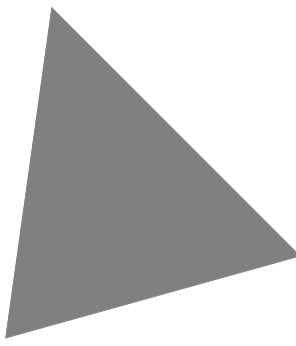




# 개발자 안내서

---



**Borland®**  
**Kylix™**  
**Delphi™ for Linux®**

볼랜드 코리아 주식회사  
서울특별시 강남구 삼성동 159-1 ASEM 타워 30 층  
연락처 : (02) 6001-3162  
[www.borlandkorea.co.kr](http://www.borlandkorea.co.kr)



Delphi GUI Internet  
**Kylix™** Linux  
RAD  
Rapid application development for Linux®

볼랜드와 볼랜드 코리아는 이 문서에 포함된 모든 내용에 대한 특허를 가지고 있습니다.

이 문서를 공급함에 있어 사용자에게 특허권에 대한 어떠한 라이선스도 주어지지 않습니다.

COPYRIGHT © 1997, 2001 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Other product names are trademarks or registered trademarks of their respective holders.

제품 구입 문의: Tel:02-6001-3194, 02-6001-3191

Fax:02-6001-3199

볼랜드 코리아

서울시 강남구 삼성동 159-1 ASEM 타워 30 층

# 목 차

## 1 장

|            |     |
|------------|-----|
| 서문         | 1-1 |
| 설명서 내용     | 1-1 |
| 설명서 규칙     | 1-2 |
| 개발자 지원 서비스 | 1-2 |
| 인쇄된 설명서 주문 | 1-2 |

## 1 부

### Kylix 프로그래밍

## 2 장

|                       |     |
|-----------------------|-----|
| Kylix 를 사용한 애플리케이션 개발 | 2-1 |
| 통합 개발 환경              | 2-1 |
| 애플리케이션 디자인            | 2-2 |
| 애플리케이션 개발             | 2-2 |
| 프로젝트 생성               | 2-3 |
| 코드 편집                 | 2-3 |
| 애플리케이션 컴파일            | 2-4 |
| 애플리케이션 디버깅            | 2-4 |
| 애플리케이션 배포             | 2-5 |

## 3 장

|                             |      |
|-----------------------------|------|
| CLX 사용                      | 3-1  |
| CLX 이해                      | 3-1  |
| 속성, 메소드 및 이벤트               | 3-3  |
| 속성                          | 3-3  |
| 메소드                         | 3-4  |
| 이벤트                         | 3-4  |
| widget 이벤트                  | 3-4  |
| 시스템 이벤트                     | 3-4  |
| 객체의 개념                      | 3-5  |
| Kylix 객체 검사                 | 3-5  |
| 컴포넌트 이름 변경                  | 3-7  |
| 객체에서 데이터와 코드 상속             | 3-8  |
| 유효 범위(scope) 및 한정자          | 3-8  |
| private, protected, public, |      |
| published 선언                | 3-9  |
| 객체 변수 사용                    | 3-10 |
| 객체 생성, 인스턴스화 및 소멸           | 3-10 |
| 컴포넌트와 소유권                   | 3-11 |
| CLX 계층의 주요 분기               | 3-12 |
| TObject 분기                  | 3-13 |
| TPersistent 분기              | 3-13 |

|                   |      |
|-------------------|------|
| TComponent 분기     | 3-14 |
| TControl 분기       | 3-15 |
| TWidgetControl 분기 | 3-16 |
| 컴포넌트 사용           | 3-17 |
| 컴포넌트 팔레트의 컴포넌트    | 3-18 |
| 텍스트 컨트롤           | 3-18 |
| 텍스트 컨트롤 속성        | 3-19 |
| 메모 컨트롤의 속성        | 3-19 |
| 특수한 입력 컨트롤        | 3-20 |
| 스크롤 막대            | 3-20 |
| 트랙 표시줄            | 3-20 |
| 스핀 편집 컨트롤         | 3-21 |
| 버튼 및 이와 유사한 컨트롤   | 3-21 |
| 버튼 컨트롤            | 3-21 |
| 비트맵 버튼            | 3-22 |
| 스피드 버튼            | 3-22 |
| 확인란               | 3-22 |
| 라디오 버튼            | 3-23 |
| 툴바                | 3-23 |
| 스플리터 컨트롤          | 3-23 |
| 목록 처리             | 3-24 |
| 목록 상자와 확인 목록 상자   | 3-24 |
| 콤보 상자             | 3-25 |
| 트리 뷰              | 3-25 |
| 목록 뷰              | 3-25 |
| 컴포넌트 그룹화          | 3-26 |
| 그룹 상자와 라디오 그룹     | 3-26 |
| 패널                | 3-26 |
| 스크롤 상자            | 3-27 |
| 탭 컨트롤             | 3-27 |
| 페이지 컨트롤           | 3-27 |
| 헤더 컨트롤            | 3-27 |
| 비주얼 피드백 제공        | 3-28 |
| 레이블               | 3-28 |
| 상태 표시줄            | 3-28 |
| 진행 표시줄            | 3-29 |
| 도움말과 힌트 속성        | 3-29 |
| 그리드               | 3-29 |
| 그리기 그리드           | 3-29 |
| 문자열 그리드           | 3-30 |
| 그래픽 표시            | 3-30 |
| 이미지               | 3-30 |
| 도형(shape)         | 3-30 |
| 빗면                | 3-31 |
| 그리기 상자            | 3-31 |
| 대화 상자             | 3-31 |
| 열린 대화 상자 사용       | 3-31 |

|                           |      |
|---------------------------|------|
| helper 객체 사용 . . . . .    | 3-32 |
| 목록 사용 . . . . .           | 3-32 |
| 문자열 목록 사용 . . . . .       | 3-32 |
| 문자열 목록의 로드 및 저장 . . . . . | 3-33 |
| 새 문자열 목록 작성 . . . . .     | 3-33 |
| 목록에서 문자열 처리 . . . . .     | 3-35 |
| 문자열 목록에 객체 연결 . . . . .   | 3-37 |
| 드로잉 공간 만들기 . . . . .      | 3-37 |
| 인쇄 . . . . .              | 3-38 |
| 스트림 사용 . . . . .          | 3-39 |

## 4 장

|                                       |      |
|---------------------------------------|------|
| 일반적인 프로그래밍 작업 . . . . .               | 4-1  |
| 클래스 이해 . . . . .                      | 4-1  |
| 클래스 정의 . . . . .                      | 4-2  |
| 예외 처리 . . . . .                       | 4-4  |
| 코드 블록 보호 . . . . .                    | 4-4  |
| 예외에 대한 응답 . . . . .                   | 4-4  |
| 예외와 제어 흐름 . . . . .                   | 4-5  |
| 예외 응답 중첩 . . . . .                    | 4-6  |
| 리소스 할당 보호 . . . . .                   | 4-6  |
| 보호가 필요한 리소스 종류 . . . . .              | 4-7  |
| 리소스 보호 블록 만들기 . . . . .               | 4-7  |
| RTL(Run-Time Library) 예외 처리 . . . . . | 4-8  |
| RTL 예외 . . . . .                      | 4-9  |
| 예외 핸들러 작성 . . . . .                   | 4-9  |
| 예외 처리 문장 . . . . .                    | 4-10 |
| 예외 인스턴스 사용 . . . . .                  | 4-11 |
| 예외 핸들러의 범위 . . . . .                  | 4-11 |
| 기본 예외 핸들러 제공 . . . . .                | 4-12 |
| 예외 클래스 다루기 . . . . .                  | 4-12 |
| 예외 재발생 . . . . .                      | 4-13 |
| 컴포넌트 예외 처리 . . . . .                  | 4-14 |
| 외부 소스를 사용한 예외 처리 . . . . .            | 4-14 |
| 조용한 예외 . . . . .                      | 4-15 |
| 사용자 고유의 예외 정의 . . . . .               | 4-16 |
| 예외 객체 타입 선언 . . . . .                 | 4-16 |
| 예외 발생 . . . . .                       | 4-16 |
| 인터페이스 사용 . . . . .                    | 4-17 |
| 랭귀지 기능의 인터페이스 . . . . .               | 4-17 |
| 계층 구조에 걸친 인터페이스 구현 . . . . .          | 4-18 |
| 프로시저와 함께 인터페이스 사용 . . . . .           | 4-19 |
| IInterface 구현 . . . . .               | 4-20 |
| TInterfacedObject . . . . .           | 4-20 |
| as 연산자 사용 . . . . .                   | 4-21 |
| 코드 재사용 및 위임 (Delegation) . . . . .    | 4-22 |
| 위임을 위한 implements 사용 . . . . .        | 4-22 |
| 인터페이스 객체의 메모리 관리 . . . . .            | 4-23 |

|                                  |      |
|----------------------------------|------|
| 참조 카운팅 사용 . . . . .              | 4-23 |
| 참조 카운팅을 사용하지 않음 . . . . .        | 4-24 |
| 문자열 작업 . . . . .                 | 4-25 |
| 문자 타입(Character types) . . . . . | 4-25 |
| 문자열 타입(String types) . . . . .   | 4-26 |
| 짧은 문자열 . . . . .                 | 4-26 |
| 긴 문자열 . . . . .                  | 4-27 |
| WideString . . . . .             | 4-27 |
| PChar 타입 . . . . .               | 4-27 |
| OpenString . . . . .             | 4-28 |
| 런타임 라이브러리 문자열 처리 루틴 . . . . .    | 4-28 |
| 와이드 문자 루틴 . . . . .              | 4-28 |
| 일반적으로 사용되는 긴 문자열 루틴 . . . . .    | 4-29 |
| 문자열 선언 및 초기화 . . . . .           | 4-31 |
| 문자열 타입의 혼합 및 변환 . . . . .        | 4-32 |
| 문자열에서 PChar 로 변환 . . . . .       | 4-33 |
| 문자열 종속성 . . . . .                | 4-33 |
| PChar 지역 변수 반환 . . . . .         | 4-33 |
| 지역 변수를 Pchar 로 전달 . . . . .      | 4-34 |
| 문자열에 대한 컴파일러 지시어 . . . . .       | 4-35 |
| 문자열과 문자: 관련 주제 . . . . .         | 4-36 |
| 파일 작업 . . . . .                  | 4-36 |
| 파일 처리 . . . . .                  | 4-36 |
| 파일 삭제 . . . . .                  | 4-37 |
| 파일 찾기 . . . . .                  | 4-37 |
| 파일 속성 변경 . . . . .               | 4-39 |
| 파일 이름 재지정 . . . . .              | 4-39 |
| 파일 date-time 루틴 . . . . .        | 4-40 |
| 파일 I/O 와 파일 형식 . . . . .         | 4-40 |
| 파일 스트림 사용 . . . . .              | 4-41 |
| 파일 생성 및 열기 . . . . .             | 4-41 |
| 파일 핸들 사용 . . . . .               | 4-42 |
| 파일 읽기 및 쓰기 . . . . .             | 4-43 |
| 문자열 읽기 및 쓰기 . . . . .            | 4-44 |
| 파일 찾기 . . . . .                  | 4-44 |
| 파일 위치 및 크기 . . . . .             | 4-45 |
| 복사 . . . . .                     | 4-45 |
| 오브젝트 파스칼 데이터 타입 . . . . .        | 4-45 |

## 5 장

|                                 |     |
|---------------------------------|-----|
| 애플리케이션과 공유 객체 구축 . . . . .      | 5-1 |
| 애플리케이션 생성 . . . . .             | 5-1 |
| GUI 애플리케이션 . . . . .            | 5-1 |
| 사용자 인터페이스 모델 . . . . .          | 5-1 |
| IDE, 프로젝트 및 컴파일 옵션 설정 . . . . . | 5-2 |
| 콘솔 애플리케이션 . . . . .             | 5-2 |
| 패키지와 공유 객체 파일 생성 . . . . .      | 5-3 |
| 공유 객체 라이브러리 사용 . . . . .        | 5-3 |



|                                      |      |
|--------------------------------------|------|
| 패키지와 공유 객체를 사용하는 경우 . . .            | 5-4  |
| 데이터베이스 애플리케이션 개발 . . . . .           | 5-4  |
| 분산 애플리케이션 구축 . . . . .               | 5-5  |
| TCP/IP 를 사용한 분산 애플리케이션 . . .         | 5-5  |
| 애플리케이션에서 소켓 사용 . . . . .             | 5-5  |
| 웹 서버 애플리케이션 만들기 . . . . .            | 5-6  |
| 데이터 모듈 사용 . . . . .                  | 5-6  |
| 데이터 모듈 만들기 . . . . .                 | 5-7  |
| 데이터 모듈에서 비즈니스 룰 만들기 . .              | 5-7  |
| 폼에서 데이터 모듈 액세스 . . . . .             | 5-7  |
| 템플릿 프로그래밍 . . . . .                  | 5-8  |
| 코드 공유 : Object Repository 사용 . . . . | 5-8  |
| 프로젝트 내에서 항목 공유 . . . . .             | 5-8  |
| 항목 추가 : Object Repository . . . . .  | 5-8  |
| 템 환경에서 객체 공유 . . . . .               | 5-9  |
| 프로젝트에서 Object Repository             |      |
| 항목 사용 . . . . .                      | 5-9  |
| 항목 복사 . . . . .                      | 5-10 |
| 항목 상속 . . . . .                      | 5-10 |
| 항목 사용 . . . . .                      | 5-10 |
| 프로젝트 템플릿 사용 . . . . .                | 5-10 |
| 공유 항목 수정 . . . . .                   | 5-10 |
| 기본 프로젝트, 새 폼 및 메인 폼 지정 .             | 5-11 |
| 컴포넌트와 컴포넌트 그룹의 재사용 . . . .           | 5-11 |
| 컴포넌트 템플릿 생성 및 사용 . . . . .           | 5-11 |
| CLX 애플리케이션에서 도움말 사용 . . . .          | 5-13 |
| 도움말 시스템 인터페이스 . . . . .              | 5-13 |
| IcustomHelpViewer 구현 . . . . .       | 5-14 |
| Help Manager 와 통신 . . . . .          | 5-14 |
| Help Manager 에 정보 요청 . . . . .       | 5-15 |
| 키워드 방식 도움말 표시 . . . . .              | 5-15 |
| 목차 표시 . . . . .                      | 5-16 |
| IextendedHelpViewer 구현 . . . . .     | 5-17 |
| IHelpSelector 구현 . . . . .           | 5-18 |
| 도움말 시스템 객체 등록 . . . . .              | 5-18 |
| 도움말 뷰어 등록 . . . . .                  | 5-18 |
| 도움말 선택기 등록 . . . . .                 | 5-19 |
| CLX 애플리케이션에서 도움말 사용 . . . .          | 5-19 |
| TApplication 이 도움말을                  |      |
| 처리하는 방법 . . . . .                    | 5-19 |
| 컨트롤의 도움말 처리 방법 . . . . .             | 5-19 |
| 도움말 시스템 직접 호출 . . . . .              | 5-20 |
| IhelpSystem 사용 . . . . .             | 5-20 |
| IDE 도움말 시스템 사용자 지정 . . . . .         | 5-21 |

## 6 장

|                         |     |
|-------------------------|-----|
| 애플리케이션 사용자 인터페이스 개발 . . | 6-1 |
| 애플리케이션 동작 제어 . . . . .  | 6-1 |

|                            |      |
|----------------------------|------|
| 메인 폼 사용 . . . . .          | 6-1  |
| 폼 추가 . . . . .             | 6-2  |
| 폼 연결 . . . . .             | 6-2  |
| 메인 폼 숨기기 . . . . .         | 6-2  |
| 애플리케이션 레벨에서 작업 . . . . .   | 6-3  |
| 애플리케이션의 록앤필 설정 . . . . .   | 6-3  |
| 화면 처리 . . . . .            | 6-3  |
| 레이아웃 관리 . . . . .          | 6-4  |
| 폼 사용 . . . . .             | 6-4  |
| 메모리에 상주하는 폼의 제어 . . . . .  | 6-5  |
| 자동 생성된 폼 표시 . . . . .      | 6-5  |
| 동적으로 폼 생성 . . . . .        | 6-5  |
| 창 같은 모달리스 폼 생성 . . . . .   | 6-6  |
| 로컬 변수로 폼 인스턴스 생성 . . . . . | 6-6  |
| 폼에 추가 인수 전달 . . . . .      | 6-7  |
| 폼에서 데이터 검색 . . . . .       | 6-8  |
| 모달리스 폼에서 데이터 검색 . . . .    | 6-8  |
| 모달 폼에서 데이터 검색 . . . . .    | 6-9  |
| 프레임 사용 . . . . .           | 6-11 |
| 프레임 생성 . . . . .           | 6-11 |
| 컴포넌트 팔레트에 프레임 추가 . . . .   | 6-12 |
| 프레임 사용 및 수정 . . . . .      | 6-12 |
| 프레임 공유 . . . . .           | 6-13 |
| 작업 목록 사용 . . . . .         | 6-13 |
| 작업(action)의 개념 . . . . .   | 6-14 |
| 작업 목록 설정 . . . . .         | 6-14 |
| 작업 실행 시 발생하는 이벤트 . . . . . | 6-15 |
| 이벤트에 응답 . . . . .          | 6-15 |
| 작업이 해당 대상을 찾는 방법 . . . .   | 6-17 |
| 작업 업데이트 . . . . .          | 6-17 |
| 이미 정의된 작업 클래스 . . . . .    | 6-18 |
| 작업 컴포넌트 작성 . . . . .       | 6-18 |
| 작업 등록 . . . . .            | 6-19 |
| 메뉴 생성 및 관리 . . . . .       | 6-20 |
| 메뉴 디자인 . . . . .           | 6-20 |
| 메뉴 작성 . . . . .            | 6-22 |
| 메뉴 이름 지정 . . . . .         | 6-23 |
| 메뉴 항목 이름 지정 . . . . .      | 6-23 |
| 메뉴 항목 추가, 삽입 및 삭제 . . . .  | 6-24 |
| 하위 메뉴 생성 . . . . .         | 6-25 |
| 메뉴 항목에 이미지 추가 . . . . .    | 6-27 |
| 메뉴 보기 . . . . .            | 6-27 |
| Object Inspector 에서        |      |
| 메뉴 항목 편집 . . . . .         | 6-28 |
| 메뉴 디자이너 컨텍스트 메뉴 사용 . . .   | 6-28 |
| 컨텍스트 메뉴의 명령 . . . . .      | 6-29 |
| 디자인 타임 시 메뉴 간의 전환 . . .    | 6-29 |
| 메뉴 템플릿 사용 . . . . .        | 6-30 |
| 메뉴를 템플릿으로 저장 . . . . .     | 6-31 |

|                               |      |
|-------------------------------|------|
| 템플릿 메뉴 항목 및 이벤트               |      |
| 핸들러의 이름 지정 규칙 . . . . .       | 6-32 |
| 런타임 시 메뉴 항목 처리 . . . . .      | 6-32 |
| 툴바 디자인 . . . . .              | 6-32 |
| 패널 컴포넌트를 사용하여 툴바 추가 . . . . . | 6-33 |
| 패널에 스피드 버튼 추가 . . . . .       | 6-34 |
| 스피드 버튼의 glyph 할당 . . . . .    | 6-34 |
| 스피드 버튼의 초기 상태 설정 . . . . .    | 6-34 |
| 스피드 버튼 그룹 생성 . . . . .        | 6-35 |
| 버튼 토글 허용 . . . . .            | 6-35 |
| 툴바 컴포넌트를 사용하여 툴바 추가 . . . . . | 6-35 |
| 툴 버튼 추가 . . . . .             | 6-36 |
| 툴 버튼에 이미지 할당 . . . . .        | 6-36 |
| 툴 버튼 모양 및 초기 상태 설정 . . . . .  | 6-37 |
| 툴 버튼 그룹 생성 . . . . .          | 6-37 |
| 툴 버튼 토글 허용 . . . . .          | 6-37 |
| 클릭 이벤트에 응답 . . . . .          | 6-37 |
| 툴 버튼에 메뉴 할당 . . . . .         | 6-38 |
| 숨겨진 툴바 추가 . . . . .           | 6-38 |
| 툴바 숨기기 및 표시 . . . . .         | 6-38 |

## 7 장

|                               |     |
|-------------------------------|-----|
| 컨트롤 사용 . . . . .              | 7-1 |
| 컨트롤 내에서 텍스트 작업 . . . . .      | 7-1 |
| 텍스트 정렬 설정 . . . . .           | 7-1 |
| 런타임 시 스크롤 막대 추가 . . . . .     | 7-2 |
| 애플리케이션에 클립보드 추가 . . . . .     | 7-3 |
| 텍스트 선택 . . . . .              | 7-3 |
| 모든 텍스트 선택 . . . . .           | 7-4 |
| 텍스트 잘라내기, 복사 및 붙여넣기 . . . . . | 7-4 |
| 선택한 텍스트 삭제 . . . . .          | 7-5 |
| 메뉴 항목 사용 불가능 . . . . .        | 7-5 |
| 팝업 메뉴 제공 . . . . .            | 7-5 |
| OnPopup 이벤트 처리 . . . . .      | 7-6 |
| 컨트롤에 그래픽 추가 . . . . .         | 7-7 |
| 컨트롤이 owner-draw               |     |
| 항목임을 나타내기 . . . . .           | 7-7 |
| 문자열 목록에 그래픽 객체 추가 . . . . .   | 7-8 |
| 애플리케이션에 이미지 추가 . . . . .      | 7-8 |
| owner-draw 항목 크기 설정 . . . . . | 7-8 |
| Owner-draw 항목 그리기 . . . . .   | 7-9 |

## 8 장

|                        |     |
|------------------------|-----|
| 그래픽 작업 . . . . .       | 8-1 |
| 그래픽 프로그래밍 개요 . . . . . | 8-1 |
| 화면 새로 고침 . . . . .     | 8-2 |
| 그래픽 객체 형식 . . . . .    | 8-2 |

|                                    |      |
|------------------------------------|------|
| 캔버스의 일반적인 속성과 메소드 . . . . .        | 8-3  |
| 캔버스 객체의 속성 사용 . . . . .            | 8-5  |
| 펜 사용 . . . . .                     | 8-5  |
| 브러시 사용 . . . . .                   | 8-7  |
| 브러시 색상 변경 . . . . .                | 8-8  |
| 브러시 스타일 변경 . . . . .               | 8-8  |
| 브러시의 비트맵 속성 설정 . . . . .           | 8-9  |
| 캔버스 메소드를 사용하여                      |      |
| 그래픽 객체 그리기 . . . . .               | 8-9  |
| 선 및 다각선 그리기 . . . . .              | 8-9  |
| 도형 그리기 . . . . .                   | 8-10 |
| 애플리케이션에서 여러 드로잉 객체 처리 . . . . .    | 8-12 |
| 사용할 드로잉 툴 파악 . . . . .             | 8-12 |
| 스피드 버튼으로 툴 변경 . . . . .            | 8-13 |
| 드로잉 툴 사용 . . . . .                 | 8-13 |
| 도형 그리기 . . . . .                   | 8-14 |
| 이벤트 핸들러 간의 코드 공유 . . . . .         | 8-15 |
| 그래픽에서 그리기 . . . . .                | 8-16 |
| 스크롤 가능한 그래픽 만들기 . . . . .          | 8-16 |
| 이미지 컨트롤 추가 . . . . .               | 8-17 |
| 컨트롤 두기 . . . . .                   | 8-17 |
| 초기 비트맵 크기 설정 . . . . .             | 8-17 |
| 비트맵에 그리기 . . . . .                 | 8-18 |
| 그래픽 파일 로드 및 저장 . . . . .           | 8-19 |
| 파일에서 그림 로드 . . . . .               | 8-19 |
| 그림을 파일에 저장 . . . . .               | 8-19 |
| 그림 교체 . . . . .                    | 8-20 |
| 클립보드에서 그래픽 사용 . . . . .            | 8-21 |
| 클립보드에 그래픽 복사 . . . . .             | 8-21 |
| 클립보드로 그림 잘라내기 . . . . .            | 8-21 |
| 클립보드에서 그림 붙여넣기 . . . . .           | 8-22 |
| 양단 묶음(Rubber banding) 예제 . . . . . | 8-23 |
| 마우스에 응답 . . . . .                  | 8-23 |
| 마우스 이벤트의 내용 . . . . .              | 8-23 |
| 마우스 다운 동작에 응답 . . . . .            | 8-24 |
| 마우스 업 동작에 응답 . . . . .             | 8-25 |
| 마우스 이동에 응답 . . . . .               | 8-25 |
| 마우스 동작을 추적하기 위해                    |      |
| 폼 객체에 필드 추가 . . . . .              | 8-25 |
| 선 그리기 다듬기 . . . . .                | 8-27 |
| 원래 지점 추적 . . . . .                 | 8-27 |
| 이동 추적 . . . . .                    | 8-27 |

## 9 장

|                            |     |
|----------------------------|-----|
| 다중 스레드 애플리케이션 작성 . . . . . | 9-1 |
| 스레드 객체 정의 . . . . .        | 9-1 |
| 스레드 초기화 . . . . .          | 9-2 |
| 기본 우선 순위 할당 . . . . .      | 9-3 |

|                             |      |
|-----------------------------|------|
| 스레드 해제 시기 표시 . . . . .      | 9-3  |
| 스레드 함수 작성 . . . . .         | 9-3  |
| 메인 CLX 스레드 사용 . . . . .     | 9-3  |
| 스레드 로컬 변수 사용 . . . . .      | 9-5  |
| 다른 스레드로 종료 확인 . . . . .     | 9-5  |
| 스레드 함수에서의 예외 처리 . . . . .   | 9-6  |
| 지우기 코드 작성 . . . . .         | 9-6  |
| 스레드 조정 . . . . .            | 9-6  |
| 동시 액세스 피하기 . . . . .        | 9-7  |
| 객체 잠금 . . . . .             | 9-7  |
| 임계 구역 사용 . . . . .          | 9-7  |
| 다중 읽기 배타적 쓰기                |      |
| 동기화 장치 사용 . . . . .         | 9-8  |
| 메모리 공유에 이용되는                |      |
| 그 밖의 기술 . . . . .           | 9-8  |
| 다른 스레드 기다리기 . . . . .       | 9-9  |
| 스레드의 실행 완료 대기 . . . . .     | 9-9  |
| 작업 완료 대기 . . . . .          | 9-9  |
| 스레드 객체 실행 . . . . .         | 9-10 |
| 기본 우선 순위 오버라이드 . . . . .    | 9-11 |
| 스레드 시작 및 중지 . . . . .       | 9-11 |
| 다중 스레드 애플리케이션 디버깅 . . . . . | 9-12 |

## 10 장

|                                      |       |
|--------------------------------------|-------|
| 크로스 플랫폼 애플리케이션 개발 . . . . .          | 10-1  |
| Windows 애플리케이션을 Linux 로 포팅 . . . . . | 10-1  |
| 포팅 기법 . . . . .                      | 10-1  |
| 플랫폼 특정 포팅 . . . . .                  | 10-2  |
| 크로스 플랫폼 포팅 . . . . .                 | 10-2  |
| Windows 에뮬레이션 포팅 . . . . .           | 10-2  |
| 애플리케이션 포팅 . . . . .                  | 10-2  |
| CLX 와 VCL 비교 . . . . .               | 10-3  |
| CLX 가 다르게 하는 동작 . . . . .            | 10-4  |
| 룩앤필 (Look and feel) . . . . .        | 10-4  |
| 스타일 . . . . .                        | 10-5  |
| 가변 타입 . . . . .                      | 10-5  |
| 레지스트리 사용 안함 . . . . .                | 10-5  |
| 그 밖의 차이점 . . . . .                   | 10-6  |
| CLX 에서 빠진 내용 . . . . .               | 10-6  |
| 포팅되지 않는 기능 . . . . .                 | 10-7  |
| Kylix 와 Delphi 유닛 비교 . . . . .       | 10-7  |
| CLX 객체 생성자의 차이점 . . . . .            | 10-11 |
| Windows 와 Linux 간의                   |       |
| 소스 파일 공유 . . . . .                   | 10-11 |
| Windows 와 Linux 간의                   |       |
| 환경적 차이점 . . . . .                    | 10-12 |
| Linux 의 디렉토리 구조 . . . . .            | 10-14 |
| 포팅 가능한 코드 작성 . . . . .               | 10-15 |

|                               |       |
|-------------------------------|-------|
| 조건 지시어 사용 . . . . .           | 10-16 |
| 조건 지시어 종료 . . . . .           | 10-17 |
| 메시지 나타내기 . . . . .            | 10-18 |
| 인라인 어셈블러 코드 포함 . . . . .      | 10-18 |
| 메시지와 시스템 이벤트 . . . . .        | 10-19 |
| Linux 에서의 프로그래밍 차이점 . . . . . | 10-20 |
| 크로스 플랫폼 데이터베이스                |       |
| 애플리케이션 . . . . .              | 10-21 |
| dbExpress 차이점 . . . . .       | 10-21 |
| 컴포넌트 수준 차이점 . . . . .         | 10-22 |
| 사용자 인터페이스 수준 차이점 . . . . .    | 10-23 |
| 데이터베이스 애플리케이션을                |       |
| Linux 로 포팅 . . . . .          | 10-24 |
| dbExpress 애플리케이션에서            |       |
| 데이터 업데이트 . . . . .            | 10-26 |
| 크로스 플랫폼 인터넷 애플리케이션 . . . . .  | 10-28 |
| 인터넷 애플리케이션을 Linux 로           |       |
| 포팅하기 . . . . .                | 10-28 |

## 11 장

|                             |       |
|-----------------------------|-------|
| 패키지와 컴포넌트 사용 . . . . .      | 11-1  |
| 패키지의 이점 . . . . .           | 11-1  |
| 패키지와 표준 공유 객체 파일 . . . . .  | 11-2  |
| 런타임 패키지 . . . . .           | 11-2  |
| 애플리케이션에서 패키지 사용 . . . . .   | 11-2  |
| 패키지 동적 로딩 . . . . .         | 11-3  |
| 사용할 런타임 패키지 결정 . . . . .    | 11-4  |
| 사용자 지정 패키지 . . . . .        | 11-4  |
| 디자인 타임 패키지 . . . . .        | 11-4  |
| 컴포넌트 패키지 설치 . . . . .       | 11-5  |
| 패키지 생성 및 편집 . . . . .       | 11-6  |
| 패키지 생성 . . . . .            | 11-6  |
| 기존 패키지 편집 . . . . .         | 11-7  |
| 수동으로 패키지 소스 파일 편집 . . . . . | 11-7  |
| 패키지 구조 이해 . . . . .         | 11-8  |
| 패키지 이름 지정 . . . . .         | 11-8  |
| Requires 절 . . . . .        | 11-8  |
| Contains 절 . . . . .        | 11-9  |
| 패키지 컴파일 . . . . .           | 11-9  |
| 패키지 특정 컴파일러 지시어 . . . . .   | 11-10 |
| 명령줄 컴파일러와 링커 사용 . . . . .   | 11-11 |
| 컴파일이 성공했을 때 생성되는            |       |
| 패키지 파일 . . . . .            | 11-11 |
| 패키지 배포 . . . . .            | 11-12 |
| 패키지를 사용하는                   |       |
| 애플리케이션 배포 . . . . .         | 11-12 |
| 다른 개발자에 패키지 배포 . . . . .    | 11-12 |

## 12 장

|                 |      |
|-----------------|------|
| 국제적인 애플리케이션 만들기 | 12-1 |
| 국제화 및 지역화       | 12-1 |
| 국제화             | 12-1 |
| 지역화             | 12-1 |
| 애플리케이션 국제화      | 12-2 |
| 애플리케이션 코드 활성화   | 12-2 |
| 문자 집합           | 12-2 |
| 멀티바이트 문자 집합     | 12-2 |
| 와이드 문자          | 12-3 |
| 사용자 인터페이스 디자인   | 12-3 |
| 텍스트             | 12-4 |
| 그래픽 이미지         | 12-4 |
| 형식 및 정렬 순서      | 12-4 |
| 키보드 매핑          | 12-4 |
| 리소스 분리          | 12-5 |
| 리소스 모듈 만들기      | 12-5 |
| 리소스 모듈 사용       | 12-5 |
| 애플리케이션 지역화      | 12-6 |

## 13 장

|                                |       |
|--------------------------------|-------|
| 애플리케이션 배포                      | 13-1  |
| 일반 애플리케이션 배포                   | 13-1  |
| 배포 문제                          | 13-2  |
| 설치 프로그램 사용                     | 13-3  |
| 애플리케이션 파일 식별                   | 13-3  |
| 패키지 파일                         | 13-4  |
| Helper 애플리케이션                  | 13-4  |
| 공유 객체 파일 위치                    | 13-4  |
| 데이터베이스 애플리케이션 배포               | 13-4  |
| 데이터베이스에 연결                     | 13-5  |
| 구성 파일 업데이트                     | 13-5  |
| 웹 애플리케이션 배포                    | 13-6  |
| Apache 에 배포                    | 13-6  |
| 다양한 호스트 환경을 위한 프로그래밍           | 13-7  |
| 화면 해상도와 색상 수                   | 13-7  |
| 동적으로 크기가 조정되지 않는 경우의 고려 사항     | 13-7  |
| 폼과 컨트롤 크기를 동적으로 조정하는 경우의 고려 사항 | 13-8  |
| 다양한 색상 수 지원                    | 13-9  |
| 글꼴                             | 13-9  |
| 소프트웨어 사용권 요구 사항                | 13-9  |
| Deploy.txt                     | 13-9  |
| README                         | 13-10 |
| 사용권 계약서                        | 13-10 |
| GPL 사용권 계약                     | 13-10 |
| 타사 (third-party) 제품 설명서        | 13-10 |

## 2 부

### 데이터베이스 애플리케이션 개발

## 14 장

|                                |       |
|--------------------------------|-------|
| 데이터베이스 애플리케이션 디자인              | 14-1  |
| 데이터베이스 사용                      | 14-1  |
| 데이터베이스의 타입                     | 14-2  |
| 데이터베이스 보안                      | 14-3  |
| 트랜잭션                           | 14-3  |
| 참조 무결성, 내장 프로시저 및 트리거          | 14-4  |
| 데이터베이스 아키텍처                    | 14-4  |
| 일반 구조                          | 14-5  |
| 사용자 인터페이스 폼                    | 14-5  |
| 데이터 모듈                         | 14-5  |
| 디스크에 저장된 데이터가 있는 클라이언트 데이터셋 사용 | 14-7  |
| 단방향 데이터셋 직접 사용                 | 14-8  |
| 클라이언트 데이터셋을 사용하여 레코드 버퍼링       | 14-9  |
| 다계층 아키텍처 사용                    | 14-12 |
| 방법 조합                          | 14-13 |

## 15 장

|                          |      |
|--------------------------|------|
| 데이터 컨트롤 사용               | 15-1 |
| 공통적인 데이터 컨트롤 기능의 사용      | 15-2 |
| 데이터 컨트롤과 데이터셋 연결         | 15-3 |
| 런타임 시 연결된 데이터셋 변경        | 15-3 |
| 데이터 소스의 사용 가능 및 사용 불가능   | 15-4 |
| 데이터 소스에서 변경 사항에 대한 응답    | 15-4 |
| 데이터 편집과 업데이트             | 15-5 |
| 컨트롤에서 사용자 입력의 편집 사용 가능   | 15-5 |
| 컨트롤에서의 데이터 편집            | 15-5 |
| 데이터 표시 사용 불가능 및 사용 가능    | 15-6 |
| 데이터 표시 새로 고침             | 15-7 |
| 마우스, 키보드 및 타이머 이벤트 사용 가능 | 15-7 |
| 데이터 구성 방법 선택             | 15-7 |
| 단일 레코드 표시                | 15-7 |
| 데이터를 레이블로 표시             | 15-8 |
| 편집 상자의 필드 표시 및 편집        | 15-8 |
| 메모 컨트롤에 텍스트 표시 및 편집      | 15-9 |

|                                 |       |
|---------------------------------|-------|
| 이미지 컨트롤의 그래픽 필드                 |       |
| 표시 및 편집 . . . . .               | 15-9  |
| 목록 상자와 콤보 상자의 데이터               |       |
| 표시 및 편집 . . . . .               | 15-10 |
| 확인란으로 부울 필드 값 처리 . . . . .      | 15-12 |
| 라디오 컨트롤로 필드값 제한 . . . . .       | 15-13 |
| 여러 레코드 표시 . . . . .             | 15-14 |
| TDBGrid 로 데이터 보기 및 편집 . . . . . | 15-15 |
| 기본 상태에서 그리드 컨트롤 사용 . . . . .    | 15-15 |
| 사용자 지정 그리드 생성 . . . . .         | 15-16 |
| 영구적 열 이해 . . . . .              | 15-16 |
| 영구적 열 생성 . . . . .              | 15-17 |
| 영구적 열 삭제 . . . . .              | 15-18 |
| 영구적 열 순서 정렬 . . . . .           | 15-19 |
| 디자인 타임에 열 속성 설정 . . . . .       | 15-19 |
| 조회 목록 열 정의 . . . . .            | 15-20 |
| 열에 버튼 두기 . . . . .              | 15-21 |
| 열에 기본값 복구 . . . . .             | 15-21 |
| 복합 필드 표시 . . . . .              | 15-21 |
| 그리드 옵션 설정 . . . . .             | 15-23 |
| 그리드 편집 . . . . .                | 15-24 |
| 그리드 그리기 제어 . . . . .            | 15-24 |
| 런타임 시 사용자 동작에 응답 . . . . .      | 15-25 |
| 레코드 탐색 및 처리 . . . . .           | 15-26 |
| 표시할 탐색기 버튼 선택 . . . . .         | 15-27 |
| 디자인 타임에 탐색기 버튼                  |       |
| 보이기 및 숨기기 . . . . .             | 15-27 |
| 런타임 시 탐색기 버튼                    |       |
| 보이기 및 숨기기 . . . . .             | 15-27 |
| 풍선 도움말 표시 . . . . .             | 15-28 |
| 여러 데이터셋에 단일 탐색기 사용 . . . . .    | 15-29 |

## 16 장

|                               |       |
|-------------------------------|-------|
| 데이터셋 이해                       | 16-1  |
| 데이터셋 형식 . . . . .             | 16-2  |
| 데이터셋 열기와 닫기 . . . . .         | 16-2  |
| 데이터셋 상태 결정 및 설정 . . . . .     | 16-3  |
| 데이터셋 비활성화 . . . . .           | 16-4  |
| 데이터셋 찾아보기 . . . . .           | 16-5  |
| 데이터셋 편집 사용 가능 . . . . .       | 16-6  |
| 새 레코드 삽입 사용 가능 . . . . .      | 16-7  |
| 인덱스 기반 작업 사용 가능 . . . . .     | 16-8  |
| 필드 계산 . . . . .               | 16-8  |
| 레코드 필터링 . . . . .             | 16-8  |
| 업데이트 적용 . . . . .             | 16-8  |
| 데이터셋 탐색 . . . . .             | 16-9  |
| First 및 Last 메소드 사용 . . . . . | 16-10 |
| Next 및 Prior 메소드 사용 . . . . . | 16-10 |

|                              |       |
|------------------------------|-------|
| MoveBy 메소드 사용 . . . . .      | 16-11 |
| Eof 및 Bof 속성 사용 . . . . .    | 16-11 |
| Eof . . . . .                | 16-11 |
| Bof . . . . .                | 16-12 |
| 레코드 표시 및 반환 . . . . .        | 16-13 |
| 데이터셋 검색 . . . . .            | 16-14 |
| Locate 사용 . . . . .          | 16-15 |
| Lookup 사용 . . . . .          | 16-16 |
| 필터를 사용한 데이터 서브셋              |       |
| 표시 및 편집 . . . . .            | 16-17 |
| 필터링 사용 가능 및 사용 불가능 . . . . . | 16-17 |
| 필터 생성 . . . . .              | 16-17 |
| 필터 옵션 설정 . . . . .           | 16-20 |
| 필터링된 데이터셋의 레코드 탐색 . . . . .  | 16-21 |
| 데이터 수정 . . . . .             | 16-22 |
| 레코드 편집 . . . . .             | 16-22 |
| 새 레코드 추가 . . . . .           | 16-23 |
| 레코드 삽입 . . . . .             | 16-23 |
| 레코드 추가 . . . . .             | 16-24 |
| 레코드 삭제 . . . . .             | 16-24 |
| 데이터베이스에 데이터 포스트 . . . . .    | 16-24 |
| 변경 취소 . . . . .              | 16-25 |
| 전체 레코드 수정 . . . . .          | 16-25 |
| 데이터셋 이벤트 사용 . . . . .        | 16-26 |
| 메소드 중지 . . . . .             | 16-27 |
| OnCalcFields 사용 . . . . .    | 16-27 |

## 17 장

|                                |       |
|--------------------------------|-------|
| 필드 컴포넌트 사용                     | 17-1  |
| 동적 필드 컴포넌트 . . . . .           | 17-2  |
| 영구적 필드 컴포넌트 . . . . .          | 17-3  |
| 영구적 필드 생성 . . . . .            | 17-4  |
| 영구적 필드 정렬 . . . . .            | 17-5  |
| 새 영구적 필드 정의 . . . . .          | 17-5  |
| 데이터 필드 정의 . . . . .            | 17-6  |
| 계산된 필드 정의 . . . . .            | 17-7  |
| 조회 필드 정의 . . . . .             | 17-8  |
| 집계 필드 정의 . . . . .             | 17-10 |
| 영구적 필드 컴포넌트 삭제 . . . . .       | 17-10 |
| 영구적 필드 속성 및 이벤트 설정 . . . . .   | 17-11 |
| 디자인 타임 시 표시 및                  |       |
| 편집 속성 설정 . . . . .             | 17-11 |
| 런타임 시 필드 컴포넌트 속성 설정            | 17-12 |
| 사용자 입력 조정 및 마스킹 . . . . .      | 17-13 |
| 숫자, 날짜 및 시간 필드에                |       |
| 대한 기본 서식 사용 . . . . .          | 17-13 |
| 이벤트 처리 . . . . .               | 17-14 |
| 런타임 시 필드 컴포넌트 메소드 사용 . . . . . | 17-15 |

|                                             |       |
|---------------------------------------------|-------|
| 필드값 표시, 변환 및 액세스 . . . . .                  | 17-16 |
| 표준 컨트롤의 필드 컴포넌트 값 표시 . . . . .              | 17-16 |
| 필드값 변환 . . . . .                            | 17-17 |
| 기본 데이터셋 속성으로<br>필드값 액세스 . . . . .           | 17-18 |
| 데이터셋의 Fields 속성으로<br>필드값 액세스 . . . . .      | 17-19 |
| 데이터셋의 FieldByName<br>메소드로 필드값 액세스 . . . . . | 17-19 |
| 필드의 현재 값 확인 . . . . .                       | 17-20 |
| 필드에 대한 기본 값 설정 . . . . .                    | 17-20 |
| 제약 조건 지정 . . . . .                          | 17-20 |
| 객체 필드 사용 . . . . .                          | 17-21 |
| ADT 및 배열 필드 표시 . . . . .                    | 17-22 |
| ADT 필드 작업 . . . . .                         | 17-22 |
| 영구적 필드 컴포넌트 사용 . . . . .                    | 17-22 |
| 데이터셋의 FieldByName<br>메소드 사용 . . . . .       | 17-23 |
| 데이터셋의 FieldValues 속성 사용 . . . . .           | 17-23 |
| ADT 필드의 FieldValues 속성 사용 . . . . .         | 17-23 |
| ADT 필드의 Fields 속성 사용 . . . . .              | 17-23 |
| 배열 필드 작업 . . . . .                          | 17-24 |
| 영구적 필드 사용 . . . . .                         | 17-24 |
| 배열 필드의 FieldValues 속성 사용 . . . . .          | 17-24 |
| 배열 필드의 Fields 속성 사용 . . . . .               | 17-24 |
| 데이터셋 필드 작업 . . . . .                        | 17-25 |
| 데이터셋 필드 표시 . . . . .                        | 17-25 |
| 중첩된 데이터셋의 데이터 액세스 . . . . .                 | 17-25 |
| 참조 필드 작업 . . . . .                          | 17-26 |
| 참조 필드 표시 . . . . .                          | 17-26 |
| 참조 필드의 데이터 액세스 . . . . .                    | 17-26 |

## 18 장

|                                        |      |
|----------------------------------------|------|
| 단방향 데이터셋 사용 . . . . .                  | 18-1 |
| 단방향 데이터셋의 타입 . . . . .                 | 18-2 |
| 서버에 연결 . . . . .                       | 18-2 |
| 표시할 데이터 지정 . . . . .                   | 18-3 |
| 쿼리의 결과 표시 . . . . .                    | 18-3 |
| TSQLDataSet 을 사용하여<br>쿼리 지정 . . . . .  | 18-4 |
| TSQLQuery 를 사용하여<br>쿼리 지정 . . . . .    | 18-4 |
| 쿼리의 매개변수 사용 . . . . .                  | 18-4 |
| 테이블의 레코드 표시 . . . . .                  | 18-6 |
| TSQLDataSet 을 사용하여<br>테이블 표시 . . . . . | 18-6 |
| TSQLTable 을 사용하여 테이블 표시 . . . . .      | 18-7 |
| 내장 프로시저의 결과 표시 . . . . .               | 18-7 |

|                                                         |       |
|---------------------------------------------------------|-------|
| TSQLDataSet 을 사용하여 내장<br>프로시저 지정 . . . . .              | 18-7  |
| TSQLStoredProc 를 사용하여<br>내장 프로시저 지정 . . . . .           | 18-8  |
| 내장 프로시저 매개변수 작업 . . . . .                               | 18-8  |
| 데이터 페치 . . . . .                                        | 18-10 |
| 데이터셋 준비 . . . . .                                       | 18-10 |
| 여러 데이터셋 페치 . . . . .                                    | 18-11 |
| 레코드를 반환하지 않는 명령 실행 . . . . .                            | 18-11 |
| 실행할 명령 지정 . . . . .                                     | 18-12 |
| 명령 실행 . . . . .                                         | 18-12 |
| 서버 메타데이터 작성 및 수정 . . . . .                              | 18-12 |
| 마스터 / 디테일 관계 설정 . . . . .                               | 18-14 |
| TSQLDataSet 또는 TSQLQuery 로<br>마스터 / 디테일 관계 설정 . . . . . | 18-14 |
| TSQLTable 로 마스터 / 디테일<br>관계 설정 . . . . .                | 18-15 |
| 스키마 정보 액세스 . . . . .                                    | 18-16 |
| 메타데이터에 대한 데이터셋 사용 후<br>데이터 페치 . . . . .                 | 18-17 |
| 메타데이터 데이터셋의 구조 . . . . .                                | 18-17 |
| 테이블에 대한 정보 . . . . .                                    | 18-17 |
| 내장 프로시저에 대한 정보 . . . . .                                | 18-18 |
| 필드에 대한 정보 . . . . .                                     | 18-18 |
| 인덱스에 대한 정보 . . . . .                                    | 18-19 |
| 내장 프로시저 매개변수에<br>대한 정보 . . . . .                        | 18-20 |

## 19 장

|                                 |       |
|---------------------------------|-------|
| 데이터베이스에 연결 . . . . .            | 19-1  |
| 연결 제어 . . . . .                 | 19-2  |
| 서버 연결 설명 . . . . .              | 19-2  |
| 드라이버 식별 . . . . .               | 19-2  |
| 연결 매개변수 지정 . . . . .            | 19-2  |
| 연결에 대한 설명 이름 지정 . . . . .       | 19-3  |
| Connection Editor 사용 . . . . .  | 19-3  |
| 서버 연결 열기 및 닫기 . . . . .         | 19-4  |
| 연결 열기 . . . . .                 | 19-4  |
| 데이터베이스 서버로부터<br>연결 해제 . . . . . | 19-5  |
| 서버 로그인 제어 . . . . .             | 19-6  |
| 트랜잭션 관리 . . . . .               | 19-7  |
| 트랜잭션 시작 . . . . .               | 19-8  |
| 트랜잭션 종료 . . . . .               | 19-8  |
| 성공적인 트랜잭션 종료 . . . . .          | 19-9  |
| 성공적이지 않은 트랜잭션 종료 . . . . .      | 19-9  |
| 트랜잭션 분리 레벨 지정 . . . . .         | 19-9  |
| 서버 메타데이터 액세스 . . . . .          | 19-10 |

|                             |       |
|-----------------------------|-------|
| 연결된 데이터셋 작업 . . . . .       | 19-11 |
| 서버로부터 연결을 해제하지 않고           |       |
| 데이터셋 닫기 . . . . .           | 19-11 |
| 연결된 데이터셋을 통한 반복 . . . . .   | 19-11 |
| 서버에 명령 전송 . . . . .         | 19-12 |
| 데이터베이스 애플리케이션 디버깅 . . . . . | 19-13 |
| SQL 명령을 모니터하기 위해            |       |
| TSQLMonitor 사용 . . . . .    | 19-13 |
| SQL 명령을 모니터하기 위해            |       |
| 콜백 사용 . . . . .             | 19-14 |

## 20 장

|                                    |       |
|------------------------------------|-------|
| 클라이언트 데이터셋 사용 . . . . .            | 20-1  |
| 클라이언트 데이터셋을 사용하는                   |       |
| 데이터 작업 . . . . .                   | 20-2  |
| 클라이언트 데이터셋에서 데이터 탐색 . . . . .      | 20-2  |
| 검색에 사용할 인덱스 지정 . . . . .           | 20-3  |
| Goto 메소드로 검색 수행 . . . . .          | 20-4  |
| Find 메소드로 검색 수행 . . . . .          | 20-4  |
| 성공적인 검색 이후 현재                      |       |
| 레코드 지정 . . . . .                   | 20-5  |
| 부분 키 검색 . . . . .                  | 20-5  |
| 검색 반복 또는 확장 . . . . .              | 20-5  |
| 레코드 표시 제한 . . . . .                | 20-5  |
| 범위와 필터의 차이점 이해 . . . . .           | 20-6  |
| 범위 지정 . . . . .                    | 20-6  |
| 범위 수정 . . . . .                    | 20-8  |
| 범위의 적용 또는 취소 . . . . .             | 20-9  |
| 마스터 / 디테일 관계 표시 . . . . .          | 20-10 |
| 클라이언트 데이터셋을 다른                     |       |
| 데이터셋의 디테일로 만들기 . . . . .           | 20-10 |
| 중첩 디테일 테이블 사용 . . . . .            | 20-12 |
| 데이터 값 제약 조건 . . . . .              | 20-13 |
| 데이터를 읽기 전용으로 만들기 . . . . .         | 20-14 |
| 데이터 편집 . . . . .                   | 20-14 |
| 변경 취소 . . . . .                    | 20-15 |
| 변경 내용 저장 . . . . .                 | 20-15 |
| 정렬과 인덱싱 . . . . .                  | 20-16 |
| 새 인덱스 추가 . . . . .                 | 20-16 |
| 인덱스 삭제와 전환 . . . . .               | 20-17 |
| 인덱스 관련 정보 얻기 . . . . .             | 20-18 |
| 인덱스를 사용하여 데이터 그룹화 . . . . .        | 20-18 |
| 계산된 값 표시 . . . . .                 | 20-19 |
| 클라이언트 데이터셋에서                       |       |
| 내부적으로 계산된 필드 사용 . . . . .          | 20-20 |
| 유지 관리되는 Aggregates 속성 사용 . . . . . | 20-20 |
| Aggregates 지정 . . . . .            | 20-20 |
| 레코드 그룹의 집계 . . . . .               | 20-21 |

|                                |       |
|--------------------------------|-------|
| 집계 값 얻기 . . . . .              | 20-22 |
| 데이터에 애플리케이션                    |       |
| 특정 정보 추가 . . . . .             | 20-23 |
| 다른 데이터셋에서 데이터 복사 . . . . .     | 20-23 |
| 데이터 직접 할당 . . . . .            | 20-23 |
| 클라이언트 데이터셋 커서 복제 . . . . .     | 20-24 |
| 프로바이더와 함께 클라이언트                |       |
| 데이터셋 사용 . . . . .              | 20-25 |
| 데이터 프로바이더 지정 . . . . .         | 20-25 |
| 소스 데이터셋에서 매개변수 얻기 . . . . .    | 20-26 |
| 소스 데이터셋에 매개변수 전달 . . . . .     | 20-27 |
| 쿼리 또는 내장 프로시저                  |       |
| 매개변수 보내기 . . . . .             | 20-28 |
| 매개변수로 레코드 제한 . . . . .         | 20-28 |
| 서버에서 실행할 명령 지정 . . . . .       | 20-28 |
| 소스 데이터셋에 데이터 요청 . . . . .      | 20-29 |
| 점진적 폐치 . . . . .               | 20-30 |
| 요구 즉시 폐치 . . . . .             | 20-30 |
| 레코드 업데이트 . . . . .             | 20-31 |
| 업데이트 적용 . . . . .              | 20-31 |
| 업데이트 오류 해결 . . . . .           | 20-32 |
| 레코드 새로 고침 . . . . .            | 20-33 |
| 사용자 지정 이벤트를 사용하여               |       |
| 프로바이더와 통신 . . . . .            | 20-34 |
| SQL 클라이언트 데이터셋 사용 . . . . .    | 20-35 |
| TSQLClientDataSet 사용 . . . . . | 20-35 |
| SQL 클라이언트 데이터셋 설정 . . . . .    | 20-36 |
| 내부 프로바이더 구성 . . . . .          | 20-37 |
| 파일 기반 데이터로 클라이언트               |       |
| 데이터셋 사용 . . . . .              | 20-38 |
| 새로운 데이터셋 생성 . . . . .          | 20-38 |
| 영구적 필드를 사용하여 새로운               |       |
| 데이터셋 생성 . . . . .              | 20-39 |
| 필드와 인덱스 정의를 사용하여               |       |
| 데이터셋 생성 . . . . .              | 20-39 |
| 기존 테이블을 기반으로                   |       |
| 데이터셋 생성 . . . . .              | 20-40 |
| 파일이나 스트림에서 데이터 로드 . . . . .    | 20-41 |
| 데이터에 변경 내용 합병 . . . . .        | 20-41 |
| 파일이나 스트림에 데이터 저장 . . . . .     | 20-42 |

## 21 장

|                              |      |
|------------------------------|------|
| 프로바이더 컴포넌트 사용 . . . . .      | 21-1 |
| 데이터의 소스 정하기 . . . . .        | 21-2 |
| 클라이언트 데이터셋과의 통신 . . . . .    | 21-2 |
| 업데이트 적용 방법 선택 . . . . .      | 21-4 |
| 데이터 패킷에 포함될 정보 제어 . . . . .  | 21-4 |
| 데이터 패킷에 표시되는 필드 지정 . . . . . | 21-4 |

|                      |       |
|----------------------|-------|
| 데이터 패킷에 영향을 주는 옵션 설정 | 21-5  |
| 데이터 패킷에 사용자 지정 정보 추가 | 21-6  |
| 클라이언트 데이터 요청에 대한 응답  | 21-7  |
| 클라이언트 업데이트 요청에 대한 응답 | 21-7  |
| 데이터베이스 업데이트 이전의      |       |
| 델타 패킷 편집             | 21-8  |
| 업데이트 적용 방법 변경        | 21-9  |
| 개별 업데이트 스크린          | 21-10 |
| 프로바이더에서 업데이트 오류 해결   | 21-11 |
| 단일 테이블을 나타내지 않는      |       |
| 데이터셋에 업데이트 적용        | 21-11 |
| 클라이언트 생성 이벤트에 응답     | 21-12 |

### 3 부

## 분산 애플리케이션 개발

### 22 장

|                                |       |
|--------------------------------|-------|
| 인터넷 서버 애플리케이션 만들기              | 22-1  |
| 용어 및 표준                        | 22-1  |
| URL(Uniform Resource Locator)의 |       |
| 각 부분                           | 22-2  |
| URI와 URL 비교                    | 22-2  |
| HTTP 요청 헤더 정보                  | 22-2  |
| HTTP 서버 활동                     | 22-3  |
| 클라이언트 요청 구성                    | 22-3  |
| CGI 요청 서비스                     | 22-4  |
| 동적 공유 객체 요청 서비스                | 22-4  |
| 웹 서버 애플리케이션                    | 22-4  |
| 웹 서버 애플리케이션의 유형                | 22-4  |
| CGI 독립형                        | 22-5  |
| Apache DSO 모듈                  | 22-5  |
| 웹 서버 애플리케이션 만들기                | 22-5  |
| 웹 모듈                           | 22-5  |
| 웹 애플리케이션 객체                    | 22-6  |
| 웹 서버 애플리케이션의 구조                | 22-6  |
| 웹 디스패처                         | 22-7  |
| 디스패처에 작업 추가                    | 22-7  |
| 요청 메시지 디스패칭                    | 22-8  |
| 작업 항목                          | 22-8  |
| 작업 항목 실행 시기 결정                 | 22-9  |
| 대상 URL                         | 22-9  |
| 요청 메소드 유형                      | 22-9  |
| 작업 항목 활성화 및 비활성화               | 22-9  |
| 기본 작업 항목 선택                    | 22-10 |
| 작업 항목으로 요청 메시지에 응답             | 22-10 |
| 응답 보내기                         | 22-11 |
| 여러 작업 항목 사용                    | 22-11 |
| 클라이언트 요청 정보 액세스                | 22-11 |

|                        |       |
|------------------------|-------|
| 요청 헤더 정보를 포함하는 속성      | 22-12 |
| 대상을 식별하는 속성            | 22-12 |
| 웹 클라이언트를 설명하는 속성       | 22-12 |
| 요청 목적을 식별하는 속성         | 22-12 |
| 예상 응답을 설명하는 속성         | 22-13 |
| 컨텐츠를 설명하는 속성           | 22-13 |
| HTTP 요청 메시지의 내용        | 22-13 |
| HTTP 응답 메시지 만들기        | 22-13 |
| 응답 헤더 채우기              | 22-14 |
| 응답 상태 표시               | 22-14 |
| 클라이언트 작업에 대한 요구 표시     | 22-14 |
| 서버 애플리케이션 설명           | 22-15 |
| 컨텐츠 설명                 | 22-15 |
| 응답 컨텐츠 설정              | 22-15 |
| 응답 보내기                 | 22-16 |
| 응답 메시지 컨텐츠 생성          | 22-16 |
| 페이지 프로듀서 컴포넌트 사용       | 22-16 |
| HTML 템플릿               | 22-16 |
| HTML 템플릿 지정            | 22-17 |
| HTML-transparent 태그 변환 | 22-18 |
| 작업 항목에서 페이지            |       |
| 프로듀서 사용                | 22-18 |
| 페이지 프로듀서 간의 연결         | 22-19 |
| 응답에서 데이터베이스 정보 사용      | 22-20 |
| 웹 모듈에 세션 추가            | 22-20 |
| HTML로 데이터베이스 정보 표시     | 22-20 |
| 데이터셋 페이지 프로듀서 사용       | 22-20 |
| 테이블 프로듀서 사용            | 22-21 |
| 테이블 속성 지정              | 22-21 |
| 행 속성 지정                | 22-21 |
| 열 지정                   | 22-22 |
| HTML 문서에 테이블 포함시키기     | 22-22 |
| 데이터셋 테이블 프로듀서 설정       | 22-22 |
| 쿼리 테이블 프로듀서 설정         | 22-22 |
| 서버 애플리케이션 디버깅          | 22-23 |
| CGI 애플리케이션 디버깅         | 22-23 |
| 공유 객체로 디버깅             | 22-23 |
| Apache DSO 애플리케이션 디버깅  | 22-23 |
| DSO 지원을 위한 Apache      |       |
| 애플리케이션 컴파일             | 22-24 |
| 애플리케이션 디버깅             | 22-24 |

### 23 장

|             |      |
|-------------|------|
| 소켓 작업       | 23-1 |
| 서비스 구현      | 23-1 |
| 서비스 프로토콜 이해 | 23-2 |
| 애플리케이션과 통신  | 23-2 |
| 서비스와 포트     | 23-2 |



|                              |       |
|------------------------------|-------|
| 소켓 연결의 타입 . . . . .          | 23-3  |
| 클라이언트 연결 . . . . .           | 23-3  |
| 리스닝 (listening) 연결 . . . . . | 23-3  |
| 서버 연결 . . . . .              | 23-3  |
| 소켓 설명 . . . . .              | 23-4  |
| 호스트 설명 . . . . .             | 23-4  |
| 호스트 이름 또는 IP 주소 선택 . . . . . | 23-5  |
| 포트 사용 . . . . .              | 23-5  |
| 소켓 컴포넌트 사용 . . . . .         | 23-5  |
| 클라이언트 소켓 사용 . . . . .        | 23-6  |
| 대상 서버 지정 . . . . .           | 23-6  |
| 연결 구성 . . . . .              | 23-6  |
| 연결에 대한 정보 얻기 . . . . .       | 23-6  |
| 연결 끊기 . . . . .              | 23-6  |
| 서버 소켓 사용 . . . . .           | 23-6  |
| 포트 지정 . . . . .              | 23-7  |
| 클라이언트 요청 리스닝 . . . . .       | 23-7  |
| 클라이언트에 연결 . . . . .          | 23-7  |
| 서버 연결 끊기 . . . . .           | 23-7  |
| 소켓 이벤트에 응답 . . . . .         | 23-7  |
| 오류 이벤트 . . . . .             | 23-8  |
| 클라이언트 이벤트 . . . . .          | 23-8  |
| 서버 이벤트 . . . . .             | 23-8  |
| 리스닝 시의 이벤트 . . . . .         | 23-8  |
| 클라이언트 연결 시의 이벤트 . . . . .    | 23-9  |
| 소켓 연결을 통해 읽기 및 쓰기 . . . . .  | 23-9  |
| 비차단 연결 . . . . .             | 23-9  |
| 읽기 및 쓰기 이벤트 . . . . .        | 23-10 |
| 차단 연결 . . . . .              | 23-10 |

## 4 부

### 사용자 지정 컴포넌트 생성

#### 24 장

|                           |      |
|---------------------------|------|
| 컴포넌트 생성 개요 . . . . .      | 24-1 |
| 크로스 플랫폼용 컴포넌트             |      |
| 라이브러리 (CLX) . . . . .     | 24-1 |
| 컴포넌트 및 클래스 . . . . .      | 24-2 |
| 컴포넌트 생성 방법 . . . . .      | 24-3 |
| 기존 컨트롤 수정 . . . . .       | 24-3 |
| 컨트롤 생성 . . . . .          | 24-4 |
| 그래픽 컨트롤 작성 . . . . .      | 24-4 |
| 컨트롤 하위 분류 . . . . .       | 24-4 |
| 넌비주얼 컴포넌트 생성 . . . . .    | 24-5 |
| 컴포넌트 작성 시 고려 사항 . . . . . | 24-5 |
| 종속성 제거 . . . . .          | 24-5 |
| 속성, 메소드 및 이벤트 . . . . .   | 24-6 |
| 속성 . . . . .              | 24-6 |

|                            |       |
|----------------------------|-------|
| 이벤트 . . . . .              | 24-6  |
| 메소드 . . . . .              | 24-7  |
| 그래픽 캡슐화 . . . . .          | 24-7  |
| 등록 . . . . .               | 24-7  |
| 새 컴포넌트 생성 . . . . .        | 24-8  |
| 컴포넌트 마법사 사용 . . . . .      | 24-8  |
| 수동으로 컴포넌트 생성 . . . . .     | 24-10 |
| 유닛 파일 생성 . . . . .         | 24-10 |
| 컴포넌트 파생 . . . . .          | 24-11 |
| 컴포넌트 등록 . . . . .          | 24-11 |
| 설치되지 않은 컴포넌트 테스트 . . . . . | 24-12 |
| 설치된 컴포넌트 테스트 . . . . .     | 24-14 |

#### 25 장

|                              |      |
|------------------------------|------|
| 컴포넌트 작성자를 위한                 |      |
| 객체 지향 프로그래밍 . . . . .        | 25-1 |
| 새 클래스 정의 . . . . .           | 25-1 |
| 새 클래스 파생 . . . . .           | 25-2 |
| 반복을 피하기 위한 클래스               |      |
| 기본값 변경 . . . . .             | 25-2 |
| 클래스에 새 기능 추가 . . . . .       | 25-2 |
| 새 컴포넌트 클래스 선언 . . . . .      | 25-3 |
| 조상, 자손 및 클래스 계층 구조 . . . . . | 25-3 |
| 액세스 제어 . . . . .             | 25-4 |
| 구현 세부 사항 숨기기 . . . . .       | 25-4 |
| 컴포넌트 작성자의 인터페이스 정의 . . . . . | 25-5 |
| 런타임 인터페이스 정의 . . . . .       | 25-6 |
| 디자인 타임 인터페이스 정의 . . . . .    | 25-6 |
| 메소드 디스패칭 . . . . .           | 25-7 |
| 정적 메소드 . . . . .             | 25-7 |
| 가상 메소드 . . . . .             | 25-8 |
| 메소드 오버라이드 . . . . .          | 25-8 |
| 추상 클래스 멤버 . . . . .          | 25-9 |
| 클래스 및 포인터 . . . . .          | 25-9 |

#### 26 장

|                       |      |
|-----------------------|------|
| 속성 생성 . . . . .       | 26-1 |
| 속성을 생성하는 이유 . . . . . | 26-1 |
| 속성 타입 . . . . .       | 26-2 |
| 상속된 속성 게시 . . . . .   | 26-3 |
| 속성 정의 . . . . .       | 26-3 |
| 속성 선언 . . . . .       | 26-4 |
| 내부 데이터 저장소 . . . . .  | 26-4 |
| 직접 액세스 . . . . .      | 26-5 |
| 액세스 메소드 . . . . .     | 26-5 |
| read 메소드 . . . . .    | 26-6 |
| write 메소드 . . . . .   | 26-7 |

|                           |       |
|---------------------------|-------|
| 기본 속성 값 . . . . .         | 26-7  |
| 기본값을 갖지 않도록 지정 . . . . .  | 26-8  |
| 배열 속성 생성 . . . . .        | 26-8  |
| 하위 컴포넌트의 속성 생성 . . . . .  | 26-9  |
| 인터페이스의 속성 생성 . . . . .    | 26-10 |
| 속성 저장 및 로드 . . . . .      | 26-11 |
| 저장 및 로드 메커니즘 사용 . . . . . | 26-12 |
| 기본값 지정 . . . . .          | 26-12 |
| 저장할 대상 결정 . . . . .       | 26-13 |
| 로드 후 초기화 . . . . .        | 26-13 |
| published 가 아닌 속성         |       |
| 저장 및 로드 . . . . .         | 26-13 |
| 속성 값을 저장 및 로드하는           |       |
| 메소드 생성 . . . . .          | 26-14 |
| DefineProperties 메소드      |       |
| 오버라이드 . . . . .           | 26-14 |

## 27 장

|                               |      |
|-------------------------------|------|
| 이벤트 생성 . . . . .              | 27-1 |
| 이벤트의 개념 . . . . .             | 27-1 |
| 이벤트는 메소드 포인터입니다 . . . . .     | 27-2 |
| 이벤트는 속성입니다 . . . . .          | 27-3 |
| 이벤트 타입은 메소드                   |      |
| 포인터 타입입니다. . . . .            | 27-3 |
| 이벤트 핸들러 타입은 프로시저입니다 . . . . . | 27-3 |
| 이벤트 핸들러는 옵션입니다 . . . . .      | 27-4 |
| 표준 이벤트 구현 . . . . .           | 27-4 |
| 표준 이벤트 식별 . . . . .           | 27-5 |
| 모든 컨트롤에 대한 표준 이벤트 . . . . .   | 27-5 |
| widget 기반 컨트롤에 대한             |      |
| 표준 이벤트 . . . . .              | 27-5 |
| 이벤트 보이기 . . . . .             | 27-5 |
| 표준 이벤트 처리 변경 . . . . .        | 27-6 |
| 사용자 고유 이벤트 정의 . . . . .       | 27-7 |
| 이벤트 트리거 . . . . .             | 27-7 |
| 두 종류의 이벤트. . . . .            | 27-7 |
| 핸들러 타입 정의 . . . . .           | 27-8 |
| 간단한 공지 . . . . .              | 27-8 |
| 이벤트 특정 핸들러. . . . .           | 27-8 |
| 핸들러의 정보 반환. . . . .           | 27-8 |
| 이벤트 선언 . . . . .              | 27-8 |
| "On" 으로 시작하는 이벤트 이름 . . . . . | 27-9 |
| 이벤트 호출 . . . . .              | 27-9 |

## 28 장

|                   |      |
|-------------------|------|
| 메소드 생성 . . . . .  | 28-1 |
| 종속성 피하기 . . . . . | 28-1 |

|                               |      |
|-------------------------------|------|
| 메소드 이름 지정 . . . . .           | 28-2 |
| 메소드 보호 . . . . .              | 28-2 |
| public 이어야 하는 메소드 . . . . .   | 28-3 |
| protected 여야 하는 메소드 . . . . . | 28-3 |
| 추상 메소드 . . . . .              | 28-3 |
| 가상 메소드 생성 . . . . .           | 28-4 |
| 메소드 선언 . . . . .              | 28-4 |

## 29 장

|                                 |      |
|---------------------------------|------|
| 컴포넌트에서 그래픽 사용 . . . . .         | 29-1 |
| 캔버스 사용 . . . . .                | 29-1 |
| 그림 작업 . . . . .                 | 29-2 |
| 그림, 그래픽 또는 캔버스 사용 . . . . .     | 29-2 |
| 그래픽 로드 및 저장 . . . . .           | 29-3 |
| 오프스크린(off-screen) 비트맵 . . . . . | 29-3 |
| 오프스크린 비트맵 생성 및 관리 . . . . .     | 29-4 |
| 비트맵 이미지 복사 . . . . .            | 29-4 |
| 변화에 대한 반응 . . . . .             | 29-5 |

## 30 장

|                                      |       |
|--------------------------------------|-------|
| 디자인 타임 시 컴포넌트 사용 . . . . .           | 30-1  |
| 컴포넌트 등록 . . . . .                    | 30-1  |
| Register 프로시저 선언 . . . . .           | 30-2  |
| Register 프로시저 작성 . . . . .           | 30-2  |
| 컴포넌트 지정 . . . . .                    | 30-3  |
| 팔레트 페이지 지정 . . . . .                 | 30-3  |
| RegisterComponents 함수 사용 . . . . .   | 30-3  |
| 팔레트 비트맵 추가 . . . . .                 | 30-4  |
| 속성 편집기 추가 . . . . .                  | 30-4  |
| 속성 편집기 클래스 파생 . . . . .              | 30-4  |
| 텍스트로 속성 편집 . . . . .                 | 30-5  |
| 속성 값 표시 . . . . .                    | 30-6  |
| 속성 값 설정 . . . . .                    | 30-6  |
| 전체 속성 편집 . . . . .                   | 30-7  |
| 편집기 속성 지정 . . . . .                  | 30-8  |
| 속성 편집기 등록 . . . . .                  | 30-9  |
| 속성 범주 . . . . .                      | 30-10 |
| 한 번에 하나의 속성 등록 . . . . .             | 30-10 |
| 한 번에 여러 속성 등록 . . . . .              | 30-11 |
| 속성 범주 지정 . . . . .                   | 30-12 |
| IsPropertyInCategory 함수 사용 . . . . . | 30-12 |
| 컴포넌트 편집기 추가 . . . . .                | 30-13 |
| 컨텍스트 메뉴에 항목 추가 . . . . .             | 30-13 |
| 메뉴 항목 지정 . . . . .                   | 30-14 |
| 명령 구현 . . . . .                      | 30-14 |
| 더블 클릭 행동 변경 . . . . .                | 30-15 |
| 클립보드 형식 추가 . . . . .                 | 30-15 |

|                          |       |
|--------------------------|-------|
| 컴포넌트 편집기 등록 . . . . .    | 30-16 |
| 컴포넌트를 패키지로 컴파일 . . . . . | 30-16 |

## 31 장

|                        |      |
|------------------------|------|
| 기존 컴포넌트 수정 . . . . .   | 31-1 |
| 컴포넌트 생성 및 등록 . . . . . | 31-1 |
| 컴포넌트 클래스 수정 . . . . .  | 31-2 |
| 생성자 오버라이드 . . . . .    | 31-2 |
| 새 기본 속성 값 지정 . . . . . | 31-3 |

## 32 장

|                           |      |
|---------------------------|------|
| 그래픽 컴포넌트 생성 . . . . .     | 32-1 |
| 컴포넌트 생성 및 등록 . . . . .    | 32-1 |
| 상속된 속성 게시 . . . . .       | 32-2 |
| 그래픽 기능 추가 . . . . .       | 32-3 |
| 그릴 대상 결정 . . . . .        | 32-3 |
| 속성 타입 선언 . . . . .        | 32-3 |
| 속성 선언 . . . . .           | 32-4 |
| 구현 메소드 작성 . . . . .       | 32-4 |
| 생성자 및 소멸자 오버라이드 . . . . . | 32-4 |
| 기본 속성 값 변경 . . . . .      | 32-4 |
| 펜과 브러시 게시 . . . . .       | 32-5 |
| 클래스 필드 선언 . . . . .       | 32-5 |
| 액세스 속성 선언 . . . . .       | 32-6 |
| 소유된 클래스 초기화 . . . . .     | 32-7 |
| 소유된 클래스의 속성 설정 . . . . .  | 32-7 |
| 컴포넌트 이미지 그리기 . . . . .    | 32-8 |
| 정교한 도형 그리기 . . . . .      | 32-9 |

## 33 장

|                           |       |
|---------------------------|-------|
| 그리드 사용자 지정 . . . . .      | 33-1  |
| 컴포넌트 생성 및 등록 . . . . .    | 33-1  |
| 상속된 속성 게시 . . . . .       | 33-2  |
| 초기 값 변경 . . . . .         | 33-3  |
| 셀 크기 조정 . . . . .         | 33-4  |
| 셀 채우기 . . . . .           | 33-4  |
| 날짜 추적 . . . . .           | 33-5  |
| 내부 날짜 저장 . . . . .        | 33-5  |
| 일, 월, 연도 액세스 . . . . .    | 33-6  |
| 일(day) 수 생성 . . . . .     | 33-7  |
| 오늘 날짜 선택 . . . . .        | 33-9  |
| 월과 연도 탐색 . . . . .        | 33-9  |
| 일(day) 탐색 . . . . .       | 33-10 |
| 선택 이동 . . . . .           | 33-10 |
| OnChange 이벤트 제공 . . . . . | 33-11 |
| 빈 셀 배제 . . . . .          | 33-12 |

## 34 장

|                                |       |
|--------------------------------|-------|
| data aware 컨트롤 만들기 . . . . .   | 34-1  |
| 데이터 찾아보기 컨트롤 생성 . . . . .      | 34-1  |
| 컴포넌트 생성 및 등록 . . . . .         | 34-2  |
| 읽기 전용 컨트롤 만들기 . . . . .        | 34-3  |
| ReadOnly 속성 추가 . . . . .       | 34-3  |
| 필요한 업데이트 허용 . . . . .          | 34-4  |
| 데이터 연결 추가 . . . . .            | 34-5  |
| 클래스 필드 선언 . . . . .            | 34-5  |
| 액세스 속성 선언 . . . . .            | 34-5  |
| 액세스 속성 선언의 예제 . . . . .        | 34-6  |
| 데이터 연결 초기화 . . . . .           | 34-6  |
| 데이터 변경 내용에 응답 . . . . .        | 34-7  |
| 데이터 편집 컨트롤 생성 . . . . .        | 34-8  |
| FReadOnly의 기본값 변경 . . . . .    | 34-8  |
| 마우스 다운 및 키 다운 이벤트 처리 . . . . . | 34-9  |
| 마우스 다운 이벤트에 응답 . . . . .       | 34-9  |
| 키 다운 이벤트에 응답 . . . . .         | 34-10 |
| 필드 데이터 연결 클래스 업데이트 . . . . .   | 34-11 |
| Change 메소드 수정 . . . . .        | 34-12 |
| 데이터셋 업데이트 . . . . .            | 34-12 |

## 색인

I-1



# 표

|      |                                              |       |       |                                                  |       |
|------|----------------------------------------------|-------|-------|--------------------------------------------------|-------|
| 1.1  | 글꼴과 기호 . . . . .                             | 1-2   | 10.9  | 시스템 이벤트에 응답하기 위한<br>TWidgetControl protected 메소드 | 10-19 |
| 3.1  | 중요한 기본 클래스 . . . . .                         | 3-12  | 10.10 | 유사한 데이터 액세스 컴포넌트 . . . . .                       | 10-23 |
| 3.2  | 그래픽 컨트롤 . . . . .                            | 3-16  | 10.11 | 캐시된 업데이트를 위한 속성,<br>메소드 및 이벤트 . . . . .          | 10-27 |
| 3.3  | 컴포넌트 팔레트 페이지 . . . . .                       | 3-18  | 11.1  | 컴파일된 패키지 파일 . . . . .                            | 11-2  |
| 3.4  | 텍스트 컨트롤 속성 . . . . .                         | 3-19  | 11.2  | 디자인 타임 패키지 . . . . .                             | 11-4  |
| 3.5  | 목록의 생성 및 관리를 위한<br>컴포넌트 . . . . .            | 3-32  | 11.3  | 패키지 특정 컴파일러 지시어 . . . . .                        | 11-10 |
| 4.1  | RTL 예외 . . . . .                             | 4-9   | 11.4  | 패키지 특정 명령줄 컴파일러 스위치                              | 11-11 |
| 4.2  | 오브젝트 파스칼 문자 타입 . . . . .                     | 4-25  | 11.5  | 컴파일된 패키지 파일 . . . . .                            | 11-12 |
| 4.3  | 문자열 비교 루틴 . . . . .                          | 4-29  | 12.1  | 문자열 길이 추정 . . . . .                              | 12-4  |
| 4.4  | 대소문자 변환 루틴 . . . . .                         | 4-30  | 12.2  | Resbind 옵션 . . . . .                             | 12-6  |
| 4.5  | 문자열 수정 루틴 . . . . .                          | 4-30  | 15.1  | 데이터 컨트롤 . . . . .                                | 15-2  |
| 4.6  | 하위 문자열 루틴 . . . . .                          | 4-30  | 15.2  | 열 속성 . . . . .                                   | 15-19 |
| 4.7  | 문자열 처리 루틴 . . . . .                          | 4-31  | 15.3  | 확장된 TColumn Title 속성 . . . . .                   | 15-20 |
| 4.8  | 문자열에 대한 컴파일러 지시어 . . . . .                   | 4-35  | 15.4  | 복합 필드가 나타나는 방식에<br>영향을 주는 속성 . . . . .           | 15-22 |
| 4.9  | 속성 상속 및 값 . . . . .                          | 4-38  | 15.5  | Expanded TDBGrid Options 속성 . . . . .            | 15-23 |
| 4.10 | 파일 I/O 를 위한 파일 형식 . . . . .                  | 4-40  | 15.6  | 그리드 컨트롤 이벤트 . . . . .                            | 15-25 |
| 4.11 | 개방 모드 . . . . .                              | 4-41  | 15.7  | TDBNavigator 버튼 . . . . .                        | 15-26 |
| 4.12 | 공유 모드 . . . . .                              | 4-42  | 16.1  | 데이터셋 State 속성의 값 . . . . .                       | 16-3  |
| 4.13 | 각 개방 모드에서 사용할 수 있는<br>공유 모드 . . . . .        | 4-42  | 16.2  | 데이터셋의 탐색 메소드 . . . . .                           | 16-9  |
| 5.1  | 라이브러리용 컴파일러<br>지시어 ( 계속 ) . . . . .          | 5-4   | 16.3  | 데이터셋의 탐색 속성 . . . . .                            | 16-9  |
| 6.1  | 작업 범주 ( 계속 ) . . . . .                       | 6-18  | 16.4  | 필터에 사용할 수 있는 연산자 . . . . .                       | 16-18 |
| 6.2  | 예제 캡션 및 파생된 이름 . . . . .                     | 6-23  | 16.5  | FilterOptions 값 . . . . .                        | 16-20 |
| 6.3  | 메뉴 디자이너 컨텍스트 메뉴 명령 . . . . .                 | 6-29  | 16.6  | 필터링된 데이터셋의 탐색 메소드 . . . . .                      | 16-21 |
| 6.4  | 스피드 버튼의 모양 설정 . . . . .                      | 6-34  | 16.7  | 데이터 삽입, 업데이트 및 삭제<br>위한 데이터셋 메소드 . . . . .       | 16-22 |
| 6.5  | 툴 버튼 모양 설정 . . . . .                         | 6-37  | 16.8  | 전체 레코드에 적용되는 메소드 . . . . .                       | 16-26 |
| 7.1  | 선택한 텍스트의 속성 . . . . .                        | 7-3   | 16.9  | 데이터셋 이벤트 . . . . .                               | 16-27 |
| 7.2  | Fixed 및 variable owner-draw<br>스타일 . . . . . | 7-8   | 17.1  | 데이터 표시에 영향을 주는<br>TFloatField 속성 . . . . .       | 17-1  |
| 8.1  | 그래픽 객체 형식 . . . . .                          | 8-3   | 17.2  | 특수한 영구적 필드 종류 . . . . .                          | 17-6  |
| 8.2  | 캔버스 객체의 일반적인 속성 . . . . .                    | 8-3   | 17.3  | 필드 컴포넌트 속성 . . . . .                             | 17-11 |
| 8.3  | 캔버스 객체의 일반적인 속성 . . . . .                    | 8-4   | 17.4  | 필드 컴포넌트 서식 루틴 . . . . .                          | 17-13 |
| 8.4  | 마우스 이벤트 . . . . .                            | 8-23  | 17.5  | 필드 컴포넌트 이벤트 . . . . .                            | 17-14 |
| 8.5  | 마우스 이벤트 매개변수 . . . . .                       | 8-24  | 17.6  | 선택된 필드 컴포넌트 메소드 . . . . .                        | 17-15 |
| 9.1  | WaitFor 반환 값 . . . . .                       | 9-10  | 17.7  | 특수한 변환 결과 . . . . .                              | 17-18 |
| 10.1 | 포팅 기법 . . . . .                              | 10-1  | 17.8  | 객체 필드 컴포넌트의 타입 . . . . .                         | 17-21 |
| 10.2 | CLX 부분 . . . . .                             | 10-4  | 17.9  | 공통 객체 필드 자손 속성 . . . . .                         | 17-21 |
| 10.3 | 변경되었거나 다른 기능 . . . . .                       | 10-7  | 18.1  | 테이블을 나열하는<br>메타데이터 테이블의 열 . . . . .              | 18-17 |
| 10.4 | Kylix 와 Delphi 의 유닛 . . . . .                | 10-8  | 18.2  | 내장 프로시저를 나열하는<br>메타데이터 테이블의 열 . . . . .          | 18-18 |
| 10.5 | Kylix 에는 있고 Delphi 에는 없는<br>유닛 . . . . .     | 10-9  | 18.3  | 필드를 나열하는 메타데이터<br>테이블의 열 . . . . .               | 18-18 |
| 10.6 | Kylix 에는 있고 Kylix 에는 없는<br>유닛 . . . . .      | 10-10 |       |                                                  |       |
| 10.7 | Linux 환경의 차이점 . . . . .                      | 10-12 |       |                                                  |       |
| 10.8 | 공통된 Linux 디렉토리 . . . . .                     | 10-14 |       |                                                  |       |

|      |                     |       |
|------|---------------------|-------|
| 18.4 | 인덱스를 나열하는 메타데이터     |       |
|      | 테이블의 열              | 18-19 |
| 18.5 | 매개변수를 나열하는 메타데이터    |       |
|      | 테이블의 열              | 18-20 |
| 20.1 | 인덱스 기반 검색 메소드       | 20-3  |
| 20.2 | 유지 관리되는 집계의 요약 연산자  | 20-21 |
| 20.3 | 데이터 요청 처리를 위한 클라이언트 |       |
|      | 데이터셋 속성 및 메소드       | 20-29 |
| 21.1 | AppServer 인터페이스 멤버  | 21-3  |
| 21.2 | 프로바이더 옵션            | 21-5  |
| 21.3 | UpdateStatus 값      | 21-8  |
| 21.4 | UpdateMode 값        | 21-9  |
| 21.5 | ProviderFlags 값     | 21-10 |

|      |                        |       |
|------|------------------------|-------|
| 22.1 | 웹 서버 애플리케이션 컴포넌트       | 22-4  |
| 22.2 | MethodType 값           | 22-9  |
| 24.1 | 컴포넌트 생성 기본 작업          | 24-3  |
| 25.1 | 객체 내의 가시성 수준           | 25-4  |
| 26.1 | Object Inspector에서 속성이 |       |
|      | 표시되는 방법                | 26-2  |
| 29.1 | 캔버스 기능 요약              | 29-2  |
| 29.2 | 이미지 복사 메소드             | 29-4  |
| 30.1 | 이미 정의된 속성 편집기 타입       | 30-5  |
| 30.2 | 속성 값을 읽고 쓰기 위한 메소드     | 30-6  |
| 30.3 | 속성 편집기 특성 플래그          | 30-8  |
| 30.4 | 속성 범주                  | 30-12 |

## 그림

|      |                                |       |
|------|--------------------------------|-------|
| 3.1  | CLX 구성                         | 3-2   |
| 3.2  | 단순한 폼                          | 3-6   |
| 3.3  | 단순화된 계층 구조                     | 3-12  |
| 3.4  | 트랙 표시줄 컴포넌트의                   |       |
|      | 세 가지 보기                        | 3-20  |
| 3.5  | 진행 표시줄                         | 3-29  |
| 6.1  | data-aware 컨트롤과 데이터            |       |
|      | 소스 컴포넌트를 갖는 프레임                | 6-13  |
| 6.2  | 메뉴 용어                          | 6-20  |
| 6.3  | MainMenu 컴포넌트                  |       |
|      | 및 PopupMenu 컴포넌트               | 6-21  |
| 6.4  | 팝업 메뉴의 메뉴 디자이너                 | 6-21  |
| 6.5  | 메인 메뉴의 메뉴 디자이너                 | 6-22  |
| 6.6  | 중첩 메뉴 구조                       | 6-26  |
| 6.7  | Select Menu 대화 상자              | 6-29  |
| 6.8  | 메뉴에 대한 예제 Insert Template      |       |
|      | 대화 상자                          | 6-30  |
| 6.9  | 메뉴에 대한 Save Template           |       |
|      | 대화 상자                          | 6-31  |
| 14.1 | 일반 데이터베이스 아키텍처                 | 14-5  |
| 14.2 | 파일 기반 데이터베이스                   |       |
|      | 애플리케이션의 아키텍처                   | 14-7  |
| 14.3 | 단방향 데이터베이스                     |       |
|      | 애플리케이션의 아키텍처                   | 14-8  |
| 14.4 | 클라이언트 데이터셋과                    |       |
|      | 단방향 데이터셋을 조합하는                 |       |
|      | 아키텍처                           | 14-10 |
| 14.5 | 내부 단방향 데이터셋이 있는                |       |
|      | 클라이언트 데이터셋을 사용하는               |       |
|      | 아키텍처                           | 14-11 |
| 14.6 | 다계층 데이터베이스 아키텍처                | 14-12 |
| 15.1 | TDBGrid 컨트롤                    | 15-15 |
| 15.2 | TDBNavigator 컨트롤의 버튼           | 15-26 |
| 16.1 | Inactive 및 Browse 상태의 관계       | 16-5  |
| 16.2 | 찾아보기와 데이터셋 상태의 관계              | 16-6  |
| 22.1 | URL(Uniform Resource Locator)의 |       |
|      | 각 부분                           | 22-2  |
| 22.2 | 서버 애플리케이션의 구조                  | 22-7  |
| 24.1 | CLX 클래스 계층                     | 24-2  |

# 1

## 서문

*개발자 안내서*에서는 Linux 운영 체제용 텔파이인 Kylix를 사용하여 데이터베이스 애플리케이션 제작, 사용자 지정 컴포넌트 작성, 인터넷 웹 서버 애플리케이션 만들기 등과 같은 중급 및 고급 개발 항목에 대해 설명합니다. *개발자 안내서*를 사용하는 사람은 Linux 사용에 익숙하고 기본적인 프로그래밍 기술을 이해하고 있다고 가정합니다. Kylix 프로그래밍과 통합 개발 환경(IDE)에 대한 소개는 온라인 도움말과 *입문서* 설명서를 참조하십시오.

## 설명서 내용

---

이 설명서에는 다음과 같은 내용이 들어 있습니다.

- **1부 "Kylix 프로그래밍"**에서는 범용 Kylix 애플리케이션을 제작하는 방법에 대해 설명합니다. 이 부분에서는 Kylix 애플리케이션에서 사용할 수 있는 프로그래밍 기술에 대한 세부적인 내용을 제공합니다. 예를 들어, 문자열 및 텍스트 처리 같은 사용자 인터페이스 개발을 단순화하는 일반적인 볼랜드 크로스 플랫폼용 컴포넌트 라이브러리(CLX) 객체를 사용하는 방법에 대해 설명합니다. 또한 그래픽, 컨트롤, 오류와 예외 처리 및 국제적인 애플리케이션 작성 등의 작업에 대한 장들이 있습니다. 배포에 대한 장에서는 애플리케이션 사용자들에게 애플리케이션 배포와 관련된 작업에 대해 설명합니다.
- **2부 "데이터베이스 애플리케이션 개발"**에서는 데이터베이스 툴과 컴포넌트를 사용하여 데이터베이스 애플리케이션을 제작하는 방법에 대해 설명합니다. Kylix에서는 DBExpress를 사용하여 SQL 서버 데이터베이스에 액세스하도록 합니다. Kylix 버전은 특정 데이터베이스에 연결하는 DBExpress 드라이버 집합과 함께 출시됩니다. 다른 데이터베이스에 연결하는 추가 DBExpress 드라이버는 별도로 구입할 수 있습니다.
- **3부 "분산 애플리케이션 개발"**에서는 로컬 네트워크에 배포되는 애플리케이션을 만드는 방법에 대해 설명합니다. 여기에는 Apache 및 CGI 애플리케이션과 같은 웹 서버 애플리케이션들이 있습니다. 분산 애플리케이션의 낮은 지원에 대하여, 이 단원에서는 TCP/IP 및 관련 프로토콜을 사용하여 통신 세부 사항을 처리하는 소켓 컴포넌트

트를 사용하는 방법에 대해 설명합니다. 소켓과 웹 서버 애플리케이션을 지원하는 컴포넌트를 모든 Kylix 에디션에서 사용할 수 있는 것은 아닙니다.

- 4부 "사용자 지정 컴포넌트 생성"에서는 자신의 컴포넌트를 디자인하고 구현하는 방법 및 컴포넌트를 Kylix 개발 환경의 컴포넌트 팔레트에서 사용하는 방법에 대해 설명합니다. 컴포넌트는 디자인 타임에 처리하려는 거의 모든 프로그램 요소를 나타낼 수 있습니다. 사용자 지정 컴포넌트의 구현은 CLX 클래스 라이브러리의 기존 클래스 타입에서 새 클래스를 파생함을 의미합니다.

## 설명서 규칙

이 설명서는 특별한 텍스트를 표시하기 위하여 표 1.1에서 설명한 글꼴과 기호를 사용합니다.

표 1.1 글꼴과 기호

| 글꼴 또는 기호              | 의미                                                                                           |
|-----------------------|----------------------------------------------------------------------------------------------|
| <b>Monospace type</b> | 고정 폭 텍스트는 텍스트를 화면이나 오브젝트 파스칼 코드에 나타나는 대로 표시합니다. 사용자가 입력해야 하는 내용 역시 이 글꼴로 나타냅니다.              |
| [ ]                   | 텍스트나 구문 목록에서 대괄호는 옵션 항목을 묶습니다. 이러한 종류의 텍스트는 그대로 입력하면 안 됩니다.                                  |
| <b>Boldface</b>       | 텍스트나 코드 목록에서 굵게 쓰여진 글자는 오브젝트 파스칼 키워드 또는 컴파일러 옵션을 나타냅니다.                                      |
| <i>Italics</i>        | 텍스트에서 이탤릭체 글자는 변수 또는 타입 이름 같은 오브젝트 파스칼 식별자를 나타냅니다. 이탤릭체는 새로운 용어 같은 일부 글자를 강조하기 위해 사용하기도 합니다. |
| <i>Keycaps</i>        | 이 글꼴은 키보드에 있는 특정 키를 나타냅니다. 예를 들면, 다음과 같습니다. "메뉴를 종료하려면 <i>Esc</i> 를 누릅니다."                   |

## 개발자 지원 서비스

Borland는 다양한 개발자 커뮤니티의 요구에 맞출 수 있는 다양한 지원 옵션을 제공합니다. Kylix 지원에 대해 알아보려면

<http://www.borland.com/devsupport>를 참조하십시오. 추가적인 Kylix 기술 정보 문서 및 FAQ에 대한 대답도 이 웹 사이트에서 얻을 수 있습니다.

웹 사이트를 통해 Kylix 개발자들이 정보, 팁, 기술 등을 교환하는 많은 뉴스그룹에 액세스할 수 있습니다.

또한 <http://community.borland.com>에 있는 볼랜드 커뮤니티 사이트를 참조하십시오. 여기서 Kylix에 대한 많은 정보, 기사, 코드 예제, 최신 뉴스에 액세스할 수 있습니다.

## 인쇄된 설명서 주문

추가 설명서 주문에 대한 내용은 [shop.borland.com](http://shop.borland.com) 웹 사이트를 참조하십시오.





# Kylix 프로그래밍

"Kylix 프로그래밍"의 각 장들은 카일릭스 애플리케이션 만들기에 필요한 개념과 기술에 대해 소개합니다. 또한 *개발자 안내서*의 이후 단원에서 설명될 개념에 대해서도 소개합니다.



## Kylix를 사용한 애플리케이션 개발

Borland Kylix는 32비트 애플리케이션의 신속한 개발을 위한 객체 지향적인 비주얼 프로그래밍 환경입니다. Kylix를 사용하면, 코딩을 최소화하면서 효율이 높은 크로스 플랫폼 애플리케이션을 만들 수 있습니다.

Kylix는 애플리케이션, 폼 템플릿, 프로그래밍 마법사를 비롯한 볼랜드 크로스 플랫폼 컴포넌트 라이브러리(C LX)라는 포괄적인 클래스 라이브러리와 신속한 애플리케이션 개발(RAD) 디자인 툴을 제공합니다. Kylix는 진정한 객체 지향 프로그래밍을 지원합니다. 애플리케이션 개발 중에 사용할 수 있는 많은 유용한 객체들이 클래스 라이브러리에 들어 있습니다.

이 장에서는 Kylix 개발 환경에 대해 간단히 설명합니다. 이 설명서의 나머지 부분에서는 범용 데이터베이스와 인터넷 및 인트라넷 애플리케이션의 개발에 대한 기술적인 세부 사항을 제공하고, 고유한 컴포넌트를 작성하는 데 필요한 정보가 들어 있습니다.

### 통합 개발 환경

Kylix를 시작하면, IDE라는 통합 개발 환경 내에 바로 놓이게 됩니다. 이 환경은 애플리케이션을 디자인, 개발, 테스트, 디버그, 배포하는 데 필요한 모든 툴을 제공합니다.

Kylix의 개발 환경에는 비주얼 폼 디자이너, Object Inspector, 컴포넌트 팔레트, Project Manager, 소스 코드 에디터, 디버거 등이 들어 있습니다. 폼 디자이너에 있는 객체의 비주얼 표현에서부터 객체의 초기 런타임 상태를 편집하는 Object Inspector 또는 객체의 실행 로직을 편집하는 소스 코드 에디터까지 자유로이 이동할 수 있습니다. Object Inspector에서 이벤트 핸들러의 이름 같은 코드 관련 속성을 변경하면 해당 소스 코드는 자동으로 바뀝니다. 게다가, 폼 클래스 선언에서 이벤트 핸들러 메소드의 이름을 재지정하는 것 같은 소스 코드의 변경 사항은 즉시 Object Inspector에 반영됩니다.

IDE는 디자인에서 배포까지 제품 라이프 사이클의 단계를 통한 애플리케이션 개발을 지원합니다. IDE에 있는 툴을 사용하면 프로토타입을 신속하게 만들 수 있고 개발 기간을 단축할 수 있습니다.

개발 환경에 대한 더욱 완벽한 개요는 이 제품과 함께 들어 있는 *입문서*에 있습니다. 또한, 온라인 도움말 시스템에서는 모든 메뉴, 대화 상자, 창에 대한 도움말을 제공합니다.

## 애플리케이션 디자인

---

Kylix에는 애플리케이션 디자인을 시작하는 데 필요한 모든 툴이 들어 있습니다. 툴의 종류는 다음과 같습니다.

- 애플리케이션용 UI를 디자인하는 *폼*이라고 하는 빈 창
- 재사용 가능한 많은 객체가 들어 있는 광범위한 클래스 라이브러리(C LX)
- 객체 특성을 조사하고 변경하기 위한 Object Inspector
- 원본으로 사용하는 프로그램 로직에 직접 액세스할 수 있는 코드 에디터
- 하나 이상의 프로젝트를 구성하는 파일을 관리하기 위한 Project Manager
- IDE에서 애플리케이션 개발을 지원하는 통합 디버거 같은 그 밖의 많은 툴
- 컴파일러, 링커 및 기타 다른 유틸리티가 들어 있는 명령줄(command-line) 툴

Kylix를 사용하여 범용 유틸리티에서부터 복잡한 데이터 액세스 프로그램 또는 분산 애플리케이션에 이르기까지 모든 종류의 32비트 애플리케이션을 디자인할 수 있습니다. Kylix의 데이터베이스 툴과 data-aware 컴포넌트를 사용하면 강력한 데스크탑 데이터베이스 및 클라이언트/서버 애플리케이션을 신속하게 개발할 수 있습니다. Kylix의 data-aware 컨트롤을 사용하면, 애플리케이션을 디자인하면서 생생한 데이터를 볼 수 있고 데이터베이스 쿼리의 결과와 애플리케이션 인터페이스에 대한 변경 사항을 즉시 볼 수 있습니다.

클래스 라이브러리에 제공된 많은 객체는 IDE의 컴포넌트 팔레트에서 액세스할 수 있습니다. 컴포넌트 팔레트는 폼에 놓을 수 있는 모든 비주얼 컨트롤과 논비주얼(nonvisual) 컨트롤을 보여줍니다. 각 탭에는 기능별로 그룹화된 컴포넌트가 들어 있습니다. 규칙에 따라 클래스 라이브러리에 있는 객체의 이름은 *TStatusBar*와 같이 T로 시작합니다.

Kylix에서 혁신적인 것 중의 하나는 오브젝트 파스칼을 사용하여 직접 컴포넌트를 만들 수 있다는 것입니다. 제공되는 대부분의 컴포넌트는 오브젝트 파스칼로 작성된 것입니다. 작성한 컴포넌트를 컴포넌트 팔레트에 추가할 수 있고 필요한 경우 새 탭을 포함시켜 팔레트를 사용자 지정하여 사용할 수 있습니다.

5장 "애플리케이션과 공유 객체 구축"에서는 Kylix가 다른 종류의 애플리케이션을 지원하는 것에 대해 설명합니다.

## 애플리케이션 개발

---

애플리케이션의 사용자 인터페이스를 시각적으로 디자인할 때 Kylix는 애플리케이션을 지원하는 원본으로 사용하는 오브젝트 파스칼 코드를 생성합니다. 컴포넌트와 폼의 속성을 선택하고 수정하면, 이러한 변경 사항의 결과가 소스 코드에 자동으로 나타나며 그 반대의 경우도 마찬가지입니다. 기본 제공된 코드 에디터를 비롯한 모든 텍스트 에디터를 사용하여 소스 파일을 직접 수정할 수 있습니다. 변경 사항은 비주얼 환경에도 바로 반영됩니다. 이 기능을 "two-way 툴"이라고 합니다.

## 프로젝트 생성

Kylix의 애플리케이션 개발은 모두 프로젝트 중심으로 이루어집니다. Kylix에서 애플리케이션을 만들면 프로젝트를 만드는 것입니다. 프로젝트는 애플리케이션을 구성하는 파일의 모음입니다. 이러한 파일들 중 일부는 디자인 시 생성됩니다. 그 외 다른 파일들은 프로젝트 소스 코드를 컴파일할 때 자동으로 생성됩니다.

Project Manager라는 프로젝트 관리 툴에서 프로젝트 내용을 볼 수 있습니다. Project Manager는 유닛 이름, 유닛에 들어 있는 폼(하나 있는 경우)을 계층적인 보기로 나열하고, 프로젝트에서 파일로 가는 경로를 보여줍니다. 이러한 많은 파일을 직접 편집할 수도 있지만, Kylix에서 비주얼 툴을 사용하는 것이 때때로 쉽고 신뢰성이 있습니다.

그룹 파일은 프로젝트 계층의 맨 위에 있습니다. 여러 프로젝트를 프로젝트 그룹으로 결합할 수 있습니다. 이것을 통해 Project Manager에서 한 번에 둘 이상의 프로젝트를 열 수 있습니다. 프로젝트 그룹을 사용하여 함께 기능하는 애플리케이션 이나 다계층 애플리케이션의 일부와 같은 관련 프로젝트에 대해 구성하고 작업할 수 있습니다. 한 가지 프로젝트에서만 작업 중인 경우에는 애플리케이션을 만드는 데 프로젝트 그룹 파일이 필요하지 않습니다.

개별 프로젝트, 파일 및 연관된 옵션에 대해 설명하는 프로젝트 파일은 .dpr이라는 확장자를 가집니다. 프로젝트 파일에는 애플리케이션 또는 공유 객체 빌드에 대한 설명이 들어 있습니다. Project Manager를 사용하여 파일을 추가하고 제거하면, 프로젝트 파일이 업데이트됩니다. 폼, 애플리케이션, 컴파일러 등과 같은 프로젝트의 여러 측면에 대한 탭을 가진 Project Options 대화 상자를 사용하여 프로젝트 옵션을 지정합니다. 이러한 프로젝트 옵션은 프로젝트와 함께 프로젝트 파일에 저장됩니다.

유닛과 폼은 Kylix 애플리케이션의 기본 작성 블록입니다. 프로젝트는 프로젝트 디렉토리 트리 밖에 있는 것들을 포함한 모든 기존 폼과 유닛 파일을 공유할 수 있습니다. 여기에는 독립형 루틴으로 작성된 사용자 지정 프로시저와 함수가 들어 있습니다.

프로젝트에 공유 파일을 추가하는 경우 파일이 현재 프로젝트 디렉토리에 복사되는 것이 아니고 현재 위치에 남아 있다는 것을 알 수 있습니다. 현재 프로젝트에 공유 파일을 추가하면 파일 이름과 경로가 프로젝트 파일의 **uses** 절에 기록됩니다. 프로젝트에 유닛을 추가하면 Kylix는 이름과 경로의 기록을 자동으로 처리합니다.

프로젝트를 컴파일할 때 프로젝트를 구성하는 파일들이 있는 위치는 중요하지 않습니다. 컴파일러는 공유 파일을 프로젝트 자체에 의해 생성된 파일과 동일하게 취급합니다.

## 코드 편집

Kylix 코드 에디터는 완전한 기능을 갖춘 아스키 에디터입니다. 비주얼 프로그래밍 환경을 사용하는 경우 폼은 새 프로젝트의 일부로 자동적으로 표시됩니다. 객체를 폼에 놓고 Object Inspector에서 객체가 작동하는 방식을 수정함으로써 애플리케이션 인터페이스의 디자인을 시작할 수 있습니다. 그러나 객체에 대한 이벤트 핸들러 작성 같은 다른 프로그래밍 작업은 코드를 입력해야 합니다.

폼의 내용, 폼의 모든 속성, 컴포넌트 및 컴포넌트의 속성은 코드 에디터에서 텍스트로 볼 수 있고 편집할 수 있습니다. 코드 에디터에서 생성된 코드를 조정할 수 있고 코드를 입력하여 에디터 내에서 더 많은 컴포넌트를 추가할 수 있습니다. 에디터에 코드를 입력

할 때, 컴파일러는 계속 변경된 사항을 스캔하고 폼을 새 레이아웃으로 업데이트합니다. 그런 다음 폼으로 돌아가서 에디터에서 변경 사항을 보고 테스트하며 계속 폼을 조정할 수 있습니다.

Kylix 코드 생성 및 속성 스트리밍 시스템은 검사에 대해 완전히 열려 있습니다. 최종 실행 파일에 들어 있는 모든 것에 대한 소스 코드(모든 CLX 객체, RTL(Run-Time Library) 소스, 모든 Kylix 프로젝트 파일)를 코드 에디터에서 보고 편집할 수 있습니다.

## 애플리케이션 컴파일

---

폼에서 애플리케이션 인터페이스의 디자인, 추가 코드 작성을 원하는 대로 마치면, IDE 또는 명령줄에서 프로젝트를 컴파일할 수 있습니다.

모든 프로젝트는 단일 분산 가능 실행 파일을 대상으로 가집니다. 다양한 개발 단계에서 다음과 같이 애플리케이션을 컴파일, 빌드, 실행함으로써 애플리케이션을 보거나 테스트할 수 있습니다.

- 컴파일 시, 마지막 컴파일 이후에 변경된 유닛만 다시 컴파일됩니다.
- 빌드 시, 마지막 컴파일 이후의 변경 여부에 관계 없이 프로젝트의 모든 유닛이 컴파일됩니다. 이 기술은 어떤 파일이 변경되었는지 정확하게 모르거나 모든 파일이 현재 동기화되었는지 확인하려 할 때 유용합니다. 또한 전역 컴파일러 지시어를 변경했을 때 Build를 사용하여 모든 코드가 적절한 상태로 컴파일되었는지 확인하는 것도 중요합니다. 프로젝트를 컴파일하지 않고도 소스 코드의 유효성을 테스트할 수도 있습니다.
- 실행 시, 컴파일한 후에 애플리케이션을 실행합니다. 마지막 컴파일 이후에 소스 코드를 수정한 경우에 컴파일러는 변경된 모듈을 다시 컴파일하고 애플리케이션을 다시 연결합니다.

여러 프로젝트를 함께 그룹화한 경우 단일 프로젝트 그룹에서 한 번에 모든 프로젝트를 컴파일하거나 빌드할 수 있습니다. Project Manager에서 선택한 프로젝트 그룹과 함께 Project|Compile All Projects 또는 Project|Build All Projects를 선택합니다.

## 애플리케이션 디버깅

---

Kylix는 애플리케이션의 오류를 찾아서 고치는 데 도움을 주는 통합 디버거를 제공합니다. 통합 디버거를 통해 프로그램 실행을 제어하고, 데이터 구조의 변수 값과 항목을 모니터링할 수 있고, 디버깅 중에 데이터 값을 수정할 수 있습니다.

통합 디버거는 런타임 오류와 논리 오류 모두 추적할 수 있습니다. 특정 프로그램 위치에서 실행하고, 변수 값, 호출 스택의 함수 값, 프로그램 출력 값을 봄으로써 프로그램의 동작 방식을 모니터링할 수 있고 디자인한 대로 동작하지 않는 영역을 찾을 수 있습니다. 디버거에 대해서는 온라인 도움말에서 설명합니다.

또한 예외 처리를 사용하여 오류를 인식하고, 위치를 찾아내고, 처리할 수 있습니다. Kylix에서 예외는 다른 클래스에서 T로 시작하는 것 대신 규칙에 따라 E로 시작하는 것을 제외하면 Kylix에 있는 다른 클래스와 같은 클래스입니다. 예외 처리에 대한 자세한 내용은 4-4 페이지의 "예외 처리"를 참조하십시오.

## 애플리케이션 배포

---

Kylix에는 애플리케이션 배포를 돕는 툴이 들어 있습니다. 분산 애플리케이션 실행에 필요한 모든 파일을 포함하는 애플리케이션용 설치 패키지를 만들 수 있습니다. 배포에 대한 특정 내용은 13장 "애플리케이션 배포"를 참조하십시오.

**참고** 모든 Kylix 버전에 배포 기능이 있는 것은 아닙니다.





# 3

## CLX 사용

이 장에서는 Borland 크로스 플랫폼용 컴포넌트 라이브러리(C LX)에 대한 개요를 제공하고 애플리케이션 개발 중에 사용할 수 있는 객체 중 일부를 설명합니다.

### CLX 이해

---

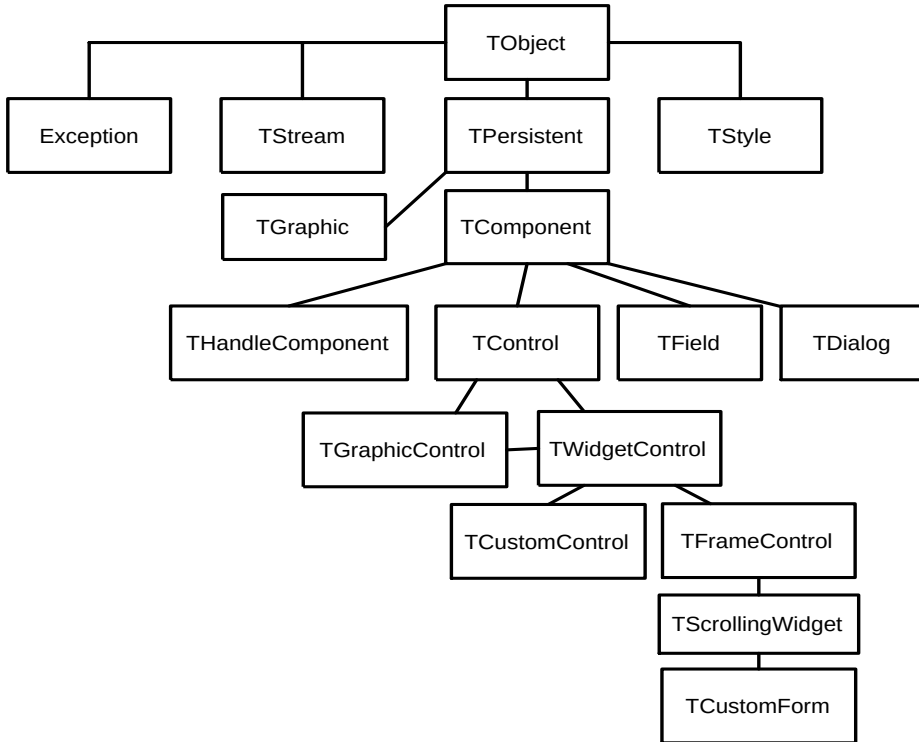
Kylix는 그래픽 사용자 인터페이스(GUI), 인터넷, 데이터베이스 및 서버 애플리케이션의 two-way 비주얼 개발을 위한 컴포넌트 기반 개발 환경입니다. 컴포넌트는 Borland 크로스 플랫폼용 컴포넌트 라이브러리(C LX, "클릭스"로 발음)라는 컴포넌트 라이브러리의 크로스 플랫폼 버전에서 Linux로 구현됩니다. CLX는 원시 Linux 애플리케이션 개발 시간을 대폭 단축하고 Linux 및 Windows 운영 체제용 크로스 플랫폼 개발을 단순화하도록 디자인되었습니다.

객체 지향 프로그래밍은 코드 재사용과 데이터의 기능별 캡슐화를 강조하는 구조적 프로그래밍의 확장입니다. 일단 객체(또는 형식적으로는 클래스)가 생성되면, 프로그래머는 생성된 객체를 다양한 애플리케이션에서 사용할 수 있으므로 개발 시간을 단축하고 생산성을 높일 수 있습니다.

CLX는 여러 객체로 구성된 클래스 라이브러리로서, 이 객체 중 일부는 애플리케이션을 개발할 때 사용하는 컴포넌트 또는 컨트롤입니다.

그림 3.1에서는 몇가지 중요한 객체들을 포함하고 있는 CLX의 구성을 보여줍니다.

그림 3.1 CLX 구성



CLX 객체는 모든 필요한 데이터와 데이터를 수정하는 "메소드"(코드)를 포함하는 활성 엔티티입니다. 데이터는 객체의 필드와 속성에 저장되고, 코드는 필드와 속성 값에 작용하는 메소드들로 구성됩니다. 각 객체는 "클래스"로 선언됩니다. 사용자가 오브젝트 파스칼로 개발한 객체를 비롯한 모든 CLX 객체는 조상 객체인 *TObject*의 자손입니다.

객체의 서브셋은 컴포넌트입니다. CLX의 컴포넌트는 추상 클래스 *TComponent*의 자손입니다. 컴포넌트는 폼 또는 데이터 모듈에 두고 디자인 타임에 처리할 수 있는 객체입니다. 컴포넌트는 컴포넌트 팔레트에 나타납니다. 코드를 작성하지 않고 컴포넌트의 속성을 지정할 수 있습니다. 모든 CLX 컴포넌트는 *TComponent* 객체의 자손입니다.

컴포넌트는 다음과 같은 이유 때문에 진정한 객체 지향 프로그래밍(OOP)으로 구현된 객체라고 볼 수 있습니다

- 데이터와 데이터 액세스 함수의 일부 집합을 캡슐화합니다
- 조상 객체로부터 데이터와 행동을 상속받습니다
- 다형성이라는 개념을 통해, 공통 조상에서 파생된 다른 객체와 상호 교환할 수 있습니다.

대부분의 컴포넌트와 달리 객체는 컴포넌트 팔레트에 나타나지 않습니다. 그 대신, 기본 인스턴스 변수는 객체의 유닛에 선언되거나 사용자가 자체적으로 선언해야 합니다.

런타임에 화면에 나타나는 *TForm*과 *TSpeedButton* 같은 비주얼 컴포넌트는 *컨트롤*이라고 하며 *TControl*의 자손입니다. 컨트롤은 런타임에 사용자가 볼 수 있는 특별한 종류의 컴포넌트입니다. 컨트롤은 컴포넌트의 서브셋입니다. 애플리케이션이 실행 중일 때 사용자 인터페이스에서 컨트롤을 볼 수 있습니다. 모든 컨트롤은 *Height*와 *Width* 같은 *비주얼한 속성*을 지정하는 *공통의 속성*을 가집니다. 컨트롤이 공통적으로 가지는 속성, 메소드 및 이벤트는 모두 *TControl*에서 상속됩니다.

CLX에는 폼에 가져다 놓음으로써 사용자의 프로그램에 추가할 수 있는 *년비주얼* (nonvisual) 객체도 많이 있습니다. 예를 들어, 데이터베이스에 연결하는 애플리케이션을 작성하는 경우에 폼에 *TDataSource* 컴포넌트를 놓을 수 있습니다. *TDataSource*는 보이지 않지만 타임시 나타나지 않는 아이콘에 의해서 폼에 나타납니다. Object Inspector에서 *TDataSource*의 속성과 이벤트를 비주얼 컨트롤의 경우와 마찬가지로 처리할 수 있습니다.

프로그래밍 도중 온라인 도움말을 사용하여 CLX의 모든 객체에 대한 자세한 참조 자료에 액세스할 수 있습니다.

## 속성, 메소드 및 이벤트

CLX는 오브젝트 파스칼로 개발된 객체의 계층구조이며, 애플리케이션을 신속하게 개발할 수 있는 Kylix IDE에 연결됩니다. CLX는 속성, 메소드 및 이벤트에 기반을 두고 있습니다. CLX는 데이터 멤버(속성), 데이터에 작동하는 함수(메소드), 클래스의 사용자와 상호 작용하는 방법(이벤트)을 정의합니다.

### 속성

속성은 객체의 비주얼한 행동 또는 작동에 영향을 주는 객체의 특성입니다. 예를 들어, *Visible* 속성은 객체가 애플리케이션 인터페이스에서 보이는지 여부를 결정합니다. 속성을 잘 디자인하면 컴포넌트를 유지 관리하기가 쉽고, 다른 사용자도 쉽게 사용할 수 있습니다.

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.
- Object Inspector에서 속성에 액세스하여 객체 값을 비주얼하게 수정할 수 있습니다. 디자인 타임에 속성을 설정하여 코드를 직접 작성하는 노력을 줄이고 코드 유지 관리를 쉽게 합니다.
- 데이터가 캡슐화되어 있으므로 데이터가 보호되며 실제 객체에 *private*으로 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.
- 속성에 액세스하는 동안 이벤트를 트리거하거나 다른 데이터를 수정하는 로직을 구현할 수 있습니다. 예를 들어, 특정 속성 값을 변경하려면 다른 속성을 수정해야 합니다. 속성에 대해 만든 메소드를 변경할 수 있습니다.

- 속성은 가상(virtual)일 수 있습니다.
- 속성은 단일 객체에 국한되지 않습니다. 한 객체에 대한 속성을 변경하는 것은 여러 객체들에 영향을 미칠 수 있습니다. 예를 들어, 하나의 라디오 버튼에 *Checked* 속성을 설정하는 것은 그룹의 모든 라디오 버튼에 영향을 줍니다.

## 메소드

메소드는 클래스에 연결된 프로시저입니다. 메소드는 객체의 행동을 정의합니다. 클래스 메소드는 모든 *public*, *protected*, *private* 속성 및 클래스의 데이터 멤버에 액세스할 수 있고 일반적으로 클래스 메소드를 멤버 함수라고 합니다.

## 이벤트

이벤트는 프로그램에 의해 탐지된 동작이나 발생한 사건입니다. 대부분의 최근 애플리케이션은 이벤트에 응답하도록 디자인되기 때문에 이벤트 방식이라고 합니다. 프로그램에서 프로그래머는 사용자가 다음에 수행할 일련의 동작을 예측할 수 없습니다. 사용자는 메뉴 항목을 선택하거나 버튼을 누르거나 또는 텍스트 일부를 표시할 수도 있습니다. 항상 똑같은 순서로 실행되는 코드를 작성하기보다는 흥미 있는 이벤트를 처리하는 코드를 작성할 수 있습니다.

이벤트 호출 방법에 상관 없이, Kylix에서는 이벤트를 처리하는 코드를 작성했는지를 확인합니다. 사용자가 이벤트를 처리하는 코드를 작성한 경우 이 코드가 실행되지만, 그렇지 않은 경우에 기본 이벤트 처리 행동이 실행됩니다.

이벤트에는 메뉴 항목 선택 같은 widget 이벤트 또는 타이머나 키 입력 작업 같은 시스템 이벤트가 있습니다.

## widget 이벤트

widget 이벤트는 사용자가 widget를 사용할 때 발생하는 동작입니다. widget 이벤트는 처리를 위해 CLX 컴포넌트로 전달된 신호를 생성합니다. CLX 컴포넌트는 전달된 신호에 대해 설치된 연결된 이벤트 핸들러를 가집니다. widget 이벤트의 예제로는 편집 컨트롤에서 텍스트를 변경했을 때 발생하는 *OnChange*, 메뉴에서 메뉴 항목을 선택했을 때 발생하는 *OnHightlighted*, 메모 컨트롤에서 *Enter*를 눌렀을 때 발생하는 *OnReturn Pressed* 등이 있습니다. 이러한 이벤트들은 특정 widget에 항상 연결되어 있고 이러한 widget 내에 정의됩니다.

## 시스템 이벤트

시스템 이벤트는 운영 체제에서 생성하는 이벤트입니다. 시스템 이벤트의 예제로는 *TTimer* 컴포넌트에서 정해진 시간이 경과되면 발생시키는 이벤트인 *OnTimer* 이벤트, 컴포넌트를 생성할 때 발생하는 *OnCreate* 이벤트, 컴포넌트나 widget를 다시 그릴 때 발생하는 *OnPaint* 이벤트, 키보드에서 키를 누르면 발생하는 *OnKeyPress* 이벤트 등이 있습니다. 원하는 대로 처리되지 않는 경우, 애플리케이션 프로그래머가 직접 응답해야 하는 이벤트입니다.

## 객체의 개념

객체나 클래스는 단일 유닛에서 *데이터*와 *데이터에 대한 작업*을 캡슐화하는 데이터 타입입니다. 객체 지향 프로그래밍 개념이 일반화되기 전에는 데이터와 연산(함수)은 별도의 요소로 취급되었습니다.

오브젝트 파스칼의 *레코드*를 이해하면 객체를 이해할 수 있습니다. C 랭귀지의 *구조체*와 비슷한 개념인 레코드는 데이터가 포함된 필드로 구성되는데, 여기서 각 필드는 다양한 타입을 가집니다. 레코드를 사용하여 다양한 데이터 요소의 컬렉션을 쉽게 참조할 수 있습니다.

또한 객체는 데이터 요소의 컬렉션입니다. 그러나 레코드와 달리 객체에는 데이터에 작동하는 프로시저와 함수가 포함됩니다. 이러한 프로시저와 함수를 *메소드*라고 합니다.

객체의 데이터 요소는 속성을 통해 액세스됩니다. CLX 객체의 속성은 코드를 작성하지 않고 디자인 타임에 변경할 수 있는 값을 가집니다. 속성 값을 런타임에 변경하려면 약간의 코드만 작성하면 됩니다.

단일 유닛의 데이터와 기능의 조합을 *캡슐화*라고 합니다. 캡슐화(Encapsulation)이외에도 객체 지향 프로그래밍은 *상속*(Inheritance)과 *다형성*(Polymorphism)이라는 특징이 있습니다. 상속은 객체가 *조상*이라고 부르는 객체로부터 기능을 파생시키는 것을 의미합니다. 파생된 객체는 상속된 행동을 수정할 수 있습니다. 다형성은 동일한 조상에서 파생된 다양한 객체들이 동일한 메소드와 속성 인터페이스를 지원한다는 것을 의미하므로, 종종 상호 교환 가능하다고 할 수 있습니다.

## Kylix 객체 검사

새로운 프로젝트를 만들 때, Kylix는 사용자가 지정할 새 폼을 표시합니다. 코드 에디터에서, Kylix는 폼의 새 클래스 타입을 선언하고 새로운 폼 인스턴스를 생성하는 코드를 만듭니다. 생성된 코드는 다음과 같습니다.

```
unit Unit1;
interface
uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
type
  TForm1 = class(TForm) { The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end; { The type declaration of the form ends here }
var
  Form1: TForm1;
implementation { Beginning of implementation part }
{$R *.DFM}
end.{ End of implementation part and unit}
```

새 클래스 타입은 *TForm1*이고 마찬가지로 클래스인 *TForm* 타입에서 파생됩니다.

클래스는 데이터 필드를 가지고 있다는 점에서 레코드와 유사하지만, 클래스는 객체의 데이터에 작동하는 코드인 메소드도 포함합니다. 지금까지, 폼에 컴포넌트(새 객체의 필드)를 추가하지 않고 이벤트 핸들러(새 객체의 메소드)를 만들지 않았기 때문에,

*TForm1*은 필드나 메소드가 없는 것처럼 보입니다. 타입 선언에서 보이지 않지만 *TForm1*은 상속된 필드와 메소드를 포함합니다.

다음 변수 선언은 새 타입 *TForm1*의 *Form1*이라는 변수를 선언합니다.

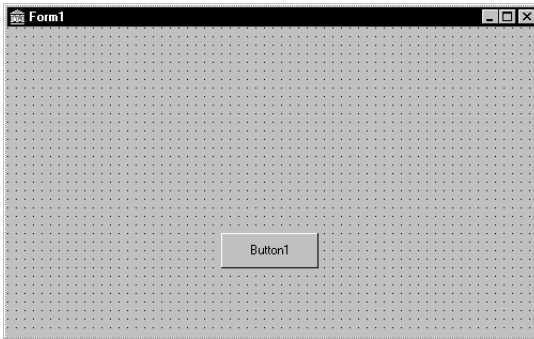
```
var
  Form1:TForm1;
```

*Form1*은 클래스 타입 *TForm1*의 인스턴스 또는 객체를 나타냅니다. 클래스 타입의 인스턴스를 두 개 이상 선언할 수 있습니다. 각 인스턴스는 자체 데이터를 유지하지만 모든 인스턴스는 동일한 코드를 사용하여 메소드를 실행합니다.

폼에 컴포넌트를 추가하거나 코드를 작성하지 않아도 컴파일하고 실행할 수 있는 완전한 Kylix 애플리케이션이 이미 있습니다. 이 애플리케이션을 실행하면 빈 폼이 표시됩니다.

이 폼에 버튼 컴포넌트를 추가하고 사용자가 버튼을 누르면 폼의 색상을 변경하는 *OnClick* 이벤트 핸들러를 작성한다고 가정합니다. 그 결과는 다음과 같습니다.

**그림 3.2** 단순한 폼



사용자가 버튼을 누를 때 폼의 색상이 녹색으로 바뀝니다. 다음은 버튼의 *OnClick* 이벤트에 대한 이벤트 핸들러 코드입니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  Form1.Color := clGreen;
end;
```

객체는 데이터 필드와 같은 다른 객체를 포함할 수 있습니다. 폼에 컴포넌트를 둘 때마다 새 필드가 폼의 타입 선언에 나타납니다. 위에 설명된 애플리케이션을 만들면 코드 에디터에서 다음 코드를 볼 수 있습니다.

```
unit Unit1;

interface

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TForm1 = class(TForm)
    Button1: TButton;{ New data field }
    procedure Button1Click(Sender: TObject);{ New method declaration }
  private
```

```

    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1:TForm1;

implementation
{$R *.xfm}

procedure TForm1.Button1Click(Sender: TObject);{ The code of the new method }
begin
  Form1.Color := clGreen;
end;

end.

```

*TForm1*은 폼에 추가한 버튼에 해당하는 *Button1* 필드를 가집니다. *TButton*은 클래스 타입이므로 *Button1*은 객체를 참조합니다.

Kylix에서 작성한 모든 이벤트 핸들러는 폼 객체의 메소드입니다. 이벤트 핸들러를 만들 때마다 메소드가 폼 객체 타입으로 선언됩니다. 이제 *TForm1* 타입은 *TForm1* 타입 선언에서 선언된 새 메소드인 *Button1Click* 프로시저를 포함합니다. *Button1Click* 메소드를 구현하는 코드는 유닛의 **implementation** 부분에 나타납니다.

## 컴포넌트 이름 변경

컴포넌트의 이름을 변경하려면 항상 Object Inspector를 사용해야 합니다. 예를 들어, 폼의 이름을 기본 *Form1*에서 *ColorBox*와 같은 보다 구체적인 이름으로 변경하고자 한다고 가정합니다. Object Inspector에서 폼의 *Name* 속성을 변경하면, 폼의 .xfrm 파일(보통은 수동으로 편집하지 않음)과 Kylix가 생성하는 오브젝트 파스칼 소스 코드에 다음과 같이 새 이름이 자동으로 반영됩니다.

```

unit Unit1;

interface

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TColorBox = class(TForm){ Changed Form1 to TColorBox }
    Button1:TButton;
    procedure Button1Click(Sender:TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ColorBox: TColorBox;{ Changed Form1 to ColorBox }

implementation
{$R *.xfm}

procedure TColorBox.Button1Click(Sender:TObject);
begin

```

```
Form1.Color := clGreen; { The reference to Form1 didn't change! }
end;
end.
```

버튼에 대한 *OnClick* 이벤트 핸들러의 코드가 변경되지 않았다는 것을 염두에 두십시오. 사용자가 코드를 작성했기 때문에 그 코드를 사용자가 업데이트하고 폼에 대한 참조를 수정해야 합니다.

```
procedure TColorBox.Button1Click(Sender:TObject);
begin
    ColorBox.Color := clGreen;
end;
```

## 객체에서 데이터와 코드 상속

3-5 페이지에서 설명한 *TForm1* 객체는 간단한 예제입니다. *TForm1*은 하나의 필드(*Button1*)와 하나의 메소드(*Button1Click*)를 포함하지만 속성은 포함하지 않습니다. 이제 폼을 보여주거나 숨기거나 크기를 조정할 수 있고, 표준 테두리 아이콘을 추가하거나 삭제할 수 있으며, MDI(Multiple Document Interface) 애플리케이션의 일부가 되도록 폼을 설정할 수 있습니다. 폼이 CLX 컴포넌트 *TForm*의 모든 속성과 메소드를 상속했기 때문에 이러한 모든 작업을 수행할 수 있습니다. 프로젝트에 새 폼을 추가할 때, *TForm*을 시작한 다음에 컴포넌트를 추가하고 속성 값을 변경하고 이벤트 핸들러를 작성하여 폼을 사용자 지정합니다. 객체를 사용자 지정하려면 먼저 기존 객체로부터 새 객체를 파생시킵니다. 프로젝트에 새 폼을 추가하는 경우 Kylix는 자동으로 *TForm* 타입으로부터 새 폼을 파생시킵니다.

```
TForm1 = class(TForm)
```

파생된 객체는 파생시킨 객체의 모든 속성, 이벤트 및 메소드를 상속받습니다. 파생된 객체는 *자손*이라고 하고, 이 자손을 파생시킨 객체는 *조상*이라고 합니다. 온라인 도움말에서 *TForm*을 조회하면 *TForm*이 조상으로부터 상속받은 것을 포함한 *TForm*의 속성, 이벤트 및 메소드의 목록을 볼 수 있습니다. 객체는 직계 조상을 단 하나만 가질 수 있지만, 직계 자손은 많이 가질 수 있습니다.

## 유효 범위(scope) 및 한정자

*유효 범위*는 객체의 필드, 속성 및 메소드의 액세스 가능성을 결정합니다. 객체 내에서 선언된 모든 멤버는 객체 및 객체의 자손에서 사용할 수 있습니다. 메소드의 구현 코드가 객체 선언 밖에 나타나더라도, 메소드는 객체 선언 내에서 선언되기 때문에 여전히 객체의 유효 범위 내에 있습니다.

메소드가 선언된 객체의 속성, 메소드 또는 필드를 참조하는 메소드를 구현하는 코드를 작성할 때, 식별자를 객체 이름으로 시작할 필요는 없습니다. 예를 들어, 새 폼에 버튼을 놓는 경우 다음과 같이 버튼의 *OnClick* 이벤트에 대한 이벤트 핸들러를 작성할 수 있습니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
    Color := clFuchsia;
    Button1.Color := clLime;
```



```
end;
```

첫 번째 문장은 다음과 동일합니다.

```
Form1.Color := clFuchsia
```

*Button1Click* 메소드가 *TForm1*의 일부이기 때문에 *Form1*을 사용하여 *Color*를 한정할 필요가 없고, 따라서 메소드 몸체의 식별자는 메소드가 호출되는 *TForm1* 인스턴스의 유효 범위 내에 있게 됩니다. 반면에 두 번째 문장은 이벤트 핸들러가 선언된 폼이 아닌 버튼 객체의 색상을 참조하므로 한정되어야 합니다.

Kylix는 각 폼에 대하여 별도의 유닛(소스 코드) 파일을 만듭니다. 다른 폼의 유닛 파일로부터 특정 폼의 컴포넌트에 액세스하려면, 다음과 같이 컴포넌트 이름을 한정해야 합니다.

```
Form2.Edit1.Color := clLime;
```

동일한 방법으로 다른 폼에 있는 컴포넌트의 메소드에 액세스할 수 있습니다. 예를 들어,

```
Form2.Edit1.Clear;
```

*Form1*의 유닛 파일에서 *Form2*의 컴포넌트에 액세스하려면 *Form2*의 유닛을 *Form1* 유닛의 **uses** 절에 추가해야 합니다.

객체의 유효 범위는 객체의 자손으로 확장됩니다. 그러나, 자손 객체 내에서 필드, 속성 및 메소드를 재선언할 수 있습니다. 그러한 재선언은 상속된 멤버를 숨기거나 오버라이드합니다.

유효 범위, 상속 및 **uses** 절에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

## **private, protected, public, published 선언**

필드, 속성 또는 메소드를 선언할 때, 새 멤버는 키워드 **private**, **protected**, **public** 또는 **published** 중의 하나에 의해 지시되는 *가시성(visibility)*을 가집니다. 멤버의 가시성은 다른 객체와 유닛에 대한 멤버의 액세스 가능성을 결정합니다.

- **private** 멤버는 멤버가 선언된 유닛 내에서만 멤버에 액세스할 수 있습니다. **private** 멤버를 가끔 클래스 내에서 사용하여 **public** 또는 **published**로 선언된 메소드와 속성을 구현합니다.
- **protected** 멤버는 멤버의 클래스가 선언된 유닛과 자손 클래스 내에서 자손 클래스의 유닛에 상관 없이 액세스할 수 있습니다.
- **public** 멤버는 멤버가 속하는 객체에 액세스할 수 있는 어디에서라도, 즉 클래스가 선언된 유닛과 그 유닛을 참조하는 모든 유닛에서 액세스할 수 있습니다.
- **published** 멤버는 **public** 멤버와 동일한 가시성을 갖지만, 컴파일러는 **published** 멤버에 대해 런타임 타입 정보를 생성합니다. **published** 속성은 디자인 타임시 Object Inspector에 나타납니다.

가시성에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

## 객체 변수 사용

변수들이 동일한 타입이거나 할당 호환이 있는 경우, 특정 객체 변수를 다른 객체 변수에 할당할 수 있습니다. 특히, 할당하는 변수 타입이 할당되는 변수 타입의 조상인 경우에 객체 변수를 다른 객체 변수에 할당할 수 있습니다. 예를 들면, *TSimpleForm* 타입 선언과 변수 *AForm* 및 *SimpleForm*을 선언하는 변수 선언은 다음과 같습니다.

```
type
  TSimpleForm = class(TForm)
    Button1:TButton;
    Edit1: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AForm: TForm;
  SimpleForm: TSimpleForm;
```

*AForm*은 *TForm* 타입이고, *SimpleForm*은 *TSimpleForm* 타입입니다. *TSimpleForm*은 *TForm*의 자손이므로, 다음과 같은 할당문은 적합합니다.

```
AForm := SimpleForm;
```

버튼의 *OnClick* 이벤트에 대한 이벤트 핸들러를 작성한다고 가정합니다. 버튼을 누르면 *OnClick* 이벤트에 대한 이벤트 핸들러가 호출됩니다. 각 이벤트 핸들러에는 다음과 같이 *TObject* 타입의 *Sender* 매개변수가 있습니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  ...
end;
```

*Sender*는 *TObject* 타입이므로 모든 객체는 *Sender*에 할당될 수 있습니다. *Sender*의 값은 항상 이벤트에 응답하는 컨트롤이나 컴포넌트입니다. 예약어 **is**를 사용하여 이벤트 핸들러를 호출한 컴포넌트나 컨트롤의 타입을 찾기 위해 *Sender*를 테스트할 수 있습니다. 예를 들면 다음과 같습니다.

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

## 객체 생성, 인스턴스화 및 소멸

버튼과 편집 상자 같이 Kylix에서 사용하는 객체 중에서 다수는 디자인 타임과 런타임에 모두 보입니다. 공용 대화 상자 같은 일부 객체는 런타임에만 나타납니다. 그러나, 타이머와 데이터소스 컴포넌트 같은 객체들은 런타임에도 보이지 않습니다.

사용자가 객체를 직접 만들 수도 있습니다. 예를 들어, *Name*, *Title*, *HourlyPayRate* 속성이 포함된 *TEmployee* 객체를 만들 수 있습니다. 그런 다음, *HourlyPayRate*의

데이터를 사용하여 급여를 계산하는 *CalculatePay* 메소드를 추가할 수 있습니다. *TEmployee* 타입 선언은 다음과 같습니다.

```
type
  TEmployee = class(TObject)
  private
    FName:string;
    FTitle:string;
    FHourlyPayRate:Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay:Double;
  end;
```

사용자가 정의한 필드, 속성 및 메소드 외에 *TEmployee*는 *TObject*의 모든 메소드를 상속받습니다. 유닛의 **interface** 또는 **implementation** 부분에 이와 비슷한 타입 선언을 한 다음, 다음과 같이 *TEmployee*가 *TObject*에서 상속받은 *Create* 메소드를 호출하여 새 클래스의 인스턴스를 만들 수 있습니다.

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

*Create* 메소드를 *생성자*라고 합니다. 이 메소드는 새 인스턴스 객체에 메모리를 할당하고 객체에 대한 참조를 반환합니다.

폼에 있는 컴포넌트는 Kylix에 의해서 자동으로 생성되고 소멸됩니다. 그러나 객체를 인스턴스화하는 코드를 직접 작성하는 경우에는 객체의 소멸 역시 직접해야 합니다. 모든 객체는 *소멸자*라고 하는 *Destroy* 메소드를 *TObject*로부터 상속받습니다. 그러나, 객체를 소멸하려면 *TObject*에서 상속받은 *Free* 메소드를 호출해야 합니다. *Free* 메소드가 *Destroy*를 호출하기 전에 **nil** 참조를 확인하기 때문입니다. 예를 들어,

```
Employee.Free
```

이 문장은 *Employee* 객체를 소멸하고 메모리의 할당을 해제합니다.

## 컴포넌트와 소유권

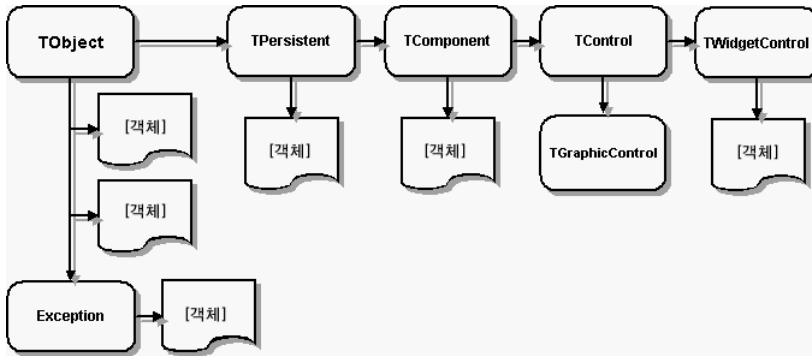
Kylix는 특정 컴포넌트가 다른 컴포넌트 해제를 책임지는 기본 제공 메모리 관리 메커니즘을 가집니다. 컴포넌트를 해제하는 컴포넌트는 해제되는 컴포넌트를 *소유(own)*한다고 말합니다. 소유자(owner)의 메모리가 해제되면 소유자가 소유한 컴포넌트의 메모리는 자동으로 해제됩니다. 컴포넌트의 소유자(*Owner* 속성 값)는 컴포넌트가 생성될 때 생성자에게 전달된 매개변수에 의해 결정됩니다. 기본적으로 폼에 있는 모든 컴포넌트는 폼이 소유하고 폼은 애플리케이션이 소유합니다. 따라서, 애플리케이션이 종료되면, 애플리케이션의 모든 폼과 컴포넌트의 메모리는 해제됩니다.

소유권은 *TComponent*와 이 컴포넌트의 자손에만 적용됩니다. 예를 들면, *TStringList* 또는 *TCollection* 객체를 사용자가 생성하는 경우 생성한 객체가 폼에 연결되어 있더라도 사용자가 객체를 해제해야 합니다.

## CLX 계층의 주요 분기

그림 3.3은 Borland 크로스 플랫폼용 컴포넌트 라이브러리(C LX)에 대한 상속 트리의 주요 분기를 요약해서 보여줍니다.

**그림 3.3** 단순화된 계층 구조



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

**표 3.1** 중요한 기본 클래스

| 클래스                   | 설명                                                                                                                                                                                             |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>TObject</i>        | CLX에서 모든 클래스의 기본 클래스이며 최상위 조상입니다. <i>TObject</i> 는 객체의 인스턴스 생성, 유지 관리 및 소멸과 같은 기본 기능을 수행하는 메소드를 소개하여 CLX의 모든 객체에 공통적인 기본 행동을 캡슐화합니다.                                                           |
| <i>Exception</i>      | 예외와 관련된 모든 클래스의 기본 클래스입니다. <i>예외</i> 는 오류 상태에 대해 일관된 인터페이스를 제공하고, 애플리케이션이 오류 상태를 잘 처리하도록 합니다.                                                                                                  |
| <i>TPersistent</i>    | 속성을 구현하는 모든 객체에 대한 기본 클래스입니다. <i>TPersistent</i> 의 자손 클래스는 데이터를 스트림에 보내는 것을 처리하고 클래스의 할당을 허용합니다.                                                                                               |
| <i>TComponent</i>     | <i>TApplication</i> 과 같은 모든 논비주얼(nonvisual) 컴포넌트에 대한 기본 클래스입니다. <i>TComponent</i> 는 모든 컴포넌트의 공통 조상입니다. 이 클래스를 사용하여 컴포넌트 팔레트에 컴포넌트를 표시할 수 있고, 컴포넌트가 다른 컴포넌트를 소유할 수 있고, 컴포넌트를 폼에서 직접 처리할 수 있습니다. |
| <i>TControl</i>       | 런타임에 보이는 모든 컨트롤에 대한 기본 클래스입니다. <i>TControl</i> 은 모든 비주얼 컴포넌트의 공통 조상이고 위치나 커서 같은 표준 비주얼 컨트롤을 제공합니다. 이 클래스는 또한 마우스 동작에 응답하는 이벤트를 제공합니다.                                                          |
| <i>TWidgetControl</i> | widget라고도 하는 모든 사용자 인터페이스 객체의 기본 클래스입니다. <i>TWidgetControl</i> 의 자손 컨트롤은 키보드 입력을 캡처할 수 있는 widget 기반 컨트롤입니다.                                                                                    |

이어지는 단원에서 주요 CLX 분기에 대해 설명합니다. CLX 객체 계층 구조에 대한 완전한 그림은 이 제품과 함께 들어 있는 CLX Object Hierarchy 월 차트(wall chart)를 참조하십시오.

## TObject 분기

*TObject* 분기에는 *TObject*의 자손인 모든 객체들이 포함되지만 *TPersistent*의 자손 객체는 포함되지 않습니다. 모든 CLX 객체는 생성, 소멸 및 이벤트 핸들러와 같은 기본 행동을 정의하는 메소드를 가지는 추상 클래스인 *TObject*의 자손입니다. CLX 객체의 강력한 기능 대부분은 *TObject*의 메소드로 구현됩니다. *TObject*는 다음을 제공하는 메소드로 CLX의 모든 객체에 공통적인 기본 행동을 캡슐화합니다.

- 객체 인스턴스가 생성되고 소멸될 때 응답하는 기능.
- 클래스 타입, 객체에 대한 인스턴스 정보 및 객체의 published 속성에 대한 런타임 타입 정보(RTTI).

*TObject*는 다른 기본 클래스 중의 하나가 아닌 *TObject*의 자손인 많은 일반(simple) 클래스의 직계 조상입니다. 이 분기의 클래스들은 일시적인 클래스라는 한 가지 중요한 공통 특성이 있습니다. 이것은 클래스가 소멸되기 전에 상태를 저장하는 메소드를 가지지 않는다는 것을 의미하므로, 이 클래스들은 영구적이지 않습니다.

오브젝트 파스칼로 사용자의 클래스를 작성하는 경우, 이 클래스는 *TObject*의 자손이어야 합니다. CLX의 기본 클래스(또는 그 자손 중의 하나)에서 새 클래스를 파생시킴으로써, 사용자는 클래스에 필수적인 기능을 제공하고 CLX를 사용하는 클래스를 만들 수 있습니다.

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

*TObject* 분기에 있는 다른 클래스 타입은 데이터 구조를 캡슐화하는 다음과 같은 클래스들입니다.

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

*TStream*은 이 분기에서 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. 스트림에 대한 자세한 내용을 알려면 3-39 페이지의 "스트림 사용"을 참조하십시오.

## TPersistent 분기

CLX의 이 분기에 있는 객체들은 *TComponent*가 아닌 *TPersistent*의 자손입니다. *TPersistent*는 객체에 영구성을 추가합니다. 영구성은 폼 파일이나 데이터 모듈에 저장되는 것과 메모리에서 검색할 때 폼이나 데이터 모듈로 로드되는 것을 결정합니다.

이 분기에 있는 객체들은 컴포넌트의 속성을 구현합니다. 속성은 소유자(owner)가 있는 경우에 폼과 함께 로드되고 저장됩니다. 소유자는 컴포넌트여야 합니다. 이 분기에는 속성의 소유자를 결정할 수 있는 *GetOwner* 함수가 있습니다.

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

이 분기의 다른 클래스는 다음과 같습니다.

- *TGraphicsObject*는 *TBrush*, *TFont*, *TPen* 등과 같은 그래픽 객체의 추상 기본 클래스입니다.
- *TGraphic*은 *TBitmap* 및 *TPicture* 등의 아이콘과 비트맵과 같은 비주얼 이미지를 저장하고 표시할 수 있는 객체의 추상 기본 클래스입니다.
- *TStrings*은 문자열 목록을 나타내는 객체의 기본 클래스입니다.
- *TClipboard*는 애플리케이션에서 잘라내거나 복사한 텍스트 또는 그래픽을 포함하는 클래스입니다.
- *TCollection*, *TOwnedCollection* 및 *TCollectionItem*은 특별히 정의된 항목의 인덱스된 컬렉션을 유지하는 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 영구적인 객체입니다.

- 컴포넌트 팔레트에 나타나고 폼 디자이너에서 변경할 수 있습니다.
- 다른 컴포넌트를 소유하고 관리합니다.
- 객체를 로드하고 저장합니다.

*TComponent*의 여러 메소드는 디자인 타임 동안 컴포넌트가 작동하는 방법과 그 컴포넌트와 함께 저장하는 정보를 지시합니다. 스트리밍은 이 분기의 CLX에 있습니다. Kylix는 대부분의 스트리밍 작업을 자동으로 처리합니다. 속성이 *published*인 경우 속성은 영구적이고, *published* 속성은 자동으로 스트리밍됩니다.

*TComponent* 클래스는 CLX를 통해 전달되는 소유권의 개념도 있습니다. 다음의 두 속성은 다음의 소유권을 지원합니다. *Owner*와 *Components*. 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 목록에 추가됩니다. 소유된 컴포넌트를 참조하는 컴포넌트 목록을 사용하는 것과 별도로, 이 속성은 또한 소유된 컴포넌트의 자동 소멸을 제공합니다. 컴포넌트의 소유자가 있으면 소유자가 소멸되면 소유된 컴포넌트도 함께 소멸됩니다. 예를 들어, *TForm*은 *TComponent*의 자손이므로, 폼이 소유한 모든 컴포넌트는 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*이 아닌 *TPersistent*인 경우, 스트리밍 시스템은 속성을 통해 사용 가능한 기존 인스턴스를 사용하고 해당 인스턴스에 대한 속성 값을 읽습니다.

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

이 분기에서 찾을 수 있는 클래스 타입은 다음과 같습니다.

- *TMainMenu*는 폼에 대한 메뉴 바와 메뉴 바의 드롭다운 메뉴를 제공하는 클래스입니다.
- *TTimer*는 타이머 함수를 포함하는 클래스입니다.
- *TOpenDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog* 등은 공용으로 사용되는 대화 상자를 제공합니다.
- *TActionList*는 메뉴 항목 및 버튼과 같이 컴포넌트 및 컨트롤과 함께 사용되는 동작의 목록을 유지 관리하는 클래스입니다.
- *TScreen*은 애플리케이션에 의해 인스턴스화된 폼과 데이터 모듈, 활성 폼, 폼의 활성 컨트롤, 화면 크기 및 해상도, 애플리케이션에 사용 가능한 커서 및 글꼴 등을 유지 관리하는 클래스입니다.

*TComponent* 분기는 *THandleComponent*도 포함합니다. 이 클래스는 대화 상자와 메뉴 같이 기본으로 사용한 Qt 객체에 대한 핸들을 요구하는 논비주얼(nonvisual) 컴포넌트의 기본 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.

## TControl 분기

*TControl* 분기는 *TWidgetControl*이 아닌 *TControl*의 자손인 컴포넌트로 구성됩니다. 이 분기의 객체는 애플리케이션 사용자가 런타임 시 보고 처리할 수 있는 비주얼 객체인 컨트롤입니다. 모든 컨트롤은 컨트롤의 위치, 컨트롤의 widget와 연결된 커서, 컨트롤을 그리거나 이동하는 메소드 그리고 마우스 동작에 응답하는 이벤트와 같은 컨트롤의 모양과 관련 있는 공통적인 속성, 메소드 및 이벤트를 가집니다. 컨트롤은 키보드 입력을 받을 수 없습니다.

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

기본 타입의 컨트롤에는 다음 두 가지가 있습니다.

- 자체 widget을 가지는 컨트롤
- "부모" widget을 사용하는 컨트롤

자체 widget을 가지는 컨트롤은 "widget 기반" 컨트롤이라 하며 *TWidgetControl*의 자손입니다. 버튼과 확인란은 이 클래스에 속합니다.

부모 widget을 사용하는 컨트롤은 "그래픽" 컨트롤이라 하며 *TGraphicControl*의 자손입니다. 이미지 컨트롤은 이 클래스에 속합니다. 이러한 타입의 컴포넌트들 간의 주요 차이는 그래픽 컨트롤에 연결된 widget이 없어서 입력 포커스를 받을 수 없거나 다른 컨트롤을 포함할 수 없다는 것입니다. 그래픽 컨트롤에 핸들이 필요없기 때문에, 시스템 리소스에 대한 수요가 감소하고 그래픽 컨트롤을 widget 기반 컨트롤보다 더 빨리 그릴 수 있습니다.

*TGraphicControl* 컨트롤은 스스로 그려지고 다음과 같은 컨트롤을 포함해야만 합니다.

**표 3.2 그래픽 컨트롤**

| 컨트롤              | 설명                                             |
|------------------|------------------------------------------------|
| <i>TImage</i>    | 그래픽 이미지를 표시합니다.                                |
| <i>TBevel</i>    | 빋면이 있는 액자 모양의 윤곽을 나타냅니다.                       |
| <i>TLabel</i>    | 폼에 텍스트를 표시합니다.                                 |
| <i>TPaintBox</i> | 애플리케이션이 이미지를 그리거나 렌더링하는 데 사용할 수 있는 캔버스를 제공합니다. |

여기에는 포커스를 받을 필요가 없는 공통 그리기 루틴 (*Repaint*, *Invalidate* 등)을 포함한다는 것에 유의하십시오.

입력 포커스를 받거나 다른 컨트롤을 포함하지만 *Canvas* 속성과 *Paint* 메소드를 필요로 하는 컨트롤을 만들려면, *TCustomControl*에서 클래스를 파생시킵니다.

## TWidgetControl 분기

*TWidgetControl* 분기에는 *TWidgetControl*의 자손인 모든 컨트롤이 포함됩니다. *TWidgetControl*은 모든 widget 기반 컨트롤 또는 *widgets*에 대한 기본 클래스입니다. widget이라는 용어는 "window"와 "gadget"을 결합한 것입니다. widget은 일반적으로 애플리케이션의 사용자 인터페이스에서 사용하는 컨트롤을 가리키는 용어입니다. widget의 예로는 버튼, 레이블 및 스크롤 막대가 있습니다.

다음은 widget의 특징입니다.

- widget 기반 컨트롤은 연결된 widget을 가집니다.
- widget 기반 컨트롤은 애플리케이션이 실행되는 중에 포커스를 받을 수 있습니다.
- 다른 컨트롤은 데이터를 표시할 수 있지만 사용자는 키보드만 사용하여 widget 기반 컨트롤과 상호 작용할 수 있습니다.
- widget 기반 컨트롤은 다른 컨트롤을 포함할 수 있습니다.
- 다른 컨트롤을 포함하는 컨트롤을 부모라고 합니다. widget 기반 컨트롤만이 하나 이상의 자식 컨트롤의 부모가 될 수 있습니다.



*TWidgetControl*의 자손은 포커스를 받을 수 있는 컨트롤인데 이는 애플리케이션 사용자로부터 키보드 입력을 받을 수 있다는 것을 의미합니다. 이것은 더 많은 표준 이벤트가 컨트롤에 적용된다는 것을 암시합니다.

이 분기에는 *TEdit*, *TListBox*, *TComboBox*, *TPageControl* 등을 포함한 자동으로 그려지는 컨트롤 및 *TDBNavigator*와 같은 Kylix가 그려야 하는 사용자 지정 컨트롤 두 가지 모두를 포함합니다. *TWidgetControl*의 직계 자손은 일반적으로 편집 필드, 콤보 상자, 목록 상자, 페이지 컨트롤 등과 같은 표준 컨트롤을 구현하므로 자신을 그리는 방법을 이미 알고 있습니다. *TCustomControl* 클래스는 핸들 자체를 다시 그리는 기능을 포함하는 표준 컨트롤을 캡슐화하지 않는 컴포넌트에 제공됩니다.

컨트롤 사용에 대한 자세한 내용은 7장 "컨트롤 사용"을 참조하십시오.

## 컴포넌트 사용

컴포넌트 팔레트에 있는 개발 환경 자체에서 많은 비주얼 컴포넌트가 제공됩니다. Kylix의 모든 비주얼 디자인 작업은 폼에서 이루어집니다. Kylix를 열거나 새 프로젝트를 열 때, 빈 폼이 화면에 표시됩니다. 컴포넌트 팔레트에서 컴포넌트를 선택하여 폼에 가져다 놓습니다. 폼에 있는 버튼 및 목록 상자과 같은 비주얼 컴포넌트를 정렬하여 애플리케이션의 사용자 인터페이스의 룩앤필(look and feel)을 디자인합니다. 일단 비주얼 컴포넌트가 폼에 있으면, 컴포넌트의 위치, 크기 및 기타 디자인 타임 속성을 조정할 수 있습니다. Kylix는 원본으로 사용한 프로그래밍 세부 사항을 관리합니다.

또한 폼에 비시각적인 컴포넌트를 두어서 데이터베이스에서 정보를 캡처하고, 계산을 수행하며, 다른 상호 작용을 관리할 수 있습니다. 6장 "애플리케이션 사용자 인터페이스 개발"은 동적으로 모달 폼 생성, 매개변수를 폼에 전달, 폼에서 데이터 검색 등 폼 사용에 대한 세부 사항을 제공합니다.

Kylix 컴포넌트는 기능별로 그룹화되어 컴포넌트 팔레트의 페이지에 정렬되어 있습니다. 예를 들어, 메뉴, 편집 상자 또는 버튼을 만드는 컴포넌트와 같이 공통으로 사용되는 컴포넌트는 컴포넌트 팔레트의 Standard 페이지에 있습니다.

언뜻 보면, Kylix의 컴포넌트는 다른 클래스와 비슷하게 나타납니다. 그러나 Kylix에 있는 컴포넌트와 대부분의 프로그래머가 사용하는 표준 클래스 계층 간에 차이가 있습니다. 몇가지 차이점은 다음과 같습니다.

- 모든 Kylix 컴포넌트는 *TComponent*의 자손입니다.
- 컴포넌트는 기능을 추가하거나 변경하기 위해 서브 클래스인 "기본 클래스"로 사용하기보다는 대부분 있는 그대로 사용되고 속성을 통해 변경됩니다. 상속된 컴포넌트인 경우, 보통 기존 이벤트 핸들링 멤버 함수에 특정 코드를 추가해야 합니다.
- 컴포넌트의 속성은 본질적으로 런타임 타입 정보를 포함합니다.
- 컴포넌트는 Kylix 사용자 인터페이스의 컴포넌트 팔레트에 추가되어 폼에서 처리할 수 있습니다.

대부분의 Linux 시스템 이벤트는 Kylix 컴포넌트로 처리합니다. 메시지에 응답하려면 이벤트 핸들러만 제공하면 됩니다.

## 컴포넌트 팔레트의 컴포넌트

컴포넌트 팔레트에는 광범위한 프로그래밍 작업을 처리하는 여러 가지 컴포넌트들이 포함되어 있습니다. 이 팔레트에서 컴포넌트를 추가, 삭제 및 재정렬할 수 있습니다. 여러 가지 컴포넌트로 구성된 컴포넌트 템플릿과 프레임을 생성하여 컴포넌트 팔레트에 추가할 수도 있습니다.

팔레트에 있는 컴포넌트는 용도와 기능별로 페이지에 정렬되어 있습니다. 기본 구성에서 나타나는 페이지는 실행 중인 Kylix 버전에 따라 다릅니다. 표 3.3은 전형적인 기본 페이지와 기본 페이지에 포함되는 컴포넌트 타입을 나열한 것입니다.

**표 3.3** 컴포넌트 팔레트 페이지

| 페이지 이름          | 내용                                   |
|-----------------|--------------------------------------|
| Standard        | 표준 컨트롤, 메뉴                           |
| Additional      | 애플리케이션에서 사용하는 특화된 컨트롤                |
| Common Controls | 그래픽 사용자 인터페이스를 개발하는 데 사용되는 공통 컨트롤    |
| Dialogs         | 대화 상자                                |
| dbExpress       | dbExpress를 사용하는 데이터베이스 컴포넌트          |
| Data Access     | 데이터셋 프로바이더, 클라이언트 데이터셋 및 데이터 소스 컴포넌트 |
| Data Controls   | 비주얼, data-aware 컨트롤                  |
| Internet        | 인터넷 통신 프로토콜과 웹 애플리케이션용 컴포넌트          |
| Indy Clients    | 인터넷 클라이언트 애플리케이션 개발 컴포넌트             |
| Indy Servers    | 인터넷 서버 애플리케이션 개발 컴포넌트                |
| Indy Misc       | 추가 인터넷 컴포넌트                          |

온라인 도움말은 기본 팔레트에 있는 여러 컴포넌트에 대한 정보를 제공합니다. 도움말을 표시하기 위해 폼에 컴포넌트를 가져다 놓은 후 컴포넌트 팔레트 위에서, 또는 컴포넌트 위에서 F1 키를 누릅니다.

## 텍스트 컨트롤

많은 애플리케이션이 텍스트를 사용자에게 보여주거나 사용자가 텍스트를 입력할 수 있도록 합니다. 이러한 목적으로 사용된 컨트롤 타입은 정보의 크기와 서식에 따라 다릅니다.

| 컴포넌트             | 수행 작업                           |
|------------------|---------------------------------|
| <i>TEdit</i>     | 한 줄의 텍스트를 편집합니다.                |
| <i>TMemo</i>     | 여러 줄의 텍스트를 편집합니다.               |
| <i>TMaskEdit</i> | 우편 번호 또는 전화 번호 같은 특정 서식을 사용합니다. |

*TEdit*와 *TMaskEdit*는 정보를 입력할 수 있는 한 줄의 텍스트 편집 상자가 포함된 단순한 텍스트 컨트롤입니다. 편집 상자에 포커스가 있으면 깜빡이는 삽입 지점이 나타납니다.

문자열 값을 텍스트의 *Text* 속성에 할당하여 텍스트를 편집 상자에 포함할 수 있습니다. *Font* 속성에 값을 할당하여 편집 상자에 있는 텍스트 글꼴을 조정합니다. 글꼴 종류, 크기, 색상 및 속성을 지정할 수 있습니다. 속성은 편집 상자의 모든 텍스트에 영향을 미치고 개별 문자에 적용될 수 없습니다.

포함하는 글꼴의 크기에 따라 편집 상자의 크기가 변경되도록 할 수 있습니다. *AutoSize* 속성을 True로 설정하면 글꼴 크기에 따라 편집 상자의 크기가 변합니다. *MaxLength* 속성에 값을 할당하여 편집 상자의 문자 수를 제한할 수 있습니다.

*TMaskEdit*은 텍스트의 유효한 폼을 암호화하는 마스크에 대하여 입력된 텍스트를 확인하는 특별 편집 컨트롤입니다. 마스크는 사용자에게 표시된 텍스트의 서식을 지정할 수 있습니다.

*TMemo*는 여러 줄의 텍스트를 추가할 때 사용합니다.

## 텍스트 컨트롤 속성

다음은 텍스트 컨트롤의 중요한 속성 중 일부를 보여줍니다.

**표 3.4** 텍스트 컨트롤 속성

| 속성               | 설명                                            |
|------------------|-----------------------------------------------|
| <i>Text</i>      | 편집 상자 또는 메모 컨트롤에 나타나는 텍스트를 결정합니다.             |
| <i>Font</i>      | 편집 상자나 메모 컨트롤에 쓰인 텍스트의 속성 (attribute)을 제어합니다. |
| <i>AutoSize</i>  | 현재 선택된 글꼴에 맞게 편집 상자의 높이를 동적으로 변경할 수 있도록 합니다.  |
| <i>ReadOnly</i>  | 사용자가 텍스트를 변경할 수 있는지 여부를 지정합니다.                |
| <i>MaxLength</i> | 단순 텍스트 컨트롤에서 문자 수를 제한합니다.                     |

## 메모 컨트롤의 속성

*TMemo*는 여러 줄의 텍스트를 처리하는 다른 타입의 편집 상자입니다. 메모 컨트롤의 줄은 편집 상자의 오른쪽 테두리 이상으로 확장할 수 있거나, 다음 줄로 줄 바꿈할 수 있습니다. *WordWrap* 속성을 사용하여 줄 바꿈 여부를 조정합니다.

메모 컨트롤에는 다음과 같은 속성이 포함되어 있습니다.

- *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 목록으로 포함합니다.
- *WordWrap*은 텍스트가 오른쪽 여백을 만나면 줄 바꿈할지 여부를 결정합니다.
- *WantReturns*는 사용자가 텍스트에 하드 리턴을 삽입할 수 있는지 여부를 결정합니다.
- *WantTabs*은 사용자가 텍스트에 탭을 삽입할 수 있는지 여부를 결정합니다.
- *SelectText*는 텍스트의 현재 선택된(강조된) 부분을 포함합니다.
- *SelectStart*와 *SelectLength*는 텍스트의 선택된 부분의 위치와 길이를 나타냅니다.

런타임에 *SelectAll* 메소드를 사용하여 메모의 모든 텍스트를 선택할 수 있습니다.

## 특수한 입력 컨트롤

다음 컴포넌트는 입력을 캡처하는 추가 방법을 제공합니다.

| 컴포넌트              | 수행 작업                                    |
|-------------------|------------------------------------------|
| <i>TScrollBar</i> | 연속적인 범위의 값을 선택합니다.                       |
| <i>TTrackBar</i>  | 스크롤 막대보다 시각적으로 더 효과적인 연속적인 범위의 값을 선택합니다. |
| <i>TSpinEdit</i>  | 스피너 widget에서 값을 선택합니다.                   |

## 스크롤 막대

스크롤 막대 컴포넌트는 창, 폼 또는 다른 컨트롤의 내용을 스크롤하는 데 사용할 수 있는 스크롤 막대를 만듭니다. *OnScroll* 이벤트 핸들러에서, 사용자가 스크롤 막대를 움직일 때 컨트롤의 행동을 결정하는 코드를 작성합니다.

많은 비주얼 컴포넌트에는 자체 스크롤 막대가 포함되어 있어 추가 코딩이 필요 없습니다. 따라서 스크롤 막대 컴포넌트를 자주 사용하지 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 있습니다. 이것은 색상, 부피 및 밝기 같은 속성을 조정하는 데 유용합니다. 사용자는 슬라이드 지시자를 특정 위치로 끌어 놓거나 표시줄 내부를 눌러서 슬라이드 지시자를 옮깁니다.

- *Max* 속성과 *Min* 속성을 사용하여 트랙 표시줄의 범위 시작과 끝을 설정합니다.
- *SelEnd*와 *SelStart*를 사용하여 선택 범위를 강조 표시합니다. 그림 3.4를 참조하십시오.
- *Orientation* 속성은 트랙 표시줄이 수직인지 수평인지 결정합니다.
- 기본적으로 트랙 표시줄에는 트랙을 따라 움직이는 틱(tick)이 있습니다. *TickMarks* 속성을 사용하여 틱 위치를 변경합니다. 틱의 간격을 조정하려면, *TickStyle* 속성과 *SetTick* 메소드를 사용하십시오.

그림 3.4 트랙 표시줄 컴포넌트의 세 가지 보기



- *Position*은 트랙 표시줄에 대한 기본 위치를 설정하고 런타임 시 위치를 추적합니다.
- 기본적으로 사용자는 위쪽 화살표와 아래쪽 화살표 키를 눌러서 틱을 위나 아래로 움직일 수 있습니다. 틱의 이동 단위를 변경하려면 *LineSize*를 설정합니다.
- *PageSize*를 설정하여 사용자가 *Page Up* 키와 *Page Down* 키를 누를 때 틱의 이동 정도를 결정할 수 있습니다.

## 스핀 편집 컨트롤

스핀 편집 컨트롤은 엽다운 widget, 작은 화살표 widget 또는 스핀 버튼이라고도 합니다. 이 컨트롤을 사용하여 위 또는 아래 화살표 버튼을 눌러 현재 표시된 값을 증가 또는 감소시키거나 스핀 상자에 직접 값을 입력하여, 애플리케이션 사용자가 고정된 증분으로 정수 값을 변경할 수 있습니다.

현재 값은 *Value* 속성에 의해 주어지고, 기본값이 1인 증분은 *Increment* 속성에 의해 지정됩니다. *Associate* 속성을 사용하여 편집 컨트롤 같은 다른 컴포넌트를 엽다운 컨트롤에 첨부합니다.

## 버튼 및 이와 유사한 컨트롤

메뉴와 함께 버튼은 애플리케이션에서 동작이나 명령을 초기화하는 가장 일반적인 방법을 제공합니다. Kylix는 다음과 같은 몇 가지 버튼 유형의 컨트롤을 제공합니다.

| 컴포넌트                | 수행 작업                                       |
|---------------------|---------------------------------------------|
| <i>TButton</i>      | 텍스트가 있는 버튼에서 명령 선택을 보여줍니다.                  |
| <i>TSpeedButton</i> | 그룹화된 툴바 버튼을 만듭니다.                           |
| <i>TCheckBox</i>    | 선택/선택 해제 옵션으로 사용합니다.                        |
| <i>TRadioButton</i> | 상호 배타적인 선택 집합을 나타냅니다.                       |
| <i>TToolBar</i>     | 행에서 툴 버튼 및 기타 컨트롤을 정렬하고 크기와 위치를 자동으로 조정합니다. |

작업 목록을 사용하여 명령에 응답하는 메뉴와 버튼과 같은 객체에 대해 사용자 명령(동작)에 응답을 집중할 수 있습니다. 버튼과 툴바와 함께 작업 목록을 사용하는 방법에 대한 자세한 내용은 6-13 페이지의 "작업 목록 사용"을 참조하십시오.

CLX를 사용하여 그리기 버튼을 따로 사용자 지정하거나 애플리케이션 전체에 사용자 지정할 수 있습니다. 다음 장을 참조하십시오. 6장 "애플리케이션 사용자 인터페이스 개발"

## 버튼 컨트롤

마우스로 버튼 컨트롤을 눌러 동작을 초기화합니다. 버튼에는 동작을 나타내는 텍스트 레이블이 있습니다. *Caption* 속성에 문자열 값을 할당하여 텍스트를 지정합니다. 또한 키보드의 단축키를 눌러서 대부분의 버튼을 선택할 수 있습니다. 단축키는 버튼에서 밑줄 친 문자로 나타냅니다.

*OnClick* 이벤트 핸들러를 만들어 *TButton* 컴포넌트에 동작을 할당할 수 있습니다. 디자인 타임에 버튼을 두 번 눌러서 코드 에디터에 있는 버튼의 *OnClick* 이벤트 핸들러로 이동할 수 있습니다.

- 사용자가 *Esc* 키를 누를 때 버튼의 *OnClick* 이벤트를 트리거하게 하려면 *Cancel*을 *True*로 설정합니다.
- *Enter* 키를 누르면 버튼의 *OnClick* 이벤트를 트리거하도록 하려면 *Default*를 *True*로 설정합니다.

## 비트맵 버튼

비트맵 버튼(*TBitBtn*)은 비트맵 이미지가 있는 버튼 컨트롤입니다.

- 버튼에 표시할 비트맵을 선택하려면 *Glyph* 속성을 설정합니다.
- *Kind*를 사용하여 *glyph*와 기본 행동을 가지는 버튼을 자동으로 구성합니다.
- 기본적으로 *glyph*는 텍스트의 왼쪽에 나타납니다. 이동하려면, *Layout* 속성을 사용합니다.
- *glyph*와 텍스트는 자동으로 버튼의 가운데에 표시됩니다. 표시 위치를 이동하려면 *Margin* 속성을 사용합니다. *Margin*은 이미지의 가장자리와 버튼의 가장자리 사이의 픽셀의 수를 결정합니다.
- 기본적으로 이미지와 텍스트는 4개의 픽셀로 분리됩니다. *Spacing*을 사용하여 간격을 늘리거나 줄입니다.
- 비트맵 버튼에는 up, down 및 held down이라는 3가지 상태가 있습니다. *NumGlyphs* 속성을 3으로 설정하면 각 상태에 대해 각각 다른 비트맵을 보여줄 수 있습니다.

## 스피드 버튼

스피드 버튼(*TSpeedButton*)은 보통 이미지를 가지고 있고 그룹으로 묶어서 사용합니다. 일반적으로 스피드 버튼을 패널과 함께 사용하여 툴바를 만듭니다.

- 스피드 버튼을 그룹으로 작동하게 하려면, 모든 버튼의 *GroupIndex* 속성에 0이 아닌 동일한 값을 할당합니다.
- 기본적으로 스피드 버튼은 버튼을 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 그룹을 라디오 그룹처럼 동작하게 하려면 *AllowAllUp*을 *False*로 설정합니다.

스피드 버튼에 대한 자세한 내용은 6-33 페이지의 "패널 컴포넌트를 사용하여 툴바 추가" 단원을 참조하십시오.

## 확인란

확인란은 사용자가 선택 또는 선택 해제를 선택할 수 있는 토글입니다. 선택하면 확인란에 체크 표시가 나타납니다. 선택하지 않으면, 확인란은 빈 상태가 됩니다. *TCheckBox*를 사용하여 확인란을 만듭니다.

- 기본적으로 확인란을 선택 표시되게 하려면 *Checked*를 *True*로 설정합니다.
- *AllowGrayed*를 *True*로 설정하여 확인란에 선택, 선택 해제 및 비활성(grayed)라는 세 가지의 상태를 부여합니다.
- *State* 속성은 확인란이 선택되었는지(*cbChecked*), 선택되지 않았는지(*cbUnchecked*) 또는 비활성인지(*cbGrayed*) 나타냅니다.

**참고** 확인란 컨트롤은 두 가지 바이너리 상태 중 하나를 표시합니다. 다른 설정으로 확인란의 현재 값을 결정할 수 없는 경우 미결정 상태가 사용됩니다.

## 라디오 버튼

옵션 버튼이라고도 하는 라디오 버튼은 상호 배타적인 선택 집합을 나타냅니다. *TRadioButton*을 사용하여 라디오 버튼을 개별적으로 만들거나 *TRadioGroup*를 사용하여 자동으로 라디오 버튼을 그룹으로 정렬합니다. 라디오 버튼을 그룹화하여 사용자는 제한된 선택 집합에서 버튼을 하나 선택할 수 있습니다. 자세한 내용은 3-26 페이지의 "컴포넌트 그룹화"를 참조하십시오.

선택된 라디오 버튼은 가운데에 점이 있는 원으로 나타납니다. 선택되지 않은 경우 라디오 버튼은 빈 원으로 나타납니다. *Checked* 속성에 *True* 또는 *False* 값을 지정하여 라디오 버튼의 비주얼 상태를 변경합니다.

## 툴바

툴바는 비주얼 컨트롤을 정렬하고 관리하는 쉬운 방법을 제공합니다. 패널 컴포넌트와 스피드 버튼으로부터 툴바를 만들거나 *TToolBar* 컴포넌트를 사용한 다음, 마우스 오른쪽 버튼을 눌러 *New Button*을 선택하여 툴 버튼을 툴바에 추가할 수 있습니다.

*TToolBar* 컴포넌트를 사용하면 다음과 같은 여러 가지 장점이 있습니다. 다른 컨트롤은 상대적인 위치와 높이를 유지하는 반면 툴바에 있는 버튼은 자동으로 일정한 크기와 간격을 유지하고, 컨트롤들이 수평 정렬되지 않으면 자동으로 래퍼라운드되어 새 줄에서 시작하며, *TToolBar*는 투명도, 팝업 테두리, 공간과 스플리터 등 표시 옵션을 제공하여 컨트롤을 그룹화합니다.

작업 목록을 만들어서 툴바와 메뉴에서의 동작들을 한 곳으로 모아서 사용할 수 있습니다. 작업 목록을 버튼과 툴바와 함께 사용하는 방법에 대한 자세한 내용은 6-13 페이지의 "작업 목록 사용"을 참조하십시오.

툴바는 편집 상자, 콤보 상자 등과 같은 다른 컨트롤의 부모가 될 수 있습니다.

## 스플리터 컨트롤

정렬된 컨트롤 사이에 있는 스플리터 (*TSplitter*)를 사용하면 사용자는 컨트롤의 크기를 조정할 수 있습니다. 패널과 그룹 상자 같은 컴포넌트와 함께 스플리터를 사용하면 사용자는 각 창에 다양한 컨트롤을 가지는 여러 개의 창으로 폼을 나눌 수 있습니다.

패널이나 다른 컨트롤을 폼에 놓은 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 *allLeft*로 설정한 다음, *allLeft*로 정렬된 스플리터를 패널의 오른쪽에 두고, 마지막으로 *allLeft* 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.

## 목록 처리

목록은 선택할 항목들을 사용자에게 보여줍니다. 목록을 표시하는 일부 컴포넌트는 다음과 같습니다.

| 컴포넌트             | 표시 내용                              |
|------------------|------------------------------------|
| <i>TListBox</i>  | 텍스트 문자열 목록.                        |
| <i>TComboBox</i> | 스크롤할 수 있는 드롭다운 목록이 있는 편집 상자.       |
| <i>TTreeView</i> | 계층구조 목록.                           |
| <i>TListView</i> | 아이콘, 열 및 헤더가 있는 끌어 놓을 수 있는 항목의 목록. |

논비주얼(nonvisual) *TStringList*와 *TImageList* 컴포넌트를 사용하여 문자열과 이미지 집합을 관리합니다. 문자열 목록에 대한 자세한 내용은 3-32 페이지의 "문자열 목록 사용"을 참조하십시오.

## 목록 상자와 확인 목록 상자

목록 상자 (*TListBox*)와 확인 목록 상자는 사용자가 가능한 옵션 목록에서 하나 이상 선택할 수 있는 목록을 표시합니다. 선택은 텍스트, 그래픽 또는 두 가지를 모두 사용하여 표현됩니다.

- *Items*는 *TStrings*를 사용하여 컨트롤을 값으로 채웁니다.
- *ItemIndex*는 목록에서 선택된 항목을 나타냅니다.
- *MultiSelect*는 한 번에 둘 이상의 항목을 선택할 수 있는지 여부를 지정합니다.
- *Sorted*는 목록이 알파벳순으로 정렬되는지 결정합니다.
- *Columns*는 목록 컨트롤에 있는 열의 수를 지정합니다.
- *ItemHeight*는 각 항목의 높이를 픽셀 단위로 지정합니다. *Style* 속성은 *ItemHeight*를 무시하도록 할 수 있습니다.
- *Style* 속성은 목록 컨트롤이 항목을 표시하는 방법을 결정합니다. 기본적으로 항목은 문자열로 표시됩니다. *Style* 값을 변경하여 항목을 그래픽이나 변하는 높이로 표시하는 *owner-draw* 목록 상자를 만들 수 있습니다. *owner-draw* 컨트롤에 대한 내용은 7-7 페이지의 "컨트롤에 그래픽 추가"를 참조하십시오.

다음과 같은 방법으로 간단한 목록 상자를 만듭니다.

- 1 프로젝트 내에서 컴포넌트 팔레트의 목록 상자 컴포넌트를 폼에 가져다 놓습니다.
- 2 목록 상자의 크기를 정하고 필요하면 목록 상자를 정렬합니다.
- 3 *Items* 속성의 오른쪽을 두 번 클릭하거나 생략 버튼을 선택하여 String List Editor를 표시합니다.
- 4 에디터를 사용하여 목록 상자의 내용에 대해 줄로 정렬된 프리 폼 텍스트를 입력합니다.
- 5 OK를 선택합니다.

목록 상자에서 여러 항목을 선택하기 위해 *ExtendedSelect* 속성과 *MultiSelect* 속성을 사용할 수 있습니다.



## 콤보 상자

콤보 상자 (*TComboBox*)는 편집 상자를 스크롤 가능한 목록과 결합합니다. 입력하거나 목록에서 선택하여 데이터를 컨트롤에 입력하면 *Text* 속성 값이 변경됩니다. *AutoComplete*를 사용 가능으로하면 사용자가 데이터를 입력할 때 애플리케이션은 목록에서 가장 가깝게 일치하는 것을 찾아서 표시합니다.

콤보 상자의 세 가지 종류에는 표준, 드롭다운(기본값) 및 드롭다운 목록이 있습니다.

- *Style* 속성을 사용하여 필요한 콤보 상자의 종류를 선택합니다.
- *csDropDown*을 사용하여 드롭다운 목록이 있는 편집 상자를 만듭니다. *csDropDownList*를 사용하여 편집 상자를 읽기 전용(사용자가 목록에서 선택하도록 함)으로 만듭니다. *DropDownCount* 속성을 설정하여 목록에 표시된 항목의 수를 변경합니다.
- *csSimple*을 사용하여 닫히지 않은 고정된 목록을 가진 표준 콤보 상자를 만듭니다. 콤보 상자의 크기를 조정하여 목록 항목이 모두 표시되는지 확인하십시오.
- *csOwnerDrawFixed* 또는 *csOwnerDrawVariable*을 사용하여 항목을 그래픽하게, 또는 변하는 높이로 항목을 표시하는 *owner-draw* 콤보 상자를 만듭니다. *owner-draw* 컨트롤에 대한 내용은 7-7 페이지의 "컨트롤에 그래픽 추가"를 참조하십시오.

런타임시 콤보 상자는 Kylix와 Delphi에서 다르게 작동합니다. Kylix에서는 텍스트를 입력하고 콤보 상자의 편집 필드에서 Enter 키를 눌러서 드롭다운에 항목을 추가할 수 있지만, Delphi는 그렇지 않습니다. *InsertMode*를 *ciNone*으로 설정하여 이 기능을 선택 해제할 수 있습니다. 또한 콤보 상자의 목록에 빈(문자열이 없는) 항목을 추가할 수 있습니다. 아래쪽 화살표를 계속 누르고 있으면, 콤보 상자 목록의 마지막 항목에서 계속 스크롤됩니다. 위에서부터 다시 반복합니다.

## 트리 뷰

트리 뷰 (*TTreeView*)는 들여쓰기로 항목을 표시합니다. 컨트롤에는 노드를 확장하거나 수축시키는 버튼이 있습니다. *Indent*는 항목을 부모로부터 수평으로 분리하는 픽셀 수를 설정합니다. 사용자는 항목의 텍스트 레이블이 있는 아이콘을 포함할 수 있고, 노드가 확장되거나 축소되는지에 따라 각각 다른 아이콘을 표시할 수 있습니다. 항목에 대한 상태 정보를 반영하는 확인란과 같은 그래픽을 포함할 수도 있습니다.

디자인 타임에 항목을 트리 뷰 컨트롤에 추가하려면, 컨트롤을 더블 클릭하여 *TreeView Items* 에디터를 표시합니다. 추가한 항목은 *Items* 속성 값이 됩니다.

트리 뷰는 *vsReport* 모드에 있는 목록 뷰와 유사한 열과 하위 항목을 표시할 수 있습니다.

## 목록 뷰

*TListView*를 사용하여 만든 목록 뷰는 다양한 서식으로 목록을 표시합니다. 다음과 같이 *ViewStyle* 속성을 사용하여 원하는 목록의 종류를 선택합니다.

- *vsList*는 끌어 놓을 수 없는, 레이블이 있는 아이콘으로 항목을 표시합니다.
- *vsReport*는 열로 정렬된 정보를 가진 별도의 행에 항목을 표시합니다. 가장 왼쪽에 있는 열에는 작은 아이콘과 레이블이 있고, 다음 열에는 애플리케이션에 의해 지정된 하위 항목에 있습니다. *ShowColumnHeaders* 속성을 사용하여 열에 대한 헤더를 표시합니다.

## 컴포넌트 그룹화

그래픽 인터페이스는 관련 컨트롤과 정보를 그룹화했을 때 사용하기가 쉽습니다. Kylix는 컴포넌트를 그룹화하기 위해 다음과 같은 여러 컴포넌트를 제공합니다.

| 컴포넌트                  | 수행 작업                                                   |
|-----------------------|---------------------------------------------------------|
| <i>TGroupBox</i>      | 제목을 가지는 표준 그룹 상자.                                       |
| <i>TRadioGroup</i>    | 라디오 버튼의 단순 그룹.                                          |
| <i>TPanel</i>         | 시각적으로 더 유연한 컨트롤 그룹.                                     |
| <i>TScrollBar</i>     | 컨트롤을 포함하는 스크롤 가능한 영역.                                   |
| <i>TTabControl</i>    | 상호 배타적인 노트북 스타일 탭 집합.                                   |
| <i>TPageControl</i>   | 탭마다 페이지를 가지는 상호 배타적인 노트북 스타일 탭의 집합(각 페이지에는 다른 컨트롤을 포함). |
| <i>THeaderControl</i> | 크기 조정 가능한 열 헤더.                                         |

### 그룹 상자와 라디오 그룹

그룹 상자(*TGroupBox*)는 폼에서 관련 있는 컨트롤을 정렬합니다. 그룹화된 컨트롤 중에 가장 일반적인 것은 라디오 버튼입니다. 그룹 상자를 폼에 둔 후에 컴포넌트 팔레트에서 컴포넌트를 선택하여 그룹 상자에 둡니다. *Caption* 속성은 런타임에 그룹 상자에 레이블을 표시하는 텍스트를 포함합니다.

라디오 그룹 컴포넌트(*TRadioGroup*)는 라디오 버튼을 조합하여 함께 작동하도록 만드는 작업을 간편하게 합니다. 라디오 버튼을 라디오 그룹에 추가하려면, Object Inspector에서 *Items* 속성을 편집합니다. 즉, *Items*에 있는 각 문자열은 캡션 문자열이 있는 그룹 상자에 라디오 버튼을 나타나게 합니다. *ItemIndex* 속성 값은 현재 선택된 라디오 버튼을 결정합니다. *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 동일한 상대적인 위치를 유지할 수 있습니다. *BorderWidth* 속성은 패널 주위의 테두리 너비를 픽셀 단위로 지정합니다.

또한 다른 컨트롤을 패널에 놓고 *Align* 속성을 사용하여 폼에 있는 그룹의 모든 컨트롤의 적절한 위치를 확인합니다. 폼의 크기가 변경되더라도 패널의 위치를 제자리에 두기 위해 패널을 *atTop* 정렬할 수 있습니다.

*BevelOuter* 속성과 *BevelInner* 속성을 사용하여 패널의 모습을 올라가거나 내려간 모습으로 변경할 수 있습니다. 이러한 속성 값들을 변경하여 다양한 비주얼 3차원 효과를 낼 수 있습니다. 단순히 올라가거나 내려간 빗면을 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 상자

스크롤 상자(*TScrollBox*)는 폼 내에 스크롤 영역을 만듭니다. 경우에 따라 애플리케이션은 지정된 영역에 비해 더 많은 정보를 표시해야 합니다. 목록 상자, 메모 및 폼 자체 등 일부 컨트롤은 자동으로 내용을 스크롤할 수 있습니다.

스크롤 상자의 다른 용도는 창에서 다양한 스크롤 영역(뷰)을 만드는 것입니다. 뷰는 상용 워드 프로세서, 스프레드시트 및 프로젝트 관리 애플리케이션에서 일반적입니다. 스크롤 상자는 폼의 임의 스크롤 하위 영역을 정의할 수 있는 추가 유연성을 제공합니다.

패널과 그룹 상자와 마찬가지로, 스크롤 상자는 *TButton* 객체 및 *TCheckBox* 객체와 같은 다른 컨트롤을 포함합니다. 그러나 스크롤 상자는 일반적으로 보이지 않습니다. 스크롤 상자에 있는 컨트롤이 시각적인 영역보다 크면 스크롤 상자는 스크롤 막대를 자동으로 표시합니다.

스크롤 상자의 다른 용도는 툴바 또는 상태 표시줄(*TPanel* 컴포넌트)과 같은 곳에서 스크롤 영역을 제한하는 것입니다. 툴바와 상태 표시줄이 스크롤되는 것을 방지하려면, 스크롤 막대를 숨긴 다음 스크롤 상자를 툴바와 상태 표시줄 사이에 있는 창의 클라이언트 영역에 둡니다. 스크롤 상자에 연결된 스크롤 막대는 창에 속하는 것처럼 보이지만 스크롤 상자 내의 영역만을 스크롤합니다.

## 탭 컨트롤

탭 컨트롤 컴포넌트(*TTabControl*)는 노트북 분할기(divider)처럼 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. 멀티페이지 대화 상자를 만들려면 탭 컨트롤 대신 페이지 컨트롤을 사용합니다.

## 페이지 컨트롤

페이지 컨트롤 컴포넌트(*TPageControl*)는 멀티페이지 대화 상자에 적합한 페이지 집합입니다. 페이지 컨트롤은 *TTabSheet* 객체인 여러 개가 오버랩된 페이지를 표시합니다. 컨트롤의 상단에 있는 탭을 눌러 사용자 인터페이스에서 페이지를 선택합니다.

디자인 타임에 페이지 컨트롤에서 새 페이지를 만들려면, 컨트롤을 마우스 오른쪽 버튼으로 누르고 *New Page*를 선택합니다. 페이지에 대한 객체를 만들고 객체의 *PageControl* 속성을 설정하여 런타임에 새 페이지를 추가합니다.

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

코드에서 활성 페이지에 액세스하려면 *ActivePage* 속성을 사용합니다. 활성 페이지를 프로그램으로 변경하기 위해 *ActivePage* 속성이나 *ActivePageIndex* 속성을 설정할 수 있습니다.

## 헤더 컨트롤

헤더 컨트롤(*THeaderControl*)은 런타임에 선택하거나 크기 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 목록 상자(*TListBox*) 위에 둘 수 있습니다.

## 비주얼 피드백 제공

애플리케이션의 상태 정보를 제공하는 방법은 여러 가지가 있습니다. 예를 들어, *TForm*을 비롯한 일부 컴포넌트는 객체를 식별하기 위해 런타임에 설정할 수 있는 *Caption* 속성을 가집니다. 대화 상자를 만들어 메시지를 표시할 수도 있습니다. 그 밖에, 다음과 같은 컴포넌트가 런타임에 비주얼 피드백을 제공하는 데 특히 유용합니다.

| 컴포넌트 또는 속성                           | 수행 작업                              |
|--------------------------------------|------------------------------------|
| <i>TLabel</i>                        | 편집할 수 없는 텍스트를 표시합니다.               |
| <i>TStatusBar</i>                    | 일반적으로 창 하단에 상태 영역을 표시합니다.          |
| <i>TProgressBar</i>                  | 특정 작업에 대한 완료한 작업량을 보여줍니다.          |
| <i>Hint</i> 와 <i>ShowHint</i>        | 플라이바이(fly-by) 또는 "툴팁" 도움말을 활성화합니다. |
| <i>HelpContext</i> 와 <i>HelpFile</i> | 문맥에 따른 온라인 도움말을 연결합니다.             |

## 레이블

레이블(*TLabel*)은 텍스트 또는 픽스맵(pixmap) 필드를 표시하고 보통 다른 컨트롤 옆에 둡니다. 편집 상자와 같은 다른 컴포넌트를 식별하거나 표시하거나 또는 텍스트를 폼에 포함하고자 할 때 폼에 레이블을 놓습니다. 표준 레이블 컴포넌트 *TLabel*은 widget 기반 컨트롤이 아니므로 포커스를 받을 수 없습니다.

Label 속성은 다음과 같습니다.

- *Caption*은 레이블의 텍스트 문자열을 포함합니다.
- *Font*, *Color* 및 다른 속성은 레이블의 모습을 결정합니다. 각 레이블은 글꼴, 크기 및 색상을 하나만 사용할 수 있습니다.
- *FocusControl*은 레이블을 폼에 있는 다른 컨트롤에 연결합니다. *Caption*에 가속키가 있으면, *FocusControl*에 의해 지정된 컨트롤은 가속키를 누를 때 포커스를 받습니다.
- *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 사용합니다.

## 상태 표시줄

패널을 사용하여 상태 표시줄을 만들 수 있지만, 상태 표시줄 컴포넌트(*TStatusBar*)를 사용하는 것이 더 간단합니다. 기본적으로 상태 표시줄의 *Align* 속성은 위치와 크기를 모두 관리하는 *alBottom*로 설정됩니다.

상태 표시줄에서 한 번에 하나의 텍스트 문자열만 표시하려면, *SimplePanel* 속성을 *True*로 설정하고 *SimpleText* 속성을 사용하여 상태 표시줄에서 표시된 텍스트를 조정할 수 있습니다.

상태 표시줄을 패널이라는 여러 개의 텍스트 영역으로 분리할 수 있습니다. 패널을 만들려면, Object Inspector에서 *Panels* 속성을 편집하여 Panels 에디터에서 각 패널의 *Width*, *Alignment* 및 *Text* 속성을 설정합니다. 각 패널의 *Text* 속성에는 패널에 표시된 텍스트가 포함됩니다.

## 진행 표시줄

애플리케이션이 시간이 많이 소요되는 작업을 수행할 때, 진행 표시줄을 사용하여 작업이 얼마나 진행되었는지를 표시할 수 있습니다. 진행 표시줄(*TProgressBar*)은 왼쪽에서 오른쪽으로 증가하는 점선을 표시합니다.

그림 3.5 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*이 사용하는 증가 단위를 결정합니다.

## 도움말과 힌트 속성

대부분의 비주얼 컨트롤은 런타임 시 플라이바이(fly-by) 도움말 뿐만 아니라 문맥에 따른 도움말도 표시할 수 있습니다. *HelpContext* 속성과 *HelpFile* 속성은 컨트롤에 대한 도움말 컨텍스트 번호와 도움말 파일을 만듭니다.

*Hint* 속성은 마우스 포인터를 컨트롤 또는 메뉴 항목 위로 옮길 때 나타나는 텍스트 문자열을 포함합니다. 힌트를 사용하기 위해 *ShowHint*를 *True*로 설정합니다. *ParentShowHint*를 *True*로 설정하면 컨트롤의 *ShowHint* 속성이 부모와 동일한 값을 가지게 됩니다.

## 그리드

그리드는 정보를 행과 열로 표시합니다. 데이터베이스 애플리케이션을 개발하는 경우에는 15장 "데이터 컨트롤 사용"에서 설명한 *TDBGrid* 컴포넌트를 사용합니다. 그 외의 경우 표준 그리기 그리드 또는 문자열 그리드를 사용합니다.

### 그리기 그리드

그리기 그리드(*TDrawGrid*)는 임의의 데이터를 표 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 비주얼 열을 결정합니다. *VisibleColCount*와 *VisibleRowCount*는 그리드에서 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 행동에 영향을 미칠 수 있습니다.

## 문자열 그리드

문자열 그리드 컴포넌트는 특수한 기능을 추가하여 문자열의 표시를 단순화하는 *TDrawGrid*의 자손입니다. *Cells* 속성은 그리드의 각 셀에 대한 문자열을 나열하고, *Objects* 객체는 각 문자열에 연결된 객체를 나열합니다. 모든 문자열과 특정한 열이나 행에 관련된 객체는 *Cols* 속성 또는 *Rows* 속성을 통해 액세스할 수 있습니다.

## 그래픽 표시

다음 컴포넌트를 사용하면 그래픽을 애플리케이션에 쉽게 통합할 수 있습니다.

| 컴포넌트             | 표시 내용              |
|------------------|--------------------|
| <i>TImage</i>    | 그래픽 파일             |
| <i>TShape</i>    | 기하학적 도형            |
| <i>TBevel</i>    | 3차원 선과 프레임         |
| <i>TPaintBox</i> | 런타임에 프로그램이 그리는 그래픽 |

## 이미지

이미지 컴포넌트 (*TImage*)는 비트맵, 아이콘 또는 드로잉과 같은 그래픽 이미지를 표시합니다. *Picture* 속성은 표시할 그래픽을 결정합니다. *Center*, *AutoSize*, *Stretch*, *Transparent*를 사용하여 표시 옵션을 설정합니다. 자세한 내용은 8-1 페이지의 "그래픽 프로그래밍 개요"를 참조하십시오.

## 도형 (shape)

도형 컴포넌트 (*TShape*)는 기하학적 도형을 표시합니다. 형상 컴포넌트는 widget 기반 컨트롤이 아니므로 사용자 입력을 받을 수 없습니다. *Shape* 속성은 컨트롤이 가정한 도형을 결정합니다. 도형의 색상을 변경하거나 패턴을 추가하려면, *TBrush* 객체의 *Brush* 속성을 사용합니다. 도형을 그리는 방법은 *TBrush*의 *Color* 속성과 *Style* 속성에 따라 다릅니다.

## 빗면

빗면 컴포넌트(*TBevel*)는 음각과 양각 모습으로 나타낼 수 있는 선입니다. *TPanel*과 같은 일부 컴포넌트는 빗면이 있는 액자 모양의 테두리를 만들 수 있는 기본 제공 속성을 가지고 있습니다. 기본 제공 속성을 사용할 수 없을 때, *TBevel*를 사용하여 액자 모양의 윤곽, 상자 또는 프레임을 만듭니다.

## 그리기 상자

그리기 상자 컴포넌트(*TPaintBox*)를 사용하면 애플리케이션이 폼에서 그릴 수 있습니다. *OnPaint* 이벤트 핸들러를 작성하여 이미지를 그리기 상자의 *Canvas*에서 직접 렌더링할 수 있습니다. 그리기 상자의 테두리 밖에서 그리는 것이 방지됩니다. 자세한 내용은 8-1 페이지의 "그래픽 프로그래밍 개요"를 참조하십시오.

## 대화 상자

컴포넌트 팔레트의 Dialog 페이지에 있는 대화 상자 컴포넌트는 다양한 대화 상자가 애플리케이션에서 사용될 수 있도록 합니다. 이러한 대화 상자는 파일 열기 및 저장과 같은 공통 파일 작업을 수행할 수 있는 익숙하고 일관성 있는 인터페이스를 애플리케이션에 제공합니다. 대화 상자는 데이터를 표시하거나 연습니다.

대화 상자는 *Execute* 메소드가 호출되면 열립니다. *Execute*는 부울 값을 반환합니다. OK를 선택하여 대화 상자의 변경 사항을 승인하는 경우 *Execute*는 *True*를 반환하고, Cancel을 선택하여 변경 사항을 저장하지 않고 대화 상자를 빠져 나가는 경우 *Execute*는 *False*를 반환합니다.

## 열린 대화 상자 사용

공통으로 사용되는 대화 상자 컴포넌트 중의 하나는 *TOpenDialog*입니다. 이 컴포넌트는 보통 폼의 메인 메뉴 바에 있는 File 옵션 아래의 New 또는 Open 메뉴 항목에 의해 호출됩니다. 대화 상자는 와일드카드 문자를 사용하여 파일 그룹을 선택하고 디렉토리를 통해 탐색할 수 있는 컨트롤을 포함합니다.

*TOpenDialog* 컴포넌트는 Open 대화 상자를 애플리케이션에 사용할 수 있도록 합니다. 이 대화 상자의 용도는 사용자에게 열려는 파일을 선택하도록 하는 것입니다. *Execute* 메소드를 사용하여 대화 상자를 표시합니다.

사용자가 대화 상자에서 OK를 선택하면, 사용자의 파일은 *TOpenDialog*의 *FileName* 속성에 저장된 후에 원하는대로 처리할 수 있습니다.

다음 코드를 *Action*에 둘 수 있고 *TMainMenu* 하위 항목의 *Action* 속성에 연결하거나 하위 항목의 *OnClick* 이벤트에 둘 수 있습니다.

```
if OpenDialog1.Execute then
    filename := OpenDialog1.FileName;
```

이 코드는 대화 상자를 보여주고, 사용자가 OK 버튼을 누르면 이전에 선언한 *filename*이라는 *AnsiString* 변수에 파일 이름을 복사합니다.

# helper 객체 사용

CLX는 공통 프로그래밍 작업을 단순화하는 다양한 논비주얼(nonvisual) 객체를 포함합니다. 여기서는 다음 작업을 보다 쉽게 수행하도록 하는 몇개의 helper 객체에 대해 설명합니다.

- 목록 사용
- 문자열 목록 사용
- 드로잉 공간 만들기
- 인쇄
- 스트림 사용

## 목록 사용

일부 CLX 객체는 목록을 생성하거나 관리하기 위해 다음 기능을 제공합니다.

표 3.5 목록의 생성 및 관리를 위한 컴포넌트

| 객체                                                                          | 보유                                                       |
|-----------------------------------------------------------------------------|----------------------------------------------------------|
| <i>TList</i>                                                                | 포인터 목록                                                   |
| <i>TObjectList</i>                                                          | 인스턴스 객체의 메모리 관리 목록                                       |
| <i>TComponentList</i>                                                       | 컴포넌트의 메모리 관리 목록<br>(즉, <i>TComponent</i> 의 자손 클래스의 인스턴스) |
| <i>TQueue</i>                                                               | 포인터의 선입선출(FIFO) 목록                                       |
| <i>TStack</i>                                                               | 포인터의 후입선출(LIFO) 목록                                       |
| <i>TObjectQueue</i>                                                         | 객체의 선입선출 목록                                              |
| <i>TObjectStack</i>                                                         | 객체의 후입선출 목록                                              |
| <i>TClassList</i>                                                           | 클래스 타입의 목록                                               |
| <i>TCollection</i> ,<br><i>TOwnedCollection</i> ,<br><i>TCollectionItem</i> | 특수하게 정의된 항목의 인덱스된 컬렉션                                    |
| <i>TStringList</i>                                                          | 문자열 목록                                                   |

이러한 객체들에 대한 자세한 내용은 온라인 참조를 보십시오.

## 문자열 목록 사용

경우에 따라 애플리케이션은 문자열 목록을 관리해야 합니다. 문자열 목록의 예로는 콤보 상자의 항목, 메모의 행, 글꼴 이름, 문자열 그리드의 열과 행 이름이 있습니다. CLX는 *TStrings*이라는 객체와 이 객체의 자손인 *TStringList*를 통해 모든 문자열 목록에 공통 인터페이스를 제공합니다. *TStringList*는 *TStrings*에 있는 추상 속성과 메소드를 구현하고, 속성, 이벤트 및 메소드를 사용하여 다음을 수행합니다.

- 목록에서 문자열을 정렬합니다.
- 정렬된 목록에서 문자열 중복을 금합니다.
- 목록 내용의 변경 사항에 응답합니다.



문자열 목록을 보유하는 기능을 제공하는 것 외에도 이러한 객체들은 상호 운용성을 용이하게 합니다. 예를 들어, *TStrings*의 인스턴스인 메모의 행을 편집한 다음 이 행들을 *TStrings*의 인스턴스인 콤보 상자의 항목으로 사용할 수 있습니다.

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

- 문자열 목록의 로드 및 저장
- 새 문자열 목록 작성
- 목록에서 문자열 처리
- 문자열 목록에 객체 연결

## 문자열 목록의 로드 및 저장

문자열 목록 객체는 *SaveToFile* 메소드와 *LoadFromFile* 메소드를 제공하므로 사용자는 문자열 목록을 텍스트 파일에 저장하고 텍스트 파일을 문자열 목록에 로드할 수 있습니다. 텍스트 파일의 각 행은 목록의 문자열에 해당합니다. 이러한 메소드를 사용하면, 파일을 메모 컴포넌트에 로드하여 단순한 텍스트 에디터를 만들거나 콤보 상자의 항목 목록을 저장할 수 있습니다.

다음 예제에서는 passwd 파일의 복사본을 메모 필드에 로드하고 passwd.bak이라는 백업 복사본을 만듭니다.

```
procedure EditFile;
var
  FileName: string;{ storage for file name }
begin
  FileName := '/etc/passwd';{ set the file name }
  with Form1.Memo1.Lines do
    begin
      LoadFromFile(FileName);{ load from file }
      SaveToFile(ChangeFileExt(FileName, '.bak'));{ save into backup file }
    end;
  end;
```

## 새 문자열 목록 작성

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

### 기간이 짧은 문자열 목록

문자열 목록을 단일 루틴 기간 동안만 사용하는 경우 그 문자열 목록을 한 루틴 내에서 생성, 사용 및 소멸할 수 있습니다. 이것은 문자열 목록을 사용하는 가장 안전한 방법입니다. 문자열 목록 객체는 목록 자체와 목록의 문자열에 대해 메모리를 할당하기 때문에, 예외가 발생하더라도 **try...finally** 블록을 사용하여 메모리가 해제되었는지 확인해야 합니다.

1 문자열 목록 객체를 생성합니다.

2 **try...finally** 블록의 **try** 부분에서 문자열 목록을 사용합니다.

**3 finally** 부분에서 문자열 목록 객체를 해제합니다.

다음 이벤트 핸들러는 문자열 목록을 생성, 사용 및 소멸함으로써 버튼 클릭에 응답합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try
    { use the string list }
  finally
    TempList.Free;{ destroy the list object }
  end;
end;
```

### 기간이 긴 문자열 목록

애플리케이션을 실행하는 동안 항상 문자열 목록을 사용 가능하도록 하려면, 애플리케이션 시작 시 목록을 생성하고 애플리케이션 종료 전에 소멸합니다.

- 1 애플리케이션의 메인 폼에 대한 유닛 파일에서 *TStrings* 타입의 필드를 폼의 선언에 추가합니다.
- 2 메인 폼의 *생성자*에 대한 이벤트 핸들러를 작성합니다. 이 이벤트 핸들러는 폼이 나타나기 전에 실행됩니다. 생성자는 문자열 목록을 만들어서 첫 번째 단계에서 선언한 필드에 할당해야 합니다.
- 3 폼의 *OnClose* 이벤트용 문자열 목록을 해제하는 이벤트 핸들러를 작성합니다.

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

```
unit Unit1;
interface
uses SysUtils, Classes, QGraphics, QControls, QForms, Qialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender:TObject);
    procedure FormDestroy(Sender:TObject);
    procedure FormMouseDown(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;

var
  Form1:TForm1;

implementation
```

```
{ $R *.xfrm }

procedure TForm1.FormCreate(Sender:TObject);
begin
    ClickList := TStringList.Create; { construct the list }
end;

procedure TForm1.FormDestroy(Sender:TObject);
begin
    ClickList.SaveToFile(ExtractFilePath(Application.ExeName) + '.LOG'); { save the
list }
    ClickList.Free; { destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
Shift:TShiftState; X, Y:Integer);
begin
    ClickList.Add(Format('Click at (%d, %d)', [X, Y])); { add a string to the list }
end;

end.
```

## 목록에서 문자열 처리

다음은 문자열 목록에서 공통으로 수행되는 작업입니다

- 목록의 문자열 수 계산
- 특정 문자열 액세스
- 문자열 목록에서 항목 찾기
- 목록에서 문자열을 통한 반복
- 목록에 문자열 추가
- 목록 내에서 문자열 이동
- 목록에서 문자열 삭제
- 문자열 목록 전체 복사

### 목록의 문자열 수 계산

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

### 특정 문자열 액세스

*Strings* 배열 속성에는 0부터 시작하는 인덱스에 의해 참조되는 목록의 문자열을 포함합니다. *Strings*는 문자열 목록의 기본 속성이므로 목록에 액세스할 때 *Strings* 식별자를 생략할 수 있습니다. 따라서,

```
StringList1.Strings[0] := 'This is the first string.';
```

이 구문은 다음과 동일합니다.

```
StringList1[0] := 'This is the first string.';
```

## 문자열 목록에서 항목 찾기

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

예를 들어, 다음과 같이 *IndexOf*를 사용하여 목록 상자의 *Items* 중에서 파일 이름의 발견 여부를 알 수 있습니다.

```
if FileListBox1.Items.IndexOf('solitaire.so') > -1 ...
```

## 목록에서 문자열을 통한 반복

목록의 문자열을 통해 반복하려면 0에서 *Count* - 1까지 실행되는 **for** 순환문을 사용합니다.

다음 예제는 목록 상자에 있는 각 문자열을 대문자로 변환합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  Index:Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
  end;
```

## 목록에 문자열 추가

문자열을 문자열 목록의 끝에 추가하려면 *Add* 메소드를 호출하여 매개변수로 새 문자열을 전달합니다. 문자열을 목록에 삽입하려면 *Insert* 메소드를 호출하여 다음 두 개의 매개변수인 문자열과 문자열을 삽입할 위치에 해당하는 인덱스를 넘깁니다. 예를 들어, 문자열 "Three"를 목록에서 세 번째 문자열로 만들려면 다음과 같이 합니다.

```
Insert(2, 'Three');
```

문자열을 한 목록에서 다른 목록으로 추가하려면 다음과 같이 *AddStrings*를 호출하십시오.

```
StringList1.AddStrings(StringList2); { append the strings from StringList2 to
StringList1 }
```

## 목록 내에서 문자열 이동

문자열 목록에서 문자열을 이동하려면 *Move* 메소드를 호출하여 매개변수로 문자열의 현재 인덱스와 문자열을 옮길 인덱스를 넘깁니다. 예를 들어, 목록에서 세 번째 문자열을 다섯 번째 위치로 이동하려면 다음과 같이 합니다.

```
Move(2, 4)
```

## 목록에서 문자열 삭제

문자열 목록에서 문자열을 삭제하려면 목록의 *Delete* 메소드를 호출하여 삭제하려는 문자열의 인덱스를 전달합니다. 삭제하려는 문자열의 인덱스를 모르는 경우 *IndexOf* 메소드를 사용하여 위치를 찾아냅니다. 문자열 목록에서 모든 문자열을 삭제하려면 *Clear* 메소드를 사용합니다.

이 예제에서는 다음과 같이 *IndexOf*와 *Delete*를 사용하여 문자열을 찾아서 삭제합니다.

```
with ListBox1.Items do
begin
  BIndex := IndexOf('bureaucracy');
  if BIndex > -1 then
    Delete(BIndex);
end;
```

## 문자열 목록 전체 복사

*Assign* 메소드를 사용하여 소스 목록에서 대상 목록으로 문자열을 복사할 수 있습니다. 이 메소드는 대상 목록의 내용을 덮어 씹니다. 대상 목록에 덮어 쓰지 않고 문자열을 추가하려면 *AddStrings*를 사용합니다. 예를 들어,

```
Memo1.Lines.Assign(ComboBox1.Items); { overwrite original strings }
```

이 구문은 콤보 상자의 행을 메모에 복사(메모를 덮어씀)합니다. 반면에

```
Memo1.Lines.AddStrings(ComboBox1.Items); { appends strings to end }
```

이 구문은 콤보 상자의 행을 메모에 추가합니다.

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

```
StringList1 := StringList2;
```

원래 문자열 목록 객체를 잃게 되고 가끔 예상치 못한 결과를 가져옵니다.

## 문자열 목록에 객체 연결

*Strings* 속성에 저장된 문자열 외에, 문자열 목록은 객체에 대한 참조를 유지 관리할 수 있고 *Objects* 속성에 저장합니다. *Strings*와 마찬가지로 *Objects* 속성은 인덱스가 0부터 시작하는 배열입니다. 비트맵을 owner-draw 컨트롤의 문자열에 연결할 때 *Objects* 속성이 가장 많이 사용됩니다.

*AddObject* 메소드 또는 *InsertObject* 메소드를 사용하여 목록에 문자열과 연결 객체를 한 번에 추가합니다. *IndexOfObject*는 지정된 객체에 연결된 목록에서 첫 번째 문자열의 인덱스를 반환합니다. *Delete*, *Clear* 및 *Move*와 같은 메소드는 문자열과 객체 모두에서 작동합니다. 예를 들어, 문자열을 삭제하면 해당 객체가 있으면 해당 객체가 제거됩니다.

기존 문자열에 객체를 연결하려면 객체를 동일한 인덱스에 있는 *Objects* 속성에 할당합니다. 해당 문자열을 추가하지 않고 객체를 추가할 수 없습니다.

## 드로잉 공간 만들기

자체 이미지를 렌더링해야 하는 객체에 대해 *TCanvas*를 사용하여 추상 드로잉 공간을 만들 수 있습니다. *TCanvas* 객체는 드로잉 장치(Qt painter)를 캡슐화하여 폼, 패널과 같은 비주얼 컨테이너 및 프린터 객체(3-38 페이지의 "인쇄"에서 설명)에 대한 모든 드로잉을 처리합니다. 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 여러가지 기본적인 그래픽 루틴을 포함합니다. 예를 들어, 다음 코드는 폼의 왼쪽 상단 모서리에서 가운데로 선을 그리고 텍스트를 폼에 출력하는 버튼 이벤트 핸들러를 보여줍니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  Canvas.Pen.Color := clBlue;
  Canvas.MoveTo( 10, 10 );
  Canvas.LineTo( 100, 100 );
  Canvas.Brush.Color := clred;
  Canvas.Font.Name := 'Arial';
  Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y, 'This is the end of the line');
end;
```

*TCanvas*는 드로잉이 필요하거나 드로잉이 가능한 Kylix에서 사용되고, 그래픽 드로잉을 쉽게 만들 수 있습니다.

속성과 메소드의 완전한 목록은 *TCanvas*의 온라인 도움말을 참조하십시오.

## 인쇄

*TPrinter* 객체는 프린터에 그리는 그리기 장치입니다. 이 장치는 포스트스크립트를 생성하여 lpr, lp 또는 다른 프린트 명령에 포스트스크립트를 보냅니다. 프린터 객체는 폼의 *TCanvas*와 동일한 *TCanvas*를 사용하므로 폼에 그릴 수 있는 모든 것을 또한 인쇄할 수 있습니다. 이미지를 인쇄하려면, *BeginDoc* 메소드를 호출한 다음 인쇄하려는 캔버스 그래픽과 텍스트 (*TextOut* 메소드 사용)를 나열한 다음, *EndDoc* 메소드를 호출하여 인쇄할 작업을 프린터에 보냅니다.

이 예제에서는 폼에 버튼과 메모를 사용합니다. 사용자가 버튼을 누르면 메모의 내용은 페이지 주위의 200픽셀 테두리로 인쇄됩니다.

이 예제를 성공적으로 실행하려면, *Printers*를 **uses** 절에 추가합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  r:TRect;
  i:Integer;
begin
  with Printer do
  begin
    r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
    BeginDoc;
    for i := 0 to Memo1.Lines.Count do
      Canvas.TextOut(200,200 + (i *
        Canvas.TextHeight(Memo1.Lines.Strings[i])),
        Memo1.Lines.Strings[i]);
      Canvas.Brush.Color := clBlack;
      Canvas.FrameRect(r);
    EndDoc;
  end;
end;
```

*TPrinter* 객체의 사용에 대한 자세한 내용은 *TPrinter*의 온라인 도움말을 보십시오.

## 스트림 사용

스트림은 데이터를 읽고 쓰는 수단입니다. 스트림은 메모리, 문자열, 소켓 및 BLOB 스트림과 같은 다른 매체에 읽고 쓰기 위한 공통적인 인터페이스를 제공합니다.

다음 스트리밍 예제에서는 스트림을 사용하여 한 파일을 다른 파일에 복사합니다. 애플리케이션에는 두 가지 편집 컨트롤(From 및 To)과 Copy File 버튼이 있습니다.

```

procedure TForm1.CopyFileClick(Sender:TObject);
var
    stream1, stream2:TStream;
begin
    stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
    try
        stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
        try
            stream2.CopyFrom(Stream1,Stream1.Size);
        finally
            stream2.Free;
        finally
            stream1.Free
    end;

```

저장 매체를 읽고 쓰려면 특수한 스트림 객체를 사용합니다. *TStream*의 각 자손은 디스크 파일, 동적 메모리 등 특정 매체에 액세스하기 위한 메소드를 구현합니다. *TStream* 자손으로는 *TFileStream*, *TStringStream* 및 *TMemoryStream*이 있습니다. 읽기 및 쓰기용 메소드 이외에 이러한 객체는 애플리케이션에서 스트림의 임의 위치를 찾을 수 있습니다. *TStream*의 속성은 크기 및 현재 위치와 같은 스트림에 대한 정보를 제공합니다.





# 4

## 일반적인 프로그래밍 작업

이 장에서는 Kylix의 다음과 같은 일반적인 프로그래밍 작업 수행 방법에 대해 설명합니다.

- 클래스 이해
- 클래스 정의
- 예외 처리
- 인터페이스 사용
- 문자열 작업
- 파일 작업

### 클래스 이해

---

클래스는 속성, 메소드, 이벤트, 그리고 클래스에 대한 지역 변수 같은 클래스 멤버에 대한 추상적인 정의입니다. 클래스의 인스턴스를 만들면, 이 인스턴스를 객체라고 합니다. 객체라는 용어는 Kylix 설명서에서 가끔 부정확하게 사용되고, 클래스와 클래스의 인스턴스 간의 구별이 중요하지 않은 곳에서는 "객체"가 클래스를 가리키기도 합니다.

Kylix가 CLX에 많은 클래스를 포함하고 있기는 하지만, 객체 지향 프로그램을 작성할 경우에는 클래스를 추가해야 할 수도 있습니다. 사용자가 작성하는 클래스는 *TObject*의 자손이거나 그 자손 중 하나여야 합니다. 클래스 타입 선언에는 클래스의 필드와 메소드의 액세스를 제어하는 다음과 같은 세 가지의 이용 가능한 섹션이 있습니다.

```
Type
TClassName = Class(TObject)
    public
    protected
    private
end;
```

- public 섹션은 액세스 제한이 없는 필드와 메소드를 선언하고, 클래스 인스턴스와 자손 클래스는 이러한 필드와 메소드에 액세스할 수 있습니다.

- `protected` 섹션은 일부 액세스 제한이 있는 필드와 메소드를 포함하고, 자손 클래스는 이러한 필드와 메소드에 액세스할 수 있습니다.
- `private` 섹션은 액세스가 엄격하게 제한된 필드와 메소드를 선언하고, 클래스 인스턴스와 자손 클래스는 이러한 필드와 메소드에 액세스할 수 없습니다.

클래스를 사용하면 새 클래스를 기존 클래스의 자손으로 생성할 수 있다는 이점이 있습니다. 각 자손 클래스는 그 부모 및 조상 클래스의 필드와 메소드를 상속합니다. 또한 상속된 클래스를 오버라이드하는 새 클래스에서 메소드를 선언하여, 좀 더 특별한 새로운 동작을 도입할 수 있습니다.

자손 클래스의 일반 구문은 다음과 같습니다.

```
Type
  TClassName = Class (TParentClass)
    public
    protected
    private
  end;
```

부모 클래스 이름이 지정되지 않은 경우 클래스는 `TObject`로부터 직접 상속합니다. `TObject`는 기본 생성자와 소멸자를 포함한 몇 가지 메소드만 정의합니다.

클래스의 구문, 랭귀지 정의, 규칙에 대한 자세한 내용은 클래스 타입에 대한 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말을 참조하십시오.

## 클래스 정의

Kylix에서는 클래스를 선언하여 애플리케이션에서 사용해야 하는 프로그래밍 기능을 구현할 수 있습니다. Kylix의 일부 버전에는 클래스 완성(class completion)이라는 기능이 있는데, 이 기능은 사용자가 선언하는 클래스 멤버에 대한 뼈대 코드를 생성하여 새 클래스를 정의하고 구현하는 작업을 간단하게 합니다.

다음과 같이 클래스를 정의합니다.

- 1 IDE에서 프로젝트를 열고 File|New Unit를 선택하여 새 클래스를 정의할 수 있는 새 유닛을 만듭니다.
- 2 **interface** 섹션에 **uses** 절과 **type** 섹션을 추가합니다.
- 3 **type** 섹션에 클래스 선언을 작성합니다. 모든 멤버 변수, 속성, 메소드 및 이벤트를 선언해야 합니다.

```
TMyClass = class; {This implicitly descends from TObject}
public
private
published {If descended from TPersistent or below}
```

### 참고

`TPersistent` 또는 그 아래의 자손 객체에 대한 `published` 섹션만 포함해야 합니다.

특정 클래스의 자손 클래스를 원하는 경우에는 다음과 같은 정의에서 그 클래스를 나타내야 합니다.

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

예를 들면, 다음과 같습니다.

```

type TMyButton = class(TButton)
    property Size:Integer;
    procedure DoSomething;
end;

```

Kylix 버전에서 클래스 완성이 포함된 경우, **interface** 섹션에 있는 메소드 정의 안에 커서를 두고 *Ctrl+Shift+C*를 누르거나 마우스 오른쪽 버튼을 클릭하여 Complete Class at Cursor를 선택합니다. Kylix는 모든 완료되지 않은 속성 선언을 완성하고 **implementation** 섹션에서 필요한 비어 있는 메소드를 생성합니다. 클래스 완성이 없으면, 사용자가 직접 코드를 작성하여 속성 선언을 완성하고 메소드를 작성합니다.

위와 같은 예에서, 클래스 완성이 있으면, Kylix는 인터페이스 선언에 **read** 지정자와 **write** 지정자를 추가하여 모든 지원 필드나 메소드를 포함시킵니다.

```

type TMyButton = class(TButton)
    property Size: Integer read FSize write SetSize;
    procedure DoSomething;
private
    FSize:Integer;
    procedure SetSize(const Value:Integer);

```

또한, Kylix는 유닛의 **implementation** 섹션에 다음 코드를 추가합니다.

```

{ TMyButton }
procedure TMyButton.DoSomething;
begin
end;
procedure TMyButton.SetSize(const Value:Integer);
begin
    FSize := Value;
end;

```

- 4 메소드를 채웁니다. 예를 들면, 버튼에서 DoSomething 메소드를 호출할 때 경고음이 울리게 하려면, **begin**과 **end** 사이에 Beep를 추가합니다.

```

{ TMyButton }
procedure TMyButton.DoSomething;
begin
    Beep;
end;

procedure TMyButton.SetSize(const Value:Integer);
begin
    FSize := Value;
    DoSomething;
end;

```

버튼의 크기를 변경하기 위해 SetSize를 호출할 때도 경고음이 울린다는 것에 유의하십시오.

클래스와 메소드의 구문, 랭귀지 정의, 규칙에 대한 자세한 내용은 클래스 타입 및 메소드에 대한 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말을 참조하십시오.

## 예외 처리

Kylix는 일관된 방식으로 오류를 처리하는 메커니즘을 제공합니다. 예외 처리를 통해서 애플리케이션은 데이터나 리소스를 잃지 않고 복구 가능한 오류를 복구하고 필요한 경우 종료합니다. Kylix의 오류 조건은 예외에 의해 나타납니다. 이 단원에서는 예외를 사용하여 안전한 애플리케이션을 만들 수 있는 다음과 같은 작업들을 설명합니다.

- 코드 블록 보호
- 리소스 할당 보호
- RTL(Run-Time Library) 예외 처리
- 컴포넌트 예외 처리
- 외부 소스를 사용한 예외 처리
- 조용한 예외
- 사용자 고유의 예외 정의

### 코드 블록 보호

안정적인 애플리케이션을 만들려면, 예외가 발생할 때 코드에서 예외를 인식하고 이에 대해 응답해야 합니다. 응답을 지정하지 않으면 애플리케이션은 오류를 설명하는 메시지 상자를 보여줍니다. 이 때, 오류가 발생할 수 있는 곳을 알아내고 오류로 인해 데이터나 시스템 리소스의 손실을 가져올 수 있는 곳에 대해 반드시 응답을 정의해야 합니다.

예외에 대한 응답을 만들 때는 코드의 블록에서도 응답을 만듭니다. 일련의 문장들이 모두 오류에 대한 동일한 종류의 응답이 필요한 경우에는 그 문장들을 블록으로 그룹화한 다음 전체 블록에 적용되는 오류 응답을 정의할 수 있습니다.

예외에 대해 특정한 응답을 가진 블록을 `protected` 블록이라고 하는데, 그 이유는 이 블록이 애플리케이션을 종료시키거나 데이터를 손상시키는 오류를 막을 수 있기 때문입니다.

코드의 블록을 보호하려면 다음을 이해해야 합니다.

- 예외에 대한 응답
- 예외와 제어 흐름
- 예외 응답 중첩

### 예외에 대한 응답

오류 조건이 발생하면 애플리케이션은 예외를 발생시키는데, 이것은 애플리케이션이 예외 객체를 만든다는 것을 의미합니다. 예외가 일단 발생하면, 애플리케이션은 클린업 코드를 실행하거나 예외를 처리하거나 아니면 둘 다 수행합니다.

### 클린업 코드 실행

예외에 응답하는 가장 간단한 방법은 일부 클린업 코드가 실행되는 것을 보장하는 것입니다. 이러한 종류의 응답은 오류를 일으키는 조건을 고쳐주지는 않지만 애플리케이션이 불안정한 상태의 환경에 머무르지 않도록 해줍니다. 보통 이러한 종류의 응답을 사용하여 오류 발생 여부와 상관 없이 애플리케이션이 할당된 리소스를 확실히 해제하도록 합니다.

## 예외 처리

이것은 특정한 예외에 대한 특정한 응답입니다. 예외를 처리하면 오류 조건을 없애고 예외 객체를 소멸시키며 이를 통해 애플리케이션을 계속 실행할 수 있습니다. 보통 애플리케이션이 오류를 복구하고 계속 실행할 수 있도록 하기 위해 예외 핸들러를 정의합니다. 처리해야 할 예외의 종류에는 존재하지 않는 파일 열기 시도, 용량이 꽉 찬 디스크에 쓰기, 허용 한도를 초과하는 계산 등이 있습니다. 이러한 예외 중에서 "File not found" 같은 예외는 고치고 재시도하기가 쉽지만, 메모리 부족과 같은 예외는 애플리케이션이나 사용자가 고치기가 더 어렵습니다.

예외를 효과적으로 처리하려면 다음을 이해해야 합니다.

- 예외 핸들러 작성
- 예외 처리 문장
- 예외 인스턴스 사용
- 예외 핸들러의 범위
- 기본 예외 핸들러 제공
- 예외 클래스 다루기
- 예외 재발생

## 예외와 제어 흐름

예외는 코드의 정상적인 흐름을 방해하지 않으므로 오브젝트 파스칼을 사용하여 애플리케이션에 오류 처리 기능을 통합하기가 쉽습니다. 사실, 알고리즘의 주요 과정에 오류 확인과 오류 처리를 두지 않으면 예외는 사용자가 작성하는 코드를 단순화해 줄 것입니다.

Protected 블록을 선언할 때 블록 안에서 발생할 수 있는 예외에 대한 특정 응답을 정의하게 됩니다. 그 블록에서 예외가 발생하면 즉시 사용자가 정의한 응답으로 건너서 실행된 다음 그 블록을 벗어납니다.

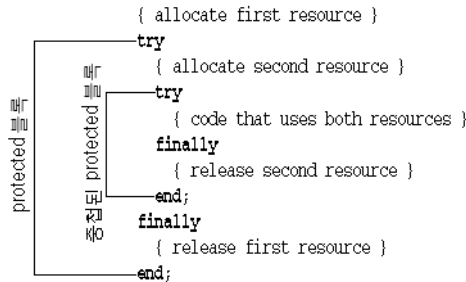
**예** 다음 코드는 protected 블록을 포함하고 있습니다. 예외가 protected 블록에서 발생하면 예외 처리 부분으로 건너 뛰어 실행되면서 경고음이 울립니다. 블록 밖에서 실행이 재개됩니다.

```
try
  AssignFile(F, FileName);
  Reset(F);
  :
except
  on Exception do Beep;
end;
: { execution resumes here, outside the protected block }
```

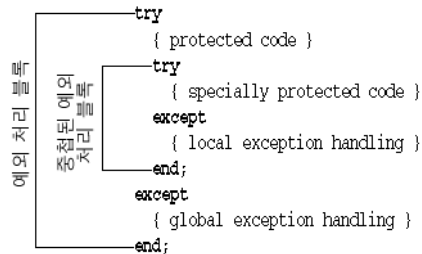
## 예외 응답 중첩

사용자 코드는 블록 내에서 발생하는 예외에 대한 응답을 정의합니다. 파스칼에서는 코드의 블록을 중첩할 수 있기 때문에, 사용자 지정된 응답을 이미 포함하고 있는 블록 내에서도 응답을 다시 사용자 지정할 수 있습니다.

가장 간단한 경우를 예로 들면 리소스 할당을 보호할 수 있고, 그 `protected` 블록 내에 다른 리소스를 할당하고 보호하는 블록을 정의할 수 있습니다. 개념적으로 보면 다음과 같습니다.



또한 주변 블록 안에서의 예외 처리를 오버라이드하는 특정 예외에 대한 지역 처리를 정의하기 위해서 중첩 블록을 사용할 수 있습니다. 개념적으로 보면 다음과 같습니다.



또한 예외 처리 블록 내에 리소스 보호를 중첩시키면서 다른 종류의 예외 응답 블록을 혼합할 수 있으며 그 반대로도 할 수 있습니다.

## 리소스 할당 보호

강력한 애플리케이션을 만드는 한 가지 중요한 요소는 예외가 발생하더라도 할당한 리소스를 확실히 해제하도록 하는 것입니다. 예를 들어, 애플리케이션이 메모리를 할당할 경우 반드시 메모리도 해제하도록 해야 합니다. 애플리케이션이 파일을 열 경우 나중에 파일도 반드시 닫도록 해야 합니다.

예외는 코드에서만 발생하는 것이 아니라는 점을 명심하십시오. 예를 들면, RTL 루틴 호출이나 애플리케이션의 다른 컴포넌트가 예외를 발생시킬 수 있습니다. 개발자가 작성한 코드는 이러한 조건이 발생했을 경우 할당된 리소스를 확실히 해제하도록 해야 합니다.

리소스를 효과적으로 보호하기 위해서는 다음 사항을 이해해야 합니다.

- 보호가 필요한 리소스 종류
- 리소스 보호 블록 만들기

## 보호가 필요한 리소스 종류

일반적인 환경에서는 애플리케이션이 할당과 해제에 대한 코드를 포함하여 할당된 리소스를 해제하도록 할 수 있습니다. 그러나 예외가 발생할 때에도, 애플리케이션이 리소스 해제 코드를 실행하게 해야 합니다.

항상 해제를 염두에 두어야 하는 몇 가지 일반적인 리소스는 다음과 같습니다.

- 파일
- 메모리
- 객체

**예** 다음 이벤트 핸들러는 메모리를 할당한 다음 오류를 생성하므로 메모리를 해제하는 코드를 절대 실행하지 않습니다.

```
procedure TForm1.Button1Click(Sender:TComponent);
var
  APointer:Pointer;
  AnInteger, ADividend:Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an error }
  FreeMem(APointer, 1024);{ it never gets here }
end;
```

대개의 오류가 이렇게 명백한 것은 아니지만 위의 예제는 중요한 점을 보여줍니다. 0으로 나누는 오류가 발생하면 블록 밖으로 건너 뛰어 실행되므로, *FreeMem* 문장이 절대로 메모리를 해제하지 못합니다.

*FreeMem*이 *GetMem*에 의해 할당된 메모리를 확실히 해제하도록 하려면, 코드를 리소스 보호 블록에 넣어야 합니다.

## 리소스 보호 블록 만들기

예외인 경우에도 할당된 리소스를 확실히 해제하려면, *protected* 블록 안에 리소스 사용 코드를 포함시키고 이 블록의 특별한 부분에 리소스 해제 코드를 추가해야 합니다. 전형적인 *protected* 리소스 할당 형태는 다음과 같습니다.

```
{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;
```

**try..finally** 블록의 주요 특징은 **protected** 블록에서 예외가 발생하더라도 애플리케이션이 블록의 **finally** 부분에 있는 모든 문장을 항상 실행한다는 것입니다. 블록의 **try** 부분의 코드 또는 **try** 부분의 코드에 의해 호출된 루틴이 예외를 발생시키면, 그 시점에서 실행이 중지됩니다. 예외 핸들러가 일단 발견되면, 클린업 코드라는 **finally** 부분으로 건너 뛰어 실행됩니다. **finally** 부분이 실행되고 나면 예외 핸들러가 호출됩니다. 예외가 발생하지 않으면, 클린업 코드가 정상적인 순서로 **try** 부분의 모든 문장 뒤에서 실행됩니다.

**예** 다음 코드에서는 메모리를 할당하고 오류를 생성하면서 할당된 메모리를 해제하는 이벤트 핸들러의 예를 보여줍니다.

```
procedure TForm1.Button1Click(Sender:TComponent);
var
  APointer:Pointer;
  AnInteger, ADividend:Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an error }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the error }
  end;
end;
```

**finally** 블록의 문장은 예외 발생에 영향을 받지 않습니다. **try** 부분의 문장에서 예외가 발생하지 않으면, **finally** 블록을 거쳐 계속실행됩니다.

## RTL(Run-Time Library) 예외 처리

수학 함수나 파일 처리 프로시저 같은 런타임 라이브러리(RTL)의 루틴을 호출하는 코드를 작성할 경우, RTL은 애플리케이션에 예외 형태로 오류를 보고합니다. 기본적으로 RTL 예외는 애플리케이션이 사용자에게 보여주는 메시지를 생성합니다. RTL 예외를 다른 방법으로 처리하는 자신만의 예외 핸들러를 정의할 수 있습니다.

기본적으로 메시지를 보여주지 않는 조용한 예외(silent exceptions)도 있습니다.

RTL 예외는 다른 모든 예외처럼 처리됩니다. RTL 예외를 효과적으로 처리하려면 다음 사항을 이해해야 합니다.

- RTL 예외
- 예외 핸들러 작성
- 예외 처리 문장
- 예외 인스턴스 사용
- 예외 핸들러의 범위
- 기본 예외 핸들러 제공
- 예외 클래스 다루기
- 예외 재발생



## RTL 예외

런타임 라이브러리의 예외는 SysUtils 유닛에서 정의되며, 이 예외는 모두 *Exception*이라는 일반적인 예외 객체 타입의 자손입니다. 모든 예외 클래스는 *Exception*의 자손입니다. 이것은 RTL 예외가 기본적으로 보여주는 메시지에 대한 문자열을 제공합니다.

다음 표에서 설명한대로 여러 종류의 예외가 RTL에 의해 발생합니다.

표 4.1 RTL 예외

| 오류 타입                                     | 원인                                        | 의미                                                                                                                         |
|-------------------------------------------|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| I/O                                       | 파일 또는 I/O 장치 액세스 오류                       | 대부분의 I/O 예외는 파일에 액세스할 때 반환된 오류 코드와 관련이 있습니다.                                                                               |
| 힙                                         | 동적 메모리 사용 오류                              | 사용 가능한 메모리가 불충분하거나 애플리케이션이 힙 외부로 가리키는 포인터를 처리할 때 힙 오류가 발생할 수 있습니다.                                                         |
| 정수 함수                                     | 정수 타입 표현식의 부적합한 연산                        | 오류에는 0으로 나누기, 범위 밖의 숫자 또는 표현식, 오버플로 등이 있습니다.                                                                               |
| 부동 소수점 수학                                 | 실수 타입 표현식의 부적합한 연산                        | 부동 소수점 오류는 하드웨어 보조프로세서 또는 소프트웨어 에뮬레이터가 원인입니다. 오류에는 잘못된 명령, 0으로 나누기, 오버플로 또는 언더플로 등이 있습니다.                                  |
| 타입 변환<br>(Typecast)<br>변환<br>(Conversion) | <b>as</b> 연산자를 사용한 잘못된 타입 변환<br>잘못된 타입 변환 | 객체는 호환 가능한 타입으로만 타입 변환할 수 있습니다.<br>IntToStr, StrToInt, StrToFloat 등과 같은 타입 변환 함수는 매개변수를 원하는 타입으로 변환할 수 없을 때 변환 예외를 발생시킵니다. |
| 하드웨어                                      | 시스템 상태                                    | 하드웨어 예외는 프로세서 또는 사용자가 액세스 위반, 스택 오버플로 또는 키보드 인터럽트 등과 같은 어떤 오류 조건이나 인터럽트를 생성했음을 나타냅니다.                                      |
| 가변                                        | 부적합한 타입 강제 변환                             | 가변(variant)을 호환 가능한 타입으로 강제 변환할 수 없는 표현식에서 가변을 참조하면 오류가 발생합니다.                                                             |

RTL 예외 타입의 전체 목록은 SysUtils 유닛을 참조하십시오.

## 예외 핸들러 작성

예외 핸들러는 특정 예외 또는 코드의 protected 블록 안에서 발생하는 예외를 처리하는 코드입니다. 일반적인 Kylix 프로그래밍에서는 예외 핸들러를 작성해야 하는 경우가 거의 없습니다. 대부분의 예외는 4-4 페이지의 "코드 블록 보호"와 4-6 페이지의 "리소스 할당 보호"에서 설명한대로 **try..finally** 블록을 사용하여 처리할 수 있습니다.

예외 핸들러를 정의하려면, 보호하려는 코드를 예외 처리 블록 안에 포함시키고 블록의 **except** 부분에 예외 처리 문장을 지정합니다. 전형적인 예외 처리 블록의 형태는 다음과 같습니다.

```
try
{ statements you want to protect }
except
{ exception-handling statements }
end;
```

애플리케이션은 예외가 **try** 부분의 문장이 실행되는 동안에 발생하는 경우에만 **except** 부분의 문장을 실행합니다. **try** 부분 문장의 실행에는 **try** 부분의 코드에 의해 호출되는 루틴이 포함됩니다. 즉, **try** 부분의 코드가 자체 예외 핸들러를 정의하지 않는 루틴을 호출하면, 예외 처리 블록으로 돌아와서 실행됩니다.

**try** 부분의 문장이 예외를 발생시키면, 즉시 **except** 부분으로 건너 뛰어 실행되고, 여기서 현재 예외에 적용되는 핸들러를 찾을 때까지 특정 예외 처리 문장 또는 예외 핸들러를 통한 단계를 거칩니다.

일단 애플리케이션이 예외를 처리하는 예외 핸들러를 찾게 되면, 그 문장을 실행한 후 예외 객체를 자동적으로 소멸시킵니다. 그러면 현재 블록의 끝에서 계속 실행됩니다.

## 예외 처리 문장

**try..except** 블록에서 **except** 부분의 각 **on** 문장들은 특별한 유형의 예외 처리 코드를 정의합니다. 이러한 예외 처리 문장의 형태는 다음과 같습니다.

```
on <type of exception> do <statement>;
```

**예** 0으로 나누기에 대한 기본 결과를 제공하는 예외 핸들러를 다음과 같이 정의할 수 있습니다.

```
function GetAverage(Sum, NumberOfItems:Integer):Integer;
begin
  try
    Result := Sum div NumberOfItems;{ handle the normal case }
  except
    on EDivByZero do Result := 0;{ handle the exception only if needed }
  end;
end;
```

함수를 호출할 때마다 0에 대한 검사를 하는 것보다는 위와 같이 하는 것이 더 명확합니다. 다음은 예외를 이용하지 않는 동일한 함수입니다.

```
function GetAverage(Sum, NumberOfItems:Integer):Integer;
begin
  if NumberOfItems <> 0 then{ always test }
    Result := Sum div NumberOfItems{ use normal calculation }
  else Result := 0;{ handle exceptional case }
  end;
```

위의 두 가지 함수의 차이는 예외를 가진 프로그래밍과 갖지 않은 프로그래밍 간의 차이를 정의합니다. 이러한 예는 매우 간단하지만, 수십 개의 입력한 내용 중 하나가 잘못되면 수백 개의 단계 중의 하나가 실패할 수 있는 경우와 같은 더 복잡한 계산을 생각해 볼 수 있습니다.

예외를 사용하면, 알고리즘의 "정상적인" 표현식을 모두 작성한 다음 이 알고리즘이 적용되지 않는 예외적인 경우를 만들 수 있습니다. 예외를 사용하지 않으면, 프로그램의 매 단계를 검사하여 계산의 각 단계를 진행할 수 있는지 확인해야 합니다.

## 예외 인스턴스 사용

대부분의 경우 예외 핸들러는 타입 이외의 예외에 대한 정보를 필요로 하지 않으므로, **on..do** 다음 문장은 예외의 타입에만 한정됩니다. 그러나, 어떤 경우에는 예외 인스턴스에 포함된 정보가 다소 필요할 수도 있습니다.

예외 핸들러에서 예외 인스턴스에 대한 특정 정보를 읽으려면, 예외 인스턴스에 액세스할 수 있게 해주는 **on..do**의 특별한 변형을 사용합니다. 이 특별한 폼에는 인스턴스를 유지하기 위한 임시 변수를 제공해야 합니다.

**예** 단일 폼을 포함하는 새 프로젝트를 만들 경우, 스크롤 막대와 명령 버튼을 폼에 추가할 수 있습니다. 버튼을 더블 클릭하고 클릭 이벤트 핸들러에 다음 줄을 추가합니다.

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

스크롤 막대의 최대 값이 최소 값을 항상 초과해야 하기 때문에 이 줄은 예외를 발생시킵니다. 애플리케이션에 대한 기본 예외 핸들러는 예외 객체에 메시지를 포함하는 대화 상자를 엽니다. 이 핸들러에서는 예외 처리를 오버라이드할 수 있으며 예외 메시지 문자열을 포함하는 자신의 메시지 상자를 만들 수 있습니다.

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

임시 변수(이 예에서 *E*)는 콜론 뒤에 지정된 타입입니다(이 예에서 *EInvalidOperation*). 필요한 경우에는 예외를 더 특정한 타입으로 타입 변환하기 위해 **as** 연산자를 사용할 수 있습니다.

**참고** 임시 예외 객체를 절대 소멸시키지 마십시오. 예외 처리는 자동으로 예외 객체를 소멸시킵니다. 사용자가 직접 객체를 소멸시키면 애플리케이션이 객체를 다시 소멸하여 액세스 위반을 일으킵니다.

## 예외 핸들러의 범위

모든 블록에 포함된 모든 종류의 예외에 핸들러를 제공할 필요는 없습니다. 실제로는, 특정 블록 안에서 특별하게 처리하고자 하는 예외에 대한 핸들러만 있으면 됩니다.

블록이 특정 예외를 처리하지 않으면, 그 블록을 떠나 예외를 발생시킨 채 블록을 호출한 코드가 들어 있는 블록으로 돌아가서 실행됩니다. 이 과정은 실행이 애플리케이션의 가장 바깥 범위에 도달하거나 블록이 어느 정도 예외를 처리할 때까지 범위를 넓혀 반복합니다.

## 기본 예외 핸들러 제공

하나의 기본 예외 핸들러를 제공하여 특정 핸들러를 제공하지 않은 모든 예외를 처리할 수 있습니다. 이를 위해 예외 처리 블록의 **except** 부분에 **else** 부분을 추가합니다.

```
try
    { statements }
except
    on ESomething do
        { specific exception-handling code };
    else
        { default exception-handling code };
end;
```

기본 예외 처리를 블록에 추가하면 블록이 어떤 방법으로든지 모든 예외를 처리해주므로, 포함된 블록의 모든 처리를 오버라이드합니다.

**주의** 모든 것을 다 처리하는 기본 예외 핸들러는 사용하지 않는 것이 좋습니다. **else** 절은 개발자가 알고 있지 못한 것까지 포함하여 모든 예외를 처리합니다. 코드는 일반적으로 처리 방법을 알고 있는 예외만을 처리해야 합니다. 클린업을 처리한 다음 예외와 그 처리 방법에 대해 더 많은 정보를 가지고 있는 코드가 예외를 처리하게 하려면, 다음과 같이 **try..finally** 블록을 포함시키는 방법을 사용합니다.

```
try
    try
        { statements }
    except
        on ESomething do { specific exception-handling code };
    end;
finally
    {cleanup code };
end;
```

다른 예외처리 방법은 예외 재발생을 참조하십시오.

## 예외 클래스 다루기

예외 객체는 계층 구조의 부분이기 때문에, 예외 객체 부분의 조상인 예외 클래스에 대한 핸들러를 제공함으로써 계층 구조의 전체 부분에 대한 핸들러를 지정할 수 있습니다.

**예** 다음 블록은 모든 정수 수학 예외를 처리하는 예를 보여줍니다.

```
try
    { statements that perform integer math operations }
except
    on EIntError do { special handling for integer math errors };
end;
```

좀 더 특정한 예외에 대한 특정 핸들러를 지정할 수 있지만, 이러한 핸들러는 일반적인 핸들러 위에 위치해야 합니다. 왜냐하면 애플리케이션은 핸들러를 나타난 순서대로 찾기 때문에 처음 찾는 적용 가능한 핸들러를 실행합니다. 예를 들면, 다음 블록에서는 범 위 오류에 대한 특별한 처리와 다른 모든 정수 수학 오류에 대한 처리를 제공합니다.

```
try
{ statements performing integer math }
except
on ERangeError do { out-of-range handling };
on EIntError do { handling for other integer math errors };
end;
```

*EIntError*에 대한 핸들러가 *ERangeError*에 대한 핸들러보다 먼저 오면, *ERangeError*는 *EIntError*의 자손이기 때문에 실행은 *ERangeError*에 대한 특정 핸들러에 걸 코 미치지 못합니다.

## 예외 재발생

가끔 지역적으로 예외를 처리할 때, 예외를 대체하지 않고 바깥쪽 블록에서의 처리를 원 할 수 있습니다. 물론, 로컬 핸들러가 예외 처리를 완료하는 경우 예외 인스턴스를 소멸 시키므로, 바깥쪽 블록의 핸들러는 절대 작동하지 않게 됩니다. 하지만 핸들러가 예외를 소멸시키는 것을 막아서 바깥쪽 핸들러에게 응답할 기회를 줄 수 있습니다.

**예** 예외가 발생할 때 표준 처리를 먼저 하고 나서 오류를 로그 파일에 기록하고자 할 경 우가 있습니다. 그렇게 하려면, 메시지를 보여주는 로컬 예외 핸들러를 선언한 다음 예악 어 **raise**를 호출합니다. 다음 예에 표시된 이러한 것을 예외 재발생이라고 합니다.

```
try
{ statements }
try
{ special statements }
except
on ESomething do
begin
{ handling for only the special statements }
raise;{ reraise the exception }
end;
end;
except
on ESomething do ...;{ handling you want in all cases }
end;
```

{ statements } 부분에 있는 코드가 *ESomething* 예외를 발생시키면, 바깥에 있는 **except** 부분의 핸들러만 실행됩니다. 하지만 { special statements } 부분에 있는 코드가 *ESomething* 예외를 발생시키면, 안쪽 **except** 부분에 있는 처리가 실행되고 나서 바깥 **except** 부분의 보다 일반적인 처리가 실행됩니다.

예외를 재발생시킴으로써 기존 핸들러를 잃거나 복제하지 않고 특별한 경우의 예외에 대한 특별한 처리를 쉽게 만들 수 있습니다.

## 컴포넌트 예외 처리

Kylix의 컴포넌트는 오류 조건을 나타내기 위해 예외를 발생시킵니다. 대부분의 컴포넌트 예외는 이것이 발생하지 않으면 런타임 오류를 생성시킬 프로그래밍 오류를 나타냅니다. 컴포넌트 예외 처리 메커니즘은 RTL 예외 처리와 같습니다.

**예** 컴포넌트에서 오류의 공통적인 원인은 인덱스된 속성에서의 범위 오류입니다. 예를 들어, 목록 상자의 목록에 세 개의 항목(0..2)이 있고 애플리케이션이 항목 번호 3에 액세스하려 할 경우, 목록 상자에서 "List index out of bounds" 예외를 발생시킵니다.

다음 이벤트 핸들러는 사용자에게 목록 상자의 잘못된 인덱스 액세스를 공지하는 예외 핸들러를 포함합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption := ListBox1.Items[3];{ set form caption to fourth string in list box }
  except
    on EStringListError do
      MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
  end;
end;
```

버튼을 한 번 클릭하면 목록 상자에는 세 개의 문자열만 있으므로, 네 번째 문자열(Items[3])에 액세스하면 예외가 발생합니다. 두 번째로 버튼을 클릭하면 더 많은 문자열을 목록에 추가하므로, 더 이상 예외를 일으키지 않습니다.

## 외부 소스를 사용한 예외 처리

*HandleException*은 애플리케이션의 예외를 기본 처리를 제공합니다. 일반적으로는 *TApplication.HandleException*을 호출하지 않아도 됩니다. 그렇지만 공유 객체 파일 또는 callback 함수를 작성할 때에는 필요합니다. 특히 예외를 지원하지 않는 외부 소스로부터 코드가 호출되고 있을 때 예외가 코드에서 벗어나지 않도록 하기 위해 *TApplication.HandleException*을 사용할 수 있습니다.

예를 들어, 예외가 애플리케이션 코드에 있는 모든 **try** 블록을 통과하면, 애플리케이션은 자동적으로 *HandleException* 메소드를 호출하고, 이 메소드는 오류가 발생했다는 것을 나타내는 대화 상자를 표시합니다. 다음과 같은 방법으로 *HandleException*을 사용할 수 있습니다.

```
try
  { statements }
except
  Application.HandleException(Self);
end;
```

*EAbort*를 제외한 모든 예외에 대해 *HandleException*은 *OnException* 이벤트 핸들러가 있을 경우 그것을 호출합니다. 따라서, 예외를 처리하고 동시에 CLX처럼 이러한 기본 동작을 제공하려면, 다음과 같이 *HandleException*에 대한 호출을 코드에 추가할 수 있습니다.

```
try
{ special statements }
except
on ESomething do
begin
{ handling for only the special statements }
Application.HandleException(Self);{ call HandleException }
end;
end;
```

**참고** 스레드의 예외 처리 코드 내에서 *HandleException*을 호출하지 마십시오.  
자세한 내용은 도움말 색인에서 exception handling routines를 참조하십시오.

## 조용한 예외

Kylix 애플리케이션은 예외 객체에서 메시지 문자열을 보여주는 메시지를 표시함으로써 사용자의 코드가 특별하게 다루지 않은 대부분의 예외를 처리합니다. 또한 기본적으로 애플리케이션이 오류 메시지를 보여주지 않도록 하는 "조용한" 예외를 정의할 수 있습니다.

조용한 예외는 예외를 사용자에게 보고하지 않고 작업을 중지하고자 할 때 유용합니다. 연산 중지는 *Break* 또는 *Exit* 프로시저를 사용하여 블록을 빠져나가는 것과 유사하지만, 여러 단계로 중첩된 레벨의 블록을 빠져나가지는 못합니다.

조용한 예외는 모두 표준 예외 타입 *EAbort*의 자손입니다. Kylix CLX 애플리케이션용 기본 예외 핸들러는 *EAbort*의 자손을 제외한 모든 예외들에 대해서 오류 메시지 대화 상자를 표시합니다.

**참고** 콘솔 애플리케이션의 경우 오류 메시지 대화 상자는 처리되지 않은 *EAbort* 예외에 표시됩니다.

간단하게 조용한 예외를 만들 수 있는 방법은 다음과 같습니다. 수동으로 객체를 구성하는 대신 *Abort* 프로시저를 호출합니다. 그러면 *Abort*는 자동적으로 *EAbort* 예외를 발생시면서, 오류 메시지를 보여주지 않고 현재 연산을 빠져 나갑니다.

**예** 다음 코드는 작업 중지에 대한 간단한 예를 보여줍니다. 빈 목록 상자와 버튼을 포함하는 폼에서 다음 코드를 버튼의 *OnClick* 이벤트에 첨부합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
var
I:Integer;
begin
for I := 1 to 10 do{ loop ten times }
begin
ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
if I = 7 then Abort;{ abort after the seventh one }
end;
end;
```

## 사용자 고유의 예외 정의

코드를 런타임 라이브러리와 다양한 컴포넌트에 의해 생성된 예외로부터 보호하는 것 이외에, 자신의 코드에서 동일한 메커니즘을 사용하여 사용자 고유의 코드에서 예외 조건을 관리할 수 있습니다.

코드에서 예외를 사용하려면 다음 단계들을 완료해야 합니다.

- 예외 객체 타입 선언
- 예외 발생

### 예외 객체 타입 선언

모든 예외 클래스는 *Exception* 또는 다른 표준 예외 중의 하나로부터 파생되어야 하는 것이 규칙입니다. 이런 방식으로, 그 예외에 대한 특정 예외 핸들러가 보호하지 않는 코드 블록에서 새로운 예외를 발생시키면, 표준 핸들러 중 하나가 이 예외를 대신 처리합니다.

**예** 예를 들면, 다음과 같은 선언을 생각해 봅시다.

```
type
    EMyException = class(Exception);
```

*EMyException*을 발생시키고 그에 대한 특정 핸들러를 제공하지 않으면, *Exception*에 대한 핸들러 또는 기본 예외 핸들러가 그 예외를 처리합니다. *Exception*에 대한 표준 처리가 발생된 예외의 이름을 표시하기 때문에, 발생된 것이 새 예외라는 것을 알 수 있습니다.

### 예외 발생

애플리케이션에서 오류 상태를 나타내기 위해서는, 예외 타입의 인스턴스를 생성하고 예약어 **raise**를 호출하는 것을 포함하는 예외를 발생시키면 됩니다.

예외를 발생시키려면, 예외 객체의 인스턴스 앞에 예약어 **raise**를 호출합니다. 이런 방법으로 특별한 주소에서 오는 예외를 발생시킬 수 있습니다. 예외 핸들러가 실제로 예외를 처리할 때, 예외 인스턴스를 소멸하면 예외가 완료되므로, 그 작업을 직접 수행할 필요는 없습니다.

예외를 발생시키면 시스템 유닛의 *ErrorAddr* 변수를 애플리케이션이 예외를 발생시킨 주소에 설정합니다. 예를 들어, 예외 핸들러에 있는 *ErrorAddr*을 참조하면 오류가 발생한 장소를 사용자에게 알려줄 수 있습니다. 또한 예외가 발생할 때 *ErrorAddr*에 나타나는 **raise**절에 값을 지정할 수 있습니다.



**경고** *ErrorAddr*에 직접 값을 할당하지 마십시오. 이것은 읽기 전용입니다.

예외의 오류 주소를 지정하려면 식별자 같은 주소 표현식 앞에 있는 예외 인스턴스 뒤에 예약어 **at**을 추가합니다.

예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type
  EPasswordInvalid = class(Exception);
```

다음과 같이 *EPasswordInvalid*의 인스턴스를 사용하여 **raise**를 호출하면 언제든지 "password invalid" 예외를 발생시킬 수 있습니다.

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered') at SomeProcedure;
```

## 인터페이스 사용

Kylix의 **interface** 키워드를 사용하면 애플리케이션에서 인터페이스를 만들어 사용할 수 있습니다. 인터페이스는 단일 상속 모델의 CLX를 확장하는 방법으로써, 단일 클래스를 사용하여 둘 이상의 인터페이스를 구현할 수 있고 다른 기반의 자손인 여러 클래스를 사용하여 동일한 인터페이스를 공유할 수 있습니다. 인터페이스는 스트리밍 같은 동일한 집합의 작업을 광범위한 객체에 걸쳐 사용할 때 유용합니다.

## 랭귀지 기능의 인터페이스

인터페이스는 추상 메소드와 기능의 명확한 정의만 포함하는 클래스와 유사합니다. 엄밀하게 말하자면, 인터페이스 메소드 정의에는 매개변수의 개수와 타입, 반환 타입, 예상 동작 등을 포함합니다. 인터페이스 메소드의 이름은 대개는 인터페이스의 목적을 나타내도록 지정됩니다. 인터페이스의 이름을 그 동작에 따라 지정하고 대문자 *I*로 시작하는 것이 규칙입니다. 예를 들어, *IMalloc* 인터페이스는 메모리를 할당, 해제, 관리합니다. 이와 마찬가지로, *IPersist* 인터페이스는 자손들에 대한 일반적인 기본 인터페이스로 사용될 수 있으며, 각 자손은 객체의 상태를 저장 장치나 스트림 또는 파일에 저장하고 로드하는 특정 메소드의 프로토타입을 정의합니다.

인터페이스에는 다음과 같은 구문이 있습니다.

```
IMyObject = interface
  Procedure MyProcedure;
end;
```

다음은 인터페이스를 선언하는 간단한 예입니다.

```
type
  IEdit = interface
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
  end;
```

추상 클래스처럼 인터페이스 자체는 결코 인스턴스화될 수 없습니다. 인터페이스를 사용하려면, 구현 클래스에서 인터페이스를 얻어야 합니다.

인터페이스를 구현하려면 조상 목록에서 인터페이스를 선언하는 클래스를 정의해야 하며, 해당 인터페이스의 모든 메소드를 구현한다는 것을 다음과 같이 표시합니다.

```
TEditor = class(TInterfacedObject, IEdit)
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
end;
```

인터페이스는 메소드의 동작과 시그니처를 정의하지만, 구현을 정의하지는 않습니다. 클래스의 구현이 인터페이스 정의에 따르는 한 인터페이스는 완전한 다형성을 가지며, 인터페이스의 액세스와 사용이 인터페이스의 모든 구현과 동일하다는 것을 의미합니다.

## 계층 구조에 걸친 인터페이스 구현

인터페이스를 사용하면 클래스 사용 방법과 클래스 구현 방법을 구분하기 위한 설계 방법을 제공합니다. 두 가지 클래스는 동일한 기본 클래스의 자손이 아니라도 동일한 인터페이스를 구현할 수 있습니다. 관련이 없는 객체에 대한 동일한 메소드의 이러한 다형적 호출은 객체가 동일한 인터페이스를 구현하는 한 가능합니다. 예를 들어, 다음과 같은 인터페이스를 생각해 봅시다.

```
IPaint = interface
    procedure Paint;
end;
```

그리고 다음 두 가지 클래스도 생각해 봅시다.

```
TSquare = class(TPolygonObject, IPaint)
    procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
    procedure Paint;
end;
```

두 클래스가 공통 조상을 공유하는지 여부에 관계 없이, 두 클래스는 다음과 같이 *IPaint*의 변수와 여전히 호환하는 할당입니다.

```
var
    Painter:IPaint;
begin
    Painter := TSquare.Create;
    Painter.Paint;
    Painter := TCircle.Create;
    Painter.Paint;
end;
```

위의 예는 가상 메소드 *Paint*를 구현한 *TFigure*의 자손인 *TCircle*과 *TSquare*를 가짐으로써 이루어질 수 있습니다. 그 다음에 *TCircle*과 *TSquare*가 *Paint* 메소드를 오버라이드합니다. 위의 예에서 *IPaint*는 *TFigure*로 대체할 수 있습니다. 그러나 다음과 같은 인터페이스를 생각해 봅시다.

```
IRotate = interface
  procedure Rotate(Degrees:Integer);
end;
```

여기서는 원이 아닌 사각형을 지원하도록 하는 것이 바람직합니다. 클래스는 다음과 같습니다.

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees:Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

나중에 *TFilledCircle* 클래스를 만들 수 있으며, *IRotate* 인터페이스를 구현하면 단순 원에 회전을 추가하지 않고 원을 채우는 데 사용되는 패턴을 회전할 수 있습니다.

**참고** 이러한 예들에서, 직접 기본 클래스 또는 조상 클래스는 참조 카운팅을 관리하는 *IInterface* 메소드를 구현한 것으로 가정할 수 있습니다. 자세한 내용은 4-20 페이지의 "IInterface 구현" 및 4-23 페이지의 "인터페이스 객체의 메모리 관리"를 참조하십시오.

## 프로시저와 함께 인터페이스 사용

또한 인터페이스를 통해 객체가 특별한 기본 클래스의 자손이 아니라도 객체를 처리할 수 있는 일반적인 프로시저를 작성할 수 있습니다. 위의 *IPaint*와 *IRotate* 인터페이스를 사용하여 다음과 같은 프로시저를 작성할 수 있습니다.

```
procedure PaintObjects(Painters: array of IPaint);
var
  I:Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I:Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

*RotateObjects*를 사용하면 객체를 저질로 그려지게 하지 않아도 되며, *PaintObjects*를 사용하면 객체를 회전시키는 방법을 몰라도 됩니다. 이러한 이유에서 *TFigure* 클래스에 대해서만 작동하도록 작성된 경우보다 위의 일반적인 프로시저를 더 자주 사용할 수 있습니다.

인터페이스의 구문, 랭귀지 정의 및 규칙에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말의 Object interfaces 단원을 참조하십시오.

## Interface 구현

---

모든 인터페이스는 *IInterface* 인터페이스로부터 직접 또는 간접적으로 파생됩니다. 이 인터페이스는 동적 쿼리와 수명 관리 (lifetime management) 라는 핵심적인 인터페이스 기능을 제공합니다. 이러한 기능은 다음 세 가지 *IInterface* 메소드로 이루어집니다.

- *QueryInterface* 는 주어진 객체에 동적으로 쿼리하고 그 객체가 지원하는 인터페이스에 대한 인터페이스 참조를 얻을 수 있는 메소드를 제공합니다.
- *\_AddRef* 는 *QueryInterface* 호출이 이루어질 때마다 카운트를 증가시키는 참조 카운팅 메소드입니다. 참조 카운트가 0이 아니면 객체는 메모리에 남아 있게 됩니다.
- *\_Release* 는 *\_AddRef* 와 함께 객체가 자신의 수명을 추적하고 객체 자체를 안전하게 삭제하는 시기를 결정하는 데 사용됩니다. 참조 카운트가 일단 0이 되면 객체는 메모리에서 해제됩니다.

인터페이스를 구현하는 모든 클래스는 다른 조상 인터페이스가 선언한 모든 메소드, 인터페이스 자체가 선언한 모든 메소드 뿐만 아니라 세 가지 *IInterface* 메소드도 구현해야 합니다. 그러나 개발자는 클래스에서 선언한 인터페이스의 메소드 구현을 상속할 수 있습니다.

자신이 직접 이러한 메소드들을 구현함으로써, 수명 관리의 대체 수단을 제공하여 참조 카운팅 사용 불가능으로 설정할 수 있습니다. 이 방법은 인터페이스를 참조 카운팅에서 분리할 수 있는 강력한 기술입니다.

## TInterfacedObject

---

CLX는 *IInterface*의 메소드를 구현하기 때문에 편리하게 기본 클래스로 사용할 수 있는 일반 (simple) 클래스 *TInterfacedObject*를 정의합니다. *TInterfacedObject* 클래스는 다음과 같이 *시스템* 유닛에 선언됩니다.

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

*TInterfacedObject*에서 직접 파생할 수 있습니다. 다음과 같은 선언 예제에서, *TDerived*는 *TInterfacedObject*의 직접적인 자손이고 가상적인 *IPaint* 인터페이스를 구현합니다.

```
type
  TDerived = class(TInterfacedObject, IPaint)
  ...
end;
```

여기서 *IInterface*의 메소드를 구현하기 때문에, *TInterfacedObject*는 자동적으로 참조 카운팅과 인터페이스 객체의 메모리 관리를 처리합니다. 자세한 내용은 4-23 페이지의 "인터페이스 객체의 메모리 관리"를 참조하십시오. 여기서는 또한 인터페이스를 구현하는 사용자 고유의 클래스를 작성하는 것에 대해 설명하지만, *TInterfacedObject*에서의 참조 카운팅 메커니즘을 따르지는 않습니다.

## as 연산자 사용

인터페이스를 구현하는 클래스는 **as** 연산자를 인터페이스에서의 동적 바인딩에 사용할 수 있습니다. 다음과 같은 예를 들 수 있습니다.

```
procedure PaintObjects(P:TInterfacedObject)
var
  X:IPaint;
begin
  X := P as IPaint;
  { statements }
end;
```

여기서, *TInterfacedObject* 타입의 변수 *P*는 *IPaint* 인터페이스 참조인 변수 *X*에 할당될 수 있습니다. 동적 바인딩은 이러한 할당을 가능하게 만듭니다. 컴파일러는 이러한 할당에 대해 *P*의 *IInterface* 인터페이스의 *QueryInterface* 메소드를 호출하는 코드를 생성합니다. 왜냐하면 컴파일러는 *P*의 선언 타입으로는 *P*의 인스턴스가 실제로 *IPaint*를 지원하는지 알 수 없기 때문입니다. 런타임에 *P*는 *IPaint* 참조로 확인되거나 그렇지 않으면 예외가 발생합니다. 어떤 경우라도 *P*를 *X*에 할당하는 것은 *P*가 *IInterface*를 구현하지 않은 클래스 타입일 때처럼 컴파일 타임 오류를 발생시키지 않습니다.

인터페이스에서 **as** 연산자를 동적 바인딩에 사용할 때, 다음과 같은 요구 사항을 알고 있어야 합니다.

- 명시적인 *IInterface* 선언: 모든 인터페이스가 *IInterface*에서 파생된다 하더라도, **as** 연산자를 사용하고자 한다면 클래스가 단순히 *IInterface*의 메소드를 구현하는 것으로는 불충분합니다. 이러한 상황은 클래스가 명시적으로 선언한 인터페이스를 구현하는 경우에도 마찬가지입니다. 클래스는 인터페이스 목록 안에 *IInterface*를 명시적으로 선언해야 합니다.
- IID 사용: 인터페이스는 GUID(globally unique identifier)에 기반한 식별자를 사용할 수 있습니다. GUID는 인터페이스를 식별하는 데 사용되므로 인터페이스 식별자(IID)라고 합니다. 인터페이스에 **as** 연산자를 사용하고 있을 경우에는 연결된 IID가 있어야 합니다. 소스 코드에서 새 GUID를 만들기 위해 *Ctrl+Shift+G* 편집 단축키를 사용할 수 있습니다.

## 코드 재사용 및 위임(Delegation)

인터페이스와 함께 코드를 재사용하는 한 가지 방법은 어떤 객체를 포함하거나 다른 객체에 포함된 객체를 갖도록 하는 것입니다. CLX는 객체 타입인 속성을 포함 관계와 코드 재사용에 대한 방법으로 사용합니다. 이러한 디자인을 인터페이스용으로 지원하기 위한 목적으로, Kylix에는 하위 객체에 대한 인터페이스의 전부 또는 일부를 위임하는 코드를 쉽게 작성할 수 있는 키워드 **implements**가 들어 있습니다.

### 위임을 위한 implements 사용

CLX에 있는 많은 클래스에는 하위 객체인 속성들이 있습니다. 인터페이스는 속성 타입으로도 사용할 수 있습니다. 속성이 인터페이스 타입 또는 인터페이스의 메소드를 구현하는 클래스 타입일 경우, 키워드 **implements**를 사용하여 그 인터페이스의 메소드가 속성 인스턴스인 인터페이스 참조 또는 객체에 위임되는지를 지정할 수 있습니다. 위임은 메소드에 대한 구현만 제공하면 됩니다. 인터페이스 지원을 선언할 필요는 없습니다. 속성을 포함하는 클래스는 조상 목록에 인터페이스를 포함해야 합니다.

기본적으로 키워드 **implements**를 사용하면 모든 인터페이스 메소드를 위임합니다. 하지만 메소드 확인 절을 사용하거나 클래스에서 이러한 기본 동작을 오버라이드하기 위해서 일부 인터페이스 메소드를 구현하는 메소드를 선언할 수도 있습니다.

다음 예에서는 8비트 RGB 색상 값을 *Color* 참조로 변환하는 색상 어댑터 객체의 디자인에서 **implements** 키워드를 사용합니다.

```
unit cadapt;

type
  IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red:Byte;
    function Green:Byte;
    function Blue:Byte;
  end;

  IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color:Integer;
  end;

{ TRGB8ColorRefAdapter  map an IRGB8bit to an IColorRef }
TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
private
  FRGB8bit:IRGB8bit;
  FPalRelative:Boolean;
public
  constructor Create(rgb:IRGB8bit);
  property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
  property PalRelative: Boolean read FPalRelative write FPalRelative;
  function Color:Integer;
end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb:IRGB8bit);
begin
```

```

    FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color:Integer;
begin
    if FPalRelative then
        Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
    else
        Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
    end;
end.

```

**implements** 키워드의 구문, 구현 세부 사항, 랭귀지 규칙 등에 대한 자세한 내용은 *오픈팩트 파스칼 랭귀지 안내서* 온라인 도움말의 Object interfaces 단원을 참조하십시오.

## 인터페이스 객체의 메모리 관리

---

인터페이스 디자인의 목적 중 하나는 인터페이스를 구현하는 객체의 수명 관리를 확실히 하는 데 있습니다. *IInterface*의 *\_AddRef* 메소드와 *\_Release* 메소드는 이러한 기능을 구현하는 방법을 제공합니다. 이 메소드들의 정의된 동작은 인터페이스 참조가 클라이언트에 전달될 때 객체에 대한 참조 카운트를 늘려서 객체의 수명을 추적하고, 참조 카운트가 0일 때 객체를 소멸하는 것입니다.

## 참조 카운팅 사용

Kylix는 인터페이스 쿼리와 참조 카운팅의 구현을 통해 많은 *IInterface* 메모리 관리를 제공합니다. 따라서, 인터페이스에 의해 존재하고 소멸하는 객체를 가지고 있는 경우, 이러한 클래스로부터 파생시켜서 참조 카운팅을 쉽게 사용할 수 있습니다. *TInterfacedObject*는 이러한 동작을 제공하는 클래스입니다. 참조 카운팅을 사용하기로 결정했다면, 객체를 인터페이스 참조로만 유지하고 참조 카운팅이 일관성을 갖도록 유의해야 합니다. 예를 들면,

```

procedure beep(x:ITest);

function test_func()
var
    y:ITest;
begin
    y := TTest.Create; // because y is of type ITest, the reference count is one
    beep(y); // the act of calling the beep function increments the reference count
              // and then decrements it when it returns
    y.something; // object is still here with a reference count of one
end;

```

위의 예는 메모리 관리에 대한 가장 깨끗하고 안전한 방법이고, *TInterfacedObject*를 사용하는 경우에는 자동으로 처리됩니다. 이러한 규칙을 따르지 않으면, 객체는 다음 코드에서 볼 수 있듯이 갑자기 사라질 수 있습니다.

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
  // and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

**참고** 위의 예에서 *beep* 프로시저는 선언된대로 매개변수에 대한 참조 카운트(*AddRef* 호출)를 증가시키지만, 다음 선언에서는 증가시키지 않습니다.

```
procedure beep(const x:ITest);
```

또는

```
procedure beep(var x:ITest);
```

이러한 선언은 더 작고 더 빠른 코드를 생성합니다.

참조 카운팅을 일관성 있게 적용할 수 없어서 사용할 수 없는 경우가 있다면 객체가 컴포넌트이거나 다른 컴포넌트에 의해 소유된 컨트롤인 경우입니다. 그러한 상황에서는, 인터페이스를 사용할 수도 있지만, 그 객체의 수명이 인터페이스에 의해 결정되지 않기 때문에 참조 카운팅을 사용하지 말아야 합니다.

## 참조 카운팅을 사용하지 않음

객체가 CLX 컴포넌트이거나 다른 컴포넌트가 소유한 컨트롤인 경우, 객체는 *TComponent*에 기반한 다른 메모리 관리 시스템의 일부가 됩니다. CLX 컴포넌트의 객체 수명 관리 방법과 인터페이스 참조 카운팅을 혼합해서는 안됩니다. 인터페이스를 지원하는 객체를 만들고자 한다면, 다음과 같이 *IInterface* *\_AddRef* 및 *\_Release* 메소드를 빈 함수로 구현하여 인터페이스 참조 카운팅 메커니즘을 피할 수 있습니다.

```
function TMyObject._AddRef:Integer;
begin
  Result := -1;
end;

function TMyObject._Release:Integer;
begin
  Result := -1;
end;
```

이 경우에도 여전히 *QueryInterface*를 구현하여 객체에 대한 동적 쿼리를 제공할 것입니다. 여기서 *QueryInterface*를 구현하기 때문에, 인터페이스 식별자(IID)를 만들 동안 컴포넌트에서 인터페이스를 위한 **as** 연산자를 사용할 수 있습니다.

*TComponent*에서 이 메소드들을 이러한 방법으로 미리 구현하므로 객체가 *TComponent* 또는 그 자손에 기반하는 경우에는 구현할 필요가 없습니다.



## 문자열 작업

Kylix에는 오브젝트 파스칼 언어의 개발을 통해 도입된 많은 다른 문자와 문자열 타입이 있습니다. 이 단원은 이러한 타입, 용도 및 사용에 대한 개요를 담고 있습니다. 언어에 대한 자세한 내용은 오브젝트 파스칼 언어 온라인 도움말의 String types를 참조하십시오.

### 문자 타입(Character types)

Kylix에는 다음 세 가지 문자 타입인 *Char*, *AnsiChar*, *WideChar*가 있습니다.

*Char* 문자 타입은 표준 파스칼에서 유래하여 터보 파스칼에서 사용되고 나서 오브젝트 파스칼에서 사용되었습니다. 나중에 오브젝트 파스칼은 문자 표현에 대한 표준을 지원하는 특정 문자 타입으로 *AnsiChar*와 *WideChar*를 추가했습니다. *AnsiChar*는 8비트 문자를 지원하며 *WideChar*는 16비트 유니코드 표준을 지원합니다. *WideChar*란 이름은 유니코드 문자가 와이드 문자로 알려져 있어서 그렇게 사용합니다. 와이드 문자는 한 바이트가 아니라 두 바이트이므로 문자 집합으로 훨씬 많은 다른 문자들을 표현할 수 있습니다. *AnsiChar*와 *WideChar*를 사용하게 되면서, *Char*는 현재 권장되는 구현을 나타내는 기본 문자 타입이 되었습니다.

#### 참고

Linux `wchar_t` `widechar`는 문자당 23비트입니다. 오브젝트 파스칼 `widechar`가 지원하는 16비트 유니코드 표준은 Linux와 GNU(Gnu is Not Unix) 라이브러리가 지원하는 32비트 UCS(Universal Multiple-Octet Coded Character Set) 표준의 하위 집합입니다. 파스칼 `widechar` 데이터는 `wchar_t`로 OS 함수로 전달되기 전에 문자당 32비트로 확장되어야 합니다.

다음 표에서는 이러한 문자 타입에 대해 요약합니다.

**표 4.2** 오브젝트 파스칼 문자 타입

| 타입       | 바이트 | 내용         | 용도           |
|----------|-----|------------|--------------|
| Char     | 1   | 단일 문자      | 기본 문자 타입     |
| AnsiChar | 1   | 단일 문자      | 8비트 문자       |
| WideChar | 2   | 단일 유니코드 문자 | 16비트 유니코드 표준 |

이러한 문자 타입 사용에 대한 자세한 내용은 *오브젝트 파스칼 언어 안내서* 온라인 도움말의 Character types를 참조하십시오. 유니코드 문자에 대한 자세한 내용은 *오브젝트 파스칼 언어 안내서* 온라인 도움말의 About extended character sets를 참조하십시오.

### 문자열 타입(String types)

Kylix에는 다음 세 가지 문자열 타입이 있습니다.

- 문자 포인터
- 문자열 타입
- 문자열 클래스

이 단원에서는 문자열 타입에 대해 요약하고, 문자열 포인터와 함께 문자열 타입 사용에 대해 설명합니다. 문자열 클래스 사용에 대한 내용은 온라인 도움말의 *TStrings*를 참조하십시오.

Kylix에는 세 가지의 문자열 구현, 즉 짧은 문자열, 긴 문자열, 와이드 문자열이 들어 있습니다. 몇 가지 다른 문자열 타입은 이러한 구현을 표현합니다. 그 외에, 예약어 **string**은 현재 권장하는 문자열 구현의 기본값입니다.

## 짧은 문자열

**String**은 터보 파스칼에서 사용되었던 첫 번째 문자열 타입입니다. **String**은 원래 짧은 문자열로 구현되었습니다. 짧은 문자열은 1바이트에서 256바이트까지 할당되고, 그 중 첫 번째 바이트에는 문자열의 길이에 대한 정보가 들어 있으며 나머지 바이트에는 문자열의 문자가 들어 있습니다.

**S: string[0..n]**// the original string type

긴 문자열을 구현할 때, **string**은 기본으로 긴 문자열 구현으로 바뀌고 *ShortString*이 역 호환성 타입으로 도입됩니다. 다음과 같이 *ShortString*은 최대 길이 문자열에 대한 이미 정의된 타입입니다.

**S: string[255]**// the ShortString type

*ShortString*에 대해 할당되는 메모리의 크기는 정적이고, 이는 크기가 컴파일 타임에 결정된다는 것을 의미합니다. 하지만 예를 들어 *ShortString*에 대한 포인터인 *PShortString*를 사용하는 경우에 *ShortString*에 대한 메모리 위치가 동적으로 할당될 수 있습니다. 짧은 문자열 타입 변수가 차지한 저장 장치의 바이트 수는 짧은 문자열 타입의 최대 길이에 한 바이트를 더한 것이 됩니다. 이미 정의된 *ShortString* 타입의 크기는 256바이트입니다.

구문 **string[0..n]**으로 선언된 짧은 문자열과 이미 정의된 *ShortString* 타입은 모두 이전 버전의 Kylix 및 Borland 파스칼과의 역 호환성을 위해 주로 존재합니다.

컴파일러 지시어 \$H는 예약어 **string**이 짧은 문자열 또는 긴 문자열을 나타내는지 여부를 제어합니다. 기본 상태에서 { \$H+ }, **string**은 긴 문자열을 나타냅니다. { \$H- } 지시어를 사용하여 긴 문자열을 *ShortString*으로 변경할 수 있습니다. { \$H- } 상태는 기본으로 짧은 문자열을 사용하는 오브젝트 파스칼 버전의 코드를 사용하는 데 가장 유용합니다. 하지만 짧은 문자열은 고정된 크기의 컴포넌트가 필요한 데이터 구조에서 유용하게 사용될 수 있습니다. 짧은 문자열의 생성을 확인하기 위해 문자열 타입 정의의 의미를 지역적으로 오버라이드할 수 있습니다. 또한 짧은 문자열 타입의 선언을 명확하게 \$H 설정에 무관한 **string[255]** 또는 *ShortString*으로 변경할 수 있습니다.

짧은 문자열과 *ShortString* 타입에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말의 Short strings를 참조하십시오.

## 긴 문자열

긴 문자열은 동적으로 할당되는 문자열로서 최대 길이가 2기가 바이트입니다. 짧은 문자열과 마찬가지로, 긴 문자열은 8비트 문자를 사용하고 길이 지시자를 가집니다. 짧은 문자열과 달리, 긴 문자열에는 동적 문자열 길이를 포함하는 영순위 요소가 없습니다. 긴 문자열의 길이를 알아내려면 *Length* 표준 함수를 사용해야 하고, 긴 문자열의 길이를 설정하려면 *SetLength* 표준 프로시저를 사용해야 합니다. 또한 긴 문자열은 참조 카운트되고, *PChars* 처럼 Null 종료됩니다. 긴 문자열의 구현에 대한 자세한 내용은 Long strings에 대한 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말을 참조하십시오.

긴 문자열은 예약어 **string**과 이미 정의된 식별자 *AnsiString*로 표시합니다. 새 애플리케이션에서 긴 문자열 타입을 사용하는 것이 좋습니다. Visual Component Library의 모든 컴포넌트는 이러한 상태에서 일반적으로 **string**을 사용하여 컴파일됩니다. 컴포넌트를 작성하는 경우에는, 문자열 타입 속성에서 데이터를 받는 모든 코드에서와 마찬가지로 긴 문자열을 사용해야 합니다. 항상 긴 문자열을 사용하는 특정 코드를 작성하려면, *AnsiString*을 사용해야 합니다. 새 문자열 구현이 표준이 될 때 타입을 쉽게 변경할 수 있는 융통성 있는 코드를 작성하고자 한다면, **string**을 사용해야 합니다.

## WideString

*WideChar* 타입을 통해 와이드 문자열을 *WideChars*의 배열로 나타낼 수 있습니다. 와이드 문자열은 16비트 유니코드 문자로 구성된 문자열입니다. 긴 문자열과 마찬가지로 *WideStrings*은 최대 2기가 바이트의 길이로 동적으로 할당되고 참조 카운트됩니다. *WideString* 타입은 이미 정의된 식별자 *WideString*에 의해 표시됩니다.

*WideStrings*에 대한 자세한 내용은 *WideString*에 대한 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말을 참조하십시오.

## PChar 타입

*PChar*는 *Char* 타입의 Null 종료 문자열에 대한 포인터입니다. 또한 세 가지 문자 타입 모두 다음과 같이 기본 제공 포인터 타입을 가집니다.

- *PChar*는 8비트 문자의 Null 종료 문자열에 대한 포인터입니다.
- *PAnsiChar*는 8비트 문자의 Null 종료 문자열에 대한 포인터입니다.
- *PWideChar*는 16비트 문자의 Null 종료 문자열에 대한 포인터입니다.

*PChars*는 짧은 문자열과 함께 원래의 오브젝트 파스칼 문자열 타입 중 하나입니다.

## OpenString

*OpenString*은 이제 사용하지 않지만 예전의 코드에서는 볼 수 있습니다. 이 문자열 타입은 16비트 호환이고 매개변수에서만 사용할 수 있습니다. *OpenString*은 긴 문자열이 구현되기 전에 길이가 지정되지 않은 짧은 문자열을 매개변수로 전달하는 데 사용되었습니다. 예를 들면,

```
procedure a(v : openstring);
```

어떠한 길이의 문자열도 매개변수로 전달될 수 있으며, 여기서 실제 매개변수와 정식 매개변수의 문자열 길이가 정확하게 일치해야 합니다. 자신이 작성하는 새 애플리케이션에서는 *OpenString*을 사용하면 안됩니다.

또한 4-35 페이지의 "문자열에 대한 컴파일러 지시어"에서 {\$P+/-} 스위치를 참조하십시오.

## 런타임 라이브러리 문자열 처리 루틴

---

런타임 라이브러리는 어떤 문자열 타입에만 해당되는 많은 특화된 문자열 처리 루틴을 제공합니다. 이 루틴들은 와이드 문자열, 긴 문자열, Null 종료 문자열 (*PChars*를 의미함)을 위한 것입니다. *PChar* 타입을 처리하는 루틴은 Null 종료를 사용하여 문자열 길이를 결정합니다. Null 종료 문자열에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서* 온라인 도움말에서 Working with null-terminated strings를 참조하십시오.

또한 런타임 라이브러리에는 문자열 서식 루틴의 범주가 포함됩니다. *ShortString* 타입에 대한 루틴 범주는 없습니다. 그러나, 일부 기본 제공 컴파일러는 *ShortString* 타입을 처리합니다. 예를 들어 여기에는 *Low* 표준 함수와 *High* 표준 함수가 포함됩니다.

와이드 문자열과 긴 문자열은 일반적으로 사용되는 타입이므로 나머지 단원에서 이러한 루틴들을 설명합니다.

## 와이드 문자 루틴

문자열을 사용할 때 애플리케이션에 있는 코드가 다양한 대상 로케일에서 접하게 될 문자열을 처리할 수 있는지 확인해야 합니다. 가끔 와이드 문자와 와이드 문자열을 사용해야 할 때가 있습니다. 실제로, 표의 문자 집합을 사용하는 한 가지 방법은 모든 문자를 유니코드 같은 와이드 문자 인코딩 구성으로 변환하는 것입니다. CLX 컴포넌트는 모든 문자열 값을 와이드 문자열로 나타냅니다.

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.

## 일반적으로 사용되는 긴 문자열 루틴

긴 문자열 처리 루틴은 여러 함수 영역을 처리합니다. 이 영역 중 일부는 동일한 목적으로 사용되는데, 처리 루틴들 간의 차이는 계산에서의 특정 기준 사용 여부입니다. 나중에 나오는 테이블에는 이러한 함수 영역별 루틴을 나열합니다.

- 비교
- 대소문자 변환
- 수정
- 하위 문자열
- 문자열 처리

적당한 곳에서 테이블은 또한 루틴이 다음 기준을 만족하는지 여부를 나타내는 열을 제공합니다.

- 대소문자 구별 사용: 로케일 설정이 사용되면 대소문자의 정의를 결정합니다. 루틴이 로케일 설정을 사용하지 않으면, 분석은 문자의 순서 값에 기반합니다. 루틴이 대소문자를 구별하지 않으면, 이미 정의된 패턴에 의해 결정된 대소문자를 논리적으로 병합합니다.
- 로케일 설정 사용: 로케일 설정을 통해 특정한 로케일에 대한 애플리케이션을 사용자 지정할 수 있습니다. 특히, 아시아 랭귀지 환경에서 그렇습니다. 대부분의 로케일 설정에서는 소문자가 해당 대문자보다 값이 더 적다고 간주합니다. 이것은 소문자가 대문자보다 값이 더 큰 ASCII 순서와는 반대입니다.
- MBCS(멀티바이트 문자 집합 지원): MBCS는 극동 아시아 로케일용 코드를 작성할 때 사용됩니다. 멀티바이트 문자는 한 바이트에서 여섯 바이트 문자 코드의 혼합으로 표현되므로, 바이트 길이가 반드시 문자열의 길이에 해당되는 것은 아닙니다. MBCS를 지원하는 루틴은 문자를 구문 분석하기 위해 작성됩니다. *ByteType*과 *StrByteType*은 두 바이트 이상 문자에서의 선행 바이트를 결정합니다. 멀티바이트 문자를 사용할 때 멀티바이트 문자가 절반으로 잘리지 않도록 유의하십시오. 문자 크기는 미리 결정할 수 없으므로 문자를 함수 또는 프로시저에 매개변수로 전달하지 마십시오. 대신 문자 또는 문자열에 대한 포인터를 전달합니다. MBCS에 대한 자세한 내용은 12장 "국제적인 애플리케이션 만들기"의 12-2 페이지의 "애플리케이션 코드 활성화"를 참조하십시오.

### 참고

최신 버전의 glibc에는 제한이 있기 때문에, 대형 문자열(50K 이상)을 멀티바이트 사용이 가능 루틴에 전달하지 마십시오. 이후 버전의 glibc에는 이러한 제한을 두지 않을 것입니다.

표 4.3 문자열 비교 루틴

| 루틴                  | 대소문자 구별 | 로케일 설정 사용 | MBCS 지원 |
|---------------------|---------|-----------|---------|
| AnsiCompareStr      | 예       | 예         | 예       |
| AnsiCompareText     | 아니오     | 예         | 예       |
| AnsiCompareFileName | 예       | 예         | 예       |
| CompareStr          | 예       | 아니오       | 아니오     |
| CompareText         | 아니오     | 아니오       | 아니오     |

**표 4.4** 대소문자 변환 루틴

| 루틴                    | 로케일 설정 사용 | MBCS 지원 |
|-----------------------|-----------|---------|
| AnsiLowerCase         | 예         | 예       |
| AnsiLowerCaseFileName | 예         | 예       |
| AnsiUpperCaseFileName | 예         | 예       |
| AnsiUpperCase         | 예         | 예       |
| LowerCase             | 아니오       | 아니오     |
| UpperCase             | 아니오       | 아니오     |

문자열 파일 이름에 사용되는 루틴: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, *AnsiUpperCaseFileName*은 모두 로케일 설정을 사용합니다. 파일 이름에 사용된 로케일 (문자 집합)은 기본 사용자 인터페이스와 다를 수 있으므로 항상 포팅 가능한 파일 이름을 사용해야 합니다. 대부분의 포팅 가능한 파일 이름은 간단한 ASCII이고 특수 문자를 포함하지 않습니다. 애플리케이션이 생성되면 파일 이름은 모두 소문자로 이루어져야 합니다. 사용자 입력의 경우에 파일 이름은 사용자가 입력한 것과 대소문자까지 똑같아야 합니다.

**표 4.5** 문자열 수정 루틴

| 루틴               | 대소문자 구별 | MBCS 지원 |
|------------------|---------|---------|
| AdjustLineBreaks | 해당없음    | 예       |
| AnsiQuotedStr    | 해당없음    | 예       |
| StringReplace    | 플래그별 옵션 | 예       |
| Trim             | 해당없음    | 예       |
| TrimLeft         | 해당없음    | 예       |
| TrimRight        | 해당없음    | 예       |
| WrapText         | 해당없음    | 예       |

**표 4.6** 하위 문자열 루틴

| 루틴                   | 대소문자 구별 | MBCS 지원 |
|----------------------|---------|---------|
| AnsiExtractQuotedStr | 해당없음    | 예       |
| AnsiPos              | 예       | 예       |
| IsDelimiter          | 예       | 예       |
| IsPathDelimiter      | 예       | 예       |
| LastDelimiter        | 예       | 예       |
| QuotedStr            | 아니오     | 아니오     |

표 4.7 문자열 처리 루틴

| 루틴                | 대소문자 구별 | MBCS 지원 |
|-------------------|---------|---------|
| AnsiContainsText  | 아니오     | 예       |
| AnsiEndsText      | 아니오     | 아니오     |
| AnsiIndexText     | 아니오     | 예       |
| AnsiMatchText     | 아니오     | 예       |
| AnsiResemblesText | 아니오     | 아니오     |
| AnsiStartsText    | 아니오     | 예       |
| IfThen            | 해당없음    | 예       |
| LeftStr           | 예       | 아니오     |
| RightStr          | 예       | 아니오     |
| SoundEx           | 해당없음    | 아니오     |
| SoundExInt        | 해당없음    | 아니오     |
| DecodeSoundExInt  | 해당없음    | 아니오     |
| SoundExWord       | 해당없음    | 아니오     |
| DecodeSoundExWord | 해당없음    | 아니오     |
| SoundExSimilar    | 해당없음    | 아니오     |
| SoundExCompare    | 해당없음    | 아니오     |

## 문자열 선언 및 초기화

긴 문자열은 다음과 같이 선언합니다.

```
S:string;
```

긴 문자열을 초기화할 필요는 없습니다. 긴 문자열은 비어있는 것으로 자동적으로 초기화됩니다. 문자열이 비어있는지 검사하려면 다음과 같이 *EmptyStr* 변수를 사용합니다.

```
S = EmptyStr;
```

아니면, 다음과 같이 빈 문자열을 검사합니다.

```
S = '';
```

빈 문자열에는 유효한 데이터가 없습니다. 따라서 다음과 같이 빈 문자열을 인덱스 참조하려는 것은 *nil*에 액세스하려는 것과 같고 액세스 위반을 초래합니다.

```
var
  S:string;
begin
  S[i]; // this will cause an access violation
  // statements
end;
```

이와 마찬가지로, 빈 문자열을 *PChar*로 변환하면, 그 결과는 **nil** 포인터가 됩니다. 따라서, 그러한 *PChar*를 읽거나 써야 하는 루틴에 전달할 경우에는 다음과 같이 루틴이 **nil**을 처리할 수 있는지 확인하십시오.

```
var
  S: string; // empty string
begin
  proc(PChar(S)); // be sure that proc can handle nil
  // statements
end;
```

만일 루틴이 처리할 수 없다면 다음과 같이 문자열을 초기화합니다.

```
S := 'No longer nil';
proc(PChar(S)); // proc does not need to handle nil now
```

아니면 다음 같이 *SetLength* 프로시저를 사용하여 길이를 설정합니다.

```
SetLength(S, 100); // sets the dynamic length of S to 100
proc(PChar(S)); // proc does not need to handle nil now
```

*SetLength*를 사용할 경우 문자열의 기존 문자는 유지되지만 모든 새로 할당된 공간의 내용은 정의되어 있지 않습니다. *SetLength*를 호출한 다음, *S*는 참조 카운트가 1인 고유한 문자열을 반드시 참조합니다. 문자열의 길이를 알려면 *Length* 함수를 사용합니다.

다음과 같이 문자열을 선언하는 경우를 생각해 봅니다.

```
S: string[n];
```

이 구문은 *n* 길이의 긴 문자열이 아닌 짧은 문자열을 암시적으로 선언합니다. 특별히 *n* 길이의 긴 문자열을 선언하려면 **string** 타입의 변수를 선언하고 *SetLength* 프로시저를 사용하십시오.

```
S: string;
SetLength(S, n);
```

## 문자열 타입의 혼합 및 변환

짧은 문자열, 긴 문자열, 와이드 문자열은 할당문과 표현식에서 혼합할 수 있고, 컴파일러는 필요한 문자열 타입 변환을 수행하기 위한 코드를 자동으로 생성합니다. 하지만 문자열 값을 짧은 문자열 변수에 할당할 때, 문자열 값이 짧은 문자열 변수의 선언된 최대 길이보다 더 긴 경우에는 문자열 값이 잘린다는 점에 유의하십시오.

긴 문자열은 미리 동적으로 할당됩니다. *PAnsiString*, *PString* 또는 *PWideString* 같은 기본 제공 포인터 타입 중의 하나를 사용하는 경우 간접적인 또 다른 레벨의 방법을 도입하는 것임을 알아야 합니다. 자신이 의도한대로 되었는지 확인하십시오.

기본으로 하는 Qt 문자열 타입과 Kylix 문자열 타입을 변환하기 위해 추가 함수(*Copy QStringListToTstrings*, *Copy TStringsToQStringList*, *QStringListToTStringList*)를 제공합니다. 이러한 함수들은 *Qtypes.pas*에 있습니다.



## 문자열에서 PChar로 변환

긴 문자열은 *PChar*로 자동 변환되지 않습니다. 문자열과 *PChars* 간의 몇 가지 차이로 인해 다음과 같은 변환 문제점이 야기될 수 있습니다.

- 긴 문자열은 참조 카운트되지만 *PChars*는 참조 카운트되지 않습니다.
- 문자열에 할당하면 데이터를 복사하지만, *PChar*가 메모리에 대한 포인터입니다.
- 긴 문자열은 Null 종료되고 또한 문자열의 길이를 포함하지만, *PChars*는 단순히 Null 종료됩니다.

이러한 차이로 인해 미묘한 오류가 발생할 수 있는 상황을 이 단원에서 설명합니다.

### 문자열 종속성

예를 들면 *PChar*를 취하는 함수를 사용할 때, 긴 문자열을 Null 종료된 문자열로 변환해야 할 경우가 있을 것입니다. 문자열을 *PChar*로 변환해야 하는 경우, 결과 *PChar*의 수명을 관리해야 한다는 점에 유의하십시오. 긴 문자열이 참조 카운트되기 때문에, 문자열을 *PChar*로 타입 변환하면 실제로 참조 카운트를 증가시키지 않고 문자열에 대한 종속을 하나 증가시킵니다. 참조 카운트가 0이면, 문자열은 이에 대한 추가 종속이 있더라도 소멸됩니다. 이 문자열을 전달하는 루틴이 계속 이 타입 변환을 이용하고 있는 동안 *PChar* 변환이 사라질 수도 있습니다. 예를 들면, 다음과 같습니다.

```
procedure my_func(x:string);
begin
    // do something with x
    some_proc(PChar(x)); // cast the string to a PChar
    // you now need to guarantee that the string remains
    // as long as the some_proc procedure needs to use it
end;
```

### PChar 지역 변수 반환

*PChars*를 사용할 때의 공통적으로 나타나는 오류는 지역 변수를 데이터 구조에 저장하거나 값으로 반환하는 것입니다. *PChar*는 단순히 메모리에 대한 포인터이지 문자열의 참조 카운트된 복사가 아니기 때문에 루틴이 종료하면 사라집니다. 예를 들면, 다음과 같습니다.

```
function title(n:Integer):PChar;
var
    s:string;
begin
    s := Format('title - %d', [n]);
    Result := PChar(s); // DON'T DO THIS
end;
```

이 예에서는 *title* 함수가 반환될 때 해제되는 문자열 데이터에 대한 포인터를 반환합니다.

## 지역 변수를 Pchar 로 전달

*PChar*를 사용하는 함수를 호출하여 초기화해야 하는 지역 문자열 변수를 가지고 있는 경우를 생각해 봅시다. 한 가지 방법은 다음과 같이 지역 **char 타입 지역 배열**을 만들어 함수에 전달한 다음 지역 문자열 변수를 문자열에 할당하는 것입니다.

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
    . . .
end;

// assume MAXSIZE is a predefined constant
var
    i:Integer;
    buf: array[0..MAX_SIZE] of char;
    S:string;
begin
    i := FillBuffer(0, @buf, SizeOf(buf)); // treats @buf as a PChar
    S := buf;
    // statements
end;
```

이 방법은 버퍼가 스택에 할당되기 때문에 버퍼 크기가 상대적으로 작은 경우에 유용합니다. 또한 **array of char 타입**과 **string 타입** 간의 변환이 자동 수행되므로 이 방법은 안전합니다. *FillBuffer*가 반환되면, 문자열의 *Length*는 *buf*에 쓰인 바이트 수를 정확하게 표시합니다.

버퍼 복사의 오버헤드를 제거하기 위해 문자열을 *PChar*로 변환할 수 있습니다(루틴에서 *PChar*를 메모리에 남겨둘 필요가 없다고 확신할 경우). 그러나 **array of char 타입**을 **string**에 할당할 때처럼 문자열 길이의 동기화가 자동적으로 발생하지는 않습니다. 실제 문자열의 길이를 반영하도록 문자열 *Length*를 다시 설정해야 합니다. 복사된 바이트 수를 반환하는 함수를 사용하는 경우에는 다음 한 줄의 코드로 이 작업을 안전하게 수행할 수 있습니다.

```
var
    S:string;
begin
    SetLength(S, MAX_SIZE); // when casting to a PChar, be sure the string is not empty
    SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ));
    // statements
end;
```

## 문자열에 대한 컴파일러 지시어

다음 컴파일러 지시어는 문자와 문자열 타입에 영향을 미칩니다.

표 4.8 문자열에 대한 컴파일러 지시어

| 지시어               | 설명                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| { <i>\$H</i> +/-} | 컴파일러 지시어 <i>\$H</i> 는 예약어 <b>string</b> 이 짧은 문자열 또는 긴 문자열을 나타내는 지 여부를 제어합니다. 기본 상태 { <i>\$H</i> +}에서, <b>string</b> 은 긴 문자열을 나타냅니다. { <i>\$H</i> -} 지시어를 사용하면 긴 문자열을 <i>ShortString</i> 으로 바꿀 수 있습니다.                                                                                                                                                                                                                                             |
| { <i>\$P</i> +/-} | <i>\$P</i> 지시어는 { <i>\$H</i> -} 상태에서 컴파일된 코드에서만 의미가 있으며 역 호환성을 위해 제공됩니다. <i>\$P</i> 는 { <i>\$H</i> -} 상태에서 <b>string</b> 키워드를 사용하여 선언된 변수 매개변수의 의미를 제어합니다.<br><br>string 키워드를 사용하여 선언된 변수 매개변수는 { <i>\$P</i> -} 상태에서는 일반적인 변수 매개변수이지만, { <i>\$P</i> +} 상태에서는 개방형(open) 문자열 매개변수입니다. <i>\$P</i> 지시어의 설정에 상관 없이, <b>OpenString</b> 식별자는 항상 개방형 문자열 매개변수를 선언하는 데 사용될 수 있습니다.                                                                         |
| { <i>\$V</i> +/-} | <i>\$V</i> 지시어는 변수 매개변수로 전달된 짧은 문자열에 대한 타입 확인을 제어합니다. { <i>\$V</i> +} 상태에서는 형식 매개변수와 실제 매개변수의 문자열 타입이 동일해야 하는 엄격한 타입 확인이 수행됩니다.<br><br>{ <i>\$V</i> -} (완화된) 상태에서 모든 짧은 문자열 타입 변수는 선언된 최대 길이가 형식 매개변수의 길이와 동일하지 않더라도 실제 매개변수로 허용됩니다. 이렇게 하면 메모리 훼손을 유발할 수 있다는 점에 유의하십시오. 예를 들면, 다음과 같습니다.<br><br><pre>var S: string[3];<br/><br/>procedure Test(var T:string);<br/>begin<br/>  T := '1234';<br/>end;<br/><br/>begin<br/>  Test(S);<br/>end.</pre> |
| { <i>\$X</i> +/-} | Kylix에서는 { <i>\$X</i> +} 컴파일러 지시어를 사용하여 기본 제공 <i>PChar</i> 타입과 인덱스가 0부터 시작하는 문자 배열에 적용되는 특별한 규칙을 활성화하여 Null 종료 문자열을 지원합니다. 이러한 규칙을 통해 인덱스가 0부터 시작하는 배열과 문자 포인터를 시스템 유닛의 <i>Write</i> , <i>Writeln</i> , <i>Val</i> , <i>Assign</i> , <i>Rename</i> 등과 함께 사용할 수 있습니다.                                                                                                                                                                              |

## 문자열과 문자: 관련 주제

오브젝트 파스칼 랭귀지 안내서의 다음 항목에서 문자열과 문자 집합에 대해 설명합니다. 12장 "국제적인 애플리케이션 만들기"도 참조하십시오.

- "확장 문자 집합 정보"(국제 문자 집합에 대해 설명)
- "Null 종료 문자열 사용"(문자 배열에 대한 내용 포함)
- "문자열"
- "문자 포인터"
- "문자열 연산자"

## 파일 작업

---

이 단원에서는 파일 작업에 대해 설명하고 디스크에서 파일을 처리하는 것과 파일 읽기 및 쓰기 같은 I/O 작업을 구별합니다. 첫 번째 단원에서는 디스크에서 파일 처리를 포함하는 일반적인 프로그래밍 작업에서 사용하는 루틴과 런타임 라이브러리에 대해 설명합니다. 다음 단원은 파일 I/O와 함께 사용되는 파일 형식에 대한 개요입니다. 마지막 단원에서는 파일 I/O에 권장되는 방법인 파일 스트림 사용 방법을 중점적으로 다룹니다.

오브젝트 파스칼 랭귀지는 대소문자를 구별하지 않지만 Linux 운영 체제는 구별합니다. Linux에서 파일을 처리할 때 대소문자에 주의하십시오.

**참고** 오브젝트 파스칼 랭귀지의 이전 버전에서는 근래에 보편적으로 사용되는 파일 이름 매개변수가 아닌 파일 자체에 대한 작업을 수행하였습니다. 이러한 파일 형식을 사용할 경우, 이를 떼면 파일 이름을 재지정하기 전에 파일을 찾아 파일 변수에 할당해야 했습니다.

## 파일 처리

---

몇 가지 일반적인 파일 작업들은 오브젝트 파스칼의 런타임 라이브러리에 기본 제공되어 있습니다. 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. 파일 핸들은 어떤 것을 식별하기 위해 Kyxix에서 할당하는 번호입니다.

오브젝트 파스칼은 대부분의 파일 처리용 루틴을 제공합니다. 만일 제공되지 않는 경우에는 대체 루틴에 대해 설명합니다.

**주의** 오브젝트 파스칼 랭귀지는 대소문자를 구별하지 않지만 Linux 운영 체제는 구별합니다. Linux에서 파일을 처리할 때 대소문자에 주의하십시오.

## 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

파일을 찾는 데 사용되는 세 가지 루틴은, *FindFirst*, *FindNext*, *FindClose*입니다. *FindFirst*는 지정된 디렉토리에서 주어진 속성을 이용해서 파일 이름의 첫 번째 인스턴스를 찾습니다. *FindNext*는 이전 *FindFirst* 호출에서 지정된 이름과 속성에 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 경우, *FileExists* 함수를 사용하면 파일이 존재하는 경우에 *True*를 반환하고 존재하지 않는 경우에는 *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.

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer; //Time contains the time stamp of the file.
    Size: Integer; //Size contains the size of the file in bytes.
    Attr: Integer; //Attr represents the file attributes of the file.
    Name: TFileName; //Name contains the file name and extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData; //FindData contains additional information such as
    //file creation time, last access time, long and short filenames.
  end;
```

파일을 찾으면, *TSearchRec* 타입의 매개변수 필드는 찾아낸 파일을 설명하도록 수정됩니다. 파일에 특정한 속성이 있는지 확인하기 위해 다음 속성 상수 또는 값에 대한 *Attr*을 검사할 수 있습니다.

**표 4.9** 속성 상수 및 값

| 상수                 | 값          | 설명       |
|--------------------|------------|----------|
| <i>faReadOnly</i>  | \$00000001 | 읽기 전용 파일 |
| <i>faHidden</i>    | \$00000002 | 숨김 파일    |
| <i>faSysFile</i>   | \$00000004 | 시스템 파일   |
| <i>faVolumeID</i>  | \$00000008 | 볼륨 ID 파일 |
| <i>faDirectory</i> | \$00000010 | 디렉토리 파일  |
| <i>faArchive</i>   | \$00000020 | 아카이브 파일  |
| <i>faAnyFile</i>   | \$0000003F | 모든 파일    |

속성을 검사하려면 **and** 연산자를 사용하여 *Attr* 필드의 값을 속성 상수와 조합합니다. 파일이 그 속성을 가지면 결과는 0보다 큼니다. 예를 들어, 찾아낸 파일이 숨김 파일이면, 다음 표현식은 *True*로 평가됩니다 (*SearchRec.Attr and faHidden > 0*) 속성들은 그 상수 또는 값을 OR로 추가해서 조합될 수 있습니다. 예를 들면, 일반적인 파일 이외에 읽기 전용 파일과 숨김 파일을 찾으려면, (*faReadOnly or faHidden*) *Attr* 매개변수를 전달합니다.

**예** 이 예는 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.

```
var
    SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender:TObject);
begin
    FindFirst('/usr/Kylix/bin/*.*', faAnyFile, SearchRec);
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
end;

procedure TForm1.Edit1Click(Sender:TObject);
begin
    if (FindNext(SearchRec) = 0)
        Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
    else
        FindClose(SearchRec);
end;
```

## 파일 속성 변경

모든 파일에는 운영 체제에 의해 비트맵 플래그로 저장된 여러 가지 속성들이 있습니다. 파일 속성은 파일이 읽기 전용인지 숨김 파일인지 표시하기 위한 항목들을 포함합니다. 파일 속성을 변경하려면 읽기, 변경, 설정의 세 단계가 필요합니다.

파일 액세스 권한에 따라 수정하는 데 제한이 있을 수 있습니다.

**파일 속성 읽기:** 운영 체제는 다양한 방법으로 파일 속성을 저장하는데, 보통 비트맵 플래그로 저장합니다. 파일 속성을 읽으려면, 파일 이름을 *FileGetAttr* 함수로 전달하고, 여기서 파일 속성을 반환합니다. 반환값은 *Word* 타입의 비트맵 파일 속성의 그룹입니다. 속성은 *TSearchRec*에서 정의된 상수를 가진 속성들을 AND로 연결하여 조사할 수 있습니다. 반환값 -1은 오류가 발생했다는 것을 가리킵니다.

**개별적인 파일 속성 변경:** Kylix에서는 파일 속성을 비트 플래그로 나타내므로, 일반적인 논리 연산자를 사용하여 개별적인 속성을 처리할 수 있습니다. 각 속성에는 SysUtils 유닛에 정의되어 있는 니모닉 이름(mnemonic name)이 있습니다. 예를 들어, 파일의 읽기 전용 속성을 설정하려면 다음과 같이 합니다.

```
Attributes := Attributes or faReadOnly;
```

또한 여러 속성들을 한꺼번에 설정하거나 지울 수 있습니다. 예를 들어, 다음은 시스템 파일과 숨김 파일 속성 모두 지웁니다.

```
Attributes := Attributes and not (faSysFile or faHidden);
```

**파일 속성 설정:** Kylix에서는 언제든지 모든 파일의 속성을 설정할 수 있습니다. 파일의 속성을 설정하려면 원하는 파일 이름과 속성을 *FileSetAttr* 함수에 전달합니다. *FileSetAttr*은 지정된 파일의 파일 속성을 설정합니다.

파일 속성만 정하거나 이전 설정과 상관 없이 속성을 설정하는 경우 읽기 작업과 설정 작업을 따로 사용할 수 있습니다. 그러나 이전 설정에 기반하여 속성을 변경하려면 기존 속성을 읽고, 수정하고, 수정된 속성을 기록해야 합니다.

## 파일 이름 재지정

파일 이름을 변경하려면, 단순히 다음 *RenameFile* 함수를 사용합니다.

```
function RenameFile(const OldFileName, NewFileName:string):Boolean;
```

여기서, *OldFileName*으로 식별한 파일 이름을 *NewFileName*에서 지정한 파일 이름으로 변경합니다. 이 작업이 성공하면, *RenameFile*은 *True*를 반환합니다. 만일 파일 이름을 재지정하지 못 하면, 예를 들어 *NewFileName*이라는 파일이 이미 있을 경우, *False*를 반환합니다. 예를 들면, 다음과 같습니다.

```
if not RenameFile('OLDNAME.TXT', 'NEWNAME.TXT') then
  ErrorMessage('Error renaming file!');
```

여기서 *RenameFile*을 사용하여 다른 드라이브로 파일을 이름 재지정(이동)할 수 없습니다. 먼저 파일을 복사한 다음 이전 파일을 삭제해야 합니다.

## 파일 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*와 *FileSetDate*는 *Handle* 타입을 매개변수로 사용합니다. *Handle*에 대한 액세스를 얻으려면, *TFileStream*, *FileOpen*을 인스턴스화하거나 *\_open()*과 같은 OS 루틴을 호출하여 파일을 만들거나 엽니다. 그런 다음 *Handle* 속성을 사용합니다. 자세한 내용은 4-40 페이지의 "파일 I/O와 파일 형식"을 참조하십시오.

## 파일 I/O와 파일 형식

파일 I/O 작업을 할 때, 파스칼 파일 형식, 파일 핸들, 파일 스트림 객체의 세 가지 형식을 사용할 수 있습니다. 다음 표는 이러한 형식들을 요약한 것입니다.

**표 4.10** 파일 I/O를 위한 파일 형식

| 파일 형식     | 설명                                                                                                                                                                                                                                                                |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 파스칼 파일 형식 | 시스템 유닛에 있습니다. 이러한 형식은 대개 "F:Text 또는 "F: File" 형식의 파일 변수에 사용됩니다. 이러한 파일에는 타입이 지정된 (typed), 텍스트, 타입이 지정되지 않은 (untyped)이란 세 가지 형식이 있습니다. <i>AssignPrn</i> 과 <i>writeln</i> 과 같은 많은 Kylix 파일 처리 루틴에서는 이러한 형식들을 사용합니다. 이 형식들을 사용하려면, <i>오브젝트 파스칼 런타임 안내서</i> 를 참조하십시오. |
| 파일 핸들     | 시스템 유닛에 있습니다. 많은 루틴이 핸들을 사용하여 파일을 식별합니다. 파일을 열거나 만들 때 핸들이 생깁니다( <i>FileOpen</i> 또는 <i>FileCreate</i> 사용). 일단 핸들이 있으면, 그 핸들을 사용하여 파일의 내용을 작업할 수 있는 루틴이 생깁니다 (한 줄 작성, 텍스트 읽기 등).                                                                                    |
| 파일 스트림    | 파일 스트림은 디스크 파일의 정보를 액세스하는 데 사용되는 <i>TFileStream</i> 클래스의 객체 인스턴스입니다. 파일 스트림은 파일 I/O를 위한 포팅 가능한 수준 높은 방법입니다. <i>TFileStream</i> 에는 파일 핸들에 액세스할 수 있는 <i>Handle</i> 속성이 있습니다. 다음 섹션에서는 <i>TFileStream</i> 에 대해 설명합니다.                                                |



## 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 영구성, 다른 스트림과의 상호 작용, 파일 I/O와 같은 많은 기능을 제공합니다.

- *TFileStream*은 스트림 클래스의 자손입니다. 이와 같은 파일 스트림 사용상의 한 가지 이점은 파일 스트림이 컴포넌트 속성을 영구적으로 저장하는 능력을 상속한다는 것입니다. 스트림 클래스는 *TFileer* 클래스, *TReader*, *TWriter*를 사용하여 객체를 디스크로 스트림합니다. 따라서, 파일 스트림을 가지고 있을 때, CLX 스트리밍 메커니즘용으로 동일한 코드를 사용할 수 있습니다. 스트리밍 시스템 사용에 대한 자세한 내용은 *TStream*, *TFileer*, *TReader*, *TWriter*, *TComponent* 클래스 등에 대한 *CLX Reference* 온라인 도움말을 참조하십시오.
- *TFileStream*은 다른 스트림 클래스와 쉽게 상호 작용할 수 있습니다. 예를 들어, *TFileStream*과 *TMemoryStream*을 사용하면 동적 메모리 블록을 디스크에 덤핑할 수 있습니다.
- *TFileStream*은 파일 I/O용 기본 메소드와 속성을 제공합니다. 나머지 단원에서 파일 스트림의 이러한 면을 중점적으로 다룹니다

## 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*은 예외를 발생시킵니다.

```
constructor Create(const filename: string; Mode:Word);
```

*Mode* 매개변수는 파일 스트림을 생성할 때 파일을 여는 방법을 지정합니다. *Mode* 매개변수는 개방 모드와 공유 모드 두 가지로 구성됩니다. 개방 모드는 다음 값 중의 하나여야 합니다.

**표 4.11** 개방 모드

| 값               | 의미                                                                     |
|-----------------|------------------------------------------------------------------------|
| fmCreate        | TFileStream은 파일을 주어진 이름으로 생성합니다. 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 대체하기 보다는 수정하기 위해 파일을 엽니다.                                       |

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

표 4.12 공유 모드

| 값                | 의미                                              |
|------------------|-------------------------------------------------|
| fmShareCompat    | 다른 애플리케이션에서 파일을 절대 열 수 없습니다.                    |
| 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 읽거나 쓰기가 금지되어 있습니다.               |

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

표 4.13 각 개방 모드에서 사용할 수 있는 공유 모드

| 개방 모드           | fmShareCompat | fmShareExclusive | fmShareDenyWrite | fmShareDenyRead | fmShareDenyNone |
|-----------------|---------------|------------------|------------------|-----------------|-----------------|
| fmOpenRead      | 사용 불가         | 사용 불가            | 사용 가능            | 사용 불가           | 사용 가능           |
| fmOpenWrite     | 사용 가능         | 사용 가능            | 사용 불가            | 사용 가능           | 사용 가능           |
| fmOpenReadWrite | 사용 가능         | 사용 가능            | 사용 가능            | 사용 가능           | 사용 가능           |

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

## 파일 핸들 사용

*TFileStream*을 인스턴스화하면 파일 핸들에 대한 액세스를 얻게 됩니다. 파일 핸들은 *Handle* 속성에 포함되어 있습니다. *Handle* 속성은 읽기 전용이며 파일이 열린 모드를 나타냅니다. 파일 *Handle*의 속성을 변경하려면, 새로운 파일 스트림 객체를 만들어야 합니다.

일부 파일 처리 루틴은 window의 파일 핸들을 매개변수로 사용합니다. 일단 파일 스트림을 가지면, window의 파일 핸들을 사용하는 모든 상황에서 *Handle* 속성을 사용할 수 있습니다. 파일 스트림은 핸들 스트림과 달리 객체가 소멸될 때 파일 핸들을 닫는다는 점에 유의하십시오.

## 파일 읽기 및 쓰기

*TFileStream*에는 파일을 읽고 쓰는 몇 가지 다른 방법이 있습니다. 이러한 방법들은 다음을 수행하는지에 따라 구별됩니다.

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

*Read*는 파일 스트림과 연결된 파일을 현재 *Position*에서 시작하여 *Count* 바이트까지 *Buffer*에 읽는 함수입니다. 그리고 나서 *Read*는 파일의 현재 위치를 실제로 전송된 바이트 수만큼 앞으로 나가게 합니다. 다음은 *Read*에 대한 프로토타입입니다.

```
function Read(var Buffer; Count:Longint):Longint; override;
```

*Read*는 파일의 바이트 수를 모를 때 유용합니다. *Read*는 실제로 옮겨진 바이트 수를 반환하는데, 파일 표시의 끝에 도달하면 *Count*보다 작아집니다.

*Write*는 *Position*에서 시작하여 파일 스트림과 연결된 파일에 *Buffer*로부터 나온 *Count* 바이트를 쓰는 함수입니다. 다음은 *Write*에 대한 프로토타입입니다.

```
function Write(const Buffer; Count:Longint):Longint; override;
```

파일에 기록한 후, *Write*는 현재 위치를 기록된 바이트 수만큼 앞으로 나가게 하고, 실제로 기록된 바이트 수를 반환하는데, 버퍼의 끝에 도달하면 *Count*보다 작아집니다.

그 반대 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*와 *WriteBuffer*는 오류 시 (*EReadError*와 *EWriteError*) 예외를 발생시키지만 *Read* 메소드와 *Write* 메소드는 발생시키지 않습니다. 다음은 *ReadBuffer*와 *WriteBuffer*에 대한 프로토타입입니다.

```
procedure ReadBuffer(var Buffer; Count:Longint);
```

```
procedure WriteBuffer(const Buffer; Count:Longint);
```

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

## 문자열 읽기 및 쓰기

문자열을 read 함수 또는 write 함수에 전달하는 경우 정확한 구문을 사용해야 합니다. read 루틴과 write 루틴에 대한 *Buffer* 매개변수는 각각 **var** 타입과 **const** 타입입니다. 이 매개변수들은 타입이 지정되지 않았으므로 루틴은 변수의 주소를 사용합니다.

문자열을 사용할 때 가장 많이 사용하는 타입은 긴 문자열입니다. 그러나 긴 문자열을 *Buffer* 매개변수로 전달하면 올바른 결과를 얻을 수 없습니다. 긴 문자열에는 문자열에 있는 문자의 크기, 참조 카운트, 포인터가 들어 있습니다. 따라서, 긴 문자열을 역참조 (dereference) 하여 포인터 요소에만 그 결과를 나타내는 것은 아닙니다. 사용자가 먼저 해야 할 일은 문자열을 *Pointer* 또는 *PChar* 로 변환한 다음 문자열을 역참조하는 것입니다. 예를 들면,

```
procedure caststring;
var
  fs:TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s)); // this will give you garbage
  fs.Write(PChar(s)^, Length(s)); // this is the correct way
end;
```

## 파일 찾기

가장 전형적인 파일 I/O 메커니즘에는 파일 안의 특정 위치에서 읽고 쓰기 위해 파일을 찾는 프로세스가 있습니다. 이러한 목적으로 *TFileStream*은 *Seek* 메소드를 갖고 있습니다. 다음은 *Seek*에 대한 프로토타입입니다.

```
function Seek(Offset: Longint; Origin: Word):Longint; override;
```

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

| 값               | 의미                                                                                                           |
|-----------------|--------------------------------------------------------------------------------------------------------------|
| soFromBeginning | 리소스의 시작 지점에서 <i>Offset</i> 이 시작합니다. <i>Seek</i> 는 <i>Offset</i> 위치로 이동합니다. <i>Offset</i> 은 $\geq 0$ 이어야 합니다. |
| soFromCurrent   | 리소스의 현재 위치에서 <i>Offset</i> 이 시작합니다. <i>Seek</i> 는 $Position + Offset$ 으로 이동합니다.                              |
| soFromEnd       | 리소스의 끝에서 <i>Offset</i> 이 시작합니다. 파일 끝으로부터의 바이트 수를 나타내려면 <i>Offset</i> 은 $\leq 0$ 이어야 합니다.                     |

*Seek*는 스트림의 현재 *Position*을 재설정하여 표시된 오프셋만큼 이동합니다. *Seek*는 *Position* 속성의 새로운 값, 즉 리소스에서 새로운 현재 위치를 반환합니다.

## 파일 위치 및 크기

*TFileStream*은 파일의 현재 위치와 크기를 포함하는 속성을 가집니다. 이러한 속성들은 *Seek*, *read*, *write* 등의 메소드에 의해 사용됩니다.

*TFileStream*의 *Position* 속성은 현재 오프셋을 스트림에(스트림 데이터의 시작부터) 바이트 단위로 나타내는데 사용합니다. 다음은 *Position*에 대한 선언입니다.

```
property Position:Longint;
```

*Size* 속성은 스트림의 크기를 바이트 단위로 표시합니다. 이 속성은 파일을 잘라내기 위한 파일 끝 표시로 이용됩니다. 다음은 *Size*에 대한 선언입니다.

```
property Size:Longint;
```

*Size*는 스트림에 읽고 쓰는 루틴에서 내부적으로 사용합니다.

*Size* 속성을 설정하면 파일 크기가 변경됩니다. 파일의 *Size*를 변경되지 않으면 예외가 발생합니다. 예를 들어, *fmOpenRead* 모드에서 열린 파일의 *Size*를 변경하려고 하면 예외가 발생합니다.

## 복사

*CopyFrom*은 지정된 바이트 수만큼 한 파일 스트림에서 다른 스트림으로 복사합니다.

```
function CopyFrom(Source: TStream; Count:Longint):Longint;
```

*CopyFrom*을 사용하면 데이터를 복사할 때 버퍼를 만들고, 버퍼로 읽어 들이고, 버퍼에서 옮겨 작성하고, 버퍼를 해제할 필요가 없습니다.

*CopyFrom*은 *Source*에서 스트림으로 *Count* 바이트 만큼 복사합니다. 그 다음 *CopyFrom*은 현재 위치를 *Count*만큼 이동하고 복사된 바이트 수를 반환합니다. *Count*가 0인 경우, *CopyFrom*은 읽기 전에 *Source* 위치를 0으로 설정한 다음 *Source*의 전체 내용을 스트림에 복사합니다. *Count*가 0보다 더 크거나 더 작은 경우, *CopyFrom*은 *Source*에 있는 현재 위치에서 읽습니다.

## 오브젝트 파스칼 데이터 타입

오브젝트 파스칼에는 이미 정의된 데이터 타입이 많이 들어 있습니다. 이러한 이미 정의된 타입을 사용하여 애플리케이션에서 필요로 하는 내용에 따라 새로운 타입을 만들 수 있습니다. 타입에 대한 개요는 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.



## 애플리케이션과 공유 객체 구축

이 장에서는 Kylix를 사용하여 애플리케이션과 공유 객체를 구축하는 방법에 대한 개요를 제공합니다.

### 애플리케이션 생성

---

Kylix의 주된 용도는 다음 두 가지 애플리케이션을 디자인하고 구축하는 것입니다.

- GUI 애플리케이션
- 콘솔 애플리케이션

### GUI 애플리케이션

---

프로젝트를 컴파일하면 실행 파일이 생성됩니다. 실행 파일은 보통 사용자 프로그램의 기본적인 기능을 제공하고, 간단한 프로그램은 대체로 실행 파일 하나로만 구성됩니다. 실행 파일로부터 공유 객체 파일, 패키지 및 기타 지원 파일을 호출하여 애플리케이션을 확장할 수 있습니다.

Kylix는 다음 두 가지 애플리케이션 UI 모델을 제공합니다.

- 단일 문서 인터페이스(SDI)
- 다중 문서 인터페이스(MDI)

애플리케이션의 구현 모델 외에, 프로젝트의 디자인 타임 행동과 애플리케이션의 런타임 행동은 IDE에서 프로젝트 옵션을 설정하여 처리할 수 있습니다.

### 사용자 인터페이스 모델

모든 폼은 단일 문서 인터페이스(SDI) 또는 다중 문서 인터페이스(MDI) 폼으로 구현될 수 있습니다. MDI 애플리케이션에서는 단일 부모 창에서 둘 이상의 문서나 자식 창을 열 수 있습니다. MDI는 스프레드시트나 워드 프로세서 같은 애플리케이션에서는 일반적인 인터페이스입니다. 반면에, SDI 애플리케이션은 단일 문서 뷰를 가집니다. 폼을

SDI 애플리케이션으로 만들려면 *Form* 객체의 *FormStyle* 속성을 *fsNormal*로 설정합니다.

애플리케이션용 UI 개발에 대한 자세한 내용은 6장 "애플리케이션 사용자 인터페이스 개발"을 참조하십시오.

## SDI 애플리케이션

새 SDI 애플리케이션을 만들려면, 다음과 같이 합니다.

- 1 File|New를 선택하여 New Items 대화 상자를 실행합니다.
- 2 Projects 페이지를 눌러 SDI Application 을 선택합니다.
- 3 OK 를 클릭합니다.

기본적으로 *Form* 객체의 *FormStyle* 속성은 *fsNormal*로 설정되므로 Kylix에서는 모든 새 애플리케이션이 SDI 애플리케이션이라고 가정합니다.

## MDI 애플리케이션

새 MDI 애플리케이션을 생성하려면, 다음과 같이 합니다.

- 1 File|New 를 선택하여 New Items 대화 상자를 실행합니다.
- 2 Projects 페이지를 누르고 MDI Application 을 선택합니다.
- 3 OK 를 클릭합니다.

MDI 애플리케이션은 보다 계획적이어야 하고, SDI 애플리케이션보다 디자인하기가 다소 복잡합니다. MDI 애플리케이션은 클라이언트 창 내에 상주하는 자식 창을 생성하고, 메인 폼은 자식 폼을 포함합니다. *TForm* 객체의 *FormStyle* 속성을 설정하여 폼이 자식 (*fsMDIForm*)인지 또는 메인 폼 (*fsMDIChild*)인지를 지정합니다. 자식 폼의 속성 재설정을 피하려면 자식 폼의 기본 클래스를 정의하고 이 클래스에서 각 자식 폼을 파생시키는 것이 좋습니다.

## IDE, 프로젝트 및 컴파일 옵션 설정

Project|Project Options를 사용하여 프로젝트에 대한 여러가지 옵션을 지정합니다. 자세한 내용은 온라인 도움말을 참조하십시오.

차후의 모든 프로젝트에 적용될 기본 옵션을 변경하려면, Project Options 대화 상자에서 옵션을 설정하고 창의 오른쪽 하단에 있는 Default 상자를 선택 표시합니다. 모든 새 프로젝트는 현재 옵션을 사용합니다.

## 콘솔 애플리케이션

콘솔 애플리케이션은 창 관리 레이어를 요구하지 않고 콘솔 창에서 직접 작동할 수 있습니다. 콘솔 애플리케이션은 그래픽 인터페이스 없이 실행되는 32비트 프로그램입니다. 콘솔 애플리케이션은 일반적으로 사용자 입력이 많지 않으며, 제한된 함수 집합을 수행합니다.

새 콘솔 애플리케이션을 생성하려면 다음과 같이 합니다.

- 1 File|New를 선택한 다음, New Items 대화 상자에서 Console Application을 선택합니다.



Kylix는 콘솔형 소스 파일에 대한 프로젝트 파일을 생성하고 코드 에디터를 표시합니다.

**참고** 새 콘솔 애플리케이션을 생성하면, IDE는 새 폼을 만들지 않고, 코드 에디터만 표시됩니다.

## 패키지와 공유 객체 파일 생성

공유 객체 파일은 실행 파일과 함께 작동되며, 애플리케이션에 기능을 제공하는 컴파일된 코드 모듈입니다.

패키지는 Kylix 애플리케이션, IDE 또는 둘 모두에서 사용하는 특별한 공유 객체 파일입니다. 패키지의 종류에는 런타임 패키지과 디자인 타임 패키지가 있습니다. 런타임 패키지는 프로그램 실행 중에 프로그램에 기능을 제공합니다. 디자인 타임 패키지는 IDE의 기능을 확장합니다.

자세한 내용은 11장 "패키지와 컴포넌트 사용"을 참조하십시오.

## 공유 객체 라이브러리 사용

Linux의 공유 객체 라이브러리는 Windows DLL과 유사합니다. Windows에서 DLL 함수를 사용하는 것처럼, 외부 함수 선언을 사용하여 타사(third-party) 공유 객체와 연결할 수 있습니다.

Linux 프로그램 로더는 외부 함수 참조를 해석할 때 모듈 이름 바인딩을 무시합니다. 애플리케이션이 Foo라는 함수를 export하는 .so 라이브러리를 두 개 사용하면, 로더는 로더가 찾는 첫 번째 Foo 함수에 모든 Foo 참조를 바인딩합니다. (검색 순서는 ELF 표준에 있는 "Shared Object Dependencies"에 대한 단원에서 설명합니다.) 이름 충돌을 피할 수 없을 때, dlopen()을 사용하여 객체를 동적으로 로드함으로써 의도하지 않은 행동을 방지할 수 있습니다.

라이브러리 프로젝트를 구축할 때, 컴파일러는 일반적인 실행 파일 대신에 공유 객체(.so 파일)를 생성합니다. 생성된 파일 이름은 기본적으로 "lib"(표준 라이브러리인 경우) 또는 "bpl"(패키지인 경우)로 시작합니다. 예를 들어, 프로젝트 파일이 something.pas이면, 컴파일러는 libsomething.so 또는 bplsomething.so라는 공유 객체를 생성합니다.

다음 컴파일러 지시어를 라이브러리 프로젝트 파일에 사용할 수 있습니다.

**표 5.1** 라이브러리용 컴파일러 지시어 ( 계속 )

| 컴파일러 지시어                             | 설명                                                                                                                                                              |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>{\${SOPREFIX} 'string'}</code> | 출력 파일 이름에서 기본 접두어 'lib' 또는 'bpl'을 오버라이드합니다. 예를 들어, 디자인 타임 패키지에 <code>{\${SOPREFIX} 'dcl'}</code> 을 지정하거나, <code>{\${SOPREFIX} '}'</code> 를 사용하여 접두어를 없앨 수 있습니다. |
| <code>{\${SOSUFFIX} 'string'}</code> | 출력 파일 이름에서 .so 확장자 앞에 지정된 접미어를 추가합니다. 예를 들어, something.pas에서 <code>{\${SOSUFFIX} '-2.1.3'}</code> 을 사용하여 libsomething-2.1.3.so를 만듭니다.                           |

**표 5.1** 라이브러리용 컴파일러 지시어 ( 계속 )

| 컴파일러 지시어                              | 설명                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>{\${SOVERSION 'string'}}</code> | 출력 파일 이름에서 .so 확장자 뒤에 두 번째 확장자를 추가합니다. 예를 들어, something.pas에서 <code>{\${SOVERSION '2.1.3'}}</code> 를 사용하여 libsomething.so.2.1.3을 만듭니다.                                                                                                                                                                                                          |
| <code>{\${SONAME 'string'}}</code>    | 공유 객체의 동적 문자열 테이블에서 내부적으로 코딩된 라이브러리 이름을 지정합니다. 이 지시어는 출력 파일 자체의 이름을 바꾸지 않지만, 실제 파일을 가리키는 출력 디렉토리에서 컴파일러가 심볼릭 링크 (symbolic link)를 만들도록 합니다. 예를 들어, something.pas에 <code>{\${SONAME 'libsomething.so.1'}}</code> 를 사용하면, 컴파일러는 libsomething.so라는 공유 객체 파일 (내부적으로 libsomething.so.1로 표시)과 libsomething.so를 가리키는 libsomething.so.1이라는 심볼릭 링크를 만듭니다. |

## 패키지와 공유 객체를 사용하는 경우

Kylix로 개발된 대부분의 애플리케이션에서, 패키지는 공유 객체보다 더 많은 유연성을 제공하고 더 쉽게 생성될 수 있습니다. 그러나, 프로젝트에 패키지보다 공유 객체가 더 적합한 경우도 있습니다.

- 코드 모듈이 Kylix 이외의 애플리케이션에서 호출되는 경우
- 웹 서버의 기능을 확장하는 경우
- 타사 개발자를 위해 코드 모듈을 생성하는 경우

## 데이터베이스 애플리케이션 개발

Kylix의 장점 중 하나는 고급 데이터베이스 애플리케이션에 대한 지원입니다. Kylix에는 애플리케이션 간에 투명한 데이터 공유를 제공하면서 InterBase, MySQL 또는 기타 서버에 연결할 수 있도록 하는 기본 제공 툴이 있습니다.

데이터베이스 애플리케이션은 사용자 인터페이스 요소, 데이터베이스 정보 (데이터셋)를 나타내는 컴포넌트 및 데이터베이스 정보 자체에 연결하는 컴포넌트로 구축됩니다. Kylix는 다음 두 종류의 데이터셋을 지원합니다.

- dbExpress
- 클라이언트

다양한 종류의 데이터셋은 원본으로 사용하는 데이터베이스 정보에 다양한 방식으로 연결됩니다. *dbExpress*는 데이터베이스 정보에 대한 빠른 액세스를 제공하고, 크로스 플랫폼 개발을 지원하지만, 많은 데이터 처리 함수들을 포함하지 않습니다. 클라이언트 데이터셋에는 기본 제공 데이터베이스 액세스 메커니즘이 없기 때문에, 클라이언트 데이터셋은 데이터를 메모리에 버퍼하지만 클라이언트 데이터셋을 데이터베이스 서버에 직접 연결할 수 없습니다. 대신, 데이터 액세스를 처리할 수 있는 다른 데이터셋에 클라이언트 데이터셋을 연결해야 합니다. 자세한 내용은 14-4 페이지의 "데이터베이스 아키텍처"를 참조하십시오.

이 설명서의 2부 "데이터베이스 애플리케이션 개발"에서는 Kylix를 사용하여 데이터베이스 애플리케이션을 디자인하는 방법에 대한 자세한 내용을 제공합니다.

## 분산 애플리케이션 구축

분산 애플리케이션은 다양한 시스템과 플랫폼에 배포된 애플리케이션이고, 보통 네트워크 상에서 함께 작동하여 관련 함수 집합을 수행합니다. 예를 들면, 전국에 분포되어 있는 매장에 대한 상품 구매를 관리하고 추적하는 애플리케이션에는 모든 매장에 대한 개별 클라이언트 애플리케이션과 클라이언트의 요청을 처리하는 메인 서버 및 트랜잭션에서 발생한 모든 정보를 저장하는 데이터베이스 인터페이스 등이 필요합니다. 분산 클라이언트 애플리케이션 (예: 웹 기반 애플리케이션) 을 구축하면 개별 클라이언트를 유지 관리하고 업데이트하는 작업이 대폭 단순화됩니다.

Kylix는 다음과 같은 분산 애플리케이션 구현을 위한 옵션을 제공합니다.

- TCP/IP 애플리케이션

### TCP/IP를 사용한 분산 애플리케이션

TCP/IP는 네트워크 상에서 통신하는 애플리케이션을 개발할 수 있는 통신 프로토콜입니다. 가상으로 애플리케이션의 모든 디자인을 구현할 수 있습니다. TCP/IP는 전송 계층 (transport layer) 을 제공하지만, 분산 애플리케이션 생성을 위한 특별한 아키텍처가 필요하지는 않습니다.

인터넷의 성장으로 인해 대부분의 컴퓨터가 이미 어떤 형태의 TCP/IP 액세스가 있는 환경을 갖추고 있어, 애플리케이션의 배포와 설치가 단순화되었습니다.

TCP/IP를 사용하는 애플리케이션은 HTTP 요청 메시지를 서비스하는 웹 서버 애플리케이션 같은 메시지 기반 분산 애플리케이션 또는 소켓을 사용하여 통신하는 분산 데이터베이스 애플리케이션 같은 분산 객체 애플리케이션입니다.

TCP/IP 기능을 애플리케이션에 추가하는 가장 기본적인 방법은 클라이언트 또는 서버 소켓을 사용하는 것입니다. Kylix는 또한 CGI 스크립트를 작성하여 웹 서버를 확장하는 애플리케이션도 지원합니다.

### 애플리케이션에서 소켓 사용

*TTCPClient* 클래스와 *TTCPServer* 클래스를 사용하여 TCP/IP 소켓 연결을 만들어 다른 원격 애플리케이션과 통신할 수 있습니다. 소켓에 대한 자세한 내용은 23장 "소켓 작업"을 참조하십시오.

## 웹 서버 애플리케이션 만들기

새 웹 서버 애플리케이션을 만들려면 File|New를 선택한 다음 New Items 대화 상자에서 Web Server Application을 선택합니다. 그런 다음 웹 서버 애플리케이션 종류를 다음 중에서 선택합니다.

- Apache
- CGI 독립형

Common Gateway Interface (CGI)는 웹 페이지로부터 서버의 프로그램을 실행하기 위한 표준입니다. CGI 프로그램 또는 스크립트는 스스로 실행되는 실행 프로그램입니다. CGI 애플리케이션은 Apache 애플리케이션보다 서버의 시스템 리소스를 많이 사용하므로 복잡한 애플리케이션은 Apache 애플리케이션으로 개발하는 것이 좋습니다. Apache 웹 서버는 최신 프로토콜을 구현하는 유연한 HTTP/1.1 호환 웹 서버입니다. Apache 웹 서버는 구성 가능한 다중 플랫폼입니다.

웹 서버 애플리케이션 구축에 대한 자세한 내용은 22장 "인터넷 서버 애플리케이션 만들기"를 참조하십시오.

### Apache 웹 서버 애플리케이션

이 애플리케이션을 선택하면 프로젝트를 Apache 웹 서버로 설정합니다. Apache 웹 서버에 의해 처리된 정보가 클라이언트에 반환됩니다.

*TApacheApplication*은 Apache 서버 애플리케이션에 대한 기본적인 기능을 제공합니다. Apache 서버에서 HTTP 요청을 받으면 *TApacheApplication*이 실행을 시작합니다. *TApacheApplication*은 요청 메시지 (*TApacheRequest*)와 반환할 모든 응답 (*TApacheResponse*)을 나타내는 객체를 만든 다음 응답을 채울 수 있도록 이 객체를 디스패처(dispatcher)로 전달한 다음 아직 보내지 않았다면 채운 응답을 보냅니다.

### CGI 독립형 웹 서버 애플리케이션

CGI 웹 서버 애플리케이션은 표준 입력에서 클라이언트의 요청을 받아 처리한 다음, 클라이언트에 보낼 표준 출력으로 결과를 서버에 돌려 보내는 콘솔 애플리케이션입니다.

## 데이터 모듈 사용

*데이터 모듈*은 논비주얼(nonvisual) 컴포넌트를 포함하는 특수한 폼과 비슷합니다. 데이터 모듈의 모든 컴포넌트는 비주얼 컨트롤과 함께 일반적인 폼에 둘 수 있습니다. 그러나 데이터베이스와 시스템 객체 그룹을 재사용하거나 데이터베이스 연결과 비즈니스 로직을 처리하는 애플리케이션의 일부를 분리하려는 경우, 데이터 모듈은 간편한 구성 틀을 제공합니다.

데이터 모듈의 종류에는 표준 데이터 모듈과 웹 모듈, 두 가지가 있습니다.

- 표준 데이터 모듈을 사용하여 애플리케이션을 보다 쉽게 구성하고 작성할 수 있습니다. 표준 데이터 모듈은 단일 및 2계층(2 tier) 데이터베이스 애플리케이션에 특히 유용하지만 모든 애플리케이션에서 논비주얼(nonvisual) 컴포넌트를 구성하는 데 사용할 수 있습니다. 자세한 내용은 아래의 "데이터 모듈 만들기"를 참조하십시오.

- 웹 모듈은 웹 서버 애플리케이션의 기본을 형성합니다. 웹 모듈은 HTTP 응답 메시지의 내용을 만드는 컴포넌트를 유지하는 것 외에 클라이언트 애플리케이션에서 HTTP 메시지의 디스패칭을 처리합니다. 웹 모듈 사용에 대한 자세한 내용은 "인터넷 서버 애플리케이션 만들기"를 참조하십시오.

## 데이터 모듈 만들기

데이터 모듈을 만들려면, File|New를 선택하고 Data Module을 두 번 누릅니다. Kylix는 빈 데이터 모듈을 열고, 코드 에디터에서 새 모듈을 만들 유닛 파일을 표시한 다음 현재 프로젝트에 모듈을 추가합니다. 기존 데이터 모듈을 다시 열 때, Kylix는 데이터 모듈 창에 컴포넌트를 표시합니다.

디자인 타임에 컴포넌트 팔레트에서 논비주얼(nonvisual) 컴포넌트를 선택한 다음 데이터 모듈 창에서 더블 클릭하여 논비주얼(nonvisual) 컴포넌트를 데이터 모듈에 추가할 수 있습니다. 데이터 모듈 창에서 컴포넌트를 선택하면, 폼에 있는 컴포넌트처럼 Object Inspector에서 컴포넌트의 속성을 편집할 수 있습니다. 데이터 모듈에서 컴포넌트에 대해 설정한 속성은 모듈을 사용하는 애플리케이션의 모든 폼에 일관성 있게 적용됩니다.

데이터 모듈의 이름은 데이터 모듈 창의 제목 표시줄에 나타납니다. 기본 이름은 DataModule $n$ 이고, 여기서  $n$ 은 프로젝트에서 사용되지 않은 유닛의 최소 번호를 나타냅니다. 예를 들어, 새 프로젝트를 시작하면 다른 애플리케이션을 구축하기 전에 모듈을 프로젝트에 추가하고, 데이터 모듈의 이름은 기본적으로 DataModule2가 됩니다. DataModule2에 대한 해당 유닛 파일은 Unit2입니다(Unit1은 폼입니다). 데이터 모듈 창을 선택하고 Object Inspector에서 Name 속성을 편집하여 데이터 모듈의 이름을 변경할 수 있습니다.

## 데이터 모듈에서 비즈니스 룰 만들기

데이터 모듈의 유닛 파일에서, 비즈니스 룰을 캡슐화하는 전역 루틴뿐만 아니라 모듈의 컴포넌트에 대한 이벤트 핸들러를 포함한 메소드를 작성할 수 있습니다. 예를 들어, 월별, 분기별, 연도별 회계 장부를 작성하는 프로시저를 작성할 수 있고, 모듈의 컴포넌트에 대한 이벤트 핸들러나 모듈을 사용하는 유닛에서 해당 프로시저를 호출할 수 있습니다.

## 폼에서 데이터 모듈 액세스

폼에서 비주얼 컨트롤을 데이터 모듈에 연결하려면, 먼저 데이터 모듈을 폼의 **uses** 절에 추가해야 합니다. 다음과 같은 여러가지 방법으로 수행할 수 있습니다.

- 코드 에디터에서, 폼의 유닛 파일을 열고 데이터 모듈의 이름을 **interface** 섹션에 있는 **uses** 절에 추가합니다.
- File|Use Unit을 선택한 다음 모듈 이름을 입력하거나 Use Unit 대화 상자에 있는 목록 상자에서 모듈 이름을 선택합니다.
- 데이터 모듈에서 테이블 또는 쿼리 컴포넌트를 더블 클릭하여 Fields Editor를 엽니다. Fields Editor에서 필드를 폼으로 끌어 놓습니다. Kylix는 모듈을 폼의 **uses** 절

에 추가할지를 사용자에게 물어 본 다음, 편집 상자와 같은 필드용 컨트롤을 만듭니다.

## 템플릿 프로그래밍

템플릿 프로그래밍은 공통적으로 사용되는 "뼈대" 구조를 세우는 것입니다. 템플릿을 소스 코드에 추가한 다음 필요한 코드를 삽입할 수 있습니다. 예를 들어, 코드에서 **for** 순환문을 사용하기 위해 다음 템플릿을 삽입할 수 있습니다.

```
for := to do
begin

end;
```

코드 에디터에 코드 템플릿을 삽입하려면 *Ctrl-J*를 누르고 사용하려는 템플릿을 선택합니다. 사용자가 직접 만든 템플릿을 이 컬렉션에 추가할 수도 있습니다. 다음과 같은 방법으로 템플릿을 추가합니다.

- 1 Tools|Editor Options를 선택합니다.
- 2 Code Insight 탭을 클릭합니다.
- 3 템플릿 섹션에서 Add를 클릭합니다.
- 4 바로 가기 이름을 선택하고 새 템플릿에 대한 간단한 설명을 입력합니다.
- 5 템플릿 코드를 Code 텍스트 상자에 추가합니다.
- 6 OK를 클릭합니다.

## 코드 공유: Object Repository 사용

Object Repository (Tools|Repository)를 사용하면 폼, 대화 상자, 프레임 및 데이터 모듈을 쉽게 공유할 수 있습니다. Object Repository는 폼과 프로젝트를 만들 때 사용자를 안내하는 마법사 및 새 프로젝트에 대한 템플릿도 제공합니다. 이 레포지토리는 Repository 및 New Items 대화 상자에 나타나는 항목에 대한 참조를 포함하는 텍스트 파일인 delphi60dro 파일(기본적으로 .borland 디렉토리에 있음)에서 유지 관리됩니다.

### 프로젝트 내에서 항목 공유

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

### 항목 추가: Object Repository

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

- 1 항목이 프로젝트이거나 프로젝트에 있는 경우에 해당 프로젝트를 엽니다.

- 2 프로젝트에서 Project|Add To Repository를 선택합니다. 폼 또는 데이터 모듈에서 항목을 마우스 오른쪽 버튼으로 클릭하고 Add To Repository를 선택합니다.
- 3 설명, 제목 및 작성자를 입력합니다.
- 4 New Items 대화 상자에서 항목을 나타내려는 페이지를 결정한 다음, 페이지의 이름을 입력하거나 Page 콤보 상자에서 페이지의 이름을 선택합니다. 존재하지 않는 페이지 이름을 입력한 경우, Kylix는 새 페이지를 생성합니다.
- 5 Object Repository에 있는 객체를 나타내는 아이콘을 선택하려면 Browse를 선택합니다.
- 6 OK를 선택합니다.

## 팀 환경에서 객체 공유

네트워크 상에서 레포지토리를 사용 가능하게 하여 작업 그룹 또는 개발 팀과 객체를 공유할 수 있습니다. 공유 레포지토리를 사용하려면, 팀원 모두 Environment Options 대화 상자에서 동일한 Shared Repository 디렉토리를 선택해야 합니다.

- 1 Tools|Environment Options를 선택합니다.
- 2 Preferences 페이지에서 Shared Repository 패널을 찾습니다. Directory 편집 상자에서 공유 레포지토리를 검색하려는 디렉토리를 입력합니다. 반드시 모든 팀원이 액세스할 수 있는 디렉토리를 지정합니다.

항목이 레포지토리에 처음으로 추가되면, delphi60dro 파일이 없는 경우 Kylix는 Shared Repository 디렉토리에 이 파일을 만듭니다.

**참고** Object Repository 디렉토리(objrepos)에 대한 액세스 권한을 올바르게 설정하는 것이 중요합니다. 왜냐하면 사용자가 디렉토리에 대한 쓰기 권한이 없으면 항목을 추가할 수 없기 때문입니다. 따라서, 공통 Object Repository 액세스를 여러 사용자에게 허용하려면, 그룹을 만들고 그룹의 멤버에게 objrepos 디렉토리에 대한 읽기 쓰기 액세스 권한을 주어야 합니다. 예를 들어, "dev" 그룹의 이름을 지정할 때, 다음을 입력하여 명령 줄에서 사용 권한을 설정합니다.

```
cd <install directory>
chmod -R 775 objrepos
chgrp -R dev objrepos
```

자세한 내용은 *group(5) man* 페이지를 참조하십시오.

## 프로젝트에서 Object Repository 항목 사용

Object Repository의 항목에 액세스하려면 File|New를 선택합니다. New Items 대화 상자가 나타나 모든 사용 가능한 항목을 보여줍니다. 사용하려는 항목의 종류에 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, 복사본의 변경된 내용은 원래의 Object Repository 항목에 영향을 미치지 않습니다.

Copy는 프로젝트 템플릿에서 사용할 수 있는 유일한 옵션입니다.

## 항목 상속

Inherit를 선택하고 Object Repository의 선택된 항목에서 새 클래스를 파생시켜 프로젝트에 추가합니다. 프로젝트를 다시 컴파일할 때, Object Repository에서 항목의 모든 변경된 내용은 프로젝트의 항목 변경 내용과 더불어 파생된 클래스에 반영됩니다. 파생된 클래스의 변경 내용은 Object Repository의 공유 항목에 영향을 미치지 않습니다.

상속은 폼, 대화 상자 및 데이터 모듈에 사용할 수 있지만 프로젝트 템플릿에는 사용할 수 없습니다. 상속은 동일한 프로젝트 내에서 항목을 재사용할 때 사용하는 유일한 옵션입니다.

## 항목 사용

선택된 항목 자체를 프로젝트의 일부가 되게 하려면 Use를 선택합니다. 프로젝트에서 항목의 변경된 내용은 Inherit 옵션 또는 Use 옵션으로 항목을 추가했던 다른 모든 프로젝트에 나타납니다. 이 옵션은 신중하게 선택하십시오.

Use 옵션은 폼, 대화 상자 및 데이터 모듈에 사용할 수 있습니다.

## 프로젝트 템플릿 사용

---

템플릿은 애플리케이션 개발의 첫 단계에서 사용할 수 있도록 미리 디자인됩니다. 다음과 같은 방법으로 템플릿에서 새 프로젝트를 만듭니다.

- 1 File|New 를 선택하여 New Items 대화 상자를 표시합니다.
- 2 Projects 탭을 선택합니다.
- 3 원하는 프로젝트 템플릿을 선택하고 OK 를 클릭합니다.
- 4 Select Directory 대화 상자에서 새 프로젝트 파일의 디렉토리를 지정합니다.

Kylix는 템플릿 파일을 지정된 디렉토리에 복사하는데, 이 템플릿 파일은 수정할 수 있습니다. 프로젝트 파일 원본은 변경된 내용에 영향을 받지 않습니다.

## 공유 항목 수정

---

Object Repository에서 항목을 수정하는 경우, 변경된 내용은 Use 옵션 또는 Inherit 옵션으로 항목을 추가했던 기존 프로젝트 뿐만 아니라 동일한 항목을 사용하는 향후의 모든 프로젝트에 영향을 미칩니다. 변경된 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 Object Repository에 추가합니다.



- 항목으로부터 컴포넌트, 공유 객체, 컴포넌트 템플릿 또는 프레임을 만듭니다. 컴포넌트 또는 공유 객체를 만드는 경우에 다른 개발자와 공유할 수 있습니다.

## 기본 프로젝트, 새 폼 및 메인 폼 지정

기본적으로 File|New Application 또는 File|New Form을 선택하면, Kylix는 빈 폼을 표시합니다. 다음과 같이 레포지토리를 재구성하여 빈 폼을 다른 폼으로 변경할 수 있습니다.

- 1 Tools|Repository를 선택합니다.
- 2 기본 프로젝트를 지정하려면 Projects 페이지를 선택하고 Objects 있는 항목을 선택합니다. 그런 다음 New Project 확인란을 선택합니다.
- 3 기본 폼을 지정하려면 Forms 과 같은 Repository 페이지를 선택한 다음 Objects 아래에 있는 폼을 선택합니다. 기본 새 폼 (File|New Form) 을 지정하려면 New Form 확인란을 선택합니다. 새 프로젝트에 사용할 기본 메인 폼을 지정하려면 Main Form 확인란을 선택합니다.
- 4 OK를 클릭합니다.

## 컴포넌트와 컴포넌트 그룹의 재사용

Kylix에서는 CLX 컴포넌트로 수행했던 작업을 저장하고 재사용하는 여러 가지 방법을 제공합니다.

- 컴포넌트 템플릿은 컴포넌트 그룹을 구성하고 저장하기 위한 간단하고 빠른 방법을 제공합니다. 5-11 페이지의 "컴포넌트 템플릿 생성 및 사용"을 참조하십시오.
- 폼, 데이터 모듈 및 프로젝트를 레포지토리에 저장할 수 있습니다. 레포지토리는 재사용 가능한 요소들의 집중식 데이터베이스를 제공하고, 폼을 상속하여 변경 내용을 전파할 수 있습니다. 5-8 페이지의 "코드 공유: Object Repository 사용"을 참조하십시오.
- 프레임을 컴포넌트 팔레트 또는 레포지토리에 저장할 수 있습니다. 프레임은 폼 상속을 사용하고 폼이나 다른 프레임에 포함(embedded)될 수 있습니다. 6-36 페이지의 "프레임 사용"을 참조하십시오.
- 사용자 지정 컴포넌트를 생성하는 것은 코드를 재사용하기 위한 가장 복잡한 방법이지만 가장 많은 유연성을 제공합니다. 25장 "컴포넌트 생성 개요"를 참조하십시오.

## 컴포넌트 템플릿 생성 및 사용

하나 이상의 컴포넌트로 구성된 템플릿을 생성할 수 있습니다. 컴포넌트를 폼에 정렬한 후 폼의 속성을 설정하고, 컴포넌트 코드를 작성한 다음 *컴포넌트 템플릿*으로 저장합니다. 나중에, 컴포넌트 팔레트에서 템플릿을 선택하여 미리 구성된 컴포넌트를 폼 위에 두면, 모든 연결된 속성과 이벤트 처리 코드가 프로젝트에 동시에 추가됩니다.

일단 템플릿을 폼에 두면, 컴포넌트를 직접 폼에 둔 것처럼 각 컴포넌트의 위치를 재지정할 수 있고, 속성을 재설정할 수 있으며, 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 Name 편집 상자에서 컴포넌트 템플릿의 이름을 지정합니다. 기본적으로 단계 2에서 선택된 첫 번째 컴포넌트의 종류 다음에 "Template"이라는 단어가 오는 것이 좋습니다. 예를 들어, 레이블을 선택한 후 편집 상자를 선택하면 제안된 이름은 "TLabelTemplate"이 됩니다. 이름을 변경할 수 있지만, 기존 컴포넌트 이름과 중복되지 않도록 주의하십시오.
- 5 Palette Page 편집 상자에서 템플릿을 넣으려는 컴포넌트 팔레트 페이지를 지정합니다. 존재하지 않는 페이지를 지정하면 템플릿을 저장할 때 새 페이지가 생성됩니다.
- 6 Palette Icon 아래에서 팔레트의 템플릿을 나타내는 비트맵을 선택합니다. 기본적으로 2단계 에서 선택된 첫 번째 컴포넌트의 컴포넌트 종류에서 사용하는 비트맵을 사용합니다. 다른 비트맵을 찾으려면 Change를 누릅니다. 선택한 비트맵은 24픽셀 x 24픽셀 이하여야 합니다.
- 7 OK를 클릭합니다.

컴포넌트 팔레트에서 템플릿을 제거하려면 Component|Configure Palette를 선택합니다.

## CLX 애플리케이션에서 도움말 사용

CLX는 도움말을 직접 제공하지 않지만, 구성 가능한 키를 눌러 트리거된 도움말 요청이 많은 외부 도움말 뷰어(예: Man, Info, HyperHelp) 중 하나에 전달될 수 있는 수단을 제공합니다. 도움말을 지원하려면 애플리케이션 개발자는 *ICustomHelpViewer* 인터페이스(및 옵션으로 자손 인터페이스 중 하나)를 구현하는 클래스를 생성한 다음 이 클래스의 인스턴스를 전역 Help Manager에 등록해야 합니다.

Help Manager는 등록된 뷰어의 목록을 유지 관리하고 다음 두 단계 과정으로 요청을 도움말 뷰어에게 넘깁니다. 먼저, 각 특정한 도움말 키워드 또는 컨텍스트를 지원하는지를 뷰어에게 묻은 다음에 지원할 수 있다고 대답하는 뷰어에게 도움말 요청을 전달합니다. 둘 이상의 뷰어가 키워드를 지원하면, Man과 Info에 대해 동시에 등록된 뷰어를 가진 애플리케이션처럼 Help Manager는 애플리케이션 사용자가 사용할 도움말 뷰어를 선택하는 선택 상자를 표시할 수 있습니다. 그외에는 첫 번째로 응답하는 도움말 시스템을 표시합니다.

### 도움말 시스템 인터페이스

CLX 도움말 시스템은 일련의 인터페이스를 통해 애플리케이션과 도움말 뷰어 간의 통신을 가능하도록 합니다. 이 인터페이스는 모두 *HelpIntfs.pas*에 정의되어 있으며, Help Manager의 구현도 포함하고 있습니다.

*ICustomHelpViewer*는 입력된 키워드에 기반한 도움말을 표시하는 지원과 특정한 뷰어에서 사용 가능한 모든 도움말을 나열하는 목차로 표시하는 지원을 제공합니다.

*IExtendedHelpViewer*는 숫자 도움말 컨텍스트에 기반한 도움말과 항목 표시를 지원합니다. 대부분의 도움말 시스템에서 항목은 상위 수준의 키워드로 사용됩니다(예를 들어, "IntToStr"는 Kylix 도움말 시스템에서 키워드일 수 있지만 "String manipulation routines"는 항목 이름이 될 수 있습니다).

*ISpecialWinHelpViewer*는 Windows에서 실행하는 애플리케이션이 받을 수 있지만 쉽게 일반화되지는 않는 전문화된 WinHelp 메시지에 응답하는 것을 지원합니다. 일반적으로, Windows 환경에서 실행되는 애플리케이션만이 이러한 인터페이스를 구현하는데 이는 비표준 WinHelp 메시지를 광범위하게 사용하는 애플리케이션에만 요구되는 인터페이스입니다.

*IHelpManager*는 애플리케이션의 Help Manager와 통신하고 추가 정보를 요청하는 도움말 뷰어용 메커니즘을 제공합니다. *IHelpManager*는 도움말 뷰어가 자신을 등록하는 시간에 얻어집니다.

*IHelpSystem*은 *TApplication*이 도움말 요청을 도움말 시스템으로 전달하는 메커니즘을 제공합니다. *TApplication*은 애플리케이션을 로드할 때 *IHelpSystem*과 *IHelpManager*를 모두 구현하는 객체의 인스턴스를 얻어서 속성으로 export하는데, 이를 통해 애플리케이션의 다른 코드가 적절한 때 도움말을 요청할 수 있습니다.

*IHelpSelector*는 둘 이상의 뷰어가 도움말 요청을 처리할 수 있을 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 구축되지 않습니다.

## IcustomHelpViewer 구현

---

*ICustomHelpViewer* 인터페이스에는 다음 세 가지 종류의 메소드가 있습니다. 즉, 시스템 수준 정보(예를 들어, 특정한 도움말 요청에 관계되지 않은 정보)를 Help Manager와 통신하는 데 사용하는 메소드, Help Manager가 제공하는 키워드에 기반한 도움말 보여주기과 관련된 메소드, 목차를 표시하는 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 다음 환경에서 이 함수들을 사용하여 호출합니다.

- *ICustomHelpViewer.GetViewerName : String*은 Help Manager가 뷰어의 이름을 알려고 할 때(예를 들어, 애플리케이션이 모든 등록된 뷰어의 목록을 표시하도록 요청 받았을 경우) 호출됩니다. 이러한 정보는 문자열로 반환되고 논리적으로 정적이어야 합니다. 즉, 애플리케이션 실행 중에 변경할 수 없습니다. 멀티바이트 문자 집합은 지원되지 않습니다.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)*는 뷰어를 식별하는 고유한 쿠키를 뷰어에 제공하기 위해 등록 **바로** 다음에 호출됩니다. 이러한 정보는 차후 사용을 위해 저장해야 하고, 뷰어가 스스로 종료하는 경우(Help Manager의 공지에 응답하는 것과는 반대)에 Help Manager에게 쿠키 식별 기능을 제공하여 Help Manager가 모든 뷰어에 대한 참조를 해제하도록 해야 합니다. 쿠키를 제공하는 데 실패하거나 틀린 쿠키를 제공하면, Help Manager는 틀린 뷰어에 대한 참조를 잠재적으로 해제합니다.
- *ICustomHelpViewer.ShutDown*은 Help Manager에 의해 호출되어 Manager가 종료 중이고 도움말 뷰어가 할당한 리소스가 해제되어야 한다는 것을 도움말 뷰어에게 알립니다. 모든 리소스 해제를 이 메소드에 맡기는 것이 좋습니다.
- *ICustomHelpViewer.SoftShutDown*은 Help Manager에 의해 호출되어 도움말 시스템의 외부에서 볼 수 있는 창(예를 들어, 도움말 정보를 표시하는 창)을 뷰어를 언로드하지 않고 종료하도록 요청합니다.

## Help Manager에 정보 요청

도움말 뷰어는 *IHelpManager* 인터페이스를 통해 Help Manager와 통신하는데, 도움말 뷰어가 Help Manager에 등록될 때 인터페이스의 인스턴스는 도움말 뷰어에 반환됩니다. 도움말 뷰어는 *IHelpManager*를 통해 다음 네 가지를 통신할 수 있습니다. 이러한 네 가지에는 현재 활성 컨트롤의 창 핸들에 대한 요청, 현재 활성 컨트롤에 대한 도움말을 포함해야 한다고 Help Manager가 알고 있는 도움말 파일 이름에 대한 요청, 도움말 파일의 경로에 대한 요청, 그리고 Help Manager의 요청이 아닌 다른 것에 대한 응답으로 도움말 뷰어가 자체 종료 중이라는 공지가 있습니다.

*IHelpManager.GetHandle* : *LongInt*는 현재 활성 컨트롤의 핸들을 알아야 하는 경우에 도움말 뷰어에 의해 호출되며 창 핸들이 반환됩니다.

*IHelpManager.GetHelpFile*: *String*은 현재 활성 컨트롤이 도움말을 포함하는 것으로 알고 있는 도움말 파일의 이름을 알고자 할 때 도움말 뷰어에 의해 호출됩니다.

*IHelpManager.GetHelpPath*: *String*은 도움말 파일의 경로를 알고자 할 때 도움말 뷰어에 의해 호출됩니다. 외부 도움말 시스템은 환경 변수 등을 통해 경로를 알 수 있으므로 명시적인 경로 참조를 필요 없기 때문에 이 정보는 기본적으로 제공되지 않습니다.

*IHelpManager.Release*는 도움말 뷰어가 연결 해제 중이라는 것을 Help Manager에 게 알리기 위해 호출됩니다. 이것은 *ICustomHelpViewer.ShutDown*을 통한 요청에 대한 응답으로는 절대 호출되지 않고, 예상하지 못한 연결 해제를 Help Manager에게 알리는 데만 사용됩니다.

## 키워드 방식 도움말 표시

도움말 문의는 *키워드 방식* 도움말로서 도움말 뷰어에 요청 될 때, 뷰어가 특정한 문자열 방식의 도움말을 제공하도록 요청되거나, 또는 *컨텍스트 방식* 도움말로서 요청될 때 뷰어가 특정한 숫자 식별자 방식의 도움말을 제공하도록 요청됩니다. 숫자 도움말 컨텍스트는 WinHelp 시스템을 사용하는 Windows에서 실행 중인 애플리케이션에서 도움말 요청의 기본 형태입니다. 대부분의 Linux 도움말 시스템에서는 수치 도움말 컨텍스트를 인식하지 못하기 때문에 CLX 애플리케이션에서 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 처리하는 세 가지 메소드를 제공합니다.

- *CanShowKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.CanShowKeyword(const HelpString:String):Integer
```

이 구문은 Help Manager에 의해 호출된 세 가지 메소드 중에서 첫 번째로서 뷰어가 문자열에 대한 도움말을 제공하는지 묻기 위해 동일한 문자열을 사용하여 *각각의* 등록된 도움말을 호출하는데, 여기서 뷰어는 도움말 요청에 응답하여 표시할 수 있는 여러 도움말 페이지의 수를 나타내는 정수로 응답할 것으로 예상됩니다. 뷰어는 원하는 메소드를 사용하여 이를 결정할 수 있습니다. Kylix IDE에서 HyperHelp 뷰어는 자체 인덱스를 유지 관리하고 인덱스를 찾는 반면 Man 페이지 뷰어는 man 프로그램을 호출하여 인덱스를 찾습니다. 뷰어가 이 키워드에 대한 도움말을 지원하지 않으면, 뷰어는 0을 반환해야 합니다. 음수는 현재 0를 의미하는 것으로 해석되지만, 이 행동은 차후 릴리스에서는 보장되지 않습니다.

```
ICustomHelpViewer.GetHelpStrings(const HelpString:String):TStringList
```

이 구문은 하나 이상의 뷰어가 그 항목에 대한 도움말을 제공할 수 있는 경우에 Help Manager에 의해 호출됩니다. 뷰어는 *TStringList*를 반환할 것입니다. 반환된 목록에 있는 문자열은 그 키워드에 사용할 수 있는 페이지에 매핑되어야 하지만, 그 매핑의 특성은 뷰어로 결정될 수 있습니다. HyperHelp 뷰어의 경우에 문자열 목록은 항상 정확히 하나의 항목(entry)을 포함하지만(HyperHelp가 자체 인덱싱 및 기타 다른 곳에서는 점 없는(pointless) 복제가 될 복제를 제공함), 온라인 설명서 뷰어의 경우에 문자열 목록은 여러 개의 문자열들로 이루어지는데 해당 키워드에 대한 페이지를 포함하는 설명서의 각 섹션당 문자열 하나입니다.

```
ICustomHelpViewer.ShowHelp(const HelpString:String)
```

이 구문은 도움말 뷰어가 특정한 키워드에 대한 도움말을 표시해야 하는 경우에 Help Manager에 의해 호출됩니다. 이 메소드는 이 작업의 마지막 메소드 호출인데 언제나 *CanShowKeyword*가 먼저 호출된 다음에 호출됩니다.

## 목차 표시

---

*ICustomHelpViewer*는 목차 표시와 관련된 두 가지 메소드를 제공합니다.

- *CanShowTableOfContents*
- *ShowTableOfContents*

이 작업의 이론은 다음 키워드 도움말 요청 함수의 작업과 유사합니다. 즉, Help Manager는 먼저 *ICustomHelpViewer.CanShowTableOfContents: Boolean*을 호출하여 모든 도움말 뷰어를 쿼리한 다음, *ICustomHelpViewer.ShowTableOfContents*를 호출하여 특정한 도움말 뷰어를 호출합니다.

특정한 뷰어가 목차 지원 요청을 거부할 수도 있다는 것에 유의하십시오. 예를 들어, 목차의 개념은 Man 페이지의 작동 방식에 제대로 매핑되지 않기 때문에 이런 현상이 일어날 수도 있습니다. 이와 반대로 HyperHelp 뷰어는 목차를 표시하는 요청을 직접 HyperHelp에 전달함으로써 목차를 지원합니다. 그러나, *ICustomHelpViewer*의 구현이 *CanShowTableOfContents*를 통한 쿼리에 *true*로 응답하고 *ShowTableOfContents*를 통한 요청을 무시하는 것은 타당하지 않습니다.

## IextendedHelpViewer 구현

*ICustomHelpViewer*는 키워드 방식 도움말에 대한 직접적인 지원만을 제공합니다. 일부 도움말 시스템 (특히, WinHelp)은 애플리케이션에 보이지 않는 내부적인 방식으로 *컨텍스트 ID*로 알려진 숫자를 키워드에 연결시켜서 작동합니다. 이 도움말 시스템은 문자열이 아닌 컨텍스트를 사용하여 도움말 시스템을 호출하는 컨텍스트 방식 도움말을 애플리케이션에서 지원해야 하고, 도움말 시스템은 번호 자체를 번역합니다.

CLX로 작성된 애플리케이션은 *ICustomHelpViewer*를 구현하는 객체를 확장함으로써 컨텍스트 방식 도움말이 필요한 시스템과 통신하여 *IExtendedHelpViewer*를 구현할 수 있습니다. 또한 *IExtendedHelpViewer*는 도움말 시스템과의 통신을 지원하여 키워드 검색을 사용하는 대신 상위 수준의 항목으로 직접 이동할 수 있도록 합니다.

*IExtendedHelpViewer*는 네 가지 함수를 사용합니다. 네 가지 함수 중에서 *Understands Context*와 *DisplayHelpByContext*는 컨텍스트 방식 도움말을 지원하는 데 사용되고, 나머지 *UnderstandsTopic*과 *DisplayTopic*은 항목을 지원하는 데 사용됩니다.

애플리케이션 사용자가 F1 키를 누르면 Help Manager는 다음을 호출합니다.

```
IExtendedHelpViewer.UnderstandsContext(const ContextID:Integer;  
const HelpFileName:String):Boolean
```

그리고 현재 활성화된 컨트롤은 키워드 방식 도움말이 아닌 컨텍스트 방식 도움말을 지원합니다. *ICustomHelpViewer.CanShowKeyword*에서와 같이 Help Manager는 등록된 모든 도움말 뷰어를 반복하여 쿼리합니다. 그러나 *ICustomHelpViewer.CanShowKeyword*의 경우와 달리, 둘 이상의 뷰어가 지정된 컨텍스트를 지원하면 주어진 컨텍스트를 지원하는 첫 번째 등록된 뷰어가 호출됩니다.

다음 구문은

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID:Integer;  
const HelpFileName:String)
```

Help Manager가 등록된 도움말 뷰어를 폴링 (polling)한 후에 호출합니다.

항목 지원 함수는 동일한 방식으로 작동합니다.

```
IExtendedHelpViewer.UnderstandsTopic(const Topic:String):Boolean
```

이 구문은 항목을 지원하는지 묻는 도움말 뷰어를 폴링 (polling)하는 데 사용되고,

```
IExtendedHelpViewer.DisplayTopic(const Topic:String)
```

이 구문은 항목에 대한 도움말을 제공할 수 있다고 보고하는 첫 번째 등록된 뷰어를 호출하는 데 사용됩니다.

## IHelpSelector 구현

*IHelpSelector*는 *ICustomHelpViewer*의 짝 (companion) 입니다. 둘 이상의 등록된 뷰어가 입력된 키워드, 컨텍스트 또는 항목에 대한 지원을 제공하거나 목차를 제공할 것이라고 요청하면, Help Manager는 이들 중에서 선택해야 합니다. 컨텍스트 또는 항목의 경우에 언제나 Help Manager는 지원을 제공할 것이라고 요청하는 첫 번째 도움말 뷰어를 선택합니다. 키워드나 목차의 경우에 Help Manager는 기본적으로 첫 번째 도움말 뷰어를 선택합니다. 어떤 뷰어를 선택할지는 애플리케이션으로 조정할 수 있습니다.

Help Manager의 선택을 무시하고 애플리케이션에서 뷰어를 선택하려면 애플리케이션은 *IHelpSelector* 인터페이스의 구현을 제공하는 클래스를 등록해야 합니다. *IHelpSelector*는 *SelectKeyword*와 *TableOfContents*라는 두 가지 함수를 export합니다. 두 함수는 가능한 키워드 일치 사항이나 목차 제공을 요청하는 뷰어의 이름을 포함하는 *TStrings*를 하나씩 인수로 사용합니다. 구현 프로그램은 선택한 문자열을 나타내는 *TStrings*에서 인덱스를 반환해야 합니다.

**참고** 문자열이 재정렬되면 Help Manager에서 혼란스러울 수 있으므로 *IHelpSelector*의 구현 프로그램에서 문자열을 재정렬하지 않는 것이 좋습니다. 도움말 시스템은 오직 *하나*의 HelpSelector만 지원하므로 새 선택기 (selector)가 등록되면 이전에 사용한 기존 선택기는 연결이 끊어집니다.

## 도움말 시스템 객체 등록

Help Manager가 객체들과 통신하려면 *ICustomHelpViewer*, *IExtendedHelpViewer*, *ISpecialWinHelpViewer*, *IHelpSelector*를 구현하는 객체들은 Help Manager에 등록해야 합니다.

도움말 시스템 객체들을 Help Manager에 등록하려면 다음 작업을 수행해야 합니다.

- 도움말 뷰어 등록
- 도움말 선택기 등록

## 도움말 뷰어 등록

객체 구현을 포함하는 유닛은 *HelpIntfs*를 사용해야 합니다. 객체의 인스턴스는 구현 유닛의 **var** 섹션에서 선언해야 합니다.

구현 유닛의 초기화 섹션은 인스턴스 변수를 할당하여 *RegisterViewer* 함수로 전달해야 합니다. *RegisterViewer*는 *ICustomHelpViewer*를 인수로 사용하고 *IHelpManager*를 반환하는 *HelpIntfs.pas*에 의해 export된 평면 함수 (flat function)입니다. *IHelpManager*는 차후 사용을 위해 저장되어야 합니다.



## 도움말 선택기 등록

객체 구현을 포함하는 유닛은 `HelpIntfs`와 `Qforms`를 사용해야 합니다. 객체의 인스턴스는 구현 유닛의 **var** 섹션에서 선언해야 합니다.

구현 유닛의 초기화 섹션은 다음과 같은 전역 Application 객체의 `HelpSystem` 속성을 통해 도움말 선택기를 등록해야 합니다.

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

이 프로시저는 값을 반환하지 않습니다.

## CLX 애플리케이션에서 도움말 사용

다음 단원에서는 CLX 애플리케이션 내에서 도움말을 사용하는 방법에 대해 설명합니다.

### TApplication이 도움말을 처리하는 방법

*TApplication*은 애플리케이션 코드에 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 전달(pass)되는 컨텍스트 또는 키워드를 인수로 사용하고, 도움말 시스템을 나타내는 *TApplication*의 데이터 멤버를 통해 요청을 전달(forward)합니다. 이 데이터 멤버는 읽기 전용 속성인 *HelpSystem*을 통해 직접 액세스할 수 있습니다.

### 컨트롤의 도움말 처리 방법

*TControl*에서 파생되는 모든 컨트롤은 도움말 시스템이 사용하는 *HelpType*, *HelpFile*, *HelpContext*, *HelpKeyword*라는 네 가지 속성을 나타냅니다. *HelpFile*은 컨트롤의 도움말이 있는 파일의 이름을 포함하는 것으로 가정되고, 도움말이 파일 이름에 대해 신경 쓰지 않는 외부 도움말 시스템(예: Man 페이지 시스템)에 있는 경우 속성은 비워져 있어야 합니다.

*HelpType* 속성은 컨트롤의 디자이너가 도움말이 키워드 방식 도움말이나 컨텍스트 방식 도움말을 통해 제공되는 것을 예상하는지 결정하는 열거 타입의 인스턴스를 포함하고, 다른 두 가지 속성은 여기에 연결됩니다. *HelpType*이 *htKeyword*로 설정될 경우, 도움말 시스템은 컨트롤이 키워드 방식 도움말을 사용하는 것으로 예상하고, 도움말 시스템은 *HelpKeyword* 속성의 내용만 봅니다. 이와 반대로 *HelpType*이 *htContext*로 설정될 경우, 도움말 시스템은 컨트롤이 컨텍스트 방식 도움말을 사용하는 것으로 예상하고, 도움말 시스템은 *HelpContext* 속성의 내용만 봅니다.

속성 이외에 컨트롤은 단일 메소드인 *InvokeHelp*를 보여주는데, 이 메소드는 요청을 도움말 시스템으로 전달하기 위해 호출됩니다. 이 메소드는 매개변수를 사용하지 않고 전역 Application 객체에서 컨트롤이 지원하는 도움말의 종류에 해당하는 메소드를 호출합니다.

*TWidgetControl*의 *KeyDown* 메소드가 *InvokeHelp*를 호출하기 때문에 F1 키를 누르면 도움말 메시지는 자동으로 호출됩니다.

## 도움말 시스템 직접 호출

---

이 메커니즘에 의해 나타나지 않은 추가 도움말 시스템 기능에 있어서, *TApplication*은 도움말 시스템에 대해 직접 액세스할 수 있는 읽기 전용 속성을 제공합니다. 이 속성은 인터페이스 *IHelpSystem* 구현의 인스턴스입니다. *IHelpSystem*과 *IHelpManager*는 동일한 객체의 의해 구현되지만 어떤 인터페이스는 애플리케이션이 Help Manager와 통신하는 데 사용되고, 어떤 인터페이스는 도움말 뷰어가 Help Manager와 통신하는 데 사용됩니다.

## IhelpSystem 사용

---

*IHelpSystem*을 통해 애플리케이션은 다음 세 가지를 수행할 수 있습니다.

- Help Manager에 경로 정보 제공
- 새 도움말 선택기 제공
- Help Manager가 도움말을 표시하도록 요청

Help Manager는 플랫폼 독립형 및 도움말 시스템 독립형이므로 도움말 파일의 위치를 확인할 수 없기 때문에 경로 정보를 제공하는 것은 중요합니다. 애플리케이션에서 파일 위치 자체를 확인할 수 없는 외부 도움말 시스템이 도움말을 제공한다고 예상하는 경우, *IHelpSystem.ProvideHelpPath*를 통해 이러한 정보를 제공하여 *IHelpManager.GetHelpPath*를 통해 정보를 사용 가능하도록 만듭니다. 이러한 정보는 도움말 뷰어가 요청하는 경우에만 외부로 전파(propagate)됩니다.

여러 외부 도움말 시스템에서 동일한 키워드에 대한 도움말을 제공할 수 있는 경우에는, 도움말 선택기 할당을 통해 Help Manager는 의사 결정을 위임(delegate)할 수 있습니다. 자세한 내용은 5-18 페이지의 "IHelpSelector 구현" 단원을 참조하십시오.

*IHelpSystem*은 Help Manager에게 도움말을 표시하도록 요청하기 위해 다음 네 가지 프로시저와 한 가지 함수를 export합니다.

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

*Hook*는 전적으로 WinHelp 호환성을 위해 고안되었으므로 CLX 애플리케이션에서 사용해서는 안됩니다. *Hook*를 통해 키보드 방식, 컨텍스트 방식 및 항목 방식 도움말에 대한 요청에 직접적으로 매핑될 수 없는 WM\_HELP 메시지를 처리할 수 있습니다. 나머지 각 메소드는 키워드, 컨텍스트 ID를 인수로 사용하고 도움말에 요청하는 항목과 도움말이 찾으리라고 예상되는 도움말 파일을 사용합니다.

일반적으로 항목 방식 도움말을 요청하지 않는 경우 컨트롤의 *InvokeHelp* 메소드를 통해 도움말 요청을 Help Manager에 전달하는 것이 효과적이고 더욱 확실합니다.

## IDE 도움말 시스템 사용자 지정

Kylix IDE는 CLX 애플리케이션과 정확하게 동일한 방식으로 다양한 도움말 뷰어를 지원하고, 도움말 요청을 Help Manager에 위임한 다음 등록된 도움말 뷰어에 전달(forward)합니다. IDE는 두 가지 도움말 뷰어를 설치하여 출시됩니다. 그 중 하나는 HyperHelp 뷰어로서 이를 통해 Kylix 도움말 파일을 보는 외부 WinHelp 애플레이터인 HyperHelp로 도움말 요청을 전달하고, 다른 하나는 Man 페이지 뷰어로서 이를 통해 대부분의 Unix 시스템에 설치된 Man 시스템에 액세스할 수 있습니다. Kylix 도움말이 작동하는 데 필요하기 때문에 HyperHelp 뷰어는 제거될 수 없습니다. Man 페이지 뷰어는 그 소스를 예제 디렉토리에서 사용할 수 있는 별도의 패키지로 출시됩니다.

IDE에서 새 도움말 뷰어를 설치하려면 한 가지만 제외하고 CLX 애플리케이션에서 하던 것과 똑같이 하십시오. *ICustomHelpViewer*(및 원하는 경우 *IExtendedHelpViewer*)를 구현하는 객체를 작성하여 도움말 요청을 자신이 선택한 외부 뷰어에 전달하고, *ICustomHelpViewer*를 IDE에 등록합니다.

사용자 지정 도움말 뷰어를 IDE에 등록하려면 다음과 같이 합니다.

- 1 도움말 뷰어를 구현하는 유닛이 HelpIntfs.pas를 포함하는지 확인합니다.
- 2 유닛을 IDE에 등록된 디자인 타임 패키지로 만들고, 런타임 패키지를 선택하여 패키지를 만듭니다. (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 유닛의 초기화 섹션에서, 인스턴스가 *RegisterHelpViewer* 함수로 넘겨지는지 확인합니다.



# 애플리케이션 사용자 인터페이스 개발

Kylix에서 컴포넌트 팔레트의 컴포넌트를 선택한 후 폼에 선택한 컴포넌트를 가져다 놓아 사용자 인터페이스 (UI) 를 디자인합니다. 컴포넌트의 속성을 설정하고 컴포넌트의 이벤트 핸들러를 코딩하여 원하는 동작을 수행합니다.

## 애플리케이션 동작 제어

*TApplication*, *TScreen* 및 *TForm*은 프로젝트의 동작을 제어하여 모든 Kylix 애플리케이션의 중요한 요소를 형성하는 클래스입니다. *TApplication* 클래스는 표준 Linux 프로그램의 동작을 캡슐화하는 속성 및 메소드를 제공하여 애플리케이션의 기초를 형성합니다. *TScreen*은 런타임에 화면 해상도 및 사용 가능한 디스플레이 글꼴 등의 시스템 특정 정보를 유지할 뿐만 아니라 로드된 폼과 데이터 모듈을 추적하는 데 사용됩니다. *TForm*의 인스턴스는 애플리케이션 사용자 인터페이스의 구축 블록입니다. 애플리케이션의 창과 대화 상자는 *TForm*을 기반으로 합니다.

## 메인 폼 사용

*TForm*은 GUI 애플리케이션을 작성하기 위한 핵심 클래스입니다. 기본 프로젝트를 표시하는 Kylix를 열거나 프로젝트를 새로 만들 때 사용자가 UI 디자인을 시작할 수 있도록 폼이 표시됩니다.

프로젝트에서 만들고 저장한 첫 번째 폼은 기본적으로 런타임에 생성되는 첫 번째 폼인 프로젝트의 메인 폼이 됩니다. 프로젝트에 폼을 추가할 때 애플리케이션의 메인 폼으로 다른 폼을 지정할 수 있습니다. 또한 한 폼을 메인 폼으로 지정하면 런타임에 테스트하기가 쉽습니다. 폼 생성 순서를 변경하지 않는 한 메인 폼은 실행되는 애플리케이션에서 표시되는 첫 번째 폼이기 때문입니다.

다음과 같은 방법으로 프로젝트 메인 폼을 변경합니다.

- 1 Project|Options를 선택한 다음 Forms 페이지를 선택합니다.
- 2 Main Form 콤보 상자에서 프로젝트 메인 폼으로 사용할 폼을 선택한 다음 OK를 선택합니다.

이제 애플리케이션을 실행하면 메인 폼으로 선택한 폼이 먼저 표시됩니다.

## 폼 추가

프로젝트에 폼을 추가하려면 File|New Form을 선택합니다. Project Manager (View|Project Manager)에 나열된 모든 프로젝트 폼 및 연결 유닛을 볼 수 있고 View|Forms를 선택하여 폼 목록을 표시할 수 있습니다.

## 폼 연결

프로젝트에 폼을 추가하면 프로젝트 파일의 폼에 참조가 추가되지만 프로젝트의 다른 유닛에는 추가되지 않습니다. 새로운 폼을 참조하는 코드를 작성하기 전에 참조 폼의 유닛 파일에 있는 폼에 참조를 추가해야 합니다. 이것을 **폼 연결**이라고 합니다.

폼을 연결하는 일반적인 이유는 해당 폼의 컴포넌트에 대한 액세스를 제공하기 위함입니다. 예를 들면, 경우에 따라 폼 연결을 통해 데이터 모듈의 데이터 액세스 컴포넌트에 연결하는 data-aware 컴포넌트가 들어 있는 폼을 사용할 수 있습니다.

다음과 같은 방법으로 하나의 폼을 다른 폼에 연결합니다.

- 1 다른 폼을 참조할 폼을 선택합니다.
- 2 File|Use Unit을 선택합니다.
- 3 참조할 폼의 폼 유닛 이름을 선택합니다.
- 4 OK를 선택합니다.

하나의 폼을 다른 폼에 연결한다는 것은 하나의 폼 유닛의 **uses** 절에 다른 폼 유닛에 대한 참조가 들어 있어 연결된 폼과 해당 컴포넌트가 이제 연결 폼의 유효 범위(scope)에 있음을 의미합니다.

## 순환 유닛 참조 방지

두 개의 폼이 서로 참조해야 하는 경우 프로그램 컴파일 시 "순환 참조" 오류가 발생할 수 있습니다. 이러한 오류를 방지하려면 다음 중 하나를 수행하십시오.

- 유닛 식별자를 사용하여 두 개의 **uses** 절을 모두 각 유닛 파일의 **implementation** 부분에 둡니다. File|Use Unit 명령을 사용합니다.
- **uses** 절 하나는 **interface** 부분에 두고 다른 하나는 **implementation** 부분에 둡니다. 이 유닛의 **interface** 부분에 다른 폼의 유닛 식별자를 둘 필요는 없습니다.

두 개의 **uses** 절을 모두 각 유닛 파일에 있는 **interface** 부분에 두지 마십시오. 그렇게 할 경우 컴파일 타임에 "순환 참조" 오류가 생깁니다.

## 메인 폼 숨기기

애플리케이션이 처음 시작할 때 메인 폼이 표시되지 않도록 할 수 있습니다. 그렇게 하려면 다음 항목에 설명되어 있는 *Application* 전역 변수를 사용해야 합니다.

다음과 같이 하여, 시작할 때 메인 폼을 숨깁니다.

- 1 Project|View Source를 선택하여 메인 프로젝트 파일을 표시합니다.

- 2 Application.CreateForm에 대한 호출 뒤 그리고 Application.Run에 대한 호출 앞에 다음 줄을 추가합니다.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

**참고** 위에서처럼 런타임에 설정하지 않고, 디자인 타임에 Object Inspector를 사용하여 폼의 *Visible* 속성을 *False*로 설정할 수 있습니다.

## 애플리케이션 레벨에서 작업

*TApplication* 타입의 *Application* 전역 변수는 모든 CLX 기반 애플리케이션 안에 포함됩니다. *Application*은 프로그램의 백그라운드에서 발생하는 다양한 함수를 제공할 뿐만 아니라 애플리케이션을 캡슐화합니다. 예를 들어, *Application*은 프로그램 메뉴에서 도움말 파일을 호출하는 방식을 처리합니다. *TApplication* 작동 방식에 대한 이해는 독립형 애플리케이션 개발자보다 컴포넌트 작성자에게 더 중요하며, 프로젝트를 만들 때 Project|Options 메뉴의 Application 페이지에서 *Application*이 처리하는 옵션을 설정해야 합니다.

또한, *Application*은 애플리케이션 전체에 적용되는 많은 이벤트를 받습니다. 예를 들면, *OnActivate* 이벤트는 애플리케이션이 처음 시작할 때 동작을 수행할 수 있게 해주고, *OnIdle* 이벤트는 애플리케이션이 유휴 상태일 때 백그라운드 프로세스를 수행할 수 있게 해주며, *OnEvent* 이벤트는 이벤트를 가로챌 수 있게 해줍니다. IDE를 사용하여 *Application* 전역 변수의 속성 및 이벤트를 조사할 수는 없지만 다른 컴포넌트인 *TApplicationEvents*가 이벤트를 가로채서 IDE를 이용해 이벤트 핸들러를 제공할 수 있게 해줍니다.

## 애플리케이션의 룩앤필 설정

*TApplication.Style* 속성을 사용하여 애플리케이션의 그래픽 요소의 일반적인 룩앤필 (look and feel)을 지정할 수 있습니다. *TApplication.Style*은 *QStyle.pas*의 *TStyle* 조상과 함께 선언된 *TApplicationStyle*의 인스턴스를 가집니다. *TStyle.DefaultStyle*은 *dsMotif* 또는 *dsWindows*로 설정할 수 있습니다. 예를 들면, 다음 코드에서는 애플리케이션의 일반적인 모습을 Motif 스타일로 설정합니다.

```
Application.Style.DefaultStyle := dsMotif;
```

*TStyle*의 다른 속성은 대부분 버튼, 트랙 표시줄, 분할자, 확인란 등의 드로잉을 제어하는 사용자 지정 이벤트 핸들러입니다. 탭 컨트롤의 개별 탭, 개별 버튼 컨트롤 및 개별 메뉴 항목을 제어할 수도 있습니다. *CLXStyle* 예제 애플리케이션에서는 사용자 지정 드로잉을 구현하는 방법을 보여 줍니다.

## 화면 처리

*Screen*이라는 *TScreen* 타입의 전역 변수는 프로젝트를 생성할 때 만들어집니다. *Screen*은 애플리케이션이 실행 중인 화면의 상태를 캡슐화합니다. *Screen*에서 실행되는 일반 작업은 다음 내용들을 지정합니다.

- 커서 모양

- 실행 중인 애플리케이션의 창 크기
- 화면 장치에서 사용 가능한 글꼴 목록

기본적으로 애플리케이션에서 현재 화면 장치 정보에 기반하는 화면 컴포넌트를 만든 다음 이것을 Screen에 할당합니다.

## 레이아웃 관리

---

가장 간단하게 하려면 폼에서 컨트롤을 두는 위치에 따라 사용자 인터페이스의 레이아웃을 제어합니다. 위치 선택은 컨트롤의 *Top*, *Left*, *Width* 및 *Height* 속성에 반영됩니다. 런타임에 이러한 값을 변경하여 폼에서의 컨트롤 위치 및 크기를 변경할 수 있습니다.

컨트롤은 그 밖에도 콘텐츠 또는 컨테이너에 자동으로 조정할 수 있게 하는 다양한 속성을 가집니다. 따라서 작은 부분들이 통일된 전체로 조화를 이루도록 폼을 레이아웃할 수 있습니다.

두 속성은 부모 컨트롤과 관련해서 컨트롤 위치 및 크기를 지정하는 방법에 영향을 줍니다. *Align* 속성을 사용하여 특정 가장자리를 따라서 부모 컨트롤 내에서 컨트롤을 완벽하게 맞출 수 있으며 다른 모든 컨트롤이 정렬된 후 전체 클라이언트 영역을 채울 수 있습니다. 부모 컨트롤의 크기가 조정되면 해당 부모 컨트롤에 정렬된 컨트롤은 자동으로 크기가 조정되며 특정 가장자리에 맞도록 위치가 유지됩니다.

크기를 변경할 수 있는 항목이 있는 컨트롤은 컨트롤 자체 크기를 글꼴에 맞게 조정하거나 포함된 객체에 맞게 조정할 수 있도록 *AutoSize* 속성을 가집니다.

## 폼 사용

---

Kylix의 IDE에서 폼을 만들면 Kylix는 애플리케이션의 메인 엔트리 포인트에 코드를 포함시켜 메모리에 폼을 자동으로 만듭니다. 일반적으로 이것은 바람직한 동작이므로 변경할 필요가 없습니다. 즉, 메인 창은 프로그램이 실행되는 동안 계속 유지되므로 메인 창에서 폼을 만들 때 기본 Kylix 동작을 변경하지 않습니다.

그러나 프로그램이 실행되는 동안 메모리에서 애플리케이션의 모든 폼을 원하지 않을 수도 있습니다. 즉, 한 번에 애플리케이션의 모든 대화 상자가 메모리에 있는 것을 원하지 않을 경우 대화 상자가 나타나기를 원할 때 동적으로 해당 대화 상자를 만들 수 있습니다.

폼은 모달 폼이거나 모달리스 폼일 수 있습니다. 모달 폼은 사용자가 다른 폼으로 전환하기 전에 상호 작용해야 하는 폼입니다(예: 사용자 입력을 필요로 하는 대화 상자). 모드 없는 폼은 다른 창에 의해 가려지거나 사용자가 닫거나 또는 최소화할 때까지 표시되는 창입니다.



## 메모리에 상주하는 폼의 제어

기본적으로 Kylix는 애플리케이션의 프로젝트 소스 유닛에 다음 코드를 포함하여 메모리에 애플리케이션의 메인 폼을 자동으로 만듭니다.

```
Application.CreateForm(TForm1, Form1);
```

이 함수는 폼과 동일한 이름을 갖는 전역 변수를 만듭니다. 따라서 애플리케이션의 모든 폼은 연결된 전역 변수를 갖습니다. 이 변수는 폼 클래스의 인스턴스에 대한 포인터이고 애플리케이션이 실행하고 있는 동안 폼을 참조하는 데 사용합니다. 폼의 유닛을 **uses** 절에 포함하는 모든 유닛은 이 변수를 통해 폼에 액세스할 수 있습니다.

프로젝트 유닛에서 이 방법으로 만든 모든 폼은 프로그램이 호출되면 나타나며 애플리케이션이 실행되는 동안 메모리에 있습니다.

## 자동 생성된 폼 표시

시작할 때 폼을 만들고 나중에 프로그램 실행 중 일정 시간 동안 표시되지 않기를 원하는 경우 폼의 이벤트 핸들러에서 *ShowModal* 메소드를 사용하여 메모리에 이미 로드된 폼을 표시할 수 있습니다.

```
procedure TMainForm.Button1Click(Sender:TObject);
begin
    ResultsForm.ShowModal;
end;
```

이 경우에는 메모리에 이미 폼이 있으므로 다른 인스턴스를 만들거나 해당 인스턴스를 소멸할 필요가 없습니다.

## 동적으로 폼 생성

항상 모든 애플리케이션이 한 번에 메모리에 있기를 원하지 않을 수도 있습니다. 로드 시 필요한 메모리 양을 줄이려면 폼을 사용해야 할 때만 일부 폼을 만듭니다. 예를 들어, 대화 상자는 사용자가 상호 작용하는 중에만 메모리에 있으면 됩니다.

IDE를 사용하여 실행하는 동안 다른 단계에서 폼을 만들려면 다음과 같이 수행합니다.

- 1 메인 메뉴에서 File|New Form을 선택하여 새로운 폼을 표시합니다.
- 2 Project|Options|Forms 페이지의 자동 생성 폼 목록에서 해당 폼을 제거합니다.  
이렇게 하여 시작 시 폼 호출을 제거합니다. 또는 프로젝트 소스에서 다음 줄을 수동으로 제거할 수 있습니다.

```
Application.CreateForm(TResultsForm, ResultsForm);
```

- 3 모달리스 폼일 경우 폼의 *Show* 메소드를 사용하고, 모달 폼의 경우에는 *ShowModal* 메소드를 사용하여 필요할 때 폼을 호출합니다.

메인 폼의 이벤트 핸들러는 결과 폼의 인스턴스를 만들고 소멸해야 합니다. 결과 폼을 호출하는 방법 중 한 가지로 다음과 같이 전역 변수를 사용합니다. *ResultsForm*이 모달 폼이므로 핸들러는 *ShowModal* 메소드를 사용합니다.

```
procedure TMainForm.Button1Click(Sender:TObject);
begin
    ResultsForm:=TResultForm.Create(self)
    ResultsForm.ShowModal;
    ResultsForm.Free;
end;
```

예제의 이벤트 핸들러는 폼을 닫은 후 해당 폼을 삭제하므로 애플리케이션에서 *ResultsForm*을 사용해야 할 경우에는 폼을 다시 생성해야 합니다. *Show*를 사용하여 폼을 표시한 경우 폼이 열려 있는 동안 *Show*가 반환되기 때문에 이벤트 핸들러 내에서 해당 폼을 삭제할 수 없습니다.

**참고** 생성자를 사용하여 폼을 만드는 경우 해당 폼이 Project Options|Forms 페이지의 자동 생성 폼 목록에 없는지를 반드시 확인해야 합니다. 특히, 목록에서 동일한 이름을 갖는 폼을 삭제하지 않고 새로운 폼을 만드는 경우, Kylix가 시작 시 폼을 만들고 해당 이벤트 핸들러가 폼의 새 인스턴스를 만들어 자동 생성된 인스턴스에 대한 참조를 덮어씁니다. 자동 생성된 인스턴스가 여전히 존재하지만 애플리케이션에서 더 이상 해당 인스턴스에 액세스할 수 없습니다. 이벤트 핸들러가 종료되면 전역 변수는 더 이상 유효한 폼을 가리키지 않습니다. use 전역 변수를 사용하려고 시도할 경우 애플리케이션이 중단될 수 있습니다.

## 창 같은 모달리스 폼 생성

모달리스 폼의 참조 변수는 폼을 사용하는 동안에 계속 존재해야 합니다. 이것은 모달리스 폼의 참조 변수가 전역 유효 범위(scope)를 가져야만 한다는 것을 의미합니다. 대부분의 경우 폼(폼의 이름 속성과 일치하는 변수 이름)을 만들 때 생성된 전역 참조 변수를 사용합니다. 애플리케이션에서 폼의 추가 인스턴스를 필요로 하는 경우 각 인스턴스에 대해 별도의 전역 변수를 선언합니다.

## 로컬 변수로 폼 인스턴스 생성

모달 폼의 고유 인스턴스를 안전하게 만들려면 이벤트 핸들러의 로컬 변수를 새 인스턴스에 대한 참조로 사용합니다. 로컬 변수를 사용하는 경우 *ResultsForm*이 자동 생성된 것인지 여부는 중요하지 않습니다. 이벤트 핸들러의 코드는 전역 폼 변수를 참조하지 않습니다. 예를 들면, 다음과 같습니다.

```
procedure TMainForm.Button1Click(Sender:TObject);
var
    RF:TResultForm;
begin
    RF:=TResultForm.Create(self)
    RF.ShowModal;
    RF.Free;
end;
```

폼의 전역 인스턴스는 이 버전의 이벤트 핸들러에서 사용되지 않습니다.

일반적으로 애플리케이션은 폼의 전역 인스턴스를 사용합니다. 그러나 모달 폼의 새 인스턴스가 필요하고 하나의 함수와 같은 애플리케이션의 제한된 별도 섹션에서 폼을 사용하는 경우 로컬 인스턴스는 폼을 사용하는 가장 안전하고 효율적인 방법입니다.

물론 로컬 변수는 폼을 사용하는 동안 해당 폼이 존재하도록 하는 전역 유효 범위(scope)를 가져야 하기 때문에 모달리스 폼의 이벤트 핸들러에서 로컬 변수를 사용할 수는 없습니다. *Show*는 폼이 열리는 즉시 반환되므로 로컬 변수를 사용한 경우 로컬 변수는 유효 범위(scope)에서 즉시 벗어납니다.

## 폼에 추가 인수 전달

일반적으로 IDE 안에서 애플리케이션에 폼을 만듭니다. 이렇게 만들면 폼은 만드는 폼의 소유자인 *Owner*라는 인수를 취하는 생성자를 가집니다. (소유자는 호출 애플리케이션 객체 또는 폼 객체입니다.) *Owner*는 **nil** 값을 가질 수 있습니다.

추가 인수를 폼에 전달하려면 이 새로운 생성자를 사용하여 별개의 생성자를 만들고 폼을 인스턴스화합니다. 아래 예제의 폼 클래스는 *whichButton* 별도 인수를 갖는 추가 생성자를 표시합니다. 이 새로운 생성자는 수동으로 폼 클래스에 추가됩니다.

```
TResultsForm = class(TForm)
    ResultsLabel:TLabel
    OKButton:TButton;
    procedure OKButtonClick(Sender:TObject);
private
public
    constructor CreateWithButton(whichButton:Integer; Owner:TComponent);
end;
```

이것은 *whichButton* 추가 인수를 전달하는 수동으로 코딩된 생성자입니다. 이 생성자는 *whichButton* 매개변수를 사용하여 폼에서 *Label* 컨트롤의 *Caption* 속성을 설정합니다.

```
constructor CreateWithButton(whichButton:Integer; Owner:TComponent);
begin
    case whichButton of
        1: ResultsLabel.Caption := 'You picked the first button.';
        2: ResultsLabel.Caption := 'You picked the second button.';
        3: ResultsLabel.Caption := 'You picked the third button.';
    end;
end;
```

여러 생성자를 사용하여 폼의 인스턴스를 만드는 경우 용도에 가장 적합한 생성자를 선택할 수 있습니다. 예를 들면, 폼 호출에서 버튼에 대한 다음과 같은 *OnClick* 핸들러는 별도 매개변수를 사용하는 *TResultsForm*의 인스턴스를 만듭니다.

```
procedure TMainForm.SecondButtonClick(Sender:TObject);
var
    rf: TResultsForm;
begin
    rf := TResultsForm.CreateWithButton(2, self);
    rf.ShowModal;
    rf.Free;
end;
```

## 폼에서 데이터 검색

대부분의 실제 애플리케이션들은 여러 폼으로 구성됩니다. 이러한 폼들 사이에 정보를 전달해야 하는 경우가 종종 있습니다. 정보를 받는 폼의 생성자에 대한 매개변수로써 또는 폼 속성에 값을 할당하여 정보를 전달할 수 있습니다. 폼에서 정보를 얻는 방법은 모달 폼인지 모달리스 폼인지에 따라 다릅니다.

### 모달리스 폼에서 데이터 검색

폼의 `public` 멤버 함수를 호출하거나 폼의 속성을 쿼리하면 모달리스 폼에서 정확한 정보를 쉽게 추출할 수 있습니다. 예를 들어, 애플리케이션에 색상 목록("Red", "Green", "Blue" 및 기타 등등)을 갖는 *ColorListBox*라는 목록 상자가 들어 있는 *ColorForm*이라는 모달리스 폼이 있다고 가정합니다. *ColorListBox*에서 선택한 색상 이름 문자열은 사용자가 색상을 새로 선택할 때마다 *CurrentColor*라는 속성에 자동으로 저장됩니다. 폼의 클래스 선언은 다음과 같습니다.

```
TColorForm = class(TForm)
    ColorListBox:TListBox;
    procedure ColorListBoxClick(Sender:TObject);
private
    FColor:String;
public
    property CurColor:String read FColor write FColor;
end;
```

목록 상자의 *OnClick* 이벤트 핸들러인 *ColorListBoxClick*은 목록 상자에서 항목을 새로 선택할 때마다 *CurrentColor* 속성 값을 설정합니다. 이벤트 핸들러는 색상 이름이 들어 있는 목록 상자에서 문자열을 선택하여 *CurrentColor*에 할당합니다. *CurrentColor* 속성은 *SetColor* 설정 함수를 사용하여 *FColor* private 데이터 멤버에 실제 속성 값을 저장합니다.

```
procedure TColorForm.ColorListBoxClick(Sender:TObject);
var
    Index:Integer;
begin
    Index := ColorListBox.ItemIndex;
    if Index >= 0 then
        CurrentColor := ColorListBox.Items[Index]
    else
        CurrentColor := '';
end;
```

이제 애플리케이션 내의 *ResultsForm*이라는 또 다른 폼에서 *ResultsForm*의 *UpdateButton*이라는 버튼을 누를 때마다 *ColorForm*의 현재 선택되어 있는 색상을 알아야 한다고 가정합니다. *UpdateButton*의 *OnClick* 이벤트 핸들러는 다음과 같습니다.

```
procedure TResultForm.UpdateButtonClick(Sender:TObject);
var
    MainColor:String;
begin
    if Assigned(ColorForm) then
    begin
        MainColor := ColorForm.CurrentColor;
```

```

    {do something with the string MainColor}
  end;
end;

```

이벤트 핸들러는 먼저 *Assigned* 함수를 사용하여 *ColorForm*이 존재하는지 확인합니다. 그런 다음 이벤트 핸들러는 *ColorForm*의 *CurrentColor* 속성의 값을 얻습니다.

또는, *ColorForm*이 *GetColor*라는 public 함수를 가지는 경우 다른 폼은 *CurrentColor* 속성(예: *MainColor := ColorForm.GetColor;*)을 사용하지 않고 현재 색상을 얻을 수 있습니다. 실제로는 목록 상자에서 직접 선택하여 다른 폼에서 *ColorForm*의 현재 선택된 색상을 얻지 못하게 할 방법이 없습니다.

```

with ColorForm.ColorListBox do
  MainColor := Items[Index];

```

하지만 속성을 사용하면 *ColorForm*에 대한 인터페이스를 매우 단순하게 만들 수 있습니다. 폼이 *ColorForm*에 대해 알아야 하는 것은 *CurrentColor*의 값을 확인하는 것 뿐입니다.

## 모달 폼에서 데이터 검색

모달리스 폼과 마찬가지로 모달 폼에는 다른 폼에서 필요로 하는 정보가 들어 있는 경우가 많습니다. 가장 일반적인 예는 폼 A가 모달 폼 B를 실행하는 것입니다. 폼 B가 닫혀 있으면 폼 A는 폼 A 처리를 진행할 방법을 결정하기 위해서 사용자가 폼 B에서 수행한 작업을 알아야 합니다. 폼 B가 아직 메모리에 있는 경우 위 예의 모달리스 폼에서처럼 속성이나 멤버 함수를 통해 쿼리할 수 있습니다. 그러나 종료 시에 폼 B가 메모리에서 삭제되는 상황은 어떻게 처리해야 할까요? 폼은 명시적인 반환 값을 갖지 않으므로 폼을 소멸하기 전에 해당 폼의 중요 정보를 보존해야 합니다.

예를 들어, 모달 폼으로 디자인된 *ColorForm* 폼의 수정된 버전을 살펴 봅시다. 클래스 선언은 다음과 같습니다.

```

TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton:TButton;
  CancelButton:TButton;
  procedure CancelButtonClick(Sender:TObject);
  procedure SelectButtonClick(Sender:TObject);
private
  FColor:Pointer;
public
  constructor CreateWithColor(Value:Pointer; Owner:TComponent);
end;

```

폼은 색상 이름 목록을 갖는 *ColorListBox*라는 목록 상자를 가집니다. *SelectButton*이라는 버튼을 누르면 *ColorListBox*에서 현재 선택된 색상 이름을 확인한 다음 폼을 닫습니다. *CancelButton*은 폼을 닫는 버튼입니다.

사용자 정의 생성자가 *Pointer* 인수를 취하는 클래스에 추가되었습니다. 이 *Pointer*가 *ColorForm*을 실행하는 폼이 알고 있는 문자열을 가리킨다고 가정합니다. 이 생성자의 구현은 다음과 같습니다.

```

constructor TColorForm(Value:Pointer; Owner:TComponent);
begin
    FColor := Value;
    String(FColor^) := '';
end;

```

생성자는 *FColor* private 데이터 멤버에 대한 포인터를 저장하고 문자열을 빈 문자열에 대해 초기화합니다.

**참고** 위의 사용자 정의 생성자를 사용하려면 폼을 명시적으로 만들어야 합니다. 애플리케이션을 시작할 때 자동적으로 생성되는 폼일 수 없습니다. 자세한 내용은 6-4페이지의 "폼의 메모리 로드 제어"를 참조하십시오.

애플리케이션의 목록 상자에서 색상을 선택하고 *SelectButton*을 눌러 선택 내용을 저장한 다음 폼을 닫습니다. *SelectButton*의 *OnClick* 이벤트 핸들러는 다음과 같습니다.

```

procedure TColorForm.SelectButtonClick(Sender:TObject);
begin
    with ColorListBox do
        if ItemIndex >= 0 then
            String(FColor^) := ColorListBox.Items[ItemIndex];
        end;
    Close;
end;

```

이벤트 핸들러는 생성자에게 전달된 포인터가 참조하는 문자열에 선택한 색상 이름을 저장합니다.

*ColorForm*을 효과적으로 사용하려면 호출하는 폼은 생성자에 기존 문자열에 대한 포인터를 전달해야 합니다. 예를 들어, 불러진 *ResultsForm*의 *UpdateButton*이라는 버튼에 대한 응답으로 *ResultsForm*이라는 폼에 의해 *ColorForm*이 인스턴스화된다고 가정합니다. 이벤트 핸들러는 다음과 같을 것입니다.

```

procedure TResultsForm.UpdateButtonClick(Sender:TObject);
var
    MainColor:String;
begin
    GetColor(Addr(MainColor));
    if MainColor <> '' then
        {do something with the MainColor string}
    else
        {do something else because no color was picked}
    end;

    procedure GetColor(PColor:Pointer);
    begin
        ColorForm := TColorForm.CreateWithColor(PColor, Self);
        ColorForm.ShowModal;
        ColorForm.Free;
    end;

```

*UpdateButtonClick*은 *MainColor* 라는 문자열을 만듭니다. *MainColor* 의 주소는 *ColorForm*을 만들어 *MainColor*에 대한 포인터를 생성자에 대한 인수로 전달하는 *GetColor* 함수에 전달됩니다. *ColorForm*은 닫는 즉시 삭제되지만 선택된 색상 이름은 해당 색상이 선택된 것처럼 *MainColor*에 여전히 보존됩니다. 그렇지 않을 경우, *MainColor*에는 색상을 선택하지 않고 *ColorForm*을 종료했음을 명백히 나타내는 빈 문자열이 들어 있습니다.

이 예제는 하나의 문자열 변수를 사용하여 모달 폼의 정보를 가집니다. 물론 필요할 경우 더 복잡한 객체를 사용할 수도 있습니다. 변경하거나 선택하지 않고(예: 빈 문자열에 *MainColor* 기본값 설정) 모달 폼을 닫았는지 여부를 호출하는 폼에게 항상 알려야 합니다.

## 프레임 사용

폼과 마찬가지로 프레임(*TFrame*)은 다른 컴포넌트의 컨테이너입니다. 프레임은 자동 인스턴스화, 컴포넌트 소멸, 컴포넌트 속성 동기화를 위한 동일한 부모-자식 관계에 대해서 폼과 동일한 소유 관계 메커니즘을 사용합니다.

그러나 어떤 면에서 프레임은 폼보다 사용자 지정된 컴포넌트와 더 유사합니다. 프레임은 쉽게 재사용할 수 있도록 컴포넌트 팔레트에 저장할 수 있고 폼이나 다른 컨테이너 객체 내에 중첩할 수 있습니다. 프레임을 만들고 저장한 후 프레임은 계속 유닛으로 기능하고 프레임에 들어 있는 컴포넌트(다른 프레임 포함)의 변경 내용을 상속합니다. 다른 프레임이나 폼에 포함된 프레임은 자신이 파생된 프레임의 변경 내용을 계속 상속합니다.

프레임은 애플리케이션의 여러 위치에 사용되는 컨트롤 그룹을 구성하는 데 유용합니다. 예를 들어, 여러 폼에서 사용되는 비트맵이 있는 경우 해당 비트맵을 프레임에 넣으면 비트맵의 복사본 하나만이 애플리케이션의 리소스에 있습니다. 또한 프레임이 있는 테이블을 편집하기 위한 편집 필드 집합을 기술하고 테이블에 데이터를 입력할 때마다 해당 편집 필드를 사용할 수 있습니다.

## 프레임 생성

빈 프레임을 만들려면 *File|New Frame*을 선택하거나, *File|New*를 선택한 다음 *Frame*을 더블 클릭합니다. 그런 다음 새 프레임에 컴포넌트 혹은 다른 프레임을 가져다 놓습니다.

필수적이지는 않지만 프레임을 프로젝트 일부로 저장하는 것이 좋습니다. 폼은 없고 프레임만 들어 있는 프로젝트를 만들려면 *File|New Application*을 선택하고 새 폼과 유닛을 저장하지 않고 닫은 다음 *File|New Frame*을 선택하고 프로젝트를 저장합니다.

**참고** 프레임을 저장할 때 *Unit1*, *Project1* 등 기본 이름을 사용하지 않아야 합니다. 이러한 기본 이름은 나중에 프레임을 사용하려고 할 때 충돌을 일으킬 수 있기 때문입니다.

디자인 타임에 *View|Forms*를 선택하고 프레임을 선택하면 현재 프로젝트에 있는 프레임을 표시할 수 있습니다. 폼과 데이터 모듈에서 마우스 오른쪽 버튼을 클릭하고 *View as Form* 또는 *View as Text*를 선택하면 폼 디자이너와 프레임의 폼 파일 사이를 토글할 수 있습니다.

## 컴포넌트 팔레트에 프레임 추가

프레임을 컴포넌트 팔레트에 컴포넌트 템플릿으로 추가합니다. 컴포넌트 팔레트에 프레임을 추가하려면 폼 디자이너에서 프레임을 열고 (다른 컴포넌트에 포함된 프레임은 이러한 용도로 사용할 수 없음), 프레임을 마우스 오른쪽 버튼으로 클릭한 다음 Add to Palette를 선택합니다. Component Template Information 대화 상자가 열리면 이름, 팔레트 페이지 및 새 템플릿의 아이콘을 선택합니다.

## 프레임 사용 및 수정

애플리케이션에서 프레임을 사용하려면 폼에 직접 또는 간접으로 프레임을 두어야 합니다. 폼, 다른 프레임 또는 패널 및 스크롤 상자 같은 다른 컨테이너 객체에 프레임을 직접 추가할 수 있습니다.

폼 디자이너에는 애플리케이션에 프레임을 추가하는 방법이 두 가지 있습니다.

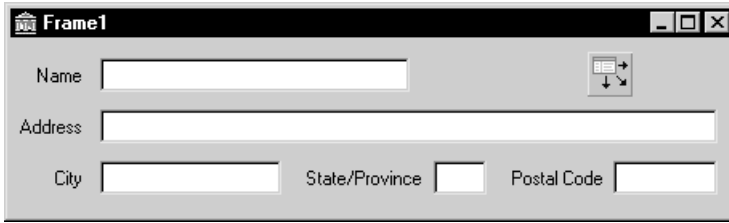
- 컴포넌트 팔레트에서 프레임을 선택하여 폼, 다른 프레임 또는 다른 컨테이너 객체에 가져다 놓습니다. 필요한 경우 폼 디자이너에서 프로젝트에 프레임 유닛 파일을 포함할 권한이 있는지 묻습니다.
- 컴포넌트 팔레트의 Standard 페이지에서 *Frames*을 선택한 후 폼 또는 다른 프레임에 클릭합니다. 프로젝트에 이미 들어 있는 프레임 목록을 보여주는 대화 상자가 나타나면 프레임을 선택한 다음 OK를 클릭합니다.

폼 또는 다른 컨테이너에 프레임을 가져다 놓으면 Kylix는 선택한 프레임의 자손인 새로운 클래스를 선언합니다. (마찬가지로, 프로젝트에 새로운 폼을 추가하면 Kylix는 *TForm*의 자손인 새로운 클래스를 선언합니다.) 이는 원래 (조상) 프레임에 나중에 발생한 변경 내용은 포함된 프레임에 전파되지만 포함된 프레임의 변경 내용은 조상 프레임에 역전파되지 않는다는 것을 의미합니다.

예를 들어, 두 개 이상의 애플리케이션에서 반복적으로 사용하기 위해 데이터 액세스 컴포넌트 및 data-aware 컨트롤을 그룹화하기를 원한다고 가정합니다. 이 작업을 수행하는 한 가지 방법은 컴포넌트를 컴포넌트 템플릿에 모으는 것이지만 템플릿을 사용하기 시작하고 나중에 컨트롤 배열을 바꾸고자 할 경우에는 다시 되돌아가서 해당 템플릿이 위치한 각 프로젝트를 수동으로 변경해야 합니다.

한편, 데이터베이스 컴포넌트들을 템플릿 대신에 프레임에 놓을 경우에는 원래 프레임에 대한 변경 내용이 프로젝트가 재컴파일될 때 포함된 자손 프레임에 자동으로 전파됩니다. 동시에 원래 프레임이나 다른 포함된 자손 프레임에 영향을 주지 않고 포함된 프레임을 수정할 수 있습니다. 포함된 프레임 수정에 대한 유일한 제한은 컴포넌트를 프레임에 추가할 수 없다는 것입니다.



**그림 6.1** data-aware 컨트롤과 데이터 소스 컴포넌트를 갖는 프레임

프레임은 유지를 간편하게 해줄 뿐만 아니라 리소스를 보다 효율적으로 사용할 수 있게 해줍니다. 예를 들어, 애플리케이션에서 비트맵 또는 다른 그래픽을 사용하려면 *TImage* 컨트롤의 *Picture* 속성에 그래픽을 로드할 수 있습니다. 그러나 하나의 애플리케이션에서 동일한 그래픽을 반복적으로 사용하는 경우 폼에 각 *Image* 객체를 두면 폼의 리소스 파일에 그래픽의 또 다른 복사본이 추가됩니다. (이는 *TImage.Picture*를 한 번 설정하고 *Image* 컨트롤을 컴포넌트 템플릿으로 저장하는 경우에도 마찬가지입니다.) 보다 나은 해결책은 *Image* 객체를 프레임에 가져다 놓고, 해당 객체에 그래픽을 로드한 후 그래픽이 나타날 프레임을 사용하는 것입니다. 이렇게 하면 크기가 작은 폼 파일이 생기며 원래 프레임의 *Image*를 수정하면 해당되는 모든 위치에서 그래픽을 변경할 수 있는 추가적인 이점이 있습니다.

## 프레임 공유

다른 개발자와 두 가지 방법으로 프레임을 공유할 수 있습니다.

- Object Repository에 프레임을 추가합니다.
- 프레임 유닛(.pas)과 폼(.xfm) 파일을 배포합니다.

Repository에 프레임을 추가하려면 프레임이 들어 있는 프로젝트를 열고 폼 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Add to Repository를 선택합니다. 자세한 내용은 2-35 페이지의 "Object Repository 사용"을 참조하십시오.

프레임 유닛과 폼 파일을 다른 개발자에게 보내면 그 개발자는 해당 프레임 유닛과 폼 파일을 열고 컴포넌트 팔레트에 추가할 수 있습니다. 프레임 내에 다른 프레임이 포함되어 있는 경우 해당 프레임을 프로젝트의 일부로 열어야만 팔레트에 추가할 수 있습니다.

## 작업 목록 사용

작업 목록에는 사용자가 수행한 작업에 대해 애플리케이션에서 취할 수 있는 작업 목록이 들어 있습니다. 작업 객체를 사용하여 사용자 인터페이스에서 애플리케이션에 의해 실행된 함수를 중앙 집중화합니다. 이렇게 하면 애플리케이션 상태에 따라 작업을 중앙 집중화된 방식으로 설정 및 해제할 수 있을 뿐만 아니라 (예를 들어, 툴바 버튼과 메뉴 항목이 동일한 작업을 수행할 때) 작업을 수행하는 공통적인 코드를 공유할 수 있습니다.

## 작업(action)의 개념

작업은 메뉴 명령이나 툴바 버튼 같은 하나 이상의 사용자 인터페이스 요소에 대응합니다. 작업에는 다음 두 가지 기능이 있습니다. (1) 컨트롤이 설정 또는 선택되어 있는지 여부 같은 사용자 인터페이스 요소에 대한 일반적인 속성을 나타내며, (2) 컨트롤이 실행될 때 (예를 들어, 애플리케이션 사용자가 버튼을 누르거나 메뉴 항목을 선택할 때) 응답합니다.

작업은 다른 컴포넌트와 연결됩니다.

- **Clients:** 하나 이상의 클라이언트가 작업을 사용합니다. 클라이언트는 종종 메뉴 항목 또는 버튼(예를 들어, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton* 등)을 나타냅니다. 클라이언트가 마우스 클릭 같은 사용자 명령을 받으면 연결 작업이 초기화됩니다. 일반적으로 클라이언트의 *OnClick* 이벤트는 해당 작업의 *Execute* 이벤트와 연결됩니다.
- **Target:** 작업은 대상(target)에서 작동합니다. 대상은 보통 메모나 데이터 컨트롤 같은 컨트롤입니다. 컴포넌트 작성자는 모듈 애플리케이션을 보다 많이 만들기 위해 유닛을 디자인하고, 사용하고, 패키징하는 컨트롤 요구에 따른 작업을 만들 수 있습니다. 모든 작업이 대상을 사용하지는 않습니다. 예를 들어, 표준 도움말 작업은 대상을 무시하고 도움말 시스템을 실행합니다.

대상은 컴포넌트일 수도 있습니다. 예를 들어, 데이터 컨트롤은 대상을 연결된 데이터셋으로 변경합니다.

클라이언트는 작업에 영향을 미칩니다. 작업은 클라이언트가 작업을 실행할 때 응답합니다. 작업은 클라이언트에도 영향을 미칩니다. 작업 속성은 클라이언트 속성을 동적으로 업데이트합니다. 예를 들어, 런타임에 작업이 해제되어 있으면(작업의 *Enabled* 속성을 *False*로 설정하여), 해당 작업의 모든 클라이언트가 해제되어 회색으로 나타납니다.

작업 목록 객체를 더블 클릭하면 표시되는 Action List 편집기를 사용하여 작업을 추가, 삭제 및 재배열합니다. 이러한 작업은 나중에 클라이언트 컨트롤에 연결됩니다.

## 작업 목록 설정

작업 목록 설정은 다음과 같은 기본 단계를 이해하고 나면 매우 쉽습니다.

- 작업 목록을 만듭니다.
- 작업 목록에 작업을 추가합니다.
- 작업의 속성을 설정합니다.
- 작업을 클라이언트에 첨부합니다.

보다 자세한 단계는 다음과 같습니다.

- 1 폼 또는 데이터 모듈에 *TActionList* 객체를 가져다 놓습니다. (ActionList는 컴포넌트 팔레트의 Standard 페이지에 있습니다.)
- 2 *TActionList* 객체를 더블 클릭하여 Action List 편집기를 표시합니다.
  - a 편집기에서 이미 정의된 작업 목록 중 하나를 사용합니다. 마우스 오른쪽 버튼을 클릭하여 New Standard Action을 선택합니다.

- b 이미 정의된 작업은 Standard Actions 대화 상자에서 Dataset, Edit, Help 및 Window 등의 범주별로 구성됩니다. 작업 목록에 추가할 모든 표준 작업을 선택한 다음 OK를 클릭합니다.

또는,

- c 사용자 고유의 새 작업을 만듭니다. 마우스 오른쪽 버튼을 클릭하여 New Action을 선택합니다.
- 3 Object Inspector에서 각 작업의 속성을 설정합니다. (설정하는 속성은 작업의 모든 클라이언트에 영향을 미칩니다.)

*Name* 속성은 작업을 식별하고, 다른 속성 및 이벤트(*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible* 및 *Execute*)는 클라이언트 컨트롤의 속성 및 이벤트에 해당합니다. 클라이언트의 해당 속성은 반드시는 아니지만 일반적으로 해당 클라이언트 속성과 동일한 이름입니다. 예를 들어, 작업의 *Enabled* 속성은 *TToolButton*의 *Enabled* 속성에 해당합니다. 그러나 작업의 *Checked* 속성은 *TToolButton*의 *Down* 속성에 해당합니다.

- 4 이미 정의된 작업을 사용하는 경우 작업에는 자동으로 발생하는 표준 응답이 들어 있습니다. 사용자 고유의 작업을 만드는 경우 실행 시 작업 응답 방법을 정의하는 이벤트 핸들러를 작성해야 합니다. 자세한 내용은 6-15 페이지의 "작업 실행 시 발생하는 이벤트"를 참조하십시오.
- 5 작업 목록의 작업을 해당 작업이 필요한 클라이언트에 첨부합니다.

- 폼이나 데이터 모듈에서 버튼이나 메뉴 항목 같은 컨트롤을 클릭합니다. Object Inspector에서 *Action* 속성은 사용 가능한 작업 목록을 표시합니다.
- 원하는 작업을 선택합니다.

*TEditDelete* 또는 *TDataSetPost* 같은 표준 작업은 모두 사용자가 예상하는 작업을 수행합니다. 필요한 경우 표준 작업 전체의 사용 방법에 대한 자세한 내용은 온라인 참조 도움말을 참조하십시오. 사용자 고유의 작업을 작성하는 경우에는 작업 실행 시 발생할 일에 대해 자세히 알고 있어야 합니다.

## 작업 실행 시 발생하는 이벤트

이벤트를 실행하면 일반적인 작업에 대한 일련의 이벤트가 일어납니다. 그런 다음 이벤트가 작업을 처리하지 않으면 또 다른 일련의 이벤트가 일어납니다.

## 이벤트에 응답

클라이언트 컴포넌트 또는 컨트롤이 선택되어 있거나 작동되어 있으면 응답할 수 있는 작업에 대해 일련의 이벤트가 발생합니다. 예를 들어, 다음 코드는 작업 실행 시 툴바의 가시성을 토글하는 작업의 이벤트 핸들러를 설명합니다.

```
procedure TForm1.Action1Execute(Sender:TObject);
begin
    { Toggle Toolbar1's visibility }
    Toolbar1.Visible := not Toolbar1.Visible;
end;
```

세 가지 다른 레벨인 작업, 작업 목록, 애플리케이션 중 하나에서 응답하는 이벤트 핸들러를 제공할 수 있습니다. 이 내용은 이미 정의된 표준 작업이 아닌 새로운 일반 작업을 사용할 경우에만 해당합니다. 표준 작업은 이러한 이벤트가 일어날 때 실행되는 기본 제공 동작을 가지고 있으므로 표준 작업을 사용할 경우 이 내용에 대해 염려하지 않아도 됩니다.

이벤트 핸들러가 이벤트에 대해 응답하는 순서는 다음과 같습니다.

- 작업 목록
- 애플리케이션
- 작업

사용자가 클라이언트 컨트롤을 클릭했을 때, Kylix는 작업 목록이나 애플리케이션이 작업을 처리하지 않는 경우 맨처음에 작업 목록, 그 다음에 애플리케이션 객체, 그 다음에 해당 작업으로 이어지는 작업의 *Execute* 메소드를 호출합니다. 이에 대해 더 자세히 설명하기 위해서 Kylix는 사용자 작업에 응답하는 방법을 찾을 때 디스패칭 순서를 따릅니다.

- 1 작업 목록에 *OnExecute* 이벤트 핸들러를 제공하고 이 이벤트 핸들러가 작업을 처리하면 애플리케이션에서 진행됩니다.

작업 목록의 이벤트 핸들러는 기본값으로 *False*를 반환하는 *Handled*라는 매개변수를 가집니다. 핸들러가 할당되어 이벤트를 처리하면 *True*가 반환되고 처리 순서가 여기서 끝납니다. 예를 들면, 다음과 같습니다.

```
procedure TForm1.ActionList1ExecuteAction(Action:TBasicAction; var Handled:Boolean);
begin
    Handled := True;
end;
```

작업 목록 이벤트 핸들러에서 *Handled*를 *True*로 설정하지 않으면 처리 과정이 진행됩니다.

- 2 작업 목록의 *OnExecute* 이벤트 핸들러를 작성하지 않은 경우 또는 이벤트 핸들러가 작업을 처리하지 않는 경우, 애플리케이션의 *OnActionExecute* 이벤트 핸들러가 실행됩니다. 이벤트 핸들러가 작업을 처리하면 애플리케이션에서 진행됩니다.

애플리케이션의 작업 목록이 이벤트를 처리하지 못하는 경우 전역 *Application* 객체는 *OnActionExecute* 이벤트를 받습니다. 작업 목록의 *OnExecute* 이벤트 핸들러와 마찬가지로 *OnActionExecute* 핸들러는 기본값으로 *False*를 반환하는 *Handled*라는 매개변수를 가집니다. 이벤트 핸들러가 할당되어 이벤트를 처리하면 *True*가 반환되고 처리 순서가 여기서 끝납니다. 예를 들면, 다음과 같습니다.

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled:Boolean);
begin
    { Prevent execution of all actions in Application }
    Handled := True;
end;
```

- 3 애플리케이션의 *OnExecute* 이벤트 핸들러가 작업을 처리하지 않으면 작업의 *OnExecute* 이벤트 핸들러가 실행됩니다.

기본 제공 작업을 사용하거나, 편집 컨트롤과 같은 특정 대상 클래스에서 작동하는 방법을 인지하는 사용자 고유의 작업을 만들 수 있습니다. 이벤트 핸들러를 어떤 레벨에서도 찾을 수 없으면 애플리케이션은 작업을 실행할 대상을 찾고자 합니다. 애플리케이션이 작업을 실행할 대상을 찾으면 작업이 실행됩니다. 애플리케이션에서 이미 정의된 작업 클래스에 응답할 수 있는 대상을 찾는 방법에 대한 자세한 내용은 6-17 페이지의 "작업이 해당 대상을 찾는 방법"을 참조하십시오.

## 작업이 해당 대상을 찾는 방법

6-15 페이지의 "작업 실행 시 발생하는 이벤트"에서는 작업 실행 시 일어나는 실행 주기에 대해 설명합니다. 작업 목록, 애플리케이션 또는 작업 레벨에서 작업에 응답할 이벤트 핸들러가 할당되어 있지 않으면 애플리케이션은 작업이 적용될 대상 객체를 식별하고자 합니다.

애플리케이션은 다음 순서로 대상을 찾습니다.

- 1 활성화된 컨트롤: 애플리케이션은 먼저 활성화된 컨트롤을 잠재적 대상으로 찾습니다.
- 2 활성화된 폼: 애플리케이션이 활성화된 컨트롤을 찾지 못하는 경우나 활성화된 컨트롤이 대상으로 작용할 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 대상에서 연결된 컴포넌트로 연장할 수 있습니다. 예를 들어, data-aware 컨트롤은 연결된 데이터셋 컴포넌트로 연장됩니다. 또한, 일부 이미 정의된 작업에서는 File Open 대화 상자 같은 대상을 사용하지 않습니다.

## 작업 업데이트

애플리케이션이 유휴 상태일 때 *OnUpdate* 이벤트가 표시되는 컨트롤 또는 메뉴 항목에 연결되어 있는 모든 작업에 대해 발생합니다. 이를 통해 애플리케이션에서 설정, 설정 해제, 선택 또는 선택 해제 등을 위한 중앙 집중화된 코드를 실행할 수 있습니다. 예를 들어, 다음 코드는 툴바가 표시될 때 "선택"되어 있는 작업에 대한 *OnUpdate* 이벤트 핸들러를 설명합니다.

```
procedure TForm1.Action1Update(Sender:TObject);
begin
    { Indicate whether ToolBar1 is currently visible }
    (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

**경고** 시간을 많이 소요하는 코드를 *OnUpdate* 이벤트 핸들러에 추가하지 마십시오. 이것은 애플리케이션이 유휴 상태일 때마다 실행됩니다. 이벤트 핸들러가 시간을 너무 많이 소요되면 전체 애플리케이션의 성능에 부정적인 영향을 끼칩니다.

## 이미 정의된 작업 클래스

Action List 편집기를 사용하여 작업을 자동으로 수행하는 이미 정의된 작업 클래스를 사용할 수 있습니다. 이미 정의된 작업은 다음과 같은 범주로 나뉘어집니다.

표 6.1 작업 범주 ( 계속 )

| 범주        | 설명                                                                                                                                                                                                                                                                                                |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 표준 편집 작업  | 편집 컨트롤 대상에서 사용합니다. <i>TEditAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 메소드를 오버라이드하여 클립보드를 통해 복사, 잘라내기 및 붙여넣기 작업을 실행하는 자손 클래스의 기본 클래스입니다.                                                                                                                                                             |
| 표준 창 작업   | MDI 애플리케이션의 대상인 폼에서 사용합니다. <i>TWindowAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 메소드를 오버라이드하여 MDI 자식 폼의 배열, 계단식 배열, 단기, 바둑판식 배열 및 최소화 작업을 수행하는 자손 클래스의 기본 클래스입니다.                                                                                                                                    |
| 표준 도움말 작업 | 모든 대상에서 사용합니다. <i>THelpAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 메소드를 오버라이드하여 도움말 시스템에 명령을 전달하는 자손 클래스의 기본 클래스입니다.                                                                                                                                                                                 |
| 데이터셋 작업   | 데이터셋 컴포넌트 대상에서 사용합니다. <i>TDataSetAction</i> 은 각 클래스가 <i>ExecuteTarget</i> 및 <i>UpdateTarget</i> 메소드를 오버라이드하여 대상의 탐색과 편집 작업을 구현하는 자손 클래스의 기본 클래스입니다.<br><i>TDataSetAction</i> 은 작업이 해당 데이터셋에서 수행되도록 하는 <i>DataSource</i> 속성을 제공합니다. <i>DataSource</i> 가 nil인 경우 현재 포커스가 맞춰진 data-aware 컨트롤을 사용합니다. |
| 대화 상자 작업  | 대화 상자 컴포넌트에서 사용합니다. <i>TDialogAction</i> 은 실행 시 대화 상자를 표시하는 작업의 일반 동작을 구현합니다. 각 자손 클래스는 특정 대화 상자를 나타냅니다.                                                                                                                                                                                          |
| 파일 작업     | File Open 또는 File Exit 같은 파일 작업에서 사용합니다.                                                                                                                                                                                                                                                          |
| 검색 작업     | 검색 옵션에서 사용합니다. <i>TSearchAction</i> 은 사용자가 편집 컨트롤 검색을 위해 검색 문자열을 입력할 수 있는 모달리스 대화 상자를 표시하는 작업의 일반 동작을 구현합니다.                                                                                                                                                                                      |

모든 작업 객체는 온라인 참조 도움말의 작업 객체 이름에 기술되어 있습니다. 작업 객체가 작동하는 방법에 대한 자세한 내용은 도움말을 참조하십시오.

## 작업 컴포넌트 작성

사용자가 직접 이미 정의된 작업 클래스를 만들 수도 있습니다. 사용자가 직접 작업 클래스를 작성할 때 객체의 특정 대상 클래스에서 실행하기 위한 기능을 기본 제공할 수 있습니다. 그런 다음 이미 정의된 작업 클래스를 사용하는 것과 동일한 방법으로 사용자 지정 작업을 사용할 수 있습니다. 즉, 작업이 인식하여 대상 클래스에 자신을 적용할 수 있고 사용자가 클라이언트 컨트롤에 작업을 할당할 수 있을 때 해당 작업은 이벤트 핸들러를 작성할 필요 없이 대상에서 동작합니다.

컴포넌트 작성자는 QStdActns 및 DBActns 유닛의 클래스를 그 고유의 작업 클래스를 파생하여 특정 컨트롤이나 컴포넌트에 관한 동작을 구현하기 위한 예제로 사용할 수 있습니다. 이러한 특수화된 작업 (*TEditAction*, *TWindowAction* 및 기타 등등)의 기본 클래스는 일반적으로 *HandlesTarget*, *UpdateTarget*, 및 다른 메소드를 오버라이드

하여 객체의 특정 클래스에 대한 작업의 대상을 제한합니다. 자손 클래스는 일반적으로 *ExecuteTarget*을 오버라이드하여 특화된 작업을 수행합니다. 이러한 메소드는 다음과 같습니다.

| 메소드                  | 설명                                                                                                                                                                                                                     |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>HandlesTarget</i> | 사용자가 작업에 연결된 툴바 또는 메뉴 항목 같은 객체를 호출할 때 자동으로 호출됩니다. <i>HandlesTarget</i> 메소드를 사용하여 "대상(target)"으로서의 <i>Target</i> 매개변수에 의해 지정된 객체에서 이번에 실행하는 데 적절한지 여부를 작업 객체에 나타낼 수 있습니다. 자세한 내용은 6-17 페이지의 "작업이 해당 대상을 찾는 방법"을 참조하십시오. |
| <i>UpdateTarget</i>  | 작업이 현재 상황에 따라 자체적으로 업데이트할 수 있도록 애플리케이션이 유희 상태일 때 자동으로 호출됩니다. <i>OnUpdateAction</i> 대신 사용합니다. 자세한 내용은 6-17 페이지의 "작업 업데이트"를 참조하십시오.                                                                                      |
| <i>ExecuteTarget</i> | <i>OnExecute</i> 대신에 사용자 작업에 대한 응답으로 작업이 실행될 때 자동으로 호출됩니다(예를 들어, 사용자가 현재 작업에 연결된 메뉴 항목을 선택하거나 툴 버튼을 누를 때). 자세한 내용은 6-15 페이지의 "작업 실행 시 발생하는 이벤트"를 참조하십시오.                                                               |

## 작업 등록

사용자 고유의 작업을 작성할 때 작업을 등록하여 Action List 편집기에 작업이 나타나도록 할 수 있습니다. Actnlist 유닛의 전역 루틴을 사용하면 작업을 등록 및 등록 해제할 수 있습니다.

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);
```

```
procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

*RegisterActions*을 호출하면 등록한 작업이 애플리케이션에서 사용할 수 있도록 Action List 편집기에 나타납니다. 작업을 구성하기 위한 범주 이름 뿐만 아니라 기본 속성 값을 제공할 수 있게 해주는 *Resource* 매개변수도 제공할 수 있습니다.

예를 들어, 다음 코드는 IDE에 표준 작업을 등록합니다.

```
{ Standard action registration }
RegisterActions('', [TAction], nil);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

*UnRegisterActions*을 호출하면 해당 작업이 Action List 편집기에 더 이상 나타나지 않습니다.

## 메뉴 생성 및 관리

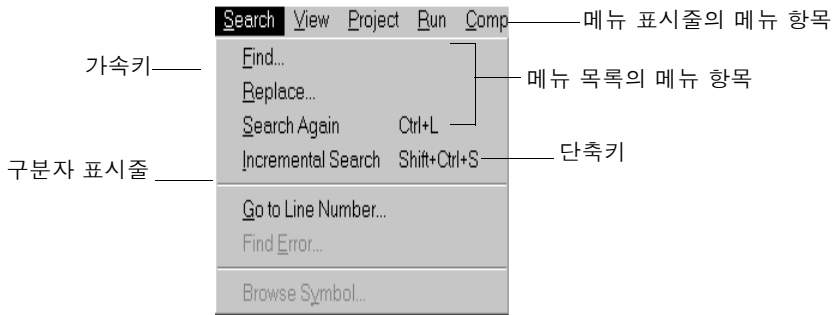
메뉴를 통해 논리적으로 그룹화된 명령을 쉽게 실행할 수 있습니다. 메뉴 디자이너를 사용하면 폼에 (이미 디자인된 또는 사용자 지정에 맞게) 메뉴를 쉽게 추가할 수 있습니다. 폼에 메뉴 컴포넌트를 추가하고 메뉴 디자이너를 연 다음 메뉴 디자이너 창에 메뉴 항목을 직접 입력합니다. 메뉴 항목을 추가하거나 삭제할 수 있고 메뉴 항목을 드래그 앤 드롭하여 디자인 타임에 메뉴 항목을 재정렬할 수 있습니다.

런타임에 나타날 모양과 동일하게 디자인이 폼에서 즉시 보이므로 결과를 보기 위해 프로그램을 실행할 필요가 없습니다. 또한 코드에서 사용자에게 자세한 내용이나 옵션을 제공하기 위해 런타임에 메뉴를 변경할 수 있습니다.

이 장에서는 메뉴 디자이너를 사용하여 메뉴 표시줄과 팝업(로컬) 메뉴를 디자인하는 방법에 대해 설명합니다. 또한 디자인 타임 및 런타임에 메뉴를 사용하는 방법을 설명합니다.

- 메뉴 디자이너 열기
- 메뉴 작성
- Object Inspector에서 메뉴 항목 편집
- 메뉴 디자이너 컨텍스트 메뉴 사용
- 메뉴 템플릿 사용
- 메뉴를 템플릿으로 저장
- 메뉴 항목에 이미지 추가

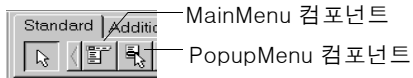
**그림 6.2** 메뉴 용어



## 메뉴 디자인

메뉴 디자이너를 사용하여 메뉴를 디자인합니다. 메뉴 디자이너를 사용하여 시작하기 전에 먼저 *TMainMenu* 또는 *TPopupMenu* 컴포넌트를 폼에 추가합니다. 두 메뉴 컴포넌트는 모두 컴포넌트 팔레트의 Standard 페이지에 있습니다.



**그림 6.3** MainMenu 컴포넌트 및 PopupMenu 컴포넌트

MainMenu 컴포넌트는 폼의 제목 표시줄에 붙여 지는 메뉴를 만듭니다. PopupMenu 컴포넌트는 폼에서 마우스 오른쪽 버튼을 클릭하면 나타나는 메뉴를 만듭니다. 팝업 메뉴에는 메뉴 표시줄이 없습니다.

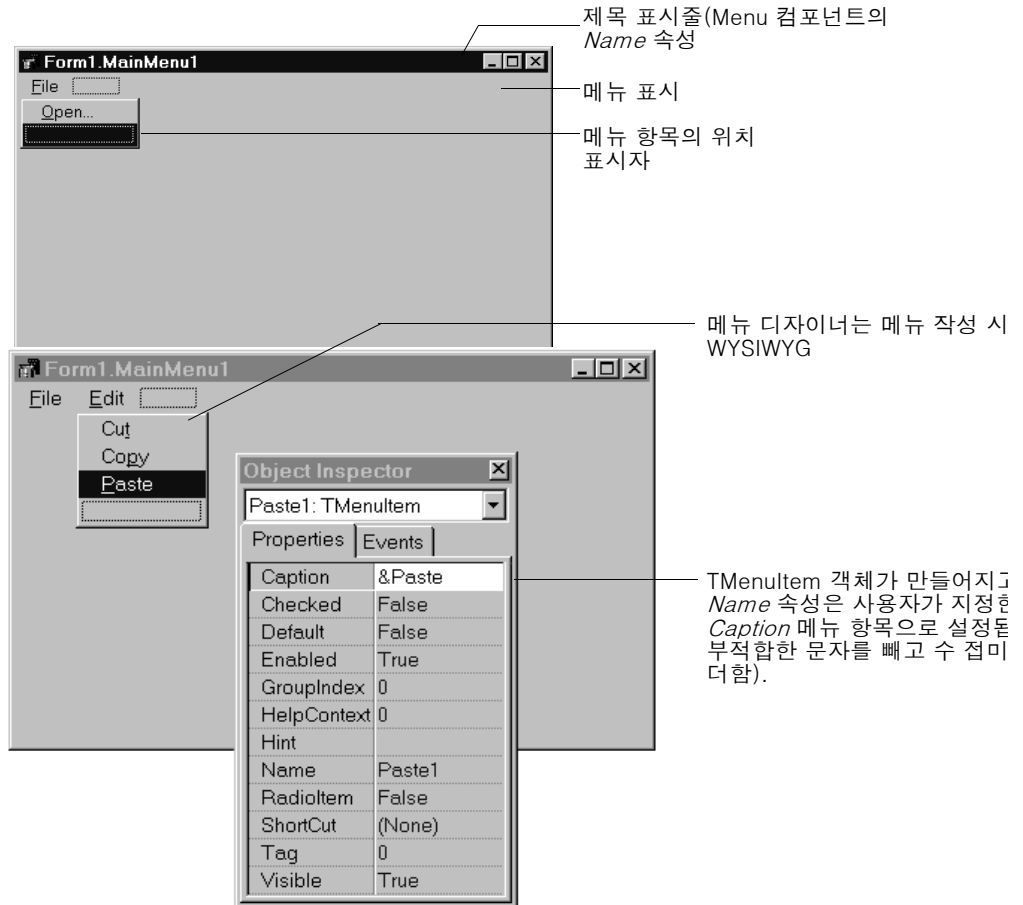
메뉴 디자인어를 열려면 폼에서 메뉴 컴포넌트를 선택한 다음,

- 메뉴 컴포넌트를 더블 클릭합니다.  
또는,
- Object Inspector의 Properties 페이지에서 *Items* 속성을 선택하고 값 열의 [Menu]를 더블 클릭하거나 생략(...) 버튼을 클릭합니다.

메뉴 디자인어는 디자이너에서 선택한 첫 번째(비어 있는) 메뉴 항목과 Object Inspector에서 선택한 *Caption* 속성이 함께 표시됩니다.

**그림 6.4** 팝업 메뉴의 메뉴 디자인어

그림 6.5 메인 메뉴의 메뉴 디자이너



## 메뉴 작성

애플리케이션에 포함할 모든 메뉴들에 대해서 메뉴 컴포넌트를 폼이나 폼들에 추가합니다. 완전히 처음부터 각 메뉴 구조를 만들거나 이미 디자인된 메뉴 템플릿 중 하나에서 시작할 수 있습니다.

이 단원에서는 디자인 타임에 메뉴를 만들기 위한 기본 사항을 설명합니다. 메뉴 템플릿에 대한 자세한 내용은 페이지 6-30의 "메뉴 템플릿 사용"을 참조하십시오.

## 메뉴 이름 지정

모든 컴포넌트에서 폼에 메뉴 컴포넌트를 추가할 때 Kylix에서는 메뉴 항목에 *MainMenu1*과 같은 기본 이름을 제공합니다. 사용자는 오브젝트 파스칼 이름 지정 규칙에 따르는 좀 더 의미 있는 메뉴 이름을 지정할 수도 있습니다.

Kylix에서 폼의 타입 선언에 메뉴 이름을 추가하면 메뉴 이름이 컴포넌트 목록에 나타납니다.

## 메뉴 항목 이름 지정

메뉴 컴포넌트와 대조적으로 폼에 메뉴 항목을 추가할 때는 명시적으로 메뉴 항목의 이름을 지정해야 합니다. 다음 두 가지 방법 중 하나로 메뉴 항목 이름을 지정할 수 있습니다.

- *Name* 속성에 값을 직접 입력합니다.
- *Caption* 속성에 값을 먼저 입력하고 Kylix가 캡션의 *Name* 속성을 파생합니다.

예를 들어, 메뉴 항목에 *File*의 *Caption* 속성 값을 제공하는 경우 Kylix는 해당 메뉴 항목에 *File1*의 *Name* 속성을 할당합니다. *Caption* 속성을 입력하기 전에 *Name* 속성을 채우면 Kylix는 값이 입력될 때까지 *Caption* 속성을 비워 둡니다.

### 참고

*Caption* 속성에 오브젝트 파스칼 식별자에 유효하지 않은 문자를 입력하면 Kylix는 그에 맞게 *Name* 속성을 수정합니다. 예를 들어, 사용자가 캡션에서 숫자로 시작할 것 원하면 Kylix는 *Name* 속성을 파생한 문자에서 숫자가 선행하도록 합니다.

다음 표에서는 표시된 모든 메뉴 항목이 동일한 메뉴 표시줄에 나타난다는 가정하에 이에 관한 예제를 보여 줍니다.

**표 6.2** 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 비정규 문자를 제거하고 선행 문자 및 번호 추가      |
| - (하이픈)         | N2     | 비정규 문자가 있는 캡션이 두 번째 나타날 때 순차적으로 정렬 |

메뉴 컴포넌트에서 Kylix가 폼 타입 선언에 메뉴 항목 이름을 추가하면 컴포넌트 목록에 추가된 메뉴 항목 이름이 나타납니다.

## 메뉴 항목 추가, 삽입 및 삭제

다음 절차는 메뉴 구조를 만드는 것과 관련된 기본 작업 수행 방법에 대해 설명합니다. 각 절차는 메뉴 디자이너 창이 열려 있다고 가정합니다.

다음과 같은 방법으로 디자인 타임에 메뉴 항목을 추가합니다.

- 1 메뉴 항목을 만들 위치를 선택합니다.

메뉴 디자이너를 여는 경우, 메뉴 표시줄의 첫 번째 위치가 이미 선택되어 있습니다.

- 2 캡션을 입력하기 시작합니다. 또는 Object Inspector에 커서를 두고 값을 입력하여 먼저 *Name* 속성을 입력합니다. 이러한 경우 *Caption* 속성을 다시 선택하고 값을 입력해야 합니다.

- 3 *Enter* 키를 누릅니다.

그러면 메뉴 항목의 다음 위치 표시자가 선택됩니다.

먼저 *Caption* 속성을 입력한 경우 화살표 키를 사용하여 방금 입력한 메뉴 항목으로 돌아갑니다. Kylix는 입력한 캡션 값에 기반하여 *Name* 속성을 채웁니다. (6-23 페이지의 "메뉴 항목 이름 지정" 참조)

- 4 만들려는 새 항목마다 *Name* 및 *Caption* 속성에 값을 계속 입력하거나 *Esc*를 눌러 메뉴 표시줄로 돌아갑니다.

화살표 키를 사용하여 메뉴 표시줄에서 메뉴로 이동한 다음 목록의 항목과 항목 사이를 이동하고 *Enter* 를 눌러 작업을 완료합니다. 메뉴 표시줄로 돌아가려면 *Esc*를 누릅니다.

다음과 같은 방법으로 비어 있는 새 메뉴 항목을 삽입합니다.

- 1 메뉴 항목에 커서를 둡니다.

- 2 *Ins*를 누릅니다.

메뉴 표시줄의 선택한 항목 왼쪽 및 메뉴 목록의 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 목록에서 마지막으로 입력된 항목 아래 또는 메뉴 표시줄의 마지막 항목 옆에 나타나는 기본 위치 표시자를 삭제할 수 없습니다. 이 위치 표시자는 런타임에 메뉴에 나타나지 않습니다.

## 구분자 표시줄 추가

- 구분자 표시줄은 메뉴 항목 사이에 선을 삽입합니다. 구분자 표시줄을 사용하여 메뉴 목록 내에서 그룹화를 표시하거나 메뉴 목록에서 시각적으로 구분할 수 있습니다.

메뉴 항목을 구분자 표시줄로 만들려면 캡션에 하이픈(-)을 입력합니다.

## 가속키 및 단축키 지정

가속키를 사용하여 *Alt*+ 해당 문자를 눌러 키보드에서 메뉴 명령에 액세스할 수 있으며 코드에서 선행 앰퍼샌드에 의해 표시됩니다. 앰퍼샌드 다음에 오는 문자는 메뉴에서 밑줄이 그어져 표시됩니다.

Kylix는 중복된 가속키를 자동으로 확인하고 런타임에 조정합니다. 이렇게 함으로써 런타임에 동적으로 작성한 메뉴에는 중복된 가속키가 들어 있지 않고 모든 메뉴 항목에는 하나의 가속키가 있습니다.

다음과 같은 방법으로 가속키를 지정합니다.

- 해당 문자 앞에 앰퍼샌드를 추가합니다.

예를 들어, *S*를 갖는 *Save* 메뉴 명령을 가속키로 추가하려면 *&Save*를 입력합니다.

단축키를 사용하여 메뉴를 직접 사용하지 않고 단축키 조합으로 입력하여 작업을 수행할 수 있습니다.

다음과 같은 방법으로 단축키를 지정합니다.

- Object Inspector를 사용하여 *ShortCut* 속성에 값을 입력하거나 드롭다운 목록에서 키 조합을 선택합니다.

이 목록은 사용자가 입력할 수 있는 키 조합들입니다.

추가한 단축키는 메뉴 항목 캡션 옆에 나타납니다.

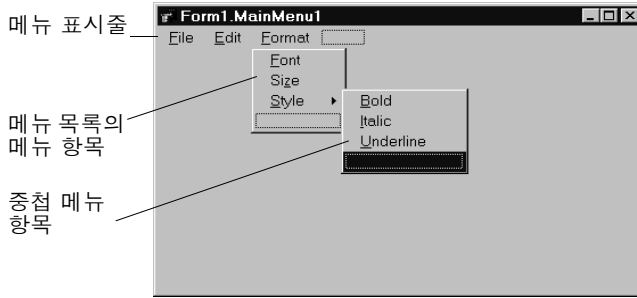
**주의** 가속키와 달리 단축키는 중복에 대해 자동으로 검사하지 않습니다. 사용자 자신이 고유성을 확인해야 합니다.

## 하위 메뉴 생성

많은 애플리케이션에는 메뉴 항목 옆에 드롭다운 목록이 있어 해당 메뉴 항목과 관련된 추가 명령을 제공합니다. 이러한 목록은 메뉴 항목의 오른쪽에 화살표 키가 나타납니다. Kylix는 메뉴에 만드는 다양한 레벨의 하위 메뉴를 지원합니다.

이런 방법으로 메뉴 구조를 구성하면 화면의 세로 공간을 절약할 수 있습니다. 그러나 최적화된 디자인을 위해서 인터페이스 디자인에서 메뉴 레벨을 둘이나 세 개 이상 사용하지 않는 것이 좋습니다. 팝업 메뉴에서 하위 메뉴가 있는 경우 하위 메뉴를 하나만 사용하기를 원할 수 있습니다.

**그림 6.6** 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 Submenu**를 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 3단계와 4단계를 반복합니다.
- 6 **Esc**를 눌러 이전 메뉴 레벨로 돌아갑니다.

### 기존 메뉴를 밑으로 내려 하위 메뉴 생성

목록의 메뉴 항목 사이에 메뉴 표시줄 또는 메뉴 템플릿의 메뉴 항목을 삽입하여 하위 메뉴를 만들 수 있습니다. 기존 메뉴 구조로 메뉴를 이동하면 메뉴와 연결된 항목 모두가 해당 메뉴와 함께 이동하여 완전히 그대로 유지된 하위 메뉴를 만듭니다. 이는 하위 메뉴에도 적용됩니다. 기존 하위 메뉴로 메뉴 항목을 이동하면 중첩 레벨이 하나 더 만들어 집니다.

### 메뉴 항목 이동

디자인 타임에 드래그 앤 드롭으로 메뉴 항목을 이동할 수 있습니다. 메뉴 표시줄에서 메뉴 목록의 다른 위치 또는 완전히 다른 메뉴로 메뉴 항목을 이동할 수 있습니다.

메뉴 이동에 대한 유일한 예외는 계층 구조와 관련된 것입니다. 메뉴 표시줄에서 메뉴 항목을 자신의 메뉴로 내릴 수 없고 메뉴 항목을 자신의 하위 메뉴로 이동할 수 없습니다. 그러나 항목의 원래 위치와 상관 없이 메뉴 항목을 다른 메뉴로 이동할 수 있습니다.

메뉴 항목을 끌어 놓는 동안 커서 모양이 바뀌면서 새 위치에 메뉴 항목을 놓을 수 있는지 여부를 나타냅니다. 메뉴 항목을 이동하면 아래의 모든 항목도 함께 이동합니다.

다음과 같은 방법으로 메뉴 표시줄에서 메뉴 항목을 이동합니다.

- 1 끄는 동안 커서의 화살표 끝이 새 위치를 가리킬 때까지 메뉴 표시줄에서 메뉴 항목을 끕니다.
- 2 마우스 버튼을 놓아 메뉴 항목을 새 위치에 놓습니다.

다음과 같은 방법으로 메뉴 목록으로 메뉴 항목을 이동합니다.

- 1 드래그 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 목록으로 끌어 놓은 다음 마우스 버튼을 놓아 메뉴 항목을 새 위치에 놓습니다.

## 메뉴 항목에 이미지 추가

이미지는 툴바 이미지와 마찬가지로 메뉴 항목 동작에 glyph 및 이미지를 연결함으로써 메뉴를 찾는 데 도움을 줍니다. 메뉴 항목에 단일 비트맵을 추가하거나 애플리케이션의 이미지를 이미지 목록으로 구성한 다음 이미지 목록의 메뉴에 추가할 수 있습니다. 애플리케이션에서 동일한 크기의 여러 비트맵을 사용하는 경우 이러한 비트맵을 이미지 목록에 넣는 것이 유용합니다.

메뉴 또는 메뉴 항목에 단일 이미지를 추가하려면 이미지의 *Bitmap* 속성이 비트맵 이름을 참조하도록 설정하여 메뉴 또는 메뉴 항목에서 사용합니다.

다음과 같은 방법으로 이미지 목록을 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* 편집기를 엽니다.
- 4 Add를 눌러 메뉴에서 사용할 비트맵 또는 비트맵 그룹을 선택합니다. OK를 클릭합니다.
- 5 *TMainMenu* 또는 *TPopupMenu* 객체의 *Images* 속성을 방금 만든 *ImageList*로 설정합니다.
- 6 이전에 설명한대로 메뉴 항목과 하위 메뉴 항목을 만듭니다.
- 7 Object Inspector에서 이미지를 갖고자하는 메뉴 항목을 선택하고 *ImageIndex* 속성을 *ImageList*의 해당 이미지 번호로 설정합니다(*ImageIndex*의 기본값은 -1이며 이 값은 이미지를 나타내지 않음).

**참고** 메뉴에서 적절하게 표시되도록 16 x 16 픽셀의 이미지를 사용합니다. 다른 크기의 메뉴 이미지를 사용할 수는 있지만 16 x 16 픽셀보다 크거나 작은 이미지를 사용하면 정렬 및 일관성 문제가 발생할 수 있습니다.

## 메뉴 보기

디자인 타임에 프로그램 코드를 먼저 실행하지 않고도 폼에서 메뉴를 볼 수 있습니다. 디자인 타임에 폼에서 팝업 메뉴 컴포넌트는 보이지만 팝업 메뉴 자체는 보이지 않습니다. 메뉴 디자이너를 사용하면 디자인 타임에 팝업 메뉴를 볼 수 있습니다.

다음과 같은 방법으로 메뉴를 봅니다.

- 1 폼이 보이는 경우 폼을 클릭하거나 View 메뉴에서 보고자 하는 메뉴의 폼을 선택합니다.
- 2 폼에 메뉴가 하나 이상 있으면 폼의 *Menu* 속성 드롭다운 목록에서 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 설명합니다(예를 들면, 메뉴 디자인너를 사용하여 *Name* 및 *Caption* 속성 설정).

또한 이 단원에서는 폼에서 선택한 컴포넌트의 속성을 설정하는 것처럼 Object Inspector에서 직접 *ShortCut* 속성과 같은 메뉴 항목 속성을 설정하는 방법에 대해 설명합니다.

메뉴 디자인너를 사용하여 메뉴 항목을 편집할 때 메뉴 항목의 속성은 Object Inspector에 표시됩니다. Object Inspector로 포커스를 전환한 다음 메뉴 항목 속성을 계속 편집할 수 있습니다. 또는 메뉴 디자인너를 열지 않고 Object Inspector의 컴포넌트 목록에서 메뉴 항목을 선택한 다음 메뉴 항목의 속성을 편집할 수 있습니다.

다음과 같은 방법으로 메뉴 디자인너 창을 닫고 메뉴 항목을 계속 편집합니다.

- 1 Object Inspector의 속성 페이지를 눌러 메뉴 디자인너 창에서 Object Inspector로 포커스를 전환합니다.
- 2 일반적인 방법으로 메뉴 디자인너를 닫습니다.

선택한 메뉴 항목의 속성을 계속 편집할 수 있는 Object Inspector에 포커스가 그대로 유지됩니다. 다른 메뉴 항목을 편집하려면 컴포넌트 목록에서 메뉴 항목을 선택합니다.

## 메뉴 디자인너 컨텍스트 메뉴 사용

---

메뉴 디자인너 컨텍스트 메뉴를 통해 가장 많이 사용하는 메뉴 디자인너 명령 및 메뉴 템플릿 옵션에 빠르게 액세스할 수 있습니다. (메뉴 템플릿에 대한 자세한 내용은 6-30 페이지의 "메뉴 템플릿 사용"을 참조하십시오.)

컨텍스트 메뉴를 표시하려면 메뉴 디자인너 창을 마우스 오른쪽 버튼으로 클릭하거나 커서가 메뉴 디자인너 창에 있을 때 *Alt+F10*을 누릅니다.



## 컨텍스트 메뉴의 명령

다음 표는 메뉴 디자이너 컨텍스트 메뉴의 명령에 대해 요약하여 설명한 것입니다.

**표 6.3** 메뉴 디자이너 컨텍스트 메뉴 명령

| 메뉴 명령                | 동작                                                                     |
|----------------------|------------------------------------------------------------------------|
| 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.                          |
| Select Menu          | 현재 폼에서 메뉴 목록이 열립니다. 메뉴 이름을 더블 클릭하면 메뉴 디자이너 창이 열립니다.                    |
| Save As Template     | 다음에 재사용할 메뉴를 저장할 수 있는 Save Template 대화 상자가 열립니다.                       |
| Insert From Template | 재사용할 템플릿을 선택할 수 있는 Insert Template 대화 상자가 열립니다.                        |
| Delete Templates     | 기존 템플릿을 삭제할 수 있는 Delete Templates 대화 상자가 열립니다.                         |
| Insert From Resource | 현재 폼에서 열 .mnu 파일을 선택할 수 있는 Insert Menu from Resource file 대화 상자가 열립니다. |

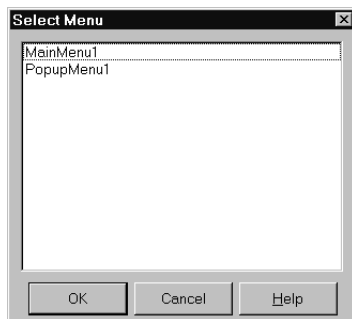
## 디자인 타임 시 메뉴 간의 전환

폼에 대한 여러 메뉴를 디자인하는 경우, 메뉴 디자이너 컨텍스트 메뉴 또는 Object Inspector를 사용하여 메뉴 간에 쉽게 선택하고 이동할 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 대화 상자가 나타납니다.

**그림 6.7** Select Menu 대화 상자



이 대화 상자는 메뉴 디자이너에서 현재 메뉴가 열려 있는 폼에 연결된 모든 메뉴 목록을 나열합니다.

- 2 Select Menu 대화 상자의 목록에서 보거나 편집할 메뉴를 선택합니다.  
다음과 같은 방법으로 Object Inspector를 사용하여 폼의 메뉴 간에 전환합니다.
- 1 선택할 메뉴의 폼에 포커스를 맞춥니다.

- 2 컴포넌트 목록에서 편집할 메뉴를 선택합니다.
- 3 Object Inspector의 Properties 페이지에서 이 메뉴에 대한 *Items* 속성을 선택한 다음 생략 버튼을 클릭하거나 Menu를 더블 클릭합니다.

## 메뉴 템플릿 사용

Kylix는 자주 사용하는 명령이 들어 있는 이미 디자인된 메뉴 또는 메뉴 템플릿을 여러 개 제공합니다. 메뉴를 수정하지 않고 애플리케이션에서 이러한 메뉴를 사용할 수 있고 (코드 작성은 제외), 이러한 메뉴를 기본 메뉴로 사용하여 사용자 자신이 원래 디자인한 메뉴에서 작업했던 대로 메뉴를 사용자 지정할 수 있습니다. 메뉴 템플릿에는 이벤트 핸들러 코드가 들어 있지 않습니다.

Kylix와 함께 제공된 메뉴 템플릿은 기본 설치에서 .borland 디렉토리의 delphi60dmt 에 저장되어 있습니다.

메뉴 디자이너를 사용하여 디자인한 메뉴를 템플릿으로 저장할 수도 있습니다. 메뉴를 템플릿으로 저장한 후 이미 디자인된 메뉴에서 작업했던 대로 사용할 수 있습니다. 특정 메뉴 템플릿을 더 이상 원하지 않을 경우 목록에서 해당 메뉴 템플릿을 삭제할 수 있습니다.

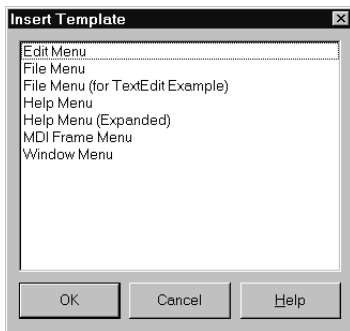
다음과 같은 방법으로 애플리케이션에 메뉴 템플릿을 추가합니다.

- 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 대화 상자가 열리면서 사용 가능한 메뉴 템플릿 목록을 표시합니다.

**그림 6.8** 메뉴에 대한 예제 Insert Template 대화 상자



- 2 삽입할 메뉴 템플릿을 선택한 다음 *Enter* 키를 누르거나 OK를 선택합니다.

그러면 커서가 위치한 품에 메뉴가 삽입됩니다. 예를 들어, 커서가 목록의 메뉴 항목에 있는 경우 메뉴 템플릿이 선택한 항목 위에 삽입됩니다. 커서가 메뉴 표시줄에 있으면 메뉴 템플릿이 커서의 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 대화 상자가 열리면서 사용 가능한 템플릿 목록을 나열합니다.
- 2 삭제할 메뉴 템플릿을 선택한 다음 *Del* 키를 누릅니다.  
Kylix는 템플릿 목록과 하드 디스크에서 템플릿을 삭제합니다.

## 메뉴를 템플릿으로 저장

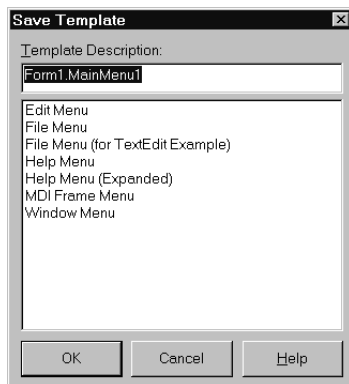
디자인한 메뉴를 다시 사용할 수 있도록 템플릿으로 저장할 수 있습니다. 메뉴 템플릿을 사용하여 애플리케이션에 일관된 모양을 제공하거나 이러한 메뉴 템플릿을 기본 메뉴 템플릿으로 사용하여 추가로 사용자 지정할 수 있습니다.

저장한 메뉴 템플릿은 .borland 디렉토리의 delphi60dmt 파일에 저장됩니다.

다음과 같은 방법으로 메뉴를 템플릿으로 저장합니다.

- 1 재사용할 수 있는 메뉴를 디자인합니다.  
이 메뉴에는 원하는 수 만큼의 항목, 명령 및 하위 메뉴가 많이 들어 있을 수 있습니다. 활성화된 메뉴 디자이너 창의 모든 것이 재사용 가능한 메뉴로 저장됩니다.
- 2 메뉴 디자이너에서 마우스 오른쪽 버튼을 클릭한 다음 Save As Template을 선택합니다.  
Save Template 대화 상자가 나타납니다.

**그림 6.9** 메뉴에 대한 Save Template 대화 상자



- 3 Template Description 편집 상자에 이 메뉴에 대한 간단한 설명을 입력한 다음 OK를 선택합니다.  
Save Template 대화 상자가 닫히면서 메뉴 디자인을 저장하고 메뉴 디자이너 창으로 돌아갑니다.

**참고** 입력한 설명은 Save Template, Insert Template 및 Delete Templates 대화 상자에만 표시됩니다. 이 설명은 메뉴의 *Name* 또는 *Caption* 속성과 관련이 없습니다.

## 템플릿 메뉴 항목 및 이벤트 핸들러의 이름 지정 규칙

메뉴를 템플릿으로 저장할 때 Kylix는 모든 메뉴가 메뉴 소유자(폼)의 유효 범위(scope) 내에서 고유 이름을 가져야 하므로 메뉴의 *Name* 속성을 저장하지 않습니다. 그러나 메뉴를 사용하여 메뉴를 템플릿으로 새 폼에 삽입할 때 Kylix는 메뉴의 새 이름 및 모든 항목을 생성합니다.

예를 들어, File 메뉴를 템플릿으로 저장한다고 가정합니다. 원래의 메뉴에서 메뉴 이름을 *MyFile*이라고 지정합니다. 이 메뉴를 템플릿으로 새 메뉴에 삽입할 경우 Kylix는 해당 메뉴의 이름을 *File1*이라고 지정합니다. *File1*이라는 이름으로 지정된 기존 메뉴 항목을 갖는 메뉴에 이 메뉴를 삽입하면 Kylix는 이 메뉴의 이름을 *File2*라고 지정합니다.

또한 Kylix는 템플릿으로 저장된 메뉴에 연결된 *OnClick* 이벤트 핸들러를 저장하지 않습니다. 코드를 새 폼에서 적용할 수 있는지 여부를 테스트할 수 있는 방법이 없기 때문입니다. 메뉴 템플릿 항목에 대해 이벤트 핸들러를 새로 생성할 때 Kylix는 이벤트 핸들러 이름을 생성합니다. 메뉴 템플릿의 항목을 폼의 기존 *OnClick* 이벤트 핸들러와 연결할 수 있습니다.

## 런타임 시 메뉴 항목 처리

애플리케이션이 실행 중일 때 기본 메뉴 구조에 메뉴 항목을 추가하여 자세한 내용이나 옵션을 사용자에게 제공하고자 할 경우가 종종 있습니다. 메뉴 항목의 *Add* 또는 *Insert* 메소드를 사용하여 메뉴 항목을 삽입할 수 있고, 메뉴 항목의 *Visible* 속성을 변경하여 메뉴 항목을 숨기거나 표시할 수 있습니다. *Visible* 속성을 통해 메뉴 항목을 메뉴에 표시할지 여부를 결정합니다. 메뉴를 숨기지 않고 흐리게 표시하려면 *Enabled* 속성을 사용합니다.

메뉴 항목의 *Visible* 및 *Enabled* 속성 사용에 대한 예제는 페이지 7-4의 "메뉴 항목 사용 불가능"을 참조하십시오.

## 툴바 디자인

툴바는 버튼과 다른 컨트롤을 포함하는 일반적으로 폼의 맨 위에 위치하면서 메뉴 표시줄 아래에 있는 가로로 된 하나의 패널입니다. 툴바에 모든 종류의 컨트롤을 놓을 수 있습니다. 버튼은 물론 색상 그리드, 스크롤 막대 및 레이블 등을 놓을 수 있습니다.

다음과 같은 여러 방법으로 폼에 툴바를 추가합니다.

- 폼에 패널(*TPanel*)을 두고 패널에 컨트롤(전형적으로 스피드 버튼)을 추가합니다.
- *TPanel* 대신 툴바 컴포넌트(*TToolBar*)를 사용하여 컨트롤을 추가합니다.  
*TToolBar*는 버튼 및 다른 컨트롤을 행에 배치하고 크기와 위치를 자동으로 조정하여 버튼 및 다른 컨트롤을 관리합니다. 툴바에서 툴 버튼(*TToolButton*) 컨트롤을 사용하는 경우 *TToolBar*를 사용하면 버튼을 기능에 따라 쉽게 그룹화하고 다른 표시 옵션을 제공할 수 있습니다.

툴바 구현 방법은 애플리케이션에 따라 다릅니다. *Panel* 컴포넌트를 사용하면 툴바의 룩앤필을 완전히 제어할 수 있다는 이점이 있습니다.

툴바 컴포넌트를 사용하여 애플리케이션이 일관성 있는 룩앤필을 갖도록 합니다. 이러한 운영 체제 컨트롤이 나중에 변경되면 애플리케이션도 변경됩니다.

다음 단원에서는 다음과 같은 작업을 수행하는 방법에 대해 설명합니다.

- 패널 컴포넌트를 사용하여 툴바 및 해당 스피드 버튼 컨트롤 추가
- Toolbar 컴포넌트를 사용하여 툴바 및 해당 툴 버튼 컨트롤 추가
- 클릭 이벤트에 응답
- 숨겨진 툴바 추가
- 툴바 숨기기 및 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 정렬될 때 창 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 디자인되었습니다. 스피드 버튼에는 일반적으로 캡션은 없고 버튼 기능을 나타내는 *glyph*라는 작은 그래픽만을 포함합니다.

스피드 버튼에는 세 가지 작동 모드가 있습니다. 스피드 버튼은 다음 작동을 수행할 수 있습니다.

- 일반 푸시 버튼처럼 동작
- 눌러서 켜고 끄기
- 라디오 버튼 집합처럼 동작

툴바에서 스피드 버튼을 구현하려면 다음을 수행하십시오.

- 툴바 패널에 스피드 버튼 추가
- 스피드 버튼의 *glyph* 할당
- 스피드 버튼의 초기 상태 설정
- 스피드 버튼 그룹 생성
- 버튼 토크 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 기본 높이는 41이고 스피드 버튼의 기본 높이는 25입니다. 각 버튼의 *Top* 속성을 8로 설정하면 버튼이 수직 중앙에 위치합니다. 기본 그리드 설정에 의해 스피드 버튼이 해당 수직 위치에 맞춰집니다.

## 스피드 버튼의 glyph 할당

각 스피드 버튼에는 버튼이 수행하는 작업을 사용자에게 나타내기 위해 *glyph*라는 그래픽 이미지가 필요합니다. 스피드 버튼에 이미지를 하나만 제공하는 경우 버튼은 해당 이미지를 조작하여 버튼이 눌러져 있는지, 선택되어 있는지 또는 사용 불가능한지 여부를 나타냅니다. 사용자가 원할 경우 각 상태에 대해서 별도의 특정 이미지를 제공할 수 있습니다.

런타임에 다른 glyph를 할당할 수 있지만 일반적으로 디자인 타임에 glyph를 스피드 버튼에 지정합니다.

다음과 같은 방법으로 디자인 타임에 glyph를 스피드 버튼에 할당합니다.

- 1 스피드 버튼을 선택합니다.
- 2 Object Inspector에서 *Glyph* 속성을 선택합니다.
- 3 *Glyph* 옆의 값 열을 더블 클릭하여 Picture Editor를 열고 원하는 비트맵을 선택합니다.

## 스피드 버튼의 초기 상태 설정

사용자는 스피드 버튼의 모양을 통해 버튼의 상태 및 용도에 대해 알 수 있습니다. 스피드 버튼에는 캡션이 없으므로 사용자에게 도움을 줄 수 있는 적절한 시각적 역할이 중요합니다.

표 6.4는 스피드 버튼의 모양을 변경하기 위해 설정할 수 있는 일부 동작을 나열합니다.

표 6.4 스피드 버튼의 모양 설정

| 스피드 버튼의 모양 변경 | 툴바의 속성 설정                                                                 |
|---------------|---------------------------------------------------------------------------|
| 눌려진 상태로 표시    | <i>GroupIndex</i> 속성을 0이 아닌 값으로 설정하고 <i>Down</i> 속성을 <i>True</i> 로 설정합니다. |
| 사용 불가능으로 표시   | <i>Enabled</i> 속성을 <i>False</i> 로 설정합니다.                                  |
| 왼쪽 여백 표시      | <i>Indent</i> 속성값을 0보다 큰 값으로 설정합니다.                                       |

애플리케이션에 기본 드로잉 툴이 있는 경우 애플리케이션 시작 시 툴바의 해당 버튼이 눌러져 있는지 확인합니다. 그렇게 하려면 *GroupIndex* 속성을 0이 아닌 값으로 설정하고 *Down* 속성을 *True*로 설정합니다.

## 스피드 버튼 그룹 생성

경우에 따라 일련의 스피드 버튼은 상호 배타적인 선택의 집합을 나타냅니다. 그러한 경우 버튼을 그룹으로 연결하면 그룹 내의 임의 버튼을 눌렀을 때 그룹 내의 다른 버튼이 팝업됩니다.

임의 수의 스피드 버튼을 그룹으로 연결하려면 동일한 수를 각 스피드 버튼의 *Group Index* 속성에 할당합니다.

이렇게 하는 가장 쉬운 방법은 그룹에서 원하는 버튼을 모두 선택한 다음 선택한 전체 그룹에서 *GroupIndex*를 고유 값으로 설정하는 것입니다.

## 버튼 토글 허용

때때로 그룹 내에 있는 버튼 하나를 클릭하여 다른 버튼에는 영향을 주지 않고 눌러진 상태로 바뀌고 다시 한 번 클릭하면 팝업되게 만들고 싶은 경우가 있습니다. 이를 버튼 토글이라고 합니다. *AllowAllUp*을 사용하여 토글처럼 동작하는 그룹화된 버튼을 만들 수 있습니다. 그룹화된 버튼을 한 번 누르면 눌러지고 다시 누르면 팝업됩니다.

그룹화된 스피드 버튼을 토글로 만들려면 *AllowAllUp* 속성을 *True*로 설정합니다.

그룹 내 임의의 스피드 버튼의 *AllowAllUp* 속성을 *True*로 설정하면 그룹 내 모든 버튼에 대해 동일한 속성 값이 자동으로 설정됩니다. 이렇게 하면 그룹이 한 번에 한 버튼만 눌러지지만 동시에 모든 버튼이 팝업될 수 있게 해 주는, 일반 그룹처럼 동작하는 그룹을 사용할 수 있습니다.

## 툴바 컴포넌트를 사용하여 툴바 추가

툴바 컴포넌트(*TToolBar*)는 패널 컴포넌트가 제공하지 않는 버튼 관리 및 표시 기능을 제공합니다. 다음과 같은 방법으로 툴바 컴포넌트를 사용하여 폼에 툴바를 추가합니다.

1 폼에 툴바 컴포넌트를 추가합니다. 툴바는 폼 맨 위에 자동으로 정렬됩니다.

2 툴바에 툴 버튼 또는 다른 컨트롤을 추가합니다.

툴 버튼은 툴바 컴포넌트에서 작동하도록 디자인되었습니다. 스피드 버튼과 마찬가지로 툴 버튼은 다음과 같은 동작을 수행할 수 있습니다.

- 일반 푸시 버튼처럼 동작
- 눌러서 켜고 끄기
- 라디오 버튼 집합처럼 동작

툴바에서 툴 버튼을 구현하려면 다음을 수행하십시오.

- 툴 버튼 추가
- 툴 버튼에 이미지 할당
- 툴 버튼 모양 설정
- 툴 버튼 그룹 생성
- 툴 버튼 토글 허용

## 툴 버튼 추가

툴바에 툴 버튼을 추가하려면 툴바를 마우스 오른쪽 버튼을 클릭한 후 New Button을 선택합니다.

툴바는 툴버튼을 "소유"하므로 툴바를 이동하거나 숨기면 툴 버튼도 이동하거나 숨겨집니다. 또한 툴바의 모든 툴 버튼은 동일한 높이와 너비를 자동으로 유지합니다. 컴포넌트 팔레트의 다른 컨트롤을 툴바에 가져다 놓을 수 있으며 이러한 컨트롤은 동일한 높이를 자동으로 유지합니다. 컨트롤이 툴바의 수평 공간 안에 다 들어 가지 않으면 랩어라운드하거나 새 행에서 시작합니다.

예를 들어, 다음 코드는 1에서 10을 왼쪽에서 오른쪽의 순서로 툴 버튼을 추가하여 루프에서 툴 버튼을 추가하는 방법을 보여 줍니다.

```
Toolbar1:= TToolbar.create(self);
Toolbar1.Parent := Form1;
Toolbar1.ShowCaptions := True;
for i := 0 to 10 do
begin
  Toolbutton := TToolbutton.Create(Self);
  {$IFDEF LINUX}
  Toolbutton.Toolbar := Toolbar1;
  {$ENDIF}
  {$IFDEF WINDOWS}
  Toolbutton.Parent := Toolbar1;
  {$ENDIF}
  Toolbutton.Caption := '#' + inttostr(i);
end;
end;
```

Windows의 동일한 코드는 툴바의 왼쪽에서 시작하여 10에서 1의 순서로 툴 버튼을 추가한다는 점을 크로스 플랫폼 개발자는 참고하십시오.

## 툴 버튼에 이미지 할당

각각의 툴 버튼에는 런타임에 버튼에 나타날 이미지를 결정하는 *ImageIndex* 속성이 있습니다. 툴 버튼에 이미지를 하나만 제공할 경우, 버튼은 해당 이미지를 처리하여 버튼이 사용 불가능한지 여부를 나타냅니다. 다음과 같은 방법으로 디자인 타임 시 툴 버튼에 이미지를 할당합니다.

- 1 버튼을 표시하는 툴바를 선택합니다.
- 2 Object Inspector에서 *TImageList* 객체를 툴바의 *Images* 속성에 할당합니다. 이미지 목록은 크기가 같은 아이콘 또는 비트맵의 컬렉션입니다.
- 3 툴 버튼을 선택합니다.
- 4 Object Inspector에서 버튼에 지정할 이미지 목록의 이미지에 해당하는 툴 버튼의 *ImageIndex* 속성에 정수를 할당합니다.

툴 버튼이 사용 불가능하거나 마우스 포인터 아래에 있을 때 별도의 이미지가 해당 툴 버튼에 나타나도록 지정할 수도 있습니다. 그렇게 하려면 툴바의 *DisabledImages* 및 *HotImages* 속성에 별도의 이미지 목록을 할당합니다.



## 툴 버튼 모양 및 초기 상태 설정

표 6.5는 툴 버튼 모양을 변경하기 위해 설정할 수 있는 일부 동작을 나열합니다.

표 6.5 툴 버튼 모양 설정

| 툴 버튼의 모양 변경               | 툴바의 속성 설정                                                                                          |
|---------------------------|----------------------------------------------------------------------------------------------------|
| 눌려진 상태로 표시                | <i>TToolButton</i> 에서 <i>Style</i> 속성을 <i>tbsCheck</i> 로 설정하고 <i>Down</i> 속성을 <i>True</i> 로 설정합니다. |
| 사용 불가능으로 표시               | <i>Enabled</i> 속성을 <i>False</i> 로 설정합니다.                                                           |
| 왼쪽 여백 표시                  | <i>Indent</i> 속성값을 0보다 큰 값으로 설정합니다.                                                                |
| "팝업" 테두리가 있어 툴바를 알기 쉽게 표시 | <i>Flat</i> 속성을 <i>True</i> 로 설정합니다.                                                               |

특정 툴 버튼 다음에 새로운 행의 컨트롤을 놓으려면 행의 마지막에 나타날 툴 버튼을 선택하고 *Wrap* 속성을 *True*로 설정합니다.

툴바의 자동 줄 바꿈(auto-wrap) 기능을 해제하려면 툴바의 *Wrapable* 속성을 *False*로 설정합니다.

## 툴 버튼 그룹 생성

툴 버튼 그룹을 만들려면 연결할 버튼을 선택하고 *Style* 속성을 *tbsCheck*로 설정한 다음 *Grouped* 속성을 *True*로 설정합니다. 그룹화된 툴 버튼을 선택하면 그룹 내의 다른 버튼이 팝업되어 상호 배타적 선택 집합을 나타내는 데 도움이 됩니다.

*Style*은 *tbsCheck*로 설정되고 *Grouped*은 *True*로 설정된 일련의 인접한 툴 버튼은 단일 그룹을 형성합니다. 툴 버튼 그룹을 해제하려면 다음과 같은 버튼을 이용해서 분리합니다.

- *Grouped* 속성이 *False*인 툴 버튼
- *Style* 속성이 *tbsCheck*로 설정되지 않은 툴 버튼 툴바에 공백이나 구분선을 만들려면 *Style*이 *tbsSeparator* 또는 *tbsDivider*로 설정된 툴 버튼을 추가합니다.
- 툴 버튼 이외의 다른 컨트롤

## 툴 버튼 토글 허용

*AllowAllUp*을 사용하여 토글처럼 동작하는 그룹화된 툴 버튼을 만듭니다. 그룹화된 버튼을 한 번 누르면 눌러지고 다시 누르면 팝업됩니다. 그룹화된 툴 버튼을 토글로 만들려면 *AllowAllUp* 속성을 *True*로 설정합니다.

스피드 버튼에서 그룹 내 임의의 툴 버튼의 *AllowAllUp*을 *True*로 설정하면 그룹 내 모든 버튼에 대해 동일한 속성 값이 자동으로 설정됩니다.

## 클릭 이벤트에 응답

사용자가 툴바에서 버튼 같은 컨트롤을 클릭하는 경우 애플리케이션에서 이벤트 핸들러를 통해 응답하도록 하는 *OnClick* 이벤트를 생성합니다. *OnClick*은 버튼의 기본 이벤트이므로 디자인 타임에 버튼을 더블 클릭하여 이벤트의 빠른 핸들러를 생성할 수 있습니다.

## 툴 버튼에 메뉴 할당

툴 버튼(*TToolButton*)을 가지는 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 누를 때 메뉴가 자동으로 나타납니다.

## 숨겨진 툴바 추가

툴바가 항상 보일 필요는 없습니다. 경우에 따라서는 실제로 많은 툴바를 가지고 있어도 사용할 때만 툴바를 보이게 하는 것이 편리합니다. 툴바가 여러 개 있는 폼을 만들지만 툴바의 일부 또는 전부를 숨기는 경우도 있습니다.

다음과 같은 방법으로 숨겨진 툴바를 만듭니다.

- 1 폼에 툴바 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

경우에 따라, 애플리케이션에 툴바가 여러 개 있기를 원하지만 한 번에 모든 툴바를 폼에 나타내고 싶지 않은 경우도 있습니다. 또는 사용자가 툴바를 표시할지 여부를 결정하도록 할 수 있습니다. 모든 컴포넌트에서 런타임에 필요에 따라 툴바를 표시하거나 숨길 수 있습니다.

런타임에 툴바를 숨기거나 표시하려면 툴바의 *Visible* 속성을 각각 *False* 또는 *True*로 설정합니다. 대개 특정 사용자 이벤트에 대응해서 이렇게 설정하거나 애플리케이션의 작동 모드에서 이것을 변경합니다. 이렇게 하려면 전형적으로 각각의 툴바에서 닫기 버튼을 갖도록 합니다. 사용자가 닫기 버튼을 클릭하면 애플리케이션에서 해당 툴바가 숨겨집니다.

툴바를 토글하는 방법을 제공할 수도 있습니다. 다음 예제에서 펜(Pen)의 툴바는 메인 툴바의 버튼에서 토글됩니다. 한 번 클릭할 때마다 버튼이 눌러지거나 해제되므로 *OnClick* 이벤트 핸들러는 버튼이 눌러져 있는지 여부에 따라 Pen 툴바를 표시하거나 숨길 수 있습니다.

```
procedure TForm1.PenButtonClick(Sender:TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
```

## 컨트롤 사용

컨트롤은 스크롤 막대, 버튼, 텍스트 상자, 목록 상자 등과 같이 애플리케이션 사용자가 런타임 시 사용할 수 있는 비주얼 컴포넌트입니다. 일반적으로 컨트롤은 객체 계층 구조의 *TControl*에서 상속받는 객체입니다. 이 장에서는 일반적으로 사용되는 몇 가지 컨트롤에 대해 설명합니다.

### 컨트롤 내에서 텍스트 작업

---

다음 단원에서 다양한 편집 및 메모 컨트롤 기능 사용 방법을 설명합니다. 이 기능 중 일부는 편집 컨트롤에서도 사용됩니다.

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

### 텍스트 정렬 설정

---

메모 컴포넌트에서 텍스트는 왼쪽, 오른쪽 또는 가운데로 정렬할 수 있습니다. 텍스트 정렬을 변경하려면 편집 컴포넌트의 *Alignment* 속성을 설정합니다. *Alignment*는 *WordWrap* 속성이 *True*일 때에만 적용되며 금칙 처리가 해제되어 있으면 정렬할 여백이 없습니다. *WordWrap*은 금칙 처리를 설정하고 해제합니다. 설정되어 있으면 *WrapMode*, *WrapBreak*와 *WrapAtValue* 속성으로 금칙 처리가 적용되는 방법을 자세히 제어할 수 있습니다.

또한 *HMargin* 속성을 사용하여 메모 컨트롤에서 왼쪽 및 오른쪽 여백을 조정할 수 있습니다.

예를 들어, 다음 코드는 *Character|Left* 메뉴 항목에서 *OnClick* 이벤트 핸들러를 구현한 다음 *Character* 메뉴의 *Right* 및 *Center* 메뉴 항목에 동일한 이벤트 핸들러를 붙입니다.

```
procedure TForm1.Edit1Click(Sender:TObject);
begin
  Left1.Checked := False; { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True; { check the item clicked }
  with Editor do { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

## 런타임 시 스크롤 막대 추가

필요한 경우 편집과 메모 컴포넌트에 수평 또는 수직 스크롤 막대를 포함시킬 수 있습니다. 금칙 처리를 사용할 수 있을 경우 컴포넌트에는 수직 스크롤 막대만 필요합니다. 사용자가 금칙 처리를 해제하면 해당 텍스트가 편집기의 오른쪽에 제한받지 않으므로 컴포넌트에 수평 스크롤 막대가 필요할 수도 있습니다.

런타임 시 스크롤 막대를 추가하려면 다음과 같이 수행합니다.

- 1 텍스트가 오른쪽 여백을 초과하는 지 확인합니다. 대부분의 경우 이것은 금칙 처리의 사용 여부 확인을 의미합니다. 또한 텍스트 행이 실제로 컨트롤의 폭을 초과하는지 여부를 확인할 수도 있습니다.
- 2 편집 또는 메모 컴포넌트의 *ScrollBars* 속성에 스크롤 막대를 포함할 지 또는 제외할 지를 설정합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  with DataSet do
    begin
      WordWrap := not WordWrap; { toggle word-wrapping }
      if WordWrap then
        ScrollBars := ssVertical { wrapped requires only vertical }
      else
        ScrollBars := ssBoth; { unwrapped might need both }
        WordWrap1.Checked := WordWrap; { check menu item to match property }
    end;
end;
```

메모 사용 중에는 항상 스크롤 막대가 표시됩니다.

## 애플리케이션에 클립보드 추가

텍스트를 다루는 대부분의 애플리케이션은 다른 애플리케이션의 문서를 비롯해서 선택한 텍스트를 문서 사이에서 이동하는 방법을 제공합니다. Kylix의 *TClipboard* 객체에는 하나의 클립보드가 있으며 텍스트 및 그래픽을 포함한 다른 형태를 잘라내고, 복사하고 붙여넣는 메소드가 들어 있습니다. *TClipboard* 객체는 *QClipboard* 유닛에 선언되어 있습니다.

다음과 같은 방법으로 애플리케이션에 *TClipboard* 객체를 추가합니다.

- 1 클립보드를 사용할 유닛을 선택합니다.
- 2 **implementation** 예약어를 찾습니다.
- 3 **implementation** 아래의 **uses** 절에 *QClipboard*를 추가합니다.
  - **implementation** 부분에 이미 **uses** 절이 있으면 그 끝에 *QClipboard*를 추가합니다.
  - **uses** 절이 없으면 다음을 추가합니다.

```
uses QClipboard;
```

## 텍스트 선택

클립보드에 텍스트를 보내기 전에 해당 텍스트가 선택되어야 합니다. 선택한 텍스트의 선택 표시 기능은 편집 컴포넌트에 내장되어 있습니다. 사용자가 텍스트를 선택하면 선택 표시되어 나타납니다.

표 7.1은 선택한 텍스트를 처리하는 데 일반적으로 사용하는 속성을 나열한 것입니다.

**표 7.1** 선택한 텍스트의 속성

| 속성               | 설명                                   |
|------------------|--------------------------------------|
| <i>SelText</i>   | 컴포넌트 내에서 선택된 텍스트를 나타내는 문자열이 들어 있습니다. |
| <i>SelLength</i> | 선택한 문자열의 길이가 들어 있습니다.                |
| <i>SelStart</i>  | 문자열의 시작 위치가 들어 있습니다.                 |

예를 들어, 다음 *OnFind* 이벤트 핸들러는 찾기 대화 상자 컴포넌트의 *FindText* 속성에 지정된 텍스트에서 Memo 컴포넌트를 검색합니다. 검색하여 찾게 되면 Memo1에 처음으로 나타난 텍스트가 선택됩니다.

```
procedure TForm1.FindDialog1Find(Sender:TObject);
var
  I, J, PosReturn, SkipChars:Integer;
begin
  for I := 0 to Memo1.Lines.Count do
    begin
      PosReturn := Pos(FindDialog1.FindText,Memo1.Lines[I]);
      if PosReturn <> 0 then {found!}
      begin
        Skipchars := 0;
        for J := 0 to I - 1 do
```

```

        SkipChars := SkipChars + Length(Memo1.Lines[J]);
SkipChars := SkipChars + (I*2);
SkipChars := SkipChars + PosReturn - 1;
Memo1.SetFocus;
Memo1.SelStart := SkipChars;
Memo1.SelLength := Length(FindDialog1.FindText);
Break;
    end;
end;
end;

```

## 모든 텍스트 선택

*SelectAll* 메소드는 메모 컴포넌트의 전체 콘텐츠를 선택합니다. 이 메소드는 컴포넌트의 내용이 컴포넌트의 가시 영역을 벗어날 때 특히 유용합니다. 대부분의 다른 경우에는 사용자 키 입력이나 마우스를 끌어 텍스트를 선택합니다.

메모 컨트롤의 모든 내용을 선택하려면 해당 컨트롤의 *SelectAll* 메소드를 선택합니다. 예를 들면, 다음과 같습니다.

```

procedure TMainForm.SelectAll(Sender:TObject);
begin
    Memo1.SelectAll; { select all text in memo }
end;

```

## 텍스트 잘라내기, 복사 및 붙여넣기

*Clipbrd* 유닛을 사용하는 애플리케이션은 클립보드를 통해 텍스트, 그래픽 및 객체를 잘라내고, 복사하고 붙여 넣을 수 있습니다. 표준 텍스트 처리 컨트롤을 캡슐화하는 편집 컴포넌트에는 모두 클립보드를 사용하는 데 필요한 메소드가 들어 있습니다. (클립보드에서 그래픽을 사용하는 방법에 대한 자세한 내용은 8-21 페이지의 "클립보드에서 그래픽 사용"을 참조하십시오.)

클립보드로 텍스트를 잘라내고, 복사하거나 붙여 넣으려면 편집 컴포넌트의 *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 각각 호출합니다.

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

```

procedure TForm1.CutToClipboard(Sender:TObject);
begin
    Editor.CutToClipboard;
end;
procedure TForm1.CopyToClipboard(Sender:TObject);
begin
    Editor.CopyToClipboard;
end;
procedure TForm1.PasteFromClipboard(Sender:TObject);
begin
    Editor.PasteFromClipboard;
end;

```

## 선택한 텍스트 삭제

선택한 텍스트를 클립보드에서 잘라내지 않고 편집 컴포넌트에서 삭제할 수 있습니다. 삭제하려면 `ClearSelection` 메소드를 호출합니다. 예를 들어, Edit 메뉴에 Delete 항목이 있을 경우 다음과 같은 코드가 나타납니다.

```
procedure TForm1.FormCreate(Sender:TObject);
begin
    Memo1.ClearSelection;
end;
```

## 메뉴 항목 사용 불가능

가끔 메뉴에서 메뉴 명령을 제거하지 않고 사용 불가능하게 하는 것이 유용할 때가 있습니다. 예를 들어, 텍스트 편집기에서 텍스트를 선택하지 않으면 Cut, Copy 및 Delete 명령은 적용할 수 없습니다. 메뉴 항목을 사용 가능 또는 불가능하게 하는 할 적절한 시기는 사용자가 메뉴를 선택할 때입니다. 메뉴 항목을 사용 불가능하게 하려면 해당 `Enabled` 속성을 `False`로 설정합니다.

다음 예제에서, 이벤트 핸들러는 자식 폼 메뉴 모음의 Edit 항목에서 `OnClick` 이벤트에 붙습니다. `Memo1`에 선택한 텍스트가 있는지 여부에 따라 Edit 메뉴의 Cut, Copy 및 Delete 메뉴 항목에 `Enabled`를 설정합니다. Paste 명령은 클립보드에 텍스트가 있는지 여부에 따라 사용 가능 및 사용 불가능하게 됩니다.

```
procedure TForm1.Edit1Click(Sender:TObject);
var
    HasSelection:Boolean; { declare a temporary variable }
begin
    Paste1.Enabled := Clipboard.Provides('text'); {Enable/disable paste menu item}
    ...
end;
```

클립보드의 `Provides` 메소드는 클립보드에 특정 형식의 객체, 텍스트 또는 이미지가 있는지 여부에 따라 부울 값을 반환합니다. (이 경우 텍스트는 일반 문자입니다. 일반 텍스트의 경우는 `text/plain`을, `html`의 경우에는 `text/html`과 같은 하위 형식을 사용하여 텍스트 형식을 지정할 수 있습니다.) 매개변수 텍스트가 있는 `Provides`를 호출하면 클립보드에 텍스트가 있는지를 알 수 있으며 이에 따라 Paste 항목을 적절히 사용 가능 또는 불가능하게 할 수 있습니다.

클립보드에서 그래픽을 사용하는 것에 대한 자세한 내용은 8장 "그래픽 작업"에 있습니다.

## 팝업 메뉴 제공

팝업, 로컬, 메뉴는 모든 애플리케이션에서 사용하기 쉬운 일반적인 기능입니다. 이 기능을 사용하면 애플리케이션 작업 영역에서 마우스 오른쪽 단추를 클릭하여 마우스 움직임을 최소화할 수 있으므로 자주 사용하는 명령의 목록에 액세스할 수 있습니다.

예를 들어, 텍스트 편집기 애플리케이션에서 Cut, Copy 및 Paste 편집 명령을 반복하는 팝업 메뉴를 추가할 수 있습니다. 이 팝업 메뉴 항목에서 Edit 메뉴의 해당 항목과 동

일반 이벤트 핸들러를 사용할 수 있습니다. 해당 메뉴 항목에 이미 바로 가기가 있으므로 팝업 메뉴에 대한 가속기나 바로 가기 키를 만들 필요가 없습니다.

폼의 *PopupMenu* 속성에서 사용자는 폼에서 항목을 마우스 오른쪽 단추로 누를 때 표시할 팝업 메뉴를 지정합니다. 각 컨트롤에도 특정 컨트롤에 대해 사용자 지정 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 붙입니다.

## OnPopup 이벤트 처리

일반 메뉴에서 항목을 사용 가능 또는 불가능하게 하는 것처럼 메뉴를 표시하기 전에 팝업 메뉴 항목을 조정할 수 있습니다. 7-5 페이지의 "메뉴 항목 사용 불가능"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 이벤트를 메뉴 컴포넌트 자체에서 처리해야 합니다. *OnPopup*이라고 불리는 팝업 메뉴 컴포넌트는 자체 처리하는 목적만을 위한 이벤트를 제공합니다.

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

- 1 팝업 메뉴 컴포넌트를 선택합니다.
- 2 해당 *OnPopup* 이벤트에 이벤트 핸들러를 붙입니다.
- 3 이벤트 핸들러에 메뉴 항목을 사용 가능, 사용 불가능, 숨기거나 보이게 하는 코드를 작성합니다.

다음 코드에서는 앞의 7-5 페이지의 "메뉴 항목 사용 불가능"에 설명되었던 *Edit1Click* 이벤트 핸들러가 팝업 메뉴 컴포넌트의 *OnPopup* 이벤트에 붙습니다. 한 줄의 코드가 팝업 메뉴에 있는 각 항목에 대한 *Edit1Click*에 추가됩니다.

```
procedure TForm1.Edit1Click(Sender:TObject);
var
    HasSelection:Boolean;
begin
    Paste1.Enabled := Clipboard.Provides('text');
    Paste2.Enabled := Paste1.Enabled;{Add this line}
    HasSelection := Editor.SelLength <> 0;
    Cut1.Enabled := HasSelection;
    Cut2.Enabled := HasSelection;{Add this line}
    Copy1.Enabled := HasSelection;
    Copy2.Enabled := HasSelection;{Add this line}
    Delete1.Enabled := HasSelection;
end;
```



## 컨트롤에 그래픽 추가

몇몇 컨트롤에서는 컨트롤이 렌더링되는 방법을 사용자 지정할 수 있습니다. 여기에는 목록 상자, 콤보 상자, 메뉴, 헤더, 탭 컨트롤, 목록 뷰, 상태 표시줄, 트리 뷰 및 툴바 등이 있습니다. 컨트롤이나 해당 항목을 그리는 표준 메소드를 사용하는 대신 컨트롤의 소유자, 일반적으로 폼이 런타임 시 컨트롤이나 항목을 그립니다. `owner-draw` 컨트롤은 항목에 대한 텍스트 대신 또는 텍스트에 추가해서 그래픽을 제공하는 데 가장 일반적으로 사용됩니다. 메뉴에 이미지를 추가하기 위한 `owner-draw` 사용 방법에 대한 자세한 내용은 6-27 페이지의 "메뉴 항목에 이미지 추가"를 참조하십시오.

`TComboBox`와 `TListBox`는 항목의 목록을 포함하는 `owner-draw` 컨트롤입니다. 일반적으로 이 목록들은 텍스트로 표시되는 문자열 목록이거나 텍스트로 표시되는 문자열을 포함하는 객체의 목록입니다. 객체를 목록의 각 항목과 연결하여 항목을 그릴 때 해당 객체를 쉽게 사용할 수 있습니다.

일반적으로 Kylix에서 `owner-draw` 컨트롤을 작성하려면 다음과 같은 단계를 따릅니다.

- 1 컨트롤이 `owner-draw` 항목임을 나타내기
- 2 문자열 목록에 그래픽 객체 추가
- 3 `Owner-draw` 항목 그리기

### 컨트롤이 `owner-draw` 항목임을 나타내기

컨트롤 그리기를 사용자 지정하려면 색칠해야 할 때 컨트롤의 이미지를 렌더링하는 이벤트 핸들러를 제공해야 합니다. 일부 컨트롤은 이러한 이벤트를 자동으로 받습니다. 예를 들어 목록 뷰, 트리 뷰 및 툴바는 모두 사용자가 속성을 설정할 필요없이 그리는 과정에서 여러 단계에서 이벤트를 받습니다. 이러한 이벤트에는 `OnCustomDraw` 또는 `OnAdvancedCustomDraw`와 같은 이름이 있습니다.

그러나 다른 컨트롤은 `owner-draw` 이벤트를 받기 전에 사용자가 속성을 설정해야 합니다. 목록 상자, 콤보 상자, 헤더 컨트롤 및 상태 표시줄에는 `Style`이라는 속성이 있습니다. `Style`에서는 컨트롤에 "표준" 스타일이라고 하는 default drawing을 사용할지 `owner drawing`을 사용할지 여부를 결정합니다. 그리드는 `DefaultDrawing`이라는 속성을 사용하여 default drawing을 사용 가능 또는 사용 불가능하게 합니다. 탭 컨트롤에는 default drawing을 사용 가능 또는 사용 불가능하게 하는 `OwnerDraw`라는 속성이 있습니다.

목록 상자와 콤보 상자에는 표 7.2가 설명하는 것 처럼 *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 7.2** Fixed 및 variable *owner-draw* 스타일

| Owner-draw 스타일 | 의미                                             | 예                                                          |
|----------------|------------------------------------------------|------------------------------------------------------------|
| Fixed          | 각 항목은 <i>ItemHeight</i> 속성에서 결정된 동일한 높이를 갖습니다. | <i>lbOwnerDrawFixed</i> ,<br><i>csOwnerDrawFixed</i>       |
| Variable       | 각 항목은 런타임 시 데이터에 의해 결정된 다른 높이를 가질 수도 있습니다.     | <i>lbOwnerDrawVariable</i> ,<br><i>csOwnerDrawVariable</i> |

## 문자열 목록에 그래픽 객체 추가

모든 문자열 목록에는 해당 문자열 목록 외에 객체 목록을 포함하는 기능이 있습니다. 문자열 목록에 다양한 크기의 그래픽 객체를 추가할 수 있습니다.

예를 들어 파일 관리자 애플리케이션에서 드라이브 문자와 함께 드라이브의 형식을 나타내는 비트맵을 추가할 수 있습니다. 그렇게 하려면 애플리케이션에 비트맵 이미지를 추가한 후 다음 단원에 설명되어 있는대로 이 이미지들을 문자열 목록의 적당한 위치에 복사합니다.

*TImageList* 를 만들어 이미지 목록을 사용하여 그래픽 객체를 구성할 수 있습니다. 그러나 이 이미지들은 크기가 모두 같아야 합니다. 이미지 목록 설정의 예제를 보려면 6-27 페이지의 "메뉴 항목에 이미지 추가"를 참조하십시오.

## 애플리케이션에 이미지 추가

이미지 컨트롤은 비트맵과 같은 그래픽 이미지가 들어 있는 보이지 않는 컨트롤입니다. 이미지 컨트롤을 사용하면 폼에 그래픽 이미지를 표시할 수 있습니다. 또한 애플리케이션에서 사용할 숨겨진 이미지를 포착하는 데에도 사용할 수 있습니다. 예를 들어 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 원하는 비트맵을 선택하여 각 이미지의 *Picture* 속성을 설정합니다.

애플리케이션을 실행하면 이미지 컨트롤은 보이지 않습니다. 이미지는 폼과 함께 저장되므로 런타임 시 파일에서 로드하지 않아도 됩니다.

## owner-draw 항목 크기 설정

*variable owner-draw* 컨트롤에서 애플리케이션이 각 항목을 그리게 하기 전에 *measure-item* 이벤트 (*TMeasureItemEvent*) 가 생성됩니다. *measure-item* 이벤트에서는 컨트롤에 나타나는 항목의 위치를 애플리케이션에서 표시합니다.

Kylix는 항목의 크기를 결정합니다. 일반적으로 현재 글꼴로 항목의 텍스트를 표시하기에는 충분합니다. 애플리케이션은 이벤트를 처리하고 선택된 사각형을 변경할 수 있습니다.

다. 예를 들어 항목의 텍스트를 비트맵으로 대체하려는 경우 사각형을 비트맵의 크기로 변경합니다. 비트맵과 텍스트를 모두 넣으려면 둘 다 들어갈 수 있도록 사각형의 크기를 조정합니다.

owner-draw 항목의 크기를 변경하려면 owner-draw 컨트롤 내의 measure-item 이벤트에 이벤트 핸들러를 붙입니다. 목록 상자와 콤보 상자는 *OnMeasureItem*을 사용합니다.

크기 변경 이벤트에는 항목 인덱스 번호와 해당 항목의 높이라는 두 개의 중요한 매개변수가 있습니다. 높이는 변수이므로 애플리케이션에서 높이를 작거나 크게 할 수 있습니다. 연속되는 항목의 위치는 선행 항목의 크기에 따라 다릅니다.

예를 들어 variable owner-draw 목록 상자에서 애플리케이션이 첫 항목의 높이를 5 픽셀로 설정하면 두 번째 항목은 위로부터 6픽셀 아래에서 시작합니다. 목록 상자와 콤보 상자에서 애플리케이션이 변경할 수 있는 유일한 것은 항목의 높이입니다. 항목의 폭은 항상 컨트롤의 폭입니다.

owner-draw 목록 상자의 *OnMeasureItem* 이벤트에 붙은 다음 코드는 연결된 비트맵에 맞추기 위해 각 목록 항목의 높이를 늘립니다.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender:TObject; Index:Integer;
  var TabWidth:Integer); { note that TabWidth is a var parameter}
var
  BitmapWidth:Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

**참고** 문자열 목록의 *Objects* 속성에서 항목을 타입 변환해야 합니다. *Objects*는 *TObject* 타입의 속성으로 모든 형식의 객체를 포함할 수 있습니다. 배열에서 객체를 검색할 때 객체를 항목의 실제 타입으로 타입 변환해야 합니다.

## Owner-draw 항목 그리기

애플리케이션이 owner-draw 컨트롤을 그리거나 다시 그릴 때 Windows는 각 컨트롤의 보이는 항목의 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 붙입니다.

일반적으로 Owner drawing의 이벤트 이름은 다음 중 하나로 시작합니다.

- *OnDrawItem*이나 *OnDrawCell*처럼 *OnDraw*
- *OnCustomDrawItem*처럼 *OnCustomDraw*

Draw-item 이벤트에는 그릴 항목을 식별하는 매개변수, 그릴 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 목록 상자에 항목을 그리는 방법을 표시합니다. 다음과 같이 목록 상자의 *OnDrawItem* 이벤트에 이 핸들러를 붙입니다.

```
procedure TFMForm.DriveTabSetDrawTab(Sender:TObject; TabCanvas:TCanvas;  
    R:TRect; Index:Integer; Selected:Boolean);  
var  
    Bitmap:TBitmap;  
begin  
    Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);  
    with Canvas do  
        begin  
            Draw(R.Left, R.Top + 4, Bitmap); { draw bitmap }  
            TextOut(R.Left + 2 + Bitmap.Width, { position text }  
                R.Top + 2, DriveTabSet.Tabs[Index]); { and draw it to the right of the  
                }  
        end;  
end;
```

# 8

## 그래픽 작업

그래픽 요소로 애플리케이션을 더욱 화려하게 할 수 있습니다. CLX는 다양한 방법으로 애플리케이션에 그래픽을 추가할 수 있습니다. 그래픽 요소를 추가하려면 디자인 타임에 미리 그려놓은 그림을 삽입하거나 그래픽 컨트롤을 사용하여 만들거나 런타임 시 동적으로 그릴 수 있습니다.

### 그래픽 프로그래밍 개요

CLX 그래픽 컴포넌트는 Qt 그래픽 widget를 캡슐화하여 Linux 애플리케이션에 그래픽을 쉽게 추가할 수 있습니다.

CLX 애플리케이션에서 그래픽을 그리려면 객체 위에 직접 그리지 않고 객체의 *캔버스*에 그립니다. 캔버스는 객체의 속성이면서 그 자체가 객체이기도 합니다. 캔버스 객체의 주요 이점은 리소스를 효과적으로 처리하고 그래픽 컨텍스트를 관리하므로 프로그램에서 사용자가 화면이나 프린터, 비트맵 또는 드로잉에서 그림을 그리든지 동일한 메소드를 사용할 수 있다는 것입니다. 캔버스는 런타임 시에만 사용할 수 있으므로 코드를 작성하면 캔버스에서 모든 작업을 할 수 있습니다.

**참고** *TCanvas*는 Qt painter 주변의 래퍼(wrapper) 리소스 관리자입니다. 캔버스의 *Handle* 속성은 Qt painter 객체의 인스턴스에 대한 타입이 지정된 포인터입니다. 이 인스턴스 포인터를 드러내면 painter 객체에 대한 인스턴스 포인터를 필요로 하는 낮은 Qt 그래픽 라이브러리 기능을 사용할 수 있습니다.

애플리케이션에 그래픽 이미지가 나타나는 방식은 그리는 캔버스의 객체 형식에 따라 다릅니다. 컨트롤의 캔버스에 직접 그리면 그림은 즉시 표시됩니다. 그러나 *TBitmap* 캔버스와 같은 오프스크린 이미지에서 그리는 경우 이미지는 컨트롤이 비트맵에서 컨트롤의 캔버스로 복사되어야 표시됩니다. 즉, 비트맵을 그릴 이미지 컨트롤에 할당할 때 이미지는 컨트롤이 해당 *OnPaint* 이벤트를 처리할 수 있을 때에만 나타납니다.

**참고** 기본 명령줄에 *TBitmap*을 사용하면 프로그램에서 Xwindows가 실행되지 않는 경우 작동하지 않습니다. "ProjectName: cannot connect to X server."라는 메시지가 나타납니다.

그래픽을 사용하다 보면 *그리기 (Drawing)*와 *색칠 (Painting)*이라는 용어가 종종 나옵니다.

- 그리기는 코드가 있는 선이나 모양과 같은 특정 단일 그래픽 요소를 작성하는 것입니다. 코드에서 캔버스의 그리기 메소드를 호출하면 객체가 캔버스의 특정 위치에 특정 그래픽을 그릴 수 있습니다.
- 색칠은 객체의 전체 모양을 만드는 것입니다. 색칠은 일반적으로 그리기를 필요로 합니다. 즉, *OnPaint* 이벤트의 응답으로 객체는 일반적으로 일부 그래픽을 그립니다. 예를 들어, 편집 상자는 사각형을 그려 자신을 색칠한 다음 내부에 일부 텍스트를 그립니다. 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. 후반부에는 다른 이벤트에 응답하여 같은 종류의 그리기를 수행하는 방법을 보여줍니다.

## 화면 새로 고침

간혹 시스템에서는 화면 상의 객체 모양을 새로 고쳐야 한다는 결정을 내립니다. 예를 들어, 창을 움직이는 동안 폼이나 컨트롤이 일시적으로 흐려지면 해당 폼이나 컨트롤을 다른 위치에 다시 놓을 때 흐려진 영역을 다시 색칠해야 합니다. 이런 동작이 발생하면 색칠 이벤트가 생성되어 CLX는 *OnPaint* 이벤트로 라우트됩니다. CLX는 사용자가 *Refresh* 메소드를 사용할 때 해당 객체에 대해 작성한 모든 *OnPaint* 이벤트 핸들러를 호출합니다. 폼의 *OnPaint* 이벤트 핸들러에 생성된 기본 이름은 *FormPaint*입니다. 경우에 따라 *Refresh* 메소드를 사용하여 컴포넌트나 폼을 새로 고칠 수도 있습니다. 예를 들어, 폼의 *OnResize* 이벤트 핸들러에서 *Refresh*를 호출하여 그래픽을 다시 표시할 수 있습니다.

*TImage* 컨트롤을 사용하여 폼에 그래픽 이미지를 표시하는 경우 *TImage*에 들어 있는 그래픽의 색칠 및 새로 고침은 CLX에 의해 자동으로 처리됩니다. *Picture* 속성은 *TImage*가 표시하는 실제 비트맵, 그리기 또는 다른 그래픽 객체를 지정합니다. *TImage*의 그리기를 통해 영구적인 이미지를 만듭니다. 따라서 포함된 이미지를 다시 그릴 필요가 없습니다. 반면에 *TPaintBox*의 캔버스는 painter에 직접 연결되어 있으므로 *PaintBox*의 캔버스에 그려진 모든 그림은 일시적으로 그려진 것입니다. 폼 자체를 비롯한 거의 모든 컨트롤 또한 일시적으로 그려진 것입니다. 따라서 해당 생성자에 있는 *TPaintBox* 상에서 그리거나 색칠하는 경우 클라이언트 영역이 바뀔 때마다 이미지가 다시 칠해지도록 *OnPaint* 이벤트 핸들러에 해당 코드를 추가해야 합니다.

## 그래픽 객체 형식

CLX는 표 8.1에 표시된 그래픽 객체를 제공합니다. 이러한 객체는 8-9 페이지의 "캔버스 메소드를 사용하여 그래픽 객체 그리기"에 설명되어 있는 캔버스에 그릴 수 있고,

8-21 페이지의 "클립보드에서 그래픽 사용"에서 설명한대로 그래픽 파일로 로드 및 저장할 수 있는 메소드를 가집니다.

**표 8.1** 그래픽 객체 형식

| 객체   | 설명                                                                                                                                                                                                                          |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 그림   | 모든 그래픽 이미지를 갖기 위해 사용됩니다. 추가 그래픽 파일 형식을 추가하려면 <i>Picture Register</i> 메소드를 사용합니다. 이 메소드를 사용하면 이미지 컨트롤에서 이미지를 표시하는 것과 같이 임의의 파일을 처리할 수 있습니다.                                                                                  |
| 비트맵  | 이미지를 작성, 처리(크기 변경, 스크롤, 회전 및 색칠) 및 디스크에 파일로 저장하는 데 사용하는 강력한 그래픽 객체입니다. 비트맵의 사본을 만드는 것은 이미지가 아닌 <i>처리</i> 가 복사되는 것이므로 빠릅니다.                                                                                                  |
| 클립보드 | 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.                                                                |
| 아이콘  | 아이콘 파일에서 로드된 값을 나타냅니다.                                                                                                                                                                                                      |
| 그리기  | 이미지의 실제 비트맵 픽셀이 들어 있는 것이 아니라 이미지를 생성하는 데 필요한 작업을 기록하는 파일이 들어 있습니다. 그리기는 이미지의 상세 정보를 잃지 않고 크기를 조정할 수 있으며 특히 프린터와 같은 고해상도 장치에서는 비트맵보다 훨씬 적은 메모리를 필요로 하기도 합니다. 그러나 그리기는 비트맵만큼 빨리 표시되지 않습니다. 실행보다 기능과 정확성이 더 중요할 때 그리기를 사용하십시오. |

## 캔버스의 일반적인 속성과 메소드

표 8.2는 캔버스 객체에서 공통적으로 사용되는 속성을 나열한 것입니다. 속성과 메소드의 전체 목록은 온라인 도움말의 *TCanvas* 컴포넌트를 참조하십시오.

**표 8.2** 캔버스 객체의 일반적인 속성

| 속성     | 설명                                                                                                     |
|--------|--------------------------------------------------------------------------------------------------------|
| Font   | 이미지 위에 텍스트를 작성할 때 사용할 글꼴을 지정합니다. TFont 객체의 속성을 설정하여 글꼴, 글꼴 색, 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.                                                                                   |

이 속성은 8-5 페이지의 "캔버스 객체의 속성 사용"에 더 자세히 설명되어 있습니다.

표 8.3은 사용 가능한 메소드의 목록입니다.

**표 8.3 캔버스 객체의 일반적인 속성**

| 메소드                   | 설명                                                                                                                                             |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Arc                   | 지정된 사각형으로 경계를 이루는 타원의 둘레를 따라 이미지에 원호를 그립니다.                                                                                                    |
| Chord                 | 선과 타원의 교차점으로 표시되는 닫힌 그림을 그립니다.                                                                                                                 |
| CopyRect              | 다른 캔버스에서 현재 캔버스로 이미지의 부분을 복사합니다.                                                                                                               |
| Draw                  | 좌표 (X, Y)에 의해 지정된 위치의 캔버스에 Graphic 매개변수에 의해 지정된 그래픽 객체를 렌더링합니다.                                                                                |
| DrawFocusRect         | Mode가 자동으로 pmXor로 설정된 펜으로 키보드 포커스를 나타내는 데 사용되는 점선인 사각형을 그립니다. 이 기능은 애플리케이션 객체의 Style 속성에 있는 DefaultStyle 멤버가 dsWindows로 설정되어 있을 경우에만 영향을 미칩니다. |
| DrawPoint             | 현재 펜을 사용하여 캔버스에 단일점을 그립니다.                                                                                                                     |
| DrawPoints            | 현재 펜을 사용하여 캔버스에 일련의 점을 그립니다.                                                                                                                   |
| Ellipse               | 캔버스에 경계를 이루는 사각형에 의해 정의된 타원을 그립니다.                                                                                                             |
| FillRect              | 현재 브러시를 사용하여 캔버스에 지정된 사각형을 채웁니다.                                                                                                               |
| GetClipRegion         | 포인터를 현재 설정된 잘라낸 지역으로 반환합니다.                                                                                                                    |
| LineTo                | 캔버스에 PenPos에서 X와 Y로 지정된 지점까지 선을 그리고 펜 위치를 (X, Y) 지점으로 설정합니다.                                                                                   |
| MoveTo                | 현재 그리기 위치를 (X,Y) 지점으로 변경합니다.                                                                                                                   |
| Pie                   | 캔버스의 (X1, Y1)과 (X2, Y2) 사각형에 의해 경계를 이루는 파이 모양의 타원 섹션을 그립니다.                                                                                    |
| PolyBezier            | 현재 펜을 사용하여 입체 베지어 곡선을 그립니다.                                                                                                                    |
| PolyBezierTo          | 베지어 곡선 집합을 그리고 PenPos 값을 업데이트합니다.                                                                                                              |
| Polygon               | 마지막 지점에서 첫 번째 지점까지 선을 그려서 통과 지점을 연결하고 모양을 닫는 일련의 선을 캔버스에 그립니다.                                                                                 |
| Polyline              | 현재 펜으로 캔버스 위에 Points에 통과한 각 지점을 연결시키는 일련의 선을 그립니다.                                                                                             |
| Rectangle             | 왼쪽 위 모서리 (X1, Y1) 점과 우측 아래 모서리의 (X2, Y2) 점으로 이루어진 사각형을 캔버스에 그립니다. 펜을 사용하여 <i>Rectangle</i> 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 나타나지 않습니다.                                                                                                     |
| TiledDraw             | 지정된 사각형에 바둑판식 비트맵을 그립니다.                                                                                                                       |

이 메소드는 8-9 페이지의 "캔버스 메소드를 사용하여 그래픽 객체 그리기"에 더 자세히 설명되어 있습니다.



## 캔버스 객체의 속성 사용

캔버스 객체를 사용하여 선을 그리는 펜, 모양을 채우는 브러시, 텍스트를 작성하는 글꼴, 이미지를 나타내는 픽셀의 배열 속성을 설정할 수 있습니다.

이 단원에는 다음과 같은 내용이 설명되어 있습니다.

- 펜 사용
- 브러시 사용

### 펜 사용

캔버스 컨트롤의 *Pen* 속성은 모양의 윤곽으로 그려진 선을 비롯하여 선이 나타나는 방법을 제어합니다. 직선을 그리는 것은 두 지점 간에 있는 픽셀 그룹을 변경하는 것입니다.

펜에는 사용자가 변경할 수 있는 *Color*, *Width*, *Style* 및 *Mode*와 같은 네 가지 속성이 있습니다.

- Color 속성: 펜 색상 변경
- Width 속성: 펜 너비 변경
- Style 속성: 펜 스타일 변경
- Mode 속성: 펜 모드 변경

이 속성 값으로 펜이 선의 픽셀을 변경하는 방법을 결정합니다. 기본적으로 모든 펜은 1 픽셀의 너비, 실선 스타일 및 캔버스에 이미 존재하는 모든 것을 오버라이드하는 copy 라는 모드를 가지고 검정으로 시작합니다.

### 펜 색상 변경

런타임 시 다른 *Color* 속성을 설정하는 것처럼 색상을 설정할 수 있습니다. 펜 색상으로 다른 선 및 다각선 뿐만 아니라 도형의 경계로 그려진 선을 비롯한 펜이 그린 선의 색상을 결정합니다. 펜 색상을 변경하려면 펜의 *Color* 속성에 값을 할당합니다.

사용자가 펜의 새 색상을 선택하도록 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.

```
procedure TForm1.PenColorClick(Sender:TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

## 펜 너비 변경

펜 너비는 펜이 그리는 선의 두께를 픽셀로 결정합니다.

**참고** Windows 95/98에 배포하기 위한 크로스 플랫폼 애플리케이션을 개발하고 두께가 1보다 큰 경우, Windows 95/98은 펜의 *Style* 속성 값에 관계없이 실선을 그립니다.

펜 너비를 변경하려면 펜의 *Width* 속성에 숫자로 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.

```
procedure TForm1.PenWidthChange(Sender:TObject);
begin
    Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
    PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
```

## 펜 스타일 변경

펜의 *Style* 속성을 통해 실선, 점괵선, 점선 등을 설정할 수 있습니다.

**참고** Windows 95/98에 배포하기 위한 크로스 플랫폼 애플리케이션을 개발하는 경우 Windows 95/98은 사용자가 지정하는 스타일에 관계없이 1픽셀이 넘는 점괵선이나 점선을 지원하지 않고 그보다 너비가 넓은 모든 펜은 실선으로 처리합니다.

펜 속성을 설정하는 작업은 이벤트를 처리하기 위해 서로 다른 컨트롤이 동일한 이벤트 핸들러를 공유하도록 하는 경우 이상적입니다. 어느 컨트롤이 실제로 이벤트를 갖는지 알려면 *Sender* 매개변수를 확인합니다.

펜의 툴바에 여섯 개의 펜 스타일 버튼에 대한 하나의 클릭 이벤트 핸들러를 만들려면 다음을 수행하십시오.

- 1 여섯 개의 펜 스타일 버튼을 모두 선택하고 Object Inspector|Events| *OnClick* 이벤트를 선택한 다음 Handler 열에 *SetPenStyle*을 입력합니다.
- 2 IDE는 *SetPenStyle*이라는 빈 누르기 이벤트 핸들러를 생성하고 모든 여섯 개 버튼의 *OnClick* 이벤트에 첨부합니다.

- 3 클릭 이벤트를 보낸 컨트롤인 *Sender* 값에 따라 다른 펜 스타일을 설정하여 클릭 이벤트 핸들러를 다음과 같은 방법으로 채웁니다.

```
procedure TForm1.SetPenStyle(Sender:TObject);
begin
  with Canvas.Pen do
    begin
      if Sender = SolidPen then Style := psSolid
      else if Sender = DashPen then Style := psDash
      else if Sender = DotPen then Style := psDot
      else if Sender = DashDotPen then Style := psDashDot
      else if Sender = DashDotDotPen then Style := psDashDotDot
      else if Sender = ClearPen then Style := psClear;
    end;
  end;
```

## 펜 모드 변경

펜의 *Mode* 속성을 사용하여 펜 색상과 캔버스 색상을 결합하기 위해 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. 펜 위치는 선 그리기에만 영향을 미치며 도형과 텍스트의 경우 필요한 모든 좌표를 사용자가 입력합니다.

펜 위치를 설정하려면 캔버스의 *MoveTo* 메소드를 호출합니다. 예를 들어, 다음 코드를 사용하여 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 인접한 많은 픽셀을 변경하는 방법입니다.

브러시에는 사용자가 처리할 수 있는 다음과 같은 세 가지 속성이 있습니다.

- Color 속성: 채우기 색상 변경
- Style 속성: 브러시 스타일 변경
- Bitmap 속성: 비트맵을 브러시 패턴으로 사용

이 속성 값으로 캔버스가 도형이나 다른 영역을 채우는 방법을 결정합니다. 기본적으로 모든 브러시는 흰색의 실선 스타일로 시작하며 패턴 비트맵을 사용하지 않습니다.

## 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 배경색 속성을 설정합니다.

브러시 툴바의 색상 그리드를 클릭하면, 펜 색상과 동일한 방법으로 브러시 색상을 설정할 수 있습니다(8-5 페이지의 "펜 색상 변경" 참고).

```
procedure TForm1.BrushColorClick(Sender:TObject);
begin
    Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

## 브러시 스타일 변경

브러시 스타일은 캔버스가 도형을 채우는 데 사용하는 패턴을 결정합니다. 브러시 스타일을 사용하여 브러시 색상과 캔버스에 이미 있는 색상을 결합하기 위해 다양한 방법을 지정할 수 있습니다. 이미 정의된 스타일에는 단색, 무색 및 다양한 선과 해칭 패턴이 있습니다.

브러시 스타일을 변경하려면 해당 *Style* 속성을 *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, *bsDiagCross*, *bsDense1*, *bsDense2*, *bsDense3*, *bsDense4*, *bsDense5*, *bsDense6*, 또는 *bsDense7* 같은 이미 정의된 값 중 하나로 설정합니다.

이 예제는 15개의 브러시 스타일 버튼 집합에 대한 클릭 이벤트 핸들러를 공유하여 브러시 스타일을 설정합니다. 15개의 버튼을 모두 선택하고 Object Inspector|Events|OnClick 으로 설정한 다음 OnClick 핸들러의 이름을 *SetBrushStyle*로 지정합니다. 핸들러 코드는 다음과 같습니다.

```
procedure TForm1.SetBrushStyle(Sender:TObject);
begin
    with Canvas.Brush do
    begin
        if Sender = SolidBrush then Style := bsSolid
        else if Sender = ClearBrush then Style := bsClear
        else if Sender = HorizontalBrush then Style := bsHorizontal
        else if Sender = VerticalBrush then Style := bsVertical
        else if Sender = FDiagonalBrush then Style := bsFDiagonal
        else if Sender = BDiagonalBrush then Style := bsBDiagonal
        else if Sender = CrossBrush then Style := bsCross
        else if Sender = DiagCrossBrush then Style := bsDiagCross;
    end;
end;
```

## 브러시의 비트맵 속성 설정

브러시의 *Bitmap* 속성을 사용하여 도형 및 다른 영역을 채우기 위한 패턴으로 사용하는 브러시의 비트맵 이미지를 지정합니다.

다음 예제는 파일에서 비트맵을 로드하고 비트맵을 Form1 Canvas의 Brush에 할당합니다.

```
var
  Bitmap:TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;
```

**참고** 브러시는 자신의 *Bitmap* 속성에 할당된 비트맵 객체의 소유권을 가지고 있다고 가정하지 않습니다. 브러시를 사용하는 동안에는 비트맵 객체가 유효한 상태로 있는지를 확인해야 하고 나중에 비트맵 객체를 해제해야 합니다.

## 캔버스 메소드를 사용하여 그래픽 객체 그리기

---

이 단원에서는 일반적인 일부 메소드를 사용하여 그래픽 객체를 그리는 방법을 설명합니다. 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

캔버스에서 직선을 그리려면 캔버스의 *LineTo* 메소드를 사용합니다.

*LineTo*는 현재의 펜 위치에서 사용자가 지정하는 펜 위치까지 선을 그리고 선의 끝점을 현재 위치로 나타냅니다. 캔버스에서는 펜을 사용하여 선을 그립니다.

예를 들어, 다음 메소드는 폼을 색칠할 때마다 폼 전반에 교차된 대각선을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  with Canvas do
    begin
      MoveTo(0, 0);
      LineTo(ClientWidth, ClientHeight);
      MoveTo(0, ClientHeight);
      LineTo(ClientWidth, 0);
    end;
end;
```

## 다각선 그리기

개별적인 선 외에도 캔버스에서는 연결된 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 빨리 그립니다. 그 이유는 많은 호출 오버헤드를 제거하기 때문입니다.

예를 들어, 다음 메소드는 폼에 마름모꼴을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  with Canvas do
    Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

위의 예제는 개방형 배열 매개변수를 즉시 만드는 Kylix 기능을 이용한 것입니다. 어떤 점의 배열이라도 전달할 수 있지만 빠르게 배열을 구성할 수 있는 쉬운 방법은 해당 요소를 괄호 안에 넣고 전체를 매개변수로 전달하는 것입니다. 자세한 내용은 온라인 도움말을 참조하십시오.

## 도형 그리기

캔버스에는 다른 종류의 도형을 그리는 데 사용하는 메소드가 있습니다. 캔버스에서는 펜으로 도형의 윤곽을 그린 다음 브러시로 내부를 채웁니다. 도형의 테두리를 형성하는 선은 현재 *Pen* 객체에 의해 제어됩니다.

이 단원에서는 다음과 같은 내용을 다룹니다.

- 사각형 및 타원 그리기
- 모서리가 둥근 사각형 그리기
- 다각형 그리기

## 사각형 및 타원 그리기

캔버스에 사각형이나 타원을 그리려면 경계를 이루는 사각형의 좌표를 전달하는 캔버스의 *Rectangle* 메소드나 *Ellipse* 메소드를 호출합니다.

*Rectangle* 메소드는 경계를 이루는 사각형을 그리고 *Ellipse* 메소드는 사각형의 모든 면에 닿는 타원을 그립니다.

다음 메소드는 폼의 왼쪽 위 사분면을 채우는 사각형을 그린 다음 동일한 영역에 타원을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
  Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

## 모서리가 둥근 사각형 그리기

캔버스에 모서리가 둥근 사각형을 그리려면 캔버스의 *RoundRect* 메소드를 호출합니다.

*RoundRect*에 전달된 첫 번째 네 개의 매개변수는 *Rectangle* 메소드나 *Ellipse* 메소드의 경우처럼 경계를 이루는 사각형입니다. *RoundRect*는 둥근 모서리를 그리는 방법을 나타내는 매개변수를 두 개 더 가집니다.

예를 들어, 다음 메소드는 지름이 10 픽셀인 원처럼 모서리를 둥글게 하면서 폼의 왼쪽 위 사분면에 모서리가 둥근 사각형을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

## 다각형 그리기

캔버스에 여러 개의 면을 갖는 다각형을 그리려면 캔버스의 *Polygon* 메소드를 호출합니다.

*Polygon* 메소드는 점의 배열을 자신의 유일한 매개변수로 갖고 펜으로 점을 연결한 다음 마지막 점을 첫 번째 점과 연결하여 다각형을 닫습니다. 선을 그린 후에는, *Polygon* 메소드가 브러시를 사용하여 다각형 내부의 영역을 채웁니다.

예를 들어, 다음 코드는 폼의 왼쪽 아래 중간 부분에 직각 삼각형을 그립니다.

```
procedure TForm1.FormPaint(Sender:TObject);
begin
  Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
    Point(ClientWidth, ClientHeight)]);
end;
```

## 애플리케이션에서 여러 드로잉 객체 처리

다양한 그리기 메소드(사각형, 도형, 선 등)는 일반적으로 툴바와 버튼 패널에서 사용할 수 있습니다. 애플리케이션은 원하는 드로잉 객체를 설정하기 위해 스피드 버튼 누르기에 응답할 수 있습니다. 이 단원에서는 다음과 같은 방법을 설명합니다.

- 사용할 드로잉 툴 파악
- 스피드 버튼으로 툴 변경
- 드로잉 툴 사용

### 사용할 드로잉 툴 파악

그래픽 프로그램은 사용자가 언제든 사용하려는 드로잉 툴(선, 사각형, 타원 또는 모서리가 둥근 사각형)의 종류를 파악해야 합니다. 각 툴의 종류마다 번호를 할당할 수는 있지만 그럴 경우 각 번호가 어느 툴에 해당하는지를 기억해야 합니다. 각 번호에 기억을 돕는 상수 이름을 할당하면 쉽게 구별할 수 있지만 코드는 어떤 번호가 해당 범위 내에 있고 알맞은 타입인지를 구별하지 못합니다. 다행스럽게도 오브젝트 파스칼은 이러한 두 가지 부족함을 처리할 방법을 제공합니다. 바로 열거 타입을 선언하는 방법입니다.

열거 타입은 상수에 순차적으로 값을 할당하는 빠른 방법입니다. 이 방법 또한 타입 선언이므로 오브젝트 파스칼의 타입 검사를 사용하여 이러한 특정 값만을 할당하도록 할 수 있습니다.

열거 타입을 선언하려면 예약 작업 타입 다음에 타입 식별자, 등호 및 타입 내의 값 식별자를 괄호에 넣고 쉼표로 구분합니다.

예를 들어, 다음 코드는 그래픽 애플리케이션에서 사용 가능한 각 드로잉 툴의 열거 타입을 선언합니다.

```
type
    TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

규칙에 의해 타입 식별자는 T로 시작하고 열거 타입을 구성하는 것과 같은 유사한 상수 그룹은 두 글자의 접두사로 시작합니다. 예를 들어, 드로잉 툴(drawing tool)에서는 dt와 같이 두 글자의 접두사로 시작합니다.

TDrawingTool 타입의 선언은 상수 그룹을 선언하는 것과 같습니다.

```
const
    dtLine = 0;
    dtRectangle = 1;
    dtEllipse = 2;
    dtRoundRect = 3;
```

주요 차이점은 열거 타입을 선언함으로써 상수에 값을 부여하는 것이 아니라 오브젝트 파스칼의 타입 검사를 사용하여 많은 오류를 방지해 줄 타입도 부여한다는 것입니다. TDrawingTool 타입의 변수는 dtLine..dtRoundRect 상수 중 오직 하나에만 할당될 수 있습니다. 다른 숫자(0..3의 범위 중 하나)를 할당하려고 시도하면 컴파일 시 오류가 발생합니다.



다음 코드에서 폼에 추가된 필드는 폼의 드로잉 툴을 확보합니다.

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
  TForm1 = class(TForm)
    ...{ method declarations }
  public
    Drawing: Boolean;
    Origin, MovePt: TPoint;
    DrawingTool: TDrawingTool; { field to hold current tool }
  end;
```

## 스피드 버튼으로 툴 변경

---

각 드로잉 툴은 연결된 *OnClick* 이벤트 핸들러가 필요합니다. 애플리케이션에 선, 사각형, 타원 및 모서리가 둥근 사각형과 같은 각 네 개의 드로잉 툴에 툴바 버튼이 있다고 가정합니다. 다음 이벤트 핸들러를 네 개의 드로잉 툴 버튼의 *OnClick* 이벤트에 첨부하고 *DrawingTool*을 각각에 알맞는 값으로 설정합니다.

```
procedure TForm1.LineButtonClick(Sender: TObject); { LineButton }
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject); { RectangleButton }
begin
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject); { EllipseButton }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject); { RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;
```

## 드로잉 툴 사용

---

이제 어느 툴을 사용해야 하는지 구별할 수 있으므로 다른 도형을 그리는 방법을 나타내야 합니다. 그리기를 수행하는 유일한 메소드는 마우스 이동과 마우스 업 핸들러이며 어떤 툴을 선택하든지 그리기 코드 만이 선을 그립니다.

다른 드로잉 툴을 사용하려면 코드는 선택한 툴을 기반으로 그리는 방법을 지정해야 합니다. 각 툴의 이벤트 핸들러에 이 지침을 추가합니다.

이 단원에서 다음을 설명합니다.

- 도형 그리기
- 이벤트 핸들러 간에 코드 공유

## 도형 그리기

도형을 그리는 것은 선을 그리는 것 만큼 쉽습니다. 각 도형들이 단일문을 취하므로 사용자는 좌표만 있으면 됩니다.

네 개의 툴 모두에 대해 도형을 그리기 위한 *OnMouseUp* 이벤트 핸들러를 다시 작성하는 방법은 다음과 같습니다.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                               X,Y:Integer);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
        Canvas.LineTo(X, Y);
      end;
    dtRectangle:Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
    dtEllipse:Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
    dtRoundRect:Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                                   (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
  Drawing := False;
end;
```

물론 도형을 그리려면 *OnMouseMove* 핸들러를 다음과 같이 업데이트해야 합니다.

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift:TShiftState; X, Y:Integer);
begin
  if Drawing then
    begin
      Canvas.Pen.Mode := pmNotXor;
      case DrawingTool of
        dtLine:begin
          Canvas.MoveTo(Origin.X, Origin.Y);
          Canvas.LineTo(MovePt.X, MovePt.Y);
          Canvas.MoveTo(Origin.X, Origin.Y);
          Canvas.LineTo(X, Y);
        end;
        dtRectangle:begin
          Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
          Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
        end;
        dtEllipse:begin
          Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
          Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
        end;
        dtRoundRect:begin
          Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                             (Origin.X - X) div 2, (Origin.Y - Y) div 2);
          Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                             (Origin.X - X) div 2, (Origin.Y - Y) div 2);
        end;
```

```

    end;
    MovePt := Point(X, Y);
  end;
  Canvas.Pen.Mode := pmCopy;
end;

```

일반적으로 위 예제에 있는 모든 반복적인 코드들은 별도의 루틴에 있습니다. 다음 단원은 모든 마우스 이벤트 핸들러에서 호출할 수 있는 단일 루틴의 모든 도형 그리기 코드를 보여줍니다.

## 이벤트 핸들러 간의 코드 공유

대다수의 이벤트 핸들러가 동일한 코드를 사용할 경우 모든 이벤트 핸들러가 공유할 수 있는 루틴으로 반복 코드를 이동시키면 애플리케이션을 더 효율적으로 만들 수 있습니다.

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 안전할 것입니다.

- 2 폼 유닛의 구현 부분에 메소드 구현을 작성합니다.

메소드 구현의 헤더는 동일한 순서의 동일한 매개변수를 가진 선언과 정확히 일치해야 합니다.

다음 코드는 *DrawShape*라는 폼에 메소드를 추가하고 각 핸들러에서 메소드를 호출합니다. 먼저 *DrawShape*의 선언이 폼 객체의 선언에 다음과 같은 방법으로 추가됩니다.

```

type
  TForm1 = class(TForm)
    ...{ fields and methods declared here}
  public
    { Public declarations }
    procedure DrawShape(TopLeft, BottomRight:TPoint; AMode:TPenMode);
  end;

```

그런 다음 *DrawShape*의 구현을 유닛의 구현 부분에 다음과 같이 작성합니다.

```

implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight:TPoint; AMode:TPenMode);
begin
  with Canvas do
    begin
      Pen.Mode := AMode;
      case DrawingTool of
        dtLine:
          begin
            MoveTo(TopLeft.X, TopLeft.Y);
            LineTo(BottomRight.X, BottomRight.Y);
          end;

```

```

dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
    (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
end;
end;
end;

```

다른 이벤트 핸들러는 *DrawShape*를 호출하기 위해 수정됩니다.

```

procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
    Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender:TObject; Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    if Drawing then
    begin
        DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
        MovePt := Point(X, Y);{ record the current point }
        DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
    end;
end;

```

## 그래픽에서 그리기

애플리케이션의 그래픽 객체를 처리하는 데 컴포넌트가 필요하지 않습니다. 화면에 아무 것도 그리지 않고 그래픽 객체를 생성하고, 그리고, 저장하고 소멸시킬 수 있습니다. 사실 애플리케이션에서는 폼에 직접 그리지 않습니다. 오히려 애플리케이션은 그래픽에서 작동하며 CLX 이미지 컨트롤 컴포넌트를 사용하여 폼에 그래픽을 표시합니다.

일단 애플리케이션의 드로잉을 이미지 컨트롤의 그래픽으로 이동하면 그래픽 객체에 인쇄, 클립보드 및 로딩과 저장 동작을 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 할당하게 되면 이미지는 컨트롤이 해당 색칠 메시지를 처리할 수 있을 경우에만 나타납니다. 그러나 컨트롤의 캔버스 속성에 직접 그리는 경우 그림 객체는 즉시 표시됩니다.

## 스크롤 가능한 그래픽 만들기

그래픽은 폼과 크기가 같지 않아도 됩니다. 더 크거나 작을 수 있습니다. 폼에 스크롤 상자 컨트롤을 추가하고 내부에 그래픽 이미지를 두면 폼보다 더 크거나 심지어 화면보다도 큰 그래픽을 표시할 수 있습니다. 스크롤 가능한 그래픽을 추가하려면 먼저 *TScrollBar* 컴포넌트를 추가한 다음 이미지 컨트롤을 추가합니다.

## 이미지 컨트롤 추가

이미지 컨트롤은 비트맵 객체를 표시할 수 있는 컨테이너 컴포넌트입니다. 이미지 컨트롤을 사용하면 항상 표시될 필요가 없거나 애플리케이션에서 다른 그림을 생성하기 위해 필요한 비트맵을 가질 수 있습니다.

**참고** 7-7 페이지의 "컨트롤에 그래픽 추가"에서는 컨트롤에서 그래픽을 사용하는 방법을 보여줍니다.

### 컨트롤 두기

이미지 컨트롤은 폼의 어느 곳에도 둘 수 있습니다. 그림에 자신의 크기를 맞추는 이미지 컨트롤 기능을 이용할 경우, 왼쪽 위 모서리만 설정하면 됩니다. 이미지 컨트롤이 비트맵에서 보이지 않는 표시자인 경우에는 논비주얼 (nonvisual) 컴포넌트와 마찬가지로 아무 곳에도 둘 수 있습니다.

폼의 클라이언트 영역에 이미 정렬된 스크롤 상자에 이미지 컨트롤을 가져다 놓는 경우 스크롤 상자가 이미지의 그림 중 오프스크린 부분에 액세스하는 데 필요한 모든 스크롤 막대를 추가합니다. 그런 다음 이미지 컨트롤의 속성을 설정합니다.

### 초기 비트맵 크기 설정

어떤 이미지 컨트롤을 두게 되면 단지 컨테이너에 불과합니다. 그러나 디자인 타임에 이미지 컨트롤의 *Picture* 속성을 설정하여 정적 그래픽을 포함할 수는 있습니다. 그리고 8-21 페이지의 "클립보드에서 그래픽 사용"에서 설명한대로 컨트롤은 런타임 시 파일에서 컨트롤 그림을 로드할 수도 있습니다.

다음과 같은 방법으로 애플리케이션이 시작할 때 빈 비트맵을 만듭니다.

- 1 이미지가 들어 있는 폼의 *OnCreate* 이벤트에 핸들러를 첨부합니다.
- 2 비트맵 객체를 만들고 비트맵 객체를 이미지 컨트롤의 *Picture.Graphic* 속성에 할당합니다.

이 예제에서 이미지는 애플리케이션의 메인 폼인 *Form1*에 있으므로 코드는 다음과 같은 방법으로 *Form1*의 *OnCreate* 이벤트에 핸들러를 첨부합니다.

```
procedure TForm1.FormCreate(Sender:TObject);
var
  Bitmap: TBitmap;{ temporary variable to hold the bitmap }
begin
  Bitmap := TBitmap.Create;{ construct 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 }
  Bitmap.Free; {We are done with the bitmap, so free it }
end;
```

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 창의 크기를 조정하는 경우 이미지의 나머지 부분을 표시할 수 있도록 스크롤 상자가 스크롤 막대를 자동으로 표시하는 것을 볼 수 있습니다. 그러나 이미지 위에서 그리려는 경우 현재 이미지와 스크롤 상자 뒤에 있는 폼에서 애플리케이션의 그리기 작업이 아직 진행 중이므로 그래픽이 나타나지 않습니다.

## 비트맵에 그리기

비트맵에서 그리려면 이미지 컨트롤의 캔버스를 사용하고 이미지 컨트롤의 해당 이벤트에 마우스 이벤트 핸들러를 첨부합니다. 일반적으로 영역 동작(채우기, 사각형, 다각선 등)을 사용합니다. 영역 동작은 빠르고 효율적인 그리기 방법입니다.

각 픽셀에 액세스해야 할 때 이미지를 그리는 효과적인 방법은 비트맵 *ScanLine* 속성을 사용하는 것입니다. 일반적인 목적으로 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 얻는 방법을 보여줍니다.

```
procedure TForm1.Button1Click(Sender:TObject);
// This example shows drawing directly to the Bitmap
var
  x,y :integer;
  Bitmap:TBitmap;
  P : PByteArray;
begin
  Bitmap := TBitmap.create;
  try
    if OpenFileDialog1.Execute then
      begin
        Bitmap.LoadFromFile(OpenDialog1.FileName);
        for y := 0 to Bitmap.height -1 do
          begin
            P := Bitmap.ScanLine[y];
            for x := 0 to Bitmap.width -1 do
              P[x] := y;
            end;
          end;
        canvas.draw(0,0,Bitmap);
      finally
        Bitmap.free;
      end;
    end;
```

## 그래픽 파일 로드 및 저장

하나의 애플리케이션이 실행되는 동안에만 존재하는 그래픽 이미지는 가치가 매우 제한적입니다. 경우에 따라 동일한 그림을 항상 사용하려고 하거나 만들어진 그림을 나중에 사용하기 위해 저장하고자 할 수 있습니다. 이미지 컴포넌트를 사용하면 파일에서 쉽게 그림을 로드하고 다시 저장할 수 있습니다.

그래픽 이미지를 로드, 저장 및 교체하는 데 사용하는 CLX 컴포넌트는 비트맵 파일, png 파일, xpm, Windows 아이콘 등 많은 그래픽 형식을 지원합니다. 또한 CLX 컴포넌트는 설치 가능한 그래픽 클래스를 지원합니다.

그래픽 파일을 로드하고 저장하는 방법은 다른 파일과 유사하며 다음 단원에 설명되어 있습니다.

- 파일에서 그림 로드
- 그림을 파일로 저장
- 그림 교체

### 파일에서 그림 로드

애플리케이션에서 그림을 수정해야 하거나 애플리케이션 외부에 그림을 저장하려는 경우 파일에서 그림을 로드하는 기능을 제공해야 합니다. 그러면 다른 사용자나 애플리케이션에서 그림을 수정할 수 있습니다.

이미지 컨트롤로 그래픽 파일을 로드하려면 이미지 컨트롤의 *Picture* 객체의 *LoadFromFile* 메소드를 호출합니다.

다음 코드는 그림 파일 열기 대화 상자에서 파일 이름을 선택한 다음 해당 파일을 *Image*라는 이름의 이미지 컨트롤로 해당 파일을 로드합니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  if OpenFileDialog1.Execute then
    begin
      CurrentFile := OpenFileDialog1.FileName;
      Image.Picture.LoadFromFile(CurrentFile);
    end;
end;
```

### 그림을 파일에 저장

그림 객체는 여러 가지 형태로 그래픽을 로드 및 저장할 수 있으며 그림 객체가 그래픽을 로드하고 저장할 수 있도록 그래픽 파일 형식을 직접 만들고 등록할 수 있습니다.

파일에 이미지 컨트롤의 내용을 저장하려면 이미지 컨트롤의 *Picture* 객체의 *SaveToFile* 메소드를 호출합니다.

*SaveToFile* 메소드에는 저장할 파일 이름이 필요합니다. 그림을 새로 만든 경우 파일 이름이 없거나 사용자가 기존 그림을 다른 파일에 저장할 수도 있습니다. 어느 경우든지 다음 단원에 나와 있는대로 사용자는 애플리케이션에 저장하기 전에 파일 이름을 부여해야 합니다.

File|Save 및 File|Save As 메뉴 항목에 첨부된 다음 이벤트 핸들러 쌍은 차례대로, 이름이 있는 파일의 재저장, 이름이 없는 파일의 저장 그리고 기존 파일을 새 이름으로 저장하는 기능을 처리합니다.

```

procedure TForm1.Button1Click(Sender:TObject);
begin
    if CurrentFile <> '' then
        Image.Picture.SaveToFile(CurrentFile){ save if already named }
    else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Button1Click(Sender:TObject);
begin
    if SaveDialog1.Execute then{ get a file name }
    begin
        CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
        Save1Click(Sender);{ then save normally }
    end;
end;

```

## 그림 교체

이미지 컨트롤의 그림은 언제든지 교체할 수 있습니다. 이미 그래픽이 있는 그림에 새 그래픽을 할당하면 기존 그래픽이 새 그래픽으로 바뀝니다.

이미지 컨트롤에서 그림을 변경하려면 이미지 컨트롤의 *Picture* 객체에 새 그래픽을 할당합니다.

새 그래픽을 만드는 것은 처음으로 그래픽을 만드는 것과 동일한 과정을 거치지만(8-17 페이지의 "초기 비트맵 크기 설정" 참조), 초기 그래픽에 사용된 기본 크기와 다른 새 그래픽에 사용할 크기를 선택하는 방법도 제공해야 합니다. 크기 선택 옵션을 제공하는 간단한 방법은 대화 상자를 제시하는 것입니다. 프로젝트에서 대화 상자를 사용하여 메인 폼의 유닛에 있는 *uses* 절에 대화 상자를 추가합니다. 그런 다음 File|New 메뉴 항목의 *OnClick* 이벤트에 이벤트 핸들러를 첨부할 수 있습니다. 예제는 다음과 같습니다.

```

procedure TForm1.New1Click(Sender:TObject);
var
    Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
    with Canvas do
    begin
        ActiveControl := WidthEdit;{ make sure focus is on width field }
        WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
        HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
        if ShowModal <> idCancel then{ continue if user doesn't cancel dialog box }
        begin
            Bitmap := TBitmap.Create;{ create fresh bitmap object }
            Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
            Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
            Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
            CurrentFile := '';{ indicate unnamed file }
            Bitmap.Free;
        end;
    end;
end;

```



**참고** 그림 객체의 *Graphic* 속성에 새 비트맵을 할당하면 그림 객체는 새 그래픽을 복사하지만 소유권을 갖지는 않습니다. 그림 객체는 자신의 내부 그래픽 객체를 유지합니다. 이로 인해 이전의 코드는 비트맵 객체를 할당한 다음 해제합니다.

## 클립보드에서 그래픽 사용

클립보드를 사용하여 애플리케이션 내에서 그래픽을 복사하고 붙여넣거나 다른 애플리케이션과 그래픽을 교환할 수 있습니다. 클립보드 객체를 사용하면 그래픽이 들어 있는 다른 종류의 정보를 쉽게 처리할 수 있습니다.

애플리케이션에서 클립보드 객체를 사용하기 전에 클립보드 데이터에 액세스해야 하는 유닛의 *uses* 절에 *QClipbrd* 유닛을 추가해야 합니다.

클립보드에 저장된 데이터는 연결된 *TStream* 객체를 갖는 *mime* 타입으로 저장됩니다. CLX는 다음의 이미 정의된 *mime* 소스와 *mime* 타입 문자열 상수를 다음 CLX 객체에 제공합니다.

- *TBitmap* = 'image/delphi.bitmap'
- *TComponent* = 'application/delphi.component'
- *TPicture* = 'image/delphi.picture'
- *TDrawing* = 'image/delphi.drawing'

## 클립보드에 그래픽 복사

클립보드에 이미지 컨트롤의 콘텐츠를 포함하는 모든 그림을 복사할 수 있습니다. 클립보드에 저장되면 그림은 모든 애플리케이션에서 사용할 수 있습니다.

클립보드에 그림을 복사하려면 *Assign* 메소드를 사용하여 클립보드 객체에 그림을 할당합니다.

다음 코드는 *Edit|Copy* 메뉴 항목을 눌러서 *Image*라고 명명된 이미지 컨트롤에서 클립보드로 그림을 복사하는 방법을 보여줍니다.

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

## 클립보드로 그림 잘라내기

클립보드로 그림을 잘라내는 것은 복사하는 것과 완전히 똑같은 방법이지만 잘라내기 경우 원본에서 그래픽을 지워줍니다.

그림에서 그래픽을 잘라내어 클립보드에 넣으려면 먼저 클립보드에 그래픽을 복사하고 원본을 지웁니다.

대부분의 경우 잘라내기의 유일한 문제점은 원본 이미지가 지워졌다는 것을 보여주는 방법에 있습니다. 다음과 같이 코드가 *Edit|Cut* 메뉴 항목의 *OnClick* 이벤트에 이벤트

핸들러를 첨부하는 경우에서처럼 해당 영역을 하얗게 설정하는 것이 일반적인 해결 방법입니다.

```

procedure TForm1.Cut1Click(Sender:TObject);
var
    ARect:TRect;
begin
    Copy1Click(Sender);{ copy picture to clipboard }
    with Image.Canvas do
        begin
            CopyMode := cmWhiteness;{ copy everything as white }
            ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
            CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
            CopyMode := cmSrcCopy;{ restore normal mode }
        end;
    end;

```

## 클립보드에서 그림 붙여넣기

클립보드에 비트맵 그래픽이 들어 있는 경우 이미지 컨트롤과 폼의 표면을 비롯한 모든 이미지 객체에 클립보드를 붙여넣을 수 있습니다.

다음과 같은 방법으로 클립보드에서 그래픽을 붙여넣습니다.

- 1 클립보드의 *Provides* 메소드를 호출하여 클립보드에 그래픽이 들어 있는지 확인합니다.

*Provides*는 부울 함수입니다. *Provides*는 클립보드에 매개변수에 지정된 형식의 항목이 들어 있는 경우 *True*를 반환합니다.

- 2 대상에 클립보드를 할당합니다.

다음 코드는 Edit|Paste 메뉴 항목을 눌러서 클립보드에서 이미지 컨트롤로 그림을 붙여넣는 방법을 보여줍니다.

```

procedure TForm1.PasteButtonClick(Sender:TObject);
var
    Bitmap:TBitmap;
begin
    if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? }
        begin
            Image1.Picture.Bitmap.Assign(Clipboard);
        end;
    end;

```

클립보드의 그래픽은 이 애플리케이션에 가져오거나 다른 애플리케이션에서 복사할 수 있습니다. 클립보드에 지원하는 형식이 없을 때에는 붙여 넣기 메뉴를 사용할 수 없으므로 이러한 경우에서 클립보드 형식은 확인할 필요가 없습니다.

## 양단 묶음(Rubber banding) 예제

이 단원에서는 사용자가 런타임 시 그래픽을 그릴 때 마우스 이동을 추적하는 그래픽 애플리케이션에서의 "양단 묶음" 효과를 구현하는 것에 대한 자세한 내용을 설명합니다. 애플리케이션에서는 누르고 드래그하여 창의 캔버스에 선과 도형을 그립니다. 즉, 마우스 버튼을 누르면 그리기를 시작하고 버튼을 놓으면 그리기를 끝냅니다.

처음에는 예제 코드에서 메인 폼의 표면에 그리는 방법을 보여줍니다. 이후의 예제에서는 비트맵에 그리는 것을 보여줍니다.

다음 항목에서는 해당 예제를 설명합니다.

- 마우스에 응답
- 마우스 동작을 추적하기 위해 폼 객체에 필드 추가
- 선 그리기 작업 정제

### 마우스에 응답

애플리케이션은 마우스 버튼 다운, 마우스 이동 및 마우스 버튼 업 등의 마우스 동작에 응답할 수 있습니다. 또한 모드 있는 대화 상자에서 *Enter*를 누르는 것과 같이 키 입력으로 만들 수 있는 누르기(한 번에 누르기 및 놓기 가능)에 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

마우스 이벤트는 사용자가 애플리케이션의 사용자 인터페이스에서 마우스를 이동할 때 발생합니다. CLX에는 세 가지의 마우스 이벤트가 있습니다.

**표 8.4** 마우스 이벤트

| 이벤트                    | 설명                                          |
|------------------------|---------------------------------------------|
| <i>OnMouseDown</i> 이벤트 | 컨트롤 위에 마우스 포인터를 놓고 마우스 버튼을 누를 때 발생합니다.      |
| <i>OnMouseMove</i> 이벤트 | 컨트롤 위에 마우스 포인터가 있는 동안 마우스를 움직일 때 발생합니다.     |
| <i>OnMouseUp</i> 이벤트   | 컴포넌트 위에 마우스 포인터를 놓고 눌렀던 마우스 버튼을 놓을 때 발생합니다. |

CLX 애플리케이션은 마우스 동작을 발견하면 다섯 개의 매개변수를 전달해서 해당 이벤트에 대해 사용자가 정의한 이벤트 핸들러를 호출합니다. 이벤트에 대한 응답을 사용자 지정하려면 이 매개변수의 정보를 사용하십시오. 다섯 개의 매개변수는 다음과 같습니다.

**표 8.5** 마우스 이벤트 매개변수

| 매개변수          | 의미                                                                               |
|---------------|----------------------------------------------------------------------------------|
| <i>Sender</i> | 마우스 동작을 발견한 객체입니다.                                                               |
| <i>Button</i> | 다음의 <i>mbLeft</i> , <i>mbMiddle</i> 또는 <i>mbRight</i> 중 어느 마우스 버튼과 관계 있는지 나타냅니다. |
| <i>Shift</i>  | 마우스 동작 시 <i>Alt</i> , <i>Ctrl</i> 및 <i>Shift</i> 키의 상태를 나타냅니다.                   |
| <i>X, Y</i>   | 이벤트가 발생한 좌표입니다.                                                                  |

대부분 마우스 이벤트 핸들러에 반환된 좌표가 필요하지만 경우에 따라 어떤 마우스 버튼이 이벤트를 유발했는지 알려면 *Button*을 확인해야 합니다.

## 마우스 다운 동작에 응답

사용자가 마우스의 버튼을 누를 때마다 *OnMouseDown* 이벤트는 포인터가 위치한 객체로 전달됩니다. 그러면 객체는 이벤트에 응답할 수 있습니다.

마우스 다운 동작에 응답하려면, *OnMouseDown* 이벤트에 이벤트 핸들러를 첨부합니다.

CLX는 다음과 같은 방법으로 폼에 마우스 다운 이벤트의 빈 핸들러를 생성합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
end;
```

다음 코드를 사용하면 마우스로 누른 폼의 해당 위치에 'Here!'라는 문자열을 표시합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.TextOut(X, Y, 'Here!');{ write text at (X, Y) }
end;
```

애플리케이션이 실행되면 폼에 마우스 커서로 마우스 버튼을 눌러 눌러진 지점에 "Here!" 문자열이 나타나게 할 수 있습니다. 다음 코드는 사용자가 버튼을 누른 좌표에 현재 그리기 위치를 설정합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.MoveTo(X, Y);{ set pen position }
end;
```

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. 버튼을 누를 때 마우스 커서가 위치한 객체로 이벤트가 전달되지만 버튼을 놓을 때의 커서가 놓인 위치가 동일한 객체일 필요는 없습니다. 이러한 방식을 사용하면 폼의 테두리를 넘어 선을 그릴 수 있습니다.

마우스 업 동작에 응답하려면 *OnMouseUp* 이벤트의 핸들러를 정의해야 합니다.

다음은 마우스 버튼을 놓는 지점까지 선을 그리는 간단한 *OnMouseUp* 이벤트 핸들러입니다.

```
procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
```

이 코드를 통해 사용자는 누르기, 끌기 및 놓기를 사용하여 선을 그릴 수 있습니다. 이러한 경우 버튼을 놓으면 선을 볼 수 있습니다.

## 마우스 이동에 응답

*OnMouseMove* 이벤트는 마우스를 이동할 때 정기적으로 발생합니다. 이벤트는 버튼을 누를 때 마우스 포인터 아래에 있던 객체로 전달됩니다. 이를 통해 마우스가 움직이는 동안 일시적인 선을 그림으로써 사용자에게 중간 피드백을 줄 수 있습니다.

마우스 이동에 응답하려면 *OnMouseMove* 이벤트의 이벤트 핸들러를 정의해야 합니다. 다음 예제에서는 마우스 이동 이벤트를 사용하여 사용자가 마우스 버튼을 누르고 있는 동안 폼에 중간 도형을 그려서 사용자에게 일부 피드백을 제공합니다. *OnMouseMove* 이벤트 핸들러는 *OnMouseMove* 이벤트가 발생한 위치의 폼에 선을 그립니다.

```
procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.LineTo(X, Y);{ draw line to current position }
end;
```

위의 코드를 사용하여 마우스를 폼 위로 이동하면 마우스 버튼을 누르기 전에도 마우스를 따라 그려지게 됩니다.

마우스 이동 이벤트는 마우스 버튼을 누르지 않았을 때에도 발생합니다.

눌려진 마우스 버튼이 있는지 추적하려면 폼 객체에 객체 필드를 추가해야 합니다.

## 마우스 동작을 추적하기 위해 폼 객체에 필드 추가

마우스 버튼이 눌러 졌는지 추적하려면 폼 객체에 객체 필드를 추가해야 합니다. 폼에 컴포넌트를 추가할 때 Kylix는 해당 필드 이름으로 컴포넌트를 참조할 수 있도록 폼 객체에 해당 컴포넌트를 나타내는 필드를 추가합니다. 폼 유닛의 헤더 파일에 타입 선언을 편집하여 폼에 직접 만든 필드를 추가할 수도 있습니다.

다음 예제를 보면 폼은 사용자가 마우스 버튼을 눌렀는지 여부를 추적해야 합니다. 그렇게 하려면 폼에서 부울 필드를 추가하고 사용자가 마우스 버튼을 누를 때 해당 값을 설정합니다.

객체에 필드를 추가하려면 선언 아래의 **public** 지시어 뒤에 필드 식별자와 타입을 지정하는 객체 타입 정의를 편집합니다.

Kylix는 **public** 지시어 앞의 모든 선언을 "소유"합니다. 그러므로 **public** 지시어 앞에 컨트롤을 나타내는 필드와 이벤트에 응답하는 메소드를 넣습니다.

다음 코드는 폼 객체의 선언에서 부울 타입의 *Drawing*이라는 필드를 폼에 제공합니다. 또한 *Origin* 지점과 *TPoint* 타입의 *MovePt*를 저장하는 두 개의 필드도 추가합니다.

```
type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
    procedure FormMouseDown(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
    procedure TForm1.FormMouseMove(Sender:TObject; Button:TMouseButton;
      Shift:TShiftState; X, Y:Integer);
  public
    Drawing: Boolean;{ field to track whether button was pressed }
    Origin, MovePt: TPoint;{ fields to store points }
  end;
```

그릴 것인지 여부를 추적하는 *Drawing* 필드가 있으면 다음과 같은 방법으로 사용자가 마우스 버튼을 누를 때 *True*로 설정하고 놓을 때 *False*로 설정합니다.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Drawing := True;{ set the Drawing flag }
  Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.LineTo(X, Y);
  Drawing := False;{ clear the Drawing flag }
end;
```

그런 다음 다음과 같은 방법으로 *OnMouseMove* 이벤트 핸들러를 수정하면 *Drawing*이 *True*일 때에만 그릴 수 있습니다.

```
procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  if Drawing then{ only draw if Drawing flag is set }
    Canvas.LineTo(X, Y);
end;
```

이렇게 하면 마우스 다운과 마우스 업 이벤트 간에만 그릴 수 있지만 여전히 직선 대신 마우스 이동을 추적하는 점선이 나타납니다.

그러나 문제는 마우스를 이동할 때마다 마우스 이동 이벤트 핸들러가 펜 위치를 이동시키는 *LineTo*를 호출하여 사용자가 마우스 버튼을 놓을 때까지 직선이 시작하는 지점을 잃게 되는 것입니다.

## 선 그리기 다듬기

다양한 지점을 추적하는 필드를 적절히 사용하면 애플리케이션의 선 그리기를 다듬을 수 있습니다.

### 원래 지점 추적

선을 그릴 때 선이 *Origin* 필드와 함께 시작하는 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.

```
procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender:TObject; Button:TMouseButton;
  Shift:TShiftState; X, Y:Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
  Canvas.LineTo(X, Y);
  Drawing := False;
end;
```

이렇게 변경하면 애플리케이션에서는 마지막 선을 다시 그리지만 중간 동작을 그리지 않고 애플리케이션은 "양단 묶음"을 지원하지 않습니다.

### 이동 추적

이 예제의 문제점은 현재 작성된 *OnMouseMove* 이벤트 핸들러가 원래 위치가 아니라 마지막 *마우스 위치*에서 현재의 마우스 위치로 선을 그린다는 것입니다. 원래 지점으로 그리기 위치를 이동한 다음 현재 위치에서 그림으로써 이 문제를 해결할 수 있습니다.

```

procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    if Drawing then
        begin
            Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
            Canvas.LineTo(X, Y);
        end;
    end;

```

위의 코드는 현재 마우스 위치를 추적하지만 중간 선이 사라지지 않으므로 마지막 선을 거의 볼 수 없습니다. 예제에서는 다음 선을 그리기 전에 이전의 선이 있던 위치를 계속 추적하여 선을 하나씩 지워야 합니다. *MovePt* 필드를 통해 선을 지우는 작업을 할 수 있습니다.

*MovePt*는 각 중간 선의 끝점으로 설정되어야 하므로 *MovePt*와 *Origin*을 사용하여 다음에 선이 그려질 때 해당 선을 지웁니다.

```

procedure TForm1.FormMouseDown(Sender:TObject; Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    Drawing := True;
    Canvas.MoveTo(X, Y);
    Origin := Point(X, Y);
    MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender:TObject;Button:TMouseButton;
    Shift:TShiftState; X, Y:Integer);
begin
    if Drawing then
        begin
            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 the old line }
            Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
            Canvas.LineTo(X, Y);{ draw the new line }
        end;
        MovePt := Point(X, Y);{ record point for next move }
        Canvas.Pen.Mode := pmCopy;
    end;

```

이제 선을 그릴 때 "양단 묶음" 효과가 나타납니다. 펜의 모드를 *pmNotXor*로 변경하면 선을 배경 픽셀과 조합할 수 있습니다. 선을 지울 때 픽셀을 이전의 상태로 설정하게 됩니다. 선을 그린 후 펜 모드를 기본값인 *pmCopy*로 다시 변경하여 마우스 버튼을 놓으면 펜은 마지막 그리기 작업을 준비합니다.



## 다중 스레드 애플리케이션 작성

CLX는 다중 스레드 애플리케이션을 쉽게 작성할 수 있도록 여러 가지 객체를 제공합니다. 다중 스레드 애플리케이션은 실행을 동시에 할 수 있는 여러 개의 경로가 들어 있는 애플리케이션입니다. 다중 스레드를 사용하는 데에는 세심한 주의를 요하지만 다음과 같은 작업을 수행하여 프로그램을 향상시킬 수 있습니다.

- **병목 현상 방지.** 스레드가 하나인 경우에는 디스크의 파일 액세스, 다른 컴퓨터와의 통신 또는 멀티미디어 콘텐츠 표시 등의 느린 프로세스를 기다리는 동안 프로그램에서 모든 실행을 중지해야 합니다. CPU는 프로세스가 완료될 때까지 유휴 상태로 있습니다. 그러나 다중 스레드를 사용하면 애플리케이션에서 하나의 스레드가 느린 프로세스의 결과를 기다리는 동안 별도의 스레드에서 다른 프로세스를 계속 실행할 수 있습니다.
- **프로그램 동작 조직화.** 경우에 따라 프로그램의 동작을 독립적으로 기능하는 여러 개의 병렬 프로세스로 조직화할 수 있습니다. 스레드를 사용하면 이러한 병렬 프로세스들에 대해 단일 코드 섹션을 동시에 실행할 수 있습니다. 보다 중요한 작업에 CPU 시간을 부여할 수 있도록 스레드를 사용하여 다양한 프로그램 작업에 우선 순위를 할당하십시오.
- **멀티프로세스.** 프로그램을 실행하는 시스템에 다중 프로세서가 있으면 작업을 여러 스레드로 나누고 별도의 프로세서에서 동시에 실행시켜 성능을 향상시킬 수 있습니다.

### 참고

Linux는 Intel MP 아키텍처를 갖는 멀티프로세스 운영 체제입니다. 프로세스는 자신의 권한과 책임을 갖는 별도의 작업입니다. 각 프로세스는 자신의 가상 주소 공간에서 실행되며 안전하고 핵심적으로 관리되는 메커니즘을 제외하고는 다른 프로세스와 상호작용할 수 없습니다.

## 스레드 객체 정의

대부분의 애플리케이션에서 스레드 객체를 사용하면 애플리케이션에 실행 스레드를 나타낼 수 있습니다. 스레드 객체는 가장 일반적으로 필요한 스레드 사용을 캡슐화하여 다중 스레드 애플리케이션 작성을 단순화합니다.

스레드 객체를 통해 보안 속성이나 스레드의 스택 크기를 제어할 수는 없습니다. 보안 속성이나 스레드의 스택 크기를 제어해야 할 경우 *BeginThread* 함수를 사용해야 합니다. *BeginThread*를 사용하더라도 9-6 페이지의 "스레드 조정"에 설명되어 있는 일부 스레드 동기화 객체 및 메소드에서 이점을 얻을 수 있습니다. *BeginThread* 사용에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

애플리케이션에서 스레드 객체를 사용하려면 *TThread*의 새 자손을 만들어야 합니다. *TThread*의 자손을 만들려면 메인 메뉴에서 File|New를 선택합니다. 새 객체 대화 상자에서 Thread Object를 선택합니다. 새 스레드 객체의 클래스 이름을 입력하라는 메시지가 나타납니다. 이름을 입력하면 Kylix는 새 유닛 파일을 만들어 스레드를 구현합니다.

**참고** 클래스 이름을 필요로 하는 대부분의 IDE 대화 상자와는 달리 New Thread Object 대화 상자는 사용자가 입력한 클래스 이름 앞에 "T"를 자동으로 붙이지 않습니다.

자동으로 생성된 유닛 파일에는 새 스레드 객체의 뼈대 코드가 들어 있습니다. 스레드의 이름을 *TMyThread*로 지정한 경우 다음과 같이 나타납니다.

```
unit Unit1;
interface
uses
  Classes
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

*Execute* 메소드에 코드를 채워야 합니다. 그 단계에 대해서는 다음 단원에서 설명합니다.

## 스레드 초기화

새 스레드 클래스의 초기화 코드를 작성하려면 Create 메소드를 오버라이드해야 합니다. 스레드 클래스의 선언에 새 생성자를 추가하고 선언을 구현하는 초기화 코드를 작성합니다. 여기서 스레드의 기본 우선 순위를 할당하고 스레드가 실행 종료될 때 자동으로 해제되어야 할지 여부를 나타낼 수 있습니다.

## 기본 우선 순위 할당

우선 순위는 운영 체제가 애플리케이션의 모든 스레드 중에서 CPU 시간의 일정을 정할 때 스레드가 갖는 기본 설정 수를 나타냅니다. 시간적으로 중요한 작업을 처리할 때에는 높은 우선 순위의 스레드를 사용하고 다른 작업을 수행할 때에는 낮은 우선 순위의 스레드를 사용합니다. 스레드 객체의 우선 순위를 나타내려면 *Priority* 속성을 설정하십시오.

Linux에서 *Priority* 값은 정수이며 낮은 숫자가 더 높은 우선 순위를 나타냅니다.

**경고** CPU를 많이 사용하는 작업의 스레드 우선 순위를 높이면 애플리케이션의 다른 스레드를 사용하지 못하는 "기아" 상태가 될 수 있습니다. 외부 이벤트를 기다리는 데 대부분의 시간을 소비하는 스레드에만 높은 우선 순위를 적용합니다.

다음 코드는 다른 애플리케이션의 성능을 방해하지 말아야 할 백그라운드 작업을 수행하는 경우의 우선 순위가 낮은 스레드 생성자를 보여줍니다.

```
constructor TMyThread.Create(CreateSuspended:Boolean);
begin
    inherited Create(CreateSuspended);
    Priority := tpIdle;
end;
```

## 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 하나의 스레드 종료 시기를 다른 스레드 종료 시기와 일치시켜야 하는 경우가 있습니다. 예를 들어, 다른 스레드에서 동작을 수행하기 전에 값을 반환할 스레드를 기다릴 경우가 있습니다. 이렇게 두 개의 스레드 종료 작업을 일치시키기 위해 두 번째 스레드가 반환 값을 받은 다음 첫 번째 스레드를 해제하려고 합니다. 이러한 상황에서는 *FreeOnTerminate*를 *False*로 설정한 다음 두 번째 스레드에서 첫 번째 스레드를 명시적으로 해제함으로써 처리할 수 있습니다.

## 스레드 함수 작성

*Execute* 메소드가 스레드 함수입니다. 동일한 프로세스 공간을 공유하는 것을 제외하고는 스레드 함수가 애플리케이션에 의해 실행되는 프로그램으로 생각할 수 있습니다. 스레드 함수를 작성하는 것은 애플리케이션에서 다른 스레드에 의해 사용되는 메모리를 겹쳐 쓰지 않아야 하므로 별도의 프로그램을 작성하는 것보다 다소 까다롭습니다. 반면에 스레드가 다른 스레드와 동일한 프로세스 공간을 공유하므로 공유 메모리를 사용하여 스레드 간에 통신할 수 있습니다.

## 메인 CLX 스레드 사용

CLX 객체 계층에서 객체를 사용할 때 해당 속성과 메소드는 스레드에 대한 안전을 보장받지 못합니다. 즉, 속성에 액세스하거나 메소드를 실행하면 다른 스레드 동작에서 보호되지 않는 메모리를 사용하는 일부 동작을 수행할 수 있습니다. 이러한 이유로 CLX 객체를 액세스하기 위해 메인 CLX 스레드를 따로 둡니다.

모든 객체가 단일 스레드에서 해당 속성에 액세스하고 메소드를 실행할 경우에는 사용자의 객체들이 서로 방해하는 것을 걱정하지 않아도 됩니다. 메인 CLX 스레드를 사용하려면 필요한 동작을 수행하는 별도의 루틴을 만드십시오. 스레드의 *Synchronize* 메소드 내에서 별도의 루틴을 호출합니다. 예를 들면, 다음과 같습니다.

```
procedure TMyThread.PushTheButton;
begin
    Button1.Click;
end;
...
procedure TMyThread.Execute;
begin
    ...
    Synchronize(PushTheButton);
    ...
end;
```

*Synchronize*는 메인 CLX 스레드가 메시지 루프로 들어올 때까지 기다린 다음 전달된 메소드를 실행합니다.

**참고** *Synchronize*는 메시지 루프를 사용하므로 콘솔 애플리케이션에서는 작동하지 않습니다. 콘솔 애플리케이션에서 CLX 객체에 대한 액세스를 보호하려면 임계 구역 같은 다른 메커니즘을 사용해야 합니다.

항상 메인 CLX 스레드를 사용할 필요는 없습니다. 일부 객체에는 thread-aware 기능이 있습니다. 객체의 메소드가 스레드에 대해 안전하다는 것을 알고 *Synchronize* 메소드를 사용하지 않으면 CLX 스레드가 메시지 루프로 들어올 때까지 기다릴 필요가 없으므로 성능이 향상됩니다. 다음과 같은 상황에서는 *Synchronize* 메소드를 사용할 필요가 없습니다.

- 데이터 액세스 컴포넌트가 다음과 같이 스레드에 대해 안전한 경우 즉, 타사의 클라이언트 라이브러리가 스레드에 대해 안전하면 dbDirect 컴포넌트가 스레드에 대해 안전한 경우.

데이터 액세스 컴포넌트를 사용할 때 *Synchronize* 메소드의 data-aware 컨트롤을 포함하는 모든 호출을 연결해야 합니다. 그러므로, 예를 들어, 데이터 소스 객체의 *DataSet* 속성을 설정하여 데이터셋에 데이터 컨트롤을 연결하는 호출을 동기화해야 하지만 동기화하여 데이터셋 필드의 데이터에 액세스할 필요는 없습니다.

- VisualCLX 객체가 스레드에 대해 안전하지 않은 경우.
- DataCLX 객체가 스레드에 대해 안전한 경우.
- 그래픽 객체가 스레드에 대해 안전한 경우. 따라서 메인 CLX 스레드를 사용하여 *TFont*, *TPen*, *TBrush*, *TBitmap*, *TDrawing*, 또는 *TIcon*에 액세스하지 않아도 됩니다. 캔버스 객체는 *TFont*, *TPen*, *TBrush*, *TBitmap*, *TDrawing*, *TIcon* 등을 잠금으로써 *Synchronize* 메소드 외부에서 사용할 수 있습니다(9-7 페이지의 "객체 잠금" 참조).

- 목록 객체가 스레드에 대해 안전하지 않지만 *TList* 대신 스레드에 대해 안전한 버전인 *TThreadList*를 사용할 수 있을 경우.

백그라운드 스레드가 자신의 실행을 메인 스레드와 동기화할 수 있도록 애플리케이션의 메인 스레드 내에서 *CheckSynchronize* 루틴을 정기적으로 호출하십시오. *CheckSynchronize*를 호출할 최적의 조건은 *OnIdle* 이벤트 핸들러에서 애플리케이션이 유휴 상태일 때입니다. 이렇게 하면 백그라운드 스레드에서 메소드를 호출하는 것이 안전합니다.

## 스레드 로컬 변수 사용

*Execute* 메소드와 이 메소드가 호출하는 모든 루틴은 오브젝트 파스칼 루틴과 마찬가지로 자신의 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 이와 같이 수행할 수 있습니다. **threadvar** 섹션에 변수를 선언하여 변수를 스레드 로컬로 만듭니다. 예를 들면, 다음과 같습니다.

```
threadvar
  x :integer;
```

애플리케이션의 각 스레드에 개별적이지만 각 스레드 내에서는 전역인 정수형 변수를 선언합니다.

threadvar 섹션은 전역 변수에서만 사용할 수 있습니다. Pointer 및 Function 변수는 스레드 변수가 될 수 없습니다. 긴 문자열 같은 copy-on-write 구문을 사용하는 타입도 스레드 변수로 사용할 수 없습니다.

## 다른 스레드로 종료 확인

스레드는 *Execute* 메소드가 호출될 때 실행을 시작하고(9-10 페이지의 "스레드 객체 실행" 참조) *Execute*이 완료될 때까지 계속됩니다. 이는 스레드가 특정 작업을 수행한 다음 완료되면 중지한다는 모델을 보여줍니다. 그러나 경우에 따라서 어떤 애플리케이션은 일부 외부 기준을 충족할 때까지 실행되는 스레드를 필요로 합니다.

*Terminated* 속성을 확인함으로써 스레드가 실행을 완료해야 할 시간이라는 것을 다른 스레드를 통해 나타낼 수 있습니다. 다른 스레드는 사용자의 스레드를 종료시키려고 할 때 *Terminate* 메소드를 호출합니다. *Terminate*는 사용자 스레드의 *Terminated* 속성을 *True*로 설정합니다. *Terminated* 속성 확인 및 *Terminated* 속성에 대한 응답을 수행하여 *Terminate* 메소드를 구현하는 것은 *Execute* 메소드에 달려 있습니다. 다음 예제는 이러한 방법 중 하나를 보여줍니다.

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

## 스레드 함수에서의 예외 처리

*Execute* 메소드는 스레드에서 발생하는 모든 예외를 감지해야 합니다. 스레드 함수에서 예외를 감지하지 못할 경우 애플리케이션에서 액세스 위반이 발생할 수 있습니다. 애플리케이션을 개발할 때 IDE가 예외를 감지하므로 위의 설명이 명확하지 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...except** 블록을 다음과 같은 방법으로 추가합니다.

```
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
      PerformSomeTask;
    except
      { do something with exceptions }
    end;
  end;
end;
```

## 지우기 코드 작성

스레드 실행이 완료되었을 때 이를 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 실행되게 하십시오.

*OnTerminate* 이벤트 핸들러는 스레드에서 부분적으로는 실행되지 않습니다. 대신 애플리케이션에 있는 메인 CLX 스레드의 컨텍스트에서 실행됩니다. 이것은 다음과 같은 두 가지 의미를 가집니다.

- 메인 CLX 스레드 값을 원하지 않는 한 *OnTerminate* 이벤트 핸들러에서는 스레드 로컬 변수를 사용할 수 없습니다.
- 다른 스레드와의 충돌에 대한 걱정없이 *OnTerminate* 이벤트 핸들러에서 모든 컴포넌트와 CLX 객체에 액세스할 수 있습니다.

메인 CLX 스레드에 대한 자세한 내용은 9-3 페이지의 "메인 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 주의할 기울여야 합니다. 뿐만 아니라 하나의 스레드 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 완료할 때까지 다른 스레드의 실행을 막아야 합니다. 이 때 실행 중인 다른 스레드를 불필요하게 막지 않도록 주의해야 합니다. 그렇게 할 경우 성능이 심하게 저하되어 다중 스레드를 사용하여 얻을 수 있는 대부분의 이점을 얻지 못할 수 있습니다.

### 객체 잠금

일부 객체에는 다른 스레드가 실행되어 해당 객체 인스턴스를 사용하지 못하게 하는 잠금 기능이 기본 제공되어 있습니다.

예를 들어, *TCanvas*와 자손 같은 캔버스 객체에는 *Unlock* 메소드가 호출되기 전까지 다른 스레드가 캔버스에 액세스하지 못하도록 막아주는 *Lock* 메소드가 있습니다.

CLX에는 스레드에 대해 안전한 목록 객체인 *TThreadList*도 들어 있습니다. *TThreadList.LockList*를 호출하면 *UnlockList* 메소드가 호출될 때까지 실행 중인 다른 스레드가 목록을 사용하지 못하도록 막으면서 목록 객체를 반환합니다. *TCanvas.Lock* 이나 *TThreadList.LockList*를 호출하면 안전하게 중첩됩니다. 마지막 잠금 호출이 동일한 스레드 내의 해당 잠금 해제 호출과 일치해야만 이 잠금이 해제됩니다.

### 임계 구역(Critical sections) 사용

객체가 기본 제공된 잠금 기능을 제공하지 않으면 임계 구역을 사용할 수 있습니다. 임계 구역은 한 번에 하나의 스레드만 들어갈 수 있게 하는 문과 같습니다. 임계 구역을 사용하려면 *TCriticalSection*의 전역 인스턴스를 작성합니다. *TCriticalSection*에는 다른 스레드가 구역을 실행하지 못하게 막는 *Acquire*와 블록을 제거하는 *Release*라는 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. 완료되면 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 *LockXY*를 갖고 있는 애플리케이션의 경우를 고려해 보십시오. X나 Y를 사용하는 모든 스레드는 다음과 같은 임계 구역에 대한 호출과 함께 사용하는 경우처럼 예외해야 합니다.

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

## 다중 읽기 배타적 쓰기 동기화 장치 (Multi-read exclusive-write synchronizer) 사용

임계 구역을 사용하여 전역 메모리를 보호할 경우 한 번에 단 하나의 스레드만 해당 메모리를 사용할 수 있습니다. 특히 가끔씩 읽어야 하지만 거의 쓰지 않는 객체나 변수가 있는 경우에는 필요 이상의 보호가 될 수도 있습니다. 그렇지만 다른 스레드가 메모리에 쓰지 않는 한 동일한 메모리를 동시에 읽는 다중 스레드에는 위험은 없습니다.

갖고 있는 일부 전역 메모리가 가끔 읽히지만 스레드에서 간혹 쓰기도 할 경우에는 *TMultiReadExclusiveWriteSynchronizer*를 사용하여 보호할 수 있습니다. 이 객체는 임계 구역처럼 동작하지만 다른 스레드가 쓰고 있지만 않는다면 다중 스레드를 통해 자신이 보호하는 메모리를 읽을 수 있습니다. 스레드는 *TMultiReadExclusiveWriteSynchronizer*에서 보호하는 메모리를 배타적으로 액세스 할 수 있어야 합니다.

다중 읽기 배타적 쓰기 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 완료하면 *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 완료하면 메모리를 읽기 위해 기다린 스레드가 시작할 수 있도록 *EndWrite* 메소드를 호출합니다.

**경고** 다중 읽기 배타적 쓰기 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 이용되는 그 밖의 기술

CLX에서 객체를 사용하는 경우에는 메인 CLX 스레드를 사용하여 코드를 실행합니다. 메인 CLX 스레드를 사용하면 객체가 다른 스레드의 CLX 객체에서 사용하는 모든 메모리에 간접적으로 액세스하지 못하게 합니다. 메인 CLX 스레드에 대한 자세한 내용은 9-3 페이지의 "메인 CLX 스레드 사용"을 참조하십시오.

전역 메모리를 다중 스레드로 공유할 필요가 없을 경우 전역 변수 대신 스레드 로컬 변수 사용을 고려해 보십시오. 스레드 로컬 변수를 사용하면 사용자의 스레드가 다른 스레드를 기다리거나 잠글 필요가 없습니다. 스레드 로컬 변수에 대한 자세한 내용은 9-5 페이지의 "스레드 로컬 변수 사용"을 참조하십시오.



## 다른 스레드 기다리기

사용자의 스레드가 다른 스레드의 작업이 완료되기를 기다려야 하는 경우 스레드에게 일시적으로 실행을 중지하도록 알릴 수 있습니다. 다른 스레드가 실행을 완전히 완료할 때까지 기다리거나 다른 스레드가 작업 완료를 신호할 때까지 기다릴 수 있습니다.

### 스레드의 실행 완료 대기

다른 스레드가 실행 완료되기를 기다리려면 해당 다른 스레드의 *WaitFor* 메소드를 사용합니다. *WaitFor*은 자신의 *Execute* 메소드를 완료하거나 예외로 인한 종료를 통해 다른 스레드가 종료되면 반환됩니다. 예를 들어, 다음 코드는 목록에 있는 객체에 액세스하기 전에 스레드 목록 객체를 다른 스레드가 채울 때까지 기다립니다.

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
    end;
    ThreadList1.UnlockList;
  end;
end;
```

이전의 예제에서 보면 *WaitFor* 메소드가 목록이 성공적으로 채워졌음을 표시할 때만 목록 항목을 액세스할 수 있었습니다. 이 반환값은 기다린 스레드의 *Execute* 메소드에 의해 할당되어야 합니다. 그러나 *WaitFor*을 호출하는 스레드가 *Execute*를 호출하는 코드가 아닌 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 그 의미를 결정합니다.

### 작업 완료 대기

간혹 특정 스레드의 실행 완료를 기다리는 대신 스레드의 일부 작업이 완료되는 것을 기다려야 할 경우도 있습니다. 그렇게 하려면 이벤트 객체를 사용합니다. 이벤트 객체(*TEvent*)는 모든 스레드에 보이는 신호처럼 동작할 수 있도록 전역 유효 범위(scope)로 만들어야 합니다.

어떤 스레드에서 다른 스레드가 의존하는 작업을 완료할 때에는 *TEvent.SetEvent*를 호출합니다. *SetEvent*는 신호를 켜서 검사하는 다른 스레드에 작업이 완료되었음을 알려줍니다. 신호를 끄려면 *ResetEvent* 메소드를 사용합니다.

예를 들어, 단일 스레드가 아닌 여러 스레드가 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 증가시킬 수 있으며 이벤트를 설정하여 마지막 스레드가 모든 작업이 완료되었다는 신호를 보내게 할 수 있습니다.

다음 코드는 완료되어야 하는 모든 스레드의 마지막 *OnTerminate* 이벤트 핸들러를 보여줍니다. *CounterGuard*는 다중 스레드가 동시에 카운터를 사용하지 못하도록 막는 임계 구역 전역 객체입니다. *Counter*는 완료된 스레드의 수를 계산하는 전역 변수입니다.

```

procedure TDataModule.TaskThreadTerminate(Sender:TObject);
begin
  ...
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter); { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
    CounterGuard.Release; { release the lock on the counter }
  ...
end;

```

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 9.1의 값 중 하나를 반환합니다.

**표 9.1** 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 실행하는 방법과 모두 완료되었을 때 다시 시작하는 방법을 보여줍니다.

```

Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ now continue with the main thread. All task threads have finished }

```

**참고** 지정된 시간 후 이벤트 대기 중지를 원하지 않는 경우 INFINITE의 매개변수 값인 *WaitFor* 메소드를 전달합니다. INFINITE를 사용할 때에는 예상되는 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 시작되도록 정지된 상태에서 스레드를 작성할 수 있습니다. 즉시 시작하도록 스레드를 작성하려면 생성자의 *CreateSuspended* 매개변수를 *False*로 설정합니다. 예를 들어, 다음 줄을 작성하여 스레드를 작성하고 실행을 시작합니다.

```

SecondProcess := TMyThread.Create(false); {create and run the thread }

```

**경고** 애플리케이션에 너무 많은 스레드를 작성하지 마십시오. 다중 스레드를 관리하는 오버헤드의 성능에 영향을 미칠 수 있습니다. 단일 프로세서 시스템의 프로세스당 16개의 스레드까지만 사용할 것을 권장합니다. 이러한 제한은 대부분의 스레드가 외부 이벤트를 기다리는 것을 가정한 것입니다. 모든 스레드가 활성화되었을 경우에는 더 적게 사용합니다.

동일한 스레드 형식의 다중 인스턴스를 작성하여 병렬 코드를 실행할 수 있습니다. 예를 들어, 각 스레드에서 예상된 응답을 수행하게 하여 일부 사용자 동작에 대한 응답으로 새 스레드 인스턴스를 실행할 수 있습니다.

## 기본 우선 순위 오버라이드

---

스레드가 수신해야 하는 CPU 시간의 양이 스레드 작업에 암시된 경우 생성자에 해당 우선 순위를 설정해 둡니다. 이 내용은 9-2 페이지의 "스레드 초기화"에 설명되어 있습니다. 그러나 스레드 실행 시기에 따라 스레드 우선 순위가 달라지는 경우 정지된 상태에서 스레드를 작성하고 우선 순위를 설정한 다음 다음과 같은 방법으로 스레드 실행을 시작합니다.

```
SecondProcess := TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.Resume; { now run the thread }
```

## 스레드 시작 및 중지

---

스레드는 실행을 완료하기 전에 얼마든지 시작되고 중지될 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 중지가 *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 *Execute*는 *Terminated* 속성을 정기적으로 확인하고 *Terminated*가 *True*일 때 실행을 중지합니다.

## 다중 스레드 애플리케이션 디버깅

---

다중 스레드 애플리케이션 디버깅할 경우 동시에 실행되는 모든 스레드의 상태를 추적하거나 심지어 브레이크포인트에서 중지할 때 어떤 스레드가 실행 중인지 확인하는 것은 혼돈될 수 있습니다. Thread Status 상자를 사용하면 애플리케이션의 모든 스레드를 추적하고 처리할 수 있습니다. Thread Status 상자를 표시하려면 메인 메뉴에서 View | Threads를 선택합니다.

브레이크포인트, 예외, 일시 정지 등의 디버그 이벤트가 발생하면 Thread Status 보기에서 각 스레드의 상태를 나타냅니다. Thread Status 상자를 마우스 오른쪽 버튼으로 클릭하여 해당 소스 위치를 찾는 명령에 액세스하거나 다른 스레드를 현재 사용하는 것으로 만듭니다. 스레드가 현재 사용 중인 것으로 표시되면 해당 스레드에 따라 다음 단계나 실행 작업이 결정됩니다.

Thread Status 상자는 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*에 대한 호출에 의해 반환됩니다.

Thread Status 상자에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

## 크로스 플랫폼 애플리케이션 개발

Linux 및 Windows 운영 체제 모두에서 실행되는 크로스 플랫폼 32비트 애플리케이션을 개발할 수 있습니다. 기존 Windows 애플리케이션을 수정하거나 권장하는 플랫폼 독립적인 코드 작성 방법을 따라 새 애플리케이션을 만들 수 있습니다.

이 장에서는 Kylix Windows 애플리케이션을 Linux로 포팅하는 방법을 설명하고 Windows와 Linux에서 애플리케이션 개발의 차이에 관한 정보도 제공합니다. 또한 서로 다른 두 환경 사이에서 이식 가능한 코드 작성에 대한 지침도 제공합니다.

**참고** Linux용으로 개발된 대부분의 Linux 특정 API 호출이 없는 애플리케이션은 Linux에서 실행되며 다시 컴파일할 경우 CLX가 사용 가능할 때 Windows에서 실행할 수 있습니다.

### Windows 애플리케이션을 Linux로 포팅

Windows 환경용으로 개발된 Kylix 애플리케이션이 있는 경우 이 애플리케이션을 Linux 환경으로 포팅할 수 있습니다. 포팅의 난이도는 애플리케이션의 특성 및 복잡성, Windows 종속 정도에 따라 다릅니다.

다음 단원에서는 Windows와 Linux 환경 간의 주요한 차이점 몇 가지를 설명하고 애플리케이션 포팅 입문에 대한 지침을 제공합니다.

### 포팅 기법

다음은 애플리케이션을 한 플랫폼에서 다른 플랫폼으로 포팅하기 위해 이용할 수 있는 여러 가지 방법들입니다.

**표 10.1** 포팅 기법

| 기술            | 설명                             |
|---------------|--------------------------------|
| 플랫폼 특정 포팅     | 운영 체제와 기본으로 사용한 API를 대상으로 함    |
| 크로스 플랫폼 포팅    | 크로스 플랫폼 API를 대상으로 함            |
| Windows 에뮬레이션 | 코드는 그대로 놔두고 코드에서 사용하는 API를 포팅함 |

## 플랫폼 특정 포팅

플랫폼 특정 포팅은 시간이 걸리고, 비용이 많이 들며 주로 해당 플랫폼에서만 애플리케이션을 사용할 수 있습니다. 플랫폼 특정 포팅은 특히 유지 관리가 어려운 다른 코드를 기준으로 만듭니다. 하지만 각 포팅은 특정 운영 체제를 위해 설계된 것으로 플랫폼 특정 기능을 활용할 수 있습니다. 따라서 애플리케이션이 대체로 보다 빠르게 실행됩니다.

## 크로스 플랫폼 포팅

크로스 플랫폼 포팅은 일반적으로 가장 최신 기법을 제공하며 포팅된 애플리케이션은 다중 플랫폼을 대상으로 합니다. 실제로 크로스 플랫폼 애플리케이션 개발에 포함되는 작업의 양은 기존 코드에 따라 결정됩니다. 코드가 플랫폼 독립성을 염두에 두지 않고 개발되었다면 플랫폼 독립적인 "로직"과 플랫폼 독립적인 "구현"이 같이 섞이는 경우가 있을 수 있습니다.

크로스 플랫폼 접근법은 비즈니스 로직이 플랫폼 독립적인 관점에서 개발되므로 더 선호됩니다. 일부 서비스는 모든 플랫폼에서 같아 보이지만 개별적으로는 특정한 구현을 가진 내부 인터페이스를 통해 추출됩니다. 그 한 가지 예가 Kylix의 런타임 라이브러리입니다. 두 플랫폼에서 인터페이스가 매우 유사하지만 구현 내용은 크게 다릅니다. 크로스 플랫폼 부분을 분리한 후 그 위에 특정 서비스를 구현해야 합니다. 결국 이 접근법은 폭 넓게 공유되는 소스 기반과 개선된 애플리케이션 아키텍처로 인한 유지 비용 감소를 통해 가장 비용이 적게 드는 솔루션입니다.

## Windows 에뮬레이션 포팅

Windows 에뮬레이션은 가장 복잡한 방법이며 비용이 많이 들 수 있지만 결과물인 Linux 애플리케이션이 기존 Windows 애플리케이션과는 가장 유사하게 보입니다. 이 접근법은 Linux에 Windows 기능을 구현하는 것을 포함합니다. 엔지니어링의 관점에서 볼 때 이 솔루션은 유지 관리가 가장 어렵습니다.

## 애플리케이션 포팅

Linux에서만 실행시키기 위해 애플리케이션을 Linux로 포팅하는 경우 Windows 특정 기능 전체를 제거할 수 있습니다. 하지만 두 플랫폼 모두에서 실행하기 위해 포팅하는 경우 코드를 수정하거나 **\$IFDEF**를 사용하여 Windows 또는 Linux에 특별히 적용하는 코드의 섹션을 표시합니다.

Windows 애플리케이션을 Linux로 포팅하려면 다음과 같은 일반적인 단계를 따르십시오.

- 1 Kylix Windows 애플리케이션 소스 파일과 기타 프로젝트 관련 파일을 Linux 컴퓨터로 옮깁니다. (프로그램을 두 플랫폼 모두에서 실행하려는 경우 Linux와 Windows 간에 소스 파일을 공유할 수 있습니다. 또는 ASCII 모드를 사용하는 ftp 같은 툴을 사용하여 파일을 전송할 수도 있습니다.)

소스 파일에는 유닛 파일(.pas 파일), 프로젝트 파일(.dpr 파일) 및 모든 패키지 파일(.dpk 파일)이 포함됩니다. 프로젝트 관련 파일에는 폼 파일(.dfm 파일), 리소스 파일(.res 파일) 및 프로젝트 옵션 파일(.dof 파일)이 포함됩니다. IDE를 사용하지 않고 명령줄에서만 애플리케이션을 컴파일하려면 구성 파일(.cfg 파일)이 필요합니다.

- 2 단일 소스 애플리케이션을 Windows와 Linux 모두에서 사용하려는 경우 .dfm 파일을 동일한 이름의 .xfm 파일로 복사합니다 (예를 들어 unit1.dfm을 unit1.xfm으로 이름 변경). 유닛 파일에서 .dfm 파일에 대한 참조를 {\$R \*.dfm}에서 {\$R \*.xfm}로 이름 변경 (또는 **\$IFDEF**) 합니다. (.dfm 파일은 Kylix에서 동작하지만 Delphi에서는 동작하지 않도록 변경됩니다.)
- 3 모든 **uses** 절을 변경 (또는 **\$IFDEF**) 하여 Kylix의 정확한 유닛을 참조하게 합니다. (자세한 내용은 10-7 페이지의 "Kylix와 Delphi 유닛 비교" 참조)
- 4 Windows 종속성을 필요로 하지 않는 모든 코드를 다시 작성하여 플랫폼 독립적으로 만듭니다. 런타임 라이브러리 루틴과 상수를 사용하여 이 작업을 수행합니다. (자세한 내용은 10-15 페이지의 "포팅 가능한 코드 작성" 참조)
- 5 Linux에서 다른 기능에 대해 동등한 기능을 찾습니다. **\$IFDEF**를 가능한 적게 사용하여 Windows 특정 정보를 구분하는 데 합니다. (자세한 내용은 10-16 페이지의 "조건 지시어 사용" 참조)

예를 들어 소스 파일에서 다음과 같이 플랫폼 특정 코드를 **\$IFDEF**할 수 있습니다.

```
[IFDEF MSWINDOWS]
IniFile.LoadFromFile('c:\x.txt');
[ENDIF]

[IFDEF LINUX]
IniFile.LoadFromFile('/home/name/x.txt');
[ENDIF]
```

- 6 모든 프로젝트 파일에서 경로에 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.
  - 다중 경로를 지정하는 곳에서 경로 분리자를 세미콜론 (;)에서 콜론 (:)으로 변경합니다. SysUtils에서 PathSep 상수를 사용하여 시스템에 맞는 경로 분리자를 지정합니다.
  - Linux에서는 파일 이름의 대소문자를 구별하므로 애플리케이션이 파일 이름의 대소문자를 변경하거나 대소문자 중 하나인 것으로 가정하지 않았는지 확인합니다.
- 7 Linux에서 프로젝트를 컴파일합니다. 추가로 변경해야 할 곳이 없는지를 알아보기 위해서 모든 오류 메시지를 검토합니다.

## CLX와 VCL 비교

Kylix는 Visual Component Library (VCL) 대신 Cross Platform (CLX) 용 Borland Component Library를 사용합니다. VCL 내에서 많은 컨트롤이 Windows 컨트롤에 액세스하기 위한 쉬운 방법을 제공합니다. 이와 마찬가지로 CLX는 Qt 공유 라이브러리에서 Qt widget (window와 gadget을 합한 것)에 대한 액세스를 제공합니다.

CLX는 VCL과 매우 유사합니다. 대부분의 컴포넌트 이름이 동일하며 많은 속성들이 동일한 이름을 가집니다. 또한 VCL 뿐만 아니라 CLX도 Windows에서 사용 가능하게 될 것입니다(사용 가능한 지 알아보려면 Delphi의 최신 버전을 확인합니다).

CLX 컴포넌트는 다음과 같이 그룹화될 수 있습니다.

**표 10.2 CLX 부분**

| 부분        | 설명                                                                                                        |
|-----------|-----------------------------------------------------------------------------------------------------------|
| VisualCLX | 원시 크로스 플랫폼 GUI 컴포넌트와 그래픽. 이 부분의 컴포넌트는 Linux와 Windows에서 서로 다를 수 있습니다.                                      |
| DataCLX   | 클라이언트 데이터 액세스 컴포넌트. 이 부분의 컴포넌트는 클라이언트 데이터셋에 기반한 로컬, 클라이언트/서버 및 n계층의 하위 집합입니다. 코드는 Linux와 Windows에서 동일합니다. |
| NetCLX    | Apache DSO 및 CGI WebBroker를 포함하는 인터넷 컴포넌트. Linux와 Windows에서 동일합니다.                                        |
| BaseCLX   | 구현되고 Classes.pas를 포함하는 런타임 라이브러리. 코드는 Linux와 Windows에서 동일합니다.                                             |

VisualCLX의 widget은 Windows 컨트롤을 대체합니다. CLX에서 *TWidgetControl*은 VCL의 *TWinControl*을 대체합니다. *TScrollingWidget* 같은 다른 컴포넌트는 해당 이름을 가집니다. 하지만 *TWinControl*을 *TWidgetControl*로 변경할 필요는 없습니다. 예를 들어 다음과 같은 타입 선언의 경우

```
TWinControl = TWidgetControl;
```

QControls.pas 소스 파일에 사용하여 소스 코드의 공유를 간소화합니다. *TWidgetControl*과 해당 자손들은 모두 Qt 객체의 참조인 *Handle* 속성과 이벤트 메커니즘을 처리하는 혹은 객체의 참조인 *Hooks* 속성을 갖습니다.

CLX의 경우에는 유닛 이름과 일부 클래스의 위치가 다릅니다. Kylix에 존재하지 않는 유닛에 대한 참조를 제거하고 Kylix 유닛에 대한 이름을 변경하려면 **uses** 절을 수정해야 합니다. (프로젝트 파일과 대부분 유닛의 인터페이스 섹션에는 대부분 **uses** 절이 있습니다. 유닛의 구현 섹션도 자체 **uses** 절을 가집니다.)

## CLX가 다르게 하는 동작

많은 CLX가 VCL과 일관되도록 구현되지만 일부 기능은 다르게 구현됩니다. 이 단원에서 CLX와 VCL 구현 간의 일부 차이점에 대한 개요를 제공합니다.

### 룩앤필 (Look and feel)

Linux의 비주얼 환경은 Windows와 약간 다르게 보입니다. 대화 상자의 모양이 어떤 윈도우 관리자(예를 들어, KDE나 Gnome)를 사용하는가에 따라 다를 수 있습니다.



## 스타일

애플리케이션 전반의 "스타일"을 *OwnerDraw* 속성에 부가하여 사용할 수 있습니다. *TApplication.Style* 속성을 사용하여 애플리케이션 그래픽 요소의 록엔필을 지정할 수 있습니다. 스타일을 사용하여 widget이나 애플리케이션이 완전히 새로운 모습을 갖출 수 있습니다. Linux에서 *owner-draw*를 사용할 수 있지만 스타일 사용을 권합니다.

## 가변 타입

시스템에 있던 모든 가변 타입 배열/*SafeArray* 코드는 이제는 다음 두 개의 새로운 유닛입니다.

- *Variants.pas*
- *VarUtils.pas*

운영 체제 의존 코드는 이제 *VarUtils.pas*에 분리되어 있고 *Variants.pas*가 필요로 하는 모든 일반 버전을 포함합니다. Windows로부터 Windows 호출을 포함하는 코드를 포팅하는 경우 이 호출을 *VarUtils.pas* 호출로 대체해야 합니다.

가변 타입을 사용하려면 **uses** 절에서 *Variants* 유닛을 포함해야 합니다.

*VarIsEmpty*는 *varEmpty*에 대해 간단한 검사를 하여 가변이 지워졌는지 확인하며 Linux에서는 *VarIsClear* 함수를 사용하여 가변을 지웁니다.

## 사용자 지정 가변 타입 데이터 핸들러

가변에 대해 사용자 지정 데이터 타입을 정의할 수 있습니다. 타입이 가변으로 할당될 때 연산자가 오버로드되게 합니다. 새 가변 타입을 만들려면 *TCustomVariantType* 클래스의 자손으로 새로운 가변 타입을 인스턴스화합니다.

예를 들어, *VarCmplx.pas*를 보면 이 유닛은 사용자 지정 가변 타입을 통해 복잡한 수학적 지원을 구현합니다. 이 유닛은 더하기, 빼기, 곱하기, 나누기(정수 나누기는 아님), 부정 같은 가변 연산을 지원합니다. 또한 *SmallInt*, 정수, *Single*, *Double*, 통화, 날짜, 부울, 바이트, *OleStr* 및 문자열의 변환을 처리합니다. 모든 실수 타입/순서 타입 변환에서는 복잡한 값의 허수 부분을 버립니다.

## 레지스트리 사용 안함

Linux에서는 구성 정보를 저장하기 위한 레지스트리는 사용하지 않습니다. 레지스트리 대신 텍스트 구성 파일과 환경 변수를 사용합니다. Linux의 시스템 구성 파일이 가끔 */etc/hosts*처럼 */etc*에 위치하기도 합니다. 다른 사용자 프로파일은 *bash* 셸 설정이나 X 프로그램의 기본값을 설정하는 데 사용하는 *.Xdefaults*를 보유하는 점(.)이 앞에 오는 숨김 파일(예: *.bashrc*)에 위치합니다.

레지스트리 의존 코드는 애플리케이션과 동일한 디렉토리에 저장하는 대신에 로컬 구성 텍스트 파일을 사용하여 변경할 수 있습니다. 로컬 구성 파일로 모든 결과를 내보내는 것을 제외하고 모든 레지스트리 함수로 유닛을 작성하는 것이 레지스트리에 대한 이전의 의존성을 처리하는 한 방법입니다.

전역 위치에 정보를 입력하기 위해 루트 디렉토리에 전역 구성 파일을 저장할 수 있습니다. 이는 사용자의 애플리케이션이 동일한 구성 파일에 액세스할 수 있게 합니다. 하지만 파일 권한과 액세스 권한이 올바르게 설정되었는지 확인해야 합니다.

Windows에서는 ini 파일도 사용할 수 있습니다. 그러나 Kylix에서는 *TRegIniFile* 대신 *TMemIniFile*을 사용해야 합니다.

## 그 밖의 차이점

Kylix 구현에서는 애플리케이션이 동작하는 방법에 영향을 끼치는 일부 다른 차이점도 있습니다. 이 단원에서는 이러한 차이점에 대해 설명합니다.

*ToggleButton*은 Enter 키로 토글되지 않습니다. Enter 키를 눌러도 Kylix에서는 Delphi에서처럼 클릭 이벤트가 발생하지 않습니다.

*TColorDialog*는 설정할 *TColorDialog.Options* 속성이 없습니다. 따라서 색상 대화 상자의 모양 및 기능을 사용자 지정할 수 없습니다. 또한 *TColordialog*가 항상 모달은 아닙니다. Kylix에서 애플리케이션의 제목 표시줄을 모달 대화 상자로 조작할 수 있습니다(즉, 색상 대화 상자의 부모 폼을 선택하고 색상 대화 상자가 열려있는 동안 최대화하는 등의 작업을 할 수 있습니다).

런타임에 콤보 상자는 Delphi에서와 다르게 Kylix에서 동작합니다. Kylix (Delphi는 아님)에서, 콤보 상자의 편집 필드에 텍스트를 입력하고 Enter 키를 눌러서 드롭다운에 항목을 추가할 수 있습니다. *InsertMode*를 *ciNone*으로 설정하여 이 기능을 끌 수 있습니다. 콤보 상자의 목록에 빈(문자열이 아닌) 항목을 추가할 수도 있습니다. 또한 아래쪽 화살표를 계속 누르면, 콤보 상자 목록의 마지막 항목에서 계속 스크롤됩니다. 위에서부터 다시 반복됩니다.

*TCustomEdit*은 *Undo*, *ClearUndo* 또는 *CanUndo*를 구현하지 않습니다. 따라서 프로그램에서 편집을 취소할 방법이 없습니다. 그러나 애플리케이션 사용자는 런타임에 편집 상자를 마우스 오른쪽 단추로 클릭하고 Undo 명령을 선택하여 편집 상자(*TEdit*)의 편집 내용을 취소할 수 있습니다.

Windows에서 Enter 키에 대한 *OnKeyDown* 이벤트나 *KeyUp* 이벤트의 키 값은 13입니다. Linux에서 이 값은 4100입니다. Enter 키에 대한 13과 같은 키에 대해 하드코딩된 숫자 값을 검사하는 경우에는 Delphi 애플리케이션을 Kylix로 포팅할 때 이 값을 변경해야 합니다.

## CLX에서 빠진 내용

---

다음과 같은 일반적인 기능이 CLX에 빠져 있습니다.

- 오른쪽에서 왼쪽 텍스트 입력이나 출력을 위한 양방향 속성 (*BidiMode*)
- 공용 컨트롤의 일반 베벨 속성(일부 객체에는 아직도 베벨 속성이 있음)
- 도킹 속성 및 메소드
- Win3.1 탭 및 *Ctl3D* 같은 컴포넌트에 대한 역호환 기능
- *DragCursor*와 *DragKind*(끌어서 놓기는 포함되지 않음)
- 메뉴 항목에 대한 힌트

## 포팅되지 않는 기능

Windows 버전의 Kylix에서 지원하는 일부 Windows 특정 기능은 Linux 환경으로 직접 이전되지 않습니다. COM, ActiveX, OLE, BDE 및 ADO 같은 기능은 Windows 기술에 종속되므로 Kylix에서는 사용할 수 없습니다. 다음 표는 두 플랫폼 간에 다르게 나타나는 기능들과, Kylix에 해당 기능이 있는 경우 그 기능들을 나열한 것입니다.

표 10.3 변경되었거나 다른 기능

| Delphi/Windows 기능                                                                  | Kylix/Linux 기능                                 |
|------------------------------------------------------------------------------------|------------------------------------------------|
| Windows API 호출                                                                     | CLX 메소드, Qt 호출, libc 호출 또는 다른 시스템 라이브러리에 대한 호출 |
| COM 컴포넌트(ActiveX 포함)                                                               | 사용 불가                                          |
| ADO 컴포넌트                                                                           | 일반 데이터베이스 컴포넌트 사용 가능                           |
| Windows 메시징                                                                        | Qt 이벤트                                         |
| Winsock                                                                            | BSD 소켓                                         |
| MAPI (Messaging Application Programming Interface)에는 Windows 메시징 기능의 표준 라이브러리가 있음. | SMTP/POP3로 전자 우편 메시지를 보내고 받고 저장할 수 있음          |
| 레거시 컴포넌트 (예: Win 3.1 컴포넌트 팔레트 탭의 항목)                                               | 사용 불가                                          |

Kylix에서 Windows DLL에 해당하는 것은 공유 객체 라이브러리(.so 파일)로서 위치 독립 코드(PIC)를 포함합니다. 다음과 같은 결과가 나타납니다.

- 메모리의 절대 주소를 참조하는 변수(**absolute** 지시어 사용)는 허용되지 않습니다.
- 외부 함수에 대한 전역 메모리 참조와 호출은 호출 간에 보존되어야 하는 EBX 레지스터에 상대적으로 만들어집니다.

Kylix는 정확한 코드를 생성하므로 어셈블리를 사용할 경우 외부 함수에 대한 전역 메모리 참조와 호출만 신경쓰면 됩니다. 자세한 내용은 10-18 페이지의 "인라인 어셈블러 코드 포함"을 참조하십시오.

Kylix 라이브러리 모듈과 패키지는 .so 파일을 사용하여 구현합니다.

## Kylix와 Delphi 유닛 비교

VCL 또는 CLX의 모든 객체는 유닛 파일인 .pas 소스 파일에 정의되어 있습니다. 예를 들어, 시스템 유닛에는 *TObject*의 구현이 있으며 클래스리스 유닛은 기본 *TComponent* 클래스를 정의합니다. 폼에 객체를 갖다 놓거나 애플리케이션 내에서 객체를 사용하면 해당 유닛의 이름이 **uses** 절에 추가되어 컴파일러에게 프로젝트에 어느 유닛을 연결해야 하는지 알려줍니다.

이 단원에는 Kylix와 Delphi에 모두 들어 있는 유닛, Kylix에만 있는 유닛 및 Delphi에만 있는 유닛을 나열한 표가 있습니다. 만약 Kylix에 포함된 유닛이 Delphi의 향후 버전에 나타날 예정일 경우에는 Delphi 유닛 열에 명기되어 있습니다.

다음 표는 Delphi와 Kylix에 모두 들어 있는 유닛들을 각각 나열한 것입니다.

**표 10.4** Kylix 와 Delphi 의 유닛

| Delphi 유닛                      | Kylix 유닛                       |
|--------------------------------|--------------------------------|
| ActnList                       | QActnList                      |
| Buttons                        | QButtons                       |
| CheckLst                       | QCheckLst                      |
| Classes                        | Classes                        |
| Clipbrd                        | QClipbrd                       |
| ComCtrls                       | QComCtrls                      |
| Consts                         | Consts, Qconsts<br>및 RTLConsts |
| Contrns                        | Contrns                        |
| Controls                       | QControls                      |
| ConvUtils (Delphi의<br>향후 버전)   | ConvUtils                      |
| DateUtils (Delphi의<br>향후 버전)   | DateUtils                      |
| DB                             | DB                             |
| DBActns                        | QDBActns                       |
| DBClient                       | DBClient                       |
| DBCommon                       | DBCommon                       |
| DBConnAdmin<br>(Delphi의 향후 버전) | DBConnAdmin                    |
| DBConsts                       | DBConsts                       |
| DBCtrls                        | QDBCtrls                       |
| DBGrids                        | QDBGrids                       |
| DBLocal                        | DBLocal                        |
| DBLocalS                       | DBLocalS                       |
| DBLogDlg                       | DBLogDlg                       |
| DBXpress                       | DBXpress                       |
| Dialogs                        | QDialogs                       |
| DSIntf                         | DSIntf                         |
| ExtCtrls                       | QExtCtrls                      |
| FMTBCD (Delphi의<br>향후 버전)      | FMTBCD                         |
| Forms                          | QForms                         |
| Graphics                       | QGraphics                      |
| Grids                          | QGrids                         |
| HelpIntfs                      | HelpIntfs                      |
| ImgList                        | QImgList                       |
| IniFiles                       | IniFiles                       |
| Mask                           | QMask                          |

**표 10.4** Kylix 와 Delphi 의 유닛 ( 계속 )

| Delphi 유닛                 | Kylix 유닛      |
|---------------------------|---------------|
| MaskUtils                 | MaskUtils     |
| Masks                     | Masks         |
| Math                      | Math          |
| Menus                     | QMenus        |
| Midas                     | Midas         |
| MidConst                  | MidConst      |
| Printers                  | QPrinters     |
| Provider                  | Provider      |
| Qt (Delphi의 향후 버전)        | Qt            |
| Search (Delphi 의 향후 버전)   | QSearch       |
| Sockets (Delphi의 향후 버전)   | Sockets       |
| StdActns                  | QStdActns     |
| StdCtrls                  | QStdCtrls     |
| SqlConst                  | SqlConst      |
| SqlExpr                   | SqlExpr       |
| SqlTimSt                  | SqlTimSt      |
| StdConvs (Delphi 의 향후 버전) | StdConvs      |
| SyncObjs                  | SyncObjs      |
| SysConst                  | SysConst      |
| SysInit                   | SysInit       |
| System                    | System        |
| SysUtils                  | SysUtils      |
| Types                     | Types와 QTypes |
| TypeInfo                  | TypeInfo      |
| VarCmplx                  | VarCmplx      |
| Variants(Delphi의 향후 버전)   | Variants      |
| VarUtils(Delphi의 향후 버전)   | VarUtils      |

다음 유닛은 Kylix에는 있지만 Windows에는 없습니다.

**표 10.5** Kylix 에는 있고 Delphi 에는 없는 유닛

| 유닛     | 설명      |
|--------|---------|
| DirSel | 디렉토리 선택 |
| QStyle | GUI 룩엔필 |

다음 Windows 유닛은 ADO, COM 및 Borland Database Engine 같은 Linux에는 없는 Windows 특정 기능이므로 Kylix에 포함되어 있지 않습니다. 이 유닛들이 포함되지 않은 이유들이 그 옆에 나열되어 있습니다.

**표 10.6 Delphi 에는 있고 Kylix 에는 없는 유닛**

| 유닛          | 포함되지 않은 이유              |
|-------------|-------------------------|
| ADOConst    | ADO 기능 없음               |
| ADODB       | ADO 기능 없음               |
| AppEvnts    | TApplicationEvent 객체 없음 |
| AxCtrls     | COM 기능 없음               |
| BdeConst    | BDE 기능 없음               |
| ComStrs     | COM 기능 없음               |
| CorbaCon    | CORBA 기능 없음             |
| CorbaStd    | CORBA 기능 없음             |
| CorbaVCL    | CORBA 기능 없음             |
| CtlPanel    | Windows 제어판 지원 안함       |
| DataBkr     | 나중에 upsell에 있을 수도 있음.   |
| 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      | ADO 기능 없음               |
| DRTable     | BDE 기능 없음               |
| ExtDlgs     | 그림 대화 상자 없음             |
| FileCtrl    | 폐기됨                     |
| MConnect    | COM 기능 없음               |
| Messages    | Windows 특정 영역           |
| MidasCon    | 폐기됨                     |
| MPlayer     | Windows 특정 미디어 플레이어     |
| Mtsobj      | COM 기능 없음               |
| MtsRdm      | COM 기능 없음               |
| Mtx         | COM 기능 없음               |
| mxConsts    | COM 기능 없음               |
| ObjBrkr     | 나중에 upsell에 있을 수도 있음.   |
| OleConstMay | COM 기능 없음               |
| OleCtnrs    | COM 기능 없음               |
| OleCtrls    | COM 기능 없음               |
| OLEDB       | COM 기능 없음               |
| OleServer   | COM 기능 없음               |

표 10.6 Delphi 에는 있고 Kylix 에는 없는 유닛 ( 계속 )

| 유닛       | 포함되지 않은 이유          |
|----------|---------------------|
| Outline  | 폐기됨                 |
| Registry | Windows 특정 레지스트리 지원 |
| ScktCnst | 소켓에 의해 교체됨          |
| ScktComp | 소켓에 의해 교체됨          |
| SConnect | 지원되지 않는 연결 프로토콜     |
| SvcMgr   | NT 서비스 지원           |
| Tabnotbk | 폐기됨                 |
| Tabs     | 폐기됨                 |
| ToolWin  | 도킹 기능 없음            |
| VCLCom   | COM 기능 없음           |
| WebConst | Windows 특정 상수       |
| Windows  | Windows 특정 가상 키 코드  |

## CLX 객체 생성자의 차이점

폼 디자이너에서 암시적으로 폼에 해당 객체를 넣거나 또는 코드에서 객체의 *Create* 메소드를 명시적으로 사용하여 CLX 객체가 만들면 기본으로 사용하는 연결된 widget의 인스턴스도 생성됩니다. widget의 인스턴스는 이 CLX 객체가 소유합니다. *Free* 메소드를 사용하여 CLX 객체를 삭제하거나 CLX 객체의 부모 컨테이너에 의해 CLX가 자동으로 삭제되는 경우에는 기본으로 사용하는 widget도 삭제됩니다. 이 기능은 Windows 애플리케이션의 VCL에서 볼 수 있는 것과 동일한 유형입니다.

객체의 *Create(AHandle)* 메소드를 사용하여 코드에서 명시적으로 CLX 객체를 만드는 경우에는 생성 중 사용할 기존 Qt widget의 인스턴스가 CLX 객체로 전달됩니다. CLX 객체가 자신에게 전달된 Qt widget을 소유하지 않는다는 사실을 잘 알고 있어야 합니다. 따라서 이러한 방법으로 객체를 만든 후 *Free* 메소드를 호출하면 CLX 객체만 소멸되고 기본으로 사용하는 Qt widget 인스턴스는 소멸되지 않습니다. 이것이 VCL과 다른 점입니다.

일부 CLX 객체는 *OwnHandle* 메소드를 사용하여 기본으로 사용하는 widget의 소유권을 추정해 볼 수 있습니다. *OwnHandle*을 호출한 후 CLX 객체를 삭제하면 기본으로 사용하는 widget도 소멸됩니다.

## 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 공유하여 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 두고 공유를 만들어 Linux 컴퓨터가 액세스할 수 있게 할 수 있습니다.

VCL과 CLX 모두에서 지원하는 객체를 사용하면 Kylix에서 파일을 계속적으로 개발하고 컴파일할 수 있습니다. 완료되면 Linux와 Windows 모두에서 컴파일할 수 있습니다.

폼 파일(Delphi의 .dfm 파일)이 Kylix에서는 .xfm 파일입니다. 코드를 단일 소스로 만들려면 Windows의 .dfm 파일을 Linux의 .xfm 으로 복사하여 둘 다 관리합니다. 그렇지 않으면 .dfm 파일은 Linux에서 수정되어 Windows에서 사용할 수 없게 됩니다. 크로스 플랫폼 애플리케이션을 작성하려는 경우에는 CLX를 지원하는 Delphi 버전에서 .xfm을 사용하면 됩니다.

## Windows와 Linux 간의 환경적 차이점

다음 표는 Windows 환경에서 작업하는 것에 익숙해 있을 경우 알아야 할 Linux에서의 차이점 몇 가지를 나열한 것입니다.

표 10.7 Linux 환경의 차이점

| 차이점           | 설명                                                                                                                                                                                                            |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 파일 이름 대소문자 구별 | Linux에서 대문자는 소문자와 <i>다릅니다</i> . Test.txt 파일은 test.txt와 <i>다릅니다</i> . Linux에서는 파일 이름의 대소문자에 많은 주의를 기울여야 합니다.                                                                                                   |
| 줄 끝 문자        | Windows에서 텍스트 행은 CR/LF(즉 ASCII 13 + ASCII 10)으로 종료되지만 Linux에서는 LF로 종료됩니다. Kylix의 코드 편집기가 이 차이점을 처리하기는 하지만 Windows에서 코드를 import 할 때 이러한 차이점을 인식하고 있어야 합니다.                                                     |
| 파일 끝 문자       | DOS와 Windows에서 #26(Ctrl-Z) 문자값은 해당 문자 뒤에 데이터가 있더라도 텍스트 파일의 끝으로 취급됩니다. Linux에서는 특별한 파일 끝 문자가 없으며 텍스트 데이터는 파일의 끝에서 끝납니다.                                                                                        |
| 배치 파일/셸 스크립트  | Linux에서 .bat 파일에 해당하는 것은 셸 스크립트입니다. 스크립트는 명령어를 포함하는 텍스트 파일로, 저장되어 <code>chmod +x &lt;scriptfile&gt;</code> 명령으로 실행 가능하게 됩니다. 스크립트를 실행하려면 해당 이름을 입력합니다. (스크립트 언어는 Linux에서 사용하는 셸에 따라 다릅니다. Bash를 일반적으로 사용합니다.) |
| 명령 확인         | DOS나 Windows에서는 파일이나 폴더를 삭제하려는 경우 "파일을 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.                                                                                                         |



표 10.7 Linux 환경의 차이점 ( 계속 )

| 차이점       | 설명                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 구성 파일     | <p>Windows에서 구성은 레지스트리나 autoexec.bat 과 같은 파일에서 이루어집니다.</p> <p>Linux에서 구성 파일은 점(.)으로 시작하는 숨김 파일로 만들어집니다. 많은 구성 파일들은 /etc 디렉토리와 사용자의 홈 디렉토리에 있습니다.</p> <p>Linux도 LD_LIBRARY_PATH(라이브러리의 경로를 찾음)와 같은 환경 변수를 사용합니다. 기타 중요한 환경 변수는 다음과 같습니다.</p> <p>HOME 사용자의 홈 디렉토리(/home/sam)</p> <p>TERM 터미널 유형(xterm, vt100, console)</p> <p>SHELL 사용자 셸의 경로(/bin/bash)</p> <p>USER 사용자의 로그인 이름(sfuller)</p> <p>PATH 프로그램을 검색하는 경로</p> <p>이 환경 변수들은 셸이나 .bashrc 같은 rc 파일에 지정되어 있습니다.</p> |
| DLL       | Linux에서는 공유 객체 파일(.so)을 사용합니다. Windows에서 이러한 파일들은 동적 연결 라이브러리(DLL)입니다.                                                                                                                                                                                                                                                                                                                                                                                                  |
| 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 응용 프로그램과 연결하지 않습니다.                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 파일 권한     | <p>Linux에서는 파일(및 디렉토리)에 파일 소유자, 그룹 및 기타에 대해 읽고, 쓰고 실행할 권한이 할당됩니다. 예를 들어, -rwxr-xr-x는 왼쪽에서 오른쪽으로 다음과 같은 의미입니다.</p> <p>-은 파일 형식(- = 일반 파일, d = 디렉토리, l = 링크)이고, rwx는 파일 소유자의 권한(읽기, 쓰기, 실행)이며 r-x는 파일 소유자 그룹의 권한(읽기, 실행)이고 r-x는 다른 모든 사용자의 권한(읽기, 실행)입니다. 루트 사용자(superuser)는 이 권한을 무시할 수 있습니다.</p> <p>애플리케이션이 올바른 사용자에게서 실행되고 필요 파일에 적절한 액세스 권한이 있는지 확인해야 합니다.</p>                                                                                                        |
| Make 유틸리티 | Borland의 make 유틸리티는 Linux 플랫폼에서는 사용할 수 없습니다. 그 대신 Linux에 있는 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 사용하여 플랫폼에 맞는 문자를 지정할 수 있습니다.런타임 라이브러리에서 PathDelim을 참조하십시오.                                                                                                                                                                                                                                                                                                                                                   |

표 10.7 Linux 환경의 차이점 ( 계속 )

| 차이점       | 설명                                                                                                                                                                                                                                                  |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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.<br>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.<br>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 경로명에서 역 슬래시를 사용하는 것과 반대로 Linux 경로명에서는 슬래시를 사용합니다. 첫 번째 슬래시는 루트 디렉토리를 의미합니다.

다음은 Linux에서 일반적으로 사용되는 디렉토리입니다.

표 10.8 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           | 구성 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, 스푼 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.

계층적 파일 시스템의 구성에 대한 자세한 내용 및 *Filesystem Hierarchy Standard*를 읽으려면 [www.pathname.com](http://www.pathname.com)을 참조하십시오.

## 포팅 가능한 코드 작성

두 운영 체제 모두에서 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 포팅할 수 있는 애플리케이션을 만들려면 반드시 다음 사항을 염두에 두어야 합니다.

- 플랫폼 특정 (Win32 또는 Linux) API에 대한 호출을 줄이거나 분리시키고 대신 CLX 메소드를 사용합니다.
- 애플리케이션 내에서 구성되는 Windows 메시징 (PostMessage, SendMessage)을 제거합니다.
- *TRegIniFile* 대신 *TMemIniFile*을 사용합니다.
- 파일과 디렉토리 이름의 대소문자 구분에 유의합니다.
- 모든 외부 어셈블러 TASM 코드를 포팅합니다. GNU 어셈블러 "as"는 TASM 구문을 지원하지 않습니다. (10-18 페이지의 "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 do
begin
  if p^ in LeadBytes then
    inc(p);
  inc(p);
end;
```

위의 코드를 플랫폼 독립적인 코드로 대체할 수 있습니다.

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

이 예는 플랫폼 포팅이 가능하고 2바이트보다 긴 멀티바이트 문자를 지원하지만 멀티바이트가 아닌 로케일에 대해서는 프로시저 호출의 성능 비용을 고려하지 않습니다.

런타임 라이브러리 함수를 사용해도 해결되지 않으면 플랫폼 특정 코드를 루틴에 하나로 넣어 분리하거나 서브루틴으로 넣습니다. 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. 하지만 **\$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 "이 코드를 **\$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 때 계산됩니다. C/C++와 달리 Kylix는 프로젝트를 컴파일하는 데 유닛 소스(헤더 파일)를 필요로 하지 않습니다. 모든 소스 코드를 완전히 다시 빌드하는 것은 대부분의 Kylix 프로젝트에서 일반적인 작업이 아닙니다.
- 패키지 파일(.dpk)에서는 **\$IFDEF**를 사용하지 마십시오. 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** MSWINDOWS를 사용하여 WIN32를 포함한 Windows 플랫폼에 대해 검사합니다. **\$IFDEF** WIN32는 32비트와 64비트 Windows 같은 특정 Windows 플랫폼 간의 구분을 위해 사용합니다. WIN64에서 사용하지 않는 것이 확실하지 않다면 코드를 WIN32로 국한시키지 마십시오.
- **\$IFDEF**와 같은 부정 검사는 반드시 필요한 경우 외에는 하지 않습니다. **\$IFDEF** LINUX는 **\$IFDEF** MSWINDOWS와 같지 않습니다.
- **\$IFDEF/\$ELSE** 조합을 피하십시오. 그 대신 가독성을 높이려면 긍정 검사(**\$IFDEF**)를 사용합니다.
- 플랫폼에 따라 다른 **\$IFDEF**에는 **\$ELSE** 절을 사용하지 않습니다. **\$IFDEF** LINUX/**\$ELSE** 또는 **\$IFDEF** MSWINDOWS/**\$ELSE** 대신에 LINUX용의 분리된 **\$IFDEF** 블록과 MSWINDOWS 특정 코드를 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 포팅 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 검사에는 **\$IF** 구문을 사용합니다. 중첩된 **\$IFDEF**를 **\$IF** 지시어의 부울 표현식으로 바꿉니다. **\$IF** 지시어는 **\$ENDIF**가 아닌 **\$IFEND**를 사용해서 종료해야 합니다. 이전 컴파일러로부터 새로운 **\$IF** 구문을 숨기려면 **\$IF** 표현식을 **\$IFDEF** 안에 둡니다.

모든 조건 지시어가 온라인 도움말에 있습니다. 자세한 내용은 도움말의 "Conditional Compilation" 항목을 참조하십시오.

## 조건 지시어 종료

**\$IFEND** 지시어를 사용하여 **\$IF**와 **\$ELSEIF** 조건부 지시어를 종료합니다. **\$IFDEF**/**\$ENDIF**를 사용하는 대신 이렇게 하면 **\$IF**/**\$IFEND** 블록이 이전 컴파일러로부터 숨겨집니다. 그러면 이전 컴파일러가 **\$IFEND** 지시어를 인식하지 못합니다. **\$IF**는 **\$IFEND**로만 종료할 수 있습니다. 이전의 지시어(**\$IFDEF**, **\$IFNDEF**, **\$IFOPT**)는 **\$ENDIF**로만 종료할 수 있습니다.

**참고** **\$IFDEF**/**\$ENDIF** 안에 **\$IF**를 중첩할 경우 **\$ELSE**를 **\$IF**와 함께 사용하지 마십시오. 이전의 컴파일러는 **\$ELSE**를 보고 **\$IFDEF**의 일부로 생각하여 컴파일러 오류를 만듭니다. **\$IF**가 먼저 받아들여지면 **\$ELSEIF**는 받아들여지지 않고 이전 컴파일러는 **\$ELSEIF**를 모르기 때문에, 이러한 상황에서는 **{ \$ELSE }** 대신 **{ \$ELSEIF True }**를 사용할 수 있습니다. 코드를 여러 다른 버전에서 코드가 실행되는 것을 원하는 협력 업체와 애플리케이션 개발자들에게는 역호환성을 위해 **\$IF**를 숨기는 것이 1차적인 문제가 됩니다.

**\$ELSEIF**는 **\$ELSE**와 **\$IF**의 조합입니다. **\$ELSEIF** 지시어를 통해 여러 조건부 블록 중 하나만 받아들여지게 작성할 수 있습니다. 예를 들면,

```
{ $IFDEF doit }
do_doit
{ $ELSEIF RTLVersion >= 14 }
goforit
{ $ELSEIF somestring = 'yes' }
beep
{ $ELSE }
last chance
{ $IFEND }
```

위의 네 가지 경우 중 하나만을 받아들입니다. 처음 세 개의 조건이 true가 아닌 경우 **\$ELSE** 절이 받아들여집니다. **\$ELSEIF**는 **\$IFEND**로 종료해야 합니다. **\$ELSEIF**는 **\$ELSE** 뒤에 나타날 수 없습니다. 조건은 일반적인 **\$IF...\$ELSE**처럼 위에서 아래로 값이 구해집니다. 예제에서 **doit**이 정의되지 않고, **RTLVersion**은 15이며, **somestring** = 'yes'인 경우 조건이 둘 다 True일지라도 "beep" 블록이 아닌 "goforit" 블록만 받아들여집니다.

**\$ENDIF**를 사용하여 **\$IFDEF**를 종료하는 것을 잊을 경우에는 컴파일러가 소스 파일의 끝에 다음 오류 메시지를 보고합니다.

Missing ENDIF

소스 파일에 **\$IF/\$IFDEF** 지시어가 여러 개 있는 경우에는 어느 것이 문제를 일으키는 지 알아내기 어려울 수도 있습니다. Kylix는 일치하지 않는 **\$ENDIF/\$IFEND**가 있는 마지막 **\$IF/\$IFDEF** 컴파일러 지시어의 소스 행에 다음 오류 메시지를 보고합니다.

Unterminated conditional directive

해당 위치에서 문제 찾기를 시작합니다.

## 메시지 나타내기

**\$MESSAGE** 컴파일러 지시어는 컴파일러가 하듯이 소스 코드에서 힌트, 경고 및 오류를 나타내게 할 수 있습니다.

```
{$MESSAGE HINT|WARN|ERROR|FATAL 'text string' }
```

메시지 유형은 옵션입니다. 메시지 유형이 지정되지 않은 경우 기본값은 HINT입니다. 텍스트 문자열이 필요하며 작은 따옴표 안에 포함되어야 합니다.

예제:

```
{$MESSAGE 'Boo!'} 힌트를 표시합니다.
```

```
{$Message Hint 'Feed the cats'} 힌트를 표시합니다.
```

```
{$Message Warn 'Looks like rain.'} 경고를 표시합니다.
```

```
{$Message Error 'Not implemented'} 오류를 표시하고 컴파일을 계속합니다.
```

```
{$Message Fatal 'Bang. Yer dead.'} 오류를 표시하고 컴파일러를 종료합니다.
```

## 인라인 어셈블러 코드 포함

Windows 애플리케이션에 인라인 어셈블러 코드를 포함하면 Linux의 위치 독립 코드 (PIC) 요구 조건 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.

오브젝트 파스칼 코드만을 포함하는 유닛의 경우 컴파일러는 필요할 때 자동으로 PIC를 생성합니다. PIC 유닛은 .dcu가 아닌 .dpu 확장자를 가집니다. 모든 파스칼 유닛 소스 파일은 PIC와 PIC가 아닌 두 가지 형식으로 컴파일하는 것이 좋습니다. -p 컴파일러 스위치를 사용하여 PIC를 생성합니다. 미리 컴파일되어 있는 유닛은 두 가지 형식으로 사용할 수 있습니다.

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 오브젝트 파스칼에서 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.

## 메시지와 시스템 이벤트

메시지 루프와 이벤트는 Linux에서 다르게 수행되지만 컴포넌트 작성에 1차적인 영향을 줍니다. 대부분의 컴포넌트와 속성 편집기는 쉽게 포팅됩니다. 클래스 상의 *TObject.Dispatch*와 메시지 메소드 구문이 Linux에서 제대로 수행되는 반면, Linux 환경에서 운영 체제 공지는 메시지 보다 시스템 이벤트를 사용하여 처리됩니다.

이벤트 핸들러를 작성하려면 Windows 메시지에 응답하는 대신 표 10.9에 설명된 메소드 중 하나를 오버라이드하여 사용자 지정 메시지를 작성할 수 있습니다. 오버라이드 시 상속된 메소드를 호출하면 모든 기본 프로세스 (default process)를 계속 발생시킬 수 있습니다.

**표 10.9** 시스템 이벤트에 응답하기 위한 TWidgetControl protected 메소드

| 메소드                    | 설명                                                                                                                                                                         |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>ChangeBounds</i>    | <i>TWidgetControl</i> 의 크기가 변경될 때 사용됩니다. Windows의 WM_SIZE 또는 WM_MOVE와 대체로 유사합니다. Qt는 클라이언트 영역에 기반하여 widget의 "geometry"를 설정하며 VCL은 Qt가 프레임으로 참조하는 것을 포함하는 전체 컨트롤 크기를 사용합니다. |
| <i>ChangeScale</i>     | 컨트롤 크기 변경 시 자동으로 호출됩니다. 폼의 크기와 다른 화면 해상도에 대한 모든 컨트롤 크기 또는 글꼴 크기를 변경하는 데 사용됩니다. Change Scale은 컨트롤의 Top, Left, Width 및 Height 속성을 수정하므로 크기는 물론 컨트롤과 컨트롤의 자식의 위치를 변경합니다.      |
| <i>ColorChanged</i>    | 컨트롤의 색상이 변경되었을 때 호출됩니다.                                                                                                                                                    |
| <i>CursorChanged</i>   | 커서 모양이 바뀔 때 호출됩니다. 마우스 커서가 이 widget 위에 있을 때 마우스 커서의 모양을 가집니다.                                                                                                              |
| <i>EnabledChanged</i>  | 창이나 컨트롤의 활성화 상태를 애플리케이션이 변경할 때 호출됩니다.                                                                                                                                      |
| <i>FontChanged</i>     | 글꼴 리소스의 컬렉션이 변경될 때 호출됩니다. widget의 글꼴을 설정하고 모든 자식에게 변경 내용을 알려줍니다. WM_FONTCHANGE 메시지와 대체로 유사합니다.                                                                             |
| <i>PaletteChanged</i>  | 시스템 팔레트가 변경될 때 호출됩니다. WM_PALETTECHANGED 메시지와 대체로 유사합니다.                                                                                                                    |
| <i>ShowHintChanged</i> | 도움말 힌트가 표시되거나 컨트롤에 숨겨졌을 때 호출됩니다.                                                                                                                                           |
| <i>StyleChanged</i>    | 창이나 컨트롤의 GUI 스타일이 변경되었을 때 호출됩니다. WM_STYLECHANGED 메시지와 대체로 유사합니다.                                                                                                           |
| <i>TabStopChanged</i>  | 폼 상의 탭 순서가 변경되었을 때 호출됩니다.                                                                                                                                                  |

**표 10.9** 시스템 이벤트에 응답하기 위한 TWidgetControl protected 메소드 (계속)

| 메소드                    | 설명                                  |
|------------------------|-------------------------------------|
| <i>VisibleChanged</i>  | 컨트롤이 숨겨지거나 보여질 때 호출됩니다.             |
| <i>WidgetDestroyed</i> | 컨트롤을 기본으로 사용하는 widget이 소멸될 때 호출됩니다. |

Qt는 C++ 툴킷이므로 모든 Qt widget은 C++ 객체입니다. CLX는 오브젝트 파스칼로 작성되는데 오브젝트 파스칼은 C++ 객체와 직접적으로 상호 작용하지 않습니다. 또한 Qt는 여러 곳에서 다중 상속을 사용합니다. 그러므로 Kylix에는 인터페이스 레이어가 들어 있어 모든 Qt 클래스를 일련의 직접적인 C 함수로 변환합니다. 그런 다음 레이어들은 Linux에서는 공유 객체로, Windows에서는 DLL로 랩됩니다.

모든 *TWidgetControl*은 *CreateWidget*, *InitWidget*과 거의 항상 오버라이드해야 하는 *HookEvents* 가상 메소드를 가집니다. *CreateWidget*은 Qt widget을 만들고 Handle을 FHandle private field 변수로 할당합니다. *InitWidget*은 widget이 생성된 후 호출되고 Handle이 유효하게 됩니다.

Linux에서 일부 속성 할당은 Create 생성자에서 *InitWidget*으로 옮겨졌습니다. 그래서 실제로 필요할 때까지 Qt 객체의 생성을 지연시킵니다. 예를 들어, *Color*라는 이름의 속성이 있다고 가정합니다. *SetColor*에서 Qt 핸들이 있는지 보려면 *HandleAllocated*에서 확인할 수 있습니다. Handle이 할당되어 있으면 Qt을 적절히 호출하여 색상을 설정할 수 있습니다. 할당되지 않았으면 값을 private field 변수에 저장하고 *InitWidget*에서 속성을 설정합니다.

Linux는 Widget과 System이라는 두 가지 유형의 이벤트를 지원합니다. *HookEvents*는 Qt 객체와 통신하는 특별한 혹 객체에 CLX 컨트롤 이벤트 메소드를 호출하는 가상 메소드입니다. 혹 객체는 메소드 포인터의 집합입니다. Linux에서 시스템 이벤트는 *WndProc*에 기본으로 대체되는 *EventHandler*를 거칩니다.

## Linux에서의 프로그래밍 차이점

Linux *wchar\_t* widechar는 문자당 32비트입니다. 오브젝트 파스칼 widechar가 지원하는 16비트 유니코드 표준은 Linux와 GNU 라이브러리가 지원하는 32비트 UCS 표준의 서브셋입니다. 파스칼 widechar 데이터는 *wchar\_t*로 OS 함수에 전달되기 전에 문자당 32비트로 확장되어야 합니다.

Linux에서 와이드 문자열은 긴 문자열처럼 카운트되는 참조입니다 (Windows에서는 아님).

Linux에서의 멀티바이트 처리는 다릅니다. Windows에서 멀티바이트 문자(MBCS)는 1바이트와 2바이트 char 코드로 표시됩니다. Linux에서는 1에서 6바이트로 표시됩니다.

AnsiStrings는 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 포팅할 수 없지만 유니코드는 포팅이 가능합니다.



Linux에서는 절대 주소에 변수를 사용할 수 없습니다. `var X: Integer absolute $1234;` 구문은 PIC에서는 지원하지 않으며 CLX를 포함한 모든 Kylix 버전에서 허용되지 않습니다.

## 크로스 플랫폼 데이터베이스 애플리케이션

Windows에서 Delphi는 데이터베이스 정보 액세스 방법에 대한 여러 선택 사항을 제공합니다. ADO, Borland Database Engine (BDE) 및 InterBase Express 사용이 들어 있습니다. 하지만 Kylix에서는 이 세 가지의 선택 사항을 사용할 수 없습니다. 그 대신 Delphi 버전 6과 함께 출시되어 Windows에서도 사용할 수 있는 새로운 크로스 플랫폼 데이터 액세스 기술인, *dbExpress*를 사용할 수 있습니다.

*dbExpress*가 Linux에서도 실행되도록 데이터베이스 애플리케이션을 포팅하기 전에 *dbExpress*의 사용 방법과 사용자가 이전에 사용하던 데이터 액세스 메커니즘 사용 방법 사이의 차이점을 이해해야 합니다. 그 차이는 여러 수준에서 일어납니다.

- 가장 낮은 수준에는 애플리케이션과 데이터베이스 서버 간에 통신하는 레이어가 있습니다. 이 레이어는 ADO, BDE 또는 InterBase 클라이언트 소프트웨어일 수 있습니다. 이 레이어는 동적 SQL 프로세싱을 위한 경량급 (lightweight) 드라이버 집합인 *dbExpress*로 대체됩니다.
- 낮은 수준의 데이터 액세스는 데이터 모듈이나 폼에 추가하는 컴포넌트 집합에 랩됩니다. 이 컴포넌트에는 데이터베이스 서버로의 연결을 나타내는 데이터베이스 연결 컴포넌트와 서버로부터 페치 (fetch) 된 데이터를 나타내는 데이터셋이 있습니다. 중요한 차이점이 있기는 하지만 *dbExpress* 커서의 단방향성 때문에 이 레벨에서는 그 차이가 적습니다. 그 이유는 데이터베이스 연결 컴포넌트처럼 데이터셋이 공통 조상을 공유하기 때문입니다.
- 사용자 인터페이스 수준에는 차이가 거의 없습니다. CLX data-aware 컨트롤은 해당 Windows 컨트롤에 가능한 유사하게 디자인되었습니다. 사용자 인터페이스 레벨에서의 주요 차이점은 캐시된 업데이트의 사용을 적용하기 위해 변경이 필요할 때 나타납니다.

기존의 데이터베이스 애플리케이션을 *dbExpress*로 포팅하는 방법에 대한 자세한 내용은 10-24 페이지의 "데이터베이스 애플리케이션을 Linux로 포팅"을 참조합니다. 새 *dbExpress* 애플리케이션 디자인에 대한 자세한 내용은 14장 "데이터베이스 애플리케이션 디자인"을 참조합니다.

### dbExpress 차이점

Linux에서 *dbExpress*는 데이터베이스 서버와의 통신을 관리합니다. *dbExpress*는 공통 인터페이스를 구현하는 경량급 (lightweight) 드라이버의 집합으로 구성되어 있습니다. 드라이버는 모두 사용자 애플리케이션에 연결되어야 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 협력 업체가 제공하는 클라이언트측 소프트웨어를 필요로 합니다. 또한 데이터베이스 특정 드라이버와 두 개의 구성 파일 *dbxconnection*과 *dbxdriver*를 사용합니다. 주요한 Borland Database Engine 라이브러리(*Idapi32.dll*)와 데이터베이스 특정 드라이버 및 많은 기타 지원 라이브러리를 필요로 하는 BDE 등의 경우보다는 훨씬 적은 것입니다.

다음은 *dbExpress*와 애플리케이션을 포팅하기 위해 필요한 기타 데이터 액세스 레이어 간의 차이점입니다.

- *dbExpress*는 원격 데이터베이스로의 보다 간단하고 빠른 경로를 가능하게 합니다. 그 결과로 간단하고 빠른 데이터 액세스에서 눈에 띄게 향상된 성능을 기대할 수 있습니다.
- *dbExpress*는 쿼리와 내장 프로시저를 처리할 수 있지만 개방형 테이블의 개념은 지원하지 않습니다.
- *dbExpress*는 단방향 커서만을 반환합니다.
- *dbExpress*는 INSERT, DELETE 또는 UPDATE 쿼리를 실행하는 기능 말고는 기본 제공된 업데이트 지원이 없습니다.
- *dbExpress*는 메타데이터 캐싱을 하지 않으며 디자인 타임 메타데이터 액세스 인터페이스는 코어 데이터 액세스 인터페이스를 사용하여 구현됩니다.
- *dbExpress*는 사용자가 요청한 쿼리만을 실행하므로 다른 쿼리를 끌어들이지 않고 데이터베이스 액세스를 최적화합니다.
- *dbExpress*는 레코드 버퍼나 레코드 버퍼 블록을 내부적으로 관리합니다. 이것이 레코드 버퍼에 사용되는 메모리를 클라이언트가 할당해야 하는 BDE와 다른 점입니다.
- *dbExpress*는 SQL 기반이 아닌(예: Paradox, dBase 또는 FoxPro) 로컬 테이블은 지원하지 않습니다.
- *dbExpress* 드라이버는 InterBase, Oracle, DB2 및 MySQL를 지원합니다. 다른 데이터베이스 서버를 사용하고 있을 경우에는 이 데이터베이스들 중 하나로 데이터를 포팅하고 사용 중인 데이터베이스 서버용 *dbExpress* 드라이버를 작성하거나 사용자의 데이터베이스 서버용으로 사용할 외부 *dbExpress* 드라이버를 구해야 합니다.

## 컴포넌트 수준 차이점

*dbExpress* 애플리케이션을 작성할 때에는 기존의 데이터베이스 애플리케이션에 사용되는 것과 다른 데이터 액세스 컴포넌트 집합이 필요합니다. *dbExpress* 컴포넌트는 다른 데이터 액세스 컴포넌트(*TDataSet* 및 *TCustomConnection*)와 동일한 기본 클래스를 공유하며 이는 속성, 메소드 및 이벤트가 기존 애플리케이션에 사용되는 컴포넌트와 동일하다는 것을 의미합니다.

표 10.10은 Windows의 InterBase Express, BDE 및 ADO에서 사용되는 중요한 데이터베이스 컴포넌트를 나열하며 Linux와 크로스 플랫폼 애플리케이션에서 사용되는 유사한 *dbExpress* 컴포넌트를 보여줍니다.

**표 10.10** 유사한 데이터 액세스 컴포넌트

| InterBase Express 컴포넌트 | BDE 컴포넌트           | ADO 컴포넌트              | dbExpress 컴포넌트               |
|------------------------|--------------------|-----------------------|------------------------------|
| <i>TIBDatabase</i>     | <i>TDatabase</i>   | <i>TADOConnection</i> | <b><i>TSQLConnection</i></b> |
| <i>TIBTable</i>        | <i>TTable</i>      | <i>TADOTable</i>      | <b><i>TSQLTable</i></b>      |
| <i>TIBQuery</i>        | <i>TQuery</i>      | <i>TADOQuery</i>      | <b><i>TSQLQuery</i></b>      |
| <i>TIBStoredProc</i>   | <i>TStoredProc</i> | <i>TADOStoredProc</i> | <b><i>TSQLStoredProc</i></b> |
| <i>TIBDataSet</i>      |                    | <i>TADODataSet</i>    | <b><i>TSQLDataSet</i></b>    |

하지만 *dbExpress* 데이터셋 (*TSQLTable*, *TSQLQuery*, *TSQLStoredProc* 및 *TSQLDataSet*)은 편집을 지원하지 않고 포워드(forward) 탐색만을 허용하기 때문에 자신과 유사한 대상보다 더 제한적입니다. *dbExpress* 데이터셋과 기타 Windows에서 사용 가능한 기타 데이터셋의 차이점에 대한 자세한 내용은 18장 "단방향 데이터셋 사용"을 참조하십시오.

편집 및 탐색 지원이 없으므로 대부분의 *dbExpress* 애플리케이션은 *dbExpress* 데이터셋과 직접 연동하지 않습니다. 오히려 *dbExpress* 데이터셋을 메모리에 레코드를 버퍼하고 편집과 탐색 지원을 제공하는 클라이언트 데이터셋에 연결합니다. 이 아키텍처에 대한 자세한 내용은 14-4 페이지의 "데이터베이스 아키텍처"를 참조하십시오.

**참고** 아주 간단한 애플리케이션의 경우 클라이언트 데이터셋에 연결된 *dbExpress* 데이터셋 대신 *TSQLClientDataSet*을 사용할 수 있습니다. 이는 포팅하려는 애플리케이션의 데이터셋과 포팅된 애플리케이션의 데이터셋 간에 일대일로 대응되기 때문에 간단하다는 이점이 있지만 *dbExpress* 데이터셋을 클라이언트 데이터셋에 명시적으로 연결하는 것 보다는 유연하지 못합니다. 대부분의 애플리케이션에서는 *TClientDataSet* 컴포넌트에 연결된 *dbExpress* 데이터셋을 사용하는 것이 좋습니다.

## 사용자 인터페이스 수준 차이점

CLX data-aware 컨트롤은 해당 Windows 컨트롤과 최대한 유사하게 디자인되어 있습니다. 그 결과 데이터베이스 애플리케이션의 사용자 인터페이스 부분을 포팅하면 Windows 애플리케이션을 CLX로 포팅하는 것 외에 추가로 고려할 사항이 거의 없습니다.

사용자 인터페이스 레벨의 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, 예를 들어 사용자가 이전의 레코드로 이동하는 것을 허용하는 컨트롤을 제거해야 할 수도 있습니다. *dbExpress* 데이터셋은 데이터를 버퍼하지 않으므로 데이터를 data-aware 그리드에 표시할 수 없습니다. 한 번에 데이터 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. 기본적으로 Windows의 대부분의 데이터셋은 포스트될 때 (예를 들어 사용자가 새 레코드로 이동할 때) 자동으로 데이터베이스 서버에 업데이트를 씁니다. 반면 클라이언트 데이터셋은 항상 메모리에 업데이트를 캐시합니다. 이러한 차이점을 조정하는 방법에 대한 내용은 10-26 페이지의 "dbExpress 애플리케이션에서 데이터 업데이트"를 참조합니다.

## 데이터베이스 애플리케이션을 Linux로 포팅

데이터베이스 애플리케이션을 *dbExpress*로 포팅하면 Windows와 Linux 모두에서 실행되는 크로스 플랫폼 애플리케이션을 만들 수 있습니다. 포팅 기법이 다르기 때문에 포팅 과정에는 애플리케이션을 변경하는 것이 포함됩니다. 포팅이 얼마나 어려운지는 애플리케이션의 유형, 복잡한 정도 및 필요 조건 등에 따라 다릅니다. ADO 같은 Windows 특정 기술을 많이 사용하는 애플리케이션은 Delphi 데이터베이스 기술을 사용하는 애플리케이션을 포팅하는 것보다 더 어렵습니다.

다음과 같은 일반적인 절차를 따라 Windows 데이터베이스 애플리케이션을 Linux로 포팅하십시오.

- 1 데이터베이스 데이터를 저장할 곳을 생각합니다. *dbExpress*는 Oracle, Interbase, DB2 및 MySQL용 드라이버를 제공합니다. 데이터는 이들 SQL 서버 중 하나에 두어야 합니다.

Delphi 5 Enterprise 버전이 있으면 Data Pump 유틸리티를 사용하여 Paradox, dBase 및 FoxPro 같은 플랫폼으로부터 로컬 데이터베이스 데이터를 지원하는 플랫폼 중 하나로 옮길 수 있습니다. (이 유틸리티의 사용 방법은 Program Files\Common Files\Borland\Shared\BDE에 있는 datapump.hlp 파일을 참조하십시오.)

- 2 사용자 인터페이스 폼과 데이터셋 및 연결 컴포넌트를 가진 데이터 모듈을 분리하지 않았을 경우 포팅을 시작하기 전에 분리해도 됩니다. 이렇게 하면 데이터 모듈에 완전히 새로운 컴포넌트 집합을 필요로 하는 애플리케이션의 부분을 분리할 수 있습니다. 그런 다음 사용자 인터페이스를 나타내는 폼을 다른 애플리케이션과 마찬가지로 포팅할 수 있습니다. 자세한 내용은 10-2 페이지의 "애플리케이션 포팅"을 참조하십시오.

나머지 단계에서는 데이터셋과 연결 컴포넌트가 자신의 데이터 모듈에서 분리되어 있는 것으로 가정합니다.

- 3 데이터셋의 CLX 버전과 연결 컴포넌트를 보유한 새로운 데이터 모듈을 만듭니다.
- 4 원본 애플리케이션의 각 데이터셋의 경우 *dbExpress* 데이터셋, *TDataSetProvider* 컴포넌트 및 *TClientDataSet* 컴포넌트를 추가합니다. 표 10.10에서 해당되는 것을 사용하여 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* 컴포넌트 보다는 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* 속성과 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 4단계의 각 *dbExpress* 데이터셋마다 해당 *SQLConnection* 속성을 적절한 데이터베이스 연결에 해당하는 *TSQLConnection* 컴포넌트로 설정합니다.
  - 8 각 *TSQLConnection* 컴포넌트마다 데이터베이스 연결에 필요한 정보를 지정합니다. 그렇게 하려면 *TSQLConnection* 컴포넌트를 더블 클릭하여 Connection Editor를 표시하고 매개변수 값을 설정하여 적절한 설정을 나타냅니다. 1단계의 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 앨리어스와 관련되었거나 데이터셋의 *ConnectionString* 속성에 나타난 정보를 전송해야 합니다.

이 매개변수 집합을 새로운 연결 이름으로 저장할 수 있습니다. 이 과정에 대한 자세한 내용은 19-2 페이지의 "서버 연결 설명"을 참조하십시오.

## dbExpress 애플리케이션에서 데이터 업데이트

*dbExpress* 애플리케이션은 클라이언트 데이터셋을 사용하여 편집을 지원합니다. 편집 내용을 클라이언트 데이터셋에 포스트하면 변경 사항이 클라이언트 데이터셋의 메모리 데이터 스냅샷에는 기록되지만 데이터베이스 서버에는 자동으로 기록되지 않습니다. 원본 애플리케이션이 업데이트 캐시를 위해 클라이언트 데이터셋을 사용했으면 Linux에서의 편집 지원을 위해 변경할 필요가 없습니다. 그러나 Windows의 대부분의 데이터셋의 기본 동작에 의존하여, 레코드를 포스트할 때 데이터베이스 서버에 편집 내용을 기록했을 경우 클라이언트 데이터셋을 사용하기 위해 변경해야 합니다.

이전에 업데이트를 캐시하지 않은 애플리케이션을 변환하는 방법에는 두 가지가 있습니다.

- 각 업데이트된 레코드가 포스트되자마자 데이터베이스 서버에 적용하기 위한 코드를 작성하여 Windows에서 데이터셋의 동작을 흉내낼 수 있습니다. 그러려면 업데이트를 데이터베이스 서버에 적용하는 *AfterPost* 이벤트 핸들러가 있는 클라이언트 데이터셋을 다음과 같이 사용합니다.

```
procedure TForm1.ClientDataSet1AfterPost(DataSet:TDataSet);
begin
    with DataSet as TClientDataSet do
        ApplyUpdates(1);
    end;
```

- 캐시된 업데이트를 다루기 위해 사용자 인터페이스를 조정할 수 있습니다. 이 접근법은 네트워크 트래픽의 양을 줄이고 트랜잭션 시간을 최소화하는 등의 이점이 있습니다. 하지만 캐시된 업데이트 사용으로 전환하면 해당 업데이트를 데이터베이스 서버에 다시 적용할 시기를 정해야 하고, 사용자 인터페이스를 변경하여 업데이트된 애플리케이션을 사용자들이 초기화하게 하거나 사용자의 편집 내용이 데이터베이스에 기록했는지 여부에 대한 피드백을 제공해야 할 것입니다. 또한 사용자가 레코드를 포스트할 때 업데이트 오류가 감지되지 않으므로 사용자에게 대해 이러한 오류를 보고하는 방법을 변경하여 어떤 업데이트로 문제를 일으켰는지와 어떤 유형의 문제가 일어났는지를 사용자가 알 수 있게 합니다.

원본 애플리케이션이 업데이트를 캐시하기 위해 BDE나 ADO가 제공하는 지원을 사용했을 경우 코드를 수정하여 클라이언트 데이터셋 사용으로 전환해야 합니다. 다음 표는 BDE와 ADO 데이터셋에서 캐시된 업데이트를 지원하는 속성, 이벤트 및 메소드, *TClientDataSet*의 해당 속성, 메소드 및 이벤트를 나열한 것입니다.

**표 10.11** 캐시된 업데이트를 위한 속성, 메소드 및 이벤트

| BDE 데이터셋<br>(또는<br>TDatabase)               | ADO 데이터셋                            | TclientDataSet                                       | 용도                                                 |
|---------------------------------------------|-------------------------------------|------------------------------------------------------|----------------------------------------------------|
| <i>CachedUpdates</i>                        | <i>LockType</i>                     | 필요없음, 클라이언트 데이터셋은 항상 업데이트를 캐시함.                      | 캐시된 업데이트가 효력이 있는지 결정합니다.                           |
| 지원하지 않음.                                    | <i>CursorType</i>                   | 지원하지 않음.                                             | 서버의 변경으로 인해 데이터셋이 분리된 방법을 지정합니다.                   |
| <i>UpdatesPending</i>                       | 지원하지 않음.                            | <i>ChangeCount</i>                                   | 로컬 캐시가 데이터베이스에 적용하는 데 필요한 업데이트된 레코드를 포함하는지를 나타냅니다. |
| <i>UpdateRecordTypes</i>                    | <i>FilterGroup</i>                  | <i>StatusFilter</i>                                  | 업데이트된 레코드의 종류를 나타내어 캐시된 업데이트를 적용할 때 보이게 합니다.       |
| <i>UpdateStatus</i>                         | <i>RecordStatus</i>                 | <i>UpdateStatus</i>                                  | 레코드가 변경되지 않았는지, 수정되었는지, 삽입되었는지 또는 삭제되었는지를 나타냅니다.   |
| <i>OnUpdateError</i>                        | 지원하지 않음.                            | <i>OnReconcileError</i>                              | 레코드별로 업데이트 오류를 처리하기 위한 이벤트입니다.                     |
| <i>ApplyUpdates</i><br>(데이터셋이나<br>데이터베이스에서) | <i>UpdateBatch</i>                  | <i>ApplyUpdates</i>                                  | 레코드를 로컬 캐시의 데이터베이스에 적용합니다.                         |
| <i>CancelUpdates</i>                        | Cancel<br>Updates or<br>CancelBatch | <i>CancelUpdates</i>                                 | 대기 중인 업데이트를 적용하지 않고 로컬 캐시에서 제거합니다.                 |
| <i>CommitUpdates</i>                        | 자동 처리됨                              | <i>Reconcile</i>                                     | 성공적으로 업데이트된 애플리케이션 다음에 이어지는 업데이트 캐시를 지웁니다.         |
| <i>FetchAll</i>                             | 지원하지 않음                             | <i>GetNextPacket</i><br>(및<br><i>PacketRecords</i> ) | 데이터베이스 레코드를 편집 및 업데이트할 목적으로 로컬 캐시에 복사합니다.          |
| <i>RevertRecord</i>                         | <i>CancelBatch</i>                  | <i>RevertRecord</i>                                  | 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 서버측 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.

Kylix를 사용하면 Linux에 배치하기 위한 CGI나 Apache 애플리케이션과 같은 웹 서버 애플리케이션을 작성할 수 있습니다. Windows에서는 Microsoft Server DLL (ISAPI), Netscape Server DLL (NSAPI) 및 Windows CGI 애플리케이션과 같은 다른 유형의 웹 서버를 만들 수 있습니다. 직접적인 CGI 애플리케이션과 WebBroker를 사용하는 일부 애플리케이션만이 Windows와 Linux 모두에서 실행됩니다.

### 인터넷 애플리케이션을 Linux로 포팅하기

---

Linux로 옮기려는 인터넷 애플리케이션을 사용하고 있는 경우 웹 서버 애플리케이션을 포팅할 것인지 아니면 Linux에서 새 애플리케이션을 작성할 것인지를 고려해야 합니다. 웹 서버 작성에 대한 내용은 22장 "인터넷 서버 애플리케이션 만들기"을 참조하십시오. 애플리케이션이 WebBroker를 사용하고 WebBroker 인터페이스에 쓰지만 원시 API 호출을 사용하지 않는 경우에는 Linux로 포팅하는 것만큼 어렵지는 않습니다.

애플리케이션이 ISAPI, NSAPI, Windows CGI 또는 기타 웹 API를 사용하는 경우에는 포팅하기가 더 어렵습니다. 소스 파일을 통해 검색하고 이 API 호출을 Apache (Apache API용 함수 프로토타입에 대해서는 인터넷 디렉토리에서 httpd.pas 참조)나 CGI 호출로 변환해야 합니다. 10-1 페이지의 "Windows 애플리케이션을 Linux로 포팅"에 설명되어 있는 다른 모든 내용들도 변경해야 합니다.



## 패키지와 컴포넌트 사용

*패키지*는 Kylix 애플리케이션과 IDE가 사용하는 특수 공유 객체 파일입니다. *런타임 패키지*는 사용자가 애플리케이션을 실행할 때 기능을 제공합니다. *디자인 타임 패키지*는 IDE에 컴포넌트를 설치하고 사용자 지정 컴포넌트에 대한 특수 속성 편집기를 생성하는 데 사용됩니다. 단일 패키지가 디자인 타임과 런타임 모두 사용할 수 있고, 디자인 타임 패키지는 종종 런타임 패키지를 호출하여 작동합니다. 패키지는 보통 bpl 접미어가 있는 bplpackage.so 같은 공유 객체 파일에 저장됩니다.

다른 런타임 라이브러리처럼 패키지는 애플리케이션 사이에 공유할 수 있는 코드를 포함할 수 있습니다. 예를 들어, 가장 많이 사용되는 Kylix 컴포넌트는 bplclx라는 패키지에 있습니다. 애플리케이션을 생성할 때마다 bplclx를 사용하도록 지정할 수 있습니다. 이러한 방법으로 생성한 애플리케이션을 컴파일하면 애플리케이션의 실행 이미지는 코드와 코드에 고유한 데이터만 포함합니다. 공통 코드는 bplclx.so.6에 있습니다. 몇 개의 패키지 사용 가능 애플리케이션을 설치한 컴퓨터는 bplclx.so.6 복사본 하나만 필요합니다. 이는 모든 애플리케이션과 IDE가 bplclx.so.6 복사본을 공유하기 때문입니다.

Kylix에는 CLX 컴포넌트를 캡슐화하는 몇 개의 미리 컴파일된 런타임 패키지가 함께 포함되어 있습니다. Kylix는 또한 IDE에서 컴포넌트를 처리하는 디자인 타임 패키지를 사용합니다.

애플리케이션을 패키지와 함께 또는 패키지 없이 구축할 수 있습니다. 그러나 IDE에 사용자 지정 컴포넌트를 추가하려면 패키지를 디자인 타임 패키지로 설치해야 합니다.

애플리케이션 사이에 공유하는 런타임 패키지를 사용자가 직접 만들 수 있습니다. Kylix 컴포넌트를 작성하는 경우 컴포넌트를 설치하기 전에 디자인 타임 패키지로 컴파일할 수 있습니다.

### 패키지의 이점

디자인 타임 패키지는 사용자 지정 컴포넌트를 배포하고 설치하는 작업을 단순화합니다. 옵션으로 사용하는 런타임 패키지는 전통적인 프로그래밍을 증가하는 몇 가지 이점을 제공합니다. 재사용된 코드를 런타임 라이브러리로 컴파일하여 애플리케이션 간에 공유할 수 있습니다. 예를 들어, Kylix를 비롯한 모든 애플리케이션은 패키지를 통해 표

준 컴포넌트에 액세스할 수 있습니다. 애플리케이션에는 실행 파일에 바운드된 별도의 컴포넌트 라이브러리의 복사본이 없기 때문에 실행 파일은 더욱 작아져 시스템 자원과 하드 디스크 저장소가 절약됩니다. 더욱이, 애플리케이션에 고유한 코드만이 빌드 시 컴파일되기 때문에 패키지를 사용하면 컴파일이 빨라집니다.

## 패키지와 표준 공유 객체 파일

IDE를 통해 사용할 수 있는 사용자 지정 컴포넌트를 만들려고 하는 경우 패키지를 만듭니다. 애플리케이션 구축에 사용된 개발 툴에 상관없이 애플리케이션에서 호출할 수 있는 라이브러리를 구축하려는 경우 표준 공유 객체 파일을 만듭니다.

다음 표는 패키지에 연결된 파일 형식을 나열합니다.

표 11.1 컴파일된 패키지 파일

| 파일 확장자 | 내용                                                                                                                                                                                  |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| dpc    | 패키지에 포함된 유닛을 나열하는 소스 파일.                                                                                                                                                            |
| dcp    | 컴파일러가 필요로 하는 모든 기호 정보를 비롯한 패키지 헤더와 패키지의 모든 dpu 파일의 연결을 포함하는 바이너리 이미지. 각 패키지에 대해 dcp 파일이 하나 생성됩니다. dcp에 대한 기본 이름은 dpc 소스 파일의 기본(base) 이름입니다. 패키지과 함께 애플리케이션을 구축하려면 .dcp 파일이 있어야 합니다. |
| dpu    | 패키지에 포함된 유닛 파일에 대한 바이너리 이미지 각 유닛 파일에 대해 필요한 경우 dpu 하나가 생성됩니다                                                                                                                        |
| so     | 런타임 패키지. 이 파일은 특별한 Kylix 특정 기능을 가진 공유 객체 파일입니다. 패키지 이름은 bplpackage.so이며 여기서 <i>package</i> 는 dpc 파일의 기본(base) 이름입니다.                                                                |

**참고** 패키지는 애플리케이션에 있는 다른 모듈과 전역 데이터를 공유합니다.

공유 객체 파일과 패키지에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.

## 런타임 패키지

런타임 패키지는 Kylix 애플리케이션과 함께 배포됩니다. 런타임 패키지는 사용자가 애플리케이션을 실행할 때 기능을 제공합니다.

패키지를 사용하는 애플리케이션을 실행하려면 컴퓨터에 애플리케이션의 실행 파일과 애플리케이션이 사용하는 모든 패키지가 있어야 합니다. 패키지 파일은 애플리케이션이 사용하는 시스템 경로에 있어야 합니다. 애플리케이션을 배포하는 경우 사용자가 필요한 패키지의 올바른 버전을 가지고 있는지 확인해야 합니다.

## 애플리케이션에서 패키지 사용

애플리케이션에서 패키지를 사용하려면 다음과 같이 합니다.

- 1 IDE에서 프로젝트를 로드하거나 만듭니다.
- 2 Project|Options를 선택합니다.

- 3 Packages 탭을 선택합니다.
- 4 "Build with Runtime Packages" 확인란을 선택하고, 아래에 있는 편집 상자에 패키지 이름을 하나 이상 입력합니다. 설치된 디자인 타임 패키지에 연결된 런타임 패키지를 편집 상자에서 선택해도 됩니다.
- 5 기존 목록에 패키지를 추가하려면 Add 버튼을 클릭하고 Add Runtime Package 대화 상자에 새 패키지 이름을 입력합니다.
- 6 사용 가능한 패키지 목록에서 검색하려면 Add 버튼을 클릭한 다음 Add Runtime Package 대화 상자의 Package Name 편집 상자 옆에 있는 Browse 버튼을 클릭합니다.

Add Runtime Package 대화 상자의 Search Path 편집 상자를 편집하면 Kylix의 전역 Library Path가 변경됩니다.

패키지 이름에 파일 확장자를 쓸 필요는 없습니다. Runtime Packages 편집 상자에 직접 입력하는 경우 여러 개의 이름은 세미콜론으로 분리해야 합니다.

Runtime Packages 편집 상자에 나열된 패키지는 컴파일할 때 애플리케이션에 자동으로 연결됩니다. 중복되는 패키지 이름은 무시되며 편집 상자가 비어 있으면 애플리케이션은 패키지 없이 컴파일됩니다.

런타임 패키지는 현재 프로젝트에 대해서만 선택됩니다. 현재 선택값을 새 프로젝트에 대한 자동 기본값으로 만들려면 대화 상자 아래쪽에 있는 "Defaults" 확인란을 선택합니다.

**참고** 패키지가 포함된 애플리케이션을 생성하려면 원래의 Kylix 유닛 이름을 소스 파일의 **uses** 절에 계속 포함해야 합니다. 예를 들어, 주요 폼의 소스 파일은 다음과 같이 시작합니다.

```
unit MainForm;

interface

uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

이 예제에서 참조되는 각 유닛은 bplclx 패키지에 포함되어 있습니다. 애플리케이션에 bplclx를 사용함에도 불구하고 **uses** 절에 이러한 참조가 포함되어야 합니다. 그렇지 않으면 컴파일러 오류가 발생합니다. 생성된 소스 파일에서 Kylix는 이 유닛을 자동으로 **uses** 절에 추가합니다.

## 패키지 동적 로딩

런타임에 패키지를 로드하려면 *LoadPackage* 함수를 호출합니다. 예를 들어, 다음 코드는 파일이 파일 선택 대화 상자에서 선택될 때 실행될 수 있습니다.

```
with OpenFileDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

패키지를 동적으로 언로드하려면 *UnloadPackage*를 호출합니다. 패키지에 정의된 클래스 인스턴스를 소멸하고 패키지로 등록된 클래스의 등록을 해제할 때 주의하십시오.

## 사용할 런타임 패키지 결정

Kylix에는 bplclx를 포함하여 기본 런타임과 컴포넌트 지원을 제공하는 몇 개의 미리 컴파일된 런타임 패키지가 함께 포함되어 있습니다.

bplclx 패키지에는 가장 일반적으로 사용되는 컴포넌트, 시스템 함수 및 사용자 인터페이스 요소가 있습니다. 그러나 별도의 패키지에서 사용할 수 있는 데이터베이스 컴포넌트는 포함하지 않습니다.

Kylix에 포함된 다른 런타임 패키지 목록은 온라인 도움말 인덱스의 "runtime packages, precompiled"를 참조하십시오.

패키지를 사용하는 클라이언트/서버 데이터베이스 애플리케이션을 생성하려면 최소한 bplclx와 bpldataclx 런타임 패키지가 필요합니다. 이 패키지를 사용하려면 Project|Options를 선택하고 Packages 탭을 선택하여 Runtime Packages 편집 상자에 사용하는 패키지를 나열합니다.

## 사용자 지정 패키지

사용자 지정 패키지는 사용자가 직접 코딩하고 컴파일한 패키지 또는 벤더가 제공한 미리 컴파일된 패키지입니다. 애플리케이션과 함께 사용자 지정 런타임 패키지를 사용하려면 Project|Options를 선택하고 Packages 페이지에 있는 Runtime Packages 편집 상자에 패키지 이름을 추가합니다. 예를 들어, bplstats.so라는 통계 패키지가 있다고 가정합니다. 애플리케이션에서 사용하려면 Runtime Packages 편집 상자에 이 패키지를 포함합니다.

사용자가 패키지를 직접 만드는 경우 필요에 따라 목록에 추가할 수 있습니다.

## 디자인 타임 패키지

디자인 타임 패키지는 IDE의 컴포넌트 팔레트에 컴포넌트를 설치하고 사용자 지정 컴포넌트에 대한 특수 속성 편집기를 생성하는 데 사용됩니다.

Kylix에는 IDE에 미리 설치된 다음의 디자인 타임 컴포넌트 패키지가 함께 포함되어 있습니다.

표 11.2 디자인 타임 패키지

| 패키지         | 컴포넌트 팔레트 페이지                                   |
|-------------|------------------------------------------------|
| dclstd      | Standard, Additional, Common Controls, Dialogs |
| dcldbdesign | Data Controls, dbExpress, Data Access          |
| dclnet      | Internet                                       |
| dclindy     | Indy Clients, Indy Servers, Indy Misc          |

디자인 타임 패키지는 Requires clauses에 참조하는 런타임 패키지를 호출함으로써 실행됩니다. 11-8 페이지의 "Requires 절"을 참조하십시오. 예를 들어, dclstd는 bplvcl을 참조합니다. dclstd 패키지에는 많은 표준 컴포넌트를 컴포넌트 팔레트에 사용할 수 있는 추가 기능이 있습니다.

이미 설치된 패키지 외에 IDE에 사용자의 컴포넌트 패키지나 타사 개발자가 제공한 컴포넌트 패키지를 설치할 수 있습니다. dclusr 디자인 타임 패키지는 새 컴포넌트에 대한 기본 컨테이너로 제공됩니다.

## 컴포넌트 패키지 설치

모든 컴포넌트는 패키지로 IDE에 설치됩니다. 직접 컴포넌트를 만든 경우 컴포넌트를 포함하는 패키지를 만들고 컴파일합니다. 11-6 페이지의 "패키지 생성 및 편집"을 참조하십시오. 사용자의 컴포넌트 소스 코드는 4부 "사용자 지정 컴포넌트 생성"에서 설명한 방법을 따라야 합니다.

사용자의 컴포넌트 또는 타사 공급자가 제공하는 컴포넌트를 설치하거나 설치를 제거하려면 다음 단계를 따릅니다.

- 1 새 패키지를 설치하는 경우 패키지 파일을 로컬 디렉토리로 복사하거나 이동합니다. 패키지에 추가 파일이 포함되어 있는 경우 추가 파일을 모두 복사했는지 확인합니다. 추가된 파일에 대한 자세한 내용은 11-11 페이지의 "컴파일이 성공했을 때 생성되는 패키지 파일"을 참조하십시오.

.dcp 파일과 .dpu 파일이 배포본에 포함된 경우 이 파일을 저장하는 디렉토리는 Kylix Library Path에 있어야 합니다.

- 2 Component|Install Packages를 IDE 메뉴에서 선택하거나 Project|Options를 선택하고 Packages 탭을 클릭합니다.
- 3 사용 가능한 패키지 목록이 "Design packages" 아래에 나타납니다.
  - IDE에 패키지를 설치하려면 옆에 있는 확인란을 선택합니다.
  - 패키지를 설치 제거하려면 확인란의 선택을 해제합니다.
  - 설치된 패키지에 포함된 컴포넌트 목록을 보려면 패키지를 선택하고 Components를 클릭합니다.
  - 패키지를 목록에 추가하려면 Add를 클릭하고 Open Package 대화 상자에서 패키지 파일이 있는 디렉토리를 검색합니다 (단계 1 참조). 패키지 파일을 선택하고 Open을 클릭합니다.
  - 목록에서 패키지를 제거하려면 패키지를 선택하고 Remove를 클릭합니다.
- 4 OK를 클릭합니다.

컴포넌트의 *RegisterComponents* 프로시저에서 지정한 컴포넌트 팔레트에 패키지의 컴포넌트를 동일한 프로시저에서 할당된 이름으로 설치합니다.

기본 설정값을 변경하지 않는 경우 모든 사용 가능한 패키지를 설치한 새 프로젝트가 생성됩니다. 현재 설치 선택 사항을 새 프로젝트의 자동 기본값으로 만들려면 대화 상자 아래쪽에 있는 Default 확인란을 선택합니다.

패키지를 설치 제거하지 않고 컴포넌트 팔레트에서 컴포넌트를 제거하려면 Component |Configure Palette를 선택하거나 Tools|Environment Options를 선택하고 Palette 탭을 클릭합니다. Palette 옵션 탭은 컴포넌트가 나타나는 컴포넌트 팔레트 페이지의 이름과 함께 설치된 각 컴포넌트를 나열합니다. 컴포넌트를 선택하고 Hide를 클릭하면 팔레트에서 컴포넌트가 제거됩니다.

## 패키지 생성 및 편집

패키지를 만들려면 다음을 지정해야 합니다.

- 패키지의 이름.
- 새 패키지에 필요하거나 새 패키지에 연결되는 다른 패키지의 목록.
- 컴파일될 때 패키지가 포함하거나 패키지에 바운드되는 유닛 파일의 목록. 패키지는 본질적으로 컴파일된 패키지의 기능을 포함하는 소스 코드 유닛에 대한 래퍼(wrapper)입니다. Contains 절에 패키지에 컴파일하려는 사용자 지정 컴포넌트의 소스 코드 유닛을 지정합니다.

.dpk 확장자로 끝나는 패키지 소스 파일은 Package Editor에 의해 생성됩니다.

## 패키지 생성

패키지를 만들려면 다음 절차를 따릅니다. 여기에서 대략적으로 설명한 단계에 대한 자세한 내용은 11-8 페이지의 "패키지 구조 이해"를 참조하십시오.

**참고** 크로스 플랫폼 애플리케이션을 개발하는 경우라면 IFDEF을 패키지 파일(.dpk)에 사용하지 마십시오. 그러나 소스 코드에는 사용할 수 있습니다.

- 1 File|New를 선택하고 Package 아이콘을 선택한 다음 OK를 클릭합니다.
- 2 생성된 패키지는 Package Editor에 나타납니다.
- 3 Package Editor는 새 패키지의 *Requires* 노드와 *Contains* 노드를 보여줍니다.
- 4 Add to package 스피드 버튼을 클릭하여 유닛을 **contains** 절에 추가합니다. Add unit 페이지에서, Unit file name 편집 상자에 .pas 파일명을 입력하거나 Browse를 클릭하여 파일을 검색한 다음 OK를 클릭합니다. 선택한 유닛은 Package Editor의 Contains 노드 아래에 나타납니다. 추가 유닛도 이 단계를 반복하여 추가할 수 있습니다.
- 5 Add to package 스피드 버튼을 클릭하여 패키지를 **requires** 절에 추가합니다. Requires 페이지에서 Package name 편집 상자에 .dcp 파일명을 입력하거나 Browse를 클릭하여 파일을 검색한 다음 OK를 클릭합니다. 선택한 패키지는 Package Editor의 Requires 노드 아래에 나타납니다. 추가 패키지도 이 단계를 반복하여 추가할 수 있습니다.
- 6 Options 스피드 버튼을 클릭하고 빌드하려는 패키지 종류를 결정합니다.
  - 디자인 타임 전용 패키지(런타임에 사용할 수 없는 패키지)를 만들려면 Designtime only 라디오 버튼을 선택합니다. 또는 {\$DESIGNONLY} 컴파일러 지시어를 dpk 파일에 추가합니다.

- 런타임 전용 패키지(설치될 수 없는 패키지)를 만들려면 Runtime only 라디오 버튼을 선택합니다. 또는 `{$RUNONLY}` 컴파일러 지시어를 dpk 파일에 추가합니다.
  - 디자인 타임과 런타임 모두에 사용할 수 있는 패키지를 만들려면 Designtime and runtime 라디오 버튼을 선택합니다.
- 7 Package Editor에서 Compile package 스피드 버튼을 클릭하여 패키지를 컴파일합니다.

## 기존 패키지 편집

다음과 같은 몇 가지 방법으로 편집을 위해 기존 패키지를 열 수 있습니다.

- File|Open (또는 File|Reopen)을 선택하고 dpk 파일을 선택합니다.
- Component|Install Packages를 선택하고 Design Packages 목록에서 패키지를 선택한 다음 Edit 버튼을 클릭합니다.
- Package Editor가 열리면 Requires 노드의 패키지 중 하나를 선택하여 마우스 오른쪽 버튼을 클릭한 다음 Open을 선택합니다.

패키지의 설명을 편집하거나 사용 옵션을 설정하려면 Package Editor에서 Options 스피드 버튼을 클릭하고 Description 탭을 선택합니다.

Project Options 대화 상자는 하단 왼쪽 모서리에 Default 확인란이 있습니다. 이 확인란을 선택하고 OK를 클릭하면 선택한 옵션은 새 프로젝트에 대한 기본 설정값으로 저장됩니다. 원래의 기본값을 복원하려면 defjproj.dof 파일을 삭제하거나 이름을 변경합니다.

## 수동으로 패키지 소스 파일 편집

프로젝트 파일처럼 패키지 소스 파일은 사용자가 제공하는 정보로부터 Kylix에 의해 생성됩니다. 프로젝트 파일처럼 패키지 소스 파일도 수동으로 편집할 수 있습니다. 패키지 소스 파일은 오브젝트 파스칼 소스 코드를 포함하는 다른 파일과의 혼동을 피하기 위해 .dpk (Kylix 패키지) 확장자로 저장해야 합니다.

코드 에디터에서 패키지 소스 파일을 열려면 다음과 같이 합니다.

- 1 Package Editor에서 패키지를 엽니다.
- 2 Package Editor를 마우스 오른쪽 버튼으로 클릭하고 View Source를 선택합니다.
  - **package** 헤더는 패키지 이름을 지정합니다.
  - **requires** 절은 현재 패키지가 사용하는 다른 외부 패키지를 나열합니다. 다른 패키지에서 유닛을 사용하는 어떤 유닛도 패키지에 포함되지 않으면 **requires** 절은 필요하지 않습니다.
  - **contains** 절은 컴파일되고 패키지에 바운드된 유닛 파일을 식별합니다. 필요한 패키지에 없는 포함된 유닛이 사용하는 모든 유닛은 contains 절에 나열되지는 않지만(컴파일러가 경고를 보냄) 패키지에 바운드됩니다.

예를 들어, 다음 코드는 bplclxdb 패키지를 선언합니다.

```
package bplclxdb;
requires bplclx;
contains Db, Dbcgrids, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables,
mycomponent in 'usr/components/mycomponent.pas';
end.
```

## 패키지 구조 이해

패키지는 다음을 포함합니다.

- 패키지 이름
- Requires 절
- Contains 절

## 패키지 이름 지정

패키지 이름은 프로젝트 내에서 고유해야 합니다. 패키지 이름을 **stats**라고 지정하면 Package Editor는 stats.dpk라는 소스 파일을 생성합니다. 컴파일러는 각각 bplstats.so와 stats.dcp라는 실행 파일과 바이너리 이미지를 생성합니다. 다른 패키지의 **requires** 절에서 패키지를 참조하거나 또는 애플리케이션에서 패키지를 사용할 때 stats를 사용합니다.

## Requires 절

**requires** 절은 현재 패키지가 사용하는 다른 외부 패키지를 지정합니다. **requires** 절에 포함된 외부 패키지는 외부 패키지에 포함된 유닛 중 하나와 현재 패키지를 모두 사용하는 애플리케이션에 컴파일 시 자동으로 연결됩니다.

패키지에 포함된 유닛 파일이 다른 패키지화된 유닛을 참조하면 다른 패키지는 사용자 패키지의 **requires** 절에 나타나야 하거나 사용자가 다른 패키지를 추가해야 합니다. 다른 패키지가 **requires** 절에서 생략되면 컴파일러는 다른 패키지를 "암시적으로 포함된 유닛"인 사용자의 패키지에 import합니다.

**참고** 사용자가 만든 대부분의 패키지는 bplclx가 필요합니다. SysUtils 유닛을 비롯한 CLX 유닛에 의존하는 패키지는 **requires** 절에 bplclx를 나열하거나 또는 bplclx가 필요한 다른 패키지를 나열합니다.

## 순환 패키지 참조 피하기

패키지는 **requires** 절에 순환 참조를 포함할 수 없습니다. 이는 다음을 의미합니다.

- 패키지는 자신의 **requires** 절에서 자신을 참조할 수 없습니다.
- 참조 체인은 체인에서 패키지를 참조하지 않고 종료해야 합니다. 패키지 A가 패키지 B를 요구하면 패키지 B는 패키지 A를 요구할 수 없습니다. 패키지 A가 패키지 B를 요구하고 패키지 B가 패키지 C를 요구하면 패키지 C는 패키지 A를 요구할 수 없습니다.



## 중복 패키지 참조 처리

패키지의 **requires** 절이나 Runtime Packages 편집 상자의 중복 참조는 컴파일러에 의해 무시됩니다. 그러나 프로그래밍의 명확성과 가독성을 위해 중복 패키지 참조를 찾아서 제거해야 합니다.

## Contains 절

**contains** 절은 패키지로 바운드되는 유닛 파일을 식별합니다. 직접 패키지를 작성하는 경우 소스 코드를 pas 파일로 저장하고 **contains** 절에 이 파일을 포함합니다.

## 소스 코드 중복 사용 피하기

패키지는 다른 패키지의 **contains** 절에 나타날 수 없습니다.

패키지의 **contains** 절에 직접적으로 포함되거나 유닛에 간접적으로 포함된 모든 유닛은 컴파일 시 패키지에 바운드됩니다.

*Kylix IDE*를 비롯한 동일한 애플리케이션이 사용하는 둘 이상의 패키지에 유닛을 직접적 또는 간접적으로 포함할 수 없습니다. 이는 bplclx에 있는 유닛 중 하나를 포함하는 패키지를 만들면 IDE에 패키지를 설치할 수 없다는 것을 의미합니다. 다른 패키지에 이미 패키지로 만든 유닛 파일을 사용하면 첫 번째 패키지를 두 번째 패키지의 **requires** 절에 둡니다.

## 패키지 컴파일

---

IDE나 명령줄에서 패키지를 컴파일할 수 있습니다. IDE에서 패키지를 다시 컴파일하려면 다음과 같이 합니다.

- 1 File|Open을 선택합니다.
- 2 Kylix Package (\*.dpk)를 Files of Type 드롭다운 목록에서 선택합니다.
- 3 대화 상자에서 .dpk 파일을 선택합니다.
- 4 Package Editor를 열 때 Compile 스피드 버튼을 클릭합니다.

컴파일러 지시어를 패키지 소스 코드에 삽입할 수 있습니다. 자세한 내용은 아래의 "패키지 특정 컴파일러 지시어"를 참조하십시오.

명령줄에서 컴파일하면 몇 가지 패키지 특정 스위치를 사용할 수 있습니다. 자세한 내용은 11-11 페이지의 "명령줄 컴파일러와 링커 사용"을 참조하십시오.

## 패키지 특정 컴파일러 지시어

다음 표는 소스 코드에 삽입할 수 있는 패키지 특정 컴파일러 지시어를 나열한 것입니다.

**표 11.3** 패키지 특정 컴파일러 지시어

| 지시어                          | 용도                                                                                                          |
|------------------------------|-------------------------------------------------------------------------------------------------------------|
| {\$IMPLICITBUILD OFF}        | 나중에 패키지가 암시적으로 다시 컴파일되지 않게 합니다. 저수준 기능을 제공하고, 빌드 사이에서 드물게 변경되거나 소스 코드가 배포되지 않을 패키지를 컴파일할 때 .dpk 파일에서 사용합니다. |
| {\$G-} 또는 {IMPORTEDDATA OFF} | import한 데이터 참조를 생성하지 못하게 합니다. 이 지시어는 메모리 액세스 효율을 높이지만 지시어가 있는 유닛은 다른 패키지에서 변수를 참조할 수 없습니다.                  |
| {\$WEAKPACKAGEUNIT ON}       | 유닛을 "약하게" 패키지화합니다. 아래의 11-10 페이지의 "약한 패키지"를 참조하십시오.                                                         |
| {\$DENYPACKAGEUNIT ON}       | 유닛을 패키지에 놓을 수 없습니다.                                                                                         |
| {\$DESIGNONLY ON}            | IDE에서 설치를 위해 패키지를 컴파일합니다. .dpk 파일에 놓습니다.                                                                    |
| {\$RUNONLY ON}               | 패키지를 런타임 전용으로 컴파일합니다. .dpk 파일에 놓습니다.                                                                        |

**참고** {\$DENYPACKAGEUNIT ON}을 소스 코드에 포함시키면 유닛 파일은 패키지화될 수 없습니다. {\$G-} 또는 {IMPORTEDDATA OFF}를 포함시키면 패키지는 동일한 애플리케이션에서 다른 패키지와 함께 사용할 수 없습니다. {\$DESIGNONLY ON} 지시어와 함께 컴파일된 패키지는 IDE가 요구한 여분의 코드를 포함하기 때문에 보통 애플리케이션에서 사용하지 않아야 합니다. 적절한 경우 다른 컴파일러 지시어를 패키지 소스 코드에 포함시켜야 합니다. 여기에서 논의되지 않은 컴파일러 지시어에 대한 내용은 온라인 도움말의 Compiler directives를 참조하십시오.

### 약한 패키지

**\$WEAKPACKAGEUNIT** 지시어는 .dpu 파일을 패키지의 파일에 저장하는 방식에 영향을 줍니다. 컴파일러가 생성하는 파일에 대한 내용은 11-11 페이지의 "컴파일이 성공했을 때 생성되는 패키지 파일"을 참조하십시오. {\$WEAKPACKAGEUNIT ON}이 유닛 파일에 나타나면, 컴파일러는 가능한 경우 유닛을 패키지에서 생략하고, 다른 애플리케이션이나 패키지가 요구할 때 유닛의 패키지화되지 않은 로컬 복사본을 생성합니다. 이 지시어와 함께 컴파일된 유닛을 "약하게 패키지화"되었다고 합니다.

예를 들어, unit1이라는 하나의 유닛만을 포함하는 **pack**이라는 패키지를 만들었다고 가정합니다. unit1은 다른 유닛을 사용하지 않지만 rare.so를 호출했다고 가정합니다. 패키지를 컴파일할 때 {\$WEAKPACKAGEUNIT ON}을 unit1.pas에 놓으면 unit1은 bplpack.so에 포함되지 않습니다. rare.so의 복사본을 pack과 함께 배포할 필요는 없습니다. 그러나 unit1은 여전히 pack.dcp에 포함됩니다. pack을 사용하는 다른 패키지나 애플리케이션이 unit1을 참조하면, pack.dcp에서 unit1을 복사하고 프로젝트에 직접 컴파일합니다.

이제 두 번째 유닛인 unit2를 pack에 추가한다고 가정합니다. unit2가 unit1을 사용한다고 가정합니다. 이번에는 pack을 **{\$WEAKPACKAGEUNIT ON}**과 함께 unit1.pas에서 컴파일해도 컴파일러는 unit1을 bplpack.so에 포함합니다. 그러나 unit1을 참조하는 다른 패키지나 애플리케이션은 pack.dcp에서 가져온 패키지화되지 않은 복사본을 사용합니다.

**참고** **{\$WEAKPACKAGEUNIT ON}** 지시어를 포함하는 유닛 파일은 전역 변수, 초기화 섹션 또는 완료 섹션이 없어야 합니다.

**\$WEAKPACKAGEUNIT** 지시어는 다른 Kylix 프로그래머에게 패키지를 배포하는 개발자를 위한 고급 기능입니다. 이 지시어를 사용하면 드물게 사용되는 공유 객체 파일의 배포를 막을 수 있고, 동일한 외부 라이브러리에 의존하는 패키지 간의 충돌을 없앨 수 있습니다.

## 명령줄 컴파일러와 링커 사용

명령줄에서 컴파일하는 경우 다음 테이블에서 나열하는 패키지 특정 스위치를 사용할 수 있습니다.

표 11.4 패키지 특정 명령줄 컴파일러 스위치

| 스위치        | 용도                                                                                                      |
|------------|---------------------------------------------------------------------------------------------------------|
| -G-        | import한 데이터 참조를 생성할 수 없습니다 이 스위치를 사용하면 메모리 액세스 효율을 증가시키지만 이 스위치를 사용하여 컴파일된 패키지는 다른 패키지의 변수를 참조할 수 없습니다. |
| -LEpath    | 패키지(bcppackage.so) 파일을 저장할 디렉토리를 지정합니다.                                                                 |
| -LNpath    | 패키지(package.dcp) 파일을 저장할 디렉토리를 지정합니다.                                                                   |
| -LUpackage | 패키지를 사용합니다.                                                                                             |
| -Z         | 나중에 패키지가 암시적으로 다시 컴파일되지 않게 합니다. 저수준의 기능을 제공하고 드물게 빌드 사이에서 변경되거나 소스 코드가 배포되지 않을 패키지를 컴파일할 때 사용합니다.       |

**참고** **-G-** 스위치를 사용하면 패키지를 동일한 애플리케이션에서 다른 패키지와 함께 사용할 수 없습니다. 적절한 경우 다른 명령줄 옵션을 패키지를 컴파일할 때 사용할 수도 있습니다. 여기에서 설명하지 않은 명령줄 옵션에 대한 내용은 온라인 도움말의 "The Command-line compiler"를 참조하십시오.

## 컴파일이 성공했을 때 생성되는 패키지 파일

패키지를 생성하려면 확장자가 .dpk인 소스 파일을 컴파일합니다. .dpk 파일의 기본(base) 이름은 컴파일러가 생성한 파일의 기본 이름이 됩니다. 예를 들어, traypak.dpk라는 패키지 소스 파일을 컴파일하면 컴파일러는 bpltraypak.so라는 패키지를 생성합니다.

다음 테이블은 성공적인 패키지 컴파일이 생성하는 파일을 나열합니다.

**표 11.5 컴파일된 패키지 파일**

| 파일명 또는 확장자    | 내용                                                                                                         |
|---------------|------------------------------------------------------------------------------------------------------------|
| dcp           | 패키지 헤더를 포함하는 바이너리 이미지와 패키지의 모든 dpu 파일의 연결. 각 패키지에 대해 dcp 파일이 하나 생성됩니다. dcp에 대한 기본 이름은 dpk 소스 파일의 기본 이름입니다. |
| dpu           | 패키지에 포함된 유닛 파일에 대한 바이너리 이미지각 유닛 파일에 대해 필요한 경우 dpu 하나가 생성됩니다                                                |
| bplpackage.so | 런타임 패키지. 이 파일은 특별한 Kylix 특정 기능을 가진 공유 객체 파일입니다. 이름에서 <i>package</i> 는 dpk 소스 파일의 기본(base) 이름입니다.           |

컴파일할 때 패키지와 라이브러리 파일은 기본적으로 Tools|Environment Options 대화 상자의 Library 페이지에서 지정한 디렉토리에 생성됩니다. Package Editor의 Options 스피드 버튼을 클릭하여 Project Options 대화 상자가 표시되면 Directories/Conditionals 페이지를 변경하여 기본 설정을 오버라이드할 수 있습니다.

## 패키지 배포

다음 단원은 패키지 배포에 대한 권장 사항을 알려줍니다.

### 패키지를 사용하는 애플리케이션 배포

런타임 패키지를 사용하는 애플리케이션을 배포하는 경우 사용자가 애플리케이션이 호출하는 모든 라이브러리 파일과 더불어 애플리케이션의 실행 파일도 가지고 있는지 확인합니다. 라이브러리 파일이 실행 파일과 다른 디렉토리에 있으면 사용자의 경로를 통해 액세스할 수 있어야 합니다.

### 다른 개발자에 패키지 배포

런타임이나 디자인 타임 패키지를 다른 Kylix 개발자에게 배포하는 경우 .dcp와 .so 파일을 제공해야 합니다. .dpu 파일도 포함시킬 수 있습니다.

**참고** 관리자(root)가 새 패키지를 Kylix에 추가하면 사용자는 delphi60rc 파일을 .borland 디렉토리에서 제거해야만 패키지를 볼 수 있습니다.

## 국제적인 애플리케이션 만들기

이 장에서는 해외 시장에 배포하려는 애플리케이션 개발에 대해 설명합니다. 미리 계획을 잘 세우면 국내 시장뿐 아니라 해외 시장에서 애플리케이션을 사용할 수 있도록 하는데 필요한 시간과 코드를 줄일 수 있습니다.

### 국제화 및 지역화

---

해외 시장에 배포할 수 있는 애플리케이션을 만들려면 다음과 같은 중요한 두 가지 단계를 수행해야 합니다.

- 국제화
- 지역화

#### 국제화

---

국제화란 프로그램이 여러 로케일에서 작동할 수 있도록 하는 것입니다. *로케일*은 언어뿐만 아니라 대상 국가의 문화적 전통을 비롯한 사용자의 환경을 의미합니다. Linux는 언어와 국가의 쌍(예: English/ United States는 en\_US)으로 기술되는 많은 로케일을 지원합니다. LANG 환경 변수를 검색하여 운영 체제에서의 사용자 로케일 식별자를 얻을 수 있습니다.

대부분의 Kylix 애플리케이션에서는 애플리케이션을 실행하는 정확한 로케일을 정할 필요는 없지만 로케일 독립적인 방식으로 문자열을 처리해야 합니다.

#### 지역화

---

지역화란 특정 로케일에서 작동하도록 애플리케이션을 번역하는 것입니다. 지역화는 사용자 인터페이스를 번역하는 것 외에 기능의 사용자 지정을 포함하기도 합니다. 예를 들어, 재무 애플리케이션은 각 나라에 적합한 세법을 인식하도록 수정될 것입니다.

## 애플리케이션 국제화

국제화된 애플리케이션을 만드는 작업은 어렵지 않습니다. 다음 단계를 따릅니다.

- 1 코드에서 국제적인 문자 집합의 문자열을 처리할 수 있게 해야 합니다.
- 2 지역화로 인한 변경 내용을 수용할 수 있도록 사용자 인터페이스를 디자인해야 합니다.
- 3 지역화가 필요한 모든 리소스를 분리해야 합니다.

### 애플리케이션 코드 활성화

애플리케이션의 코드는 다양한 대상 로케일의 문자열을 처리할 수 있어야 합니다.

### 문자 집합

Linux 운영 체제는 UTF-8을 사용하여 운영 체제 커널에 전달된 파일 이름, 경로 및 기타 문자열을 인코딩합니다. 일반적으로 Linux 운영 체제는 로케일 특정 사항을 인식하지 못합니다.

일반적으로 일반 사용자 애플리케이션에서는 적절한 문자 집합을 화면에 표시하는 것보다 많은 특정 로케일 지원이 필요합니다. 로케일과 언어에는 문자열 데이터가 정렬되는 방법, 비교 시 동등하다고 판단하는 문자, 숫자 표시의 서식을 지정하는 방법 등에 여러가지 규칙이 적용됩니다. UTF-8은 위와 같은 규칙을 지원하지 않으므로 로케일과 로케일 문자 집합을 사용해야 합니다.

### 멀티바이트 문자 집합

아시아에서 사용되는 표의 문자 집합은 해당 언어의 문자와 1바이트(8비트) *char* 타입의 문자 사이에 단순한 1:1 매핑을 사용할 수 없습니다. 이러한 언어에는 1바이트 *char* 타입을 사용하여 표현할 수 없는 문자들이 많이 있습니다. 대신 멀티바이트 문자열의 문자는 한 문자당 하나 이상의 바이트를 사용할 수 있습니다. *AnsiString*은 싱글바이트와 멀티바이트 문자를 함께 혼합하여 포함할 수 있습니다.

모든 멀티바이트 문자 코드의 첫 바이트는 특정 문자 집합에 따라 예약된 범위를 사용합니다. 경우에 따라 두 번째 및 다음 바이트들은 각각의 1바이트 문자의 문자 코드와 동일하거나 멀티바이트 문자의 첫 번째 바이트에 대한 예약된 범위를 사용합니다. 따라서 문자열의 특정 바이트가 단일 문자를 나타내는지 또는 멀티바이트 문자인지를 구별할 수 있는 유일한 방법은 문자열을 처음부터 읽어서 예약된 범위에서 선행 바이트를 만났을 때 2바이트 이상의 문자로 분석해 보는 것입니다.

아시아 로케일에 대한 코드를 작성할 때는 멀티바이트 문자로 문자열을 구문 분석할 수 있는 함수를 사용하여 모든 문자열을 처리해야 합니다. Kylix는 또한 이러한 작업을 수행할 수 있도록 다음과 같은 여러가지 런타임 라이브러리 함수를 제공합니다.

|                     |               |                |
|---------------------|---------------|----------------|
| AdjustLineBreaks    | AnsiStrLIComp | CharToByteLen  |
| AnsiCompareFileName | AnsiStrLower  | ExtractFileDir |
| AnsiCompareText     | AnsiStrPos    | ExtractFileExt |

|                       |                       |                     |
|-----------------------|-----------------------|---------------------|
| AnsiCompareStr        | AnsiStrRScan          | ExtractFileName     |
| AnsiDequotedStr       | AnsiStrScan           | ExtractFilePath     |
| AnsiExtractQuotedStr  | AnsiStrUpper          | ExtractRelativePath |
| AnsiLastChar          | AnsiToUtf8            | FileSearch          |
| AnsiLowerCase         | AnsiUpperCase         | IsDelimiter         |
| AnsiLowerCaseFileName | AnsiUpperCaseFileName | IsPathDelimiter     |
| AnsiPos               | ByteToCharIndex       | LastDelimiter       |
| AnsiQuotedStr         | ByteToCharLen         | StrByteType         |
| AnsiStrComp           | ByteType              | StringReplace       |
| AnsiStrIComp          | ChangeFileExt         | Utf8ToAnsi          |
| AnsiStrLastChar       | AnsiStrLComp          | WrapText            |
| AnsiStrLComp          | CharToByteIndex       |                     |

**참고** glibc 2.2 이전 버전을 사용하여 Linux 시스템에서 애플리케이션을 실행할 경우 50K 이상의 긴 문자열을 멀티바이트 사용 가능 루틴에 전달하지 마십시오. glibc 2.2 이상 버전을 사용하는 경우에는 문자열 제한이 없습니다.

바이트 단위의 문자열 길이가 문자열의 문자 수와 같을 필요는 없습니다. 멀티바이트 문자를 반으로 나누어 문자열을 잘라내지 않도록 주의하십시오. 문자의 크기를 알 수 없으므로 함수 또는 프로시저의 매개변수로 문자를 전달하지 마십시오. 대신 문자나 문자열에 대한 포인터를 항상 전달하십시오.

## 와이드 문자

캡션, 설명, 텍스트 속성, 콤보 상자 등과 같은 표시 문자열 속성은 모두 와이드 문자열입니다.

**참고** Linux wchar\_t WideChar는 문자당 32비트입니다. 오브젝트 파스칼 WideChar가 지원하는 16비트 유니코드 표준은 Linux와 GNU 라이브러리가 지원하는 32비트 UCS 표준의 서브셋입니다. 파스칼 WideChar 데이터는 wchar\_t로 OS 함수에 전달되기 전에 문자당 32비트로 확장되어야 합니다.

Linux 커널은 4바이트 와이드 문자를 사용합니다. 커널은 파일 이름 등이 UTF-8로 인코딩된 문자열을 예상합니다. Kylix WideChar 및 WideString는 문자당 2바이트 유니코드로 UCS-4 규격의 서브셋입니다. 유니코드 2바이트 문자를 UCS 4바이트 문자로 번역하려면 앞에 2바이트의 0을 추가해야 합니다.

## 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 텍스트가 번역되어야 합니다. 영문 텍스트는 대부분의 경우 번역문보다 짧습니다. 늘어날 텍스트 문자열에 대한 여유 공간을 고려하여 텍스트를 표시하는 사용자 인터페이스의 요소를 디자인하십시오. 쉽게 긴 문자열을 표시할 수 있도록 텍스트를 표시하는 대화 상자, 메뉴, 상태 표시줄 및 기타 사용자 인터페이스 요소를 만드십시오. 표의 문자를 사용하는 언어에 없는 약자는 사용하지 마십시오.

짧은 문자열은 번역할 때 긴 문장보다 더 많이 늘어나는 경향이 있습니다. 표 12.1은 영문 문자열이 번역되었을 때 어느 정도 늘어나는지를 대략적으로 보여줍니다.

표 12.1 문자열 길이 추정

| 영문 문자열의 길이(문자 수) | 예상 증가량 |
|------------------|--------|
| 1-5              | 100%   |
| 6-12             | 80%    |
| 13-20            | 60%    |
| 21-30            | 40%    |
| 31-50            | 20%    |
| 50 이상            | 10%    |

## 그래픽 이미지

번역이 필요없는 이미지를 사용하려면 그래픽 이미지에 번역해야 하는 텍스트가 없어야 합니다. 이미지에 텍스트를 넣어야 하는 경우 이미지의 일부로 텍스트를 포함하는 것보다는 이미지 위에 투명한 배경과 함께 레이블 객체를 사용하는 것이 좋습니다.

그래픽 이미지를 만들 때 고려해야 할 다른 사항이 있습니다. 특정 문화에만 적합한 이미지는 사용하지 마십시오. 예를 들어, 우체통의 모양은 나라마다 다를 수 있습니다. 우세한 종교는 나라마다 다르므로 이들 국가를 대상으로 하는 애플리케이션에 종교적인 기호는 적절하지 않습니다. 색상조차도 문화마다 다른 상징적인 의미를 가질 수 있습니다.

## 형식 및 정렬 순서

애플리케이션에서 사용되는 날짜, 시간, 숫자 및 통화 형식은 대상 로케일에 적합하게 지역화되어야 합니다. 사용자 고유 형식 문자열을 지정하는 경우, 리소스화된 상수로 선언해서 지역화될 수 있도록 하십시오.

문자열이 정렬되는 순서도 나라마다 다릅니다. 많은 유럽 언어에서는 로케일에 따라 다르게 정렬되는 구분 표시들을 가지고 있습니다. 또한 일부 국가에서는 정렬할 때 2문자 조합을 단일 문자로 취급합니다. 예를 들어, 스페인어에서는 조합 문자 *ch*는 *c*와 *d* 사이의 고유한 단일 문자처럼 정렬됩니다. 간혹 독일어의 *eszett* 같은 단일 문자는 두 개의 개별 문자인 것처럼 정렬됩니다.

## 키보드 매핑

키 조합으로 단축키를 지정할 때 주의하십시오. US 키보드에 있는 일부 문자는 다른 국제적인 키보드로 쉽게 매핑할 수 없는 경우도 있습니다. 가능한 한 키보드의 모든 키를 사용하여 숫자 키와 단축키를 위한 기능 키를 사용하십시오.



## 리소스 분리

애플리케이션을 지역화할 때 가장 두드러진 작업은 사용자 인터페이스에 나타나는 문자열을 번역하는 것입니다. 어디서나 코드를 수정하지 않고 번역할 수 있는 애플리케이션을 만들려면 사용자 인터페이스의 문자열을 단일 모듈로 분리해야 합니다. Kylix는 메뉴, 대화 상자 및 비트맵에 대한 리소스를 포함하는 폼 파일을 자동으로 만듭니다.

사용자 인터페이스 요소뿐 아니라 사용자에게 나타나는 오류 메시지 같은 문자열도 분리해야 합니다. 문자열 리소스는 폼 파일에 포함되지 않습니다. 문자열 리소스에 대해서는 **resourcestring** 키워드를 사용하여 상수로 선언하여 분리해야 합니다. 리소스 문자열 상수에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오. 모든 리소스 문자열을 분리된 유닛 하나에 포함하는 것이 가장 좋습니다.

Kylix 리소스 문자열은 실행 파일에서 UTF-8(대개 문자당 1바이트)로 인코딩됩니다.

## 리소스 모듈 만들기

리소스를 분리하면 번역 처리가 단순화됩니다. 리소스 분리의 다음 단계는 리소스 모듈을 만드는 것입니다. 리소스 모듈이란 모든 리소스 및 코드가 아닌 프로그램에 대한 리소스 문자열만 포함하는 라이브러리입니다. 리소스 모듈을 사용하여 리소스 모듈을 교체함으로써 많은 번역을 단순화시키는 프로그램을 만들 수 있습니다.

프로그램에 대한 리소스 모듈을 만들려면 해당 프로젝트에 대한 **resourcestring** 문자열을 포함하는 파일을 만들고 연관된 폼을 포함하는 공유된 객체 파일에만 리소스에 대해 프로젝트를 생성합니다. 리소스는 실행 파일의 구분 영역으로 컴파일됩니다.

지원하려는 언어별로 각 번역에 대한 리소스 모듈을 만들어야 합니다. 각 리소스 모듈은 대상 로케일에 적합한 파일 이름을 가져야 합니다(예: US English의 경우 en\_US).

## 리소스 모듈 사용

애플리케이션을 구성하는 실행 파일, 공유 객체 파일 및 패키지는 필요한 모든 리소스를 포함합니다. 하지만, 이러한 리소스를 지역화된 버전으로 교체하려면 실행 파일, 공유 객체 파일 또는 패키지 파일과 동일한 이름을 가진 지역화된 리소스 모듈과 함께 애플리케이션을 배포해야 합니다.

애플리케이션이 시작될 때 LANG 환경 변수에서 로컬 시스템의 로케일을 확인합니다. 애플리케이션이 사용하는 실행 파일, 공유 객체 파일 또는 패키지 파일과 동일한 이름을 가진 리소스 모듈을 찾는 다음 발견한 공유 객체 파일의 확장자를 확인합니다. 리소스 모듈의 확장자가 시스템 로케일의 언어 및 국가와 일치하는 경우 애플리케이션은 실행, 공유 객체 파일 또는 패키지의 리소스 대신 해당 리소스 모듈의 리소스를 사용합니다. 리소스 모듈이 언어와 국가 모두 일치하지 않는 경우, 애플리케이션은 언어만 일치하는 리소스 모듈을 찾을 것입니다. 리소스 모듈 파일 이름 확장자가 해당 언어와 일치하지 않는 경우, 애플리케이션은 실행 파일, 공유 객체 파일 또는 패키지로 컴파일된 리소스를 사용합니다.

적절한 리소스 모듈만 제공하면 애플리케이션이 실행 중인 시스템의 로케일에 맞게 자동으로 적용하는 단일 애플리케이션을 만들 수 있습니다.

## 애플리케이션 지역화

애플리케이션이 일단 국제화되면, 배포하려는 다른 해외 시장에 대한 지역화된 버전을 만들 수 있습니다.

리소스가 폼 파일을 포함하는 리소스 모듈로 분리되어 있으면 IDE에서 폼을 열고 관련 속성을 번역한 다음, 애플리케이션을 컴파일할 수 있습니다. ~/kylix/bin 에 있는 *resbind* 명령줄 유틸리티를 이용하여 필요한 모든 리소스를 추출할 수 있습니다.

*Resbind*는 애플리케이션에서 Borland 리소스를 추출하고 해당 리소스를 포함하는 공유 객체 파일을 만듭니다. 그런 다음 런타임에 리소스를 동적으로 연결하거나 애플리케이션이 실행 중인 로컬 시스템에서 애플리케이션이 환경 변수를 확인하도록 할 수 있습니다.

*Resbind*는 ELF 파일 및 경우에 따라 하나 이상의 리소스 파일에서 리소스를 읽습니다. ELF 파일은 Linux i386 실행 파일 또는 공유 객체 파일입니다. *Resbind*는 Windows 32비트, 16비트 형식 및 Linux용 Borland 리소스 파일 형식의 리소스 파일을 사용합니다.

*Resbind*는 리소스를 인쇄하고 새 리소스 파일에 해당 리소스를 복사하거나 다음 구문을 사용하여 리소스 파일에서 읽은 리소스로 교체된 리소스를 사용하여 ELF 파일의 사본에 해당 리소스를 복사할 수 있습니다.

```
resbind [option]... ELF-FILE RESOURCE-FILE...
```

다음은 표는 *resbind* 옵션을 나열한 것입니다.

**표 12.2** Resbind 옵션

| 옵션        | 설명                                            |
|-----------|-----------------------------------------------|
| -h        | 도움말을 표시 및 종료                                  |
| -V        | 버전 정보 출력 및 종료                                 |
| -f FMT    | 리소스 파일 출력 형식 제어. FMT는 'w32'(기본값) 또는 'borland' |
| -l FILE   | -s에 대해 사용하는 동적 링커, 기본값 /lib/ld-linux.so.2     |
| -o FILE   | FILE에 업데이트된 ELF-FILE 작성                       |
| -p        | 조합된 리소스 인쇄                                    |
| -r FILE   | 리소스 파일 FILE에 조합된 리소스 작성                       |
| -s FILE   | 리소스 공유 객체 FILE에 조합된 리소스 작성                    |
| -S SONAME | -s에 대한 soname 정의                              |

*Resbind*는 애플리케이션에서 편집용 리소스 파일로 리소스를 복사하는 데 사용될 수 있습니다. 나중에 해당 애플리케이션으로 편집된 리소스를 다시 복사할 수 있습니다.

예를 들어, 다음 명령은 리소스를 "program"이라고 하는 ELF 파일에서 "program.res"라고 하는 Windows 32비트 리소스 파일로 복사합니다.

```
$ resbind -r program.res program
```

그런 다음 "program.res"에 있는 편집된 리소스를 "program"이라고 하는 ELF 파일과 조합할 수 있고 해당 결과를 "program.new"에 기록할 수 있습니다.

```
$ resbind -o program.new program program.res
```

"program.res"에 포함되지 않은 리소스는 변경되지 않은 채로 남아 있습니다.

리소스 공유 객체에 조합된 리소스를 작성할 수도 있습니다.

```
$ resbind -s program-res.so program program.res
```



## 애플리케이션 배포

Kylix 애플리케이션 테스트 결과 이상 없이 실행된다면 배포할 수 있습니다. 즉 다른 사람이 애플리케이션을 설치해서 실행하게 할 수 있습니다. 애플리케이션을 다른 컴퓨터에 배포하여 애플리케이션이 완전하게 기능하려면 많은 단계가 필요합니다. 주어진 애플리케이션이 요구하는 단계는 애플리케이션의 종류에 따라 다양합니다. 다음 단원에서 애플리케이션 배포를 위한 단계를 설명합니다.

- 일반 애플리케이션 배포
- 데이터베이스 애플리케이션 배포
- 웹 애플리케이션 배포
- 다양한 호스트 환경을 위한 프로그래밍
- 소프트웨어 사용권 요구 사항

### 일반 애플리케이션 배포

---

Linux에서 애플리케이션은 보통 다음 방법 중 하나를 사용하여 배포합니다.

- RPM Package Manager (RPM) 같은 툴 사용
- tar 파일(tar.gz) 또는 gnu zip 파일(gzip) 생성
- Loki Entertainment Software의 오픈 소스 프로그램인 Setup Graphic Installer 같은 Linux용으로 개발된 설치 프로그램 사용

각각의 방법에는 장단점이 있으며 애플리케이션의 종류와 애플리케이션 실행에 필요한 파일에 따라 사용할 방법을 결정합니다. 간단한 프로그램은 tar 파일로 배포할 수 있습니다. 그러나 좀 더 복잡한 프로그램은 RPM 또는 설치 프로그램을 사용하여 배포해야 합니다.

RPM은 소프트웨어 패키지의 설치, 설치 제거, 업그레이드, 검증 및 빌드에 사용됩니다. RPM은 파일의 아카이브와 이름, 버전, 간단한 설명 등의 애플리케이션 정보를 생성합니다. RPM은 Linux와 UNIX의 여러 버전에서 사용할 수 있으며 소프트웨어 배포에 많이 사용됩니다. RPM을 사용하는 경우 *패키지*라는 용어는 하나의 .rpm 파일을 나타내거나 RPM을 사용하여 설치되는 소프트웨어를 포함하고 있는, 미리 모아놓은 유닛을 나타냅니다. RPM은 셸에서 실행되는 명령입니다.

```
rpm -i [options] [packages]
```

다른 설치 프로그램도 Linux의 애플리케이션 배포에 사용할 수 있습니다. 예를 들어 Setup Graphic Installer는 개발자가 모든 종류의 애플리케이션에 대해 쉬운 설치를 만드는 그래픽 설치 유틸리티입니다.

## 배포 문제

---

실행 파일 외에 애플리케이션은 공유 객체 파일, 초기화 파일, 패키지 파일, helper 애플리케이션 같은 많은 지원 파일이 필요할 수 있습니다. 애플리케이션의 파일을 컴퓨터에 복사하는 과정과 필요한 설정값을 만드는 과정을 설치 프로그램으로 자동화할 수 있습니다. 다음은 거의 모든 종류의 애플리케이션에 공통적인 주요 배포 관련 사항입니다.

- 설치 프로그램 사용
- 애플리케이션 파일 식별

데이터베이스에 액세스하는 Kylix 애플리케이션과 웹에서 실행되는 애플리케이션은 일반 애플리케이션에 적용되는 단계 외에 추가 설치 단계가 필요합니다. 데이터베이스 애플리케이션 설치에 대한 자세한 내용은 13-4 페이지의 "데이터베이스 애플리케이션 배포"를 참조하십시오. 웹 애플리케이션 설치에 대한 자세한 내용은 13-6 페이지의 "웹 애플리케이션 배포"를 참조하십시오.

많은 Kylix 애플리케이션은 다음을 포함하는 공유 라이브러리의 특정 집합에 따라 다릅니다.

- libqt.so.2 (Qt 라이브러리)
- libqtintf.so.2 (Qt 인터페이스 라이브러리)
- libborunwind.so.6 (공유 예외 처리)
- 런타임 패키지(예: baseclx)
- 데이터베이스 DBxpress 드라이버(libsqldb.so, libsqlora.so 등)

성공적으로 애플리케이션을 배포하려면 필요한 모든 라이브러리가 포함되고 런타임에 애플리케이션에 대해 액세스할 수 있는 위치에 있어야 합니다. 필요한 라이브러리는 애플리케이션에 따라 다릅니다. 예를 들어 CLX 컴포넌트를 사용하는 경우 위에 나열된 두 Qt 라이브러리를 배포해야 합니다.

라이브러리의 액세스 가능성은 본래 Linux 시스템의 두 설정값에 의해 제어됩니다. 필요한 라이브러리를 포함하는 디렉토리를 나열하는 항목은 다음 두 곳 중 하나에 위치합니다.

- /etc/ld.so.conf  
또는
- LD\_LIBRARY\_PATH

일반적으로 /etc/ld.so.conf는 이미 나열된 많은 디렉토리를 가지고 있습니다. 그러므로 단순히 파일을 이미 나열된 디렉토리에 복사하는 것만으로 충분합니다.

Windows 2000과 비교해 볼 때, /etc/ld.so.conf 또는 이 파일에 나열된 디렉토리의 사용은 공유 라이브러리를 \WinNT\System32 폴더에 복사하는 것과 동일합니다. LD\_LIBRARY\_PATH의 사용은 애플리케이션 특정 검색 경로를 지정하는 것과 동일합니다.

라이브러리 경로 환경 변수를 사용하면 애플리케이션이 로드하는 공유 객체 파일을 포함하는 디렉토리를 LD\_LIBRARY\_PATH에 포함해야 합니다. LD\_LIBRARY\_PATH는 런타임 공유 라이브러리 로더를 공유 라이브러리 검색 시 찾는 추가 디렉토리 집합에 제공하도록 설정하는 환경 변수입니다. 여러 디렉토리를 콜론(:)으로 분리하여 나열할 수 있습니다. 예를 들어 애플리케이션을 /usr/local/app에 배포하고 공유 객체 /usr/local/app/mylib.so를 사용하는 경우 /usr/local/app을 LD\_LIBRARY\_PATH에 포함할 수 있습니다.

## 설치 프로그램 사용

실행 파일로만 구성된 간단한 Kylix 애플리케이션은 대상 컴퓨터에 설치하기 쉽습니다. 실행 파일을 컴퓨터에 복사하기만 하면 됩니다. 그러나 여러 파일을 포함하는 복잡한 애플리케이션은 확장된 설치 과정이 필요합니다. 이런 애플리케이션은 설치 전용 프로그램이 필요합니다.

## 애플리케이션 파일 식별

실행 파일 외에도 많은 파일이 애플리케이션과 함께 배포되어야 합니다.

다음과 같은 파일이 애플리케이션과 함께 배포되어야 합니다.

- 애플리케이션 파일
- 공유 객체 파일
- 초기화 파일
- 패키지 파일
- Helper 애플리케이션
- 도움말 파일
- 데이터베이스 파일

애플리케이션에서 필요로 하는 모든 파일을 확인하려면 다음을 입력합니다.

```
ldd <application_name>
```

운영 체제 파일을 비롯한 애플리케이션이 로드를 시도한 모든 파일이 출력됩니다. 운영 체제 관련 파일은 배포할 필요가 없지만 Kylix 파일이나 애플리케이션에서 필요로 하는 파일 및 참조하는 라이브러리는 포함해야 합니다.

## 패키지 파일

애플리케이션이 런타임 패키지를 사용하는 경우 패키지 파일은 애플리케이션과 함께 배포되어야 합니다. 일반적으로 패키지 파일은 다른 공유 객체와 함께 lib 디렉토리에 있습니다. 이 디렉토리는 공용 디렉토리로 사용되기 때문에 여러 애플리케이션이 파일의 단일 인스턴스에 액세스할 수 있습니다. 사용자가 만든 패키지의 경우 애플리케이션과 같은 디렉토리에 설치하는 것이 좋습니다. .so 파일만 배포하면 됩니다.

다른 개발자에게 패키지를 배포하는 경우 .so 파일과 .dcp 파일을 제공해야 합니다.

## Helper 애플리케이션

Helper 애플리케이션은 별도의 프로그램으로 이 프로그램이 없으면 Kylix 애플리케이션의 부분적인 기능 또는 기능 전체를 사용할 수 없습니다. Helper 애플리케이션은 Borland가 제공하는 프로그램이거나 타사(third-party) 제품일 수 있습니다.

helper 프로그램을 필요로 하는 애플리케이션이라면 가능하면 이 프로그램을 애플리케이션과 함께 배포해야 합니다. helper 프로그램의 배포는 재배포 사용권 계약에 의합니다. 자세한 내용은 helper 프로그램과 함께 제공된 설명서를 참조하십시오.

## 공유 객체 파일 위치

애플리케이션과 동일한 디렉토리에 하나의 애플리케이션만이 사용하는 공유 객체 파일을 설치할 수 있습니다. 많은 애플리케이션이 사용하는 공유 객체 파일은 모든 애플리케이션이 액세스할 수 있는 위치에 설치해야 합니다. 여러 애플리케이션이 공유하는 공유 객체 파일을 Linux 시스템에서 찾기 위한 일반적 규칙은 /bin 디렉토리에 이 파일을 두는 것입니다. 애플리케이션에서 로드하는 모든 공유 객체 파일은 LD\_LIBRARY\_PATH에 있어야 합니다.

## 데이터베이스 애플리케이션 배포

---

데이터베이스에 액세스하는 애플리케이션은 애플리케이션의 실행 파일을 호스트 컴퓨터에 복사하는 것 외에 특별한 설치 고려사항이 있습니다. 별도의 데이터베이스 엔진, 애플리케이션의 실행 파일에 연결될 수 없는 파일들에 의해 데이터베이스 액세스가 자주 처리됩니다. 미리 생성되지 않은 경우 데이터 파일은 애플리케이션에서 사용할 수 있게 해야 합니다. 애플리케이션을 구성하는 파일은 보통 여러 컴퓨터에 설치되기 때문에 다계층 데이터베이스 애플리케이션은 설치에 추가적인 처리가 필요합니다.



Kylix 애플리케이션은 *dbExpress*를 사용하여 데이터베이스에 연결합니다. *dbExpress*는 SQL 데이터베이스 서버에 빠른 액세스를 제공하는 데이터베이스 드라이버 집합입니다. 각 지원 데이터베이스에 대해, *dbExpress*는 일정한 *dbExpress* 인터페이스 집합에 적합한 서버 특정 소프트웨어를 사용하는 드라이버를 제공합니다. 사용자가 직접 만든 애플리케이션을 배포하는 경우 빌드한 애플리케이션 파일에는 단일 공유 객체(서버 특정 드라이버)만 포함해야 합니다. 여러 데이터베이스를 지원하는 경우 각 데이터베이스에 대한 공유 객체 파일이 필요합니다.

클라이언트 데이터셋 (*TClientDataSet* 또는 *TSQLClientDataSet*)을 사용하는 경우나 *TDataSetProvider*를 사용하는 경우 *midas.so*도 배포해야 합니다.

## 데이터베이스에 연결

데이터베이스 연결을 열려면 사용하는 드라이버와 그 드라이버에 전달할 연결 매개변수 집합 모두를 식별해야 합니다. 드라이버는 설치된 *dbExpress* 드라이버의 이름인 *DriverName* 속성으로 식별됩니다. 지원 드라이버는 InterBase, MySQL, Oracle 또는 DB2 같은 데이터베이스를 포함합니다. 다음 두 파일이 드라이버 이름에 연결됩니다.

- *dbExpress* 드라이버. 이름이 *libsqlib.so*, *libsqlmys.so*, *libsqlora.so* 또는 *libsqldb2.so*인 공유 객체 파일.
- 데이터베이스 벤더가 제공하는 클라이언트 측 공유 객체 파일.

이들 두 공유 객체 파일과 데이터베이스 이름 사이의 관계는 *dbxdrivers*라는 파일에 저장됩니다. *dbExpress*를 통해서도 *dbxconnections*라는 파일에 데이터베이스 연결 이름을 정의하고 저장합니다. 이는 다른 데이터베이스에 액세스하는 시스템에 데이터베이스 애플리케이션을 배포할 수 있게 합니다. 런타임에 데이터베이스 정보를 로드하는 경우 *dbxdrivers*와 *dbxconnections* 파일을 배포해야 합니다. 데이터베이스 연결에 대한 자세한 내용은 19-2 페이지의 "연결 제어"에서 다룹니다.

드라이버와 연결 파일을 실행 파일과 같은 디렉토리 또는 홈 디렉토리 아래의 *.borland* 디렉토리에 설치합니다. 개발하고 있는 여러 애플리케이션 사이에 *dbExpress* 드라이버 및 구성 파일을 공유하려면 *.borland* 디렉토리에 둡니다. 이 디렉토리는 Kylix의 *dbxdrivers*와 *dbxconnections* 파일이 있는 디렉토리나 동일한 디렉토리입니다.

## 구성 파일 업데이트

공유 구성 파일을 사용하고 있는 경우 설치 프로그램은 제품과 함께 제공되는 MergeIniFile 유틸리티를 사용하여 기존 드라이버의 내용이나 연결 파일을 새 파일이나 업데이트된 파일과 합병할 수 있습니다. 구문은 다음과 같습니다.

```
MergeIniFile( SourceIniFile, TargetIniFile, FOverwrite )
```

MergeIniFile 유틸리티는 추가된 섹션의 계수를 반환합니다. FOverwrite는 기본적으로 *False*입니다. *True*로 설정하면 이미 존재하는 경우에도 섹션을 추가하며 SourceIniFile에서 찾은 섹션의 모든 설정값을 덮어씁니다. SourceIniFile에 없는 섹션은 변경되지 않고 남으며 구성 파일에 없는 설정값은 삭제되지 않습니다.

## 웹 애플리케이션 배포

일부 Kylix 애플리케이션은 Apache 공유 객체 파일과 CGI 애플리케이션에서 같이 웹에서 실행되도록 디자인됩니다. CGI 애플리케이션은 CGI 인터페이스를 지원하는 웹 서버에 배포할 수 있습니다. 웹 애플리케이션 설치를 위한 단계는 애플리케이션의 파일을 웹 서버에 배포하는 것만 다를 뿐 일반적인 애플리케이션의 단계와 동일합니다. 일반적인 애플리케이션 설치에 대한 자세한 내용은 13-1 페이지의 "일반 애플리케이션 배포"를 참조합니다.

웹 애플리케이션 배포에 대한 특별한 주의 사항은 다음과 같습니다.

- 데이터베이스 애플리케이션의 경우 *dbExpress* 드라이버는 웹 서버에 애플리케이션 파일과 함께 설치됩니다.
- 애플리케이션 파일이나 데이터베이스 파일에 대한 액세스가 불가능할 만큼 디렉토리의 보안 수준이 너무 높지 않아야 합니다.
- 애플리케이션을 포함한 디렉토리는 읽기와 실행 속성을 가져야 합니다.
- Apache 구성 파일에서 CGI 애플리케이션의 위치를 지정해야 합니다. 이 디렉토리는 읽기 및 실행 속성을 가져야 합니다.
- 애플리케이션은 데이터베이스나 기타 파일에 액세스하기 위해 하드 코딩된 경로를 사용해서는 안 됩니다.

### Apache에 배포

WebBroker는 Apache 버전 1.3.9 이상을 지원합니다. Apache를 사용하면 `httpd.conf` 라는 Apache 구성 파일에 적절한 지시어를 설정해야 합니다. `SetEnv` 지시어는 `LD_LIBRARY_PATH` 변수를 `export`하는 데 설정되고 런타임에 애플리케이션이 요구하는 모든 공유 객체의 위치를 가리키도록 설정되어야 합니다. 그러므로 `httpd.conf`는 다음을 포함해야 합니다.

```
SetEnv LD_LIBRARY_PATH <directory>
```

여기서 `<directory>`는 Kylix 설치 디렉토리의 전체 경로입니다.

물리적 디렉토리는 프로그램의 실행을 허용하도록 설정된 `ExecCGI` 옵션을 가져야 합니다. `httpd.conf`는 다음과 유사한 줄을 포함해야 합니다.

```
<Directory "/<directory>/httpd/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

`HomeEnv` 변수를 또한 `httpd.conf` 파일에 설정해야 합니다. 데이터베이스 애플리케이션의 경우 *dbExpress*는 홈 디렉토리에 `.borland` 디렉토리가 있어야 합니다.

Apache는 `httpd.conf` 파일의 `User` 지시어에 지정된 숫자 내에서 서버에서 지역적으로 실행합니다.

## 다양한 호스트 환경을 위한 프로그래밍

Linux 환경의 특성 때문에 사용자 기본 설정이나 구성을 변경하는 요인이 많이 있습니다. 다음 요인은 다른 컴퓨터에 배포되는 애플리케이션에 영향을 줍니다.

- 화면 해상도와 색상 수
- 글꼴
- Helper 애플리케이션
- 공유 객체 파일 위치

### 화면 해상도와 색상 수

데스크탑의 크기와 컴퓨터에서 사용 가능한 색상 수는 하드웨어와 설치된 Xserver에 따라 구성됩니다. 이런 속성은 개발 컴퓨터와 배포 컴퓨터가 다를 수도 있습니다.

다른 화면 해상도를 위해 구성된 컴퓨터의 애플리케이션의 모습(창, 객체 및 글꼴 크기)은 다양한 방법으로 처리할 수 있습니다.

- 사용자가 가지는 가장 낮은 해상도(보통, 640x480)로 애플리케이션을 디자인합니다. 호스트 컴퓨터의 화면 표시에 비례하여 동적으로 객체의 크기를 조절하는 어떠한 특별한 작업을 하지 않습니다. 시각적으로 해상도가 더 높게 설정될수록 객체는 더 작게 보입니다.
- 개발 컴퓨터에서 임의의 화면 해상도를 사용하여 디자인하고 런타임에 개발 컴퓨터와 배포 컴퓨터 사이의 화면 해상도 차이에 비례하여 모든 폼과 객체의 크기를 동적으로 조정합니다(화면 해상도 차이 비율).
- 개발 컴퓨터에서 임의의 화면 해상도를 사용하여 디자인하고 런타임에 애플리케이션의 폼 크기만 동적으로 조정합니다. 폼의 비주얼 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 컨트롤이 런타임에 동적으로 크기가 조정되지 않으면 애플리케이션의 요소를 가장 낮은 해상도로 디자인합니다. 그렇지 않으면 개발 컴퓨터보다 낮은 화면 해상도로 구성된 컴퓨터에서 실행하는 애플리케이션의 폼은 화면 경계를 오버랩할 수 있습니다.

예를 들어 개발 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 컨트롤을 동적으로 크기 조정하는 경우 가능한 모든 화면 해상도에서 최적의 애플리케이션 모습이 되도록 크기 조정 처리의 모든 측면을 고려합니다. 애플리케이션의 비주얼 요소를 동적으로 크기 조정할 때 고려하는 요인은 다음과 같습니다.

- 폼과 비주얼 컨트롤의 크기를 개발 컴퓨터의 화면 해상도와 애플리케이션이 설치된 컴퓨터의 화면 해상도를 비교하여 계산된 비율로 조정합니다. 개발 컴퓨터의 화면 해상도의 높이 또는 너비를 픽셀 단위의 치수로 표현하는 상수를 사용합니다. *TScreen.Height* 또는 *TScreen.Width* 속성을 사용하여 런타임에 사용자의 컴퓨터에 대해서 동일한 치수를 가져옵니다. 두 컴퓨터의 화면 해상도 사이의 차이 비율을 계산하기 위해 개발 컴퓨터의 값을 사용자 컴퓨터의 값으로 나눕니다.
- 폼에 있는 요소의 크기와 위치를 줄이거나 늘려 애플리케이션(폼과 컨트롤)의 비주얼 요소를 크기 조정합니다. 이 크기 조정은 개발 컴퓨터와 사용자 컴퓨터 사이의 화면 해상도 차이에 비례합니다. *CustomForm.Scaled* 속성을 *True*로 설정하고 *TWidgetControl.ScaleBy* 메소드를 호출하여 폼의 비주얼 컨트롤의 크기를 자동으로 조정하고 재배치합니다. *ScaleBy* 메소드가 폼의 높이와 너비는 변경하지는 않습니다. *Height*와 *Width* 속성의 현재 값을 화면 해상도 차이 비율로 곱하여 수동으로 폼의 높이와 너비를 변경합니다.
- 폼의 컨트롤은 *TWidgetControl.ScaleBy* 메소드로 자동으로 크기를 조정하는 대신에 순환문에서 각 비주얼 컨트롤을 참조하고 치수와 위치를 설정하여 수동으로 크기 조정할 수 있습니다. 비주얼 컨트롤의 *Height* 및 *Width* 속성 값을 화면 해상도 차이 비율로 곱합니다. *Top* 및 *Left* 속성값을 같은 비율로 곱하여 화면 해상도 차이에 비례하여 비주얼 컨트롤을 재배치합니다.
- 사용자 컴퓨터의 해상도보다 더 높은 화면 해상도로 구성된 컴퓨터에서 애플리케이션이 디자인된 경우 글꼴 크기는 비주얼 컨트롤의 비례적인 크기 조정 과정에서 줄어듭니다. 디자인 타임의 글꼴 크기가 너무 작으면 런타임에 크기 조정된 글꼴은 읽을 수가 없습니다. 예를 들어 폼의 기본 글꼴 크기는 8입니다. 개발 컴퓨터의 화면 해상도가 1024x768이고 사용자의 컴퓨터가 640x480인 경우 비주얼 컨트롤 치수는 0.625 ( $640 / 1024 = 0.625$ )로 감소됩니다. 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 컨트롤은 컨트롤의 글꼴 크기가 변경될 때 동적으로 크기 조정됩니다. 이것은 폼과 컨트롤이 동적으로 크기가 조정되면 배포된 애플리케이션에 영향을 줍니다. 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 보상할 수 있습니다.
- 캔버스에 직접 그리는 것처럼 명시적인 픽셀 좌표 사용은 피합니다. 대신 비율 비례에 의한 조정으로 개발 컴퓨터와 사용자 컴퓨터 사이의 화면 해상도 차이를 수정합니다. 예를 들어 애플리케이션이 10픽셀 높이와 20픽셀 너비로 캔버스에 직사각형을 그리는 경우 화면 해상도 차이 비율에 10과 20을 곱합니다. 이렇게 하면 다른 화면 해상도에서도 같은 크기로 나타나게 합니다.

## 다양한 색상 수 지원

색상 표현 능력이 서로 다르게 설정된 모든 배포 컴퓨터에 대한 가장 안전한 방법은 가능한 최소한의 색상 수로 그래픽을 사용하는 것입니다. 이 방법은 전형적인 16 색상 그래픽을 사용해야 하는 컨트롤 glyphs에 대해 적합합니다. 그림을 표시하기 위해서는 다른 해상도와 다른 색상 수를 이용하는 이미지 복사본을 여러 개 제공하거나 애플리케이션에 대해 요구되는 조건으로 최소 해상도와 색상 수를 나타냅니다.

## 글꼴

Linux에는 글꼴의 표준 집합이 포함되어 있습니다. 다른 컴퓨터에 배포할 애플리케이션을 디자인할 때 일부 컴퓨터는 표준 집합의 글꼴만 가지고 있다는 점을 염두에 두어야 합니다.

애플리케이션에 사용되는 텍스트 컴포넌트는 모든 배포 컴퓨터에서 사용 가능한 글꼴을 모두 사용해야 합니다.

애플리케이션에서 비표준 글꼴을 꼭 사용해야 하는 경우 해당 글꼴을 애플리케이션과 함께 배포해야 합니다. 설치 프로그램이나 애플리케이션은 배포 컴퓨터에 글꼴을 설치해야 합니다. 타사(third-party) 글꼴의 배포는 글꼴 생성자가 부과하는 제약 사항을 지켜야 합니다.

## 소프트웨어 사용권 요구 사항

Kylix 애플리케이션에 관련된 일부 파일의 배포는 제한 사항이 있거나 전혀 재배포할 수 없습니다. 다음 문서는 이런 파일의 배포에 관한 법적 규정을 설명합니다.

- deploy.txt
- README
- 사용권 계약서
- GPL 사용권 계약
- 타사(third-party) 제품 설명서

## Deploy.txt

Deploy.txt는 다양한 컴포넌트와 유틸리티, Kylix 애플리케이션의 일부 또는 관련된 다른 제품의 배포에 관련된 일부 법률적인 내용이 들어 있습니다. Deploy.txt는 메인 Kylix 디렉토리에 설치되는 텍스트 파일로 다음과 같은 주제를 다룹니다.

- 실행 파일, 공유 객체 파일 및 패키지 파일
- 컴포넌트 및 디자인 타임 패키지
- 샘플 이미지
- 클라이언트 데이터셋(*TDataSetProvider*, *TClientDataSet* 또는 *TSQLDataSet*)

## README

---

README 파일은 컴포넌트, 유틸리티 또는 기타 제품의 재배포 권리에 관련된 정보를 비롯한 Kylix에 대한 최종 정보를 포함합니다. README는 메인 Kylix 디렉토리에 설치되는 텍스트 파일입니다.

## 사용권 계약서

---

인쇄된 문서인 Kylix 사용권 계약서는 Kylix에 관련된 기타 법적 권리와 의무를 규정합니다.

## GPL 사용권 계약

---

GPL (General Public License) 계약은 CLX를 사용해서 오픈 소스 애플리케이션을 개발할 때의 오픈 소스 사용권에 관한 정보를 포함하고 있습니다.

## 타사(third-party) 제품 설명서

---

타사 컴포넌트, 유틸리티, helper 애플리케이션, 데이터베이스 엔진 및 기타 제품에 대한 재배포 권리는 제품을 공급하는 협력 업체가 가지고 있습니다. Kylix 애플리케이션에 포함된 제품의 재배포에 관한 정보는 배포에 앞서 제품 설명서나 협력 업체에 문의하십시오.

# 데이터베이스 애플리케이션 개발

"데이터베이스 애플리케이션 개발"에 포함된 장에서는 Kylix 데이터베이스 애플리케이션을 만드는 데 필요한 개념 및 기술을 설명합니다.

**참고** Kylix의 일부 버전에서는 데이터베이스 컴포넌트를 사용할 수 없습니다.





## 데이터베이스 애플리케이션 디자인

데이터베이스 애플리케이션은 사용자로 하여금 데이터베이스에 저장되어 있는 정보를 다룰 수 있게 해줍니다. 데이터베이스는 그러한 정보에 대한 구조를 제공하여 각기 다른 애플리케이션 간에 정보를 공유할 수 있게 합니다.

Kylix는 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 때 사용 가능한 여러 가지 선택 사항이 있습니다. 이 장에서는 사용 가능한 다양한 아키텍처에 대해 설명합니다. 아키텍처의 타입에 따른 장단점을 고려하여 사용자의 요구에 가장 적합한 방법을 선택하십시오. 단순한 방법으로 시작한 다음에 나중에 보다 강력한 아키텍처로 조정할 수도 있습니다. 동일한 컴포넌트가 대부분의 아키텍처에서 사용되기 때문에 Kylix 데이터베이스 지원은 이러한 조정을 쉽게 만들어 줍니다.

### 데이터베이스 사용

컴포넌트 팔레트의 dbExpress 페이지에 있는 컴포넌트를 사용하면 애플리케이션을 데이터베이스와 연결하여 데이터를 읽어올 수 있습니다. 이러한 컴포넌트는 *dbExpress*를 사용하여 데이터베이스 정보에 액세스합니다. 이것은 정보를 다시 패키지화하고 버퍼링하는 다른 컴포넌트에서 이러한 컴포넌트를 사용 가능하도록 만들거나 최종 사용자에게 정보를 표시합니다.

*dbExpress*는 다른 타입의 데이터베이스에 대한 드라이버의 집합입니다. 모든 타입의 데이터베이스가 정보를 저장하는 테이블을 포함하지만 이러한 데이터베이스 타입은 다음과 같은 추가 기능을 지원합니다.

- 데이터베이스 보안
- 트랜잭션
- 참조 무결성, 내장 프로시저 및 트리거

## 데이터베이스의 타입

관계형 데이터베이스 서버는 정보를 저장하는 방법 및 다중 사용자가 동시에 해당 정보를 액세스할 수 있도록 하는 다양한 방법을 가집니다. Kylix는 다음과 같은 두 가지 타입의 관계형 데이터베이스 서버를 지원합니다.

- **원격 데이터베이스 서버**는 일반적으로 개별적인 시스템에 상주합니다. 경우에 따라 원격 데이터베이스 서버의 데이터는 단일 시스템에 상주하지 않고 여러 서버에 분산됩니다. 원격 데이터베이스 서버는 정보를 저장하는 방법은 다양하지만 클라이언트에 공통적인 인터페이스를 제공합니다. 이 공통적인 인터페이스는 SQL (Structured Query Language)입니다. SQL을 사용하여 액세스하기 때문에 때로 SQL 서버라고도 합니다. (또 다른 이름은 원격 데이터베이스 관리 시스템 또는 RDBMS입니다.) SQL을 구성하는 일반적인 명령에 추가하여 대부분의 원격 데이터베이스 서버는 "조금씩 변경된" 고유한 SQL을 지원합니다. SQL 서버의 예로는 InterBase, Oracle, DB2 및 MySQL이 있습니다.
- **로컬 데이터베이스**는 로컬 드라이브 또는 로컬 영역 네트워크에 상주합니다. 로컬 데이터베이스에는 종종 데이터를 액세스하기 위한 전용 API가 있습니다. 여러 사용자가 로컬 데이터베이스를 공유할 때 파일 기반 잠금 메커니즘을 사용합니다. 이 메커니즘 때문에 로컬 데이터베이스는 때로 파일 기반 데이터베이스라고도 합니다. Kylix는 다음 두 가지 타입의 로컬 데이터베이스를 지원합니다. InterBase 서버의 로컬 버전인 Local InterBase 및 클라이언트 데이터셋에 저장된 데이터를 위한 전용 파일 형식.

로컬 데이터베이스를 사용하는 애플리케이션은 애플리케이션과 데이터베이스가 하나의 파일 시스템을 공유하기 때문에 **단일 계층 애플리케이션**이라고 합니다. 원격 데이터베이스 서버를 사용하는 애플리케이션은 애플리케이션과 데이터베이스가 독립 시스템 (또는 계층)에서 작동하므로 **2계층 애플리케이션** 또는 **다계층 애플리케이션**이라고 합니다.

사용할 데이터베이스 타입 선택은 여러 요인에 따라 다릅니다. 예를 들어, 사용자의 데이터가 기존 데이터베이스에 이미 저장되어 있을 수도 있습니다. 애플리케이션이 사용하는 데이터베이스 테이블을 만드는 경우, 다음 사항을 고려해야 할 것입니다.

- 얼마나 많은 사용자가 이 테이블을 공유할 것인가? 원격 데이터베이스 서버는 동시에 여러 사용자가 액세스할 수 있도록 고안되었습니다. 원격 데이터베이스 서버는 트랜잭션이라는 메커니즘을 통해 다중 사용자를 지원합니다. 또한 Local InterBase 같은 일부 로컬 데이터베이스는 트랜잭션을 지원하지만 대부분의 로컬 데이터베이스는 파일 기반 잠금 메커니즘만 제공하고 클라이언트 데이터셋 파일 같은 일부 로컬 데이터베이스는 다중 사용자를 지원하지 않습니다.
- 테이블은 얼마나 많은 데이터를 보유할 것인가? 원격 데이터베이스 서버는 로컬 데이터베이스보다 많은 데이터를 보유할 수 있습니다. 일부 데이터베이스 서버가 다른 조건 (예: 빠른 업데이트)에 대해 최적화되어 있는 반면에 다른 원격 데이터베이스 서버는 대량의 데이터를 보관하도록 고안되어 있습니다.

- 데이터베이스에서 어떤 성능(속도)이 필요한가? 로컬 데이터베이스는 데이터베이스 애플리케이션과 동일한 시스템에 상주하기 때문에 일반적으로 원격 데이터베이스 서버보다 빠릅니다. 다른 원격 데이터베이스 서버는 다른 타입의 연산을 지원하도록 최적화되어 있으므로 원격 데이터베이스 서버를 선택할 때 성능을 고려할 수 있습니다.
- 데이터베이스 관리에 어떤 지원이 가능한가? 로컬 데이터베이스는 원격 데이터베이스 서버보다 더 많은 지원이 필요하지 않습니다. 일반적으로 로컬 데이터베이스는 서버를 따로 설치하거나 비싼 사이트 사용권이 필요하지 않기 때문에 사용하는 데 비용이 덜 듭니다.

## 데이터베이스 보안

데이터베이스는 종종 기밀을 다루는 정보를 포함합니다. 대부분의 SQL 서버는 다중 레벨에서 보안을 제공합니다. 일반적으로 SQL 서버는 데이터베이스 서버를 사용하기 위해서 암호와 사용자 이름이 필요합니다. 일단 사용자가 데이터베이스에 로그인하면, 사용자 이름과 암호는 사용될 수 있는 테이블을 결정합니다. SQL 서버에 암호를 제공하는 것에 대한 내용은 19-6 페이지의 "서버 로그인 제어"를 참조하십시오.

데이터베이스 애플리케이션을 디자인할 때 어떤 종류의 인증이 사용자의 데이터베이스 서버에 필요한지를 고려해야 합니다. 종종 애플리케이션은 애플리케이션 자체에 대한 로그인만이 필요한 사용자에게 암시적인 데이터베이스 로그인을 표시하지 않도록 디자인됩니다. 사용자가 데이터베이스 암호를 입력하게 만들고 싶지 않다면, 암호를 필요로 하지 않는 데이터베이스를 사용하거나 프로그램적으로 서버에 암호와 사용자 이름을 제공해야만 합니다. 암호를 프로그램에서 제공할 때 애플리케이션에서 암호를 읽는 것으로부터 보안에 위험이 생기지 않도록 주의해야 합니다.

여러 개의 보호되는 시스템 또는 데이터베이스에 로그인하기 때문에 사용자의 애플리케이션이 여러 개의 암호를 필요로 하는 경우라면 보호된 시스템이 필요로 하는 암호들의 테이블을 액세스하는 데 사용되는 하나의 마스터 암호를 제공할 수 있습니다. 그러면 애플리케이션은 사용자가 여러 암호를 제공할 필요없이 프로그램적으로 암호를 제공합니다.

다계층 애플리케이션에서 다른 보안 모델을 함께 사용할 수 있습니다. HTTP 같은 프로토콜을 사용하여 중간 계층에 액세스를 제어할 수 있고, 중간 계층에서 데이터베이스 서버로의 로그인에 대한 모든 상세한 부분을 처리하도록 할 수 있습니다.

## 트랜잭션

트랜잭션은 데이터베이스에 있는 하나 이상의 테이블들이 커밋되기 전에 성공적으로 모두 수행되어야 하는 작업의 그룹입니다. 그룹 내 작업이 하나라도 실패할 경우 모든 작업이 롤백(취소)됩니다.

트랜잭션은 다음 사항을 보증합니다.

- 단일 트랜잭션의 모든 업데이트가 커밋되거나 중지되고 이전 상태로 롤백됩니다. 이것을 **원자성**이라고 합니다.
- 트랜잭션은 상태 불변값을 보존하는 시스템 상태의 합당한 변환입니다. 이것을 **일관성**이라고 합니다.

- 병행 트랜잭션은 서로의 부분적인 결과 또는 커밋되지 않은 결과를 알지 못하므로 애플리케이션 상태에서 비일관성을 만들 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. 디스크 매체 실패 후 트랜잭션 로그를 사용하여 지속 상태를 복구합니다. 트랜잭션은 또한 SQL 서버에서 다중 사용자 병행 제어의 기반을 형성합니다. 각 사용자가 트랜잭션을 통해서만 데이터베이스와 상호 작용할 때 사용자의 명령이 다른 사용자의 트랜잭션 단위를 방해할 수 없습니다. 대신에 SQL 서버는 전체로서 성공하거나 실패하는 수신 트랜잭션을 계획합니다.

대부분의 SQL 서버는 트랜잭션을 지원합니다. 그러나 MySQL 같은 일부 서버는 트랜잭션을 지원하지 않습니다. 이와 비슷하게, 클라이언트 데이터셋만 사용하고 파일에 데이터를 저장하는 경우 트랜잭션을 지원하지 않습니다.

데이터베이스 애플리케이션에서의 트랜잭션 사용에 대한 자세한 내용은 19-7 페이지의 "트랜잭션 관리"를 참조하십시오.

## 참조 무결성, 내장 프로시저 및 트리거

---

모든 관계형 데이터베이스는 애플리케이션이 데이터를 저장하고 조작하도록 하는 특정 기능을 공통적으로 가지고 있습니다. 또한, 데이터베이스는 데이터베이스의 테이블 간에 일관적인 관계를 보증하는 데 유용한 데이터베이스 특정 기능을 경우에 따라 제공합니다. 다음과 같은 기능이 있습니다.

- **참조 무결성.** 참조 무결성은 테이블 간의 마스터/디테일 관계가 손상되지 않도록 하는 메커니즘을 제공합니다. 고아 디테일 레코드를 야기할 수 있는 마스터 테이블의 필드를 사용자가 삭제하려는 경우, 참조 무결성 규칙은 삭제를 막거나 자동으로 고아 디테일 레코드를 삭제합니다.
- **내장 프로시저.** 내장 프로시저는 SQL 서버에 이름이 지정되고 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

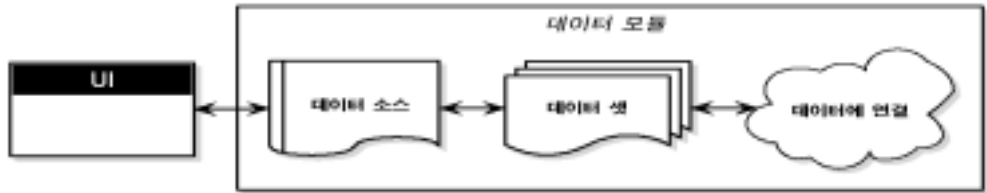
---

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.

## 일반 구조

데이터베이스 애플리케이션의 컴포넌트를 구성하는 많은 다양한 방법들이 있지만 대부분은 다음 그림 14.1에 설명되어 있는 일반적인 스키마를 따릅니다.

그림 14.1 일반 데이터베이스 아키텍처



## 사용자 인터페이스 폼

애플리케이션의 나머지 부분과 완전히 분리된 폼에 사용자 인터페이스를 분리하는 것이 좋습니다. 다음과 같은 여러 가지 장점이 있습니다. 데이터베이스 정보 자체를 나타내는 컴포넌트에서 사용자 인터페이스를 분리하여 디자인에 보다 많은 유연성을 부여합니다. 데이터베이스 정보를 관리하는 방법에 대한 변경으로 사용자 인터페이스를 다시 작성할 필요가 없고, 사용자 인터페이스에 대한 변경으로 데이터베이스를 사용하는 애플리케이션의 일부를 변경할 필요는 없습니다. 또한 이러한 타입의 분리를 사용하여 다중 애플리케이션 간에 공유할 수 있는 공통 폼을 개발할 수 있어서 일관성 있는 사용자 인터페이스를 제공합니다. Object Repository에 잘 디자인된 폼과의 연결을 저장하여 개발자들은 새 프로젝트마다 처음부터 시작하지 않고 기존의 기반에서 구축할 수 있습니다. 또한 폼을 공유하여 애플리케이션 인터페이스에 대한 기업 표준을 개발할 수 있습니다. 데이터베이스 애플리케이션용 사용자 인터페이스 생성에 대한 자세한 내용은 15장 "데이터 컨트롤 사용"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 간에 연결체로서 작용합니다. 폼의 여러 data-aware 컨트롤은 하나의 데이터 소스를 공유할 수 있고, 각 컨트롤의 표시는 동기화되어 사용자가 레코드 전체를 스크롤하고 현재 레코드에 대한 필드의 해당 값이 각 컨트롤에 표시됩니다.

## 데이터셋

데이터베이스 애플리케이션의 핵심은 데이터셋입니다. 이 컴포넌트는 원본으로 사용한 데이터베이스의 레코드 집합을 나타냅니다. 이러한 레코드는 단일 데이터베이스 테이블의 데이터, 테이블의 필드나 레코드의 서브셋, 또는 싱글 뷰로 조인된 하나 이상의 테이블 정보일 수 있습니다. 데이터셋을 사용함으로써 애플리케이션 로직은 데이터베이스의 물리적 테이블을 재구성해서 버퍼링됩니다. 원본으로 사용한 데이터베이스가 변경되면 데이터셋 컴포넌트가 포함하고 있는 데이터를 지정하는 방법을 변경해야 할 수도 있지만 애플리케이션의 나머지 부분은 변경하지 않아도 계속 작동됩니다. 데이터셋의 속성 및 메소드에 대한 자세한 내용은 16장 "데이터셋 이해"를 참조하십시오.

## 데이터 연결

다양한 타입의 데이터셋은 원본으로 사용한 데이터베이스 정보에 연결하기 위한 다양한 메커니즘을 사용합니다. 이러한 다양한 메커니즘은 구축할 수 있는 데이터베이스 애플리케이션의 아키텍처에서 주요 차이점을 차례로 구성합니다. Kylix는 다음 두 가지 타입의 데이터셋을 지원합니다.

- **클라이언트 데이터셋**은 레코드 전체를 보다 쉽게 탐색하고 레코드 필터링이나 데이터를 요약하는 집계 값의 유지 관리와 같은 데이터에서의 연산을 수행할 수 있도록 메모리에서 데이터를 버퍼링합니다. 클라이언트 데이터셋은 원본으로 사용한 데이터베이스에 메모리 캐시의 업데이트를 적용하는 것을 지원합니다. 클라이언트 데이터셋은 메모리에 레코드를 캐시로 저장하기 때문에 제한된 레코드 수만을 보유할 수 있습니다. 일반적인 클라이언트 데이터셋 및 SQL 클라이언트 데이터셋의 두 가지 타입의 데이터셋이 있습니다. 일반적인 클라이언트 데이터셋은 로컬로 저장된 파일을 직접 사용하여 데이터에 액세스하여 동일한 데이터 모듈의 다른 데이터셋에 연결하거나 다른 (서버) 시스템의 애플리케이션 서버에 연결합니다. SQL 클라이언트 데이터셋은 디스크에 로컬로 저장된 파일을 사용하거나 데이터베이스 서버에 연결할 수 있습니다. 클라이언트 데이터셋에 대한 자세한 내용은 20장 "클라이언트 데이터셋 사용"을 참조하십시오.
- **단방향 데이터셋**은 SQL 쿼리로 설명되거나 내장 프로시저가 반환한 데이터를 읽을 수 있습니다. 단방향 데이터셋은 데이터를 버퍼링하지 않으므로 클라이언트 데이터셋보다 유연성이 부족합니다. 단방향 데이터셋의 레코드 전체를 탐색하는 유일한 방법은 SQL 쿼리의 ORDER BY 절로 지정한 순서로 레코드를 반복하는 것입니다. 또한 데이터를 업데이트할 때는 단방향 데이터셋을 사용할 수 없습니다. 그러나 단방향 데이터셋은 데이터베이스 서버의 정보에 빠른 액세스를 제공하고 클라이언트 데이터셋보다 더 큰 데이터의 집합을 나타낼 수 있습니다. 단방향 데이터셋은 항상 SQL 연결 컴포넌트를 사용하는 데이터를 폐지합니다. 단방향 데이터셋에 대한 자세한 내용은 18장 "단방향 데이터셋 사용"을 참조하십시오.

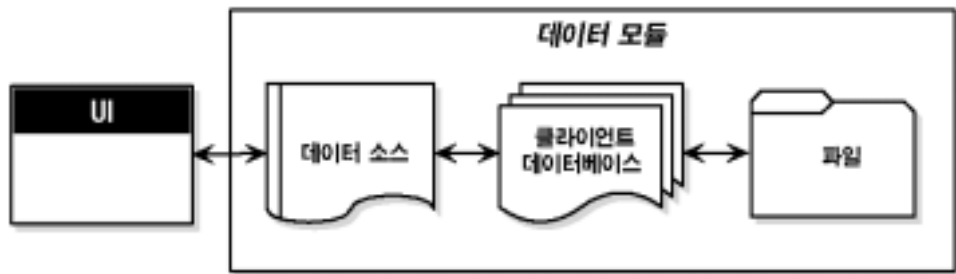
또한 자신만의 사용자 지정 데이터셋을 만들 수 있습니다. 사용자 지정 데이터셋은 작성한 코드에서 만들거나 액세스한 데이터의 몸체를 나타내는 *TDataSet*의 자손입니다. 사용자 지정 데이터셋을 작성하면 여전히 사용자 인터페이스를 구축하는 데 CLX 데이터 컨트롤을 사용할 수 있는 반면에 선택한 메소드를 사용하는 데이터를 관리하는 데 유연성을 부여할 수 있습니다.

다음 항목은 데이터베이스 애플리케이션의 클라이언트 데이터셋, 로컬 데이터셋 및 단방향 데이터셋을 사용하는 가장 일반적인 방법에 대해 설명합니다.

## 디스크에 저장된 데이터가 있는 클라이언트 데이터셋 사용

작성할 수 있는 데이터베이스 애플리케이션의 가장 단순한 형태에서는 데이터베이스 서버를 사용하지 않습니다. 그 대신 파일에 데이터를 저장하고 파일의 데이터를 로드할 때 클라이언트 데이터셋 기능을 사용합니다. 다음 그림 14.2에서 이러한 타입의 애플리케이션 아키텍처를 그림으로 설명합니다.

**그림 14.2** 파일 기반 데이터베이스 애플리케이션의 아키텍처



이 단순한 파일 기반 아키텍처는 단일 계층 애플리케이션입니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

파일 기반 접근 방법은 단순함이 장점입니다. 클라이언트 데이터셋은 midas.so가 필요하지만 설치, 구성 또는 배포할 데이터베이스 서버가 없습니다. 사이트 사용권이나 데이터베이스 관리가 필요하지 않습니다.

그러나 다중 사용자는 지원하지 않습니다. 데이터셋은 애플리케이션 전용이어야 합니다. 데이터는 디스크의 파일에 저장되고 나중에 로드되지만 다중 사용자가 각각의 다른 데이터 파일을 겹쳐 쓰는 것을 방지하는 기본 제공 보호 기능은 없습니다.

분리된 데이터베이스 서버가 없으므로 원본으로 사용할 테이블을 만들어야 합니다. 이 테이블을 만들고 나면 파일에 이 테이블을 저장할 수 있습니다. 이 때부터 테이블을 다시 만들지 않아도 되며 저장한 파일에서 테이블을 로드만 하면 됩니다. 파일 기반 데이터베이스 애플리케이션을 시작할 때 애플리케이션 자체를 작성하기 전에 먼저 데이터베이스에 대한 빈 파일을 만들고 저장해야 하는 경우도 있습니다. 이 방법은 최종 애플리케이션의 클라이언트 데이터셋에 대한 메타데이터를 정의하지 않아도 됩니다. 파일 기반 애플리케이션에 대한 테이블 만들기에 대한 자세한 내용은 20-38 페이지의 "새로운 데이터셋 생성"을 참조하십시오.

이 파일 기반 모델에서 데이터에 대한 모든 편집 내용은 메모리 내 변경 로그에만 있습니다. 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. 그러나 변경 내용을 취소하지 않고자 하는 경우 *MergeChangeLog* 메소드를 호출하여 변경 로그를 클라이언트 데이터셋의 데이터로 병합해야 합니다. 변경 로그에 대한 자세한 내용은 20-14 페이지의 "데이터 편집"을 참조하십시오.

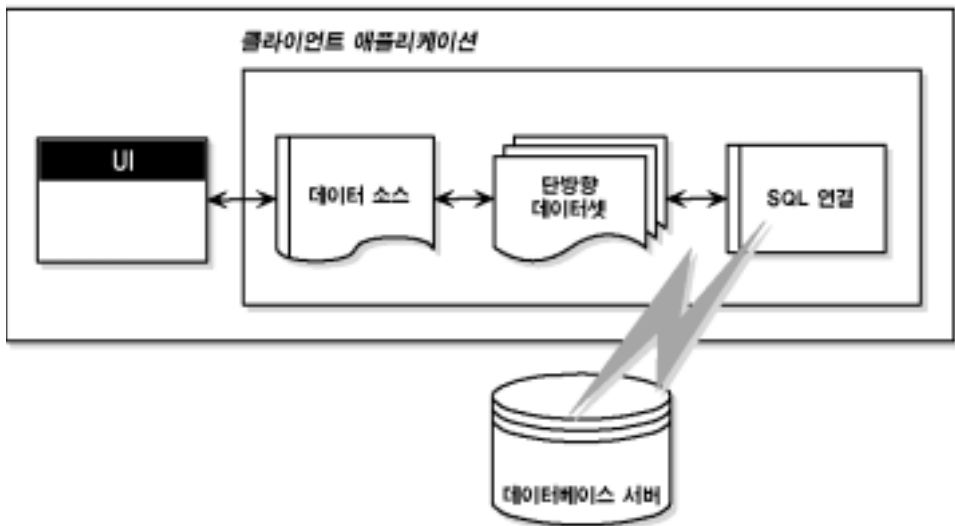
클라이언트 데이터셋의 데이터에 변경 내용을 합병할 때도 이 데이터는 여전히 메모리에만 존재합니다. 클라이언트 데이터셋을 닫고 애플리케이션에서 다시 여는 경우에도 데이터는 여전히 메모리에 존재하지만 애플리케이션을 종료하면 사라집니다. 데이터를 영구적으로 만들려면 데이터를 디스크에 써야 합니다. *SaveToFile* 메소드를 사용하여 디스크에 변경 내용을 쓰십시오. *SaveToFile*은 하나의 매개변수를 취하는 데 이 매개변수는 테이블을 포함하는 작성되거나 겹쳐 쓴 파일의 이름입니다. *SaveToFile* 메소드를 사용하여 이전에 작성한 테이블을 읽으려면, *LoadFromFile* 메소드를 사용하십시오. *LoadFromFile* 또한 하나의 매개변수를 취하는 데 이 매개변수는 테이블을 포함하는 파일의 이름입니다.

항상 동일한 파일을 로드하고 저장하는 경우 *SaveToFile*과 *LoadFromFile* 메소드 대신 *FileName* 속성을 사용할 수 있습니다. *FileName*이 유효한 파일 이름으로 설정된 경우 클라이언트 데이터셋이 열려 있으면 데이터는 자동적으로 파일에서 로드되고, 클라이언트 데이터셋이 닫혀 있으면 파일에 저장됩니다.

## 단방향 데이터셋 직접 사용

Kylix에서는 데이터베이스 서버에 연결하는 데 단방향 데이터셋을 필요로 합니다. 따라서, SQL 서버의 데이터를 사용하는 가장 단순한 방법은 사용자 인터페이스에 이 단방향 데이터셋을 직접 연결하는 것입니다. 다음 그림 14.3에서 이러한 타입의 애플리케이션 아키텍처를 설명합니다.

**그림 14.3** 단방향 데이터베이스 애플리케이션의 아키텍처



데이터베이스 서버가 Local InterBase 같은 로컬 데이터베이스인지 원격 데이터베이스 서버인지에 따라 단일 계층 또는 2계층 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.



이 모델은 데이터베이스 서버를 사용할 때 매우 빠른 액세스를 제공합니다. 데이터 버퍼링이나 메타데이터 관리에 대한 오버헤드는 없습니다. 데이터가 메모리에 버퍼링되지 않으므로 임의의 대형 데이터셋으로 이 모델을 사용할 수 있습니다.

그러나 이 모델을 사용할 때 애플리케이션은 한 번에 하나의 레코드만을 볼 수 있고 전체 레코드를 순서대로만 진행할 수 있습니다. 또한 데이터는 읽기 전용입니다. 데이터베이스에 변경 사항을 포스트하는 기본 제공 방법은 없습니다.

이 모델은 데이터를 읽고 요약 통계 집합을 나타내거나 리포트를 출력하기 위해 분석하는 애플리케이션에 가장 적합합니다. 그렇지만 사용자에게 검색할 값을 주는 데 사용할 수 있습니다.

이 모델의 데이터셋은 단방향 데이터셋입니다(*TSQLDataSet*, *TSQLQuery*, *TSQLTable* 또는 *TSQLStoredProcedure*). 단방향 데이터셋은 데이터베이스 서버에서 SQL 문을 실행하고 SQL 문이 데이터를 반환하는 경우(즉, SELECT 문인 경우), 결과 데이터를 액세스하기 위한 단방향 커서를 얻습니다.

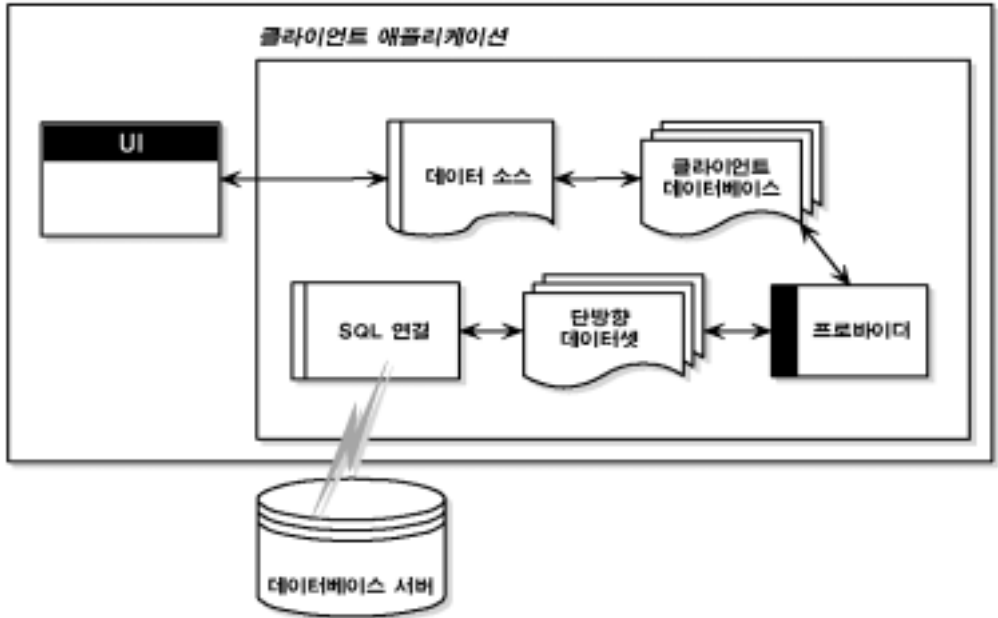
단방향 데이터셋을 서버에 연결하려면 데이터 모듈은 SQL 연결 컴포넌트(*TSQLConnection*)를 포함해야 합니다. 단방향 데이터셋은 *SQLConnection* 속성을 통해 이 SQL 연결 컴포넌트에 연결됩니다. SQL 연결 컴포넌트는 데이터 서버와의 연결을 나타내며 dbExpress가 필요합니다. SQL 연결 컴포넌트를 더블 클릭하여 Connection Editor에서 서버를 확인하십시오. 이 편집기를 사용하여 특정 드라이버에 대한 구성 매개변수의 명명된 집합을 나타내는 연결 이름을 지정합니다. 각 명명된 연결 구성은 *connections.ini*라고 하는 특수한 파일에 정의됩니다. Connection Editor를 사용하여 구성을 선택하고 명명된 구성을 수정하거나 새 구성을 정의할 수 있습니다.

**참고** 일부 드라이버를 별도로 구입해야 하는 경우도 있습니다.

## 클라이언트 데이터셋을 사용하여 레코드 버퍼링

다중 레코드를 표시하거나 data-aware 컨트롤을 사용하여 데이터를 업데이트하거나 전체 레코드를 뒤로 이동하거나 특정 조건에 맞는 레코드로 이동하려면 클라이언트 데이터셋을 사용해야 합니다. 이에 반해서, 데이터베이스 서버에 연결하기 위해서는 단방향 데이터셋을 사용해야만 합니다. 이러한 데이터셋을 조합하여 클라이언트 데이터셋의 유연성을 가지면서 데이터베이스 서버의 데이터를 사용하는 애플리케이션을 만들 수 있습니다. 다음 그림 14.4에서 이러한 타입의 애플리케이션 아키텍처를 그림으로 설명합니다.

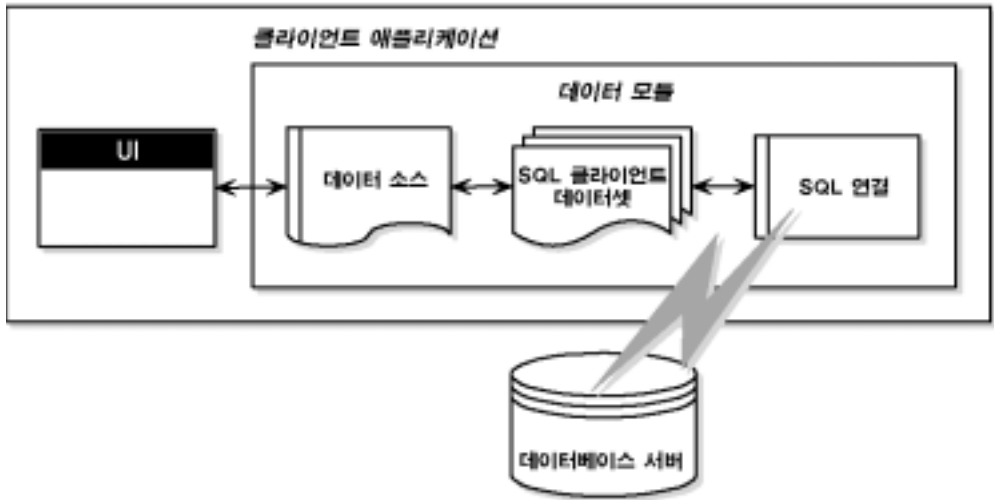
**그림 14.4** 클라이언트 데이터셋과 단방향 데이터셋을 조합하는 아키텍처



클라이언트 데이터셋과 단방향 데이터셋 간의 연결은 데이터셋 프로바이더에 의해 제공됩니다. 프로바이더는 데이터베이스 정보를 클라이언트 데이터셋이 사용할 수 있는 전송 가능한 데이터 패킷으로 패키징화하고 클라이언트 데이터셋이 만든 델타 패킷에 수신된 업데이트를 데이터베이스 서버에 적용합니다. 클라이언트 데이터셋을 프로바이더에 연결하려면 *ProviderName* 속성을 프로바이더 컴포넌트의 이름으로 설정하십시오. 프로바이더와 클라이언트 데이터셋은 동일한 데이터 모듈에 있어야 합니다. 프로바이더를 단방향 데이터 셋에 연결하려면 *DataSet* 속성을 단방향 데이터셋으로 설정하십시오.

이 아키텍처를 단순화하려면, Kylix는 자체의 내부 프로바이더 및 단방향 데이터셋을 포함하는 SQL 클라이언트 데이터셋이라고 하는 특수한 타입의 클라이언트 데이터셋을 포함합니다. SQL 클라이언트 데이터셋을 사용함으로써 그림 14.5와 같이 단순화됩니다.

그림 14.5 내부 단방향 데이터셋이 있는 클라이언트 데이터셋을 사용하는 아키텍처



SQL 클라이언트 데이터셋을 사용할 때 데이터 모듈에 데이터셋 프로바이더와 단방향 데이터셋을 명시적으로 추가하지 않아도 됩니다. 이러한 컴포넌트는 SQL 클라이언트 데이터셋에 대해 내부입니다. 이는 애플리케이션을 단순화하지만 다음과 같은 일부 컨트롤을 사용해야 합니다.

- 단방향 데이터셋은 SQL 클라이언트 데이터셋 컴포넌트에 대해 내부이지만 중첩 디테일 셋을 얻는데 마스터/디테일 관계의 두 단방향 데이터셋을 연결할 수 없습니다. (그러나 SQL 클라이언트 데이터셋 두 개를 마스터/디테일 관계에 연결할 수 있습니다.)
- SQL 클라이언트 데이터셋 컴포넌트는 내부 데이터셋 프로바이더에서 발생하는 대부분의 이벤트를 표면화하지 않습니다. 그러나 대부분의 경우에 이러한 이벤트는 다계층 애플리케이션에서 사용되고, 2계층 애플리케이션에서는 필요하지 않습니다.

이러한 두 모델 모두 데이터베이스가 Local Interbase 같은 로컬 데이터베이스인지 원격 데이터베이스 서버인지에 따라 단일 계층 또는 2계층 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 데이터 모듈로 분리되었지만 사용자 인터페이스를 구현하는 동일한 애플리케이션에 있습니다.

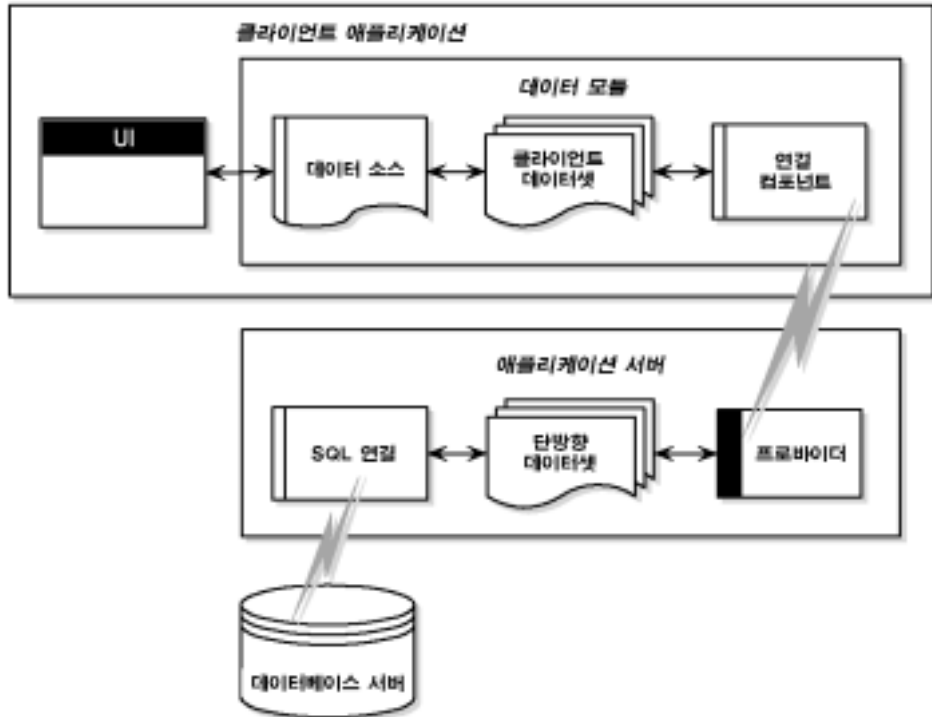
이 모델은 이전 두 모델의 혼성을 나타냅니다. 이 모델은 사용자가 데이터를 편집할 때 클라이언트 데이터셋이 레코드를 버퍼링하여 유연성을 향상시키고 데이터베이스 서버로 업데이트를 다시 적용하는 반면에, 데이터베이스 서버에 액세스하고 레코드를 폐치할 때는 (내부) 단방향 데이터셋을 사용합니다.

클라이언트 데이터셋은 데이터베이스 레코드 전체를 폐치, 표시 및 탐색하는 데 필요한 모든 상세한 부분을 자동으로 처리합니다. 데이터베이스에 사용자가 편집한 내용을 적용하기 위해서는 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출하기만 하면 됩니다. (디스크에 저장된 데이터를 가지고 데이터셋을 이용할 때와는 달리 *MergeChangeLog* 메소드를 호출해서는 안 되고 클라이언트는 프로바이더에 대한 정확한 델타 패킷을 생성할 수 없음에 유의하십시오.)

## 다계층 아키텍처 사용

데이터베이스 정보가 여러 테이블 간의 복합 관계나 클라이언트가 증가하는 수를 포함할 때 다계층 모델을 사용하려고 할 수 있습니다. 다계층 애플리케이션에는 클라이언트 애플리케이션과 데이터베이스 사이에 중간 계층이 있습니다. 다음 그림 14.6에서 이 아키텍처를 그림으로 설명합니다.

**그림 14.6** 다계층 데이터베이스 아키텍처



앞의 그림은 3계층 애플리케이션을 나타냅니다. 데이터베이스 정보를 처리하는 로직은 개별 시스템 또는 계층에 있습니다. 이 중간 계층은 데이터베이스 상호 작용을 관리하는 로직을 중심에 모으므로 데이터 관계에 대한 중앙화된 제어가 있는 것입니다. 이를 사용하면 데이터 로직이 일관성 있을 때 다른 클라이언트 애플리케이션에서 동일한 데이터를 사용할 수 있습니다. 또한 많은 프로세스가 중간 계층으로 분담되기 때문에 더 작은 클라이언트 애플리케이션에서도 사용할 수 있습니다. 이러한 작은 클라이언트 애플리케이션은 설치, 구성 및 유지 관리가 더 용이합니다. 또한 다계층 애플리케이션은 여러 시스템에 데이터 처리 작업이 퍼져 있어서 성능을 향상시킬 수 있습니다.

다계층 아키텍처는 외부 프로바이더 및 단일 계층 데이터셋의 레코드를 버퍼링하는 클라이언트 데이터셋을 사용하는 것과 매우 유사합니다. 단방향 데이터셋과 클라이언트 데이터셋 간에 매개체로서 작동하는 프로바이더와 데이터베이스 서버로 연결하는 단방향 데이터셋이 둘 다 분리 애플리케이션으로 이동했다는 점이 다릅니다. 이러한 분리 애플리케이션을 애플리케이션 서버(또는 "원격 데이터 브로커")라고 합니다.

프로바이더가 분리 애플리케이션으로 이동했기 때문에 단순히 *ProviderName* 속성을 설정하여 단방향 데이터셋에 연결할 수 없습니다. 또한 애플리케이션 서버를 검색하고 연결하는 데 일부 타입의 연결 컴포넌트를 사용해야 합니다. 이 컴포넌트는 *TSoap Connection* 같은 *TCustomRemoteServer*의 자손입니다. *RemoteServer* 속성을 설정하여 연결 컴포넌트에 클라이언트 데이터셋을 연결합니다.

**참고** 원격 애플리케이션 서버와 별도로 클라이언트 데이터셋에 연결하는 컴포넌트를 구입해야 할 수도 있습니다.

연결 컴포넌트는 애플리케이션 서버에 연결을 설정하고 클라이언트 데이터셋이 사용하는 인터페이스를 반환하여 *ProviderName* 속성으로 지정된 프로바이더를 호출합니다. 클라이언트 데이터셋은 애플리케이션 서버를 호출할 때마다 *ProviderName*의 값을 전달하고 애플리케이션 서버는 프로바이더에 호출을 전달합니다.

## 방법 조합

이전 단원에서는 데이터베이스 애플리케이션을 작성할 때 사용할 수 있는 여러 가지 아키텍처에 대해서 설명했습니다. 단일 애플리케이션에서 둘 이상의 사용 가능한 아키텍처를 조합할 수 있습니다. 사실 일부 조합은 매우 강력합니다.

예를 들어, 14-7 페이지의 "디스크에 저장된 데이터가 있는 클라이언트 데이터셋 사용"에서 설명한 디스크 기반 아키텍처를 14-9 페이지의 "클라이언트 데이터셋을 사용하여 레코드 버퍼링" 또는 14-12 페이지의 "다계층 아키텍처 사용"과 같은 다른 방법으로 조합할 수 있습니다. 이 결과를 브리프케이스 모델(경우에 따라 연결 해제된 모델 또는 모바일 컴퓨팅)이라고 합니다.

이 브리프케이스 모델은 다음과 같은 상황에서 유용합니다. 현장의 회사 데이터베이스는 영업 사원이 해당 현장에서 사용하고 업데이트할 수 있는 고객 연락처 데이터를 포함합니다. 현장에 있을 때, 영업 사원은 데이터베이스에서 정보를 다운로드합니다. 나중에 해외 출장 중에도 랩탑으로 데이터를 사용하고 기존 고객 또는 새 고객 위치에 레코드를 업데이트할 수도 있습니다. 영업 사원이 현장으로 복귀하면 모든 사람들이 사용할 수 있도록 회사 데이터베이스에 데이터 변경 내용을 업로드합니다.

현장에서 사용할 때 브리프케이스 모델 애플리케이션의 클라이언트 데이터셋은 프로바이더의 데이터를 폐치합니다. 그 결과 데이터셋이 데이터에 연결되고 프로바이더를 통해 서버 데이터를 폐치하고 다시 서버로 업데이트 내용을 보낼 수 있습니다. 프로바이더에서 연결을 해제하기 전에 클라이언트 데이터셋은 디스크의 파일에 정보의 스냅샷을 저장합니다. 현장이 아닐 때 클라이언트 데이터셋은 파일의 데이터를 로드하고 파일에 변경 내용을 저장합니다. 마침내 현장에 복귀하면, 클라이언트 데이터셋은 프로바이더와 다시 연결하여 데이터베이스 서버로 업데이트 내용을 적용하거나 데이터의 스냅샷을 새로 고칠 수 있습니다.



## 데이터 컨트롤 사용

컴포넌트 팔레트의 Data Controls 페이지는 데이터베이스 레코드의 필드에서 읽어 온 데이터를 표시하는 data-aware 컨트롤을 제공합니다. 그리고 데이터셋이 허용하는 경우 사용자는 그 데이터를 편집하여 변경 사항을 데이터베이스에 다시 포스트할 수 있습니다. 데이터 컨트롤을 사용자의 데이터베이스 애플리케이션 폼에 두면, 데이터베이스 애플리케이션의 사용자 인터페이스(UI)를 구축하여 정보를 볼 수 있고 액세스할 수 있습니다.

사용자 인터페이스에 추가하는 data-aware 컨트롤은 다음과 같은 몇 가지 요인에 따라 달라집니다.

- 표시하려는 데이터 타입. 일반 텍스트, 서식있는 텍스트와 함께 작동하는 컨트롤, 그 래픽용 컨트롤, 멀티미디어 요소 등을 표시하고 편집하도록 고안된 컨트롤들을 선택할 수 있습니다. 다양한 타입의 정보를 표시하는 컨트롤은 15-7 페이지의 "단일 레코드 표시"에서 설명합니다.
- 정보를 구성하는 방법. 화면에 단일 레코드의 정보를 표시하거나 그리드를 사용하여 여러 레코드의 정보를 나열할 수도 있습니다. 데이터를 구성하는 몇 가지 방법은 15-7 페이지의 "데이터 구성 방법 선택"에서 설명합니다.
- 컨트롤에 데이터를 제공하는 데이터셋 타입. 원본으로 사용한 데이터셋의 제한 사항을 반영하는 컨트롤을 사용할 수 있습니다. 예를 들어, 단방향의 데이터셋은 한 번에 하나의 레코드만을 공급할 수 있기 때문에 단방향의 데이터셋에는 그리드를 사용하지 않을 것입니다.
- 사용자들이 데이터셋의 레코드를 탐색하고 데이터를 추가하거나 편집할 수 있도록 하는 방법. 탐색과 편집을 위해 자신의 컨트롤 또는 메커니즘을 추가하거나 데이터 탐색기와 같은 기본 제공 컨트롤을 사용할 수도 있습니다. 데이터 탐색기 사용에 대한 자세한 내용은 15-26 페이지의 "레코드 탐색 및 처리"를 참조하십시오.

사용자의 인터페이스에 추가하기 위해 선택하는 data-aware 컨트롤과는 상관없이 공통적인 기능이 적용됩니다. 이러한 기능은 다음과 같습니다.

# 공통적인 데이터 컨트롤 기능의 사용

다음은 대부분의 데이터 컨트롤에서 공통적인 작업입니다.

- 데이터 컨트롤과 데이터셋 연결
- 데이터 편집과 업데이트
- 데이터 표시 사용 불가능 및 사용 가능
- 데이터 표시 새로 고침
- 마우스, 키보드 및 타이머 이벤트 사용 가능

일반적으로 데이터 컨트롤을 사용하면 데이터셋의 현재 레코드에 연결된 데이터 필드를 표시하고 편집할 수 있습니다. 표 15.1은 컴포넌트 팔레트의 Data Controls 페이지에 나타나는 데이터 컨트롤을 요약해서 보여줍니다.

표 15.1 데이터 컨트롤

| 데이터 컨트롤                  | 설명                                                                                       |
|--------------------------|------------------------------------------------------------------------------------------|
| <i>TDBGrid</i>           | 데이터 소스의 정보를 표 서식으로 표시합니다. 그리드의 열은 원본으로 사용한 테이블이나 쿼리의 데이터셋의 열에 해당합니다. 그리드의 행은 레코드에 해당합니다. |
| <i>TDBNavigator</i>      | 데이터셋의 데이터 레코드를 탐색합니다. 레코드 업데이트, 레코드 포스트, 레코드 삭제, 레코드 편집 취소, 및 데이터 표시 새로 고침.              |
| <i>TDBText</i>           | 필드의 데이터를 레이블로 표시합니다.                                                                     |
| <i>TDBEdit</i>           | 필드의 데이터를 편집 상자에 표시합니다.                                                                   |
| <i>TDBMemo</i>           | 메모 또는 BLOB 필드의 데이터를 스크롤 가능한 여러 줄 편집 상자에 표시합니다.                                           |
| <i>TDBImage</i>          | 데이터 필드의 그래픽을 그래픽 상자에 표시합니다.                                                              |
| <i>TDBListBox</i>        | 현재 데이터 레코드의 필드를 업데이트할 항목의 목록을 표시합니다.                                                     |
| <i>TDBComboBox</i>       | 필드를 업데이트할 항목의 목록을 표시하고, 표준 data-aware 편집 상자과 같이 직접적인 텍스트 입력을 허용합니다.                      |
| <i>TDBCheckBox</i>       | 부울 필드의 값을 나타내는 확인란을 표시합니다.                                                               |
| <i>TDBRadioGroup</i>     | 필드에 대해 상호 배타적인 옵션을 표시합니다.                                                                |
| <i>TDBLookupListBox</i>  | 필드 값을 기준으로 다른 데이터셋에서 조회한 항목의 목록을 표시합니다.                                                  |
| <i>TDBLookupComboBox</i> | 필드 값을 기준으로 다른 데이터셋에서 조회된 항목의 목록을 표시하고, 표준 data-aware 편집 상자과 같은 직접적인 텍스트 입력을 허용합니다.       |

데이터 컨트롤은 디자인 타임에도 데이터를 인식(data-aware)합니다. 애플리케이션을 구축하는 동안 활성 데이터셋과 데이터 컨트롤을 연결하면, 컨트롤에서 활성 데이터를 즉시 볼 수 있습니다. Fields Editor를 사용하면 디자인 타임에 데이터셋을 스크롤하여, 애플리케이션을 컴파일하고 실행할 필요 없이 애플리케이션이 데이터를 올바르게 표시하는지 확인할 수 있습니다. Fields Editor에 대한 자세한 내용은 17-4 페이지의 "영구적 필드 생성"을 참조하십시오.

런타임 시 데이터 컨트롤은 데이터를 표시합니다. 그리고 애플리케이션, 컨트롤 및 데이터셋이 모두 허용한다면 컨트롤을 통해 데이터를 편집할 수 있습니다.



## 데이터 컨트롤과 데이터셋 연결

데이터 컨트롤은 데이터 소스를 사용하여 데이터베이스에 연결됩니다. 데이터 소스 컴포넌트(*TDataSource*)는 컨트롤과 데이터를 포함하고 있는 데이터셋 사이의 중개자 같은 역할을 합니다. 데이터를 표시하고 처리하려면 각 data-aware 컨트롤을 데이터 소스 컴포넌트에 연결해야 합니다. 이와 비슷하게, 데이터를 폼의 data-aware 컨트롤에 표시하고 처리하려면 모든 데이터셋을 데이터 소스 컴포넌트에 연결해야 합니다.

**참고** 데이터 소스 컴포넌트는 마스터 디테일 관계의 중첩되지 않은 데이터셋 연결을 위해서도 필요합니다.

데이터 컨트롤을 데이터셋에 연결하려면 다음과 같이 합니다.

- 1 데이터셋을 데이터 모듈이나 폼에 두고, 속성을 적절히 설정합니다.
- 2 데이터 소스를 동일한 데이터 모듈이나 폼에 둡니다. Object Inspector를 사용하여 *DataSet* 속성을 1단계에서 둔 데이터셋으로 설정합니다.
- 3 컴포넌트 팔레트의 Data Access 페이지의 데이터 컨트롤을 폼에 둡니다.
- 4 Object Inspector를 사용하여 컨트롤의 *DataSource* 속성을 2단계에서 둔 데이터 소스 컴포넌트로 설정합니다.
- 5 컨트롤의 *DataField* 속성을 표시할 필드 이름으로 설정하거나 속성의 드롭다운 목록에서 필드 이름을 선택합니다. 이 단계는 *TDBGrid* 및 *TDBNavigator*에 적용되지 않습니다. 왜냐하면 이들은 데이터셋의 사용 가능한 모든 필드에 액세스하기 때문입니다.
- 6 데이터셋의 *Active* 속성을 *True*로 설정하여 컨트롤에 데이터를 표시합니다.

## 런타임 시 연결된 데이터셋 변경

앞의 예제에서는 디자인 타임에 *DataSet* 속성을 설정하여 데이터 소스를 해당 데이터셋에 연결했습니다. 런타임 시 필요에 따라 데이터 소스 컴포넌트에 대한 데이터셋을 교체할 수 있습니다. 예를 들어, 다음의 코드는 *CustSource* 데이터 소스 컴포넌트에 대한 데이터셋을 *Customers*와 *Orders*라는 데이터셋 컴포넌트로 바꿉니다.

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

또한 *DataSet* 속성을 다른 폼의 데이터셋으로 설정하여 두 폼에 있는 데이터 컨트롤을 동기화할 수 있습니다. 예를 들면,

```
procedure TForm2.FormCreate (Sender :TObject);
begin
  DataSource1.DataSet := Form1.ClientDataSet1;
end;
```

## 데이터 소스의 사용 가능 및 사용 불가능

데이터 소스에는 데이터셋과의 연결 여부를 결정하는 *Enabled* 속성이 있습니다. *Enabled* 속성이 *True*이면 데이터 소스는 데이터셋에 연결됩니다.

*Enabled* 속성을 *False*로 설정하여 데이터셋에서 단일 데이터 소스를 일시적으로 연결 해제할 수 있습니다. *Enabled* 속성이 *False*가 되면 데이터 소스 컴포넌트에 첨부된 모든 데이터 컨트롤은 빈 상태가 되고 *Enabled*를 *True*로 설정할 때까지 비활성 상태가 됩니다. 그러나 데이터셋 컴포넌트의 *DisableControls*와 *EnableControls* 메소드는 첨부된 모든 데이터 소스에 영향을 주기 때문에 이들 메소드를 통해 데이터셋에 대한 액세스를 제어하는 것이 좋습니다.

## 데이터 소스에서 변경 사항에 대한 응답

데이터 소스는 데이터 컨트롤과 데이터셋 사이의 연결을 제공하기 때문에 이 둘 사이에 발생하는 모든 통신을 조정합니다. 일반적으로 data-aware 컨트롤은 데이터셋의 변경 사항에 자동으로 응답합니다. 그러나 사용자 인터페이스에서 data-aware가 아닌 컨트롤을 사용하는 경우 데이터 소스 컴포넌트의 이벤트를 사용하여 같은 종류의 응답을 수동으로 제공할 수 있습니다.

*OnDataChange* 이벤트는 필드 편집을 비롯하여 레코드의 데이터가 변경되거나 커서가 새 레코드로 이동할 때 발생합니다. 이 이벤트는 변경이 발생할 때마다 트리거되기 때문에 컨트롤이 데이터셋의 현재 필드 값 반영을 확실히 해야 할 때 유용합니다. 일반적으로 *OnDataChange* 이벤트 핸들러는 필드 데이터를 표시하는 data-aware가 아닌 컨트롤의 값을 새로 고칩니다.

*OnUpdateData* 이벤트는 현재 레코드의 데이터가 포스트되기 직전에 발생합니다. 예를 들어 *OnUpdateData* 이벤트는 *Post*가 호출된 이후, 데이터가 실제로 변경 로그에 포스트되기 전에 발생합니다.

*OnStateChange* 이벤트는 데이터셋의 상태가 변경될 때 발생합니다. 이 이벤트가 발생할 때 데이터셋의 *State* 속성을 검사하여 현재 상태를 알 수 있습니다.

예를 들어, 다음의 *OnStateChange* 이벤트 핸들러는 현재 상태에 따라 버튼이나 메뉴 항목을 활성화하거나 비활성화합니다.

```
procedure Form1.DataSource1.StateChange(Sender:TObject);
begin
    CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
    CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
    CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
    ...
end;
```

**참고** 데이터셋 상태에 대한 자세한 내용은 16-3 페이지의 "데이터셋 상태 결정 및 설정"을 참조하십시오.

## 데이터 편집과 업데이트

탐색기를 제외한 모든 데이터 컨트롤은 데이터베이스 필드의 데이터를 표시합니다. 또한 원본으로 사용한 데이터셋이 허용하는 한 이들 컨트롤을 사용하여 데이터를 편집하고 업데이트할 수 있습니다.

**참고** 단방향 데이터셋에서는 사용자가 데이터를 편집하고 업데이트할 수 없습니다.

### 컨트롤에서 사용자 입력의 편집 사용 가능

데이터를 편집하기 위해서는 데이터셋이 *dsEdit* 상태여야 합니다. 데이터 소스의 *AutoEdit* 속성이 기본값 *True*인 경우, 데이터 컨트롤은 사용자가 데이터를 편집하는 순간 데이터셋을 *dsEdit* 모드로 변경합니다.

*AutoEdit*가 *False*인 경우에는 데이터셋을 편집 모드로 변경하려면 다른 대안 메커니즘을 제공해야 합니다. 대안 메커니즘 중 하나는 *Edit* 버튼과 함께 *TDBNavigator* 컨트롤을 사용하여 데이터셋을 명시적으로 편집 모드로 변경하는 것입니다. *TDBNavigator*에 대한 자세한 내용은 15-26 페이지의 "레코드 탐색 및 처리"를 참조하십시오. 다른 방법으로는 데이터셋을 편집 모드로 변경하고 싶을 때 데이터셋의 *Edit* 메소드를 호출하는 코드를 생성하는 것입니다.

### 컨트롤에서의 데이터 편집

데이터셋의 *CanModify* 속성이 *True*인 경우 데이터 컨트롤은 편집 사항을 연결된 데이터셋에 포스트만 할 수 있습니다. *CanModify*는 단방향 데이터셋에 대해 항상 *False*입니다. 클라이언트 데이터셋에는 *CanModify*가 *True*인지 지정할 수 있는 *ReadOnly* 속성이 있습니다.

**참고** 클라이언트 데이터셋이 데이터를 업데이트할 수 있는지의 여부는 소스 데이터셋이 업데이트를 허용하는지의 여부가 아닌 원본으로 사용한 데이터베이스 테이블이 업데이트를 허용하는지의 여부에 달려 있습니다. 그러므로 클라이언트 데이터셋은 항상 읽기 전용인 단방향 데이터셋에서 데이터를 페치(fetch)하는 경우라도 *ReadOnly*를 *False*로 설정합니다. 이는 클라이언트 데이터셋이 원본으로 사용한 데이터베이스 서버에 직접 업데이트를 적용할 수 있기 때문입니다.

데이터셋의 *CanModify* 속성이 *True*인 경우라도, 컨트롤이 데이터베이스 테이블로 업데이트 사항을 다시 포스트하기 전에 데이터셋을 컨트롤에 연결하는 데이터 소스의 *Enabled* 속성이 *True*여야 합니다. 데이터 소스의 *Enabled* 속성은 컨트롤이 데이터셋의 필드 값을 표시할 수 있는지를 결정하며, 그에 따라 사용자가 값을 편집하고 포스트할 수 있는지의 여부도 결정합니다. *Enabled*가 기본값 *True*인 경우, 컨트롤은 필드 값을 표시할 수 있습니다.

마지막으로 사용자가 컨트롤에 표시된 데이터에 편집 사항을 입력할 수 있는지의 여부를 제어할 수 있습니다. 데이터 컨트롤의 *ReadOnly* 속성은 사용자가 컨트롤로 표시된 데이터를 편집할 수 있는지를 결정합니다. 기본값 *False*이면 데이터를 사용자가 편집할 수 있습니다. 데이터셋의 *CanModify* 속성이 *False*인 경우, 사용자는 컨트롤의 *ReadOnly* 속성이 *True*인지를 확인하려고 할 것입니다. 그렇지 않으면 원본으로 사용한 데이터베이스 테이블의 데이터에 영향을 줄 수 있는 잘못된 느낌을 사용자에게 제공합니다.

필드를 수정할 때, *TDBGrid*를 제외한 모든 데이터 컨트롤에서 수정 사항은 컨트롤에서 *Tab*을 누르면 원본으로 사용한 데이터셋으로 복사됩니다. 필드에서 *Tab*을 누르기 전에 *Esc*를 누르면 데이터 컨트롤은 수정 사항을 버리고 필드 값은 수정하기 전의 값으로 되돌아갑니다.

*TDBGrid*에서 수정 사항은 사용자가 다른 레코드로 이동할 때 포스트됩니다. 다른 레코드로 이동하기 전에 필드의 아무 레코드에서나 *Esc*를 누르면 레코드의 모든 변경 사항을 취소할 수 있습니다.

클라이언트 데이터셋은 데이터에 대한 모든 수정 사항을 변경 로그에 저장합니다. 사용자가 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출한 후 또는 디스크의 파일에 데이터를 저장하는 경우에는 클라이언트 데이터셋의 *MergeChangeLog* 메소드를 호출한 후에 이러한 수정 사항이 원본으로 사용한 데이터베이스 테이블에 적용되며, 그런 다음 클라이언트 데이터셋을 저장합니다.

## 데이터 표시 사용 불가능 및 사용 가능

애플리케이션이 데이터셋을 반복하거나 검색을 수행할 때, 현재 레코드가 변경될 때마다 data-aware 컨트롤에 표시된 값을 일시적으로 새로 고쳐지지 않게 해야 합니다. 새로 고침을 일시적으로 중지하면 반복 또는 검색 속도가 빨라지고 화면이 깜빡이는 현상을 막을 수 있습니다.

*DisableControls*는 데이터셋에 연결된 모든 data-aware 컨트롤의 표시를 사용 가능하게 하는 데이터셋 메소드입니다. 반복이나 검색이 끝나자마자 애플리케이션은 즉시 데이터셋의 *EnableControls* 메소드를 호출하여 컨트롤의 표시를 다시 사용 가능하게 합니다.

일반적으로 반복 프로세스를 시작하기 전에 컨트롤을 사용 못하게 합니다. 반복 프로세스 자체는 **try...finally** 문장 내에 있으며, 프로세스 동안 예외가 발생하더라도 컨트롤을 다시 사용 가능하게 할 수 있습니다. **finally** 절은 *EnableControls*를 호출합니다. 다음 코드는 *DisableControls*와 *EnableControls*를 사용하는 방법을 보여줍니다.

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    :
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

## 데이터 표시 새로 고침

데이터셋의 *Refresh* 메소드는 지역 버퍼를 플리시하고 개방형 데이터셋의 데이터를 다시 페치(fetch)합니다. 이 메소드를 사용하면, 다른 애플리케이션이 사용자의 애플리케이션에서 사용되는 데이터에 대한 동시 액세스를 가지기 때문에 원본으로 사용한 데이터가 변경되었다고 생각하는 경우, data-aware 컨트롤의 표시를 업데이트할 수 있습니다. 그러나 데이터셋을 새로 고치기 전에 데이터셋이 현재 캐시한 모든 업데이트 사항을 적용했는지 확인하십시오. 보류 중인 업데이트 사항을 적용하기 전에 데이터를 새로 고치려고 시도하는 경우 클라이언트 데이터셋에서 예외가 발생합니다.

새로 고침은 가끔 예기치 못한 결과를 초래합니다. 예를 들어, 사용자가 다른 애플리케이션에서 삭제된 레코드를 보고 있는 경우, 사용자의 애플리케이션이 *Refresh*를 호출하는 순간 레코드가 사라집니다. 사용자가 데이터를 페치한 후 그리고 *Refresh*를 호출하기 전에 다른 사용자가 레코드를 변경하면 데이터는 변경 내용이 나타날 수도 있습니다.

## 마우스, 키보드 및 타이머 이벤트 사용 가능

데이터 컨트롤의 *Enabled* 속성은 마우스, 키보드 또는 타이머 이벤트에 대해 응답할지의 여부를 결정하고 데이터 소스에 정보를 전달합니다. 이 속성의 기본 설정은 *True*입니다.

마우스, 키보드, 또는 타이머 이벤트를 데이터 컨트롤에 접근하지 못하게 하려면, *Enabled* 속성을 *False*로 설정합니다. *Enabled*가 *False*인 경우 컨트롤을 데이터셋에 연결하는 데이터 소스는 데이터 컨트롤에서 정보를 받지 못합니다. 데이터 컨트롤은 데이터를 계속 표시하지만 컨트롤에 표시된 텍스트는 흐리게 표시됩니다.

## 데이터 구성 방법 선택

데이터베이스 애플리케이션에 대한 사용자 인터페이스를 구축할 경우 정보 표시 및 해당 정보를 처리하는 컨트롤의 구성 방법을 선택합니다.

먼저 내려야 하는 결정 중 하나는 한 번에 레코드 하나를 표시할지 아니면 여러 레코드를 표시할지를 선택하는 것입니다.

레코드를 탐색하고 처리하는 컨트롤을 추가하기를 원할 수도 있습니다. *TDBNavigator* 컨트롤은 수행하고자 하는 많은 기능에 대해 기본 제공 지원을 제공합니다.

## 단일 레코드 표시

많은 애플리케이션의 경우, 한 번에 데이터의 레코드 하나에 대한 정보를 표시할 수 있습니다. 예를 들어, 주문 입력 애플리케이션은 다른 주문이 현재 기록되었는지를 나타내지 않고도 하나의 주문에 대한 정보를 표시할 수도 있습니다. 이 정보는 주문 데이터셋의 단일 레코드로부터 비롯됩니다. 사용자 인터페이스를 단방향 데이터셋에 직접 연결하는 경우, 단일 레코드 사용자 인터페이스를 유일하게 사용할 수 있습니다.

모든 데이터베이스 정보가 동일한 것(앞에서는 동일한 주문)에 관한 것이기 때문에 단일 레코드를 표시하는 애플리케이션이 일반적으로 읽고 이해하기 쉽습니다. 이러한 사용자 인터페이스의 data-aware 컨트롤은 데이터베이스 레코드의 단일 필드를 나타냅니다. 컴포넌트 팔레트의 Data Controls 페이지는 여러 종류의 필드를 나타내는 여러가지 컨트롤을 제공합니다. 이러한 컨트롤은 일반적으로 컴포넌트 팔레트에서 사용할 수 있는 다른 컨트롤의 data-aware 버전입니다. 예를 들어, *TDBEdit* 컨트롤은 사용자가 텍스트 문자열(string)을 보면서 편집할 수 있는 표준 *TEdit* 컨트롤의 data-aware 버전입니다.

어느 컨트롤을 사용할지는 필드에 포함된 데이터의 타입(텍스트, 서식있는 텍스트, 그래픽, 부울 정보, 등)에 따라 달라집니다.

## 데이터를 레이블로 표시

*TDBText*는 컴포넌트 팔레트의 Standard 페이지에 있는 *TLabel* 컴포넌트와 비슷한 읽기 전용 컨트롤입니다. *TDBText* 컨트롤은 사용자가 다른 컨트롤에 입력할 수 있도록 하는 폼으로 표시 전용 데이터를 제공하고자 할 때 유용합니다. 예를 들어, 고객 목록 테이블의 필드에 폼을 생성한다고 가정하십시오. 사용자가 주소, 시, 도 정보를 폼에 입력하면, 별도의 테이블에서 우편 번호 필드를 자동으로 결정하기 위해 동적 조회를 사용한다고 간주합니다. 우편 번호 테이블에 연결된 *TDBText* 컴포넌트는 사용자가 입력한 주소와 일치하는 우편 번호 필드를 표시하는 데 사용할 수 있습니다.

*TDBText*는 표시하는 텍스트를 데이터셋의 현재 레코드에 지정된 필드에서 얻습니다. *TDBText*는 데이터셋에서 텍스트를 얻기 때문에, 표시하는 텍스트는 동적입니다. 즉, 사용자가 데이터베이스 테이블을 탐색하면 텍스트가 변경됩니다. 따라서 *TLabel*에서와 같이 디자인 타임에 *TDBText*의 표시 텍스트를 지정할 수는 없습니다.

**참고** *TDBText* 컴포넌트를 폼에 둘 때는 *AutoSize* 속성이 기본값 *True*인지를 확인하여 컨트롤이 폭이 변하는 데이터를 표시하기 위해 필요에 따라 자체 크기를 변경하도록 합니다. *AutoSize*가 *False*이고, 컨트롤이 너무 작으면 데이터가 잘려서 표시됩니다.

## 편집 상자의 필드 표시 및 편집

*TDBEdit*는 편집 상자 컴포넌트의 data-aware 버전입니다. *TDBEdit*는 연결된 데이터 필드의 현재 값을 표시하고 표준 편집 상자 기법을 사용하여 편집될 수 있도록 합니다.

예를 들어, *CustomersSource*가 *CustomersTable*이라는 개방형 *TSQLClientDataSet*에 연결되고 활성화된 *TDataSource* 컴포넌트라고 가정하십시오. 그런 다음 폼에 *TDBEdit* 컴포넌트를 두고 속성을 다음과 같이 설정할 수 있습니다.

- *DataSource*: CustomersSource
- *DataField*: CustNo

data-aware 편집 상자 컴포넌트는 디자인 타임과 런타임 모두 *CustomersTable* 데이터셋의 *CustNo* 열의 현재 행의 값을 즉시 표시합니다.

## 메모 컨트롤에 텍스트 표시 및 편집

*TDBMemo*는 긴 텍스트 데이터를 표시할 수 있는 표준 *TMemo* 컴포넌트와 비슷한 data-aware 컴포넌트입니다. *TDBMemo*는 여러 줄의 텍스트를 표시할 뿐만 아니라 여러 줄의 텍스트를 입력할 수도 있습니다. *TDBMemo* 컨트롤을 사용하여 큰 텍스트 필드나 BLOB 필드에 포함된 텍스트 데이터를 표시할 수 있습니다.

기본적으로 *TDBMemo*를 사용하면 사용자가 메모 텍스트를 편집할 수 있습니다. 편집을 방지하려면, 메모 컨트롤의 *ReadOnly* 속성을 *True*로 설정합니다. Tab을 표시하고 사용자가 메모에 이를 입력하도록 하려면 *WantTabs* 속성을 *True*로 설정합니다. 사용자가 데이터베이스 메모에 입력할 수 있는 문자 수를 제한하려면 *MaxLength* 속성을 사용합니다. *MaxLength*의 기본값은 0입니다. 즉, 운영 체제에서 부과된 것 이외의 문자 제한은 없습니다.

일부 속성은 데이터베이스 메모의 표시 방법과 텍스트 입력 방법에 영향을 줍니다. 줄바꿈을 방지하려면 *WordWrap* 속성을 *False*로 설정합니다. *Alignment* 속성은 컨트롤 내에서 텍스트의 정렬 방법을 결정합니다. 기본값인 *taLeftJustify*, *taCenter* 및 *taRightJustify*에서 선택할 수 있습니다. 텍스트의 글꼴을 변경하려면 *Font* 속성을 사용합니다.

런타임 시 사용자는 데이터베이스 메모 컨트롤에서 텍스트를 잘라내고, 복사하고, 붙여넣을 수 있습니다. *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 사용하여 프로그램으로 동일한 작업을 구현할 수 있습니다.

*TDBMemo*는 많은 양의 데이터를 표시할 수 있기 때문에 런타임 시 표시하는 데 시간이 걸릴 수 있습니다. 데이터 레코드를 스크롤하는 데 걸리는 시간을 줄이기 위해, *TDBMemo*에는 액세스된 데이터를 자동으로 표시할지 여부를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 *False*로 설정하면, *TDBMemo*는 실제 데이터가 아닌 필드 이름을 표시합니다. 컨트롤 안을 두 번 누르면 실제 데이터를 볼 수 있습니다.

## 이미지 컨트롤의 그래픽 필드 표시 및 편집

*TDBImage*는 BLOB 필드에 포함된 그래픽을 표시하는 data-aware 컨트롤입니다.

기본적으로, *TDBImage*를 통해서 사용자는 *CutToClipboard*, *CopyToClipboard* 및 *PasteFromClipboard* 메소드를 사용하여 클립보드에서 잘라내기 및 붙여넣기를 함으로써 그래픽 이미지를 편집할 수 있습니다. 대신 이벤트 핸들러에 첨부된 사용자의 편집 메소드를 컨트롤에 제공할 수도 있습니다.

기본적으로 이미지 컨트롤은 너무 큰 이미지를 잘라내어 컨트롤에 들어갈 수 있는 만큼의 그래픽을 표시합니다. *Stretch* 속성을 *True*로 설정하여 이미지 컨트롤의 크기가 변경되면 이미지 컨트롤에 맞도록 그래픽의 크기를 조정할 수 있습니다.

*TDBImage*는 많은 양의 데이터를 표시할 수 있기 때문에 런타임 시 표시하는 데 시간이 걸릴 수 있습니다. 데이터 레코드를 스크롤하는 시간을 줄이기 위해 *TDBImage*에는 액세스된 데이터를 자동으로 표시할지 여부를 제어하는 *AutoDisplay* 속성이 있습니다. *AutoDisplay*를 *False*로 설정하면, *TDBImage*는 실제 데이터가 아닌 필드 이름을 표시합니다. 컨트롤 안쪽을 더블 클릭하면 실제 데이터가 표시됩니다.

## 목록 상자와 콤보 상자의 데이터 표시 및 편집

런타임 시 사용자에게 선택할 수 있는 기본 데이터 값 모음을 제공하는 데이터 컨트롤은 4개가 있습니다. 이러한 컨트롤은 표준 목록 상자와 콤보 상자 컨트롤의 data-aware 버전입니다.

- *TDBListBox*는 사용자가 데이터 필드에 입력하기 위해 선택할 수 있는 항목의 스크롤 가능한 목록을 표시합니다. data-aware 목록 상자는 현재 레코드의 필드에 대한 기본값을 표시하고 목록의 해당 입력을 강조하여 표시합니다. 현재 행의 필드 값이 목록에 없는 경우 목록 상자에 어떤 값도 강조하여 표시되지 않습니다. 사용자가 목록 항목을 선택하면, 원본으로 사용한 데이터셋에서 해당 필드 값이 바뀝니다.
- *TDBComboBox*는 data-aware 편집 컨트롤의 기능과 드롭다운 목록을 조합합니다. 런타임 시 이 컨트롤은 사용자가 이미 정의된 값 모음에서 선택할 수 있는 드롭다운 목록을 표시하고, 사용자가 완전히 다른 값을 선택할 수 있도록 합니다.
- *TDBLookupListBox*는 표시 항목의 목록이 다른 데이터셋에서 조회된다는 점을 제외하면 *TDBListBox*처럼 행동합니다.
- *TDBLookupComboBox*는 표시 항목의 목록이 다른 데이터셋에서 조회된다는 점을 제외하면 *TDBComboBox*처럼 행동합니다.

**참고** 런타임 시 사용자는 증가 검색을 사용하여 목록 상자 항목을 찾을 수 있습니다. 예를 들어, 컨트롤에 포커스가 있을 때, "ROB"라고 입력하면 목록 상자에서 "ROB" 글자로 시작하는 첫 번째 항목이 선택됩니다. 추가로 "E"를 입력하면 "Robert Johnson"과 같이 "ROBE"로 시작하는 첫 번째 항목이 선택됩니다. 검색은 대소문자를 구분합니다. *Backspace*와 *Esc*를 누르면 키를 입력하는 사이에 두 번째 멈춤을 두 번 할 때와 같이 현재 검색 문자열(선택은 바뀌지 않음)을 취소합니다.

### TDBListBox 및 TDBComboBox 사용

*TDBListBox* 또는 *TDBComboBox*를 사용할 때는 표시할 항목의 목록을 생성하기 위해 디자인 타임에 String List Editor를 사용해야 합니다. String List Editor를 열려면, Object Inspector의 *Items* 속성에서 생략 버튼을 누릅니다. 그런 다음 목록에 표시하려는 항목을 입력합니다. 런타임 시, *Items* 속성의 메소드를 사용하면 문자열 목록을 처리할 수 있습니다. *Sorted* 속성을 *True*로 설정하면 목록의 항목이 알파벳 순서로 표시되도록 지정할 수 있습니다.

*TDBListBox* 또는 *TDBComboBox* 컨트롤이 *DataField* 속성을 통해 필드에 연결된 경우, 필드 값은 목록에 선택되어 나타납니다. 현재 값이 목록에 없는 경우 어떤 항목도 선택되어 표시되지 않습니다. 그러나 *TDBComboBox*는 *Items* 목록에 나타나는지의 여부와 상관없이 편집 상자에 필드에 대한 현재 값을 표시합니다.

*TDBListBox*의 목록이나 *TDBComboBox*의 드롭다운 목록의 항목 각각은 *Item Height*에 의해 지정된 높이를 가집니다. 목록 아래가 *TDBListBox*에서 완전히 보이는지 확인하려면, *Height* 속성을 여러 *ItemHeight*로 설정합니다. *TDBComboBox*의 경우에는 드롭다운 목록의 크기를 *Height* 속성으로 조정할 수 없기 때문에 위와 같이 할 필요가 없습니다. 그 대신에 *DropDownCount*를 목록에 나타나는 항목의 최대 수로 설정할 수 있습니다. 목록의 항목 수가 목록의 크기를 초과하면 (*TDBListBox*의 *Height* 또는 *TDBComboBox*의 *DropDownCount*에서 지정된 것과 같이), 사용자는 목록을 스크롤할 수 있습니다.



*TDBComboBox*의 경우, *Style* 속성은 사용자와 컨트롤과의 상호 작용을 결정합니다. 기본적으로, *Style*은 사용자가 키보드로 값을 입력하거나 드롭다운 목록에서 항목을 선택할 수 있는 것을 의미하는 *csDropDown*입니다. *Style*의 가능한 값은 다음과 같습니다.

- *csDropDown*(기본값): 사용자가 텍스트를 입력할 수 있는 편집 상자와 드롭다운 목록을 표시합니다. 모든 항목은 문자열이고 높이가 동일합니다.
- *csDropDownList*: 드롭다운 목록 상자와 편집 상자를 표시합니다. 그러나 사용자는 런타임에 드롭다운 목록에 없는 값을 입력하거나 변경할 수 없습니다.
- *csOwnerDrawFixed* 및 *csOwnerDrawVariable*: 항목 목록에서 비트맵 같이 문자열 이외의 값을 표시할 수 있도록 하거나 목록의 개별 항목에 대해 다른 글꼴을 사용할 수 있도록 합니다.

### 조회 목록 상자와 콤보 상자의 데이터 표시 및 편집

조회 목록 상자와 조회 콤보 상자(*TDBLookupListBox*와 *TDBLookupComboBox*)는 사용자에게 유효한 필드 값을 설정하는 제한된 선택 목록을 제공합니다. 사용자가 목록 항목을 선택하면 원본으로 사용한 데이터셋에서 해당 필드 값이 바뀝니다.

예를 들어, *OrdersTable*에 연결된 필드가 있는 주문 폼을 생각해 보십시오. *OrdersTable*에는 고객 ID에 해당하는 *CustNo* 필드는 있지만 그외 고객 정보는 *OrdersTable*에 없습니다. 반면에, *CustomersTable*은 고객 ID에 해당하는 *CustNo* 필드가 있으며, 고객의 회사나 메일 주소와 같은 추가적인 정보도 포함되어 있습니다. 점원이 송장을 생성할 때 주문 폼에서 고객 ID 대신 회사 이름으로 고객을 선택할 수 있다면 편리할 것입니다. *CustomersTable*에 모든 회사 이름을 표시하는 *TDBLookupListBox*를 사용하면 사용자가 목록에서 회사 이름을 선택할 수 있고, 주문 폼에서 적절한 *CustNo*를 설정할 수 있습니다.

이런 조회 컨트롤은 다음 두 소스 중 하나로부터 표시 항목 목록을 파생시킵니다.

- **데이터셋에 대해 정의된 조회 필드.** 조회 필드를 사용하여 목록 상자 항목을 지정하려면, 컨트롤을 연결하는 데이터셋은 이미 조회 필드를 정의해야 합니다. 이 프로세스는 17-8 페이지의 "조회 필드 정의"에 설명되어 있습니다. 목록 상자 항목에 대한 조회 필드를 지정하려면 다음과 같이 합니다.
  - 1 목록 상자의 *DataSource* 속성을 사용할 조회 필드를 포함하는 데이터셋에 대한 데이터 소스로 설정합니다.
  - 2 *DataField* 속성에 대한 드롭다운 목록에서 사용할 조회 필드를 선택합니다.  
조회 컨트롤에 연결된 테이블을 활성화할 때, 컨트롤은 데이터 필드가 조회 필드라는 것을 인식하고 조회로부터 적절한 값을 표시합니다.
- **보조 데이터 소스, 데이터 필드 및 키.** 데이터셋에 대한 조회 필드를 정의하지 않은 경우, 보조 데이터 소스, 보조 데이터 소스에서 검색할 필드 값 및 목록 항목으로 반환할 필드 값을 사용하여 유사한 관계를 구성할 수 있습니다. 목록 상자 항목의 보조 데이터 소스를 지정하려면 다음과 같이 합니다.
  - 1 목록 상자의 *DataSource* 속성을 컨트롤에 대한 데이터 소스로 설정합니다.

- 2 *DataField* 속성에 대한 드롭다운 목록에서 조회 값을 삽입할 필드를 선택합니다. 선택한 필드는 조회 필드가 될 수 없습니다.
- 3 목록 상자의 *ListSource* 속성을 조회하려는 값의 필드가 포함된 데이터셋에 대한 데이터 소스로 설정합니다.
- 4 *KeyField* 속성에 대한 드롭다운 목록에서 조회 키로 사용할 필드를 선택합니다. 드롭다운 목록은 3 단계에서 지정한 데이터 소스에 연결된 데이터셋에 대한 필드를 표시합니다. 선택한 필드는 색인의 일부가 될 필요는 없지만, 만약 색인의 일부라면 조회 성능이 훨씬 더 빨라집니다.
- 5 *ListField* 속성에 대한 드롭다운 목록에서 값을 반환할 필드를 선택합니다. 드롭다운 목록은 3 단계에서 지정한 데이터 소스에 연결된 데이터셋의 필드를 표시합니다.

조회 컨트롤에 연결된 테이블을 활성화하면, 컨트롤은 목록 항목이 보조 소스로부터 파생된 것을 인식하고 해당 소스로부터 적절한 값을 표시합니다.

*TDBLookupListBox* 컨트롤에 단 한 번 나타나는 항목의 수를 지정하려면 *RowCount* 속성을 사용합니다. 목록 상자의 높이는 이 행 수와 정확히 일치하도록 조정됩니다.

*TDBLookupComboBox*의 드롭다운 목록에 나타나는 항목의 수를 지정하려면 *DropDownRows* 속성을 대신 사용합니다.

**참고** 또한 데이터 그리드의 열을 설정하여 조회 콤보 상자처럼 행동하게 할 수 있습니다. 자세한 내용은 15-20 페이지의 "조회 목록 열 정의"를 참조하십시오.

## 확인란으로 부울 필드 값 처리

*TDBCCheckBox*는 data-aware 확인란 컨트롤입니다. 이 컨트롤은 데이터셋의 부울 필드 값을 설정하는 데 사용될 수 있습니다. 예를 들어, 고객 송장 폼에는 컨트롤이 선택된 경우 고객의 세금이 면제되고, 선택되지 않은 경우 고객의 세금이 면제되지 않는다는 것을 나타내는 확인란 컨트롤이 포함될 수 있습니다.

data-aware 확인란 컨트롤은 *ValueChecked* 및 *ValueUnchecked* 속성에 대한 현재 필드의 값을 비교하여 컨트롤의 선택 여부를 관리합니다. 필드 값이 *ValueChecked* 속성과 일치하면, 컨트롤이 선택된 것입니다. 필드가 *ValueUnchecked* 속성과 일치하면, 컨트롤이 선택되지 않은 것입니다.

**참고** *ValueChecked* 및 *ValueUnchecked*의 값은 동일할 수 없습니다.

사용자가 다른 레코드로 이동할 때 컨트롤이 선택되는 경우, *ValueChecked* 속성을 컨트롤이 데이터베이스에 포스트해야 하는 값으로 설정합니다. 기본적으로 이 값은 "true"로 설정되지만, 사용자의 필요에 따라 적절히 영숫자 값으로 설정할 수 있습니다. *ValueChecked*의 값으로서 세미콜론으로 구분된 항목 목록을 입력할 수도 있습니다. 현재 레코드의 필드 내용과 일치하는 항목이 있는 경우에는 확인란이 선택됩니다. 예를 들어, *ValueChecked* 문자열을 다음과 같이 지정할 수 있습니다.

```
DBCCheckBox1.ValueChecked := 'True;Yes;On';
```

현재 레코드의 필드에 "true", "Yes", 또는 "On" 값이 있으면 확인란이 선택됩니다. *ValueChecked* 문자열의 필드 비교는 대소문자가 구분됩니다. 사용자가 여러 *ValueChecked* 문자열이 있는 상자를 선택하면, 첫 번째 문자열이 데이터베이스에 포스트되는 값입니다.

사용자가 다른 레코드로 이동할 때 컨트롤이 선택되지 않으면, *ValueUnchecked* 속성을 컨트롤이 데이터베이스에 포스트해야 하는 값에 설정합니다. 기본적으로, 이 값은 "false"로 설정되지만 필요에 따라 적절한 영숫자 값으로 설정할 수 있습니다. 또한 *ValueUnchecked* 값으로서 세미콜론으로 구분된 항목 목록을 입력할 수도 있습니다. 현재 레코드의 필드 내용과 일치하는 항목이 있으면, 확인란이 선택되지 않습니다.

data-aware 확인란은 현재 레코드의 필드에 *ValueChecked* 또는 *ValueUnchecked* 속성에 나열된 값이 하나도 포함되지 않을 때마다 사용 불가능이 됩니다.

확인란이 연결되는 필드가 논리 필드라면 필드 내용이 *True*인 경우 항상 선택되고, 필드 내용이 *False*인 경우 선택되지 않습니다. 이런 경우, *ValueChecked* 및 *ValueUnchecked* 속성에 입력된 문자열은 논리 필드에 어떤 영향도 주지 않습니다.

## 라디오 컨트롤로 필드값 제한

*TDBRadioGroup*은 라디오 그룹 컨트롤의 data-aware 버전입니다. 이 컨트롤을 사용하면 필드에 대한 가능한 값의 수가 제한되어 있는 라디오 버튼 컨트롤로 데이터 필드 값을 설정할 수 있습니다. 라디오 그룹에는 필드가 사용할 수 있는 각 값에 대해 버튼이 하나 있습니다. 사용자는 원하는 라디오 버튼을 선택함으로써 데이터 필드에 대한 값을 설정할 수 있습니다.

*Items* 속성은 그룹에 나타나는 라디오 버튼을 결정합니다. *Items*는 문자열 목록입니다. *Items*의 문자열 하나에 대해 라디오 버튼이 하나 표시되고, 각 문자열은 버튼의 레이블로서 라디오 버튼의 오른쪽에 나타납니다.

라디오 그룹에 연결된 필드의 현재 값이 *Items* 속성의 문자열 중 하나와 일치하면, 해당 라디오 버튼이 선택됩니다. 예를 들어, "Red", "Yellow" 및 "Blue"의 세 문자열이 *Items*에 나열되고, 현재 레코드의 필드에 "Blue" 값이 포함되는 경우 그룹의 세 번째 버튼이 선택되어 표시됩니다.

**참고** 필드가 *Items*의 어떤 문자열과도 일치하지 않으면, 필드가 *Values* 속성의 문자열과 일치하는 경우, 라디오 버튼은 여전히 선택될 수 있습니다. 현재 레코드에 대한 필드가 *Items* 또는 *Values*의 어떤 문자열과도 일치하지 않는 경우 라디오 버튼은 선택되지 않습니다.

*Values* 속성은 사용자가 라디오 버튼을 선택하고 레코드를 포스트할 때 데이터셋에 반환될 수 있는 문자열 목록을 옵션으로 포함할 수 있습니다. 문자열은 버튼에 순서대로 연결됩니다. 첫 번째 문자열은 첫 번째 단추, 두 번째 문자열은 두 번째 단추 등의 순으로 연결됩니다. 예를 들어, *Items*에는 "Red", "Yellow" 및 "Blue"가 포함되고, *Values*에는 "Magenta", "Yellow" 및 "Cyan"이 포함된다고 가정하십시오. 사용자가 "Red" 버튼을 선택하면 "Magenta"가 데이터베이스에 포스트됩니다.

*Values*에 대한 문자열이 제공되지 않으면, 레코드가 포스트될 때 선택된 라디오 버튼에 대한 *Item* 문자열이 데이터베이스에 반환됩니다.

## 여러 레코드 표시

경우에 따라 동일한 폼에 많은 레코드를 표시해야 할 수도 있습니다. 예를 들어, 송장 애플리케이션은 동일한 폼에 한 명의 고객이 낸 모든 주문을 보여주어야 합니다.

여러 레코드를 표시하려면 그리드 컨트롤을 사용합니다. 그리드 컨트롤은 애플리케이션의 사용자 인터페이스를 보다 우수하고 효과적으로 만들 수 있는 데이터의 여러 필드 및 여러 레코드 뷰를 제공합니다. 이 내용은 15-15 페이지의 "TDBGrid로 데이터 보기 및 편집"에서 설명합니다.

**참고** 단방향 데이터셋을 사용할 때는 여러 레코드를 표시할 수 없습니다.

여러 레코드를 나타내는 그리드와 단일 레코드의 필드 모두를 표시하는 사용자 인터페이스를 디자인할 수도 있습니다. 이들을 조합하는 두 가지 모델이 있습니다.

- **마스터/디테일 폼:** 단일 필드 및 그리드 컨트롤을 표시하는 두 컨트롤을 포함시켜서 마스터 테이블과 디테일 테이블 모두로부터 정보를 나타낼 수 있습니다. 예를 들어, 한 명의 고객에 대한 정보를 해당 고객에 대한 주문을 표시하는 디테일 그리드로 표시할 수 있습니다. 마스터/디테일 폼의 원본으로 사용한 테이블 연결에 대한 내용은 18-14 페이지의 "마스터/디테일 관계 설정" 및 20-10 페이지의 "클라이언트 데이터셋을 다른 데이터셋의 디테일로 만들기"를 참조하십시오.
- **드릴 다운 폼:** 여러 레코드를 표시하는 폼에서, 현재 레코드에 대한 상세한 정보를 표시하는 단일 필드 컨트롤을 포함할 수 있습니다. 이러한 접근 방식은 특히 레코드에 긴 메모나 그래픽 정보가 있을 때 유용합니다. 사용자가 그리드의 레코드를 통해 스크롤할 때, 메모나 그래픽은 현재 레코드의 값을 나타내도록 업데이트됩니다. 이렇게 설정하는 것은 매우 쉽습니다. 두 개의 화면 표시 사이의 동기화는 그리드와 메모 또는 이미지 컨트롤이 공통적인 데이터 소스를 공유하는 경우 자동으로 처리됩니다.

**팁** 일반적으로 이러한 두 접근 방식을 하나의 폼에 조합하는 것은 좋은 방법이 아닙니다. 때로는 결과가 효과적일 수 있지만, 사용자가 데이터 관계를 이해하는 데 혼란을 줄 수 있습니다.

## TDBGrid로 데이터 보기 및 편집

*TDBGrid* 컨트롤을 사용하여 표 모양의 그리드 형식으로 데이터셋의 레코드를 보고 편집할 수 있습니다.

**그림 15.1** TDBGrid 컨트롤

|         | 현재 필드               | 열 제목               |                  |
|---------|---------------------|--------------------|------------------|
|         | VendorName          | Address1           | City             |
| 레코드 지시자 | Cacor Corporation   | 161 Southfield Rd  | Southfield       |
|         | Underwater          | 50 N 3rd Street    | Indianapolis     |
|         | J.W. Luscher Mfg.   | 65 Addams Street   | Berkely          |
|         | Scuba Professionals | 3105 East Brace    | Rancho Dominguez |
|         | Divers' Supply Shop | 5208 University Dr | Macon            |
|         | Techniques          | 52 Dolphin Drive   | Redwood City     |
|         | Perry Scuba         | 3443 James Ave     | Hapeville        |

그리드 컨트롤에 표시되는 레코드의 외형에 영향을 주는 세 가지 요인은 다음과 같습니다.

- Columns Editor를 사용하는 그리드에 대해 정의된 영구적 열 객체의 존재. 영구적 열 객체는 그리드 및 데이터 모습의 설정에 많은 유용성을 제공합니다. 영구적 열 사용에 대한 내용은 15-16 페이지의 "사용자 지정 그리드 생성"을 참조하십시오.
- 그리드에 표시된 데이터셋에 대한 영구적 필드 컴포넌트의 생성 Fields Editor를 사용한 영구적 필드 컴포넌트 생성에 대한 내용은 17장 "필드 컴포넌트 사용"을 참조하십시오.
- 중첩된 데이터셋 또는 배열 필드를 표시하는 그리드에 대해 설정하는 데이터셋의 *ObjectView* 속성 그리드의 이러한 복합 필드 표시에 대한 내용은 15-21 페이지의 "복합 필드 표시"를 참조하십시오.

그리드 컨트롤은 자체적으로 *TDBGridColumn* 객체의 중첩기인 *Columns* 속성을 가집니다. *TDBGridColumn*는 그리드 컨트롤에 모든 열을 나타내는 *TColumn* 객체의 모음입니다. Columns Editor를 사용하여 디자인 타임에 열 속성을 설정하거나 또는 그리드의 *Columns* 속성을 사용하여 런타임 시 *TDBGridColumn*의 속성, 이벤트 및 메소드에 액세스할 수 있습니다.

### 기본 상태에서 그리드 컨트롤 사용

그리드의 *Columns* 속성의 *State* 속성은 그리드에 대한 영구적 열 객체의 존재 여부를 나타냅니다. *Columns.State*는 그리드에 대해 자동으로 설정되는 런타임 전용 속성입니다. 기본 상태는 *csDefault*이며, 영구적 열 객체는 그리드에 존재하지 않습니다. 이러한 경우, 그리드의 데이터 표시는 주로 그리드의 데이터셋의 필드 속성에 의해 결정되거나 또는 영구적 필드 컴포넌트가 없을 경우에는 표시 특성의 기본 설정에 의해 결정됩니다.

그리드의 *Columns.State* 속성이 *csDefault*인 경우 그리드 열은 데이터셋의 시각적 필드로부터 동적으로 생성되고, 그리드의 열 순서는 데이터셋의 필드 순서와 일치합니다.

그리드의 각 열은 필드 컴포넌트에 연결됩니다. 필드 컴포넌트의 속성 변경은 즉시 그리드에 나타납니다.

동적으로 생성된 열이 있는 그리드 컨트롤을 사용하면 런타임 시 선택된 임의의 테이블의 내용을 보거나 편집하는 데 유용합니다. 그리드의 구조는 설정되어 있지 않기 때문에 다양한 데이터셋에 맞도록 동적으로 변경될 수 있습니다. 동적으로 생성된 열을 가지는 단일 그리드는 Local InterBase 테이블을 표시하고, 그런 다음 그리드의 *DataSource* 속성이 변하거나 데이터 소스의 *DataSet* 속성이 변할 때 MySQL 쿼리 결과를 표시하도록 전환될 수 있습니다.

디자인 타임 또는 런타임에 동적 열의 외형을 변경할 수 있습니다. 그러나 실제로 수정하는 것은 열에 표시된 필드 컴포넌트의 속성입니다. 동적 열의 속성은 열이 단일 데이터셋의 특정 필드에 연결된 경우에만 존재합니다. 예를 들어, 열의 *Width* 속성을 변경하면 해당 열에 연결된 필드의 *DisplayWidth* 속성이 변경됩니다. *Font*와 같이 필드 속성에 기준하지 않는 열 속성에 취해진 변경 사항은 열의 수명 동안만 존재합니다.

그리드의 데이터셋이 동적 필드 컴포넌트로 구성되는 경우, 필드는 데이터셋이 닫힐 때마다 없어집니다. 필드 컴포넌트가 없어지면, 이와 연결된 모든 동적 열도 함께 없어집니다. 그리드의 데이터셋이 영구적 필드 컴포넌트로 구성되는 경우, 필드는 데이터셋이 닫힌 경우에도 존재합니다. 따라서 이들 필드에 연결된 열도 데이터셋이 닫힌 경우라도 속성을 계속 유지합니다.

**참고** 그리드의 *Columns.State* 속성을 런타임 시 *csDefault*로 바꾸면, 영구적 열을 비롯한 그리드의 모든 열 객체가 삭제되고 그리드의 데이터셋의 시각적 필드에 따라 동적 열이 재구성됩니다.

## 사용자 지정 그리드 생성

사용자 지정 그리드는 열이 나타나는 방법과 열의 데이터가 표시되는 방법을 설명하는 영구적 열 객체를 정의하기 위한 것입니다. 사용자 지정 그리드를 사용하면 동일한 데이터셋의 다양한 뷰(예를 들어, 다양한 열 순서, 다양한 필드 선택 및 다양한 열 색상과 글꼴)를 나타내는 여러 그리드를 구성할 수 있습니다. 사용자 지정 그리드를 사용하면, 그리드에서 사용되는 필드나 데이터셋의 필드 순서에 영향을 주지 않고 런타임 시 그리드의 모습을 사용자가 수정할 수도 있습니다.

사용자 지정 그리드는 디자인 타임에 구조가 알려진 데이터셋과 함께 많이 사용됩니다. 사용자 지정 그리드는 디자인 타임에 선언된 필드 이름이 데이터셋에 존재한다고 예상하기 때문에 런타임 시 선택되는 임의의 테이블을 검색하는 데에는 적합하지 않습니다.

## 영구적 열 이해

그리드에 대한 영구적 열 객체를 만들 때, 이 열들은 그리드의 데이터셋의 원본으로 사용한 필드에 느슨하게 연결됩니다. 영구적 열에 대한 기본 속성 값은 열 속성에 값이 할당될 때까지 기본 소스(연결된 필드 또는 그리드 자체)에 동적으로 페치(fetch)됩니다. 열 속성에 값을 할당하기 전까지는 기본 소스가 변경되면 이 값도 변경됩니다. 열 속성에 값을 할당하면 기본 소스가 변경되어도 이 값은 더 이상 변경되지 않습니다.

예를 들어, 열 제목 캡션에 대한 기본 소스는 연결된 필드의 *DisplayLabel* 속성입니다. *DisplayLabel* 속성을 수정하면, 열 제목은 변경 내용을 즉시 반영합니다. 열 제목의 캡

션에 문자열(string)을 할당하면, 제목 캡션은 연결된 필드의 *DisplayLabel* 속성과는 무관하게 됩니다. 필드의 *DisplayLabel* 속성에 대한 추후 변경 사항은 더 이상 열 제목에 영향을 주지 않습니다.

영구적 열은 연결된 필드 컴포넌트와는 독립적으로 존재합니다. 실제로, 영구적 열은 필드 객체에 연결될 필요가 전혀 없습니다. 영구적 열의 *FieldName* 속성이 비어 있거나, 필드 이름이 그리드의 현재 데이터셋의 어떤 필드 이름과도 일치하지 않으면, 열의 *Field* 속성은 NULL이고, 열은 빈 셀로 그려집니다. 셀의 기본 그리기 메소드를 오버라이드하면, 빈 셀에 자신의 사용자 정의 정보를 표시할 수 있습니다. 예를 들어, 빈 열을 사용하여 요약을 집계하는 레코드 그룹의 마지막 레코드에 대한 집계 값을 표시할 수 있습니다. 다른 가능성은 레코드 데이터의 일부를 그래픽으로 묘사하는 비트맵이나 막대도표를 표시하는 것입니다.

두 개 이상의 영구적 열은 데이터셋의 동일한 필드에 연결될 수 있습니다. 예를 들어, 그리드를 스크롤하지 않고도 부분 번호를 찾기 쉽도록 하기 위해 보다 넓은 그리드의 왼쪽과 오른쪽 맨끝에 부분 번호 필드를 표시할 수도 있습니다.

## 참고

영구적 필드는 데이터셋의 필드에 연결될 필요가 없기 때문에, 그리고 여러 열은 동일한 필드를 참조할 수 있기 때문에, 사용자 지정 그리드의 *FieldCount* 속성은 그리드의 열 수보다 작거나 같습니다. 또한 사용자 지정 그리드의 현재 선택된 열이 필드에 연결되지 않은 경우에는 그리드의 *SelectedField* 속성은 Null이고 *SelectedIndex* 속성은 -1임에 주의하십시오.

영구적 열은 그리드 셀을 다른 데이터셋이나 정적 선택 목록의 조회 값의 콤보 상자 드롭다운 목록으로 또는 현재 셀과 관련된 특수 데이터 뷰어나 대화 상자를 실행하기 위해 사용자가 누를 수 있는 셀의 생략 버튼(...)으로 표시하도록 구성될 수 있습니다.

## 영구적 열 생성

디자인 타임에 그리드의 외양을 사용자 정의하려면 Columns Editor를 실행하여 그리드에 대한 영구적 열 객체들을 생성합니다. 런타임 시, 영구적 열 객체가 있는 그리드의 *State* 속성은 *csCustomized*으로 자동 설정됩니다.

그리드 컨트롤에 대한 영구적 열을 생성하려면 다음과 같이 합니다.

- 1 폼에서 그리드 컴포넌트를 선택합니다.
- 2 Object Inspector에서 그리드의 *Columns* 속성을 두 번 눌러서 Columns Editor를 실행합니다.

Columns 목록 상자가 선택된 그리드에 대해 정의된 영구적 열을 표시합니다. Columns Editor를 처음 실행하면, 그리드는 동적 열만 포함하는 기본 상태이기 때문에 목록이 비어 있습니다.

데이터셋의 모든 필드에 대한 영구적 열을 한 번에 생성하거나 영구적 열을 개별적으로 생성할 수 있습니다. 모든 필드에 대한 영구적 열을 생성하려면 다음과 같이 합니다.

- 1 그리드를 마우스 오른쪽으로 눌러서 컨텍스트 메뉴를 표시하고 Add All Fields를 선택합니다. 그리드가 아직 데이터 소스에 연결되지 않은 경우에는 Add All Fields를 사용할 수 없다는 점에 주의하십시오. Add All Fields를 선택하기 전에 활성 데이터셋을 갖고 있는 데이터 소스에 그리드를 연결하십시오.
- 2 그리드에 이미 영구적 열이 있는 경우에는 대화 상자가 나타나 기존의 열을 삭제할지 아니면 열에 첨부할지를 묻습니다. Yes를 선택하면 기존의 영구적 열 정보가 제거되고 필드 이름에 의해 현재 데이터셋의 모든 필드가 데이터셋의 순서대로 삽입됩니다. No를 선택하면 모든 기존의 영구적 열 정보가 유지되고, 새로운 열 정보가 데이터셋의 추가 필드에 따라 데이터셋에 추가됩니다.
- 3 Close를 누르면 그리드에 영구적 열이 적용되고 대화 상자가 닫힙니다.  
영구적 열을 개별적으로 생성하려면 다음과 같이 합니다.

- 1 Columns Editor에서 Add 버튼을 선택합니다. 목록 상자에서 새로운 열이 선택됩니다. 새로운 열에는 순차적인 번호와 기본 이름(예를 들어, 0 - TColumn)이 있습니다.
- 2 이 새로운 열에 필드를 연결하려면 Object Inspector의 *FieldName* 속성을 설정합니다.
- 3 새로운 열의 제목을 설정하려면, Object Inspector의 *Title* 속성을 확장하고 *Caption* 속성을 설정합니다.
- 4 Columns Editor를 닫으면 그리드에 영구적 열이 적용되고 대화 상자가 닫힙니다.  
런타임 시, *Columns.State* 속성에 *csCustomized*를 할당하여 영구적 열로 전환할 수 있습니다. 그리드의 기존 열이 모두 없어지고 새로운 영구적 열이 그리드의 데이터셋의 각 필드에 대해 생성됩니다. 그런 다음 열 목록에 대해 *Add* 메소드를 호출하여 런타임 시 영구적 열을 추가할 수 있습니다.

```
DBGrid1.Columns.Add;
```

## 영구적 열 삭제

그리드에서 영구적 열을 삭제하는 것은 표시하고 싶지 않은 필드를 제거하는 데 유용합니다. 그리드에서 영구적 열을 삭제하려면 다음과 같이 합니다.

- 1 그리드를 두 번 눌러서 Columns Editor를 표시합니다.
- 2 Columns 목록 상자에서 제거할 필드를 선택합니다.
- 3 Delete를 누릅니다. 컨텍스트 메뉴나 *Del* 키를 사용해도 열을 삭제할 수 있습니다.

**참고** 그리드에서 모든 열을 삭제하면, *Columns.State* 속성이 *csDefault* 상태로 되돌아가고 데이터셋의 각 필드에 대해 동적인 열을 자동으로 구성합니다.

런타임 시 열 객체를 해제해도 영구적 열을 삭제할 수 있습니다.

```
DBGrid1.Columns[5].Free;
```



## 영구적 열 순서 정렬

열이 Columns Editor에 나타나는 순서는 열이 그리드에 나타나는 순서와 동일합니다. Columns 목록 상자 내에 열을 끌어다 놓음으로써 열 순서를 변경할 수 있습니다.

열 순서를 변경하려면 다음과 같이 합니다.

- 1 Columns 목록 상자에서 열을 선택합니다.
- 2 열을 목록 상자의 새로운 위치로 끌어 놓습니다.

또한 런타임에서와 같이 실제 그리드의 열 제목을 누르고 열을 새로운 위치로 끌어 놓아 열 순서를 변경할 수 있습니다.

**참고** Fields Editor에서의 영구적 필드를 재정렬하면 사용자 지정 그리드가 아닌 기본 그리드의 열도 재정렬됩니다.

**중요** 디자인 타임에 동적 열과 동적 필드를 모두 포함하는 그리드의 열은 변경된 필드나 열 순서를 기록할 영구적인 것이 아무것도 없기 때문에 재정렬할 수 없습니다.

런타임 시, *DragMode* 속성이 *dmManual*로 설정되어 있으면 사용자는 마우스를 사용하여 열을 그리드의 새로운 위치로 끌어 놓을 수 있습니다. *csDefault* 상태의 *State* 속성으로 그리드의 열을 재정렬하면 그리드의 원본인 데이터셋의 필드 컴포넌트도 재정렬됩니다. 실제 테이블의 필드 순서는 영향을 받지 않습니다. 사용자가 런타임 시 열을 재정렬할 수 없도록 하려면, 그리드의 *DragMode* 속성을 *dmAutomatic*으로 설정합니다.

런타임 시, 그리드의 *OnColumnMoved* 이벤트는 열이 이동된 후에 발생합니다.

## 디자인 타임에 열 속성 설정

열 속성은 해당 열의 셀에 데이터가 표시되는 방법을 결정합니다. 대부분의 열 속성은 그리드나 연결된 필드 컴포넌트와 같은 *default source*라는 다른 컴포넌트에 연결된 속성에서 기본 값을 얻습니다.

열의 속성을 설정하려면, Columns Editor에서 열을 선택하고 Object Inspector에서 속성을 설정합니다. 다음 표는 사용자가 설정할 수 있는 주요 열 속성을 요약해서 보여줍니다.

**표 15.2 열 속성**

| 속성           | 용도                                                                                                                                                                                                                                                     |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Alignment    | 열에서 필드 데이터를 왼쪽, 오른쪽 또는 가운데 정렬합니다. 기본 소스: <i>TField.Alignment</i> .                                                                                                                                                                                     |
| ButtonStyle  | <i>cbsAuto</i> : (기본값) 연결된 필드가 조회 필드이거나 열의 <i>PickList</i> 속성에 데이터가 포함된 경우 드롭다운 목록을 표시합니다.<br><i>cbsEllipsis</i> : 셀의 오른쪽에 생략(...) 단추를 표시합니다. 이 단추를 누르면 그리드의 <i>OnEditButtonClick</i> 이벤트가 발생합니다.<br><i>cbsNone</i> : 일반적인 편집 컨트롤만 사용하여 열의 데이터를 편집합니다. |
| Color        | 열의 셀 배경색을 지정합니다. 기본 소스: <i>TDBGrid.Color</i> . (텍스트 전경 색은 Font 속성 참조)                                                                                                                                                                                  |
| DropDownRows | 드롭다운 목록에서 표시되는 텍스트의 줄 수. 기본값: 7.                                                                                                                                                                                                                       |

표 15.2 열 속성 ( 계속 )

| 속성        | 용도                                                                                 |
|-----------|------------------------------------------------------------------------------------|
| Expanded  | 열을 확장할지 여부를 지정합니다. 단지 혼합 필드를 나타내는 열에 적용됩니다.                                        |
| FieldName | 이 열에 연결된 필드 이름을 지정합니다. 비워 둘 수 있습니다.                                                |
| ReadOnly  | <i>True</i> : 사용자가 열의 데이터를 편집할 수 없습니다.<br><i>False</i> : (기본값) 열의 데이터를 편집할 수 있습니다. |
| Width     | 화면 픽셀로 열의 폭을 지정합니다. 기본 소스: <i>derived from TField.DisplayWidth</i> .               |
| Font      | 열에서 텍스트를 나타내는 데 사용되는 글꼴, 크기 및 색상을 지정합니다. 기본 소스: <i>TDBGrid.Font</i> .              |
| PickList  | 열의 드롭다운 목록에 표시할 값의 목록을 포함합니다.                                                      |
| Title     | 선택된 열의 제목에 대한 속성을 설정합니다.                                                           |

다음 표는 *Title* 속성에 대해 지정할 수 있는 옵션을 요약해서 보여줍니다.

표 15.3 확장된 TColumn Title 속성

| 속성        | 용도                                                                           |
|-----------|------------------------------------------------------------------------------|
| Alignment | 열 제목의 캡션 텍스트를 왼쪽(기본값), 오른쪽, 또는 가운데 정렬합니다.                                    |
| Caption   | 열 제목에 표시할 텍스트를 지정합니다. 기본 소스: <i>TField.DisplayLabel</i> .                    |
| Color     | 열 제목 셀에 그리는 데 사용되는 배경 색을 지정합니다. 기본 소스: <i>TDBGrid.FixedColor</i> .           |
| Font      | 열 제목에 텍스트를 나타내는 데 사용되는 글꼴, 크기 및 색상을 지정합니다. 기본 소스: <i>TDBGrid.TitleFont</i> . |

## 조회 목록 열 정의

조회 콤보 상자 컨트롤과 유사하게 값의 드롭다운 목록을 표시하는 열을 생성할 수 있습니다. 열이 콤보 상자처럼 행동하도록 지정하려면, 열의 *ButtonStyle* 속성을 *cbsAuto* 로 설정합니다. 목록을 값으로 채우고 나면, 해당 열의 셀이 편집 모드로 될 때 그리드가 콤보 상자와 같은 드롭다운 버튼을 자동으로 표시합니다.

사용자가 선택한 값으로 목록을 채우는 방법에는 두 가지가 있습니다.

- 조회 테이블에서 값을 페치(fetch)할 수 있습니다. 열이 독립된 조회 테이블로부터 그려진 값의 드롭다운 목록을 표시하게 하려면, 데이터셋의 조회 필드를 정의해야 합니다. 조회 필드 생성에 대한 내용은 17-8 페이지의 "조회 필드 정의"를 참조하십시오. 일단 조회 필드를 정의하면 열의 *FieldName*을 조회 필드 이름으로 설정합니다. 드롭다운 목록은 조회 필드에서 정의된 조회 값으로 자동으로 채워집니다.
- 디자인 타임에 값 목록을 명시적으로 지정할 수 있습니다. 디자인 타임에 목록 값을 입력하려면, Object Inspector에서 열에 대한 *PickList* 속성을 더블 클릭합니다. 그러면 String List Editor가 나타나서, 열에 대한 선택 목록을 채울 값을 입력할 수 있습니다.

기본적으로, 드롭다운 목록은 7개의 값을 표시합니다. 이 목록의 길이는 *DropDown Rows* 속성을 설정하여 변경할 수 있습니다.

**참고** 명시적인 선택 목록이 있는 열을 정상 행동으로 복구하려면, String List Editor를 사용하여 선택 목록에서 모든 텍스트를 삭제합니다.

## 열에 버튼 두기

열은 생략(...) 버튼을 보통 셀 편집기의 오른쪽에 표시할 수 있습니다. *Ctrl+Enter* 또는 마우스를 누르면, 그리드의 *OnEditButtonClick* 이벤트가 시작됩니다. 생략 버튼을 사용하면 열의 데이터에 대한 상세 뷰가 있는 폼이 나타납니다. 예를 들어, 송장 요약을 표시하는 테이블에서 송장 전체 열에 생략 버튼을 설정하여 해당 송장의 항목, 세금 계산 방법 등을 표시하는 폼을 불러올 수 있습니다. 그래픽 필드의 경우에는 이 생략 버튼을 사용하여 이미지를 표시하는 폼을 불러올 수 있습니다.

열에 생략 버튼을 생성하려면 다음과 같이 합니다.

- 1 *Columns* 목록 상자에서 열을 선택합니다.
- 2 *ButtonStyle*을 *cbsEllipsis*로 설정합니다.
- 3 *OnEditButtonClick* 이벤트 핸들러를 생성합니다.

## 열에 기본값 복구

런타임 시, 열의 *AssignedValues* 속성을 테스트하여 열 속성이 명시적으로 할당되었는지의 여부를 결정할 수 있습니다. 명시적으로 정의되지 않은 값은 연결된 필드나 그리드의 기본값에 동적으로 유도된 값으로 복원됩니다.

하나 이상의 열에 대해 변경된 속성을 실행 취소할 수 있습니다. Columns Editor에서 복구할 열을 선택하고 컨텍스트 메뉴에서 Restore Defaults를 선택합니다. Restore defaults는 할당된 속성 설정을 버리고, 원본으로 사용한 필드 컴포넌트에서 파생된 값으로 열 속성을 복구합니다.

런타임 시, 열의 *RestoreDefaults* 메소드를 호출하면 단일 열에 대한 모든 기본 속성을 재설정할 수 있습니다. 또한 열 목록의 *RestoreDefaults* 메소드를 사용하면 그리드의 모든 열에 대한 기본 속성을 재설정할 수도 있습니다.

```
DBGrid1.Columns.RestoreDefaults;
```

## 복합 필드 표시

경우에 따라 그리드의 데이터셋의 필드는 텍스트, 그래픽, 숫자값 등과 같은 간단한 값을 나타내지 않습니다. 일부 데이터베이스 서버는 ADT 필드나 배열 필드와 같이 보다 단순한 데이터 타입의 복합인 필드를 허용합니다.

그리드가 복합 필드를 표시하는 방법은 두 가지입니다.

- 그리드는 복합 필드 타입을 "평평하게 (flatten out)" 하여 필드를 구성하는 보다 단순한 각각의 타입이 데이터셋에서 독립된 필드로 나타나도록 합니다. 복합 필드가 평평하게 되면, 필드의 구성 요소는 원본으로 사용한 데이터베이스 테이블의 공통 부모 필드의 이름이 각 필드 이름에 선행하는 공통 소스를 반영하는 독립된 필드로서 나타납니다.

복합 필드를 평평하게(flatten out) 된 것처럼 표시하려면 데이터셋의 *ObjectView* 속성을 *False*로 설정합니다. 데이터셋은 복합 필드를 별개의 필드 모음으로 저장하고, 그리드는 각 구성 요소에 독립된 열을 할당하여 이를 반영합니다.

- 이 속성은 단일 열에 복합 필드를 표시하여, 필드가 단일 필드라는 사실을 반영할 수 있습니다. 복합 필드를 단일 열에 표시할 때는 필드의 제목 표시줄의 화살표를 누르거나 열의 *Expanded* 속성을 설정하여 열을 확장하거나 축소할 수 있습니다.
- 열을 확장하는 경우, 각각의 자식 필드는 부모 필드의 제목 표시줄 아래에 열이 분리되어 나타납니다. 즉, 그리드에 대한 제목 표시줄은 높이가 늘어나면서, 첫 번째 행에는 복합 필드의 이름이 표시되고 두 번째 행에서는 각각의 열마다 이름이 표시됩니다. 복합이 아닌 필드는 높이가 늘어난 제목 표시줄로 나타납니다. 이러한 확장은 교대로 복합 필드인 구성 요소에 대해서도 계속되며(예를 들어, 디테일 테이블에 중첩된 디테일 테이블), 그에 따라 제목 표시줄의 높이가 증가합니다.
- 필드가 축소되면, 자식 필드를 포함한 쉽표로 구분된 편집 불가능한 문자열과 함께 하나의 열만이 나타납니다.

확장 및 제거 열에 복합 필드를 표시하려면, 데이터셋의 *ObjectView* 속성을 *True*로 설정합니다. 데이터셋은 중첩된 하위 필드를 포함하는 단일 필드 컴포넌트로 복합 필드를 저장합니다. 그리드는 확장하거나 축소할 수 있는 열에 이를 반영합니다.

**표 15.4** 복합 필드가 나타나는 방식에 영향을 주는 속성

| 속성           | 개체       | 용도                                                                    |
|--------------|----------|-----------------------------------------------------------------------|
| Expandable   | TColumn  | 독립되고 편집 가능한 열로 자식 필드를 나타내도록 열을 확장할 수 있는지 여부를 나타냅니다.(읽기 전용)            |
| Expanded     | TColumn  | 열을 확장할지 여부를 지정합니다.                                                    |
| MaxTitleRows | TDBGrid  | 그리드에 나타날 수 있는 최대 제목 행의 수를 지정합니다.                                      |
| ObjectView   | TDataSet | 필드가 객체 필드를 확장하고 축소할 수 있는 객체 모드로 또는 평평하게(flatten out) 표시되는지 여부를 지정합니다. |
| ParentColumn | TColumn  | 자식 필드의 열을 소유하는 TColumn 객체를 참조합니다.                                     |

**참고** Composite 필드를 비롯한 일부 데이터셋은 다른 데이터셋(데이터셋 필드)이나 다른 데이터셋(참조) 필드의 레코드를 참조하는 필드를 포함합니다. Data-aware 그리드는 이 필드를 "(DataSet)" 또는 "(Reference)"로 각각 표시합니다. 런타임 시, 생략 버튼은 오른쪽에 나타납니다. 생략 버튼을 클릭하면 그리드가 필드 내용을 표시하는 새로운 폼을 불러옵니다. 데이터셋 필드의 경우, 이 그리드는 필드의 값인 데이터셋을 표시합니다. 참조 필드의 경우, 이 그리드는 다른 데이터셋의 레코드를 표시하는 단일 행을 포함합니다.

## 그리드 옵션 설정

디자인 타임에 그리드 *Options* 속성을 사용하면 기본 그리드의 행동과 런타임 시의 외형을 조정할 수 있습니다. 디자인 타임에 그리드 컴포넌트를 처음으로 폼에 두면, Object Inspector의 *Options* 속성이 + 기호로 표시되어, *Options* 속성을 확장하여 개별적으로 설정할 수 있는 일련의 Boolean 속성을 표시할 수 있다는 것을 나타냅니다. 이러한 속성을 보거나 설정하려면, + 기호를 누릅니다. Object Inspector의 옵션 목록이 *Options* 속성 아래에 표시됩니다. - 기호를 누르면 목록이 축소되면서 + 기호로 변경됩니다.

다음 표는 설정할 수 있는 *Options* 속성을 보여주며, 이 속성은 런타임 시 그리드에 영향을 줍니다.

**표 15.5** Expanded TDBGrid Options 속성

| 옵션                    | 용도                                                                                                                                                             |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| dgEditing             | <i>True</i> :(기본값). 그리드 레코드의 편집, 삽입 및 삭제를 가능하게 합니다.<br><i>False</i> : 그리드 레코드의 편집, 삽입 및 삭제를 불가능하게 합니다.                                                         |
| dgAlwaysShowEditor    | <i>True</i> : 필드를 선택하면 Edit 상태가 됩니다.<br><i>False</i> :(기본값). 필드는 선택될 때 자동으로 Edit 상태가 되지 않습니다.                                                                  |
| dgTitles              | <i>True</i> :(기본값). 그리드 상단에 필드 이름을 표시합니다.<br><i>False</i> : 필드 이름 표시가 비활성화 됩니다.                                                                                |
| dgIndicator           | <i>True</i> :(기본값). 지시자 열이 그리드 왼쪽에 표시되고, 현재 레코드 지시자(그리드 왼쪽의 화살표)가 활성화되어 현재 레코드를 보여줍니다. 삽입 시, 화살표는 별표가 됩니다. 편집 시, 화살표는 I 빔이 됩니다.<br><i>False</i> : 지시자 열이 꺼집니다. |
| dgColumnResize        | <i>True</i> :(기본값). 제목 부분의 열 조정자를 끌어서 열 크기를 조정할 수 있습니다. 열 크기를 조정하면 원본으로 사용한 <i>TField</i> 컴포넌트의 폭을 변경할 수 있습니다.<br><i>False</i> : 그리드에서 열 크기를 조정할 수 없습니다.       |
| dgColLines            | <i>True</i> :(기본값). 열 사이의 수직 구분선을 표시합니다.<br><i>False</i> : 열 사이의 구분 선을 표시하지 않습니다.                                                                              |
| dgRowLines            | <i>True</i> :(기본값). 레코드 사이의 수평 구분 선을 표시합니다.<br><i>False</i> : 레코드 사이의 구분 선을 표시하지 않습니다.                                                                         |
| dgTabs                | <i>True</i> :(기본값). 레코드의 필드 사이에 탭 이동을 가능하게 합니다.<br><i>False</i> : 탭을 누르면 그리드 컨트롤을 빠져나갑니다.                                                                      |
| dgRowSelect           | <i>True</i> : 선택 막대가 그리드의 전체 폭에 미칩니다.<br><i>False</i> :(기본값). 레코드의 필드를 선택하면 해당 필드만 선택됩니다.                                                                      |
| dgAlwaysShowSelection | <i>True</i> :(기본값). 다른 컨트롤에 포커스가 있는 경우에도 그리드의 선택 막대는 항상 보입니다.<br><i>False</i> : 그리드에 포커스가 있는 경우에만 그리드의 선택 막대가 보입니다.                                            |

**표 15.5** Expanded TDBGrid Options 속성 ( 계속 )

| 옵션              | 용도                                                                                                                                                |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| dgConfirmDelete | <i>True</i> :(기본값). 레코드 삭제 ( <i>Ctrl+Del</i> ) 시 사용자에게 확인을 요구합니다.<br><i>False</i> : 확인 없이 레코드를 삭제합니다.                                             |
| dgCancelOnExit  | <i>True</i> :(기본값). 그리드에서 포커스가 사라지면 보류 중인 삽입을 취소합니다. 이 옵션은 부분 또는 빈 레코드의 우연한 포스트를 방지합니다.<br><i>False</i> : 보류 중인 삽입을 허용합니다.                        |
| dgMultiSelect   | <i>True</i> : 사용자가 <i>Ctrl+Shift</i> 또는 <i>Shift+화살표</i> 키를 사용하여 그리드의 비연속적인 행을 선택할 수 있도록 합니다.<br><i>False</i> :(기본값). 사용자가 행을 여러 개 선택할 수 없도록 합니다. |

## 그리드 편집

런타임 시, 다음의 기본 조건이 만족되는 경우에는 그리드를 사용하여 기존의 데이터를 수정하거나 새로운 레코드를 입력할 수 있습니다.

- *Dataset*의 *CanModify* 속성은 *True*입니다.
- 그리드의 *ReadOnly* 속성은 *False*입니다.

사용자가 그리드의 레코드를 편집할 때, 각 필드의 변경 사항은 내부 레코드 버퍼에 포스트됩니다. 변경 내용은 사용자가 그리드의 다른 레코드로 이동한 다음에 포스트됩니다. 폼의 다른 컨트롤로 포커스가 변경되어도, 데이터셋에 대한 커서가 다른 레코드로 이동할 때까지 그리드는 변경 사항을 포스트하지 않습니다. 레코드가 포스트되는 경우 데이터셋은 모든 연결된 data-aware 컴포넌트의 상태 변경을 확인합니다. 수정된 데이터를 포함하는 필드를 업데이트하는 데 문제가 발생하면, 예외를 발생시키고 레코드를 수정하지 않습니다.

**참고** 레코드 변경 사항을 클라이언트 데이터셋에 포스트하면 이를 클라이언트 데이터셋의 내부 변경 로그에 추가만 합니다. 이러한 변경 사항은 클라이언트 데이터셋이 업데이트를 적용해야만 원본으로 사용한 데이터베이스에 포스트됩니다.

다른 레코드로 이동하기 전에 필드에서 *Esc*를 누르면 레코드의 모든 편집 사항이 취소됩니다.

## 그리드 그리기 제어

그리드 컨트롤이 자신을 그리는 방법에 대한 첫 번째 수준의 조정은 열 속성을 설정하는 것입니다. 그리드는 자동으로 열의 글꼴, 색상 및 정렬 속성을 사용하여 해당 열의 셀을 그립니다. 열에 연결된 필드 컴포넌트의 *DisplayFormat* 또는 *EditFormat* 속성을 사용하여 데이터 필드의 텍스트를 그립니다.

기본 그리드 표시 로직을 그리드의 *OnDrawColumnCell* 이벤트 코드의 인수로 전달할 수 있습니다. 그리드의 *DefaultDrawing* 속성이 *True*이면, 모든 일반적인 그리기는 *OnDrawColumnCell* 이벤트 핸들러가 호출되기 전에 수행됩니다. 그런 다음 코드가 기

본 표시 위에 그려질 수 있습니다. 이는 빈 영구적 열을 정의하고 해당 열의 셀에 특수 값을 그리려고 할 때 주로 유용합니다.

그리드의 그리기 로직을 완전히 바꾸고자 하면, *DefaultDrawing*을 *False*로 설정하고 그리기 코드를 그리드의 *OnDrawColumnCell* 이벤트에 둡니다. 그리기 로직을 특정 열이나 특정 필드 데이터 타입에 대해서만 교체하려면 *OnDrawColumnCell* 이벤트 핸들러 내의 *DefaultDrawColumnCell* 메소드를 호출하여 그리드가 선택된 열에 대해 일반적인 그리기 코드를 이용하도록 할 수 있습니다. 예를 들어, 부울 필드 타입에 대해서만 그리기 방식을 변경하려는 경우에 사용자가 해야 하는 작업의 양이 줄어듭니다.

## 런타임 시 사용자 동작에 응답

그리드 내의 특정 동작에 응답하도록 이벤트 핸들러를 작성하여 런타임 시 그리드 동작을 수정할 수 있습니다. 그리드는 보통 한 번에 많은 필드와 레코드를 표시하기 때문에, 개별적인 열에 대한 변경 사항에 응답해야 할 필요가 있을 수 있습니다. 예를 들어, 사용자가 특정 열을 입력하고 열을 빠져나갈 때마다 사용자는 폼에서 버튼을 활성화하고 비활성화할 수도 있습니다.

다음 표는 Object Inspector에서 사용할 수 있는 그리드 이벤트를 보여줍니다.

**표 15.6** 그리드 컨트롤 이벤트

| 이벤트               | 용도                                                                            |
|-------------------|-------------------------------------------------------------------------------|
| OnCellClick       | 사용자가 그리드의 셀을 누를 때 발생합니다.                                                      |
| OnColEnter        | 사용자가 그리드의 열로 이동할 때 발생합니다.                                                     |
| OnColExit         | 사용자가 그리드의 열을 떠날 때 발생합니다.                                                      |
| OnColumnMoved     | 사용자가 새로운 위치로 열을 이동할 때 발생합니다.                                                  |
| OnDblClick        | 사용자가 그리드를 두 번 누를 때 발생합니다.                                                     |
| OnDragDrop        | 사용자가 그리드에서 끌어다 놓을 때 발생합니다.                                                    |
| OnDragOver        | 사용자가 그리드 위로 끌 때 발생합니다.                                                        |
| OnDrawColumnCell  | 애플리케이션이 개별적인 셀을 그릴 필요가 있을 때 발생합니다.                                            |
| OnDrawDataCell    | (폐기됨) <i>State</i> 가 <i>csDefault</i> 인 경우 애플리케이션이 개별적인 셀을 그릴 필요가 있을 때 발생합니다. |
| OnEditButtonClick | 사용자가 열에서 생략 버튼을 눌렀을 때 발생합니다.                                                  |
| OnEndDrag         | 사용자가 그리드에서 끌기를 멈추었을 때 발생합니다.                                                  |
| OnEnter           | 그리드가 포커스를 얻었을 때 발생합니다.                                                        |
| OnExit            | 그리드가 포커스를 잃었을 때 발생합니다.                                                        |
| OnKeyDown         | 그리드에 포커스가 있을 때, 사용자가 키보드에서 하나의 키나 키 조합을 눌렀을 때 발생합니다.                          |
| OnKeyPress        | 그리드에 포커스가 있을 때 사용자가 키보드에서 단일 영숫자 키를 눌렀을 때 발생합니다.                              |
| OnKeyUp           | 사용자가 그리드에서 키를 놓았을 때 발생합니다.                                                    |
| OnMouseDown       | 사용자가 그리드에서 마우스를 눌렀을 때 발생합니다.                                                  |
| OnMouseMove       | 사용자가 그리드 위로 마우스를 움직였을 때 발생합니다.                                                |
| OnMouseUp         | 사용자가 그리드 위에서 마우스 버튼을 놓았을 때 발생합니다.                                             |

표 15.6 그리드 컨트롤 이벤트 ( 계속 )

| 이벤트          | 용도                            |
|--------------|-------------------------------|
| OnStartDrag  | 사용자가 그리드에서 끌여가기를 시작할 때 발생합니다. |
| OnTitleClick | 사용자가 열에 대한 제목을 눌렀을 때 발생합니다.   |

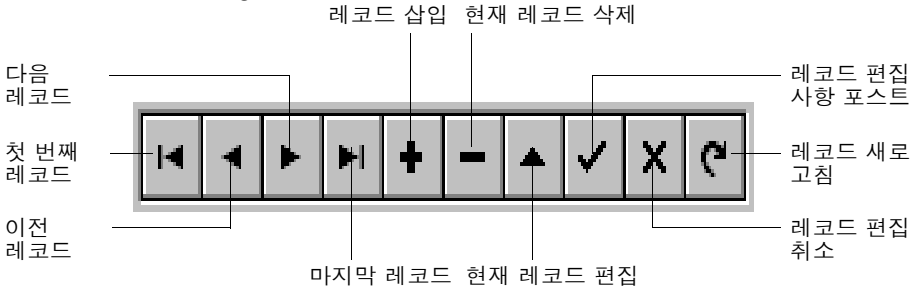
이러한 이벤트는 여러 가지 용도로 사용됩니다. 예를 들어, *OnDbClick* 이벤트에 대한 핸들러를 작성하여 사용자가 열에 값을 입력하려고 할 때 가능한 값이 있는 팝업 목록이 표시되도록 할 수 있습니다. 이러한 핸들러는 *SelectedField* 속성을 사용하여 현재 행과 열을 결정합니다.

## 레코드 탐색 및 처리

*TDBNavigator*는 사용자에게 데이터셋의 레코드를 탐색하고 처리하는 간단한 컨트롤을 제공합니다. 탐색기는 일련의 버튼으로 구성됩니다. 이 버튼을 이용하여 사용자는 한 레코드씩 앞으로 또는 뒤로 스크롤하고, 첫 번째 레코드 또는 마지막 레코드로 이동하며, 새로운 레코드를 삽입하고, 기존 레코드를 업데이트하고, 데이터 변경 사항을 포스트하고, 데이터 변경을 취소하고, 레코드를 삭제하며, 레코드 표시를 새로 고칠 수 있습니다.

그림 15.2는 디자인 타임에 폼에 추가할 때 기본으로 나타나는 탐색기를 보여줍니다. 탐색기는 사용자가 데이터셋의 한 레코드에서 다른 레코드로 이동하거나, 레코드를 편집, 삭제, 삽입 및 포스트할 수 있게 해주는 일련의 버튼으로 구성됩니다. 탐색기의 *VisibleButtons* 속성은 사용자가 이러한 버튼의 일부를 동적으로 숨기거나 보이게 합니다.

그림 15.2 TDBNavigator 컨트롤의 버튼



다음 표는 탐색기의 버튼을 설명합니다.

표 15.7 TDBNavigator 버튼

| 버튼   | 용도                                                                            |
|------|-------------------------------------------------------------------------------|
| 첫 번째 | 데이터셋의 <i>First</i> 메소드를 호출하여 현재 레코드를 첫 번째 레코드로 설정합니다.                         |
| 이전   | 데이터셋의 <i>Prior</i> 메소드를 호출하여 현재 레코드를 이전 레코드로 설정합니다.                           |
| 다음   | 데이터셋의 <i>Next</i> 메소드를 호출하여 현재 레코드를 다음 레코드로 설정합니다.                            |
| 마지막  | 데이터셋의 <i>Last</i> 메소드를 호출하여 현재 레코드를 마지막 레코드로 설정합니다.                           |
| 삽입   | 데이터셋의 <i>Insert</i> 메소드를 호출하여 현재 레코드 앞에 새로운 레코드를 삽입하고 데이터셋을 Insert 상태로 설정합니다. |



표 15.7 TDBNavigator 버튼 ( 계속 )

| 버튼    | 용도                                                                                         |
|-------|--------------------------------------------------------------------------------------------|
| 삭제    | 현재 레코드를 삭제합니다. <i>ConfirmDelete</i> 속성이 <i>True</i> 인 경우, 삭제하기 전에 사용자에게 확인을 요구합니다.         |
| 편집    | 데이터셋을 Edit 상태로 만들어 현재 레코드를 수정할 수 있도록 합니다.                                                  |
| 포스트   | 현재 레코드의 변경 사항을 데이터베이스에 적용합니다.                                                              |
| 취소    | 현재 레코드의 편집을 취소하고 데이터셋을 Browse 상태로 만듭니다.                                                    |
| 새로 고침 | 데이터 컨트롤 표시 버퍼를 지우고, 실제 테이블이나 쿼리로부터 버퍼를 새로 고칩니다. 원본으로 사용한 데이터가 다른 애플리케이션에 의해 변경된 경우에 유용합니다. |

## 표시할 탐색기 버튼 선택

디자인 타임에 *TDBNavigator*를 폼에 처음으로 두면, 모든 버튼이 보입니다. *VisibleButtons* 속성을 사용하면 폼에서 사용하고 싶지 않은 버튼을 보이지 않게 할 수 있습니다. 예를 들어, 단방향 데이터셋에서 작업하는 경우에, 첫 번째, 다음 및 새로 고침 버튼만 보여주고, 다른 버튼을 숨기고자 할 것입니다. 편집보다는 탐색을 목적으로 하는 폼에서는 *편집*, *삽입*, *삭제*, *포스트* 및 *취소* 버튼을 사용 못하게 할 수도 있습니다.

## 디자인 타임에 탐색기 버튼 보이기 및 숨기기

Object Inspector의 *VisibleButtons* 속성은 + 기호와 함께 표시되며 속성을 확장하여 탐색기의 각 버튼에 대한 부울 값을 표시할 수 있습니다. 이들 값을 보거나 설정하려면 + 기호를 누릅니다. 커거나 끌 수 있는 버튼 목록이 *VisibleButtons* 속성 아래의 Object Inspector에 나타납니다. - 기호를 누르면 속성 목록이 축소되면서 + 기호로 바뀝니다.

버튼의 가시성(visibility)은 버튼 값의 *Boolean* 상태에 의해 결정됩니다. 값이 *True*로 설정된 경우, 버튼은 *TDBNavigator*에 나타납니다. 값이 *False*인 경우, 버튼은 디자인 타임 및 런타임에 탐색기에서 제거됩니다.

**참고** 버튼 값을 *False*로 설정하면, 폼의 *TDBNavigator*에서 버튼이 제거되며, 남아 있는 버튼은 폭이 넓어져서 컨트롤을 채웁니다. 컨트롤의 핸들을 마우스로 끌어서 버튼의 크기를 조정합니다.

## 런타임 시 탐색기 버튼 보이기 및 숨기기

런타임 시, 사용자의 동작이나 애플리케이션의 상태에 따라 탐색기 버튼을 보이게 하거나 숨길 수 있습니다. 예를 들어, 하나는 사용자가 레코드를 편집하도록 허용하고 다른 하나는 읽기 전용인 두 개의 다른 데이터셋의 검색을 위한 단일 탐색기를 제공한다고 가정하십시오. 데이터셋 사이를 전환할 때 읽기 전용인 데이터셋에 대해서는 탐색기의 *삽입*, *삭제*, *편집*, *포스트*, *취소* 및 *새로 고침* 버튼을 숨기고, 다른 데이터셋에 대해서는 이 버튼들을 보여주려고 합니다.

예를 들어, 탐색기의 *삽입*, *삭제*, *편집*, *포스트*, *취소* 및 *새로 고침* 버튼을 숨겨서 *OrdersTable*에 대한 편집을 막거나, *CustomersTable*에 대해서는 편집을 허용하는 경우를 가정하십시오. *VisibleButtons* 속성은 어느 버튼을 탐색기에 표시할지를 제어합니다. 다음 코드는 *OnEnter* 이벤트 핸들러를 코딩하는 방법을 보여줍니다.

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    begin
      DBNavigatorAll.DataSource := CustomerCompany.DataSource;
      DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
    end
  else
    begin
      DBNavigatorAll.DataSource := OrderNum.DataSource;
      DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
        nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
    end;
end;
```

## 풍선 도움말 표시

---

런타임 시 각 탐색기 버튼에 대한 풍선 도움말을 표시하려면, 탐색기의 *ShowHint* 속성을 *True*로 설정합니다. *ShowHint*가 *True*인 경우 마우스 커서를 탐색기 버튼 위로 가져갈 때마다 풍선 도움말 힌트가 표시됩니다. *ShowHint*는 기본값이 *False*입니다.

*Hints* 속성은 각 버튼에 대한 풍선 도움말 텍스트를 제어합니다. 기본적으로 *Hints*는 빈 문자열 목록입니다. *Hints*가 비어 있으면, 각 탐색기 버튼은 기본 도움말 텍스트를 표시합니다. 탐색기 버튼에 사용자 지정 풍선 도움말을 제공하려면, String List Editor를 사용하여 *Hints* 속성의 각 버튼에 대해 독립된 힌트 텍스트 줄을 입력합니다. 텍스트를 입력하면 탐색기 컨트롤에서 제공한 기본 힌트 대신 사용자가 제공한 문자열이 풍선 도움말에 나타납니다.

## 여러 데이터셋에 단일 탐색기 사용

다른 data-aware 컨트롤에서와 같이 탐색기의 *DataSource* 속성은 컨트롤을 데이터셋에 연결하는 데이터 소스를 지정합니다. 런타임 시 탐색기의 *DataSource* 속성을 변경하면, 단일 탐색기로 여러 데이터셋에 대한 레코드를 탐색하거나 처리할 수 있습니다.

각각의 *CustomersSource* 및 *OrdersSource* 데이터 소스를 통해 *CustomersTable* 및 *OrdersTable* 데이터셋에 연결된 두 개의 편집 컨트롤이 폼에 있다고 가정하십시오. 사용자가 *CustomersSource*에 연결된 편집 컨트롤을 입력하면 탐색기는 또한 *CustomersSource*를 사용할 수 있고, 사용자가 *OrdersSource*에 연결된 편집 컨트롤을 입력하면 탐색기는 또한 *OrdersSource*로 전환할 수 있습니다. 편집 컨트롤 중 하나에 대해 *OnEnter* 이벤트 핸들러를 코딩할 수 있고, 그런 다음 이 이벤트를 다른 편집 컨트롤과 공유할 수 있습니다. 예를 들면, 다음과 같습니다.

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```



# 16

## 데이터셋 이해

데이터 액세스의 기본 유닛은 객체의 데이터셋 패밀리에 있습니다. 애플리케이션은 모든 데이터베이스 액세스를 위해 데이터셋을 사용합니다. 데이터셋 객체는 논리 테이블로 구성된 데이터베이스의 레코드 집합을 나타냅니다. 이러한 레코드는 데이터베이스에 액세스하는 쿼리나 내장 프로시저 또는 다른 데이터셋의 레코드일 수 있습니다.

데이터베이스 애플리케이션에서 사용하는 모든 데이터셋 객체는 가상화된(virtualized) 데이터셋 객체인 *TDataSet*의 자손이고, *TDataSet*의 데이터 필드, 속성, 이벤트 및 메소드를 상속 받습니다. 이 장에서는 데이터베이스 애플리케이션에서 사용할 데이터셋 객체가 상속하는 *TDataSet*의 기능에 대해 설명합니다. 데이터셋 객체를 사용하려면 이러한 공유 기능을 이해해야 합니다.

*TDataSet*은 가상화된 데이터셋인데 이는 속성과 메소드의 많은 부분이 **virtual** 또는 **abstract**임을 의미합니다. *가상 메소드*는 자손 객체에 메소드의 구현을 오버라이드할 수 있는(대개는 오버라이드되는) 함수 또는 프로시저 선언입니다. 실제 구현이 없는 함수 또는 프로시저 선언을 *추상 메소드*라고 합니다. 선언은 모든 자손 데이터셋 객체에 구현되어야 하지만, 객체마다 다르게 구현될 수 있는 메소드(매개변수와 반환 타입)의 프로토타입입니다.

*TDataSet*은 **abstract** 메소드를 포함하기 때문에 런타임 오류를 발생하지 않고 애플리케이션에서 직접 사용할 수 없습니다. 대신, *TClientDataSet*, *TSQLClientDataSet*, *TSQLDataSet*, *TSQLQuery*, *TSQLTable* 및 *TSQLStoredProc* 같은 *TDataSet* 자손들의 인스턴스를 생성하거나, *TDataSet* 또는 그 자손에서 직접 데이터셋 객체를 파생시킨 다음, 모든 **abstract** 메소드에 대한 구현을 작성합니다.

*TDataSet*은 모든 데이터셋 객체에 공통적인 많은 부분을 정의합니다. 예를 들어, *TDataSet*은 하나 이상의 데이터베이스 테이블의 실제 열에 해당하는 *TField* 컴포넌트의 배열, 애플리케이션에서 제공하는 조회(lookup) 필드 또는 애플리케이션에서 제공하는 계산된 필드 등 모든 데이터셋의 기본 구조를 정의합니다. *TField* 컴포넌트에 대한 자세한 내용은 17장 "필드 컴포넌트 사용"을 참조하십시오.

이 장은 *TDataSet*에 선언된 공통 데이터베이스 기능의 사용 방법을 설명합니다. 그러나 *TDataSet*이 이 기능에 대한 메소드를 선언하더라도 모든 *TDataSet* 자손이 의미있는 방식으로 그 메소드를 구현하는 것은 아니라는 점에 유의하십시오.

## 데이터셋 형식

*TDataSet*은 *TCustomClientDataSet* 및 *TCustomSQLDataSet*의 두 가지 직계 자손을 가집니다. 예를 들어 데이터베이스 서버가 아닌 프로세스에서 데이터를 공급하기 위해 사용자 지정 *TDataSet* 자손을 생성할 수 있습니다.

*TCustomClientDataSet*은 모든 클라이언트 데이터셋에 대한 기본 클래스입니다. 이 기본 클래스는 많은 중요한 속성들이 *protected*이기 때문에 애플리케이션에서 직접 사용되지는 않습니다. 클라이언트 데이터셋은 데이터와 업데이트 내용을 메모리에 버퍼링하여, 사용자가 데이터셋을 검색하거나 값을 편집하여 이를 다시 데이터베이스 서버에 적용할 때 사용할 수 있습니다. 클라이언트 데이터셋은 *TDataSet*에서 선언된 대부분의 기능을 구현할 뿐만 아니라 유지 관리된 집계 (maintained aggregates)와 같은 추가 기능을 선언합니다. 클라이언트 데이터베이스에서 선언하는 기능에 대한 내용은 20장 "클라이언트 데이터셋 사용"을 참조하십시오.

*TCustomSQLDataSet*은 모든 단방향 데이터셋의 기본 클래스입니다. 이 클래스는 액세스할 데이터를 지정하는 속성이 모두 *protected*이기 때문에 애플리케이션에 직접 사용할 수 없습니다. 단방향 데이터셋은 사용자가 다음 레코드로의 이동이 아닌 탐색을 시도할 때 예외를 발생시킵니다. 단방향 데이터셋은 모든 데이터 버퍼링 또는 데이터 버퍼링이 필요한 기능(예를 들어 데이터, 필터 및 조회 필드 업데이트)을 지원합니다. 애플리케이션에서 사용할 수 있는 *TCustomSQLDataSet* 자손에 대한 내용은 18장 "단방향 데이터셋 사용"을 참조하십시오.

각각의 *TDataSet* 자손에는 장단점이 있습니다. 데이터베이스 애플리케이션에 이러한 데이터셋 타입을 구축하는 방법에 대한 개요는 14-4 페이지의 "데이터베이스 아키텍처"를 참조하십시오.

## 데이터셋 열기와 닫기

데이터셋의 데이터를 읽거나 편집하려면, 애플리케이션이 먼저 이를 열어야 합니다. 데이터셋은 두 가지 방법으로 열 수 있습니다.

- 디자인 타임에 Object Inspector에서 또는 런타임 시 코드에서 데이터셋의 *Active* 속성을 *True*로 설정합니다.

```
CustTable.Active := True;
```

- 런타임 시에 데이터셋에 대한 *Open* 메소드를 호출합니다.

```
CustQuery.Open;
```

데이터셋은 두 가지 방법으로 닫을 수 있습니다.

- 디자인 타임에 Object Inspector에서 또는 런타임 시 코드에서 데이터셋의 *Active* 속성을 *False*로 설정합니다.

```
CustQuery.Active := False;
```

- 런타임 시 데이터셋에 대한 *Close* 메소드를 호출합니다.

```
CustTable.Close;
```

*TSQLDataSet* 컴포넌트의 *CommandText*와 같이 특정 속성을 변경하는 경우 데이터셋을 닫아야 할 수도 있습니다. 데이터셋을 다시 열 때, 새로운 속성 값이 적용됩니다.

## 데이터셋 상태 결정 및 설정

데이터셋의 **상태**(또는 **모드**)는 데이터에 수행될 작업을 결정합니다. 예를 들어, 데이터셋이 닫혀 있을 때 데이터셋의 상태는 *dsInactive*이며, 이는 데이터에 어떤 작업도 수행될 수 없다는 것을 의미합니다. 런타임 시 데이터셋의 읽기 전용 *State* 속성을 검사하여 현재 상태를 알 수 있습니다. 다음 표는 *State* 속성에 대한 가능한 값과 의미를 요약해서 보여줍니다.

표 16.1 데이터셋 State 속성의 값

| 값                     | 상태            | 의미                                                                                                  |
|-----------------------|---------------|-----------------------------------------------------------------------------------------------------|
| <i>dsInactive</i>     | Inactive      | DataSet이 닫힙니다. 데이터를 사용할 수 없습니다.                                                                     |
| <i>dsBrowse</i>       | Browse        | DataSet이 열립니다. 데이터를 볼 수는 있지만 변경할 수 없습니다. 개방형 데이터셋의 기본 상태입니다.                                        |
| <i>dsEdit</i>         | Edit          | DataSet이 열립니다. 현재 행을 수정할 수 있습니다. 단방향 데이터셋에서는 지원되지 않습니다.                                             |
| <i>dsInsert</i>       | Insert        | DataSet이 열립니다. 새 행이 삽입되거나 추가됩니다. 단방향 데이터셋에서는 지원되지 않습니다.                                             |
| <i>dsSetKey</i>       | SetKey        | <i>TClientDataSet</i> 만 해당. DataSet이 열립니다. 범위 설정 및 범위와 <i>GotoKey</i> 작동에 사용되는 키 값의 설정을 가능하게 합니다.   |
| <i>dsCalcFields</i>   | CalcFields    | DataSet이 열립니다. <i>OnCalcFields</i> 이벤트가 진행 중임을 나타냅니다. 계산되지 않은 필드에 대한 변경을 방지합니다.                     |
| <i>dsCurValue</i>     | CurValue      | 내부 사용만 해당.                                                                                          |
| <i>dsNewValue</i>     | NewValue      | 내부 사용만 해당.                                                                                          |
| <i>dsOldValue</i>     | OldValue      | 내부 사용만 해당.                                                                                          |
| <i>dsFilter</i>       | Filter        | DataSet이 열립니다. 필터 작동이 진행 중임을 나타냅니다. 제한된 데이터 집합을 볼 수 있지만, 어떤 데이터도 변경될 수 없습니다. 단방향 데이터셋에서는 지원되지 않습니다. |
| <i>dsBlockRead</i>    | Block Read    | DataSet이 열립니다. data-aware 컨트롤이 업데이트되지 않고 현재 레코드가 변경될 때 이벤트가 트리거되지 않습니다.                             |
| <i>dsInternalCalc</i> | Internal Calc | DataSet이 열립니다. <i>OnCalcFields</i> 이벤트가 레코드로 저장된 계산된 값에 대해 진행 중에 있습니다. (클라이언트 데이터셋만 해당)             |

애플리케이션이 데이터셋을 열 때, 기본적으로 *dsBrowse* 모드로 나타납니다. 데이터셋의 상태는 애플리케이션이 데이터를 처리하는 대로 변경됩니다. 개방형 (open) 데이터셋은 다음 중 하나에 따라 한 상태에서 다른 상태로 변경됩니다.

- 애플리케이션의 코드
- 데이터 관련 컴포넌트의 기본 제공 행동

데이터셋을 *dsBrowse*, *dsEdit*, *dsInsert*, *dsSetKey* 또는 *dsBlockRead* 상태로 두려면, 메소드를 호출하거나 상태 이름에 해당하는 속성을 설정합니다. 예를 들어, 다음 코드는 *CustTable*을 *dsInsert* 상태로 두고, 새로운 레코드에 대한 사용자 입력을 사용하며, 새 레코드를 변경 로그에 작성합니다.

```
CustTable.Insert; { Your application explicitly sets dataset state to Insert }
AddressPromptDialog.ShowModal;
if (AddressPromptDialog.ModalResult = mrOK) then
    CustTable.Post { Kylix sets dataset state to Browse on successful completion }
else
    CustTable.Cancel; {Kylix sets dataset state to Browse on cancel }
```

이 예제는 또한 다음의 경우에 데이터셋의 상태가 자동으로 *dsBrowse*로 변경되는 것을 보여줍니다.

- *Post* 메소드가 레코드를 변경 로그에 성공적으로 작성할 때. *Post*가 실패하는 경우 데이터셋 상태는 변경되지 않은 상태로 남습니다.
- *Cancel* 메소드가 호출될 때.

일부 상태는 직접 설정될 수 없습니다. 예를 들어, 데이터셋을 *dsInactive* 상태로 두려면, *Active* 속성을 *False*로 설정하거나 데이터셋에 대해 *Close* 메소드를 호출합니다. 다음의 문장들은 동일합니다.

```
CustTable.Active := False;
CustTable.Close;
```

나머지 상태 (*dsCalcFields*, *dsCurValue*, *dsNewValue*, *dsOldValue*, *dsFilter* 및 *dsInternalCalc*)는 사용자의 애플리케이션에서 설정될 수 없습니다. 그 대신, 데이터셋의 상태는 필요에 따라 자동으로 해당 값으로 변경됩니다. 예를 들어, *dsCalcFields*는 데이터셋의 *OnCalcFields* 이벤트가 발생할 때 설정됩니다. *OnCalcFields* 이벤트가 종료되면, 데이터셋이 이전의 상태로 복구됩니다.

**참고** 데이터셋의 상태가 변경될 때마다 *OnStateChange* 이벤트는 데이터셋에 연결된 모든 데이터 소스 컴포넌트에 대해 호출됩니다. 데이터 소스 컴포넌트와 *OnStateChange*에 대한 자세한 내용은 15-4 페이지의 "데이터 소스에서 변경 사항에 대한 응답"을 참조하십시오.

다음 단원은 해당하는 경우 대부분의 일반적인 상태, 상태 설정 방법과 시기, 상태의 상호 연관 및 관련 정보의 이동 등에 대한 개요를 제공합니다.

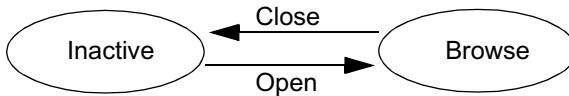
## 데이터셋 비활성화

데이터셋은 닫혀질 때 비활성화됩니다. 닫힌 데이터셋의 레코드에 액세스할 수 없습니다. 디자인 타임에 데이터셋은 *Active* 속성을 *True*로 설정할 때까지 닫혀 있습니다. 런타임 시, 애플리케이션이 *Open* 메소드를 호출하거나 *Active* 속성을 *True*로 설정하여 열 때까지 데이터셋은 처음에 닫혀 있습니다.

비활성 데이터셋을 열면 상태가 자동으로 *dsBrowse* 상태로 변경됩니다. 다음의 그림은 이러한 상태와 이를 설정하는 메소드와의 관계를 보여줍니다.



그림 16.1 Inactive 및 Browse 상태의 관계



데이터셋을 비활성으로 만들려면 *Close* 메소드를 호출합니다. 데이터셋에 대한 *Close* 메소드에 응답하는 *BeforeClose* 및 *AfterClose* 이벤트 핸들러를 작성할 수 있습니다. 예를 들어, 애플리케이션이 데이터셋이 *Close*를 호출할 때, 데이터셋이 *dsEdit* 또는 *dsInsert* 모드인 경우에는 데이터셋을 닫기 전에 보류 중인 변경 내용을 포스트하도록 표시하거나 이를 취소하도록 사용자에게 메시지를 표시할 수 있습니다. 다음 코드는 이러한 핸들러를 보여줍니다.

```

procedure TForm1.CustTableVerifyBeforeClose(DataSet:TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation,
      mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;    { save the changes }
      mrNo:    CustTable.Cancel;  { abandon the changes}
      mrCancel: Abort;           { abort closing the dataset }
    end;
  end;
end;

```

디자인 타임에 데이터셋에 대한 *BeforeClose* 이벤트에 프로시저를 연결하려면 다음과 같이 합니다.

- 1 데이터 모듈 또는 폼의 테이블을 선택합니다.
- 2 Object Inspector의 Event 페이지를 클릭합니다.
- 3 *BeforeClose* 이벤트에 대한 프로시저 이름을 입력하거나 드롭다운 목록에서 선택합니다.

## 데이터셋 찾아보기

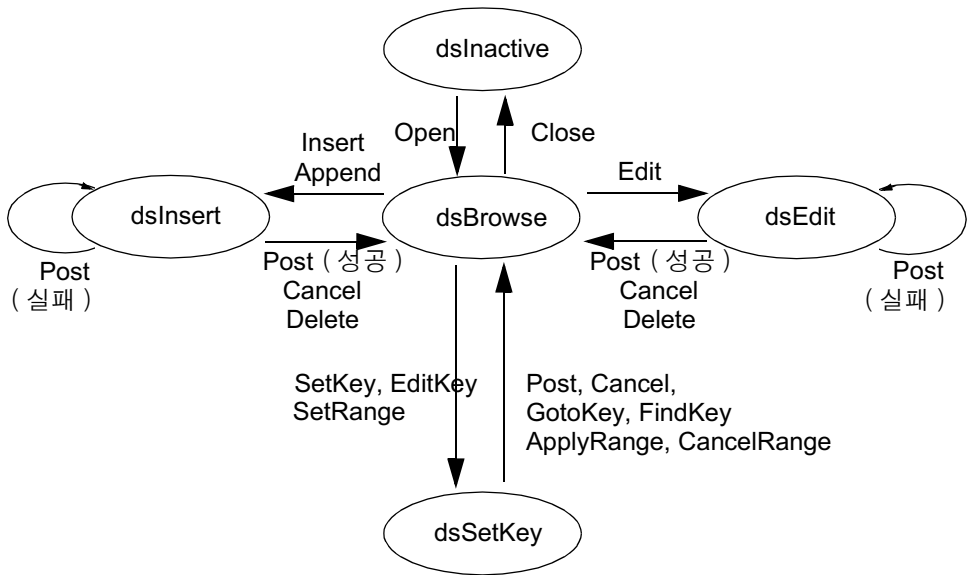
애플리케이션이 데이터셋을 열 때, 데이터셋은 자동으로 *dsBrowse* 상태가 됩니다. 찾아보기를 사용하면 데이터셋의 레코드를 볼 수 있지만 레코드를 편집하거나 새 레코드를 삽입할 수는 없습니다. *dsBrowse*는 주로 데이터셋의 레코드에서 레코드로 스크롤하기 위해 사용됩니다. 레코드에서 레코드로의 스크롤에 대한 자세한 내용은 16-9 페이지의 "데이터셋 탐색"을 참조하십시오.

*dsBrowse*에서는 데이터셋이 지원하는 모든 다른 데이터셋 상태를 설정할 수 있습니다. 예를 들어, 데이터셋에 대하여 *Insert* 또는 *Append* 메소드를 호출하면 상태가 *dsBrowse*에서 *dsInsert*로 변경됩니다(다른 요인과 *CanModify* 같은 데이터 속성이 이러한 변경을 막지 않는 경우). 레코드를 찾기 위해 *SetKey*를 호출하면 클라이언트 데이터셋은 *dsSetKey* 모드가 됩니다. 데이터셋에 레코드 삽입 및 추가에 대한 자세한 내용은 16-22 페이지의 "데이터 수정"을 참조하십시오.

모든 데이터셋에 연결된 두 개의 메소드는 데이터셋을 *dsBrowse* 상태로 반환할 수 있습니다. *Cancel*은 현재의 편집, 삽입 또는 검색 작업을 종료하고 항상 데이터셋을 *dsBrowse* 상태로 반환합니다. *Post*는 데이터셋에 변경 내용을 작성하려고 시도하고, 성공하면 데이터셋을 *dsBrowse* 상태로 반환합니다. *Post*가 실패하면 현재 상태는 변경되지 않습니다.

다음의 그림은 애플리케이션에서 설정할 수 있는 다른 데이터셋 모드와 이들 모드를 설정하는 메소드 모두에 대한 *dsBrowse*의 관계를 보여줍니다.

**그림 16.2** 찾아보기와 데이터셋 상태의 관계



## 데이터셋 편집 사용 가능

데이터셋은 애플리케이션이 레코드를 수정하기 전에 *dsEdit* 모드여야 합니다. 데이터셋의 읽기 전용 *CanModify* 속성이 *True*인 경우, 코드에서 *Edit* 메소드를 사용하여 데이터셋을 *dsEdit* 모드로 둡니다. 데이터셋이 단방향이나, 데이터셋을 원본으로 하는 데이터베이스가 쓰기 및 읽기 권한을 허용하지 않거나, 일부 다른 요인(클라이언트 데이터셋의 *ReadOnly* 속성 또는 데이터셋 프로바이더의 *poReadOnly* 옵션)이 없다면, *CanModify*는 *True*입니다.

다음의 모든 조건이 적용되는 경우, 애플리케이션의 폼에서 일부 data-aware 컨트롤은 데이터셋을 자동으로 *dsEdit* 상태로 만듭니다.

- 컨트롤의 *ReadOnly* 속성은 기본값인 *False*입니다
- 컨트롤에 대한 데이터 소스의 *AutoEdit* 속성은 *True*입니다
- *CanModify*는 데이터셋에 대해 *True*입니다.

**참고** 데이터셋이 *dsEdit* 상태인 경우라도 애플리케이션 사용자가 적절한 SQL 액세스 권한이 없으면 레코드 편집이 성공하지 못합니다.

*Cancel*, *Post* 또는 *Delete* 메소드를 호출하여 코드에서 *dsEdit* 상태에서 *dsBrowse* 상태로 데이터셋을 반환할 수 있습니다. *Cancel*은 현재 필드나 레코드의 편집 내용을 버립니다. *Post*는 수정된 레코드를 변경 로그에 작성하려고 시도하고, 성공하면 데이터셋을 *dsBrowse*로 반환합니다. *Post*가 변경 내용을 작성할 수 없으면, 데이터셋은 *dsEdit* 상태로 남습니다. *Delete*는 데이터셋의 현재 레코드를 삭제하려고 시도하고, 성공하면 데이터셋을 *dsBrowse* 상태로 반환합니다. *Delete*가 실패하면 데이터셋은 *dsEdit* 상태로 남습니다.

그리드의 다른 레코드로 이동하는 것처럼 현재 레코드를 변경하거나 또는 폼의 다른 컨트롤로 이동하는 것처럼 컨트롤이 포커스를 잃도록 하는 모든 작업을 사용자가 수행할 때, 편집이 가능한 data-aware 컨트롤은 자동으로 *Post*를 호출합니다.

데이터셋의 편집 필드 및 레코드에 대한 자세한 설명은 16-22 페이지의 "데이터 수정"을 참조하십시오.

## 새 레코드 삽입 사용 가능

애플리케이션이 새 레코드를 추가하기 전에 데이터셋은 *dsInsert* 모드여야 합니다. 데이터셋의 읽기 전용 *CanModify* 속성이 *True*인 경우 코드에서 *Insert* 또는 *Append* 메소드를 사용하여 데이터베이스를 *dsInsert* 모드로 만들 수 있습니다. 데이터셋이 단방향이지 아니거나, 데이터셋을 원본으로 하는 데이터베이스가 쓰기 및 읽기 권한을 허용하지 않거나, 일부 다른 요인(클라이언트 데이터셋의 *ReadOnly* 속성 또는 데이터셋 프로바이더의 *poReadOnly* 옵션)이 없다면, *CanModify*는 *True*입니다.

애플리케이션의 폼에서 data-aware 그리드와 탐색기 컨트롤은 다음의 모든 조건이 적용되는 경우 데이터베이스를 *dsInsert* 상태로 둘 수 있습니다.

- 컨트롤의 *ReadOnly* 속성은 기본값인 *False*입니다
- 컨트롤에 대한 데이터 소스의 *AutoEdit* 속성은 *True*입니다
- *CanModify*는 데이터셋에 대해 *True*입니다.

**참고** 데이터셋이 *dsInsert* 상태인 경우에도 애플리케이션 사용자가 적절한 SQL 권한이 없으면, 레코드 삽입이 성공하지 못합니다.

*Cancel*, *Post* 또는 *Delete* 메소드를 호출하여 코드에서 데이터셋을 *dsInsert* 상태에서 *dsBrowse* 상태로 반환할 수 있습니다. *Delete*와 *Cancel*은 새 레코드를 버립니다. *Post*는 새 레코드를 변경 로그에 작성하려고 시도하고, 성공하면 데이터셋을 *dsBrowse*로 반환합니다. *Post*가 레코드를 작성하지 못하면, 데이터셋은 *dsInsert* 상태로 남습니다.

그리드의 다른 레코드로의 이동처럼 현재 레코드를 변경하는 작업을 사용자가 실행할 때, 삽입 가능한 data-aware 컨트롤은 자동으로 *Post*를 호출합니다.

데이터셋에 레코드 삽입 및 추가에 대한 자세한 내용은 16-22 페이지의 "데이터 수정"을 참조하십시오.

## 인덱스 기반 작업 사용 가능

---

데이터셋을 *dsSetKey* 상태로 두는 *TDataSet*에는 정의된 메소드가 없습니다. 이 상태는 인덱스 기반의 검색이나 지정된 범위로 레코드를 제한하는 것 같은 인덱스 기반 작업에 대한 정보를 설정할 때 사용하는 클라이언트 데이터셋을 위해 존재합니다. 런타임 시 *SetKey* 또는 *EditKey* 메소드를 사용하여 클라이언트 데이터셋을 *dsSetKey* 모드로 둘 수 있습니다.

데이터셋이 *dsSetKey* 상태일 때, 필드에 값을 할당하면 현재 레코드의 필드 값이 아닌 인덱스 기반의 작업에 대한 조건을 포함하는 내부 버퍼가 변경됩니다.

인덱스 기반의 작업을 수행하거나 *Post* 메소드를 호출하기 전까지 데이터셋은 *dsSetKey* 상태로 남습니다.

클라이언트 데이터셋의 인덱스 기반 작업에 대한 자세한 내용은 20-2 페이지의 "클라이언트 데이터셋에서 데이터 탐색" 및 20-5 페이지의 "레코드 표시 제한"을 참조하십시오.

## 필드 계산

---

데이터셋은 *OnCalcFields* 이벤트가 발생할 때마다 *dsCalcFields* 또는 *dsInternalCalc* 모드가 됩니다. 이들 상태에서는 핸들러가 수정하도록 고안된 계산된 필드를 제외하고는 데이터셋의 레코드에 수정이나 추가가 방지됩니다. 모든 기타 수정 사항이 방지되는 이유는 *OnCalcFields*가 다른 필드의 값을 사용하여 계산된 필드에서 값을 파생시키기 때문입니다. 다른 필드에 변경이 있으면 계산된 필드에 할당된 값을 무효화합니다.

*OnCalcFields* 핸들러가 종료되면 데이터셋은 *dsBrowse* 상태로 반환됩니다.

계산된 필드 생성과 *OnCalcFields* 이벤트 핸들러에 대한 자세한 내용은 16-27 페이지의 "OnCalcFields 사용"을 참조하십시오.

## 레코드 필터링

---

데이터셋이 단방향인 경우, 데이터셋은 애플리케이션이 데이터셋의 *OnFilterRecord* 이벤트 핸들러를 호출할 때마다 *dsFilter* 모드가 됩니다. (단방향 데이터셋은 필터를 지원하지 않습니다). 이 상태에서는 필터링 프로세스 동안 데이터셋의 레코드에 대한 수정 또는 추가가 방지되어 필터 요청이 무효화되지 않도록 합니다.

*OnFilterRecord* 핸들러가 종료되면 데이터셋은 *dsBrowse* 상태로 반환됩니다.

필터링에 대한 자세한 내용은 16-17 페이지의 "필터를 사용한 데이터 서브셋 표시 및 편집"을 참조하십시오.

## 업데이트 적용

---

변경 로그의 업데이트를 데이터베이스 테이블에 다시 적용할 때, 클라이언트 데이터셋은 일시적으로 *dsNewValue*, *dsOldValue* 또는 *dsCurValue* 상태가 될 수 있습니다. 이러한 상태는 필드 컴포넌트 (*NewValue*, *OldValue* 및 *CurValue*)의 해당 속성이 보

통 *OnReconcileError* 이벤트 핸들러에서 액세스되고 있다는 것을 나타냅니다. 애플리케이션은 이들 상태를 보거나 설정할 수 없습니다.

## 데이터셋 탐색

각각의 활성 데이터셋은 데이터셋의 현재 행에 커서나 포인터가 있습니다. 데이터셋의 현재 행은 필드 값이 *TDBEdit*, *TDBLabel* 및 *TDBMemo*와 같이 폼의 단일 필드의 data-aware 컨트롤에 현재 나타나는 행입니다. 데이터셋이 편집을 지원하면, 현재 레코드는 편집, 삽입 및 삭제 메소드로 처리될 수 있는 값을 포함합니다.

커서를 이동시켜 다른 행을 가리키면 현재 행을 변경할 수 있습니다. 다음 표는 다른 레코드로 이동하기 위해 애플리케이션 코드에서 사용할 수 있는 메소드를 나열한 것입니다.

표 16.2 데이터셋의 탐색 메소드

| 메소드           | 커서 이동                                                           |
|---------------|-----------------------------------------------------------------|
| <i>First</i>  | 데이터셋의 첫 번째 행(모든 데이터셋)                                           |
| <i>Last</i>   | 데이터셋의 마지막 행(단방향 데이터셋에는 사용할 수 없음)                                |
| <i>Next</i>   | 데이터셋의 다음 행(모든 데이터셋)                                             |
| <i>Prior</i>  | 데이터셋의 이전 행(단방향 데이터셋에는 사용할 수 없음)                                 |
| <i>MoveBy</i> | 데이터셋의 앞 또는 뒤쪽 행의 지정된 수(뒤로 이동하려고 시도하는 경우, 단방향 데이터셋은 예외를 발생시킵니다.) |

비주얼 data-ware 컴포넌트 *TDBNavigator*는 사용자가 런타임 시 레코드 사이로 이동할 때 클릭하는 단추로 이러한 메소드를 캡슐화합니다. 탐색기 컴포넌트에 대한 자세한 내용은 15-26 페이지의 "레코드 탐색 및 처리"를 참조하십시오.

이러한 메소드 외에, *TDataSet*은 데이터셋의 레코드를 반복할 때 유용한 정보를 제공하는 두 개의 부울 속성을 정의합니다.

표 16.3 데이터셋의 탐색 속성

| 속성                             | 설명                                                                              |
|--------------------------------|---------------------------------------------------------------------------------|
| <i>Bof</i> (Beginning-of-file) | <i>True</i> : 커서가 데이터셋의 첫 번째 행에 있습니다.<br><i>False</i> : 커서가 데이터셋의 첫 번째 행에 없습니다. |
| <i>Eof</i> (End-of-file)       | <i>True</i> : 커서가 데이터셋의 마지막 행에 있습니다.<br><i>False</i> : 커서가 데이터셋의 마지막 행에 없습니다.   |

## First 및 Last 메소드 사용

*First* 메소드는 커서를 데이터셋의 첫 번째 행으로 이동하고, *Bof* 속성을 *True*로 설정합니다. 커서가 이미 데이터셋의 첫 번째 행에 있으면 *First* 메소드는 아무것도 하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 첫 번째 레코드로 이동합니다.

```
CustTable.First;
```

*Last* 메소드는 커서를 데이터셋의 마지막 행으로 이동하고, *Eof* 속성을 *True*로 설정합니다. 커서가 이미 데이터셋의 마지막 행에 있으면 *Last* 메소드는 아무것도 하지 않습니다.

다음 코드는 *CustTable*의 마지막 레코드로 이동합니다.

```
CustTable.Last;
```

**참고** *Last* 메소드는 단방향 데이터셋에서 예외를 발생시킵니다.

**팁** 사용자의 조작 없이 데이터셋의 첫 번째 또는 마지막 행으로 이동하도록 프로그램으로 만들 수도 있지만, 사용자가 *TDBNavigator* 컴포넌트를 사용하여 레코드에서 레코드로 탐색하도록 할 수도 있습니다. 활성이고 보이는(visible) 탐색기 컴포넌트는 사용자가 활성 데이터셋의 첫 번째 및 마지막 행으로 이동할 수 있는 버튼을 포함합니다. 이러한 버튼에 대한 *OnClick* 이벤트는 탐색기가 데이터셋의 *First* 및 *Last* 메소드를 호출한 후 즉시 발생합니다. 탐색기 컴포넌트의 효과적인 사용에 대한 자세한 내용은 15-26 페이지의 "레코드 탐색 및 처리"를 참조하십시오.

## Next 및 Prior 메소드 사용

*Next* 메소드는 데이터셋에서 한 행 앞으로 커서를 이동하고, 데이터셋이 비어 있지 않으면 *Bof* 속성을 *False*로 설정합니다. *Next*를 호출할 때 이미 커서가 데이터셋의 마지막 행에 있으면 아무것도 발생하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 다음 레코드로 이동합니다.

```
CustTable.Next;
```

*Prior* 메소드는 데이터셋의 한 행 뒤로 커서를 이동하고, 데이터셋이 비어 있지 않으면 *Eof*를 *False*로 설정합니다. *Prior*를 호출할 때 커서가 이미 데이터셋의 첫 번째 행에 있으면 *Prior*는 아무것도 하지 않습니다.

예를 들어, 다음 코드는 *CustTable*의 이전 레코드로 이동합니다.

```
CustTable.Prior;
```

**참고** *Prior* 메소드는 단방향 데이터셋에서 예외를 발생시킵니다.

## MoveBy 메소드 사용

*MoveBy*를 사용하면 데이터셋에서 커서를 앞이나 뒤의 행으로 얼마나 많이 이동할 지를 지정할 수 있습니다. *MoveBy*가 호출되면 현재 레코드에 상대적으로 이동합니다. *MoveBy*는 또한 필요에 따라 적절히 데이터셋의 *Bof* 및 *Eof* 속성을 설정합니다.

이 함수는 이동할 레코드의 수인 정수 매개변수를 사용합니다. 양의 정수는 앞으로 이동을 나타내고, 음의 정수는 뒤로 이동을 나타냅니다.

**참고** *MoveBy*는 음의 인수를 사용할 때 단방향 데이터셋에서 예외를 발생시킵니다.

*MoveBy*는 이동하는 행의 수를 반환합니다. 데이터셋의 시작 또는 끝 부분을 지나서 이동하려는 경우, *MoveBy*에서 반환한 행의 수는 이동을 요청한 행의 수와 다릅니다. *MoveBy*가 데이터셋의 첫 번째 레코드나 마지막 레코드에 도달하면 멈추기 때문입니다.

다음 코드는 *CustTable* 두 레코드 뒤로 커서를 이동합니다.

```
CustTable.MoveBy(-2);
```

## Eof 및 Bof 속성 사용

두 개의 읽기 전용 런타임 속성인 *Eof* (End-of-file) 및 *Bof* (Beginning-of-file)는 특히 데이터셋의 모든 레코드를 반복하려는 경우와 같은 데이터셋 탐색을 제어하는 데 유용합니다.

### Eof

*Eof*가 *True*인 경우 커서가 데이터셋의 마지막 행에 있다는 것을 나타냅니다. 애플리케이션이 다음의 작업을 수행할 때 *Eof*는 *True*로 설정됩니다.

- 빈 데이터셋을 엽니다.
- 데이터셋의 *Last* 메소드를 성공적으로 호출합니다.
- 데이터셋의 *Next* 메소드를 호출했지만 커서가 현재 데이터셋의 마지막 행에 있기 때문에 메소드는 실패합니다.
- 빈 범위 또는 데이터셋에서 *SetRange*를 호출합니다.(클라이언트 데이터셋만 해당)

다른 모든 경우에는 *Eof*가 *False*로 설정됩니다. 위의 조건 중 하나라도 만족하지 않고, 속성을 직접 테스트하지 않은 경우에는 *Eof*를 *False*로 생각해야 합니다.

*Eof*는 일반적으로 순환문의 조건에서 테스트되어 데이터셋의 모든 레코드의 반복 처리를 제어합니다. 레코드가 포함된 데이터셋을 열면 (또는 *First*를 호출하면), *Eof*는 *False*입니다. 데이터셋에서 한 번에 한 레코드를 반복하려면, *Eof*가 *True*이면 종료되는 순환문을 작성합니다. 순환문 내에서 데이터셋의 각 레코드에 대해 *Next*를 호출합니다. *Eof*는 커서가 이미 마지막 레코드에 있을 때 사용자가 *Next*를 호출하기 전까지는 *False* 상태로 남습니다.

다음 코드는 *CustTable*이라는 데이터셋에 대한 레코드 처리 순환문을 코딩할 수 있는 방법 중 한 가지를 보여줍니다.

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ...
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

**팁** 이 예제는 또한 데이터셋에 연결된 data-aware 비주얼 컨트롤을 사용 가능 및 사용 불가능하게 하는 방법을 보여줍니다. 데이터셋 반복 동안 비주얼 컨트롤을 사용 불가능하게 하면, 현재 레코드가 변경될 때 애플리케이션이 컨트롤의 내용을 업데이트할 필요가 없기 때문에 처리 속도가 빨라집니다. 반복이 완료된 후, 컨트롤을 다시 사용 가능하게 해야 새로운 현재 행의 값으로 업데이트할 수 있습니다. 비주얼 컨트롤을 사용 가능하게 하는 것은 **try...finally** 문의 **finally** 절에서 이루어집니다. 이는 예외로 인해 순환문 처리가 너무 빨리 종료된 경우라도 컨트롤이 사용 불가능한 상태로 남지 않도록 보장합니다.

## Bof

*Bof*가 *True*인 경우는 커서가 명백히 데이터셋의 첫 번째 행에 있다는 것을 나타냅니다. *Bof*는 애플리케이션이 다음과 같은 작업을 수행할 때에는 *True*로 설정됩니다.

- 데이터셋을 엽니다.
- 데이터셋의 *First* 메소드를 호출합니다.
- 데이터셋의 *Prior* 메소드를 호출했지만 커서가 현재 데이터셋의 첫 번째 행에 있기 때문에 메소드가 실패합니다. (이 조건은 단방향 데이터셋에는 적용되지 않습니다.)
- 빈 범위 또는 데이터셋에서 *SetRange*를 호출합니다. (클라이언트 데이터셋만 해당)

*Bof*는 다른 모든 경우에 *False*로 설정됩니다. 위의 조건 중 하나라도 만족하지 않고 속성을 직접 테스트하지 않는 경우에는 *Bof*는 *False*라고 생각해야 합니다.

*Eof*와 같이, *Bof*는 데이터셋의 반복적인 레코드 처리를 제어하기 위해 순환문 조건에 있을 수 있습니다. 다음 코드는 *CustTable*이라는 데이터셋에 대한 레코드 처리 순환문을 코딩할 수 있는 방법 중 한 가지를 보여줍니다.



```

CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.BOF do { Cycle until BOF is True }
  begin
    { Process each record here }
    :
    CustTable.Prior; { BOF False on success; BOF True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;

```

## 레코드 표시 및 반환

데이터셋의 레코드에서 레코드로 이동하는 것(또는 특정 레코드 수만큼 한 레코드에서 다른 레코드로 이동) 외에도, 데이터셋의 특정 위치를 표시하여 원할 때마다 표시된 위치로 빨리 돌아 올 수 있습니다. *TDataSet*은 레코드를 표시하고 나중에 그 레코드로 반환되도록 하는 북마크 기능이 있습니다. 북마크 기능은 *Bookmark* 속성과 다섯 개의 북마크 메소드로 구성됩니다.

*Bookmark* 속성은 애플리케이션의 여러 북마크 중에서 어떤 북마크가 현재 북마크인지를 나타냅니다. *Bookmark*는 현재 북마크를 식별하는 문자열입니다. 다른 북마크를 추가할 때마다 그 북마크가 현재 북마크가 됩니다.

*TDataSet*은 **virtual** 북마크 메소드를 구현합니다. 이 메소드는 북마크 메소드가 호출되는 경우, *TDataSet*에서 파생된 데이터셋 객체가 값을 반환하도록 보장하지만, 반환 값은 현재 위치를 추적하지 않는 단순한 기본값입니다. 단방향 데이터셋은 북마크 지원을 추가하지 않습니다. 그러나 *TCustomClientDataSet*은 다음 목록의 설명과 같이 의미있는 값을 반환하도록 북마크 메소드를 재구현합니다.

- *BookmarkValid*는 지정된 북마크가 사용 중인지를 결정합니다.
- *CompareBookmarks*는 두 개의 북마크가 동일한지를 테스트합니다.
- *GetBookmark*는 데이터셋의 현재 위치에 대한 북마크를 할당합니다.
- *GotoBookmark*는 *GetBookmark*에 의해 이전에 만든 북마크로 돌아갑니다.
- *FreeBookmark*는 *GetBookmark*에 의해 이전에 할당된 북마크를 해제합니다.

북마크를 만들려면, 애플리케이션에서 *TBookmark* 타입의 변수를 선언해야 합니다. *TBookmark* 타입은 포인터입니다. *GetBookmark*를 호출하면 데이터셋이 북마크에 대한 저장소를 할당하고 변수가 이 저장소를 가리키도록 설정합니다. 북마크에는 데이터셋의 특정 위치를 식별하는 정보가 포함됩니다.

*GotoBookmark*를 호출하여 특정 레코드로 이동하기 전에, *BookmarkValid*를 호출하여 북마크가 레코드를 가리키는지 여부를 알 수 있습니다. 지정된 북마크가 레코드를 가리키면, *BookmarkValid*는 *True*를 반환합니다. *TDataSet*에서 *BookmarkValid*는 항상 *False*를 반환하는 가상 메소드로서 북마크가 유효하지 않음을 나타냅니다. 북마크를 지원하는 자손이 이 메소드를 재구현하여 의미있는 반환값을 제공합니다.

*CompareBookmarks*를 호출하여 이동하려는 북마크가 다른 (또는 현재) 북마크와 다른지 확인할 수 있습니다. *TDataSet.CompareBookmarks*는 항상 0을 반환하는데 이는 두 개의 북마크가 동일하다는 것을 나타냅니다. 북마크를 지원하는 자손이 이 메소드를 재구현하여 의미있는 반환값을 제공합니다.

북마크를 전달할 때, *GotoBookmark*는 데이터셋의 커서를 북마크에서 지정된 위치로 이동합니다. *TDataSet*은 *GotoBookmark*를 호출하면 오류를 발생시킵니다. 단방향 데이터셋은 아무것도 하지 않습니다. 북마크를 지원하는 자손이 이 메소드를 재구현하여 의미있는 반환값을 제공합니다.

*FreeBookmark*는 지정된 북마크가 더 이상 필요하지 않을 때 지정된 북마크에 할당된 메모리를 해제합니다. 기존 북마크를 재사용하기 전에 *FreeBookmark*를 호출해야 합니다.

다음 코드는 북마크 사용 예를 하나 보여줍니다.

```
procedure DoSomething (const MyDS:TSQClientDataSet)
var
  Bookmark: TBookmark;
begin
  Bookmark := MyDS.GetBookmark; { Allocate memory and assign a value }
  MyDS.DisableControls; { Turn off display of records in data controls }
  try
    MyDS.First; { Go to first record in table }
    while not MyDS.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      ...
      MyDS.Next;
    end;
  finally
    MyDS.GotoBookmark(Bookmark);
    MyDS.EnableControls; { Turn on display of records in data controls, if necessary }
    MyDS.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
```

레코드를 반복하기 전에 컨트롤은 사용 불가능합니다. 레코드 반복 도중에 오류가 발생하는 경우, **finally** 절을 사용하여 컨트롤이 항상 사용 가능하고 순환문이 너무 빨리 종료되더라도 북마크가 항상 해제되어 있는지 보장할 수 있습니다.

## 데이터셋 검색

데이터셋이 단방향이 아닌 경우에는 *TDataSet*의 *Locate* 및 *Lookup* 메소드를 사용하여 데이터셋을 검색할 수 있습니다. 이러한 메소드를 사용하면 모든 데이터셋의 모든 열 타입을 검색할 수 있습니다.

**참고** 클라이언트 데이터셋에는 인덱스 기반의 레코드를 검색할 수 있도록 하는 추가 메소드 패밀리가 있습니다. 인덱스 기반의 클라이언트 데이터셋 검색에 대한 내용은 20-2 페이지의 "클라이언트 데이터셋에서 데이터 탐색"을 참조하십시오.

## Locate 사용

*Locate*는 지정된 검색 조건과 일치하는 첫 번째 행으로 커서를 이동합니다. 가장 간단한 폼에서, 검색할 열 이름, 일치하는 필드 값 및 검색이 대소문자 구별인지 또는 부분 키 일치를 사용할 수 있는지를 지정하는 옵션 플래그를 *Locate*에 전달합니다. (부분 키 일치하는 기준 문자열이 필드 값의 접두사일 필요가 있을 경우입니다.) 예를 들어, 다음 코드는 *Company* 열의 값이 "Professional Divers, Ltd."인 *CustTable*의 첫 번째 행으로 커서를 이동시킵니다.

```
var
    LocateSuccess:Boolean;
    SearchOptions: TLocateOptions;
begin
    SearchOptions := [loPartialKey];
    LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
        SearchOptions);
end;
```

*Locate*가 일치하는 값을 찾으면 이 일치 값을 포함하는 첫 번째 레코드는 현재 레코드가 됩니다. *Locate*는 일치하는 레코드를 찾으면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다. 검색이 실패하면 현재 레코드는 변경되지 않습니다.

*Locate*의 장점은 여러 열을 검색하고 검색할 여러 값을 지정하려는 경우에 나타납니다. 검색 값은 가변 타입인데 이는 검색 조건에 각각 다른 데이터 타입을 지정할 수 있다는 것을 의미합니다. 검색 문자열에서 여러 열을 지정하려면, 문자열의 개별 항목을 세미콜론으로 구분합니다.

검색값이 가변 타입이기 때문에, 값을 여러 개 전달하는 경우, 인수로 가변 배열 타입을 넘겨 주거나(예: *Lookup* 메소드로부터의 반환값), 아니면 *VarArrayOf* 함수를 사용하여 가변 배열을 구성해야 합니다. 다음 코드는 여러 검색값과 부분 키 일치를 사용하는 여러 열에서의 검색을 보여줍니다.

```
with CustTable do
    Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver', 'P']), loPartialKey);
```

*Locate*는 가능한 가장 빠른 메소드를 사용하여 일치하는 레코드를 찾습니다. 검색하려는 열이 인덱싱 되어 있고 이 인덱스가 사용자가 지정하는 검색 옵션과 호환되는 경우, *Locate*는 이 인덱스를 사용합니다.

## Lookup 사용

*Lookup*은 지정된 검색 조건과 일치하는 첫 번째 행을 검색합니다. 일치하는 행을 찾으면 모든 계산된 필드와 데이터셋에 연결된 조희 필드를 다시 계산한 다음 일치하는 행에서 하나 이상의 필드를 반환합니다. *Lookup*은 커서를 일치하는 행으로 이동하지 않고 행의 값만 반환합니다.

가장 간단한 폼에서, 검색하는 필드 이름, 일치하는 필드 값 및 반환하는 필드를 *Lookup*에 전달합니다. 예를 들어, 다음 코드는 *Company* 필드의 값이 "Professional Divers, Ltd."인 *CustTable*의 첫 번째 레코드를 찾은 다음, 회사 이름, 담당자 및 전화 번호를 반환합니다.

```
var
    LookupResults:Variant;
begin
    with CustTable do
        LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company;
            Contact; Phone');
    end;
```

*Lookup*은 첫 번째 찾아낸 레코드에서 지정된 필드에 대한 값을 반환합니다. 값은 가변 타입으로 반환됩니다. 둘 이상의 반환값이 요청되면, *Lookup*은 가변 배열을 반환합니다. 일치하는 레코드가 없으면, *Lookup*은 Null 가변을 반환합니다. 가변 배열에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

*Lookup*의 장점은 여러 열을 검색하고 검색할 여러 값을 지정하려는 경우에 나타납니다. 여러 열이나 결과 필드를 포함하는 문자열을 지정하려면, 문자열의 개별 필드를 세미콜론으로 구분합니다.

검색값이 가변 타입이기 때문에, 값을 여러 개 넘겨주는 경우, 인수로 가변 배열 타입을 전달하거나(예: *Lookup* 메소드로부터의 반환값), 아니면 *VarArrayOf* 함수를 사용하여 가변 배열을 구성해야 합니다. 다음 코드는 여러 열에서의 조희 검색을 설명합니다.

```
var
    LookupResults:Variant;
begin
    with CustTable do
        LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver',
            'Christiansted']),'Company; Addr1; Addr2; State; Zip');
    end;
```

*Lookup*은 가능한 가장 빠른 메소드를 사용하여 일치하는 레코드를 찾습니다. 검색할 열이 인덱싱 되어 있는 경우, *Lookup*은 이 인덱스를 사용합니다.

## 필터를 사용한 데이터 서브셋 표시 및 편집

애플리케이션은 데이터셋 내의 레코드 서브셋에만 주로 관심이 있습니다. 예를 들어, 고객 데이터베이스에서 캘리포니아에 소재한 회사에 대한 레코드만 검색, 보기 또는 특정 필드 값의 집합을 포함하는 레코드를 찾을 수도 있습니다.

단방향 데이터셋에서는 레코드를 폐치하는 SQL 명령을 지정할 때 데이터셋의 레코드를 제한만 할 수 있습니다. 그러나 다른 *TDataSet* 자손의 경우 이미 폐치된 데이터의 서브셋을 정의할 수 있습니다. 애플리케이션의 데이터셋 모든 레코드의 서브셋에 대한 액세스를 제한하기 위해 필터를 사용할 수 있습니다.

필터는 레코드를 표시하기 위해 만족해야 하는 조건을 지정합니다. 필터 조건은 데이터셋의 *Filter* 속성으로 규정되거나 *OnFilterRecord* 이벤트 핸들러로 코드화될 수 있습니다. 필터 조건은 필드가 인덱싱 되어 있는지 아닌지 간에 데이터셋에서 지정된 수의 필드 값을 기준으로 합니다. 예를 들어, 캘리포니아에 소재한 회사의 레코드를 보기 위해 간단한 필터는 레코드에 State 필드의 값이 "CA"인 필드를 필터링할 것입니다.

**참고** 필터는 데이터셋에서 검색된 모든 레코드에 적용됩니다. 많은 양의 데이터를 필터하려는 경우, 데이터를 폐치하는 SQL 명령을 사용하여 레코드 검색을 제한하거나, 필터를 사용하기 보다는 인덱싱된 클라이언트 데이터셋에 범위를 설정하는 것이 보다 효과적일 수 있습니다.

## 필터링 사용 가능 및 사용 불가능

데이터셋의 필터를 사용 가능하도록 하려면 다음 세 단계 과정을 거쳐야 합니다.

- 1 필터를 만듭니다.
- 2 필요한 경우 문자열 기반의 필터 테스트를 위한 필터 옵션을 설정합니다.
- 3 *Filtered* 속성을 *True*로 설정합니다.

필터링이 사용 가능한 경우, 필터 조건을 만족하는 레코드만 애플리케이션에서 사용할 수 있습니다. 필터링은 항상 임시 조건입니다. *Filtered* 속성을 *False*로 설정하여 필터링을 해제할 수 있습니다.

## 필터 생성

데이터셋에 대한 필터를 만드는 방법은 다음 두 가지입니다.

- *Filter* 속성에서 단순한 필터 조건을 지정합니다. *Filter*는 특히 런 타임 시 필터를 만들고 사용하는 데 유용합니다.
- 단순하거나 복잡한 필터 조건을 위해 *OnFilterRecord* 이벤트 핸들러를 작성합니다. *OnFilterRecord*를 사용하여 디자인 타임에 필터 조건을 지정합니다. 필터 로직을 포함하는 단일 문자열로 제한된 *Filter* 속성과는 달리, *OnFilterRecord* 이벤트는 분기 및 순환문 로직을 사용하여 복잡하고 다양한 수준의 필터 조건을 만들 수 있습니다.

*Filter* 속성을 사용하여 필터를 만들면 애플리케이션에서 필터를 동적으로 생성, 변경 및 적용할 수 있다는 장점이 있습니다(예를 들어, 사용자 입력에 대한 응답으로). 그러나 필터 조건이 단일 텍스트 문자열에 표현될 수 있어야 하고, 분기 및 순환문 구조를 사용할 수 없으며, 데이터셋에 아직 없는 값에 대해 그 값을 비교하거나 테스트할 수 없다는 단점이 있습니다.

*OnFilterRecord* 이벤트는 필터가 복잡하고 다양할 수 있으며, 분기 및 순환문 구조를 사용하는 여러 줄의 코드에 적용할 수 있으며, 편집 상자의 텍스트와 같은 데이터셋 외부의 값에 대해 데이터셋 값을 테스트할 수 있다는 장점이 있습니다. *OnFilterRecord*를 사용하면 필터를 디자인 타임에 설정하고, 사용자 입력의 응답으로 수정할 수 없다는 단점이 있습니다. 그러나, 몇몇 필터 핸들러를 작성하여 일반적인 애플리케이션 조건에 대한 응답 시 핸들러를 전환할 수 있습니다.

다음 섹션은 *Filter* 속성과 *OnFilterRecord* 이벤트 핸들러를 사용하여 필터를 만드는 방법을 설명합니다.

### Filter 속성 설정

*Filter* 속성을 사용하여 필터를 작성하려면, 속성 값을 필터 조건을 포함하는 문자열로 설정합니다. 문자열에는 필터의 테스트 조건이 포함됩니다. 예를 들어, 다음 문장은 캘리포니아 주에 대한 값이 포함되는지를 알기 위해서 데이터셋의 *State* 필드를 테스트하는 필터를 작성합니다.

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

또한 컨트롤에 입력된 텍스트에 따라 *Filter*에 대한 값을 제공할 수도 있습니다. 예를 들어, 다음 문장은 편집 상자의 텍스트를 *Filter*에 할당합니다.

```
Dataset1.Filter := Edit1.Text;
```

물론, 하드 코딩된 텍스트와 사용자가 컨트롤에 입력한 데이터로 문자열을 생성할 수 있습니다.

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

필터에 명시적으로 포함되지 않는 경우 빈 레코드는 나타나지 않습니다.

```
Dataset1.Filter := 'State <> 'CA' or State = BLANK';
```

데이터셋에 필터를 적용하기 위해 *Filter*에 대한 값을 지정한 다음에는 *Filtered* 속성을 *True*로 설정합니다.

또한 필드 값을 리터럴 및 상수와 비교하거나 다음 연산자를 사용하여 값을 계산할 수 있습니다.

**표 16.4** 필터에 사용할 수 있는 연산자

| 연산자 | 의미     |
|-----|--------|
| <   | 작다     |
| >   | 크다     |
| >=  | 크거나 같다 |
| <=  | 작거나 같다 |
| =   | 같다     |

표 16.4 필터에 사용할 수 있는 연산자 ( 계속 )

| 연산자         | 의미                                    |
|-------------|---------------------------------------|
| <>          | 같지 않다                                 |
| AND         | 두 문장이 모두 <i>True</i> 인지 테스트           |
| NOT         | 뒤에 있는 문장이 <i>True</i> 가 아닌지 테스트       |
| OR          | 두 문장 중 적어도 하나가 <i>True</i> 인지 테스트     |
| IS NULL     | 필드 값이 Null인지 테스트                      |
| IS NOT NULL | 필드 값이 Null이 아닌지 테스트                   |
| +           | 숫자 더하기, 문자열을 연결, 날짜/시간 값에 숫자를 더하기     |
| -           | 숫자 빼기, 날짜 빼기 또는 날짜에서 숫자를 빼기           |
| *           | 두 수를 곱하기                              |
| /           | 두 수를 나누기                              |
| Upper       | 문자열을 대문자로 바꾸기                         |
| Lower       | 문자열을 소문자로 바꾸기                         |
| Substring   | 지정된 위치에서 시작하는 부분 문자열을 반환합니다.          |
| Trim        | 공백이나 지정된 문자열의 앞 또는 뒤를 잘라냅니다.          |
| TrimLeft    | 공백이나 지정된 문자열 앞을 잘라냅니다.                |
| TrimRight   | 공백이나 지정된 문자열 뒤를 잘라냅니다.                |
| Year        | 날짜/시간 값에서 연도를 반환합니다.                  |
| Month       | 날짜/시간 값에서 월을 반환합니다.                   |
| Day         | 날짜/시간 값에서 일을 반환합니다.                   |
| Hour        | 시간 값에서 시간을 반환합니다.                     |
| Minute      | 시간 값에서 분을 반환합니다.                      |
| Second      | 시간 값에서 초를 반환합니다.                      |
| GetDate     | 현재 날짜를 반환합니다.                         |
| Date        | 날짜/시간 값에서 날짜 부분을 반환합니다.               |
| Time        | 날짜/시간 값에서 시간 부분을 반환합니다.               |
| Like        | 문자열 비교에서 패턴 일치(pattern match)를 제공합니다. |
| In          | 집합의 포함 관계를 테스트합니다.                    |
| *           | 부분 비교를 위한 대표 문자(와일드카드)입니다.            |

이들 연산자를 조합하여 매우 복잡한 필터를 만들 수 있습니다. 예를 들어, 다음 문장은 레코드를 표시하기 전에 두 테스트 조건을 만족하는지 확인합니다.

(Custno > 1400) AND (Custno < 1500);

**참고** 필터링을 사용하는 경우, 사용자가 레코드를 편집하면 레코드는 더 이상 필터의 테스트 조건을 만족하지 않을 수도 있습니다. 따라서 데이터셋에서 레코드를 검색한 다음에는 레코드가 "사라질" 수도 있습니다. 레코드가 사라지면, 필터 조건을 만족하는 다음 레코드가 현재 레코드가 됩니다.

## OnFilterRecord 이벤트 핸들러 작성

데이터셋의 필터는 검색하는 각각의 레코드에 대해 데이터셋에서 생성된 *OnFilterRecord* 이벤트에 응답하는 이벤트 핸들러입니다. 모든 필터 핸들러의 핵심은 레코드를 테스트하여 애플리케이션에 레코드를 표시할지 여부를 결정하는 것입니다.

레코드가 필터 조건을 만족하는지 여부를 나타내려면, 필터 핸들러에서 *Accept* 매개변수를 *True*로 설정하여 레코드를 포함하거나, *False*로 설정하여 제외시켜야 합니다. 예를 들어, 다음 필터는 *State* 필드가 "CA"로 설정된 레코드만 표시합니다.

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept:Boolean);
begin
    Accept := DataSet['State'].AsString = 'CA';
end;
```

필터링이 사용 가능하면, *OnFilterRecord* 이벤트가 검색된 각 레코드에 대해 생성됩니다. 이벤트 핸들러는 각각의 레코드를 테스트하고, 필터 조건을 만족하는 레코드만 표시됩니다. *OnFilterRecord* 이벤트는 데이터셋의 모든 레코드에 대해 생성되기 때문에, 가능한 한 빈틈없이 코드를 작성하여 이벤트 핸들러를 유지 관리하므로 애플리케이션의 성능에 영향을 주지 않도록 합니다.

## 런타임 시 필터 이벤트 핸들러 전환

필요한 만큼의 필터 이벤트 핸들러를 코드에 작성하여 런타임 시 이벤트를 전환할 수 있습니다. 런타임 시 다른 필터 이벤트 핸들러로 전환하려면, 데이터셋의 *OnFilterRecord* 속성에 새 이벤트 핸들러를 할당합니다.

예를 들어, 다음 문장은 *NewYorkFilter*라고 하는 *OnFilterRecord* 이벤트 핸들러로 전환합니다.

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

## 필터 옵션 설정

*FilterOptions* 속성을 사용하면 문자열 기반의 필드를 비교하는 필터가 레코드를 부분 비교하는지, 대소문자를 구별하는 비교인지를 지정할 수 있습니다. *FilterOptions*은 빈 설정(기본값)이거나 다음 값을 포함할 수 있는 설정 속성입니다.

**표 16.5** FilterOptions 값

| 값                         | 의미                                                    |
|---------------------------|-------------------------------------------------------|
| <i>foCaseInsensitive</i>  | 문자열 비교 시 대소문자를 무시합니다.                                 |
| <i>foNoPartialCompare</i> | 부분 문자열 일치 사용 불가능하게 합니다. 즉, 별표(*)로 끝나는 문자열을 사용하지 않습니다. |

예를 들어, 다음 문장은 *State* 필드의 값을 비교할 때 대소문자를 무시하는 필터를 설정합니다.

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```



## 필터링된 데이터셋의 레코드 탐색

필터링된 데이터셋의 레코드를 탐색하는 데이터셋 메소드는 네 가지가 있습니다. 다음 표는 이러한 메소드를 나열하고 메소드의 기능을 설명합니다.

표 16.6 필터링된 데이터셋의 탐색 메소드

| 메소드              | 용도                                                                                            |
|------------------|-----------------------------------------------------------------------------------------------|
| <i>FindFirst</i> | 현재 필터 조건과 일치하는 데이터셋의 첫 번째 레코드로 이동합니다. 첫 번째로 일치하는 레코드의 검색은 항상 필터링되지 않은 데이터셋의 첫 번째 레코드에서 시작합니다. |
| <i>FindLast</i>  | 현재 필터 조건과 일치하는 데이터셋의 마지막 레코드로 이동합니다.                                                          |
| <i>FindNext</i>  | 필터링된 데이터셋의 현재 레코드에서 다음 레코드로 이동합니다.                                                            |
| <i>FindPrior</i> | 필터링된 데이터셋의 현재 레코드에서 이전 레코드로 이동합니다.                                                            |

예를 들어, 다음 문장은 데이터셋의 첫 번째 필터링된 레코드를 찾습니다.

```
DataSet1.FindFirst;
```

애플리케이션에 대해 *Filter* 속성을 설정하거나 *OnFilterRecord* 이벤트 핸들러를 생성한다고 가정할 때, 이들 메소드는 현재 필터링이 사용 가능한지와는 상관없이 지정된 레코드에 커서를 둡니다. 필터링이 사용 가능하지 않을 때 이들 메소드를 호출하면 다음과 같이 됩니다.

- 일시적으로 필터링을 사용 가능하게 합니다.
- 일치하는 레코드를 찾으면 이 레코드에 커서를 둡니다.
- 필터링을 사용 불가능하게 합니다.

**참고** 필터링이 사용 불가능하고, *Filter* 속성을 설정하지 않거나 *OnFilterRecord* 이벤트 핸들러를 생성하지 않는 경우, 이 메소드는 *First*, *Last*, *Next* 및 *Prior*와 동일한 작업을 수행합니다.

모든 탐색 필터 메소드는 커서를 일치하는 레코드(있을 경우)에 두고, 이 레코드를 현재 레코드로 만든 다음 *True*를 반환합니다. 일치하는 레코드를 찾지 못하면, 커서 위치는 변하지 않고 이 메소드는 *False*를 반환합니다. 호출을 랩(wrap)하기 위해 *Found* 속성의 상태를 확인하여, *Found*가 *True*인 적절한 동작을 수행할 수 있습니다. 예를 들어, 커서가 이미 데이터셋의 일치하는 마지막 레코드에 있고 *FindNext*를 호출하는 경우, 메소드는 *False*를 반환하고 현재 레코드는 변경되지 않습니다.

## 데이터 수정

데이터셋의 읽기 전용 *CanModify* 속성이 *True*이면, 아래 나열된 데이터셋 메소드를 사용하여 데이터를 삽입, 업데이트 및 삭제할 수 있습니다. 데이터셋이 단방향이 아니거나, 데이터셋을 원본으로 하는 데이터베이스가 쓰기 및 읽기 권한을 허용하지 않거나, 일부 다른 요인 (클라이언트 데이터셋의 *ReadOnly* 속성 또는 데이터셋 프로바이더의 *poReadOnly* 옵션)이 없다면, *CanModify*는 *True*입니다.

**표 16.7** 데이터 삽입, 업데이트 및 삭제를 위한 데이터셋 메소드

| 메소드           | 설명                                                                                                 |
|---------------|----------------------------------------------------------------------------------------------------|
| <i>Edit</i>   | 데이터셋이 아직 <i>dsEdit</i> 또는 <i>dsInsert</i> 상태가 아닌 경우, 데이터셋을 <i>dsEdit</i> 상태로 둡니다.                  |
| <i>Append</i> | 보류 중인 데이터를 포스트하고, 현재 레코드를 데이터셋의 끝 부분으로 이동하며, 데이터셋을 <i>dsInsert</i> 상태로 둡니다.                        |
| <i>Insert</i> | 모든 보류 중인 데이터를 포스트하고, 데이터셋을 <i>dsInsert</i> 상태로 둡니다.                                                |
| <i>Post</i>   | 새로운 레코드나 변경된 레코드를 데이터베이스에 포스트하려고 시도합니다. 성공하면 데이터셋은 <i>dsBrowse</i> 상태가 되고, 실패하면 데이터셋은 현재 상태로 남습니다. |
| <i>Cancel</i> | 현재 작업을 취소하고 데이터셋을 <i>dsBrowse</i> 상태로 둡니다.                                                         |
| <i>Delete</i> | 현재 레코드를 삭제하고 데이터셋을 <i>dsBrowse</i> 상태로 둡니다.                                                        |

## 레코드 편집

데이터셋은 애플리케이션이 레코드를 수정하기 전에 *dsEdit* 모드여야 합니다. 데이터셋의 읽기 전용 *CanModify* 속성이 *True*인 경우, 코드에서 *Edit* 메소드를 사용하여 데이터셋을 *dsEdit* 모드로 둘 수 있습니다.

애플리케이션의 폼에서 다음의 경우에 일부 data-aware 컨트롤은 데이터셋을 자동으로 *dsEdit* 상태로 둘 수 있습니다.

- 컨트롤의 *ReadOnly* 속성은 기본값인 *False*입니다.
- 컨트롤에 대한 데이터 소스의 *AutoEdit* 속성은 *True*입니다.
- *CanModify*는 데이터셋에 대해 *True*입니다.

**참고** 데이터셋이 *dsEdit* 상태인 경우라도, 애플리케이션 사용자가 적절한 SQL 액세스 권한이 없으면 레코드 편집이 성공하지 못할 수도 있습니다.

일단 데이터셋이 *dsEdit* 모드이면, 사용자는 폼의 data-aware 컨트롤에 나타나는 현재 레코드에 대한 필드 값을 수정할 수 있습니다. 사용자가 현재 레코드를 변경하는 작업(예, 그리드의 다른 레코드로 이동)을 수행할 때 편집이 사용 가능한 data-aware 컨트롤은 자동으로 *Post*를 호출합니다.

폼에 탐색기 컴포넌트를 제공하면, 사용자는 탐색기의 Cancel 버튼을 클릭하여 편집을 취소할 수 있습니다. 편집을 취소하면 데이터셋을 *dsBrowse* 상태로 돌려 놓습니다.

코드에서 적절한 메소드를 호출하여 편집 내용을 작성하거나 취소해야 합니다. 변경 내용은 *Post*를 호출하여 작성합니다. *Cancel*을 호출하여 변경 내용을 취소합니다. 코드에서 *Edit*와 *Post*는 주로 함께 사용됩니다. 예를 들어,

```

with CustTable do
begin
    Edit;
    FieldValues['CustNo'] := 1234;
    Post;
end;

```

앞의 예제에서 코드의 첫 번째 줄은 데이터셋을 *dsEdit* 모드로 둡니다. 코드의 다음 줄은 숫자 1234를 현재 레코드의 *CustNo* 필드에 할당합니다. 마지막으로, 마지막 줄은 수정된 레코드를 변경 로그에 작성하거나 포스트합니다.

**참고** 변경 로그의 편집 내용을 데이터베이스에 다시 작성하는 경우, 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출해야 합니다. 클라이언트 데이터셋의 편집에 대한 자세한 내용은 20-14 페이지의 "데이터 편집"을 참조하십시오.

## 새 레코드 추가

애플리케이션이 새 레코드를 추가할 수 있으려면 데이터셋은 *dsInsert* 모드여야 합니다. 코드에서, 데이터셋의 읽기 전용 *CanModify* 속성이 *True*인 경우, *Insert* 또는 *Append* 메소드를 사용하여 데이터셋을 *dsInsert* 모드로 둘 수 있습니다.

애플리케이션의 폼에서, 다음의 경우에 data-aware 그리드와 탐색기 컨트롤은 데이터셋을 *dsInsert* 상태로 둘 수 있습니다.

- 컨트롤의 *ReadOnly* 속성은 기본값인 *False*입니다.
- *CanModify*는 데이터셋에 대해 *True*입니다.

일단 데이터셋이 *dsInsert* 모드이면, 사용자나 애플리케이션은 새 레코드에 연결된 필드에 값을 입력할 수 있습니다. 그리드와 탐색 컨트롤을 제외하면, 사용자에게 *Insert*와 *Append* 사이에 가시적인 차이는 없습니다. *Insert*를 호출하면 현재 레코드의 위 그리드에 빈 행이 새로 나타납니다. *Append*를 호출하면 그리드는 데이터셋의 마지막 레코드로 스크롤되고, 그리드의 아래에 빈 행이 나타나고, 데이터셋에 연결된 모든 탐색기 컴포넌트에서 *Next* 및 *Last* 버튼이 흐리게 표시됩니다.

삽입할 수 있는 data-aware 컨트롤은 사용자가 현재 레코드를 변경하는 작업(예를 들어, 그리드의 다른 레코드로의 이동)을 수행할 때 자동으로 *Post*를 호출합니다. 그렇지 않으면, 코드에서 *Post*를 호출해야 합니다.

*Post*는 새로운 레코드를 변경 로그에 기록합니다. 삽입과 추가 사항을 변경 로그에서 데이터베이스로 작성하려면, 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출합니다.

## 레코드 삽입

*Insert*는 현재 레코드 앞에 새로운 빈 레코드를 열고 빈 레코드를 현재 레코드로 만들어서, 사용자나 애플리케이션 코드가 레코드에 대한 필드 값을 입력할 수 있도록 합니다.

애플리케이션에서 *Post*를 호출하면 새로 삽입된 레코드는 다음과 같이 변경 로그에 작성됩니다.

- 인덱싱된 데이터셋에서 레코드는 데이터셋의 해당 인덱스를 기준으로한 위치에 삽입됩니다
- 인덱싱되지 않은 테이블에서 레코드는 데이터셋의 현재 위치에 삽입됩니다.

## 레코드 추가

*Append*는 데이터셋 끝에 새로운 빈 레코드를 열고, 빈 레코드를 현재 레코드로 만들어서 사용자나 애플리케이션 코드가 레코드에 대한 필드 값을 입력할 수 있도록 합니다.

애플리케이션에서 *Post*를 호출하면, 새로 추가된 레코드는 다음과 같이 변경 로그에 작성됩니다.

- 인덱싱된 데이터셋에서 레코드는 데이터셋의 해당 인덱스를 기준으로한 위치에 삽입됩니다
- 인덱싱되지 않은 데이터셋에서 레코드는 데이터셋의 끝에 추가됩니다.

**참고** 데이터셋이 업데이트를 변경 로그에 적용할 때, 원본으로 사용한 SQL 데이터베이스에 삽입 및 추가된 레코드의 실제 위치는 구현에 따라 다릅니다. 테이블이 인덱싱되면, 인덱스는 새로운 레코드 정보로 업데이트됩니다.

## 레코드 삭제

애플리케이션이 레코드를 삭제하기 전에 데이터셋은 활성 상태여야 합니다. *Delete*는 데이터셋의 현재 레코드를 변경 로그로 삭제하고, 데이터셋을 *dsBrowse* 모드로 둡니다. 삭제된 레코드 다음에 있는 레코드는 현재 레코드가 됩니다. 사용자가 *Apply Updates*를 호출할 때까지 삭제된 레코드는 변경 로그에서 삭제되지 않습니다.

폼에 탐색기 컴포넌트를 제공하면, 사용자가 탐색기의 *Delete* 버튼을 클릭하여 현재 레코드를 삭제할 수 있습니다. 코드에서 현재 레코드를 제거하려면 *Delete*를 명시적으로 호출해야 합니다.

## 데이터베이스에 데이터 포스트

*Post* 메소드는 데이터베이스 애플리케이션의 편집 기능에서 중요합니다. *Post*는 현재 레코드의 변경 내용을 변경 로그에 기록하지만 데이터셋의 상태에 따라 다르게 행동합니다.

- *dsEdit* 상태에서 *Post*는 수정된 레코드를 변경 로그에 작성합니다.
- *dsInsert* 상태에서 *Post*는 새 레코드를 변경 로그에 작성합니다.
- *dsSetKey* 상태에서 *Post*는 데이터셋을 *dsBrowse* 상태로 돌려 놓습니다.

포스트는 다른 프로시저의 일부로 명시적으로 또는 암시적으로 수행될 수 있습니다. 애플리케이션이 현재 레코드에서 떠날 때는 *Post*가 암시적으로 호출됩니다. 테이블이 *dsEdit* 또는 *dsInsert* 모드인 경우, *First*, *Next*, *Prior* 및 *Last* 메소드를 호출하면 *Post*가 수행됩니다. 또한 *Append* 및 *Insert* 메소드는 모든 보류 중인 데이터를 암시적으로 포스트합니다.

**경고** *Close* 메소드는 *Post*를 암시적으로 호출하지 않습니다. *BeforeClose* 이벤트를 사용하여 모든 보류 중인 편집 내용을 명시적으로 포스트합니다.

## 변경 취소

아직 *Post*를 직접 또는 간접적으로 호출하지 않은 경우, 애플리케이션은 현재 레코드의 변경 내용을 언제라도 취소할 수 있습니다. 예를 들어, 데이터셋이 *dsEdit* 모드이고 사용자가 하나 이상의 필드에서 데이터를 변경한 경우, 애플리케이션은 데이터셋에 대한 *Cancel* 메소드를 호출하여 레코드를 원래 값으로 다시 돌려 놓을 수 있습니다. *Cancel*을 호출하면 항상 데이터셋을 *dsBrowse* 상태로 돌려 놓습니다.

폼에서, 데이터셋에 연결된 탐색기 컴포넌트에 *Cancel* 버튼을 포함시켜서 사용자가 작업을 편집, 삽입, 또는 추가할 수 있도록 하거나 폼에 *Cancel* 버튼에 대한 코드를 직접 작성할 수 있습니다.

클라이언트 데이터셋은 추가 메소드를 사용하여 현재 레코드 이외의 레코드에서 편집을 취소합니다. 자세한 내용은 20-15 페이지의 "변경 취소"를 참조하십시오.

## 전체 레코드 수정

폼에서 그리드와 탐색기를 제외한 모든 data-aware 컨트롤은 레코드의 단일 필드에 대한 액세스를 제공합니다.

그러나 코드에서 데이터셋을 원본으로 하는 데이터베이스 테이블의 구조가 안정적이고 변경되지 않는다면, 전체 레코드 구조에 적용되는 메소드를 사용할 수 있습니다. 다음 표는 이들 레코드의 개별 필드가 아닌 전체 레코드에 사용할 수 있는 메소드를 요약해서 보여줍니다.

**표 16.8** 전체 레코드에 적용되는 메소드

| 메소드                                     | 설명                                                                                     |
|-----------------------------------------|----------------------------------------------------------------------------------------|
| <i>AppendRecord</i> ([array of values]) | 테이블의 끝에 지정된 열 값의 레코드를 추가합니다. <i>Append</i> 와 유사합니다. 암시적 <i>Post</i> 를 수행합니다.           |
| <i>InsertRecord</i> ([array of values]) | 레코드로 지정된 값을 테이블의 현재 커서 위치 앞에 삽입합니다. <i>Insert</i> 와 유사합니다. 암시적 <i>Post</i> 를 수행합니다.    |
| <i>SetFields</i> ([array of values])    | 해당 필드의 값을 설정합니다. <i>TField</i> 에 값을 할당하는 것과 유사합니다. 애플리케이션은 명시적 <i>Post</i> 를 수행해야 합니다. |

이들 메소드 각각은 인수로 값 배열을 사용합니다. 각각의 배열 값은 원본으로 사용한 데이터셋의 열에 해당합니다. 이 값은 리터럴, 변수 또는 Null일 수 있습니다. 인수 값의 수가 데이터셋 열의 수보다 작으면, 나머지 값들은 Null로 처리됩니다.

인덱싱되지 않은 데이터셋에서, *AppendRecord*는 데이터셋 끝에 레코드를 추가하고, *InsertRecord*는 현재 커서 위치 다음에 레코드를 삽입합니다. 인덱싱된 테이블에서 두 메소드는 모두 인덱스에 따라 테이블의 올바른 위치에 레코드를 둡니다. 두 가지 경우에 메소드는 커서를 레코드의 위치로 이동합니다.

*SetFields*는 매개변수 배열에 지정된 값을 데이터셋의 필드에 할당합니다. *SetFields*를 사용하려면 애플리케이션이 먼저 *Edit*를 호출하여 데이터셋을 *dsEdit* 모드로 두어야 합니다. 현재 레코드에 대한 변경 내용을 기록하려면 *Post*를 수행해야 합니다.

*SetFields*를 사용하여 기존 레코드에서 모든 필드가 아닌 일부 필드를 수정하는 경우, 변경을 원하지 않는 필드에 대해 Null 값을 전달할 수 있습니다. 레코드의 모든 필드에 대한 충분한 값을 제공하지 않으면, *SetFields*는 필드에 Null 값을 할당합니다. Null 값은 이미 필드에 있는 모든 기존 값을 덮어씁니다.

예를 들어, Name, Capital, Continent, Area 및 Population 열이 있는 CountryTable 이는 클라이언트 데이터셋을 생각해 봅니다. 다음 문장은 클라이언트 데이터셋에 레코드를 삽입합니다.

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

이 문장은 Area 및 Population에 대한 값을 지정하지 않으므로 이들 열에 대해서는 Null 값이 삽입됩니다. 데이터셋은 Name으로 인덱싱되어 있으므로 문장은 "Japan"의 알파벳 순서에 따라 레코드를 삽입합니다.

레코드를 업데이트하기 위해서 애플리케이션은 다음의 코드를 사용할 수 있습니다.

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

이 코드는 Area 및 Population 필드에 값을 할당하고 변경 로그에 포스트합니다. 세 개의 nil 인수는 첫 번째 세 개의 열에 대한 위치 표시자로서의 역할을 하여 현재 내용을 보존합니다.

## 데이터셋 이벤트 사용

*TDataSet*은 애플리케이션에서 검증을 수행하고 합계를 계산하며 다른 작업을 수행할 수 있도록 하는 많은 이벤트를 정의합니다. 다음 표는 이벤트를 나열한 것입니다.

표 16.9 데이터셋 이벤트

| 이벤트                       | 설명                              |
|---------------------------|---------------------------------|
| BeforeOpen, AfterOpen     | 데이터셋이 열리기 전/후에 호출됩니다.           |
| BeforeClose, AfterClose   | 데이터셋이 닫히기 전/후에 호출됩니다.           |
| BeforeInsert, AfterInsert | 데이터셋이 Insert 상태가 되기 전/후에 호출됩니다. |
| BeforeEdit, AfterEdit     | 데이터셋이 Edit 상태가 되기 전/후에 호출됩니다.   |
| BeforePost, AfterPost     | 테이블의 변경 내용이 포스트되기 전/후에 호출됩니다.   |
| BeforeCancel, AfterCancel | 이전 상태가 취소되기 전/후에 호출됩니다.         |

표 16.9 데이터셋 이벤트 ( 계속 )

| 이벤트                          | 설명                                    |
|------------------------------|---------------------------------------|
| BeforeDelete,<br>AfterDelete | 레코드가 삭제되기 전/후에 호출됩니다.                 |
| OnNewRecord                  | 새 레코드가 생성될 때 호출되어 기본 값을 설정하는 데 사용됩니다. |
| OnCalcFields                 | 계산된 필드가 계산될 때 호출됩니다.                  |

## 메소드 중지

*Open* 또는 *Insert*와 같은 메소드를 중지하려면, *Before* 이벤트 핸들러 (*BeforeOpen*, *BeforeInsert* 등)에서 *Abort* 프로시저를 호출합니다. 예를 들어, 다음 코드는 사용자에게 삭제 작업을 확인하도록 요청하거나 *Delete* 호출을 중지합니다.

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)begin
    if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
        Abort;
    end;
```

## OnCalcFields 사용

*OnCalcFields* 이벤트는 계산된 필드의 값을 설정하는 데 사용됩니다. *AutoCalcFields* 속성은 *OnCalcFields*를 호출할 시기를 결정합니다. *AutoCalcFields*가 *True*이면, 다음의 경우에 *OnCalcFields*가 호출됩니다.

- 데이터셋이 열릴 때
- 포커스가 한 비주얼 컴포넌트에서 다른 비주얼 컴포넌트로, 또는 data-aware 그리드 컨트롤의 한 열에서 다른 열로 이동하고, 현재 레코드가 수정되었을 때
- 데이터베이스에서 레코드를 검색할 때

*OnCalcFields*는 *AutoCalcFields*의 설정과는 상관없이 계산되지 않은 필드의 값이 변경될 때마다 항상 호출됩니다.

**주의** *OnCalcFields*는 자주 호출되기 되므로 이에 대해 작성된 코드는 짧아야 합니다. 또한 *AutoCalcFields*가 *True*이면, *OnCalcFields*는 데이터셋(또는 마스터-디테일 관계의 일부인 경우에 연결된 데이터셋)을 수정하는 어떤 작업도 수행하지 않아야 합니다. 왜냐하면 이 경우는 재귀 호출을 초래할 수 있기 때문입니다. 예를 들어, *OnCalcFields*가 *Post*를 수행하고 *AutoCalcFields*가 *True*인 경우, *OnCalcFields*가 다시 호출되어 *Post*를 다시 수행하게 됩니다.

*AutoCalcFields*가 *False*이면, *OnCalcFields*는 단일 레코드의 개별 필드가 수정될 때 호출되지 않습니다.

*OnCalcFields*가 실행될 때, 데이터셋은 *dsCalcFields* 모드이므로 사용자는 계산된 필드 이외의 필드 값을 설정할 수 없습니다. *OnCalcFields*가 완료된 후 데이터셋은 *dsBrowse* 상태로 돌아옵니다.





## 필드 컴포넌트 사용

이 장은 *TField* 객체와 그 자손에 공통적인 속성, 이벤트 및 메소드를 설명합니다. 필드 컴포넌트는 데이터셋의 개별 필드(열)를 나타냅니다. 이 장은 또한 애플리케이션의 데이터 표시 및 편집을 조정하기 위해 필드 컴포넌트를 사용하는 방법을 설명합니다.

필드 컴포넌트는 항상 데이터셋과 연결되어 있습니다. *TField* 객체는 사용자의 애플리케이션에서 직접 사용되지 않습니다. 대신, 사용자 애플리케이션의 각 필드 컴포넌트는 데이터셋의 열 데이터 타입별 *TField* 자손입니다. 필드 컴포넌트는 연결된 데이터셋의 특정 열의 데이터에 대한 *TDBEdit* 및 *TDBGrid* 액세스와 같이 data-aware 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*에는 데이터의 모습에 직접적으로 영향을 주는 속성이 네 가지 있습니다.

**표 17.1** 데이터 표시에 영향을 주는 *TFloatField* 속성

| 속성                   | 용도                                             |
|----------------------|------------------------------------------------|
| <i>Alignment</i>     | 왼쪽, 가운데, 또는 오른쪽 정렬의 데이터 표시 방법을 지정합니다.          |
| <i>DisplayWidth</i>  | 한 번에 컨트롤에 표시할 수 있는 숫자의 수를 지정합니다.               |
| <i>DisplayFormat</i> | 소수점 이하 자릿수를 얼마나 표시할 것인가와 같은 표시할 데이터 서식을 지정합니다. |
| <i>EditFormat</i>    | 편집 중에 값을 표시하는 방법을 지정합니다.                       |

데이터셋의 레코드에서 레코드로 스크롤할 때, 필드 컴포넌트를 사용하면 현재 레코드의 필드값을 보거나 변경할 수 있습니다.

필드 컴포넌트는 서로 공통적인 많은 속성(예를 들어, *DisplayWidth* 및 *Alignment*)을 갖고 있으며, 데이터 타입별 속성(예를 들어, *TFloatField*의 *Precision*)도 갖고 있습니다. 이들 각 속성은 데이터가 폼에서 데이터가 애플리케이션의 사용자에게 나타나는 방식에 영향을 줍니다. *Precision*과 같은 일부 속성은 또한 데이터 수정이나 입력 시 사용자가 컨트롤에 입력할 수 있는 데이터 값에도 영향을 줍니다.

데이터셋의 모든 필드 컴포넌트는 동적(원본으로 사용한 데이터베이스 테이블의 구조에 따라 자동으로 생성)이거나 영구적(Fields editor에서 설정하는 특정 필드 이름과 속성으로 생성)입니다. 동적 필드와 영구적 필드는 각기 다른 장점을 갖고 있으며, 각기 다른 애플리케이션 타입에 적합합니다. 다음 단원에서는 동적 필드와 영구적 필드를 좀 더 자세히 설명하고, 이들의 선택에 대한 조언을 제공합니다.

## 동적 필드 컴포넌트

동적으로 생성된 필드 컴포넌트는 기본값입니다. 사실, 데이터셋의 모든 필드 컴포넌트는 데이터 모듈에 데이터셋을 처음 둘 때 동적 필드로 시작되고, 데이터셋이 데이터를 폐치하는 방법을 지정하며 데이터셋을 엽니다. 데이터셋에서 나타나는 원본으로 사용한 데이터의 실제 특성에 따라 자동으로 생성되는 필드 컴포넌트는 동적필드 컴포넌트입니다. 데이터셋은 원본으로 사용한 데이터의 각 열에 대해 하나의 필드 컴포넌트를 작성합니다. 각각의 열에 대해 생성된 정확한 *TField* 자손은 클라이언트 데이터셋에 대한 데이터베이스나 프로바이더 컴포넌트에서 받은 필드 타입 정보에 의해 결정됩니다.

동적 필드는 일시적입니다. 동적 필드는 데이터셋이 열려 있는 동안에만 존재합니다. 동적 필드를 사용하는 데이터셋을 다시 열 때마다, 데이터셋은 데이터셋을 원본으로 하는 데이터의 현재 구조에 따라 완전히 새로운 동적 필드 컴포넌트 집합을 다시 만듭니다. 원본으로 사용한 데이터의 열이 변경되고 동적 필드 컴포넌트를 사용하는 데이터셋을 연 경우, 자동으로 생성되는 필드 컴포넌트는 또한 이를 반영하기 위해 변경됩니다.

데이터 표시 및 편집에 대해 유동적이어야 하는 애플리케이션에서 동적 필드를 사용합니다. 예를 들어, 데이터베이스 찾아보기 툴을 생성하려면, 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 둡니다.
- 2 데이터셋을 데이터에 연결합니다. 이렇게 하면 데이터 소스에 연결하기 위해 연결 컴포넌트나 프로바이더를 사용할 수 있으며 데이터셋이 나타내는 데이터를 지정하는 속성을 설정할 수 있습니다.
- 3 데이터 소스를 데이터셋에 연결합니다.
- 4 data-aware 컨트롤을 애플리케이션의 폼에 두고, 데이터 모듈을 각 폼의 유닛에 대한 각각의 사용 질에 추가하고, 각 data-aware 컨트롤을 모듈의 데이터 소스에 연결합니다. 또한 이를 필요로 하는 각각의 data-aware 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 필드는 제한적으로 사용될 수 있습니다. 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 생성할 수 없고,

동적 필드의 기본 데이터 타입을 오버라이드할 수 없습니다. 데이터베이스 애플리케이션에서 필드에 대한 조정 및 유동성을 얻으려면, Fields editor를 불러서 데이터셋에 대한 영구적 필드 컴포넌트를 생성해야 합니다.

## 영구적 필드 컴포넌트

기본적으로 데이터셋 필드는 동적입니다. 필드의 속성과 사용 가능성은 자동으로 설정되고 어떠한 방식으로든 변경될 수 없습니다. 필드의 속성과 이벤트를 조정하려면, 데이터셋에 대한 영구적 필드를 생성해야 합니다. 영구적 필드를 사용하면 다음 작업을 수행할 수 있습니다.

- 필드의 표시를 설정 또는 변경하거나 디자인 타임이나 런타임 시 특성을 편집합니다.
- 조회 필드, 계산된 필드 및 집계 필드와 같은 새 필드를 생성할 때 데이터셋의 기존 필드의 값을 기준으로 합니다.
- 데이터 입력을 검증합니다.
- 영구적 컴포넌트 목록에서 필드 컴포넌트를 제거하여 애플리케이션 원본으로 사용하는 데이터베이스의 특정 열에 액세스하지 못하도록 합니다.
- 새 필드를 정의하여 데이터셋을 원본으로 사용하는 테이블이나 쿼리의 열에 따라 기존 필드를 교체합니다.

디자인 타임에, Fields editor를 사용하여 애플리케이션의 데이터셋에서 사용하는 필드 컴포넌트의 영구적 목록을 만들 수 있습니다. 영구적 필드 컴포넌트 목록은 애플리케이션에 저장되는데 원본으로 사용한 데이터셋의 데이터베이스 구조가 변경되더라도 변경되지 않습니다. 일단 Fields editor로 영구적 필드를 생성하면, 또한 데이터 값의 변경에 응답하고 데이터 입력을 검증하는 이벤트 핸들러를 생성할 수 있습니다.

**참고** 데이터셋에 대한 영구적 필드를 생성할 때 사용자가 선택하는 필드만이 디자인 타임 및 런타임 시 사용자 애플리케이션에서 사용될 수 있습니다. 디자인 타임에, 항상 Fields editor를 사용하여 데이터셋에 대한 영구적 필드를 추가하거나 삭제할 수 있습니다.

단일 데이터셋에서 사용하는 모든 필드는 영구적이거나 동적입니다. 단일 데이터셋의 필드 타입들을 혼합할 수 없습니다. 데이터셋에 대한 영구적 필드를 작성한 다음 동적 필드로 변환하려면, 데이터셋에서 모든 영구적 필드를 제거해야 합니다. 동적 필드에 대한 자세한 내용은 17-2 페이지의 "동적 필드 컴포넌트"를 참조하십시오.

**참고** 영구적 필드를 일차적인 용도 중 하나는 데이터의 모습과 표시를 조정하는 것입니다. 또한 data-aware 그리드의 열 모습을 조정할 수도 있습니다. 그리드의 열 모양 조정에 대한 내용은 15-16 페이지의 "사용자 지정 그리드 생성"을 참조하십시오.

## 영구적 필드 생성

Fields editor로 생성된 영구적 필드 컴포넌트는 원본으로 사용한 데이터에 대해 효과적이고 읽기 가능하며 타입을 고려하지 않아도 되는 (type-safe) 프로그램 액세스를 제공합니다. 영구적 필드 컴포넌트를 사용하면, 애플리케이션이 실행될 때마다 원본으로 사용한 데이터의 실제 구조가 변경되더라도 항상 동일한 순서로 동일한 열이 사용되고 표시됩니다. 특정 필드에 따라 달라지는 data-aware 컴포넌트와 프로그램 코드는 항상 의도하는대로 작동합니다. 영구적 필드 컴포넌트를 기반으로 하는 열이 삭제되거나 변경되면, 존재하지 않는 열이나 일치하지 않는 데이터에 대해 애플리케이션을 실행하지 않고 애플리케이션이 예외를 발생시킵니다.

다음과 같은 방법으로 데이터셋에 대한 영구적 필드를 생성합니다.

- 1 데이터 모듈에 데이터셋을 둡니다.
- 2 데이터셋을 원본으로 사용한 데이터에 결합시킵니다. 이렇게 하면 일반적으로 데이터셋이 연결 컴포넌트나 프로바이더에 연결되고 데이터를 설명하는 모든 속성이 지정됩니다. 예를 들어, *TSQLDataSet*을 사용하는 경우에는 적절히 구성된 *TSQLConnection* 컴포넌트에 대한 *SQLConnection* 속성을 설정하고, 유효한 쿼리에 대한 *CommandText* 속성을 설정할 수 있습니다.
- 3 데이터 모듈의 데이터셋 컴포넌트를 더블 클릭하면 Fields editor를 불러옵니다. Fields editor에는 제목 표시줄, 탐색기 버튼 및 목록 상자가 있습니다.

Fields editor의 제목 표시줄은 데이터 모듈 또는 데이터셋을 포함하는 품의 이름과 데이터셋의 이름을 직접 표시합니다. 예를 들어, *CustomerData* 데이터 모듈의 *Customers* 데이터셋을 열면, 제목 표시줄은 'CustomerData.Customers' 또는 적합한 만큼의 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 활성이 아니거나 비어 있으면 탐색 버튼이 흐리게 표시됩니다. 데이터셋이 단방향인 경우에는 마지막 레코드와 이전 레코드로 이동하는 버튼이 항상 흐리게 표시됩니다.

목록 상자에는 데이터셋에 대한 영구적 필드 컴포넌트의 이름을 표시합니다. 새 데이터셋에 대해 Fields editor를 처음 불러오면 데이터셋의 필드 컴포넌트가 영구적이지 않고 동적이기 때문에 목록이 비어 있습니다. 이미 영구적 필드 컴포넌트가 있는 데이터셋에 대해 Fields editor를 실행하면, 목록 상자에 필드 컴포넌트 이름이 표시됩니다.

- 4 Fields editor 컨텍스트 메뉴에서 Add Fields를 선택합니다.
- 5 Add Fields 대화 상자에서 영구적으로 만들 필드를 선택합니다. 기본적으로 대화 상자가 열릴 때 모든 필드가 선택됩니다. 사용자가 선택하는 모든 필드는 영구적 필드가 됩니다.

Add Fields 대화 상자가 닫히면, 사용자가 선택한 필드가 Fields editor 목록 상자에 나타납니다. Fields editor 목록 상자의 필드는 영구적입니다. 또한 데이터셋이 활성 상태이면, 목록 상자 위의 다음 및 마지막 탐색 버튼(데이터셋이 단방향이 아닌 경우)을 사용할 수 있습니다.

그러면 이제부터는 데이터셋을 열 때마다, 원본으로 사용한 데이터베이스의 모든 열에 대한 동적 필드 컴포넌트를 더 이상 생성하지 않습니다. 그 대신, 사용자가 지정한 필드에 대한 영구적 컴포넌트만을 생성합니다.

데이터셋을 열 때마다, 각각의 계산되지 않은 영구적 필드가 데이터베이스에 있거나 데이터베이스의 데이터에서 생성될 수 있는지를 검증합니다. 그렇지 않으면, 데이터셋은 필드가 유효하지 않아 열리지 않는다는 것을 사용자에게 경고하는 예외를 발생시킵니다.

## 영구적 필드 정렬

영구적 필드 컴포넌트가 Fields editor 목록 상자에 나열되는 순서는 필드가 data-aware 그리드 컴포넌트에 나타나는 기본 순서입니다. 목록 상자에서 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.

여러 비연속적인 필드 집합을 선택하여 새로운 위치로 끌어가면, 연속적인 블록으로 삽입됩니다. 이 블록 내에서 필드의 순서는 변경되지 않습니다.

다른 방법으로는, 필드를 선택하고 *Ctrl+Up* 및 *Ctrl+Dn*을 사용하여 목록에서 개별 필드의 순서를 변경할 수 있습니다.

## 새 영구적 필드 정의

기존 데이터셋 필드를 영구적 필드로 만드는 것 외에, 데이터셋의 다른 영구적 필드에 대한 교체 또는 추가로서 특별한 영구적 필드를 생성할 수도 있습니다.

사용자가 생성하는 새 영구적 필드는 표시 목적으로만 사용합니다. 이 필드가 런타임 시 포함하는 데이터는 이미 데이터베이스의 다른 곳에 있거나 임시적이기 때문에 유지되지 않습니다. 데이터셋의 원본으로 사용한 데이터의 실제 구조는 어떤 방식으로든 변경되지 않습니다.

새 영구적 필드 컴포넌트를 작성하려면, Fields editor에서 컨텍스트 메뉴를 불러서 New field를 선택합니다. New Field 대화 상자가 나타납니다.

New Field 대화 상자에는 Field properties, Field type 및 Lookup definition 세 개의 그룹 상자가 있습니다.

- Field properties 그룹 상자를 사용하면 일반적인 필드 컴포넌트 정보를 입력할 수 있습니다. Name 편집 상자에 필드 이름을 입력합니다. 여기서 입력하는 이름은 필드 컴포넌트의 *FieldName* 속성과 일치합니다. New Field 대화 상자는 이 이름을 사용하여 Component 편집 상자의 컴포넌트 이름을 만듭니다. Component 편집 상자에 표시되는 이름은 필드 컴포넌트의 *Name* 속성과 일치하고, 정보 목적을 위해서만 제공됩니다 (*Name*은 사용자 소스 코드의 필드 컴포넌트를 참조하는 식별자입니다). 대화 상자는 Component 편집 상자에 직접 입력하는 모든 내용을 버립니다.

- Field properties 그룹의 Type 콤보 상자를 사용하면 필드 컴포넌트의 데이터 타입을 지정할 수 있습니다. 생성하는 모든 새로운 필드 컴포넌트에 대해 데이터 타입을 제공해야 합니다. 예를 들어, 필드에 부동 소수점 화폐 값을 표시하려면, 드롭다운 목록에서 *Currency*를 선택합니다. Size 편집 상자를 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. 기본 타입은 Data입니다. Lookup을 선택하면, Lookup definition 그룹 상자의 Dataset 및 Source Fields 편집 상자를 사용할 수 있습니다. 또한 Calculated 필드를 작성할 수 있으며, 클라이언트 데이터셋에서 작업하는 경우에는 InternalCalc 필드나 Aggregate 필드를 작성할 수 있습니다. 다음 표는 작성할 수 있는 필드 타입을 설명한 것입니다.

표 17.2 특수한 영구적 필드 종류

| 필드 종류        | 용도                                                                 |
|--------------|--------------------------------------------------------------------|
| Data         | 기존의 필드를 대체합니다(예를 들어, 데이터 타입 변경).                                   |
| Calculated   | 런타임 시 데이터셋의 <i>OnCalcFields</i> 이벤트 핸들러로 계산된 값을 표시합니다.             |
| Lookup       | 런타임 시 사용자가 지정한 검색 기준에 따라 지정된 데이터셋에서 값을 검색합니다.(단방향 데이터셋에서는 지원되지 않음) |
| InternalCalc | 런타임 시 클라이언트 데이터셋으로 계산되고 데이터가 저장된 값을 표시합니다.                         |
| Aggregate    | 클라이언트 데이터셋에서 레코드 집합 데이터를 요약하는 값을 표시합니다.                            |

Lookup definition 그룹 상자는 조회 필드를 작성하기 위해서만 사용됩니다. 보다 자세한 설명은 17-8 페이지의 "조회 필드 정의"를 참조합니다.

## 데이터 필드 정의

데이터 필드는 데이터셋의 기존 필드를 교체합니다. 예를 들어, 프로그램적인 이유로 인해 *TSmallIntField*를 *TIntegerField*로 교체해야 할 수도 있습니다. 필드의 데이터 타입은 직접 변경할 수 없기 때문에 이를 교체할 새로운 필드를 정의해야 합니다.

**중요** 기존의 필드를 교체할 새로운 필드를 정의하는 경우에도, 정의하는 필드는 원본으로 사용한 데이터셋의 테이블에 있는 기존의 열에서 데이터 값을 파생해야 합니다.

원본으로 사용한 데이터셋의 테이블에 있는 필드에 대한 교체 데이터 필드를 작성하려면, 다음 단계를 따르십시오.

- 1 데이터셋에 할당된 영구적 필드 목록에서 필드를 제거하고, 컨텍스트 메뉴에서 New Field를 선택합니다.
- 2 New Field 대화 상자에서 Name 편집 상자의 데이터베이스 테이블에 있는 기존 필드의 이름을 입력합니다. 새로운 필드 이름을 입력하지 마십시오. 실제로는 새로운 필드가 데이터를 파생할 필드의 이름을 지정하는 것입니다.
- 3 Type 콤보 상자에서 필드의 데이터 타입을 선택합니다. 선택하는 데이터 타입은 교체하는 필드의 데이터 타입과 달라야 합니다. 크기가 다른 문자열 필드는 서로 교체할 수 없습니다. 데이터 타입은 달라야 하지만, 원본 테이블의 필드의 실제 데이터 타입과는 호환되어야 합니다.

- 4 Size 편집 상자에 적당한 필드의 크기를 입력합니다. 크기는 *TStringField*, *TBytesField*, *TVarBytesField* 등과 같은 필드 타입에만 적용됩니다.
- 5 아직 선택되지 않았으면 Field type 라디오 그룹에서 Data를 선택합니다.
- 6 OK를 선택합니다. New Field 대화 상자가 닫히고, 새로 정의된 데이터 필드가 1 단계에서 지정한 기존 필드를 교체하고, 데이터 모듈의 컴포넌트 선언이나 폼의 **type** 선언이 업데이트됩니다.

필드 컴포넌트와 연결된 속성 또는 이벤트를 편집하려면, Field Editor 목록 상자에서 컴포넌트 이름을 선택한 다음 Object Inspector를 사용하여 속성 또는 이벤트를 편집합니다. 필드 컴포넌트 속성 및 이벤트의 편집에 대한 자세한 내용은 17-11 페이지의 "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 대화 상자에서 계산된 필드를 작성합니다.

- 1 Name 편집 상자에 계산된 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.
- 2 Type 콤보 상자에서 필드의 데이터 타입을 선택합니다.
- 3 Size 편집 상자에 적당한 필드의 크기를 입력합니다. 크기는 *TStringField*, *TBytesField*, *TVarBytesField* 등과 같은 필드 타입에만 적용됩니다.
- 4 Field type 라디오 그룹에서 Calculated 또는 InternalCalc를 선택합니다. InternalCalc는 클라이언트 데이터셋에서 작업하는 경우에만 사용할 수 있습니다. 이들 계산된 필드의 타입 사이의 중요한 차이점은 InternalCalc 필드에 대해 계산된 값은 클라이언트 데이터셋의 데이터의 일부로서 저장 및 검색된다는 점입니다.
- 5 OK를 선택합니다. 새로 정의된 계산된 필드는 Field Editor 목록 상자의 영구적 필드 목록 끝에 자동으로 추가되며, 컴포넌트 선언은 폼 또는 데이터 모듈의 **type** 선언에 자동으로 추가됩니다.
- 6 필드에 대해 값을 계산하는 코드를 데이터셋에 대한 *OnCalcFields* 이벤트 핸들러에 둡니다. 필드값 계산을 위한 코드 작성에 대한 자세한 내용은 17-8 페이지의 "계산된 필드 프로그래밍"을 참조하십시오.

**참고** 필드 컴포넌트와 연결된 속성 또는 이벤트를 편집하려면, Field Editor 목록 상자에서 컴포넌트 이름을 선택한 다음 Object Inspector를 사용하여 속성 또는 이벤트를 편집합니다. 필드 컴포넌트 속성 및 이벤트 편집에 대한 자세한 내용은 17-11 페이지의 "영구적 필드 속성 및 이벤트 설정"을 참조하십시오.

## 계산된 필드 프로그래밍

계산된 필드를 정의하고 나면 값을 계산하기 위한 코드를 작성해야 합니다. 그렇지 않으면 항상 Null 값을 가집니다. 계산된 필드의 코드는 데이터셋에 대한 *OnCalcFields* 이벤트에 둡니다.

다음과 같은 방법으로 계산된 필드에 대한 값을 프로그래밍합니다.

- 1 Object Inspector 드롭다운 목록에서 데이터셋 컴포넌트를 선택합니다.
- 2 Object Inspector Events 페이지를 선택합니다.
- 3 *OnCalcFields* 속성을 더블 클릭하여 데이터셋 컴포넌트에 대한 *CalcFields* 이벤트 핸들러를 불러오거나 생성합니다.
- 4 계산된 필드의 값과 속성을 설정하는 코드를 원하는대로 작성합니다.

예를 들어, *CustomerData* 데이터 모듈의 *Customers* 테이블에 대한 *CityStateZip*이라는 계산된 필드를 생성했다고 가정합니다. *CityStateZip*은 data-aware 컨트롤에서 한 줄로 회사의 주소와 우편 번호를 표시합니다.

*Customers* 테이블에 대한 *CalcFields* 프로시저에 코드를 추가하려면, Object Inspector 드롭다운 목록에서 *Customers* 테이블을 선택하고, Events 페이지로 변환하여, *OnCalcFields* 속성을 더블 클릭합니다.

*TCustomerData.CustomersCalcFields* 프로시저는 유닛의 소스 코드 창에 나타납니다. 다음 코드를 프로시저에 추가하여 필드를 계산합니다.

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
+ ' ' + CustomersZip.Value;
```

**참고** 내부적으로 계산된 필드에 대한 *OnCalcFields* 이벤트 핸들러를 작성할 때, *State가 dsInternalCalc*인 경우에 클라이언트 데이터셋의 *State* 속성을 확인하고 그 값을 단지 다시 계산하여 성능을 향상시킬 수 있습니다. 자세한 내용은 20-20 페이지의 "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 필드값, 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 가정합니다. 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 대화 상자에서 조회 필드를 생성합니다.

- 1 Name 편집 상자에 조회 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.
- 2 Type 콤보 상자에서 필드의 데이터 타입을 선택합니다.
- 3 Size 편집 상자에 적당한 필드의 크기를 입력합니다. 크기는 *TStringField*, *TBytesField*, *TVarBytesField* 등과 같은 필드 타입에만 적용됩니다.
- 4 Field type 라디오 그룹에서 Lookup을 선택합니다. Lookup을 선택하면 Dataset 및 Key Fields 콤보 상자를 사용할 수 있습니다.
- 5 Dataset 콤보 상자 드롭다운 목록에서 필드값을 조회할 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 상자를 사용할 수 있습니다.
- 6 Key Fields 드롭다운 목록에서 값을 일치시킬 현재 데이터셋의 필드를 선택합니다. 하나 이상의 필드를 일치시키려면, 드롭다운 목록에서 선택하는 대신 필드 이름을 직접 입력합니다. 세미콜론을 사용하여 여러 필드 이름을 구분합니다. 하나 이상의 필드를 사용하는 경우에는 영구적 필드 컴포넌트를 사용해야 합니다.
- 7 Lookup Keys 드롭다운 목록에서 6 단계에서 지정한 Source Fields 필드에 대해 일치시킬 조회 데이터셋의 필드를 선택합니다. 하나 이상의 키 필드를 지정한 경우에는 동일한 수의 조회 키를 지정해야 합니다. 하나 이상의 필드를 지정하려면, 필드 이름을 직접 입력하여 세미콜론으로 여러 필드 이름을 구분합니다.
- 8 Result Field 드롭다운 목록에서 생성 중인 조회 필드의 값으로 반환할 조회 데이터셋의 필드를 선택합니다.

애플리케이션을 디자인하고 실행할 때, 조회 필드값은 계산된 필드값이 계산되기 전에 결정됩니다. 계산된 필드는 조회 필드에 기준을 둘 수 있지만, 조회 필드는 계산된 필드에 기준을 둘 수가 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 값이 동적으로 조회되는지의 여부를 결정합니다. *LookupCache*를 *True*로 설정하면, *LookupDataSet*이 거의 변경되지 않으며 다른 조회 값의 수가 작은 경우 조회 필드값을 캐시로 저장합니다. 조회 값을 캐시로 저장하면, *DataSet*이 열릴 때 *LookupKeyFields* 값의 모든 집합에 대한 조회 값이 미리 로드되기 때문에 수행 속도가 빨라집니다. *DataSet*의 현재 레코드가 변경되면, 필드 객체는 *LookupDataSet*에 액세스하지 않고 캐시에서 *Value*를 찾을 수 있습니다. 이러한 성능 향상은 *LookupDataSet*이 액세스가 느린 네트워크 상에 있는 경우 특히 두드러집니다.

**팁** 조회 캐시를 사용하면 보조 데이터셋에서가 아닌 프로그램으로 조회 값을 제공할 수 있습니다. *LookupDataSet* 속성이 *nil*인지 확인합니다. 그런 다음, *LookupList* 속성의 *Add* 메소드를 사용하여 조회 값으로 채웁니다. *LookupCache* 속성을 *True*로 설정합니다. 필드는 조회 데이터셋의 값으로 겹쳐쓰지 않고 제공된 조회 목록을 사용합니다.

*DataSet*의 모든 레코드가 *KeyFields*에 대해 각기 다른 값을 가지면, 캐시의 값을 찾는 오버헤드가 캐시에서 제공하는 성능상의 장점보다 더 클 수 있습니다. 캐시의 값을 찾는 오버헤드는 *KeyFields*에서 취할 수 있는 다른 값의 수로 증가합니다.

*LookupDataSet*이 순간적인 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 추출된 집계로부터의 값을 표시합니다. 집계는 레코드 집합의 데이터를 요약하는 계산입니다. 추출된 집계에 대한 자세한 사항은 20-20 페이지의 "유지 관리되는 Aggregates 속성 사용"을 참조하십시오.

다음과 같은 방법으로 New Field 대화 상자에서 집계 필드를 생성합니다.

- 1 Name 편집 상자에 집계 필드의 이름을 입력합니다. 기존 필드의 이름을 입력하지 마십시오.
- 2 Type 콤보 상자에서 필드의 데이터 타입을 선택합니다.
- 3 Field 타입 라디오 그룹에서 Aggregate를 선택합니다.
- 4 OK를 선택합니다. 새로 정의된 집계 필드가 자동으로 추가되고, 클라이언트 데이터셋의 *Aggregates* 속성이 적절한 집계 사양을 포함하도록 자동으로 업데이트됩니다.
- 5 새로 생성된 집계 필드의 *ExprText* 속성에 집계에 대한 계산을 둡니다. 집계 정의에 대한 자세한 내용은 20-20 페이지의 "Aggregates 지정"을 참조하십시오.

일단 영구적 *TAAggregateField*를 생성하면, *TDBText* 컨트롤을 집계 필드에 연결할 수 있습니다. 그러면 *TDBText* 컨트롤에서 클라이언트 데이터셋의 현재 레코드와 상대적인 집계 필드의 값을 표시합니다.

## 영구적 필드 컴포넌트 삭제

영구적 필드 컴포넌트를 삭제하면, 테이블의 사용 가능한 열의 하위 집합을 액세스하고 테이블의 열을 교체하기 위해 자신의 영구적 필드를 정의할 경우에 유용합니다. 다음과 같은 방법으로 데이터셋에 대한 하나 이상의 영구적 필드 컴포넌트를 삭제합니다.

- 1 Fields 편집 상자에서 삭제할 필드를 선택합니다.
- 2 *Del*을 누릅니다.

**참고** 또한 컨텍스트 메뉴의 Delete를 선택하여 선택된 필드를 삭제할 수 있습니다.

삭제하는 필드는 더 이상 데이터셋에서 사용할 수 없으며, data-aware 컨트롤로 표시할 수도 없습니다. 실수로 삭제한 영구적 필드 컴포넌트는 항상 다시 생성할 수 있지만 속성이나 이벤트에 이전의 변경 내용은 없어집니다. 자세한 내용은 17-4 페이지의 "영구적 필드 생성"을 참조하십시오.

**참고** 데이터셋의 모든 영구적 필드 컴포넌트를 삭제하면, 데이터셋은 원본으로 사용한 데이터베이스 테이블의 모든 열에 동적 필드 컴포넌트를 사용하는 상태로 변환됩니다.

## 영구적 필드 속성 및 이벤트 설정

디자인 타임에 영구적 필드 컴포넌트에 대한 속성을 설정하고 이벤트를 사용자 지정할 수 있습니다. 속성은 `data-aware` 컴포넌트에서, 예를 들어, 필드가 `TDBGrid`에 나타날 수 있는지의 여부나 값을 수정할 수 있는지의 여부와 같은, 필드 표시 방식을 조정합니다. 이벤트는 필드의 데이터가 폐치, 변경, 설정, 또는 검증될 때 발생하는 내용들을 조정합니다.

필드 컴포넌트의 속성을 설정하거나 그에 대한 사용자 지정 이벤트 핸들러를 작성하려면, Fields editor에서 컴포넌트를 선택하거나 Object Inspector의 컴포넌트 목록에서 컴포넌트를 선택합니다.

## 디자인 타임 시 표시 및 편집 속성 설정

선택된 필드 컴포넌트의 표시 속성을 편집하려면, Object Inspector 창의 Properties 페이지로 변환합니다. 다음 표는 편집될 수 있는 표시 속성을 요약한 것입니다.

**표 17.3** 필드 컴포넌트 속성

| 속성                            | 용도                                                                                        |
|-------------------------------|-------------------------------------------------------------------------------------------|
| <i>Alignment</i>              | <code>data-aware</code> 컴포넌트 내에서 필드 내용을 왼쪽, 오른쪽 또는 가운데에 정렬합니다.                            |
| <i>ConstraintErrorMessage</i> | 제약 조건으로 충돌을 편집할 때 표시할 텍스트를 지정합니다.                                                         |
| <i>CustomConstraint</i>       | 편집 중에 데이터에 적용할 지역 제약 조건을 지정합니다.                                                           |
| <i>Currency</i>               | Numeric 필드에만 해당됩니다. <i>True</i> : 화폐 값을 표시합니다. <i>False</i> (기본값): 화폐 값을 표시하지 않습니다.       |
| <i>DisplayFormat</i>          | <code>data-aware</code> 컨트롤에 표시되는 데이터 형식을 지정합니다.                                          |
| <i>DisplayLabel</i>           | <code>data-aware</code> 그리드의 필드에 대한 열 이름을 지정합니다.                                          |
| <i>DisplayWidth</i>           | 이 필드를 표시하는 그리드 열의 문자 폭을 지정합니다.                                                            |
| <i>EditFormat</i>             | <code>data-aware</code> 컨트롤의 데이터의 편집 형식을 지정합니다.                                           |
| <i>EditMask</i>               | 문자의 지정된 타입 및 범위로 편집 가능한 필드의 데이터 입력을 제한하고, 필드 내에 나타나는 모든 특수한 편집 불가능한 문자(하이픈, 괄호 등)를 지정합니다. |
| <i>FieldKind</i>              | 데이터, 조회, 계산, 또는 집계 등의 필드 타입을 지정합니다.                                                       |
| <i>FieldName</i>              | 필드가 값과 데이터 타입을 파생시키는 테이블의 열 이름을 지정합니다.                                                    |
| <i>HasConstraints</i>         | 필드에 주어진 제약 조건이 있는지 나타냅니다.                                                                 |
| <i>ImportedConstraint</i>     | 데이터베이스 서버에서 가져온 SQL 제약 조건을 지정합니다. 이것은 필드가 다른 플랫폼에서 실행되는 애플리케이션 서버에서 값을 가져올 때에만 사용됩니다.     |
| <i>Index</i>                  | 데이터셋의 필드 순서를 지정합니다.                                                                       |
| <i>LookupDataSet</i>          | <i>Lookup</i> 이 <i>True</i> 일 때 필드값을 조회하는 데 사용되는 데이터셋을 지정합니다.                             |
| <i>LookupKeyFields</i>        | 조회를 수행할 때 일치시킬 조회 데이터셋의 필드를 지정합니다.                                                        |

표 17.3 필드 컴포넌트 속성 ( 계속 )

| 속성                       | 용도                                                                                                                                                 |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>LookupResultField</i> | 이 필드로 값을 복사할 조회 데이터셋의 필드를 지정합니다.                                                                                                                   |
| <i>MaxValue</i>          | Numeric 필드에만 해당됩니다. 사용자가 필드에 입력할 수 있는 최대 값을 지정합니다.                                                                                                 |
| <i>MinValue</i>          | Numeric 필드에만 해당됩니다. 사용자가 필드에 입력할 수 있는 최소 값을 지정합니다.                                                                                                 |
| <i>Name</i>              | 코드에서 필드 컴포넌트를 참조하기 위해 사용되는 이름을 지정합니다.                                                                                                              |
| <i>Origin</i>            | 원본으로 사용한 데이터베이스에 나타나는 것과 똑같이 필드의 이름을 지정합니다.                                                                                                        |
| <i>Precision</i>         | Numeric 필드에만 해당됩니다. 유효 자리수를 지정합니다.                                                                                                                 |
| <i>ReadOnly</i>          | <i>True</i> : data-aware 컴포넌트에 필드값을 표시하지만 편집할 수 없습니다.<br><i>False</i> (기본값): 필드값을 표시하고 편집할 수 있습니다.                                                 |
| <i>Size</i>              | 문자열 기반 필드에 표시하거나 입력할 수 있는 문자의 최대 수 또는 <i>TBytesField</i> 및 <i>TVarBytesField</i> 필드의 크기(바이트)를 지정합니다.                                               |
| <i>Tag</i>               | 필요에 따라 모든 컴포넌트에서 프로그래머가 사용할 수 있는 Long 정수                                                                                                           |
| <i>Transliterate</i>     | <i>True</i> (기본값): 데이터셋과 데이터베이스 사이로 데이터가 전송될 때 해당 로케일에 대한 번역이 일어나도록 지정합니다.<br><i>False</i> : 로케일 번역이 일어나지 않습니다.                                    |
| <i>Visible</i>           | <i>True</i> (기본값): data-aware 그리드에 필드를 표시할 수 있습니다.<br><i>False</i> : data-aware 그리드 컴포넌트의 필드를 표시할 수 없습니다. 사용자 지정 컴포넌트는 이 속성에 따라 표시 여부를 결정할 수 있습니다. |

모든 필드 컴포넌트가 다 모든 속성을 사용할 수 있는 것은 아닙니다. 예를 들어, *TStringField* 타입의 필드 컴포넌트는 *Currency*, *MaxValue*, 또는 *DisplayFormat* 속성이 없으며, *TFloatField* 타입의 컴포넌트는 *Size* 속성이 없습니다.

대부분의 속성의 목적은 직관적이지만, *Calculated*와 같은 일부 속성은 추가적인 프로그래밍 단계를 거쳐야 유용해질 수 있습니다. *DisplayFormat*, *EditFormat* 및 *EditMask*와 같은 기타 속성은 상호 관계를 가지며, 설정이 조정되어야 합니다. *DisplayFormat*, *EditFormat* 및 *EditMask* 사용에 대한 자세한 내용은 17-13 페이지의 "사용자 입력 조정 및 마스킹"을 참조하십시오.

## 런타임 시 필드 컴포넌트 속성 설정

런타임 시 필드 컴포넌트의 속성을 사용하고 조작할 수 있습니다. 예를 들어, 다음 코드는 *Customers* 테이블의 *CityStateZip* 필드에 대한 *ReadOnly* 속성을 *True*로 설정합니다.

```
CustomersCityStateZip.ReadOnly := True;
```

그리고 다음 문장은 *Customers* 테이블에 있는 *CityStateZip* 필드의 *Index* 속성을 3으로 설정하여 필드 순서를 변경합니다.

```
CustomersCityStateZip.Index := 3;
```

## 사용자 입력 조정 및 마스킹

*EditMask* 속성은 사용자가 *TStringField*, *TDateField*, *TTimeField*, *TDateTimeField* 및 *TSQLTimeStampField* 컴포넌트와 연결된 data-aware 컴포넌트에 입력할 수 있는 값의 타입과 범위를 조정하는 방법을 제공합니다. 기존의 마스크를 사용하거나 자신의 마스크를 생성할 수 있습니다. 가장 쉽게 편집 마스크를 사용하고 작성하는 방법은 Input Mask 편집기를 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 편집기를 실행합니다.

- 1 Fields editor나 Object Inspector에서 컴포넌트를 선택합니다.
- 2 Object Inspector의 Properties 페이지를 클릭합니다.
- 3 Object Inspector의 EditMask 필드에 대한 값 열을 더블 클릭하거나 생략 버튼을 클릭합니다. Input Mask 편집기가 열립니다.

Input Mask 편집 상자를 사용하면 마스크 형식을 만들고 편집할 수 있습니다. Sample Masks 그리드를 사용하면 이미 정의된 마스크에서 선택할 수 있습니다. 예제 마스크를 선택하면, 사용자가 수정하거나 있는 그대로 사용할 수 있는 Input Mask 편집 상자에 마스크 형식이 나타납니다. Test Input 편집 상자에서 마스크에 대해 허용할 수 있는 사용자 입력을 테스트할 수 있습니다.

사용자가 사용자 지정 마스크 집합을 생성한 경우에는 Masks 버튼을 사용하면 선택을 쉽게 하기 위해 그 마스크 집합을 Sample Masks 그리드로 로드할 수 있습니다.

## 숫자, 날짜 및 시간 필드에 대한 기본 서식 사용

Kylix는 *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, *TTimeField* 및 *TSQLTimeStampField* 컴포넌트에 기본 제공된 표시 및 편집 형식 루틴과 기능적인 기본 형식을 제공합니다. 이 루틴들을 사용하기 위해 해야 할 작업은 없습니다.

기본 서식은 다음과 같은 루틴으로 수행됩니다.

**표 17.4** 필드 컴포넌트 서식 루틴

| 루틴                          | 필드 컴포넌트...                                                                                |
|-----------------------------|-------------------------------------------------------------------------------------------|
| <i>FormatFloat</i>          | <i>TFloatField</i> , <i>TCurrencyField</i>                                                |
| <i>FormatDateTime</i>       | <i>TDateField</i> , <i>TTimeField</i> , <i>TDateTimeField</i> , <i>TSQLTimeStampField</i> |
| <i>SQLTimeStampToString</i> | <i>TSQLTimeStampField</i>                                                                 |
| <i>FormatCurr</i>           | <i>TCurrencyField</i> , <i>TBCDField</i>                                                  |
| <i>BcdToStrF</i>            | <i>TFMTBcdField</i>                                                                       |

필드 컴포넌트의 데이터 타입에만 적합한 형식 속성은 지정된 컴포넌트에 대해 사용할 수 있습니다.

날짜, 시간, 화폐 및 숫자 값에 대한 기본 서식 규칙은 시스템 로케일을 기준으로 합니다. 예를 들어, 미국의 기본 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 기본 표시 설정을 오버라이드할 수 있습니다. 또한 *OnGetText* 및 *OnSetText* 이벤트 핸들러를 작성하여 런타임 시 필드 컴포넌트에 대한 서식을 사용자 지정할 수 있습니다.

## 이벤트 처리

대부분의 컴포넌트처럼, 필드 컴포넌트도 연결된 이벤트가 있습니다. 메소드는 이벤트에 대한 핸들러로서 할당될 수 있습니다. 이들 핸들러를 작성하면, data-aware 컨트롤을 통해 필드에 입력된 데이터에 영향을 주는 이벤트의 발생에 응답하고 사용자가 의도한대로 디자인 동작을 수행할 수 있습니다. 다음 표는 필드 컴포넌트에 연결된 이벤트를 나열한 것입니다.

**표 17.5 필드 컴포넌트 이벤트**

| 이벤트               | 용도                                                   |
|-------------------|------------------------------------------------------|
| <i>OnChange</i>   | 필드값이 변경될 때 호출됩니다.                                    |
| <i>OnGetText</i>  | 필드 컴포넌트 값을 표시하거나 편집하기 위해 검색할 때 호출됩니다.                |
| <i>OnSetText</i>  | 필드 컴포넌트 값이 설정될 때 호출됩니다.                              |
| <i>OnValidate</i> | 편집이나 삽입 작업으로 인해 값이 변경될 때마다 필드 컴포넌트 값을 확인하기 위해 호출됩니다. |

*OnGetText* 및 *OnSetText* 이벤트는 주로 기본 제공 서식 함수보다 정교한 사용자 지정 서식을 원하는 프로그래머에게 유용합니다. *OnChange*는 메뉴나 비주얼 컨트롤의 사용 가능 또는 사용 불가능과 같은 데이터 변경에 연결된 애플리케이션별 작업을 수행하는 경우에 유용합니다. *OnValidate*는 데이터베이스 서버에 값을 반환하기 전에 애플리케이션에서 데이터 입력 검증을 조정하고자 하는 경우에 유용합니다.

다음과 같은 방법으로 필드 컴포넌트에 대한 이벤트 핸들러를 작성합니다.

- 1 컴포넌트를 선택합니다.
- 2 Object Inspector에서 Events 페이지를 선택합니다.
- 3 이벤트 핸들러의 Value 필드를 더블 클릭하여 소스 코드 창을 표시합니다.
- 4 핸들러 코드를 생성하거나 편집합니다.

## 런타임 시 필드 컴포넌트 메소드 사용

런타임 시 사용 가능한 필드 컴포넌트 메소드를 사용하면 한 데이터 타입에서 다른 데이터 타입으로 필드값을 변환할 수 있고, 필드 컴포넌트에 연결된 폼의 첫 번째 data-aware 컨트롤에 포커스를 설정할 수 있습니다.

필드에 연결된 data-aware 컴포넌트의 포커스를 조정하는 것은 애플리케이션이 *BeforePost*와 같은 데이터셋 이벤트 핸들러에서 레코드 지향 데이터 검증을 수행할 때 중요합니다. 검증은 연결된 data-aware 컨트롤이 포커스를 갖고 있는지 여부와 상관없이 레코드의 필드에서 수행될 수 있습니다. 레코드의 특정 필드에서 검증이 실패하면, 잘못된 데이터를 포함하는 data-aware 컨트롤에 포커스를 두어 수정 사항을 입력할 수 있습니다.

필드의 *FocusControl* 메소드를 사용하여 필드의 data-aware 컴포넌트에 대한 포커스를 조정합니다. *FocusControl*은 포커스를 필드에 연결된 폼의 첫 번째 data-aware 컨트롤로 설정합니다. 이벤트 핸들러는 필드를 검증하기 전에 필드의 *FocusControl* 메소드를 호출할 수 있습니다. 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 완전한 목록과 자세한 내용은 온라인 *VCL Reference*에서 *TField*와 자손에 대한 항목을 참조하십시오.

**표 17.6** 선택된 필드 컴포넌트 메소드

| 메소드         | 용도                                                          |
|-------------|-------------------------------------------------------------|
| AssignValue | 자동 변환 기능을 사용하여 필드의 타입에 따라 지정된 값으로 필드값을 설정합니다.               |
| Clear       | 필드를 해제하고 필드값을 Null로 설정합니다.                                  |
| GetData     | 필드에서 서식이 없는 데이터를 검색합니다.                                     |
| IsValidChar | 값을 설정하기 위해 data-aware 컨트롤에서 사용자가 입력한 문자가 이 필드에 허용되는지 결정합니다. |
| SetData     | 이 필드에 서식이 없는 데이터를 할당합니다.                                    |

## 필드값 표시, 변환 및 액세스

*TDBEdit* 및 *TDBGrid*와 같은 data-aware 컨트롤은 필드 컴포넌트에 연결된 값을 자동으로 표시합니다. 데이터셋과 컨트롤에서 편집이 가능하면, data-aware 컨트롤 또한 데이터베이스에 새로운 값과 변경된 값을 전송할 수 있습니다. 일반적으로 data-aware 컨트롤의 기본 제공 속성과 메소드를 사용하면 사용자가 추가 프로그래밍 없이도 데이터셋에 연결하고 값을 표시하며 업데이트를 할 수 있습니다. 데이터베이스 애플리케이션에서 사용 가능한 경우에는 언제든지 사용하십시오. data-aware 컨트롤에 대한 자세한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

표준 컨트롤 또한 필드 컴포넌트에 연결된 데이터베이스 값을 표시하고 편집할 수 있습니다. 그러나 표준 컨트롤을 사용하려면 사용자가 추가로 프로그래밍 해야 합니다. 예를 들어, 애플리케이션에서는 표준 컨트롤을 사용할 때 필드값이 변경되기 때문에 컨트롤을 업데이트할 시기를 추적해야 합니다. 데이터셋에 데이터소스 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 컨트롤의 사용 가능 및 사용 불가능 시기를 결정할 수 있습니다. 이들 이벤트에 대한 자세한 내용은 15-4 페이지의 "데이터 소스에서 변경 사항에 대한 응답"을 참조하십시오.

다음 항목에서는 표준 컨트롤에 표시하기 위해 필드값으로 작동하는 방법을 설명합니다.

### 표준 컨트롤의 필드 컴포넌트 값 표시

애플리케이션은 필드 컴포넌트의 *Value* 속성을 통해 데이터셋 열의 값을 액세스할 수 있습니다. 예를 들어, *CustomersCompany* 필드의 값이 변경될 수도 있기 때문에 다음과 같은 *OnDataChange* 이벤트 핸들러는 *TEdit* 컨트롤의 텍스트를 업데이트합니다.

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
    Edit3.Text := CustomersCompany.Value;
end;
```

이 메소드는 문자열 값에는 잘 수행되지만, 다른 데이터 타입에 대해서는 변환을 처리할 추가 프로그램이 필요할 수도 있습니다. 다행히도, 필드 컴포넌트에는 기본 제공된 변환 처리 기능이 있습니다.

**참고** 또한 가변 타입을 사용하여 필드값을 액세스하고 설정할 수 있습니다. 필드값 액세스 및 설정을 위한 가변 타입 사용에 대한 자세한 내용은 17-18 페이지의 "기본 데이터셋 속성으로 필드값 액세스"를 참조하십시오.



## 필드값 변환

변환 속성은 한 데이터 타입에서 다른 데이터 타입으로 변환하려고 시도합니다. 예를 들어, *AsString* 속성은 숫자 및 부울 값을 문자열 표시로 변환합니다. 다음 표는 필드 컴포넌트 변환 속성을 나열하고, 필드 컴포넌트 클래스에 의해 필드 컴포넌트에 권장되는 속성을 설명한 것입니다.

|                        | AsVariant | AsString | AsInteger | AsFloat<br>AsCurrency<br>AsBCD | AsDateTime<br>AsSQLTimeStamp | AsBoolean |
|------------------------|-----------|----------|-----------|--------------------------------|------------------------------|-----------|
| TStringField;          | 예         | 해당없음     | 예         | 예                              | 예                            | 예         |
| TWideStringField       | 예         | 예        | 예         | 예                              | 예                            | 예         |
| TIntegerField          | 예         | 예        | 해당없음      | 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                            |           |
| TSQL<br>TimeStampField | 예         | 예        |           | 예                              | 예                            |           |
| TBooleanField          | 예         | 예        |           |                                |                              |           |
| TBytesField            | 예         | 예        |           |                                |                              |           |
| TVarBytesField         | 예         | 예        |           |                                |                              |           |
| TBlobField             | 예         | 예        |           |                                |                              |           |
| TMemoField             | 예         | 예        |           |                                |                              |           |
| TGraphicField          | 예         | 예        |           |                                |                              |           |
| TVariantField          | 해당없음      | 예        | 예         | 예                              | 예                            | 예         |
| TAggregateField        | 예         | 예        |           |                                |                              |           |

테이블의 일부 열들은 하나 이상의 변환 속성 (예를 들어, *AsFloat*, *AsCurrency* 및 *AsBCD*)을 참조한다는 것에 유의하십시오. 이는 이들 속성 중 하나를 지원하는 모든 필드 데이터 타입이 항상 다른 속성도 함께 지원하기 때문입니다.

또한 *AsVariant* 속성은 모든 데이터 타입을 변환할 수 있다는 것에 유의하십시오. 위에 나열되지 않은 다른 데이터 타입에서 실제로는 유일한 옵션이라고 할 수 있는 *AsVariant*도 사용할 수 있습니다. 확실하지 않을 경우에는 *AsVariant*를 사용합니다.

어떤 경우에 대해서는 변환이 항상 가능하지는 않습니다. 예를 들어, *AsDateTime*은 문자열 값이 인식할 수 있는 날짜 시간 형식으로 되어 있는 경우에만 문자열이 날짜, 시간,

또는 날짜시간 형식으로 변환하는 데 사용될 수 있습니다. 변환 시도가 실패하면 예외가 발생합니다.

일부 다른 경우에, 변환은 가능하지만 변환의 결과가 항상 직관적이지는 않습니다. 예를 들어, *TDateTimeField* 값을 부동 소수점 형식으로 변환한다는 것은 *AsFloat*는 필드의 날짜 부분을 12/31/1899 이후의 일 수로 변환하고, 필드의 시간 부분은 24시간의 단위로 변환한다는 것을 의미합니다. 표 17.7은 특수한 결과를 보여주는 허용 가능한 변환을 나열한 것입니다.

**표 17.7 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 반올림합니다.                             |
| <i>DateTime</i> 또는 <i>SQLTimeStamp</i> 에서 부동 소수점으로 변환 | 날짜를 12/31/1899 이후의 일 수로 변환하고, 시간을 24시간의 단위로 변환합니다.          |
| 부울에서 문자열로 변환                                          | 모든 부울 값을 "True"나 "False"로 변환합니다.                            |

그 외의 경우에는 변환이 전혀 가능하지 않습니다. 이러한 경우에, 변환을 시도하면 예외가 발생합니다.

변환은 항상 실제로 할당되기 전에 발생합니다. 예를 들어, 다음 문장은 *CustomersCustNo* 값을 문자열로 변환하고 문자열을 편집 컨트롤의 텍스트에 할당합니다.

```
Edit1.Text := CustomersCustNo.AsString;
```

이와 반대로 다음 문장은 편집 컨트롤의 텍스트를 정수로 *CustomersCustNo* 필드에 할당합니다.

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

## 기본 데이터셋 속성으로 필드값 액세스

필드값 액세스를 위한 가장 일반적인 메소드는 *FieldValues* 속성에 가변 타입을 사용하는 것입니다. 예를 들어, 다음 문장은 편집 상자의 값을 *Customers* 테이블의 *CustNo* 필드에 둡니다.

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

*FieldValues* 속성은 가변 타입이므로, 다른 데이터 타입을 가변 값으로 자동 변환합니다.

가변 타입에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

## 데이터셋의 Fields 속성으로 필드값 액세스

필드가 속하는 데이터셋 컴포넌트의 *Fields* 속성으로 필드값을 액세스할 수 있습니다. *Fields*는 데이터셋의 모든 필드의 인덱스된 목록을 유지합니다. 필드값을 *Fields* 속성으로 액세스하면, 많은 열을 반복해야 할 경우나 애플리케이션에서 디자인 타임에는 사용할 수 없는 테이블 작업을 할 경우에 유용합니다.

*Fields* 속성을 사용하려면, 데이터셋의 필드의 순서와 데이터 타입을 알아야 합니다. 순서 번호를 사용하여 액세스할 필드를 지정합니다. 데이터셋의 첫 번째 필드는 번호가 0입니다. 필드값은 각 필드 컴포넌트의 변환 속성을 사용하여 적절하게 변환되어야 합니다. 필드 컴포넌트 변환 속성에 대한 자세한 내용은 17-17 페이지의 "필드값 변환"을 참조하십시오.

예를 들어, 다음 문장은 *Customers* 테이블의 일곱 번째 열 (Country)의 현재 값을 편집 컨트롤에 할당합니다.

```
Edit1.Text := CustTable.Fields[6].AsString;
```

이와 반대로, 데이터셋의 *Fields* 속성을 원하는 필드로 설정하여 값을 필드에 할당할 수 있습니다. 예를 들어,

```
begin
    Customers.Edit;
    Customers.Fields[6].AsString := Edit1.Text;
    Customers.Post;
end;
```

## 데이터셋의 FieldByName 메소드로 필드값 액세스

또한 데이터셋의 *FieldByName* 메소드로 필드값을 액세스할 수 있습니다. 이 메소드는 액세스하려는 필드 이름은 알지만 디자인 타임 시 원본으로 사용한 테이블에 액세스할 수 없을 경우에 유용합니다.

*FieldByName*을 사용하려면, 액세스하려는 데이터셋과 필드 이름을 알아야 합니다. 필드의 이름을 인수로서 메소드에 전달합니다. 필드값을 액세스하거나 변경하려면, *AsString*이나 *AsInteger*와 같은 적절한 필드 컴포넌트 변환 속성을 사용하여 결과를 변환합니다. 예를 들어, 다음 문장은 *Customers* 데이터셋의 *CustNo* 필드값을 편집 컨트롤에 할당합니다.

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

이와 반대로, 필드에 값을 할당할 수 있습니다.

```
begin
    Customers.Edit;
    Customers.FieldByName('CustNo').AsString := Edit2.Text;
    Customers.Post;
end;
```

## 필드의 현재 값 확인

애플리케이션이 클라이언트 데이터셋을 사용하여 데이터베이스 서버로부터 데이터를 업데이트하고, 레코드를 업데이트할 때 어려움이 있을 경우, *CurValue* 속성을 사용하여 문제를 일으키는 레코드의 필드값을 검사할 수 있습니다. *CurValue*는 데이터베이스 서버에서 필드의 현재 값을 나타내고, 다른 사용자가 데이터베이스를 변경한 모든 내용을 반영합니다.

*CurValue*는 *OnUpdateError* 또는 *OnReconcileError* 이벤트 핸들러 내부에서만 사용 가능합니다. 이들 이벤트 핸들러에서, *CurValue*(서버에 있는 값)를 *NewValue*(문제를 일으키는 포스트되지 않은 값) 및 *OldValue*(편집되기 전 필드에 원래 할당되었던 값)와 비교합니다. *OldValue*를 읽은 후 다른 사용자가 필드값을 변경한 경우에만, *CurValue*가 *OldValue*와 다릅니다.

## 필드에 대한 기본 값 설정

*DefaultExpression* 속성을 사용하여 런타임 시 클라이언트 데이터셋의 필드에 대한 기본 값을 계산해야 하는 방법을 지정할 수 있습니다. *DefaultExpression*은 필드 값을 참조하지 않는 유효한 SQL 값 표현식이 될 수 있습니다. 표현식에 숫자 값이 아닌 리터럴이 포함되는 경우에는 인용 부호로 나타냅니다. 예를 들어, 시간에 대한 정오의 기본 값은 다음과 같이 표시됩니다.

```
'12:00:00'
```

리터럴 값 주위에 인용 부호가 포함됩니다.

**참고** 원본으로 사용한 데이터베이스 테이블이 필드에 대한 기본 값을 정의하는 경우, 사용자가 *DefaultExpression*에 지정하는 기본 값이 우선권을 가집니다. 이는 클라이언트 데이터셋이 필드를 포함한 레코드를 포스트할 때 *DefaultExpression*이 적용되고 이것이 편집된 레코드가 데이터베이스 서버에 적용되기 전에 발생하기 때문입니다.

## 제약 조건 지정

대부분의 SQL 데이터베이스는 제약 조건을 사용하여 필드에 대한 가능한 값에 조건을 부여합니다. 예를 들어, 어떤 필드는 Null 값을 허용하지 않을 수도 있고, 값이 해당 열에 대해 고유해야 하거나 0보다는 크고 150보다 작아야 할 수도 있습니다. 이러한 제약 조건은 애플리케이션이 데이터베이스 서버에 업데이트를 적용할 때 시행됩니다.

이들 서버에 시행되는 제약 조건 외에도, 클라이언트 데이터셋의 필드에 지역적으로 적용되는 사용자 지정 제약 조건을 만들고 사용할 수 있습니다. 사용자 지정 제약 조건은 업데이트를 적용하려고 시도한 이후가 아니라 변경 로그에 편집 사항을 포스트할 때 즉시 오류를 감지하기 위해 서버 제약 조건을 복사하거나 애플리케이션에 정의된 제한 사항을 부여할 수 있습니다. 사용자 지정 제약 조건은 데이터 항목의 검증을 제공하지만 서버 애플리케이션과 주고 받은 데이터에 대해서는 적용될 수 없습니다.

사용자 지정 제약 조건을 만들려면, *CustomConstraint* 속성을 설정하여 제약 조건을 지정하고, 사용자가 런타임 시 제약 조건을 위반할 때 *ConstraintErrorMessage*를 표시하는 메시지로 설정합니다.

*CustomConstraint*는 필드값에 부여된 애플리케이션별 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.

$$x > 0 \text{ and } x < 100$$

필드값을 참조하는 데 사용되는 이름은 제약 조건 표현식에서 지속적으로 사용되는 한 예약된 SQL 키워드가 아닌 모든 문자열이 될 수 있습니다.

## 객체 필드 사용

객체 필드는 보다 단순한 다른 데이터 타입의 복합물을 나타내는 필드입니다. 여기에는 ADT(Abstract Data Type) 필드, 배열 필드, DataSet 필드 및 Reference 필드가 포함됩니다. 이러한 필드 타입은 모두 자식 필드나 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. 이들 자식 필드는 서로 타입이 다를 수 있습니다. 배열 필드에는 동일한 타입의 자식 필드 배열이 포함됩니다.

데이터셋 및 참조 필드는 다른 데이터셋을 액세스하는 필드에 매핑됩니다. 데이터셋 필드는 중첩된(디테일) 데이터셋에 액세스하고, 참조 필드는 포인터(참조)를 다른 영구적 객체(ADT)에 저장합니다.

**표 17.8 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 REF 필드를 나타냅니다.        |

Fields editor로 필드를 객체 필드가 있는 데이터셋에 추가할 때, 올바른 타입의 영구적 객체 필드가 자동으로 생성됩니다. 데이터셋에 영구적 객체 필드를 추가하면 데이터셋의 *ObjectView* 속성을 *True*로 자동으로 설정합니다. 이는 구성 자식 필드가 구분된 독립 필드인 것처럼 이들 필드를 같은 평평하게(flatten out) 하기 보다는 이들 필드를 계층적으로 저장하도록 데이터셋에 알립니다.

다음 속성들은 모든 객체 필드에 공통적이며, 자식 필드와 데이터셋을 처리할 수 있는 기능을 제공합니다.

**표 17.9 공통 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 필드값에 대한 액세스를 제공합니다.       |

## ADT 및 배열 필드 표시

ADT 및 배열 필드는 모두 data-aware 컨트롤을 통해 표시될 수 있는 자식 필드를 포함합니다.

단일 필드값을 표시하는 *TDBEdit*와 같은 data-aware 컨트롤은 쉼표로 구분된 편집할 수 없는 문자열에 자식 필드값을 표시합니다. 또한 컨트롤의 *DataField* 속성을 객체 필드 자체 대신에 자식 필드로 설정하면, 자식 필드는 다른 일반 데이터 필드와 같이 편집되어 보여질 수 있습니다.

*TDBGrid* 컨트롤은 데이터셋의 *ObjectView* 속성 값에 따라 ADT와 배열 필드를 다르게 표시합니다. *ObjectView*가 *False*이면, 각 자식 필드는 단일 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 필드 자체의 제목 표시줄 아래에 자신의 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 지정 타입이며, 레코드 타입과 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 필드값을 *CityEdit*라는 편집 상자에 할당하고 다음과 같은 ADT 구조를 사용하는 것에 대해 설명합니다.

```
Address
  Street
  City
  State
  Zip
```

## 영구적 필드 컴포넌트 사용

ADT 필드값에 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 이름으로 ADT 필드의 자식 필드에 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 필드 없이 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['Address.City'];
```

*FieldValues*는 데이터셋의 기본 속성이기 때문에 속성 이름 (*FieldValues*)을 생략할 수 있습니다.

**참고** ADT 자식 필드값에 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 필드값에 액세스할 수 있습니다. *FieldValues*는 *Variant*를 수용하고 반환하여, 모든 타입의 필드를 처리하고 변환할 수 있습니다. 인덱스 매개변수는 필드의 오프셋을 지정하는 정수 값입니다.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

*FieldValues*는 *TADTField*의 기본 속성이기 때문에 필드 이름 (*FieldValues*)을 생략할 수 있습니다. 따라서 다음 문장은 위의 문장과 동일합니다.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[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[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 아님(ADT) 수 있지만 배열에서는 배열 필드를 사용할 수 없습니다. *TDataSet*의 *SparseArrays* 속성은 배열 필드의 각 요소에 대해 고유한 *TField* 객체가 생성되는지를 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 사용

영구적 필드는 배열 필드의 배열 요소에 개별적으로 매핑할 수 있습니다. 예를 들어, 문자열의 6가지 요소 배열인 *TelNos\_Array* 배열 필드를 고려해 보십시오. *Customer* 테이블 컴포넌트에 생성된 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 편집 상자에 할당합니다.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

## 배열 필드의 FieldValues 속성 사용

배열 필드의 *FieldValues* 속성으로 자식 필드 값에 액세스할 수 있습니다. *FieldValues*는 *Variant*를 수용하고 반환하므로 모든 타입의 자식 필드를 처리하고 변환할 수 있습니다. 예를 들어,

```
TelEdit.Text := TArrayField(Customer.FieldName('TelNos_Array')).FieldValues[1];
```

*FieldValues*는 *TArrayField*의 기본 속성이기 때문에 또한 다음과 같이 작성할 수 있습니다.

```
TelEdit.Text := TArrayField(Customer.FieldName('TelNos_Array'))[1];
```

## 배열 필드의 Fields 속성 사용

*TArrayField*에는 개별 하위 필드를 액세스하는 데 사용할 수 있는 *Fields* 속성이 있습니다. 이것은 배열 필드(*OrderDates*)가 Null이 아닌 배열 요소로 목록 상자를 채우는 데 사용되는 다음 예제에서 설명됩니다.



```

for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;

```

## 데이터셋 필드 작업

데이터셋 필드는 중첩된 데이터셋에 저장된 데이터에 대한 액세스를 제공합니다. *NestedDataSet* 속성은 중첩된 데이터셋을 참조합니다. 그런 다음, 중첩된 데이터셋의 데이터가 중첩된 데이터셋의 필드를 통해 액세스됩니다.

## 데이터셋 필드 표시

*TDBGrid* 컨트롤을 사용하면 데이터셋 필드에 저장된 데이터를 표시할 수 있습니다. *TDBGrid* 컨트롤에서, 데이터셋 필드는 "(DataSet)"을 사용하여 데이터셋 열의 각 셀에 나타나고, 또한 런타임 시 생략 버튼이 오른쪽에 표시됩니다. 생략 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[7]);
```

## 중첩된 데이터셋의 데이터 액세스

데이터셋 필드는 일반적으로 data-aware 컨트롤에 직접 연결되지 않습니다. 그보다는, 중첩된 데이터셋은 다른 데이터셋이기 때문에 다른 데이터셋 컴포넌트를 사용하여 데이터에 액세스합니다. 사용하는 데이터셋의 타입은 부모 데이터셋(데이터셋 필드가 있는 데이터셋)에 의해 결정됩니다. 예를 들어, *TClientDataSet* 객체의 데이터셋 필드는 다른 *TClientDataSet*을 사용하여 그 값을 나타내야 합니다.

다음과 같은 방법으로 데이터셋 필드의 데이터에 액세스합니다.

- 1 부모 데이터셋용 Fields editor 를 호출하여 영구적 *TDataSetField* 객체를 만듭니다.
- 2 데이터셋 필드의 값을 나타내는 데이터셋을 생성합니다.
- 3 2 단계에서 만든 데이터셋의 *DataSetField* 속성을 1 단계에서 만든 영구적 데이터셋 필드로 설정합니다.

현재 레코드의 중첩된 데이터셋 필드에 값이 있는 경우에는 디테일 데이터셋 컴포넌트가 중첩된 데이터를 가진 레코드를 포함할 것이며, 그렇지 않을 경우에는 디테일 데이터셋이 비어 있을 것입니다.

## 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 중첩된 데이터셋으로 반환되지만, 또한 *TReferenceField*의 *Fields* 속성을 통해 액세스될 수도 있습니다.

### 참조 필드 표시

*TDBGrid* 컨트롤에서, 참조 필드는 런타임 시 오른쪽에 생략 버튼이 있는 데이터셋 열의 각 셀에 (Reference)가 지정됩니다. 런타임 시 생략 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[7]);
```

### 참조 필드의 데이터 액세스

중첩된 데이터셋에 액세스하는 것과 동일한 방법으로 참조 필드의 데이터에 액세스할 수 있습니다.

- 1 부모 데이터셋용 *Fields* editor를 호출하여 영구적 *TDataSetField* 객체를 만듭니다.
- 2 해당 데이터셋 필드의 값을 나타내는 데이터셋을 생성합니다.
- 3 2 단계에서 만든 데이터셋의 *DataSetField* 속성을 1 단계에서 만든 영구적 데이터셋 필드로 설정합니다.

참조가 할당되면, 참조 데이터셋은 참조된 데이터를 가진 단일 레코드를 갖습니다. 참조가 Null이면, 참조 데이터셋은 비게 됩니다.

또한 참조 필드의 *Fields* 속성을 사용하여 참조 필드의 데이터에 액세스할 수도 있습니다. 예를 들어, 다음 두 줄은 동일하게, 참조 필드 *CustomerRefCity*의 데이터를 *CityEdit*라고 하는 편집 상자에 할당합니다.

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;  
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

## 단방향 데이터셋 사용

단방향 데이터셋은 애플리케이션에서 SQL 데이터베이스 테이블로부터 데이터를 읽을 수 있는 메커니즘을 제공합니다. 단방향 데이터셋은 최소한의 오버헤드를 가지고 데이터베이스 정보에 대한 빠른 경량급 (lightweight) 액세스를 하도록 디자인되었습니다. 단방향 데이터셋은 SQL 명령을 데이터베이스 서버에 보내고, 명령이 레코드 집합을 반환하면, 이들 레코드의 액세스를 위한 단방향 커서가 생깁니다. 이것은 메모리에 데이터를 버퍼로 저장하지 않으며, 다른 타입의 데이터셋에 비해 실행 속도가 더 빠르고 리소스를 덜 차지합니다. 하지만 레코드가 버퍼로 저장되지 않기 때문에, 단방향 데이터셋은 다른 데이터셋 보다 유연하지 않습니다. *TDataSet*에 있는 많은 기능들이 단방향 데이터셋에서 구현되지 않거나 또는 예외를 발생시킵니다. 예를 들면, 다음과 같습니다.

- 지원되는 유일한 탐색 메소드는 *First* 와 *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 제공 지원이 없습니다. *CanModify* 속성이 항상 *False*를 표시하기 때문에, 데이터셋을 편집 모드로 두려고 해도 항상 실패하게 됩니다. 하지만, 이 속성을 사용하면, SQL UPDATE 명령을 사용하여 데이터를 업데이트하거나, 14-9 페이지의 "클라이언트 데이터셋을 사용하여 레코드 버퍼링"에 설명된대로 클라이언트 데이터셋에 연결하여 이전의 편집 지원을 제공할 수 있습니다.
- 필터는 버퍼를 필요로 하는 다중 레코드와 함께 사용해야 하기 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 값을 포함해서 여러 레코드를 보유하기 위한 버퍼가 필요하기 때문에 조회 필드에 대한 지원이 없습니다. 단방향 데이터셋에서 조회 필드를 정의하는 경우 아무런 적절한 기능도 하지 않습니다.

이러한 제한 사항에도 불구하고, 단방향 데이터셋은 데이터를 액세스하는 확실한 방법입니다. 단방향 데이터셋은 빠르고, 사용하고 배포하기에 매우 간단합니다.

## 단방향 데이터셋의 타입

컴포넌트 팔레트의 *dbExpress* 페이지에는 *TSQLDataSet*, *TSQLQuery*, *TSQLTable* 및 *TSQLStoredProc* 같은 네 가지 타입의 단방향 데이터셋이 들어 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 사용할 것을 권장합니다.

*TSQLDataSet*으로 수행할 수 있는 거의 모든 작업을 *TSQLQuery*에서 수행할 수 있습니다. *TSQLQuery*는 내장 프로시저에서 사용될 수 없다는 점이 기본적으로 다릅니다 (많은 서버들은 내장 프로시저를 실행하기 위해 쿼리를 작성할 수 있게 하는 SQL에 대한 확장을 지원하지만 이를 위한 표준 구문은 존재하지 않습니다). *TSQLQuery*는 Windows 애플리케이션들과 호환되므로, 쿼리 컴포넌트를 사용하는 애플리케이션을 Linux로 이식하기가 더 쉽습니다.

*TSQLTable*은 단일 데이터베이스 테이블에 모든 필드와 모든 레코드를 나타내려는 경우의 특수한 목적을 위한 데이터셋입니다. *TSQLTable*은 Windows 애플리케이션들과 호환되므로 테이블 컴포넌트를 사용하는 애플리케이션을 Linux로 이식하기가 더 쉽습니다.

*TSQLStoredProc*는 내장 프로시저 실행을 위한 특수한 목적의 데이터셋입니다. 또한 *TSQLDataSet*를 사용해서 내장 프로시저를 실행할 수도 있습니다. *TSQLStoredProc*는 Windows 애플리케이션들과 호환되므로, 내장 프로시저 컴포넌트를 사용하는 애플리케이션을 Linux로 이식하기가 더 쉽습니다.

## 서버에 연결

단방향 데이터셋을 사용하는 첫 번째 단계는 이를 데이터베이스 서버에 연결하는 것입니다. 디자인 타임에 데이터셋의 데이터베이스 서버에 대한 연결이 활성화되면, Object Inspector는 다른 속성에 대한 값의 드롭다운 목록을 제공할 수 있습니다. 예를 들어, 내장 프로시저를 나타낼 때는 연결이 활성화되어야 Object Inspector가 서버에서 사용할 수 있는 내장 프로시저의 목록을 나타낼 수 있습니다.

데이터베이스 서버에 대한 연결은 개별 *TSQLConnection* 컴포넌트에 의해 나타납니다. *TSQLConnection*은 데이터베이스 서버와 여러 연결 매개변수를 식별합니다(서버에서 사용하는 데이터베이스, 서버를 실행 중인 시스템의 호스트 이름, 사용자 이름, 암호 등 포함). *TSQLConnection*을 사용한 연결 설정에 대한 자세한 내용은 19장 "데이터베이스에 연결"을 참조하십시오.

단방향 데이터셋을 연결하려면, 연결을 형성하는 *TSQLConnection*을 지정해야 합니다. *SQLConnection* 속성을 사용하여 지정합니다. 디자인 타임에 Object Inspector의 드롭다운 목록에서 SQL 연결 컴포넌트를 선택할 수도 있습니다. 런타임에 이러한 할당을 지정하는 경우에는 연결이 활성화되었는지를 확인하십시오.

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

일반적으로 애플리케이션의 모든 단방향 데이터셋은 여러 문장을 지원하지 않는 MySQL 같은 하나의 데이터베이스 서버나 여러 데이터베이스 서버의 데이터를 사용하는 경우가 아니라면 동일한 연결 컴포넌트를 공유합니다.

## 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 프로시저중 어디에서 가져 오는지에 따라 방법이 다릅니다.

*TSQLDataSet* 컴포넌트를 사용할 때, *CommandType* 속성을 사용하여 데이터셋이 데이터를 얻는 위치를 지정합니다. *CommandType*은 다음 중 어떤 값이라도 취할 수 있습니다.

- *ctQuery*: *CommandType*이 *ctQuery*이면, *TSQLDataSet*은 사용자가 지정하는 쿼리를 실행합니다. 쿼리가 SELECT 명령이면 데이터셋은 레코드의 결과 집합을 포함합니다.
- *ctTable*: *CommandType*이 *ctTable*이면 *TSQLDataSet*은 지정된 테이블로부터의 모든 레코드를 검색합니다.
- *ctStoredProc*: *CommandType*이 *ctStoredProc*이면, *TSQLDataSet*은 내장 프로시저를 실행합니다. 내장 프로시저가 커서를 반환하면 데이터셋은 반환된 레코드를 포함합니다.

**참고** 서버에서 이용할 수 있는 메타데이터가 단방향 데이터셋에 있을 수도 있습니다. 이러한 방법에 대한 자세한 내용은 18-16 페이지의 "스키마 정보 액세스"를 참조하십시오.

## 쿼리의 결과 표시

쿼리 사용은 레코드 집합을 지정하는 가장 일반적인 방법입니다. 쿼리는 단순히 SQL로 작성된 명령입니다. 레코드의 집합을 반환할 필요는 없으며, SQL은 서버에서 작업을 수행하는 UPDATE 쿼리 같은 쿼리를 정의합니다. 레코드를 반환하지 않는 쿼리에 대한 자세한 내용은 18-11 페이지의 "레코드를 반환하지 않는 명령 실행"에서 설명합니다.

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,

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

쿼리 결과를 나타내기 위해 *TSQLDataSet* 또는 *TSQLQuery*를 사용할 수 있습니다.

## TSQDataSet 을 사용하여 쿼리 지정

*TSQDataSet*을 사용할 경우 쿼리 문장을 *CommandText* 속성에 할당합니다.

```
SQLDataSet1.CommandText := 'SELECT CustName, Address FROM Customer';
```

디자인 타임에 Object Inspector에 직접 쿼리를 입력할 수 있습니다. 또한, SQL 데이터셋의 데이터베이스에 대한 연결이 이미 활성화된 경우에는 *CommandText* 속성의 생략 버튼을 클릭해서 Command Text 편집기를 표시할 수 있습니다. Command Text 편집기는 사용 가능한 테이블과 이들 테이블의 필드를 나열하여 쿼리를 쉽게 작성할 수 있도록 해줍니다.

## TSQLQuery 를 사용하여 쿼리 지정

*TSQLQuery*를 사용할 경우 쿼리를 *SQL* 속성에 할당합니다. 문자열인 *TSQDataSet*의 *CommandText* 속성과 달리, *SQL* 속성은 *TStrings* 객체입니다. 이 *TStrings* 객체의 분리된 각 문자열은 쿼리의 분리된 행입니다. 여러 행을 사용하는 것은 쿼리가 서버에서 실행되는 방법에는 영향을 주지 않지만, 여러 소스로부터 쿼리가 더 쉽게 조합할 수 있습니다.

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('SELECT ' + Edit1.Text + ' FROM ' + Edit2.Text);
if Length(Edit3.Text) <> 0 then
    SQLQuery1.SQL.Add('ORDER BY ' + Edit3.Text)
```

디자인 타임에 String List 편집기를 사용하여 쿼리를 지정합니다. Object Inspector의 *SQL* 속성의 생략 버튼을 클릭해서 String List 편집기를 표시합니다.

*TSQLQuery*를 사용할 때의 이점 중 하나는 *SQL* 속성이 *TStrings* 객체이기 때문에 *TStrings.LoadFromFile* 메소드를 호출하여 파일에서 쿼리의 텍스트를 로드할 수 있다는 것입니다.

```
SQLQuery1.SQL.LoadFromFile('/usr/queries/custquery.sql');
```

## 쿼리의 매개변수 사용

항목에서는 런타임 시 쿼리를 동적으로 만드는 방법을 설명했습니다. 하지만 디자인 타임에 작성된 쿼리에 대해서도 매개변수를 사용하여 동일한 작업을 수행할 수 있습니다.

매개변수화된 SQL 문은 매개변수, 변수, 디자인 타임이나 런타임에 다룰 수 있는 매개변수의 값을 포함합니다. 매개변수는 SQL 문에 나타나는 데이터 값을 대체할 수 있습니다. 예를 들면, 다음과 같은 SELECT 문에서 매개변수는 선택 기준을 지정하기 위해 사용됩니다.

```
SELECT CustNo, OrderNo, SaleDate FROM Orders
WHERE CustNo = :CustNumber
```

위의 SQL 문에서 *:CustNumber*는 애플리케이션이 런타임 시 문장에 제공하는 실제 값에 대한 위치 표시자입니다. *:CustNumber*처럼 이름이 콜론으로 시작하는 것에 유의하십시오. 매개변수 이름은 콜론(:)으로 시작해야 합니다. 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. 데이터셋은 *Params* 속성을 사용하여 이 값들을 저장합니다. *Params*는 각각의 *TParam* 객체가 하나의 매개변수를 나타내는 *TParam* 객체의 컬렉션입니다. *CommandText* 또는 *SQL* 속성을 사용해서 쿼리에 대한 텍스트를 지정할 때, 데이터셋은 이 *TParam* 객체 집합을 생성하고 쿼리로부터 추론할 수 있는 모든 속성을 초기화합니다.

**참고** *ParamCheck* 속성을 *False*로 설정함으로써, 쿼리 텍스트의 변경에 응답하는 *TParam* 객체의 자동 생성을 억제할 수 있습니다. 이에 관한 유용한 예제를 보려면 18-12 페이지의 "서버 메타데이터 작성 및 수정"을 참조하십시오.

**참고** 데이터셋이 데이터소스를 사용하여 다른 데이터셋에서 매개변수 값을 얻는 경우에는 매개변수 값을 명시적으로 할당할 필요가 없습니다. 이 프로세스는 18-14 페이지의 "마스터/디테일 관계 설정"에서 다룹니다.

## 디자인 타임에 매개변수 설정

디자인 타임에 매개변수 컬렉션 편집기를 사용하여 매개변수 값을 지정할 수 있습니다. 매개변수 컬렉션 편집기를 표시하려면, Object Inspector에서 *Params* 속성에 대한 생략 버튼을 클릭합니다. SQL 문에 매개변수가 없으면, 컬렉션 편집기에 *TParam* 객체가 나열되지 않습니다.

**참고** 매개변수 컬렉션 편집기는 다른 컬렉션 속성에 나타나는 것과 동일한 컬렉션 편집기입니다. 편집기가 다른 속성과 공유되기 때문에, 오른쪽 버튼을 클릭해서 표시되는 컨텍스트 메뉴에는 Add 및 Delete 명령이 포함됩니다. 하지만 이 명령들을 데이터셋 매개변수에 대해서는 사용이 불가능합니다. 매개변수를 추가하거나 삭제할 수 있는 유일한 방법은 SQL 문 자체에 있습니다.

각 매개변수는 매개변수 컬렉션 편집기에서 선택합니다. 그런 다음 Object Inspector를 사용하여 속성을 수정합니다.

- *DataType* 속성은 매개변수 값에 대한 데이터 타입을 나열합니다. 매개변수 값은 데이터셋이 쿼리로부터 값 타입을 추론할 수 있을 경우에 올바르게 초기화될 수 있습니다. 데이터셋이 타입을 추론할 수 없을 경우 *DataType*은 *ftUnknown*이며 이를 변경하여 매개변수 값의 타입을 나타내야 합니다.
- *ParamType* 속성은 선택된 매개변수의 타입을 나열합니다. 쿼리의 경우에는, 입력 매개변수만 포함할 수 있기 때문에 항상 *ptInput*으로 초기화됩니다.
- *Value* 속성은 선택된 매개변수에 대한 값을 지정합니다. 애플리케이션이 런타임 시 매개변수 값을 제공하는 경우에는 *Value*를 공백으로 둘 수 있습니다.

## 런타임 시 매개변수 설정

런타임에 매개변수를 작성하기 위해 다음을 사용할 수 있습니다.

- 이름을 기반으로 매개변수에 값을 할당하기 위한 *ParamByName* 메소드
- SQL 문에서의 매개변수 순서 위치를 기반으로 매개변수에 값을 할당하기 위한 *Params* 속성
- *Params.ParamValues* 속성은 설정된 각 매개변수 집합의 이름을 기반으로 단일 명령 행의 하나 이상의 매개변수에 대한 값을 할당합니다.

다음 코드에서는 *ParamByName*을 사용하여 :Capital 매개변수에 편집 상자의 텍스트를 할당합니다.

```
SQLDataSet1.ParamByName('Capital').AsString := Edit1.Text;
```

인덱스를 0으로 하여 (:Capital 매개변수가 SQL 문의 첫 번째 매개변수라고 가정) *Params* 속성을 사용하는 동일한 코드를 다시 작성할 수 있습니다.

```
SQLDataSet1.Params[0].AsString := Edit1.Text;
```

아래의 명령 행은 *Params.ParamValues* 속성을 사용하여 한 번에 세 개의 매개변수를 설정합니다.

```
SQLDataSet1.Params.ParamValues['Name;Capital;Continent'] :=  
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

*ParamValues*가 값을 변환하지 않아도 되는 Variants를 사용한다는 점에 유의하십시오.

## 테이블의 레코드 표시

하나의 기본 데이터베이스 테이블에 모든 필드와 모든 레코드를 표시하려는 경우, SQL 문을 사용자 스스로 작성하는 것보다 *TSQLDataSet* 또는 *TSQLTable*을 사용하여 쿼리를 작성할 수 있습니다.

**참고** 서버 성능이 문제가 될 경우에는, 자동 생성되는 쿼리에 의존하기보다 명시적으로 쿼리를 작성할 수도 있습니다. 자동으로 생성되는 쿼리는 테이블에 있는 필드를 명시적으로 나열하기보다는 대표 문자(와일드카드)를 사용합니다. 그러면 서버 성능에 약간의 저하를 초래합니다. 자동으로 생성되는 쿼리의 대표 문자(\*)는 서버의 필드 변경 사항에 대해 좀 더 강력합니다.

## TSQLDataSet을 사용하여 테이블 표시

*TSQLDataSet*이 단일 데이터베이스 테이블의 모든 필드와 모든 레코드를 페치하는 쿼리를 생성하도록 하려면 *CommandType* 속성을 *ctTable*로 설정합니다.

*CommandType*이 *ctTable*이면, *TSQLDataSet*은 두 가지 속성 값을 기반으로 쿼리를 생성합니다.

- *CommandText*는 *TSQLDataSet* 객체가 나타낼 데이터베이스 테이블의 이름을 지정합니다.
- *SortFieldNames*는 데이터 정렬을 위해 사용되는 필드 이름을 중요도 순으로 나열합니다.

예를 들어, 다음과 같이 지정한다고 가정해 보십시오.

```
SQLDataSet1.CommandType := ctTable;  
SQLDataSet1.CommandText := 'Employee';  
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

*TSQLDataSet*은 Employee 테이블에서 HireDate로 정렬하고 그 안에서 Salary로 정렬하여 모든 레코드를 나열하는 다음과 같은 쿼리를 생성합니다.

```
select * from Employee order by HireDate, Salary
```



## TSQTable 을 사용하여 테이블 표시

*TSQTable*을 사용할 때에는, *TableName* 속성을 사용하여 원하는 테이블을 지정합니다.

데이터셋의 필드 순서를 지정하려면 인덱스를 지정해야 합니다. 이를 수행하는 방법은 두 가지가 있습니다.

- *IndexName* 속성을 원하는 순서를 부여하는 서버에 정의된 인덱스의 이름으로 설정합니다.
- *IndexFieldNames* 속성을 정렬할 필드 이름의 세미콜론으로 구분한 목록으로 설정합니다. *IndexFieldNames*는 쉼표가 아닌 세미콜론을 구분자로 사용한다는 점만 제외하면, *TSQDataSet*의 *SortFieldNames* 속성과 동일한 기능을 합니다.

## 내장 프로시저의 결과 표시

내장 프로시저는 SQL 서버에 이름이 지정되고 저장된 SQL 문의 집합입니다. 일반적으로 내장 프로시저는 자주 반복되는 데이터베이스 관련 작업을 처리합니다. 내장 프로시저는 특히 집계 또는 산술 기능을 사용하는 작업이나 데이터베이스 테이블의 많은 행에서 수행되는 작업에 유용합니다. 내장 프로시저를 사용할 경우, 다음과 같은 이유에서 데이터베이스 애플리케이션의 성능이 향상됩니다.

- 서버의 높은 처리 능력과 속도를 이용합니다.
- 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 결정하십시오.

실행하려는 내장 프로시저를 나타내는 방법은 사용하고 있는 단방향 데이터셋의 타입에 따라 다릅니다.

## TSQDataSet 을 사용하여 내장 프로시저 지정

*TSQDataSet*을 사용할 때에는, 다음과 같은 방법으로 내장 프로시저를 지정합니다.

- *CommandType* 속성을 *ctStoredProc*로 설정합니다.
- *CommandText* 속성의 값으로서 내장 프로시저의 이름을 지정합니다.

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

## TSQLStoredProc 를 사용하여 내장 프로시저 지정

*TSQLStoredProc*를 사용할 때에는, *StoredProcName* 속성값으로서 내장 프로시저의 이름만 지정하면 됩니다.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

## 내장 프로시저 매개변수 작업

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 프로시저가 특정 타입의 매개변수를 사용할지는 데이터베이스 서버에 있는 내장 프로시저의 일반적인 랭귀지 구현과 내장 프로시저의 특정 인스턴스에 따라 달라집니다. 서버에서는 각각의 내장 프로시저가 입력 매개변수를 사용할 수도 있고 사용하지 않을 수도 있습니다. 반면에 일부 매개변수는 서버마다 다르게 사용됩니다. 예를 들어, 내장 프로시저의 *InterBase* 구현은 결과 매개변수를 반환하지 않습니다.

내장 프로시저 매개변수에 대한 액세스는 *Params* 속성의 *TParam* 객체에 의해 제공됩니다. 쿼리 매개변수에서와 같이, *TSQLDataSet* 및 *TSQLStoredProc*는 *CommandText(TSQLDataSet)* 또는 *StoredProcName(TSQLStoredProc)* 속성을 설정할 때 매개변수마다 *TParam* 객체를 자동으로 생성합니다.

## 디자인 타임 시 매개변수 설정

쿼리 매개변수에서와 같이, 디자인 타임에 매개변수 컬렉션 편집기를 사용하여 내장 프로시저 매개변수 값을 지정할 수 있습니다. 매개변수 컬렉션 편집기를 표시하려면, Object Inspector에서 *Params* 속성의 생략 버튼을 클릭합니다.

**중요** 매개변수 컬렉션 편집기에서 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 편집기를 사용하여 매개변수를 직접 설정해야 합니다. 오른쪽 버튼을 클릭하고 Add를 선택하여 매개변수를 추가합니다. 추가하는 각 매개변수에 대해 다음 작업을 수행합니다.

- *Name* 속성의 값으로 서버의 프로시저에 정의되어 있는 이름을 할당합니다.
- *ParamType* 속성을 설정하여 입력, 출력, 입/출력 또는 결과 매개변수인지를 나타냅니다.
- *DataType* 속성을 설정하여 매개변수 값의 데이터 타입을 나타냅니다.

일부 서버는 매개변수 이름을 제공하지만, 모든 매개변수 정보를 반환하지는 않습니다. 각 매개변수에 대해 *Data Type* 및 *Param Type* 속성을 확인하고 *ftUnknown* 또는 *ptUnknown*로 설정된 속성을 수정합니다.

**참고** 출력 매개변수와 결과 매개변수에 대한 값은 설정할 수 없습니다. 이러한 타입의 매개변수는 내장 프로시저 실행에 의해 설정된 값을 가집니다.

## 런타임 시 매개변수 사용

런타임에 개별 매개변수 객체에 액세스하여 입력 매개변수에 값을 할당하고 출력 매개변수에서 값을 검색할 수 있습니다. 쿼리 매개변수에서와 같이 *ParamByName* 메소드를 사용하여 이름을 기반으로 개별 매개변수에 액세스할 수 있습니다. 예를 들어, 다음 코드는 입/출력 매개변수의 값을 설정하고, 내장 프로시저를 실행하고, 반환된 값을 검색합니다.

```
with SQLDataSet1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

일부 서버는 매개변수 이름이나 데이터 타입을 보고하지 않기 때문에, 런타임 전에 내장 프로시저의 이름을 지정하지 않을 경우에는 코드에서 매개변수를 설정해야 합니다. 이 작업이 필요한지를 확인하려면, 디자인 타임에 대상 데이터베이스의 내장 프로시저를 지정하고, 매개변수 컬렉션 편집기를 사용하여 데이터셋이 자동으로 매개변수를 설정하는지를 확인합니다.

다음 예제는 매개변수가 자동으로 제공되지 않을 경우, 런타임 시 매개변수를 설정하는 방법을 보여 줍니다.

```
var
  P1, P2:Tparam;
begin
  with SQLDataSet1 do
  begin
    CommandType := ctStoredProc;
    CommandText := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[0].DataType := ftSmallint;
      Params[1].Name := 'PROJ_ID';
      Params[1].DataType := ftString;
      ParamByName('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByName('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
end;
```

## 데이터 페치

데이터의 소스를 지정한 다음에는, 데이터를 페치해야만 애플리케이션에서 액세스할 수 있습니다. 데이터셋이 데이터를 페치하고 나면, 데이터 소스를 통해 데이터셋에 연결된 `data-aware` 컨트롤이 자동으로 데이터 값을 표시하고, 프로바이더를 통해 데이터셋에 연결된 클라이언트 데이터셋을 레코드로 채울 수 있습니다.

다른 데이터셋에서와 같이, 단방향 데이터셋에 대해 데이터를 페치하는 방법은 두 가지가 있습니다.

- 디자인 타임 시 Object Inspector에서 또는 런타임 시 코드에서 *Active* 속성을 *True*로 설정합니다.

```
CustQuery.Active := True;
```

- 런타임 시 다음과 같이 *Open* 메소드를 호출합니다.

```
CustQuery.Open;
```

서버에서 레코드를 얻는 단방향 데이터셋과 함께 *Active* 속성이나 *Open* 메소드를 사용합니다. 이러한 레코드들이 *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* 및 서버가 문장 및 매개변수에 대한 리소스를 할당하는 것을 의미합니다. *CommandType*이 *ctTable*인 경우, 이는 데이터셋이 *SELECT* 쿼리를 생성할 때입니다. 서버에 의해 바인딩되지 않은 모든 매개변수는 이 시점에서 쿼리에 포함됩니다.

단방향 데이터셋은 *Active*를 *True*로 설정하거나 *Open* 메소드를 호출할 때 자동으로 준비됩니다. 데이터셋을 닫으면, 문장 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, *Prepared* 속성을 *True*로 설정합니다.

```
CustQuery.Prepared := True;
```

데이터셋을 명시적으로 준비하면, *Prepared*를 *False*로 설정하기 전에는 구문 실행을 위해 할당된 리소스가 해제되지 않습니다.

데이터셋이 실행되기 전에 다시 준비되는지 확인하려면 (예를 들어, 매개변수 값이나 *SortFieldNames* 속성을 변경하는 경우), *Prepared* 속성을 *False*로 설정합니다.

## 여러 데이터셋 폐치

일부 내장 프로시저는 레코드의 여러 집합을 반환합니다. 데이터셋은 사용자가 이를 열 때 첫 번째 집합만 폐치합니다. 다른 레코드 집합에 액세스하려면 *NextRecordSet* 메소드를 호출합니다.

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

*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, 이는 더 이상 남은 레코드 집합이 없을 때까지 반복됩니다. 남은 데이터셋이 없으면, *NextRecordSet*은 **nil**을 반환합니다.

## 레코드를 반환하지 않는 명령 실행

단방향 데이터셋이 나타내는 쿼리 또는 내장 프로시저가 어떤 레코드도 반환하지 않는 경우에도 단방향 데이터셋을 사용할 수 있습니다. 이러한 명령은 SELECT 문(예를 들면 INSERT, DELETE, UPDATE, CREATE INDEX 및 ALTER TABLE 명령은 어떤 레코드도 반환하지 않습니다)이 아닌 DDL (Data Definition Language) 또는 DML (Data Manipulation Language) 문을 사용하는 문장을 포함합니다. 명령에 사용되는 랭귀지는 서버 특정적이지만, 일반적으로 SQL 랭귀지의 SQL-92 표준과 호환됩니다.

실행하는 SQL 명령은 사용 중인 서버에서 수용 가능해야 합니다. 단방향 데이터셋은 SQL을 평가하지 않고 이를 실행하지도 않습니다. 단방향 데이터셋은 실행을 위해 서버에 명령을 전달할 뿐입니다.

**참고** 명령이 어떠한 레코드도 반환하지 않는 경우에는 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 19-12 페이지의 "서버에 명령 전송"을 참조하십시오.

## 실행할 명령 지정

---

단방향 데이터셋에서 실행할 명령을 지정하는 방법은 명령이 데이터셋의 결과이든 아니든 간에 동일합니다. 즉, 다음과 같습니다.

*TSQIDataSet*을 사용할 경우, *CommandType* 및 *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 경우, *SQL* 속성을 사용하여 서버에 전달할 SQL 문을 지정합니다.

*TSQLStoredProc*를 사용할 경우, *StoredProcName* 속성을 사용하여 실행할 내장 프로시저의 이름을 지정합니다.

레코드를 검색하는 것과 동일한 방법으로 명령을 지정할 때처럼, 레코드를 반환하는 쿼리 및 내장 프로시저에서와 같이 쿼리 매개변수 또는 내장 프로시저 매개변수를 사용합니다. 자세한 내용은 18-4 페이지의 "쿼리의 매개변수 사용"과 18-8 페이지의 "내장 프로시저 매개변수 작업"을 참조하십시오.

## 명령 실행

---

레코드를 반환하지 않는 쿼리나 내장 프로시저를 실행하려는 경우에는 *Active* 속성이거나 *Open* 메소드를 사용하지 않습니다. 그 대신, 다음을 사용해야 합니다.

- 데이터셋이 *TSQIDataSet* 또는 *TSQLQuery*의 인스턴스인 경우 *ExecSQL* 메소드를 사용해야 합니다.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';  
FixTicket.ExecSQL;
```

- 데이터셋이 *TSQLStoredProc*의 인스턴스인 경우 *ExecProc* 메소드를 사용해야 합니다.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';  
SQLStoredProc1.ExecProc;
```

**팁** 쿼리나 내장 프로시저를 여러번 실행 중인 경우에는 *Prepared* 속성을 *True*로 설정하는 것이 좋습니다.

## 서버 메타데이터 작성 및 수정

---

데이터를 반환하지 않는 대부분의 명령은 두 가지로 분류할 수 있습니다. 하나는 데이터를 편집하기 위해 사용하는 명령 (예: INSERT, DELETE 및 UPDATE 명령) 과 다른 하나는 테이블, 인덱스 및 내장 프로시저 같은 서버의 항목을 작성하거나 수정하기 위해 사용하는 명령입니다.

편집을 위해 SQL 명령을 명시적으로 사용하는 것을 원하지 않는 경우에는, 단방향 데이터셋을 클라이언트 데이터셋에 연결하고 편집과 관련하여 생성된 모든 SQL 명령을 처리하게 할 수 있습니다(14-9 페이지의 "클라이언트 데이터셋을 사용하여 레코드 버퍼링" 참조). 사실, data-aware 컨트롤은 클라이언트 데이터셋이나 편집이 가능한 사용자 지정 데이터셋을 통해 편집을 수행하도록 디자인되었기 때문에 위의 방법을 권장합니다.

애플리케이션이 서버의 메타데이터를 작성 또는 수정할 수 있는 유일한 방법은 사실상 명령을 전송하는 것입니다. 모든 데이터베이스 드라이버가 동일한 SQL 구문을 지원하지는 않습니다. 각 데이터베이스 타입에 의해 지원되는 SQL 구문과 데이터베이스 타입 사이의 차이점에 대해서는 여기에서 설명할 수 없습니다. 특정 데이터베이스 시스템의 SQL 구현에 대한 포괄적인 최신 정보는 해당 시스템에서 제공하는 설명서를 참조하십시오.

일반적으로 데이터베이스에 테이블을 작성하기 위해서 CREATE TABLE 문을 사용하고, 이들 테이블에 대한 새로운 인덱스를 작성하기 위해서 CREATE INDEX 문을 사용합니다. 지원이 가능한 경우에는, 여러 메타데이터 객체 추가를 위해 CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, 및 CREATE PROCEDURE 같은 다른 CREATE 문을 사용합니다.

각 CREATE 문마다 메타데이터 객체를 삭제하는 데 이용하는 DROP 문이 있습니다. 이러한 문장에는 DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, 및 DROP PROCEDURE가 포함됩니다.

테이블의 구조를 변경하려면 ALTER TABLE 문을 사용합니다. ALTER TABLE은 테이블에 새로운 요소를 작성하고 이를 삭제할 수 있는 ADD 및 DROP 절이 있습니다. 예를 들어, 테이블에 새로운 열을 추가하려면 ADD COLUMN 절을 사용하고, 테이블에서 기존의 제약 조건을 삭제하려면 DROP CONSTRAINT 절을 사용합니다.

예를 들면, 다음 문장은 InterBase 데이터베이스에 GET\_EMP\_PROJ라는 내장 프로시저를 작성합니다.

```
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 *TSQIDataSet*를 사용하여 이 내장 프로시저를 작성합니다. 데이터셋이 내장 프로시저 정의(:EMP\_NO 및 :PROJ\_ID)의 매개변수를 내장 프로시저를 작성하는 쿼리의 매개변수와 혼동하지 않도록 방지하기 위해 *ParamCheck* 속성을 사용한다는 점을 유의 하십시오.

```
with SQLDataSet1 do
begin
    ParamCheck := False;
```

```
CommandType := ctQuery;
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';
ExecSQL;
end;
```

## 마스터/디테일 관계 설정

디테일 셋으로서 단방향 데이터셋을 사용하는 마스터/디테일 관계를 설정하는 방법에는 두 가지가 있습니다. 사용할 방법은 사용하고 있는 단방향 데이터셋의 타입에 의존합니다. 마스터/디테일 관계를 설정하고 나면 단방향 데이터셋(일대다 관계의 "다")은 마스터 셋(일대다 관계의 "일")의 현재 레코드에 해당하는 레코드에 대해서만 액세스를 제공합니다.

### TSQIDataSet 또는 TSQLQuery 로 마스터 / 디테일 관계 설정

디테일 셋이 *TSQIDataSet* 또는 *TSQLQuery*의 인스턴스인 경우 마스터/디테일 관계를 설정하려면 매개변수를 사용하는 쿼리를 지정해야 합니다. 이들 매개변수는 마스터 데이터셋의 현재 필드 값을 참조합니다. 마스터 데이터셋의 현재 필드 값은 런타임에 동적으로 변경되기 때문에, 마스터 레코드가 변경될 때마다 디테일 셋의 매개변수를 다시 바인딩되어야 합니다. 이벤트 핸들러를 사용하여 이러한 작업을 수행할 코드를 작성할 수는 있지만, *TSQIDataSet* 및 *TSQLQuery*를 이용하면 *DataSource* 속성을 사용하여 보다 쉬운 메커니즘을 제공할 수 있습니다.

매개변수화된 쿼리의 매개변수 값이 디자인 타임에 바인딩되지 않거나 런타임에 지정되지 않을 경우, *TSQIDataSet* 및 *TSQLQuery*는 *DataSource* 속성에 따라 이에 대한 값을 제공하려고 시도합니다. *DataSource*는 바인딩되지 않은 매개변수의 이름과 일치하는 필드 이름으로 검색된 다른 데이터셋을 식별합니다. 이 검색 데이터셋은 모든 타입의 데이터셋이 될 수 있으며 또 다른 단방향 데이터셋일 필요는 없습니다. 검색 데이터셋은 이를 사용하는 디테일 데이터셋을 작성하기 전에 작성되고 채워져야 합니다. 검색 데이터셋에서 일치하는 항목을 찾으려면, 디테일 데이터셋은 매개변수 값을 데이터 소스에 의해 지시된 현재 레코드의 필드 값에 연결합니다.

이것을 설명을 하기 위해서 고객 테이블과 주문 테이블을 가정해 봅니다. 모든 고객에 대해, 주문 테이블은 고객이 만드는 주문 집합을 포함합니다. 고객 테이블은 고유한 고객 ID를 지정하는 ID 필드를 포함합니다. 주문 테이블은 주문을 작성한 고객의 ID를 지정하는 CustID 필드를 포함합니다.

첫 번째 단계는 Customer 데이터셋을 설정하는 것입니다.

- 1 애플리케이션에 *TSQIDataSet* 컴포넌트를 추가합니다. *CommandType* 속성은 *ctTable*로 설정하고 *CommandText* 속성은 Customer 테이블의 이름으로 설정합니다.



- 2 이름이 *CustomerSource*인 *TDataSource*를 추가합니다. *DataSet* 속성을 1 단계에서 추가한 데이터셋으로 설정합니다. 이 데이터 소스는 이제 Customer 데이터셋을 나타냅니다.
- 3 다른 *TSQLDataSet* 컴포넌트를 추가하고 *CommandType* 속성을 *ctQuery*로 설정합니다. *CommandText* 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 쿼리로 됩니다.

런타임 시 디테일 데이터셋의 SQL 문에 있는 *:ID* 매개변수는 값이 할당되지 않습니다. 따라서 데이터셋은 *CustomersSource*로 식별되는 데이터셋의 열에 대하여 이름별로 매개변수를 일치시키려고 시도합니다. *CustomersSource*는 Customer 테이블에서 데이터를 차례로 파생하는 마스터 데이터셋에서 데이터를 얻습니다. Customer 테이블이 "ID"라는 열을 포함하고 있기 때문에, 마스터 데이터셋의 현재 레코드에 있는 *ID* 필드의 값은 디테일 데이터셋의 SQL 문의 *:ID* 매개변수에 할당됩니다. 데이터셋은 마스터/디테일 관계로 연결됩니다. 현재 레코드가 Customers 데이터셋에서 변경될 때마다 디테일 데이터셋의 SELECT 문은 현재 고객의 ID에 기반한 모든 주문을 검색하기 위해 실행됩니다.

## TSQTable 로 마스터 / 디테일 관계 설정

디테일 셋이 *TSQLTable*의 인스턴스인 마스터/디테일 관계를 설정하려면, *MasterSource* 및 *MasterFields* 속성을 사용합니다.

디테일이 쿼리일 때 사용하는 *DataSource* 속성과 마찬가지로 *MasterSource* 속성은 마스터 테이블에 바인딩된 데이터 소스를 지정합니다. *MasterFields* 속성은 *TSQLTable* 객체 인덱스 필드에 해당하는 마스터 테이블의 필드를 나열합니다.

*MasterFields* 속성을 설정하기 전에 먼저 해당 필드로 시작하는 인덱스를 지정합니다. *IndexName* 속성 또는 *IndexFieldNames* 속성을 사용할 수 있습니다.

사용할 인덱스를 지정하고 나면 *MasterFields* 속성을 사용하여 디테일 SQL 테이블의 인덱스된 필드에 해당하는 마스터 테이블의 열을 나타냅니다. 데이터셋을 여러 열 이름으로 연결하려면 다음과 같이 세미콜론으로 구분하십시오.

```
SQLTable1.MasterFields := 'OrderNo;ItemNo';
```

- 팁** *MasterSource*를 할당한 후에 Object Inspector의 *MasterFields* 속성을 더블 클릭하면 Field Link 편집기가 나타나서 마스터 데이터셋 및 *TSQLTable*의 연결 필드를 선택할 수 있습니다.

## 스키마 정보 액세스

단방향 데이터셋에 쿼리의 결과나 내장 프로시저 대신 서버에서 사용 가능한 것에 대한 정보로 채워질 수 있습니다. 메타데이터라고 하는 이 정보에는 서버에서 어떤 테이블과 내장 프로시저를 이용할 수 있는지와 이러한 테이블 및 내장 프로시저에 대한 정보(예: 테이블에 있는 필드, 정의된 인덱스, 내장 프로시저가 사용하는 매개변수 등)가 포함되어 있습니다.

데이터베이스 서버로부터 메타데이터를 폐치하려면, 먼저 *SetSchemaInfo* 메소드를 사용하여 보고자 하는 데이터를 나타내야 합니다. *SetSchemaInfo*는 다음 세 가지 매개변수를 취합니다.

- 폐치하려는 스키마 정보(메타데이터)의 타입. 테이블 목록(*stTables*), 시스템 테이블 목록(*stSysTables*), 내장 프로시저 목록(*stProcedures*), 테이블의 필드 목록(*stColumns*), 인덱스 목록(*stIndexes*) 또는 내장 프로시저에서 사용되는 매개변수 목록(*stProcedureParams*)들이 여기에 해당됩니다. 각각의 정보 타입은 다른 필드 집합을 사용하여 목록의 항목을 설명합니다. 이들 데이터셋의 구조에 대한 자세한 내용은 18-17 페이지의 "메타데이터 데이터셋의 구조"를 참조하십시오.
- 필드, 인덱스 또는 내장 프로시저 매개변수에 대한 정보를 폐치할 경우에 적용할 테이블 또는 내장 프로시저의 이름. 스키마 정보의 다른 타입을 폐치하는 경우, 이 매개변수는 nil입니다.
- 반환되는 모든 이름에 대해 일치되어야 하는 패턴. 이 패턴은 대표 문자(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 씩습니다(%% 또는 \_\_). 패턴을 사용하지 않으려면 매개변수가 nil이어도 됩니다.

**참고** 테이블(*stTables*)에 대한 스키마 정보를 폐치하려는 경우, 결과 스키마 정보는 SQL 연결의 *TableScope* 속성 값에 따라 순서 테이블, 시스템 테이블, 뷰, 동의어(synonym)를 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 데이터셋은 테이블 이름, 타입, 스키마 이름 등을 제공하는 열이 있는 각 테이블에 대한 레코드를 가집니다. 서버가 MySQL과 같은 메타데이터를 저장하기 위한 시스템 테이블을 사용하지 않는 경우에는, 데이터셋을 열 때 아무런 레코드도 포함되지 않습니다.

앞의 예제는 첫 번째 매개변수만 사용하였습니다. 그 대신 'MyProc'라는 내장 프로시저에 대한 입력 매개변수의 목록을 얻는 경우를 가정해 보십시오. 또한 이 내장 프로시저를 작성한 사람이 점두어를 사용하여 모든 매개변수에 이름을 지정하여 입력 매개변수인지 또는 출력 매개변수인지를 나타내는 경우를 가정해 보십시오. ('inName', 'out Value' 등). 다음과 같이 *SetSchemaInfo*를 호출합니다.

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

결과 데이터셋은 각 매개변수의 속성을 기술하는 열이 있는 입력 매개변수 테이블입니다.

**참고** 데이터셋을 사용하지 않고 데이터베이스 서버에서 목록 객체로 메타데이터를 페치할 수 있습니다. 데이터셋이 아닌 *TSQLConnection*을 사용하는 이러한 방식은 정보가 더 제공되지 않으며, 패턴 일치 문자열을 지원하지도 않습니다. 자세한 내용은 19-10 페이지의 "서버 메타데이터 액세스"를 참조하십시오.

## 메타데이터에 대한 데이터셋 사용 후 데이터 페치

*SetSchemaInfo* 호출 이후에 데이터셋의 쿼리나 내장 프로시저 실행으로 반환되는 방법은 두 가지입니다.

- 데이터를 페치하려는 쿼리, 테이블, 또는 내장 프로시저를 지정하여 *CommandText* 속성을 변경합니다.
- 첫 번째 매개변수를 *stNoSchema*로 설정하여 *SetSchemaInfo*를 호출합니다. 이 경우에, 데이터셋은 *CommandText*의 현재값에 의해 지정된 데이터 페치로 되돌아갑니다.

## 메타데이터 데이터셋의 구조

*TSQLDataSet*을 사용하여 액세스할 수 있는 각각의 메타데이터 타입에는, 요청된 테이블의 항목에 대한 정보가 있는 열(필드)의 이미 정의된 집합이 있습니다.

## 테이블에 대한 정보

테이블(*stTables* 또는 *stSysTables*)에 대한 정보를 요청하면, 결과 데이터셋은 각 테이블에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

표 18.1 테이블을 나열하는 메타데이터 테이블의 열

| 열 이름         | 필드 타입     | 내용                                                                                                                                           |
|--------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------|
| RECNO        | ftInteger | 각 레코드를 고유하게 식별하는 레코드 번호.                                                                                                                     |
| CATALOG_NAME | ftString  | 테이블을 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.                                                                 |
| SCHEMA_NAME  | ftString  | 테이블의 소유자를 식별하는 스키마 이름.                                                                                                                       |
| TABLE_NAME   | ftString  | 테이블 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.                                                                                                            |
| TABLE_TYPE   | ftInteger | 테이블의 타입을 식별합니다. 이 값은 다음 값들 중 하나 이상의 합입니다.<br>1: Table<br>2: View<br>4: System table<br>8: Synonym<br>16: Temporary table<br>32: Local table. |

## 내장 프로시저에 대한 정보

내장 프로시저(*stProcedures*)에 대한 정보를 요청하면 결과 데이터셋은 각 내장 프로시저에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

**표 18.2** 내장 프로시저를 나열하는 메타데이터 테이블의 열

| 열 이름         | 필드 타입      | 내용                                                                                                                |
|--------------|------------|-------------------------------------------------------------------------------------------------------------------|
| RECNO        | ftInteger  | 각 레코드를 고유하게 식별하는 레코드 번호.                                                                                          |
| CATALOG_NAME | ftString   | 내장 프로시저를 포함하는 카탈로그(데이터베이스) 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 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 중 하나 이상의 합입니다.<br>1: Procedure<br>2: Function<br>4: Package<br>8: System procedure |
| IN_PARAMS    | ftSmallint | 입력 매개변수의 수                                                                                                        |
| OUT_PARAMS   | ftSmallint | 출력 매개변수의 수.                                                                                                       |

## 필드에 대한 정보

특정 테이블(*stColumns*)의 필드에 대한 정보를 요청하면 결과 데이터셋은 각 필드에 대한 레코드를 포함합니다. 다음과 같은 열이 포함됩니다.

**표 18.3** 필드를 나열하는 메타데이터 테이블의 열

| 열 이름            | 필드 타입      | 내용                                                                                                                                 |
|-----------------|------------|------------------------------------------------------------------------------------------------------------------------------------|
| RECNO           | ftInteger  | 각 레코드를 고유하게 식별하는 레코드 번호.                                                                                                           |
| CATALOG_NAME    | ftString   | 나열하는 필드의 테이블을 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.                                              |
| SCHEMA_NAME     | ftString   | 필드의 소유자를 식별하는 스키마의 이름.                                                                                                             |
| TABLE_NAME      | ftString   | 필드를 포함하는 테이블의 이름.                                                                                                                  |
| COLUMN_NAME     | ftString   | 필드 이름이 값인 데이터셋의 정렬 순서를 결정합니다.                                                                                                      |
| COLUMN_POSITION | ftSmallint | 테이블에서 열의 위치.                                                                                                                       |
| COLUMN_TYPE     | ftInteger  | 필드 값의 타입을 식별합니다. 이 값은 다음 중 하나 이상의 합입니다.<br>1: Row ID<br>2: Row Version<br>4: Auto increment field<br>8: Field with a default value |
| COLUMN_DATATYPE | ftSmallint | 열의 데이터 타입. 이것은 <i>sqllinks.pas</i> 에 정의된 논리 필드 타입 상수들 중의 하나입니다.                                                                    |

표 18.3 필드를 나열하는 메타데이터 테이블의 열 ( 계속 )

| 열 이름             | 필드 타입      | 내용                                                                                          |
|------------------|------------|---------------------------------------------------------------------------------------------|
| COLUMN_TYPENAME  | ftString   | 데이터 타입을 기술하는 문자열. 이것은 COLUMN_DATATYPE와 COLUMN_SUBTYPE에 포함된 일부 DDL 문에 사용된 형식의 정보와 동일한 정보입니다. |
| COLUMN_SUBTYPE   | ftSmallint | 열의 데이터 타입에 대한 하위 타입. 이것은 sqllinks.pas에 정의된 논리 하위 타입 상수들 중의 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 숫자 수 또는 ADT 및 배열 필드의 자손 수.                                                  |
| COLUMN_LENGTH    | ftInteger  | 필드 값 저장에 필요한 바이트 수                                                                          |
| COLUMN_NULLABLE  | ftSmallint | 필드를 공백으로 둘 수 있는지를 나타내는 부울. (0은 필드에 값이 필요함을 의미합니다)                                           |

## 인덱스에 대한 정보

테이블(stIndexes)의 인덱스에 대한 정보를 요청하면 결과 데이터셋은 각 레코드별로 각 필드 레코드를 포함합니다. (다중 레코드 인덱스는 다중 레코드를 사용하여 기술됩니다) 데이터셋은 다음과 같은 열을 갖습니다.

표 18.4 인덱스를 나열하는 메타데이터 테이블의 열

| 열 이름            | 필드 타입      | 내용                                                                                           |
|-----------------|------------|----------------------------------------------------------------------------------------------|
| RECNO           | ftInteger  | 각 레코드를 고유하게 식별하는 레코드 번호.                                                                     |
| CATALOG_NAME    | ftString   | 인덱스를 포함하는 카탈로그(데이터베이스)의 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 매개변수와 동일합니다.                 |
| SCHEMA_NAME     | ftString   | 인덱스 소유자를 식별하는 스키마의 이름.                                                                       |
| TABLE_NAME      | ftString   | 인덱스가 정의된 테이블의 이름.                                                                            |
| INDEX_NAME      | ftString   | 인덱스 이름. 이 필드는 데이터셋의 정렬 순서를 결정합니다.                                                            |
| PKEY_NAME       | ftString   | 기본 키의 이름을 나타냅니다.                                                                             |
| COLUMN_NAME     | ftString   | 인덱스의 필드(열) 이름.                                                                               |
| COLUMN_POSITION | ftSmallint | 인덱스의 이 필드의 위치.                                                                               |
| INDEX_TYPE      | ftSmallint | 인덱스의 타입을 식별합니다. 이 값은 다음 값들 중에서의 하나 이상의 합입니다.<br>1: Non-unique<br>2: Unique<br>4: Primary key |
| SORT_ORDER      | ftString   | 이 필드에서 인덱스가 오름차순(a)인지 내림차순(d)인지를 나타냅니다.                                                      |
| FILTER          | ftString   | 인덱스된 레코드를 제한하는 필터 조건을 기술합니다.                                                                 |

## 내장 프로시저 매개변수에 대한 정보

내장 프로시저 (*stProcedureParams*) 의 매개변수에 대한 정보를 요청하면 결과 데이터셋은 각 매개변수에 대한 레코드 포함합니다. 다음과 같은 열이 포함됩니다.

**표 18.5** 매개변수를 나열하는 메타데이터 테이블의 열

| 열 이름            | 필드 타입      | 내용                                                                                                             |
|-----------------|------------|----------------------------------------------------------------------------------------------------------------|
| RECNO           | ftInteger  | 각 레코드를 고유하게 식별하는 레코드 번호.                                                                                       |
| CATALOG_NAME    | ftString   | 내장 프로시저를 포함하는 카탈로그(데이터베이스) 이름. 이것은 SQL 연결 컴포넌트에 있는 <i>Database</i> 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. 이것은 <i>TParam</i> 객체의 <i>ParamType</i> 속성과 동일합니다.                                             |
| PARAM_DATATYPE  | ftSmallint | 매개변수의 데이터 타입. 이것은 <i>sqllinks.pas</i> 에 정의된 논리 필드 타입 상수들 중의 하나입니다.                                             |
| PARAM_SUBTYPE   | ftSmallint | 매개변수의 데이터 타입에 대한 하위 타입. 이것은 <i>sqllinks.pas</i> 에 정의된 논리 하위 타입 상수들 중의 하나입니다.                                   |
| PARAM_TYPENAME  | ftString   | 데이터 타입을 기술하는 문자열. 이것은 <i>PARAM_DATATYPE</i> 및 <i>PARAM_SUBTYPE</i> 에 포함되어 있지만 일부 DDL 문에 사용되는 형식으로 된 동일한 정보입니다. |
| PARAM_PRECISION | ftInteger  | 부동 소수점 값의 최대 수 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 공백으로 둘 수 있는지의 여부를 나타내는 부울. (0은 매개변수에 값이 필요함을 의미합니다)                                                      |

## 데이터베이스에 연결

Kylix 애플리케이션은 *dbExpress*를 사용하여 데이터베이스에 연결합니다. *dbExpress*는 SQL 데이터베이스 서버에 대한 빠른 액세스를 제공하는 경량급(lightweight) 데이터베이스 드라이버 집합입니다. 각 지원 데이터베이스에 대해, *dbExpress*는 일정한 *dbExpress* 인터페이스 집합에 적합한 서버 특정 소프트웨어를 사용하는 드라이버를 제공합니다. 자신이 만든 애플리케이션을 배포할 때에는, 애플리케이션 파일에 하나의 공유 객체(서버 특정 드라이버)만 포함하면 됩니다.

각각의 *dbExpress* 연결은 *TSQLConnection* 컴포넌트에 의해 캡슐화됩니다. *TSQLConnection*은 *dbExpress*를 사용하는 데이터베이스 연결 구성에 필요한 모든 정보를 제공하며, 단방향 데이터셋 컴포넌트에서 작동되도록 디자인되었습니다. 여러 단방향 데이터셋은 SQL 연결 컴포넌트를 공유할 수 없을 경우 자체의 연결을 사용합니다.

서버가 각각의 연결에 대해 다중 문장을 지원하지 않는 경우에는 각각의 데이터셋에 대해 개별 연결 컴포넌트를 사용해야 합니다. *MaxStmtsPerConn* 속성을 읽어서 데이터베이스 서버에서 각 데이터셋에 대한 개별 연결이 필요한지 확인하십시오. 기본적으로 서버가 한 연결에서 실행될 수 있는 문장의 수를 제한하는 경우, *TSQLConnection*은 필요에 따라 연결을 생성합니다. 사용 중인 연결을 더 엄격하게 추적하려면 *AutoClone* 속성을 *False*로 설정합니다.

*TSQLConnection*을 사용하면 데이터베이스 서버의 연결이나 사용 방법과 관련한 많은 작업을 수행할 수 있습니다. 여기에는 다음과 같은 작업이 포함됩니다.

- 연결 제어
- 서버 로그인 제어
- 트랜잭션 관리
- 서버 메타데이터 액세스
- 연결된 데이터셋 작업
- 서버에 명령 전송
- 데이터베이스 애플리케이션 디버깅

## 연결 제어

데이터베이스 서버에 대한 연결을 설정하려면, 먼저 원하는 서버 연결을 설명하는 어떤 중요 정보를 애플리케이션이 제공해야 합니다. 서버를 식별하고 로그인 세부 사항을 제공하면, *TSQLConnection*이 서버에 대한 연결을 열거나 닫을 수 있습니다.

### 서버 연결 설명

연결을 여는 *TSQLConnection*에 대한 데이터베이스 연결의 세부 사항을 충분히 설명하려면, 사용할 드라이버와 이 드라이버에 전달되는 일련의 연결 매개변수들을 모두 식별해야 합니다.

### 드라이버 식별

드라이버는 INTERBASE, MYSQL, ORACLE, 또는 DB2 같은 설치된 *dbExpress* 드라이버의 이름인 *DriverName* 속성에 의해 식별됩니다. 드라이버 이름은 다음 두 파일에 연결됩니다.

- *dbExpress* 드라이버는 이름이 libsqlib.so, libsqlmys.so, libsqlora.so, 또는 libsqldb2.so 와 같은 공유 객체 파일입니다.
- 데이터베이스 공급 업체가 제공하는 공유 객체 파일.

이들 두 공유 객체 파일과 데이터베이스 이름 사이의 관계는 *dbExpress* 드라이버를 설치할 때 업데이트되는 "dbxdrivers" 파일에 저장됩니다. 일반적으로 SQL 연결 컴포넌트는 *DriverName*의 값이 제공될 때 dbxdrivers에서 이들을 조회하기 때문에 이들 파일에 대해서는 걱정할 필요가 없습니다. *DriverName* 속성을 설정할 때, *TSQLConnection*은 자동으로 *LibraryName*과 *VendorLib* 속성을 연결된 공유 객체의 이름으로 설정합니다. *LibraryName*과 *VendorLib*가 *GetDriverFunc*과 함께 설정되고 나면, 애플리케이션은 dbxdrivers에 의존할 필요가 없습니다. (즉, 애플리케이션과 함께 dbxdrivers 파일을 배포하지 않아도 됩니다.)

### 연결 매개변수 지정

*Params* 속성은 이름/값 쌍을 나열하는 문자열 목록입니다. 각각의 쌍은 *Name=Value*의 형식을 취하며, 여기서 *Name*은 매개변수의 이름이고, *Value*는 할당하려는 값입니다.

필요한 특정 매개변수는 사용 중인 데이터베이스 서버에 따라 다릅니다. 하지만, *Database* 매개변수는 모든 서버에 필요합니다. 매개변수의 값은 사용 중인 서버에 따라 다릅니다. 예를 들어 InterBase의 경우에 *Database*는 .gdb 파일의 이름이며, Oracle의 경우에는 DB2의 TNSName.ora의 항목이며, 클라이언트 측 노드입니다. 반면에 MySQL의 경우에는 CREATE DATABASE 명령에 의해 할당된 데이터베이스 이름입니다.

다른 일반적인 매개변수에는 *User\_Name*(로그인할 때 사용하는 이름), *Password*(*User\_Name*에 대한 암호), *HostName*(시스템 이름 또는 서버가 위치하는 IP 주소) 및 *TransIsolation*(도입하는 트랜잭션이 다른 트랜잭션에 의한 변경 사항을 기본적으로 인식하는 정도)이 포함됩니다. 드라이버 이름을 지정할 때, *Params* 속성은 해당 드라이버 타입에 필요한 모든 매개변수가 기본값으로 초기화되어 미리 로드됩니다.



*Params*는 문자열 목록이기 때문에, 디자인 타임에 Object Inspector의 *Params* 속성을 더블 클릭하여 String List 편집기를 사용해서 매개변수를 편집할 수 있습니다. 런타임 시 *Params.Values* 속성을 사용하여 개별적인 매개변수에 값을 할당합니다.

## 연결에 대한 설명 이름 지정

항상 *DatabaseName*과 *Params* 속성만 사용하여 연결을 지정할 수도 있지만, 특정 조합에 이름을 지정하고 이름만 가지고 연결을 식별한다면 더 편리할 것입니다. *dbExpress*를 사용하면 데이터베이스와 매개변수의 조합에 대한 이름을 지정할 수 있으며, 이러한 조합은 "dbxconnections"라는 파일에 저장됩니다. 각 조합의 이름을 연결 이름이라고 합니다.

연결 이름을 정의하고 나면 *ConnectionName* 속성을 올바른 연결 이름으로 간단히 설정하여 데이터베이스 연결을 식별할 수 있습니다. *ConnectionName*을 설정하면 자동으로 *DriverName* 및 *Params* 속성이 설정됩니다. *ConnectionName*을 설정한 다음 *Params* 속성을 편집하여 저장된 매개변수 값들에 일시적으로 변화를 줄 수는 있지만 *DriverName* 속성을 변경할 경우 *Params* 및 *ConnectionName*이 모두 지워집니다.

연결 이름을 사용하면 하나의 데이터베이스(예: Local InterBase)를 사용해서 애플리케이션을 개발하여 다른 데이터베이스(예: ORACLE)에서 사용할 목적으로 배포하는 경우에 유용합니다. 이런 경우, *DriverName* 및 *Params*는 개발 당시 사용한 값이 애플리케이션을 배포하는 시스템에 따라 다를 수 있습니다. 두 가지 버전의 dbxconnections 파일을 사용하면 두 가지 연결에 대한 설명 간에 쉽게 전환할 수 있습니다. 디자인 타임에, 애플리케이션은 dbxconnections의 디자인 타임 버전에서 *DriverName* 및 *Params*를 로드합니다. 그런 다음 애플리케이션을 배포할 때 "실제" 데이터베이스를 사용하는 dbxconnections의 개별 버전으로부터 이들 값을 로드합니다. 하지만, 이렇게 하기 위해서는 연결 컴포넌트가 런타임에 *DriverName* 및 *Params* 속성을 다시 로드하도록 지시해야 합니다. 이를 수행하는 방법은 두 가지가 있습니다.

- *LoadParamsOnConnect* 속성을 *True*로 설정합니다. 그러면, 연결이 열릴 때 *TSQLConnection*이 *DriverName* 및 *Params*를 dbxconnections 파일의 *ConnectionName*에 연결된 값들로 자동으로 설정합니다.
- *LoadParamsFromIniFile* 메소드를 호출합니다. 이 메소드는 *DriverName* 및 *Params*를 dbxconnections 파일의 *ConnectionName*에 연결된 값들로 설정합니다. 연결을 열기 전에 특정 매개변수 값을 오버라이드하려는 경우 이 메소드를 사용할 수 있습니다.

## Connection Editor 사용

연결 이름과 연결된 드라이버 및 연결 매개변수 사이의 관계는 dbxconnections 파일에 저장됩니다. Connection Editor를 사용하면 연결을 만들거나 수정할 수 있습니다.

Connection Editor를 표시하려면 *TSQLConnection* 컴포넌트를 더블 클릭합니다. Connection Editor에는 모든 사용 가능한 드라이버가 들어있는 드롭 다운 목록, 현재 선택된 드라이버에 대한 연결 이름 목록, 그리고 현재 선택된 연결 이름에 대한 연결 매개변수를 나열한 테이블이 나타납니다.

이 대화 상자를 사용하여 드라이버 및 연결 이름을 선택하고 일시적으로 오버라이드하려는 매개변수 값을 편집하여 사용하려는 연결을 나타낼 수 있습니다. 원하는 구성을 선택한 다음, Test Connection 버튼을 클릭해서 잘못이 없는지 확인합니다.

이 대화 상자를 사용하여 드라이버와 연결 이름을 선택하여 사용할 연결을 나타낼 수 있습니다. 원하는 구성을 선택한 다음 Test Connection 버튼을 클릭해서 올바른 구성을 선택했는지 확인합니다.

또한 이 대화상자를 사용하여 dbxconnections의 명명된 연결을 편집할 수 있습니다.

- 매개변수 테이블의 매개변수 값을 편집하여 현재 선택되어 있는 명명된 연결을 변경합니다. OK를 클릭해서 대화 상자를 빠져 나가면, 새로운 매개변수 값이 dbxconnections 파일에 저장됩니다.
- Add Connection 버튼을 클릭해서 새로운 명명된 연결을 정의합니다. 대화 상자가 나타나서 사용할 드라이버와 새로운 연결의 이름을 지정할 수 있습니다. 연결이 명명된 후에는, 매개변수를 편집하여 원하는 연결을 지정하고 OK 버튼을 클릭해서 새로운 연결을 dbxconnections에 저장합니다.
- Delete Connection 버튼을 클릭해서 dbxconnections에서 현재 선택되어 있는 명명된 연결을 삭제합니다.
- Rename Connection 버튼을 클릭해서 현재 선택되어 있는 명명된 연결의 이름을 변경합니다. 매개변수에 대한 편집 내용이 OK 버튼을 클릭할 때 새로운 이름으로 저장되는 것에 유의하십시오.

## 서버 연결 열기 및 닫기

*TSQLConnection*은 서버에 대한 연결을 열거나 닫을 때 이벤트를 생성합니다. 이를 사용하여 데이터베이스 연결의 변경 사항에 따른 애플리케이션의 동작을 사용자 지정할 수 있습니다.

### 연결 열기

*TSQLConnection*을 사용하여 데이터베이스 서버에 연결하는 방법에는 두 가지가 있습니다.

- *Open* 메소드를 호출합니다.
- *Connected* 속성을 *True*로 설정합니다.

*Open* 메소드를 호출하면, *Connected*가 *True*로 설정됩니다.

*Connected*를 *True*로 설정하면, *TSQLConnection*이 먼저 *BeforeConnect* 이벤트를 생성하여 초기화를 수행할 수 있습니다. 예를 들어, 이 이벤트를 사용하면 서버를 지정하는 속성을 변경한 다음 연결됩니다. 하지만, 매개변수를 변경하는 경우에는 *Load ParamsOnConnect* 속성이 *False*인지 확인해야 합니다. 그렇지 않으면 이벤트 핸들러가 종료된 다음 *TSQLConnection*이 모든 연결 매개변수를 dbxconnections 파일의 *ConnectionName*에 연결된 값으로 교체합니다.

*BeforeConnect* 이벤트 후에 *TSQLConnection*은 서버 로그인 제어를 선택하는 방법에 따라 기본 로그인 대화 상자를 표시합니다. 그런 다음 사용자 이름과 암호를 드라이버에 전달하고 연결을 엽니다.

연결이 열리고 나면, *TSQLConnection*이 *AfterConnect* 이벤트를 생성하여 서버로부터 메타데이터 페치를 비롯한 열린 연결이 필요한 모든 작업을 수행할 수 있습니다.

일단 연결이 만들어지면 연결을 사용하는 활성 데이터셋이 하나 이상되는 한 계속 유지됩니다. 활성 데이터셋이 더 이상 없을 경우 *TSQLConnection*은 *Keep Connection* 속성의 값에 따라 연결을 해제할 수도 있습니다.

*KeepConnection*은 데이터베이스와 연결된 모든 데이터셋이 닫힌 경우에도 애플리케이션이 데이터베이스에 대한 연결을 유지할 지의 여부를 결정합니다. *True*인 경우에는 연결이 유지됩니다. 원격 데이터베이스 서버에 대한 연결의 경우 또는 데이터셋을 자주 열고 닫는 애플리케이션의 경우에 *KeepConnection*을 *True*로 설정하면 네트워크 트래픽이 감소되고 애플리케이션의 속도가 빨라집니다. *KeepConnection*이 *False*인 경우에는 데이터베이스를 사용하는 활성 데이터셋이 없을 경우 연결이 해제됩니다. 데이터베이스를 사용하는 데이터셋이 나중에 열리는 경우에는 연결이 재구성되고 초기화되어야 합니다.

## 데이터베이스 서버로부터 연결 해제

*TSQLConnection*을 사용하여 서버로부터 연결을 해제하는 방법은 두 가지입니다.

- *Connected* 속성을 *False*로 설정합니다.
- *Close* 메소드를 호출합니다.

*Close*를 호출하면, *Connected*가 *False*로 설정됩니다.

*Connected*가 *False*로 설정되면, *TSQLConnection*은 *BeforeDisconnect* 이벤트를 생성하여 연결을 닫기 전에 모든 해제 작업을 수행할 수 있습니다. 예를 들어, 이 이벤트를 사용하면 데이터셋이 닫히기 전에 모든 열린 데이터셋에 대한 정보를 캐시로 저장할 수 있습니다.

*TSQLConnection*은 *BeforeConnect* 이벤트 후에 모든 열린 데이터셋을 닫고 서버로부터 연결을 해제합니다.

마지막으로, *TSQLConnection*은 *AfterDisconnect* 이벤트를 생성하여 사용자 인터페이스의 Connect 버튼을 사용 가능하게 만드는 것 같은 연결 상태의 변경에 응답할 수 있습니다.

**참고** *Close*를 호출하거나 *Connected*를 *False*로 설정하면 *KeepConnection*이 *True*일지라도 데이터베이스 서버로부터 연결이 해제됩니다.

# 서버 로그인 제어

대부분의 원격 데이터베이스 서버는 승인되지 않은 액세스를 방지하는 보안 기능을 포함합니다. 일반적으로 서버는 데이터베이스 액세스를 허용하기 전에 사용자 이름 및 암호 로그인을 필요로 합니다.

디자인 타임 시 서버에 로그인이 필요하면 데이터베이스에 대한 연결을 처음으로 시도할 때 표준 로그인 대화 상자가 나타나 사용자 이름과 암호 입력을 기다립니다.

런타임 시 서버의 로그인 요청을 처리할 수 있는 방법에는 여러 가지가 있습니다.

- 기본 로그인 대화 상자와 프로세스가 로그인을 처리하도록 합니다. 이것이 기본적인 방법입니다. *TSQLConnection* 객체의 *LoginPrompt* 속성을 *True*(기본값)로 설정하고 *DBLogDlg*를 연결 컴포넌트를 선언하는 유닛의 **uses** 절에 추가합니다. 애플리케이션은 서버가 사용자 이름과 암호를 요청할 때 표준 로그인 대화 상자를 표시합니다.
- *dbxconnections* 파일의 *User\_Name*과 *Password*에 값을 제공합니다. 특정 *ConnectionName* 아래의 Connection Editor로부터 이들 매개변수 값을 저장할 수 있습니다. 기본 로그인 대화 상자가 나타나지 않도록 하려면 *LoginPrompt*를 *False*로 설정합니다. 일반적으로 개발 중에 로그인 요구를 피하기 위해서 이러한 방법을 사용합니다. *dbxconnections* 파일에 암호화되지 않은 암호를 남겨둘 경우 누구라도 이를 읽을 수 있으므로 이러한 방법은 보안에 심각한 영향을 초래합니다.
- *Params* 속성을 사용하여 로그인을 시도하기 전에 로그인 정보를 제공합니다. 디자인 타임에 Object Inspector를 통해서 또는 런타임에 프로그램적으로 *User\_Name*과 *Password* 매개변수에 값을 할당할 수 있습니다. 기본 로그인 대화 상자가 나타나지 않도록 하려면 *LoginPrompt*를 *False*로 설정합니다. 예를 들어, 다음 코드에서는 *BeforeConnect* 이벤트 핸들러에서 *dbxconnections* 파일에 저장된 암호화된 암호를 해독하여 사용자 이름과 암호를 설정합니다.

```
procedure TForm1.SQLConnectionBeforeConnect(Sender:TObject);
begin
  with Sender as TSQLConnection do
    begin
      if LoginPrompt = False then
        begin
          Params.Values['User_Name'] := 'SYSDBA';
          Params.Values['Password'] := Decrypt(Params.Values['Password']);
        end;
      end;
    end;
end;
```

디자인 타임에 *Params* 속성의 값을 설정하거나 코드에서 하드 코딩된 문자열을 사용하면 애플리케이션의 실행 파일에 값이 포함된다는 점에 유의하십시오. *dbxconnections* 파일에서와 같이 자유롭게 사용할 수는 없지만 서버 보안도 충족 해주면서 이를 쉽게 찾을 수 있도록 해줍니다

- *OnLogin* 이벤트에 대한 이벤트 핸들러를 작성합니다. *LoginPrompt* 속성을 *True*로 설정하고, *OnLogin* 이벤트 핸들러에서 로그인 매개변수를 설정합니다. *User\_Name*, *Password*, 및 *Database* 매개변수의 사본이 *LoginParams* 매개변수의 이벤트 핸들러로 전달됩니다. 이 문자열 목록을 사용하여 이들 매개변수에 값을

할당하여 필요한 값을 제공합니다. 종료 시 *LoginParams*에서 반환된 값은 연결을 구성하는 데 사용됩니다. 다음 예에서는 전역 변수(*UserName*)와 특정 사용자 이름(*PasswordSearch*)에 따른 암호를 반환하는 메소드를 사용하여 *User\_Name* 및 *Password* 데이터베이스 매개변수에 값을 할당합니다.

```
procedure TForm1.SQLConnection1Login(Database: TDatabase; LoginParams: TStrings);
begin
    LoginParams.Values['User_Name'] := UserName;
    LoginParams.Values['Password'] := PasswordSearch(UserName);
end;
```

로그인 매개변수를 제공하는 다른 메소드에서와 같이, *OnLogin* 이벤트 핸들러를 작성할 때는 애플리케이션 코드에 암호를 하드 코딩하지 마십시오. 암호는 암호화된 값, 즉 애플리케이션이 값을 조회하기 위해 사용하는 보안 데이터베이스의 항목으로만 나타나거나 사용자로부터 동적으로 얻어질 수 있어야만 합니다.

**경고** *OnLogin* 이벤트는 *LoginPrompt* 속성이 *True*가 아니면 발생하지 않습니다. *LoginPrompt* 값을 *False*로 하고 *OnLogin* 이벤트 핸들러에서만 로그인 정보를 제공하면, 데이터베이스에 로그인할 수 없는 상황이 발생합니다. 기본 대화 상자가 나타나지 않고 *OnLogin* 이벤트 핸들러가 실행되지 않습니다.

## 트랜잭션 관리

트랜잭션은 커밋(영구적으로)되기 전에 데이터베이스의 하나 이상의 테이블에서 모두 성공적으로 수행되어야 하는 작업 그룹입니다. 그룹의 작업 중 하나라도 실패하면, 모든 작업이 롤백(실행 취소)됩니다. 트랜잭션을 사용함으로써, 트랜잭션을 구성하는 작업 중 하나를 완료하면서 문제가 발생할 때 데이터베이스가 불일치 상태가 되지 않도록 합니다.

예를 들면, 은행 업무용 애플리케이션에서 한 계좌에서 다른 계좌로의 자금 전송은 트랜잭션으로 보호해야 하는 작업입니다. 한 계좌의 잔액을 줄이고 나서 다른 계좌의 잔액을 증가시킬 때 오류가 발생하면, 트랜잭션을 롤백하여 데이터베이스가 정확한 총 잔액을 반영하도록 합니다.

SQL 명령을 데이터베이스에 직접 전송하여 트랜잭션을 관리하는 것은 언제나 가능합니다. MySQL 같은 일부 데이터베이스는 트랜잭션을 전혀 지원하지 않지만, 각각의 데이터베이스는 자체적인 트랜잭션 관리 모델을 제공합니다. 트랜잭션을 지원하는 서버의 경우, 스키마 캐시 같은 특정 데이터베이스 서버의 고급 트랜잭션 관리 기능을 이용하여 사용자 고유의 트랜잭션 관리를 코드로 직접 작성할 수 있습니다.

고급 트랜잭션 관리 기능을 사용할 필요가 없는 경우, *TSQLConnection*은 SQL 명령을 명시적으로 전송하지 않고 트랜잭션을 관리하기 위해 사용할 수 있는 일련의 메소드와 속성을 제공합니다. 이들 속성과 메소드를 사용하면 서버가 트랜잭션을 지원하는 한 사용하는 데이터베이스 서버의 각 타입에 따라 애플리케이션을 사용자 지정할 필요가 없다는 이점이 있습니다. (MySQL 같은 트랜잭션을 지원하지 않는 데이터베이스에서 트랜잭션을 사용하려고 시도하면 *TSQLConnection*이 예외를 발생시킵니다.)

**경고** 클라이언트 데이터셋이 *TSQLConnection* 컴포넌트를 통해 액세스된 데이터를 편집할 때 (*TSQLConnection* 컴포넌트를 사용하는 단방향 데이터셋에 직접 연결되거나 또는 프로바이더를 통해서 연결된 경우), 프로바이더는 모든 업데이트에 대한 트랜잭션 지원을 암시적으로 제공합니다. 명시적으로 시작하는 트랜잭션이 프로바이더에 의해 생성된 트랜잭션과 충돌하지 않는지 유의하십시오.

## 트랜잭션 시작

*StartTransaction* 메소드를 호출하여 트랜잭션을 시작합니다. *StartTransaction*은 단일 매개변수를 취하는데, 이 매개변수는 사용자가 여러 트랜잭션을 동시에 관리할 수 있고 각 트랜잭션 기반에 따라 트랜잭션 분리 레벨을 지정할 수 있는 트랜잭션 디스크립터입니다. (트랜잭션 레벨에 대한 자세한 내용은 19-9 페이지의 "트랜잭션 분리 레벨 지정"을 참조하십시오.)

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

여러 트랜잭션을 동시에 관리하려면, 트랜잭션 디스크립터의 *TransactionID* 필드를 고유한 값으로 설정합니다. *TransactionID*가 고유한(현재 사용 중인 다른 트랜잭션과 충돌하지 않는) 값인 경우, 어떤 값이라도 선택할 수 있습니다. 서버에 따라, *TSQLConnection*으로 시작된 트랜잭션이 중첩되거나 오버랩될 수 있습니다. 여러 트랜잭션을 동시에 생성하기 전에 데이터베이스 서버가 이를 지원하는지 확인하십시오.

기본적으로 트랜잭션을 시작할 경우, 트랜잭션이 명시적으로 종료될 때까지 또는 오버랩된 트랜잭션의 경우는 다른 트랜잭션이 시작할 때까지 데이터베이스에서 읽거나 쓰기 작업을 하는 이어지는 모든 문장은 해당 트랜잭션의 컨텍스트에 있습니다. 각 문장은 그룹의 일부로 간주됩니다. 변경 사항이 데이터베이스에 성공적으로 커밋되어야 하며, 그렇지 않으면 그룹에 취해진 모든 변경 사항이 실행 취소됩니다.

오버랩된 트랜잭션의 경우 두 번째 트랜잭션이 시작할 때 첫 번째 트랜잭션은 비활성화되지만 첫 번째 트랜잭션의 커밋이나 롤백을 나중에 연기할 수는 있습니다. 하지만, InterBase 데이터베이스를 사용하는 경우에는 *TransactionLevel* 속성을 설정하여 개별 활성 트랜잭션으로 애플리케이션의 각 데이터셋을 식별할 수 있습니다. 즉, 두 번째 트랜잭션을 시작한 후에 단지 데이터셋을 원하는 트랜잭션에 연결함으로써 두 트랜잭션 모두 동시에 작업할 수 있습니다.

*InTransaction* 속성을 확인하면 트랜잭션이 처리 중인지를 알아낼 수 있습니다.

## 트랜잭션 종료

이상적으로 보면 트랜잭션은 필요한 만큼 지속되어야 합니다. 트랜잭션이 활성화되는 시간이 길면 길수록, 데이터베이스에 액세스하는 동시 사용자가 더 많아지고, 트랜잭션 수명 동안 시작하고 종료하는 동시에 발생하는 트랜잭션이 더 많아지며, 어떤 변경 사항을 커밋하려고 시도할 때 트랜잭션이 다른 트랜잭션과 충돌할 가능성이 더 커집니다.

## 성공적인 트랜잭션 종료

트랜잭션을 구성하는 작업이 모두 성공하면, *Commit* 메소드를 사용하여 데이터베이스의 변경 사항을 영구적으로 만들 수 있습니다. *Commit*은 단일 매개변수를 취합니다. 이 매개변수는 *StartTransaction* 메소드에 제공하는 트랜잭션 디스크립터입니다.

```
SQLConnection1.Commit(TD);
```

**참고** 중첩 트랜잭션을 작업 중인 경우에는 기본 트랜잭션이 롤백될 때, 나중에 보조 트랜잭션을 롤백하기 위해서만 보조(중첩) 트랜잭션을 커밋하는 것이 가능합니다.

*Commit*은 대개 **try...except** 문에서 시도됩니다. 이러한 방식으로 트랜잭션을 성공적으로 커밋할 수 없으면, **except** 블록을 사용하여 오류를 처리하고 작업을 재시도하거나 트랜잭션을 롤백할 수 있습니다.

## 성공적이지 않은 트랜잭션 종료

트랜잭션의 일부인 변경 사항을 작성하거나 트랜잭션 커밋을 시도할 때 오류가 발생하면, 트랜잭션을 구성하는 모든 변경 사항을 취소할 수 있습니다. 이들 변경 사항을 취소하려면, *Rollback* 메소드를 사용합니다. *Rollback*은 단일 매개변수를 취합니다. 이 매개변수는 *StartTransaction* 메소드에 제공하는 트랜잭션 디스크립터입니다.

```
SQLConnection1.Rollback(TD);
```

*Rollback*은 대개 다음에서 발생합니다.

- 데이터베이스 오류를 복구할 수 없는 경우의 예외 처리 코드.
- 사용자가 Cancel 버튼을 클릭할 때와 같은 버튼이나 메뉴 이벤트 코드.

## 트랜잭션 분리 레벨 지정

트랜잭션을 시작할 때 제공하는 트랜잭션 디스크립터에는 동일한 테이블을 사용할 때 다른 동시 트랜잭션에 트랜잭션이 반응하는 방식을 제어할 수 있는 *IsolationLevel* 필드가 포함됩니다. 특히 이 필드는 얼마나 많은 트랜잭션이 테이블에 대한 다른 트랜잭션의 변경 사항을 "보는" 지에 영향을 줍니다.

각 서버 타입은 *IsolationLevel*에 대해 서로 다른 가능한 값을 지원하고, 일부 서버 타입은 다른 분리 레벨을 전혀 지원하지 않습니다. 서버가 다른 분리 레벨을 지원하는 경우 서버에 따라 허용 가능한 값에 다음 중 두 개 이상이 포함됩니다.

- *DirtyRead : TransIsolation*이 *DirtyRead*인 경우 트랜잭션은 커밋되지 않은 경우에도 다른 트랜잭션에 의해 만들어진 모든 변경 사항을 봅니다. 커밋되지 않은 변경 사항은 영구적이지 않으며, 언제든지 롤백될 수도 있습니다. 이 값은 최소한의 분리 레벨을 제공하며, 많은 데이터베이스 서버에서 사용할 수 없습니다(예: Oracle 또는 Inter Base).
- *ReadCommitted : TransIsolation*이 *ReadCommitted*인 경우 다른 트랜잭션에서 취해진 커밋된 변경 사항만 볼 수 있습니다. 이 설정은 롤백될 수도 있는 커밋되지 않은 변경 사항을 트랜잭션이 보지 못하도록 보호하지만, 사용자가 읽는 중일 때 다른 트랜잭션이 커밋될 경우 데이터베이스 상태가 일치되어 보이지 않을 수도 있습니다.

- *RepeatableRead : TransIsolation*이 *RepeatableRead*일 경우 트랜잭션은 분명 데이터베이스 데이터의 상태가 일치된 것으로 보게 됩니다. 트랜잭션은 데이터의 단일 스냅샷을 봅니다. 트랜잭션은 다른 동시 트랜잭션이 커밋된 경우에도 다른 트랜잭션에 의한 이후의 데이터 변경 사항을 볼 수 없습니다. 이 분리 레벨은 분명 사용자의 트랜잭션이 레코드를 읽고난 후에는 레코드의 뷰가 변하지 않게 합니다. 이 레벨에서 트랜잭션은 다른 트랜잭션에 의한 변경 사항으로부터 가장 많이 분리됩니다.

**참고** 또한, 트랜잭션 디스크립터를 사용하면 데이터베이스 드라이버에 의해 정의된 사용자 지정 분리 레벨을 지정할 수 있습니다. Kylix가 제공하는 어떠한 dbExpress 드라이버도 사용자 지정 분리 레벨을 지원하지 않습니다.

각 분리 레벨의 구현 방법에 대한 자세한 설명은 서버 설명서를 참조하십시오.

## 서버 메타데이터 액세스

연결이 열린 후에는 *TSQLConnection*을 사용하여 데이터베이스 서버에서 사용할 수 있는 항목에 대한 정보를 얻을 수 있습니다. 메타데이터라고 하는 이러한 정보에는 어떤 테이블 및 내장 프로시저를 서버에서 이용할 수 있는지 그리고 이러한 테이블 및 내장 프로시저에 대한 정보(예: 테이블에 있는 필드, 정의된 색인, 내장 프로시저가 사용하는 매개변수 등)가 포함되어 있습니다.

*TSQLConnection*에는 이 메타 데이터로 목록을 채우기 위해 호출할 수 있는 여러 메소드가 있습니다.

- *GetTableNames*는 *TStrings* 자손(예: *TStringList*)을 서버에서 사용할 수 있는 테이블의 이름으로 채웁니다. 반환되는 테이블은 다음 두 가지에 따라 다릅니다. *TableScope* 속성 및 사용자가 단지 시스템 테이블에만 관심이 있는지를 나타내는 부울 매개변수. 시스템 테이블만 원할 경우에는 *SystemTables* 매개변수를 사용할 수 있습니다. 그러면 *TableScope*의 값과는 상관없이 목록에 시스템 테이블만 포함됩니다. (모든 서버가 메타데이터를 저장하기 위해 시스템 테이블을 사용하는 것은 아닙니다. 따라서 시스템 테이블을 요구했을 때 빈 목록이 반환될 수도 있습니다.) 시스템 테이블만 요청하지 않을 경우, *GetTableNames*는 *TableScope* 속성에 의해 지정된 타입과 일치하는 테이블의 이름을 반환합니다. 여기에는 시스템 테이블, 데이터 테이블, 뷰 및 동의어가 포함될 수 있습니다.
- *GetFieldNames*는 지정된 테이블의 모든 필드(열)의 이름으로 *TStrings* 자손을 채웁니다.
- *GetIndexNames*는 특정 테이블에 정의된 모든 인덱스의 이름으로 *TStrings* 자손을 채웁니다.
- *GetProcedureNames*는 모든 사용 가능한 내장 프로시저가 있으면 그 이름으로 *TStrings* 자손을 채웁니다.
- *GetProcedureParams*는 각각의 레코드가 이름, 인덱스, 매개변수 타입, 필드 타입 등을 포함한 특정 내장 프로시저의 매개변수를 설명하는 매개변수 설명 레코드에 대한 포인터로 *TList* 객체를 채웁니다. 이들 매개변수 설명을 전역 *LoadParamListItems* 프로시저를 호출하여 더 익숙한 *TParams* 객체로 변환할 수 있습니다.



*GetProcedureParams*는 개별 레코드를 동적으로 할당하기 때문에 애플리케이션은 정보를 끝마쳤을 때 이를 해제해야 합니다.

**참고** 또한 *TSQLDataSet*을 사용하여 서버 메타데이터를 액세스할 수도 있습니다. *TSQL DataSet*은 반환된 정보에 대한 더 자세한 정보와 더 큰 제어 기능을 제공합니다(예를 들어, 반환된 항목을 제한하기 위해 패턴 마스크를 제공할 수 있습니다). *TSQL DataSet*으로 서버 메타데이터를 액세스하는 방법에 대한 자세한 내용은 18-16 페이지의 "스키마 정보 액세스"를 참조하십시오.

## 연결된 데이터셋 작업

*TSQLConnection*은 데이터베이스에 연결하기 위해 사용하는 모든 활성 데이터셋의 목록을 유지합니다. 예를 들면, 이 목록을 사용하여 데이터베이스 연결을 닫을 때 모든 데이터셋을 닫습니다.

또한 이 목록을 사용하여 특정 데이터베이스에 연결하기 위해 특정 *TSQLConnection*을 사용하는 모든 데이터셋에서 작업을 수행할 수도 있습니다.

### 서버로부터 연결을 해제하지 않고 데이터셋 닫기

데이터베이스 서버로부터 연결을 해제하지 않고 모든 데이터셋을 닫고자 하는 경우가 있습니다. 서버로부터 연결을 해제하지 않고 열려 있는 모든 데이터셋을 닫으려면, 다음과 같은 단계를 따릅니다.

- 1 *TSQLConnection* 컴포넌트의 *KeepConnection* 속성을 *True*로 설정합니다.
- 2 *TSQLConnection* 컴포넌트의 *CloseDataSets* 메소드를 호출합니다.

### 연결된 데이터셋을 통한 반복

*TSQLConnection* 인스턴스를 사용하는 모든 데이터셋에서 어떤 작업(데이터셋을 모두 닫는 작업은 제외)을 수행하려면, *DataSets* 및 *DataSetCount* 속성을 사용합니다. *Data Sets*은 *SQLConnection* 속성을 통해 SQL 연결 컴포넌트에 연결된 모든 활성 데이터셋의 인덱스된 배열입니다. *DataSetCount*는 이 배열에 있는 데이터셋의 수입니다.

**참고** SQL 클라이언트 데이터셋 (*TSQLClientDataSet*)을 사용할 때, 연결 컴포넌트의 *DataSets* 속성은 클라이언트 데이터셋 자체를 나열하지는 않습니다. 그 대신, 클라이언트 데이터셋이 데이터를 액세스하기 위해 사용하는 내부 단방향 데이터셋을 나열합니다.

*DataSets*은 활성(열린) 데이터셋만 나열합니다. 데이터셋이 닫힌 경우에는 목록에 나타나지 않습니다.

*DataSetCount*로 *DataSets*을 사용하여 코드의 현재 활성 상태인 모든 데이터셋을 순환할 수 있습니다. 예를 들면, 다음 코드에서는 모든 활성 데이터셋을 순환하고 제공하는 데이터를 사용하는 모든 컨트롤을 사용 불가능하게 합니다.

```
var
  I:Integer;
begin
  with SQLConnection1 do
    begin
      for I := 0 to DataSetCount - 1 do
        DataSets[I].DisableControls;
      end;
    end;
end;
```

## 서버에 명령 전송

CREATE INDEX, ALTER TABLE 및 DROP DOMAIN 문 같은 일반 Data Definition Language (DDL) SQL 문은 *Execute* 메소드를 사용하여 *TSQLConnection* 컴포넌트로부터 직접 실행될 수 있습니다. 이 문장들은 결과 셋을 반환하지 않고, 단지 데이터베이스의 메타데이터에서 작동만 하거나 이를 작성합니다. *Execute* 메소드는 또한 INSERT, DELETE, 및 UPDATE 문 같은 Data Manipulation Language (DML) SQL 문을 실행하기 위해 사용될 수도 있습니다.

*Execute*는 실행하려는 단일 SQL 문을 지정하는 문자열, 이 문장에 대한 모든 매개변수 값을 제공하는 *TParams* 객체, 그리고 문장 실행의 결과 셋으로 채워지는 동적으로 생성된 *TCustomSQLDataSet* 객체를 수신할 수 있는 포인트, 이 세 개의 매개변수를 취합니다. 세 번째 매개변수를 사용하여 문 실행의 결과로 반환된 레코드를 수신할 수도 있지만, 그 대신 SQL 데이터셋을 사용하는 것을 권장합니다. 세 번째 매개변수를 사용하지 않으면, 애플리케이션이 세 번째 매개변수에서 반환된 데이터셋을 해제해야 합니다.

**참고** *Execute*는 한 번에 하나의 SQL 문만 실행할 수 있습니다. SQL 스크립트 유틸리티에서와 같이 *Execute*의 단일 호출로 여러 SQL 문을 실행하는 것은 불가능합니다. 하나 이상의 문장을 실행하려면, 새로운 매개변수마다 *Execute*를 호출합니다.

어떠한 매개변수도 포함하지 않는 문장을 실행하는 것은 비교적 쉽습니다. 예를 들어, 다음 코드는 어떠한 매개변수 없이 CREATE TABLE 문 (DDL)을 실행합니다.

```
procedure TForm1.CreateTableButtonClick(Sender:TObject);
var
  SQLstmt:String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '(' +
    ' CustNo INTEGER, ' +
    ' Company CHAR(40), ' +
    ' State CHAR(2), ' +
    ' PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

매개변수를 사용하려면 *TParams* 객체를 생성해야 합니다. 각각의 매개변수 값에 대해, *TParams.CreateParam* 메소드를 사용하여 *TParam* 객체를 추가합니다. 그런 다음 *TParam* 속성을 사용하여 매개변수를 설명하고 값을 설정합니다.

이 프로세스는 INSERT 문을 실행하는 다음 예제에 설명되어 있습니다. INSERT 문에는 이름이 *:StateParam* 인 단일 매개변수가 포함됩니다. *stmtParams* 라고 하는 *TParams* 객체는 해당 매개변수에 대해 "CA" 값을 제공하기 위해 생성됩니다.

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender:TObject);
var
  SQLstmt:String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    SQLConnection1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    SQLConnection1.Execute(SQLstmt, stmtParams, nil);
  finally
    stmtParams.Free;
  end;
end;
```

SQL 문에 매개변수가 있는 경우 이에 대한 값을 제공하기 위한 *TParam* 객체를 제공하지 않으면, SQL 문이 실행될 때 오류를 발생시킬 수도 있습니다(이것은 백엔드에서 사용되는 특정 데이터베이스에 따라 다릅니다). *TParam* 객체가 제공되지만 SQL 문에 일치하는 매개변수가 없는 경우에는, 애플리케이션이 *TParam*를 사용하려고 시도할 때 예외가 발생합니다.

## 데이터베이스 애플리케이션 디버깅

데이터베이스 애플리케이션을 디버깅하는 동안에, 사용자를 위해 자동으로(예를 들면, 프로바이더 컴포넌트나 *dbExpress* 드라이버에 의해) 생성되는 메시지를 비롯하여 연결 컴포넌트를 통해 데이터베이스 서버와 주고 받는 SQL 메시지를 모니터링하는 것이 좋습니다.

### SQL 명령을 모니터링하기 위해 TSQLMonitor 사용

*TSQLConnection*는 짝을 이루는 컴포넌트인 *TSQLMonitor*를 사용하여 이 메시지들을 인터셉트하고 이를 문자열 목록에 저장합니다. *TSQLMonitor*를 사용하려면 다음과 같이 합니다.

- 1 모니터하려는 SQL 명령이 있는 *TSQLConnection* 컴포넌트를 갖고 있는 폼이나 데이터 모듈에 *TSQLMonitor* 컴포넌트를 추가합니다.
- 2 *SQLConnection* 속성을 *TSQLConnection* 컴포넌트로 설정합니다.
- 3 SQL 모니터의 *Active* 속성을 *True*로 설정합니다.

SQL 명령이 서버에 전송될 때, SQL 모니터의 *TraceList* 속성은 인터셉트되는 모든 SQL 명령을 나열하도록 자동으로 업데이트됩니다.

*FileName* 속성에 대한 값을 지정한 다음 *AutoSave* 속성을 *True*로 설정하여 이 목록을 파일로 저장할 수 있습니다. *AutoSave*는 SQL 모니터가 새로운 메시지가 기록될 때마다 *TraceList* 속성의 내용을 파일로 저장하게 합니다.

메시지가 기록될 때마다 파일을 저장해야 하는 수고를 하지 않으려면 *OnLogTrace* 이벤트 핸들러를 사용하여 일정 수의 메시지가 기록된 후에만 파일로 저장하도록 할 수 있습니다. 예를 들면, 다음과 같은 이벤트 핸들러는 10번째 메시지마다 *TraceList*의 내용을 저장하고, 저장한 다음 로그를 지워서 목록이 너무 커지지 않게 합니다.

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo:Pointer);
var
  LogFileName:string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
  end;
end;
```

**참고** 앞의 이벤트 핸들러를 사용했다면 애플리케이션이 종료될 때 항목이 10 이하인 부분 목록을 저장할 수도 있습니다.

## SQL 명령을 모니터하기 위해 콜백 사용

*TSQLMonitor*를 사용하는 대신, SQL 연결 컴포넌트의 *SetTraceCallbackEvent* 메소드를 사용하여 애플리케이션이 SQL 명령을 추적하는 방법을 사용자 지정할 수 있습니다. *SetTraceCallbackEvent*는 *TSQLCallbackEvent* 타입의 콜백 그리고 콜백 함수로 전달되는 사용자 정의 값, 두 개의 매개변수를 취합니다.

콜백 함수는 *CallType* 및 *CBInfo* 두 개의 매개변수를 취합니다.

- *CallType*은 나중에 사용하기 위해 저장해 둡니다.
- *CBInfo*는 카테고리 (*CallType*에서와 같음), SQL 명령의 텍스트 및 *SetTraceCallbackEvent* 메소드로 전달되는 사용자 정의 값을 포함하는 레코드에 대한 포인터입니다.

콜백은 일반적으로 *cbrUSEDEF*인 *CBRTYPE* 타입의 값을 반환합니다.

*dbExpress* 드라이버는 SQL 연결 컴포넌트가 서버에 명령을 전달하거나 서버가 오류 메시지를 반환할 때마다 사용자의 콜백을 호출합니다.

**경고** *TSQLConnection* 객체에 연결된 *TSQLMonitor* 컴포넌트가 있을 경우에는 *SetTraceCallbackEvent*를 호출하지 마십시오. *TSQLMonitor*는 작업을 위해 콜백 메커니즘을 사용하며, *TSQLConnection*은 한 번에 하나의 콜백만 지원할 수 있습니다.

## 클라이언트 데이터셋 사용

클라이언트 데이터셋은 *TDataSet*이 가지고 있는 모든 데이터 액세스, 편집, 탐색, 데이터 제약 조건 및 필터링 지원 등을 제공합니다. 클라이언트 데이터셋은 모든 데이터를 메모리에 캐시하고 인메모리 (in-memory) 캐시를 처리합니다. 메모리에 저장하는 데이터를 처리하기 위한 지원은 공유 객체 파일인 *midas.so*에서 제공하며, 이 파일은 클라이언트 데이터셋을 사용하는 모든 애플리케이션에 배포되어야 합니다.

Kylix에서는 다음과 같은 두 가지 타입의 클라이언트 데이터셋을 제공합니다.

- *TClientDataSet* - *dbExpress* 같은 특정 타입의 데이터베이스 연결 지원 없이 작동하도록 디자인되었습니다.
- *TSQLClientDataSet* - *dbExpress*를 사용하여 인메모리 캐시 데이터를 페치 (fetch) 합니다.

클라이언트 데이터셋의 두 타입은 *TDataSet*에서 상속된 속성 및 메소드에 추가되어 데이터 처리를 위한 확장된 기능 집합을 제공합니다. 두 타입은 메모리에 캐시된 데이터를 가져오는 방법에 있어서 기본적으로 다릅니다.

클라이언트 데이터셋의 두 타입 모두 데이터 읽기와 업데이트 쓰기를 위한 다음과 같은 메커니즘을 지원합니다.

- 클라이언트 데이터셋에서 직접 액세스하는 파일 읽기 및 쓰기. 이 메커니즘만 사용할 경우 애플리케이션은 *TClientDataSet*를 사용해야 하며, 이렇게 하면 오버헤드가 줄어듭니다. 그러나 브리프케이스 (briefcase) 모델을 사용할 경우 이 메커니즘은 클라이언트 데이터셋의 두 타입 모두에 적합합니다. 이 메커니즘을 지원하는 속성 및 메소드는 20-38 페이지의 "파일 기반 데이터로 클라이언트 데이터셋 사용"에서 설명합니다.
- 다른 데이터셋에서 읽기. 클라이언트 데이터셋은 다른 데이터셋에서 데이터를 복사하기 위한 다양한 메커니즘을 제공합니다. 이 내용은 20-23 페이지의 "다른 데이터셋에서 데이터 복사"에서 설명합니다.

또한, *TClientDataSet*는 프로바이더 컴포넌트에서 데이터를 페치할 수 있습니다. 프로바이더 컴포넌트는 *TClientDataSet*를 다른 데이터셋에 연결합니다. 프로바이더는 소스

데이터셋에서 클라이언트 데이터셋으로 데이터를 전달 (forward) 하고 편집 내용을 클라이언트 데이터셋에서 소스 데이터셋으로 다시 보냅니다. 프로바이더는 옵션으로서 서버에 업데이트 내용을 적용하는 SQL 명령을 작성하여, 이를 소스 데이터셋에서 서버로 전달 (forward) 합니다. 프로바이더 작업을 위한 속성 및 메소드는 20-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"에서 설명합니다.

*TSQLClientDataSet*은 외부 프로바이더에 연결할 수 없지만, 그 대신 데이터베이스 서버에서 데이터를 페치합니다. 클라이언트 데이터셋은 해당 인메모리 캐시에 업데이트 내용을 계속 캐시하고 캐시된 업데이트 내용을 데이터베이스 서버에 다시 적용하기 위한 메소드를 포함합니다. 데이터베이스 서버에 연결하는 *TSQLClientDataSet*를 사용하기 위한 속성 및 메소드는 20-35 페이지의 "SQL 클라이언트 데이터셋 사용"에서 설명합니다.

## 클라이언트 데이터셋을 사용하는 데이터 작업

---

데이터셋과 마찬가지로 클라이언트 데이터셋은 데이터 소스 컴포넌트를 사용하여 data-aware 컨트롤에 데이터를 제공할 수 있습니다. data-aware 컨트롤에 데이터베이스 정보를 표시하는 방법에 대한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

*TDataSet*의 자손으로서 클라이언트 데이터셋은 모든 데이터셋 컴포넌트에 정의된 강력하고 유용한 속성, 메소드, 이벤트들을 상속받습니다. 이러한 일반적인 데이터셋 행동에 대한 전체적인 내용은 16장 "데이터셋 이해"를 참조하십시오.

클라이언트 데이터셋은 모든 데이터를 메모리에 보관한다는 점에서 다른 데이터셋과 다릅니다. 이러한 이유에서 공통적인 데이터베이스 기능을 지원하기 위해 추가적인 기능이나 고려사항이 필요할 수도 있습니다.

## 클라이언트 데이터셋에서 데이터 탐색

---

애플리케이션에서 표준 data-aware 컨트롤을 사용한다면 사용자는 이러한 컨트롤의 기본 제공 행동을 사용하여 클라이언트 데이터셋의 레코드를 탐색할 수 있습니다. 또한 프로그램에서 *First*, *Last*, *Next*, *Prior* 같은 표준 데이터셋 메소드를 사용하여 레코드를 탐색할 수 있습니다. 이 메소드에 대한 자세한 내용은 16-9 페이지의 "데이터셋 탐색"을 참조하십시오.

*Locate*와 *Lookup* 메소드 역시 *TDataSet*에서 상속되었으며, 특정 필드의 값으로 특정 레코드를 탐색합니다. 이러한 메소드는 16-14 페이지의 "데이터셋 검색"에서 설명합니다.

클라이언트 데이터셋에는 특정 레코드를 표시하고 탐색하기 위해 *TDataSet*의 표준 북마크 기능이 구현되어 있습니다. 북마크에 대한 자세한 내용은 16-13 페이지의 "레코드 표시 및 반환"을 참조하십시오.

*TDataSet*에 있는 이러한 메소드 이외에도 클라이언트 데이터셋은 인메모리 레코드를 저장하고 인덱싱하는 여러 추가적인 탐색 메소드를 사용합니다.

예를 들어 북마크를 사용하는 대신 북마크를 저장할 메모리를 할당 및 해제하는 오버헤드로 클라이언트 데이터셋은 *RecNo* 속성을 사용하여 어떠한 특정 레코드에도 커서를 둘 수 있습니다. 다른 데이터셋은 *RecNo*를 사용하여 현재 레코드의 레코드 번호를 알

아낼 수 있지만 클라이언트 데이터셋은 *RecNo*를 특정 레코드 번호로 설정하여 이 레코드를 현재 레코드로 만들 수도 있습니다.

그러나 *TClientDataSet*이 가지고 있는 가장 강력한 메소드는 인덱스된 필드로 레코드를 검색하는 *Goto* 및 *Find* 검색 메소드입니다. 클라이언트 데이터셋에 대해 사용자가 정의한 인덱스를 명시적으로 사용함으로써 클라이언트 데이터셋은 *Locate* 메소드와 *Lookup* 메소드에서 제공하는 검색 성능을 향상시킬 수 있습니다.

다음 표는 애플리케이션에서 레코드를 검색하는데 사용할 수 있는 *Goto* 메소드와 *Find* 메소드와 관련된 6개의 메소드를 요약한 것입니다.

**표 20.1** 인덱스 기반 검색 메소드

| 메소드                | 용도                                                                                                |
|--------------------|---------------------------------------------------------------------------------------------------|
| <i>EditKey</i>     | 검색 키 버퍼의 현재 내용을 보관하고 데이터셋을 <i>dsSetKey</i> 상태로 두어 사용하는 애플리케이션에서 검색을 수행하기 전에 기존 검색 조건을 수정할 수 있습니다. |
| <i>FindKey</i>     | <i>SetKey</i> 메소드와 <i>GotoKey</i> 메소드를 하나의 메소드로 결합합니다.                                            |
| <i>FindNearest</i> | <i>SetKey</i> 메소드와 <i>GotoNearest</i> 메소드를 하나의 메소드로 결합합니다.                                        |
| <i>GotoKey</i>     | 데이터셋에서 검색 조건에 정확하게 일치하는 첫 번째 레코드를 검색하고 찾은 레코드로 커서를 이동합니다.                                         |
| <i>GotoNearest</i> | 문자열 기반 필드에서 부분 키 값에 기반하여 가장 가깝게 일치하는 레코드를 검색하고 해당 레코드로 커서를 이동합니다.                                 |
| <i>SetKey</i>      | 검색 키 버퍼를 지우고 데이터셋을 <i>dsSetKey</i> 상태로 두어 사용하는 애플리케이션에서 검색을 수행하기 전에 새로운 검색 조건을 지정할 수 있습니다.        |

*GotoKey* 및 *FindKey*는 부울 함수로, 검색이 성공하면 일치하는 레코드로 커서가 이동하고 *True*를 반환합니다. 검색이 실패하는 경우 커서는 이동하지 않으며 *False*를 반환합니다.

*GotoNearest* 및 *FindNearest*는 정확하게 일치하는 첫 번째 레코드나, 일치하는 레코드가 없는 경우에는 지정한 검색 조건보다 큰 첫 번째 레코드로 커서를 이동합니다.

## 검색에 사용할 인덱스 지정

*Goto* 및 *Find* 검색 메소드를 사용하기 전에 검색 속도를 높이도록 인덱스나 키를 정의해야 합니다. 사용자의 클라이언트 데이터셋에 인덱스가 이미 있는 경우에는 이 인덱스를 *IndexName* 속성을 사용하여 현재 사용할 수 있도록 만들어야 합니다. 예를 들어, 클라이언트 데이터셋에 사용자가 값을 검색하고자 하는 "CityIndex"라는 이름의 인덱스가 있다면 *IndexName* 속성을 "CityIndex"라고 설정해야 합니다.

```
ClientDataSet1.Close;
ClientDataSet1.IndexName := 'CityIndex';
ClientDataSet1.Open;
ClientDataSet1.SetKey;
ClientDataSet1['City'] := Edit1.Text;
ClientDataSet1.GotoNearest;
```

인덱스 이름을 지정하는 대신, *IndexFieldNames* 속성에 키로 사용할 필드를 나열할 수 있습니다.

클라이언트 데이터셋에서 인덱스를 만들고 사용하는 것에 대한 자세한 내용은 20-16 페이지의 "정렬과 인덱싱"을 참조하십시오.

## Goto 메소드로 검색 수행

*Goto* 메소드를 사용하여 검색을 수행하려면 다음과 같은 일반적인 단계를 따릅니다.

- 1 검색을 위해 사용할 인덱스를 지정합니다 (위에서 설명한대로).
- 2 클라이언트 데이터셋을 엽니다.
- 3 *SetKey*를 이용하여 데이터셋을 *dsSetKey* 상태로 둡니다.
- 4 검색할 값을 *Fields* 속성에 지정합니다. *Fields*는 *TFields* 객체로서, 열에 해당하는 서수를 지정함으로써 액세스 가능한 필드 컴포넌트의 인덱스 목록을 유지 관리합니다. 데이터셋의 첫 번째 열 번호는 0입니다.
- 5 *GotoKey* 또는 *GotoNearest*로 찾은 첫 번째 일치하는 레코드를 검색하여 이동합니다.

예를 들어, 버튼의 *OnClick* 이벤트에 추가된 다음 코드는 인덱스의 첫 번째 필드의 값이 편집 상자의 텍스트와 정확하게 일치하는 첫 번째 레코드로 이동합니다.

```
procedure TSearchDemo.SearchExactClick(Sender:TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

*GotoNearest*도 이와 비슷하며, 부분 필드 값과 가장 가깝게 일치하는 값을 검색합니다. 문자열 필드에서만 사용될 수 있습니다. 예를 들면, 다음과 같습니다.

```
ClientDataSet1.SetKey;
ClientDataSet1.Fields[0].AsString := 'Sm';
ClientDataSet1.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 메소드로 검색 수행

*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 검색할 인덱스를 지정합니다(위에서 설명한대로).
- 2 클라이언트 데이터셋을 엽니다.
- 3 *FindKey* 또는 *FindNearest*를 사용하여 첫 번째 레코드 또는 가장 가까운 레코드를 검색하여 이동합니다. 두 메소드에는 하나의 매개변수, 필드 값의 검색으로 분리된 목록이 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

기본적으로는 검색이 성공하면 검색 조건에 일치하는 첫 번째 레코드로 커서가 이동합니다. 원한다면 *KeyExclusive* 속성을 *True*로 설정하여 첫 번째 일치하는 레코드의 다음 레코드로 커서를 이동할 수도 있습니다.

기본적으로 *KeyExclusive*는 *False*이며, 이는 검색이 성공하면 첫 번째 일치하는 레코드에 커서가 이동한다는 의미입니다.

## 부분 키 검색

클라이언트 데이터셋에 두 개 이상의 키 열이 있고 이 키의 서브셋에서 값을 검색하려는 경우에는 *KeyFieldCount*를 검색할 열의 번호로 설정합니다. 예를 들어, 클라이언트 데이터셋의 현재 인덱스에 세 개의 열이 있고 첫 번째 열만 사용하여 값을 검색하려면 *KeyFieldCount*를 1로 설정합니다.

여러 열 키를 가진 클라이언트 데이터셋의 경우 첫 번째 열부터 시작하여 이웃하는 열의 값만을 검색할 수 있습니다. 예를 들면, 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* 또는 *FindKey*가 호출될 때마다 *Fields* 속성에 있는 이전 값이 모두 지워집니다. 이전에 설정된 필드를 사용하여 검색을 반복하거나 또는 검색에 사용되는 필드를 추가하려면 *SetKey*와 *FindKey* 대신 *EditKey*를 호출하십시오. 예를 들어, "CityIndex" 인덱스의 City 필드를 기반으로 이미 검색을 수행했다고 가정해 보십시오. 계속해서 "CityIndex"는 City 필드와 Company 필드를 모두 포함한다고 가정해 보십시오. 특정 도시에 있는 특정 회사 이름으로 레코드를 찾으려면 다음과 같은 코드를 사용합니다.

```
ClientDataSet1.KeyFieldCount := 2;
ClientDataSet1.EditKey;
ClientDataSet1['Company'] := Edit2.Text;
ClientDataSet1.GotoNearest;
```

## 레코드 표시 제한

사용 가능한 데이터의 서브셋을 사용하도록 하는 임시 제한을 두기 위한 방법으로 범위와 필터를 사용할 수 있습니다. 범위나 필터를 사용하면 클라이언트 데이터셋은 인메모리 캐시에 있는 모든 데이터를 보여 주지 않고 범위나 필터 조건에 맞는 데이터만을 표시합니다.

필터는 *TDataSet* 클래스에 있으며 클라이언트 데이터셋은 물론 사용자 지정 데이터셋에 잠재적으로 적용됩니다. 필터는 단방향 데이터셋에는 구현되지 않습니다. 필터 사용에 대한 자세한 내용은 16-17 페이지의 "필터를 사용한 데이터 서브셋 표시 및 편집"을 참조하십시오.

범위는 *TClientDataSet* 컴포넌트에만 적용됩니다. 범위와 필터는 서로 유사하지만 다른 용도를 가집니다. 다음 항목에서는 범위와 필터의 차이점과 범위를 사용하는 방법에 대해 설명합니다.

## 범위와 필터의 차이점 이해

범위와 필터는 보이는 레코드를 모든 사용 가능 레코드의 서브셋으로 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 찾는 데 사용할 인덱스를 지정할 때 *IndexName* 또는 *IndexFieldNames* 속성을 사용하여 범위를 정의할 인덱스를 할당할 수 있습니다.

반면, 필터는 인덱스에 상관없이 특정 데이터를 공유하는 레코드 집합을 말합니다. 예를 들어, 캘리포니아에 살면서 회사에 근무한지 5년 이상되는 직원들을 모두 표시하도록 직원 데이터베이스를 필터링할 수 있습니다. 필터에서 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 데이터셋의 경우에는 클라이언트 데이터셋의 인메모리 캐시를 채우기 전에 쿼리의 WHERE 절로 데이터를 선택(select) 하는 것이 훨씬 더 효과적일 수 있습니다.

## 범위 지정

클라이언트 데이터셋의 범위를 지정하는 데는 다음과 같은 두 가지 상호 배타적인 방법이 있습니다.

- *SetRangeStart*와 *SetRangeEnd*를 사용하여 시작과 끝을 각각 지정합니다.
- *SetRange*를 사용하여 시작과 끝을 한꺼번에 지정합니다.

## 범위의 시작 설정

*SetRangeStart* 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 시작 값 목록을 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 이름의 클라이언트 데이터셋을 사용하며 사용자가 *Customers* 데이터셋의 각 필드에 대해 영구적(persistent) 필드 컴포넌트를 작성했다고 가정해 보십시오. 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 생성하고 적용하는 데 사용할 수 있습니다.

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

이 코드는 *Fields*에 값을 할당하기 전에 *EndVal*에 입력된 텍스트가 Null이 아님을 확인합니다. *StartVal*에 입력된 텍스트가 Null인 경우 어떠한 값도 Null보다는 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 20-9 페이지의 "범위의 적용 또는 취소"를 참조하십시오.

## 범위의 끝 설정

*SetRangeEnd* 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 끝 값 목록을 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 지정하지 않으면 Kylix에서는 범위의 끝 값을 Null 값으로 간주합니다. 끝 값이 Null인 범위는 항상 비어 있습니다.

끝 값을 할당하는 가장 쉬운 방법은 *FieldByName* 메소드를 호출하는 것입니다. 예를 들어, 다음과 같습니다.

```
with ClientDataSet1 do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

범위 값의 시작을 지정할 때 인덱스에 없는 필드에 대해 값을 설정하고자 하면 데이터셋은 예외가 발생합니다.

범위의 끝 지정을 완료하려면 범위를 적용하거나 취소하십시오. 범위 적용 및 취소에 대한 자세한 내용은 20-9 페이지의 "범위의 적용 또는 취소"를 참조하십시오.

## 시작 및 끝 범위 값 설정

*SetRangeStart*와 *SetRangeEnd*를 각각 호출하여 범위 경계를 지정하는 대신 데이터셋을 *dsSetKey* 상태로 두고 *SetRange* 프로시저를 한 번 호출하여 범위의 시작과 끝 값을 설정할 수 있습니다.

*SetRange*는 시작 값 집합과 끝 값 집합의 두 개의 상수 배열 매개변수를 사용합니다. 예를 들어 다음과 같은 문장은 2열 인덱스를 기반으로 범위를 지정합니다.

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

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 생략하려면 생략된 필드에 Null을 전달하십시오.

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 기반하여 범위 지정

키가 하나 이상의 문자열 필드로 구성되는 경우 *SetRange* 메소드는 부분 키를 지원합니다. 예를 들어 인덱스가 *LastName*과 *FirstName* 열에 기반하는 경우 다음과 같은 범위 지정이 유효합니다.

```
ClientDataSet1.SetRangeStart;
ClientDataSet1['LastName'] := 'Smith';
ClientDataSet1.SetRangeEnd;
ClientDataSet1['LastName'] := 'Zzzzzz';
ClientDataSet1.ApplyRange;
```

이 코드는 *LastName*이 "Smith"보다 크거나 같은 범위의 모든 레코드를 포함시킵니다. 다음과 같은 값 지정도 가능합니다.

```
ClientDataSet1['LastName'] := 'Sm';
```

이 문장은 *LastName*이 "Sm"보다 크거나 같은 레코드를 포함시킵니다.

## 경계 값과 일치하는 레코드 포함 또는 제외

기본적으로 범위에는 지정된 시작 범위보다 크거나 같은 레코드와 지정된 끝 범위보다 작거나 같은 모든 레코드가 포함됩니다. 이 행동은 *KeyExclusive* 속성에 의해 제어됩니다. *KeyExclusive*는 기본값이 *False*입니다.

원한다면 클라이언트 데이터셋의 *KeyExclusive* 속성을 *True*로 설정하여 끝 범위와 동일한 레코드를 제외할 수 있습니다. 예를 들면, 다음과 같습니다.

```
ClientDataSet1.KeyExclusive := True;
ClientDataSet1.SetRangeStart;
ClientDataSet1['LastName'] := 'Smith';
ClientDataSet1.SetRangeEnd;
ClientDataSet1['LastName'] := 'Tyler';
ClientDataSet1.ApplyRange;
```

이 코드는 *LastName*의 범위가 "Smith"보다 크거나 같고 "Tyler"보다 작은 모든 레코드를 포함시킵니다.

## 범위 수정

다음 두 함수를 사용하면 범위에 대한 기존의 경계 조건을 수정할 수 있습니다. *EditRangeStart*는 범위의 시작 값을 변경할 경우에 사용하고 *EditRangeEnd*는 범위의 끝 값을 변경할 경우에 사용합니다.

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. 데이터셋에 현재 적용된 범위의 경계 조건을 수정하는 경우 *ApplyRange*를 다시 호출하기 전까지는 변경 내용이 적용되지 않습니다.

## 범위의 시작 편집

*EditRangeStart* 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 만들고 범위 시작 값의 현재 목록을 수정하기 시작합니다. 일단 *EditRangeStart*를 호출하면 *Fields* 속성에 대한 다음 할당은 범위를 적용할 때 사용하는 현재 인덱스 값을 덮어씁니다.

**팁** 처음에 부분 키에 기반하여 시작 범위를 만든 경우 *EditRangeStart*를 사용하여 범위의 시작 값을 확장할 수 있습니다. 부분 키에 기반한 범위에 대한 자세한 내용은 20-8 페이지의 "부분 키에 기반하여 범위 지정"을 참조하십시오.

## 범위의 끝 편집

*EditRangeEnd* 프로시저를 호출하여 데이터셋을 *dsSetKey* 상태로 두고 범위의 끝 값 목록을 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 범위의 끝을 지정하기 위해 *EditRangeStart*를 호출하거나, *SetRangeEnd* 또는 *EditRangeEnd*를 호출할 경우 데이터셋은 *dsSetKey* 상태가 됩니다. 사용자가 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 발생하도록 하려면 *ApplyRange* 프로시저를 호출하십시오. *ApplyRange*는 사용자가 지정된 데이터셋의 서브셋에 있는 데이터를 보거나 액세스하는 것을 즉시 제한합니다.

## 범위 취소

*CancelRange* 메소드는 범위의 애플리케이션을 종료하고 전체 데이터셋에 대한 액세스를 복구합니다. 범위를 취소하면 데이터셋의 모든 레코드에 대한 액세스가 복구되더라도 해당 범위의 경계 범위 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.

```
...
ClientDataSet1.CancelRange;
...
{later on, use the same range again. No need to call SetRangeStart, etc.}
ClientDataSet1.ApplyRange;
...
```

## 마스터/디테일 관계 표시

클라이언트 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.

클라이언트 데이터셋은 다음의 두 가지 방식 중 하나를 이용해서 마스터/디테일 관계를 지원합니다.

- 커서들을 연결하여 클라이언트 데이터셋을 다른 데이터셋의 디테일로 만듭니다. 이 프로세스는 아래에 있는 "클라이언트 데이터셋을 다른 데이터셋의 디테일로 만들기"에서 설명합니다.
- 중첩 디테일 테이블을 사용하여 클라이언트 데이터셋을 마스터/디테일 관계의 마스터로 만듭니다. 이 프로세스는 20-12 페이지의 "중첩 디테일 테이블 사용"에서 설명합니다.

이러한 방법은 저마다 고유한 장점이 있습니다. 커서를 연결하면 마스터 테이블이 클라이언트 데이터셋이 아닌 마스터/디테일 관계를 만들 수 있습니다. 중첩 디테일의 경우, 두 데이터셋 모두 클라이언트 데이터셋이어야 하지만 데이터를 표시하는 방법에 있어 더 많은 옵션을 제공하며 편집 내용을 다시 서버로 적용하는 데 더 적합합니다.

## 클라이언트 데이터셋을 다른 데이터셋의 디테일로 만들기

클라이언트 데이터셋의 *MasterSource*와 *MasterFields* 속성은 두 데이터셋 사이의 일대다 관계를 구축하는 데 사용할 수 있습니다.

*MasterSource* 속성은 클라이언트 데이터셋에서 마스터 테이블의 데이터를 가져오는 데이터 소스를 지정하는 데 사용됩니다. 이 데이터 소스는 데이터셋의 타입에 상관 없으며, 다른 클라이언트 데이터셋일 필요는 없습니다. 예를 들어 이 속성에 단방향 데이터셋의 데이터 소스 컴포넌트를 지정함으로써 클라이언트 데이터셋에서 단방향 데이터셋에서 발생하는 이벤트를 추적할 수 있도록 클라이언트 데이터셋을 단방향 데이터셋에 연결할 수 있습니다.

클라이언트 데이터셋은 현재 인덱스를 기반으로 마스터 테이블에 연결됩니다. (디테일) 클라이언트 데이터셋에서 추적하는 마스터 데이터셋의 필드를 지정하기 전에 해당 필드로 시작하는 클라이언트 데이터셋의 인덱스를 우선 지정하십시오. *IndexName* 속성 또는 *IndexFieldNames* 속성을 사용할 수 있습니다.

사용할 인덱스를 지정하고 나면 *MasterFields* 속성을 사용하여 (디테일) 클라이언트 데이터셋의 인덱싱된 필드에 해당하는 마스터 데이터셋의 열을 표시합니다. 데이터셋을 여러 열 이름으로 연결하려면 다음과 같이 세미콜론으로 구분하십시오.

```
ClientDataSet1.MasterFields := 'OrderNo;ItemNo';
```

두 데이터셋 사이의 의미 있는 연결을 돕는 Field Link 디자인어를 사용할 수 있습니다. Field Link 디자인어를 사용하려면 *MasterSource*와 인덱스를 할당한 후 Object Inspector의 *MasterFields* 속성을 더블 클릭하십시오.

다음 단계에 따라 사용자가 고객 레코드를 둘러보고 현재 고객에 대한 모든 순서를 표시할 수 있는 일반 폼이 작성됩니다. 마스터 데이터셋은 *CustomersTable*, 디테일 데이터셋은 *OrdersTable*이라고 합니다.

- 1 *TSQLConnection*, *TSQLDataSet* 컴포넌트, *TSQLClientDataSet* 및 두 개의 *TDataSource* 컴포넌트를 데이터 모듈에 둡니다.
- 2 *TSQLConnection* 컴포넌트의 속성을 다음과 같이 설정합니다.
  - *DriverName*: INTERBASE
  - *Params*:
    - *Database*: employee.gdb의 전체 경로 이름. 이 데이터베이스는 INTERBASE를 설치할 때 설치되어야 합니다.
    - *UserName*: 사용자 이름.
    - *Password*: 사용자 암호.
- 3 *TSQLDataSet* 컴포넌트의 속성을 다음과 같이 설정합니다.
  - *SQLConnection*: *SQLConnection1*
  - *CommandType*: *ctTable*
  - *CommandText*: CUSTOMER
  - *Name*: CustomersTable
- 4 첫 번째 *TDataSource* 컴포넌트의 속성을 다음과 같이 설정합니다.
  - *Name*: CustSource
  - *DataSet*: CustomersTable
- 5 *TSQLClientDataSet* 컴포넌트의 속성을 다음과 같이 설정합니다.
  - *DBConnection*: *SQLConnection1*
  - *CommandText*: 'Select \* from SALES'
  - *Name*: OrdersTable
  - *IndexFieldNames*: 'Cust\_No'
  - *MasterSource*: CustSource.

마지막 두 속성(*IndexFieldNames*와 *MasterSource*)은 주문 테이블의 CUST\_No 필드를 사용하여 CUSTOMER 테이블(마스터 테이블)을 ORDERS 테이블(디테일 테이블)에 연결합니다.
- 6 Object Inspector의 *MasterFields* 속성 값 상자를 더블 클릭하여 Field Link 디자이너를 호출하여 다음과 같은 속성을 설정합니다.
  - Detail Fields 필드 목록과 Master Fields 필드 목록 모두에서 *Cust\_No*를 선택합니다.
  - Add 버튼을 클릭하여 조인 조건을 추가합니다. Joined Fields 목록에 "CustNo -> CustNo"가 나타납니다.
  - 선택한 내용을 커밋하려면 OK를 선택하고 Field Link 디자이너를 종료합니다.
- 7 다음과 같이 두 번째 *TDataSource* 컴포넌트의 속성을 설정합니다.
  - *Name*: OrdersSource
  - *DataSet*: OrdersTable
- 8 *TDBText*, *TDBGrid*, *TButton* 을 폼에 둡니다.
- 9 File|Use Unit을 선택하여 폼에서 데이터를 사용한다고 지정합니다.

10 DB 그리드의 *DataSource* 속성을 "OrdersSource"로 설정합니다.

11 DB 텍스트 컨트롤에 다음과 같은 속성을 설정하십시오.

- *DataSource*: CustSource
- *DataField*: CUSTOMER.

12 버튼을 더블 클릭하고 버튼의 *OnClick* 이벤트 핸들러에 다음을 입력합니다.

```
CustomersTable.Next;
```

13 데이터를 폼에 표시하도록 *CustomersTable*과 *OrdersTable*의 *Active* 속성을 *True*로 설정합니다.

14 애플리케이션을 컴파일하여 실행합니다.

이제 애플리케이션을 실행하면 데이터셋이 함께 연결되어 있고, DB 텍스트 컨트롤에 새로운 고객 이름을 표시하기 위해 버튼을 누르면 현재 고객이 속하는 ORDERS 테이블에서 해당 레코드만 표시됩니다.

## 중첩 디테일 테이블 사용

중첩 테이블을 사용하여 클라이언트 데이터셋에 마스터/디테일 관계를 설정하는 방법에는 다음 두 가지가 있습니다.

- 프로바이더 컴포넌트에서 중첩 디테일을 포함하는 레코드를 가져옵니다. 프로바이더 컴포넌트가 마스터/디테일 관계의 마스터 테이블을 나타내는 경우 각 레코드의 디테일을 나타내는 중첩 데이터셋 필드를 자동으로 만듭니다. 이 메소드는 *TClientDataSet*에만 사용되는데, 그 이유는 사용자가 *TSQLClientDataSet*을 사용할 때 마스터 테이블을 나타내는 별도의 프로바이더 컴포넌트를 설정할 수 없기 때문입니다.
- Fields Editor를 사용하여 중첩 디테일을 정의합니다. 디자인 타임에 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하여 Fields Editor를 선택합니다. Add Fields를 마우스 오른쪽 버튼으로 클릭하여 사용자의 클라이언트 데이터셋에 새로운 영구적 필드를 추가합니다. DataSet Field 타입의 새로운 필드를 정의합니다. Fields Editor에서 사용자의 디테일 테이블 구조를 정의합니다.

**참고** 중첩 디테일 셋을 사용하려면 클라이언트 데이터셋의 *ObjectView* 속성이 *True*여야 합니다.

클라이언트 데이터셋에 중첩 디테일 데이터셋이 포함된 경우 *TDBGrid*에서 중첩 디테일을 팝업 메뉴에 표시하도록 지원합니다. 이 방법은 17-25 페이지의 "데이터셋 필드 표시"에서 다룹니다.

그 대신, 디테일 셋에 대해 별도의 클라이언트 데이터셋을 사용하여 data-aware 컨트롤에 이러한 데이터셋을 표시하거나 편집할 수 있습니다. 디자인 타임에 중첩 디테일 셋에 대한 DataSet 필드를 비롯한 (마스터) 클라이언트 데이터셋에서 필드에 대한 영구적 필드를 만드십시오.

이제 중첩 디테일 셋을 나타내는 클라이언트 데이터셋을 만들 수 있습니다. 이 디테일 셋의 *DataSetField* 속성을 마스터 데이터셋의 영구적 dataSet 필드로 설정하십시오.

마스터 테이블과 디테일 테이블에서 데이터베이스 서버로 업데이트 내용을 적용하려면 중첩 디테일 셋을 사용해야 합니다. 파일 기반 데이터베이스 애플리케이션에서 중첩 디



테일 셋을 사용하면 사용자가 두 데이터셋을 개별적으로 로드하지 않아도 마스터 레코드가 있는 디테일을 한 파일에 저장할 수 있으며, 그런 다음 인덱스를 다시 생성하여 마스터/디테일 관계를 다시 만들 수 있습니다.

## 데이터 값 제약 조건

클라이언트 데이터셋에서 데이터를 편집할 때 사용자가 입력할 수 있는 값을 제한할 수 있습니다. 사용자 편집 내용에 부과할 수 있는 제약 조건에는 필드 수준 제약 조건과 레코드 수준 제약 조건의 두 가지 종류가 있습니다.

각 필드 컴포넌트에는 다음과 같이 제약 조건을 지정하는 데 사용할 수 있는 두 가지 속성이 있습니다.

- *DefaultExpression* 속성은 사용자가 값을 입력하지 않은 경우에 필드에 할당되는 기본값을 정의합니다. 데이터베이스 서버나 소스 데이터셋에도 필드에 대해 기본 표현식을 할당하는 경우 업데이트 내용이 프로바이더로 반환되기 전에 할당되기 때문에 클라이언트 데이터셋의 버전이 우선한다는 점에 유의하십시오.
- *CustomConstraint* 속성을 사용하면 필드 값을 포스트하기 전에 만족해야 하는 제약 조건을 할당할 수 있습니다. 사용자 지정 제약 조건은 데이터를 서버로 전송하기 전에 데이터를 확인할 때 유용합니다. 예를 들어 데이터베이스 서버에서 부과한 제약 조건을 복제하고자 합니다. 이렇게 하면, 서버 제약 조건을 만족하지 않는 사용자의 편집 내용이 클라이언트 측에 적용되므로, 편집 내용이 데이터베이스 서버에서 기각되어 편집 내용이 결코 전달되지 않습니다. 이는 업데이트가 멈추면 업데이트 프로세스 동안 오류 조건이 발생한다는 의미입니다. 필드 컴포넌트의 사용자 지정 제약 조건에 대한 자세한 내용은 17-20 페이지의 "제약 조건 지정"을 참조하십시오.

레코드 수준에서 클라이언트 데이터셋의 *Constraints* 속성을 사용하여 제약 조건을 지정할 수 있습니다. *Constraints*는 *TCheckConstraint* 객체의 컬렉션으로, 각 객체는 각각의 조건을 나타냅니다. *TCheckConstraint* 객체의 *CustomConstraint* 속성을 사용하여 레코드를 포스트할 때 검사하는 사용자가 만든 제약 조건을 추가하십시오.

*TSQLClientDataSet*을 사용하거나 프로바이더 컴포넌트가 있는 *TClientDataSet*을 사용하여 데이터베이스 서버에서 데이터를 페치(fetch)할 때 데이터 제약 조건 실행을 중단해야 하는 경우, 특히 클라이언트 데이터셋에 소스 데이터셋(또는 데이터베이스 서버)의 모든 레코드가 포함되어 있지 않은 경우가 종종 있습니다. 예를 들어 서버 제약 조건이 필드의 현재 최대 값에 기반을 둔 것이지만 클라이언트 데이터셋에서 레코드 패킷을 여러 개 페치하는 경우, 클라이언트 데이터셋에 있는 필드의 현재 최대 값이 데이터베이스 서버의 최대 값과 다를 수 있으며 제약 조건이 다르게 실행될 수 있습니다. 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 사용 불가능으로 만들 수 있습니다.

제약 조건을 일시적으로 사용 불가능으로 만들려면 클라이언트 데이터셋의 *DisableConstraints* 메소드를 호출하십시오. *DisableConstraints*가 호출될 때마다 참조 카운트가 늘어납니다. 참조 카운트가 0보다 크면 클라이언트 데이터셋에서 제약 조건이 실행되지 않습니다.

클라이언트 데이터셋의 제약 조건을 다시 사용 가능하도록 만들려면 데이터셋의 *Enable Constraints* 메소드를 호출하십시오. *EnableConstraints*를 호출할 때마다 참조 카운트가 줄어듭니다. 참조 카운트가 0이면 제약 조건이 다시 사용 가능하게 됩니다.

- 팁** 의도한대로 제약 조건이 사용 가능하게 되었는지를 확인하려면 항상 *Disable Constraints*와 *EnableConstraints*를 쌍으로 호출하십시오.

## 데이터를 읽기 전용으로 만들기

*TDataSet*에는 애플리케이션에서 데이터셋의 데이터 편집여부를 결정할 수 있게 하는 *CanModify* 속성이 있습니다. 단방향 데이터셋 같은 일부 *TDataSet* 자손에서는 편집을 포스트하기 위한 어떠한 기본 제공 지원도 갖고 있지 않기 때문에 애플리케이션은 *CanModify* 속성을 변경할 수 없습니다.

그러나, 클라이언트 데이터셋은 인메모리(in-memory) 데이터를 나타내기 때문에 사용자가 해당 데이터를 편집할 수 있는지 여부를 애플리케이션에서 항상 제어할 수 있습니다. 하지만, 서버 테이블이 읽기 전용이라면 해당 애플리케이션에서 업데이트를 데이터베이스 서버에 포스트할 수 없습니다. 사용자가 데이터를 수정할 수 없도록 하려면 클라이언트 데이터셋의 *ReadOnly* 속성을 *True*로 설정하십시오. *ReadOnly*를 *True*로 설정하면 *CanModify* 속성이 *False*로 설정됩니다.

읽기 전용 상태를 변경하기 위해 클라이언트 데이터셋을 닫지 않아도 됩니다. 애플리케이션에서 현재의 *ReadOnly* 속성의 설정을 변경하기만 하면 언제라도 클라이언트 데이터셋을 읽기 전용으로 만들 수도 있고 읽기 전용을 해제할 수도 있습니다.

## 데이터 편집

클라이언트 데이터셋은 자체 데이터를 인메모리 데이터 패킷으로 나타냅니다. 이 패킷은 클라이언트 데이터셋의 *Data* 속성의 값입니다. 그러나 기본적으로 편집 내용은 *Data* 속성에 저장되지 않습니다. 대신 사용자나 프로그램에 의한 추가, 삭제, 수정 내용은 *Delta* 속성으로 나타내는 내부 변경 로그에 저장됩니다. 변경 로그는 다음과 같이 두 가지 용도로 사용됩니다.

- 프로바이더 작업 시, 업데이트를 서버에 적용하기 위한 메커니즘에서 변경 로그가 필요합니다.
- 모든 애플리케이션에서 변경 로그는 변경 취소를 위한 정교한 지원을 제공합니다.

*LogChanges* 속성을 사용하면 로그를 일시적으로 사용 불가능하게 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 사용 불가능으로 만들 수 있습니다.

변경 로그의 편집 내용은 애플리케이션에서 삭제하기 전까지 계속 변경 로그에 남아 있습니다. 애플리케이션에서는 다음의 경우 편집 내용을 삭제합니다.

- 변경 취소 시
- 변경 저장 시

**참고** 클라이언트 데이터셋을 파일에 저장하면 편집 내용은 변경 로그에서 삭제되지 않습니다. 데이터셋을 다시 로드할 때 *Data* 속성과 *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 레코드의 변경된 각 버전은 개별 항목으로 변경 로그에 저장됩니다.

레코드에 대한 각 변경을 저장하면 레코드의 이전 상태를 복구해야 할 경우 여러 단계의 실행 취소 연산을 지원할 수 있게 됩니다.

- 레코드의 마지막 변경을 삭제하려면 *UndoLastChange*를 호출하십시오. *UndoLastChange*에는 부울 매개변수로 *FollowChange*가 필요하며, 이는 커서의 위치를 복구된 레코드에 다시 돌지 (*True*) 아니면 현재 레코드에 그냥 남겨둘지 (*False*) 여부를 나타냅니다. 하나의 레코드를 여러 번 변경한 경우 *UndoLastChange*를 호출할 때마다 단계적으로 변경 내용이 삭제됩니다. *UndoLastChange*는 변경 삭제의 성공 또는 실패를 나타내는 부울 값을 반환합니다. 삭제가 발생하면 *UndoLastChange*는 *False*를 반환합니다. *ChangeCount* 속성을 사용하여 실행 취소할 변경은 더 없는지 알 수 있습니다. *ChangeCount*는 변경 로그에 저장된 변경 횟수를 의미합니다.
- 하나의 레코드의 각 변경 내용을 삭제하는 대신 한 번에 모두 삭제할 수 있습니다. 레코드의 모든 변경 내용을 삭제하려면 레코드를 선택하고 *RevertRecord*를 호출하십시오. *RevertRecord*는 변경 로그에서 현재 레코드에 대한 모든 변경을 삭제합니다.
- 편집 도중 언제든지 *SavePoint* 속성을 사용하여 변경 로그의 현재 상태를 저장할 수 있습니다. *SavePoint*를 읽으면 마커를 변경 로그의 현재 위치로 반환합니다. 나중에 저장 위치 (save point)를 읽은 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 호출하면 변경 로그에 기록된 모든 변경을 취소할 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 서버의 데이터를 나타내는 지 여부에 따라 변경 로그의 변경 내용을 통합하는데 있어 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 합병할 수 있습니다. 또한 다른 사용자에게 의해 이루어진 변경 내용으로 로컬 편집을 해석하는데 신경 쓸 필요가 없습니다. 변경 로그를 *Data* 속성으로 합병하려면 *MergeChangeLog* 메소드를 호출하십시오. 20-41 페이지의 "데이터에 변경 내용 합병"에서 이 프로세스에 대해 설명합니다.

서버 데이터를 나타내는 클라이언트 데이터셋을 사용하고 있다면 *MergeChangeLog*를 사용할 수 없습니다. 업데이트된 레코드가 데이터베이스(또는 소스 데이터셋)에 저장된 데이터로 해석될 수 있도록 변경 로그의 정보가 필요합니다. 그 대신, 변경 내용을 데이터베이스 서버(또는 소스 데이터셋)에 전달하는 *ApplyUpdates*를 호출하고 수정 내용이 성공적으로 포스트되고 난 경우에만 *Data* 속성을 업데이트합니다. 이 프로세스에 대한 자세한 내용은 20-31 페이지의 "업데이트 적용"을 참조하십시오.

## 정렬과 인덱싱

인덱스를 사용하면 여러 가지 장점이 있습니다.

- 클라이언트 데이터셋에서 데이터를 빨리 찾을 수 있습니다.
- 사용 가능 레코드를 제한하는 범위를 적용할 수 있습니다.
- 애플리케이션에서 조회 테이블이나 마스터/디테일 연결과 같은 클라이언트 데이터셋 사이의 관계를 설정할 수 있습니다.
- 레코드가 나타나는 순서를 지정합니다.

클라이언트 데이터셋에서 프로바이더(*TSQLClientDataSet*에서 사용하는 내부 프로바이더 포함)를 사용하는 경우는 기본 인덱스를 상속하고 프로바이더에서 수신하는 데이터에 따라 순서를 정렬합니다. 기본 인덱스를 *DEFAULT\_ORDER*라고 합니다. 이 순서를 사용할 수는 있지만 인덱스를 변경하거나 삭제할 수 없습니다.

기본 인덱스 이외에도 클라이언트 데이터셋은 변경 로그(*Delta* 속성)에 저장된 변경된 레코드에서 *CHANGEINDEX*라는 보조 인덱스를 유지합니다. *CHANGEINDEX*는 클라이언트 데이터셋의 모든 레코드를 *Delta*에 지정된 변경 내용이 적용되었을 경우 나타나는 대로 정렬합니다. *CHANGEINDEX*는 *DEFAULT\_ORDER*에서 상속된 순서에 기반합니다. *DEFAULT\_ORDER*에서 *CHANGEINDEX* 인덱스를 변경하거나 삭제할 수 없습니다.

데이터셋에 대한 기존의 다른 인덱스를 사용할 수 있으며 직접 인덱스를 만들 수 있습니다. 다음 단원에서 클라이언트 데이터셋에서 인덱스를 만들고 사용하는 방법을 설명합니다.

## 새 인덱스 추가

클라이언트 데이터셋에 인덱스를 추가하는 방법에는 세 가지가 있습니다.

- 클라이언트 데이터셋에 레코드를 정렬하는 임시 인덱스를 런타임에 만들기 위해 *IndexFieldNames* 속성을 사용할 수 있습니다. 세미콜론으로 구분하여 필드 이름을 지정합니다. 목록에 있는 필드 이름 순서대로 인덱스에서의 순서가 결정됩니다.

이 메소드는 인덱스를 추가하는 가장 약한 메소드입니다. 내림차순 또는 대소문자 구별 인덱스를 지정할 수 없으며 결과 인덱스에서 그룹화를 지원하지 않습니다. 이 인덱스들은 데이터셋을 닫으면 영구적으로 남지 않으며 클라이언트 데이터셋을 파일에 저장할 때 저장되지 않습니다.

- 런타임에 그룹화를 위해 사용될 수 있는 인덱스를 만들려면 *AddIndex*를 호출하십시오. *AddIndex*를 사용하면 다음을 포함하여 인덱스의 속성을 지정할 수 있습니다.
  - 인덱스 이름. 런타임 시 인덱스 전환을 위해 사용할 수 있습니다.
  - 인덱스를 구성하는 필드. 인덱스는 이 필드를 사용하여 레코드를 정렬하고 이 필드에서 특정 값을 가진 레코드를 찾습니다.
  - 인덱스에서 레코드를 정렬하는 방법. 기본적으로 인덱스는 컴퓨터의 로케일에 따라 오름차순으로 정렬됩니다. 기본 정렬 순서는 대소문자가 구별됩니다. 전체 인덱스를 대소문자로 구별할지, 내림차순으로 정렬할지 옵션을 지정할 수 있습니다. 또는 대소문자를 구별해서 정렬할 필드의 목록과 내림차순으로 정렬할 필드의 목록을 제공할 수 있습니다.
  - 인덱스에 대한 기본 수준의 그룹화 지원.

*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 생성하는 세 번째 방법은 클라이언트 데이터셋이 만들어질 때 생성된 인덱스를 사용하는 것입니다. 클라이언트 데이터셋을 만들기 전에 *IndexDefs* 속성을 사용하여 원하는 인덱스를 지정하십시오. 그러면 *CreateDataSet*을 호출할 때 원본으로 사용하는 데이터셋과 함께 인덱스가 만들어집니다. 자세한 내용은 20-39 페이지의 "필드와 인덱스 정의를 사용하여 데이터셋 생성"을 참조하십시오.

*AddIndex*에서 데이터셋 지원 그룹화로 만드는 인덱스는 일부 필드에서는 올림차순으로, 또 일부 필드에서는 내림차순으로 정렬이 가능하고, 일부 필드에서는 대소문자 구별이 불가능하며 또 일부 필드에서는 대소문자 구별이 가능합니다. 이러한 방법으로 생성되는 인덱스는 영구적으로 남으며 클라이언트 데이터셋을 파일에 저장할 때 저장됩니다.

**팁** 내부적으로 계산된 (calculated) 필드에서 클라이언트 데이터셋으로 인덱스를 만들고 정렬할 수 있습니다.

## 인덱스 삭제와 전환

클라이언트 데이터셋에 대해 만든 인덱스를 삭제하려면 *DeleteIndex*를 호출하고 삭제할 인덱스의 이름을 지정하십시오. DEFAULT\_ORDER 인덱스와 CHANGEINDEX 인덱스는 삭제할 수 없습니다.

둘 이상의 인덱스를 사용할 수 있을 때 클라이언트 데이터셋에서 다른 인덱스를 사용하려면 *IndexName* 속성을 사용하여 사용할 인덱스를 선택합니다. 디자인 타임에 Object Inspector의 *IndexName* 속성 드롭 다운 상자에 있는 사용할 수 있는 인덱스 중에서 선택할 수 있습니다.

## 인덱스 관련 정보 얻기

런타임 시 애플리케이션은 사용할 수 있는 인덱스에 대한 정보를 클라이언트 데이터셋으로 요청할 수 있습니다.

*GetIndexNames* 메소드는 테이블에 대해 사용할 수 있는 인덱스 목록을 검색합니다. *GetIndexNames*는 유효한 인덱스 이름으로 문자열 목록을 만듭니다. 예를 들어, 다음 코드는 *ClientDataSet1*에 대해 사용할 수 있는 인덱스 목록을 페치합니다.

```
var
  IndexList: TList;
begin
  ...
  IndexList := TStringList.Create;
  ClientDataSet1.GetIndexNames(IndexList);
```

또한, 사용자도 현재 인덱스의 필드 목록을 검사할 수 있습니다. 현재 인덱스의 모든 필드를 반복하려면 다음 두 속성을 사용하십시오.

- *IndexFieldCount* 속성. 이 속성은 인덱스의 열 숫자를 나타냅니다.
- *IndexFields* 속성. 이 속성은 인덱스를 구성하는 필드를 나열합니다.

*IndexFields*는 *TField* 컴포넌트의 배열이며 인덱스에서 각 필드는 컴포넌트 하나에 해당합니다. 다음 코드 부분은 현재 인덱스의 필드를 반복하기 위해 *IndexFieldCount*와 *IndexFields*를 사용하는 방법을 보여 줍니다.

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20] of string;
begin
  with ClientDataSet1 do
    begin
      for I := 0 to IndexFieldCount - 1 do
        ListOfIndexFields[I] := IndexFields[I].FieldName;
      end;
    end;
end;
```

## 인덱스를 사용하여 데이터 그룹화

클라이언트 데이터셋에서 인덱스를 사용할 경우 자동적으로 레코드에 정렬 순서를 부여합니다. 이 순서로 인해 이웃하는 레코드에는 대개 인덱스를 구성하는 필드에 값이 중복적으로 포함됩니다. 예를 들어 SalesRep 필드와 Customer 필드로 인덱스된 주문 테이블에서 다음 부분을 생각해 보십시오.

| 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 이웃하는 값이 중복됩니다. SalesRep 1의 레코드 내에서, Customer 열의 이웃하는 값이 중복됩니다. 즉, 데이터가 SalesRep에 의해 그룹화되며, SalesRep 그룹 내에서는 Customer에 의해 그룹화됩니다. 각 그룹화에는 관련 수준이 있습니다. 이 경우, SalesRep 그룹은 다른 그룹에 의해 중첩되지 않았으므로 수준 1이 되고, Customer 그룹은 수준 1인 그룹에 중첩되었으므로 수준 2가 됩니다. 그룹화 수준은 인덱스에서 필드의 순서에 해당합니다.

클라이언트 데이터셋은 특정 그룹화 수준 내에서 현재 레코드가 어디에 놓일지 결정하게 해줍니다. 이렇게 하면 그룹에서 첫 번째 레코드인지, 그룹의 중간에 있는지 또는 그룹의 마지막 레코드인지에 따라 애플리케이션에서 레코드를 다르게 표시할 수 있습니다. 예를 들어 필드 값이 그룹의 첫 번째 레코드에 있는 경우 중복 값을 삭제하여 필드 값을 표시만 하려고 합니다. 앞 테이블에 대해 이를 적용하면 다음과 같은 결과를 얻습니다.

| SalesRep | Customer | OrderNo | Amount |
|----------|----------|---------|--------|
| 1        | 1        | 5       | 100    |
|          |          | 2       | 50     |
|          | 2        | 3       | 200    |
|          |          | 6       | 75     |
| 2        | 1        | 1       | 10     |
|          | 3        | 4       | 200    |

현재 레코드가 그룹의 어느 위치에 있는지를 알아내려면 *GetGroupState* 메소드를 사용하십시오. *GetGroupState*는 그룹의 수준을 부여하는 정수를 사용하며 그룹에서 현재 레코드가 그룹의 어느 곳에 있는지(첫 번째 레코드, 마지막 레코드, 또는 사이 레코드) 알려주는 값을 반환합니다.

인덱스를 생성할 때 인덱스가 지원하는 그룹화 수준을 인덱스에 있는 필드 수만큼 지정할 수 있습니다. *GetGroupState*는 인덱스에서 추가 필드의 레코드를 정렬하더라도 해당 수준 이상으로 그룹에 대한 정보를 제공할 수는 없습니다.

## 계산된 값 표시

모든 데이터셋에서 클라이언트 데이터셋에 계산된(calculated) 필드를 추가할 수 있습니다. 이 필드는 동일한 레코드에 있는 다른 필드의 값에 따라 동적으로 계산할 수 있습니다. 계산된 필드 사용에 대한 자세한 내용은 17-7 페이지의 "계산된 필드 정의"를 참조하십시오.

그러나 클라이언트 데이터셋을 사용하면 내부적으로 계산된 필드를 사용하여 필드가 계산될 때 최적화할 수 있습니다. 내부적으로 계산된(calculated) 필드에 대한 자세한 내용은 다음에 나오는 "클라이언트 데이터셋에서 내부적으로 계산된 필드 사용"을 참조하십시오.

또한 유지 관리되는 집계계를 사용하여 클라이언트 데이터셋에서 일부 레코드의 데이터를 요약하는 계산된 값을 생성하도록 할 수 있습니다. 유지 관리되는 집계계에 대한 자세한 내용은 20-20 페이지의 "유지 관리되는 Aggregates 속성 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. 애플리케이션은 *OnCalcFields* 이벤트 핸들러에서 이를 수행합니다.

클라이언트 데이터셋을 사용하면 클라이언트 데이터셋의 데이터에 계산된 값을 저장하여 계산된 필드를 다시 계산해야 하는 횟수를 최소화할 수 있습니다. 계산된 값이 클라이언트 데이터셋으로 저장되면 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 생성하기 전에 필드를 내부적으로 계산된 필드로 정의해야 합니다. 영구적 필드를 사용하여 클라이언트 데이터셋을 만드는 경우, Fields Editor에서 InternalCalc를 선택하여 필드를 내부적으로 계산된 필드로 정의하십시오. 필드 정의를 사용하여 클라이언트 데이터셋을 만들려면, 관련된 필드 정의의 *InternalCalcField* 속성을 *True*로 설정하십시오.

**참고** 다른 종류의 데이터셋은 내부적으로 계산된 필드를 사용합니다. 그러나 다른 데이터셋의 경우 *OnCalcFields* 이벤트 핸들러에서 이 값들을 계산하지 마십시오. 그 대신, 원격 데이터베이스 서버에서 이 값들은 자동으로 계산됩니다.

## 유지 관리되는 Aggregates 속성 사용

클라이언트 데이터셋은 여러 레코드 그룹에 대해서 데이터를 요약할 수 있는 기능을 제공합니다. 이 요약들은 데이터셋에서 데이터를 편집할 때 자동적으로 업데이트되기 때문에 이 요약된 데이터를 "유지 관리되는 집계(Aggregates)"라고 합니다.

가장 단순한 형태의 유지 관리되는 집계를 사용하여 클라이언트 데이터셋의 열에 있는 모든 값의 합계 같은 정보를 얻을 수 있습니다. 하지만 유지 관리 되는 집계에서는 다양한 요약 계산 기능을 제공하며 그룹화를 지원하는 인덱스의 필드에 의해 정의되는 여러 레코드 그룹에 대한 요약을 제공할 수 있을 정도로 충분한 유연성을 갖추고 있습니다.

### Aggregates 지정

클라이언트 데이터셋의 레코드에 대한 요약을 계산하려면 *Aggregates* 속성을 사용하십시오. *Aggregates*는 집계 사양(*TAggregate*)의 컬렉션입니다. 디자인 타임에 Collection Editor를 사용하거나 또는 런타임에 *Aggregates*의 *Add* 메소드를 사용하여 클라이언트 데이터셋에 집계 사양을 추가할 수 있습니다. 집계에 대한 필드 컴포넌트를 생성하려면 Fields Editor에서 집계된 값에 대한 영구적 필드를 만드십시오.

**참고** 집계된 필드를 생성할 때 적절한 집계 객체가 클라이언트 데이터셋의 *Aggregates* 속성에 자동으로 추가됩니다. 집계된 영구적인 필드를 생성할 때는 명시적으로 이를 추가하



지 마십시오. 집계되는 영구적 필드 생성에 대한 자세한 내용은 17-10 페이지의 "집계 필드 정의"를 참조하십시오.

각 집계에서 *Expression* 속성은 집계가 나타내는 요약 계산을 나타냅니다. *Expression*에는 단순한 요약 표현식을 포함할 수 있습니다.

`Sum(Field1)`

여러 필드의 정보를 결합하는 복잡한 표현식을 포함할 수도 있습니다.

`Sum(Qty * Price) - Sum(AmountPaid)`

집계 표현식에는 표 20.2에 있는 하나 이상의 요약 연산자를 사용합니다.

**표 20.2** 유지 관리되는 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   |
| Min   | 문자열, 숫자, 날짜 시간 필드나 표현식의 최소값 |
| Max   | 문자열, 숫자, 날짜 시간 필드나 표현식의 최대값 |

요약 연산자는 필터 작성에 사용하는 연산자와 같은 연산자로 필드 값 또는 필드 값으로부터 만들어진 표현식에 사용합니다. 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.

|                                        |                        |
|----------------------------------------|------------------------|
| <code>Sum(Qty * Price)</code>          | {적합 -- 필드 상에서 표현식을 요약} |
| <code>Max(Field1) - Max(Field2)</code> | {적합 -- 요약의 표현식}        |
| <code>Avg(DiscountRate) * 100</code>   | {적합 -- 요약과 상수의 표현식}    |
| <code>Min(Sum(Field1))</code>          | {부적합 -- 중첩된 요약}        |
| <code>Count(Field1) - Field2</code>    | {부적합 -- 요약과 필드의 표현식}   |

## 레코드 그룹의 집계

유지 관리되는 집계는 기본적으로 클라이언트 데이터셋의 모든 레코드를 요약하도록 계산됩니다. 하지만 그 대신 그룹 안의 레코드에 대해서만 요약하도록 지정할 수 있습니다. 이렇게 하면 공통 필드값을 공유하는 레코드 그룹의 합계 같은 중간 요약을 제공할 수 있습니다.

레코드 그룹의 유지 관리되는 집계를 지정하기 전에 적절한 그룹화를 제공하는 인덱스를 이용해야 합니다. 그룹화 지원에 대한 자세한 내용은 20-18 페이지의 "인덱스를 사용하여 데이터 그룹화"를 참조하십시오.

요약하고자 하는 방식으로 데이터를 그룹화하는 인덱스를 만들었다면 어떤 인덱스를 사용하는지와 해당 인덱스의 어떤 그룹이 인덱스가 요약하는 레코드를 정의하는지를 나타내는 집계의 *IndexName*과 *GroupingLevel* 속성을 지정합니다.

예를 들어, SalesRep로 그룹화 되고, SalesRep에서는 Customer로 그룹화된 주문 테이블의 다음 부분을 봅시다.

| 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    |

다음 코드는 각 판매 대리점의 총 수량을 나타내는 유지 관리되는 집계를 설정합니다.

```

Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';

```

특정 판매 대리점 내의 각 고객에 대해 요약하는 집계를 추가하려면 수준 2의 유지 관리되는 집계를 만듭니다.

레코드 그룹을 요약하는 유지 관리되는 집계는 지정된 인덱스에 연결됩니다. *Aggregates* 속성은 다른 인덱스를 사용하는 집계를 포함할 수 있습니다. 그러나 전체 데이터셋을 요약하는 집계와 현재 인덱스를 사용하는 집계만이 유효합니다. 현재 인덱스를 변경하면 유효한 집계도 변경됩니다. 임의의 시간에 어떤 집계가 유효한지 결정하려면 *ActiveAggs* 속성을 사용합니다.

## 집계 값 얻기

유지 관리되는 집계의 값을 얻으려면 집계를 나타내는 *TAggregate* 객체의 *Value* 메소드를 호출합니다. *Value*는 클라이언트 데이터셋의 현재 레코드를 가지고 있는 그룹의 유지 관리되는 집계를 반환합니다.

전체 클라이언트 데이터셋을 요약하는 경우, 유지 관리되는 집계를 얻기 위해 언제라도 *Value*를 호출할 수 있습니다. 그러나 그룹화된 정보를 요약하는 경우 현재 레코드가 요약할 원하는 그룹 내에 있는지를 확인해야 합니다. 이 때문에 그룹의 첫 레코드로 이동할 때나 그룹의 마지막 레코드로 이동할 때처럼 명확하게 지정된 시기에 집계 값을 구하는 것이 좋습니다. 그룹 내에서 현재 레코드가 어디에 있는지 알아내기 위해서는 *GetGroupState* 메소드를 사용합니다.

유지 관리되는 집계를 data-aware 컨트롤에 표시하려면 Fields Editor를 사용하여 영구적인 집계 (aggregate) 필드 컴포넌트를 만듭니다. Fields Editor에서 집계 필드를 지정하는 경우 클라이언트 데이터셋의 *Aggregates*는 적절한 집계 사양을 포함하도록 자동으로 업데이트됩니다. *AggFields* 속성은 새로 집계되는 필드 컴포넌트를 포함하고 *FindField* 메소드는 이를 반환합니다.

## 데이터에 애플리케이션 특정 정보 추가

애플리케이션 개발자는 클라이언트 데이터셋의 *Data* 속성에 고객 정보를 추가할 수 있습니다. 이 정보는 데이터 패킷과 함께 제공되기 때문에 데이터를 파일이나 스트림에 저장할 때 포함됩니다. 그리고, 이 정보는 데이터를 다른 데이터셋에서 복사할 때 복사됩니다. 옵션으로서 클라이언트 데이터셋에서 업데이트를 받을 때 이 정보를 프로바이더가 읽을 수 있도록 *Delta* 속성과 함께 포함될 수 있습니다.

*Data* 속성과 함께 애플리케이션 특정 정보를 저장하려면 *SetOptionalParam* 메소드를 사용합니다. 이 메소드를 통해 특정 이름의 데이터를 포함하는 *OleVariant*를 저장할 수 있습니다.

이 애플리케이션 특정 정보를 검색하려면 정보를 저장할 때 사용한 이름을 전달하는 *GetOptionalParam* 메소드를 사용합니다.

## 다른 데이터셋에서 데이터 복사

디자인 타임에 다른 데이터셋으로부터 데이터를 복사하려면 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Assign Local Data를 선택합니다. 프로젝트에서 사용 가능한 모든 데이터셋을 나열하는 대화 상자가 나타납니다. 대화 상자에서 복사할 데이터셋을 선택한 다음 OK를 선택합니다. 소스 데이터셋을 복사하면 클라이언트 데이터셋은 자동으로 활성화됩니다.

런타임에 다른 데이터셋에서 복사하려면 데이터를 직접 할당하거나 소스가 다른 클라이언트 데이터셋에 있는 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 올 수 있습니다. 일단 데이터 패킷을 *Data*에 할당하면 데이터 소스 컴포넌트에 의해 클라이언트 데이터셋에 연결된 data-aware 컨트롤에 내용이 자동으로 표시됩니다.

*TSQLClientDataSet* 객체나 프로바이더에 연결된 *TClientDataSet* 객체를 여는 경우 데이터 패킷은 자동으로 *Data*에 할당됩니다. *TSQLClientDataSet* 사용에 대한 자세한 내용은 20-35 페이지의 "SQL 클라이언트 데이터셋 사용"을 참조하십시오. 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용하는 것에 대한 자세한 내용은 20-25 페이지의 "프로바이더와 함께 클라이언트 데이터셋 사용"을 참조하십시오.

데이터베이스나 프로바이더로부터 데이터를 사용하지 않는 경우, 다음과 같이 다른 클라이언트 데이터셋에서 데이터를 복사할 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.

데이터셋 프로바이더 컴포넌트를 만들고 소스 데이터셋에 연결한 다음 데이터를 복사하여 데이터를 클라이언트 데이터셋이 아닌 데이터셋에서 *TClientDataSet*으로 복사할 수 있습니다.

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

**참고** 사용자가 직접 *Data* 속성에 할당하면 새 데이터 패킷은 기존의 데이터에 합병되지 않습니다. 그 대신, 모든 이전 데이터가 교체됩니다.

임의의 클라이언트 데이터셋으로 프로바이더 컴포넌트를 사용하여 다른 클라이언트 데이터셋의 변경 사항을 합병할 수 있습니다. 앞의 예제에서처럼 데이터셋 프로바이더를 만듭니다. 그러나, 대상 데이터셋에 연결한 다음 데이터 속성을 복사하는 대신에 *ApplyUpdates* 메소드를 사용합니다.

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SQLClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

**참고** 데이터를 이 방법으로 합병하면 대상 클라이언트 데이터셋의 인메모리 캐시에 합병됩니다. 합병한 다음 변경 사항을 데이터베이스 서버에 다시 적용하려면, 해당 클라이언트 데이터셋의 *ApplyUpdates* 메소드를 호출해야 합니다.

## 클라이언트 데이터셋 커서 복제

클라이언트 데이터셋은 사용자가 런타임에 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 모든 원래의 데이터를 복사하는 것보다 비용이 적게 들지만 데이터를 공유하기 때문에 두 번째 클라이언트 데이터셋의 데이터를 수정하면 원래 클라이언트 데이터셋이 영향을 받습니다.

*CloneCursor*는 세 개의 매개변수를 사용하는데, 이 중 *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 정보를 포함할 때), *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 (아무런 인덱스도 없고, 아무런 필터도 없으며 *ReadOnly*는 *False* 그리고 연결 컴포넌트 또는 프로바이더가 지정되지 않음)이 설정됩니다. *KeepSettings*가 *True*이면 대상 데이터셋의 속성은 변경되지 않습니다.

## 프로바이더와 함께 클라이언트 데이터셋 사용

클라이언트 데이터셋을 사용하여 데이터베이스 서버나 다른 데이터셋으로부터 데이터를 나타내는 경우 클라이언트 데이터셋은 프로바이더 컴포넌트를 사용합니다. 프로바이더 컴포넌트는 데이터를 소스 데이터셋에서 클라이언트 데이터셋으로 전달합니다. *TSQLClientDataSet*을 사용하는 경우, 이 소스 데이터셋은 폼이나 데이터 모듈에 추가하는 데이터셋 컴포넌트인 *TClientDataSet*과 함께 데이터에 액세스하는 내부 쿼리 컴포넌트입니다. 메모리에서 데이터를 편집한 후 클라이언트 데이터셋은 프로바이더를 통해 업데이트를 다시 소스 데이터셋에 또는 원격 데이터베이스 서버에 직접 적용합니다.

*TSQLClientDataSet* 컴포넌트와 함께 프로바이더는 내부용이기 때문에 직접 액세스할 수는 없습니다. 그러나 클라이언트 데이터셋과 내부 프로바이더 사이의 통신에 영향을 주고 그 결과 클라이언트 데이터셋과 데이터베이스 서버의 통신에 영향을 주는 메소드와 클라이언트 데이터셋의 속성은 계속 사용할 수 있습니다.

*TClientDataSet*과 함께 프로바이더는 클라이언트 데이터셋과 동일한 애플리케이션에 있거나 다른 시스템에서 실행 중인 별도 애플리케이션의 일부일 수 있습니다.

프로바이더 컴포넌트에 대한 자세한 내용은 21 장 "프로바이더 컴포넌트 사용"을 참조하십시오.

다음 단계들은 프로바이더와 함께 클라이언트 데이터셋을 사용하는 방법을 설명합니다.

- 1 *TClientDataSet*을 사용하는 경우 데이터 프로바이더를 지정합니다.
- 2 옵션으로 소스 데이터셋에서 매개변수를 얻거나 소스 데이터셋으로 매개변수를 전달합니다.
- 3 클라이언트 데이터셋과 프로바이더에 따라 서버에서 실행할 명령을 지정할 수 있습니다.
- 4 소스 데이터셋에서 데이터를 요청합니다.
- 5 레코드를 업데이트합니다.
- 6 레코드를 새로 고칩니다.

*TClientDataSet*을 통해 사용자 지정 이벤트를 사용하여 프로바이더와 통신할 수도 있습니다.

### 데이터 프로바이더 지정

*TClientDataSet*은 다른 데이터셋에서 데이터를 받고 해당 데이터셋이나 데이터베이스 서버에 업데이트를 적용하기 전에 데이터셋 프로바이더에 연결되어야 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 있는 경우, Object Inspector의 *ProviderName* 속성의 드롭다운 목록에서 프로바이더를 선택하여 *TClientDataSet*을 프로바이더에 연결할 수 있습니다. 이는 프로바이더가 클라이언트 데이터셋과 동일한 소유자(*Owner*)를 가지는 동안 작동합니다. 클라이언트 데이터셋과 프로바이더는 동일한 폼이나 데이터 모듈에 있는 경우 동일한 소유자를 가집니다. 다른 소유자를 가지는 지역 프로바이더를 사용하려면 클라이언트 데이터셋의 *SetProvider* 메소드를 사용하여 런타임에 연결을 만들어야 합니다.
- 프로바이더가 원격 애플리케이션 서버에 있으면 *RemoteServer*와 *ProviderName* 속성을 모두 사용해야 합니다. *RemoteServer*는 프로바이더 목록을 얻기 위한 연결 컴포넌트의 이름을 지정합니다. 연결 컴포넌트는 클라이언트 데이터셋과 동일한 데이터 모듈에 있습니다. 연결 컴포넌트는 애플리케이션 서버에 연결을 만들고 유지 관리하므로 "데이터 브로커"라고 부르기도 합니다.

디자인 타임에 *RemoteServer*를 지정한 다음 Object Inspector의 *ProviderName* 속성의 드롭다운 목록에서 프로바이더를 선택할 수 있습니다. 이 목록에는 동일한 폼 또는 데이터 모듈의 지역 프로바이더와 연결 컴포넌트를 통해 액세스할 수 있는 원격 프로바이더가 있습니다.

**참고** *TClientDataSet*을 원격 애플리케이션 서버의 프로바이더에 연결하는 연결 컴포넌트는 별도로 구입해야 합니다.

런타임 시 코드에 *ProviderName*을 설정하여 사용 가능한 지역 프로바이더와 원격 프로바이더 사이를 전환할 수 있습니다.

## 소스 데이터셋에서 매개변수 얻기

다음과 같은 두 상황에서, 클라이언트 데이터셋은 소스 데이터셋에서 매개변수 값을 얻어야 합니다.

- 클라이언트에 내장 프로시저의 출력 매개변수 값이 필요합니다.
- 클라이언트가 쿼리나 내장 프로시저의 입력 매개변수를 소스 데이터셋의 현재 값으로 초기화하려고 합니다.

클라이언트 데이터셋은 매개변수 값을 *Params* 속성에 저장합니다. 클라이언트 데이터셋이 소스 데이터셋에서 데이터를 페치(fetch)할 때마다 이 값은 출력 매개변수로 새로 고쳐집니다. 그러나, 클라이언트 애플리케이션의 *TClientDataSet* 컴포넌트가 데이터를 페치하지 않는 경우에도 출력 매개변수가 필요할 때가 있습니다.

레코드를 페치하지 않는 경우에도 출력 매개변수를 페치하거나 입력 매개변수를 초기화하기 위해서 클라이언트 데이터셋이 *FetchParams* 메소드를 호출하여 소스 데이터셋에서 매개변수 값을 요청할 수 있습니다. 매개변수는 프로바이더로부터 데이터 패킷에 반환되고 클라이언트 데이터셋의 *Params* 속성에 할당됩니다.

디자인 타임에 *Params* 속성은 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Fetch Params를 선택하여 초기화할 수 있습니다.

**참고** *Params* 속성이 내부 소스 데이터셋에 매개변수를 항상 반영하기 때문에 *TSQL Client DataSet*으로 작업할 때 *FetchParams*를 호출할 필요가 없습니다. *TClient DataSet*과 함께 *FetchParams* 메소드나 *Fetch Params* 명령은 연결된 데이터셋이 매개변수를 제공할 수 있는 프로바이더에 클라이언트 데이터셋이 연결된 경우에만 작동합니다. 예를 들어, 소스 데이터셋이 *ctTable*의 *CommandType*을 가진 *TSQLDataSet* 객체인 경우 페치할 매개변수가 없습니다.

프로바이더가 스테이트리스 (stateless) 애플리케이션 서버의 일부로서 별도의 시스템에 있는 경우 *FetchParams*를 사용하여 출력 매개변수를 검색할 수 없습니다. 스테이트리스 애플리케이션 서버에서 다른 클라이언트는 *FetchParams*로의 호출 전에 출력 매개변수를 변경하여 쿼리나 내장 프로시저를 변경하고 재실행할 수 있습니다. 스테이트리스 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 다음을 지정할 수 있습니다.

- 애플리케이션 서버에서 실행되는 내장 프로시저 또는 쿼리의 매개변수 값
- 데이터 패킷으로 보내는 레코드를 제한하는 필드 값

디자인 타임이나 런타임에 클라이언트 데이터셋이 프로바이더에 보내는 매개변수 값을 지정할 수 있습니다. 디자인 타임에 클라이언트 데이터셋을 선택한 다음 Object Inspector의 *Params* 속성을 더블 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.

런타임에 *Params* 속성의 *CreateParam* 메소드를 사용하여 클라이언트 데이터셋에 매개변수를 추가합니다. *CreateParam*는 매개변수 객체, 지정된 이름, 매개변수 타입 및 데이터 타입을 반환합니다. 그런 다음 해당 매개변수 객체의 속성을 사용하여 매개변수에 값을 할당합니다.

예를 들면, 다음 코드는 CustNo 매개변수의 값을 605로 설정합니다.

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
    AsInteger := 605;
```

클라이언트 데이터셋이 활성화되지 않은 경우, 매개변수를 프로바이더에 보낼 수 있고 단순히 *Active* 속성을 *True*로 설정하여 매개변수 값을 반영하는 데이터 패킷을 검색할 수 있습니다.

## 쿼리 또는 내장 프로시저 매개변수 보내기

프로바이더가 쿼리나 내장 프로시저의 결과를 표시하는 경우 *Params* 속성을 사용하여 매개변수 값을 지정할 수 있습니다. 클라이언트 데이터셋이 소스 데이터셋으로부터 데이터를 요청하거나 *Execute* 메소드를 사용하여 데이터셋을 반환하지 않는 쿼리나 내장 프로시저를 실행하는 경우 이 매개변수 값을 데이터에 대한 요청이나 실행 명령과 함께 전달합니다. 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

프로바이더가 *TSQLTable* 컴포넌트를 나타내는 경우 *Params* 속성을 사용하여 *Data* 속성에 제공되는 레코드를 제한할 수 있습니다.

각 매개변수 이름은 프로바이더에 연결된 *TSQLTable* 컴포넌트의 필드 이름과 일치해야 합니다. 그러면 프로바이더는 해당 필드의 값이 매개변수에 할당된 값과 일치하는 레코드만 보냅니다.

예를 들어, 고객 한 명의 주문 내용을 보여주는 애플리케이션을 생각해 봅시다. 사용자가 고객 한 명을 식별할 때 클라이언트 데이터셋은 그 고객의 주문 정보들만 보이도록 식별하는 값인 *CustID*라는 단일 매개변수를 포함하도록 *Params* 속성을 설정합니다. 클라이언트 데이터셋이 소스 데이터셋에서 데이터를 요청하면 이 매개변수 값을 전달합니다. 그러면, 프로바이더는 이 고객에 대한 레코드만 보냅니다. 이것은 프로바이더가 모든 주문 레코드를 클라이언트 애플리케이션에 보내고 클라이언트 데이터셋을 사용하여 레코드를 필터링하는 것보다 훨씬 효율적입니다.

## 서버에서 실행할 명령 지정

*TSQLClientDataSet*을 사용하는 경우 애플리케이션은 소스 데이터셋에 직접 액세스할 수 없습니다. 그 대신, 클라이언트 데이터셋은 데이터 패킷을 만드는 SQL 문을 지정해야 합니다. *CommandText* 속성을 사용합니다. *CommandType* 속성은 *CommandText*가 서버가 실행할 SQL 문, 테이블의 이름 또는 내장 프로시저의 이름을 나타내는지 여부를 나타냅니다.

*TClientDataSet*을 사용하는 경우 일반적으로 소스 데이터셋은 클라이언트에 제공되는 데이터를 결정합니다. 소스 데이터셋은 데이터를 생성하는 SQL 문을 지정하는 속성을 가지거나 특정한 데이터베이스 테이블 또는 내장 프로시저를 나타낼 수 있습니다. 예를 들어, *TSQLDataSet*의 *CommandText* 속성은 SQL 문, 테이블 이름 또는 내장 프로시저 이름을 *CommandType* 속성의 값에 따라 지정합니다. 반면 *TSQLStoredProc*의 *ProcedureName* 속성은 실행하는 내장 프로시저를 지정합니다.



프로바이더가 허용하는 경우 *TClientDataSet*은 데이터셋이 어떤 데이터를 나타내는지를 표시하는 속성을 오버라이드할 수 있습니다. 이를 이용하면 클라이언트 데이터셋이 보려는 데이터를 동적으로 지정할 수 있습니다. *TSQLClientDataSet*을 가지고 *CommandText*를 프로바이더 데이터셋의 SQL 문, 테이블 이름 또는 내장 프로시저의 이름을 대체하는 SQL 문으로 설정할 수 있습니다. 사용할 값의 타입은 프로바이더에 연결된 데이터셋의 타입에 의해 결정됩니다.

기본적으로 *TSQLClientDataSet*에서 사용하는 내부 프로바이더를 사용하면 페치할 데이터를 지정하는 다른 방법이 없기 때문에 항상 클라이언트 데이터셋이 *CommandText* 값을 지정할 수 있습니다. 그러나 외부 프로바이더 컴포넌트의 경우 기본적으로 클라이언트 데이터셋이 이러한 방법으로 *CommandText*를 지정하는 것을 허용하지 않습니다. *TClientDataSet*이 *CommandText* 속성을 사용하도록 하려면 *poAllowCommandText*를 프로바이더의 *Options* 속성에 추가해야 합니다. 그렇지 않으면 *CommandText*의 값이 무시됩니다.

**경고** 데이터셋을 열기 전에 *TSQLClientDataSet*의 *Options* 속성에서 *poAllowCommandText*를 제거하지 마십시오.

다음의 두 경우에 클라이언트 데이터셋은 *CommandText* 문자열을 프로바이더로 보냅니다.

- 클라이언트 데이터셋을 처음 열 때, 프로바이더에서 첫 번째 데이터 패킷을 검색한 후 다음의 연속적인 데이터 패킷을 페치할 때 클라이언트 데이터셋은 *CommandText*를 보내지 않습니다.
- 클라이언트 데이터셋이 *Execute* 명령을 프로바이더로 보낼 때.

나중에 언제든지 SQL 명령을 보내거나 테이블 또는 내장 프로시저 이름을 변경하려면 *AppServer* 속성으로 사용 가능한 *IAppServer* 인터페이스를 명시적으로 사용해야 합니다.

## 소스 데이터셋에 데이터 요청

다음 표는 데이터를 프로바이더에서 페치하는 방법을 결정하는 클라이언트 데이터셋의 속성 및 메소드를 나열한 것입니다.

**표 20.3** 데이터 요청 처리를 위한 클라이언트 데이터셋 속성 및 메소드

| 속성 또는 메소드         | 용도                                                                                                                                            |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| FetchOnDemand 속성  | 필요할 때 클라이언트 데이터셋이 자동으로 데이터를 페치할지 또는 애플리케이션에 의존하여 추가 데이터를 검색하는 <i>GetNextPacket</i> , <i>FetchBlobs</i> 및 <i>FetchDetails</i> 함수를 호출할지를 결정합니다. |
| PacketRecords 속성  | 각 데이터 패킷에 포함하는 레코드의 타입이나 수를 지정합니다.                                                                                                            |
| GetNextPacket 메소드 | 프로바이더에서 다음 데이터 패킷을 페치합니다.                                                                                                                     |
| FetchBlobs 메소드    | 프로바이더가 BLOB 데이터를 자동적으로 포함하지 않는 경우 현재 레코드에 대한 BLOB 필드를 페치합니다.                                                                                  |
| FetchDetails 메소드  | 프로바이더가 데이터 패킷에 디테일 정보를 자동적으로 포함하지 않는 경우 현재 레코드의 중첩 디테일 데이터셋을 페치합니다.                                                                           |

기본적으로 클라이언트 데이터셋은 소스 데이터셋으로부터 모든 레코드를 가져 옵니다. *PacketRecords*와 *FetchOnDemand*를 사용하여 이를 변경할 수 있습니다.

## 점진적 페치

*PacketRecords*는 한 번에 페치하는 레코드 수 또는 반환하는 레코드의 타입을 지정합니다. 기본적으로 *PacketRecords*는  $-1$ 로 설정되는데 이는 클라이언트 데이터셋이 처음 열린 때나 애플리케이션이 명시적으로 *GetNextPacket*을 호출할 때 사용 가능한 모든 레코드를 한꺼번에 페치함을 의미합니다. *PacketRecords*가  $-1$ 인 경우 클라이언트 데이터셋이 데이터를 처음 페치하고 나면 이미 사용 가능한 모든 레코드가 있기 때문에 더 이상 데이터를 페치할 필요가 없습니다.

작은 배치(batch) 단위로 레코드를 페치하려면 *PacketRecords*를 페치할 레코드의 수로 설정합니다. 예를 들어, 다음 문장은 각 데이터 패킷의 크기를 10개 레코드로 설정합니다.

```
ClientDataSet1.PacketRecords := 10;
```

배치로 레코드를 페치하는 이러한 프로세스를 "점진적 페치(incremental fetch)"라고 합니다. 클라이언트 데이터셋은 *PacketRecords*가 0보다 크면 점진적 페치를 사용합니다. 기본적으로 클라이언트 데이터셋은 필요한 경우 데이터를 페치하도록 *GetNextPacket*을 자동으로 호출합니다. 새로 페치된 패킷은 클라이언트 데이터셋에 이미 있는 데이터의 끝에 추가됩니다.

*GetNextPacket*은 페치한 레코드 수를 반환합니다. 반환 값이 *PacketRecords*와 동일한 경우 사용 가능한 레코드의 끝에 도달하지 않습니다. 반환 값이 0보다 크고 *PacketRecords*보다 작은 경우 마지막 레코드는 페치 작업 중에 도달됩니다. *GetNextPacket*이 0을 반환하면 더 이상 페치할 레코드가 없습니다.

**경고** 점진적 페치는 스테이트리스(stateless) 애플리케이션 서버의 원격 프로바이더로부터 데이터를 페치하는 경우 작동하지 않습니다.

또한 *PacketRecords*를 사용하여 소스 데이터셋에 대한 메타데이터 정보를 페치할 수 있습니다. 메타데이터 정보를 검색하려면 *PacketRecords*를 0으로 설정합니다.

## 요구 즉시 페치

레코드의 자동 페치는 *FetchOnDemand* 속성에서 제어합니다. *FetchOnDemand*가 기본값 *True*인 경우 자동 페치가 활성화됩니다. 필요한 경우 클라이언트 데이터셋이 자동으로 레코드를 페치하지 못하게 하려면 *FetchOnDemand*를 *False*로 설정합니다. *FetchOnDemand*가 *False*인 경우 애플리케이션은 레코드를 페치하기 위해서 *GetNextPacket*을 명시적으로 호출해야 합니다.

예를 들면, 대량의 읽기 전용 데이터셋을 나타내야 하는 애플리케이션은 *FetchOnDemand*를 해제하여 클라이언트 데이터셋이 메모리를 초과하는 데이터를 로드하지 않게 할 수 있습니다. 페치 사이에서 클라이언트 데이터셋은 *EmptyDataSet* 메소드를 사용하여 캐시를 해제합니다. 그러나 이 방법은 클라이언트가 서버에 업데이트를 포스트해야 하는 경우에는 잘 작동하지 않습니다.

*TSQLClientDataSet*을 사용하는 경우 *Options* 속성은 데이터 패킷의 레코드가 BLOB 데이터 및 중첩 디테일 데이터셋을 포함하는지 여부를 제어합니다. *TClientDataSet*을 사용하는 경우 외부 프로바이더는 데이터 패킷의 정보 포함 여부를 제어합니다. 데이터 패킷이 이 정보를 포함하지 않으면 *FetchOnDemand* 속성은 클라이언트 데이터셋이 필요에 따라 BLOB 데이터와 디테일 데이터셋을 자동으로 페치하게 합니다. *FetchOnDemand*가 *False*이고 BLOB 데이터와 디테일 데이터셋이 데이터 패킷에 포함되지 않은 경우 이 정보를 검색하려면 *FetchBlobs* 또는 *FetchDetails* 메소드를 명시적으로 호출해야 합니다.

## 레코드 업데이트

클라이언트 데이터셋은 데이터의 로컬 복사본을 사용합니다. 사용자는 클라이언트 애플리케이션의 data-aware 컨트롤에서 이 복사본을 보고 편집합니다. 사용자 변경 내용은 클라이언트 데이터셋에 의해 내부적으로 유지 관리되는 변경 로그에 임시로 저장됩니다. 변경 로그의 내용은 *Delta* 속성에 데이터 패킷으로 저장됩니다. *Delta*의 변경 내용을 영구적으로 만들려면 데이터베이스에 적용해야 합니다.

클라이언트는 다음과 같은 단계를 따라 서버에 업데이트를 적용합니다.

- 1 애플리케이션은 클라이언트 데이터셋 객체의 *ApplyUpdates* 메소드를 호출합니다. 이 메소드는 클라이언트 데이터셋의 *Delta* 속성 내용을 프로바이더에 전달합니다. *Delta*는 클라이언트 데이터셋의 업데이트, 삽입 및 삭제된 레코드를 포함하는 데이터 패킷입니다.
- 2 프로바이더는 해결하지 못하는 모든 문제 레코드를 캐시하며 데이터베이스 또는 소스 데이터셋에 업데이트를 적용합니다. 프로바이더가 업데이트를 적용하는 방법에 대한 자세한 내용은 21-7 페이지의 "클라이언트 업데이트 요청에 대한 응답"을 참조하십시오.
- 3 프로바이더는 해결하지 못한 모든 레코드를 *Result* 데이터 패킷의 클라이언트 데이터셋에 반환합니다. *Result* 데이터 패킷에는 업데이트되지 않은 모든 레코드가 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 복사본의 변경 내용은 클라이언트 애플리케이션이 데이터셋의 *ApplyUpdates* 메소드를 호출해야만 데이터베이스 서버에 보내집니다. *ApplyUpdates*는 변경 로그를 사용하여 프로바이더에 *Delta*라는 데이터 패킷으로 변경 내용을 보냅니다.

*ApplyUpdates*는 *MaxErrors*라는 매개변수를 하나 사용하는데 이는 업데이트 프로세스를 중단할 때까지 허용할 수 있는 최대 오류 수를 나타냅니다. *MaxErrors*가 0이면 업데이트 오류가 발생하는 즉시 전체 업데이트 프로세스가 종료됩니다. 데이터베이스에 대해 아무런 변경이 없으면 클라이언트 데이터셋의 변경 로그는 바뀌지 않은 상태로 남습니다. *MaxErrors*가 -1인 경우에는 수에 상관없이 오류가 허용되며 변경 로그는 성공적으로 적용되지 않은 모든 레코드를 포함한 채로 끝납니다. *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 기록될 수 없는 레코드의 수를 나타냅니다.

클라이언트 데이터셋의 *ApplyUpdates* 메소드는 다음을 수행합니다.

- 1 프로바이더의 *ApplyUpdates* 메소드를 간접적으로 호출합니다. 프로바이더의 *ApplyUpdates* 메소드는 데이터베이스나 소스 데이터셋에 업데이트를 기록하고 발생한 오류를 수정하려고 합니다. 오류 조건으로 인해 적용될 수 없는 레코드는 클라이언트 데이터셋으로 다시 보내집니다.
- 2 그런 다음 클라이언트 데이터셋의 *ApplyUpdates* 메소드는 *Reconcile* 메소드를 호출하여 이런 문제 레코드를 해결하려 합니다. *Reconcile*은 *OnReconcileError* 이벤트 핸들러를 호출하는 오류 처리 루틴입니다. 오류를 수정하려면 *OnReconcileError* 이벤트 핸들러를 직접 코딩해야 합니다. *OnReconcileError* 사용에 대한 자세한 내용은 20-32 페이지의 "업데이트 오류 해결"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.

**팁** 프로바이더가 스테이트리스(stateless) 애플리케이션 서버에 있는 경우 업데이트를 적용하기 전후에 영구적인 상태 정보에 대해 프로바이더와 통신하기를 원할 수도 있습니다. 클라이언트 데이터셋은 업데이트를 보내기 전에 *BeforeApplyUpdates* 이벤트를 받는데, 이를 사용하여 영구적인 상태 정보를 서버에 보낼 수 있습니다. 업데이트를 적용한 후 해결 프로세스 전에 클라이언트 데이터셋은 애플리케이션 서버가 반환하는 영구적인 상태 정보에 응답할 수 있는 *AfterApplyUpdates* 이벤트를 받습니다.

## 업데이트 오류 해결

프로바이더는 오류 레코드와 오류 정보를 결과 데이터 패킷으로 클라이언트 데이터셋에 반환합니다. 프로바이더가 0보다 큰 오류 수를 반환한다면 결과 데이터 패킷의 각 레코드에 대해 클라이언트 데이터셋의 *OnReconcileError* 이벤트가 발생합니다.

사용자는 프로바이더가 반환한 레코드를 버리는 경우에도 *OnReconcileError* 이벤트 핸들러를 코딩해야 합니다. *OnReconcileError* 이벤트 핸들러에는 다음과 같은 4개의 매개변수가 전달됩니다.

- *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 업데이트 문제의 원인을 알아 내기 위해 현재 레코드에 있는 필드의 *CurValue*, *OldValue* 및 *NewValue* 속성 사용을 원할 수도 있습니다. 그러나, *OnReconcileError* 이벤트 핸들러의 현재 레코드를 변경하는 클라이언트 데이터셋 메소드를 호출하면 안됩니다.
- *E*: 발생한 문제를 나타내는 *EReconcileError* 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* (기본 레코드를 삭제할 때 발생하는 문제) 중의 하나입니다.
- *Action*: *OnReconcileError* 핸들러를 종료할 때 수행할 작업을 표시할 수 있는 **var** 매개변수. 핸들러의 엔트리에서 *Action*은 프로바이더의 해결 프로세스가 취하는 작업에 설정됩니다. 이벤트 핸들러에서 이 매개변수를 설정하여 다음을 수행합니다.
  - 변경 로그에 남겨둔 채로 이 레코드를 건너뛵니다. (raSkip)
  - 모든 해결 작업을 중지합니다. (raAbort)
  - 서버에서 해당 레코드에 실패한 수정을 합병합니다. (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 교체합니다. (raCorrect)
  - 클라이언트 데이터셋에서 이 레코드의 변경 내용을 원래 제공된 값으로 복귀하여 되돌립니다. (raCancel)
  - 현재 레코드 값을 업데이트하여 서버의 레코드와 일치하도록 합니다. (raRefresh)

다음 코드는 객체 레포지토리 디렉토리에 있는 RecError 유닛에서 오류 해결 대화 상자를 사용하는 *OnReconcileError* 이벤트 핸들러를 보여 줍니다. 이 대화 상자를 사용하기 위해 uses 절에 RecError를 추가합니다.

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
    Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

## 레코드 새로 고침

클라이언트 데이터셋은 데이터의 인메모리 (in-memory) 스냅샷으로 작업합니다. 해당 데이터가 데이터베이스 서버에서 오는 경우 시간이 경과하면 다른 사용자가 데이터를 수정할 수 있습니다. 그러면 클라이언트 데이터셋의 데이터는 원본 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 애플리케이션은 변경 로그는 그대로 남겨 둔 채로 데이터를 업데이트할 수도 있습니다. 이렇게 하려면 클라이언트 데이터셋의 *RefreshRecord* 메소드를 호출합니다. *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* 호출이 항상 적절한 것은 아닙니다. 사용자의 편집이 다른 사용자가 만든 원본으로 사용하는 데이터셋에 대한 변경 내용과 충돌을 일으키는 경우 *RefreshRecord* 호출은 이 충돌을 마스크합니다. 클라이언트 애플리케이션이 업데이트를 적용할 때, 오류 해결이 발생하지 않으며 애플리케이션은 충돌을 해결할 수 없습니다.

업데이트 오류 마스킹을 막으려면 *RefreshRecord*를 호출하기 전에 현재 레코드에 대해 보류 중인 업데이트가 없는지 확인해야 합니다. 예를 들어, 다음 코드는 수정된 레코드를 새로 고치려는 경우 예외를 발생시킵니다.

```
if ClientDataSet1.UpdateStatus <> usUnModified then
    raise Exception.Create('You must apply updates before refreshing the current
record.');
```

ClientDataSet1.RefreshRecord;

## 사용자 지정 이벤트를 사용하여 프로바이더와 통신

클라이언트 데이터셋은 *IAppServer*라는 특수한 인터페이스를 통해 프로바이더 컴포넌트와 통신합니다. 지역 프로바이더의 경우, *IAppServer*는 클라이언트 데이터셋과 프로바이더 사이의 모든 통신을 처리하는 자동으로 생성된 객체에 대한 인터페이스입니다. 원격 프로바이더의 경우, *IAppServer*는 애플리케이션 서버의 원격 데이터 모듈에 대한 인터페이스입니다.

*TClientDataSet*은 *IAppServer* 인터페이스를 사용하는 통신을 사용자 지정할 수 있는 기회를 많이 제공합니다. 클라이언트 데이터셋의 프로바이더에서 다이렉트되는 모든 *IAppServer* 메소드 호출 후 *TClientDataSet*은 프로바이더와 정보를 교환할 수 있게 하는 특별한 이벤트를 받습니다. 이러한 이벤트는 프로바이더의 비슷한 이벤트들과 일치합니다. 예를 들어, 클라이언트 데이터셋이 *ApplyUpdates* 메소드를 호출하면 다음 이벤트가 발생합니다.

- 1 클라이언트 데이터셋은 OleVariant 타입의 *OwnerData*라는 속성에 임의의 사용자 지정 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 조합하기 위한 일반적인 프로세스를 수행합니다(여기에 따른 이벤트들이 발생합니다).
- 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.

다른 모든 *IAppServer* 메소드 호출도 *TClientDataSet*과 프로바이더 사이의 통신을 사용자 지정할 수 있게 하는 이와 비슷한 일련의 *BeforeXXX* 및 *AfterXXX* 이벤트들이 발생합니다.

또한 *TClientDataSet*은 *DataRequest*라는 특별한 메소드를 가집니다. 이 메소드의 유일한 용도는 프로바이더와 애플리케이션 특정 통신을 할 수 있게 하는 것입니다. 클라이언트 데이터셋이 *DataRequest*를 호출하는 경우 원하는 모든 정보를 포함할 수 있는 *OleVariant*를 매개변수로 전달합니다. 그런 다음, 사용자가 임의의 애플리케이션 정의 방식으로 응답할 수 있고 클라이언트 데이터셋에 값을 반환할 수 있는 *OnDataRequest* 이벤트를 프로바이더에 생성합니다.

## SQL 클라이언트 데이터셋 사용

*TSQClientDataSet*은 2계층 애플리케이션을 위해 고안된 특별한 타입의 클라이언트 데이터셋입니다. 단방향 데이터셋처럼 SQL 연결 컴포넌트를 사용하여 데이터베이스 서버에 연결하고 서버에서 실행할 SQL 문을 지정할 수 있습니다. 다른 클라이언트 데이터셋처럼 데이터를 메모리에 버퍼하여 전체 탐색과 편집 지원을 가능하게 합니다.

*TSQClientDataSet*은 데이터셋 프로바이더에 의해 단방향 데이터셋에 연결되는 일반적인 클라이언트 데이터셋 (*TClientDataSet*) 과 동일한 방법으로 작동합니다. 사실 *TSQClientDataSet*에는 내부적으로 생성된 단방향 데이터셋과의 통신에 사용하는 자체 내부 프로바이더가 있습니다.

SQL 클라이언트 데이터셋을 사용하면 많은 컴포넌트를 가지고 작업할 필요가 없으므로 2계층 애플리케이션 개발 프로세스를 단순화할 수 있습니다. 사용자는 내부 프로바이더에 직접 액세스할 수 없지만 SQL 클라이언트 데이터셋은 업데이트를 적용하는 방법을 구성할 수 있게 하는 속성을 게시합니다. 그러나 대개 *TSQClientDataSet*이 제공하는 기본값이면 충분합니다.

## TSQClientDataSet 사용

*TSQClientDataSet*은 2계층 데이터베이스 애플리케이션과 브리프케이스 (briefcase) 모델 애플리케이션에서의 사용을 위해 고안되었습니다. 이는 데이터베이스 서버로의 연결, 데이터 페치, 업데이트 캐싱 및 서버에 다시 적용을 위한 컴포넌트의 손쉬운 설정을 제공합니다. 대부분의 2계층 애플리케이션에서 사용할 수 있습니다.

다음과 같은 경우에서는, *TClientDataSet*을 사용하는 것이 더 적절합니다.

- 데이터베이스 서버로부터의 데이터를 사용하지 않는 경우 (예를 들어, 디스크 상의 전용 파일을 사용하는 경우), *TClientDataSet*은 오버헤드가 작다는 장점이 있습니다.
- *TClientDataSet*만 다계층 데이터베이스 애플리케이션에 사용할 수 있습니다. 그러므로 다계층 애플리케이션을 작성하고 있거나 언젠가는 다계층 애플리케이션으로 확장하려는 경우 외부 프로바이더 및 소스 데이터셋과 함께 *TClientDataSet*을 사용해야 합니다.

- SQL 클라이언트 데이터셋 컴포넌트에 대해 소스 데이터셋은 내부용이기 때문에 마스터/디테일 관계에서 중첩 디테일 셋을 얻기 위해 두 소스 데이터셋을 연결할 수 없습니다. 그러나 마스터/디테일 관계에서 두 개의 SQL 클라이언트 데이터셋은 연결할 수 있습니다.
- SQL 클라이언트 데이터셋은 내부 데이터셋 프로바이더에 발생한 많은 이벤트를 표면화하지 않습니다. 그러나 대부분의 경우에 이러한 이벤트는 다계층 애플리케이션에서 사용되고, 2계층 애플리케이션에서는 필요하지 않습니다.

## SQL 클라이언트 데이터셋 설정

*TSQLClientDataSet*을 사용하려면 다음과 같이 합니다.

- 1 데이터 모듈이나 폼에 *TSQLClientDataSet* 컴포넌트를 두고 *Name* 속성을 애플리케이션에 적절한 고유한 값으로 설정합니다.
- 2 데이터를 포함하는 데이터베이스 서버를 식별합니다. 이를 수행하는 방법은 다음 두 가지가 있습니다.
  - 연결 파일에 명명된 연결을 정의한 경우 해당 연결 이름을 *ConnectionName* 속성의 값으로 간단하게 지정할 수 있습니다. 명명된 연결에 대한 자세한 내용은 19-3 페이지의 "연결에 대한 설명 이름 지정"을 참조하십시오.
  - 연결 속성, 트랜잭션 지원, 로그인 지원 및 둘 이상의 데이터셋에 대한 단일 연결을 사용하는 기능에 대해 더 많이 제어하려면 별도의 *TSQLConnection* 컴포넌트를 사용해야 합니다. *TSQLConnection* 컴포넌트를 *DBConnection* 속성 값으로 지정합니다. *TSQLConnection*에 대한 자세한 내용은 19장 "데이터베이스에 연결"을 참조하십시오.
- 3 서버에서 페치하려는 데이터를 지시합니다. 이를 수행하는 방법은 다음 세 가지가 있습니다.
  - *CommandType*을 *ctQuery*로 설정하고 *CommandText*을 서버에서 실행하려는 SQL 문으로 설정합니다. 이 문장은 전형적인 SELECT 문입니다. *Params* 속성을 사용하여 임의의 매개변수 값을 제공합니다.
  - *CommandType*을 *ctStoredProc*으로 설정하고 *CommandText*를 실행하려는 내장 프로시저 이름으로 설정합니다. *Params* 속성을 사용하여 임의의 입력 매개변수 값을 제공합니다.
  - *CommandType*을 *ctTable*로 설정하고 *CommandText*를 레코드를 사용하려는 데이터베이스 테이블의 이름으로 설정합니다.
- 4 데이터를 비주얼 데이터 컨트롤과 함께 사용하려면 데이터 소스 컴포넌트를 폼 또는 데이터 모듈에 추가하고 *DataSet* 속성을 *TSQLClientDataSet* 객체로 설정합니다. 데이터 소스 컴포넌트는 데이터 표시를 위해 data-aware 컴포넌트로 클라이언트 데이터셋의 인메모리 캐시에 데이터를 전달(forward) 합니다. *DataSource* 및 *DataField* 속성을 사용하여 data-aware 컴포넌트를 데이터 소스에 연결합니다.
- 5 원한다면 내부 프로바이더가 업데이트를 적용하는 방법을 구성합니다. 이를 가능하게 하는 속성과 이벤트는 아래의 "내부 프로바이더 구성"에서 설명합니다.



- 6 *Active* 속성을 *True*로 설정(또는 런타임에 *Open* 메소드 호출)하여 데이터셋을 활성화합니다.
- 7 내장 프로시저를 실행하는 경우, *Params* 속성을 사용하여 출력 매개변수를 검색합니다.
- 8 사용자가 SQL 클라이언트 데이터셋에서 데이터를 편집한 경우 *ApplyUpdates* 메소드를 호출하여 편집 내용을 다시 데이터베이스 서버에 적용할 수 있습니다. *OnUpdateError* 이벤트 핸들러에서 모든 업데이트 오류를 해결합니다. 업데이트 적용에 대한 자세한 내용은 20-31 페이지의 "레코드 업데이트"를 참조하십시오.

## 내부 프로바이더 구성

*TSQClientDataSet*은 외부 프로바이더 컴포넌트를 사용하지 않기 때문에 데이터를 제공하고 업데이트를 적용하는 방법에 대한 제어가 줄어듭니다. 일반적으로, *TSQClientDataSet*이 필요한 내부 프로바이더의 속성과 이벤트를 게시하기 때문에 이것은 문제가 되지 않습니다.

실행할 SQL 문을 SQL 클라이언트 데이터셋의 *CommandText* 속성 값으로 지정하기 때문에 데이터 패킷에 포함된 데이터를 제어할 수 있습니다. 필드 속성은 클라이언트 데이터셋의 영구적 필드를 사용하여 설정할 수 있습니다.

*TSQClientDataSet*은 내부 프로바이더가 업데이트를 적용하는 방법에 영향을 주는 두 개의 *published* 속성을 가집니다. 이들 속성은 프로바이더의 속성과 같고 *TSQClientDataSet* 컴포넌트는 설정한 값을 내부 프로바이더에 단순히 전달(forward)합니다.

- *Options*는 중첩 디테일 셋과 BLOB 데이터가 데이터 패킷에 포함되는지 또는 별도로 폐치되는지, 삽입, 수정 또는 삭제 같은 특정 타입의 편집이 사용 불가능한지, 단일 업데이트가 여러 서버 레코드에 영향을 줄 수 있는지, 클라이언트 데이터셋의 레코드가 업데이트를 적용할 때 다시 고쳐지는지를 제어합니다. *Options*는 프로바이더의 *Options* 속성과 일치합니다. 결과적으로 *TSQClientDataSet*에 관련되지 않거나 적절하지 않은 옵션을 설정할 수 있게 합니다. 예를 들어, 내부 소스 데이터셋은 영구적인 필드 컴포넌트를 사용하지 않기 때문에 *poIncFieldProps*를 포함할 이유가 없습니다. 이와 유사하게 *poAllowCommandText*는 클라이언트 데이터셋이 실행할 쿼리를 지정할 수 없게 하므로 기본적으로 포함되는 *poAllowCommandText*를 제외하는 것을 원하지 않을 수도 있습니다. 프로바이더의 *Options* 속성에 대한 내용은 21-5 페이지의 "데이터 패킷에 영향을 주는 옵션 설정"을 참조하십시오.
- *UpdateMode*는 프로바이더가 업데이트를 적용하기 위해 생성한 SQL 문에서 레코드를 찾는 데 사용하는 필드를 제어합니다. *UpdateMode*는 프로바이더의 *UpdateMode* 속성과 동일합니다. 프로바이더의 *UpdateMode* 속성에 대한 내용은 21-9 페이지의 "업데이트 적용 방법 변경"을 참조하십시오.

*TSQClientDataSet*은 내부 프로바이더의 이벤트에 해당하는 여러 이벤트도 게시합니다. 이 이벤트 중 가장 중요한 것은 다음과 같습니다.

- *OnGetTableName*. 이 이벤트는 데이터셋이 업데이트를 적용해야 하는 데이터베이스 테이블의 이름을 제공할 수 있게 합니다. 이를 통해 내부 프로바이더는 *Command Text*에 의해 지정된 쿼리에서 데이터베이스 테이블을 식별할 수 없을 때 업데이트에 대한 SQL 문을 생성합니다. 예를 들면, 쿼리가 내장 프로시저나 하나의 테이블에만 업데이트가 필요한 다중 테이블 조인을 실행하는 경우 *OnGetTableName* 이벤트 핸들러를 제공하면 내부 프로바이더가 정확하게 업데이트를 적용할 수 있습니다.
- *BeforeUpdateRecord*. 이 이벤트는 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. 사용자가 만든 SQL 문을 실행하려면 *TSQL Connection* 컴포넌트의 *Execute* 메소드를 사용합니다.
- *OnUpdateError*. 이 이벤트는 내부 프로바이더가 데이터베이스 서버에 업데이트를 적용할 수 없을 때마다 발생합니다. 이 이벤트를 통해 업데이트 프로세스 동안 업데이트 오류를 수정하여 전체 업데이트 작업에서 허용된 최대 오류 수에 이르지 않도록 할 수 있습니다.

## 파일 기반 데이터로 클라이언트 데이터셋 사용

---

클라이언트 데이터셋은 파일 기반 데이터베이스 애플리케이션과 "브리프케이스 모델" 애플리케이션 같이 프로바이더에 독립적으로 작동할 수 있습니다. 데이터에 대해 클라이언트 데이터셋이 사용하는 특수 파일을 MyBase라고 합니다.

MyBase 애플리케이션에서 (프로바이더를 사용하지 않는 경우) 클라이언트 데이터셋은 서버로부터 테이블 정의와 데이터를 얻을 수 없고 업데이트를 적용할 수 있는 서버가 없습니다. 그 대신 클라이언트 데이터셋은 다음을 독립적으로 수행해야 합니다.

- 테이블 정의와 생성
- 저장된 데이터 로드
- 데이터에 편집 내용 합병
- 데이터 저장

## 새로운 데이터셋 생성

---

다음과 같은 세 가지 방법으로 서버 데이터를 나타내지 않는 클라이언트 데이터셋을 정의하고 생성합니다.

- 데이터셋에 대한 영구적 필드를 생성한 다음 컨텍스트 메뉴에서 Create Dataset을 선택하여 새 클라이언트 데이터셋을 정의하고 생성할 수 있습니다.
- 필드 정의와 인덱스 정의를 기반으로 새 클라이언트 데이터셋을 정의하고 생성할 수 있습니다.
- 기존의 데이터셋을 복사할 수 있습니다.

## 영구적 필드를 사용하여 새로운 데이터셋 생성

다음 단계는 Fields Editor를 사용하여 새 클라이언트 데이터셋을 생성하는 방법을 설명합니다.

- 1 컴포넌트 팔레트에서 *TClientDataSet* 컴포넌트를 애플리케이션에 추가합니다.
- 2 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Fields Editor를 선택합니다. Fields Editor에서 마우스 오른쪽 버튼을 클릭하고 New Field 명령을 선택합니다. 필드 정의의 기본 속성을 설명합니다. 필드가 생성되면 Fields Editor에서 필드를 선택하여 Object Inspector에서 속성을 변경할 수 있습니다.  
클라이언트 데이터셋 설명이 끝날 때까지 Fields Editor에 필드를 계속 추가합니다.
- 3 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Create DataSet을 선택합니다. 이 작업은 Fields Editor에서 추가한 영구적 필드에 빈 클라이언트 데이터셋을 생성합니다.
- 4 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭한 다음 Save To File을 선택합니다. 이 명령은 클라이언트 데이터셋이 데이터셋을 포함할 때만 사용할 수 있습니다.
- 5 File Save 대화 상자에서 파일 이름을 선택하고 해당 파일에 클라이언트 데이터셋의 복사본을 저장합니다.

**참고** 클라이언트 데이터셋과 함께 저장한 영구적 필드를 사용하여 런타임에 클라이언트 데이터셋을 생성할 수도 있습니다. 단순히 *CreateDataSet* 메소드를 호출합니다.

## 필드와 인덱스 정의를 사용하여 데이터셋 생성

클라이언트 데이터셋에 대해 필드와 더불어 영구적인 인덱스를 생성하려면 필드와 인덱스 정의를 사용해야 합니다. *FieldDefs* 속성을 사용하여 데이터셋에 필드를 지정하고 *IndexDefs* 속성을 사용하여 인덱스를 지정합니다. 데이터셋이 지정되면 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 디자인 타임에 Create DataSet을 선택하거나 런타임에 *CreateDataSet* 메소드를 호출합니다.

클라이언트 데이터셋에 대한 인덱스 정의를 할 때 인덱스 정의의 두 속성만을 클라이언트 데이터셋에 적용합니다. 두 속성은 *TIndexDef.DescFields*와 *TIndexDef.CaseInsensitiveFields*입니다.

*DescFields*를 통해 어떤 필드에서는 오름차순으로, 다른 필드에서는 내림차순으로 레코드를 정렬하는 인덱스를 정의할 수 있습니다. 인덱스의 모든 필드에서 내림차순으로 정렬하는 *ixDescending* 옵션을 사용하는 대신에 *DescFields*의 값에 따라 내림차순으로 정렬해야 하는 필드만 나열합니다. 예를 들면, Field1, Field2, Field3 순으로 정렬하는 인덱스를 정의할 때 *DescFields*를 다음과 같이 설정합니다.

```
Field1;Field3
```

Field2를 오름차순으로 정렬하고 Field1과 Field3은 내림차순으로 정렬하는 인덱스가 만들어집니다.

*CaseInsFields* 를 사용하여 어떤 필드에서는 대소문자를 구별하고 다른 필드에서는 구별하지 않게 레코드를 정렬하는 인덱스를 정의할 수 있습니다. *isCaseInsensitive* 옵션을 사용하여 인덱스의 모든 필드에서 대소문자를 구별하지 않고 정렬하는 대신, *CaseInsFields*의 값에 따라 대소문자를 구별하지 않고 정렬해야 하는 필드만 나열합니다. *DescFields*처럼 *CaseInsFields*는 필드 이름을 세미콜론으로 구분된 목록으로 나열합니다.

Collection Editor를 사용하여 디자인 타임에 필드와 인덱스 정의를 지정할 수 있습니다. Object Inspector에서 적절한 속성 (*FieldDefs* 또는 *IndexDefs*)을 선택하고 마우스를 더블 클릭하여 Collection Editor를 표시합니다. Collection Editor를 사용하여 정의를 추가, 삭제 및 재배열합니다. Collection Editor에서 정의를 선택하여 Object Inspector의 속성을 편집할 수 있습니다.

런타임에 코드에서 필드와 인덱스 정의를 지정할 수도 있습니다. 예를 들면 다음 코드는 폼의 *OnCreate* 이벤트 핸들러에 클라이언트 데이터셋을 생성하고 활성화합니다.

```
procedure TForm1.FormCreate(Sender:TObject);
begin
  with ClientDataSet1 do
  begin
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftInteger;
      Name := 'Field1';
    end;
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftString;
      Size := 10;
      Name := 'Field2';
    end;
    with IndexDefs.AddIndexDef do
    begin
      Fields := 'Field1';
      Name := 'IntIndex';
    end;
    CreateDataSet;
  end;
end;
```

## 기존 테이블을 기반으로 데이터셋 생성

서버 기반 데이터베이스 애플리케이션을 파일 기반 애플리케이션으로 변환하는 경우 기존 테이블을 복사하여 IDE에서 파일로 저장할 수 있습니다. 다음 단계들은 기존 테이블을 복사하는 방법을 나타냅니다.

- 1 컴포넌트 팔레트의 dbExpress 페이지에서 *TSQLDataSet* 컴포넌트와 *TSQLConnection* 컴포넌트를 애플리케이션에 추가합니다. SQL 데이터셋의 *CommandType*과 *CommandText* 속성을 설정하여 기존 데이터베이스 테이블을 식별합니다. *SQLConnection* 속성을 *TSQLConnection* 컴포넌트로 설정합니다. *TSQLConnection* 컴포넌트의 속성을 설정하여 데이터베이스 서버에 첨부합니다. SQL 데이터셋의 *Active* 속성을 *True*로 설정합니다.

- 2 컴포넌트 팔레트에서 *TClientDataSet* 컴포넌트를 추가합니다.
- 3 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Assign Local Data를 선택합니다. 나타나는 대화상자에서 1 단계에서 추가한 SQL 데이터셋 컴포넌트를 선택합니다. OK를 선택합니다.
- 4 클라이언트 데이터셋을 마우스 오른쪽 버튼으로 클릭하고 Save To File을 선택합니다. 이 명령은 클라이언트 데이터셋이 데이터를 포함해야 사용할 수 있습니다.
- 5 File Save 대화상자에서 파일 이름을 선택하고 데이터베이스 테이블의 복사본을 저장합니다.

런타임에 다른 데이터셋에서 복사하기 위해 데이터를 직접 할당하거나 소스가 다른 클라이언트 데이터셋인 경우 커서를 복제할 수 있습니다.

## 파일이나 스트림에서 데이터 로드

파일에서 데이터를 로드하려면 클라이언트 데이터셋의 *LoadFromFile* 메소드를 호출합니다. *LoadFromFile*은 데이터를 읽어 올 파일을 지정하는 문자열 매개변수가 하나 있습니다. 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*)가 로드하는 데이터는 *LoadFromFile*이 클라이언트 데이터셋의 데이터 서식으로 이전에 저장했거나 *SaveToFile* (*SaveToStream*) 메소드를 사용하여 다른 클라이언트 데이터셋에 저장했어야 합니다. 데이터를 파일이나 스트림에 저장하는 것에 대한 내용은 20-42 페이지의 "파일이나 스트림에 데이터 저장"을 참조하십시오.

*LoadFromFile* 또는 *LoadFromStream*을 호출할 때 파일의 모든 데이터를 *Data* 속성으로 읽습니다. 데이터가 저장될 때 변경 로그에 있는 모든 편집 내용을 *Delta* 속성으로 읽습니다.

## 데이터에 변경 내용 합병

클라이언트 데이터셋의 데이터를 편집하면 변경 내용이 변경 로그에 기록되지만 변경은 데이터의 원래 버전에는 영향을 미치지 않습니다.

변경 내용을 영구적으로 만들려면 *MergeChangeLog*를 호출합니다. *MergeChangeLog*는 변경 로그의 변경된 필드값으로 *Data*의 레코드를 덮어 씁니다.

*MergeChangeLog*를 실행한 후 *Data*는 기존 데이터와 변경 로그에 있는 변경 내용을 혼합하여 포함합니다. 이 혼합은 향후 변경이 수행될 때 새로운 *Data* 기준이 됩니다. *MergeChangeLog*는 모든 레코드의 변경 로그를 지우고 *ChangeCount* 속성을 0으로 재설정합니다.

**경고** 데이터베이스 서버나 소스 데이터셋으로부터 데이터를 나타내는 클라이언트 데이터셋에 대해서는 *MergeChangeLog*를 호출하지 마십시오. 이 경우에는 *ApplyUpdates*를 호출하여 변경 내용을 데이터베이스나 소스 데이터셋에 기록합니다. 자세한 내용은 20-31 페이지의 "업데이트 적용"을 참조하십시오.

**참고** 또한 해당 데이터셋이 원래 *Data* 속성에 제공되었던 경우 각각의 클라이언트 데이터셋의 데이터로 변경 내용을 합병할 수 있습니다. 이 작업을 하려면 데이터셋 프로바이더를 사용해야 합니다. 이 작업을 하는 방법의 예는 20-23 페이지의 "데이터 직접 할당"을 참조하십시오.

## 파일이나 스트림에 데이터 저장

---

파일 기반 애플리케이션에 클라이언트 데이터셋을 사용하면 데이터를 편집하고 합병할 때 수행한 변경 내용은 메모리에만 존재합니다. 변경 내용을 저장하기 위해서는 디스크에 기록해야 합니다. *SaveToFile* 메소드를 사용하여 디스크에 데이터를 저장할 수 있습니다.

*SaveToFile*은 데이터를 기록할 파일을 지정하는 문자열인 하나의 매개변수를 가집니다. 파일 이름은 전체 경로 이름이 될 수도 있습니다. 파일이 이미 존재하는 경우 현재 내용을 완전히 덮어 씁니다.

데이터를 항상 같은 파일에 저장하는 경우 *FileName* 속성을 대신 사용할 수 있습니다. *FileName*을 설정하면 클라이언트 데이터셋이 닫힐 때 데이터를 자동으로 명명된 파일에 저장합니다.

또한 *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 합병하지 않습니다. *LoadFromFile* 또는 *LoadFromStream* 메소드를 사용하여 데이터를 다시 로드하면 변경 로그에는 여전히 합병되지 않은 편집 내용이 있습니다. 이는 변경 내용을 프로바이더 컴포넌트에 언젠가는 적용해야 하는 브리프케이스 모델을 지니는 애플리케이션의 경우에 중요합니다.

**참고** *SaveToFile*은 인덱스 정의를 사용하여 데이터셋과 함께 인덱스를 생성하지 않은 경우에는 클라이언트 데이터셋에 추가한 어떤 인덱스도 보존하지 않습니다.

## 프로바이더 컴포넌트 사용

프로바이더 컴포넌트(*TDataSetProvider*)는 클라이언트 데이터셋이 디스크 상의 전용 파일을 사용하지 않고 데이터를 얻을 수 있는 메커니즘을 제공합니다. 프로바이더는 다음을 수행합니다.

- 클라이언트 데이터셋의 데이터 요청을 받고, 데이터베이스 서버로부터 요청된 데이터를 페치하고, 전송 가능한 데이터 패킷으로 데이터를 패키지와하고, 클라이언트 데이터셋에 데이터를 반환합니다. 이러한 활동을 "공급(providing)"이라고 합니다.
- 클라이언트 데이터셋에서 업데이트된 데이터를 받고, 데이터베이스 또는 소스 데이터셋에 업데이트를 적용하고, 적용할 수 없는 모든 업데이트를 기록하고, 해결되지 않은 업데이트를 나중에 조정하기 위해 클라이언트 데이터셋에 반환합니다. 이러한 활동을 "해결(resolving)"이라고 합니다.

프로바이더 컴포넌트의 대부분의 작업은 자동으로 이루어집니다. 따라서, 다른 데이터셋에 있는 데이터에서 데이터 패킷을 만들거나 데이터셋 또는 데이터베이스 서버에 업데이트를 적용하기 위하여 프로바이더에 코드를 작성할 필요가 없습니다. 그러나, 클라이언트와의 상호 작용을 제어하기 위해서 프로바이더 컴포넌트의 속성 및 이벤트를 사용할 수도 있습니다.

*TSQLClientDataSet*을 사용할 때 프로바이더는 클라이언트 데이터셋 내부에 있기 때문에 애플리케이션이 이를 직접 액세스할 수는 없습니다. 그러나, *TClientDataSet*을 사용할 경우에 프로바이더는 클라이언트를 위해, 그리고 공급 및 해결 프로세스에서 발생하는 이벤트에 응답하기 위해 패키지와된 정보를 제어하는 데 사용할 수 있는 별도의 컴포넌트입니다. *TSQLClientDataSet* 프로바이더의 일부 속성 및 이벤트를 자신의 속성 및 이벤트로 나타내지만 사용자는 많은 양의 제어를 수행하기 위해 별도의 프로바이더 컴포넌트와 함께 *TClientDataSet*을 사용할 수 있습니다.

별도의 프로바이더 컴포넌트 사용 시 클라이언트 데이터셋으로서 동일한 애플리케이션에 상주하거나 다계층 애플리케이션의 일부로서 애플리케이션 서버에 상주할 수 있습니다.

**참고** 클라이언트 데이터셋을 원격 애플리케이션 서버의 프로바이더에 연결하는 컴포넌트는 별도로 구입해야 합니다.

이 장에서는 프로바이더 컴포넌트를 사용하여 클라이언트 데이터셋과의 상호 작용을 제어하는 방법에 대해 설명합니다.

## 데이터의 소스 정하기

---

프로바이더 컴포넌트를 사용할 경우 데이터 패킷으로 만들 데이터를 폐치하기 위해 사용할 수 있는 데이터셋을 지정해야 합니다. 이 데이터셋은 프로바이더의 *DataSet* 속성을 설정하여 지정합니다. 디자인 타임 시 Object Inspector의 *DataSet* 속성 드롭다운 목록에 있는 사용 가능한 데이터셋 중에서 선택합니다.

*TDataSetProvider*는 *IProviderSupport* 인터페이스를 사용하여 소스 데이터셋과 상호 작용합니다. 이 인터페이스는 *TDataSet*에 포함되어 있기 때문에 모든 데이터셋에서 사용할 수 있습니다. 그러나, *TDataSet*에서 구현된 *IProviderSupport* 메소드는 대개 아무 작업도 하지 않거나 예외를 일으키는 스텝(stub)입니다.

Kylix 와 함께 제공된 단방향 데이터셋 클래스는 좀 더 유용한 방식으로 *IProviderSupport* 인터페이스를 구현하기 위해 이러한 메소드를 오버라이드합니다. 클라이언트 데이터셋은 *TDataSet*에서 상속된 기본 구현을 사용하지만 프로바이더의 *ResolveToDataSet* 속성이 *True*이면 소스 데이터셋으로 계속 사용될 수 있습니다.

*TDataSet*으로부터 고유한 사용자 지정 자손을 만드는 컴포넌트 작성자는 자신의 데이터셋이 프로바이더에 데이터를 공급할 경우 모든 해당 *IProviderSupport* 메소드를 오버라이드해야 합니다. 프로바이더가 단지 읽기 전용으로 데이터 패킷을 공급해야 할 경우, 즉 업데이트를 적용할 필요가 없을 경우 *TDataSet*에서 구현되는 *IProviderSupport* 메소드로 충분할 것입니다.

## 클라이언트 데이터셋과의 통신

---

프로바이더와 클라이언트 데이터셋 사이의 모든 통신은 *IAppServer* 인터페이스를 통해 이루어집니다. 프로바이더가 클라이언트와 동일한 애플리케이션에 존재할 경우 이 인터페이스는 사용자를 위해 자동으로 생성된 숨겨진 객체에 의해 구현됩니다. 프로바이더가 다계층 애플리케이션의 일부인 경우에는 애플리케이션 서버의 원격 데이터 모듈을 위한 인터페이스가 됩니다.

*IAppServer*는 클라이언트 데이터셋과 프로바이더 간의 브리지를 제공합니다. 대부분의 애플리케이션은 *IAppServer*를 직접 사용하지 않지만 클라이언트 데이터셋의 속성 및 메소드를 통해 간접적으로 호출합니다. 그러나 필요한 경우 클라이언트 데이터셋의 *AppServer* 속성을 사용하여 *IAppServer* 인터페이스를 직접 호출할 수 있습니다.



표 21.1에는 *IAppServer* 인터페이스의 메소드와 함께 프로바이더 컴포넌트와 클라이언트 데이터셋의 해당 메소드 및 이벤트가 나열되어 있습니다. 이러한 *IAppServer* 메소드에는 *Provider* 매개변수가 포함됩니다. 다계층 애플리케이션에서 이 매개변수는 클라이언트 데이터셋과 서로 통신하는 애플리케이션 서버 상의 프로바이더를 나타냅니다. 또한 대부분의 메소드에는 *OwnerData*라 불리는 *OleVariant* 매개변수가 포함됩니다. 이 매개변수는 클라이언트 애플리케이션과 애플리케이션 서버가 사용자 지정 정보를 앞뒤로 전달할 수 있게 해 줍니다. *OwnerData*는 기본으로 사용되지 않지만 모든 이벤트 핸들러에 전달됩니다. 따라서 사용자는 프로바이더가 클라이언트 데이터셋에서 각 호출을 받기 전후에 애플리케이션 정의 정보를 조정할 수 있게 하는 코드를 작성할 수 있습니다.

**표 21.1** AppServer 인터페이스 멤버

| <b>IAppServer</b>       | <b>TDataSetProvider</b>                                               | <b>TClientDataSet</b>                                                                                      |
|-------------------------|-----------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| AS_ApplyUpdates 메소드     | ApplyUpdates 메소드,<br>BeforeApplyUpdates 이벤트,<br>AfterApplyUpdates 이벤트 | ApplyUpdates 메소드,<br>BeforeApplyUpdates<br>이벤트,<br>AfterApplyUpdates 이벤트                                   |
| AS_DataRequest 메소드      | DataRequest 메소드,<br>OnDataRequest 이벤트                                 | DataRequest 메소드                                                                                            |
| AS_Execute 메소드          | Execute 메소드,<br>BeforeExecute 이벤트,<br>AfterExecute 이벤트                | Execute 메소드,<br>BeforeExecute 이벤트,<br>AfterExecute 이벤트                                                     |
| AS_GetParams 메소드        | GetParams 메소드,<br>BeforeGetParams 이벤트,<br>AfterGetParams 이벤트          | FetchParams 메소드,<br>BeforeGetParams 이벤트,<br>AfterGetParams 이벤트                                             |
| AS_GetProviderNames 메소드 | 사용 가능한 모든 프로바이더<br>를 식별하는 데 사용                                        | ProviderName 속성에 대한<br>디자인 타임 목록을 만드는데<br>사용                                                               |
| AS_GetRecords 메소드       | GetRecords 메소드,<br>BeforeGetRecords 이벤트,<br>AfterGetRecords 이벤트       | GetNextPacket 메소드,<br>Data 속성,<br>BeforeGetRecords 이벤트,<br>AfterGetRecords 이벤트                             |
| AS_RowRequest 메소드       | RowRequest 메소드,<br>BeforeRowRequest 이벤트,<br>AfterRowRequest 이벤트       | FetchBlobs 메소드,<br>FetchDetails 메소드,<br>RefreshRecord 메소드,<br>BeforeRowRequest 이벤트,<br>AfterRowRequest 이벤트 |

## 업데이트 적용 방법 선택

기본적으로 *TDataSetProvider* 컴포넌트는 업데이트를 적용하고 업데이트 오류를 해결할 때 동적으로 생성되는 SQL 문을 사용하여 데이터베이스 서버와 직접 통신합니다. 이러한 접근 방법은 애플리케이션이 소스 데이터셋과 원격 데이터베이스 서버에 대해 차례대로 업데이트를 두 번 병합하지 않아도 된다는 이점이 있습니다.

그러나 이 접근 방법을 사용하지 않을 수도 있습니다. 예를 들어, *TClientDataSet* 컴포넌트에서 데이터를 제공하는 경우처럼 SQL 서버에서 데이터를 가져오지 않는 데이터셋을 사용할 수 있습니다. 또는 SQL 서버에 업데이트를 적용할 수 있는 사용자 지정 데이터셋을 사용하고 있지만 데이터셋 이벤트 중 일부를 사용할 수도 있습니다.

*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*일 경우 업데이트 내용이 데이터셋에 적용되고, *False*일 경우 프로바이더는 SQL 문을 생성하여 업데이트를 적용하고 소스 데이터셋은 데이터베이스 서버에 업데이트를 전달합니다.

## 데이터 패킷에 포함될 정보 제어

클라이언트와 주고 받을 데이터 패킷에 어떤 정보를 포함할 것인지는 여러 가지 방법으로 제어할 수 있습니다. 다음과 같은 방법이 있습니다.

- 데이터 패킷에 표시되는 필드 지정
- 데이터 패킷에 영향을 주는 옵션 설정
- 데이터 패킷에 사용자 지정 정보 추가

### 데이터 패킷에 표시되는 필드 지정

데이터 패킷에 포함될 필드를 제어하려면 패킷을 작성하기 위해 프로바이더가 사용하는 데이터셋에 영구적 필드를 만듭니다. 그러면 프로바이더는 이러한 영구적 필드만 포함합니다. 계산된 필드와 같은 소스 데이터셋에 의해 그 값이 동적으로 생성되는 영구적 필드는 데이터 패킷에 포함될 수 있지만 맨 마지막으로 받는 클라이언트 데이터셋에 대해 정적 읽기 전용 필드로 표시됩니다. 영구적 필드에 대한 자세한 내용은 17-3페이지의 "영구적 필드 컴포넌트"를 참조하십시오.

클라이언트 데이터셋이 데이터를 편집하고 데이터베이스 서버에 업데이트를 적용하는 경우에는 데이터 패킷에 중복된 레코드가 존재하지 않도록 충분한 필드를 포함해야 합니다. 그렇지 않으면 업데이트가 적용될 때 업데이트할 레코드를 결정할 수 없습니다. 단지 고유성을 보장하기 위해 제공되는 추가 필드를 클라이언트 데이터셋이 확인 및 사용할 수 없도록 하려면 이러한 필드의 *ProviderFlags* 속성을 설정하여 *pfHidden*을 포함시킵니다.

**참고** 중복 레코드 방지를 위해 충분한 필드를 포함시키는 것은 소스 데이터셋에서 쿼리를 지정할 때도 고려해야 할 사항입니다. 애플리케이션이 모든 필드를 사용하지 않을 경우에도 레코드 고유성 보장을 위해 쿼리에 충분한 필드를 포함해야 합니다.

## 데이터 패킷에 영향을 주는 옵션 설정

프로바이더 컴포넌트의 *Options* 속성을 사용하면 BLOB 또는 중첩 디테일 테이블을 전달하는 시기, 필드 표시 속성의 포함 여부, 허용할 업데이트 유형 등을 지정할 수 있습니다. 다음 표는 *Options*에 포함할 수 있는 값을 나열한 것입니다.

표 21.2 프로바이더 옵션

| 값                      | 의미                                                                                                                                                                                                                                                                |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| poReadOnly             | 클라이언트 데이터셋은 프로바이더에 업데이트를 적용할 수 없습니다.                                                                                                                                                                                                                              |
| poDisableEdits         | 클라이언트 데이터셋은 기존 데이터 값을 수정할 수 없습니다. 사용자가 필드를 편집하려고 하면 클라이언트 데이터셋은 예외를 일으킵니다. 그러나 레코드를 삽입하거나 삭제하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.                                                                                                                                      |
| poDisableInserts       | 클라이언트 데이터셋은 새 레코드를 삽입할 수 없습니다. 사용자가 새 레코드를 삽입하려고 하면 클라이언트 데이터셋은 예외를 일으킵니다. 그러나 레코드를 삭제하거나 기존 데이터를 수정하는 클라이언트 데이터셋의 기능에는 영향을 주지 않습니다.                                                                                                                              |
| poDisableDeletes       | 클라이언트 데이터셋은 레코드를 삭제할 수 없습니다. 사용자가 레코드를 삭제하려고 하면 클라이언트 데이터셋은 예외를 일으킵니다. 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 <i>FetchOnDemand</i> 속성이 <i>True</i> 이면 이러한 값을 자동으로 요청합니다. 그렇지 않으면 애플리케이션이 클라이언트 데이터셋의 <i>FetchBlobs</i> 메소드를 호출하여 BLOB 데이터를 검색해야 합니다.                                         |
| poFetchDetailsOnDemand | 프로바이더가 마스터/디테일 관계의 마스터를 나타내면 중첩 디테일 값이 데이터 패킷에 포함되지 않습니다. 그 대신 클라이언트 애플리케이션은 이러한 값을 필요에 따라 요청합니다. 클라이언트 데이터셋의 <i>FetchOnDemand</i> 속성이 <i>True</i> 이면 이러한 값을 자동으로 요청합니다. 그렇지 않으면 애플리케이션이 클라이언트 데이터셋의 <i>FetchDetails</i> 메소드를 호출하여 중첩 디테일을 검색해야 합니다.              |
| poIncFieldProps        | 데이터 패킷은 해당 사항이 있을 경우 다음과 같은 필드 속성을 포함합니다. <i>Alignment</i> , <i>DisplayLabel</i> , <i>DisplayWidth</i> , <i>Visible</i> , <i>DisplayFormat</i> , <i>EditFormat</i> , <i>MaxValue</i> , <i>MinValue</i> , <i>Currency</i> , <i>EditMask</i> , <i>DisplayValues</i> |
| poCascadeDeletes       | 프로바이더가 마스터/디테일 관계의 마스터를 나타내면 서버는 마스터 레코드가 삭제될 때 디테일 레코드를 삭제합니다. 이 옵션을 사용하려면 참조 무결성의 일부로 연쇄 삭제를 수행하도록 데이터베이스 서버를 설정해야 합니다.                                                                                                                                         |
| poCascadeUpdates       | 프로바이더가 마스터/디테일 관계의 마스터를 나타내면 마스터 레코드에서 해당 값이 바뀔 때 디테일 테이블의 키 값이 자동으로 업데이트됩니다. 이 옵션을 사용하려면 참조 무결성의 일부로 연쇄 업데이트를 수행하도록 데이터베이스 서버를 설정해야 합니다.                                                                                                                         |

표 21.2 프로바이더 옵션 ( 계속 )

| 값                         | 의미                                                                                                                                                                                     |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 발생할 경우 이벤트 핸들러는 결과적으로 변경되는 다른 레코드가 아니라 업데이트된 레코드에 대한 액세스를 제한한다는 점에 유의하십시오. |
| poNoReset                 | 클라이언트 데이터셋은 데이터를 제공하기 전에 프로바이더가 첫 번째 레코드에 커서를 다시 위치시키도록 지정할 수 없습니다.                                                                                                                    |
| poAutoRefresh             | 프로바이더는 업데이트를 적용할 때마다 클라이언트 데이터셋을 현재 레코드 값으로 새로 고칩니다.                                                                                                                                   |
| poPropagateChanges        | 서버가 업데이트 프로세스의 일부로 업데이트된 레코드를 변경한 내용은 클라이언트로 다시 보내져 클라이언트 데이터셋에 병합됩니다.                                                                                                                 |
| poAllowCommandText        | 클라이언트 데이터셋은 연결된 데이터셋의 SQL 텍스트 또는 자신이 나타내는 테이블 또는 내장 프로시저의 이름을 오버라이드할 수 있습니다.                                                                                                           |
| poRetainServerOrder       | 클라이언트는 기본 순서를 설정하기 위해 레코드를 재정렬해서는 안 됩니다.                                                                                                                                               |

## 데이터 패킷에 사용자 지정 정보 추가

프로바이더는 *OnGetDataSetProperties* 이벤트를 사용하여 애플리케이션 정의 정보를 데이터 패킷에 보낼 수 있습니다. 이 정보는 OleVariant로 인코딩되고 사용자가 지정한 이름으로 저장됩니다. 그러면 클라이언트 데이터셋은 *GetOptionalParam* 메소드를 사용하여 정보를 검색할 수 있습니다. 또한 사용자는 레코드를 업데이트할 때 클라이언트 데이터셋이 보내는 델타 패킷에 정보가 포함되도록 지정할 수 있습니다. 이 경우 클라이언트는 정보를 인식하지 못하지만 프로바이더는 자신에게 왕복 메시지를 보낼 수 있습니다.

*OnGetDataSetProperties* 이벤트에서 사용자 지정 정보를 추가할 때 세 개 요소를 포함한 가변 배열을 사용하여 "옵션 매개변수"라고도 불리는 각각의 개별 속성을 지정합니다. 여기서 세 개 요소는 이름(문자열), 값(가변), 그리고 클라이언트가 업데이트를 적용할 때 델타 패킷에 정보가 포함되는지 여부를 나타내는 부울 플래그를 말합니다. 가변 배열에 대한 가변 배열을 만들어 여러 속성을 추가할 수 있습니다. 예를 들어, 다음 *OnGetDataSetProperties* 이벤트 핸들러는 두 개의 값, 즉 데이터가 제공된 시간과 소스 데이터셋의 총 레코드 수를 보냅니다. 클라이언트 데이터셋이 업데이트를 적용할 경우 데이터가 제공된 시간에 대한 정보만 반환됩니다.

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender:TObject; DataSet:
TDataSet; out Properties: OleVariant);
begin
    Properties := VarArrayCreate([0,1], varVariant);
    Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
    Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

클라이언트 데이터셋이 업데이트를 적용할 때 프로바이더의 *OnUpdateData* 이벤트에서 원래 레코드가 제공된 시간을 읽을 수 있습니다.

```
procedure TMyDataModule1.Provider1UpdateData(Sender:TObject;
DataSet:TCustomClientDataSet);
var
    WhenProvided:TDateTime;
begin
    WhenProvided := DataSet.GetOptionalParam('TimeProvided');
    ...
end;
```

## 클라이언트 데이터 요청에 대한 응답

대부분의 애플리케이션에서 데이터에 대한 요청은 자동으로 처리됩니다. 클라이언트 데이터셋은 *IAppServer* 인터페이스를 통해 간접적으로 *GetRecords*를 호출하여 데이터 패킷을 요청합니다. 프로바이더는 연결된 데이터셋에서 데이터를 페치하고 데이터 패킷을 만든 다음 클라이언트 데이터셋에 패킷을 반환하여 자동으로 응답합니다.

프로바이더는 데이터 패킷을 만든 후 아직 클라이언트 데이터셋에 패킷이 보내지지 않은 상태에서 데이터를 편집할 수 있는 옵션을 갖고 있습니다. 예를 들어, 사용자의 액세스 레벨과 같은 몇 가지 기준에 기초하여 패킷에서 레코드를 제거하거나 다계층 애플리케이션의 경우 클라이언트에 보내기 전 중요한 데이터를 암호화할 수 있습니다.

클라이언트 데이터셋에 보내기 전에 데이터 패킷을 편집하려면 *OnGetData* 이벤트 핸들러를 작성합니다. 데이터 패킷은 클라이언트 데이터셋 형태를 가진 매개변수로 제공됩니다. 이 클라이언트 데이터셋의 메소드를 사용하면 클라이언트에 보내기 전에 데이터를 편집할 수 있습니다.

*IAppServer* 인터페이스를 통해 만들어지는 모든 메소드 호출과 마찬가지로 프로바이더는 *GetRecords*를 호출하기 전후에 클라이언트 데이터셋과 영구적인 상태 정보를 통신할 수 있습니다. 이 통신은 *BeforeGetRecords* 및 *AfterGetRecords* 이벤트 핸들러를 통해 이루어집니다.

## 클라이언트 업데이트 요청에 대한 응답

프로바이더는 클라이언트 데이터셋에서 받은 *델타* 데이터 패킷에 기초하여 데이터베이스 레코드에 업데이트를 적용합니다. 클라이언트 데이터셋은 *IAppServer* 인터페이스를 통해 간접적으로 *ApplyUpdates* 메소드를 호출하여 업데이트를 요청합니다.

*IAppServer* 인터페이스를 통해 만들어지는 모든 메소드 호출과 마찬가지로 프로바이더는 *ApplyUpdates*를 호출하기 전후에 클라이언트 데이터셋과 영구적인 상태 정보를 통신합니다. 이 통신은 *BeforeApplyUpdates* 및 *AfterApplyUpdates* 이벤트 핸들러를 통해 이루어집니다.

업데이트 요청을 받으면 프로바이더는 *OnUpdateData* 이벤트를 생성합니다. 여기서 사용자는 데이터셋에 기록되거나 업데이트 적용 방법에 영향을 주기 전에 델타 패킷을 편집할 수 있습니다. *OnUpdateData* 이벤트 이후 프로바이더는 변경된 내용을 데이터베이스에 기록합니다.

프로바이더는 레코드 단위로 업데이트를 수행합니다. 프로바이더는 각 레코드를 적용하기 전에 *BeforeUpdateRecord* 이벤트를 생성하고 사용자는 이 이벤트를 사용하여 업데이트가 적용되기 전에 스크린할 수 있습니다. 레코드를 업데이트할 때 오류가 발생하면 프로바이더는 오류를 해결할 수 있는 *OnUpdateError* 이벤트를 받습니다. 변경 내용이 서버 제약 조건을 위반하거나 프로바이더가 검색한 이후 클라이언트 데이터셋이 업데이트 적용을 요청하기 전에 데이터베이스 레코드가 다른 애플리케이션에 의해 변경된 경우에 대체로 오류가 발생합니다.

업데이트 오류는 프로바이더 또는 클라이언트 데이터셋에 의해 처리될 수 있습니다. 다계층 애플리케이션의 일부인 경우 프로바이더는 해결을 위해 사용자 상호 작용이 필요하지 않은 모든 업데이트 오류를 처리해야 합니다. 오류 조건을 해결할 수 없으면 프로바이더는 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 데이터셋으로서 업데이트 집합이 전달됩니다. 이벤트 핸들러가 특정 유형의 업데이트만 처리할 경우 레코드의 업데이트 상태에 기초하여 데이터셋을 필터링할 수 있습니다. 레코드를 필터링하면 이벤트 핸들러는 사용하지 않는 레코드를 정렬할 필요가 없어집니다. 레코드의 업데이트 상태에 따라 클라이언트 데이터셋을 필터링하려면 클라이언트 데이터셋의 *StatusFilter* 속성을 설정합니다.

**참고** 단일 테이블을 나타내지 않는 데이터셋에 업데이트를 적용할 경우 애플리케이션은 추가 지원을 제공해야 합니다. 이에 대한 자세한 내용은 21-11페이지의 "단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용"을 참조하십시오.

## 데이터베이스 업데이트 이전의 델타 패킷 편집

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. 여기서 특히 유용한 속성 하나가 *UpdateStatus* 속성입니다. *UpdateStatus*는 델타 패킷의 현재 레코드가 나타내는 수정 유형을 표시하며, 표 21.3에 있는 값들을 지정할 수 있습니다.

**표 21.3** 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 데이터를 삽입합니다.

```

procedure TMyDataModule1.Provider1UpdateData(Sender:TObject;
DataSet:TCustomClientDataSet);
begin
  with DataSet do
    begin
      First;
      while not Eof do
        begin
          if UpdateStatus = usInserted then
            begin
              Edit;
              FieldByName('DateCreated').AsDateTime := Date;
              Post;
            end;
            Next;
          end;
        end;
      end;
    end;

```

## 업데이트 적용 방법 변경

*OnUpdateData* 이벤트는 또한 프로바이더가 데이터베이스에 델타 패킷 레코드가 적용되는 방법을 나타내게 해 줍니다.

기본적으로 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. 일부 필드를 제외하는 방법 중 하나는 프로바이더의 *UpdateMode* 속성을 설정하는 것입니다. *UpdateMode*에는 다음 값이 지정될 수 있습니다.

표 21.4 UpdateMode 값

| 값              | 의미                              |
|----------------|---------------------------------|
| upWhereAll     | 모든 필드를 사용하여 레코드를 찾습니다(WHERE 절). |
| upWhereChanged | 키 필드와 변경된 필드만 사용하여 레코드를 찾습니다.   |
| upWhereOnly    | 키 필드만 사용하여 레코드를 찾습니다.           |

단지 일부 필드를 제외하는 것 말고 더 세부적으로 제어할 수도 있습니다. 예를 들어, 앞의 문장에서 EMPNO 필드를 UPDATE 절에서 제외하여 수정을 방지하고 TITLE 및 DEPT 필드를 WHERE 절에서 제외하여 다른 애플리케이션이 데이터를 수정했을 때 업데이트 충돌을 피할 수 있습니다. 특정 필드가 표시되는 절을 지정하려면 해당 필드의 *ProviderFlags* 속성을 사용합니다. *ProviderFlags*는 표 21.5에 있는 값을 포함할 수 있는 집합입니다.

표 21.5 ProviderFlags 값

| 값          | 설명                                                                                              |
|------------|-------------------------------------------------------------------------------------------------|
| pfInWhere  | UpdateMode가 upWhereAll 또는 upWhereChanged일 경우 생성된 INSERT, DELETE 및 UPDATE 문의 WHERE 절에 필드가 표시됩니다. |
| pfInUpdate | 생성된 UPDATE 문의 UPDATE 절에 필드가 표시될 수 있습니다.                                                         |
| pfInKey    | UpdateMode가 upWhereKeyOnly일 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 핸들러는 TITLE 필드의 업데이트를 허용하고 EMPNO 및 DEPT 필드를 사용하여 원하는 레코드를 찾습니다. 오류가 발생하고 키 필드에만 기초하여 레코드를 찾는 두 번째 시도가 이루어질 경우 생성된 SQL은 EMPNO 필드만 검색합니다.

```
procedure TMyDataModule1.Provider1UpdateData(Sender:TObject;
DataSet:TCustomClientDataSet);
begin
  with DataSet do
    begin
      FieldByName('TITLE').ProviderFlags := [pfInUpdate];
      FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
      FieldByName('DEPT').ProviderFlags := [pfInWhere];
    end;
  end;
```

**참고** 동적으로 생성된 SQL을 사용하지 않고 데이터셋에 직접 업데이트를 시도하는 경우에도 *ProviderFlags* 속성을 사용하여 업데이트가 적용되는 방법에 영향을 줄 수 있습니다. 이러한 플래그는 레코드를 찾기 위해 사용되는 필드와 업데이트할 필드를 결정합니다.

## 개별 업데이트 스크린

프로바이더는 각 업데이트가 적용되기 바로 전에 *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 스크린하거나 거부할 수 있습니다. *BeforeUpdateRecord* 이벤트 핸들러는 업데이트가 이미 처리되었기 때문에 적용되어서는 안된다는 사실을 알려 줍니다. 이 경우 프로바이더는 해당 레코드를 건너뛰지만 업데이트 오류로 간주하지는 않습니다. 예를 들어, 이 이벤트는 이벤트 핸들러에서 레코드가 업데이트된 경우 프로바이더가 자동 처리를 건너뛰도록 하면서 자동으로 업데이트할 수 없는 내장 프로시저에 업데이트를 적용할 수 있는 메커니즘을 제공합니다.

## 프로바이더에서 업데이트 오류 해결

프로바이더가 델타 패킷에서 레코드를 포스트할 때 오류 조건이 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 클라이언트 데이터셋에 다시 전달합니다.

다계층 애플리케이션에서 이 메커니즘은 오류 조건 수정을 위한 클라이언트 애플리케이션의 사용자 상호 작용을 계속 허용하면서 애플리케이션 서버에서 메커니즘적으로 해결할 수 있는 모든 업데이트 오류를 처리할 수 있게 해 줍니다.

*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. 클라이언트 데이터셋에서는 데이터 탐색 메소드를 전혀 사용하지 않지만, 이 데이터셋의 각 필드에 대해서는 *NewValue*, *OldValue* 및 *CurValue* 속성을 사용하여 문제 원인을 확인하고 필요한 내용을 수정하여 업데이트 오류를 해결할 수 있습니다. *OnUpdateError* 이벤트 핸들러는 문제를 수정할 수 있는 경우 *Response* 매개변수를 설정하여 수정된 레코드를 적용합니다.

## 단일 테이블을 나타내지 않는 데이터셋에 업데이트 적용

프로바이더가 데이터베이스 서버에 직접 업데이트를 적용하는 SQL 문을 생성할 때 레코드를 포함하는 데이터베이스 테이블의 이름이 필요합니다. 프로바이더가 내장 프로시저 또는 다중 테이블 쿼리를 원본으로 사용하는 데이터에 업데이트를 적용할 경우 이 테이블 이름을 얻는 것은 문제가 될 수 있습니다. 즉, 이 경우 업데이트를 적용해야 할 테이블의 이름을 쉽게 확인할 수 없습니다.

*OnGetTableName* 이벤트 핸들러에서 프로그램적으로 테이블 이름을 제공할 수 있습니다. 이벤트 핸들러가 테이블 이름을 제공하면 프로바이더는 적절한 SQL 문을 생성하여 업데이트를 적용할 수 있습니다.

**참고** 업데이트 대상이 단일 데이터베이스 테이블인 경우에만, 즉 단일 테이블에 있는 레코드만 업데이트해야 할 경우에 테이블 이름이 제공됩니다. 원본으로 사용하는 여러 데이터베이스 테이블에서 변경이 필요한 업데이트의 경우에는 프로바이더의 *BeforeUpdateRecord* 이벤트를 사용하여 코드에서 업데이트를 명시적으로 적용해야 합니다. 이 이벤트 핸들러가 업데이트를 적용한 후에는 프로바이더가 오류를 생성하지 않도록 *Applied* 매개변수를 *True*로 설정할 수 있습니다.

## 클라이언트 생성 이벤트에 응답

---

프로바이더 컴포넌트는 일반적인 용도의 이벤트, 즉 *OnDataRequest*를 구현하므로 사용자는 이 이벤트로 클라이언트 데이터셋에서 직접 프로바이더를 호출할 수 있습니다.

*OnDataRequest*는 프로바이더가 제공하는 일반적인 기능의 일부가 아니며 클라이언트 데이터셋이 프로바이더와 직접 통신할 수 있도록 해 줍니다. 이벤트 핸들러는 *OleVariant*를 입력 매개변수로 가져와 *OleVariant*를 반환합니다. *OleVariant*를 사용하기 때문에 인터페이스는 매우 일반적이어서 사용자가 프로바이더와 주고 받고자 하는 거의 모든 정보를 수용할 수 있습니다.

애플리케이션은 *OnDataRequest* 이벤트를 생성하기 위해 클라이언트 데이터셋의 *DataRequest* 메소드를 호출합니다.



# 분산 애플리케이션 개발

"분산 애플리케이션 작성"에 포함된 각 장에서는 LAN 또는 인터넷에 분산되는 애플리케이션을 제작하는 데 필요한 개념과 기술을 설명합니다. 일부 Kylix 에디션에서는 여기서 다른 컴포넌트를 사용하지 못할 수도 있습니다.



## 인터넷 서버 애플리케이션 만들기

Kylix를 사용하면 웹 서버 애플리케이션을 CGI 또는 Apache 애플리케이션으로 만들 수 있습니다. 이러한 웹 서버 애플리케이션에는 년비주얼 컴포넌트를 모두 포함할 수 있습니다. Internet 팔레트 페이지에 있는 특수 컴포넌트를 사용하면 특정 URI(Uniform Resource Identifier)와 연결된 이벤트 핸들러를 쉽게 만들 수 있고 처리 완료 시 HTML 문서를 프로그램에서 생성하여 클라이언트에 전송할 수 있습니다. 이러한 컴포넌트와 그 관계를 정의하는 아키텍처는 전체적으로 WebBroker 기술을 바탕으로 한 것입니다.

웹 페이지의 콘텐츠는 종종 데이터베이스에서 가져옵니다. 인터넷 컴포넌트를 사용하면 단일 서버로 무수히 많은 스레드에 대해 안전하게 데이터베이스 연결을 동시에 처리할 수 있게 하면서 데이터베이스에 대한 연결을 자동으로 관리할 수 있습니다.

이 장에서는 이러한 인터넷 컴포넌트를 설명하고 여러 유형의 인터넷 애플리케이션을 만드는 방법을 설명합니다.

### 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 관리됩니다. 인터넷 애플리케이션을 작성할 때에는 몇 가지 중요한 RFC가 유용하게 사용됩니다.

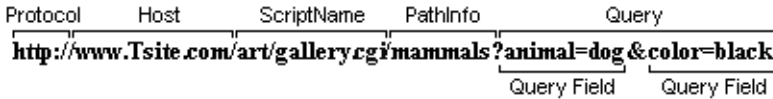
- RFC822, "인터넷 텍스트 메시지의 포맷에 관한 표준"에서는 메시지 헤더의 구조 및 내용을 설명합니다.
- RFC1521, "MIME (Multipurpose Internet Mail Extensions) 1부: 인터넷 메시지 몸체(Body)의 포맷을 지정하고 설명하기 위한 메커니즘"에서는 multipart와 multiformat 메시지를 캡슐화하고 전송하는 데 사용되는 방법을 설명합니다.
- RFC1945, "Hypertext Transfer Protocol - HTTP/1.0"에서는 공동 하이퍼미디어 문서를 배포하는 데 사용되는 전송 메커니즘을 설명합니다.

IETF는 자체 웹 사이트인 [www.ietf.cnri.reston.va.us](http://www.ietf.cnri.reston.va.us)에서 RFC 라이브러리를 유지 관리합니다.

## URL(Uniform Resource Locator)의 각 부분

URL(Uniform Resource Locator)은 네트워크에서 사용할 수 있는 리소스 위치에 대해 완전하게 설명한 것으로, 애플리케이션에서 액세스할 수 있는 여러 부분들로 구성되어 있습니다. 그림 22.1은 URL을 구성하는 부분들을 보여 줍니다.

**그림 22.1** URL(Uniform Resource Locator)의 각 부분



첫 번째 부분은 기술적으로 URL의 일부는 아니지만 프로토콜(http)을 식별합니다. 이 부분은 https(보안 http), ftp 등의 다른 프로토콜을 지정할 수 있습니다.

Host 부분은 웹 서버와 웹 서버 애플리케이션을 실행하는 시스템을 식별합니다. 위의 그림에는 나와 있지 않지만 이 부분은 메시지를 받는 포트를 오버라이드할 수 있습니다. 일반적으로 포트 번호가 프로토콜에 함축되어 있으므로 포트를 지정할 필요가 없습니다.

ScriptName 부분은 웹 서버 애플리케이션의 이름을 지정합니다. 웹 서버는 이 애플리케이션에 메시지를 전달합니다.

스크립트 이름 다음에는 PathInfo가 옵니다. 이 부분은 웹 서버 애플리케이션 내의 메시지의 대상을 식별합니다. PathInfo 값은 호스트 시스템의 디렉토리, 특정 메시지에 응답하는 컴포넌트의 이름, 또는 들어오는 메시지의 처리를 분할하기 위해 웹 서버 애플리케이션이 사용하는 다른 메커니즘이 될 수 있습니다.

Query 부분에는 명명된 값 집합이 포함됩니다. 이러한 값과 그 이름은 웹 서버 애플리케이션에 의해 정의됩니다.

## URI와 URL 비교

URL은 HTTP 표준인 RFC1945에 정의된 URI(Uniform Resource Identifier)의 하위 집합입니다. Web 서버 애플리케이션은 최종 결과가 특정 위치에 있지는 않지만 필요에 따라 만들어지는 많은 소스로부터 콘텐츠를 만드는 경우가 많습니다. URI는 특정 위치와 관련되지 않은 리소스를 정의할 수 있습니다.

## HTTP 요청 헤더 정보

HTTP 요청 메시지에는 클라이언트, 요청 대상, 요청 처리 방법, 요청과 함께 보내진 콘텐츠 등에 대한 정보를 설명하는 여러 개의 헤더가 들어 있습니다. 각 헤더는 문자열 값이 따라오는 "Host" 같은 이름으로 식별합니다. 예를 들어, 다음과 같은 HTTP 요청이라고 가정해 보십시오.

```
GET /art/gallery.cgi/animals?animal=dog&color=black HTTP/1.0
Connection:Keep-Alive
User-Agent: Mozilla/3.0b4Gold (Linux; 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.cgi/animals?animal=dog&color=black)와 연결된 콘텐츠를 반환하도록 웹 서버 애플리케이션에 요청합니다. 첫 번째 줄의 마지막 부분은 클라이언트가 HTTP 1.0 표준을 사용하고 있음을 나타냅니다.

두 번째 줄은 Connection 헤더이며 요청이 서비스되었을 때에도 연결을 끊어서는 안된다는 것을 나타냅니다. 세 번째 줄은 User-Agent 헤더이며 요청을 생성한 프로그램에 대한 정보를 제공합니다. 네 번째 줄은 Host 헤더이며 Host 이름과 연결을 설정하기 위해 연결되는 서버의 포트를 제공합니다. 마지막 줄은 Accept 헤더로 클라이언트가 유효한 응답으로 받을 수 있는 미디어 유형을 나열합니다.

## HTTP 서버 활동

웹 브라우저의 클라이언트/서버 특성은 언뜻 보기엔 간단합니다. 링크만 클릭하면 정보가 화면에 표시되기 때문에 웹에서의 정보 검색은 대부분의 사용자에게 매우 단순한 절차입니다. 반면, 좀 더 경험있는 사용자들은 HTML 문법의 특성과 프로토콜의 클라이언트/서버 특성을 어느 정도는 이해하고 있습니다. 이러한 정도면 일반적으로 간단한 페이지 지향 웹 사이트 콘텐츠를 충분히 만들 수 있습니다. 그러나 더 복잡한 웹 페이지를 만드는 사람은 HTML을 통해 정보 수집 및 표시를 자동화하는 다양한 옵션을 여러 가지 사용합니다.

웹 서버 애플리케이션을 제작하기 전에 클라이언트가 요청을 하는 방법과 서버가 클라이언트 요청에 응답하는 방법을 이해하는 것이 좋습니다.

## 클라이언트 요청 구성

HTML 하이퍼텍스트 링크가 선택되거나 사용자가 URL을 지정하면 브라우저는 프로토콜, 지정된 도메인, 정보 경로, 날짜 및 시간, 운영 환경, 브라우저 자체 등에 대한 정보와 기타 콘텐츠 정보를 수집한 다음 요청을 구성합니다.

예를 들어, 폼에서 버튼을 클릭하여 선택된 기준에 따라 이미지 페이지를 표시하려면 클라이언트는 다음과 같은 URL을 만듭니다.

```
http://www.TSite.com/art/gallery.cgi/animals?animal=dog&color=black
```

이 URL은 www.TSite.com 도메인에 있는 HTTP 서버를 지정합니다. 클라이언트는 www.TSite.com에 접속하고 HTTP 서버에 연결하여 요청을 전달합니다. 이 요청은 다음과 같이 나타냅니다.

```
GET /art/gallery.cgi/animals?animal=dog&color=black HTTP/1.0
Connection:Keep-Alive
User-Agent: Mozilla/3.0b4Gold (Apache; I)
Host:www.TSite.com:1024
Accept:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

## CGI 요청 서비스

웹 서버는 클라이언트 요청을 받고 그 구성에 따른 여러 작업을 수행할 수 있습니다. 요청의 `/gallery.cgi` 부분을 프로그램으로 인식하도록 구성된 경우 웹 서버는 요청에 포함된 정보를 직접 CGI 프로그램에 전달합니다. 웹 서버는 CGI 프로그램이 실행되는 동안 기다립니다. CGI 프로그램이 종료하면 `stdout` 파일을 통해 HTML 콘텐츠가 웹 서버에 다시 전달됩니다.

CGI 프로그램은 언제나 프로그래머의 요청에 따라 작동하고 데이터베이스 액세스, 간단한 테이블 조회 또는 계산, HTML 문서 만들기 또는 선택 등과 같이 프로그래머가 지정한 작업을 수행합니다.

## 동적 공유 객체 요청 서비스

웹 서버 애플리케이션은 클라이언트 요청을 모두 처리했을 때, HTML 코드 페이지 또는 다른 MIME 콘텐츠를 생성하고 클라이언트가 이것을 표시할 수 있도록 서버를 통해 클라이언트로 다시 전달합니다. 동적 공유 객체(DSO)가 모두 처리되면 HTML 페이지와 응답 정보는 직접 서버로 전달되고 서버에서 다시 클라이언트로 전달합니다. 웹 서버 애플리케이션을 공유 객체로 만들면 개별 요청을 서비스하는 데 필요한 프로세스 및 디스크 액세스 수가 감소하여 시스템 로드 및 리소스 사용이 줄어듭니다. DSO에 대한 자세한 내용은 다음 사이트를 참조하십시오.

<http://httpd.apache.org/docs/dso.html>

## 웹 서버 애플리케이션

웹 서버 애플리케이션은 기존 웹 서버의 기능을 확장합니다. 웹 서버 애플리케이션은 웹 서버로부터 HTTP 요청 메시지를 받고 메시지에서 요청한 모든 작업을 수행한 다음 응답을 생성하여 웹 서버로 다시 전달합니다. Kylix 애플리케이션을 사용하여 수행할 수 있는 모든 작업은 웹 서버 애플리케이션에 포함시킬 수 있습니다.

## 웹 서버 애플리케이션의 유형

인터넷 컴포넌트를 사용하면 두 가지 유형의 웹 서버 애플리케이션을 만들 수 있습니다. 이 두 가지 유형은 *TWebApplication*, *TWebRequest* 및 *TWebResponse* 등의 유형별로 지정된 자손을 사용합니다.

**표 22.1** 웹 서버 애플리케이션 컴포넌트

| 애플리케이션 유형     | 애플리케이션 객체                 | 요청 객체                 | 응답 객체                  |
|---------------|---------------------------|-----------------------|------------------------|
| 콘솔 CGI 애플리케이션 | <i>TCGIApplication</i>    | <i>TCGIRequest</i>    | <i>TCGIResponse</i>    |
| Apache DSO 모듈 | <i>TApacheApplication</i> | <i>TApacheRequest</i> | <i>TApacheResponse</i> |



## CGI 독립형

CGI 독립형 웹 서버 애플리케이션은 표준 입력으로 클라이언트 요청 정보를 받아 표준 출력으로 결과를 서버에 전달하는 콘솔 애플리케이션입니다. 이 데이터는 *TCGIRequest* 및 *TCGIResponse* 객체를 만드는 *TCGIApplication*에 의해 평가됩니다. 각 요청 메시지들은 별도의 애플리케이션 인스턴스에 의해 처리됩니다.

## Apache DSO 모듈

**참고** DSO를 지원하려면 소스 코드를 다운로드하여 Apache를 재설치해야 합니다. 22-24 페이지의 "DSO 지원을 위한 Apache 애플리케이션 컴파일"을 참조하십시오.

Apache DSO 모듈은 라이브러리 애플리케이션입니다. Apache 프로그램은 공유 객체를 메모리에 로드한 다음 Apache 프로그램과 모듈 간에 데이터를 전달하는 데 사용되는 메모리 블록을 예약합니다. 이 데이터는 *TApacheRequest* 및 *TApacheResponse* 객체를 만드는 *TApacheApplication*에 의해 평가됩니다.

## 웹 서버 애플리케이션 만들기

모든 새 웹 서버 애플리케이션은 메인 윈도우의 메뉴에서 File|New를 선택하고 New Items 대화 상자에서 Web Server Application을 선택하여 만듭니다. 그러면 다음 웹 서버 애플리케이션 유형 중 하나를 선택할 수 있는 대화 상자가 나타납니다.

- CGI 독립형: 이 애플리케이션 유형을 선택하면 프로젝트가 콘솔 애플리케이션으로 설정되고 프로젝트 파일의 uses 절에 필요한 항목이 추가됩니다.
- Apache DSO 모듈: 이 애플리케이션 유형을 선택하면 프로젝트가 라이브러리 애플리케이션으로 설정되고 프로젝트 파일의 uses 절에 필요한 항목이 추가됩니다.

애플리케이션이 사용하게 될 웹 서버 유형과 통신하는 웹 서버 애플리케이션의 유형을 선택합니다. 그러면 인터넷 컴포넌트를 사용하도록 구성되고 빈 웹 모듈을 포함한 새 프로젝트가 만들어집니다.

## 웹 모듈

웹 모듈(*TWebModule*)은 *TDataModule*의 자손이며, 웹 애플리케이션의 비즈니스 룰과 년비주얼 컴포넌트에 대해서 중앙에서 일괄적으로 제어하기 위해 조상과 동일한 방법으로 사용됩니다.

애플리케이션이 응답 메시지를 생성하는 데 사용하는 콘텐츠 프로듀서를 추가합니다. 이러한 프로듀서에는 *TPageProducer*와 같은 기본 제공 콘텐츠 프로듀서나 사용자가 직접 작성한 *TCustomContentProducer*의 자손이 있습니다. 애플리케이션이 데이터 베이스에서 가져온 자료를 포함하는 응답 메시지를 생성할 경우 사용자는 데이터 액세스 컴포넌트를 추가할 수 있습니다.

웹 모듈은 년비주얼 컴포넌트와 비즈니스 룰을 저장하는 작업 뿐만 아니라, 들어오는 HTTP 요청 메시지를 이러한 요청에 대한 응답을 생성하는 작업 항목에 일치시키는 디스패처 역할도 수행합니다.

웹 애플리케이션에서 사용할 많은 런타임 컴포넌트와 비즈니스 룰과 함께 설정된 데이터 모듈을 이미 갖고 있는 경우가 있습니다. 이럴 경우 웹 모듈을 기존 데이터 모듈로 바꿀 수 있습니다. 자동으로 생성된 웹 모듈을 단순히 삭제하여 사용자의 데이터 모듈로 바꿉니다. 그런 다음, 웹 모듈의 경우처럼 데이터 모듈에 *TWebDispatcher* 컴포넌트를 추가하여 요청 메시지를 작업 항목에 디스패치할 수 있도록 합니다. 들어오는 HTTP 요청 메시지에 응답하기 위해 작업 항목이 선택되는 방법을 변경하려면 *TCustomWebDispatcher*에서 새 디스패처 컴포넌트를 파생시켜 데이터 모듈에 추가합니다.

프로젝트는 단 하나의 디스패처만 포함할 수 있습니다. 디스패처는 프로젝트를 만들 때 자동으로 생성된 웹 모듈이 되거나 웹 모듈을 대체하는 데이터 모듈에 추가된 *TWebDispatcher* 컴포넌트가 될 수 있습니다. 실행 중에 디스패처를 포함한 두 번째 데이터 모듈이 만들어지면 웹 서버 애플리케이션은 런타임 오류를 생성합니다.

**참고** 디자인 타임에 설정한 웹 모듈은 실제적으로는 템플릿이 됩니다.

## 웹 애플리케이션 객체

---

웹 애플리케이션을 위해 설정되는 프로젝트에는 *Application*이라는 전역 변수가 포함됩니다. *Application*은 사용자가 만들고 있는 애플리케이션 유형에 해당하는 *TWebApplication*(*TApacheApplication* 또는 *TCGIApplication*)의 자손이며, 웹 서버로부터 받은 HTTP 요청 메시지에 응답하여 실행됩니다.

**경고** 프로젝트의 **uses** 절에서 CGIApp 또는 ApacheApp 유닛 뒤에 폼 유닛을 포함해서는 안됩니다. 폼도 *Application*이라는 전역 변수를 선언하기 때문에 CGIApp 또는 ApacheApp 유닛 뒤에 올 경우 *Application*이 잘못된 유형의 객체로 초기화됩니다.

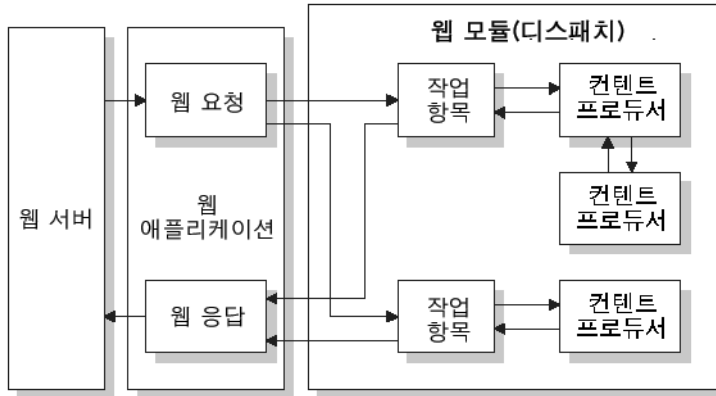
## 웹 서버 애플리케이션의 구조

---

웹 애플리케이션은 HTTP 요청 메시지를 받으면 *TWebRequest* 객체를 만들어 HTTP 요청 메시지를 나타내고 *TWebResponse* 객체를 만들어 반환해야 할 응답을 나타냅니다. 그런 다음, 웹 애플리케이션은 이 객체들을 웹 디스패처(웹 모듈 또는 *TWebDispatcher* 컴포넌트)에 전달합니다.

웹 디스패처는 웹 서버 애플리케이션의 흐름을 제어합니다. 디스패처는 특정 유형의 HTTP 요청 메시지에 대한 처리 방법을 알고 있는 작업 항목(*TWebActionItem*)의 컬렉션을 유지 관리합니다. 디스패처는 적절한 작업 항목 또는 자동 디스패칭 컴포넌트를 식별하여 HTTP 요청 메시지를 처리하며, 요청된 모든 작업을 수행하거나 응답 메시지를 생성할 수 있도록 요청 및 응답 객체를 식별된 핸들러에 전달합니다. 자세한 내용은 22-7 페이지의 "웹 디스패처"에 설명되어 있습니다.

그림 22.2 서버 애플리케이션의 구조



작업 항목은 요청을 읽고 응답 메시지를 결합합니다. 특수한 콘텐츠 프로듀서 컴포넌트는 작업 항목이 사용자 지정 HTML 코드 또는 다른 MIME 콘텐츠를 포함할 수 있는 응답 메시지의 콘텐츠를 동적으로 생성하는 데 도움을 줍니다. 콘텐츠 프로듀서는 다른 콘텐츠 프로듀서나 *THTMLTagAttributes*의 자손을 사용하여 작업 항목이 응답 메시지의 콘텐츠를 만들 수 있게 합니다. 콘텐츠 프로듀서에 대한 자세한 내용은 22-16 페이지의 에서 "응답 메시지 콘텐츠 생성"을 참조하십시오.

모든 작업 항목 또는 자동 디스패칭 컴포넌트가 *TWebResponse* 객체를 채워 응답 생성을 끝내면 디스패처는 결과를 웹 애플리케이션에 전달합니다. 그러면 웹 애플리케이션은 웹 서버를 통해 클라이언트에게 응답을 보냅니다.

## 웹 디스패처

웹 모듈을 사용하고 있다면 웹 모듈이 웹 디스패처 역할도 합니다. 이미 존재하는 데이터 모듈을 사용하는 중이라면 *TWebDispatcher* 디스패처 컴포넌트 하나를 그 데이터 모듈에 추가해야 합니다. 디스패처는 특정 유형의 HTTP 요청 메시지에 대한 처리 방법을 알고 있는 작업 항목의 컬렉션을 유지 관리합니다. 웹 애플리케이션이 요청 객체와 응답 객체를 디스패처에 전달하면 디스패처는 요청에 응답하기 위해 하나 이상의 작업 항목을 선택합니다.

## 디스패처에 작업 추가

Object Inspector에서 디스패처의 *Actions* 속성의 생략 버튼을 클릭하여 작업 편집기를 엽니다. 이 작업 편집기에서 Add 버튼을 클릭하면 디스패처에 작업 항목을 추가할 수 있습니다.

다른 요청 메소드나 대상 URI에 응답하려면 디스패처에 작업을 추가합니다. 작업 항목은 다양한 방법으로 설정할 수 있습니다. 요청을 미리 처리하는 작업 항목에서 시작하여 응답을 보냈는지, 아니면 오류 코드를 반환했는지의 응답 완료 여부를 검사하는 기본 작업으로 마무리할 수 있습니다. 또는 각 작업 항목이 요청을 완벽하게 처리할 수 있도록 모든 유형의 요청에 대해 별도의 작업 항목을 추가할 수 있습니다.

작업 항목은 22-8 페이지의 "작업 항목"에 더 자세히 설명되어 있습니다.

## 요청 메시지 디스패칭

---

클라이언트 요청을 받으면 디스패처는 *BeforeDispatch* 이벤트를 생성합니다. 이 이벤트가 생성되면 사용자가 작업 항목으로 확인하기 전에 애플리케이션에서 요청 메시지를 미리 처리할 수 있는 기회를 제공합니다.

그런 다음 디스패처는 요청 메시지에 있는 대상 URL의 경로 정보 부분과 일치하고 요청 메시지의 메소드로서 지정된 서비스를 제공할 수 있는 항목을 작업 항목 목록에서 검색합니다. 즉, *TWebRequest* 객체의 *PathInfo* 및 *MethodType* 속성을 작업 항목에 있는 같은 이름의 속성과 비교합니다.

적절한 작업 항목을 찾으면 디스패처는 해당 작업 항목을 실행시키며, 이 때 작업 항목은 다음 중 하나를 실행합니다.

- 응답 콘텐츠를 완성하고 요청이 완전히 처리된 응답 또는 신호를 보냅니다.
- 응답을 추가하고 다른 작업 항목이 작업을 완료할 수 있게 해 줍니다.
- 요청을 다른 작업 항목에 전달합니다.

모든 작업 항목을 검사한 후 요청 메시지가 완전히 처리되지 않은 경우 디스패처는 기본 작업 항목을 호출합니다. 기본 작업 항목은 대상 URL이나 요청 메소드와 반드시 일치할 필요는 없습니다.

기본 작업이 있어서 디스패처가 기본 작업을 포함한 작업 목록의 끝에 도달했고 트리거된 작업이 없을 경우 서버에는 아무 것도 전달되지 않습니다. 이 경우 서버는 클라이언트와의 연결을 끊습니다.

요청이 작업 항목에 의해 처리될 경우 디스패처는 *AfterDispatch* 이벤트를 생성합니다. 이 이벤트가 생성되면 애플리케이션은 생성된 응답을 검사하여 최종적으로 필요한 내용을 변경할 수 있습니다.

## 작업 항목

---

각 작업 항목(*TWebActionItem*)은 주어진 유형의 요청 메시지에 응답하여 특정 작업을 수행합니다.

작업 항목은 요청에 완전하게 응답하거나 응답의 일부를 수행하고 다른 작업 항목이 작업을 완료하도록 허용할 수 있습니다. 작업 항목은 요청에 대해 HTTP 응답 메시지를 보내거나 다른 작업 항목이 완료할 수 있도록 응답의 일부를 설정할 수 있습니다. 작업 항목이 응답을 완료하고 보내지 않았을 경우 웹 서버 애플리케이션이 응답 메시지를 보냅니다.

## 작업 항목 실행 시기 결정

대부분의 작업 항목 속성은 HTTP 요청 메시지를 처리하기 위해 디스패처가 작업 항목을 선택하는 시기를 결정합니다. 작업 항목의 속성을 설정하려면 Object Inspector에서 디스패처의 *Actions* 속성을 선택하고 생략 버튼을 클릭하여 작업 편집기를 열어야 합니다. 작업 편집기에서 작업을 선택하면 Object Inspector에서 해당 작업의 속성을 수정할 수 있습니다.

### 대상 URL

디스패처는 작업 항목의 *PathInfo* 속성과 요청 메시지의 *PathInfo*를 비교합니다. 이 속성 값은 작업 항목이 처리할 준비가 된 모든 요청의 URL에 있는 경로 정보 부분이어야 합니다. 예를 들어, 다음과 같은 URL이 있다고 가정해 보십시오.

`http://www.TSite.com/art/gallery.cgi/mammals?animal=dog&color=black`

여기서 `/gallery.cgi` 부분이 웹 서버 애플리케이션을 나타낸다고 가정하면 다음 부분이 경로 정보입니다.

`/mammals`

요청을 서비스할 때 웹 애플리케이션이 정보를 검색해야 할 장소를 나타내거나 웹 애플리케이션을 논리적 하위 서비스로 나누기 위해 경로 정보를 사용합니다.

### 요청 메소드 유형

작업 항목의 *MethodType* 속성은 처리할 수 있는 요청 메시지 유형을 나타냅니다. 디스패처는 작업 항목의 *MethodType* 속성과 요청 메시지의 *MethodType*을 비교합니다. *MethodType*은 다음 값들 중 하나를 가질 수 있습니다.

표 22.2 MethodType 값

| 값             | 의미                                                                                      |
|---------------|-----------------------------------------------------------------------------------------|
| <i>mtGet</i>  | 요청은 응답 메시지로 반환될 대상 URI와 연결된 정보를 요구합니다.                                                  |
| <i>mtHead</i> | 요청은 <i>mtGet</i> 요청에 서비스하지만 응답 콘텐츠를 생략하는 경우처럼 응답의 헤더 속성을 요구합니다.                         |
| <i>mtPost</i> | 요청은 웹 애플리케이션에 포스트될 정보를 제공합니다.                                                           |
| <i>mtPut</i>  | 요청은 대상 URI와 연결된 리소스를 요청 메시지의 콘텐츠로 바꾸도록 요구합니다.                                           |
| <i>mtAny</i>  | <i>mtGet</i> , <i>mtHead</i> , <i>mtPut</i> 및 <i>mtPost</i> 를 비롯한 모든 요청 메소드 유형을 일치시킵니다. |

### 작업 항목 활성화 및 비활성화

각 작업 항목에는 해당 작업 항목을 활성화 또는 비활성화하는 데 사용할 수 있는 *Enabled* 속성이 있습니다. *Enabled*를 *False*로 설정하면 작업 항목이 비활성화되므로 디스패처가 요청 처리를 위해 작업 항목을 검색할 때 그 작업 항목은 고려하지 않습니다.

*BeforeDispatch* 이벤트 핸들러는 디스패처가 작업 항목을 요청 메시지와 일치시키기 전에 작업 항목의 *Enabled* 속성을 변경하여 작업 항목이 처리해야 할 요청을 제어할 수 있습니다.

**주의** 실행 도중에 작업의 *Enabled* 속성을 변경하면 후속 요청에 대해 예기치 않은 결과가 발생할 수 있습니다. 모든 작업 항목을 해당 시작 상태로 올바르게 초기화하려면 *Before Dispatch* 이벤트를 사용합니다.

## 기본 작업 항목 선택

작업 항목 중 하나만 기본 작업 항목이 될 수 있습니다. 기본 작업 항목을 선택하려면 해당 작업 항목의 *Default* 속성을 *True*로 설정합니다. 특정 작업 항목의 *Default* 속성이 *True*로 설정되면 이전 기본 작업 항목이 있을 경우에는 *Default* 속성은 *False*로 설정됩니다.

디스패처는 요청 처리를 위해 작업 항목 목록을 검색하여 특정 작업 항목을 선택할 때 기본 작업 항목의 이름을 저장합니다. 작업 항목 목록의 끝에 도달했지만 요청이 완전히 처리되지 않은 경우 디스패처는 기본 작업 항목을 실행합니다.

디스패처는 기본 작업 항목의 *PathInfo* 또는 *MethodType*을 검사하지 않습니다. 또한 디스패처는 기본 작업 항목의 *Enabled* 속성도 검사하지 않습니다. 따라서 기본 작업 항목의 *Enabled* 속성을 *False*로 설정함으로써 맨 끝 부분에서만 기본 작업 항목을 호출할 수 있습니다.

잘못된 URI 또는 *MethodType*을 나타내는 오류 코드만 반환될 경우에도 기본 작업 항목은 발생하는 모든 요청을 처리하도록 준비되어야 합니다. 기본 작업 항목이 요청을 처리하지 않을 경우 웹 클라이언트에 응답이 보내지지 않습니다.

**주의** 실행 도중에 작업의 *Default* 속성을 변경하면 현재 요청에 대해 예기치 않은 결과가 생길 수 있습니다. 이미 트리거된 작업의 *Default* 속성을 *True*로 설정할 경우 해당 작업은 다시 평가되지 않고 작업 목록의 끝에 도달했을 때 디스패처는 이 작업을 실행하지 않습니다.

## 작업 항목으로 요청 메시지에 응답

작업 항목은 실행 시 웹 서버 애플리케이션의 실제 작업을 수행합니다. 웹 디스패처가 작업 항목을 실행시키면 해당 작업 항목은 다음 두 가지 방법으로 현재 요청 메시지에 응답할 수 있습니다.

- 작업 항목이 *Producer* 속성 값으로 연결된 프로듀서 컴포넌트를 가질 경우 해당 프로듀서는 *Content* 메소드를 사용하여 응답 메시지의 *Content*를 자동으로 지정합니다. 컴포넌트 팔레트의 인터넷 페이지에는 응답 메시지 콘텐츠의 HTML 페이지를 만드는 데 도움이 되는 여러 콘텐츠 프로듀서 컴포넌트가 포함되어 있습니다.
- 연결된 프로듀서가 있고 이 프로듀서가 응답을 지정한 후 작업 항목은 *OnAction* 이벤트를 받습니다. *OnAction* 이벤트 핸들러에는 모든 응답 정보를 채울 *TWebResponse* 객체와 HTTP 요청 메시지를 나타내는 *TWebRequest* 객체가 전달됩니다.

하나의 콘텐츠 프로듀서가 작업 항목의 내용을 생성할 수 있을 경우 가장 간단한 방법은 콘텐츠 프로듀서를 작업 항목의 *Producer* 속성으로 지정하는 것입니다. 그러나 *OnAction* 이벤트 핸들러에서 언제든지 모든 콘텐츠 프로듀서에 액세스할 수 있습니다. *OnAction* 이벤트 핸들러는 여러 콘텐츠 프로듀서를 사용하고 응답 메시지 속성을 지정하는 등의 작업을 수행할 수 있도록 더 많은 유연성을 부여합니다.

컨텐츠 프로듀서 컴포넌트와 *OnAction* 이벤트 핸들러는 모두 객체 또는 런타임 라이브러리 메소드를 사용하여 요청 메시지에 응답합니다. 또한 데이터베이스 액세스, 계산 수행, HTML 문서 만들기 또는 선택 등의 작업을 할 수 있습니다. 컨텐츠 프로듀서 컴포넌트를 사용한 응답 컨텐츠 생성에 대한 자세한 내용은 22-16 페이지의 "응답 메시지 컨텐츠 생성"을 참조하십시오.

## 응답 보내기

*OnAction* 이벤트 핸들러는 *TWebResponse* 객체의 메소드를 사용하여 웹 클라이언트에게 응답을 돌려 보낼 수 있습니다. 그러나 클라이언트에게 응답을 보내는 작업 항목이 없으면 마지막 작업 항목이 요청이 처리되었다는 것을 알 때까지 웹 서버 애플리케이션에서 계속하여 요청을 보냅니다.

## 여러 작업 항목 사용

단일 작업 항목에서 요청에 응답하거나 여러 작업 항목 간에 작업을 분할할 수 있습니다. 작업 항목이 응답 메시지 설정을 완전히 끝내지 않았을 경우 작업 항목은 *Handled* 매개변수를 *False*로 설정하여 *OnAction* 이벤트 핸들러에서 이러한 상태를 알려야 합니다.

요청 메시지에 응답하는 작업이 여러 작업 항목 간에 분할되면 다른 작업 항목이 작업을 계속할 수 있도록 각 *Handled* 매개변수를 *False*로 설정합니다. 이 때 기본 작업 항목에서는 *Handled* 매개변수를 *True*로 설정된대로 두어야 합니다. 그렇지 않으면 웹 클라이언트에게 응답이 보내지지 않습니다.

여러 작업 항목 간에 작업을 분할할 때 기본 작업 항목의 *OnAction* 이벤트 핸들러 또는 디스패치의 *AfterDispatch* 이벤트 핸들러는 모든 작업이 수행되었는지 검사하고 그렇지 않을 경우 적절한 오류 코드를 설정해야 합니다.

## 클라이언트 요청 정보 액세스

웹 서버 애플리케이션이 HTTP 요청 메시지를 받으면 클라이언트 요청의 헤더가 *TWebRequest* 객체의 속성으로 로드됩니다. Apache DSO 애플리케이션에서 요청 메시지는 *TApacheRequest* 객체에 의해 캡슐화됩니다. 콘솔 CGI 애플리케이션은 *TCGIRequest* 객체를 사용합니다.

요청 객체의 속성은 읽기 전용입니다. 이러한 속성을 사용하면 클라이언트 요청에서 사용할 수 있는 모든 정보를 수집할 수 있습니다.

## 요청 헤더 정보를 포함하는 속성

요청 객체 속성은 대부분 HTTP 요청 헤더에서 가져온 요청에 대한 정보를 포함합니다. 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 객체 속성 중 하나로 표시되지 않는 요청 헤더 필드의 값을 얻으려면 *GetFieldByName* 메소드를 사용하십시오.

## 대상을 식별하는 속성

요청 메시지의 전체 대상은 *URL* 속성에 의해 주어집니다. 일반적으로 요청 메시지의 전체 대상은 프로토콜(HTTP), *Host*(서버 시스템), *ScriptName*(서버 애플리케이션), *PathInfo*(호스트 상의 위치) 및 *Query*로 나눌 수 있는 URL입니다.

이러한 각 부분은 고유한 속성으로 표시됩니다. 프로토콜은 항상 HTTP이고 *Host* 및 *ScriptName*은 웹 서버 애플리케이션을 식별합니다. 디스패처는 작업 항목을 요청 메시지에 일치시킬 때 *PathInfo* 부분을 사용합니다. 일부 요청에서는 요청된 정보의 세부 사항을 지정하기 위해 *Query*가 사용되며, 이 부분의 값도 *QueryFields* 속성으로 구분 분석됩니다.

## 웹 클라이언트를 설명하는 속성

또한 요청에는 요청이 시작된 장소에 대한 정보를 제공하는 여러 속성이 포함되어 있습니다. 이러한 속성은 보낸 사람의 전자 메일 주소(*From* 속성)에서 메시지가 시작된 URI(*Referer* 또는 *RemoteHost* 속성)에 이르기까지 모든 정보를 포함합니다. 요청에 콘텐츠가 포함되어 있고 이 콘텐츠가 요청과 동일한 URI에서 가져온 것이 아닌 경우 *DerivedFrom* 속성에 의해 콘텐츠의 소스가 주어집니다. 이외에도 클라이언트의 IP 주소(*RemoteAddr* 속성) 및 요청을 보낸 애플리케이션의 이름과 버전(*UserAgent* 속성)을 확인할 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                                                           |
|----------------|------------------------------------------------------------------------|
| <i>OPTIONS</i> | 사용할 수 있는 통신 옵션에 대한 정보                                                  |
| <i>GET</i>     | <i>URL</i> 속성에 의해 식별된 정보                                               |
| <i>HEAD</i>    | 응답 콘텐츠 없이 해당되는 GET 메시지에서 가져온 헤더 정보                                     |
| <i>POST</i>    | <i>Content</i> 속성에 포함된 데이터를 적절하게 포스트하도록 서버 애플리케이션에 요청                  |
| <i>PUT</i>     | <i>URL</i> 속성이 나타낸 리소스를 <i>Content</i> 속성에 포함된 데이터로 바꾸도록 서버 애플리케이션에 요청 |
| <i>DELETE</i>  | <i>URL</i> 속성에 의해 식별되는 리소스를 삭제하거나 숨기도록 서버 애플리케이션에 요청                   |
| <i>TRACE</i>   | 요청 수신을 확인하는 루프백을 보내도록 서버 애플리케이션에 요청                                    |



*Method* 속성은 웹 클라이언트가 요청하는 서버의 다른 메소드를 나타낼 수 있습니다.

웹 서버 애플리케이션은 *Method*의 모든 가능한 값에 대해 응답을 제공할 필요가 없습니다. 단, HTTP 표준에서는 GET 및 HEAD 요청 모두에 대해 서비스할 것을 요구합니다.

*MethodType* 속성은 *Method* 값이 GET(mtGet), HEAD(mtHead), POST(mtPost), PUT(mtPut) 또는 다른 문자열(mtAny) 중에서 무엇인지 나타냅니다. 디스패처는 *MethodType* 속성의 값을 각 작업 항목의 *MethodType*과 일치시킵니다.

## 예상 응답을 설명하는 속성

*Accept* 속성은 웹 클라이언트가 응답 메시지의 콘텐츠로 적용하는 미디어 유형을 나타냅니다. *IfModifiedSince* 속성은 클라이언트가 최근에 변경된 정보만 원하는지를 지정합니다. *Cookie* 속성은 일반적으로 애플리케이션에서 이전에 추가한 상태 정보를 포함하고 있으며 여기서 응답을 수정할 수 있습니다.

## 컨텐츠를 설명하는 속성

대부분의 요청은 정보에 대한 요청이기 때문에 콘텐츠를 포함하지 않습니다. 그러나 POST 요청과 같은 일부 요청은 웹 서버 애플리케이션이 사용할 콘텐츠를 제공합니다. 콘텐츠의 미디어 유형은 *ContentType* 속성에서 제공되고 그 길이는 *ContentLength* 속성에서 제공됩니다. 데이터 압축과 같이 메시지 콘텐츠가 인코딩된 경우에는 *ContentEncoding* 속성에 정보가 들어 있습니다. 콘텐츠를 만든 애플리케이션의 이름과 버전 번호는 *ContentVersion* 속성에 의해 지정됩니다. 또한 *Title* 속성이 콘텐츠에 대한 정보를 제공할 수 있습니다.

## HTTP 요청 메시지의 내용

헤더 필드 이외에 일부 요청 메시지는 웹 서버 애플리케이션이 몇몇 방법으로 처리해야 할 콘텐츠 부분을 포함합니다. 예를 들어, POST 요청은 웹 서버 애플리케이션이 유지 관리하는 데이터베이스에 추가해야 할 정보를 포함할 수 있습니다.

컨텐츠의 처리되지 않은 값은 *Content* 속성에 의해 주어집니다. 앰퍼샌드(&)로 분리된 필드로 콘텐츠를 구문 분석할 수 있는 경우 *ContentFields* 속성에서 구문 분석된 버전을 사용할 수 있습니다.

## HTTP 응답 메시지 만들기

들어오는 HTTP 요청 메시지에 대해 *TWebRequest* 객체를 만들 경우 웹 서버 애플리케이션은 반환 시 보내질 응답 메시지를 나타내는 해당 *TWebResponse* 객체도 함께 만듭니다. Apache DSO 애플리케이션에서 응답 메시지는 *TApacheResponse* 객체에 의해 캡슐화됩니다. 콘솔 CGI 애플리케이션은 *TCGIResponse* 객체를 사용합니다.

웹 클라이언트 요청에 대한 응답을 생성하는 작업 항목은 응답 객체의 속성을 채웁니다. 어떤 경우는 오류 코드를 반환하거나 요청을 다른 URI에 리디렉션하는 것만큼 작업이 간단할 수 있지만, 또 다른 경우 다른 소스에서 정보를 페치하여 완성된 폼으로 결합하도록 작업 항목에 요청하는 복잡한 계산을 포함할 수 있습니다. 요청 메시지가 단지 요청된 작업이 수행되었다는 사실을 알리는 것에 불과할지라도 대부분의 요청 메시지는 응답을 필요로 합니다.

## 응답 헤더 채우기

*TWebResponse* 객체의 속성 대부분은 웹 클라이언트로 돌려 보내지는 HTTP 응답 메시지의 헤더 정보를 나타냅니다. 작업 항목은 자신의 *OnAction* 이벤트 핸들러에서 이러한 속성을 설정합니다.

모든 응답 메시지가 헤더 속성 모두에 대해 값을 지정할 필요는 없습니다. 설정해야 할 속성은 요청의 특성과 응답 상태에 따라 달라집니다.

## 응답 상태 표시

모든 응답 메시지는 응답 상태를 나타내는 상태 코드를 포함해야 합니다. 상태 코드는 *StatusCode* 속성을 설정하여 지정할 수 있습니다. HTTP 표준은 이미 정의된 의미로 다수의 표준 상태 코드를 정의합니다. 이외에도 사용자는 사용되지 않은 가능한 값을 사용하여 고유한 상태 코드를 정의할 수 있습니다.

각 상태 코드는 다음과 같이 세 자리 숫자이며, 여기서 최상위 숫자가 응답의 종류를 나타냅니다.

- 1xx: 정보(요청을 받았지만 완전히 처리하지 않았습니다.)
- 2xx: 성공(요청을 받아 이해하고 승인하였습니다.)
- 3xx: 리디렉션(요청을 완료하기 위해 클라이언트가 추가 작업을 수행해야 합니다.)
- 4xx: 클라이언트 오류(요청을 이해하거나 서비스할 수 없습니다.)
- 5xx: 서버 오류(요청은 유효하지만 서버가 요청을 처리할 수 없습니다.)

각 상태 코드에 연결된 문자열은 상태 코드의 의미를 설명합니다. 이 문자열은 *ReasonString* 속성에 의해 주어집니다. 이미 정의된 상태 코드의 경우 *ReasonString* 속성을 설정할 필요가 없습니다. 고유한 상태 코드를 정의할 경우에는 *ReasonString* 속성도 설정해야 합니다.

## 클라이언트 작업에 대한 요구 표시

상태 코드의 범위가 300–399 사이인 경우 웹 서버 애플리케이션이 요청을 완료할 수 있으려면 클라이언트가 추가 작업을 수행해야 합니다. 클라이언트를 다른 URI로 리디렉션해야 하거나 요청 처리를 위해 새 URI를 만들었다는 것을 나타내야 한다면 *Location* 속성을 설정합니다. 클라이언트가 암호를 제공해야만 사용자가 작업을 계속 진행할 수 있는 경우에는 *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 사용되는 애플리케이션의 이름과 버전 번호를 제공합니다. *Cookies* 속성은 후속 요청 메시지에 포함되는 클라이언트의 서버 애플리케이션 사용에 대한 상태 정보를 유지할 수 있습니다.

## 컨텐츠 설명

몇 가지 속성들은 응답 콘텐츠를 설명합니다. *ContentType*은 응답의 미디어 유형을 제공하고 *ContentVersion*은 해당 미디어 유형에 대한 버전 번호이며 *ContentLength*는 응답의 길이를 제공합니다. 데이터 압축의 경우와 같이 콘텐츠가 인코딩될 경우 *ContentEncoding* 속성을 사용하여 나타냅니다. 콘텐츠를 다른 URI에서 가져온 경우 *DerivedFrom* 속성에서 이를 나타내야 합니다. 콘텐츠 값이 시간에 민감한 경우에는 해당 값이 여전히 유효한지 여부를 *LastModified* 및 *Expires* 속성이 나타냅니다. *Title* 속성은 콘텐츠에 대한 설명 정보를 제공할 수 있습니다.

## 응답 콘텐츠 설정

---

일부 요청의 경우 요청 메시지에 대한 응답이 응답의 헤더 속성에 전부 포함됩니다. 그러나 대부분의 경우 작업 항목은 응답 메시지에 일부 콘텐츠를 지정합니다. 파일에 저장된 정적 정보나 작업 항목, 또는 콘텐츠 프로듀서에 의해 동적으로 만들어진 정보도 콘텐츠가 될 수 있습니다.

*Content* 속성이나 *ContentStream* 속성을 사용하면 응답 메시지의 콘텐츠를 설정할 수 있습니다.

*Content* 속성은 문자열입니다. Kylix 문자열은 텍스트 값에만 국한되는 것이 아니므로 *Content* 속성 값은 HTML 명령의 문자열 또는 비트 스트림과 같은 그래픽 콘텐츠나 다른 MIME 콘텐츠 형식이 될 수 있습니다.

응답 메시지의 콘텐츠를 스트림에서 읽을 수 있는 경우, *ContentStream* 속성을 사용합니다. 예를 들어, 응답 메시지가 파일 콘텐츠를 보내야 할 경우에는 *ContentStream* 속성에 대해 *TFileStream* 객체를 사용합니다. *Content* 속성과 마찬가지로 *ContentStream*은 HTML 명령의 문자열이나 다른 MIME 콘텐츠 형식을 제공할 수 있습니다. *ContentStream* 속성을 사용할 경우에는 웹 응답 객체가 자동으로 스트림을 해제하므로 사용자가 직접 스트림을 해제하지 마십시오.

**참고** *ContentStream* 속성 값이 **nil**이 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 위한 두 개의 메소드, 즉 *SendResponse* 및 *SendRedirect*를 제공합니다. *TWebResponse* 객체의 모든 헤더 속성과 지정된 콘텐츠를 사용하는 응답을 보내려면 *SendResponse*를 호출합니다. 단지 웹 클라이언트를 다른 URI로 리디렉션해야 할 경우에는 *SendRedirect* 메소드가 더 효율적입니다.

응답을 보내는 이벤트 핸들러가 없을 경우 웹 애플리케이션 객체는 디스패처가 종료된 후 응답을 보냅니다. 그러나 응답을 처리했다는 사실을 나타내는 작업 항목이 없으면 웹 애플리케이션은 응답을 보내지 않고 웹 클라이언트와의 연결을 끊습니다.

## 응답 메시지 콘텐츠 생성

Kylix에서는 작업 항목이 HTTP 응답 메시지의 콘텐츠를 만들 수 있도록 도와줄 여러 가지 객체를 제공합니다. 이 객체들을 사용하면 파일에 저장되거나 웹 클라이언트에게 직접 돌려 보내지는 HTML 명령의 문자열을 생성할 수 있습니다. *TCustomContentProducer* 또는 그 자손 중 하나에서 콘텐츠 프로듀서를 파생시켜 자신만의 고유한 콘텐츠 프로듀서를 작성할 수 있습니다.

*TCustomContentProducer*는 MIME 형식을 만들기 위한 일반 인터페이스를 HTTP 응답 메시지로 제공하며, 자손으로는 페이지 프로듀서와 테이블 프로듀서를 포함합니다.

- 페이지 프로듀서는 사용자 지정 HTML 코드로 교체할 특수 태그를 HTML 문서에서 스캔합니다. 페이지 프로듀서에 대한 자세한 내용은 다음 단원에서 설명합니다.
- 테이블 프로듀서는 데이터셋의 정보에 기초하여 HTML 명령을 만듭니다. 자세한 내용은 22-20 페이지의 "응답에서 데이터베이스 정보 사용"에 설명되어 있습니다.

## 페이지 프로듀서 컴포넌트 사용

페이지 프로듀서(*TPageProducer*와 그 자손)는 HTML 템플릿을 가져온 다음 특수 HTML-transparent 태그를 사용자 지정 HTML 코드로 교체하여 템플릿을 변환합니다. 사용자는 HTTP 요청 메시지에 대한 응답을 생성해야 할 경우 페이지 프로듀서가 채우는 표준 응답 템플릿 집합을 저장할 수 있습니다. 또한 페이지 프로듀서를 서로 연결하면 HTML-transparent 태그를 계속 구체화하여 HTML 문서를 반복적으로 작성할 수 있습니다.

## HTML 템플릿

HTML 템플릿은 연속적인 HTML 명령과 HTML-transparent 태그들입니다. HTML-transparent 태그는 다음과 같은 형태를 갖습니다.

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

꺾쇠 괄호(< 및 >)는 태그의 전체 유효 범위(scope)를 정의합니다. 열린 꺾쇠 괄호(<) 바로 뒤에는 공백없이 파운드 기호(#)가 오며 꺾쇠 괄호와 구분됩니다. 파운드 기호는 페이지 프로듀서에 대한 문자열을 HTML-transparent 태그로 식별합니다. 파운드 기호 바로 뒤에는 공백없이 태그 이름이 오며 파운드 기호와 구분됩니다. 태그 이름은 유효한 식별자가 될 수 있고 태그가 나타내는 변환 유형을 식별합니다.

태그 이름 다음에 오는 HTML-transparent 태그는 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 오지 않습니다. 각 매개변수들은 공백으로 구분합니다.

꺾쇠 괄호(< 및 >)는 #TagName 구조를 인식하지 않는 HTML 브라우저가 태그를 알 수 있게 해 줍니다.

자신의 고유한 HTML 투명 태그를 만들어 페이지 프로듀서가 처리하는 모든 종류의 정보를 나타낼 수도 있지만, *TTag* 데이터 형식 값에 연결된 여러 가지 태그 이름들이 이미 정의되어 있습니다. 이와 같이 이미 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                                                                     |
|-----------------|-------------------|---------------------------------------------------------------------------|
| <i>Link</i>     | <i>tgLink</i>     | 하이퍼텍스트 링크. 결과는 <A> 태그로 시작하여 </A> 태그로 끝나는 HTML입니다.                         |
| <i>Image</i>    | <i>tgImage</i>    | 그래픽 이미지. 결과는 HTML <IMG> 태그입니다.                                            |
| <i>Table</i>    | <i>tgTable</i>    | HTML 테이블. 결과는 <TABLE> 태그로 시작하여 </TABLE> 태그로 끝나는 HTML입니다.                  |
| <i>ImageMap</i> | <i>tgImageMap</i> | 연결된 핫 존(hot zone)이 있는 그래픽 이미지. 결과는 <MAP> 태그로 시작하여 </MAP> 태그로 끝나는 HTML입니다. |

다른 모든 태그 이름은 *tgCustom*에 연결됩니다. 페이지 프로듀서는 이미 정의된 태그 이름의 기본 제공 처리를 제공하지 않습니다. 이미 정의된 태그 이름은 애플리케이션이 변환 프로세스를 더 일반적인 작업으로 구성할 수 있도록 돕기 위해 제공되는 것입니다.

**참고** 이미 정의된 태그 이름은 대소문자를 구분하지 않습니다.

## HTML 템플릿 지정

페이지 프로듀서는 HTML 템플릿을 지정하는 방법에 대한 많은 선택을 제공합니다. *HTMLFile* 속성을 HTML 템플릿을 포함하는 파일의 이름으로 설정할 수 있고, *HTMLDoc* 속성을 HTML 템플릿을 포함하는 *TStrings* 객체로 설정할 수 있습니다. *HTMLFile* 속성 또는 *HTMLDoc* 속성을 사용하여 템플릿을 지정하는 경우, *Content* 메소드를 호출하여 변환된 HTML 명령을 생성할 수 있습니다.

이외에도 *ContentFromString* 메소드를 호출하여 매개변수로 전달되는 단일 문자열인 HTML 템플릿을 직접 변환할 수 있습니다. 또한 *ContentFromStream* 메소드를 호출하여 스트림에서 HTML 템플릿을 읽을 수 있습니다. 따라서 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-transparent 태그 변환

페이지 프로듀서는 사용자가 *Content* 메소드 중 하나를 호출할 때 HTML 템플릿을 변환합니다. *Content* 메소드는 HTML-transparent 태그를 만나면 *OnHTMLTag* 이벤트를 트리거합니다. 확인되는 태그 유형을 결정하고 이를 사용자 지정 콘텐츠로 바꾸기 위해 이벤트 핸들러를 작성해야 합니다.

페이지 프로듀서에 대해 *OnHTMLTag* 이벤트 핸들러를 만들지 않은 경우 HTML-transparent 태그는 빈 문자열로 바뀝니다.

## 작업 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 HTML로 변환합니다.

```
procedure WebModule1.MyActionEventHandler(Sender:TObject; Request:TWebRequest;
  Response:TWebResponse; var Handled:Boolean);
begin
  PageProducer1.HTMLFile := 'Greeting.html';
  Response.Content := PageProducer1.Content;
end;
```

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* 이벤트는 다음과 같은 방법으로 실행 도중에 HTML에 있는 사용자 지정 태그(<#UserName>)를 바꿉니다.

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
  const TagString: string; TagParams: TStrings; var ReplaceText:string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

요청 메시지의 콘텐츠가 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 예제는 *OnAction* 이벤트 핸들러를 사용하여 콘텐츠 프로듀서를 호출하고 응답 메시지의 콘텐츠를 할당합니다. 디자인 타임에 페이지 프로듀서의 *HTMLFile* 속성을 지정하는 경우 *OnAction* 이벤트 핸들러를 작성할 필요가 없습니다. 이 경우에는 간단히

*PageProducer1*을 작업 항목의 *Producer* 속성으로 지정하면 위의 *OnAction* 이벤트 핸들러와 동일한 효과를 거둘 수 있습니다.

## 페이지 프로듀서 간의 연결

*OnHTML Tag* 이벤트 핸들러에서 가져온 대체 텍스트는 HTTP 응답 메시지에서 사용할 최종적인 HTML가 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 별도의 작업 항목에 연결하는 것이며, 이 때 모든 작업 항목은 동일한 *PathInfo*와 *MethodType*을 가집니다. 첫 번째 작업 항목은 콘텐츠 프로듀서에 가져온 웹 응답 메시지를 설정하지만 이 작업 항목의 *OnAction* 이벤트 핸들러는 메시지를 처리되지 않은 것으로 간주합니다. 다음 작업 항목은 연결된 프로듀서의 *ContentFromString* 메소드를 사용하여 웹 응답 메시지 콘텐츠를 조작하는 등의 작업을 수행합니다. 첫 번째 이후의 작업 항목은 다음과 같이 *OnAction* 이벤트 핸들러를 사용합니다.

```
procedure WebModule1.Action2Action(Sender:TObject; Request:TWebRequest;
  Response:TWebResponse; var Handled:Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

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.



이 달력의 일반 폼은 템플릿 파일에 저장되며 다음과 같이 나타납니다.

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

첫 번째 페이지 프로듀서의 *OnHTML Tag* 이벤트 핸들러는 요청 메시지에서 월과 연도를 조회합니다. 조회한 정보와 템플릿 파일을 사용하여 이벤트 핸들러가 다음과 같이 수행됩니다.

- <#MonthlyImage>를 <#Image Month=August Year=2001>로 바꿉니다.
- <#TitleLine>을 <#Calendar Month=July Year=2001 Size=Small> August 2001 <#Calendar Month=September Year=2001 Size=Small>로 바꿉니다.
- <#MainBody>를 <#Calendar Month=August Year=2001 Size=Large>로 바꿉니다.

다음 페이지 프로듀서의 *OnHTMLTag* 이벤트 핸들러는 첫 번째 페이지 프로듀서가 만든 콘텐츠를 사용하고 <#Image Month=August Year=2001> 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 정보를 포함할 수 있습니다. 인터넷 팔레트 페이지에 있는 특수한 콘텐츠 프로듀서는 HTML을 생성하여 데이터베이스의 레코드를 HTML 테이블에 나타낼 수 있습니다.

### 웹 모듈에 세션 추가

---

콘솔 CGI 애플리케이션 및 Apache DSO 애플리케이션은 HTTP 요청 메시지에 응답하여 시작됩니다. 이러한 유형의 애플리케이션에서 데이터베이스 작업을 수행할 경우 각각의 요청 메시지가 고유한 애플리케이션 인스턴스를 갖기 때문에 사용자는 기본 세션을 사용하여 데이터베이스 연결을 관리할 수 있습니다.

### HTML로 데이터베이스 정보 표시

---

인터넷 팔레트 페이지의 특수한 콘텐츠 프로듀서 컴포넌트는 데이터셋의 레코드에 기초하여 HTML 명령을 제공합니다. 다음과 같이 두 가지 유형의 data-aware 콘텐츠 프로듀서가 있습니다.

- 데이터셋 필드 서식을 HTML 문서의 텍스트로 설정하는 데이터셋 페이지 프로듀서
- 데이터셋 레코드 서식을 HTML 테이블로 설정하는 테이블 프로듀서

### 데이터셋 페이지 프로듀서 사용

데이터셋 페이지 프로듀서는 다른 페이지 프로듀서 컴포넌트와 같은 방식으로 사용됩니다. 즉, HTML-transparent 태그를 포함하는 템플릿을 최종 HTML 형태로 변환합니다. 그러나 데이터셋 페이지 프로듀서는 데이터셋에 있는 필드 이름과 일치하는 태그 이름을 가진 태그를 해당 필드의 현재 값으로 변환하는 특수 기능을 포함합니다. 페이지 프로듀서의 일반적인 사용에 관한 자세한 내용은 22-16 페이지의 "페이지 프로듀서 컴포넌트 사용"을 참조하십시오.

데이터셋 페이지 프로듀서를 사용하려면 *TDataSetPageProducer* 컴포넌트를 웹 모듈에 추가하고 *DataSet* 속성을 필드 값이 HTML 콘텐츠로 표시되어야 하는 데이터셋으로 설정합니다. 데이터셋 페이지 프로듀서의 출력을 설명하는 HTML 템플릿을 만듭니다.



그리고 표시하려는 모든 필드 값에 대해 다음과 같은 형태의 태그를 HTML 템플릿에 포함시킵니다.

```
<#FieldName>
```

여기서 *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 메시지가 제공한 매개변수를 설정한 후 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 사용자 지정할 수 있습니다. 디자인 타임에 테이블 프로듀서의 속성을 설정하려면 테이블 프로듀서 컴포넌트를 더블 클릭하여 Response Editor 대화 상자를 표시합니다.

## 테이블 속성 지정

테이블 프로듀서는 *THTMLTableAttributes* 객체를 사용하여 데이터셋의 레코드를 표시하는 HTML 테이블의 시각적 모양을 설명합니다. *THTMLTableAttributes* 객체에는 HTML 문서에서의 테이블 너비와 간격, 배경 색, 테두리 두께, 셀 안쪽 여백, 셀 간격 등에 대한 속성이 포함되어 있습니다. 이러한 모든 속성은 테이블 프로듀서가 만든 HTML <TABLE> 태그에서 옵션으로 전환됩니다.

디자인 타임에 Object Inspector로 이러한 속성을 지정합니다. Object Inspector에서 테이블 프로듀서 객체를 선택하고 *TableAttributes* 속성을 확장하여 *THTMLTableAttributes* 객체의 표시 속성에 액세스합니다.

또한 이러한 속성을 런타임에 프로그램에서 지정할 수도 있습니다.

## 행 속성 지정

데이터를 표시하는 테이블의 행에서 테이블 속성처럼 셀의 배경 색 및 정렬을 지정할 수 있습니다. 이 때 사용되는 *RowAttributes* 속성은 *THTMLTableRowAttributes* 객체입니다.

디자인 타임에 Object Inspector에서 *RowAttributes* 속성을 확장하여 이러한 속성을 지정합니다. 또한 이러한 속성을 런타임에 프로그램에서 지정할 수도 있습니다.

이 밖에도 *MaxRows* 속성을 설정하면 HTML 테이블에 표시되는 행 수를 조정할 수 있습니다.

## 열 지정

디자인 타임에 테이블의 데이터셋을 알고 있다면 Columns 편집기를 사용하여 열의 필드 바인딩을 사용자 지정하고 속성을 표시할 수 있습니다. 이렇게 하려면 테이블 프로듀서 컴포넌트를 선택하고 마우스 오른쪽 버튼을 클릭한 다음 컨텍스트 메뉴에서 Columns 편집기를 선택합니다. 이 편집기는 테이블 열을 추가, 삭제 또는 재정렬할 수 있게 해 줍니다. Columns 편집기에서 열을 선택한 후 Object Inspector에서 개별 열의 필드 바인딩을 설정하고 속성을 표시할 수 있습니다.

HTTP 요청 메시지에서 데이터셋의 이름을 가져오는 중이면 디자인 타임에 Columns 편집기에서 필드를 바인딩할 수 없습니다. 그러나 해당 *THTMLTableColumn* 객체를 설정하고 Columns 속성의 메소드를 통해 테이블에 열을 추가하여 런타임에 프로그램에서 열을 사용자 지정할 수 있습니다. Columns 속성을 설정하지 않을 경우 테이블 프로듀서는 데이터셋 필드와 일치하는 열의 기본 집합을 만들고 특별한 표시 특성을 지정하지 않습니다.

## HTML 문서에 테이블 포함시키기

테이블 프로듀서의 *Header* 및 *Footer* 속성을 사용하면 더 큰 문서의 데이터셋을 나타내는 HTML 테이블을 포함할 수 있습니다. *Header*는 테이블 앞에 오는 모든 내용을 지정하기 위하여, *Footer*는 테이블 뒤에 오는 모든 내용을 지정하기 위하여 사용합니다.

페이지 프로듀서 같은 다른 콘텐츠 프로듀서를 사용하여 *Header* 및 *Footer* 속성에 대한 값을 만들 수도 있습니다.

더 큰 문서의 테이블을 포함할 경우 테이블에 캡션을 추가할 수 있습니다. 테이블에 캡션을 제공하려면 *Caption* 및 *CaptionAlignment* 속성을 사용합니다.

## 데이터셋 테이블 프로듀서 설정

*TDataSetTableProducer*는 데이터셋의 HTML 테이블을 만드는 테이블 프로듀서입니다. 표시할 레코드를 포함하는 데이터셋을 지정하려면 *TDataSetTableProducer*의 *DataSet* 속성을 설정합니다. 기본 데이터베이스 애플리케이션에서의 대부분의 data-aware 객체와 마찬가지로 *DataSource* 속성은 설정하지 않습니다. 이 속성을 설정하지 않는 이유는 *TDataSetTableProducer*가 고유한 데이터 소스를 내부적으로 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 기초한다면 런타임에 *DataSet* 속성을 설정해야 합니다.

## 쿼리 테이블 프로듀서 설정

HTML 테이블을 만들어 HTTP 요청 메시지에서 매개변수를 가져온 쿼리 결과를 표시할 수 있습니다. 이렇게 하려면 쿼리 매개변수를 *TSQLQueryTableProducer* 컴포넌트의 *Query* 속성으로 사용하는 *TSQLQuery* 객체를 지정합니다.

요청 메시지가 GET 요청인 경우 HTTP 요청 메시지의 대상으로 제공된 URL의 *Query* 필드에서 쿼리의 매개변수를 가져옵니다. 요청 메시지가 POST 요청인 경우에는 요청 메시지의 콘텐츠에서 쿼리 매개변수를 가져옵니다.

*TSQLQueryTableProducer*의 *Content* 메소드는 호출 시 요청 객체에서 찾은 매개변수를 사용하여 쿼리를 실행한 다음 HTML 테이블 서식을 설정하여 결과 데이터셋의 레코드를 표시합니다.

다른 테이블 프로듀서와 마찬가지로 사용자는 HTML 테이블의 표시 속성이나 열 바인딩을 사용자 지정하거나 더 큰 HTML 문서에 테이블을 포함할 수 있습니다.

## 서버 애플리케이션 디버깅

웹 서버 애플리케이션은 웹 서버에서 온 메시지에 응답하여 실행되므로 이 애플리케이션에 대한 디버깅은 몇 가지 특수한 문제를 야기합니다. 또한 웹 서버가 루프를 벗어나게 되고 애플리케이션이 예상 요청 메시지를 찾지 않기 때문에 간단하게 IDE에서 애플리케이션을 시작할 수 없습니다. 웹 서버 애플리케이션을 디버깅하는 방법은 해당 애플리케이션의 유형에 따라 달라집니다.

### CGI 애플리케이션 디버깅

웹 서버가 애플리케이션 자체를 시작해야 하기 때문에 CGI 애플리케이션 디버깅은 더 어렵습니다.

#### 공유 객체로 디버깅

CGI 애플리케이션을 다루는 또다른 방법은 우선 애플리케이션을 Apache DSO 애플리케이션으로 만들어 디버깅하는 것입니다. Apache DSO 애플리케이션이 제대로 작동하면 CGI 애플리케이션으로 변환하십시오. 다음 단계에 따라 애플리케이션을 변환할 수 있습니다.

- 1 웹 모듈을 마우스 오른쪽 버튼으로 클릭하고 Add To Repository를 선택합니다.
- 2 Add To Repository 대화 상자에서 웹 모듈에 제목, 텍스트 설명, 레포지토리 페이지(대체로 데이터 모듈), 작성자 이름 및 아이콘 등을 제공합니다.
- 3 OK를 선택하여 웹 모듈을 템플릿으로 저장합니다.
- 4 메인 메뉴에서 File|New를 선택한 다음 Web Server Application을 선택합니다. New Web Server Application 대화 상자에서 적절하게 CGI를 선택합니다.
- 5 자동으로 생성된 웹 모듈을 삭제합니다.
- 6 메인 메뉴에서 File|New를 선택한 다음 3단계에서 저장한 템플릿을 선택합니다. 이 템플릿은 2단계에서 지정한 페이지에 놓여집니다.

### Apache DSO 애플리케이션 디버깅

Apache DSO 애플리케이션을 디버깅하기 전에 먼저 컴파일해야 합니다.

## DSO 지원을 위한 Apache 애플리케이션 컴파일

최신 정보를 보려면 다음 Apache 웹 사이트에서 온라인 문서를 참조하십시오.

<http://httpd.apache.org/docs/dso.html>

다음과 같은 방법으로 애플리케이션을 컴파일합니다.

- 1 Apache 소스 코드를 다운로드합니다.

현재 사용할 수 있는 소스 코드는 다음과 같습니다.

[http://httpd.apache.org/dist/apache\\_1.3.14.tar.gz](http://httpd.apache.org/dist/apache_1.3.14.tar.gz)

- 2 다운로드한 소스 타르(tar)의 압축을 풉니다.
- 3 /usr/apache\_1.3.14와 같이 압축을 푼 디렉토리의 루트로 이동합니다.
- 4 config.status 파일이 존재하지 않으면 새로 만들고 다음 줄을 추가합니다.

```
#!/bin/sh
##
## config.status -- APACI auto-generated configuration restore script
##
## Use this shell script to re-run the APACI configure script for
## restoring your configuration. Additional parameters can be supplied.
##
```

```
CFLAGS="-g" \
CFLAGS_SHLIB="-g" \
LIBS="/usr/lib/libpthreads.so" \
./configure \
"--with-layout=Apache" \
"--enable-module=so" \
"--enable-rule=SHARED_CORE" \
"$@"
```

**참고** LIBS="/usr/lib/libpthreads.so" \ 줄을 config.status 파일에 추가해야 합니다.

- 5 명령 프롬프트(\$=prompt)에서 다음 줄을 입력합니다.

```
$ chmod 755 config.status
$ ./config.status
$ make
$ make install
```

이제 대상 디렉토리에 Apache가 설치되어 DSO를 지원하도록 구성됩니다.

## 애플리케이션 디버깅

실행 매개변수 설정으로 IDE에서 호스트 애플리케이션을 시작하여 Apache 애플리케이션을 디버깅할 수 있습니다.

Apache DSO 애플리케이션을 디버깅하려면 루트 사용자로써 Kylix를 실행하는 중이어야 합니다. DSO를 지원하도록 Apache를 먼저 컴파일했는지 libpthreads.so로 확인합니다.

다음과 같은 방법으로 Apache DSO 애플리케이션을 디버깅합니다.

- 1 다음과 같이 실행 매개변수를 설정하여 호스트 애플리케이션을 시작합니다.

- Run|Parameters를 선택합니다.
  - 호스트 애플리케이션을 /usr/local/apache/bin/httpd로 설정합니다.
  - 매개변수를 -X로 설정합니다.
- 2 출력 디렉토리를 설정합니다.
    - Project|Options|Directories를 선택합니다
    - 출력 디렉토리를 ../apache/libexec로 설정합니다.
  - 3 브레이크포인트를 설정합니다.
  - 4 프로젝트를 작성합니다.
  - 5 애플리케이션을 실행합니다.
  - 6 웹 브라우저를 열고 모듈을 호출할 URL을 요청합니다.



## 소켓 작업

이 장에서는 TCP/IP 및 관련 프로토콜을 사용하여 다른 시스템과 통신할 수 있는 애플리케이션을 만들 수 있게 해 주는 소켓 컴포넌트에 대하여 설명합니다. 소켓을 사용하면 실제적인 네트워킹 소프트웨어에 대해 걱정하지 않아도 다른 컴퓨터들과 연결해서 읽고 쓸 수 있습니다. 소켓은 TCP/IP 프로토콜에 기반하여 연결되지만 Xerox Network System (XNS), Digital의 DECnet 또는 Novell의 IPX/SPX 제품군 같은 관련 프로토콜에서 일반적으로 작동합니다.

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 서비스에만 사용됩니다. 서버 소켓을 사용하면 이러한 서비스 중 하나를 제공하는 애플리케이션을 해당 서비스를 사용하고자 하는 클라이언트 애플리케이션에 연결할 수 있습니다. 클라이언트 소켓을 사용하여 이러한 서비스 중 하나를 사용하는 애플리케이션은 해당 서비스를 제공하는 서버 애플리케이션에 연결할 수 있습니다.

### 서비스 구현

소켓은 네트워크 서버 또는 클라이언트 애플리케이션을 작성하는 데 필요한 서비스 중 하나를 제공합니다. HTTP 또는 FTP와 같은 대부분의 서비스에서는 타사의 서버를 쉽게 사용할 수 있습니다. 운영 체제와 함께 제공되는 경우도 있기 때문에 사용자가 직접 작성할 필요가 없습니다. 그러나 서비스가 구현되는 방법을 더 자세히 제어해야 하거나, 애플리케이션과 네트워크 통신 간의 보다 긴밀한 통합이 필요하거나, 또는 특정 서비스에 사용할 수 있는 서버가 없는 경우에는 사용자의 고유한 서버나 클라이언트 애플리케이션을 만들 수 있습니다. 예를 들어, 분산 데이터셋에서 작업하는 경우 다른 시스템의 데이터베이스와 통신하기 위해 계층을 작성할 수 있습니다.

## 서비스 프로토콜 이해

---

네트워크 서버 또는 클라이언트를 작성하기 전에 애플리케이션에서 제공하거나 사용 중인 서비스를 이해해야 합니다. 대부분의 서비스는 네트워크 애플리케이션이 지원해야 하는 표준 프로토콜을 가집니다. HTTP, FTP, finger 또는 time 같은 표준 서비스에 대해 네트워크 애플리케이션을 작성 중인 경우, 먼저 다른 시스템과 통신하는 데 사용되는 프로토콜을 이해해야 합니다. 이를 위해서는 사용자가 제공하거나 사용하려는 특정 서비스에 대한 설명서를 참조하는 것이 좋습니다.

다른 시스템과 통신하는 애플리케이션에 대해 새로운 서비스를 제공하려면 우선 해당 서비스의 서버와 클라이언트를 위한 통신 프로토콜을 디자인하는 것이 필요합니다. 어떤 메시지들이 보내지는가? 이러한 메시지들이 어떻게 통합되는가? 정보를 인코딩하는 방법은 무엇인가?

## 애플리케이션과 통신

종종 네트워크 서버 또는 클라이언트 애플리케이션은 네트워킹 소프트웨어와 서비스를 사용하는 애플리케이션 간의 계층(layer)을 제공합니다. 예를 들어, HTTP 서버는 콘텐츠를 제공하고 HTTP 요청 메시지에 응답하는 웹 서버 애플리케이션과 인터넷 사이에 자리합니다.

소켓은 네트워크 서버 또는 클라이언트 애플리케이션과 네트워킹 소프트웨어 간의 인터페이스를 제공합니다. 사용자는 자신의 애플리케이션과 이를 사용하는 다른 애플리케이션 간의 인터페이스를 제공해야 합니다.

## 서비스와 포트

---

일반적으로 대부분의 표준 서비스는 특정 포트 번호와 연결됩니다. 포트 번호에 대한 자세한 내용은 나중에 설명이 되며, 여기서는 포트 번호를 서비스에 대한 숫자 코드라고 간주합니다.

표준 서비스를 구현하려는 경우 *TTcpClient* 소켓 컴포넌트에서 서비스의 포트 번호를 조회할 수 있는 메소드를 제공합니다. 새로운 서비스를 제공하려는 경우 */etc/services* 파일에서 연결되는 포트 번호를 지정할 수 있습니다. 이 서비스 파일은 이름, 포트 번호 및 프로토콜 타입을 포함한 인터넷 네트워크 서비스를 나열하는 ASCII 파일입니다. 서비스 파일 설정에 대한 자세한 내용은 서비스 man 페이지를 참조하십시오.



## 소켓 연결의 타입

---

소켓의 연결이 시작되는 방법과 로컬 소켓이 연결되는 대상에 따라 소켓 연결을 다음과 같이 세 가지 기본 타입으로 구분할 수 있습니다.

- 클라이언트 연결
- 리스닝 연결
- 서버 연결

일단 클라이언트 소켓과의 연결이 완료되면 서버 연결은 클라이언트 연결과 구분될 수 없습니다. 두 끝점은 그 기능이 동일하고 같은 타입의 이벤트를 받습니다. 리스닝 연결은 끝점을 하나만 갖기 때문에 기초부터 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 소켓을 기술해야 합니다. 그 다음, 클라이언트 소켓은 서버 소켓을 조회하고, 서버를 찾을 때 연결을 요청합니다. 서버 소켓은 연결을 즉시 완료하지 않을 수도 있습니다. 서버 소켓은 클라이언트 요청의 대기열을 유지 관리하다가 시간이 되면 연결을 완료합니다. 클라이언트 연결을 승인하면 서버 소켓은 연결하고자 하는 서버 소켓에 대한 완전한 정보를 클라이언트 소켓으로 보내고 클라이언트가 연결을 완료합니다.

### 리스닝(listening) 연결

---

서버 소켓은 클라이언트를 찾지 않습니다. 대신, 서버 소켓은 클라이언트 요청을 리스닝 하는 수동 "반 연결"을 구성합니다. 서버 소켓은 대기열을 리스닝 연결에 연결하며, 대기열은 클라이언트 연결 요청이 들어올 때 그것을 기록합니다. 클라이언트 연결 요청을 승인하면 서버 소켓은 리스닝 연결이 계속 열린 상태에서 다른 클라이언트 요청을 받을 수 있도록 새 소켓을 구성하여 클라이언트에 연결합니다.

### 서버 연결

---

서버 연결은 리스닝 소켓이 클라이언트 요청을 승인할 때 서버 소켓에 의해 구성됩니다. 서버가 연결을 승인하면 클라이언트에 대한 연결을 완료하는 서버 소켓의 정보가 클라이언트에게 보내집니다. 클라이언트 소켓이 이 정보를 받고 연결을 완료하면 연결이 설정됩니다.

## 소켓 설명

---

네트워크 애플리케이션은 소켓을 사용하여 네트워크상에서 다른 시스템과 통신할 수 있습니다. 각 소켓은 네트워크 연결의 끝점으로 보여질 수 있으며, 다음 사항을 지정하는 주소를 가집니다.

- 실행되는 시스템
- 인터페이스의 타입들
- 연결을 위해 사용하는 포트

소켓 연결에 대한 전체 설명에는 연결의 두 끝점에 있는 소켓 주소가 포함됩니다. IP 주소나 호스트와 포트 번호로 소켓 양 끝점의 주소를 설명할 수 있습니다.

소켓 연결을 하기 전에 우선 끝점을 구성하는 소켓을 완전히 설명해야 합니다. 일부 정보는 애플리케이션이 실행 중인 시스템에서 얻을 수 있습니다. 예를 들어, 클라이언트 소켓의 로컬 IP 주소는 운영 체제에서 얻을 수 있으므로 설명할 필요가 없습니다.

사용자가 제공해야 할 정보는 사용 중인 소켓 타입에 따라 달라집니다. 클라이언트 소켓은 연결하고자 하는 서버 정보를 제공해야 합니다. 리스닝 소켓은 제공하는 서비스를 나타내는 포트 정보를 제공해야 합니다.

## 호스트 설명

---

호스트는 소켓이 포함된 애플리케이션을 실행 중인 시스템입니다. 다음과 같이 표준 인터넷 도트 표시법으로 된 4개의 숫자(바이트) 값의 문자열인 IP 주소를 제공하여 소켓의 호스트 정보를 제공할 수 있습니다.

123.197.1.2

하나의 시스템이 하나 이상의 IP 주소를 지원할 수도 있습니다.

IP 주소는 종종 기억하기가 어려워서 잘못 입력하는 경우가 있습니다. 이 경우 대안은 호스트 이름을 사용하는 것입니다. 호스트 이름은 URL (Uniform Resource Locator) 에서 자주 볼 수 있는 IP 주소의 별칭입니다. 이것은 다음과 같은 도메인 이름 및 서비스를 포함한 문자열입니다.

<http://www.wSite.Com>

대부분의 인트라넷은 인터넷에 있는 시스템의 IP 주소에 대한 호스트 이름을 제공합니다. Linux 시스템에서 호스트 이름을 사용할 수 없는 경우 /etc/hosts 파일에 이름을 입력하여 로컬 IP 주소에 대한 호스트 이름을 만들 수 있습니다. /etc/hosts 파일에 대한 자세한 내용은 Linux 설명서를 참조하십시오.

서버 소켓은 호스트를 지정할 필요가 없습니다. 로컬 IP 주소는 시스템에서 얻을 수 있습니다. 로컬 시스템이 하나 이상의 IP 주소를 지원할 경우 서버 소켓은 모든 IP 주소에서 동시에 클라이언트 요청을 리스닝합니다. 서버 소켓이 연결을 승인하면 클라이언트 소켓은 원격 IP 주소를 제공합니다.

클라이언트 소켓은 호스트 이름 또는 IP 주소를 제공하여 원격 호스트를 지정해야 합니다.

## 호스트 이름 또는 IP 주소 선택

대부분의 애플리케이션은 호스트 이름을 사용하여 시스템을 지정합니다. 호스트 이름은 기억하기가 쉽고 입력 오류를 쉽게 확인할 수 있습니다. 또한 서버는 특정 호스트 이름과 연결된 시스템 또는 IP 주소를 변경할 수 있습니다. 호스트 이름을 사용하면 클라이언트 소켓은 호스트 이름이 나타내는 추상적 사이트가 새 IP 주소로 이동한 경우에도 해당 사이트를 찾을 수 있습니다.

호스트 이름이 없는 경우 클라이언트 소켓은 IP 주소를 사용하여 서버 시스템을 지정해야 합니다. IP 주소를 제공하면 서버 시스템을 더 빠르게 지정할 수 있습니다. 호스트 이름이 제공되는 경우에는 소켓이 호스트 이름과 연결된 IP 주소를 검색한 후에만 서버 시스템을 찾을 수 있습니다.

## 포트 사용

IP 주소가 소켓 연결의 다른 끝에 있는 시스템을 찾는 데 필요한 충분한 정보를 제공하지만, 해당 시스템의 포트 번호가 필요합니다. 포트 번호가 없으면 시스템은 한 번에 하나의 연결만 구성할 수 있습니다. 포트 번호는 단일 시스템이 각 연결에 별도의 포트를 제공하여 여러 연결을 동시에 호스트할 수 있도록 하는 고유 식별자입니다.

앞에서 이미 포트 번호가 네트워크 애플리케이션에 의해 구현된 서비스에 대한 숫자 코드를 설명한 바 있습니다. 이것은 고정 포트 번호에서 리스닝 서버를 사용하여 클라이언트 소켓이 리스닝 서버를 찾을 수 있도록 하는 규칙입니다. 서버 소켓은 자신이 제공하는 서비스와 연결된 포트 번호를 리스닝합니다. 서버 소켓이 클라이언트 소켓에 대한 연결을 승인하면 다른 임의의 포트 번호를 사용하는 별도의 소켓 연결을 만듭니다. 이와 같은 방법으로 리스닝 연결은 서비스와 연결된 포트 번호를 계속 리스닝할 수 있습니다.

클라이언트 소켓은 다른 소켓에 의해 발견될 필요가 없기 때문에 임의의 로컬 포트 번호를 사용합니다. 클라이언트 소켓은 서버 애플리케이션을 찾을 수 있도록 연결하고자 하는 서버 소켓의 포트 번호를 지정합니다. 종종 이 포트 번호는 원하는 서비스 이름을 지정하여 간접적으로 지정됩니다.

## 소켓 컴포넌트 사용

Internet 팔레트 페이지에는 소켓 컴포넌트(클라이언트 소켓과 서버 소켓)가 포함되어 있습니다. 이러한 컴포넌트를 사용하면 네트워크 애플리케이션은 다른 시스템에 대한 연결을 구성할 수 있고 사용자는 이 연결을 통해 정보를 읽고 쓸 수 있습니다. 각 소켓 컴포넌트에는 실제 소켓 연결의 끝점을 나타내는 소켓 객체가 연결됩니다. 소켓 컴포넌트는 소켓 객체를 사용하여 운영 체제 API 호출을 캡슐화하므로 애플리케이션은 연결을 구성하거나 소켓 메시지를 관리하는 세부 사항과 관련된 필요가 없습니다.

운영 체제 API 호출을 사용하거나 소켓 컴포넌트가 사용자 대신 연결한 세부 내용을 사용자 지정하고자 할 때, 소켓 객체의 속성, 이벤트 및 메소드를 사용할 수 있습니다.

## 클라이언트 소켓 사용

클라이언트 소켓 컴포넌트(*TTcpClient*)를 폼이나 데이터 모듈에 추가하여 애플리케이션을 TCP/IP 클라이언트로 전환합니다. 클라이언트 소켓을 사용하면 연결하고자 하는 서버 소켓과 서버가 제공하게 할 서비스를 지정할 수 있습니다. 원하는 연결 정보를 제공했다면 클라이언트 소켓 컴포넌트를 사용하여 서버에 대한 연결을 완료할 수 있습니다.

### 대상 서버 지정

클라이언트 소켓 컴포넌트에는 사용자가 연결하고자 하는 서버 시스템과 포트를 지정할 수 있게 해 주는 여러 속성들이 있습니다. *RemoteHost* 속성을 사용하여 호스트 이름 또는 IP 주소로 원격 호스트 서버를 지정합니다.

호스트 이외에도 추가하여 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 서버 정보를 제공했다면 *Open* 메소드를 호출하여 런타임에 연결을 구성할 수 있습니다. 또는 애플리케이션이 시작될 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* 및 *LocalPort* 속성을 사용하면 클라이언트 및 서버 소켓이 연결의 끝점을 구성하기 위해 사용한 주소와 포트 번호를 정할 수 있습니다. *Handle* 속성을 사용하면 소켓 호출을 생성할 때 사용하는 소켓 연결에 대한 핸들을 얻을 수 있습니다.

### 연결 끊기

소켓 연결을 통한 서버 애플리케이션과의 통신이 끝나면 *Close* 메소드를 호출하여 연결을 종료할 수 있습니다. 서버 끝에서 연결이 끊길 수도 있습니다. 이 경우에 사용자는 *OnDisconnect* 이벤트를 통해 공지를 받게 됩니다.

## 서버 소켓 사용

서버 소켓 컴포넌트(*TTcpServer*)를 폼이나 데이터 모듈에 추가하여 애플리케이션을 TCP/IP 서버로 전환합니다. 서버 소켓을 사용하면 제공 중인 서비스를 지정하거나 클라이언트 요청을 리스닝하는 데 사용할 포트를 지정할 수 있습니다. 서버 소켓 컴포넌트를 사용하면 클라이언트 연결 요청을 리스닝하고 승인할 수 있습니다.

서버 소켓 컴포넌트는 클라이언트로부터 연결 요청을 받으면 *TTcpClient* 컴포넌트를 만듭니다. *TTcpClient* 컴포넌트는 연결을 위한 끝점으로 사용됩니다.

## 포트 지정

서버 소켓이 클라이언트 요청을 리스닝할 수 있으려면 서버가 리스닝할 포트를 지정해야 합니다. 이 포트를 지정하려면 *LocalPort* 속성을 사용합니다. 서버 애플리케이션이 일반적으로 특정 포트 번호와 연결된 표준 서비스를 제공하는 경우에 *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 리스닝

일단 서버 소켓 컴포넌트의 포트 번호를 설정했다면 *Open* 메소드를 호출하여 런타임 시 리스닝 연결을 구성할 수 있습니다. 또는 애플리케이션이 시작될 때 자동으로 리스닝 연결을 구성하도록 하려면 디자인 타임에 Object Inspector를 사용하여 *Active* 속성을 *True*로 설정합니다.

## 클라이언트에 연결

*AutoAccept* 속성이 *True*로 설정된 경우 리스닝 서버 소켓 컴포넌트는 클라이언트 연결 요청을 받았을 때 자동으로 승인합니다. 사용자는 이러한 자동 승인이 *OnAccept* 이벤트에서 발생할 때마다 공지를 받습니다.

## 서버 연결 끊기

리스닝 연결을 종료하려면 *Close* 메소드를 호출하거나 *Active* 속성을 *False*로 설정합니다. 그러면 클라이언트 애플리케이션에 대한 승인된 모든 연결이 종료되고 아직 승인되지 않고 보류 중인 연결이 취소된 다음 리스닝 연결이 종료됩니다. 따라서 서버 소켓 컴포넌트는 더 이상 새 연결을 승인하지 않습니다.

클라이언트가 서버 소켓에 대한 개별 연결을 종료하면 *OnDisconnect* 이벤트는 사용자에게 이러한 사실을 알립니다.

## 소켓 이벤트에 응답

소켓을 사용하는 애플리케이션을 작성할 때 프로그램의 어디에서든지 소켓을 읽거나 쓸 수 있습니다. 소켓이 열린 후 프로그램에서 *SendBuf* 메소드를 사용하여 소켓에 쓸 수 있습니다. 소켓에서 읽기 또는 쓰기가 수행될 때마다 *OnSend* 및 *RecvBuf* 이벤트가 트리거됩니다. 필터링을 위해 이러한 이벤트를 사용할 수 있습니다. 사용자가 읽기 또는 쓰기 작업을 할 때마다 읽기 또는 쓰기 이벤트가 트리거됩니다.

연결로부터 오류 메시지를 받으면 클라이언트 소켓과 서버 소켓은 모두 오류 이벤트를 생성합니다.

소켓 컴포넌트는 또한 연결을 열고 완료하는 과정에서 두 개의 이벤트를 받습니다. 애플리케이션이 소켓 열기가 진행되는 방법에 영향을 끼쳐야 할 경우 *SendBuf* 및 *RecvBuf* 메소드를 사용하여 이러한 클라이언트 이벤트 또는 서버 이벤트에 응답해야 합니다.

## 오류 이벤트

---

연결로부터 오류 메시지를 받으면 클라이언트와 서버 소켓은 모두 *OnError* 이벤트를 생성합니다. *OnError* 이벤트 핸들러를 작성하면 이러한 오류 메시지에 응답할 수 있습니다. 이 이벤트 핸들러에는 다음에 관한 정보가 전달됩니다.

- 오류 공지를 받은 소켓 객체
- 오류 발생 시 소켓이 시도하던 작업
- 오류 메시지가 제공한 오류 코드

이벤트 핸들러에서 오류에 응답할 수 있고 오류 코드를 0으로 변경하여 소켓이 예외를 일으키는 것을 방지할 수 있습니다.

## 클라이언트 이벤트

---

클라이언트 소켓이 연결되면 다음 이벤트가 발생합니다.

- 이벤트 공지를 위해 소켓이 설정되고 초기화됩니다.
- 서버와 서버 소켓이 만들어진 후 *OnCreateHandle* 이벤트가 발생합니다. *Handle* 속성을 통해 사용할 수 있는 소켓 객체는 연결의 다른 끝을 구성할 서버나 클라이언트 소켓에 대한 정보를 제공할 수 있습니다. 이 때 연결에 사용되는 실제 포트를 처음으로 얻을 수 있으며, 실제 포트는 연결을 승인한 리스닝 소켓의 포트와 다를 수도 있습니다.
- 연결 요청이 서버에 의해 승인되고 클라이언트 소켓에 의해 완료됩니다.
- 연결이 설정되면 *OnConnect* 공지 이벤트가 발생합니다.

## 서버 이벤트

---

서버 소켓 컴포넌트는 두 가지 타입의 연결, 즉 리스닝 연결과 클라이언트 애플리케이션에 대한 연결을 구성합니다. 이러한 각 연결이 구성되는 동안 서버 소켓은 이벤트를 받습니다.

### 리스닝 시의 이벤트

리스닝 연결이 구성되기 바로 전에 *OnCreateHandle* 이벤트가 발생합니다. *OnConnect* 속성을 사용하면 소켓이 리스닝을 위해 열리기 전에 소켓을 변경할 수 있습니다. 예를 들어, 리스닝을 위해 서버가 사용하는 IP 주소를 제한하려면 *OnCreateHandle* 이벤트 핸들러에서 작업을 수행합니다.

## 클라이언트 연결 시의 이벤트

서버 소켓이 클라이언트 연결 요청을 승인하면 다음 이벤트가 발생합니다.

- *OnAccept* 이벤트가 발생하고, 새 *TTcpClient* 객체가 이벤트 핸들러에 전달됩니다. 이것은 *TTcpClient*의 속성을 사용하여 클라이언트에 대한 연결의 서버 끝점에 대한 정보를 얻을 수 있는 첫 번째 시점입니다.
- *BlockMode*가 *bmThreadBlocking*일 경우 *OnCreateServerSocketThread* 이벤트가 발생합니다. *TServerSocketThread*의 고유한 사용자 지정 자손을 제공하려는 경우 *OnCreateServerSocketThread* 이벤트 핸들러에서 사용자 지정 자손을 만들어서, *TServerSocketThread* 대신 사용할 수 있습니다.
- *BlockMode*가 *bmThreadBlocking*일 경우, 스레드(thread)가 실행을 시작할 때 *OnCreateServerSocketThread* 이벤트가 발생합니다. 스레드를 초기화하려는 경우 또는 스레드가 연결을 통해 읽기나 쓰기를 시작하기 전에 소켓 API를 호출하려는 경우 *OnCreateServerSocketThread* 이벤트 핸들러를 사용합니다.
- 클라이언트가 연결을 완료한 후 *OnAccept* 이벤트가 발생합니다. 비차단 서버의 경우에는, 이 때 소켓 연결을 통해 읽기 또는 쓰기를 시작할 수도 있습니다.

## 소켓 연결을 통한 읽기 및 쓰기

소켓 연결을 통해 정보를 읽거나 쓰기 위하여 다른 시스템에 대한 소켓 연결을 구성합니다. 읽거나 쓸 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 차단되지 않습니다. 이를 비차단 연결이라고 부릅니다. 또한 차단 연결을 구성할 수도 있는데 다음 코드 줄을 실행하기 전에 애플리케이션은 읽기 또는 쓰기가 완료되기를 기다립니다.

## 비차단 연결(Non-blocking Connections)

비차단 연결은 비동기적으로 읽고 쓰기 때문에 네트워크 애플리케이션에서 데이터 전송이 다른 코드의 실행을 차단하지 않습니다. 다음과 같은 방법으로 비차단 연결을 만듭니다.

- 클라이언트 소켓에서 *BlockMode* 속성을 *bmNonBlocking*으로 설정합니다.
- 서버 소켓에서 *BlockMode* 속성을 *bmNonBlocking*으로 설정합니다.

비차단 연결인 경우, 소켓은 읽기 및 쓰기 이벤트를 통해 연결의 다른 끝에 있는 소켓이 정보를 읽거나 쓰려고 하는 시기를 알 수 있습니다.

## 읽기 및 쓰기 이벤트

비차단 소켓은 연결을 통해 읽거나 써야 할 경우 읽기 및 쓰기 이벤트를 생성합니다. 클라이언트 소켓의 경우, *OnRead* 또는 *OnWrite* 이벤트 핸들러에서 이러한 공지에 응답할 수 있습니다. 서버 소켓의 경우, *OnClientRead* 또는 *OnClientWrite* 이벤트 핸들러에서 이러한 이벤트에 응답할 수 있습니다.

소켓 연결과 연결된 소켓 객체는 이벤트 핸들러를 읽거나 쓰기 위한 매개변수로 제공됩니다. 소켓 객체는 사용자가 연결을 통해 정보를 읽거나 쓸 수 있도록 하는 여러 메소드를 제공합니다.

소켓 연결에서 정보를 읽으려면 *ReceiveBuf* 또는 *ReceiveText* 메소드를 사용합니다. *ReceiveBuf* 메소드를 사용하기 전에, *ReceiveLength* 메소드를 사용하여 현재 로컬 버퍼에서 대기 중인 바이트 수를 정할 수 있습니다. 그러나 *ReceiveLength*가 호출될 때 유효한 값을 반환하지만 *ReceiveLength*를 호출한 다음과 *ReceiveBuf*를 호출하기 이전에 추가 데이터가 도착할 수 있다는 점에 유의해야 합니다. 이 때 추가 데이터가 도착하면 코드는 해당 *OnRead* 이벤트를 받지 않을 수도 있습니다. 따라서 대기 중인 추가 데이터가 없도록 *ReceiveLength*를 반복적으로 호출할 수도 있습니다.

소켓 연결을 통해 정보를 쓰려면 *SendBuf*, *SendStream* 또는 *SendText* 메소드를 사용합니다. 소켓을 통해 정보를 쓴 후 더 이상 소켓 연결이 필요하지 않을 경우, *SendStreamThenDrop* 메소드를 사용할 수 있습니다. *SendStreamThenDrop*은 스트림에서 읽을 수 있는 모든 정보를 쓴 후 소켓 연결을 끊습니다. *SendStream* 또는 *SendStreamThenDrop* 메소드를 사용할 경우 스트림 객체를 해제하지 않습니다. 연결이 끊어지면 소켓은 스트림을 자동으로 해제합니다.

**참고** *SendStreamThenDrop*은 리스닝 연결이 아니라 개별 클라이언트에 대한 서버 연결을 끊습니다.

## 차단 연결(Blocking Connection)

연결이 차단된 경우, 소켓은 소켓 연결로부터 공지를 수동적으로 기다리는 대신 연결을 통해 읽기 또는 쓰기를 시작해야 합니다. 읽기 및 쓰기가 발생하는 시기가 연결의 끝에서 결정될 경우 차단 소켓을 사용합니다.

클라이언트 소켓의 경우 *BlockMode* 속성을 *bmBlocking*으로 설정하여 차단 연결을 구성합니다. 클라이언트 애플리케이션이 수행하는 작업에 따라 사용자는 애플리케이션이 연결을 통한 읽기 또는 쓰기가 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 차단 연결을 구성합니다. 소켓이 연결을 통한 정보 읽기 또는 쓰기가 완료되기를 기다리는 동안 차단 연결이 다른 모든 코드의 실행을 보류하기 때문에, 서버 소켓 컴포넌트가 *BlockMode*가 *bmThreadBlocking*일 경우 항상 모든 클라이언트 연결에 대해 새 실행 스레드를 생성합니다. *BlockMode*가 *bmBlocking*이면 새 연결이 설정될 때까지 프로그램 실행이 차단됩니다.



# IV

## 사용자 지정 컴포넌트 생성

"사용자 지정 컴포넌트 생성"에 포함된 각 장에서는 Kylix에서 사용자 지정 컴포넌트를 디자인하고 구현하는 데 필요한 개념을 설명합니다.



## 컴포넌트 생성 개요

이 장에서는 Kylix 애플리케이션을 위한 컴포넌트 디자인 및 컴포넌트 작성 과정에 대한 개요를 제공합니다. 여기서는 사용자가 Kylix 및 표준 컴포넌트에 익숙하다고 가정합니다.

새 컴포넌트 설치에 대한 자세한 내용은 11-5 페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

### 크로스 플랫폼용 컴포넌트 라이브러리(CLX)

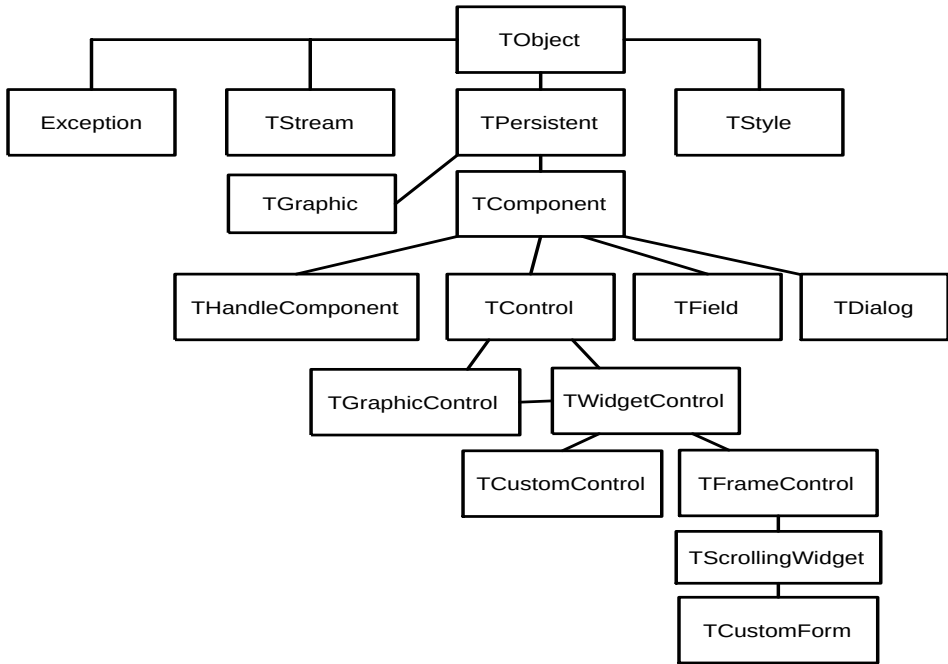
---

Kylix의 컴포넌트는 모두 크로스 플랫폼용 컴포넌트 라이브러리(CLX)라고 불리는 클래스 계층의 일부입니다. CLX 기본에 대한 자세한 내용은 3장장을 참조하십시오.

그림 24.1 장은 CLX를 구성하는 선택된 클래스의 관계를 보여 줍니다. 클래스 계층과 클래스 간의 상속 관계에 대한 자세한 내용을 보려면 25장 "컴포넌트 작성자를 위한 객체 지향 프로그래밍"을 참조하십시오.

*TComponent* 클래스는 CLX에 있는 모든 컴포넌트의 공유 조상입니다. *TComponent*는 Kylix에서 컴포넌트가 작동하는 데 필요한 최소한의 속성 및 이벤트를 제공합니다. 라이브러리의 다양한 분기는 보다 전문화된 다른 기능을 제공합니다.

그림 24.1 CLX 클래스 계층



컴포넌트를 만들 때는 계층 구조의 기존 클래스 타입 중 하나에서 새 클래스를 파생시켜 CLX에 추가합니다.

## 컴포넌트 및 클래스

컴포넌트가 클래스이기 때문에 컴포넌트 작성자는 애플리케이션 개발자와는 다른 수준에서 객체를 다룹니다. 새 컴포넌트를 생성하려면 새 클래스를 파생시켜야 합니다.

간단히 말해서, 컴포넌트를 생성하는 것과 애플리케이션에서 컴포넌트를 사용하는 것 사이에는 두 가지 중요한 차이가 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 애플리케이션 개발자가 컴포넌트를 사용하는 방법에 대해 고려해야 합니다.

## 컴포넌트 생성 방법

컴포넌트는 사용자가 디자인 타임에 조작하고자 하는 거의 모든 프로그램 요소가 될 수 있습니다. 컴포넌트 생성은 기존 클래스에서 새 클래스를 파생시키는 것을 의미합니다. 모든 기존 컴포넌트에서 새 컴포넌트를 파생시킬 수 있지만 일반적으로 다음과 같은 방법으로 컴포넌트를 생성합니다.

- 기존 컨트롤 수정
- 컨트롤 생성
- 그래픽 컨트롤 작성
- 컨트롤 하위 분류
- 년비주얼 컴포넌트 생성

표 24.1 장에는 각 작업에 대해 기본적으로 사용할 각기 다른 종류의 컴포넌트 및 클래스가 요약되어 있습니다.

**표 24.1** 컴포넌트 생성 기본 작업

| 수행할 작업           | 사용할 타입                                                                             |
|------------------|------------------------------------------------------------------------------------|
| 기존 컴포넌트 수정       | <i>TButton</i> 또는 <i>TPanel</i> 같은 모든 기존 컴포넌트나 <i>TCustomListBox</i> 같은 추상 컴포넌트 타입 |
| widget 기반 컨트롤 생성 | <i>TWidgetControl</i>                                                              |
| 그래픽 컨트롤 생성       | <i>TGraphicControl</i>                                                             |
| 컨트롤 하위 분류        | 모든 widget 기반 컨트롤                                                                   |
| 년비주얼 컴포넌트 생성     | <i>TComponent</i>                                                                  |

컴포넌트가 아니고 폼에서 조작할 수 없는 클래스를 파생시킬 수도 있습니다. Kylix 에는 *TFont*와 같은 여러 클래스가 많이 포함되어 있습니다.

### 기존 컨트롤 수정

컴포넌트를 만드는 가장 간단한 방법은 기존 컴포넌트를 사용자 지정하는 것입니다. Kylix와 함께 제공되는 모든 컴포넌트에서 새로운 컴포넌트가 파생될 수 있습니다. 목록 상자나 그리드(grid) 같은 일부 컨트롤은 기본 테마에서 여러 가지로 변형됩니다. 이 경우에 CLX는 사용자 지정 버전을 파생시키는 *TCustomGrid*와 같이 "custom"이라는 단어가 있는 추상 클래스를 포함합니다.

예를 들어, 표준 *TListBox* 클래스의 일부 속성을 갖지 않는 특수 목록 상자를 만들 수도 있습니다. 조상 클래스에서 상속된 속성은 제거하거나 숨길 수 없기 때문에 계층에서 *TListBox* 위에 있는 클래스에서 컴포넌트를 파생시켜야 합니다. 추상 *TWidgetControl* 클래스에서 작업을 시작하여 모든 목록 상자 기능을 다시 구성하도록 강제하는 대신 CLX는 목록 상자의 속성을 구현하지만 일부 속성을 published로 선언하지 않는 *TCustomListBox*를 제공합니다. 따라서 *TCustomListBox*와 같은 추상 클래스에서 컴포넌트를 파생시키는 경우, 컴포넌트에서 사용 가능하게 하려는 속성만 게시하고, 나머지 속성은 protected로 그대로 둡니다.

26장 "속성 생성"에서는 상속된 속성 게시에 대해 설명합니다. 31장 "기존 컴포넌트 수정" 및 33장 "그리드 사용자 지정"에는 기존 컨트롤을 수정하는 예제가 나와 있습니다.

## 컨트롤 생성

컨트롤은 런타임 시 나타나고 사용자가 상호 작용할 수 있는 객체입니다. 각 widget 기반 컨트롤에는 *Handle* 속성을 통해 액세스되고 원본으로 사용한 widget을 식별하는 핸들이 있습니다.

*TWidgetControl* 클래스는 모든 사용자 인터페이스 widget을 위한 기본 클래스입니다. 모든 widget 기반 컨트롤은 *TWidgetControl*의 자손입니다. 누름 버튼, 목록 상자, 편집 상자과 같은 대부분의 표준 컨트롤이 여기에 포함됩니다. 기존 컨트롤과 관련되지 않는 원래 컨트롤을 *TWidgetControl*에서 직접 파생시킬 수 있지만, Kylix는 이러한 목적으로 *TCustomControl* 컴포넌트를 제공합니다. *TCustomControl*은 복잡한 비주얼 이미지를 보다 쉽게 그릴 수 있도록 하는 전문화된 컨트롤입니다.

*TCustomControl*에 기초하여 사용자 지정 컨트롤을 생성하거나 CLX에 의해 아직 캡슐화되지 않은 widget과 같은 widget을 사용할 수 있습니다. 그리고 관련 객체나 *TWidgetControl* 자체를 조상으로 하여 컨트롤을 캡슐화할 수 있습니다.

33장 "그리드 사용자 지정"에는 컨트롤을 생성하는 예제가 나와 있습니다.

## 그래픽 컨트롤 작성

입력 포커스를 받을 필요가 없는 컨트롤인 경우, 사용자는 컨트롤을 그래픽 컨트롤로 만들 수 있습니다. 입력 포커스를 받지 않는 *TProgressBar*와 같은 컴포넌트는 그래픽 컨트롤입니다. 이러한 컨트롤이 포커스를 받을 수는 없지만 시스템 이벤트에 반응하도록 디자인할 수 있습니다.

Kylix는 *TGraphicControl* 컴포넌트를 통해 사용자 지정 컨트롤의 생성을 지원합니다. *TGraphicControl*은 *TControl*에서 파생된 추상 클래스입니다. *TControl*에서 컨트롤을 직접 파생시킬 수 있지만 그리기 캔버스를 제공하는 *TGraphicControl*에서 시작하는 것이 좋습니다.

32장 "그래픽 컴포넌트 생성"에는 그래픽 컨트롤을 생성하는 예제가 나와 있습니다.

## 컨트롤 하위 분류

새 widget을 정의하면 사용자 지정 컨트롤이 생성됩니다. widget에는 같은 종류의 컨트롤의 인스턴스 간에 공유된 정보를 포함하므로 *subclassing*이라고 하는 기존 클래스를 기준으로 새 widget을 만들 수 있습니다. 그런 다음, 표준 컨트롤의 경우처럼 사용자의 컨트롤을 공유 객체 파일에 두고 인터페이스를 제공합니다.

Kylix를 사용하면, 기존 widget 클래스의 주위에 컴포넌트 "래퍼(wrapper)"를 만들 수 있습니다. 따라서 Kylix 애플리케이션에서 사용하고자 하는 사용자 지정 컨트롤의 라이브러리를 이미 갖고 있는 경우, 사용자의 컨트롤처럼 작동하는 Kylix 컴포넌트를 생성할 수 있고, 다른 컴포넌트의 경우와 마찬가지로 Kylix 컴포넌트에서 새 컨트롤을 파생시킬 수 있습니다.

컨트롤을 하위 분류하는 데 사용되는 기술에 대한 예제를 보려면 *TEdit*와 같은 표준 컨트롤을 나타내는 *QStdCtrls* 유닛의 컴포넌트를 참조하십시오.

## 년비주얼 컴포넌트 생성

년비주얼 컴포넌트는 데이터베이스 (*TSQLConnection*)와 시스템 시계 (*TTimer*) 같은 요소를 위한 인터페이스로 사용되거나 대화 상자 (*TDialog*)와 그 자손을 위한 위치 표시자로서 사용됩니다. 사용자가 작성하는 대부분의 컴포넌트는 비주얼 컨트롤이 될 가능성이 많습니다. 년비주얼 컴포넌트는 모든 컴포넌트의 추상 기본 클래스인 *TComponent*에서 직접 파생될 수 있습니다. *THandleComponent*는 메뉴와 같이 원본으로 사용한 widget에 대한 핸들이 필요한 컴포넌트의 기본 클래스입니다.

## 컴포넌트 작성 시 고려 사항

Kylix 환경에서 컴포넌트를 안정적인 부분으로 만들려면 디자인에 대한 특정 규칙을 따라야 합니다. 이 단원에서는 다음과 같은 항목을 다룹니다.

- 종속성 제거
- 속성, 메소드 및 이벤트
- 그래픽 캡슐화
- 등록

### 종속성 제거

컴포넌트를 사용 가능하게 하는 특성 중 하나는 코드의 어떤 부분에서도 컴포넌트가 수행할 수 있도록 작업에 대한 제한을 두지 않는 것입니다. 이러한 특성 때문에 컴포넌트는 다양한 조합, 순서 및 컨텍스트로 애플리케이션에 통합됩니다. 사용자는 전제 조건 없이 어떤 상황에서도 기능을 수행하는 컴포넌트를 디자인해야 합니다.

종속성을 제거한 한 가지 예가 widget 컨트롤의 *Handle* 속성입니다. GUI 애플리케이션을 작성한 경험이 있다면, 창이나 컨트롤을 다 만들 때까지 액세스하지 않도록 주의하는 것이 프로그램을 실행시키는 데 있어서 가장 어렵고 오류가 발생하기 쉬운 측면 중의 하나임을 알 수 있습니다. *Handle*은 원본으로 사용한 widget의 인스턴스에 대한 고유 식별자입니다. 하위 수준의 함수 호출을 Qt 공유 라이브러리 등으로 만들 때 *Handle*을 사용할 수 있는데, 여기에서 *QWidget* 객체에 대한 고유 식별자가 함수 호출에 필요합니다. widget이 존재하지 않을 경우, *Handle*을 읽으면 컴포넌트가 원본으로 사용한 widget을 만듭니다. 따라서 애플리케이션 코드는 *Handle* 속성을 액세스할 때마다 유효한 핸들을 확실하게 얻게 됩니다.

Kylix는 widget 작성과 같은 백그라운드 작업을 제거하여 개발자가 정말로 원하는 작업에 전념할 수 있게 해 줍니다. 개발자는 핸들을 widget에 전달하기 전 핸들이 존재하는지 확인하거나 창을 만들 필요가 없습니다. 따라서 응용프로그램 개발자는 문제가 발생하는지 계속 확인하는 대신 모든 것이 정상적으로 작동한다고 가정할 수 있습니다.

종속성이 없는 컴포넌트를 작성하려면 많은 시간이 소요될 수 있지만 대체로 시간을 투자할 만한 가치가 있습니다. 애플리케이션 개발자의 반복되는 과중한 업무를 줄여 주는 것은 물론 설명서와 지원 부담을 덜어 주기 때문입니다.

## 속성, 메소드 및 이벤트

---

폼 디자이너에서 조작하는 시각적 이미지 이외에 컴포넌트의 가장 분명한 특성은 속성, 이벤트 및 메소드입니다. 이 설명서에서는 이러한 특성을 각 장에서 개별적으로 다루고 있지만 여기서는 각 특성을 사용하는 이유에 대해 간단하게 설명합니다.

### 속성

속성은 애플리케이션 개발자에게 변수 값의 설정이나 읽기에 대한 오해를 제공하는 동시에, 컴포넌트 작성자가 원본으로 사용한 데이터 구조를 숨기거나 값이 액세스될 때 특별한 처리를 구현할 수 있도록 합니다.

속성을 사용하면 다음과 같은 여러 가지 이점이 있습니다.

- 디자인 타임에 속성을 사용할 수 있습니다. 애플리케이션 개발자는 코드를 작성하지 않고 속성의 초기 값을 설정하거나 변경할 수 있습니다.
- 속성은 애플리케이션 개발자가 할당하는 값이나 형식을 확인할 수 있습니다. 디자인 타임에 입력을 검증하면 오류가 방지됩니다.
- 컴포넌트는 요청 시 적절한 값을 생성할 수 있습니다. 프로그래머가 범하는 가장 일반적인 오류는 초기화되지 않은 변수를 참조하는 것입니다. 데이터를 속성과 함께 나태내면 필요에 따라 항상 값을 사용할 수 있습니다.
- 속성을 사용하면 사용자는 간단하고 일관된 인터페이스에서 데이터를 숨길 수 있습니다. 변경 내용을 애플리케이션 개발자가 볼 수 있도록 설정하지 않아도 속성에 정보를 구성하는 방법을 바꿀 수 있습니다.

26장 "속성 생성"에는 컴포넌트에 속성을 추가하는 방법이 설명되어 있습니다.

### 이벤트

이벤트는 런타임에 입력이나 다른 활동에 응답하여 코드를 호출하는 특별한 속성입니다. 이벤트는 특정 코드 블록을 마우스 동작, 키 입력 등과 같은 특정 런타임 발생 항목에 연결할 수 있는 방법을 애플리케이션 개발자에게 제공합니다. 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.

27장 "이벤트 생성"에는 표준 이벤트를 구현하는 방법과 새 이벤트를 정의하는 방법이 설명되어 있습니다.



## 메소드

애플리케이션 개발자는 메소드를 사용하여 컴포넌트에게 특정 작업을 수행하거나 다른 속성이 포함하지 않는 값을 반환하도록 지시합니다. 메소드에는 두 가지 타입, 즉 클래스 메소드와 컴포넌트 메소드가 있습니다. 클래스 메소드는 클래스의 특정 인스턴스가 아니라 클래스에서 동작하는 프로시저 및 함수입니다. 예를 들어, 모든 컴포넌트의 생성자(*Create* 메소드)는 클래스 메소드입니다. 컴포넌트 메소드는 컴포넌트 인스턴스 자체에서 동작하는 프로시저 및 함수입니다.

메소드에는 코드 실행이 필요하기 때문에 런타임 시에만 메소드를 호출할 수 있습니다. 메소드가 유용한 몇 가지 이유는 다음과 같습니다.

- 메소드는 데이터가 상주하는 동일한 객체내에서 컴포넌트 기능을 캡슐화합니다.
- 메소드는 간단하고 일관된 인터페이스에 복잡한 프로시저를 숨길 수 있습니다. 애플리케이션 개발자는 특정 컴포넌트의 *AlignControls* 메소드가 작동하는 방법이나 다른 컴포넌트에 있는 동일한 메소드와의 차이점을 알지 못해도 그 메소드를 호출할 수 있습니다.
- 메소드를 사용하면 단일 호출로 여러 속성을 업데이트할 수 있습니다.

28장 "메소드 생성"에는 컴포넌트에 메소드를 추가하는 방법이 설명되어 있습니다.

## 그래픽 캡슐화

Kylix는 다양한 그래픽 도구를 캔버스에 캡슐화하여 그래픽을 단순화합니다. 캔버스는 창이나 컨트롤의 그리기 표면을 나타내며 펜, 브러시, 글꼴 등과 같은 다른 클래스를 포함합니다. 캔버스는 사용자를 위해 모든 부기 작업을 관리하는 페인터입니다.

폼이나 다른 컴포넌트에 그리려면 컴포넌트의 *Canvas* 속성을 액세스합니다. *Canvas*는 속성인 동시에 *TCanvas*라고 하는 객체입니다. *TCanvas*는 *Handle* 속성을 통해 액세스할 수 있는 Qt 페인터를 둘러싸는 래퍼입니다. 핸들을 사용하면 하위 수준의 Qt 그래픽 라이브러리 함수에 액세스할 수 있습니다.

펜이나 브러시를 사용자 지정하려면 색 또는 스타일을 설정합니다. 설정이 끝나면 Kylix는 리소스를 처리합니다. 또한 Kylix는 애플리케이션이 동일한 종류의 리소스를 자주 사용할 경우 리소스를 캐시로 저장하므로 다시 만들 필요가 없습니다.

Kylix 컴포넌트에서 파생하여 Kylix 컴포넌트에 내장된 캔버스를 사용할 수 있습니다. 그래픽 이미지가 컴포넌트에서 작동하는 방법은 컴포넌트가 파생된 객체의 캔버스에 따라 달라집니다. 그래픽 기능에 대한 자세한 내용은 29장 "컴포넌트에서 그래픽 사용"에 설명되어 있습니다.

## 등록

Kylix IDE에 컴포넌트를 설치할 수 있으려면 우선 컴포넌트를 등록해야 합니다. 컴포넌트를 등록하면 Kylix에서 컴포넌트를 컴포넌트 팔레트의 어느 위치에 두어야 할지 알려 줍니다. 또한 Kylix가 컴포넌트를 폼 파일에 저장하는 방법을 사용자 지정할 수도 있습니다. 컴포넌트 등록에 대한 내용은 30장 "디자인 타임 시 컴포넌트 사용"을 참조하십시오.

## 새 컴포넌트 생성

---

다음 두 가지 방법으로 새 컴포넌트를 만들 수 있습니다.

- 컴포넌트 마법사 사용
- 수동으로 컴포넌트 생성

이러한 방법 중 하나를 사용하여 컴포넌트 팔레트에 설치할 수 있는 최소한의 기능을 가진 컴포넌트를 만들 수 있습니다. 설치 후에는 새 컴포넌트를 폼에 추가하고, 디자인 타임 및 런타임 시 새 컴포넌트를 테스트할 수 있습니다. 그런 다음, 더 많은 기능을 컴포넌트에 추가하고, 컴포넌트 팔레트를 업데이트하며, 테스트를 계속 할 수 있습니다.

새 컴포넌트를 만들 때마다 수행되는 몇 가지 기본 단계가 있습니다. 이러한 단계는 아래에 설명되어 있습니다. 이 문서의 다른 예제에서는 사용자가 이러한 단계를 수행하는 방법을 알고 있다고 가정합니다.

- 1 새 컴포넌트의 유닛을 만듭니다.
- 2 기존 컴포넌트 타입에서 컴포넌트를 파생시킵니다.
- 3 속성, 메소드 및 이벤트를 추가합니다.
- 4 Kylix에 컴포넌트를 등록합니다.
- 5 컴포넌트와 그 속성, 메소드 및 이벤트에 대한 도움말 파일을 만듭니다.
- 6 Kylix IDE에 컴포넌트를 설치할 수 있도록 패키지(특수 공유 객체 파일)를 만듭니다.

이를 완료하면 완성된 컴포넌트에 다음과 같은 파일이 포함됩니다.

- 패키지 (bpl<packagename>.so)
- 컴파일된 패키지 (.dcp 파일)
- 컴파일된 유닛 (.dcu 및 .dpu 파일)
- 팔레트 비트맵 (.dcr 파일)
- 도움말 파일

컴포넌트 사용자에게 컴포넌트 사용법을 알려 주는 도움말 파일의 생성은 선택 사항입니다.

4부의 나머지 장에서는 컴포넌트 구축에 대한 모든 측면을 설명하고 다른 종류의 컴포넌트를 작성하는 몇 가지 예제를 제공합니다.

## 컴포넌트 마법사 사용

---

컴포넌트 마법사는 컴포넌트 생성의 초기 단계를 단순화합니다. 컴포넌트 마법사를 사용할 경우 다음을 지정해야 합니다.

- 새 컴포넌트가 파생되는 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 컴포넌트 마법사는 컴포넌트를 수동으로 만들 때와 동일한 작업을 수행합니다.

- 유닛 생성
- 컴포넌트 파생
- 컴포넌트 등록

컴포넌트 마법사는 기존 유닛에 컴포넌트를 추가할 수 없습니다. 따라서 수동으로 기존 유닛에 컴포넌트를 추가해야 합니다.

컴포넌트 마법사를 시작하려면 다음 두 가지 방법 중 하나를 선택합니다.

- Component|New Component를 선택합니다

또는

- File|New를 선택하고 Component를 더블 클릭합니다

다음과 같이 컴포넌트 마법사의 필드에 입력합니다.

- 1 Ancestor Type 필드에서 새 컴포넌트를 파생시킬 클래스를 지정합니다.
- 2 Class Name 필드에서 새 컴포넌트 클래스의 이름을 지정합니다.
- 3 Palette Page 필드에서 새 컴포넌트를 설치하고자 하는 컴포넌트 팔레트의 페이지를 지정합니다.
- 4 Unit file name 필드에서 컴포넌트 클래스를 선언하고자 하는 유닛의 이름을 지정합니다.
- 5 유닛이 검색 경로에 없는 경우 필요에 따라 Search Path 필드에서 검색 경로를 편집합니다.

컴포넌트를 새 패키지나 기존 패키지에 두려면 Component|Install을 클릭한 다음 나타나는 대화 상자를 사용하여 패키지를 지정합니다.

**경고** *TCustomControl*과 같이 이름이 "custom"으로 시작하는 CLX 클래스에서 컴포넌트를 파생시킬 경우, 원래 컴포넌트의 모든 추상 메소드를 오버라이드할 때까지 새 컴포넌트를 폼에 두려고 하면 안됩니다. Kylix는 추상 메소드가 있는 클래스의 인스턴스 객체를 만들 수 없습니다.

Kylix는 클래스 선언 및 *Register* 프로시저를 포함한 새 유닛을 만들고, 모든 표준 Kylix 유닛을 포함하는 **uses** 절을 추가합니다. 유닛의 소스 코드를 보려면 View Unit을 클릭합니다. 컴포넌트 마법사를 이미 닫은 경우 File|Open을 선택하여 코드 에디터에서 유닛 파일을 엽니다. 유닛은 다음과 같이 나타납니다.

```
unit MyControl;

interface

uses
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
    TMyControl = class(TCustomControl)
    private
```

```
{ Private declarations }
protected
{ Protected declarations }
public
{ Public declarations }
published
{ Published declarations }
end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyControl]);
end;

end.
```

## 수동으로 컴포넌트 생성

---

새 컴포넌트를 만드는 가장 쉬운 방법은 컴포넌트 마법사를 사용하는 것입니다. 그러나 같은 단계를 수동으로 수행할 수도 있습니다.

컴포넌트를 수동으로 만드는 과정에는 다음과 같은 작업이 포함됩니다.

- 유닛 파일 생성
- 컴포넌트 파생
- 컴포넌트 등록

### 유닛 파일 생성

유닛은 별도로 컴파일된 오브젝트 파스칼 코드의 모듈입니다. Kylix는 유닛을 여러 용도로 사용합니다. 모든 폼에는 고유한 유닛이 있고, 대부분의 컴포넌트 또는 관련 컴포넌트의 그룹에도 고유한 유닛이 있습니다.

컴포넌트를 만들 때 컴포넌트의 새 유닛을 만들거나 기존 유닛에 새 컴포넌트를 추가합니다.

유닛을 만들려면 File|New를 선택하고 Unit을 더블 클릭합니다. 그러면 Kylix는 새 유닛 파일을 만들고 코드 에디터에서 그 파일을 엽니다.

기존 유닛을 열려면 File|Open을 선택한 다음 컴포넌트를 추가할 소스 코드 유닛을 선택합니다.

**참고** 기존 유닛에 컴포넌트를 추가할 경우, 유닛에 컴포넌트 코드만 포함되어 있는지 확인합니다. 예를 들어, 폼이 포함된 유닛에 컴포넌트 코드를 추가하면 컴포넌트 팔레트에서 오류가 발생합니다.

컴포넌트에 대한 새 유닛 또는 기존 유닛을 갖게 되면, 컴포넌트 클래스를 파생시킬 수 있습니다.

## 컴포넌트 파생

모든 컴포넌트는 *TComponent*에서 파생되거나, *TControl*이나 *TGraphicControl*과 같이 보다 특수한 자손 중 하나에서 파생되거나, 또는 기존 컴포넌트 클래스에서 파생된 클래스입니다. 24-3 페이지의 "컴포넌트 생성 방법"에는 각기 다른 종류의 컴포넌트를 파생시키는 클래스에 대한 지침이 제공됩니다. 그러나 사용자는 가장 적합한 객체를 결정하기 위해 CLX 객체 계층 구조를 검토해야 합니다.

컴포넌트를 파생시키려면 컴포넌트를 포함할 유닛의 **interface** 부분에 객체 타입 선언을 추가합니다. 클래스 파생에 대한 자세한 내용은 25-1 페이지의 "새 클래스 정의" 단원에 설명되어 있습니다.

일반 컴포넌트 클래스는 *TComponent*에서 직접 파생된 년비주얼 컴포넌트입니다.

일반 컴포넌트를 만들려면 다음 클래스 선언을 컴포넌트 유닛의 **interface** 부분에 추가합니다.

```
type
  TNewComponent = class(TComponent)
  end;
```

이제까지 작업을 보면 새 컴포넌트는 *TComponent*와 다르지 않습니다. 새 컴포넌트 구축에 대한 프레임워크가 만들어졌습니다.

## 컴포넌트 등록

등록은 Kylix에게 컴포넌트 라이브러리에 추가할 컴포넌트와 컴포넌트를 나타내야 할 컴포넌트 팔레트의 페이지를 알려 주는 간단한 과정입니다. 등록 과정에 대한 자세한 설명을 보려면 30장 "디자인 타임 시 컴포넌트 사용"을 참조하십시오.

다음과 같은 방법으로 컴포넌트를 등록합니다.

- 1 *Register*라는 프로시저를 컴포넌트 유닛의 **interface** 부분에 추가합니다. *Register*는 매개변수를 갖지 않기 때문에 다음과 같이 매우 간단하게 선언할 수 있습니다.

```
procedure Register;
```

이미 컴포넌트가 포함된 유닛에 컴포넌트를 추가하는 경우에, 선언된 *Register* 프로시저를 유닛이 갖고 있으므로 선언을 변경할 필요가 없습니다.

- 2 *Register* 프로시저를 유닛의 **implementation** 부분에 작성하고, 등록하고자 하는 각 컴포넌트에 대해 *RegisterComponents*를 호출합니다. *RegisterComponents*는 두 가지 매개변수, 즉 컴포넌트 팔레트 페이지의 이름과 컴포넌트 타입 집합을 갖는 프로시저입니다. 기존 등록에 컴포넌트를 추가하는 경우, 기존 문장의 집합에 새 컴포넌트를 추가하거나 *RegisterComponents*를 호출하는 새 문장을 추가할 수 있습니다.

*TMyControl*이라는 컴포넌트를 등록하여 팔레트의 Samples 페이지에 놓으려면, 다음 *Register* 프로시저를 *TMyControl*의 선언을 포함하는 유닛에 추가합니다.

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewControl]);
end;
```

이 *Register* 프로시저는 *TMyControl*을 컴포넌트 팔레트의 Samples 페이지에 둡니다. 일단 컴포넌트를 등록하면, 패키지로 컴파일하고 (30장 참조) 컴포넌트 팔레트에 설치할 수 있습니다.

## 설치되지 않은 컴포넌트 테스트

컴포넌트 팔레트에 설치하기 전 컴포넌트의 런타임 작동을 테스트할 수 있습니다. 이것은 새로 만든 컴포넌트를 디버깅하는 데 특히 유용하지만, 동일한 기술이 컴포넌트 팔레트에 있는지 여부에 상관없이 모든 컴포넌트에 작동합니다. 이미 설치된 컴포넌트를 테스트하는 것에 대한 자세한 내용은 24-14 페이지의 "설치된 컴포넌트 테스트"를 참조하십시오.

팔레트에서 컴포넌트를 선택하여 폼에 놓을 때 Kylix가 수행하는 작업을 에뮬레이트하여 설치되지 않는 컴포넌트를 테스트합니다.

다음과 같은 방법으로 설치되지 않은 컴포넌트를 테스트합니다.

- 1 컴포넌트 유닛의 이름을 폼 유닛의 **uses** 절에 추가합니다.
- 2 객체 필드를 폼에 추가하여 컴포넌트를 나타냅니다.

이 작업은 사용자가 컴포넌트를 추가하는 방법과 IDE가 컴포넌트를 추가하는 방법 사이에서 중요한 차이점 중의 하나입니다. 사용자는 폼의 타입 선언의 맨 아래에 있는 public 부분에 객체 필드를 추가합니다. IDE는 자신이 관리하는 타입 선언 부분에서 위의 객체 필드를 추가합니다.

IDE가 관리하는 폼 타입의 선언 부분에 필드를 추가해서는 안됩니다. 타입 선언의 이 부분에 있는 항목은 폼 파일에 저장된 항목에 해당합니다. 폼에 존재하지 않는 컴포넌트의 이름을 추가할 경우 폼 파일을 유효하지 않게 만들 수 있습니다.

- 3 폼의 *OnCreate* 이벤트에 핸들러를 첨부합니다.
- 4 폼의 *OnCreate* 핸들러에서 컴포넌트를 생성합니다.

컴포넌트의 생성자를 호출한 경우, 컴포넌트 소유자 (컴포넌트를 폼 파일에 저장하고 시간이 되었을 때 컴포넌트를 소멸시키는 작업을 담당하는 컴포넌트)를 지정하는 매개변수를 전달해야 합니다. 거의 항상 *Self*를 소유자로 전달합니다. 메소드에서 *Self*는 메소드를 포함한 객체에 대한 참조가 됩니다. 이 경우에는 폼의 *OnCreate* 핸들러에서 *Self*가 폼을 참조합니다.

- 5 컴포넌트가 컨트롤인 경우 *Parent* 속성을 할당합니다.

*Parent* 속성의 설정은 항상 컨트롤을 생성한 후 가장 먼저 해야 할 작업입니다. *Parent* 는 사용자의 컨트롤을 시각적으로 포함한 컨트롤이며, 대개 컨트롤이 표시되는 폼이지만, 그룹 상자나 패널이 될 수도 있습니다. 보통 *Parent*를 *Self*, 즉 폼으로 설정합니다. 컨트롤의 다른 속성을 설정하기 전에 항상 *Parent*를 먼저 설정합니다.

## 경고

컴포넌트가 컨트롤이 아닌 경우, 다시 말해 *TControl*이 컴포넌트의 조상 중 하나가 아닌 경우에는 이 단계를 건너뛵니다. 실수로 컴포넌트 대신에 폼의 *Parent* 속성을 *Self*로 설정하면 운영 체제에 문제가 발생할 수 있습니다.

## 6 다른 모든 컴포넌트 속성을 원하는대로 설정합니다.

예를 들어, *MyControl*이라고 하는 유닛에서 *TMyControl* 타입의 새 컴포넌트를 테스트하려면, 새 프로젝트를 만든 후 단계에 따라 다음과 같이 나타나는 폼 유닛을 만듭니다.

```
unit Unit1;
interface

uses
  SysUtils, Types, Classes, QGraphics, QControls;
  QForms, QDialogs, MyControl;           { 1. Add MyControl to uses clause }

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);   { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    MyControl1: TMyControl1;                 { 2. Add an object field }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender:TObject);
begin
  MyControl1 := TMyControl.Create(Self);      { 4. Construct the component }
  MyControl1.Parent := Self;                  { 5. Set Parent property if component is a control }
  MyControl1.Left := 12;                      { 6. Set other properties... }
  :                                           ...continue as needed }
end;
end.
```

## 설치된 컴포넌트 테스트

---

컴포넌트 팔레트에 설치한 후 컴포넌트의 디자인 타임 동작을 테스트할 수 있습니다. 이것은 새로 만든 컴포넌트를 디버깅하는 데 특히 유용하지만, 동일한 기술이 컴포넌트 팔레트에 있는지 여부에 상관없이 모든 컴포넌트에 작동합니다. 아직 설치하지 않은 컴포넌트를 테스트하는 것에 대한 자세한 내용은 24-12 페이지의 "설치되지 않은 컴포넌트 테스트"를 참조하십시오.

설치 후 컴포넌트를 테스트하면 폼에 가져다 놓을 때 디자인 타임 예외만을 생성하는 컴포넌트를 디버깅할 수 있습니다.

Kylix의 두 번째 실행 인스턴스를 사용하여 설치된 컴포넌트를 테스트합니다.

- 1 Kylix IDE 메뉴에서 Project|Options를 선택하고 Directories/Conditional 페이지에서 Debug Source Path를 컴포넌트 소스 파일로 설정합니다.
- 2 그런 다음, Tools|Debugger Options를 선택합니다. Language Exceptions 페이지에서 추적하고자 하는 예외를 활성화합니다.
- 3 컴포넌트 소스 파일을 열고 브레이크포인트를 설정합니다.
- 4 Run|Parameters를 선택하고 Host Application 필드를 Kylix 실행 파일의 이름과 위치로 설정합니다.
- 5 Run Parameters 대화 상자에서 Load 버튼을 클릭해서 Kylix의 두 번째 인스턴스를 시작합니다.
- 6 그리고 나서, 테스트할 컴포넌트를 폼에 가져다 놓습니다. 이 작업은 소스의 브레이크포인트에서 중단되어야 합니다.



## 컴포넌트 작성자를 위한 객체 지향 프로그래밍

Kylix를 사용하여 애플리케이션을 작성했다면, 클래스에 데이터와 코드가 포함되며 사용자가 디자인 타임과 런타임에 클래스를 조작할 수 있습니다. 이러한 사실을 안다는 것은 이미 컴포넌트 사용자가 되었다는 것을 의미합니다.

사용자가 새 컴포넌트를 만들 때는 애플리케이션 개발자가 작업을 할 필요가 없도록 클래스를 처리합니다. 또한 사용자는 컴포넌트를 사용할 개발자가 볼 수 없도록 컴포넌트의 내부 작업을 숨기려고 합니다. 컴포넌트를 위한 적절한 조상을 선택하고, 개발자에게 필요한 속성과 메소드만을 표시하는 인터페이스를 디자인하며, 이 장에서 제공하는 기타 지침에 따르면 다양한 기능의 재사용 가능한 컴포넌트를 만들 수 있습니다.

컴포넌트를 만들고자 한다면 우선 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 조작한다는 점입니다.

클래스는 기본적으로 하나의 타입입니다. 프로그래머라면 항상 타입과 인스턴스를 다루게 되며, 이러한 용어를 사용하지 않는 경우에도 마찬가지입니다. 예를 들면, *Integer* 같은 타입의 변수를 만들 수 있습니다. 대개 클래스는 일반 데이터 타입보다 복잡하지만 동작 방식은 동일합니다. 즉, 같은 타입의 인스턴스에 다른 값을 할당하면 다른 작업을 수행할 수 있습니다.

예를 들어, 하나는 OK로, 다른 하나는 Cancel로 레이블이 지정된 버튼 두 개를 포함한 폼을 만드는 경우를 흔히 볼 수 있습니다. 여기서 각 버튼은 *TButton* 클래스의 인스턴스지만, *Caption* 속성에 다른 값을 할당하고 *OnClick* 이벤트에 다른 핸들러를 할당하여 다르게 동작하는 두 개의 인스턴스를 만듭니다.

## 새 클래스 파생

---

새 클래스를 파생시키는 이유는 다음 두 가지입니다.

- 반복을 피하기 위한 클래스 기본값 변경
- 클래스에 새 기능 추가

각 경우에서의 목표는 재사용할 수 있는 객체를 만드는 것입니다. 재사용을 염두에 두면서 컴포넌트를 디자인하는 경우, 나중에 수행할 작업을 줄일 수 있습니다. 클래스에 사용 가능한 기본값을 제공하되 기본값을 사용자 지정할 수 있도록 합니다.

### 반복을 피하기 위한 클래스 기본값 변경

대부분의 프로그래머는 반복을 피하려고 합니다. 따라서 사용자가 같은 코드 줄을 반복하여 재작성하는 것을 깨닫는 경우, 하위 루틴 또는 함수에 코드를 두거나 여러 프로그램에서 사용할 수 있는 루틴 라이브러리를 만듭니다. 컴포넌트에도 같은 상황이 적용됩니다. 즉, 동일한 속성을 변경하거나 동일한 메소드를 호출하는 대신 이러한 작업을 기본적으로 수행하는 새 컴포넌트를 만들 수 있습니다.

예를 들어, 애플리케이션을 만들 때마다 대화 상자를 추가하여 특정 작업을 수행한다고 가정합니다. 매번 대화 상자를 다시 만드는 것이 어렵지는 않지만 불필요한 일이기도 합니다. 그 대신 대화 상자를 한 번만 디자인하고 속성을 설정한 다음 대화 상자에 연결된 래퍼(wrapper) 컴포넌트를 컴포넌트 팔레트에 설치할 수 있습니다. 이렇게 대화 상자를 재사용할 수 있는 컴포넌트로 만들면, 반복적인 작업을 없앨 수 있을 뿐만 아니라 표준화를 개선하고 대화 상자를 다시 만들 때마다 발생할 수 있는 오류 가능성을 줄일 수 있습니다.

31장 "기존 컴포넌트 수정"에는 컴포넌트의 기본 속성을 변경하는 예제가 나와 있습니다.

**참고** 기존 컴포넌트의 *published*인 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 만드는 일반적인 이유는 기존 컴포넌트에 없는 기능을 추가하기 위해서입니다. 이러한 작업을 하는 경우에는 *TComponent* 또는 *TControl*과 같은 추상 기본 클래스나 기존 컴포넌트에서 새 컴포넌트를 파생시킵니다.

새 컴포넌트를 파생시킬 때는 원하는 기능과 가장 가까운 기능의 서브셋이 포함된 클래스에서 파생시킵니다. 클래스에 기능을 추가할 수 있지만 제거할 수는 없습니다. 따라서 기존 컴포넌트 클래스에 포함하고 싶지 않은 속성이 들어 있는 경우에는 해당 컴포넌트의 조상에서 파생시켜야 합니다.

예를 들어, 목록 상자에 기능을 추가하려는 경우 *TListBox*에서 컴포넌트를 파생시킬 수 있습니다. 그러나 새 기능을 추가하되 표준 목록 상자의 몇 가지 기능을 제외하려는 경우에는 *TListBox*의 조상인 *TCustomListBox*에서 컴포넌트를 파생시켜야 합니다. 그 다음, 원하는 목록 상자의 기능만을 다시 만들거나 보이도록 하고, 새 기능을 추가할 수 있습니다.

33장 "그리드 사용자 지정"에는 추상 컴포넌트 클래스를 사용자 지정하는 예제가 나와 있습니다.

## 새 컴포넌트 클래스 선언

Kylix는 표준 컴포넌트 외에, 새 컴포넌트를 파생시키기 위한 기본으로서 여러 개의 추상 기본 클래스를 제공합니다. 24-3 페이지의 표 24.1은 사용자 고유 컴포넌트를 만들 때부터 시작할 수 있는 클래스를 보여 줍니다.

새 컴포넌트 클래스를 선언하려면, 컴포넌트의 유닛 파일에 클래스 선언을 추가합니다.

다음은 간단한 그래픽 컴포넌트를 선언한 것입니다.

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

완성된 컴포넌트 선언에는 **end** 앞에 속성, 이벤트 및 메소드 선언이 일반적으로 포함됩니다. 그러나 위와 같은 선언도 유효하며, 컴포넌트 기능 추가를 위한 출발점을 제공합니다.

## 조상, 자손 및 클래스 계층 구조

애플리케이션 개발자는 모든 컨트롤이 폼에서의 위치를 결정하는 *Top* 및 *Left*라고 하는 속성을 가지는 것을 당연하다고 생각합니다. 애플리케이션 개발자에게 있어서 모든 컨트롤이 공통 조상인 *TControl*로부터 이러한 속성을 상속한다는 것은 중요한 문제 아닙니다. 그러나, 컴포넌트를 만들 경우에는 컴포넌트가 적절한 기능을 상속하도록 컴포넌트가 파생되는 클래스를 알아야 합니다. 또한 상속된 기능을 다시 만들지 않고 활용할 수 있도록 컨트롤이 상속하는 모든 기능을 알아야 합니다.

컴포넌트를 파생시키는 클래스를 컴포넌트의 *직계 조상*이라고 합니다. 각 컴포넌트는 자신의 직계 조상은 물론 직계 조상의 직계 조상으로부터도 기능을 상속합니다. 컴포넌트가 기능을 상속하는 모든 클래스를 *조상*이라고 부르며, 결국 컴포넌트는 조상의 *자손*이 됩니다.

애플리케이션에서 조상과 자손의 모든 관계는 클래스의 계층 구조를 구성합니다. 특정 클래스가 조상으로부터 모든 것을 상속하기 때문에 계층 구조에 있는 각 클래스는 조상보다 더 많은 것을 포함하고, 따라서 새 속성과 메소드를 추가하거나 기존의 속성과 메소드를 다시 정의합니다.

직계 조상이 지정되지 않은 경우 Kylix는 기본 조상인 *TObject*에서 클래스를 파생시킵니다. *TObject*는 객체 계층 구조에 있는 모든 클래스의 궁극적인 조상입니다.

파생시킬 객체를 선택하기 위한 일반 규칙은 다음과 같이 간단합니다. 즉, 새 객체에 포함하려는 기능을 가능한 많이 가지고 있고 새 객체에 포함하지 않으려는 기능을 가지고 있지 않은 객체를 선택합니다. 객체에 기능을 추가할 수 있지만 제거할 수는 없습니다.

## 엑세스 제어

속성, 메소드 및 필드에는 *가시성*이라고 하는 여러 수준의 *엑세스 제어*가 있습니다. 가시성은 클래스의 특정 부분에 액세스할 수 있는 코드를 결정합니다. 가시성을 지정하여 컴포넌트에 대한 *인터페이스*를 정의합니다.

표 25.1에는 액세스가 가장 제한적인 것부터 가장 허용적인 것까지 가시성의 수준이 나와 있습니다.

**표 25.1** 객체 내의 가시성 수준

| 가시성       | 의미                               | 용도                  |
|-----------|----------------------------------|---------------------|
| private   | 클래스가 정의된 유닛의 코드에서만 액세스 가능        | 구현 세부 사항 숨기기.       |
| protected | 클래스 및 자손이 정의된 유닛의 코드에서 액세스 가능    | 컴포넌트 작성자의 인터페이스 정의. |
| public    | 모든 코드에서 액세스 가능                   | 런타임 인터페이스 정의.       |
| published | 모든 코드와 Object Inspector에서 액세스 가능 | 디자인 타임 인터페이스 정의.    |

멤버가 정의된 클래스에서만 사용할 수 있도록 하려면 멤버를 **private**로 선언하고, 클래스와 자손에서만 사용할 수 있도록 하려면 멤버를 **protected**로 선언합니다. 유닛 파일 내의 위치에 상관없이 멤버를 사용할 수 있는 경우 파일의 *모든 곳*에서 멤버를 사용할 수 있다는 것을 기억합니다. 따라서 같은 유닛에서 두 개의 클래스를 정의할 경우 각 클래스는 다른 클래스의 **private** 메소드에 액세스할 수 있습니다. 또한 조상으로부터 다른 유닛의 클래스를 파생시킬 경우 새 유닛의 모든 클래스는 조상의 **protected** 메소드에 액세스할 수 있습니다.

## 구현 세부 사항 숨기기

클래스 부분을 **private**로 선언하면 클래스 유닛 파일의 외부에 있는 코드로 해당 부분을 볼 수 없게 됩니다. 선언이 포함된 유닛 내에서는 코드가 **public**인 것처럼 이 부분에 액세스할 수 있습니다.

다음 예제는 필드를 **private**로 선언하여 애플리케이션 개발자가 볼 수 없도록 숨기는 방법을 보여 줍니다. 목록에서는 두 개의 폼 유닛을 보여 줍니다. 각 폼에는 **private** 필드에 값을 할당하는 *OnCreate* 이벤트에 대한 핸들러가 있습니다. 컴파일러는 자신이 선언된 폼에서만 필드에 대한 할당을 허용합니다.

```

unit HideInfo;
interface

uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TSecretForm = class(TForm)                                { declare new form }
  procedure FormCreate(Sender:TObject);
  private                                                    { declare private part }
    FSecretCode: Integer;                                     { declare a private field }
  end;

var
  SecretForm: TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender:TObject);
begin
  FSecretCode := 42;                                         { this compiles correctly }
end;
end.                                                         { end of unit }

unit TestHide;                                              { this is the main form file }

interface
uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs,
  HideInfo;                                                  { use the unit with TSecretForm }

type
  TTestForm = class(TForm)
  procedure FormCreate(Sender:TObject);
  end;

var
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender:TObject);
begin
  SecretForm.FSecretCode := 13;    { compiler stops with "Field identifier expected" }
end;
end.                                                         { end of unit }

```

*HideInfo* 유닛을 사용하는 프로그램은 *TSecretForm* 타입의 객체를 사용할 수 있지만 이러한 객체에 있는 *FSecretCode* 필드는 액세스할 수 없습니다.

## 컴포넌트 작성자의 인터페이스 정의

클래스 부분을 **protected**로 선언하면 클래스 및 그 자손과 유닛 파일을 공유하는 다른 클래스에서만 해당 부분을 볼 수 있습니다.

**protected** 선언을 사용하면 클래스에 대한 *컴포넌트 작성자의 인터페이스*를 정의할 수 있습니다. 애플리케이션 유닛은 **protected** 부분에 액세스할 수 없지만 파생된 클래스는 이 부분에 액세스할 수 있습니다. 이는 컴포넌트 작성자가 애플리케이션 개발자가 세부 사항을 볼 수 있도록 만들지 않아도 클래스의 작동 방법을 변경할 수 있다는 것을 의미합니다.

**참고** 공통적인 실수 중 하나가 이벤트 핸들러에서 `protected` 메소드에 액세스하려는 것입니다. 이벤트 핸들러는 이벤트를 받는 컴포넌트가 아니라 폼의 메소드입니다. 결과적으로 이벤트 핸들러는 컴포넌트가 동일한 유닛에서 폼으로 선언되지 않는 경우에 컴포넌트의 `protected` 메소드에 액세스할 수 없습니다.

## 런타임 인터페이스 정의

클래스 부분을 **public**으로 선언하면 클래스에 전체적으로 액세스할 수 있는 모든 코드에서 해당 부분을 볼 수 있습니다.

`public` 부분은 런타임 시 모든 코드에서 사용할 수 있으므로 클래스의 `public` 부분은 *런타임 인터페이스*를 정의합니다. 런타임 인터페이스는 런타임 입력에 따라 달라지는 속성이나 읽기 전용인 속성 같이 디자인 타임에 무의미하거나 적절하지 않은 항목에 대해 사용할 수 있습니다. 또한 애플리케이션 개발자가 호출하도록 만드는 메소드도 `public`이어야 합니다.

다음 예제는 컴포넌트의 런타임 인터페이스 부분으로 선언된 두 개의 읽기 전용 속성을 보여 줍니다.

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;           { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;      { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
:
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

## 디자인 타임 인터페이스 정의

클래스 부분을 **published**로 선언하면, 해당 부분이 `public`이 되고 런타임 타입 정보를 생성합니다. 무엇보다도 런타임 타입 정보는 Object Inspector가 속성 및 이벤트에 액세스할 수 있도록 합니다.

`published` 부분은 Object Inspector에서 나타나기 때문에 해당 클래스의 *디자인 타임 인터페이스*를 정의합니다. 디자인 타임 인터페이스는 애플리케이션 개발자가 디자인 타임에 사용자 지정하려는 클래스의 모든 측면을 포함해야 하지만 런타임 환경에 대한 특정 정보에 의존하는 속성을 포함해서는 안 됩니다.

다음 예제는 *Temperature*라고 하는 published 속성을 보여 줍니다. 이 속성은 published 이기 때문에 디자인 타임에 Object Inspector에 나타납니다.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;           { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature; { writable! }
  end;
```

## 메소드 디스패칭

*디스패칭*은 메소드 호출이 발생했을 때 프로그램이 메소드를 호출해야 하는 위치를 결정하는 방법을 나타냅니다. 메소드를 호출하는 코드는 다른 모든 프로시저나 함수 호출과 같지만 클래스는 메소드를 디스패칭하는 각기 다른 방법을 가집니다.

메소드 디스패치에는 다음과 같은 세 가지 타입이 있습니다.

- 정적
- 가상
- 동적

### 정적 메소드

메소드 선언 시기를 지정하지 않은 경우 모든 메소드는 정적입니다. 정적 메소드는 일반 프로시저나 함수와 같은 방식으로 작동합니다. 컴파일러는 메소드의 정확한 주소를 확인하고 컴파일 타임에 해당 메소드를 연결합니다.

정적 메소드의 가장 큰 이점은 메소드를 매우 빠르게 디스패칭한다는 것입니다. 컴파일러는 메소드의 정확한 주소를 결정하기 때문에 메소드를 직접 연결합니다. 반대로 가상 메소드 및 동적 메소드는 런타임에 간접적인 방법으로 메소드 주소를 찾기 때문에 시간이 다소 오래 걸립니다.

정적 메소드는 자손 클래스가 상속할 때 변경되지 않습니다. 정적 메소드가 포함된 클래스를 선언한 다음 여기서 새 클래스를 파생시킬 경우, 파생된 클래스는 정확하게 동일한 주소에서 동일한 메소드를 공유합니다. 이는 정적 메소드를 오버라이드할 수 없으며 정적 메소드는 자신이 호출된 클래스에 상관 없이 항상 동일한 작업을 정확히 수행한다는 것을 의미합니다. 이름이 같은 파생된 클래스에 있는 메소드를 조상 클래스의 정적 메소드로 선언할 경우, 새 메소드는 단지 파생된 클래스의 상속된 메소드를 대체합니다.

### 정적 메소드의 예

다음 코드에서, 첫 번째 컴포넌트가 두 개의 정적 메소드를 선언합니다. 두 번째 컴포넌트는 첫 번째 컴포넌트에 상속된 메소드를 대체하는 동일한 이름의 정적 메소드 두 개를 선언합니다.

```
type
  TFirstComponent = class(TComponent)
  procedure Move;
```

```

    procedure Flash;
end;

TSecondComponent = class(TFirstComponent)
    procedure Move; { different from the inherited method, despite same declaration }
    function Flash(HowOften:Integer): Integer; { this is also different }
end;

```

## 가상 메소드

가상 메소드는 정적 메소드보다 복잡하고 유연성 있는 디스패치 메커니즘을 사용합니다. 가상 메소드는 자손 클래스에서 재정의할 수 있지만 호출은 여전히 조상 클래스에서 할 수 있습니다. 가상 메소드의 주소는 컴파일 타임에 결정되지 않습니다. 그 대신, 메소드가 정의된 객체가 런타임에 주소를 찾습니다.

가상 메소드를 만들려면 메소드 선언 뒤에 지시어 **virtual**을 추가합니다. **virtual** 지시어는 모든 가상 메소드 주소를 객체 타입으로 유지하는 객체의 *가상 메소드 테이블* (VMT)에 항목을 만듭니다.

기존 클래스에서 새 클래스를 파생시키면 새 클래스는 조상의 VMT에 있는 모든 항목과 새 클래스에서 추가로 선언된 가상 메소드를 포함하는 고유한 VMT를 갖게 됩니다.

## 메소드 오버라이드

메소드 *오버라이드*는 메소드를 대체하는 것이 아니라 확장 또는 개량하는 것을 의미합니다. 자손 클래스는 상속된 모든 가상 메소드를 오버라이드할 수 있습니다.

자손 클래스에서 메소드를 오버라이드하려면 지시어 **override**를 메소드 선언의 끝 부분에 추가합니다.

다음과 같은 경우 메소드를 오버라이드하면 컴파일 오류가 발생합니다.

- 조상 클래스에 메소드가 존재하지 않는 경우
- 해당 이름의 조상 메소드가 정적인 경우
- 선언이 동일하지 않은 경우 (인수 매개변수의 개수와 타입이 다른 경우)

다음 코드는 두 개의 일반 컴포넌트의 선언을 보여 줍니다. 첫 번째 컴포넌트는 각기 다른 종류의 디스패칭을 가진 메소드 세 개를 선언합니다. 첫 번째 컴포넌트에서 파생된 두 번째 컴포넌트는 정적 메소드를 대체하고 가상 메소드를 오버라이드합니다.

```

type
    TFirstComponent = class(TCustomControl)
        procedure Move;           { static method }
        procedure Flash; virtual; { virtual method }
        procedure Beep; dynamic;  { dynamic virtual method }
    end;

    TSecondComponent = class(TFirstComponent)
        procedure Move;           { declares new method }
        procedure Flash; override; { overrides inherited method }
        procedure Beep; override; { overrides inherited method }
    end;

```



## 동적 메소드

동적 메소드는 디스패치 매커니즘이 약간 다른 가상 메소드입니다. 동적 메소드는 객체의 가상 메소드 테이블에 항목이 없기 때문에 객체가 소비하는 메모리 양을 줄일 수 있습니다. 그러나 동적 메소드 디스패치는 일반적인 가상 메소드 디스패치보다 속도가 다소 느립니다. 따라서 메소드를 자주 호출하거나 메소드 실행에 시간이 중요한 경우, 메소드를 동적이 아니라 가상으로 선언해야 할 것입니다.

객체는 동적 메소드의 주소를 저장해야 합니다. 그러나 이 경우 가상 메소드 테이블에서 항목을 받는 대신 동적 메소드가 별도로 나열됩니다. 동적 메소드 목록은 특정 클래스가 사용하거나 오버라이드한 메소드에 대한 항목만 포함됩니다. 반대로 가상 메소드 테이블은 상속 및 도입된 객체의 모든 가상 메소드를 포함합니다. 상속 트리에서 뒤로 움직이면서 각 조상의 동적 메소드 목록을 검색하여 상속된 동적 메소드를 디스패치합니다.

동적 메소드를 만들려면 지시어 **dynamic**을 메소드 선언 뒤에 추가합니다.

## 추상 클래스 멤버

---

조상 클래스에서 메소드가 **abstract**로 선언된 경우 애플리케이션에서 새 컴포넌트를 사용할 수 있으려면 우선 해당 클래스를 다시 선언하고 구현하여 자손 컴포넌트에서 표면화해야 합니다. Kylix에서는 추상 멤버가 포함된 클래스의 인스턴스를 만들 수 없습니다. 클래스의 상속된 부분을 표면화하는 것에 대한 자세한 내용은 26장 "속성 생성" 및 28장 "메소드 생성"을 참조하십시오.

## 클래스 및 포인터

---

모든 클래스와 컴포넌트는 사실상 포인터입니다. 컴파일러는 클래스 포인터를 자동으로 역참조하기 때문에 대개 사용자는 이에 대해 신경 쓸 필요가 없습니다. 포인터로서의 클래스 상태는 클래스를 매개변수로 전달할 때 중요해집니다. 클래스를 전달할 때는 참조가 아닌 값으로 전달해야 합니다. 클래스가 이미 참조인 포인터이기 때문에 클래스를 참조로 전달하면 참조를 참조에 전달하는 결과가 됩니다.



# 26

## 속성 생성

속성은 컴포넌트에서 가장 시각적인 부분입니다. 애플리케이션 개발자는 디자인 타임에 속성을 확인 및 조작하고, 폼 디자이너에서 컴포넌트 반응으로 즉각적인 피드백을 얻을 수 있습니다. 속성을 잘 디자인하면 컴포넌트를 다른 사람이 사용하기 쉽고 자신이 유지 관리하기도 쉽습니다.

컴포넌트의 속성을 가장 적절하게 사용하려면 다음 사항을 이해해야 합니다.

- 속성을 생성하는 이유
- 속성 타입
- 상속된 속성 계층
- 속성 정의
- 배열 속성 생성
- 속성 저장 및 로드

### 속성을 생성하는 이유

---

애플리케이션 개발자의 관점에서 보면 속성은 변수와 같아 보입니다. 개발자는 속성 값을 필드인 것처럼 설정하거나 읽을 수 있습니다. 변수를 사용하면 할 수 있지만 속성을 사용해서 할 수 없는 유일한 작업은 속성을 **var** 매개변수로 전달하는 것입니다.

다음과 같은 이유에서 속성은 일반 필드보다 강력한 기능을 제공합니다.

- 애플리케이션 개발자는 디자인 타임에 속성을 설정할 수 있습니다. 런타임에만 사용할 수 있는 필드 및 메소드와는 달리, 속성은 애플리케이션을 실행하기 전에 개발자가 컴포넌트를 사용자 지정할 수 있도록 합니다. 속성은 프로그래머의 작업을 단순화하는 Object Inspector에 표시할 수 있습니다. 즉, 객체를 생성하기 위해 여러 매개변수를 처리하는 대신 Kylix에서 Object Inspector의 값을 읽을 수 있습니다. 또한 Object Inspector는 속성 할당이 이루어진 후 즉시 유효성을 검사합니다.

- 속성은 구현 세부 사항을 숨길 수 있습니다. 예를 들면, 암호화된 형식으로 내부적으로 저장된 데이터가 속성의 값으로 암호가 풀려진 채 나타날 수 있습니다. 즉, 값이 간단한 숫자이더라도 컴포넌트는 데이터베이스에서 해당 값을 조회하거나 복잡한 계산을 수행하여 값에 도달합니다. 속성은 외견상 간단한 할당에 복잡한 효과를 덧붙일 수 있도록 해줍니다. 즉 필드에 대한 할당으로 보이는 것이 복잡한 처리를 구현하는 메소드에 대한 호출이 될 수 있습니다.
- 속성은 가상(virtual)일 수 있습니다. 따라서 애플리케이션 개발자에게 하나의 속성처럼 보이는 것이 다른 컴포넌트에서 다르게 구현될 수도 있습니다.

간단한 예가 모든 컨트롤의 *Top* 속성입니다. *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. 다른 타입의 속성은 Object Inspector에서 다르게 표시되는데, 이것으로 디자인 타임에 속성이 할당될 때 그 속성 할당의 유효성을 검사합니다.

**표 26.1** Object Inspector에서 속성이 표시되는 방법

| 속성 타입       | Object Inspector의 처리                                                                                                                                                                                             |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 일반 (simple) | 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 목록이 있습니다.                                                                                                       |
| 집합          | 집합 타입의 속성은 집합으로 나타납니다. 개발자는 속성을 더블 클릭하여 집합을 확장하고 각 요소를 부울 값(집합에 포함된 경우 <b>true</b> )으로 처리할 수 있습니다.                                                                                                               |
| 객체          | 자기 자신이 클래스인 속성으로 종종 컴포넌트의 등록 프로시저에 지정된 고유한 속성 편집기를 가집니다. 속성이 보유한 클래스에 자기 자신의 published 속성이 있는 경우, 개발자는 Object Inspector를 사용하여 더블 클릭으로 목록을 확장하여 이러한 속성을 포함시키고 개별적으로 편집합니다. 객체 속성은 <i>TPersistent</i> 의 자손이어야 합니다. |
| 인터페이스       | 해당 값이 <i>TComponent</i> 의 자손인 컴포넌트에 의해 구현된 인터페이스이면 인터페이스인 속성은 Object Inspector에 표시될 수 있습니다. 인터페이스 속성은 종종 고유한 속성 편집기를 가집니다.                                                                                       |
| 배열          | Object Inspector에서 배열 속성의 편집을 기본적으로 제공하지 않으므로 배열 속성은 자기 자신의 속성 편집기를 가져야 합니다. 컴포넌트를 등록할 때 속성 편집기를 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 `published`가 아니라 `protected` 또는 `public`입니다.

Object Inspector에서 디자인 타임에 `protected` 또는 `public` 속성을 사용할 수 있도록 하려면 해당 속성을 `published`로 재선언해야 합니다. 재선언은 상속된 속성에 대한 선언을 자손 클래스의 선언에 추가하는 것을 의미합니다.

`TWidgetControl`에서 컴포넌트를 파생시킬 경우, 새 컴포넌트는 `protected Bitmap` 속성을 상속합니다. 이 때 새 컴포넌트에서 `Bitmap`을 다시 선언하여 `public` 또는 `published`로 보호 수준을 변경할 수 있습니다.

다음 코드는 `Bitmap`을 `published`로 재선언하여 디자인 타임에 사용할 수 있도록 만듭니다.

```
type
  TSampleComponent = class(TWidgetControl)
    published
      property Bitmap;
    end;
```

속성을 재선언할 때는 속성 이름만 지정하고 아래의 "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 `public`으로 만들 수는 있지만 속성을 `protected`로 재선언하여 `public` 속성을 숨길 수는 없습니다.

## 속성 정의

이 단원에서는 새 속성을 선언하는 방법과 표준 컴포넌트에서 따라야 할 몇 가지 규칙을 설명합니다. 다음과 같은 항목이 들어 있습니다.

- 속성 선언
- 내부 데이터 저장소
- 직접 액세스
- 액세스 메소드
- 기본 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** 섹션에 선언된 속성은 디자인 타임에 Object Inspector에서 편집할 수 있습니다. **published** 속성의 값은 컴포넌트와 함께 폼 파일에 저장됩니다. **public** 섹션에 선언된 속성은 런타임에 사용할 수 있고, 프로그램 코드에서 읽거나 설정할 수 있습니다.

다음 예제는 *Count*라는 속성에 대한 전형적인 선언의 예입니다.

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;           { used for internal storage }
    procedure SetCount (Value: Integer); { write method }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

## 내부 데이터 저장소

속성 데이터를 저장하는 방법에는 제한이 없습니다. 하지만, 일반적으로 컴포넌트는 다음 규칙을 따릅니다.

- 속성 데이터는 클래스 필드에 저장됩니다.
- 속성 데이터를 저장하는 데 사용되는 필드는 **private**이므로 컴포넌트 내부에서만 액세스해야 합니다. 파생된 컴포넌트는 상속된 속성을 사용해야 하기 때문에 속성의 내부 데이터 저장소에 직접 액세스할 필요가 없습니다.
- 이러한 필드에 대한 식별자는 *F*로 시작하여 속성의 이름을 뒤에 붙여 구성합니다. 예를 들어, *TControl*에 정의된 *Width* 속성의 원시 데이터는 *FWidth*라고 하는 필드에 저장됩니다.

이러한 규칙의 토대가 되는 원칙은 특정 속성에 대한 구현 메소드만이 관련 데이터에 액세스할 수 있다는 것입니다. 만일 메소드나 다른 속성이 특정 속성과 관련된 데이터를 변경할 필요가 있는 경우, 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 액세스는 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 필드에서 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.

```
type
  TSampleComponent = class(TComponent)
  private
    FMyProperty: Boolean;           { internal storage is private }
  published                        { declare field to hold property value }
    property MyProperty: Boolean read FMyProperty write FMyProperty; { make property available at design time }
  end;
```

## 엑세스 메소드

속성 선언의 **read** 및 **write** 부분에서 필드 대신 액세스 메소드를 지정할 수 있습니다. 액세스 메소드는 **protected**이어야 하고 대개 **virtual**로 선언되기 때문에, 자손 컴포넌트는 속성의 구현을 오버라이드할 수 있습니다.

엑세스 메소드를 **public**으로 만들어서는 안 됩니다. 액세스 메소드를 **protected**로 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.

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day:Integer index 3 read GetDateElement write SetDateElement;
    property Month:Integer index 2 read GetDateElement write SetDateElement;
    property Year:Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index:Integer):Integer; { note the Index parameter }
    procedure SetDateElement(Index:Integer; Value:Integer);
  ;
```

날짜의 각 요소(연/월/일)가 정수이고 각 설정에 날짜 인코딩이 필요하기 때문에, 이 코드에서는 세 개의 속성 모두에 대해 **read** 및 **write** 메소드를 공유하여 중복을 방지합니다. 따라서 하나의 날짜 요소를 읽고 쓰는 데에는 오직 하나의 메소드만 필요합니다.

다음은 날짜 요소를 얻는 read 메소드입니다.

```
function TSampleCalendar.GetDateElement(Index:Integer):Integer;
var
  AYear, AMonth, ADay:Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);           { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

다음은 해당 날짜 요소를 설정하는 write 메소드입니다.

```
procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
var
  AYear, AMonth, ADay:Word;
begin
  if Value > 0 then                                { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);        { get current date elements }
    case Index of                                  { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);      { encode the modified date }
    Refresh;                                       { update the visible calendar }
  end;
end;
```

## read 메소드

속성에 대한 read 메소드는 아래에서 언급한 경우를 제외하고 매개변수를 사용하지 않고 속성과 동일한 타입의 값을 반환하는 함수입니다. 규칙에 따라 이 함수의 이름은 *Get* 으로 시작하여 속성의 이름을 뒤에 붙입니다. 예를 들어, *Count*라고 하는 속성에 대한 read 메소드는 *GetCount*입니다. read 메소드는 적절한 타입의 속성 값을 만들기 위해 필요한 경우 내부 저장소 데이터를 조작합니다.

예외적으로 배열 속성과 인덱스 지정자를 사용하는 속성(26-8 페이지의 "배열 속성 생성" 참조)에는 매개변수를 사용하지 않는 규칙이 적용되지 않으며, 두 속성 모두 인덱스 값을 매개변수로 전달합니다. (인덱스 지정자를 사용하여 여러 속성이 공유하는 단일 read 메소드를 만들 수도 있습니다. 인덱스 지정자에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.)

read 메소드를 선언하지 않으면 속성은 쓰기 전용이 됩니다. 일반적으로 쓰기 전용 속성은 거의 사용되지 않습니다.



## write 메소드

속성에 대한 write 메소드는 아래에서 언급한 경우를 제외하고 속성과 동일한 타입의 단일 매개변수를 취하는 프로시저입니다. 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 매개 변수로 전달합니다. 인덱스 지정자를 사용하여 여러 속성이 공유하는 단일 write 메소드를 만듭니다. (인덱스 지정자에 대한 자세한 내용은 *오브젝트 파스칼 랭귀지 안내서*를 참조하십시오.)

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 필드에 현재 값을 저장하는 *Count*라는 정수 속성에 대한 간단한 write 메소드입니다.

```
procedure TMyComponent.SetCount(Value:Integer);
begin
  if Value <> FCount then
    begin
      FCount := Value;
      Update;
    end;
end;
```

## 기본 속성 값

속성을 선언할 때 *기본값*을 지정할 수 있습니다. Kylix는 기본값을 사용하여 폼 파일에 속성을 저장할지 여부를 결정합니다. 사용자가 속성에 대한 기본값을 지정하지 않으면 Kylix는 항상 속성을 저장합니다.

속성에 대한 기본값을 지정하려면, 속성의 선언 또는 재선언에 **default** 지시어와 기본값을 차례대로 추가합니다. 예를 들면, 다음과 같습니다.

```
property Cool Boolean read GetCool write SetCool default True;
```

**참고** 기본값을 선언해도 속성이 기본값으로 설정되지는 않습니다. 컴포넌트의 생성자는 적절한 시기에 속성 값을 초기화해야 합니다. 그러나 객체가 항상 자신의 필드를 0으로 초기화하기 때문에 생성자가 반드시 정수 속성을 0으로, 문자열 속성을 Null로, 또는 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.

속성이 기본값을 갖지 않도록 지정하려면 속성의 선언 부분에 **nodefault** 지시어를 추가합니다. 예를 들면, 다음과 같습니다.

```
property FavoriteFlavor string nodefault;
```

속성을 처음 선언할 때 **nodefault**를 포함할 필요는 없습니다. 선언된 기본값이 없다는 것은 기본값이 존재하지 않음을 의미합니다.

다음 컴포넌트 선언에는 기본값이 *True*인 *IsTrue*라고 하는 단일 부울 속성이 포함되어 있습니다. 아래 선언에서 유닛의 **implementation** 섹션에는 속성을 초기화하는 생성자가 있습니다.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
:
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { call the inherited constructor }
  FIsTrue := True;                     { set the default value }
end;
```

## 배열 속성 생성

몇몇 속성들은 배열처럼 인덱스를 붙일 수 있습니다. 예를 들어, *TCustomTreeView*의 *Items* 속성은 트리에 있는 노드의 인덱스된 목록이므로 사용자는 속성을 트리 노드의 배열로 처리할 수 있습니다. *Items*는 방대한 데이터의 집합(트리에 있는 모든 노드의 집합)에 있는 특정 요소(노드)에 대한 자연스러운 액세스를 제공합니다.

배열 속성은 다음 사항을 제외하고는 다른 속성과 똑같이 선언됩니다.

- 선언에는 지정된 타입의 인덱스가 하나 이상 포함됩니다. 인덱스는 어떠한 타입도 될 수 있습니다.
- 속성 선언의 **read** 및 **write** 부분은 지정된 경우에 메소드여야 하고, 필드가 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 인덱스할 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.

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index:Integer):string;
  public
    property NumberName[Index:Integer]: string read GetNumberName;
  end;
:
function TDemoComponent.GetNumberName(Index:Integer):string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

## 하위 컴포넌트의 속성 생성

기본적으로 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 영구적인 객체, 즉 *TPersistent*의 자손이 될 수 있습니다. 속성 값으로 할당될 수 있는 별도의 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 생성된 컴포넌트를 반환하도록 영구적인 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 nil로 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.

```

procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
  if Value <> FTimer then
    begin
      if Value <> nil then
        begin
          if (FTimer <> nil and FTimer.Owner = self then
            FTimer.Free;
            FTimer := Value;
            FTimer.FreeNotification(self);
          end
        else { nil value }
        begin
          if FTimer.Owner <> self then
            {
              FTimer := TTimer.Create(self);
              FTimer.SetSubComponent(True);
              FTimer.FreeNotification(self);
            }
          end;
        end;
      end;
    end;
  end;

```

위의 속성 설정자가 속성 값으로 설정된 컴포넌트의 *FreeNotification* 메소드를 호출했다는 점에 유의합니다. 이 호출은 속성 값인 컴포넌트가 소멸되려는 순간에 공지를 보내도록 합니다. *Notification* 메소드를 호출하여 이러한 공지를 보냅니다. 다음과 같이 *Notification* 메소드를 오버라이드하여 이 호출을 처리합니다.

```

procedure TDemoComponent.Notification(AComponent:TComponent; Operation:TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = FTimer) then
    FTimer := nil;
  end;

```

## 인터페이스의 속성 생성

published 속성 값으로 객체를 사용하는 것과 마찬가지로 인터페이스를 사용할 수 있습니다. 그러나 컴포넌트가 해당 인터페이스의 구현으로부터 공지를 받는 메커니즘에는 차이가 있습니다. 이전 항목에서 속성 설정자는 속성 값으로 할당된 컴포넌트의 *FreeNotification* 메소드를 호출하였습니다. 이로 인해 속성 값인 컴포넌트가 해제될 때 컴포넌트는 자신을 업데이트할 수 있었습니다. 그러나 속성 값이 인터페이스인 경우에는 해당 인터페이스를 구현하는 컴포넌트에 액세스할 수 없습니다. 결과적으로 컴포넌트의 *FreeNotification* 메소드를 호출할 수 없습니다.

이러한 상황을 처리하려면 다음과 같이 컴포넌트의 *ReferenceInterface* 메소드를 호출합니다.

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

지정된 인터페이스로 *ReferenceInterface*를 호출하면 다른 컴포넌트의 *FreeNotification* 메소드를 호출한 것과 같습니다. 따라서 속성 설정자에서 *ReferenceInterface*를 호출한 후 다음과 같이 *Notification* 메소드를 오버라이드하여 인터페이스 구현에서 공지를 처리할 수 있습니다.

```
procedure TDemoComponent.Notification(AComponent:TComponent; Operation:TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.ImplementorOf(MyIntfProp)) then
    MyIntfProp := nil;
end;
```

*Notification* 코드가 private 필드 (*FIntfField*)가 아닌 *MyIntfProp* 속성에 *nil*을 할당한다는 점에 유의합니다. 이렇게 하면 속성 값이 이미 설정된 경우 만들어지는 공지 요청을 제거하기 위해 *ReferenceInterface*를 호출하는 속성 설정자를 *Notification*에서 호출하게 합니다. 인터페이스 속성에 대한 모든 할당은 속성 설정자를 통해 이루어져야 합니다.

## 속성 저장 및 로드

Kylix에서는 폼과 컴포넌트가 .xfm 파일에 저장됩니다. 폼 파일에는 폼과 컴포넌트의 속성이 포함됩니다. Kylix 개발자가 사용자가 작성한 컴포넌트를 폼에 추가할 경우, 사용자가 만든 컴포넌트는 저장될 때 컴포넌트의 속성을 폼 파일에 기록할 수 있는 능력을 가지고 있어야 합니다. 이와 마찬가지로 Kylix로 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 기본 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 함께 폼 속성의 목록으로 구성됩니다. 폼 자체를 포함하여 각 컴포넌트는 자신의 설명을 저장하고 로드합니다.

기본적으로 컴포넌트는 자신을 저장할 때 기본값과 다른 모든 `public` 및 `published` 속성의 값을 선언 순서대로 기록합니다. 로드할 때는 우선 모든 속성을 기본값으로 설정하면서 자기 자신을 생성하고 기본값이 아닌 저장된 속성 값을 읽습니다.

이 기본 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 지정할 수 있습니다.

## 기본값 지정

Kylix 컴포넌트는 속성 값이 기본값과 다를 경우에만 저장합니다. 기본값이 지정되지 않으면, Kylix는 속성에 기본값이 없다고 가정합니다. 따라서 컴포넌트는 그 값에 상관없이 항상 속성을 저장합니다.

속성에 기본값을 지정하려면 **default** 지시어와 새 기본값을 속성 선언의 끝 부분에 추가합니다.

또한 속성을 재선언할 때 기본값을 지정할 수도 있습니다. 실제로 속성을 재선언하는 이유 중 하나가 다른 기본 값을 지정하기 위한 데 있습니다.

### 참고

기본값을 지정한다고 해서 기본값이 자동으로 속성에 할당되는 것은 아닙니다. 따라서 컴포넌트 생성자가 필요한 값을 할당하도록 해야 합니다. 컴포넌트 생성자가 값을 설정하지 않은 속성은 0값, 즉 저장소 메모리가 0으로 설정될 때 가정하는 모든 값을 가집니다. 따라서 숫자 값에서는 0, 부울 값에서는 *False*, 그리고 포인터에서는 *nil*이 기본값이 됩니다. 확실하지 않은 경우에는, 생성자의 값을 할당합니다.

다음 코드는 *Align* 속성의 기본값을 지정하는 컴포넌트 선언과 기본값을 설정하는 컴포넌트 생성자의 구현을 보여 줍니다. 여기서 새 컴포넌트는 창의 상태 표시줄에 사용되는 표준 패널 컴포넌트의 특별한 경우이기 때문에 기본 정렬은 소유자의 맨 아래쪽에 있어야 합니다.

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override; { override to set new default }
  published
    property Align default alBottom; { redeclare with new default value }
  end;
  ...
  constructor TStatusBar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner); { perform inherited initialization }
  Align := alBottom; { assign new default value for Align }
end;
```

## 저장할 대상 결정

Kylix에서 각 컴포넌트의 속성 저장 여부를 제어할 수 있습니다. 기본적으로 클래스 선언의 `published` 부분에 있는 모든 속성이 저장됩니다. 지정한 속성을 저장하지 않도록 선택하거나 속성 저장 여부를 동적으로 결정하는 함수를 지정할 수 있습니다.

Kylix의 속성 저장 여부를 제어하려면 속성 선언에 **stored** 지시어를 추가한 다음 바로 뒤에 `True`, `False` 또는 부울 함수 이름을 추가합니다.

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.

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt:Boolean;
  public
    :
  published
    property Important: Integer stored True;      { always stored }
    property Unimportant: Integer stored False;  { never stored }
    property Sometimes: Integer stored StoreIt;  { storage depends on function value }
  end;
```

## 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 화면에 깜빡임을 야기하는 초기화에 대해 염려할 필요가 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

기본적으로 `published` 속성들만 컴포넌트와 함께 로드되고 저장됩니다. 하지만 `published`가 아닌 속성을 로드하고 저장하는 것도 가능합니다. 이는 Object Inspector에 나타나지 않는 영구적인 속성을 가질 수 있도록 합니다. 또한 속성 값이 너무 복잡하기 때문에 컴포넌트가 Kylix에서 읽거나 쓰는 방법을 알지 못하는 속성 값을 저장하고 로드할 수 있게 합니다. 예를 들어, *TStrings* 객체는 Kylix의 자동적 동작에 의존하여 자신이 나타내는 문자열을 저장 및 로드할 수 없고 다음과 같은 메커니즘을 사용해야만 합니다.

속성 값의 로드 및 저장 방법을 Kylix에게 알려주는 코드를 추가하여 `published`가 아닌 속성을 저장할 수 있습니다.

속성을 로드하고 저장하는 코드를 작성하려면 다음 단계를 따릅니다.

- 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 Kylix의 기본 제공 기능을 이용할 수 있습니다. Kylix가 저장 및 로드 방법을 알고 있는 타입으로 속성 값이 작성된 경우 이 방법을 사용합니다.
- *TStreamProc* 타입의 메소드를 두 개 만들어 각각 속성 값을 저장하고 로드하는 데 사용합니다. *TStreamProc*는 스트림을 인수로 가져오며, 이 스트림의 메소드를 사용하여 속성 값을 읽고 쓸 수 있습니다.

예를 들어, 런타임 시 생성되는 컴포넌트를 나타내는 속성을 고려해 봅시다. Kylix는 이 값을 기록하는 방법을 알고 있지만 폼 디자이너에서 컴포넌트를 만들었기 때문에 자동으로 기록하지는 않습니다. 스트리밍 시스템은 이미 컴포넌트들을 로드하고 저장할 수 있기 때문에 첫 번째 방법을 사용할 수 있습니다. 다음 메소드는 *MyCompProperty*라고 하는 속성의 값인 동적으로 생성된 컴포넌트를 로드하고 저장합니다.

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
    if Reader.ReadBoolean then
        MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
    Writer.WriteBoolean(MyCompProperty <> nil);
    if MyCompProperty <> nil then
        Writer.WriteComponent(MyCompProperty);
end;
```

## DefineProperties 메소드 오버라이드

속성 값을 저장하고 로드하기 위한 메소드를 만들었다면 컴포넌트의 *DefineProperties* 메소드를 오버라이드할 수 있습니다. Kylix는 컴포넌트를 저장하거나 로드할 때 이 메소드를 호출합니다. *DefineProperties* 메소드에서는 현재 파일러(filer)의 *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, 파일러(filer)의 *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. 상속할 수 있거나 기본값을 가진 값인 경우에는 그 값을 작성할 필요가 없습니다.



예를 들어, *TReaderProc* 타입의 *LoadCompProperty* 메소드와 *TWriterProc* 타입의 *StoreCompProperty* 메소드가 있는 경우, 다음과 같이 *DefineProperties*를 오버라이드합니다.

```

procedure TSampleComponent.DefineProperties(Filer: TFiler);
function DoWrite:Boolean;
begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
        if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
            Result := MyCompProperty <> nil
        else if MyCompProperty = nil or
            TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name
    then
        Result := True
    else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
        Result := MyCompProperty <> nil;
    end;
begin
    inherited; { allow base classes to define properties }
    Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;

```



## 이벤트 생성

이벤트는 사용자의 동작이나 포커스 변경 등과 같은 시스템에서 일어나는 사건과 그러한 사건에 응답하는 코드를 연결합니다. 응답 코드는 *이벤트 핸들러*이고, 거의 대부분 애플리케이션 개발자에 의해 작성됩니다. 이벤트를 통해 애플리케이션 개발자는 클래스 자체를 변경하지 않고도 컴포넌트의 행동을 사용자 지정할 수 있습니다. 이것을 *위임 (delegation)*이라고 합니다.

마우스 동작 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.

- 이벤트의 개념
- 표준 이벤트 구현
- 사용자 고유 이벤트 정의

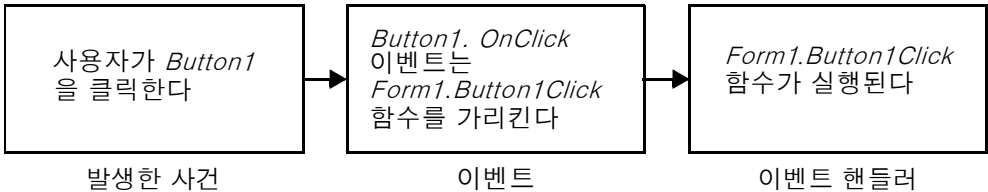
이벤트는 속성으로 구현되므로 컴포넌트의 이벤트를 만들거나 변경하기 전에 26장 "속성 생성"의 내용을 먼저 숙지해야 합니다.

### 이벤트의 개념

이벤트는 사건을 어떤 코드에 연결하는 메커니즘입니다. 보다 자세히 말하면, 이벤트는 특정 클래스 인스턴스의 메소드를 가리키는 메소드 포인터입니다.

애플리케이션 개발자의 관점에서 보면 이벤트는 특정 코드를 덧붙일 수 있는 *OnClick*과 같은 시스템의 사건에 관련된 이름일 뿐입니다. 예를 들어, *Button1*이라고 하는 푸시 버튼은 *OnClick* 이벤트를 가집니다. 기본적으로, *OnClick* 이벤트에 값을 할당하면 Kylix는 버튼을 포함하는 폼에서 *Button1Click*이라는 이벤트 핸들러를 생성하고 *OnClick*에 할당합니다. 클릭 이벤트가 버튼에서 발생하면 버튼은 *OnClick*에 할당된 메소드, 즉 이 경우 *Button1Click*을 호출합니다.

이벤트를 작성하려면 다음을 이해해야 합니다.



- 이벤트는 메소드 포인터입니다.
- 이벤트는 속성입니다.
- 이벤트 타입은 메소드 포인터 타입입니다
- 이벤트 핸들러 타입은 프로시저입니다.
- 이벤트 핸들러는 옵션입니다.

## 이벤트는 메소드 포인터입니다.

Kylix는 메소드 포인터를 사용하여 이벤트를 구현합니다. 메소드 포인터는 인스턴스 객체에서 특정 메소드를 가리키는 특별한 포인터 타입입니다. 컴포넌트 작성자는 메소드 포인터를 위치 표시자로 취급할 수 있습니다. 코드에서 이벤트가 발생했음을 탐지한 경우 해당 이벤트에 대해 사용자가 지정한 메소드가 있는 경우, 이를 호출합니다.

메소드 포인터는 다른 프로시저 타입처럼 작동하지만 객체에 대한 숨겨진 포인터를 유지합니다. 애플리케이션 개발자가 핸들러를 컴포넌트 이벤트에 할당하면 특정 이름을 갖는 메소드 뿐만 아니라 특정 인스턴스 객체의 메소드에 대해서도 할당이 적용됩니다. 그 객체는 일반적으로 컴포넌트를 포함하는 폼이지만 반드시 그럴 필요는 없습니다.

예를 들어, 모든 컨트롤은 클릭 이벤트를 처리하기 위해 *Click*이라는 동적 메소드를 상속합니다.

```
procedure Click; dynamic;
```

*Click*의 구현은 사용자의 클릭 이벤트 핸들러가 있을 경우 그것을 호출합니다. 사용자가 컨트롤의 *OnClick* 이벤트에 핸들러를 할당한 경우, 컨트롤을 클릭하면 해당 메소드가 호출됩니다. 핸들러를 할당하지 않은 경우에는 아무 일도 일어나지 않습니다.

## 이벤트는 속성입니다.

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` 클래스 필드를 사용합니다.

규칙에 따라 필드 이름은 *F* 문자가 앞에 오는 속성의 이름입니다. 예를 들면, *OnClick* 메소드의 포인터는 *TNotifyEvent* 타입의 *FOnClick*이라고 하는 필드에 저장되고, *OnClick* 이벤트 속성의 선언은 다음과 같습니다.

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;           { declare a field to hold the method pointer }
    :
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

*TNotifyEvent* 및 다른 이벤트 타입에 대한 내용은 다음 단원의 "이벤트 타입은 메소드 포인터 타입입니다."를 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 메소드 포인터 타입입니다.

이벤트는 이벤트 핸들러에 대한 포인터이기 때문에 이벤트 속성의 타입은 메소드 포인터 타입이어야 합니다. 마찬가지로, 이벤트 핸들러로 사용된 모든 코드는 적절하게 타입이 지정된 객체 메소드여야 합니다.

모든 이벤트 핸들러 메소드는 프로시저입니다. 주어진 타입의 이벤트와 호환 가능하려면 이벤트 핸들러 메소드는 동일한 순서로 동일한 매개변수의 수와 타입을 가지고 있어서 동일한 방식으로 전달되어야 합니다.

CLX는 모든 표준 이벤트에 대한 메소드 타입을 정의합니다. 사용자 고유의 이벤트를 만들 경우, 적절한 기존 타입을 사용하거나 사용자 고유 타입을 정의할 수 있습니다.

## 이벤트 핸들러 타입은 프로시저입니다.

컴파일러를 통해 함수인 메소드 포인터 타입을 선언할 수 있지만 이벤트를 처리하기 위해 그렇게 해서는 안됩니다. 빈 함수는 정의되지 않은 결과를 반환하기 때문에 함수로 작성된 비어 있는 이벤트 핸들러는 항상 유효할 수 없습니다. 이런 이유로 이벤트 및 그와 연결된 이벤트 핸들러는 모두 프로시저여야 합니다.

이벤트 핸들러가 함수일 수는 없지만 **var** 매개변수를 사용하여 애플리케이션 개발자의 코드로부터 여전히 정보를 얻을 수 있습니다. 이런 작업을 할 때 값을 변경하는 데 사용자 코드가 필요하지 않도록 하기 위해 핸들러를 호출하기 전에 매개변수에 유효한 값을 할당합니다.

이벤트 핸들러에 대한 **var** 매개변수 전달의 예는 *TKeyPressEvent* 타입의 *OnKeyPress* 이벤트입니다. *TKeyPressEvent*는 두 개의 매개변수를 정의합니다. 한 매개변수는 이벤트를 생성한 객체를 나타내고, 다른 매개변수는 눌러진 키를 나타냅니다.

**type**

```
TKeyPressEvent = procedure(Sender:TObject; var Key: Char) of object;
```

보통 *Key* 매개변수는 사용자가 누른 문자를 포함합니다. 그러나 어떤 상황에서는 컴포넌트 사용자가 문자를 변경하고자 할 수도 있습니다. 편집기의 모든 문자를 대문자로 바꾸려 할 때가 이러한 경우입니다. 이 경우, 사용자는 키 입력에 대한 다음과 같은 핸들러를 정의할 수 있습니다.

```
procedure TForm1.Edit1KeyPressed(Sender:TObject; var Key: Char);  
begin  
  Key := UpCase(Key);  
end;
```

또한 **var** 매개변수를 사용하여 사용자가 기본 처리를 오버라이드하도록 할 수 있습니다.

## 이벤트 핸들러는 옵션입니다.

이벤트를 생성할 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 27-9 페이지의 "이벤트 호출"에서 설명합니다.)

이벤트는 GUI 애플리케이션에서 거의 지속적으로 발생합니다. 비주얼 컴포넌트에 마우스 포인터를 이동하면 컴포넌트가 *OnMouseMove* 이벤트로 표면화되는 다양한 마우스 이동 이벤트를 발생시킵니다. 대부분의 경우, 개발자는 마우스 이동 이벤트를 처리하려고 하지 않는데 이것은 문제가 되지 않습니다. 생성된 컴포넌트는 자신의 이벤트에 대해 핸들러를 요구하지 않습니다.

CLX 컴포넌트는 이런 방식으로 작성된 이벤트를 가지므로 이벤트 핸들러의 오류 발생률이 최소화됩니다. 애플리케이션 코드에서 로직 오류는 막을 수 없지만, 애플리케이션 개발자가 유효하지 않은 데이터를 액세스하지 않도록 이벤트를 호출하기 전에 데이터 구조가 초기화되었는지는 확인할 수 있습니다.

## 표준 이벤트 구현

Kylix와 함께 제공되는 컨트롤은 가장 공통적인 항목에 대한 이벤트를 상속합니다. 이러한 이벤트를 *표준 이벤트*라고 합니다. 이러한 이벤트가 모두 컨트롤에 내장되어 있더라도 종종 **protected**이고, 이는 개발자가 핸들러를 추가할 수 없다는 것을 의미합니다. 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 보이기
- 표준 이벤트 처리 변경

## 표준 이벤트 식별

---

표준 이벤트에는 모든 컨트롤에 대해 정의된 표준 이벤트 및 widget 컨트롤에 대해서만 정의된 표준 이벤트의 두 가지 부류가 있습니다.

### 모든 컨트롤에 대한 표준 이벤트

가장 기본적인 이벤트는 *TControl* 클래스에서 정의됩니다. widget 기반, 그래픽 또는 사용자 지정 여부와 관계 없이 모든 컨트롤은 이러한 이벤트를 상속합니다. 다음 이벤트는 모든 컨트롤에서 사용 가능합니다.

|                                 |                    |                         |                       |
|---------------------------------|--------------------|-------------------------|-----------------------|
| <i>OnClick</i>                  | <i>OnDragDrop</i>  | <i>OnMouseMove</i>      | <i>OnMouseWheelUp</i> |
| <i>OnConstrained<br/>Resize</i> | <i>OnDragOver</i>  | <i>OnMouseUp</i>        | <i>OnResize</i>       |
| <i>OnContextPopup</i>           | <i>OnEndDrag</i>   | <i>OnMouseWheel</i>     | <i>OnStartDrag</i>    |
| <i>OnDbClick</i>                | <i>OnMouseDown</i> | <i>OnMouseWheelDown</i> |                       |

표준 이벤트에는 이벤트 이름에 해당하는 이름을 갖는 *TControl*에서 선언된 해당 protected 비주얼 메소드가 있습니다. 예를 들어, *OnClick* 이벤트는 *Click*이라는 메소드를 호출하고, *OnEndDrag* 이벤트는 *DoEndDrag*라는 메소드를 호출합니다.

### widget 기반 컨트롤에 대한 표준 이벤트

모든 컨트롤에 공통된 이벤트 외에, 표준 widget 기반 컨트롤(*TWidgetControl*의 자손)은 다음 이벤트들을 갖습니다.

|                |                  |                   |
|----------------|------------------|-------------------|
| <i>OnEnter</i> | <i>OnKeyDown</i> | <i>OnKeyPress</i> |
| <i>OnKeyUp</i> | <i>OnExit</i>    |                   |

*TControl*의 표준 이벤트와 마찬가지로, widget 기반 이벤트는 해당 메소드를 가집니다.

## 이벤트 보이기

---

*TControl* 및 *TWidgetControl*의 표준 이벤트 선언은 protected이고, 그에 해당하는 메소드 역시 protected입니다. 이러한 추상 클래스 중 하나에서 상속하고 런타임 시 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 표면화할 수 있습니다.

예를 들어, 디자인 타임에 *OnClick* 이벤트를 표면화하는 컴포넌트를 만들려면 컴포넌트의 클래스 선언에 다음을 추가합니다.

```
type
  TMyControl = class(TCustomControl)
  :
    published
      property OnClick;
  end;
```

## 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 메소드에 대한 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 순서가 중요합니다. 일반적으로 상속된 메소드를 먼저 호출하여 사용자 지정 이전에 애플리케이션 개발자의 이벤트 핸들러가 실행되도록 합니다(어떤 경우에는 사용자 지정이 실행되지 않도록 합니다). 그러나 상속된 메소드를 호출하기 전에 자신의 코드를 먼저 실행하고자 할 경우도 종종 있습니다. 예를 들어, 상속된 코드가 컴포넌트의 상태에 다소 종속되어 있고 코드로 그 상태를 변경하는 경우에는, 변경하고 나서 사용자 코드가 그 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.

```
procedure click override      { forward declaration }
:
:
procedure TMyControl.Click;
begin
  inherited Click;           { perform standard handling, including calling handler }
  ...                       { your customizations go here }
end;
```



## 사용자 고유 이벤트 정의

완전히 새로운 이벤트를 정의하는 것은 그다지 일반적이지 않습니다. 그러나 컴포넌트가 다른 컴포넌트의 행동과 완전히 다른 행동을 수행할 때 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 트리거하는 것이 무엇인지 알아야 합니다. 일부 이벤트의 경우에는 그 대답이 명백합니다. 예를 들어, 마우스 다운 이벤트는 사용자가 마우스 왼쪽 버튼을 눌렀을 때 발생합니다. 시스템에서 이벤트를 알리면 컴포넌트는 *MouseDown* 메소드를 호출하는데, 이 메소드는 사용자가 *OnMouseDown* 이벤트에 첨부한 코드를 호출합니다.

하지만 몇몇 이벤트는 특정 외부 사건에 대해 덜 명확하게 연결되어 있습니다. 예를 들어 스크롤 막대는 *OnChange* 이벤트를 가지는데, 이 이벤트는 키 입력, 마우스 클릭 및 다른 컨트롤에서의 변경 사항 등을 포함한 여러 가지 종류의 사건에 의해 발생합니다. 이벤트를 정의할 때 모든 적절한 사건들이 이벤트를 트리거하는 메소드를 호출하도록 해야 합니다.

### 두 종류의 이벤트

이벤트에는 메뉴 항목 선택 같은 widget 이벤트와 타이머나 키 입력 작업 같은 시스템 이벤트가 있습니다.

widget 이벤트는 사용자가 widget과 상호작용을 통해 생성되는 동작입니다. widget 이벤트는 CLX 컴포넌트로 전달된 신호를 생성하여 처리합니다. CLX 컴포넌트는 전달된 신호에 대해 설치된 관련 이벤트 핸들러를 가집니다. widget 이벤트로는 *OnChange* (편집 컨트롤에서 텍스트 변경), *OnHighlighted* (메뉴에서 메뉴 항목 선택), *OnReturn Pressed* (메모 컨트롤에서 *Enter* 누름) 등이 있습니다. 이러한 이벤트들은 특정 widget에 항상 연결되어 있고 이러한 widget 내에 정의됩니다.

시스템 이벤트는 운영 체제가 생성하는 이벤트입니다. 예를 들면, *OnTimer* 이벤트 (*Timer* 컴포넌트는 정해진 시간이 경과되면 이벤트를 발생시킴), *OnCreate* 이벤트 (컴포넌트 생성 중), *OnPaint* 이벤트 (다시 그려야 하는 컴포넌트 또는 widget), *OnKey Press* 이벤트 (키보드에서 키 입력) 등이 있습니다. 애플리케이션 프로그래머는 이 이벤트들이 원하는대로 처리되지 않을 경우, 직접 작업해야 합니다.

자신이 정의하는 이벤트의 트리거를 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 일반 공지 타입 또는 이벤트 특정 타입입니다. 핸들러에서 거꾸로 정보를 얻는 것도 가능합니다.

### 간단한 공지

공지 이벤트는 이벤트 발생 시기나 위치에 대한 자세한 정보 없이 특정 이벤트가 발생했다는 것만을 알려주는 이벤트입니다. 공지 이벤트는 단 하나의 매개변수, 이벤트를 보내는 사람만을 알리는 *TNotifyEvent* 타입을 사용합니다. 공지를 위한 모든 핸들러는 이벤트가 어떤 타입인지, 어떤 컴포넌트에서 이벤트가 발생했는지에 대해 "알고" 있습니다. 예를 들어, 클릭 이벤트는 공지에 해당합니다. 클릭 이벤트의 핸들러를 작성할 때에는 어떤 클릭이 발생했으며, 어떤 컴포넌트를 클릭했는지만 알 수 있습니다.

공지는 단방향 프로세스입니다. 피드백을 제공하거나 공지가 더 이상 처리되지 않도록 하는 메커니즘은 없습니다.

### 이벤트 특정 핸들러

어떤 경우에는 발생한 이벤트와 이벤트가 발생한 컴포넌트를 아는 것만으로 충분하지 않습니다. 예를 들어, 이벤트가 키 입력 이벤트인 경우 핸들러는 사용자가 누른 키를 알고자 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 핸들러는 프로시저이기 때문에 핸들러에서 역으로 정보를 전달하는 유일한 방법은 **var** 매개변수를 통하는 것입니다. 컴포넌트는 이 정보를 사용하여 사용자의 핸들러가 실행된 후 이벤트를 처리할 지 여부 및 그 방법을 결정합니다.

예를 들어, 모든 키 이벤트(*OnKeyDown*, *OnKeyUp* 및 *OnKeyPress*)는 *Key*라는 매개변수에서 눌러진 키 값을 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 결정했으면 이제 이벤트의 메소드 포인터와 속성을 선언할 수 있습니다. 사용자가 이벤트가 수행하는 작업을 잘 알 수 있도록 이벤트에 의미있고 서술적인 이름을 제공하도록 합니다. 다른 컴포넌트의 비슷한 속성들의 이름과 일관성을 유지하도록 합니다.

## "On" 으로 시작하는 이벤트 이름

Kylix에 있는 대부분의 이벤트 이름은 "On"으로 시작합니다. 이것은 단지 규칙일 뿐이며, 컴파일러에서 이것을 요구하는 것은 아닙니다. Object Inspector는 속성 타입을 보고 이벤트인지 알아 냅니다. 모든 메소드 포인터 속성은 이벤트로 간주되어 Events 페이지에 나타납니다.

개발자는 "On"으로 시작하는 이름의 알파벳 순서 목록에서 이벤트를 찾고자 합니다. 다른 종류의 이름을 사용하면 혼동을 일으킬 수 있습니다.

**참고** 이 규칙에 대한 주요 예외는 "Before" 및 "After"로 시작하는 일부 발생 사건 전후에 발생하는 다양한 이벤트입니다.

## 이벤트 호출

이벤트에 대한 호출을 중앙 집중화해야 합니다. 즉, 컴포넌트 안에 할당된 것이 있을 경우 애플리케이션의 이벤트 핸들러를 호출하고 기본 처리를 제공하는 가상 메소드를 만듭니다.

모든 이벤트 호출을 한 곳에 두면 사용자의 컴포넌트에서 새 컴포넌트를 파생시키는 누군가가 이벤트가 호출되는 장소를 찾기 위해 사용자의 코드를 검색하지 않고도, 하나의 메소드를 오버라이드함으로써 이벤트 처리를 사용자 지정할 수 있습니다.

이벤트를 호출할 때 더 고려해야 할 두 가지 사항은 다음과 같습니다.

- 비어 있는 핸들러도 유효해야 합니다.
- 사용자는 기본 처리를 오버라이드할 수 있습니다.

## 비어 있는 핸들러도 유효해야 합니다

비어 있는 이벤트 핸들러가 오류를 발생하는 상황을 만들거나 컴포넌트의 기능 수행이 애플리케이션의 이벤트 처리 코드의 특정 응답에 종속되어서는 안됩니다.

비어 있는 핸들러는 핸들러가 전혀 없는 것과 동일한 결과를 만들지 말아야 합니다. 따라서 애플리케이션의 이벤트 핸들러를 호출하기 위한 코드는 다음과 같아야 합니다.

```
if Assigned(OnClick) then OnClick(Self);
... { perform default handling }
```

다음과 같이 코드를 작성해서는 안됩니다.

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

## 사용자는 기본 처리를 오버라이드할 수 있습니다

어떤 종류의 이벤트에 대해서는 개발자가 기본 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 내서는 안된다는 규칙을 지키려는 것입니다. 비어 있는 핸들러는 참조에 의해 전달된 인수 값을 변경하지 않기 때문에 기본 처리는 비어 있는 핸들러를 호출한 후에 항상 발생합니다.

예를 들어, 키 입력 이벤트를 처리할 때, 애플리케이션 개발자는 **var** 매개변수 *Key*를 Null 문자(#0)로 설정하여 키 입력의 컴포넌트 기본 처리를 막을 수 있습니다. 이를 지원하는 로직은 다음과 같습니다.

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);  
if Key <> #0 then ... { perform default handling }
```

시스템 이벤트를 다루기 때문에 실제 코드는 위와 약간 다르지만 로직은 같습니다. 기본적으로 컴포넌트는 사용자 지정 핸들러를 호출한 다음 표준 처리를 수행합니다. 사용자 핸들러가 *Key*를 Null 문자로 설정하면 컴포넌트는 기본 처리를 생략합니다.

# 28

## 메소드 생성

컴포넌트 메소드는 클래스 구조에 내장된 프로시저 및 함수입니다. 컴포넌트 메소드를 통해 수행할 수 있는 작업에 특별한 제한은 없지만 CLX는 준수해야 할 특정 표준을 사용합니다. 다음 내용이 포함되어 있습니다.

- 종속성 피하기
- 메소드 이름 지정
- 메소드 보호
- 가상 메소드 생성
- 메소드 선언

일반적으로 컴포넌트가 많은 메소드를 포함해서는 안되며, 애플리케이션에서 호출해야 할 메소드 수를 최소화해야 합니다. 메소드로 구현하고 싶은 기능이 속성으로 보다 잘 캡슐화되는 경우가 많습니다. 속성은 Kylix 환경에 맞는 인터페이스를 제공하고 디자인 타임에 액세스할 수 있습니다.

### 종속성 피하기

---

컴포넌트 작성 시 개발자에게 주어지는 전제 조건을 최소화해야 합니다. 확장성을 최대화하려면 개발자가 원할 때 언제든지 컴포넌트에서 모든 원하는 것을 할 수 있어야 합니다. 그렇게 적용되지 않는 경우가 있겠지만 가능한 한 적용시키는 데 목표를 두어야 합니다.

다음 목록은 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

이러한 상황을 처리하는 가장 좋은 방법은 그 상황에서 빠져나올 수 있는 방법을 제공하는 것입니다. 예를 들어 어떤 한 메소드를 호출하는 경우 컴포넌트에서 다른 메소드를 호출하지 못하는 상태에 빠진다면, 또 다른 메소드를 작성해서 컴포넌트가 비정상적인 상태에 빠졌을 때 그 메소드를 호출해서 메인 코드 실행 전에 컴포넌트의 비정상적 상태를 바로잡도록 만들 수 있을 것입니다. 사용자가 유효하지 않은 메소드를 호출하는 경우에는 최소한 예외를 발생시켜야 합니다.

다시 말해, 사용자가 코드의 일부가 서로 종속되어 있는 상황을 만든 경우에 코드를 잘못 사용하여 문제가 일어나지 않도록 하는 책임은 *사용자*에게 있습니다. 예를 들어, 사용자가 종속성을 적용하지 않는 경우, 경고 메시지를 내보내는 것이 시스템 실패 보다는 나은 방법입니다.

## 메소드 이름 지정

---

Kylix는 메소드나 그 매개변수의 이름 지정에 대해 제한을 두지 않습니다. 단지 애플리케이션 개발자가 메소드를 더 쉽게 사용할 수 있도록 하는 몇 가지 규칙이 있습니다. 다양한 부류의 사람들이 컴포넌트를 사용할 수 있도록 컴포넌트 아키텍처를 구성해야 합니다.

사용자가 자신이나 소그룹의 프로그래머만이 사용하는 코드를 작성하는 데 익숙해져 있다면 이름 지정 방법에 대해 크게 염려하지 않았을 수도 있습니다. 컴포넌트는 사용자의 코드에 익숙하지 않은 사람들과 코드에 익숙하지 않은 사람들이 사용해야 하므로 메소드 이름을 명확하게 지정하는 것이 좋습니다.

다음은 메소드의 이름을 명확하게 지정하기 위한 몇 가지 제안입니다.

- 서술적인 이름을 지정합니다. 의미 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.

마지막으로 고려할 사항은, 메소드가 실제로 메소드여야 하는지 확인합니다. 메소드 이름에 동사가 포함되는 것이 좋습니다. 이름에 동사가 없는 메소드를 많이 만들었다면 그러한 메소드가 속성이어야 할지를 고려하십시오.

## 메소드 보호

---

필드, 메소드와 속성 등 클래스의 모든 부분은 25-4 페이지의 "엑세스 제어"에서 설명한대로 보호 또는 "가시성" 레벨을 가집니다. 메소드에 대한 적절한 가시성을 선택하는 것은 간단합니다.

컴포넌트에서 작성하는 대부분의 메소드는 **public** 또는 **protected**입니다. 메소드가 정말로 컴포넌트의 그 타입에 특정적이지 않다면 메소드를 **private**로 만들고 심지어 파생된 컴포넌트에서도 액세스할 수 없게 할 필요가 없습니다.

## public이어야 하는 메소드

애플리케이션 개발자가 호출하고자 하는 모든 메소드는 **public**으로 선언되어야 합니다. 대부분의 메소드 호출은 이벤트 핸들러에서 일어난다는 것을 기억해 두십시오. 따라서 메소드가 시스템 리소스를 잡아 두거나 운영 체제를 사용자에게 응답할 수 없는 상태로 가져가는 것을 피해야 합니다.

**참고** 생성자와 소멸자는 항상 **public**이어야 합니다.

## protected여야 하는 메소드

애플리케이션이 적절하지 않은 시기에 메소드를 호출하지 않도록 컴포넌트의 구현 메소드는 **protected**여야 합니다. 애플리케이션 코드에서 호출되어서는 안되지만 파생 클래스에서 호출되는 메소드를 가지고 있는 경우, 이 메소드를 **protected**로 선언합니다.

예를 들어, 이미 설정된 특정 데이터를 가지는 메소드가 있다고 가정합니다. 이 메소드를 **public**으로 만들 경우, 애플리케이션이 데이터를 설정하기 전에 메소드를 호출할 가능성이 있습니다. 그 반면, 이 메소드를 **protected**로 만들면 애플리케이션이 직접 메소드를 호출할 수 없습니다. 그럴 경우, **protected** 메소드를 호출하기 전에 데이터가 설정되도록 하는 다른 **public** 메소드를 설정할 수 있습니다.

속성 구현 메소드는 가장 **protected** 메소드로 선언되어야 합니다. 애플리케이션 개발자는 이와 같이 선언된 메소드를 통해 속성 구현을 오버라이드할 수 있으며, 그 기능을 확대하거나 완전히 교체할 수 있습니다. 그러한 속성은 완전히 다형적입니다. 메소드 액세스를 **protected**로 둬으로써 개발자가 잘못하여 메소드를 호출하여 부주의하게 속성을 수정하지 않도록 막을 수 있습니다.

## 추상 메소드

메소드는 경우에 따라 컴포넌트에서 **abstract**로 선언됩니다. CLX에서 추상 메소드는 *TCustomGrid*와 같이 일반적으로 "custom"으로 시작하는 이름의 클래스에서 발생합니다. 이러한 클래스는 오직 파생된 자손 클래스에 영향을 주기 위해 만들어지므로 그들 자신은 추상적입니다.

추상 멤버를 포함하는 클래스의 인스턴스 객체를 만들 수는 있지만 권장되지 않습니다. 추상 멤버를 호출하면 *EAbstractError* 예외가 발생합니다.

**abstract** 지시어는 자손 컴포넌트에서 표면화되고 정의되어야 하는 클래스의 부분을 나타내는 데 사용되며, 컴포넌트 작성자가 클래스의 실제 인스턴스를 생성하기 전에 자손 클래스에서 추상 멤버를 재선언하도록 합니다.

## 가상 메소드 생성

동일한 메소드 호출에 대한 응답으로 서로 다른 타입의 반응을 보이는 코드를 실행할 수 있도록 할 경우 메소드를 **virtual**로 만듭니다.

애플리케이션 개발자가 직접 사용할 수 있는 컴포넌트를 만들 경우, 모든 메소드를 비가상(nonvirtual)으로 만들 수 있습니다. 그 반면, 다른 컴포넌트가 파생될 추상 컴포넌트를 만들 경우, 추가된 메소드를 **virtual**로 만들 수 있습니다. 이렇게 하면 파생된 컴포넌트는 상속된 **가상** 메소드를 오버라이드할 수 있습니다.

## 메소드 선언

컴포넌트에서 메소드를 선언하는 것은 클래스 메소드를 선언하는 것과 동일합니다.

컴포넌트에서 새 메소드를 선언하려면 다음 두 가지 작업을 수행합니다.

- 선언을 컴포넌트의 객체 타입 선언에 추가합니다.
- 컴포넌트 유닛의 **implementation** 부분에 메소드를 구현합니다.

다음 코드는 두 개의 새 메소드 즉 protected 정적 메소드와 public 가상 메소드를 정의하는 컴포넌트를 보여줍니다.

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                                { declare protected static method }
  public
    function CalculateArea: Integer; virtual;           { declare public virtual method }
  end;
:
implementation
:
procedure TSampleComponent.MakeBigger;                  { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer;      { implement second method }
begin
  Result := Width * Height;
end;
```



## 컴포넌트에서 그래픽 사용

CLX에서는 자세한 수준까지 다루지 않아도 되는, 간단하면서도 완벽한 인터페이스 즉, 컴포넌트의 *Canvas* 속성을 제공합니다. 캔버스는 현재 사용하는 펜, 브러시 및 글꼴을 나타내는 속성을 가집니다.

캔버스에서 리소스를 관리하므로 사용자가 펜 핸들과 같은 것을 생성하고, 선택하고, 해제하는 것에 신경을 쓰지 않아도 됩니다. 단지 캔버스에 사용할 펜의 유형을 알려주면 나머지는 자동으로 처리됩니다.

Kylix에서 그래픽 리소스를 관리할 수 있도록 한 이점 중 하나는 나중에 사용하기 위해 리소스를 캐시로 저장할 수 있어서 반복적인 작업을 신속하게 수행할 수 있다는 것입니다. 예를 들어, 사용자가 특정한 유형의 도구를 반복적으로 생성, 사용 및 처분하는 프로그램 가지고 있는 경우, 해당 도구를 사용할 때마다 그러한 단계를 반복해야 합니다. Kylix에서는 그래픽 리소스를 캐시하기 때문에 반복적으로 사용하는 도구가 캐시에 아직 남아 있는 경우 Kylix에서는 도구를 다시 만들지 않고 기존 도구를 사용하면 됩니다.

이러한 예는 수백 개의 컨트롤을 가지고 수십 개의 폼을 여는 애플리케이션에서 볼 수 있습니다. 이러한 컨트롤 각각에 하나 이상의 *TFont* 속성이 있을 수 있습니다. 결과적으로 *TFont* 객체의 수백, 수천 개의 인스턴스가 생성되지만, 대부분의 애플리케이션은 두 개 또는 세 개의 글꼴 핸들만을 사용하는 것으로 모든 것이 해결됩니다.

### 캔버스 사용

캔버스 클래스는 개별 선, 모양 및 텍스트를 그리기 위한 수준 높은 함수 및 캔버스의 그리기 기능을 조작하기 위한 중간 속성을 포함하여 여러 수준에서 그래픽 컨트롤을 캡슐화합니다.

표 29.1은 캔버스의 기능을 요약합니다.

**표 29.1** 캔버스 기능 요약

| 수준 | 작동              | 툴                                                                                        |
|----|-----------------|------------------------------------------------------------------------------------------|
| 높음 | 선과 모양 그리기       | Methods such as <i>MoveTo</i> , <i>LineTo</i> , <i>Rectangle</i> 및 <i>Ellipse</i> 같은 메소드 |
|    | 텍스트 표시 및 측정     | <i>TextOut</i> , <i>TextHeight</i> , <i>TextWidth</i> 및 <i>TextRect</i> 메소드              |
|    | 영역 채우기          | <i>FillRect</i> 및 <i>FloodFill</i> 메소드                                                   |
| 중간 | 텍스트와 그래픽 사용자 지정 | <i>Pen</i> , <i>Brush</i> 및 <i>Font</i> 속성                                               |
|    | 이미지 복사 및 합치기    | <i>Draw</i> , <i>StretchDraw</i> 및 <i>CopyRect</i> 메소드, <i>CopyMode</i> 속성               |
| 낮음 | 저수준의 그래픽 함수 호출  | <i>Handle</i> 속성                                                                         |

캔버스 클래스 및 그 메소드와 속성에 대한 자세한 내용은 온라인 도움말을 참조하십시오.

## 그림 작업

Kylix에서 수행하는 대부분의 그래픽 작업은 컴포넌트와 폼의 캔버스에 직접 그리는 것으로 제한됩니다. Kylix에서는 또한 비트맵, 그리기 및 아이콘 같은 독립적인 그래픽 이미지 처리를 제공합니다.

Kylix에서 그림 작업을 하는 데에는 다음의 세 가지 중요한 사항이 있습니다.

- 그림, 그래픽 또는 캔버스 사용
- 그래픽 로드 및 저장

## 그림, 그래픽 또는 캔버스 사용

Kylix에는 그래픽을 다루는 세 가지 종류의 클래스가 있습니다.

- *canvas*는 폼, 그래픽, 컨트롤, 프린터 또는 비트맵의 드로잉 표면을 나타냅니다. 캔버스는 독립적인 클래스가 아닌 다른 클래스의 속성입니다.
- *graphic*은 비트맵, 아이콘 및 그리기 같은 파일이나 리소스에서 일반적으로 발견되는 유형의 그래픽 이미지를 나타냅니다. Kylix에서는 *TBitmap*, *TDrawing* 및 *TIcon* 클래스를 정의하는데 이 모든 클래스는 일반 *TGraphic*의 자손 클래스입니다. 또한 사용자는 자신의 그래픽 클래스를 정의할 수도 있습니다. 모든 그래픽의 최소 표준 인터페이스를 정의하여 *TGraphic*은 애플리케이션의 단순한 메커니즘을 제공하여 다양한 유형의 그래픽을 쉽게 이용할 수 있습니다.
- *picture*는 그래픽을 담아 두는 컨테이너로서 임의의 그래픽 클래스를 포함할 수 있습니다. 즉, *TPicture* 타입의 항목은 비트맵, 아이콘, 그리기 또는 사용자 정의 그래픽 타입을 포함할 수 있고, 애플리케이션은 그림 클래스를 통해 동일한 방법으로 모든 그래픽에 액세스할 수 있습니다. 예를 들어, 이미지 컨트롤은 *TPicture* 타입의 *Picture*라는 속성을 가지며, 이를 통해 컨트롤은 다양한 종류의 그래픽 이미지를 표시할 수 있습니다.

picture 클래스는 항상 graphic을 가지고 있고 graphic은 canvas를 가질 수 있다는 것을 명심하십시오. 일반적으로, 그림을 처리할 때 *TPicture*에 의해 드러나는 그래픽 클래스의 부분만을 사용합니다. 그래픽 클래스 자체의 특정 부분에 액세스해야 하는 경우, picture의 *Graphic* 속성을 참조할 수 있습니다.

## 그래픽 로드 및 저장

Kylix의 모든 그림과 그래픽은 파일에서 이미지를 로드하고 다시 그 파일(또는 다른 파일)에 저장할 수 있습니다. 언제든지 그림의 이미지를 로드하거나 저장할 수 있습니다.

이미지를 파일의 그림으로 로드하려면 *LoadFromFile* 메소드를 호출합니다.

그림의 이미지를 파일에 저장하려면 *SaveToFile* 메소드를 호출합니다.

*LoadFromFile*과 *SaveToFile*은 각각 유일한 매개변수로서 파일의 이름을 취합니다. *LoadFromFile*은 생성하고 로드할 그래픽 객체의 유형을 결정하기 위해 파일 이름의 확장자를 사용합니다. *SaveToFile*은 저장되는 그래픽 객체 타입에 적절한 파일 타입으로 저장합니다.

**참고** 이미지를 로드하여 Qt MIME 소스나 스트림 객체로 저장할 수도 있습니다.

예를 들어, 비트맵을 이미지 컨트롤의 그림으로 로드하려면 비트맵 파일의 이름을 그림의 *LoadFromFile* 메소드에 전달합니다.

```
procedure TForm1.LoadBitmapClick(Sender:TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.bmp');
end;
```

그림은 bmp를 비트맵 파일의 표준 확장자로 인식하므로 *TBitmap*으로 그래픽을 생성한 다음 그래픽의 *LoadFromFile* 메소드를 호출합니다. 그래픽이 비트맵이기 때문에 파일의 이미지를 비트맵으로 로드합니다.

## 오프스크린(off-screen) 비트맵

복잡한 그래픽 이미지를 그릴 때 그래픽 프로그래밍의 일반적인 방법은 오프스크린 비트맵을 생성하여 비트맵에 이미지를 그리고 비트맵으로부터 완성된 이미지를 최종 대상 온스크린(onscreen)에 복사하는 것입니다. 오프스크린 이미지를 사용하면 스크린에 직접 그리기를 반복함으로써 야기되는 화면 떨림을 줄일 수 있습니다.

리소스와 파일의 비트맵된 이미지를 나타내는 Kylix의 비트맵 클래스는 오프스크린 이미지로 작용할 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 생성할 때 온스크린(onscreen)에 나타나는 캔버스에 직접 그리는 것을 피하는 것이 좋습니다. 폼이나 컨트롤을 캔버스에 그리는 대신 비트맵 객체를 구성하고, 캔버스에 그린 다음 완성된 이미지를 온스크린 캔버스에 복사하면 됩니다.

오프스크린 비트맵의 가장 일반적인 사용은 그래픽 컨트롤의 *Paint* 메소드에서 이루어집니다. 임시 객체로 비트맵 객체는 **try..finally** 블록 안에서 보호되어야 합니다.

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;           { override the Paint method }
  end;

  procedure TFancyControl.Paint;
  var
    Bitmap: TBitmap;                    { temporary variable for the off-screen bitmap }
  begin
    Bitmap := TBitmap.Create;           { construct the bitmap object }
    try
      { draw on the bitmap }
      { copy the result into the control's canvas }
    finally
      Bitmap.Free;                      { destroy the bitmap object }
    end;
  end;
```

## 비트맵 이미지 복사

Kylix는 하나의 캔버스에서 다른 곳으로 이미지를 복사하는 네 가지 다른 방법을 제공합니다. 만들어 내고자 하는 효과에 따라서 다른 메소드를 호출합니다.

표 29.2는 캔버스 객체의 이미지 복사 메소드를 요약합니다.

표 29.2 이미지 복사 메소드

| 효과                               | 호출하는 메소드    |
|----------------------------------|-------------|
| 전체 그래픽 복사                        | Draw        |
| 그래픽 복사하고 크기 조정                   | StretchDraw |
| 캔버스 일부분 복사                       | CopyRect    |
| 그래픽을 반복적으로 복사해서 영역에 타일 모양으로 나타내기 | TiledDraw   |

## 변화에 대한 반응

캔버스와 그에 속한 객체(펜, 브러시 및 글꼴)들을 포함해서 모든 그래픽 객체에는 객체의 변화에 반응하기 위한 이벤트가 내장되어 있습니다. 이러한 이벤트를 사용하여 이미지를 다시 그려서 컴포넌트나 컴포넌트를 사용하는 애플리케이션이 변화에 반응하도록 할 수 있습니다.

그래픽 객체의 변화에 반응하는 것은 컴포넌트의 디자인 타임 인터페이스의 일부로 게시하는 경우 매우 중요합니다. 컴포넌트의 디자인 타임 외관이 Object Inspector에 설정된 속성과 일치하는지 확인하는 유일한 방법은 객체의 변화에 반응하는 것입니다.

그래픽 객체의 변화에 반응하려면 클래스의 *OnChange* 이벤트에 메소드를 할당합니다.

Tshape 컴포넌트는 형태를 그릴 때 사용하는 펜과 브러시를 나타내는 속성을 게시합니다. 컴포넌트의 생성자는 *OnChange* 이벤트에 펜이나 브러시가 변경되는 경우 컴포넌트가 자신의 이미지를 새로 고치도록 하는 메소드를 할당합니다.

```

type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender:TObject);
  end;
:
implementation
:
constructor TShape.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                 { construct the pen }
  FPen.OnChange := StyleChanged;       { assign method to OnChange event }
  FBrush := TBrush.Create;             { construct the brush }
  FBrush.OnChange := StyleChanged;     { assign method to OnChange event }
end;

procedure TShape.StyleChanged(Sender:TObject);
begin
  Invalidate();                       { erase and repaint the component }
end;

```



## 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
- 속성 편집기 추가
- 컴포넌트 편집기 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 11-5 페이지의 "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 *Register* 프로시저를 유닛에 추가합니다. *Register* 프로시저 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 프로시저 선언
- Register 프로시저 작성

## Register 프로시저 선언

---

등록에는 유닛에서 *Register*라는 이름의 단일 프로시저 작성 단계가 포함됩니다. *Register* 프로시저는 유닛의 인터페이스 부분에 나타나야 하며, 오브젝트 파스칼의 나머지 부분과 달리 이름의 대소문자를 구별합니다.

다음 코드는 새로운 컴포넌트를 생성하고 등록하는 단순한 유닛에 대한 개요를 보여줍니다.

```
unit MyBtns;
interface
type
    ...                                { declare your component types here }
procedure Register;                   { this must appear in the interface section }
implementation
    ...                                { component implementation goes here }

procedure Register;
begin
    ...                                { register the components }
end;
end.
```

*Register* 프로시저 내에서 컴포넌트 팔레트에 추가하고자 하는 각각의 컴포넌트에 대해 *RegisterComponents*를 호출합니다. 유닛이 여러 컴포넌트를 포함하고 있는 경우 모두 한꺼번에 등록할 수 있습니다.

## Register 프로시저 작성

---

컴포넌트를 포함하는 유닛의 *Register* 프로시저 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 한번씩 *RegisterComponents* 프로시저를 호출합니다. *RegisterComponents*에 관련된 세 가지 중요한 내용은 다음과 같습니다.

- 1 컴포넌트 지정
- 2 팔레트 페이지 지정
- 3 RegisterComponents 함수 사용



## 컴포넌트 지정

Register 프로시저 내에서, RegisterComponents에 대한 호출에서 구성할 수 있는 개방형 배열로 컴포넌트 이름을 전달합니다.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

다음 코드에 표시된대로, 여러 컴포넌트를 같은 페이지에 한꺼번에 등록하거나 다른 페이지에 등록할 수도 있습니다.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);           { two on this page... }
  RegisterComponents('Assorted', [TThird]);                         { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);{ ...and one on the Standard page }
end;
```

## 팔레트 페이지 지정

팔레트 페이지 이름은 문자열입니다. 팔레트 페이지에서 지정한 이름이 이미 존재하는 이름이 아닌 경우, 폼 디자이너는 그 이름을 갖는 새 페이지를 만듭니다. Kylix는 제품의 국제 버전이 페이지를 해당 모국어로 이름을 지정할 수 있도록 문자열 목록 리소스에 표준 페이지의 이름을 저장합니다. 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 프로시저 내에서 RegisterComponents를 호출하여 클래스 배열로 컴포넌트를 등록합니다. RegisterComponents는 두 매개변수 즉, 컴포넌트 팔레트 페이지의 이름과 컴포넌트 클래스의 배열을 취하는 함수입니다.

Page 매개변수를 컴포넌트가 나타나야 할 컴포넌트 팔레트의 페이지 이름으로 설정합니다. 명명된 페이지가 이미 있는 경우, 그 페이지에 컴포넌트가 추가됩니다. 명명된 페이지가 없는 경우, 폼 디자이너는 그 이름을 갖는 새 팔레트 페이지를 만듭니다.

사용자 지정 컴포넌트를 정의하는 유닛 중 하나의 Register 프로시저 구현에서 RegisterComponents를 호출합니다. 그런 다음 컴포넌트를 정의하는 유닛이 패키지로 컴파일되어 사용자 지정 컴포넌트가 컴포넌트 팔레트에 추가되기 전에 패키지를 설치해야 합니다.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);           {add to system page}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);      { new page}
end;
```

## 팔레트 비트맵 추가

모든 컴포넌트는 컴포넌트 팔레트에 컴포넌트를 나타내는 비트맵이 필요합니다. 비트맵을 따로 지정하지 않는 경우, Kylix는 기본 비트맵을 사용합니다.

각 비트맵은 가로 세로 24 픽셀이어야 합니다. Linux에는 비트맵 파일을 생성하는 도구가 많이 있습니다. 그 중 한 가지 방법은 Gimp와 같은 도구를 사용하여 이미지를 .ppm 파일로 만든 다음 ppmtobmp를 사용하여 .bmp 서식으로 변환하는 것입니다.

팔레트 비트맵은 디자인 타임에만 필요하므로 컴포넌트의 컴파일 유닛 안으로 비트맵을 컴파일하지 않습니다. 그 대신, 유닛과 동일한 이름을 갖지만 확장자(.dcr 동적 컴포넌트 리소스)를 갖는 리소스 파일에 팔레트 비트맵을 공급합니다.

설치할 컴포넌트가 있는 각 유닛에 팔레트 비트맵 리소스 파일을 제공하고, 각 리소스 파일에서 사용자가 등록하는 각 컴포넌트에 비트맵을 제공합니다. 비트맵 이미지는 컴포넌트와 동일한 이름을 가집니다. 예를 들어, *TMyControl*이라는 컴포넌트를 만드는 경우, *TMYCONTROL*이라는 비트맵이 들어 있는 .dcr 또는 .res 리소스 파일을 만들어야 합니다. 리소스 이름은 대소문자를 구별하지 않지만 규칙에 따라 보통 대문자로 되어 있습니다.

컴파일 파일이 있는 동일한 디렉토리에 리소스 파일을 보관하므로 Kylix는 컴포넌트 팔레트에 컴포넌트를 설치할 때 비트맵을 찾을 수 있습니다.

## 속성 편집기 추가

Object Inspector는 모든 타입의 속성에 기본 편집 기능을 제공합니다. 그러나 속성 편집기를 작성하고 등록하여 특정 속성의 대체 편집기를 제공할 수 있습니다. 작성하는 컴포넌트의 속성에만 적용되는 속성 편집기를 등록할 수 있고, 특정 타입의 모든 속성에 적용되는 편집기를 만들 수도 있습니다.

가장 간단한 수준에서는, 속성 편집기가 다음 두 가지 방법 중 하나 또는 모두에 의해 작동됩니다. 사용자가 현재 값을 텍스트 문자열로 편집하게 하여 표시하거나, 다른 종류를 편집할 수 있도록 대화 상자를 표시합니다.

속성 편집기를 작성하려면 다음 다섯 단계를 거쳐야 합니다.

- 1 속성 편집기 클래스 파생
- 2 텍스트로 속성 편집
- 3 전체 속성 편집
- 4 편집기 속성 지정
- 5 속성 편집기 등록

### 속성 편집기 클래스 파생

CLX는 여러 유형의 속성 편집기를 정의하는데 이 모든 편집기는 *TPropertyEditor*의 자손입니다. 속성 편집기를 만들 때 속성 편집기 클래스는 *TPropertyEditor*의 직계 자손이 되거나 표 30.1에 설명되어 있는 속성 편집기 클래스 중 하나의 간접 자손이 될 수 있습니다.

**참고** 속성 편집기는 반드시 *TBasePropertyEditor*의 자손이어야 하고, *IProperty* 인터페이스를 지원해야 합니다. 그러나 *TPropertyEditor*는 *IProperty* 인터페이스의 기본 구현을 제공합니다.

또한 CLX는 컴포넌트 이름같은 고유 속성에 의해 사용되는 매우 전문적인 속성 편집기를 정의합니다. 아래 목록에 나열된 속성 편집기는 사용자 정의 속성에 가장 유용합니다.

**표 30.1** 이미 정의된 속성 편집기 타입

| 타입                  | 편집되는 속성                                                                                                       |
|---------------------|---------------------------------------------------------------------------------------------------------------|
| TOrdinalProperty    | 정수, 문자 및 열거 타입 속성을 위한 모든 순서 타입 속성 편집기는 <i>TOrdinalProperty</i> 의 자손입니다.                                       |
| TIntegerProperty    | 이미 정의되고 사용자 지정된 부분범위를 포함한 모든 정수 타입.                                                                           |
| TCharProperty       | <i>Char</i> 타입 및 'A'..'Z' 같은 <i>Char</i> 의 부분범위.                                                              |
| TEnumProperty       | 열거 타입.                                                                                                        |
| TFloatProperty      | 모든 부동 소수점 숫자.                                                                                                 |
| TStringProperty     | 문자열.                                                                                                          |
| TSetElementProperty | 부울 값으로 표시되는 개별 요소 집합                                                                                          |
| TSetProperty        | 모든 집합. 집합은 직접 편집할 수는 없지만 집합 요소 속성의 목록으로 확장할 수 있습니다.                                                           |
| TClassProperty      | 클래스. 클래스 이름을 표시하고 클래스 속성을 확장할 수 있습니다.                                                                         |
| TMethodProperty     | 메소드 포인터들로, 가장 확실한 이벤트입니다.                                                                                     |
| TComponentProperty  | 동일한 품의 컴포넌트. 사용자는 컴포넌트 속성을 편집할 수 없지만 호환 가능한 타입의 특정 컴포넌트를 가리킬 수 있습니다.                                          |
| TColorProperty      | 컴포넌트 색상. 적용되는 경우 색상 상수를 표시하고, 그렇지 않은 경우 16진수 값을 표시합니다. 드롭다운 목록에 색상 상수가 포함되어 있습니다. 더블 클릭하면 색상 선택 대화 상자가 나타납니다. |
| TFontNameProperty   | 글꼴 이름. 드롭다운 목록에 현재 설치되어 있는 글꼴이 모두 표시됩니다.                                                                      |
| TFontProperty       | 글꼴. 글꼴 대화 상자에 액세스할 수 있을 뿐만 아니라 개별 글꼴 속성을 확장할 수 있습니다.                                                          |

다음 예제는 *TMyPropertyEditor*라는 간단한 속성 편집기의 선언을 보여줍니다.

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual:Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

## 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. *TPropertyEditor* 및 그 자손은 텍스트 표현과 실제 값 사이에서 변환하기 위해 오버라이드할 수 있는 가상 메소드를 제공합니다.

오버라이드해야 할 메소드는 *GetValue* 및 *SetValue* 입니다. 또한 속성 편집기는 *TPropertyEditor*에서 각기 다른 종류의 값을 지정하고 읽기 위한 메소드를 상속합니다. 이러한 메소드의 일부가 표 30.2에 나열되어 있습니다.

**표 30.2** 속성 값을 읽고 쓰기 위한 메소드

| 속성 타입        | Get 메소드        | Set 메소드        |
|--------------|----------------|----------------|
| 부동 소수점       | GetFloatValue  | SetFloatValue  |
| 메소드 포인터(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 편집기의 *GetValue* 메소드는 속성의 현재 값을 나타내는 문자열을 반환합니다. Object Inspector는 속성의 값 열에 이 문자열을 사용합니다. 기본적으로 *GetValue*는 "unknown"을 반환합니다.

속성에 대한 문자열 표현을 제공하려면 속성 편집기의 *GetValue* 메소드를 오버라이드합니다.

속성이 문자열 값이 아닌 경우, *GetValue*는 그 값을 문자열 표현으로 변환해야 합니다.

## 속성 값 설정

속성 편집기의 *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 편집기의 *SetValue* 메소드를 오버라이드합니다.

*SetValue*는 Set 메소드 중 하나를 호출하기 전에 값을 변환하고 검사해야 합니다.

다음은 *TIntegerProperty*의 *GetValue* 및 *SetValue* 메소드입니다. 정수는 순서 타입이므로 *GetValue*는 *GetOrdValue*를 호출하고 그 결과를 문자열로 변환합니다. *SetValue*는 문자열을 정수로 변환하고 일부 범위 점검을 수행한 다음 *SetOrdValue*를 호출합니다.

```
function TIntegerProperty.GetValue:string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // unsigned
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
  end;
  procedure TIntegerProperty.SetValue(const Value:string);
```

```

procedure Error(const Args: array of const);
begin
    raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
end;
var
    L: Int64;
begin
    L := StrToInt64(Value);
    with GetTypeData(GetPropType)^ do
        if OrdType = otULong then
            begin // unsigned compare and reporting needed
                if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
                    // bump up to Int64 to get past the %d in the format string
                    Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
            end
            else if (L < MinValue) or (L > MaxValue) then
                Error([MinValue, MaxValue]);
        SetOrdValue(L);
    end;
end;

```

여기서 설명한 예제보다는 다음 원칙이 더 중요합니다. *GetValue*는 값을 문자열로 변환하고, *SetValue*는 문자열을 변환하고 값을 검사한 다음 "Set" 메소드 중 하나를 호출합니다.

## 전체 속성 편집

사용자가 속성을 시각적으로 편집할 수 있는 대화 상자를 제공할 수도 있습니다. 속성 편집기는 자신이 클래스인 속성에 가장 많이 사용됩니다. 그 예로, *Font* 속성에서 사용하는 글꼴 대화 상자를 열어 글꼴의 속성을 모두 한꺼번에 선택할 수 있습니다.

전체 속성 편집기 대화 상자를 제공하려면 속성 편집기 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. 편집기는 타입별로 정의되므로 속성 값을 문자열로 변환할 필요가 없습니다. 편집기는 일반적으로 값을 "가져온 상태"대로 처리합니다.

사용자가 속성 옆에 있는 ".." 버튼을 클릭하거나 값 열을 더블 클릭하면 Object Inspector는 속성 편집기의 *Edit* 메소드를 호출합니다.

*Edit* 메소드의 구현 내에서 다음 단계를 따릅니다.

- 1 속성에 사용할 편집기를 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 편집기를 소멸시킵니다.

대부분의 컴포넌트에 있는 *Color* 속성은 표준 색상 대화 상자를 속성 편집기로 사용합니다. 다음은 대화 상자를 불러내서 결과를 사용하는 *TColorProperty*의 *Edit* 메소드입니다.

```

procedure TColorProperty.Edit;
var
    ColorDialog: TColorDialog;
begin
    ColorDialog := TColorDialog.Create(Application);           { construct the editor }
    try
        ColorDialog.Color := GetOrdValue;                     { use the existing value }
        if ColorDialog.Execute then                          { if the user OKs the dialog... }
            SetOrdValue(ColorDialog.Color);                    { ...use the result to set value }
        finally
            ColorDialog.Free;                                  { destroy the editor }
        end;
    end;

```

## 편집기 속성 지정

속성 편집기는 Object Inspector가 표시할 도구를 결정하는 데 사용할 수 있는 정보를 제공해야 합니다. 예를 들어, Object Inspector는 속성이 하위 속성을 가지고 있는지 또는 가능한 값의 목록을 표시할 수 있는지 여부를 알아야 합니다.

편집기 속성을 지정하려면 속성 편집기의 *GetAttributes* 메소드를 오버라이드합니다.

*GetAttributes*는 다음 값 중 하나 또는 전부를 포함할 수 있는 *TPropertyAttributes* 타입의 값 집합을 반환하는 메소드입니다.

**표 30.3** 속성 편집기 특성 플래그

| 플래그             | 관련 메소드        | 포함된 경우의 의미                                  |
|-----------------|---------------|---------------------------------------------|
| paValueList     | GetValues     | 편집기는 열거 타입 값의 목록을 제공할 수 있습니다.               |
| paSubProperties | GetProperties | 속성은 표시할 수 있는 하위 속성을 가집니다.                   |
| paDialog        | Edit          | 편집기는 전체 속성을 편집하기 위한 대화 상자를 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 목록을 정렬해야 합니다.           |
| paReadOnly      | 해당 없음         | 사용자는 속성 값을 수정할 수 없습니다.                      |

표 30.3 속성 편집기 특성 플래그 ( 계속 )

| 플래그                     | 관련 메소드            | 포함된 경우의 의미                                                                                                                      |
|-------------------------|-------------------|---------------------------------------------------------------------------------------------------------------------------------|
| paRevertable            | 해당 없음             | Object Inspector의 컨텍스트 메뉴에 있는 Inherited 메뉴 항목에 Revert를 사용 가능하게 합니다. 메뉴 항목은 현재 속성 값을 버리고 이전에 설정된 기본값이나 표준값으로 반환하도록 속성 편집기에 알립니다. |
| paFullWidthName         | 해당 없음             | 값을 표시할 필요가 없습니다. 그 대신, Object Inspector는 속성 이름에 전체 폭을 사용합니다.                                                                    |
| paVolatileSubProperties | GetProperties     | Object Inspector는 속성 값이 변경될 때 모든 하위 속성의 값을 다시 패치합니다.                                                                            |
| paReference             | GetComponentValue | 값은 기타 다른 것에 대한 참조입니다. paSubProperties와 연결되어 사용되는 경우, 참조된 객체는 이 속성에 대한 하위 속성으로 표시되어야 합니다.                                        |

*Color* 속성이 사용자가 Object Inspector에서 몇 가지 방법, 즉 속성값을 입력하고, 목록과 사용자 지정된 편집기로부터 선택하는 것을 허용한다는 의미에서 대부분의 다른 속성들보다 더 많은 유연성이 있습니다. 따라서, *TColorProperty*의 *GetAttributes* 메소드는 그 반환 값에 다음과 같은 몇 가지 속성을 포함합니다.

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
    Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

## 속성 편집기 등록

일단 속성 편집기를 생성하면 Kylix에 등록해야 합니다. 속성 편집기를 등록하려면 속성 타입을 특정 속성 편집기에 연결해야 합니다. 지정된 타입의 모든 속성을 갖는 편집기를 등록하거나 특정 컴포넌트 타입의 특정 속성만을 갖는 편집기를 등록할 수 있습니다.

속성 편집기를 등록하려면 *RegisterPropertyEditor* 프로시저를 호출합니다.

*RegisterPropertyEditor*는 다음의 네가지 매개변수를 취합니다.

- 편집할 속성의 타입에 대한 타입 정보 포인터.  
기본 제공 함수인 *TypeInfo*는 항상 *TypeInfo(TMyComponent)*와 같은 형태로 호출합니다.
- 이 편집기가 적용되는 컴포넌트 타입. 이 매개변수가 **nil**인 경우, 편집기는 지정된 타입의 모든 속성에 적용됩니다.
- 속성의 이름. 이 매개변수는 이전의 매개변수가 특정 컴포넌트 타입을 지정하는 경우에만 의미를 가집니다. 이 경우, 이 편집기가 적용되는 컴포넌트 타입의 특정 속성의 이름을 지정할 수 있습니다.
- 지정된 속성을 편집하는 데 사용할 속성 편집기의 타입.

다음은 컴포넌트 팔레트에 표준 컴포넌트의 편집기를 등록하는 프로시저에서 발췌된 것입니다.

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenuItem, '', TMenuItemProperty);
end;
```

이 프로시저의 세 문장은 *RegisterPropertyEditor*가 다르게 사용된 내용을 다룹니다.

- 첫 번째 문장은 가장 일반적입니다. 이 문장은 *TComponent*(또는 고유 편집기가 등록되어 있지 않은 *TComponent*의 자손) 타입의 모든 속성에 대해 *TComponentProperty* 속성 편집기를 등록합니다. 일반적으로, 속성 편집기를 등록할 때 특정 타입의 편집기를 만든 다음 그 타입의 모든 속성에 대해 그 편집기를 사용하고자 하므로 두 번째와 세 번째 매개변수는 각각 *nil*과 비어 있는 문자열입니다.
- 두 번째 문장은 가장 특정한 등록 유형입니다. 이 문장은 특정 컴포넌트 타입에서 특정 속성에 대한 편집기를 등록합니다. 이런 경우, 편집기는 모든 컴포넌트의 *TComponentName* 타입의 *Name* 속성에 대한 것입니다.
- 세 번째 문장은 첫 번째 문장보다 특정적이면서 두 번째 문장만큼 제한되어 있지 않습니다. 이 문장은 *TMenuItem* 타입의 컴포넌트에서 *TMenuItem* 타입의 모든 속성에 대한 편집기를 등록합니다.

## 속성 범주

IDE에서 Object Inspector를 통해 속성 범주별로 속성을 선택적으로 표시하고 숨길 수 있습니다. 새로운 사용자 지정 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.

속성을 등록하거나 또는 일부가 등록될 때 사용자 지정 컴포넌트의 모든 속성을 등록하는 것은 필수 사항이 아닙니다. 범주에 명시적으로 연결되어 있지 않은 속성은 *TMiscellaneousCategory* 범주에 포함됩니다. 이러한 속성은 기본 범주에 따라 Object Inspector에서 표시되거나 숨겨집니다.

속성을 등록하기 위한 이 두 가지 함수 외에도 *IsPropertyInCategory* 함수가 있습니다. 이 함수는 지역화 유틸리티를 생성하는 데 유용합니다. 이 유틸리티에서 지정된 속성 범주에 속성이 등록되어 있는지를 결정해야 합니다.

## 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 지정 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. 아래 예제에서는 *TMyButton* 사용자 지정 클래스의 컴포넌트의 "HelpContext"라는 속성을 'Help and Hints' 범주로 등록합니다.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

세 번째 변형은 이름이 아닌 타입을 사용하여 속성을 식별합니다. 아래 예제에서는 정수 타입에 기반하여 속성을 등록합니다.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

마지막 변형은 속성 타입과 이름을 사용하여 속성을 식별합니다. 아래 예제에서는 *TBitmap* 타입과 "Pattern" 이름을 기준으로 속성을 등록합니다.

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

유용한 속성 범주와 사용 설명은 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 지정 컴포넌트의 속성을 식별하기 위해 다른 기준 집합을 제공합니다.

첫 번째 변형을 사용하면 속성 이름이나 타입에 기반하여 속성을 식별할 수 있습니다. 목록은 상수의 배열로 전달됩니다. 아래 예제에서 "Text" 이름을 갖거나 *TEdit* 타입의 클래스에 속하는 속성은 'Localizable' 범주에 등록됩니다.

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

두 번째 변형을 사용하면 특정 컴포넌트에 속하는 속성 범주에 대해 등록된 속성을 제한할 수 있습니다. 등록할 속성의 목록에는 타입이 포함되지 않고 이름만 포함됩니다. 예를 들어, 다음 코드는 모든 컴포넌트에 대해 다수의 속성을 'Help and Hints' 범주에 등록합니다.

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint', 'ParentShowHint', 'ShowHint']);
```

세 번째 변형을 사용하면 사용자가 특정 타입을 갖는 속성 범주에 대해 등록된 속성을 제한할 수 있습니다. 두 번째 변형에서 등록할 속성의 목록에는 이름만 포함될 수 있습니다.

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

유용한 속성 범주와 사용 설명은 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 제공된 범주 중 하나에 속성을 등록할 수도 있습니다. 기본 제공 속성 범주는 표 30.4에 설명되어 있습니다.

표 30.4 속성 범주

| 범주                             | 용도                                                                                                                |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <i>Action</i>                  | <i>TEdit</i> 의 <i>Enabled</i> 및 <i>Hint</i> 속성은 런타임 작업에 관련된 속성입니다.                                                |
| <i>Database</i>                | <i>TQuery</i> 의 <i>DatabaseName</i> 및 <i>SQL</i> 속성은 데이터베이스 연산에 관련된 속성입니다.                                        |
| <i>Drag, Drop, and Docking</i> | <i>TImage</i> 의 <i>DragCursor</i> 및 <i>DragKind</i> 속성은 끌어서 가져다 놓기 및 도킹 연산에 관련된 속성입니다.                            |
| <i>Help and Hints</i>          | <i>TMemo</i> 의 <i>HelpContext</i> 및 <i>Hint</i> 속성은 온라인 도움말이나 힌트 사용에 관련된 속성입니다.                                   |
| <i>Layout</i>                  | <i>TDBEdit</i> 의 <i>Top</i> 및 <i>Left</i> 속성은 디자인 타임에 컨트롤의 시각적 표시와 관련된 속성입니다.                                     |
| <i>Legacy</i>                  | <i>TComboBox</i> 의 <i>Ctl3D</i> 및 <i>ParentCtl3D</i> 속성은 폐기된 연산에 관련된 속성입니다.                                       |
| <i>Linkage</i>                 | <i>TDataSource</i> 의 <i>DataSet</i> 속성은 한 컴포넌트를 다른 컴포넌트에 연결하는 것과 관련된 속성입니다.                                       |
| <i>Locale</i>                  | <i>TMainMenu</i> 의 <i>BiDiMode</i> 및 <i>ParentBiDiMode</i> 속성은 국제 로케일에 관련된 속성입니다.                                 |
| <i>Localizable</i>             | 애플리케이션의 지역화 버전에서 수정이 필요한 속성. <i>Caption</i> 과 같은 문자열 속성, 즉 컨트롤의 크기 및 위치를 결정하는 속성입니다.                              |
| <i>Visual</i>                  | <i>TScrollBar</i> 의 <i>Align</i> 및 <i>Visible</i> 속성은 런타임 시 컨트롤의 가시적인 표시와 관련된 속성입니다.                              |
| <i>Input</i>                   | <i>TEdit</i> 의 <i>Enabled</i> 및 <i>ReadOnly</i> 속성은 데이터베이스 연산에 관련될 필요는 없고 데이터 입력과 관련이 있는 속성입니다.                   |
| <i>Miscellaneous</i>           | <i>TSpeedButton</i> 의 <i>AllowAllUp</i> 및 <i>Name</i> 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', 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 편집기 추가

컴포넌트 편집기는 디자이너에서 컴포넌트를 더블 클릭했을 때 일어나는 작업을 결정하고 마우스 오른쪽 버튼으로 컴포넌트를 클릭하면 나타나는 컨텍스트 메뉴에 명령을 추가합니다. 또한 컴포넌트 편집기는 컴포넌트를 사용자 지정 형식으로 클립보드에 복사할 수 있습니다.

컴포넌트 편집기에 컴포넌트를 제공하지 않으면 Kylix는 기본 컴포넌트 편집기를 사용합니다. 기본 컴포넌트 편집기는 *TDefaultEditor* 클래스에 의해 구현됩니다. *TDefaultEditor*는 컴포넌트의 컨텍스트 메뉴에 새 항목을 추가하지 않습니다. 컴포넌트를 더블 클릭하면 *TDefaultEditor*는 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 지정 컴포넌트 편집기 추가 작업은 다음 단계로 이루어집니다.

- 컨텍스트 메뉴에 항목 추가
- 더블 클릭 행동 변경
- 클립보드 형식 추가
- 컴포넌트 편집기 등록

### 컨텍스트 메뉴에 항목 추가

컴포넌트에서 마우스 오른쪽 버튼을 클릭하면 컴포넌트 편집기의 *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 다음 문자가 컨텍스트 메뉴에 밑줄이 나타나며 메뉴 항목을 선택하기 위한 바로가기 키로 사용됩니다. 대화 상자가 나타날 경우에는, 명령 끝에 생략 기호(...)를 추가해야 합니다. *GetVerb*는 명령 인덱스를 나타내는 하나의 매개변수를 가집니다.

다음 코드는 *GetVerbCount* 및 *GetVerb* 메소드를 오버라이드하여 컨텍스트 메뉴에 두 개의 명령을 추가합니다.

```
function TMyEditor.GetVerbCount:Integer;
begin
    Result := 2;
end;

function TMyEditor.GetVerb(Index:Integer):String;
begin
    case Index of
        0: Result := "&DoThis ...";
        1: Result := "Do&That";
    end;
end;
```

**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. 편집기의 *Component* 속성을 사용하여 편집 중인 컴포넌트에 액세스할 수 있습니다.

예를 들어, 다음 *ExecuteVerb* 메소드는 이전 예제에서 *GetVerb* 메소드의 명령을 구현합니다.

```
procedure TMyEditor.ExecuteVerb(Index:Integer);
var
    MySpecialDialog: TMyDialog;
begin
    case Index of
        0: begin
            MyDialog := TMySpecialDialog.Create(Application);{ instantiate the editor }
            if MySpecialDialog.Execute then; { if the user OKs the dialog... }
                MyComponent.FThisProperty := MySpecialDialog.ReturnValue; { ...use the value }
                MySpecialDialog.Free; { destroy the editor }
            end;
        1: That; { call the That method }
    end;
end;
```

## 더블 클릭 행동 변경

컴포넌트를 더블 클릭하면 컴포넌트 편집기의 *Edit* 메소드가 호출됩니다. 기본적으로 *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 기본 행동을 변경할 수도 있습니다. 예를 들어, 다음과 같은 경우에 대체 행동을 제공할 수 있습니다.

- 컨텍스트 메뉴에 명령을 추가하지 않을 경우
- 컴포넌트를 더블 클릭했을 때 여러 개의 명령이 결합된 대화 상자를 표시하기를 원할 경우

컴포넌트를 더블 클릭한 경우, *Edit* 메소드를 오버라이드하여 새로운 행동을 지정합니다. 예를 들어, 다음 *Edit* 메소드는 컴포넌트를 더블 클릭할 때 글꼴 대화 상자를 나타냅니다.

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free;
  end;
end;
```

**참고** 컴포넌트를 더블 클릭해서 이벤트 핸들러의 코드 에디터를 표시하고자 하는 경우, *TComponentEditor* 대신 *TDefaultEditor*를 컴포넌트 편집기의 기본 클래스로 사용합니다. 그런 다음 *Edit* 메소드를 오버라이드하지 않고 보호되는 *TDefaultEditor.EditProperty* 메소드를 사용합니다. *EditProperty*는 컴포넌트의 이벤트 핸들러를 통해 검색하고 첫 번째로 찾은 이벤트를 나타냅니다. 특정 이벤트를 보기 위해서 이 작업을 변경할 수도 있습니다. 예를 들면, 다음과 같습니다.

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor:Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
  end;
```

## 클립보드 형식 추가

기본적으로 컴포넌트가 IDE에 선택되어 있는 동안 사용자가 Copy를 선택한 경우 해당 컴포넌트는 Kylix의 내부 형식으로 복사됩니다. 그런 다음 다른 폼이나 데이터 모듈로 붙여 넣을 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 Kylix IDE에서 무시되지만 다른 애플리케이션에 붙여넣을 수 있습니다.

```
procedure TMyComponent.Copy;
var
  Data:TMemoryStream;
begin
  Data := TMemoryStream.Create;
  try
    TImage(Component).Picture.SaveToStream(Data);
    Data.Position := 0;
    Clipboard.SetFormat('image/delphi.bitmap', Data);
  finally
    Data.Free;
  end;
end;
```

## 컴포넌트 편집기 등록

일단 컴포넌트 편집기가 정의되면 특정 컴포넌트 클래스를 사용하기 위해 등록될 수 있습니다. 등록된 컴포넌트 편집기가 폼 디자이너에서 선택되어 있는 경우, 해당 클래스의 각 컴포넌트에 대해 등록된 컴포넌트 편집기를 생성합니다.

컴포넌트 편집기와 컴포넌트 클래스를 연결하려면 *RegisterComponentEditor*를 호출합니다. *RegisterComponentEditor*는 편집기를 사용하는 컴포넌트 클래스의 이름과 정의된 컴포넌트 편집기 클래스의 이름을 취합니다. 예를 들어, 다음 문장은 *TMyComponent* 타입의 모든 컴포넌트를 사용하는 *TMyEditor*라는 컴포넌트 편집기 클래스를 등록합니다.

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

컴포넌트를 등록하는 *Register* 프로시저에서 *RegisterComponentEditor*를 호출합니다. 예를 들어, *TMyComponent*라는 새로운 컴포넌트 및 해당 컴포넌트 편집기인 *TMyEditor*가 모두 동일한 유닛에서 구현되는 경우, 다음 코드는 컴포넌트 및 컴포넌트 편집기에 연결된 것을 등록합니다.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent]);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

## 컴포넌트를 패키지로 컴파일

일단 컴포넌트가 등록되면 IDE에 설치하기 전에 패키지로 컴파일해야 합니다. 패키지에는 사용자 지정 속성 편집기 뿐만 아니라 하나 이상의 컴포넌트를 포함할 수 있습니다. 패키지에 대한 자세한 내용은 11장 "패키지와 컴포넌트 사용"을 참조하십시오.

패키지를 생성하고 컴파일하려면 11-6 페이지의 "패키지 생성 및 편집"을 참조하십시오. 패키지의 포함 목록에 사용자 지정 컴포넌트의 소스 코드 유닛을 포함시킵니다. 컴포넌트가 다른 패키지에 종속되어 있는 경우 해당 패키지를 요청 목록에 포함시킵니다.

IDE에 컴포넌트를 설치하려면 11-5 페이지의 "컴포넌트 패키지 설치"를 참조하십시오.

## 기존 컴포넌트 수정

컴포넌트를 만드는 가장 쉬운 방법은 사용자가 원하는 대부분의 기능을 가진 컴포넌트를 가져와서 필요에 맞게 변경하는 것입니다. 다음 예는 표준 메모 컴포넌트를 수정하여 기본값으로 줄 바꿈하지 않는 메모를 만드는 것을 보여 줍니다.

메모 컴포넌트의 *WordWrap* 속성 값은 *True*로 초기화됩니다. 줄 바꿈 하지 않는 메모를 자주 사용하는 경우 기본값으로 줄 바꿈하지 않는 새 메모 컴포넌트를 만들 수 있습니다.

**참고** *published* 속성을 수정하거나 기존 컴포넌트의 특정 이벤트 핸들러를 저장하려면 새로운 클래스를 만드는 것보다는 *컴포넌트 템플릿*을 사용하는 것이 때로는 더 쉽습니다.

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 등록한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 24-8 페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같이 컴포넌트를 만드는 일반적인 프로시저를 따릅니다.

- 컴포넌트의 유닛 *Memos*를 호출합니다.
- *TMemo*의 자손인 *TWrapMemo*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트 팔레트의 Samples 페이지에 *TWrapMemo*를 등록합니다.
- 결과 유닛은 다음과 같아야 합니다.

```
unit Memos;
interface
uses
```

```

SysUtils, Types, Classes, QGraphics, QControls, QForms, Qdialogs, QStdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.

```

새 컴포넌트를 지금 컴파일하고 설치하면 그 조상인 *TMemo*와 똑같은 역할을 수행합니다. 다음 단원에서, 이 컴포넌트를 간단히 변경하게 됩니다.

## 컴포넌트 클래스 수정

일단 새 컴포넌트 클래스를 생성하면 어떤 방식으로든 수정할 수 있습니다. 여기서는, 메모 컴포넌트의 한 속성의 초기값만을 변경합니다. 이를 위해서는 컴포넌트 클래스에 두 가지 작은 변경을 수행해야 합니다.

- 생성자 오버라이드
- 새 기본 속성 값 지정

생성자는 실제로 속성 값을 설정합니다. 기본값은 Kylix에 어떤 값이 폼 파일에 저장될 지를 알려줍니다. Kylix는 기본값과 다른 값을 저장하므로 두 단계를 모두 수행하는 것이 중요합니다.

## 생성자 오버라이드

디자인 타임에 컴포넌트가 폼에 위치해 있거나 런타임에 애플리케이션이 컴포넌트를 생성할 때, 컴포넌트의 생성자는 속성 값을 설정합니다. 컴포넌트가 폼 파일에서 로드될 때 애플리케이션은 디자인 타임에 변경된 모든 속성을 설정합니다.

**참고** 생성자를 오버라이드할 때 새 생성자는 상속된 생성자를 호출한 후 다른 작업을 수행해야 합니다. 자세한 내용은 25-8 페이지의 "메소드 오버라이드"를 참조하십시오.

이 예제에서 새 컴포넌트가 *WordWrap* 속성을 *False*로 설정하기 위해서 *TMemo*에서 상속된 생성자를 오버라이드해야 합니다. 이렇게 하려면 생성자 오버라이드를 forward 선언에 첨가한 다음 **implementation** 부분에서 새 생성자를 작성합니다.

```

type
  TWrapMemo = class(TMemo)
  public
    constructor Create(AOwner: TComponent); override; { constructors are always public }
  end;
  :
  constructor TWrapMemo.Create(AOwner: TComponent); { this goes after implementation }
begin
  inherited Create(AOwner); { ALWAYS do this first! }
  WordWrap := False; { set the new desired value }
end;

```



이제 컴포넌트 팔레트에 새 컴포넌트를 설치한 다음 폼에 추가할 수 있습니다. *WordWrap* 속성이 이제는 *False*로 초기화되어 있음을 유의하십시오.

초기 속성 값을 변경하려면 그 값을 기본값으로 지정해야 합니다. 생성자가 설정한 값이 특정 기본값과 일치하지 않는 경우, Kylix는 적절한 값을 저장하거나 복원할 수 없습니다.

## 새 기본 속성 값 지정

Kylix가 폼 파일에 폼에 대한 설명을 저장할 때, 기본값과 다른 속성 값만을 저장합니다. 기본값과 다른 값만을 저장하면 폼 파일 크기를 작게 유지할 수 있고 폼을 더 빨리 로드할 수 있습니다. 속성을 만들거나 기본값을 변경할 때 속성 선언이 새 기본값을 포함하도록 업데이트하는 것이 좋습니다. 폼 파일, 로드 및 기본값은 30장 "디자인 타임 시 컴포넌트 사용"에 자세하게 설명되어 있습니다.

속성의 기본값을 변경하려면 **default** 지시어가 뒤에 오는 속성 이름과 새 기본값을 재선언합니다. 전체 속성을 재선언할 필요 없이 이름과 기본값만을 재선언합니다.

줄 바꿈 메모 컴포넌트에서 객체 선언의 **published** 부분에 있는 *WordWrap* 속성을 *False* 기본 값으로 재선언합니다.

```
type
  TWrapMemo = class(TMemo)
  :
  published
    property WordWrap default False;
  end;
```

기본 속성 값 지정이 컴포넌트의 작업에 영향을 주지는 않습니다. 따라서 컴포넌트 생성자에서 값을 초기화해야 합니다. 기본값 재선언은 Kylix에게 폼 파일에 *WordWrap*을 언제 기록할지 확실히 알려주는 것입니다.



## 그래픽 컴포넌트 생성

그래픽 컨트롤은 간단한 유형의 컴포넌트입니다. 순수한 그래픽 컨트롤은 포커스를 받지 않으므로 고유 창을 갖거나 필요로 하지 않습니다. 사용자는 마우스로 컨트롤을 조작할 수 있지만 키보드 인터페이스는 없습니다.

이 장에 설명되어 있는 그래픽 컴포넌트는 컴포넌트 팔레트의 추가 페이지에 있는 도형 컴포넌트인 *TShape*입니다. 생성된 컴포넌트가 표준 도형 컴포넌트와 동일하더라도 중복 식별자를 피하기 위해서는 생성된 컴포넌트를 다른 이름으로 불러야 합니다. 이 장에서는 도형 컴포넌트를 *TSampleShape*라고 하며, 도형 컴포넌트의 생성에 관련된 모든 단계를 설명합니다.

- 컴포넌트 생성 및 등록
- 상속된 속성 게시
- 그래픽 기능 추가

### 컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 모두 동일한 방법으로 시작됩니다. 유닛을 만들고, 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 24-8 페이지의 "새 컴포넌트 생성"에 개괄적으로 설명되어 있습니다.

이 예제에서는 다음과 같은 특성을 사용하여 컴포넌트를 만드는 일반적인 절차를 따릅니다.

- 컴포넌트의 유닛의 이름을 *Shapes*라고 합니다.
- *TGraphicControl*의 자손인 *TSampleShape*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트 팔레트의 Sample 페이지에 *TSampleShape*를 등록합니다.

결과 유닛은 다음과 같습니다.

```
unit Shapes;
interface
uses SysUtils, Types, Classes, QGraphics, QControls, QForms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
```

## 상속된 속성 게시

일단 컴포넌트 타입을 과생시키면 새 컴포넌트에서 표면에 나타내고자 하는 조상 클래스의 protected 부분에서 선언된 속성과 이벤트를 결정할 수 있습니다. *TGraphicControl*은 컴포넌트가 컨트롤로 작용하도록 하는 모든 속성을 이미 게시하고 있으므로 사용자는 마우스 이벤트에 응답하고 드래그 앤 드롭 작업을 처리하는 기능을 게시해야 합니다.

상속된 속성 및 이벤트 게시에 대해서는 26-3 페이지의 "상속된 속성 게시"와 27-5 페이지의 "이벤트 보이기"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.

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;           { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;           { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;          { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

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 컨트롤 생성 시 수행할 두 가지 작업은 다음과 같습니다.

- 1 그릴 대상 결정
- 2 컴포넌트 이미지 그리기

또한, 도형 컨트롤 예제에서는 애플리케이션은 개발자가 디자인 타임에 도형 모양을 사용자 지정할 수 있는 속성을 추가합니다.

### 그릴 대상 결정

그래픽 컨트롤을 통해 사용자 입력을 포함한 동적 조건을 반영하도록 모양을 변경할 수 있습니다. 항상 똑같이 보이는 그래픽 컨트롤은 컴포넌트가 아닐 수 있습니다. 정적 이미지를 원할 경우, 컨트롤을 사용하는 대신 이미지를 가져올 수 있습니다.

일반적으로 그래픽 컨트롤의 모양은 그 속성의 조합에 따라 달라집니다. 예를 들어, 게이지(gauge) 컨트롤에는 모양과 방향, 진행 상황을 그래픽 뿐만 아니라 수치적으로 표시할지 여부를 결정하는 속성이 있습니다. 이와 마찬가지로, 도형 컨트롤에는 그려야 할 도형의 유형을 결정하는 속성이 있습니다.

그릴 도형을 결정하는 속성을 컨트롤에 제공하려면 *Shape*라는 속성을 추가합니다. 이 작업을 수행하려면 다음 단계가 필요합니다.

- 1 속성 타입 선언
- 2 속성 선언
- 3 구현 메소드 작성

속성 생성에 대해서는 26장 "속성 생성"에 자세히 설명되어 있습니다.

### 속성 타입 선언

사용자 정의 타입의 속성을 선언할 경우, 속성을 포함하는 클래스를 선언하기 전에 먼저 그 타입을 선언해야 합니다. 속성에 대한 사용자 정의 타입의 가장 일반적인 유형은 열거(enum) 타입입니다.

도형 컨트롤에서 컨트롤이 그릴 수 있는 각 도형의 종류에 대한 요소를 갖는 열거 타입이 필요합니다.

도형 컨트롤 클래스의 선언 위에 다음과 같은 타입 정의를 추가합니다.

```
type
    TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
        sstEllipse, sstCircle);
    TSampleShape = class(TGraphicControl) { this is already there }
```

이제 이 타입을 사용하여 클래스에서 새 속성을 선언할 수 있습니다.

## 속성 선언

속성을 선언할 때 속성의 데이터를 저장하려면 `private` 필드를 선언한 다음 속성 값을 읽고 쓰기 위한 메소드를 지정해야 합니다. 값을 읽기 위해 메소드를 사용할 필요가 없는 경우가 종종 있으며 대신 저장된 데이터를 가리킬 수 있습니다.

도형 컨트롤에서는 현재 도형을 유지하는 필드를 선언한 다음 그 필드를 읽는 속성을 선언하고 메소드 호출을 통해 그 필드에 씁니다.

다음 선언을 `TSampleShape`에 추가합니다.

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType; { field to hold property value }
    procedure SetShape(Value:TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

이제 남은 작업은 `SetShape`의 구현을 추가하는 것입니다.

## 구현 메소드 작성

속성 정의의 `read` 또는 `write` 부분이 저장된 속성 데이터에 직접 액세스하는 대신 메소드를 사용하는 경우에는, 그 메소드를 구현해야 합니다.

`SetShape` 메소드의 구현을 유닛의 **implementation** 부분에 추가합니다.

```
procedure TSampleShape.SetShape(Value:TSampleShapeType);
begin
  if FShape <> Value then { ignore if this isn't a change }
  begin
    FShape := Value; { store the new value }
    Invalidate; { force a repaint with the new shape }
  end;
end;
```

## 생성자 및 소멸자 오버라이드

기본 속성 값을 변경하고 컴포넌트의 소유된 클래스를 초기화하려면 상속된 생성자와 소멸자를 오버라이드합니다. 두 가지 경우 모두, 새로운 생성자나 소멸자에서 상속된 메소드를 항상 호출해야 합니다.

## 기본 속성 값 변경

그래픽 컨트롤의 기본 크기가 다소 작기 때문에 생성자에서 너비와 높이를 변경할 수 있습니다. 기본 속성 값 변경에 대해서는 31장 "기존 컴포넌트 수정"에 자세히 설명되어 있습니다.

이 예에서 도형 컨트롤의 크기는 각 변이 65픽셀인 사각형으로 설정됩니다.

컴포넌트 클래스의 선언에 오버라이드된 생성자를 추가합니다.

```
type
  TSampleShape = class(TGraphicControl)
  public
    constructor Create(AOwner: TComponent); override { constructors are always public }
  end;
```

- 1 새 기본값을 갖는 *높이*와 *너비* 속성을 재선언합니다.

```
type
  TSampleShape = class(TGraphicControl)
  :
  published
    property Height default 65;
    property Width default 65;
  end;
```

- 2 유닛의 **implementation** 부분에 새로운 생성자를 작성합니다.

```
constructor TSampleShape.Create(AOwner:TComponent);
begin
  inherited Create(AOwner); { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

## 펜과 브러시 게시

기본적으로, 캔버스에는 가는 검정색 펜과 굵은 흰색 브러시가 있습니다. 개발자가 펜과 브러시를 변경하도록 하려면 디자인 타임에 조작할 수 있는 클래스를 제공한 다음 색칠하는 도중에 캔버스에 클래스를 복사해야 합니다. 컴포넌트는 보조 펜이나 브러시를 소유하고 그 생성과 소멸을 책임지므로 때문에 보조 펜이나 브러시 같은 클래스를 *소유된 클래스*라고 합니다.

소유된 클래스 관리에는 다음과 같은 단계가 필요합니다.

- 1 클래스 필드 선언
- 2 액세스 속성 선언
- 3 소유된 클래스 초기화
- 4 소유된 클래스의 속성 설정

## 클래스 필드 선언

컴포넌트가 소유하는 각 클래스에는 컴포넌트에서 클래스에 대해 선언된 클래스 필드가 있어야 합니다. 클래스 필드는 컴포넌트가 자신을 소멸하기 전에 클래스를 소멸할 수 있도록 소유된 객체에 대한 포인터를 항상 가지도록 합니다. 일반적으로 컴포넌트는 자신의 생성자에서 소유된 객체를 초기화하고, 자신의 소멸자에서 소유된 객체를 소멸합니다.

소유된 객체의 필드는 거의 항상 `private`로 선언됩니다. 애플리케이션이나 다른 컴포넌트가 소유된 객체에 액세스해야 하는 경우에는 **published** 또는 **public** 속성을 선언할 수 있습니다.

도형 컨트롤에 펜과 브러시의 필드를 추가합니다.

```
type
  TSampleShape = class(TGraphicControl)
  private
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush;  { a field for the brush object }
    :
  end;
```

## 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 클래스 필드를 참조하지만, 쓰기 부분은 컴포넌트가 소유된 객체의 변경 내용에 반응할 수 있도록 하는 메소드를 호출합니다.

펜과 브러시 필드에 대한 액세스를 제공하는 속성을 도형 컨트롤에 추가합니다. 또한 펜이나 브러시의 변경 사항에 반응하기 위한 메소드를 선언합니다.

```
type
  TSampleShape = class(TGraphicControl)
  :
  private
    { these methods should be private }
    procedure SetBrush(Value:TBrush);
    procedure SetPen(Value:TPen);
  published
    { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

그런 다음, 유닛의 구현 부분에 *SetBrush* 및 *SetPen* 메소드를 작성합니다.

```
procedure TSampleShape.SetBrush(Value:TBrush);
begin
  FBrush.Assign(Value);          { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value:TPen);
begin
  FPen.Assign(Value);            { replace existing pen with parameter }
end;
```

다음과 같은 방법으로 *Value*의 값을 *FBrush*에 직접 할당하면

```
FBrush := Value;
```

*FBrush*의 내부 포인터를 덮어 쓰고, 메모리를 손실하며, 소유권 문제를 많이 일으킵니다.



## 소유된 클래스 초기화

클래스를 컴포넌트에 추가할 때 컴포넌트의 생성자는 사용자가 런타임에 객체와 상호 작용할 수 있도록 객체를 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.

```

constructor TSampleShape.Create(AOwner:TComponent);
begin
    inherited Create(AOwner);                { always call the inherited constructor }
    Width := 65;
    Height := 65;
    FPen := TPen.Create;                      { construct the pen }
    FBrush := TBrush.Create;                  { construct the brush }
end;

```

- 2 컴포넌트 클래스의 선언에 오버라이드된 소멸자를 추가합니다.

```

type
    TSampleShape = class(TGraphicControl)
    public                                { destructors are always public}
        constructor Create(AOwner:TComponent); override;
        destructor Destroy; override;        { remember override directive }
    end;

```

- 3 유닛의 **implementation** 부분에 새 생성자를 작성합니다.

```

destructor TSampleShape.Destroy;
begin
    FPen.Free;                               { destroy the pen object }
    FBrush.Free;                             { destroy the brush object }
    inherited Destroy;                       { always call the inherited destructor, too }
end;

```

## 소유된 클래스의 속성 설정

펜과 브러시 클래스를 처리하는 최종 단계에서는, 펜과 브러시의 변경으로 도형 컨트롤이 그 자체를 다시 그리는지 확인할 필요가 있습니다. 펜과 브러시 클래스는 모두 *OnChange* 이벤트를 가지므로 도형 컨트롤에서 메소드를 만들어서 두 *OnChange* 이벤트가 모두 메소드를 가리키도록 할 수 있습니다.

다음 메소드를 도형 컨트롤에 추가하고, 컴포넌트 생성자를 업데이트하여 펜과 브러시 이벤트를 새로운 메소드로 설정합니다.

```

type
    TSampleShape = class(TGraphicControl)
    published
        procedure StyleChanged(Sender:TObject);
    end;
    ...
implementation
    ...
    constructor TSampleShape.Create(AOwner:TComponent);

```

```

begin
  inherited Create(AOwner);           { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                { construct the pen }
  FPen.OnChange := StyleChanged;      { assign method to OnChange event }
  FBrush := TBrush.Create;            { construct the brush }
  FBrush.OnChange := StyleChanged;    { assign method to On Change }
end;

procedure TSampleShape.StyleChanged(Sender:TObject);
begin
  Invalidate;                         { erase and repaint the component }
end;

```

이렇게 변경하면, 컴포넌트는 다시 그려서 펜이나 브러시에 대한 변경 사항을 반영합니다.

## 컴포넌트 이미지 그리기

그래픽 컨트롤의 필수 요소는 그래픽 컨트롤이 화면에서 자신의 이미지를 그리는 방식입니다. *TGraphicControl* 추상 타입은 컨트롤에 원하는 이미지를 그리기 위해 오버라이드하는 *Paint* 라는 메소드를 정의합니다.

도형 컨트롤의 *Paint* 메소드는 몇 가지 해야 할 일들이 있습니다.

- 사용자가 선택한 펜과 브러시를 사용합니다.
- 선택한 도형을 사용합니다.
- 정사각형과 원이 동일한 너비와 높이를 사용하도록 조정합니다.

*Paint* 메소드 오버라이드에는 다음 두 단계의 작업이 필요합니다.

- 1 *Paint*를 컴포넌트 선언에 추가합니다.
  - 2 유닛의 **implementation** 부분에 *Paint* 메소드를 작성합니다.
- 도형 컨트롤의 경우, 클래스 선언에 다음 선언을 추가합니다.

```

type
  TSampleShape = class(TGraphicControl)
  :
  protected
    procedure Paint; override;
  :
  end;

```

그런 다음, 유닛의 **implementation** 부분에 메소드를 작성합니다.

```

procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                { copy the component's pen }
    Brush := FBrush;            { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);    { draw rectangles and squares }
    end;
  end;
end;

```

```

        sstRoundRect, sstRoundSquare:
            RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
        sstCircle, sstEllipse:
            Ellipse(0, 0, Width, Height); { draw round shapes }
    end;
end;
end;

```

*Paint*는 컨트롤이 자신의 이미지를 업데이트해야 할 때마다 호출됩니다. 컨트롤이 처음 나타나거나 컨트롤 앞의 창이 없어질 때 컨트롤이 그려집니다. 또한, *StyleChanged* 메소드가 하는 것과 같이 *Invalidate* 메소드를 호출하여 다시 그리도록 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. 그렇게 하려면 가장 짧은 면을 찾아 이미지를 가운데에 배치하는 코드를 작성해야 합니다.

다음은 정사각형과 타원에 맞게 조정하는 정교한 *Paint* 메소드입니다.

```

procedure TSampleShape.Paint;
var
    X, Y, W, H, S: Integer;
begin
    with Canvas do
        begin
            Pen := FPen; { copy the component's pen }
            Brush := FBrush; { copy the component's brush }
            W := Width; { use the component width }
            H := Height; { use the component height }
            if W < H then S := W else S := H; { save smallest for circles/squares }
            case FShape of { adjust height, width and position }
                sstRectangle, sstRoundRect, sstEllipse:
                    begin
                        X := 0; { origin is top-left for these shapes }
                        Y := 0;
                    end;
                sstSquare, sstRoundSquare, sstCircle:
                    begin
                        X := (W - S) div 2; { center these horizontally... }
                        Y := (H - S) div 2; { ...and vertically }
                        W := S; { use shortest dimension for width... }
                        H := S; { ...and for height }
                    end;
            end;
        end;
        case FShape of
            sstRectangle, sstSquare:
                Rectangle(X, Y, X + W, Y + H); { draw rectangles and squares }
            sstRoundRect, sstRoundSquare:
                RoundRect(X, Y, X + W, Y + H, S div 4, S div 4); { draw rounded shapes }
            sstCircle, sstEllipse:
                Ellipse(X, Y, X + W, Y + H); { draw round shapes }
        end;
    end;
end;
end;

```



## 그리드 사용자 지정

Kylix는 사용자 지정 컴포넌트의 기반으로 사용할 수 있는 추상 컴포넌트를 제공합니다. 추상 컴포넌트에서 가장 중요한 것은 그리드와 목록 상자입니다. 이 장에서는 기본 그리드 컴포넌트인 *TCustomGrid*에서 한 달을 출력하는 작은 달력을 만드는 방법을 보여줍니다.

달력을 만드는 데에는 다음과 같은 작업들이 포함됩니다.

- 컴포넌트 생성 및 등록
- 상속된 속성 게시
- 초기 값 변경
- 셀 크기 조정
- 셀 채우기
- 월과 연도 탐색
- 일(day) 탐색

결과 컴포넌트는 컴포넌트 팔레트의 Samples 페이지에 있는 *TCalendar* 컴포넌트와 유사합니다.

### 컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 다음과 동일한 방법으로 시작됩니다. 유닛을 만들고, 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 24-8 페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같은 특성으로 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- 컴포넌트의 *CalSamp* 유닛을 호출합니다.
- *TCustomGrid*의 자손인 *TSampleCalendar*라는 새 컴포넌트 타입을 파생시킵니다.
- 컴포넌트 팔레트의 Samples 페이지에 *TSampleCalendar*를 등록합니다.

결과 유닛은 다음과 같습니다.

```

unit CalSamp;
interface
uses
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QGrids;
type
    TSampleCalendar = class(TCustomGrid)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TSampleCalendar]);
end;
end.

```

달력 컴포넌트를 지금 설치한다면 Sample 페이지에 나타날 것입니다. 가장 기본적인 컨트롤 속성만 사용할 수 있습니다. 다음 단계는 보다 특화된 일부 속성을 달력 사용자가 사용할 수 있게 합니다.

**참고** 방금 컴파일한 샘플 달력 컴포넌트를 바로 설치할 수는 있지만 아직은 그 컴포넌트를 폼에 두지 마십시오. *TCustomGrid* 컴포넌트는 인스턴스 객체를 만들기 전에 재선언해야 하는 추상 *DrawCell* 메소드를 가집니다. *DrawCell* 메소드 오버라이드에 대해서는 이후에 나오는 "셀 채우기"에서 설명됩니다.

## 상속된 속성 게시

---

추상 그리드 컴포넌트인 *TCustomGrid*는 다수의 **protected** 속성을 제공합니다. 이러한 속성들 중에서 달력 컨트롤의 사용자가 사용할 수 있는 속성을 선택할 수 있습니다.

컴포넌트의 사용자가 상속된 **protected** 속성을 사용할 수 있도록 하려면 컴포넌트 선언의 **published** 부분의 속성을 재선언합니다.

달력 컨트롤에서는 여기에 표시된 속성과 이벤트를 게시합니다.

```

type
    TSampleCalendar = class(TCustomGrid)
    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;
end;

```

사용자가 어떤 그리드 선을 그릴지 선택할 수 있게 해주는 *Options* 속성 같은 게시할 수 있는 다른 많은 속성들도 있지만, 달력에 적용하지는 못 합니다.

컴포넌트 팔레트에 수정된 달력 컴포넌트를 설치하고 그 컴포넌트를 애플리케이션에서 사용하는 경우, 완전한 기능을 갖는 달력에서 다양한 속성과 이벤트를 사용할 수 있습니다. 이제 사용자는 사용자가 만든 디자인의 새 기능을 추가할 수 있습니다.

## 초기 값 변경

달력은 고정된 수의 행과 열을 갖는 그리드가 필수적이지만 모든 행이 날짜를 가지는 것은 아닙니다. 이런 이유에서 *ColCount* 및 *RowCount* 그리드 속성을 게시하지 않습니다. 달력의 사용자가 주 7일 이외에 다른 것을 표시하려고 하지는 않을 것이기 때문입니다. 그러나, 속성의 초기 값을 설정하여 일주일이 항상 7일이 되게 해야 합니다.

컴포넌트 속성의 초기 값을 변경하려면 생성자를 오버라이드하여 원하는 값을 설정합니다. 생성자는 가상이어야 합니다.

생성자를 컴포넌트 객체 선언의 **public** 부분에 추가한 다음 컴포넌트 유닛의 **implementation** 부분에 새 생성자를 작성해야 합니다. 새 생성자의 첫 번째 문장은 항상 상속된 생성자에 대한 호출이어야 합니다.

```

type
  TSampleCalendar = class(TCustomGrid
    public
      constructor Create(AOwner:TComponent); override;
      :
    end;
  :
  constructor TSampleCalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);           { call inherited constructor }
  ColCount := 7;                      { always seven days/week }
  RowCount := 7;                      { always six weeks plus the headings }
  FixedCols := 0;                     { no row labels }
  FixedRows := 1;                    { one row for day names }
  ScrollBars := ssNone;               { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected]; { disable range selection }
end;

```

이제 달력에는 열과 행이 각각 일곱 개씩 있으며, 맨 위의 행이 고정되어 있거나 스크롤되지 않습니다.

## 셀 크기 조정

사용자 또는 애플리케이션이 창이나 컨트롤의 크기를 변경하면 호출을 통해 *protected BoundsChanged* 메소드에 자동으로 공지됩니다. 컴포넌트는 셀 크기를 변경하여 이 공지에 대해 반응할 수 있으므로 컨트롤 경계 내에서 모두 맞춰집니다.

이 경우, 달력 컨트롤은 모든 셀이 새로운 크기로 보여질 수 있는 적합한 셀 크기를 계산하기 위해서 *BoundsChanged*를 오버라이드해야 합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  ...
  end;
  ...
  procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer; { temporary local variable }
begin
  GridLines := 6 * GridLineWidth; { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7; { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7; { and cell height }
  inherited; {now call the inherited method }
end;
```

이제 크기가 조정된 달력은 컨트롤에 맞는 최대 크기로 모든 셀을 표시합니다.

## 셀 채우기

그리드 컨트롤은 셀 단위로 내용을 입력합니다. 달력의 경우는 각 셀에 속하는 날짜를 계산하는 것을 의미합니다. 그리드 셀의 기본 그리기는 *DrawCell*이라는 가상 메소드에서 일어납니다.

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.

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow:Longint; ARect:TRect; AState: TGridDrawState);
  override;
  end;
  ...
  procedure TSampleCalendar.DrawCell(ACol, ARow:Longint; ARect:TRect;
    AState:TGridDrawState);
  begin
    if ARow = 0 then
      Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);{ use RTL strings }
  end;
```



## 날짜 추적

달력 컨트롤이 유용하게 쓰이도록 하려면, 사용자와 애플리케이션이 일, 월, 연도를 설정하기 위한 메커니즘이 있어야 합니다. CLX는 *TDateTime* 타입의 변수로 날짜와 시간을 저장합니다. *TDateTime*은 날짜와 시간의 인코딩된 숫자 표시로 프로그래밍 조작에는 유용하나 사람이 사용하기에는 불편합니다.

따라서 인코딩된 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
- 일, 월, 연도 액세스
- 일(day) 수 생성
- 오늘 날짜 선택

## 내부 날짜 저장

달력의 날짜를 저장하려면, 날짜를 담은 private 필드와 그 날짜에 액세스를 제공하는 런타임 전용 속성이 필요합니다.

내부 날짜를 달력에 추가하려면 다음과 같은 세 단계를 거쳐야 합니다.

- 1 날짜를 유지할 private 필드를 선언합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate:TDateTime;
  :
```

- 2 생성자에서 날짜 필드를 초기화합니다.

```
constructor TSampleCalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);      { this is already here }
  :                               { other initializations here }
  FDate := Date;                 { get current date from RTL }
end;
```

- 3 인코딩된 날짜에 액세스하도록 하는 런타임 속성을 선언합니다.

날짜를 설정하려면 컨트롤의 온스크린 이미지를 업데이트해야 하기 때문에 날짜를 설정하기 위한 메소드가 필요합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value:TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  :
  procedure TSampleCalendar.SetCalendarDate(Value:TDateTime);
  begin
    FDate := Value;              { set new date value }
    Refresh;                     { update the onscreen image }
  end;
```

## 일, 월, 연도 액세스

인코딩된 숫자로 표현된 날짜는 애플리케이션에는 적합하지만 사람들은 일, 월, 연도 표시를 사용하는 것을 더 선호합니다. 속성을 생성함으로써 인코딩된 저장된 요소들에 대하여 대체 액세스를 제공할 수 있습니다.

날짜의 각 요소(일, 월, 연도)가 정수이고, 각 요소를 설정하려면 설정 시 날짜를 인코딩해야 하기 때문에, 세 개의 속성 모두에서 구현 메소드를 공유하면 번번이 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.

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day:Integer index 3 read GetDateElement write SetDateElement;
    property Month:Integer index 2 read GetDateElement write SetDateElement;
    property Year:Integer index 1 read GetDateElement write SetDateElement;
  ;
```

- 2 구현 메소드를 선언하고 작성하여 인덱스 값마다 다른 요소를 설정합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index:Integer):Integer;      { note the Index parameter }
    procedure SetDateElement(Index:Integer; Value:Integer);
  ;
function TSampleCalendar.GetDateElement(Index:Integer):Integer;
var
  AYear, AMonth, ADay:Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);                { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
var
  AYear, AMonth, ADay:Word;
begin
  if Value > 0 then                                       { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);              { get current date elements }
    case Index of
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
```

```

        else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);           { encode the modified date }
    Refresh;                                           { update the visible calendar }
end;
end;

```

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 가지 고려해야 할 것이 있습니다. 월의 일 (day) 수는 어떤 달인지와 해당 연도가 윤년인지에 따라 달라집니다. 또한, 월은 월과 연도에 따라 다른 요일에서 시작됩니다. *IsLeapYear* 함수를 사용하여 해당 연도가 윤년인지 여부를 확인합니다. SysUtils 유닛의 *MonthDays* 배열을 사용하여 월의 일 수를 얻습니다.

윤년과 월별 일 수에 대한 정보가 있다면, 각 날짜가 입력되는 그리드 내에서의 위치를 계산할 수 있습니다. 첫째 주, 첫째 날을 기준으로 계산합니다.

각각의 셀에 대해 채울 달을 기준으로 오프셋(month-offset) 숫자가 필요하기 때문에 월이나 연도를 변경할 때마다 계산하고 참조하는 것이 좋습니다. 클래스 필드에 그 값을 저장하여 날짜가 변경될 때마다 그 필드를 업데이트할 수 있습니다.

다음과 같은 방법으로 적절한 셀에 날(day)들을 채웁니다.

- 1 필드 값을 업데이트하는 메소드와 객체에 달 오프셋 필드를 추가합니다.

```

type
    TSampleCalendar = class(TCustomGrid)
    private
        FMonthOffset: Integer;           { storage for the offset }
    protected
        procedure UpdateCalendar; virtual; { property for offset access }
    end;
    procedure TSampleCalendar.UpdateCalendar;
    var
        AYear, AMonth, ADay: Word;
        FirstDate: TDateTime;           { date of the first day of the month }
    begin
        if FDate <> 0 then               { only calculate offset if date is valid }
        begin
            DecodeDate(FDate, AYear, AMonth, ADay); { get elements of date }
            FirstDate := EncodeDate(AYear, AMonth, 1); { date of the first }
            FMonthOffset := 2 - DayOfWeek(FirstDate); { generate the offset into the grid }
        end;
        Refresh;                         { always repaint the control }
    end;

```

- 2 생성자와 날짜가 변경될 때마다 새로운 업데이트 메소드를 호출하는 *SetCalendarDate* 및 *SetDateElement* 메소드와 생성자에 다음 문장을 추가합니다.

```

constructor TSampleCalendar.Create(AOwner:TComponent);
begin
    inherited Create(AOwner);
    : { this is already here }
    UpdateCalendar; { other initializations here }
    { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value:TDateTime);
begin
    FDate := Value; { this was already here }
    UpdateCalendar; { this previously called Refresh }
end;

procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
begin
    :
    FDate := EncodeDate(AYear, AMonth, ADay); { encode the modified date }
    UpdateCalendar; { this previously called Refresh }
end;

```

- 3 셀의 행과 열 조합을 전달할 때 일 수를 반환하는 메소드를 달력에 추가합니다.

```

function TSampleCalendar.DayNum(ACol, ARow:Integer):Integer;
begin
    Result := FMonthOffset + ACol + (ARow - 1) * 7; { calculate day for this cell }
    if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
        Result := -1; { return -1 if invalid }
end;

```

*DayNum*의 선언을 컴포넌트의 타입 선언에 추가해야 합니다.

- 4 이제 날짜가 입력되는 위치를 계산할 수 있으므로 *DrawCell*을 업데이트하여 날짜를 입력할 수 있습니다.

```

procedure TCalendar.DrawCell(ACol, ARow:Longint; ARect:TRect; AState:TGridDrawState);
var
    TheText:string;
    TempDay:Integer;
begin
    if ARow = 0 then { if this is the header row ... }
        TheText := ShortDayNames[ACol + 1] { just use the day name }
    else begin
        TheText := ''; { blank cell is the default }
        TempDay := DayNum(ACol, ARow); { get number for this cell }
        if TempDay <> -1 then TheText := IntToStr(TempDay); { use the number if valid }
    end;
    with ARect, Canvas do
        TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
            Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;

```

달력 컴포넌트를 다시 설치하여 폼에 두면 현재 월에 대한 적절한 정보가 나타납니다.

## 오늘 날짜 선택

이제 달력 셀에 숫자를 가지게 되었기 때문에 오늘 날짜가 있는 셀로 선택 반전(selection highlighting)을 이동하는 것이 바람직합니다. 기본적으로 선택은 맨 위의 왼쪽 셀에서 시작하므로 달력을 처음 만들 때와 날짜를 변경할 때 *Row* 및 *Column* 속성을 설정해야 합니다.

오늘 날짜에 대한 선택을 설정하려면 *Refresh*를 호출하기 전에 *Row* 및 *Column*을 설정하는 *UpdateCalendar* 메소드를 변경합니다.

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    : { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
```

날짜를 디코딩하여 이전에 설정된 *ADay* 변수를 재사용하고 있다는 것에 유의합니다.

## 월과 연도 탐색

속성들은 특히 디자인 타임에 컴포넌트를 조작할 때 유용합니다. 그러나 때때로 하나 이상의 속성을 포함하는 일반적이거나 자연스러운 조작 타입이 있는 경우 이 조작들을 처리하기 위한 메소드를 제공하는 것이 바람직합니다. 이러한 자연스러운 조작의 한 예로 달력의 "다음 달(next month)" 기능이 있습니다. 월의 랩어라운드 및 연도의 증분 처리는 간단하면서도 컴포넌트를 사용하는 개발자에게는 매우 편리합니다.

일반적인 조작들을 메소드로 캡슐화하는 유일한 단점은 메소드가 런타임에만 사용될 수 있다는 것입니다. 하지만 그러한 조작들은 보통 반복적으로 수행될 때만 귀찮은 작업이 되고, 디자인 타임에는 아주 드문 일입니다.

달력에 대해, 다음과 이전의 월과 연도에 대한 다음의 네 가지 메소드를 추가합니다. 이러한 메소드마다 약간씩 다른 방법으로 *IncMonth* 함수를 사용하여 월 또는 연도의 증분에 의해 *CalendarDate*을 증가시키거나 감소시킵니다. *CalendarDate*을 증가시키거나 감소시킨 후 날짜 값을 디코딩하여 Year, Month 및 Day 속성에 해당하는 새 값을 입력합니다.

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
```

```

end;
procedure TCalendar.PrevYear;
begin
    DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
end;

```

클래스 선언에 새 메소드의 선언을 추가해야 합니다.

이제 달력 컴포넌트를 사용하는 애플리케이션을 생성할 때, 월 또는 연도를 검색하는 것을 쉽게 구현할 수 있습니다.

## 일(day) 탐색

---

주어진 월에서 일(day)을 탐색하는 두 가지 확실한 방법이 있습니다. 한 가지 방법은 화살표 키를 사용하는 것이고, 또다른 방법은 마우스 클릭에 반응하는 것입니다. 표준 그리드 컴포넌트는 두 가지 모두 클릭처럼 처리합니다. 즉, 화살표 이동은 인접한 셀을 클릭하는 것처럼 처리됩니다.

일(day) 탐색 과정은 다음과 같습니다.

- 선택 이동
- OnChange 이벤트 제공
- 빈 셀 배제

### 선택 이동

---

그리드의 상속된 동작은 화살표 키나 클릭에 대한 반응으로 선택 이동을 처리하지만, 선택된 일(day)을 변경할 경우 그 기본 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. 나중에 **override** 지시어를 포함하여 *TSampleCalendar*에 *Click*의 선언을 추가해야 합니다.

```

procedure TSampleCalendar.Click;
var
    TempDay: Integer;
begin
    inherited Click;                                { remember to call the inherited method! }
    TempDay := DayNum(Col, Row);                     { get the day number for the clicked cell }
    if TempDay <> -1 then Day := TempDay;              { change day if valid }
end;

```

## 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 필드 및 이벤트를 호출할 동적 메소드를 선언합니다.

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  :
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  :
```

- 2 *Change* 메소드를 작성합니다.

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

- 3 *SetCalendarDate* 및 *SetDateElement* 메소드의 끝에 *Change*를 호출하는 문장을 추가합니다.

```
procedure TSampleCalendar.SetCalendarDate(Value:TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                     { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index:Integer; Value:Integer);
begin
  :                                           { many statements setting element values }
  FDate := EncodeDate(AYear, AMonth, ADay);
  UpdateCalendar;
  Change;                                     { this is new }
end;
end;
```

이제 달력 컴포넌트를 사용하는 애플리케이션은 *OnChange* 이벤트에 핸들러를 첨부함으로써 컴포넌트 날짜의 변경 사항에 반응할 수 있습니다.

## 빈 셀 배제

---

달력을 작성할 때 사용자가 빈 셀을 선택할 수 있지만 날짜는 변경되지 않습니다. 그러므로 빈 셀의 선택을 하지 않도록 하는 것이 바람직합니다.

지정한 셀의 선택 가능 여부를 조정하려면 그리드의 *SelectCell* 메소드를 오버라이드합니다.

*SelectCell*은 열과 행을 매개변수로 취하고, 지정한 셀의 선택 가능 여부를 나타내는 부울 값을 반환합니다.

셀에 유효한 날짜가 없으면 *SelectCell*을 오버라이드하여 *False*를 반환할 수 있습니다.

```
function TSampleCalendar.SelectCell(ACol, ARow:Longint):Boolean;  
begin  
    if DayNum(ACol, ARow) = -1 then Result := False           { -1 indicates invalid date }  
    else Result := inherited SelectCell(ACol, ARow); { otherwise, use inherited value }  
end;
```

이제 사용자가 빈 셀을 클릭하거나 화살표 키에 의해 빈 셀로 이동하려고 하면 달력에 현재 셀이 선택된 상태로 남겨집니다.



## data aware 컨트롤 만들기

데이터베이스 연결을 사용할 때 *data aware* 컨트롤이 있으면 편리합니다. 즉, 애플리케이션에서 컨트롤과 데이터베이스의 일부를 연결할 수 있습니다. CLX에는 data-aware 레이블, 편집 상자, 목록 상자, 콤보 상자, 조회 컨트롤 및 그리드가 있습니다. 또한 사용자 고유의 data aware 컨트롤을 만들 수도 있습니다. data-aware 컨트롤 사용에 대한 자세한 내용은 15장 "데이터 컨트롤 사용"을 참조하십시오.

data awareness에는 여러 단계가 있습니다. 가장 간단한 단계는 읽기 전용 data awareness 또는 *데이터 찾아보기*인데 현재 데이터베이스의 상태를 반영하는 기능입니다. 보다 복잡한 단계는 편집 가능 data awareness 또는 *데이터 편집*인데 사용자가 컨트롤을 조작하여 데이터베이스의 값을 편집할 수 있습니다. 데이터베이스 관련 단계는 가장 간단한 경우인 단일 필드와의 연결부터 가장 복잡한 경우인 다중 레코드 컨트롤에 이르기까지 다양하다는 점에 유의합니다.

이 장에서는 먼저 데이터셋의 단일 필드와 연결하는 읽기 전용 컨트롤을 만드는 가장 간단한 방법을 설명합니다. 사용된 특정 컨트롤은 33장 "그리드 사용자 지정"에서 만들어진 *TSampleCalendar* 달력입니다.

그리고 나서 같은 장에서 새로운 데이터 찾아보기 컨트롤을 데이터 편집 컨트롤로 만드는 방법을 설명합니다.

### 데이터 찾아보기 컨트롤 생성

data-aware 달력 컨트롤 생성은 읽기 전용 컨트롤인지 또는 데이터셋의 원본으로 사용한 데이터를 변경할 수 있는 컨트롤인지 여부에 관계없이 다음과 같은 단계로 이루어 집니다.

- 컴포넌트 생성 및 등록
- 데이터 연결 추가
- 데이터 변경 내용에 응답

## 컴포넌트 생성 및 등록

모든 컴포넌트의 생성은 모두 동일한 방법으로 시작됩니다. 유닛을 만들고, 컴포넌트 클래스를 파생시켜 파생된 컴포넌트를 등록하고 컴파일한 다음 컴포넌트 팔레트에 설치합니다. 이러한 과정은 24-8 페이지의 "새 컴포넌트 생성"에서 개괄적으로 설명합니다.

이 예제에서는 다음과 같은 특성으로 컴포넌트를 생성하는 일반적인 절차를 따릅니다.

- 컴포넌트 유닛 *DBCal*을 호출합니다.
- *TSampleCalendar*의 자손인 *TDBCalendar*라는 새로운 컴포넌트 클래스를 파생시킵니다. 33장 "그리드 사용자 지정"에서는 *TSampleCalendar* 컴포넌트를 생성하는 방법을 보여 줍니다.
- 컴포넌트 팔레트의 Samples 페이지에 *TDBCalendar*를 등록합니다.

결과 유닛은 다음과 같을 것입니다.

```
unit DBCal;
interface
uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QGrids, Calendar;
type
  TDBCalendar = class(TSampleCalendar)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;
end.
```

이제 새 달력을 데이터 찾아보기로 만드는 작업을 진행할 수 있습니다.

## 읽기 전용 컨트롤 만들기

이 데이터 달력은 데이터에 대해 읽기 전용이므로 읽기 전용 컨트롤을 만들어 사용자들이 컨트롤 내에 변경을 만들지 않고 데이터베이스에 변경 내용이 반영되리라 생각하지 못하게 합니다.

읽기 전용 달력 만들기는 다음으로 이루어집니다.

- ReadOnly 속성 추가
- 필요한 업데이트 허용

### ReadOnly 속성 추가

*ReadOnly* 속성을 추가하면 디자인 타임에 읽기 전용 컨트롤을 만드는 방법을 제공합니다. 읽기 전용 속성이 *True*로 설정되어 있으면 선택 불가능한 컨트롤에 모든 셀을 만들 수 있습니다.

- 1 속성 선언과 값을 유지할 **private** 필드를 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean; { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override; { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
:
constructor TDBCcalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner); { always call the inherited constructor! }
  FReadOnly := True; { set the default value }
end;
```

- 2 컨트롤이 읽기 전용인 경우, 선택을 허용하지 않도록 *SelectCell* 메소드를 오버라이드합니다. *SelectCell* 사용에 대해서는 33-12 페이지의 "빈 셀 배제"에서 설명합니다.

```
function TDBCcalendar.SelectCell(ACol, ARow:Longint):Boolean;
begin
  if FReadOnly then Result := False { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow); { otherwise, use inherited method }
end;
```

*SelectCell*의 선언을 *TDBCcalendar*의 타입 선언에 추가하고 **override** 지시어를 첨부해야 합니다.

이제 달력을 폼에 추가하는 경우 컴포넌트는 클릭과 키 입력을 무시합니다. 또한 날짜 변경시 선택 위치를 업데이트하지 않습니다.

## 필요한 업데이트 허용

읽기 전용 달력은 *Row* 및 *Col* 속성을 포함하여 모든 종류의 변경에 대해 *SelectCell* 메소드를 사용합니다. *UpdateCalendar* 메소드는 날짜가 변경될 때마다 *Row* 및 *Col*을 설정하지만 *SelectCell*이 변경을 허용하지 않기 때문에, 날짜가 변경되더라도 선택은 그대로 유지됩니다.

변경에 대한 절대적 금지를 피하려면 내부 부울 플래그를 달력에 추가하고 그 플래그가 *True*로 설정되었을 때 변경을 허용하도록 합니다.

```
type
  TDBCcalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean; { private flag for internal use }
  protected
    function SelectCell(ACol, ARow:Longint):Boolean; override;
  public
    procedure UpdateCalendar; override; { remember the override directive }
  end;
  ...
function TDBCcalendar.SelectCell(ACol, ARow:Longint):Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False { allow select if updating }
  else Result := inherited SelectCell(ACol, ARow); { otherwise, use inherited method }
end;

procedure TDBCcalendar.UpdateCalendar;
begin
  FUpdating := True; { set flag to allow updates }
  try
    inherited UpdateCalendar; { update as usual }
  finally
    FUpdating := False; { always clear the flag }
  end;
end;
```

달력은 여전히 사용자 변경을 허용하지 않지만 이제 날짜 속성을 변경하여 해당 날짜에서 만들어진 변경을 제대로 반영합니다. 읽기 전용 달력 컨트롤이 있으므로 데이터 찾아보기 기능을 추가할 수 있습니다.

## 데이터 연결 추가

컨트롤과 데이터베이스 사이의 연결은 *data link*라는 클래스에 의해 처리됩니다. 컨트롤과 데이터베이스의 단일 필드를 연결하는 데이터 연결은 *TFieldDataLink*입니다. 전체 테이블에 대한 데이터 연결도 있습니다.

*data-aware* 컨트롤은 데이터 연결 클래스를 *소유(own)*합니다. 즉, 컨트롤이 데이터 연결을 생성하고 소멸하는 것을 담당합니다.

데이터 연결을 소유된 클래스로 만드는 데에는 다음 세 단계가 필요합니다.

- 1 클래스 필드 선언
- 2 액세스 속성 선언
- 3 데이터 연결 초기화

### 클래스 필드선언

32-5 페이지의 "클래스 필드 선언"에서 설명된 대로 컴포넌트는 그 소유된 클래스 각각에 대해 필드를 필요로 합니다. 이 경우, 달력은 그 데이터 연결에 대해 *TFieldDataLink* 타입의 필드를 필요로 합니다.

달력의 데이터 연결에 대한 필드를 선언합니다.

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
  :
  end;
```

애플리케이션을 컴파일하려면 DB 및 DBCtrls를 유닛의 **uses** 절에 추가해야 합니다.

### 액세스 속성 선언

모든 *data-aware* 컨트롤에는 데이터를 컨트롤에 제공하는 애플리케이션의 데이터 소스 클래스를 지정하는 *DataSource* 속성이 있습니다. 또한 단일 필드에 액세스하는 컨트롤은 데이터 소스에서 그 필드를 지정하기 위해서 *DataField* 속성이 필요합니다.

클래스의 *private* 데이터 멤버에 액세스할 수 있도록 하는 소유된 클래스의 액세스 속성과 달리, 이러한 액세스 속성은 소유된 클래스에 의해 유지 관리됩니다. 즉, 컨트롤과 그 데이터 연결이 동일한 데이터 소스와 필드를 공유할 수 있도록 하는 속성을 만듭니다.

*DataSource* 및 *DataField* 속성과 그 구현 메소드를 선언한 다음, 그 메소드를 데이터 연결 클래스의 해당 속성에 대한 "전달(*pass-through*)" 메소드로 작성합니다.

## 액세스 속성 선언의 예제

```

type
  TDBCcalendar = class(TSampleCalendar)
  private
    ...
    { implementation methods are private }
    ...
    function GetDataField: string;           { returns the name of the data field }
    function GetDataSource: TDataSource;      { returns reference to the data source }
    procedure SetDataField(const Value: string); { assigns name of data field }
    procedure SetDataSource(Value: TDataSource); { assigns new data source }
  published
    { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
:
function TDBCcalendar.GetDataField:string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCcalendar.GetDataSource:TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCcalendar.SetDataField(const Value:string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCcalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

```

달력과 데이터 연결 사이에 연결이 이루어졌으므로 한 가지 중요한 단계가 더 있습니다. 달력 컨트롤이 생성되었을 때 데이터 연결 클래스를 생성하고, 달력을 소멸하기 전에 데이터 연결을 소멸해야 합니다.

## 데이터 연결 초기화

data-aware 컨트롤은 그 데이터 연결에 대한 액세스가 필요하므로 데이터 연결 객체를 그 자신의 생성자 일부로 생성하고 자신이 소멸되기 전에 데이터 연결 객체를 소멸해야 합니다.

달력의 *Create* 및 *Destroy* 메소드를 오버라이드하여 데이터 연결 객체를 각각 생성하고 소멸합니다.

```

type
  TDBCcalendar = class(TSampleCalendar)
  public
    { constructors and destructors are always public }
    constructor Create(AOwner:TComponent); override;
    destructor Destroy; override;
  :
  end;
:

```

```

constructor TDBCcalendar.Create(AOwner:TComponent);
begin
    inherited Create(AOwner);           { always call the inherited constructor first }
    FDataLink := TFieldDataLink.Create; { construct the datalink object }
    FDataLink.Control := self;         {let the datalink know about the calendar }
    FReadOnly := True;                 { this is already here }
end;

destructor TDBCcalendar.Destroy;
begin
    FDataLink.Free;                    { always destroy owned objects first... }
    inherited Destroy;                 { ...then call inherited destructor }
end;

```

이제 완벽한 데이터 연결이 갖추어졌지만 연결된 필드에서 읽어야 하는 데이터를 컨트롤에 아직 알리지 않았습니다. 다음 단원에서 이 작업을 수행하는 방법을 설명합니다.

## 데이터 변경 내용에 응답

컨트롤에 데이터 소스와 데이터 필드를 지정할 데이터 연결 및 속성을 갖추면 다른 레코드로의 이동이나 그 필드의 변경에 기인하여 그 필드에서 데이터 변경에 응답해야 합니다.

데이터 연결에는 모두 *OnChange*라는 이벤트가 있습니다. 데이터 소스가 데이터의 변경을 표시하면 데이터 연결 객체는 *OnChange* 이벤트에 첨부된 이벤트 핸들러를 호출합니다.

데이터 변경에 대한 응답으로 컨트롤을 업데이트하기 위해 데이터 연결의 *OnChange* 이벤트에 핸들러를 첨부합니다.

이 경우, 메소드를 달력에 추가한 다음 그 메소드를 데이터 연결의 *OnChange*에 대한 핸들러로 지정합니다.

*OnChange* 메소드를 선언하고 구현한 다음 그 메소드를 생성자에서 데이터 연결의 *OnChange* 이벤트에 할당합니다. 소멸자에서 객체를 소멸하기 전에 *OnChange* 핸들러를 분리합니다.

```

type
    TDBCcalendar = class(TSampleCalendar)
    private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject); { must have proper parameters for event }
    end;
;

constructor TDBCcalendar.Create(AOwner:TComponent);
begin
    inherited Create(AOwner);           { always call the inherited constructor first }
    FReadOnly := True;                 { this is already here }
    FDataLink := TFieldDataLink.Create; { construct the datalink object }
    FDataLink.OnDataChange := DataChange; { attach handler to event }
end;

destructor TDBCcalendar.Destroy;
begin
    FDataLink.OnDataChange := nil;     { detach handler before destroying object }
end;

```

```

FDataLink.Free;
inherited Destroy;
end;

procedure TDBCalendar.DataChange(Sender:TObject);
begin
    if FDataLink.Field = nil then                { if there is no field assigned... }
        CalendarDate := 0                        { ...set to invalid date }
    else CalendarDate := FDataLink.Field.AsDateTime; { otherwise, set calendar to the date }
end;

```

이제 데이터 찾아보기 컨트롤이 만들어졌습니다.

## 데이터 편집 컨트롤 생성

데이터 편집 컨트롤을 만들 때 컴포넌트를 생성하고 등록한 다음 데이터 찾아보기 컨트롤에서 행한대로 데이터 연결을 추가합니다. 또한 비슷한 방법으로 원본으로 사용한 필드의 데이터 변경 내용에 응답하며 몇 가지 문제를 더 처리해야 합니다.

예를 들어, 컨트롤이 키와 마우스 이벤트에 모두 응답하도록 하고자 합니다. 컨트롤은 사용자가 컨트롤의 내용을 변경할 때 응답해야 합니다. 사용자가 컨트롤을 종료할 때 컨트롤의 변경 내용이 데이터셋에 반영되어야 합니다.

여기에서 설명되는 데이터 편집 컨트롤은 이 장의 첫 번째 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 및 키 다운 이벤트 처리
- 필드 데이터 연결 클래스 업데이트
- Change 메소드 수정
- 데이터셋 업데이트

## FReadOnly의 기본값 변경

이것은 데이터 편집 컨트롤이므로 *ReadOnly* 속성은 *False*로 기본 설정되어야 합니다. *ReadOnly* 속성을 *False*로 만들려면 생성자에서 *FReadOnly*의 값을 변경합니다.

```

constructor TDBCalendar.Create(AOwner:TComponent);
begin
    ...
    FReadOnly := False; { set the default value }
    ...
end;

```



## 마우스 다운 및 키 다운 이벤트 처리

컨트롤 사용자가 상호 작용하기 시작할 때 컨트롤은 마우스 다운 이벤트 또는 키 다운 이벤트를 받습니다. 컨트롤이 이러한 메시지에 응답하도록 하려면 이러한 메시지에 응답하는 핸들러를 작성해야 합니다.

- 마우스 다운 이벤트에 응답
- 키 다운 이벤트에 응답

### 마우스 다운 이벤트에 응답

*MouseDown* 메소드는 컨트롤의 *OnMouseDown* 이벤트에 대한 *protected* 메소드입니다. 컨트롤은 자체적으로 운영 체제의 공지에 응답하여 *MouseDown*을 호출합니다. 상속된 *MouseDown* 메소드를 오버라이드할 때 *OnMouseDown* 이벤트를 호출하는 것은 물론 기타 응답을 제공하는 코드를 포함시킬 수 있습니다.

*MouseDown*을 오버라이드하려면 *MouseDown* 메소드를 *TDBCcalendar* 클래스에 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  ...
protected
  procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y:Integer);
    override;
  ...
end;

procedure TDBCcalendar.MouseDown(Button: TButton; Shift:TShiftState; X, Y:Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
    begin
      MyMouseDown := OnMouseDown;
      if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
    end;
  end;
end;
```

*MouseDown* 이 마우스 다운 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 없는 경우, 프로그래머가 *OnMouseDown* 이벤트 핸들러에 입력한 코드가 실행됩니다.

## 키 다운 이벤트에 응답

*KeyDown* 메소드는 컨트롤의 *OnKeyDown* 이벤트에 대한 *protected* 메소드입니다. 컨트롤은 자체적으로 운영 체제의 공지에 응답하여 *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 따릅니다.

- 1 *KeyDown* 메소드를 *TDBCcalendar* 클래스에 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  ...
  protected
    procedure KeyDown(var Key:Word; Shift: TShiftState; X: Integer; Y:Integer);
      override;
  ...
end;
```

- 2 *KeyDown* 메소드를 구현합니다.

```
procedure KeyDown(var Key:Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [Key_Up, Key_Down, Key_Left, Key_Right, Key_End,
    Key_Home, Key_Prior, Key_Next]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
    begin
      MyKeyDown := OnKeyDown;
      if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
    end;
end;
```

*KeyDown* 이 마우스 다운 이벤트에 응답할 때 상속된 *KeyDown* 메소드는 컨트롤의 *ReadOnly* 속성이 *False*이고 선택된 키가 커서 컨트롤 키 중 하나이며 데이터 연결 객체가 필드를 편집할 수 있는 편집 모드에 있는 경우에만 호출됩니다. 필드를 편집할 수 없거나 다른 키가 선택된 경우, 프로그래머가 *OnKeyDown* 이벤트 핸들러에 입력한 코드가 실행됩니다.

## 필드 데이터 연결 클래스 업데이트

데이터 변경에는 두 가지 유형이 있습니다.

- data-aware 컨트롤에 반영되어야 하는 필드 값 변경
- 필드 값에 반영되어야 하는 data-aware 컨트롤 변경

*TDBCcalendar* 컴포넌트에는 *CalendarDate* 속성에 값을 할당하여 데이터셋의 필드 값 변경을 처리하는 *DataChange* 메소드가 이미 있습니다. *DataChange* 메소드는 *OnDataChange* 이벤트의 핸들러입니다. 그러므로, 달력 컴포넌트는 데이터 변경의 첫 번째 유형을 처리할 수 있습니다.

이와 마찬가지로, 필드 데이터 연결에는 또한 컨트롤 사용자가 data-aware 컨트롤의 내용을 수정할 때 발생하는 *OnUpdateData* 이벤트가 있습니다. 달력 컨트롤에는 *OnUpdateData* 이벤트에 대한 이벤트 핸들러가 되는 *UpdateData* 메소드가 있습니다. *UpdateData*는 data-aware 컨트롤의 변경된 값을 필드 데이터 연결에 할당합니다.

- 1 필드 값에 달력의 값 변경을 반영하려면 *UpdateData* 메소드를 달력 컴포넌트의 private 섹션에 추가합니다.

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender:TObject);
    ...
  end;
```

- 2 *UpdateData* 메소드를 구현합니다.

```
procedure UpdateData(Sender:TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;      { set field link to calendar date }
end;
```

- 3 *TDBCcalendar*의 생성자에서 *UpdateData* 메소드를 *OnUpdateData* 이벤트에 할당합니다.

```
constructor TDBCcalendar.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

## Change 메소드 수정

*TDBCalendar*의 *Change* 메소드는 새로운 날짜 값이 설정될 때마다 호출됩니다. *Change*는 *OnChange* 이벤트 핸들러를 호출합니다. 컴포넌트 사용자는 *OnChange* 이벤트 핸들러에 코드를 작성하여 날짜 변경에 응답할 수 있습니다.

달력의 날짜가 변경되면, 원본으로 사용한 데이터셋에 변경이 발생했음을 공지해야 합니다. *Change* 메소드를 오버라이드하고 코드를 한 줄 더 추가하여 이 작업을 수행할 수 있습니다. 다음과 같은 단계를 따릅니다.

- 1 새로운 *Change* 메소드를 *TDBCalendar* 컴포넌트에 추가합니다.

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
    ...
  end;
```

- 2 데이터가 변경된 데이터셋을 공지하는 *Modified* 메소드를 호출하여 *Change* 메소드를 작성한 다음 상속된 *Change* 메소드를 호출합니다.

```
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;           { call the Modified method }
  inherited Change;             { call the inherited Change method }
end;
```

## 데이터셋 업데이트

지금까지 data-aware 컨트롤 내의 변경은 필드 데이터 연결 클래스의 값을 변경시켰습니다. 데이터 편집 컨트롤 생성의 최종 단계는 새 값을 갖는 데이터셋을 업데이트하는 것입니다. 이것은 data-aware 컨트롤의 값을 변경한 사람이 컨트롤 외부를 클릭하거나 *Tab* 키를 눌러 컨트롤을 종료한 후 발생합니다.

*TWidgetControl*에는 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 따릅니다.

- 1 *DoExit* 메소드에 대한 오버라이드를 *TDBCalendar* 컴포넌트에 추가합니다.

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
    ...
  end;
```

## 2 *DoExit* 메소드를 구현하면 다음과 같습니다.

```

procedure TDBCcalendar.CMExit(var Message: TWMNoParams);
begin
    try
        FDataLink.UpdateRecord;                { tell data link to update database }
    except
        on Exception do SetFocus;              { if it failed, don't let focus leave }
    end;
    inherited;                                { let the inherited method generate an OnExit event }
end;

```



# 색인

## 가

가변 오류 4-9  
가상  
    메소드 25-8, 28-4  
        속성 26-2  
        속성 편집기 30-6 ~ 30-7  
    메소드 테이블 25-8  
    지시어 25-8  
가속 키 6-25  
가시성(visibility) 3-9  
값 26-2  
    Null 16-26  
    기본 데이터 15-10  
    기본 속성 26-7, 26-12  
        재정의 31-2, 31-3  
    부울 26-2, 26-12, 34-4  
    테스트 26-7  
값 표현식 17-20  
개발자 지원 1-2  
객체 3-1, 3-5 ~ 3-12  
객체 생성자 10-11  
객체 잠금  
    스레드 9-7  
    호출 중첩 9-7  
객체 지향 프로그래밍 3-1 ~ 3-11, 25-1 ~ 25-9  
    상속 3-8  
    선언 25-3, 25-9  
        메소드 25-7, 25-8, 25-9  
        클래스 25-6  
    정의 3-1  
객체 포인터 역참조 25-9  
객체 필드 17-21 ~ 17-26  
    타입 17-21  
검색 경로 10-14  
검색 경로 구분자 10-14  
결과 매개변수 18-8  
경계를 이루는 사각형 8-11  
경로(URL) 22-2  
경로명 10-13  
계산된 필드 16-8, 16-28, 17-6  
    값 할당 17-8  
    정의 17-7 ~ 17-8  
    조회 필드 및 17-9  
    클라이언트 데이터셋 20-19, 20-20  
계층 구조(클래스) 25-3  
공급 21-1

공유 객체  
    국제화 12-5  
    동적(HTTP 서버) 22-4  
    생성 5-3  
    패키지 11-1  
공유 객체 파일 10-7, 10-13  
    설치 13-4  
    패키지 11-2  
공지 이벤트 27-8  
공통 대화 상자 3-31  
관계형 데이터베이스 14-1  
구분 표시 12-4  
구분자 표시줄(메뉴) 6-24  
구성 파일 10-13, 13-5  
국제적인 애플리케이션 12-1  
    약자 12-4  
    지역화 12-5  
국제화 12-1  
그래픽 29-1 ~ 29-5  
    HTML에 추가 22-17  
    owner-draw 컨트롤 7-7  
    객체 형식 8-3  
    교체 8-20  
    국제화 12-4  
    그리기와 색칠 대조 8-4  
    독립적인 29-2  
    드로잉 툴 29-1, 29-5, 32-5  
        변경 32-7  
    로드 8-19, 29-3  
    메소드 29-2, 29-3, 29-4  
        이미지 복사 29-4  
    문자열 목록 7-8 ~ 7-10  
    문자열에 연결 3-37  
    복사 8-21  
    복잡 29-3  
    붙여넣기 8-22  
    삭제 8-21  
    선 그리기 8-5, 8-9 ~ 8-10, 8-27 ~ 8-28  
        이벤트 핸들러 8-25  
        펜 너비 변경 8-6  
    양단 묶음 예제 8-23 ~ 8-28  
    이미지 다시 그리기 29-5  
    이미지 변경 8-20  
    저장 8-19, 29-3  
    컨테이너 29-2  
    컨트롤 추가 8-17  
크기 조정 8-20, 15-9, 29-4

파일 8-19 ~ 8-21  
파일 형식 8-3  
표시 3-30  
프레임 6-13  
프로그래밍 개요 8-1 ~ 8-3  
그래픽 객체  
    스레드 9-4  
그래픽 상자 15-2  
그래픽 컨트롤 24-4, 29-2, 32-1 ~ 32-10  
    그리기 32-3 ~ 32-9  
    비트맵과 비교 32-3  
    생성 24-4, 32-3  
    시스템 리소스 절감 24-4  
    이벤트 29-5  
그룹 상자 3-26  
그룹화 수준 20-19  
    유지 관리되는 집계 20-22  
그리기 그리드 3-29  
그리기 모드 8-28  
그리기 상자 3-31  
그리드 3-29 ~ 3-30, 33-1, 33-2, 33-4, 33-10  
    data-aware 15-14  
    값 얻기 15-16  
    그리기 15-24 ~ 15-25  
    기본 상태 15-15  
        복원 15-21  
    데이터 편집 15-6, 15-24  
    데이터 표시 15-15, 15-16  
    런타임 옵션 15-23 ~ 15-24  
    사용자 지정 15-16 ~ 15-21  
    색상 8-5  
    열 삽입 15-17  
    열 재정렬 15-19  
    열 제거 15-16, 15-18  
그림 8-17, 29-2  
    교체 8-20  
    로드 8-19  
    변경 8-20  
    저장 8-19  
그림 객체 8-3  
글꼴 13-9  
    높이 8-4  
금칙 처리 7-2  
기능

- 포팅 불가능 Windows 기능 10-6
- 기본
  - 값 15-10
  - 속성 값 26-7
  - 변경 31-2, 31-3
  - 지정 26-12
- 예약어 26-7
- 조상 클래스 25-4
- 지시어 31-3
- 프로젝트 옵션 5-2
- 핸들러
  - 오버라이드 27-9
  - 이벤트 27-9
- 기술 지원 1-2
- 기존 컨트롤 24-5
- 기하학적 도형
  - 그리기 32-9
- 긴 문자열 4-27
- 끝점
  - 소켓 연결 23-5

## 나

- 날짜
  - 국제화 12-4
- 날짜 필드
  - 서식 17-13
- 내장 프로시저 14-4, 18-7 ~ 18-10
  - TSQLDataSet 18-7
  - TSQLStoredProc 18-8
  - 데이터 반환 18-7
  - 매개변수 18-8 ~ 18-10
  - 클라이언트 데이터셋 20-28
  - 성능 18-7
  - 쿼리 18-2
- 년비주얼 컴포넌트 24-5, 24-11
- 년비주얼(nonvisual) 컴포넌트 3-10
- 노트북 분할기(divider) 3-27
- 논리값 15-2, 15-12
- 뉴스그룹 1-2
- 느린 프로세스
  - 스레드 사용 9-1

## 다

- 달력 33-1 ~ 33-12
  - 날짜 추가 33-4 ~ 33-9
  - 속성과 이벤트 정의 33-2, 33-6, 33-10
  - 오늘 날짜 선택 33-9
  - 이동 33-9 ~ 33-12

- 읽기 전용으로 생성 34-3 ~ 34-4
- 크기 조정 33-4
- 다각선 8-9, 8-10
- 그리기 8-9
- 다각형 8-11
- 그리기 8-11
- 다계층 애플리케이션 14-2, 14-12 ~ 14-13
- 매개변수 20-27
- 다중 문서 인터페이스 5-2
- 다중 스레드 애플리케이션 9-1
- 다중 읽기 배타적 쓰기 동기화 장치 9-8
- 사용 경고 9-8
- 다형성 3-2, 3-5
- 단방향 데이터셋 14-6, 18-1 ~ 18-20
- 데이터 페치 18-10 ~ 18-11
- 마스터/디테일 관계 18-14 ~ 18-15
- 명령 실행 18-11 ~ 18-14
- 바인딩 18-3 ~ 18-10
- 제한 사항 18-1
- 준비 18-10
- 타입 18-2
- 단일 계층 애플리케이션 14-2, 14-7, 14-8, 14-11
- 단일 문서 인터페이스 5-2
- 단축키
  - 메뉴에 추가 6-25
- 대상, 작업 목록 6-14
- 대소문자 구별 10-12
- 인덱스 20-17, 20-40
- 대화 상자
  - 공통 3-31
  - 국제화 12-4, 12-5
  - 멀티페이지 3-27
  - 속성 편집기 30-7
- 더블 바이트 문자 집합 12-2
- 더블 클릭
  - 응답 30-15
- 더블 클릭하기
  - 컴포넌트 30-13
- 데이터
  - 기본값 15-10, 17-20
  - 동기화
    - 여러 폼 15-3
  - 변경 16-22 ~ 16-27
  - 액세스 34-1
  - 입력 16-23
  - 저장 16-24
  - 페치 18-10 ~ 18-11
  - 표시 17-16, 17-16

- 그리드 15-15
- 다시 그리기 사용 불가능 15-6
- 현재값 15-8
- 표시 전용 15-8
- 형식, 국제화 12-4
- 데이터 그리드 15-2, 15-14, 15-15 ~ 15-26
- 값 얻기 15-16
- 그리기 15-24 ~ 15-25
- 기본 상태 15-15
- 복원 15-21
- 데이터 편집 15-6, 15-24
- 데이터 표시 15-15, 15-16
- ADT 필드 15-21
- 배열 필드 15-21
- 런타임 옵션 15-23 ~ 15-24
- 사용자 지정 15-16 ~ 15-21
- 열 삽입 15-17
- 열 재정렬 15-19
- 열 제거 15-16, 15-18
- 이벤트 15-25 ~ 15-26
- 데이터 동기화
  - 여러 폼 15-3
- 데이터 멤버 3-3
- 데이터 모듈 5-6 ~ 5-8, 14-5
- 비즈니스 룰 5-7
- 생성 5-7
- 원격과 표준 비교 5-6
- 웹 애플리케이션 22-6, 22-7
- 이름 재지정 5-7
- 폼에서 액세스 5-7
- 데이터 무결성 14-4
- 데이터 브로커 14-12, 20-26
- 데이터 소스 14-5, 15-3 ~ 15-4
- 사용 가능 15-4
- 사용 불가능 15-4
- 이벤트 15-4
- 데이터 액세스
  - 컴포넌트 14-1
- 데이터 액세스 컴포넌트
  - 스레드 9-4
- 데이터 연결 18-15, 34-5 ~ 34-7
- 초기화 34-6
- 데이터 입력 검증 17-14
- 데이터 제약 조건 제약 조건 참조
- 데이터 타입
  - 영구적 필드 17-6



- 데이터 패킷
  - 고유한 레코드 보장 21-4
  - 복사 20-23 ~ 20-24
  - 애플리케이션 정의 정보 20-23, 21-6
  - 업데이트된 레코드 새로 고침 21-6
  - 읽기 전용 21-5
  - 클라이언트 편집 제한 21-5
  - 페이지 20-29 ~ 20-31, 21-7
  - 편집 21-7
  - 필드 속성 포함 21-5
  - 필드 제어 21-4
- 데이터 필드 17-6
  - 정의 17-6 ~ 17-7
- 데이터 필터 16-8, 16-17 ~ 16-20
  - 런타임 시 설정 16-20
  - 범위와 비교 20-6
  - 빈 필드 16-18
  - 사용 가능/사용 불가능 16-17
  - 연산자 16-18
  - 정의 16-16 ~ 16-20
  - 쿼리와 비교 16-17
  - 클라이언트 데이터셋 매개변수 사용 20-28
- 데이터 형식
  - 기본 17-13
- 데이터 형식화
  - 국제적인 애플리케이션 12-4
- 데이터베이스 5-4, 14-1 ~ 14-4, 34-1
  - HTML 응답 생성 22-20 ~ 22-23
  - 관계형 14-1
  - 데이터 저장 16-24
  - 데이터 추가 16-26
  - 로그인 14-3, 19-6 ~ 19-7
  - 보안 14-3
  - 선택 14-2
  - 승인되지 않은 액세스 19-6
  - 액세스 16-1
  - 액세스 속성 34-5 ~ 34-6
  - 연결 19-1 ~ 19-14
  - 웹 애플리케이션 22-20
  - 타입 14-2
  - 트랜잭션 14-3 ~ 14-4
  - 파일 기반 14-2
- 데이터베이스 서버 5-4
  - 연결 14-8, 19-4
  - 연결 해제 19-5
- 계약 조건 17-20
  - 타입 14-2
- 데이터베이스 애플리케이션 5-4, 14-1
  - 아키텍처 14-4 ~ 14-13
  - 배포 13-4
  - 분산 5-6
  - 파일 기반 14-7 ~ 14-8, 20-38 ~ 20-42
  - 포팅 10-24
- 데이터베이스 탐색기 15-2, 15-26 ~ 15-29, 16-9, 16-10
  - 데이터 삭제 16-24
  - 도움말 힌트 15-28
  - 버튼 15-26
  - 버튼 사용 가능/사용 불가능 15-27
  - 편집 16-22
- 데이터셋 14-6, 16-1 ~ 16-28
  - HTML 문서 22-22
  - 검색 16-14 ~ 16-16, 20-3 ~ 20-5
  - 여러 열 16-15, 16-16
  - 인덱스 사용 16-15, 16-16, 20-3 ~ 20-5
  - 기본 상태 16-3
  - 단방향 14-6, 18-1 ~ 18-20
  - 제한 사항 18-1
  - 닫기 16-2 ~ 16-3, 16-4 ~ 16-5
  - 레코드 포스트 16-25
- 데이터 변경 16-22 ~ 16-27
  - 레코드 삭제 16-24
  - 레코드 추가 16-7, 16-23 ~ 16-24, 16-26
  - 레코드 포스트 16-24 ~ 16-25
  - 레코드 표시 16-13 ~ 16-14
  - 레코드 필터링 16-8, 16-17 ~ 16-20
  - 모드 16-3 ~ 16-4
  - 반복 19-11
  - 변경 취소 16-26
  - 사용자 지정 14-6
  - 상태 16-3 ~ 16-4
  - 서버에 연결 18-2 ~ 18-3, 20-36
  - 연결 20-10
  - 연결 해제하지 않고 닫기 19-11
- 열기 16-2
- 이벤트 16-27
- 인덱스 16-8
- 인덱싱되지 않은 데이터셋 16-26
- 작업 취소 16-28
- 찾아보기 16-5 ~ 16-6
- 커서 16-9
- 탐색 15-26, 16-9 ~ 16-13, 16-21
- 편집 16-6 ~ 16-7, 16-22 ~ 16-23
- 프로바이더 21-2
- 필드 16-1
- 현재 행 16-9
- 활성화 19-11
- 데이터셋 페이지 프로듀서 22-20
  - 필드 값 변환 22-21
- 데이터셋 프로바이더 14-10
- 데이터셋 필드 17-21, 17-25
  - 클라이언트 데이터셋 20-12
  - 표시 15-22
- 델타 패킷 21-7, 21-8
- 업데이트 스크린 21-10
- 편집 21-7, 21-8 ~ 21-9
- 도움말
  - 문맥에 따른 도움말 3-29
  - 툴팁 3-29
  - 힌트 3-29, 15-28
- 도움말 객체 등록 5-18
- 도움말 뷰어 5-13
- 도움말 선택기 5-19, 5-20
- 도움말 시스템 5-13
  - 객체 등록 5-18
  - 인터페이스 5-13
  - 툴 버튼 6-38
- 도형 3-30, 8-10 ~ 8-11, 8-14
  - 그리기 8-10, 8-14
  - 비트맵 속성으로 채우기 8-9
  - 채우기 8-7, 8-8
- 동적 공유 객체(DSO) 22-4
- 동적 메모리 3-39
- 동적 메소드 25-9
- 동적 열 15-16
  - 속성 15-16
- 동적 필드 17-2 ~ 17-3
- 드라이버 파일 13-5
- 드라이브 문자 10-13
- 드래그 앤 드롭
  - 이벤트 32-2
- 드로잉 3-30

- 드로잉 툴 29-1, 29-5, 32-5
  - 기본값으로 할당 6-34
  - 변경 8-13, 32-7
  - 애플리케이션에서
    - 여러 개 처리 8-12
  - 테스트 8-12, 8-13
- 드롭다운 메뉴 6-25 ~ 6-26
- 드롭다운 목록 15-10
  - 데이터 그리드 15-20
- 드릴 다운 폼 15-14
- 등록
  - 속성 편집기 30-9 ~ 30-10
  - 컴포넌트 24-11
  - 컴포넌트 편집기 30-16
- 디렉토리, Linux 10-14
- 디버깅
  - dbExpress 애플리케이션
    - 19-13 ~ 19-14
  - 웹 서버 애플리케이션
    - 22-23
- 디스패처 22-6
- 디자인
  - 애플리케이션 2-2
- 디자인 타임 인터페이스 25-6
- 디자인 타임 패키지 11-1, 11-4 ~ 11-6
- 디자인 툴 2-2

## 라

- 라디오 그룹 3-26
- 라디오 버튼 3-23, 15-2
  - data-aware 15-13
- 그룹화 3-26
- 선택 15-13
- 라이브러리
  - 사용자 지정 컨트롤 24-5
- 래퍼 24-5
  - 컴포넌트 래퍼 참조
- 런타임 라이브러리(RTL) 4-8
- 런타임 인터페이스 25-6
- 런타임 타입 정보 25-6
- 런타임 패키지 11-1, 11-2 ~ 11-4
- 레이블 3-28, 12-4, 15-2
  - 열 15-17
- 레지스트리 10-13
- 레코드
  - 객체와 비교 3-5
  - 검색 반복 20-5
  - 검색 조건 16-15, 16-16
  - 반복 16-11
  - 삭제 16-24
  - 새로 고침 15-7, 20-33 ~ 20-34
- 업데이트 16-26 ~ 16-27, 21-7 ~ 21-11
  - 델타 패킷 21-7, 21-8
  - 업데이트 스크린 21-10
  - 여러 21-6
  - 클라이언트 데이터셋
    - 20-31 ~ 20-33, 20-37
  - 테이블 식별 21-11
  - 업데이트 해결 20-32
  - 이동 15-26, 16-9 ~ 16-13, 16-21
  - 추가 16-7, 16-23 ~ 16-24, 16-26
  - 찾기 16-14 ~ 16-16, 20-3 ~ 20-5
  - 패치 18-10 ~ 18-11, 20-29 ~ 20-31
  - 포스트 16-24 ~ 16-25
  - 데이터 그리드 15-24
  - 데이터셋을 닫을 때
    - 포스트 16-25
  - 표시 16-13 ~ 16-14
  - 필터링 16-8, 16-17 ~ 16-20
- 레포지토리 5-8 ~ 5-11
  - 항목 사용 5-10
  - 항목 추가 5-9
- 로그인 스크립트 19-6 ~ 19-7
- 로컬 데이터베이스 14-2
- 로케일 12-1
  - 데이터 형식 12-4
  - 리소스 모듈 12-5
- 로케일 설정
  - 문자열 루틴 4-29
- 루트 디렉토리 10-14
- 리소스 24-7
  - 문자열 12-5
  - 분리 12-5
  - 시스템, 최적화 24-4
  - 이름 지정 30-4
  - 지역화 12-5
  - 추출 12-6
  - 캐시로 저장 29-1
- 리소스 모듈 12-5
- 리소스 지역화 12-5
- 리소스 할당 4-6
- 리소스를 캐시로 저장 29-1
- 리스닝 연결 23-3, 23-7, 23-8
  - 끊기 23-7
  - 포트 번호 23-5
- 리터럴 17-20
- 릴리스 노트 13-10

- 링크 HTML 태그(<A>) 22-17

## 마

- 마법사 5-8
  - Console Wizard 5-3
  - 컴포넌트 24-8
- 마스크 편집 컨트롤 3-18
- 마스터/디테일 관계 15-14, 18-14 ~ 18-15
  - TSQLClientDataSet
    - 20-36
  - 업데이트 적용 20-12
  - 연결된 커서 20-10 ~ 20-12
  - 연쇄 업데이트 21-5
  - 연쇄 삭제 21-5
  - 중첩 테이블 20-12 ~ 20-13
  - 참조 무결성 14-4
  - 클라이언트 데이터셋
    - 20-10 ~ 20-13
- 마스터/디테일 폼 15-14
- 마우스 버튼 8-24
  - 클릭 8-24, 8-25
  - 마우스 이동 이벤트 8-25
- 마우스 버튼 놓기 8-25
- 마우스 이벤트 8-23 ~ 8-25, 32-2
  - 매개변수 8-24
  - 상태 정보 8-24
  - 정의 8-23
  - 테스트 8-26
- 매개변수
  - HTML 태그 22-17
  - SQL 명령 19-12
  - SQL 정의 18-13
  - 결과 18-8
  - 내장 프로시저 18-8
  - 데이터베이스 연결 19-2 ~ 19-3
  - 마스터/디테일 관계 18-14 ~ 18-15
  - 마우스 이벤트 8-23, 8-24
  - 명명된 매개변수와 명명되지 않은 매개변수 비교 18-4
  - 속성 설정 26-6, 26-7
  - 배열 속성 26-8
  - 이름 18-8
  - 이벤트 핸들러 27-3, 27-8, 27-9
  - 입/출력 18-8
  - 입력 18-8
  - 추가 18-8

- 출력 18-8, 20-26
- 쿼리 18-4
- 클라이언트 데이터셋
  - 20-26 ~ 20-28
  - 레코드 필터링 20-28
- 클래스 25-9
- 매개변수 컬렉션 편집기 18-5, 18-8
- 매개변수화된 쿼리 18-4 ~ 18-6
- 멀티바이트 문자(MBCS)
  - 10-15, 10-20
- 멀티태스킹 10-13
- 멀티페이지 대화 상자 3-27
- 멀티프로세싱
  - 스레드 9-1
- 메뉴 6-20 ~ 6-32
  - owner-draw 7-7
  - 국제화 12-4, 12-5
  - 명령 액세스 6-25
  - 사이 이동 6-29
  - 이름 지정 6-23
  - 이미지 추가 6-27
  - 이벤트 처리 6-32
  - 작업 목록 6-14
  - 재사용 6-29
  - 추가 6-22 ~ 6-28
    - 드롭다운 6-25 ~ 6-26
  - 템플릿 6-22, 6-29, 6-30
    - 로드 6-30
    - 삭제 6-31
  - 템플릿으로 저장 6-30, 6-31 ~ 6-32
  - 팝업 7-6
  - 표시 6-27, 6-29
  - 항목 사용 불가능 7-5
- 메뉴 디자이너 6-20 ~ 6-21
- 컨텍스트 메뉴 6-28
- 메뉴 컴포넌트 6-21
- 메뉴 템플릿 6-31
- 메뉴 표시줄
  - 항목 이동 6-26
- 메뉴 항목 6-24 ~ 6-25
  - 구분자 표시줄 6-24
  - 그룹화 6-24
  - 문자에 밑줄 긋기 6-25
  - 삭제 6-24, 6-29
  - 속성 설정 6-28
  - 위치 표시자 6-29
  - 이동 6-26
  - 이름 지정 6-23, 6-32
  - 정의 6-20
  - 중첩 6-25
  - 추가 6-24, 6-32
  - 편집 6-28

- 메모 컨트롤 7-1
  - 속성 3-19
- 메모 필드 15-2, 15-9
- 메모리
  - 보존 25-9
  - 폼에서 누수(leak) 6-5
- 메모리 관리
  - COM 객체 4-20
  - 인터페이스 4-24
- 메소드 3-4, 8-15, 24-7, 28-1, 33-9
  - protected 28-3
  - public 28-3
  - 가상 25-8, 28-4
  - 객체 3-5, 3-7
  - 그래픽 29-2, 29-3, 29-4
  - 그리기 32-8, 32-9
  - 디스패칭 25-7
  - 상속 27-6
  - 선언 8-15, 28-4
    - public 28-3
    - 가상 25-8
    - 동적 25-9
    - 정적 25-7
  - 속성 26-5 ~ 26-7, 28-1, 28-2, 32-4
  - 오버라이드 25-8, 33-10
  - 이벤트 핸들러 27-3, 27-5
    - 오버라이드 27-6
  - 재정의 25-8
  - 초기화 26-13
  - 호출 27-6, 28-3, 32-4
- 메소드 재정의 25-8
- 메소드 포인터 27-2, 27-3, 27-8
- 메시지 33-4
  - 핸들러 33-4
- 메시지 기반 서버
  - 웹 서버 애플리케이션 참조
- 메시지 루프
  - 스레드 9-4
- 메시지 헤더(HTTP) 22-1, 22-2
- 메시징 10-19
- 메인 폼 6-1
- 메타데이터 18-16 ~ 18-20, 19-10 ~ 19-11
  - 수정 18-13
  - 프로바이더에서 얻기 20-30
  - 형식 18-17 ~ 18-20
- 메타파일 8-16, 8-19, 29-2
  - 사용 시기 8-3
- 멤버 함수 3-4
- 명령 스위치 10-12

- 명령, 작업 목록 6-14
- 명명되지 않은 매개변수 18-4
- 명명된 매개변수 18-4
- 명명된 연결 14-9, 19-3 ~ 19-4
  - 런타임 시 로드 19-3
  - 삭제 19-4
  - 이름 재지정 19-4
  - 추가 19-4
- 모달 폼 6-4
- 모달리스 6-6
- 모달리스 폼 6-4
- 모듈 24-10
- 모바일 컴퓨팅 14-13
- 모서리가 둥근 사각형 8-11
- 모양
  - 윤곽 8-5
- 목록
  - 문자열 3-32 ~ 3-37
  - 스레드에서 사용 9-5
- 목록 뷰
  - owner draw 7-7
- 목록 상자 3-24, 15-2, 15-11, 33-1
  - data-aware 15-10 ~ 15-12
  - owner-draw 7-7
    - draw-item 이벤트 7-10
  - measure-item 이벤트 7-9
- 속성 저장
  - 예제 6-8
  - 채우기 15-10
- 목록 컨트롤 3-24 ~ 3-25
- 문자 26-2
- 문자 집합 4-28, 12-2, 12-2 ~ 12-3
  - 2바이트 변환 12-2
  - 국제적인 정렬 순서 12-4
  - 더블 바이트 12-2
- 문자 타입 4-25, 12-2
- 문자열 4-25, 26-2
  - 2바이트 변환 12-2
  - PChar 변환 4-33
  - 긴 문자열 4-27
- 루틴
  - 대소문자 구별 4-29
  - 런타임 라이브러리 4-28
- 멀티바이트 문자 지원
  - 4-29
- 메모리 훼손 4-35
- 반환 26-9
- 번역 12-2, 12-5
- 변수 매개변수 4-35

- 선언 및 초기화 4-31
- 시작 위치 7-3
- 잘라내기 12-3
- 정렬 12-4
- 지역 변수 4-33, 4-34
- 참조 카운팅 문제 4-27, 4-33
- 컴파일러 지시어 4-35
- 크기 7-3
- 타입 개요 4-26
- 타입의 혼합 및 변환 4-32
- 파일 4-44
- 확장 문자 집합 4-36
- 문자열 그리드 3-30
- 문자열 목록 3-32 ~ 3-37
- 객체 추가 7-8 ~ 7-10
- 기간이 긴 문자열 3-34
- 기간이 짧은 문자열 3-33
- 문자열 삭제 3-36
- 문자열 이동 3-36
- 문자열 찾기 3-36
- 반복 3-36
- 복사 3-37
- 부분 문자열 3-36
- 생성 3-33
- 연결 객체 3-37
- 위치 3-35, 3-36
- 정렬 3-36
- 추가 3-36
- 파일에 저장 3-33
- 파일에서 로드 3-33
- 문자열 변경 12-2, 12-5
- 2바이트 변환 12-2
- 문자열 속성 4-27
- 문자열 연산자 4-36
- 문자열 예약어
- 속성 타입 4-27
- 문자열 필드
- 크기 17-6

## 바

- 배경 12-4
- 배열 26-2
- 배열 필드 17-21, 17-24 ~ 17-25
- 영구적 필드 17-24
- 평평하게 하기 (flattening) 15-21
- 표시 15-21, 17-22
- 배치 파일 10-12
- 배포
- 공유 객체 파일 13-4
- 글꼴 13-9
- 데이터베이스 애플리케이션 13-4

- 애플리케이션 13-1
- 일반 애플리케이션 13-2
- 웹 애플리케이션 13-6
- 패키지 파일 13-4
- 버튼 3-21 ~ 3-23
- glyph 할당 6-34
- 탐색기 15-26
- 틀바 6-32
- 틀바에 추가 6-33 ~ 6-35, 6-36
- 틀바에서 사용 불가능 6-36
- 번역 12-4
- 범위 20-6 ~ 20-9
- Null 값 20-7
- 경계 20-8
- 변경 20-8
- 인덱스 20-6
- 적용 20-9
- 지정 20-6 ~ 20-9
- 취소 20-9
- 필터와 비교 20-6
- 부동 소수점 오류 4-9
- 북마크 16-13 ~ 16-14
- 데이터셋 타입으로 지원 16-13
- 부분 키
- 검색 20-5
- 범위 설정 20-8
- 부울 값 26-2, 26-12, 34-4
- 부울 필드 15-2, 15-12
- 변경
- 프로젝트 파일 2-2
- 변경 로그 20-14 ~ 20-16, 20-31, 20-41
- 변경 내용 저장 20-15 ~ 20-16
- 변경 취소 20-15
- 변수
- 객체 3-10
- 선언
- 예제 3-10

## 변환

- PChar 4-33
- 문자열 4-33
- 필드값 17-15, 17-17 ~ 17-18

- 병렬 프로세스
- 스래드 9-1

## 분리

- 트랜잭션 14-4

- 분리 레벨 19-9 ~ 19-10

## 보안

- 데이터베이스 14-3, 19-6 ~ 19-7

- 복사(Object Repository) 5-10
- 분산 애플리케이션 5-5
- 데이터베이스 5-6
- 브러시 8-7 ~ 8-9, 32-5
- 변경 32-7
- 비트맵 속성 8-9
- 색상 8-8
- 스타일 8-8
- 브리프케이스 모델 14-13
- 비즈니스 룰
- 데이터 모듈 5-7
- 비차단 연결 23-9 ~ 23-10
- 차단 연결과 비교 23-9
- 비트맵 3-30, 8-17, 29-2
- draw-item 이벤트 7-10
- 교체 8-20
- 국제화 12-5
- 그래픽 컨트롤과 비교 32-3
- 그리기 8-18
- 드로잉 표면 29-2
- 로드 29-3
- 문자열과 연결 7-8
- 문자열에 연결 3-37
- 브러시 8-9
- 브러시 속성 8-7, 8-9
- 빈 비트맵 8-17
- 소멸 8-21
- 스크롤 8-17
- 스크롤 추가 8-16
- 애플리케이션에 나타날 때 8-1
- 오프스크린 29-3 ~ 29-4
- 임시 8-17
- 초기 크기 설정 8-17
- 컴포넌트에 추가 30-4
- 틀바 6-36
- 프레임 6-13
- 비트맵 객체 8-3
- 비트맵 버튼 3-22
- 빗면 3-31

## 사

### 사각형

- 그리기 8-10, 32-9
- 사용권 계약서 13-10
- 사용자 인터페이스 3-17
- 단일 레코드 15-7
- 데이터 구성 15-7 ~ 15-8, 15-14
- 레이아웃 6-4
- 분리 14-5
- 여러 레코드 15-14
- 폼 6-1 ~ 6-2
- 사용자 정의 타입 32-3

- 사용자 지정 데이터셋 14-6
- 사용자 지정 컨트롤 24-4
  - 라이브러리 24-5
- 삼각형 8-11
- 상속 3-5, 3-8
  - 메소드 27-6
  - 속성 32-2, 33-2
  - 게시 26-3
  - 이벤트 27-5
- 상속(Object Repository) 5-10
- 상태 정보 3-28
  - 마우스 이벤트 8-24
  - 통신 21-7
- 상태 표시줄 3-28
  - owner draw 7-7
  - 국제화 12-4
- 상호 배타적 옵션 6-35
- 색상
  - 국제화 12-4
  - 펜 8-5
- 색상 그리드 8-5
- 색상 수 13-7
  - 프로그래밍 13-9
- 생략(...)
  - 그리드의 버튼 15-21
- 생성
  - 애플리케이션 3-17
- 생성자 3-11, 24-12, 26-12, 28-3, 33-3, 33-4, 34-6
  - 소유된 객체 32-5, 32-7
  - 여러 6-7
  - 오버라이드 31-2
- 서버 소켓 23-6 ~ 23-7
  - 오류 메시지 23-7
  - 이벤트 처리 23-8
  - 지정 23-6
  - 클라이언트 승인 23-9
  - 클라이언트 요청 승인 23-6
- 서버 애플리케이션
  - 서비스 23-1
  - 인터페이스 23-2
- 서버 연결 23-3
  - 포트 번호 23-5
- 서비스
  - 구현 23-1 ~ 23-2, 23-6
  - 네트워크 서버 23-1
  - 요청 23-6
  - 포트 23-2
- 서식있는 텍스트 컨트롤 7-1
- 선
  - 그리기 8-5, 8-9, 8-9 ~ 8-10, 8-27 ~ 8-28
  - 이벤트 핸들러 8-25
  - 펜 너비 변경 8-6

- 지우기 8-28
- 선언
  - 메소드 8-15, 28-4
  - public 28-3
  - 가상 25-8
  - 동적 25-9
  - 정적 25-7
- 변수
  - 예제 3-10
- 새 컴포넌트 타입 25-3
- 속성 26-3, 26-3 ~ 26-7, 26-8, 26-13, 27-8, 32-4
  - stored 26-13
  - 사용자 정의 타입 32-3
- 이벤트 핸들러 27-5, 27-8, 33-11
- 클래스 25-9, 32-5
  - public 25-6
  - published 25-6
- 선택기
  - 도움말 5-19
- 설명서
  - 주문 1-2
- 세션
  - 웹 애플리케이션 22-20
- 셀(그리드) 3-29
- 셀 스크립트 10-12
- 소멸자 3-11, 28-3, 34-6
  - 소유된 객체 32-5, 32-7
- 소스 코드
  - 재사용 5-11
  - 최적화 8-15
  - 편집 2-2
- 소스 파일
  - 변경 2-2
  - 패키지 11-2, 11-6, 11-7, 11-12
- 소스 파일, 공유 10-11
- 소유된 객체 32-5 ~ 32-8
  - 초기화 32-7
- 소켓 5-5, 23-1 ~ 23-10
  - 쓰기 23-10
  - 네트워크 주소 23-4
  - 서비스 구현 23-1 ~ 23-2, 23-6
  - 설명 23-4
  - 오류 처리 23-8
  - 이벤트 처리 23-7 ~ 23-9, 23-10
  - 읽기 23-10
  - 읽기/쓰기 23-9 ~ 23-10
  - 정보 제공 23-4
  - 클라이언트 요청 승인 23-3
  - 호스트 지정 23-4

- 소켓 객체 23-5
  - 클라이언트 23-6
  - 클라이언트 소켓 23-7
- 소켓 연결 23-3
  - 끊기 23-6, 23-7
  - 끝점 23-4, 23-5
  - 여러 23-5
  - 열기 23-6, 23-7
  - 정보 보내기/받기 23-9
  - 타입 23-3
- 소켓 컴포넌트 23-5 ~ 23-7
- 소프트웨어 사용권 요구 사항 13-9
- 속성 3-3, 26-1 ~ 26-13
  - HTML 테이블 22-21
  - published 33-2
  - published가 아닌 저장 및 로드 26-13 ~ 26-15
  - read 및 write 26-5
  - 값 쓰기 26-7, 30-6
  - 값 읽기 30-6
  - 값 지정 30-6
  - 개요 24-6
  - 객체 3-5
  - 기본값 26-7, 26-12
  - 재정의 31-2, 31-3
  - 내부 데이터 저장소 26-4, 26-6, 26-7
  - 로드 26-13
  - 배열 26-2, 26-8
  - 변경 30-4 ~ 30-10, 31-2, 31-3
  - 보기 30-6
  - 상속 26-3, 32-2, 33-2
  - 선언 26-3, 26-3 ~ 26-7, 26-8, 26-13, 27-8, 32-4
  - stored 26-13
  - 사용자 정의 타입 32-3
- 속성 편집기 30-8
- 쓰기 전용 26-6
- 엑세스 26-5 ~ 26-7
- 업데이트 24-7
- 이벤트 27-1, 27-3
- 읽기 전용 25-6, 26-7, 34-3
- 재선언 26-12, 27-5
- 저장 26-13
- 클래스 26-2
- 타입 26-2, 26-9, 30-6, 32-3
- 편집
  - 텍스트로 30-5
- 속성 설정
  - 쓰기 26-8

- 읽기 26-8
- 속성 설정 읽기 26-6
- 속성 편집기 26-2, 30-4 ~ 30-10
- 대화 상자 30-7
- 등록 30-9 ~ 30-10
- 속성 30-8
- 파생된 클래스 30-4
- 수직 트랙 표시줄 3-20
- 수평 트랙 표시줄 3-20
- 수학 오류 4-9
- 순환 참조 6-2
- 숨겨진 필드 21-4
- 숫자 26-2
- 국제화 12-4
- 속성 값 26-12
- 숫자 필드
- 서식 17-13
- 스레드 9-1 ~ 9-11
- ID 9-11
- 객체 잠금 9-7
- 그래픽 객체 9-4
- 대기 9-8
- 여러 9-9
- 데이터 액세스 컴포넌트 9-4
- 동시 액세스 피하기 9-7
- 메시지 루프 9-4
- 목록 사용 9-5
- 반환 값 9-9
- 수 제한 9-10
- 실행 9-10
- 실행 막기 9-7
- 예외 9-6
- 우선 순위 9-1, 9-2
- 오버라이드 9-11
- 이벤트 대기 9-9
- 작성 9-10
- 조정 9-3, 9-6 ~ 9-10
- 종료 9-5
- 중요 섹션 9-7
- 중지 9-11
- 초기화 9-2
- 프로세스 공간 9-3
- 해제 9-2, 9-3
- 스레드 객체 9-1
- 정의 9-2
- 초기화 9-2
- 스레드 콜백 변수 9-5
- OnTerminate 이벤트 9-6
- 스레드 변수 9-5
- 스레드 함수 9-3
- 스레드에 대해 안전한 객체 9-4
- 스크롤 가능한 비트맵 8-16

- 스크롤 막대 3-20
- 텍스트 창 7-2
- 스크롤 상자 3-27
- 스크립트(URL) 22-2
- 스키마 정보 18-16 ~ 18-20, 19-10 ~ 19-11
- 내장 프로시저 18-18
- 매개변수 18-20
- 인덱스 18-19
- 테이블 18-17
- 필드 18-18 ~ 18-19
- 스타일 10-5
- 스플리터 3-23
- 스피드 버튼 3-22
- glyph 할당 6-34
- 가운데 맞춤 6-34
- 그룹화 6-35
- 드로잉 툴 8-13
- 이벤트 핸들러 8-13
- 작동 모드 6-33
- 초기 상태, 설정 6-34
- 토글로 사용 6-35
- 툴바에 추가 6-33 ~ 6-35
- 스핀 편집 컨트롤 3-21
- 시간
- 국제화 12-4
- 시간 초과 이벤트 9-10
- 시간 필드
- 서식 17-13
- 시그널 10-13
- 시스템 공지 10-19
- 시스템 리소스 최적화 24-4
- 시스템 리소스, 절감 24-4
- 시스템 이벤트 10-19
- 식별자
- 리소스 30-4
- 속성 설정 26-6
- 이벤트 27-9
- 잘못된 6-23
- 클래스 필드 27-3
- 실행 파일 10-13
- 국제화 12-5, 12-6
- 심볼릭 링크 10-14
- 쓰기 전용 속성 26-6

## 아

---

- 아이콘 3-30, 29-2
- 그래픽 객체 8-3
- 툴바 6-36
- 트리 뷰 3-25
- 아키텍처
- 데이터베이스 애플리케이션 14-4 ~ 14-13
- 웹 서버 애플리케이션 22-6
- 알기 쉬운 툴바 6-37

- 애플리케이션
- Apache 웹 서버 5-6
- CGI 5-6
- MDI 5-2
- SDI 5-2
- 국제화 12-1
- 다중 스레드 9-1
- 데이터베이스 5-4, 14-1
- 배포 13-1
- 분산 5-5
- 상태 정보 3-28
- 생성 3-17
- 웹 서버 5-6
- 크로스 플랫폼 10-1 ~ 10-28
- 파일 13-3
- 포팅 10-15
- 포팅 가능 10-1 ~ 10-28
- 애플리케이션 변수 22-6
- 애플리케이션 서버 14-12
- 애플리케이션 지역화 12-6
- 애플리케이션 포팅 10-1 ~ 10-28
- 액세스 3-8 ~ 3-9
- helper 3-32
- TObject 3-13
- 개요 4-1
- 년비주얼(nonvisual) 3-10
- 사용자 지정 3-8
- 상속 3-8
- 소멸 3-11
- 소유된 32-5 ~ 32-8
- 초기화 32-7
- 속성 3-5
- 여러 인스턴스 3-6
- 인스턴스화 3-6, 27-2
- 이벤트 3-7
- 임시 29-4
- 정의 3-5
- 액세스 위반
- 문자열 4-31
- 액자 모양의 패널 3-31
- 앰퍼샌드(&) 문자 6-25
- 양단 묶음 예제 8-23 ~ 8-28
- 양방향 속성 10-6
- 어셈블러 코드 10-18
- 업다운 컨트롤 3-21
- 업데이트
- 작업 6-17
- 클라이언트 데이터셋 20-37
- 업데이트 오류
- 해결 20-32, 20-32 ~ 20-33, 21-8, 21-11

- 여러 줄의 텍스트 컨트롤 15-9
- 여러 폼 15-3
- 연결 18-15
  - SQL 연결
    - 닫기 19-5
    - 바인딩 19-2 ~ 19-4
    - 열기 19-4
    - 유지 19-5
    - 이름 지정 19-3 ~ 19-4
  - TCP/IP 23-3
  - 데이터베이스 13-5
  - 열기 23-6
  - 종료 23-7
  - 클라이언트 23-3
- 연결 매개변수 19-2 ~ 19-3, 19-4
  - 로그인 정보 19-6
- 연결 이름 19-3 ~ 19-4
  - 변경 19-4
  - 삭제 19-4
  - 정의 19-4
- 연결 차단
- 이벤트 처리 23-9
- 연결 컴포넌트
  - SQL 연결 14-9, 18-2 ~ 18-3, 19-1 ~ 19-14
  - SQL 명령 실행 19-12 ~ 19-13
  - 각 연결에 대한 문장 19-1
  - TSoapConnection 14-13
- 연결 파일 13-5
- 연결 해제된 모델 14-13
- 연결된 선분 8-10
- 연쇄 업데이트 21-5
- 연쇄 삭제 21-5
- 열 3-29
  - HTML 테이블에 포함 22-22
  - 기본 상태 15-15, 15-21
  - 삭제 15-16
  - 속성 15-16, 15-19 ~ 15-20
  - 재설정 15-21
- 영구적인 열 15-15, 15-16 ~ 15-17
  - 삭제 15-18
  - 삽입 15-18
  - 생성 15-17 ~ 15-21
  - 재정렬 15-19
- 열 헤더 3-27, 15-17, 15-20
- 열거 타입 26-2, 32-3
  - 상수 8-13
  - 선언 8-12
- 영구적 열 15-15, 15-16 ~ 15-17
  - 삭제 15-16, 15-18
  - 삽입 15-18
  - 생성 15-17 ~ 15-21
  - 재정렬 15-19
- 영구적 필드 15-15, 17-3 ~ 17-14
  - ADT 필드 17-22 ~ 17-23
  - 데이터 타입 17-6
  - 데이터 패킷 21-4
  - 동적으로 변환 17-3
  - 배열 필드 17-24
  - 삭제 17-10 ~ 17-11
  - 생성 17-4 ~ 17-5, 17-5 ~ 17-10
  - 속성 17-11 ~ 17-14
  - 열거 17-4, 17-5
  - 이름 지정 17-5
  - 정렬 17-5
  - 정의 17-5 ~ 17-10
  - 클라이언트 데이터셋 생성 20-39
  - 특별 타입 17-5, 17-6
- 예약어 발생 4-16
- 예약어 쓰기 32-4
- 예약어 읽기 32-4
- 예약어 입력 8-12
- 예외 4-4 ~ 4-17, 10-13, 28-2
  - RTL 4-8, 4-9
  - 발생 4-16
  - 사용자 정의 4-16
  - 스레드 9-6
  - 인스턴스 4-11
  - 응답 4-4
  - 재발생 4-13
  - 중첩 4-6
  - 조용한 예외 4-15
  - 처리 4-5
  - 컴포넌트 4-14
  - 클래스 4-12
- 예외
  - 예외 처리 참조
- 예외 처리 4-4 ~ 4-17
  - TApplication 4-14
  - 개요 4-4 ~ 4-17
  - 객체 선언 4-16
  - 기본 핸들러 4-12
  - 리소스 보호 블록 4-7
  - 리소스 할당 보호 4-6
  - 문장 4-10
  - 범위 4-11
  - 제어 흐름 4-5
- 코드 블록 보호 4-4
- 클린업 코드 실행 4-4
- 핸들러 작성 4-9
- 오류
  - override 지시어 25-8
- 오류 메시지
  - 국제화 12-5
- 오버라이드
  - 메소드 25-8, 33-10
- 오브젝트 파스칼 3-3
- 오프스크린 비트맵 29-3 ~ 29-4
- 옵션 매개변수 20-23, 21-6
- 와이드 문자 12-3
- 런타임 라이브러리 루틴 4-28
- 외국어 번역 12-1
- 요약 값
  - 유지 관리되는 집계 20-22
- 요구 즉시 폐지 20-30
- 요청 객체
  - 헤더 정보 22-7
- 요청 메시지 22-6
- HTTP 개요 22-3 ~ 22-4
- 디스패칭 22-8
- 유형 22-12
- 응답 22-10 ~ 22-11, 22-15
- 작업 항목 22-8
- 처리 22-8
- 컨텐츠 22-13
- 헤더 정보 22-11 ~ 22-13
- 요청 목록(패키지) 30-16
- 요청 헤더 22-12
- 우선 순위
  - 스레드 사용 9-1, 9-2
- 원, 그리기 32-9
- 원격 데이터베이스 서버 14-2
- 원격 서버
  - 승인되지 않은 액세스 19-6
- 원격 애플리케이션
  - TCP/IP 23-1
- 원자성
  - 트랜잭션 14-3
- 웹 디스패처 22-5, 22-7 ~ 22-8
  - 요청 처리 22-6, 22-11
  - 작업 항목 선택 22-9, 22-10
- 웹 모듈 22-5, 22-7
- 데이터베이스 세션 추가 22-20
- 웹 브라우저 22-3
- URL 22-3

웹 사이트(Kylix 지원)  
 1-2  
 웹 서버  
 클라이언트 요청 22-4  
 웹 서버 애플리케이션 5-6,  
 22-1 ~ 22-23  
 개요 22-4 ~ 22-7  
 데이터 포스트 대상 22-13  
 데이터베이스 액세스  
 22-20  
 데이터베이스 연결 관리  
 22-20  
 디버깅 22-23  
 리소스 위치 22-2  
 만들기 22-5  
 변환 22-23  
 웹 디스패처 22-7  
 유형 22-4  
 응답 만들기 22-10  
 응답 템플릿 22-16  
 이벤트 처리 22-8, 22-9,  
 22-11  
 테이블 쿼리 22-22  
 템플릿 22-6, 22-24  
 파일 보내기 22-15  
 표준 22-1  
 프로젝트에 추가 22-6  
 웹 애플리케이션  
 객체 22-6  
 배포 13-6  
 웹 페이지 22-3  
 위임(delegation) 27-1  
 위치 독립 코드(PIC) 10-7,  
 10-18  
 유니코드 문자 12-3  
 문자열 4-27  
 유니코드 표준  
 문자열 4-25  
 유닛  
 기존  
 컴포넌트 추가 24-10  
 컴포넌트 추가 24-10  
 패키지 포함 11-3  
 유닛 비교 10-7  
 유지 관리되는 집계 20-20 ~  
 20-22  
 값 20-22  
 요약 연산자 20-21  
 지정 20-20 ~ 20-21  
 집계 20-21  
 유효 범위(객체) 3-8 ~ 3-9  
 윤곽, 그리기 8-5  
 윤년 33-7  
 응답 메시지 22-6  
 데이터베이스 정보 22-20  
 ~ 22-23  
 만들기 22-16 ~ 22-23  
 보내기 22-11, 22-16  
 상태 정보 22-14  
 생성 22-13 ~ 22-15  
 콘텐츠 22-15, 22-16 ~  
 22-23  
 헤더 정보 22-14 ~ 22-15  
 응답 템플릿 22-16  
 응답 헤더 22-15  
 이름 지정 규칙  
 리소스 30-4  
 속성 26-6  
 이벤트 27-9  
 필드 27-3  
 이미지 3-30, 15-2, 29-2  
 국제화 12-4  
 그리기 32-8  
 다시 그리기 29-5  
 메뉴에 추가 6-27  
 변경 8-20  
 브러시 8-9  
 스크롤 8-17  
 저장 8-19  
 지우기 8-21  
 추가 8-17  
 컨트롤 8-1, 8-16  
 컨트롤 추가 7-8  
 톨 버튼 6-36  
 표시 3-30  
 프레임 6-13  
 화면 펄름 감소 29-3  
 이미지 다시 그리기 29-5  
 이벤트 3-4, 10-19, 24-6,  
 27-1 ~ 27-10  
 data-aware 컨트롤  
 사용 가능 15-7  
 widget 3-4  
 객체 3-10  
 검색 27-3  
 구현 27-2, 27-3, 27-4  
 그래픽 컨트롤 29-5  
 대기 9-9  
 데이터 그리드 15-25 ~  
 15-26  
 데이터 소스 15-4  
 데이터셋 16-27  
 마우스 8-23 ~ 8-25  
 테스트 8-26  
 상속 27-5  
 새로운 정의 27-7 ~  
 27-10  
 시간 초과 9-10  
 시스템 3-4  
 신호 지정 9-9  
 애플리케이션 레벨 6-3  
 액세스 27-5  
 응답 27-6, 27-8, 27-9,  
 34-7  
 이름 지정 27-9  
 표준 27-4, 27-4 ~ 27-6  
 필드 객체 17-14  
 이벤트 객체 9-9  
 이벤트 신호 지정 9-9  
 이벤트 핸들러 3-7, 10-19,  
 24-6, 27-2, 34-7  
 공유 8-15  
 기본, 오버라이드 27-9  
 매개변수 27-3, 27-8,  
 27-9  
 공지 이벤트 27-8  
 메뉴 7-6  
 템플릿 6-32  
 메소드 27-3, 27-5  
 오버라이드 27-6  
 버튼 클릭에 응답 8-13  
 빈 27-9  
 선 그리기 8-25  
 선언 27-5, 27-8, 33-11  
 작성 3-8  
 참조에 의한 매개변수 전  
 달 27-9  
 코드 에디터 표시 30-15  
 타입 27-3 ~ 27-7, 27-8  
 포인터 27-2, 27-3, 27-8  
 인덱스 26-8, 26-9  
 관련 정보 20-18  
 데이터 그룹화 20-18 ~  
 20-19  
 레코드 정렬 20-16  
 범위 20-6  
 부분 키 20-5  
 삭제 20-17  
 임시 20-17  
 지정 20-3  
 클라이언트 데이터셋  
 20-16 ~ 20-40  
 테이블 18-7  
 인덱스 기반 검색 16-15,  
 16-16, 20-3 ~ 20-5  
 인덱스 정의  
 클라이언트 데이터셋  
 20-39  
 인덱싱되지 않은 데이터셋  
 16-26  
 인쇄 3-38  
 인스턴스 27-2  
 인터넷 표준 및 프로토콜 22-1



- 인터페이스 4-17 ~ 4-24, 25-4
  - as 연산자 4-21
  - Ctrl+Shift+G 4-21
  - IID 4-21, 4-24
  - IUnknown, 구현 4-20
  - TComponent 4-24
  - 개요 4-17 ~ 4-24
  - 국제화 12-3, 12-5
  - 년비주얼 프로그램 요소 24-5
  - 다형성 4-18
  - 단일 상속 확장 4-17, 4-18
  - 도움말 시스템 5-13
  - 동적 바인딩 4-21
  - 동적 쿼리 4-20
  - 디자인 타임 25-6
  - 랭귀지 기능 4-17
  - 런타임 25-6
  - 메모리 관리 4-21, 4-23
  - 수명 관리 4-20, 4-23
  - 예제 코드 4-18, 4-22, 4-23
  - 위임 4-22
  - 참조 카운팅 4-20, 4-21
  - 추상화(aggregation) 4-22
  - 컴포넌트 4-24
  - 코드 재사용 4-22
  - 코드 최적화 4-24
  - 클래스 간 공유 4-18
  - 파생 4-20
  - 프로시저 4-19
- 인터페이스 키워드 4-17
- 인트라넷
  - 호스트 이름 23-4
- 일관성
  - 트랜잭션 14-3
- 일대다 관계 18-14, 20-10
- 일반(simple) 타입 26-2
- 읽기 전용 속성 25-6, 26-7, 34-3
- 읽기 전용 필드 15-5
- 임시 객체 29-4
- 입/출력 매개변수 18-8
- 입력 매개변수 18-8
- 입력 컨트롤 3-20
- 입력 포커스 24-4
- 필드 17-15

## 자

- 자손 클래스 3-8, 25-3 ~ 25-4
  - 메소드 재정의 25-8

- 작업 6-13 ~ 6-19
  - 업데이트 6-17
  - 등록 6-19
  - 실행 6-15
  - 이미 정의된 6-18
- 작업 목록 6-13 ~ 6-19
- 작업 편집기
  - 작업 변경 22-9
  - 작업 추가 22-7
- 작업 항목 22-6, 22-7, 22-8 ~ 22-11
  - 기본 22-8, 22-10
  - 선택 22-9, 22-10
  - 연결 22-11
  - 요청에 응답 22-10
  - 이벤트 핸들러 22-7
  - 추가 22-7
  - 페이지 프로듀서 22-18
  - 활성화 및 비활성화 22-9
- 장치 컨텍스트 8-1, 24-7
- 재배치 가능 코드 10-18
- 전역 오프셋 테이블 (GOT) 10-18
- 절대 주소 10-7
- 점진적 페치 20-30
- 정렬 순서 12-4
  - 내림차순 20-17
  - 인덱스 20-39
  - 클라이언트 데이터셋 20-16
- 정사각형, 그리기 32-9
- 정적 메소드 25-7
- 정적 텍스트 3-28
- 정적 텍스트 컴포넌트 3-28
- 제약 조건
  - 데이터 17-20 ~ 17-21
  - 생성 17-20 ~ 17-21
  - 레코드 수준 20-13
  - 사용 불가능 20-13
  - 컨트롤 6-4
  - 클라이언트 데이터셋 20-13 ~ 20-14
  - 필드 수준 20-13
- 조건부 컴파일 10-16, 10-17
- 조상 클래스 3-5, 3-8, 25-3 ~ 25-4
  - 기본 25-4
- 조회 값 15-17
- 조회 목록 상자 15-2, 15-11 ~ 15-12
  - 보조 데이터 소스 15-11
  - 조회 필드 15-11
- 조회 콤보 상자 15-2, 15-11 ~ 15-12
  - 데이터 그리드 15-20

- 보조 데이터 소스 15-11
- 조회 필드 15-11
- 채우기 15-20
- 조회 필드 15-11, 17-6
  - 값을 캐시로 저장 17-9
  - 데이터 그리드 15-20
  - 성능 17-9, 17-10
  - 정의 17-8 ~ 17-10
  - 지정 15-20
  - 프로그램으로 값 제공 17-9
- 좌표
  - 현재 그리기 위치 8-24
- 주소
  - 소켓 연결 23-4
- 줄 끝 문자 10-12
- 중요 섹션 9-7
  - 사용 경고 9-7, 9-8
- 중첩 디테일 17-25, 20-12 ~ 20-13
  - 요구불 페치 20-29, 21-5
- 중첩 테이블 17-25, 20-12 ~ 20-13
  - 파일 기반 애플리케이션 20-12
- 증가 검색 15-10
- 지속성
  - 트랜잭션 14-4
- 지시어 10-17
  - \$DENYPACKAGEUNIT 컴파일러 지시어 11-10
  - \$DESIGNONLY 컴파일러 지시어 11-10
  - \$ELSEIF 10-17
  - \$ENDIF 10-17
  - \$G 컴파일러 지시어 11-10
  - \$H 컴파일러 지시어 4-26, 4-35
  - \$IF 10-17
  - \$IFDEF 10-16
  - \$IFEND 10-17
  - \$IFDEF 10-16
  - \$IMPLICITBUILD 컴파일러 지시어 11-10
  - \$IMPORTEDDATA 컴파일러 지시어 11-10
  - \$MESSAGE 컴파일러 지시어 10-18
  - \$P 컴파일러 지시어 4-35
  - \$RUNONLY 컴파일러 지시어 11-10
  - \$SONAME 컴파일러 지시어 5-4
  - \$SOPREFIX 컴파일러 지시어 5-4

- \$SOSUFFIX 컴파일러 지시어 5-4
- \$SOVERSION 컴파일러 지시어 5-4
- \$V 컴파일러 지시어 4-35
- \$WEAKPACKAGEUNIT 컴파일러 지시어 11-10
- \$X 컴파일러 지시어 4-35
- default 26-12
- LEpath 컴파일러 지시어 11-11
- LNpath 컴파일러 지시어 11-11
- Lupackage 컴파일러 지시어 11-11
- override 25-8
- protected 27-5
- public 27-5
- published 26-3, 27-5
- stored 26-13
- Z 컴파일러 지시어 11-11
- 가상 25-8
- 기본 31-3
- 동적 25-9
- 문자열 관련 4-35
- 조건부 컴파일 10-16
- 패키지 관련 11-10
- 지역화 12-6
  - 개요 12-1
- 지원 서비스 1-2
- 진행 표시줄 3-29
- 집계
  - 클라이언트 데이터셋 20-20 ~ 20-22
- 집계 필드 17-6
  - 정의 17-10
  - 표시 17-10
- 집계 (aggregate) 필드 20-22
- 집합 26-2
- 집합 타입 26-2
- 짧은 문자열 4-26

## 차

- 차단 연결 23-10
  - 비차단 연결과 비교 23-9
- 참조
  - 패키지 11-3
  - 폼 6-2
- 참조 무결성 14-4
- 참조 카운팅
  - COM 객체 4-20
- 참조 필드 17-21, 17-26
  - 표시 15-22
- 창 3-23

- 메시지 처리 33-4
- 크기 조정 3-23
- 핸들 24-4
- 찾기
  - 파일 4-44
- 채우기 패턴 8-7, 8-8
- 추상 클래스 24-3
- 추출된 집계
  - 집계 필드 17-10
- 출력 매개변수 18-8, 20-26

## 카

- 캔버스 24-7, 29-1
  - 개요 8-1 ~ 8-3
  - 그리기와 색칠 비교 8-4
  - 기본 드로잉 톨 32-5
  - 도형 추가 8-10 ~ 8-11, 8-14
  - 선 그리기 8-5, 8-9 ~ 8-10, 8-27 ~ 8-28
  - 이벤트 핸들러 8-25
  - 팬 너비 변경 8-6
  - 일반적인 속성, 메소드 8-3
  - 화면 새로 고침 8-2
- 캡슐화 3-5
- 커서 16-9
  - 복제 20-24
  - 이동 16-10, 16-11, 20-3, 20-5
  - 마지막 행으로 이동 16-10, 16-11
  - 조건 16-15
  - 첫 번째 행으로 이동 16-10
  - 첫 행으로 이동 16-12
- 컨텍스트 ID 5-17
- 컨텍스트 메뉴
  - 메뉴 디자이너 6-28
  - 툴바 6-38
  - 항목 추가 30-13 ~ 30-14
- 컨텍스트 번호 (도움말) 3-29
- 컨텐츠 속성
  - 웹 응답 객체 22-15
- 컨텐츠 프로듀서 22-7, 22-16
  - 이벤트 처리 22-18, 22-19
- 컨트롤 3-3, 3-12 ~ 3-31
  - data-aware 15-1 ~ 15-29
  - owner-draw 7-7
  - 선언 7-7
  - widget 기반 24-4
  - 그래픽 29-2, 32-1 ~ 32-10

- 그리기 32-3 ~ 32-9
  - 생성 24-4, 32-3
  - 이벤트 29-5
- 그룹화 3-26 ~ 3-27
- 다시 그리기 32-7, 32-9, 33-4
- 데이터 찾아보기 34-1 ~ 34-7
- 데이터 편집 34-8 ~ 34-12
- 도형 32-7
- 변경 24-3
- 사용자 지정 24-4
- 크기 조정 29-4, 33-4
- 포커스 받기 24-4
- 표준
  - 데이터 표시 15-4, 17-16, 17-16
- 하위 분류 24-4
- 컨트롤 다시 그리기 32-7, 32-9, 33-4
- 컨트롤 크기 조정 3-23, 13-8, 33-4
- 그래픽 29-4
- 컨트롤 하위 분류 24-4
- 컴파일 타임 오류
  - override 지시어 25-8
- 컴파일러 옵션 5-2
- 컴파일러 지시어 10-17
  - 문자열 4-35
  - 패키지 특정 11-10
- 컴포넌트 3-2, 3-12 ~ 3-31, 24-1, 25-1, 26-3
  - data-aware 34-1
  - 그룹화 3-26 ~ 3-27
  - 기준 유닛에 추가 24-10
  - 년비주얼 24-5, 24-11
  - 년비주얼 (nonvisual) 3-32
  - 더블 클릭 30-15
  - 더블 클릭하기 30-13
  - 데이터 찾아보기 34-1 ~ 34-7
  - 데이터 편집 34-8 ~ 34-12
  - 등록 24-11, 30-2
  - 변경 31-1 ~ 31-3
  - 사용자 지정 5-11, 24-3, 26-1, 27-1
  - 생성 24-2, 24-8
  - 설치 11-5 ~ 11-6, 30-16
  - 인터페이스 25-4
    - 디자인 타임 25-6
    - 런타임 25-6

- 유닛에 추가 24-10
- 이름 재지정 3-7 ~ 3-8
- 이벤트에 응답 27-6, 27-8, 27-9, 34-7
- 종속성 24-5 ~ 24-6
- 초기화 26-13, 32-7, 34-6
- 추상 24-3
- 컨텍스트 메뉴 30-13, 30-13 ~ 30-14
- 컴포넌트 팔레트에 추가 30-1
- 크기 조정 3-23
- 테스트 24-12, 24-14
- 파생된 클래스 24-3, 24-11, 32-2
- 팔레트 비트맵 30-4
- 패키지 30-16
- 컴포넌트 그룹화 3-26 ~ 3-27
- 컴포넌트 패러 24-5
- 컴포넌트 마법사 24-8
- 컴포넌트 사용자 지정 26-1
- 컴포넌트 템플릿 5-11, 5-12, 25-2
- 프레임 6-12, 6-13
- 컴포넌트 팔레트 3-18
- dbExpress 페이지 14-1, 18-2
- 컴포넌트 추가 11-5, 30-1, 30-4
- 페이지 목록 3-18
- 프레임 6-12
- 컴포넌트 편집기 30-13 ~ 30-16
- 기본 30-13
- 등록 30-16
- 코드 28-4
- 포팅 가능 10-13, 10-15
- 코드 에디터
- 패키지 열기 11-7
- 표시 30-15
- 코드 최적화
- 인터페이스 4-24
- 코드 편집 2-2
- 콘솔 애플리케이션 5-3
- CGI 22-5
- 콤보 상자 3-25, 10-6, 15-2, 15-11
- data-aware 15-10 ~ 15-12
- owner-draw 7-7
- measure-item 이벤트 7-9
- 조회(look up) 15-20

- 컬바
- 디자인 6-32 ~ 6-38
- 숨기기 6-38
- 쿼리 18-3 ~ 18-6
- HTML 테이블 22-22
- TSQLClientDataSet 20-36
- TSQLDataSet 18-4
- TSQLQuery 18-4
- 대표 문자 18-6
- 마스터/디테일 관계 18-14 ~ 18-15
- 매개변수 18-4 ~ 18-6
- 클라이언트 데이터셋 20-28
- 웹 애플리케이션 22-22
- 자동으로 생성 18-6
- 파일에서 로드 18-4
- 필터와 비교 16-17
- 쿼리 부분(URL) 22-2
- 크로스 플랫폼 애플리케이션 10-1 ~ 10-28
- 클라이언트
- 작업 목록 6-14
- 클라이언트 애플리케이션
- 소켓 23-1
- 인터페이스 23-2
- 쿼리 제공 21-6
- 클라이언트 데이터셋 14-6, 20-1 ~ 20-42
- 검색 20-3 ~ 20-5
- 계산된(calculated) 필드 20-19 ~ 20-20
- 기존 테이블 복사 20-40
- 내부 소스 데이터셋 사용 20-35 ~ 20-38
- 다른 데이터셋에 연결 20-25 ~ 20-35
- 데이터 공유 20-24
- 데이터 그룹화 20-18 ~ 20-19
- 데이터 복사 20-23 ~ 20-24
- 데이터 집계 20-20 ~ 20-22
- 데이터 합병 20-24
- 레코드 업데이트 20-31 ~ 20-33
- 레코드 버퍼링 14-9 ~ 14-11
- 레코드 새로 고침 20-33 ~ 20-34
- 레코드 제한 20-5 ~ 20-9, 20-28

- 마스터/디테일 관계 20-10 ~ 20-13
- 매개변수 20-26 ~ 20-28
- 변경 내용 저장 20-15 ~ 20-16
- 변경 취소 20-15
- 변경 합병 20-41
- 업데이트 오류 해결 20-32, 20-32 ~ 20-33
- 업데이트 적용 20-31 ~ 20-33, 20-37
- 인덱스 20-16 ~ 20-40
- 추가 20-16 ~ 20-17
- 인덱스 삭제 20-17
- 인덱스 전환 20-17
- 읽기 전용 20-14
- 제약 조건 20-13 ~ 20-14
- 쿼리 제공 20-28 ~ 20-29
- 탐색 20-2 ~ 20-5
- 테이블 생성 20-38 ~ 20-41
- 파일 기반 애플리케이션 20-38 ~ 20-42
- 파일 로드 20-41
- 파일 저장 20-42
- 편집 14-7, 20-14 ~ 20-16
- 프로바이더 20-25 ~ 20-35
- 프로바이더 지정 20-25, 20-25 ~ 20-26
- 클라이언트 소켓 23-3, 23-6
- 서버 식별 23-6
- 서버에 연결 23-8
- 서비스 요청 23-6
- 소켓 객체 23-7
- 속성 23-6
- 오류 메시지 23-7
- 이벤트 처리 23-8
- 호스트 지정 23-4
- 클라이언트 연결 23-3
- 열기 23-6
- 요청 승인 23-7
- 포트 번호 23-5
- 클라이언트 요청 22-3 ~ 22-4
- 클라이언트/서버 애플리케이션 5-4
- 클래스 24-2, 24-3, 25-1, 26-2
- public 부분 25-6
- published 부분 25-6
- 계층 구조 25-3
- 기본 25-4
- 매개변수로 전달 25-9

- 상속 25-7
- 새로 파생 25-2, 25-8
- 생성 25-1 ~ 25-3
  - 일반 구문 4-2
- 속성 26-2
- 속성 편집기 30-4
- 엑세스 25-4 ~ 25-7, 32-6
- 인스턴스화 25-2
- 자손 25-3 ~ 25-4, 25-8
- 정의 24-11, 25-2
  - 가상 메소드 25-8
  - 정적 메소드 25-7
- 조상 25-3 ~ 25-4
- 추상 24-3
- 파생 25-8
- 클래스 완성(class completion) 4-2
- 클래스 타입 선언 4-1
- 클래스 파생 25-8
- 클래스 포인터 25-9
- 클래스 필드 32-4
  - 선언 32-5
  - 이름 지정 27-3
- 클래스에서 상속 3-8, 25-7
- 클릭 이벤트 8-24, 8-25, 27-1, 27-2, 27-8
- 클립보드 7-3, 7-4, 15-9
  - 그래픽 8-21 ~ 8-22
  - 그래픽 객체 8-3, 15-9
  - 선택 부분 지우기 7-5
  - 이미지 테스트 8-22
  - 형식
    - 추가 30-13, 30-15
- 키
  - 검색 20-5
  - 범위 설정 20-8
- 키 필드
  - 여러 키 필드 20-5, 20-7, 20-8
- 키보드 매핑 12-4
- 키보드 이벤트 27-4, 27-10
- 키워드
  - protected 27-5
- 키워드 방식 도움말 5-17

## 타

- 타원
  - 그리기 8-10, 32-9
- 타입
  - Char 12-2
  - 사용자 정의 32-3
  - 속성 26-2, 26-9, 30-6
- 타입 변환 오류 4-9
- 타입 선언

- 속성 32-3
- 열거 타입 8-12
- 탐색기 15-2, 15-26 ~ 15-29, 16-9, 16-10
  - 데이터 삭제 16-24
  - 데이터셋끼리 공유 15-29
  - 도움말 힌트 15-28
  - 버튼 15-26
  - 버튼 사용 가능/사용 불가능 15-27
  - 편집 16-22
- 탭
  - draw-item 이벤트 7-10
- 탭 모양의 페이지 3-27
- 탭 집합 3-27
- 탭 컨트롤 3-27
  - owner-draw 7-7
- 터미널 유형 10-13
- 테두리
  - 패널 3-26
- 테스트
  - 값 26-7
  - 컴포넌트 24-12, 24-14
- 테이블 18-6 ~ 18-7
  - TSQDataSet 18-6 ~ 18-7
  - TSQTable 18-7
  - 그리드에 표시 15-16
  - 데이터베이스 이외의 그리드 3-29
  - 레코드 삽입 16-23 ~ 16-24, 16-26
  - 레코드 정렬 18-6, 18-7
  - 마스터/디테일 관계 18-15
  - 인덱스 18-7
- 테이블 HTML 태그
  - (<TABLE>) 22-17
- 테이블 프로듀서 22-21 ~ 22-23
  - 속성 설정 22-21
- 텍스트
  - owner-draw 컨트롤 7-7
  - 국제화 12-4
  - 복사, 잘라내기, 붙여넣기 7-4
  - 사용 7-1 ~ 7-7
  - 삭제 7-5
  - 선택 7-3, 7-3 ~ 7-4
  - 컨트롤의 텍스트 7-1
- 텍스트 컨트롤 3-18 ~ 3-19
- 템플릿 5-8, 5-10
  - 메뉴 6-22, 6-29, 6-30
  - 로드 6-30
  - 웹 애플리케이션 22-6, 22-24
  - 컴포넌트 5-11, 5-12

- 프로그래밍 5-8
- 토글 6-35, 6-37
- 통신 23-1
  - 표준 22-1
  - 프로토콜 22-1, 23-2
- 통화
  - 국제화 12-4
  - 형식 12-4
- 투명한 배경 12-4
- 툴
  - 버튼 6-36
  - 그룹화/그룹 해제 6-37
  - 도움말 얻기 6-38
  - 사용 불가능 6-36
  - 여러 행 6-37
  - 이미지 추가 6-36
  - 줄 바꿈(wrapping) 6-37
  - 초기 상태, 설정 6-37
  - 토글로 사용 6-37
- 툴바 3-23, 6-32
  - owner-draw 7-7
  - 기본 드로잉 툴 6-34
  - 디자인 6-32 ~ 6-38
  - 버튼 사용 불가능 6-36
  - 버튼 삽입 6-33 ~ 6-35, 6-36
  - 숨기기 6-38
  - 스피드 버튼 3-22
  - 알기 쉬운 6-37
  - 여백 설정 6-34
  - 작업 목록 6-14
  - 추가 6-35 ~ 6-37
  - 컨텍스트 메뉴 6-38
  - 패널 추가 6-33 ~ 6-35
- 툴팁 도움말 3-29
- 트랙 표시줄 3-20
- 트랜잭션 14-3 ~ 14-4, 19-7 ~ 19-10
  - SQL 명령 사용 19-7
  - 끝 19-8 ~ 19-9
  - 롤 백 19-9
  - 분리 14-4
  - 레벨 19-9 ~ 19-10
  - 시작 19-8
  - 암시적 19-8
  - 오버랩 19-8
  - 원자성 14-3
  - 일관성 14-3
  - 종료 19-8 ~ 19-9
  - 중첩 19-9
  - 지속성 14-4
  - 커밋 19-9
  - 클라이언트 데이터셋 19-8
- 트리 뷰 3-25
  - owner-draw 7-7
- 트리거 14-4

## 파

파일 4-36 ~ 4-45  
date-time 루틴 4-40  
그래픽 8-19 ~ 8-21,  
29-3  
루틴  
date-time 루틴 4-40  
런타임 라이브러리 4-36  
모드 4-42  
문자열 4-44  
바이트 복사 4-45  
사용 4-36 ~ 4-45  
삭제 4-37  
속성 4-39  
쓰기 4-43  
웹으로 보내기 22-15  
위치 4-45  
이름 재지정 4-39  
읽기 4-43  
찾기 4-37, 4-44  
처리 4-36 ~ 4-40  
크기 4-45  
타입  
I/O 4-40  
텍스트 4-40  
타입이 지정되지 않은  
파일 4-40  
타입이 지정된 파일 4-40  
핸들 4-42  
파일 I/O  
타입 4-40  
파일 권한 10-13  
파일 기반 애플리케이션  
14-7 ~ 14-8  
클라이언트 데이터셋  
20-38 ~ 20-42  
파일 끝 문자 10-12  
파일 스트림 4-41 ~ 4-45  
CLX 스트리밍 4-41  
TMemoryStream 4-41  
끝 표시 4-45  
생성 4-41  
열기 4-41  
예외 4-43  
크기 변경 4-45  
파일 I/O 4-41 ~ 4-45  
포팅 가능 4-40  
핸들 얻기 4-40  
팔레트 비트맵 파일 30-4  
팝업 메뉴 7-6 ~ 7-7  
드롭다운 메뉴 6-25  
표시 6-27  
패널 3-26  
스피드 버튼 3-22  
스피드 버튼 추가 6-34

액자 모양 3-31  
폼 위에 추가 6-33  
패턴 8-9  
패키지 11-1 ~ 11-12,  
30-16  
Constains 목록 11-6,  
11-9  
Requires 목록 11-6,  
11-8  
RPM 13-2  
공유 객체 파일 11-2  
국제화 12-5, 12-6  
기본 설정 11-7  
디자인 전용 옵션 11-6  
디자인 타임 11-1, 11-4 ~  
11-6  
런타임 11-1, 11-2 ~  
11-4, 11-7  
사용 5-4  
사용자 지정 11-4  
생성 5-3, 11-6 ~ 11-12  
설치 11-5 ~ 11-6  
소스 파일 11-2, 11-12  
애플리케이션 배포 11-2,  
11-12  
애플리케이션에서 사용  
11-2 ~ 11-4  
옵션 11-7  
요청 목록 30-16  
중복 참조 11-9  
참조 11-3  
컴파일 11-9 ~ 11-12  
옵션 11-10  
컴파일러 지시어 11-10  
컴포넌트 30-16  
파일명 확장자 11-1  
편집 11-7  
포함 목록 30-16  
패키지 파일 13-4  
페이지 컨트롤 3-27  
페이지 프로듀서 22-16 ~  
22-20  
data-aware 22-20  
연결 22-19  
이벤트 처리 22-18,  
22-19  
템플릿 변환 22-18  
펜 8-5, 32-5  
그리기 모드 8-28  
기본 설정 8-5  
너비 8-6  
변경 32-7  
브러시 8-5  
색상 8-5  
스타일 8-6

위치 파악 8-7  
위치, 설정 8-7, 8-24  
펜 너비 지원 8-6  
편집 모드 16-22  
취소 16-22  
편집 컨트롤 3-18 ~ 3-19,  
7-1, 15-2, 15-8  
여러 줄 15-9  
텍스트 선택 7-3, 7-4  
포인트  
기본 속성 값 26-12  
메소드 27-2, 27-3, 27-8  
클래스 25-9  
포커스 24-4  
필드 17-15  
포트 23-5  
서버 소켓 23-7  
서비스 23-2  
여러 연결 23-5  
클라이언트 소켓 23-6  
포팅 가능 코드 10-15  
포함 목록(패키지) 30-16  
폼 3-17  
데이터 검색 6-8 ~ 6-11  
데이터 동기화  
여러 폼 15-3  
드릴 다운 15-14  
로컬 변수로 생성 6-6  
런타임에 생성 6-5  
마스터/디테일 테이블  
15-14  
메모리 관리 6-5  
메인 폼 6-1  
모달 6-4  
모달리스 6-4, 6-6  
새 객체 타입 3-5 ~ 3-7  
속성 쿼리  
예제 6-8  
스크롤 영역 3-27  
연결 6-2  
유닛 참조 추가 6-2  
이벤트 핸들러 공유 8-15  
인수 전달 6-7  
인스턴스화 3-6  
전역 변수 6-5  
참조 6-2  
표시 6-5  
프로젝트에 추가 3-8,  
6-1 ~ 6-2  
필드 추가 8-26 ~ 8-27  
폼 연결 6-2  
폼 파일 3-7, 3-15, 12-6  
폼과 대화 상자 공유 5-8 ~  
5-11

표 형식으로 표시(그리드)  
3-29  
표의 문자 12-2  
  약자 12-4  
표준 이벤트 27-4, 27-4 ~  
  27-6  
  사용자 지정 27-6  
표현식 17-20  
풍선 도움말 15-28  
프레임 5-11, 6-11 ~ 6-13  
  공유 및 배포 6-13  
  그래픽 6-13  
  리소스 6-13  
  컴포넌트 템플릿 6-12,  
    6-13  
프로바이더 14-10, 21-1 ~  
  21-12  
  데이터셋 관련 21-2  
  업데이트 적용 21-4,  
    21-7 ~ 21-11  
  업데이트 스크린 21-10  
  오류 처리 21-11  
  원격 20-26, 21-2  
  지역 20-26, 21-2  
  클라이언트 데이터셋  
    20-25 ~ 20-35  
  클라이언트 생성 이벤트  
    21-12  
  트랜잭션 19-8  
프로바이더, 내부 20-37  
프로세스 9-1, 10-13  
프로시저 24-7, 27-3  
  속성 설정 30-9  
  이름 지정 28-2  
프로젝트  
  폼 추가 6-1 ~ 6-2  
프로젝트 옵션 5-2  
  기본 5-2  
프로젝트 템플릿 5-10  
프로젝트 파일  
  변경 2-2  
프로토콜  
  인터넷 22-1, 23-1  
플라이바이 도움말 3-29  
플래그 34-4  
필드 17-1 ~ 17-26  
  Null 값 16-26  
  값 변경 15-6  
  값 업데이트 15-5  
  값 지정 16-26  
  값 표시 15-10, 17-16  
  기본 형식 17-13  
  기본값 17-20  
  데이터 검색 17-16

데이터 입력 16-23,  
  17-13  
데이터베이스 34-5, 34-7  
상호 배타적인 옵션 15-2  
속성 17-1  
  숨겨진 21-4  
  영구적 열 15-17  
  유효한 데이터 제한 17-20  
    ~ 17-21  
  읽기 전용 15-5  
  추상 데이터 타입 17-21 ~  
    17-26  
  폼에 추가 8-26 ~ 8-27  
  현재 값 17-20  
  활성화 17-15  
필드 객체 17-1 ~ 17-26  
  동적 17-2 ~ 17-3  
  영구적 필드와 비교  
    17-2  
  삭제 17-10 ~ 17-11  
  속성 17-1, 17-11 ~  
    17-14  
  런타임 17-12  
  액세스 값 17-18 ~ 17-20  
  영구적 17-3 ~ 17-14  
  동적 필드와 비교 17-2  
  이벤트 17-14  
  정의 17-5 ~ 17-10  
  표시 및 편집 속성 17-11  
필드 속성  
  데이터 패킷 21-5  
필드 정의  
  클라이언트 데이터셋 생성  
    20-39  
필드 타입  
  변환 17-15, 17-17 ~  
    17-18  
필터 16-8, 16-17 ~ 16-20  
  대소문자 구별 16-20  
  런타임 시 설정 16-20  
  문자열 비교 16-20  
  범위와 비교 20-6  
  빈 필드 16-18  
  사용 가능/사용 불가능  
    16-17  
  연산자 16-18  
  정의 16-16 ~ 16-20  
  쿼리와 비교 16-17  
  클라이언트 데이터셋  
    매개변수 사용 20-28  
  텍스트 필드의 옵션 16-20

## 하

하드웨어 예외 4-9  
하위 메뉴 6-25  
하이퍼텍스트 링크  
  HTML에 추가 22-17  
한정자 3-8 ~ 3-9  
할당 문 26-2  
함수 24-7  
  속성 읽기 26-6, 30-6,  
    30-8  
  이름 지정 28-2  
  이벤트 27-3  
해결 21-1  
핸들 4-36  
  소켓 연결 23-6  
행 3-29  
헤더  
  HTTP 요청 22-2  
  owner-draw 7-7  
헤더 컨트롤 3-27  
현재 월 반환 33-7  
확인 목록 상자 3-24  
확인란 3-22, 15-2  
  data-aware 15-12 ~  
    15-13  
호스트 23-4  
  URL 22-2  
  주소 23-4  
호스트 이름 23-4  
  IP 주소 23-5  
홈 디렉토리 10-13  
화면  
  새로 고침 8-2  
  해상도 13-7  
  프로그래밍 13-7, 13-8  
힌트 3-29  
힙 오류 4-9

## 기호

&(앰퍼샌드) 문자 6-25  
... (생략) 버튼 15-21

## 숫자

2계층 애플리케이션 14-2,  
  14-8, 14-11, 20-35  
2바이트 문자 코드 12-2

## A

Abort 메소드 16-28  
Acquire 메소드 9-7  
Actions 속성 22-7  
Active 속성  
  데이터셋 16-2, 16-4,  
    18-10

- 서버 소켓 23-7
- 클라이언트 소켓 23-6
- ActiveAggs 속성 20-22
- ActiveX 데이터 객체
  - ADO 컴포넌트, ADO 데이터셋, ADO 객체 참조
- ActnList 유닛 6-19
- Add Fields 대화 상자 17-4
- Add To Repository 명령 5-9
- Add 메소드
  - 메뉴 6-32
  - 문자열 3-36
  - 영구적 열 15-18
- AddIndex 메소드 20-17
- Additional 페이지 (컴포넌트 팔레트) 3-18
- AddObject 메소드 3-37
- AddRef 메소드 4-20, 4-24
- AddStrings 메소드 3-36, 3-37
- ADT 필드 17-21, 17-22 ~ 17-23
  - 영구적 필드 17-22 ~ 17-23
  - 평평하게 하기 (flattening) 15-21
  - 표시 15-21, 17-22
- AfterApplyUpdates 이벤트 20-34, 21-7
- AfterClose 이벤트 16-5
- AfterConnect 이벤트 19-5
- AfterDisconnect 이벤트 19-5
- AfterDispatch 이벤트 22-8, 22-11
- AfterGetRecords 이벤트 21-7
- AggFields 속성 20-22
- Aggregates 속성 20-20, 20-22
- Align 속성 6-4
  - 메모 3-19
  - 상태 표시줄 3-28
  - 텍스트 컨트롤 7-1
  - 패널 6-33
- Alignment 속성 3-23
  - data-aware 메모 컨트롤 15-9
  - 데이터 그리드 15-19
  - 상태 표시줄 3-29
  - 열 헤더 15-20
  - 필드 17-11
- AllowAllUp 속성 3-22
  - 스피드 버튼 6-35
  - 툴 버튼 6-37
- AllowGrayed 속성 3-22

- alTop 상수 6-33
- AnsiChar 4-25
- AnsiString 4-27
- Apache 웹 서버 애플리케이션 5-6
- Apache 프로그램
  - 만들기 22-5
- Append 메소드 16-23, 16-24
  - Insert와 비교 16-23
- AppendRecord 메소드 16-26
- Application 변수 6-3
- ApplyRange 메소드 20-9
- ApplyUpdates 메소드 10-27
  - 다계층 애플리케이션 20-31
  - 클라이언트 데이터셋 20-31, 20-31 ~ 20-32, 21-3
  - 프로바이더 20-32, 21-7
- AppServer 속성 20-29, 21-2
- Arc 메소드 8-4
- AS\_ApplyUpdates 메소드
  - 프로바이더 21-3
- AS\_DataRequest 메소드 21-3
- AS\_Execute 메소드 21-3
- AS\_GetParams 메소드 21-3
- AS\_GetProviderNames 메소드 21-3
- AS\_GetRecords 메소드 21-3
- AS\_RowRequest 메소드 21-3
- Assign Local Data 명령 20-23, 20-41
- Assign 메소드
  - 문자열 목록 3-37
- AssignedValues 속성 15-21
- AssignValue 메소드 17-15
- Associate 속성 3-21
- AutoAccept 속성 23-7
- AutoCalcFields 속성 16-28
- AutoDisplay 속성 15-9
- AutoEdit 속성 15-5
- AutoPopup 속성 6-38
- AutoSize 속성 3-19, 6-4, 13-8, 15-8

## B

- BaseCLX 10-4
- .bashrc 10-13
- BeforeApplyUpdates 이벤트 20-34, 21-7

- BeforeClose 이벤트 16-5, 16-25
- BeforeDisconnect 이벤트 19-5
- BeforeDispatch 이벤트 22-8, 22-9
- BeforeGetRecords 이벤트 21-7
- BeforeUpdateRecord 이벤트 20-38, 21-11
- BeginRead 메소드 9-8
- BeginWrite 메소드 9-8
- Beveled 속성 3-23
- bin 디렉토리 10-14
- BLOB 필드 15-2
  - 값 표시 15-9
  - 그래픽 보기 15-9
  - 요구불 페치 20-29, 21-5
- BLOBs 15-9
- BlockMode 속성 23-9, 23-10
- bmBlocking 23-10
- bmNonBlocking 23-9
- bmThreadBlocking 23-9, 23-10
- Bof 속성 16-10, 16-12
- Bookmark 속성 16-13
- BookmarkValid 속성 16-13
- BorderWidth 속성 3-26
- Brush 속성 3-30, 8-3, 8-7, 29-2
- BrushCopy 메소드 29-2
- ButtonStyle 속성
  - 데이터 그리드 15-19, 15-20, 15-21
- ByteType 4-29

## C

- CachedUpdates 속성 10-27
- Cancel 메소드 16-6, 16-7, 16-22, 16-26
- Cancel 속성 3-21
- CancelBatch 메소드 10-27
- CancelRange 메소드 20-9
- CancelUpdates 메소드 10-27
- CanModify 속성
  - 데이터 그리드 15-24
  - 데이터셋 16-6, 16-22
- Canvas 속성 3-31, 24-7
- Caption 속성
  - TForm 3-28
  - 그룹 상자와 라디오 그룹 3-26
  - 레이블 3-28

- 열 헤더 15-20
- 잘못된 입력 6-23
- CaseInsensitiveFields 속성 20-40
- CellRect 메소드 3-29
- Cells 속성 3-30
- CGI 프로그램 5-6, 22-5
- 만들기 22-5
- Change 메소드 34-12
- ChangeBounds 속성 10-19
- ChangeCount 속성 10-27
- CHANGEINDEX 20-16
- ChangeScale 속성 10-19
- Char 데이터 타입
  - 4-25, 12-2
- Checked 속성 3-22
- CheckSynchronize 루틴 9-5
- Chord 메소드 8-4
- Clear 메소드
  - 문자열 목록 3-36, 3-37
  - 필드 17-15
- ClearSelection method 7-5
- Click 메소드 27-2
  - 오버라이드 27-6, 33-10
- CloneCursor 메소드 20-24
- Close 메소드
  - TSQLConnection 19-5
  - 데이터셋 16-2, 16-5
- CloseDataSets 메소드 19-11
- CLX
  - TComponent 분기 3-14
  - TControl 분기 3-15
  - TObject 분기 3-13
  - TPersistent 분기 3-13
  - TWidgetControl 분기 3-16
  - VCL과 비교 10-4
  - 개요 3-1 ~ 3-4
  - 객체 3-2
  - 객체 생성자 10-11
  - 메인 스레드 9-3
- Code Editor
  - 개요 2-3
- Code Insight
  - 템플릿 5-8
- Color 속성 3-28, 3-30
  - 데이터 그리드 15-19
  - 브러시 8-7, 8-8
  - 열 헤더 15-20
  - 펜 8-5
- ColorChanged 속성 10-19
- Cols 속성 3-30
- Columns Editor
  - 열 삭제 15-18
  - 열 재정렬 15-19
  - 영구적인 열 생성 15-17

- Columns 속성 3-24, 15-17
  - 그리드 15-15
  - 라디오 그룹 3-26
- ColWidths 속성 3-30
- COM 객체
  - 수명 관리 4-20
- Command Text 편집기 18-4
- CommandText 속성 18-4,
  - 18-6, 18-7, 18-12, 20-28
- CommandType 속성 18-3,
  - 18-6, 18-7, 18-12
- Commit 메소드 19-9
- CommitUpdates 메소드 10-27
- Common Controls 페이지
  - (컴포넌트 팔레트) 3-18
- CompareBookmarks 메소드 16-13, 16-14
- Component 팔레트
  - Data Controls 페이지 15-1, 15-2
- Connected 속성
  - TSQLConnection 19-4, 19-5
  - 데이터베이스 컴포넌트 19-4
- Connection Editor 19-3 ~ 19-4
- ConnectionName 속성 19-3
- Console Wizard 5-3
- Constairs 목록
  - (패키지) 11-6, 11-9
- ConstraintErrorMessage 속성 17-11, 17-20
- Constraints 속성 20-13
- Content 메소드
  - 페이지 프로듀서 22-17
- ContentFromStream 메소드
  - 페이지 프로듀서 22-17
- ContentFromString 메소드
  - 페이지 프로듀서 22-17
- ContentStream 속성
  - 웹 응답 객체 22-15
- ContextHelp 5-19
- CopyFrom 함수 4-45
- CopyMode 속성 29-2
- CopyRect 메소드 8-4, 29-2, 29-4
- CopyToClipboard 메소드 7-4
  - data-aware 메모 컨트롤 15-9
  - 그래픽 15-9
- Count 속성 3-35

- Create DataSet 명령 20-39
- Create Submenu 명령
  - (Menu designer) 6-29
- Create Submenu 명령
  - (메뉴 디자이너) 6-26
- Create 메소드 3-11
- CreateDataSet 메소드 20-39
- CreateParam 메소드 20-27
- CreateSuspended 매개변수 9-10
- Currency 속성
  - 필드 17-11
- CursorChanged 속성 10-19
- CurValue 속성 17-20, 21-11
- CustomConstraint 속성 17-11, 17-20, 20-13
- CutToClipboard 메소드 7-4
  - data-aware 메모 컨트롤 15-9
  - 그래픽 15-9

## D

---

- Data Controls 페이지
  - (Component 팔레트) 15-1, 15-2
- Data Controls 페이지
  - (컴포넌트 팔레트) 3-18
- Data 속성 20-14, 20-23, 20-41
- data-aware 컨트롤 15-1 ~ 15-29, 17-16, 34-1
  - 공통적인 기능 15-2
  - 그래픽 표시 15-9
  - 그리드 15-14
  - 다시 그리기 사용 불가능 15-6, 16-12
  - 데이터 새로 고침 15-7
  - 데이터 입력 17-13
  - 데이터 찾아보기 34-1 ~ 34-7
  - 데이터 편집 34-8 ~ 34-12
  - 데이터 표시 15-6 ~ 15-7
    - 그리드에서 표시 15-15
  - 현재값 15-8
  - 데이터셋에 연결 15-3
  - 레코드 삽입 16-23
  - 목록 15-2
  - 변경 내용에 응답 34-7
  - 생성 34-1 ~ 34-13
  - 소멸 34-6
  - 읽기 전용 15-8
  - 편집 15-5 ~ 15-6, 16-7, 16-22



- 필드 표시 15-8
- Database 매개변수 19-2
- DataChange 메소드 34-11
- DataCLX 10-4
- DataField 속성 15-10, 34-5
  - 조회 목록 상자와 콤보 상자 15-12
- DataRequest 메소드 20-35, 21-3
- DataSet 속성
  - 데이터 그리드 15-16
  - 프로바이더 21-2
- DataSetCount 속성 19-11
- DataSetField 속성
  - 클라이언트 데이터셋 20-12
- DataSets 속성 19-11
- DataSource 속성
  - data-aware 컨트롤 34-5
  - 데이터 그리드 15-16
  - 데이터 탐색기 15-29
  - 조회 목록 상자와 콤보 상자 15-11
  - 쿼리 18-14
- DataType 속성
  - 매개변수 18-5, 18-9
- Day 속성 33-5
- DBCheckBox 컴포넌트 15-2, 15-12 ~ 15-13
- DBComboBox 컴포넌트 15-2, 15-10 ~ 15-11
- DBEdit 컴포넌트 15-2, 15-8
- dbExpress 10-21 ~ 10-26, 14-1, 19-1, 20-1
  - 드라이버 19-1, 19-2
  - 디버깅 19-13 ~ 19-14
  - 배포 13-5
- dbExpress 페이지(컴포넌트 팔레트) 3-18, 14-1, 18-2
- DBGrid 컴포넌트 15-2, 15-15 ~ 15-26
  - 속성 15-19
  - 이벤트 15-25
- DBGridColumn 컴포넌트 15-15
- DBImage 컴포넌트 15-2, 15-9
- DBListBox 컴포넌트 15-2, 15-10 ~ 15-11
- DBLogDlg 유닛 19-6
- DBLookupComboBox 컴포넌트 15-2, 15-11 ~ 15-12
- DBLookupListBox 컴포넌트 15-2, 15-11 ~ 15-12

- DBMemo 컴포넌트 15-2, 15-9
- DBNavigator 컴포넌트 15-2, 15-26 ~ 15-29
- DBRadioGroup 컴포넌트 15-2, 15-13
- DBText 컴포넌트 15-2, 15-8
- dbxconnections 파일 19-3, 19-3 ~ 19-4
  - 로그인 정보 19-6
- dbxdrivers 파일 19-2
- .dcp 파일 11-2, 11-12
- .dcr 파일 30-4
- DDL 18-11
- DECnet 프로토콜 (Digital) 23-1
- default
  - 지시어 26-12
- Default 속성
  - 작업 항목 22-10
- DEFAULT\_ORDER 20-16
- DefaultColWidth 속성 3-30
- DefaultDrawing 속성 7-7, 15-24
- DefaultExpression 속성 17-20, 20-13
- DefaultRowHeight 속성 3-30
- Delete Templates 대화 상자 6-31
- Delete Templates 명령 (메뉴 디자이너) 6-29, 6-31
- Delete 메소드 16-7, 16-24
  - 문자열 목록 3-36, 3-37
- Delete 명령 (메뉴 디자이너) 6-29
- DELETE 문 21-9
- DeleteFile 함수 4-37
- DeleteIndex 메소드 20-17
- Delphi/Kylix 유닛 비교 10-7
- delphi60dmt 6-30, 6-31
- Delta 속성 20-31
- \$DENYPACKAGEUNIT 컴파일러 지시어 11-10
- deploy.txt 13-9
- DescFields 속성 20-39
- \$DESIGNONLY 컴파일러 지시어 11-10
- Destroy 메소드 3-11
- Dialogs 페이지 (컴포넌트 팔레트) 3-18
- DirtyRead 19-9
- DisableConstraints 메소드 20-13
- DisableControls 메소드 15-6

- DisabledImages 속성 6-36
- DisplayFormat 속성 15-24, 17-11, 17-14
- DisplayLabel 속성 15-17, 17-11
- DisplayWidth 속성 15-16, 17-11
- DLL 10-7, 10-13
  - 국제화 12-6
  - 공유 객체 참조
- DML 18-11
- Down 속성 3-22
  - 스피드 버튼 6-34
- .dpk 파일 11-2, 11-6
- .dpu 파일 11-2, 11-12
- DragMode 속성
  - 그리드 15-19
- Draw 메소드 8-4, 29-2, 29-4
- DrawShape 8-15
- DriverName 속성 19-2
- DropDownCount 속성 3-25
- DropDownMenu 속성 6-38
- DropDownRows 속성
  - 데이터 그리드 15-19, 15-20
  - 조회 콤보 상자 15-12
- DSO 22-4
- dynamic 지시어 25-9

## E

---

- EAbort 4-15
- EBX 레지스터 10-7, 10-19
- Edit 메소드 16-6, 16-22, 30-7, 30-8
- Edit 컨트롤 3-18
- EditFormat 속성 15-24, 17-11, 17-14
- EditKey 메소드 20-3, 20-5
- EditMask 속성 17-13
  - 필드 17-11
- EditRangeEnd 메소드 20-8, 20-9
- EditRangeStart 메소드 20-8, 20-9
- ELF 파일 12-6
- Ellipse 메소드 8-4, 8-11, 29-2
- \$ELSEIF 지시어 10-17
- EmptyDataSet 메소드 20-30
- EmptyStr variable 4-31
- EnableConstraints 메소드 20-14
- EnableControls 메소드 15-6
- Enabled 속성

- data-aware 컨트롤 15-7
- 데이터 소스 15-4, 15-5
- 메뉴 6-32, 7-5
- 스피드 버튼 6-34
- 작업 항목 22-9
- EnabledChanged 속성 10-19
- EndRead 메소드 9-8
- EndWrite 메소드 9-8
- Eof 속성 16-10, 16-11
- EOF 표시 4-45
- EPasswordInvalid 4-17
- EReadError 4-43
- ErrorAddr 변수 4-16
- EWriteError 4-43
- ExecProc 메소드 18-12
- ExecSQL 메소드 18-12
- Execute 메소드 3-31, 9-3
  - TSQLConnection 19-12
  - 클라이언트 데이터셋
    - 20-27, 21-3
    - 프로마이터 21-3
- ExecuteTarget 메소드 6-18
- Expandable 속성 15-22
- Expanded 속성
  - 데이터 그리드 15-20
  - 열 15-22
- Expression 속성 20-21
- ExprText 속성 17-10

## F

- Fetch Params 명령 20-26
- FetchAll 메소드 10-27
- FetchBlobs 메소드 20-31, 21-3
- FetchDetails 메소드 20-31, 21-3
- FetchOnDemand 속성 20-30
- FetchParams 메소드 20-26, 21-3
- Field Link 편집기 18-15, 20-10
- FieldByName 메소드 17-19
- FieldCount 속성
  - 영구적 필드 15-17
- FieldKind 속성 17-11
- FieldName 속성 17-5, 17-11
  - 데이터 그리드 15-20
  - 영구적 필드 15-17
- Fields Editor 5-8
  - 열 재정렬 15-19
  - 클라이언트 데이터셋 생성 20-39
- Fields 속성 17-19
- Fields 편집기 17-3

- 영구적 필드 생성 17-4 ~ 17-5, 17-5 ~ 17-10
- 제목 표시줄 17-4
- 탐색 버튼 17-4
- 필드 객체 삭제 17-10
- 필드 목록 17-4
- FieldValues 속성 17-18
- FileAge 함수 4-40
- FileExists 함수 4-37
- FileGetDate 함수 4-40
- FileName 속성
  - 클라이언트 데이터셋 14-8, 20-41, 20-42
- 파일
  - 파일 스트림 참조
- FileSetDate 함수 4-40
- FillRect 메소드 8-4, 29-2
- Filter 속성 16-17, 16-18 ~ 16-19
- Filtered 속성 16-17
- FilterOptions 속성 16-20
- finally 예약어 29-4
- FindClose 프로시저 4-37
- FindFirst 메소드 16-21
- FindFirst 함수 4-37
- FindKey 메소드 20-3, 20-4
  - EditKey와 비교 20-5
- FindLast 메소드 16-21
- FindNearest 메소드 20-3, 20-4
- FindNext 메소드 16-21
- FindNext 함수 4-37
- FindPrior 메소드 16-21
- First 메소드 16-10
- FixedColor 속성 3-30
- FixedCols 속성 3-30
- FixedRows 속성 3-30
- FloodFill 메소드 29-2
- FocusControl 메소드 17-15
- FocusControl 속성 3-28
- Font 속성 3-19, 3-28, 8-3, 29-2
  - data-aware 메모 컨트롤 15-9
  - 데이터 그리드 15-20
  - 열 헤더 15-20
- FontChanged 속성 10-19
- Footer 속성 22-22
- Found 속성 16-21
- FrameRect 메소드 8-4
- Free 메소드 3-11, 10-11
- FreeBookmark 메소드 16-13, 16-14

## G

- \$G 컴파일러 지시어 11-10, 11-11
- GetAttributes 메소드 30-8
- GetBookmark 메소드 16-13
- GetData 메소드
  - 필드 17-15
- GetFieldByName 메소드 22-12
- GetFieldNames 메소드 19-10
- GetFloatValue 메소드 30-6
- GetGroupState 메소드 20-19
- GetHandle 5-15
- GetHelpFile 5-15
- GetHelpPath 5-15
- GetHelpStrings 5-16
- GetIndexNames 메소드 19-10, 20-18
- GetMethodValue 메소드 30-6
- GetNextPacket 메소드 10-27, 20-30, 21-3
- GetOptionalParam 메소드 20-23, 21-6
- GetOrdValue 메소드 30-6
- GetParams 메소드 21-3
- GetProcedureNames 메소드 19-10
- GetProcedureParams 메소드 19-10
- GetProperties 메소드 30-8
- GetRecords 메소드 21-7
- GetStrValue 메소드 30-6
- GetTableNames 메소드 19-10
- GetValue 메소드 30-6
- GetViewerName 5-14
- Glyph 속성 3-22, 6-34
- GNU 어셈블러 10-15
- GNU make 유틸리티 10-13
- GotoBookmark 메소드 16-13, 16-14
- GotoKey 메소드 20-3, 20-4
- GotoNearest 메소드 20-3, 20-4
- Graphic 속성 8-17, 8-21, 29-3
- GridLineWidth 속성 3-30
- Grouped 속성
  - 툴 버튼 6-37
- GroupIndex 속성 3-22
  - 스피드 버튼 6-34, 6-35
- GUI 애플리케이션 3-17
- GUID 4-21
  - 생성 4-21

## H

\$H 컴파일러 지시어 4-26, 4-35  
Handle 속성 4-42, 23-6, 24-4, 24-5, 29-2  
장치 컨텍스트 8-1  
HandleException 4-14  
HandlesTarget 메소드 6-18  
HasConstraints 속성 17-11  
HasFormat 메소드 8-22  
Header 속성 22-22  
Height 속성 6-4  
TScreen 13-8  
Help Manager 5-13, 5-13 ~ 5-21  
HelpContext 5-19  
HelpContext 속성 3-29  
HelpFile 5-19  
HelpFile 속성 3-29  
HelpIntfs.pas 5-13  
HelpKeyword 5-19  
HelpSystem 5-19  
HelpType 5-19  
Hint 속성 3-29  
Hints 속성 15-28  
HorzScrollBar 3-20  
HotImages 속성 6-36  
HTML 명령 22-16  
데이터베이스 정보 22-20  
생성 22-17  
HTML 문서 22-3  
HTTP 응답 메시지 22-4  
데이터베이스 22-20  
데이터셋 22-22  
데이터셋 페이지 프로듀서 22-20  
테이블 포함시키기 22-22  
테이블 프로듀서 22-21 ~ 22-23  
템플릿 22-16 ~ 22-18  
페이지 프로듀서 22-16 ~ 22-20  
HTML 테이블 22-17, 22-22  
만들기 22-21 ~ 22-23  
속성 설정 22-21  
캡션 22-22  
HTML 템플릿 22-16  
HTMLDoc 속성 22-17  
HTMLFile 속성 22-17  
HTML-transparent 태그  
구문 22-16  
매개변수 22-17  
변환 22-16, 22-18  
이미 정의된 22-17  
HTTP 22-1

개요 22-3 ~ 22-4  
메시지 헤더 22-1  
상태 코드 22-14  
요청 메시지 요청 메시지  
참조  
요청 헤더 22-2, 22-12  
응답 메시지 응답 메시지  
참조  
응답 헤더 22-15  
HyperHelp 5-13  
HyperHelp 뷰어 5-21  
  
I  
I/O 예외 4-9  
IAppServer 인터페이스  
20-29, 20-34, 21-2 ~ 21-3  
원격 프로바이더 21-2  
지역 프로바이더 21-2  
ICustomHelpViewer 5-13, 5-14, 5-15  
구현 5-14  
IETF 프로토콜 및 표준 22-1  
IExtendedHelpViewer 5-13, 5-17  
\$IFDEF 지시어 10-16  
\$IFEND 지시어 10-17  
\$IFDEF 지시어 10-16  
IHelpManager 5-13, 5-20  
IHelpSelector 5-14, 5-18  
IHelpSystem 5-13, 5-20  
Image HTML 태그  
(<IMG>) 22-17  
ImageIndex 속성 6-36  
ImageMap HTML 태그  
(<MAP>) 22-17  
Images 속성  
툴 버튼 6-36  
IMalloc 인터페이스 4-17  
implements 키워드 4-23  
\$SIMPLICITBUILD 컴파일러 지시어 11-10  
ImportedConstraint 속성 17-11  
\$IMPORTEDDATA 컴파일러 지시어 11-10  
Increment 속성 3-21  
Indent 속성 3-25, 6-34, 6-37  
Index 속성  
필드 17-11  
index 예약어 33-6  
IndexDefs 속성 20-17  
IndexFieldCount 속성 20-18

IndexFieldNames 속성 18-7, 20-16  
IndexFields 속성 20-18  
IndexName 속성 18-7, 20-3, 20-17  
IndexOf 메소드 3-36  
INFINITE 상수 9-10  
ini 파일 10-6  
Input Mask 편집기 17-13  
Insert From Resource 명령  
(메뉴 디자이너) 6-29  
Insert From Template 명령  
(메뉴 디자이너) 6-29, 6-30  
Insert Template 대화 상자 6-30  
Insert 메소드 16-23  
Append와 비교 16-23  
메뉴 6-32  
문자열 3-36  
Insert 명령  
(메뉴 디자이너) 6-29  
INSERT 문 21-9  
InsertObject 메소드 3-37  
InsertRecord 메소드 16-26  
InterBaseExpress 10-23  
InternalCalc 필드 17-6, 20-19 ~ 20-20  
인덱스 20-17  
Internet Engineering Task Force 22-1  
Internet 페이지(컴포넌트 팔레트) 3-18  
InTransaction 속성 19-8  
Invalidate 메소드 32-9  
IP 주소 23-4, 23-6  
호스트 23-4  
호스트 이름 23-4, 23-5  
IPaint 인터페이스 4-18  
IPersist 인터페이스 4-17  
IProviderSupport 인터페이스 21-2  
IPX/SPX 프로토콜 23-1  
IRotate 인터페이스 4-19  
is 연산자 3-10  
ISpecialWinHelpViewer 5-13  
IsValidChar 메소드 17-15  
ItemHeight 속성 3-24, 15-10  
ItemIndex 속성 3-24  
라디오 그룹 3-26  
Items 속성 3-24  
TDBComboBox 15-10  
TDBListBox 15-10

- 라디오 그룹 3-26
- 라디오 컨트롤 15-13
- IUnknown 인터페이스 4-20, 4-24
  - TInterfacedObject에서 구현 4-20

## K

- KeepConnection 속성 19-5, 19-11
- KeyDown 메소드 34-10
- KeyExclusive 속성 20-5, 20-8
- KeyField 속성 15-12
- KeyFieldCount 속성 20-5
- KeywordHelp 5-19
- Kind 속성
  - 비트맵 버튼 3-22
- Kylix/Delphi 유닛 비교 10-7

## L

- Last 메소드 16-10
- Layout 속성 3-22
- LD\_LIBRARY\_PATH 13-3, 13-4
- LEpath 컴파일러 지시어 11-11
- Left 속성 6-4
- LeftCol 속성 3-29
- Length 함수 4-32
- LibraryName 속성 19-2
- Lines 속성 3-19
- LineSize 속성 3-20
- LineTo 메소드 8-4, 8-7, 8-10, 29-2
- Linux
  - 디렉토리 10-14
- Linux 환경 10-12
- ListField 속성 15-12
- ListSource 속성 15-12
- LNpath 컴파일러 지시어 11-11
- Loaded 메소드 26-13
- LoadFromFile 메소드 29-3
  - 그래픽 8-19
  - 문자열 3-33
  - 클라이언트 데이터셋 14-8, 20-41
- LoadFromStream 메소드
  - 클라이언트 데이터셋 20-41
- LoadParamListItems 프로시저 19-10
- LoadParamsFromIniFile 메소드 19-3

- LoadParamsOnConnect 속성 19-3
- LocalHost 속성
  - 클라이언트 소켓 23-6
- LocalPort 속성
  - 클라이언트 소켓 23-6
- Locate 메소드 16-15
- Lock 메소드 9-7
- LockList 메소드 9-7
- Login 대화 상자 19-6
- LoginPrompt 속성 19-6
- Lookup 메소드 16-16
- LookupCache 속성 17-9
- LookupDataSet 속성 17-9, 17-11
- LookupKeyFields 속성 17-9, 17-11
- LookupResultField 속성 17-12
- LUpackage 컴파일러 지시어 11-11

## M

- MainMenu 컴포넌트 6-21
- make 유틸리티 10-13
- Man 페이지 5-13
- Margin 속성 3-22
- 마스크 17-13
- MasterFields 속성 18-15, 20-10
- MasterSource 속성 18-15, 20-10
- Max 속성
  - 진행 표시줄 3-29
  - 트랙 표시줄 3-20
- MaxLength 속성 3-19
  - data-aware 메모 컨트롤 15-9
- MaxRows 속성 22-21
- MaxStmtsPerConn 속성 19-1
- MaxTitleRows 속성 15-22
- MaxValue 속성 17-12
- MBCS 4-28
- MDI 애플리케이션 5-2
  - 생성 5-2
- Memo 컨트롤 3-18
- MergeChangeLog 메소드 20-16, 20-41
- MergeIniFile 유틸리티 13-5
- \$MESSAGE 지시어 10-18
- Method 속성 22-12
- MethodType 속성 22-9, 22-13
- midas.so 13-5, 20-1

- MIME 메시지 22-4
- Min 속성
  - 진행 표시줄 3-29
  - 트랙 표시줄 3-20
- MinSize 속성 3-23
- MinValue 속성 17-12
- Mode 속성
  - 펜 8-5
- Modified 메소드 34-12
- Modified 속성 3-19
- Month 속성 33-5
- MouseDown 메소드 34-9
- MouseToCell 메소드 3-29
- Move 메소드
  - 문자열 목록 3-36, 3-37
- MoveBy 메소드 16-11
- MovePt 8-28
- MoveTo 메소드 8-4, 8-7, 29-2
- MultiSelect 속성 3-24
- MyBase 20-38

## N

- Name 속성
  - 매개변수 18-8
  - 필드 17-12
- NetCLX 10-4
- New Field 대화 상자 17-5
  - Field properties 17-5
  - Lookup definition 17-6
  - Result Field 17-9
- 조회(look up) 정의
  - Dataset 17-9
  - Lookup Keys 17-9
  - Key Fields 17-9
- 타입 17-6
- 필드 정의 17-6, 17-7, 17-9, 17-10
- 필드 타입 17-6
- New Items 대화 상자 5-8, 5-9, 5-10
- New Thread Object 대화 상자 9-2
- New 명령 24-10
- NewValue 속성 17-20, 21-11
- Next 메소드 16-10
- NextRecordSet 메소드 18-11
- NotifyID 5-14
- Null 값 16-26
  - 범위 20-7
- NumGlyphs 속성 3-22

## O

- Object Inspector 3-7, 26-2, 30-4
  - 메뉴 선택 6-29
  - 배열 속성 편집 26-2
- Object Repository 5-8 ~ 5-11
  - 공유 디렉토리 지정 5-9
  - 웹 서버 애플리케이션 변환 22-23
  - 항목 사용 5-10
  - 항목 추가 5-9
- Objects 속성 3-30
  - 문자열 목록 3-37, 7-9
- ObjectView 속성 15-22, 17-22
- objrepos 디렉토리 5-9
- OldValue 속성 17-20, 21-11
- OnAccept 이벤트 23-7
  - 서버 소켓 23-9
- OnAction 이벤트 22-10
- OnCalcFields 이벤트 16-8, 16-28, 17-7, 17-8, 20-20
- OnCellClick 이벤트 15-25
- OnChange 이벤트 17-14, 29-5, 32-7, 33-11, 34-12
- OnClick 이벤트 3-21, 27-1, 27-3, 27-5
  - 버튼 3-6
- OnClientRead 이벤트
  - 서버 소켓 23-10
- OnClientWrite 이벤트
  - 서버 소켓 23-10
- OnColEnter 이벤트 15-25
- OnColExit 이벤트 15-25
- OnColumnMoved 이벤트 15-19, 15-25
- OnConnect 이벤트
  - 클라이언트 소켓 23-8
- OnCreate 이벤트 24-12
- OnCreateHandle 이벤트
  - 서버 소켓 23-8
- OnDataChange 이벤트 15-4, 34-7, 34-11
- OnDataRequest 이벤트 20-35, 21-3, 21-12
- OnDbClick 이벤트 15-25, 27-5
- OnDisconnect 이벤트
  - 클라이언트 소켓 23-6
- OnDragDrop 이벤트 15-25, 27-5
- OnDragOver 이벤트 15-25, 27-5
- OnDrawCell 이벤트 3-29
- OnDrawColumnCell 이벤트 15-24, 15-25
- OnDrawDataCell 이벤트 15-25
- OnDrawItem 이벤트 7-10
- OnEditButtonClick 이벤트 15-21, 15-25
- OnEndDrag 이벤트 15-25, 27-5
- OnEnter 이벤트 15-25, 27-5
- OnError 이벤트
  - 소켓 23-8
- OnExit 이벤트 15-25
- OnFilterRecord 이벤트 16-8, 16-17, 16-20
- OnGetData 이벤트 21-7
- OnGetdataSetProperties 이벤트 21-6
- OnGetTableName 이벤트 20-38, 21-11
- OnGetText 이벤트 17-14
- OnGetThread 이벤트 23-9
- OnHandleActive 이벤트
  - 클라이언트 소켓 23-8
- OnHTMLTag 이벤트 22-18, 22-19
- OnIdle 이벤트 핸들러 9-5
- OnKeyDown 이벤트 15-25, 27-5, 34-10
- OnKeyPress 이벤트 15-25, 27-5
- OnKeyUp 이벤트 15-25, 27-5
- OnLogin 이벤트 19-6
- OnLogTrace 이벤트 19-14
- OnMeasureItem 이벤트 7-9
- OnMouseDown 이벤트 8-23, 8-24, 15-25, 27-5, 34-9
  - 전달된 매개변수 8-23
- OnMouseMove 이벤트 8-23, 8-25, 15-25, 27-5
  - 전달된 매개변수 8-23
- OnMouseUp 이벤트 8-14, 8-23, 8-25, 15-25, 27-5
  - 전달된 매개변수 8-23
- OnPaint 이벤트 3-31, 8-2
- OnPopUp 이벤트 7-6
- OnRead 이벤트
  - 클라이언트 소켓 23-10
- OnReconcileError 이벤트 10-27, 20-32
- OnResize 이벤트 8-2
- OnScroll 이벤트 3-20
- OnSetText 이벤트 17-14
- OnStartDrag 이벤트 15-26
- OnStateChange 이벤트 16-4
- OnTerminate 이벤트 9-6
- OnThreadStart 이벤트
  - 서버 소켓 23-9
- OnTitleClick 이벤트 15-26
- OnUpdateData 이벤트 15-4, 21-7, 21-8
- OnUpdateError 이벤트 10-27, 20-38, 21-11
- OnValidate 이벤트 17-14
- OnWrite 이벤트
  - 클라이언트 소켓 23-10
- Open 메소드
  - TSQLConnection 19-4
  - 데이터베이스 19-4
  - 데이터셋 16-2, 18-10
  - 서버 소켓 23-7
- OpenString 4-28
- Options 속성 3-30
  - TSQLClientDataSet 20-37
  - 데이터 그리드 15-23
  - 프로바이더 21-5 ~ 21-6
- Orientation 속성
  - 트랙 표시줄 3-20
- Origin 속성 8-27, 17-12
- override 지시어 25-8
- Owner 속성 24-12
- OwnerDraw 속성 7-7
- owner-draw 컨트롤 3-37, 7-7
  - 그리기 7-9
  - 목록 상자 3-24, 3-25
  - 선언 7-7
  - 크기 지정 7-9

## P

- \$P 컴파일러 지시어 4-35
- PacketRecords 속성 10-27, 20-30
- PageSize 속성 3-20
- Paint 메소드 29-4, 32-8, 32-9
- PaletteChanged 속성 10-19
- Panels 속성 3-29
- PAnsiChar 4-27
- PAnsiString 4-32
- ParamByName 메소드 18-5, 18-9
- ParamCheck 속성 18-5, 18-13

- Params 속성 18-5, 18-8, 19-2, 19-6
  - 클라이언트 데이터셋 20-26, 20-27
- ParamType 속성 18-5, 18-8
- ParamValues 속성 18-6
- Parent 속성 24-12
- ParentColumn 속성 15-22
- ParentShowHint 속성 3-29
- PasteFromClipboard 메소드 7-4
  - data-aware 메모 컨트롤 15-9
  - 그래픽 15-9
- PathInfo 속성 22-9
- PChar 4-27
  - 문자열 변환 4-33
- Pen 속성 8-3, 8-5, 29-2
- PenPos 속성 8-3, 8-7
- PickList 속성 15-20
- picture 객체 29-3
- Picture 속성 3-30, 8-17
  - 프레임 6-13
- Pie 메소드 8-4
- Pixels 속성 8-5
- pmCopy 상수 8-28
- pmNotXor 상수 8-28
- Polygon 메소드 8-4, 8-11
- PolyLine 메소드 8-4, 8-10
- PopupMenu 속성 7-6
- PopupMenu 컴포넌트 6-21
- Port 속성
  - 서버 소켓 23-7
- Position 속성 3-20, 3-29
- Post 메소드 16-6, 16-7, 16-24
  - Edit 메소드 16-22
- Precision 속성
  - 필드 17-12
- Prepared 속성 18-10
- Prior 메소드 16-10
- Priority 속성 9-3
- private 3-9
- private 섹션 4-2
- private 속성 26-5
- Project Manager 6-2
- Project Options 대화 상자 5-2
- protected 3-9
  - 이벤트 27-5
  - 지시어 27-5
  - 키워드 26-3, 27-5
- protected 섹션 4-1

- ProviderFlags 속성 21-4, 21-10
- ProviderName 속성 20-26, 21-3
- PString 4-32
- public 3-9
  - 속성 26-12
  - 지시어 27-5
  - 클래스의 부분 25-6
  - 키워드 27-5
- public 섹션 4-1
- published 3-9, 26-3
  - 속성 26-12, 26-13
  - 예제 32-2, 33-2
  - 지시어 26-3, 27-5
  - 클래스의 부분 25-6
  - 키워드 27-5
- PWideChar 4-27
- PWideString 4-32

## Q

- QClipboard 유닛 7-3
- Qt widget 10-11
- QueryInterface 메소드 4-20, 4-24

## R

- RDBMS 14-2
- Read 메소드
  - TFileStream 4-43
- read 메소드 26-6
- read 예약어 26-8
- ReadBuffer 메소드
  - TFileStream 4-43
- ReadCommitted 19-9
- README 13-9, 13-10
- ReadOnly 속성 3-19, 34-3
  - data-aware 메모 컨트롤 15-9
  - data-aware 컨트롤 15-5
  - 데이터 그리드 15-20, 15-24
  - 클라이언트 데이터셋 20-14
  - 필드 17-12
- ReasonString 속성 22-14
- RecNo 속성
  - 클라이언트 데이터셋 20-2
- Reconcile 메소드 10-27
- Rectangle 메소드 8-4, 8-11, 29-2
- Refresh 메소드 15-7, 20-34
- RefreshLookupList 속성 17-10

- RefreshRecord 메소드 20-34, 21-3
- Register 메소드 8-3
- Register 프로시저 24-11, 30-2
- RegisterComponents
  - 프로시저 11-5, 24-11, 30-2
- RegisterHelpViewer 5-21
- RegisterPropertyEditor
  - 프로시저 30-9
- RegisterViewer 함수 5-18
- Release 메소드 4-20, 4-23, 4-24
  - TCriticalSection 9-7
- RemoteHost 속성
  - 클라이언트 소켓 23-6
- RemotePort 속성
  - 클라이언트 소켓 23-6
- RemoteServer 속성 20-26
- RenameFile 함수 4-39
- RepeatableRead 19-10
- Repository 대화 상자 5-8
- Requires 목록 (패키지) 11-6, 11-8
- Resbind 유틸리티 12-6
- ResetEvent 메소드 9-9
- ResolveToDataSet 속성 21-4
- resourcestring 예약어 12-5
- RestoreDefaults 메소드 15-21
- Result 데이터 패킷 20-31
- Resume 메소드 9-10, 9-11
- ReturnValue 속성 9-9
- RevertRecord 메소드 10-27
- RFC 문서 22-1
- RFC (Request for Comment) 문서 22-1
- Rollback 메소드 19-9
- RoundRect 메소드 8-4, 8-11
- RowAttributes 속성 22-21
- RowCount 속성 15-12
- RowHeights 속성 3-30
- RowRequest 메소드 21-3
- Rows 속성 3-30
- RPM 13-1, 13-2
- RTTI 25-6
- \$RUNONLY 컴파일 지시어 11-10

## S

- Save as Template 명령 (메뉴 디자이너) 6-29, 6-31
- Save Template 대화 상자 6-31
- Save To File 명령 20-39
- SaveToFile 메소드 8-19, 29-3
  - 문자열 3-33
  - 클라이언트 데이터셋 14-8, 20-42
- SaveToStream 메소드
  - 클라이언트 데이터셋 20-42
- ScaleBy 속성
  - TCustomForm 13-8
- Scaled 속성
  - TCustomForm 13-8
- ScanLine 속성
  - 비트맵 예제 8-18
- Screen 변수 6-4
- ScrollBars 속성 3-30, 7-2
- SDI 애플리케이션 5-2
- Sections 속성 3-27
- Select Menu 대화 상자 6-29
- Select Menu 명령 (메뉴 디자이너) 6-29
- SELECT 문 18-3
- SelectAll 메소드 3-19
- SelectCell 메소드 33-12, 34-4
- Selection 속성 3-29
- SelectKeyword 5-18
- SelEnd 속성 3-20
- Self 매개변수 24-12
- SelLength 속성 3-19, 7-3
- SelStart 속성 3-19, 3-20, 7-3
- SelText 속성 3-19, 7-3
- SendBuf 메소드 23-7, 23-8
- Sender 매개변수
  - 예제 8-6
- SetBrushStyle 메소드 8-8
- SetData 메소드 17-15
- SetEvent 메소드 9-9
- SetFields 메소드 16-26
- SetFloatValue 메소드 30-6
- SetKey 메소드 20-3
  - EditKey와 비교 20-5
- SetLength 프로시저 4-32
- SetMethodValue 메소드 30-6
- SetOptionalParam 메소드 20-23
- SetOrdValue 메소드 30-6
- SetPenStyle 메소드 8-6
- SetProvider 메소드 20-26
- SetRange 메소드 20-7, 20-8
- SetRangeEnd 메소드 20-7
- SetRangeStart 메소드 20-6
- SetSchemaInfo 메소드 18-16
- SetStrValue 메소드 30-6
- SetTraceCallbackEvent 메소드 19-14
- Setup Graphic Installer 13-1
- SetValue 메소드 30-6
- Shape 속성 3-30
- Shift 상태 8-24
- ShortCut 속성 6-25
- ShortString 4-26
- Show 메소드 6-5, 6-7
- ShowColumnHeaders 속성 3-25
- ShowHint 속성 3-29, 15-28
- ShowHintChanged 속성 10-19
- ShowModal 메소드 6-5
- ShutDown 5-14
- Size 속성
  - 필드 17-12
- SoftShutDown 5-14
- \$SONAME 지시어 5-4
- \$SOPREFIX 지시어 5-4
- Sorted 속성 3-24
  - TDBComboBox 15-10
  - TDBListBox 15-10
- SortFieldNames 속성 18-6
- \$SOSUFFIX 지시어 5-4
- \$SOVERSION 지시어 5-4
- Spacing 속성 3-22
- SQL 14-2, 18-3
  - 명령 실행 19-12 ~ 19-13
  - 자동으로 생성 18-6, 18-12
- SQL 문
  - 모니터링 19-13 ~ 19-14
  - 생성 18-6, 18-12
    - TSQLDataSet 18-10
    - 프로바이더 21-4, 21-9 ~ 21-10
  - 실행 18-11 ~ 18-14
  - 클라이언트 제공 20-28, 21-6
  - 프로바이더 생성 21-11
- SQL 서버
  - 데이터 편집 16-7
  - 레코드 삽입 16-7
- 로그인 14-3
- 액세스 18-1
  - 제약 조건 17-20
- SQL 속성 18-4, 18-12
- SQL 클라이언트 데이터셋 20-35 ~ 20-38
- SQLConnection 속성 18-2
- Standard 페이지
  - (컴포넌트 팔레트) 3-18
- StartTransaction 메소드 19-8
- State 속성 3-22
  - 그리드 15-15, 15-17
  - 그리드 열 15-16
  - 데이터셋 16-3, 17-8
- StatusCode 속성 22-14
- StatusFilter 속성 10-27, 21-8
- Step 속성 3-29
- StepBy 메소드 3-29
- StepIt 메소드 3-29
- stored 지시어 26-13
- StoredProcName 속성 18-8, 18-12
- StrByteType 4-29
- Stretch 속성 15-9
- StretchDraw 메소드 8-4, 29-2, 29-4
- String List Editor
  - 표시 15-10
- string 예약어 4-27
  - 기본 타입 4-26
- Strings 속성 3-35
- StrNextChar 함수 10-15
- Structured Query Language SQL 참조
- Style 속성 3-24, 3-25, 3-30, 7-7
  - 브러시 8-8
  - 콤보 상자 15-11
  - 툴 버튼 6-37
  - 펜 8-5
- StyleChanged 속성 10-19
- Suspend 메소드 9-11
- Synchronize 메소드 9-4

## T

- TableAttributes 속성 22-21
- TableName 속성 18-7
- TableOfContents 5-18
- Tabs 속성 3-27
- TabStopChanged 속성 10-19
- Tag 속성 17-12
- TApacheApplication 5-6

TApplication 5-13, 5-19, 10-5  
 TApplicationEvents 6-3  
 tar 파일 13-1  
 TASM 코드 10-15  
 TBCDField  
   기본 서식 17-13  
 TBevel 3-31  
 TBitmap 29-2  
 TBrush 3-30  
 tbsCheck 상수 6-37  
 TCalendar 33-1  
 TCanvas 3-37  
 TCGIApplication 22-5  
 TCGIRequest 22-5  
 TCGIResponse 22-5  
 TcharProperty 타입 30-5  
 TclassProperty 타입 30-5  
 TClientDataSet 13-5, 20-1  
 TClipboard 7-3  
 TcolorProperty 타입 30-5  
 TComponent 3-11, 3-14, 24-5  
 TcomponentProperty 타입 30-5  
 TControl 3-15, 24-4, 27-5  
 TCP/IP 23-1  
   서버 23-6  
   클라이언트 23-6  
 TCurrencyField  
   기본 서식 17-13  
 TCustomClientDataSet 16-2  
 TCustomContentProducer 22-16  
 TCustomControl 3-16, 24-4  
 TCustomEdit 10-6  
 TCustomGrid 33-1, 33-2  
 TCustomListBox 24-3  
 TCustomSQLDataSet 16-2  
 TDataSet 16-1  
   자손 16-2  
 TDataSetProvider 13-5, 21-1, 21-2  
 TDataSetTableProducer 22-22  
 TDataSource 15-3 ~ 15-4  
 TDateField  
   기본 서식 17-13  
 TDateTime 타입 33-5  
 TDateTimeField  
   기본 서식 17-13  
 TDBCheckBox 15-2, 15-12 ~ 15-13  
 TDBComboBox 15-2, 15-10 ~ 15-11  
 TDBEdit 15-2, 15-8  
 TDBGrid 15-2, 15-15 ~ 15-26  
   속성 15-19  
   이벤트 15-25  
 TDBGridColumns 15-15  
 TDBImage 15-2, 15-9  
 TDBListBox 15-2, 15-10, 15-10 ~ 15-11  
 TDBLookupComboBox 15-2, 15-11 ~ 15-12  
 TDBLookupCombox 15-10  
 TDBLookupListBox 15-2, 15-10, 15-11 ~ 15-12  
 TDBMemo 15-2, 15-9  
 TDBNavigator 15-2, 15-26 ~ 15-29, 16-9, 16-10  
 TDBRadioGroup 15-2, 15-13  
 TDBText 15-2, 15-8  
 TDefaultEditor 30-13  
 TDrawGrid 3-29  
 TEdit 컨트롤 3-18  
 TenumProperty 타입 30-5  
 Terminate 메소드 9-5  
 Terminated 속성 9-5  
 TEvent 9-9  
 Text 속성 3-19, 3-25, 3-29  
 TextHeight 메소드 8-4, 29-2  
 TextOut 메소드 8-4, 29-2  
 TextRect 메소드 8-4, 29-2  
 TextWidth 메소드 8-4, 29-2  
 TField 16-1, 17-1 ~ 17-26  
   메소드 17-15  
   속성 17-1, 17-11 ~ 17-14  
   런타임 17-12  
   이벤트 17-14  
 TFieldDataLink 34-5  
 TFile 4-41  
 TFileStream 3-39  
   파일 I/O 4-41 ~ 4-45  
 TFloatField  
   기본 서식 17-13  
 TfloatProperty 타입 30-5  
 TFMTBcdField  
   기본 서식 17-13  
 TfontNameProperty 타입 30-5  
 TfontProperty 타입 30-5  
 TForm  
   스크롤 막대 속성 3-20  
 TFrame 6-11  
 TGraphic 29-2  
 TGraphicControl 3-16, 24-4, 32-2  
 THeaderControl 3-27  
 Thread Status 상자 9-11  
 thread-aware 객체 9-4  
 ThreadID 속성 9-11  
 threadvar 9-5  
 THTMLTableAttributes 22-21  
 THTMLTableColumn 22-22  
 TickMarks 속성 3-20  
 TickStyle 속성 3-20  
 TIcon 29-2  
 TiledDraw 메소드 29-4  
 TImage 3-30  
   프레임 6-13  
 TImageList 6-36  
 TintegerProperty 타입 30-5, 30-6  
 TInterfacedObject 4-23  
   unknown 구현 4-20  
   동적 바인딩 4-21  
   파생 4-21  
 Title 속성  
   데이터 그리드 15-20  
 TKeyPressEvent 타입 27-4  
 TLabel 3-28  
 TListBox 24-3  
 TMainMenu 컴포넌트 6-21  
 TMemIniFile 10-6  
 TMemo 컨트롤 3-18  
 TMemoryStream 3-39  
 TmethodProperty 타입 30-5  
 TMultiReadExclusiveWriteSynchronizer 9-8  
 TNotifyEvent 27-8  
 TObject 3-13, 4-1, 25-4  
 ToggleButton 10-6  
 Top 속성 6-4, 6-34  
 TopRow 속성 3-29  
 TordinalProperty 타입 30-5  
 TPageControl 3-27  
 TPageProducer 22-16  
 TPaintBox 3-31  
 TPanel 3-26, 6-32  
 TParam 18-5, 18-8  
 TPersistent 3-13  
 TPicture 타입 29-2  
 TPopupMenu 6-38  
 TPrinter 3-38



사용 3-38  
 TPropertyAttributes 30-8  
 TPropertyEditor 클래스 30-4  
 TQuery 5-8  
 TraceList 속성 19-13  
 Transliterate 속성 17-12  
 Transparent 속성 3-28  
 TReader 4-41  
 TRegIniFile 10-6  
 try 예약어 29-4  
 TScrollBar 3-20, 3-27  
 TSearchRec 4-37  
 TsetElementProperty 타입 30-5  
 TsetProperty 타입 30-5  
 TShape 3-30  
 TSpinEdit 컨트롤 3-21  
 TSQLClientDataSet 13-5  
 TSQLClientDataSet, 장점과 단점 20-35  
 TSQLConnection 18-2 ~ 18-3, 19-1 ~ 19-14  
   메시지 모니터링 19-14  
   바인딩 19-2 ~ 19-4  
 TSQLDataSet 18-2, 18-4, 18-6 ~ 18-7, 18-14  
 TSQLMonitor 19-13 ~ 19-14  
 TSQLQuery 18-2, 18-4, 18-14, 22-22  
 TSQLQueryTableProducer 22-22  
 TSQLStoredProc 18-2, 18-8  
 TSQLTable 18-2, 18-7, 18-15  
 TSQLTimeStampField  
   기본 서식 17-13  
 TStream 3-39  
 TStringList 3-32 ~ 3-37, 5-16  
 TStringProperty 타입 30-5  
 TStringStrings 3-32 ~ 3-37  
 TStringStream 3-39  
 TTabControl 3-27  
 TTable 5-8  
 TTcpClient 23-6  
 TTcpServer 23-6  
 TThread 9-2  
 TThreadList 9-5, 9-7  
 TTimeField  
   기본 서식 17-13  
 TToolBar 6-32, 6-35  
 TToolButton 6-32

TTreeView 3-25  
 TWebActionItem 22-6  
 TWebApplication 22-4  
 TWebRequest 22-4  
 TWebResponse 22-4, 22-6  
 TWidgetControl 3-16, 10-4, 24-4, 27-5  
 TWinControl 10-4  
   TWidgetControl 참조  
 TWriter 4-41

## U

Unlock 메소드 9-7  
 UnlockList 메소드 9-7  
 UPDATE 문 21-9  
 UpdateBatch 메소드 10-27  
 UpdateCalendar 메소드 34-4  
 UpdateMode 속성 21-9  
   TSQLClientDataSet 20-37  
 UpdateRecordTypes 속성 10-27  
 UpdatesPending 속성 10-27  
 UpdateStatus 속성 10-27, 21-8  
 UpdateTarget 메소드 6-18  
 URI  
   URL과 비교 22-2  
 URL 22-2  
   IP 주소 23-4  
   URI와 비교 22-2  
   웹 브라우저 22-3  
   호스트 이름 23-4  
 URL 속성 22-12  
 Use Unit 명령 5-7, 6-2  
 USEPACKAGE 매크로 11-6  
 uses 절 3-9, 10-4  
   데이터 모듈 추가 5-7  
   순환 참조 방지 6-2  
   패키지 포함 11-3  
 UTF-8 문자 집합 10-15, 12-2

## V

\$V 컴파일러 지시어 4-35  
 Value 속성  
   매개변수 18-5, 18-8  
   집계 20-22  
   필드 17-16  
 ValueChecked 속성 15-12  
 Values 속성  
   라디오 그룹 15-13  
 ValueUnchecked 속성 15-13  
 var 예약어  
   이벤트 핸들러 27-3

VendorLib 속성 19-2  
 VertScrollBar 3-20  
 ViewStyle 속성 3-25  
 Visible 속성 3-3  
   메뉴 6-32  
   콜바 6-38  
   툴바 6-38  
   필드 17-12  
 VisibleButtons 속성 15-27, 15-28  
 VisibleChanged 속성 10-20  
 VisibleColCount 속성 3-29  
 VisibleRowCount 속성 3-29  
 VisualCLX 10-4

## W

WaitFor 메소드 9-8, 9-10  
 WantReturns 속성 3-19  
 WantTabs 속성 3-19  
   data-aware 메모 컨트롤 15-9  
 wchar\_t widechar 10-20  
 \$WEAKPACKAGEUNIT  
   컴파일러 지시어 11-10  
 WebBroker 22-1  
 WideChar 4-25, 4-27, 12-3  
 WideString 4-27, 12-3  
 widestrings 10-20  
 widget 3-16, 10-4  
   컨트롤 24-4  
 WidgetDestroyed 속성 10-20  
 Width 속성 3-29, 6-4  
   TScreen 13-8  
   데이터 그리드 15-20  
   데이터 그리드 열 15-16  
   펜 8-5, 8-6  
 WIN32 10-16  
 WIN64 10-16  
 Windows 애플리케이션 10-1  
   포팅 10-1 ~ 10-14  
 Windows 메시징 10-15  
 WM\_SIZE 메시지 33-4  
 WordWrap 속성 3-19, 7-1, 31-1  
   data-aware 메모 컨트롤 15-9  
 wrAbandoned 상수 9-10  
 Wrap 속성 6-37  
 Wrapable 속성 6-37  
 wrError 상수 9-10  
 Write 메소드  
   TFileStream 4-43  
 write 메소드 26-7

write 예약어 26-8  
WriteBuffer 메소드  
    TFileStream 4-43  
wrSignaled 상수 9-10  
wrTimeout 상수 9-10

## X

---

\$X 컴파일러 지시어 4-35  
Xerox Network System  
    (XNS) 23-1  
.xfm 파일 12-5, 26-11  
    생성 12-6

## Y

---

Year 속성 33-5

## Z

---

-Z 컴파일러 지시어 11-11