WHITE PAPER

Delphi™

비주얼베이직에서 델파이로의 마이그레이션

프로그래머/개발자를 위한 개론

Mitchell C. Kerman

목차

서론

리눅스 OS는 최근 대단히 많은 미디어에서 거론되고 있습니다. 리눅스는 윈도우에 대한 대안으로써 거의 열광적이라 할만큼의 반응을 얻고 있습니다. 그 결과 프로그래머와 개발자는 전세계 최종 사용자 및 고객을 위해 리눅스의 장점을 최대한 살린 시스템 개발 방법을 연구해오고 있습니다. Borland 델파이도 이러한 작업에 동참해 왔습니다. 그동안 델파이는 윈도우를 위한 고급 애플리케이션 개발툴로만 인식되어 왔습니다. 카일릭스(리눅스용 델파이) 프로젝트를 통해 델파이를 사용하는 개발자는 크로스플랫폼 윈도우/리눅스 개발을 위한 빠른 방법을 제공받게 됩니다.

마이크로소프트사의 핵심 제품인 비주얼베이직(VB)은 가장 인기 있는 윈도우 OS 개발툴입니다. VB는 BASIC 언어에 기반한 RAD(Rapid Application Development)입니다. 최근, 전세계적으로 7백만이 넘는 개발자가 VB를 사용하는 것으로 알려지고 있습니다. VB와 델파이는 애플리케이션 개발 시간을 줄여주어 프로그래머 생산성을 강화해주지만, 델파이는 비주얼베이직과는 달리 윈도우와 리눅스에 모두 사용할 수 있는 개발툴이 될 것이라는 점에서 차이점이 있습니다.

리눅스란 무엇인가?

기술적으로 말해서 리눅스라는 이름은 운영 체제의 핵심인 커널만을 의미합니다. 그러나 이 문서를 포함하여, 대부분의 경우 리눅스를 마이크로소프트

Borland®

윈도우나 MacOS 같은 OS에 대한 대안으로 생각하여 전체 OS와 패키지 애플리케이션을 일컫습니다.

리눅스는 무료로 배포되는, 유닉스와 흡사한 운영 체제로 Linus Torvalds가 전세계의 프로그래머, 취미가, 컴퓨터광들의 도움을 받아 처음 개발하였습니다. 리눅스는 Intel의 80386 프로세서용으로 1991년에 고안되었지만, 지금은 Alpha, SPAR C, PowerPC, Intel x86 프로세서 제품군을 포함하는 다양한 하드웨어에서 작동하고 있습니다.

현재 리눅스는 다음과 같은 기능들을 제공하고 있습니다.

- 32 비트 아키텍쳐
- 선점형 멀티태스킹
- 보호 메모리
- 멀티유저지원
- TCP/IP를 포함한 다양한 네트워킹 지원

리눅스를 사용하는 웹 사이트와 ISP의 수가 점점 더 늘어나고 있습니다. 리눅스는 전세계적으로 C/C++ 프로그래머가 선택하는 개발 플랫폼일 뿐 아니라 전형적인 유닉스 서버 애플리케이션도 작동합니다. 여기에 포함되는 애플리케이션은 다음과 같습니다.

- 웹 서버 (예: Apache)
- 메일 서버 (예: Sendmail)
- 데이터베이스 서버 (예: Oracle, Informix)
- 윈도우/윈도우 관리자 (예: X-윈도우, GNOME, KDE)
- 오피스 제품 (예: Applixware, StarOffice, KOffice)

리눅스는 GNU General Public License (GPL)에 근거하여 배포되는데, 이는 리눅스용 소스 코드는 모든 사람이 무료로 사용할 수 있음을 의미합니다. 누구나 코드를 수정할 수 있으며 수정 사항 또한 소스코드와 함께 무료로 배포됩니다.

소스 코드를 모든 사람들에게 오픈함으로써 다음과 같은 이점이 있습니다.

- **유연성**. 리눅스는 커스터마이즈하기가 쉬우므로 휴대용 장치부터 서버 클러스터에 이르기까지 다양한 플랫폼에서 동작할 수 있습니다.
- **신뢰성**. 리눅스는 전체적으로 디버깅할 수 있으며 전세계의 수 많은 프로그래머의 손을 거쳐 OS의 새 버전마다 신속하게 리뷰 및 테스트 가능합니다.
- **경제성**. 다른 운영 체제에 비해 리눅스의 초기 설치비용은 상당히 저렴합니다. 라이센스나 관련 요금이 전혀 필요 없습니다. 뿐만 아니라 프로그래머와 상업적 벤더가 제공하는 공개소스를 통해 고객 지원도 받을 수 있습니다.

리눅스는 일종의 인터넷 현상입니다. 리눅스는 인터넷을 통해 탄생하였고 개발에 요구되는 필수적인 협업 환경을 제공받으며 성장하였습니다. 전세계 프로그래머는 리눅스에 대한 개선점과 추가 사항을 위해 코드를 작성하고 개선할 수 있습니다.

현재는 여러 개발툴을 리눅스에 이용할 수 있습니다. 그러나 도구마다 기능, 속도, cross-platform 기능, 비용이 다릅니다. 몇 가지 대중적인 리눅스 프로그래밍 도구 및 도구 공급업체는 아래와 같습니다.

- **GNU: GCC/EGCS**. OS와 환경의 개발에 있어 리눅스에 적합한 공개소스 개발툴입니다. 공개소스 도구이기 때문에 인터넷을 통해 무료로 얻을 수 있습니다. 시작부터 막대한 시장점유율을 기록하여 현재 대부분의 리눅스 도구는 GCC/EGCS로 쓰여지고 있습니다. 이 도구는 RAD, 인터넷, 데이터베이스나 GUI (Graphical User Interface) 개발 기능이 없다는 점에서 아래에 언급된 Code Warrior와 매우 흡사합니다. 그러나 이러한 기능이 없기 때문에 기업의 애플리케이션 요구에 부합하지 못하고 있으며, 또한 단순 C/C++ 기반으로 대규모 프로젝트의 개발에는 복잡하고 시간이 오래 걸리는 툴로 인식되고 있습니다.
- **Cygnus: Code Fusion for Linux**. Cygnus는 GCC 같은 GNU 도구에 대한 리눅스 IDE(통합 개발 환경)를 개발하였습니다. 1999년 8월에 출시된 Code Fusion은 리눅스 시장에 등장한 초기 도구 중의 하나입니다. GNU 도구와 Code Warrior처럼 Code Fusion의 단점은 RAD, 인터넷, 데이터 베이스, GUI 기능이 없다는 것입니다. Cygnus는 공개소스 커뮤니티에서 좋은 평판을 얻고 있으며 최근 Intel Optimizations를 도구에 포함시켰지만 일부 매니아 사용자 외에는 브랜드 인지도가 떨어집니다.
- **Metrowerks: Code Warrior**. Code Warrior는 첫 상업용

리눅스 개발툴로서 1999년월에 처음 출시되었습니다. 윈도우, Solaris, 리눅스, BeOS, Palm, WinCE, Java를 포함하는 제품라인과 함께 멀티 플랫폼 개발이 가능하지만 시장 점유율이 가장 낮습니다. 또한 RAD 툴이 아니며 GUI나 데이터베이스 기능도 포함하고 있지 않습니다.

- **마이크로소프트**. 현재 마이크로소프트는 리눅스 개발툴과 관련한 어떠한 계획도 발표하지 않고 있습니다. 마이크로소프트는 윈도우 OS의 마케팅 목적으로 개발툴을 이용하므로 마이크로소프트가 리눅스 툴 시장에 뛰어들 가능성은 거의 없습니다.
- **볼랜드**. 볼랜드는 최근 리눅스 개발툴인 J빌더를 제공하고 있으며, 곧 C++빌더와 델파이로 그 영역을 넓힐 것입니다. J빌더는 볼랜드의 자바 개발 환경으로 현재 윈도우, 리눅스, 솔라리스에 사용하실 수 있습니다. 또한 J빌더는 확장 가능한 데이터베이스 개발툴로써 자바2를 지원하고 있습니다. C++빌더는 C++ 플랫폼이며 델파이는 대표적인 RAD 개발툴입니다. C++빌더는 C++ 프로그래밍 언어를 사용하고 있는 반면, 델파이는 오브젝트 파스칼을 이용합니다. 두 가지 도구는 모두 같은 IDE를 공유하고 있습니다. 이는 매우 직관적이며 VB의 IDE 지식을 직접적으로 적용할 수 있는 IDE입니다. 지금 이 도구의 윈도우 버전은 이용할 수 있으며 리눅스 버전은 차기에 이용이 가능합니다. 볼랜드는 카일릭스를 통해 C++빌더와 델파이를 리눅스 세계에 통합하는데 모든 노력을 기울이고 있습니다.

델파이의 장점

지금까지 리눅스 OS의 기능과 다양한 리눅스 개발툴에 대해 개략적으로 알아보았습니다. 이제 VB와 델파이로 주제를 바꾸겠습니다. VB와 비교하여 델파이가 제공하는 장점은 다음과 같습니다.

- **Cross-platform 개발**. 앞에서 언급한 바와 같이 델파이는 윈도우에 가능하고 델파이의 리눅스 버전이라고 할 수 있는 카일릭스를 이용하면 동일한 소스를 리눅스에서도 이용할 수 있습니다. 두 가지 운영 체제에 같은 코드를 사용할 수 있지만, OS 고유의 차이 때문에 몇 가지의 수정 작업은 불가피합니다.
- **뛰어난 개발 환경**. 델파이 IDE는 RAD 툴과 기타 기능을 모두 제공합니다. 직관적이고 사용이 쉬운 환경 뿐 아니라 IDE가 유연하여 프로그래머는 환경을 자신의 요구 및 기호에 맞추어 사용자 정의할 수 있습니다.
- **강력한 컴포넌트 및 컨트롤**. VB 도구 상자처럼 델파이는 가장 흔히 사용하는 컴포넌트 및 컨트롤의 VCL(Visual Component Library)를 포함하고 있습니다. 이 라이브러리에 포함된 컴포넌트의 갯수 및 종류는 사용하는 델파이의 에디션에 따라 다릅니다(퍼스널, 프로페셔널, 엔터프라이즈). 모든 라이브러리 컴포넌트는 오브젝트 파스칼로 작성되어 있습니다. 이에 따라 개발자가 이 라이브러리를 수정하고 확장할 수 있는 기능을 가지고 있습니다.
- **진정한 객체 지향 프로그래밍**. 마이크로소프트는 VB가 객체 지향임을 강조하지만 '객체 기반'일 뿐 객체 지향이 아닙니다. VB에서는 진정한 객체 상속과 다형성을 찾아볼 수 없습니다. 델파이의 객체 모델은 완벽하며, 캡슐화, 상속, 다형성을 모두 지원합니다.
- **포인터 및 동적 변수**. VB는 동적인 변수를 제공하지만 명시적인 포인터 변수는 지원하지 않습니다. 효율적 코드나 더 깔끔한 데이터 구조를 원하는 경우 이러한 문제점은 대단히 자주 발생합니다. "이 데이터를 적절히 표현할 수 있는 트리 구조가 정말로 필요한데", VB에 숙달한 프로그래머는 VB 객체 변수를(이것은 어떠한 방식으로든 명시적 포인터가 됩니다) 사용하여 이러한 딜레마를 극복할 수 있지만, 많은 프로그래머는 알고리즘과 메모리 효율성을 이용하여 이러한 문제를 해결하려 합니다. 델파이의 오브젝트 파스칼은 동적 변수 및 명시적 포인터를 제공하여 알고리즘 효율성과 데이터 구조 문제를 효과적으로 해결합니다.
- **바른 프로그래밍 습관 장려**. 이제 제가 가진 불만 사항을 이야기하려 합니다. 저자로서, 그리고

프로그래밍 입문 교육자로서, 제가 겪는 가장 큰 어려움은 학생에게 *항상* 변수를 선언하도록 가르쳐야 한다는 것입니다. VB는 프로그래머에게 명시적 변수 선언(전신인 BASIC과 유사한)을 요구하지 않는 옵션을 제공합니다. 저는 학생에게 종종 "옵션 정의가 VB 코드 맨 위에 보이는지 확인하여 주십시오." 라고 이야기합니다. 아마 제 방식에 동의할 수 없을지도 모르겠습니다. 그러나 여러분이 처음 프로그램을 배울 때를 떠올려 보십시오. 이것이 전문 프로그래머에게는 강력하고 시간을 절약해주는 옵션일지 몰라도 초보 프로그래머에게는 그저 사용할 만한 옵션일 뿐입니다. 명시적으로 선언한 변수는 변수 이름을 스스로 설명해주고(self-commenting) 가독성을 강화하며 메모리 및 공간 요구에 대해 좀 더 강력한 컨트롤을 제공합니다. 또한 다른 소스 코드 위치 내에 존재하는 같은 이름의 변수 사이에 발생하는 혼란을 피할 수 있게 도와줍니다. 오브젝트 파스칼은 항상 명시적 변수 선언을 요구하기 때문에 올바른 프로그래밍 습관을 장려합니다. 또한 오브젝트 파스칼의 구문은 구조적인 모듈별 프로그래밍을 돕습니다.

- **강력한 타입 지정 규칙**. 올바른 프로그래밍 습관에 이어 언어의 데이터타입식 지정 규칙에 대하여 비교합니다. VB는 약한 타입 지정 언어입니다. 예를 들어 데이터 타입 Double의(배정도 부동 소수점 숫자) 변수는 정수형 변수에 별 영향 없이 할당될 수 있습니다. VB는 자동으로 배정도 값을 정수 값으로 변환합니다. 이러한 자동 변환 기능은 여러 문제점을 갖고 있습니다. VB는 부동 소수점 부분을 반올림하는지 절단하는지 기억합니까? 양수 값을 보면 VB는 부동 소수점을 자른다는 것을 알 수 있습니다. VB에서는 다음으로 가장 낮은 정수 값으로 부동 소수점 부분을 0.5 이하로 자르고(부동 소수점을 절단하는 것은 필수적입니다) 부동 소수점 값이 0.5 이상이 되도록 다음으로 가장 높은 정수 값까지 잘라 올립니다. 물론 우리는 VB의 Fix나 Int 함수 및 같은 데이터 타입 변환 함수를 이용해서 이러한 문제에서 벗어나면서 동시에 코드의 가독성을 개선할 수 있습니다. VB와는 달리 델파이의 오브젝트 파스칼은 강력한 타입 지정을 특징으로 합니다. 배정도 값은 필요한 데이터 타입 변환을 거치지 않고서는 Integer 변수에 할당될 수 없습니다. 그러므로 앞에서 언급한 문제점에서 완전히 벗어날 수 있습니다. 이외에 다른 장점들은 앞으로 더욱 상세히 짚어볼 것입니다.

이 문서의 목적

이 문서는 VB에 익숙하며 델파이에 대해 더 알고 싶어하는 프로그래머 및 개발자를 대상으로 쓰여진 것입니다. 두 가지의 제품과 언어 기능에 대해 비교하고 있지만 완전한 비교는 아닙니다. 이 문서는 GUI 디자인 개념, 컴포넌트 속성과 메소드, 이벤트 기반 프로그래밍을 포함한 RAD의 기본 지식을 다루고 있습니다. 이 문서는 학생과 초보 프로그래머, 숙련된 프로그래머 모두에게 도움이 될 것입니다. 이제부터는 VB 및 델파이의 유사성과 차이점을 짚어볼 것입니다. 이러한 비교는 IDE, 프로그래밍 언어, 내장 디버거, 애플리케이션 배포의 4가지 단계로 이루어집니다. 다시 한번 강조하지만 이 문서는 완전한 레퍼런스 매뉴얼이기 보다는 VB에 익숙한 사용자가 델파이를 배움으로써 기존 지식을 강화할 수 있도록 도와주는 문서에 가깝습니다.

통합 개발환경

VB 개발자는 델파이의 IDE에 익숙할 것입니다. 메뉴, 툴바, 윈도우 등 여러 가지 요소가 VB와 비슷하게 디자인되었기 때문입니다. 델파이는 현재의 RAD 환경에 필요한 모든 도구를 포함하고 있으며 이러한 도구들은 델파이 디자인 팀의 무수한 사전작업과 노력에 의해 제작되었습니다.

VB 및 델파이는 비슷한 이름과 기능의 윈도우를 포함하고 있으며, 일례로 컨트롤 속성을 수정할 때 나타나는 윈도우가 같다는 것을 들 수 있습니다. 비주얼베이직은 MDI(Multiple Document Interface: 다중 문서 인터페이스) 개발 환경으로 모든 윈도우가 주

애플리케이션 윈도우 내에 완전히 통합됩니다. 그러나 델파이는 SDI(Single Document Interface:단일 문서 인터페이스) 환경으로 모든 윈도우가 독립적입니다. 다음에 이어지는 내용은 델파이 IDE의 요소를 소개하고 윈도우에 있어서 VB와의 차이점과 유사성을 설명하고 있습니다.

델파이의 디폴트 IDE는 다음과 같은 윈도우로 구성됩니다.

- **메뉴 바**
- **여섯개의 툴바들**
  1. Standard 툴바
  2. View 툴바
  3. Debug 툴바
  4. Custom 툴바
  5. Desktops 툴바
  6. 컴포넌트 팔레트
- **작은 윈도우들**
  1. 폼 윈도우
  2. Object Inspector 윈도우
  3. 코드 에디터 윈도우
  4. 기타 윈도우

그림 1은 델파이 IDE 기본 배치를 보여줍니다.

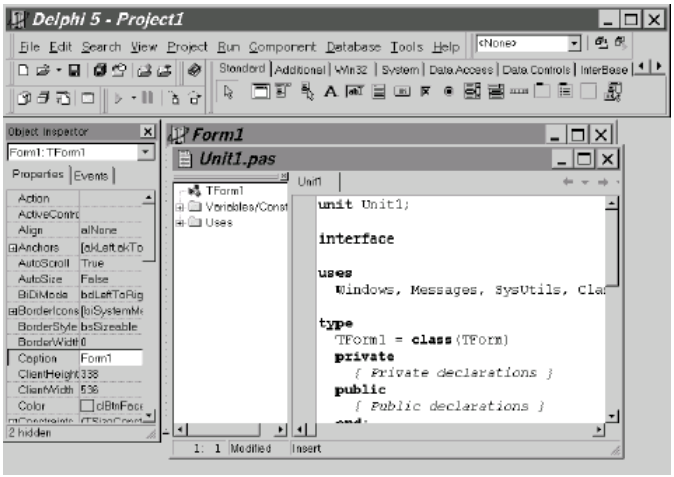

그림 1. *델파이의 기본 통합 개발 환경*

메뉴 바

VB와 같이 델파이는 메뉴 바를 포함합니다.. 델파이 메뉴 바는 전형적인 드롭 다운 메뉴입니다. 여러 메뉴 옵션들은 오른쪽 클릭으로 나타나는 드롭 다운 메뉴에서, 단축 키를 통해 직접 불러올 수 있습니다. 메뉴 바는 개발자가 애플리케이션을 작성하는데 필요한 모든 기능을 제공합니다. 메뉴 바 및 드롭 다운 메뉴는 그림 2에서 확인할 수 있습니다.

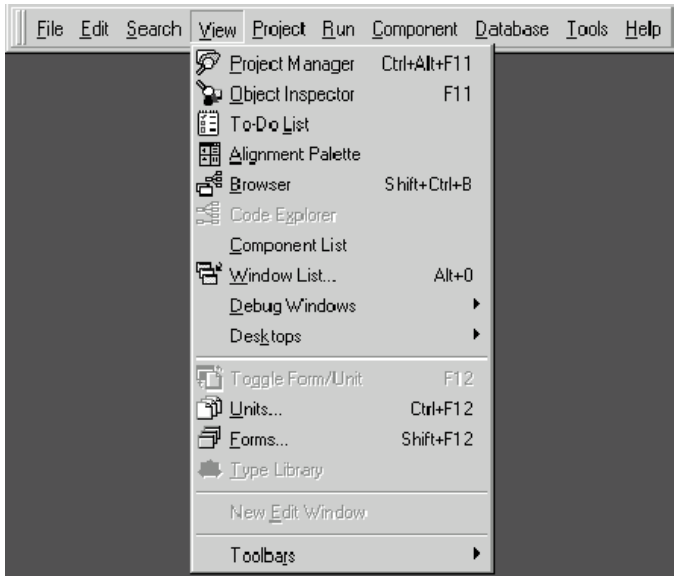

그림 2. *드롭 다운 메뉴가 활성화된 및 델파이 메뉴 바*

툴바

툴바는 일반적인 작업을 빠르게 수행할 수 있도록 도와주는 아이콘을 포함하고 있습니다. VB는 네 가지의 분리된 툴바로 목적에 따라 태스크를 분류하고 구성합니다. 여기에는 표준, 디버그, 편집, 폼 편집기 툴바가 포함되어 있습니다. VB는 기본적으로 표준 툴바 만을 표시합니다. 델파이는 여섯 개의 툴바를 가지고 있으며, 이를 모두 기본으로 표시합니다.

VB 및 델파이는 View 메뉴의 토글기능을 통해 다양한 툴바를 보이게 합니다. VB에서는 표준 툴바에서 오른쪽 클릭을 통해 이용 가능한 툴바를 보여주는 팝업 메뉴를 열 수 있습니다. 델파이에서도 비슷한 작업을 통해 팝업 메뉴를 열 수 있습니다. 또한 VB 및 델파이 툴바는 모두 사용자 정의가 가능합니다

Standard 툴바

Standard 툴바(그림 3)에는 열기, 저장, 델파이 프로젝트 및 관련 파일 작성 같은 공통 태스크에 대한 아이콘이 포함되어 있습니다.

그림 3. *Standard 툴바*

View 툴바

그림 4의 View 툴바에는 새 폼 작성, 폼과 코드 유닛 보기, 폼과 코드 유닛을 토글하는 아이콘이 포함되어 있습니다. 이 툴바를 사용하면 델파이 IDE에서 윈도우 사이를 더 빠르게 이동할 수 있습니다.

그림 4. *View 툴바*

Debug 툴바

VB에서처럼 Debug 툴바는(그림 5) 프로그램의 테스트 및 디버깅을 위해 사용됩니다. 또한 Run 메뉴에 포함되어 있는 델파이 디버거 기능들을 빠르게 실행할 수 있습니다. VB 디버거와 같이 델파이 디버거는 디자인 타임(design-time) 유틸리티입니다. 소스 코드에서 작업하는 동안에는 델파이 개발 환경 내부에서만 사용이 가능합니다.

그림 5. *Debug 툴바*

Custom 툴바

그림 6은 Custom 툴바를 보여주고 있습니다. 이 툴바는 기본값으로 델파이 온라인 도움말을 표시할 수 있는 단 하나의 버튼을 포함하고 있습니다.

그림 6. *Custom 툴바*

Desktops 툴바

VB는 모든 윈도우와 툴바를 이전의 위치에서(가장 최근의 위치) 열고 있습니다. 프로그래머는 다양한 Desktop 레이아웃을 생성할 수 없습니다.. 그러나 그림 7에서 보여주는 Desktop 툴바를 이용하면 델파이의 데스크탑 설정을 사용자 정의할 수 있습니다. 이 툴바는 가능한 데스크탑 레이아웃의 목록을 포함하고 있으며 이를 통해 프로그래머는 다양한 레이아웃을 불러오고 저장할 수 있습니다. 데스크탑 레이아웃은 IDE에서의 윈도우 화면, 사이즈 조절, 도킹을 포함합니다. 선택된 레이아웃은 모든 프로젝트에 이용할 수 있으며 다음에 델파이를 시작할 때 사용할 수 있습니다. 이에 반해 VB는 이러한 기능을 갖춘 툴바가 없습니다.

그림 7. *Desktops 툴바*

컴포넌트 팔레트

VB에서 도구 상자는 현재 프로젝트에 이용되는 모든 ActiveX 컨트롤을 가지고 있습니다. 델파이에서는 이와 같은 기능을 하는 윈도우로 그림 8과 같은 컴포넌트 팔레트가 있습니다.

그림 8. *컴포넌트 팔레트*

VB의 툴박스와 델파이의 컴포넌트 팔레트 사이의 가장 큰 차이점은 컴포넌트 팔레트가 탭으로 분류되어 있다는 점입니다. 이 탭 레이아웃을 변경하기 위해서는 컴포넌트 팔레트 위에서 오른쪽 클릭하고 팝업 메뉴에서 Properties를 선택합니다. 이제 팔레트 프로퍼티 다이얼로그가 열리면 컴포넌트 팔레트를 사용자 정의할 수 있게 됩니다. VB 툴박스는 탭으로 분류되어 있지 않은 것이 기본값입니다. 그러나 프로그래머는 도구 상자를 사용자 정의하여 빠르고 쉽게 컨트롤을 구성할 수 있도록 탭을 추가할 수 있는데, 이는 델파이의 컴포넌트 팔레트와 유사합니다.

VB의 툴박스와 델파이의 컴포넌트 팔레트의 또 다른 차이점은 VB 도구 상자가 현재 프로젝트에만 사용할 수 있는 컨트롤을 포함하고 있다는 점입니다. 반면 컴포넌트 팔레트는 항상 모든 종류의 컨트롤을 포함합니다. 또한 VB 애플리케이션을 컴파일할 때, 각 ActiveX 컨트롤은 여전히 실행 가능한 파일에서 분리되어 있지만 델파이는 필요한 컨트롤을 실행가능한 파일로 컴파일합니다.

델파이는 ActiveX 컨트롤을 지원합니다. ActiveX 컨트롤은 애플리케이션에서 사용되는데, 일련의

코드로 래핑되어 컴포넌트 팔레트에 위치하고 델파이 내에서 사용될 수 있습니다.

작은 윈도우

델파이의 디폴트 IDE에서 작은 윈도우들에는 폼 윈도우, Object Inspector 윈도우, 코드 에디터 윈도우 및 기타 윈도우를 포함됩니다.

폼 윈도우

델파이에서 폼 윈도우(그림 9)는 VB의 폼 윈도우와 비슷한 모습으로 작동합니다. 주요한 차이점은 단위에 있습니다. VB는 트윕(twip)을 이용하는 반면, Delpi는 픽셀을 사용합니다.

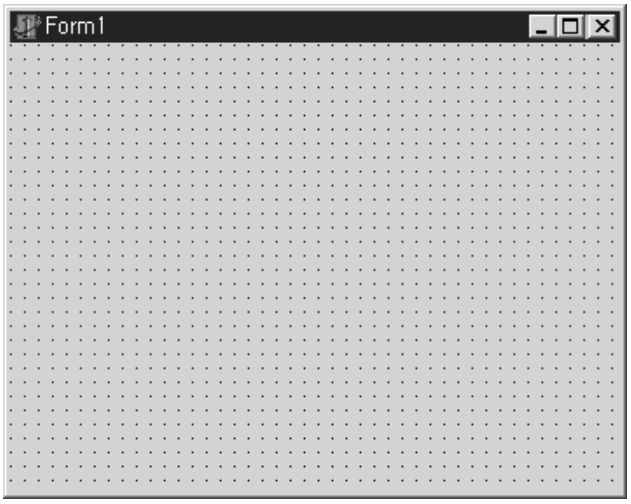

그림 9. 폼 윈도우

VB처럼 폼의 격자점은 컨트롤을 정렬하고 크기를 정하는 데 쓰입니다. 폼 디자이너가 이를 변경하기 위해서는 메뉴 바나 오른쪽 클릭으로 등록정보 탭을 열어 Tools|Environment 옵션을 선택해야 합니다. 탭에서 폼 디자이너 프레임은 아래와 같이 옵션을 변경할 수 있도록 도와줍니다.

- Display Grid – 격자점이 표시되는 것을 제어.
- Snap To Grid – 컨트롤의 정렬을 지시. 이 옵션이 활성화되면 컨트롤의 모든 코너가 정렬됨.
- Show Component Captions – 보이지 않는 컴포넌트의 이름을 폼 디자이너에 표시함.
- Show Designer Hints – 컨트롤의 크기를 조정하거나 이동할 때 이 옵션을 활성화하면 크기 및 위치를 도구 팁 힌트로 표시함
- New Forms As Text – 새롭게 작성한 폼을 텍스트나 이진 포맷중 어느 쪽으로 저장할지 지정
- AutoCreate Forms – 애플리케이션이 실행되면 새 폼을 자동으로 작성할지 여부를 결정
- Grid Size X, Grid Size Y – 격자점의 픽셀 수를 결정

VB와 같이 델파이는 컨트롤을 폼에 배치하는 몇 가지 방법을 제공합니다. 우선 더블 클릭으로 컴포넌트 팔레트 상에서 원하는 컨트롤을 폼 중앙에 디폴트 사이즈의 컨트롤로 배치합니다. 두 번째 방법은 컴포넌트 팔레트 상의 컨트롤을 왼쪽 클릭하여 폼에서 다시 왼쪽 클릭하는 것입니다. 디폴트 사이즈의 컨트롤이 상위 왼쪽 코너에서 클릭한 위치에 정렬되어 폼 상에 배치됩니다. 마지막으로 컴포넌트 팔레트 상의 컨트롤을 클릭하여 드래그로 폼에 배치합니다. 이 방법을 통해 프로그래머는 컨트롤의 크기 및 위치를 직접 지정할 수 있습니다.

같은 유형의 다중 컨트롤은 Shift키를 누른 채 컴포넌트 팔레트에서 컨트롤을 선택하는 방식으로 폼에 배치할 수 있습니다. 컨트롤이 일단 선택되면, 위에서 언급한 마지막 두 가지 방법을 통해 이러한 유형의 컨트롤을 폼에 배치할 수 있습니다.

Object Inspector 윈도우

Object Inspector 윈도우는 그림 10에 나와 있습니다. 델파이의 Object Inspector는 VB의 등록 정보와 밀접한 관련이 있습니다. 두 가지 모두 최근 선택된 오브젝트에 대하여 가능한 디자인 타임 속성의 목록을 보여주고 있습니다. 기본값으로 Object Inspector는 이 목록을 알파벳 순서로 표시합니다. 델파이는 속성을 카테고리별로 보여줄 수도 있습니다. 간단히 오른쪽 클릭만으로 팝업 메뉴에서 Arrange|by Category를 선택할 수 있습니다.

델파이의 오브젝트 속성에는 단순형, 열거형, 집합형 및 속성 편집기를 포함하는 것의 네 가지 기본 타입이 있습니다. 단순 속성은 속성 값을 키보드를 사용하여 직접 입력할 수 있습니다. 열거형 속성은 속성 값의 가능한 목록중에서 선택할 수 있습니다. 예를 들어 폼의 BorderStyle 속성은 열거형 속성입니다. 집합형 속성은 여러 값을 할당할 수 있게 해주는 단일 속성입니다. 오브젝트의 Font 속성 내에서 Style 속성이 집합형 속성입니다. Style 속성에서 가능한 값은 이탤릭,

강조체, 밑줄, 취소선 등의 값이 있습니다. VB처럼, 속성 편집기(Property Editors)를 이용하는 속성은 Object Inspector의 오른쪽에 생략부호(점 세개)가 나타납니다. 생략부호를 왼쪽 클릭하여 속성 편집기를 나타나게 할 수 있습니다.

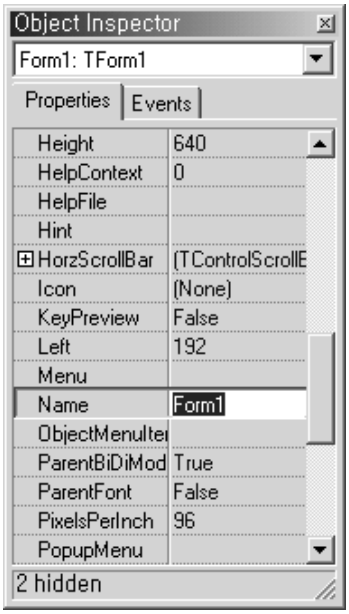

그림 10. *Object Inspector 윈도우*

Object Inspector는 객체에 가능한 디자인 타임 속성을 표시해줄 뿐 아니라, 객체가 반응할 수 있는 모든 종류의 이벤트를 탭으로 표시합니다. VB에서는 객체에 대해 가능한 모든 이벤트를 목록으로 보기위해서 코드 에디터를 사용하여 오브젝트 드롭다운 목록에서 객체를 선택한 후, 프로시저 드롭다운 목록에서 이벤트를 선택하였습니다. 델파이를 사용하면, 여러 컨트롤(또는 다른 종류의 이벤트)에서 같은 이벤트 핸들러(event-handler)를 호출할 수 있습니다. 이벤트 핸들러를 작성한 후, 오브젝트 인스펙터의 이벤트 탭을 사용하여 여러 컨트롤에 대해(혹은 다른 이벤트에 대해) 같은 이벤트 핸들러를 선택할 수 있습니다. 오브젝트 인스펙터에서 드롭다운 목록은 같은 매개변수 목록을 갖고 있는 모든 이벤트 핸들러를 표시합니다. 델파이는 객체 및 이벤트 핸들러에 대해서 매우 유연하며 강력한 언어임을 알 수 있습니다. VB에서 비슷한 작업을 하려면 한 이벤트 핸들러에서 다른 이벤트 핸들러를 호출해야 합니다.

코드 에디터 윈도우

VB의 코드 에디터는 하나의 새 윈도우에서 각 모듈을 엽니다. 델파이에서는 그림 11에서 보듯 코드 에디터가 단일 윈도우입니다. 이 윈도우는 열려 있는 유닛 및 모듈 각각에 대하여 탭을 포함하고 있습니다. 주의할 점은, 코드 편집이 끝난 후에 코드 에디터 윈도우를 습관적으로 닫기 쉽다는 것입니다. 델파이에서 유닛을 닫으면 유닛을 사용하는 폼까지 함께 닫힙니다. 단일 유닛 및 관련 폼까지 닫기 위해서는 코드 에디터 상의 탭을 오른쪽 클릭하고 팝업 메뉴에서 페이지 닫기를 선택해야 합니다.

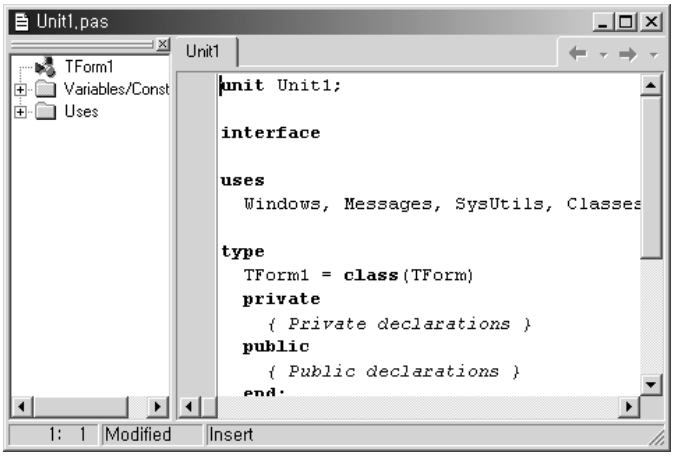

그림 11. *코드 에디터 윈도우*

델파이의 코드 에디터는 VB 편집기와 유사한 컬러 코딩(color-coding)을 사용합니다. 색을 변경하려면 Tools|Editor Option에서 Colors 탭을 선택합니다. 그리고 나서 바꾸고자 하는 컬러 요소를 선택합니다. 왼쪽 마우스 버튼으로 컬러를 선택하여 전경색을 변경하고 오른쪽 마우스 버튼으로 바탕색을 변경합니다. 굵은체나 기울임꼴로 표시할 수도 있습니다.

코드 에디터의 단축키는 표준 윈도우 네비게이션 키를 포함합니다.

키	기능
Home	라인 시작부분으로
End	라인 끝부분으로
Ctrl+Home	유닛 시작부분으로
Ctrl+End	유닛 끝부분으로
PgUp	이전 페이지
PgDn	다음 페이지
Ctrl+PgUp	현재 페이지의 맨 위

Ctrl+PgDn 현재 페이지의 맨 아래

VB의 IntelliSense 기능과 비슷하게, 델파이에는 개발자를 돕기 위해 Code Insight라는 다섯 가지 툴이 포함되어 있습니다.

1. 코드 컴플리션(Code Completion)은 객체 사용중에 변수를 선언하거나 속성이나 메소드의 목록을 알아보려 할 때 가능한 데이터 타입의 목록을 표시해줍니다. 데이터 타입, 속성이나 메소드를 입력할 때 델파이는 인크리멘털 검색으로 드롭다운 목록을 표시합니다. 이 목록은 기본값으로 스코프에 따라 정렬됩니다. 알파벳 순서로 표시하기 위해서는 드롭 다운 목록을 오른쪽 클릭하여 팝업 메뉴에서 Sort by Name을 선택합니다. 사용하고자 하는 항목을 정했으면 코드에 삽입하기 위해 엔터 키를 눌러 그 항목을 선택합니다.
2. 코드 파라미터(Code Parameter)는 함수, 메소드, 프로시저에 대한 매개변수 이름과 타입을 대화상자에 표시합니다. 그러므로, 코드에 입력하면서 기능, 메소드나 프로시저에 대해 요구되는 인자를 볼 수 있습니다.
3. 코드 템플릿(Code Template)은 VB에서 델파이로 마이그레이션할 때 가장 유용한 기능을 제공합니다. 코드 템플릿은 기본 코드 구조를 위한 구문 템플릿을 제공합니다. Ctrl-J를 누르면 이 기능이 활성화되어 가능한 템플릿들의 팝업 메뉴가 표시됩니다. 또한 구문의 처음에 타이핑을 한 후에 Ctrl-J를 누릅니다. 만약 델파이가 이 구문을 분석할 수 있으면 팝업메뉴는 코드 내에서 적용 가능한 템플릿으로 채워질 것입니다. 그러나 델파이가 이 구문을 해석할 수 없으면 구문에 가장 근접한 템플릿 목록이 표시됩니다. 코드 템플릿을 수정하거나 추가하려면 Tools|Editor 옵션에서 Code Insight 탭을 선택하십시오.
4. 툴팁 익스프레션 이밸류에이션(Tooltip Expression Evaluation)은 인터렉티브 디버깅 작업 도중에 툴팁 텍스트의 형태로 변수 값을 표시해줍니다.
5. 툴팁 심볼 인사이트(Tooltip Symbol Insight)는 코드 에디터에서 모든 식별자에 대해 선언 정보를 표시할 수 있습니다. 팝업 윈도우는 식별자의 종류(프로시저, 함수, 유형, 상수, 변수, 유닛 등) 뿐 아니라 선언의 유닛 파일과 줄 수를 표시합니다.

델파이에서는 열린 유닛이 각각 코드 에디터 상에서 개별 탭을 갖고 있습니다. 만약 편집하려는 유닛이 열리지 않았으면 메뉴에서 View|Units 및 View|Forms 중 하나를 선택하십시오. View|Units 옵션은 프로젝트에서 가능한 모든 목록을 표시합니다. 이와 비슷하게 View|Forms는 프로젝트에서 가능한 모든 폼을 표시합니다. 폼을 열면 연관된 유닛도 열립니다.

델파이의 코드 에디터에서는 북마크라는 또 다른 네비게이션 기능을 찾아볼 수 있습니다. 델파이는 코드 에디터에 (0)에서 (9)까지 10개의 북마크를 쓸 수 있습니다. 북마크를 토글하려면 원하는 코드라인에 커서를 놓고 Shif-Ctrl-*숫자*를 누릅니다. 여기서 숫자는 숫자 키 0에서 9의 숫자 중 하나입니다. 숫자를 단 북마크로 직접 가려면 Ctrl-*숫자*를 누르십시오.

기타 윈도우

기본값으로 나타나는 세 개의 윈도우 외에 델파이는 몇 가지 유용한 윈도우들을 제공합니다. 그림 11에서 코드 익스플로러(Code Explorer) 윈도우는 디폴트 위치에 배치되어 코드 에디터 윈도우 내에서 활성 에디터 페이지 탭의 좌측에 위치합니다. 다시 말해서 이 윈도우는 코드 에디터 윈도우의 좌측에 도킹됩니다. 코드 익스플로러는 프로그래머가 유닛 파일을 통해 쉽게 탐색할 수 있도록 돕습니다. 여기에는 최근 코드 에디터 윈도우에서 편집된, 코드 유닛에서 선언한 모든 유형과 클래스, 속성, 메소드, 전역 변수, 전역 루틴의 모든 유형을 세 가지 다이어그램으로 표시합니다. 또한 최근 편집된 유닛에서 이용한 다른 유닛을 나열합니다. 아무 것도 보이지 않으면 View|코드 익스플로러를 선택하여 코드 익스플로러 윈도우를 여십시오.

프로그래머는 프로젝트 매니저 윈도우에서 델파이 프로젝트를 작성한 파일을 볼 수 있습니다. 프로젝트 매니저에서 델파이 프로젝트는 프로젝트 그룹으로 배열되는데, 여기서 프로젝트 그룹은 관련된 프로젝트나 멀티 티어 애플리케이션의 일부로 함께 동작하는 프로젝트로 구성됩니다. 또한 이 윈도우를

통해 프로그래머는 다양한 프로젝트 및 프로젝트 그룹 내의 각 프로젝트 구성 파일 사이를 쉽게 오갈 수 있습니다. View|Project 관리자를 선택하거나 Ctrl-Alt-F11를 눌러서 프로젝트 매니저 윈도우를 여십시오.

성공적으로 프로그램을 완성하기 위해서, 특히 대규모 프로그램일 경우에는 프로그래머가 수행해야 할 작업이 많아집니다. 델파이의 To-Do 리스트는 프로그래머가 이러한 태스크를 구성할 수 있도록 내장 메모장을 제공하고 있습니다. 이 목록은 프로그래머로 구성된 팀에서 작성한 대규모 프로젝트를 계획, 프로그래밍, 테스팅, 디버깅하는데 큰 도움이 됩니다. 메뉴 바에서 View|To-Do List를 선택하여 이 윈도우를 여십시오. 그러면 To-Do List 윈도우에서 오른쪽 클릭하여 목록 항목을 추가, 편집, 삭제할 수 있습니다.

얼라인먼트 팔레트 윈도우는 폼에 컴포넌트들을 배열하기 위한 빠른 방법을 제공합니다. 메뉴 바에서 View|Alignment Palette를 선택하여 이 윈도우를 여십시오.

프로젝트 브라우저는 현재 프로젝트에서 선언하거나 사용한 유닛, 클래스, 유형, 속성, 메소드, 변수 및 루틴을 나열합니다. 프로젝트 브라우저는 트리 구조로 정보를 배열합니다. View|Browser나 Shift-Ctrl-B를 눌러서 Project Browser를 여십시오.

Component List (Components 윈도우)는 메뉴 바의 View|Component List를 선택하여 열 수 있습니다. 이 윈도우는 사용자의 델파이 버전에서 가능한 모든 컴포넌트의 목록을 알파벳 순서대로 표시합니다. 키보드나 마우스로 이 윈도우로부터 컴포넌트를 선택하여 델파이 프로그램에 이를 추가할 수 있습니다. 컴포넌트 팔레트는 기능에 따라 컴포넌트들을 정리하고 있으므로 여기서 마우스로 컴포넌트를 선택하고 애플리케이션에 배치하는 것이 더 빠른 방법입니다. 그러므로 컴포넌트 윈도우 대신에 컴포넌트 팔레트를 사용할 것을 권합니다.

윈도우 목록은 델파이 IDE에서 윈도우 사이를 더 빨리 전환할 수 있도록 도와줍니다. 이는 윈도우가 여러 개 열려 있는 경우에 윈도우를 찾고 활성화하는 데 유용한 방법입니다. 윈도우 목록을 선택하고 OK 버튼을 클릭하십시오.

델파이는 내장 디버거와 관련하여 몇가지 윈도우를 포함하고 있습니다. 문서의 디버깅 섹션에서 이와 관련된 사항을 논하도록 하겠습니다.

프로그래밍 언어

이 섹션에서는 델파이 파일 타입과 오브젝트 파스칼 언어 구문에 대해서 알아보겠습니다.

델파이의 파일 타입들

VB와 마찬가지로 델파이 애플리케이션은 (프로젝트라고 부름) 여러 다양한 파일 타입들로 구성됩니다. 세 가지 주요 파일 타입에는 프로젝트 파일, 유닛 파일, 폼 파일이 있습니다. 프로젝트 파일은(.DPR) "메인 프로그램"으로, 델파이 프로젝트를 구성하는 유닛 파일과 폼 파일을 포함합니다. 그러므로 프로젝트 파일은 특정 프로젝트와 관련한 파일들을 서로 연결시켜주는 역할을 합니다. 일반적으로 유닛 파일과 폼 파일은 일대일 관계를 이룹니다. 각 유닛 파일은 관련 폼 파일과 일대일로 대응됩니다. 폼 파일은(.DFM) 폼 상에서 오브젝트와 오브젝트 속성 설정을 나열하고 유닛 파일은(.PAS) 폼과 관련된 소스코드를 포함하고 있습니다. 이에 비해 VB는 오브젝트, 속성 설정 및 코드를 폼 파일(.FRM)만으로 결합합니다.

중요한 점은 델파이 애플리케이션에 대한 프로젝트 및 폼을 모두 저장해야 한다는 것입니다. 폼을 저장할 때 유닛 파일과 폼 파일은 적절한 파일 확장자를 갖는 같은 이름으로 저장되어야 합니다. 폼을 저장하기 위해서는 File | Save나 File |Save As를 선택하고 표준 툴바 상의 Save 아이콘을 클릭하거나 Ctrl-S를 누르십시오. 프로젝트 파일을 저장하기 위해서는 메뉴 툴바에서 File | Save Project As를 선택하십시오. 프로젝트 및 관련된 파일을 빠르게 저장하려면 File | Save All을 메뉴에서 선택하거나 표준 툴바에서 Save All 아이콘을 클릭하십시오.

프로젝트 파일

다시 한번 각 애플리케이션은 프로젝트 파일에서

"메인 프로그램"으로 구성된다는 점을 밝힙니다. 이 파일은 애플리케이션에 관련된 모든 유닛을 연결합니다.

프로젝트 파일의 일반적인 형태는 다음과 같습니다.

```
program Project1;
uses
Forms,
Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

위에서 본 것처럼 프로젝트 파일은 다음 세 가지 요소로 구성됩니다.

1. 프로그램 헤딩
2. uses 절 (옵션)
3. begin과 end 키워드 사이의 선언과 구문 블록

프로그램 헤딩은 프로그램에 대한 이름을 기술합니다. uses 절은 프로그램에서 사용하는 유닛 파일을 나열합니다. 블록(begin과 end 키워드 사이)은 프로그램이 동작할 때 실행되는 선언과 구문을 포함합니다. 델파이 IDE에서는 이 세 요소를 단일 프로젝트 파일에서 찾을 수 있습니다.

마지막으로 주의할 점은 IDE에서 생성된 코드를 수동으로 바꾸지 말아야 합니다. 이는 프로젝트 파일에만 적용되는 것이 아니라 유닛 파일에도 해당됩니다. 다시 말해 IDE가 작업하도록 놓아두는 것이 좋습니다. 예를 들어 프로젝트에 새로운 폼을 추가하려면 File|New Form을 선택하거나 View 툴바 상의 New Form 버튼을 클릭하십시오. 또한 이벤트 핸들러를 유닛 파일에 추가하려면 오브젝트 및 이벤트를 Object Inspector에서 적절한 폼으로 선택하여 이벤트 핸들러용 이름을 입력하십시오. 델파이는 이벤트 핸들러에 대한 헤딩과 블록을 자동으로 생성합니다.

유닛(Units)

델파이 유닛은 데이터 타입들(클래스 포함), 상수 및 변수 선언, 서브루틴(함수, 프로시저)으로 구성됩니다. 각 유닛은 개별 유닛 파일에 존재합니다.

유닛 파일은 헤딩, 인터페이스, 구현, 초기화, 종결화(finalization) 섹션을 포함합니다. 초기화 및 종결화 섹션은 선택 사양입니다. 프로젝트 파일처럼 유닛 파일은 마침표가 따라붙는 end 키워드로 종결되어야 합니다. 유닛 파일의 일반 구문은 다음과 같습니다.

```
unit Unit1;

interface

uses {List of used units goes here}

const
  {Public constants go here}

type
  {Public types go here}

var
  {Public variables go here}

{Remainder of interface section goes here}

implementation

uses {List of used units goes here}

const
  {Private constants go here}

type
  {Private types go here}

var
  {Private variables go here}

{Remainder of implementation section goes
here}

initialization
{Initialization section goes here}

finalization
{Finalization section goes here}
end.
```

인터페이스 섹션은 예약어 interface로 시작하여, 임플멘테이션 섹션이 시작될 때까지 계속됩니다. 인터페이스 섹션은 선언 시 유닛을 사용하는 클라이언트에서 가능한 상수, 데이터 타입, 변수 및 서브 루틴을 정의합니다. 그러므로 인터페이스 섹션에서 표시되는 엔티티는 모두 퍼블릭 범위(public scope)를 갖고 있는데 이는 마치 참조하는 쪽에서 엔티티가 선언된 것처럼 참조 코드가 접근할 수 있게 하기 위해서 입니다.

임플멘테이션 섹션은 예약어 implementation으로 시작해서 초기화 섹션이(존재하는 경우) 시작되거나

유닛이 끝날 때까지 계속됩니다. 임플멘테이션 섹션은 유닛에 private한 상수 및 데이터 타입, 변수, 서브 루틴을 선언합니다. 이러한 엔티티는 전용 유효범위(private scope)를 가지고 있으며 외부에서는 접근할 수 없습니다.

프로시저 및 함수의 인터페이스 선언은 루틴의 헤딩만을 포함합니다. 임플멘테이션 섹션에서는 루틴의 블록이 뒤따릅니다. 그러므로 인터페이스 섹션의 프로시저 및 함수는 포워드 선언처럼 작동합니다.

인터페이스 및 임플멘테이션 섹션은 섹션 헤딩(interface 및 implementation 키워드) 직후 표시되는 자신만의 uses 절을 포함할 수도 있습니다. uses 절은 사용되는 유닛을 기술합니다. System 유닛은 모든 델파이 애플리케이션에 의해 자동적으로 사용되며, uses 절 내에 명시적으로 나열될 수 없습니다. System 유닛은 파일 입력과 출력에 대한 루틴, 문자열 처리, 부동 소수점 연산, 동적 메모리 할당 등을 구현합니다. SysUtils 같은 기타의 표준 라이브러리 유닛은 uses절에 포함되어야 합니다. 대부분의 경우 델파이는 소스 파일을 생성하고 유지할 때, 모든 필수적인 유닛을 uses 절에 배치합니다.

유닛 이름은 프로젝트 내에서는 고유해야 합니다. 유닛 파일이 서로 다른 디렉터리에 위치하여도 같은 이름의 두 유닛은 한 프로그램 안에서 사용될 수 없습니다.

프로그래밍의 요소

이제 델파이 파일 타입의 목적과 구성에 대해 어느 정도 익숙해졌을 것입니다. 지금부터는 프로그래밍 요소에 초점을 맞추어 대부분의 고급 언어에서 공통적인 요소들에 대해 알아보겠습니다.

주석

VB는 Rem(remark 구문)과 어퍼스트로피(')의 두가지 주석문을 갖고 있습니다. VB 컴파일러는 이 두 가지 주석문을 항상 무시합니다. 오브젝트 파스칼에서도 컴파일러가 주석문을 무시하지만, 주석문이 분리자(인접 토큰을 구분)나 컴파일러 지시문의 역할을 하는 경우는 예외입니다. 다음은 주석문을 작성하는 방법들입니다.

- { 왼쪽 중괄호와 오른쪽 중괄호 사이의 구문은 주석문이 됩니다 }
- (* 왼쪽 괄호에 별표를 붙인 것과 오른쪽 괄호 앞에 별표가 있는 것 사이의 구문은 주석문이 됩니다 *)
- // 두개의 슬래시와 라인의 끝 사이의 구문은 주석문

{ 또는 (* 바로 뒤에 달러 표시($)가 포함된 주석은 컴파일러 지시문입니다. 컴파일러 지시문은 코드 내에서 컴파일러 옵션을 변경하기 위한 것으로, 실행할 수 없는 문장입니다. 예를 들어 {$WARNINGS OFF}는 오브젝트 파스칼 컴파일러가 경고 메시지를 생성하지 않게 합니다. VB에서는 Option 문이 컴파일러 지시문을 지칭합니다.

문장 종결(Statement Termination)

VB에서는 다음 라인에서 문장을 이어가기 위해서 라인 연속 문자(underscore: 밑줄)를 사용한 경우를 제외하고는 라인의 끝에서 문장이 끝납니다. 오브젝트 파스칼은 라인 연속 문자 없이 문장이 여러 라인에 걸쳐 연결될 수 있습니다. 세미콜론(;)은 문장 구분자이자 종결자가 되어 한 문장과 다음 문장 사이를 구분합니다.

식별자(Identifiers)

오브젝트 파스칼 식별자는 상수, 변수, 필드, 데이터 타입, 속성, 프로시저, 함수, 프로그램, 유닛, 라이브러리, 패키지 등을 표시합니다. 식별자는 어떠한 길이도 표시할 수 있지만, 처음 255 문자만 의미가 있습니다. 식별자의 첫번째 문자는 영문자 혹은 밑줄이어야 합니다. 첫 문자 뒤에는 영문자나 숫자, 밑줄이 몇 개라도 따라올 수 있습니다. 식별자는 공백 문자를 포함할 수 없으며, 예약어는 식별자로 사용할 수 없습니다.

오브젝트 파스칼은 대소문자를 구분하지 않기 때문에 FindItem라는 이름을 갖는 식별자는 finditem, findItem, Finditem, FINDITEM 같은 다양한 방식으로 쓰일 수 있습니다.

데이터 타입

데이터 타입은 변수가 가질 수 있는 데이터의 종류를 기술합니다. 오브젝트 파스칼에서 사전

정의(내장된) 데이터 타입은 표 1에 요약되어 있습니다. 표에는 각 데이터 타입의 유효 범위 값과 필요한 메모리 공간이 표시되어 있습니다. 이 데이터 타입들은 델파이의 윈도우 버전에 적용한 것임을 주의하십시오. Varient 데이터 타입은 윈도우만 존재하기 때문에, 리눅스 버전에는 포함되어 있지 않습니다.

논리적 및 숫자 데이터 타입

데이터 타입	범위	포맷 또는 크기	유효 자리
Shortint	–128 ~ +127	signed 8-bit	
Smallint	–32768 ~ +32767	signed 16-bit	
Integer (or Longint)	–2147483648 ~ +2147483647	signed 32-bit	
Int64	$-2^{63} \sim +2^{63}-1$	signed 64-bit	
Byte	0 ~ 255	unsigned 8-bit	
Word	0 ~ 65535	unsigned 16-bit	
Longword (or Cardinal)	0 ~ 4294967295	unsigned 32-bit	
Boolean (or ByteBool)	True or False	1 byte	
WordBool	True or False	2 byte	
LongBool	True or False	4 byte	
Real48	$2.9 \times 10^{-39} \sim 1.7 \times 10^{38}$	6 byte	11 to 12
Single	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$	4 byte	7 to 8
Real (or Double)	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$	8 byte	15 to 16
Extended	$3.6 \times 10^{-4951} \sim 1.1 \times 10^{4932}$	10 byte	19 to 20
Comp	$-2^{63}+1 \sim +2^{63}-1$	8 byte	19 to 20
Currency	–922337203685477.5808 ~ +922337203685477.5807	8 byte	19 to 20

문자 및 문자열 데이터 타입

데이터 타입	최대 길이	메모리 요구량
Char (or AnsiChar)	1 ANSI character	1 B
WideChar	1 Unicode character	2 B
ShortString	255 ANSI character	2 to 256 B
String (or AnsiString)	2^{31} ANSI character	4 B to 2 GB
WideString	2^{30} Unicode character	4 B to 2 GB

표 1. *오브젝트 파스칼 데이터 타입*

상수

상수는 프로그램 실행 전반에 걸쳐 일정 값을 유지하는 항목으로, 어떠한 수학적 표현이나 문자열 표현으로도 정의될 수 있습니다. 컴파일러는 컴파일하는 동안 상수 이름을 다른 관련 값으로 대체합니다. 수치 상수는 숫자나 수치 문자를 의미하는 것이며 문자열 상수는 문자열 문자를 의미합니다. 예를 들어 숫자 7은 수치 상수이며 'days per week'는 문자열 상수입니다.

오브젝트 파스칼에서는 상수를 정의하는 데 `const` 문을 사용합니다. VB에서처럼 `const` 문은 상수 그룹을 정의할 수 있습니다. 이 문의 일반적인 폼은 다음과 같이 표시됩니다.

```
const
  constantName = Expression;
  [constantName = Expression;]
```

위의 일반 구문에서는 옵션 항목을 지정하기 위해 몇 번이든 표시할 수 있는 꺽쇠 괄호 ([…])를 사용하고 있습니다. 예를 들어 다음의 코드 단락은 세 가지 상수를 정의하고 있습니다.

```
const
  MINS_PER_HR = 60;
  HRS_PER_DAY = 24;
  DAYS_PER_WK = 7;
```

VB에서, 이런 상수들은 다음과 같은 구문으로 정의합니다.

```
Const MINS_PER_HR = 60, HRS_PER_DAY = 24, _
      DAYS_PER_WK = 7
```

변수

변수는 값이 저장되는, 메모리에 지정된 위치를 의미합니다. 이러한 값은 프로그램 실행 전반에 걸쳐 변경될 수 있습니다.

VB가 변수를 정의하기 위해서 `Dim (dimension)` 문을 사용하는 반면, 오브젝트 파스칼은 `var` 문을 사용합니다. `var` 문의 일반 구문은 다음과 같습니다.:

```
var
variableName[, variableName]: DataType;
[variableName[, variableName]: DataType;]
```

여기서 *DataType*은 사전 정의되거나 사용자가 정의한 데이터 타입입니다. 예를 들어 다음과 같은 VB 변수 정의를 살펴보겠습니다.

```
Dim dollars As Integer, cents As Integer
Dim cost As Double
Dim myMessage As String
```

델파이에서의 동일한 의미의 변수 선언은 다음과 같습니다.

```
var
dollars: Integer;
cents: Integer;
cost: Real;
myMessage: String;
```

연산자

VB에서는 등호(=)는 오버로드된 연산자로서, 지정 연산자 및 일치 여부를 결정하는 비교 연산자 양쪽 모두의 기능을 합니다. 그러나 오브젝트 파스칼에서 등호는 일치 여부를 결정하는 비교 연산자로서의

역할만 하기 때문에, 일치 여부를 결정하기 위해 양쪽 변수의 내용을 항상 비교하게 됩니다. 오브젝트 파스칼 지정 연산자는 콜론(:)과 등호(=)의 결합, 즉 := 입니다.

이제 VB 코드 단락을 살펴보겠습니다.

```
Dim value1 As Integer, value2 As Integer Dim
check As Boolean
value1 = 5
value2 = 7
check = (value1 = value2)
```

이 코드에서 value1과 value2는 5 및 7이라는 두 값에 각각 할당됩니다. Boolean 변수 check 는 value1가 value2와 같은지 여부에 따라 값을 할당받습니다. 이 경우 check는 변수 값이 같지 않기 때문에 False가 할당됩니다. 동등한 오브젝트 파스칼 코드는 다음과 같습니다.

```
var
  value1: Integer;
  value2: Integer;
  check: Boolean;

begin
  value1 := 5;
  value2 := 7;
  check := (value1 = value2);
end;
```

오브젝트 파스칼의 산술 연산자 및 관계 연산자는 다음 두 개의 표에 표시되어 있습니다. 표 2는 산술 연산자를, 표 3은 관계 연산자를 표시하고 있는데, 여기서 관계 연산자는 VB와 동일합니다. 산술 연산자 역시 VB와 동일하지만 정수 나눗셈과 나머지 구하기, 지수화에서는 예외입니다. 오브젝트 파스칼에서 정수 나눗셈과 나머지 구하기의 경우에는 내장 연산자를 갖고 있지만 지수화에 대한 연산자는 포함되어 있지 않습니다. 프로그래머는 지수화를 수행하기 위해서 수학 라이브러리 내의 함수를 호출해야 합니다.

연산	연산자	피연산자 타입	결과 타입	예
Sign Identity	+ (단항)	integer, real	integer, real	+x
Sign Negation	- (단항)	integer, real	integer, real	-x
Mutliplication	*	integer, real	integer, real	x * y
Division	/	integer, real	real	x / y
Integer Division	div	integer	integer	x / div y
Modulo Division	mod	integer	integer	x mod y
Addition	+	integer, real	integer, real	x + y
Subtraction	-	integer, real	integer, real	x – y

표 2. *오브젝트 파스칼 산술 연산자*

관계 연산자	오브젝트 파스칼	수학식
Less than	<	<
Less than or equal to	<=	≤
Greater than	>	>
Greater than or equal to	>=	≥
Equal to	=	=
Not equal to	<>	≠

표 3. *오브젝트 파스칼 관계 연산자*

오브젝트 파스칼의 논리 연산자에는 and, or, not, xor가 있습니다. 기본적으로, 오브젝트 파스칼에서는 and 및 or 연산자에 대해 단축 계산을 수행할 수 있습니다. 이는 최종 값을 결정하기 위해 필요한 만큼의 식만 계산하는 것을 뜻합니다. 이러한 식을 완전히 계산하기 위해서는 Project|Options를 선택하고 Complier 탭을 클릭합니다. 그리고 나서 Syntax 옵션 프레임에서 "Complete boolean eval"을 클릭합니다. 이외에 {$B+} 컴파일러 지시문을 코드에 삽입하는 방법도 있습니다.

VB에서처럼, 델파이의 문자열 연산은 문자열 연결 한가지 뿐입니다. VB는 앰퍼센드(&), 덧셈 기호(+)의 두가지 연산자를 이용해 문자열 연결을 하지만 역할은 같습니다. 오브젝트 파스칼에서 덧셈 기호(+)만을 문자열 연결에 사용합니다. 그러므로 덧셈 기호(+)는 오브젝트 파스칼에서 오버로드된 연산자이며, 부호 지정, 덧셈, 문자열 연결에 사용됩니다.

문자열에 있어서 VB와 오브젝트 파스칼의 또 다른 차이점은 문자열 구분 문자에 있습니다. 문자열을 구분하는 데 있어 VB는 큰 따옴표(")를 사용하는데 비해 오브젝트 파스칼에서는 작은 따옴표(')를 사용합니다. 예를 들어 문자열과 문자열 연결 연산자를 사용하는 코드 라인은 다음과 같습니다.

```
myName := 'Mitchell' + ' ' + 'Kerman';
```

이 코드 라인은 다음과 동일합니다.

```
myName := 'Mitchell Kerman';
```

결정 구조(Decision Structures)

VB와 같이 델파이의 오브젝트 파스칼은 두 가지 유형의 결정 구조를 갖고 있는데, 바로 if문과 case문입니다. 두 언어 모두 비슷한 구조를 갖고 있습니다.

if 문

거의 모든 고급 언어들은 if 문을 갖고 있습니다. if 문에 있어서 VB와 델파이의 주된 차이점은 오브젝트

파스칼이 복합문 형태에서 여러 줄의 코드를 필요로 하며, 여기서 복합문은 begin과 end 키워드에 의해 구분된다는 점입니다. 간단히 말해 복합문이 하나의 문장으로 구성된 경우라도 항상 복합문을 사용한다는 뜻입니다. 이는 장기적으로 볼 때, 구문상의 여러 가지 문제를 해결합니다. 예를 들어 사용자가 하나의 조건 아래에서 문장의 수를 증가시키려 하면 키워드가 이미 제자리에 위치하고 있기 때문에 시작과 끝 키워드 추가를 잊지 않을 수 있습니다. 필자는 이러한 이유로 이 관습 채택을 권장합니다.

오브젝트 파스칼 if 문의 일반적 형태는 다음과 같습니다.

```
if condition1 then begin
  [statements1;]
end
else if condition2 then begin
  [statements2;]
end
       .
       .
       .
else if conditionN then begin
  [statementsN;]
end
else begin
  [statementsX;]
end;
```

if 문은 여러 개의 else if 절을 가질 수 있지만 하나의 else 절을 포함할 수도 있습니다. 계산을 할 때 if 문은 VB와 같은 방식으로 동작하게 됩니다. *statements1*은 *condition1*이 True일 때 실행됩니다. *statements2*는 *condition1*이 False이고 *condition2*가 True일 때 실행됩니다. *statementsN*은 *conditionN*이 True이고 다른 모든 전제 조건이 (*condition1*에서 *condition*{*N-1*}까지) False일 때 실행됩니다. 결국 *statementsX*는 모든 조건이(*condition1*에서 *condition*{*N-1*}까지) False일 때 실행됩니다.

다음 예제 코드는 점수 차이에 기초하여 친구의 골프 핸디캡을 계산합니다.

```
difference := yourAverageScore-myAverageScore;
if (difference >= 10) then begin
  handicap := 5;
end
else if (difference >= 7) then begin
  handicap := 3;
end
else if (difference >= 4) then begin
  handicap := 2;
end
else begin
  handicap := 0;
end;
```

이 코드의 동일한 내용의 VB는 다음과 같습니다.

```
difference = yourAverageScore-myAverageScore
If (difference >= 10) Then
  handicap = 5
ElseIf (difference >= 7) Then
  handicap = 3
ElseIf (difference >= 4) Then
  handicap = 2
Else
  handicap = 0
End If
```

여기서는 언급할 만한 몇 가지 중요한 구문상의 차이가 있습니다. VB에서 ElseIf는 키워드이지만 오브젝트 파스칼에서는 if에 이어지는 else의 의미로, 두 개의 다른 단어가 됩니다. 또한 오브젝트 파스칼에서는 세미콜론(;)을 사용하여 프로그램 문을 종료하기 때문에 VB의 end If 문에 해당하는 구문이 없습니다. 이제 위에 언급한 델파이 구문에서 세미콜론의 위치를 주목하십시오. else문 바로 앞에는 세미콜론이 없는데, 이는 구문 에러의 원인이 될 수 있기 때문입니다.

case 문

델파이의 case문은 VB의 Select Case 문과 매우 흡사하지만 주된 차이점은 VB Select Case 문은 문자열과 실수값을 테스트할 수 있다는 것입니다. 델파이의 case문은 정수 및 문자값과 같은 서수 데이터타입(ordinal data type)만 테스트할 수 있습니다. 만약 문자열이나 실수값을 테스트할 필요가 있으면 델파이의 if문을 사용해야 합니다.

오브젝트 파스칼 case 문장의 구문은 다음과 같습니다.

```
case selectorExpression of
  caseList1: begin
               statements1;
             end;
  caseList2: begin
               statements2;
             end;
            .
            .
            .
  caseListN: begin
               statementsN;
             end;
  else begin
               statementsX;
             end;
end;
```

이 구문에서 *selectorExpression*은 각 *caseList* 식과 비교되는 식입니다.

*selectorExpression*은 Integer, Char, Boolean을 포함하는 순서형의 표현식이 되어야 합니다. 또한 *caseList* 에서 표현식은 서수식(Ordinal Expression)이 되어야 컴파일 시간에 계산이 가능합니다. 예를 들어 12, True, 4 – 9 * 5, 'X', Integer('Z')는 유효한 *caseList* 식이지만 변수들 및 대부분의 함수 호출은 사용할 수 없습니다. *caseList* 는 또한 *firstExpr..lastExpr* 형태의 부분범위가 될 수 있으며 여기서 *firstExpr* 및 *lastExpr*는 *firstExpr* ≤ *lastExpr*인 서수식 형태입니다. 마지막으로 *caseList*는 *expr1*, *expr2*, …, *exprN* 형태의 목록으로, 여기서 각 *expr*는 서수식이나 위에서 언급한 부분범위입니다.

하나의 case문은 여러 개의 *caseList*를 가질 수 있지만 else 절은 하나만 가질 수 있습니다. case문의 실행은 if 구문 구조의 실행과 유사합니다. 만약 *selectorExpression*가 *caseLis*의 식과 일치한다면 뒤에 나오는 문장에서 *caseList*가 실행되면서 컨트롤이 *case*문 다음 코드에 전달됩니다. 만약 *selectorExpression*이 하나 이상의 *caseList* 식에 일치하지 않는다면, 처음으로 일치하는 *caseList* 식 다음의 문장만 실행됩니다. 만약 *selectorExpression*이 어떠한 *caseList* 식에도 일치하지 않는다면 else 절 다음 문장인 *statementsX*가 실행됩니다. *case*문에서 else 절을 필요로 하지 않는다 해도 else 문을 사용하면 코드가 우연한 *selectorExpression* 값을 처리할 수 있게 됩니다. 만약 *selectorExpression* 이 어떠한 *caseList* 식에도 일치하지 않고 else절도 포함하지 않는다면 case 문 다음에 나오는 코드로 실행은 계속됩니다.

아래 코드에서는 이전의 골프 핸디캡 예제가 case 문으로 바뀌어 있습니다. 이 코드는 여러분의 점수와 친구의 점수차가 최대 126타까지 벌어질 수 있다고 가정합니다.

```
difference := yourAverageScore-myAverageScore;
case difference of
  4, 5, 6: begin
             handicap := 2;
           end;
  7, 8, 9: begin
             handicap := 3;
           end;
  10..126: begin
             handicap := 5;
           end;
  else begin
    handicap := 0;
end;
end;
```

비교를 위해, 다음 코드는 동일한 내용의 VB 코드입니다.

```
difference = yourAverageScore - myAverageScore
Select Case difference
  Case 4, 5, 6
    handicap = 2
  Case 7, 8, 9
    handicap = 3
  Case 10 To 126
    handicap = 5
  Case Else
    handicap = 0
End Select
```

반복 구조

루프라고도 부르는 반복 구조는 제한적이거나 무한대일 수 있습니다. 제한적인 반복 구조에서는 루프의 횟수를 알 수 있거나 계산될 수 있습니다. 무한정 반복 구조에서는 루프가 몇 번이나 실행되었는지 알려질 필요가 없습니다.

VB에서는 Do 루프가 무한 루프 구조인 반면, For 루프가 제한적인 루프 구조를 이룹니다. VB의 Do 루프 구조는 Do While…Loop, Do…Loop Until, Do Until…Loop, Do…Loop While를 포함합니다. 오브젝트 파스칼의 for 루프는 제한적인 루프 구조이며 `while`과 `repeat` 루프는 무한 루프 구조입니다. 다음 단락에서는 이러한 루프 구조에 대해 논합니다.

for 루프

VB에서 For 루프의 루프 제어 변수는 정수형과 실수형을 포함한 어떤 종류의 수치 데이터 타입도 가능합니다. 오브젝트 파스칼에서 `for` 루프의 루프 제어 변수는 반드시 서수 타입이어야 합니다. 또한 어떠한 스텝 값도 오브젝트 파스칼에서는 사용되지 않으며 `for` 루프는 구문에 따라 항상 다음 서수 값으로 증가하거나 감소하게 됩니다.

다음 VB 코드는 1에서 100까지의 합을 구하는 점증하는 For 루프를 사용합니다.

```
Dim counter As Integer, sum As Integer

sum = 0
For counter = 1 To 100
  sum = sum + counter
Next counter
```

오브젝트 파스칼에서 점증식 for 루프의 일반 구문은 다음과 같습니다.

```
for counter := start to finish do begin
  [statements;]
end;
```

위 VB 코드를 오브젝트 파스칼 구문으로 변환하면 다음과 같습니다.

```
var
  counter: Integer;
  sum: Integer;
begin
  sum := 0;
  for counter := 1 to 100 do begin
    sum := sum + counter;
  end;
end;
```

점증식과 반대인 점감식 For 루프를 사용하여 1에서 100까지의 정수 합을 쉽게 계산할 수 있습니다. 필요한 VB 코드는 다음과 같습니다.

```
Dim counter As Integer, sum As Integer

sum = 0
For counter = 100 To 1 Step -1
  sum = sum + counter
Next counter
```

오브젝트 파스칼의 점감식 For 루프에 대한 일반 구문은 다음과 같습니다.

```
for counter := start downto finish do begin
  [statements;]
end;
```

오브젝트 파스칼의 점감식 루프 구조를 다시 작성하면 다음과 같습니다.

```
var
  counter: Integer;
  sum: Integer;
  begin
  sum := 0;
  for counter := 100 downto 1 do begin
    sum := sum + counter;
  end;
end;
```

while과 repeat 루프

델파이의 while과 repeat 루프는 무한 루프 구조를 갖고 있습니다. while 루프는 VB의 Do While... Loop에 해당하며 repeat 루프는 VB의 Do…Loop Until에 해당합니다. VB의 Do Until…Loop, Do…Loop While 구조와 등가를 이루는 요소가 델파이에는 없지만 기존 보유 구조의 논리 조건을 부정함으로써 다른 형태로 쉽게 전환된다는 점에 알아두십시오.

while 루프는 다음과 같은 일반 구문을 갖습니다.

```
while condition do begin
  [statements;]
end;
```

while 루프는 조건이 True인 동안 루프를 실행합니다. 이러한 유형의 루프는 루프 시작에서 조건을 테스트하고, 첫 번째 루프가 실행 되기 전에 조건이 False가 되면 루프는 결코 실행되지 않습니다.

다음은 while 루프를 사용하여 1부터 100까지 더하는 코드입니다.

```
sum := 0;
count := 1;
while (count <= 100) do begin
  sum := sum + count;
  count := count + 1;
end;
```

이 코드와 대응하는 내용의 VB는 다음과 같습니다.

```
sum = 0
count = 1
Do While (count <= 100)
  sum = sum + count
  count = count + 1
Loop
```

repeat 루프의 구문은 다음과 같습니다.

```
repeat
  [statements;]
until condition;
```

repeat 루프는 조건이 True가 될 때까지 코드 블록을 실행합니다. 이 루프는 루프 끝에서 조건을 테스트하고, 최소한 한번은 실행됩니다. 이 루프 구조에는 시작과 끝 키워드가 필요하지 않음을 알려드립니다. 루프는 repeat 및 until 키워드에 의해서 구분됩니다.

다음 코드는 repeat 루프를 사용하여 1부터 100까지 더하는 코드입니다.

```
sum := 0;
count := 1;
repeat
  sum := sum + count;
  count := count + 1;
until (count > 100);
```

다음은 대응하는 VB 코드입니다.

```
sum = 0
count = 1
Do
```

```
  sum = sum + count
  count = count + 1
Loop Until (count > 100)
```

break문과 continue문

VB에서는 Exit Do문이나 Exit For문을 사용하면 프로그램이 무조건적으로 루프 구조에서 벗어날 수 있습니다. 이 두 가지 문은 컨트롤을 루프 구조 다음의 문에 전달할 수 있습니다. 델파이에서 같은 작업을 수행하기 위해서는 break 문을 사용합니다. 델파이에는 VB에서는 불가능한 continue 문이 포함되어 있습니다. 이 문장은 루프의 나머지 부분을 건너뛰어 루프 시작으로 제어권을 전달합니다.

서브프로그램

다른 고급 언어들처럼 오브젝트 파스칼은 프로시저 및 함수라는 두 가지 유형의 sub프로그램을 가지고 있습니다. 프로시저 및 함수는 오브젝트 파스칼에 내장된 기능은 아니며 사용자 정의라는 특징을 지니는데 이는 프로그래머들(컴파일러 사용자)이 이를 정의해야 하기 때문입니다. 델파이에서 프로시저는 VB의 하위 프로시저에 해당합니다. 델파이에서 함수는 VB의 함수와 비슷한 방식으로 동작합니다.

VB에서 이벤트 핸들러는 실제로는 서브 프로시저입니다. 델파이에서 이벤트 핸들러는 이벤트가 관련 오브젝트 상에 발생할 때, 자동으로 호출되는 프로시저입니다. 예를 들어 다음의 VB 이벤트 핸들러는 사용자 입력 값의 제곱근을 찾아 표시하고 있습니다.

```
Private Sub cmdComputeSqrRt_Click()
Dim value As Double

  value = Sqr(Val(txtInputNumber.Text))
  picOutput.Print "The square root of " & _
                  txtInputNumber.Text & _
                  " is " & CStr(value)
End Sub
```

다음은 동일한 이벤트 핸들러를 델파이로 작성한 것입니다.

```
procedure TfrmSquareRoot.SquareRoot(Sender:
TObject);
var
  value: Real;
  code: Integer;
  result: String;
begin
  Val(edtInputNumber.Text, value, code);
  value := Sqrt(value);
  Str(value, result);
  memOutput.Lines.Add('The square root of ' +
                      edtInputNumber.Text +
                      ' is ' + result);
end;
```

델파이 코드에 대한 사용자 인터페이스는 폼(frmSquareRoot)과 이에 대한 네 가지 컴포넌트로 구성되어 있습니다. 레이블 (lblInputNumber), 편집 상자 (edtInputNumber), 버튼 (btnComputeSqrRt), 메모 상자 (memOutput). 이 코드는 edtInputNumber에서 사용자가 입력한 수치의 제곱근을 계산해주고 결과를 memOutput에 표시합니다. 더불어 이 코드는 btnComputeSqrRt 버튼의 OnClick 이벤트와 관련되어 있습니다. 다시 말해 마우스 포인터를 btnComputeSqrRt 버튼 위에 놓고 왼쪽 마우스 버튼을 클릭하면 이 코드가 실행됩니다.

프로시저

오브젝트 파스칼 프로시저의 일반 형태는 다음과 같습니다.

```
procedure ProcedureName(param1: Type1;
                        param2: Type2; ...);
[localDeclarations;]
begin
  [statements;]
end;
```

프로시저는 필요한 인수와 함께 프로시저 이름을 기술함으로써 실행됩니다. 프로시저 호출의 일반 구문은 다음과 같습니다.

```
ProcedureName(argument1, argument2, ...);
```

예를 들면,. Adder라는 프로시저는 두 숫자를 더합니다.

```
{Add num1 and num2 and store the result in
sum}
procedure Adder(num1: Real; num2: Real;
                var sum: Real);
begin
  sum := num1 + num2;
end;
```

num1, num2 및 sum은 Adder 프로시저의 매개변수입니다. 매개변수는 sub프로그램이 호출될 때 전달되는 정보의 위치 보유자(place holder)일 뿐입니다. Adder 프로시저에 지정된 매개변수 목록에는 세 개의 구성요소가 들어있고, 이것으로 델파이 컴파일러에게 Adder 프로시저가 세 개의 실수를 전달 받아야 한다는 정보를 제공합니다. Var

키워드가 sum 매개변수 앞에 붙어 있음에 주의하십시오. 이것에 대해서는 나중에 더 자세히 설명합니다.

Adder 프로시저의 가장 간단한 시험용 프로그램은 다음과 같습니다. 사용자가 btnAdd 버튼을 클릭하면 AddNumbers 이벤트 핸들러가 호출됩니다. 이 이벤트 핸들러는 Adder 프로시저를 호출하여 edtNum1 및 edtNum2 편집 상자에 숫자를 추가한 다음 그 결과를 edtResult에 저장합니다.

```
{Add two user-entered numbers}
procedure
TfrmAdderProcedure.AddNumbers(Sender:
TObject);
var
  firstNum: Real;
  secondNum: Real;
  result: Real;
  code: Integer;

begin
  Val(edtNum1.Text, firstNum, code);
  Val(edtNum2.Text, secondNum, code);
  Adder(firstNum, secondNum, result);
  edtResult.Text := Format('%15.5f',
                           [result]);
end;
```

firstNum, secondNum 및 result 변수는 Adder 프로시저의 인수입니다. 이 변수들은 프로시저에 전달할 값을 포함합니다. 인수는 해당 매개변수와 동일한 유형의 표현식이 될 수 있다는 것에 주의하십시오.

함수

함수는 많은 인수를 가질 수 있지만 호출 루틴에 단 하나의 값을 리턴합니다. 반면 프로시저는 값을 자동으로 리턴하지 않습니다. 호출 루틴에 정확하게 하나의 값만 리턴해야 할 경우 함수를 사용하는 것이 일반적입니다. Adder 프로시저는 함수인 것이 더 낫습니다.

```
{Return the sum of num1 and num2}
function Adder(num1: Real; num2: Real): Real;
begin
  Adder := num1 + num2;
end;
```

Adder 함수의 시험용 프로그램은 다음과 같습니다.

```
{Add two user-entered numbers}
procedure TfrmAdderFunction.AddNumbers(Sender:
TObject);
var
  firstNum: Real;
  secondNum: Real;
  result: Real;
  code: Integer;
begin
  Val(edtNum1.Text, firstNum, code);
  Val(edtNum2.Text, secondNum, code);
  result := Adder(firstNum, secondNum);
  edtResult.Text := Format('%15.5f',
                           [result]);
end;
```

이 예제에서는 사용자 정의된 함수의 일반 형태를 볼 수 있습니다.

```
function FunctionName(param1: Type1; ...):
FunctionType;
[localDeclarations;]

begin
  [statements;]
  FunctionName := ReturnValue;
end;
```

특정한 데이터타입과 함께 함수가 정의되어 있는 것을 주목해 주십시오. 이 데이터타입은 함수가 호출한 루틴에 반환하는 값의 데이터타입입니다. 값을 호출하는 루틴에 반환하기 위해서는 반환값이 함수 코드 블록 내의 함수 이름에 할당되어야 하는데, 이 예제에서는 end 키워드 직전 행에서 볼 수 있습니다. 또한 반환 값은 모든 오브젝트 파스칼 함수의 암시적 매개변수인 Result 매개변수에 할당될 수 있습니다. Result 및 *FunctionName*은 같은 값을 가리키고 있습니다.

프로시저 호출처럼 함수는 필요한 인자와 함께 함수 이름을 씀으로써 간단히 불러올 수 있습니다. 그러나 프로시저 호출과는 달리 함수 호출은 하나의 값을 반환하므로 프로그램이 이 값을 처리해야 합니다(변수에 저장하고 표시하는 작업 등). 변수에 반환값 할당과 함께 함수를 호출하는 일반적인 형태는 다음과 같습니다.

```
variableName := FunctionName(argument1,
                             argument2, ...);
```

매개변수 전달

인자는 sub프로그램에 전달되는 정보의 일부입니다. 정보를 불러올 때, 매개변수는 sub프로그램에 전달되는 정보를 간직하고 있습니다. 각 인자는 대응하는 매개변수를 갖고 있습니다. 인자 및 매개변수와 관련하여 오브젝트 파스칼을 포함하여 대부분의 고급 언어에 적용되는 사항은 다음과 같습니다.

1. 넘겨주는 인자의 수는 받는 매개변수의 개수와 같아야 합니다.
2. 순서도 중요합니다. 첫번째 인자는 첫 번째 매개변수와 대응하고 두 번째 인자는 두 번째 매개변수와 순서대로 대응해야 합니다.
3. 각 인자의 데이터타입은 대응하는 매개변수의 데이터타입과 일치해야 합니다.
4. 이름은 중요하지 않습니다. 인자의 이름이 매개변수의 이름과 같을 필요는 없습니다.
5. 데이터가 전달되는 방식을 알고 있어야 합니다.

오브젝트 파스칼 언어에서는, 동일한 데이터타입을 가지고 같은 방식으로 넘겨지는 매개변수는 sub프로그램의 매개변수 목록 안에 결합될 수 있습니다. 예를 들어, Adder 함수를 아래와 같이 바꿀 수 있습니다.

```
{Return the sum of num1 and num2}
function Adder(num1, num2: Real): Real;
begin
  Adder := num1 + num2;
end;
```

Sub프로그램의 매개변수 목록 안에 있는 매개변수를 묶을 때는 정확한 순서를 유지하도록 확인해야 합니다.

VB같은 오브젝트 파스칼은 참조(by reference)나 값(by value) 두가지 방법 중 하나로 매개변수가 넘겨질 수 있습니다. 참조에 의해 매개변수를 넘기는 것은 독립변수 값 대신에 실제 독립변수의 기억장소를 sub프로그램으로 넘기는 것입니다. 이것으로 sub프로그램은 실제변수에 접근할 수 있습니다. 결과적으로 원래의 변수 값이 sub프로그램에 의해 바뀔 수 있는 것입니다. 반면 값에 의해 매개변수를 전달하는 것은 단지 변수 값만을 sub프로그램으로 전달합니다. sub프로그램은 변수사본에 접근하고 변수의 실제 값은 sub프로그램에 의해 변화될 수 없습니다. 값에 의한 전달은 오브젝트 파스칼에서 매개변수의 기본 방법입니다; 달리 지정되지 않는다면 오브젝트 파스칼은 값으로 매개변수를 전달합니다. 참조에 의한 매개변수 전달을 위해서 매개변수는 반드시 sub프로그램의 헤딩에서 var 키워드가 앞에 붙어야 합니다. 그러나 VB에서는 기본 방법에 의한 매개변수 전달은 참조에 의한 것입니다. 그리고 매개변수가 값에 의해 전달되도록 하려면 프로그래머는 반드시 sub프로그램 헤딩에 매개변수 앞에 ByVal 키워드를 놓아야 합니다.

VB에서는 매개변수를 상수값으로 전달할 수 없으나 오브젝트 파스칼에서는 가능합니다. 이런 상수를 이용한 매개변수 전달방법은 최소의 리소스를 사용합니다. 매개변수가 상수로서 전달될 때는 서브프로그램은 이 값을 변경할 수 없습니다. 만약 서브프로그램에서 변경하려 하면 컴파일러 에러가 생깁니다. 상수로 매개변수를 전달하려면 sub프로그램 헤딩에 있는 매개변수 이름 앞에 const 키워드를 둡니다.

데이터 구조

오브젝트 파스칼은 데이터 구조를 저장하기 위해 다양한 내장 데이터타입을 포함하고 있습니다. 덧붙여서 프로그래머는 자신만의 데이터타입을 정의할 수 있습니다. (예; 사용자 정의의 데이터타입을 만드는 것)

배열

배열은 동일한 데이터타입으로 이루어진, 순차적으로 색인된 요소들의 집합입니다. 오브젝트 파스칼의 경우 배열 변수는 array of 키워드를 사용하여 선언합니다. 정적 배열은 다음 구문을 사용하여 선언합니다.

```
var
arrayVariable: array[indexType1, ...,
indexTypeN] of BaseType;
```

arrayVariable란 가능한 변수명입니다. 그리고 BaseType은 배열안에 있는 각 요소의 데이터타입입니다. 각 IndexType은 배열의 분리된 색인을 대표하며 반드시 서수 데이터타입이어야 합니다. 이들은 보통 정수 부분범위입니다.

동적 배열을 선언하려면, array of 문을 색인 지정 없이 사용합니다. 예를 들어 일차원 동적 배열을 선언하려면 다음 구문을 사용합니다.

```
var
dynamicArrayVariable: array of BaseType;
```

그리고 나서 동적 배열의 메모리를 할당하고 SetLength 프로시저를 사용하여 크기를 설정합니다.

```
SetLength(dynamicArrayVariable, length);
```

한 배열의 각 요소는 고유 색인 값을 사용하여

접근할 수 있는 별개 변수로 사용됩니다. 특정 배열요소에 접근하려면 다음 중 하나를 사용합니다.

```
arrayName[indexValue1, …, indexValueN]
```

또는

```
arrayName[indexValue1] … [indexValueN]
```

예를 들어 다음 문장은 값 7을 myArray의 네 번째 요소로 지정합니다.

```
myArray[4] := 7;
```

짧은 문자열

짧은 문자열은 255문자를 넘지 않는 길이로 이루어진 문자열입니다. 짧은 문자열을 가늠해 보기 위해, 다음의 구문을 예로 들겠습니다.

```
var
stringName: String[n];
```

n은 문자로 이루어진 문자열의 길이입니다. StringName 변수가 n+1바이트의 메모리를 차지한다는 것에 주의하십시오(0~n). 1에서 n까지의 바이트가 문자열의 문자를 포함하는 곳에서 0번째 바이트는 문자열의 크기를 포함하고, n은 255와 같거나 작습니다. ShortString 데이터타입은 String[255]와 같습니다. 오브젝트 파스칼에서 모든 문자열은 단지 문자의 배열일 뿐입니다. 한 문자열 변수 안에 있는 특정 문자에 접근하려면 StringName[i]구문을 사용합니다. i는 문자열 안에 있는 문자의 색인이며(예, 문자열의 i번째 문자에 접근하려 함), 문자열 안의 첫 문자는 1의 색인 값을 가집니다. StringName[i]는 Char 데이터타입입니다.

열거형

열거형이란 프로그래머에 의해 정의된 순서를 가진 값 세트입니다. 값은 고유한 의미가 없으나 그 순서는 값이 나열된 순서를 따릅니다. 오브젝트 파스칼에서 type 키워드를 사용해 프로그래머는 사용자 정의 데이터타입을 만들 수 있습니다. 열거형을 선언하려면 다음 구문을 사용합니다.

```
type
TypeName = (value1, …, valueN);
```

TypeName과 Value1부터 ValueN까지는 유효한 인식자입니다. 다음의 예를 참고하십시오.

```
type
Days = (Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday);
```

위 코드는 한 주의 요일을 포함할 수 있는 값을 가진 Days라 명명된 열거형을 정의합니다.

Set

한 Set은 동일한 데이터타입 값의 집합으로 이루어집니다. Set의 값은 고유 순서가 없으며 한 Set 안에 한번 이상 포함되는 값은 의미가 없습니다.

Set는 set of 키워드로 정의됩니다. 구문은 다음과 같습니다.

```
type
SetName = set of BaseType;
```

SetName은 Set의 이름이며 반드시 유효 식별자여야 합니다. SetName의 가능한 값은 BaseType의 모든 부분 집합을 포함하며, 여기에는 빈 셋(오브젝트 파스칼에서 []로 표시됨)도 포함됩니다. 오브젝트 파스칼은 BaseType의 크기를 256의 가능값을 넘지 않도록 제한합니다.

Set는 보통 두개의 마침표(. .)로 표시되는 부분범위를 이용해서 정의됩니다. Set 정의는 사각 괄호 안에 쉼표로 분리된 값 리스트나 부분범위로 이루어집니다. 다음의 코드를 살펴보십시오.

```
type
  LowercaseLetters = 'a'..'z';
  LowercaseSet = set of LowercaseLetters;

var
  myLetters: LowercaseSet;

begin
  myLetters := ['a'..'c', 'm', 'n', 'x'..'z'];
    .
    .
    .
end;
```

이 코드는 LowercaseSet 데이터타입과 이 데이터타입의 MyLetters 변수를 만듭니다. 코드 블록은 소문자 a, b, c, m, n, x, y 및 z로 이루어진 Set에 myLetters를 할당합니다.

레코드

VB에서와 같이 레코드 타입 선언은 레코드의 각 항목에 대한 이름 및 데이터타입처럼 반드시 레코드 타입 이름을 지정해야 합니다. 오브젝트 파스칼 레코드 타입 선언 구문은 다음과 같습니다.

```
type
  RecordTypeName = record
                     fieldList1: DataType1;
                     fieldList2: DataType2;
                     .
                     fieldListN: DataTypeN;
                   end;
```

오브젝트 파스칼은 VB과 같이 레코드 항목에 접근하기 위해 점 분리자를 사용합니다. fieldName 레코드 항목 변수 recVarName에 접근하려면 recVarName.fieldname 형식을 사용하십시오.

포인터

포인터는 다른 변수의 기억장소를 "가리키는" 변수입니다. 따라서 포인터는 간접 변수 참조입니다. VB은 명시적인 포인터 변수를 허용하지 않으나 오브젝트 파스칼은 허용합니다. 오브젝트 파스칼에서는 포인터를 표시하거나 포인터 참조를 해제할 때 탈자기호(^)가 사용됩니다. 다음의 구문은 포인터 타입을 선언합니다.

```
type
PointerTypeName = ^DataType;
```

예를 들어 IntPointer라는 이름의 정수값을 가리키는 포인터 타입을 선언할 때는 다음의 코드를 사용합니다.

```
type
IntPointer = ^Integer;
```

IntPointer 변수 타입은 정수값을 가지는 변수의 기억 주소를 가지게 됩니다.

탈자기호가 포인터 변수 다음에 나타나면 포인터 변수를 역참조하며, 포인터가 주소에 저장한 값을 돌려줍니다. 구문 PointerVariable^은 pointerVariable를 역참조합니다. IntPointer형의 ptr변수에 있어서 예를 들면, ptr^은 정수값이나 nil을 되돌립니다. nil은 예약어이며 "nothing"을 참조하기 위한 모든 포인터 변수에 지정될 수 있는 특별 상수인데, VB에서 객체에 대한 Nothing이라는 예약어와 비슷합니다.

New와 Dispose 프로시저는 각각 포인터 변수에 연결될 새로운 메모리 공간을 만들거나 파괴합니다. New는 새로운 동적 변수를 위한 메모리를 할당하고 포인터 변수가 그것을 가리키도록 합니다. 포인터를 사용한 애플리케이션이 종료될 때, 애플리케이션은 Dispose 프로시저를 이용해서 그것에 할당된 메모리를 해제해야 합니다. 구문은 다음과 같습니다.

```
New(pointerVariable);
Dispose(pointerVariable);
```

pointerVariable 값은 Dispose 프로시저에 호출한 후에는 정의되지 않은 값입니다.

파일 입력 및 출력

VB은 파일 입력 및 출력(파일I/O)을 위해 두개의 다른 파일형, 순차 파일과 임의 액세스 파일을 제공합니다. 델파이에서 텍스트 파일은 VB의 순차파일과 동일한 역할이며, 이진 파일은 임의 액세스 파일과 같은 목적으로 사용됩니다.

텍스트 파일

오브젝트 파스칼에서 텍스트 파일에 접근하려면 텍스트 파일을 참조하는 변수를 먼저 만들어야 합니다. TextFile 데이터타입 변수 선언은 그러한 파일 참조 변수를 만듭니다. 파일 참조 변수는 파일 포인터를 포함합니다. 파일 포인터는 텍스트 에디터의 커서와 비슷합니다. 커서는 텍스트 에디터에서 파일 안에서 위치를 지시하고 파일 포인터는 열린 파일 안에서 위치를 지정합니다. 입력 파일에서 파일 포인터는 파일로부터 읽혀지는 다음 데이터 위치를 지시하고 출력 파일에서 파일 포인터는 파일로 기록되는 다음 데이터 위치를 지시합니다.

다음으로 AssignFile 프로시저를 사용해서 반드시 파일 참조 변수를 데이터 파일과 연결해야 합니다. 구문은 다음과 같습니다.

```
AssignFile(fileRef, fileName);
```

fileRef는 파일 참조 변수입니다. fileName은 유효한 윈도우 파일명이며, 파일 경로를 지정할 수도 있습니다. 이 때 경로는 파일이 저장된 드라이브와 하위 디렉터리를 지정합니다.

마지막으로, 데이터파일에 접근하려면 열어야(open) 합니다. 텍스트 파일은 입출력 모두에 대해 열릴 수 있으나 그 과정이 동시에 일어날 수는 없습니다. Reset 프로시저는 입력을 위한 문자 파일을 열거나 다시 열며, Rewrite와 Append 프로시저는 출력을 위한 문자파일을 열거나 다시 엽니다. 이러한 프로시저를 위한 구문은 다음과 같습니다.

```
Reset(fileRef);
Rewrite(fileRef);
Append(fileRef);
```

앞에서와 같이 fileRef는 파일 참조 변수입니다. Reset은 파일명이 fileRef와 연결되어 있는 존재하는 데이터 파일을 열고, 파일 포인터를 파일 시작부분에

둡니다. 파일이 이미 열려있다면 그것은 먼저 닫혔다가 열릴 것입니다. 주어진 이름의 데이터 파일이 존재하지 않으면 "file not found" 라는 런타임 오류가 발생합니다.

Rewrite프로시저는 파일명이 fileRef와 연결된 새로운 파일을 만들고 파일 포인터를 그 파일의 시작부분에 둡니다. 만약 같은 이름의 데이터파일명이 벌써 존재한다면 이것은 지워지고 새로운 빈 파일이 그 자리에 생길 것입니다. 만약 파일이 벌써 열려졌다면 이는 먼저 닫혔다가 다시 열릴 것입니다. 요약하면, Rewrite프로시저는 새로운 파일을 만들기도 하고 이미 존재하는 파일 위에 덮어쓰기도 합니다.

파일 끝에 데이터를 덧붙이려면 Append 프로시저를 사용합니다. Append는 파일명이 fileRef와 연결된 존재하는 텍스트 파일을 열고 파일 포인터를 그 파일의 끝에 둡니다. 파일이 이미 열려져 있다면 먼저 닫혔다가 다시 열릴 것입니다. 주어진 이름의 데이터 파일이 존재하지 않으면 " file not found "이라는 런타임 오류가 발생합니다. Reset 문이 입력용 문자파일을 한번 열고 나면 Read문을 사용해서 파일로부터 데이터를 읽을 수 있습니다.

```
Read(fileRef, variable);
```

이 구문은 fileRef와 연결된 입력 파일의 파일 포인터에 의해 지시된 데이터의 다음 부분을 읽고, Variable안에 이 데이터를 저장하고, 파일 포인터를 입력 파일의 다음 문자로 이동합니다. variable 데이터타입은 입력 파일로부터 읽는 데이터타입과 부합해야 합니다. 예를 들어 만약 데이터 파일이 정수로 이루어져 있다면, 데이터는 정수형 변수로 읽어야 합니다. 공백 문자(스페이스, 탭)은 문자 파일에서 수치데이터를 구분합니다.

Read문을 이용해서 여러 개의 변수를 읽을 수도 있습니다. 따라서 Read문의 일반적 구문은 이러합니다.

```
Read(fileRef, variable1 [, variable2, …]);
```

Read문이 텍스트 파일에서부터 항목 별로 데이터를 읽는데 비해, Readln문은 한 라인 안에서 데이터 항목의 특정한 숫자만을 읽습니다. 하나의 데이터 항목을 변수로 읽어들이기 위한 구문은 다음과 같습니다.

```
Readln(fileRef, variable);
```

이 문장은 fileRef와 연결된 입력 파일의 파일 포인터에 의해 지시된 데이터의 다음 부분을 읽고, 이 데이터를 variable로 저장하고, 입력파일의 다음 줄의 시작 부분에 파일 포인터를 이동합니다.

Readln문의 일반 구문은 다음과 같습니다.

```
Readln(fileRef, variable1 [, variable2, …]);
```

파일이 Rewrite나 Append문을 사용해 출력용으로 열렸을 때, 데이터는 Write 프로시저를 사용해서 파일로 기록될 수 있습니다. 다음은 일반 구문입니다.

```
Write(fileRef
      [, expression[:minWidth[:decPlaces]]]);
```

앞에서처럼 fileRef는 파일 참조 변수입니다. minWidth나 decPlaces는 정수이며 Expression은 모든 단순 혹은 문자열 데이터타입입니다. minWidth 매개변수는 생략 가능하며 출력에서 최소 문자 개수를 지정합니다. 식의 길이가 minWidth보다 짧으면 Write프로시저가 빈칸으로 출력의 왼쪽 부분을 채웁니다. 길이가 minWidth를 초과하면 모든 문자들이 출력됩니다. 생략가능한 decPlaces 매개변수는 Real 타입인 경우 소수점 뒤에 오는 숫자를 지정합니다. 쉼표로 나누면 여러 개의 출력을 단 하나의 Write문으로 출력될 수 있습니다. 예를 들어, 다음의 구문은,

```
Write(myFile, 10:5, 10.47589:8:2);
```

다음의 텍스트를 myFile과 연결된 파일에 출력합니다.

```
   10   10.48
```

Writeln 문은 Write와 같은 방식으로 동작하지만, Writeln 문은 데이터를 모두 출력한 후에 캐리지 리턴과 라인피드(<CR><LF>)를 출력하는 차이가 있습니다. 예를 들어 다음의 코드는,

```
Write(myFile, 'Hello ');
Write(myFile, 'and Good-bye');
Writeln(myFile);        {Skip to next line}
Writeln(myFile, 'Hello ');
Writeln(myFile, 'and Good-bye');
```

다음의 텍스트를 출력합니다.

```
Hello and Good-bye
Hello
and Good-bye
```

Read 및 ReadIn처럼 Write문은 결합될 수 있으나 Writeln문은 그렇지 못합니다. 따라서 위의 코드를

아래와 같이 바꿀 수 있습니다.

```
Write(myFile, 'Hello ', 'and Good-bye');
Writeln(myFile); {Skip to next line}
Writeln(myFile, 'Hello ');
Writeln(myFile, 'and Good-bye');
```

마지막으로 프로그램의 파일 작업이 끝나면, 파일을 닫아야 합니다. CloseFile문은 파일 참조 변수와 데이터파일 사이의 연결을 끝내고 리소스를 시스템으로 돌려줍니다. 출력파일에 대해서는 ClosseFile문이 파일을 닫기 전에 파일끝 문자(EOF)를 써넣습니다. CloseFile문의 구문은 아래와 같습니다.

```
CloseFile(fileRef);
```

텍스트 파일로 작업하는 데 있어서 아주 중요한 두 가지 함수가 있습니다. Eof와 Eoln함수가 그것입니다. VB에서처럼 Eof(end-of-file) 함수는 입력파일의 끝이 도달했는지를 지시하는 Boolean 값을 돌려줍니다. 파일포인터가 fileRef와 연결된 파일의 마지막 문자에 있을 때 Eof(fileRef)는 True입니다. 이 함수는 파일 끝에 도달하기 전까지 입력파일에서 나온 데이터를 읽기 위한 무한 루프 구조에서 유용합니다. Eoln(end-of-line) 함수는 파일 포인터가 현재 줄의 끝에 있는가를 지시하는 Boolean 값을 돌려줍니다. fileRef와 입력파일에 연결되어 있을 때, Eoln(fileRef)는 파일 포인터가 현재 줄의 끝에 있거나 Eof(fileRef)가 True일 때 True입니다. Eoln함수는 문자별로 입력파일을 처리하기 위한 무한루프 구조에서도 역시 유용합니다. VB에는 Eoln와 같은 함수가 없습니다.

이진 파일

텍스트 파일과는 달리 이진(binary) 파일은 어떤 순서로든지 데이터에 접근할 수 있습니다; 파일 내에서 어떤 장소에서든지 데이터를 읽거나 쓸 수 있습니다. 따라서 이진 파일은 임의 접근(Random-Access) 파일로도 알려져 있습니다. 더 나아가 이진 파일은 한번에 전체 데이터 구조에 접근할 수도 있습니다. 이런 이유로 이진 파일은 파일의 데이터구조를 알고 있을 때 파일에 포함된 정보를 저장하고 읽어들이는 더 빠르고 향상된 방법을 제공합니다.

텍스트 파일이 아스키 형식으로 저장되는 데 반해 이진 파일은 그렇지 않습니다. 이진 파일을 텍스트 파일로서 읽으면 파일 안의 모든 문자들이 의미를 가지지 않을 것이기 때문입니다. 이진 파일에 접근하려면 프로그램은 반드시 파일 안에 포함된 정확한 데이터 구조를 알아야 합니다.

오브젝트 파스칼은 typed 파일과 untyped 파일 두 가지 종류의 이진 파일을 가지고 있습니다. 우리의 논의는 두가지 중 더 일반적인 typed 파일에만 집중됩니다. typed 파일은 동일한 데이터타입 요소로 순서가 지정된 파일입니다. 여러분은 아마 배열과 typed 파일의 정의 사이에서 강한 유사성을 알아차렸을 수도 있습니다. 사실, typed 파일을 파일 형태를 가진 배열로 생각할 수도 있습니다. 곧 알게 되겠지만 데이터의 특정 부분에 접근하기 위해 배열 색인을 사용하는 대신 레코드 숫자를 사용합니다. typed 파일 데이터 타입을 정의하려면 아래에 나타난 file of 구문을 사용하면 됩니다.

```
type
FileTypeName = file of DataType;
```

FileTypeName은 임의의 유효한 식별자이며 DataType은 고정된 크기의 데이터타입입니다. DataType이 고정된 크기이기 때문에 암시적이거나 명시적인 포인터 타입은 허용되지 않습니다. 다른 말로 하면 typed 파일은 동적배열, Long 문자열, 클래스, 객체, 포인터, variant , 다른 파일 등과 이런 타입들을 포함하는 복합적인 타입을 포함할 수 없습니다.

다음은 코드 예입니다.

```
type
  StudentRec = record
                 lastName: String[30];
                 firstName: String[20];
                 ID: String[12];
                 GPA: Real;
                 crdtHrs: Real;
               end;
  StudentDB = file of StudentRec;
var
  studentFile: StudentDB;
```

이 코드는 StudentRec 레코드에 연결된 파일인 StudentDB 데이터타입을 선언하고 있습니다. studentFile은 StudentDB 타입의 typed 파일 변수이며, 연결된 파일이 특정학교 학생의 이름, 아이디, 학점 평균, 누적 신용시간을 포함하고

있습니다.

텍스트 파일에서처럼, AssignFile 프로시저는 파일 변수를 이진 파일에 연결합니다. Reset과 Rewrite 프로시저는 텍스트 파일에서와 마찬가지로 이진 파일에서 동일한 작업을 수행합니다. 기본적으로 이진 파일은 두 프로시저중 어느 것이 사용되더라도 입출력 양쪽 모두가 가능합니다. 텍스트 파일에서는 Reset 호출은 파일을 읽기 전용(입력)으로서 접근하고 Rewrite는 쓰기 전용(출력)으로 지정한다는 것을 상기하십시오. 두 프로시저는 파일 포인터를 파일의 시작부분으로 이동시킵니다. Append 프로시저는 텍스트 파일에서만 사용할 수 있으며 이진 파일과 함께 사용할 수 없습니다.

전역 변수 FileMode는 이진 파일이 Reset 프로시저로 열렸을 때 접근모드를 결정합니다. 파일 모드의 값은 읽기 전용으로는 0이고 쓰기 전용 으로는 1이며, 쓰기/읽기으로는 2입니다. 기본 파일모드는 2입니다. 파일모드에 다른 값을 지정하면 그 후부터는 모든 Reset이 그 모드로 호출됩니다.

Read 프로시저는 이진 파일의 데이터를 호환 가능한 데이터타입 변수로 읽어들입니다. 비슷하게 Write 프로시저는 변수의 내용을 호환 가능한 데이터타입의 이진 파일에 써넣습니다. 두 동작(쓰기/읽기)은 파일 포인터의 현재 위치에서 일어납니다. 실행 후에 Read와 Write는 자동적으로 파일 포인터를 이진 파일의 다음 데이터 위치로 증가시킵니다. 텍스트 파일에서처럼 여러 개의 Read 문과 여러 개의 Write문은 결합될 수 있습니다. 따라서 Read와 Write 프로시저의 일반적 구문은 아래와 같습니다.

```
Read(fileRef, dataVar [, dataVar2, …]);
Write(fileRef, dataVar [, dataVar2, …]);
```

이진 파일이 텍스트 라인으로 구성되어 있지 않기 때문에 ReadIn이나 Writeln 프로시저를 사용하려고 하면 구문 오류가 생깁니다. 간단히 말하자면, ReadIn과 Writeln은 텍스트 파일 전용입니다. 비슷하게, Eof 기능은 이진 파일에 사용될 수 있지만 Eoln은 사용할 수 없습니다.

Seek 프로시저는 이진 파일 안의 파일 포인터를 지정된 레코드나 데이터 요소로 이동합니다. 구문은 다음과 같습니다.

```
Seek(fileRef, recNum);
```

fileRef는 이진 파일 변수입니다. recNum은 long integer인데, 가지고 있는 파일안의 레코드 번호(혹은 요소 번호)를 나타내며, 첫번째 데이터 요소의 recNum은 0입니다. 한편, FileSize 함수는 특정 이진 파일안의 레코드(혹은 요소)의 갯수를 리턴합니다. fileRef 파일 변수와 대응하는 이진 파일에서 recNum의 값은 0에서 FileSize(fileRef)-1의 범위에 있습니다. 그러므로 파일 포인터를 파일의 끝으로 이동시키려면 다음과 같은 형태의 코드를 사용하면 됩니다.

```
Seek(fileRef, FileSize(fileRef));
```

위의 문에 뒤이어 바로 Write 프로시저를 실행하면 이진 파일이 하나의 데이터 요소만큼 늘어납니다. Truncate프로시저는 이진 파일에서 파일 포인터의 현재위치부터 다음의 모든 데이터 요소를 지웁니다. 그러므로 현재 파일 위치가 파일의 끝이 됩니다. 모든 파일 작업이 완료되면 이진 파일 변수와 외부 파일 사이의 연결을 종료하기 위해 CloseFile 프로시저가 사용됩니다.

예를 들어 다음의 코드 유닛은 레코드와 이진 파일 입출력을 이용해 주소록 프로그램을 구현합니다. 이 프로그램은 드라이브 C의 홈 디렉터리에 위치한 Address.dat이라는 이름의 이진 파일에 데이터를 저장합니다.

```
{---------------------------------------------
Address Book Program
---------------------------------------------}
unit Address;
interface
uses
  Windows, Messages, SysUtils, Classes,
  Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TfrmAddressBook = class(TForm)
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtAddress: TEdit;
    edtCity: TEdit;
    edtState: TEdit;
    edtZip: TEdit;
    edtPhoneNumber: TEdit;
```

```
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblAddress: TLabel;
    lblCity: TLabel;
    lblState: TLabel;
    lblZip: TLabel;
    lblPhoneNumber: TLabel;
    btnAdd: TButton;
    btnClear: TButton;
    btnRemove: TButton;
    btnFind: TButton;
    procedure AddCard(Sender: TObject);
    procedure Initialize(Sender: TObject);
    procedure ClearForm(Sender: TObject);
    procedure Terminate(Sender: TObject;
                var Action: TCloseAction);
    procedure RemoveCard(Sender: TObject);
    procedure FindCard(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  AddressCard = record
                  firstName: String[20];
                  {First Name}
                  lastName: String[20];
                  {Last Name}
                  address: String[30];
                  {Street Address}
                  city: String[20];
                  {City}
                  state: String[20];
                  {State}
                  zipCode: String[15];
                  {Zip Code}
                  phoneNumber: String[20];
                  {Telephone Number}
                end;
{
GLOBAL VARIABLES
These global variables make the coding of this
Program sufficiently easier.
}
var
  frmAddressBook: TfrmAddressBook;
  dataFile: File of AddressCard;

implementation

{$R *.DFM}

{Return the record number of the address card
that matches the first and last names. The
search is not case sensitive. If a matching
address card is not found, Find returns -1.}
function Find(first, last: String): Integer;
var
  addrCard: AddressCard;
  findCard: AddressCard;
  found: Boolean;

begin
  Reset(dataFile);
  found := False;
  findCard.firstName :=
    Trim(UpperCase(first));
  findCard.lastName := Trim(UpperCase(last));
  while not(Eof(datafile) or found) do begin
    Read(dataFile, addrCard);
    found := (UpperCase(addrCard.firstName) =
              findCard.firstName) and
             (UpperCase(addrCard.lastName) =
              findCard.lastName);
  end;
  if found then begin
    Find := FilePos(dataFile) - 1;
  end
  else begin
    Find := -1;
  end;
end;

{Add the address card to the database}
procedure TfrmAddressBook.AddCard(Sender:
TObject);

var
  addrCard: AddressCard;

begin
  with addrCard do begin
    firstName := Trim(edtFirstName.Text);
    lastName := Trim(edtLastName.Text);
    address := Trim(edtAddress.Text);
    city := Trim(edtCity.Text);
    state := Trim(edtState.Text);
    zipCode := Trim(edtZip.Text);
    phoneNumber := Trim(edtPhoneNumber.Text);
  end;
  Seek(dataFile, FileSize(dataFile));
  Write(dataFile, addrCard);
  Application.MessageBox(
    PChar('Address card added!'),
    'ADD', MB_OK);
  ClearForm(Sender);
end;

{Remove the address card from the database}
procedure TfrmAddressBook.RemoveCard(Sender:
TObject);
```

```
var
  addrCard: AddressCard;
  pos: Integer;
  recNum: Integer;
begin
  recNum := Find(edtFirstName.Text,
  edtLastName.Text);
  if (recNum >= 0) then begin
    {Display the address card}
    Seek(dataFile, recNum);
    Read(dataFile, addrCard);
    edtFirstName.Text := addrCard.firstName;
    edtLastName.Text := addrCard.lastName;
    edtAddress.Text := addrCard.address;
    edtCity.Text := addrCard.city;
    edtState.Text := addrCard.state;
    edtZip.Text := addrCard.zipCode;
    edtPhoneNumber.Text :=
      addrCard.phoneNumber;

    {Remove the address card from the
    database}
    for pos := recNum to (FileSize(dataFile) -
                          2) do begin
      Seek(dataFile, pos + 1);
      Read(dataFile, addrCard);
      Seek(dataFile, pos);
      Write(dataFile, addrCard);
    end;
    Seek(dataFile, FileSize(dataFile) - 1);
    Truncate(dataFile);

    Application.MessageBox(
      PChar('Address card removed!'),
      'REMOVE', MB_OK);
  end
  else begin
    Application.MessageBox(
      PChar('Address card NOT found!'),
      'REMOVE', MB_OK);
  end;
end;

{Find and display the address card that
matches the first and last names}
procedure TfrmAddressBook.FindCard(Sender:
TObject);

var
  addrCard: AddressCard;
  recNum: Integer;

begin
  recNum := Find(edtFirstName.Text,
              edtLastName.Text);
  if (recNum >= 0) then begin
    Seek(dataFile, recNum);
    Read(dataFile, addrCard);
    edtFirstName.Text := addrCard.firstName;
    edtLastName.Text := addrCard.lastName;
    edtAddress.Text := addrCard.address;
    edtCity.Text := addrCard.city;
    edtState.Text := addrCard.state;
    edtZip.Text := addrCard.zipCode;
    edtPhoneNumber.Text :=
      addrCard.phoneNumber;
  end
  else begin
    Application.MessageBox(
      PChar('Address card NOT found!'),
      'FIND', MB_OK);
  end;
end;

{Initialize the program by opening the Address
Book database -- File Name: c:\Address.dat}
procedure TfrmAddressBook.Initialize(Sender:
TObject);

begin
  AssignFile(dataFile, 'c:\Address.dat');
  try
    Reset(dataFile);
  except
    Rewrite(dataFile);
  end;
end;

{Clear the edit boxes on the form}
procedure TfrmAddressBook.ClearForm(Sender:
TObject);

begin
  edtFirstName.Clear;
  edtLastName.Clear;
  edtAddress.Clear;
  edtCity.Clear;
  edtState.Clear;
  edtZip.Clear;
  edtPhoneNumber.Clear;
end;

{Close the database file and terminate the
program}
procedure TfrmAddressBook.Terminate(Sender:
TObject;
  var Action: TCloseAction);
begin
  CloseFile(dataFile);
end;
end.
```

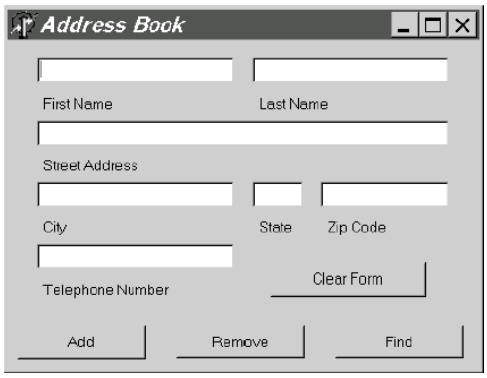

그림 12. *주소록 프로그램의 사용자 인터페이스*

마지막으로 알아두어야 할 것은, 델파이에는 객체지향 파일 입출력을 위한 TFileStream 클래스가 있다는 것입니다. 이것으로 프로그래머는 파일 입출력을 위해 이식 가능한 고수준 방법을 사용할 수 있습니다. 다음 부분에서는 델파이의 객체지향 프로그래밍을 소개합니다.

객체지향 프로그래밍

앞서 논의한 것처럼 델파이는 완전한 객체모델을 가지고 있습니다. 모든 컴포넌트와 컨트롤은 베이스 객체 클래스인 TObject가 그 조상입니다. 따라서 모든 컴포넌트와 컨트롤은 객체지향 프로그래밍(OOP)을 통해 완전히 확장 가능합니다. 그러나 VB은 완벽한 객체모델을 가지지 않습니다. VB는 델파이처럼 진정한 객체 상속과 다형성을 제공하지 않습니다.

OOP의 장점을 격찬하고 오브젝트 파스칼에서 그 구문을 자세히 살피는 것보다는, 한가지 예를 보기로 하겠습니다. 다음의 델파이 유닛 파일은 Employee와 Supervisor라는 두 클래스를 정의합니다. Supervisor 클래스는 Employee클래스의 자손입니다. 이 코드의 실행된 예는 그림 13에 나타나 있습니다.

```
unit OOPEx;

interface

uses
  Windows, Messages, SysUtils, Classes,
  Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TfrmOOPEx = class(TForm)
    btnTest: TButton;
    memOutput: TMemo;
    procedure DoTest(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

type
  NameStr = String[25];
  IDStr = String[10];
  DivType = (Operations, Production,
             Maintenance);
  Employee = class
    lastName: NameStr;
    firstName: NameStr;
    division: DivType;
    procedure SetAll(lname, fname:
                NameStr; dv: DivType);
    procedure SetLastName(lname: NameStr);
    function GetLastName: NameStr;
    procedure SetFirstName(fname: NameStr);
    function GetFirstName: NameStr;
    procedure SetDivision(dv: DivType);
    function GetDivision: DivType;
  end;
  Supervisor = class(Employee)
    managerID: IDStr;
    procedure SetAll(lname, fname: NameStr;
                     dv: DivType;
                     id: IDStr);
    procedure SetID(id: IDStr);
    function GetID: IDStr;
  end;

var
  frmOOPEx: TfrmOOPEx;
  implementation

{$R *.DFM}

procedure Employee.SetLastName(lname:
                                NameStr);
begin
  lastName := lname;
end;

function Employee.GetLastName: NameStr;
begin
  GetLastName := lastName;
end;

procedure Employee.SetFirstName(fname:
                                 NameStr);
begin
  firstName := fname;
```

```
end;

function Employee.GetFirstName: NameStr;
begin
  GetFirstName := firstName;
end;

procedure Employee.SetDivision(dv: DivType);
begin
  division := dv;
end;

function Employee.GetDivision: DivType;
begin
  GetDivision := division;
end;

procedure Employee.SetAll(lname, fname:
                          NameStr;
                          dv: DivType);
begin
  Self.SetLastName(lname);
  Self.SetFirstName(fname);
  Self.SetDivision(dv);
end;

procedure Supervisor.SetID(id: IDStr);
begin
  managerID := id;
end;

function Supervisor.GetID: IDStr;
begin
  GetID := managerID;
end;

procedure Supervisor.SetAll(lname, fname:
                            NameStr; dv:
                            DivType;
                            id: IDStr);
begin
  Self.SetLastName(lname);
  Self.SetFirstName(fname);
  Self.SetDivision(dv);
  Self.SetID(id);
end;

procedure TfrmOOPEx.DoTest(Sender:
                           TObject);

var
  emp: Employee;
  mgr: Supervisor;

begin
  emp := Employee.Create;
  mgr := Supervisor.Create;
  emp.SetAll('Thompson', 'James',
    Operations);
  mgr.SetAll('Stewart', 'Linda',
    Operations, '003685');
  memOutput.Clear;
  memOutput.Lines.Add('OPERATIONS DIVISION');
  memOutput.Lines.Add('Supervisor: ' +
                      mgr.GetLastName + ', '
                      + mgr.GetFirstName);
  memOutput.Lines.Add('Employee: ' +
                      emp.GetLastName + ', '
                      + emp.GetFirstName);
  emp.Free;
  mgr.Free;
end;

end.
```

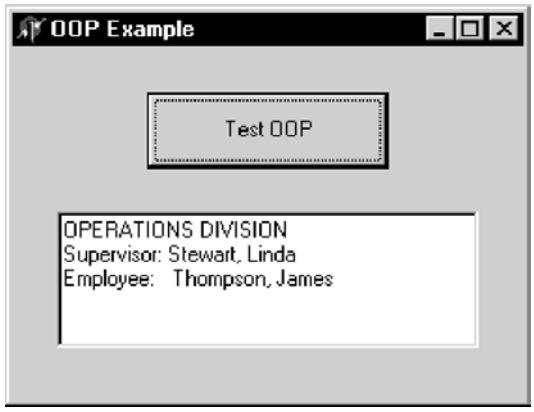

그림 13. *객체지향 프로그래밍 예*

리눅스 호환성 문제

윈도우에서 개발하는 데 있어 ActiveX나 ODBC 같은 리눅스로 이식될 수 없는 기술에 의존하는 경우가 많습니다. ActiveX 컨트롤을 사용한 부분은 리눅스로 이식되지 않지만 델파이의 다른 측면들은 이식이 가능합니다. 새로운 컴포넌트를 만드는 것을 포함하여, 네이티브 컴포넌트에 관련한 어떠한 작업이든 Kylix에서도 가능합니다. ODBC를 대신하기 위해 MySQL, InterBase 포함한 많은 유명한 데이터베이스로의 드라이버들이 준비되어 있습니다.

여러분의 애플리케이션에서 ADO나 COM/DCOM을 사용하고 있다면, 그런 기술들이 리눅스에서 사용 가능한지에 대해 알고 싶을 것입니다. COM/DCOM을 제외하면, 현재로서는 마이크로소프트가 리눅스를 위해 이러한 기술을 발표하려는 어떤 노력도 알려진 바가 없습니다.

컴포넌트 개발자에게는 델파이의 윈도우 및 리눅스 버전 사이에는 실질적인 차이점이 있지만, 애플리케이션 개발자에게는 대단치 않습니다. 몇가지 중요한 차이점은 아래와 같습니다.

- 대부분의 컴포넌트와 속성 이름은 같습니다. 그러나 새로운 속성이나 사라진 속성도 있습니다.
- 리눅스 파일 시스템은 DOS/윈도우와 다릅니다. 드라이브 접근 방법이 다르며, 파일명도 대소문자를 구분합니다.
- 앞서 언급한대로 ActiveX는 리눅스에서 지원되지 않습니다. 덧붙여 OLE도 불가능합니다. 따라서 ComObj, ComServ, ActiveX, Windows 유닛은 Kylix에서 존재하지 않습니다.

내장 디버거

VB처럼 델파이에도 프로그래머가 코드를 추적하고 오류 위치를 알아내는 것을 돕기 위한 내장 디버거가 포함되어 있습니다. 델파이 디버거를 사용하려면 통합 디버깅 옵션이 사용가능하도록 되어있어야 합니다. 통합 디버깅을 사용하려면 메뉴에서 Tools|Debugger Options…을 선택한 후 나타난 Debugger Options 윈도우의 왼쪽 아래의 Integrated Debugging 체크박스를 체크하고 확인 버튼을 누릅니다.

디버거 명령은 실행 메뉴와 디버그 툴바를 통해 사용할 수 있습니다. 덧붙여 디버거 명령은 코드 에디터 윈도우를 통해 빠르게 이용할 수 있습니다. 코드 에디터안에 아무 곳이나 오른쪽 클릭을 하여 팝업 메뉴를 엽니다. 이 팝업 메뉴의 디버그 항목에는 실행 가능한 디버거 명령들이 나열됩니다.

델파이 디버거는 오류 위치를 알아내는 반자동 방법을 제공합니다. 프로그래머는 이것으로 특정 변수나 식을 프로그램 코드를 수정하지 않고 감시할 수 있습니다. 덧붙여 디버거는 설정된 중단점에서 프로그램 실행을 중지시키거나 단계별로 프로그램 코드를 실행할 수 있습니다. 디버거는 디자인타임 유틸리티라는 것에 유의하십시오. 모든 디버거 명령들은 런타임에 델파이 IDE 바깥의 실행모듈에서 사용될 수 없습니다.

소스 중단점은 프로그래머가 코딩한 코드 라인에서 토글됩니다. 중단점은 실행 가능한 코드 라인에서만 지정될 수 있습니다. 빈 라인, 선언문, 주석문은 중단점을 가질 수 없습니다. 중단점을 만나면, 프로그래머가 Run 메뉴에서 Run을 선택하거나 F9키를 누르거나 디버그 툴바의 Run 버튼을 왼쪽 클릭하기 전까지 프로그램 실행이 일시적으로 멈춰집니다. 델파이 디버거는 또한 주소, 데이터, module load 중단점을 허용합니다.

프로그램이 멈춰진 동안 프로그래머는 즉시 Evaluate/Modify 윈도우 안에서 식을 계산하거나 수정할 수 있습니다(그림 14). 하나의 식은 그림 15의 Inspector 윈도우 안에서 나타나거나 변경할 수 있습니다. Inspector 윈도우를 이용해 프로그래머는 복잡한 데이터 구조를 가진 객체를 더욱 알기 쉽게 볼 수 있습니다.

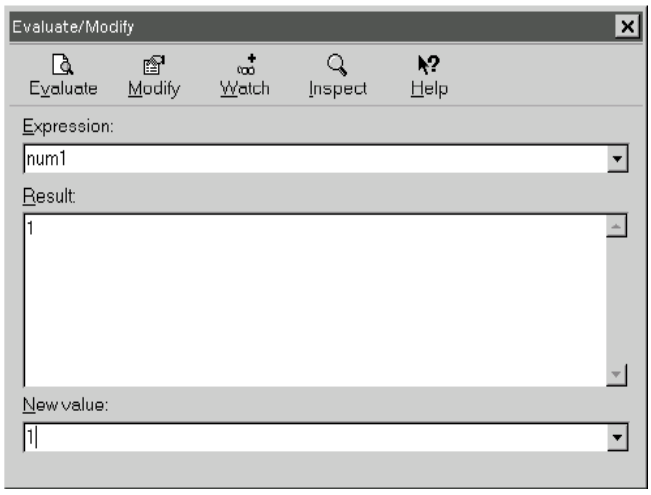

그림 14. *Evaluate/Modify 윈도우*

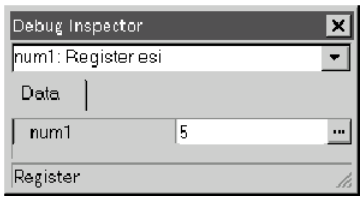

그림 15. *Inspector 윈도우*

델파이 디버거는 두 가지 단계별(step) 디버깅 방식을 가지고 있습니다. Trace Into는 한번에 한 구문의 코드를 실행합니다. 만약 구문이 서브 프로그램으로의 호출이면 다음에 표시되는 구문은 서브 프로그램의 첫 번째 문입니다. Step Over는 서브 프로그램 호출을 한번의 스텝으로 실행한 후 서브 프로그램 호출의

다음 문으로 넘어갑니다. 따라서 Step Over는 언제나 현재 서브 프로그램의 다음 문으로 이동합니다.

watch expression은 사용자 정의 식으로서, 프로그래머가 동작을 관찰할 수 있습니다. watch expression은 Watch 윈도우(그림 16)에 나타나는데 그 값은 중지된 상태에서 자동으로 갱신됩니다. 더 나아가 Local Variables 윈도우는 그림 17처럼 자동으로 현재 서브 프로그램 안의 모든 선언된 변수(모든 지역 변수)의 값을 표시합니다.

그림 16. *Watch 윈도우*

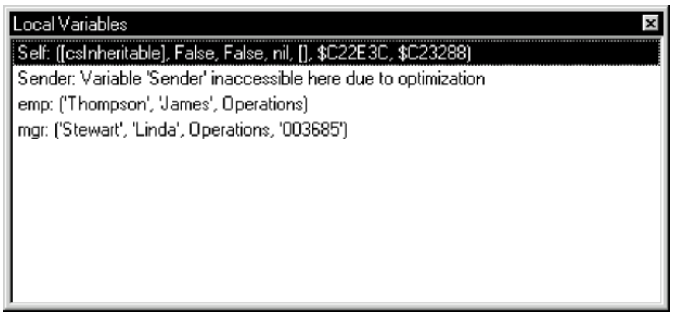

그림 17. *Local Variables 윈도우*

애플리케이션 배포

이 섹션에서는 델파이 프로젝트의 컴파일 및 최종 애플리케이션 배포 방법을 논의합니다.

실행 파일 구축

C/C++ 등 다른 고급 언어와 비슷하게, 델파이를 이용해 실행가능파일을 생성하는 것은 두 단계의 과정을 거칩니다. 첫번째 단계는 프로젝트를 컴파일하는 것입니다. 이 단계는 프로젝트의 각 유닛의 구문을 검사하고 유닛별로 오브젝트 파일을 만들어냅니다. 유닛 오브젝트 파일은 DCU 확장자를 가지는데, 이는 Delphi Compiled Unit을 뜻합니다.

다음으로, 이러한 오브젝트 파일들은 링크되어야 합니다. 링커는 프로젝트의 각 유닛 오브젝트 파일을 가져와서 서로 결합시켜 하나의 실행 파일을 만들어냅니다.

델파이의 프로젝트 메뉴는 실행 파일을 만들기 위한 명령을 가지고 있습니다. Compile 메뉴 항목은 프로젝트의 각 유닛를 컴파일하지만 그 유닛들을 링크시키지는 않습니다. Build 메뉴 항목을 선택하면 유닛들을 컴파일하고 링크하여 최종 실행파일을 만듭니다.

애플리케이션 배포

프로젝트가 데이터베이스, 써드 파티 DDL이나 ActiveX 컨트롤을 사용하지 않는다면 생성되는 실행가능 파일은 독립적으로 실행 가능합니다(Stand Alone). 여러분은 단순히 이 실행파일을 플로피 디스크나 다른 미디어로 복사한 다음, 대상 운영 체제를 사용하는 시스템에서 실행시키기만 하면 됩니다. 프로젝트에 사용되는 모든 델파이 컨트롤들은 결과물인 실행파일로 컴파일되어 포함됩니다. 애플리케이션 안에 포함될 필요가 있는 특별한 다른 것은 없습니다.

만약 VB에서처럼 프로젝트 내에서 ActiveX 컨트롤을 사용한다면 어플리케이션의 배포는 더 복잡해집니다. 반드시 프로젝트에서 사용된 각각의 ActiveX 컨트롤에 대한 OCX파일을 배포해야 합니다. 써드 파티 ActiveX 컨트롤에 대한 첨부 문서를 꼭 읽어보십시오; 어떤 것은 추가적으로 DLL을 요구합니다.

윈도우에서 애플리케이션을 멋있게 배포하는 한가지 방법은 셋업 프로그램을 만드는 것입니다. 셋업 프로그램을 만드는 경우 애플리케이션에서 사용되는 모든 ActiveX를 등록하는 것을 잊지 마십시오. 이것은 RegSvr32.exe 유틸리티를 실행키면 됩니다.

셋업 프로그램을 만드는 인기있는 방법은 상업적으로 판매되고 있는 많은 설치 패키지 중 하나를 이용하는 것입니다. VB은 이런 목적을 위한 패키지와 배치 마법사를 포함하고 있습니다. 비슷하게 델파이에는 InstallShield Express가 포함되어 있습니다. 이것은 애플리케이션 배포를 위한 셋업 프로그램을 만들 수 있게 해주는 메뉴 기반 설치 유틸리티입니다.

추가 레퍼런스

본 저자는 VB 서적의 공동저자진 중 리더이며, 출판 예정인 델파이 서적을 저술한 바 있습니다. 두 책 모두 프로그래밍 과정의 입문용으로서 기획되었으며, 고급 프로그래머들에게도 귀중한 참고 자료임이 증명되었습니다. 두 권 모두 Addison Wesley Longman (AWL)사에서 출판되었습니다. 책에 대한 더 자세한 정보를 찾아보려면 AWL사의 웹사이트를 참조하십시오. (http://www.awl.com/cs)

Computer Programming Fundamentals with Applications in Visual Basic 6.0, by Mitchell C. Kerman and Ronald L. Brown

An Introduction to Computer Science: Programming and Problem Solving with Delphi (가제), by Mitchell C. Kerman

덧붙여, 델파이에 관해 세미나, 강의를 통한 교육 등을 제공하는 몇몇 회사들이 있습니다. RayTech Software 주식회사(VB/델파이 교육기관으로 공인되어 있습니다)는 교육 과정과 컨설팅 서비스를 제공합니다. RayTech사는 최근 VB 개발자를 위한 델파이 일일 세미나를 개설했습니다. RayTech 사의 수업과정은 표준 교재를 사용하는 공인 과정에서부터 개개인의 특별한 필요에 맞추어진 주문형 수업 과정까지 포함하고 있습니다. 더 자세한 내용을 찾아보려면 RayTech사의 웹사이트 http://www.raytech-software.com를 참조하십시오.

서적이나 교육과정이 충분치 않은 경우, 필자는 제작사로부터 나온 문서들을 추천합니다. 필자는 마이크로소프트와 볼랜드의 개발자 웹사이트, 온라인 헬프, 그리고 매뉴얼 등을 이용합니다. 다른 참고서가 필요하다면 제작사가 직접 출판한 책을 선택하십시오. 마이크로소프트 제품에 대해서는 Microsoft Press, 볼랜드의 제품에 대해서는 Macmillan Publishing을 통해 Borland Press에서 책들을 내고 있습니다.

결론

델파이는 강력한 오브젝트 파스칼 언어를 이용하는 대단히 뛰어난 RAD 개발툴입니다. 델파이는 다양하고 쉽게 사용할 수 있는 컴포넌트와 툴들, 계층적인 컴포넌트 디자인, 진정한 객체 지향, 그리고 직관적인 IDE를 제공합니다. 게다가 델파이는 윈도우와 리눅스 양쪽 모두에서 사용할 수 있습니다.

단 하나의 문서로 델파이의 장점을 모두 설명하지는 못하겠지만, 본 저자는 여러분이 VB에서 델파이로 마이그레이션하여 숙련된 델파이 개발자가 되는 것이 어렵지 않다는 것에 관심을 가지기를 바랍니다.

윈도우/리눅스용 델파이와 함께 멋진 미래를 즐겨보십시오. 볼랜드사는 윈도우/리눅스 개발을 위해 델파이라는 요술램프 속의 지니를 불러냈습니다. 이제 여러분에게는 무한한 소원을 말할 수 있는 기회가 주어졌습니다.

Borland Software Corporation
100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com

Borland®