니다. 컴포넌트에서 특정 메시지를 이전에 처리한 적이 없으면 새로운 메시지 처리 메소드를 선언해야 합니다.

다음과 같은 방법으로 메시지 처리 메소드를 오버라이드합니다.

1 컴포넌트 선언의 protected 부분에서 오버라이드한 메소드와 같은 이름을 가진 컴포넌트에 새 메소드를 선언합니다.

2 다음 세 매크로를 사용하여 오버라이드한 메시지에 메소드를 매핑합니다.

   매크로의 형태는 다음과 같습니다.

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(parameter1, parameter2, parameter3)
END_MESSAGE_MAP
```

   *Parameter1*은 Windows에서 정의한 메시지 인덱스이고, *parameter2*는 메시지 구조체 타입이고, *parameter3*은 메시지 메소드의 이름입니다.

   BEGIN_MESSAGE_MAP 매크로와 *END_MESSAGE_MAP* 매크로 사이에 *MESSAGE_HANDLER* 매크로를 원하는 수만큼 포함시킬 수 있습니다.

예를 들어, 컴포넌트의 *WM_PAINT* 메시지 처리를 오버라이드하려면 *WMPaint* 메소드를 재선언하고 세 매크로를 사용하여 메소드를 *WM_PAINT* 메시지에 매핑합니다.

```
class PACKAGE TMyComponent : public TComponent
{
protected:
    void __fastcall WMPaint(TWMPaint* Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint)
END_MESSAGE_MAP(TComponent)
};
```

메시지 매개변수 사용

메시지 처리 메소드 내에서는 컴포넌트가 메시지 구조체의 모든 매개변수에 액세스할 수 있습니다. 메시지 핸들러에 전달된 매개변수가 포인터이기 때문에 필요하면 핸들러에서 매개변수 값을 변경할 수 있습니다. 자주 바뀌는 매개변수는 메시지의 반환 값, 즉 메시지를 보내는 *SendMessage* 호출에서 반환하는 값뿐입니다.

메시지 처리 메소드의 *Message* 매개변수 타입은 처리되는 메시지에 따라 다르므로 개별 매개변수의 이름과 의미를 Windows 메시지 설명서에서 참조해야 합니다. 어떤 이유 때문에 이전 스타일의 이름(*WParam*, *LParam* 등)으로 메시지 매개변수를 참조해야 할 경우 *Message*를 해당 매개변수 이름을 사용하는 일반적인 타입의 *TMessage*로 타입 변환할 수 있습니다.

메시지 트래핑

컴포넌트에서 메시지를 무시하게 하려는 경우도 있는데, 컴포넌트에서 메시지를 해당 핸들러로 디스패치할 수 없게 하려는 경우가 이에 해당합니다. 메시지를 트랩하려면 가상 메소드 *WndProc*를 오버라이드합니다.

VCL 컴포넌트의 경우 *WndProc* 메소드는 메시지를 *Dispatch* 메소드로 전달하기 전에 메시지를 검사합니다. 그런 다음 Dispatch 메소드에서 메시지를 처리할 메소드를 결정합니다. *WndProc*를 오버라이드하면 컴포넌트에서 메시지를 디스패치하기 전에 필터링할 수 있습니다. *TWinControl*에서 파생된 컨트롤에 대한 *WndProc*의 오버라이드는 다음과 같습니다.

```
void __fastcall TMyControl::WndProc(TMessage& Message)
{
   // tests to determine whether to continue processing
  if(Message.Msg != WM_LBUTTONDOWN)
    TWinControl::WndProc(Message);
}
```

TControl 컴포넌트에서는 사용자가 컨트롤을 끌어 놓을 때 필터링하는 마우스 메시지의 전체 범위를 정의합니다. *WndProc* 오버라이드는 다음 두 가지 측면에서 유용합니다.

- 각각의 메시지에 대한 핸들러를 지정하는 대신 메시지 범위를 필터링할 수 있습니다.
- 메시지 디스패칭을 완전히 제외시킬 수 있으므로 핸들러를 호출하지 않습니다.

다음은 오브젝트 파스칼에서 VCL에 구현된 *TControl*의 *WndProc* 메소드 일부입니다.

```
procedure TControl.WndProc(var Message: TMessage);
begin
  f
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST)
then
    if Dragging then                    { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
     f                                { handle others normally }
    end;
  f                                 { otherwise process normally }
end;
```

새 메시지 핸들러 생성

VCL에서는 대부분의 일반적인 메시지에 대한 핸들러를 제공하기 때문에 새 메시지 핸들러를 만들어야 할 대부분의 경우는 고유한 메시지를 정의할 때입니다. 사용자 정의 메시지를 사용하는 세 가지 방법은 다음과 같습니다.

- 새로운 메시지 정의
- 새 메시지 처리 메소드 선언
- 메시지 보내기

새로운 메시지 정의

여러 표준 컴포넌트에서는 내부에서 사용하기 위한 메시지를 정의합니다. 메시지를 정의하는 가장 일반적인 이유는 표준 메시지와 상태 변경 통지에 포함되지 않은 정보를 브로드캐스트하는 것입니다. VCL에서 고유한 메시지를 정의할 수 있습니다.

메시지 정의에는 다음과 같은 2단계 프로세스가 있습니다.

1 메시지 식별자 선언

2 메시지 구조체 타입 선언

메시지 식별자 선언

메시지 식별자는 정수 크기의 상수입니다. Windows에서 자체 사용을 위해 1,024 이하의 메시지를 예약하므로 고유한 메시지를 선언할 경우 이 레벨 이상에서 시작해야 합니다.

상수 *WM_APP*는 사용자 정의 메시지의 시작 번호를 나타냅니다. 메시지 식별자를 정의할 경우 *WM_APP*를 기준으로 해야 합니다.

일부 표준 Windows 컨트롤에서 사용자 정의한 범위의 메시지를 사용한다는 사실을 명심하십시오. 여기에는 리스트 박스, 콤보 박스, 에디트 박스 및 명령 버튼이 포함됩니다. 이 중 하나에서 컴포넌트를 파생시키고 컴포넌트에 대한 새로운 메시지를 정의할 경우 MESSAGES.HPP 파일을 검사하여 Windows에서 해당 컨트롤에 미리 정의한 메시지를 확인합니다.

다음 코드에서는 두 개의 사용자 정의 메시지를 보여 줍니다.

```
#define MY_MYFIRSTMESSAGE  (WM_APP + 400)
#define MY_MYSECONDMESSAGE (WM_APP + 401)
```

메시지 구조체 타입 선언

메시지의 매개변수에 의미있는 이름을 지정하려는 경우 해당 메시지에 대한 메시지 구조체 타입을 선언해야 합니다. 메시지 구조체는 메시지 처리 메소드에 전달된 매개변수의 타입입니다. 메시지의 매개변수를 사용하지 않거나 이전 스타일의 매개변수 표시법(*wParam*, *lParam* 등)을 사용할 경우 디폴트 메시지 구조체인, *TMessage*를 사용할 수 있습니다.

메시지 구조체 타입을 선언하려면 다음 규칙을 따릅니다.

1 앞에 *T*가 오는 메시지 다음에 구조체 타입을 지정합니다.

2 *TMsgParam* 타입의 *Msg* 구조에서 첫 번째 데이터 멤버를 호출합니다.

3 *Word* 매개변수에 해당하는 다음 두 바이트를 정의하고, 그 다음 두 바이트는 사용하지 않은 바이트로 정의합니다.

또는

Longint 매개변수에 해당하는 다음 네 바이트를 정의합니다.

4 *Longint*타입의 *Result*라는 마지막 데이터 멤버를 추가합니다.

예를 들어, 다음은 모든 마우스 메시지인 *TWMKey*에 대한 메시지 구조체입니다.

```
struct TWMKey
{
  Cardinal Msg;                    // first parameter is the message ID
  Word CharCode;                   // this is the first wParam
  Word Unused;
  Longint KeyData;                 // this is the lParam
  Longint Result;                  // this is the result data member
};
```

새 메시지 처리 메소드 선언

새로운 메시지 처리 메소드를 선언해야하는 두 가지 경우는 다음과 같습니다.

- 표준 컴포넌트에서 처리하지 않은 Windows 메시지를 사용자의 컴포넌트에서 처리해야 합니다.
- 컴포넌트에서 사용하기 위해 고유한 메시지를 정의했습니다.

메시지 처리 메소드를 선언하려면 다음을 수행합니다.

1 BEGIN_MESSAGE_MAP ... END_MESSAGE_MAP 매크로를 사용하여 컴포넌트의 클래스 선언의 **protected** 부분에서 메소드를 선언합니다.

2 메소드에서 void를 반환하는지 확인합니다.

3 처리한 메시지 다음에 메소드를 밑줄을 사용하지 않고 지정합니다.

4 메시지 구조체 타입의 *Message*라는 포인터를 전달합니다.

5 매크로를 사용하여 메시지에 메소드를 매핑합니다.

6 메시지 메소드 구현에서 컴포넌트에 관련된 처리를 위한 코드를 작성합니다.

7 상속받은 메시지 핸들러를 호출합니다.

예를 들어, 다음은 *CM_CHANGECOLOR*라고 하는 사용자 정의 메시지에 대한 메시지 핸들러의 선언입니다.

```
#define CM_CHANGECOLOR (WM_APP + 400)

class TMyControl : public TControl
{
protected:
    void __fastcall CMChangeColor(TMessage &Message);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_CHANGECOLOR, TMessage, CMChangeColor)
END_MESSAGE_MAP(TControl)
};

void __fastcall TMyControl::CMChangeColor(TMessage &Message)
{
  Color = Message.LParam;                    // set color from long
parameter
  TControl::CMChangeColor(Message);          // call the inherited message
handler
}
```

메시지 보내기

대개 애플리케이션에서는 상태 변경 통지를 보내거나 정보를 브로드캐스트하기 위해 메시지를 보냅니다. 컴포넌트는 메시지를 폼에 있는 모든 컨트롤에 브로드캐스트할 수 있고, 메시지를 특정 컨트롤이나 애플리케이션 자체에 보낼 수 있으며, 자신에게도 메시지를 보낼 수 있습니다.

여러 가지 방법을 사용하여 Windows 메시지를 보낼 수 있습니다. 메시지를 보내는 이유에 따라 사용하는 메소드가 다릅니다. 다음 항목에서는 Windows 메시지를 보내는 여러 가지 방법에 대해 설명합니다.

메시지를 폼에 있는 모든 컨트롤에 브로드캐스트

컴포넌트에서 폼이나 다른 컨테이너에 있는 모든 컨트롤에 영향을 미치는 전역 설정을 변경하는 경우, 해당 컨트롤이 자체적으로 업데이트될 수 있도록 컨트롤에게 메시지를 보내야 합니다. 모든 컨트롤이 통지에 응답할 필요는 없지만 메시지를 브로드캐스트할 때, 응답하는 방법을 알고 있는 모든 컨트롤에게는 응답을 하도록 하고, 그 밖의 컨트롤은 메시지를 무시하는 것을 허용할 수 있습니다.

다른 컨트롤의 모든 컨트롤에 메시지를 브로드캐스트하려면 *Broadcast* 메소드를 사용합니다. 메시지를 브로드캐스트하기 전에 메시지 구조체를 전달할 정보로 채웁니다. 메시지 구조체에 대한 자세한 내용은 51-6페이지의 "메시지 구조체 타입 선언"을 참조하십시오.

```
TMessage Msg;
Msg.Msg = MY_MYCUSTOMMESSAGE;
Msg.WParam = 0;
Msg.LParam = (int)(this);
Msg.Result = 0;
```

그런 다음 이 메시지 구조체를 통지할 모든 컨트롤의 부모에게 전달합니다. 애플리케이션의 모든 컨트롤이 가능합니다. 예를 들어, 다음과 같이 작성하고 있는 컨트롤의 부모도 가능합니다.

```
Parent->Broadcast(Msg);
```

다음과 같이 컨트롤을 포함하고 있는 폼이 될 수 있습니다.

```
GetParentForm(this)->Broadcast(Msg);
```

다음과 같이 활성 폼이 될 수 있습니다.

```
Screen->ActiveForm->Broadcast(Msg);
```

다음과 같이 애플리케이션의 모든 폼이 될 수도 있습니다.

```
for (int i = 0; i< Screen->FormCount; i++)
  Screen->Forms[i]->Broadcast(Msg);
```

컨트롤의 메시지 핸들러 직접 호출

때로 메시지에 응답해야 할 컨트롤이 하나만 있는 경우도 있습니다. 메시지를 받아야 할 컨트롤을 알고 있으면 메시지를 보내는 가장 간단한 방법은 해당 컨트롤의 *Perform* 메소드를 호출하는 것입니다.

컨트롤의 *Perform* 메소드를 호출하는 두 가지 중요한 이유는 다음과 같습니다.

- 컨트롤의 똑같은 응답을 표준 Windows 메시지나 다른 메시지로 실행하고자 합니다. 예를 들어, 그리드 컨트롤에서 키스트로크 메시지를 받으면 인라인 편집 컨트롤을 만든 다음 키스트로크 메시지를 편집 컨트롤로 보냅니다.
- 통지할 컨트롤을 알고 있을 수 있지만 컨트롤 타입은 모릅니다. 대상 컨트롤 타입을 모르기 때문에 대상 컨트롤의 특화된 메소드를 알 수 없습니다. 그러나 모든 컨트롤에 메시지 처리 기능이 있기 때문에 항상 메시지를 보낼 수 있습니다. 컨트롤에 보낸 메시지에 대한 메시지 핸들러가 있으면 적절하게 응답합니다. 그렇지 않으면 보낸 메시지를 무시하고 0을 반환합니다.

Perform 메소드를 호출하려면 메시지 구조체를 만들지 않아도 됩니다. 메시지 식별자 WParam 및 LParam을 매개변수로 전달하기만 하면 됩니다. Perform에서 메시지 결과를 반환합니다.

Windows 메시지 대기열을 사용하여 메시지 보내기

멀티 스레드 애플리케이션에서는 *Perform* 메소드를 호출할 수 없습니다. 대상 컨트롤이 실행된 스레드가 아닌 다른 스레드에 있기 때문입니다. 그러나 Windows 메시지 대기열을 사용하면 안전하게 다른 스레드와 통신할 수 있습니다. 메시지 처리는 항상 메인 VCL 스레드에서 발생하지만 Windows 메시지 대기열을 사용하여 애플리케이션의 스레드에서 메시지를 보낼 수 있습니다. *SendMessage* 호출이 동시에 이루어 집니다. 즉, *SendMessage*는 대상 컨트롤이 다른 스레드에 있더라도 대상 컨트롤에서 메시지를 처리할 때까지 반환하지 않습니다.

Windows 메시지 대기열을 사용하여 컨트롤에 메시지를 보내려면 Windows API 호출, *SendMessage*를 사용하십시오. *SendMessage*는 대상 컨트롤의 Windows 핸들을 전달하여 대상 컨트롤을 인식해야 한다는 점만 제외하고 *Perform* 메소드와 같은 매개변수를 갖고 있습니다. 따라서 다음을 작성하는 대신

```
MsgResult = TargetControl->Perform(MY_MYMESSAGE, 0, 0);
```

다음을 작성할 것입니다.

```
MsgResult = SendMessage(TargetControl->Handle, MYMESSAGE, 0, 0);
```

SendMessage 함수에 대한 자세한 내용은 Microsoft MSDN 설명서를 참조하십시오. 동시에 실행할 수 있는 복수 스레드 작성에 대한 자세한 내용은 11-7페이지의 "스레드 조정"을 참조하십시오.

곧바로 실행되지 않는 메시지 보내기

메시지를 보내고 싶지만 메시지의 대상을 바로 실행하기가 안전한지 알지 못할 경우가 있습니다. 예를 들어, 메시지를 보낸 코드를 대상 컨트롤의 이벤트 핸들러에서 호출한 경우 컨트롤에서 메시지를 실행하기 전에 이벤트 핸들러에서 실행을 완료했는지 확인할 수 있습니다. 메시지 결과를 알 필요가 없는 한 이 상황을 처리할 수 있습니다.

Windows API 호출, *PostMessage*를 사용하여 메시지를 컨트롤에 보냅니다. 그러나 해당 메시지를 처리하기 전에 다른 메시지를 완료할 때까지 컨트롤에서 기다리도록 할 수 있습니다. *PostMessage*에는 *SendMessage*와 똑같은 매개변수가 있습니다.

PostMessage 함수에 대한 자세한 내용은 Microsoft MSDN 설명서를 참조하십시오.

CLX를 사용하여 시스템 통지에 응답

Windows를 사용할 경우 운영 체제에서 Windows 메시지를 사용하여 사용자 애플리케이션과 애플리케이션에 포함된 컨트롤에게 직접 통지를 보냅니다. 그러나 이 방법은 CLX에는 적합하지 않습니다. CLX는 크로스 플랫폼 라이브러리이고 Linux에서는 Windows 메시지를 사용하지 않기 때문입니다. 대신 CLX에서는 플랫폼에 중립적인 방법을 사용하여 시스템 통지에 응답합니다.

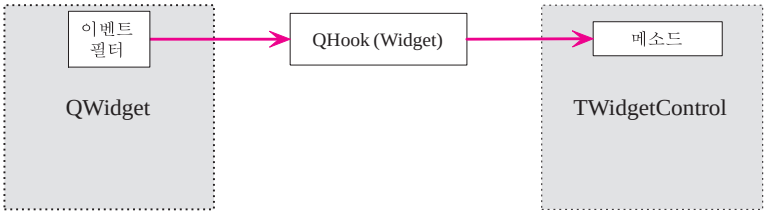
CLX에서는 Windows 메시지에 대한 아날로그가 원본으로 사용하는 widget 레이어의 신호 시스템입니다. VCL에서는 Windows 메시지가 운영 체제에서 발생하거나 VCL에서 랩핑한 원시 Windows 컨트롤에서 발생할 수 있지만, CLX에서 사용하는 widget 레이어는 이 두 개를 구별합니다. Widget에서 통지가 발생한 경우에는 이를 신호라고 합니다. 운영 체제에서 통지가 발생한 경우에는 시스템 이벤트라고 부릅니다. Widget 레이어에서는 시스템 이벤트를 *이벤트* 타입의 신호로 CLX 컴포넌트에 전달합니다.

신호에 응답

원본으로 사용하는 widget 레이어에서는 다양한 신호를 보내고, 각각은 다른 타입의 통지를 나타냅니다. 이런 신호에는 신호를 만든 widget에 관련된 통지 뿐만 아니라 시스템 이벤트(이벤트 신호)도 포함됩니다. 예를 들어, widget이 해제된 경우에는 모든 widget에서 소멸된 신호를 생성하고, 트랙 표시줄 widget에서는 *valueChanged* 신호를 생성하고, 헤더 컨트롤에서는 *sectionClicked* 신호를 생성하는 등입니다.

모든 CLX 컴포넌트에서는 메소드를 신호에 대한 핸들러로 할당하여 원본으로 사용하는 widget의 신호에 응답합니다. 이 때, 원본으로 사용하는 widget과 연결된 특수한 후크 객체를 사용합니다. 후크 객체는 단순히 메소드 포인터들을 모아 놓은 lightweight 객체이며 각 메소드 포인터는 특정 신호와 관련됩니다. CLX 컴포넌트의 메소드를 특정 신호에 대한 핸들러로 후크 객체에 할당한 경우 widget 에서 특정 신호를 만들 때마다 COM 컴포넌트의 메소드를 호출합니다. 그림 51.1에서 이 내용을 설명합니다.

그림 51.1 신호 라우팅



참고 각 후크 객체의 메소드는 Qt 유닛에 선언되어 있습니다. 지정한 후크 객체에 사용할 수 있는 메소드를 보려면 qt.hpp를 확인합니다. 메소드는 자신이 속하는 후크 객체를 반영하는 이름을 가진 전역 루틴으로 결합됩니다. 예를 들어, 애플리케이션 widget(QApplication)과 연관된 후크 객체의 모든 메소드는 'QApplication_hook'로 시작합니다. 오브젝트 파스칼 CLX 객체에서 C++ 후크 객체의 메소드를 액세스하려면 이렇게 결합해야 합니다.

사용자 정의 신호 핸들러 할당

대부분의 CLX 컨트롤은 원본으로 사용하는 widget의 신호를 처리하기 위한 메소드를 미리 할당합니다. 대개 이 메소드는 private 메소드이며 가상 메소드가 아닙니다. 따라서 고유한 메소드를 작성하여 신호에 응답할 경우 widget 과 연관된 후크 객체에 고유한 메소드를 할당해야 합니다. 이렇게 하려면 *HookEvents* 메소드를 오버라이드합니다.

참고 응답할 신호가 시스템 이벤트 통지인 경우 HookEvents 메소드의 오버라이드를 사용하지 말아야 합니다. 시스템 이벤트에 응답하는 방법에 대한 자세한 내용은 "시스템 이벤트에 응답"을 참조하십시오.

HookEvents 메소드의 오버라이드에서 *TMethod* 타입의 변수를 선언합니다. 그런 다음 후크 객체에 신호 핸들러로 할당할 각 메소드에 대해 다음을 수행합니다.

1. 신호의 메소드 핸들러를 나타내는 *TMethod* 타입의 변수를 초기화합니다.
2. 이 변수를 후크 객체에 할당합니다. 컴포넌트가 *THandleComponent*나 *TWidgetControl*에서 상속받은 *Hooks* 속성을 사용하여 후크 객체를 액세스할 수 있습니다.

오버라이드에서 항상 상속받은 *HookEvents* 메소드를 호출하여 기본 클래스에서 할당한 신호 핸들러도 후크되게 합니다.

다음 코드는 *TTrackBar*의 *HookEvents* 메소드를 변환한 것입니다. *HookEvents* 메소드를 오버라이드하여 사용자 정의 신호 핸들러를 추가하는 방법을 설명합니다.

```
virtual void __fastcall TTrackBar::HookEvents(void)
{
  TMethod Method;
  // initialize Method to represent a handler for the QSlider valueChanged
signal
  // ValueChangedHook is a method of TTrackBar that responds to the signal.
  QSlider_valueChanged_Event(Method) = @ValueChangedHook;
  // Assign Method to the hook object. Note that you can cast Hooks to the
  // type of hook object associated with the underlying widget.
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks),
Method);
```

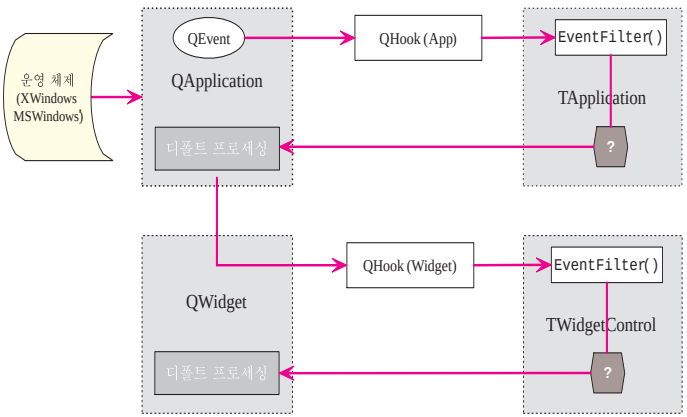
```
  // Repeat the process for the sliderMoved event:
  QSlider_sliderMoved_Event(Method) := @ValueChangedHook;
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks),
Method);
  // Call the inherited method so that inherited signal handlers are
hooked up:
  TWidgetControl::HookEvents();
}
```

시스템 이벤트에 응답

Widget 레이어에서 운영 체제의 이벤트 통지를 받으면 특수한 이벤트 객체(*QEvent* 또는 해당하는 자손 중 하나)를 생성하여 이벤트를 나타냅니다. 이벤트 객체에는 발생한 이벤트에 대한 읽기 전용 정보가 포함되어 있습니다. 이벤트 객체의 타입은 발생한 이벤트의 타입을 나타냅니다.

Widget 레이어는 타입 이벤트의 특수한 신호를 사용하여 CLX 컴포넌트에게 시스템 이벤트를 통지합니다. 그리고 이벤트의 신호 핸들러에 *QEvent* 객체를 전달합니다. 이벤트 신호 처리는 다른 신호 처리보다 애플리케이션 객체에 먼저 전송되기 때문에 약간 더 복잡합니다. 이것은 애플리케이션이 시스템 이벤트에 응답할 기회가 두 번이라는 뜻입니다. 한 번은 애플리케이션 레벨(*TApplication*)에서, 또 한 번은 개별 컴포넌트 레벨(*TWidgetControl* 또는 *THandleComponent* 자손)에서입니다. 이 클래스(*TApplication*, *TWidgetControl* 및 *THandleComponent*)에서는 모두 widget 레이어의 이벤트 신호에 대한 신호 핸들러를 할당합니다. 즉 모든 시스템 이벤트는 자동으로 *EventFilter* 메소드로 전달되는데, 이 메소드는 VCL 컨트롤의 *WndProc* 메소드와 유사한 역할을 합니다. 그림 51.2에서 시스템 이벤트 처리를 설명합니다.

그림 51.2 시스템 이벤트 라우팅



EventFilter 에서는 일반적으로 사용하는 대부분의 시스템 통지를 처리하고 , 이를 컴포넌트의 기본 클래스에 의해 도입된 이벤트로 변환합니다. 따라서 예를 들어, *TWidgetControl* 의 *EventFilter* 메소드는 *OnMouseDown*, *OnMouseMove* 및 *OnMouseUp* 이벤트를 생성하여 마우스 이벤트(*QMouseEvent*)에 응답하고 *OnKeyDown*, *OnKeyPress*, *OnKeyString* 및 *OnKeyUp* 이벤트를 생성하여 키보드 이벤트(*QKeyEvent*)에 응답하는 식입니다.

일반적으로 사용하는 이벤트

*TWidgetControl*의 *EventFilter* 메소드는 *TControl*이나 *TWidgetControl*에서 도입한 protected 메소드에서 호출하여 일반적인 시스템 통지 대부분을 처리합니다. 이 메소드 중 대부분은 가상 메소드이기 때문에 고유한 컴포넌트를 작성할 때 오버라이드할 수 있고 시스템 이벤트에 대한 고유한 응답을 구현할 수 있습니다. 이 메소드를 오버라이드할 경우 이벤트 객체 작업이나 원본으로 사용하는 widget 레이어의 다른 객체 작업(대부분의 경우)에 대해 걱정할 필요가 없습니다.

CLX 컴포넌트에서 시스템 통지에 응답하게 하려면 먼저 통지에 응답하는 protected 메소드가 있는지 확인하는 것이 좋습니다. *TControl* 설명서나 *TWidgetControl* 및 컴포넌트가 파생된 다른 기본 클래스 설명서를 확인하여 해당하는 이벤트에 응답하는 protected 메소드가 있는지 알아볼 수 있습니다. 표 51.1은 *TControl* 및 *TWidgetControl*에서 가장 일반적으로 사용하는 protected 메소드를 나열한 것입니다.

표 51.1 시스템 통지에 응답하기 위한 TWidgetControl protected 메소드

메소드	설명
BeginAutoDrag	컨트롤에 *dmAutomatic*의 *DragMode*가 있을 경우 왼쪽 마우스 버튼을 클릭할 때 호출됩니다.
Click	컨트롤에서 마우스 버튼을 놓을 때 호출됩니다.
DblClick	컨트롤에서 마우스를 더블 클릭할 때 호출됩니다.
DoMouseWheel	마우스 휠을 회전할 때 호출됩니다.
DragOver	컨트롤에서 마우스 커서를 끌 때 호출됩니다.
KeyDown	컨트롤에 포커스가 있을 경우 키를 누를 때 호출됩니다.
KeyPress	*KeyDown*에서 키스토로크를 처리하지 않은 경우 *KeyDown* 다음에 호출됩니다.
KeyString	시스템에서 멀티바이트 문자 시스템을 사용할 경우 키스트로크를 입력할 때 호출됩니다.
KeyUp	컨트롤에 포커스가 있을 경우 키를 놓을 때 호출됩니다.
MouseDown	컨트롤에서 마우스 버튼을 클릭할 때 호출됩니다.
MouseMove	컨트롤에서 마우스 커서를 이동할 때 호출됩니다.
MouseUp	컨트롤에서 마우스 버튼을 놓을 때 호출됩니다.
PaintRequest	시스템에서 컨트롤을 다시 그려야할 때 호출됩니다.
WidgetDestroyed	컨트롤의 원본으로 사용하는 widget이 소멸될 때 호출됩니다.

오버라이드에서 상속받은 메소드를 호출하여 디폴트 프로세스가 여전히 place.respond를 신호로 가져가게 합니다.

참고 시스템 이벤트에 응답하는 메소드 외에도 컨트롤에는 다양한 이벤트에게 컨트롤을 통지하기 위해 *TControl*이나 *TWidgetControl*을 사용하여 발생한 유사한 여러 메소드가 포함됩니다. 이 메소드들은 시스템 이벤트에 응답하지 않지만 VCL 컨트롤에 보내는 많은 Windows 메시지와 같은 작업을 수행합니다. 표 51.1은 이런 메소드 중 일부를 나열한 것입니다.

메소드	설명
BoundsChanged	컨트롤 크기를 다시 조정할 때 호출됩니다.
ColorChanged	컨트롤의 색상이 바뀔 때 호출됩니다.
CursorChanged	커서 모양이 바뀔 때 호출됩니다. 마우스 커서가 이 widget 위에 있을 때 마우스 커서의 모양을 가정합니다.
EnabledChanged	애플리케이션이 윈도우나 컨트롤의 활성화 상태를 변경할 때 호출됩니다.
FontChanged	글꼴 리소스의 컬렉션이 바뀔 때 호출됩니다.
PaletteChanged	Widget의 팔레트가 변경될 때 호출됩니다.
ShowHintChanged	도움말 힌트가 표시되거나 컨트롤에 숨겨졌을 때 호출됩니다.
StyleChanged	윈도우나 컨트롤의 GUI 스타일이 변경되었을 때 호출됩니다.
TabStopChanged	폼의 탭 순서가 변경되었을 때 호출됩니다.
TextChanged	컨트롤의 텍스트가 변경되었을 때 호출됩니다.
VisibleChanged	컨트롤이 숨겨지거나 보여질 때 호출됩니다.

EventFilter 메소드 오버라이드

이벤트 통지에 응답하려하고 오버라이드할 수 있는 해당 이벤트에 protected 메소드가 없을 경우 *EventFilter* 메소드를 오버라이드할 수 있습니다. 오버라이드에서 *EventFilter* 메소드의 *Event* 매개변수 타입을 확인하고 이 타입이 응답하려는 통지 타입을 나타낼 경우 특수한 처리를 수행합니다. *EventFilter* 메소드에서 **true**를 반환하게 하여 이벤트 통지를 더 이상 처리할 수 없게 할 수 있습니다.

참고 다른 타입의 *QEvent* 객체에 대한 자세한 내용은 TrollTech의 Qt 설명서를 참조하십시오.

다음 코드는 *TCustomControl*에서의 *EventFilter* 메소드 변환을 나타냅니다. *EventFilter*를 오버라이드할 때 *QEvent* 객체에서 이벤트 타입을 가져오는 방법을 설명합니다. 여기에서 표시하지는 않았지만 이벤트 타입을 식별했으면 *QEvent* 객체를 해당하는 특화된 *QEvent* 자손(예: *QMouseEvent*)으로 변환할 수 있습니다.

```
virtual bool __fastcall TCustomControl::EventFilter(Qt::QObjectH* Sender,
Qt::QEventH* Event)
{
  bool retval = TWidgetControl::EventFilter(Sender, Event);
  switch (QEvent_type(Event))
  {
    case QEventType_Resize:
    case QEventType_FocusIn:
    case QEventType_FocusOut:
      UpdateMask();
  }
  return retval;
}
```

Qt 이벤트 생성

VCL 컨트롤에서 사용자 정의 Windows 메시지를 정의하고 보낼 수 있는 것처럼 CLX 컨트롤에서 Qt 시스템 이벤트를 정의하고 생성하게 할 수 있습니다. 첫 번째 단계는 이벤트의 고유한 ID를 정의하는 것입니다. 이것은 사용자 정의 Windows 메시지를 정의할 때 메시지 ID를 정의하는 방법과 유사합니다.

```
static const MyEvent_ID = (int) QCLXEventType_ClxUser + 50;
```

이벤트를 생성하는 코드에서 Qt.hpp에서 선언한 *QCustomEvent_create* 함수를 사용하여 새 이벤트 ID를 가진 이벤트 객체를 만듭니다. 옵션인 두 번째 매개변수를 사용하면 이벤트 객체에 이벤트와 연결할 정보의 포인터인 데이터 값을 제공할 수 있습니다.

```
QCustomEventH *MyEvent = QCustomEvent_create(MyEvent_ID, this);
```

이벤트 객체를 만들었으면 *QApplication_postEvent* 메소드를 호출하여 이벤트 객체를 포스트할 수 있습니다.

```
QApplication_postEvent(Application->Handle, MyEvent);
```

컴포넌트가 이 통지에 응답하려면 *MyEvent_ID* 의 이벤트 타입을 확인하여 *EventFilter* 메소드만 오버라이드하면 됩니다. *EventFilter* 메소드는 Qt.hpp에 선언된 *QCustomEvent_data* 메소드를 호출하여 생성자에게 제공한 데이터를 검색할 수 있습니다.

장

52

디자인 타임 시 컴포넌트 사용

이 장에서는 작성한 컴포넌트를 IDE에서 사용 가능하도록 만들기 위한 단계를 설명합니다. 디자인 타임에 컴포넌트를 사용 가능하게 하려면 여러 단계가 필요합니다.

- 컴포넌트 등록
- 팔레트 비트맵 추가
- 컴포넌트를 위한 도움말 제공
- 속성 에디터 추가
- 컴포넌트 에디터 추가
- 컴포넌트를 패키지로 컴파일

모든 컴포넌트에 이러한 모든 단계가 적용되는 것은 아닙니다. 예를 들어, 새로운 속성이나 이벤트를 정의하지 않는 경우 해당 속성이나 이벤트에 대한 도움말을 제공할 필요가 없습니다. 반드시 필요한 단계는 등록과 컴파일입니다.

일단 컴포넌트가 패키지에 등록되고 컴파일되면 다른 개발자에게 배포될 수 있고 IDE에 설치될 수 있습니다. IDE에서의 패키지 설치에 대한 내용은 15-5페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

컴포넌트 등록

등록 작업은 유닛 기반 컴파일을 통해 이루어지므로 하나의 컴파일 유닛에 여러 컴포넌트를 만들면 모두 한꺼번에 등록할 수 있습니다.

컴포넌트를 등록하려면 유닛의 .CPP 파일에 *Register* 함수를 추가해야 합니다. *Register* 함수 내에서 컴포넌트를 등록하고 컴포넌트 팔레트에 설치할 위치를 결정합니다.

참고 IDE에서 Component|New Component를 선택하여 컴포넌트를 생성하는 경우, 컴포넌트를 등록하는 데 필요한 코드가 자동으로 추가됩니다.

컴포넌트를 수동으로 등록하는 단계는 다음과 같습니다.

- Register 함수 선언
- Register 함수 작성

Register 함수 선언

등록은 유닛의 .CPP 파일에 단일 함수를 작성하는 것과 관련됩니다. 이 때 함수에는 반드시 *Register*라는 이름이 있어야 합니다. *Register* 함수는 반드시 네임스페이스 내에 있어야 합니다. 네임스페이스는 컴포넌트가 위치한 파일의 이름이며 첫 문자 외는 모두 소문자로 구성됩니다.

다음은 네임스페이스 내에 *Register* 함수가 구현되는 방법을 보여 주는 코드입니다. 네임스페이스의 이름은 Newcomp이며 파일 이름은 Newcomp.CPP입니다.

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
  }
}
```

Register 함수 내에서 컴포넌트 팔레트에 추가하고자 하는 각각의 컴포넌트에 대해 *RegisterComponents*를 호출합니다. 헤더와 .CPP 파일의 조합에 여러 개의 컴포넌트가 포함되는 경우, 동시에 모두 등록할 수 있습니다. PACKAGE 매크로는 클래스를 임포트하고 엑스포트할 수 있는 명령문으로 확장됩니다.

Register 함수 작성

컴포넌트를 포함하는 유닛의 *Register* 함수 내에서 컴포넌트 팔레트에 추가하고자 하는 각각의 컴포넌트를 등록해야만 합니다. 유닛이 여러 컴포넌트를 포함하고 있는 경우, 동시에 모두 등록할 수 있습니다.

컴포넌트를 등록하려면 컴포넌트를 추가하고자 하는 컴포넌트 팔레트의 각 페이지에 대해 한 번씩 *RegisterComponents* 함수를 호출합니다. *RegisterComponents*에 관련된 세 가지 중요한 내용은 다음과 같습니다.

1 컴포넌트 지정

2 팔레트 페이지 지정

3 RegisterComponents 함수 사용

컴포넌트 지정

Register 함수 내에서, 등록 중인 컴포넌트의 배열이 있는 개방형 배열 타입인 *TComponentClass*를 선언합니다. 구문은 다음과 같습니다.

```
TMetaClass classes[1] = {__classid(TNewComponent)};
```

이런 경우, 클래스의 배열은 단 하나의 컴포넌트만을 포함하지만 원하는 컴포넌트를 모두 추가하여 배열에 등록할 수 있습니다. 예를 들어, 이 코드는 배열에 두 개의 컴포넌트를 둡니다.

```
TMetaClass classes[2] =
                      {__classid(TNewComponent),
__classid(TAnotherComponent)};
```

배열에 컴포넌트를 추가하는 또 다른 방법은 각각의 명령문에서 배열에 컴포넌트를 할당하는 것입니다. 이러한 명령문은 앞의 예제와 같이 두 개의 동일한 컴포넌트를 배열에 추가합니다.

```
TMetaClass classes[2];
classes[0] = __classid(TNewComponent);
classes[1] = __classid(TAnotherComponent);
```

팔레트 페이지 지정

팔레트 페이지 이름은 AnsiString입니다. 팔레트 페이지에 지정한 이름이 존재하지 않는 경우 C++Builder는 해당 이름의 새 페이지를 생성합니다. C++Builder는 문자열 리스트 리소스에 표준 페이지의 이름을 저장하므로 제품의 국제적인 버전에서는 해당 언어로 페이지 이름을 지정할 수 있습니다. 표준 페이지 중 하나에 컴포넌트를 설치할 경우, *LoadStr* 함수를 호출하여 페이지 이름의 문자열을 얻어야 하는데 이 함수는 System 페이지의 *srSystem* 같은 해당 페이지의 문자열 리소스를 나타내는 상수를 전달합니다.

RegisterComponents 함수 사용

Register 함수 내에서 *RegisterComponents*를 호출하여 클래스 배열로 컴포넌트를 등록합니다. *RegisterComponents*는 세 개의 매개변수를 사용하는 함수입니다. 즉, 컴포넌트 팔레트 페이지의 이름, 컴포넌트 클래스의 배열 및 배열의 마지막 엔트리 인덱스를 사용합니다.

NEWCOMP.CPP 파일에 있는 다음 Register 함수는 *TMyComponent*라는 컴포넌트를 등록하고, 이 컴포넌트를 Miscellaneous라는 컴포넌트 팔레트 페이지에 둡니다.

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
      TMetaClass classes[1] = {__classid(TMyComponent)};
      RegisterComponents("Miscellaneous", classes, 0);
  }
}
```

RegisterComponents 호출의 세 번째 인수는 0이며 이는 클래스 배열의 마지막 엔트리의 인덱스, 즉 배열 크기의 -1입니다.

다음 코드에 표시된 대로, 여러 컴포넌트를 같은 페이지에 한꺼번에 등록하거나 다른 페이지에 등록할 수도 있습니다.

```
namespace Mycomps
{
  void __fastcall PACKAGE Register()
  {
      // declares an array that holds two components
      TMetaClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
      // adds a new palette page with the two components in the classes1
array
      RegisterComponents("Miscellaneous", classes1, 1);
      // declares a second array
      TMetaClass classes2[1];
```

```
        // assigns a component to be the first element in the array
        classes2[0]  = __classid(TThird);
        // adds the component in the classes2 array to the Samples page
        RegisterComponents("Samples", classes2, 0);
    }
}
```

예제에서는 두 배열, 즉 classes1과 classes2가 선언되었습니다. 첫 번째 *RegisterComponents* 호출에서 classes1 배열에는 엔트리가 두 개 있으므로 세 번째 인수는 두 번째 엔트리의 인덱스이며 이는 1입니다. 두 번째 *RegisterComponents* 호출에서 classes2 배열에는 요소가 하나이므로 세 번째 인수가 0이 됩니다.

팔레트 비트맵 추가

모든 컴포넌트는 컴포넌트 팔레트에 컴포넌트를 나타내는 비트맵이 필요합니다. 비트맵을 따로 지정하지 않는 경우는 디폴트 비트맵을 사용합니다.

팔레트 비트맵은 디자인 타임에만 필요하므로 컴포넌트의 컴파일 유닛 안으로 비트맵을 컴파일하지 않습니다. 그 대신, .CPP 파일과 동일한 이름을 갖지만 .DCR(동적 컴포넌트 리소스) 확장자를 갖는 Windows 리소스 파일에 팔레트 비트맵을 공급합니다. C++Builder에서 Image Editor를 사용하여 리소스 파일을 생성할 수 있습니다. 각 비트맵은 가로 세로 24 픽셀이어야 합니다.

설치하려는 각 컴포넌트에 대해 팔레트 비트맵 파일을 제공하고 각 팔레트 비트맵 파일 내에서는 등록하려는 각 컴포넌트에 대해 비트맵을 제공합니다. 비트맵 이미지는 컴포넌트 클래스와 동일한 이름을 가집니다. 컴파일된 파일이 있는 동일한 디렉토리에 팔레트 비트맵 파일을 보관하므로 C++Builder가 컴포넌트 팔레트에 컴포넌트를 설치할 때 비트맵을 찾을 수 있습니다.

예를 들어, *TMyControl*이라는 컴포넌트를 만드는 경우, TMYCONTROL이라는 비트맵이 들어있는 .DCR 또는 .RES 리소스 파일을 만들어야 합니다. 리소스 이름은 대소문자를 구별하지 않지만 규칙에 따라 보통 대문자로 되어 있습니다.

컴포넌트를 위한 도움말 제공

폼에서 표준 컴포넌트를 선택하거나 Object Inspector에서 속성이나 이벤트를 선택하는 경우 *F1*을 누르면 해당 항목에 대한 도움말을 볼 수 있습니다. 적절한 도움말 파일을 생성하면 개발자들에게 자신의 컴포넌트에 대해서도 이러한 설명서를 제공할 수 있습니다.

컴포넌트를 설명하는 작은 도움말 파일을 제공할 수도 있고, 도움말 파일이 사용자의 전체 C++Builder 도움말 시스템의 일부가 될 수도 있습니다.

컴포넌트와 함께 사용할 수 있도록 도움말 파일을 구성하는 방법에 대한 자세한 내용은 52-5 페이지의 "도움말 파일 생성" 단원을 참조하십시오.

도움말 파일 생성

Windows 도움말 파일(.rtf 형식)용 소스 파일 생성에 필요한 모든 도구를 사용할 수 있습니다. C++Builder에는 도움말 파일을 컴파일하고 온라인 도움말 작성에 대한 지침을 제공하는 Microsoft Help Workshop이 포함되어 있습니다. Help Workshop용 온라인 안내서에서는 도움말 파일 작성에 필요한 모든 정보를 찾을 수 있습니다.

컴포넌트용 도움말 작성은 다음 단계로 구성됩니다.

- 항목 작성
- 상황에 맞는 컴포넌트 도움말 작성
- 컴포넌트 도움말 파일 추가

항목 작성

컴포넌트의 도움말을 작성하려면 라이브러리 내의 나머지 컴포넌트에 대한 도움말을 유기적으로 통합하고 다음 규칙을 따라야 합니다.

1 **각 컴포넌트는 도움말 항목이 있어야 합니다**.

컴포넌트 항목은 어떤 유닛에서 컴포넌트가 선언되었는지 보여 주어야 하며 컴포넌트에 대한 간략한 설명이 포함되어야 합니다. 컴포넌트 주제는 객체 계층 내에서의 컴포넌트 위치를 설명하는 보조 윈도우에 연결되어야 하며 모든 속성, 이벤트 및 메소드 리스트가 있어야 합니다. 애플리케이션 개발자가 폼에서 컴포넌트를 선택하고 *F1*을 눌러 해당 항목에 액세스할 수 있어야 합니다. 컴포넌트 항목의 예를 보려면 폼의 아무 컴포넌트에 놓고 *F1*을 눌러 보십시오.

컴포넌트 항목에는 항목에 고유한 값이 있는 # 각주가 있어야 합니다. # 각주는 도움말 시스템별로 각 항목을 고유하게 식별합니다.

컴포넌트 항목에는 컴포넌트 클래스의 이름을 포함하는 도움말 시스템 인덱스에서 키워드 검색용 K 각주가 있어야 합니다. 예를 들어, *TMemo* 컴포넌트에 대한 키워드 각주는 "TMemo"입니다.

컴포넌트 항목에는 또한 항목의 타이틀을 제공하는 $ 각주가 있어야 합니다. 타이틀은 Topics Found 다이얼로그 박스, Bookmark 다이얼로그 박스 및 History 윈도우에 나타납니다.

2 **각 컴포넌트는 다음과 같은 검색용 보조 항목이 있어야 합니다**.

- 컴포넌트 계층 내의 컴포넌트의 모든 조상에 대한 링크가 있는 계층 항목
- 컴포넌트에서 사용할 수 있는 모든 속성 리스트. 단, 이러한 속성을 설명하는 항목에 대한 링크가 있어야 함
- 컴포넌트에서 사용할 수 있는 모든 이벤트 리스트. 단, 이러한 이벤트를 설명하는 항목에 대한 링크가 있어야 함
- 컴포넌트에서 사용할 수 있는 모든 메소드 리스트. 단, 이러한 메소드를 설명하는 항목에 대한 링크가 있어야 함

C++Builder 도움말 시스템 내의 객체 클래스, 속성, 메소드 또는 이벤트에 대한 링크는 Alinks를 사용하여 만들 수 있습니다. 객체 클래스에 링크하는 경우 Alink는 객체의 클래스

이름 뒤에 밑줄과 "object" 문자열을 사용합니다. 예를 들어, *TCustomPanel* 객체에 액세스하려면 다음을 사용하십시오.

```
!AL(TCustomPanel_object,1)
```

속성, 메소드 또는 이벤트에 링크하는 경우 속성, 메소드 또는 이벤트의 이름 앞에 이를 구현하는 객체의 이름과 밑줄을 먼저 사용하십시오. 예를 들어, *TControl*에 의해 구현된 *Text* 속성에 링크하려면 다음과 같이 사용하십시오.

```
!AL(TControl_Text,1)
```

검색용 보조 항목의 예를 보려면 임의의 컴포넌트에 대한 도움말을 표시하고 계층, 속성, 메소드 또는 이벤트라는 레이블이 붙은 링크를 클릭하십시오.

3 컴포넌트 내에서 선언된 각 속성, 메소드 및 이벤트는 항목이 있어야 합니다.

속성, 메소드 및 이벤트 항목은 반드시 항목의 선언을 보여 주고 그 사용법을 설명해야 합니다. 애플리케이션 개발자는 Object Inspector 내의 항목을 강조하고 *F1*을 누르거나 항목의 이름에 대한 코드 에디터에 커서를 두고 *F1*을 눌러 이러한 항목을 볼 수 있습니다. 속성 항목의 예를 보려면 Object Inspector 내의 임의의 항목을 선택하고 *F1*을 누르십시오.

속성, 이벤트 및 메소드 항목은 속성, 이벤트 및 메소드의 이름을 나열하는 K 각주 및 컴포넌트 이름과 결합된 각 이름을 포함하고 있어야 합니다. 따라서 *TControl*의 *Text* 속성은 다음과 같은 K 각주를 갖습니다.

```
Text,TControl;TControl,Text;Text,
```

또한 속성, 이벤트 및 메소드 항목은 TControl::Text처럼 항목의 제목을 나타내는 $ 각주를 포함해야 합니다.

이러한 모든 항목은 # 각주로 입력되는, 항목에 고유한 항목 ID가 있어야 합니다.

상황에 맞는 컴포넌트 도움말 작성

각 컴포넌트, 속성, 메소드 및 이벤트 항목에는 A 각주가 있어야 합니다. A 각주는 사용자가 컴포넌트를 선택하고 *F1*을 누르거나 Object Inspector에서 속성이나 이벤트를 선택하고 *F1*을 누르면 항목을 표시하는 데 사용됩니다. A 각주는 반드시 몇가지 명명 규칙을 따라야 합니다.

도움말 항목이 컴포넌트용인 경우, A 각주는 다음 구문을 사용하여 세미콜론으로 분리된 두 개의 독립적인 항목으로 구성됩니다.

```
ComponentClass_Object;ComponentClass
```

여기서, *ComponentClass*는 컴포넌트 클래스의 이름입니다.

도움말 항목이 속성이나 이벤트에 대한 것일 경우, A 각주는 다음 구문처럼 세미콜론으로 분리된 세 개의 항목으로 구성됩니다.

```
ComponentClass_Element;Element_Type;Element
```

여기서, *ComponentClass*는 컴포넌트 클래스의 이름이고 *Element*는 속성, 메소드 또는 이벤트의 이름이며 *Type*은 Property, Method 또는 Event입니다.

예를 들어, *TMyGrid*라는 컴포넌트의 *BackgroundColor*라는 속성의 경우, A 각주는 다음과 같습니다.

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

컴포넌트 도움말 파일 추가

C++Builder에 도움말 파일을 추가하려면 bin 디렉토리나 IDE의 Help|Customize를 통해 액세스할 수 있는 OpenHelp 유틸리티(oh.exe)를 사용합니다.

OpenHelp.hlp 파일에서는 도움말 시스템에 도움말 파일을 추가하는 방법을 비롯하여 OpenHelp 사용에 대한 정보를 얻을 수 있습니다.

속성 에디터 추가

Object Inspector는 모든 타입의 속성에 대한 디폴트 편집 기능을 제공합니다. 그러나 속성 에디터를 작성하고 등록하여 특정 속성에 대한 보조 에디터를 제공할 수 있습니다. 작성하는 컴포넌트의 속성에만 적용되는 속성 에디터를 등록하거나, 특정 타입의 모든 속성에 적용되는 에디터를 만들 수도 있습니다.

가장 간단한 수준에서는, 속성 에디터가 다음 두 가지 방법 중 하나 또는 모두에 의해 작동됩니다. 즉, 텍스트 문자열로 표시하여 사용자가 현재 값을 편집할 수 있게 하거나, 다이얼로그 박스를 표시하여 다른 방법으로 편집할 수 있게 합니다. 편집하는 속성에 따라 어느 한 가지 방법을 사용하거나 두 방법을 모두 사용하는 것이 좋습니다.

속성 에디터를 작성하려면 다음 다섯 단계를 거쳐야 합니다.

1. 속성 에디터 클래스 파생
2. 텍스트로 속성 편집
3. 전체 속성 편집
4. 에디터 어트리뷰트(attribute) 지정
5. 속성 에디터 등록

속성 에디터 클래스 파생

VCL 및 CLX 모두 *TPropertyEditor*의 자손인 여러 종류의 속성 에디터를 사용합니다. 속성 에디터를 만들 때 속성 에디터 클래스는 *TPropertyEditor*의 직계 자손이 되거나 표 52.1에 설명되어 있는 속성 에디터 클래스 중 하나의 간접 자손이 될 수 있습니다. DesignEditors 유닛에 있는 클래스는 VCL 및 CLX 애플리케이션 모두에 대해 사용할 수 있습니다. 그러나 속성 에디터 클래스 중 일부는 특화된 다이얼로그 박스를 제공하여 VCL 또는 CLX 중 하나에 대해서만 사용할 수 있습니다. 이러한 사항은 각각 VCLEditors 및 CLXEditors 유닛에서 찾을 수 있습니다.

참고 속성 에디터는 반드시 *TBasePropertyEditor*의 자손이어야 하고, *IProperty* 인터페이스를 지원해야 합니다. 그러나 *TPropertyEditor*는 *IProperty* 인터페이스의 디폴트 구현을 제공합니다.

표 52.1에 나와 있는 리스트는 전체 리스트가 아닙니다. VCLEditors 및 CLXEditors 유닛도 컴포넌트 이름과 같이 고유한 속성에 의해 사용되는 매우 특화된 속성 에디터를 정의합니다. 아래 리스트에 나열된 속성 에디터는 사용자 정의 속성에 가장 유용합니다.

표 52.1 미리 정의된 속성 에디터 타입

타입	편집되는 속성
TOrdinalProperty	정수, 문자 및 열거 타입 속성을 위한 모든 순서 타입 속성 에디터는 *TOrdinalProperty*의 자손입니다.
TIntegerProperty	미리 정의되고 사용자 정의된 부분범위를 포함하는 모든 정수 타입
TCharProperty	*Char* 타입 및 'A'..'Z' 같은 *Char*의 부분범위
TEnumProperty	열거 타입
TFloatProperty	모든 부동 소수점 숫자
TStringProperty	AnsiStrings
TSetElementProperty	부울 값으로 표시되는 개별 요소 집합
TSetProperty	모든 집합. 집합은 직접 편집할 수는 없지만 집합 요소 속성의 리스트로 확장할 수 있습니다.
TClassProperty	클래스. 클래스 이름을 표시하고 클래스 속성을 확장할 수 있습니다.
TMethodProperty	메소드 포인터들로, 가장 확실한 이벤트입니다.
TComponentProperty	동일한 폼의 컴포넌트. 사용자는 컴포넌트 속성을 편집할 수 없지만 호환 가능한 타입의 특정 컴포넌트를 지정할 수 있습니다.
TColorProperty	컴포넌트 색상. 적용되는 경우 색상 수를 표시하고, 그렇지 않은 경우 16진수 값을 표시합니다. 드롭다운 리스트에 색상 수가 포함되어 있습니다. 더블 클릭하면 색상 선택 다이얼로그 박스가 나타납니다.
TFontNameProperty	글꼴 이름. 드롭다운 리스트에 현재 설치되어 있는 글꼴이 모두 표시됩니다.
TFontProperty	글꼴. 글꼴 다이얼로그 박스에 액세스할 수 있을 뿐만 아니라 개별 글꼴 속성을 확장할 수 있습니다.

다음 예제는 *TMyPropertyEditor*라는 간단한 속성 에디터의 선언을 보여 줍니다.

```
class PACKAGE TMyPropertyEditor : public TPropertyEditor
{
  public:
    virtual bool __fastcall AllEqual(void);
    virtual System::AnsiString __fastcall GetValue(void);
    virtual void __fastcall SetValue(const System::AnsiString Value);
   __fastcall virtual ~TMyPropertyEditor(void) { }
    __fastcall TMyPropertyEditor(void) : Dsgnintf::TPropertyEditor() { }
};
```

텍스트로 속성 편집

모든 속성은 Object Inspector에 표시할 수 있도록 값에 대한 문자열 표현을 제공해야 합니다. 대부분의 속성은 사용자가 새로운 속성 값을 입력할 수 있도록 합니다. 속성 에디터 클래스는 개발자가 오버라이드하여 텍스트 표현과 실제 값을 변환할 수 있는 가상 메소드를 제공합니다.

오버라이드해야 할 메소드는 *GetValue* 및 *SetValue*입니다. 속성 에디터는 표 52.2와 같이 여러 종류의 값을 할당하고 읽는 데 필요한 메소드 집합도 상속합니다.

표 52.2 속성 값을 읽고 쓰기 위한 메소드

속성 타입	Get 메소드	Set 메소드
부동 소수점	GetFloatValue	SetFloatValue
클로저(이벤트)	GetMethodValue	SetMethodValue
순서 타입	GetOrdValue	SetOrdValue
문자열	GetStrValue	SetStrValue

GetValue 메소드를 오버라이드할 때 Get 메소드 중 하나를 호출하고, *SetValue*를 오버라이드할 때 Set 메소드 중 하나를 호출합니다.

속성 값 표시

속성 에디터의 *GetValue* 메소드는 속성의 현재 값을 나타내는 문자열을 반환합니다. Object Inspector는 속성의 값 열에 이 문자열을 사용합니다. 디폴트로, *GetValue*는 "unknown"을 반환합니다.

속성에 대한 문자열 표현을 제공하려면 속성 에디터의 *GetValue* 메소드를 오버라이드합니다.

속성이 문자열 값이 아닌 경우, *GetValue*는 그 값을 문자열 표현으로 변환해야 합니다.

속성 값 설정

속성 에디터의 *SetValue* 메소드는 Object Inspector에서 사용자가 입력한 문자열을 취하여 적절한 타입으로 변환한 다음 속성 값을 설정합니다. 문자열이 속성에 적합한 값을 나타내지 않는 경우, *SetValue*는 예외를 버리고 부적절한 값을 사용하지 않아야 합니다.

문자열 값을 속성으로 읽어들이려면 속성 에디터의 *SetValue* 메소드를 오버라이드합니다.

*SetValue*는 Set 메소드 중 하나를 호출하기 전에 값을 변환하고 검사해야 합니다.

전체 속성 편집

개발자는 사용자가 속성을 시각적으로 편집할 수 있는 다이얼로그 박스를 제공할 수도 있습니다. 속성 에디터는 자신이 클래스인 속성에 가장 많이 사용됩니다. 그 예로, *Font* 속성에서 사용자는 글꼴 다이얼로그 박스를 열어 글꼴의 어트리뷰트(attribute)를 모두 한꺼번에 선택할 수 있습니다.

전체 속성 에디터 다이얼로그 박스를 제공하려면 속성 에디터 클래스의 *Edit* 메소드를 오버라이드합니다.

Edit 메소드는 *GetValue*와 *SetValue* 메소드 작성 시 동일한 Get 메소드와 Set 메소드를 사용합니다. 실제로 *Edit* 메소드는 Get 메소드와 Set 메소드를 모두 호출합니다. 에디터는 타입별로 정의되므로 속성 값을 문자열로 변환할 필요가 없습니다. 에디터는 일반적으로 값을 "검색되는 대로" 처리합니다.

사용자가 속성 옆에 있는 '...' 버튼을 클릭하거나 값 열을 더블 클릭하면 Object Inspector는 속성 에디터의 *Edit* 메소드를 호출합니다.

Edit 메소드의 구현 내에서 다음 단계를 따릅니다.

1 속성에 사용할 에디터를 생성합니다.

2 현재 값을 읽고, Get 메소드를 사용하여 그 값을 속성에 지정합니다.

3 사용자가 새로운 값을 선택하는 경우, Set 메소드를 사용하여 그 값을 속성에 지정합니다.

4 에디터를 제거합니다.

에디터 어트리뷰트(attribute) 지정

속성 에디터는 Object Inspector가 표시할 도구를 결정하는 데 사용할 수 있는 정보를 제공해야 합니다.예를 들어, Object Inspector는 속성이 하위 속성을 가지고 있는지 또는 가능한 값의 리스트를 표시할 수 있는지 여부를 알아야 합니다.

에디터 어트리뷰트를 지정하려면 속성 에디터의 *GetAttributes* 메소드를 오버라이드합니다.

*GetAttributes*는 다음 값 중 하나 또는 전부를 포함할 수 있는 *TPropertyAttributes* 타입의 값 집합을 반환하는 메소드입니다.

표 52.3 속성 에디터 어트리뷰트 플래그

플래그	관련 메소드	포함된 경우의 의미
paValueList	GetValues	에디터는 열거 타입 값의 리스트를 제공할 수 있습니다.
paSubProperties	GetProperties	속성은 표시할 수 있는 하위 속성을 가집니다.
paDialog	Edit	에디터는 전체 속성을 편집하기 위한 다이얼로그 박스를 표시할 수 있습니다.
paMultiSelect	해당 없음	속성은 사용자가 하나 이상의 컴포넌트를 선택할 때 표시되어야 합니다.
paAutoUpdate	SetValue	값이 승인되기를 기다리는 대신 변경이 일어날 때마다 컴포넌트를 업데이트합니다.
paSortList	해당 없음	Object Inspector는 값 리스트를 정렬해야 합니다.
paReadOnly	해당 없음	사용자는 속성 값을 수정할 수 없습니다.
paRevertable	해당 없음	Object Inspector의 컨텍스트 메뉴에 있는 Inherited 메뉴 항목에 Revert를 사용 가능하게 합니다. 메뉴 항목은 현재 속성 값을 버리고 이전에 설정된 기본값이나 표준 값으로 반환하도록 속성 에디터에 알립니다.
paFullWidthName	해당 없음	값을 표시할 필요가 없습니다. 그 대신, Object Inspector는 속성 이름에 전체 폭을 사용합니다.
paVolatileSubProperties	GetProperties	Object Inspector는 속성 값이 변경될 때마다 모든 하위 속성의 값을 다시 가져옵니다.
paReference	GetComponent Value	값이 다른 사항에 대한 참조입니다. paSubProperties와 함께 사용되는 경우 참조되는 객체가 해당 속성에 대한 하위 속성으로 표시되어야 합니다.

Color 속성은 사용자가 Object Inspector에서 선택할 수 있는 여러 가지 방법, 즉 입력, 리스트에서 선택 및 사용자 정의 에디터 등을 제공한다는 점에서 대부분의 속성에 비해 다방면으로 많이 사용됩니다. *TColorProperty* 의 *GetAttributes* 메소드는 그 반환 값에 다음과 같은 몇 가지 어트리뷰트(attribute)를 포함합니다.

```
virtual __fastcall TPropertyAttributes TColorProperty::GetAttributes()
{
  return TPropertyAttributes() << paMultiSelect << paDialog <<
paValueList << paRevertable;
}
```

속성 에디터 등록

일단 속성 에디터를 생성하면 C++Builder에 등록해야 합니다. 속성 에디터를 등록하려면 속성 타입을 특정 속성 에디터에 연결해야 합니다. 지정된 타입의 모든 속성을 갖는 에디터를 등록하거나 특정 컴포넌트 타입의 특정 속성만을 갖는 에디터를 등록할 수 있습니다.

속성 에디터를 등록하려면 *RegisterPropertyEditor* 함수를 호출하십시오.

*RegisterPropertyEditor*는 다음의 네 가지 매개변수를 취합니다.

- 편집할 속성의 타입에 대한 타입 정보 포인터. 타입 정보를 다음과 같이 지정하십시오.

  ```
  __typeinfo(TMyComponent)
  ```

- 이 에디터가 적용되는 컴포넌트 타입. 이 매개변수가 Null인 경우, 에디터는 지정된 타입의 모든 속성에 적용됩니다.
- 속성의 이름. 이 매개변수는 이전의 매개변수가 특정 컴포넌트 타입을 지정하는 경우에만 의미를 가집니다. 이 경우, 이 에디터가 적용되는 컴포넌트 타입의 특정 속성의 이름을 지정할 수 있습니다.
- 지정된 속성을 편집하는 데 사용할 속성 에디터의 타입

다음은 컴포넌트 팔레트에 표준 컴포넌트의 에디터를 등록하는 함수에서 발췌한 것입니다.

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    RegisterPropertyEditor(__typeinfo(TComponent), 0L, "",
__classid(TComponentProperty));
    RegisterPropertyEditor(__typeinfo(TComponentName),
__classid(TComponent), "Name",
        __classid(TComponentNameProperty));
    RegisterPropertyEditor(__typeinfo(TMenuItem), __classid(TMenu), "",
        __classid(TMenuItemProperty));
    f
  }
}
```

이 함수의 세 명령문은 *RegisterPropertyEditor*가 다르게 사용된 예를 보여 줍니다.

- 첫 번째 명령문은 가장 일반적입니다. 이 명령문은 *TComponent*(또는 고유한 에디터가 등록되어 있지 않은 *TComponent*의 자손) 타입의 모든 속성에 대해 *TComponent Property* 속성 에디터를 등록합니다. 일반적으로, 속성 에디터를 등록하는 경우 특정 타입의 에디터를 만든 다음 그 타입의 모든 속성에 대해 그 에디터를 사용하고자 하므로 두 번째와 세 번째 매개변수는 각각 **NULL**과 비어 있는 문자열입니다.
- 두 번째 명령문은 가장 구체적인 등록 유형입니다. 이 명령문은 특정 컴포넌트 타입에서 특정 속성에 대한 에디터를 등록합니다. 이런 경우, 에디터는 모든 컴포넌트의 *TComponentName* 타입의 *Name* 속성에 대한 것입니다.
- 세 번째 문장은 첫 번째 문장보다는 구체적이지만 두 번째 문장만큼 제한적이지는 않습니다. 이 명령문은 *TMenu* 타입의 컴포넌트에서 *TMenuItem* 타입의 모든 속성에 대한 에디터를 등록합니다.

속성 범주

IDE에서 Object Inspector를 통해 속성 범주별로 속성을 선택적으로 표시하고 숨길 수 있습니다. 새로운 사용자 정의 컴포넌트의 속성은 범주에 속성을 등록하여 이 스키마에 맞출 수 있습니다. *RegisterPropertyInCategory* 또는 *RegisterPropertiesInCategory*를 호출하여 컴포넌트를 등록하는 동시에 이 작업을 수행합니다. *RegisterPropertyInCategory*를 사용하여 단일 속성을 등록합니다. *RegisterPropertiesInCategory*를 사용하여 단일 함수 호출에서 여러 속성을 등록합니다. 이러한 함수는 DesignIntf 유닛에 정의되어 있습니다.

반드시 속성을 등록해야 하는 것은 아니며, 또한 일부 속성을 등록할 때 반드시 사용자 정의 컴포넌트의 모든 속성을 등록해야 하는 것은 아닙니다. 범주에 명시적으로 연결되어 있지 않은 속성은 *TMiscellaneousCategory* 범주에 포함됩니다. 이러한 속성은 디폴트 범주에 따라 Object Inspector에서 표시되거나 숨겨집니다.

속성 등록과 관련하여 이 두 가지 함수 외에도 *IsPropertyInCategory* 함수가 있습니다. 이 함수는 지정된 속성 범주에 특정 속성이 등록되어 있는지를 확인해야 하는 지역화 유틸리티를 생성하는 데 유용합니다.

한 번에 하나의 속성 등록

한 번에 하나의 속성을 등록하고, *RegisterPropertyInCategory* 함수를 사용하여 그 속성을 속성 범주에 연결합니다. *RegisterPropertyInCategory*는 네 가지 오버로드된 변형으로 나타나는데, 각 변형은 속성 범주에 연결할 사용자 정의 컴포넌트의 속성을 식별하기 위해 다른 기준 집합을 제공합니다.

첫 번째 변형을 사용하면 속성 이름에 의해서 속성을 식별할 수 있습니다. 아래 행은 컴포넌트의 시각적 표시에 관련된 속성을 등록하여 "AutoSize"라는 이름으로 속성을 식별합니다.

```
RegisterPropertyInCategory(“Visual”, “AutoSize”);
```

두 번째 변형은 지정된 타입의 컴포넌트에 나타나는 지정된 이름의 속성에만 범주를 제한한다는 점을 제외하면 첫 번째 변형과 매우 유사합니다. 다음은 *TMyButton*이라는 사용자 정의 클래스 컴포넌트의 "HelpContext" 속성을 'Help and Hints' 범주에 등록하는 예제입니다.

```
RegisterPropertyInCategory(“Help and Hints”, __classid(TMyButton),
“HelpContext”);
```

세 번째 변형은 이름이 아닌 타입을 사용하여 속성을 식별합니다. 다음은 TArrangement라는 특수 클래스 타입을 기준으로 속성을 등록하는 예제입니다.

```
RegisterPropertyInCategory('Visual', typeid(TArrangement));
```

마지막 변형은 속성 타입과 이름을 사용하여 속성을 식별합니다. 다음은 *TBitmap* 타입과 "Pattern" 이름을 기준으로 속성을 등록하는 예제입니다.

```
RegisterPropertyInCategory(“Visual”, typeid(TBitmap), “Pattern”);
```

유용한 속성 범주와 사용 설명을 나열한 리스트를 보려면 속성 범주 지정 단원을 참조하십시오.

한 번에 여러 속성 등록

한 번에 여러 속성을 등록하고, *RegisterPropertiesInCategory* 함수를 사용하여 그 속성들을 속성 범주에 연결합니다. *RegisterPropertiesInCategory*는 세 가지 오버로드된 변형으로 나타나는데 각 변형은 속성 범주에 연결할 사용자 정의 컴포넌트의 속성을 식별하기 위해 다른 기준 집합을 제공합니다.

첫 번째 변형을 사용하면 속성 이름이나 타입에 기반하여 속성을 식별할 수 있습니다. 리스트는 상수의 배열로 전달됩니다. 아래 예제에서 "Text"라는 이름을 갖거나 *TEdit* 타입의 클래스에 속하는 속성은 'Localizable' 범주에 등록됩니다.

```
RegisterPropertiesInCategory("Localizable", ARRAYOFCONST("Text",
__typeinfo(TEdit)));
```

두 번째 변형을 사용하면 특정 컴포넌트에 속하는 속성 범주에 대해 등록된 속성을 제한할 수 있습니다. 등록할 속성의 리스트에는 타입이 포함되지 않고 이름만 포함됩니다. 예를 들어, 다음 코드는 모든 컴포넌트에 대해 다수의 속성을 'Help and Hints' 범주에 등록합니다.

```
RegisterPropertyInCategory("Help and Hints", __classid(TComponent),
ARRAYOFCONST("HelpContext", "Hint", "ParentShowHint"));
```

세 번째 변형을 사용하면 사용자가 특정 타입을 갖는 속성 범주에 대해 등록된 속성을 제한할 수 있습니다. 두 번째 변형에서 등록할 속성의 리스트에는 이름만 포함될 수 있습니다.

```
RegisterPropertiesInCategory("Localizable", __typeinfo(TStrings),
ARRAYOFCONST("Lines", "Commands"));
```

유용한 속성 범주와 사용 설명을 나열한 리스트를 보려면 속성 범주 지정 단원을 참조하십시오.

속성 범주 지정

범주에서 속성을 등록할 때 원하는 모든 문자열을 범주 이름으로 사용할 수 있습니다. 이전에 사용되지 않았던 문자열을 사용하는 경우, Object Inspector는 그 이름을 갖는 새로운 속성 범주 클래스를 생성합니다. 또한 기본 범주 중 하나에 속성을 등록할 수도 있습니다. 기본 속성 범주는 표 52.4에 설명되어 있습니다.

표 52.4 속성 범주

범주	용도
Action	*TEdit*의 *Enabled* 및 *Hint* 속성은 런타임 작업에 관련된 속성입니다.
Database	*TQuery*의 *DatabaseName* 및 *SQL* 속성은 데이터베이스 연산에 관련된 속성입니다.
Drag, Drop, and Docking	*TImage*의 *DragCursor* 및 *DragKind* 속성은 드래그 앤 드롭 및 도킹 연산에 관련된 속성입니다.
Help and Hints	*TMemo*의 *HelpContext* 및 *Hint* 속성은 온라인 도움말이나 힌트 사용에 관련된 속성입니다.
Layout	*TDBEdit*의 *Top* 및 *Left* 속성은 디자인 타임에 컨트롤의 시각적 표시와 관련된 속성입니다.
Legacy	*TComboBox*의 *Ctl3D* 및 *ParentCtl3D* 속성은 폐기된 연산에 관련된 속성입니다.
Linkage	*TDataSource*의 *DataSet* 속성은 한 컴포넌트를 다른 컴포넌트에 연결하는 것과 관련된 속성입니다.
Locale	*TMainMenu*의 *BiDiMode* 및 *ParentBiDiMode* 속성은 국제 로케일에 관련된 속성입니다.
Localizable	애플리케이션의 지역화 버전에서 수정이 필요한 속성. *Caption*과 같은 많은 문자열 속성, 즉 컨트롤의 크기 및 위치를 결정하는 속성입니다.
Visual	*TScrollBox*의 *Align* 및 *Visible* 속성은 런타임 시 컨트롤의 가시적인 표시와 관련된 속성입니다.
Input	*TEdit*의 *Enabled* 및 *ReadOnly* 속성은 데이터베이스 연산에 관련될 필요는 없고 데이터 입력과 관련이 있는 속성입니다.
Miscellaneous	TSpeedButton의 AllowAllUp 및 Name 속성은 범주에 맞지 않거나 범주화될 필요가 없는 속성 및 특정 범주에 명시적으로 등록되지 않은 속성입니다.

IsPropertyInCategory 함수 사용

애플리케이션에서 기존 등록된 속성을 쿼리하여 지정된 속성이 이미 특정 범주에 등록되어 있는지 여부를 결정합니다. 이런 작업은 지역화 작업을 수행하기 위해 속성의 범주화를 확인하는 지역화 유틸리티와 같은 상황에서 특히 유용합니다. *IsPropertyInCategory* 함수의 두 가지 오버로드된 변형은 속성이 범주에 있는지 여부를 결정할 때 다른 기준을 고려합니다.

첫 번째 변형을 사용하여 소유하고 있는 컴포넌트의 클래스 타입과 속성 이름의 결합에 대한 비교 기준을 만들 수 있습니다. 아래 명령줄에서 **true**를 반환하는 *IsPropertyInCategory*에 대한 속성은 *TCustomEdit* 자손에 속해야 하고 "Text"라는 이름을 가지며 'Localizable' 속성 범주에 포함되어야 합니다.

```
IsItThere = IsPropertyInCategory(“Localizable”, __classid(TCustomEdit),
“Text”);
```

두 번째 변형을 사용하면 소유하고 있는 컴포넌트의 클래스 이름과 속성 이름의 결합에 대한 비교 기준을 만들 수 있습니다. 아래 명령줄에서 **true**를 반환하는 *IsPropertyInCategory*에 대한 속성은 *TCustomEdit*의 자손이어야 하고 "Text"라는 이름을 가지며 'Localizable' 속성 범주에 포함되어야 합니다.

```
IsItThere = IsPropertyInCategory(“Localizable”, “TCustomEdit”, “Text”);
```

컴포넌트 에디터 추가

컴포넌트 에디터는 디자이너에서 컴포넌트를 더블 클릭했을 때 일어나는 작업을 결정하고 마우스 오른쪽 버튼으로 컴포넌트를 클릭하면 나타나는 컨텍스트 메뉴에 명령을 추가합니다. 또한 컴포넌트 에디터는 컴포넌트를 사용자 정의 형식으로 Windows 클립보드에 복사할 수 있습니다.

컴포넌트 에디터에 컴포넌트를 제공하지 않으면 C++Builder가 디폴트 컴포넌트 에디터를 사용합니다. 디폴트 컴포넌트 에디터는 *TDefaultEditor* 클래스에 의해 구현됩니다. *TDefaultEditor*는 컴포넌트의 컨텍스트 메뉴에 새 항목을 추가하지 않습니다. 컴포넌트를 더블 클릭하면 *TDefaultEditor*가 해당 컴포넌트의 속성을 검색하고 첫 번째로 찾은 이벤트 핸들러를 생성하거나 탐색합니다.

컨텍스트 메뉴에 항목을 추가하려면 컴포넌트를 더블 클릭할 때 행동을 변경하거나, 새 클립보드 형식을 추가하고 *TComponentEditor*에서 새 클래스를 파생한 후 사용자의 컴포넌트에 그 용도를 등록합니다. 오버라이드된 메소드에서 *TComponentEditor*의 *Component* 속성을 사용하여 편집 중인 컴포넌트에 액세스할 수 있습니다.

사용자 정의 컴포넌트 에디터 추가 작업은 다음 단계로 이루어집니다.

- 컨텍스트 메뉴에 항목 추가
- 더블 클릭 동작 변경
- 클립보드 형식 추가
- 컴포넌트 에디터 등록

컨텍스트 메뉴에 항목 추가

컴포넌트에서 마우스 오른쪽 버튼을 클릭하면 컴포넌트 에디터의 *GetVerbCount* 및 *GetVerb* 메소드가 호출되어 컨텍스트 메뉴를 만듭니다. 이러한 메소드를 오버라이드하여 컨텍스트 메뉴에 명령(동사)을 추가할 수 있습니다.

컨텍스트 메뉴에 항목을 추가하려면 다음 단계가 필요합니다.

- 메뉴 항목 지정
- 명령 구현

메뉴 항목 지정

GetVerbCount 메소드를 오버라이드하여 컨텍스트 메뉴에 추가할 명령의 수를 반환합니다. *GetVerb* 메소드를 오버라이드하여 이러한 명령 각각에 대해 추가할 문자열을 반환합니다. *GetVerb*를 오버라이드할 때 문자열에 앰퍼샌드(&)를 추가하면 컨텍스트 메뉴에서 다음 문자에 밑줄이 표시되며 메뉴 항목을 선택하기 위한 단축키로 작동합니다. 다이얼로그 박스가 나타날 경우에는, 명령 끝에 생략 부호(...)를 추가해야 합니다. *GetVerb*는 명령 인덱스를 나타내는 하나의 매개변수를 가집니다.

다음 코드는 *GetVerbCount* 및 *GetVerb* 메소드를 오버라이드하여 컨텍스트 메뉴에 두 개의 명령을 추가합니다.

```
int __fastcall TMyEditor::GetVerbCount(void)
{
  return 2;
}
System::AnsiString __fastcall TMyEditor::GetVerb(int Index)
{
  switch (Index)
  {
    case 0: return “&DoThis ...”; break;
    case 1: return “Do&That”; break;
  }
}
```

참고 *GetVerb* 메소드는 *GetVerbCount*에 의해 표시된 모든 가능한 인덱스의 값을 반환해야 합니다.

명령 구현

*GetVerb*가 제공하는 명령을 디자이너에서 선택하면 *ExecuteVerb* 메소드가 호출됩니다. *GetVerb* 메소드에서 제공하는 모든 명령에 대해 *ExecuteVerb* 메소드에서 동작을 구현합니다. 에디터의 *Component* 속성을 사용하여 편집 중인 컴포넌트에 액세스할 수 있습니다.

예를 들어, 다음 *ExecuteVerb* 메소드는 이전 예제에서 *GetVerb* 메소드의 명령을 구현합니다.

```
void __fastcall TMyEditor::ExecuteVerb(int Index)
{
  switch (Index)
  {
```

```
    case 0:
      TMyDialog *MySpecialDialog = new TMyDialog();
      MySpecialDialog->Execute();
      ((TMyComponent *)Component)->ThisProperty = MySpecialDialog-
>ReturnValue;
      delete MySpecialDialog;
      break;
    case 1:
      That();  // call the “That” method
      break;
  }
}
```

더블 클릭 동작 변경

컴포넌트를 더블 클릭하면 컴포넌트 에디터의 *Edit* 메소드가 호출됩니다. 디폴트로, *Edit* 메소드는 컨텍스트 메뉴에 추가된 첫 번째 명령을 실행합니다. 따라서, 이전 예제에서 컴포넌트를 더블 클릭하면 *DoThis* 명령이 실행됩니다.

일반적으로 첫 번째 명령을 실행하는 것이 바람직하지만 이 디폴트 동작을 변경할 수도 있습니다. 예를 들어, 다음과 같은 경우에 대체 동작을 제공할 수 있습니다.

- 컨텍스트 메뉴에 명령을 추가하지 않을 경우
- 컴포넌트를 더블 클릭했을 때 여러 개의 명령이 결합된 다이얼로그 박스를 표시하기를 원할 경우

컴포넌트를 더블 클릭한 경우, *Edit* 메소드를 오버라이드하여 새로운 동작을 지정합니다. 예를 들어, 다음 *Edit* 메소드는 사용자가 컴포넌트를 더블 클릭할 때 글꼴 다이얼로그 박스를 나타냅니다.

```
void __fastcall TMyEditor::Edit(void)
{
  TFontDialog *pFontDlg = new TFontDialog(NULL);
  pFontDlg->Execute();
  ((TMyComponent *)Component)->Font = pFontDlg->Font;
  delete pFontDlg;
}
```

참고 컴포넌트를 더블 클릭해서 이벤트 핸들러의 코드 에디터를 표시하고자 하는 경우, *TComponentEditor* 대신 *TDefaultEditor*를 컴포넌트 에디터의 기본 클래스로 사용합니다. 그런 다음 *Edit* 메소드를 오버라이드하지 않고 보호되는 *TDefaultEditor*::*EditProperty* 메소드를 사용합니다. *EditProperty*는 컴포넌트의 이벤트 핸들러를 전체적으로 검색하여 첫 번째로 찾은 이벤트를 표시합니다. 특정 이벤트를 보기 위해서 이 작업을 변경할 수도 있습니다. 예를 들면, 다음과 같습니다.

```
void __fastcall TMyEditor::EditProperty(TPropertyEditor* PropertyEditor,
                                        bool &Continue, bool &FreeEditor)
{
  if (PropertyEditor->ClassNameIs(“TMethodProperty”) &&
      CompareText(PropertyEditor->GetName, “OnSpecialEvent”) == 0)
  {
    TDefaultEditor::EditProperty(PropertyEditor, Continue, FreeEditor);
```

```
  }
}
```

클립보드 형식 추가

디폴트로, 컴포넌트가 IDE에 선택되어 있는 동안 사용자가 Copy를 선택한 경우 해당 컴포넌트는 C++Builder의 내부 형식으로 복사됩니다. 그런 다음 다른 폼이나 데이터 모듈로 붙여넣을 수 있습니다. 컴포넌트는 *Copy* 메소드를 오버라이드하여 추가 형식을 클립보드에 복사할 수 있습니다.

예를 들어, 다음 *Copy* 메소드를 통해 *TImage* 컴포넌트에서 해당 그림을 클립보드에 복사할 수 있습니다. 이 그림은 C++Builder IDE에서 무시되지만 다른 애플리케이션에 붙여 넣을 수 있습니다.

```
void __fastcall TMyComponentEditor::Copy(void)
{
  WORD AFormat;
  int AData;
  HPALETTE APalette;
  ((TImage *)Component)->Picture->SaveToClipboardFormat(AFormat, AData,
APalette);
  TClipboard *pClip = Clipboard(); // don't clear the clipboard!
  pClip->SetAsHandle(AFormat, AData);
}
```

컴포넌트 에디터 등록

일단 컴포넌트 에디터가 정의되면 특정 컴포넌트 클래스를 사용하기 위해 등록될 수 있습니다. 등록된 컴포넌트 에디터가 폼 디자이너에서 선택되어 있는 경우, 해당 클래스의 각 컴포넌트에 대해 등록된 컴포넌트 에디터를 만듭니다.

컴포넌트 에디터와 컴포넌트 클래스를 연결하려면 *RegisterComponentEditor*를 호출합니다. *RegisterComponentEditor*는 에디터를 사용하는 컴포넌트 클래스의 이름과 정의된 컴포넌트 에디터 클래스의 이름을 취합니다. 예를 들어, 다음 명령문은 *TMy Component* 타입의 모든 컴포넌트를 사용하는 *TMyEditor*라는 컴포넌트 에디터 클래스를 등록합니다.

```
RegisterComponentEditor(__classid( TMyComponent), __classid(TMyEditor));
```

컴포넌트를 등록하는 namespace에서 *RegisterComponentEditor*를 호출합니다. 예를 들어, *TMyComponent*라는 새로운 컴포넌트 및 해당 컴포넌트 에디터인 *TMyEditor*가 모두 NewComp.cpp에서 구현되는 경우, NewComp.cpp에 있는 다음 코드는 컴포넌트 및 컴포넌트 에디터와의 연결 관계를 등록합니다.

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    TMetaClass classes[1] = {__classid(TMyComponent)};
    RegisterComponents("Miscellaneous", classes, 0);
    RegisterComponentEditor(classes[0], __classid(TMyEditor));
  }
}
```

컴포넌트를 패키지로 컴파일

일단 컴포넌트가 등록되면 IDE에 설치하기 전에 패키지로 컴파일해야 합니다. 패키지는 사용자 정의된 속성 에디터뿐만 아니라 하나 이상의 컴포넌트를 가질 수 있습니다. 패키지에 대한 자세한 내용은 15장,"패키지와 컴포넌트 사용"을 참조하십시오.

패키지를 생성하고 컴파일하려면 15-6페이지의 "패키지 생성 및 편집"을 참조하십시오. 패키지의 Contains 리스트에 사용자 정의 컴포넌트의 소스 코드 유닛을 포함시킵니다. 컴포넌트가 다른 패키지에 종속되어 있는 경우에는 해당 패키지를 Requires 리스트에 포함시킵니다.

IDE에 컴포넌트를 설치하려면 15-5페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

사용자 정의 컴포넌트의 문제 해결

사용자 정의 컴포넌트를 등록하고 설치할 때 발생하는 가장 일반적인 문제는 패키지가 성공적으로 설치된 후에도 해당 컴포넌트가 컴포넌트 리스트에 표시되지 않는다는 점입니다.

컴포넌트가 리스트나 팔레트에 표시되지 않는 가장 일반적인 이유는 다음과 같습니다.

- Register 함수에서 PACKAGE 변경자 누락
- 클래스에서 PACKAGE 변경자 누락
- C++ 소스 파일에서 **#pragma** `package(smart_init)` 누락
- 소스 코드 모듈과 이름이 같은 Register 함수를 네임스페이스에서 찾을 수 없음
- Register가 성공적으로 엑스포트되지 않음. .BPL에서 tdump를 사용하여 엑스포트된 함수를 검색하십시오.

  ```
  tdump -ebpl mypack.bpl  mypack.dmp
  ```

  덤프의 엑스포트 섹션에 네임스페이스 안에 있는 Register 함수가 엑스포트되는 것이 표시되어야 합니다.

장

53

기존 컴포넌트 수정

컴포넌트를 만드는 가장 쉬운 방법은 원하는 대부분의 작업을 수행하는 컴포넌트에서 파생시킨 컴포넌트를 필요한 대로 변경하는 것입니다. 이 장의 예제에서는 표준 메모 컴포넌트를 수정하여 노란색 배경을 가진 메모를 만듭니다. 이 부분의 다른 장에서는 더 복잡한 컴포넌트 생성 방법에 대해 설명합니다. 기본 프로세스는 항상 같지만 더 복잡한 컴포넌트에서는 새로운 클래스를 사용자 정의하기 위한 단계가 더 필요합니다.

기존 컴포넌트 수정은 다음 두 단계만 거치면 됩니다.

- 컴포넌트 생성 및 등록
- 컴포넌트 클래스 수정

컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 동일한 방법으로 시작됩니다. 유닛을 만들고, 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록한 다음 컴포넌트 팔레트에 설치합니다. 이 프로세스는 45-8페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같이 컴포넌트를 만드는 일반적인 프로시저를 따릅니다.

1. 컴포넌트의 유닛 이름을 *YelMemo*라고 지정한 다음 저장합니다. 그러면 헤더 파일은 YELMEMO.H가 되고 CPP 파일은 YELMEMO.CPP가 됩니다.
2. *TMemo*의 자손인 *TYellowMemo*라는 새 컴포넌트 클래스를 파생시킵니다.
3. 컴포넌트 팔레트의 Samples 페이지(또는 CLX의 다른 페이지)에서 *TYellowMemo*를 등록합니다.

결과 헤더 파일은 다음과 같아야 합니다.

```
#ifndef YelMemoH
#define YelmemoH
//---------------------------------------------------------------------------
```

```
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------------------
class PACKAGE TYellowMemo : public TMemo
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

함께 만들어지는 .CPP 파일은 다음과 같아야 합니다.

```
#include <vcl.h>
#pragma hdrstop
#include "Yelmemo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not
have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TYellowMemo *)
{
  new TYellowMemo(NULL);
}
//---------------------------------------------------------------------------
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
        : TMemo(Owner)
{
}
//---------------------------------------------------------------------------
namespace Yelmemo
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TYellowMemo)};
    RegisterComponents("Samples", classes, 0); //"Common Controls" in CLX
applications
  }
}
```

CLX CLX 애플리케이션은 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX의 `<clx\qcontrols.hpp>`입니다.

참고 이 예제에서는 컴포넌트 마법사를 사용하여 이 컴포넌트를 만들지 않고 수동으로 만들고 있다고 가정합니다. 컴포넌트 마법사를 사용할 경우 자동으로 생성자가 *TYellowMemo*에 추가됩니다.

컴포넌트 클래스 수정

새 컴포넌트 클래스를 만들었으면 거의 모든 방법으로 새 컴포넌트 클래스를 수정할 수 있습니다. 이 경우에는 메모 컴포넌트에서 한 속성의 초기 값만 변경합니다. 여기에는 다음과 같이 컴포넌트 클래스에 대한 두 가지 작은 변경 사항이 포함됩니다.

- 생성자 오버라이드
- 새 디폴트 속성 값 지정

실제로 생성자는 속성의 값을 설정합니다. 기본값은 C++Builder에게 폼 파일(VCL의 .dfm 및 CLX의 .xfm)에 저장할 값을 알려줍니다. C++Builder에서는 기본값과 다른 값만 저장하므로 두 단계를 모두 수행해야 합니다.

생성자 오버라이드

디자인 타임 시 폼에 컴포넌트가 있거나 런타임 시 애플리케이션에서 컴포넌트를 생성할 경우 컴포넌트의 생성자가 속성 값을 설정합니다. 컴포넌트를 폼 파일에서 로드한 경우 애플리케이션에서 디자인 타임 시 변경된 속성을 설정합니다.

참고 생성자를 오버라이드할 경우 다른 작업을 수행하기 전에 새로운 생성자가 상속받은 생성자를 호출해야 합니다. 자세한 내용은 46-9페이지의 "가상 메소드"를 참조하십시오.

이 예제의 경우 새로운 컴포넌트가 *TMemo*에서 상속받은 생성자를 오버라이드하여 *Color* 속성을 *clYellow*로 설정해야 합니다. 이렇게 하기 위해 생성자 오버라이드 선언을 클래스 선언에 추가한 다음 .CPP 부분 파일에 새로운 생성자를 작성합니다.

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner); // the constructor
declaration
__published:
    __property Color;
};

__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
  : TMemo(Owner)                                    // the constructor
implementation first...
                                                     // ...calls the
constructor for TMemo
{
  Color = clYellow;                                 // colors the component
yellow
}
```

참고 컴포넌트 마법사를 사용하여 컴포넌트를 만든 경우 `Color = clYellow;`를 기존 생성자에 추가하기만 하면 됩니다.

이제는 컴포넌트 팔레트에 새 컴포넌트를 설치하고 이를 폼에 추가할 수 있습니다. 이제 *Color* 속성은 디폴트로 *clYellow*가 됩니다.

새 디폴트 속성 값 지정

C++Builder에서 폼에 대한 설명을 폼 파일에 저장한 경우 기본값과 다른 속성의 값만 저장합니다. 다른 값만 저장하면 폼 파일이 작아지고 폼을 더 빠르게 로드할 수 있습니다. 속성을 만들거나 기본값을 변경할 경우 속성 선언을 업데이트하여 새 기본값을 포함시키는 것이 좋습니다. 폼 파일, 로드 및 기본값에 대해서는 52장, "디자인 타임 시 컴포넌트 사용"에서 자세히 설명합니다.

다음과 같은 방법으로 속성의 기본값을 변경합니다.

1 속성 이름을 재선언합니다.

2 속성 이름 다음에 등호(=)를 붙입니다.

3 괄호 안에 키워드 **default**를 입력하고, 등호를 입력한 다음 기본값을 입력합니다.

전체 속성을 재선언할 필요는 없고 이름과 기본값만 재선언합니다.

노란색의 메모 컴포넌트의 경우 클래스 선언의 **__published** 부분에서 *clYellow* 기본값을 사용하여 *Color* 속성을 재선언합니다.

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
};
```

여기서 *TYellowMemo*가 다시 나오지만, 이번에는 다른 published 속성 *WordWrap*을 갖고 있고 기본값은 **false**입니다.

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
    __property WordWrap = {default=false};
};
```

디폴트 속성 값을 지정해도 컴포넌트의 작업에 전혀 영향을 미치지 않습니다. 컴포넌트의 생성자에서 기본값을 명시적으로 설정해야 합니다. 애플리케이션의 내부 작업에는 차이가 있습니다. *TYellowMemo*의 경우에는 WordWrap이 **false**일 경우 C++Builder에서 더 이상 폼 파일에 *WordWrap*을 작성하지 않습니다. 생성자가 해당 값을 자동으로 설정할 것이라고 이미 알려주었기 때문입니다. 다음은 생성자입니다.

```
__fastcall TYellowMemo::TYellowMemo(TComponent* AOwner) : TMemo(AOwner)
{
  Color = clYellow;
  WordWrap = false;
}
```

장

54

그래픽 컴포넌트 생성

그래픽 컨트롤은 간단한 유형의 컴포넌트입니다. 순수한 그래픽 컨트롤은 포커스를 받지 않으므로 윈도우 핸들을 갖거나 필요로 하지 않습니다. 사용자는 마우스로 컨트롤을 처리할 수 있지만 키보드 인터페이스는 없습니다.

이 장에 설명되어 있는 그래픽 컨트롤은 컴포넌트 팔레트의 Additional 페이지에 있는 도형 컴포넌트인 *TShape*입니다. 만든 컴포넌트가 표준 도형 컴포넌트와 동일하더라도 중복 식별자를 피하기 위해서는 만든 컴포넌트에 다른 이름을 지정해야 합니다. 이 장에서는 도형 컴포넌트를 *TSampleShape*라고 하며 도형 컴포넌트의 만드는데 관련된 모든 단계를 설명합니다.

- 컴포넌트 생성 및 등록
- 상속된 속성 게시
- 그래픽 기능 추가

컴포넌트 생성 및 등록

모든 컴포넌트 생성 과정은 동일한 방법으로 시작됩니다. 컴포넌트 클래스를 파생시켜 해당 컴포넌트의 .CPP 및 .H 파일을 저장하고, 컴포넌트 클래스를 파생시켜 해당 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이 프로세스는 45-8페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

참고 이 예제에서는 개발자가 컴포넌트 마법사를 사용하지 않고 수동으로 컴포넌트를 만든다고 가정합니다.

이 예제에서는 다음과 같이 컴포넌트를 만드는 일반적인 프로시저를 따릅니다.

1. *TGraphicControl*의 자손인 *TSampleShape*라는 새 컴포넌트 타입을 파생시킵니다.
2. 컴포넌트의 헤더 파일인 SHAPES.H와 .CPP 파일 SHAPES.CPP을 호출합니다.
3. 컴포넌트 팔레트의 Sample 페이지(CLX의 경우 다른 페이지)에 *TSampleShape*를 등록합니다.

결과 헤더 파일은 다음과 같아야 합니다.

```
//---------------------------------------------------------------------------
#ifndef ShapesH
#define ShapesH
//---------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleShape : public TGraphicControl
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

.CPP 파일은 다음과 같아야 합니다.

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Shapes.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//

static inline void ValidCtrCheck(TSampleShape *)
{
  new TSampleShape(NULL);
}
//---------------------------------------------------------------------------
__fastcall TSampleShape::TGraphicControl(TComponent* Owner)
        : TGraphicControl(Owner)
{
}
//---------------------------------------------------------------------------
namespace Shapes
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleShape)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

CLX CLX 애플리케이션은 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX의 `<clx\qcontrols.hpp>`입니다.

상속된 속성 게시

일단 컴포넌트 타입을 파생시키면 조상 클래스의 protected 부분에서 선언된 속성과 이벤트 중 어떤 속성과 이벤트를 새 컴포넌트에서 표면에 나타낼 지 결정할 수 있습니다. *TGraphic Control* 이 컴포넌트가 컨트롤 기능을 수행하도록 하는 모든 속성을 이미 게시하고 있으므로 개발자는 마우스 이벤트에 응답하고 드래그 앤 드롭 작업을 처리하는 기능을 게시하기만 하면 합니다.

상속된 속성 및 이벤트 게시에 대해서는 47-2페이지의 "상속된 속성 게시"와 48-5페이지의 "이벤트 표시"에서 설명합니다. 두 가지 프로세스에서는 클래스 선언의 published 부분에 있는 속성 이름만을 재정의하면 됩니다.

도형 컨트롤에서 세 개의 마우스 이벤트, 세 개의 드래그 앤 드롭 이벤트, 두 개의 드래그 앤 드롭 속성을 게시할 수 있습니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
__published:
    __property DragCursor ;
    __property DragMode ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnMouseDown ;
    __property OnMouseMove ;
    __property OnMouseUp ;
};
```

이제 사용자는 샘플 도형 컨트롤을 통해 마우스 및 드래그 앤 드롭 기능을 사용할 수 있습니다.

그래픽 기능 추가

일단 그래픽 컴포넌트를 선언하고 사용하고자 하는 상속된 속성을 게시하면 컴포넌트를 구별하는 그래픽 기능을 추가할 수 있습니다. 그래픽 컨트롤을 만들 때 수행할 두 가지 작업은 다음과 같습니다.

1 그릴 대상 결정

2 컴포넌트 이미지 그리기

두 가지 작업에 덧붙여서 다음의 도형 컨트롤 예제에서는 애플리케이션 개발자가 디자인 타임에 도형 모양을 사용자 정의할 수 있도록 하는 속성을 추가합니다.

그릴 대상 결정

그래픽 컨트롤은 모양을 변경하여 사용자 입력을 포함한 동적 조건을 반영할 수 있습니다. 모양이 변경되지 않는 그래픽 컨트롤은 컴포넌트가 아닐 수 있습니다. 정적 이미지를 원할 경우, 컨트롤을 사용하는 대신 이미지를 임포트할 수 있습니다.

일반적으로 그래픽 컨트롤의 모양은 일부 속성 조합에 따라 달라집니다. 예를 들어, 계측기(gauge) 컨트롤에는 모양과 방향 그리고 진행 상황을 그래픽 뿐만 아니라 수치적으로 표시할지 여부를 결정하는 속성이 있습니다. 이와 마찬가지로, 도형 컨트롤에는 그려야 할 도형의 종류를 결정하는 속성이 있습니다.

그릴 도형을 결정하는 속성을 컨트롤에 제공하려면 *Shape*라는 속성을 추가합니다. 속성 제공을 위해서는 다음 작업을 수행해야 합니다.

1 속성 타입 선언

2 속성 선언

3 구현 메소드 작성

속성 생성 방법에 대해서는 47장, "속성 생성"에 자세히 설명되어 있습니다.

속성 타입 선언

사용자 정의 타입의 속성을 선언할 경우, 속성을 포함하는 클래스를 선언하기 전에 먼저 속성 타입을 선언해야 합니다. 속성에 대한 사용자 정의 타입의 가장 일반적인 유형은 열거(enumerated) 타입입니다.

도형 컨트롤에서는 컨트롤이 그릴 수 있는 각 도형의 종류에 대한 요소를 갖는 열거 타입이 필요합니다.

도형 컨트롤 클래스의 선언 위에 다음과 같은 타입 정의를 추가합니다.

```
enum TSampleShapeType { sstRectangle, sstSquare, sstRoundRect,
sstRoundSquare, sstEllipse, sstCircle };

class PACKAGE TSampleShape : public TGraphicControl         // this is
already there
```

이제 이 타입을 사용하여 클래스에서 새 속성을 선언할 수 있습니다.

속성 선언

속성을 선언할 때 속성에 대한 데이터를 저장하려면 private 데이터 멤버를 선언한 다음 속성 값을 읽고 쓰기 위한 메소드를 지정해야 합니다. 경우에 따라 값을 읽기 위해 메소드를 사용할 필요가 없으며 대신 저장된 데이터를 가리키기만 하면 됩니다.

도형 컨트롤에서는 현재 도형을 유지하는 데이터 멤버를 선언한 다음 그 데이터 멤버를 읽는 속성을 선언하고 메소드 호출을 통해 해당 데이터 멤버에 씁니다.

다음 선언을 *TSampleShape*에 추가합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
    TSampleShapeType FShape;
    void __fastcall SetShape(TSampleShapeType Value);
__published:
    __property TSampleShapeType Shape = {read=FShape, write=SetShape,
nodefault};
};
```

이제 마지막으로 *SetShape*의 구현을 추가하기만 하면 됩니다.

구현 메소드 작성

속성 정의의 **read** 또는 **write** 부분이 저장된 속성 데이터에 직접 액세스하는 대신 메소드를 사용하는 경우에는 해당 메소드를 구현해야 합니다.

SetShape 메소드의 구현을 SHAPES.CPP 파일에 추가합니다.

```
void __fastcall TSampleShape::SetShape(TSampleShapeType Value)
{
  if (FShape != Value)             // ignore if this isn't a change
  {
     FShape = Value;               // store the new value
     Invalidate();                 // force a repaint with the new shape
  }
}
```

생성자 및 소멸자 오버라이드

디폴트 속성 값을 변경하고 컴포넌트에 대해 소유된 클래스를 초기화하려면 상속된 생성자와 소멸자를 오버라이드합니다. 두 가지 경우 모두, 새로운 생성자나 소멸자에서 상속된 메소드를 항상 호출해야 합니다.

디폴트 속성 값 변경

그래픽 컨트롤의 디폴트 크기가 다소 작기 때문에 생성자에서 너비와 높이를 변경할 수 있습니다. 디폴트 속성 값 변경에 대해서는 53장, "기존 컴포넌트 수정"에 자세히 설명되어 있습니다.

이 예제에서 도형 컨트롤의 크기는 각 변이 65픽셀인 사각형으로 설정됩니다.

컴포넌트 클래스의 선언에 오버라이드된 생성자를 추가합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent *Owner);
};
```

컴포넌트 마법사를 사용하여 컴포넌트를 만들기 시작했다면 다음 단계는 컴포넌트 마법사가 이미 수행했을 것입니다.

1 새 기본값을 갖는 *Height*와 *Width* 속성을 재선언합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
  f
__published:
    __property Height;
    __property Width;
}
```

2 .CPP 파일에 새 생성자를 씁니다.

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) :
TGraphicControl(Owner)
```

```
{
  Width = 65;
  Height = 65;
}
```

컴포넌트 마법사를 사용하는 경우 새 기본값을 기존의 생성자에 추가하기만 하면 됩니다.

펜과 브러시 게시

디폴트로, 캔버스에는 가는 검정색 펜과 단색 흰색 브러시가 있습니다. 개발자가 펜과 브러시를 변경하도록 하려면 디자인 타임에 처리할 수 있는 클래스를 제공한 다음 색칠하는 도중에 캔버스에 클래스를 복사해야 합니다. 컴포넌트는 보조 펜이나 브러시를 소유하고 컴포넌트가 보조 펜과 브러시를 만들고 소멸시키므로 보조 펜이나 브러시 같은 클래스를 *소유된 클래스*라고 합니다.

소유된 클래스 관리에는 다음과 같은 단계가 필요합니다.

1. 클래스 데이터 멤버 선언
2. 액세스 속성 선언
3. 소유된 클래스 초기화
4. 소유된 클래스의 속성 설정

데이터 멤버 선언

컴포넌트가 소유하는 각 클래스에는 컴포넌트에서 해당 클래스에 대해 선언된 데이터 멤버가 있어야 합니다. 데이터 멤버는 컴포넌트가 항상 소유하고 있는 객체에 대한 포인터를 갖고 있어서 자신을 제거하기 전에 클래스를 제거할 수 있도록 합니다. 일반적으로 컴포넌트는 소유하고 있는 객체를 각각의 생성자에서 초기화하고, 소멸자에서 제거합니다.

소유된 객체의 데이터 멤버는 거의 항상 private로 선언됩니다. 애플리케이션이나 다른 컴포넌트가 소유된 객체에 액세스해야 하는 경우에는 **published** 또는 **public** 속성을 선언하여 액세스할 수 있습니다.

도형 컨트롤에 펜과 브러시의 데이터 멤버를 추가합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:                // data members are always private
    TPen *FPen;         // a data member for the pen object
    TBrush *FBrush;     // a data member for the brush object
    f
};
```

액세스 속성 선언

객체 타입의 속성을 선언하여 컴포넌트의 소유된 객체에 대한 액세스를 제공할 수 있습니다. 이렇게 하면 개발자가 디자인 타임이나 런타임 시 객체에 액세스할 수 있습니다. 일반적으로 속성의 읽기 부분은 데이터 멤버를 참조하지만, 쓰기 부분은 컴포넌트가 소유된 객체의 변경 내용에 반응할 수 있도록 하는 메소드를 호출합니다.

펜과 브러시 데이터 멤버에 대한 액세스를 제공하는 속성을 도형 컨트롤에 추가합니다. 또한 펜이나 브러시의 변경 사항에 반응하기 위한 메소드를 선언합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
  f
private:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall SetBrush(TBrush *Value);
    void __fastcall SetPen(TPen *Value);
    f
__published:
    __property TBrush* Brush = {read=FBrush, write=SetBrush, nodefault};
    __property TPen* Pen = {read=FPen, write=SetPen, nodefault};
};
```

그리고 나서 *SetBrush* 및 *SetPen* 메소드를 .CPP 파일에 씁니다.

```
void __fastcall TSampleShape::SetBrush( TBrush* Value)
{
  FBrush->Assign(Value);
}
void __fastcall TSampleShape::SetPen( TPen* Value)
{
  FPen->Assign(Value);
}
```

다음과 같은 방법으로 *Value*의 값을 *FBrush*에 직접 할당하면

```
  FBrush = Value;
```

*FBrush*에 대한 내부 포인터를 덮어 쓰고, 메모리를 손실하며, 소유권 문제가 많이 발생합니다.

소유된 클래스 초기화

클래스를 컴포넌트에 추가할 때 컴포넌트의 생성자는 사용자가 런타임에 객체와 상호 작용할 수 있도록 클래스를 초기화해야 합니다. 이와 마찬가지로, 컴포넌트의 소멸자는 소유된 객체를 소멸하고 나서 컴포넌트 자체를 소멸해야 합니다.

펜과 브러시를 도형 컨트롤에 추가했기 때문에 해당 펜과 브러시를 도형 컨트롤 생성자에서 초기화하고 컨트롤 소멸자에서 소멸해야 합니다.

1 도형 컨트롤 생성자에서 펜과 브러시를 생성합니다.

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) :
TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();                                  // construct the
pen
  FPen = new TPen();                                      // construct the
brush
}
```

2 컴포넌트 클래스의 선언에 오버라이드된 소멸자를 추가합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
  f
public:                                                 // destructors are
always public
    virtual __fastcall TSampleShape(TComponent* Owner);
    __fastcall ~TSampleShape();                         // the destructor
    f
};
```

3 새로운 소멸자를 .CPP 파일에 씁니다.

```
__fastcall TSampleShape::~TSampleShape()
{
  delete FPen;                                          // delete the pen
object
  delete FBrush;                                        // delete the
brush object
}
```

소유된 클래스의 속성 설정

펜과 브러시 클래스를 처리하는 최종 단계에서는, 펜과 브러시가 변경되면 도형 컨트롤이 그 자체를 다시 그리는지 확인할 필요가 있습니다. 펜과 브러시 클래스는 모두 *OnChange* 이벤트를 가지므로 도형 컨트롤에서 메소드를 만들어서 펜과 브러시의 *OnChange* 이벤트가 모두 메소드를 가리키도록 할 수 있습니다.

다음 메소드를 도형 컨트롤에 추가하고, 컴포넌트 생성자를 업데이트하여 펜과 브러시 이벤트를 새로운 메소드로 설정합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
  f
public:
    void __fastcall StyleChanged(TObject* Owner);
    f
};
```

SHAPES.CPP 파일에서 *StyleChanged* 메소드를 *TSampleShape* 생성자에 있는 펜과 브러시 클래스에 대한 *OnChange* 이벤트에 할당합니다.

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) :
TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();
  FBrush->OnChange = StyleChanged;
  FPen = new TPen();
  FPen->OnChange = StyleChanged;
}
```

StyleChanged 메소드의 구현을 포함합니다.

```
void __fastcall TSampleShape::StyleChanged( TObject* Sender)
{
  Invalidate();         // repaints the component
}
```

이렇게 변경되면 컴포넌트는 다시 그려서 펜이나 브러시에 대한 변경 사항을 반영합니다.

컴포넌트 이미지 그리기

그래픽 컨트롤의 필수 요소는 그래픽 컨트롤이 화면에 이미지를 그리는 방법입니다. 추상 타입인 *TGraphicControl*은 컨트롤에 원하는 이미지를 그리기 위해 오버라이드하는 *Paint*라는 메소드를 정의합니다.

도형 컨트롤에 대한 *Paint* 메소드는 다음을 수행해야 합니다.

- 사용자가 선택한 펜과 브러시를 사용합니다.
- 선택한 도형을 사용합니다.
- 정사각형과 원이 동일한 너비와 높이를 사용하도록 좌표를 조정합니다.

Paint 메소드를 오버라이드하기 위해서는 다음 두 단계의 작업이 필요합니다.

1. *Paint*를 컴포넌트 선언에 추가합니다.
2. *Paint* 메소드를 .CPP 파일에 씁니다.

도형 컨트롤의 경우 클래스 선언에 다음 선언을 추가합니다.

```
class PACKAGE TSampleShape : public TGraphicControl
{
  f
protected:
    virtual void __fastcall Paint();
    f
};
```

그리고 나서 해당 메소드를 .CPP 파일에 씁니다.

```
void __fastcall TSampleShape::Paint()
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                                  // copy the
component's pen
  Canvas->Brush = FBrush;                              // copy the
component's brush
  W=Width;                                             // use the component
width
  H=Height;                                            // use the component
height
  X=Y=0;                                               // save smallest for
circles/squares
  if( W<H )
    S=W;
  else
```

```
    S=H;
  switch(FShape)
  {
    case sstRectangle:                                  // draw rectangles
and squares
    case sstSquare:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundRect:                                  // draw rounded
rectangles and squares
    case sstRoundSquare:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                                     // draw circles and
ellipses
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

*Paint*는 컨트롤이 이미지를 업데이트해야 할 때마다 호출됩니다. 컨트롤은 처음 나타나거나 컨트롤 앞에 있는 윈도우가 사라질 때 그려집니다. 또한, *StyleChanged* 메소드와 마찬가지로 *Invalidate* 메소드를 호출하면 강제적으로 다시 그리게 할 수 있습니다.

정교한 도형 그리기

표준 도형 컨트롤은 예제 도형 컨트롤이 아직 수행하지 않은 작업을 한 가지 더 수행합니다. 표준 도형 컨트롤은 직사각형과 타원 뿐만 아니라 정사각형과 원을 처리합니다. 정사각형과 원을 처리하려면 가장 짧은 면을 찾아 이미지를 가운데에 배치하는 코드를 작성해야 합니다.

다음은 정사각형과 타원에 대해 조정하는 정교한 *Paint* 메소드입니다.

```
void __fastcall TSampleShape::Paint(void)
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                          // copy the component's pen
  Canvas->Brush = FBrush;                      // copy the component's
brush
  W=Width;                                     // use the component width
  H=Height;                                    // use the component height
  X=Y=0;                                       // save smallest for
circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)                                // adjust height, width and
position
```

```
  {
    case sstRectangle:
    case sstRoundRect:
    case sstEllipse:
      Y=X=0;                                  // origin is top-left for
these shapes
      break;
    case sstSquare:
    case sstRoundSquare:
    case sstCircle:
      X= (W-S)/2;                              // center these horizontally
      Y= (H-S)/2;                              // and vertically
      break;
    default:
    break;
  }
  switch(FShape)
  {
    case sstSquare:                            // draw rectangles and
squares
      W=H=S;                                   // use shortest dimension
for width and height
    case sstRectangle:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundSquare:                       // draw rounded rectangles
and squares
      W=H=S;
    case sstRoundRect:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                            // draw circles and ellipses
      W=H=S;
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

장

55

그리드 사용자 정의

C++Builder는 사용자 정의 컴포넌트의 기반으로 사용할 수 있는 추상 컴포넌트를 제공합니다. 추상 컴포넌트에서 가장 중요한 것은 그리드와 리스트 박스입니다. 이 장에서는 기본 그리드 컴포넌트인 *TCustomGrid*에서 한 달을 출력하는 작은 달력을 만드는 방법을 보여 줍니다.

달력을 만드는 데는 다음과 같은 작업들이 포함됩니다.

- 컴포넌트 생성 및 등록
- 상속된 속성 게시
- 초기 값 변경
- 셀 크기 조정
- 셀 채우기
- 연도와 월 탐색
- 일 탐색

VCL 애플리케이션에서의 결과 컴포넌트는 컴포넌트 팔레트의 Sample 페이지에 있는 *TCalendar* 컴포넌트와 유사합니다. CLX 애플리케이션에서는 컴포넌트를 다른 페이지에 저장하거나 새 팔레트 페이지를 만듭니다. 자세한 내용은 52-3페이지의 "팔레트 페이지 지정" 또는 온라인 도움말의 "Component palette, adding pages"를 참조하십시오.

컴포넌트 생성 및 등록

모든 컴포넌트를 같은 방법으로 만듭니다. 컴포넌트 클래스를 파생시켜 해당 컴포넌트의 .CPP 및 .H 파일을 저장하고, 컴포넌트 클래스를 파생시켜 해당 컴포넌트 클래스를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이 프로세스는 45-8페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같은 특성으로 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

1 *TCustomGrid*의 자손인 *TSampleCalendar*라는 새 컴포넌트 타입을 파생시킵니다.

2 컴포넌트의 헤더 파일인 CALSAMP.H와 해당 .CPP 파일인 CALSAMP.CPP를 호출합니다.

3 컴포넌트 팔레트의 Samples 페이지나 CLX의 다른 페이지에 *TSampleCalendar*를 등록합니다. CLX 애플리케이션의 경우 다른 컴포넌트 팔레트 페이지를 사용합니다.

결과 헤더 파일은 다음과 같아야 합니다.

```
#ifndef CalSampH
#define CalSampH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
#include <vcl\grids.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

CALSAMP.CPP 파일은 다음과 같아야 합니다.

```
#include <vcl\vcl.h>
#pragma hdrstop
#include "CalSamp.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TSampleCalendar *ValidCtrCheck()
{
  return new TSampleCalendar(NULL);
}
//---------------------------------------------------------------------------
namespace Calsamp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleCalendar)};
    RegisterComponents("Samples", classes, 0); //In CLX, use a different
page than “Samples”
  }
}
```

참고 컴포넌트 마법사를 사용하여 컴포넌트를 작성하면 헤더 파일이 새 생성자도 선언하며, CALSAMP.CPP 파일에 생성자의 시작 부분이 포함됩니다. 생성자가 없으면 나중에 추가할 수 있습니다.

CLX CLX 애플리케이션은 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX의 `<clx\qcontrols.hpp>`입니다.

상속된 속성 게시

추상 그리드 컴포넌트인 *TCustomGrid*는 많은 **protected** 속성을 제공합니다. 이러한 속성 중에서 달력 컨트롤 사용자가 사용할 수 있는 속성을 선택할 수 있습니다.

컴포넌트의 사용자가 상속된 protected 속성을 사용할 수 있게 하려면 컴포넌트 선언의 **_published** 부분에서 속성을 재선언하십시오.

달력 컨트롤에서는 여기에 표시된 속성과 이벤트를 게시합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
f
__published:
    __property Align ;                    // publish properties
    __property BorderStyle ;
    __property Color ;
    __property Font ;
    __property GridLineWidth ;
    __property ParentColor ;
    __property ParentFont ;
    __property OnClick ;                   // publish events
    __property OnDblClick ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnKeyDown ;
    __property OnKeyPress ;
    __property OnKeyUp ;
};
```

사용자가 그릴 그리드 선을 선택할 수 있는 *Options* 속성처럼, 게시할 수는 있지만 달력에 적용하지는 못하는 많은 속성들이 있습니다.

컴포넌트 팔레트에 수정된 달력 컴포넌트를 설치하고 그 컴포넌트를 애플리케이션에서 사용하는 경우, 완전한 기능을 갖는 달력에서 다양한 속성과 이벤트를 사용할 수 있습니다. 이제 직접 디자인한 새 기능을 추가할 수 있습니다.

초기 값 변경

달력은 고정된 수의 행과 열을 갖는 그리드가 필수적이지만 모든 행에 항상 날짜가 포함되는 것은 아닙니다. 따라서 *ColCount* 및 *RowCount* 그리드 속성을 게시하지 않습니다. 이는 달력 사용자가 주 7일 이외에 다른 것을 표시하려고 하지 않을 것이기 때문입니다. 그러나 이러한 속성의 초기 값을 설정하여 일주일이 항상 7일이 되게 해야 합니다.

컴포넌트 속성의 초기 값을 변경하려면 생성자를 오버라이드하여 원하는 값을 설정합니다.

생성자를 컴포넌트 클래스 선언의 **public** 부분에 추가한 다음 헤더 파일에 새 생성자를 작성해야 한다는 것에 유의하십시오.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
protected:
    virtual void __fastcall DrawCell(int ACol, int ARow, const
Windows::TRect &Rect,
      TGridDrawState AState);
    f
public:
    __fastcall TSampleCalendar(TComponent *Owner);   // the added
constructor
    f
};
```

CALSAMP.CPP 파일에 생성자 코드를 작성합니다.

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) :
TCustomGrid(Owner)
{
  ColCount = 7;
  RowCount = 7;
  FixedCols = 0;
  FixedRows = 1;
  ScrollBars = ssNone;
  Options = (Options >> goRangeSelect) << goDrawFocusSelected;
}
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const
Windows::TRect
  &ARect, TGridDrawState AState)
{
}
```

참고 *DrawCell* 메소드도 클래스 선언에 추가되고 .CPP 파일에서 *DrawCell* 메소드가 시작된 것을 알 수 있습니다. 지금 반드시 필요한 것은 아니지만 *DrawCell*을 오버라이드하기 전에 *TSampleCalendar*를 테스트해야 하는 경우에는 가상 함수 오류가 발생합니다. 이는 *TCustomGrid*가 추상 클래스이기 때문입니다. *DrawCell* 오버라이드에 대해서는 "셀 채우기" 단원의 뒷부분에서 설명합니다.

이제 달력에는 열과 행이 각각 일곱 개씩 있으며, 맨 위의 행이 고정되어 있거나 스크롤되지 않습니다.

셀 크기 조정

VCL 사용자나 애플리케이션이 윈도우 또는 컨트롤의 크기를 변경하면 나중에 이미지를 새 크기로 그리는 데 필요한 설정을 조정할 수 있도록 Windows에서 *WM_SIZE*라는 메시지를 해당 윈도우나 컨트롤로 보냅니다. VCL 컴포넌트는 셀 크기를 변경하여 이 통지에 대해 반응할 수 있으므로 컨트롤 경계 내에서 모두 맞춰집니다. *WM_SIZE* 메시지에 응답하기 위해 컴포넌트에 메시지 처리 메소드를 추가합니다.

메시지 처리 메소드를 만드는 데 대한 자세한 내용은 51-6페이지의 "새 메시지 핸들러 생성"을 참조하십시오.

이 경우 달력 컨트롤에는 *WM_SIZE*에 대한 응답이 필요하므로 *WMSize*라는 protected 메소드를 컨트롤에 추가한 다음 모든 셀이 새로운 크기로 보여질 수 있는 적합한 셀 크기를 계산하도록 메소드를 작성합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
f
protected:
    void __fastcall WMSize(TWMSize &Message);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize)
END_MESSAGE_MAP(TCustomGrid)
};
```

다음은 CALSAMP.CPP 파일의 메시지를 위한 코드입니다.

```
void __fastcall TSampleCalendar::WMSize(TWMSize &Message)
{
  int GridLines;                                    // temporary local
variable
  GridLines = 6 * GridLineWidth;                    // calculated combined
size of all lines
  DefaultColWidth = (Message.Width - GridLines) / 7;    // set new
default cell width
  DefaultRowHeight = (Message.Height - GridLines) / 7;  // and cell height
}
```

이제 크기가 조정된 달력은 컨트롤에 맞는 최대 크기로 모든 셀을 표시합니다.

CLX CLX에서 윈도우나 컨트롤의 크기를 변경하면 보호된 *BoundsChanged* 메소드가 호출되어 자동으로 알립니다. CLX 컴포넌트는 셀 크기를 변경하여 이 알림에 대해 반응할 수 있으므로 컨트롤 경계 내에 모두 맞춰집니다.

이 경우 달력 컨트롤은 모든 셀이 새로운 크기로 보여질 수 있는 적합한 셀 크기를 계산하기 위해서 *BoundsChanged*를 오버라이드해야 합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
f
protected:
    void __fastcall BoundsChanged(void);
};
```

다음은 CALSAMP.CPP 파일의 메소드를 위한 코드입니다.

```
void __fastcall TSampleCalendar::BoundsChanged(void)
{
  int GridLines;                                    // temporary local
variable
  GridLines = 6 * GridLineWidth;                    // calculated combined
size of all lines
  DefaultColWidth = (Width - GridLines) / 7;    // set new default cell
width
  DefaultRowHeight = (Height - GridLines) / 7;  // and cell height
  TCustomGrid::BoundsChanged(); // now call the inherited method
}
```

셀 채우기

그리드 컨트롤은 셀 단위로 내용을 채웁니다. 달력의 경우는 각 셀에 속하는 날짜를 계산하는 것을 의미합니다. 그리드 셀의 디폴트 그리기는 *DrawCell*이라는 가상 메소드에서 수행됩니다.

그리드 셀의 내용을 채우려면 *DrawCell* 메소드를 오버라이드합니다.

채우기가 가장 쉬운 부분은 고정 행의 헤더 셀입니다. 런타임 라이브러리에는 짧은 요일 이름의 배열이 들어 있으므로 달력의 경우 각 열에 적절한 요일 이름을 사용합니다.

다음은 *DrawCell* 메소드를 위한 코드입니다.

```
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const
Windows::TRect &ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol + 1];
  else
  {
    TheText = "";
    TempDay = DayNum(ACol, ARow);                    // DayNum is defined
later
    if (TempDay != -1) TheText = IntToStr(TempDay);
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left
   - Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) /
2, TheText);
}
```

날짜 추적

달력 컨트롤을 유용하게 사용하려면 사용자와 애플리케이션이 일, 월 및 연도를 설정할 수 있는 메커니즘이 있어야 합니다. C++Builder는 *TDateTime* 타입의 변수로 날짜와 시간을 저장합니다. *TDateTime*은 날짜와 시간의 인코드된 숫자 표시로 프로그래밍 조작에는 유용하나 사용자가 사용하기에는 불편합니다.

따라서 인코드된 폼으로 날짜를 저장하고 그 값에 런타임 액세스를 제공할 수 있으며, 달력 컴포넌트의 사용자가 디자인 타임에 설정할 수 있는 *Day*, *Month* 및 *Year* 속성을 제공할 수 있습니다.

달력에서의 날짜 추적은 다음과 같은 단계로 수행됩니다.

- 내부 날짜 저장
- 년, 월, 일 액세스
- 일(day) 수 생성
- 현재 날짜 선택

내부 날짜 저장

달력의 날짜를 저장하려면, 날짜를 담는 private 데이터 구성원과 그 날짜에 액세스를 제공하는 런타임 전용 속성이 필요합니다.

1 날짜를 유지할 private 필드를 선언합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TDateTime FDate;
    f
};
```

2 생성자에서 날짜 필드를 초기화합니다.

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) :
TCustomGrid(Owner)
{
  f
  FDate = FDate.CurrentDate();
}
```

3 인코드된 날짜에 액세스하도록 하는 런타임 속성을 선언합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
public:
    __property TDateTime CalendarDate = {read=FDate,
write=SetCalendarDate, nodefault};
    f
};
```

날짜를 설정하려면 컨트롤의 온스크린 이미지를 업데이트해야 하기 때문에 날짜를 설정하기 위한 메소드가 필요합니다. *TSampleCalendar*에서 *SetCalendarDate*를 선언합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    void __fastcall SetCalendarDate(TDateTime Value);
    f
};
```

다음은 *SetCalendarDate* 메소드입니다.

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                        // Set the new date value
  Refresh();                            // Update the onscreen image
}
```

년, 월, 일 액세스

애플리케이션에는 인코드된 숫자로 표현된 날짜가 적합하지만 사람들은 년, 월, 일 표시를 사용하는 것을 더 선호합니다 . 속성을 생성함으로써 인코드되어 저장된 요소들에 대하여 대체 액세스를 제공할 수 있습니다.

날짜의 각 요소(일, 월, 연도)가 정수이고, 각 요소를 설정하려면 설정 시 날짜를 인코드해야 하기 때문에, 세 개의 속성 모두에서 구현 메소드를 공유하면 매번 코드를 중복시키는 것을 피할 수 있습니다. 즉, 두 개의 메소드를 작성할 수 있는데 하나는 요소를 읽는 메소드이고 다른 하나는 요소를 쓰는 메소드입니다. 이러한 메소드를 사용하여 세 개의 속성을 모두 만들고 설정할 수 있습니다.

일, 월, 연도에 디자인 타임 액세스를 제공하려면 다음을 수행하십시오.

1 세 개의 속성을 선언하고 각 속성에 고유한 인덱스 번호를 할당합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
  f
public:
    __property int Day = {read=GetDateElement, write=SetDateElement,
index=3,
      nodefault};
    __property int Month = {read=GetDateElement, write=SetDateElement,
index=2,       nodefault};
    __property int Year = {read=GetDateElement, write=SetDateElement,
index=1,       nodefault};
};
```

2 구현 메소드를 선언하고 작성하여 인덱스 값마다 다른 요소를 설정합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int __fastcall GetDateElement(int Index);           // note the Index
parameter
    void __fastcall SetDateElement(int Index, int Value);
    f
};
```

다음은 *GetDateElement* 및 *SetDateElement* 메소드입니다.

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay);            // break encoded
date into elements
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}

void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
```

```
  if (Value > 0)                                        // all elements must
be positive
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);      // get current date
elements
    switch (Index)
    {
      case 1: AYear = Value;     break;
      case 2: AMonth = Value;    break;
      case 3: ADay = Value;      break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay);          // encode the modified
date
  Refresh();                                            // update the visible
calendar
}
```

이제 디자인 타임에 Object Inspector를 사용하거나 런타임에 코드를 사용하여 달력의 일, 월, 연도를 설정할 수 있습니다. 물론 셀에 날짜를 쓰기 위한 코드는 아직 추가하지 않았지만 이제 필요한 데이터는 가지고 있습니다.

일(day) 수 생성

달력에 숫자를 입력할 때는 여러 가지를 고려해야 합니다. 월의 일(day) 수는 몇 월인지와 해당 연도가 윤년인지 여부에 따라 달라집니다. 또한 월을 시작하는 요일은 월과 연도에 따라 달라집니다. *IsLeapYear* 함수를 사용하여 해당 연도가 윤년인지 여부를 확인합니다. SysUtils 헤더 파일의 *MonthDays* 배열을 사용하여 월의 일 수를 구합니다.

윤년과 월별 일 수에 대한 정보가 있다면 각 날짜가 입력되는 그리드 내에서의 위치를 계산할 수 있습니다. 계산은 월이 시작하는 요일을 기준으로 계산합니다.

데이터를 입력하는 각 셀에 대해 달을 오프셋하는 (month-offset) 숫자가 필요하기 때문에 월이나 연도를 변경할 때 한 번 계산하고 매번 이를 참조하는 것이 좋습니다. 클래스 데이터 구성원에 값을 저장한 다음 날짜를 변경할 때마다 이 데이터 구성원을 업데이트할 수 있습니다.

다음과 같은 방법으로 해당 셀에 일을 채웁니다.

1 데이터 구성원 값을 업데이트하는 메소드와 클래스에 달을 오프셋하는 오프셋 데이터 구성원을 추가합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int FMonthOffset;                          // storage for the offset
    f
protected:
    virtual void __fastcall UpdateCalendar(void);
    f
};

void __fastcall TSampleCalendar::UpdateCalendar(void)
{
```

```
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;                          // date of first day of
the month
  if ((int)FDate != 0)                          // only calculate offset
if date is valid
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);  // get elements of date
    FirstDate = TDateTime(AYear, AMonth, 1);   // date of the first
    FMonthOffset = 2 - FirstDate.DayOfWeek();  // generate the offset
into the grid
  }
  Refresh();                                    // always repaint the
control
}
```

2 날짜가 변경될 때마다 새로운 업데이트 메소드를 호출하는 *SetCalendarDate* 및 *SetDateElement* 메소드와 생성자에 다음 명령문을 추가합니다.

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner)
  : TCustomGrid(Owner)
{
  f
  UpdateCalendar();
}

void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                                  // this was already
here
  UpdateCalendar();                               // this previously
called Refresh
}

void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  f
  FDate = TDateTime(AYear, AMonth, ADay);        // this was already here
  UpdateCalendar();                               // this previously
called Refresh
}
```

3 셀의 행과 열 좌표를 전달할 때 일 수를 반환하는 메소드를 달력에 추가합니다.

```
int __fastcall TSampleCalendar::DayNum(int ACol, int ARow)
{
  int result = FMonthOffset + ACol + (ARow - 1) * 7;       // calculate
day for this cell
  if ((result < 1)||(result > MonthDays[IsLeapYear(Year)][Month]))
    result = -1;   // return -1 if invalid
  return result;
}
```

*DayNum*의 선언을 컴포넌트의 타입 선언에 추가해야 합니다.

4 이제 날짜가 입력되는 위치를 계산할 수 있으므로 *DrawCell*을 업데이트하여 날짜를 입력할 수 있습니다.

```
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const TRect
&ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0)                                    // this is the header
row
    TheText = ShortDayNames[ACol + 1];              // just use the day
name
  else
  {
    TheText = "";                                   // blank cell is the
default
    TempDay = DayNum(ACol, ARow);                   // get number for
this cell
    if (TempDay != -1) TheText = IntToStr(TempDay); // use the number if
valid
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) /
2, TheText);
}
```

달력 컴포넌트를 다시 설치하여 폼에 두면 현재 월에 대한 적절한 정보가 나타납니다.

현재 날짜 선택

이제 달력 셀에 숫자가 있으므로 현재 날짜가 있는 셀로 선택하여 강조 표시할 대상을 이동하는 것이 바람직합니다. 디폴트로, 선택은 맨 위의 왼쪽 셀에서 시작하므로 달력을 처음 만들 때와 날짜를 변경할 때 *Row* 및 *Column* 속성을 설정해야 합니다.

현재 날짜에 대한 선택을 설정하려면 *Refresh*를 호출하기 전에 *Row* 및 *Column*을 설정하도록 *UpdateCalendar* 메소드를 변경합니다.

```
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int) FDate != 0)
  {
  f                                             // existing statements to
set FMonthOffset
    Row = (ADay - FMonthOffset) / 7 + 1;
    Col = (ADay - FMonthOffset) % 7;
  }
  Refresh();                                    // this is already here
}
```

날짜를 디코딩하여 이전에 설정된 *ADay* 변수를 재사용하고 있다는 것에 유의합니다.

연도와 월 탐색

속성은 특히 디자인 타임에 컴포넌트를 조작하는 데 유용합니다. 그러나 일부 조작 타입은 둘 이상의 속성을 포함하며 매우 일반적이거나 자연스러워서 메소드를 제공하여 조작을 처리하는 것이 바람직한 경우도 있습니다. 이러한 자연스러운 조작의 한 예로 달력의 "다음 달(next month)" 함수가 있습니다. 월의 랩어라운드 및 연도의 증분 처리는 간단하지만 컴포넌트를 사용하는 개발자에게 매우 편리합니다.

일반적인 조작들을 메소드로 캡슐화할 때의 유일한 단점은 메소드가 런타임에만 사용될 수 있다는 것입니다. 하지만 보통 이러한 조작은 반복적으로 수행할 때만 귀찮은 작업이 되고, 디자인 타임에는 아주 드문 일입니다.

달력의 경우 다음과 이전 월과 연도에 대한 다음의 네 가지 메소드를 추가합니다. 이러한 메소드마다 약간씩 다른 방법으로 *IncMonth* 함수를 사용하여 월 또는 연도의 증분에 의해 *CalendarDate*를 증가시키거나 감소시킵니다.

```
void __fastcall TSampleCalendar::NextMonth()
{
  CalendarDate = IncMonth(CalendarDate, 1);
}
void __fastcall TSampleCalendar::PrevMonth()
{
  CalendarDate = IncMonth(CalendarDate, -1);
}
void __fastcall TSampleCalendar::NextYear()
{
  CalendarDate = IncMonth(CalendarDate, 12);
}

void __fastcall TSampleCalendar::PrevYear()
{
  CalendarDate = IncMonth(CalendarDate, -12);
}
```

클래스 선언에 새 메소드의 선언을 추가해야 합니다.

이제 달력 컴포넌트를 사용하는 애플리케이션을 생성할 때 월 또는 연도를 검색하는 기능을 쉽게 구현할 수 있습니다.

일 탐색

주어진 월에서 일(day)을 확실하게 탐색하는 방법에는 두 가지가 있습니다. 한 가지 방법은 화살표 키를 사용하는 것이고 또다른 방법은 마우스 클릭에 응답하는 것입니다. 표준 그리드 컴포넌트는 두 가지 모두 클릭처럼 처리합니다. 즉, 화살표 이동은 인접한 셀을 클릭하는 것처럼 처리됩니다.

일 탐색 과정은 다음과 같습니다.

- 선택 항목 이동
- OnChange 이벤트 제공
- 비어 있는 셀 제외

선택 항목 이동

그리드의 상속된 동작은 화살표 키나 클릭에 대한 반응으로 선택이 이동되는 것을 처리하지만, 선택된 일을 변경할 경우 그 디폴트 동작을 수정해야 합니다.

달력에서의 이동을 처리하려면 그리드의 *Click* 메소드를 오버라이드합니다.

사용자 상호 작용과 연결된 *Click* 같은 메소드를 오버라이드할 때 표준 동작을 잃지 않도록 상속된 메소드에 대한 호출을 거의 항상 포함시킵니다.

다음은 달력 그리드의 오버라이드된 *Click* 메소드입니다. *Click* 선언을 *TSampleCalendar*에 추가해야 합니다.

```
void __fastcall TSampleCalendar::Click()
{
  int TempDay = DayNum(Col, Row);            // get the day number for
the clicked cell
  if (TempDay != -1) Day = TempDay;          // change day if valid
}
```

OnChange 이벤트 제공

달력 사용자가 달력의 날짜를 변경할 수 있기 때문에 애플리케이션이 이러한 변경 사항에 반응할 수 있도록 하는 것이 바람직합니다.

*TSampleCalendar*에 *OnChange* 이벤트를 추가합니다.

1 이벤트, 이벤트를 저장할 데이터 구성원 및 이벤트를 호출할 가상 메소드를 선언합니다.

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TNotifyEvent FOnChange;
    f
protected:
    virtual void __fastcall Change();
__published:
    __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
    f
}
```

2 *Change* 메소드를 작성합니다.

```
void __fastcall TSampleCalendar::Change()
{
  if(FOnChange != NULL) FOnChange(this);
}
```

3 *SetCalendarDate* 및 *SetDateElement* 메소드의 끝에 *Change*를 호출하는 명령문을 추가합니다.

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
  Change();              // this is the only new
statement
}

void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  f   // many statements setting element values
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
  Change();              // this is new
}
```

이제 달력 컴포넌트를 사용하는 애플리케이션은 *OnChange* 이벤트에 핸들러를 연결함으로써 컴포넌트 날짜의 변경 사항에 응답할 수 있습니다.

비어 있는 셀 제외

달력을 작성할 때 사용자가 비어 있는 셀을 선택할 수 있지만 날짜는 변경되지 않습니다. 그러므로 비어 있는 셀의 선택을 허용하지 않는 것이 바람직합니다.

지정한 셀의 선택 가능 여부를 제어하려면 그리드의 *SelectCell* 메소드를 오버라이드하십시오.

*SelectCell*은 열과 행을 매개변수로 취하고, 지정한 셀의 선택 가능 여부를 나타내는 부울 값을 반환하는 함수입니다.

셀에 유효한 날짜가 없으면 *SelectCell*을 오버라이드하여 **false**를 반환할 수 있습니다.

```
bool __fastcall TSampleCalendar::SelectCell(int ACol, int ARow)
{
  if (DayNum(ACol,ARow) == -1) return false;          // -1 indicates
invalid date
  else return TCustomGrid::SelectCell(ACol, ARow);    // otherwise, use
inherited value
}
```

이제 사용자가 비어 있는 셀을 클릭하거나 화살표 키를 사용하여 비어 있는 셀로 이동하려고 하면 달력에 현재 셀이 선택된 상태로 남겨집니다.

장

56

데이터 인식 컨트롤 만들기

데이터베이스 연결을 사용할 때 *데이터 인식* 컨트롤이 있으면 편리합니다. 즉, 애플리케이션에서 컨트롤과 데이터베이스의 일부를 연결할 수 있습니다. C++Builder에는 데이터 인식 레이블, 에디트 박스, 리스트 박스, 콤보 박스, 조회 컨트롤 및 그리드가 있습니다. 또한 데이터 인식 컨트롤을 직접 만들 수도 있습니다. 데이터 인식 컨트롤 사용에 대한 자세한 내용은 19장, "데이터 컨트롤 사용"을 참조하십시오.

데이터 인식 컨트롤에는 여러 등급이 있습니다. 가장 간단한 단계는 읽기 전용 데이터 인식 컨트롤, 즉 데이터베이스의 현재 상태를 살펴볼 수 있는 *데이터 찾아보기* 기능입니다. 보다 복잡한 단계로는 편집 가능한 데이터 인식 컨트롤, 즉 사용자가 컨트롤을 조작하여 데이터베이스의 값을 편집할 수 있는 *데이터 편집* 기능입니다. 데이터베이스 관련 등급은 가장 간단한 경우인 단일 필드와의 연결로부터 가장 복잡한 경우인 다중 레코드 컨트롤에 이르기까지 다양합니다.

이 장에서는 먼저 데이터셋의 단일 필드와 연결하는 읽기 전용 컨트롤을 만드는 가장 간단한 방법을 설명합니다. 사용된 컨트롤은 55장, "그리드 사용자 정의"에서 만든 *TSampleCalendar* 달력입니다. 컴포넌트 팔레트의 Samples 페이지에 있는 표준 달력 컨트롤인 *TCCalendar*(VCL만 해당)를 사용할 수도 있습니다.

그리고 나서 같은 장에서 새로운 데이터 찾아보기 컨트롤을 데이터 편집 컨트롤로 만드는 방법을 설명합니다.

데이터 찾아보기 컨트롤 생성

데이터 인식 달력 컨트롤 생성은 읽기 전용 컨트롤인지 또는 사용자가 데이터셋에서 원본으로 사용하는 데이터를 변경할 수 있는 컨트롤인지 여부에 관계 없이 다음과 같은 단계로 이루어집니다.

- 컴포넌트 생성 및 등록
- 데이터 연결 추가
- 데이터 변경 내용에 응답

컴포넌트 생성 및 등록

모든 컴포넌트를 같은 방법으로 만듭니다. 컴포넌트 클래스를 파생시켜 해당 컴포넌트의 .CPP 및 .H 파일을 저장하고, 컴포넌트 클래스를 파생시켜 해당 컴포넌트 클래스를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이 프로세스는 45-8페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같은 특성으로 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- *TSampleCalendar* 컴포넌트의 자손인 *TDBCalendar*라는 새 컴포넌트 클래스를 파생시킵니다. *TSampleCalendar* 컴포넌트를 만드는 방법에 대해서는 55장, "그리드 사용자 정의"를 참조하십시오.
- 헤더 파일의 이름을 DBCAL.H로, .CPP 파일의 이름을 DBCAL.CPP로 지정합니다.
- 컴포넌트 팔레트의 Samples 페이지나 CLX 애플리케이션의 다른 페이지에서 *TDBCalendar*를 등록합니다.

CLX CLX 애플리케이션에서는 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX의 `<clx\qcontrols.hpp>`입니다.

결과 헤더 파일은 다음과 같아야 합니다.

```
#ifndef DBCalH
#define DBCalH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
#include <vcl\grids.hpp>              // include the Grids header
#include “calsamp.h”                 // include the header that declares
TSampleCalendar
//---------------------------------------------------------------------------
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

.CPP 파일은 다음과 같아야 합니다.

```
#pragma link “Calsamp”                        // link in TSampleCalendar

#include <vcl\vcl.h>
#pragma hdrstop
#include "DBCal.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
```

```
static inline TDBCalendar *ValidCtrCheck()
{
   return new TDBCalendar(NULL);
}
//-----------------------------------------------------------------------
namespace Dbcal
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TDBCalendar)};
    RegisterComponents("Samples", classes, 0); //Use different page in
CLX applications
  }
}
```

참고 컴포넌트 마법사를 사용하여 *TDBCalendar* 컴포넌트를 시작하면 헤더 파일이 미리 선언된 생성자를 갖게 되고 .CPP 파일에 생성자의 정의가 저장됩니다.

이제 새 달력을 데이터 찾아보기로 만드는 작업을 진행할 수 있습니다.

읽기 전용 컨트롤 만들기

이 데이터 달력은 데이터에 대해 읽기 전용이어서 컨트롤 자체가 읽기 전용이어야 하므로 사용자는 컨트롤 내에서 변경할 수 없고 따라서 데이터베이스에 변경 내용이 반영되지 않습니다.

읽기 전용 달력은 다음과 같이 만듭니다.

- ReadOnly 속성 추가
- 필요한 업데이트 허용

VCL *TSampleCalendar* 대신 C++Builder 의 Samples 페이지에서 *TCalendar* 컴포넌트를 시작한 경우에는 이미 *ReadOnly* 속성이 있으므로 이 단계를 생략할 수 있습니다.

ReadOnly 속성 추가

ReadOnly 속성을 추가하면 디자인 타임에 읽기 전용 컨트롤을 만들 수 있습니다. 이 속성을 **true**로 설정하면 컨트롤의 모든 셀을 선택할 수 없게 할 수 있습니다.

1 속성 선언과 **private** 데이터 멤버를 추가하여 DBCAL.H 파일에 값을 저장합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    bool FReadOnly;                                    // field for
internal storage
protected:
public:
    virtual __fastcall TDBCalendar(TComponent* Owner);
__published:
    __property ReadOnly = {read=FReadOnly, write=FReadOnly,
default=true};
};
```

2 DBCAL.CPP에서 생성자를 작성합니다.

```
virtual __fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
  TSampleCalendar(Owner)
{
  FReadOnly = true;                                    // sets the default
value
}
```

3 컨트롤이 읽기 전용인 경우 선택을 허용하지 않도록 *SelectCell* 메소드를 오버라이드합니다. *SelectCell* 사용에 대해서는 55-14페이지의 "비어 있는 셀 제외"에서 설명합니다.

```
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
  if (FReadOnly) return false;                          // can't select if
read only
  return TSampleCalendar::SelectCell(ACol, ARow);      // otherwise, use
inherited method
}
```

SelectCell 선언을 *TDBCalendar*의 클래스 선언에 추가해야 합니다.

이제 달력을 폼에 추가하면 컴포넌트에서 클릭과 키스트로크를 무시합니다. 또한 날짜를 변경해도 선택 위치가 업데이트되지 않습니다.

필요한 업데이트 허용

읽기 전용 달력은 *Row* 및 *Col* 속성 설정을 포함하여 모든 종류의 변경에 대해 *SelectCell* 메소드를 사용합니다. *UpdateCalendar* 메소드는 날짜가 변경될 때마다 *Row* 및 *Col*을 설정하지만 *SelectCell*은 변경을 허용하지 않기 때문에 날짜가 변경되더라도 선택은 그대로 유지됩니다.

경우에 따라 변경할 수 있게 하려면 내부 부울 플래그를 달력에 추가하여 플래그가 **true**로 설정되었을 때 변경을 허용하도록 할 수 있습니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    f
    bool FUpdating;                                    // private flag for
internal use
protected:
    virtual bool __fastcall SelectCell(long ACol, long ARow);
public:
    f
    virtual void __fastcall UpdateCalendar();
    f
};
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
    if (!FUpdating && FReadOnly) return false;          // can't select
if read only
    return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use
inherited method
}
```

```
void __fastcall TDBCalendar::UpdateCalendar()
{
  FUpdating=true;                                          // set flag to
allow updates
  try
  {
     TSampleCalendar::UpdateCalendar();                   // update as usual
  }
  catch(...)
  {
    FUpdating = false;
    throw;
  }
  FUpdating = false;                                       // always clear the
flag
}
```

사용자는 여전히 달력을 변경할 수 없지만 이제 날짜 속성을 변경하여 해당 날짜의 변경 내용이 제대로 반영됩니다. 이제 진정한 의미의 읽기 전용 달력 컨트롤이 있으므로 데이터 찾아보기 기능을 추가할 수 있습니다.

데이터 연결 추가

컨트롤과 데이터베이스의 연결은 *데이터 연결*라는 클래스에 의해 처리됩니다. 컨트롤과 데이터베이스의 단일 데이터 멤버를 연결하는 데이터 연결은 *TFieldDataLink*입니다. 전체 테이블에 대한 데이터 연결도 있습니다.

데이터 인식 컨트롤은 데이터 연결 클래스를 *소유*합니다. 즉 컨트롤이 데이터 연결의 생성과 소멸을 담당합니다. 소유된 클래스 관리에 대한 자세한 내용은 54장,"그래픽 컴포넌트 생성"을 참조하십시오.

데이터 연결을 소유된 클래스로 만드는 데에는 다음 세 단계가 필요합니다.

1 클래스 데이터 멤버를 선언합니다.

2 액세스 속성 선언

3 데이터 연결 초기화

데이터 멤버 선언

54-6페이지의 "데이터 멤버 선언"에서 설명한 것처럼 컴포넌트에는 각 소유된 클래스마다 데이터 멤버가 필요합니다. 이 경우 달력은 해당 데이터 연결에 대해 *TFieldDataLink* 타입의 데이터 멤버가 필요합니다.

달력의 데이터 연결에 대한 필드를 선언합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    TFieldDataLink *FDataLink;
    f
};
```

애플리케이션을 컴파일하려면 DB.HPP 및 DBTABLES.HPP 파일을 DBCAL.H 파일에 포함시켜야 합니다.

```
#include <DB.hpp>
#include <DBTables.hpp>
```

액세스 속성 선언

모든 데이터 인식 컨트롤에는 데이터를 컨트롤에 제공하는 애플리케이션의 데이터 소스 클래스를 지정하는 *DataSource* 속성이 있습니다. 또한 단일 필드에 액세스하는 컨트롤에는 데이터 소스에서 해당 필드를 지정하기 위한 *DataField* 속성이 필요합니다.

54장, "그래픽 컴포넌트 생성" 예제의 소유된 클래스에 대한 액세스 속성과 달리, 이러한 액세스 속성은 소유된 클래스 자체에 대한 액세스가 아니라 소유된 클래스 내의 해당 속성에 대한 액세스를 제공합니다. 즉, 컨트롤과 그 데이터 연결이 동일한 데이터 소스와 필드를 공유할 수 있는 속성을 만듭니다.

DataSource 및 *DataField* 속성과 해당 구현 메소드를 선언한 다음, 해당 메소드를 데이터 연결 클래스의 해당 속성에 대한 "통과(pass-through)" 메소드로 작성합니다.

액세스 속성 선언의 예제

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
   f
    AnsiString __fastcall GetDataField();             // methods are
private
    TDataSource *__fastcall GetDataSource();          // returns name of
data field
    void __fastcall SetDataField(AnsiString Value); // returns reference
to data                                                          //
source
    void __fastcall SetDataSource(TDataSource *Value);    // assigns name
of data field
    f
__published:                                  // make properties available
at design time
    __property AnsiString DataField = {read=GetDataField,
write=SetDataField, nodefault};
    __property TDataSource * DataSource = {read=GetDataSource,
write=SetDataSource,       nodefault};
   f
};
AnsiString __fastcall TDBCalendar::GetDataField()
{
  return FDataLink->FieldName;
}
TDataSource *__fastcall TDBCalendar::GetDataSource()
{
  return FDataLink->DataSource;
}
```

```
void __fastcall TDBCalendar::SetDataField(AnsiString Value)
{
  FDataLink->FieldName = Value;
}
void __fastcall TDBCalendar::SetDataSource(TDataSource *Value)
{
  if(Value != NULL)
    Value->FreeNotification(this);
  FDataLink->DataSource = Value;
}
```

이제 달력과 데이터 연결 사이에 연결이 이루어졌으므로 한 가지 중요한 단계가 남았습니다. 달력 컨트롤이 생성되면 데이터 연결 클래스를 생성하고 달력을 소멸하기 전에 데이터 연결을 소멸해야 합니다.

데이터 연결 초기화

데이터 인식 컨트롤은 존재하는 동안 계속해서 데이터 연결에 대한 액세스를 필요로 하므로 데이터 연결 객체를 각각의 생성자의 일부로 생성하고 자신이 제거되기 전에 데이터 연결 객체를 제거해야 합니다.

달력의 생성자와 소멸자를 오버라이드합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
public:
    virtual __fastcall TDBCalendar(TComponent *Owner);
    __fastcall ~TDBCalendar();
};
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
TSampleCalendar(Owner)
{
    FReadOnly = true;
    FDataLink = new TFieldDataLink();
    FDataLink->Control = this;
}
__fastcall TDBCalendar::~TDBCalendar()
{
    FDataLink->Control = NULL;
    FDataLink->OnUpdateData = NULL;
    delete FDataLink;
}
```

이제 완벽한 데이터 연결이 갖추어졌지만 연결된 필드에서 읽어야 하는 데이터를 아직 컨트롤에 알리지 않았습니다. 다음 단원에서는 이 작업을 수행하는 방법을 설명합니다.

데이터 변경 내용에 응답

컨트롤에 데이터 소스와 데이터 필드를 지정할 데이터 연결 및 속성을 지정했으면 다른 레코드로의 이동이나 해당 필드의 변경 등으로 인한 필드의 데이터 변경에 대해 응답해야 합니다.

데이터 연결 클래스에는 모두 *OnDataChange*라는 이벤트가 있습니다. 데이터 소스에서 데이터가 변경되었음을 알리면 데이터 연결 객체가 해당 *OnDataChange* 이벤트에 연결된 이벤트 핸들러를 호출합니다.

데이터 변경에 대한 응답으로 컨트롤을 업데이트하려면 데이터 연결의 *OnDataChange* 이벤트에 핸들러를 연결합니다.

이 경우 달력에 메소드를 추가한 다음 해당 메소드를 데이터 연결의 *OnDataChange*에 대한 핸들러로 지정합니다.

DataChange 메소드를 선언하고 구현한 다음 생성자에서 이 메소드를 데이터 연결의 *OnDataChange* 이벤트에 할당합니다. 소멸자에서 객체를 소멸하기 전에 *OnDataChange* 핸들러를 분리합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall DataChange(TObject *Sender);
    f
};
```

```
void __fastcall TDBCalendar::DataChange( TObject* Sender)
{
  if (FDataLink->Field == NULL)                          // if no field
is assigned ...
  CalendarDate = 0;                                      // ...set to
invalid date
  else CalendarDate = FDataLink->Field->AsDateTime;  // otherwise, set to
new data
}
```

```
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
 TSampleCalendar(AOwner)
{
  FReadOnly = true;
  FDataLink = new TFieldDataLink();             // construct the datalink
object
  FDataLink->Control = this;
  FDataLink->OnDataChange = DataChange;         // attach the handler
}
```

```
__fastcall TDBCalendar::~TDBCalendar()
{
  FDataLink->Control = NULL;
  FDataLink->OnUpdateData = NULL;
  FDataLink->OnDataChange = NULL;                 // detach the handler
before...
  delete FDataLink;                               // ...destroying the
datalink object
}
```

이제 데이터 찾아보기 컨트롤이 만들어졌습니다.

데이터 편집 컨트롤 생성

데이터 편집 컨트롤을 만들 때는 컴포넌트를 생성하고 등록한 다음 데이터 찾아보기 컨트롤에서처럼 데이터 연결을 추가합니다. 또한 원본으로 사용하는 필드의 데이터 변경 내용에도 비슷한 방법으로 응답하지만 몇 가지 문제를 더 처리해야 합니다.

예를 들어, 컨트롤이 키와 마우스 이벤트에 모두 응답하게 하고자 합니다. 컨트롤은 사용자가 컨트롤의 내용을 변경할 때 응답해야 합니다. 사용자가 컨트롤을 종료하면 컨트롤의 변경 내용이 데이터셋에 반영되게 하고 싶습니다.

여기에서 설명되는 데이터 편집 컨트롤은 이 장의 처음 부분에 설명된 컨트롤과 동일한 달력 컨트롤입니다. 컨트롤은 연결 필드에서 데이터를 보고 편집할 수 있도록 수정됩니다.

기존 컨트롤을 수정하여 데이터 편집 컨트롤로 만들려면 다음 단계가 필요합니다.

- FReadOnly의 기본값 변경
- 마우스 다운(mouse-down) 및 키 다운(key-down) 이벤트 처리
- 필드 데이터 연결 클래스 업데이트
- Change 메소드 수정
- 데이터셋 업데이트

FReadOnly의 기본값 변경

이 컨트롤은 데이터 편집 컨트롤이므로 *ReadOnly* 속성이 디폴트로 **false**로 설정되어야 합니다. *ReadOnly* 속성을 **false**로 만들려면 생성자에서 *FReadOnly*의 값을 변경합니다.

```
__fastcall TDBCalendar::TDBCalendar (TComponent* Owner) :
TSampleCalendar(Owner)
{
  FReadOnly = false;             // set the default value
  f
}
```

마우스 다운(mouse-down) 및 키 다운(key-down) 이벤트 처리

컨트롤 사용자가 컨트롤과 상호 작용하기 시작하면 컨트롤에서 Windows로부터 마우스 다운(mouse-down) 메시지(*WM_LBUTTONDOWN, WM_MBUTTONDOWN* 또는 *WM_RBUTTONDOWN*) 또는 키 다운(key-down) 메시지(*WM_KEYDOWN*)를 받습니다. 컨트롤이 이러한 메시지에 응답하도록 하려면 이러한 메시지에 응답하는 핸들러를 작성해야 합니다.

- 마우스 다운(mouse-down) 이벤트에 응답
- 키 다운(key-down) 이벤트에 응답

CLX CLX를 사용하는 경우에는 운영 체제로부터의 알림이 시스템 이벤트 형태로 전달됩니다. 시스템과 widget 이벤트에 응답하는 컴포넌트 작성에 대한 자세한 내용은 51-10페이지의 "CLX를 사용하여 시스템 통지에 응답"을 참조하십시오.

마우스 다운(mouse-down) 이벤트에 응답

MouseDown 메소드는 컨트롤의 *OnMouseDown* 이벤트에 대한 protected 메소드입니다. 컨트롤은 Windows 마우스 다운(mouse-down) 메시지에 대한 응답으로 *MouseDown*을 호출합니다. 상속된 *MouseDown* 메소드를 오버라이드하면 *OnMouseDown* 이벤트를 호출하는 것은 물론 기타 응답을 제공하는 코드를 포함시킬 수 있습니다.

*MouseDown*을 오버라이드하려면 *MouseDown* 메소드를 *TDBCalendar* 클래스에 추가합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
f
protected:
    virtual void __fastcall MouseDown(TMouseButton Button,  TShiftState
Shift, int X,       int Y);
    f
};
```

.CPP 파일에 *MouseDown* 메소드를 작성합니다.

```
void __fastcall TDBCalendar::MouseDown(TMouseButton Button,  TShiftState
Shift, int  X,  int Y)
{
  TMouseEvent MyMouseDown;                                  // declare event
type
  if (!FReadOnly && FDataLink->Edit())                      // if the field
can be edited
    TSampleCalendar::MouseDown(Button, Shift, X, Y);   // call the
inherited MouseDown
  else
  {
    MyMouseDown = OnMouseDown;                              // assign
OnMouseDown event
    if (MyMouseDown != NULL) MyMouseDown(this, Button,  // execute code
in the...
       Shift, X, Y);                                        // ...OnMouseDown
event handler
  }
}
```

*MouseDown*이 마우스 다운(mouse-down) 메시지에 응답할 때 상속된 *MouseDown* 메소드는 컨트롤의 *ReadOnly* 속성이 **false**이고 데이터 연결 객체가 필드를 편집할 수 있는 편집 모드에 있는 경우에만 호출됩니다. 필드를 편집할 수 없으면 프로그래머가 *OnMouseDown* 이벤트 핸들러에 입력한 코드가 실행됩니다.

키 다운(key-down) 이벤트에 응답

KeyDown 메소드는 컨트롤의 *OnKeyDown* 이벤트에 대한 protected 메소드입니다. 컨트롤은 Windows 키 다운(key-down) 메시지에 대한 응답으로 *KeyDown*을 호출합니다. 상속된 *KeyDown* 메소드를 오버라이드할 때 *OnKeyDown* 이벤트를 호출하는 것은 물론 기타 응답을 제공하는 코드를 포함시킬 수 있습니다.

*KeyDown*을 오버라이드하려면 다음 단계를 따르십시오.

1 *KeyDown* 메소드를 *TDBCalendar* 클래스에 추가합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
   f
protected:
    virtual void __fastcall KeyDown(unsigned short &Key,  TShiftState
Shift);
   f
};
```

2 .CPP 파일에 *KeyDown* 메소드를 작성합니다.

```
void __fastcall TDBCalendar::KeyDown(unsigned short &Key,  TShiftState
Shift)
{
  TKeyEvent MyKeyDown;                                  // declare event
type
  Set<unsigned short,0,8> keySet;
  keySet = keySet << VK_UP << VK_DOWN << VK_LEFT      // assign virtual
keys to set
    << VK_RIGHT << VK_END << VK_HOME << VK_PRIOR <<
VK_NEXT;
  if (!FReadOnly &&                                 // if control is not
read only...
      (keySet.Contains(Key)) &&                      // ...and key is in the
set...
      FDataLink->Edit() )                            // ...and field is in
edit mode
  {
    TCustomGrid::KeyDown(Key, Shift);          // call the inherited
KeyDown method
  }
  else
  {
    MyKeyDown = OnKeyDown;                              // assign OnKeyDown
event
    if (MyKeyDown != NULL) MyKeyDown(this,Key,Shift); // execute code
in...
  }                                                        // ...OnKeyDown
event handler
}
```

*KeyDown*이 마우스 다운(mouse-down) 이벤트에 응답할 때 상속된 *KeyDown* 메소드는 컨트롤의 *ReadOnly* 속성이 **false**이고 눌려진 키가 커서 컨트롤 키 중 하나이며 데이터 연결 객체가 필드를 편집할 수 있는 편집 모드에 있는 경우에만 호출됩니다. 필드를 편집할 수 없거나 다른 키가 눌려진 경우에는 프로그래머가 *OnKeyDown* 이벤트 핸들러에 입력한 코드(있는 경우)가 실행됩니다.

필드 데이터 연결 클래스 업데이트

데이터 변경에는 두 가지 유형이 있습니다.

- 데이터 인식 컨트롤에 반영되어야 하는 필드 값 변경
- 필드 값에 반영되어야 하는 데이터 인식 컨트롤 변경

TDBCalendar 컴포넌트에는 *CalendarDate* 속성에 값을 할당하여 데이터셋의 필드 값 변경을 처리하는 *DataChange* 메소드가 이미 있습니다. *DataChange* 메소드는 *OnData Change* 이벤트의 핸들러입니다. 그러므로 달력 컴포넌트는 데이터 변경의 첫 번째 유형을 처리할 수 있습니다.

마찬가지로 필드 데이터 연결 클래스에는 또한 컨트롤 사용자가 데이터 인식 컨트롤의 내용을 수정할 때 발생하는 *OnUpdateData* 이벤트가 있습니다. 달력 컨트롤에는 *OnUpdateData* 이벤트에 대한 이벤트 핸들러가 되는 *UpdateData* 메소드가 있습니다. *UpdateData*는 데이터 인식 컨트롤의 변경된 값을 필드 데이터 연결에 할당합니다.

1 달력의 값을 변경한 내용을 필드 값에 반영하려면 *UpdateData* 메소드를 달력 컴포넌트의 private 섹션에 추가합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall UpdateData(TObject *Sender);
};
```

2 .CPP 파일에 *UpdateData* 메소드를 작성합니다.

```
void __fastcall TDBCalendar::UpdateData( TObject* Sender)
{
    FDataLink->Field->AsDateTime = CalendarDate;     // set field link to
calendar date
}
```

3 *TDBCalendar*의 생성자에서 *UpdateData* 메소드를 *OnUpdateData* 이벤트에 할당합니다.

```
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
  : TSampleCalendar(Owner)
{
  FDataLink = new TFieldDataLink();       // this was already here
  FDataLink->OnDataChange = DataChange;   // this was here too
  FDataLink->OnUpdateData = UpdateData;   // assign UpdateData to the
OnUpdateData event
}
```

Change 메소드 수정

*TDBCalendar*의 *Change* 메소드는 새로운 날짜 값이 설정될 때마다 호출됩니다. *Change*는 *OnChange* 이벤트 핸들러(있는 경우)를 호출합니다. 컴포넌트 사용자는 *OnChange* 이벤트 핸들러에 코드를 작성하여 날짜 변경에 대해 응답할 수 있습니다.

달력의 날짜가 변경되면 원본으로 사용하는 데이터셋에 변경이 발생했음을 알려야 합니다. *Change* 메소드를 오버라이드하고 코드를 한 줄 더 추가하여 이 작업을 수행할 수 있습니다. 다음 단계를 따르십시오.

1 새로운 *Change* 메소드를 *TDBCalendar* 컴포넌트에 추가합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
protected:
    virtual void __fastcall Change();
    f
};
```

2 데이터셋에 데이터가 변경된 것을 알리는 *Modified* 메소드를 호출하여 *Change* 메소드를 작성한 다음 상속된 *Change* 메소드를 호출합니다.

```
void __fastcall TDBCalendar::Change()
{
  if (FDataLink != NULL)
    FDataLink->Modified();                // call the Modified method
  TSampleCalendar::Change();              // call the inherited Change
method
}
```

데이터셋 업데이트

지금까지 데이터 인식 컨트롤 내에서 변경하면 필드 데이터 연결 클래스의 값이 변경되었습니다. 데이터 편집 컨트롤을 생성하는 마지막 단계는 새 값으로 데이터셋을 업데이트하는 것입니다. 이 작업은 데이터 인식 컨트롤의 값을 변경한 사람이 컨트롤 밖을 클릭하거나 *Tab* 키를 눌러 컨트롤을 종료한 후 발생합니다. 이 프로세스는 VCL와 CLX에서 다르게 일어납니다.

VCL VCL에는 컨트롤 작업을 위해 정의된 메시지 컨트롤 ID가 있습니다. 예를 들어, 사용자가 컨트롤을 종료하면 *CM_EXIT* 메시지가 컨트롤로 전달됩니다. 메시지에 응답하는 메시지 핸들러를 작성할 수 있습니다. 이 경우 사용자가 컨트롤을 종료하면 *CM_EXIT*의 메시지 핸들러인 *CMExit* 메소드가 필드 데이터 연결 클래스의 변경된 값을 사용하여 데이터셋의 레코드를 업데이트하여 응답합니다. 메시지 핸들러에 대한 자세한 내용은 51장, "메시지 및 시스템 통지 처리"를 참조하십시오.

메시지 핸들러 내의 데이터셋을 업데이트하려면 다음 단계를 따르십시오.

1 메시지 핸들러를 *TDBCalendar* 컴포넌트에 추가합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall CMExit(TWMNoParams Message);

BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP
};
```

2 .CPP 파일에 다음과 같은 코드를 작성합니다.

```
void __fastcall TDBCalendar::CMExit(TWMNoParams &Message)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update
database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don’t let focus
leave
    throw;
  }
}
```

CLX CLX에서는 *TWidgetControl*에 입력 포커스가 컨트롤에서 벗어났을 때 호출되는 protected *DoExit* 메소드가 있습니다. 이 메소드는 *OnExit* 이벤트에 대한 이벤트 핸들러를 호출합니다. 이 메소드를 오버라이드하여 *OnExit* 이벤트 핸들러를 생성하기 전에 데이터셋의 레코드를 업데이트할 수 있습니다.

사용자가 컨트롤을 종료할 때 데이터셋을 업데이트하려면 다음 단계를 따르십시오.

1 *DoExit* 메소드에 대한 오버라이드를 *TDBCalendar* 컴포넌트에 추가합니다.

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    DYNAMIC void __fastcall DoExit(void);
   f
};
```

2 .CPP 파일에 다음과 같은 코드를 작성합니다.

```
void __fastcall TDBCalendar::DoExit(void)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update
database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don’t let focus
leave
    throw;
  }
  TCustomGrid::DoExit(); // let the inherited method generate an OnExit
event
}
```

장

57

다이얼로그 박스를 컴포넌트로 만들기

자주 사용하는 다이얼로그 박스를 컴포넌트로 만들어서 컴포넌트 팔레트에 추가해 두면 편리합니다. 다이얼로그 박스 컴포넌트는 일반적인 표준 다이얼로그 박스를 나타내는 컴포넌트처럼 사용할 수 있습니다. 이 장에서는 사용자가 프로젝트에 추가하고 디자인 타임에 속성을 설정할 수 있는 간단한 컴포넌트를 만드는 것을 목적으로 합니다.

다이얼로그 박스를 컴포넌트로 만들려면 다음 단계를 따르십시오.

1 컴포넌트 인터페이스 정의

2 컴포넌트 생성 및 등록

3 컴포넌트 인터페이스 생성

4 컴포넌트 파생

C++Builder에서 다이얼로그 박스와 관련된 "랩퍼" 컴포넌트는 다이얼로그 박스를 만들어 런타임에 실행하고 사용자가 지정한 데이터를 전달합니다. 다이얼로그 박스 컴포넌트는 다시 사용하고 변경할 수 있습니다.

이 장에서는 C++Builder Object Repository에서 제공한 일반적인 About Box 폼 주위에 랩퍼 컴포넌트를 만드는 방법을 소개합니다.

참고 ABOUT.H, ABOUT.CPP 및 ABOUT.DFM 파일을 작업 디렉토리에 복사합니다. ABOUT.CPP를 프로젝트에 추가하여 다이얼로그 박스 랩퍼 컴포넌트를 생성할 때 ABOUT.OBJ 파일이 생성되도록 합니다.

컴포넌트로 랩핑될 다이얼로그 박스를 디자인할 때는 특별히 고려해야 할 사항이 많지 않습니다. 이 컨텍스트에서는 거의 모든 폼을 다이얼로그 박스로 사용할 수 있습니다.

컴포넌트 인터페이스 정의

다이얼로그 박스를 위한 컴포넌트를 작성하려면 먼저 개발자가 이 컴포넌트를 어떻게 사용하게 할 것인지 결정해야 합니다. 다이얼로그 박스와 이 다이얼로그 박스를 사용하는 애플리케이션 간에 인터페이스를 만듭니다.

예를 들어, 일반적인 다이얼로그 박스 컴포넌트의 속성을 살펴 보십시오. 일반적인 다이얼로그 박스 컴포넌트를 사용하면 개발자가 캡션과 초기 컨트롤 설정 등 다이얼로그 박스의 초기 상태를 설정한 다음 다이얼로그 박스를 닫은 후 필요한 정보를 다시 읽게 할 수 있습니다. 다이얼로그 박스에 있는 각 컨트롤과의 직접적인 상호 작용은 없으며 단지 랩퍼 컴포넌트의 속성과만 작업합니다.

따라서 인터페이스에는 다이얼로그 박스 폼이 개발자가 지정한 방식에 따라 나타나고 애플리케이션에 필요한 모든 정보를 반환할 수 있도록 충분한 정보를 포함해야 합니다. 랩퍼 컴포넌트의 속성은 일시적인 다이얼로그 박스를 위한 영구적 데이터로 생각할 수 있습니다.

About 박스의 경우 정보를 반환할 필요가 없으므로 랩퍼의 속성에 About 박스를 제대로 표시하는 데 필요한 정보만 포함하면 됩니다. About 박스에는 애플리케이션에 의해 영향을 받을 수 있는 네 개의 필드가 있으므로 이를 제공하기 위해 네 개의 문자열 타입 속성을 제공합니다.

컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 동일한 방법으로 시작됩니다. 컴포넌트 클래스를 파생시켜 해당 컴포넌트의 .CPP 및 .H 파일을 저장하고, 컴포넌트 클래스를 파생시켜 해당 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이 과정은 45-8페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같은 특성으로 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- *TComponent*의 자손인 *TAboutBoxDlg*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트의 유닛 파일인 ABOUTDLG.H 및 ABOUTDLG.CPP를 호출합니다.
- 컴포넌트 팔레트의 Sample 페이지에 *TAboutBoxDlg*를 등록합니다.

결과 .HPP 파일은 다음과 같아야 합니다.

```
#ifndef AboutDlgH
#define AboutDlgH
//----------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
//----------------------------------------------------------------------------
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
protected:
public:
__published:
};
//----------------------------------------------------------------------------
#endif
```

유닛의 .CPP 파일은 다음과 같아야 합니다.

```
#include <vcl\vcl.h>
#pragma hdrstop
#include "AboutDlg.h"
```

```
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TAboutBoxDlg *ValidCtrCheck()
{
  return new TAboutBoxDlg(NULL);
}
//---------------------------------------------------------------------------
namespace AboutDlg {
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TAboutBoxDlg)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

참고 컴포넌트 마법사를 사용하여 이 컴포넌트를 시작하면 *TAboutDlg*도 추가된 생성자를 갖게 됩니다.

CLX CLX 애플리케이션에서는 일부 헤더 파일의 이름과 위치가 다릅니다. 예를 들면, `<vcl\controls.hpp>`는 CLX의 `<clx\qcontrols.hpp>`입니다.

현재 새 컴포넌트에는 *TComponent*에 내장된 기능 밖에 없습니다. 가장 간단한 넌비주얼(nonvisual) 컴포넌트입니다. 다음 단원에서는 컴포넌트와 다이얼로그 박스 사이에 인터페이스를 작성합니다.

컴포넌트 인터페이스 생성

다음은 컴포넌트 인터페이스를 생성하는 단계입니다.

1. 폼 유닛 파일 포함
2. 인터페이스 속성 추가
3. Execute 메소드 추가

폼 유닛 파일 포함

랩퍼 컴포넌트가 초기화되고 랩핑된 다이얼로그 박스를 표시하게 하려면 폼의 파일을 프로젝트에 추가해야 합니다.

ABOUT.HPP를 포함시키고 컴포넌트의 헤더 파일에 있는 ABOUT.OBJ에 연결합니다.

```
#include "About.h"
#pragma link "About.obj"
```

폼 헤더 파일은 항상 폼 클래스의 인스턴스를 선언합니다. About 박스의 경우 폼 클래스는 *TAboutBox*이고 ABOUT.H 파일에는 다음이 포함됩니다.

```
extern TAboutBox *AboutBox;
```

인터페이스 속성 추가

진행하기 전에 개발자가 애플리케이션에서 이 다이얼로그 박스를 컴포넌트로 사용하도록 하기 위해 랩퍼에 필요한 속성을 결정합니다. 그런 다음 이러한 속성의 선언을 컴포넌트의 클래스 선언에 추가할 수 있습니다.

랩퍼 컴포넌트의 속성은 일반적인 컴포넌트를 작성할 때 만드는 속성에 비해 약간 간단합니다. 이 경우에는 랩퍼가 다이얼로그 박스와 주고 받을 수 있는 일부 영구 데이터를 만듭니다. 데이터를 속성의 폼에 두면 개발자가 디자인 타임에 데이터를 설정하고 랩퍼가 런타임에 이 데이터를 다이얼로그 박스로 전달하게 할 수 있습니다.

인터페이스 속성을 선언하려면 컴포넌트의 클래스 선언에 두 가지를 추가해야 합니다.

- 랩퍼가 속성 값을 저장하는 데 사용하는 변수인 private 데이터 멤버
- 속성 이름을 지정하고 저장에 사용할 데이터 멤버를 지정하는 published 속성 선언

이러한 종류의 인터페이스 속성은 액세스 메소드를 사용하지 않고 저장된 데이터에 직접 액세스합니다. 규칙상 속성 값을 저장하는 데이터 멤버는 속성과 같은 이름을 갖지만 앞에 *F*자가 추가됩니다. 데이터 멤버와 속성은 *같은 타입이어야 합니다*.

예를 들어, *Year*라는 정수 타입의 인터페이스 속성을 선언하려면 다음과 같이 선언합니다.

```
class PACKAGE TWrapper : public TComponent
{
private:
    int FYear;                                  // data member to hold the
Year-property data
protected:
public:
__published:
    __property int Year = {read=FYear, write=FYear};        // property
matched with storage
};
```

이 About 박스의 경우 각각 제품 이름, 버전 정보, 저작권 정보 및 설명을 위한 네 개의 문자열 타입 속성이 필요합니다. ABOUTDLG.H 파일은 다음과 같아야 합니다.

```
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
    int FYear;
    String FProductName, FVersion, FCopyright, FComments;
protected:
public:
__published:
    __property int Year = {read=FYear, write=FYear};
    __property String ProductName = {read=FProductName,
write=FProductName};
    __property String Version = {read=FVersion, write=FVersion};
    __property String Copyright = {read=FCopyright, write=FCopyright};
    __property String Comments = {read=FComments, write=FComments};
};
```

컴포넌트를 컴포넌트 팔레트에 설치한 다음 폼에 컴포넌트를 추가할 때는 속성을 설정하여 이 값이 자동으로 폼에 저장되게 할 수 있습니다. 그러면 랩퍼가 랩핑된 다이얼로그 박스를 실행할 때 이 값을 사용할 수 있습니다.

Execute 메소드 추가

컴포넌트 인터페이스의 마지막 부분은 다이얼로그 박스를 열었다가 닫을 때 결과를 반환하는 방법입니다. 일반적인 다이얼로그 박스 컴포넌트에서처럼 사용자가 OK를 클릭할 때 **true**를 반환하거나 사용자가 다이얼로그 박스를 취소할 때 **false** 를 반환하는 *Execute* 라는 부울 함수를 사용합니다.

Execute 메소드 선언은 항상 다음과 같습니다.

```
class PACKAGE TMyWrapper : public TComponent
{
  f
public:
    bool __fastcall Execute();
  f
};
```

*Execute*를 최소로 구현하려면 다이얼로그 박스 폼을 생성하여 이 폼을 modal 다이얼로그 박스로 표시한 다음 *ShowModal*의 반환 값에 따라 **true**나 **false**를 반환해야 합니다.

다음은 *TMyDialogBox* 타입의 다이얼로그 박스 폼에 대한 최소 *Execute* 메소드입니다.

```
bool __fastcall TMyWrapper::Execute()
{
  DialogBox = new TMyDialogBox(Application);    // construct the form
    bool Result;
  try
  {
    Result = (DialogBox->ShowModal()  IDOK);  // execute; set result
based on how closed
  }
  catch(...)
  {
    Result = false;                             // if it fails, set
Result to false
  }
  DialogBox->Free();                            // dispose of form
}
```

실제로는 예외 핸들러 내부에 더 많은 코드가 있을 것입니다. 특히 랩퍼는 *ShowModal*을 호출하기 전에 랩퍼 컴포넌트의 인터페이스 속성에 따라 일부 다이얼로그 박스의 속성을 설정할 것이고, *ShowModal*이 반환된 다음 다이얼로그 박스 실행 결과를 기준으로 해당 인터페이스의 속성을 일부 설정할 것입니다.

About 박스의 경우 랩퍼 컴포넌트의 네 가지 인터페이스 속성을 사용하여 About 박스 폼의 레이블 내용을 설정해야 합니다. About 박스는 애플리케이션으로 정보를 반환하지 않으므로 *ShowModal*을 호출한 후에 수행할 작업이 하나도 없습니다. ABOUTDLG.CPP 파일에 다음과 같이 About 박스 랩퍼의 *Execute* 메소드를 작성합니다.

```
bool __fastcall TAboutBoxDlg::Execute()
{
  AboutBox = new TAboutBox(Application);                // construct the
About box
  bool Result;
  try
  {
    if (ProductName == “”)                              // if product
name’s left blank ...
      ProductName = Application->Title;                 // ... use
application title instead
    AboutBox->ProductName->Caption = ProductName;       // copy product
name
    AboutBox->Version->Caption = Version;               // copy version
information
    AboutBox->Copyright->Caption = Copyright;           // copy copyright
information
    AboutBox->Comments->Caption = Comments;             // copy comments
    AboutBox->Caption = “About “ + ProductName;         // set About-box
caption
    Result = (AboutBox->ShowModal() == IDOK);           // execute and
set result
  }
  catch(...)
  {
    Result = false;                       // if it fails, set Result to
false
    f
  }
  AboutBox->Free();                       // dispose of About box
  return Result == IDOK;                  // compare Result to IDOK and
return Boolean value
}
```

ABOUTDLG.H 헤더에서 *TAboutDlg* 클래스의 public 부분에 *Execute* 메소드를 위한 선언을 추가합니다.

```
class PACKAGE TAboutDlg : public TComponent
{
public:
     virtual bool __fastcall Execute();
};
```

컴포넌트 파생

다이얼로그 박스 컴포넌트를 설치했으면 폼에 추가하고 실행하여 일반적인 다이얼로그 박스의 다른 컴포넌트를 사용하는 것처럼 이 컴포넌트를 사용할 수 있습니다. About 박스를 빠르게 테스트하려면 명령 버튼을 폼에 추가한 다음 사용자가 버튼을 클릭할 때 다이얼로그 박스를 실행하게 합니다.

예를 들어, About 다이얼로그 박스를 작성하여 컴포넌트로 만든 다음 컴포넌트 팔레트에 추가한 경우에는 다음 단계를 사용하여 테스트할 수 있습니다.

1. 새 프로젝트를 만듭니다.
2. About 박스 컴포넌트를 메인 폼에 둡니다.
3. 명령 버튼을 폼에 둡니다.
4. 명령 버튼을 더블 클릭하여 비어 있는 클릭 이벤트(click-event) 핸들러를 만듭니다.
5. 클릭 이벤트(click-event) 핸들러에서 다음 코드 행을 입력합니다.

```
AboutBoxDlg1->Execute();
```

6. 애플리케이션을 실행합니다.

메인 폼이 나타나면 명령 버튼을 클릭합니다. About 박스가 디폴트 프로젝트 아이콘과 Project1이라는 이름으로 나타납니다. OK를 선택하여 다이얼로그 박스를 닫습니다.

About 박스 컴포넌트의 여러 가지 속성을 설정하고 다시 애플리케이션을 실행하여 컴포넌트를 더 테스트할 수 있습니다.

장

58

IDE 확장

Open Tools API(Tools API라고도 함)를 사용하여IDE를 확장하고, 고유한 메뉴 항목, 툴바 버튼, 동적인 폼 생성 마법사 등을 포함하도록 사용자 정의할 수 있습니다. Tools API는 메인 메뉴, 툴바, 메인 액션 리스트와 이미지 리스트, 소스 에디터의 내부 버퍼, 키보드 매크로와 연결, 폼과 폼 에디터의 컴포넌트, 디버거와 디버깅되는 프로세스, Code Completion, 메시지 뷰, To-do list 등 IDE와 상호 작용하고 제어하는 100개 이상의 인터페이스(추상 클래스)를 갖추고 있습니다.

Tools API를 사용하면 간편하게 특정 인터페이스를 구현하는 클래스를 작성하고, 다른 인터페이스가 제공하는 서비스로 이동할 수 있습니다. Tools API 코드는 디자인 타임에 디자인 타임 패키지나 DLL로 IDE에 컴파일되고 로드되어야 합니다. 따라서 Tools API 확장을 작성하는 것은 속성 또는 컴포넌트 에디터를 작성하는 것과 어느 정도 비슷합니다. 이 장의 내용을 이해하기 위해서는 패키지 사용 (15장, "패키지와 컴포넌트 사용") 및 컴포넌트 등록 (52장, "디자인 타임 시 컴포넌트 사용")의 기초를 알아야 합니다. 또한 Delphi 스타일 인터페이스에 대한 자세한 내용을 보려면 13장, "VCL 및 CLX에 대한 C++ 랭귀지 지원", 특히 13-2 페이지의 "상속 및 인터페이스" 단원을 읽어 보는 것이 좋습니다.

이 장에서는 다음 항목을 다룹니다.

- Tools API의 개요
- 마법사 클래스 작성
- Tools API 서비스 사용
- 파일 및 에디터 사용
- 폼 및 프로젝트 생성
- 마법사에게 IDE 이벤트 통지
- 마법사 DLL 설치

Tools API의 개요

모든 Tools API 선언은 단일 헤더 파일인 ToolsAPI.hpp에 들어 있습니다. 네임스페이스는 Toolsapi입니다. Tools API를 사용하려면 대개 디자인된 패키지를 사용합니다. 즉, Tools API 애드 인을 디자인 타임 패키지

로 생성하거나, 런타임 패키지를 사용하는 DLL로 생성해야 합니다. 패키지 및 라이브러리 문제에 대한 자세한 내용은 58-6페이지의 "마법사 패키지 설치" 및 58-21페이지의 "마법사 DLL 설치"를 참조하십시오.

Tools API 확장을 작성하기 위한 주요 인터페이스는 *IOTAWizard* 이므로 대부분의 IDE 애드 인을 마법사(wizard)라고 합니다. 대부분의 경우 C++Builder 및 Delphi 마법사는 상호 운용적입니다. Delphi에서 마법사를 작성하고 컴파일한 다음 C++Builder에서 사용하거나, 그 반대의 경우도 가능합니다. 상호 운용성은 같은 버전 번호에서 가장 원활하게 작동하지만 두 제품의 이후 버전에서도 사용할 수 있도록 마법사를 작성할 수 있습니다. 앞으로의 호환성에 대한 자세한 내용은 58-22페이지의 "런타임 패키지 없이 DLL 사용"을 참조하십시오.

Tools API를 사용하기 위해 ToolsAPI 유닛에 정의된 하나 이상의 인터페이스를 구현하는 마법사 클래스를 작성합니다. 13장에서 C++Builder가 객체 파스칼 인터페이스를 추상 클래스로 나타낸다는 것을 알 수 있습니다. 인터페이스를 구현하려면 추상 클래스와 그 조상의 멤버 함수를 오버라이드하고, QueryInterface 함수를 구현하여 인터페이스 GUID를 인식해야 합니다.

마법사는 Tools API가 제공하는 서비스를 사용합니다. 각 서비스는 관련 함수 집합을 나타내는 인터페이스입니다. 인터페이스의 구현은 IDE 내에 숨겨집니다. Tools API는 인터페이스의 구현을 고려하지 않고 마법사를 작성하는 데 사용할 수 있는 인터페이스만 게시합니다. 다양하나 서비스를 통해 소스 에디터, 폼 디자이너, 디버거 등에 액세스할 수 있습니다. 58-7페이지의 "Tools API 서비스 사용" 단원에서 이 문제를 자세히 다룹니다.

서비스 및 다른 인터페이스는 두 개의 기본 범주로 분류됩니다. 타입 이름에 사용되는 접두사로 구별할 수 있습니다.

- NTA(Native Tools API)는 IDE의 *TMainMenu* 객체와 같은 실제 IDE 객체에 직접 액세스를 부여합니다. 이러한 인터페이스를 사용할 때 마법사가 Borland 패키지를 사용해야 합니다. 즉, 마법사는 특정 버전의 IDE에 연결됩니다. 마법사는 디자인 타임 패키지에 상주하거나, 런타임 패키지를 사용하는 DLL에 상주할 수 있습니다.
- OTA(Open Tools API)에서는 패키지 및 인터페이스만을 통한 IDE 액세스가 필요하지 않습니다. Borland의 __fastcall 호출 규칙과 AnsiString 등의 오브젝트 파스칼 타입도 사용할 수 있는 경우 이론상 COM 스타일을 지원하는 모든 언어로 마법사를 작성할 수 있습니다. OTA 인터페이스는 IDE에 대한 전체 액세스를 부여하지 않지만 OTA 인터페이스를 통해 Tools API의 대부분의 기능을 사용할 수 있습니다. 마법사가 OTA 인터페이스만 사용할 경우 특정 버전의 IDE에 의존하지 않는 DLL을 작성할 수 있습니다.

Tools API에는 두 종류의 인터페이스가 있습니다. 프로그래머가 구현해야 하는 인터페이스와 IDE가 구현하는 인터페이스입니다. 대부분의 인터페이스는 후자에 속합니다. 이러한 인터페이스는 IDE의 기능을 정의하지만 실제 구현을 숨깁니다. 프로그래머가 구현해야 하는 인터페이스는 세 범주인 마법사, 통지자, 작성자로 분류할 수 있습니다.

- 이 단원에서 앞서 언급한대로 마법사 클래스는 *IOTAWizard* 인터페이스를 구현하며, 파생된 인터페이스를 구현할 수도 있습니다.
- 통지자는 Tools API에 있는 다른 종류의 인터페이스입니다. IDE는 특별한 상황이 발생할 때 통지자를 사용하여 마법사를 콜백합니다. 통지자 인터페이스를 구현하는 클래스를 작성하고, Tools API를 사용하여 통지자를 등록하며, 사용자가 파일을 열고, 소스 코드를 편집하고, 폼을 수정하고, 디버깅 세션을 시작하는 등의 경우에 IDE가 통지자 객체를 콜백합니다. 58-17페이지의 "마법사에게 IDE 이벤트 통지"에서 통지자에 대해 다룹니다.
- 작성자는 프로그래머가 구현해야 하는 다른 종류의 인터페이스입니다. Tools API는 작성자를 사용하여 새 유닛, 프로젝트 또는 다른 파일을 만들거나 기존의 파일을 엽니다. 58-13페이지의 "폼 및 프로젝트 생성" 단원에서 이 주제를 보다 자세히 설명합니다.

다른 중요한 인터페이스로는 모듈과 에디터가 있습니다. 모듈 인터페이스는 파일이 한 개 이상 있는 열린 유닛을 나타냅니다. 에디터 인터페이스는 열린 파일을 나타냅니다. 소스 파일용 소스 에디터, 폼 파일용 폼 디자이너, 리소스 파일용 프로젝트 리소스 등 IDE의 다른 측면에 대한 액세스를 부여하는 다른 종류의 에디터 인터페이스도 있습니다. 58-11페이지의 "파일 및 에디터 사용" 단원에서 이 항목을 보다 자세히 다룹니다.

다음 단원에서는 마법사 작성 단계를 설명합니다. 각 인터페이스에 대한 자세한 내용은 온라인 도움말 파일을 참조하십시오.

마법사 클래스 작성

마법사에는 네 종류가 있으며, 마법사의 종류는 마법사 클래스가 구현하는 인터페이스에 따라 다릅니다. 표 58.1은 네 종류의 마법사를 설명한 것입니다.

표 58.1 네 종류의 마법사

인터페이스	설명
IOTAFormWizard	대개 새 유닛, 폼 또는 다른 파일을 만듭니다.
IOTAMenuWizard	Help 메뉴에 자동으로 추가됩니다.
IOTAProjectWizard	대개 새 애플리케이션이나 다른 프로젝트를 만듭니다.
IOTAWizard	다른 범주에 속하지 않는 기타 마법사입니다.

네 종류의 마법사는 사용자가 마법사를 호출하는 방식만 다릅니다.

- 메뉴 마법사는 IDE의 Help 메뉴에 추가됩니다. 사용자가 메뉴 항목을 선택하면 IDE가 마법사의 *Execute()* 함수를 호출합니다. 일반적인 마법사는 훨씬 더 유연하므로 대개 메뉴 마법사는 프로토타입과 디버깅에만 사용됩니다.
- 폼 및 프로젝트 마법사는 Object Repository에 상주하므로 이 마법사를 레포지토리 마법사라고 합니다. 사용자는 New Items 다이얼로그 박스에서 이 마법사를 호출합니다. Object Repository(Tools|Repository 메뉴 항목 선택)에서도 마법사를 볼 수 있습니다. New Form 체크 박스를 선택하여 폼 마법사를 호출할 수 있습니다. File|New Form 메뉴 항목을 선택하면 IDE가 폼 마법사를 호출합니다. 또한 Main Form 체크 박스를 선택할 수도 있습니다. 그러면 IDE가 폼 마법사를 새 애플리케이션의 디폴트 폼으로 사용합니다. New Project 체크 박스를 선택하면 프로젝트 마법사를 호출할 수 있습니다. 사용자가 File|New Application을 선택하면 IDE가 선택된 프로젝트 마법사를 호출합니다.
- 네 번째 종류의 마법사는 다른 범주에 속하지 않는 마법사입니다. 일반적인 마법사는 자동으로 또는 단독으로 작업을 수행하지 않습니다. 대신 마법사가 호출되는 방법을 정의해야 합니다.

Tools API는 프로젝트 마법사에게 프로젝트를 만들도록 요구하는 등 마법사에게 제한을 부여하지 않습니다. 필요할 경우 쉽게 프로젝트 마법사를 작성하여 폼을 만들고, 폼 마법사를 작성하여 프로젝트를 만들 수도 있습니다.

마법사 인터페이스 구현

모든 마법사 클래스는 최소한 조상인 *IOTANotifier*와 *IInterface*도 함께 구현해야 하는 *IOTAWizard*를 구현해야 합니다. 폼 마법사와 프로젝트 마법사는 모든 조상 인터페이스 즉, *IOTARepositoryWizard*, *IOTAWizard*, *IOTANotifier* 및 *IInterface*를 구현해야 합니다.

IInterface 구현은 오브젝트 파스칼 인터페이스의 일반적인 규칙을 따라야 합니다. 이 규칙은 COM 인터페이스용 규칙과 같습니다. 즉, *QueryInterface*는 타입 변환을 수행하고, *AddRef*와 *Release*는 참조 카운팅을 관리합니다. 일반적인 기본 클래스를 사용하여 마법사 및 통지자 클래스 작성을 단순화할 수 있습니다.

예를 들어, Delphi에는 함수 바디가 비어 있는 *IOTANotifier*를 구현하는 *TNotifierObject* 클래스가 있습니다. 아래에 표시된대로 C++로 비슷한 클래스를 작성할 수 있습니다.

```
class PACKAGE NotifierObject : public IOTANotifier {
public:
  __fastcall NotifierObject() : ref_count(0) {}
  virtual __fastcall ~NotifierObject();
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
};
```

IInterface 인터페이스의 구현은 간단합니다.

```
ULONG __stdcall NotifierObject::AddRef()
{
  return InterlockedIncrement(&ref_count);
}
ULONG __stdcall NotifierObject::Release()
{
  ULONG result = InterlockedDecrement(&ref_count);
  if (ref_count == 0)
    delete this;
  return result;
}
HRESULT __stdcall NotifierObject::QueryInterface(const GUID& iid, void** obj)
{
  if (iid == __uuidof(IInterface)) {
    *obj = static_cast<IInterface*>(this);
    static_cast<IInterface*>(*obj)->AddRef();
    return S_OK;
  }
  if (iid == __uuidof(IOTANotifier)) {
    *obj = static_cast<IOTANotifier*>(this);
```

```
    static_cast<IOTANotifier*>(*obj)->AddRef();
    return S_OK;
  }
  return E_NOINTERFACE;
}
```

마법사가 *IOTANotifier*에서 상속되고, 따라서 마법사의 모든 함수를 구현해야 하는 경우에도 대개 IDE는 이러한 함수를 사용하지 않으므로 구현이 비어 있을 수 있습니다.

```
void __fastcall NotifierObject::AfterSave()  {}
void __fastcall NotifierObject::BeforeSave() {}
void __fastcall NotifierObject::Destroyed()  {}
void __fastcall NotifierObject::Modified()   {}
```

*NotifierObject*를 기본 클래스로 사용하려면 다중 상속을 사용해야 합니다. 마법사 클래스가 *NotifierObject*에서 상속되어야 하며, 또한 *IOTAWizard*와 같은 구현할 마법사 인터페이스에서 상속되어야 합니다. *IOTAWizard*는 *IOTANotifier*와 *IInterface*에서 상속되므로 파생된 클래스가 모호합니다. *AddRef*()와 같은 함수가 조상 상속 그래프의 모든 분기에서 선언됩니다. 이 문제를 해결하려면 기본 클래스 하나를 주 기본 클래스로 선택하고, 모든 모호한 클래스를 해당 클래스에 위임하십시오. 예를 들어, 클래스 선언은 다음과 같이 나타납니다.

```
class PACKAGE MyWizard : public NotifierObject, public IOTAMenuWizard {
  typedef NotifierObject inherited;
public:
  // IOTAWizard
  virtual AnsiString __fastcall GetIDString();
  virtual AnsiString __fastcall GetName();
  virtual TWizardState __fastcall GetState();
  virtual void __fastcall Execute();

// IOTAMenuWizard
  virtual AnsiString __fastcall GetMenuText();

  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
};
```

클래스 구현이 다음을 포함할 수 있습니다.

```
ULONG __stdcall MyWizard::AddRef()  { return inherited::AddRef(); }
ULONG __stdcall MyWizard::Release() { return inherited::Release(); }
HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void** obj)
{
  if (iid == __uuidof(IOTAMenuWizard)) {
    *obj = static_cast<IOTAMenuWizard*>(this);
    static_cast<IOTAMenuWizard*>(*obj)->AddRef();
    return S_OK;
  }
```

```
    if (iid == __uuidof(IOTAWizard)) {
      *obj = static_cast<IOTAWizard*>(this);
      static_cast<IOTAWizard*>(*obj)->AddRef();
      return S_OK;
    }
    return inherited::QueryInterface(iid, obj);
  }
```

AfterSave, *BeforeSave* 등은 기본 함수에 비어 있는 함수 바디가 있으므로, 파생된 클래스에서도 이들을 비어 있는 함수 바디로 남겨두고, 불필요한 *inherited::AfterSave*() 호출을 피할 수 있습니다.

인터페이스 구현 단순화

오브젝트 파스칼 인터페이스는 COM 인터페이스와 비슷하므로 COM 마법사를 사용하여 마법사 클래스를 작성할 수 있습니다. 그러나 COM은 간단한 오브젝트 파스칼 인터페이스에 필요한 것보다 훨씬 많은 오버헤드를 포함합니다. 기본 클래스를 주의해서 사용하고, 복사나 붙여넣기를 많이 수행하지 않는 경우에는 COM 마법사를 사용하지 않고 단순한 구현을 사용하는 편이 더 쉽습니다. 예를 들어, 간단한 매크로를 정의하여 *QueryInterface*를 쉽게 구현할 수 있습니다.

```
  #define QUERY_INTERFACE(T, iid, obj)   \
    if ((iid) == __uuidof(T)) {          \
      *(obj) = static_cast<T*>(this);    \
      static_cast<T*>(*(obj))->AddRef(); \
      return S_OK;                       \
    }
```

이 매크로를 다음과 같이 사용하십시오.

```
  HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void* obj)
  {
    QUERY_INTERFACE(IOTAMenuWizard, iid, obj);
    QUERY_INTERFACE(IOTAWizard, iid, obj);
    return inherited::QueryInterface(iid, obj);
  }
```

또한 *IOTAWizard*와 사용 중인 파생된 클래스의 모든 멤버 함수를 오버라이드해야 합니다. 여러 마법사 인터페이스의 함수 대부분은 보면 알 수 있습니다. IDE는 마법사의 함수를 호출하여 엔드 유저에게 마법사를 나타낼 방법 및 사용자가 호출할 때 마법사를 실행할 방법을 결정합니다.

마법사 클래스 작성을 마치면 다음 단계에 따라 마법사를 설치합니다.

마법사 패키지 설치

다른 디자인 타임 패키지와 마찬가지로 마법사 패키지에는 *Register* 함수가 최소한 한 개는 있어야 합니다. *Register* 함수에 대한 자세한 내용은 52장, "디자인 타임 시 컴포넌트 사용"을 참조하십시오. 아래에 표시된 대로 *Register* 함수에서 *RegisterPackageWizard*를 호출하고, 마법사 객체를 유일한 인수로 전달하여 마법사를 필요한 수만큼 등록할 수 있습니다.

```
  namespace Example {
    void __fastcall PACKAGE Register()
    {
      RegisterPackageWizard(new MyWizard());
```

```
    RegisterPackageWizard(new MyOtherWizard());
  }
}
```

또한 속성 에디터, 컴포넌트 등을 같은 패키지의 일부로 등록할 수 있습니다.

디자인 타임 패키지는 메인 C++Builder 애플리케이션의 일부입니다. 즉, 전체 애플리케이션과 다른 모든 디자인 타임 패키지에서 폼 이름이 고유해야 합니다. 패키지 사용의 주된 단점은 다른 사람이 지정한 폼 이름을 알 수 없다는 점입니다.

개발 도중 다른 디자인 타임 패키지에서와 같이 패키지 관리자에서 Install 버튼을 클릭하여 마법사 패키지를 설치합니다. IDE가 패키지를 컴파일 및 링크하여 로드를 시도합니다. 패키지가 성공적으로 로드되었는지 알리는 다이얼로그 박스가 표시됩니다.

Tools API 서비스 사용

필요한 작업을 수행하기 위해 마법사는 에디터, 윈도우, 메뉴 등 IDE에 액세스해야 합니다. 이것이 서비스 인터페이스의 역할입니다. Tools API에는 파일 작업을 수행할 액션 서비스, 소스 코드 에디터에 액세스할 에디터 서비스, 디버거에 액세스할 디버거 서비스 등 많은 서비스가 포함되어 있습니다. 표 58.2는 모든 서비스 인터페이스를 요약한 것입니다.

표 58.2 Tools API 서비스 인터페이스

인터페이스	설명
INTAServices	원시 IDE 객체인 메인 메뉴, 액션 리스트, 이미지 리스트 및 툴바에 대한 액세스를 제공합니다.
IOTAActionServices	열기, 닫기, 저장 및 파일 다시 로드와 같은 기본 파일 작업을 수행합니다.
IOTACodeCompletionServices	마법사가 사용자 정의 Code Completion 관리자를 설치하도록 허용하여 Code Completion에 대한 액세스를 제공합니다.
IOTADebuggerServices	디버거에 대한 액세스를 제공합니다.
IOTAEditorServices	소스 코드 에디터와 해당 내부 버퍼에 대한 액세스를 제공합니다.
IOTAKeyBindingServices	마법사가 사용자 정의 키보드 연결을 등록하도록 허용합니다.
IOTAKeyboardServices	키보드 매크로와 연결에 대한 액세스를 제공합니다.
IOTAKeyboardDiagnostics	키스트로크의 디버깅을 토글합니다.
IOTAMessageServices	메시지 뷰에 대한 액세스를 제공합니다.
IOTAModuleServices	열려 있는 파일에 대한 액세스를 제공합니다.
IOTAPackageServices	설치된 모든 패키지와 해당 컴포넌트 이름을 쿼리합니다.
IOTAServices	기타 서비스입니다.
IOTAToDoServices	마법사가 사용자 정의 To-Do 관리자를 설치하도록 허용하여 To-Do list에 대한 액세스를 제공합니다.
IOTAToolsFilter	도구 필터 통지자를 등록합니다.
IOTAWizardServices	마법사를 등록하거나 등록 취소합니다.

서비스 인터페이스를 사용하려면 *BorlandIDEServices* 변수를 필요한 서비스로 타입 변환하십시오. *QueryInterface*를 호출하거나, 좀 더 편리한 *Sysutils::Supports* 함수를 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
void set_keystroke_debugging(bool debugging)
{
  _di_IOTAKeyboardDiagnostics diag;
  if (Supports(BorlandIDEServices, IID_IOTAKeyboardDiagnostics, &diag))
    diag->KeyTracing = debugging;
}
```

마법사가 특정 서비스를 사용해야 할 경우 서비스에 대한 포인터를 마법사 클래스의 데이터 멤버로 유지할 수 있습니다. *DelphiInterface* 템플릿(예: *_di_IOTAModuleServices*)을 사용하면 Tools API가 객체 수명을 자동으로 관리하므로 마법사의 소멸자에 특별한 조치를 취할 필요가 없습니다.

원시 IDE 객체 사용

마법사는 IDE의 메인 메뉴, 툴바, 액션 리스트 및 이미지 리스트에 대한 전체 액세스를 가지고 있습니다. IDE의 여러 컨텍스트 메뉴에 Tools API를 통해 액세스할 수 없습니다. 이 단원에는 마법사가 이 원시 IDE 객체를 사용하여 IDE와 상호 작용하는 방법에 대한 간단한 예제가 있습니다.

INTAServices 인터페이스 사용

원시 IDE 객체 사용을 위한 출발점은 *INTAServices* 인터페이스입니다. 이 인터페이스를 사용하여 이미지 리스트에 이미지를 추가하고, 액션 리스트에 작업을 추가하고, 메인 메뉴에 메뉴 항목을 추가하며, 툴바에 버튼을 추가합니다. 작업을 메뉴 항목과 툴 버튼에 연결할 수 있습니다. 마법사가 배포될 때 마법사가 만드는 객체를 클린업해야 하지만, 이미지 리스트에 추가한 이미지를 삭제하면 안됩니다. 이미지를 삭제하면 이 마법사보다 나중에 추가된 모든 이미지의 인덱스가 뒤섞입니다.

마법사는 IDE의 실제 *TMainMenu*, *TActionList*, *TImageList* 및 *TToolBar* 객체를 사용하므로 다른 애플리케이션에서 사용하는 방법으로 코드를 작성할 수 있습니다. IDE의 작동이 중지되거나, File 메뉴 삭제와 같은 중요한 기능을 사용할 수 없는 경우가 많다는 의미이기도 합니다.

이미지 리스트에 이미지 추가

마법사를 호출할 메뉴 항목을 추가한다고 가정합니다. 또한 사용자가 마법사를 호출하는 툴바 버튼을 추가할 수 있게 하려고 합니다. 먼저 IDE의 이미지 리스트에 이미지를 추가합니다. 그러면 작업에 이미지 인덱스를 사용할 수 있으며, 메뉴 항목과 툴바 버튼에서 이 이미지 인덱스가 차례로 사용됩니다. Image Editor를 사용하여 16*16 비트맵 리소스가 들어 있는 리소스 파일을 만듭니다. 마법사의 생성자에 다음 코드를 추가합니다.

```
_di_INTAServices services;
Supports(BorlandIDEServices, IID_INTAServices, &services);

// Add an image to the image list.
Graphics::TBitmap* bitmap(new Graphics::TBitmap());
bitmap->LoadFromResourceName(reinterpret_cast<unsigned>(HInstance), "Bitmap1");
int image = services->AddMasked(bitmap, bitmap->TransparentColor,
                                 "Tempest Software.intro wizard image");
delete bitmap;
```

HInstance에 잘못된 타입이 있으므로 LoadFromResourceID가 필요로 하는 타입으로 타입 변환해야 합니다. 또한 리소스 파일에서 지정하는 이름이나 ID로 리소스를 로드해야 합니다. #pragma 리소스를 사용하여 리소스 파일을 패키지에 추가합니다. 이미지의 배경색으로 해석될 색상을 선택해야 합니다. 배경색을 사용하지 않으려면 비트맵에 없는 색상을 선택하십시오.

액션 리스트에 작업 추가

이미지 인덱스는 아래에 표시된대로 작업을 만드는 데 사용됩니다. 마법사는 *OnExecute* 및 *OnUpdate* 이벤트를 사용합니다. 일반적인 시나리오는 마법사가 *OnUpdate* 이벤트를 사용하여 해당 작업을 사용하거나 사용할 수 없게 하는 것입니다. *OnUpdate* 이벤트가 빨리 반환되는지 확인하십시오. 그렇지 않으면 사용자가 마법사를 로딩한 후에 IDE가 느려질 수 있습니다. 작업의 *OnExecute* 이벤트는 마법사의 *Execute* 메소드와 비슷합니다. 메뉴 항목을 사용하여 폼 마법사나 프로젝트 마법사를 호출하는 경우 *OnExecute*를 사용하여 직접 *Execute*를 호출할 수도 있습니다.

```
action = new TAction(0);
action->ActionList = services->ActionList;
action->Caption    = GetMenuText();
action->Hint       = "Display a silly dialog box";
action->ImageIndex = image;
action->OnUpdate   = action_update;
action->OnExecute  = action_execute;
```

메뉴 항목은 해당 *Action* 속성을 새로 만든 작업으로 설정합니다. 메뉴 항목을 만들려면 메뉴 항목을 삽입할 위치를 아는 것이 중요합니다. 아래의 예제에서는 View 메뉴를 찾아 새 메뉴 항목을 View 메뉴에 첫 번째 항목으로 삽입합니다. 일반적으로 절대 위치에 의존하는 것은 좋지 않습니다. 그러면 다른 마법사가 메뉴에 삽입되더라도 알 수가 없습니다. 다음 버전의 C++Builder에서는 메뉴 순서를 다시 지정할 수도 있습니다. 특정 이름을 사용하여 메뉴 항목의 메뉴를 검색하는 것이 더 좋은 방법입니다. 다음은 이 사실을 분명히 보여 주는 간단한 예제입니다.

```
for (int i = 0; i< services->MainMenu->Items->Count; ++i)
{
  TMenuItem* item = services->MainMenu->Items->Items[i];
  if (CompareText(item->Name, "ViewsMenu") == 0)
  {
    menu_item = new TMenuItem(0);
    menu_item->Action = action;
    item->Insert(0, menu_item);
  }
}
```

IDE 액션 리스트에 작업을 추가하면 툴바를 사용자 정의할 때 사용자가 작업을 볼 수 있습니다. 사용자는 작업을 선택하여 툴바에 버튼으로 추가할 수 있습니다. 그러면 마법사가 언로드될 때 문제가 생길 수 있습니다. 모든 툴 버튼이 존재하지 않는 작업과 *OnClick* 이벤트 핸들러에 대한 잘못된 포인터가 됩니다. 액세스 위반을 방지하려면 마법사가 해당 작업을 참조하는 모든 툴 버튼을 찾아 제거해야 합니다.

툴바 버튼 삭제

툴바에서 버튼을 제거하는 데 사용할 편리한 버튼은 없습니다. 첫 번째 매개변수가 변경할 컨트롤이고, 두 번째 매개변수가 이 컨트롤을 제거할 0이거나 툴바에 추가할 0 이외의 값인 위치로 CM_CONTROLCHANGE 메시지를 보내야 합니다. 툴바 버튼을 제거한 후에 소멸자가 작업 및 메뉴 항목을 삭제합니다. 이 항목을 삭제하면 IDE의 ActionList와 MainMenu에서 항목이 자동으로 제거됩니다.

```
void __fastcall remove_action (TAction* action, TToolBar* toolbar)
{
  for (int i = toolbar->ButtonCount; --i >= 0; )
  {
    TToolButton* button = toolbar->Buttons[i];
    if (button->Action == action)
    {
      // Remove “button” from “toolbar”.
      toolbar->Perform(CM_CONTROLCHANGE, WPARAM(button), 0);
      delete button;
    }
  }
}
__fastcall MyWizard::~MyWizard()
{
  _di_INTAServices services;
  Supports(BorlandIDEServices, INTAServices, &services);
  // Check all the toolbars, and remove any buttons that use
  // this action.
  remove_action(action, services->ToolBar[sCustomToolBar]);
  remove_action(action, services->ToolBar[sDesktopToolBar]);
  remove_action(action, services->ToolBar[sStandardToolBar]);
  remove_action(action, services->ToolBar[sDebugToolBar]);
  remove_action(action, services->ToolBar[sViewToolBar]);
  remove_action(action, services->ToolBar[sInternetToolBar]);
  delete menu_item;
  delete action;
}
```

이 간단한 예제에서 알 수 있듯이 마법사는 다양한 방법으로 IDE와 상호 작용합니다. 그러나 이러한 유연성에는 책임이 뒤따릅니다. 잘못 연결된 포인터나 다른 액세스 위반이 발생하기 쉽습니다. 다음 단원에는 이러한 종류의 문제를 진단하는 데 도움이 되는 팁이 있습니다.

마법사 디버깅

NTA(Native Tools API)를 사용하는 마법사를 작성할 때 IDE의 작동을 중지시키는 코드를 작성할 수 있습니다. 또한 설치되기는 하지만 원하는 대로 작동하지 않는 마법사를 작성할 수도 있습니다. 디자인 타임 코드를 사용할 때의 문제 중 하나가 디버깅입니다. 그러나 해결하기 어려운 문제는 아닙니다. 마법사가 C++Builder 자체에 설치되기 때문에 Run|Parameters 메뉴 항목에서 패키지의 Host Application을 C++Builder 실행 파일(bcb.exe)로 설정하기만 하면 됩니다.

패키지를 디버깅해야 할 때 패키지를 설치하지 마십시오. 대신 메인 메뉴 표시줄에서 Run|Run을 선택합니다. 그러면 C++Builder의 새 인스턴스가 시작됩니다. 새 인스턴스에서 메인 메뉴 표시줄의 Components|Install Package를 선택하여 이미 컴파일된 패키지를 설치합니다. C++Builder의 원래 인스턴스로 돌아가, 마법사 소스 코드에서 브레이크포인트를 설정할 수 있는 위치를 표시하는 파란 점을 확인해야 합니다. 그렇지 않으면 컴파일러 옵션을 더블 클릭하여 디버깅을 사용할 수 있는지 확인하고, 올바른 패키지가 로드되었는지 확인합니다. 그리고 프로세스 모듈을 두 번 확인하여 필요한 .bpl 파일을 로드했는지 다시 한 번 점검합니다.

이 방법을 사용하여 VCL, CLX 또는 RTL 코드로 디버그할 수는 없지만 마법사 자체에 대해서는 문제점을 알려 주는 데 충분한 전체 디버그 기능을 가지고 있습니다.

인터페이스 버전 번호

*IOTAMessageServices*와 같은 일부 인터페이스의 선언을 자세히 살펴보면 *IOTAMessageServices40*에서 *IOTAMessageServices50*이 상속되는 것처럼 이 선언이 비슷한 이름을 가진 다른 인터페이스에서 상속되는 것을 알 수 있습니다. 버전 번호를 사용하면 C++Builder 릴리스 간에 변경되지 않도록 코드를 보호할 수 있습니다.

Tools API는 COM의 기본 원칙 즉, 인터페이스와 그 GUID는 변경되지 않는다는 원칙을 준수합니다. 새 릴리스에서 인터페이스에 기능이 추가되면 Tools API가 이전 릴리스에서 상속되는 새 인터페이스를 선언합니다. GUID는 똑같이 유지되고 인터페이스가 변경되지 않은 채 이전 GUID에 추가됩니다. 새 인터페이스는 새로운 GUID를 갖습니다. 이전 GUID를 사용하는 이전 마법사는 계속 작동합니다.

Tools API는 인터페이스 이름을 변경하여 소스 코드 호환성을 유지합니다. 이 작업을 보려면 Tools API에 있는 Borland 구현 인터페이스와 사용자 구현 인터페이스를 구별하는 것이 중요합니다. IDE가 인터페이스를 구현할 경우 인터페이스 이름이 최신 버전의 인터페이스로 남아 있습니다. 새 기능은 기존 코드에 영향을 주지 않습니다. 이전 인터페이스에는 이전 버전 번호가 부여되어 있습니다.

사용자 구현 인터페이스의 경우에는 기본 인터페이스의 새 멤버 함수가 코드에 새 함수를 필요로 합니다. 따라서 이름이 이전 인터페이스와 결합되고 새 인터페이스 끝에 버전 번호가 부여됩니다.

예를 들어, 메시지 서비스를 생각해 보십시오. C++Builder 6에는 새로운 기능인 메시지 그룹이 도입되었습니다. 따라서 기본 메시지 서비스 인터페이스는 멤버 함수를 필요로 합니다. 이 함수는 새 인터페이스 클래스에서 선언되고, 이름이 *IOTAMessageServices*로 유지됩니다. 이전 메시지 서비스 인터페이스의 이름은 *IOTAMessageServices50*(버전 5)으로 변경되었습니다. 멤버 함수가 같기 때문에 이전 *IOTAMessageServices*의 GUID는 새 *IOTAMessageServices50*의 GUID와 같습니다.

사용자 구현 인터페이스의 예로 *IOTAIDENotifier*에 대해 생각해 보십시오. C++Builder 5는 새로 오버로드된 함수 *AfterCompile*과 *BeforeCompile*을 추가했습니다. *IOTAIDENotifier*를 사용하는 기존 코드를 변경할 필요는 없지만, 새 기능을 필요로 하는 새로운 코드는 *IOTAIDENotifier50*에서 상속되는 새 함수를 오버라이드하도록 수정되어야 합니다. 버전 6에는 새로 추가된 기능이 없으므로 사용할 현재 버전은 *IOTAIDENotifier50*입니다.

경험상 새 코드를 작성할 때 가장 많이 파생된 클래스를 사용합니다. C++Builder의 새 릴리스에서 기존 마법사를 단지 다시 컴파일하려면 소스 코드를 그대로 유지하십시오.

파일 및 에디터 사용

작업을 계속하기 전에 Tools API가 파일을 사용하는 방법을 이해해야 합니다. 주요 인터페이스는 *IOTAModule*입니다. 모듈은 논리적으로 관련된 열려 있는 파일의 집합을 나타냅니다. 예를 들어, 하나의 모듈은 하나의 유닛을 나타냅니다. 모듈은 차례대로 하나 이상의 에디터를 사용하며, 각 에디터는 구현(.cpp), 인터페이스(.h) 또는 폼(.dfm or .xfm) 파일과 같은 하나의 파일을 나타냅니다. 에디터 인터페이스는 IDE 에디터의 내부 상태를 반영하므로 사용자가 변경 내용을 저장하지 않은 경우에도 사용자에게 표시되는 수정된 코드와 폼이 마법사에 표시됩니다.

모듈 인터페이스 사용

모듈 인터페이스를 가져오려면 모듈 서비스(*IOTAModuleServices*)를 사용하여 시작하십시오. 모든 열린 모듈에 대해 모듈을 쿼리하거나, 파일 이름이나 폼 이름에서 모듈을 알아내거나, 파일을 열어 해당 모듈 인터페이스를 가져올 수 있습니다.

파일의 종류가 다르면 모듈의 종류도 다릅니다(예: 프로젝트, 리소스 및 타입 라이브러리). 모듈 인터페이스를 특정 모듈 인터페이스 종류로 타입 변환하여 모듈이 해당 타입의 모듈인지 알아봅니다. 예를 들어, 현재 프로젝트 그룹 인터페이스를 가져오는 방법은 다음과 같습니다.

```
// Return the current project group, or 0 if there is no project group.
_di_IOTAProjectGroup __fastcall CurrentProjectGroup()
{
  _di_IOTAModuleServices svc;
  Supports(BorlandIDEServices, IID_IOTAModuleServices, &svc);
  for (int i = 0; i< svc->ModuleCount; ++i)
  {
    _di_IOTAModule module = svc->Modules[i];
    _di_IOTAProjectGroup group;
    if (Supports(module, IID_IOTAProjectGroup, &group))
      return group;
  }
  return 0;
}
```

에디터 인터페이스 사용

모든 모듈에는 최소한 하나의 에디터 인터페이스가 있습니다. 일부 모듈에는 구현 파일(.cpp), 인터페이스 파일(.h) 및 폼 설명 파일(.dfm)과 같은 여러 에디터가 있습니다. 모든 데이터는 *IOTAEditor* 인터페이스를 구현하고, 데이터를 특정 타입으로 타입 변환하여 에디터 종류를 확인합니다. 예를 들어, 유닛에 대한 폼 에디터 인터페이스를 가져오려면 다음과 같이 할 수 있습니다.

```
// Return the form editor for a module, or 0 if the unit has no form.
_di_IOTAFormEditor __fastcall GetFormEditor(_di_IOTAModule module)
{
  for (int i = 0; i< module->ModuleFileCount; ++i)
  {
    _di_IOTAEditor editor = module->ModuleFileEditors[i];
    _di_IOTAFormEditor formEditor;
    if (Supports(editor, IID_IOTAFormEditor, &formEditor))
      return formEditor;
  }
  return 0;
}
```

에디터 인터페이스는 에디터의 내부 상태에 대한 액세스를 제공합니다. 사용자가 편집하고 있는 소스 코드나 컴포넌트를 검사하고, 소스 코드, 컴포넌트 또는 속성을 변경하고, 소스 및 폼 에디터에서 선택 사항을 변경하고, 엔드 유저가 수행할 수 있는 거의 모든 에디터 작업을 실행할 수 있습니다.

폼 에디터 인터페이스를 사용하면 마법사가 폼의 모든 컴포넌트에 액세스할 수 있습니다. 루트 폼이나 데이터 모듈을 비롯한 각 컴포넌트에는 연결된 *IOTAComponent* 인터페이스가 있습니다. 마법사는 대부분의 컴포넌트 속성을 조사하거나 변경할 수 있습니다. 컴포넌트를 완전하게 제어해야 할 경우 *IOTAComponent* 인터페이스를 *INTAComponent*로 타입 변환할 수 있습니다. 원시 컴포넌트 인터페이스를 사용하면 마법사가 *TComponent* 속성에 직접 액세스할 수 있습니다. *TFont*와 같이 NTA 스타일 인터페이스를 통해서만 가능한 클래스 타입 속성을 읽거나 수정해야 할 경우 이 사항이 중요합니다.

폼 및 프로젝트 생성

C++Builder에는 이미 설치된 많은 폼 및 프로젝트 마법사가 있으며, 프로그래머가 직접 마법사를 작성할 수도 있습니다. Object Repository를 사용하면 프로젝트에 사용할 수 있는 정적 템플릿을 만들 수 있습니다. 그러나 동적 템플릿일 경우에는 마법사의 기능이 훨씬 강화됩니다. 마법사는 사용자에게 프롬프트를 표시하고, 사용자의 응답에 따라 다른 종류의 파일을 만들 수 있습니다. 이 단원에서는 폼 또는 프로젝트 마법사를 작성하는 방법을 설명합니다.

모듈 생성

대개 폼 마법사나 프로젝트 마법사는 하나 이상의 새 파일을 만듭니다. 그러나 실제 파일 대신 이름이 지정되지 않고 저장되지 않은 모듈을 만드는 것이 가장 좋습니다. 모듈을 저장할 때 파일 이름을 묻는 프롬프트가 나타납니다. 마법사는 작성자 객체를 사용하여 그러한 모듈을 만듭니다.

작성자 클래스는 *IOTACreator*에서 상속되는 작성자 인터페이스를 구현합니다. 마법사는 작성자 객체를 모듈 서비스의 *CreateModule* 메소드로 전달하고, IDE는 모듈을 만드는 데 필요한 매개변수에 대해 작성자 객체를 콜백합니다.

예를 들어, 새 폼을 만드는 폼 마법사는 대개 *GetExisting*()을 구현하여 **false**를 반환하고, *GetUnnamed*()를 구현하여 **true**를 반환합니다. 폼 마법사는 이름이 없고(파일을 저장하기 전에 이름을 선택해야 함) 기존의 파일이 돌아가지 않는(변경하지 않은 경우에도 사용자가 파일을 저장해야 함) 모듈을 만듭니다. 작성자의 다른 메소드는 IDE에 작성 중인 파일 종류(예: 프로젝트, 유닛 또는 폼)를 알리거나, 파일 내용을 제공하거나, 폼 이름, 조상 이름 및 다른 중요한 정보를 반환합니다. 추가 콜백이 있을 경우 마법사가 새로 만든 프로젝트에 모듈을 추가하거나, 새로 만든 폼에 컴포넌트를 추가합니다.

폼 마법사나 프로젝트 마법사에서 때때로 요구하는 새 파일을 만들려면 대개 새 파일의 내용을 제공해야 합니다. 파일의 내용을 제공하려면 *IOTAFile* 인터페이스를 구현하는 새 클래스를 작성하십시오. 마법사가 디폴트 파일 내용을 사용할 수 있을 경우 *IOTAFile*을 반환하는 함수에서 0 포인터를 반환할 수 있습니다.

예를 들어, 구성 안에 각 소스 파일 맨 위쪽에 표시할 표준 주석 블록이 있다고 가정합니다. Object Repository에서 정적 템플릿을 사용하여 이 작업을 수행할 수 있지만 주석 블록을 수동으로 업데이트하여 작성자와 작성 날짜를 반영할 필요는 없습니다. 대신 파일이 만들어지면 작성자를 사용하여 동적으로 주석 블록을 채울 수 있습니다.

첫 단계는 새 유닛과 폼을 만드는 마법사를 작성하는 것입니다. 대부분의 작성자 함수는 0, 빈 문자열 또는 다른 기본값을 반환합니다. Tools API는 이 값을 통해 새 유닛이나 폼을 작성하기 위한 디폴트 동작을 사용합니다. *GetCreatorType*을 오버라이드하여 Tools API에 만들 모듈 종류가 유닛인지 폼인지 알립니다. 유닛을 만들려면 sUnit 매크로를 반환하십시오. 폼을 만들려면 sForm을 반환하십시오. 코드를 단순화하려면 작성자 타입을 생성자에 대한 인수로 사용하는 단일 클래스를 사용하십시오. *GetCreatorType*이 값을 반환할 수 있도록 작성자 타입을 데이터 멤버에 저장합니다. *NewImplSource*와 *NewIntfSource*를 오버라이드하여 원하는 해당 파일 내용을 반환합니다.

```
class PACKAGE Creator : public IOTAModuleCreator {
public:
  __fastcall Creator(const AnsiString creator_type)
  : ref_count(0), creator_type(creator_type) {}
  virtual __fastcall ~Creator();

// IOTAModuleCreator
  virtual AnsiString __fastcall GetAncestorName();
  virtual AnsiString __fastcall GetImplFileName();
  virtual AnsiString __fastcall GetIntfFileName();
  virtual AnsiString __fastcall GetFormName();
  virtual bool __fastcall GetMainForm();
  virtual bool __fastcall GetShowForm();
  virtual bool __fastcall GetShowSource();
  virtual _di_IOTAFile __fastcall NewFormFile(
    const AnsiString FormIdent, const AnsiString AncestorIdent);
  virtual _di_IOTAFile __fastcall NewImplSource(
    const AnsiString ModuleIdent, const AnsiString FormIdent,
    const AnsiString AncestorIdent);
  virtual _di_IOTAFile __fastcall NewIntfSource(
    const AnsiString ModuleIdent, const AnsiString FormIdent,
    const AnsiString AncestorIdent);
  virtual void __fastcall FormCreated(
    const _di_IOTAFormEditor FormEditor);

// IOTACreator
  virtual AnsiString __fastcall GetCreatorType();
  virtual bool __fastcall GetExisting();
  virtual AnsiString __fastcall GetFileSystem();
  virtual _di_IOTAModule __fastcall GetOwner();
  virtual bool __fastcall GetUnnamed();

protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

private:
  long ref_count;
  const AnsiString creator_type;
};
```

대부분의 *Creator* 멤버는 0이나 빈 문자열을 반환합니다. **false**를 반환하는 *GetExisting*을 제외하고, 부울 메소드는 **true**를 반환합니다. 가장 주목할 만한 메소드는 현재 프로젝트 모듈에 대한 포인터를 반환하거나 프로젝트가 없을 경우 0을 반환하는 *GetOwner*입니다. 간단하게 현재 프로젝트나 현재 프로젝트 그룹을 발견하는 방법은 없습니다. 대신 *GetOwner*모든 열려 있는 모듈에서 반복되어야 합니다. 프로젝트 그룹이 있을 경우 열려 있는 유일한 프로젝트 그룹이므로 *GetOwner*가 현재 프로젝트를 반환합니다. 그렇지 않으면 함수가 처음 발견한 프로젝트를 반환하거나, 열려 있는 프로젝트가 없을 경우에는 0을 반환합니다.

```
_di_IOTAModule __fastcall Creator::GetOwner()
{
  // Return the current project.
  _di_IOTAProject result = 0;
  _di_IOTAModuleServices svc =
interface_cast<IOTAModuleServices>(BorlandIDEServices);
  for (int i = 0; i< svc->ModuleCount; ++i)
  begin
    _di_IOTAModule module = svc->Modules[i];
    _di_IOTAProject project;
    _di_IOTAProjectGroup group;
    if (Supports(module, IID_IOTAProject, &project)) {
      // Remember the first project module.
      if (result == 0)
        result = project;
    } else if (Supports(module, IID_IOTAProjectGroup, &group)) {
      // Found the project group, so return its active project.
      result = group->ActiveProject;
      break;
    }
  }
  return result;
}
```

작성자는 *NewFormSource*에서 0을 반환하여 디폴트 폼 파일을 생성합니다. 주목할 만한 메소드는 파일 내용을 반환하는 *IOTAFile* 인스턴스를 만드는 *NewImplSource* 및 *NewIntfSource*입니다.

File 클래스는 *IOTAFile* 인터페이스를 구현합니다. 이 클래스는 파일 보존 기간으로 –1을 반환하고(파일이 존재하지 않음을 의미함), 문자열로 파일 내용을 반환합니다. *File* 클래스를 간단하게 유지하기 위해 작성자가 문자열을 생성하고 *File* 클래스는 문자열을 전달하기만 합니다.

```
class File : public IOTAFile {
public:
  __fastcall File(const AnsiString source);
  virtual __fastcall ~File();
  AnsiString __fastcall GetSource();
  System::TDateTime __fastcall GetAge();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
  AnsiString source;
};
__fastcall File::File(const AnsiString source)
: ref_count(0), source(source)
{}
AnsiString __fastcall File::GetSource()
```

```
{
  return source;
}
System::TDateTime __fastcall File::GetAge()
{
  return -1;
}
```

파일 내용의 텍스트를 리소스에 저장하여 수정하기 편리하게 할 수 있지만, 이 예제에서는 단순화하기 위해 소스 코드를 마법사에 하드코드합니다. 아래의 예제에서는 폼이 있다고 가정하여 소스 코드를 생성합니다. 보다 단순한 경우의 유닛을 쉽게 추가할 수 있습니다. *FormIdent*를 테스트하고, 비어 있을 경우 일반적인 유닛을 만들고, 그렇지 않으면 폼 유닛을 만드십시오. 코드의 기본 스켈레톤은 IDE의 기본값(맨 위에 주석으로 추가)과 같지만, 원하는 대로 수정할 수 있습니다.

```
_di_IOTAFile __fastcall Creator::NewImplSource(
  const AnsiString ModuleIdent,
  const AnsiString FormIdent,
  const AnsiString AncestorIdent)
{
  const AnsiString form_source =
"/*-----------------------------------------------------------------\n"
" %m -description\n"
" Copyright © %yYour company, inc.\n"
" Created on %d\n"
" By %u\n"
"  -----------------------------------------------------------------*/\n"
"\n"
"#include <vcl.h>\n"
"#pragma hdrstop\n"
"\n"
"#include \"%m.h\"\n"
"//-----------------------------------------------------------------\n"
"#pragma package(smart_init)\n"
"#pragma resource \"*.dfm\"\n"
"T%f *%f;\n"
"//-----------------------------------------------------------------\n"
"__fastcall T%m::T%m(TComponent* Owner)\n"
"        : T%a(Owner)\n"
"{\n"
"}\n"
"//-----------------------------------------------------------------\n";
  return new File(expand(form_source, ModuleIdent, FormIdent,
                         AncestorIdent));
}
```

소스 코드에는 폼 %m 및 %y의 문자열이 들어 있습니다. 이 문자열은 특성상 printf 또는 Format 컨트롤과 비슷하지만, 마법사의 확장 함수를 사용하여 확장됩니다. 특히 %m은 모듈이나 유닛 식별자로 확장되고, %f는 폼 이름으로, %a는 조상 이름으로 확장됩니다. 대문자 T를 삽입하여 폼 이름이 폼의 타입 이름에서 어떻게 사용되는지 확인하십시오. 날짜를 나타내는 %d, 사용자를 나타내는 %u, 연도를 나타내는 %y와 같은 추가 형식 지정자를 사용하면 더 쉽게 주석 블록을 생성할 수 있습니다. 쓰기 확장은 Tools API와 관련이 없으며 읽는 사람을 위해 시험 삼아 남겨 둡니다.

*NewIntfSource*는 *NewImplSource*와 비슷하지만 인터페이스 파일(.h)을 생성합니다.

마지막 단계는 두 개의 폼 마법사 생성입니다. 하나는 sUnit을 작성자 타입으로 사용하고, 다른 하나는 sForm을 사용합니다. 사용자를 위해 추가된 이점으로는 *INTAServices*를 사용하여 File|New 메뉴에 각 마법사를 호출할 메뉴 항목을 추가할 수 있다는 것입니다. 메뉴 항목의 *OnClick* 이벤트 핸들러는 마법사의 *Execute* 함수를 호출할 수 있습니다.

일부 마법사는 IDE의 상황에 따라 메뉴 항목을 사용하거나 사용할 수 없도록 설정해야 합니다. 예를 들어, 소스 코드 제어 시스템으로 프로젝트를 체크 인하는 마법사는 IDE에 열려 있는 파일이 없을 경우 Check In 메뉴 항목을 사용할 수 없도록 설정해야 합니다. 통지자를 사용하여 이 기능을 마법사에 추가할 수 있습니다. 통지자 사용에 대한 내용은 다음 단원에서 다루겠습니다.

마법사에게 IDE 이벤트 통지

제대로 동작하는 마법사를 작성하기 위해서는 마법사가 IDE 이벤트에 응답하게 하는 단계가 중요합니다. 특히 모듈 인터페이스를 추적하는 마법사는 사용자가 모듈을 닫는 시기를 알아야 하며, 이 시기를 알면 인터페이스를 해제할 수 있습니다. 그러려면 마법사에 통지자가 필요합니다. 즉, 통지자 클래스를 작성해야 합니다.

모든 통지자 클래스는 하나 이상의 통지자 인터페이스를 구현합니다. 통지자 인터페이스는 콜백 메소드를 정의하고, 마법사는 Tools API를 사용하여 통지자 객체를 등록하며, IDE는 중요한 상황이 발생할 때 통지자를 콜백합니다.

해당 메소드가 모두 특정 통지자에 사용되지 않더라도 모든 통지자 인터페이스는 *IOTANotifier*에서 상속됩니다. 표 58.3은 모든 통지자 인터페이스를 나열한 것이며, 각 통지자에 대한 간단한 설명을 제공합니다.

표 58.3 통지자 인터페이스

인터페이스	설명
IOTANotifier	모든 통지자의 기본 추상 클래스
IOTABreakpointNotifier	디버거에서 브레이크포인트 실행 또는 변경
IOTADebuggerNotifier	디버거에서 프로그램을 실행하거나 브레이크포인트 추가 또는 삭제
IOTAEditLineNotifier	소스 데이터에서 줄의 이동 추적
IOTAEditorNotifier	소스 파일을 수정하거나 저장 또는 에디터에서 파일 전환
IOTAFormNotifier	폼 저장 또는 폼이나 폼(또는 데이터 모듈)에 있는 컴포넌트 수정
IOTAIDENotifier	프로젝트 로딩, 패키지 설치 및 기타 전역 IDE 이벤트
IOTAMessageNotifier	메시지 뷰에서 탭(메시지 그룹) 추가 또는 제거
IOTAModuleNotifier	모듈 변경, 저장 또는 이름 다시 지정
IOTAProcessModNotifier	디버거에서 프로세스 모듈 로딩
IOTAProcessNotifier	디버거에서 스레드 및 프로세스 생성 또는 소멸
IOTAThreadNotifier	디버거에서 스레드의 상태 생성
IOTAToolsFilterNotifier	도구 필터 호출

통지자 사용 방법을 확인하려면 이전 예제를 참조하십시오. 예제에서는 모듈 작성자를 사용하여 각 소스 파일에 주석을 추가하는 마법사를 만듭니다. 주석에는 유닛의 초기 이름이 포함되지만 사용자는 거의 항상 다른 이름으로 파일을 저장합니다. 이런 경우 마법사가 파일의 실제 이름과 일치시키기 위해 주석을 업데이트한다면 사용자에게 바람직할 것입니다.

그러려면 모듈 통지자가 필요합니다. 마법사는 *CreateModule*이 반환하는 모듈 인터페이스를 저장하고, 이를 사용하여 모듈 통지자를 등록합니다. 사용자가 파일을 수정하거나 저장하면 모듈 통지자가 통지를 받지만 마법사에게는 이러한 이벤트가 중요하지 않으므로 *AfterSave* 및 관련 함수의 바디는 모두 비어 있습니다. 사용자가 새 이름으로 파일을 저장할 때 IDE가 호출하는 *ModuleRenamed* 함수가 중요합니다. 모듈 통지자 클래스에 대한 선언은 아래와 같습니다.

```
class ModuleNotifier : public NotifierObject, public IOTAModuleNotifier
{
  typedef NotifierObject inherited;
public:
  __fastcall ModuleNotifier(const _di_IOTAModule module);
  __fastcall ~ModuleNotifier();

// IOTAModuleNotifier
  virtual bool __fastcall CheckOverwrite();
  virtual void __fastcall ModuleRenamed(const AnsiString NewName);

// IOTANotifier
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  _di_IOTAModule module;
  AnsiString name;        // Remember the module's old name.
  int index;              // Notifier index.
};
```

통지자를 작성하는 한 가지 방법은 해당 생성자에서 통지자가 자동으로 등록되게 하는 것입니다. 소멸자는 통지자 등록을 취소합니다. 모듈 통지자의 경우 사용자가 파일을 닫을 때 IDE가 *Destroyed* 메소드를 호출합니다. 이 경우에 통지자는 등록을 취소하고 모듈 인터페이스에 대한 참조를 해제해야 합니다. IDE는 통지자에 대한 IDE의 참조를 해제합니다. 그러면 참조 카운트가 0으로 줄어들고 객체가 해제됩니다. 따라서 소멸자를 수동적으로 작성해야 합니다. 통지자는 이미 등록이 취소되었을 수 있습니다.

```
__fastcall ModuleNotifier::ModuleNotifier(const _di_IOTAModule module)
: index(-1), module(module)
{
  // Register this notifier.
  index = module->AddNotifier(this);
  // Remember the module's old name.
  name = ChangeFileExt(ExtractFileName(module->FileName), "");
}

__fastcall ModuleNotifier::~ModuleNotifier()
{
  // Unregister the notifier if that hasn't happened already.
  if (index >= 0)
    module->RemoveNotifier(index);
}
```

```
void __fastcall ModuleNotifier::Destroyed()
{
  // The module interface is being destroyed, so clean up the notifier.
  if (index >= 0)
  {
    // Unregister the notifier.
    module->RemoveNotifier(index);
    index = -1;
  }
  module = 0;
}
```

사용자가 파일 이름을 다시 지정할 때 IDE가 통지자의 *ModuleRenamed* 함수를 콜백합니다. 함수는 마법사가 파일에서 주석을 업데이트하는 데 사용하는 새 이름을 매개변수로 사용합니다. 소스 버퍼를 편집하기 위해 마법사가 편집 위치 인터페이스를 사용합니다. 마법사가 올바른 위치를 찾아 올바른 텍스트를 찾았는지 두 번 확인하고, 해당 텍스트를 새 이름으로 바꿉니다.

```
void __fastcall ModuleNotifier::ModuleRenamed(const AnsiString NewName)
{
  // Get the module name from the new file name.
  AnsiString ModuleName = ChangeFileExt(ExtractFileName(NewName), "");
  for (int i = 0; i< module->GetModuleFileCount(); ++i)
  {
    // Update every source editor buffer.
    _di_IOTAEditor editor = module->GetModuleFileEditor(i);
    _di_IOTAEditBuffer buffer;
    if (Supports(editor, IID_IOTAEditBuffer, &buffer))
    {
      _di_IOTAEditPosition pos = buffer->GetEditPosition();
      // The module name is on line 2 of the comment.
      // Skip leading white space and copy the old module name,
      // to double check we have the right spot.
      pos->Move(2, 1);
      pos->MoveCursor(mmSkipWhite | mmSkipRight);
      AnsiString check = pos->RipText("", rfIncludeNumericChars |
rfIncludeAlphaChars);
      if (check == name)
      {
        pos->Delete(check.Length());    // Delete the old name.
        pos->InsertText(ModuleName);    // Insert the new name.
        name = ModuleName;              // Remember the new name.
      }
    }
  }
}
```

사용자가 추가 주석을 삽입할 경우 모듈 이름에 무엇이 사용될까요? 이 경우에 마법사가 편집 줄 통지자를 사용하여 모듈 이름이 있는 줄 번호를 추적해야 합니다. 그러려면 온라인 도움말에 설명되어 있는 *IOTAEditLineNotifier* 및 *IOTAEditLineTracker* 인터페이스를 사용하십시오.

통지자를 작성할 때 주의해야 합니다. 마법사보다 오래된 통지자가 없는지 확인해야 합니다. 예를 들어, 사용자가 마법사를 사용하여 새 유닛을 만들고 나서 마법사를 언로드할 경우 통지자가 여전히 유닛에 추가되어 있습니다. 결과를 예측할 수는 없지만 대부분 IDE 작동이 중지됩니다. 따라서 마법사는 모든 해당 통지자를 추적하여 마법사가 소멸되기 전에 모든 통지자의 등록을 취소해야 합니다. 반대로, 사용자가 먼저 파일을 닫을 경우 모듈 통지자는 *Destroyed* 통지를 받습니다. 즉, 통지자가 스스로 등록을 취소하고 모듈에 대한 모든 참조를 해제해야 합니다. 통지자는 마법사의 마스터 통지자 리스트에서도 자신을 제거해야 합니다.

다음은 최종 버전 마법사의 *Execute* 함수입니다. 이 함수는 새 모듈을 만들고, 모듈 인터페이스를 사용하여 모듈 통지자를 만든 다음, 인터페이스 리스트(*TInterfaceList*)에 모듈 통지자를 저장합니다.

```
void __fastcall DocWizard::Execute()
{
  _di_IOTAModuleServices svc;
  Supports(BorlandIDEServices, IID_IOTAModuleServices, &svc);
  _di_IOTAModule module = svc->CreateModule(new Creator(creator_type));
  _di_IOTAModuleNotifier notifier = new ModuleNotifier(module);
  list->Add(notifier);
}
```

마법사의 소멸자가 인터페이스 리스트에서 반복되고, 리스트에서 모든 통지자의 등록을 취소합니다. IDE에도 같은 인터페이스가 있으므로 인터페이스 리스트가 리스트에 있는 인터페이스를 해제하게 하는 것만으로는 부족합니다. IDE에 통지자 인터페이스를 해제하도록 알려 통지자 객체를 해제해야 합니다. 이 경우에 소멸자는 통지자로 하여금 모듈이 소멸되었다고 간주하게 만듭니다. 더 복잡한 상황에서는 통지자 클래스에 대한 Unregister 함수를 따로 작성하는 것이 가장 좋은 방법입니다.

```
__fastcall DocWizard::~DocWizard()
{
  // Unregister all the notifiers in the list.
  for (int i = list->Count; --i >= 0; )
  {
    _di_IOTANotifier notifier;
    Supports(list->Items[i], IID_IOTANotifier, &notifier);
    // Pretend the associated object has been destroyed.
    // That convinces the notifier to clean itself up.
    notifier->Destroyed();
    list->Delete(i);
  }
  delete list;
  delete item;
}
```

마법사의 나머지는 마법사 등록, 메뉴 항목 설치 등에 대한 실제적인 세부 사항을 관리합니다. 다음 단원에서는 패키지 대신 DLL에서 마법사를 등록하는 방법을 설명하여 여기에 대한 자세한 내용을 보다 깊이 있게 다룹니다.

마법사 DLL 설치

DLL에서도 마법사를 설치할 수 있습니다. 마법사가 동적 RTL과 런타임 패키지를 사용하는지 확인하십시오. Project Options에서 런타임 패키지 리스트에 디자인 패키지를 추가합니다. DLL은 이름이 INITWIZARD0001인 초기화 함수를 엑스포트해야 합니다. IDE는 DLL을 로드할 때 특수 엑스포트 이름을 찾아 해당 함수를 호출합니다. 모듈 정의 파일을 사용하여 함수를 엑스포트하거나 __declspec(dllexport) 선언을 사용할 수 있습니다. 후자의 경우 이름이 훼손되는 것을 피하기 위해 함수를 외부 "C"로 선언해야 합니다.

함수는 *TWizardInitProc* 타입이어야 합니다. 이 함수는 등록 함수에 대한 포인터를 해당 인수 중 하나로 사용합니다. 패키지 마법사가 *RegisterPackageWizard*를 호출하는 것과 같은 방법으로 인수가 이 함수를 호출하여 각 마법사 객체를 등록합니다. 초기화 함수는 성공일 경우 true를 반환하고, 실패일 경우 false를 반환해야 합니다. 다음 예제는 초기화 함수 작성 방법을 나타낸 것입니다.

```
extern "C" bool __stdcall __declspec(dllexport) INITWIZARD0001(
  const _di_IBorlandIDEServices,
  TWizardRegisterProc RegisterProc,
  TWizardTerminateProc&)
{
  RegisterProc(new MyWizard());
  RegisterProc(new MyOtherWizard());
  return true;
}
```

런타임 패키지를 사용할 경우 초기화 함수에 대한 첫 번째 인수는 중요하지 않습니다. 첫 번째 매개변수에 대해서는 58-22페이지의 "런타임 패키지 없이 DLL 사용"에서 설명합니다. 마지막 인수는 종료 프로시저의 함수 포인터에 대한 참조입니다. 이 함수를 사용하여 전체 클린업을 수행합니다. 대개 해당 소멸자에서 마법사가 자신 다음에 클린업해야 하므로 마지막 매개변수를 무시해도 좋습니다.

DLL을 설치하려면 다음 키 아래의 레지스트리에 항목을 추가하십시오.

```
HKEY_CURRENT_USER\Software\Borland\C++Builder\6.0\Experts
```

항목 이름은 마법사의 ID 문자열과 같이 고유한 이름이어야 합니다. 값은 DLL의 전체 경로여야 합니다. 다음에 C++Builder를 시작하면 레지스트리를 검사하고 Experts 키 아래에 나열된 모든 DLL을 로드합니다. IDE가 실행되는 동안 DLL은 계속 로드되어 있습니다. C++Builder가 종료될 때 DLL을 언로드합니다. 개발 도중에는 디자인 타임 패키지가 더 적합하기 때문에 이 작업으로 인해 디버깅 속도라 느려집니다. 마법사를 릴리스할 준비가 되면 디자인 타임 패키지를 DLL로 변경할 수 있습니다.

C++Builder에서 마법사의 버전이 잘못되었다고 표시하는 경우에는 INITWIZARD0001 함수를 찾을 수 없다는 의미입니다. 이름 철자가 맞는지, 이름이 손상되지 않고 함수를 올바로 엑스포트했는지 다시 확인하십시오.

패키지 대신 DLL을 사용하는 주된 장점은 이름 충돌 문제를 피할 수 있다는 것입니다. 다른 마법사에서 사용한 것과 똑같은 이름을 사용할 염려 없이 DLL에 필요한 폼을 포함할 수 있습니다. DLL의 또 다른 장점은 C++Builder의 버전에 관계 없이 DLL을 디자인할 수 있다는 것입니다. 그러나 특정 버전에 의존하도록 DLL을 디자인할 경우 런타임 패키지를 전혀 사용할 수 없습니다. 이 내용은 다음 단원의 주제입니다.

런타임 패키지 없이 DLL 사용

마법사 DLL과 함께 런타임 패키지를 사용할 필요가 없습니다. 런타임 패키지 사용의 가장 큰 장점은 DLL이 작다는 것입니다. IDE로 로드될 때만 마법사가 유용하기 때문에 C++Builder 패키지가 사용 가능하며, 작은 DLL만 배포해야 합니다. 반면에 패키지는 버전 특정적입니다. 여러 버전의 C++Builder와 Delphi에서 작동하는 마법사를 배포하려면 런타임 패키지를 사용해야 합니다.

BorlandIDEServices 변수가 디자인 패키지에만 정의되어 있으므로 프로그래머는 이 변수를 사용할 수 없습니다. 대신 초기화 함수에 전달되는 첫 번째 매개변수를 사용해야 합니다. 이때 값이 같아집니다. 예를 들어, 마법사는 이 값을 저장하고 Tools API 서비스에서 사용해야 합니다.

```
extern "C" bool __stdcall __declspec(dllexport) INITWIZARD0001(
  const _di_IBorlandIDEServices svc,
  TWizardRegisterProc reg,
  TWizardTerminateProc&)
{
  LocalIDEServices = svc;
  reg(new DocWizard(sUnit));
  reg(new DocWizard(sForm));
  return true;
}
AnsiString __fastcall DocWizard::GetDesigner()
{
  _di_IOTAServices svc;
  Supports(LocalIDEServices, IID_IOTAServices, &svc);
  return svc->GetActiveDesignerType();
}
```

Tools API의 디자인은 DLL이 계속 C++Builder의 새 릴리스용 함수가 되도록 보장합니다. 모든 이전 인터페이스에는 해당 GUID가 있으며, 새 인터페이스는 새 GUID를 보유합니다. Tools API가 버전 번호와 GUID를 사용하는 방법에 대한 자세한 내용은 58-11페이지의 "인터페이스 버전 번호"를 참조하십시오.

버전 독립적이 되려면 마법사가 원시(NTA) 인터페이스를 사용하면 안됩니다. 그러면 마법사의 기능이 제한됩니다. 가장 엄격한 제한은 마법사가 메뉴 표시줄 및 다른 IDE 객체에 액세스할 수 없다는 점입니다. 대부분의 Tools API 인터페이스는 OTA 인터페이스이므로 마법사 설치 방법과 관계 없이 유용하고 흥미로운 마법사를 계속 작성할 수 있습니다.

부록

A

ANSI 구현 특정 표준

ANSI C 표준의 특정 측면은 명시적으로 정의되지 않습니다. 대신 C 컴파일러의 각 구현에서 이러한 측면을 개별적으로 정의할 수 있습니다. 이 장에서는 Borland에서 이러한 구현 특정 세부 사항을 정의한 방법에 대해 설명합니다. 단원 번호는 1990년 2월 C ANSI/ISO 표준을 나타냅니다.

C와 C++ 간에 차이점이 있다는 것에 주의합니다. 여기서는 C에 대해서만 설명합니다. C++ 규정 준수에 대한 자세한 내용은 community.borland.com/cpp에서 Borland Community 웹 사이트를 참조하십시오.

2.1.1.3 진단 식별 방법

옵션을 제대로 조합하여 컴파일러를 실행하면 컴파일러가 표시하는 메시지 중에서 *Fatal*, *Error* 또는 *Warning*으로 시작하는 모든 메시지는 ANSI 규정에 따른 진단입니다. ANSI 규정 준수에 필요한 옵션은 다음과 같습니다.

표 A.1 ANSI 규정 준수에 필요한 옵션

Option	동작
–A	ANSI 키워드만 활성화합니다.
–C–	중첩된 주석을 허용하지 않습니다.
–i32	식별자의 최대 유효 문자 수를 32개로 합니다.
–p–	C 호출 규칙을 사용합니다.
–w–	모든 경고를 표시하지 않습니다.
–wbei	부적절한 이니셜라이저에 대한 경고를 표시합니다.
–wbig	너무 큰 상수에 대한 경고를 표시합니다.
–wcpt	이식할 수 없는 포인터 비교에 대한 경고를 표시합니다.
–wdcl	타입 또는 저장소 클래스가 없는 선언에 대한 경고를 표시합니다.
–wdup	중복된 서로 다른 매크로 정의에 대한 경고를 표시합니다.

표 A.1 ANSI 규정 준수에 필요한 옵션(계속)

Option	동작
–wext	외부와 정적 모두로 선언된 변수에 대한 경고를 표시합니다.
–wfdt	Typedef를 사용하는 함수 정의에 대한 경고를 표시합니다.
–wrpt	이식할 수 없는 포인터 변환에 대한 경고를 표시합니다.
–wstu	정의되지 않은 구조에 대한 경고를 표시합니다.
–wsus	의심스러운 포인터 변환에 대한 경고를 표시합니다.
–wucp	부호 있는 및 부호 없는 char에 대한 포인터를 혼합하는 것에 대한 경고를 표시합니다.
–wvrt	값을 반환하는 void 함수에 대한 경고를 표시합니다.

여기서 특별히 언급하지 않은 다른 옵션은 원하는 대로 설정할 수 있습니다.

2.1.2.2.1 메인에 대한 인수의 의미론
프로그램이 DOS에서 실행되면 *argv*[0]은 프로그램 이름을 가리킵니다.

나머지 *argv* 문자열은 DOS 명령줄 인수의 각 컴포넌트를 가리킵니다. 인수를 분리하는 공백은 제거되고 일련의 연속된 비공백 문자는 각각 단일 인수로 처리됩니다. 인용 문자열은 공백을 포함한 하나의 문자열로서 제대로 처리됩니다.

2.1.2.3 대화형 장치를 구성하는 요소
대화형 장치는 콘솔처럼 보이는 모든 장치입니다.

2.2.1 실행 문자 집합의 정렬 시퀀스
실행 문자 집합의 정렬 시퀀스는 ASCII의 문자 값을 사용합니다.

2.2.1 소스 및 실행 문자 집합의 멤버
소스 및 실행 문자 집합은 IBM PC에서 지원하는 확장 ASCII 집합입니다. *Ctrl+Z*가 아닌 모든 문자는 문자열 리터럴, 문자 상수 또는 주석에 표시될 수 있습니다.

2.2.1.2 멀티바이트 문자
C++Builder는 멀티바이트 문자를 지원합니다.

2.2.2 인쇄 방향
인쇄는 PC의 정상 방향인 왼쪽에서 오른쪽으로 이루어집니다.

2.2.4.2 실행 문자 집합에 있는 문자의 비트 수
실행 문자 집합에는 문자당 8비트가 존재합니다.

3.1.2 식별자의 유효 초기 문자 수
처음 250개의 문자가 유효합니다. 단, 명령줄 옵션(**–i**)을 사용하여 이 문자 수를 변경할 수 있습니다. 내부 및 외부 식별자 모두에서 동일한 수의 유효 문자를 사용합니다. C++ 식별자에서는 유효 문자 수에 제한을 받지 않습니다.

3.1.2 외부 식별자에서 대소문자 구별이 유효한지 여부

일반적으로 컴파일러는 링커에서 대소문자를 구별하도록 강제합니다. 명령줄 컴파일러 옵션(**–lc–**)을 사용하면 대소문자 구별 기능을 끌 수 있습니다. IDE에서는 Project|Options|Advanced Linker를 선택하고 Case-insensitive link를 선택할 수도 있습니다.

3.1.2.5 다양한 정수 타입의 표현 및 값 집합

표 A.2 C++에서의 진단 식별

타입	최소값	최대값
부호 있는 char	–128	127
부호 없는 char	0	255
부호 있는 short	–32,768	32,767
부호 없는 short	0	65,535
부호 있는 int	–2,147,483,648	–2,147,483,647
부호 없는 int	0	4,294,967,295
부호 있는 long	–2,147,483,648	2,147,483,647
부호 없는 long	0	4,294,967,295

모든 **char** 타입은 저장할 때 1바이트(8비트)를 사용합니다.

모든 **short** 타입은 2바이트를 사용합니다.

모든 **int** 타입은 4바이트를 사용합니다.

모든 **long** 타입은 4바이트를 사용합니다.

정렬이 요청된 경우(**–a**), **char**가 아닌 모든 정수 타입 객체는 짝수 바이트 경계로 정렬됩니다. 요청된 정렬이 **–a4**이면 결과는 4바이트 정렬이 됩니다. 문자 타입은 정렬되지 않습니다.

3.1.2.5 다양한 부동 소수점 숫자 타입의 표현 및 값 집합

Intel 8086에서 사용하는 것과 같은 IEEE 부동 소수점 형식이 모든 C++Builder 부동 소수점 타입에 사용됩니다. **float** 타입은 32비트 IEEE 실수 형식을 사용하고 **double** 타입은 64비트 IEEE 실수 형식을 사용합니다. **long double** 타입은 80비트 IEEE 확장 실수 형식을 사용합니다.

3.1.3.4 소스 및 실행 문자 집합 간의 매핑

문자열 리터럴 또는 문자 상수의 모든 문자는 실행 프로그램에서 바뀌지 않습니다. 따라서 소스 및 실행 문자 집합은 동일합니다.

3.1.3.4 와이드 문자 상수의 기본 실행 문자 집합 또는 확장 문자 집합에 나타나지 않는 문자 또는 이스케이프 시퀀스를 포함하는 정수 문자 상수의 값

와이드 문자가 지원됩니다.

3.1.3.4 멀티바이트 문자를 와이드 문자 상수의 해당 와이드 문자로 변환하는 데 사용되는 현재 로케일

와이드 문자 상수가 인식됩니다.

3.1.3.4 둘 이상의 문자를 포함하는 정수 상수나 둘 이상의 멀티바이트 문자를 포함하는 와이드 문자 상수의 값

문자 상수는 하나 이상의 문자를 포함할 수 있습니다. 문자 두 개가 포함된 경우, 첫 번째 문자는 상수의 하위 바이트를 차지하고 두 번째 문자는 상위 바이트를 차지합니다.

3.2.1.2 정수를 더 짧은 부호 있는 정수로 변환한 결과 또는 값을 나타낼 수 없는 경우 부호 없는 정수를 동일한 길이의 부호 있는 정수로 변환한 결과

이러한 변환은 단순히 상위 바이트를 잘라내는 방법으로 수행됩니다. 부호 있는 정수는 2의 보수 값으로 저장되므로 결과 숫자는 이러한 값으로 해석됩니다. 이 값은 더 작은 정수의 상위 비트가 0이 아니면 음수로 해석되고, 그렇지 않으면 양수로 해석됩니다.

3.2.1.3 정수가 원래 값을 정확하게 나타낼 수 없는 부동 소수점 숫자로 변환될 경우의 잘라내기 방향

정수는 나타낼 수 있는 가장 가까운 값으로 반올림됩니다. 따라서 **long** 값인 $(2^{31} – 1)$의 경우는 **float** 값인 2^{31}로 변환됩니다. 연결은 IEEE 표준 산술 규칙에 따라 끊어집니다.

3.2.1.4 부동 소수점 숫자가 더 좁은 부동 소수점 숫자로 변환될 경우의 잘라내기 또는 반올림 방향

값은 나타낼 수 있는 가장 가까운 값으로 반올림됩니다. 연결은 IEEE 표준 산술 규칙에 따라 끊어집니다.

3.3 부호 있는 정수에서의 비트 단위 작업 결과

비트 단위 연산자는 부호 없는 해당 타입인 것처럼 부호 있는 정수에 적용됩니다. 부호 비트는 일반적인 데이터 비트로 처리됩니다 . 그런 다음 결과는 부호 있는 일반적인 2 의 보수 정수로 해석됩니다.

3.3.2.3 합집합 객체의 멤버를 다른 타입의 멤버를 사용하여 액세스할 경우의 처리 방법

이 액세스는 허용되며 다른 타입 멤버는 합집합 객체 멤버에 저장된 비트를 액세스합니다. 다른 멤버를 사용하여 부동 소수점 타입 멤버를 액세스하는 방법을 이해하기 위해서는 부동 소수점 값의 비트 인코딩을 자세하게 알아야 합니다. 저장된 멤버가 값을 액세스하는 데 사용되는 멤버보다 짧을 경우, 초과 비트는 짧은 멤버가 저장되기 전에 가졌던 값을 가집니다.

3.3.3.4 배열의 최대 크기를 유지하는 데 필요한 정수 타입

이 타입은 일반적인 배열의 경우는 **부호 없는 int**이고 대규모 배열의 경우는 **부호 있는 long**입니다.

3.3.4 포인터를 정수로, 또는 그 반대로 타입 변환한 결과

동일한 크기의 정수 및 포인터 간에 변환하면 비트가 변경되지 않습니다. 더 긴 타입에서 더 짧은 타입으로 변환하면 상위 비트가 잘립니다. 더 짧은 정수 타입에서 더 긴 포인터 타입으로 변환하면 우선 정수가 포인터 타입과 동일한 크기의 정수 타입으로 확장됩니다.

따라서 부호 있는 정수는 부호가 확장되어 새 바이트를 채웁니다. 마찬가지로 더 큰 정수 타입으로 변환되는 더 작은 포인터 타입은 우선 정수 타입과 동일한 크기의 포인터 타입으로 확장됩니다.

3.3.5 정수 나누기의 나머지 부호

피연산자 중 하나만 음수이면 나머지의 부호는 음수입니다. 둘 다 음수이거나, 음수가 아니면 나머지는 양수입니다.

3.3.6 동일한 배열인 ptrdiff_t의 요소에 대한 두 포인터 간의 차이점을 보유하는 데 필요한 정수 타입
이 타입은 부호 있는 **int**입니다.

3.3.7 부호 있는 음수 정수 타입의 오른쪽 시프트 결과
부호 있는 음수 값은 오른쪽으로 시프트되면 부호가 확장됩니다.

3.5.1 *Register* 저장소 클래스 지정자를 사용하여 객체를 실제로 레지스터에 둘 수 있는 범위
임의의 1, 2 또는 4바이트 정수 또는 포인터 타입을 사용하여 선언한 객체는 레지스터에 둘 수 있습니다. 최소 2개에서 최대 7개까지의 레지스터를 사용할 수 있습니다. 실제로 사용되는 레지스터 수는 함수의 임시 값에 필요한 레지스터에 따라 다릅니다.

3.5.2.1 일반적인 int 비트 필드가 부호 있는 int로 처리되는지, 아니면 부호 없는 int 비트 필드로 처리되는지 여부
일반적인 **int** 비트 필드는 **부호 있는 int** 비트 필드로 처리됩니다.

3.5.2.1 int 내에서 비트 필드의 할당 순서
비트 필드는 하위 비트 위치에서 상위 비트 위치로 할당됩니다.

3.5.2.1 구조 멤버의 채우기 및 정렬
디폴트로, 구조에서는 채우기가 수행되지 않습니다. 단어 정렬 옵션인 (**–a**)를 사용할 경우, 구조는 동일한 크기로 채워지고 문자 또는 문자 배열 타입이 없는 모든 멤버는 동일한 다중 오프셋으로 정렬됩니다.

3.5.2.1 비트 필드가 저장소 유닛 경계를 늘릴 수 있는지 여부
정렬(**–a**)이 요청되지 않으면 비트 필드는 dword 경계를 늘릴 수 있지만 5개 이상의 인접 바이트에 저장되지 않습니다.

3.5.2.2 열거 타입의 값을 나타내기 위해 선택하는 정수 타입
모든 열거를 완전한 **int**로 저장합니다. 값이 **int**에 맞지 않으면 열거를 **long** 또는 **부호 없는 long**에 저장합니다. 이것은 **–b** 컴파일러 옵션에 지정된 대로 디폴트 동작입니다.

이 **–b-** 동작은 값을 나타낼 수 있는 가장 작은 정수 타입에 열거를 저장하도록 지정합니다. 여기에는 모든 정수 타입(예: **부호 있는 char, 부호 없는 char, 부호 있는 short, 부호 없는 short, 부호 있는 int, 부호 없는 int, 부호 있는 long, 부호 없는 long** 등)이 포함됩니다.

C++의 경우 모든 열거를 **int**로 저장하는 것은 옳지 않으므로 C++ 규정을 준수하려면 **–b-**를 지정해야 합니다.

3.5.3 volatile 한정 타입을 가진 객체에 대한 액세스를 구성하는 요소
volatile 객체에 대한 모든 참조는 해당 객체를 액세스합니다. 인접 메모리 위치 액세스도 객체를 액세스할 것인지 여부는 하드웨어에서의 메모리 구성 방법에 따라 달라집니다. 비디오 디스플레이 메모리와 같은 특수한 장치 메모리의 경우에는 장치 구성 방법에 따라 달라집니다. 일반적인 PC 메모리의 경우, volatile 객체는 비동기 중단이 액세스할 수 있는 메모리에만 사용되므로 인접 객체 액세스가 영향을 미치지 않습니다.

3.5.4 산술, 구조 또는 합집합 타입을 수정할 수 있는 최대 선언자 수
선언자 수는 특별히 제한을 받지 않습니다. 허용되는 선언자 수는 상당히 크지만, 함수의 블록 집합 내에서 깊게 중첩된 경우에는 선언자 수가 줄어듭니다. 파일 레벨에서 허용되는 수는 최소 50개 이상입니다.

3.6.4.2 switch 문의 최대 case 값 수
switch 문의 case 수는 특별히 제한을 받지 않습니다. Case 정보를 보유할 수 있는 충분한 메모리가 있다면 컴파일러는 모든 case를 승인합니다.

3.8.1 조건부 포함을 제어하는 상수 표현식의 단일 문자 상수의 값이 실행 문자 집합의 동일한 문자 상수의 값과 일치하는지 여부와 이러한 문자 상수가 음수 값을 가질 수 있는지 여부
조건부 지시어에 있는 경우를 비롯하여 모든 문자 상수는 동일한 문자 집합(실행)을 사용합니다. 문자 타입이 signed(기본값이고 **–K**가 요청되지 않음)이면 단일 문자 상수는 음수가 됩니다.

3.8.2 포함할 수 있는 소스 파일을 찾는 방법
꺽쇠 괄호로 둘러싼 include 파일 이름의 경우, 명령줄에서 include 디렉토리가 지정되면 각 include 디렉토리에서 이 파일을 검색합니다. Include 디렉토리는 다음 순서로 검색됩니다.

1 명령줄에 지정된 디렉토리

2 BCC32.CFG에 지정된 디렉토리

3 지정된 include 디렉토리가 없는 경우, 현재 디렉토리만 검색

3.8.2 포함할 수 있는 소스 파일에 대한 인용 이름 지원
포함할 수 있는 인용 파일 이름의 경우, 파일은 다음 순서로 검색됩니다.

1 **#include** 문이 포함된 파일과 동일한 디렉토리

2 해당 파일을 포함한(**#include**) 파일의 디렉토리

3 현재 디렉토리

4 **/I** 컴파일러 옵션으로 지정된 경로

5 INCLUDE 환경 변수로 지정된 경로

3.8.2 소스 파일 이름 문자 시퀀스의 매핑
Include 파일 이름의 백슬래시는 이스케이프 문자가 아니라 고유한 문자로 처리됩니다. 대소문자 차이는 무시됩니다.

3.8.8 __DATE__ 및 __TIME__을 사용할 수 없는 경우의 이러한 매크로에 대한 정의
날짜와 시간은 항상 사용할 수 있습니다. 운영 체제 날짜 및 시간이 사용됩니다.

4.1.1 소수점 문자
소수점 문자는 마침표(.)입니다.

4.1.5 Sizeof 연산자의 타입인 *size_t*
size_t 타입은 ***부호 없는***입니다.

4.1.5 NULL 매크로가 확장되는 Null 포인터 상수

NULL은 **int** 0 또는 **long** 0으로 확장됩니다. 둘 다 32비트 **부호 있는** 숫자입니다.

4.2 Assert 함수가 출력하는 진단과 이 함수의 종료 동작

"Assertion failed: *expression*, file *filename*, line *nn*"라는 진단 메시지가 출력됩니다. 여기서 *expression*은 실패한 assert 표현식이고 *filename*은 소스 파일 이름이며 *nn*은 assertion이 발생한 줄 번호입니다.

Assertion 메시지가 표시된 직후 **abort**가 호출됩니다.

4.3 문자 테스트 및 대소문자 매핑 함수의 구현 정의 측면

4.3.1에서 언급된 함수 외에는 이러한 측면이 없습니다.

4.3.1 Isalnum, isalpha, iscntrl, islower, isprint 및 isupper 함수에 의해 테스트되는 문자 집합

디폴트 C 로케일의 경우 처음 ASCII 문자 128개가 테스트됩니다. 이외의 경우에는 모든 문자 256개가 테스트됩니다.

4.5.1 도메인 오류 시 수치 연산 함수에 의해 반환되는 값

IEEE NAN(숫자가 아님)입니다.

4.5.1 언더플로 범위 오류 시 수치 연산 함수가 정수 표현식인 *errno*를 ERANGE 매크로의 값으로 설정하는지 여부

도메인, 단수, 오버플로, 정밀도 전체 손실 등의 다른 오류에 대해서만 설정이 이루어집니다.

4.5.6.4 Fmod 함수가 *0*인 또 다른 인수를 가질 경우 도메인 오류가 발생하는지, 아니면 0이 반환되는지 여부

Fmod(x,0)는 0을 반환합니다.

4.7.1.1 Signal 함수의 신호 집합

SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM 등이 포함됩니다.

4.7.1.1 Signal 함수가 인식하는 각 신호의 의미론

신호에 대한 설명을 참조하십시오.

4.7.1.1 Signal 함수가 인식하는 각 신호에 대한 디폴트 처리 및 프로그램 시작 시의 처리

신호에 대한 설명을 참조하십시오.

4.7.1.1 신호 핸들러 호출 이전에 signal(sig, SIG_DFL)과 동일한 결과를 생성하는 함수를 실행하지 않은 경우 수행되는 신호 차단

signal(sig, SIG_DFL)과 동일한 결과를 생성하는 함수는 항상 실행됩니다.

4.7.1.1 Signal 함수에 지정된 핸들러가 SIGILL 신호를 받을 경우 디폴트 처리가 다시 설정되는지 여부

디폴트 처리는 다시 설정되지 않습니다.

4.9.2 텍스트 스트림의 마지막 줄에 새 줄 문자가 필요한지 여부

따로 필요한 문자는 없습니다.

4.9.2 새 줄 문자 바로 앞에서 텍스트 스트림에 쓰여진 공백 문자가 읽을 때 표시되는지 여부
공백 문자는 표시됩니다.

4.9.2 바이너리 스트림에 쓰여진 데이터에 추가할 수 있는 Null 문자 수
Null 문자를 추가할 수 없습니다.

4.9.3 Append 모드 스트림의 파일 위치 지시자가 초기에 파일 시작 부분에 놓이는지, 아니면 끝 부분에 놓이는지 여부
Append 모드 스트림의 파일 위치 지시자는 초기에 파일의 시작 부분에 놓입니다. 이 지시자는 각 쓰기 작업이 수행되기 전에 파일의 끝 부분으로 다시 설정됩니다.

4.9.3 텍스트 스트림에서의 쓰기로 인해 연결된 파일이 해당 지점 이후에 잘리는지 여부
0바이트 쓰기는 파일의 버퍼링 방법에 따라 파일을 자르거나 자르지 않을 수 있습니다. 길이가 0인 쓰기를 확정되지 않은 동작을 가진 것으로 분류하는 것이 가장 안전합니다.

4.9.3 파일 버퍼링 특성
파일은 완전하게 버퍼링하거나, 줄 버퍼링하거나, 버퍼링하지 않을 수 있습니다. 파일을 버퍼링하면 파일을 열 때 512바이트의 디폴트 버퍼가 만들어집니다.

4.9.3 길이가 0인 파일이 실제로 존재하는지 여부
실제로 존재합니다.

4.9.3 동일한 파일을 여러 번 열 수 있는지 여부
여러 번 열 수 있습니다.

4.9.4.1 Remove 함수가 열려진 파일에 미치는 영향
이미 열려 있는 파일에는 특별한 검사가 수행되지 않습니다. 필요한 작업은 프로그래머가 수행합니다.

4.9.4.2 Rename 함수를 호출하기 전에 새 이름을 가진 파일이 존재하는 경우 수행되는 작업
*Rename*이 –1을 반환하고 *errno*가 EEXIST로 설정됩니다.

4.9.6.1 Fprintf의 %p 변환에 대한 출력
출력은 8자리의 16진수(XXXXXXXX)로서 0으로 채워진 대문자입니다(%08lX와 동일).

4.9.6.2 Fscanf의 %p 변환에 대한 입력
4.9.6.1을 참조하십시오.

4.9.6.2 Fscanf의 %[변환에 대한 검색 리스트에서 처음 또는 마지막 문자가 아닌 –(하이픈) 문자의 해석
Scanf에 대한 설명을 참조하십시오.

4.9.9.1 실패 시 fgetpos 또는 ftell 함수에 의해 errno 매크로가 설정되는 값
EBADF 잘못된 파일 번호입니다.

4.9.10.4 Perror가 생성하는 메시지

Win32에서 생성되는 메시지	
Arg list too big	Math argument
Attempted to remove current directory	Memory arena trashed
Bad address	Name too long
Bad file number	No child processes
Block device required	No more files
Broken pipe	No space left on device
Cross-device link	No such device
Error 0	No such device or address
Exec format error	No such file or directory
Executable file in use	No such process
File already exists	Not a directory
File too large	Not enough memory
Illegal seek	Not same device
Inappropriate I/O control operation	Operation not permitted
Input/output error	Path not found
Interrupted function call	Permission denied
Invalid access code	Possible deadlock
Invalid argument	Read-only file system
Invalid data	Resource busy
Invalid environment	Resource temporarily unavailable
Invalid format	Result too large
Invalid function number	Too many links
Invalid memory block address	Too many open files
Is a directory	

4.10.3 요청된 크기가 0인 경우 calloc, malloc 또는 realloc의 동작

Calloc 및 malloc은 요청을 무시하고 0을 반환합니다. Realloc은 블록을 해제합니다.

4.10.4.1 열려 있는 파일 및 임시 파일과 관련된 abort 함수의 동작

파일 버퍼가 플러시되지 않고 파일이 닫히지 않습니다.

4.10.4.3 인수 값이 0, EXIT_SUCCESS 또는 EXIT_FAILURE가 아닌 경우 exit 함수가 반환하는 상태

특별한 상태가 반환되는 것은 아닙니다. 상태는 전달된 그대로 반환되고 **부호 있는 char**로 나타납니다.

4.10.4.4 환경 이름 집합과 getenv에 사용되는 환경 리스트를 변경하는 방법

환경 문자열은 SET 명령을 통해 운영 체제에서 정의됩니다. *Putenv*를 사용하면 현재 프로그램 도중에 환경 문자열을 변경할 수 있지만, 이러한 문자열을 영구적으로 변경하려면 SET 명령을 사용해야 합니다.

4.10.4.5 시스템 함수에 의한 문자열 실행의 내용과 모드

이 문자열은 운영 체제 명령으로 해석됩니다. COMSPEC 또는 CMD.EXE가 사용되고 인수 문자열은 실행할 명령으로서 전달됩니다. 모든 기본 운영 체제 명령과 배치 파일 및 실행 가능 프로그램을 실행할 수 있습니다.

4.11.6.2 Strerror가 반환하는 오류 메시지 문자열의 내용

4.9.10.4를 참조하십시오.

4.12.1 현지 표준 시간대와 일광 절약 시간

로컬 PC 시간 및 날짜로 정의됩니다.

4.12.2.1 클럭 시간

프로그램 실행과 함께 시작되는 클럭 작동 시간을 나타냅니다.

4.12.3.5 날짜 및 시간 형식

C++Builder는 ANSI 형식을 구현합니다.

부록

B

WebSnap 서버사이드 스크립트 참조

이 부록에서는 WebSnap 어댑터가 WebSnap 웹 서버 애플리케이션에서 동적 HTML 페이지를 생성하기 위해 스크립트를 사용하는 방법을 설명합니다. 이 부록은 웹 페이지 모듈에서 어댑터 페이지 프로듀서 대신 페이지 프로듀서를 사용하는 개발자를 위한 것입니다. 어댑터 페이지 프로듀서는 자동으로 스크립트 생성을 처리하고 일반적인 페이지 프로듀서는 페이지 템플릿에 스크립트를 수동으로 추가해야 합니다. 여기에 포함된 정보는 페이지 템플릿에 고유의 스크립트를 작성하는 데 유용합니다.

또한 이 부록은 어댑터 페이지 프로듀서 사용자가 어댑터 페이지 프로듀서의 출력을 보다 잘 이해할 수 있도록 도와 줍니다. 단, 스크립트 조작은 고급 기능입니다. 스크립트 작성 방법을 이해하지 않고도 기본적인 WebSnap 애플리케이션을 작성할 수 있습니다.

이 부록에는 세 개의 단원이 있습니다. 첫 번째 단원에서는 스크립트에서 액세스할 수 있는 다양한 객체 타입에 대해 설명합니다. 두 번째 단원에서는 WebSnap 애플리케이션에서 정의된 전역 객체에 대해 설명합니다. 세 번째 단원에는 HTML 페이지 템플릿에서 스크립트를 사용하여 웹 서버 애플리케이션으로부터 정보를 추출하는 방법을 보여주는 JScript 예제가 포함되어 있습니다.

이 부록은 객체의 스크립트 인터페이스에 대한 API 참조입니다. 객체 속성 설명에서는 속성 이름, 텍스트 또는 부울과 같은 속성 타입 및 속성을 읽거나 쓸 수 있는지 여부에 대해 설명합니다. 메소드 설명은 메소드 이름 및 호출 구문으로 시작됩니다.

객체 타입

표 58.4에는 스크립팅이 가능한 일반적인 객체 타입이 나열되어 있습니다. 이러한 타입은 일반적으로 전역 객체와 같은 객체의 속성으로 표시됩니다. 이 표에서 스크립팅이 가능한 모든 객체 타입을 다루는 것은 아니며, 인스턴스 이름이 달라질 수 있는 객체 타입에 대해서만 설명합니다. 예를 들어, 모든 애플리케이션에서 애플리케이션 객체 타입은 단 한 번만 인스턴스화되므로 애플리케이션 타입은 전역 객체 단원에서 Application 객체로 설명됩니다.

표 58.4 WebSnap 객체 타입

객체 타입	설명
Adapter 타입 (B-2페이지)	어댑터의 속성과 메소드를 정의합니다. 어댑터는 이름을 사용하여 모듈의 한 속성으로 액세스할 수 있습니다.
AdapterAction 타입 (B-4페이지)	어댑터 액션의 속성과 메소드를 정의합니다. 액션은 이름을 사용하여 어댑터의 한 속성으로 참조됩니다.
AdapterErrors 타입 (B-6페이지)	어댑터의 Errors 속성을 정의합니다. Errors 속성은 액션을 실행하거나 페이지를 생성할 때 발생하는 오류를 나열하는 데 사용됩니다.
AdapterField 타입 (B-6페이지)	어댑터 필드의 속성 및 메소드를 정의합니다. 필드는 이름을 사용하여 어댑터의 한 속성으로 참조됩니다.
AdapterFieldValues 타입 (B-10페이지)	어댑터 필드의 Values 속성의 속성 및 메소드를 정의합니다.
AdapterFieldValuesList 타입 (B-10페이지)	어댑터 필드의 ValuesList 속성의 속성 및 메소드를 정의합니다.
AdapterHiddenFields 타입 (B-11페이지)	어댑터의 HiddenFields 및 HiddenRecordFields 속성을 정의합니다.
AdapterImage 타입 (B-11페이지)	어댑터 필드와 어댑터 액션의 Image 속성을 정의합니다.
Module 타입 (B-12페이지)	모듈의 속성을 정의합니다. 모듈은 이름을 사용하여 Modules 변수의 한 속성으로 액세스할 수 있습니다.
Page 타입 (B-12페이지)	페이지의 속성을 정의합니다. 페이지는 이름에 의해 Pages 객체의 속성으로 액세스할 수 있습니다. 생성 중인 페이지는 Page 객체를 사용하여 액세스할 수 있습니다.

Adapter 타입

어댑터의 속성과 메소드를 정의합니다. 어댑터는 이름을 사용하여 모듈의 한 속성으로 액세스할 수 있습니다(예: ModuleName.Adapter).

어댑터에는 데이터 항목과 명령을 각각 나타내는 필드 컴포넌트 및 액션 컴포넌트가 포함됩니다. 서버사이드 스크립트 문은 어댑터 필드의 값과 어댑터 액션의 매개변수에 액세스하여 HTML 폼과 테이블을 생성합니다.

속성

Actions: 열거자

참고 항목: Adapter 타입의 *Fields* 속성(아래 참조), 예제 8(B-22페이지)

액션 객체를 열거합니다. 어댑터의 액션을 모두 반복하려면 *Actions* 속성을 사용하십시오.

CanModify: 부울, 읽기

참고 항목: Adapter 타입(아래 참조)의 AdapterField 타입(B-6페이지)의 *CanView* 속성

엔드 유저가 이 어댑터의 필드를 수정할 권한이 있는지 여부를 나타냅니다. 엔드 유저의 권한을 중시하는 HTML을 동적으로 생성하려면 CanModify 속성을 사용하십시오. 예를 들어, *CanModify*가 *True*인 경우 <input> 요소가 포함되고, *CanModify*가 *False*인 경우 <p> 요소가 페이지에 포함될 수 있습니다.

CanView: 부울, 읽기

참고 항목: Adapter 타입(위 참조) 및 AdapterField 타입(B-6페이지)의 *CanModify* 속성

엔드 유저가 이 어댑터의 필드를 볼 수 있는 권한이 있는지 여부를 나타냅니다. 엔드 유저의 권한을 중시하는 HTML을 동적으로 생성하려면 *CanModify* 속성을 사용하십시오.

ClassName_: 텍스트, 읽기

참고 항목: *Name_*(아래 참조)

어댑터 컴포넌트의 클래스 이름을 나타냅니다.

Errors: AdapterErrors, 읽기

참고 항목: AdapterErrors 타입(B-6페이지), 예제 7(B-22페이지)

HTTP 요청을 처리하는 동안 검색된 오류를 열거합니다. 어댑터가 HTML 페이지를 생성하거나 어댑터 액션을 실행하는 동안 발생하는 오류를 캡처합니다. 오류를 열거하고 HTML 페이지에 오류 메시지를 표시하려면 *Errors* 속성을 사용하십시오.

Fields: 열거자

참고 항목: *Actions*

필드 객체를 열거합니다. 어댑터의 필드를 반복하려면 Fields 속성을 사용하십시오.

HiddenFields: AdapterHiddenFields

참고 항목: *HiddenRecordFields*, AdapterHiddenFields 타입(B-11페이지), 예제 10(B-24페이지), 예제 22(B-37페이지)

어댑터 상태 정보를 전달하는 숨겨진 입력 필드를 정의합니다. 상태 정보의 한 예는 *TDataSetAdapter*의 모드입니다. Edit 및 Insert가 사용 가능한 두 모드 값입니다. HTML 폼을 생성하기 위해 *TDataSetAdapter*를 사용하는 경우 *HiddenFields* 속성이 모드에 대한 숨겨진 필드를 정의합니다. HTML 폼이 제출되는 경우 HTTP 요청에는 이 숨겨진 필드 값이 포함됩니다. 액션을 실행하는 경우 모드 값이 HTTP 요청으로부터 추출됩니다. 모드가 Insert이면 새 행이 데이터셋에 추가되며 모드가 Edit이면 데이터셋 행이 업데이트됩니다.

HiddenRecordFields: AdapterHiddenFields

참고 항목: *HiddenFields*, AdapterHiddenFields 타입(B-11페이지), 예제 10(B-24페이지), 예제 22(B-37페이지)

HTML 폼의 각 행 또는 레코드에 필요한 상태 정보를 전달하는 숨겨진 입력 필드를 정의합니다. 예를 들어, HTML 폼을 생성하기 위해 *TDataSetAdapter*를 사용하는 경우 *HiddenRecordFields* 속성이 HTML 테이블에 있는 각 행의 키 값을 나타내는 숨겨진 필드를 정의합니다. HTML 폼이 제출되는 경우 HTTP 요청에는 이 숨겨진 필드 값이 포함됩니다. 데이터셋의 여러 행을 업데이트하는 액션을 실행하는 경우 *TDataSetAdapter*가 이러한 키 값을 사용하여 업데이트할 행을 찾습니다.

Mode: 텍스트, 읽기/쓰기

참고 항목: 예제 10(B-24페이지)

어댑터의 모드를 설정하거나 가져옵니다.

일부 어댑터는 모드를 지원합니다. 예를 들어, *TDataSetAdapter*는 Edit, Insert, Browse 및 Query 모드를 지원합니다. 모드는 어댑터의 동작에 영향을 미칩니다. *TDataSetAdapter*가 Edit 모드이면 제출되는 폼이 테이블의 행을 업데이트합니다. *TDataSetAdapter*가 Insert 모드이면 제출되는 폼이 테이블에 행을 삽입합니다.

Name_: 텍스트, 읽기

어댑터의 변수 이름을 나타냅니다.

Records: 열거자, 읽기

참고 항목: 예제 9(B-23페이지)

어댑터의 레코드를 열거합니다. 어댑터 레코드를 반복하여 HTML 테이블을 생성하려면 Records 속성을 사용합니다.

AdapterAction 타입

참고 항목: Adapter 타입(B-2페이지), AdapterField 타입(B-6페이지)

AdapterAction 타입은 어댑터 액션의 속성 및 메소드를 정의합니다.

속성

Array: 열거자

참고 항목: 예제 11(B-26페이지)

어댑터 액션의 명령을 열거합니다. 명령을 반복하려면 *Array* 속성을 사용하십시오. 액션이 다중 명령을 지원하지 않으면 *Array*가 *Null*이 됩니다.

*TAdapterGotoPageAction*은 다중 명령이 있는 액션의 예입니다. 이 액션에는 부모 어댑터에 의해 정의된 각 페이지에 대한 명령이 있습니다. *Array* 속성은 엔드 유저가 하이퍼링크를 클릭하여 페이지로 이동할 수 있도록 일련의 하이퍼링크를 생성하는 데 사용됩니다.

AsFieldValue: 텍스트, 읽기

참고 항목: *AsHREF*, 예제 10(B-24페이지), 예제 21(B-36페이지)

숨겨진 필드에서 제출될 수 있는 텍스트 값을 제공합니다.

*AsFieldValue*는 액션 및 액션 매개변수의 이름을 나타냅니다. 이 값은 __act라는 숨겨진 필드에 두십시오. HTML 폼이 제출되는 경우 어댑터 디스패처가 HTTP 요청으로부터 값을 추출하고 이 값을 사용하여 어댑터 액션의 위치를 찾아 호출합니다.

AsHREF: 텍스트, 읽기

참고 항목: *AsFieldValue*, 예제 11(B-26페이지)

<a> 태그에서 href 어트리뷰트(attribute) 값으로 사용할 수 있는 텍스트 값을 제공합니다.

*AsHREF*는 액션 및 액션 매개변수의 이름을 나타냅니다. 요청을 제출하여 액션을 실행하려면 이 값을 anchor 태그에 두십시오. HTML 폼의 anchor 태그는 폼을 제출하지 않습니다. 액션이 제출된 폼 값을 사용하면 숨겨진 폼 필드와 *AsFieldValue*를 사용하여 액션을 표시하십시오.

CanExecute: 부울, 읽기

엔드 유저가 이 액션을 실행할 권한이 있는지 여부를 나타냅니다.

DisplayLabel: 텍스트, 읽기

참고 항목: 예제 21(B-36페이지)

이 어댑터 액션에 대한 HTML 표시 레이블을 제공합니다.

DisplayStyle: 문자열, 읽기

참고 항목: 예제 21(B-36페이지)

이 액션에 대한 HTML 표시 스타일을 제공합니다.

서버사이드 스크립트는 *DisplayStyle*을 사용하여 HTML 생성 방법을 결정할 수 있습니다. 기본 어댑터는 다음 표시 스타일 중 하나를 반환할 수 있습니다.

값	의미
''	정의되지 않은 표시 스타일
'Button'	<input type="submit">으로 표시
'Anchor'	<a> 사용

Enabled: 부울, 읽기

참고 항목: 예제 21(B-36페이지)

HTML 페이지에서 이 액션의 활성화 여부를 나타냅니다.

Name: 문자열, 읽기

이 어댑터 액션의 변수 이름을 제공합니다.

Visible: 부울, 읽기

HTML 페이지에 이 어댑터 필드를 표시할 것인지 여부를 나타냅니다.

메소드

LinkToPage(PageSuccess, PageFail): AdapterAction, 읽기

참고 항목: 예제 10(B-24페이지), 예제 11(B-26페이지), 예제 21(B-36페이지), Page 객체, AdapterAction 타입(B-4페이지)

액션이 실행된 후에 표시할 페이지를 지정하려면 *LinkToPage*를 사용하십시오. 첫 번째 매개변수는 액션이 성공적으로 수행되면 표시될 페이지의 이름입니다. 두 번째 매개변수는 실행 중 오류가 발생하면 표시될 페이지의 이름입니다.

AdapterErrors 타입

참고 항목: Adapter 타입의 *Errors* 속성(B-2페이지)

AdapterErrors 타입은 어댑터의 *Errors* 속성의 속성을 정의합니다.

속성

Field: AdapterField, 읽기

참고 항목: AdapterField 타입(B-6페이지)

오류를 발생시킨 어댑터 필드를 표시합니다.

이 속성은 오류가 특정 어댑터 필드와 연결되지 않으면 *Null*이 됩니다.

ID: 정수, 읽기

오류의 숫자 식별자를 제공합니다.

이 속성은 ID가 정의되지 않으면 0이 됩니다.

Message: 텍스트, 읽기

참고 항목: 예제 7(B-22페이지)

오류에 대한 텍스트 설명을 제공합니다.

AdapterField 타입

참고 항목: Adapter 타입(B-2페이지), AdapterAction 타입(B-4페이지)

AdapterField 타입은 어댑터 필드의 속성과 메소드를 정의합니다.

속성

CanModify: 부울, 읽기

참고 항목: AdapterField 타입(아래 참조) 및 Adapter 타입(B-2페이지)의 *CanView* 속성

엔드 유저가 이 필드의 값을 수정할 권한이 있는지 여부를 나타냅니다.

CanView: 부울, 읽기

참고 항목: AdapterField 타입(위 참조) 및 Adapter 타입(B-2페이지)의 *CanModify* 속성

엔드 유저가 이 필드의 값을 볼 수 있는 권한이 있는지 여부를 나타냅니다.

DisplayLabel: 텍스트, 읽기

이 어댑터 필드에 대한 HTML 표시 레이블을 제공합니다.

DisplayStyle: 텍스트, 읽기

참고 항목: *InputStyle*(아래 참조), *ViewMode*(아래 참조), 예제 17(B-33페이지)

*DisplayStyle*은 필드 값의 읽기 전용 여부를 표시하는 방법을 제공합니다.

서버사이드 스크립트는 *DisplayStyle*을 사용하여 HTML 생성 방법을 결정할 수 있습니다. 어댑터 필드는 다음 표시 스타일 중 하나를 반환할 수 있습니다.

값	HTML 표시 스타일
''	정의되지 않았습니다.
'Text'	<p>를 사용하십시오.
'Image'	<img>를 사용하십시오. 필드의 Image 속성은 src 속성을 정의합니다.
'List'	<UL>을 사용하십시오. 각 <LI> 항목을 생성하기 위해 Values 속성을 열거하십시오.

ViewMode 속성은 *InputStyle* 또는 *DisplayStyle*을 사용하여 HTML을 생성할지 여부를 나타냅니다.

DisplayText: 텍스트, 읽기

참고 항목: *EditText*(아래 참조), 예제 9(B-23페이지)

읽기 전용으로 어댑터 필드의 값을 표시할 때 사용할 수 있는 필드를 제공합니다. DisplayText의 값은 숫자 형식을 포함할 수 있습니다.

DisplayWidth: 정수, 읽기

참고 항목: *MaxLength*(아래 참조)

어댑터 필드의 값에 대한 표시 너비를 문자 수로 제공합니다.

표시 너비가 정의되지 않으면 -1이 반환됩니다.

EditText: 텍스트, 읽기

참고 항목: *DisplayText*(위 참조), 예제 10(B-24페이지)

이 어댑터 필드에 대한 HTML 입력을 정의할 때 사용할 텍스트를 제공합니다. EditText의 값은 일반적으로 서식화되지 않습니다.

Image: AdapterImage 타입, 읽기

참고 항목: AdapterImage 타입(B-11페이지), 예제 12(B-27페이지)

이 어댑터 필드에 대한 이미지를 정의하는 객체를 제공합니다.

어댑터 필드가 이미지를 제공하지 않는 경우 Null이 반환됩니다.

InputStyle: 텍스트, 읽기

참고 항목: *DisplayStyle*(위 참조), *ViewMode*(아래 참조), 예제 17(B-33페이지)

이 필드에 대한 HTML 입력 스타일을 제공합니다.

서버사이드 스크립트는 InputStyle을 사용하여 HTML 생성 방법을 결정할 수 있습니다. 어댑터 필드는 다음 입력 스타일 중 하나를 반환할 수 있습니다.

값	의미
''	정의되지 않은 입력 스타일입니다.
'TextInput'	<input type="text">를 사용하십시오.
'PasswordInput'	<input type="password">를 사용하십시오.
'Select'	<select>를 사용하십시오. ValuesList 속성을 열거하여 각 <option> 요소를 생성하십시오.
'SelectMultiple'	<select multiple>을 사용하십시오. ValuesList 속성을 열거하여 각 <option> 요소를 생성하십시오.
'Radio'	ValuesList 속성을 열거하여 하나 이상의 <input type="radio">를 생성하십시오.
'CheckBox'	ValuesList 속성을 열거하여 하나 이상의 <input type="checkbox">를 생성하십시오.
'TextArea'	<textarea>를 사용하십시오.
File	<input type="file">를 사용하십시오.

ViewMode 속성은 *InputStyle* 또는 *DisplayStyle*을 사용하여 HTML을 생성할지 여부를 나타냅니다.

InputName: 텍스트, 읽기

참고 항목: 예제 10(B-24페이지)

어댑터 필드를 편집하는 데 필요한 HTML 입력 요소 이름을 제공합니다.

HTML <input>, <select> 또는 <textarea> 요소를 생성할 때 *InputName*을 사용하여 어댑터 컴포넌트가 HTTP 요청의 이름/값 쌍을 어댑터 필드와 연결시킬 수 있도록 하십시오.

MaxLength: 정수, 읽기

참고 항목: *DisplayWidth*(위 참조)

필드에 입력할 수 있는 최대 길이를 문자 수로 나타냅니다.

최대 길이가 정의되지 않은 경우 *MaxLength*가 -1이 됩니다.

Name: 텍스트, 읽기

어댑터 필드의 변수 이름을 반환합니다.

Required: 부울, 읽기

폼을 제출할 때 어댑터 필드의 값이 필수인지 여부를 나타냅니다.

Value: 가변, 읽기

참고 항목: *Values*(아래 참조), *DisplayText*(위 참조), *EditText*(위 참조)

계산에서 사용할 수 있는 값을 반환합니다. 예를 들어, 두 어댑터 필드 값을 더할 때는 *Value*를 사용하십시오.

Values: AdapterFieldValues, 읽기

참고 항목: *ValuesList*(위 참조), AdapterFieldValues 타입(B-9페이지), *Value*(위 참조), 예제 13(B-28페이지)

필드 값 리스트를 반환합니다. 이 어댑터 필드가 여러 값을 지원하지 않으면 *Values* 속성이 Null이 됩니다. 여러 값 필드가 사용됩니다. 예를 들어, 엔드 유저가 선택 리스트에서 여러 값을 선택할 수 있도록 허용합니다.

ValuesList: AdapterFieldValuesList, 읽기

참고 항목: *Values*(위 참조), AdapterFieldValuesList 타입(B-10페이지), 예제 13(B-28페이지)

이 어댑터 필드에 대한 선택 리스트를 제공합니다. HTML 선택 리스트, 체크 박스 그룹 또는 라디오 버튼 그룹을 생성하는 경우 *ValuesList*를 사용하십시오. *ValuesList* 내의 각 항목에는 값이 있으며 이름이 있을 수 있습니다.

Visible: 부울, 읽기

HTML 페이지에 이 어댑터 필드를 표시할 것인지 여부를 나타냅니다.

ViewMode: 텍스트, 읽기

참고 항목: *DisplayStyle*(위 참조), *InputStyle*(위 참조), 예제 17(B-33페이지)

HTML 페이지에서 이 어댑터 필드 값을 표시하는 방법을 제공합니다.

어댑터 필드는 다음 보기 모드 중 하나를 반환할 수 있습니다.

값	보기 모드
''	정의되지 않았습니다.
'Input'	<input>, <textarea> 또는 <select>를 사용하여 편집할 수 있는 HTML 폼 요소를 생성합니다.
'Display'	<p>, 또는 <img>를 사용하여 읽기 전용 HTML을 생성합니다.

ViewMode 속성은 *InputStyle* 또는 *DisplayStyle*을 사용하여 HTML을 생성할지 여부를 나타냅니다.

메소드

IsEqual(Value): 부울

참고 항목: 예제 16(B-32페이지)

이 함수를 호출하여 변수와 어댑터 필드의 값을 비교합니다.

AdapterFieldValues 타입

참고 항목: AdapterField 타입의 *Values* 속성(B-6페이지)

필드 값 리스트를 제공합니다. 여러 값 어댑터 필드가 이 속성을 지원합니다. 예를 들어, 다중 값 필드를 사용하여 엔드 유저가 선택 리스트에서 여러 값을 선택할 수 있게 합니다.

속성

Records: 열거자, 읽기

참고 항목: 예제 15(B-31페이지)

값 리스트에 레코드를 열거합니다

Value: 가변, 읽기

참고 항목: *ValueField*(아래 참조)

현재 열거 항목의 값을 반환합니다.

ValueField: AdapterField, 읽기

참고 항목: AdapterField 타입(B-6페이지), 예제 15(B-31페이지)

현재 열거 항목에 대한 어댑터 필드를 반환합니다. 예를 들어, 현재 열거 항목에 대한 *DisplayText*를 가져오려면 *ValueField*를 사용하십시오.

메소드

HasValue(Value): 부울

참고 항목: 예제 14(B-29페이지)

지정한 값이 필드 값 리스트에 있는지 여부를 나타냅니다. 이 메소드는 HTML 선택 리스트에서 항목을 선택하거나 체크 박스 그룹에서 항목을 선택할지 여부를 결정하는 데 사용됩니다.

AdapterFieldValuesList 타입

참고 항목: Adapter 타입(B-2페이지)

이 어댑터 필드에 대한 가능한 값 리스트를 제공합니다.

HTML 선택 리스트, 체크 박스 그룹 또는 라디오 버튼 그룹을 생성하는 경우 *ValuesList*를 사용하십시오. *ValuesList*의 각 항목에는 값이 포함되며 이름이 포함될 수도 있습니다.

속성

Image: AdapterImage, 읽기

현재 열거 항목의 이미지를 반환하며 항목에 이미지가 없으면 Null을 반환합니다.

Records: 열거자, 읽기

값 리스트에서 레코드를 열거합니다.

Value: 가변, 읽기

현재 열거 항목의 값을 반환합니다.

ValueField: AdapterField, 읽기

참고 항목: AdapterField 타입(B-6페이지), 예제 15(B-31페이지)

현재 열거 항목에 대한 어댑터 필드를 반환합니다. 예를 들어, 현재 열거 항목에 대한 *DisplayText*를 가져오려면 *ValueField*를 사용하십시오.

ValueName: 텍스트, 읽기

현재 항목의 텍스트 이름을 반환합니다. 값에 이름이 없으면 *ValueName*이 공백이 됩니다.

메소드

ImageOfValue(Value): AdapterImage

이 값과 연결된 이미지를 찾습니다. 이미지가 없으면 Null을 반환합니다.

NameOfValue(Value): 텍스트

이 값과 연결된 이름을 찾습니다. 값을 찾을 수 없거나 값에 이름이 없으면 빈 문자열을 반환합니다.

AdapterHiddenFields 타입

참고 항목: Adapter 타입의 *HiddenFields* 및 *HiddenRecordFields* 속성(B-2페이지)

어댑터가 HTML 폼에서 변경 사항을 제출하기 위해 필요로 하는 숨겨진 필드 이름 및 값에 대한 액세스를 제공합니다.

속성

Name: 텍스트, 읽기

열거되는 숨겨진 필드의 이름을 반환합니다.

Value: 텍스트, 읽기

열거되는 숨겨진 필드의 문자열 값을 반환합니다.

메소드

WriteFields(Response)

참고 항목: 예제 10(B-24페이지), 예제 22(B-37페이지)

<input type="hidden">을 사용하여 숨겨진 필드 이름과 값을 작성합니다.

HTML 폼에 숨겨진 HTML 필드를 모두 작성하려면 이 메소드를 호출하십시오.

AdapterImage 타입

참고 항목: AdapterField 타입(B-11페이지), AdapterAction 타입(B-4페이지)

액션이나 필드와 연결된 이미지를 표시합니다.

속성

AsHREF: 텍스트, 읽기

참고 항목: 예제 11(B-26페이지), 예제 12(B-27페이지)

HTML <img> 요소를 정의할 때 사용할 수 있는 URL을 제공합니다.

Module 타입

참고 항목: 모듈 객체(B-16페이지)

어댑터 컴포넌트는 이름을 사용하여 모듈의 한 속성으로 참조할 수 있습니다. 또한 모듈을 사용하여 모듈의 스크립트 가능한 객체(일반적으로 어댑터)를 열거할 수 있습니다.

속성

Name_: 텍스트, 읽기

참고 항목: 예제 20(B-35페이지)

모듈의 변수 이름을 나타냅니다. Modules 변수의 속성으로 모듈에 액세스하는 데 사용되는 이름입니다.

ClassName_: 텍스트, 읽기

참고 항목: 예제 20(B-35페이지)

모듈의 클래스 이름을 나타냅니다.

Objects: 열거자

참고 항목: 예제 20(B-35페이지)

모듈 내에서 스크립트 가능한 객체(일반적으로 어댑터)를 열거하려면 *Objects*를 사용하십시오.

Page 타입

참고 항목: Page 객체(B-16페이지), 예제 20(B-35페이지)

페이지의 속성과 메소드를 정의합니다.

속성

CanView: 부울, 읽기

엔드 유저가 이 페이지를 볼 수 있는 권한이 있는지 여부를 나타냅니다.

페이지는 액세스 권한을 등록합니다. *CanView*는 페이지에 의해 등록된 권한과 엔드 유저에게 부여된 권한을 비교합니다.

DefaultAction: AdapterAction 타입, 읽기

참고 항목: 예제 6(B-21페이지)

이 페이지와 연결된 디폴트 어댑터 액션을 나타냅니다.

일반적으로 디폴트 액션은 매개변수가 반드시 페이지에 전달되어야 하는 경우에 사용됩니다. *DefaultAction*은 Null이 될 수 있습니다.

HREF: 텍스트, 읽기

참고 항목: 예제 5(B-21페이지)

<a> 태그를 사용하여 이 페이지에 대한 하이퍼링크를 생성하는 데 사용할 수 있는 URL을 제공합니다.

LoginRequired: 부울, 읽기

엔드 유저가 이 페이지에 액세스하기 전에 반드시 로그인해야 하는지 여부를 나타냅니다.

페이지가 *LoginRequired* 플래그를 등록합니다. *True*이면 엔드 유저가 로그인하지 않고 이 페이지에 액세스할 수 없습니다.

Name: 텍스트, 읽기

참고 항목: 예제 5(B-21페이지)

등록된 페이지의 이름을 제공합니다.

페이지가 게시되면 페이지 이름이 HTTP 요청의 Path Info의 접미사일 때 *PageDispatcher* 페이지를 생성합니다.

Published: 부울, 읽기

참고 항목: 예제 5(B-21페이지)

엔드 유저가 URL에 대한 접미사를 페이지 이름으로 지정하여 이 페이지에 액세스할 수 있는지 여부를 나타냅니다.

페이지는 게시된 플래그를 등록합니다. 페이지 디스패처는 자동으로 게시된 페이지를 디스패치합니다. 일반적으로 *Published* 속성은 페이지에 대한 하이퍼링크를 사용하여 메뉴를 생성할 때 사용됩니다. *Published*를 *False*로 설정한 페이지는 메뉴에 나열되지 않습니다.

Title: 텍스트, 읽기

참고 항목: 예제 5(B-21페이지), 예제 18(B-35페이지)

페이지 제목을 제공합니다.

제목은 일반적으로 사용자에게 표시됩니다.

전역 객체

전역 객체는 서버사이드 스크립트로 참조될 수 있으며 개발자는 소스 코드의 객체 참조와 비슷한 전역 객체 참조를 스크립트에서 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
<%= Application.Title %>
```

위의 전역 객체는 웹 페이지에 애플리케이션의 제목을 표시합니다.

다음 표는 전역 스크립트 객체와 그에 대한 설명을 나열한 것입니다.

표 58.5 WebSnap 전역 객체

객체	설명
Application 객체(B-14페이지)	Title 필드와 같은 애플리케이션 어댑터의 필드 및 액션을 액세스합니다.
EndUser 객체(B-15페이지)	엔드 유저의 DisplayName, Login 액션, Logout 액션과 같은 엔드 유저 어댑터의 필드 및 액션을 액세스합니다.
Modules 객체(B-16페이지)	데이터 모듈이나 페이지 모듈을 이름으로 참조합니다. Modules 변수도 애플리케이션의 모듈을 열거하는 데 사용할 수 있습니다.
Page 객체(B-16페이지)	페이지 Title처럼 생성되는 페이지의 속성에 액세스합니다.
Pages 객체(B-16페이지)	등록된 페이지를 이름으로 참조합니다. 또한 Pages 변수를 사용하여 애플리케이션의 등록된 페이지를 열거할 수 있습니다.
Producer 객체(B-16페이지)	투명 태그를 포함할 수 있는 HTML 내용을 작성합니다.
Request 객체(B-17페이지)	HTTP 요청의 속성 및 메소드에 액세스합니다.
Response 객체(B-17페이지)	HTTP 응답에 대한 HTML 내용을 작성합니다.
Session 객체(B-18페이지)	엔드 유저 세션의 속성에 액세스합니다.

Application 객체

참고 항목: 어댑터 타입(B-2페이지)

Application 객체는 애플리케이션에 관한 정보에 대한 액세스를 제공합니다.

Title 필드와 같은 애플리케이션 어댑터의 필드와 액션에 액세스하려면 Application 객체를 사용하십시오. Application 객체는 Adapter이므로 추가 필드와 액션을 사용하여 정의할 수 있습니다. 애플리케이션 어댑터에 추가된 필드와 액션은 이름을 Application 객체의 속성으로 사용하여 액세스할 수 있습니다.

속성

Designing: 부울, 읽기

참고 항목: 예제 1(B-19페이지)

웹 애플리케이션이 IDE에서 디자인되고 있는지 여부를 나타냅니다.

웹 애플리케이션이 실행 중인 경우 디자인 모드에서 반드시 달라야 하는 HTML을 조건적으로 생성하려면 *Designing* 속성을 사용하십시오.

ModulePath: 텍스트, 읽기

참고 항목: *QualifyFileName* 메소드

웹 애플리케이션 실행 파일의 위치를 나타냅니다.

실행 파일과 동일한 디렉토리에 있는 파일 이름을 구성하려면 *ModulePath*를 사용하십시오.

ModuleFileName: 텍스트, 읽기

참고 항목: *QualifyFileName* 메소드

실행 파일의 전체 파일 이름을 나타냅니다.

Title: 텍스트, 읽기

참고 항목: 예제 18(B-35페이지)

애플리케이션의 제목을 제공합니다.

Title 속성에는 *TApplicationAdapter*의 *Title* 속성의 값이 있습니다. 일반적으로 이 값은 HTML 페이지의 맨 위에 표시됩니다.

메소드

QualifyFileName(FileName): 텍스트

참고 항목: 예제 1(B-19페이지)

상대 파일 이름이나 디렉토리 참조를 절대 참조로 변환합니다.

*QualifyFileName*은 웹 애플리케이션 실행 파일의 디렉토리 위치를 사용하여 전체가 다 한정되지 않은 파일 이름을 전체 파일 이름으로 변경합니다. 메소드가 전체 파일 이름을 반환합니다. FileName 매개변수가 전체 경로로 표시된 경우 파일 이름이 변경되지 않고 반환됩니다. 디자인 모드에서 FileName 매개변수는 프로젝트 파일의 디렉토리 위치로 한정됩니다.

EndUser 객체

참고 항목: Adapter 타입(B-2페이지)

현재 엔드 유저에 관한 정보에 대한 액세스를 제공합니다.

엔드 유저를 위한 *DisplayName* 등 엔드 유저 어댑터의 필드와 액션에 액세스하려면 *EndUser*를 사용하십시오. *EndUser* 이름을 객체의 속성으로 사용하여 엔드 유저 어댑터에 추가된 필드와 액션에 액세스할 수 있습니다.

속성

DisplayName: 텍스트, 읽기

참고 항목: 예제 19(B-35페이지)

엔드 유저의 이름을 제공합니다.

LoggedIn: 부울, 읽기

참고 항목: 예제 19(B-35페이지)

엔드 유저가 로그인했는지 여부를 나타냅니다.

LoginFormAction: AdapterAction 타입, 읽기

참고 항목: 예제 19(B-35페이지), AdapterAction 타입(B-4페이지)

사용자를 로그인시키는 데 필요한 어댑터 액션을 제공합니다.

LogoutAction: AdapterAction 타입, 읽기

참고 항목: 예제 19(B-35페이지), AdapterAction 타입(B-4페이지)

사용자를 로그아웃시키는 데 필요한 어댑터 액션을 제공합니다.

Modules 객체

참고 항목: 예제 2(B-20페이지), 예제 20(B-35페이지)

Modules 객체는 현재 HTTP 요청에 대한 서비스를 제공하기 위해 인스턴스화되었거나 활성화된 모든 모듈에 대한 액세스를 제공합니다.

특정 모듈을 참조하려면 모듈의 이름을 Modules 변수의 이름으로 사용하십시오. 애플리케이션 내의 모든 모듈을 열거하려면 Modules 객체를 사용하여 열거자(Enumerator)를 생성하십시오.

Page 객체

참고 항목: 예제 5(B-21페이지), Page 타입(B-12페이지)

Page 객체는 생성되고 있는 페이지의 속성에 대한 액세스를 제공합니다.

Page 객체의 속성 및 메소드에 관한 설명을 보려면 Page 타입을 참조하십시오.

Pages 객체

참고 항목: 예제 5(B-21페이지)

Pages 객체는 애플리케이션에 의해 등록된 모든 페이지에 대한 액세스를 제공합니다.

특정 페이지를 참조하려면 페이지의 이름을 Pages 변수의 속성으로 사용하십시오. 애플리케이션 내의 모든 페이지를 열거하려면 Pages 객체를 사용하여 열거자(Enumerator)를 생성하십시오.

Producer 객체

참고 항목: Response 객체(B-17페이지)

투명 태그를 포함한 텍스트를 작성하려면 Producer 객체를 사용하십시오. 태그는 페이지 프로듀서에 의해 번역되고 HTTP 응답에 기록됩니다. 텍스트에 투명 태그가 없는 경우 Response 객체를 사용하면 성능이 향상됩니다.

속성

Content: 텍스트, 읽기/쓰기

HTTP 응답의 내용 부분에 대한 액세스를 제공합니다.

HTTP 응답의 전체 내용 부분을 읽거나 쓰려면 *Content*를 사용하십시오. *Content*를 설정하면 투명 태그를 번역합니다. 투명 태그를 사용하지 않는 경우 Response.*Content*를 사용하면 성능이 향상됩니다.

메소드

Write(Value)

투명 태그를 지원하여 HTTP 요청의 내용 부분에 추가합니다.

HTTP 요청 내용의 내용 부분에 추가하려면 *Write* 메소드를 사용하십시오. Write 메소드는 다음과 같은 투명 태그를 번역합니다.

```
Write('Translate this: <#MyTag>')
```

투명 태그를 사용하지 않는 경우 Response.*Write*를 사용하면 성능이 향상됩니다.

Request 객체

HTTP 요청에 대한 액세스를 제공합니다.

HTTP 요청에 대한 정보에 액세스하려면 Response 객체의 속성을 사용하십시오.

속성

Host: 텍스트, 읽기

HTTP 요청의 Host 헤더의 값을 보고합니다.

Host는 *TWebRequest*의 *Host* 속성과 동일합니다.

PathInfo: 텍스트, 읽기

URL의 PathInfo 부분을 포함합니다.

PathInfo는 *TWebRequest*의 *InternalPathInfo* 속성과 동일합니다.

ScriptName: 텍스트, 읽기

웹 서버 애플리케이션의 이름을 지정하는 URL의 스크립트 이름 부분을 포함합니다.

ScriptName은 *TWebRequest*의 *InternalScriptName* 속성과 동일합니다.

Response 객체

참고 항목: Producer 객체(B-16페이지)

HTTP 응답에 대한 액세스를 제공합니다. HTTP 응답의 내용 부분에 기록하려면 Response 객체를 사용하십시오. 투명 태그를 사용할 경우에는 Response 객체 대신 Producer 객체를 사용하십시오.

속성

Content: 텍스트, 읽기/쓰기

HTTP 응답의 내용 부분에 대한 액세스를 제공합니다.

HTTP 응답의 전체 내용 부분을 읽거나 쓰려면 *Content*를 사용하십시오.

메소드

Write(Value)

참고 항목: 예제 5(B-21페이지)

HTTP 응답의 내용 부분에 Value를 추가합니다.

HTTP 응답의 내용에 Value를 추가하려면 *Write* 메소드를 사용하십시오. Write 메소드가 투명 태그를 번역하지는 않습니다.

하나 이상의 투명 태그를 포함한 문자열을 생성하려면 Producer 객체의 *Write* 메소드를 사용하십시오.

Session 객체

Session 객체는 세션 ID 및 값에 대한 액세스를 제공합니다.

세션은 짧은 시간 동안 엔드 유저에 대한 정보를 추적하는 데 사용됩니다.

속성

SessionID.Value: 텍스트, 읽기/쓰기

현재 엔드 유저 세션의 ID에 대한 액세스를 제공합니다.

property(Name): 가변, 읽기

현재 엔드 유저의 세션에 저장된 값에 대한 액세스를 제공합니다.

JScript 예제

다음은 다양한 서버사이드 스크립트 속성과 메소드의 사용 방법을 보여 주는 JScript 예제입니다.

표 58.6 서버사이드 스크립트의 JScript 예제

예제	설명
예제 1(B-19페이지)	Application 객체의 *QualifyFilename* 메소드를 사용하여 이미지에 대한 상대 경로 참조를 생성합니다.
예제 2(B-20페이지)	모듈을 참조하는 변수를 선언합니다.
예제 3(B-20페이지)	웹 애플리케이션 내의 모듈을 열거하고 HTML 테이블에 모듈 이름을 표시합니다.
예제 4(B-20페이지)	등록된 페이지를 참조하는 변수를 선언합니다.
예제 5(B-21페이지)	등록된 페이지를 열거하여 게시된 페이지에 대한 하이퍼링크 메뉴를 생성합니다.
예제 6(B-21페이지)	등록된 페이지를 열거하여 게시된 페이지의 디폴트 액션에 대한 하이퍼링크 메뉴를 생성합니다.
예제 7(B-22페이지)	어댑터가 검색한 오류 리스트를 작성합니다.
예제 8(B-22페이지)	어댑터의 액션 객체를 모두 열거하여 HTML 테이블에 액션 객체 속성 값을 표시합니다.

표 58.6 서버사이드 스크립트의 JScript 예제(계속)

예제	설명
예제 9(B-23페이지)	어댑터의 레코드를 열거하여 HTML 테이블에 어댑터 필드 값을 표시합니다.
예제 10(B-24페이지)	HTML 폼을 생성하여 어댑터 필드를 편집하고 어댑터 액션을 제출합니다.
예제 11(B-26페이지)	어댑터 액션을 표시하여 페이징을 지원합니다.
예제 12(B-27페이지)	<img> 태그를 사용하여 어댑터 필드의 이미지를 표시합니다.
예제 13(B-28페이지)	<select> 및 <option> 태그를 사용하여 어댑터 필드를 표시합니다.
예제 14(B-29페이지)	어댑터 필드를 체크 박스 그룹으로 표시합니다.
예제 15(B-31페이지)	및 태그를 사용하여 어댑터 필드의 값을 표시합니다.
예제 16(B-32페이지)	어댑터 필드를 라디오 버튼 그룹으로 표시합니다.
예제 17(B-33페이지)	어댑터 필드의 DisplayStyle, InputStyle 및 ViewMode 속성을 사용하여 HTML을 생성합니다.
예제 18(B-35페이지)	Application 객체 및 Page 객체의 속성을 사용하여 페이지 헤더를 생성합니다.
예제 19(B-35페이지)	EndUser 객체의 속성을 사용하여 엔드 유저의 이름, 로그인 명령 및 로그아웃 명령을 표시합니다.
예제 20(B-35페이지)	모듈 내에 스크립트 가능한 객체를 나열합니다.
예제 21(B-36페이지)	어댑터 액션의 DisplayStyle 속성을 사용하여 HTML을 생성합니다.
예제 22(B-37페이지)	HTML 테이블을 생성하여 여러 디테일 레코드를 업데이트합니다.

예제 1

참고 항목: Application 객체의 *Designing* 및 *QualifyFileName* 속성(B-14페이지), Request 객체의 *PathInfo* 속성(B-17페이지)

다음은 이미지에 대한 상대 경로 참조를 생성하는 예제입니다. 스크립트가 디자인 모드에 있으면 실제 디렉토리를 참조하며 그렇지 않은 경우에는 가상 디렉토리를 참조합니다.

```
<%
     function PathInfoToRelativePath(S)
     {
       var R = '';
       var L = S.length
       I = 0
       while (I < L)
       {
         if (S.charAt(I) == '/')
           R = R + '../'
         I++
       }
       return R
     }

     function QualifyImage(S)
```

```
    {
      if (Application.Designing)
        return Application.QualifyFileName("..\\images\\" + S);   //
relative directory
      else
        return PathInfoToRelativePath(Request.PathInfo) + '../images/' +
S;  // virtual directory
    }
  %>
```

예제 2

참고 항목: Modules 객체(B-16페이지)

다음은 WebModule1을 참조하는 변수를 선언하는 예제입니다.

```
<% var M  = Modules.WebModule1 %>
```

예제 3

참고 항목: Modules 객체(B-16페이지)

다음은 인스턴스화된 모듈을 열거하고 그 변수 이름 및 클래스 이름을 테이블에 표시하는 예제입니다.

```
<table border=1>
<tr><th>Name</th><th>ClassName</th></tr>
<%
   var e = new Enumerator(Modules)
   for (; !e.atEnd(); e.moveNext())
   {
%>
  <tr><td><%=e.item().Name_%></td><td><%=e.item()ClassName._%></td></tr>
<%
   }
%>
</table>
```

예제 4

참고 항목: Pages 객체(B-16페이지), Page 객체의 *Title* 속성(B-16페이지)

다음은 Home이라는 페이지를 참조하는 변수를 선언하는 예제입니다. 또한 Home의 제목도 표시합니다.

```
<% var P = Pages.Home %>
<p><%= P.Title %></p>
```

예제 5

참고 항목: Pages 객체(B-16페이지), Page 객체의 *Published* 및 *HREF* 속성(B-16페이지), Response 객체의 *Write* 메소드(B-17페이지)

다음은 등록된 페이지를 열거하고 게시된 모든 페이지에 대한 하이퍼링크를 표시하는 메뉴를 생성하는 예제입니다.

```
<table>
<td>
<%   e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           s += '<ahref="' + e.item().HREF + '">' + e.item().Title + '</
a>'
         else
           s += e.item().Title
         c++
       }
     }
     if (c>1)Response.Write(s)
%>
</td>
</table>
```

예제 6

참고 항목: Page 타입의 *DefaultAction* 속성(B-12페이지)

다음은 등록된 페이지를 열거하고 디폴트 액션에 대한 하이퍼링크를 표시하는 메뉴를 생성하는 예제입니다.

```
<table>
<td>
<%  e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           if (e.item().DefaultAction != null)
             s += '<ahref="' + e.item().DefaultAction.AsHREF+'">
+e.item().Title+'</a>'
```

```
            else
              s += '<ahref="' + e.item().HREF + '">' + e.item().Title +
'</a>'
          else
            s += e.item().Title
          c++
        }
      }
      if (c>1)Response.Write(s)
%>
</td>
</table>
```

예제 7

참고 항목: Adapter 타입의 *Errors* 속성(B-2페이지), AdapterErrors 타입(B-6페이지), Modules 객체(B-16페이지), Response 객체의 *Write* 메소드(B-17페이지)

다음은 어댑터가 검색한 오류 리스트를 작성하는 예제입니다.

```
<% {
   var e = new Enumerator(Modules.CountryTable.Adapter.Errors)
   for (; !e.atEnd(); e.moveNext())
   {
     Response.Write("<li>" + e.item().Message)
   }
   e.moveFirst()
   } %>
```

예제 8

참고 항목: Adapter 타입의 *Actions* 속성(B-2페이지), AdapterAction 타입(B-4페이지)

다음은 어댑터의 모든 액션을 열거하고 테이블에 액션 속성 값을 표시하는 예제입니다.

```
<% // Display some properties of an action in a table.
   function DumpAction(A)
   {
%>
      <table border="1">
         <tr><th COLSPAN=2><%=A.Name%></th>
         <tr><th>AsFieldValue:</th><td><%=A.AsFieldValue %></td>
         <tr><th>AsHREF:</th><td><%=A.AsHREF %></span>
         <tr><th>DisplayLabel:</th><td><%=A.DisplayLabel %></td>
         <tr><th>Enabled:</th><td><%=A.Enabled %></td>
         <tr><th>CanExecute:</th><td><span class="value"><%=A.CanExecute
%></td>
     </table>
<%
  }
%>
```

```
<% // Call the DumpAction function for every action in an adapter.
   function DumpActions(A)
   {
     var e =new Enumerator(A)
     for (; !e.atEnd(); e.moveNext())
     {
       DumpAction(e.item())
     }
   }
%>


<%
// Display properties of actions in the adapter named Adapter1.
DumpActions(Adapter1.Actions) %>
```

예제 9

참고 항목: Adapter 타입의 *Records* 속성(B-2페이지), AdapterField 타입의 *DisplayText* 속성(B-6페이지)

다음은 어댑터의 레코드를 열거하여 HTML 테이블을 생성하는 예제입니다.

```
<%
// Define variables for the adapter and fields.

vAdapter=Modules.CountryTable.Adapter
vAdapter_Name=vAdapter.Name
vAdapter_Capital=vAdapter.Capital
vAdapter_Continent=vAdapter.Continent
%>

<%
// Function to write column text so that all cells have borders.
function WriteColText(t)
{
   Response.Write((t!="")?t:" ")
}
%>

<table border="1">
  <tr>
  <th>Name</th>
  <th>Capital</th>
  <th>Continent</th>
<%
   // Enumerate all the records in the adapter and write the field
values.

   var e = new Enumerator(vAdapter.Records)
   for (; !e.atEnd(); e.moveNext())
   { %>
```

```
      <tr>
      <td><div><% WriteColText(vAdapter_Name.DisplayText) %></div></td>
      <td><div><% WriteColText(vAdapter_Capital.DisplayText) %></div></
td>
      <td><div><% WriteColText(vAdapter_Continent.DisplayText) %></div></
td>
      </tr>
<%
   }
%>
</table>
```

예제 10

참고 항목: AdapterAction 타입의 *LinkToPage* 및 *AsFieldValue* 속성(B-4페이지), AdapterField 타입의 *InputName* 및 *DisplayText* 속성(B-6페이지), Adapter 타입의 *HiddenFields* 및 *HiddenRecordFields* 속성(B-2페이지)

다음은 HTML 폼을 생성하여 어댑터 필드를 편집하고 어댑터 액션을 제출하는 예제입니다.

```
<%
// Define some variables for the adapter, fields, and actions.

vAdapter=Modules.CountryTable.Adapter
vAdapter_Name=vAdapter.Name
vAdapter_Capital=vAdapter.Capital
vAdapter_Continent=vAdapter.Continent
vAdapter_Apply=vAdapter.Apply
vAdapter_RefreshRow=vAdapter.RefreshRow

// Put the adapter in Edit mode unless the mode is already set.  If the
mode is already
// set this is probably because an adapter action set the mode. For
example, an insert
// row action would put the adapter in Insert mode.

if (vAdapter.Mode=="")
  vAdapter.Mode="Edit"
%>
<form name="AdapterForm1" method="post">

  <!-- This hidden field defines the action that is executed when the
form is submitted. -->

  <input type="hidden" name="__act">


<%
   // Write hidden fields defined by the adapter.

   if (vAdapter.HiddenFields != null)
   {
```

```
     vAdapter.HiddenFields.WriteFields(Response)
   } %>
<% if (vAdapter.HiddenRecordFields != null)
   {
     vAdapter.HiddenRecordFields.WriteFields(Response)
   } %>
  <table>
    <tr>
      <td>
        <table>
          <tr>
          <!-- Write input fields to edit the fields of the adapter -->

          <td>Name</td>
          <td ><input type="text" size="24"
name="<%=vAdapter_Name.InputName%>" value="
             <%= vAdapter_Name.EditText %>" ></td>
          </tr>
          <tr>
          <td>Capital</td>
          <td ><input type="text" size="24"
name="<%=vAdapter_Capital.InputName%>"
             value="<%= vAdapter_Capital.EditText %>" ></td>
          </tr>
          <tr>
          <td>Continent</td>
          <td ><input type="text" size="24"
name="<%=vAdapter_Continent.InputName%>"
             value="<%= vAdapter_Continent.EditText %>" ></td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td>
        <table>
          <!-- Write submit buttons to execute actions.  Use LinkToPage
so this
               page is regenerated after executing an action. -->

          <tr>
          <td><input type="submit" value="Apply"
            onclick = "AdapterForm1.__act.value='
              <%=vAdapter_Apply.LinkToPage(Page.Name).AsFieldValue%>'">
</td>
          <td><input type="submit" value="Refresh"
            onclick = "AdapterForm1.__act.value='
```

```
<%=vAdapter_RefreshRow.LinkToPage(Page.Name).AsFieldValue%>'"> </td>
          </tr>
        </table>
      </td>
    </tr>
  </table>
</form>
```

예제 11

참고 항목: AdapterAction 타입의 *Array* 및 *AsHREF* 속성(B-4페이지)

다음은 어댑터 액션을 표시하여 페이징을 지원하는 예제입니다. PrevPage, GotoPage 및 NextPage 액션이 하이퍼링크로 표시됩니다. GotoPage 액션에는 명령 배열이 있습니다. 명령을 열거하여 각 페이지로 가는 하이퍼링크를 생성합니다.

```
<%
   // Define variables for the adapter and actions.

   vAdapter = Modules.WebDataModule1.QueryAdapter
   vPrevPage = vAdapter.PrevPage
   vGotoPage = vAdapter.GotoPage
   vNextPage = vAdapter.NextPage
%>

<!-- Generate a table that displays hyperlinks between pages. -->

<table cellpadding="5">
<tr>
<td>
<%
   // Prevpage displays "<<".  Use an anchor tag only if the command is
enabled.

   if (vPrevPage.Enabled)
   { %>
     <ahref="<%=vPrevPage.LinkToPage(Page.Name).AsHREF%>"><<<</a>
<%
   }
   else
   {%>
     <a><<<</a>
 <%} %>
<%
   // GotoPage has a list of commands. Loop through the list.
   // Use an anchor tag only if the command is enabled.

   if (vGotoPage.Array != null)
   {
     var e = new Enumerator(vGotoPage.Array)
     for (; !e.atEnd(); e.moveNext())
```

```
     {
%>
       <td>
<%     if (vGotoPage.Enabled)
       { %>
          <a href="<%=vGotoPage.LinkToPage(Page.Name).AsHREF%>">
            <%=vGotoPage.DisplayLabel%></a>
<%     }
       else
       { %>
          <a><%=vGotoPage.DisplayLabel%></a>
<%     }
%>
       </td>
<%
     }
   }
%>
<td>
<%
   // NextPage displays ">>".  Use an anchor tag only if the command is
enabled.

   if (vNextPage.Enabled)
   { %>
     <ahref="<%=vNextPage.LinkToPage(Page.Name).AsHREF%>">>></a>
<%
   }
   else
   {%>
     <a>>></a>
 <%} %>
</td>
</table>
```

예제 12

참고 항목: AdapterField 타입의 *Image* 속성(B-6페이지)

다음은 어댑터 필드의 이미지를 표시하는 예제입니다.

```
<%
// Declare variables for the adapter and field.

vAdapter=Modules.WebDataModule3.DataSetAdapter1
vGraphic=vAdapter.Graphic
%>

<!-- Display the adapter field as an image. -->
<img src="<%=(vGraphic.Image!=null) ? vGraphic.Image.AsHREF : ''%>"
alt="<%=vGraphic.DisplayText%>">
```

예제 13

참고 항목: AdapterField 타입의 *Values* 및 *ValuesList* 속성(B-6페이지)

다음은 HTML <select> 및 <option> 요소가 있는 어댑터 필드를 작성하는 예제입니다.

```
<%
// Return an object that defines HTML select options for an adapter
field.
// The returned object has the following elements:
//
// text - string containing the <option> elements.
// count - the number of <option> elements.
// multiple - string containing the either 'multiple' or ''.
//            Use this value as an attribute of the <select> element.
//
// Use as follows:
//   obj=SelOptions(f)
//   Response.Write('<select size="' + obj.count + '" name="' +
f.InputName + '" ' +
//     obj.multiple + '>' + obj.text + '</select>')

function SelOptions(f)
{
   var s = ''
   var v=''
   var n=''
   var c=0
   if (f.ValuesList != null)
   {
     var e =new Enumerator(f.ValuesList.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       s+= '<option'
       v = f.ValuesList.Value;
       var selected
       if (f.Values == null)
          selected = f.IsEqual(v)
       else
          selected = f.Values.HasValue(v)
       if (selected)
          s += ' selected'
       n = f.ValuesList.ValueName;
       if (n=='')
       {
         n = v
         v=''
       }
       if (v!='') s += ' value="' + v + '"'
       s += '>' + n + '</option>\r\n'
       c++
     }
```

```
      e.moveFirst()
    }
    r = new Object;
    r.text = s
    r.count = c
    r.multiple = (f.Values == null) ? '' : 'multiple'
    return r;
  }
%>

<%
    // Generate HTML select options for an adapter field
    function WriteSelectOptions(f)
    {
       obj=SelOptions(f)
%>
       <select size="<%=obj.count%>" name="<%=f.InputName%>"
<%=obj.multiple%> >
         <%=obj.text%>
       </select>
<%
    }
%>
```

예제 14

참고 항목: AdapterField 타입의 *Values* 및 *ValuesList* 속성(B-6페이지)

다음은 어댑터 필드를 <input type="checkbox"> 요소 그룹으로 작성하는 예제입니다.

```
<%
// Return an object that defines HTML check boxes for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <input type=checkbox> elements.
// count - the number of <option> elements.
//
// Use as follows to define a check box group with three columns and no
additional
// attributes:
//   obj=CheckBoxGroup(f, 3, '')
//   Response.Write(obj.text)
//
function CheckBoxGroup(f,cols,attr)
{
   var s = ''
   var v=''
   var n=''
   var c=0;
   var nm= f.InputName
   if (f.ValuesList == null)
   {
```

```
      s+= '<input type="checkbox"'
      if (f.IsEqual(true)) s += ' checked'
      s += ' value="true"' + ' name="' + nm + '"'
      if (attr!='') s+= ' ' + attr
      s += '></input>\r\n'
      c = 1
    }
    else
    {
      s+= '<table><tr>'
      var e =new Enumerator(f.ValuesList.Records)
      for (; !e.atEnd(); e.moveNext())
      {
        if (c % cols == 0 && c != 0) s += '</tr><tr>'
        s+= '<td><input type="checkbox"'
        v = f.ValuesList.Value;
        var checked
        if (f.Values == null)
          checked = (f.IsEqual(v))
        else
          checked = f.Values.HasValue(v)
        if (checked)
          s += ' checked'
        n = f.ValuesList.ValueName;
        if (n=='')
          n = v
        s += 'value="' + v + '"' + ' name="' + nm + '"'
        if (attr!='') s+= ' ' + attr
        s += '>' + n + '</input></td>\r\n'
        c++
      }
      e.moveFirst()
      s += '</tr></table>'
    }
    r = new Object;
    r.text = s
    r.count = c
    return r;
  }
%>

<%
    // Write an adapter field as a check box group
    function WriteCheckBoxGroup(f, cols, attr)
    {
       obj=CheckBoxGroup(f, cols, attr)
       Response.Write(obj.text);
    }
%>
```

예제 15

참고 항목: AdapterField 타입의 *Values* 및 *ValuesList* 속성(B-6페이지), AdapterFieldValues 타입(B-9페이지)

다음은 <ul> 및 <li> 요소를 사용하여 어댑터 필드를 읽기 전용 값 리스트로 작성하는 예제입니다.

```
<%
// Return an object that defines HTML list values for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <li> elements.
// count - the number of elements.
//
// text will be blank and count will be zero if the adapter field does
not
// support multiple values.
//
// Use as follows to define adisplays a read only list of this an adapter
// fields values.
//   obj=ListValues(f)
//   Response.Write('<ul>' + obj.text + '</ul>')
//
function ListValues(f)
{
   var s = ''
   var v=''
   var n=''
   var c=0;
   r = new Object;
   if (f.Values != null)
   {
     var e =new Enumerator(f.Values.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       s+= '<li>'
       s += f.Values.ValueField.DisplayText;
       s += '</li>'
       c++
     }
     e.moveFirst()
   }
   r.text = s
   r.count = c
   return r;
}
%>

<%
// Write an adapter field as a list of read-only values.
function WriteListValues(f)
```

```
{
   obj=ListValues(f)
%>
   <ul><%=obj.text%></ul>
<%
}
%>
```

예제 16

참고 항목: AdapterFieldValuesList 타입(B-10페이지), AdapterField 타입의 *Values* 및 *ValuesList* 속성(B-6페이지)

다음은 어댑터 필드를 <input type="radio"> 요소의 그룹으로 작성하는 예제입니다.

```
<%
// Return an object that defines HTML radio buttons for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <input type=radio> elements.
// count - the number of elements.
//
// Use as follows to define a radio button group with three columns and
no additional
// attributes:
//   obj=RadioGroup(f, 3, '')
//   Response.Write(obj.text)
//

function RadioGroup(f,cols,attr)
{
   var s = ''
   var v=''
   var n=''
   var c=0;
   var nm= f.InputName
   if (f.ValuesList == null)
   {
     s+= '<input type="radio"'
     if (f.IsEqual(true)) s += ' checked'
     s += ' value="true"' + ' name="' + nm + '"'
     if (attr!='') s+= ' ' + attr
     s += '></input>\r\n'
     c = 1
   }
   else
   {
     s+= '<table><tr>'
     var e =new Enumerator(f.ValuesList.Records)
     for (; !e.atEnd(); e.moveNext())
```

```
      {
        if (c % cols == 0 && c != 0) s += '</tr><tr>'
        s+= '<td><input type="radio"'
        v = f.ValuesList.Value;
        var checked
        if (f.Values == null)
          checked = (f.IsEqual(v))
        else
          checked = f.Values.HasValue(v)
        if (checked)
          s += ' checked'
        n = f.ValuesList.ValueName;
        if (n=='')
        {
        n = v
        }
        s += 'value="' + v + '"' + ' name="' + nm + '"'
        if (attr!='') s+= ' ' + attr
        s += '>' + n + '</input></td>\r\n'
        c++
      }
      e.moveFirst()
      s += '</tr></table>'
    }
    r = new Object;
    r.text = s
    r.count = c
    return r;
  }
%>

<%
   // Write an adapter field as a radio button group
   function WriteRadioGroup(f, cols, attr)
   {
      obj=RadioGroup(f, cols, attr)
      Response.Write(obj.text);
   }
%>
```

예제 17

참고 항목: AdapterField 타입의 *DisplayStyle, ViewMode* 및 *InputStyle* 속성(B-6페이지)

다음은 *InputStyle*, *DisplayStyle* 및 *ViewMode* 속성 값을 기준으로 어댑터 필드의 HTML을 생성하는 예제입니다.

```
<%
   function WriteField(f)
   {
     Mode = f.ViewMode
     if (Mode == 'Input')
```

```
      {
        Style = f.InputStyle
        if (Style == 'SelectMultiple' || Style == 'Select')
           WriteSelectOptions(f)
        else if (Style == 'CheckBox')
          WriteCheckBoxGroup(f, 2, '')
        else if (Style == 'Radio')
          WriteRadioGroup(f, 2, '')
        else if (Style == 'TextArea')
        {
%>
          <textarea wrap=OFF name="<%=f.InputName%>"><%= f.EditText %></
textarea>
<%
        }
        else if (Style == 'PasswordInput')
        {
%>
          <input type="password" name="<%=f.InputName%>"/>
<%
        }
        else if (Style == 'File')
        {
%>
           <input type="file" name="<%=f.InputName%>"/>
<%
        }
        else
        {
%>
          <input type="input" name="<%=f.InputName%>" value="<%=f.EditText
%>"/>
<%
        }
      }
      else
      {
        Style = f.DisplayStyle
        if (Style == 'List')
          WriteListValues(f)
        else if (Style == 'Image')
        {
%>
          <img src="<%=(f.Image!=null) ? f.Image.AsHREF : ''%>"
alt="<%=f.DisplayText%>">
<%      }
        else
          Response.Write('<p>' + f.DisplayText + '</p>')
      }
    }
%>
```

예제 18

참고 항목: Page 객체(B-16페이지), Application 객체의 *Title* 속성(B-14페이지)

다음은 Application 객체 및 Page 객체의 속성을 사용하여 페이지 헤더를 생성하는 예제입니다.

```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1><%= Application.Title %></h1>

<h2><%= Page.Title %></h2>
```

예제 19

참고 항목: EndUser 객체(B-15페이지)

다음은 EndUser 객체의 속성을 사용하여 엔드 유저의 이름, 로그인 명령 및 로그아웃 명령을 표시하는 예제입니다.

```
<% if (EndUser.Logout != null)
   {
     if (EndUser.DisplayName != '')
     {
%>
       <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   }
     if (EndUser.Logout.Enabled) {
%>
       <ahref="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   }
     if (EndUser.LoginForm.Enabled) {
%>
       <ahref="<%=EndUser.LoginForm.AsHREF%>">Login</a>
<%   }
   }
%>
```

예제 20

참고 항목: Module 타입(B-12페이지)

다음은 모듈 내의 스크립트 가능한 객체를 나열하는 예제입니다.

```
<%
   // Write an HTML table list the name and classname of all scriptable
objects in a module.
   function ListModuleObjects(m)
```

```
   {
%>
      <p></p>
      <table border="1">
      <tr>
      <th colspan="2"><%=m.Name_ + ': ' + m.ClassName_%></th>
      </tr>
<%
      var e = new Enumerator(m.Objects)
      for (; !e.atEnd(); e.moveNext())
      {
%>
      <tr>
      <td>
         <%=e.item().Name_ + ': ' + e.item().ClassName_ %>
      </td>
      </tr>
<%
      }
%>
      </table>
<%
   }
%>
```

예제 21

참고 항목: AdapterAction 타입의 *DisplayStyle* 및 *Enabled* 속성(B-4페이지)

다음은 *DisplayStyle* 속성을 기초로 어댑터 액션에 대한 HTML을 생성하는 예제입니다.

```
<%
   // Write HTML for an adapter action using the DisplayStyle property.
   //
   // a - action
   // cap - caption.  If blank the action's display label is used.
   // fm - name of the HTML form
   // p - page name to goto after action execution.  If blank, the
current page is used.
   //
   // Note that this function does not use the action's Array property.
Is is assumed that
   // the action has a single command.
   //
   function WriteAction(a, cap, fm, p)
   {
       if (cap == '')
         cap = a.DisplayLabel
       if (p == '')
         p = Page.Name
       Style = a.DisplayStyle
       if (Style == 'Anchor')
```

```
       {

          if (a.Enabled)
          {
            // Do not use the href property.  Instead, submit the form so
that  HTML form
            // fields are part of the HTTP request.
%>
            <a href=""onclick="<%=fm%>.__act.value='
               <%=a.LinkToPage(p).AsFieldValue%>';<%=fm%>.submit();return
false;">
               <%=cap%></a>
<%
          }
          else
          {
%>
            <a><%=cap%></a>
<%
          }
       }
       else
       {
%>
          <input type="submit" value="<%= cap%>"
onclick="<%=fm%>.__act.value='<%=a.LinkToPage(p).AsFieldValue%>'">
<%
       }
   }
%>
```

예제 22

참고 항목: Adapter type의 *HiddenFields*, *HiddenRecordFields* 및 *Mode* 속성(B-2페이지)

다음은 HTML 테이블을 생성하여 여러 디테일 레코드를 업데이트하는 예제입니다.

```
<%
vItemsAdapter=Modules.DM.ItemsAdapter
vOrdersAdapter=Modules.DM.OrdersAdapter
vOrderNo=vOrdersAdapter.OrderNo
vCustNo=vOrdersAdapter.CustNo
vPrevRow=vOrdersAdapter.PrevRow
vNextRow=vOrdersAdapter.NextRow
vRefreshRow=vOrdersAdapter.RefreshRow
vApply=vOrdersAdapter.Apply
vItemNo=vItemsAdapter.ItemNo
vPartNo=vItemsAdapter.PartNo
vDiscount=vItemsAdapter.Discount
vQty=vItemsAdapter.Qty
%>
```

```
<!-- Use two adapters to update multiple detail records.
     The orders adapter is associated with the master dataset.
     The items adapter is associated with the detail dataset.
     Each row in a grid displays values from the items adapter.  One
     column display an <input> element for editing Qty.  The apply button
     updates the Qty value in each detail record.-->

<!-- Display the order number and customer number values. -->
<h2>OrderNo: <%= vOrderNo.DisplayText %></h2>
<h2>CustNo: <% vCustNo.DisplayText %></h2>

<%
   // Put the items adapter in edit mode because this form updates
   // the Qty field.
   vItemsAdapter.Mode = 'Edit'
%>

<form name="AdapterForm1" method="post">

  <!-- Define a hidden field for submitted the action name and parameters
-->
  <input type="hidden" name="__act">

<%
   // Write hidden fields containing state information about the
   // orders adapter and items adapter.
   if (vOrdersAdapter.HiddenFields != null)
     vOrdersAdapter.HiddenFields.WriteFields(Response)
   if (vItemsAdapter.HiddenFields != null)
     vItemsAdapter.HiddenFields.WriteFields(Response)

   // Write hidden fields containing state information about the current
   // record of the orders adapter.
   if (vOrdersAdapter.HiddenRecordFields != null)
     vOrdersAdapter.HiddenRecordFields.WriteFields(Response)%>

  <table border="1">
    <tr>
    <th>ItemNo</th>
    <th>PartNo</th>
    <th>Discount</th>
    <th>Qty</th>
  </tr>
<%
   var e = new Enumerator(vItemsAdapter.Records)
   for (; !e.atEnd(); e.moveNext())
   { %>
     <tr>
     <td><%=vItemNo.DisplayText%></td>
     <td><%=vPartNo.DisplayText%></td>
     <td><%=vDiscount.DisplayText%></td>
```

```
     <td><input type="text" name="<%=vQty.InputName%>"
value="<%= vQty.EditText %>" ></td>
     </tr>
<%
     // Write hidden fields containing state information about each
record of the
     // items adapter. This is needed by the items adapter when updating
the Qty field.

     if (vItemsAdapter.HiddenRecordFields != null)
       vItemsAdapter.HiddenRecordFields.WriteFields(Response)
   }
%>
  </table>
  <p></p>
  <table>
    <td><input type="submit" value="Prev Order"

onclick="AdapterForm1.__act.value='<%=vPrevRow.LinkToPage(Page.Name)
.AsFieldValue%>'"></td>
    <td><input type="submit" value="Next Order"

onclick="AdapterForm1.__act.value='<%=vNextRow.LinkToPage(Page.Name)
.AsFieldValue%>'"></td>
    <td><input type="submit" value="Refresh"

onclick="AdapterForm1.__act.value='<%=vRefreshRow.LinkToPage(Page.Name)
.AsFieldValue%>'"></td>
    <td><input type="submit" value="Apply"
      onclick="AdapterForm1.__act.value='<%=vApply.LinkToPage(Page.Name)
.AsFieldValue%>'"></td>
  </table>
</form>
```

색인

Symbols

&(앰퍼샌드) 문자 8-33
... (생략 부호) 버튼 19-21

Numerics

2단계 커밋 29-17
2티어 애플리케이션 18-3, 18-9, 18-12
3D 9-17
3D 패널 9-17
3티어 애플리케이션 멀티 티어 애플리케이션 *참조*
80x87 보조 프로세서 A-3

A

-a 컴파일러 옵션 A-5
Abort 프로시저
  편집 방지 22-19
abort 함수 A-9
AbortOnKeyViol 속성 24-51
AbortOnProblem 속성 24-50
About 박스 57-2
  ActiveX 컨트롤에 추가 43-5
  Execute 메소드 57-5
  속성 추가 57-4
Access 테이블
  로컬 트랜잭션 24-31
Acquire 메소드 11-8
Action List Editor 8-18
Action Manager 8-18, 8-19, 8-21
  정의 8-17
Actions 속성 33-5
Active Document 38-10, 38-14
  참조 IOleDocumentSite 인터페이스
Active Server Object 42-1 ~ 42-8
  in-process 서버 42-7
  out-of-process 서버 42-7
  등록 42-7 ~ 42-8
  디버깅 42-8
  만들기 42-2 ~ 42-7
Active Server Object 마법사 42-2 ~ 42-3
Active Server Page, ASP *참조*
Active Server Pages, ASP *참조*
Active Template Library ATL *참조*
Active 속성
  데이터셋 22-4
  서버 소켓 37-7
  세션 24-17, 24-18
  클라이언트 소켓 37-6
ActiveAggs 속성 27-13
ActiveFlag 속성 20-19
ActiveForm 43-6 ~ 43-7
  마법사 43-6 ~ 43-7
  만들기 43-2
ActiveForms
  InternetExpress와 비교 29-29
  데이터베이스 웹 애플리케이션 29-30
  멀티 티어 애플리케이션 29-30
ActiveX 38-13 ~ 38-14, 43-1
  ASP와 비교 42-7
  InternetExpress와 비교 29-29
  웹 애플리케이션 38-14, 43-1, 43-16 ~ 43-18
  인터페이스 38-20
ActiveX 컨트롤 17-5, 38-10, 38-13, 43-1 ~ 43-18
  Automation 호환 타입 사용 43-4, 43-8
  HTML에 포함 33-14
  VCL 컨트롤에서 43-4 ~ 43-7
  데이터 인식 40-8 ~ 40-10, 43-8, 43-11 ~ 43-13
  등록 43-15 ~ 43-16
  디버깅 43-16
  디자인 43-4
  라이센스 43-5, 43-7
  마법사 43-4 ~ 43-5
  만들기 43-2, 43-4 ~ 43-7
  메소드 추가 43-9 ~ 43-10
  속성 추가 43-9 ~ 43-10
  속성 페이지 40-7, 43-3, 43-13 ~ 43-15
  스레드 모델 43-5
  영구적 속성 43-13
  요소 43-2 ~ 43-3
  웹 배포 43-16 ~ 43-18
  웹 애플리케이션 38-14, 43-1, 43-16 ~ 43-18
  이벤트 발생 43-11
  이벤트 처리 43-10 ~ 43-11
  인터페이스 43-8 ~ 43-13
  임포트 40-4 ~ 40-5
  컴포넌트 랩퍼 40-6, 40-7, 40-8 ~ 40-10
    _OCX 유닛 40-4
  타입 라이브러리 38-17, 43-3
ActiveX 페이지(컴포넌트 팔레트) 5-8, 40-5
ActnList 유닛 8-28
Adapter Dispatcher 34-9
AdapterPageProducer 34-10
Add Fields 다이얼로그 박스 23-4
Add ~ Repository 명령 7-24
Add 메소드
  메뉴 8-40
  문자열 4-18
  영구적 열 19-18
AddAlias 메소드 24-24
AddFieldDef 메소드 22-38
AddFontResource 함수 17-14
AddIndexDef 메소드 22-38
Additional 페이지(컴포넌트 팔레트) 5-6
AddObject 메소드 4-19
AddParam 메소드 22-51
AddPassword 메소드 24-21
AddRef 메소드 38-4
Address 속성, TSocketConnection 29-23
AddStandardAlias 메소드 24-24
AddStrings 메소드 4-18, 4-19
ADO 18-2, 22-2, 25-1, 25-2
  데이터 저장소 25-2, 25-3
  리소스 디스펜서 44-6
  배포 17-6
  암시적 트랜잭션 25-6 ~ 25-7
  컴포넌트 25-1 ~ 25-19
    개요 25-1 ~ 25-2
  프로바이더 25-3
ADO 객체 25-1
  RDS DataSpace 25-16
  레코드셋 25-8, 25-10
  연결 객체 25-4

ADO 데이터셋 25-8 ~ 25-16
데이터 파일 25-14 ~ 25-15
배치 업데이트 25-11 ~ 25-14
비동기 페치 25-11
연결 25-9 ~ 25-10
인덱스 기반 검색 22-27
ADO 명령 25-7, 25-16 ~ 25-19
데이터 검색 25-18
매개변수 25-18 ~ 25-19
반복 21-12
비동기 25-18
실행 25-17
지정 25-17
취소 25-17 ~ 25-18
ADO 연결 25-2 ~ 25-8
데이터 저장소에 연결 25-2 ~ 25-7
명령 실행 25-5
비동기 25-5
시간 제한 25-5
이벤트 25-7 ~ 25-8
ADO 페이지(컴포넌트 팔레트) 5-7, 18-2, 25-1
ADT 필드 23-22, 23-23 ~ 23-24
단순화 19-22
영구적 필드 23-23
표시 19-22, 23-23
ADTG 파일 25-14
AfterApplyUpdates 이벤트 27-31, 28-8
AfterCancel 이벤트 22-21
AfterClose 이벤트 22-4
AfterConnect 이벤트 21-3, 29-26
AfterConstruction 13-15
AfterDelete 이벤트 22-19
AfterDisconnect 이벤트 21-4, 29-26
AfterDispatch 이벤트 33-5, 33-8
AfterEdit 이벤트 22-17
AfterGetRecords 이벤트 28-7
AfterInsert 이벤트 22-18
AfterOpen 이벤트 22-4
AfterPost 이벤트 22-20
AfterScroll 이벤트 22-5
AggFields 속성 27-13
Aggregates 속성 27-11, 27-13
AliasName 속성 24-13
Align 속성 8-4
상태 표시줄 9-14
패널 8-43
Alignment 속성 9-6
decision grid 20-12
데이터 그리드 19-20
데이터 인식 메모 컨트롤 19-8
메모 및 리치 에디트(rich edit) 컨트롤 9-3
상태 표시줄 9-14
열 헤더 19-20
텍스트 컨트롤 6-7
필드 23-11
AllowAllUp 속성 9-7
스피드 버튼 8-44
툴 버튼 8-46
AllowDelete 속성 19-28
AllowGrayed 속성 9-8
AllowInsert 속성 19-28
alTop 상수 8-43
ANSI C
구현 규정 A-1
날짜 및 시간 형식 A-10
멀티바이트 문자 A-2
메인 함수 A-2
진단 A-1
표준, 구현 A-1
하이픈, 해석 A-8
ANSI 문자 집합 16-2
확장 A-2
AnsiString 52-8
Apache DLL 17-10
배포 17-10
Apache 서버 DLL 32-7
생성 33-1, 34-8
Apache 애플리케이션 32-7
디버깅 32-10
생성 33-1, 34-8
Apartment 스레드 41-7 ~ 41-8
Append 메소드 22-18, 22-19
Insert과 비교 22-18
AppendRecord 메소드 22-21
Application Adapter 34-9
Application 변수 8-1
Apply 메소드 24-44
ApplyRange 메소드 22-33
ApplyUpdates 메소드 14-25, 24-32
BDE 데이터셋 24-35
TDatabase 24-34
TXMLTransformClient 30-10
클라이언트 데이터셋 25-12, 27-6, 27-19, 27-20 ~ 27-21, 28-3
프로바이더 27-20, 28-3, 28-8
AppNamespacePrefix 변수 36-3
AppServer 속성 27-32, 28-3, 29-16, 29-27
Arc 메소드 10-4
array of const 13-18
ARRAYSIZE 매크로 13-17
as 연산자 13-22
AS_ApplyUpdates 메소드 28-3
AS_ATTRIBUTE 36-7
AS_DataRequest 메소드 28-3
AS_Execute 메소드 28-3
AS_GetParams 메소드 28-3
AS_GetProviderNames 메소드 28-3
AS_GetRecords 메소드 28-3
AS_RowRequest 메소드 28-3
ASCII 코드 A-2
ASCII 테이블 24-5
ASP 38-13, 42-1 ~ 42-8
ActiveX와 비교 42-7
HTML 문서 42-1
UI 디자인 42-1
Web Broker와 비교 42-1
성능 제한 42-1
스크립트 언어 38-13, 42-3
페이지 작성 42-3
ASP 내장 함수 42-3 ~ 42-6
Application 객체 42-3 ~ 42-4
Request 객체 42-4
Response 객체 42-5
Server 객체 42-6
Session 객체 42-5 ~ 42-6
액세스 42-2 ~ 42-3
Assign Local Data 명령 27-13
Assign 메소드
문자열 리스트 4-19
AssignedValues 속성 19-21
AssignValue 메소드 23-16
Associate 속성 9-5
as-soon-as-possible 비활성화 29-7
ATL 38-22 ~ 38-23
옵션 41-8
헤더 파일 38-22
호출 추적 41-8, 41-17

Attributes 속성
TADOConnection 25-6
매개변수 22-45, 22-51
auto_ptr 12-5
AutoCalcFields 속성 22-22
AutoComplete 속성 29-8
AutoDisplay 속성 19-9
AutoEdit 속성 19-5
AutoHotKeys 속성 8-33
Automation
Active Server Object 42-2
IDispatch 인터페이스 41-13
인터페이스 41-12 ~ 41-14
초기 연결 38-18
최적화 38-18
타입 설명 38-12
타입 호환성 39-11, 41-14 ~ 41-15
후기 연결 41-13
Automation 객체 38-12
마법사 41-4 ~ 41-8
컴포넌트 랩퍼 40-7 ~ 40-8
예제 40-10 ~ 40-12
Automation 서버 38-10, 38-12 ~ 38-13
객체 액세스 41-13
타입 라이브러리 38-17
Automation 컨트롤러 38-12, 40-1, 40-13 ~ 40-16, 41-13
dispatch 인터페이스 40-13 ~ 40-14
객체 만들기 40-13
예제 40-10 ~ 40-12
이벤트 40-14 ~ 40-16
이중 인터페이스 40-13
AutoPopup 속성 8-48
AutoSelect 속성 9-3
AutoSessionName 속성 24-17, 24-29, 33-18
AutoSize 속성 8-4, 9-2, 17-13, 19-8
.AVI 클립 9-17, 10-29, 10-32
.AVI 파일 10-32

B

Background 속성 8-20
Bands 속성 8-47, 9-9
BaseCLX
예외 클래스 12-16
정의 4-1, 14-5
Basic Object Adapter BOA *참조*
BatchMove 메소드 24-7
BDE
리소스 디스펜서 44-6
BDE Administration 유틸리티 24-14, 24-53
BDE 데이터셋 18-1, 22-2, 24-2 ~ 24-12
decision 지원 컴포넌트 20-4
데이터베이스 24-3 ~ 24-4
로컬 데이터베이스 지원 24-5
로컬 테이터베이스 지원 24-7
복사 24-49
세션 24-3 ~ 24-4
일괄 작업 24-47 ~ 24-51
캐싱된 업데이트 적용 24-35
형식 24-2
BDE 페이지(컴포넌트 팔레트) 5-7, 18-1
BeforeApplyUpdates 이벤트 27-30, 28-8
BeforeCancel 이벤트 22-21
BeforeClose 이벤트 22-4
BeforeConnect 이벤트 21-3, 29-26
BeforeDelete 이벤트 22-19
BeforeDestruction 메소드 13-15
BeforeDisconnect 이벤트 21-4, 29-26
BeforeDispatch 이벤트 33-5, 33-7
BeforeEdit 이벤트 22-17
BeforeGetRecords 이벤트 28-7
BeforeInsert 이벤트 22-18
BeforeOpen 이벤트 22-4
BeforePost 이벤트 22-20
BeforeScroll 이벤트 22-5
BeforeUpdateRecord 이벤트 24-32, 24-39, 27-21, 28-11
BEGIN_MESSAGE_MAP 매크로 51-4, 51-7
BeginAutoDrag 메소드 51-13
BeginDrag 메소드 6-1
BeginRead 메소드 11-8
BeginTrans 메소드 21-6
BeginWrite 메소드 11-8
Beveled 9-6
Bind 메소드
TAutoDriver 40-13
BLOB 19-8, 19-9
캐싱 24-4
BLOB 필드 19-2
값 얻기 24-4
값 표시 19-8, 19-9
그래픽 보기 19-9
요청 시 가져오기 28-5
BlockMode 속성 37-9, 37-10
bmBlocking 37-10
BMPDlg 유닛 10-21
bmThreadBlocking 37-9, 37-10
BOA 31-2, 31-4, 31-5
BOA_init 31-7
스레드 충돌 31-11
초기화 31-13
Bof 속성 22-6, 22-8
Bookmark 속성 22-9
BookmarkValid 메소드 22-9
Borland Database Engine 7-15, 18-1, 22-2, 24-1
API 호출 24-1, 24-4
ODBC 드라이버 24-16
데이터 검색 22-46, 24-2, 24-10
데이터베이스 연결 열기 24-19
데이터베이스에 연결 24-12 ~ 24-16
데이터셋 24-2
드라이버 24-1, 24-14
드라이버 이름 24-14
디폴트 연결 속성 24-18
배포 17-8
사용권 요구 사항 17-15
세션 24-16
알리아스 24-3, 24-14, 24-16, 24-24 ~ 24-25
사용 가능성 24-25
삭제 24-25
생성 24-24 ~ 24-25
이질적 쿼리 24-9
지정 24-13, 24-14 ~ 24-15
암시적 트랜잭션 24-29
연결 관리 24-19 ~ 24-21
연결 닫기 24-19
웹 애플리케이션 17-10
유틸리티 24-53
이질적 쿼리 24-9 ~ 24-10
일괄 작업 24-47 ~ 24-51
캐싱된 업데이트 24-31 ~ 24-47
업데이트 오류 24-37

데이블 타입 24-5
BorlandIDEServices 변수 58-7, 58-22
Borland에 문의 1-3
BoundsChanged 메소드 51-14
.bpi 파일 15-2, 15-13
.bpk 파일 15-2, 15-6, 15-8
.bpl 파일 15-1, 15-13, 17-3
briefcase model 18-14
Broadcast 메소드 51-8
Brush 속성 9-17, 10-4, 10-8, 50-3
BrushCopy 메소드 50-3, 50-6
ButtonAutoSize 속성 20-10
ButtonStyle 속성
데이터 그리드 19-20, 19-21
ByteType 4-21

C

C 기반 예외 처리 12-6
C++ 객체 모델 13-1
C++ 예외 처리 12-1
C++와 오브젝트 파스칼 비교
RTTI 13-22
가상 메소드 호출 13-10, 13-14
객체 복사 13-6
객체 생성 13-7
객체 소멸 13-13
랭귀지 대응 부분 13-16
복사 생성자 13-7
부울 타입 13-20
생성자 13-21
차이점 13-15, 13-20
참조 13-5
참조에 의한 전달 13-16
클래스 초기화 13-12
타입이 지정되지 않은 매개변수 13-17
할당 13-6
함수 인수 13-7, 13-19, 13-23
CacheBlobs 속성 24-4
CachedUpdates 속성 14-25, 24-32
calloc 함수 A-9
CanBePooled 메소드 44-9
Cancel 메소드 22-18, 22-20, 25-18
Cancel 속성 9-7
CancelBatch 메소드 14-25, 25-12, 25-14
CancelRange 메소드 22-33
CancelUpdates 메소드 14-25, 24-32, 25-12, 27-6
CanModify 속성
데이터 그리드 19-25
데이터셋 19-5, 22-17, 22-37
쿼리 24-10
Canvas 속성 9-17, 45-7
Caption 속성
decision grid 20-12
그룹 박스와 라디오 그룹 9-12
레이블 9-4
잘못된 입력 8-31
catch 문 12-1, 12-3, 12-16
catch되지 않은 예외 12-6
CComCoClass 38-23, 41-2, 41-4, 42-2
CComModule 38-23
CComObjectRootEx 38-23, 41-2, 41-4, 42-2
CDaudio 디스크 10-32
CellDrawState 함수 20-12
CellRect 메소드 9-15
Cells 속성 9-16
Cells 함수 20-12
CellValueArray 함수 20-12
CGI 애플리케이션 17-10, 32-5, 32-6
생성 33-2, 34-8
Change 메소드 56-12
ChangeCount 속성 14-25, 24-32, 27-5
ChangedTableName 속성 24-51
CHANGEINDEX 27-7
Char 데이터 타입 16-3
Chart Editing 다이얼로그 박스 20-15 ~ 20-17
Chart FX 17-5
CHECK 제약 조건 28-12
Checked 속성 9-8
CheckSynchronize 루틴 11-5
ChildName 속성 29-28
Chord 메소드 10-4
__classid 연산자 13-23
ClassInfo 메소드 13-22
ClassName 메소드 13-22
ClassNameIs 메소드 13-22
ClassParent 메소드 13-22
ClassType 메소드 13-22
Clear 메소드
문자열 리스트 4-19
필드 23-16
ClearSelection 메소드 6-9
Click 메소드 48-2, 51-13
오버라이드 48-6, 55-13
Clipbrd 유닛 6-8, 10-22
clock 함수 A-10
CloneCursor 메소드 27-14
Close 메소드
데이터베이스 연결 24-19
데이터셋 22-4
세션 24-18
연결 컴포넌트 21-4
CloseDatabase 메소드 24-19
CloseDataSets 메소드 21-11
__closure 키워드 13-24
CLSID 38-5, 38-6, 38-16, 40-5
라이센스 패키지 파일 43-7
CLX
VCL과 비교 14-5 ~ 14-7
객체 생성 14-11
시스템 이벤트 51-12 ~ 51-15
시스템 통지 51-10 ~ 51-15
신호 51-10 ~ 51-12
유닛 14-9 ~ 14-11
정의 3-1
CLX 애플리케이션
Linux로 이식 14-2 ~ 14-19
개요 14-1
데이터베이스 애플리케이션 14-19 ~ 14-25
만들기 14-1 ~ 14-2
배포 17-6
인터넷 애플리케이션 14-25 ~ 14-26
clx60.bpl 17-6
CM_EXIT 메시지 56-13
CMExit 메소드 56-13
CoClass
ActiveX 컨트롤 43-4
Type Library Editor 39-9, 39-15 ~ 39-16
만들기 39-13, 40-5, 40-13
선언 40-5
업데이트 39-14
컴포넌트 랩퍼 40-1, 40-3
_OCX 유닛 40-2
제한 사항 40-2
CoClasses 38-6
CLSID 38-6
생성 38-6

이름 지정 41-3, 41-4
Code Insight
템플릿 7-3
COInit 플래그 41-8
ColCount 속성 19-28
Color 속성 9-4, 9-17
decision grid 20-12
데이터 그리드 19-20
브러시 10-8
열 헤더 19-20
펜 10-5, 10-6
ColorChanged 메소드 51-14
Cols 속성 9-16
Columns Editor
열 재정렬 19-18
영구적 열 만들기 19-17
Columns 속성 9-9, 19-17
그리드 19-15
라디오 그룹 9-12
Columns 에디터
열 삭제 19-18
ColWidths 속성 6-14, 9-15
COM 7-19
CORBA와 비교 31-1
개요 38-1 ~ 38-24
대리자 38-7, 38-8
마법사 38-19 ~ 38-24, 41-1
사양 38-1
스텁 38-8
애플리케이션 38-2 ~ 38-10, 38-19
분산 7-19
정의 38-1 ~ 38-2
집합체(aggregation) 38-9
초기 연결 38-17
컨테이너 38-10, 40-1
컨트롤러 38-10, 40-1
클라이언트 38-2, 38-9, 39-13, 40-1 ~ 40-16
확장 38-2, 38-10 ~ 38-12
COM 객체 38-2, 38-5 ~ 38-9, 41-1 ~ 41-18
등록 41-16
디버깅 41-8, 41-17
디자인 41-2
마법사 41-2 ~ 41-4, 41-5 ~ 41-8
만들기 41-15
생성 41-1
스레드 모델 41-5 ~ 41-8
인터페이스 38-3, 41-9 ~ 41-14
집계 38-9
컴포넌트 랩퍼 40-1, 40-2, 40-3, 40-4, 40-6 ~ 40-12
COM 라이브러리 38-2
COM 서버 38-2, 38-5 ~ 38-9, 41-1 ~ 41-18
in-process 38-6
out-of-process 서버 38-6
디자인 41-2
스레드 모델 41-6, 41-8
원격 38-6
인스턴싱 41-8
최적화 38-18
COM 인터페이스 38-3 ~ 38-4, 41-3
Automation 41-12 ~ 41-14
IUnknown 38-4
구현 38-6, 38-24
디스패치 식별자 41-13
마샬링 38-9
속성 39-9
수정 39-14 ~ 39-15, 41-9 ~ 41-11
이중 인터페이스 41-12 ~ 41-13
인터페이스 포인터 38-4
최적화 38-18
타입 라이브러리에 추가 39-14
타입 정보 38-16
COM+ 7-19, 29-6, 38-10, 38-14, 44-1
Component Manager 44-28
in-process 서버 38-7
MTS와 비교 44-1
객체 풀링 44-9 ~ 44-10
애플리케이션 44-6, 44-28
이벤트 40-15 ~ 40-16, 44-21 ~ 44-25
이벤트 객체 44-23 ~ 44-24
이벤트 구독자 객체 44-24
인터페이스 포인터 38-5
트랜잭션 29-17
트랜잭션 객체 38-14 ~ 38-15
트랜잭션 객체 *참조*
호출 동기화 44-21
활동 구성 44-21
COMCTL32.DLL 8-42
Command Text Editor 22-43
CommandCount 속성 21-12, 25-7
Commands 속성 21-12, 25-7
CommandText 속성 22-43, 25-15, 25-16, 25-17, 25-18, 26-6, 26-7, 27-31
CommandTimeout 속성 25-5, 25-18
CommandType 속성 25-15, 25-16, 25-17, 26-5, 26-6, 26-7, 27-31
Commit 메소드 21-8
CommitTrans 메소드 21-8
CommitUpdates 메소드 14-25, 24-32, 24-35
Common Object Request Broker Architecture CORBA *참조*
CompareBookmarks 메소드 22-9
ComputerName 속성 29-23
ConfigMode 속성 24-25
ConnectEvents 메소드 40-15
Connection Editor 26-5
Connection Points 맵 41-10
Connection String Editor 25-4
Connection 속성 25-3, 25-9
ConnectionBroker 27-25
ConnectionName 속성 26-4
ConnectionObject 속성 25-4
ConnectionString 속성 21-2, 21-4, 25-3, 25-9
ConnectionTimeout 속성 25-5
ConnectOptions 속성 25-5
Console Wizard 7-4
CONSTRAINT 제약 조건 28-12
ConstraintErrorMessage 속성 23-11, 23-21, 23-22
Constraints 속성 8-4, 27-7, 28-13
Contains 리스트 (패키지) 15-7
Contains 리스트(패키지) 15-6, 15-8, 15-10, 52-19
Content 메소드
페이지 프로듀서 33-15
Content 속성
웹 응답 객체 33-12
ContentFromStream 메소드
페이지 프로듀서 33-15
ContentFromString 메소드
페이지 프로듀서 33-15
ContentStream 속성

웹 응답 객체 33-12, 33-13
ContextHelp 7-34
contravariance 13-24
ControlType 속성 20-9, 20-15
Convert 함수 4-26, 4-27, 4-28, 4-30, 4-33
ConvUtils 유닛 4-26
Copy(Object Repository) 7-25
CopyFile 함수 4-9
CopyFrom
TStream 4-3
CopyMode 속성 50-3
CopyRect 메소드 10-4, 50-3, 50-6
CopyToClipboard 메소드 6-9
그래픽 19-9
데이터 인식 메모 컨트롤 19-9
CORBA 31-1 ~ 31-17
COM과 비교 31-1
IDL 파일 31-5
VCL 31-8, 31-13
개요 31-1 ~ 31-4
객체 구현 31-6, 31-9 ~ 31-12
객체 인스턴스화 31-7
스레드 31-11
위임 모델 31-8
자동으로 생성된 코드 31-9
클라이언트 요청 받기 31-6
클라이언트 요청 승인 31-8
테스트 31-16 ~ 31-17
표준 31-1 ~ 31-17
CORBA Client 마법사 31-13
CORBA Object 마법사 31-6 ~ 31-7, 31-13
CORBA Server 마법사 31-5
CORBA 객체
연결 31-15
인터페이스 정의 31-5 ~ 31-12
일반적 31-15
CORBA 애플리케이션
VCL 사용 31-8, 31-13
개요 31-1 ~ 31-4
서버 31-4 ~ 31-12
클라이언트 31-13 ~ 31-16
Count 속성
TSessionList 24-28
문자열 리스트 4-18
CP32MT.lib RTL 라이브러리 12-18
cp32mti.lib 임포트 라이브러리 12-18
.cpp 파일 15-2, 15-13
Create Data Set 명령 22-38
Create Submenu 명령(메뉴 디자이너) 8-34, 8-36
CREATE TABLE 21-10
Create Table 명령 22-38
CreateDataSet 메소드 22-38
CreateObject 메소드 42-3
CreateParam 메소드 27-27
CreateSharedPropertyGroup 44-6
CreateSuspended 매개변수 11-11
CreateTable 메소드 22-38
CreateTransactionContextEx
예제 44-14 ~ 44-15
crtl.dcu 17-6
Currency 속성
필드 23-11
CursorChanged 메소드 51-14
CursorType 속성 25-12
CurValue 속성 28-11
Custom 속성 29-38
CustomConstraint 속성 23-11, 23-21, 27-7
CutToClipboard 메소드 6-9
그래픽 19-9
데이터 인식 메모 컨트롤 19-9
cw32mt.lib RTL 라이브러리 12-18
cw32mti.lib 임포트 라이브러리 12-18

D

-D 링커 옵션 15-12
Data Access 페이지(컴포넌트 팔레트) 5-7, 18-2, 29-2
Data Bindings Editor 40-8 ~ 40-9
Data Controls 페이지(컴포넌트 팔레트) 5-7, 18-15, 19-1, 19-2
Data Definition Language 21-10, 22-41, 24-8, 26-10
Data Dictionary 23-12 ~ 23-14, 24-51 ~ 24-52, 29-3
제약 조건 28-13
Data Manipulation Language 21-10, 22-41, 24-8
Data 속성 27-5, 27-14, 27-15, 27-33
Database Desktop 24-53
Database Explorer 24-14, 24-53
Database Properties Editor 24-14
연결 매개변수 보기 24-15
Database 매개변수 26-4
DatabaseCount 속성 24-20
DatabaseName 속성 21-2, 24-3, 24-14
이질적 쿼리 24-9
Databases 속성 24-20
DataChange 메소드 56-12
DataCLX
정의 14-5
DataField 속성 19-10, 56-6
조회 리스트 박스 및 콤보 박스 19-12
DataRequest 메소드 27-31, 28-3
DataSet 속성 21-12
데이터 그리드 19-16
프로바이더 28-2
DataSetCount 속성 21-12
DataSetField 속성 22-36
DataSnap 페이지(컴포넌트 팔레트) 5-7, 29-2, 29-5, 29-6
DataSource 속성
ActiveX 컨트롤 40-8
데이터 그리드 19-16
데이터 인식 컨트롤 56-6
데이터 탐색기 19-30
조회 리스트 박스 및 콤보 박스 19-12
쿼리 22-46
DataType 속성
매개변수 22-44, 22-50, 22-51
__DATE__ 매크로
사용 가능성 A-6
DateTimePicker 컴포넌트 9-11
Day 속성 55-6
DB/2 드라이버
배포 17-9
dBASE 테이블 24-5
DatabaseName 24-3
데이터 액세스 24-9
레코드 추가 22-19
로컬 트랜잭션 24-31
암호 보호 24-21
암호 사용 24-23
이름 변경 24-7
인덱스 24-6
DBChart 컴포넌트 18-15

DBCheckBox 컴포넌트 19-2, 19-13
DBComboBox 컴포넌트 19-2, 19-10 ~ 19-11
DBConnection 속성 27-16
DBCtrlGrid 컴포넌트 19-2, 19-27 ~ 19-28
  속성 19-28
DBEdit 컴포넌트 19-2, 19-8
dbExpress 17-7, 18-2, 26-1 ~ 26-2
  드라이버 26
  디버깅 26-17 ~ 26-18
  메타데이터 26-12 ~ 26-16
  배포 26-1
  컴포넌트 26-1 ~ 26-18
  크로스 플랫폼 애플리케이션 14-19 ~ 14-24
dbExpress 애플리케이션 17-10
dbExpress 페이지(컴포넌트 팔레트) 5-7, 18-2, 26-2
dbGo 25-1
DBGrid 컴포넌트 19-2, 19-15 ~ 19-27
  속성 19-20
  이벤트 19-26
DBGridColumns 컴포넌트 19-15
DBImage 컴포넌트 19-2, 19-9
DblClick 메소드 51-13
DBListBox 컴포넌트 19-2, 19-10 ~ 19-11
DBLogDlg 유닛 21-4
DBLookupComboBox 컴포넌트 19-2, 19-11 ~ 19-12
DBLookupListBox 컴포넌트 19-2, 19-11 ~ 19-12
DBMemo 컴포넌트 19-2, 19-8 ~ 19-9
DBMS 29-1
DBNavigator 컴포넌트 19-2, 19-28 ~ 19-30
DBRadioGroup 컴포넌트 19-2, 19-13 ~ 19-14
DBRichEdit 컴포넌트 19-2, 19-9
DBSession 속성 24-3
DBText 컴포넌트 19-2, 19-8
dbxconnections.ini 26-4, 26-5
dbxdrivers.ini 26-3
DCOM 38-6, 38-8
  InternetExpress 애플리케이션 29-33
  멀티 티어 애플리케이션 29-9
  분산 애플리케이션 7-19
  애플리케이션 서버에 연결 27-25, 29-23
DCOM 연결 29-9, 29-23
DCOMCnfg.exe 29-33
.dcr 파일 52-4
  비트맵 45-14
DDL 21-10, 22-41, 22-47, 24-8, 26-10
decision cube 20-7 ~ 20-8
  데이터 얻기 20-4
  데이터 표시 20-9, 20-11
  드릴다운 20-5, 20-9, 20-11, 20-20
  디자인 옵션 20-8
  메모리 관리 20-8
  부분합 20-5
  새로 고침 20-7
  속성 20-7
  차원
    열기/닫기 20-9
    페이징된 20-20
  차원 맵 20-5, 20-7, 20-19
  피벗팅 20-5, 20-9
Decision Cube Editor 20-7 ~ 20-8
  Memory Control 20-8
  차원 설정 20-7
  큐브 용량 20-19
Decision Cube 페이지(컴포넌트 팔레트) 18-15, 20-1
decision 그래프 20-13 ~ 20-17
  그래프 타입 20-16
  데이터 시리즈 20-17
  런타임 동작 20-19
  사용자 정의 20-15 ~ 20-17
  차원 20-13
  템플릿 20-16
  표시 옵션 20-14
  피벗 상태 20-8, 20-9
decision grid 20-10 ~ 20-12
  런타임 동작 20-18
  속성 20-12
  이벤트 20-12
  차원
    드릴다운 20-11
    선택 20-12
    열기/닫기 20-11
    재정렬 20-11
  피벗 상태 20-8, 20-9, 20-11
decision pivot 20-9 ~ 20-10
  런타임 동작 20-18
  방향 20-10
  속성 20-10
  차원 버튼 20-10
Decision Query Editor 20-6
decision query, 정의 20-6
decision source 20-8 ~ 20-9
  속성 20-9
  이벤트 20-9
decision 데이터셋 20-4 ~ 20-6
decision 지원 컴포넌트 18-15, 20-1 ~ 20-20
  데이터 할당 20-4 ~ 20-6
  디자인 옵션 20-8
  런타임 20-18 ~ 20-19
  메모리 관리 20-19
  추가 20-3 ~ 20-4
__declspec 13-2
__declspec(delphirtti) 36-2
__declspec 키워드 7-12, 13-28
DECnet 프로토콜(Digital) 37-1
default
  키워드 47-7
Default 속성
  액션 항목 33-7
Default 체크 박스 7-3
DEFAULT_ORDER 인덱스 27-7
DefaultColWidth 속성 9-15
DefaultDatabase 속성 25-4
DefaultDrawing 속성 6-12, 19-26
DefaultExpression 속성 23-20, 27-7
DefaultHandler 메소드 51-3
DefaultPage 속성 34-40
DefaultRowHeight 속성 9-15
Delete Table 명령 22-39
Delete Templates 다이얼로그 박스 8-38
Delete Templates 명령(메뉴 디자이너) 8-37, 8-38
Delete 메소드 22-19
  문자열 리스트 4-19
Delete 명령(메뉴 디자이너) 8-36
DELETE 문 24-39, 24-42, 28-9
DeleteAlias 메소드 24-25
DeleteFile 함수 4-7

DeleteFontResource 함수 17-14
DeleteIndex 메소드 27-9
DeleteRecords 메소드 22-40
DeleteSQL 속성 24-39
DeleteTable 메소드 22-39
delphiclass 인수 13-28
DelphiInterface 클래스 13-3, 13-20
delphireturn 인수 13-29
delphirtti 인수 13-29
Delta 속성 27-5, 27-19
DEPLOY 문서 17-8, 17-9
DEPLOY 설명서 17-15
DeviceType 속성 10-31
.dfm 파일 14-2, 16-10
  생성 16-12
Dialogs 페이지(컴포넌트 팔레트) 5-8
DII 31-13, 31-15
  Interface Repository 31-12
DimensionMap 속성 20-5, 20-7
Dimensions 속성 20-12
Direction 속성
  매개변수 22-45, 22-51
Directory 지시어 17-11
DirtyRead 21-9
DisableCommit 메소드 44-13
DisableConstraints 메소드 27-29
DisableControls 메소드 19-6
DisabledImages 속성 8-46
DisconnectEvents 메소드 40-15
Dispatch Actions 34-9
Dispatch 메소드 51-3, 51-5
dispatch 인터페이스 41-12, 41-13
  메소드 호출 40-13 ~ 40-14
  식별자 41-13
  타입 라이브러리 39-9
  타입 호환성 41-14
dispID 38-16, 40-14, 41-13
  연결 41-14
dispinterface 41-12, 41-13
  동적 연결 39-9
  타입 라이브러리 39-9
DisplayFormat 속성 19-25, 23-11, 23-15
DisplayLabel 속성 19-16, 23-11
DisplayWidth 속성 19-16, 23-11
DLL 7-12
  Apache 17-10
  COM 서버 38-6
  스레드 모델 41-6
  HTML에 포함 33-14
  HTTP 서버 32-6
  Linux .so 파일 *참조*
  MTS 44-2
  국제화 16-10, 16-12
  배포 17-10
  생성 7-10, 7-11
  설치 17-5
  연결 7-15
  패키지 15-1, 15-2, 15-11
dllexport 7-11, 7-12
DllGetClassObject 44-3
dllimport 7-11
DllRegisterServer 44-3
DML 21-10, 22-41, 24-8
.dmt 파일 8-38, 8-39
DocumentElement 속성 35-3
DoExit 메소드 56-14
DOM 35-2, 35-2 ~ 35-3
  구현 35-2
    사용 35-3
DoMouseWheel 메소드 51-13
Down 속성 9-7
  스피드 버튼 8-44
.dpl 파일 15-2
DragMode 속성 6-1
  그리드 19-19
DragOver 메소드 51-13
Draw 메소드 10-4, 50-3, 50-6
DrawShape 10-15
drintf 유닛 24-52
DriverName 속성 24-14, 26-3
DropConnections 메소드 24-13, 24-20
DropDownCount 속성 9-10, 19-11
DropDownMenu 속성 8-48
DropDownRows 속성
  데이터 그리드 19-20, 19-21
  조회 콤보 박스 19-12
dynamic 인수 13-29

E

EBX 레지스터 14-8, 14-18
Edit 메소드 22-17, 52-9, 52-10
EditFormat 속성 19-25, 23-11, 23-15
EditKey 메소드 22-27, 22-29
EditMask 속성 23-14
  필드 23-11
EditRangeEnd 메소드 22-33
EditRangeStart 메소드 22-33
Ellipse 메소드 10-4, 10-11, 50-3
Embed HTML 태그 (<EMBED>) 33-14
EmptyDataSet 메소드 22-40, 27-26
EmptyTable 메소드 22-39
EnableCommit 메소드 44-13
EnableConstraints 메소드 27-29
EnableControls 메소드 19-6
Enabled 속성
  데이터 소스 19-4, 19-5
  데이터 인식 컨트롤 19-7
  메뉴 6-9, 8-40
  스피드 버튼 8-44
  액션 항목 33-7
EnabledChanged 메소드 51-14
End User Adapter 34-9
END_MESSAGE_MAP 매크로 51-4, 51-7
EndRead 메소드 11-8
EndWrite 메소드 11-8
Eof 속성 22-6, 22-7
EOF 표시 4-4
EReadError 4-2
ERemotableException 36-14, 36-15
EventFilter 메소드
  시스템 이벤트 51-14
EWriteError 4-2
__except 키워드 12-7, 12-8
Exception
  정의 3-5
__except 키워드 12-10
Exclusive 속성 24-6
ExecProc 메소드 22-52, 26-10
ExecSQL 메소드 22-47, 26-10
  업데이트 객체 24-45
Execute 11-6
Execute 메소드
  ADO 명령 25-17, 25-19
  TBatchMove 24-50
  다이얼로그 박스 8-15, 57-5
  스레드 11-4
  연결 컴포넌트 21-10
  클라이언트 데이터셋 27-27, 28-3
  프로바이더 28-3

ExecuteOptions 속성 25-11
ExecuteTarget 메소드 8-28
EXISTINGARRAY 매크로 13-17, 13-19
exit 함수 A-9
Expandable 속성 19-23
Expanded 속성
  데이터 그리드 19-20
  열 19-22, 19-23
Expression 속성 27-11
ExprText 속성 23-10

F

FastNet 페이지(컴포넌트 팔레트) 5-7
Fetch Params 명령 27-27
FetchAll 메소드 14-25, 24-32
FetchBlobs 메소드 27-26, 28-3
FetchDetails 메소드 27-26, 28-3
FetchOnDemand 속성 27-26
FetchParams 메소드 27-27, 28-3
fgetpos 함수 A-8
Field Link 디자이너 22-34
FieldAddress 메소드 13-22
FieldByName 메소드 22-31, 23-20
FieldCount 속성
  영구적 필드 19-17
FieldDefs 속성 22-38
FieldKind 속성 23-11
FieldName 속성 23-5, 23-11, 29-38
  decision grid 20-12
  데이터 그리드 19-20, 19-21
  영구적 필드 19-17
Fields Editor 7-23, 23-3
  속성 집합 정의 23-13
  속성 집합 제거 23-14
  열 재정렬 19-18
  영구적 필드 만들기 23-4 ~ 23-5, 23-5 ~ 23-10
  영구적 필드 삭제 23-10
  제목 표시줄 23-4
  탐색 버튼 23-4
  필드 리스트 23-4
  필드 속성 적용 23-13
Fields 속성 23-19
FieldValues 속성 23-19
FileAge 함수 4-9
FileExists 함수 4-7
FileGetDate 함수 4-9
FileName 속성
  클라이언트 데이터셋 18-9, 27-33, 27-34
FileSetDate 함수 4-9
FillRect 메소드 10-4, 50-3
Filter 속성 22-13, 22-14 ~ 22-15
Filtered 속성 22-13
FilterGroup 속성 25-12, 25-13
FilterOnBookmarks 메소드 25-10
__finally 키워드 12-7, 12-13
FindClose 함수 4-7
FindDatabase 메소드 24-20
FindFirst 메소드 22-16
FindFirst 함수 4-7
FindKey 메소드 22-27, 22-28
  EditKey와 비교 22-29
FindLast 메소드 22-16
FindNearest 메소드 22-27, 22-28
FindNext 메소드 22-16
FindNext 함수 4-7
FindPrior 메소드 22-16
FindResourceHInstance 함수 16-11
FindSession 메소드 24-28
FireOnChanged 43-13
FireOnRequestEdit 43-12
First Impression 17-5
First 메소드 22-6
FixedColor 속성 9-15
FixedCols 속성 9-15
FixedOrder 속성 8-47, 9-9
FixedRows 속성 9-15
FixedSize 속성 9-9
FlipChildren 메소드 16-7
FloodFill 메소드 10-4, 50-3
fmod 함수 A-7
FocusControl 메소드 23-16
FocusControl 속성 9-4
Font 속성 9-2, 9-4, 10-4, 50-3
  데이터 그리드 19-20
  데이터 인식 메모 컨트롤 19-8
  열 헤더 19-20
FontChanged 메소드 51-14
Footer 속성 33-20
FOREIGN KEY 제약 조건 28-13
Format 속성 20-12
Formula One 17-5
Found 속성 22-16
FoxPro 테이블
  로컬 트랜잭션 24-31
fprintf 함수 A-8
FrameRect 메소드 10-4
FReadOnly 56-9
Free 스레드 41-6 ~ 41-7
FreeBookmark 메소드 22-9
Free-threaded 마샬러 41-7
FromCommon 4-30
fscanf 함수 A-8
ftell 함수 A-8

G

GDI 애플리케이션 45-7, 50-1
Generate Event Support Code 41-10
GetAliasDriverName 메소드 24-26
GetAliasNames 메소드 24-26
GetAliasParams 메소드 24-26
GetAttributes 메소드 52-10
GetBookmark 메소드 22-9
GetConfigParams 메소드 24-26
GetData 메소드
  필드 23-16
GetDatabaseNames 메소드 24-26
GetDriverNames 메소드 24-26
GetDriverParams 메소드 24-26
getenv 함수 A-9
GetExceptionCode 함수 12-7
GetExceptionInformation 함수 12-7, 12-8
GetFieldByName 메소드 33-9
GetFieldNames 메소드 21-13, 24-26
GetFloatValue 메소드 52-9
GetGroupState 메소드 27-10
GetHandle 7-29
GetHelpFile 7-29
GetHelpStrings 7-30
GetIDsOfNames 메소드 40-14, 41-13
GetIndexNames 메소드 21-13, 22-26
GetMethodValue 메소드 52-9
GetNextPacket 메소드 14-25, 24-32, 27-25, 27-26, 28-3
GetOptionalParam 메소드 27-15, 28-6

GetOrdValue 메소드 52-9
GetPalette 메소드 50-5
GetParams 메소드 28-3
GetPassword 메소드 24-22
GetProcAddress 7-11
GetProcedureNames 메소드 21-13
GetProcedureParams 메소드 21-13
GetProperties 메소드 52-10
GetRecords 메소드 28-3, 28-7
GetSessionNames 메소드 24-28
GetStoredProcNames 메소드 24-26
GetStrValue 메소드 52-9
GetTableNames 메소드 21-12, 24-26
GetValue 메소드 52-9
GetVersionEx 함수 17-14
GetViewerName 7-28
GetXML 메소드 30-9
-Gi 링커 옵션 15-12
-Gl 링커 옵션 15-12
Glyph 속성 8-44, 9-7
GNU make 유틸리티
  Linux 14-14
GNU 어셈블러
  Linux 14-16
GotoBookmark 메소드 22-9
GotoCurrent 메소드 22-40
GotoKey 메소드 22-27, 22-28
GotoNearest 메소드 22-27, 22-28
-Gpd 링커 옵션 15-12
-Gpr 링커 옵션 15-12
Graph Custom Control 17-5
Graphic 속성 10-18, 10-22, 50-4
GridLineWidth 속성 9-15
Grouped 속성
  툴 버튼 8-46
GroupIndex 속성 9-7
  메뉴 8-40
  스피드 버튼 8-44
GroupLayout 속성 20-10
Groups 속성 20-10
GUI 애플리케이션 8-1
GUID 38-3, 39-8, 40-5

H

.h 파일 15-2, 15-13
Handle 속성 4-7, 37-6, 45-3, 45-5, 50-3
  장치 컨텍스트 10-1, 10-2
HANDLE_MSG 매크로 51-2
HandleException 메소드 51-3
HandleShared 속성 24-16
HandlesTarget 메소드 8-28
HasConstraints 속성 23-11
HasFormat 메소드 6-10, 10-23
Header 속성 33-20
Height 속성 8-3
  리스트 박스 19-10
Help Manager 7-27, 7-27 ~ 7-36
HelpContext 7-34
HelpContext 속성 7-33, 9-15
helper 객체 4-1
HelpFile 속성 7-34, 9-15
HelpIntfs 유닛 7-27
HelpKeyword 7-34
HelpKeyword 속성 7-33
HelpSystem 7-34
HelpType 7-33, 7-34
HelpType 속성 7-33
hidesbase 인수 13-29
Hint 속성 9-15
Hints 속성 19-30
HookEvents 메소드 51-11
HorzScrollBar 9-5
Host 속성
  TSocketConnection 29-23
HotImages 속성 8-46
HotKey 속성 9-6
HTML Result 탭 34-2
HTML Script 탭 34-2
HTML 명령 33-14
  데이터베이스 정보 33-18
  생성 33-15
HTML 문서 32-5
  ActiveForm을 위해 생성 43-6
  ASP 42-1
  HTTP 응답 메시지 32-6
  InternetExpress 애플리케이션 29-31
  데이터베이스 33-17
  데이터셋 33-20
  데이터셋 페이지 프로듀서 33-18
  스타일 시트 29-38
  테이블 포함시키기 33-20
  테이블 프로듀서 33-19 ~ 33-21
  템플릿 29-36, 29-38 ~ 29-40, 33-14 ~ 33-15
  페이지 프로듀서 33-14 ~ 33-17
  포함된 ActiveX 컨트롤 43-1
HTML 테이블 33-14, 33-20
  생성 33-19 ~ 33-21
  속성 설정 33-19
  캡션 33-20
HTML 템플릿 29-38 ~ 29-40, 33-14 ~ 33-17, 34-4
  디폴트 29-36, 29-38
HTML 투명 태그
  구문 33-14
  매개변수 33-14
  미리 정의된 29-39, 33-14
  변환 33-14, 33-15
HTML 폼 29-37
HTMLDoc 속성 29-36, 33-15
HTMLFile 속성 33-15
HTTP 32-3
  SOAP 36-1
  개요 32-5 ~ 32-6
  멀티 티어 애플리케이션 29-10 ~ 29-11
  메시지 헤더 32-3
  상태 코드 33-11
  애플리케이션 서버에 연결 29-24
  요청 메시지 요청 메시지 *참조*
  요청 헤더 32-4, 33-9, 42-4
  응답 메시지 응답 메시지 *참조*
  응답 헤더 33-12, 42-5
HTTP 요청
  이미지 34-38
HTTP 응답
  액션 34-37
  이미지 34-38
httpd.conf 17-10
httpsrvr.dll 29-10, 29-13, 29-24
HyperHelp 뷰어 7-27

I

IApplicationObject 인터페이스 42-3
IAppServer 인터페이스 27-30, 27-32, 28-3 ~ 28-4, 29-4, 29-5
  XML 브로커 29-32

로컬 프로바이더 28-3
상태 정보 29-19
원격 프로바이더 28-3
트랜잭션 29-17
호출 29-27
확장 29-16
IAppServerSOAP 인터페이스 29-5, 29-25
IConnectionPoint 인터페이스 40-15, 41-11
IConnectionPointContainer 인터페이스 40-15, 41-11
IConnectionPointContainerImpl 구현 41-11
ICustomHelpViewer 7-27, 7-28, 7-29
구현 7-28
IDataIntercept 인터페이스 29-24
IDE
버튼 삭제 58-9 ~ 58-10
사용자 정의 58-1
이미지 추가 58-8
작업 추가 58-9
IDispatch 인터페이스 38-8, 38-20, 41-12, 41-13
Automation 38-12, 40-14
식별자 41-13, 41-14
IDL 컴파일러 38-19
IDL 파일 31-5
CORBA 서버 마법사 31-5
CORBA 클라이언트 31-13
라이브러리에서 엑스포트 39-19
컴파일 31-6
IDL(Interface Definition Language) 31-5, 38-17, 38-19
Type Library Editor 39-8
IDOMImplementation 35-3
IEEE
반올림 A-4
부동 소수점 형식 A-3
IETF 프로토콜 및 표준 32-3
IExtendedHelpViewer 7-27, 7-31
#ifdef 지시어 14-16
#ifndef 지시어 14-17
IHelpManager 7-27, 7-35
IHelpSelector 7-28, 7-31, 7-32
IHelpSystem 7-27, 7-35
IID 38-3, 40-5
IInterface
구현 13-3
수명 관리 13-5
IInvokable 36-2
IIS 42-1
버전 42-2
Image Editor
사용 45-14
Image HTML 태그(<IMG>) 33-14
ImageIndex 속성 8-45, 8-47
ImageList 8-19
ImageMap HTML 태그 (<MAP>) 33-14
Images 속성
툴 버튼 8-45
IMarshal 인터페이스 41-14, 41-15
IME 16-8
IME(Input method editor) 16-8
ImeMode 속성 16-8
ImeName 속성 16-8
Implementation Repository 31-3
Import ActiveX Control 명령 40-2, 40-4
Import Type Library 명령 40-2, 40-3
ImportedConstraint 속성 23-11, 23-21
Include Unit Hdr 명령 8-3
include 파일
검색 A-6
Increment 속성 9-5
Indent 속성 8-44, 8-46, 8-47, 9-11
Index Files Editor 24-6
Index 속성
필드 23-11
index 예약어 55-8
IndexDefs 속성 22-38
IndexFieldCount 속성 22-26
IndexFieldNames 속성 22-27, 26-7
IndexName과 비교 22-27
IndexFields 속성 22-26
IndexFiles 속성 24-6
IndexName 속성 24-6, 26-7, 27-9
IndexFieldNames와 비교 22-27
IndexOf 메소드 4-18, 4-19
Indy Clients 페이지(컴포넌트 팔레트) 5-8
Indy Misc 페이지(컴포넌트 팔레트) 5-8
Indy Servers 페이지(컴포넌트 팔레트) 5-8
INFINITE 상수 11-11
Informix 드라이버
배포 17-9
Inherit(Object Repository) 7-25
inherited 키워드 13-8, 13-12, 13-14
InheritsFrom 메소드 13-23
.ini 파일 14-6
Win-CGI 애플리케이션 32-7
InitializeControl 메소드 43-11
InitWidget 속성 14-12
INITWIZARD0001 58-21
in-process 서버 38-6
ActiveX 38-13
ASP 42-7
MTS 44-2
Input Mask Editor 23-14
Insert from Resource 다이얼로그 박스 8-41
Insert From Resource 명령(메뉴 디자이너) 8-37, 8-41
Insert From Template 명령(메뉴 디자이너) 8-37, 8-38
Insert Template 다이얼로그 박스 8-38
Insert 메소드 22-18, 22-19
Append와 비교 22-18
메뉴 8-40
문자열 4-18
Insert 명령(메뉴 디자이너) 8-36
INSERT 문 21-11, 24-39, 24-42, 28-9
InsertObject 메소드 4-19
InsertRecord 메소드 22-21
InsertSQL 속성 24-39
Install COM+ objects 명령 44-28
Install Components 다이얼로그 박스 45-19
Install MTS objects 명령 44-28
Install 명령(컴포넌트) 45-19
InstallShield Express 2-5, 17-1
배포
BDE 17-8
SQL Links 17-9
애플리케이션 17-2
패키지 17-3

int 타입 A-3
INTAComponent 58-13
INTAServices 58-7, 58-8, 58-17
IntegralHeight 속성 9-10, 19-10
InterBase 드라이버
  배포 17-9
InterBase 테이블 24-9
InterBase 페이지(컴포넌트 팔레트) 5-7, 18-2
InterBaseAdmin 페이지(컴포넌트 팔레트) 5-7
InterBaseExpress 14-21
__interface 13-2, 36-2
Interface Definition Language
  IDL *참조*
Interface Repository 31-4
  CORBA 인터페이스 등록 31-12
INTERFACE_UUID 매크로 13-2, 36-2
InternalCalc 필드 23-6, 27-10 ~ 27-11
  인덱스 27-9
Internet Engineering Task Force 32-3
Internet Information Server(IIS) 42-1
  버전 42-2
Internet 페이지(컴포넌트 팔레트) 5-7
InternetExpress 7-18, 29-31 ~ 29-40
  ActiveForms와 비교 29-29 ~
InternetExpress 페이지(컴포넌트 팔레트) 5-7
InTransaction 속성 21-7
Invalidate 메소드 54-10
Invoke 메소드 41-13
InvokeRegistry.hpp 36-4
IObjectContext 인터페이스 38-14, 42-3, 44-4, 44-5
  트랜잭션 종료 메소드 44-12
IObjectControl 인터페이스 38-14, 44-2
IOleClientSite 인터페이스 40-16
IOleDocumentSite 인터페이스 40-16
iostreams A-8
IOTAActionServices 58-7
IOTABreakpointNotifier 58-17
IOTACodeCompletionServices 58-7
IOTAComponent 58-13
IOTACreator 58-13
IOTADebuggerNotifier 58-17
IOTADebuggerServices 58-7
IOTAEditLineNotifier 58-17
IOTAEditor 58-12
IOTAEditorNotifier 58-17
IOTAEditorServices 58-7
IOTAFile 58-13, 58-15
IOTAFormNotifier 58-17
IOTAFormWizard 58-3
IOTAIDENotifier 58-17
IOTAKeyBindingServices 58-7
IOTAKeyboardDiagnostics 58-7
IOTAKeyboardServices 58-7
IOTAMenuWizard 58-3
IOTAMessageNotifier 58-17
IOTAMessageServices 58-7
IOTAModule 58-11
IOTAModuleNotifier 58-17
IOTAModuleServices 58-7, 58-12
IOTANotifier 58-17
IOTAPackageServices 58-7
IOTAProcessModNotifier 58-17
IOTAProcessNotifier 58-17
IOTAProjectWizard 58-3
IOTAServices 58-7
IOTAThreadNotifier 58-17
IOTAToDoServices 58-7
IOTAToolsFilter 58-7
IOTAToolsFilterNotifier 58-17
IOTAWizard 58-2, 58-3
IOTAWizardServices 58-7
IP 주소 37-4, 37-6
  호스트 37-4
  호스트 이름과 비교 37-4
IProvideClassInfo 인터페이스 38-17
IProviderSupport 인터페이스 28-2
IPX/SPX 프로토콜 37-1
IRequest 인터페이스 42-4
IResponse 인터페이스 42-5
is 연산자 13-22
isalnum 함수 A-7
isalpha 함수 A-7
ISAPI DLL 17-10
ISAPI 애플리케이션 32-6, 32-7
  디버깅 32-10
  생성 33-1, 34-8
  요청 메시지 33-2
IsCallerInRole 메소드 44-17
iscntrl 함수 A-7
IScriptingContext 인터페이스 42-2
ISecurityProperty 인터페이스 44-18
IServer 인터페이스 42-6
ISessionObject 인터페이스 42-5
islower 함수 A-7
ISpecialWinHelpViewer 7-27
isprint 함수 A-7
IsSecurityEnabled 44-17
isupper 함수 A-7
IsValidChar 메소드 23-16
ItemHeight 속성 9-10
  리스트 박스 19-10
  콤보 박스 19-11
ItemIndex 속성 9-9
  라디오 그룹 9-12
Items 속성
  라디오 그룹 9-12
  라디오 컨트롤 19-13
  리스트 박스 9-9
ITypeComp 인터페이스 38-18
ITypeInfo 인터페이스 38-18
ITypeInfo2 인터페이스 38-18
ITypeLib 인터페이스 38-18
ITypeLib2 인터페이스 38-18
IUnknown 인터페이스 38-3, 38-4, 38-19
  ATL 지원 38-23
  Automation 컨트롤러 41-13
  구현 13-3
  수명 관리 13-5
  호출 추적 41-8
IXMLNode 35-4 ~ 35-5, 35-7

J

javascript 라이브러리 29-31, 29-33
  위치 찾기 29-32, 29-33
just-in-time 활성화 29-7, 44-4 ~ 44-5
  사용 가능 44-5

K

K 각주(도움말 시스템) 52-5

KeepConnection 속성 21-3, 21-11, 24-13, 24-18
KeyDown 메소드 51-13, 56-10
KeyExclusive 속성 22-29, 22-32
KeyField 속성 19-12
KeyFieldCount 속성 22-29
KeyPress 메소드 51-13
KeyString 메소드 51-13
KeyUp 메소드 51-13
KeyViolTableName 속성 24-51
KeywordHelp 7-34
Kind 속성
  비트맵 버튼 9-7

L

Last 메소드 22-6
Layout 속성 9-7
Left 속성 8-3
LeftCol 속성 9-15
.lib 파일 15-2, 15-13
  패키지 15-13
$LIBPREFIX 지시어 7-10
LibraryName 속성 26-3
$LIBSUFFIX 지시어 7-10
$LIBVERSION 지시어 7-10
.lic 파일 43-7
Lines 속성 9-3, 47-8
LineSize 속성 9-5
LineTo 메소드 10-4, 10-8, 10-10, 50-3
Link HTML 태그(<A>) 33-14
Linux
  Windows와 비교 14-13 ~ 14-14
  디렉토리 14-15
  레지스트리 14-13
  배치 파일 14-13
  시스템 통지 51-10 ~ 51-15
  크로스 플랫폼 애플리케이션 14-1 ~ 14-26
List 속성 24-28
ListField 속성 19-12
ListSource 속성 19-12
Loaded 메소드 47-13
LoadFromFile 메소드
  ADO 데이터셋 25-14
  그래픽 10-20, 50-4
  문자열 4-15
  클라이언트 데이터셋 18-9, 27-33
LoadFromStream 메소드
  클라이언트 데이터셋 27-33
LoadLibrary 7-11
LoadPackage 함수 15-4
LoadParamListItems 프로시저 21-14
LoadParamsFromIniFile 메소드 26-4
LoadParamsOnConnect 속성 26-4
Local SQL 24-9, 24-10
  이질적 쿼리 24-9
LocalHost 속성
  클라이언트 소켓 37-6
LocalPort 속성
  클라이언트 소켓 37-6
Locate 메소드 22-10
Lock 메소드 11-7
LockList 메소드 11-7
LockType 속성 25-12
LogChanges 속성 27-5, 27-33
LoginPrompt 속성 21-4
LookupCache 속성 23-9
LookupDataSet 속성 23-9, 23-11
LookupKeyFields 속성 23-9, 23-11
LookupResultField 속성 23-11
LParam 매개변수 51-9
.lpk 파일 43-7
LPK_TOOL.EXE 43-7

M

m_spObjectContext 44-4
m_VclCtl 43-10
MainMenu 컴포넌트 8-30
MainWndProc 메소드 51-3
make 유틸리티
  Linux 14-14
malloc 함수 A-9
Man 페이지 7-27
Mappings 속성 24-49
Margin 속성 9-7
MasterFields 속성 22-34, 26-11
MasterSource 속성 22-34, 26-11
Max 속성
  진행 표시줄 9-15
  트랙 표시줄 9-5
MaxDimensions 속성 20-19
MaxLength 속성 9-2
  데이터 인식 리치 에디트 (rich edit) 컨트롤 19-9
  데이터 인식 메모 컨트롤 19-8
MaxRecords 속성 29-34
MaxRows 속성 33-19
MaxStmtsPerConn 속성 26-3
MaxSummaries 속성 20-19
MaxTitleRows 속성 19-23
MaxValue 속성 23-11
MBCS 4-20
MDAC 17-6
MDI 애플리케이션 7-1 ~ 7-3
  메뉴 병합 8-40 ~ 8-41
  생성 7-2
  활성 메뉴 8-40
Menu 속성 8-40
MergeChangeLog 메소드 27-6, 27-33
MESSAGE_HANDLER 매크로 51-4
MESSAGE_MAP 매크로 51-7
messages.hpp 파일 51-2
Method 속성 33-10
MethodAddress 메소드 13-23
MethodType 속성 33-6, 33-10
Microsoft Server DLL 32-6, 32-7
  생성 33-1, 34-8
  요청 메시지 33-2
Microsoft SQL Server
  드라이버 배포 17-9
Microsoft Transaction Server 7-19
Microsoft Transaction Server
  MTS *참조*
Microsoft Transaction Server *참조*
  MTS
midas.dll 27-1, 29-3
midaslib.dcu 17-6, 29-3
MIDI 파일 10-32
MIDL 38-19
  IDL *참조*
MIME 메시지 32-6
MIME 타입과 상수 10-22
Min 속성
  진행 표시줄 9-15
  트랙 표시줄 9-5
MinSize 속성 9-6
MinValue 속성 23-11
MM 동영상 10-32
Mode 속성 24-48

펜 10-5
Modified 메소드 56-13
Modified 속성 9-3
Modifiers 속성 9-6
ModifyAlias 메소드 24-25
ModifySQL 속성 24-39
Month Calendar 컴포넌트 9-11
Month 속성 55-6
MonthCalendar 컴포넌트 9-11
MouseDown 메소드 51-13, 56-10
MouseMove 메소드 51-13
MouseToCell 메소드 9-15
MouseUp 메소드 51-13
.MOV 파일 10-32
Move 메소드
문자열 리스트 4-18, 4-19
MoveBy 메소드 22-6
MoveCount 속성 24-50
MoveFile 함수 4-9
MoveTo 메소드 10-4, 10-8, 50-3
.MPG 파일 10-32
Msg 매개변수 51-3
MSI 기술 17-3
MTS 7-19, 29-6, 38-10, 38-14, 44-1
COM+와 비교 44-1
in-process 서버 44-2
객체 *참조* 44-25 ~ 44-27
런타임 환경 44-2
요구 사항 44-3
트랜잭션 29-17
트랜잭션 객체 38-14 ~ 38-15
트랜잭션 객체 *참조*
MTS Explorer 44-28
MTS 실행 파일 44-2
MTS 패키지 44-6, 44-27
MultiSelect 속성 9-9
Multitier 페이지(New Items 다이얼로그 박스) 29-2
MyBase 27-32
MyEvent_ID 타입 51-15

N

Name 속성
매개변수 22-50, 22-51
메뉴 항목 5-5, 5-6
필드 23-11
Native ~ ols API 58-2
NDX 인덱스 24-6
.Net
웹서비스 36-1
NetCLX 7-17
정의 14-5
NetFileDir 속성 24-23
Netscape Server DLL
생성 33-1
Neutral 스레드 41-8
New Field 다이얼로그 박스 23-5
Field properties 23-5
조회 정의 23-6
Dataset 23-9
Key Fields 23-9
Lookup Keys 23-9
Result Field 23-9
타입 23-6
필드 정의 23-6, 23-7, 23-8, 23-10
필드 타입 23-6
New Items 다이얼로그 박스 7-24, 7-25, 7-26
New Thread Object 다이얼로그 박스 11-2
New Unit 명령 45-12
New 명령 45-12
NewValue 속성 24-37, 28-11
Next 메소드 22-6
NextRecordSet 메소드 22-53, 26-9
nodefault 키워드 47-7
non-production 인덱스 파일 24-6
NOT NULL UNIQUE 제약 조건 28-12
NOT NULL 제약 조건 28-12
NotifyID 7-28
NSAPI 애플리케이션 32-6
디버깅 32-10
생성 33-1, 34-8
요청 메시지 33-2
NTA(Native ~ ols API) 58-8 ~ 58-11
Null 값
범위 22-31
NULL 매크로 A-7
Null 문자 A-8
Null 종료 루틴 4-23 ~ 4-25
NULL 포인터 A-7
NumericScale 속성 22-44, 22-50, 22-51
NumGlyphs 속성 9-7

O

OAD 31-3 ~ 31-4, 31-12
.obj 파일 15-2, 15-13
패키지 15-13
Object Activation Daemon OAD *참조*
Object Broker 29-25
Object HTML 태그 (<OBJECT>) 33-14
Object Inspector 5-2, 52-7
도움말 52-4
메뉴 선택 8-37
배열 속성 편집 47-2
Object Management Group OMG *참조*
Object Repository 7-24 ~ 7-26
공유 디렉토리 지정 7-25
데이터베이스 컴포넌트 24-16
마법사 58-3
세션 24-17
항목 사용 7-25
항목 추가 7-24
Object Repository 다이얼로그 박스 7-24
Object Repository 마법사 58-3
ObjectBroker 속성 29-23, 29-24, 29-25
ObjectContext 속성
Active Server Object 42-3
예제 44-16
Objects 속성 9-16
문자열 리스트 4-19, 6-14
ObjectView 속성 19-22, 22-36, 23-23
.ocx 파일 17-5
ODBC 드라이버
ADO와 함께 사용 25-1, 25-2
BDE와 함께 사용 24-1, 24-15, 24-16
ODL(Object Description Language) 38-17
OEM 문자 집합 16-2
OEMConvert 속성 9-3
OldValue 속성 24-37, 28-11
OLE
메뉴 병합 8-40
OLE Automation, Automation *참조*
OLE DB 25-1, 25-2

OleFunction 메소드 40-14
OleObject 속성 43-14, 43-15
OleProcedure 메소드 40-14
OlePropertyGet 메소드 40-14
OlePropertyPut 메소드 40-14
OLEView 38-19
OMG 31-1, 31-5
OnAccept 이벤트 37-7, 37-9
  서버 소켓 37-9
OnAction 이벤트 33-8
OnAfterPivot 이벤트 20-9
OnBeforePivot 이벤트 20-9
OnBeginTransComplete 이벤트 21-7, 25-8
OnCalcFields 이벤트 22-22, 23-7, 27-10
OnCellClick 이벤트 19-26
OnChange 이벤트 23-15, 50-6, 54-8, 55-13, 56-12
OnClick 이벤트 9-7, 48-1, 48-2, 48-4
  메뉴 5-5
OnColEnter 이벤트 19-26
OnColExit 이벤트 19-26
OnColumnMoved 이벤트 19-19, 19-26
OnCommitTransComplete 이벤트 21-8, 25-8
OnConnect 이벤트 37-8
OnConnectComplete 이벤트 25-7
OnConstrainedResize 이벤트 8-4
OnDataChange 이벤트 19-4, 56-8, 56-12
OnDataRequest 이벤트 27-31, 28-3, 28-12
OnDblClick 이벤트 19-26, 48-4
OnDecisionDrawCell 이벤트 20-12
OnDecisionExamineCell 이벤트 20-12
OnDeleteError 이벤트 22-20
OnDisconnect 이벤트 25-8, 37-7
OnDragDrop 이벤트 6-2, 19-26, 48-4
OnDragOver 이벤트 6-2, 19-26, 48-4
OnDrawCell 이벤트 9-15
OnDrawColumnCell 이벤트 19-26
OnDrawDataCell 이벤트 19-26
OnDrawItem 이벤트 6-15
OnEditButtonClick 이벤트 19-21, 19-26
OnEditError 이벤트 22-17
OnEndDrag 이벤트 6-3, 19-26, 48-4
OnEndPage 메소드 42-2
OnEnter 이벤트 19-26, 48-5
OnError 이벤트 37-8
OnExecuteComplete 이벤트 25-8
OnExit 이벤트 19-26, 56-14
OnFilterRecord 이벤트 22-13, 22-15
OnGetData 이벤트 28-7
OnGetDataSetProperties 이벤트 28-6
OnGetTableName 이벤트 24-11, 27-21, 28-12
OnGetText 이벤트 23-15
OnGetThread 이벤트 37-9
OnHandleActive 이벤트 37-8
OnHTMLTag 이벤트 29-39, 33-15, 33-16, 33-17
OnIdle 이벤트 핸들러 11-5
OnInfoMessage 이벤트 25-8
OnKeyDown 이벤트 19-26, 48-5, 51-12, 56-10
OnKeyPress 이벤트 19-26, 48-5, 51-12
OnKeyString 이벤트 51-12
OnKeyUp 이벤트 19-26, 48-5, 51-12
OnLayoutChange 이벤트 20-9
OnListening 이벤트 37-9
OnLogin 이벤트 21-5
OnMeasureItem 이벤트 6-14
OnMouseDown 이벤트 10-24, 10-25, 48-4, 51-12, 56-10
  전달된 매개변수 10-24, 10-25
OnMouseMove 이벤트 10-24, 10-26, 48-4, 51-12
  전달된 매개변수 10-24, 10-25
OnMouseUp 이벤트 10-14, 10-24, 10-25, 48-4, 51-12
  전달된 매개변수 10-24, 10-25
OnNewDimensions 이벤트 20-9
OnNewRecord 이벤트 22-18
OnPaint 이벤트 9-17, 10-2
OnPassword 이벤트 24-13, 24-22
OnPopup 이벤트 6-10
OnPostError 이벤트 22-20
OnReceive 이벤트 37-8, 37-10
OnReconcileError 이벤트 14-25, 24-32, 27-20, 27-23
OnRefresh 이벤트 20-7
OnRequestRecords 이벤트 29-35
OnResize 이벤트 10-2
OnRollbackTransComplete 이벤트 21-8, 25-8
OnScroll 이벤트 9-4
OnSend 이벤트 37-8, 37-10
OnSetText 이벤트 23-15
OnStartDrag 이벤트 19-26
OnStartPage 메소드 42-2
OnStartup 이벤트 24-17
OnStateChange 이벤트 19-4, 20-9, 22-3
OnSummaryChange 이벤트 20-9
OnTerminate 이벤트 11-7
OnTitleClick 이벤트 19-26
OnTranslate 이벤트 30-7
OnUpdateData 이벤트 19-4, 28-8, 28-9
OnUpdateError 이벤트 14-25, 24-32, 24-37 ~ 24-38, 27-22, 28-11
OnUpdateRecord 이벤트 24-32, 24-35 ~ 24-37, 24-39, 24-45
OnValidate 이벤트 23-15
OnWillConnect 이벤트 21-5, 25-7
Open ~ ols API ~ ols API *참조*
Open 메소드
  데이터셋 22-4
  서버 소켓 37-7
  세션 24-18
  연결 컴포넌트 21-3
  쿼리 22-47
OPENARRAY 매크로 13-19
OpenDatabase 메소드 24-18, 24-19
OpenSession 메소드 24-28
Options 속성 9-15
  decision grid 20-12
  TSQLClientDataSet 27-16
  데이터 그리드 19-24
  프로바이더 28-5 ~ 28-6
Oracle 드라이버
  배포 17-9

Oracle 테이블 24-12
Oracle8
  테이블 만들 때의 제한 22-38
ORB 31-1, 31-5
  ORB_init 31-7
  초기화 31-3
ORDER BY 절 22-26
Orientation 속성
  데이터 그리드 19-28
  트랙 표시줄 9-5
Origin 속성 10-28, 23-11
osagent 31-2, 31-3
out-of-process 서버 38-6
  ASP 42-7
Overload 속성 24-12
Owner 속성 45-16
OwnerDraw 속성 6-12
owner-draw 컨트롤 4-19, 6-11
  그리기 6-13, 6-15
  리스트 박스 9-10
  선언 6-12
  크기 지정 6-14

P

Package Collection Editor 15-14
package 인수 13-29
PacketRecords 속성 14-25, 24-32, 27-25
Page Dispatcher 34-9
PageSize 속성 9-5
Paint 메소드 50-6, 54-9, 54-10
PaintRequest 메소드 51-13
PaletteChanged 메소드 50-5, 51-14
PanelHeight 속성 19-28
Panels 속성 9-14
PanelWidth 속성 19-28
Paradox 테이블 24-3, 24-5
  DatabaseName 24-3
  네트워크 제어 파일 24-23
  데이터 액세스 24-9
  디렉토리 24-23 ~ 24-24
  레코드 추가 22-19
  로컬 트랜잭션 24-31
  암호 보호 24-21
  암호 사용 24-23
  이름 변경 24-7
  인덱스 검색 22-26
  일괄 이동 24-51
ParamBindMode 속성 24-12
ParamByName 메소드
  내장 프로시저 22-52
  쿼리 22-45
ParamCheck 속성 22-43, 26-11
Parameters 속성 25-19
  TADOCommand 25-19
  TADOQuery 22-43
  TADOStoredProc 22-50
ParamName 속성 29-38
Params 속성
  TDatabase 24-14
  TSQLConnection 26-4
  XML 브로커 29-34
  내장 프로시저 22-50
  쿼리 22-43, 22-45
  클라이언트 데이터셋 27-27
ParamType 속성 22-44, 22-50
ParamValues 속성 22-45
Parent 속성 45-16
ParentColumn 속성 19-23
ParentConnection 속성 29-28
ParentShowHint 속성 9-15
pascalimplementation 인수 13-30
PasteFromClipboard 메소드 6-9
  그래픽 19-9
  데이터 인식 메모 컨트롤 19-9
PathInfo 속성 33-6
.pce 파일 15-13
pdoxusrs.net 24-23
Pen 속성 10-4, 10-5, 50-3
PenPos 속성 10-4, 10-7
penwin.dll 15-11
Perform 메소드 51-9
perror 함수 A-9
PickList 속성 19-20, 19-21
Picture 객체 10-3
Picture 속성 9-17, 10-18
  프레임 8-14
Pie 메소드 10-4
Pixels 속성 10-4, 10-5, 10-9, 50-3
pmCopy 상수 10-29
pmNotXor 상수 10-29
Polygon 메소드 10-5, 10-12
PolyLine 메소드 10-5, 10-10
PopupMenu 속성 6-10
PopupMenu 컴포넌트 8-30
Port 속성 37-7
  TSocketConnection 29-23
Position 속성 9-5, 9-15
Post 메소드 22-20
  Edit 22-18
PostMessage 메소드 51-10
#pragma package 52-19
Precision 속성
  매개변수 22-44, 22-50, 22-51
  필드 23-11
Prepared 속성
  내장 프로시저 22-52
  단방향 데이터셋 26-8
  쿼리 22-47
Preview 탭 34-2
PRIMARY KEY 제약 조건 28-13
Prior 메소드 22-6
Priority 속성 11-3
private 속성 47-5
PrivateDir 속성 24-24
ProblemCount 속성 24-51
ProblemTableName 속성 24-50, 24-51
ProcedureName 속성 22-49
Project Manager 8-3
Project Options 다이얼로그 박스 7-3
Project Updates 다이얼로그 박스 31-9
PROP_PAGE 매크로 43-15
Property Page 마법사 43-13 ~ 43-14
__property 키워드 13-26
PROPERTYPAGE_IMPL 매크로 43-14
Proportional 설정 10-3
protected
  이벤트 48-5
  지시어 48-5
  클래스의 부분 46-6
  키워드 47-3, 48-5
Provider 속성 25-4
ProviderFlags 속성 28-4, 28-10
ProviderName 속성 18-12, 27-24, 28-3, 29-22, 29-34, 30-8
Providing 28-1, 29-3
public
  속성 47-11
  클래스의 부분 46-7
  키워드 48-5
published 47-3
  속성 47-11, 47-12
    예제 54-3

지시어 47-3, 57-4
클래스의 부분 46-7
키워드 48-5
__published 키워드 13-27
putenv 함수 A-9
PVCS Version Manager 2-5

Q

QApplication_postEvent 메소드 51-15
QCustomEvent_create 함수 51-15
QEvent 51-12
QKeyEvent 51-12
QMouseEvent 51-12
QReport 페이지(컴포넌트 팔레트) 5-7
Qt widget
만들기 14-11
Qt 이벤트
메시지 51-15
Query Builder 22-42
Query 속성
업데이트 객체 24-46
QueryInterface 메소드 38-4
집합체(aggregation) 38-9

R

RaiseException 함수 12-12
RC 파일 8-41
RDBMS 18-3, 29-1
RDSConnection 속성 25-16
read 메소드 47-6
ReadBuffer 메소드
TFileStream 4-2
ReadCommitted 21-9
README 설명서 17-15
ReadOnly 속성 9-2, 56-3, 56-9, 56-10
데이터 그리드 19-20, 19-25
데이터 인식 리치 에디트 (rich edit) 컨트롤 19-9
데이터 인식 메모 컨트롤 19-8
데이터 인식 컨트롤 19-5
테이블 22-37
필드 23-12
Real48 타입 13-23
realloc 함수 A-9
ReasonString 속성 33-12
ReceiveBuf 메소드 37-8
Receiveln 메소드 37-8
RecNo 속성
클라이언트 데이터셋 27-2
Reconcile 메소드 14-25, 24-32
RecordCount 속성
TBatchMove 24-50
RecordSet 속성 25-18
Recordset 속성 25-10
RecordsetState 속성 25-10
RecordStatus 속성 25-12, 25-13
Rectangle 메소드 10-5, 10-11, 50-3
Refresh 메소드 19-6, 27-30
RefreshLookupList 속성 23-9
RefreshRecord 메소드 27-30, 28-3
Register 메소드 10-3
Register 프로시저 52-2
RegisterComponents 프로시저 15-6, 52-2
RegisterComponents 함수 45-14
RegisterConversionType 함수 4-27, 4-28
RegisterHelpViewer 7-36
RegisterNonActiveX 프로시저 43-3
RegisterPooled 플래그 29-9
RegisterPropertyEditor 함수 52-11
RegisterTypeLib 함수 38-18
RegisterViewer 함수 7-32
REGSERV32.EXE 17-5
Release 7-29
Release 메소드 38-4
TCriticalSection 11-8
Remote Data Module 마법사 29-13 ~ 29-14
Remote Database Management system 18-3
REMOTEDATAMODULE_IMPL 매크로 29-5
RemoteHost 속성 37-6
RemotePort 속성 37-6
RemoteServer 속성 27-25, 29-22, 29-26, 29-32, 29-34, 30-8
remove 함수 A-8
RemoveAllPasswords 메소드 24-22
RemovePassword 메소드 24-22
rename 함수 A-8
RenameFile 함수 4-9
RepeatableRead 21-9
RequestLive 속성 24-10
RequestRecords 메소드 29-35
Requires 리스트(패키지) 15-6, 15-7, 15-8, 15-9, 52-19
.res 파일 45-15
ResetEvent 메소드 11-10
ResolveToDataSet 속성 28-4
Resolving 28-1
resourcestring 매크로 13-21
RestoreDefaults 메소드 19-21
Result 매개변수 51-7
Resume 메소드 11-11
retaining aborts 25-6
retaining commits 25-6
ReturnValue 속성 11-9
RevertRecord 메소드 14-25, 24-32, 27-6
RFC 문서 32-3
RFC(Request for Comment) 문서 32-3
Rollback 메소드 21-8
RollbackTrans 메소드 21-8
root 디렉토리
Linux 14-15
RoundRect 메소드 10-5, 10-11
RowAttributes 속성 33-19
RowCount 속성 19-12, 19-28
RowHeights 속성 6-14, 9-15
RowRequest 메소드 28-3
Rows 속성 9-16
RowsAffected 속성 22-47
RPC 38-8
RTTI 46-7
C++와 오브젝트 파스칼 비교 13-22
인보커블 인터페이스 36-2

S

safe array 39-12
SafeArray 39-12
SafeRef 메소드 44-26
Samples 페이지(컴포넌트 팔레트) 5-8
Save as Template 명령(메뉴 디자이너) 8-37, 8-39
Save Attributes 명령 23-13
Save Template 다이얼로그 박스 8-39

SaveConfigFile 메소드 24-25
SavePoint 속성 27-6
SaveToFile 메소드 10-20
  ADO 데이터셋 25-14
  그래픽 50-4
  문자열 4-15
  클라이언트 데이터셋 18-9, 27-34
SaveToStream 메소드
  클라이언트 데이터셋 27-34
ScanLine 속성
  비트맵 예제 10-18
ScktSrvr.exe 29-10, 29-13, 29-23
SCM 7-5
Screen 변수 8-2, 16-8
ScriptAlias 지시어 17-11
ScrollBars 속성 6-7, 9-15
  데이터 인식 메모 컨트롤 19-8
SDI 애플리케이션 7-1 ~ 7-3
Sections 속성 9-14
Seek 메소드
  ADO 데이터셋 22-27
Select Menu 다이얼로그 박스 8-37
Select Menu 명령(메뉴 디자이너) 8-37
SELECT 문 22-41
SelectAll 메소드 9-3
SelectCell 메소드 55-14, 56-4
Selection 속성 9-15
SelectKeyword 7-32
SelEnd 속성 9-5
SelLength 속성 6-8, 9-2
SelStart 속성 6-8, 9-2, 9-5
SelText 속성 6-8, 9-2
SendBuf 메소드 37-8
Sender 매개변수 5-5
  예제 10-7
SendIn 메소드 37-8
SendMessage 메소드 51-9
SendStream 메소드 37-8
ServerGUID 속성 29-22
ServerName 속성 29-22
Servers 페이지(컴포넌트 팔레트) 5-8
Service Control Manager 7-5, 7-9
Session 변수 24-3, 24-16
SessionName 속성 24-3, 24-12, 24-27 ~ 24-28, 33-18
Sessions Service 34-9
Sessions 변수 24-17, 24-28
Sessions 속성 24-28
SetAbort 메소드 44-5, 44-9, 44-13
SetBrushStyle 메소드 10-8
SetComplete 메소드 29-17, 44-5, 44-9, 44-13
SetData 메소드 23-16
SetEvent 메소드 11-10
SetFields 메소드 22-21
SetFloatValue 메소드 52-9
SetKey 메소드 22-27
  EditKey와 비교 22-29
SetMethodValue 메소드 52-9
SetOptionalParam 메소드 27-15
SetOrdValue 메소드 52-9
SetPenStyle 메소드 10-7
SetProvider 메소드 27-24
SetRange 메소드 22-31, 22-32
SetRangeEnd 메소드 22-31
  SetRange와 비교 22-31
SetRangeStart 메소드 22-30
  SetRange와 비교 22-31
SetSchemaInfo 메소드 26-12
SetStrValue 메소드 52-9
SetUnhandledExceptionFilter 함수 12-7
SetValue 메소드 52-9
Shape 속성 9-17
Shared Property Manager 44-6 ~ 44-9
  예제 44-7 ~ 44-9
Shift 상태 10-24
ShortCut 속성 8-33
Show 메소드 8-5, 8-6
ShowAccelChar 속성 9-4
ShowButtons 속성 9-11
ShowFocus 속성 19-28
ShowHint 속성 9-15, 19-30
ShowHintChanged 메소드 51-14
ShowLines 속성 9-11
ShowModal 메소드 8-5
ShowRoot 속성 9-11
ShutDown 7-28, 7-29
signal 함수 A-7
Simple Object Access Protocol
  SOAP *참조*
Size 속성
  매개변수 22-44, 22-50, 22-51
  필드 23-12
sizeof 연산자 13-17, A-6
Smart Agent 31-2, 31-3
  위치 찾기 31-3
.so 파일 14-8, 14-13
SOAP 36-1
  데이터 모듈 29-6
  멀티 티어 애플리케이션 29-11
  애플리케이션 마법사 36-11
  애플리케이션 서버에 연결 29-25
  연결 29-11, 29-25
  오류 패킷 36-14
SOAP Data Module 마법사 29-15 ~ 29-16
SoftShutDown 7-28
Sorted 속성 9-9, 19-11
SortFieldNames 속성 26-6
SourceXml 속성 30-6
SourceXmlDocument 속성 30-6
SourceXmlFile 속성 30-6
Spacing 속성 9-7
SparseCols 속성 20-9
SparseRows 속성 20-9
SPX/IPX 24-15
SQL 18-3, 24-8
  로컬 24-9
  명령 실행 21-10 ~ 21-11
  표준 28-12
    Decision Query Editor 20-6
SQL Builder 22-42
SQL Links 17-8, 24-1
  드라이버 24-9, 24-15, 24-30
  드라이버 파일 17-9
  배포 17-9
  사용권 요구 사항 17-15
SQL Monitor 24-53
SQL 문
  decision 데이터셋 20-4, 20-5
  리소스 공유 객체 FILE에 조합된 리소스 생성
    프로바이더 28-10
  매개변수 21-11
  생성
    TSQLDataSet 26-8
    프로바이더 28-4, 28-9
  실행 26-9 ~ 26-10
  업데이트 객체 24-40 ~ 24-43
  클라이언트 제공 27-31, 28-6

통과 SQL 24-30
프로바이더 생성 28-11
SQL 서버
로그인 18-4
SQL 속성 22-42
변경 22-47
SQL 쿼리 22-41 ~ 22-43
결과 집합 22-47
매개변수 22-43 ~ 22-45, 24-41 ~ 24-42
디자인 타임 시 설정 22-44
런타임 시 설정 22-45
마스터/디테일 관계 22-45 ~ 22-46
연결 22-43
복사 22-42
수정 22-42
실행 22-47
업데이트 객체 24-45
준비 22-46 ~ 22-47
최적화 22-48
파일에서 로드 22-42
SQL 클라이언트 데이터셋 27-21 ~ 27-22
SQL 탐색기 24-53, 29-3
속성 집합 정의 23-13
SQLConnection 속성 26-3, 26-17
SQLPASSTHRUMODE 24-30
Standard 페이지(컴포넌트 팔레트) 5-6
StartTransaction 메소드 21-7
State 속성 9-8
그리드 19-15, 19-17
그리드 열 19-15
데이터셋 22-3, 23-8
stateless 객체 44-12
StatusCode 속성 33-11
StatusFilter 속성 14-25, 24-32, 25-12, 27-6, 27-19, 28-8
StdConvs 유닛 4-26, 4-27, 4-28
StepBy 메소드 9-15
StepIt 메소드 9-15
StoredProcName 속성 22-49
StrByteType 4-21
strerror 함수 A-10
Stretch 속성 19-9
StretchDraw 메소드 10-5, 50-3, 50-6
String List Editor
표시 19-10
Strings 속성 4-18
StrNextChar 함수
Linux 14-16
Structured Query Language
SQL *참조*
Style 속성 6-12, 9-10
리스트 박스 9-10
브러시 9-17, 10-8
웹 항목 29-38
콤보 박스 9-10, 19-11
툴 버튼 8-46
펜 10-5
StyleChanged 메소드 51-14
StyleRule 속성 29-38
Styles 속성 29-38
StylesFile 속성 29-38
Subtotals 속성 20-12
SupportCallbacks 속성 29-16
Suspend 메소드 11-11
switch 문
case 값 A-6
Sybase 드라이버
배포 17-9
Synchronize 메소드 11-4
System 페이지(컴포넌트 팔레트) 5-7

T

Table HTML 태그 (<TABLE>) 33-14
TableAttributes 속성 33-19
TableName 속성 22-25, 22-38, 26-7
TableOfContents 7-32
TableType 속성 22-37, 24-5 ~ 24-6
Tabs 속성 9-13
TabStopChanged 메소드 51-14
TAction 8-20
TActionClientItem 8-22
TActionList 8-18, 8-19
TActionMainMenuBar 8-16, 8-18, 8-19, 8-21
TActionManager 8-16, 8-18, 8-19
TActionToolBar 8-16, 8-18, 8-19, 8-21
TActiveForm 43-3, 43-6
TAdapterDispatcher 34-34
TAdapterPageProducer 34-33
TADOCommand 25-2, 25-7, 25-9, 25-16 ~ 25-19
TADOConnection 18-8, 21-1, 25-2, 25-2 ~ 25-8, 25-9
데이터 저장소에 연결 25-3 ~ 25-4
TADODataSet 25-2, 25-9, 25-15 ~ 25-16
TADOQuery 25-2, 25-9
SQL 명령 25-16
TADOStoredProc 25-2, 25-9
TADOTable 25-2, 25-9
Tag 속성 23-12
TApplication 7-27, 7-34
시스템 이벤트 51-12
TApplicationEvents 8-2
TASM 코드
Linux 14-16
TASPObject 42-2
TAutoDriver 40-5, 40-13, 40-14
TBatchMove 24-47 ~ 24-51
오류 처리 24-50 ~ 24-51
TBCDField
디폴트 서식 23-15
TBDEClientDataSet 24-2
TBDEDataSet 22-2
TBevel 9-17
TBitmap 50-3
tbsCheck 상수 8-46
TByteDynArray 36-4
TCalendar 55-1
TcharProperty 타입 52-8
TclassProperty 타입 52-8
TClientDataSet 27-17
TClientDataset 7-23
TClientSocket 37-6
TColorProperty 타입 52-8
TComInterface 40-5, 40-13
TComponent 3-4, 3-7, 45-5
인터페이스 및 13-5
정의 3-5
TComponentClass 45-13
TComponentProperty 타입 52-8
TControl 3-8, 45-4, 48-4, 48-5
정의 3-5
TConvType 값 4-27
TConvTypeInfo 4-30
TCoolBand 9-9
TCoolBar 8-42
TCP/IP 24-15, 37-1

멀티 티어 애플리케이션 29-9 ~ 29-10
서버 37-7
애플리케이션 서버에 연결 29-23
클라이언트 37-6
TCRemoteDataModule 29-13, 29-14
TCurrencyField
디폴트 서식 23-15
TCustomADODataSet 22-2
TCustomClientDataSet 22-2
TCustomContentProducer 33-13
TCustomControl 45-4
TCustomEdit 14-7
TCustomGrid 55-1, 55-3
TCustomIniFile 4-11
TCustomizeDlg 8-22
TCustomListBox 45-3
TDatabase 18-8, 21-1, 24-3, 24-12 ~ 24-16
DatabaseName 속성 24-3
임시 인스턴스 24-19
가져다 놓기 24-20
TDataSet 22-1
자손 22-2
TDataSetProvider 28-1, 28-2
TDataSetTableProducer 33-20
TDataSource 19-3 ~ 19-4
TDateField
디폴트 서식 23-15
TDateTime 타입 55-6
TDateTimeField
디폴트 서식 23-15
TDBChart 18-15
TDBCheckBox 19-2, 19-13
TDBComboBox 19-2, 19-10, 19-10 ~ 19-11
TDBCtrlGrid 19-2, 19-27 ~ 19-28
속성 19-28
TDBEdit 19-2, 19-8
TDBGrid 19-2, 19-15 ~ 19-27
속성 19-20
이벤트 19-26
TDBGridColumns 19-15
TDBImage 19-2, 19-9
TDBListBox 19-2, 19-10, 19-10 ~ 19-11
TDBLookupComboBox 19-2, 19-10, 19-11 ~ 19-12
TDBLookupListBox 19-2, 19-10, 19-11 ~ 19-12
TDBMemo 19-2, 19-8 ~ 19-9
TDBNavigator 19-2, 19-28 ~ 19-30, 22-5, 22-6
TDBRadioGroup 19-2, 19-13 ~ 19-14
TDBRichEdit 19-2, 19-9
TDBText 19-2, 19-8
TDCOMConnection 29-23
TDecisionCube 20-1, 20-4, 20-7 ~ 20-8
이벤트 20-7
TDecisionDrawState 20-12
TDecisionGraph 20-1, 20-2, 20-13 ~ 20-17
TDecisionGrid 20-1, 20-2, 20-10 ~ 20-12
속성 20-12
이벤트 20-12
TDecisionPivot 20-1, 20-2, 20-9 ~ 20-10
속성 20-10
TDecisionQuery 20-1, 20-4, 20-6
TDecisionSource 20-1, 20-8 ~ 20-9
속성 20-9
이벤트 20-9
TDefaultEditor 52-15
TDependency_object 7-9
TDragObject 6-3
TDragObjectEx 6-3
TDrawingTool 10-12
TEdit 9-1
TEnumProperty 타입 52-8
Terminate 메소드 11-6
terminate 함수 12-5
Terminated 속성 11-6
TEvent 11-10
TEventDispatcher 40-14
Text 속성 9-2, 9-3, 9-10, 9-14
TextChanged 메소드 51-14
TextHeight 메소드 10-5, 50-3
TextOut 메소드 10-5, 50-3
TextRect 메소드 10-5, 50-3
TextWidth 메소드 10-5, 50-3
TField 22-1, 23-1 ~ 23-27
메소드 23-16
속성 23-1, 23-10 ~ 23-15
런타임 23-12
이벤트 23-15 ~ 23-16
TFieldDataLink 56-5
TFiler 4-3
TFileStream 4-2, 4-5
파일 I/O 4-5 ~ 4-7
TFloatField
디폴트 서식 23-15
TFloatProperty 타입 52-8
TFMTBcdField
디폴트 서식 23-15
TfontNameProperty 타입 52-8
TFontProperty 타입 52-8
TForm
스크롤 막대 속성 9-5
TFrame 8-13
TGraphic 50-3
TGraphicControl 45-4, 54-3
THandleComponent 51-11
THeaderControl 9-14
this 인수 45-16
Thread Status 박스 11-12
__thread 변경자 11-5
ThreadID 속성 11-12
throw 문 12-1, 12-2, 12-17
THTMLTableAttributes 33-19
THTMLTableColumn 33-20
THTTPRio 36-16
THTTPSoapCppInvoker 36-9, 36-11
THTTPSoapDispatcher 36-9, 36-11
TIBCustomDataSet 22-2
TIBDatabase 18-8, 21-1
TickMarks 속성 9-5
TickStyle 속성 9-5
TIcon 50-3
TiledDraw 메소드 50-6
TImage
프레임 8-14
TImageList 8-45
__TIME__ 매크로 A-6
TIniFile 4-10
TintegerProperty 타입 52-8
TInterfacedObject 13-5
TInvokableClass 36-12
Title 속성
데이터 그리드 19-20
TKeyPressEvent 48-3
TLabel 9-2, 9-4, 45-4
.TLB 파일 38-17, 39-2, 39-19

TLCDNumber 9-2
TLIBIMP 명령줄 도구 38-19, 40-2, 40-6, 41-14
TListBox 45-3
TLocalConnection 27-25, 29-5
TMainMenu 8-19
TMaskEdit 9-1
TMemIniFile 4-10, 14-6
TMemo 9-1
TMemoryStream 4-2
TMessage 51-5, 51-6
TMetafile 50-3
TMethod 타입 51-11
TMethodProperty 타입 52-8
TMTSASPObject 42-2
TMtsDll 44-4, 44-26
TMultiReadExclusiveWriteSynchronizer 11-8
TNestedDataSet 22-36
TNotifyEvent 48-7
TObject 3-4, 13-9, 13-23, 13-28, 46-4
  정의 3-5
ToCommon 4-30
TOleContainer 40-16
  Active Document 38-14
TOleControl 40-6, 40-7
TOleServer 40-6
Tools API 58-1 ~ 58-22
  디버깅 58-10 ~ 58-11
  마법사 58-2, 58-3 ~ 58-7
  모듈 58-3, 58-11 ~ 58-13
  서비스 58-2, 58-7 ~ 58-13
  에디터 58-3, 58-11 ~ 58-13
  작성자 58-2, 58-13 ~ 58-17
  통지자 58-2
  파일 만들기 58-13 ~ 58-17
ToolsAPI 유닛 58-1
Top 속성 8-3, 8-43
TopRow 속성 9-15
TordinalProperty 타입 52-8
TPageControl 9-13
TPageDispatcher 34-35
TPageProducer 33-14
TPaintBox 9-17
TPanel 8-42, 9-12
TPersistent 13-7
  정의 3-5
tpHigher 상수 11-3
tpHighest 상수 11-3
TPicture 타입 50-4
tpIdle 상수 11-3
tpLower 상수 11-3
tpLowest 상수 11-3
tpNormal 상수 11-3
TPopupMenu 8-48
-Tpp 링커 옵션 15-12
TPrinter 4-25
  사용 3-3
TPropertyAttributes 52-10
TPropertyEditor 클래스 52-7
TPropertyPage 43-14
tpTimeCritical 상수 11-3
TQuery 24-2, 24-8 ~ 24-11
  decision 데이터셋 20-5
TQueryTableProducer 33-20
Transactional Data Module 마법사 29-14 ~ 29-15
Transactional Object 마법사 44-18 ~ 44-21
TransformGetData 속성 30-9
TransformRead 속성 30-8
TransformSetParams 속성 30-9
TransformWrite 속성 30-8
TransIsolation 속성 21-9
  로컬 트랜잭션 24-31
Transliterate 속성 23-12, 24-48
Transparent 속성 9-4
TReader 4-3
TRegIniFile 14-6
TRegistry 4-10
TRegistryIniFile 4-10, 4-11
TRegSvr 17-5, 38-19
TRemotable 36-7
TRemoteDataModuleRegistrar 29-9
TRichEdit 9-1
try 문 12-10
try 블록 12-1, 12-2
__try 키워드 12-7
TScrollBox 9-5, 9-13
TSearchRec 4-7
TServerSocket 37-7
TService_object 7-9
TSession 24-16 ~ 24-29
  추가 24-27, 24-28
TSetElementProperty 타입 52-8
TSetProperty 타입 52-8
TSharedConnection 29-28
TSocketConnection 29-23
TSpinEdit 컨트롤 9-5
TSQLClientDataSet 26-2
TSQLConnection 18-8, 21-1, 26-2 ~ 26-5
  메시지 모니터 26-17
  연결 26-3 ~ 26-5
TSQLDataSet 26-2, 26-6, 26-7
TSQLMonitor 26-17
TSQLQuery 26-2, 26-6
TSQLStoredProc 26-2, 26-7
TSQLTable 26-2, 26-7
TSQLTimeStampField
  디폴트 서식 23-15
TStaticText 9-2
TStoredProc 24-2, 24-11 ~ 24-12
TStringList 4-15 ~ 4-19, 7-30
TStringProperty 타입 52-8
TStrings 4-15 ~ 4-19
TTabControl 9-13
TTable 24-2, 24-4 ~ 24-8
  decision 데이터셋 20-5
TTextBrowser 9-2
TTextViewer 9-2
TThread 11-2
TThreadList 11-5, 11-7
TTimeField
  디폴트 서식 23-15
TToolBar 8-19, 8-42, 8-45
TToolButton 8-42
TTreeView 9-11
TUpdateSQL 24-39 ~ 24-47
  프로바이더 24-11
TWebActionItem 33-3
TWebAppDataModule 34-2
TWebAppPageModule 34-2
TWebConnection 29-24
TWebContext 34-34
TWebDataModule 34-2
TWebDispatcher 34-35, 34-39
TWebPageModule 34-2
TWebResponse 33-3
TWidgetControl 14-5
  정의 3-5
TWinControl 3-9, 13-8, 13-11, 14-5, 16-8, 45-4, 48-5
  정의 3-5
TWMMouse 타입 51-7
TWriter 4-3
TWSDLHTMLPublish 36-10, 36-15

TWSDLHTMLPublisher 36-11
TXMLDocument 35-3, 35-8
TXMLTransform 30-6 ~ 30-7
  소스 문서 30-6
TXMLTransformClient 30-8 ~ 30-10
  매개변수 30-9
TXMLTransformProvider 28-1, 28-2, 30-7 ~ 30-8
Type Library Editor 38-17, 39-2 ~ 39-19
  CoClass 39-9
    추가 39-15 ~ 39-16
  COM+ 페이지 44-5, 44-9
  dispinterface 39-9
  Text 페이지 39-8, 39-15
  객체 리스트 창 39-5
  구성 요소 39-3 ~ 39-8
  라이브러리 열기 39-13
  레코드 및 합집합 39-10
    추가 39-17
  메소드
    추가 39-14 ~ 39-15
  모듈 39-10
    추가 39-17
  상태 표시줄 39-5
  속성
    추가 39-14 ~ 39-15
  알리아스 39-10
    추가 39-16
  애플리케이션 서버 29-16
  업데이트 39-18
  연결 어트리뷰트 (attribute) 43-12
  열거 타입 39-9 ~ 39-10
    추가 39-16
  오류 메시지 39-5, 39-8
  요소 39-8 ~ 39-10
    공통 특성 39-8
  요소 선택 39-5
  인터페이스 39-8 ~ 39-9
    수정 39-14 ~ 39-15
  인터페이스 추가 39-14
  타입 정보 저장 및 등록 39-17 ~ 39-19
  타입 정보 페이지 39-6 ~ 39-8
  타입 정의 39-9 ~ 39-10
  툴바 39-3 ~ 39-5
typedef, 오브젝트 파스칼에서 C++로 13-16

U

UCS 표준 14-19
UDP 프로토콜 37-1
Unassociate Attributes 명령 23-14
UndoLastChange 메소드 27-5
unexpected 함수 12-4
UnhandledExceptionFilter 함수 12-7
UniDirectional 속성 22-48
Unlock 메소드 11-7
UnlockList 메소드 11-7
UnRegisterTypeLib 함수 38-18
Update SQL Editor 24-40 ~ 24-41
  Options 페이지 24-40
  SQL 페이지 24-41
UPDATE 문 24-39, 24-43, 28-9
UpdateBatch 메소드 14-25, 25-12, 25-13
UpdateCalendar 메소드 56-4
UpdateMode 속성 28-10
  클라이언트 데이터셋 27-21
UpdateObject 메소드 43-14, 43-15
UpdateObject 속성 24-11, 24-32, 24-39, 24-44
UpdatePropertyPage 메소드 43-14
UpdateRecordTypes 속성 14-25, 24-32, 27-18
UpdateRegistry 메소드 29-8
UpdatesPending 속성 14-25, 24-32
UpdateStatus 속성 14-25, 24-32, 25-12, 27-18, 28-9
UpdateTarget 메소드 8-28
URI와 URL 비교 32-4
URL 32-3
  IP 주소 37-4
  javascript 라이브러리 29-32, 29-33
  SOAP 연결 29-25
  URI와 비교 32-4
  웹 브라우저 32-5
  웹 연결 29-24
  호스트 이름 37-4
URL 속성 29-24, 29-25, 33-9, 36-16
Use CORBA Object 마법사 31-14
User List Service 34-9
uses 절
  데이터 모듈 추가 7-23
uuid 인수 13-2

V

Value 속성
  매개변수 22-44, 22-45, 22-51
  집계 27-13
  필드 23-17
ValueChecked 속성 19-13
Values 속성
  라디오 그룹 19-14
ValueUnchecked 속성 19-13
var 매개변수 13-16
VCL 7-12, 45-1 ~ 45-2
  C++ 랭귀지 지원 13-1 ~ 13-30
  CLX와 비교 14-5 ~ 14-7
  TComponent 분기 3-7
  TControl 분기 3-8
  TObject 분기 3-5
  TPersistent 분기 3-6
  TWinControl 분기 3-9
  개요 3-1 ~ 3-2
  객체 생성 13-9
  메인 스레드 11-4
  예외 처리 12-15
  예외 클래스 12-16
  유닛 14-9 ~ 14-11
VCL 스타일 클래스
  상속 13-2
VCL 애플리케이션
  Linux로 이식 14-2 ~ 14-14
vcl60.bpl 15-1, 15-9, 17-6
  penwin.dll 15-11
VCLCONTROL_IMPL 매크로 43-3, 43-5
VendorLib 속성 26-3
VertScrollBar 9-5
Visible 속성 3-2
  메뉴 8-40
  쿨바(cool bar) 8-48
  툴바 8-48
  필드 23-12
VisibleButtons 속성 19-29, 19-30
VisibleChanged 메소드 51-14
VisibleColCount 속성 9-15
VisibleRowCount 속성 9-15

VisualCLX
정의 14-5
패키지 15-9
VisualSpeller Control 17-5
vtable
dispinterface와 비교 39-9
작성자 클래스 40-5, 40-13
컴포넌트 랩퍼 40-6
vtables 38-4
COM 인터페이스 포인터 38-4
이중 인터페이스 41-12
타입 라이브러리 38-17

W

W3C 35-2
WaitFor 메소드 11-9, 11-10
WantReturns 속성 9-3
WantTabs 속성 9-3
데이터 인식 리치 에디트(rich edit) 컨트롤 19-9
데이터 인식 메모 컨트롤 19-8
.WAV 파일 10-32
wchar_t widechar 14-19
wchar_t 문자 상수 A-3
Web Broker 7-17
Web Broker 서버 애플리케이션 32-1 ~ 32-3, 33-1 ~ 33-21
개요 33-1 ~ 33-4
데이터 포스트 대상 33-10
데이터베이스 액세스 33-17
데이터베이스 연결 관리 33-18
생성 33-1 ~ 33-3
아키텍처 33-3
웹 디스패처 33-4
응답 만들기 33-8
응답 템플릿 33-14
이벤트 처리 33-5, 33-7, 33-8
테이블 쿼리 33-20
템플릿 33-2
파일 보내기 33-13
프로젝트에 추가 33-3
Web Deployment Options 다이얼로그 박스 43-17
Web Service Definition Language WSDL *참조*
Web Services Importer 36-13
WebDispatch 속성 29-35, 36-11
WebPageItems 속성 29-36
WebServices 페이지(New Items 다이얼로그 박스) 29-2
WebServices 페이지(컴포넌트 팔레트) 29-2
WebSnap 32-1 ~ 32-3
로그인 지원 34-25 ~ 34-31
로그인 페이지 34-27 ~ 34-28
로그인 필요 34-29
서버사이드 스크립트 34-31 ~ 34-34, B-1 ~ B-39
서버사이드 스크립트 예제 B-18
액세스 권한 34-29 ~ 34-31
자습서 34-11 ~ 34-23
전역 스크립트 객체 B-13
WebSnap 페이지(컴포넌트 팔레트) 5-7
widechar 14-19
WideString 14-19
widget 3-9
Windows 컨트롤과 비교 14-5
만들기 14-11
WidgetDestroyed 속성 51-13
Width 속성 8-3, 9-14
데이터 그리드 19-20
데이터 그리드 열 19-16
펜 10-5, 10-6
Win 3.1 페이지(컴포넌트 팔레트) 5-8
WIN32 14-17
Win32 예외 처리 12-6
Win32 페이지(컴포넌트 팔레트) 5-7
Win-CGI 애플리케이션 32-5, 32-6, 32-7
.ini 파일 32-7
생성 33-2, 34-8
Windows
API 기능 45-3, 51-2
API 함수 50-1
GDI(Graphics Device Interface) 10-1
메시지 51-3
메시징 51-1 ~ 51-10
이벤트 48-4
일반 대화 상자
실행 57-5
일반적인 다이얼로그 박스 57-1
생성 57-2
장치 컨텍스트 45-7, 50-1
컨트롤, 서브클래싱 45-4
Windows 컨트롤 서브클래싱 45-4
wininet.dll 29-24, 29-25
WM_APP 상수 51-6
WM_KEYDOWN 메시지 56-9
WM_LBUTTONBUTTON 메시지 56-9
WM_MBUTTONDOWN 메시지 56-9
WM_PAINT 메시지 10-2
WM_RBUTTONDOWN 메시지 56-9
WM_SIZE 메시지 55-4
WndProc 메소드 51-3, 51-5
WordWrap 속성 6-7, 9-3
데이터 인식 메모 컨트롤 19-8
WParam 매개변수 51-9
Wrap 속성 8-46
Wrapable 속성 8-46
Write By Reference
COM 인터페이스 속성 39-9
Write 메소드
TFileStream 4-2
write 메소드 47-6
WriteBuffer 메소드
TFileStream 4-2
WSDL 36-2
게시 36-15
임포트 36-3, 36-13 ~ 36-14, 36-16
파일 36-15
WSDL 관리자 36-15
WSDL 퍼블리셔 36-11
WSDLIMP 36-14

X

XDR 파일 35-2
.xfm 파일 14-2
XML 30-1, 35-1
SOAP 36-1
데이터베이스 애플리케이션 30-1 ~ 30-10
매핑 30-2 ~ 30-3
정의 30-4
문서 타입 선언 35-2
처리 명령 35-1
파서 35-2

XML Data Binding 마법사 35-5 ~ 35-9
XML 문서 30-1, 35-1 ~ 35-9
  노드 35-2, 35-4 ~ 35-5
  노드의 속성 35-6
  데이터 패킷으로 변환 30-1 ~ 30-7
  데이터베이스 정보 게시 30-8
  루트 노드 35-3, 35-6, 35-8
  변환 파일 30-1
  어트리뷰트(attribute) 30-5, 35-5
  인터페이스 생성 대상 35-6
  자식 노드 35-5
  컴포넌트 35-3, 35-8
  필드에 노드 매핑 30-2
XML 브로커 29-32, 29-34 ~ 29-36
  HTTP 메시지 29-35
XML 스키마 35-2
XML 파일 25-14
XMLBroker 속성 29-38
XMLDataFile 속성 28-2, 30-7
XMLDataSetField 속성 29-38
XMLMapper 30-1, 30-4 ~ 30-5
XNS(Xerox Network System) 37-1
XSD 파일 35-2

Y

Year 속성 55-6

ㄱ

가상
  메소드 49-3
    속성 47-2
    속성 에디터 52-8
    에디터 52-9
  메소드 테이블 46-9
  클래스 메소드 13-15
  키워드 46-9
  함수 13-12
가속키 8-33
  메뉴에 추가 8-33
값 47-2
  기본값 데이터 19-10
  디폴트 속성 47-7, 47-11 ~
    재정의 53-3, 53-4
  부울 47-2, 47-11, 56-4
  순차 10-12
  테스트 47-7
개발자 지원 1-3
개방형 배열 13-17
  임시 13-18
객체 47-2
  COM 객체 *참조*
  TObject 3-5
  volatile, 액세스 A-5
  드래그 앤 드롭 6-1
  복사 13-6
  분산 31-1
  상속 3-3 ~ 3-5
  생성 13-7 ~ 13-13, 14-11
  소유된 54-6 ~ 54-9
    초기화 54-7
  스크립트 34-33
  임시 50-6
  초기화 10-13
  함수 인수 13-7
객체 맵 38-23
객체 모델 13-1
객체 잠금
  스레드 11-7
  호출 중첩 11-7
객체 지향 프로그래밍 46-1 ~ 46-10
  선언 46-3, 46-10
    메소드 46-10
    클래스 46-4, 46-6, 46-7
객체 *참조* 13-5
객체 컨텍스트 44-4, 44-5
  ASP 42-3
  트랜잭션 44-10
객체 풀링 44-9 ~ 44-10
  사용 불가능 44-9
  원격 데이터 모듈 29-8 ~ 29-9
객체 필드 23-22 ~ 23-27
  타입 23-22
검색 목록(도움말 시스템) 52-5
게시
  속성
    예제 55-3
결과 매개변수 22-49
경계 사각형 10-11
경로(URL) 32-3
경로명
  Linux 14-14
계산된 필드 22-22 ~ 22-23, 23-6
  값 할당 23-7
  정의 23-7 ~ 23-8
  조회 필드 23-9
  클라이언트 데이터셋 27-10 ~ 27-11
계층 구조(클래스) 46-3
고유한 클래스 45-3
공유 객체 파일 .so 파일 *참조*
공유 속성 그룹 44-6
관계형 데이터베이스 18-1
구독자 객체 40-15 ~ 40-16
구분자 표시줄(메뉴) 8-33
구조 A-5
구조적 예외 12-6
규정 준수 A-1
그래픽 45-7, 50-1 ~ 50-7
  HTML에 추가 33-14
  owner-draw 컨트롤 6-11
  객체 타입 10-3
  국제화 16-9
  그리기와 색칠 비교 10-4, 10-22
  독립적인 50-3
  드로잉 툴 50-2, 50-6, 54-6
    변경 54-8
  로드 10-20, 50-4
  메소드 50-3, 50-4
    이미지 복사 50-6
    팔레트 50-5
  문자열과 연결 4-19
  바꾸기 10-21
  복사 10-22
  복잡 50-5
  붙여넣기 10-23
  삭제 10-22
  선 그리기 10-5, 10-10 ~ 10-11, 10-27 ~ 10-29
    이벤트 핸들러 10-26
    펜 너비 변경 10-6
  Rubber banding 예제 10-24 ~ 10-29
  이미지 다시 그리기 50-6
  이미지 변경 10-21
  저장 10-20, 50-4
  컨테이너 50-4
  컨트롤 추가 10-17
  크기 조정 10-21, 19-9, 50-6
  파일 10-19 ~ 10-22
  파일 형식 10-3
  표시 9-17
  프레임 8-14

프로그래밍 개요 10-1 ~ 10-3
함수, 호출 50-1
그래픽 객체
스레드 11-5
그래픽 메소드 50-6
팔레트 50-5
그래픽 박스 19-2
그래픽 컨트롤 45-4, 50-3, 54-1 ~ 54-11
그리기 54-3 ~ 54-5, 54-9 ~ 54-11
만들기 45-4, 54-3
비트맵과 비교 54-3
시스템 리소스 절감 45-4
이벤트 50-6
그룹 박스 9-12
그룹화 레벨 27-9
유지 보수된 집계 27-12
그룹화 컴포넌트 9-12 ~ 9-14
그리기 그리드 9-15
그리기 모드 10-29
그리기 박스 9-17
그리드 9-15 ~ 9-16, 19-2, 55-1, 55-3, 55-6, 55-13
값 얻기 19-16
그리기 19-25 ~ 19-26
기본값 복구 19-21
데이터 인식 19-14, 19-27
데이터 편집 19-5, 19-25
데이터 표시 19-15, 19-16, 19-27
디폴트 상태 19-15
런타임 옵션 19-24 ~ 19-25
사용자 정의 19-16 ~ 19-21
색상 10-6
열 삽입 19-17
열 재정렬 19-18
열 제거 19-16, 19-18
행 추가 22-18
그림 10-17, 50-3 ~ 50-5
로드 10-20
바꾸기 10-21
변경 10-21
저장 10-20
그림 객체 50-4
글꼴 17-14
높이 10-5
기능
이식 불가능 Windows 14-7
기본
속성 값
지정 47-11
기본 단위 4-27, 4-29
기본 클라이언트 44-2
기본 클래스
생성자 13-11
기술 지원 1-3
기업 간 통신 35-1
기존 컨트롤 45-4
기하학적 도형
그리기 54-10
길이가 0인 파일 A-8
끌기 객체 6-3
끌기 커서 6-2
끝점
소켓 연결 37-5

ㄴ

나누기
나머지의 부호 A-4
날짜
Month Calendar 컴포넌트 9-11
국제화 16-9
설정 A-6
입력 9-11
현지 A-10
날짜 필드
서식 23-14
날짜 형식 A-10
내부 객체 38-9
내장 프로시저 18-5, 22-23, 22-48 ~ 22-53
BDE 기반 24-2, 24-11 ~ 24-12
매개변수 연결 24-11
dbExpress 26-7
데이터베이스 지정 22-48
만들기 26-11
매개변수 22-49 ~ 22-52
디자인 타임 22-50 ~ 22-51
런타임 22-51 ~ 22-52
속성 22-50 ~ 22-51
클라이언트 데이터셋 27-28
실행 22-52
열거 21-13
오버로드된 24-12
준비 22-52
넌블로킹 연결 37-10
넌비주얼 컴포넌트 57-3
넌비주얼(nonvisual) 컴포넌트 45-4, 45-12
네임스페이스 45-13
인보커블 인터페이스 36-3
네트워크
데이터베이스에 연결 24-15
통신 레이어 31-1
네트워크 제어 파일 24-23
노트북 칸막이 9-13
논리값 19-2, 19-13
느린 프로세스
스레드 사용 11-1

ㄷ

다각선 10-10
그리기 10-10
다각형 10-12
그리기 10-12
다이얼로그 박스 57-1 ~ 57-8
DLL 예제 7-12
Windows 공통 57-1
생성 57-2
공통 8-15
국제화 16-8, 16-9
멀티페이지 9-13
생성 57-1
속성 에디터 52-9
초기 상태 설정 57-1
다중 문서 인터페이스 7-1 ~ 7-3
다차원 크로스탭 20-3
단독 잠금
테이블 24-6
단방향 데이터셋 26-1 ~ 26-18
데이터 가져오기 26-8
데이터 편집 26-10
메타데이터 가져오기 26-12 ~ 26-16
명령 실행 26-9 ~ 26-10
서버에 연결 26-2
연결 26-5 ~ 26-7
제한 사항 26-1
준비 26-8
타입 26-2
단방향 커서 22-48
단어 정렬 A-5
단위, 변환 4-27
단일 문서 인터페이스 7-1 ~ 7-3

단일 티어 애플리케이션 18-9, 18-12
  파일 기반 18-9
단축키 9-6
  메뉴에 추가 8-33
달력 55-1 ~ 55-14
  날짜 추가 55-6 ~ 55-11
  속성과 이벤트 정의 55-3, 55-7, 55-12
  오늘 날짜 선택 55-11
  이동 55-12 ~ 55-14
  읽기 전용으로 만들기 56-3 ~ 56-5
  크기 조정 55-4
대리자 38-7, 38-8
  이벤트 인터페이스 40-5
  트랜잭션 객체 44-2
대상 데이터셋 정의 24-47
대소문자 구별
  Linux 14-13
  숨기기 A-3
  외부 식별자 A-3
대소문자 구분
  인덱스 27-8
대화 상자
  Windows 공통
    실행 57-5
더블 클릭
  응답 52-17 ~ 52-18
  컴포넌트 52-15
데이터
  그래프 18-15
  기본값 19-10, 23-20
  변경 22-17 ~ 22-22
  보고서 작성 18-15
  분석 18-15, 20-2
  액세스 56-1
  인쇄 18-15
  입력 22-18
  폼 동기화 19-4
  표시 23-17, 23-17
    그리드 19-15, 19-27
    다시 그리기 비활성화 19-6
    현재 값 19-8
  표시 전용 19-8
  형식, 국제화 16-9
데이터 가져오기
  가져오기 27-26
데이터 그리드 19-2, 19-14, 19-15 ~ 19-27
  값 얻기 19-16
  그리기 19-25 ~ 19-26
  기본값 복구 19-21
  데이터 편집 19-5, 19-25
  데이터 표시 19-15, 19-16, 19-27
    ADT 필드 19-22
    배열 필드 19-22
  디폴트 상태 19-15
  런타임 옵션 19-24 ~ 19-25
  사용자 정의 19-16 ~ 19-21
  속성 19-28
  열 삽입 19-17
  열 재정렬 19-18
  열 제거 19-16, 19-18
  이벤트 19-26 ~ 19-27
데이터 동기화
  여러 폼에서 19-4
데이터 멤버 3-2
  메시지 구조 51-5
  이름 지정 48-2
  초기화 13-12
데이터 모듈 18-6
  Web Broker 애플리케이션 33-4
  데이터베이스 컴포넌트 24-16
  생성 7-20
  세션 24-17
  원격과 표준 비교 7-20
  웹 34-2, 34-3, 34-4 ~ 34-5
  웹 애플리케이션 33-2
  편집 7-20
  폼에서 액세스 7-23
데이터 무결성 18-5, 28-12
데이터 브로커 27-25, 29-1
데이터 소스 18-6, 19-3 ~ 19-4
  비활성화 19-4
  이벤트 19-4
  활성화 19-4
데이터 압축
  TSocketConnection 29-24
데이터 액세스
  메커니즘 7-15 ~ 7-16, 18-1 ~ 18-2, 22-2
  컴포넌트 7-15, 18-1
    스레드 11-4
  크로스 플랫폼 17-7, 18-2
데이터 연결 43-11, 56-5 ~ 56-7
  초기화 56-7
데이터 인식 컨트롤 18-15, 19-1 ~ 19-30, 23-17, 56-1
  공통 기능 19-2
  그래픽 표시 19-9
  그리드 19-14
  다시 그리기 비활성화 19-6, 22-8
  데이터 새로 고침 19-6
  데이터 입력 23-14
  데이터 찾아보기 56-1 ~ 56-8
  데이터 편집 56-9 ~ 56-13
  데이터 표시 19-6 ~ 19-7
    그리드 19-15, 19-27
    현재 값 19-8
  데이터셋과 연결 19-3 ~ 19-4
  레코드 삽입 22-18
  리스트 19-2
  변경 내용에 응답 56-7
  생성 56-1 ~ 56-14
  소멸 56-7
  읽기 전용 19-8
  편집 19-5 ~ 19-6, 22-17
  필드 표시 19-7
데이터 입력 검증 23-15
데이터 저장소 25-2
데이터 제약 조건 참조 제약 조건
데이터 컨텍스트
  웹 서비스 애플리케이션 36-8
데이터 타입
  영구적 필드 23-6
데이터 패킷 30-4
  XML 29-29, 29-31, 29-34
    가져오기 29-35
    편집 29-35
  XML 문서로 변환 30-1 ~ 30-7
  XML 문서에 매핑 30-2
  가져오기 27-25 ~ 28-7
  레코드의 고유성 보장 28-4
  복사 27-13 ~ 27-15
  애플리케이션 정의 정보 27-15, 28-6
  업데이트된 레코드 새로 고침 28-6
  읽기 전용 28-5
  클라이언트 편집 제한 28-5
  편집 28-7
  필드 속성 포함 28-6
  필드 제어 28-4
데이터 필드 23-6
  정의 23-6 ~ 23-7

데이터 필터 22-12 ~ 22-16
  런타임 시 설정 22-15
  북마크 사용 25-10 ~ 25-11
  비어 있는 필드 22-14
  연산자 22-14
  정의 22-13 ~ 22-15
  쿼리와 비교 22-12
  클라이언트 데이터셋 27-3 ~ 27-5
    매개변수 사용 27-28
  활성화/비활성화 22-13
데이터 형식 지정
  애플리케이션 국제화 16-9
데이터베이스 7-15, 18-1 ~ 18-5, 56-1
  HTML 응답 생성 33-17 ~ 33-21
  관계형 18-1
  데이터 추가 22-21
  로그인 18-4, 21-4 ~ 21-5
  보안 18-4
  선택 18-3
  승인되지 않은 액세스 21-4
  식별 24-13 ~ 24-15
  알리아스 24-14
  암시적 연결 21-2
  액세스 22-1
  액세스 속성 56-6 ~ 56-7
  연결 21-1 ~ 21-14
  웹 애플리케이션 33-17
  이름 지정 24-14
  타입 18-2
  트랜잭션 18-4 ~ 18-5
  파일 기반 18-3
데이터베이스 관리 시스템 29-1
데이터베이스 드라이버
  BDE 24-1, 24-3, 24-14
  dbExpress 26-3
데이터베이스 서버 7-15, 21-3, 24-15
  설명 21-2
  연결 18-7 ~ 18-9
  제약 조건 23-21, 23-21 ~ 23-22, 28-12
  타입 18-2
데이터베이스 애플리케이션 7-15, 18-1
  XML 30-1 ~ 30-10
  멀티 티어 29-3 ~ 29-4
  배포 17-6
  분산 7-16
  아키텍처 18-5 ~ 18-14, 29-28
  이식 14-22
  파일 기반 18-9 ~ 18-10, 25-14 ~ 25-15, 27-32 ~ 27-34
  확장 18-11
데이터베이스 엔진
  서드파티 17-7
데이터베이스 연결 21-2 ~ 21-5
  끊기 21-3, 21-3 ~ 21-4
  영구적 24-18
  유지 21-3
  제한 29-8
  풀링 44-6
  풀링(pooling) 29-7
데이터베이스 컴포넌트 7-15, 24-3, 24-12 ~ 24-16
  공유 24-16
  데이터베이스 식별 24-13 ~ 24-15
  세션 24-12 ~ 24-13, 24-20 ~ 24-21
  임시 24-19
    가져다 놓기 24-20
  캐싱된 업데이트 적용 24-34
데이터베이스 탐색기 19-2, 19-28 ~ 19-30, 22-5, 22-6
  데이터 삭제 22-20
  도움말 힌트 19-30
  버튼 19-28
  버튼 사용 가능/사용 불가능 19-29
  편집 22-18
데이터셋 18-6, 22-1 ~ 22-53
  ADO 기반 25-8 ~ 25-16
  BDE 기반 24-2 ~ 24-12
  decision 지원 컴포넌트 20-6
  decision 컴포넌트 20-4
  HTML 문서 33-20
  검색 22-10 ~ 22-12
    검색 확장 22-29
    부분 키 22-29
    여러 열 22-11, 22-12
    인덱스 사용 22-11, 22-12, 22-27 ~ 22-29
  내장 프로시저 22-23, 22-48 ~ 22-53
  단방향 26-1 ~ 26-18
  닫기 22-4
    레코드 포스트 22-20
    연결을 끊지 않고 21-11
  데이터 변경 22-17 ~ 22-22
  레코드 삭제 22-19 ~ 22-20
  레코드 추가 22-18 ~ 22-19, 22-21
  레코드 포스트 22-20
  레코드 표시 22-9 ~ 22-10
  레코드 필터링 22-12 ~ 22-16
  모드 22-2 ~ 22-3
  반복 21-12
  범주 22-23 ~ 22-24
  변경 취소 22-20 ~ 22-21
  사용자 정의 22-2
  상태 22-2 ~ 22-3
  생성 22-37 ~ 22-39
  열기 22-4
  인덱싱되지 않음 22-21
  읽기 전용, 업데이트 24-11
  커서 22-4
  쿼리 22-23, 22-40 ~ 22-48
  탐색 19-28, 22-4 ~ 22-8, 22-16
  테이블 22-23, 22-24 ~ 22-40
  편집 22-17 ~ 22-18
  프로바이더 28-2
  필드 22-1
  현재 행 22-4
데이터셋 페이지 프로듀서 33-18
  필드 값 변환 33-19
데이터셋 프로바이더 18-11
데이터셋 필드 23-22, 23-25 ~ 23-26
  영구적 22-36
  표시 19-24
델타 패킷 28-8, 28-9
  XML 29-34, 29-35 ~ 29-36
  업데이트 검열 28-11
  편집 28-8, 28-9
도메인 오류 A-7
도움말 52-4
  문맥에 따른 도움말 9-15
  타입 정보 39-8
  툴팁 9-15
  힌트 9-15
도움말 객체 등록 7-32
도움말 뷰어 7-27
도움말 선택자 7-33, 7-35
도움말 시스템 7-27, 52-4
  객체 등록 7-32

인터페이스 7-27
키워드 52-5
툴 버튼 8-48
파일 52-4
도움말 힌트 19-30
도킹 6-4
도킹 사이트 6-5
도형 9-17, 10-11 ~ 10-12, 10-14
그리기 10-11, 10-14
비트맵 속성으로 채우기 10-9
윤곽 10-5
채우기 10-8
동시 읽기는 허용하고 쓰기는 배타적인 동기화 장치
사용 경고 11-9
동적 배열 타입 36-4
동적 연결 7-11, 7-12, 31-13
CORBA 31-4, 31-15
DII 31-4
동적 열 19-15
속성 19-16
동적 필드 23-2 ~ 23-3
동적 호출 인터페이스 DII *참조*
드라이버 이름 24-14
드래그 앤 도킹 6-4 ~ 6-6
드래그 앤 드롭 6-1 ~ 6-4
DLL 6-4
마우스 포인터 6-4
사용자 정의 6-3
상태 정보 보기 6-3
이벤트 54-3
드로잉 툴 50-2, 50-6, 54-6
기본값으로 할당 8-44
변경 10-13, 54-8
애플리케이션에서 여러 그리기 객체 처리 10-12
테스트 10-12, 10-13
드롭다운 리스트 19-20
드롭다운 메뉴 8-33 ~ 8-34
드릴다운 폼 19-14
등록
Active Server Object 42-7 ~ 42-8
ActiveX 컨트롤 43-15 ~ 43-16
COM 객체 41-16
CORBA 인터페이스 31-12
변환 패밀리 4-27
속성 에디터 52-11 ~ 52-12
컴포넌트 45-13, 45-14
컴포넌트 에디터 52-18
디렉토리
include 파일 A-6
Linux 14-15
디렉토리 서비스 31-3
디버깅
Active Server Object 42-8
ActiveX 컨트롤 43-16
COM 객체 41-8, 41-17
CORBA 31-16 ~ 31-17
dbExpress 애플리케이션 26-17 ~ 26-18
마법사 58-10 ~ 58-11
서비스 애플리케이션 7-9
웹 서버 애플리케이션 32-9 ~ 32-10, 33-2, 34-8
코드 2-4
트랜잭션 객체 44-27
디스패처 33-2, 33-4 ~ 33-5, 34-34, 34-35, 34-40
DLL 기반 애플리케이션 33-3
액션 항목 선택 33-6, 33-7
요청 처리 33-8
자동 디스패칭 객체 33-5
디자인 타임 인터페이스 46-7
디자인 타임 패키지 15-1, 15-5 ~ 15-6
디지털 오디오 테이프 10-32
디폴트
값 19-10
매개변수 13-21
속성 값 47-7
변경 53-3, 53-4
조상 클래스 46-4
프로젝트 옵션 7-3
핸들러
메시지 51-3
오버라이드 48-9
이벤트 48-9
디폴트 서식, 디폴트 23-14

ㄹ

라디오 그룹 9-12
라디오 버튼 9-8, 19-2
그룹화 9-12
데이터 인식 19-13 ~ 19-14
선택 19-14
라이브러리
사용자 정의 컨트롤 45-4
예외 12-18
라이센스
ActiveX 컨트롤 43-5, 43-7
Internet Explorer 43-7
라이센스 키 43-7
라이센스 패키지 파일 43-7
래스터 작업 50-6
랩퍼 45-4, 57-2
초기화 57-3
컴포넌트 랩퍼 *참조*
랭귀지 확장 13-28
런타임 라이브러리 4-1
런타임 인터페이스 46-7
런타임 타입 정보 46-7
런타임 패키지 15-1, 15-3 ~ 15-4
레이블 9-4, 16-9, 19-2, 45-4
열 19-16
레지스터
객체 및 A-5
레지스트리 16-9
레코드
Type Library Editor 39-10, 39-17
가져오기 26-8, 27-25 ~ 27-26
검색 기준 22-10, 22-11
검색 반복 22-29
반복 22-7
복사 24-8, 24-49
삭제 22-19 ~ 22-20, 22-39 ~ 22-40, 24-8, 24-49
새로 고침 19-6, 27-30
업데이트 22-21 ~ 22-22, 24-8, 24-49, 28-8, 29-35 ~ 29-36
XML 문서에서 30-10
델타 패킷 28-8, 28-9
업데이트 검열 28-11
여러 28-6
쿼리 24-11
클라이언트 데이터셋 27-19 ~ 27-24
테이블 식별 28-11
이동 19-28, 22-4 ~ 22-8, 22-16
일괄 작업 24-8, 24-48, 24-49
작업 24-8
정렬 22-25 ~ 22-27
찾기 22-10 ~ 22-12, 22-27 ~ 22-29
추가 22-18, 22-19, 22-19, 22-21, 24-8, 24-48, 24-49
페치
비동기 25-11

포스트 19-6, 22-20
데이터 그리드 19-25
데이터셋을 닫는 경우 22-20
표시 19-27, 22-9 ~ 22-10
필터링 22-12 ~ 22-16
현재 레코드 동기화 22-40
레코드 전송 57-2
로그인
SOAP 연결 29-25
웹 연결 29-24
로그인 다이얼로그 박스 21-4
로그인 스크립트 21-4 ~ 21-5
로그인 이벤트 21-5
로그인 정보
지정 21-4
로그인 지원
WebSnap 34-25 ~ 34-31
로그인 페이지
WebSnap 34-27 ~ 34-28
로그인, 필요 34-29
로드 밸런싱 31-3
로컬 네트워크 31-3
로컬 데이터베이스 18-3
BDE 지원 24-5 ~ 24-7
알리아스 24-24
액세스 24-5
테이블 이름 변경 24-7
로컬 트랜잭션 24-31
로케일 16-1
데이터 형식 16-9
리소스 모듈 16-10
로케일 설정 4-21
루틴
Null 종료 4-23 ~ 4-25
리모터블 클래스 36-4, 36-7 ~ 36-9
기본 36-7
등록 36-5
수명 관리 36-8
예외 36-14 ~ 36-15
예제 36-8 ~ 36-9
리모터블 타입 레지스트리 36-4, 36-15
리바(rebar) 8-42, 8-47
리소스 45-7, 50-1
문자열 16-10
변환 예제 4-30
분리 16-9
시스템, 최적화 45-4
이름 지정 52-4
지역화 16-10, 16-12
캐시로 저장 50-2
리소스 DLL
동적 전환 16-12
마법사 16-10
리소스 디스펜서 44-5
ADO 44-6
BDE 44-6
리소스 모듈 16-9, 16-10
리소스 문자열 13-20
리소스 파일 8-41
로드 8-41
리소스 풀링 44-5 ~ 44-9
리소스를 캐시로 저장 50-2
리스트
문자열 4-14, 4-15 ~ 4-19
삭제 4-14
스레드에서 사용 11-5
액세스 4-14
영구적(persistent) 4-14
재정렬 4-14
추가 4-13
컬렉션 4-15
리스트 박스 9-9, 19-2, 19-11, 55-1
owner-draw 6-11
draw-item 이벤트 6-15
measure-item 이벤트 6-14
데이터 인식 19-10 ~ 19-12
속성 저장
예제 8-8
채우기 19-10
항목 끌기 6-2, 6-3
항목 놓기 6-2
리스트 컨트롤 9-9 ~ 9-11
리치 에디트(rich edit) 컨트롤 9-3
리포지토리
Object Repository *참조*
릴리스 노트 17-15
링커 스위치
패키지 15-12

ㅁ

마법사 7-24
Active Server Object 38-21, 42-2 ~ 42-3
ActiveForm 38-22, 43-6 ~ 43-7
ActiveX Control 38-21
ActiveX Library 38-22
ActiveX 컨트롤 43-4 ~ 43-5
Automation Server 38-21
Automation 객체 41-4 ~ 41-8
COM 38-19 ~ 38-24, 41-1
COM Server 38-21
COM 객체 39-13, 41-2 ~ 41-4, 41-5 ~ 41-8
COM+ Event Object 38-22
COM+ Event 객체 44-23 ~ 44-24
COM+ Event 구독자 객체 44-24
Console Wizard 7-4
CORBA Client 31-13
CORBA Object 31-6, 31-13
CORBA 서버 31-5
IDE 이벤트에 응답 58-17
Property Page 38-22, 43-13 ~ 43-14
Remote Data Module 29-13 ~ 29-14
SOAP Data Module 29-15 ~ 29-16
Tools API 58-2
Transactional Data Module 29-14 ~ 29-15
Transactional Object 38-22
Type Library 38-22, 39-13
Use CORBA Object 31-14
XML Data Binding 35-5 ~ 35-9
디버깅 58-10 ~ 58-11
리소스 DLL 16-10
생성 58-2, 58-3 ~ 58-7
설치 58-6 ~ 58-7, 58-21 ~ 58-22
웹 서비스 36-11 ~ 36-14
컴포넌트 45-9
타입 58-3
트랜잭션 객체 44-18 ~ 44-21
마샬링 38-7
COM 인터페이스 38-9, 41-3, 41-14 ~ 41-15
CORBA 인터페이스 31-2
IDispatch 인터페이스 38-13, 41-14
사용자 정의 41-15
웹 서비스 36-3
트랜잭션 객체 44-3
마스크 23-14

마스터/디테일 관계 19-14, 22-34 ~ 22-36, 22-45 ~ 22-46
단방향 데이터셋 26-11
멀티 티어 애플리케이션 29-18
연결 삭제 28-6
연결 업데이트 28-6
인덱스 22-34
중첩 테이블 22-36, 29-18
참조 무결성 18-5
클라이언트 데이터셋 27-18
마스터/디테일 폼 19-14
예제 22-34 ~ 22-36
마우스 다운(mouse-down) 메시지 56-9
마우스 메시지 56-9
마우스 버튼 10-24
마우스 이동 이벤트 10-26
클릭 10-25
마우스 버튼 놓기 10-25
마우스 이벤트 10-24 ~ 10-26, 54-3
드래그 앤 드롭 6-1 ~ 6-4
매개변수 10-24
상태 정보 10-24
정의 10-24
테스트 10-26
마우스 포인터
드래그 앤 드롭 6-4
매개변수
HTML 태그 33-14
TXMLTransformClient 30-9
XML 브로커에서 29-34
결과 22-49
마우스 이벤트 10-24, 10-25
메시지 51-3, 51-5, 51-6, 51-9
속성 설정 47-6
배열 속성 47-8
연결 모드 24-12
이벤트 핸들러 48-7, 48-9
이중 인터페이스 41-15
입/출력 22-49
입력 22-49
출력 22-49, 27-27
클라이언트 데이터셋 27-26 ~ 27-28
레코드 필터링 27-28
클래스 46-10
매개변수 컬렉션 에디터 22-44, 22-50
매개변수화된 쿼리 22-42, 22-43 ~ 22-45
만들기
디자인 타임 시 22-44
생성
런타임 시 22-45
매크로 13-17, 13-21, 13-28, 51-4
HANDLE_MSG 51-2
확장 A-7
매핑
XML 30-2 ~ 30-3
정의 30-4
매핑되지 않은 타입 13-23
멀티 스레드 애플리케이션 11-1
메시지 보내기 51-9
세션 24-12, 24-28 ~ 24-29
멀티 티어 애플리케이션 18-3, 18-12, 29-1 ~ 29-40
개요 29-3 ~ 29-4
마스터/디테일 관계 29-18
매개변수 27-27
배포 17-9
생성 29-11 ~ 29-27
서버 라이센스 29-3
아키텍처 29-4, 29-5
웹 애플리케이션 29-28 ~ 29-40
만들기 29-30
생성 29-31 ~ 29-40
장점 29-2
컴포넌트 29-2 ~ 29-3
콜백 29-16
크로스 플랫폼 29-11
멀티미디어 애플리케이션 10-29 ~ 10-33
멀티바이트 문자 집합 16-3
멀티바이트 문자 코드 16-3
멀티바이트 문자(MBCS) 14-19, A-2, A-4
크로스 플랫폼 애플리케이션 14-16
멀티페이지 다이얼로그 박스 9-13
멀티프로세싱
스레드 11-1
메뉴 8-29 ~ 8-40
owner-draw 6-11
국제화 16-8, 16-9
단축키 8-33
명령 액세스 8-33
액션 리스트 8-18
이동 8-37
이름 지정 8-31
이미지 추가 8-35
이벤트 처리 5-5 ~ 5-6, 8-39
임포트 8-41
재사용 8-37
정의 8-17
추가 8-30, 8-33 ~ 8-34
템플릿 8-30, 8-37, 8-38
템플릿으로 저장 8-38, 8-39
팝업 6-10
표시 8-35, 8-37
항목 비활성화 6-9
항목 이동 8-35
메뉴 디자이너 5-5, 8-30 ~ 8-33
컨텍스트 메뉴 8-36
메뉴 마법사 58-3
메뉴 컴포넌트 8-30
메뉴 항목 8-31 ~ 8-33
Delete 8-36
구분자 표시줄 8-33
그룹화 8-33
문자에 밑줄 표시 8-33
삭제 8-32
속성 설정 8-36
위치 표시자 8-36
이동 8-34
이름 지정 8-31, 8-39
정의 8-29
중첩 8-33
추가 8-31, 8-40
편집 8-36
메모 컨트롤 6-6, 9-3, 47-8
수정 53-1
메모 필드 19-2, 19-8 ~ 19-9
리치 에디트(rich edit) 19-9
메모리 관리
decision 컴포넌트 20-8, 20-19
폼 8-5
메소드 3-3, 10-15, 45-6, 49-1, 55-12
ActiveX 컨트롤에 추가 43-9 ~ 43-10
protected 49-3
public 49-3
가상 46-9, 49-3
상속 13-2
그래픽 50-3, 50-4, 50-6
팔레트 50-5

그리기 54-9, 54-10
메시지 처리 51-1, 51-3, 51-5
삭제 5-6
상속된 48-6
선언 10-15, 46-10, 49-4
public 49-3
속성 47-5 ~ 47-7, 49-1, 54-4
오버라이드 51-4, 51-5, 55-13
이름 지정 49-2
이벤트 핸들러 48-4, 48-5
오버라이드 48-5
인터페이스에 추가 41-10
재정의 51-7
초기화 47-13
호출 48-5, 49-3, 54-5
메시지 51-1 ~ 51-8, 55-4, A-9
Linux 시스템 통지 *참조*
Linux 시스템 통지 *참조*
Windows 51-1 ~ 51-10
구조 51-5
레코드
타입, 선언 51-6
마우스 56-9
마우스 다운(mouse-down) 및 키 다운(key-down) 56-9
멀티 스레드 애플리케이션 51-9
보내기 51-8 ~ 51-10
사용자 정의 51-6, 51-8
식별자 51-6
정의됨 51-1
키 56-9
트래핑 51-5
메시지 기반 서버
웹 서버 애플리케이션 *참조*
메시지 루프
스레드 11-4
메시지 보내기 51-8 ~ 51-10
메시지 처리 51-4 ~ 51-5
메시지 핸들러 51-1, 51-3, 55-5
디폴트 51-3
만들기 51-6 ~ 51-8
메소드, 재정의 51-7
선언 51-5, 51-6, 51-8
오버라이드 51-4
메시지 헤더(HTTP) 32-3, 32-4
메인 VCL 스레드 11-4
OnTerminate 이벤트 11-7
메인 폼 8-2
메인 함수 A-2
메타데이터 21-12 ~ 21-14
dbExpress 26-12 ~ 26-16
수정 26-10 ~ 26-11
프로바이더에서 얻기 27-26
메타파일 9-17, 10-1, 10-19, 50-3, 50-4
사용 시기 10-3
멤버 함수 3-3
속성 설정 47-6
명령 객체 25-16 ~ 25-19
반복 21-12
명령, 액션 리스트 8-18
명명된 연결 26-4 ~ 26-5
런타임에 로딩 26-4
삭제 26-5
이름 변경 26-5
추가 26-5
modal 폼 8-4
modeless 폼 8-4, 8-6
모듈
Tools API 58-3, 58-11 ~ 58-13
Type Library Editor 39-10, 39-17
웹 타입 34-2
타입 7-20
모바일 컴퓨팅 18-14
모서리가 둥근 사각형 10-11
목록 뷰
owner draw 6-11
무결성 위반 24-50
문서 리터럴 스타일 36-1
문자 47-2
Null A-8
멀티바이트 A-2
새 줄 A-8
소수점 A-6
와이드 A-3
문자 집합 16-2, 16-2 ~ 16-3, A-2, A-3
ANSI 16-2
OEM 16-2
국제적인 정렬 순서 16-9
기본 16-2
매핑 A-3
멀티바이트 16-3
멀티바이트 변환 16-3
상수 A-6
테스트 A-7
확장 A-2
문자 타입 16-3
문자열 4-19, 47-2, 47-8
2바이트 변환 16-3
HTML 템플릿 33-15
Null 종료 4-23 ~ 4-25
그래픽 연결 6-13
루틴
대소문자 구별 4-21
런타임 라이브러리 4-19
멀티바이트 문자 지원 4-21
반환 47-8
번역 16-8, 16-9
사용 4-19
시작 위치 6-8
영구적으로 변경 A-9
와이드 4-20
인쇄 4-25
잘라내기 16-3
정렬 16-9
크기 6-8
문자열 그리드 9-16
문자열 리스트 4-15 ~ 4-19
owner-draw 컨트롤 6-12 ~ 6-13
기간이 긴 문자열 4-16
기간이 짧은 문자열 4-16
문자열 삭제 4-19
문자열 이동 4-18
문자열 찾기 4-18
반복 4-18
복사 4-19
부분 문자열 4-18
생성 4-17
연결 객체 4-19
영구적 4-14
위치 4-18
작성 4-16
정렬 4-18
추가 4-18
파일에 저장 4-15
파일에서 로드 4-15
문자열 번역 16-8, 16-9
문자열 변환
2바이트 변환 16-3
문자열 필드
크기 23-6
문제 테이블 24-50
뮤텍스
CORBA 31-11

미디어 장치 10-31
미디어 플레이어 10-31 ~ 10-33
  예제 10-33

ㅂ

바이너리 스트림
  Null 문자 및 A-8
반올림 규칙 A-4
배경 16-9
배열 47-2, 47-8
  safe 39-12
  개방형 13-17
  반환 타입으로서의 13-23
  속성 13-23
  정수 타입 A-4
    포인터 A-5
  함수 인수로서의 13-23
배열 속성 13-23
배열 필드 23-22, 23-24 ~ 23-25
  단순화 19-22
  영구적 필드 23-25
  표시 19-22, 23-23
배치 업데이트 25-11 ~ 25-14
  적용 25-13 ~ 25-14
  취소 25-14
배치 파일
  Linux 14-13
  실행 A-10
배포
  ActiveX 컨트롤 17-5
  Borland Database Engine 17-8
  CLX 애플리케이션 17-6
  dbExpress 26-1
  DLL 파일 17-5
  IDE 확장 58-6 ~ 58-7, 58-21 ~ 58-22
  MIDAS 애플리케이션 17-9
  글꼴 17-14
  데이터베이스 애플리케이션 17-6
  애플리케이션 17-1
  웹 애플리케이션 17-10
  일반적인 애플리케이션 17-1
  패키지 파일 17-3
백슬래시 문자(\)
  include 파일 A-6
버전 정보
  ActiveX 컨트롤 43-5
  타입 정보 39-8
버전 제어 2-5
버튼 9-6 ~ 9-8
  문자 모양 할당 8-43
  탐색기 19-28
  툴바 8-42
  툴바에 추가 8-43 ~ 8-44, 8-45
  툴바에서 비활성화 8-45
번역 16-1, 16-8
번역 도구 16-1
범위 22-30 ~ 22-33
  Null 값 22-31, 22-32
  경계 22-32
  변경 22-33
  인덱스 22-30
  적용 22-33
  지정 22-30 ~ 22-32
  취소 22-33
  필터와 비교 22-30
변경 로그 27-5, 27-19, 27-33
  변경 내용 저장 27-6
  변경 취소 27-5
변환
  나타나지 않는 값 A-4
  데이터 타입 A-4
    부동 소수점 A-4
    와이드 문자 상수 A-4
    정수 A-4
  반올림 규칙 A-4
  부동 소수점 A-4
  정수 A-4
  포인터에서 정수로 A-4
  필드 값 23-16, 23-18 ~ 23-19
변환 파일 30-1 ~ 30-5
  TXMLTransform 30-6
  TXMLTransformClient 30-9
  TXMLTransformProvider 30-8
  사용자 정의 노드 30-5, 30-7
병렬 프로세스
  스레드 11-1
병합 모듈 17-3
보고서 18-15
보안
  DCOM 29-33
  SOAP 연결 29-25
  데이터베이스 18-4, 21-4 ~ 21-5, 24-21 ~ 24-23
  로컬 테이블 24-21 ~ 24-23
  멀티 티어 애플리케이션 29-2
  소켓 연결 등록 29-10
  웹 연결 29-10, 29-24
  트랜잭션 객체 44-17 ~ 44-18
  트랜잭션 데이터 모듈 29-7, 29-9
복사
  객체 13-6
  비트맵 이미지 50-6
복사 생성자 13-7
부동 소수점 값 A-4
  소수점 문자 A-6
  형식 지정자 A-3
부분 키
  검색 22-29
  범위 설정 22-32
부울 값 47-2, 47-11, 56-4
부울 필드 19-2, 19-13
북마크 22-9 ~ 22-10
  데이터셋 타입별 지원 22-9
  레코드 필터링 25-10 ~ 25-11
분리
  트랜잭션 18-4, 44-10
분산 객체
  CORBA CORBA 객체 *참조*
분산 애플리케이션
  CORBA 31-1 ~ 31-17
  MTS 및 COM+ 7-19
  데이터베이스 7-16
분산된 데이터 처리 29-2
분음 기호 16-9
브러시 10-8 ~ 10-9, 54-6
  변경 54-8
  비트맵 속성 10-9
  색상 10-8
  스타일 10-8
블로킹 연결 37-9, 37-10
  이벤트 처리 37-9
비디오 카세트 10-32
비디오 클립 10-29, 10-31
비즈니스 룰 29-2, 29-13
  ASP 42-1
  트랜잭션 객체 44-2
비트 단위 연산자
  부호 있는 정수 및 A-4
비트 필드 A-5
  단어 경계 늘리기 A-5
  정렬 A-5
  할당 순서 A-5
비트맵 9-17, 10-18 ~ 10-19, 50-3, 50-4
  draw-item 이벤트 6-15
  국제화 16-9

그래픽 컨트롤과 비교 54-3
그리기 10-18
그리기 화면 50-3
로드 50-4
문자열과 연결 4-19, 6-12
바꾸기 10-21
브러시 10-9
브러시 속성 10-8, 10-9
비어 있는 10-18
소멸 10-22
스크롤 10-18
스크롤 추가 10-17
애플리케이션에 표시될 때 10-2
오프스크린 50-5 ~ 50-6
임시 10-17, 10-18
초기 크기 설정 10-18
컴포넌트에 추가 45-14, 52-4
툴바 8-45
프레임 8-14
비트맵 버튼 9-7

ㅅ

사각형
그리기 10-11
사용권 계약서 17-15
사용자 목록 서비스 34-9
사용자 인터페이스 8-1, 18-15 ~ 18-16
단일 레코드 19-7
데이터 구성 19-7, 19-14
레이아웃 8-3 ~ 8-4
분리 18-6
여러 레코드 19-14
폼 8-1 ~ 8-3
사용자 정의 메시지 51-6, 51-8
사용자 정의 컨트롤 45-4
라이브러리 45-4
사용자 정의 컴포넌트 5-8
사용자 정의 타입 54-4
사용자별 구독 40-16
삼각형 10-12
상속
여러 13-2
제한 13-2
상속된
메소드 48-6
속성 54-3, 55-3
게시 47-2
이벤트 48-4
상수
Null 포인터 A-7
문자 A-6
값 A-4
와이드 A-3
순차 값 할당 10-12
와이드 문자 A-3
이름 지정 10-12
상태 정보 9-14
공유 속성 44-6
관리 44-5
마우스 이벤트 10-24
통신 28-7, 28-8, 29-18 ~ 29-20
트랜잭션 객체 44-12
상태 표시줄 9-14
owner draw 6-11
국제화 16-8
새 줄 문자 A-8
색상
국제화 16-9
펜 10-6
색상 그리드 10-6
색상 깊이 17-12
프로그래밍 17-13
생략 부호(...)
그리드의 버튼 19-21
생성자 12-5, 12-15, 45-16, 47-11, 49-3, 55-3, 56-7
C++와 오브젝트 파스칼 비교 13-21
기본 클래스 13-8, 13-13
복사 13-7
선언 45-13
소유된 객체 54-7
여러 8-7
오버라이드 53-3
크로스 플랫폼 애플리케이션 14-12
서버
웹 애플리케이션 디버거 34-8
인터넷 32-1 ~ 32-10
서버 소켓 37-7
소켓 객체 37-7
오류 메시지 37-8
이벤트 처리 37-9
지정 37-6
클라이언트 요청 승인 37-7, 37-9
서버 애플리케이션
COM 38-5 ~ 38-9, 41-1 ~ 41-18
CORBA 31-2, 31-4 ~ 31-12
등록 29-11, 29-21, 31-12
OAD 31-3
멀티 티어 29-5 ~ 29-11, 29-12 ~ 29-17
서비스 37-1
아키텍처 29-5
웹 서비스 36-9 ~ 36-15
인터페이스 37-2
서버 연결 37-2, 37-3
포트 번호 37-5
서버 타입 34-8
서버사이드 스크립트 34-7, 34-31 ~ 34-34, B-1 ~ B-39
JScript 예제 B-18 ~ B-39
객체 타입 B-1 ~ B-13
전역 객체 B-13 ~ B-18
서비스 7-4 ~ 7-9
CORBA 31-1
Name 속성 7-9
Tools API 58-2, 58-7 ~ 58-13
구현 37-1 ~ 37-2, 37-7
네트워크 서버 37-1
디렉토리 31-2, 31-3
설치 7-5
예제 7-7
예제 코드 7-5, 7-7
요청 37-6
제거 7-5
포트 37-2
서비스 스레드 7-7
서비스 시작 이름 7-9
서비스 애플리케이션 7-4 ~ 7-9
디버깅 7-9
예제 7-7
예제 코드 7-5, 7-7
서식있는 텍스트 컨트롤 6-6, 19-9
선
그리기 10-5, 10-10, 10-10 ~ 10-11, 10-27 ~ 10-29
이벤트 핸들러 10-26
펜 너비 변경 10-6
지우기 10-28
선언
메소드 10-15, 46-10, 49-4
public 49-3
메시지 핸들러 51-5, 51-6, 51-8

새 컴포넌트 타입 46-3
속성 47-3, 47-3 ~ 47-7, 47-12, 48-8, 54-4
사용자 정의 타입 54-4
이벤트 핸들러 48-5, 48-8, 55-13
클래스 46-10, 54-6
private 46-4
protected 46-6
public 46-7
published 46-7
인터페이스 13-2
선언자
개수 A-6
중첩 A-6
선택 매개변수 28-6
선택자
도움말 7-33
설정 파일
Linux 14-13
설치
지원 1-3
프로그램 17-2
설치 지원 1-3
세션 24-16 ~ 24-29
다시 시작 24-18
닫기 24-18
데이터베이스 24-12 ~ 24-13
데이터셋 24-3 ~ 24-4
디폴트 24-3, 24-12, 24-16 ~ 24-17
디폴트 연결 속성 24-18
멀티 스레드 애플리케이션 24-12, 24-28 ~ 24-29
메소드 24-13
생성 24-26 ~ 24-27, 24-28
알리아스 관리 24-24
암시적 데이터베이스 연결 24-13
여러 24-12, 24-26, 24-28 ~ 24-29
연결 관리 24-19 ~ 24-21
연결 닫기 24-19
연결 열기 24-19
연결된 데이터베이스 24-20 ~ 24-21
웹 애플리케이션 33-18
이름 지정 24-27 ~ 24-28, 33-18
정보 가져오기 24-26
현재 상태 24-17
활성화 24-17 ~ 24-18
세션 서비스 34-25, 34-26 ~ 34-27
셀(그리드) 9-15
셸 스크립트
Linux 14-13
소멸자 12-5, 49-3, 56-7
VCL 함축 정보 13-13, 13-13 ~ 13-14
소유된 객체 및 54-7
소수점 A-6
소스 데이터셋 정의 24-47
소스 코드
보기
특정 이벤트 핸들러 5-4
재사용
최적화 10-15
편집 2-2
소스 파일 A-6
공유(Linux) 14-12
변경 2-2
패키지 15-2
소유된 객체 54-6 ~ 54-9
초기화 54-7
소유자(holder) 클래스 36-6
소켓 37-1
네트워크 주소 37-3, 37-4
서비스 구현 37-1 ~ 37-2, 37-7
설명 37-3
쓰기 37-10
오류 처리 37-8
이벤트 처리 37-8 ~ 37-9, 37-10
읽기 37-10
읽기/쓰기 37-9
정보 제공 37-4
클라이언트 요청 승인 37-3
호스트 할당 37-4
소켓 객체 37-5
서버 소켓 37-7
클라이언트 37-6
클라이언트 소켓 37-6
소켓 디스패처 애플리케이션 29-10, 29-13, 29-23
소켓 연결 29-9 ~ 29-10, 29-23, 37-2 ~ 37-3
끝점 37-3, 37-5
닫기 37-7
여러 37-5
열기 37-6, 37-7
정보 보내기/받기 37-9
타입 37-2
소켓 컴포넌트 37-5 ~ 37-7
소프트웨어 사용권 요구 사항 17-15
속성 47-1 ~ 47-13
ActiveX 컨트롤에 추가 43-9 ~ 43-10
COM 38-2, 39-8
Write By Reference 39-9
COM 인터페이스 39-9
HTML 테이블 33-19
nodefault 47-7
published가 아닌 저장 및 로드 47-13 ~ 47-15
read 및 write 47-5
값 쓰기 47-6, 52-9
값 읽기 52-9
값 지정 47-11, 52-9
개요 45-6
게시 55-3
기본값 47-7, 47-11
재정의 53-3, 53-4
내부 데이터 저장소 47-4, 47-6
도움말 제공 52-4
랩퍼 컴포넌트 57-4
로드 47-13
배열 47-2, 47-8
변경 52-7 ~ 52-12, 53-4
보기 52-9
상속된 47-2, 54-3, 55-3
선언 47-3, 47-3 ~ 47-7, 47-12, 48-8, 54-4
사용자 정의 타입 54-4
설정 5-2 ~ 5-3
수정 53-3
쓰기 전용 47-6
액세스 47-5 ~ 47-7
업데이트 45-7
이벤트 48-1
인터페이스에 추가 41-9
일반적인 다이얼로그 박스 57-2
읽기 전용 46-7, 47-7, 56-3
재선언 47-11, 48-5
저장 47-12
타입 47-2, 47-8, 52-9, 54-4
편집
텍스트로 52-8
하위 컴포넌트 47-9

속성 설정
  읽기 47-8
  작성 47-8
속성 설정 읽기 47-6
속성 에디터 5-2, 52-7 ~ 52-12
  다이얼로그 박스 52-9
  등록 52-11 ~ 52-12
  어트리뷰트(attribute) 52-10
  파생된 클래스 52-7
속성 페이지 43-13 ~ 43-15
  ActiveX 컨트롤 40-7, 43-3, 43-15
  ActiveX 컨트롤 속성에 연결 43-14
  ActiveX 컨트롤 업데이트 43-15
  만들기 43-13 ~ 43-15
  업데이트 43-14
  임포트된 컨트롤 40-5
  컨트롤 추가 43-14 ~ 43-15
속성, 메모 및 리치 에디트(rich edit) 컨트롤 9-3
수신 대기 연결 37-2, 37-3, 37-7, 37-9
  닫기 37-7
  포트 번호 37-5
수직 트랙 표시줄 9-5
수치 연산 함수
  도메인 오류 A-7
  언더플로 범위 오류 A-7
수평 트랙 표시줄 9-5
순차 값, 상수에 할당 10-12
순환 *참조* 8-3
숫자 47-2
  국제화 16-9
  속성 값 47-11
숫자 보조 프로세서
  부동 소수점 형식 A-3
숫자 필드
  서식 23-14
스래드 클래스
  정의 11-1
스레드 11-1 ~ 11-13
  BDE 24-12
  CORBA 31-11
  ID 11-12
  ISAPI/NSAPI 프로그램 33-2, 33-18
  VCL 스레드 11-4
  객체 잠금 11-7
  그래픽 객체 11-5
  대기 11-9
    동시 11-10
  데이터 액세스 컴포넌트 11-4
  동시 액세스 피하기 11-7
  리스트 사용 11-5
  메시지 루프 및 11-4
  반환 값 11-9
  생성 11-11
  서비스 7-7
  수 제한 11-11
  실행 11-11
  실행 막기 11-7
  예외 11-6
  우선 순위 11-1, 11-2
    오버라이드 11-11
  이름 없는 스레드를 이름이 지정된 스레드로 변환 11-12
  이름 지정 11-12 ~ 11-13
  이벤트 대기 11-10
  임계 구역 11-8
  조정 11-4, 11-7 ~ 11-11
  종료 11-6
  중지 11-11
  초기화 11-2
  프로세스 공간 11-4
  해제 11-2, 11-3
  활동 44-20
스레드 객체 11-1
  정의 11-2
  제한 사항 11-2
  초기화 11-2
스레드 로컬 변수 11-5
  OnTerminate 이벤트 11-7
스레드 모델 41-5 ~ 41-8 41-4
  ActiveX 컨트롤 43-5
  COM 객체 41-3
  시스템 레지스트리 41-6
  원격 데이터 모듈 29-14
  트랜잭션 객체 44-19 ~ 44-20
  트랜잭션 데이터 모듈 29-15
스레드 변수 11-5
  CORBA 31-11
스레드 이름 지정 11-12 ~ 11-13
스레드 함수 11-4
스켈레톤 31-2, 31-2, 31-6
  마샬링 31-2
  위임 31-8
스크롤 막대 9-4
  텍스트 창 6-7
스크롤할 수 있는 비트맵 10-17
스크립트 34-7
  URL 32-3
  WebSnap에서 생성 34-32
  서버사이드 34-31 ~ 34-34
  액티브 34-32
  편집 및 보기 34-33
스크립트 객체 34-33
스크립트 보기 34-33
스크립트 편집 34-33
스키마 정보 26-12 ~ 26-16
  내장 프로시저 26-14, 26-16
  인덱스 26-15 ~ 26-16
  테이블 26-13 ~ 26-14
  필드 26-14 ~ 26-15
스타일
  TApplication 14-6
스타일 시트 29-38
스텁 31-2, 31-6, 31-14 ~ 31-15
  COM 38-8
  마샬링 31-2
  전역 변수 31-15
  트랜잭션 객체 44-2
스트림
  데이터 복사 4-3
  데이터 읽기 및 쓰기 4-2
  위치 4-3
  저장 매체 4-3
  찾기 4-3
  크기 4-3
스플리터 9-6
스피드 버튼 9-7
  가운데 맞춤 8-43
  그룹화 8-44
  드로잉 툴 10-13
  문자 모양 할당 8-43
  이벤트 핸들러 10-13
  작동 모드 8-43
  초기 상태, 설정 8-44
  토글로 사용 8-44
  툴바에 추가 8-43 ~ 8-44
스핀 편집 컨트롤 9-5
시간 A-10
  국제화 16-9
  입력 9-11
시간 변환 4-27
시간 초과 이벤트 11-11

시간 필드
서식 23-14
시간 형식 A-10
시간, clock 함수 및 A-10
시그널
Linux 14-14
시스템 리소스 최적화 45-4
시스템 리소스, 절감 45-4
시스템 이벤트
사용자 정의 51-15
시스템 통지 51-10 ~ 51-15
식별자
GUID *참조*
길이 A-2
대소문자 구별 A-3
데이터 멤버 48-2
리소스 52-4
메소드 49-2
메시지 레코드 타입 51-6
상수 10-12
속성 설정 47-6
외부 A-3
유효 문자 A-2
이벤트 48-8
잘못된 8-31
타입 10-12
신호
응답(CLX) 51-10 ~ 51-12
실수 타입 13-23
실행 파일
COM 서버 38-6
Linux인 경우 14-14
국제화 16-10, 16-12
쓰기 전용 속성 47-6
씬 클라이언트 애플리케이션 29-2, 29-29

O

아날로그 비디오 10-32
아이콘 9-17, 50-3, 50-4
메뉴에 추가 8-21
컴포넌트에 추가 45-14
툴바 8-45
트리 뷰 9-11
아키텍처
BDE 기반 애플리케이션 24-1 ~ 24-2
CORBA 애플리케이션 31-1 ~ 31-4
Web Broker 서버
애플리케이션 33-3
다계층 29-4
데이터베이스
애플리케이션 18-5 ~ 18-14, 24-1 ~ 24-2
서버 29-5
클라이언트 29-4
멀티 티어 29-5
안전 포인터
예외 처리 12-5
안전한 *참조* 44-26
알기 쉬운 툴바 8-47
알리아스
BDE 24-3, 24-14, 24-24 ~ 24-25
삭제 24-25
생성 24-24 ~ 24-25
지역 24-24
지정 24-13, 24-14 ~ 24-15
Type Library Editor 39-10, 39-16
암호
dBASE 테이블 24-21 ~ 24-23
Paradox 테이블 24-21 ~ 24-23
암시적 연결 24-13
암호화, TSocketConnection 29-24
애니메이션 컨트롤 9-17, 10-29 ~ 10-31
예제 10-30
애플리케이션
Apache 32-7, 33-1, 34-8
CGI 독립 실행형 34-8
COM 7-19
CORBA 31-1 ~ 31-17
ISAPI 32-6, 32-7, 33-1, 34-8
Linux로 이식 14-2 ~ 14-19
MDI 7-2
MTS 7-19
NSAPI 32-6, 33-1, 34-8
SDI 7-2
Web Broker 33-1 ~ 33-21
Win-CGI 독립 실행형 34-8
개발 8-1
국제적 16-1
그래픽 45-7, 50-1
데이터베이스 18-1
멀티 스레드 11-1
멀티 티어 29-1 ~ 29-40
개요 29-3 ~ 29-4
배포 17-1
상태 정보 9-14
서비스 7-4
웹 기반 클라이언트 애플리케이션 29-28 ~ 29-40
웹 서버 7-16, 7-17, 34-7
크로스 플랫폼 14-1 ~ 14-26
클라이언트/서버 29-1
네트워크 프로토콜 24-15
파일 17-2
팔레트 실현 50-5
애플리케이션 국제화 16-1
약어 16-9
지역화 16-11
키보드 입력 변환 16-8
애플리케이션 배포
패키지 15-13
애플리케이션 서버 18-13, 29-1, 29-12 ~ 29-17
COM 기반 29-5, 29-16, 29-20
등록 29-11, 29-21
식별 29-22
여러 데이터 모듈 29-20
연결 끊기 29-26
연결 열기 29-26
원격 데이터 모듈 7-23
인터페이스 29-16 ~ 29-17, 29-27
작성 29-13
콜백 29-16
애플리케이션 이식
Linux로 14-2 ~ 14-19
액세스
volatile 객체 A-5
메모리 지역 A-5
합집합 A-4
액세스 권한
WebSnap 34-29 ~ 34-31
액션 8-23 ~ 8-29
대상 8-18
등록 8-28
미리 정의 8-28
실행 8-24
액션 클래스 8-26
업데이트 8-26
정의 8-17, 8-18
클라이언트 8-18

액션 리스트 5-5, 8-17, 8-19, 8-23 ~ 8-48
액션 밴드 8-18
  정의 8-17
액션 에디터
  액션 변경 33-6
  액션 추가 33-5
액션 요청 34-36
액션 응답 34-37
액션 클라이언트, 정의 8-17
액션 항목 33-3, 33-4, 33-6 ~ 33-8
  디스패칭 34-39
  디폴트 33-5, 33-7
  변경시 주의 33-3
  선택 33-6, 33-7
  연결 33-8
  요청에 응답 33-8
  이벤트 핸들러 33-4
  추가 33-5
  페이지 프로듀서 33-15
  활성화 및 비활성화 33-7
액티브 스크립트 34-32
앰퍼샌드(&) 문자 8-33
약한 패키징 15-11
Rubber banding 예제 10-24 ~ 10-29
양방향 애플리케이션
  메소드 16-7
  속성 16-6
양방향 커서 22-48
어댑터 34-2, 34-5 ~ 34-6
어댑터 디스패처 34-8, 34-34, 34-35
어댑터 디스패처 요청 34-36
어댑터 페이지 프로듀서 B-1
어셈블러 코드 14-18
어트리뷰트(attribute)
  속성 에디터 52-10
언더플로 범위 오류
  수치 연산 함수 A-7
업다운(up-down) 컨트롤 9-5
업데이트 객체 24-39 ~ 24-47, 27-18
  SQL 문 24-40 ~ 24-43
  매개변수 24-41 ~ 24-42, 24-45, 24-46 ~ 24-47
  실행 24-44 ~ 24-46
  여러 업데이트 객체 사용 24-43 ~ 24-46
  쿼리 24-46 ~ 24-47
  프로바이더 24-11
업데이트 오류
  응답 메시지 29-36
  해결 27-20, 27-24, 28-8, 28-11
에디터
  Tools API 58-3, 58-11 ~ 58-13
엑스포트된 함수 7-11, 7-12
엔드 유저 어댑터 34-25
여러 줄 텍스트 컨트롤 19-8, 19-9
역참조된 포인터 13-6
역할 기반 보안 44-17
연결 7-11
  DCOM 29-9, 29-23
  HTTP 29-10 ~ 29-11, 29-24
  SOAP 29-11, 29-25
  TCP/IP 29-9 ~ 29-10, 29-23, 37-2 ~ 37-3
  끊기 29-26
  데이터베이스 21-2 ~ 21-5
    관리 24-19 ~ 24-21
    네트워크 프로토콜 24-15
    닫기 24-19
    비동기 25-5
    열기 24-18, 24-19
    영구적 24-18
    이름 지정 26-4 ~ 26-5
    임시 24-20
    제한 29-8
    풀링(pooling) 29-7
  데이터베이스 서버 21-3, 24-15
  열기 29-26, 37-6
  종료 37-7
  클라이언트 37-3
  프로토콜 29-9 ~ 29-11, 29-22
연결 매개변수 24-14 ~ 24-15
  ADO 25-3 ~ 25-4
  dbExpress 26-4, 26-5
  로그인 정보 21-4, 25-4
연결 브로커 29-25
연결 삭제 28-6
연결 업데이트 28-6
연결 이름 26-4 ~ 26-5
  변경 26-5
  삭제 26-5
  정의 26-5
연결 컴포넌트
  DataSnap 18-14, 29-3, 29-4 ~ 29-5, 29-21, 29-22 ~ 29-27
  연결 관리 29-26
  연결 끊기 29-26
  연결 열기 29-26
  프로토콜 29-9 ~ 29-11, 29-22
  데이터베이스 18-7 ~ 18-9, 21-1 ~ 21-14, 24-3
    ADO 25-2 ~ 25-8
    BDE 24-12 ~ 24-16
    dbExpress 26-2 ~ 26-5
    SQL 명령 실행 21-10 ~ 21-11, 25-5
    메타데이터 액세스 21-12 ~ 21-14
    암시적 21-2, 24-3, 24-13, 24-19, 25-3
    연결 24-13 ~ 24-15, 25-2 ~ 25-4, 26-3 ~ 26-5
    연결별 명령문 26-3
    원격 데이터 모듈 29-6
연결 해제된 모델 18-14
연결된 선분 10-10
연결된 속성 21-3
  연결 컴포넌트 21-3
연산자
  비트 단위
    부호 있는 정수 A-4
  할당 13-6
열 9-15
  decision grid 20-11
  HTML 테이블에 포함 33-20
  기본값 19-21
  디폴트 상태 19-15
  삭제 19-16
  속성 19-16, 19-19 ~ 19-20
    재설정 19-21
  영구적 19-15, 19-16 ~ 19-17
    만들기 19-17
    삭제 19-18
    삽입 19-18
    생성 19-21
    재정렬 19-18
열 헤더 9-14, 19-16, 19-20
열거 47-2, A-5
열거 타입 54-4
  Type Library Editor 39-9 ~ 39-10, 39-16
  웹 서비스 36-6
열거형
  상수 10-13

선언 10-12, 10-13
영구 구독 40-16
영구적 열 19-15, 19-16 ~ 19-17
만들기 19-17
삭제 19-16, 19-18
삽입 19-18
생성 19-21
재정렬 19-18
영구적 필드 23-3 ~ 23-16
ADT 필드 23-23
나열 23-4, 23-5
데이터 타입 23-6
데이터 패킷 28-4
데이터셋 필드 22-36
동적으로 변환 23-3
만들기 23-4 ~ 23-5, 23-5 ~ 23-10
배열 필드 23-25
삭제 23-10
속성 23-10 ~ 23-15
이름 지정 23-5
정렬 23-5
정의 23-5 ~ 23-10
테이블 생성 22-38
특수 타입 23-5, 23-6
영구적 필드(persistent field) 19-15
예약어 쓰기 47-8, 54-5
예약어 읽기 47-8, 54-5
예약어 입력 10-12
예외 49-2, 51-3
Linux 14-14
발생 4-2
비트 형식 12-12
생성자 13-13
스레드 11-6
재발생 4-2
정의 12-1
예외 발생 12-12
예외 처리 12-1 ~ 12-18
C++ 구문 12-1
catch 문 12-3
helper 함수 12-7
throw 문 12-2
try 블록 12-2
VCL 12-15
구조적 예외 12-6
구문 12-7
예제 12-11
생성자 및 소멸자 12-5
안전 포인터 12-5
예외 규정 12-4
컴파일러 옵션 12-14
필터 12-8
예외 클래스 12-16
오디오 클립 10-31
오류
도메인 A-7
소켓 37-8
언더플로 범위 A-7
오류 메시지 A-9, A-10
국제화 16-10
오버라이드
가상 메소드 13-11
메소드 51-4, 51-5, 55-13
오버로드된 내장 프로시저 24-12
오브젝트 파스칼
객체 모델 13-1
오프스크린 비트맵 50-5 ~ 50-6
온도 단위 4-29
온라인 도움말 52-4
옵션
동시에 선택할 수 없음 8-44
옵션 매개변수 27-15
와이드 문자 16-3
와이드 문자 상수 A-3
외부 객체 38-9
요약 값
decision cube 20-19
decision 그래프 20-14
유지 보수된 집계 27-13
크로스탭 20-3
요청
디스패칭 34-34
어댑터 34-36
이미지 34-38
요청 객체
헤더 정보 33-4
요청 디스패칭
WebSnap 34-34
요청 메시지 33-2, 33-3, 42-4
HTTP 개요 32-5 ~ 32-6
XML 브로커 29-35
디스패칭 33-5
액션 항목 33-6
응답 33-8, 33-12
처리 33-5
컨텐트 33-11
타입 33-10
헤더 정보 33-9 ~ 33-11
요청 시 가져오기 27-26
요청 헤더 33-9
우선 순위
스레드 사용 11-1, 11-2
운영 체제 환경
문자열, 영구적으로 변경 A-9
원, 그리기 54-10
원격 데이터 모듈 7-23, 29-3, 29-12, 29-13 ~ 29-16
COM 기반 29-5, 29-20
stateless 29-7, 29-9, 29-18 ~ 29-20
구현 객체 29-5
구현 클래스 29-13
부모 29-20
스레드 모델 29-14, 29-15
여러 29-20, 29-28
자식 29-20
풀링(pooling) 29-8 ~ 29-9
원격 데이터베이스 서버 18-3
원격 서버 24-9, 38-6
승인되지 않은 액세스 21-4
연결 관리 24-18
원격 애플리케이션
TCP/IP 37-1
원격 연결 37-2 ~ 37-3
여러 37-5
열기 37-6, 37-7
정보 보내기/받기 37-9
종료 37-7
원자성
트랜잭션 18-4, 44-10
웹 데이터 모듈 34-2, 34-3, 34-4 ~ 34-5
구조 34-5
웹 디스패처
요청 처리 33-3
자동 디스패칭 객체 29-35
웹 모듈 33-2, 33-4, 34-2, 34-2 ~ 34-5
DLL 및 주의 33-3
데이터베이스 세션 추가 33-18
타입 34-2

웹 배포 43-16 ~ 43-18
  멀티 티어 애플리케이션 29-30
웹 브라우저
  URL 32-5
웹 서버 29-30, 32-1 ~ 32-10, 42-6
  디버깅 33-2
  클라이언트 요청 32-5
  타입 34-8
웹 서버 애플리케이션 7-16, 7-17, 32-1 ~ 32-10
  ASP 42-1
  개요 32-6 ~ 32-10
  디버깅 32-9 ~ 32-10
  리소스 위치 32-3
  멀티 티어 29-31 ~ 29-40
  타입 32-6
  표준 32-3
웹 서비스 36-1 ~ 36-17
  구현 클래스 36-12 ~ 36-13
  구현 클래스 등록 36-12
  네임스페이스 36-3
  데이터 컨텍스트 36-8
  마법사 36-11 ~ 36-14
  복잡한 타입 36-3 ~ 36-9
  서버 36-9 ~ 36-15
  서버 작성 36-10 ~ 36-15
  소유자(holder) 클래스 36-6
  스칼라 타입 36-3
  열거 타입 36-6
  예외 36-14 ~ 36-15
  임포트 36-13 ~ 36-14
  추가 36-12 ~ 36-13
  클라이언트 36-16 ~ 36-17
웹 스크립트 34-7
웹 애플리케이션
  ActiveX 38-14, 43-1, 43-16 ~ 43-18
    멀티 티어 클라이언트 29-30
  ASP 38-13, 42-1
  데이터베이스 29-28 ~ 29-40
  배포 17-10
웹 애플리케이션 객체 33-3
웹 애플리케이션 디버거 32-9, 33-2, 34-8
웹 애플리케이션 모듈 34-2, 34-3
웹 연결 29-10 ~ 29-11, 29-24
웹 페이지 32-5
  InternetExpress 페이지 프로듀서 29-36 ~ 29-40
웹 페이지 모듈 34-2, 34-4
웹 페이지 모듈 만들기 34-18
웹 페이지 에디터 29-37
웹 항목 29-37
  속성 29-37 ~ 29-38
위치 독립 코드(PIC) 14-8, 14-18
윈도우
  메시지 처리 55-4
  컨트롤 45-3
  클래스 45-4
  프로시저 51-3
  핸들 45-3, 45-5
유니코드 문자 16-3
유닛
  C++Builder 45-11
  CLX 14-9 ~ 14-11
  VCL 14-9 ~ 14-11
  기존
    컴포넌트 추가 45-12
  컴포넌트 추가 45-12
유로 변환 4-30, 4-33
유저 리스트 서비스 34-25
유지 보수된 집계 18-15, 27-11 ~ 27-13
  값 27-13
  부분합 27-12
  요약 연산자 27-12
  지정 27-11 ~ 27-12
  집계(aggregate) 필드 23-10
윤곽, 그리기 10-5
윤년 55-9
응답
  액션 34-37
  어댑터 34-36
  이미지 34-38
응답 메시지 33-3, 42-5
  데이터베이스 정보 33-17 ~ 33-21
  보내기 33-8, 33-13
  상태 정보 33-11
  생성 33-11 ~ 33-13, 33-13 ~ 33-21
  컨텐트 33-12, 33-13 ~ 33-21
  헤더 정보 33-11 ~ 33-12
응답 템플릿 33-14
응답 헤더 33-12
이름 지정 규칙
  데이터 멤버 48-2
  리소스 52-4
  메소드 49-2
  메시지 레코드 타입 51-6
  속성 47-6
  이벤트 48-8
이미지 9-17, 19-2, 50-3
  국제화 16-9
  그리기 54-9
  다시 그리기 50-6
  다시 생성 10-2
  메뉴에 추가 8-35
  변경 10-21
  복사 50-6
  브러시 10-9
  스크롤 10-18
  저장 10-20
  지우기 10-22
  추가 10-17
  컨트롤 10-2, 10-17
  컨트롤 추가 6-12
  툴 버튼 8-45
  표시 9-17
  프레임 8-14
  화면 떨림 감소 50-5
이미지 다시 그리기 50-6
이미지 요청 34-38
이벤트 5-3 ~ 5-6, 45-6, 48-1 ~ 48-9
  ActiveX 컨트롤 43-10 ~ 43-11
  ADO 연결 25-7 ~ 25-8
  Automation 객체 41-4
  Automation 컨트롤러 40-11, 40-14 ~ 40-16
  COM 41-10, 41-11
  COM 객체 41-3, 41-10 ~ 41-11
  COM+ 40-15 ~ 40-16, 44-21 ~ 44-25
  VCL 컴포넌트 랩퍼 40-3
  XML 브로커 29-35
  공유 5-5
  구현 48-2, 48-4
  그래픽 컨트롤 50-6
  대기 11-10
  데이터 그리드 19-26 ~ 19-27
  데이터 소스 19-4

데이터 인식 컨트롤
활성화 19-7
도움말 제공 52-4
디폴트 5-4
로그인 21-5
마우스 10-24 ~ 10-26
테스트 10-26
메시지 처리 51-4, 51-6
발생 43-11
사용자 3-3
상속된 48-4
새로운 정의 48-6 ~ 48-9
시간 초과 11-11
시스템 3-3
신호 11-10
애플리케이션 레벨 8-2
액세스 48-5
응답 48-5, 48-7, 48-8, 56-8
이름 지정 48-8
인터페이스 41-10
타입 3-3
표준 48-4
필드 객체 23-15 ~ 23-16
핸들러 연결 5-4
이벤트 객체 11-10
이벤트 신호 11-10
이벤트 싱크 41-11
정의 40-14 ~ 40-15
이벤트 핸들러 5-3 ~ 5-6, 45-6, 48-2, 48-8, 56-8
Sender 매개변수 5-5
공유 5-4 ~ 5-6, 10-15
공집합 48-8
디폴트, 오버라이드 48-9
매개변수 48-7, 48-9
통지 이벤트 48-7
메뉴 5-5 ~ 5-6, 6-10
메뉴 템플릿 8-39
메소드 48-4, 48-5
오버라이드 48-5
버튼 클릭에 응답 10-13
삭제 5-6
생성 5-4
선 그리기 10-26
선언 48-5, 48-8, 55-13
이벤트 연결 5-4
정의 5-3
참조에 의한 매개변수 전달 48-9
찾기 5-4
코드 에디터 표시 52-17
타입 48-3, 48-7
이스케이프 시퀀스
소스 파일 A-6
이중 인터페이스 41-12 ~ 41-13
Active Server Object 42-3
매개변수 41-15
메소드 호출 40-13
타입 호환성 41-14
트랜잭션 객체 44-3, 44-19
이질적 쿼리 24-9 ~ 24-10
Local SQL 24-9
인덱스 22-25 ~ 22-36, 47-8
dBASE 테이블 24-6 ~ 24-7
데이터 그룹화 27-9 ~ 27-10
레코드 정렬 22-25 ~ 22-27, 27-7
마스터/디테일 관계 22-34
범위 22-30
부분 키 검색 22-29
삭제 27-9
열거 21-13, 22-26
지정 22-26 ~ 22-27
클라이언트 데이터셋 27-7 ~ 27-10
인덱스 기반 검색 22-11, 22-12, 22-27 ~ 22-29
인덱스 정의 22-38
복사 22-38
인덱스 파일 24-6
인덱싱되지 않은 데이터셋 22-19, 22-21
인보커 36-11
인보커블 인터페이스 36-2 ~ 36-9
구현 36-12 ~ 36-13
네임스페이스 36-3
등록 36-3
호출 36-16 ~ 36-17
인보커블 클래스
생성 36-13
인보케이션 레지스트리 36-3, 36-12
인보커블 클래스 만들기 36-13
인쇄 4-25
인수
fmod 함수 A-7
인스턴스 48-2
인스턴스화 13-5
인스턴싱
COM 서버 41-8
인터넷 서버 32-1 ~ 32-10
인터넷 표준 및 프로토콜 32-3
인터셉터 38-5
인터페이스 7-12, 46-4, 46-6, 57-1, 57-3
ActiveX 38-20
사용자 정의 43-8 ~ 43-13
Automation 41-12 ~ 41-14
COM 7-19, 38-1, 38-3 ~ 38-4, 39-8 ~ 39-9, 40-1, 41-3, 41-9 ~ 41-14
랩퍼 40-5
선언 40-5
이벤트 41-10
COM+ Event 객체 44-24
CORBA 31-2, 31-5 ~ 31-12
dispatch 41-13
DOM 35-2
Tools API 58-1, 58-4 ~ 58-6
버전 번호 58-11
Type Library Editor 39-8 ~ 39-9, 39-14, 41-9
XML 노드 35-4
구현 38-6, 41-3
국제화 16-8, 16-9, 16-12
나가는 41-10, 41-11
넌비주얼(nonvisual) 프로그램 요소 45-4
단일 상속 확장 3-3
도움말 시스템 7-27
동적 연결 31-4, 39-9, 41-12
등록 31-12
디자인 타임 46-7
런타임 46-7
메소드 추가 41-10
복수 상속 13-2
분산 애플리케이션 3-3
사용자 정의 41-14
선언 13-2
속성 선언 57-4
속성 추가 41-9
스켈레톤 및 31-2
스텁 31-2
애플리케이션 서버 29-16 ~ 29-17, 29-27
웹 서비스 36-1
인보커블 36-2 ~ 36-9
타입 라이브러리 38-12, 38-18, 40-5, 41-9

인터페이스 맵 38-23
인터페이스 포인터 38-5
인트라넷
로컬 네트워크 *참조*
호스트 이름 37-4
일관성
트랜잭션 18-4, 44-10
일괄 작업 24-7 ~ 24-8, 24-47 ~ 24-51
다른 데이터베이스 24-49
데이터 업데이트 24-49
데이터 추가 24-48, 24-49
데이터 타입 매핑 24-49 ~ 24-50
데이터셋 복사 24-49
레코드 삭제 24-49
모드 24-8, 24-48
설정 24-47 ~ 24-48
실행 24-50
오류 처리 24-50 ~ 24-51
일대다 관계 22-34, 26-11
일반 47-2
일반 다이얼로그 박스 8-15
일반 대화 상자
실행 57-5
일반적인 다이얼로그 박스 57-1
생성 57-2
일시 구독 40-15
읽기 전용
데이터셋, 업데이트 18-10
속성 46-7, 47-7, 56-3
테이블 22-37
필드 19-5
임계 구역 11-8
사용 경고 11-8, 11-9
임시 객체 50-6
임시 파일 A-9
임포트 라이브러리 7-11, 7-15
임포트된 함수 7-11
입/출력 매개변수 22-49
입력 매개변수 22-49
입력 컨트롤 9-4
입력 포커스 45-4
필드 23-16

ㅈ

자동 디스패칭 컴포넌트 36-11, 36-15
자동 줄 바꿈 6-7
자손 클래스 46-3 ~ 46-4
자습서
WebSnap 34-11 ~ 34-23
작성자 58-2, 58-13 ~ 58-17
작성자 클래스
CoClass 40-5, 40-13
장치 컨텍스트 10-1, 10-2, 45-7, 50-1
장치, 대화형 A-2
장치에 독립적인 그래픽 50-1
재배치 코드 14-18
저장소 클래스 지정자
레지스터 A-5
전역 루틴 4-1
전역 오프셋 테이블(GOT) 14-18
정렬 A-3
구조 멤버 A-5
단어 A-5
비트 필드 A-5
정렬 순서 16-9
TSQLTable 26-7
내림차순 27-8
설정 22-27
클라이언트 데이터셋 27-7
정렬 시퀀스 A-2
정사각형, 그리기 54-10
정수 A-5
나누기 A-4
배열 및 A-4
부호 있는 A-4
열거 A-5
오른쪽으로 시프트된 A-5
포인터 A-5
포인터로 타입 변환 A-4
정수 타입 A-3
정적 연결
COM 38-17
CORBA 31-13
정적 텍스트 컨트롤 9-4
제약 조건
데이터 23-21 ~ 23-22
가져오기 23-21
만들기 23-21
사용 불가능 27-29
임포트 23-22, 27-29, 28-12, 28-13
클라이언트 데이터셋 27-6 ~ 27-7, 27-29
컨트롤 8-3 ~ 8-4
제한 A-1
조상 클래스 46-3 ~ 46-4
디폴트 46-4
조회 값 19-17
조회 리스트 박스 19-2, 19-11 ~ 19-12
보조 데이터 소스 19-12
조회 필드 19-12
조회 메소드 22-11
조회 콤보 박스 19-2, 19-11 ~ 19-12
데이터 그리드 19-20
보조 데이터 소스 19-12
조회 필드 19-12
채우기 19-21
조회 필드 19-12, 23-6
값을 캐시로 저장 23-9
데이터 그리드 19-20
성능 23-9
정의 23-8 ~ 23-9
지정 19-21
프로그램에서 값 제공 23-9
종료 블록 12-13
좌표
현재 그리기 위치 10-25
주석
ANSI 규격 A-2
주소
소켓 연결 37-3, 37-4
중첩 디테일 22-36, 23-25 ~ 23-26, 29-18
요청 시 가져오기 28-5
중첩 테이블 22-36, 23-25 ~ 23-26, 29-18
중첩된 선언자 A-6
증분 검색 19-10
증분 페치 27-25, 29-19
지속성
리소스 디스펜서 44-5
트랜잭션 18-4, 44-10
지시어
#ifdef 14-16
#ifndef 14-17
$LIBPREFIX 컴파일러 7-10
$LIBSUFFIX 컴파일러 7-10
$LIBVERSION 컴파일러 7-10
Linux 14-17
protected 48-5
published 47-3, 57-4
조건부 컴파일 14-16

지역화 16-12
  리소스 16-10, 16-12
  애플리케이션 지역화 16-1
지원 옵션 1-3
직사각형
  그리기 54-10
진단 메시지(ANSI) A-1
진행 표시줄 9-14
집계 필드 27-13
  정의 23-10
집계(aggregate) 필드 23-6
  표시 23-10
집합 47-2
집합체(aggregation)
  COM 38-9
  클라이언트 데이터셋 27-11 ~ 27-13

ㅊ

참조
  C++와 오브젝트 파스칼 비교 13-5
  폼 8-3
참조 무결성 18-5
참조 카운팅
  COM 객체 38-4
참조 필드 23-26 ~ 23-27
  표시 19-24
창 9-6
  크기 조정 9-6
채우기 패턴 10-8
체크 리스트 박스 9-9
체크 박스 9-8
  TDBCheckBox 19-2
  데이터 인식 19-13
초기 연결
  Automation 38-18, 41-12
  COM 38-17
출력 매개변수 22-49, 27-27
측정
  단위 4-28
측정값
  변환 4-26 ~ 4-33
측정값 변환
  리소스 4-30
  변환 요인 4-30
  변환 패밀리 4-26, 4-27
    등록 4-28
    예제 작성 4-27
  복잡한 변환 4-28
  유틸리티 4-26 ~ 4-33
  클래스 4-30 ~ 4-33

ㅋ

캐시된 업데이트
  클라이언트 데이터셋 18-10 ~ 18-14
    트랜잭션 21-6
캐싱된 업데이트 27-15 ~ 27-24
  ADO 25-11 ~ 25-14
    적용 25-13 ~ 25-14
    취소 25-14
  BDE 24-31 ~ 24-47
    오류 처리 24-37 ~ 24-38
    읽기 전용 데이터셋 업데이트 24-11
    적용 24-11, 24-33 ~ 24-37
      여러 테이블 24-39, 24-43
  개요 27-16 ~ 27-17
  마스터/디테일 관계 27-18
  업데이트 객체 27-18
  클라이언트 데이터셋 27-16, 27-19 ~ 27-24
    업데이트 오류 27-24, 28-11
    읽기 전용 데이터셋 업데이트 24-11
    적용 24-11, 27-20 ~ 27-21
      여러 테이블 24-39, 24-43
  프로바이더 28-8
캔버스 45-7, 50-2, 50-3
  개요 10-1 ~ 10-3
  그리기와 색칠 비교 10-4, 10-22
  기본 드로잉 툴 54-6
  도형 추가 10-11 ~ 10-12, 10-14
  선 그리기 10-5, 10-10 ~ 10-11, 10-27 ~ 10-29
    이벤트 핸들러 10-26
    펜 너비 변경 10-6
  이미지 복사 50-6
  일반적인 속성, 메소드 10-4
  팔레트 50-4 ~ 50-5
  화면 새로 고침 10-2
커서 22-4
  단방향 22-48
  동기화 22-40
  복제 27-14
  양방향 22-48
  연결 22-34, 26-11
  이동 22-6, 22-28, 22-29
    마지막 행으로 22-6, 22-7
    조건 사용 22-10
    첫 행으로 22-6, 22-8
컨텍스트 ID 7-30
컨텍스트 메뉴
  메뉴 디자이너 8-36
  툴바 8-48
  항목 추가 52-16 ~ 52-17
컨텍스트 번호(도움말) 9-15
컨텐트 프로듀서 33-4, 33-13
  이벤트 처리 33-15, 33-16, 33-17
컨트롤
  ActiveX 컨트롤 구현 43-3
  ActiveX 컨트롤 생성 43-2, 43-4 ~ 43-7
  owner-draw 6-11, 6-13
    선언 6-12
  그래픽 50-3, 54-1 ~ 54-11
    그리기 54-3 ~ 54-5, 54-9 ~ 54-11
    만들기 45-4, 54-3
    이벤트 50-6
  그룹화 9-12 ~ 9-14
  다시 그리기 54-8, 55-4, 55-5
  데이터 인식 19-1 ~ 19-30
  데이터 찾아보기 56-1 ~ 56-8
  데이터 편집 56-9 ~ 56-13
  데이터 표시 19-4, 23-17
  도형 54-8
  변경 45-3
  사용자 정의 45-4
  윈도우 45-3
  크기 조정 50-6, 55-4
  팔레트 50-4 ~ 50-5
  포커스 받기 45-4
컨트롤 다시 그리기 54-8, 55-4, 55-5
컨트롤 도킹 해제 6-6
컨트롤 크기 조정 9-6, 17-12, 55-4
컴파일러 옵션 7-3
  정렬 A-5
컴파일러 지시어
  Linux 애플리케이션 14-16
  라이브러리 7-10
  패키지 15-11

컴포넌트 45-1, 47-2
고유한 45-3
그룹화 9-12 ~ 9-14
기존 유닛에 추가 45-12
넌비주얼 57-3
넌비주얼(nonvisual) 45-4, 45-12
더블 클릭 52-15, 52-17 ~ 52-18
데이터 인식 56-1
데이터 찾아보기 56-1 ~ 56-8
데이터 편집 56-9 ~ 56-13
등록 45-13, 52-2
만들기 45-8
변경 53-4
비트맵 45-14
사용자 정의 5-8, 45-3, 47-1
생성 45-2
설치 5-8, 15-5 ~ 15-6, 45-18, 52-19
설치 문제 52-19
수정 53-1
온라인 도움말 52-4
유닛에 추가 45-12
이동 45-19
이벤트에 응답 48-5, 48-7, 48-8, 56-8
인터페이스 46-4, 46-6, 57-1
디자인 타임 46-7
런타임 46-7
종속성 45-5
초기화 47-13, 54-7, 56-7
컨텍스트 메뉴 52-15, 52-16 ~ 52-17
컴포넌트 팔레트에 추가 52-1
크기 조정 9-6
테스트 45-16, 45-18, 57-6 ~ 57-8
파생된 클래스 45-3, 45-12, 54-3
팔레트 비트맵 52-4
패키지 15-8, 52-19
표준 5-6 ~ 5-8
컴포넌트 라이브러리
컴포넌트 추가 45-19
컴포넌트 랩퍼 45-4, 57-2
ActiveX 컨트롤 40-4, 40-7, 40-8 ~ 40-10
Automation 객체 40-7 ~ 40-8
예제 40-10 ~ 40-12
COM 객체 40-1, 40-2, 40-3, 40-6 ~ 40-12
초기화 57-3
컴포넌트 마법사 45-9
컴포넌트 사용자 정의 47-1
컴포넌트 에디터 52-15 ~ 52-18
등록 52-18
디폴트 52-15
컴포넌트 인터페이스
생성 57-3
속성 선언 57-4
컴포넌트 템플릿 8-11, 8-12, 46-2
프레임 8-13, 8-14
컴포넌트 팔레트 5-6
ActiveX 페이지 5-8, 40-5
Additional 페이지 5-6
ADO 페이지 5-7, 18-2, 25-1
BDE 페이지 5-7, 18-1
Data Access 페이지 5-7, 18-2, 29-2
Data Controls 페이지 18-15, 19-1, 19-2
DataSnap 페이지 5-7, 29-2, 29-5, 29-6
dbExpress 페이지 5-7, 18-2, 26-2
Decision Cube 페이지 18-15, 20-1
Dialogs 페이지 5-8
FastNet 페이지 5-7
Indy Clients 페이지 5-8
Indy Misc 페이지 5-8
Indy Servers 페이지 5-8
InterBase 페이지 5-7, 18-2
InterBaseAdmin 페이지 5-7
Internet 페이지 5-7
InternetExpress 페이지 5-7
QReport 페이지 5-7
Samples 페이지 5-8
Servers 페이지 5-8
Standard 페이지 5-6
System 페이지 5-7
WebServices 페이지 29-2
WebSnap 페이지 5-7
Win 3.1 페이지 5-8
Win32 페이지 5-7
컴포넌트 설치 45-18
컴포넌트 이동 45-19
컴포넌트 추가 15-6, 52-1, 52-4
페이지 리스트 5-6
페이지 지정 45-14
프레임 8-13
코드 49-3
Linux로 이식 14-15 ~ 14-19
템플릿 7-3
코드 에디터 2-3
개요 2-3
이벤트 핸들러 5-4
표시 52-17
코드 이식 14-15 ~ 14-19
코드 컴파일 2-4
코드 페이지 16-2
코드 편집 2-2, 2-3
콘솔 애플리케이션 7-4
CGI 32-6
VCL 7-4
콜백
멀티 티어 애플리케이션 29-16
제한 29-11
트랜잭션 객체 44-26 ~ 44-27
콤보 박스 9-10, 14-7, 19-2, 19-11
owner-draw 6-11
measure-item 이벤트 6-14
데이터 인식 19-10 ~ 19-12
조회 19-20
쿨바(cool bar) 8-42, 9-8
구성 8-47
디자인 8-42 ~ 8-48
숨기기 8-48
추가 8-47
쿼리 22-23, 22-40 ~ 22-48
BDE 기반 24-2, 24-8 ~ 24-11
동시 24-17
라이브 결과 집합 24-10 ~ 24-11
HTML 테이블 33-20
결과 집합 22-47
단방향 커서 22-48
데이터베이스 지정 22-40
마스터/디테일 관계 22-45 ~ 22-46
매개변수 22-43 ~ 22-45
디자인 타임 시 설정 22-44
런타임 시 설정 22-45
마스터/디테일 관계 22-45 ~ 22-46
속성 22-44 ~ 22-45

이름 지정 22-43
클라이언트 데이터셋 27-28
매개변수연결 연결 22-43
매개변수화 22-42
실행 22-47
양방향 커서 22-48
업데이트 객체 24-46 ~ 24-47
웹 애플리케이션 33-20
이질적 24-9 ~ 24-10
준비 22-46 ~ 22-47
지정 22-41 ~ 22-43, 26-6
최적화 22-46 ~ 22-47, 22-48
필터링과 비교 22-12
쿼리 부분(URL) 32-3
크기 조정 컨트롤
그래픽 50-6
크로스 플랫폼 애플리케이션 14-1 ~ 14-26
Linux로 이식 14-2 ~ 14-19
데이터베이스 14-19 ~ 14-25
만들기 14-1
멀티 티어 29-11
액션 8-19
인터넷 14-25 ~ 14-26
크로스탭 20-2 ~ 20-3, 20-10
1차원 20-3
다차원 20-3
요약 값 20-3
정의 20-2
클라이언트 데이터셋 27-1 ~ 27-34, 29-3
계산된 필드 27-10 ~ 27-11
내부 소스 데이터셋 사용 27-21
다른 데이터셋에 연결 18-10 ~ 18-14, 27-24 ~ 27-31
단방향 데이터셋 사용 26-10
데이터 공유 27-14
데이터 그룹화 27-9 ~ 27-10
데이터 병합 27-14
데이터 복사 27-13 ~ 27-15
데이터 집계 27-11 ~ 27-13
레코드 새로 고침 27-30
레코드 업데이트 27-19 ~ 27-24
레코드 제한 27-28
레코드 필터링 27-2 ~ 27-5
매개변수 27-26 ~ 27-28
배포 17-6
변경 내용 병합 27-33
변경 내용 저장 27-6
변경 취소 27-5
업데이트 오류 해결 27-20, 27-24
업데이트 적용 27-20 ~ 27-21
인덱스 27-7 ~ 27-10
추가 27-8
인덱스 기반 검색 22-27
인덱스 삭제 27-9
인덱스 전환 27-9
제약 조건 27-6 ~ 27-7, 27-29
사용 불가능 27-29
쿼리 제공 27-31 ~ 27-32
타입 27-17 ~ 27-18
탐색 27-2
테이블 만들기 27-32
파일 기반 애플리케이션 27-32 ~ 27-34
파일 로드 27-33
파일 저장 27-34
편집 27-5
프로바이더 27-24 ~ 27-31
프로바이더 지정 27-24 ~ 27-25
클라이언트 소켓 37-3, 37-6 ~ 37-7
서버 식별 37-6
서버에 연결 37-8
서비스 요청 37-6
소켓 객체 37-6
속성 37-6
오류 메시지 37-8
이벤트 처리 37-8
호스트 할당 37-4
클라이언트 애플리케이션
COM 38-2, 38-9, 40-1 ~ 40-16
CORBA 31-2, 31-13 ~ 31-16
네트워크 프로토콜 24-15
만들기 40-1 ~ 40-16
멀티 티어 29-2, 29-4
사용자 인터페이스 29-1
생성 29-21 ~ 29-27
소켓 37-1
씬 29-2, 29-29
아키텍처 29-4
웹 서버 애플리케이션 29-28
웹 서비스 36-16 ~ 36-17
인터페이스 37-2
쿼리 제공 28-6
타입 라이브러리 39-13, 40-2 ~ 40-6
트랜잭션 객체 44-2
클라이언트 연결 37-2, 37-3
열기 37-6
요청 승인 37-7
클라이언트 요청 32-5 ~ 32-6, 33-9
클라이언트 클라이언트 애플리케이션 *참조*
클라이언트/서버 애플리케이션 7-15
클래스 45-2, 45-3, 46-1
private 부분 46-4
protected 부분 46-6
public 부분 46-7
published 부분 46-7
계층 구조 46-3
디폴트 46-4
매개변수로 전달 46-10
새로 파생 46-2
생성 46-1
속성 에디터 52-7
액세스 46-4 ~ 46-8, 54-6
액세스 제한 46-4
오브젝트 파스칼 지원 13-16
인스턴스화 46-2, 48-2
자손 46-3 ~ 46-4
정의 45-12, 46-2
조상 46-3 ~ 46-4
추상 45-3
클래스 팩토리 38-6
ATL 지원 38-23
클래스 포인터 46-10
클래스 필드 54-4
선언 54-6
클래스에서 상속 3-3 ~ 3-5
클로저(closure) 48-2, 48-8
클릭 이벤트 10-25, 48-1, 48-2, 48-7
클립보드 6-8, 6-9, 19-9
그래픽 10-22 ~ 10-23
그래픽 객체 19-9
내용 테스트 6-10
선택 부분 지우기 6-9
이미지 테스트 10-23
형식
추가 52-15, 52-18

키 누름(key-press) 이벤트 48-3, 48-9
키 다운(key-down) 메시지 48-5, 56-9
키 위반 24-51
키 필드 22-32
  여러 22-31, 22-32
키보드 매핑 16-8, 16-9
키보드 이벤트
  국제화 16-8
키워드 52-5
  protected 48-5
키워드 방식 도움말 7-30
키워드 확장 13-23

E

타원
  그리기 10-11, 54-10
타이(tie) 클래스 31-8
  VCL 31-8
타입
  Automation 41-14 ~ 41-15
  Char 16-3
  MIME 10-22
  메시지 레코드 51-6
  사용자 정의 54-4
  속성 47-8, 52-9
  웹 서비스 36-3 ~ 36-9
  이름 지정 10-12
  이벤트 핸들러 48-3
  타입 라이브러리 39-11 ~ 39-13
타입 라이브러리 38-11, 38-12, 38-16 ~ 38-18, 39-1 ~ 39-19
  _TLB 유닛 38-24, 39-2, 39-13, 40-2, 40-5 ~ 40-6, 41-14
  Active Server Object 42-3
  ActiveX 컨트롤 43-3
  IDL 및 ODL 38-17
  IDL로 엑스포트 39-19
  객체 등록 38-18
  내용 38-16, 39-1, 40-5 ~ 40-6
  도구 38-19
  등록 38-19, 39-18
  등록 취소 38-19
  리소스로 포함 39-19, 43-3
  마법사가 생성한 39-1
  만들기 39-13
  배포 39-19
  브라우저 38-18
  사용 시기 38-17
  생성 38-17
  설치 제거 38-18
  성능 최적화 39-9
  액세스 38-17 ~ 38-18, 39-13, 40-2 ~ 40-6
  열기 39-13
  유효한 타입 39-11 ~ 39-13
  이점 38-18
  인터페이스 38-18
  인터페이스 수정 39-14 ~ 39-15
  인터페이스 추가 39-14
  임포트 40-2 ~ 40-6
  저장 39-18
  찾아보기 38-19
  추가
    메소드 39-14 ~ 39-15
    속성 39-14 ~ 39-15
  트랜잭션 객체 44-3
타입 선언
  속성 54-4
  열거형 10-12, 10-13
타입 정보 38-16, 39-1
  dispinterface 41-12
  IDispatch 인터페이스 41-13
  도움말 39-8
  임포트 40-2 ~ 40-6
타입 정의
  Type Library Editor 39-9 ~ 39-10
탐색기 19-2, 19-28 ~ 19-30, 22-5, 22-6
  데이터 삭제 22-20
  데이터셋 간 공유 19-30
  도움말 힌트 19-30
  버튼 19-28
  버튼 사용 가능/사용 불가능 19-29
  편집 22-18
탭
  draw-item 이벤트 6-15
탭 집합 9-13
탭 컨트롤 9-13
  owner-draw 6-11
테스트
  값 47-7
  컴포넌트 45-16, 45-18, 57-6 ~ 57-8
테스트 서버, 웹 애플리케이션 디버거 34-8
테이블 22-23, 22-24 ~ 22-40
  BDE 기반 24-2, 24-4 ~ 24-8
    단독 잠금 24-6
    닫기 24-5
    레코드 복사 24-8
    레코드 삭제 24-8
    레코드 업데이트 24-8
    레코드 추가 24-8
    바인딩 24-5
    액세스 권한 24-6
    인덱스 기반 검색 22-27
    일괄 작업 24-7 ~ 24-8
  dbExpress 26-6 ~ 26-7
  검색 22-27 ~ 22-29
  그리드에서 표시 19-16
  데이터베이스 이외의 그리드 9-15
  데이터베이스 지정 22-24
  동기화 22-40
  레코드 삽입 22-18 ~ 22-19, 22-21
  마스터/디테일 관계 22-34 ~ 22-36
  범위 22-30 ~ 22-33
  비우기 22-39 ~ 22-40
  삭제 22-39
  생성 22-37 ~ 22-39
    영구적 필드 22-38
    인덱스 22-38
  열거 21-12
  인덱스 22-25 ~ 22-36
  읽기 전용 22-37
  정렬 22-25, 26-6, 26-7
  정의 22-37 ~ 22-38
  중첩 22-36
  필드 및 인덱스 정의 22-38
    미리 로드 22-38
테이블 프로듀서 33-19 ~ 33-21
테이블 형식 그리드 19-27
테이블 형식으로 표시(그리드) 9-15
텍스트
  owner-draw 컨트롤 6-11
  검색 9-3
  국제화 16-8
  문자열 4-20
  복사, 잘라내기, 붙여넣기 6-9
  삭제 6-9

선택 6-8, 6-8
오른쪽에서 왼쪽으로 읽기 16-6
인쇄 9-3
작업 6-6 ~ 6-11
컨트롤 6-6
텍스트 스트림 A-8
텍스트 컨트롤 9-1 ~ 9-3
템플릿 7-24, 7-26
decision 그래프 20-16
HTML 33-14 ~ 33-17
Web Broker 애플리케이션 33-2
메뉴 8-30, 8-37, 8-38
삭제 8-38
컴포넌트 8-11, 8-12
페이지 프로듀서 34-4
프로그래밍 7-3
토글 8-44, 8-46
통과 SQL 24-30
통과(passthrough) SQL 24-30, 24-30
통신 37-1
표준 32-3
OMG 31-1
프로토콜 24-15, 32-3, 37-2
UDP와 TCP 비교 31-3
통지 이벤트 48-7
통지자 58-2
Tools API
통지자 58-17 ~ 58-20
작성 58-20
통합 디버거 2-4
통화
국제화 16-9
형식 16-9
투명 배경 16-9
투명하게 보이는 툴바 8-46
툴 버튼 8-45
그룹화/그룹 해제 8-46
도움말 보기 8-48
비활성화 8-45
여러 행 8-46
이미지 추가 8-45
자동 줄 바꿈 8-46
초기 상태, 설정 8-46
토글로 사용 8-46
툴바 8-42, 9-8
owner-draw 6-11
디자인 8-42 ~ 8-48
디폴트 드로잉 툴 8-44
버튼 비활성화 8-45
버튼 삽입 8-43 ~ 8-44, 8-45
생성 8-18
숨기기 8-48
스피드 버튼 9-7
알기 쉬운 8-47
액션 리스트 8-18
여백 설정 8-44
정의 8-17
추가 8-45 ~ 8-46
컨텍스트 메뉴 8-48
투명 8-46
패널 추가 8-43 ~ 8-44
툴팁 도움말 9-15
트랙 표시줄 9-5
트랜잭션 18-4 ~ 18-5, 21-6 ~ 21-9
ADO 25-6 ~ 25-7, 25-8
retaining aborts 25-6
retaining commits 25-6
BDE 24-29 ~ 24-31
암시적 24-29
제어 24-29 ~ 24-30
IAppServer 29-17
MTS 및 COM+ 44-10 ~ 44-17
SQL 명령 사용 21-6, 24-30
객체 컨텍스트 44-10
끝내기 21-7 ~ 21-9
로컬 24-31
로컬 테이블 21-6
롤백 21-8 ~ 21-9
멀티 티어 애플리케이션 29-17 ~ 29-18
분리 18-4, 44-10
레벨 21-9
서버 제어 44-16
시작 21-6 ~ 21-7
어트리뷰트(attribute) 44-11 ~ 44-12
업데이트 적용 21-6, 29-17
여러 객체로 구성 44-10
여러 데이터베이스 사용 44-10
오버랩 21-7
원자성 18-4, 44-10
일관성 18-4, 44-10
자동 44-13
제한 시간 44-17, 44-27
종료 44-12 ~ 44-13
중첩 21-7
커밋 21-8
지속성 18-4, 44-10
캐싱된 업데이트 24-33
커밋 21-8
클라이언트 제어 44-14 ~ 44-15
트랜잭션 객체 44-5, 44-10 ~ 44-17
트랜잭션 데이터 모듈 29-7, 29-15, 29-17 ~ 29-18
트랜잭션 컴포넌트 21-7
트랜잭션 객체 38-10, 38-14 ~ 38-15, 44-1 ~ 44-29
stateless 44-12
객체 컨텍스트 44-4
관리 38-15, 44-28 ~ 44-29
데이터베이스 연결 풀링 44-6
디버깅 44-27
리소스 관리 44-3 ~ 44-10
리소스 릴리스 44-9
마샬링 44-3
보안 44-17 ~ 44-18
비활성화 44-5
생성 44-18 ~ 44-21
설치 44-27 ~ 44-28
속성 공유 44-6 ~ 44-9
요구 사항 44-3
이중 인터페이스 44-3
콜백 44-26 ~ 44-27
타입 라이브러리 44-3
트랜잭션 44-5, 44-10 ~ 44-17
특징 44-2 ~ 44-3
활동 44-20 ~ 44-21
트랜잭션 데이터 모듈 29-6 ~ 29-8
구현 클래스 29-14
데이터베이스 연결 29-6, 29-8
풀링(pooling) 29-7
보안 29-9
스레드 모델 29-15
인터페이스 29-17
트랜잭션 어트리뷰트 29-15
트랜잭션 매개변수
분리 레벨 21-9
트랜잭션 분리 레벨 21-9
로컬 트랜잭션 24-31
지정 21-9
트랜잭션 어트리뷰트(attribute)
설정 44-11
트랜잭션 데이터 모듈 29-15

트리 뷰 9-11
owner-draw 6-11
트리거 18-5
티어 29-1

ㅍ

파일 4-4 ~ 4-12
date-time 루틴 4-9
그래픽 10-19 ~ 10-22, 50-4
길이가 0인 A-8
루틴
date-time 루틴 4-9
Windows API 4-5
런타임 라이브러리 4-7
리소스 8-41
모드 4-6
버퍼링 A-8
복사 4-9
사용 4-4 ~ 4-12
삭제 4-7
쓰기 도중 잘라내기 A-8
열기
abort 함수 및 A-9
remove 함수 A-8
여러 번 A-8
웹으로 보내기 33-13
위치 4-4
이름 변경 4-9, A-8
임시 A-9
찾기 4-4
처리 4-7 ~ 4-9
추가 A-8
크기 4-4
폼 14-2
핸들 4-7
파일 검색 A-6
파일 권한
Linux 14-14
파일 기반 애플리케이션 18-9 ~ 18-10
클라이언트 데이터셋 27-32 ~ 27-34
파일 리스트
항목 끌기 6-2, 6-3
항목 놓기 6-2
파일 버퍼링 A-8
파일 스트림 4-5 ~ 4-7
끝 표시 4-4
생성 4-6
열기 4-6
예외 4-2
이식 가능 4-5
크기 변경 4-4
파일 I/O 4-5 ~ 4-7
핸들 얻기 4-9
파일 위치 지시자 A-8
파일 이름
검색 A-6
변경 A-8
파일 이름 변경 A-8
팔레트 50-4 ~ 50-5
디폴트 동작 50-5
지정 50-5
팔레트 비트맵 파일 52-4
팔레트 실현 50-5
팝업 메뉴 6-10 ~ 6-11
드롭다운 메뉴 8-33
표시 8-35
패널
3D 9-17
스피드 버튼 9-7
스피드 버튼 추가 8-43
폼 위에 추가 8-43
패키지 15-1 ~ 15-15, 52-19
Contains 리스트 15-6, 15-7, 15-8, 15-10, 52-19
DLL 15-1, 15-2, 15-11
Requires 리스트 15-6, 15-7, 15-8, 15-9, 52-19
국제화 16-10, 16-12
누락 52-19
다른 개발자에게 배포 15-13
동적으로 로딩 15-4
디자인 전용 옵션 15-7
디자인 타임 15-1, 15-5 ~ 15-6
디폴트 설정 15-7
런타임 15-1, 15-3 ~ 15-4, 15-7
링커 스위치 15-12
사용 7-10
사용자 정의 15-4
생성 7-10, 15-6 ~ 15-12
설치 15-5 ~ 15-6
소스 파일 15-2, 15-8
애플리케이션 배포 15-13
애플리케이션에서 사용 15-3 ~ 15-4
약한 패키징 15-11
옵션 15-7
이름 지정 15-9
중복 *참조* 15-10
컬렉션 15-13
컴파일 15-10 ~ 15-12
컴파일러 지시어 15-11
컴포넌트 15-8, 52-19
파일명 확장자 15-1, 15-8, 15-12
편집 15-7
프로젝트 옵션 파일 15-8
패키지 빌드 15-10
패키지 연결 15-12
패키지 컬렉션 파일 15-13
패키지 파일 17-3
패턴 10-9
팩토리 34-5
페이지 디스패처 34-35
페이지 모듈 34-2, 34-4
페이지 컨트롤 9-13
페이지 추가 9-13
페이지 프로듀서 33-14 ~ 33-17, 34-2, 34-4, 34-6 ~ 34-7, B-1
Content 메소드 33-15
ContentFromStream 메소드 33-15
ContentFromString 메소드 33-15
데이터 인식 29-36 ~ 29-40, 33-18
연결 33-16
이벤트 처리 33-15, 33-16, 33-17
타입 34-10
템플릿 34-4
템플릿 변환 33-15
펜 10-5, 54-6
그리기 모드 10-29
너비 10-6
디폴트 설정 10-6
변경 54-8
브러시 10-5
색상 10-6
스타일 10-6
위치 파악 10-7
위치, 설정 10-8, 10-25
편집 모드 22-17
취소 22-18
편집 컨트롤 6-6, 9-1 ~ 9-3, 19-2, 19-8
리치 에디트(rich edit) 서식 19-9
여러 줄 19-8

텍스트 선택 6-8
평균, decision cube 20-5
포인터
NULL A-7
VCL 함축 정보 13-5
디폴트 속성 값 47-11
역참조된 13-6
예외 처리 12-5
정수 타입 A-5
정수로 타입 변환 A-4
클래스 46-10
포커스 45-4
이동 9-6
필드 23-16
포트
서버 소켓 37-7
서비스 37-2
여러 연결 37-5
클라이언트 소켓 37-6
포함된 객체 38-9
폼 8-1
데이터 검색 8-8 ~ 8-11
데이터 동기화 19-4
드릴다운 19-14
런타임에 생성 8-5
로컬 변수로 생성 8-6
마스터/디테일 테이블 19-14
메모리 관리 8-5
메인 8-2
modal 8-4
modeless 8-4, 8-6
속성 쿼리
예제 8-8
연결 8-3
유닛 참조 추가 8-3
이벤트 핸들러 공유 10-15
인수 전달 8-7
전역 변수 8-5
참조 8-3
컴포넌트 57-1
표시 8-5
프로젝트에 추가 8-1 ~ 8-3
필드 추가 10-26 ~ 10-27
폼 마법사 58-3
폼 연결 8-3
폼 유닛
웹 애플리케이션 33-3
폼 파일 3-7, 14-2, 16-12
폼과 다이얼로그 박스 공유 7-24 ~ 7-26
표의 문자 16-3
약어 16-9
표준 이벤트 48-4
사용자 정의 48-5
표준 컴포넌트 5-6 ~ 5-8
프레임 8-11, 8-13 ~ 8-15
그래픽 8-14
리소스 8-14
컴포넌트 템플릿 8-13, 8-14
프로그래밍 템플릿 7-3
프로바이더 28-1 ~ 28-13, 29-3
XML 30-7 ~ 30-8
XML 문서에 데이터 제공 30-8 ~ 30-10
XML 문서에 연결 30-7
XML 문서와 연결 28-2
내부 27-18, 27-24, 28-1
데이터 제약 조건 28-12
데이터셋 관련 28-2
로컬 27-24, 28-3
업데이트 객체 사용 24-11
업데이트 검열 28-11
업데이트 적용 28-4, 28-8, 28-11
오류 처리 28-11
외부 18-11, 27-18, 27-24, 28-1
원격 27-25, 28-3, 29-6
클라이언트 데이터셋 27-24 ~ 27-31
클라이언트 생성 이벤트 28-12
프로시저
이름 지정 49-2
프로젝트
폼 추가 8-1 ~ 8-3
프로젝트 마법사 58-3
프로젝트 옵션 7-3
디폴트 7-3
프로젝트 템플릿 7-26
프로젝트 파일
변경 2-2
분산 2-5
프로토콜
네트워크 연결 24-15
선택 29-9 ~ 29-11
연결 컴포넌트 29-9 ~ 29-11, 29-22
인터넷 32-3, 37-1
프린터 A-2
플라이바이(fly-by) 도움말 9-15
플라이오버(fly-over) 도움말 19-30
플래그 56-4
픽셀
읽기 및 설정 10-9
필드 23-1 ~ 23-27
Null 값 22-21
값 변경 19-5
값 업데이트 19-5
값 표시 19-10, 23-17
값 할당 22-21
고유한 데이터 타입 23-22 ~ 23-27
기본값 23-20
데이터 검색 23-17
데이터 입력 22-18, 23-14
데이터베이스 56-6, 56-7
디폴트 형식 23-14
메시지 레코드 51-6
상호 배타적 옵션 19-2
속성 23-1
열거 21-13
영구적 열 19-17
유효한 데이터 제한 23-21 ~ 23-22
읽기 전용 19-5
폼에 추가 10-26 ~ 10-27
활성화 23-16
히든 28-4
필드 객체 23-1 ~ 23-27
동적 23-2 ~ 23-3
영구적과 비교 23-2
삭제 23-10
속성 23-1, 23-10 ~ 23-15
공유 23-12
런타임 23-12
액세스 값 23-19 ~ 23-20
영구적 23-3 ~ 23-16
동적과 비교 23-2
이벤트 23-15 ~ 23-16
정의 23-5 ~ 23-10
표시 및 편집 속성 23-11
필드 데이터 연결 클래스 56-12

필드 속성 23-12 ~ 23-14
  데이터 패킷 28-6
  제거 23-14
  할당 23-13
필드 정의 22-38
필드 타입
  변환 23-16, 23-18 ~ 23-19
필터 22-12 ~ 22-16
  런타임 시 설정 22-15
  범위와 비교 22-30
  북마크 사용 25-10 ~ 25-11
  비어 있는 필드 22-14
  연산자 22-14
  예외 처리 12-8
  정의 22-13 ~ 22-15
  쿼리와 비교 22-12
  클라이언트 데이터셋 27-3 ~ 27-5
    매개변수 사용 27-28
  활성화/비활성화 22-13

ㅎ

하위 메뉴 8-33
하위 컴포넌트
  속성 47-9
하이퍼텍스트 링크
  HTML에 추가 33-14
할당 연산자 13-6
함수 45-6
  C++와 오브젝트 파스칼 비교 13-23
  Windows API 45-3, 50-1
  가상 13-12
  그래픽 50-1
  반환 타입 49-2
  속성 설정 52-11
  속성 읽기 47-6, 52-9, 52-10
  수치 연산 A-7
  이름 지정 49-2
  인수 13-7
합집합
  Type Library Editor 39-10, 39-17
  다른 타입의 멤버 A-4
  액세스 A-4
핸들
  리소스 모듈 16-11
  소켓 연결 37-6
행 9-15
  decision grid 20-11
헤더
  HTTP 요청 32-4
  owner-draw 6-11
헤더 컨트롤 9-14
현재 월 반환 55-9
현지 날짜 A-10
현지 시간 A-10
형식 A-10
  C++와 오브젝트 파스칼 비교 13-20
  소유자(holder) 클래스 36-6
  지정되지 않음 13-17
호스트 29-23, 37-4
  URL 32-3
  주소 37-4
호스트 이름 37-4
  IP 주소와 비교 37-4
호출 동기화 44-21
화면
  새로 고침 10-2
  해상도 17-12
    프로그래밍 17-12
확장 문자 집합 A-2
확장성 18-11
환경 A-9
활동
  트랜잭션 객체 44-20 ~ 44-21
활성화 속성
  공유 속성 44-7
후기 연결 31-13
  Automation 41-12, 41-13
히든 필드 28-4
힌트 9-15