

## 이것이 델파이 4 ! (This Is Delphi 4 !)

델파이 4 는 현재 사용할 수 있는 가장 유연하면서도 강력한 개발 도구이다. 델파이 4 는 비주얼 인터페이스 디자인과 강력한 객체지향 언어로서의 특징을 가지고 있는 오브젝트 파스칼 언어를 통합하고 있다. 개발자는 이를 이용하여 빠르면서도 직관적이고, 견고한 Win32 어플리케이션을 쉽게 개발할 수 있다.

이번 장에서는 델파이 4 에서 새롭게 선보이는 여러 가지 기능과 특징 들을 소개한다. 처음으로 델파이를 접하는 사람들에게는 다소 어려운 내용이 될 수도 있으나, 대부분의 내용이 나중에 다시 자세히 언급될 것이므로 그냥 한 번 읽어두는 것도 좋을 것이다.

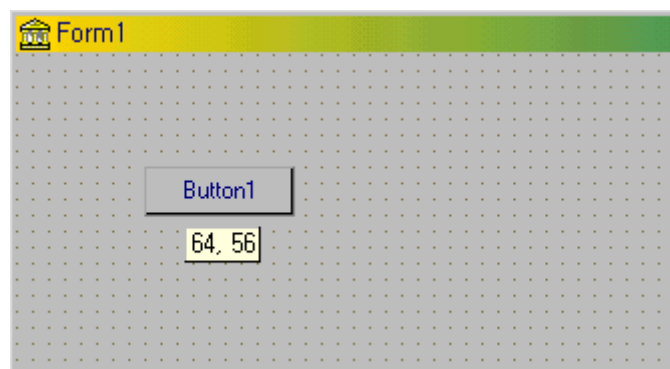
흔히 차를 새로 홍보할 때 보면, 그 기능은 잘 몰라도 각종 기술 이름을 들먹여가며 ‘이 차에는 이런, 저런 기능을 추가했습니다. ‘ 라는 문구를 많이 보게 되는데, 이번 장의 내용이 그렇다고 보면 된다.

### 몰라보게 좋아진 IDE

델파이 4 에서 가장 향상된 점을 들라고 하면, 그동안 항상 비슷하게 유지되던 IDE 의 모습이 상당히 많이 바뀐 점이다. 과거로부터 바라왔던 부분들이 반영되어 편리한 개발 환경이 되었다. 보다 자세한 사항은 다음 장에서 다루게 된다.

- Form 디자이너에 기본적인 마우스 좌표지원

델파이 4 의 폼 디자이너에서는 컨트롤의 위치를 표시하는 힌트 윈도우를 이용하여 좌표를 표시해주는 기능이 추가되었다.

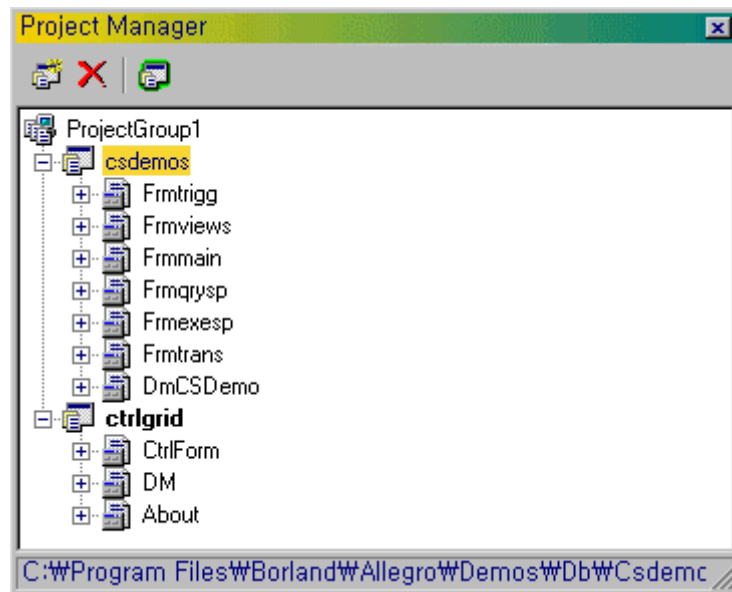


- 윈도우 98 의 듀얼 모니터 기능 지원

윈도우 98 의 듀얼 모니터 기능을 이용하여 코드 에디터를 여러 개의 모니터로 나누어 사용하거나, 실행 파일을 따로 보는 것 등이 가능해졌다.

- 프로젝트 관리자 (Project Manager)

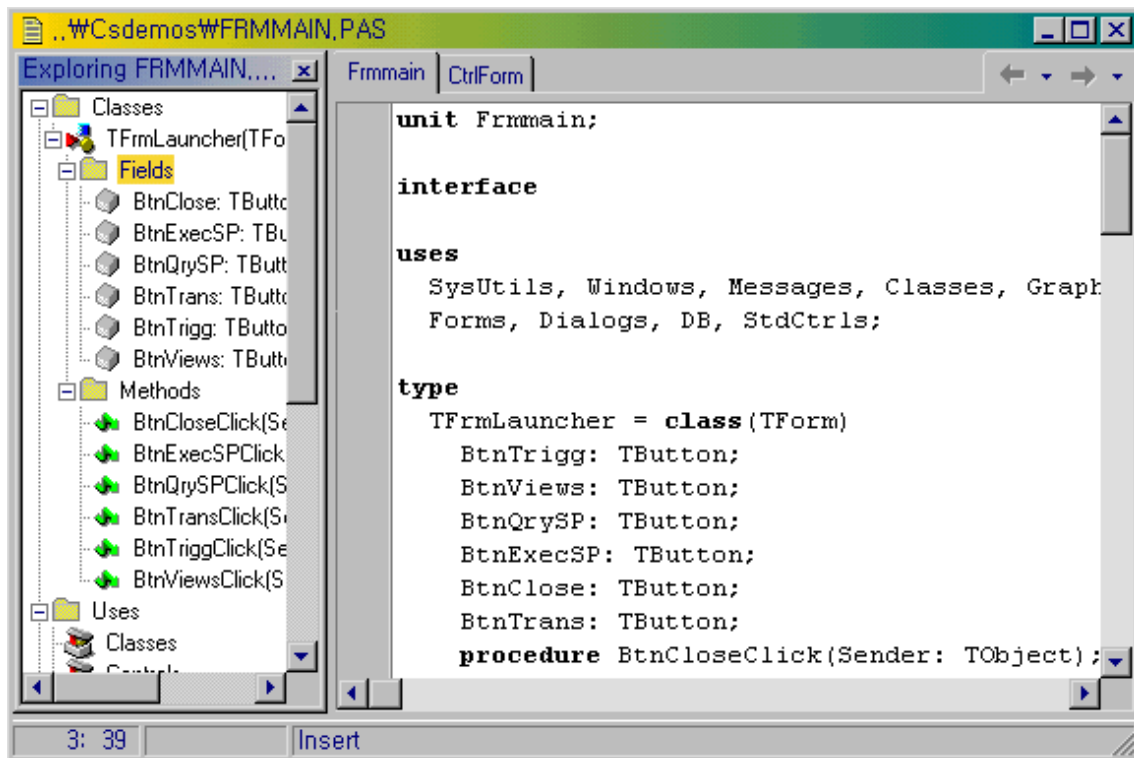
과거의 단일 프로젝트 그룹 방식에서 여러 개의 프로젝트를 동시에 관리할 수 있는 프로젝트 관리자를 지원한다.



이를 이용해서 멀티-tiered 어플리케이션의 각각의 어플리케이션 또는 DLL 과 이를 사용하는 어플리케이션과 같이 서로 관계 있는 프로젝트 들을 동시에 관리하며 개발이 가능하다

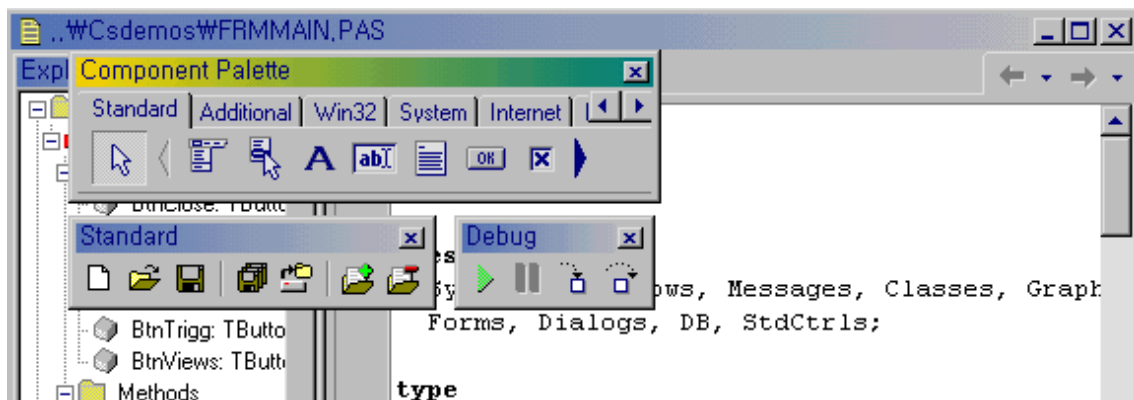
- 클래스 완료 (class completion), 모듈 탐색 (module navigation), 코드 브라우저 (code browser)를 포함한 모듈 탐색기 (Module Explorer)

새로운 모듈 탐색기는 클래스를 만드는 여러 과정을 자동화하여 클래스의 생성 과정을 쉽게 만들었다. Interface 섹션에서 메소드의 prototype 을 기록하고, 모듈 탐색기에게 skeleton code 를 작성하도록 하면, implementation 섹션에 기본 코드가 생성된다. 유닛 파일에서 interface, implementation 섹션 사이에서 객체를 탐색할 수 있는 기능도 포함되어 있다.



- 도킹 툴 윈도우 (Dockable tool windows)

IDE 의 각 윈도우가 오피스 97 과 같이 도킹이 가능한 형태로 바뀌었다. 각각의 툴 윈도우를 drag-and-drop 만으로 원하는 위치에 둘 수 있다. 모듈 탐색기와 프로젝트 매니저 역시 도킹이 가능하다.



## 오브젝트 파스칼의 확장

델파이 4 는 오브젝트 파스칼에 여러가지 언어적인 확장을 가져왔다. 여기에는 다음과 같은 것들이 있다.

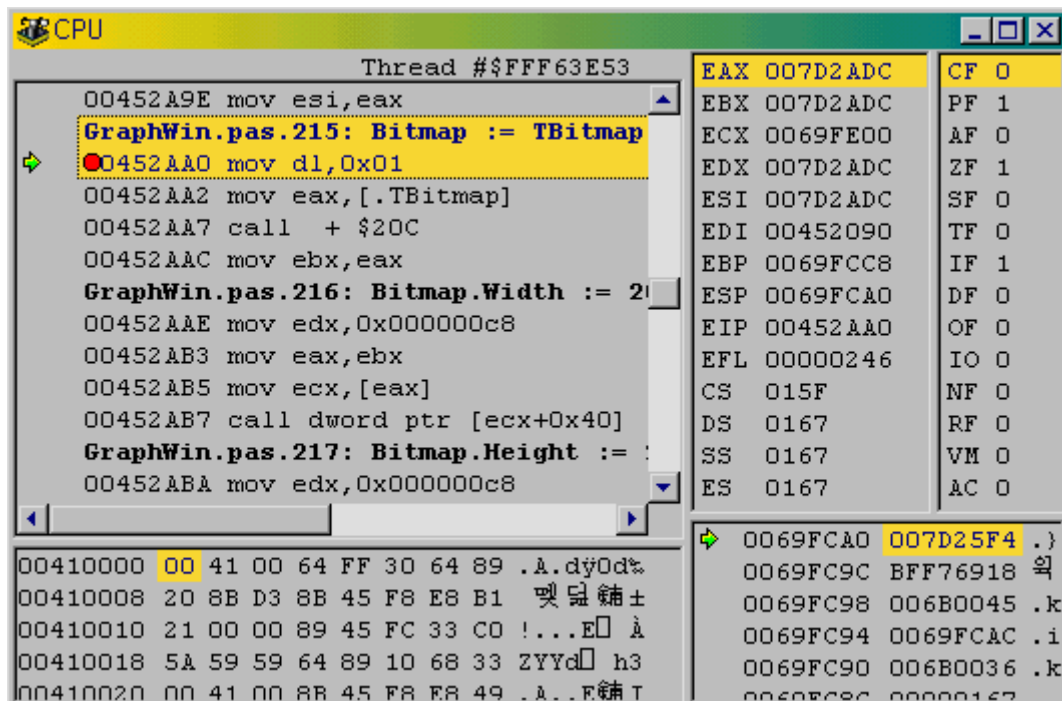
- 동적 배열
- 메소드 오버로딩
- 디폴트 파라미터
- 64 비트 정수형
- 32 비트 unsigned 정수형
- 실수형의 변화
- 인터페이스 구현 방식에 대리자(delegation) 허용

이들에 대한 자세한 내용은 4, 5, 7 장의 내용을 참고하기 바란다.

## 디버깅 기능의 강화

텔파이 4 는 C++ 빌더 3 에서 볼 수 있었던 여러 가지 디버깅 기능이 새로 추가되었다. 새로운 인스펙터와 CPU 윈도우, 모듈 윈도우와 이벤트 로그를 이용하여 보다 편리한 디버깅 환경을 제공한다. 또한, 원격 어플리케이션에 대한 디버깅과 다중 프로세스 디버깅을 지원하므로 멀티-tiered 어플리케이션의 개발이 용이하다.

추가된 CPU 윈도우 화면은 다음과 같다.



향상된 VCL



델파이 4에서는 VCL의 구조가 다소 향상되었다. 변경된 부분만 나열하면 다음과 같은 것들이 있다.

- 액션 리스트(Action Lists)는 메뉴나 각종 버튼에 의한 사용자 명령을 중앙에서 집중하여 관리할 수 있도록 해준다.
- 윈도우 NT에 대한 프로그램을 개발할 때 서비스 프로그램으로 활용할 수 있도록 TServiceApplication, TService 클래스를 제공한다. 또한, New items 대화상자에는 여기에 대한 위저드를 2가지 제공하고 있다.
- TControl과 TWinControl이 도킹을 지원하도록 변경되었으며, TControl에는 윈도우의 크기를 변경하지 못하게 설정할 수 있는 기능이 추가되었다.
- Ini 파일에 대한 지원도 확대되었다. 레지스트리와 ini 파일을 동시에 지원하는 TRegistryIniFile 클래스, 그리고 Ini 파일의 변화를 메모리에 캐쉬했다가 저장하는 TMemIniFile 클래스가 추가되었다.
- TParams 클래스의 유닛 위치가 dbtables에서 db로 바뀌었는데, 이는 TClientDataSet 클래스에 파라미터를 지원하게 하기 위한 것이다. 그리고, 스크롤 바가 좌측에 있다거나 텍스트를 우측에서 좌측으로 표현하는 등의 세계화에 걸맞는 프로퍼티가 추가되었다.
- 기본적인 TObject 클래스에 BeforeDestruction, AfterConstruction이라는 2개의 새로운 protected 메소드가 추가되었다. BeforeDestruction은 객체가 destructor를 호출하기 직전에 호출되며, AfterConstruction은 객체가 constructor를 호출한 직후에 호출된다. TObject를 상속한 클래스들은 이들 메소드를 override하여 constructor나 destructor에서 일어나면 안되는 작업이 일어나지 않게 한다. 예를 들어, 폼의 경우 이들 메소드를 오버라이드하여 OnCreate, OnDestroy 이벤트를 발생시키는데, 이것이 중요한 이유는 컴포넌트가 델파이 뿐만 아니라 C++ 빌더에서도 사용될 수도 있기 때문이다. 즉, 오브젝트 파스칼과 C++의 constructor, destructor의 행동에 차이가 있기 때문에 이 문제를 해결하기 위한 것이다.
- 윈도우 98을 지원하는 새로운 컨트롤이 추가되었다. 여기에는 TControlBar, TPageScroller, TComboBoxEx, TMonthCalendar, TFlatScrollbars 등의 컴포넌트가 있다.

## 클라이언트 데이터 세트의 향상

델파이 4에서는 BDE를 사용하지 않고도 여러가지로 활용할 수 있는 클라이언트 데이터 세트 컴포넌트가 보다 강화되어 그 효용성이 더욱 높아졌다.

TClientDataSet 컴포넌트는 BDE를 사용하지 않고, DBClient.DLL 파일만을 사용해서 데이터베이스의 기능을 활용할 수 있게 해준다. 클라이언트 데이터 세트를 사용할 때에는 데

데이터베이스 연결이 필요하지 않으므로 TDatabase 컴포넌트도 사용하지 않는다.

클라이언트 데이터 세트는 데이터에 접근, 편집, 탐색, 데이터 제한과 필터링 기능까지 제공하고 있다.

## 멀티-tiered 어플리케이션 지원의 강화

델파이 4에서는 멀티-tiered 어플리케이션에 대한 지원이 더욱 강력해졌다. 여기에는 다음과 같은 것들이 있다.

- Refresh/resync 지원
- 데이터 패킷에 대한 지원
- 중첩된 테이블을 이용한 마스터/디테일 관계 지원
- 브로커 커넥션의 후킹
- 클라이언트 데이터 세트에서 파라미터를 어플리케이션 서버에 넘기거나 사용자 정의 정보를 데이터 패킷에 저장할 수 있다.
- 쉬운 서버 인터페이스 호출
- TDataSetProvider 라는 새로운 클래스를 통한 데이터 세트 지원
- 어플리케이션 서버에 접속하는데 필요한 다양한 연결 컴포넌트
- 서버 소켓의 콜백 지원과 NT 서비스 지원
- Midas 의 OLE Server 지원

## 데이터베이스 기능 향상

테이블 컴포넌트가 폼 디자이너에서 테이블을 생성, 이름 변경, 삭제할 수 있도록 향상되었다. 테이블 컴포넌트를 디자이너에서 선택하고, 오른쪽 버튼을 클릭한다. 여기에서 적당한 메뉴를 선택하면 된다.

또한 새로운 BDE 엔진은 액세스 97 과 오라클 8, 인포믹스 9.0, 인터페이스 5.1 을 지원한다. 또한, 델파이 4에서는 SQL 을 확장하여 오라클 8 의 새로운 확장성을 지원하게 되었다. 즉, ADTs(Abstract Data Types), 배열, 참조, 중첩된 테이블 등을 지원한다. 이를 위해 새로운 TField 데이터 형인 TADTField, TReferenceField, TArrayField 를 지원한다. 또한, ADTs 와 중첩된 테이블을 보여주기 위해 Grid 컴포넌트를 향상시켰다. 또한, 기존의 Visual Query builder 에 새로운 쿼리 기능을 추가한 SQL builder 를 제공한다. 그리고, 객체지향 데이터베이스 모델을 지원한다.

## 마이크로소프트 트랜잭션 서버(MTS) 지원

MTS 는 DCOM 어플리케이션의 트랜잭션 서비스와 보안, 리소스 관리를 하는 견고한 런타임 환경이다. 델파이 4 는 MTS 자동화 위저드를 제공하여 MTS 자동화 객체를 생성할 수 있으며, 이를 이용해 MTS 환경의 여러가지 잇점을 사용할 수 있다. 또한 MTS 서버 객체 위저드를 이용하면 서버 객체를 쉽게 생성할 수 있다. MTS 는 COM 클라이언트와 서버를 생성하고, 이를 구현하는데 편리한 많은 서비스를 제공한다. MTS 컴포넌트는 많은 하위 레벨 서비스를 제공하는데, 여기에는 다음과 같은 것들이 있다.

- 동시에 많은 사용자들이 사용할 수 있는 서버 어플리케이션을 만들기 위해 프로세스나 쓰레드, 데이터베이스 연결 등의 시스템 리소스를 관리한다.
- 트랜잭션을 자동으로 시작하고, 이를 관리한다.
- 필요할 때 서버 컴포넌트를 생성, 실행, 삭제한다.
- 보안을 제공하기 때문에, 인증된 사용자만 어플리케이션에 접근할 수 있다.

MTS 는 어플리케이션의 비즈니스 로직을 MTS 자동화 객체나, MTS 원격 데이터 모듈에 구현한다. 컴포넌트를 DLL 에 구현하면, DLL 은 MTS 런타임 환경으로 설치된다. 델파이에서 MTS 클라이언트는 독립적인 어플리케이션으로 사용되거나 또는 액티브 폼일 수 있다. COM 클라이언트는 어느 것이나 MTS 런타임 환경에서 동작할 수 있다.

## 액티브 X/COM 지원의 강화와 CORBA 의 지원

델파이 4 는 기존의 COM 에 대한 지원을 한층 강화하였다. COM 에 대한 강화된 지원 내용을 열거하면 다음과 같은 것들이 있다.

- COM 객체를 생성해주는 COM 객체 위저드가 추가 되었다.
- 타입 라이브러리 에디터를 이용해서 IDL 에 호환되는 타입 라이브러리 소스를 생성할 수도 있고, 델파이 3 에서 문제가 되던 것들을 해결하였다. TypeLib2 규격을 지원한다.
- VB 데이터 인터페이스를 지원한다.

자세한 내용은 제 5 부의 내용들을 참고하기 바란다.

델파이 4 에서는 COM 이외에 산업 표준으로 자리잡고 있는 CORBA 를 지원하게 되었다. CORBA 에 대한 지원 사항을 나열하면 다음과 같은 것들이 있다.

- CORBA 에서 JAVA 의 IDL 을 지원
- CORBA 데이터 모듈 위저드 제공
- CORBA 커넥션 컴포넌트의 제공

- One-Step CORBA 지원

## 새로운 인터넷 컴포넌트의 지원

텔파이 4 에서는 인터넷 컴포넌트가 매우 많이 늘어났다. 우선, 텔파이 3 에서는 ocx 파일로 제공되던 인터넷 컴포넌트 들이, Net Master 에서 제공하는 native 컴포넌트로 제공된다. Net Master 의 인터넷 컴포넌트에는 다음과 같은 것들이 있다.

- TNMDayTime, TNMTime, TNMEcho, TNMFinger, TNMMsg, TNMMsgServ
- TNMFTP, TNMHTTP, TNMNNTP, TNMSMTP, TNMPOP3, TNMUDP
- TNMUUProcessor, TNMStrm, TNMStrmServ, TNMPowersock
- TNMGeneralServer, TNMURL

## 그 밖의 변화

그 밖에도 텔파이 4 에서는 다음과 같은 여러 가지 기능이 추가되었다.

- RC 리소스 스트링 테이블 에디터 (Resource String Table Editor) 지원
- DFM 에디터 추가
- 리소스 프로젝트 (Resource Project) DLL 위저드 지원
- OpenHelp 로 온라인 Help 시스템

## 정 리

이번 장에서는 텔파이 4 에서 새롭게 달라진 내용에 대해서 알아보았다. 자세한 내용에 대한 설명은 앞으로 이 책을 읽어가다 보면 알 수 있게 될 것이다.

텔파이 4 는 지금까지의 텔파이의 여러가지 단점을 보강한 명품이며, 가장 최근의 개발 기법을 활용할 수 있는 여러가지 도구를 지원한다.

그러면, 텔파이 4 의 세계로 여행을 떠나 보자 !

## 델파이 4 IDE 의 내부 (Internals of Delphi 4's IDE)

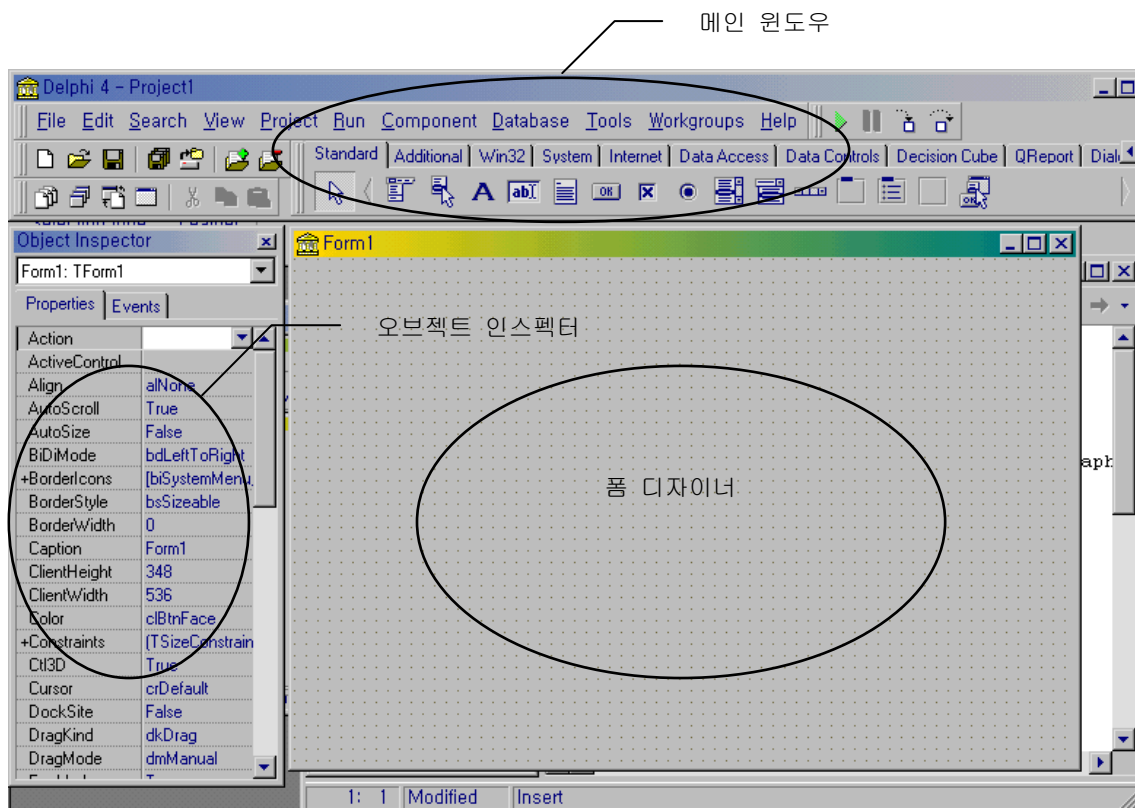
이번 장에서는 델파이의 통합개발 환경에 대하여 알아보도록 한다. IDE(Integrated Development Environment)는 어플리케이션을 설계하고, 실행시키고, 테스트할 수 있도록 해주는 환경을 말하는 것으로, 프로그램을 쉽게 개발할 수 있도록 도와주는 기능을 한다. 과거에는 개발자가 통합개발 환경이 없이 텍스트 에디터로 소스를 편집해서, 컴파일러로 컴파일 하고, 전용 디버거로 디버깅을 했었지만 볼랜드에서 터보 C 를 내놓으면서 처음으로 통합된 개발 환경을 지원하게 되었다.

그 후, 통합개발 환경은 MS 에 의해서도 지원되면서 개발자에게는 점점 더 편리한 환경으로 변모해가고 있는데 델파이 3 까지는 다소 부족한 면이 많다고 느껴왔던 통합개발 환경이 델파이 4 에서는 많이 향상되어 ‘역시 볼랜드’ 라는 탄성이 나오게 하였다.

그럼 델파이 4 의 통합개발 환경에 대해서 알아보는 시간을 가지도록 하자.

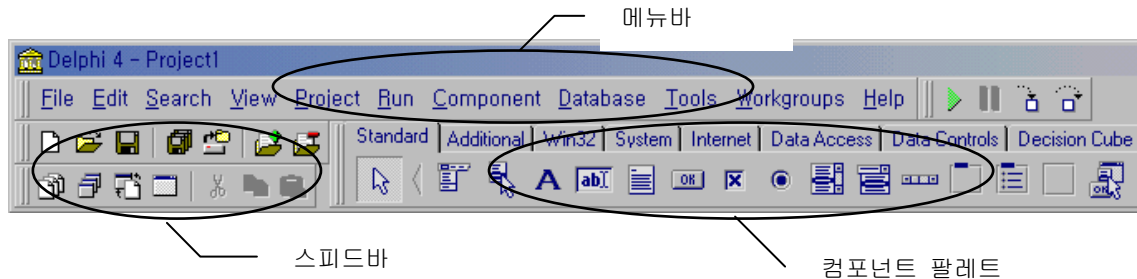
### 메인 윈도우

델파이를 실행시켜서, 프로그램이 모두 로딩되고 나면 다음과 같은 그림이 나타나게 된다.



전체적인 IDE 의 형태는 메인 윈도우와 오브젝트 인스펙터, 그리고 코드 에디터와 모듈 탐

색기와 폼 디자이너로 구성되어 있다. 일단 처음 실행하면 메인 윈도우와 오브젝트 인스펙터, 폼 디자이너가 보이게 된다. 델파이의 메인 윈도우는 크게 나누어 메뉴바(Menubar), 스피드바(Speedbar), 컴포넌트 팔레트(Component Palette)로 이루어진다.



델파이 4의 IDE에서 델파이 3와 바뀐 점을 든다면, 기본적으로 이런 윈도우들이 도킹을 지원한다는 것이다. MS 오피스 97에서부터 채용된 이런 형태의 툴바는 이제는 거의 표준이 되어간다는 느낌이다. 오피스 97과 마찬가지로 델파이 4의 메뉴바, 스피드바, 컴포넌트 팔레트도 마음대로 위치를 이동시킬 수도 있고, floating 윈도우로 나타나게 할 수도 있다.

## 스피드바 (Speedbar)

스피드바는 가장 자주 사용되는 기능들을 쉽고 빠르게 사용할 수 있도록 하기 위해 설계되었다. 기본적으로 가장 자주 사용될 것으로 예상되는 기능을 모아 놓은 것으로, 이들 기능은 델파이의 메뉴바를 통해 메뉴를 직접 선택하여 사용할 수도 있다. 이들 각각에 대해서는 나중에 주요 메뉴를 설명할 때 자세히 기술하도록 하겠다.

## 컴포넌트 팔레트 (Component Palette)

컴포넌트 팔레트는 VCL(Visual Component Library)에 포함되어 있는 구성요소를 가리킨다. 이들 항목들은 개발자가 원하는 대로 그룹을 형성할 수도 있지만, 기본적으로는 기능별로 구성되어 있다. 이들 그룹은 페이지 탭의 형태로 나뉘어 있다. 개발자는 컴포넌트 팔레트에서 사용하고자 하는 컴포넌트를 클릭해서 선택한 후, 이를 폼에 위치시키거나 더블 클릭하여 폼에 컴포넌트를 추가할 수 있다.

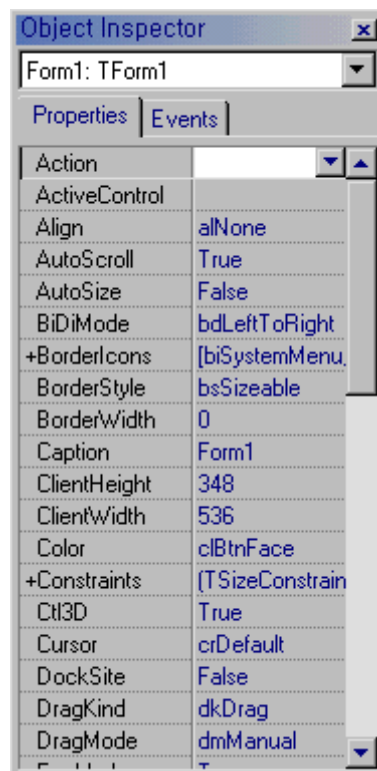
## 폼 디자이너 (Form Designer)

기본적으로 거의 모든 델파이 어플리케이션은 폼으로 구성된다. 델파이에서 폼은 다른 델파이의 컴포넌트들을 위치시킬 수 있는 장소로 사용된다. 개발자는 마우스를 가지고 폼의 위치와 크기 등을 마음대로 조절할 수 있으며, 컴포넌트를

폼에 올려 놓고 자신에 입맛에 맞도록 디자인할 수 있게 된다.

## 오브젝트 인스펙터 (Object Inspector)

오브젝트 인스펙터는 각 컴포넌트의 속성을 변경시키거나, 객체가 반응을 하게 되는 이벤트를 조정하는데 매우 편리한 인터페이스를 제공하고 있다. 오브젝트 인스펙터를 잘 살펴보면, 속성(Properties) 탭과 이벤트(Event) 탭으로 구성되어 있음을 알 수 있다.



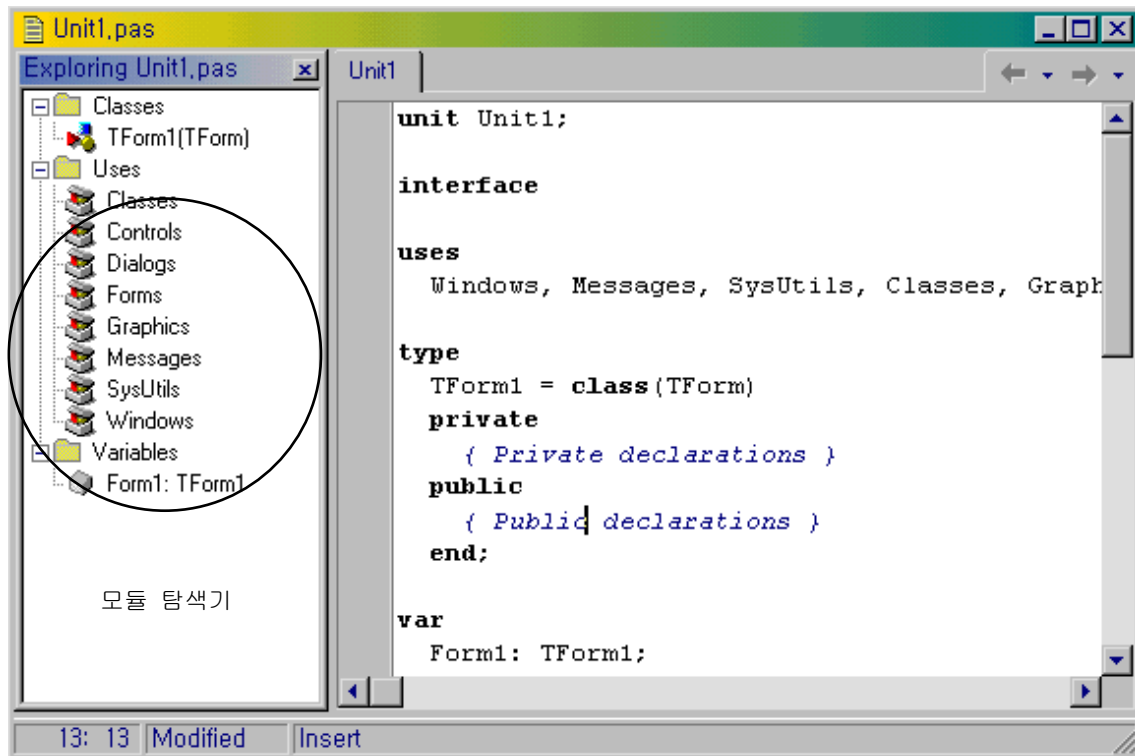
속성 탭에서는 객체의 프로퍼티를 살펴보거나 이를 마음대로 수정할 수 있는 기능을 제공하고 있다. 속성 이름의 옆에 + 기호가 나타나는 것은 그 아래에 하위 속성들이 있음을 나타낸다.

예를 들어, 폼을 선택한 다음에 오브젝트 인스펙터에서 Font 속성을 살펴보면 그 옆에 + 기호가 있음을 볼 수 있다. 그리고 Font 속성을 더블 클릭하거나 + 기호를 클릭하면 글꼴에 대한 Color, Height, Name 등의 하위 속성들이 나타나는 것을 볼 수 있다. 이러한 형식은 객체의 속성을 변경시키는데 매우 간단하면서 효과적인 방법을 제공하고 있다.

이벤트 탭에서는 개발자가 선택한 객체에 반응할 수 있는 이벤트를 선택하여, 이 이벤트가 발생할 때 어떤 동작을 취하라고 지정하는 것이 가능하다. 예를 들어, 어플리케이션에서 윈도우를 닫을 때에 어떤 작업을 실행하고자 한다면, 폼의 OnClose 이벤트를 사용하면 된다.

## 코드 에디터 (Code Editor)와 모듈 탐색기 (Module Explorer)

델파이 4 IDE 에서 가장 많은 변화가 있었던 부분을 꼽으라면 코드 에디터와 모듈 탐색기를 들 수 있다. 먼저 폼 뒤에 숨어 있는 코드 에디터를 살짝 클릭하면 다음과 같이 모듈 탐색기와 코드 에디터가 붙어서 나타나는 것을 볼 수 있다.



모듈 탐색기는 클래스의 생성을 자동화하고, 보다 쉽게 유닛 파일 들을 탐색할 수 있는 기능을 제공한다. 디폴트로 모듈 탐색기는 코드 에디터의 좌측에 도킹되어 있다.

모듈 탐색기를 닫으려면, 코드 에디터에서 패이낸 후 우상부 코너를 클릭한다. 이를 다시 열고자 할 때에는 View|Module Explorer 메뉴를 선택하면 된다.

모듈 탐색기는 유닛에 정의된 모든 데이터 형과 클래스, 프로퍼티, 메소드, 전역 변수와 전역 루틴 등을 보여주는 트리 다이어그램 (tree diagram)을 포함하고 있다. 또한, 여기에는 uses 절에 들어있는 다른 유닛의 내용도 찾아볼 수 있게 되어 있다. 트리 뷰의 노드를 확장하거나 축소하며 뒤져볼 수 있다.

모듈 탐색기와 코드 에디터 사이를 토글하려면, Ctrl+ Shift+ E 키를 누르거나 또는 우측 버튼을 클릭하고 View Editor 메뉴를 선택한다.

모듈 탐색기는 점진적 검색(incremental searching)을 지원한다. 클래스, 프로퍼티, 메소드, 변수, 루틴 등을 검색하려면 단지 그 이름 만을 적어넣으면 된다. 모듈 탐색기에서 아이템



을 선택하면 커서가 코드 에디터에서 연관된 부분으로 이동해 가며, 코드 에디터에서 커서를 이동하면 모듈 탐색기에 적절한 아이템으로 하이라이트된 부분이 옮겨 진다.

또한, 모듈 탐색기의 클래스 완료(class completion), 모듈 탐색 (module navigation) 등의 기능을 이용하면 반복적인 코딩 작업을 자동화할 수 있다.

- 클래스 완료 (Class completion)

텔파이 4 의 클래스 완료 기능을 이용하면 새로운 클래스의 뼈대를 자동으로 만들어 낼 수 있으므로 코딩에 필요한 노력을 많이 줄일 수 있다.

유닛의 interface 섹션의 클래스 선언부에 커서를 위치시키고, Ctrl+ Shift+ C 키를 누른다. 이렇게 하면, 텔파이는 자동으로 프로퍼티에 해당되는 private read, write 필드에 해당되는 부분의 코드를 생성하고, implementation 섹션에 모든 클래스 메소드에 대한 뼈대 코드를 생성한다.

예를 들어, 다음의 코드를 interface 섹션에 작성했다고 하자.

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

여기에 커서를 위치시키고, Ctrl+ Shift+ C 키를 누르면 interface 섹션에는 다음과 같은 코드가 생성된다.

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

그리고, implementation 섹션에는 다음과 같은 코드가 생성된다.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
```

```
end;
```

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

클래스 완료 기능은 implementation 섹션에 정의된 메소드에 대한 interface 선언부를 작성하게 할 수도 있다. 방법은 마찬가지로 implementation 섹션의 메소드 정의부에 커서를 위치시키고 Ctrl+ Shift+ C 키를 누르면 된다.

- 모듈 탐색 (Module navigation)

델파이 4 는 유닛 파일에서 Ctrl+ Shift 키와 각종 방향키를 누르면 쉽게 모듈 전체를 탐색할 수 있는 모듈 탐색 기능을 제공한다. 일단 유닛의 interface 섹션의 특정 메소드나 전역 프로시저의 prototype 에 커서를 위치시키고, Ctrl+ Shift+ Up 또는 Ctrl+ Shift+ Down 키를 누르면 그 프로시저나 함수의 구현 부분으로 이동한다. 마찬가지로 구현 부분에서 이들 키를 누르면 interface 섹션의 선언부로 이동하는 토크 키로 작동한다.

- 코드 브라우저 (Code browser)

코드 에디터에서 Ctrl 키를 누르면서 마우스를 특정 클래스, 변수, 프로퍼티, 메소드 등의 여러가지 identifier 이름 위로 지나가게 하면 마우스 포인터가 손모양으로 변하면서 포인터 위치의 identifier 가 하이라이트 되면서 밑줄이 쳐진다. 이를 클릭하면, 코드 에디터는 그 identifier 의 선언부로 위치를 옮겨간다. 이때 유닛의 interface 섹션에 선언된 메소드나 루틴의 선언부를 찾으려할 때에는 모듈 탐색 기능을 이용해서 Ctrl+ Shift+ Arrow 키를 이용하면 된다.

## 메뉴바 (Menubar)

델파이 환경에서 명령을 실행시키는 방법은 기본적으로 메뉴바의 메뉴를 이용하는 방법과 스피드바를 이용하는 방법, 그리고 마우스의 오른쪽 버튼을 클릭하면 나오는 스피드 메뉴를 선택하는 3 가지 방법이 있다.

여기에서 델파이 4 에서 제공되는 모든 메뉴에 대한 설명을 하는 것은 지면 낭비일 뿐이므로, 델파이의 도움말 파일을 참고하기 바람에 주요 메뉴에 대해서만 설명하도록 하겠다.

## ● File 메뉴

File 풀다운 메뉴에는 프로젝트와 소스 코드 파일에 대한 여러가지 작업 명령들을 포함하고 있다. 프로젝트와 관련이 있는 명령은 New, New Application, Open, Reopen, Save Project As, Save All, Close All 등이 있다. 이것들 외에도 프로젝트에 대해서는 Project 풀다운 메뉴가 특별히 따로 만들어져 있다. 소스코드 파일에 관계되는 명령은 New, New Form, Open, Reopen, Save, Save As, Close, Print 이다. 대부분의 명령들이 직관적으로 금방 알 수 있으므로 설명은 생략하고, 몇 가지 명령에 대한 것만 더 알아보도록 하자.

Reopen 메뉴 명령은 최근에 작업했던 프로젝트나 소스 코드 파일을 열 때 사용하는 것으로 오피스 등의 제품을 사용할 때 보는 history와 비슷한 역할을 한다.

New 명령은 Object Repository에 저장되어 있는 아이템을 재사용할 때 사용하는 메뉴로, New Items 대화 상자를 열게 된다. 여기에는 델파이의 각종 위저드를 불러내거나 새로운 어플리케이션의 형태와 기존의 폼을 상속하는 폼, 쓰레드나 DLL, 델파이 컴포넌트와 각종 액티브 X와 관련된 아이템들을 만들어낼 수 있다. Object Repository에 대한 내용은 나중에 따로 설명하도록 하겠다.

Print 명령은 소스 코드나 폼을 인쇄할 수 있는 명령이다.

## ● Edit 메뉴

Edit 풀다운 메뉴에는 Undo와 Redo, Cut, Copy, Paste와 같이 전형적인 명령들과 폼이나 코드 에디터 윈도우를 위한 몇 가지 특별한 명령들이 포함되어 있다.

이런 명령들은 윈도우 어플리케이션에서 흔히 쓰이는 것들이기 때문에 몇 번만 직접 사용해 보면 어떤 기능을 하는지 쉽게 알 수 있을 것이다.

그 밖에 폼에 관련된 많은 명령어들이 있다. 폼을 위한 명령어들은 폼의 스피드 메뉴(마우스 오른쪽 버튼을 클릭할 때 나타나는 팝업 메뉴)에도 나타나는 것들로 컨트롤을 그리드에 맞추어 정렬하게 하거나, 컨트롤의 앞뒤로 보내는 메뉴, 여러 개의 컨트롤들을 정렬 하거나, 탭 순서를 설정하는 등의 메뉴가 포함된다. 이들 각각에 대한 설명은 도움말을 참고하기 바란다.

참고로 Lock Control 명령의 경우 스피드 메뉴에 나타나지 않는데, 이 명령은 폼 위에서 컴포넌트의 위치가 잘못해서 바뀌지 않도록 하는 역할을 한다. 예를 들어, 어떤 컴포넌트를 더블 클릭하려 했는데 잘못해서 그만 위치를 옮겨 버릴 수도 있는데, 이럴 경우 폼에는 Undo 기능이 없으므로 상당히 곤란할 경우가 있다. 이럴 때에는 폼을 일단 디자인해서 더 이상 바뀔 것이 없다면 그 다음에 컨트롤을 잠궜두면 이런 실수를 막을 수 있다.

## ● Search 메뉴

Search 메뉴에는 기본적인 Search(찾기)와 Replace(바꾸기) 명령과 여러 개의 파일에서 찾기를 할 수 있는 Find in Files 명령이 있다. 또한, 찾고자 하는 문자열을 하나씩 적어나가면서 매칭되는 소스 코드를 찾아주는 Incremental Search 명령도 있다.

Find in Files 명령은 찾기를 원하는 문자열을 라디오 버튼을 체크 함에 따라 프로젝트의 소스 파일들과 또는 모든 열려있는 파일들, 또는 특정 디렉토리 안의 모든 파일들 중에서 찾을 수 있도록 해 준다. 검색 결과는 코드 에디터 윈도우 밑에 있는 메시지 영역에 표시되며, 표시된 내용을 더블 클릭하면 그 파일의 내용이 있는 곳으로 코드 에디터가 옮겨가게 된다.

Incremental Search 기능은 상당히 편리하게 사용할 수 있는데, 이 명령의 단축키인 Ctrl+E 는 외워두었다가 써먹으면 좋을 것이다. 이 명령은 일단 단축키를 누르고 나서 찾고자 하는 문자열을 적어나가면, 여기에 맞는 부분으로 계속 이동해 나간다.

Find Error 명령은 컴파일러 에러가 아닌 특정한 런타임 에러를 찾을 때 쓰이는 명령으로 어떤 단독 실행 프로그램을 실행시키는데 심각한 에러에 부딪히게 되면, 텔파이는 어떤 내부 주소를 가리키는 숫자를 표시하게 된다. 즉, 컴파일된 코드의 논리적 주소를 표시하는 것인데, 이 값을 Find Error 대화 상자에 입력하면 텔파이가 프로그램을 다시 컴파일해서 지정한 주소를 찾아준다. 만약 이 주소를 찾게 되면, 텔파이 코드 에디터에 해당 소스 코드 라인을 찾아서 표시한다. 그런데, 에러가 소스 코드에 있는 것이 아니라 라이브러리나 시스템 코드의 문제로 발생하는 경우도 있다. 이런 경우에는 Find Error 명령으로 오류를 발견할 수 없게 된다.

Browse Symbol 명령은 컴파일된 프로그램에서 정의한 모든 심볼들을 살펴볼 수 있도록 하는 명령으로 Object Browser 를 불러서 이를 표시해 준다.

## ● View 메뉴

View 풀다운 메뉴에는 텔파이 환경의 각 윈도우들을 표시하기 위해 사용된다. 텔파이 환경의 윈도우라면 프로젝트 관리자(Project Manager), 정지점(Breakpoint) 리스트, 모듈 탐색기와 컴포넌트 리스트 등 여러가지가 있게 되는데, 이들 각각은 직접 실행해보면 어떤 윈도우를 가리키는 것인지 알 수 있을 것이다.

프로젝트 관리자와 정지점에 대한 부분은 텔파이 4 에서 많이 바뀐 부분의 하나인데 프로젝트 관리자는 조금 뒤에 설명할 것이다.

View 메뉴에는 이렇게 서로 다른 윈도우를 표시하기 위한 명령들 이외에 여러가지 명령들이 포함되어 있다. Toggle Form/Unit 메뉴는 작업하고 있는 폼과 그 폼의 소스 코드 사이를 토글해주는 명령으로, 상당히 자주 쓰이게 되므로 이 명령의 단축키인 F12 는 외워두기 바란다. 또한, 편리한 명령으로는 New Edit Window 가 있는데, 이 명령을 선택하면 텔파이가 두번째 코드 에디터 윈도우를 열어 준다. 에디터 화면을 두 개 만들어 놓으면 서로

다른 파일을 각각 보이게 해 놓을 수 있고, 한 파일의 서로 다른 부분을 보이게 할 수도 있기 때문에 대단히 편리하다.

마지막 Toolbars 메뉴는 델파이 4 에서 추가된 명령으로 서브 메뉴를 살펴보면, 델파이 4 의 여러 툴바를 보이게 하거나, 숨기게 할 수 있다.

## ● Project 메뉴

Project 폴다운 메뉴는 프로젝트를 관리하고 컴파일하는 명령들을 가지고 있다. Add to Project 와 Remove from Project 명령은 폼이나 파스칼 소스 코드 유닛을 프로젝트에 추가하거나 제거할 때 사용된다.

Import Type Library 명령은 타입 라이브러리를 읽어올 때 사용하는 것으로 이 책의 후반 부에서 자세히 다루게 되므로 설명을 생략하겠다. Add to Repository 는 폼을 Object Repository 에 등록하고자 할 때 사용하는 명령으로 자주 사용되는 폼의 모양을 디자인하고, 이를 계속 재사용할 때 유용하다. View Source 명령은 프로젝트의 소스를 보여준다.

델파이 4 에서는 프로젝트와 관련된 부분이 많이 향상되었으며, 여기에 관련된 명령들이 Project 폴다운 메뉴에 많이 추가 되었다. 이들을 설명하기 위해서 프로젝트 관리자의 변화된 부분에 대해서 조금 더 알아보도록 하자.

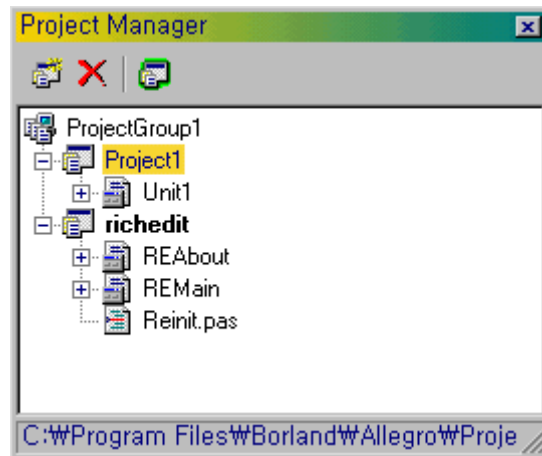
델파이 4 의 프로젝트 관리자는 프로젝트 그룹에서 여러 개의 프로젝트 들을 쉽게 관리할 수 있게 해 준다. 프로젝트 그룹은 상호연관성이 있는 프로젝트 들 (DLL 과 이를 사용하는 어플리케이션, 멀티-tiered 어플리케이션에서 각각의 tier 등)을 유기적으로 관리하는데 편리하게 사용할 수 있다.

프로젝트 관리자를 이용하면 연관된 모든 프로젝트의 파일 들을 볼 수 있으며, 이들을 디스플레이 하고, 파일을 추가 삭제하고 컴파일하는 등의 여러가지 조작을 할 수 있다. 또한, 이들을 한꺼번에 컴파일 할 수도 있다.

프로젝트 관리자의 메인 화면을 살펴보면 프로젝트 그룹이나 프로젝트에 속해있는 모든 파일 들을 트리의 형태로 관찰할 수 있다. 여기에서 트리의 루트는 프로젝트 그룹이며, 프로젝트 그룹에는 각각의 프로젝트를 나타내는 아이콘을 포함하고 있다. 프로젝트의 종류는 DLL, EXE, 패키지 또는 리소스 등일 수 있다. 프로젝트가 프로젝트 그룹의 일부이면, 프로젝트 관리자는 프로젝트 그룹에 있는 모든 프로젝트 들에 대한 정보를 제공한다. 우측의 파일 뷰에는 프로젝트 내의 모든 소스 파일(.pas, .rc 파일 등)과 이진 객체 파일(.res, .lib, .obj 파일 등)을 들을 보여 준다.

각각의 프로젝트 파일은 .dpr 확장자를 가지고 있다. 프로젝트 관리자를 이용해서 파일을 추가, 제거하면 델파이는 프로젝트 파일을 자동으로 업데이트 해 준다. 프로젝트 그룹 파일은 .bpg 확장자를 가지고 있으며, 프로젝트 그룹에 프로젝트를 추가, 삭제 할 때마다 내용이 바뀌게 된다.

델파이 4 의 프로젝트 관리자의 실제 모습은 다음과 같다.



Create New Target 과 Open New Target 명령은 델파이 4 에서 새롭게 추가된 것으로 새로운 어플리케이션이나 DLL, 패키지 등의 아이템을 프로젝트 그룹에 추가할 때 사용한다. 또한, 델파이 4 에서는 복수 프로젝트를 관리할 수 있는 프로젝트 관리자에 부합하여, 컴파일과 관련된 명령 들이 많이 추가되었다. Compile, Build, Syntax Check 등의 기존 명령 말고도 Make 명령이 추가 되었는데, 이 명령들의 뒤에는 대상이 되는 프로젝트의 이름이 같이 디스플레이된다. 또한, 프로젝트 그룹에 있는 모든 프로젝트를 한꺼번에 컴파일할 때 사용할 수 있는 Compie All Projects, Build All Projects 명령이 추가되었다.

Information 명령도 프로젝트 그룹안에서 컴파일된 특정 프로젝트에 대한 정보를 보여주어야 하므로, Information for [프로젝트 이름]의 형태로 바뀌었다.

Web Deploy, Web Deploy options 명령은 액티브 X 폼과 컨트롤에 대한 것으로 이 책의 후반부에서 자세히 다루게 될 것이다.

프로젝트 메뉴의 가장 마지막 명령은 Options 메뉴이다. 여기에서는 컴파일러와 링커 옵션, 어플리케이션 객체의 여러가지 옵션을 설정할 수 있다.

## ● Run 메뉴

Run 메뉴는 주로 디버깅에 관련된 내용을 많이 담고 있다. 델파이 환경에서 Run 명령을 선택하면 작성된 어플리케이션은 델파이의 내장 디버거 내에서 실행된다. 물론 이 기능을 환경 옵션에서 해제할 수도 있다. 어쨌든 Run 명령은 델파이를 사용할 때 가장 자주 사용하게 되는 명령이므로 F9 단축키는 외워두는 것이 좋을 것이다.

Parameters 명령은 커맨드 라인을 실행시키려고 하는 프로그램에 전달하고, DLL 을 디버그 할 때에는 실행 파일의 이름을 제공하기 위해 파라미터를 설정할 수 있게 한다. 이 명령도 델파이 4 에서 향상된 것 중에 하나인데, 기존의 파라미터 설정 탭에 원격으로 디버깅을 할 수 있게 하기 위해 호스트 어플리케이션과 원격지 패스 등을 설정할 수 있도록 변경되었다. Run 풀다운 메뉴에서 디버깅에 관련된 명령 이외에는 액티브 X 개발에 관련된 명령이 몇

가지 있다. Register ActiveX Server 와 UnRegister ActiveX Server 명령은 현재 프로젝트에 의해 정의되어 있는 액티브 X 컨트롤에 대한 윈도우 레지스트리 정보를 추가하거나 삭제하는 역할을 한다. 또한, Install MTS Objects 메뉴를 통해 마이크로소프트 트랜잭션 서버를 지원하는 객체를 설치할 수 있다.

- Component 메뉴

Component 메뉴의 명령들은 컴포넌트를 작성하고 이것들을 패키지에 넣거나 패키지를 델파이에 설치하는데 주로 사용된다. New Components 명령은 간단한 컴포넌트 위저드를 호출하여 컴포넌트를 새로 작성하는데 도움을 주며, Install Components, Import ActiveX Library, Install Packages 명령은 새로운 델파이 컴포넌트, 패키지 또는 액티브 X 컨트롤을 환경에 추가하여 사용할 수 있도록 해준다. 이들에 대한 더욱 자세한 내용은 제 4 부에서 다루게 된다.

Create Component Template 명령은 하나 이상의 컴포넌트를 폼에서 선택하고 이 명령을 호출하면 새로운 컴포넌트의 이름, 팔레트에서의 페이지, 아이콘을 입력하는 대화 상자가 나타나는데 이를 이용해 새로운 컴포넌트 템플릿을 구성할 수 있게 해준다.

- Database 메뉴

Database 풀다운 메뉴에는 Database Form Wizard, SQL Explorer, SQL Monitor 등의 데이터베이스 관련 도구를 호출할 수 있는 메뉴가 모여 있다. 이들을 선택하면 데이터베이스 도구들이 실행되는데, 여기에 대한 자세한 내용은 13 장의 내용을 참고하기 바란다.

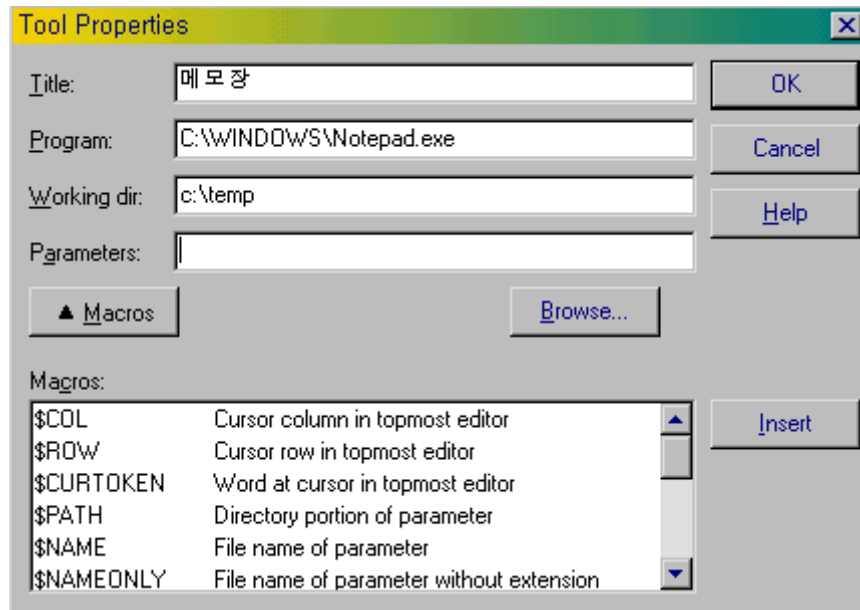
- Tools 메뉴

Tools 풀다운 메뉴는 외부 프로그램과 툴들을 실행시키기 좋게 모아놓은 것과 델파이 개발 환경의 옵션을 설정하는 명령, Object Repository 의 초기화를 위한 명령이 포함되어 있다.

Environment Options 대화 상자에는 포괄적인 환경 설정, 패키지와 라이브러리 설정, 많은 에디터 옵션, 컴포넌트 팔레트 설정, Object Browser 설정, 코드 인사이트 설정 등을 할 수 있는 많은 페이지가 있다. 이들 각각에 대한 것은 도움말을 참고하기 바란다.

또한, Configure Tools 명령을 이용하면 자신이 자주 쓰는 외부의 도구를 등록했다가 쉽게 불러낼 수 있다. 여기서 간단히 메모장(Notepad)을 등록해 보도록 하자.

Configure Tools 명령을 선택하면 대화 상자가 나타나는데, 여기서 Add 버튼을 클릭하면 등록할 도구의 속성을 설정할 수 있는 대화 상자가 보인다. 속성을 다음과 같이 설정하도록 하자.



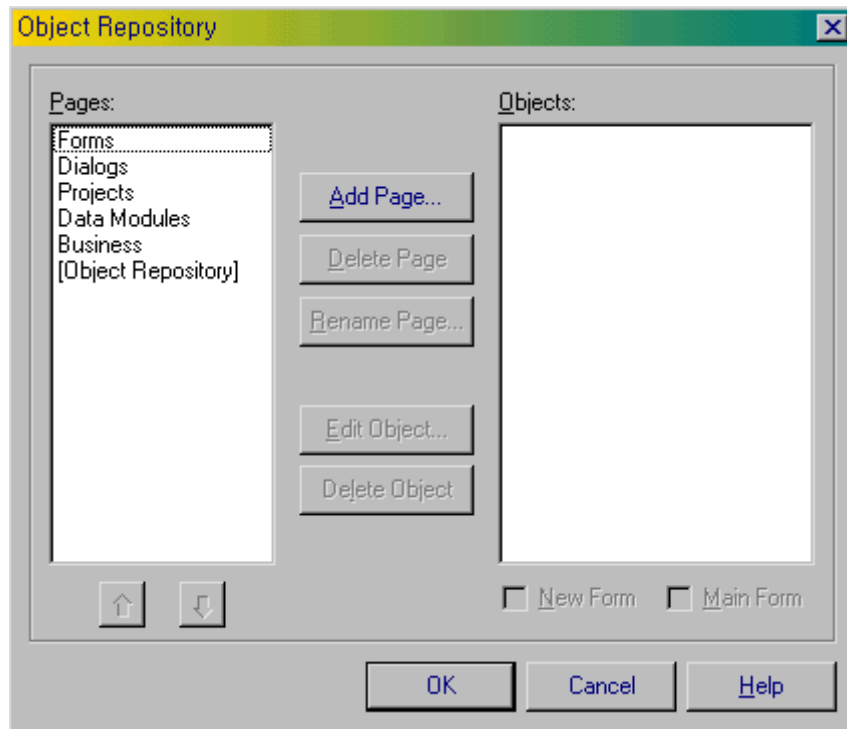
Browse 버튼을 클릭해서 추가하고자 하는 도구의 실행 파일을 선택하고, 이 도구가 메뉴에 나타나게 될 타이틀과 작업 디렉토리를 설정하고 OK 를 선택하면 Tools 메뉴에 메모장이 추가되며, 이를 선택하면 메모장이 실행된다.

- 그 밖에도 Workgroup 과 Help 메뉴가 있는데, Workgroup 메뉴에는 텔파이의 버전 컨트롤 프로그램인 PVCS 를 실행시키는 명령이 포함되어 있으며 Help 메뉴에서는 도움 말을 불러올 수 있다.

## 객체 저장소 (Object Repository)

텔파이 1.0 의 Gallery 는 템플릿의 저장과 폼 위저드의 기능으로 사용되었다. Gallery 는 텔파이 2.0 이후부터 객체 저장소로 바뀌게 되었다. Tools 메뉴에서 Repository 를 선택하면 다음과 같은 대화상자가 나타나게 된다.





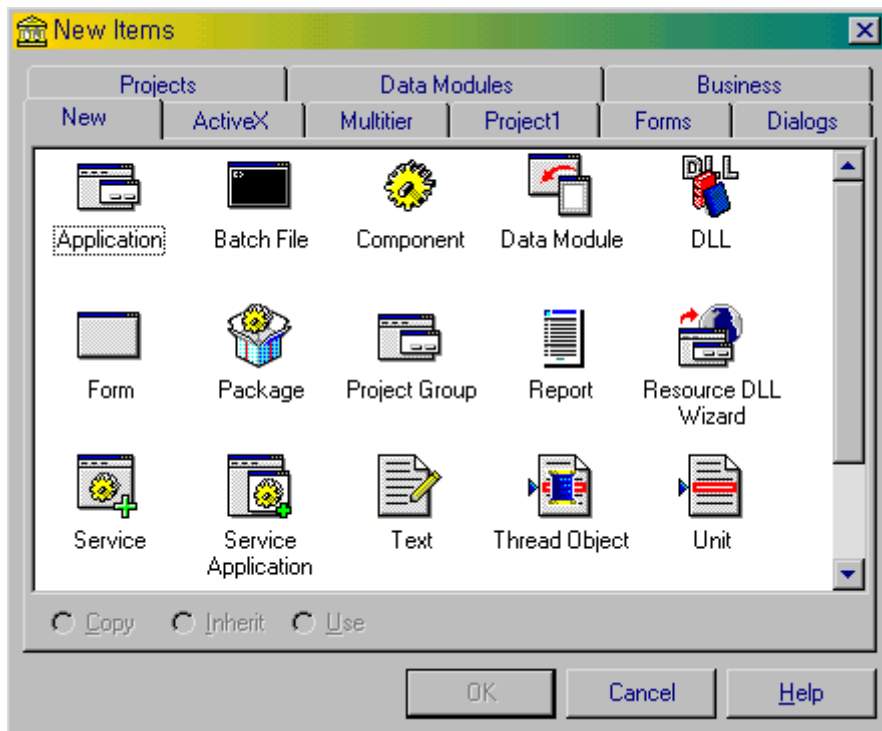
페이지 리스트 박스에는 Forms, Dialogs, Projects, Data Modules, Object Repository 의 5 가지 선택 항목이 나타난다. 이 대화상자를 사용하면 선택 항목을 편집하거나 새로운 폼이나 프로젝트를 생성할 때 사용되는 기본 값을 변경시킬 수도 있다. 여기서는 개발자 자신의 페이지를 생성하거나 Object Repository 대화 상자에서 저장소(Repository)로 추가시킬 수 있다.

이러한 객체 저장소의 기능은 델파이에서 코드의 재사용 기능을 더욱 강력하게 만들어 주고 있다. 대부분의 가장 일반적으로 사용되는 기능들은 이미 기본적으로 제공되고 있지만, 개발자 자신의 것들을 생성하여 추가하면 그 활용도는 훨씬 높을 것이다.

델파이에서 File|New 메뉴를 선택했을 경우에 델파이는 객체 저장소를 연다. 객체 저장소의 여러 페이지 중에서 New 페이지와 ActiveX 페이지에는 여러 가지 형태의 새로운 아이템을 만들 수 있도록 한다.

다음 그림의 객체 저장소 대화 상자 밑에 있는 라디오 버튼을 눌러서 기존의 아이템을 복사할 것인지, 아니면 상속할 것인지, 그대로 사용할 것인지를 지정하게 되는데 위저드의 경우처럼 이런 내용을 선택할 수 없는 경우에는 라디오 버튼의 기능이 비활성화 상태로 나타난다.

객체 저장소에 제공되는 기본 아이템에 대해 모두 설명하는 것은 지면 관계상 생략하도록 하고, 델파이 4 에서 새롭게 제공되는 것이 어떤 것들이 있는지 알아보도록 하자. 델파이 4 에서는 객체 저장소에 새로운 위저드를 많이 지원하는데, 여기에는 다음과 같은 것들이 있다. 이들 각각에 대한 자세한 내용은 해당되는 장을 참고하기 바란다.



- CORBA Data Module Wizard:  
데이터 모듈이 CORBA 를 지원하도록 해주는 위저드이다.
- MTS Data Module Wizard:  
데이터 모듈이 마이크로소프트 트랜잭션 서버를 지원하도록 하는 위저드이다.
- Project Group: 새로운 프로젝트 그룹을 생성한다.
- Resource DLL Wizard:  
세계화를 위해서 필요한 위저드로 문자열을 여러 나라의 언어로 지정할 수 있는 쉬운 방법을 제공하며, 하나의 프로젝트를 여러 나라 버전으로 저장 관리할 수 있게 해준다.
- Service Wizard: 윈도우 NT 서비스를 생성해주는 위저드이다.
- Service Application Wizard: 윈도우 NT 서비스 어플리케이션을 생성해준다.
- COM object Wizard: COM 객체를 생성해주는 위저드이다.
- MTS Automation Object Wizard: MTS 자동화 객체를 자동으로 생성해준다.

## 정 리

이번 장에서는 델파이 4 의 IDE 에 대해서 간단하게 알아보았다. 더욱 자세하게 설명할 수도 있겠으나, 이 책에서는 주로 다른 책에서는 다루지 못한 여러 가지 테크닉 들을 많이 소개하려고 하기 때문에 단순히 도움말을 찾아보면 알 수 있는 내용들은 되도록 생략하였다.

더 자세한 사항은 도움말을 직접 참고하기 바란다.

텔파이 4 는 그동안 개발자들이 불편해하던 많은 부분을 해결한 멋진 IDE 를 제공하고 있다. 복수로 프로젝트를 관리할 수 있게 되었고, 모듈 탐색기를 통해 interface 섹션과 implementation 섹션 사이의 이동과 코드 에디터 내부에서의 탐색 기능의 효율을 높였다.

## 델파이 4 프로그래밍의 이해

### (Understandings of Delphi 4 Programming)

오브젝트 파스칼과 심도있는 델파이 프로그래밍의 세계로 들어가기 전에, 이번 장에서는 델파이를 사용하여 첫번째 윈도우 어플리케이션을 제작하고 전반적인 델파이의 환경에 대해서 알아볼 것이다. 내용의 수준이 높지는 않겠지만, 흔히 알고 있었던 내용이라고 하더라도 별 생각 없이 넘어갔던 것들도 많을 것이다.

그러면, 델파이 4 의 세계의 역사적인 첫 발을 들여놓도록 하자 !

#### 첫번째 어플리케이션

델파이를 일단 시작하면 기본적으로 새로운 프로젝트가 하나 시작되며, 비어 있는 폼이 나타난다. 아마도 오브젝트 인스펙터에는 현재 가리킬 수 있는 컴포넌트가 Form1 뿐일 것이므로 Form1 의 프로퍼티 값들을 표시하고 있을 것이다. 오브젝트 인스펙터의 사용법을 익히기 위해 먼저 폼의 캡션을 바꾸어 보자.

오브젝트 인스펙터의 Caption 프로퍼티를 'Form1' 에서 'Hello'로 한번 바꾸어 보자. 이 간단한 동작만으로 폼의 타이틀 바의 이름이 바뀌는 것을 관찰할 수 있을 것이다. 이제 Run 명령을 선택하거나 화살표 모양의 스피트 버튼을 클릭하면 이 어플리케이션이 다음과 같이 실행되는 것을 관찰할 수 있을 것이다.



이제 델파이 환경에서 실행된 어플리케이션을 종료하고 다시 폼 디자이너로 돌아오자.  
 많은 작업을 하지는 않았지만, 시스템 메뉴와 기본적으로 제공되는 전체 화면 표시 버튼과  
 최소화 표시 버튼, 닫기 버튼을 가지는 완전한 어플리케이션을 방금 하나 만든 것이다. 이  
 폼을 마우스를 이용해 크기를 조절할 수도 있고, 전체 화면으로 보거나 최소화할 수도 있다.

## 어플리케이션의 저장과 파일의 종류

이제 이렇게 만든 어플리케이션 소스를 저장해보자. File 메뉴에서 Save All 을 선택하면,  
 델파이는 폼과 관련된 소스 코드와 프로젝트 파일에 이름을 붙여 저장하게 된다. 먼저 파  
 스칼 소스 코드인 .pas 파일의 이름을 물어오는데, 여기에는 U1\_Exam1.pas 라고 명명하고  
 적당한 디렉토리에 저장한다. 마찬가지로 프로젝트 파일인 .dpr 파일은 Exam1.dpr 로 명  
 명한다.

### 참고:

이 책에서 소스 코드의 이름을 명명할 때에는 Exam 이라는 문자열에다가 그 장에서 제작하는 예제가  
 몇 번째 것인지에 따라 일련 번호를 붙여서 사용한다. 이번 장과 같은 경우 3 장의 첫번째 예제이므  
 로 Exam1.dpr 이 되며, 각 유닛 파일에는 접두어로 유닛의 개수에 따라 U1, U2 ... 등의 문자열을 사  
 용한다. 각 장의 예제는 다른 디렉토리에 저장된다.

프로젝트 파일에 붙인 이름은 실행 시에 디폴트로 어플리케이션의 제목으로 사용되어 윈도우  
 의 작업 표시줄에 나타나게 된다. 그러므로, 만약 프로젝트의 이름이 메인 폼의 제목과  
 같으면 이 이름은 작업 표시줄의 이름과도 일치하게 된다.

그러면, 델파이에서 사용하는 파일의 종류와 이들이 어떤 파일 들인지에 대해서 알아보도록  
 하자. 델파이 프로젝트는 폼, 유닛, 옵션 설정과 리소스에 대한 파일 들로 구성된다. 이런  
 모든 정보가 각각 다른 파일로 저장되며, 델파이에서 어플리케이션을 제작할 때에 생성된다.  
 다음 표는 델파이 4 에서 사용되는 파일 들의 종류와 이들의 역할에 대해서 나열한 것이다.

파일의 종류	설 명
프로젝트 그룹 파일 (.bpg)	델파이 4 에서 새롭게 추가된 프로젝트 그룹에 대한 파일로, 여러 프로 젝트를 관리할 수 있는 프로젝트 그룹의 내용을 담고 있는 파일이다.
프로젝트 파일 (.dpr)	폼, 유닛 등에 대한 정보를 저장하는데 사용되는 파일이다.
유닛 파일 (.pas)	코드를 저장하는데 사용되는 파일로, 폼과 연관되기도 하며 어떤 것들 은 함수와 프로시저만을 저장하기도 한다.
폼 파일 (.dfm)	폼에 대한 정보를 저장하기 위해 생성되는 이진 파일이다. 각 폼 파 일은 유닛 파일과 연관되어 있다. 예를 들어, mine.pas 유닛 파일은 mine.dfm 이라는 폼 파일을 가진다.

프로젝트 옵션 파일 (.dfo)	프로젝트의 옵션 설정이 이 파일에 저장된다.
패키지 정보 파일 (.dfr)	델파이를 패키지와 함께 사용하는 이진 파일이다.
리소스 파일 (.res)	프로젝트에서 사용하는 각종 리소스를 저장하는 파일이다. 이 파일은 개발자가 생성하거나 변경하는 것이 아니고 델파이가 계속적으로 고치거나 다시 생성한다.
백업 파일 (.~dp, ~df, ~pa)	프로젝트, 폼, 유닛 파일 등에 대한 백업 파일이다. 여러 번 고친 경우에는 확장자의 앞글자 2 자 뒤에 고칠 때마다 하나씩 증가하는 정수로 명명된다. 예를 들어 유닛 파일의 경우 .pa1, .pa2, .pa3 등이다.
실행 파일 (.exe)	어플리케이션의 실행 파일로, 단독 실행이 가능하다.
유닛 객체 파일 (.dcu)	유닛 파일의 컴파일된 형태로, 최종 실행 파일에 링크된다.
타입 라이브러리 (.tlb)	액티브 X/COM 에서 사용되는 타입 라이브러리 파일

## 델파이 프로젝트

일반적으로 프로젝트란 어플리케이션에 있는 모든 객체를 포함하는 최상위 컨테이너를 말한다. 즉, 어플리케이션을 생성하는 각각의 파일들을 서로 연결한다. 하나의 프로젝트는 모든 객체의 저장소(repository)와 같은 역할을 하게 된다.

델파이의 프로젝트 역시 하나의 어플리케이션을 구성하는 모든 파일 들의 목록을 가지고 있다. 그런데, 다른 개발 툴과는 달리 그 자체의 기능을 가지는 프로그램 소스 코드를 포함하는데, 이를 볼 수도 있고 필요에 따라서 수정해서 쓸 수도 있다. 그렇지만, 프로젝트 파일의 코드는 대부분 델파이가 알아서 코딩을 해주기 때문에 건드릴 필요가 거의 없다.

### ● 프로젝트 파일의 이해

별로 건드릴 필요가 없는 파일이더라도, 어떻게 프로젝트를 이루는 코드가 되어 있는지는 알아야 할 것이다. 그렇다면, 프로젝트 파일의 소스 코드를 이해해 보도록 하자. 프로젝트 파일의 소스 코드를 보기 위해서는 Project|View Source 명령을 선택하면 된다. Exam1.dpr 프로젝트의 소스 코드는 다음과 같다.

```
program Exam1;

uses
  Forms,
  U1_Exam1 in 'U1_Exam1.pas' {Form1};

{$R *.RES}
```

```
begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

- program 키워드

컴파일러에게 이 파일은 실행파일이 된다는 것을 알려준다. DLL 이거나 유닛일 경우에는 library 나 unit 키워드를 사용하게 된다.

- Uses 구문

Uses 구문은 델파이가 실행파일을 만들 때 링크할 오브젝트 파스칼 유닛 들을 나열할 때 사용한다.

- \$R 지시자

\$R 컴파일러 지시자는 컴파일러에게 지정된 윈도우 리소스를 사용하라고 알려주는 역할을 한다. \$R 뒤에 나오는 별표(asterisk ‘\*’)는 리소스 파일이 프로젝트와 같은 이름을 사용한다는 것을 나타낸다. 프로젝트를 빌드하면 델파이는 프로젝트 자체와 각 폼에 대한 리소스 파일을 생성하게 된다.

- Application.CreateForm

Application.CreateForm 구문은 프로젝트의 폼을 메모리로 읽어들인다. 일반적으로 프로젝트의 모든 폼은 여기에 나열된다. Option|Project 메뉴를 이용해 폼을 자동으로 생성할 것인지 여부를 지정해줄 수 있는데, 각각의 폼은 각 폼의 인스턴스 변수(예를 들어 Form1)에 저장되며 이들은 각 폼 유닛의 interface 섹션에 정의되어 있다. Application.CreateForm 구문은 지정된 폼을 메모리로 읽어들이고, 인스턴스 변수에 그 폼에 대한 포인터를 저장한다.

프로젝트 파일에서의 Application.CreateForm 구문의 순서는 실제로 폼이 생성되는 순서이며, 첫번째로 생성되는 폼이 메인 폼이 된다. 이 순서를 바꾸려면 Project|Option 메뉴의 Application 탭을 이용하면 된다.

- Application.Run

이 구문에 의해 애플리케이션이 동작하게 된다.

● 유닛 파일의 이해

델파이의 각 폼은 그에 해당하는 유닛 파일을 하나씩 가지고 있다. 여기에는 폼의 모양을 나타내는 클래스 정의가 포함된다. 새로운 컴포넌트를 폼에 추가할 때마다 델파이의 폼 디자이너는 이를 반영하기 위해 폼의 클래스 선언부분을 변경한다. 또한, 이벤트 핸들러를 추가할 때마다 여기에 해당되는 코드가 유닛 파일에 저장된다.

유닛 파일은 기본적으로 interface, implementation, initialization, finalization 섹션으로 구분된다. 현재의 폼에 대한 유닛 파일인 U1\_Exam1.pas 파일의 소스 코드는 다음과 같다.

```
unit U1_Exam1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
  StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)  
    btnOK: TButton;  
  private  
    { Private declarations }  
  public  
    { Public declarations }  
  end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
end.
```

- interface 섹션

여기에는 유닛의 헤더 정보가 기록된다. 즉, 각종 함수, 프로시저의 선언부와 유닛의



외부에서 접근할 수 있는 변수, 상수, 타입의 정의가 위치하게 된다. 과거에 C 를 써본 경험이 있는 사람이라면 이 부분의 C 의 헤더 파일과 비슷한 역할을 한다고 생각하면 된다. C 와는 달리 이 섹션을 분리된 소스 코드 파일로 저장하지 않지만, 다른 모듈들이 이를 참조할 수 있다. 컴파일된 유닛은 인터페이스 정보를 헤더 부분에 저장하기 때문에 다른 모듈이 Uses 구문을 이용해서 유닛을 참조하면 델파이는 소스 코드를 직접 찾는 것이 아니라, 컴파일된 유닛(dcu 파일)의 헤더의 정보를 사용한다. 그렇기 때문에 델파이의 유닛을 컴파일된 dcu 파일(오브젝트 코드)로 배포할 수 있는 것이다.

#### - implementation 섹션

implementation 섹션은 유닛의 실제 프로그래밍 코드가 담겨 있는 부분이다. 여기에 적힌 코드는 interface 섹션에서 나열된 부분만 외부에서 볼 수 있게 되며, 보통은 interface 섹션에서 나열한 부분을 구현하는 코드가 존재한다.

참고: 폼 인스턴스 변수 (form instance variable)

각 유닛의 interface 섹션에 보면 폼 인스턴스 변수를 선언한 부분이 있다.

```
var
```

```
Form1: TForm1;
```

즉, TForm1 이라는 타입(type)의 Form1 이라는 변수를 선언하는 구문인데, 이때 TForm1 은 TForm 클래스를 상속받아서 델파이 폼 디자이너에 의해 생성된 새로운 클래스이다.

이렇게 선언된 폼 인스턴스 변수는 프로젝트 파일의 Application.CreateForm 이 호출될 때 초기화되며, 유닛의 interface 섹션에 선언되어 있으므로 다른 모듈에서 Uses 구문을 사용해서 이를 사용하여 폼에 접근할 수 있게 된다.

#### - initialization, finalization 섹션

앞의 소스에는 존재하지 않지만, 필요할 때 선언해서 사용하면 된다. 유닛이 처음 로드될 때 실행해야 하는 코드가 있으면, 이를 initialization 섹션에 위치시키면 된다. 또한, 어플리케이션이 메모리에서 해제될 때 마지막으로 실행해야 하는 코드가 있으면, 이는 finalization 섹션에 위치시키면 된다. 보통 유닛에서 사용한 리소스 등을 해제하는 코드 등을 여기에 사용한다.

#### ● 델파이 폼 파일의 이해

델파이의 폼 디자이너에서 작업한 내용들은 폼 파일에 저장된다. 그러므로, 폼 디자이너에서 비주얼한 환경으로 폼을 다룰 수도 있지만, 텍스트 파일을 직접 변경하는 것도 가능하다

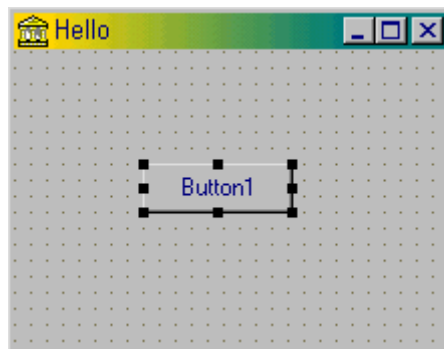
(마치 HTML 문서를 만들 때, 텍스트 에디터를 쓸 수도 있고 프론트 페이지 같은 비주얼 툴을 사용할 수도 있는 것 처럼).

## 컴포넌트의 사용

폼이 델파이 신전을 이루는 기둥이라고 하면, 컴포넌트는 기둥을 이루는 벽돌이라고 할 수 있다. 이러한 컴포넌트 들은 오브젝트 파스칼 소스 코드로 만들어져 있다.

그러면 벽돌을 이용해서 집을 지어보도록 하자. 폼에 컴포넌트를 추가하는 방법은 컴포넌트 팔레트에서 컴포넌트를 클릭하고, 마우스 커서를 폼으로 이동시킨 후, 왼쪽 버튼을 눌러서 컴포넌트의 좌상부 꼭지점을 지정하고 계속 버튼을 누른 채로 컴포넌트의 우하부 꼭지점 까지 드래그 하면 컴포넌트의 크기가 알맞게 정해진다. 또는, 그냥 컴포넌트를 선택하고 폼의 적당한 위치에 클릭하면 디폴트 크기의 컴포넌트가 생성되며, 컴포넌트 팔레트에서 컴포넌트를 더블 클릭하면 폼의 한 가운데에 컴포넌트가 생성된다.

Standard 페이지에서 버튼 컴포넌트 하나를 선택해서 폼에 올려 놓도록 하자.



### ● 프로퍼티 편집하기

앞에서 폼의 캡션을 변경한 것과 마찬가지로, 다른 컴포넌트들도 오브젝트 인스펙터를 이용해서 프로퍼티를 편집할 수 있다. 버튼의 캡션을 바꾸기 위해서는 먼저 마우스로 버튼 객체를 선택하고 오브젝트 인스펙터의 Caption 프로퍼티를 변경하면 된다. 참고로, 컴포넌트의 캡션 프로퍼티는 보통 컴포넌트의 Name 프로퍼티에 우선적으로 영향을 받는다. 물론 이름과 캡션의 문자열은 달라도 되지만, Name 프로퍼티를 변경하면 Caption 이 처음에 자동으로 설정된다. 보통 컴포넌트의 이름을 붙일 때에는 일반적으로 따르는 관습이 있는데, 많은 경우에 영어 자음으로 된 소문자 2~3 자를 붙이는 이름 규칙이 가장 많이 쓰인다. 예를 들어 버튼의 경우 'btn' 이라는 문자열을 접두어로 사용하는 경우가 많다.

#### 참고:

Name 과 Caption 프로퍼티는 처음으로 델파이를 접하는 사람들이 혼돈스러워 하는 것 중에 하나이

다. 분명히 말해서 Name 프로퍼티는 어디까지나 내부에서 사용되는 것으로 컴포넌트를 나타내는 변수의 이름이라고 생각하면 된다. 그러므로, Name 프로퍼티는 기본적으로 파스칼의 변수의 이름 규칙을 따라야 한다. 이에 비해 Caption 프로퍼티는 바깥에 보이는 부분으로 내부적인 컴포넌트의 이름과는 전혀 상관이 없다.

어쨌든 Button1 의 Name 프로퍼티를 'btnOK'로 설정하고 Caption 은 'OK' 라고 정하자. 이런 형식으로 모든 델파이 컴포넌트의 속성을 정할 수 있게 된다.

## ● 이벤트 핸들러의 작성

폼이나 컴포넌트에서 마우스 버튼을 누르면, 윈도우는 어플리케이션에 메시지를 보내서 그 이벤트를 알려준다. 델파이에서의 이벤트는 크게 2 가지 의미로 생각할 수 있다. 하나는 각 컴포넌트 별로 윈도우의 메시지를 포장한 것이다. 즉, 이벤트의 발생이라는 측면에서 접근하면 어디까지나 윈도우 메시지와 동일하다는 것이다. 예를 들어, 마우스 키를 누르면 해당 윈도우에는 WM\_MOUSEBUTTONDOWN 이라는 메시지가 전송되고, 델파이의 컴포넌트는 이를 OnMouseDown 이라는 이벤트로 발생시킨다는 것이다. 이보다 조금 언어적인 측면에서 접근하면 델파이의 이벤트는 개발자가 작성한 함수나 프로시저의 주소를 윈도우 메시지가 발생했을 때 연결해주는 함수 포인터이다. 즉, 오브젝트 인스펙터에서 객체의 published 프로퍼티로 선언된 프로시저형 데이터로 눈으로 볼 때에는 오브젝트 인스펙터의 이벤트 탭에서 나열된 메소드를 선택하여 이벤트와 메소드를 단순히 연결하는 것으로 생각할 수 있지만, 내부적으로는 호환 가능한 프로시저형의 메소드 포인터를 이벤트 탭에서 열거하면, 이를 선택하는 것으로 함수 포인터와 같은 역할을 하는 것이다.

다소 설명이 어려웠을 지도 모르지만, C 를 공부했던 독자라면 조금은 쉽게 이해했을 것으로 믿는다. 그리고, 잘 이해가 되지 않더라도 델파이로 여러 프로그램을 만들다 보면 이해가 될 날이 올 것이다.

어쨌든 이번 장의 목적은 가장 단순한 형태의 어플리케이션을 한 번 제작해보는 것이므로, 실전에 들어가 보도록 하자.

먼저 폼에 올려 놓은 버튼을 선택하고, 오브젝트 인스펙터의 이벤트(Event) 탭을 선택한다. 버튼을 클릭할 때의 이벤트 핸들러를 작성하려면 OnClick 이벤트 오른쪽 옆의 하얀 부분을 더블 클릭하거나, 여기에 새로운 메소드의 이름을 입력하고 Enter 를 치면 이벤트 핸들러를 입력할 수 있는 화면이 다음과 같이 생성된다.

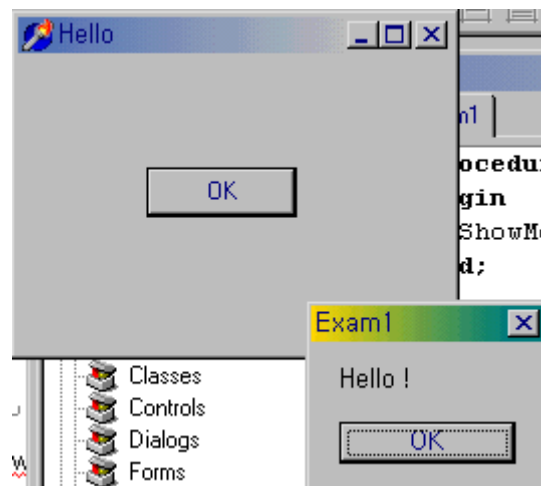
```
procedure TForm1.btnOKClick(Sender: TObject);  
begin  
  
end;
```

이제 이벤트 핸들러의 코드를 begin~end; 사이에 집어 넣으면 된다. 오브젝트 파스칼에 대한 문법을 모르더라도 걱정하지 말고, 'Hello' 라는 메시지를 출력하기 위해 다음과 같이 코드를 입력하도록 하자.

```
procedure TForm1.btnOKClick(Sender: TObject);  
begin  
    ShowMessage('Hello !');  
end;
```

ShowMessage 는 메시지 박스를 보여주는 것으로, 파라미터로 넘어간 문자열을 단순히 보여주는 역할을 하는 것이다.

그러면, 화살표 스피드 버튼을 누르거나 Run 메뉴를 선택하여 어플리케이션을 컴파일하고 실행해보자. 버튼을 누르면 다음과 같은 화면이 나타날 것이다.



어플리케이션을 실행하면, 델파이 내부에서는 폼을 구성하는 파스칼 소스 코드를 컴파일 하고, 프로젝트 파일을 컴파일 한 후 해당되는 라이브러리와 링크할 실행 파일을 만들게 된다. 그리고 나서, 디버그 모드에서 실행 파일을 실행하는 것이다.

## 정 리

델파이를 이용해서 프로그래밍을 하는 방법을 대단히 간단히 알아보았다. 물론 우리가 이번 장에서 제작한 프로그램은 그렇게 큰 의미가 있는 것이 아니다. 하지만, 이 프로그램이 실제로 어떻게 구성되고 기본적인 작동 방법은 어떻게 하는 것인지를 익히는 데에는 충분했을 것이다.

이 책의 목표가 초급자에게 텔과이에 대해서 아주 자세하게 가르치려는 것이 아니기 때문에, 아마도 다소 내용이 상세하지 못하다고 불평할지도 모르겠다. 하지만, 이런 상세한 내용들은 차차 프로그래밍을 해가면서, 그리고 도움말을 찾아가면서 익히면 되는 것이므로 많은 연습을 통해 배우도록 하자.

다음 장에서는 오브젝트 파스칼의 가장 핵심적이고 기초적인 문법에 대해서 알아보도록 한다.

## 오브젝트 파스칼의 기초

### (Fundamentals of Object Pascal)

텔파이에서 사용된 오브젝트 파스칼은 표준 파스칼에 비해 많은 진보가 있는 언어이다. 파스칼은 최초에 top-down 디자인과 구조적 프로그래밍을 가르치기 위해 개발된 언어이다. 그렇기 때문에 가장 많은 수의 대학에서 프로그래밍 언어의 표준으로 이를 이용해 강의를 하곤 했다.

그러다가 볼랜드가 터보 파스칼을 발표하게 되는데, 이름 그대로 표준 파스칼에 IBM PC 에 적절한 각종 유닛과 라이브러리를 제공했으며 동시에 언어 자체의 능력도 향상시킨 진짜 ‘터보 파스칼’ 이었다. 사실 이해하기 어려운 C 코드에 비해 직관적이면서도 깨끗한 터보 파스칼은 당시에 상당한 반향을 일으키며 터보 C 와 함께 IBM PC 시장의 양대 언어로 자리잡았다.

그 이후, 윈도우 프로그래밍 환경이 나타나면서 우리나라에서는 볼랜드 C++, 비주얼 C++ 그리고 비주얼 베이직에 밀려서 터보 파스칼의 사용자 수가 많이 줄어들었다. 그렇지만, 터보 파스칼의 인기는 유럽에서는 선풍적이었는데 인터넷이나 뉴스 그룹을 통해 알아보면 아직도 무수한 터보 파스칼에 대한 정보를 얻을 수 있다.

파스칼이 객체지향형 프로그래밍 언어의 기능을 추가하게 된 것은 터보 파스칼 5.5 버전으로, 이때부터 조금씩 진보된 환경에의 변화를 시도하다가 급기야 텔파이 1.0 이 발표되면서 명실상부한 객체지향형 파스칼로서 세상에 모습을 드러내게 된다.

이번 장에서는 텔파이의 언어적 토대를 이루고 있는 오브젝트 파스칼의 기본적인 문법에 대해서 알아보도록 한다. 무릇 기초가 튼튼해야 큰 건물을 지을 수 있는 법이니, 파스칼에 대한 이해의 정도가 깊을수록 좋은 텔파이 프로그래머가 된다는 것은 자명한 일이다.

### 변수 (Variables)

파스칼에서는 변수를 사용하기 전에는 반드시 선언을 해 주어야 한다. 또한, 변수를 선언할 때에는 데이터 형을 반드시 지정해야 한다. 변수는 값은 프로그램의 수행 동안에 변경시킬 수 있으며 다양한 값을 가질 수 있다. 변수는 다음과 같이 선언한다.

var

Number: Integer;

YesNo: Boolean;

var 키워드는 변수를 선언하기 위해 사용한다. 위치에 따라 함수나 프로시저의 코드 시작

부분에서 지역적으로 사용되는 변수를 선언할 수도 있고, 유닛의 전역 변수를 선언할 수도 있다. 일단 변수를 선언하고 나면, 그 변수의 데이터 형이 지원하는 값을 대입할 수 있다. 간단한 대입문을 예를 들면 다음과 같다.

```
Number := 7;  
YesNo := True;
```

각 문장의 끝에 ‘;’을 사용하는 것에 주의한다. 파스칼은 라인이 바뀌어도 문장이 종료된 것으로 취급하지 않고 ‘;’이 나타나야 한 문장의 끝으로 생각한다.

- 절대 주소 (Absolute addresses)

특정 메모리 주소에 있는 변수를 선언할 때에는 absolute 키워드를 사용하여 다음과 같이 선언한다.

```
var  
  CrtMode: Byte absolute $0040;
```

이 방법은 평상시에는 잘 사용되지 않지만 디바이스 드라이버 등을 제작하는 등의 저수준 프로그래밍에 유용하다. 이미 존재하는 변수와 같은 주소에 새로운 변수를 생성시키고자 할 경우에는 주소를 직접 쓰지 않고, 다음과 같이 변수의 이름을 사용하여 주소를 대신하게 할 수 있다.

```
var  
  Str: string[32];  
  StrLen: Byte absolute Str;
```

StrLen 변수는 Str 변수와 같은 주소에서 시작된다. 여기서 Str 은 32 자 길이의 ShortString 형이다. ShortString 데이터 형의 첫번째 바이트에는 문자열의 길이가 저장되므로 StrLen 변수의 값은 곧 문자열의 길이를 나타내게 된다.

- 동적 변수 (Dynamic variables)

변수를 동적으로 생성할 때에는 GetMem 또는 New 프로시저를 이용해야 한다. 이들을 이용해서 변수를 생성하게 되면 힙에 메모리가 할당되고, 자동으로 관리가 되지 않으므로 개발자가 책임지고 이들을 다 쓰고 나서 제거해 주어야 한다.

GetMem 을 통해 생성된 변수는 FreeMem 프로시저로 해제하고, New 로 생성한 변수는 Dispose 프로시저로 해제한다. 참고로 이런 동적 변수를 다루기 위해 제공되는 표준 루틴으로는 ReallocMem, Initialize, StrAlloc, StrDispose 등이 있다. 자세한 내용은 도움말을 참고하기 바란다.

델파이 4 에서 새롭게 제공되는 동적 배열에 대해서는 이 장의 후반부에서 자세히 언급하겠지만, 힙에 할당되는 동적 변수의 일종이다. 또한 긴 문자열도 마찬가지이다. 그렇지만 이들은 자동으로 관리된다는 점이 다르다.

## 상수 (Constants)

상수는 프로그램에 사용하는 어떤 값에 대한 이름을 붙이는 것이다. 상수는 선언할 때에 const 라는 키워드를 사용하며, 선언 예는 다음과 같다.

```
const
    Pi = 3.14159;
```

이렇게 상수를 선언하는 목적은 숫자를 기억하기 쉽게 만들어주며, 다른 사람이 코드를 관리하는 데에도 도움을 준다.

참고로 델파이 3 부터는 리소스 문자열 상수를 지원한다. 다음과 같이 선언하면 되는데, 여기에서 정의된 문자열 상수는 프로그램의 리소스안에 저장된다.

```
resourcestring
    Name = 'JiHoon Jeong';
```

## 기본적인 데이터 형

파스칼에서 기본적인 데이터 형은 크게 나누어 서수형(ordinal type), 실수형(real type), 문자열형(string type)의 3 가지가 있다. 그 밖에 OLE 자동화를 위한 variant 형과 subrange, enumerate, set 등의 사용자 정의 데이터 형이 제공된다. 이러한 데이터 형을 선언할 때에는 기본적으로 type 키워드를 처음에 적어준다.

### ● 서수형 (Ordinal type)

서수형은 순서가 있는 데이터 형으로 이해하면 된다. 즉, 이 데이터 형의 값이 있으면 이 중에서 어느 쪽이 큰지 비교할 수도 있고, 그 값이 특정 값의 앞에 있는지 뒤에 있는지도



알 수 있으며, 최소값과 최대값을 알아낼 수도 있다.

서수형 중에서 가장 흔히 사용하는 것은 Integer, Boolean, Char 의 3 가지 데이터 형이다. 이 중에서도 정수형인 Integer 형은 다음과 같이 세분할 수 있다.

메모리 크기	Signed	Unsigned
8 비트	ShortInt	Byte
16 비트	SmallInt	Word
32 비트	LongInt	LongWord
16/32 비트 (1.0/2.0 이후 버전)	Integer	Cardinal
64 비트	Int64	

이 중에서 32 비트 unsigned 정수형인 LongWord, 64 비트 signed 정수형인 Int64 는 델파이 4 에서 새롭게 제공되는 데이터 형이다.

Boolean 데이터 형도 윈도우 API 함수에 사용하기 위해 ByteBool, WordBool, LongBool 등을 사용하기도 한다. 델파이 3 버전 부터는 ByteBool, WordBool, LongBool 데이터 형의 True 값을 -1, False 값을 0 으로 가리키게 하였다. 이것은 비주얼 베이직과 OLE 자동화의 데이터 형과의 호환을 위한 것이다. Boolean 형은 True 가 1, False 가 0 을 가리킨다.

문자형은 Char 에도 크게 나누어 ANSIChar, WideChar 의 2 가지 데이터 형이 있다. 전통적으로 사용한 문자형은 8 비트의 ANSI 문자 세트이며, 특별한 언급이 없는 한 Char 문자형은 ANSIChar 를 가리킨다. WideChar 는 유니코드 문자를 지원하기 위한 것으로 16 비트 문자 세트이다. OLE 자동화와 유니코드를 지원하기 위해서 사용한다.

이런 서수형에는 공통적으로 다음과 같은 루틴을 이용할 수 있다. 이들은 매우 유용하게 사용되는 것들이므로 익혀두는 것이 좋을 것이다.

루틴	설명	루틴	설명
Dec	파라미터로서 전달되는 변수를 하나씩, 또는 특정 값만큼 감소	Inc	파라미터로서 전달되는 변수를 하나씩 또는 특정 값만큼씩 증가
Pred	주어진 데이터 형에 해당되는 변수 앞의 값을 리턴한다.	Succ	주어진 데이터 형에 해당되는 변수 뒤의 값을 리턴한다.
Low	파라미터로서 전달되는 서수형의 범위 내의 가장 낮은 값을 리턴한다.	High	파라미터로서 전달되는 서수형의 범위 내에서 가장 높은 값을 리턴한다.
Ord	주어진 데이터 형의 값세트 안에 있는 인자들의 순서를 나타내는 수를 리턴함	Odd	변수가 홀수이면 True 를 반환한다.

# ● 실수형 (Real type)

실수 데이터 형은 소수 부분을 갖는 숫자 데이터를 저장할 수 있다. 정수형과 마찬가지로 실수형 역시 다음과 같이 여러 가지로 세분된다.

메모리 크기	데이터 형	범 위
32 비트	Single	1.5E-45 ~ 3.4E38
64 비트	Real/Double	5.0E-324 ~ 1.7E308
64 비트	Comp	1.0 ~ 9.2E18
64 비트	Currency	0.0001 ~ 9.2E14
80 비트	Extended	3.4E-4932 ~ 1.1E4932

텔파이 3 까지는 Real 데이터 형이 48 비트 였으나, 텔파이 4.0 에서는 Real 데이터 형이 Double 과 같은 64 비트로 변경되었다.

Comp 데이터 형은 실 수가 아니라 매우 큰 정수형이다. 그렇지만 이 데이터 형은 실수형 과 같은 방식으로 실행된다. Currency 데이터 형은 큰 숫자들을 저장할 때 매우 높은 정 확도를 가지고 있으며, 금액을 나타내는 데이터베이스 형식과 호환성이 있는 이점이 있다.

#### ● 문자열형 (String type)

텔파이에서는 문자열을 여러가지 방법으로 다룰 수 있다. 다음 표는 텔파이에서 사용할 수 있는 4 가지 문자열 형식을 보여주고 있다.

데이터 형	길 이	구성 문자	Null 종료
ShortString	255	ANSIChar	No
AnsiString	3GB	ANSIChar	Yes
String	255 ~ 3GB	ANSIChar	Yes or No
WideString	1.5GB	WideChar	Yes

String 데이터 형의 경우 기본적으로는 AnsiString 과 같지만, 컴파일러 옵션에 따라서는 ShortString 으로 간주될 수도 있다. 즉, {\$H-} 옵션을 주면 ShortString 으로 사용된다.

AnsiString 은 Null 에 의해 문자열이 종료되기 때문에 그 길이를 매우 동적으로 할당할 수 있다. 좀더 긴 문자열을 변수에 대입할 경우 텔파이에서는 문자열을 저장하기 위해 메모리 를 다시 할당하게 된다. Null 종료가 장점이 되는 이유로는 Win32 API 와 같은 대부분의 시스템 루틴의 호출에서는 C 와의 호환성을 위해 Null 종료 문자열을 사용하게 된다. Null 종료 문자열을 사용하게 되면 PChar 형으로 타입 캐스팅하여 직접 C 의 문자열 형과 호환 되게 할 수 있다. 텔파이의 긴 문자열 들은 참조 계수(reference counting) 기법에 기초하

고 있다. 다시 말해 메모리의 같은 문자열을 몇 개의 문자열 변수가 참조하고 있는지를 계속 추적하여, 문자열이 더 이상 쓰이지 않을 때 (참조 계수가 0) 메모리를 비우게 된다.

- WideString

WideString 데이터 형은 동적으로 할당되는 16 비트 유니코드 문자의 문자열이다.

WideString 데이터 형은 COM의 BSTR 데이터 형과 호환이 된다. 델파이는 COM을 지원하기 위해 AnsiString 값을 WideString 데이터 형으로 변환하는 루틴을 제공하고 있으며, COM API를 호출할 때에는 반드시 문자열을 WideString 데이터 형으로 명시적인 형변환을 해 주어야 한다.

윈도우는 현재 유니코드의 지원을 위해 기본적인 SBCS(single-byte character set)과 더불어 MBCS(multibyte character set)를 지원한다. SBCS에서는 각 바이트가 한 문자를 나타내며, ANSI 문자를 사용하는 대부분의 영어 문자권의 나라들이 이를 사용하게 된다. MBCS는 어떤 문자는 한 바이트로 표현하고, 어떤 문자는 한 바이트 이상으로 표현한다. 보통 아시아권 언어의 경우 2 바이트를 사용하기 때문에 흔히 DBCS로 표현한다.

유니코드 문자 세트에서는 모든 문자들이 2 바이트로 표현된다. 이러한 유니코드 문자는 델파이의 WideChar 데이터 형과 호환되며, 유니코드 문자열은 WideString 데이터 형과 호환된다.

- Null 종료 문자열의 이해

Null 종료 문자열은 Null(#0) 값으로 끝나는 문자의 배열이다. 이 배열에는 길이를 나타내는 부분이 없기 때문에, 처음으로 나타나는 Null 문자를 문자열의 끝으로 간주하는 것이다. 예를 들어, 다음의 type 선언문이 Null 종료 문자열을 저장하기 위해 사용될 수 있다.

type

```
TIdentifier = array[0..15] of Char;  
TFileName = array[0..255] of Char;  
TMemoText = array[0..1023] of WideChar;
```

이렇게 선언한 문자 배열에 기본적으로 Null 문자를 채워넣고, ShortString 문자의 값을 대입하는 식으로 문자를 변환하는 루틴을 곳곳에서 발견할 수 있을 것이다.

- 파스칼 문자열과 Null 종료 문자열의 혼용

긴 문자열인 AnsiString 값과 Null 종료 문자열인 PChar 값을 쉽게 섞어서 사용할 수 있다.

PChar 값은 긴 문자열 변수에 직접 대입이 가능하다. 즉, S 가 AnsiString 이고 P 가 PChar 형이라면 S := P 와 같은 표현식이 사용될 수 있다.

그렇지만, 경우에 따라서는 문자열의 형변환이 필요한 경우가 있다. 예를 들어, 2 개의 PChar 문자열을 붙여서 하나의 AnsiString 으로 저장하려면 다음과 같이 사용하여야 한다.

```
S := string(P1) + string(P2);
```

긴 문자열을 Null 종료 문자열로 변환시킬 때에도 마찬가지로 형태의 형변환을 해주면 된다. 예를 들어 Str1, Str2 가 AnsiString 문자열일 때 파라미터로 Null 종료 문자열을 요구하는 MessageBox API 함수는 다음과 같이 호출한다.

```
MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
```

WideString 과 PWideChar 데이터 형의 관계 역시 AnsiString 과 PChar 의 관계와 동일하므로 동일한 규칙을 따르면 된다.

## 오브젝트 파스칼의 사용자 정의 데이터 형

표준 파스칼은 subrange, enumerated, set 등의 사용자 정의 데이터 형을 제공한다. 이들을 효과적으로 이용하면 유연한 데이터 구조를 만들어낼 수 있으며, 이해하기 쉬운 코드를 작성할 수 있다. 이 밖에도 오브젝트 파스칼에서는 배열형, 레코드 형, 포인터 형, 프로시저 형 등도 지원되는데, 이들은 이 장의 후반부에 더 자세하게 다룬다.

### ● Subrange 형

Subrange 데이터 형은 이름이 암시하듯이 사용자가 범위를 한정해서 사용할 수 있는 데이터 형이다. 정의는 매우 간단하게 가장 작은 값과 가장 큰 값을 양쪽 끝에 쓰고 이들 사이에 '..'을 사용하면 된다. 간단히 예를 들어보자.

```
type
```

```
  TNibble = 0..15;
```

여기에서 TNibble 데이터 형은 0~15 까지의 정수를 저장할 수 있다. 이 범위를 벗어난 값을 대입하려고 하면 에러를 발생시킨다. 여기에서 범위로 지정할 수 있는 것은 0 을 넘어서는 값이나, 문자도 가능하다. 즉, 다음과 같은 코드의 사용이 가능하다.

type

```
THiNibble = 16..255;
```

```
TNumericChar = '0'..'9';
```

Subrange 데이터 형은 특정 범위를 제한해야 하는 변수 등을 사용할 때나 디버깅을 할 때 아주 요긴하게 쓸 수 있는 데이터 형이다.

앞에서도 언급했듯이 범위를 벗어난 값을 대입하려 하면 에러를 발생시키는데, 이때 컴파일러의 Range checking 여부에 따라서 컴파일 또는 런타임에서 에러가 발생한다. 만약에 Range checking 이 가능한 것으로 설정되어 있으면 ({R+ }), 컴파일 할 때와 런타임에서 모두 에러를 발생시킨다. 그런데, Range checking 이 가능한 것으로 설정된 경우에는 코드에 이를 위한 부분이 추가되기 때문에 다소나마 프로그램의 수행 성능을 저하시키므로, 디버깅이 충분히 된 이후에는 이 옵션을 끄고 최종적으로 컴파일해서 배포하는 것이 좋다.

#### ● 열거(Enumerated) 형

열거형은 아마도 VCL 소스 코드에서 가장 빈번하게 찾아볼 수 있는 데이터 형일 것이다. 간단히 설명하면 괄호에 의해 둘러싸이고, 콤마를 통해 구분된 기호화된 서수(ordinal) 값이라고 생각하면 된다. 말보다 눈으로 보는 것이 더욱 알기 쉬울 것이므로 다음의 코드를 살펴 보자.

type

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

열거형으로 선언된 변수는 이들 중의 하나의 값을 대입하게 된다. 예를 들어 보면,

var

```
DOW: TDayOfWeek;
```

begin

```
DOW = Sunday;
```

end;

열거형을 이용한 VCL 소스 코드의 예를 들면, 다음과 같은 것들이 있다.

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields,
```

```
dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);
```

```
TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
```

```
TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit,  
    nbPost, nbCancel, nbRefresh);
```

열거형을 이용한 프로퍼티의 경우 오브젝트 인스펙터에서 드롭-다운 리스트에서 선택할 수 있다. 열거형은 또한 subrange 데이터 형과 같이 사용하면 매우 유용하다. 예를 들어, 앞에서 선언한 TDayOfWeek 열거형을 이용해서 다음과 같이 사용할 수 있다.

```
const  
    WorkDays = array[Monday..Friday] of string[3] = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');
```

열거형은 사실상 0 에서부터 시작하는 서수형 값의 기호형태라고 생각하면 된다. 즉, TDayOfWeek 는 다음 코드와 같은 의미이다.

```
type  
    TDayOfWeek = (0, 1, 2, 3, 4, 5, 6);
```

이렇게 열거형을 사용하는 이유는 코드가 읽기 편해지고, 여러모로 헛갈리기 쉬운 정수를 무차별 적으로 나열하기 보다는 이를 사용하면 훨씬 안전하고, 효율적이기 때문이다. 표준 ANSI C/C++ 에는 파스칼의 열거형에 해당하는 enum 형이 존재한다.

## ● 세트(Set) 형

세트형은 파스칼에서 가장 유용한 데이터 형이라고 말해도 과언이 아닌 중요한 데이터 형이다. 세트형을 간단하게 말하자면 값들의 목록 또는 집합이라고 설명할 수 있다. 세트를 구성하는 값들을 요소(element) 또는 세트 멤버라고 한다. 그러면, 실제로 세트를 정의하는 방법을 알아보자.

```
type  
    TYesNo = set of Char;  
var  
    YN: TYesNo;  
begin  
    YN := ['Y', 'y', 'N', 'n'];  
end;
```

그다지 어려운 방법은 아니다.

사실 세트 데이터 형의 가장 큰 장점은 여러가지 연산이 가능하다는 것이다. 중학교 수학에서 배우는 각종 집합 연산을 적용할 수 있으며, 이것이 아주 유용하게 쓰일 수 있다. 대표적인 연산으로는 합(union), 곱(intersection), 차(difference), 등호(equivalence), 부분 집합(membership) 등이 있다. 이해하기 어렵지는 않다고 생각되지만, 직관적으로 알아보기 위해 다음 표를 살펴보자.

연산자	의 미	표현식	결 과
+	Set union	[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
*	Set intersection	['A', 'B', 'C'] * ['B', 'C', 'D']	['B', 'C']
-	Set difference	['A', 'B', 'C'] - ['B', 'C', 'D']	['A']
=	Set equivalence	['A', 'B', 'C'] = ['A', 'B', 'C']	True
in	Set membership	'A' in ['A', 'B', 'C']	True

참고로, 세트 연산을 위해서는 같은 세트의 데이터 형은 같아야 한다. 예를 들어 다음의 표현식은 부당하다.

['Y', 'y', 'N', 'n'] + [1, 0]

세트 역시 열거형과 마찬가지로 subrange 데이터 형을 적절히 사용하면 편리하게 쓸 수 있다. 즉, 다음과 같은 선언이 가능하다.

['A'..'G']

['0'..'9', 'a'..'z', '#', '\*', '-', '+', '/']

숫자에 대해서도 마찬가지로 적용할 수 있다. 세트의 연산을 이용하면 더욱 다양하게 세트의 멤버를 구성할 수 있다. 예를 들어 1 부터 30 까지의 수를 세트로 가지되 5, 15, 25 를 제외하고 싶다면 다음과 같이 정의하면 된다.

[1..30] - [5, 15, 25]

이를 앞에서 보여준 subrange 데이터 형을 이용해서 표현하면 다음과 같다.

[1..4, 6..14, 16..24, 26..30]

세트의 연산식 중에서 가장 유용한 것을 꼽으라면 필자는 'in' 을 꼽을 것이다. 판단문에서 특히 유용하게 사용할 수 있는데, 다음의 예를 살펴보자.

```
if (Number >=10) and (Number <= 100) then DoSomething;
```

```
if (Number = 1) or (Number = 5) or (Number = 9) or (Number = 10) then DoSomething;
```

이것은 다음의 코드들과 같은 의미를 가진다.

```
if (Number in [10..100]) then DoSomething;
```

```
if (Number in [1, 5, 9, 10]) then DoSomething;
```

세트 형의 구현은 생각보다 간단하다. 즉, 각각의 요소마다 한 비트 씩을 차지하여 그 비트가 1 일 경우 해당되는 요소가 포함되며, 0 일 경우 제외된다. 비트간 연산을 통해 합, 곱, 차 등이 계산될 수 있는 것이다. 그러므로, 세트 연산은 대단히 빠르게 동작한다.

세트와 열거형, subrange 데이터 형은 같이 많이 사용되며 VCL 소스 코드를 살펴 보면 무수히 많은 예를 볼 수 있다. 보통 세트형의 경우 열거형으로 선언된 데이터 형의 이름의 제일 뒤에 's'를 붙인 이름을 많이 붙이게 된다.

예를 구체적으로 들자면, 앞에서 열거형으로 선언된 TBorderIcon 데이터 형의 세트형은 TBorderIcons 로 선언되어 있다. VCL 코드를 살펴보면 간단하게 다음과 같이 정의되어 있다.

```
TBorderIcons = set of TBorderIcon;
```

TBorderIcons 를 실제로 사용하는 예를 들어보도록 하자. 폼의 최대화, 최소화 상태를 검사하기 위해서는 다음과 같이 하면 된다.

```
if ((BorderIcons * [biMaximize, biMinimize]) = [biMaximize, biMinimize]) then DoSomething;
```

다소 복잡해 보이지만 최대화, 최소화된 상태가 biSystemMenu 일 경우를 포함하기 위해서 집합곱을 이용하였다. 즉, [biSystemMenu, biMaximize, biMinimize]인 경우도 포함한다.

## 배열 (Arrays)

배열은 base type 이라고 하는 같은 데이터 형의 요소 들로 이루어진 색인된 컬렉션이다. 배열은 정적인 배열과 동적 배열이 있다. 동적 배열형은 델파이 4 에서부터 지원되는 새로운 데이터 형이다.

- 정적 배열 (Static arrays)



정적 배열형은 다음과 같이 선언한다.

```
array[indexType1, ..., indexTypen] of baseType
```

여기서 인덱스로 사용되는 indexType 은 2GB 범위를 넘지 않는다면 어떤 서수형(ordinal type)도 사용할 수 있다. 보통은 정수의 subrange 를 이용한다. 1 차원 배열의 간단한 선언 예는 다음과 같다.

```
var
```

```
MyArray: array[1..100] of Char;
```

이렇게 하면, MyArray 는 100 자의 문자를 담게 되는 배열 변수가 된다. 정적 배열을 생성해도 값을 대입하지 않으면, 메모리는 할당되어 있지만, 그 값은 임의의 값이 들어 있는 초기화가 되지 않은 변수가 된다.

다차원 배열의 선언은 다음과 같이 한다.

```
type
```

```
TMatrix = array[1..10] of array[1..50] of Real;
```

또는,

```
type
```

```
TMatrix = array[1..10, 1..50] of Real;
```

어떤 방법으로든 TMatrix 가 선언되면, 이 배열 변수는 실제로 500 개의 실수값을 담을 수 있게 된다.

배열의 값을 검사하는 데에는 앞에서 잠시 언급한 서수형 처리 루틴인 Low, High 가 유용하게 사용된다. 이를 이용하면 배열의 첫번째 요소에서 마지막 요소까지 검사하는 루틴을 손쉽게 작성할 수 있다.

#### ● 동적 배열 (Dynamic arrays)

델파이 4 에서는 동적 배열의 선언이 가능해 졌다. 동적 배열은 타입 정보(배열의 차원과 요소의 데이터 형)를 지정하되, 요소의 수를 지정하지 않는다. 즉, 다음과 같이 선언하면 된다.

var

A: array of integer;

B: array of array of string;

A 는 정수형의 1 차원 배열이며, B 는 문자열의 2 차원 배열이다.

동적 배열은 배열에 새로운 값을 대입하거나 SetLength 프로시저를 호출하면 해당되는 메모리를 재할당한다. 앞서와 같이 선언만 한 경우에는 그에 해당되는 메모리가 할당되지 않는다. 동적 배열을 메모리에 생성하려면 SetLength 프로시저를 호출한다. 예를 들어 앞서의 A 배열의 경우 다음과 같이 선언해보자.

SetLength(A, 20);

이 문장을 통해 20 개의 정수를 저장할 수 있는 배열이 생성된다. 동적 배열은 언제나 0 부터 시작하는 정수 인덱스를 가진다.

동적 배열 변수는 내부적으로는 포인터이며, 델파이 2 에서부터 지원되는 문자열 변수 처럼 참조 계수 테크닉을 이용하여 관리된다. 동적 배열을 해제하려면 변수에 nil 을 대입하거나 Finalize 메소드를 호출한다.

만약에 X, Y 가 같은 동적 배열 형을 가진 변수이면  $X := Y$  와 같은 표현식을 이용하여 배열 Y 의 크기와 같도록 배열 X 의 크기가 결정되며, X 는 Y 와 같은 배열을 가리키게 된다. 문자열과는 달리 배열은 복사된 후의 다른 변수의 변경 값이 원본 변수에 반영된다. 다음의 코드를 살펴보자.

var

A, B: array of Integer;

begin

SetLength(A, 1);

A[0] := 1;

B := A;

B[0] := 2;

end;

이 코드를 실행하면 A[0]의 값이 2 로 설정된다.

동적 배열 변수가 비교될 때에는 참조값이 비교되는 것이지, 배열의 값들이 비교되지 않는다. 다음의 코드를 살펴보자.

```

var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;

```

여기에서  $A = B$  는 False 이지만,  $A[0] = B[0]$  는 True 를 반환한다.

Copy 함수를 이용하면 동적 배열을 잘라낼 수 있다. 예를 들어,  $A := \text{Copy}(A, 0, 20)$  이라는 표현식은 동적 배열 A 의 처음 20 개의 요소만을 남겨두고 나머지는 제거한다.

일단 동적 배열이 할당되면 Length, High, Low 등의 표준 함수를 이용할 수 있다. Length 는 배열의 요소 수를 반환하며, High 와 Low 는 각각 동적 배열의 가장 큰 인덱스와 0 을 반환한다.

다차원 동적 배열의 선언 후 이를 사용하는 방법도 일차원 동적 배열과 마찬가지로이다.

예를 들어 다음과 같이 다차원 배열을 선언해 보자.

```

type
  TMessageGrid = array of array of string;
var
  Msgs: TMessageGrid;

```

이 동적 배열을 인스턴스화 하려면, 다음과 같이 하면 된다.

```
SetLength(Msgs, I, J);
```

이 코드에서  $I \times J$  크기의 2 차원 배열이 할당된다. 또한, 이렇게 SetLength 를 이용하여 다차원 배열을 선언한 후 할당할 때 다양한 테크닉을 사용할 수 있다. 다음의 코드를 살펴보자.

```

var
  Ints: array of array of Integer;
SetLength(Ints, 10);

```

이 코드에 의해 Ints 에 10 개의 행이 할당되지만 열은 없다. 이제 한번에 하나씩의 열에

대해 배열의 할당이 가능하다. 예를 들어, 다음의 코드를 보자.

```
SetLength(Ints[2], 5);
```

이 코드는 3 번째 행에 5 개의 요소가 할당된다. 이제 이 배열에 값을 대입할 때에는 `Ints[2, 4] := 6` 과 같은 표현식을 이용하면 된다.

다음 예제는 `IntToStr` 함수와 동적 배열을 이용해서 문자열의 삼각형 모양의 매트릭스로 생성한다.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
    begin
      SetLength(A[I], I);
      for J := Low(A[I]) to High(A[I]) do
        A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
      end;
    end;
  end;
```

## 레코드 형 (Record type)

레코드는 C 에서의 구조체(structure)와 비슷한 역할을 하는 것으로, 여러가지 데이터 요소의 집합을 대표할 수 있는 데이터 형이다. 각 요소를 필드라고 하며, 레코드 형의 선언은 각 필드의 이름과 데이터 형을 지정한다. 그러면 실제 선언부분을 살펴 보자.

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

TDateRec 레코드는 3 개의 필드로 구성되는데 각 필드는 정수형인 Year, 열거형인 Month, 1~31 까지의 subrange 형인 Day 이다. 이들은 마치 다른 변수처럼 접근해서 사용하면 된다. 그러면, 이런 레코드 형 변수를 선언하고 사용하는 방법을 알아보자.

```
var  
    Record1, Record2: TDateRec;
```

```
Record1.Year := 1904;  
Record1.Month := Jun;  
Record1.Day := 16;
```

보통, 레코드 형과 같이 중복되는 부분이 있는 경우에는 다음과 같이 with 문을 사용하면 유용하다.

```
with Record1 do  
begin  
    Year := 1904;  
    Month := Jun;  
    Day := 16;  
end;
```

레코드 형에는 이런 표준적인 데이터 형 이외에도 가변 파트를 포함한 필드를 사용할 수 있다. 레코드 형에 가변 파트를 추가하고 사용할 때에는 다음과 같은 문법을 따른다.

```
type  
    recordTypeName = record  
        fieldList1: type1;  
        ...  
        fieldListn: typen;  
  
    case tag: ordinalType of  
        constantList1: (variant1);  
        ...  
        constantListn: (variantn);  
    end;
```

기본적인 선언 부분은 표준 레코드 형과 동일하다. 그런데, 특기할 것은 case 문을 이용해서 가변 파트를 지정하는 부분이다.

여기서의 tag 는 옵션으로 사용할 수 있는 identifier 로 유효한 identifier 는 어느 것이나 사용할 수 있다. 이것이 빠지면, ‘:’도 같이 빼 주어야 한다. ordinalType 은 서수형이면 어느 것이나 사용할 수 있다는 의미이다.

여기에 대해 자세히 설명하는 이유는 텔파이의 VCL 소스 코드를 보면 언뜻 처음 봐서는 도저히 이해가 가지 않는 코드들이 있는데, 이들 중 가변 파트를 이용한 레코드인 경우가 많다. 필자도 텔파이 사용자 모임에 갔을 때, 이런 소스 코드에 대해서 질문을 많이 받았는데 문법적으로 제대로 설명된 적이 없어서 다소 이해가 힘들었던 기억이 있다.

각각의 필드 리스트에는 여러가지 데이터 형을 사용할 수 있지만, 긴 문자열이나 동적 배열, 가변형, 인터페이스나 이들이 포함된 레코드 형은 사용하면 안된다. 그렇지만, 이런 데이터 형이 필요한 경우에는 포인터를 이용하면 된다.

가변 파트를 포함한 레코드는 문법적으로 사용하기가 다소 복잡하지만, 상당히 유용한 데이터 형이다. 이런 레코드에는 같은 메모리 공간을 공유하는 여러 종류의 가변 데이터를 포함할 수 있다. 즉, 데이터의 종류가 다르지만 이를 모두 독립된 필드로 구성하기 싫은 경우에 사용할 수 있다.

그러면, 실제로 사용되는 예를 살펴보자.

type

```
TEmployee = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Salaried: Boolean of
    True: (AnnualSalary: Currency);
    False: (HourlyWage: Currency);
end;
```

여기에서 앞에서도 잠시 설명했지만, ‘case Salaried: Boolean of’ 문장은 ‘case Boolean of’와 같은 의미가 된다. 아마도 텔파이 소스에서 ‘case Integer of’와 같은 문장을 보면서 어떤 의미인지 몰랐던 사람들은 같은 의미로 생각하면 된다. Integer, Boolean 데이터 형이 둘 다 일종의 서수형이기 때문에 사용에 무리가 없다.

앞의 코드에서 모든 employee 는 보수를 지급받을 때, 어떤 경우에는 연봉으로 받을 수도 있고, 시간당으로 받을 수도 있을 것이다. 그렇지만, 이 두 가지를 동시에 사용하는 경우는 없다. 참고로 여기서는 2 가지 경우를 생각했기 때문에 Boolean 을 사용했지만, 월급과 같은 경우를 추가한다면 Integer 등을 사용해야 할 것이다. 이런 경우에 TEmployee 레코드 형의 인스턴스를 생성할 때 두 필드를 모두 포함시키는 것은 다소 낭비이다. 그러므로, 가

변 파트를 추가해서, 서로 다른 데이터 형을 하나의 필드에 사용하게 하면 효율적이다.  
조금 더 복잡한 예를 알아 보자.

```
TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);  
TFigure = record  
    case TShapeList of  
        Rectangle: (Height, Width: Real);  
        Triangle: (Side1, Side2, Angle: Real);  
        Circle: (Radius: Real);  
        Ellipse, Other: ();  
    end;
```

이와 같이 미리 열거형을 이용해서 정의한 경우에는 보다 편리하게 사용할 수 있다. 각각의 레코드 인스턴스에 대해 컴파일러가 가장 큰 데이터 형을 기준으로 충분한 메모리를 할당하게 되고, Rectangle, Triangle 과 같은 상수 들은 사실상 컴파일러가 필드를 관리하는데 아무런 영향을 미치지 않는다는. 다만, 이렇게 선언할 경우 프로그래머가 코드를 읽거나 관리하는데 도움이 된다.

가변 파트를 이용하는 또 하나의 예로는, 컴파일러가 형변환을 허용하지 않는 경우 서로 다른 데이터 형의 변환 효과를 노릴 수 있다는 것이다. 예를 들어, 64 비트 실수와 32 비트 정수를 하나의 필드에 선언했을 때, 이들 간의 자연스런 변환이 가능하다.

가변 파트를 이용한 레코드 형의 가장 대표적이 사용 예라고 할 수 있는 TMessage 의 선언부를 살펴보자

```
TMessage = record  
    Msg: Cardinal;  
    case Integer of  
        0: (WParam: Longint; LParam: Longint; Result: Longint);  
        1: (WParamLo: Word; WParamHi: Word; LParamLo: Word; LParamHi: Word;  
            ResultLo: Word; ResultHi: Word);  
    end;
```

즉, 여기서는 WParam, LParam, Result 로 접근해도 되고 WParamLo, WParamHi 와 같이 접근해도 된다. 그렇지만, 기본적으로 LongInt 데이터 형이 Word 데이터 형 2 개와 크기가 같기 때문에 결국에는 어떻게 접근해도 데이터의 일관성이 유지된다. 아마도 TMessage 의 경우를 살펴 보면 가변 파트를 포함한 레코드 형의 장점을 이해할 수 있을 것이다.

## 포인터 형 (Pointer type)

포인터는 메모리 주소를 가리키는 것이다. 포인터가 특정 변수나 데이터의 주소를 가지고 있을 때에, 이를 가리켜 포인터가 변수나 데이터를 참조(reference)하고 있다고 한다. 간단한 용어 정의 같지만, 이 개념은 무척 중요한 것이므로 꼭 알아두기 바란다. 만약, 포인터가 가리키는 것이 레코드나 배열 처럼 여러 개의 요소로 이루어진 경우라면, 포인터는 첫번째 요소의 주소를 참조한다.

포인터가 특정 데이터 형을 가리키는 경우에는 이를 typed 포인터라고 하며, 여러가지 데이터 형을 모두 가질 수 있는 포인터 변수를 untyped 포인터 변수라고 한다.

다음의 코드를 살펴 보자/

```
var
  X, Y: Integer;
  P: ^Integer;
begin
  X := 17;
  P := @X;
  Y := P^;
end;
```

여기에서 P는 정수값에 대한 포인터로 선언된다. 이는 P 변수가 X나 Y 변수의 위치를 가리킬 수 있다는 것을 의미한다. 실제로 포인터인 P에 X의 주소를 대입하였으며, 그 다음 줄에서 P의 값을 Y에 대입하였다. 그러므로, 결과적으로 이 코드는 X의 값을 Y에 대입하게 되므로 X, Y 모두 17을 값으로 가지게 된다.

여기에서 ‘@’ 연산자는 변수나 함수, 프로시저의 주소를 나타내는데 사용되며, ‘^’ 기호는 데이터 형 identifier 앞에 사용될 경우에는 그 데이터 형에 대한 포인터임을 나타내고, 포인터 변수의 뒤에 쓰일 때에는 그 주소에 들어 있는 값을 나타내게 된다.

텔파이에서 포인터에 대해 잘 익혀 두어야 하는 이유는 다음과 같다.

1. 포인터에 대한 이해는 오브젝트 파스칼을 이해하는데 큰 도움이 된다. 이는 오브젝트 파스칼에서 명시적으로는 보이지 않지만, 내부적으로 작동하는 핵심적인 부분이다.
2. 데이터 형이 크고, 동적으로 할당될 필요가 있을 때에는 포인터를 사용한다. 또한, 긴 문자열의 경우 내부적으로 포인터에 의해 동작하며, 클래스 변수 들도 본질적으로 포인터이다.
3. 포인터는 오브젝트 파스칼의 엄격한 데이터 형 검사를 비켜갈 수 있는 해결책을 제시한



다. 예를 들어 다음의 코드를 살펴보자.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ...
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

이런 코드에 의해 R 에 저장된 실수형을 정수형 변수인 I 에 저장하는 것이 가능하다. 물론, 실수와 정수는 다른 형태로 저장된다. 그러므로, 이런 형태의 복사는 단순히 이진 데이터를 전달하는 것이지, 적절한 형태의 형변환은 아니다.

포인터 변수가 아무 것도 참조하지 않고 있을 때에는 nil 이라는 예약된 상수를 사용한다.

포인터를 선언할 때에 기본적인 문법은 데이터 형 앞에 ‘^’ 기호를 사용하면 된다. 포인터를 가장 유용하게 사용하는 예는 역시 레코드와 같이 다소 메모리를 차지하는 데이터 형에 포인터를 선언하여, 이들 데이터 블록이 포함된 메모리를 직접 처리하지 않고 포인터를 이용해 처리하는 것이다.

또한, 포인터 중에서는 어떤 데이터 형도 참조할 수 있는 untyped 포인터 형도 존재하는데, 이것은 단순히 Pointer 로 선언하며 직접 값을 알아내지는 못하고 일단 특정 데이터 형의 포인터로 형변환한 후에 ‘^’ 기호를 사용하여 값을 알아낸다.

## 프로시저 형 (Procedural types)

프로시저 형은 프로시저나 함수를 변수나 다른 프로시저나 함수에 파라미터로 전달할 수 있게 해준다. 예를 들어, 다음과 같이 두개의 정수 파라미터를 받아서 정수값을 반환하는 Calc 라는 함수가 있다고 하자.

```
function Calc(X, Y: Integer): Integer;
```

Calc 함수를 F 라는 변수에 다음과 같이 지정할 수 있다.

var

```
F: function(X, Y: Integer): Integer;
```

F := Calc;

이때, 함수나 프로시저의 파라미터와 리턴 값에 대한 것을 지정한 부분이 바로 프로시저 형이다. 앞의 코드와 같이 직접 변수 선언 부분에 사용할 수도 있고, 새로운 type 으로 선언하고 이를 사용하는 경우가 있는데 후자의 경우가 더 자주 쓰인다.

다음의 코드를 살펴보자.

type

```
TMethod = procedure of object;
```

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

이런 데이터 형을 메소드 포인터라고 한다. 메소드 포인터는 메소드의 주소와 메소드가 속해있는 객체의 참조값(reference)을 저장한다. 메소드 포인터는 다음과 같이 이벤트 선언에서 매우 자주 사용된다.

type

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

```
TMainForm = class(TForm)
```

```
    procedure ButtonClick(Sender: TObject);
```

```
    ...
```

```
end;
```

var

```
MainForm: TMainForm;
```

```
OnClick: TNotifyEvent
```

텔과이로 프로그래밍 하다 보면 자주 있게 되는 일인데, 다음의 대입문으로 버튼의 OnClick 이벤트 핸들러를 OnClick 이라는 메소드 포인터 변수에 저장할 수 있다.

```
OnClick := MainForm.ButtonClick;
```

프로시저 형이 서로 호환되는 것은 호출 규칙(calling convention)이 같고, 리턴 값과 파라

미터의 수와 데이터 형이 같으면 된다. 파라미터의 이름은 상관 없다.

참고로, 전역 프로시저 포인터 형은 메소드 포인터 형과는 호환되지 않는다는 것을 알아두기 바란다. 그리고, 프로시저 형 변수에는 nil 을 대입할 수 있다.

대입문에서 프로시저 형의 경우라면 같은 프로시저 형이 아니라 리턴 값의 데이터 형과 같은 변수에 대입하는 문장이 있을 수 있다. 이럴 때에는 함수의 연산 결과가 대입된다. 다음의 코드를 살펴 보자.

```
var
  F, G: function: Integer;
  I: Integer;
function SomeFunction: Integer;
...
F := SomeFunction:           //SomeFunction 함수를 F 에 대입
G := F;                      //F 를 G 에 복사
I := G;                      //함수를 실행한 결과 값을 I 에 대입
```

처음 F, G 에 대한 대입문은 프로시저 형의 값을 대입하는 것이지만, I 에 대한 대입문은 함수의 실행 결과인 정수값을 대입하게 된다.

프로시저 형을 사용할 때에 판단문의 비교 대상을 구별하는 것도 혼돈하기 쉬운 부분이다.

```
if F = MyFunction then ...;
```

이 문장에서 F 와 MyFunction 이 모두 함수 이름이라고 할 때 이 판단문의 결과는 F 함수 호출의 결과값과 MyFunction 함수 호출의 결과값을 서로 비교하는 것이다. 프로시저 형의 변수가 표현식에 있을 경우에 참조되고 있는 프로시저나 함수를 호출하는 것이 규칙이다. 만약에 앞의 문장에서 F 가 리턴 값이 없는 프로시저를 참조하고 있거나, 파라미터를 필요로 하는 함수를 참조하는 경우에는 컴파일 에러가 발생한다. 그러므로, 프로시저 값을 서로 비교할 때에는 '@' 연산자를 사용하여 다음과 같이 비교해야 한다.

```
if @F = @MyFunction then ...;
```

@F 는 F 를 untyped 포인터 변수로 변환하게 되므로 주소를 가지게 되며, @MyFunction 은 MyFunction 의 주소를 반환한다. 그러므로, 프로시저 변수 자체의 메모리 주소를 얻고자 할 때에는 @@F 와 같이 사용해야 한다.

프로시저 변수의 값은 nil 일 수도 있는데, 이 경우에는 아무것도 참조하지 않게 된다. 그렇지만 nil 포인터를 직접 사용하면 에러가 발생하기 때문에, 변수의 값이 할당되어 있는지 알

아볼 필요가 있는데 이럴 때 사용되는 것이 Assigned 함수이다. 이 함수는 다음과 같이 주로 컴포넌트에서 이벤트 핸들러가 있는지 확인할 때 많이 사용된다. :

```
if Assigned(OnClick) then OnClick(X);
```

## 가변형 (Variant types)

어쩔 때에는 여러가지 데이터 형을 다룰 필요가 있다. 그런데, 컴파일하기 전에 이런 변수를 결정하기 어려울 때에는 가변형 변수를 사용한다. 가변형은 다소 처리 속도가 느리고, 메모리를 많이 사용하지만 유연한 처리를 반드시 필요로 할 때에는 유용한 해결 방안이 되기도 한다.

가변형에 담을 수 없는 데이터 형은 레코드, 세트, 정적 배열, 파일, 클래스, 클래스 레퍼런스와 포인터이다. 즉, 가변형 변수에는 포인터와 구조체의 형태를 가지는 데이터 형을 제외한 다른 데이터 형은 모두 담을 수 있다. 가변형 변수가 중요하게 여겨지는 또 하나의 목적은 COM 객체를 이용할 수 있다는 것이다.

모든 가변형 변수는 초기화될 때 Unassigned 라는 특별한 값으로 초기화 된다. 그리고, Null 이라는 값을 가질 수 있는데 이 값은 데이터가 빠져 있거나, 알 수 없는 경우이다.

정적 배열을 가변형 데이터로 활용하려면 가변형 배열을 선언해서 사용하여야 한다. 이럴 때에는 VarArrayCreate, VarArrayOf 와 같은 표준 함수를 사용한다. 다음의 코드를 살펴보자.

```
V: Variant;
```

```
...
```

```
V := VarArrayCreate([0,9], varInteger);
```

이 코드의 결과로 10 개의 요소를 가지는 정수형의 배열이 생성되며, 이 값이 가변형 변수인 V 에 저장된다. 이렇게 가변형 배열로 선언된 가변형 변수는 인덱스를 사용해서 V[0], V[1]과 같은 형태로 배열의 요소에 접근할 수 있다. VarArrayCreate 함수의 두번째 파라미터에는 배열의 데이터 형을 지정한다. varInteger 는 정수형을 나타내며, 그 밖에 여러가지 데이터 형 코드를 사용할 수 있지만 varString 은 사용할 수 없다. 문자열의 배열을 이용해야 한다면 varOleStr 코드를 사용한다.

가변형 배열의 기초 데이터 형을 가변형으로 설정하면 더욱 유연하게 사용할 수도 있다. 가변형에 관련된 함수는 상당히 많고, 도움말도 비교적 자세하므로 더 깊은 내용에 대해서는 이를 참고하기 바란다.

가변형에서 델파이 4 에 새롭게 추가된 데이터 형이 있는데, Any 라는 데이터 형이 그것이다. 이 데이터 형은 CORBA 와 호환되는 가변형 변수로 사용된다. COM 에 의해 사용되는

OleVariant 와 비슷한 목적으로 사용된다고 이해하면 된다.

## 연산자 (Operator)

텔파이에서는 비교와 평가 등에 연산자를 사용하게 된다. 연산자에는 산술 연산자와 논리 연산자, 관계 연산자가 있다. 이들 각각에 대한 설명은 도움말을 참고하기 바라며, 각 연산자의 우선 순위에 대해서 알아보자. 같은 순위에 있는 하나 이상의 연산자를 함께 사용할 때에는 기본적으로 왼쪽에서 오른쪽으로 연산을 수행해 나간다. 연산 순위를 높은 곳에서 낮은 곳을 정리해보면 다음과 같다.

연산자	설 명
. ^	필드, 포인터 dereferencing
@ not	단일 연산자
* / div mod as and shl shr	곱셈, 형변환 관련 연산자
+ - or xor	덧셈 관련 연산자
= < > <= >= in is	관계형 연산자

2 개의 연산자들 사이의 피연산자는 높은 우선 순위의 연산자에 우선적으로 적용된다. 동등 연산자들 사이의 피연산자는 왼쪽에 있는 것을 먼저 적용하며, 괄호 안의 연산식은 하나의 피연산자로 다루기 때문에 모든 연산식에 우선하게 된다.

## 언어 문장 (Statements)

오브젝트 파스칼의 문장(statement)은 개발자의 표현식을 실제로 판단하고 실행하는 기본 단위가 된다. 문장의 종류를 크게 둘로 나누면 단일문(simple statement)과 구조문(structured statement)으로 나눌 수 있다. 파스칼의 각 문장의 끝은 ‘;’ 기호로 구별된다.

### ● 단일문 (Simple statement)

단일문은 메모리를 변경하거나 값을 대입하고, 프로시저나 함수를 활성화하는 등의 문장이다. 단일문으로 대표적인 것은 대입(assignment), 프로시저, goto 문 등이 있다. Goto 문장은 많이 사용하지 않는 것이 좋으며, 오브젝트 파스칼에서 거의 사용할 일은 없다. 대입문과 프로시저 호출은 프로그램에서 가장 핵심적으로 사용되는 문장이다.

#### 1. 대입문 (Assignment)

대입문은 프로그램 내에서 메모리의 내용을 변경하는 문장이라고 생각하면 된다. 대입문에 사용되는 대입 연산자(=)의 우측에는 대입할 값을, 좌측에는 대입의 대상을 지정한다.

앞에서도 자주 코드가 등장하였으므로 다들 이해하고 있을 것으로 믿는다.

대입문에서 중요한 것은 대입 호환성(assignment compatibility)이 지켜지고 있는지 여부이다. 대입 호환성 규칙은 다소 복잡하지만, 기본적으로 좌우의 데이터 형을 같은 것을 사용한다고 생각하기 바람에 더 자세한 내용은 도움말을 참고하면 된다.

## 2. 프로시저문 (procedure statement)

프로시저 호출문은 메소드나 프로시저를 실행하기 위해 호출하는 문장이다. 전형적인 예는 다음과 같다.

```
procedurename(parameter1, parameter2);
```

여기서 중요한 것은 파라미터의 수와 데이터 형을 일치시키는 것이다. 이때 파라미터가 넘어가는 방식에 몇 가지가 존재하는데 여기에 대해서는 이 장의 후반부에 다루게 된다.

### ● 구조문 (Structured statement)

구조문은 순서대로 실행되어야 할 문장 들로 구성된 문장이다. 구조문은 주로 문장 들이 실행되는 방법을 조절하는데 이용된다. 오브젝트 파스칼이 지원하는 구조문은 복합문(compound statement), 조건문(conditional statement), 순환문(loops), with 문의 4 가지로 구분할 수 있다.

#### 1. 복합문 (compound statement)

복합문이란 실행될 문장들을 하나의 블록으로 묶는 것으로 begin~end 키워드 사이에 수록된 문장들을 하나의 복합문으로 생각한다. 문법적인 표현은 다음과 같다.

```
begin
    statement1;
    statement2;
end;
```

#### 2. 조건문 (conditional statement)

조건문은 특정 문장이 실행될 것인지 여부를 결정하는 문장으로 오브젝트 파스칼에는 if 와 case 의 2 가지 조건문이 있다.

if...then 조건문은 boolean 표현식 여부에 따라서 조건이 결정되는 경우이다. 즉, if 뒤에 따라 나오는 표현식이 True 이면 then 이후의 문장이 실행되고, False 이면 else 이후의 문장이 실행된다. Boolean 표현식에는 논리 연산에 사용되는 not, and, or, xor 등의 연산자를 사용할 수 있다.

```
if boolean_expression then
    statement1
else
    statement2;
```

여기에서 주의할 것은 if ... then ... else 문이 하나의 문장이기 때문에 else 문의 문장 뒤에 ‘;’을 사용한다는 것이다. 초보자들이 자주 틀리는 부분 중 하나가 else 문이 있는데, then 이후의 문장 뒤에 ‘;’을 사용하는 것이다.

case 문은 if 문과 달리 조건부에 여러가지 경우가 나타날 수 있는 경우에 사용된다. case 문의 조건 표현식에는 서수형 표현식이 사용된다.

```
case ordinal_expression of
    1: statement1;
    2: statement2;
    3: statement3;
    else otherstatement;
end;
```

### 3. 순환문 (loops)

순환문은 문장들을 여러 차례 실행할 수 있게 해주는 문장으로, 오브젝트 파스칼에서는 for...to/downto...do, while...do, repeat...until 의 3 가지 순환문을 지원한다.

이들 순환문에서 빠져 나갈 때에는 break, exit 의 2 가지를 사용할 수 있는데 break 는 중첩된 루프가 있을 경우 그 이전의 루프로 빠져나가며, exit 는 모든 루프에서 빠져나간다.

for...do 루프는 가장 간단한 순환문으로, 루프를 몇 번이나 실행시킬 것인지를 알고 있을 때에 사용한다.

간단한 사용 예는 다음과 같다.

```
for Count := 1 to 10 do i := i + Count;
```

이 문장은 Count 변수의 값이 1~10 까지 증가하면서, 10 번 루프를 돌게 된다. 만약 값을 감소시키면서 루프를 돌게 하려면 to 대신 downto 를 사용한다.

while...do 순환문은 루프에 들어가기 전에 조건문을 검사하게 된다. 조건이 거짓이면, 루프가 한번도 실행되지 않는다. 참고로 다음 문장은 무한 루프를 돌게 된다.

```
while True do Something;
```

repeat...until 문장은 repeat 와 until 사이의 문장들을 until 뒤에 지정한 조건이 참이 될 때까지 계속해서 반복 실행하게 되는 것이다. while...do 문장과는 달리 조건과 상관없이 한번은 실행된다. 문법은 다음과 같다.

```
repeat Something until Condition;
```

#### 4. with...do 문

with 문장은 중복되는 코드가 있을 때에 유용하다. 사용 예는 이번 장의 앞부분에서 소개한 바 있으므로 생략하겠다.

## 프로시저와 함수 (Procedures and Functions)

파스칼에서는 루틴이라는 개념을 중요시 한다. 파스칼의 루틴에는 프로시저와 함수의 2 가지 종류가 있다. 프로시저와 함수의 유일한 실제 차이점은 함수는 리턴값을 가지며, 프로시저는 그렇지 않다는 것이다. 이들은 모두 선언부에는 루틴의 이름과 파라미터, 리턴값에 대한 선언을 해야 하며, 구현부분에서는 begin~end 블록에 둘러싸인 부분에 실제 루틴의 몸체를 구성한다.

가장 단순한 형태의 프로시저와 함수의 코드 예제를 살펴 보자.

```
procedure Hello;
```

```
begin
```

```
    ShowMessage('Hello !');
```

```
end;
```

```
function Hello: Boolean;
```

```
begin
```

```
    ShowMessage('Hello !');
```



```
Result := True;
end;
```

이 두가지 프로시저와 함수는 본질적으로 똑같은 동작을 한다. 다만 후자의 경우에는 True 를 리턴한다. 이렇게 리턴값을 지정할 때에는 앞의 코드와 같이 Result 변수에 값을 대입할 수도 있고, 다른 방법으로는 함수의 이름에 대입할 수 있다. 즉, 'Hello := True;'가 같은 의미가 된다.

## ● 파라미터의 전달 방법

파스칼 언어는 파라미터 전달을 값에 의해서도 할 수 있고(call by value), 참조(reference)에 의해서도 할 수 있다(call by reference). 파라미터를 참조에 의해 전달한다는 것은 그 값이 루틴의 형식상의 파라미터 내의 스택에 복사되지 않는다는 뜻이다. 그 대신 프로그램은 루틴의 코드 안에 있는 원래의 값을 참조하게 된다. 이렇게 하면 프로시저나 함수가 그 파라미터의 값을 바꾸게 할 수 있다. 이렇게 참조에 의해 파라미터를 전달하려면 var 키워드를 사용한다. 값에 의한 전달 방식은 프로시저 루틴에서 넘어온 파라미터의 값을 복사해서 사용하게 되므로 파라미터에 전달된 변수의 값을 변화시키지 않는다.

참조에 의한 파라미터의 전달은 미리 정의된 데이터 형, 구형의 문자열, 그릭 큰 레코드 들에 대해서도 적용된다. 실제로 델파이의 객체 들은 모두 참조에 의해 전달된다.

참고로 델파이 3 부터는 out 이라는 새로운 종류의 파라미터가 제공된다. out 파라미터는 초기값이 없고 단지 값을 반환하기 위해서만 사용된다. 이러한 파라미터 들은 오직 COM 프로시저나 함수들에 대해서만 사용된다. 초기값이 없다는 점을 빼고는 out 파라미터는 var 파라미터와 똑같이 작용한다.

그 밖에 상수 파라미터와 개방형 배열 파라미터, 가변 데이터형 개방형 배열 파라미터 등이 있는데 여기에 대해서 조금 더 알아보도록 하자.

### 1. 상수 파라미터 (const parameter)

상수 파라미터는 const 키워드를 이용해서 사용하는데, 루틴 내부에서 상수 파라미터에 값을 대입할 수 없다는 특징이 있다. 그렇기 때문에 컴파일러는 파라미터의 전달을 최적화시킬 수 있다. 다음의 코드를 살펴보자.

```
procedure Test(const Parameter1: Integer; var Parameter2: Integer);
begin
    Parameter2 := 10;           //참조에 의한 전달이므로 변수의 값을 변경할 수 있다.
    Parameter1 := 10;          //상수 파라미터에 값을 대입하려 했으므로 컴파일 에러가 발생한다.
```

end;

## 2. 개방형 배열 파라미터

파스칼에서는 파라미터의 수를 고정적으로 선언해야 하는 약점이 있는데, 이를 극복하기 위한 방법으로 개방형 배열을 사용하면 된다. 이 때 개방형 배열은 특정 데이터 형만 지정하며, 요소의 수가 몇 개가 될 지는 지정하지 않는 것을 말하며, 다음과 같이 정의한다.

var

parameter1: array of atype;

그러면 실제로 이를 이용하는 예를 살펴보자. 가장 쉽게 생각할 수 있는 것은 합을 내는 함수이다. 즉, 몇 개의 수가 될지는 모르지만 개방형 배열에 저장된 요소를 모두 더하는 것이다.

function Sum(const Parameter1: array of Integer): Integer;

var

i: integer;

begin

Result := 0;

for i := Low(Parameter1) to High(Parameter1) do Result := Result + Parameter1[i];

end;

여기에서 High 와 Low 라는 서수형 루틴을 유용하게 사용한다는 것을 잘 알 수 있을 것이다. 이 함수는 다음과 같이 호출할 수 있다.

var

Test1: array[1..5] of Integer;

i : Integer;

begin

for i := 1 to 5 do Test1[i] := i;

i := Sum(Test1);

end;

## 3. 가변 데이터형 개방형 배열 파라미터

형이 지정되지 않은 개방형 배열을 사용하여, 데이터 형을 지정하지 않고도 파라미터를 사용하는 방법도 있다. 이럴 때 사용하는 것이 array of const 라는 것으로 서로 다른 데이터 형을 지닌 요소들을 배열에 넣어서 한 번에 전달하는 것이다. 선언부는 다음과 같다.

```
const Parameter1: array of const;
```

가장 전형적인 사용 예는 문자열을 처리할 때 이용하는 Format 함수를 들 수 있다.

- 디폴트 파라미터 (Default parameters)

델파이 4 에서는 프로시저와 함수에 디폴트 파라미터를 사용할 수 있게 되었다. 디폴트 파라미터는 다음과 같은 형태로 지정한다.

Name: Type = value

프로시저나 함수를 호출할 때 선언부에 디폴트 파라미터를 포함하면, 이를 호출할 때 디폴트 값과 같다면 파라미터를 빼도 된다. 예를 들어 다음과 같이 선언한 프로시저가 있다고 하자.

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

이때, 다음의 두 문장은 같은 기능을 한다.

```
FillArray(MyArray, 0);
```

```
FillArray(MyArray);
```

- 호출 규칙 (Calling conventions)

호출 규칙이란 파라미터를 전달하는 방법을 말한다. 델파이 2.0 부터는 fastcall 이라는 호출 규칙을 디폴트로 사용하는데, 가능하면 최대 3 개까지의 파라미터를 CPU 레지스터에 전달해서 이를 처리하기 때문에 처리 속도가 빠르다. fastcall 호출 규칙을 이용하려면 register 키워드를 사용한다. 그렇지만 디폴트가 fastcall 이기 때문에 보통의 경우 특별히 지정해줄 필요는 없다.

윈도우와의 호환성을 위해서는 stdcall 호출 규칙을 사용한다. 여기에 대해 더욱 자세한 사항은 DLL 에 대해서 다루는 24 장을 참고하기 바란다.

## 정 리 (Summary)

이번 장에서는 오브젝트 파스칼의 기본적인 데이터 형과 문법에 대해서 알아 보았다. 오브젝트 파스칼의 OOP 적인 특성에 대해서는 다음 장에서 다루게 될 것이다.

무슨 일을 하든 기초가 중요하다는 것은 당연한 일이다. 오브젝트 파스칼의 문법에 대한 이해가 충분하지 않은 델파이 프로그래머는 그 발전의 정도에 한계가 있기 마련이다. 이 책에서는 분량의 문제로 충분히 자세하게 문법에 대해서 다루지는 못하고 있지만, 델파이에 서 제공되는 도움말이나 다른 서적을 이용해서 충분히 문법적인 지식을 익혀두도록 권하고 싶다.

# 객체지향 언어로서의 오브젝트 파스칼

## (Object Pascal As A OOP)

오브젝트 파스칼의 객체지향성을 몰라도 델파이 어플리케이션을 쉽게 만들 수 있다. 단순히 폼을 하나 만들고, 거기에 여러가지 컴포넌트 들을 추가하고, 이벤트 핸들러에 적당한 내용의 코드 들을 추가하면 그걸로 충분한 것이다. 그렇지만, 이것을 이해하면 델파이가 어떤 방법으로 작업을 처리하는지 이해할 수 있고, 자신만의 컴포넌트를 만들어 내거나, 비교적 커다란 프로젝트를 진행할 때에 커다란 도움을 받게 될 것이다.

### OOP 의 기본 개념

프로그램은 데이터와 이를 처리하는 알고리즘으로 구성되어 있다고 하는 유명한 말이 있다. 그렇기 때문에, 프로그래밍 언어의 구조를 가만히 살펴보면 대부분의 경우 데이터를 다루는데 필요한 각종 정의들과, 이들을 제어하기 위한 여러가지 문법으로 구성되어 있다는 것을 알 수 있다. 기본적으로 데이터의 표현은 ‘데이터 형 (data type)’이라는 개념으로 표현되는데, 이 데이터 형의 개념이 발전하면서 언어들이 다른 모습을 띠게 된다.

초기의 프로그래밍 언어는 정해진 데이터 형을 이용하였으며, 그 이후의 프로그래밍 언어 (파스칼 등)에서는 레코드나 배열 등의 개발자가 정의해서 사용할 수 있는 데이터 형을 도입하였다. 그러다가, 이러한 데이터 형에 그 데이터를 다룰 때 쓰이는 작업과 결부시키게 되었는데, 이와 같이 기본적인 데이터와 이를 다루기 위한 메소드로 이루어진 데이터 형인 클래스가 OOP 의 기초가 되는 개념이다.

데이터 형과 함께 가장 중요한 프로그래밍 언어의 구성요소인 제어 구조를 보면, 초기의 프로그래밍 언어는 점프(goto)와 분기문(if-then, case 등), 순환문(for, while 등)으로 이루어진 제어문을 가지고 있었다. 여기에 서브루틴의 개념이 추가되어 일반적인 프로그래밍 언어의 전반적인 구조를 가지게 되었다. OOP 에서는 이러한 서브루틴을 데이터와 마찬가지로 추상화 하여, 이들을 클래스로 관리하게 된다.

OOP 에 있어서 가장 중요한 3 가지 개념은 캡슐화, 상속, 다형성으로 아래에 간단하게 설명하였다.

- 캡슐화 (Encapsulation)

OOP 의 가장 핵심적인 요소라고 할 수 있는 것이 캡슐화 개념으로, 데이터와 서브루틴을 추상화 한 것이다. 이 개념은 클래스로 구현되는데, 클래스란 특정 객체 그룹을 추상적으로 정의해 놓은 것이라고 생각하면 된다. 클래스는 일종의 형(type)의 정의로서 필드(클래스의

객체 상태를 나타내는 데이터)가 있고, 이에 대한 작업을 정의하는 메소드가 있다.

여기서 혼동하지 말아야 할 것은 클래스란 일종의 데이터 형이고, 객체는 클래스라는 데이터 형인 하나의 구체적인 예(인스턴스)라는 것이다. 예를 들어 설명하면 클래스란 객체를 만들기 위한 주형이며, 객체란 클래스라는 주형에서 만들어진 병과 같은 것이다.

## ● 상속 (Inheritance)

상속은 클래스를 처음부터 만들지 않고, 기존의 클래스를 기반으로 해서 새로운 클래스를 정의하는 것이다. 이때 기반이 되는 클래스를 부모 클래스, 만들어진 클래스를 서브 클래스라고 하며 서브 클래스는 부모 클래스로부터 필드와 메소드를 상속한다.

상속 개념이 실제로 쓰일 때에는 다음과 같은 특징을 가지고 있다.

1. 상속은 일반적인 클래스의 특수한 경우를 나타낼 때 쓰인다. 즉, 자동차라는 부모 클래스에서 기본적인 자동차의 필드와 메소드를 표현했다고 하면, 여기에서 상속받은 티코라는 서브 클래스는 기본적인 자동차 클래스의 필드와 메소드 외에 자신만의 필드와 메소드를 정의할 수 있다.
2. 반대로 여러가지로 특화된 클래스들의 공통점을 가진 부모 클래스를 만들 수도 있다. 예를 들어, 기존에 학생이라는 클래스와 선생이라는 클래스를 가지고 있는 경우 이들의 공통적인 필드와 메소드를 정의하는 사람이라는 부모 클래스를 만들고, 공통의 요소를 공유하게 할 수 있다.
3. 실제로 상속 개념이 쓰일 때에는 중복되는 코드를 막을 수 있으며 (공통부분을 가지는 부모 클래스가 사용되므로), 복잡한 데이터의 개념을 쉽게 이해할 수 있게 된다.

## ● 다형성 (Polymorphism)

다형성이란 하나의 프로그램 변수를 가지고 서로 다른 클래스 객체를 참조하는 것을 말한다. 즉, 어떤 객체에 대해 작업을 할 때 객체의 type에 적절하게 반응할 수 있다는 의미이다.

## 클래스와 객체 (Classes and Objects)

### ● 텔파이 OOP에 대한 문법적인 고찰

구체적인 예제에 들어가기 전에, 텔파이의 OOP에서 나오게 되는 중요한 개념들과 문법적인 특징에 대해서 알아보도록 하자.

#### 1. 필드 (Fields)

필드는 객체에 속해있는 일종의 변수라고 이해하면 된다. 필드는 클래스 형을 포함해서 어떤 데이터 형으로도 선언해 사용할 수 있다. 필드를 선언하려면 단순히 변수를 선언하듯이 하면 된다. 예를 들어, 다음의 선언부는 TNumber 라는 컴포넌트를 선언하는데, 여기에는 Int 라는 정수 필드만을 가지고 있다.

```
type
TNumber = class
    Int: Integer;
end;
```

## 2. 메소드 (Methods)

메소드는 클래스와 연관되어 있는 프로시저나 함수를 말한다. 메소드를 호출할 때에는 반드시 객체를 지정해야 하며, 메소드는 그 객체 위에서 동작한다는 점이 일반적인 프로시저나 함수와의 차이점이다. 예를 들어 다음의 코드를 살펴보자.

```
SomeObject.Free;
```

이 코드는 SomeObject 라는 객체의 Free 메소드를 호출한다.

## 3. Inherited 키워드

메소드를 구현할 때 inherited 라는 키워드를 사용하면 클래스의 조상이 가지고 있던 메소드를 호출하게 된다. 보통 메소드를 오버라이드할 때, 그 메소드의 기능을 추가하기 위해 일단은 조상 클래스의 메소드를 호출하고 나서 부가적인 기능을 수행하거나, 사전 작업을 지정한 후 마지막에 조상 클래스의 메소드를 inherited 키워드를 이용하여 호출하는 것이 전형적인 방법이다.

## 4. Self 지시어

메소드의 구현부분에 Self 라는 지시어를 사용하면, 이 지시어는 메소드가 실행되는 객체를 지칭하는 것이다. 예를 들어, 다음의 코드는 TCollection 클래스의 Add 메소드를 구현한 부분이다.

```
function TCollection.Add: TCollectionItem;
```

```
begin
```

```
    Result := FItemClass.Create(Self);
```

```
end;
```

여기에서 Add 메소드는 FItemClass 필드에 의해 참조되는 Create 메소드를 호출하는데, 이 필드는 항상 TCollectionItem 의 자손이다. TCollectionItem.Create 메소드는 TCollection 형의 파라미터를 하나 가지기 때문에, Add 가 호출될 때 TCollection 인스턴스 객체를 넘겨 준다.

## 5. 프로퍼티 (Properties)

프로퍼티는 필드와 마찬가지로, 객체의 속성을 정의하는 것이다. 그렇지만 필드가 단지 내용이 변경되거나, 검사할 때 사용되는 단순한 저장소의 역할을 하는데 비해 프로퍼티는 데이터를 읽고, 쓰는 특별한 동작과 연관되어 있다. 프로퍼티는 객체의 속성에 접근할 수 있는 제어권을 제공한다. 프로퍼티에 대한 보다 자세한 내용은 제 4 부의 내용을 참고하기 바란다.

### ● 클래스의 선언, 생성, 파괴

클래스는 사용자가 정의한 데이터 형이다. 클래스는 내부적인 데이터와 메소드를 가지고 있으며, 유사한 많은 객체들의 전반적인 특징과 행동들을 정의한다. 이에 비해 객체는 클래스의 구체적인 변수로, 실제로 메모리를 차지하게 되는 인스턴스이다.

오브젝트 파스칼에서 새로운 클래스 데이터 형을 선언하는 구문은 다음과 같다.

```
type
```

```
    TPerson = class
```

```
        Name, ID: string;
```

```
end;
```

위의 코드는 TPerson 이라는 클래스를 선언하고, 이 클래스의 필드인 Name, ID 를 정의했다. 델파이에서는 일반적으로 모든 클래스의 이름 앞에는 T 자를 붙이도록 되어 있다. 이것은 강제 사항이 아니지만 관습을 따르는 것이 아무래도 좋을 것이다. 위의 선언문을 보면 마치 레코드 선언과 비슷하다는 것을 알 수 있을 것이다. 클래스의 상속을 하는 구문은 class 키워드 옆에 괄호를 치고 이 안에 상속 받을 클래스의 이름을 적어넣으면 된다.

```
TPerson = class(TObject)
```



라는 구문은 TObject 라는 클래스에서 상속받는 TPerson 클래스를 선언하는 구문이다.

이들에 접근하기 위해서는 다음과 같은 방법을 이용하면 된다.

```
var
  Person: TPerson;
begin
  Person.Name := '정지훈';
  Person.ID := 'ttolttol'
end;
```

그러나, 위의 코드는 실제로 실행되지 않는다. 이는 TPerson 이라는 클래스의 변수인 Person 을 선언했지만, 실제 이 클래스의 인스턴스가 생성되지 않았기 때문이다.

cf. 오브젝트 파스칼에서는 클래스 형으로 선언된 각 변수 들이 각각의 객체의 값을 가지고 있는 것이 아니라, 객체에 대한 참조값(reference), 즉 그 객체가 저장된 메모리 위치의 포인터를 가지고 있는 것이다. 이러한 패러다임을 ‘객체참조 모델 (object reference model)’ 이라고 한다. 그러므로 아래와 같이 변수를 선언하면,

```
var
  Person: TPerson;
```

메모리에 객체가 만들어지는 것이 아니라 그저 객체에 대한 메모리의 위치가 정해지고, 여기에 대한 참조값이 Person 변수에 저장되는 것이다. 그러므로, 실제로 객체의 인스턴스를 사용하려면 인스턴스를 직접 만들어야 한다. 폼에 추가하는 컴포넌트의 인스턴스는 델파이 에 의해 자동으로 만들어진다.

객체의 인스턴스를 만들기 위해서는 그 객체의 생성자(constructor)인 Create 메소드를 사용하면 된다. 생성자는 새로운 객체를 위해 메모리 배치를 하고 초기화 시키는 특수한 프로시저이다. 이러한 생성자는 델파이의 객체 모델의 가장 기본적인 클래스인 TObject 에서 상속받게 되므로, 개발자들은 아무 걱정없이 이를 사용할 수 있다. 아래의 코드를 보자.

```
var
  Person: TPerson;
begin
```

```

    Person := TPerson.Create;
    Person.Name := '정지훈';
    Person.ID := 'ttolttol';
end;

```

TPerson.Create 구문에 의해 실제 객체가 생성된다. 이 Create 메소드는 TObject 클래스의 constructor 로서, 모든 클래스가 이를 상속하게 된다.

그러므로, TPerson = class 와 TPerson = class(TObject) 는 같은 의미이다.

이렇게 일단 객체를 생성했으면 이를 결국에는 삭제해야 한다. 이때에는 Free 메소드를 호출하면 된다. Free 메소드 역시 TObject 클래스의 메소드이다. 이와 같이 필요할 때 객체를 만들어서 사용하고 일이 끝나면 없애 버리는 식으로 사용하면 된다.

그럼, 이를 이용해서 간단한 예제를 만들어 보도록 하겠다.

다음과 같이 에디트 박스 2 개와 버튼 4 개를 폼 위에 올려 놓고, 각 버튼의 Name 프로퍼티를 btnCreate, btnFree, btnAssign, btnShow 로 설정하자 (그림 5-1). 그리고, Caption 프로퍼티를 ‘생 성’, ‘해 제’, ‘대 입’, ‘보 기’로 설정한다. 각 에디트 박스의 Text 프로퍼티는 지운다. 유닛의 type 문장에 TPerson 클래스를 선언하고, 전역변수 Person 을 선언한다. 그리고, 각 버튼의 OnClick 이벤트 핸들러를 아래와 같이 작성한다.

```

type
    ...(중략)

    TPerson = class                                //클래스 선언부
        Name, ID: string;
    end;

var
    Form1: TForm1;
    Person: TPerson;                               //전역변수 선언

procedure TForm1.btnCreateClick(Sender: TObject);
begin
    Person := TPerson.Create; //인스턴스 생성
end;

procedure TForm1.btnAssignClick(Sender: TObject);
begin
    Person.Name := Edit1.Text; //에디트 박스의 값을 대입

```

```

    Person.ID := Edit2.Text;
end;

procedure TForm1.btnShowClick(Sender: TObject);
begin
    ShowMessage ('안녕하세요 ? '+Person.Name+ '씨, 당신의 ID 는 '+Person.ID+'입니다.');
```

```

end;

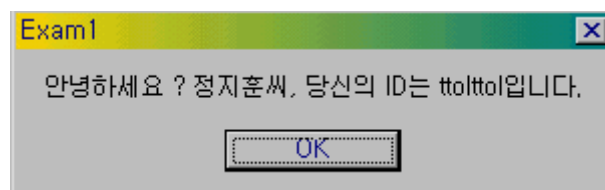
procedure TForm1.btnFreeClick(Sender: TObject);
begin
    Person.Free;
end;

```



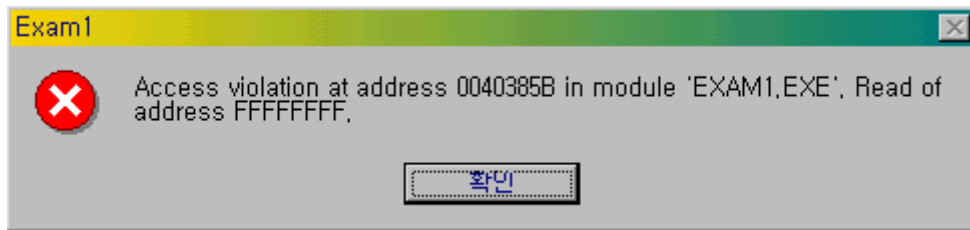
(그림 5-1) 예제 폼 그림

4 개의 버튼에 대한 이벤트 핸들러를 이와 같이 작성하면 ‘생성’ 버튼을 클릭하면 TPerson 클래스의 인스턴스가 생성되어 Person 변수에 대입되고, ‘대입’ 버튼을 클릭하면 Person 객체의 Name, ID 필드에 에디트 박스의 내용이 대입된다. ‘보기’ 버튼을 클릭하면 Person 객체의 Name, ID 필드를 이용해 다음(그림 5-2)과 같은 메시지 박스가 뜬다. 이때 TPerson 클래스의 인스턴스가 생성되지 않았다면 ‘Access violation’ 에러 메시지가 나타날 것이다 (그림 5-3). 마찬가지로 ‘생성’ 버튼을 클릭한 후, ‘해제’ 버튼을 클릭하면 생성되었던 TPerson 클래스의 인스턴스가 파괴되므로, 이 때 ‘대입’, ‘보기’ 버튼을 클릭하면 역시 ‘Access violation’ 에러가 발생한다.



(그림 5-2) 에디트 박스 이름(정지훈)과 ID(ttolttol)를 입력한 후 ‘생성’, ‘대입’, ‘보기’ 버튼을 클릭하

면 나타나는 메시지 박스



(그림 5-3) Create 하지 않았거나, Free 를 호출한 후 ‘대입’, ‘보기’ 버튼을 클릭해서 인스턴스에 접근하려 하면 이와 같은 에러 메시지가 나타난다.

- 메소드와 생성자(constructor), 파괴자(destructor)의 추가

이제 TPerson 클래스에 몇가지 메소드를 추가해서 조금은 쓸모가 있는 클래스로 만들어 보자. 현재 가지고 있는 Name 필드는 그대로 두고, ID 필드 대신, 주민등록번호를 저장할 수 있는 RegID 필드를 만들자. 이때 필드 임을 나타내기 위해 각 필드의 앞에 접두어로 ‘F’를 붙이도록 한다. 그리고, 주민등록번호 필드 값을 가지고 이 값이 유효한지 알아보는 IsValid 함수와 나이와 성별을 알 수 있는 GetAge, GetSex 라는 함수를 추가한다.

또한, 이 클래스의 객체가 생성될 때 값을 초기화 시키기 위해 생성자를 추가하자. 생성자(constructor)는 특별한 형태의 프로시저로, 이것을 클래스에 적용하면 자동적으로 그 클래스의 객체를 위해 메모리를 할당하게 되며, 초기화 작업을 지정할 수 있다. 단순히 constructor 라는 예약어로 선언하면 되며, 클래스 메소드로 사용할 때에는 객체에 대한 메모리 할당 작업을 하게 되지만, 이미 인스턴스화된 객체에서 사용될 때에는 메모리 할당 작업은 하지 않고, 정의된 초기화 작업만 하게 된다. 여기서 만드는 클래스에는 생성자로 Create 라는 프로시저를 정의하는데, 파라미터로 FName, FRegID 필드를 채울 수 있도록 선언한다.

마찬가지로 파괴자(destructor)도 정의할 수 있으며 destructor 라는 예약어로 선언하면 된다. 이 프로시저는 객체가 파괴되기 전에 시스템 자원을 release 하는 작업을 주로하게 되는데, 여기서는 특별히 추가하지 않는다. 따로 파괴자를 추가하지 않아도 모든 클래스는 TObject 에서 파생되기 때문에 기본적인 Free, Destroy 프로시저는 정의되어 있다. .

이렇게 확장한 클래스의 선언부는 다음과 같다.

type

```
TSex = (sMale, sFemale);           //GetSex 함수에서 쓰임
TPerson = class(TObject)
    FName: string;
```

```

FRegID: string;
constructor Create(Name, ID: string);
function IsValid: Boolean;
function GetSex: TSex;
function GetAge: integer;
end;

```

이제 각 함수를 구현해 보도록 하자.

먼저 생성자를 구현해 보자. 생성자에서는 Name, ID 를 파라미터로 받아서 이 값을 필드에 적용하여 초기화 한다.

```

constructor TPerson.Create(Name, ID: string);
begin
    FName := Name;
    FRegID := ID;
end;

```

다음으로 FRegID 필드에 저장된 주민등록번호가 잘못된 것이 아닌지 검사하는 IsValid 함수를 구현하도록 하자. 주민등록번호를 검증하는 방법은 주민등록번호의 13 자리 중 마지막 자리를 제외한 12 자리에 각각 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 4, 5 를 곱한 뒤 전체를 더한다. 예를 들어, 701203-1079727 이라는 주민등록번호가 있다고 하면,  $(7*2) + (0*3) + (1*4) + (2*5) + (0*6) + (3*7) + (1*8) + (0*9) + (7*2) + (9*3) + (7*4) + (2*5) = 136$  이다. 이 값을 11 로 나눈 나머지 값을 11 에서 뺀다. 위의 경우에는 나머지가 4 이므로  $11-4=7$  이 된다. 이 값이 마지막 주민등록번호 값과 일치하면 일단 유효한 번호인 것이다. 일종의 해싱 함수(hashing function)이다. 이를 이용해서 IsValid 함수를 구현해 보자.

```

function TPerson.IsValid: Boolean;
begin
    Result := False;
    if ((Length(FRegID) = 13) and ('0000000000000' < FRegID) and ('9999999999999' > FRegID))
    then
        if (11 - ((StrToInt(FRegID[1]) * 2 + StrToInt(FRegID[2]) * 3 + StrToInt(FRegID[3]) * 4 +
            StrToInt(FRegID[4]) * 5 + StrToInt(FRegID[5]) * 6 + StrToInt(FRegID[6]) * 7 +
            StrToInt(FRegID[7]) * 8 + StrToInt(FRegID[8]) * 9 + StrToInt(FRegID[9]) * 2 +
            StrToInt(FRegID[10]) * 3 + StrToInt(FRegID[11]) * 4 + StrToInt(FRegID[12]) * 5) mod 11)) =

```

```

        StrToInt(FRegID[13]) then
            Result := True
        else
            Result := False;
    end;
end;

```

즉, 일단 FRegID 필드 값이 13 자리이고, 숫자로 이루어 졌는지 확인하고, 각 자리의 값을 정수형으로 바꿔서 위에서 설명한 공식에 따라 적절한 지 알아보고 적절하면 True, 그렇지 않으면 False 를 반환한다.

Cf.

1. 파스칼의 문자열은 일종의 배열로 생각할 수 있기 때문에, 이런 형태의 코드가 가능한 것이다. 예를 들어 Name 이라는 문자열 변수의 값이 'Sample'이라고 할 때, Name[1], Name[2], Name[3], Name[4], Name[5]의 값은 각각 'S', 'a', 'm', 'p', 'l', 'e' 이다.
2. Length 함수  
문자열의 길이를 구하는 함수이다. 델파이 1.0 까지는 문자열의 0 번째 요소, 즉 위의 예를 들면 Name[0] 값에 문자열의 길이가 저장되어 있었으나, 2.0 부터는 Length 함수를 사용하여 길이를 구한다.
3. StrToInt, IntToStr 함수  
정수와 문자열의 형전환을 해주는 함수이다. 예를 들어 IntToStr(123)의 결과값은 '123'이고, StrToInt('123')의 결과값은 123 이다.

마지막으로 GetSex 와 GetAge 를 구현해 보자.

```

function TPerson.GetSex: TSex;
begin
    if IsValid then
        if FRegID[7] = '1' then Result := sMale else Result := sFeMale;
    end;
end;

```

```

function TPerson.GetAge: integer;
var
    Date: string;
begin
    if IsValid then
        begin

```

```

Date := DateToStr(Now);
Result := StrToInt(Copy(Date, 1, 2)) - StrToInt(Copy(FRegID, 1, 2));
if (Copy(Date, 4, 2) + Copy(Date, 7, 2)) < Copy(FRegID, 3, 4) then
    Result := Result - 1;
end;
end;

```

GetSex 함수는 주민등록번호의 7 번째 자리가 ‘1’이면 남자, ‘2’이면 여자로 쉽게 이해가 될 것이다. GetAge 함수는 일단 System 의 날짜를 얻을 수 있는 Now 함수를 사용한 후 이를 문자열로 변환한다. 그리고, 여기서 연도만을 뽑아 정수로 변환한 후, 주민등록번호의 첫 두자리가 태어난 해를 가리키게 되므로 이를 정수로 변환해서 뺀다. 여기에 생일이 지나지 않았으면 1 을 빼야 만으로 계산한 나이가 된다.

생일을 비교할 때, 입력 받은 주민등록번호 상의 생일은 ‘0301’ 과 같이 연속된 4 개의 문자로 구성되지만, DateToStr 로 변환된 Date 변수의 값은 ‘98-06-21’과 같은 형식으로 저장되기 때문에 이를 동등하게 비교하기 위해 Copy 함수를 이용하여 주민등록번호 상의 생일처럼 연속된 4 개의 문자로 만드는 루틴이 추가되었다.

그럼, 이 클래스를 활용하는 프로그램을 만들어 보자.

두번째 예제에 TPerson 의 구현 부분을 추가하고, 폼을 다음(그림 5-4)과 같이 디자인 한다. 여기서도 첫번째 예제와 마찬가지로 에디트 박스를 2 개 추가한다. 여기에는 이름과 주민등록번호를 입력받을 것이다. 그리고, 버튼을 2 개 추가한 후 Name 프로퍼티를 btnAssign 과 btnExecute 로 설정하고, Caption 을 각각 ‘대입’과 ‘실행’으로 설정한다.



(그림 5-4) 두번째 예제의 폼 디자인

그리고, 각 버튼의 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.btnAssignClick(Sender: TObject);
begin
    if Assigned(Person) then Person.Destroy;
    Person := TPerson.Create(Edit1.Text, Edit2.Text);
end;

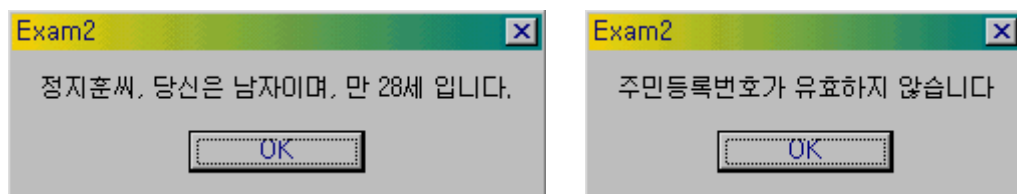
```

```

procedure TForm1.btnExecuteClick(Sender: TObject);
var
    Sex, Age: string;
begin
    if not Assigned(Person) then
    begin
        ShowMessage('클래스가 생성되지 않았습니다 !');
        Exit;
    end;
    if not Person.IsValid then
    begin
        ShowMessage('주민등록번호가 유효하지 않습니다');
        Exit;
    end
    else
    begin
        if Person.GetSex = sMale then Sex := '남자' else Sex := '여자';
        ShowMessage(Person.FName + '씨, 당신은 ' + Sex + '이며, 만 ' +
            IntToStr(Person.GetAge) + '세 입니다.');
```

그렇게 어렵지 않은 코드이므로 자세한 설명은 생략하도록 하겠다.

참고로 제대로 된 주민등록번호를 입력했을 때와 그렇지 않았을 때의 메시지 박스는 다음과 같다.



(그림 5-5) 예제 실행 결과

이런 클래스의 사용 방법은 델파이에서 TComponent 를 클래스를 상속해서 더욱 재사용성이 뛰어난 컴포넌트로 개발할 수 있다. 이를 위해서는 프로퍼티에 대한 이해와 델파이가 제공하고 있는 컴포넌트 모델에 대한 이해가 필요한데, 여기에 대해서는 제 4 부에서 자세



하게 다를 것이다.

## 상속성과 델파이 폼

새로운 프로젝트를 생성하면, 델파이는 새로운 폼을 보여준다. 코드 에디터의 새로운 프로젝트의 내용을 살펴 보면, 델파이가 폼에 대한 새로운 객체 클래스를 선언하고 새로운 폼 객체를 생성하는 코드를 만들어 낸다는 것을 알 수 있다. 그러면 초기에 생성되는 델파이의 코드를 살펴 보자.

```
unit Unit1;
```

```
interface
```

```
uses Windows, Classes, Graphics, Forms, Controls, ... ;
```

```
type
```

```
TForm1 = class(TForm)    {폼에 대한 새로운 클래스 선언}
private
    { Private declarations }
public
    { Public declarations }
end;                      {클래스 선언부는 여기서 종료된다.}
```

```
var
```

```
Form1: TForm1;
```

```
implementation    {구현 부분의 시작}
```

```
{ $R *.DFM }
```

```
end.                {구현 부분과 유닛의 끝}
```

새로운 객체 데이터 형인 TForm1 은 TForm 에서 상속받는다는 것을 알 수 있다. 객체에는 데이터 필드와 메소드를 가진다는 것은 이미 설명한 바 있다. 그런데, TForm1 에는 아

직 메소드나 데이터 필드를 지정하지 않았다. 다만 TForm 클래스에서 상속받은 메소드와 프로퍼티를 가지게 될 것이다. 개발자는 TForm1 의 선언 부분에 마음대로 메소드나 프로퍼티 등을 추가할 수 있다. 또한, 컴포넌트를 폼에 추가하는 동작에 의해 텔파이가 메소드나 데이터 필드를 추가해 준다.

그렇지만, 이 어플리케이션을 실행시켜 보면 간단한 폼이 생성되는 것을 볼 수 있다. 이것이 의미하는 것은 무엇인가? 즉, 기본적으로 TForm 클래스의 모든 메소드와 기능을 상속받아서 사용할 수 있다는 것이다.

변수 선언부에서는 새로운 변수인 Form1 을 TForm1 으로 선언한다. 이렇게 선언함으로써 TForm1 클래스의 인스턴스가 될 수 있다. 클래스의 인스턴스는 여러 개 생성될 수 있다는 것은 이미 알고 있을 텐데, 만약 TForm1 클래스를 여러 인스턴스로 생성하면 그것이 바로 MDI(Multiple Document Interface) 어플리케이션이 되는 것이다.

그러면, 폼에 버튼을 하나 추가하고 그 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

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

이렇게 이벤트 핸들러를 작성하고 나면, 폼의 유닛의 코드는 다음과 같이 바뀌어 있을 것이다.

```
unit Unit1;
```

```
interface
```

```
uses Windows, Classes, Graphics, Forms, Controls, ...;
```

```
type
```

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

```
        Button1: TButton;           {새로운 데이터 필드}
```

```
        procedure Button1Click(Sender: TObject); {새로운 메소드}
```

```
    private
```

```
        { Private declarations }
```

```
    public
```

```
        { Public declarations }
```

end;

var

Form1: TForm1;

... (후략)

TForm1 의 선언부에 새로운 Button1 이라는 필드가 추가 되었음을 알 수 있다. 이렇게 새로운 컴포넌트를 폼에 추가할 때마다 새로운 필드가 type 선언부에 추가된다. 또한, 델파이에서 작성하는 모든 이벤트 핸들러는 폼 객체의 메소드로 선언된다. TForm1 에는 이제 Button1Click 이라는 새로운 메소드 프로시저가 추가되었다.

여기서 이런 동작들을 OOP 의 개념으로 생각해보자. 델파이의 폼 디자이너는 과연 무엇인가? 결국 델파이의 폼 디자이너는 TForm 이라는 클래스를 상속받은 새로운 형태의 클래스를 쉽게 만들어주는 일종의 위저드인 셈이다.

이제 상속의 의미를 이해하기 쉬운 비유를 들어 생각해 보자.

앞에서 처음 어플리케이션이 생성되었을 때의 폼은 기본 옵션으로된 자동차를 한 대 구입한 것으로 가정하자. 기본 옵션의 자동차는 자동차 메이커에서 항상 정해진 방식으로 만들어지기 때문에 가장 기본적인 기능만을 가지고 달릴 수 있을 것이다. 그렇지만, 이 자동차를 구입한 사람이 에어컨도 달고, 파워 핸들과 에어백 등의 여러가지 옵션을 장착할 수 있다. 이를 위해서 자동차를 구입한 사람은 자동차 전체를 새로 만들 필요는 없는 것이다. 즉, 이를 상속을 통해 설명하자면 기본적인 자동차를 상속받은 사용자가 자식 클래스 자동차에 새로운 옵션들을 추가한 것이다.

마찬가지로 델파이의 폼은 상속성을 설명할 때 가장 알기 쉽고, 전형적인 방법을 보여준다고 말할 수 있다.

## 클래스의 범위 (scope)

클래스의 멤버들은 서로 다른 범위를 가질 수 있다. 이러한 범위를 나타내는 지시어로는 private, protected, public, published 의 4 가지가 있다. 참고로 오브젝트 파스칼에서는 유닛도 범위를 결정하는데 한 몫을 한다. 비록 private 로 선언되었더라도 같은 유닛에 있으면 모두 접근이 가능하다.

- private

다른 유닛에 있는 경우 private 섹션에 선언된 멤버에는 접근할 수 없다. 다른 사용자가 접근할 필요가 없는 멤버들은 여기에 선언한다.

- `protected`

이 클래스에서 상속받은 클래스에서만 접근할 수 있는 멤버들을 여기에 선언한다. 즉, 현재의 클래스를 상속받아서 새로운 클래스를 만들 때 수정이 필요하다면 개발자에게 접근이 가능해야 하지만, 일반적으로 사용할 때에는 접근할 수 없도록 할 때 사용된다.

- `public, published`

다른 클래스에서 제한 없이 사용될 수 있는 멤버들을 여기에 선언한다. `published` 는 오브젝트 인스펙터에서도 볼 수 있는 멤버들을 선언할 때 사용한다. `published` 로 선언하면 멤버에 대한 RTTI(runtime type information)가 생성되며, 이를 이용해서 다른 프로그램들이 런타임에서 객체에 대한 필드, 메소드, 프로퍼티에 접근하게 된다. 델파이는 RTTI 를 이용하여 오브젝트 인스펙터에 프로퍼티를 보여주며, 폼 파일에 프로퍼티의 값을 저장하고 불러올 수 있다. `Published` 프로퍼티에 사용할 수 있는 데이터 형은 서수형과 문자열, 클래스와 메소드 포인터형, 세트형, 실수형 등을 사용할 수 있다. 배열은 사용할 수 없다. `published` 멤버가 있는 대부분의 클래스는 `TPersistent` 클래스에서 상속받는 것이 보통이다.

범위에 대한 지시어가 없을 때에는 일단 `public` 으로 간주한다. 그렇지만, 프로그래밍을 할 때에는 꼭 이러한 범위를 지정해주는 것이 좋은 버릇이다.

## 메소드 (Method)

객체의 동작은 메소드에 의해 정의된다. 메소드는 프로시저나 함수와 비슷하지만 지정한 클래스와 그의 파생 클래스 만의 객체를 위해 정의된다는 점이 다르다. 메소드에는 보이지 않는 파라미터로서 `Self` 라는 객체 자신의 참조자가 전달된다. 메소드는 또한 클래스 메소드로서 선언될 수 있는데, 이런 경우에는 클래스 참조로는 호출될 수 있으나 객체 참조로서는 호출되지 않는다. 객체가 없으므로 `Self` 파라미터도 없다.

- 메소드의 종류

메소드에도 그 동작방식과 용도에 따라 여러 가지 종류가 있다. 다음에 이들 각각의 특징에 대해 설명하였다.

1. 정적 메소드 (Static method): 지시어 `static`;

아무런 지시어가 없을 때에는 디폴트로 정적 메소드로 간주된다. 컴파일 할 당시에 메소드가 위치한 메모리 주소가 확정되는 메소드이므로, 그만큼 실행속도가 빠르지만 상

속을 받은 클래스에서 메소드를 새롭게 정의하면, 그 메소드만(같은 이름의 경우) 사용이 가능하므로 융통성이 적다.

2. 가상 메소드 (Virtual method): 지시어 virtual;

가상 메소드는 실행 시에 late binding 이라는 과정을 통해 실제로 호출될 메모리 주소가 결정된다. 내부적으로 가상 메소드가 참조되면 변수에 의해 참조되는 객체의 실제 클래스 형이 사용되는데, 이 작업이 가상함수 테이블(virtual method table(VMT), vtable)에 기록되어 있는 주소를 참조하여 이루어진다. 그러므로, 실제 참조되는 클래스 형에 따라서 여러가지 메소드가 호출될 수 있는 ‘다형성’이 구현될 수 있다.

3. 동적 메소드 (Dynamic method): 지시어 dynamic;

기본적인 사용법이나 목적은 가상 메소드와 동일하지만, 내부적인 처리 방법에 약간의 차이가 있다. 가상 메소드가 내부적으로 VMT 를 이용해서 메모리 주소를 참조하는데 비해 동적 메소드는 메소드를 지정하는 코드를 이용해서 메모리 주소를 찾게 된다. 그렇기 때문에 가상 메소드처럼 테이블을 직접 참조하는 방법보다 다소 느리게 동작하지만 메모리는 덜 사용하게 된다.

4. 추상 메소드 (Abstract method): 지시어 abstract;

일반적으로 메소드를 클래스에 선언할 때, 컴파일러는 메소드 프로시저가 유닛의 어느 부분인가 구현되어 있을 것으로 간주하게 된다. 그런데, 추상 메소드로 선언하면 컴파일러는 구현 부분이 자손 클래스에 있을 것으로 생각하고 이를 검사하지 않는다. 그런데, 만약 자손 클래스에서 이를 구현하지 않아서 추상 메소드가 호출되면 치명적인 에러가 발생하게 되므로 주의하기 바란다.

● 메소드 오버라이드 (method override)

override 지시어는 가상 또는 동적 메소드를 재정의할 때 사용된다. 재정의하는 방법은 다음과 같다.

type

```
TMyParent = class
    procedure AMethod; virtual;
end;
```

```
TMyClass = class(TMyParent)
```

```
procedure AMethod; override;
end;
```

- 클래스 메소드 (Class method)

보통 메소드는 클래스의 인스턴스에 대한 행동을 정의한다. 그런데, 어떤 때에는 클래스 자체에 대한 메소드가 있으면 할 경우가 있다. 이럴 때에는 클래스 메소드를 정의해서 사용한다. 클래스 메소드는 객체가 생성되지 않아도 사용할 수 있다.

클래스 메소드를 선언하려면 메소드 정의 부분에서 `class` 키워드를 앞에 붙여주면 된다.

- 메소드 오버로딩

델파이 4에서는 객체들이 같은 이름을 가진 여러 개의 메소드를 가질 수 있다. 이를 메소드 오버로딩이라고 하는데, 같은 이름을 가진 메소드들은 arguments의 type signature를 가지고 서로를 구별한다. 오버로드된 메소드는 키워드 오버로드로 표시된다. 그렇기 때문에, 객체들은 다음과 같이 다른 두 개의 생성자를 가질 수 있다.

```
constructor Create(AOwner: TComponent); overload; override;
constructor Create(AOwner: TComponent; Text: string); overload;
```

전역 함수와 프로시저역시 오버로드가 가능하다.

델파이 3까지만 해도 메소드 오버로딩을 지원하지 않았기 때문에, 생성자의 이름을 다르게 했어야 했다. 예를 들어 모든 윈도우 컨트롤은 `Create`, `CreateParented`의 2개의 생성자를 공통적으로 가지고 있었다. 델파이 4부터는 `Create`라는 메소드 이름을 가진 여러 생성자를 가질 수 있다.

오브젝트 파스칼의 이전 버전에서는 같은 이름의 메소드를 선언할 경우 조상 클래스의 메소드는 사용되지 않았다. 예를 들어, `Create` 메소드를 `Owner` 파라미터를 넘겨주지 않고 호출할 경우 과거에는 `TObject`에 선언된 생성자를 호출하지 않고 컴파일 에러를 발생시켰다. 메소드 오버로딩으로 이러한 문제들이 다소 변경되었는데, 다음의 코드를 살펴보자.

```
type
  A = class
    public
      procedure p(l: Integer); virtual;
  end;
  B = class(A)
```

```

    public
        procedure p(S: string);
    end;

var
    aB: B;
begin
    aB.p('one');      { works }
    aB.p(1);          { compile error! }
end;

```

여기에서 변경된 메소드의 파라미터를 대입할 경우에는 동작하지만, 원본 클래스의 메소드는 동작하지 않는다. 이를 해결하기 위해서는 overload 지시어를 사용하면 된다. 다음의 코드를 살펴보자.

```

type
    A = class
        public
            procedure p(l: Integer); overload; virtual;
        end;
    B = class(A)
        public
            procedure p(S: string); overload;
        end;

var
    aB: B;
begin
    aB.p('one');      { works }
    aB.p(1);          { now this one works too! }
end;

```

이 경우에는 동작하지만, 컴파일러가 경고를 한다. 이를 없애기 위해서는 상속받은 메소드에 다음과 같이 reintroduce 키워드를 지정하면 된다.

```

type

```

```

A = class
  public
    procedure p(l: Integer); overload; virtual;
end;
B = class(A)
  public
    procedure p(l: Integer; S: string); reintroduce; overload;
end;

```

## 동적 바인딩(Dynamic binding)과 다형성(Polymorphism)

파스칼의 함수와 프로시저는 기본적으로 정적 바인딩(static binding)을 이용하고 있다. 이것은 메소드 호출이 컴파일러나 링커에 의해 해석되며, 컴파일러나 링커는 이 호출문을 그 함수나 프로시저가 존재하는 특정 메모리 위치의 호출로 바꾸어 놓는다는 의미이다.

오브젝트 파스칼을 비롯한 객체지향 언어는 이와 다른 형태의 동적 바인딩을 지원한다. 이 경우에는 메소드의 실제 메모리 주소가 실행 중에 결정되는 것이다. 이런 특성을 이용해 다형성을 지원할 수 있게 되는데, 다형성이란 어떤 메소드의 호출문을 작성하고 그것을 변수에 대입해도, 어느 메소드가 호출될지는 그 변수에 관계된 객체의 데이터 형에 따라 달라지는 것이다. 다시 말해, 주어진 메소드에 대해 다수의 버전이 있을 수 있고, 그래서 하나의 메소드가 이러한 버전들 각각을 가리킬 수 있다.

## 정 리 (Summary)

이번 장에서는 오브젝트 파스칼의 OOP 적인 특성에 대해 간단하게나마 알아보았다. 사실 오브젝트 파스칼의 기능을 충분히 활용하려면 OOP 에 대한 전반적인 이해가 필수적이다. 이 책의 범위가 OOP 에 대해 심도 있게 논의할 수는 없지만, 이 책을 읽는 독자들은 반드시 따로 OOP 에 대해 공부해서 그 기법과 철학을 체득하길 바란다. 그렇게 익힌 OOP 에 대한 개념 들은 실제로 델파이 프로젝트를 진행하게 되면 소중한 지식으로 활용될 것이다. 다음 장에서는 OOP 의 대표적인 언어인 C++ 과 자바를 오브젝트 파스칼과 비교하는 시간을 가지도록 한다. 서로 다른 언어의 특징에 대해서 알아보는 것도 델파이를 잘 이해하는데 큰 도움이 될 것이다.



## 오브젝트 파스칼, C++ 그리고 자바의 특징 (Characterisitcs of Object Pascal, C++ and Java)

자바는 가장 일반적인 인터넷용 언어이며, C++은 아마도 가장 흔하게 사용되는 OOP 언어일 것이다. 이들과 델파이에서 사용되는 오브젝트 파스칼의 언어적인 측면에서의 비교를 해보면서 OOP 언어에 대한 감을 조금 더 높여 보자.

### OOP 언어의 특징

객체지향 프로그래밍(OOP)라는 것은 일종의 프로그래밍 테크닉으로, 가장 최초로 구현된 것은 스폴토크에서였다. 그러다가, 80 년대에 들어서면서 C++, Objective-C, 터보 파스칼, 에펠, Ada, 자바 등에 적용되면서 프로그래밍 언어에서 가장 중요한 기법으로 각광받게 되었다.

OOP 언어의 특징은 앞 장에서도 언급한 바 있지만, 다시 한번 이를 간단하게 정리해 보도록 하겠다.

1. 클래스라는 추상적인 데이터 형의 개념을 도입한다. 이를 이용해서 캡슐화, 모듈화, 데이터 추상화 등을 구현하게 된다.
2. 이미 존재하는 요소에서 그 기능을 상속할 수 있다. 이를 이용해서 몇가지 기능을 수정하거나, 추가만 하면 새로운 형태의 데이터 형을 정의할 수 있게 된다. 이를 통해 데이터의 세분화와 일반화가 가능해진다.
3. 마지막으로 다형성이라는 특징을 들 수 있다. 이 특징을 이용하면 같은 방법으로 서로 다른 클래스의 객체를 참조할 수 있다. 다형성을 이용하면 클래스의 재사용성이 보다 높아지고, 프로그램을 보다 쉽게 확장, 유지할 수 있게 된다.

위의 클래스, 상속성, 다형성의 3 가지 특징은 OOP 언어라고 불리는 언어라면 모두 지원해야 한다. 참고로, 위의 3 가지 특징 중 상속성과 다형성을 지원하지 않는 언어를 ‘객체기반(class-based)’ 언어라고 한다.

OOP 언어들은 위의 3 가지 특징을 모두 지원하지만, 데이터 형을 검사하는 방법과 프로그래밍 모델, 객체 모델 등에서 서로 차이가 있게 된다. 여기에 대해서 좀더 심층적으로 알아보기로 하자.

### 컴파일 vs 런타임 형 검사 (Compile-Time vs. Runtime Type Checking)

프로그래밍 언어를 구별할 때, 얼마나 데이터 형을 검사하는 방법이 까다로운가에 대한 것을 많이 고려하곤 한다. 그러니까 호출된 메소드의 존재 여부, 파라미터의 데이터 형, 배열

의 범위 측정 등을 얼마나 정확하고 까다롭게 하는지 등의 여부가 각 언어들끼리 조금씩 차이가 있다.

C++, 자바, 오브젝트 파스칼 모두 컴파일 시에 주로 데이터 형을 검사한다. 그 중에서도 자바가 가장 엄격하게 컴파일 시에 데이터 형을 검사하며, C++은 가장 덜하다. C++은 C 언어와의 호환성을 위해서 데이터 형 간의 구별이 덜 엄격한 편이다. C를 많이 써본 독자들은 아마도 float 형으로 선언한 변수의 값과 int로 선언한 변수의 값을 혼용하는 것을 많이 경험했을 것이다. 그에 비해 오브젝트 파스칼이나 자바에서는 정수형은 어디까지나 정수형이며, 실수형은 어디까지나 실수형이다. 이들 간의 호환성을 위해서는 엄격한 형변환이 필요하다.

자바 가상기계(VM)는 바이트 코드를 런타임에서 해석할 수 있기 때문에, 자바 언어가 컴파일 시에 형 검사를 하지 않는다고 생각하기 쉽지만, 그와 반대로 훨씬 더 엄격하게 형 검사를 한다는 것은 잘못 알기 쉬운 사실이다.

## 하이브리드 vs. 순수 OOP 언어

순수한 OOP 언어는 OOP라는 하나의 프로그래밍 모델만을 지원하는 언어이다. 즉, 개발자는 클래스와 메소드를 선언할 수 있지만, 과거에 사용해 왔던 일반적인 함수나 프로시저, 전역 변수 등은 사용할 수 없다.

자바는 OOP 프로그래밍 모델만을 지원하는 순수 OOP 언어이다. 자바 외에 에펠(Eiffel)과 스몰토크 등을 순수 OOP 언어라고 말할 수 있다. 일반적으로 이러한 순수 OOP 언어는 프로그래머로 하여금 반드시 OOP 모델을 사용하게 만들기 때문에 OOP를 처음 시작하는 사람에게 OOP를 익히게 하는데 유리하다. 그에 비해 C++과 오브젝트 파스칼은 OOP 프로그래밍 모델 이외에 전통적인 C와 파스칼 프로그래밍 모델을 같이 지원하는 하이브리드 언어이다.

하이브리드 언어는 호환성을 유지해야 하기 때문에, 원시적인 데이터 형들이 클래스 및 객체들과 혼용되는 형태를 가질 수 밖에 없다. 또한, 기존의 컴파일러 언어의 특성상 정적으로 프로시저나 함수를 사용할 수 있어야 하며, 사실상 이것이 주로 사용되는 방법이기 때문에 순수 객체지향 언어와 같이 동적으로 다형성을 지원하는 언어에 비해 다소 복잡한 함수 호출과 구현 방식을 가질 수 밖에 없다. 이렇게 서로 다른 프로그래밍 모델을 지원해야 하는 하이브리드 언어는 다소 몸집이 크고, 복잡하지만 유연성이라는 측면에서는 장점을 가지고 있다.

## 단순 객체 모델 vs. 객체 참조 모델 (Plain Object Model vs. Object Reference Model)

전통적인 OOP 언어는 프로그래머가 객체를 스택과 힙, 정적인 저장소에 생성할 수 있도록

허용하고 있다. 이러한 언어에서는 클래스 데이터 형의 변수가 메모리에 있는 객체와 직접적으로 연관된다. C++이 이러한 단순 객체 모델을 따르는 예가 된다.

그에 비해, 오브젝트 파스칼과 자바는 각각의 객체는 힙에 동적으로 메모리를 할당받게 되며, 클래스 데이터 형의 변수는 실제로 메모리 상의 객체가 아니라 객체에 대한 메모리의 핸들을 담고 있게 된다 (포인터와 비슷한 개념이다).

## 클래스와 객체, 그리고 참조 (Classes, Objects and References)

클래스란 일종의 데이터 형이며, 객체는 클래스가 실제로 메모리에서 인스턴스화 된 것이다. 오브젝트 파스칼과 C++, 자바는 모두 클래스에 바탕을 둔 언어이다. 각각의 객체들은 직접 정의되지 않고, 일단 클래스로 정의된 후 객체는 클래스에 대한 하나의 인스턴스로 나타나는 형태를 가지고 있다. 클래스는 객체와 1:n의 관계를 가지게 된다. 클래스는 코드를 메소드의 형태로 저장하며, 여러 객체들과 이를 공유하게 된다. 어쨌든 각 객체는 자신의 데이터 필드를 저장하게 되며, 이것으로 클래스와 객체가 구별된다.

객체 내의 필드는 원시적인 데이터 형이거나 객체일 수 있다. 정상적으로 각 객체는 클래스에서 선언된 데이터 필드의 복사본을 자신의 것으로 가지고 있다.

이들의 관계가 각각의 언어에서 어떻게 특징지어 지는지 살펴보도록 하자.

### ● C++

C++에서 MyMethod 라는 메소드를 가진 MyClass 란 클래스가 있다고 하면, 다음과 같이 객체를 사용할 수 있다.

```
MyClass Obj;  
Obj.MyMethod();
```

첫째 줄의 선언으로 Obj 라는 MyClass 형의 객체를 얻게 된다. 이때 이 객체에 대한 메모리는 전형적으로 스택에서 할당받게 되므로, 바로 그 다음 줄에서 이 객체를 사용할 수 있게 된다.

### ● 자바

자바에서는 객체에 해당하는 핸들에 대한 메모리를 할당받기 때문에, 다음과 같이 사용해야 한다.

```
MyClass Obj;
```

```
Obj = new MyClass();  
Obj.MyMethod();
```

이처럼 객체를 사용하기 전에 ‘new’ 키워드를 이용해서 객체에 대한 메모리를 할당받아야 한다. 물론, 위의 문장을 아래와 같이 2 줄로 줄여 쓸 수 있다.

```
MyClass Obj = new MyClass();  
Obj.MyMethod();
```

## ● 오브젝트 파스칼

오브젝트 파스칼도 근본적으로는 자바와 비슷하지만, 문법의 구조가 다음과 같이 다소 차이가 날 뿐이다.

```
var  
  Obj: MyClass;  
begin  
  Obj := MyClass.Create;  
  Obj.MyMethod;
```

위의 비교에서, 객체참조 모델과 단순객체 모델의 차이점을 이해할 수 있을 것이다. 프로 그램의 입장에서 보면 객체참조 모델이 다소 복잡해 보인다. 그렇지만, 단순객체 모델에서는 각 변수가 직접 객체를 가리키기 때문에, 객체를 지정하거나 참조하기 위해서 포인터를 사용해야 하는 빈도가 늘어날 수 밖에 없다. 그에 비해 객체참조 모델을 사용하는 언어에서는 객체를 참조할 때 포인터를 쓰지 않아도 되며, 각각의 객체를 직접 제어할 필요도 없다. 이런 이유로 자바에서는 포인터를 지원하지 않는다.

## 메모리 모델

오브젝트 파스칼과 C++은 모두 기존의 record(오브젝트 파스칼), struct(C++) 데이터 형에다가 클래스의 VMT에 대한 포인터를 추가한 것을 클래스로 메모리에서 관리하고 있다. C++의 경우 프로그램 변수를 할당하는데 static, automatic, dynamic의 3가지 저장 클래스를 지원한다. Static 객체는 프로그램의 데이터 세그먼트에 생성되며, automatic 객체는 스택에, dynamic 객체는 힙에 생성된다. Static storage는 전역 객체에 사용되며, automatic storage는 로컬 객체에, dynamic storage는 런타임에서 생성된 객체에 사용된다.

C++은 객체를 생성하는 프로세스를 메모리의 할당과 초기화하는 크게 두 과정으로 나눈다. 메모리 할당은 new 메소드를 이용해서 이루어진다. C++의 경우에는 new 메소드를 오버라이드 하는 경우도 많은데, 이를 통해 사용자가 메모리를 할당하는 저수준 프로그래밍 루틴을 직접 작성할 수 있다.

오브젝트 파스칼은 dynamic 객체 만들 지원한다. 로컬, 전역 객체가 선언될 수 있지만, 이들은 단지 레퍼런스로서 데이터 세그먼트나 스택에 저장된다. 객체 자체는 반드시 직접 생성해서 항상 힙에 저장해야 한다. 오브젝트 파스칼은 기본적으로 사용자 정의 메모리 할당 루틴을 제공하지는 않는다.

그렇지만, 오브젝트 파스칼과 C++은 모두 자동 garbage collection 은 지원하지 않으므로, 프로그래머가 각 객체를 동적으로 생성한 경우 이를 제거할 필요가 있다.

자바의 경우는 dynamic 객체와 메모리 관리를 동적으로 해준다. 또한, garbage collection 을 해주므로 동적으로 생성한 객체에 대해 언어 차원에서 관리를 해준다.

## 생성자 (Constructors)

그러면, 각 언어의 생성자에 대해서 알아보자. 이를 이용해서 객체를 초기화하게 된다.

- C++

C++은 클래스와 같은 이름의 생성자를 가진다. 만약에 프로그래머가 클래스에 생성자를 정의하지 않으면, 컴파일러가 디폴트 생성자를 클래스에 추가한다. 메소드 오버로딩을 이용해서 여러 개의 생성자를 가질 수 있다.

- 자바

C++과 비슷하게 사용하지만, ‘initializer’라고도 불린다. 이것의 의미는 객체를 실제로 생성하는 것은 자바의 VM 이 하는 일이고, 생성자에 써 넣은 코드는 단순히 새롭게 생성된 객체를 초기화하는 역할을 한다는 의미이다.

- 오브젝트 파스칼

오브젝트 파스칼에서는 constructor 라는 키워드를 사용해서 생성자를 정의한다. 메소드 오버로딩을 지원하지 않지만, 생성자가 여러 가지 이름을 가질 수 있기 때문에 여러 개의 생성자를 가질 수 있다. 오브젝트 파스칼에서는 각각의 클래스가 디폴트 Create 생성자를 가지고 있게 되는데, 이 생성자는 기본적인 기초 클래스에서 상속받게 된다.

## 파괴자 (Destructor)

파괴자는 생성자와 반대되는 일을 한다. 즉, 객체가 메모리에서 해제될 때 호출되며 생성자에 의해 할당된 리소스를 해제하는 역할을 하게 된다.

- C++

C++의 파괴자는 객체가 범위를 벗어나거나, 동적으로 할당된 객체를 삭제할 때 자동으로 호출된다. 모든 클래스는 오직 하나의 파괴자를 가지게 된다.

- 자바

자바는 파괴자를 가지지 않는다. 참조되지 않는 객체는 백그라운드에서 실행되고 있는 가비지 컬렉션(garbage collection) 알고리즘을 통해 자동으로 파괴되며, 객체가 파괴되기 전에 finalize() 메소드를 호출한다.

- 오브젝트 파스칼

오브젝트 파스칼의 파괴자는 C++과 비슷한 역할을 한다. 오브젝트 파스칼에서는 표준 가상 파괴자인 Destroy 를 이용하는데, 이 파괴자는 Free 메소드에 의해 호출된다. 모든 객체가 동적으로 처리되기 때문에, 일단 객체가 생성되면 그 객체의 소유주(owner) 객체가 파괴될 때 자동으로 Free 메소드가 호출된다. 이론적으로 여러 개의 파괴자를 선언할 수 있다.

## 클래스 캡슐화

클래스 캡슐화의 레벨을 지정하는 지시어로 private, protected, public 이 사용된다. 그러나, 약간의 차이가 있으니 이를 살펴 보자.

- C++

C++에서 friend 키워드를 사용하면 캡슐화의 범위를 벗어날 수 있다. 기본적으로 클래스의 캡슐화 정도는 private 에 준하며, 구조체의 경우에는 public 에 준한다.

- 자바

자바에서는 디폴트로 각 요소들이 friend 로 간주된다. 즉, 각 요소 들은 같은 패키지 안에 있거나 같은 소스 코드 파일 내에 있으면 다른 클래스에서도 접근할 수 있다. protected 키워드로 지정한 경우에 서브 클래스와 같은 패키지 내의 다른 클래스에서 접근할 수 있다. 즉, private protected 를 복합적으로 사용할 때 C++ 의 protected 와 같은 의미가 된다.

- 오브젝트 파스칼

자바와 비슷하다. private, protected 키워드는 다른 유닛일 경우에는 접근할 수 없지만, 같은 유닛(같은 소스 코드 파일)에 있으면 접근이 가능하다. 델파이의 경우에는 published 키워드를 제공하는데, 이 키워드를 이용하면 그 요소에 대한 RTTI 정보가 생성된다. 그 밖에, 오브젝트 파스칼은 디자인 시에 필드의 값을 보고, 편집할 수 있는 프로퍼티를 제공한다. 프로퍼티는 간접적으로 객체의 필드에 접근하기 때문에, 캡슐화 개념을 충실하게 지키고 있다. 프로퍼티는 데이터 필드에 대한 단순한 레퍼런스일 수도 있고, 참조 무결성을 유지하는 등의 보다 여러가지 조작을 가할 수 있는 함수를 호출하는 것일 수도 있다.

## 파일, 유닛 그리고 패키지

파일에 소스 코드가 조직되어 있는 방법에 많은 차이점이 있다. C++ 컴파일러는 파일에 대한 정보를 무시하는데 비해, 자바와 오브젝트 파스칼의 경우에는 파일 단위가 일종의 모듈 단위로 인식된다.

- C++

C++ 프로그래머는 클래스의 정의 부분을 헤더 파일에 저장하고, 실제 메소드의 정의는 분리된 코드 파일에 저장하는 경향이 있다. 이런 파일 들은 대부분 같은 이름을 가지고, 다른 확장자를 가지게 된다. 이와 같이 C++에서는 파일 단위에 대한 특별한 제한점이 존재하지 않는다. 이는 다르게 말하면, 컴파일러가 서로 다른 파일에 존재하는 여러가지 선언들에 대해서 별로 신경을 쓰지 않고 컴파일을 하기 때문에, 링커가 많은 일을 해야 한다는 것을 의미하며 이것이 C++의 링킹 속도가 저하되는 원인이라고 할 수 있다.

- 자바

자바에서는 각각의 소스 코드 파일이나 컴파일 단위가 서로 분리되어 있다. 그렇기 때문에, 컴파일 유닛을 하나의 그룹으로 묶을 수가 있는데, 이를 패키지라고 한다. 클래스를 선언하면 클래스의 메소드에 대한 코드도 같이 써주어야 한다. 'import' 키워드를 이용해서 다른 파일의 내용을 읽을 수 있는데, 이 때에는 그 파일에서 public 으로 선언된 부분 만을 읽

어울 수 있다.

- 오브젝트 파스칼

오브젝트 파스칼의 경우에는 각각의 소스코드 파일을 유닛이라고 한다. 이러한 유닛은 크게 나누어 interface, implementation 이라는 2 개의 파트로 이루어져 있다. Interface 섹션은 클래스의 정의와 메소드에 대한 선언 부분을 포함하며, implementation 섹션에는 interface 섹션에서 선언한 메소드의 구현 부분이 위치하게 된다. 다른 파일에서 선언된 interface 부분을 ‘uses’ 키워드를 통해 접근할 수 있다.

또다른 특징을 언급하자면, 자바나 오브젝트 파스칼의 컴파일러는 컴파일된 파일(텔파일의 경우 .dcu 파일)에서 선언부분의 내용을 읽어올 수 있는데 비해, C++에서는 이러한 모듈 구조를 허용하지 않는다.

## 클래스 메소드와 데이터

OOP 언어는 일반적으로 특정 객체에만 해당하지 않고, 클래스에 전반적으로 적용시킬 수 있는 메소드와 데이터를 허용한다. 클래스 메소드는 클래스의 객체와 클래스 자체에서 호출할 수 있으며, 클래스 데이터는 각각의 객체에 의해 복제되지 않고, 공통으로 사용되는 데이터를 말한다. 이들을 다른 말로 정적 메소드(static method), 정적 데이터(static data)라고도 한다.

- C++

C++의 클래스 메소드와 데이터는 ‘static’ 키워드에 의해 지정된다. 클래스 데이터는 특정한 선언문에 의해서 초기화 되어야 한다.

- 자바

자바는 C++과 마찬가지로 ‘static’ 키워드를 이용해서 클래스 메소드와 데이터를 지정할 수 있다. 클래스 메소드는 매우 자주 사용되는데, 이는 자바가 전역 함수를 허용하지 않기 때문이다. 클래스 데이터는 클래스의 선언부에서 직접 초기화될 수 있다.

- 오브젝트 파스칼

오브젝트 파스칼은 클래스 메소드만 지원한다. 클래스 메소드는 ‘class’ 키워드에 의해 지



정될 수 있다. 클래스 데이터는 직접 지원하지 않고, 대신 클래스를 정의한 유닛에 private 전역 변수를 추가해서 이 기능을 대치한다.

## 전체 클래스의 조상

일부의 OOP 언어에서는 최소한 하나의 기초 클래스를 가지는 경우가 있다. 이러한 클래스는 모든 클래스에서 동일하게 가져야 하는 기본적인 특징 들을 가지고 있다. 즉, 다르게 말하면 모든 클래스는 가장 기본적인 기초 클래스를 상속받는 것이다. 이러한 개념은 스톡토크에서부터 시작된 것으로 비교적 많은 OOP 언어에서 채택되고 있는 방식이다.

- C++

C++은 기본적으로 이러한 개념을 지원하지 않는다. 그렇지만 C++에 기초한 많은 어플리케이션 프레임워크(MFC, OWL 등)에서는 이러한 기초 클래스 개념을 받아들이고 있다. 예를 들어, MFC의 경우 CObject 클래스가 기초 클래스로 사용된다.

- 자바

자바의 모든 클래스는 Object 클래스에서 상속받는다.

- 오브젝트 파스칼

오브젝트 파스칼은 TObject 클래스를 공통의 조상으로 가진다. 오브젝트 파스칼은 기본적으로 다중 상속을 지원하지 않기 때문에, 상당히 커다란 상속 트리를 가지게 된다. TObject 클래스는 RTTI를 사용할 수 있으며, 그 밖에 몇 가지 기본적인 특징을 가진다.

## 하위형 호환성 (Subtype Compatibility)

모든 OOP 언어가 형 검사를 엄격하게 하는 것은 아니지만, 우리가 지금 논의하고 있는 세 가지 언어는 비교적 형 검사가 엄격한 편이다. 이는 기본적으로 서로 다른 클래스의 객체들간의 형-호환성(type-compatibility)이 보장되지 않는다는 것이다. 이러한 규칙의 예외가 있는데, 특정 클래스를 상속한 클래스의 객체는 부모 클래스와 데이터 형의 호환성이 인정된다 (역은 성립하지 않는다.). 이러한 하위형 호환성은 다형성(polymorphism)과 late binding을 지원하는데 중요한 역할을 한다.

- C++

C++의 경우에는 이러한 하위형 호환성이 포인터와 참조(reference)에 대해서만 허용된다. 서로 다른 객체는 서로 다른 크기의 메모리를 사용하기 때문에, 이들 객체에 대한 직접적인 하위형 호환성을 보장할 수가 없다.

- 자바, 오브젝트 파스칼

이들은 모든 객체에 대해서 하위형 호환성을 지원한다. 이것이 가능한 이유는 앞에서도 설명했듯이 이들이 모두 객체참조 모델을 사용하기 때문이다. 오브젝트 파스칼의 예를 들면, 모든 객체는 TObject 클래스와 호환된다.

## 다형성(Polymorphism)과 late 바인딩

오브젝트 파스칼과 C++은 메소드를 디스패치할 때 정적, 동적 바인딩을 모두 지원한다. 정적 바인딩은 다형성을 지원하지 않으며, 컴파일 시에 동작하며 전통적인 함수 호출에서 사용되는 방식이다. 정적 메소드의 주소는 링커에 의해 코드 세그먼트에 직접 저장된다. 이에 비해 동적 바인딩 또는 가상 메소드는 다형성을 지원한다. 이를 위해 객체의 정확한 데이터 형을 모르는 런타임에서 동작하게 된다.

오브젝트 파스칼과 C++은 모두 가상 메모리 테이블(VMT)를 통해 가상 메소드의 주소를 저장한다. 각 클래스는 자신의 VMT를 가지게 되며, VMT에 있는 함수 주소의 배열을 이용한다. 가상 메소드는 이 주소를 직접 이용하므로 동작하는 속도는 그렇게 느리지 않다.

오브젝트 파스칼과 C++은 모두 명시적으로 virtual로 선언한 메소드만 다형성을 지원한다. 이는 하이브리드 언어라면 조상 언어의 호환성을 유지해야 하며, 성능 상의 문제 때문에 어쩔 수 없는 것이라고 생각된다.

부모 클래스의 메소드를 새롭게 정의한 클래스가 있을 때, 일반적인 객체에 대하여 그 메소드를 호출할 때 적절한 클래스의 메소드가 호출된다면 무척 편리할 것이다. 이러한 특징을 다형성이라고 한다. 이를 지원하려면, 위에서 지원한 하위형 호환성이 매우 중요한 역할을 하게 된다. 컴파일러는 다형성을 지원하기 위해서 late binding이라는 기법을 사용하게 되는데, 이것은 특정 함수를 호출하지 않고 런타임에서 객체가 실제로 어떤 클래스의 함수를 호출하게 될지를 알아낸 후에 호출될 함수를 결정하는 방식이다.

- C++

C++에서는 이러한 late binding이 가상 메소드에 대해서만 허용된다. 가상 메소드가 아닌 메소드는 late binding을 지원하지 않는다.

- 자바

자바의 경우에는 ‘final’ 키워드로 지정하지 않는한 모든 메소드가 late binding 을 지원한다. Final 메소드는 재정의될 수 없고, 이들은 다른 메소드보다 빠르게 동작한다. 그러니까, 기본적으로 C++ 은 early binding 이 디폴트이며, 자바는 late binding 이 디폴트인 것이다. 다르게 말하자면, C++ 은 효율성을 위해서 OOP 모델을 희생하는 경우가 많다.

- 오브젝트 파스칼

오브젝트 파스칼의 경우에는 C++ 과 마찬가지로 디폴트는 early binding 이다. 그런데, 오브젝트 파스칼은 ‘virtual’ 키워드로 지정하는 가상 함수 외에 ‘dynamic’ 키워드로 지정하는 동적 함수가 지원된다. 그러나, 이들은 기본적인 개념으로 보아서는 동일한 것이다. 오브젝트 파스칼의 경우에는 ‘override’ 키워드를 사용해야만 이들 가상 함수를 재정의할 수 있다. 이러한 ‘override’ 키워드의 의미는 이 키워드로 지정한 메소드만 컴파일러가 다시 검사하게 된다는 것이다. 이런 방법으로 비교적 효율적인 퍼포먼스를 유지할 수 있다. 또한, 오브젝트 파스칼에서는 가상 생성자(virtual constructor)를 정의할 수 있다.

오브젝트 파스칼은 가상 메소드 이외에 동적 메소드를 지원한다. 이 메소드는 원래 윈도우의 메시지를 사용하기 위해 고안된 것이다. 즉, 수백 개가 넘는 윈도우 메시지를 처리할 때에는 이들 각각에 대한 VMT 를 관리한다는 것은 메모리 공간의 낭비가 될 수 있으므로, 동적 메소드 테이블(Dynamic method table, DMT)을 통해 오버 라이드된 메시지 핸들러에 대한 주소만 관리함으로써 이러한 오버헤드를 줄일 수 있다. 그렇지만, 아무래도 수행능력이라는 측면에서는 다소 핸디캡을 가지고 있다고 보면 된다.

## 추상 메소드와 클래스

비교적 복잡한 클래스 구조를 만들 때에는, 프로그래머가 다형성을 지원하는데 유리하도록 실제로 구현되지 않는 메소드를 선언할 필요가 있을 수 있다. 이러한 메소드를 추상 메소드(abstract method)라고 하며, 이러한 추상 메소드를 하나 이상 가지고 있는 클래스를 추상 클래스라고 한다.

- C++ , 자바

C++ 의 추상 메소드는 순수한 가상 함수라고 생각하면 된다. 추상 클래스는 하나 이상의 추상 메소드를 가진 클래스로, 이러한 추상 클래스 객체는 생성할 수 없게 되어 있다. 즉, 이런 추상 클래스를 상속받아서 추상 메소드를 실제로 구현한 클래스만 객체를 생성할 수 있다. 자바의 경우에도 C++ 과 마찬가지로 추상 클래스의 인스턴스는 생성할 수 없다.

- 오브젝트 파스칼

오브젝트 파스칼의 경우에는 추상 메소드를 가지고 있는 추상 클래스의 인스턴스를 생성하는 것이 가능하다. 그렇기 때문에, 프로그램이 추상 메소드를 호출할 가능성도 있는데 이렇게 되면 런타임 에러가 발생하게 된다.

## 다중상속과 인터페이스(Interfaces)

일부의 OOP 언어는 하나 이상의 기초 클래스를 상속 받을 수 있다. 이를 다중상속(multiple inheritance)라고 한다. 그에 비해 다른 언어에서는 오직 하나의 클래스만 상속받을 수 있지만, 옵션으로 다중 인터페이스나 순수한 추상 클래스(클래스의 순수한 추상 함수로만 이루어진 경우)들을 상속 받아 다중상속의 기능을 대체한다.

- C++

C++은 다중상속을 지원하는 언어이다. 이점은 장점도 단점도 될 수 있다. 다중상속의 장점에 대한 논의는 이 책의 범위를 넘기 때문에 생략하도록 하겠다.

- 자바, 오브젝트 파스칼

자바와 오브젝트 파스칼은 공히 다중상속을 지원하지 않고, 다중 인터페이스를 지원한다. 이를 이용해서 다형성을 구현할 수 있으며, 이런 특징을 바탕으로 COM 모델에 적합한 언어환경이 지원된다. 오브젝트 파스칼은 자바보다 더욱 COM에 가까운 인터페이스 모델을 가지고 있다.

## RTTI (Runtime Type Identification/Information)

형 검사가 엄격한 OOP 언어는 컴파일러가 이러한 형 검사를 해주어야 한다. 그러므로, 실행 중인 프로그램에는 클래스와 데이터 형에 대한 정보가 별로 필요가 없다. 그러나, 어떤 경우에는 이러한 데이터 형에 대한 정보가 필요한 경우가 있는데, 이런 경우를 위해 각 언어들은 정도에 차이는 있지만 RTTI를 지원하고 있다.

- C++

C++ 언어는 본래 RTTI를 지원하지 않는다. 그러나, 이 개념의 지원을 위해 downcast

(dynamic\_cast)라는 형태로 일부 기능이 추가 되었다. 이를 이용해서 각 객체에 대한 데이터 형을 확인하거나, 2 개의 객체가 같은 클래스인지 검사할 수 있다.

- 자바

자바는 기본적인 기초 클래스로 Object 클래스가 제공된다. 이 클래스를 통해 클래스에 대한 정보를 추적할 수 있다. Object 클래스의 getClass() 메소드를 사용해서 메타클래스(클래스를 설명하는 클래스의 객체) 정보를 얻을 수 있고, getName() 함수를 사용해서 특정 문자열을 클래스의 이름으로 지정할 수 있다. 또한, instanceof 연산자를 사용할 수도 있다. 자바의 1.0 버전은 클래스에 대한 RTTI 를 광범위하게 지원하고 있지 않지만, 컴포넌트와 비주얼 환경의 발달로 이러한 RTTI 를 많이 지원하게 되었는데, 이것이 자바 빈스(Java Beans)이다.

- 오브젝트 파스칼

오브젝트 파스칼은 가장 광범위한 RTTI 정보를 제공한다. 이를 이용해서 단순한 데이터 형 검사와 downcast 를 할 수 있으며 (is 와 as 연산자를 사용한다), 더 나아가서는 published 로 선언된 요소들을 새로운 RTTI 정보로 등록할 수도 있다. 프로퍼티와 스트리밍 메커니즘(폼 파일 등의)과 오브젝트 인스펙터로 표현되는 델파이의 환경은 기본적으로 이런 RTTI 가 있기에 가능한 것이다. 델파이 클래스의 기초 클래스인 TObject 클래스에는 ClassName, ClassType 메소드가 있는데, ClassType 메소드를 사용해서 특정 객체의 클래스에 대한 참조 값을 돌려 받을 수 있다. 이 메소드를 통해서 반환되는 클래스 형은 TClass 로 이 클래스는 TObject 의 클래스로 선언되어 있는데, 이는 이렇게 TClass 형으로 반환된 클래스는 반드시 사용되기 전에 특정 클래스로 형 변환되어야 한다는 것을 의미한다.

## 예외처리

예외처리란 프로그램의 에러 처리 부분을 보다 손쉽게 하기 위해, 언어에서 제공하는 표준 메커니즘을 말한다. 예외처리 방식은 언어들 마다 비슷하지만, 내부 동작에는 다소간의 차이가 존재한다.

- C++

C++에서는 throw 라는 키워드를 사용해서 예외를 발생시키며, try 키워드로 예외를 처리하게 될 블록을 설정하고, catch 키워드로 실제로 예외를 처리할 루틴을 작성하게 된다. C++은 예외처리를 할 때 파괴자를 호출해서 스택에 할당된 객체의 메모리를 해제한다.

- 자바

자바는 C++ 과 같은 키워드를 사용한다. 자바에는 finally 키워드가 있는데, 이 키워드는 객체참조 모델을 지원하는 언어에는 대부분 존재한다. 자바는 기본적으로 가비지 컬렉션을 하기 때문에 이러한 finally 키워드를 사용해서 메모리 이외에 리소스를 해제하는 것을 제한한다. 자바는 예외를 일으킬 수 있는 모든 함수와 예외 클래스가 기본적으로 잘 대응되어야 하며, 이를 컴파일러가 일일이 검사한다. 프로그래머에게는 다소 부담되는 일이지만, 오류가 적은 프로그램을 만드는 데에는 상당히 강력한 무기가 된다. 모든 자바의 예외 객체들은 Throwable 클래스에서 상속받는다.

- 오브젝트 파스칼

오브젝트 파스칼은 raise, try, except 키워드를 사용한다. C++ 과의 차이점은 기본적으로 스택에 객체에 대한 메모리가 할당되어 있지 않기 때문에, 스택을 처리할 필요가 없다는 것이다. 또한, 자바와 마찬가지로 finally 키워드를 지원한다. 델파이의 모든 예외 클래스는 Exception 클래스를 상속한다.

## 그 밖의 특징들

- C++

C++ 은 연산자 오버로딩(operator overloading)을 지원한다. 또한, 메소드 오버로딩도 지원하지만 이것은 자바와 델파이 4 에서도 지원된다. 그리고, C++ 은 프로그래머가 전역 함수를 오버로드할 수도 있으며, 형변환을 담당하는 메소드를 정의해서 형변환 연산자도 오버로딩할 수 있다. 그리고, 템플릿이라는 개념을 지원하여 일반적인 클래스의 재사용성을 높일 수 있다.

- 자바

자바는 멀티쓰레딩을 언어에서 지원한다. 객체와 메소드는 동기화(synchronization) 메커니즘을 지원하는데, 2 개의 동기화된 메소드는 같은 클래스에서 동시에 수행될 수 없다. 이렇게 새로운 쓰레드를 생성하려면 단순히 Thread 클래스를 상속받아서 run() 메소드를 오버라이드하거나, Runnable 인터페이스를 구현하면 된다. 그리고, 백 그라운드에서 가비지 컬렉션을 해주기 때문에, 다소간의 퍼포먼스의 희생을 감수해야 하지만 보다 완벽한 메모리 관리가 가능하다. 그 밖에 자바는 포터블 바이트-코드 아이디어를 채용했기 때문에, 여러

가지 플랫폼에 적용되기 적합한 형태를 가지고 있다.

## ● 오브젝트 파스칼

오브젝트 파스칼은 클래스 참조를 지원하기 때문에, 메소드 포인터를 아주 쉽게 사용할 수 있다. 이러한 메소드 포인터는 이벤트 모델의 기초가 되며, 이를 프로퍼티로 사용할 수 있다. 프로퍼티는 메소드가 데이터에 접근하는 방법을 숨겨주는 방법으로 사용되는데, 데이터를 직접 읽거나 쓸 수도 있고, 접근 메소드(access method)를 사용해서 데이터를 조작하는 방법도 가능하다. 데이터에 접근하는 방법을 바꾸더라도 코드를 호출하는 방법을 바꿀 필요가 없기 때문에, 재사용성이라는 측면에서 대단히 유용하다. 다르게 말하면, 다른 어떤 OOP 언어보다도 강력한 캡슐화 방법을 지원하는 것이다. 또한, 델파이 4에서는 그동안 지원하지 않았던 메소드 오버로딩과 파라미터의 디폴트 값도 지원하므로 보다 강력한 언어적 특성을 가지게 되었다.

## 정 리 (Summary)

이번 장에서는 OOP 언어로는 가장 많이 사용되는 대표적인 세가지 언어에 대해서 비록 수박 겉핥기에 가까운 방법으로나마 살펴 보았다. 이들 언어는 기본적으로 OOP의 개념을 공통적으로 구현하고 있지만, 그 구현의 목적과 방향성에 다소간의 차이가 있다는 것을 알 수 있을 것이다. C++은 다소간의 복잡성을 가지고, OOP의 기본적인 개념에 배치되는 여러가지 모습을 가지고 있지만 파워와 유연성이라는 측면에 초점을 맞춘 언어라고 할 수 있고, 델파이는 쉽고, 비주얼 환경과 윈도우 환경에 적합하도록 설계 되었지만 파워도 포기하지 않은 언어이다. 자바는 이식성에 주안점을 두고, 스피드를 다소 희생하도록 설계 되었다.

오브젝트 파스칼과 C++에 대한 차이를 설명할 때에는 앞에서 둘러본 여러가지 항목들의 차이도 중요하겠지만, 보다 중요한 것은 기본적으로 이들 언어의 조상이 된 파스칼과 C가 디자인되고 사용된 배경과 목적에서 차이점을 찾아볼 수 있다.

파스칼은 명확하고, 안전하며 동시에 모듈화가 용이하도록 디자인된 언어인데 비해, C는 노력을 적게 들이면서도 가능한 저수준의 메모리를 쉽게 접근할 수 있도록 디자인 되었다. 파스칼은 데이터 추상화와 다중 레벨의 모듈화를 직접 지원하는 고수준의 언어이지만, C는 중간 레벨로서 기계와 고수준 언어의 중간 수준을 목표로 만들어졌다. 파스칼은 프로그래머를 보호하기 위해서 만들어진 언어인데 비해 C는 최대한 프로그래머를 믿고 만들어진 언어이다. 이러한 파스칼과 C의 바탕에서 객체지향 언어의 장점을 접합시킨 것이 바로 오브젝트 파스칼과 C++인 것이다. 자바는 여기에 비해 직접적인 조상이 되는 언어가 없는 순수한 객체지향 언어이다.

여기서 확실히 말할 수 있는 것은 이러한 언어의 특징들이 실제 세계에서 사용되는 개발

환경을 좌우하지는 않는 다는 것이다. 실제로 중요한 것은 운영체제와 최근에 불어닥치는 인터넷 세계, 그리고 Win32API, 액티브 X 등의 모든 주변환경들과 이들의 관계, 개발자들이 선택한 마켓 셰어 등이라고 할 수 있다. 아무리 훌륭한 언어라도 실제로 사용되지 않는 경우는 부지기수이다. 예를 들어, 오브젝트 파스칼과 자바의 모델인 에펠(Eiffel)은 대단히 훌륭한 OOP 언어이지만 대학에서나 학문적으로 사용될 뿐 실제 시장에서는 외면받고 있는 언어이다.

결국, 어떤 언어가 좋은가 ? 하는 소모적인 논쟁 보다는 실제로 사용할 언어의 특징을 잘 파악하고, 이를 잘 이용하는 것이 좋은 프로그램을 만들 수 있는 기반이 될 것이다. 그런 측면에서 이번 강의 소기의 목적이 달성되었으면 하는 것이 필자의 소박한 바람이다.



## 오브젝트 파스칼의 깊은 곳

### (Advanced Concepts of Object Pascal)

이번 장에서는 비교적 알려지지 않거나, 오브젝트 파스칼의 특성을 최대한 활용할 수 있는 문법과 특징에 대해서 알아볼 것이다. 이번 장을 통해서 오브젝트 파스칼에 숨겨져 있는 역량을 최대한 끌어내어 활용할 수 있도록 해보자.

#### Sender 파라미터의 이용

서로 다른 컴포넌트 들에 대한 서로 다른 동작을 하나의 이벤트 핸들러로 처리하고 싶을 때가 있다. 이럴 때에는 Sender 파라미터를 이용하면 도움이 된다.

Sender 파라미터는 델파이에게 이벤트를 실제로 받은 컴포넌트가 무엇인지를 알려주는 역할을 한다. 다음의 코드는 버튼에 따라서 폼의 캡션에 타이틀을 표시하기도 하고, 아무것도 보여주지 않기도 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Sender = Button1 then
        Form1.Caption := '연습입니다 !';
    else Form1.Caption := '';
end;
```

이 코드에 의해 Button1 을 클릭한 경우에는 ‘연습입니다 !’를 보여주지만, 다른 컨트롤에서 이 이벤트 핸들러를 사용한 경우에는 캡션이 비게 된다.

#### 팀 개발 환경에서의 객체 공유

델파이를 이용해서 공동으로 프로젝트를 개발하는 경우, 팀의 멤버 들이 동시에 접근할 수 있는 디렉토리를 지정할 필요가 있다. 그리고 나서, 개발자가 아이템을 객체 저장소(object repository)에 저장할 경우 delphi32.dro 파일이 생성된다. 새로운 delphi32.dro 텍스트 파일에는 공유하고자 하는 객체의 포인터가 포함된다. 공유 저장소(shared repository) 디렉토리를 지정하려면 다음과 같이 하면 된다.

1. Tools|Environment Options 메뉴를 선택한다.

2. Preferences 페이지의 Shared Repository 패널에 커서를 위치시킨다.
3. Directory 에디트 박스에서 공유 저장소로 지정하고자 하는 디렉토리의 이름을 적어 넣는다.

공유 디렉토리의 위치는 윈도우 레지스트리에 저장된다. 마찬가지로 텔파이의 Environment Options 대화 상자에서의 변경 사항도 레지스트리에 저장된다. 객체 저장소의 아이템을 팀 멤버들이 서로 공유하려면, 모든 멤버들의 Environment Options 메뉴의 Directory 설정이 같은 위치를 지정하고 있어야 한다.

## 예외 처리 (Handling exceptions)

텔파이는 어플리케이션의 에러를 쉽게 처리하여 견고한 어플리케이션의 개발이 가능하도록 효율적인 예외 처리 루틴을 제공한다. 그러면, 텔파이의 예외 처리 루틴에 대해서 알아보도록 하자.

### ● 코드 블록의 보호 (Protecting blocks of code)

어플리케이션을 견고하게 제작하려면, 에러가 발생하더라도 이를 자연스럽게 처리할 수 있어야 한다. 여기서 중요한 것은 과연 어느 곳의 코드 블록에서 에러가 발생할 것인지 예측하는 것이다. 일단 이 부분을 발견하게 되면, 처리 방법을 고안하여야 하는데 이 때에는 그 중요성에 따라서 간단한게 메시지 박스로 처리할 수도 있고, 다른 처리 방법을 강구할 수도 있다. 중요한 것은 시스템 리소스나 데이터의 유실을 가져올 수 있는 에러가 발생할 가능성이 있는 부분에는 반드시 적절한 예외 처리를 해 주어야 한다는 것이다.

이 때 예외를 처리할 수 있도록 지정된 블록을 보호된 블록(protected block)이라고 한다.

### ● 예외 처리 방법

에러가 발생하면, 어플리케이션은 예외 객체를 생성한다. 예외를 처리하는 방법은 크게 나누어 클린업(clean up) 코드를 작성하는 방법과 직접 예외를 처리해 주는 방법으로 나눌 수 있다.

클린업 코드를 실행하는 것이 가장 간단한 방법이다. 이 경우 에러를 발생시킨 조건을 해결하지는 못하지만, 어플리케이션이 불안정한 상태로 가는 것을 막는 역할을 한다. 즉, 할당된 리소스를 해제하는 것을 예로 들 수 있다. 그에 비해 직접 예외를 처리하는 방법은 실제 에러 상황을 처리하여 예외 객체를 파괴한 후, 어플리케이션이 계속 실행되도록 하는 것이다.

실제 코드를 보면서 보호 블록을 익히도록 하자. 다음의 코드를 살펴보자.

```
try                                {보호 블록의 시작}
  Font.Name := '굴림';
  Font.Size := 10;
  Color := clBlue;                {보호 블록의 끝}
except                             {예외가 발생하면 다음의 블록을 실행한다.}
  on Exception do MessageBeep(0);
end;
```

이 코드에서는 Font 객체의 이름과 크기를 설정할 때 예외가 발생하면 ‘뽵’ 소리를 내도록 처리하였다.

#### ● 리소스 할당의 보호 (Protecting resource allocations)

견고한 어플리케이션을 제작함에 있어서 리소스를 할당하고, 이를 해제하는 것은 매우 중요한 부분이다. 예를 들어, 어플리케이션에서 메모리를 할당했으면 이를 반드시 해제해야 하며, 파일을 열었으면 이것은 반드시 닫아주어야 한다.

그러므로, 메모리를 할당하는 등의 리소스를 할당하는 문장이 있을 때에는 이를 해제하는 루틴을 블록으로 처리해 주는 것이 바람직하다.

보호해야 할 리소스로는 파일, 메모리, 윈도우 리소스, 텔파이 객체 등이 있으며 이들을 보호하기 위해서는 try ... finally 구문을 사용한다.

그러면, 실제 예를 살펴 보자. 다음의 코드는 메모리를 할당하고 해제하는 루틴이 포함되어 있지만 예외가 발생할 경우 메모리의 해제가 되지 않는다.

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);          {1KB의 메모리 할당}
  AnInteger := 10 div ADividend;   {에러 발생 !}
  FreeMem(APointer, 1024);         {이 문장은 실행되지 않는다 }
end;
```

이 코드에서는 0 으로 나누었으므로 division-by-zero 에러가 발생한다. 그러므로, 에러가 발생하면서 블록의 바깥으로 나가게 되므로 FreeMem 문장이 실행되지 않는 것이다. FreeMem 이 항상 실행되도록 하기 위해서는 리소스-보호 블록을 설정할 필요가 있다. 리소스를 보호하는 블록은 다음과 같은 try...finally 키워드에 의해 둘러싸이게 된다.

```
try
    {리소스를 사용하는 블록}
finally
    {리소스를 해제}
end;
```

이 구문에서 중요한 것은 예외가 발생하는 등의 어떤 상황에서도 finally 이후의 문장들이 실행된다는 것이다. 즉, try 이후의 구문들 중에서 예외가 발생하면 바로 finally 파트로 실행부가 옮겨 진다. 이때 finally 파트의 구문들을 클린업 코드라고 부른다. 만약, 예외가 발생하지 않은 경우에는 정상적인 순서에 의해 클린업 코드가 실행된다. 앞에서의 메모리 할당 예제를 try...finally 블록으로 다시 작성한 코드는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TComponent);
var
    APointer: Pointer;
    AnInteger, ADividend: Integer;
begin
    ADividend := 0;
    GetMem(APointer, 1024);           {메모리 할당 부분}
    try
        AnInteger := 10 div ADividend; {에러 발생}
    finally
        FreeMem(APointer, 1024);       {이 부분은 언제나 실행된다.}
    end;
end;
```

리소스-보호 블록은 예외를 처리하지 않는다는 점에 주의하여야 한다. 사실상 이 코드는 예외가 일어났는지조차도 상관하지 않는다.

- RTL 예외의 처리

수학 함수나 파일 처리 프로시저 등의 RTL(run-time library)의 루틴을 호출하는 코드를 사용할 때 RTL 은 에러가 발생할 때 어플리케이션에 예외를 넘겨준다. 디폴트로 RTL 예외는 어플리케이션에서 사용자에게 메시지를 보여주는데, 이때 RTL 예외를 처리하기 위해 자신의 예외 처리 루틴을 정의해서 쓸 수 있다.

RTL 예외는 SysUtils.pas 유닛에 정의되어 있다. 이들은 Exception 객체에서 상속된 것으로 이 객체는 디스플레이할 메시지를 문자열로 제공한다. RTL 에 의해 발생하는 예외에는 다음의 7 가지가 있다.

에러 종류	원 인	의 미
Input/output	파일이나 I/O 장치에 접근할 때 발생하는 에러	대부분의 I/O 예외는 윈도우가 파일에 접근할 반환하는 에러 코드와 관련이 있다.
Heap	동적 메모리 사용 에러	힙 에러는 할당할 메모리가 부족하거나, 어플리케이션이 힙 바깥의 메모리를 가리키는 포인터를 처리할 때 발생한다.
Integer math	잘못된 정수 연산	division by zero, 범위를 넘는 경우 등
Floating-point math	잘못된 실수 연산	하드웨어 연산프로세서나 소프트웨어 에뮬레이터의 잘못된 instruction, division by zero, 오버플로우, 언더플로우 등에 의해 발생한다.
Typecast	as 연산자로 형변환을 잘못된 경우	객체는 호환가능한 데이터 형으로만 형변환이 가능하다.
Conversion	형변환 함수의 에러	IntToStr, StrToInt, StrToFloat 등의 형변환 함수의 동작에서 에러가 발생한 경우
Hardware	시스템 조건	프로세서나 사용자가 발생시킨 에러 조건이나 인터럽션에 의한 에러로 예를 들어 접근 위반(access violation), 스택 오버플로우, 키보드 인터럽트 등이 있다.
Variant	가변형을 쓸 수 없을 때	가변형 변수를 호환 가능한 데이터 형으로 사용할 수 없을 때 발생한다.

## ● 예외처리 구문의 작성

예외처리 구문은 특정한 예외를 처리하거나, 보호 블록의 내부에서 발생한 코드의 예외를 처리하게 된다. 예외처리 구문을 정의하기 위해서는 예외처리를 하고자 하는 코드 블록을 정하고, except 파트에서 이를 처리하게 된다. 전형적인 코드를 소개하면 다음과 같다.

try

```
{보호하고자 하는 구문}
except
    {예외처리 구문}
end;
```

어플리케이션은 try 파트의 구문을 실행하다가 예외가 발생하면 except 파트의 구문을 실행하게 된다. 이때 try 파트에서 다른 루틴을 호출할 경우 호출된 루틴에서 적절한 예외 처리가 되지 않으면 역시 except 파트의 구문이 실행된다.

이때 예외가 발생하면 바로 except 파트로 넘어가므로, 그 뒤의 구문들은 실행되지 않는다. 일단 어플리케이션이 예외처리 구문으로 넘어가면 예외 객체는 자동으로 파괴된다.

예외 처리 구문의 기본적인 문법은 다음과 같다.

on <예외의 종류> do <처리 구문>;

그러면, 실제 예외처리 핸들러를 작성해 보자. 다음의 코드는 0 으로 나누는 작업에 의해 예외가 발생하며, 이를 처리한다.

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
    try
        Result := Sum div NumberOfItems;           {보통은 잘 넘어간다.}
    except
        on EDivByZero do Result := 0;              {0 으로 나눈 경우 0 을 결과로 반환}
    end;
end;
```

이와 같은 역할을 하는 함수를 if 문을 이용해서 작성하면 다음과 같다.

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
    if NumberOfItems <> 0 then
        Result := Sum div NumberOfItems
    else Result := 0;                                {예외 처리에 해당}
end;
```

예외처리를 사용하면 정상적인 알고리즘을 고안한 후, 이를 비켜가는 예외에 해당되는 부분의 처리 방법만 지정하면 된다. 그에 비해 예외처리를 하지 않고 이를 처리하려면 해당되는 각각의 경우에 맞도록 루틴을 정의해 주어야 한다.

- 예외 인스턴스의 사용 (Using the exception instance)

대부분의 경우 예외처리 구문은 예외의 형을 제외한 다른 정보는 필요로 하지 않는다. 그런데, 가끔 예외 인스턴스에 담긴 정보를 필요로 하는 경우가 있다. 대부분은 예외 인스턴스의 특정 정보를 읽어서 이를 예외처리에 활용하는 경우이다.

예를 들어, 하나의 폼에 스크롤 바와 버튼을 하나씩 추가한 후 버튼의 이벤트 핸들러에 다음과 같이 작성했다고 하자.

```
procedure TForm1.Button1Click(Sender: TComponent);
begin
    ScrollBar1.Max := ScrollBar1.Min - 1;
end;
```

이 문장은 최대값이 최소값보다 작을 수 없으므로 예외가 발생한다. 이때 디폴트 예외처리에서는 예외 객체의 메시지를 보여주게 되는데, 개발자가 자신의 메시지 문자열을 보여주고 싶을 때 다음과 같이 할 수 있다.

```
try
    ScrollBar1.Max := ScrollBar1.Min - 1;
except
    on E: EInvalidOperation do
        MessageDlg(E.Message + ‘: 이를 무시합니다 !’, mtInformation, [mbOK], 0);
end;
```

여기에서 E 는 EInvalidOperation 형의 임시 변수이고, 이 예외 인스턴스의 Message 정보를 직접 활용하여 메시지를 작성하였다.

- 예외처리 구문의 범위와 디폴트 예외처리 구문

개발자는 모든 블록에 대한 모든 종류의 예외를 처리할 필요는 없다. 원하는 블록의 예외만 처리해주면 된다.

블록이 특정 예외를 처리하지 않는다면, 예외는 블록을 벗어나게 되며 블록을 호출한 코드나 블록이 포함된 바깥 블록으로 제어가 넘어간다. 여기서도 예외 처리가 없으면 예외 객체는 계속 없어지지 않고, 예외 처리가 있을 때까지 계속 바깥 블록으로 이동한다.

예외 처리에 있어서 해당되는 예외가 없을 때에는 디폴트 예외처리 구문을 작성하여 사용하는 경우가 있다. 이럴 때에는 다음과 같이 예외처리 블록의 `except` 파트에 `else` 파트를 추가하여 작성하면 된다.

```
try
    {실행할 문장}
except
    on ESomething do      {특정 예러가 발생했으면, 이를 처리한다};
    else                  {디폴트 예외처리 코드};
end;
```

디폴트 예외처리 구문의 의미는 어떤 방법으로든 블록 내에서 발생하는 예외를 처리한다는 점이다. 그러나, 이를 남발하면 곤란한 경우를 당할 수 있으므로 적당하게 사용하는 것이 현명하며 될 수 있는 한 `try ... finally` 구문을 이용한 클린업 코드로 대체하는 것이 좋다.

#### ● 예외의 `raise`

예외를 로컬에서 처리할 때, 예외 처리 블록이 끝나면 예외 객체가 해제된다는 것은 이미 언급한 바 있다. 그런데, 가끔은 예외 처리 블록이 끝난 이후에 이 예외를 다시 처리하고 싶을 때가 있다. 이럴 때에는 예외 객체가 해제되는 것을 막을 필요가 있는데, 이럴 때에 사용하는 키워드가 `raise` 이다.

예를 들어, 예외가 발생하면 메시지를 사용자에게 보여주고 그 이후에 표준적인 처리를 하고 싶다고 하자. 이를 위해서는 로컬 예외처리 구문을 선언하여 메시지를 보여주고, `raise` 를 호출하여 다음에 다시 이를 처리할 수 있도록 해야 한다. 다음은 이를 위한 `pseudo-code` 이다.

```
try
    {실행 구문}
try
    {문제가 일어날 수 있는 구문}
except
    on ESomething do
        begin
```



```

    {특별한 경우에만 처리하는 구문}

    raise:                {예외 객체가 해제되지 않도록 한다}

end;

end;

except

    on ESomething do ...:    {이곳에서 다시 예외 처리를 할 수 있다}

end;

```

여기서 바깥 블록의 try 구문에서 예러가 발생하면 바깥 블록의 except 구문의 블록에서만 예외가 처리된다. 그런데 안쪽 블록의 try 구문에서 예러가 발생하면 안쪽 블록의 except 블록에서 예외가 처리된다. 만약 여기서 raise 를 사용하지 않으면 예외 객체가 해제되므로 바깥 블록의 예외처리 구문은 실행되지 않는다. 안쪽 블록에서 예외처리가 되어도, 바깥 블록에서 다시 예외처리를 하게 하려면 raise 를 호출하여 예외 객체의 해제를 막아야 한다. raise 명령을 이용한 예외처리는 이미 존재하는 예외처리 구문을 계속 사용하면서, 특별한 경우의 예외처리를 추가하려고 할 때 유용하게 쓰인다.

## ● 사용자 정의 예외의 정의

텔파이에서는 자신이 예외 객체를 선언해 놓고, 이를 어기는 경우에 예외처리를 하는 식으로 프로그래밍할 수 있다.

예외는 객체이기 때문에, 새로운 종류의 예외는 새로운 객체 형을 선언하는 것과 마찬가지로 간단하다. 기본적으로 예외 객체의 조상은 Exception 이다. 그러므로, 다음과 같이 Exception 객체 또는 Exception 객체에서 상속 받은 객체를 상속해서 선언해야 한다.

```

type

    EMyException = class(Exception):

```

여기서 개발자가 EMyException 예외를 발생시키되, EMyException 에 대한 핸들러를 제공하지 않으면 Exception 객체에 대한 디폴트 예외처리 구문이 실행된다.

특정 예외 객체를 일으키기 위해서는 앞서서도 잠시 살펴본 바 있는 raise 키워드를 이용한다. 이때 raise 뒤에 일으키게될 예외 객체를 적어준다는 점이 다르다. 예외처리 구문이 예외를 실제로 잘 처리하게 되면, 예외 객체가 파괴된다.

실제로 사용하는 예를 살펴보자. 다음의 코드는 패스워드를 잘못 입력한 경우에 사용자가 정의한 EPasswordInvalid 예외를 발생시킨다.

```

type

```

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

```
if Password <> CorrectPassword then
```

```
    raise EPasswordInvalid.Create('패스워드가 잘못 되었습니다 !');
```

참고: at 키워드

at 키워드는 raise 문과 같이 사용되며, 예외를 유발할 때 어느 기계어 코드 위치가 지시되어야 하는지를 가리킨다. 코드의 문법은 다음과 같다.

```
raise object at location;
```

실제로 SysUtils.pas 소스 코드에는 다음과 같은 문장이 있다.

```
raise OutOfMemory at ReturnAddr;
```

이는 에러가 프로시저 자체에서 발생된 것이 아니라, 프로시저를 호출하고 있는 쪽의 코드에서 나타난 것으로 raise 한다. 즉, 시스템 함수인 ReturnAddr 에 의해 리턴된 위치에서 발생한 것으로 나타나는 것이다.

## 문자열 포맷하기 (Formatting Strings)

텔파이는 기본적인 텍스트로 된 문자열, 서식 문자, 각각의 서식 문자에 대한 배열을 그 파라미터로 이용하여 다양한 가공을 할 수 있는 Format 함수를 제공한다.

Format 함수의 선언부는 다음과 같다.

```
function Format(const Format: string; const Args: array of const): string;
```

파라미터로 4 장에서 설명했던 가변 데이터형 개방형(array of const) 배열 파라미터를 사용한다. 예를 들어, 두 개의 수를 문자열로 포맷하려면 다음과 같이 한다.

```
Format('첫번째 %d, 두번째 %d', [i, j]);
```

여기서 i, j 는 정수형 변수로 첫번째 서식 문자는 첫번째 값으로, 두번째 서식 문자는 두번째 값으로 각각 대체된다. 만약 서식 문자의 출력 형태(%뒤의 글자에 의해 지정됨)가 해당 파라미터의 데이터 형과 맞지 않을 경우에는 런타임 에러가 발생한다. 즉, 컴파일 단계에

서 검사가 이루어지는 것이 아니다.

%d 이외에도, 이 함수에서는 여러가지 다양한 종류의 서식 문자를 사용할 수 있다. 이러한 서식 문자들은 주어진 데이터 형에 대한 디폴트 출력 양식을 이용할 수 있으나, 포맷 지정자를 사용하여 디폴트 출력 양식도 바꿀 수 있다. 예를 들어, 너비 지정자는 출력되는 문자수를 지정할 수 있으며, 정밀도 지정자는 소수점 아랫자리의 자리수를 지정할 수 있다. 예를 들어 다음의 코드를 보자.

```
Format('%8d', [i]);
```

이와 같은 형식을 사용하면 정수 i 의 8 자리가 오른쪽 정렬 형태로 출력하게 되고, 왼쪽은 공백으로 채워진다.

포맷 지정자는 다음과 같은 형태를 가진다.

"%" [index ":"] ["-"] [width] ["." prec] type

이를 조금 자세하게 설명하면, 포맷 지정자는 % 문자로 시작하며 % 뒤에 다음과 같은 순서의 옵션 지정자가 순서대로 붙게 된다.

1. argument 인덱스 지정자: [index ":"]
2. 좌측 정렬 지정자: ["-"]
3. 너비 지정자: [width]
4. 정밀도 지정자: ["." prec]
5. 변환 문자: type

변환 문자로 쓰일 수 있는 것으로는 다음과 같은 것들이 있다.

변환 문자	설 명
d (decimal)	argument 는 반드시 정수값이어야 하며, 10 진수로 기록된다.
u (unsigned)	d 와 같지만, 부호가 붙지 않는다.
e (scientific)	argument 는 반드시 부동 소수점 값이어야 한다. 값이 '(-)d.ddd...E+ddd'의 지수 표기법 형태로 출력된다. 정밀도 지정자에 의해 지정된 자리수로 표현되며, 정밀도 지정자가 지정되지 않은 경우 디폴트로 15 자리로 표현된다.
f (fixed)	argument 는 반드시 부동 소수점 값이어야 한다. 값이 '(-)ddd.ddd...'의 부동 소수점 표기법 형태로 출력된다. 정밀도 지정자에 의해 지정된 자리수로 표현되며, 정밀도 지정자가 지정되지 않은 경우 디폴트로 2 자리로 표현된다.
g (general)	argument 는 반드시 부동 소수점 값이어야 한다. 해당 부동 소수점 값이 부동 소

	수점 또는 지수 표기법을 가지는 가장 짧은 문자열로 변환된다. 정밀도 지정자에 의해 자리수가 표현되나, 디폴트는 15 자리이다.
n (number)	argument 는 부동 소수점 값이어야 한다. 값은 ‘(-)d,ddd,ddd.ddd...’의 형태로 표현된다. 이 포맷은 기본적으로 1000 단위로 쉼표가 추가된다는 것을 제외하면 f 포맷과 동일하다.
m (money)	argument 는 부동 소수점 값이어야 한다. 값은 통화 단위를 나타내는 형태로 바뀌어 표현된다.
p (pointer)	argument 는 포인터 값이어야 한다. 해당 포인터 값이 16 진 자리수를 가지는 8 자의 문자로 표현된다.
s (string)	argument 가 반드시 문자, 문자열, PChar 값이어야 한다. 문자열이 포맷 지정자의 자리에 삽입된다. 정밀도 지정자는 결과 문자열의 최대 길이를 지정하게 되며, 값이 지정한 최대 값을 넘으면 문자열이 잘리게 된다.
x (hexadecimal)	argument 는 반드시 정수값이어야 한다. 값이 16 진수를 나타내는 문자열로 변환되며, 포맷 문자열에 정밀도 지정자가 있다면 여기에 맞추어야 한다.

부동 소수점 값의 경우 1000 단위를 나타내는 문자열이나 10 진수를 나타내는 문자열은 각각 ThousandSeparator, DecimalSeparator 전역 변수에 의해 결정된다.

인덱스, 너비, 정밀도 지정자(Index, width, and precision specifiers)는 ‘%10d’와 같이 직접적으로 지정할 수도 있고, ‘\*’ 문자를 이용하여 ‘%\*.f’와 같이 간접적으로 지정할 수도 있다. 예를 들어, 다음의 문장을 살펴 보자.

```
Format('%*.f', [8, 2, 123.456]);
```

이것은 결국 다음 문장과 같은 역할을 하게 된다.

```
Format('%8.2f', [123.456]);
```

너비 지정자는 변환에 필요한 가장 작은 필드의 너비를 결정한다. 만약 결과 문자열이 이보다 작으면, 공백이 채워지게 된다. 디폴트로 는 우측 정렬에 맞도록 좌측에 공백이 채워지지만, ‘-’ 문자열로 좌측 정렬을 지정하면 우측에 공백이 채워진다.

인덱스 지정자는 현재의 argument 리스트 인덱스를 지정된 값으로 설정한다. 첫번째 argument 의 인덱스 값이 0 이며, 이를 이용하여 같은 argument 를 여러 차례 사용하는 것이 가능하다. 예를 들어, 다음의 코드는 ‘10 20 10 20’이라는 문자열을 만들어 낸다.

```
Format('%d %d %0:d %1:d', [10, 20]);
```

## 인터페이스의 활용 (Using interfaces)

텔파이 3 부터 추가된 인터페이스 개념은 오브젝트 인터페이스에 의해 구현되는데, 이는 클라이언트 코드가 특정 객체의 구현된 내용에 대해 알 필요가 없이 객체에 대해 접근할 수 있는 방법을 제공한다. 인터페이스의 선언은 실제적인 인터페이스의 구현을 포함하지는 않는다. 인터페이스는 다만 선언일 뿐이며, 나중에 클래스의 한 부분으로 구현되게 된다.

텔파이의 인터페이스는 DCOM 뿐만 아니라 텔파이 4 에서부터는 CORBA 기술의 스펙과 완벽하게 호환되므로, 나중에 쉽게 COM 이나 CORBA 객체를 생성하고, 사용하게 되는 근간이 된다.

### ● 기본 문법

다음의 코드는 인터페이스 선언의 예이다.

type

```
IMalloc = interface(IUnknown)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
end;
```

앞의 코드는 IUnknown 이라는 조상 인터페이스에서 상속받은 IMalloc 이라는 인터페이스의 선언이다. 언뜻 보면 클래스의 상속과 비슷한 형식으로 인터페이스도 상속됨을 알 수 있다. 인터페이스의 이름에는 대문자 I 가 접두어로 사용된다. 대괄호와 중괄호 기호 ([ '{~~~}' ]) 사이에는 인터페이스를 구별해주는 ID 가 들어간다. 여기에서 사용되는 ID 는 globally unique identifier (GUID)의 형태를 가진다. 텔파이에서는 이를 TGUID 형으로 정의해놓고 있다. 사실상 이 ID 는 품을 지칭하는 ID 이며, 이것이 나중에 운영체제에 등록되는 COM 객체로 사용될 때의 CLSID(ClassID)로 사용된다. 이 ID 는 텔파이에서 COM 객체를 만들 때 기존에 존재하는 ID 와 중복되지 않도록 자동으로 만들어준다. 참고로 윈도우 95 에 등록된 COM 객체의 CLSID 를 보고 싶으면 윈도우 95 의 레지스트리 에디터를 열어서 CLSID 라는 폴더를 찾아 확인해보기 바란다. 이 ID 의 선언부분 뒤에 인터페이스의

메소드, 프로퍼티 등의 멤버들을 선언하게 된다.

#### 참고: 인터페이스의 일반적인 규칙

인터페이스에 대해서 공부할 때 반드시 알아야 할 몇 가지 규칙이 있는데, 이것을 이해하는 것이 전체적인 이해에 도움이 될 것이다.

##### 1. 인터페이스 메소드는 추상적이다.

인터페이스 메소드 정의에는 인자들의 수와 type, 리턴 값의 type, 함수의 기능 등이 포함된다. 이런 정의가 어떻게 구현되는지는 알 필요가 없다. 즉, 인터페이스에서는 실제적인 메소드의 구현은 철저하게 숨겨져 있다. 그러므로, 클래스의 구현이 인터페이스의 정의를 따르게 되면, 인터페이스는 완벽한 다형성을 지원할 수 있다. 즉, 같은 인터페이스이되 구현은 다르게 할 수 있다. 이런 측면에서 인터페이스는 abstract type의 메소드만을 가진 클래스와 비슷하다. 추상 클래스와 마찬가지로 인터페이스는 자기 자신이 인스턴스화 되지 못다. 그러므로, 인터페이스를 구현한 클래스들이 사용되려면 인스턴스화되어야 한다.

##### 2. 인터페이스는 포인터로 접근한다.

모든 인터페이스 메소드는 추상적이기 때문에, 인터페이스는 가상함수 테이블의 포인터로 정의될 수 있다. 각각의 가상함수 테이블의 엔트리는 인터페이스를 구현한 클래스 내에서, 해당되는 인터페이스 메소드의 구현 부분을 참조한다. 결과적으로 객체의 인터페이스 구현 부분에 접근하는 것은 인터페이스 가상함수 테이블에 대한 포인터를 제공하는 것으로 해결할 수 있다.

##### 3. 인터페이스는 특정한 기능을 캡슐화한다.

인터페이스는 일반적으로 그 기능에 따라 접두어로 I를 포함해서 명명된다. 예를 들어, IMalloc 인터페이스는 메모리를 할당하고, 해제하고, 관리하는 역할을 한다. 또한, IPersist 인터페이스는 파일이나, 스트림 등에다 객체의 상태를 저장하거나 불러올 수 있는 다른 3개의 표준 persistence-related 인터페이스의 기초 인터페이스가 된다.

##### 4. 인터페이스는 유일한 ID를 가진다.

인터페이스들은 globally unique identifier (GUID)를 써서 다른 것들과 혼동되지 않도록

한다. GUID 는 universally unique identifier (UUID)의 특정한 형태이다. 이것은 16-byte (128-bit)의 이진 값으로 유일한 값이다. GUID 중에 인터페이스를 가리키는 것을 interface identifiers (IIDs)라고 하며, 각각의 인터페이스는 반드시 그들의 IID 를 가져야 한다. 이렇게 인터페이스를 이름으로 확인하지 않고, 유일한 값으로 지정하기 때문에 기존에 존재하는 코드가 업데이트 될 때 생기는 버전 문제를 해결할 수 있다.

#### 5. 인터페이스는 변하지 않는다.

인터페이스는 특정 기능을 제공하며 변하지 않는다. 인터페이스는 그들의 메소드, 기능, input, output 을 정의한다. 결과적으로 인터페이스 정의는 일단 publish 되면 변하지 않는다. 인터페이스의 메소드나 의미의 어떠한 변화도 새로운 인터페이스를 정의함으로써 이루어 진다.

#### 6. 인터페이스는 그들의 궁극적인 조상으로부터 기본적인 기능을 상속받는다.

모든 인터페이스는 IUnknown 인터페이스로부터 직간접으로 상속 받는다. 이 인터페이스는 인터페이스의 기본적인 기능을 정의한다. 그 내용은 인터페이스를 어떻게 부르며, 생성, 파괴할 것인지에 관한 것이다. 이를 정의한 메소드를 가상함수 테이블의 순서대로 나열하면 QueryInterface, AddRef, Release 이다. 인터페이스를 구현한 어떤 클래스도 반드시 위의 세가지 메소드와 조상 인터페이스의 메소드, 자신이 선언한 메소드는 반드시 구현해야 한다.

QueryInterface 는 클라이언트가 주어진 객체를 질의하고, 지원하는 인터페이스의 포인터에 접근하는 방법을 제공한다. AddRef 와 Release 는 간단한 참조계수(reference count)를 관리하는 메소드로, 객체가 적절할 때에 제거될 수 있도록 해준다. 참조계수가 0 이 아니면 객체는 메모리에 남아 있게 되며, 0 이 되면 인터페이스가 안전하게 객체들을 제거할 수 있다.

주의: 델파이는 IUnknown 의 참조계수와 인터페이스 질의에 대한 것을 다루는 객체 들을 제공하기 때문에 이 기능을 직접 구현할 필요가 없다.

### ● 인터페이스를 제작하고 접근하기

인터페이스는 기본적으로 COM, OLE, 서드파티 라이브러리에서 많이 제공하고 있다. 여기에 자신의 인터페이스를 추가할 수도 있는데, 이렇게 하려면 인터페이스 정의를 쓰고, 이 인터페이스를 구현하는 클래스를 제작하면 된다.

델파이에서 인터페이스를 제작하는 과정을 나열해 보면 다음과 같다.

- 인터페이스를 선언한다.
- 인터페이스를 지원하는 클래스 정의를 한다.
- 클래스에서 인터페이스 메소드 들을 구현한다.
- 인터페이스 reference 를 생성한다.
- 인터페이스 reference 를 통해 인터페이스 메소드를 유발한다.
- 객체에 대한 인터페이스 들을 질의한다.

주의: CoClass 는 인터페이스 구현 부분의 세트로만 이루어진 클래스이다. 그에 비해 델파이의 클래스는 인터페이스에서 선언된 메소드만 구현할 필요는 없다. 델파이의 클래스에는 그 밖에 다른 데이터나 메소드를 포함할 수 있는데, 이렇게 추가된 데이터나 메소드 들은 CoClass 로 인스턴스화된 객체에게 인터페이스 reference 를 통한 접근이 불가능하다.

## ● 인터페이스 메소드의 구현

클래스가 하나 이상의 인터페이스들을 구현하게 되면, 반드시 각각의 인터페이스에 대한 메소드들을 구현해야 한다. 인터페이스를 구현하려면 인터페이스 메소드는 클래스 내의 해당되는 메소드에 매핑(mapping)되어야 한다.

### 1. 인터페이스 메소드의 매핑

인터페이스의 메소드들을 클래스 내부의 메소드와 매핑을 할 때 다음의 규칙을 지켜야 한다.

- 메소드들은 반드시 같은 수의 파라미터를 가져야 한다.
- 해당되는 위치의 파라미터는 반드시 동일한 데이터 형이어야 한다.
- 결과값의 데이터 형이 동일해야 한다.
- 호출규칙(calling convention)이 같아야 한다.

디폴트로 각각의 인터페이스 메소드는 클래스내의 같은 이름의 메소드에 매핑된다.

### 2. 이름이 다른 메소드의 매핑

디폴트로 메소드 이름을 기초로 매핑을 하게 되지만, 클래스 선언시에 resolution 절을 쓰면 이름이 다른 메소드를 매핑할 수 있다.

type



```

TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ...
end;

```

이 선언문에서 IMalloc 인터페이스의 Alloc, Free method 를 TMemoryManager 의 Allocate, Deallocate method 로 매핑한다. 이런 방법은 2 개 이상의 인터페이스를 구현하는 클래스의 경우에 같은 이름의 메소드를 구현할 때 유용하다. 다음의 예를 보자.

type

```

IWindow = interface
    procedure Draw;
    ...
end;

```

```

ICanvas = interface
    procedure Draw;
    ...
end;

```

```

TWindow = class(TInterfacedObject, IWindow, ICanvas)
    procedure IWindow.Draw = Drawing;
    ...
    procedure Drawing;
    procedure Draw;
end;

```

여기에서 보면 IWindow 인터페이스의 Draw 메소드는 TWindow 의 Drawing 메소드로 매핑되며, ICanvas 인터페이스의 Draw method 는 TWindow 의 Draw 메소드로 매핑된다.

## ● 인터페이스 데이터 형 호환성과 변환

2 개의 인터페이스의 데이터 형이 동일하거나, 다른 쪽에서 상속되었을 때 이들은 호환된다. 또한, nil 값은 어느 인터페이스의 데이터 형과도 호환된다. 또한, 인터페이스를 구현하는 클

래스와의 호환성을 ‘대입호환(assignment compatible)’이라고 한다. 예를 들어 클래스 T가 인터페이스 I1, I2를 구현한다고 하면, T는 I1, I2와 대입호환된다.

또한, 어떤 인터페이스도 가변형(Variant type)과는 대입호환된다. 만약 인터페이스가 IDispatch type 이거나 그 자손일 경우, 결과의 가변형 값은 varDispatch 라는 type 코드를 가지게 된다. IDispatch type 이 아니면 varUnknown 이라는 코드를 가지게 된다.

인터페이스의 형변환(type casting)은 클래스에서와 비슷한 규칙을 따른다. 클래스는 IntfType(X)의 형태로 인터페이스로의 형변환이 가능하다. 이 때, X는 클래스 형이며, IntfType은 인터페이스 형이다.

인터페이스 형의 값은 Variant(X)의 형태로 가변형변수(Variant)로 형변환될 수 있는데, 이 때 X는 인터페이스 형이다. X가 IDispatch 이거나 그 자손일 경우 결과가 되는 가변형 코드는 IDispatch가 되며, 그렇지 않으면 IUnknown이 된다.

각 가변형 변수도 IUnknown이나 IDispatch type의 인터페이스로의 형변환이 가능한데, 이렇게 하려면 IUnknown(X), IDispatch(X)와 같이 쓰면 된다. 이 때 X는 가변형 변수이며, 이 때의 type 코드는 IUnknown으로 형변환될 경우에는 varEmpty, varUnknown, varDispatch 중의 하나이어야 하며, IDispatch로 형변환될 경우에는 varEmpty, varDispatch 중의 하나이어야 한다.

#### ● 인터페이스 참조하는 법

객체는 클래스에 의해 구현된 어떠한 인터페이스와도 대입호환된다. 이는 객체와 인터페이스의 참조가 간단한 대입문으로 해결될 수 있다는 것을 의미한다. 예를 들어,

type

```
IWindow = interface
```

```
...
```

```
end;
```

```
IDragDrop = interface
```

```
...
```

```
end;
```

```
TEditWindow = class(TObject, IWindow, IDragDrop)
```

```
...
```

```
end;
```

```
var
    EditWindow: TEditWindow;
    Window: IWindow;
    DragDrop: IDragDrop;
```

이와 같은 선언문이 있을 때, IWindow 와 IDragDrop 인터페이스에 대한 객체의 참조는 다음과 같은 간단한 대입문으로 해결된다.

```
Window := EditWindow;
DragDrop := EditWindow;
```

이러한 객체와 인터페이스의 변환은 객체의 선언된 type 에 기초한다. 예를 들어,

```
var
    Instance: TObject;
    Window: IWindow;
begin
    Instance := TEditWindow.Create(...);
    Window := Instance;
    ...
end;
```

이와 같은 대입문은 컴파일 에러가 발생하는데, 이는 Instance 의 선언된 type 은 TObject 이기 때문이다. 위에서는 실행코드 부분에서 instance 를 TEditWindow 로 다시 생성하기 때문에 문제가 발생한다. 이럴 때 쓰이는 연산자가 ‘as’이다.

## ● 인터페이스 질의

인터페이스 참조는 위에서 설명한 객체 참조(object reference)에도 as 연산자를 써서 참조할 수도 있는데 이를 인터페이스 질의(interface querying)이라고 한다. 인터페이스 질의 연산은 Reference as Interface 의 형태를 가진다. 여기서 Reference 는 IUnknown 인터페이스나 그 자손 인터페이스를 구현한 클래스이며, Interface 는 인터페이스의 type 결정자이다. 이 연산의 결과로 주어진 인터페이스 type 을 참조할 수 있게 된다. 이 때 질의된 객체나 인터페이스가 주어진 인터페이스를 구현하지 못하면 예외가 발생한다.

객체의 경우 as 연산자는 객체가 IUnknown 인터페이스를 구현하고 있을 경우에만 사용할 수 있다. 쓰는 형태는 Reference as Interface 로 동일한데, Reference 가 객체이면

IUnknown(Reference) as Interface 와 같은 형태로 쓰면 된다.

- 대리(delegation)를 통한 인터페이스의 구현

텔파이 4 에서는 implements 라는 새로운 지시어를 이용하여 인터페이스를 프로퍼티로 구현할 수 있게 되었다. 다음의 코드를 살펴 보자.

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

이 코드는 MyInterface 라는 프로퍼티를 선언하고, 이 프로퍼티는 IMyInterface 인터페이스를 구현한다는 의미이다. implements 지시어는 선언부의 마지막에 위치하여야 하며, 하나 이상의 인터페이스를 나열할 경우에는 콤마로 구별한다.

대리(delegate) 프로퍼티는 다음과 같은 조건을 충족시켜야 한다.

1. 반드시 클래스 혹은 인터페이스 type 이어야 한다.
2. 배열 프로퍼티이거나 인덱스를 가져서는 안된다.
3. 반드시 read specifier 를 가져야 한다.

프로퍼티가 read 메소드를 사용한다면 그 메소드는 디폴트 register 호출 규칙을 사용해야 하며, dynamic 메소드이면 안된다. 실제로 사용할 때에는 다음과 같이 하면 된다.

type

```
IMyInterface = interface
    procedure P1;
    procedure P2;
end;
```

```
TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
end;
```

var

```
MyClass: TMyClass;
MyInterface: IMyInterface;
```

```

begin
    MyClass := TMyClass.Create;
    MyClass.FMyInterface := ...           {구현 부분을 지정}
    MyInterface := MyClass;
    MyInterface.P1;
end;

```

대리 프로퍼티가 클래스 형일 경우의 사용 예는 다음과 같다.

```

type

    IMyInterface = interface
        procedure P1;
        procedure P2;
    end;

    TMyImplClass = class
        procedure P1;
        procedure P2;
    end;

    TMyClass = class(TInterfacedObject, IMyInterface)
        FMyImplClass: TMyImplClass;
        property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
        procedure IMyInterface.P1 = MyP1;
        procedure MyP1;
    end;

procedure TMyImplClass.P1;
...

procedure TMyImplClass.P2;
...

procedure TMyClass.MyP1;
...

```

```

var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;           //TMyClass.MyP1 을 호출하게 된다.
  MyInterface.P2;           //TImplClass.P2 를 호출하게 된다.
end;

```

## 메소드 포인터

메소드 포인터에 대해서는 4 장에서 프로시저 형과 함께 언급한 바 있지만, 델파이에서는 그 중요성이 크기 때문에 다시 한번 알아보고 넘어가도록 하자.

메소드 포인터란 메소드 코드의 주소와 객체 인스턴스의 주소를 모두 가지게 되는 포인터로, 객체 인스턴스의 주소는 메소드 내부에서 Self 로 표현한다. 메소드 포인터 형의 선언은 프로시저 형의 선언과 동일하지만 맨 마지막에 of object 라는 키워드가 추가된다는 점이 다르다.

그러면 메소드 포인터를 하나 선언해 보자.

```

type
  TSampleMethod = procedure(i: Integer) of object;

```

이렇게 메소드 포인터를 선언하고 나면, 객체 내에 필드를 메소드 포인터 형으로 지정할 수 있다.

```

type
  TSampleClass1 = class
    FNumber: Integer;
    SampleMethod: TSampleMethod;
  end;

```

나중에 이 필드에 파라미터의 수와 데이터 형이 같은 형 호환(type compatible) 메소드를 대입할 수 있다. 예를 들어, 다음과 같이 같은 정수형 파라미터를 가지는 메소드가 있는

다른 클래스가 있다고 하자.

type

```
TSampleClass2 = class
    FNumber: Integer;
    procedure Sum(j: Integer);
end;
```

이런 경우에 다음과 같이 이들 클래스가 객체로 선언되면 메소드의 대입이 가능하다.

var

```
SampleObject1: TSampleClass1;
SampleObject2: TSampleClass2;
begin
    SampleObject1.SampleMethod := SampleObject2.Sum;
end;
```

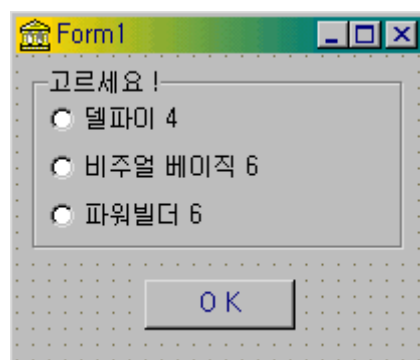
이것이 델파이 컴포넌트에서 가장 중요한 기술의 하나라고 할 수 있는 대리(delegation) 기법이다. 메소드 포인터가 있는 객체가 있으면, 새로운 메소드를 그 객체에 대입하면 자유롭게 객체의 동작을 바꿀 수가 있는 것이다.

다소 복잡하다고 생각될 수 있겠지만, 델파이 컴포넌트의 대부분의 이벤트 핸들러 등에서 이런 기법은 자주 사용된다. 예를 들어, 버튼에 대해 OnClick 이벤트 핸들러를 추가하면 델파이는 버튼의 OnClick 이라는 이름의 메소드 포인터를 사용자가 제작한 이벤트 핸들러를 가리키도록 하여, 버튼이 클릭될 때 이벤트 핸들러가 호출되는 것이다.

그러면, 메소드 포인터를 확실하게 이해할 수 있는 예제를 하나 만들어 보자.

이번에 만들 예제는 동적으로 이벤트 핸들러를 바꾸어 주는 방법을 소개하는 것으로, 이것이 가능한 이유가 바로 메소드 포인터를 활용할 수 있기 때문이다.

새로운 어플리케이션을 시작하고 폼에 TRadioGroup, TButton 컴포넌트를 하나씩 올려 놓고, 각각의 캡션을 '고르세요 !', 'OK'로 설정하고 RadioGroup1 의 Items 프로퍼티를 다음 그림과 같이 '델파이 4, 비주얼 베이직 6, 파워빌더 6'로 설정한다.



이제 코드 에디터에서 메소드 포인터를 선언하도록 하자. 지금 하려고 하는 작업은 Button1 의 OnClick 이벤트 핸들러를 전혀 작성하지 않고, 순전히 메소드 포인터의 대입을 통해서 이를 구현하려는 것으로 폼이 시작될 때 기본적인 이벤트 핸들러의 역할을 할 Start 라는 프로시저를 OnClick 메소드 포인터에 대입하고, 라디오 버튼을 클릭할 때마다 서로 다른 메소드 포인터를 대입하여 여러 프로시저를 실행하게 할 것이다.

메소드 포인터를 대입할 수 있으려면, Button1 의 OnClick 메소드 포인터와 파라미터의 수와 데이터 형이 일치하여야 한다. OnClick 메소드 포인터는 TNotifyEvent 라는 메소드 포인터 형으로 선언되어 있는데, 델파이 컴포넌트의 많은 이벤트 들이 이 메소드 포인터 형이다. TNotifyEvent 메소드 포인터는 Classes.pas 유닛에 다음과 같이 선언되어 있다.

type

TNotifyEvent = procedure (Sender: TObject) of object;

그러므로, 우리가 작성할 이벤트 핸들러 들은 파라미터의 수와 데이터 형을 여기에 맞게 작성하면 된다. 그러면 폼이 처음 시작할 때의 이벤트 핸들러를 Start 라고 하고 델파이 4, 비주얼 베이직 6, 파워빌더 6 라디오 버튼이 선택된 경우의 이벤트 핸들러를 각각 Delphi4, VisualBasic6, PowerBuilder6 라고 하고 이들을 TForm1 의 선언부에 다음과 같이 추가한다.

TForm1 = class(TForm)

RadioGroup1: TRadioGroup;

Button1: TButton;

procedure Start(Sender: TObject);

procedure Delphi4(Sender: TObject);

procedure VisualBasic6(Sender: TObject);

procedure PowerBuilder6(Sender: TObject);

... (후략)

그리고, 메소드 포인터를 대입하는 코드를 작성하기 전에 이들 각각을 다음과 같이 구현하도록 하자.

procedure TForm1.Start(Sender: TObject);

begin

ShowMessage('선택하신 것이 없군요 !');

end;



```
procedure TForm1.Delphi4(Sender: TObject);
begin
    ShowMessage('당신의 선택은 100 점 !');
end;
```

```
procedure TForm1.VisualBasic6(Sender: TObject);
begin
    ShowMessage('조금만 더 생각해 보세요 !');
end;
```

```
procedure TForm1.PowerBuilder6(Sender: TObject);
begin
    ShowMessage('옳은 선택일까요 ?');
end;
```

무슨 내용일까 ? 판단은 자유에 맡기기로 한다. 이 글을 읽고 있는 사람들은 델파이를 사랑하는 사람 들일 것이므로 다소 장난기가 있는 코드이지만 모두 이해할 것으로 믿는다. 그러면, 이제 실제로 메소드 포인터를 대입하도록 하자. 먼저 폼이 생성될 때에는 Start 를 대입해야 하므로 Form1 의 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

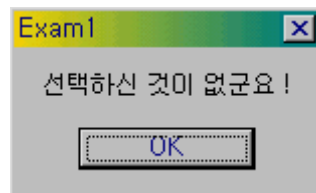
```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Button1.OnClick := Start;
end;
```

그리고, 각각의 라디오 버튼을 선택했을 때의 변화를 위해 RadioGroup1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

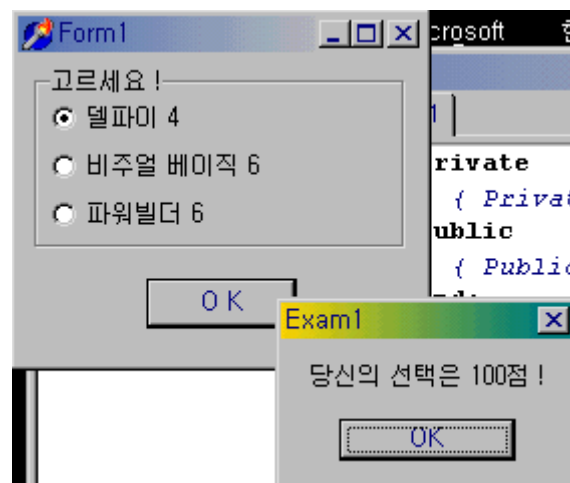
```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
    case RadioGroup1.ItemIndex of
        0: Button1.OnClick := Delphi4;
        1: Button1.OnClick := VisualBasic6;
        2: Button1.OnClick := PowerBuilder6;
    end;
```

end;

대입 호환이 보장되므로, 이렇게 프로시저의 이름을 대입하는 것으로 모든 작업은 간단히 끝난다. 그러면 이를 실행해 보자. 실행된 직후에 버튼을 클릭하면 다음과 같은 메시지를 볼 수 있을 것이다.



그러면, 델파이 4 을 선택하고 버튼을 클릭해보자. 다음과 같은 메시지를 볼 수 있다면 제대로 실행된 것이다.



## 클래스 참조 (Class Reference)

클래스 참조란 클래스 형에 대한 참조(reference)를 일컫는 말이다.

클래스 참조는 클래스의 인스턴스에 대한 조작을 하기 보다는, 클래스 자체에 대한 작업을 하려고 할 때 사용된다. 이럴 때에는 변수나 파라미터를 클래스 자체를 값으로 가지게 할 필요가 있는데, 클래스 참조형(class reference type)을 사용하면 된다.

클래스 참조형은 다른 말로 메타 클래스라고도 하며, 다음과 같이 클래스에 대한 클래스를 선언하면 된다.

```
class of class1;
```

참고로 델파이의 TClass 형은 다음과 같이 선언된 클래스 참조형이다.

type

```
TClass = class of TObject;
```

var

```
AnyObj: TClass;
```

AnyObj 변수는 어떤 클래스도 참조할 수 있다. 클래스 참조형에 대해 가장 전형적인 사용 예를 든다면 TCollection 클래스의 constructor 에 대한 선언부를 들 수 있다. 다음의 코드를 살펴 보자.

type

```
TCollectionItemClass = class of TCollectionItem;
```

```
... (중략)
```

```
constructor Create(ItemClass: TCollectionItemClass);
```

이 선언부의 의미는 TCollection 인스턴스 객체를 생성하려면, 반드시 constructor 에 TCollectionItem 에서 상속받은 클래스의 이름을 파라미터로 넘겨주어야 한다는 것이다.

클래스 참조는 다른 OOP 언어에서 사용하는 메타 클래스 개념과 비슷하지만, 오브젝트 파스칼에서 클래스 참조는 클래스가 아니라 일종의 데이터 형에 대한 포인터일 뿐이다.

클래스 참조의 가장 큰 장점은 클래스 데이터 형을 실행 도중에 조작할 수 있다는 점과 선언된 클래스 이외에 어떤 서브 클래스이든 대입할 수 있다는 것이다. 이런 측면에서 보면 모든 클래스는 TObject 에서 파생되므로, 앞에서 예를 든 TClass 클래스 참조형은 델파이로 작성하는 어느 클래스의 참조를 저장할 때에도 사용할 수 있다. 예를 들어, Application 객체의 CreateForm 메소드는 만들어 낼 폼의 클래스를 파라미터로 요구한다.

```
Application.CreateForm(TForm1, Form1);
```

이 코드에서 첫번째 파라미터는 클래스 참조이고, 두번째는 실제 객체이다.

그렇다면 델파이에서의 클래스 참조의 활용 방법에는 어떤 것이 있을까? 델파이에서 새로운 컴포넌트를 폼에 추가하면, 데이터 형을 선택하고 그 데이터 형에 대한 객체를 만들어 낸다. 이 작업은 클래스 참조를 활용하여 델파이가 해주는 작업이다.

## 정 리 (Summary)

이번 장에서는 예외 처리, 인터페이스, 메소드 포인터와 같은 비교적 고급스러운 오브젝트 파스칼의 면면을 살펴 보았다. 이런 사항 들은 몰라도 대부분의 어플리케이션을 작성할 수 있지만, 보다 효율적이고 잘된 프로그램을 개발하려면 알아두면 많은 도움이 될 것이다.

다음 장에서는 VCL 계층 구조와 의미, 일부 컴포넌트의 사용 방법 등에 대해서 알아볼 것이다.

## 비주얼 컴포넌트 라이브러리의 이해

### (Understandings of Visual Component Library)

델파이의 컴포넌트는 OOP의 관점에서 보면 진정한 객체라고 말할 수 있다. 델파이의 컴포넌트는 데이터의 세트와 데이터 접근 함수를 캡슐화 하며, 조상 컴포넌트의 기능과 데이터를 상속하고, 동일한 조상에서 상속받은 서로 다른 객체들이 다양하게 동작하는 다형성을 완벽하게 지원한다.

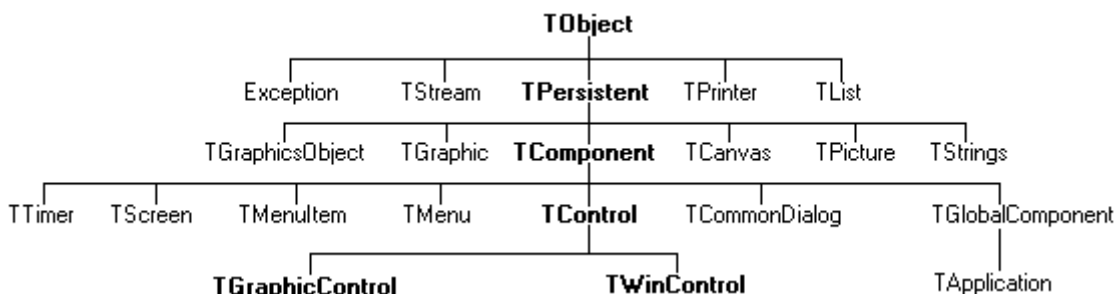
이번 장에서 델파이의 VCL 라이브러리의 구조에 대해서 알아 보고, 이를 활용하여 어플리케이션을 제작하는 방법에 대해서 알아본다. 기본적인 어플리케이션의 제작 방법은 이미 간단히 알아본 바 있으므로 구체적인 컴포넌트의 사용 방법은 독자들이 도움말과 데모 어플리케이션을 이용해서 익히기 바란다.

그리고, 이 장의 후반부에는 비교적 고급스러운 주제인 드래그-드롭(drag-and-drop)을 구현하는 방법과 델파이 4에서 새롭게 지원되는 드래그-도크(drag-and-dock)의 구현, 액션 리스트를 이용하는 방법에 대해서 알아본다.

#### VCL 객체 계층 구조

VCL은 Visual Component Library의 약자로서, 컴포넌트라고 부르는 객체들로 구성된 라이브러리를 가리킨다. VCL은 델파이의 오브젝트 파스칼로 작성되었으며, 개별적인 파일이 아니라 하나의 라이브러리에 저장되어 있다. 이들은 나중에 어플리케이션 실행 파일의 일부분으로 사용된다.

VCL에 대해 잘 이해하려면 먼저 VCL의 전체적인 계층구조에 대해서 알아야 한다. VCL의 계층 구조의 핵심 부분은 다음과 같다.



여기서 주의해서 볼 부분은 가운데의 축을 이루는 **TObject**, **TPersistent**, **TComponent**, **TControl** 클래스이다. 이들은 델파이 VCL 계층 구조에서 가장 핵심을 이루는 클래스들이다. 이들의 특징에 대해서 이번 장에서 알아볼 것이다.

간략하게 설명하면 TObject 클래스는 델파이의 모든 객체의 기초가 되는 클래스이다. 즉, 모든 델파이 객체는 TObject 클래스를 상속한다. TPersistent 클래스는 TObject 클래스에 추가적인 데이터를 저장할 때 사용자 정의 메소드를 사용하는 것을 허용한다. 다시 말해 값을 저장하고, 불러오는 등의 지속성(persistency)을 지원한다.

TComponent 클래스는 메소드와 프로퍼티를 지원하며 컴포넌트 팔레트에 나타나고, 다른 컴포넌트를 소유할 수 있는 등의 델파이에서 사용하는 각종 컴포넌트의 기본적인 행동을 정의하는 클래스이다.

델파이의 컴포넌트는 시각적인 컴포넌트와 비시각적인 컴포넌트로 나눌 수 있는데, 여기에서 시각적인 컴포넌트를 컨트롤이라고 부른다. 이런 컨트롤의 기본적인 특징을 정의하는 클래스가 TControl 이다. 그래서, TControl 이 아닌 TComponent 를 계승한 컴포넌트를 특히 비시각적 컴포넌트라고 한다.

컨트롤은 폼 위에 존재하면서 사용자에게 정보를 보여 주거나 또는 폼과 상호작용하는 컴포넌트이다. 특별한 컨트롤로는 윈도우 컨트롤이 있다. 윈도우 컨트롤에는 윈도우 핸들이 있는데, 윈도우는 핸들을 이용하여 윈도우 컨트롤에 메시지를 전달한다. 만약 윈도우 핸들을 가지고 있지 않은 컨트롤을 그래픽 컨트롤이라고 한다. 여기서 TWindowControl 로부터 상속받은 컨트롤을 윈도우 컨트롤이라고 하고, TGraphicControl 로부터 상속받은 컨트롤을 그래픽 컨트롤이라고 한다.

## ● TObject 클래스

델파이의 모든 클래스는 TObject 클래스의 서브 클래스이다. 즉, 모든 객체가 하나의 조상을 가지는 것이다. 그렇기 때문에, 개발자는 시스템의 어떤 클래스의 데이터 형이든 TObject 데이터 형으로 대체하여 사용할 수 있다. 가장 흔히 TObject 를 구경할 수 있는 것은 이벤트 핸들러의 파라미터로 사용하는 ‘Sender’ 이다. 이 파라미터는 보통 TObject 형으로 선언되어 있는데, 이는 어떤 클래스의 객체인 Sender 객체로 사용할 수 있다는 것을 의미한다.

이렇게 TObject 를 사용할 경우 객체에 대한 작업을 할 때 그 데이터 형을 알아내야 하는 경우가 생긴다. 즉, TObject 형의 변수가 있으면 이 변수는 TObject 에 의해 정의된 메소드와 프로퍼티만 변경할 수 있다. 예를 들어, 이 변수가 TComponent 와 같이 TObject 의 자식 클래스 객체를 참조하게 되면 TObject 클래스에 없는 프로퍼티나 메소드에는 직접 접근할 수가 없다.

이를 해결하기 위해서는 RTTI 연산자인 as 와 is 를 사용하여 데이터 형을 검사하고 데이터 변환을 해야 한다. 예를 들어 TObject 형의 Sender 파라미터가 에디트 박스를 가리킨다면 다음과 같이 쓸 수 있다.

```
(Sender as TEdit).Text := ‘재밌다 !’;
```

이렇게 형변환을 해서 사용하는 방법은 형변환이 실패하면 델파이가 예외를 발생시키므로, 적절한 예외 처리가 가능하다.

참고: 런타임 타입 정보(Run-Time Type Information, RTTI)에 대하여

델파이는 클래스의 가상 메소드 테이블(Virtual Method Table, VMT)내에 모든 클래스에 대한 몇 가지 데이터 형 정보를 저장한다. VMT 와 published 타입 정보는 실행시 이용 가능하다. 그래서 이를 런타임 타입 정보라고 한다.

각 클래스는 자신만의 고유한 VMT 를 가지고 있으며, 델파이는 클래스 VMT 포인터 들을 검사하여 클래스들을 구분한다. 한 클래스의 모든 인스턴스들은 해당 클래스의 VMT 를 공유하게 된다.

#### ● TPersistent 클래스

TPersistent 클래스는 다른 객체에 대입될 수 있는 모든 행동을 캡슐화한 클래스이다. 그리고 이런 속성을 스트림에 저장하고, 읽을 수 있어야 한다. 이를 위해 TPersistent 클래스는 다음과 같은 특징을 가지는 메소드 들을 지원하며, 이들을 적절히 오버라이드하여 사용해야 한다.

- publish 되지 않은 데이터를 스트림에 저장하고, 읽을 수 있는 프로시저를 정의한다.
- 값을 프로퍼티에 대입할 수 있는 방법을 제공한다.
- 하나의 객체의 내용을 다른 객체에 대입할 수 있는 방법을 제공한다.

TPersistent 클래스는 컴포넌트가 아닌 객체를 기초 클래스로 선언하는데, 값을 스트림에 저장하거나 대입 기능을 추가할 필요가 있을 때 가장 적합한 기초 클래스가 된다.

또한 TPersistent 클래스는 published 클래스를 생성시킬 때 많이 사용된다. 예를 들어, 가장 대표적으로 TComponent 는 TPersistent 를 상속한다. 따라서 모든 컴포넌트는 published 되며, 하나의 published 섹션을 가질 수 있다. 또한 모든 컨트롤과 폼은 TComponent 와 TPersistent 로부터 상속받기 때문에 published 섹션을 가질 수 있다.

델파이는 TPersistent 로부터 계승된 여러 개의 클래스를 가지고 있다. 따라서, 그 클래스 들은 프로퍼티 데이터 형으로 이용될 수 있다. 예를 들어 TStrings, TFont, TBrush 등이 여기에 해당된다. 반면에 TList 는 TObject 로부터 계승되며 published RTTI 를 가지지 않기 때문에 published 프로퍼티의 데이터 형으로 사용할 수 없다.

#### ● TComponent 클래스

TComponent 클래스는 델파이에서 사용하는 모든 컴포넌트의 기본적인 특징을 지원하는 기초 클래스이다. TComponent 클래스가 지원하는 기능은 다음과 같다.

- 컴포넌트 팔레트에 나타날 수 있으며, 폼 디자이너에서 조작할 수 있다.
- 다른 컴포넌트를 소유하고 다룰 수 있다.
- 향상된 스트림 지원과 파일 조작 기능
- 액티브 X 컨트롤 등의 인터페이스를 구현하는 객체에 대한 wrapper로서의 기능

비시각적 컴포넌트는 일반적으로 TComponent로부터 직접 상속받는다. 이러한 컴포넌트는 설계시에는 사용할 수 있도록 보이지만, 실행시에는 보이지 않으면서 백그라운드에서 유용한 서비스들을 수행한다. TTimer는 비시각적 컴포넌트이고, TField와 같은 데이터베이스 관련 컴포넌트들도 대부분 비시각적 컴포넌트이다.

## ● TControl 클래스

컨트롤이란 비주얼 컴포넌트를 의미하는 것으로 사용자가 런타임에서 이들을 직접 보고, 조작할 수 있다. 모든 컨트롤들은 공통적인 프로퍼티, 메소드, 이벤트를 가지고 있으며, 여기에는 컨트롤의 위치와 커서, 힌트 등에 대한 프로퍼티와 컨트롤을 움직이거나 칠하는 메소드와 마우스 행동에 반응하는 이벤트 등이 있다.

컨트롤에는 윈도우 컨트롤과 그래픽 컨트롤이 있는데 이들의 차이점은 다음과 같다.

### 1. TWinControl

TWinControl은 입력 포커스를 받아야 하거나, 키보드 입력이 있어야 할 경우에 사용하는 컨트롤이다. 대표적인 컴포넌트로는 TEdit 컨트롤을 들 수 있다. 또한 윈도우 컨트롤은 다른 윈도우나 컨트롤을 담을 수 있다.

### 2. TGraphicControl

TGraphicControl 컴포넌트는 객체는 포커스를 가지지 않으며, 키보드에 반응하지 않는다. 그렇기 때문에 윈도우에 대한 부담이 줄어들기 때문에 가볍다. 대표적인 그래픽 컨트롤로는 TImage 컴포넌트를 들 수 있다. 또한 TSpeedButton 컨트롤도 그래픽 컨트롤인데, 스피드 버튼은 마우스로만 접근하며 특별히 포커스를 가지지도 않기 때문에 어찌 보면 당연한 것이다.



그래픽 컨트롤과 윈도우 컨트롤 사이의 차이점은 컴포넌트에게 메시지를 전송하고자 할 때 매우 중요하다. 어떤 객체에게나 그 객체의 디스패치(dispatch) 메소드를 호출하여 메시지를 보낼 수 있다. 그러나 윈도우 컨트롤에게는 SendMessage, PostMessage 를 사용한다. 따라서 메시지를 받기 위해서는 윈도우 핸들이 필요하다.

## VC++의 공통적인 프로퍼티

모든 비주얼 컴포넌트에는 공통적으로 사용하는 많은 수의 프로퍼티들이 있다. 한번 비주얼 컴포넌트에 대해서 이해를 하면 많은 수의 프로퍼티가 비슷하다는 것을 쉽게 알 수 있다. 그러면, 이들 중에서 몇가지에 대해 알아보도록 하자.

- Name 프로퍼티

모든 델파이 컴포넌트들은 적절한 이름을 가지고 있다. 이름은 owner 컴포넌트 안에서 각각 유일한 것이어야 하는데, 여기서 owner 컴포넌트는 일반적으로는 컴포넌트를 올려놓는 폼을 말한다. 이것은 어플리케이션이 두 개의 서로 다른 폼을 가질 수 있으며, 그 각 폼들은 같은 이름을 가지는 컴포넌트를 가질 수 있다는 것을 의미한다.

컴포넌트의 Name 프로퍼티 값은 폼 클래스의 선언 안에서 객체의 이름을 정의하는데 사용된다. 이것은 객체를 가리키기 위해 일반적으로 코드 안에서 사용하게 되는 이름이다. 그러므로, 이 값은 파스칼의 변수 이름 규칙에 맞아야 한다.

- 위치와 크기 프로퍼티

폼 위에 있는 컨트롤의 크기와 위치를 정의하기 위해 사용되는 프로퍼티에는 다음의 4 가지가 있다.

1. 수직 크기를 나타내는 Height
2. 수평 크기를 나타내는 Width
3. 위쪽 끝에서의 위치를 나타내는 Top
4. 좌측 끝에서의 위치를 나타내는 Left

이들 프로퍼티는 비시각적 컴포넌트에서는 제공되지 않는다. 그리고, 대부분 폼 디자이너에서 마우스를 가지고 조절할 때 자동으로 설정되지만 경우에 따라서 런타임에서 직접 값을 대입하기도 한다.

- 활성화(Activation), 시각화(Visibility) 속성

사용자들로 하여금 컴포넌트를 활성화하거나 감추도록 하는데 사용할 수 있는 2 가지 기본 프로퍼티가 있다. Enabled 프로퍼티는 입력을 하지 못하게 한다고 생각하면 된다. 이 프로퍼티를 False 로 설정하면 디자인 시에는 그 변화가 눈에 보이지 않지만, 런타임에서는 흐리게 표시된다. 어떤 경우에는 아예 컨트롤을 눈에 보이지 않게 해버리고 싶을 때가 있는데, 이럴 때에는 Hide 메소드를 사용하거나 Visible 프로퍼티를 False 로 설정한다.

여기서 주의할 것은 Visible 프로퍼티를 가지고 그 컨트롤이 보이는지를 판단하면 안된다는 것이다. 만약 컨트롤의 컨테이너가 감추어지면, 그 컨트롤의 Visible 프로퍼티의 내용에 상관없이 그 내용을 볼 수 없다. 이럴 때에는 Showing 프로퍼티를 사용하여 현재 그 컨트롤이 보이는지 알 수 있다.

#### ● 디스플레이 프로퍼티

컨트롤의 전반적인 형태를 결정하는 프로퍼티에는 다음과 같은 것들이 있다.

1. 컨트롤의 경계부위의 형을 설정하는 BorderStyle
2. 컨트롤의 배경색을 결정하는 Color
3. 컨트롤이 3D 형태를 가질 것인지를 결정하는 Ctrl3D
4. 폰트의 경우 색상, 이름, 크기 등을 color, name, style 을 통해 모두 설정할 수 있다.

#### ● Parent, Owner 프로퍼티

어플리케이션의 컨트롤의 모양을 컨트롤을 담고 있는 컨테이너의 전반적인 색상, 폰트 등과 일치시키기 위해서는 ParentColor, ParentFont, ParentCtrl3D 등의 프로퍼티를 True 로 설정하면 된다. 보통 기본적으로 이들 프로퍼티는 True 로 설정되어 있는데 Font, Color, Ctrl3D 프로퍼티를 설정하게 되면 이런 Parent 프로퍼티 들의 설정은 False 로 바뀌게 된다.

그리고 프로퍼티 중에 컴포넌트의 Parent 와 Owner 를 결정하는 Parent, Owner 프로퍼티도 있다.

Parent 프로퍼티는 비시각적 컴포넌트 이외의 컨트롤 사이의 시각적 관계를 나타낸다. 델파이의 개발 환경은 Parent 컨트롤을 TPanel, TGroupBox 와 같은 특별히 Parent 컨트롤로 사용하려고 만든 컨트롤들로 제한하고 있다. TPanel 컨트롤 위에 컴포넌트를 놓으면 델파이는 자동으로 새 컴포넌트의 Parent 프로퍼티를 TPanel 로 설정한다. 그리고, Parent 컨트롤은 Child 컨트롤 들을 .DFM 파일에 기록한다.

Owner 프로퍼티는 일반적으로 메모리 관리에 사용된다. 시각적이든 비시각적이든 모든 컴포넌트는 Owner 프로퍼티를 가지고 있다. 컴포넌트가 자신의 Free 메소드 호출에 의해서

제거될 때, 자동으로 컴포넌트의 Owner 가 된다. 따라서 폼이 제거되면 폼 안의 모든 컴포넌트도 같이 제거된다. 보통 대부분의 컴포넌트의 Owner 는 폼이 된다.

- 드래그-드롭(Drag-and-drop) 프로퍼티

드래그-드롭을 지원하는 프로퍼티는 드래그가 어떻게 시작되는지 결정하는 DragMode, 드래그할 때 도킹을 지원할 것인지를 결정하는 DragKind, 드래그할 때 커서의 모양을 결정하는 DragCursor 등이 있다. 드래그-드롭에 대해서는 이 장의 후반부에서 자세히 다룰 것이다.

- 드래그-도크(Drag-and-dock) 프로퍼티

DockSite, DragKind, DragMode, FloatingDockSiteClass 등의 컴포넌트 프로퍼티는 드래그-도크를 지원하기 위해 사용되는 프로퍼티이다. 이들에 대해서는 이 장의 후반부에 자세히 다룰 것이다.

- Canvas 프로퍼티

Canvas 프로퍼티는 그래픽 컨트롤의 가장 중요한 프로퍼티로, 윈도우 디바이스 컨텍스트를 캡슐화한다. 이 프로퍼티를 이용하면 저수준의 그리기 함수를 이용할 수 있다. 여기에 대해서는 다른 장에서 더욱 자세하게 알아볼 것이다.

## 델파이 컴포넌트의 개괄 (Delphi Components Overview)

델파이 컴포넌트 팔레트는 어플리케이션을 작성할 때 사용할 수 있는 많은 컴포넌트로 이루어져 있다. 컴포넌트를 개발자 입맛대로 팔레트에 추가, 삭제, 재배열할 수 있으며 컴포넌트 템플릿을 이용하여 많은 수의 컴포넌트를 추가할 수도 있다.

컴포넌트는 비슷한 기능에 따라 그룹이 나누어진다. 다음에 컴포넌트 팔레트의 페이지 별로 어떤 종류의 컴포넌트를 가지고 있는지 나열해 보았다.

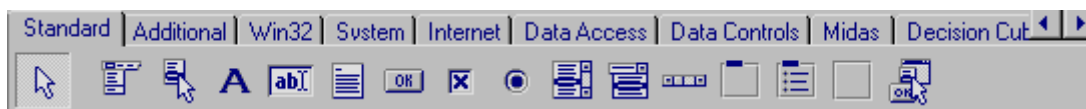
페이지 이름	내 용
Standard	메뉴 등의 표준 윈도우 컨트롤
Additional	비트맵 버튼 등의 추가적인 컨트롤
Win32	윈도우 95/NT 4.0 에서부터 지원하는 공통 컨트롤, 윈도우 98 용 컨트롤
System	DDE, 파일 시스템, 멀티 미디어, 타이머 등의 시스템 레벨의 컴포넌트
Internet	클라이언트/서버 어플리케이션을 관리하는데 도움을 주는 컴포넌트, 다양한 인터넷

	통신 프로토콜 컴포넌트, 저수준 TCP/IP 를 지원하는 소켓 관리 컴포넌트와 웹서버 어플리케이션을 위한 컴포넌트
Data Access	데이터베이스, 테이블, 쿼리, 리포트 등에 대한 비시각적 컴포넌트
Data Controls	시각적인 데이터 컨트롤
Decision Cube	데이터베이스의 정보를 요약하고 이를 다양한 시각으로 보여줄 수 있는 컨트롤
QReport	쉽게 리포트를 작성하는 것을 도와주는 Quick Report 컴포넌트
Dialogs	윈도우 공통 대화상자 컴포넌트
Win 3.1	델파이 1.0 프로젝트와의 호환성을 위한 컴포넌트
Samples	게이지, 컬러 그리드, 스피너 버튼 등의 샘플 사용자 정의 컴포넌트
ActiveX	샘플 액티브 X 컨트롤
MIDAS	멀티-tiered 데이터베이스 어플리케이션을 작성할 때 사용하는 Midas 컴포넌트

이들 각각에 대해서 어떤 컴포넌트가 있는지 알아보도록 하자. 그렇지만 이 책의 성격상 이들에 대한 자세한 사용 방법을 소개하지는 않는다. 자세한 사항은 델파이가 제공하는 도움말을 참고하기 바라며, 델파이 4 에서 새롭게 제공되는 일부 컴포넌트나 사용방법이 많이 알려지지 않은 클래스의 사용방법과 테크닉에 대해서는 이장의 후반부에서 다루게 될 것이다.

#### ● Standard 탭의 컴포넌트들

여기서는 가장 일반적으로 사용되는 컴포넌트 들을 찾아볼 수 있다. 여기에는 다음 그림에서 보듯이 15 개의 컴포넌트들로 구성된다.



그러면, 각각의 컴포넌트의 역할에 대해서 간단하게 알아보도록 하자.

컴포넌트	설 명
TMainMenu	폼의 메뉴바나 드롭다운 메뉴를 작성하는데 사용되는 비시각적 컴포넌트이다.
TPopupMenu	사용자가 오른쪽 마우스 버튼을 눌렀을 때 나타나는 팝업 메뉴를 작성하고자 할 때 사용하는 비시각적 컴포넌트이다.
TLabel	라벨 컴포넌트는 폼이나 다른 컨테이너에 문자열을 나타내도록 하는데 사용한다. 사용자들은 이를 변경할 수 없다. 시각적 컴포넌트이다.
TEdit	Edit 컴포넌트는 사용자로부터 한 줄의 문자열을 입력받는데 사용한다. Edit

	컴포넌트는 문자열을 화면에 나타내는데 사용하기도 한다. 시각적 컴포넌트이다.
TMemo	여러 줄의 텍스트를 입력받고 화면에 나타내기 위하여 사용하는 시각적 컴포넌트이다.
TButton	버튼 컴포넌트는 버튼을 생성하는데 사용하며, 어플리케이션에서 어떤 항목을 선택할 때 사용한다. 시각적 컴포넌트이다.
TCheckBox	사용자가 체크박스를 선택하거나 선택을 취소하는 것이 가능하다. 시각적 컴포넌트이다.
TRadioButton	일련의 선택 항목들을 제시하고, 이들 중에서 하나 만을 선택하도록 할 때 사용한다. 이들 항목들의 수는 제한이 없으며, 폼, 패널 등을 컨테이너로 사용하는 시각적 컴포넌트이다.
TListBox	표준 윈도우 리스트 박스로서, 항목들의 리스트를 만들어서 사용자가 이들 중에 하나를 선택할 수 있는 시각적 컴포넌트이다.
TComboBox	리스트 박스와 유사하지만, 여기에 Edit 컴포넌트의 장점을 추가한 형태이다. 콤보 박스 컴포넌트는 사용자가 하나의 항목을 선택할 수도 있지만, 원하는 텍스트를 직접 입력할 수도 있는 시각적 컴포넌트이다.
TScrollBar	표준 윈도우 스크롤바로서 폼이나 컨트롤들을 스크롤하는데 사용하는 시각적 컴포넌트이다.
TGroupBox	라디오 버튼, 체크 박스 등의 컨테이너로서 관련된 컨트롤들을 하나의 그룹으로 구성하는데 사용되는 시각적 컴포넌트이다.
TRadioGroup	그룹 박스와 라디오 버튼이 조합된 컴포넌트로서 라디오 버튼의 그룹을 생성하고자 할 때 사용한다. 여러 개의 라디오 버튼들을 사용할 수는 있지만, 다른 컨트롤을 사용할 수는 없다. 시각적 컴포넌트이다.
TPanel	컨트롤이나 컨테이너들을 하나의 그룹으로 만들어주는 또 하나의 컨테이너 컴포넌트이다. 시각적 컴포넌트이다.
TActionList	컴포넌트와 컨트롤에 의해서 사용되는 메뉴 아이템과 버튼과 같은 실제 액션(action)의 리스트를 생성하고 관리하는 비시각적 컴포넌트이다.

# ● Additional 탭의 컴포넌트들

Additional 탭은 다음 그림과 같은 14 개의 컴포넌트들로 이루어져 있다. 이들 컴포넌트의 역할은 다음과 같다.



컴포넌트	설 명
TBitBtn	비트맵 그림을 포함하는 버튼을 만들 때 사용하는 시각적 컴포넌트이다.
TSpeedButton	패널 컴포넌트에 대하여 사용할 수 있도록 만든 특수한 버튼이다. 스피드 버튼은 툴바 등의 특별한 버튼의 그룹에 대해 사용할 수 있다. 시각적 컴포넌트이다.
TMaskEdit	특별한 형식의 데이터나 적당한 문자의 입력을 위해 사용하는 시각적 컴포넌트이다.
TStringGrid	행과 열에 문자열 데이터를 나타내는데 사용하는 시각적 컴포넌트이다.
TDrawGrid	행과 열에 텍스트 이외의 정보를 나타내고자 할 때 사용하는 시각적 컴포넌트이다.
TImage	아이콘, 비트맵, 메타파일 등의 그림을 나타내기 위해 사용하는 시각적 컴포넌트이다.
TShape	사각형, 원 등의 도형을 그리는데 사용하는 시각적 컴포넌트이다.
TBevel	3 차원으로 들어가거나 튀어나온 모양을 그리는데 사용되는 시각적 컴포넌트이다.
TScrollBar	스크롤이 가능한 화면표시 영역을 생성하기 위해 사용하는 시각적 컴포넌트이다.
TCheckListBox	Listbox 와 CheckBox 의 기능을 합쳐놓은 시각적 컴포넌트이다.
TSplitter	어플리케이션에서 사용자가 크기를 조절할 수 있도록 만들어 주는 패널을 만드는데 사용된다. 시각적 컴포넌트이다.
TStaticText	Label 컴포넌트와 유사하지만, 테두리 유형을 설정할 수 있는 등의 추가적인 기능을 가지고 있다. 시각적 컴포넌트이다.
TControlBar	툴바 컴포넌트 들의 형태를 유지하는데 역할을 한다. 툴바 컴포넌트의 도킹 site 로 사용되는 시각적 컴포넌트이다.
TChart	TeeChart 에서 제공되는 컴포넌트로, 차트나 그래프를 생성할 때 사용되는 시각적 컴포넌트이다.

## ● Win32 탭

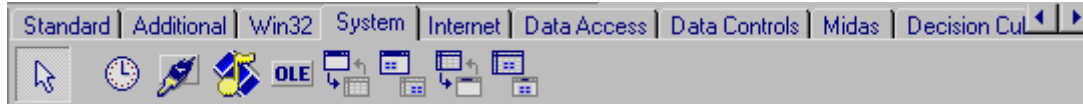
Win32 탭에는 윈도우 95/98/NT 4.0 과 유사한 어플리케이션을 작성하는데 필요한 18 개의 컴포넌트가 제공된다. 이들 중의 몇 가지는 Win3.1 탭의 컴포넌트와 유사하다. 이들 컴포넌트의 역할은 다음과 같다.



컴포넌트	설 명
TTabControl	윈도우 95 형식의 탭 컴포넌트로서 폼에 사용자가 선택할 수 있는 탭들을 추가하기 위해서 사용하는 시각적 컴포넌트이다.
TPageControl	윈도우 95 형식의 컴포넌트로서, 탭이나 다른 컨트롤로 바꿀 수도 있으며, 화면의 공간을 절약하여 사용할 수 있다. 시각적 컴포넌트이다.
TImageList	이미지의 리스트를 관리하기 위한 새로운 객체이다.
TRichEdit	윈도우 95 형식의 컴포넌트로서, 여러가지 색깔이나 폰트, 텍스트 검색 등을 지원하는 향상된 메모 컴포넌트라고 생각하면 된다. 시각적 컴포넌트이다.
TTrackBar	윈도우 95 형식의 슬라이더(slider) 컨트롤이다. 시각적 컴포넌트이다.
TProgressBar	윈도우 95 형식의 진행상태를 나타내는 바로서 시각적 컴포넌트이다.
TUpDown	윈도우 96 형식의 스피너 버튼 컨트롤이다. 시각적 컴포넌트이다.
THotKey	어플리케이션에 추가적인 단축 키를 지원하도록 하기 위해 사용하는 시각적 컴포넌트이다.
TAnimate	윈도우 95 에서 파일을 복사 또는 이동할 때 나타나는 것과 같은 AVI 동영상 클립을 재생시켜 주기 위하여 사용하는 시각적 컴포넌트이다.
TDateTimePicker	ComboBox 와 유사한 컴포넌트로서, 달력을 나타내어 데이터를 선택하도록 하는 시각적 컴포넌트이다.
TMonthCalendar	사용자가 날짜나 날짜의 범위를 선택할 수 있는 달력 컴포넌트이다. 시각적 컴포넌트이다.
TTreeView	윈도우 95 형식의 컴포넌트로서, 데이터를 계층 구조의 형식으로 나타내는데 사용되는 시각적 컴포넌트이다.
TListView	윈도우 95 형식의 컴포넌트로서, 리스트 들을 이미지를 가진 열(column)로 나타내는데 사용하는 시각적 컴포넌트이다.
THeaderControl	윈도우 95 형식의 컴포넌트로서, 여러 개의 이동이 가능한 헤더를 생성하기 위해 사용하는 시각적 컴포넌트이다.
TStatusBar	윈도우 95 형식의 컴포넌트로서, 상황에 대한 정보를 여러 개의 패널에 나누어 나타나도록 하는데 사용하는 시각적 컴포넌트이다.
TToolBar	어플리케이션에서 자주 사용되는 기능을 빠르게 사용할 수 있도록 하기 위한 툴바를 생성하기 위해 사용되는 시각적 컴포넌트이다.
TCoolBar	사용자가 크기 조절을 할 수 있도록 밴드를 컴포넌트에 추가한 시각적 컴포넌트이다.
TPageScroller	툴바와 같이 좁은 윈도우 영역의 디스플레이 방법을 정의하는 시각적 컴포넌트이다. 윈도우가 디스플레이 영역보다 큰 경우, 화살표를 윈도우의 끝 부분에 표시하여 디스플레이 영역을 스크롤하여 볼 수 있도록 해준다.

- VCL 의 System 탭

System 탭에는 윈도우의 장점을 최대한으로 이용하는데 도움을 주는 다음 그림과 같은 8 개의 컴포넌트로 이루어져 있다. 이들 컴포넌트의 역할은 다음과 같다.



컴포넌트	설 명
TTimer	일정한 시간 간격으로 프로시저나 함수, 이벤트 들을 사용하는데 사용되는 비 시각적 컴포넌트이다.
TPaintBox	그림을 그릴 수 있는 폼의 영역에서 그림을 그리게 위해 사용하게 되는 시각적 컴포넌트이다.
TMediaPlayer	VCR 과 같은 형태의 패널을 생성하여, 소리나 비디오 파일 들을 재생하는데 사용하는 시각적 컴포넌트이다.
TOleContainer	OLE 클라이언트 영역을 생성하는데 사용하는 시각적 컴포넌트이다.
TDDEClientConv	DDE 서버와의 통신을 설정하는데 사용되는 비시각적 컴포넌트이다.
TDDEClientItem	DDE 통신 중에 DDE 서버에게 전송하고자하는 클라이언트 데이터를 지정하는데 사용하는 비시각적 컴포넌트이다.
TDDEServerConv	DDE 서버 어플리케이션이 DDE 클라이언트와 통신을 하는데 사용되는 비시각적 컴포넌트이다.
TDDEServerItem	통신 중에 DDE 클라이언트에 전송하고자하는 데이터를 지정하는데 사용하는 비시각적 컴포넌트이다.

- Internet 탭

인터넷 탭에는 인터넷이나 TCP/IP 네트워크 환경의 어플리케이션을 작성하기 위한 작업을 도와주는 26 개의 컴포넌트 들이 포함되어 있다. 이들은 다음과 같은 기능을 한다.



컴포넌트	설 명
TClientSocket	네트워크 상의 다른 기계와 연결을 생성하기 위하여 사용된다. 소켓 프로그래밍을 저수준으로 할 수 있도록 지원하는 비시각적 컴포넌트이다.
TServerSocket	네트워크 상의 다른 기계로부터의 클라이언트 요청에 응답하기 위해 사용

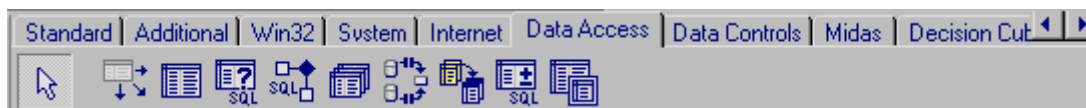


	된다. 소켓 프로그래밍을 지원하는 비시각적 컴포넌트이다.
TWebDispatcher	일반적인 데이터 모듈을 웹 모듈로 변환시키는데 사용되는 비시각적 컴포넌트이다.
TPageProducer	웹 브라우저나 다른 HTML 뷰어 어플리케이션에서 볼 수 있도록, HTML 템플릿을 HTML 문자열 코드로 변환시키는데 사용되는 비시각적 컴포넌트이다.
TQueryTableProducer	TQuery 객체로부터 HTML 테이블을 생성하는데 사용되는 비시각적 컴포넌트이다.
TDataSetTableProducer	TDataSet 객체의 레코드들로부터 HTML 테이블을 생성하는데 사용된다.
TDataSetPageProducer	필드 데이터를 포함한 HTML 템플릿에 기초한 HTML 문자열 코드를 생성하는데 사용되는 비시각적 컴포넌트이다.
TNMDatTime	인터넷 daytime 서버에서 날짜와 시간을 얻어오는데 사용되는 비시각적 컴포넌트이다.
TNMEcho	인터넷 echo 서버에 텍스트를 전송하는데 사용되는 컴포넌트로, 전송하는 텍스트는 다시 돌아온다. 주로 네트워크 무결성과 속도를 측정하기 위해 사용되는 비시각적 컴포넌트이다.
TNMFinger	사용자에 대한 정보를 인터넷 finger 서버에서 얻어오는데 사용되는 비시각적 컴포넌트이다.
TNMFTP	파일을 인터넷 FTP 서버에 FTP 프로토콜을 이용하여 전송할 때 사용되는 비시각적 컴포넌트이다.
TNMHTTP	인터넷을 통해 HTTP 전송을 수행하는데 이용되는 비시각적 컴포넌트이다.
TNMMsg	인터넷에 TCP/IP 프로토콜을 이용해서 간단한 ASCII 텍스트 메시지를 전송하는데 사용되는 비시각적 컴포넌트이다.
TNMMsgServ	TNMMsg 컴포넌트가 전송한 메시지를 받아서 이를 사용할 수 있도록 해주는 비시각적 컴포넌트이다.
TNMNNTP	인터넷 뉴스 기사를 뉴스 서버에서 읽거나, 전송하는데 사용되는 비시각적 컴포넌트이다.
TNMPOP3	인터넷 E-mail 을 POP3 서버에서 받아올 때 사용되는 비시각적 컴포넌트이다.
TNMUUProcessor	MIME 또는 UUEncode 된 파일을 코딩화 하거나 해독화할 때 사용되는 비시각적 컴포넌트이다.
TNMSMTP	인터넷 메일 서버에 E-mail 을 전송할 때 사용되는 비시각적 컴포넌트이다.
TNMStrm	인터넷을 통해 스트림 서버에 스트림을 전송하는데 사용되는 비시각적 컴포넌트이다.

TNMStrmServ	TNMStrm 컴포넌트에서 전송한 스트림을 받아서 사용할 수 있도록 해주는 스트림 서버 컴포넌트이다. 비시각적 컴포넌트이다.
TNMTime	TNMDateTime 컴포넌트와 비슷하나, 시간을 얻어올 뿐이다.
TNMUDP	UDP 프로토콜을 이용하여 데이터 그램 패킷을 인터넷에 전송할 수 있도록 구현된 비시각적 컴포넌트이다.
TPowerSock	Internet 탭에 있는 많은 컴포넌트의 기초가 되는 클래스로, 구현되지 않은 다른 프로토콜이나 사용자 정의 프로토콜을 구현할 때 사용할 수 있는 비시각적 컴포넌트이다.
TNMGeneralServer	멀티 쓰레드 인터넷 서버를 개발하기 위한 기초 클래스로 제공되는 컴포넌트이다. RFC 표준을 따르는 서버이든, 사용자가 나름대로 서비스를 제공하는 서버이든 이를 구현하는데 유용하게 사용할 수 있는 비시각적 컴포넌트이다.
THTML	웹 브라우저 등에서 사용하는 HTML 코드의 내용을 HTML 페이지로 나타내어 주는데 사용되는 시각적 컴포넌트이다.
TNMURL	URL 데이터를 읽기 쉬운 문자열로 해독하거나, 표준 문자열을 URL 데이터 포맷으로 코딩화하는 작업을 하는 비시각적 컴포넌트이다.

#### ● Data Access 탭

Data Access 탭은 다음과 같이 데이터베이스 연결하고 통신하는데 사용되는 9 개의 컴포넌트로 이루어져 있다. 이들의 역할은 다음과 같다.



컴포넌트	설 명
TDataSource	테이블이나 쿼리 컴포넌트들을 데이터 컨트롤과 연결시키기 위하여 사용하게 된다. 비시각적 컴포넌트이다.
TTable	데이터베이스 테이블을 어플리케이션에 연결시켜 주기 위하여 사용하는 비시각적 컴포넌트이다.
TQuery	원격 SQL 서버나 지역 데이터베이스에 대하여 SQL 쿼리를 실행시키거나 생성하는데 사용하는 비시각적 컴포넌트이다.
TStoredProc	SQL 서버에 저장되어 있는 프로시저를 실행시키기 위해 사용하는 비시각적 컴포넌트이다.
TDatabase	원격 데이터베이스 서버와 연결을 하기 위하여 사용하는 비시각적 컴포넌

	트이다.
TSession	어플리케이션의 데이터베이스 연결에 대한 전반적인 제어를 위하여 사용하는 비시각적 컴포넌트이다.
TBatchMove	지역적으로 작업한 레코드와 테이블 들로 서버로 올려서 서버의 정보를 갱신하는데 사용하는 비시각적 컴포넌트이다.
TUpdateSQL	SQL 데이터베이스를 갱신하기 위해 사용되는 비시각적 컴포넌트이다.
TNestedTable	오라클 8 에서부터 지원되는 중첩된 테이블을 지원하는 비시각적 컴포넌트이다.

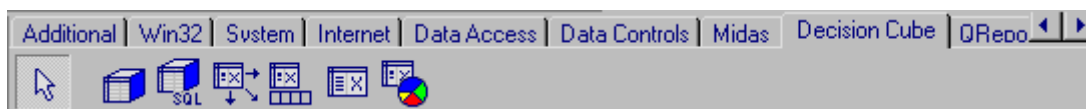
### ● Data Controls 탭

DataControls 탭은 15 개의 데이터 컨트롤을 모아 놓은 페이지이다. 이들 대부분은 Standard 나 Additional 탭에서 볼 수 있는 컴포넌트들을 데이터베이스와 함께 사용할 수 있도록 수정한 것이다. 특별히 설명할 것이 없으므로 이들에 대한 설명은 생략하겠다.



### ● Decision Cube 탭

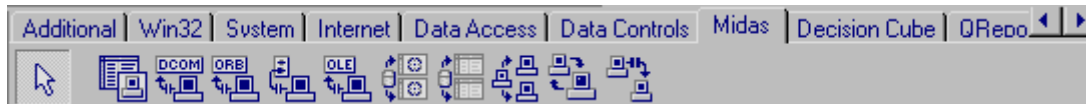
Decision Cube 탭은 데이터 분석에 유용하게 사용할 수 있는 6 개의 다차원 차트와 그래프 컴포넌트를 포함하고 있다. 이들에 대해서 간단히 알아보면 다음과 같다.



컴포넌트	설 명
TDecisionCube	다차원의 데이터를 저장하는 저장소로서, 데이터 집합으로부터 데이터를 불러들이기 위해 사용되는 비시각적 컴포넌트이다.
TDecisionQuery	TQuery 컴포넌트를 DecisionCube 에서 사용할 수 있도록 수정한 비시각적 컴포넌트이다.
TDecisionSource	DecisionGrid 나 DecisionGraph 컴포넌트의 현재 피벗(pivot) 상태를 정의하기 위해 사용하는 비시각적 컴포넌트이다.
TDecisionPivot	버튼을 통하여, DecisionCube 의 차원이나 필드를 열거나 닫도록 하는데 사용되는 시각적 컴포넌트이다.

TDecisionGrid	DecisionCube 컴포넌트의 데이터를 그리드의 형식으로 나타내는데 사용하는 시각적 컴포넌트이다.
TDecisionGraph	DecisionCube 컴포넌트의 데이터를 그래프로 나타내는데 사용되는 시각적 컴포넌트이다.

● Midas 탭



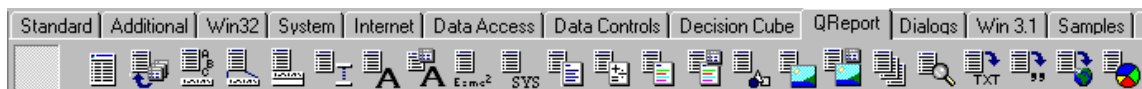
Midas 탭의 컴포넌트 들은 멀티-tiered 데이터베이스 어플리케이션 작성에 필요한 10 개의 컴포넌트로 구성되어 있다. 이들의 역할에 대한 간단한 설명을 하면 다음과 같다.

컴포넌트	설 명
TClientDataSet	데이터베이스에 독립적인 데이터 세트로 1-tiered 어플리케이션을 작성하거나, 멀티-tiered 데이터베이스 어플리케이션을 사용할 때 사용되는 비시각적 컴포넌트이다.
TDCOMConnection	멀티-tiered 데이터베이스 어플리케이션에서 DCOM 원격 서버에 접속하는 방법을 제공하는 비시각적 컴포넌트이다.
TCorbaConnection	멀티-tiered 데이터베이스 어플리케이션에서 CORBA 원격 서버에 접속하는 방법을 제공하는 비시각적 컴포넌트이다.
TSocketConnection	멀티-tiered 데이터베이스 어플리케이션에서 TCP/IP 원격 서버에 접속하는 방법을 제공하는 비시각적 컴포넌트이다.
TOLEEnterpriseConnection	멀티-tiered 데이터베이스 어플리케이션에서 OLEEnterprise 원격 서버에 접속하는 방법을 제공하는 비시각적 컴포넌트이다.
TDataSetProvider	데이터 세트의 데이터를 코딩화하여 클라이언트 어플리케이션으로 전송될 패킷을 생성하며, 클라이언트 어플리케이션의 업데이트를 반영하는 역할을 하는 비시각적 컴포넌트이다.
TProvider	원격 서버의 데이터베이스 어플리케이션 서버와 클라이언트 데이터 세트 간에 연결을 제공하는 비시각적 컴포넌트이다.
TSimpleObjectBroker	가능한 어플리케이션 서버의 리스트에서 접속 컴포넌트를 서버에 위치시키는 역할을 하는 비시각적 컴포넌트이다.
TRemoteServer	멀티-tiered 어플리케이션의 원격 서버와 DCOM 접속을 위한 비시각적 컴포넌트이다. 델파이 3 와의 호환성을 위해 사용될 뿐이다.
TMIDASConnection	멀티-tiered 어플리케이션의 원격 서버와 DCOM, TCP/IP, OLEEnterprise

	접속을 위한 비시각적 컴포넌트로, 델파이 3 와의 호환성을 위해 사용될 뿐이다.
--	--

- QReport 탭

QReport 탭은 리포트의 작성을 위한 23 개의 컴포넌트 들로 구성된다. 이들 컴포넌트의 사용법에 대해서는 40 장에 대해서 자세하게 다루게 될 것이므로 설명은 생략한다.



- Dialogs 탭

Dialogs 탭은 윈도우에서 여러가지 대화상자를 생성하는데 사용되는 10 가지 컴포넌트들로 구성된다. 이들에 대해 간단히 설명하면 다음과 같다.



컴포넌트	설 명
TOpenDialog	파일 열기 대화상자를 생성하는데 사용되는 비시각적 컴포넌트이다.
TSaveDialog	파일 저장 대화상자를 생성하는데 사용되는 비시각적 컴포넌트이다.
TOpenPictureDialog	그림 열기 대화상자를 생성하는데 사용되는 비시각적 컴포넌트이다.
TSavePictureDialog	그림 저장 대화상자를 생성하는데 사용되는 비시각적 컴포넌트이다.
TFontDialog	글꼴 선택 대화상자를 생성하는데 사용되는 비시각적 컴포넌트이다.
TColorDialog	색깔 선택 대화상자를 생성하는데 사용되는 비시각적 컴포넌트이다.
TPrintDialog	인쇄 대화상자를 생성하는데 사용하는 비시각적 컴포넌트이다.
TPrinterSetupDialog	프린터 설정 대화상자를 생성하는데 사용하는 비시각적 컴포넌트이다.
TFindDialog	찾기 대화상자를 생성하는데 사용하는 비시각적 컴포넌트이다.
TReplaceDialog	바꾸기 대화상자를 생성하는데 사용하는 비시각적 컴포넌트이다.

- Win 3.1 탭

델파이 1.0 과 2.0 간의 어플리케이션 전환을 위한 호환성을 위해 제공되고 있는 컴포넌트 들이다. 더 이상 사용하지 않는 것이 좋을 것으로 생각되므로, 이 탭에 대해서는 특별히 따로 설명하지 않는다.

- Samples 탭

Samples 탭에는 6 개의 VCL 들을 포함하고 있지만, 이들은 최소한의 설명과 함께 예제로서 포함되어 있는 것이다. 이들의 소스 코드들은 Delphi 4\Source\Samples 디렉토리에 찾아볼 수 있다. 이들에 대해 간단히 설명하면 다음과 같다.



컴포넌트	설 명
TGuage	사각형과 텍스트 또는 파이 형식으로 진행 정도를 나타내는 게이지를 표시해주는 시각적 컴포넌트이다.
TColorGrid	사용자가 색깔을 선택할 수 있는 그리드를 생성하는데 사용되는 시각적 컴포넌트이다.
TSpinButton	스핀 버튼을 생성하는데 사용되는 시각적 컴포넌트이다.
TSpinEdit	스핀 컨트롤과 Edit 상자 컴포넌트의 기능을 합쳐놓은 형태의 시각적 컴포넌트이다.
TDirectoryOutline	선택된 드라이브의 디렉토리 구조를 나타내는데 사용되는 시각적 컴포넌트이다.
TCalendar	날짜 정보를 나타내는 달력을 화면에 표시하기 위해 사용되는 시각적 컴포넌트이다.
TIBEventAlerter	이벤트 경고 컴포넌트이다. 비시각적 컴포넌트이다.

- ActiveX 탭

ActiveX 탭에는 5 가지 ocx 예제 컨트롤들이 포함되어 있다. 그렇지만, 델파이 프로젝트에는 VCL 컨트롤을 사용하는 것이 장점이 많으며, 거의 사용될 일이 없으므로 이 탭에 대해서는 특별히 따로 설명하지 않는다.

## 드래그-드롭(Drag-and-Drop)의 구현

드래그-드롭은 일반적인 컨트롤에서는 대부분이 지원하게 되는 특징이다. 먼저 간단하게 드래그-드롭을 구현하는 방법에 대해서 알아보고, 실제 예제를 하나 작성해 보도록 하자.

- 드래그-드롭의 구현 방법

## 1. 드래그 작업의 시작

모든 컨트롤은 DragMode 라는 프로퍼티를 가지고 있는데, 이 프로퍼티는 사용자가 컴포넌트를 런타임에서 드래그를 시작할 때 어떤 식으로 반응할 지를 결정한다. dmAutomatic 인 경우 끌기는 사용자가 컨트롤 위에서 마우스 버튼을 누르는 순간 자동적으로 시작된다. dmAutomatic 은 정상적인 마우스의 활동을 제한할 수 있기 때문에, 보통은 디폴트 값인 dmManual 을 사용하며 대신 OnMouseDown 이벤트를 사용하여 드래그를 시작한다.

드래그를 시작하려면 컨트롤의 BeginDrag 메소드를 호출하면 된다. BeginDrag 메소드에는 Immediate 라는 Boolean 형의 파라미터가 있는데, 이를 True 로 설정하면 드래그가 즉시 시작되므로, DragMode 를 dmAutomatic 으로 설정한 것과 별반 차이가 없는 것이 된다. 그에 비해 False 로 값을 설정한 경우 사용자가 마우스를 가지고 조금이라도 끌기 전에는 d 드래그가 시작되지 않는다. 그러므로, BeginDrag(False)를 호출하는 것이 일반적인 마우스의 클릭을 드래그 작업으로 간주하지 않게 되므로 더 좋다.

드래그를 시작하기 전에 어떤 버튼이 사용자에게 의해 눌러 졌는지 검사함으로써 드래그를 시작할 지 결정하게 되는 경우도 많은데, 이럴 때에는 OnMouseDown 이벤트에서 Button 파라미터를 검사하여 적절한 버튼이라면 드래그를 시작하도록 설정할 수 있다.

다음의 코드는 파일 리스트 박스의 OnMouseDown 이벤트에서 좌측 마우스 버튼이 눌린 경우에만 드래그를 시작하도록 하는 코드이다.

```
procedure TForm1.FileListBox1MouseDown(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Button = mbLeft then {좌측 버튼일 때에만 drag}  
        with Sender as TFileListBox do  
            begin  
                if ItemAtPos(Point(X, Y), True) >= 0 then {위치에 Item 이 있으면 ... }  
                    BeginDrag(False);  
            end;  
        end;  
end;
```

## 2. 드래그된 아이템을 받아들일지 여부 결정

사용자가 아이템을 드래그하여 어떤 컨트롤의 위를 지날 때에 그 컨트롤은 OnDragOver 이벤트를 발생시킨다. 이때 이 컨트롤이 드롭을 지원할 지를 결정하게되는데, 델파이는 컨트롤이 드롭을 지원하는지 여부를 드래그 커서의 모양을 바꿔서 사용자에게 알리게 된다.

드래그를 허용한다면 컨트롤의 OnDragOver 이벤트의 이벤트 핸들러를 작성해야 한다. OnDragOver 이벤트에는 Accept 라는 variable 파라미터가 있는데, 이 값을 True 로 설정하면 아이템을 드롭할 수 있도록 허용한다는 의미가 된다. 만약 이 값을 False 로 설정되면 이 컨트롤이 현재의 아이템에 대한 드롭을 허용하지 않겠다는 의미가 된다. OnDragOver 이벤트에는 이런 판단을 하는데 도움을 주는 드래그의 source, 마우스 커서의 현재 위치 등의 몇 가지 파라미터가 있다. 다음 코드는 디렉토리 트리뷰에 FileListBox 에서 온 아이템의 드롭 만을 허용하게 된다.

```
procedure TForm.DirectoryOutline1DragOver(Sender, Source: TObject; X, Y: Integer;
    State: TDragState; var Accept: Boolean);
begin
    if Source is TFileListBox then
        Accept := True;
    else
        Accept := False;
end;
```

### 3. 아이템의 드롭

일단 컨트롤이 드롭을 허용하면, 실제로 아이템이 드롭되었을 때 이를 처리하는 것은 OnDragDrop 이벤트이다. OnDragOver 이벤트와 마찬가지로 OnDragDrop 이벤트 역시 드래그된 아이템의 source 와 마우스 커서의 위치를 이용하여 여러가지 작업을 하게 된다.

### 4. 드래그 작업의 종료

드래그 작업이 끝나면, 드래그가 시작된 컴포넌트에 OnEndDrag 이벤트가 발생한다. OnEndDrag 이벤트의 파라미터 중에서 가장 중요한 것은 Target 을 여기에는 어떤 컨트롤이 드롭을 받았는지가 전달된다. 이 값이 nil 인 경우 드롭을 받아들인 컴포넌트가 없다는 의미이다. 그리고, 파라미터에는 드롭이 일어난 컨트롤의 위치를 나타내는 X, Y 좌표가 전달된다.

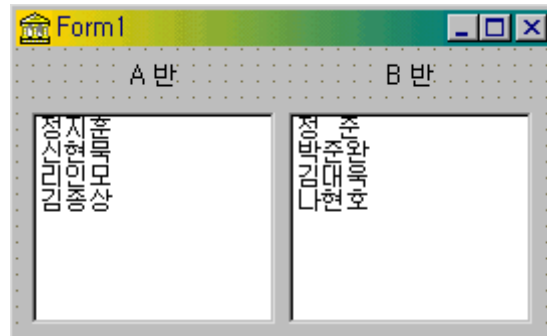
### 5. 드래그 마우스 포인터 변경

드래그를 할 때 마우스의 형태를 변경하려면 해당 컴포넌트의 DragCursor 프로퍼티를 변경해주면 된다.



- 드래그-드롭을 구현한 간단한 예제

간단한 예제로 두 개의 리스트 박스 사이에 문자열을 드래그-드롭으로 이동하도록 해보자. 먼저 폼에 다음과 같이 2 개의 리스트 박스와 2 개의 라벨 컴포넌트를 올려놓고, Label1, Label2 의 Caption 프로퍼티를 각각 ‘A 반’, ‘B 반’으로 설정한다. 그리고, ListBox1 과 ListBox2 의 Items 프로퍼티를 편집하여 다음 그림과 같은 내용을 입력한다.



그리고, 코드 에디터에서 드래그-드롭으로 옮겨 다닐 문자열의 내용을 저장할 전역변수인 SelectedText 를 다음과 같이 선언하도록 한다.

```
var
  Form1: TForm1;
  SelectedText: string;
```

그럼, 이제 드래그-드롭을 구현해 보자. 디폴트로 DragMode 프로퍼티는 dmManual 로 설정되어 있으므로, 첫번째로 해야 할 일은 ListBox1 의 OnMouseDown 이벤트 핸들러를 작성하는 것이다. OnMouseDown 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.ListBox1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Index: Integer;
begin
  SelectedText := ''; {선택된 문자열을 초기화}
  if Button = mbLeft then {좌측 버튼일 때에만 drag}
  with Sender as TListBox do
  begin
    Index := ItemAtPos(Point(X, Y), True); {커서 위치의 아이템의 인덱스를 구한다}
```

```

if Index >= 0 then                                {커서 위치에 Item 이 있으면 ... }
begin
    BeginDrag(False);
    SelectedText := Items.Strings[Index];    {선택된 문자열을 커서 위치의 아이템으로 설정}
end;
end;
end;

```

여기서 주의해서 관찰할 것은 일단 TListBox 의 ItemAtPost 메소드를 이용해서 현재 커서 위치에 아이템이 있는지를 알아본 후, 있다면(Index >= 0) BeginDrag(False) 메소드를 호출하여 드래그를 시작한다는 것이다. 그리고, 그 위치의 아이템의 문자열을 전역변수인 SelectedText 에 대입하여 이 문자열을 드롭할 때 사용하게 된다.

여기서 ItemAtPos 메소드는 첫번째 파라미터에 TPoint 형으로 위치를 전달하고 두번째 파라미터에 True 를 설정하면, 지정된 위치에 아이템이 존재하면 그 아이템의 인덱스를 반환하며, 아이템이 없으면 -1 을 반환한다.

드래그가 시작되었으면, ListBox2 에 드롭을 하게 될 것이므로 드롭을 받아들일 것인지 여부를 OnDragOver 이벤트에서 결정해 주어야 한다. ListBox2 의 OnDragOver 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.ListBox2DragOver(Sender, Source: TObject; X, Y: Integer;
    State: TDragState; var Accept: Boolean);
begin
    if (Source as TListBox).Name = 'ListBox1' then Accept := True
    else Accept := False;
    ListBox2.ItemIndex := ListBox2.ItemAtPos(Point(X, Y), True);
end;

```

이 코드에서 Source 파라미터는 드래그가 시작된 컨트롤을 가리킨다. 즉, 드래그가 시작된 컨트롤의 이름이 'ListBox1'인 경우에만 드롭을 받아들일 것이라는 의미이다. 그러므로, ListBox1 에서 드래그를 해서 ListBox2 에 드롭하는 것은 허용하겠지만, ListBox2 에서 드래그를 시작하거나 탐색기와 같은 다른 프로그램에서 드롭을 하려고 하면 ListBox2 에서는 받아들이지 않는다는 뜻이다.

마지막 줄의 TListBox 의 ItemIndex 프로퍼티는 선택된 아이템의 위치를 지정하는 것으로, 이 코드는 커서 위치에 따라 선택된 아이템의 위치를 변경하라는 코드이다.

그러면, ListBox2 의 OnDragDrop 이벤트 핸들러에서 드롭이 되었을 때 작업을 마무리하는 이벤트 핸들러를 작성해 보자.

```

procedure TForm1.ListBox2DragDrop(Sender, Source: TObject; X, Y: Integer);
var
    Index: Integer;
begin
    Index := ListBox2.ItemAtPos(Point(X, Y), True);
    if Index >= 0 then ListBox2.Items.Insert(Index, SelectedText)
    else ListBox2.Items.Add(SelectedText);
    ListBox1.Items.Delete(ListBox1.Items.IndexOf(SelectedText));
end;

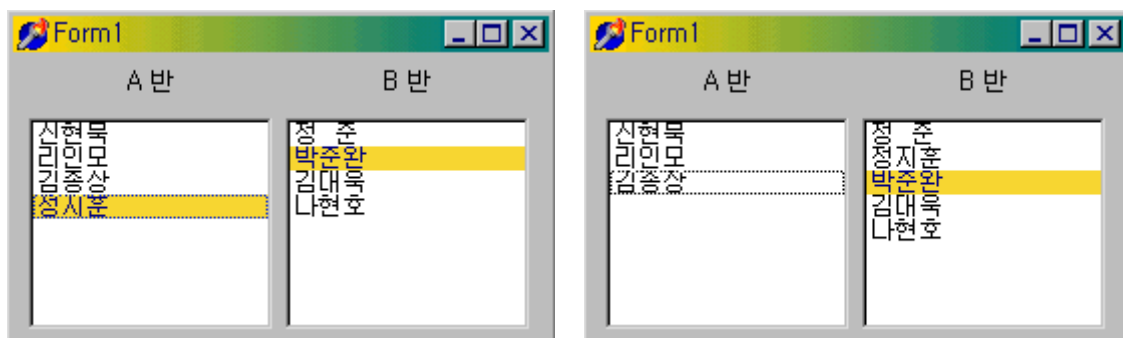
```

일단 현재 드롭되려는 위치의 아이템의 위치를 ItemAtPos 메소드를 이용해서 Index 변수에 저장한다. Index 변수의 값이 0 보다 크거나 같다면 커서의 위치에 아이템이 존재한다는 의미이므로 Items.Insert 메소드를 이용해서 현재 위치에 SelectedText 변수의 문자열(드래그하는 문자열)을 삽입한다. Index 변수의 값이 0 보다 작다면 커서의 위치에 아이템이 존재하지 않는 것이므로 제일 마지막에 SelectedText 변수의 문자열을 Items.Add 메소드를 이용해서 추가하면 된다.

그리고, 이렇게 추가하고 나면 ListBox1 에서 옮겨온 아이템은 삭제해 주어야 이동이 완료된다. 리스트 박스의 각각의 메소드에 대한 자세한 사용법은 도움말을 참고하기 바란다.

이것으로 ListBox1 에서 ListBox2 로의 드래그-드롭은 모두 구현되었다. ListBox2 에서 ListBox1 으로의 드래그-드롭의 구현도 완전히 똑같은 방법으로 하면 된다. 다만, ListBox1 을 가리키는 것은 ListBox2 로 ListBox2 는 ListBox1 으로 바꾸어 주기만 하면 된다. 이벤트 역시 반대로 적용해서 구현하면 된다. 지면 관계상 같은 내용을 반복해서 설명하지는 않는다. 자세한 것은 소스 코드를 참고하기 바란다.

제대로 했을 것으로 믿고, 실행을 한번 해보자 다음과 같이 드래그 드롭할 수 있으면 제대로 실행된 것이다.



## 드래그-도크(Drag-and-dock)의 구현

텔파이 4 의 TControl 과 TWinControl 은 기본적으로 MS 오피스 97 에서 볼 수 있는 형태의 도킹 윈도우를 지원하게 되었다. 모든 TWinControl 의 자손 컨트롤은 도킹 site 로 사용될 수 있으며, 모든 TControl 의 자손은 도킹 site 에 도킹하는 자식 윈도우로 사용될 수 있다. 예를 들어, 폼의 좌측 면을 도킹 site 로 사용하려면, 패널을 좌측으로 정렬하게 한 뒤 이 패널을 도킹 site 로 만들어 버리면 된다. 이때 도킹이 가능한 컨트롤을 패널 옆으로 끌어다 놓으면 이 윈도우는 패널의 자식 윈도우로 도킹이 된다.

### ● 드래그-도크의 구현 방법

#### 1. 윈도우 컨트롤을 도킹 site 로 만든다.

윈도우 컨트롤을 도킹 site 로 만들려면 다음과 같이 한다.

- 컨트롤의 DockSite 프로퍼티를 True 로 설정한다.
- 도킹 site 객체가 도킹 클라이언트를 포함할 때에만 나타나게 하려면, AutoSize 프로퍼티를 True 로 설정한다. 이렇게 하면 도킹된 자식 컨트롤이 없을 때에 도킹 site 의 크기가 0 이 되기 때문에 언제나 도킹될 컨트롤의 크기에 맞출 수가 있다.

#### 2. 도킹할 자식 컨트롤을 도킹이 가능하도록 설정한다.

- DragKind 프로퍼티를 dkDock 으로 설정한다. 이렇게 하면 컨트롤을 드래그할 때 도킹 site 근처에 가면 그 위치로 컨트롤이 이동해 가며, 도킹된 것을 떼어낼 때에는 플로팅 윈도우(floating window)가 된다. DragKind 프로퍼티가 dkDrag(디폴트)인 경우에는 드래그할 때 드래그-드롭 작업이 시작된다.
- DragMode 프로퍼티를 dmAutomatic 으로 설정한다. 이렇게 하면, 사용자가 컨트롤을 마우스로 드래그하기 시작하면 자동으로 드래그 작업으로 들어간다. 이 프로퍼티가 dmManual 로 설정된 경우에는 BeginDrag 메소드를 호출해야 한다.
- FloatingDockSiteClass 프로퍼티는 컨트롤이 도킹에서 떨어질 경우나 플로팅 윈도우가 되었을 때 컨트롤의 주인이 될 TWinControl 의 자손 클래스로 설정한다. 컨트롤이 도킹 site 에서 떨어져 나오면, 이 클래스의 윈도우 컨트롤이 동적으로 생성되서, 도킹 윈도우의 부모가 된다. 도킹 윈도우가 TWinControl 의 자손일 경우에는 컨트롤의 주인이 될 분리된 플로팅 도킹 site 를 생성할 필요는 없다. 이럴 때에는 컨트롤과 같은 클래스로 지정하면, 플로팅 윈도우가 될 때 특별한 부모가 없이 생성된다.

### 3. 컨트롤이 도킹 site 에 도킹하는 방법을 조정한다.

도킹 site 는 자식 컨트롤이 놓여질 때 자동으로 이를 받아들인다. 대부분의 컨트롤에서 보면 첫번째 자식이 클라이언트 영역을 모두 채우게 도킹된 경우, 두번째 자식은 영역을 분리시키곤 한다. 이렇게 세밀하게 도킹 site 에 자식 윈도우가 도킹하는 방법을 제어하기 위해 OnGetSiteInfo, OnDockOver, OnDockDrop 의 3 가지 이벤트가 사용된다. 이들 이벤트에 대해 잠시 알아보도록 하자.

먼저 OnGetSiteInfo 이벤트의 선언부와 역할에 대해서 알아보자. OnGetSiteInfo 이벤트의 선언부는 다음과 같다.

```
property OnGetSiteInfo: TGetSiteInfoEvent;
```

```
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var InfluenceRect: TRect;  
    var CanDock: Boolean) of object;
```

OnGetSiteInfo 이벤트는 도킹 site 에 도킹될 자식 컨트롤을 사용자가 드래그하면 발생하게 되는데, 여기에서 site 가 DockClient 파라미터에 지정된 컨트롤을 받아들일 것인지 여부와 받아들인다면 자식 컨트롤이 도킹할 위치를 지정할 수 있다. OnGetSiteInfo 이벤트가 발생하면 InfluenceRect 는 도킹 site 의 스크린 위치로 설정되며, CanDock 파라미터는 True 로 초기화된다. 만약 경우에 따라서 자식 컨트롤의 도킹을 허용하지 않으려면 CanDock 를 False 로 설정하고, InfluenceRect 의 영역을 제한함으로써 제한된 도킹을 지원하게 할 수 있다.

그러면 이번에는 OnDockOver 이벤트에 대해서 알아보도록 하자. 선언부는 다음과 같다.

```
property OnDockOver: TDockOverEvent;
```

```
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer;  
    State: TDragState; var Accept: Boolean) of object;
```

OnDockOver 이벤트는 사용자가 자식 컨트롤을 도킹 site 컨트롤 위로 드래그할 때 도킹 site 컨트롤에서 발생한다. 이 이벤트는 기존의 drag-and-drop 에서 사용되었던 OnDragOver 이벤트의 역할과 유사하다. 즉, 도킹을 위해 자식 컨트롤이 release 될 때 Accept 파라미터를 설정하여 이를 받아들이거나 거부한다. 만약 도킹이 가능한 컨트롤이 OnGetSiteInfo 이벤트 핸들러에서 거부된 경우에는 OnDockOver 이벤트는 발생하지 않는다.

마지막으로 OnDockDrop 이벤트에 대해 알아보자. 선언부는 다음과 같다.

```
property OnDockDrop: TDockDropEvent;
```

TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer) of object;

OnDockDrop 이벤트는 사용지기 자식 컨트롤을 도킹 site 에 올려 놓을 때 발생한다. 정상적인 drag-and-drop 작업의 OnDragDrop 이벤트와 유사한 역할을 하는데, 이 이벤트를 이용하여 실제로 자식 컨트롤이 도킹될 때 적절한 설정을 가능하게 할 수 있다. Source 파라미터가 가리키는 컨트롤이 도킹될 자식 컨트롤이므로, 이를 이용하여 여러가지 설정이 가능하다.

#### 4. 자식 컨트롤의 도킹이 해제될 때의 조정 방법

도킹 site 는 자식 컨트롤을 드래그하면 DragMode 프로퍼티가 dmAutomatic 이라면 자동으로 도킹이 해제된다. 도킹 site 는 자식 컨트롤이 도킹에서 벗어나고자 할 때 여기에 반응하여 여러가지 설정을 하거나, 아예 도킹이 해제되지 못하도록 할 수도 있는데, 이럴 때에는 OnUnDock 이벤트 핸들러를 사용한다.

property OnUnDock: TUnDockEvent;

TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean) of object;

Client 파라미터는 도킹을 해제하려는 자식 컨트롤을 나타내며, Allow 파라미터는 도킹 해제를 허용할 것인지 여부를 결정한다. OnUnDock 이벤트 핸들러를 구현할 때에는 현재 도킹되어 있는 다른 자식 컨트롤이 어떤 것이 있는지 아는 것이 유용하다. 이러한 정보는 읽기 전용인 DockClients 프로퍼티를 통해 얻을 수 있는데, 이 프로퍼티는 인덱스를 가진 TControl 의 배열이다. 도킹되어 있는 클라이언트 컨트롤의 수는 DockClientCount 프로퍼티를 통해 얻을 수 있다.

#### 5. 자식 컨트롤이 드래그-도크 작업에 적절하게 반응하는 방법

도킹이 가능한 자식 컨트롤은 drag-and-dock 작업을 수행하는 동안 2 개의 이벤트를 발생시킨다. 각각의 이벤트의 선언부와 역할에 대해서 잠시 알아보도록 하자.

OnStartDock 이벤트의 선언부는 다음과 같다.

property OnStartDock: TStartDockEvent;

TStartDockEvent = procedure(Sender: TObject; var DragObject: TDragDockObject) of object;

OnStartDock 이벤트는 drag-and-drop 작업의 OnStartDrag 이벤트와 유사한 역할을 한다. OnStartDrag 이벤트와 마찬가지로, 도킹 가능한 자식 컨트롤이 사용자 정의 드래그 객체

(drag object)를 생성하는 것을 허용할 것인지를 결정한다.

OnEndDock 이벤트의 선언부는 다음과 같다.

```
property OnEndDock: TEndDragEvent;
```

```
TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer) of object;
```

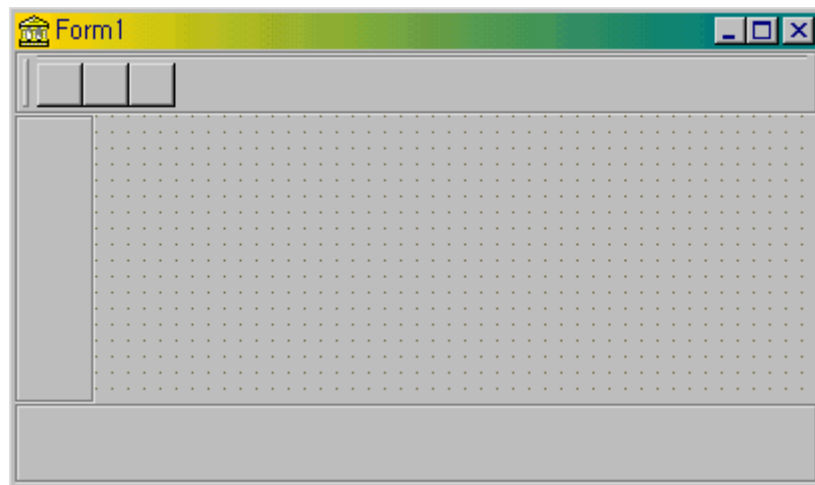
OnEndDock 이벤트는 드래그-드롭 작업의 OnEndDrag 이벤트와 비슷한 역할을 한다.

#### ● 드래그-도크를 구현한 간단한 예제

그러면, 지금까지 설명한 내용을 바탕으로 간단하게 드래그-도크를 구현한 예제를 살펴 보자. 이를 위해서는 아무런 코딩 작업도 필요 없고 단지 컴포넌트를 올려 놓고, 프로퍼티를 변경하는 것으로 모든 것을 마칠 수 있다.

최근에 경향으로 보아 쿨바와 툴바를 이용한 인터페이스가 거의 대세를 장악하고 있는 듯하다. 그러므로, 여기에서도 쿨바와 툴바를 이용하도록 하겠다.

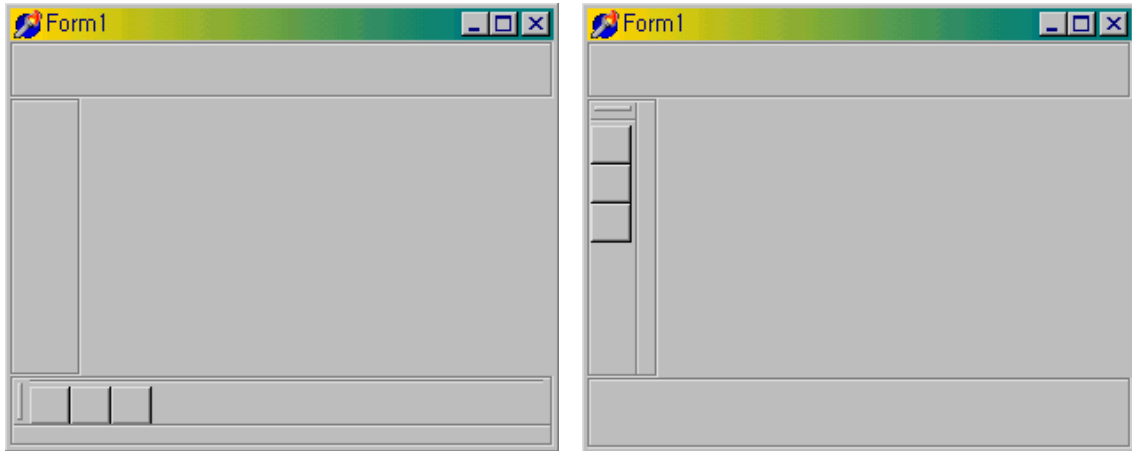
먼저 폼에 TCoolbar 컴포넌트 3 개를 올려 놓고, 각각의 Align 프로퍼티를 alTop, alBottom, alLeft 로 설정한다. 그리고, TToolbar 컴포넌트 하나를 alTop 으로 설정한 쿨바 컴포넌트 위에 다음과 같이 올려 놓도록 하자.



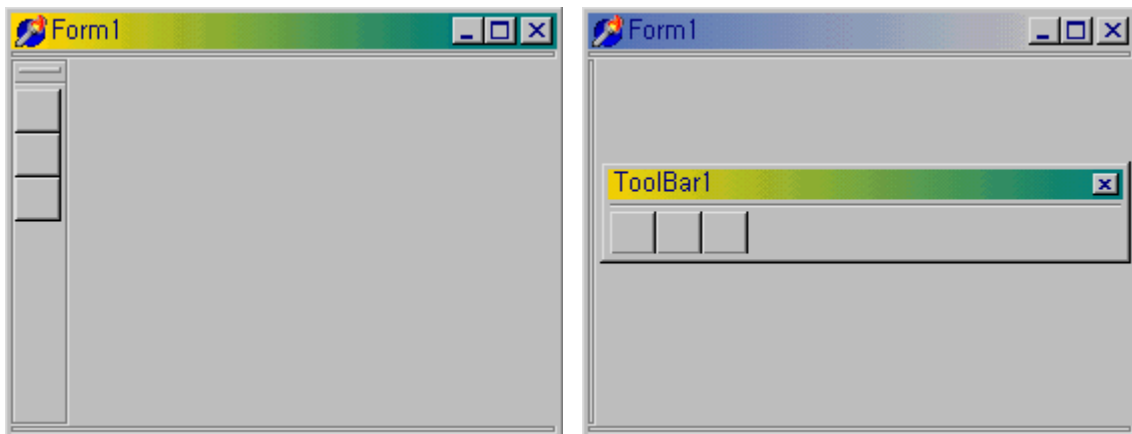
이제 이들의 프로퍼티를 각각 설정할 차례이다. 3 개의 쿨바 컴포넌트의 DockSite 프로퍼티를 모두 True 로 설정한다. 그리고, 툴바 컴포넌트 위에서 오른쪽 버튼을 클릭하고 New Button 메뉴로 툴버튼을 3 개 추가하면 앞의 그림과 같은 모양이 될 것이다.

이제 툴바 컴포넌트의 DragKind 프로퍼티를 dkDock, DragMode 프로퍼티를 dmAutomatic 으로 설정한 뒤에 컴파일하고 이를 실행하도록 하자.

그러면, 다음과 같이 툴바를 마음대로 옮겨다닐 수 있을 것이다.



그런데, 아마도 쿼바의 경계 부분이 눈에 거슬릴 것이다. 이를 제거하려면 다시 프로젝트 파일로 돌아와서 3 개의 쿼바 컴포넌트의 AutoSize 프로퍼티를 True 로 설정하도록 한다. 그리고, 다시 컴파일하고 실행을 하면 다음 그림과 같이 쉽게 드래그-도크가 구현된 윈도우를 볼 수 있을 것이다.



## 액션 리스트(Action List) 클래스의 활용

액션 리스트는 버튼과 메뉴 등의 사용자 명령어를 중앙 집중식으로 관리할 수 있도록 해준다. 액션이란 목적 객체(target object)에서 동작하는 사용자의 명령을 말한다. 액션은 액션 링크를 통해 클라이언트 컨트롤과 연결된다.

- VCL 액션 객체

액션은 클라이언트에 의해 유발되며, 보통 메뉴 아이템이나 버튼을 클릭했을 때에 시작된다. 여기에 관여하는 클래스는 기본적인 윈도우 에디트와 메뉴 명령을 구현한 TAction 클래스



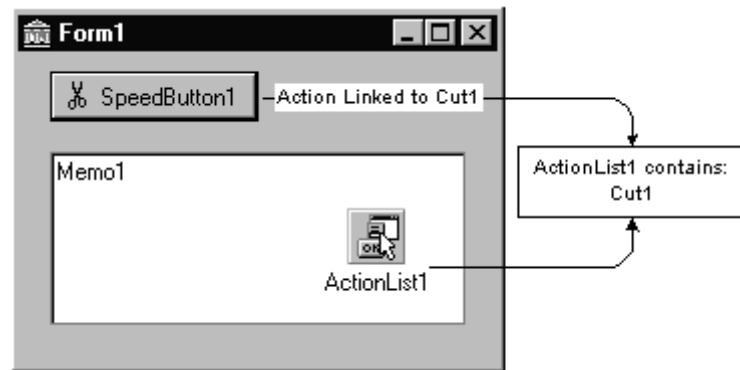
에서 상속받은 클래스들로 StdActns 유닛에 포함되어 있다.

액션의 클라이언트에는 메뉴 아이템과 여러가지 버튼(TToolButton, TSpeedButton, TMenuItem, TButton, TCheckBox, TRadioButton 등)이 있으며, 액션은 클라이언트의 해당 명령에 의해 시작된다. 보통 클라이언트의 Click 과 액션의 Execute 가 연결된다.

TActionList 컴포넌트는 TAction 클래스의 리스트를 관리하는 컴포넌트이다. 또한, TActionLink 객체는 각각의 액션과 클라이언트의 접속을 유지하는 역할을 한다.

액션 목표(action target)는 보통 메모나 각종 데이터 컨트롤 등의 컨트롤이다. 예를 들어, DBActns 유닛에는 데이터 세트 컨트롤에 대한 액션을 구현한 클래스들을 포함하고 있다. 컴포넌트 제작자는 컨트롤을 제작하거나 사용할 때 필요한 액션을 직접 제작할 수 있으며, 이 유닛을 패키지에 포함하면 보다 모듈화된 어플리케이션을 제작할 수 있다.

다음의 그림은 이들 객체들의 관계를 나타낸 것이다.



여기에서 Cut1 은 액션이며, ActionList1 에는 Cut1 을 포함하고 있다. 그리고, 여기에서 SpeedButton1 이 Cut1 액션의 클라이언트가 되며, Memo1 이 목표(target)가 된다. 또한, 이 그림에서의 액션 링크는 SpeedButton1 과 Cut1 을 연관시키고 있는 것으로, ActionList 에 포함되어 있다.

## ● 액션의 사용

액션 리스트 컴포넌트는 컴포넌트 팔레트의 standard 페이지에 위치해 있으며, 폼이나 데이터 모듈에 추가해서 사용한다. 이 컴포넌트를 더블 클릭하면 액션 리스트 에디터가 나타나며, 여기에서 액션을 추가, 삭제, 재배열 등의 작업을 할 수 있게 된다.

오브젝트 인스펙터에서 각 액션의 프로퍼티를 설정할 수 있다. 간단하게 이해하자면 Name 프로퍼티는 액션을 가리키며 Caption, Checked, Enabled 등의 다른 프로퍼티와 이벤트는 클라이언트 컨트롤의 것이다.

### 1. 코드의 중앙집중화 :

텔파이 4에서는 TToolButton, TSpeedButton, TMenuItem, TButton 등의 컴포넌트의 프로퍼티에 Action 프로퍼티가 추가되었다. 이 프로퍼티를 액션 리스트에 열거된 하나의 Action 으로 설정하면 액션 객체와 연결이 된다. Name, Tag 프로퍼티를 제외한 모든 프로퍼티와 이벤트는 액션 객체와 연관되어 사용할 수 있게 되는데, 예를 들어 버튼이나 메뉴 아이템을 disable 시키는 코드를 중복하여 작성하지 않고, 이를 액션 객체에서 중앙집중식으로 관리할 수 있다. 즉, 특정 action 이 disabled 되면 해당되는 버튼이나 메뉴 아이템이 모두 disabled 된다.

또한, 버튼이나 메뉴 아이템을 클릭하면 컨트롤의 OnClick 이벤트는 자동으로 연결된 액션 아이템의 Execute 메소드를 호출하게 된다.

## 2. 프로퍼티의 링크

클라이언트의 액션 링크는 프로퍼티를 통해서 액션과 연결을 하게 된다. 이렇게 연결된 프로퍼티는 액션이 변경되면, 액션 링크는 클라이언트의 프로퍼티를 업데이트하게 된다. 자세한 내용은 각각의 액션 링크 클래스에 대한 도움말을 참고하기 바란다.

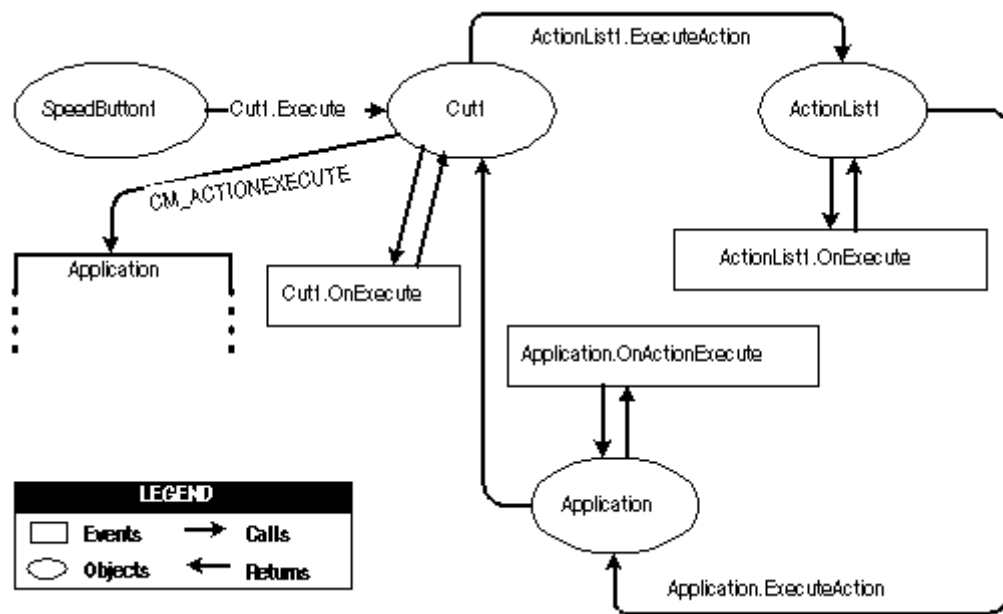
## 3. 액션의 실행

클라이언트 컴포넌트나 컨트롤이 클릭되면 연결된 액션의 OnExecute 이벤트가 발생한다. 예를 들어, 다음의 코드는 액션이 실행될 때 툴바의 visibility 를 토글한다.

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
    ToolBar1.Visible := not ToolBar1.Visible;
end;
```

툴 버튼이나 메뉴 아이템을 사용하는 경우, 해당되는 메뉴 컴포넌트나 툴바의 Images 프로퍼티를 액션 리스트의 Images 프로퍼티로 설정해야 한다.

다음 그림은 Cut1 이라는 액션의 실행 사이클을 그려본 것이다. 이 그림에서 액션 클라이언트는 Speedbutton1 이며, Cut1 과 액션 링크를 통해 연결된다. Speedbutton1 의 Action 프로퍼티는 Cut1 으로 설정된다. 이렇게 되면, Speedbutton1 의 Click 메소드는 Cut1 의 Execute 메소드를 실행하게 된다.



Speedbutton1 을 클릭하면 다음과 같은 실행 사이클을 시작하게 된다.

- Speedbutton1 의 Click 메소드는 Cut1.Execute 를 실행한다.
- Cut1 액션은 액션 리스트(ActionList1)에 Execute 메소드의 처리를 맡기게 된다. 액션 리스트는 ExecuteAction 메소드를 호출하게 된다.
- ActionList1 은 ExecuteAction 메소드에 대한 OnExecute 이벤트 핸들러를 호출하게 된다. 이때 액션 리스트의 ExecuteAction 메소드는 액션 리스트에 담겨 있는 모든 액션에 적용된다. 이 이벤트 핸들러에는 Handled 라는 파라미터를 가지고 있으며, 디폴트로 False 를 반환한다. 이벤트 핸들러가 사용되고, 이벤트를 처리한 경우에는 다음과 같이 이 파라미터에 True 를 반환해야 한다.

```

procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    {ActionList1 에 담긴 다른 액션의 실행을 막는다.}
    Handled := True;
end;

```

만약 이 시점에서 처리가 이루어지지 않으면, 액션 리스트 이벤트 핸들러는 계속해서 처리를 하게 된다.

- Cut1 액션은 어플리케이션 객체의 ExecuteAction 메소드를 실행하게 되고, 이로 인해 OnActionExecute 이벤트 핸들러가 실행된다. 어플리케이션 객체의 ExecuteAction 메

소드는 어플리케이션에 있는 모든 액션에 적용된다. 실행되는 순서는 액션 리스트의 ExecuteAction 메소드와 동일하다. 이 이벤트 핸들러의 Handled 파라미터 역시 False 를 디폴트로 가지며, 처리 순서는 이 시점에서 끝난다. 마찬가지로 다음과 같이 True 를 반환해야 한다.

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    {어플리케이션에 있는 다른 액션의 실행을 막는다.}
    Handled := True;
end;
```

만약 어플리케이션 이벤트 핸들러에서 처리되지 않으면, Cut1 은 CM\_ACTIONEXECUTE 메시지를 어플리케이션의 WndProc 에 전송한다. 이렇게 되면, 어플리케이션은 액션을 실행하기 위해 목표(target) 객체를 찾아서 실행하게 된다.

#### 4. 액션의 업데이트:

어플리케이션이 idle 상태에 있을 경우 액션이 일어날 때마다 OnUpdate 이벤트가 발생한다. 이를 이용하면 어플리케이션에서 중앙집중화된 코드를 enable, disable 또는 check, uncheck 하는 작업을 할 수 있다. 다음의 코드를 살펴보자

```
procedure TForm1.Action1Update(Sender: TObject);
begin
    {툴바가 현재 보이는지 여부를 나타낸다.}
    (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

참고로 시간이 많이 걸리는 코드는 OnUpdate 이벤트 핸들러에 추가하지 않는 것이 좋다. 자주 실행되기 때문에, 어플리케이션 전체 성능에 심각한 저하를 가져오게 된다.

#### ● 액션을 이용한 컴포넌트 제작:

델파이 4 에서는 메뉴나 버튼 컨트롤에서 프로퍼티 값을 설정하여 액션 객체에 의해 상속된 프로퍼티를 선별적으로 오버라이드할 수 있다. 이렇게 하면 액션에서의 프로퍼티는 바뀌지 않고, 버튼과 메뉴 컨트롤에만 영향을 미치게 할 수 있다.

컴포넌트 제작자는 어떤 type 의 컨트롤에든 액션 기능을 추가할 수 있게 되었다. TControl 의 protected 섹션에 선언된 Action 프로퍼티를 public 또는 published 로 섹션으로 이동하면 된다. 액션을 이용한 컴포넌트 제작에 대한 내용은 4 부에서 자세히 다루게 될 것이다.

## ● 액션의 등록

템플릿 4 에서는 컴포넌트를 이용하지 않고, 전역 루틴을 이용해서 액션을 등록하거나 해제하는 것도 가능하다. 이때 사용되는 루틴은 다음과 같다.

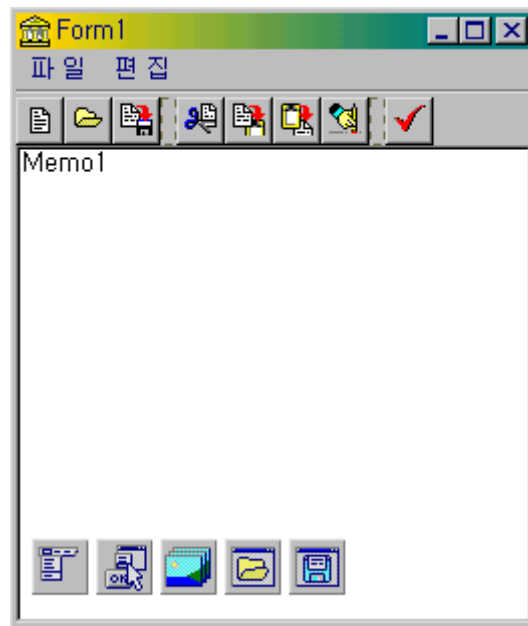
```
procedure RegisterActions(const CategoryName: string;
    const AClasses: array of TBasicActionClass);
procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

이들 프로시저는 ActnList.pas 유닛에 선언되어 있다. 이들을 이용하여 표준 액션을 등록하는 예제 코드는 다음과 같다.

```
{ Standard action registration }
RegisterActions('', [TAction]);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste]);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
    TWindowTileVertical, TWindowMinimizeAll, TWindowArrange]);
```

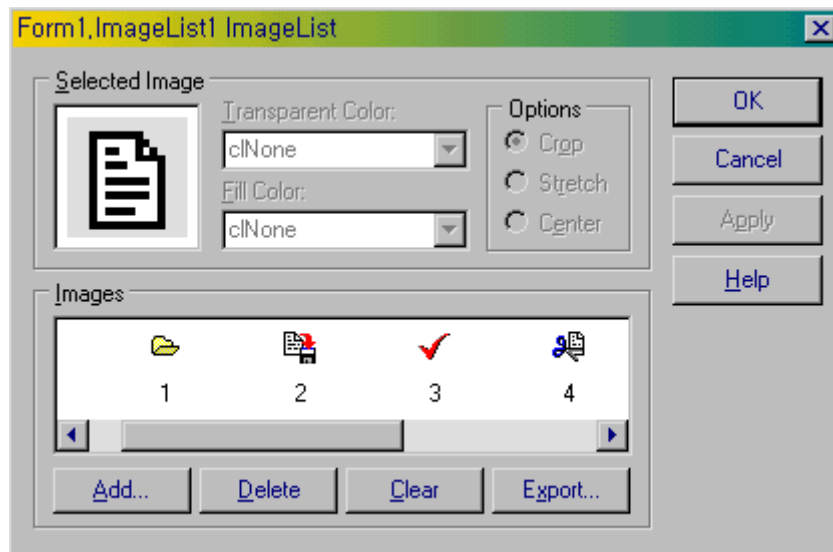
## ● 간단한 예제 어플리케이션의 제작

액션 리스트에 대한 예제로 아주 간단한 에디터를 하나 만들어 보자. 폼위에 TToolbar 컴포넌트를 하나 얹고, 툴바에서 오른쪽 버튼을 클릭하여 툴바 버튼을 8 개, 분리자 (Separator)를 2 개 추가한다. 그리고, 메모 컴포넌트를 하나 추가하고 Align 프로퍼티를 alClient 로 설정한다. 여기에 다음 그림과 같이 파일을 읽고, 쓸 수 있도록 TOpenDialog, TSaveDialog 컴포넌트를 하나씩 얹고, 메인 메뉴를 만들기 위한 TMainMenu 컴포넌트와 TActionList, TImageList 컴포넌트를 하나씩 추가하자.

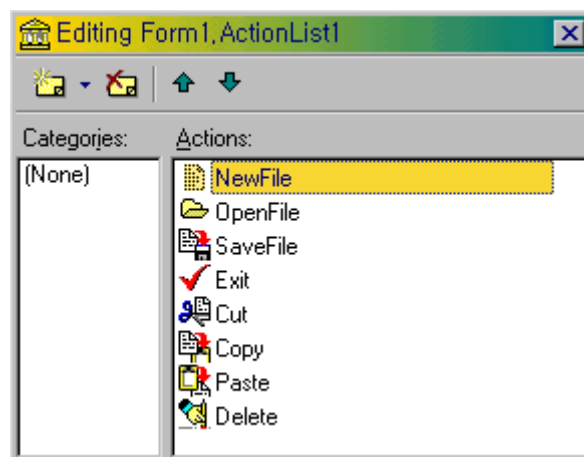


물론, 독자 들의 폼의 모양은 이와는 다를 것이다. 이것은, 아직 비트맵도 설정하지 않았고, 메뉴 아이템도 결정하지 않았기 때문이므로 계속 따라가다 보면 결국에는 이런 모습이 될 것이다.

그러면, 먼저 비트맵 이미지를 결정하도록 한다. 텔파이에서 제공하는 비트맵이나 아이콘은 보통 Common Files\Borland Shared\Images 디렉토리에서 찾아볼 수 있다. 여기서 Buttons 디렉토리에서 앞의 그림의 8 개 비트맵을 찾아서 ImageList 에 추가하자. 이때 주의할 것은 버튼의 비트맵은 보통 2 개의 이미지로 들어가게 된다는 것이다. 뒤쪽의 이미지는 마스크로 쓰이는 것으로, 보통 버튼이 클릭되는 효과를 보이게 하기 위해 사용된다. 여기서는 마스크를 사용하지 않을 것이므로, 버튼의 비트맵을 추가한 후 마스크 비트맵은 삭제하도록 하자. 8 개의 비트맵을 모두 추가하고 나면 ImageList 의 형태가 다음과 같으면 된다.



이제는 액션 리스트를 정의할 차례이다. 먼저, ActionList1 의 Images 프로퍼티를 ImageList1 으로 설정하여 앞에서 정의한 비트맵을 사용하도록 한다. 그리고, ActionList1 컴포넌트를 더블 클릭하면 액션 에디터가 뜨는데, 여기에 다음과 같이 액션을 8 개 정의한다.



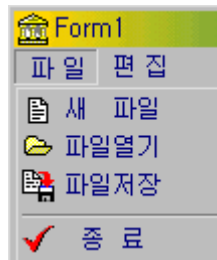
여기서 오른쪽 버튼을 클릭하면, 이와 같이 새로운 액션을 추가할 수 있게 된다. 그리고, 각각의 액션을 클릭하고 오브젝트 인스펙터에서 프로퍼티를 설정할 수 있는데, 다음과 같이 설정하면 앞의 그림과 같이 된다.

Name	Caption	ImageIndex	Name	Caption	ImageIndex
NewFile	새 파일	0	Cut	오려두기	4
OpenFile	파일열기	1	Copy	복사하기	5
SaveFile	파일저장	2	Paste	붙여넣기	6

Exit	종 료	3	Delete	삭제하기	7
------	-----	---	--------	------	---

이제 액션을 모두 정의했으므로, 메뉴와 툴바를 액션에 맞춰넣기만 하면 된다.

먼저 MainMenu1 컴포넌트의 Images 프로퍼티를 ImageList1 으로 설정한다. 그리고, MainMenu1 컴포넌트를 더블 클릭하여 메뉴를 편집한다. 여기서 특별히 할 것이 없다. 개발자가 할 일은 먼저 ‘파 일’과 ‘편 집’이라는 가장 큰 메뉴를 하나씩만 만들고 그 다음 메뉴 아이템 부터는 Action 프로퍼티를 ActionList1 에 정의한 NewFile 에서 부터 Delete 까지의 8 개의 액션을 하나씩 설정하기만 하면 된다. 여기서 ‘파 일’ 섹션에 NewFile, OpenFile, SaveFile, Exit 를 Action 프로퍼티로 설정한다. SaveFile 과 Exit 사이에는 메뉴 아이템의 Caption 을 ‘-’로 설정하여 구분선을 하나 삽입한다. 그리고, ‘편 집’ 섹션에 Cut, Copy, Paste, Delete 를 Action 프로퍼티로 설정한다. 이렇게만 하면 Caption 과 메뉴 아이템의 비트맵은 다음 그림과 같이 자동으로 설정되었을 것이다.



툴바 역시 마찬가지이다. Toolbar1 의 Images 프로퍼티를 ImageList1 으로 설정하고, 각각의 툴바 버튼의 Action 프로퍼티를 NewFile, OpenFile, SaveFile, Exit, Cut, Copy, Paste, Delete 로 각각 설정하면 그걸로 끝이다. 여기까지 그대로 설정했다면 앞에서 보았던 폼의 모양과 동일하게 변했을 것이다.

이제 프로퍼티의 설정이 모두 끝났다. 그러면, 실제로 돌아갈 수 있도록 이벤트 핸들러 들을 적당하게 작성해 보자. 먼저, 폼이 시작할 때 Memo1 의 내용을 깨끗이 지워주기 위해서 Form1 의 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Memo1.Lines.Clear;
end;
```

이제는 8 개의 액션에 대한 이벤트 핸들러를 작성해주면 된다. 오브젝트 인스펙터에서 NewFile 을 선택하고, 이벤트 페이지 탭에서 OnExecute 이벤트의 이벤트 핸들러를 다음과 같이 작성한다.



```
procedure TForm1.NewFileExecute(Sender: TObject);
begin
    Memo1.Lines.Clear;
end;
```

그리고, 파일을 읽고 쓰는 OpenFile, SaveFile 액션의 OnExecute 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.OpenFileExecute(Sender: TObject);
begin
    OpenFileDialog1.Filter := 'Text files (*.txt)|*.TXT';
    if OpenFileDialog1.Execute then
    begin
        Memo1.Lines.LoadFromFile(OpenFileDialog1.FileName);
    end;
end;
```

```
procedure TForm1.SaveFileExecute(Sender: TObject);
begin
    SaveDialog1.Filter := 'Text files (*.txt)|*.TXT';
    if SaveDialog1.Execute then
    begin
        Memo1.Lines.SaveToFile(SaveDialog1.FileName);
    end;
end;
```

파일을 읽고 쓰기 위해서 OpenFileDialog1, SaveDialog1 대화상자를 이용하며, Filter 프로퍼티를 기본적으로 텍스트 파일로 설정하였다.

Exit 액션에 대한 이벤트 핸들러는 간단히 다음과 같이 작성하면 된다.

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

클립보드 작업을 위한 Cut, Copy, Paste 액션과 선택된 문자열을 삭제하기 위한 Delete 액

선의 OnExecute 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.CutExecute(Sender: TObject);
begin
    Memo1.CutToClipboard;
end;

procedure TForm1.CopyExecute(Sender: TObject);
begin
    Memo1.CopyToClipboard;
end;

procedure TForm1.PasteExecute(Sender: TObject);
begin
    Memo1.PasteFromClipboard;
end;

procedure TForm1.DeleteExecute(Sender: TObject);
begin
    Memo1.ClearSelection;
end;
```

다들 간단한 코드이므로 자세한 설명은 생략하겠다. 이제 모든 이벤트 핸들러의 작성이 끝난 것이다. 컴파일을 하고, 프로그램을 실행하면 간단한 에디터로 사용할 수 있을 것이다. 이렇게 액션 리스트를 사용하면, 과거에 스피드 버튼과 메뉴를 통해 중복해서 작성하던 여러 코드를 간단하게 중앙 집중적으로 관리할 수 있다. 그리고, 어플리케이션의 실행 블록을 체계적으로 나눌 수 있기 때문에 메뉴 아이템과 버튼이 많은 어플리케이션의 경우 코드의 직관성과 재사용성도 월등히 향상시킬 수 있다.

아무리 좋은 연장도 제대로 쓰지 않으면 아무 소용이 없는 법이므로, 어플리케이션의 크기가 조금 복잡하다 싶으면 꼭 액션 리스트를 쓸 것을 권하는 바이다. 아마도 훨씬 관리하기도 편해지고, 디버깅도 쉬워진 것을 느낄 수 있을 것이다.

## 정 리 (Summary)

이번 장에서는 델파이의 VCL 계층 구조와 핵심적인 클래스에 대해서 소개하고, 델파이가 제공하는 컴포넌트 들에 대해 간략하게나마 어떤 것들이 있는지 알아보았다.

그리고, 비교적 많이 쓰일 수 있는 공통 주제로 드래그-드롭, 드래그-도크, 액션 리스트의 활용 방법을 자세한 설명과 예제를 활용해서 설명하였다.

여기서 설명한 내용들은 매우 자주 쓰이면서도 중요한 내용이므로, 여러 차례 연습을 해서 반드시 몸에 체득하기 바란다.

다음 장에서는 어플리케이션에 파일 입출력과 출력 기능을 추가하는 방법에 대해서 알아보도록 한다.

# 파일 입출력과 인쇄 기능의 추가

## (File I/O and Printing)

파일의 입출력과 프린터를 통한 출력은 어플리케이션을 개발할 때 중요하게 여기고 개발해야 하는 기능이다. 이번 장에서는 파일을 가지고 작업하는 방법과, 인쇄를 할 때 델파이에 사용할 수 있는 여러가지 기법에 대해서 알아볼 것이다.

### 파일의 종류와 입출력 방법

파일 입출력 루틴을 통해 사용할 수 있는 파일 종류에는 3 가지가 있다. 전통적인 스타일의 파스칼 파일과 윈도우 파일 핸들, 파일 스트림 객체가 그것이다.

전통적인 스타일의 파스칼 파일은 F: Text 또는 F: File 의 형태로 사용되는 파일 변수를 사용하게 된다. 이들 파일에는 크게 나누어 type, text, untyped 의 3 가지로 종류를 나누어 볼 수 있으며, AssignPrn, WriteLn 을 비롯한 수많은 파일 처리 루틴이 이들을 지원한다.

윈도우 파일 핸들은 실제 운영체제에서 사용되는 직접적인 형태이다. 파스칼 언어 자체는 기본적으로 멀티 플랫폼을 대상으로 하기 때문에, 궁극적으로 보면 오브젝트 파스칼의 파일 처리 루틴 역시 윈도우 파일 핸들을 다루는 wrapper 에 불과하다. 결국에는 윈도우 API 를 사용하게 되는데, 예를 들어, FileRead 함수는 윈도우 API 의 ReadFile 함수를 호출한다. 그런데, 오브젝트 파스칼의 루틴을 사용하지 않고, 직접 윈도우 API 를 사용해서 파일 처리를 하려 하면 반드시 파일의 핸들을 사용해야 한다.

파일 스트림은 VCL 클래스인 TFileStream 에 기초한 객체로, 파일 입출력에 높은 수준에서 접근할 수 있게 해준다. TFileStream 의 Handle 프로퍼티는 윈도우의 파일 핸들과 동일한 것이다. 파일 스트림을 사용하면 보다 고급스럽고, 세련된 프로그래밍을 할 수 있기 때문에 많이 사용되는 클래스이다.

### 고정길이 파일 (Fixed Length Files) 다루기

델파이의 비정형(untyped) 파일은 read, write, read write 모드로 열 수 있으며 파일의 레코드 위치를 자유롭게 할 수 있다. 또한, 한 번에 하나 이상의 고정길이 레코드를 읽고 쓸 수 있으며, 그 수행속도가 뛰어나다.

이러한 비정형 파일을 이용해서 파일을 복사하는 프로시저를 만들 수 있는데, 이 프로시저를 예로 들어서 비정형 파일의 사용법을 알아보도록 하자.

```
procedure CopyFile(SrcFileName, DestFileName: String);
```

```

var
    Buffer: array[1..8192] of Char;
    SrcFile, DestFile: File;
    ReadCount, WriteCount: Integer;
begin
    AssignFile(SrcFile, SrcFileName);
    Reset(SrcFile, 1);
    AssignFile(DestFile, DestFileName);
    Rewrite(DestFile, 1);
    repeat
        BlockRead(SrcFile, Buffer, SizeOf(Buffer), ReadCount);
        BlockWrite(DestFile, Buffer, ReadCount, WriteCount);
    until (ReadCount = 0) or (WriteCount < ReadCount);
    CloseFile(SrcFile);
    CloseFile(DestFile);
end;

```

일단은 AssignFile 프로시저를 이용해서 파일을 정의하고, 이를 읽어오는 파일은 Reset 으로 쓰는 쪽은 Rewrite 를 사용한다. Reset 프로시저는 기존에 파일이 존재할 때 이 파일의 제일 처음에 포인터를 위치시키는 프로시저이고, Rewrite 는 새로 파일을 생성하거나, 기존에 같은 이름의 파일이 있다면 이를 삭제하고 새로 파일을 만드는 프로시저 이다.

Reset, Rewrite 프로시저는 모두 두 개의 파라미터를 가질 수 있다. 첫번째 파라미터에는 대상이 되는 파일을 넘겨주게 되어 있고, 두번째 파라미터에는 레코드의 크기를 지정한다. 이때 두번째 파라미터는 생략할 수 있는데 생략할 경우 128 바이트가 디폴트로 지정된다. 파일을 복사하는 경우에는 1 바이트도 읽고, 쓸 수 있어야 하므로, 두번째 파라미터를 모두 1 로 지정한다.

실제로 복사가 이루어지는 부분은 다음 코드이다.

```

repeat
    BlockRead(SrcFile, Buffer, SizeOf(Buffer), ReadCount);
    BlockWrite(DestFile, Buffer, ReadCount, WriteCount);
until (ReadCount = 0) or (WriteCount < ReadCount);

```

비정형 파일을 읽고, 쓸 때 사용하는 프로시저들이 BlockRead 와 BlockWrite 이다. BlockRead 는 4 개의 파라미터를 가지는데 각각 파일 변수, 데이터에 대한 버퍼로 사용할 변수나 구조체의 이름, 그리고 읽을 레코드의 수를 첫번째에서 세번째 파라미터로 지정한다.

이때 읽을 레코드의 수에 레코드 크기를 곱한 값은 버퍼의 크기와 작거나 같아야 한다. 네번째 파라미터는 실제로 읽은 레코드의 수가 담겨지게 되는 변수이다. 보통 이 값은 세번째 파라미터의 값과 동일하게 되지만 파일의 끝부분에 도달하면 이 값이 더 작게 된다. 앞의 경우에 세번째 파라미터를 SizeOf(Buffer)로 지정했는데, SizeOf 함수는 변수의 바이트 크기를 돌려주는 함수이다. 이 경우에는 레코드의 크기가 1 바이트이므로 버퍼 배열의 크기와 같게 된다. BlockWrite 프로시저도 마지막 파라미터에 실제로 기록한 레코드의 수가 넘어온다는 것을 제외하면 BlockRead 와 같은 파라미터를 사용한다. ReadCount 변수의 값이 0 이라는 의미는 파일의 끝에 도달했다는 뜻이며, WriteCount 변수가 ReadCount 변수보다 작다는 의미는 디스크가 꽉 차는 등의 에러로 인해 읽은 레코드를 모두 쓸 수 없었다는 의미가 되므로 여기까지 루프를 돌리면 파일의 복사가 종료된다. 파일을 복사하는 루틴에 대한 설명으로 비정형 파일의 사용법은 거의 익숙해졌을 것으로 생각된다. 그러면, 단순한 파일의 복사가 아니라 레코드를 비정형 파일 기법을 이용해서 관리하는 방법에 대해서 알아보도록 하자. 이렇게 비정형 파일 기법을 이용하면 대단히 빠른 속도로 레코드를 읽고 쓸 수 있다. 참고로 레코드를 파일로 관리하는 방법에는 레코드 파일(File of Record)을 이용하는 방법이 있는데, 여기에 대해서는 이장의 후반부에서 더 자세히 다룰 것이다. 먼저 레코드를 다음과 같이 선언하도록 한다.

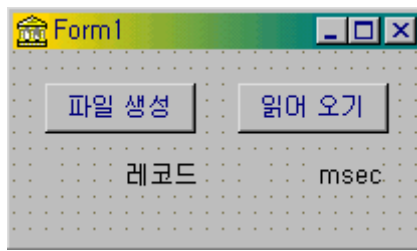
type

```
TPerson = record
    Name: array[1..10] of Char;
    ID: array[1..8] of Char;
    Password: array[1..8] of Char;
end;
```

이렇게 정의한 TPerson 레코드의 크기는 28 바이트가 된다.

그러면, 이 레코드 형의 데이터 10 만개에 특정 값을 지정해서 저장했다가, 이를 읽어오고 읽은 레코드의 수를 TLabel 컴포넌트를 이용해서 나타내는 예제를 제작해 보자.

먼저 다음과 같이 폼에 버튼 2 개와 TLabel 컴포넌트 4 개를 올려 놓고, 버튼의 캡션을 각각 ‘파일 생성’과 ‘읽어 오기’, 그리고 Label2 와 Label4 의 캡션을 ‘레코드’, ‘msec’로 설정한다.



여기서 Label1 에는 읽어온 레코드의 수를, 그리고 Label3 에는 Win32 의 GetTickCount 함수를 이용해서 10 만 레코드를 읽어오는데 걸린 시간을 백만 분의 1 초 단위로 나타낸다. 여기서 10 만 레코드를 사용한 이유는 필자가 테스트 해 본 결과 1 만 레코드 읽는데에는 백만 분의 1 초 밖에 걸리지 않아서, 의미가 없기에 그래도 조금은 큰 10 만 레코드를 읽고, 쓰는데 걸리는 시간을 사용하기 위한 것이다. 10 만 레코드를 저장하면 파일의 크기는 약 2.5MB 정도가 된다.

그리고, 비정형 파일을 다룰 때에는 레코드의 멤버가 파스칼 문자열로 존재하면, 이를 고정된 길이의 문자 배열로 바꾸어서 문자가 없는 부분은 공백으로 채우는 프로시저를 하나 만들어 주어야 한다. 이러한 역할을 하는 AssignData 라는 프로시저를 interface 섹션에 선언하고 다음과 같이 구현한다.

```
procedure AssignData(const Data: String; var ArrData: array of Char);
var
    i: Word;
begin
    for i := 0 to High(ArrData) do ArrData[i] := ' ';
    for i := 1 to Length(Data) do ArrData[i - 1] := Data[i];
end;
```

이 프로시저는 일단 ArrData 파라미터를 열린 배열(open array)로 선언하고, 구현부분에서 High 함수를 이용해서 크기를 정할 수 있도록 하는 것이 중요하다. 그리고, 일단은 문자 배열의 내용을 모두 공백문자(' ')로 채운 뒤, 파스칼 문자열의 길이 만큼 문자 배열의 앞에서부터 채워 나간다.

이제, 파일을 생성하는 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
const
    MaxRec = 100;
var
    Buffer: array[1..MaxRec] of TPerson;
```

```

PersonFile: File;
i, Count: Integer;
begin
    AssignFile(PersonFile, 'Person.dat');
    Rewrite(PersonFile, SizeOf(TPerson));
    for i := 1 to MaxRec do
        begin
            with Buffer[i] do
                begin
                    AssignData('정지훈', Name);
                    AssignData('ttolttol', ID);
                    AssignData('pswrd1', Password);
                end;
            end;
        for i := 1 to 1000 do
            BlockWrite(PersonFile, Buffer, MaxRec, Count);
        CloseFile(PersonFile);
    end;

```

일단 레코드의 수를 100 개로 설정하고, TPerson 레코드 100 개의 레코드 배열을 버퍼로 선언한다. 앞에서 설명한 파일 복사 루틴과 비교해 보면 쉽게 이해할 수 있을 것이다.

Rewrite 프로시저를 사용해서 이미 존재하는 파일이 있으면 이를 삭제하고 새로 파일을 생성하도록 하며, 두번째 파라미터에는 레코드의 크기를 SizeOf(TPerson)을 이용해서 설정해 준다.

바로 다음의 for 루프에서는 버퍼에 100 개까지의 레코드를 담을 수 있으므로, 1 부터 100까지의 레코드 배열 버퍼에 문자열을 고정 길이의 문자 배열로 변환해서 Name, ID, Password 의 각 멤버에 저장한다. 이런 문자 배열로의 변환 작업에 앞에서 설명한 AssignData 프로시저를 이용한다.

그리고, 그 다음의 for 루프에서는 실제로 데이터를 쓰는 작업을 하는데 1000 번의 루프를 돌면서 버퍼의 레코드를 쓰게 되므로 총 10 만 레코드를 기록하게 된다.

그러면, 이번에는 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button2Click(Sender: TObject);
const
    MaxRec = 500;
type

```



```

    TBuffer = array[1..MaxRec] of TPerson;
var
    Buffer: ^TBuffer;
    PersonFile: File;
    Total, Count: Integer;
    StartTime, EndTime: LongInt;
begin
    try
        New(Buffer);
    except
        on EOutOfMemory do Exit;
    end;
    try
        AssignFile(PersonFile, 'Person.dat');
        Reset(PersonFile, SizeOf(TPerson));
        Total := 0;
        StartTime := GetTickCount;
        repeat
            Count := 0;
            BlockRead(PersonFile, Buffer^, MaxRec, Count);
            Total := Total + Count;
        until Count = 0;
        EndTime := GetTickCount;
        CloseFile(PersonFile);
        Label1.Caption := IntToStr(Total);
        Label3.Caption := IntToStr(EndTime - StartTime);
    finally
        Dispose(Buffer);
    end;
end;

```

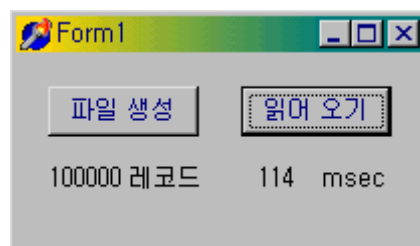
레코드를 읽어올 때에도 버퍼를 사용하게 되므로 먼저 Buffer 변수를 선언한다. 이때 먼저 type 선언문에서 500 개의 레코드를 기억할 수 있는 레코드 배열인 TBuffer 를 선언하고, Buffer 변수는 이 레코드 배열에 대한 포인터로 선언하는 것이 메모리를 효율적으로 활용할 수 있는 방안이다.

이렇게 포인터로 선언한 Buffer 변수를 활용하려면 New 프로시저를 이용해서 메모리를 할

당받아야 하는데, 이 과정에서 에러가 발생하면 예외처리를 해준다. 그리고, 마지막에는 버퍼에 할당된 메모리를 Dispose 프로시저를 이용해서 해제해야 하므로 try-finally 블록을 사용한다.

나머지 부분은 그렇게 어렵거나 새로운 부분이 없으므로 따로 설명하지 않아도 쉽게 이해할 수 있을 것이다. 다만 레코드를 읽어오는데 걸리는 시간을 측정하는 GetTickCount 함수에 대해서 간단히 설명하면 이 프로시저는 백만 분의 1 초 단위로 시스템의 시각을 읽어오는 역할을 한다. 그러므로, 특정 작업을 시작하기 전에 이 함수를 호출해서 그 값을 LongInt 형의 변수에 저장하고, 작업이 끝나고 이 함수를 호출해서 그 값을 LongInt 형의 변수에 저장한 후 그 차를 계산하면 특정 작업에 소요된 시간을 백만 분의 1 초 단위로 알아낼 수 있다.

그러면, 이 프로그램을 실행해서 28 바이트 크기의 레코드 10 만개를 읽는데 걸리는 시간을 직접 알아보자. 다음 그림의 결과는 필자의 펜티엄-150MHz, HDD 2.1GB, RAM 80MB 의 노트북에서 실행한 결과이므로 시스템에 따라서 다소간의 차이가 있을 것이다.



즉, 10 만 레코드를 읽는데 소요되는 시간이 약 1/1000 초인 셈이다. 이 정도면 비정형 과일을 이용한 레코드 관리가 얼마나 빠른 속도로 이루어지는지 쉽게 알 수 있을 것이다.

## 텍스트 파일과의 호환성

앞에서 설명한 고정길이 파일의 경우에는 레코드 사이의 delimiter 를 사용하지 않았다. 그러므로, 편집기를 이용해서 파일을 열어보면 앞의 파일의 경우 ‘정지훈 ttolttolpswrđ1 정지훈 ttolttolpswrđ1 정지훈 ttolttolpswrđ1 ...’ 과 같이 저장되어 있는 것을 발견할 수 있을 것이다. 이러한 형태의 저장 방법은 과거의 메인 프레임이나 미니 컴퓨터에서 사용하던 방식으로 현재의 PC 환경과는 맞지 않는다.

대부분의 PC 용 프로그램 들은 고정길이 파일을 만들어낼 때 레코드의 마지막에 CR/LF(Carriage Return/Line Feed) 문자를 집어 넣어서 텍스트 파일을 만들어 낸다. 그러므로, 이를 지원하기 위해서는 레코드에 다음과 같은 delimiter 멤버를 추가할 필요가 있다.

type

```

TPerson = record
  Name: array[1..10] of Char;
  ID: array[1..8] of Char;
  Password: array of [1..8] of Char;
  Delimiter: array of [1..2] of Char;
end;

```

그리고, 레코드를 쓰는 루틴에도 delimiter 를 추가하도록 수정해야 한다. 앞의 예제에서 파일을 생성하는 부분을 다음과 같은 형태로 수정하면 된다.

```

procedure TForm1.Button1Click(Sender: TObject);

```

... (중략)

```

for i := 1 to MaxRec do
begin
  with Buffer[i] do
  begin
    AssignData('정지훈', Name);
    AssignData('ttolttol', ID);
    AssignData('pswr1', Password);
    Delimiter[1] := #13;
    Delimiter[2] := #10;
  end;
end;

```

... (후략)

즉, 레코드 배열에 값을 대입할 때 CR/LF 문자를 추가해 주면 된다.

## 레코드 파일 시스템

델파이로 데이터베이스 어플리케이션을 제작하는데 단순한 레코드 형을 저장하기만 하면 되는 경우가 있다. 이럴 때에는 2MB 가 넘는 BDE 는 너무 무겁다고 할 수 있다. 이를 해결하기 위한 방편으로 BDE 를 사용하지 않는 새로운 DataSet 컴포넌트를 만들 수도 있지만, 이 역시 간단한 방법이라고는 할 수 없다. 여기에 대한 대안으로 레코드를 파일로 저장하

고 관리할 수 있는 방법이 있기에 이를 소개하고자 한다.

#### ● 레코드 파일의 선언과 특성

레코드 파일은 레코드 형의 데이터를 순차적으로 저장하는 이진 파일을 말한다. 즉, 레코드의 배열을 파일에 저장하는 것으로 생각해도 무방하다. 이러한 레코드 파일은 다음과 같은 특성을 가지고 있다.

1. 레코드 파일은 메모리 상의 배열이 동적으로 크기 변화가 되면서 파일로 저장되는 것으로 생각할 수 있다.
2. 레코드 파일은 이진 파일이기 때문에 텍스트 에디터로 쉽게 편집할 없다. 그러므로, 모든 편집 과정은 프로그램이 담당하는 것이 보통이다.
3. Read 와 Write 함수를 사용할 수 있는데, 텍스트 파일에서 사용하는 ReadLn 과 WriteLn 함수와 마찬가지로 동작 후 다음 레코드로 포인터가 넘어간다. 참고로 텍스트 파일에서 사용하는 Read, Write 함수는 한 줄을 읽고 나면 다음 줄로 넘어가지 않는다.
4. 파일 내에 있는 각각의 레코드는 0 부터 시작되는 레코드 번호를 암묵적으로 부여받는다. Seek 함수를 이용해서 파일의 특정 위치로 바로 이동할 수 있다.

#### ● 파일에 접근하기

텍스트 파일에서와 마찬가지로 AssignFile 함수를 이용해서 레코드 파일을 열게 된다. 이때 이미 존재하는 파일이라면 Reset 프로시저를, 새로 만드는 경우라면 Rewrite 프로시저를 사용한다. 레코드 파일을 닫을 때에는 CloseFile 프로시저를 사용한다.

이런 함수를 사용하려면 먼저 레코드 파일을 선언해야 하는데, 이는 레코드 데이터 형을 먼저 선언하고 이를 이용해서 다음과 같이 선언하면 된다.

type

    TMyRecord = record

        ID: Integer;

        Name: String;

    end;

var

    MyFile: File of TMyRecord;

파일을 처음 열 때에는 다음과 같이 한다.

```
AssignFile(MyFile, 'MyFile.dat');
```

```
if FileExists('MyFile.dat') then Reset(MyFile) else Rewrite(MyFile);
```

#### ● 레코드 번호와 파일 포인터

레코드 파일에서의 레코드에는 순차적으로 레코드 번호가 주어진다. 만약에 10 개의 레코드가 있으면 각각 0 에서 9 까지의 번호를 가지게 된다. 일단 레코드 파일을 열면 파일 변수는 파일 포인터를 이용해서 레코드의 위치를 가리키게 된다. 파일을 열었을 때에는 파일 포인터가 파일의 제일 처음에 위치하게 되며, 0 번 레코드를 가리킨다. 이러한 파일 포인터의 위치를 설정하는데 Seek 함수가 사용된다. 예를 들어, 30 번째 레코드로 파일 포인터를 이동하고 싶으면 다음과 같이 한다.

```
Seek(MyFile, 30);
```

이렇게 일단 파일 포인터의 위치를 옮겨 놓고 Read, Write 함수를 이용해서 그 위치의 레코드를 읽거나 쓰게 된다. 그리고, Read 또는 Write 함수에 의해서 파일 포인터는 다음으로 이동하게 된다. 즉, 2 번 레코드를 읽었으면 파일 포인터는 3 번 레코드를 가리키게 된다. 이런 식으로 읽어나가다가 파일 포인터가 마지막 레코드를 읽고 나서, 다음 레코드를 읽으려고 하면 런타임 에러가 발생한다. 이런 에러를 방지하기 위해서 파일 포인터의 위치를 알아야 하는데, 이때 사용하는 함수가 FilePos 함수이다.

FilePos 함수는 현재 레코드의 번호를 반환하는데, FileSize 함수에서 알 수 있는 레코드의 수와 비교해서 현재의 레코드 위치가 마지막에 와있는지 알아낼 수 있다. 다음의 코드를 보면 쉽게 이해할 수 있을 것이다.

```
while (FilePos(MyFile) <= (FileSize(MyFile) - 1)) do
begin
    Read(MyFile, MyRecord);
    //Do something
end;
```

#### ● 레코드 읽고 쓰기

레코드 파일에서 레코드를 읽을 때에는 Read 프로시저를 사용한다. 사용법은 앞에서도 몇 번 언급했지만 다음과 같다.

Read(MyFile, MyRecord);

앞의 문장에서 MyFile 은 파일을 담고 있는 파일 변수이며, MyRecord 는 레코드 변수이다. 레코드를 기록하는데 사용하는 Write 프로시저의 사용법 역시 비슷하다.

Write(MyFile, MyRecord);

레코드를 읽을 때에는 한 번에 여러 개의 레코드를 읽어올 수도 있다. 예를 들어 MyRecord1, MyRecord2, MyRecord3 라는 3 개의 변수에 3 개의 레코드를 다음과 같이 한꺼번에 읽어오게 할 수도 있다.

Read(MyFile, MyRecord1, MyRecord2, MyRecord3);

이들을 사용할 때 꼭 기억해야 할 것은 동작 후 파일의 포인터가 다음 레코드로 이동한다는 것이다. 별로 문제가 될 것 없는 사실인 것 같지만, 주의하지 않으면 레코드를 편집할 때 문제가 생길 수 있다. 예를 들어, 파일에서 여러 개의 레코드를 읽어서 이를 편집한다고 하자. 그리고, 이렇게 편집한 레코드를 다시 파일에 쓴다고 할 때에는 다음과 같은 형태의 프로시저를 사용 해야 한다.

procedure ChangeRecord(RecordNo: Integer; Value: String);

var

    Buffer: TMyRecord;

begin

    Seek(MyFile, RecordNo);

    Read(MyFile, Buffer);

    with Buffer do

        Name := Value;

    Seek(MyFile, RecordNo);                   //이렇게 원래 레코드 위치로 복귀해야 한다 !

    Write(MyFile, Buffer);

end;

어려운 것은 아니지만 실수하기 쉬운 부분이므로 주의해야 한다. 즉, Read 프로시저에 의해 파일 포인터가 다음 레코드로 넘어갔기 때문에 다시 한번 Seek 프로시저를 이용해서 원래의 파일 포인터로 복귀 시키는 것이다.

## ● 레코드 버퍼의 활용

이런 식으로 데이터를 레코드 파일과 사용자 인터페이스 사이에서 이동하려면, 레코드 변수를 사용해서 파일의 레코드와 실제 에디트 컨트롤 사이의 임시 버퍼로 활용할 필요가 있다. 이런 버퍼가 필요한 이유는 델파이에서 지원하는 각종 데이터 어웨어 컨트롤들과는 달리 TEdit, TComboBox 등의 표준 컨트롤 들에는 데이터 링크와 같이 파일의 레코드와 직접 연결할 수 있는 방법이 없기 때문이다. 이러한 연결을 위해서 레코드 변수를 활용한 접근 메소드가 필요하다. 다음에 소개할 예제에서는 ReadRecord 와 SetRecordValue 라는 메소드와 MyRecord (TMyRecord 데이터 형)라는 레코드 변수를 버퍼로 활용해서 이러한 문제를 해결할 것이다.

## ● 레코드의 삽입과 삭제

레코드 파일에서는 특정 위치에 레코드를 삽입하거나 삭제하는 것이 기본적으로는 불가능하다. 이를 구현하려면 레코드 버퍼를 이용해서 조작을 하고 나서, 이 데이터를 레코드 파일에 기록하는 방법을 사용해야 한다. 간단하게 구현 단계를 설명하면 다음과 같다.

1. 적절한 레코드 형으로 레코드 배열을 초기화한다.
2. 현재 파일 포인터의 위치를 얻어서, 이를 특정 변수에 저장한다.
3. 사용자 인터페이스에 있는 기록할 레코드 정보를 임시 레코드 변수에 저장한다.
4. Seek 프로시저를 사용해서 파일의 처음으로 이동한다.
5. Read 프로시저를 이용해서 삽입할 레코드 위치의 바로 앞 레코드까지 읽는다.
6. 삽입될 레코드를 배열에 저장한다.
7. 파일의 끝까지 읽어서 배열에 저장한다.
8. 레코드 배열의 내용을 파일에 저장한다.

레코드를 삭제할 때에는 삽입할 때보다는 비교적 간단하게 구현할 수 있다. 삭제 과정의 구현 단계는 다음과 같다.

1. 삭제할 파일 포인터 위치 뒤에 있는 모든 레코드를 임시 레코드 버퍼에 읽어온다.
2. Truncate 를 사용해서 삭제할 위치에서 파일의 뒤쪽 부분을 모두 절단해서 버린다.
3. 버퍼에 저장된 레코드를 파일에 저장한다.

Truncate 프로시저는 현재 파일 위치에서 뒤쪽에 있는 모든 레코드를 제거하는 역할을 한다. 그리고, 파일 포인터는 파일의 맨 뒤에 위치하게 된다.

## 간단한 주소록 프로그램 만들기

이제 앞에서 설명한 레코드 파일을 이용해서 간단한 주소록 프로그램을 하나 만들어 보자. 이 프로그램은 새로운 레코드를 추가, 삭제할 수 있고 레코드를 처음부터 끝까지 검색할 수도 있으며, 이름으로 레코드를 검색할 수도 있다. 먼저 폼을 다음 그림과 같이 디자인 한다.

이번 프로그램에서 사용할 레코드에는 이름과 주소, 전화번호 필드로 구성되어 있다. 다음과 같이 TMyRecord, TMove 그리고, 레코드와 파일 변수를 선언한다.

type

```
TMyRecord = record
```

```
    Name: String[20];
```

```
    Address: String[100];
```

```
    Phone: String[20];
```

```
end;
```

```
TRecMove = (rcFirst, rcLast, rcNext, rcPrev);
```

var

```
Form1: TForm1;
```

```
MyFile: File of TMyRecord;
```

```
MyRecord: TMyRecord;
```



```
const
```

```
    MAX = 100;
```

MAX 상수는 레코드 상한선을 정한 것으로 개발자가 마음대로 정하면 된다. 배열의 크기를 잡을 때 사용된다.

먼저 폼이 생성되는 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    AssignFile(MyFile, 'MyFile.DAT');
```

```
    if FileExists('MyFile.DAT') then
```

```
    begin
```

```
        Reset(MyFile);
```

```
        if (FileSize(MyFile) > 0) then
```

```
        begin
```

```
            Seek(MyFile, 0);
```

```
            ReadRecord;
```

```
            NewRecord := False;
```

```
        end
```

```
    end
```

```
    else
```

```
    begin
```

```
        Rewrite(MyFile);
```

```
        NewRecord := True;
```

```
    end;
```

```
    bNew.Enabled := True;
```

```
    bInsert.Enabled := False;
```

```
end;
```

간단히 설명하면 ‘MyFile.DAT’ 파일을 열고, 이미 존재한 파일이면 첫번째 레코드를 ReadRecord 라는 메소드를 이용해서 에디트 컨트롤에 읽어온다. 이때 NewRecord 라는 전역 변수를 사용했는데, 이 변수는 현재 에디트 컨트롤에 있는 내용이 새로운 내용인지를 가리키는 변수이다. ReadRecord 를 한 후에는 에디트 컨트롤의 내용이 레코드 파일에서 읽어온 내용이므로 NewRecord 를 False 로 설정한다.

그러나, 파일을 처음 생성하는 경우에는 레코드를 읽어오지 않으므로 NewRecord 를 True 로 설정한다.

폼의 OnClose 이벤트 핸들러에서는 사용한 파일을 닫아야 한다.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    CloseFile(MyFile);
end;
```

그러면 레코드 파일의 내용을 에디트 컨트롤로 읽어오는 ReadRecord 메소드와 에디트 컨트롤의 내용을 레코드 파일에 기록하는 SetRecordValue 메소드를 구현하자.

```
procedure TForm1.ReadRecord;
begin
    Read(MyFile, MyRecord);
    with MyRecord do
    begin
        Edit1.Text := Name;
        Edit2.Text := Address;
        Edit3.Text := Phone;
    end;
    Seek(MyFile, FilePos(MyFile) - 1);
end;
```

```
procedure TForm1.SetRecordValue;
begin
    with MyRecord do
    begin
        Name := Edit1.Text;
        Address := Edit2.Text;
        Phone := Edit3.Text;
    end;
end;
```

앞에서 설명했던 내용이므로 그다지 이해하기 어렵지는 않을 것이다.

이제 이동 버튼에 대한 작업을 시작하자. 제일 위 줄의 버튼들이 이동 버튼인데 ‘처음’, ‘이전’, ‘다음’, ‘끝’ 버튼을 각각 bFirst, bPrevious, bNext, bLast 라고 명명한다. 실제로는 MoveToRecord 라는 메소드를 이용하는데, 이때 앞에서 선언한 TRecMove 형의 값을 파

라미터로 호출하여 사용한다. 이들 버튼의 OnClick 이벤트 핸들러를 다음과 같이 코딩한다.

```
procedure TForm1.bFirstClick(Sender: TObject);
```

```
begin
```

```
    MoveToRecord(rcFirst);
```

```
end;
```

```
procedure TForm1.bPreviousClick(Sender: TObject);
```

```
begin
```

```
    MoveToRecord(rcPrev);
```

```
end;
```

```
procedure TForm1.bNextClick(Sender: TObject);
```

```
begin
```

```
    MoveToRecord(rcNext);
```

```
end;
```

```
procedure TForm1.bLastClick(Sender: TObject);
```

```
begin
```

```
    MoveToRecord(rcLast);
```

```
end;
```

그러면, 실제로 레코드 이동을 담당하는 MoveToRecord 를 다음과 같이 구현한다.

```
procedure TForm1.MoveToRecord(Direction: TRecMove);
```

```
var
```

```
    Pos: Integer;
```

```
begin
```

```
    EnableButtons(True);
```

```
    Pos := FilePos(MyFile);
```

```
    if (FileSize(MyFile) = 0) then Exit;
```

```
    case Direction of
```

```
        rcFirst: Pos := 0;
```

```
        rcLast: Pos := FileSize(MyFile) - 1;
```

```
        rcNext:
```

```
            if (FilePos(MyFile) < (FileSize(MyFile) - 1)) then
```

```

        Pos := FilePos(MyFile) + 1
    else Exit;
rcPrev:
    if (FilePos(MyFile) > 0) then Pos := FilePos(MyFile) - 1
    else Exit;
end;
Seek(MyFile, Pos);
ReadRecord;
NewRecord := False;
end;

```

다소 길어 보이지만, 4 가지 방향을 case 문을 이용해서 Seek 프로시저를 이용해서 이동한다. 이때 ‘이전’ 이나 ‘다음’ 버튼의 경우에는 파일의 제일 처음과 끝에 있는지에 대해서 검사하고, 그렇지 않은 경우 위치를 1 증감한다.

처음에 호출하는 EnableButtons 메소드는 새로운 레코드를 작성하거나, 여러가지 변동이 있을 때 버튼의 상태를 결정하기 위한 메소드이다. 이 메소드는 다음과 같이 구현한다.

```

procedure TForm1.EnableButtons(Enable: Boolean);
begin
    bNew.Enabled := Enable;
    bFirst.Enabled := Enable;
    bPrevious.Enabled := Enable;
    bNext.Enabled := Enable;
    bLast.Enabled := Enable;
    bInsert.Enabled := not Enable;
end;

```

간단히 설명하면 이동 버튼 들과 ‘새 레코드’ 버튼은 같게 설정하고, ‘삽입’ 버튼은 반대로 설정한다. 이는 레코드를 삽입할 때에는 반드시 ‘새 레코드’를 눌러서 ‘삽입’ 버튼이 사용 가능할 때에만 가능하도록 한 것이다. 이렇게 새 레코드를 눌렀을 때에는 레코드의 이동을 하지 못하도록 한다.

그러면, ‘새 레코드’ 버튼을 눌렀을 때 호출하는 CreateNewRecord 메소드를 구현하자.

```

procedure TForm1.CreateNewRecord;
var
    i: Integer;

```

```

begin
    if NewRecord then
        LockWindowUpdate(Handle);
        for i := 0 to ComponentCount - 1 do
            if (Components[i] is TEdit) then
                TEdit(Components[i]).Clear;
        LockWindowUpdate(0);
        NewRecord := True;
        EnableButtons(False);
    end;

    procedure TForm1.bNewClick(Sender: TObject);
    begin
        CreateNewRecord;
    end;

```

버튼의 이벤트 핸들러는 단순히 앞의 CreateNewRecord 메소드를 호출한다.

CreateNewRecord 메소드는 먼저 NewRecord 변수를 먼저 검사해서, 이 값이 True 이면 LockWindowUpdate 를 호출해서 에디트 컨트롤의 내용이 변하는 것을 막고 이들의 내용을 지운다.

그리고, EnableButton(False)를 호출해서 ‘삽입’ 버튼이 사용 가능하도록 하고, 레코드 이동 버튼 등과 ‘새 레코드’ 버튼을 사용할 수 없게 한다.

이번에는 이렇게 에디트 컨트롤에 기록된 내용을 실제로 저장, 삽입하는 부분을 구현하자.

‘저장’, ‘삽입’ 버튼의 이름을 각각 bPost, bInsert 로 명명한다. ‘저장’의 경우는 현재의 파일 포인터에 있는 레코드의 값을 변경하는 것이므로 쉽게 구현이 되지만, ‘삽입’의 경우에는 앞에서 설명한 레코드 삽입의 구현 단계에 맞추어 구현한다. 앞의 설명을 자세히 읽어보면 코드를 이해하는데 커다란 어려움이 없을 것이다.

```

procedure TForm1.UpdateRecord;
var
    CurPos: Integer;
begin
    CurPos := FilePos(MyFile);
    SetRecordValue;
    if NewRecord then
        begin

```

```

        Seek(MyFile, FileSize(MyFile));
        CurPos := FileSize(MyFile) + 1;
    end;
    Write(MyFile, MyRecord);
    if (FileSize(MyFile) > 0) then
    begin
        Seek(MyFile, CurPos);
        NewRecord := False;
    end;
    EnableButtons(True);
end;

procedure TForm1.InsertRecord:
var
    CurPos, RecordNo, i: Integer;
    Buffer: array[0..MAX] of TMyRecord;
begin
    SetRecordValue;
    CurPos := FilePos(MyFile);
    RecordNo := FileSize(MyFile);
    if FilePos(MyFile) > 0 then
    begin
        i := 0;
        Seek(MyFile, 0);
        while FilePos(MyFile) < CurPos do
        begin
            Read(MyFile, Buffer[i]);
            Inc(i);
        end;
    end;
    Buffer[CurPos] := MyRecord;
    i := CurPos + 1;
    while not EOF(MyFile) do
    begin
        Read(MyFile, Buffer[i]);
        Inc(i);
    end;
end;

```

```

end;
i := 0;
Seek(MyFile, 0);
while (i <= RecordNo) do begin
    Write(MyFile, Buffer[i]);
    Inc(i);
end;
Seek(MyFile, CurPos);
ReadRecord;
EnableButtons(True);
end;

```

```

procedure TForm1.bPostClick(Sender: TObject);
begin
    UpdateRecord;
end;

```

```

procedure TForm1.bInsertClick(Sender: TObject);
begin
    InsertRecord;
end;

```

레코드를 삭제하는 버튼인 ‘삭제’ 버튼은 bDelete 로 명명한다. 레코드의 삭제를 구현하는 단계에 대해서도 앞에서 설명했으므로, 다음의 코드를 이해하는 것이 그다지 어렵지 않을 것이다.

```

procedure TForm1.DeleteRecord;
var
    CurPos, RecordNo, i: Integer;
    Buffer: array[0..MAX] of TMyRecord;
begin
    if MessageDlg('정말 지우시겠습니까?', mtConfirmation, [mbYes, mbNo], 0) = mrNo
    then Exit;
    if NewRecord then
    begin
        ReadRecord;

```

```

    NewRecord := False;
    EnableButtons(True);
    Exit;
end;
CurPos := FilePos(MyFile);
RecordNo := FileSize(MyFile) - CurPos - 1;
if (FilePos(MyFile) < (FileSize(MyFile) - 1)) then
begin
    Seek(MyFile, FilePos(MyFile) + 1);
    i := 0;
    while not EOF(MyFile) do
    begin
        Read(MyFile, Buffer[i]);
        Inc(i);
    end;
    Seek(MyFile, CurPos);
    Truncate(MyFile);
    for i := 0 to RecordNo - 1 do
        Write(MyFile, Buffer[i]);
    end
else
begin
    Truncate(MyFile);
    Dec(CurPos);
end;
Seek(MyFile, CurPos);
ReadRecord;
end;

procedure TForm1.bDeleteClick(Sender: TObject);
begin
    DeleteRecord;
end;

```

마지막으로 검색 부분을 구현하자. 검색 버튼으로 ‘찾기’와 ‘다음’ 버튼이 있다. 즉, 이름이 일치하는 첫번째 레코드를 찾을 때에는 ‘찾기’로 검색하고, 그 위치 이후에 계속 일치하



는 것을 찾을 때에 ‘다음’ 버튼을 사용한다. 이들 버튼의 이름을 각각 bFind, bFindNext 라 명명하자. 그리고, 이들의 구현은 실제로 검색하는 방법이 동일하므로 LocateRecord 라는 동일한 메소드를 사용하되, 파라미터로 검색할 문자열과 ‘찾기’ 버튼의 경우에는 파일의 처음부터, ‘다음’ 버튼의 경우에는 현재 파일 포인터 이후부터 검색하도록 구현한다. 이 메소드는 다음과 같이 구현한다.

```
procedure TForm1.LocateRecord(Value: String; FromBOF: Boolean);
```

```
var
```

```
    CurPos, SearchPos: Integer;
```

```
    Found: Boolean;
```

```
begin
```

```
    CurPos := FilePos(MyFile);
```

```
    if FromBOF then SearchPos := 0
```

```
    else SearchPos := CurPos + 1;
```

```
    Found := False;
```

```
    while (SearchPos <= (FileSize(MyFile) - 1)) and (not Found) do
```

```
    begin
```

```
        Seek(MyFile, SearchPos);
```

```
        Read(MyFile, MyRecord);
```

```
        if MyRecord.Name = Value then
```

```
        begin
```

```
            Found := True;
```

```
            Seek(MyFile, SearchPos);
```

```
            ReadRecord;
```

```
        end;
```

```
        Inc(SearchPos);
```

```
    end;
```

```
    if not Found then ShowMessage('해당 레코드가 없습니다.');
```

```
end;
```

```
procedure TForm1.bFindClick(Sender: TObject);
```

```
begin
```

```
    if (Edit4.Text <> '') then
```

```
    begin
```

```
        if NewRecord then bPostClick(Self);
```

```
        LocateRecord(Edit4.Text, True);
```

```

    end;
end;

procedure TForm1.bFindNextClick(Sender: TObject);
begin
    LocateRecord(Edit4.Text, False);
end;

```

검색하는 방법은 소스를 보면 쉽게 이해할 수 있을 것이다. 파라미터에 따라 파일의 첫 번째 레코드 또는 현재 레코드부터 순차적으로 검색해 나가는 방법이며, 파일 마지막에 도달하면 메시지 박스를 띄우고 종료하게 된다.

이로써 레코드 파일을 이용한 예제 프로그램이 완성되었다. 직접 사용해 보면 알겠지만, 코딩하고 신경써야 할 부분이 다소 많기는 해도 나름대로 빠르고 그럴 듯한 어플리케이션을 제작할 수 있다.

물론 이런 형태의 데이터베이스 어플리케이션은 BDE 를 사용하기 보다는, 클라이언트 데이터 세트나 사용자 정의 데이터 세트에 플랫폼 파일을 이용해서 접근하면 더욱 견고하고 관리가 편하게 만들 수 있다. 하지만, 이렇게 레코드 파일을 적절히 이용하는 것도 괜찮을 것이다.

## 오브젝트 파스칼의 파일 관련 루틴

오브젝트 파스칼의 런타임 라이브러리에는 흔히 쓰이는 파일 처리 루틴을 포함하고 있다. 이 때 파일 이름을 사용하거나, 파일의 핸들을 사용하게 된다. 기본적인 루틴은 앞에서 많이 설명했으므로, 유틸리티 루틴에 대해서 알아보도록 하자.

### ● 파일의 삭제

파일을 삭제하는 것은 비교적 간단하다. 단순히 DeleteFile 함수를 사용하면 되는데, Win32 의 SHFileOperation 기능처럼 복구는 지원하지 않는다. 코드는 다음과 같이 하면 된다.

```
DeleteFile(FileName);
```

DeleteFile 루틴은 파일을 삭제했으면 True 를 파일이 읽기 전용이거나, 존재하지 않아서 파일을 삭제하지 못했으면 False 를 반환한다.

## ● 파일 찾기

파일을 검색할 때에는 FindFirst, FindNext, FindClose 의 3 가지 루틴이 사용된다. FindFirst 는 지정된 디렉토리의 검색 조건에 맞는 첫번째 파일 이름을 가져오며, FindNext 는 일단 한번 검색된 이후에 맞는 조건의 파일을 가져온다. FindClose 는 FindFirst 에 의해 할당된 메모리를 해제한다. 파일이 존재하는지 알고 싶을 때에는 레코드 파일의 예제에서도 사용한 바 있는 FileExists 함수를 사용하는데, 이 함수는 파일이 있을 경우 True, 없을 경우 False 를 반환한다.

이 루틴 들은 TSearchRec 데이터 형의 파라미터를 가지는데, TSearchRec 구조체는 다음과 같은 파일 정보를 담고 있다.

type

```
TFileName = string;
TSearchRec = record
    Time: Integer;           //파일에 기록된 시간
    Size: Integer;          //파일의 크기 (byte)
    Attr: Integer;          //파일의 속성
    Name: TFileName;        //DOS 파일 이름과 확장자
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData; //파일 생성시간, 마지막 접근 시간, 긴 파일 이름
end;
```

파일을 찾으면, TSearchRec 파라미터의 필드가 찾은 파일의 내용으로 채워지게 된다.

## ● 파일의 속성 바꾸기

모든 파일은 비트맵 플래그로 다양한 파일의 속성을 가지고 있다. 파일의 속성을 바꾸기 위해서는 속성을 읽고, 변경하고, 설정하는 3 가지 단계를 거치게 된다.

### 1. 파일 속성 읽기:

운영체제는 파일 속성을 비트맵의 형태로 저장한다. 이런 파일 속성을 읽어오기 위해서는 FileGetAttr 함수를 이용한다. 이 함수에 파일 이름을 지정하면, 그 파일의 속성이 반환된다. 반환값은 Word 형으로 TSearchRec 구조체에 정의되어 있는 Attr 필드에서 사용될 수 있는 여러 가지 종류의 상수값과 and 연산을 해서 알아낼 수 있다. 이 값이 -1 이면 에

러가 발생한 것이다.

## 2. 파일 속성 변경

텔파이에서 파일 속성은 세트로 나타난다. 그러므로, 4 장에서 설명한 것처럼 여러가지 논리 연산을 통해서 개별적 속성을 다룰 수 있다. 각 속성의 상수값은 SysUtils.pas 유닛에 정의되어 있으며, 도움말을 찾아보면 알 수 있다. 예를 들어, 파일의 읽기 전용 속성을 설정하려면 다음과 같이 하면 된다.

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

세트 데이터 형의 특성을 이용하여 한 번에 여러 개의 속성을 설정할 수도 있다. 예를 들어, 다음의 코드는 시스템 파일과 파일 숨김 속성을 제거한다.

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

## 3. 파일 속성 설정

이렇게 변경된 파일 속성은 FileSetAttr 함수를 이용해서 실제로 반영하도록 해야 한다. 이 함수에 파일의 이름을 설정하고, 속성을 넘겨주면 된다.

## 파일 스트림의 활용

TFileStream 은 고수준의 파일을 다루기 위해서 사용되는 VCL 클래스이다.

TFileStream 은 TStream 클래스에서 상속받으며, 그렇기 때문에 자동으로 지속성(persistence)을 지원한다. 스트림 클래스는 TFileer, TReader, TWriter 클래스를 가지고 작업할 수 있으며, 같은 코드로 VCL 스트림을 지원한다.

TFileStream 은 다른 스트림 클래스와 쉽게 상호작용할 수 있다. 예를 들어, 동적 메모리 블록을 디스크에 덤프하고자 하면, TFileStream 과 TMemoryStream 을 사용하면 된다.

### ● 스트림이란 ?

스트림이란 바이트 들의 연속이라고 말할 수 있다. 연속된 바이트 내에서 임의의 위치에 접근하여 이를 읽어들이거나 또는 임의의 바이트를 기록할 수 있다. 대개의 경우 데이터는 바이트나 문자열이지만 그 외의 모든 데이터 형의 데이터를 읽고, 쓰기위한 스트림 클래스도 작성할 수 있다. 스트림 클래스는 추상적 클래스인 TStream 으로부터 계승된다.

TStream 객체는 하나의 스트림을 현재 위치(position)를 기억하고 있는 바이트의 연속으로 취급한다. 스트림에서 위치란 입출력이 발생하는 곳을 의미하는데, 이 때 입출력이란 현재의 위치에서 다음 바이트 또는 연속된 바이트를 읽거나 쓰는 것을 의미한다.

TStream 클래스는 추상적 클래스일 뿐이므로 스트림의 프로토콜을 정의할 뿐이다. 파일을 읽고 쓸 수 있는 스트림을 생성하기 위해서는 TFileStream 을 이용해야 한다.

## ● 파일의 생성과 열기

파일을 생성하고, 이를 열어서 파일의 핸들을 얻으려면, TFileStream 객체의 인스턴스를 생성하기만 하면 된다. 파일이 열리지 않으면, TFileStream 은 예외를 발생시킨다.

생성자의 선언부는 다음과 같다.

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

Mode 파라미터는 파일 스트림을 생성할 때 파일을 어떤 식으로 접근할 것인지를 결정한다. Mode 파라미터에는 다음과 같은 것이 있다.

값	의 미
fmCreate	주어진 파일을 생성한다. 파일이 존재하면, 파일을 write 모드로 연다.
fmOpenRead	파일을 읽기 전용으로 연다.
fmOpenWrite	파일을 쓰기 전용으로 연다. 내용을 쓰면 현재의 내용을 완전히 대체한다.
fmOpenReadWrite	파일의 내용을 수정하고자 할 때 많이 사용하는 열기 모드이다.
fmShareCompat	FCB 가 열리는 방법으로 공유 방법을 결정
fmShareExclusive	다른 어플리케이션은 열린 파일에 접근할 수 없다.
fmShareDenyWrite	다른 어플리케이션이 읽기 위해 파일을 열 수 있으나, 쓰지는 못한다.
fmShareDenyRead	다른 어플리케이션이 쓰기 위해 파일을 열 수 있으나, 읽지는 못한다.
fmShareDenyNone	다른 어플리케이션이 마음대로 파일을 읽고 쓸 수 있다.

## ● 파일 핸들

TFileStream 객체를 인스턴스화하면 파일 핸들에 접근할 수 있게 되는데, 파일 핸들은 Handle 프로퍼티에서 얻을 수 있다. 핸들은 읽기 전용이다. 참고로, 파일 핸들의 속성을 바꾸고자 한다면, 새로운 파일 스트림 객체를 생성해야 한다. 윈도우 API 함수를 사용할 때에는 핸들을 파라미터로 사용할 경우가 많이 있다.

## ● 파일 읽기와 쓰기

TFileStream 은 파일에 데이터를 읽고, 쓰는데 몇가지 다른 메소드를 제공한다. 이들을 특징에 따라 구분하면 기록하거나 읽은 바이트 수를 반환하는지 ?, 예러가 발생할 때 예외를 발생시키는지 ?, 기록하거나 읽을 바이트 수를 요구하는지 ? 등을 생각할 수 있다.

Read 메소드는 Count 파라미터에 지정한 바이트 수만큼 현재의 위치에서 버퍼로 읽어들이는 메소드이다. 파일을 읽고 나면, 읽은 바이트 수만큼 현재 위치를 이동하게 되며 실제로 전송된 바이트 수를 반환한다. 이 메소드의 선언부는 다음과 같다.

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

Read 메소드는 파일의 크기를 모를 때 유용하다.

Write 메소드는 Count 파라미터에 지정한 바이트 수 만큼 버퍼에서, 파일 스트림의 현재 위치부터 기록을 한다. 그리고, 실제로 기록한 바이트 수를 반환한다. 이 메소드의 선언부는 다음과 같다.

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

ReadBuffer 와 WriteBuffer 메소드는 Read, Write 메소드와는 달리 읽거나, 기록한 바이트 수를 반환하지 않는다. 이들 프로시저는 구조체를 읽을 때와 같이 읽거나 쓸 데이터의 바이트 수를 정확히 알 때 많이 사용한다. 또한, ReadBuffer 와 WriteBuffer 메소드는 예러가 발생할 경우 EReadError, EWriteError 예외를 발생시킨다. 이들 메소드의 선언부는 다음과 같다.

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

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

## ● 문자열 읽기와 쓰기

Read, Write 함수에서 문자열을 사용하려면 문법적으로 다소 고려해야 할 부분들이 있다. Read 메소드의 Buffer 파라미터는 var 로 선언되어 있고, Write 메소드의 Buffer 파라미터는 const 로 선언되어 있다. 이들은 데이터 형이 정해져 있지 않기 때문에, 루틴은 변수의 주소를 사용하게 된다. 가장 흔히 사용하게 되는 데이터 형은 긴 문자열인데, 긴 문자열에는 문자열의 길이와 참조 계수 등의 내용이 담겨 있기 때문에 단순히 역참조(dereference)로 해결이 되지 않는다. 제대로 사용하려면 일단 문자열을 포인터나 PChar 로 형변환을 하고 나서 역참조를 해야 한다. 예제 코드를 소개하면 다음과 같다.

```

procedure UsingStrings;
var
    FS: TFileStream;
    S: string = 'Hello';
begin
    FS := TFileStream.Create('foo', fmCreate or fmOpenRead);
    FS.Write(s, Length(s));           //이렇게 하면 쓰레기 값이 저장된다.
    FS.Write(PChar(s)^, Length(s));   //이 방법이 옳다.
end;

```

## ● 파일 찾기

TFileStream 은 Seek 메소드를 통해서 검색 기능을 제공한다. Seek 메소드의 선언부는 다음과 같다.

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

Origin 파라미터를 이용해서 Offset 파라미터를 어떻게 해석할 지를 결정하게 된다. Origin 파라미터가 가질 수 있는 값에는 다음과 같은 것들이 있다.

값	의 미
soFromBeginning	오프셋이 리소스의 처음부터 시작된다. 오프셋은 0 보다 커야 하며, Seek 메소드에 의해 Offset 파라미터에 지정된 위치로 이동한다.
soFromCurrent	오프셋이 리소스의 현재 위치에서 시작된다. Seek 메소드에 의해 현재 위치 + Offset 파라미터의 위치로 이동한다.
soFromEnd	오프셋이 리소스의 끝에서 시작된다. 오프셋은 0 보다 작아야 하며, 파일의 끝에서 오프셋에 지정된 바이트 수만큼 이전의 위치에서 시작된다.

Seek 메소드는 스트림의 현재 위치를 재설정하고, Position 프로퍼티의 새로운 값을 반환하게 된다.

## ● 파일 위치와 크기

TFileStream 은 파일의 현재 위치와 크기를 Position, Size 프로퍼티에서 얻을 수 있다. 이 프로퍼티 들은 Seek, Read, Write 메소드에 의해 사용된다. 이들 프로퍼티의 값은 바이트

단위로 나타난다.

Size 프로퍼티를 설정하면 파일의 크기를 변경하게 된다. 파일의 크기가 바뀔 수 없는 경우에 이 프로퍼티를 바꾸려고 하면, 예외가 발생한다. 예를 들어, fmOpenRead 모드로 열린 파일의 Size 프로퍼티를 변경하려고 하면 예외가 발생할 것이다.

## ● 파일 스트림의 복사

파일 스트림을 복사할 때에는 CopyFrom 메소드를 사용한다. 이 메소드의 선언부는 다음과 같다.

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

CopyFrom 메소드를 사용하면 사용자가 간단하게 데이터를 사용할 수 있다. 이 메소드는 Source 파라미터에 지정된 스트림에서 Count 바이트 만큼 복사해 오는데, 실제로 복사한 바이트 수를 반환한다. Count 파라미터를 0 으로 설정하면 Source 의 전체 내용을 복사해 온다. Count 값이 0 이 아니면 Source 스트림의 현재 위치에서 지정한 수만큼 복사하게 된다.

## ● 파일 스트림을 이용한 파일 복사

그러면, 지금까지 설명한 파일 스트림을 이용해서 파일을 복사하는 루틴을 소개한다. 이 루틴을 살펴보면 구체적인 파일 스트림의 사용 방법을 알 수 있을 것이다.

```
procedure FileCopy(const SrcFile, DestFile: String);
const
    BufSize = 8192;
var
    SrcStream, DestStream: TFileStream;
    Buffer: Pointer;
    ReadCount: Integer;
begin
    SrcStream := TFileStream.Create(SrcFile, fmOpenRead or fmShareDenyWrite);
    //읽기 전용으로 소스 파일을 연다.

    try
        DestStream := TFileStream.Create(DestFile, fmCreate or fmShareExclusive);
        //기록할 파일이 존재하지 않으면 새로 생성하고, 존재하면 덮어 쓴다.
```



```

        //그리고, 다른 사용자는 이 파일에 접근하지 못한다.
    try
        GetMem(Buffer, BufSize);
    try
        repeat
            ReadCount := SrcStream.Read(Buffer^, BufSize); //BufSize 만큼 읽는다.
            DestStream.WriteBuffer(Buffer^, ReadCount);    //버퍼의 내용을 읽은 만큼 기록한다.
        until ReadCount = 0;                               //소스 파일의 끝까지 읽었음
    finally
        FreeMem(Buffer, BufSize);
    end;
finally
    DestStream.Free;
end;
finally
    SrcStream.Free;
end;
end;

```

이 파일 복사 루틴은 소스 파일을 읽을 때에는 Read 메소드를 사용하였다. 이 메소드를 이용하면 실제로 읽어들이는 바이트 수를 알 수 있기 때문에 이를 사용해야 한다. 대부분의 파일의 크기가 버퍼의 크기(8192)의 배수가 아니기 때문에, Read 메소드를 이용해서 실제로 읽은 바이트 수를 알아야 한다. 그리고, 기록을 할 때에는 Write, WriteBuffer 중 아무거나 사용해도 된다.

물론 이 파일 복사 루틴은 앞에서 설명한 CopyFrom 메소드를 이용하면 훨씬 더 짧고 간단하게 작성할 수 있다. 그렇지만, 지금까지 설명한 것을 예제로 하기에는 너무 단순해서 조금 복잡한 방법을 소개하였다. CopyFrom 메소드를 이용하면 다음과 같이 간단하게 작성 가능하다.

```

procedure FileCopy(const SrcFile, DestFile: string);
var
    SrcStream, DestStream: TFileStream;
begin
    SrcStream := TFileStream.Create(SrcFile, fmOpenRead);
    try
        DestStream := TFileStream.Create(DestFile, fmCreate);
    
```

```

try
    DestStream.CopyFrom(SrcStream, 0);
finally
    DestStream.Free;
end;
finally
    SrcStream.Free;
end;
end;

```

## Win32 Shell API 의 활용

Win32 기반의 운영체제인 윈도우 95/98/NT 4.0 에서는 기본적으로 파일을 다룰 때 과거의 도스에서 쓰던 디렉토리 와 파일의 방식으로 다루는 것이 아니라, 네트워크 환경과 컴퓨터에서 접근할 수 있는 모든 논리적인 객체를 가상 폴더(virtual folder)를 중심으로 접근하도록 변경되었다.

윈도우 3.1 을 쓰다가 윈도우 95 의 탐색기를 써보면, 이 차이를 잘 느낄 수 있을 것이다. 윈도우 95 의 탐색기에는 파일이 아닌 ‘내 컴퓨터’나 ‘휴지통’과 같은 가상 폴더에 파일에 접근하는 것과 같은 방식으로 접근하게 된다.

그렇기 때문에, 윈도우 95 는 과거에 도스에서 쓰던 방식으로 파일에 접근하지 않고 Win32 에서 제공되는 셸의 파일 처리 API 를 이용하여 이들을 처리하게 된다.

Win32 Shell 에 관한 내용은 방대하면서도 복잡하지만, 여기서는 파일 처리에 관한 부분 만을 다루도록 한다.

### ● SHFileOpStruct 구조체와 SHFileOperation API

ShellAPI.pas 유닛에 정의되어 있는 SHFileOpStruct 구조체와 SHFileOperation API 함수를 사용하면 파일을 삭제할 때 휴지통의 사용을 지정할 수도 있으며, 그 밖에 복사, 이동, 이름 변경 등의 여러가지 조작을 가할 수 있다. 다음의 프로시저는 파일을 삭제하는 예이다.

```

procedure DeleteFiles;
var
    T: TSHFileOpStruct;
    X: Integer;
begin

```

```

with T do
begin
    Wnd := 0;
    wFunc := FO_DELETE;
    pFrom := 'E:WTempWTestW*. *';
    fFlags := FOF_ALLOWUNDO or FOF_FILESONLY or FOF_SILENT or
        FOF_NOCONFIRMATION;
end;
SHFileOperation(T);
end;

```

이 프로시저는 E:WTempWTest 디렉토리에 있는 모든 파일을 지운다. 이때 지정한 플래그에 따라 동작을 달리 하는데, FOF\_ALLOWUNDO 플래그를 지정 했으므로 휴지통에 파일을 버리게 되며 FOF\_NOCONFIRMATION 플래그는 파일을 지울 때 확인 대화상자를 띄우지 않는 것을 의미한다. 또한, FOF\_SILENT 플래그를 지정 했으므로 파일을 삭제할 때 파일이 삭제되는 진행 상황을 나타내는 애니메이션을 보여주지 않는다.

TSHFileOpStruct 구조체를 사용할 때에는 몇 가지 주의할 사항이 있다.

우선 파일의 패스를 지정할 때에는 항상 완전한 패스 이름을 적어주어야 한다. 현재의 디렉토리를 이용해서 작업을 할 경우 보통 때에는 문제가 없으나, FOF\_ALLOWUNDO 플래그를 지정해서 파일을 휴지통에 버릴 경우, 이를 복구할 때 원래의 디렉토리를 제대로 찾지 못하는 문제가 생기게 된다.

또한, pFrom 멤버에 파일을 여러 개 지정할 때에는 각 파일 이름 사이는 #0(Null) 문자로 분리되어야 하며, 마지막 파일 이름 뒤에는 Null 문자를 2 개 연달아 배치해야 한다. 그러므로, 델파이에서는 아래와 같은 식으로 문자열을 조작해줄 필요가 있다.

```

var
    FileList: String;
    FOS: TSHFileOpStruct;
begin
    FileList := 'c:Wdelete.met' + #0 + 'c:WwindowsWtemp.$$$' + #0 + #0;
    FileList := Filename1 + #0 + Filename2 + #0#0;
    FOS.pFrom := PChar(FileList);
end;

```

- 시스템에서 정의한 아이콘 나타내기

이 작업을 하려면 SHGetFileInfo 함수와 TSHFileInfo 구조체를 이해해야 한다. 이들은 ShellAPI.pas 유닛에 선언되어 있다.

SHGetFileInfo API 함수는 파일의 패스를 지정해주면 그 파일에 대한 속성을 TSHFileInfo 구조체에 담아서 돌려 준다. 선언부는 다음과 같다.

```
function SHGetFileInfo(pszPath: PAnsiChar; dwFileAttributes: DWORD;
    var psfi: TSHFileInfo; cbFileInfo, uFlags: UINT): DWORD; stdcall;
```

파라미터를 설명하면, pszPath 파라미터에는 알아볼 파일의 패스를, dwFileAttributes 파라미터는 잘 사용하지 않으며, psfi 파라미터에는 파일의 정보를 담아올 구조체의 변수를 담아야 하며, cbFileInfo 와 uFlags 파라미터에는 각각 구조체의 크기와 플래그를 지정한다. 이렇게 호출하고 나면, TSHFileInfo type 의 구조체 변수에 파일에 대한 정보가 담기게 되는데, 이 구조체의 선언부는 다음과 같다.

```
TSHFileInfoA = record
    hlcon: HICON;                //아이콘의 핸들
    ilcon: Integer;              //아이콘의 인덱스
    dwAttributes: DWORD;         //파일의 속성 플래그
    szDisplayName: array [0..MAX_PATH-1] of AnsiChar; //디스플레이 이름이나 패스
    szTypeName: array [0..79] of AnsiChar;             //type 의 이름
end;
```

즉, hlcon 멤버에는 파일을 대표하는 아이콘의 핸들, ilcon 에는 아이콘의 인덱스가 들어 있게 된다.

그럼 해당하는 디렉토리의 파일들을 보여줄 때에는 이들의 아이콘을 TImageList 에 담아서 보여주면 된다. 다음 코드를 살펴 보자.

```
Images := TImageList.Create(Self);
Images.ShareImages := True;
Images.DrawingStyle := dsTransparent;

var
    sfi: TSHFileInfo;
    ...

Images.Handle := SHGetFileInfo(Path, 0, sfi, SizeOf(sfi), SHGFI_SYSICONINDEX
```

```
+ SHGFI_SMALLICON);  
Images.Draw(Canvas, 0, 0, sfi.ilcon);
```

TImageList 컴포넌트의 ShareImages 프로퍼티는 TImageList 컴포넌트가 파괴될 때, 이미지의 핸들도 같이 해제할 것인지 여부를 결정짓는 프로퍼티이다. 이 값을 True 로 설정하면 TImageList 컴포넌트가 파괴되어도 핸들이 해제되지 않는다.

DrawingStyle 프로퍼티는 이미지를 그리는 스타일을 지정하는 것으로 dsFocus, dsSelected, dsNormal, dsTransparent 가 있다. 각각의 값은 다음의 의미를 가진다.

dsFocused	시스템의 highlight 색상과 25% 섞어서 이미지를 나타낸다. 이 값은 마스크를 가진 이미지 리스트에만 영향을 미친다.
dsSelected	시스템의 highlight 색상과 50% 섞어서 이미지를 나타낸다. 역시 마스크를 가진 이미지 리스트에만 영향을 미친다.
dsNormal	BkColor 프로퍼티에 지정된 색상을 이용해서 이미지를 그린다. BkColor 프로퍼티에 clNone 이 지정되어 있으면 dsTransparent 와 같게 동작한다.
dsTransparent	BkColor 설정과 관계 없이 마스크를 이용해서 이미지를 그린다.

## 인쇄 기능의 추가

보통 델파이를 가지고 어플리케이션을 만들 때, 인쇄가 필요한 경우에는 델파이 3 부터 기본 제공되는 QuickReport 를 사용하게 된다. 그런데, QuickReport 를 사용하지 않고도 직접 인쇄 기능을 수행할 수 있는 방법이 있다.

QuickReport 보다는 사용 방법도 어렵고, 이쁘게 만들기도 어렵지만 직접 인쇄를 제어하는 방법을 익혀 놓으면 좀더 정교하면서도 깨끗한 인쇄를 가능하게 할 수도 있다.

### ● 인쇄의 기본

가장 원시적인 방법으로 인쇄하는 것은 파일 변수를 생성해서 문자열을 인쇄하는 것이 가능하다. 도스 시절의 표준 파스칼에서부터 사용하던 방식이다.

다음의 코드를 살펴 보자.

```
var  
  P: TextFile;  
begin  
  AssignPrn(P);  
  Rewrite(P);
```

```

WriteLn(P, '테스트 입니다.');
```

```

CloseFile(P);
```

```

end;
```

여기서 P 라는 변수를 TextFile 로 선언하고 AssignPrn 루틴을 사용하면 프린터 포트를 변수 P 에 대입하게 되며, ReWrite 문을 통해 프린터 포트를 연다. 그리고, WriteLn 프로시저를 사용하면 프린터로 텍스트를 전송할 수 있다. 마지막으로 사용이 끝난 후에는 반드시 CloseFile 을 이용하여 프린터 포트를 닫아주어야 한다.

## ● TPrinter 객체의 사용

델파이에서 윈도우의 프린터 인터페이스에 접근하기 위해서는 TPrinter 객체를 사용한다. 델파이의 Printer.pas 유닛에는 Printer 변수가 선언되어 있는데, 이것은 TPrinter 클래스로 선언되어 있다. 그러므로, TPrinter 객체를 사용하기 위해서는 Printers.pas 유닛을 uses 절에 포함시키고 사용해야 한다.

TPrinter 객체의 프로퍼티와 메소드에 대해 다음에 정리하였다.

프로퍼티/메소드	설 명
Canvas	TCanvas 클래스로 선언된다. Canvas 는 인쇄되기 전에 메모리에 페이지나 문서를 생성되는 곳이다. Canvas 에는 Pen, Brush 등의 속성을 가지고 있는데, 이들은 그림을 그리거나 텍스트를 추가하는데 사용한다.
TextOut	TCanvas 클래스의 메소드로서, Canvas 에 텍스트를 전달하는 역할을 한다.
BeginDoc	인쇄 작업을 시작할 때 사용한다.
EndDoc	인쇄 작업을 종료할 때 사용한다. EndDoc 이 호출되어야만 실제 인쇄 작업이 시작된다.
PageHeight	페이지 높이를 픽셀 단위의 값으로 반환한다.
PageWidth	페이지 폭을 픽셀 단위의 값으로 반환한다.
NewPage	강제로 새로운 페이지로 이동하도록 하며, Canvas 의 Pen 속성 값을 (0, 0)으로 초기화 시킨다.
PageNumber	현재 인쇄 중인 페이지의 수를 반환한다.
Aborted	읽기 전용으로 이 값이 True 이면 출력이 사용자에게 의해 중지된 것이다. 그러므로, 그리기 명령을 오랫동안 수행할 때에는 이 프로퍼티를 자주 확인하여 사용자의 입력에 빠르게 반응해야 한다.
Copies	현재 출력하는 문서의 매수를 지정할 수 있다.
Fonts	읽기 전용으로 프린터가 지원하는 모든 폰트의 리스트이다.
Handle	읽기 전용으로 현재 선택된 프린터의 핸들을 반환한다. 출력 중에는 이 프로퍼티

	의 값이 Canvas.Handle 프로퍼티와 같다. 이 프로퍼티는 다양한 디바이스 핸들을 사용하는 GetDeviceCaps 같은 API 함수들에게 정보를 제공할 때 사용한다.
Orientation	poPortrait 로 설정되면 문서는 일반적인 방향으로 출력되며, poLandscape 로 선택된 경우 문서를 넓게 찍는다.
Printing	읽기 전용으로 현재 출력 중인지 여부를 알려 준다.

TPrinter 클래스를 사용할 때에는 먼저 Title 프로퍼티를 설정하고, BeginDoc 프로시저를 호출한 후 Canvas 프로퍼티에 마음대로 그리고, 한 페이지를 넘는 경우에는 NewPage 를 호출하고 다시 Canvas 프로퍼티에 그리게 된다. 출력이 끝났으면 EndDoc 을 호출하면 된다.

다음의 코드는 첫번째 페이지에는 타원을 하나 그리고, 다음 페이지에는 사각형을 하나 그린다.

```
Printer.Title := '테스트 입니다 !';
Printer.BeginDoc;
Printer.Canvas.Ellipse(50, 50, 200, 200);
Printer.NewPage;
Printer.Canvas.Rectangle(50, 50, 200, 20);
Printer.EndDoc;
```

그러면 간단한 예제로 비트맵 파일을 용지의 한가운데에 인쇄하는 예제를 하나 만들어 보자. 폼을 다음과 같이 버튼 2 개와 이미지 컴포넌트와 TOpenPictureDialog 컴포넌트를 하나씩 올려 놓자.

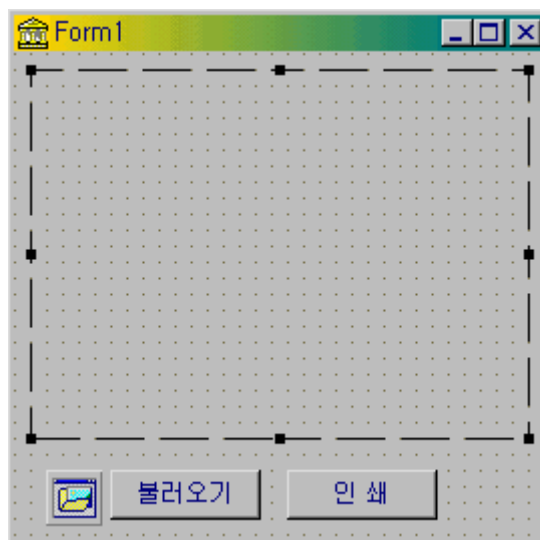


Image1 의 Stretch 프로퍼티를 True 로 설정하여 그림이 크기에 맞도록 한다.

Button1 을 클릭하면 그림을 불러올 수 있도록 다음과 같이 OnClick 이벤트 핸들러를 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if OpenPictureDialog1.Execute then
        Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

그리고, 실제 인쇄 루틴을 Button2 를 클릭하면 실행해야 하므로 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
    PrnRect, ImgRect: TRect;
    XCenter, YCenter: Integer;
begin
    with Printer do
        begin
            BeginDoc;
            XCenter := PageWidth div 2;
            YCenter := PageHeight div 2;
            PrnRect := Rect(XCenter div 2, YCenter div 2,
                PageWidth - (XCenter div 2), PageHeight - (YCenter div 2));
            ImgRect := Rect(Image1.Left, Image1.Top,
                Image1.Left + Image1.Width, Image1.Top + Image1.Height);
            Canvas.CopyRect(PrnRect, Form1.Canvas, ImgRect);
            EndDoc;
        end;
    end;
```

그림을 페이지의 중앙에 위치시키며, 구역을 용지 전체 크기로 설정하여 인쇄하는 루틴이다. 이때 그래픽을 프린터의 캔버스에 위치시키기 위해서 CopyRect 메소드를 사용했는데, 이 메소드를 이용하면 화면의 캔버스 영역을 프린터의 캔버스 영역으로 복사하게 된다.



CopyRect 메소드의 선언부는 다음과 같다.

```
procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

목적지와 원본의 사각형이 TRect 형의 변수로 전달되는데, 이 경우에 원본 캔버스는 Form1의 캔버스이다. CopyRect 문에서 Image1이 아닌, 폼의 캔버스로부터 이미지를 복사하도록 하기 위해 ImgRect의 구역을 Image1의 전체가 들어가도록 설정하였다.

TRect 데이터 형은 4개의 값이 필요하며, 4개의 정수값을 이용하여 TRect 데이터 형으로 변환시킬 때 Rect 함수를 사용한다. Button2의 OnClick 이벤트 핸들러에서 PrnRect는 프린터의 캔버스의 구역을 저장하며, ImgRect는 폼의 캔버스의 구역을 저장한다. 즉, 다음의 코드들은 PrnRect는 프린터 페이지의 중앙 1/2 영역을 지정하는 것이며, ImgRect는 폼의 Image1 컨트롤의 전체 크기를 지정하는 것이다.

```
PrnRect := Rect(XCenter div 2, YCenter div 2,  
    PageWidth - (XCenter div 2), PageHeight - (YCenter div 2));  
ImgRect := Rect(Image1.Left, Image1.Top,  
    Image1.Left + Image1.Width, Image1.Top + Image1.Height);
```

나머지 코드는 그다지 어려운 내용이 아니므로, 설명을 생략하겠다.

이것으로 간단한 이미지 인쇄 프로그램을 작성하였다. 컴파일하고 실행해서 직접 인쇄를 해보도록 하자.

#### ● GetDeviceCaps 함수의 활용

지금까지는 인쇄를 할 때 픽셀 단위로 모든 처리를 해왔다. 이 방법은 프린터의 픽셀 크기가 제각기 다르기 때문에, 어떤 프로그램에서 페이지에 인치나 밀리미터 단위로 그리기를 원하게 된다. 이를 해결하기 위해서는 먼저 GetDeviceCaps 함수를 호출하여, 선택한 유닛당 픽셀의 수를 결정해야 한다. GetDeviceCaps 함수는 다음과 같이 선언되어 있다.

```
function GetDeviceCaps(DC: HDC; Index: Integer): Integer;
```

이 함수는 디바이스 컨텍스트의 핸들과 디바이스 특성 중에서 원하는 자료를 상수로 입력받는다. 그리고, 실제 입력에 해당되는 값을 반환하게 된다. TPrinter의 경우에는 Handle이나 Canvas.Handle과 LOGPIXELSX와 LOGPIXELSY를 파라미터로 입력하면 논리적 인치당 픽셀 수가 반환된다. 하지만 화면에서의 논리적 인치라는 개념은 실제 인치보다 조금 큰데, 그 원인은 8 이하의 작은 폰트를 눈에 보이도록 하기 위해 개발된 것이 논리

적 인치이기 때문이다. 그러나, 프린터의 경우에는 논리적 인치와 실제 인치의 차이가 거의 없다.

각각의 축에 대해 상수 값으로 LOGPIXELSX 는 수평, LOGPIXELSY 는 수직을 나타낸다. 스크린에서 두 축은 서로 다른 값을 반환한다. 즉, LOGPIXELSX 로 GetDeviceCaps 를 호출한 경우와 LOGPIXELSY 로 호출한 경우 다른 값이 반환된다. 이는 화면 상에서 수직 방향 단위당 픽셀수와 수평 방향 단위당 픽셀 수가 다르기 때문인데, 프린터의 경우에는 픽셀이 수평, 수직의 크기가 같기 때문에 GetDeviceCaps 함수를 HORSIZE 나 VERTSIZE 로 호출하여도 같은 값이 반환된다.

TPrinter.Canvas 는 일반적인 Canvas 와 다를 것이 없기 때문에, 출력의 미리보기를 매우 쉽게 구현할 수 있다. 그러면, 실제로 GetDeviceCaps 함수를 사용해서 작업을 시작해보자. 먼저 각각의 축에 대한 함수를 다음과 같이 호출해야 한다.

```
XPPI := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
```

```
YPPI := GetDeviceCaps(Printer.Handle, LOGPIXELSY);
```

이 값은 프린터의 종류에 따라 다르지만 보통 300~360 정도가 될 것이다. 이 값을 이렇게 변수에 저장해 두었다가, 그리고자 하는 대상을 정확한 크기와 위치에 그릴 수 있다. 다음 코드는 2 인치 크기의 원 하나를 오른쪽으로 2 인치, 아래로 2 인치 떨어진 곳에 그리는 것이다.

```
Printer.BeginDoc;
```

```
XRRadius := XPPI div 2;
```

```
YRadius := YPPI div 2;
```

```
XCenter := XPPI * 2 - 1;
```

```
YCenter := YPPI * 2 - 1;
```

```
Printer.Canvas.Ellipse(XCenter - XRRadius, YCenter - YRadius,
```

```
    XCenter + XRRadius, YCenter + YRadius);
```

```
Printer.EndDoc;
```

(0, 0)은 사실 종이의 모서리에서 다소 떨어진 지점이다. 이때 원점이 (0, 0)이기 때문에 XCenter 나 YCenter 같은 위치를 계산할 때 1 씩 뺀 것이다.

그러면, 밀리미터 단위로 계산을 하려면 어떻게 해야 할까? 1 인치는 25.4 밀리미터이기 때문에, 1 밀리미터는 XPPI, YPPI 의 25.4 분의 1 이다. 그렇다고, 이를 덧셈 25.4 로 나누면 오차가 나게 된다.

이 문제를 해결하려면 XPPI, YPPI 값은 유지하고, 필요할 때마다 스케일해서 사용하는 것이다. 예를 들어, 2cm 지름의 원을 오른쪽으로 10cm, 아래로 10cm 떨어진 곳에 그린다고

하면 다음과 같이 하면 된다.

```
Printer.BeginDoc;
Printer.Canvas.MoveTo(0, 0);
XRadius := MulDiv(XPPI, 100, 254);
YRadius := MulDiv(YPPI, 100, 254);
XCenter := MulDiv(XPPI, 1000, 254) - 1;
YCenter := MulDiv(YPPI, 1000, 254) - 1;
Printer.Canvas.Ellipse(XCenter - XRadius, YCenter - YRadius,
    XCenter + XRadius, YCenter + YRadius);
Printer.EndDoc;
```

참고로, MulDiv 루틴은 처음 두 개의 파라미터를 곱한 값을 마지막 파라미터의 값으로 나누는(정수형 나누기) 루틴이다. 즉, 100 을 곱하고 254 로 나눈 것은 인치를 밀리미터 단위로 변환하기 위한 것이다.

#### ● 메모 인쇄하기

앞에서 간단하게 WriteLn 과 TextOut 을 이용해서 인쇄하는 방법에 대해서 알아보았지만, 보다 쓸모 있게 메모 컴포넌트의 내용을 인쇄하는 프로시저를 소개한다. 이 프로시저는 원래 Peter Below([100113.1101@compuserve.com](mailto:100113.1101@compuserve.com))에 의해 작성된 프로시저를 바탕으로 하였다. TPrinter 를 이용해서 인쇄하는 방법을 배우기에는 꽤 좋은 예제라고 생각되어 소개하는 것이다.

이 프로시저는 메모의 내용을 메모의 폰트를 이용해서 선택된 프린트로 인쇄하는 역할을 한다. 출력은 수평/수직 방향으로 가운데 정렬을 할 수 있고, 여백은 디폴트 값을 사용하게 할 것이다. 텍스트가 한 페이지가 넘을 경우에는 수직 방향 가운데 정렬은 무시된다.

먼저 사용할 유닛의 uses 절에 Printers.pas 유닛을 추가한다. 그리고, 프로시저의 선언을 해보자. 프로시저의 이름은 PrintMemo 로 하고, 파라미터로 인쇄할 메모, 수평과 수직 방향에서 가운데 정렬을 할 지 여부를 결정하는 논리값을 사용하도록 한다.

```
procedure PrintMemo(aMemo: TMemo; centerVertical, centerHorizontal: Boolean);
```

로컬 변수로는 여백을 나타내는 4 개의 변수와 x, y 축 라인의 좌표를 다용도로 사용할 2 개의 변수와 줄의 최대 길이, 줄수, 줄의 높이를 대표한 변수를 사용한다.

```
var
```

```

topMargin, bottomMargin, leftMargin, rightMargin: Integer;
x, y: Integer;
maxLineLength: Integer;
linecounter: Integer;
lineheight : Integer;

```

begin

먼저 디폴트 여백으로 위, 아래로는 1 인치, 좌우로는 0.75 인치를 설정한다. 이때 앞에서도 설명한 GetDeviceCaps API 함수를 이용해서 인치 단위를 사용할 수 있도록 한다.

```

topMargin := GetDeviceCaps( Printer.Handle, LOGPIXELSY );
bottomMargin := Printer.PageHeight - topMargin;
leftMargin := GetDeviceCaps( Printer.handle, LOGPIXELSX ) * 3 div 4;
rightMargin := Printer.PageWidth - leftMargin;

```

이제 인쇄를 시작한다. 먼저 메모의 폰트를 Printer 의 캔버스 객체에 대입한다.

```

Printer.BeginDoc;
try
    Printer.Canvas.Font.PixelsPerInch :=
        GetDeviceCaps( Printer.Canvas.Handle, LOGPIXELSY );
    Printer.Canvas.Font:= aMemo.Font;

```

한 줄의 높이를 결정한다. 보통 'Ay'를 사용하는데, 이는 가장 높은 위치('A')와 낮은 위치('y')를 포함했기 때문이다.

```

lineHeight := Printer.Canvas.TextHeight('Ay');

```

centerHorizontal 파라미터가 설정됐으면, 일단 메모의 한 줄 전체를 검사해서 가장 긴 줄의 길이를 픽셀로 알아낸다. 이를 이용해서 좌측 여백을 결정하게 된다.

```

if centerHorizontal Then
begin
    maxLineLength := 0;
    for linecounter := 0 to aMemo.Lines.Count - 1 do

```

```

begin
  x := Printer.Canvas.TextWidth(aMemo.Lines[linecounter]);
  if x > maxLineLength then maxLineLength := x;
end;
x := leftMargin + (rightMargin - leftMargin - maxLineLength) div 2;
      //이 줄의 좌측 여백을 결정한다.
if x < leftMargin then
begin
      //줄의 폭이 용지의 폭을 넘는 경우다. 그냥 잘리게 된다.
end
else
  leftMargin := x;
end;

```

centerVertical 파라미터가 설정됐으면, 인쇄할 모든 줄의 공간을 일단 계산하고, 이를 topMargin 을 결정하는데 참고로 한다.

```

if centerVertical then
begin
  y := lineHeight * aMemo.Lines.Count;
  if y < (bottomMargin - topMargin) then
    topMargin := topMargin + (bottomMargin - topMargin - y) div 2;
      //y 가 더 큰 경우에는 페이지 전체 크기보다 내용이 더 많은 경우이다.
      이때에는 수직 방향의 정렬은 무시하고, 그냥 놔둔다.
end;

```

여백이 모두 계산 되었으므로, 실제로 인쇄를 하면 된다. 다음 코드에서 x, y 변수는 다음 줄의 시작하는 지점의 좌표를 담고 있다.

```

x:= leftMargin;
y:= topMargin;
for linecounter := 0 to aMemo.Lines.Count-1 do
begin
  Printer.Canvas.TextOut(x, y, aMemo.Lines[linecounter]);
  y := y + lineHeight;
  if y >= bottomMargin then

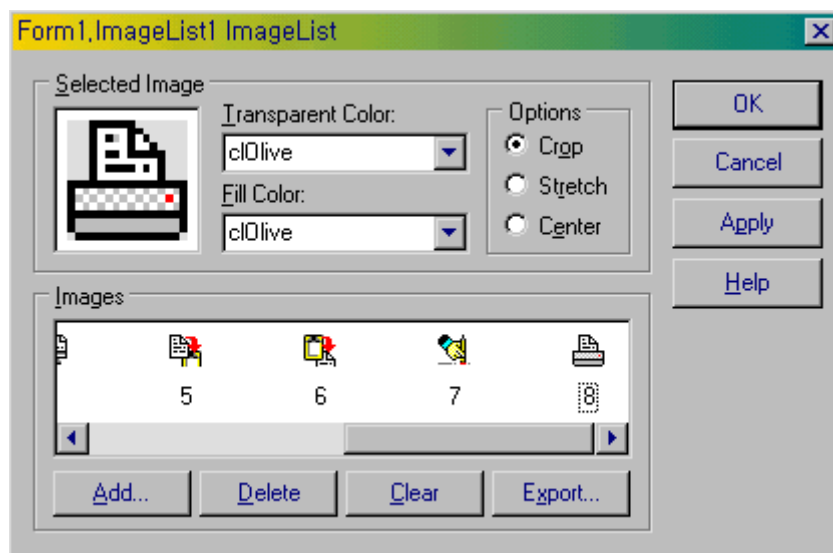
```

```

begin
    //페이지 끝이므로, 페이지를 넘긴다.
    if linecounter < (aMemo.Lines.Count - 1) then
        begin
            Printer.NewPage;
            y:= topMargin;
        end;
    end;
end;
finally
    Printer.EndDoc;
end;
end;

```

그럼, 이 프로시저를 이용해서 메모를 인쇄하는 간단한 어플리케이션을 제작해 보자. 여기에서는 8 장에서 제작했던 에디터에 인쇄기능을 추가하기로 하겠다. 먼저 8 장의 소스 디렉토리의 Exam3.dpr 파일을 읽어온 후, 유닛과 dpr 파일을 다른 이름으로 저장하자. 그리고, ImageList1 을 더블 클릭한 후 다음과 같이 인쇄 버튼에 해당되는 비트맵을 추가한다.

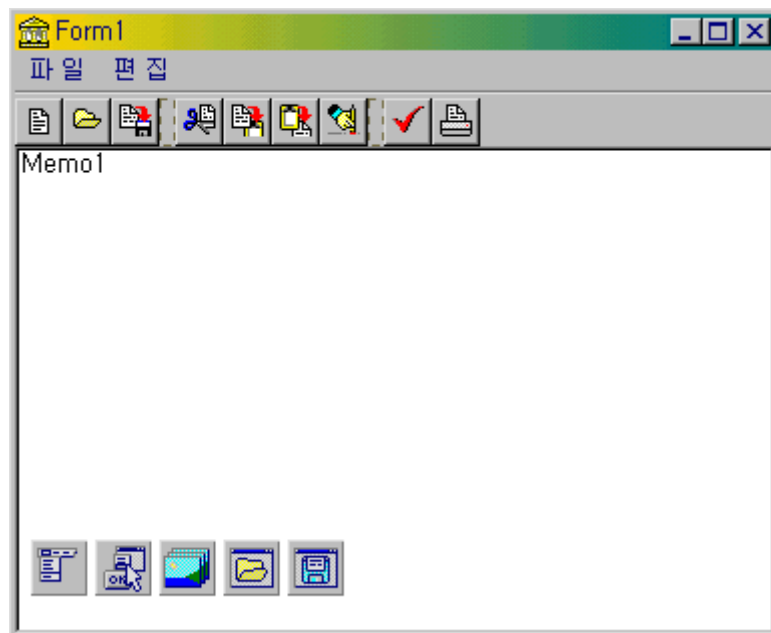


그리고, ActionList1 컴포넌트를 더블 클릭한 후, New Action 메뉴를 선택하여 액션 아이템을 하나 추가한다. 이 아이템의 Name 프로퍼티를 PrintFile, Caption 프로퍼티를 ‘인쇄하기’, 그리고 ImageIndex 프로퍼티를 8 로 설정한다. 또한, Toolbar1 컴포넌트를 클릭한 후 오른쪽 버튼을 눌러서 New Button 명령을 선택하면 새로운 툴바 버튼이 추가되는데, 이 버튼의 Action 프로퍼티를 PrintFile 로 설정한다. 마찬가지로 MainMenu1 컴포넌트를 더블

클릭한 후 ‘파일저장’과 ‘-’ 메뉴 아이템 사이에 커서를 위치시키고 Insert 키를 눌러서 메뉴 아이템을 하나 삽입한 후, 이 아이템의 Action 프로퍼티를 PrintFile 로 설정한다.

이렇게 하면 인쇄 명령을 위한 기본적인 인터페이스는 완료 되었다.

그리고 uses 절에 Printers 를 추가하고, 앞에서 작성한 PrintMemo 프로시저를 유닛의 implementation 섹션의 처음에 위치시킨다. 이 프로시저의 위치가 액션 아이템의 OnClick 이벤트 핸들러의 위치보다 앞에 있어야 하는데 주의한다. 물론 interface 섹션에 PrintMemo 프로시저의 선언부를 적어주면, 구현부분은 어디에 위치해도 상관없다. 완성된 폼의 형태는 다음과 같다.



마지막으로 PrintFile 액션 아이템의 OnExecute 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.PrintFileExecute(Sender: TObject);
begin
    PrintMemo(Memo1, True, True);
end;
```

이제 간이 에디터에 인쇄 기능이 추가되었다. 필요한 파일을 불러서 인쇄를 해보길 바란다. 이 어플리케이션은 9 장의 Exam4.dpr 파일을 로드하면 실행할 수 있을 것이다.

- 프린터 제어 코드 직접 보내기

프린터를 이용해서 도스에서 작업을 많이 한 경우, 프린터에 직접 제어 코드를 보내는 경우

가 있다. 이러한 프린터 제어 코드는 보통 Esc 키를 시작으로 나가게 되는데, 일반적인 방법으로는 윈도우 프로그램에서 이를 직접 사용할 수 없다.

그냥 생각하기에는 WriteLn 을 사용하면 될 것 같지만 제대로 동작하지 않는다. 여기서 이를 해결하기 위한 방법을 간단하게 소개한다.

먼저 TPrinter 를 사용해야 하므로, 제어 코드를 보내려고 하는 유닛의 uses 절에 Printers.pas 를 추가한다.

그리고, 다음과 같이 직접 프린터로 데이터를 보낼 수 있도록 레코드를 하나 선언한다.

```
type
```

```
TPassThroughData = Record  
    nLen : Word;  
    Data : array[0..255] of Byte;  
end;
```

이 레코드에는 제어 코드로 보낼 데이터를 Data 필드에 저장하게 된다.

그리고, 실제로 데이터를 프린터로 보내는 프로시저를 다음과 같이 작성한다.

```
procedure DirectPrint(s: string);  
var  
    PTBlock : TPassThroughData;  
begin  
    PTBlock.nLen := Length(s);  
    StrPCopy(@PTBlock.Data, s);  
    Escape(Printer.Handle, PASSTHROUGH, 0, @PTBlock, nil);  
end;
```

이 프로시저는 필자가 인터넷의 뉴스 그룹에서 발견한 것인데, 정리를 해 두었던 것으로 David Block([dblock@vdn.com](mailto:dblock@vdn.com))이란 사람이 소개한 것이다.

이 프로시저에서 핵심이 되는 부분은 Escape 라는 API 함수이다. 이 함수는 어플리케이션이 특정 디바이스에 GDI 를 거치지 않고 접근할 수 있도록 허용하는 역할을 하며, 지정된 드라이버에 데이터가 직접 날아간다.

WriteLn 으로 제어 코드를 날릴 때 이것이 작동하지 않는 이유는 언제나 이 문자를 GDI 가 중간에서 가로채기 때문이기 때문이므로, 이 함수를 사용하면 제어 코드를 날릴 수 있는 것이다. 이 함수의 첫번째 파라미터에는 디바이스 컨텍스트의 핸들을 지정한다. 여기서는 당연히 TPrinter 객체의 Handle 을 지정하면 된다. 그리고, 두번째 파라미터에 실행될 제어 코드 함수를 지정한다. 직접 제어코드를 날릴 때에는 PASSTHROUGH 를 지정하면 된다.



세번째 파라미터에는 날아갈 데이터의 크기를 지정하는데 0 을 지정하면 네번째 파라미터의 레코드 데이터가 모두 날아간다. 네번째 파라미터에 실제 데이터가 담긴 레코드의 포인터를 넘긴다. 마지막 파라미터에는 제어 코드를 날린 뒤에 돌아올 데이터를 담을 구조체의 포인터를 지정하면 되는데, 돌아올 데이터가 없을 때에는 nil 을 지정하면 된다.

사용법은 간단하다. 다음과 같이 DirectPrint 프로시저를 이용해서 제어 코드를 써 주면 된다.

```
procedure PrintTest;
begin
    Printer.BeginDoc;
    DirectPrint(CHR(27) + '&I1O' + 'Hello, World!');
    Printer.EndDoc;
end;
```

이렇게 DirectPrint 를 호출할 때에도 TPrinter 의 BeginDoc 과 EndDoc 메소드를 호출해야 하는데, BeginDoc 을 호출하면 프린터 드라이버가 프린터를 초기화하게 되며, EndDoc 메소드를 호출하면 프린터 드라이버가 페이지를 출력하게 된다.

참고로, 프린터 드라이버에 따라서는 PASSTHROUGH 를 지원하지 않을 수도 있다. 이를 알아보기 위해서는 다음과 같이 QUERYESCSUPPORT 를 이용하면 된다.

```
var
    TestInt: Integer;
begin
    TestInt := PASSTHROUGH;
    if Escape(Printer.Handle, QUERYESCSUPPORT, SizeOf(TestInt), @TestInt, nil) > 0 then
    begin
        ... (후략)
    end;
```

## 정 리 (Summary)

이번 장에서는 어플리케이션을 작성할 때 파일을 다루는 여러가지 테크닉과 델파이에서 제공하는 TPrinter 를 이용하여 인쇄를 하는 방법을 알아 보았다. 비교적 자주 쓰이면서도 잘 모르는 것이 파일의 입출력과 인쇄에 대한 방법 들이다.

지면 관계상 깊은 곳까지 다루지 못한 것이 아쉽지만, 델파이용으로 공개된 많은 VCL 소스를 살펴 보면 많은 것을 알 수 있을 것이다.

다음 장에서는 델파이를 이용해서 그래픽과 멀티 미디어 기능을 구현하고, 사용하는 방법에

대해서 다를 것이다.

# 그래픽, 멀티미디어 어플리케이션 제작

## (Creating Graphic and Multimedia Application)

텔파이를 이용해서 어플리케이션에 그래픽 기능을 추가하거나, 멀티 미디어를 지원하게 하는 것은 그다지 어려운 일이 아니다. 간단하게 미리 그려진 그림을 디자인 시에 추가할 수도 있고, 여러가지 그래픽 컨트롤을 추가하거나 런타임에서 동적으로 그리는 등의 방법이 모두 가능하다.

또한, 텔파이 4 에서 멀티미디어 컴포넌트를 사용하면 어플리케이션에서 동영상이나 각종 사운드 등을 지원하게 할 수 있다.

이번 장에서는 각종 어플리케이션을 제작할 때 화려함을 더해줄 수 있는 그래픽과 멀티 미디어를 지원하게 하는 방법에 대해서 알아보도록 한다.

### 그래픽 프로그래밍 개괄

VCL 그래픽 컴포넌트는 윈도우의 GDI(Graphics Device Interface)를 캡슐화한다. 텔파이 어플리케이션에서 그림을 그리려면, 객체의 캔버스를 이용한다. 캔버스는 객체의 프로퍼티로 제공되지만, 그 자체가 객체이다. 캔버스 객체의 가장 큰 장점은 리소스를 효과적으로 사용하며, 디바이스 컨텍스트를 관리하기 때문에 스크린, 프린터나 비트맵, 메타 파일 등의 종류에 관계 없이 같은 방법으로 사용할 수 있다는 것이다.

캔버스는 런타임에만 사용할 수 있다. 또한, TCanvas 객체가 윈도우의 디바이스 컨텍스트의 wrapper 이므로, 캔버스에 대해 윈도우 GDI 함수를 사용할 때에는 캔버스 객체의 Handle 프로퍼티를 이용해서 디바이스 컨텍스트 핸들을 얻어야 한다.

#### ● 디바이스 컨텍스트의 이해

윈도우의 그래픽에 대해 배울 때 가장 먼저 알아야 할 것이 디바이스 컨텍스트이다. 모든 윈도우 응용 프로그램들이 실제 장치 대신에 가상 화면과 가상 프린터를 사용한다. 윈도우는 본질적으로 실제 디스플레이 하기 전에, 내부적으로 그릴 것을 그리게 되는데 이때 사용하는 것이 디바이스 컨텍스트이다.

따라서 화면이나 프린터에 그리기를 원하는 어떤 것과 디바이스 컨텍스트를 분리할 필요가 있다. 디바이스 컨텍스트는 단순한 메모리라기 보다는 객체이다. 이 등록정보를 변경하면 다양한 특수 효과를 줄 수도 있다.

윈도우는 디바이스 컨텍스트를 가지고 직접 작업하는 것을 허용하지 않는다. 이를 처리하기 위해서는 API 함수를 호출해야 한다. 이때 컨텍스트 핸들에 대한 변수로 hDC 를 사용

한다.

델파이는 미리 정의된 디바이스 컨텍스트의 wrapper 를 제공한다. 이것이 바로 캔버스이다. 윈도우의 디바이스 컨텍스트의 유형에는 다음과 같은 4 가지 종류가 있다.

#### 1. 디스플레이 (Display)

보통 윈도우나 TMemo 컨트롤과 같이 그릴 수 있는 다른 영역에 부착되어 있다. 이 컨텍스트에 그리면 항상 정보가 화면에 전달된다. 디스플레이 디바이스 컨텍스트에는 Class, Common, Private 형의 3 가지 종류가 있다. 32 비트 응용 프로그램에서는 항상 private 형을 사용하면 된다. 만약 빨리 그래프나 차트를 출력하고자 할 때에는 common 형을 사용할 수 있다. Common 디바이스 컨텍스트를 얻을 때에는 GetDC 나 GetDCEx, BeginPaint 함수 등을 사용하면 된다. 물론 이때 받은 핸들은 빨리 사용하고 반납해야 한다. Private 디바이스 컨텍스트는 실제로 응용 프로그램이 소유하고 있는 컨텍스트이다. 이것의 장점은 윈도우에 돌려주기 전에 마음대로 그릴 수 있다는 점이다. 캐드나 그래픽 응용 프로그램의 경우에는 이 유형의 디스플레이 디바이스 컨텍스트를 사용한다.

#### 2. 프린터 (Printer)

디스플레이 디바이스 컨텍스트와 같이 프린터 디바이스 컨텍스트에 보내는 것도 당장 인쇄된다.

#### 3. 메모리 (Memory)

디스플레이나 프린터 디바이스 컨텍스트에 보낼 수 있는 것은 메모리에도 보낼 수 있다. 메모리 디바이스 컨텍스트는 CreateCompatibleDC 함수를 사용하여 만든다. 인쇄나 지금 디스플레이에 나타낼 때, 또는 나중에 출력할 때 메모리 디바이스 컨텍스트가 사용되고 있으며, 비트맵과 같은 것에도 이 디바이스 컨텍스트 유형을 사용할 수 있다.

#### 4. 정보 (Information)

이 디바이스 컨텍스트는 디스플레이의 경우에는 그다지 문제가 되지 않지만, 프린터의 경우에는 중요한 역할을 한다. 예를 들어 프린터에 인쇄할 컬러 문서가 있는 경우, 프린터가 컬러 출력을 지원하는지 알면 도움이 될 것이다. 만약 흑백 프린터의 경우에는 특별한 디더링 루틴을 추가하는 것이 필요하다. 정보 디바이스 컨텍스트는 CreateIC 함수를 이용하여 만든다. 디바이스 컨텍스트를 만들었으면, GetCurrentObject 나 GetObject 함수를 이용하여 특정 객체에 대한 정보를 얻을 수 있다.

## ● 윈도우의 그래픽 객체

윈도우는 7 개의 서로 다른 그래픽 객체를 가지고 있다. 이들은 윈도우의 다른 객체 들을 만들 때 사용되는 기본 객체이다. 이들 객체에 대해서 알아보자.

### 1. 비트맵 (Bitmap)

비트맵은 특정 유형의 래스터 그림이다. 예를 들어 아이콘과 PCX 파일은 물론 BMP 파일도 이 범주에 속한다. 비트맵 객체는 바이트 단위의 크기, 픽셀 단위의 차원, 컬러 포맷, 압축 스키마 등의 정보를 가진다. 비트맵 객체는 자신의 특정 유형에 기초한 다른 속성들도 다양하게 가지고 있다.

### 2. 브러쉬 (Brush)

브러쉬 색상을 적용할 때 사용된다. 벽에 칠을 할 때 붓을 사용하는 것과 같이 윈도우는 다각형이나 패스와 같은 다른 그래픽 객체의 내부를 칠할 때 브러쉬를 사용하고 있다. 특정 패턴으로 내부를 채우고자 할 때에는 비트맵을 브러쉬로 사용할 수도 있다.

### 3. 팔레트 (Palette)

윈도우는 풍부한 색상을 정의하여 사용하는 색상의 집합인 팔레트를 사용한다. 그런데, 일부 비디오 카드는 한번에 256 색상만 나타낼 수 있다. 이것이 하나 이상의 팔레트를 사용하는 이유로 서로 다른 256 색상이 필요할 때마다 서로 다른 팔레트를 사용하면 깨끗하고 견고한 색상을 화면에 나타낼 수 있다.

### 4. 폰트 (Font)

글자를 나타내는 폰트 역시 마찬가지로 그래픽 객체이다. 적절한 크기와 서체를 사용하여야 좋은 어플리케이션을 만들 수 있다.

### 5. 패스 (Path)

다각형이나 호, 선, 그리고 타원과 같은 그려진 객체의 특정 유형을 의미한다. 윈도우는 수많은 서로 다른 패스에 대한 함수를 가지고 있으며, 이들은 모두 서로 다른 속성들을 가지고 있다.

## 6. 펜 (Pen)

선을 그리거나, 도형을 그릴 때 사용하는 객체로 라인의 두께와 스타일과 같은 것들을 설정하게 된다.

## 7. 지역 (Region)

지역은 디바이스 컨텍스트에 있는 한 영역의 위치를 나타낸다. 때때로 캔버스 전체가 아니라 일정 영역에 대해서만 윈도우가 작업하도록 요청해야 할 때가 있다. 지역 객체는 이와 같은 일을 할 수 있게 해 주는 역할을 한다.

### ● 윈도우의 그래픽 모드

디바이스 컨텍스트를 사용할 때에, 디바이스 컨텍스트의 유형을 명시하면 그래픽 객체를 담을 컨테이너를 정의하는 셈이다. 그러면, 이 객체들이 어떻게 작용할 것인지를 결정해야 하는데, 이를 결정하는 것이 그래픽 모드이다. 그래픽 모드는 윈도우에게 디바이스 컨텍스트의 컨테이너에 있는 객체들이 서로 어떻게 동작할 것인지를 지정한다.

이들이 동작하는 방법은 대개 특정 유형의 함수를 사용할 때 어떤 종류의 화면 효과를 얻을 것인가를 결정한다. 이는 브러쉬를 사용하여 특정 영역을 채울 때 영향이 크다.

윈도우의 그래픽 모드에는 다음의 5 가지가 있으며, 이들을 동시에 사용한다.

### 1. 배경 (Background)

윈도우가 배경색을 섞는 방법을 정의한다. 윈도우가 한 색상을 다른 색상으로 대체해야 하는지, 또는 두 색상을 합쳐서 새로운 색상을 만들어야 하는지를 나타낸다. 텍스트와 비트맵 동작에 있어서 보통 이 모드를 사용하게 된다.

### 2. 드로잉 (Drawing)

이 모드는 전경색(foreground color)을 같이 섞는 방법을 윈도우에게 알려주는 역할을 한다. 이 모드는 보통 펜이나 브러쉬, 텍스트, 그리고 비트맵 동작을 사용할 때 사용된다.

### 3. 매핑 (Mapping)

매핑은 윈도우에게 크기 조정을 어떻게 해결할 것인지를 알려주는 역할을 한다. 예를 들어,

이론적으로 가능한 커다란 공간을 그래픽으로 할 때, 시스템에 있는 메모리 공간에 의해 이를 모두 나타내게 할 수 없을 것이다. 프린터가 디스플레이에 이를 어떻게 보여줄 것인가를 결정하는 역할을 하는 것이 매핑이다.

#### 4. 다각형 채우기 (Polygon-Fill)

이 모드는 윈도우가 다각형과 다른 도형을 채우는 방법을 정의한다. 본질적으로 윈도우가 그릴 때 사용할 도형의 모양과 브러쉬의 크기를 결정하는 역할을 한다.

#### 5. 스트레칭 (Stretching)

이미지를 압축하면, 이미지의 세세한 내용의 일부를 잃어버리게 된다. 이미지의 질을 가능한 많이 유지하려면 윈도우가 색상들을 섞는 방법을 알아야 한다.

#### ● 캔버스의 공통적인 프로퍼티와 메소드

캔버스 객체에서 공통적으로 사용하는 프로퍼티를 나열하면 다음과 같다.

프로퍼티	설 명
Font	이미지에 텍스트를 기록할 때 사용할 폰트를 지정한다. TFont 객체를 설정한다.
Brush	그래픽 모양과 배경을 채울 색깔과 패턴을 결정한다. TBrush 객체를 설정한다.
Pen	캔버스가 그림을 그릴 때 사용할 라인의 펜 종류를 결정한다. TPen 객체를 설정한다.
PenPos	펜의 현재의 위치를 지정한다.
Pixels	현재의 ClipRect 내의 픽셀 영역의 색깔을 지정한다.

캔버스 객체에서 사용하는 메소드에는 다음과 같은 것들이 있다. 이들에 대해서 모두 자세하게 설명할 수는 없으므로, 간단히 설명한다. 이 중에서 CopyRect, Ellipse, RectAngle, MoveTo, TextOut, TextWidth, TextHeight 는 이미 9 장에서 사용한 바 있다.

메소드	설 명
Arc	지정된 사각영역의 타원을 따라서 호를 그린다.
Chord	타원이 라인으로 절단된 달힌 그림을 그린다.
CopyRect	다른 캔버스의 영역에서 이미지를 복사한다.
Draw	캔버스의 Graphic 파라미터로 지정된 그래픽 객체를 지정한 위치에 그린다.
Ellipse	지정된 사각영역을 따라서 타원을 그린다.
FillRect	지정된 사각영역을 현재의 브러쉬로 채운다.

FloodFill	캔버스 전체 영역을 현재의 브러쉬로 채운다.
FrameRect	캔버스의 브러쉬를 이용하여 사각형을 그린다.
LineTo	PenPos 의 위치에서 지정된 위치까지 라인을 그린다.
MoveTo	현재의 위치를 지정된 위치로 옮긴다.
Pie	타원에서 지정된 사각영역으로 경계된 부분을 파이 형태로 그린다.
Polygon	파라미터로 넘긴 점들을 연결하여 다각형을 만든다. 처음 점과 마지막 점이 연결된다.
PolyLine	현재의 펜으로 Points 에 넘어온 점들을 서로 연결한다.
Rectangle	좌측 상단점과 우측 하단점을 대각선으로 하는 사각형을 펜을 이용해서 그리고, 내부를 브러쉬로 채운다.
RoundRect	코너가 둥근 사각형을 그린다.
StretchDraw	캔버스에 그린 그래픽을 지정된 사각영역에 맞춘다. 경우에 따라서 확대되거나 상하 좌우 비율이 변경된다.
TextHeight, TextWidth	각각 현재 폰트로 문자열을 쓸 때의 높이와 폭을 반환한다. 높이에는 줄 사이의 여백이 포함된다.
TextOut	문자열을 지정된 위치에 첫자부터 출력한다. 그리고, PenPos 프로퍼티는 문자열의 끝부분의 위치로 업데이트 된다.
TextRect	영역안에 문자열을 기록한다. 영역 바깥으로 나가는 문자열은 보이지 않게 된다.

그래픽 작업을 할 때에는 드로잉(drawing)과 페인팅(painting)이란 용어를 구별해서 사용해야 한다.

드로잉은 특정 그래픽 요소를 생성하는 것이다. 예를 들어, 라인이나 특정 형태를 코드를 이용해서 그리는 것이다. 보통 앞에서 설명한 캔버스의 드로잉 메소드를 호출해서 사용한다. 이미지가 저장되지 않기 때문에, 그 내용의 전부 또는 일부를 잃을 수 있으며, 출력도 이미지가 저장되지 않고, 어플리케이션은 어떻게 다시 그리는 지를 알지 못하므로 변할 수 있다.

그에 비해 페인팅은 객체 전체의 형태를 생성하는 것으로, 드로잉을 포함한다. 즉, 어떤 상황에서든 어플리케이션이 그 전체 화면을 다시 칠할 수 있도록 하는 것이다. 사실 사용자가 마우스 버튼을 누르거나 다른 어떤 동작을 취하면 그 위치와 다른 요소들을 저장해야 하고, 페인팅 메소드에서 이 정보를 실제로 해당 이미지를 페인팅하기 위해 사용한다. .

## ● 스크린의 리프레쉬

윈도우는 스크린 위의 객체의 형태가 변경되거나, 리프레쉬할 필요가 있을 때에는 WM\_PAINT 메시지를 생성해낸다. 이 메시지는 VCL 에서 OnPaint 이벤트로 표현된다. 그러므로, VCL 객체가 Refresh 메소드를 호출하면 언제나 OnPaint 이벤트가 발생한다.



폼에서 Refresh 메소드를 사용하면 지정된 형태로 그래픽을 다시 그리게 된다. 그러므로, 예를 들어 폼의 OnResize 이벤트 핸들러에서 폼의 형태를 변경하는 코드를 집어 넣은 경우에는 Refresh 메소드를 호출해서 변경된 사항을 반영하도록 해야 한다.

일부 운영체제에서는 윈도우의 클라이언트 영역의 일부가 무효화(invalidated)된 경우 자동으로 그 부분을 그려주지만, 윈도우는 그렇지 않다. 윈도우 운영체제에서는 일단 스크린에 그려진 것은 영구적이다. 그러므로, 폼이나 컨트롤이 드래그 등의 작업에 의해 잠시 가려지는 경우 폼이나 컨트롤은 반드시 불명확해진 영역을 다시 페인트 해야 한다.

TImage 컨트롤을 사용한다면, TImage 내부에 있는 그래픽의 페인팅과 refreshing 은 VCL 에 의해 자동으로 관리된다. 또한 TImage 에 드로잉을 한 경우에는 이것이 지속적인(persistent) 이미지이기 때문에, 포함된 이미지를 다시 그려줄 필요도 없다. 이와는 반대로 TPaintBox 의 캔버스는 스크린 디바이스에 직접 연결되어 있기 때문에, PaintBox 에 그려진 모든 것은 임시로 저장된다. 대부분의 경우 TPaintBox 와 거의 동일하다.

그러므로, TPaintBox 에 그리거나 페인팅을 한 경우에는 OnPaint 이벤트 핸들러에 클라이언트 영역이 무효화될 때마다 이를 다시 그려주는 코드를 추가할 필요가 있다.

그래픽 이미지가 어플리케이션에 나타나는 형태는 그리는 방법에 따라 틀리다. 만약 TBitmap 캔버스처럼 오프 스크린 이미지를 그리는 경우에는 컨트롤이 컨트롤의 캔버스에 비트맵을 복사하기 전에는 나타나지 않는다. 그러므로, 비트맵을 그리고 이것을 이미지 컨트롤에 대입하려면, 이미지는 컨트롤이 OnPaint 메시지를 처리할 기회가 있을 때에 보여줄 수 있다.

화면의 Refresh 와 연관된 메소드에는 Invalidate, Update, Refresh(Repaint)의 3 가지가 있다. 이들의 특징은 각각 다음과 같다.

## 1. Invalidate 메소드

이 메소드는 윈도우에게 폼의 전체 표면이 페인팅되어야 한다는 것을 알려준다. 그런데, Invalidate 는 페인팅 동작을 즉각 강제하지 않고 윈도우가 이 요구사항을 저장하고나서, 현재 프로시저가 완전히 수행되고 시스템에 다른 이벤트가 남아 있지 않을 경우에 여기에 반응하게 된다. 때때로 이 지연 시간 때문에, 페인팅 작업에 여러 변경이 일어난 후에야 비로소 폼이 페인팅되는 수가 있다. 그렇기 때문에, 느린 페인트 메소드가 여러 번 호출되서 수행속도를 더욱 느리게 하는 것을 막을 수 있다.

## 2. Update 메소드

윈도우에게 폼의 내용을 갱신할 것인지를 물어 보아서 그것을 곧장 페인팅하는 메소드이다. 이 메소드는 무효 영역(invalidated area)이 있을 경우에만 동작한다. 그러므로, 이 동작은 Invalidate 메소드가 막 호출되었을 때 일어나거나 아니면 사용자에게 의한 동작의 결과로서

일어날 수 있다. 만약 무효 영역이 없다면 아무런 동작을 하지 않으므로, 보통 Invalidate 를 호출한 뒤에 바로 Update 메소드를 호출한다.

### 3. Refresh(Repaint) 메소드

Invalidate 와 Update 메소드를 차례로 호출한다. 그 결과로 이 메소드는 OnPaint 이벤트를 즉각 동작시킨다.

이 메소드 들은 잘 사용해야 한다. 컨트롤을 반복 페이팅을 요청해야 할 경우에는 Invalidate 를 호출하는 것이 좋다. 이는 윈도우가 화면을 갱신하는데 너무 많은 시간을 소비하면 이 호출들을 한데 모아서 한 번에 처리할 수도 있기 때문이다. 윈도우의 WM\_PAINT 메시지는 낮은 우선 순위 메시지이기 때문에, 이것이 가능하고 더 효과적이다. 반면에, Refresh 를 여러 번 호출했을 경우에는 윈도우가 매번 다른 메시지만 처리할 수는 없으니 화면이 다시 그려져야 하고, 페인팅 작업이 느리기 때문에 어플리케이션 전체의 성능을 떨어뜨릴 수 있다. 그렇지만, 가능한 빠르게 화면을 다시 그려야 하는 때가 있는데 이런 경우에는 Refresh 를 호출하는 것이 좋다.

#### ● 그래픽 객체의 종류

VCL 은 다음과 같은 그래픽 객체를 제공한다.

객 체	설 명
Picture	그래픽 이미지를 담을 수 있다. 만약에 추가적인 그래픽 파일 포맷을 추가하려면 RegisterFileFormat 메소드를 사용한다. 그래픽 파일을 보여준다.
Bitmap	이미지를 생성, 관리, 저장하는데 사용되는 강력한 그래픽 객체
Clipboard	어플리케이션에 cut, copy, paste 를 지원하게 될 텍스트나 그래픽에 대한 컨테이너를 나타낸다. 적절한 포맷 관리, 참조 계수 관리 등을 한다.
Icon	윈도우 아이콘 파일을 다룬다.
Metafile	이미지를 다룰 때 실제 픽셀로서 구성되는 것이 아니라, 이미지를 구성하기 위한 여러 작업들을 기록하는 것이다. 메타 파일은 비트맵에 비해 메모리도 적게 차지하고, 이미지 손상이 적은 장점이 있으나, 처리 속도가 느리다.

### 캔버스 객체의 프로퍼티 활용

캔버스 객체에서 제공하는 여러가지 프로퍼티를 설정하여 사용하는 방법에 대해서 알아보자. 펜을 이용하여 라인을 그리고, 브러쉬를 이용하여 내부를 채우며, 적절한 폰트를 골라서 텍

스트를 기록하는 등의 작업이 캔버스 객체의 프로퍼티를 활용하여 이루어진다.

## ● 펜의 활용

캔버스의 Pen 프로퍼티는 라인의 형태를 결정한다. 선을 긋는 작업을 궁극적으로 생각해 보면 두 개의 점 사이에 있는 픽셀의 그룹을 바꾸는 것이다. 펜에 대해서 연상할 때 가장 쉽게 생각할 수 있는 것은 포토샵이나 페인트샵 프로와 같은 그래픽 프로그램의 도구 상자에서 여러가지 모양의 붓을 선택해서 선이나 도형을 그릴 수 있다는 것이다. 펜을 선택한다는 것은 이런 그래픽 프로그램에서 붓을 선택하는 것과 같은 것이다.

펜 자체에는 변경해서 사용할 수 있는 프로퍼티가 4 가지가 있다. Color, Width, Style, Mode 가 그것인데 이들 프로퍼티의 값들이 펜의 형태를 결정하게 된다. 디폴트로 모든 펜은 검정색, 두께는 1 픽셀이고 solid 형을 가지며, 모드는 캔버스에 있는 어떤 것이든 덧 씌워그리는 copy 모드로 설정되어 있다.

### 1. 펜 색상의 변경

펜의 색상을 변경하는 것은 다른 여러가지 컴포넌트의 Color 프로퍼티를 런타임에서 변경하는 것과 똑같다. 펜의 색상은 그려지는 라인의 색상을 결정하게 된다.

사용자에게 펜에 대한 새로운 색상을 결정하게 하려면, 보통 컬러 그리드를 띄워서 선택하게 한다. 컬러 그리드는 foreground 색상과 background 색상을 결정할 수 있게 구성되어 있다. 이때 foreground 색상은 보통 펜의 색상을 나타내며, background 색상은 브러쉬의 색상을 나타낸다고 생각하면 된다.

### 2. 펜의 두께 변경

펜의 두께는 그려지는 라인의 두께를 픽셀 단위로 결정하는 것이다. 펜의 두께가 1 보다 크면, 펜의 Style 프로퍼티 값에 관계 없이 윈도우 95 는 언제나 solid 라인을 그리게 된다. 펜의 두께를 변경하려면 펜의 Width 프로퍼티에 정수값을 대입하면 된다.

### 3. 펜의 스타일 변경

펜의 Style 프로퍼티는 선의 종류를 일반적인 선(solid), 점선(dotted line), 대쉬(dashed line) 등이 있다. 모두 6 개의 스타일이 존재하는데 이들은 각각 psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear 이다. 앞에서도 설명 했지만, 펜의 두께가 1 픽셀을 넘을 경우 이 프로퍼티의 값은 무시된다.

#### 4. 펜의 모드 변경

펜의 모드 프로퍼티는 펜의 색상과 캔버스의 색상을 결합하는 방법을 결정하는 것이다. 예를 들어, 펜은 언제나 검정색이게 할 수도 있고, 캔버스의 배경색의 보색으로 보이게 할 수도 있고, 펜 색상의 보색으로 보이게 할 수도 있다.

#### 5. 펜의 위치 얻기

펜이 다음 라인을 그릴 때 시작점이 되는 위치를 펜의 위치(position)라고 한다. 캔버스는 펜의 위치를 PenPos 프로퍼티에 저장한다. 펜의 위치는 라인을 그릴 때에만 영향을 미친다. 그러므로, 다른 도형이나 텍스트를 그리는 경우에는 그리려는 위치를 좌표로 전달해야 한다. 펜의 위치를 설정할 때에는 MoveTo 메소드를 사용한다. 예를 들어, 다음의 코드는 펜의 위치를 캔버스의 좌측 상단으로 이동한다.

```
Canvas.MoveTo(0, 0);
```

LineTo 메소드로 라인을 그릴 때, 라인의 끝점으로 현재의 위치를 이동시킨다.

### ● 브러시의 활용

캔버스 컨트롤의 Brush 프로퍼티는 도형의 내부를 채우는 방법을 결정한다. 브러시 객체에는 Color, Style, Bitmap 의 3 개의 프로퍼티를 이용해서 여러가지 작업을 하게 된다. 이들 프로퍼티의 디폴트 값은 백색, solid 스타일, 패턴은 없는 것이다.

#### 1. 브러시 색상의 변경

브러시의 Color 프로퍼티는 영역의 내부를 채울 색상을 결정한다. 보통 브러시는 배경색을 결정하는 것으로 이해하면 된다.

#### 2. 브러시 스타일의 변경

브러시 스타일은 캔버스에 채우는 방법을 결정한다. 스타일에 따라 브러시 색상과 캔버스의 색상을 결합하는 방법을 결정한다. 미리 지정된 스타일에는 solid, clear 와 여러가지 라인과 해치 패턴 등을 결정할 수 있다.

Style 프로퍼티에는 bsSolid, bsClear, bsHorizontal, bsVertical, bsFDDiagonal, bsBDDiagonal, bsCross, bsDiagCross 등의 값들이 있다.

### 3. 브러쉬 비트맵 프로퍼티의 설정

브러쉬의 Bitmap 프로퍼티는 브러쉬가 여기에 지정된 비트맵을 패턴으로 사용하여 영역을 채우게 된다. 다음의 코드는 비트맵을 파일에서 로드하여, 이를 브러쉬의 패턴으로 사용하는 예이다.

```
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: 브러쉬 비트맵

브러쉬의 비트맵은 8x8 픽셀의 비트맵 패턴이어야 한다. 브러쉬에 비트맵을 사용하기 위해서는 우선 비트맵을 생성하고, 이것을 지정하여 사용한 후, 모든 작업이 완료되었으면 이를 해제시켜 주어야 한다.

## 그래픽 객체 그리기

### ● 라인 그리기

캔버스에서 그릴 수 있는 라인은 일반적인 라인과 폴리 라인(polyline)이 있다. 라인은 두 개의 점을 이어주는 픽셀 들이다. 폴리 라인은 라인이 서로 연결된 것으로, 이들을 그릴 때에 캔버스는 펜을 이용하게 된다.

#### 1. 라인 그리기

캔버스에서 라인을 그리려면, LineTo 메소드를 사용한다. LineTo 메소드는 현재의 위치에서 지정한 좌표까지 현재의 펜으로 라인을 그린다. 그리고, 현재 위치가 끝점으로 이동하게 된다. 예를 들어, 다음의 코드는 폼이 그려질 때 폼에 라인으로 'X'자를 그리게 된다.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
    begin
        MoveTo(0, 0);
        LineTo(ClientWidth, ClientHeight);
        MoveTo(0, ClientHeight);
        LineTo(ClientWidth, 0);
    end;
end;
```

## 2. 폴리 라인 그리기

폴리 라인을 캔버스에 드리려면 PolyLine 메소드를 사용한다. PolyLine 메소드에서 사용하는 파라미터는 Point 의 배열이다. PolyLine 은 기능적으로는 MoveTo 와 LineTo 메소드를 계속해서 호출하는 것과 같지만 호출에 따른 오버헤드가 없기 때문에 수행 속도가 빠르다.

### ● 도형 그리기

캔버스에는 여러가지 종류의 도형을 그리는 메소드가 있다. 캔버스는 도형의 외곽선은 펜으로 그리고, 내부는 브러쉬로 채운다.

#### 1. 사각형과 타원 그리기

캔버스에 사각형과 타원을 그리기 위해서는 Rectangle, Ellipse 메소드를 사용하면 된다. 이 두 메소드 모두 좌표로는 경계가 되는 4 개의 좌표를 사용한다.

Rectangle 메소드는 넘어온 4 개의 좌표를 연결한 사각형을 그리며, Ellipse 메소드는 사각형내를 채우는 타원을 그린다.

다음의 코드는 좌측 상단의 1/4 영역에 사각형을 그리고, 내부에 타원을 그린다.

```

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;

```

이 코드를 실행하면 다음과 같은 실행 화면을 볼 수 있다.



## 2. 끝이 둥근 사각형 그리기

끝이 둥근 사각형을 그릴 때에는 RoundRect 메소드를 사용한다. RoundRect 메소드의 4개의 파라미터는 다른 메소드와 마찬가지로 사각형의 좌표를 나타내며, 이 밖에도 둥근 코너를 어떻게 그릴지를 나타내는 2개의 파라미터를 추가로 가진다.

다음의 메소드는 폼의 좌측 상단 1/4 영역에 10 픽셀의 지름의 원형 코너를 가진 끝이 둥근 사각형을 그리게 된다.

```

procedure TForm1.FormPaint(Sender: TObject);
begin
    Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;

```

## 3. 다각형 그리기

다각형을 그리기 위해서는 Polygon 메소드를 사용하면 된다. 다각형은 파라미터로 Point의 배열을 사용하며, 이들의 점이 꼭지점이 된다.

## 그래픽 컨트롤 사용하기

어플리케이션에 그래픽 객체를 다루기 위해서 특별히 다른 컴포넌트를 사용할 필요는 없다. 그렇지만, 사실상 폼에 직접 드로잉을 하는 경우는 거의 없다. 그 보다는 보통 VCL 이미지 컨트롤을 이용해서 그래픽을 보여주는 경우가 많다.

이미지 컨트롤은 비트맵 객체를 디스플레이할 수 있는 컨테이너 컴포넌트이다. 이렇게 이미지 컨트롤을 사용하면 인쇄, 클립보드, 그래픽 객체 읽기와 저장 등의 편리한 기능을 이용할 수 있다. 그래픽 객체는 비트맵 파일, 메타 파일, 아이콘 등의 여러가지 그래픽 클래스일 수 있다.

### ● 스크롤 가능한 그래픽 만들기

그래픽의 크기는 폼과 같은 크기일 필요는 없다. 그래픽의 종류에 따라서는 폼보다 더 크거나 작을 수 있다. 이럴 때 스크롤 박스 컨트롤(TScrollBox)을 폼에 추가하고, 그래픽 이미지를 그 안에 위치시키면 그래픽이 폼보다 훨씬 커도(스크린 전체 보다 커도) 스크롤 박스의 기능을 통해 전체를 볼 수 있다.

### ● 초기 비트맵 크기 설정

이미지 컨트롤을 폼위에 올려 놓으면, 처음에는 단순한 컨테이너이다. 실제로 그래픽을 사용하려면 Picture 프로퍼티를 설정해 주어야 한다. 그런데, 처음에 비트맵을 보여주지 않더라도, 자리를 차지한 컨트롤이 보이지 않으면 보기가 무척 싫을 것이다. 이를 피하기 위해서는 폼의 OnCreate 이벤트 핸들러에 비트맵 객체를 생성해서 Picture.Graphic 프로퍼티에 대입하면 된다.

다음은 어플리케이션의 폼에 초기 비트맵의 크기를 설정해서 대입하는 코드이다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
    Bitmap: TBitmap;
begin
    Bitmap := TBitmap.Create;
    Bitmap.Width := Image1.Width;
    Bitmap.Height := Image1.Height;
    Image1.Picture.Graphic := Bitmap;
end;
```



비트맵을 Picture.Graphic 프로퍼티에 대입하면, Picture 객체가 비트맵의 Owner 가 된다. 그러므로, Picture 객체가 파괴되면, 비트맵 객체도 자동으로 파괴된다. 어플리케이션을 실행하면 폼의 클라이언트 영역에 비트맵이 하얀 색으로 자리를 차지하고 있을 것이다.

## 그래픽 파일의 읽기와 쓰기

그래픽 이미지는 어플리케이션이 실행되는 동안에만 존재한다. 그렇기 때문에, 이를 파일로 저장하고, 불러오는 것은 필수적인 부분이다. 그래픽 이미지 컨트롤도 일반적인 다른 VCL 컴포넌트와 마찬가지로 이미지를 파일로 저장하고, 읽어올 수 있다. 또한, 이미지 컨트롤은 설치 가능한 그래픽 클래스를 지원한다.

### ● 파일에서 그림 읽어 오기

이미지 컨트롤에 그래픽을 로드하려면 Picture 객체의 LoadFromFile 메소드를 이용하면 된다. 9 장의 그래픽 인쇄 루틴에서 이미 한번 소개한 바 있으므로 이해가 어렵지는 않을 것이다. 다음의 코드는 OpenFileDialog 컴포넌트를 이용해서 그림 파일을 이미지 컨트롤에 불러오는 예이다.

```
procedure TForm1.Open1Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
    begin
        CurrentFile := OpenFileDialog1.FileName;
        Image.Picture.LoadFromFile(CurrentFile);
    end;
end;
```

### ● 그림을 파일로 저장하기

VCL 그림 객체는 몇 가지 포맷의 그래픽을 읽고, 쓸 수 있다. 그리고, 자신 만의 그래픽 파일 포맷을 생성하고, 등록시킬 수 있다. 이미지 컨트롤의 내용을 파일에 저장하려면 Picture 객체의 SaveToFile 메소드를 이용하면 된다.

다음의 이벤트 핸들러는 File|Save, File|Save As 메뉴를 선택했을 때 사용할 만한 코드이다. 참고로 하면, 쓸모가 있을 것이다. 이때 CurrentFile 변수는 문자열 형의 전역 변수로 선언해서 사용하면 된다.

```

procedure TForm1.Save1Click(Sender: TObject);
begin
    if CurrentFile <> '' then
        Image.Picture.SaveToFile(CurrentFile)
    else SaveAs1Click(Sender);
end;

procedure TForm1.SaveAs1Click(Sender: TObject);
begin
    if SaveDialog1.Execute then
        begin
            CurrentFile := SaveDialog1.FileName;
            Save1Click(Sender);
        end;
end;

```

## 그래픽에 클립보드 이용하기

이미지 컨트롤을 이용해서 윈도우 클립보드에 cut, copy, paste 기능을 이용하려면 VCL 의 클립보드 객체를 사용하면 된다. VCL 클립보드 객체를 사용하려면 사용하는 유닛의 uses 절에 Clipbrd.pas 유닛을 추가하여야 한다.

- 클립보드에 그래픽 복사, 잘라내기 (copy and cut)

그림을 클립보드에 복사하려면, Picture 객체를 클립보드 객체로 Assign 메소드를 사용해서 대입하면 된다. 그래픽을 클립보드에 잘라 넣는 것도 복사하는 것과 비슷한 방법을 사용하지만, 소스에서 그래픽을 삭제하게 된다.

그래픽을 클립보드에 잘라 넣으려면, 먼저 클립보드로 복사를 하고 원래의 내용을 삭제하면 된다. 삭제한 이미지를 어떻게 보여줄 것인가 하는 것도 비교적 중요한 점이다. 흔히 사용하는 방법으로는 지워진 영역을 하얗게 표시하는 방법이다. 다음의 코드는 클립보드로 복사하기와 잘라내기를 메뉴에서 선택했을 때 쓸 수 있는 코드이다.

```

procedure TForm1.Copy1Click(Sender: TObject);
begin
    Clipboard.Assign(Image1.Picture);

```

```
end;
```

```
procedure TForm1.Cut1Click(Sender: TObject);
```

```
var
```

```
    ARect: TRect;
```

```
begin
```

```
    Copy1Click(Sender);                                {클립보드로 일단 복사한다.}
```

```
    with Image1.Canvas do
```

```
    begin
```

```
        CopyMode := cmWhiteness;                        {복사 모드를 하얗게 설정}
```

```
        ARect := Rect(0, 0, Image1.Width, Image1.Height); {비트맵 크기를 설정}
```

```
        CopyRect(ARect, Image1.Canvas, ARect);          {비트맵을 복사해 넣는다.}
```

```
        CopyMode := cmSrcCopy;                          {원래의 복사 모드로 복구}
```

```
    end;
```

```
end;
```

#### ● 클립보드에서 그래픽 붙여 넣기 (Paste)

윈도우 클립보드에 비트맵 그래픽이 담겨 있으면, 이를 이미지 객체에 붙여넣을 수 있다. 이를 위해서는 먼저 클립보드의 HasFormat 메소드를 호출하여 클립보드가 그래픽을 가지고 있는지를 확인한다. HasFormat 메소드는 Boolean 값을 반환하는데, 지정한 데이터 형이 클립보드에 담겨 있으면 True 를 반환한다. 비트맵이 있는지 확인하려면 파라미터로 CF\_BITMAP 를 넘겨주면 된다. 비트맵이 있으면 클립보드에서 대입하면 된다. 다음의 코드는 붙여 넣기를 구현한 예제 코드이다.

```
procedure TForm1.PasteButtonClick(Sender: TObject);
```

```
var
```

```
    Bitmap: TBitmap;
```

```
begin
```

```
    if Clipboard.HasFormat(CF_BITMAP) then
```

```
    begin
```

```
        Image.Picture.Bitmap.Assign(Clipboard);
```

```
    end;
```

```
end;
```

## 어플리케이션에 드로잉 객체 이용하기

앞에서 설명한 다양한 드로잉 메소드는 툴바나 버튼 패널에서도 사용할 수 있다. 어플리케이션에 그래픽을 잘 사용하면 프로그램이 화려하며 고급스럽게 보이게 할 수 있지만, 너무 많은 색이나 너무 많은 그래픽을 사용할 경우에는 오히려 비생산적이 될 수도 있다. 그러므로, 적당한 수준에서 드로잉 객체를 잘 사용하는 것이 좋은 어플리케이션을 만드는데 상당히 중요한 역할을 하는 것이다.

## DIBs(Device Independent Bitmaps)의 지원

델파이 3 버전부터 TBitmap 객체에 비트맵 비트에 대한 포인터를 프로퍼티로 제공하기 시작했다. 델파이 2 까지는 DDBs(Device Dependent Bitmaps)를 사용했으나, 델파이 3 부터 DIBs 를 지원하게 된 것이다. DIB 비트에 대한 포인터를 얻으려면 Bitmap.DibMemory 프로퍼티에 접근하기만 하면 된다.

다음의 코드는 첫번째 스캔 라인의 픽셀 들을 팔레트의 첫번째 색상을 변경한다. 이 코드는 256 색상의 비트맵 팔레트를 기준으로 작성되었기 때문에, 다른 형태의 비트맵에는 동작하지 않을 수 있다.

type

TByteArray = array[0..0] of Byte;

procedure TForm1.Button1Click(Sender: TObject);

var

p: ^TByteArray;

i: Integer;

begin

Image1.Picture.LoadFromFile(c:\Windows\구름.bmp');

p := Image1.Picture.Bitmap.DibMemory;

for i := 0 to (Image1.Picture.Bitmap.Width - 1) do

p^[i] := 0;

end;

이 코드를 살펴 보면, 바이트 형의 배열을 선언하고 여기에 대한 포인터 변수 p 를 선언한다. 그리고, 비트맵을 디스크에서 읽어온 후, DibMemory 에 대한 포인터를 변수 p 에 대입하고, DIB 메모리의 첫번째 스캔 라인의 색상을 DIB 팔레트의 첫번째 색상으로 바꾼다.

## 라인을 마음대로 그리자 !

지금까지 설명한 내용을 바탕으로 간단하게 라인을 그어주는 어플리케이션을 하나 만들어 보자. 제작할 어플리케이션은 사용자가 마우스를 움직이면 이를 따라 런타임에서 라인을 그리는 간단한 드로잉 프로그램이다. 이 어플리케이션은 마우스로 클릭하고 드래그를 하면 윈도우의 캔버스에 라인을 그린다. 마우스 버튼을 누르면 그리기가 시작되고, 버튼을 떼면 그리기가 완료된다.

### ● 마우스에 반응하기

어플리케이션에서 사용하는 마우스의 동작에는 버튼을 누르고(MouseDown), 마우스를 움직이고(MouseMove), 버튼을 떼는(MouseUp) 동작과 클릭(Click)이 있다.

#### 1. 마우스 이벤트의 종류

VCL 에는 OnMouseDown, OnMouseMove, OnMouseUp 의 3 가지 마우스 이벤트가 있다. VCL 어플리케이션이 마우스의 동작을 감지하면 해당되는 이벤트 핸들러를 호출하게 되는데, 이들 이벤트는 모두 5 개의 파라미터를 사용한다. 이들 파라미터의 내용은 다음과 같다.

파라미터	의 미
Sender	마우스의 동작을 감지한 객체
Button	어떤 버튼의 동작인가 ? mbLeft, mbMiddle, mbRight 중의 하나이다.
Shift	Alt, Ctrl, Shift 키가 눌렸는가 ?
X, Y	이벤트가 발생한 좌표

#### 2.MouseDown 이벤트의 처리

사용자가 마우스의 버튼을 누를 때마다 OnMouseDown 이벤트가 발생한다. 참고로 라인을 긋는 경우를 생각해보자. 이때 마우스 버튼을 누르면, 펜의 위치가 이동하도록 해야 한다. 그러므로 다음과 유사한 이벤트 핸들러를 작성하여야 할 것이다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    Canvas.MoveTo(X, Y);  
end;
```

### 3. MouseUp 이벤트의 처리

OnMouseUp 이벤트는 사용자가 마우스 버튼을 땔 때 발생하게 된다. 이때 이 이벤트는 마우스를 누른 컨트롤에 발생하기 때문에, 커서의 위치가 컨트롤의 범위를 벗어나더라도 이를 처리할 수 있다. 그러므로, 라인을 폼의 범위를 벗어난 경우에도 처리를 할 수 있다. 어쨌든 앞에서의MouseDown 이벤트에 이어서 라인을 굿도록 하려면, OnMouseUp 이벤트 핸들러를 다음과 같이 작성하면 된다.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    Canvas.LineTo(X, Y);  
end;
```

이렇게 함으로써 사용자는 마우스 버튼을 클릭하고 드래그한 후, 이를 놓으면 라인을 그릴 수 있게 된다.

### 4. MouseMove 이벤트의 처리

OnMouseMove 이벤트는 사용자가 마우스를 움직일 때마다 주기적으로 발생한다. 다음의 예제 코드는 사용자가 마우스 버튼을 누른 채로 마우스를 움직일 때 라인을 그려주게 된다.

```
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    Canvas.LineTo(X, Y);  
end;
```

#### ● 마우스 동작의 기록

드로잉 어플리케이션을 작성하기 위해서는 마우스의 동작을 감시하는 것이 필요하다. 이를 위해서는 폼 객체에 여기에 대한 객체 필드를 추가하는 것이 좋다.

다음의 코드는 마우스 버튼의 동작을 기록하기 위해, 마우스 버튼이 눌러있는지 여부를 기록하는 Drawing 이라는 Boolean 필드를 추가하였다. 또한, 마우스의 위치를 기록하기 위해서 TPoint 형의 Orgin, MovePT 필드도 사용한다.

참고: TPoint 데이터 형

TPoint 데이터 형은 X 와 Y 의 값을 포인트(Point)라고 불리는 하나의 레코드에 저장한 것이다. 포인트를 생성하는 가장 쉬운 방법은 Point 함수를 사용하는 것이다. Point 함수는 X 와 Y 의 값을 받아들여 TPoint 라는 레코드를 반환하게 된다.

type

```
TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
public
    Drawing: Boolean;           //마우스 버튼이 눌렀는지 여부를 기록한다.
    Origin, MovePt: TPoint;    //포인트를 기록하는 필드
end;
```

사용자가 마우스 버튼을 누르면 Drawing 필드를 True 로, 버튼을 떼면 False 로 설정한다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Drawing := True;
    Canvas.MoveTo(X, Y);
end;
```

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Canvas.LineTo(X, Y);
    Drawing := False;
end;
```

OnMouseMove 이벤트 핸들러는 Drawing 프로퍼티가 True 일 때에만 동작하도록 수정한다.

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Drawing then //Drawing 플래그가 설정되었을 때만 그린다.  
        Canvas.LineTo(X, Y);  
end;
```

이렇게 하면, 마우스 버튼을 누르고 뗄 때까지 계속해서 라인을 이어서 그리게 된다.

#### ● 마우스 포인트의 추적

라인을 그릴 때, Origin 필드에는 MouseDown 이벤트가 발생한 위치를 기록하고 이를 이용해서 MouseUp 이벤트에서 라인을 그려주면, 마우스를 눌렀다가 뗄 때 하나의 라인이 그려질 것이다. 다음과 같이 구현하면 된다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    Drawing := True;  
    Canvas.MoveTo(X, Y);  
    Origin := Point(X, Y);  
end;
```

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    Canvas.MoveTo(Origin.X, Origin.Y);  
    Canvas.LineTo(X, Y);  
    Drawing := False;  
end;
```

이렇게 하면 라인을 제대로 그릴 수 있게 되지만, 마우스를 움직일 때의 라인을 볼 수가 없다. 이를 위해서는 마우스의 움직임을 점검해서 적절한 조치를 취할 필요가 있다.



- 마우스 움직임의 추적

현재의 OnMouseMove 이벤트 핸들러는 마지막 마우스의 위치를 따라서 라인을 그리기 때문에 문제가 있다. 이를 해결하기 위해서는 라인을 그리는 위치를 origin 포인트로 옮길 필요가 있다.

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Drawing then  
        begin  
            Canvas.MoveTo(Origin.X, Origin.Y);           //펜을 시작 포인트로 이동  
            Canvas.LineTo(X, Y);  
        end;  
end;
```

이렇게 하면, 현재의 마우스 위치를 따라서 라인을 그리게 되지만, 계속 라인을 겹쳐서 그리기 때문에 제대로 볼 수가 없다. 그러므로, 이를 시정하기 위해서는 다음 라인을 그리기 전에 원래의 라인을 지워야 한다. 이를 위해서는 이전에 사용한 점의 위치를 알아야 하는데, 이를 저장하기 위해 MovePt 필드를 사용한다.

MovePt 필드는 각각의 중간 라인의 끝점으로 설정되며, Origin 과 MovePt 를 연결하는 라인을 다음과 같이 그리면 이전의 라인을 지우게 된다.

```
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);  
end;  
  
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin
```

```

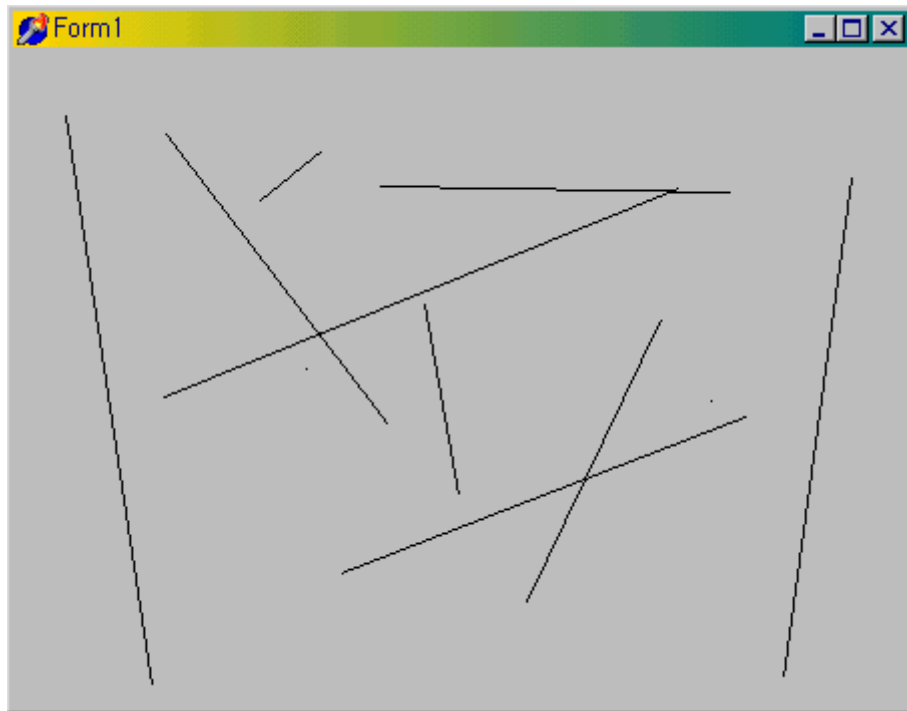
if Drawing then
begin
    Canvas.Pen.Mode := pmNotXor;           //그리고, 지우기 위해서 XOR 모드를 사용
    Canvas.MoveTo(Origin.X, Origin.Y);     //펜의 위치를 origin 으로
    Canvas.LineTo(MovePt.X, MovePt.Y);     //이전 라인을 지운다.
    Canvas.MoveTo(Origin.X, Origin.Y);     //펜의 위치를 origin 으로
    Canvas.LineTo(X, Y);                   //새로운 라인을 그린다.
end;
MovePt := Point(X, Y);                     //현재 위치를 기록한다.
Canvas.Pen.Mode := pmCopy;
end;

```

펜의 모드를 pmNotXor 로 선택하면, 라인이 배경색과 결합되어 나타나게 된다. 라인을 지우기 위해서는 현재 라인이 그려진 위치에 라인을 그리면 된다. 그리고 나면, 펜의 모드를 pmCopy(디폴트 값)로 바꾼다.

이제 완성이 되었다. 특별히 펜의 스타일이나 색상 등을 정하지 않고 디폴트 값을 사용하였으므로, 멋도 없는 매우 단순한 어플리케이션이지만 기본적으로 마우스를 사용한 드로잉 프로그램을 제작하는 방법에 대해서는 익혔을 것으로 믿는다.

이 어플리케이션을 실행하고 라인을 마음대로 그려보자. 그러면 다음과 같은 형태의 실행 화면을 얻을 수 있을 것이다.



## 스크린 캡처 어플리케이션의 제작

이번에는 간단한 스크린 캡처 어플리케이션을 제작해보자. 이번 어플리케이션에서 배울 점은 DC에 대한 개념과 비트맵을 처리할 때 많이 사용되는 API 함수를 익히는 것이다. 먼저 폼을 다음과 같이 디자인 한다. 캡처 프로그램의 폼은 그다지 화려하지 않아도 되고, 어차피 캡처를 할 때 폼이 숨겨져야 하므로 단순하게 디자인한다.



캡처한 파일을 저장해야 하므로, TSaveDialog 컴포넌트를 하나 추가한다. 그리고, DefaultExt 프로퍼티를 '.bmp', Filter 프로퍼티를 '\*.bmp|\*.bmp'로 설정하자. 캡처할 이미지를 저장할 TImage 변수를 다음과 같이 전역 변수로 선언한다.

```
var
  Form1: TForm1;
  CaptureImage: TImage;
```

그러면, 실제로 캡처를 수행하는 프로시저를 작성해보자. 다음과 같이 Capture 라는 프로시저를 private 섹션에 선언한다.

```
private
    procedure Capture;
```

그러면, Capture 프로시저를 실제로 구현해보자. 스크린 전체를 캡처하는 프로시저는 다음과 같이 구현할 수 있다.

```
procedure TForm1.Capture;
var
    DC, DCBuffer, Buffer: HDC;
    x, y: integer;
begin
    Hide;
    Sleep(200);
    DC := CreateDC('DISPLAY', nil, nil, nil);
    x:= Screen.Width;
    Y:= Screen.Height;
    DCBuffer := CreateCompatibleDC(DC);
    Buffer := CreateCompatibleBitmap(DC, x, y);
    SelectObject(DCBuffer, Buffer);
    BitBlt(DCBuffer, 0, 0, x, y, DC, 0, 0, SRCCOPY);
    BitBlt(CaptureImage.Canvas.Handle, 0, 0, CaptureImage.Width,
        CaptureImage.Height, DCBuffer, 0, 0, SRCCOPY);
    DeleteDC(DCBuffer);
    DeleteDC(DC);
    CaptureImage.Refresh;
    Show;
end;
```

지역 변수로 선언한 DC 는 디스플레이 객체의 디바이스 컨텍스트를 저장한다. 그리고, DCBuffer 에는 DC 에 대한 메모리 디바이스 컨텍스트를 생성해서 대입하며, 실제 비트맵에 대한 디바이스 컨텍스트는 Buffer 변수에 저장된다. x, y 는 캡처할 화면의 크기를 담는 변수로 쓰인다.

먼저 Hide 를 호출하여 폼을 숨긴다. 그 이후에 Sleep(200)을 호출한 이유는 폼을 숨기는데 약간의 시간을 벌여주기 위한 것이다. 필자가 테스트한 바로는 바로 캡처를 시도할 경우 폼이 숨겨지는 모양이 그대로 캡처되어 버린다. 그 다음 라인에서는 사용할 디바이스

컨텍스트를 생성한다. CreateDC 함수는 지정된 이름의 디바이스 컨텍스트를 생성한다. 여기서는 화면에 보여주는 비트맵이므로 'DISPLAY'로 설정하여 디스플레이 디바이스 컨텍스트를 선택한다. 디바이스 컨텍스트에 대한 자세한 내용은 이장의 앞 부분에 잘 정리해 놓았으므로, 이를 참고하기 바란다.

그 다음에는 캡처할 이미지의 크기를 스크린 전체 크기로 설정하고, DCBuffer와 Buffer 변수의 값을 설정한다. CreateCompatibleDC 함수는 지정된 디바이스에 대한 메모리 디바이스 컨텍스트를 생성하는 함수이다. 비트맵을 선택하면, 디바이스 컨텍스트는 스크린으로 복사되거나 인쇄될 이미지를 준비하는데 사용된다. CreateCompatibleDC 함수는 래스터 작업을 지원하는 장치에서만 사용될 수 있다.

디스플레이 DC는 보존성이 없기 때문에, 이를 통해 출력한 것은 다른 윈도우가 그 위치를 덮어 씌우기만 해도 문제가 생길 수 밖에 없다. 이렇게 하면 윈도우의 특성 상 좋지 않은 것은 당연하다. 이를 막기 위해서는 디스플레이 DC의 내용을 계속 간직하고 있을 보존성이 있는 DC가 필요하게 되는데, 이런 것이 있다면 디스플레이 DC의 내용을 여기에 보관했다가 필요할 때 다시 그려주면 될 것이다.

이 때 사용하는 것이 CreateCompatibleDC 함수를 이용해서 생성하는 메모리 DC이다. 메모리 DC의 용도로는 이미지를 반복적으로 사용할 때와 비트 연산을 하고자 할 때를 들 수 있다. 메모리 DC는 경우에 따라서 빠르게 생성해서 사용할 수 있고, 저장도 어느 정도 가능하기 때문에 장점이 많지만, 메모리를 소모한다는 단점을 가지고 있다. 별 것 아닌 것 같지만 디스플레이에 필요한 메모리가 워낙 크기 때문에, 사용을 남발하는 것은 절대로 삼가해야 하며, 사용한 뒤에는 반드시 DeleteDC를 호출하여 꼭 해제를 해 주어야 한다.

Buffer 변수에 값을 대입하기 위해서 사용하는 CreateCompatibleBitmap 함수는 지정된 디바이스 컨텍스트의 장치와 호환되는 비트맵을 생성한다. 이 함수에 의해 생성되는 비트맵의 컬러 포맷은 파라미터에 지정된 디바이스의 컬러 포맷과 일치한다. 이 비트맵은 디바이스와 호환되는 메모리 디바이스 컨텍스트에 선택되어 사용되는 경우가 많다. 메모리 디바이스 컨텍스트는 컬러와 모노 비트맵을 모두 허용하기 때문에, 디바이스 컨텍스트로 메모리 디바이스 컨텍스트가 지정될 경우, CreateCompatibleBitmap 함수에 의해 반환되는 비트맵의 포맷은 다를 수 있다. CreateCompatibleBitmap 함수의 두번째와 세번째 파라미터는 비트맵의 크기를 지정한다. 여기서는 스크린의 크기를 지정한다.

그 다음에는 SelectObject 함수를 이용해서 지정된 디바이스 컨텍스트에 사용할 그래픽 객체를 선택한다. 여기서는 비트맵 객체인 Buffer를 선택해야 한다.

이제 가장 중요한 BitBlt 함수의 사용법을 익힐 차례이다. BitBlt 함수에 의해서 실제로 비트맵에 그림을 대입하게 된다. BitBlt 함수는 지정된 소스 디바이스 컨텍스트에서 목적 디바이스 컨텍스트로 해당되는 픽셀들의 컬러 데이터를 전송하게 된다. 이때 스트레치(stretch), 압축(compress), 회전(rotate) 등의 변형을 할 수가 있다. 소스와 목적 디바이스 컨텍스트의 컬러 포맷이 맞지 않으면, 소스 컬러 포맷을 목적 포맷에 맞추어 변환을 한다. 우리가 사용한 두가지 디바이스 컨텍스트는 서로 호환되도록 설정하였으므로 컬러 포맷의

문제는 없다.

BitBlt 함수는 파라미터가 모두 9 개로 무척 많은데, 사실 그 내용을 보면 그다지 어렵지 않다. 첫번째 파라미터에는 목적 디바이스 컨텍스트의 핸들을 지정하며, 2~5 번째 파라미터는 목적 디바이스 컨텍스트에 복사할 영역의 4 군데 좌표를 설정한다. 6 번째 파라미터는 소스 디바이스 컨텍스트의 핸들을 지정하면 되고, 7~8 번째 파라미터는 소스 디바이스 컨텍스트의 좌측상단 꼭지점의 좌표를 설정한다. 마지막으로 9 번째 파라미터에는 레스터 작업의 코드를 지정한다. 여기서 가장 중요한 것은 마지막 파라미터인데 지우거나, 복사, 반전 등의 여러가지 효과를 지정할 수 있다. Capture 프로시저에 사용한 BitBlt 함수의 역할은 디바이스 컨텍스트인 디스플레이 장치의 전체 스크린 영역을 일단 메모리 디바이스 컨텍스트로 복사한 후, 이를 다시 CaptureImage 이미지 컴포넌트의 캔버스로 복사하는 것이다. 이때에는 SRCCOPY 를 사용하여 소스 영역의 내용을 직접 목적 영역으로 복사한다. 이것으로 캡처 작업이 완료된다.

레스터 작업 코드는 여러가지로 사용되기 때문에, 익혀두면 많은 도움이 된다. 이를 다음에 정리하였으므로, 참고하기 바란다.

값	설 명
BLACKNESS	목적 영역을 물리적 팔레트의 인덱스 0 인 색상(보통 검은 색)으로 채운다.
DSTINVERT	목적 영역을 반전한다.
MERGECOPY	소스 영역의 색상을 AND 연산을 통해 지정된 패턴으로 색상을 합친다.
MERGEPAINT	반전된 소스 영역의 색상을 목적 영역의 색상과 OR 연산을 통해 합친다.
NOTSRCCOPY	소스 영역을 목적 영역으로 반전하여 복사한다.
NOTSRCERASE	소스와 목적 영역의 색상을 OR 연산을 해서 합친 후, 이를 반전한다.
PATCOPY	지정된 패턴으로 목적 비트맵에 복사한다.
PATINVERT	지정된 패턴의 색상과 목적 영역의 색상을 XOR 연산으로 합친다.
PATPAINT	소스 영역의 반전된 색상과 패턴의 색상을 OR 연산을 하고, 이를 다시 목적 영역의 색상과 OR 연산을 한다.
SRCAND	소스와 목적 영역의 색상을 AND 연산한다.
SRCCOPY	소스 영역의 내용을 목적 영역으로 직접 복사한다.
SRCERASE	목적 영역의 반전된 색상과 소스 영역의 색상을 AND 연산한다.
SRCINVERT	소스와 목적 영역의 색상을 XOR 연산한다.
SRCPAINT	소스와 목적 영역의 색상을 OR 연산한다.
WHITENESS	목적 영역을 물리적 팔레트의 인덱스 1 인 색상(보통 흰색)으로 채운다.

그러면, 이제 폼의 OnCreate 이벤트 핸들러에서 이미지 컴포넌트를 생성하고, 프로퍼티를 지정한다. 그리고, Capture 함수를 호출하여 일단 스크린을 CaptureImage 변수에 캡처한다.

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    CaptureImage := TImage.Create(Self);
    CaptureImage.Left := 0;
    CaptureImage.Top := 0;
    CaptureImage.Width:=Screen.Width;
    CaptureImage.Height:=Screen.Height;
    Capture;
end;

```

Button1 과 Button2 의 이벤트 핸들러를 다음과 같이 작성하여 캡처를 하고, 캡처한 이미지를 저장할 수 있도록 한다.

```

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

```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    if SaveDialog1.Execute then
        CaptureImage.Picture.Bitmap.SaveToFile(SaveDialog1.FileName);
end;

```

## 지역(Region) 이란 ?

지역은 상상할 수 있는 모든 형태의 모양을 클리핑 영역으로 설정할 수 있도록 허용한다. 이를 이용하면 그리게 되는 모든 것들이 그 영역에 클리핑되거나 마스크되게 할 수 있다. 여기에 대해서는 델파이 자체가 지원하는 메소드는 없다. 그러므로, 이를 사용하기 위해서는 GDI 를 직접 사용해야 한다.

지역을 사용하는 예제를 간단하게 하나 만들어 보자. 폼에 버튼을 하나 얹고, Caption 을 ‘실 행’으로 설정한 뒤 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
var

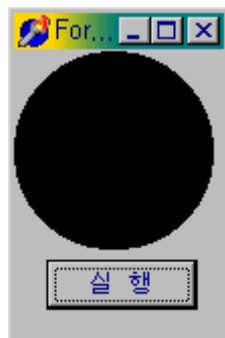
```

```

NewRgn, OldRgn: hRgn;
begin
    NewRgn := CreateEllipticRgn(0, 0, 100, 100);
    OldRgn := SelectObject(Canvas.Handle, NewRgn);
    PatBlt(Canvas.Handle, 0, 0, Width, Height, BLACKNESS);
    SelectObject(Canvas.Handle, OldRgn);
    DeleteObject(NewRgn);
end;

```

지역을 사용하기 위해서는 일단 2 개의 지역을 지정할 수 있는 핸들을 저장할 변수를 선언한 후 CreateEllipticRgn 함수를 실행하여 동그란 형태의 지역을 NewRgn 변수에 저장한다. 그리고, SelectObject 함수로 NewRgn 지역 객체를 선택한다. 이때 SelectObject 객체는 이전의 지역 객체를 반환하므로, 이를 저장해 두었다가 나중에 지역을 복구할 때 사용한다. PatBlt 함수는 주어진 사각 영역에 현재 선택된 브러쉬를 이용해서 칠하는 함수로, 칠하는 방법을 마지막 파라미터에 지정하도록 되어 있다. 여기서는 BLACKNESS 를 지정하여 까맣게 칠하도록 하였다. 그 다음에는 원래의 지역을 복구하고, 지역 객체를 해제하면 끝난다. 이 어플리케이션을 실행하면 분명히 PatBlt 로 사각형 영역을 칠하도록 했지만, NewRgn 이 동그란 영역으로 지정되었으므로 다음과 같이 동그랗게 칠해진다.



## 매핑 모드 (Mapping Modes)

일반적으로 윈도우의 기본적인 매핑 모드는 MM\_TEXT 를 사용한다. 이 모드에서는 모든 드로잉 좌표가 좌상측에서 우하로 내려올수록 X, Y 가 1 픽셀씩 증가한다. 즉, 좌상 꼭지점을 (0, 0)으로 하여 우하로 내려오면 양의 값으로 증가하는 것이다. 윈도우에는 이 모드 말고도 여러가지 매핑에 대한 모드가 존재하는데, 여기에는 다음과 같은 것들이 있다.

모 드	설 명
-----	-----



MM_TEXT	1 픽셀 단위로, y 가 증가할수록 아래로 내려온다.
MM_LOMETRIC	0.1mm 단위로, y 가 증가할수록 위로 올라간다.
MM_HIMETRIC	0.01mm 단위로, y 가 증가할수록 위로 올라간다.
MM_LOENGLISH	0.01 인치 단위로, y 가 증가할수록 위로 올라간다.
MM_HIENGLISH	0.001 인치 단위로, y 가 증가할수록 위로 올라간다.
MM_TWIPS	1/20 포인트(1/1440 인치) 단위로, y 가 증가할수록 위로 올라간다.
MM_ISOTROPIC	SetWindowExtEx 와 SetViewportExtEx 함수를 이용하여 단위와 축의 방향을 결정한다. 그런데, x 와 y 의 단위는 같은 크기이다.
MM_ANISOTROPIC	MM_ISOTROPIC 과 거의 동일하지만, x 와 y 의 단위 크기가 다를 수 있다.

매핑 모드를 사용할 때에는 이들의 변형을 위해서 몇 가지 함수를 사용할 수 있다. 이들 함수는 과거의 설정을 반환하므로 이를 임시 변수에 저장했다가, 사용이 끝나면 다시 복구시켜 주는 것이 좋다. 가장 흔히 사용되는 함수로는 다음과 같은 것들이 있다.

1. SetMapMode: 매핑 모드를 결정한다.
2. SetWindowOrgEx: (0, 0)이 위치할 좌표를 설정한다.
3. SetWindowExtEx: 논리적 단위를 결정한다.
4. SetViewportExtEx: 논리적 단위가 매핑할 유닛의 디바이스 크기를 결정한다.

간단한 예제를 하나 만들어 보자. 폼에 버튼을 2 개 얹고 각각의 캡션을 'LOMETRIC', 'ANISOTROPIC'으로 설정한다. 버튼의 캡션에서도 추측할 수 있겠지만, Button1 은 MM\_LOMETRIC 모드를, Button2 는 MM\_ANISOTROPIC 모드를 사용한다.

먼저 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    OldMapMode: Integer;
    OldOrigin: TPoint;
begin
    OldMapMode := SetMapMode(Canvas.Handle, MM_LOMETRIC);
    SetWindowOrgEx(Canvas.Handle, 0, 200, @OldOrigin);
    Canvas.Ellipse(0, 0, 200, 200);
    SetWindowOrgEx(Canvas.Handle, OldOrigin.X, OldOrigin.Y, nil);
    SetMapMode(Canvas.Handle, OldMapMode);
end;
```

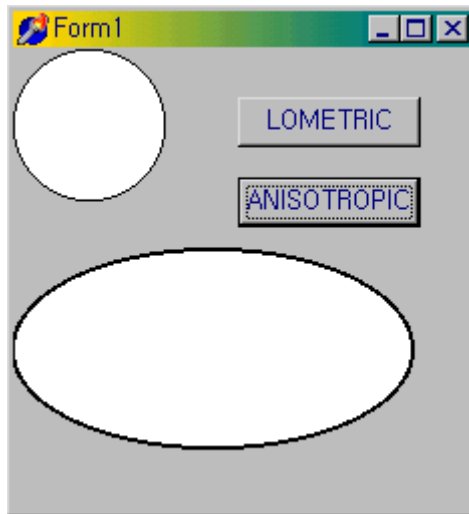
코드는 단순하다. SetMapMode 함수를 이용하여 MM\_LOMETRIC 모드로 설정하고, 원점을 (0, 200)으로 설정한다. MM\_LOMETRIC 모드는 0.1mm 단위이므로 원점은 폼의 좌상에서 아래로 2cm 아래에 위치하게 된다. 여기에서 지름이 2cm 인 원을 그리게 된다. 그리고 나서, 원래의 원점과 매핑 모드를 각각 OldMapMode 와 OldOrigin 변수에 저장했던 것을 다시 복구하면 된다.

Button2 의 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
    OldMapMode: Integer;
    OldOrigin: TSize;
    OldWindowExtent: TSize;
    OldViewportExtent: TSize;
begin
    OldMapMode := SetMapMode(Canvas.Handle, MM_ANISOTROPIC);
    SetWindowExtEx(Canvas.Handle, 100, 100, @OldWindowExtent);
    SetViewportExtEx(Canvas.Handle, 200, 100, @OldViewportExtent);
    SetWindowOrgEx(Canvas.Handle, 0, -100, @OldOrigin);
    Canvas.Ellipse(0, 0, 100, 100);
    SetWindowOrgEx(Canvas.Handle, OldOrigin.cx, OldOrigin.cy, nil);
    SetViewportExtEx(Canvas.Handle, OldViewportExtent.cx, OldViewportExtent.cy, nil);
    SetWindowExtEx(Canvas.Handle, OldWindowExtent.cx, OldWindowExtent.cy, nil);
    SetMapMode(Canvas.Handle, OldMapMode);
end;
```

별로 다를 내용은 없지만, MM\_ANISOTROPIC 모드에서 SetViewportExtEx 함수에 의해 x 축의 크기 요소를 y 축의 2 배로 설정하였으므로 Ellipse(0, 0, 100, 100) 메소드에 의해 가로 길이가 세로의 2 배인 타원이 그려지게 된다.

이 어플리케이션을 실행하고, 두 개의 버튼을 차례로 클릭하면 다음과 같은 결과를 볼 수 있다.



## 어플리케이션에 비디오 클립 추가

텔파이 4 의 애니메이션 컨트롤을 이용하면 조용한 비디오 클립을 간단하게 어플리케이션에 추가할 수 있다. Win32 페이지의 TAnimate 컴포넌트를 사용하려면 CommonAVI, FileName, ResName, ResID 프로퍼티 중의 하나를 설정해서 보여줄 비디오 클립을 선택한다. 일단 AVI 파일을 메모리에 적재한 뒤에, Active 프로퍼티를 설정하거나 Play 메소드에 의해 스크린에 AVI 클립을 보여줄 때 첫번째 프레임부터 보여주려 할 경우에는 Open 프로퍼티를 True 로 설정한다.

그리고, AVI 클립을 반복할 횟수를 Repetitions 프로퍼티에 설정하는데, 이 값에 0 을 주면 Stop 메소드가 호출될 때까지 재생이 반복된다. 그 밖에 여러가지 설정을 변경할 수 있는데 예를 들어, 첫번째 프레임부터 보여주지 않고 특정 프레임부터 시작하게 하려면 StartFrame 프로퍼티를 프레임의 번호로 설정하면 된다.

## 오디오/비디오를 모두 재생할 수 있는 어플리케이션의 제작

오디오와 비디오를 모두 재생할 수 있는 어플리케이션을 제작하려면, System 페이지에 있는 TMediaPlayer 컴포넌트를 사용한다.

이 컴포넌트를 사용하려면 먼저 DeviceType 프로퍼티를 사용하려는 적절한 디바이스 유형으로 설정해야 한다. 이 프로퍼티가 dtAutoSelect 로 설정되면 디바이스 유형은 미디어 파일의 확장자에 의해 결정된다. 디바이스가 미디어를 파일로 저장하면, 미디어 파일의 이름을 FileName 프로퍼티에 지정하여 사용하게 된다

AutoOpen 프로퍼티를 True 로 설정하면, 미디어 플레이어는 자동으로 지정된 디바이스를 열게 된다. 이 값이 False 이면 Open 메소드가 호출되어야만 디바이스가 열린다.

AutoEnable 프로퍼티는 미디어 플레이어 버튼을 자동으로 enable, disable 할 것인지를 결정

하는 것이다. 이를 설정하거나 아니면, EnableButtons 프로퍼티를 이용하여 각각의 버튼을 enable, disable 시킬 수 있다.

미디어 플레이어의 버튼으로는 다음과 같이 Play, Pause, Stop, Next, Previous 등이 있다.



경우에 따라서는 미디어 플레이어를 런타임에 보이지 않게 하고 싶을 때에는 Visible 프로퍼티를 False로 설정하면 되며, 이럴 때에는 Play, Pause, Stop, Next, Previous 등의 적절한 메소드를 호출하여 작동시키면 된다.

그 밖에도 미디어가 디스플레이 윈도우를 요구할 경우에는 Display 프로퍼티를 미디어를 디스플레이할 컨트롤로 설정하면 되며, 디바이스가 다중 트랙을 사용할 경우에는 Tracks 프로퍼티를 해당되는 트랙으로 설정하면 된다.

디바이스 유형과 재생되는 종류와 트랙과 디스플레이 윈도우의 사용 여부에 대한 사항을 다음에 나타내었다.

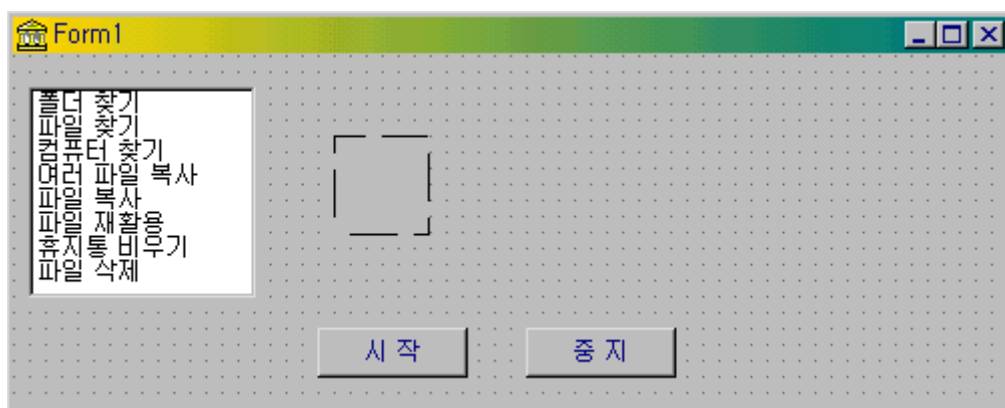
디바이스	사용되는 하드웨어/소프트웨어	트랙 사용여부	디스플레이 컨트롤 사용여부
dtAVIVideo	AVI Video Player for Windows (AVI Video files)	No	Yes
dtCDAudio	CD Audio Player for Windows CD Audio Player (CD Audio Disks)	Yes	No
dtDAT	Digital Audio Tape Player (Digital Audio Tapes)	Yes	No
dtDigitalVideo	Digital Video Player for Windows (AVI, MPG, MOV files)	No	Yes
dtMMMovie	MM Movie Player (MM film)	No	Yes
dtOverlay	Overlay device (Analog Video)	No	Yes
dtScanner	Image Scanner N/a for Play (scans images on Record)	No	No
dtSequencer	MIDI Sequencer for Windows (MIDI files)	Yes	No
dtVCR	Video Cassette Recorder (Video Cassettes)	No	Yes
dtWaveAudio	Wave Audio Player for Windows (WAV files)	No	No

### 애니메이션 컨트롤의 활용

Animate 컨트롤은 단일 비디오 스트림을 가진 AVI 파일 만을 재생할 수 있으며, RLE8 압축 기법으로 압축되거나 압축을 해제하며, 팔레트 변경은 할 수 없다. 그리고, 사운드가 있을 경우 압축이 무시된다.

Animate 컨트롤의 재미있는 특징 중에 하나는 폼 위에 올려 놓고, FileName 이나 CommonAVI 프로퍼티를 설정하고 Active 프로퍼티를 True 로 하면, 디자인 시에서도 애니메이션을 볼 수 있다. 그러면, 표준으로 제공되는 애니메이션인 CommonAVI 프로퍼티의 내용을 사용자가 선택하면 이를 보여주는 간단한 예제를 하나 만들어 보자

먼저 폼에 리스트 박스 하나와 버튼 2 개, 애니메이트 컨트롤 1 개를 얹어서 다음과 같이 디자인 한다.



리스트 박스의 Items 프로퍼티를 앞의 화면에 보이는 대로 설정하고, 버튼 2 개의 Caption 프로퍼티를 ‘시작’과 ‘중지’로 설정한다.

ListBox1 의 OnClick 이벤트 핸들러를 다음과 같이 설정한다.

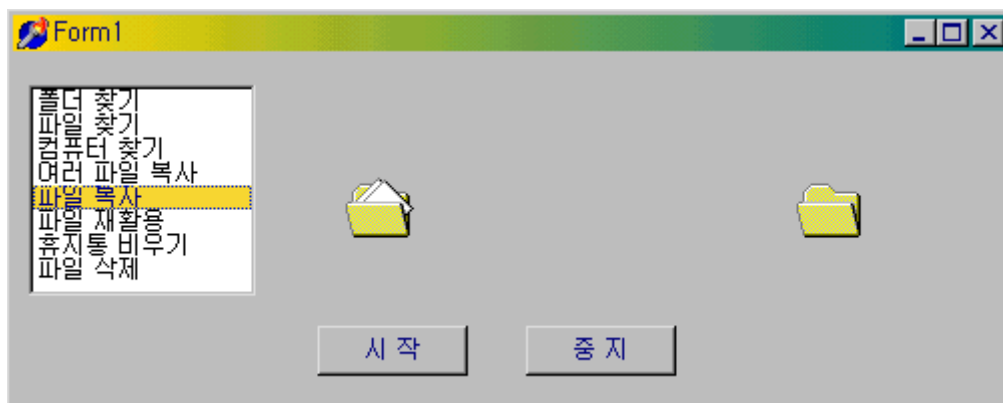
```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
    case ListBox1.ItemIndex of
        0: Animate1.CommonAVI := aviFindFolder;
        1: Animate1.CommonAVI := aviFindFile;
        2: Animate1.CommonAVI := aviFindComputer;
        3: Animate1.CommonAVI := aviCopyFiles;
        4: Animate1.CommonAVI := aviCopyFile;
        5: Animate1.CommonAVI := aviRecycleFile;
        6: Animate1.CommonAVI := aviEmptyRecycle;
        7: Animate1.CommonAVI := aviDeleteFile;
    end;
end;
```

즉, 리스트 박스에 보여주는 내용을 그대로 적용하는 것이다. 그리고, 두 개의 버튼의 OnClick 이벤트 핸들러는 각각 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Animate1.Active := True;  
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Animate1.Stop;  
end;
```

이제 완성이 되었으므로, 이를 실행해 보자. ‘파일 복사’를 선택하면 다음과 같은 화면을 볼 수 있을 것이다.



## 정 리 (Summary)

이번 장에서는 델파이를 이용하여 그림을 그리고, 비트맵을 다루는 방법과 비디오와 오디오 클립을 사용하는 방법에 대해서 알아보았다. 사실 그래픽에 대한 부분은 따로 책을 한 권 써야할 정도로 설명할 내용도 많고, 이해해야 할 내용도 많다. 그리고, 시각적 컨트롤을 나름대로 제작해서 쓰려고 하는 사람들은 여기에 대한 필수적인 이해가 있어야 한다.

팔레트에 대해서도 잘 알고 있어야 하며, 그래픽이라는 객체들이 크기가 상당히 큰 경우들이 많기 때문에 메모리를 효과적으로 사용하는 방법이나 파일 포맷과 파일을 다루는 방법 등에 대해서도 잘 알고 있어야 한다.

이 책에서는 지면 관계상 그래픽에 대해서 가장 기본적이고 기초적인 부분만 소개하였다.

하지만, 훌륭한 델파이 프로그래머가 되기 위해서는 꼭 넘어야 할 산이므로 단순히 델파이  
가 제공하는 캔버스의 메소드에 의존하지 말고, 필수적인 API 함수들의 사용법은 반드시 익  
혀놓도록 권하는 바이다.

다음 장에서는 윈도우 95 가 등장하면서부터 제공되기 시작한 Win32 의 향상된 공통 컨트롤(Common Control) 들의 사용법과 활용 방법에 대해서 알아볼 것이다. 델파이의 수많은  
컴포넌트 들 중에서 유용하게 사용될 수 있음에도 불구하고, 사용법에 대한 소개와 활용이  
비교적 미흡한 편인 컨트롤 들을 중심으로 설명할 것이다.

# Win32 공통 컨트롤 정복

## (Mastering Win32 Common Controls)

Win32 의 공통 컨트롤(common control) 컴포넌트 들은 유용하게 생각되면서도 사용하기가 비교적 까다로운 컴포넌트가 많다. 또한, 현재까지 나온 델파이 서적들에서도 기본적인 컴포넌트의 사용 방법에 대해서는 비교적 자세한 사용법을 기술하고 있지만 Win32 공통 컨트롤에 대한 설명은 비교적 부족한 것이 사실이다.

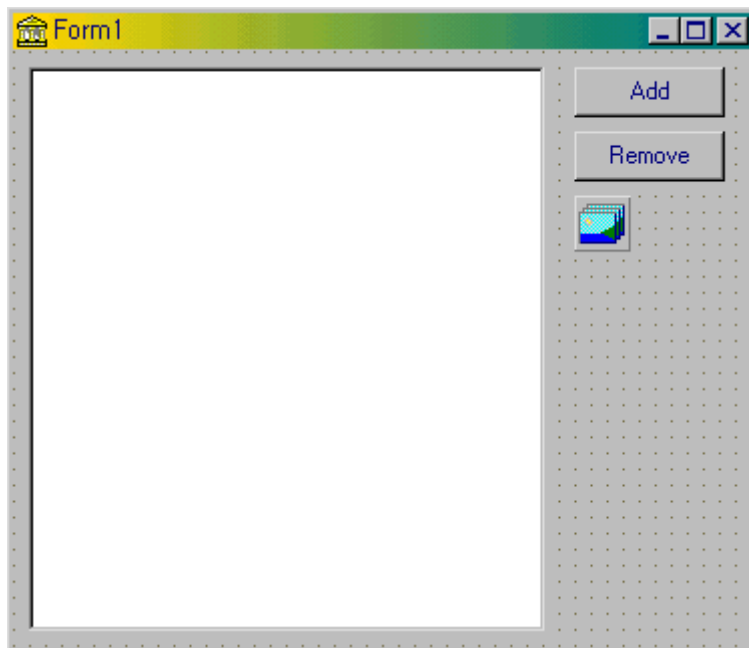
Win32 공통 컨트롤에는 트랙바(Track Bar)와 업다운(UpDown) 컨트롤, 핫키(HotKey) 컨트롤이나 진행바(Progress Bar)와 같은 단순한 컨트롤과 다소 복잡한 상태바(Statud Bar), 헤더 컨트롤, 이미지 리스트, 탭 컨트롤(Tab Control), 페이지 컨트롤(Page Control), 리스트뷰와 트리뷰 등이 있다. 그 밖에도 델파이 3 에서는 툴바(Tool bar)와 쿨바(Cool bar), 애니메이션 (Animate), 달력 컴포넌트(DateTimePicker) 등이 추가되었고 델파이 4 에서는 페이지 스크롤러(Page Scroller)와 컨트롤 바(Control Bar)가 추가 되었다.

지면 관계상 이들 모두를 다룰 수는 없고, 이번 장에서는 비교적 새로운 형태의 객체 접근 방법에 대한 사용자 인터페이스를 제공하면서, 사용법이 비교적 까다로운 트리뷰(TreeView)와 컴포넌트의 활용 방법을 소개하고, 리스트뷰(ListView) 컴포넌트와 이미지 리스트, 상태바 컨트롤을 같이 이용하여 탐색기와 유사한 어플리케이션을 작성해 볼 것이다. 그리고 명령을 수행하도록 하는 사용자 인터페이스를 제공하는 툴바(Toolbar)와 쿨바(Coolbar)의 사용방법에 대해서 간단하게 알아볼 것이다.

### 트리뷰 컴포넌트 시작하기

트리뷰 컴포넌트를 다루는 기본적인 방법에 대해서 알아보도록 하자. 새로운 어플리케이션을 시작하고 폼에 TTreeView 컴포넌트 하나와 버튼을 두개 올려 놓자. 버튼의 캡션은 각각 'Add', 'Remove'로 설정한다. 그리고 TImageList 컴포넌트를 다음과 같이 추가한다.





트리뷰 컴포넌트는 실제로 그 구조에 있어서 노드라는 객체를 사용하게 된다. 노드는 트리뷰 컴포넌트를 이루는 가장 기본적인 단위가 되며, 눈에 보이는 각각의 아이템 들이다. 이러한 노드는 TTreeNode 클래스로 선언되어 있다. 그럼 트리뷰 컴포넌트에 새로운 노드를 추가하는 방법을 알아보자.

노드를 추가할 때에는 어느 노드에 노드를 추가할 지를 지정해 주어야 한다. 이러한 노드의 부모는 루트 노드일 수도 있고, 다른 자식 노드일 수도 있다. 루트 노드의 경우에는 부모가 없다. 그러므로, 루트 노드의 경우 TTreeNode.Parent 의 값은 nil 이다. 루트 노드를 제외한 노드는 루트 노드를 포함한 다른 노드를 부모로 가지게 된다.

노드를 추가할 때에는 보통 현재 선택된 노드가 있으면 그 노드의 형제 노드로 추가되거나, 자식 노드로 추가 하는 등의 결정을 하게 된다. 이처럼 현재 선택된 노드를 알아볼 때 사용하는 것인 Selected 이다. 보통 이 프로퍼티를 사용하면 현재 선택된 노드를 TTreeNode 의 형태로 알아볼 수 있다. 만약 이 프로퍼티가 nil 이라면 현재 트리뷰가 비어 있거나, 아무 노드도 선택되지 않을 것이다. 트리뷰가 비어 있는지는 아이템의 컬렉션인 Items.Count 프로퍼티로 알아볼 수 있다. 이 값이 0 이면 트리뷰에 노드가 하나도 없는 경우이다.

노드를 추가하는 메소드에 대해서 알아보자. 트리뷰에서 사용하는 메소드로는 다음과 같은 것들이 있다.

```
function Add(Node: TTreeNode; const S: string): TTreeNode;  
function AddFirst(Node: TTreeNode; const S: string): TTreeNode;  
function Insert(Node: TTreeNode; const S: string): TTreeNode;  
function AddChild(Node: TTreeNode; const S: string): TTreeNode;
```

```

function AddChildFirst(Node: TTreeNode; const S: string): TTreeNode;
function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;
function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;
function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;
function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;
function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;

```

상당히 많아 보이지만, 크게 2 가지 카테고리로 나누어 볼 수 있다.

우선 파라미터를 보면 메소드 이름에 Object 가 들어가 있는 것들에는 Ptr 이라는 포인터 파라미터가 하나씩 더 있는 것을 알 수 있다. Object 가 들어가 있지 않은 메소드 들은 단순히 기준이 되는 노드(Node 파라미터)와 트리뷰에 추가할 문자열(S 파라미터)만 지정해 주면 트리뷰에 노드가 추가된다. 그에 비해 Object 가 들어가 있는 메소드 들은 트리뷰에 노드를 추가하고, 각 노드에 객체를 지정할 수 있다. 이때 각 객체에 대한 포인터를 노드에 지정하는 파라미터가 S 파라미터이다.

그리고, First 가 들어가 있는 메소드 들은 노드의 위치를 지정하는 것으로 추가되는 노드가 동격의 노드들 중에서는 첫번째 위치로 추가된다는 의미이다. Add, Insert, AddChild 의 차이는 Add 는 형제 노드 중에서 제일 마지막 위치에, Insert 는 형제 노드 중 현재 위치를 대신 차지하게 하는 것이며, AddChild 는 선택된 노드의 자식으로 제일 마지막 위치에 추가되게 하는 메소드이다.

그러면 이제 연습을 해 보자. Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    Text: string;
begin
    if TreeView1.Selected = nil then
        begin
            if TreeView1.Items.Count = 0 then
                begin
                    with TreeView1.Items.AddFirst(nil, 'Root') do
                        begin
                            Selected := True;
                        end;
                    end
                else
                    begin

```

```

        ShowMessage('부모 노드를 선택하세요 !');
    Exit;
end;
end
else
begin
    InputQuery('새 노드', '이름 ?', Text);
    TreeView1.Items.AddChild(TreeView1.Selected, Text);
end;
end;

```

먼저 선택된 노드가 없는지 검사한다. 이때 Selected 프로퍼티의 값이 nil 이라면 트리뷰가 현재 비어 있는 경우가 있을 것이고, 노드가 선택되지 않은 경우가 있을 것이다. 현재 비어 있다면 AddFirst 메소드에 첫번째 파라미터로 nil 을 대입해서 루트 노드를 생성한다. 그리고, TTreeNode 의 Selected 프로퍼티를 True 로 설정해서 이 노드가 선택되도록 한다. 만약 노드가 선택되지 않은 경우라면 노드를 선택하라는 메시지를 화면에 띄우도록 한다. Selected 프로퍼티의 값이 nil 이 아니면 추가할 노드의 이름을 InputQuery 함수를 이용해서 입력받고, Selected 노드의 자식 노드로 새로운 노드를 추가한다. 사용법이 그다지 어렵지 않다는 것을 알 수 있을 것이다. 그러면 노드를 제거하는 부분을 만들어 보자. Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    if TreeView1.Selected = nil then
    begin
        ShowMessage('선택된 것이 없습니다.');
```

```

        Exit;
    end;
    if TreeView1.Selected.Level = 0 then
    begin
        ShowMessage('Root 는 제거할 수 없습니다');
```

```

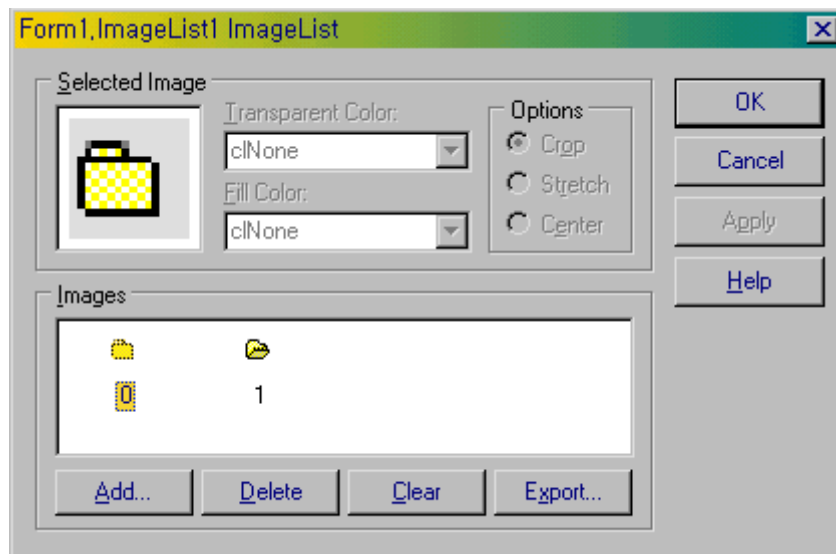
        Exit;
    end;
    TreeView1.Selected.Delete;
end;

```

선택된 것이 없거나 선택된 노드가 루트일 경우에는 제거하지 않고, 선택된 노드가 있으면 Delete 메소드를 사용해서 노드를 제거한다.

Root 인지 알아보는 프로퍼티로 Level 프로퍼티를 사용했는데, 이 값이 0 이면 루트이며, 자식 노드가 하나 늘어날 때 마다 Level 프로퍼티가 하나씩 증가한다.

이제 처음에 추가한 TImageList 컴포넌트를 이용해서 노드를 상징하는 이미지를 나타내도록 해보자. 흔히 사용하는 폴더 이미지를 이용해서, 하부 노드가 보이는 경우에는 열린 폴더의 그림이 하부 노드가 안 보이도록 된 경우에는 닫힌 폴더가 나타내도록 하자. 이때 다음과 같이 닫힌 폴더 그림의 인덱스를 0, 열린 폴더를 1 로 설정한다. .



그리고, 트리뷰 컴포넌트가 ImageList1 을 사용하게 하기 위해 오브젝트 인스펙터에서 TreeView1 의 Images 프로퍼티를 ImageList1 으로 설정한다. 이런 작업이 끝났으면 TImageList 컴포넌트를 이용해서 노드가 선택되었을 때의 이미지를 설정하기 위해서 아래와 같이 Button1 의 OnClick 이벤트 핸들러를 수정한다.

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

...(중략)

```
if TreeView1.Items.Count = 0 then
begin
  with TreeView1.Items.AddFirst(nil, 'Root') do
  begin
    Selected := True;
    ImageIndex := 0;
```

```

        SelectedIndex := 1;
    end;

```

... (중략)

```

    InputQuery('새 노드', '이름 ?', Text);
    with TreeView1.Items.AddChild(TreeView1.Selected, Text) do
    begin
        ImageIndex := 0;
        SelectedIndex := 1;
    end;

```

즉, 노드를 추가할 때 마다 그 노드의 기본적인 ImageIndex 와 SelectedIndex 프로퍼티에 대한 정보를 주는 것이다. ImageIndex 프로퍼티는 노드가 선택되지 않았을 때의 TImageList 컴포넌트에서의 이미지의 인덱스를 나타낸다. 여기서는 닫힌 폴더 그림을 나타내므로 0 을 대입한다. SelectedIndex 프로퍼티는 노드가 선택되었을 때 보여줄 이미지의 인덱스이다. 이 값이 같으면 노드가 선택되었을 때와 선택되지 않았을 때 보여지는 이미지가 동일하다.

## 트리노드(TreeNode) 정복

트리뷰 컴포넌트는 실제로 눈에 보이는 부분이다. 그렇지만, 그 내부의 동작은 기본적으로 각각의 트리 노드에 대해서 잘 알아야 트리를 정복했다고 말할 수가 있는 것이다. 이번 섹션에서는 트리 노드를 다루는 법에 대해서 알아보도록 하자.

### ● 이름의 충돌문제 해결

트리의 경우에 흔히 같은 부모 노드를 가진 노드 들의 경우에 노드의 이름이 동일하면 안되는 경우가 있다. 예를 들어, 파일 시스템을 예로 들면 하나의 디렉토리에 들어있는 파일의 이름이 동일한 것이 둘 이상 존재해서는 안된다. 그렇지만, 트리를 자체에는 이를 검사하는 방법이 없기 때문에 이런 역할을 하는 루틴을 만들어 주어야 한다. 다음에 소개하는 함수가 이러한 이름 충돌 문제를 검사해 주는 함수이다.

소스 코드를 살펴 보자.

```

function IsDuplicate(ANode: TTreeNode; NewName: string; Inclusive: Boolean): Boolean;
var

```

```

    TempNode: TTreeNode;
begin
    if ANode = nil then
        begin
            Result := False;
            Exit;
        end;
    if Inclusive then
        if CompareText(ANode.Text, NewName) = 0 then
            begin
                Result := True;
                Exit;
            end;
        TempNode := ANode;
        repeat
            TempNode := TempNode.GetPrevSibling;
            if TempNode <> nil then
                if CompareText(TempNode.Text, NewName) = 0 then
                    begin
                        Result := True;
                        Exit;
                    end;
            until TempNode = nil;
            TempNode := ANode;
            repeat
                TempNode := TempNode.GetNextSibling;
                if TempNode <> nil then
                    if CompareText(TempNode.Text, NewName) = 0 then
                        begin
                            Result := True;
                            Exit;
                        end;
                until TempNode = nil;
            Result := False;
        end;

```

이 함수의 개요를 살펴 보면 파라미터로 트리 노드(ANode)와 비교할 이름(NewName)과 넘겨진 트리 노드도 검사할 것인지 여부(Inclusive)를 설정해주면, 만약 Inclusive 가 True 라면 현재의 노드의 이름과 새로운 이름이 같은지도 검사하며, 넘겨진 ANode 의 앞쪽에 있는 모든 형제 노드와 뒷쪽에 있는 모든 형제 노드의 이름을 비교해서 NewName 과 같은 것이 있으면 결과를 True 로 반환하는 함수이다.

즉, 트리 노드를 추가할 때 추가될 노드의 이름을 입력받은 후 추가될 위치에 있는 모든 형제 노드들의 이름에 동일한 것이 있는지 먼저 검사할 때 유용하게 쓰일 수 있는 함수이다. 함수의 개요를 읽어 보면 대부분의 소스 코드를 이해할 수 있을 것으로 믿는다.

그런데, 여기서 한가지 소개할 것은 근처의 노드를 얻어오는 메소드 들이다. 앞의 함수에서는 GetPrevSibling, GetNextSibling 이 사용되었다. 이들은 모두 TTreeNode 의 메소드로 각각 앞쪽의 형제 노드와 뒷쪽의 형제 노드를 TTreeNode 형으로 반환해 주는 메소드이다. 이들과 비슷한 역할을 하는 메소드로 다음과 같은 것들이 있다.

```
function GetFirstChild: TTreeNode;  
function GetLastChild: TTreeNode;  
function GetNext: TTreeNode;  
function GetNextSibling: TTreeNode;  
function GetNextChild(Value: TTreeNode): TTreeNode;  
function GetNextVisible: TTreeNode;  
function GetPrev: TTreeNode;  
function GetPrevSibling: TTreeNode;  
function GetPrevChild(Value: TTreeNode): TTreeNode;  
function GetPrevVisible: TTreeNode;
```

Next 와 Prev 가 들어간 것의 차이는 쉽게 이해 할 수 있을 것으로 생각되며, Child 가 들어간 것과 들어가지 않은 것도 쉽게 알 수 있을 것이다. GetNext 와 GetPrev 는 Visible 프로퍼티와 자손 여부에 관계 없이 다음이나 이전 노드를 얻어오는 메소드이다. 이들 각각에 대한 자세한 설명은 도움말을 참고하기 바란다.

그러면, 실제로 IsDuplicate 함수를 사용해서 노드를 추가할 때 겹치는 이름이 발생하지 않도록 Button1 의 OnClick 이벤트 핸들러를 수정하도록 하자.

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Text: string;  
begin
```

... (중략)

```
InputQuery('새 노드', '이름 ?', Text);  
if IsDuplicate(TreeView1.Selected.GetFirstChild, Text, True) then  
begin  
    ShowMessage('같은 이름의 노드가 이미 존재합니다 !');  
    Exit;  
end;
```

... (후략)

이렇게 수정하면 InputQuery 를 이용해서 새로운 노드의 이름을 얻어온 후에 이 이름과 현재 선택된 노드의 첫번째 자식 노드를 IsDuplicate 함수의 파라미터로 넘겨주면 현재 선택된 노드의 모든 자식 노드에 대해 중복되는 이름이 있는지 검사하게 된다. 이 값이 True 라면 같은 이름의 노드가 존재하는 것이므로 새로운 노드를 추가하지 않고 끝내게 된다. 이것으로 첫번째 예제가 완성되었다. 실행하고, Add 와 Remove 버튼을 눌러서 아이템을 추가하고 삭제해보기 바란다.

## ● 트리 노드와 각종 데이터의 연결

각 트리 노드에는 Data 라는 프로퍼티가 있다. 이 프로퍼티를 이용하면 트리 노드와 여러 가지 데이터를 연결해서 사용할 수가 있다. TTreeNode 의 Data 프로퍼티는 기본적으로 untyped 포인터 데이터 형이기 때문에 어떤 객체도 지정할 수 있다. Data 프로퍼티를 사용하는 방법을 알아보자.

클래스를 사용해서 객체를 연결하는 방법은 델파이가 클래스 인스턴스를 실제로 포인터로 간주하는 객체 참조 모델을 사용하기 때문에 간단하면서도 매우 편리하게 사용할 수 있다. 객체 참조 모델에 대해서는 오브젝트 파스칼과 C++, 자바에 대해서 비교하고 있는 6 장의 내용을 참고하기 바란다. 다음의 코드를 살펴보자

```
var  
    SampleClass: TSampleClass  
begin  
    SampleClass: TSampleClass.Create;  
    SampleClass.Free;  
end;
```



앞에서 SampleClass 변수는 실제로는 포인터이다. 실제로 Data 프로퍼티에 사용하는 방법은 다음과 같다.

```
Node.Data := TSampleClass.Create;
```

```
TSampleClass(Node.Data).Free;
```

첫째 줄에서 TSampleClass 인스턴스를 생성하고 이를 TTreeNode.Data 프로퍼티에 대입한다. 두번째 줄에서 포인터를 형변환(type cast)하고 이를 메모리에서 해제한다. 이렇게 형변환이 필요한 이유는 Data 프로퍼티가 untyped 포인터이기 때문이다.

그러면 이를 이용한 예제를 하나 만들어 보자. 앞에서 만든 첫번째 예제를 Open 하고 이를 확장하겠다. Add 버튼을 누르면, 파일 이름을 선택할 수 있는 대화상자가 나타나고 여기에서 선택한 파일의 이름을 Data 프로퍼티에 저장하고 이를 보여주는 예제이다. 먼저 다음 그림과 같이 TLabel, TOpenDialog 컴포넌트를 올려 놓자.

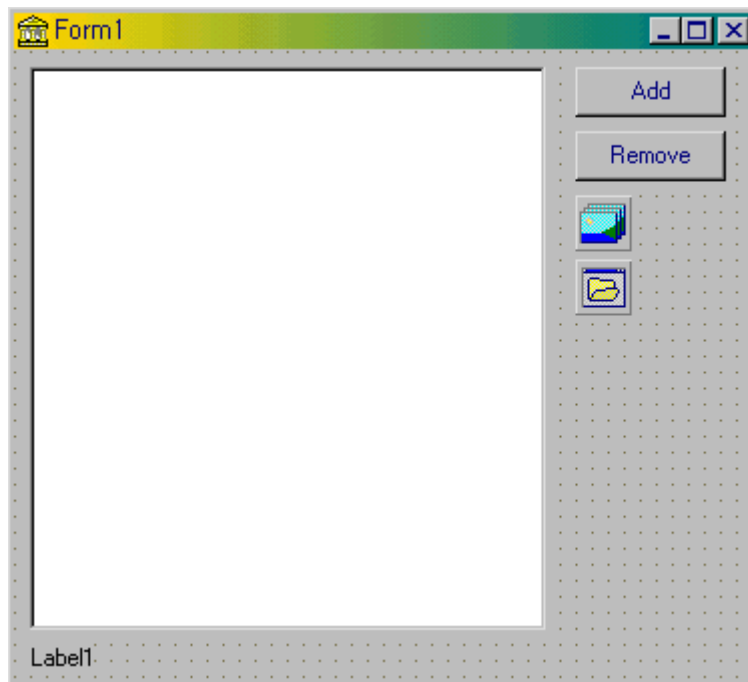
그리고, type 선언문에 사용할 데이터 클래스를 다음과 같이 선언해서 추가한다.

type

```
TNodeData = class
```

```
    Text: string;
```

```
end;
```



이제 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 수정한다.

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

... (중략)

```
    with TreeView1.Items.AddFirst(nil, 'Root') do
    begin
        Selected := True;
        Data := TNodeData.Create;
        TNodeData(Data).Text := '루트 디렉토리입니다 !';
```

... (중략)

```
    InputQuery('새 노드', '이름 ?', Text);
    OpenFileDialog1.Execute;
    if IsDuplicate(TreeView1.Selected.GetFirstChild, Text, True) then
    begin
        ShowMessage('같은 이름의 노드가 이미 존재합니다 !');
        Exit;
    end;
    with TreeView1.Items.AddChild(TreeView1.Selected, Text) do
    begin
        Data := TNodeData.Create;
        TNodeData(Data).Text := OpenFileDialog1.FileName;
```

... (후략)

처음 루트 노드를 추가할 때에는 Data 프로퍼티에 TNodeData 클래스의 인스턴스를 생성해서 대입하고, Text 멤버를 ‘루트 디렉토리 입니다 !’로 설정한다. 그리고, 다른 노드가 추가될 때에는 OpenFileDialog 대화 상자를 띄워서 여기서 선택되는 파일의 이름을 Text 멤버로 설정해준다.

주의해서 보아야 할 것은 Data 프로퍼티에 TNodeData.Create 라는 클래스 constructor 를 이용해서 초기화를 한 후에, 실제 값을 TNodeData(Data)로 casting 해서 접근하는テクニック이다.

이렇게 노드를 추가하면, 노드가 추가될 때마다 각 노드의 Data 프로퍼티가 TNodeData 클

래스의 인스턴스에 대한 참조값을 가지고 있게 되므로, 노드를 제거할 때에는 Data 프로퍼티에 저장된 값을 해제해 주어야 한다. 그러므로, Button2 의 OnClick 이벤트 핸들러도 다음과 같이 수정해 주어야 한다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    if TreeView1.Selected = nil then
    begin
        ShowMessage('선택된 것이 없습니다.');
```

Exit;

```
    end;
    if TreeView1.Selected.Level = 0 then
    begin
        ShowMessage('Root 는 제거할 수 없습니다');
```

Exit;

```
    end;
    if TreeView1.Selected.Data <> nil then
        TNodeData(TreeView1.Selected.Data).Free;
    TreeView1.Selected.Delete;
end;
```

이렇게 클래스를 이용하는 방법 외에 레코드를 사용하는 방법도 있다. 앞에서 보여준 TNodeData 클래스와 같은 내용을 레코드로 정의하고, 여기에 대한 포인터를 정의해서 사용하는 방법도 널리 쓰이고 있다. 예를 들어, 앞의 TNodeData 클래스와 동일한 일을 할 수 있도록 레코드를 다음과 같이 설정하고 사용할 수 있다.

```
pNodeData = ^TNodeData;
TNodeData = record
    Text: string;
end;
```

이렇게 하면 클래스 생성자를 사용하지 않고, 직접 레코드를 참조할 수 있는 장점이 있다. 다음과 같이 Data 프로퍼티에 적용해서 사용하면 된다.

```
Data := New(pNodeData);           //레코드 포인터에 대한 메모리를 할당한다.
pNodeData(Data)^.Text := '루트 디렉토리 입니다 !';
```

이렇게 노드에 데이터를 연결한 경우에는 메모리에서 해제할 때에도 다음과 같은 방법을 사용한다.

```
if TreeView1.Selected.Data <> nil then  
    Dispose(pNodeData(TreeView1.Selected.Data));
```

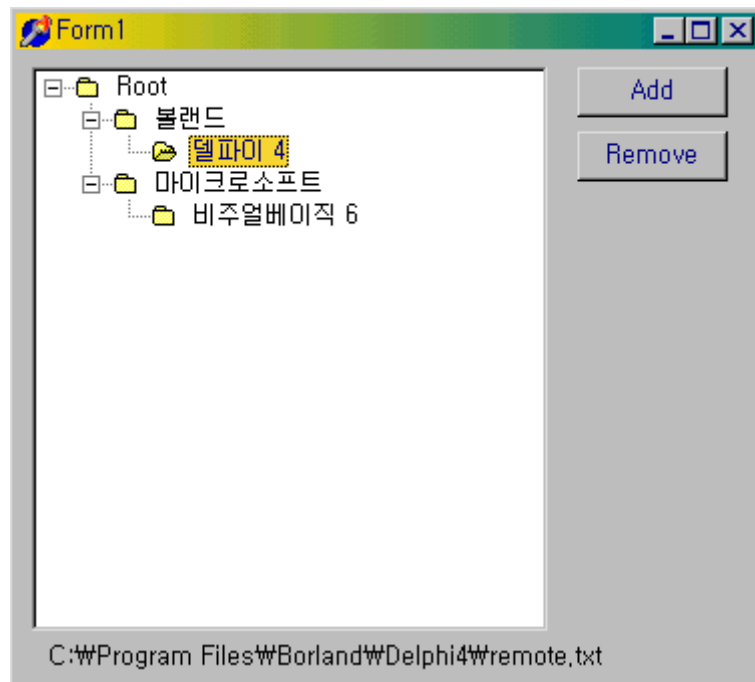
이제 거의 완성되었는데, 노드를 추가할 때마다 파일을 선택해서 이 값들을 Data 프로퍼티에 저장했으면, 트리뷰의 특정 노드를 선택했을 때에는 그 노드에 저장된 값을 볼 수 있도록 하자. 이를 위해서 TLabel 컴포넌트를 하나 폼에 추가했었다.

이렇게 트리뷰의 노드를 선택했을 때 발생하는 이벤트가 OnChange 이벤트이다. 그러면, TreeView1의 OnChange 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);  
begin  
    if TreeView1.Selected <> nil then  
        begin  
            if TreeView1.Selected.Data <> nil then  
                Label1.Caption := TNodeData(TreeView1.Selected.Data).Text;  
            end;  
        end;  
end;
```

소스가 그리 어려운 내용이 아니므로, 자세한 설명은 생략하기로 한다.

다음 그림은 이 프로그램의 실행화면이다.



## 리스트뷰 컴포넌트 시작하기

윈도우 95 에서 리스트뷰는 정말로 많이 사용되고 있다. 아마도 가장 자주 보게 되는 컨트롤이 이것이라고 생각되는데, 보통 윈도우 95 에서 파일을 보기 위해서 ‘내 컴퓨터’ 아이콘을 더블 클릭하면 나타나는 윈도우가 바로 리스트뷰로 구성되어 있다.

리스트뷰 컨트롤에서 중요한 프로퍼티에는 다음과 같은 것들이 있다.

프로퍼티	설 명
Items	TListItem 의 컬렉션인 TListItems 데이터 형 프로퍼티로, 리스트의 내용을 결정한다. 각각의 아이템은 Caption, ItemIndex, StateIndex 프로퍼티를 가진다. ItemIndex 와 StateIndex 프로퍼티는 ImageList 컴포넌트와 연결하여 사용된다.
SubItems	TListitem 의 프로퍼티로 TStringList 데이터 형이다. 리스트뷰를 Detail 모드로 보면, SubItems 는 Caption 프로퍼티의 우측 각 컬럼의 텍스트 내용을 결정한다.
ViewStyle	탐색기 보기 형식의 큰 아이콘(vslcon), 작은 아이콘(vsSmallIcon), 간단히(vslList), 자세히(vsReport)에 해당되는 보기 형식을 결정하는 프로퍼티이다.
Columns	TListColumn 의 컬렉션인 TListColumns 데이터 형 프로퍼티로, 리스트뷰를 Detail 모드로 볼 때의 각 컬럼의 형식을 결정한다.

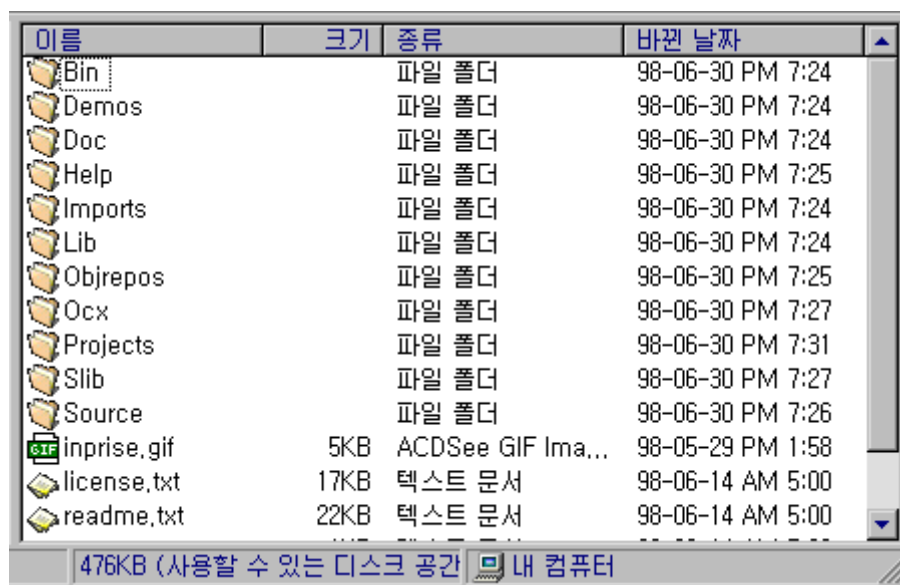
리스트뷰는 이들 프로퍼티를 활용하여 어플리케이션을 작성한다. 트리뷰 보다는 비교적 쉬운 구조라고 할 수 있다. 기본적인 사용방법은 유사하다. 예를 들어, Edit 박스의 텍스트 내용을 새로운 리스트 아이템으로 리스트뷰에 추가시키려면 다음과 같이 코딩하면 된다.

```

var
    ListItem: TListItem;
begin
    ListItem := ListView1.Items.Add;
    ListItem.Caption := Edit1.Text;

```

ViewStyle 프로퍼티가 vsReport 로 설정된 경우에는, 탐색기의 ‘자세히’ 보기 옵션을 선택한 경우를 생각하면 된다. 즉, 다음 그림과 같은 형식을 가진다.



여기에도가 새로운 컬럼을 추가하고자 하면 다음과 같은 식으로 코딩하면 된다.

```

var
    ListColumn: TListColumn;
begin
    ListColumn := ListView1.Columns.Add;
    ListColumn.Caption := Edit1.Text;
    ListColumn.Width := Length(Edit1.Text) * Font.Size;

```

여기에서 ListView.Items.Add 까지만 호출을 하여 ListItem 에 대한 프로퍼티만 설정한 경우에는 앞 그림의 파일 이름에 해당되는 것만 추가된 것이다. 우측의 3 가지 정보를 추가할 때에는 다음과 같이 SubItems 프로퍼티의 Add 메소드를 사용해야 한다.

```
ListView1.Selected.SubItems.Add('서브 아이템 추가 !');
```

## 윈도우 95 스타일의 파일 리스트뷰 어플리케이션 제작

그러면, 리스트뷰를 기본 컴포넌트로 하여 이미지 리스트, 상태바 등을 활용한 윈도우 95 스타일의 파일 리스트뷰 어플리케이션을 제작해 보자.

지금까지 만들어본 예제 프로그램 중에서 600 라인이 넘는 비교적 큰 프로그램이기 때문에, 지면 관계상 이제까지의 방법처럼 차례차례 따라하도록 설명할 수는 없다. 소스는 Chap11 디렉토리의 Exam3.dpr 파일을 열면 모두 볼 수 있다. 여기서는 리스트뷰를 중심으로 필요한 부분만 설명하겠다. 소스에 주석을 달아놓았으므로 나머지 부분은 직접 소스를 참고하기 바란다.

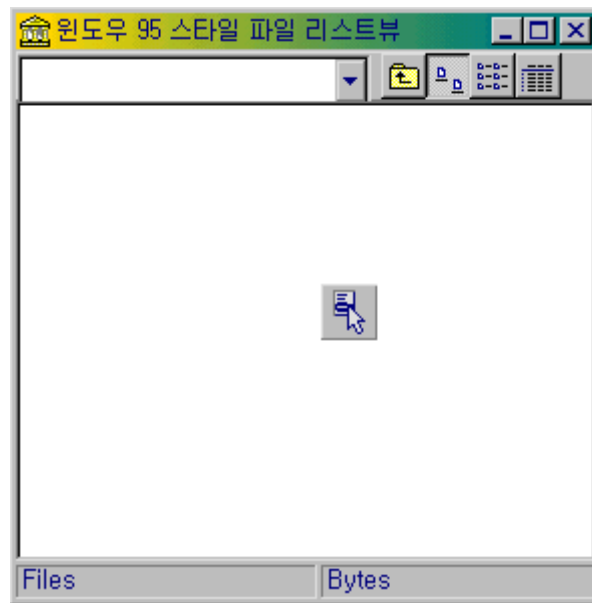
여기에서 설명하지는 않지만, 소스에는 콤보 박스의 활용 방법과 콤보 박스에 이미지를 넣는 방법, 바이트 수를 KB 나 MB 단위로 바꾸는 유틸리티 함수 등의 많은 테크닉이 사용되었으므로 좋은 공부 재료가 될 것이다. 이 예제는 Markus Stephany([MirBir.St@T-Online.de](mailto:MirBir.St@T-Online.de))가 공개한 소스 코드에 기초한 것임을 미리 밝혀둔다. Markus Stephany 는 자신의 소스 코드를 공개하면서 Brad Stower 의 코드를 이용한 것이라고 밝혔으니, 결국에는 Brad Stower 에게 가장 감사해야 할 것 같다 (어쩌면 Brad Stower 역시 다른 사람의 소스를 참고했는지 모르지만).

파일의 정보를 얻기 위해서는 TSHFileInfo 구조체와 SHGetFileInfo 함수를 사용하는데 이들은 Win32 셸의 기능을 이용하는 것이다. 이 어플리케이션을 실행하면 ‘내 컴퓨터’를 비롯한 여러가지 폴더를 더블 클릭하여 보는 모습과 거의 유사한 형태를 가진다. 그렇지만, 본질적으로는 조금 다른데 윈도우 탐색기를 비롯한 윈도우 95 의 셸 인터페이스는 NameSpace 라고 불리는 영역을 사용하며, 모든 윈도우 객체를 가상 객체로 다루게 된다. 그렇기 때문에 전통적인 디렉토리-파일 개념의 접근과는 차이가 있다. 여기에 대해서는 이 책의 후반부에서 다룰 기회가 있을 것이다 (이 장을 집필하는 도중의 계획으로는).

쉽게 말해, 가상 폴더에는 프린터나 제어판 등의 실제 디렉토리가 아닌 것들에 대한 접근이 가능하지만, 이번 장의 파일 리스트뷰에서는 전통적인 디렉토리-파일 접근 방법을 사용하므로 이들을 볼 수는 없다.

### ● 폼디자인

이 예제의 폼의 형태는 다음과 같다.



패널 하나를 기본적으로 폼에 올려 놓고 Align 프로퍼티를 alClient 로 설정하여 전체를 담을 수 있도록 하고, Panel2 를 Panel1 위에 올려 놓고 Align 프로퍼티를 alTop 으로 설정한 후 콤보 박스 1 개와 스피드 버튼 4 개를 패널에 정렬한다. 그리고 상태바 컴포넌트를 Panel1 에 올려 놓고, Align 프로퍼티를 alBottom 으로 설정한다. 여기에 리스트뷰 컴포넌트를 추가하고 Panel1 위에서 Align 프로퍼티를 alClient 로 설정하면 전체적인 모양이 완성된다. 스피드 버튼 4 개의 힌트를 보여주기 위해서, ShowHint 프로퍼티를 True 로 설정하고 각각의 비트맵을 적당하게 대입하고, 이들의 Hint 프로퍼티를 각각 ‘위로’, ‘큰 아이콘’, ‘작은 아이콘’, ‘자세히’로 설정한다.

마지막으로 TPopupMenu 컴포넌트를 폼에 올려 놓고, 이 컴포넌트를 더블 클릭한 후 MenuItem 을 추가한다. 이 메뉴 아이템의 Caption 을 ‘파일 정보’로 설정한다. 이제 사용할 컴포넌트는 모두 추가하였다. 이들의 Name 프로퍼티를 다음과 같이 설정한다.

컴포넌트	Name 프로퍼티	컴포넌트	Name 프로퍼티
리스트뷰	Files	상태바(Status Bar)	FState
팝업 메뉴	Menu	메뉴 아이템	MenuItem
콤보 박스	ComboBox1	위로 스피드 버튼	SpeedButton1
큰 아이콘 스피드 버튼	SpeedButton2	작은 아이콘 스피드 버튼	SpeedButton3
자세히 스피드 버튼	SpeedButton4		

FState 상태바를 선택하고, 오브젝트 인스펙터에서 Panels 프로퍼티를 더블 클릭하여 프로퍼티 에디터를 띄우고, 2 개의 패널을 추가한다. 각 패널의 Text 프로퍼티는 ‘파일’, ‘바이트’로 설정한다. 이제 폼 디자이너에서 해야할 일은 모두 다 끝났다.



- 전역 변수의 선언과 유틸리티 함수

이제 코딩을 할 차례인데, 먼저 파일과 이미지 리스트에 대한 처리를 하기 위해서 uses 절에 FileCtrl.pas 와 ImgList.pas 유닛을 추가한다. 또한, 쉘의 여러가지 처리를 위해서는 ShellAPI.pas 유닛도 추가해 주어야 한다.

그리고, 전역 변수를 다음과 같이 선언한다.

```
Smalls, Larges: TImageList;
```

```
Path: TFileName; //현재 디스플레이되는 패스
```

```
Drives: set of 0..25; //드라이브에 대한 플래그
```

Smalls 와 Larges 전역 변수는 파일 리스트뷰에서 사용할 이미지 리스트의 내용을 담게될 전역 변수이다. Path 는 현재 디스플레이되는 패스의 내용을 반영한다. Drives 변수는 알파벳 A~Z 까지를 드라이브로 사용할 수 있다고 볼 때, 이를 정수형의 세트로 선언하여 사용한다.

그리고, 이 예제에서는 몇 가지 유틸리티 함수를 예제 내에 포함시켜 사용하고 있는데 이들을 정리하면 다음과 같은 것들이 있다. 구체적인 구현 방법은 소스 코드를 참고하기 바란다.

```
1. function GetKb(c: Integer): string;
```

이 함수는 파라미터로 넘겨주는 정수형을 ‘KB’, ‘MB’ 단위로 계산하여 이를 문자열의 형태로 반환하는 함수이다.

```
2. function GetSize(c, typ: Integer): string;
```

파일을 ‘ KB’ 크기 단위로 변환하여 문자열로 반환하는 함수이다. 파라미터 c 는 크기를 담게 되고, typ 파라미터에는 디렉토리나 파일의 종류를 넘겨주게 된다. 여기서 faDirectory 와 같이 디렉토리가 넘어가면 빈 문자열을 반환한다.

```
3. function GetMod(a: TFileTime): string;
```

TFileTime 데이터 형의 내용을 윈도우 탐색기에서 볼 수 있는 형태의 문자열로 변환해주는 함수이다. 최근에 변경된 내용에 대한 정보는 SHGetFileInfo 함수를 이용해서 얻을 수 있다.

4. function GetCount(a: Char; b: string): Integer;

문자열에 특정 문자가 몇 개 있는지 파악해서 그 수를 반환하는 함수이다. 이 함수가 사용되는 것은 패스를 문자열로 넘겼을 때 서브 디렉토리가 몇 개 있는지를 파악하기 위해서 사용된다. 예를 들어 'c:\Program Files\Borland\Delphi 4\Demos\...' 이라고 하면 'W' 문자가 5 개 존재하므로 GetCount 함수에 'W'자와 이 문자열을 파라미터로 넘기면 5 를 반환하게 된다.

5. function NumPos(a: Char; b: String; c: Integer): Integer;

이 함수는 앞의 GetCount 와 함께 연관되어 사용되는데, 첫번째 파라미터에 찾을 문자와 두 번째 파라미터에 문자열을 넘겨주고, 마지막 파라미터로 몇 번째 것의 위치를 지정할 것인지를 결정하면 문자열에서의 특정 문자의 위치를 반환한다. 이 함수는 패스를 문자열로 사용하되, 이들을 이용해서 서브 디렉토리와 상위 디렉토리를 오갈 수 있도록 처리하기 위해서 사용하게 된다. 예를 들어 패스가 'c:\Program Files\Borland\Delphi 4\W' 디렉토리가 현재의 패스라고 할 때, '위로' 버튼을 누르면 'c:\Program Files\Borland\W'가 패스로 변경되어야 한다. 이때 NumPos('W', 'c:\Program Files\Borland\Delphi 4\W', 3)을 호출하면 'c:\Program Files\Borland\W' 문자까지의 위치가 반환되므로, 여기까지만 복사해서 지정하면 된다.

6. procedure GetInfo(FileName: TFileName; var i: TSHFileInfo);

이 함수는 파일 이름이나 디렉토리 이름과, TSHFileInfo 형으로 선언된 변수를 파라미터로 넘기면 이 파일이나 디렉토리의 정보를 TSHFileInfo 형 변수에 담아서 반환하게 된다.

#### ● 리스트뷰 컬럼의 설정

우선 '내 컴퓨터'와 같이 최상위 위치로 올라갔을 때와 디렉토리에 담긴 파일 들을 보여줄 때 리스트뷰의 컬럼의 설정이 달라지게 된다. '내 컴퓨터'의 위치라면 각 디스크 드라이브의 이름 이외에, 드라이브의 종류와 디스크 크기, 남은 공간을 나타내면 될 것이다. 이런 프로시저를 SetColumnForDrives 라고 선언하고, 이를 다음과 같이 구현한다.

```
procedure TForm1.SetColumnForDrives;
```

```
    //리스트뷰의 컬럼 헤더를 드라이브 특성에 맞게 설정한다.
```

```
begin
```

```

with Files.Columns do
begin
    Clear;
    with Add do
    begin
        Caption := '이 름';
        Width := ColumnHeaderWidth;
        Alignment := taLeftJustify;
    end;
    with Add do
    begin
        Caption := '종 류';
        Width := ColumnHeaderWidth;
        Alignment := taLeftJustify;
    end;
    with Add do
    begin
        Caption := '디스크 크기';
        Width := ColumnHeaderWidth;
        Alignment := taRightJustify;
    end;
    with Add do
    begin
        Caption := '남은 공간';
        Width := ColumnHeaderWidth;
        Alignment := taLeftJustify;
    end;
end;
end;
end;

```

마찬가지로, 디렉토리를 선택했을 때에는 각각의 파일에 대한 컬럼을 설정해야 할 것이다. 여기서는 이름과 크기, 종류, 바뀐 날짜를 설정해야 할 것이다. 이 프로시저는 SetColumnForFiles 라고 선언하고, 이를 다음과 같이 구현한다.

```

procedure TForm1.SetColumnForFiles;
    //리스트뷰의 컬럼 헤더를 설정한다.

```

```

begin
  with Files.Columns do
    begin
      Clear;
      with Add do
        begin
          Caption := '이 름';
          Width := ColumnHeaderWidth;
          Alignment := taLeftJustify;
        end;
      with Add do
        begin
          Caption := '크 기';
          Width := ColumnHeaderWidth;
          Alignment := taRightJustify;
        end;
      with Add do
        begin
          Caption := '종 류';
          Width := ColumnHeaderWidth;
          Alignment := taLeftJustify;
        end;
      with Add do
        begin
          Caption := '바뀐 날짜';
          Width := ColumnHeaderWidth;
          Alignment := taLeftJustify;
        end;
      end;
    end;
  end;
end;

```

## - 리스트뷰에 아이템 추가하기

헤더 컬럼을 설정할 때와 마찬가지로, 내 컴퓨터와 같이 드라이브를 나열할 때와 디렉토리에서 파일을 나열할 때의 구현 방법이 다르다. 먼저 드라이브의 내용을 나열하는 SetDrives 프로시저를 선언하고, 이를 다음과 같이 구현한다.

```
procedure TForm1.SetDrives: //리스트뷰에 시스템의 드라이브를 보여 준다.
```

```
var
```

```
    ct: Byte;
```

```
    new: TListItem;
```

```
    info: TSHFileInfo;
```

```
    drv: string;
```

new 변수에는 추가한 리스트 아이템을 대입하여, 여러가지 프로퍼티를 설정하도록 할 것이며, TSHFileInfo 구조체 형인 info 변수는 GetInfo 함수를 호출하여 드라이브에 대한 정보를 담게 된다. drv 변수에는 드라이브의 이름을 저장한다.

```
const
```

```
    drt: array[0..6] of string = ('알 수 없는 장치', '루트 디렉토리 없음', '플로피 디스크',  
    '로컬 드라이브', '네트워크 드라이브', 'CD-ROM 드라이브', 'RAM 디스크');
```

drt 상수는 GetDriveType API 함수를 호출했을 때 반환하는 값이 0~6 까지의 정수이기 때문에 문자열로 쉽게 보여줄 수 있도록 선언하였다.

```
begin
```

```
    SpeedButton1.Enabled := False;
```

```
    Files.Items.BeginUpdate;
```

```
    Files.Items.Clear;
```

```
    SetColumnForDrives;
```

```
    Screen.Cursor := crHourGlass;
```

여기까지의 코드는 그다지 어려울 것이 없다. SpeedButton1 은 상위 레벨로 올라가는 버튼이므로, 현재 더이상 올라갈 것이 없는 상황에서는 Enabled 프로퍼티를 False 로 설정해야 할 것이다. 그리고, BeginUpdates 메소드는 아이템의 업데이트를 시작할 때 성능을 향상시켜주는 메소드로, 리스트뷰 외에도 트리뷰를 비롯한 여러가지 컨트롤에 있는 메소드이다. 항상 컨트롤을 업데이트할 때에는 이 메소드를 호출하는 것을 습관화하는 것이 좋다.

```
for ct := 0 to 25 do //모든 드라이브를 읽어 온다.
```

```
    if ct in Drives then
```

```
        begin
```

```
            new := Files.Items.Add;
```

```

drv := Char(ct + Ord('A')) + ':\W';           //드라이브의 루트 패스
GetInfo(drv, info);                           //드라이브의 쉘 정보를 읽어온다
new.Caption := info.szDisplayName;
new.ImageIndex := info.ilcon;
new.SubItems.Add(drt[GetDriveType(PChar(drv))]);
new.SubItems.Add(GetKB(DiskSize(ct + 1)));
new.SubItems.Add(GetKB(DiskFree(ct + 1)));
new.SubItems.Add('드라이브');                //아이템의 종류 결정
new.SubItems.Add(drv);                       //OnDoubleClick 이벤트 핸들러를 위해
end;

```

컴퓨터에 있는 모든 드라이브를 for 루프를 통해 읽어와서, 이 드라이브가 Drives 에 있는지 검사한다. Drives 변수는 폼의 OnCreate 이벤트 핸들러에서 컴퓨터에 사용가능한 드라이브를 모두 저장하게 된다. new 변수는 새롭게 추가된 리스트 아이템을 저장해서 처리하며, drv 변수는 드라이브의 루트 패스 문자열로 지정한다. 일단 드라이브 루트 패스 문자열을 저장했으면, GetInfo 함수를 호출하여 TSHFileInfo 데이터 형인 info 변수에 드라이브의 정보를 담아 온다. 이 구조체의 szDisplayName 멤버는 드라이브의 이름이 저장되어 있으며, ilcon 멤버에는 시스템 이미지 리스트의 이미지 인덱스 값이 지정되어 있다. 폼의 OnCreate 이벤트 핸들러에서 시스템 이미지 리스트를 Smalls, Larges 이미지 리스트 변수에 저장하게 되므로, 이 인덱스 값을 그대로 이용할 수 있다.

SubItems 프로퍼티를 5 개 Add 하였는데, 순서대로 드라이브의 종류와 디스크의 크기, 남은 디스크 용량, 아이템의 종류(드라이브), 드라이브 루트 패스 문자열이 저장된다. 이들은 각각 SubItems[0] ~ SubItems[4]로 접근이 가능하다. 예를 들어 SubItems[3]에는 아이템의 종류인 '드라이브'가 들어 있고, SubItems[4]에는 드라이브의 루트 패스 문자열이 저장된다.

```

Files.Items.EndUpdate;
Screen.Cursor := crDefault;
FState.Panels[0].Text := IntToStr(Files.Items.Count) + ' 객체';
FState.Panels[1].Text := '';
ComboBox1.Update;
end;

```

리스트뷰의 변경이 끝났으면, 내용을 EndUpdate 메소드를 호출하여 업데이트 한다. 커서의 모양을 다시 원래의 형태로 바꿔 주고, 드라이브의 수를 상태바의 첫번째 패널에 표시한다. 마지막으로 콤보 박스의 내용을 업데이트하도록 콤보박스의 OnUpdate 이벤트 핸들러

를 호출한다.

비슷한 방식으로 먼저 디렉토리의 내용을 나열하는 SetFiles 프로시저를 선언하고, 이를 다음과 같이 구현한다. 이 프로시저는 SetDrives 프로시저의 호출을 포함하기 때문에, 다른 이벤트 핸들러에서는 SetFiles 프로시저만 호출해주면 된다.

```
procedure TForm1.SetFiles;
```

```
var
```

```
    ct: Integer;
```

```
    res: Word;
```

```
    rec: TSearchRec;
```

```
    new: TListItem;
```

```
    info: TSHFileInfo;
```

```
    oldstyle: TViewStyle;
```

로컬 변수의 선언부에서 SetDrives 프로시저와 다른 몇 개의 변수가 있다. res 변수는 파일을 나열할 때 사용하는 FindFirst, FindNext 함수의 결과값을 저장하는 변수로, 값이 0 이면 해당되는 파일이 있다는 의미가 된다. rec 변수 역시 FindFirst, FindNext 함수와 함께 사용되는 변수로, 9 장에서 이미 이들에 대해 설명한 바 있으므로 자세한 내용은 생략하겠다. oldStyle 변수는 현재의 리스트뷰 보기 스타일을 저장했다가 나중에 다시 복구하기 위해서 선언한 변수이다. ct 변수는 SetDrives 와는 달리 모든 파일 크기의 총합을 내어 저장하는 변수로 사용된다.

```
begin
```

```
    Caption := '윈도우 95 형 파일 리스트뷰 [' + Path + ']';
```

```
    if Path = 'Drives' then
```

```
        begin                                //드라이브를 보여주어야 한다면 ...
```

```
            SetDrives;
```

```
        end
```

```
    else
```

```
        begin
```

```
            SpeedButton1.Enabled := True; //드라이브를 보여주는 것이 아니면, 더 상위로 올라갈 수 있다.
```

여기까지의 코드는 그다지 어렵지 않을 것이다. 우선 폼의 캡션에 패스를 같이 표시하고, 드라이브를 보여주어야 하는 경우라면 SetDrives 프로시저를 호출한다.

```
    Files.Items.BeginUpdate;
```

```

oldstyle := Files.ViewStyle;
Files.ViewStyle := vsList;      //속도가 빠르다고 한다.
Files.Items.Clear;
Screen.Cursor := crHourGlass;
if Path[Length(Path)] <> 'W' then Path := path + 'W';

```

리스트뷰를 업데이트 하기 위해서 우선 BeginUpdate 메소드를 호출하고, ViewStyle 프로퍼티를 vsList 로 설정한다. 이 프로퍼티를 변경하는 것은 업데이트 속도가 가장 빠르기 때문이다. 이를 위해 oldStyle 변수에 현재의 ViewStyle 을 저장했다가, 나중에 복구한다. 그리고, 패스의 마지막 문자가 'W'가 아니면 이를 추가한다.

다음의 코드는 디렉토리에 있는 볼륨 ID 와 '.'과 '..' 디렉토리를 제외한 모든 파일과 디렉토리를 검색해서 이들의 정보를 리스트뷰의 아이템으로 추가하는 코드이다. 9 장에서 설명한 FindFirst, FindNext 함수의 사용하여 구현한다.

```

FillChar(rec, SizeOf(TSearchRec), 0);
ct := 0;
res := FindFirst(Path + '*,*', faAnyFile - faVolumeld, rec);
while res = 0 do
begin
    //결과가 0 이 아니면 더 이상 파일이 없는 것임
    //'.'과 '..'은 제외한다.

    if not (((rec.Attr and faDirectory) > 0) and
        ((rec.Name = '.') or (rec.Name = '..'))) then
    begin
        new := Files.Items.Add;
        GetInfo(Path + rec.Name, info); //파일이나 폴더의 쉘 정보를 얻어온다.
        new.Caption := info.szDisplayName;
        new.ImageIndex := info.ilcon;
        new.SubItems.Add(GetSize(rec.Size, rec.Attr));
        new.SubItems.Add(info.szTypeName);
        new.SubItems.Add(GetMod(rec.FindData.ftLastWriteTime));
        if (rec.Attr and faDirectory) > 0 then new.SubItems.Add('디렉토리')
        else new.SubItems.Add('파일');
        new.SubItems.Add(Path + rec.Name);
        Inc(ct, rec.Size);          //파일 크기를 더한다.
    end;
    res := FindNext(rec);

```



```
end;
FindClose(rec);
```

새로운 아이টে를 추가할 때, 유틸리티 함수인 GetSize, GetMod 함수를 이용하여 파일이나 디렉토리의 크기와 변경된 시간을 문자열로 얻어오게 되며, 마지막으로 4 번째로 Add 하는 SubItems[3]에 해당되는 내용은 디렉토리이면 ‘디렉토리’, 파일이면 ‘파일’로 설정한다. 그리고, 각 파일이 크기를 계속 더해서 디렉토리의 파일 크기의 합을 표시하는데 사용한다.

```
Files.CustomSort(@SortProc, 4);
Files.ViewStyle := oldstyle;
Files.Items.EndUpdate;
FState.Panels[0].Text := IntToStr(Files.Items.Count) + ' 객체';
FState.Panels[1].Text := GetKB(ct);
Screen.Cursor := crDefault;
ComboBox1Update;
end;
end;
```

디렉토리를 파일에 우선하여 배열하도록 하기 위해 사용자 정의 정렬 루틴을 먼저 호출한다. 정렬하는 방법에 대해서는 다시 설명할 것이다. 그리고, ViewStyle 프로퍼티의 값과 커서의 값을 복구하고, 상태바의 내용을 업데이트 한다. 마지막으로 콤보 박스의 내용을 업데이트 한다.

## ● 사용자 정의 정렬 루틴의 활용

리스트뷰 이외에도 동적으로 값을 포함하게 되는 다른 여러가지 컨트롤(리스트 박스, 콤보 박스, 트리 뷰 등)에서 정렬 루틴은 자주 사용되는 중요한 부분이다. 보통 다른 컨트롤 처럼 아이টে의 텍스트 만을 정렬하면 되는 경우에는 컨트롤의 AlphaSort 메소드를 이용하여 아스키 배열의 순서대로 간단하게 정렬을 할 수 있다. 그렇지만, 객체를 포함하고 있는 컨트롤이거나 다른 정보를 가지고 있어서 이들의 내용에 따라서 다르게 정렬을 해야할 경우에는 개발자가 직접 정렬에 대한 코드를 제공해야 한다.

이런 정렬 루틴을 사용자 정의 정렬(CustomSort)라고 한다.

사용자 정의 정렬 루틴은 기본적으로 사용 방법이 어느 컨트롤이나 동일하다. 일단 정렬 루틴을 작성해야 하는데, 이때 이 루틴은 콜백 함수로 사용된다. 콜백 함수에 대해서는 44 장에서 자세하게 설명하므로 이를 참고하기 바란다.

간단하게 설명하면 파라미터의 수와 데이터 형이 동일한 함수를 작성하고, 이 함수의 주소

를 CustomSort 메소드에 넘겨주면 된다. CustomSort 메소드는 2 개의 파라미터를 가지는데, 첫번째 파라미터에는 정렬 루틴으로 사용될 콜백 함수의 주소를 지정하고, 두번째 파라미터에는 옵션에 해당되는 내용을 설정하게 된다. 간단하게 사용 방법의 예를 들면 다음과 같다.

```
function CustomSortProc(Item1, Item2: TListItem; ParamSort: integer): integer; stdcall;
begin
    Result := -Istrcmp(PChar(TListItem(Item1).Caption), PChar(TListItem(Item2).Caption));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ListView1.CustomSort(@CustomSortProc, 0);
end;
```

각각의 컨트롤마다 CustomSort 의 파라미터의 내용은 다를 수 있기 때문에, 도움말을 참고하여 적절한 루틴을 작성해야 할 것이다.

그러면, 이번 예제에서 작성한 정렬 루틴을 살펴 보자.

윈도우 95 스타일의 리스트뷰이므로, 정렬 루틴에서 고려해야 할 것은 ‘자세히’ 보기 스타일인 vsReport 로 ViewStyle 프로퍼티가 설정된 경우, 파일의 이름 외에도 크기와 종류, 변경된 날짜에 대한 컬럼 헤더를 클릭하면 이들의 속성에 따라 정렬하도록 해야 한다는 것이다. 이때 Path 변수의 내용이 ‘Drives’라면 디렉토리에 대한 내용이 아니므로 정렬 루틴을 호출하지 않아야 한다. 그리고, 윈도우 탐색기처럼 컬럼 헤더를 반복해서 클릭하면 오름차순과 내림차순으로 번갈아가면서 정렬하도록 하기 위해서는 이들의 정보를 가지고 있을 전역 변수를 하나 선언해서 사용해야 할 것이다.

전역 변수 ReverseOrder 를 전역 변수 선언부(Form1 등의 변수가 선언된 부분)에 다음과 같이 추가한다.

```
ReverseOrder: array[0..4] of Boolean;
```

이 변수는 각각의 정렬이 오름차순인지, 내림차순인지를 결정한다. 배열로 선언한 이유는 각각의 정렬 기준에 따라서 오름차순 여부를 다르게 설정할 수 있도록 하기 위해서이다. 정렬할 요소는 0~4 까지 5 가지이다. 0 번에 해당되는 것은 파일의 이름이다. 이것은 아이템의 캡션에 해당된다. 1~3 에 해당되는 3 가지는 컬럼 헤더의 3 가지가 대응된다. 즉, 1 번은 SubItems[0]인 파일의 크기, 2 번은 SubItems[1]인 파일의 종류, 3 번은 SubItems[2]인 바뀐 날짜가 해당된다. 그렇다면, 마지막 4 번은 무엇일까 ? 컬럼 헤더에 나타나 보이

지는 않지만 디렉토리인지 파일인지 여부에 따라서 결정하는 것이다. 그러므로, SetFiles 프로시저에서 CustomSort(@SortProc, 4)의 의미는 디렉토리를 먼저 보여주고, 파일을 보여주라는 의미인 것이다.

처음에 파일을 보여줄 때 ReverseOrder 의 값을 초기화 할 필요가 있으므로, SetFiles 프로시저에 다음과 같은 코드를 추가한다.

```
procedure TForm1.SetFiles;      //리스트뷰에 파일과 폴더를 보여준다.
```

```
var
```

```
    ... (중략)
```

```
begin
```

```
    for i := Low(ReverseOrder) to High(ReverseOrder) do
```

```
        ReverseOrder[i] := False;
```

```
    ... (후략)
```

그리고, 컬럼 헤더를 클릭할 때마다 이를 이용해서 정렬하도록 호출해야 할 것이므로 Files 리스트뷰 컴포넌트의 OnColumnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FilesColumnClick(Sender: TObject; Column: TListColumn);
```

```
begin
```

```
    if not (Path = 'Drives') then
```

```
    begin
```

```
        Files.CustomSort(@SortProc, Column.Index);
```

```
        ReverseOrder[Column.Index] := not ReverseOrder[Column.Index];
```

```
    end;
```

```
end;
```

즉, 패스가 디렉토리인 경우에 클릭한 컬럼의 인덱스를 넘겨서 컬럼 내용에 따라 정렬하고, 정렬 순서의 내용을 변경한다.

그러면, 실제 정렬을 담당할 콜백 함수를 작성하도록 하자. 기본적으로 파라미터의 수와 데이터 형만 일치하면 되므로, 함수의 이름이나 파라미터의 이름은 아무런 상관이 없다. 다음과 같이 콜백 함수를 작성한다.

```
function SortProc(I1, I2: TListItem; ColumnNo: Integer): Integer; stdcall;
```

```
var
```

```
    i: Integer;
```

```
s1, s2: string;
```

```
ds1, ds2, ts1, ts2: string;
```

내부에서 사용할 지역 변수를 먼저 선언한다. *i* 는 결과 값을 임시로 저장하기 위한 것으로, 콜백 함수 마지막 부분에서 ReverseOrder 변수의 값에 따라 이 값의 부호를 바꿔서 결과 값을 반환한다. 결과 값은 보통 1, 0, -1 중 하나를 반환하는데, 2 개의 아이템의 값이 같으면 0 을 반환하고, 첫번째 아이템이 더 크면 1(사실 양수면 된다), 작으면 -1(음수면 된다) 을 반환한다.

*s1*, *s2* 는 코드의 중복을 막기 위해 임시로 문자열을 저장하기 위해 사용되며, *ds1*~*ts2* 는 날짜와 시간을 비교하기 위해서 사용된다. ColumnNo 파라미터로 클릭한 헤더의 인덱스가 넘어오므로, 이를 바탕으로 정렬 루틴을 다르게 작성해야 할 것이다.

```
begin
```

```
  i := 0;
```

```
  case ColumnNo of
```

```
    0: i := CompareStr(I1.Caption, I2.Caption);
```

ColumnNo 가 0 이면 ‘이 름’ 헤더를 클릭한 경우이므로 아이템의 캡션을 서로 비교하면 된다. 이와 같이 CompareStr 함수를 사용하면 두 문자열을 비교하여 1, 0, -1 을 반환하므로, 코딩이 편하다.

```
  1: begin
```

```
    s1 := I1.SubItems[0];
```

```
    s2 := I2.SubItems[0];
```

```
    if s1 = '' then s1 := '0 KB';
```

```
    if s2 = '' then s2 := '0 KB';
```

```
    s1 := Copy(s1, 0, Length(s1) - 3);
```

```
    s2 := Copy(s2, 0, Length(s2) - 3);
```

```
    if StrToInt(s1) > StrToInt(s2) then i := 1 else i := -1;
```

```
    if StrToInt(s1) = StrToInt(s2) then
```

```
      i := CompareStr(I1.Caption, I2.Caption);
```

```
    end;
```

‘크 기’ 헤더를 클릭한 경우이다. 먼저 *s1*, *s2* 변수에 아이템의 문자열을 대입한다. 그리고, 디렉토리인 경우에는 문자열의 내용이 비어있게 되므로 값을 ‘0 KB’로 설정한다.

Copy 함수를 사용하여 뒤의 3 자를 잘라낸다. 즉, ‘ KB’를 문자열에서 삭제한다. 이제 숫

자만 남게 되었으므로 이들을 StrToInt 함수를 이용하여 정수로 변환하여 비교한다. 만약 크기가 같다면 파일 이름 순으로 정렬한다.

```
2: begin
    i := not CompareStr(I1.SubItems[1], I2.SubItems[1]);
    if (I1.SubItems[1] = I2.SubItems[1]) then
        i := CompareStr(I1.Caption, I2.Caption);
    end;
```

‘중 류’ 헤더를 클릭한 경우로, 단순히 SubItems[1]의 문자열을 직접 비교하면 된다. 종류가 같은 경우에는 아이템의 캡션을 비교한다.

```
3: begin
    s1 := I1.SubItems[2];
    s2 := I2.SubItems[2];
    ds1 := Copy(s1, 0, Pos(' ', s1));
    ds2 := Copy(s2, 0, Pos(' ', s2));
    ts1 := Copy(s1, Pos(' ', s1) + 2, Length(s1) - Pos(' ', s1) - 1);
    ts2 := Copy(s2, Pos(' ', s2) + 2, Length(s2) - Pos(' ', s2) - 1);
    if Length(ts1) = 7 then Insert('0', ts1, 4);
    if Length(ts2) = 7 then Insert('0', ts2, 4);
    if Copy(ts1, 4, 2) = '12' then
        begin
            ts1[4] := '0';
            ts1[5] := '0';
        end;
    if Copy(ts2, 4, 2) = '12' then
        begin
            ts2[4] := '0';
            ts2[5] := '0';
        end;
    i := CompareStr(ds1, ds2);
    if ds1 = ds2 then i := CompareStr(ts1, ts2);
end;
```

‘바뀐 날짜’ 헤더를 클릭한 경우이다. SubItems[2] 문자열을 s1, s2로 대입한 후, 이 문자

열을 다시 날짜와 시간 부분으로 쪼갠 후 비교한다. 이때 시간 부분의 길이를 맞추기 위해 시간이 한 개의 문자로 되어 있는 부분에는 앞에 '0'을 추가한다(예: AM 1:23 은 AM 01:23 으로). 그리고, 12 시인 경우에는 00 시로 변경해야 올바른 정렬이 된다.

이 루틴은 과거에 필자가 사용하던 처리 루틴을 그대로 가져와서 썼기 때문에 다소 복잡하다. 그렇지만, 아마도 잘 찾아보면 시간과 날짜, 문자열을 잘 변환해서 비교할 수 있는 루틴이 아마도 존재할 것이다. 필자가 이 부분을 수정하지 않은 이유는 기본적으로 잘 동작하기 때문이다 (물론 필자가 게으른 탓도 있다). 시간이 되는 독자는 이 부분을 조금 우아하게 수정해보기 바란다.

```
4: begin
    i := CompareStr(I1.SubItems[3], I2.SubItems[3]);
    if (I1.SubItems[3] = I2.SubItems[3]) then
        i := CompareStr(I1.Caption, I2.Caption);
    end;
```

이 부분은 SetFiles 프로시저에 의해 파라미터가 4 가 넘어온 경우에만 실행된다. 디렉토리를 파일 보다 먼저 정렬하는 역할을 하는데, 역시 문자열을 비교하여 정렬한다.

```
end;
if ReverseOrder[ColumnNo] then i := -i;
Result := i;
```

마지막으로 ReverseOrder 변수 배열의 해당 필드의 내용을 검사하여, True 이면 값의 부호를 변경하고 이를 반환한다.

#### ● 시스템 이미지 리스트 얻는 방법과 패스 변경의 구현

마지막으로 패스의 변경과 시스템 이미지 리스트를 얻는 방법을 설명하겠다. 나머지 부분도 공부할 만한 내용이 많지만, 지면 관계상 소스를 직접 참고하기를 바란다. 주석을 어느 정도 달아놓았기 때문에, 텔파이의 도움말의 도움을 받아가면서 부족하면 쉽게 이해할 수 있을 것으로 믿는다.

폼의 OnCreate, OnDestroy 이벤트 핸들러의 내용을 살펴 보면 다음과 같다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Integer(Drives) := GetLogicalDrives; //PC 가 가지고 있는 드라이브를 지정한다.
```

```

CreatelImages:                                //리스트뷰의 이미지 리스트를 설정한다.
Path := ExtractFilePath(Application.ExeName);
SetFiles:                                    //리스트뷰를 업데이트 한다.
SetColumnForFiles:                            //컬럼 헤더를 설정한다.
end;

```

GetLogicalDrives 함수는 PC 가 가지고 있는 논리적 드라이브의 세트를 반환하는 값으로, 이 함수를 이용하여 Drives 변수에 값을 저장한다. CreateImages 프로시저는 시스템 이미지 리스트를 리스트뷰의 이미지 리스트로 설정하는 역할을 하는데, 여기에 대해서는 다시 설명할 것이다. 일단 패스는 현재 실행 파일이 있는 패스로 설정하고, SetFiles 와 SetColumnForFiles 프로시저를 호출한다.

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    Smalls.Free;
    Larges.Free;
end;

```

시스템 이미지 리스트를 저장했던 전역 변수의 내용을 해제한다. 이런 부분을 소홀히 하면 리소스가 새게 되므로 주의하기 바란다.

그러면, CreateImages 프로시저는 어떻게 구현했는지 알아보자.

```

procedure CreatelImages:                    //시스템 이미지 리스트를 설정한다.
var
    SysIL: DWORD;
    SFI: TSHFileInfo;
begin
    Larges := TImageList.Create(Form1);
    Smalls := TImageList.Create(Form1);

```

SysIL 변수는 SHGetFileInfo 함수의 결과값을 저장되며, SFI 변수에 실질적인 내용이 저장된다. 그리고, TImageList 형의 전역 변수 Larges, Smalls 를 생성한다.

```

SysIL := SHGetFileInfo("", 0, SFI, SizeOf(SFI), SHGFI_SYSICONINDEX or
    SHGFI_LARGEICON);
if SysIL <> 0 then

```

```

begin
    Larges.Handle := SysIL;
    Larges.ShareImages := TRUE;
end;
SysIL := SHGetFileInfo('', 0, SFI, SizeOf(SFI), SHGFI_SYSICONINDEX or
    SHGFI_SMALLICON);
if SysIL <> 0 then
begin
    Smalls.Handle := SysIL;
    Smalls.ShareImages := TRUE;
end;
Form1.Files.SmallImages := Smalls;
Form1.Files.LargeImages := Larges;
end;

```

구현 방법은 단순하다. SHGetFileInfo 함수를 호출한 후, 이 함수의 결과를 이미지 리스트의 핸들로 사용하는 것이다. 이때 SHGetFileInfo 함수의 파라미터로 5 번째 파라미터의 내용에 SHGFI\_LARGEICON, SHGFI\_SMALLICON 이 설정되는 것에 따라 큰 아이콘과 작은 아이콘의 이미지 리스트를 가져오게 된다.

이번에는 패스를 변경하는 방법을 구현하도록 한다. 이 예제에서 패스의 변경이 요구되는 경우는 콤보 박스를 클릭하여 변경하는 것과 ‘위로’ 스피드 버튼을 클릭한 경우, 그리고 폴더나 드라이브를 더블 클릭한 경우이다. 이 중에서 콤보 박스의 내용을 변경한 경우의 구현 방법은 소스 코드를 참고하기 바라며, 나머지 2 가지 경우에 대해서 살펴 보도록 하자. 폴더나 드라이브를 더블 클릭한 경우에는 리스트 뷰의 OnDoubleClick 이벤트 핸들러를 호출하게 된다.

```

procedure TForm1.FilesDbClick(Sender: TObject);
    //리스트 아이템을 더블 클릭하면 드라이브, 폴더를 연다.
begin
    if Files.Selected = nil then Exit;    //선택된 아이템이 없다.
    if Files.Selected.SubItems[3] = '디렉토리' then
    begin
        Path := Files.Selected.SubItems[4];
        SetFiles;
    end
end

```



먼저 알아야 하는 것은, 아이템의 SubItems[4]의 내용에는 현재 파일이나 디렉토리의 완전한 경로가 들어 있다는 것이다. 디렉토리를 더블 클릭한 경우, 선택된 아이템의 SubItems[3]의 내용은 ‘디렉토리’이다. 이런 경우에는 SubItems[4]의 경로를 Path 전역 변수로 지정한 후 SetFiles 프로시저를 호출하면 간단하게, 그 디렉토리로 들어가는 것을 구현할 수 있다.

```
else
if Files.Selected.SubItems[3] = '드라이브' then
begin
    if not DirectoryExists(Files.Selected.SubItems[4]) then
        ShowMessage(Files.Selected.Caption + '에 접근할 수 없습니다.')
    else
        begin
            Path := Files.Selected.SubItems[4];
            SetColumnForFiles;
            SetFiles;
        end;
    end;
end
```

선택된 아이템이 드라이브이면 (SubItems[3]의 값이 ‘드라이브’), 컬럼 헤더의 내용을 파일에 맞도록 다시 설정하고 SetFiles 프로시저를 호출한다. 이때 존재하지 않는 디렉토리를 더블 클릭한 경우에는 접근할 수 없는 메시지를 보여주도록 한다.

```
else
    ShowProperties(Files.Selected.SubItems[4], Files.Selected.SubItems[3]);
end;
```

파일이 선택된 경우의 처리 방법으로, ShowProperties 프로시저를 호출하여 파일의 정보(경로와 크기)를 보여주도록 처리한다. 이 프로시저는 팝업 메뉴에서 파일 정보를 선택한 경우에도 호출된다. 자세한 것은 소스 코드를 참고하기 바란다.

여기에 비해, 상위 레벨로 옮겨가는 것은 다소 구현이 복잡하다. SpeedButton1 (‘위로’ 버튼)의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
```

```

if Path[Length(Path) - 1] = ':' then      //루트 디렉토리에 있다면 ...
begin
    Path := 'Drives';
    SetFiles;
    Files.SetFocus;
    if Files.Items.Count > 0 then Files.Selected := Files.Items[0];
end

```

먼저, 패스가 루트 디렉토리에 있는지를 먼저 알아야 한다. 루트 디렉토리에 있다면 Path 변수의 마지막에서 두번째 문자가 ':'일 것이다 ('c:w'를 생각해보라!). 이 때에는 Path 변수값을 'Drives'로 설정하고, SetFiles 를 호출한다. 물론 SetFiles 내부에서는 SetDrives 프로시저를 호출할 것이다. 그리고, 첫번째 아이템(주로 A 드라이브)을 선택한다.

```

else
begin
    Path := Copy(Path, 1, Length(Path) - 1);      //일단 마지막 'W'를 삭제한다.
    while Path[Length(Path)] <> 'W' do            //그리고, 제일 뒤의 디렉토리 이름을 삭제한다.
        Path := Copy(Path, 1, Length(Path)-1);
    SetFiles;
    Files.SetFocus;
    if Files.Items.Count > 0 then Files.Selected := Files.Items[0];
end;
end;

```

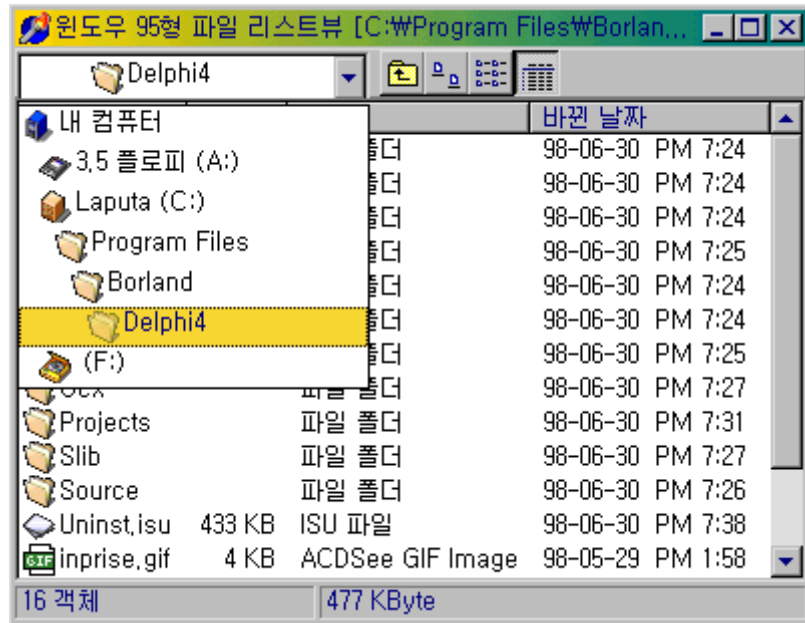
이번에는 상위 디렉토리로 올라가도록 해야 한다. 일단 마지막의 'W' 문자를 삭제한 뒤에 Copy 함수를 이용해서 'W' 문자가 다시 나타날 때까지 한 글자씩 뒤에서부터 삭제해 나간다. 이렇게 하면 제일 뒤의 디렉토리 문자열이 삭제될 것이므로, 이 문자열을 Path 변수를 하여 SetFiles 프로시저를 호출한다.

## ● 그 밖에

윈도우 95 스타일의 파일 리스트뷰를 구현하는 방법에 대해서 알아 보았다. 그 밖에도 이 예제에는 콤보 박스의 내용을 직접 그리는 방법과 탭을 이용하여 디렉토리의 깊이를 표현하는 방법, 크기와 시간, 날짜 등을 처리하는 여러가지 유틸리티 함수 등이 구현되어 있다. 이들은 소스 코드를 참고하여 익혀두기 바란다.

이제 이 예제를 한번 실행해보자. 스피드 버튼을 눌러 보이는 형태를 바꿔도 보고, 컬럼

헤더를 클릭하여 정렬도 한번 해도록 하라. 실행 화면은 다음과 같다.



## 툴바와 쿨바의 디자인

툴바는 보통 폼의 제일 윗부분에 위치하여, 버튼을 비롯한 여러가지 컨트롤을 담게 되는 일종의 패널이다. 쿨바 역시 일종의 투바이지만, 밴드를 이용하여 크기를 변경하고, 이동할 수 있다.

버튼을 추가하는 것이 가장 흔한 투바의 사용 방법이지만, 컬러 그리드나 스크롤 바, 라벨 등의 컨트롤도 추가할 수 있다. 보통 투바를 구현하기 위해서는 패널을 폼에 추가하고, 스피드 버튼 등을 추가해 사용하거나, 투바 컴포넌트를 이용하게 된다.

툴바 컴포넌트(TToolBar)를 사용하면, 버튼이나 다른 컨트롤을 자동으로 크기와 위치에 맞추어 정렬할 수 있으며, TToolButton 컨트롤을 버튼으로 사용할 경우에는 버튼들을 그룹으로 만들어 그 기능 별로 관리할 수 있으며, 그룹 별로 디스플레이 옵션을 다르게 줄 수도 있다. 쿨바 컴포넌트(TCoolBar)는 투바 컴포넌트의 편리함에 밴드를 이용해 위치와 크기를 가변할 수 있는 성능을 제공한다.

필자의 개인적인 생각으로는 패널을 이용한 방법 보다는, 투바나 쿨바 컴포넌트를 이용하려고 권하고 싶다. 이들은 윈도우의 컨트롤을 그대로 사용하는 것이기 때문에, 운영체제가 이들 컨트롤의 성능을 향상시키면, 어플리케이션의 성능도 같이 향상된다.

툴바의 사용 예제는 8 장에서 소개한 바 있으므로, 여기서는 투바 컴포넌트의 기능에 대해서 간략하게 정리하고 넘어가도록 하겠다.

- 투바 컴포넌트를 이용하여 투바 생성하기

툴바 컴포넌트는 Win32 페이지에 위치하는데, 폼에 추가하면 자동적으로 폼의 가장 위쪽에 정렬된다. 사용을 위해서는 툴 버튼이나 다른 버튼을 바에 추가하여 사용한다.

툴 버튼은 툴바 컴포넌트와 작업하기 위해 디자인 된 버튼으로 스피드 버튼과 마찬가지로 푸시 버튼으로 사용할 수도 있으며, 토글과 라디오 버튼과 같은 용도로 사용하게 할 수도 있다.

### 1. 툴 버튼의 추가

툴 버튼을 툴바에 추가하려면, 툴바에서 오른쪽 버튼을 누르면 나오는 팝업 메뉴에서 New Button 메뉴를 선택하면 된다. 툴바에 있는 모든 툴 버튼 들은 같은 높이와 폭을 가지게 된다. 다른 컨트롤 들을 툴바에 추가하면, 이들 역시 동일한 높이를 가지게 된다.

### 2. 버튼에 이미지 대입하기

각 툴 버튼은 런타임에서 보여주게될 이미지를 결정하기 위해 ImageIndex 프로퍼티를 가지고 있다. 이 때 버튼에 추가할 이미지는 Images 프로퍼티에 대입된 ImageList 컴포넌트에 의해 결정되며, 하나의 이미지만 제공된 경우 비활성화된 형태를 보여주기 위해 이미지를 가공하게 된다.

또한, 툴 버튼에는 마우스 포인터가 위를 지나가거나 버튼이 비활성화된 이미지를 따로 지정하게 할 수 있는데, 이들을 각각 DisabledImages, HotImages 프로퍼티를 이용해서 설정한다.

### 3. 툴 버튼을 라디오 그룹처럼 사용하기

툴 버튼을 그룹지어 라디오 버튼처럼 사용하려면, 서로 연관시킬 버튼 들을 선택하고 이들의 Style 프로퍼티를 tbsCheck 로 설정한다. 그리고, Grouped 프로퍼티를 True 로 지정하면 된다. 이렇게 하면, 라디오 버튼을 사용하듯이 이들 중 하나만 선택하도록 하는 것이 가능하다. 서로 인접한 버튼 들의 Style 프로퍼티와 Grouped 프로퍼티가 각각 tbsCheck, True 이면 하나의 그룹으로 취급된다. 그룹을 해제하려면, 다음의 방법들 중에 하나를 이용하면 된다.

- Grouped 프로퍼티를 False 로 설정한다.
- Style 프로퍼티를 변경한다. 툴바에 공간을 확보하거나, 툴바의 그룹을 나누기 위해서 Style 프로퍼티의 값을 tbsSeparator 나 tbsDivider 로 설정한다.
- 툴 버튼 옆에 다른 컨트롤을 위치시킨다.

#### 4. 토글되는 툴 버튼

버튼을 토글하여 사용하려면, Style 프로퍼티를 `tbsCheck` 로 선택하면 된다. 그룹을 짓고, 이들이 여러 개 눌러 있거나, 올라와 있도록 할 수 있게 하려면 `AllowAllUp` 프로퍼티를 이용하면 된다. 이 프로퍼티가 `True` 이면 여러 버튼 들이 한 번 클릭하면 누른 채로 있고, 다시 누르면 눌린 채로 있게 된다.

#### 5. 툴 버튼에 메뉴 대입

툴 버튼을 선택하고 오브젝트 인스펙터에 보면 `DropDownMenu` 프로퍼티가 있다. 이 프로퍼티에 `TPopupMenu` 컴포넌트의 이름을 대입하고, `AutoPopup` 프로퍼티를 `True` 로 설정하면 버튼이 눌릴 때마다 팝업 메뉴가 뜨게 된다.

#### 6. 툴바 숨기기와 보여주기

어플리케이션에 여러 개의 툴바를 사용할 경우에 이들 중 사용자가 보고자 하는 것만 보도록 할 수 있다. 이럴 때에는 툴바를 적당하게 보이고, 숨기기만 하면 된다. 이를 위해서는 `Visible` 프로퍼티를 이용한다.

#### ● 툴바 컴포넌트의 활용

툴바 컴포넌트는 독립적으로 이동과 크기 변화가 가능한 밴드를 가지고 있는 컴포넌트이다. 사용자는 밴드를 위치시키기 위해서는 단순히 각 밴드의 좌측에 있는 그림을 클릭하여 드래그하면 된다.

툴바 컴포넌트 역시 Win32 페이지에 있으며, 이를 폼에 추가하면 폼의 가장 위에 위치한다. 참고로, `TWindow` 컨트롤에서 상속받은 윈도우 컨트롤 들만 분리된 밴드에 위치시킬 수 있다. 그러므로, 스피드 버튼이나 라벨과 같이 그래픽 컨트롤을 상속한 경우에는 분리된 밴드에 보이게 할 수 없다.

툴바 컴포넌트에서 사용하는 중요한 프로퍼티를 소개하면 다음과 같은 것들이 있다.

프로퍼티	설 명
<code>AutoSize</code>	이 값을 <code>True</code> 로 설정하면 밴드 들에 맞게 크기를 자동으로 조절한다.
<code>FixedSize</code>	이 값을 <code>True</code> 로 설정하면 밴드의 높이를 일정하게 유지한다.
<code>Vertical</code>	이 값이 <code>True</code> 이면 수평보다 수직에 맞추어 밴드를 설정한다.
<code>ShowText</code>	이 값을 <code>False</code> 로 설정하면 각 밴드의 텍스트 프로퍼티를 런타임에 보여주지 않

	도록 한다.
BandBorderStyle	bsNone 을 선택하면 바 주위의 보더를 제거할 수 있다.
FixedOrder	이 값을 True 로 설정하면, 사용자가 밴드의 순서를 바꾸지 못하도록 한다.
Bitmap	배경 비트맵을 설정한다.
Images	각 밴드의 프로퍼티로 이미지를 저장할 TImageList 객체를 지정한다.
ImageIndex	각 밴드에 대입할 이미지의 이미지의 인덱스이다.

## 정 리 (Summary)

이번 장에서는 Win32 공통 컨트롤의 사용 방법에 대해서 알아 보았다. 물론 다루지 못한 많은 수의 컴포넌트가 있지만, 이들도 사용자 인터페이스를 향상시키는 데에 유용하므로 델파이의 도움말과 데모 프로그램 등을 통해 사용방법을 익혀둘 것을 권하고 싶다.

특히, 이번 장에서 다룬 트리뷰와 리스트뷰 컨트롤은 사용 하기에 따라서는 고급스러운 어플리케이션을 만들 수 있을 것이다.

이것으로 제 2 부의 내용을 모두 마친다. 비록 다루지 못한 컴포넌트 들이 많지만, 사용하는 방법은 대개 비슷하므로 여러 차례 연습해보면 사용하는 법을 쉽게 익힐 수 있을 것이다. 3 부에서는 데이터베이스 어플리케이션을 제작하는 기초적인 방법에서부터, 고급스러운 테크닉에 이르기까지 자세하게 알아보도록 할 것이다.

## 데이터베이스 어플리케이션 제작의 기초

### (Basics of Database Application Development)

이번 장부터 시작되는 3 부에서는 델파이로 작성하는 어플리케이션으로, 가장 흔하면서도 중요하게 사용되고 있는 데이터베이스 어플리케이션을 제작하는 방법에 대해서 다루게 된다. 델파이가 첫 버전으로 세상에 선보일 당시에는 윈도우용 데이터베이스 어플리케이션을 제작할 수 있는 몇가지 툴이 있었다. 당시에 데이터베이스 어플리케이션을 작성하는 SQL Windows 와 델파이를 많은 곳에서 비교하였다. 필자도 초기버전과 델파이 1.0 을 비교하면서 주저없이 델파이를 선택하였다. 그 이유는 SQL Windows 는 Case 도구였고, 델파이는 볼랜드 파스칼 위에 Case 도구 엔진을 올려놓은 라이브러리의 확장판과 같았기 때문이었다. 델파이는 매우 강력한 데이터베이스 어플리케이션을 제작할 수 있는 언어이자 도구이다.

데이터베이스는 데이터를 저장하기 위한 일반적인 수단을 제공하는 것이고, 데이터베이스 엔진은 이러한 데이터베이스에 데이터를 저장하고, 읽어오고, 여러가지로 처리하는 방법을 제공한다. 델파이에서는 기존에 존재하는 데이터베이스 엔진과의 접속은 물론, 자체적으로 꽤 높은 성능의 데이터베이스 엔진을 제공하고 있다.

이번 장에서는 그 첫번째로 데이터베이스에 대한 전체적인 개념과 데이터베이스 어플리케이션의 구조 등에 대해서 알아보고, 델파이를 이용하여 데이터베이스 어플리케이션을 작성하는 간단한 방법을 익히도록 한다.

델파이의 컴포넌트 중에서 데이터베이스와 관련되어 있는 컴포넌트 팔레트의 페이지는 3 가지라고 보면 된다. 주된 것은 Data Access 페이지와 Data Controls 페이지이지만, 최근의 멀티-tier 데이터베이스 어플리케이션을 작성할 때 사용하는 MIDAS 페이지의 컴포넌트들도 중요하다.

이 중에서 Data Access 페이지의 컴포넌트 들은 어플리케이션이 데이터베이스에 접근하여 데이터를 읽고, 쓰게 해주는 역할을 한다. 그리고, Data Controls 페이지에 있는 컨트롤 들에게 데이터를 전달하여, 여기에 데이터를 표시할 수 있도록 하는 것이다.

### 데이터베이스 어플리케이션 아키텍처 (Database Application architecture)

델파이의 데이터베이스 어플리케이션은 사용자 인터페이스 요소 들과 데이터베이스를 관리하는 컴포넌트, 그리고 데이터 세트라고 불리는 데이터베이스에 저장된 테이블 등의 데이터를 나타내는 컴포넌트 들로 구성된다.

이때 데이터 모듈(data module)에 데이터베이스 접근 컴포넌트 들을 분리하면 사용자 인터페이스 부분과 데이터 접근 부분을 보다 효율적으로 관리할 수 있다. 특히 잘 디자인된 폼이나 데이터 모듈이 있으면, 이를 객체 저장소(Object repository)에 저장해 두었다가 사용

한다면 제작하는 어플리케이션의 전체적인 인터페이스를 일관되게 유지할 수 있다.

데이터베이스 어플리케이션의 아키텍처는 실제로 사용하게 되는 데이터베이스의 종류가 무엇이며, 데이터베이스 정보를 공유하게 될 사용자가 몇 명인지, 그리고 사용할 정보의 종류에 따라서 몇 가지 카테고리로 나누어 생각해볼 수 있다.

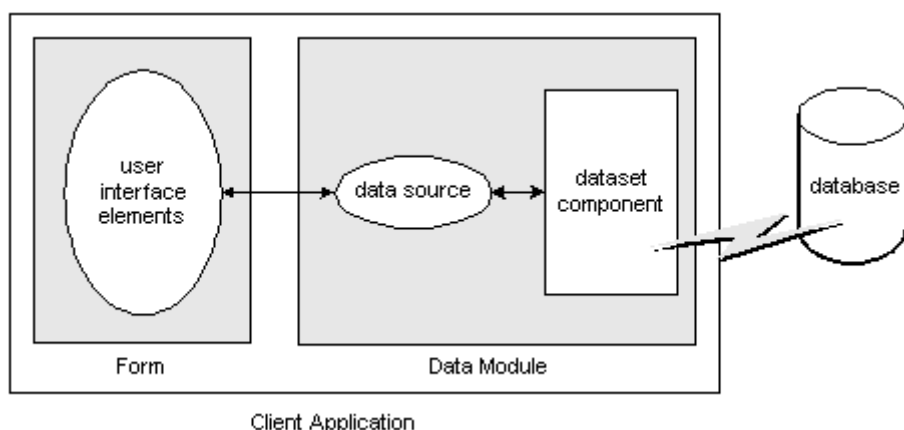
만약 수 명의 사용자가, 서로 공유하지 않는 정보를 사용하는 데이터베이스 어플리케이션을 작성한다면 1-tiered 기반의 로컬 데이터베이스를 사용하면 될 것이다. 이 방법은 데이터가 로컬에 저장되므로 속도라는 측면과 데이터베이스 서버를 따로 사지 않아도 되므로 경제성이 좋다. 그렇지만, 이 경우에도 얼마나 많은 양의 데이터와 테이블을 사용해야 되는지에 따라서 결정이 달라질 수 있다.

2-tiered 어플리케이션은 1-tiered 기반의 어플리케이션에 비해 보다 많은 사용자를 지원하며, 커다란 원격 데이터베이스를 사용할 수 있기 때문에 로컬 데이터베이스의 한계를 극복할 수 있다.

데이터베이스 정보에 몇 개의 테이블의 관계가 복잡하게 얽혀있고, 계속적으로 클라이언트 수가 증가하는 추세라면 이럴 때에는 멀티-tiered 데이터베이스 어플리케이션을 선택해야 한다. 멀티-tiered 어플리케이션에는 데이터베이스의 상호작용과 데이터 간의 관계에 대한 부분을 중앙집중적으로 관리하는 로직을 미들웨어라는 형식으로 제공하게 된다. 이렇게 함으로써 서로 다른 클라이언트 어플리케이션 들이 데이터 로직이 호환된다면 같은 데이터를 사용하도록 할 수도 있고, 클라이언트 어플리케이션을 작고 가볍게(thin) 만들어 줄 수도 있다. 가벼운 클라이언트 어플리케이션의 장점은 설치도 쉽고, 데이터베이스 접속 소프트웨어를 포함하지 않으므로 환경 설정이나 관리도 대단히 쉬운 잇점이 있다. 또한, 멀티-tiered 어플리케이션은 데이터 처리 작업을 여러 시스템에 분산시켜 처리할 수 있기 때문에 수행 성능도 높일 수 있다.

- 확장성에 대한 계획 수립 (Planning for scalability)

VCL 의 데이터인식 컨트롤(data-aware controls)을 이용하면, 데이터베이스와 데이터베이스에 저장된 데이터의 행동을 추상화할 수 있기 때문에, 쉽게 확장이 가능한 어플리케이션으로 개발할 수 있다. 개발하는 어플리케이션의 모델이 1-tiered, 2-tiered, 멀티-tiered 중 어느 것이라도, 사용자 인터페이스 부분을 다음 그림과 같이 데이터 접근층(data access layer)과 분리할 수 있다.





이 그림에서 폼은 사용자 인터페이스를 가리킨다. 폼에는 데이터 컨트롤과 다른 사용자 인터페이스 요소가 포함되는데, 데이터 컨트롤은 사용자 인터페이스를 데이터베이스의 데이터베이스의 정보에 접속하도록 해 준다. 이때 데이터 컨트롤과 연결되는 데이터 소스를 데이터 세트라고 한다. 데이터 소스와 데이터 세트를 데이터 모듈에 분리하면, 폼에는 어떤 방법으로 어플리케이션을 작성하더라도 변경될 부분이 없다. 다만 데이터 세트만 변경해 주면 되는 것이다.

데이터 모듈이 처음 소개 된 것이 델파이 2.0 버전 부터 이므로, 꽤 오랜 시간동안 사용되어온 개념이지만 아직도 생각보다 많이 사용되지 않고 있는 듯하다. 그렇지만, 지금까지 설명한 중요성을 염두에 두고 가능한 반드시 데이터 모듈을 사용하는 것을 습관처럼 하도록 권하고 싶다.

#### 참고:

일부 사용자 인터페이스 요소 들은 확장하고자 할 때, 고려해 주어야 하는 경우가 있다. 예를 들어, 데이터베이스의 보안을 확보하기 위해, 데이터베이스에 접근할 때 로그인 할 수 있도록 하는 방법 등을 들 수가 있을 것이다.

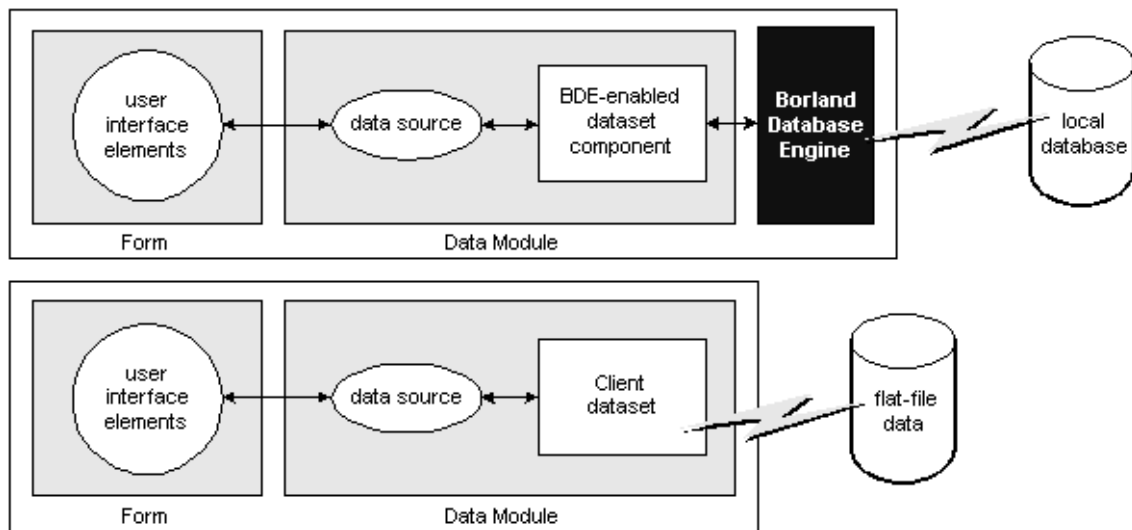
BDE 에 기반을 둔 어플리케이션을 개발한다면, 1-tiered 에서 2-tiered 모델로 확장하는 것이 매우 쉽다. 단지 데이터 세트에 있는 몇 가지 프로퍼티를 로컬 데이터베이스에서 SQL 서버에 접속하도록 바꾸기만 하면 그걸로 끝이다.

또한, 플랫폼 파일에 기반을 둔 데이터베이스 어플리케이션을 개발하면, 이 모델은 쉽게 멀티-tiered 모델로 확장할 수 있다. 이것은 이들의 구조가 동일한 클라이언트 데이터 세트 컴포넌트를 사용하기 때문이다. 또한, 플랫폼 파일 어플리케이션과 멀티-tiered 클라이언트 어플리케이션을 동시에 지원하도록 어플리케이션을 제작할 수도 있는데, 이러한 모델을 서류가방(briefcase) 모델이라고 한다.

만약, 결국에는 멀티-tiered 어플리케이션으로의 확장을 염두에 두고 있다면, 처음에 데이터베이스 어플리케이션을 제작할 때부터 이를 고려하는 것이 좋다. 사용자 인터페이스는 반드시 데이터 모듈과 분리하여 제작하고, 동시에 미들 tier 에 위치할 모든 로직 들은 따로 분리하여 데이터 모듈로 관리하도록 한다. 또한, 사용자 인터페이스 요소를 데이터 세트에 접속하도록 할 때에도 처음부터 클라이언트 데이터 세트를 경유하여 BDE 에 접속하도록 분리된 데이터 모듈에 작성하면, 단순히 접속할 대상을 수정해 주는 것으로 확장이 가능하게 할 수 있다.

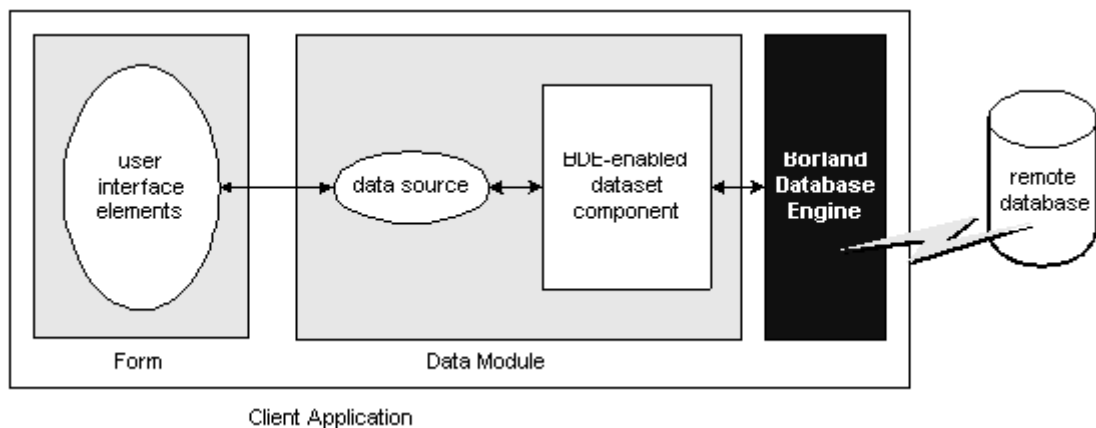
#### ● 1-tiered 데이터베이스 어플리케이션

1-tiered 데이터베이스 어플리케이션은 어플리케이션과 데이터베이스가 같은 파일 시스템을 사용한다. 로컬 데이터베이스나 파일을 이용하여 데이터베이스 정보를 저장하게 된다. 이때 어플리케이션은 사용자 인터페이스 부분과 데이터 접근 기전(BDE 를 사용하거나, 플랫폼 파일을 사용)으로 구성되며, 데이터 세트 컴포넌트의 종류가 무엇이나에 따라서 로컬 데이터베이스와 플랫폼 파일 데이터베이스로 구분할 수 있다. 로컬 데이터베이스는 BDE 를 사용하여 파라독스, 디베이스, 액세스, 폭스 프로와 같은 상용 로컬 데이터베이스에 접속하게 되며, 플랫폼 파일은 자체적인 파일을 사용한다.



- 2-tiered 데이터베이스 어플리케이션

2-tiered 데이터베이스 어플리케이션에서는 클라이언트 어플리케이션이 데이터와 원격 데이터베이스 서버에 직접 상호작용할 수 있는 인터페이스를 제공해야 한다.



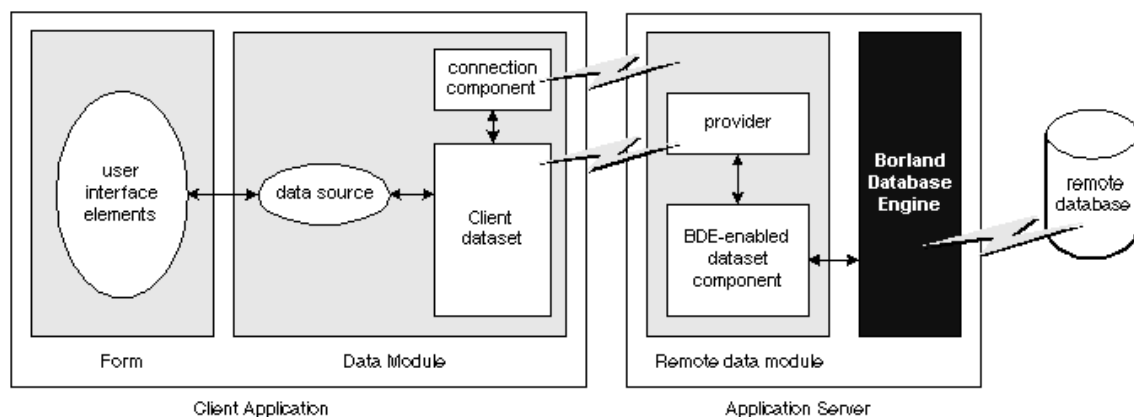
이 모델에서는 모든 어플리케이션이 데이터베이스 클라이언트가 된다. 클라이언트는 서버

에게 정보를 요구하거나, 업데이트된 정보를 데이터베이스 서버로 전송한다. 서버는 여러 클라이언트의 요구를 동시에 처리하고, 이들이 데이터에 접근하고 업데이트할 수 있도록 관리한다.

- 멀티-tiered 데이터베이스 어플리케이션

멀티-tiered 데이터베이스 어플리케이션은 어플리케이션을 서로 다른 기계에 존재할 수 있도록 쪼개어 관리하게 된다. 클라이언트 어플리케이션은 데이터에 대한 사용자 인터페이스를 제공한다.

클라이언트는 모든 데이터에 대한 요구와 업데이트된 정보를 어플리케이션 서버(리모드 데이터 브로커라고도 한다)에 전송하게 된다. 어플리케이션 서버는 원격 데이터베이스 서버와 직접 통신을 하거나, 다른 커스텀 데이터 세트를 통해 데이터베이스 서버에 접속하게 된다. 보통 이 모델에서의 클라이언트와 어플리케이션 서버, 원격 데이터베이스 서버는 다른 기계에 위치시키는 것이 좋다. 다음 그림은 멀티-tiered 어플리케이션과 BDE의 관계를 나타낸 것이다.



어플리케이션 서버와 클라이언트 어플리케이션을 모두 델파이를 이용해서 작성할 수 있다. 클라이언트 어플리케이션은 표준 데이터인식 컨트롤을 이용하여 하나 이상의 클라이언트 데이터 세트 컴포넌트에 데이터 소스를 통해 접속하게 되고, 여기서 데이터를 얻어서 보여주고, 편집할 수 있게 된다. 각각의 클라이언트 데이터 세트는 IProvider 인터페이스를 통해 어플리케이션 서버와 통신을 하게 되는데, IProvider 인터페이스는 원격 데이터 모듈(remote data module)의 한 부분으로 TCP/IP, DCOM, MTS, CORBA 등의 다양한 프로토콜을 이용하여 통신을 할 수 있다. 이러한 프로토콜은 클라이언트 어플리케이션에서 사용하는 접속 컴포넌트와 서버 어플리케이션의 원격 데이터 모듈의 종류에 따라 결정된다.

어플리케이션 서버는 2 가지 방법으로 IProvider 인터페이스를 사용할 수 있다. 어플리케이션 서버에 Provider 객체가 있으면, 이 객체를 이용하여 IProvider 인터페이스를 생성하

게 된다. 이 방법이 앞의 그림에서 표현한 모델이다. 어플리케이션 서버가 이런 컴포넌트를 포함하고 있지 않은 경우에는 BDE 가 가능한 데이터 세트에서 IProvider 인터페이스를 생성하게 된다. 그렇지만, provider 컴포넌트를 사용하는 것이 보다 확실하게 인터페이스를 사용할 수 있는 방법이다. 두가지 방법 모두, 모든 데이터는 클라이언트 어플리케이션과 어플리케이션 서버 사이에 인터페이스를 통해서 통신하게 된다.

## 데이터베이스의 특징 (Database Features)

일반적으로 데이터베이스를 관리하는 DBMS 에는 여러 명의 사용자가 동시에 접근하여, 대용량의 데이터를 다루게 되므로 이를 보다 효과적이고, 견고하게 관리하기 위해 공통적인 특징을 가지게 된다.

흔히, 이런 공통적인 특징을 어떤 DBMS 가 더 잘 지원하는지 여부를 가지고 벤치마크 테스트를 하기도 하며, 개발자에게 편리하면서도 강력한 기능을 지원하는 DBMS 가 좋은 DBMS 라고 말할 수 있다.

그렇다면, DBMS 가 갖추고 있어야 할 데이터베이스의 특징과 데이터베이스 어플리케이션을 제작할 때에 여기에 대해서 어떤 것을 고려해야 하는지 알아보도록 하자. 이들에 대한 자세한 접근은 다음 장에서 다루게 되겠지만, 기초적인 부분을 미리 소개한다. 참고로 DBMS 에는 액세스나 파라독스와 같이 로컬 환경에 기반을 둔 로컬 DBMS 와 오라클이나 인터베이스와 같이 네트워크 환경에 기반을 둔 원격 SQL 서버 DBMS 가 있다.

### ● 데이터베이스 보안 (Database security)

데이터베이스에는 정보가 저장되는데, 이 정보가 기업 기밀 등과 같이 중요한 것인 경우 이를 보호할 수 있는 보안 스키마가 제공되어야 한다. 파라독스나 디베이스와 같은 로컬 DBMS 는 보안을 테이블 또는 필드 레벨에서만 제공할 수 있다. 즉, 사용자가 보호된 테이블에 접근할 경우에 패스워드를 묻도록 할 수 있다. 일단 사용자가 인증(authentication)이 되면 이들은 허용된 필드에만 접근할 수 있다. 이에 비해 대부분의 SQL 서버는 처음에 데이터베이스 서버에 접근할 때 사용자 이름과 패스워드를 요구한다.

데이터베이스 어플리케이션을 디자인 할 때에는 데이터베이스 서버에 의해 요구되는 인증의 종류에 대해 고려해야 한다. 만약 사용자가 패스워드를 사용하지 않도록 해야 한다면, 패스워드를 요구하지 않는 데이터베이스 서버를 사용하거나, 패스워드를 프로그램에서 제공하도록 만들어야 한다. 그렇지만, 프로그램을 통해서 패스워드를 제공하도록 하면, 패스워드가 어플리케이션에 의해 읽혀지고 노출되지 않도록 주의해야 할 것이다.

그리고, 사용자가 패스워드를 입력하도록 할 경우에는, 언제 패스워드를 요구할 것인지를 생각해야 한다. 만약, 현재는 로컬 데이터베이스를 사용하고 있지만 이후 SQL 서버로 확장해야 한다면, 테이블에 접근하기 전에 패스워드를 요구하도록 어플리케이션을 제작할 수

도 있겠다.

만약, 어플리케이션이 여러 개의 보호된 시스템이나 데이터베이스에 접근해야 되기 때문에, 여러 차례 패스워드를 입력해야 한다면, 사용자에게 하나의 마스터 패스워드를 제공하여 필요한 시스템과 데이터베이스에 접근할 수 있도록 하는 것이 현명하다. 즉, 마스터 패스워드를 입력하면 어플리케이션이 패스워드를 프로그램적으로 입력하도록 하는 것이다.

멀티-tiered 어플리케이션에서는 서로 다른 보안 모델을 동시에 사용하기를 원할 수도 있다. 예를 들어, 미들-tier 를 조절하기 위해 CORBA 나 MTS 를 사용할 수도 있고, 미들-tier 가 데이터베이스 서버에 로그인 하는 작업을 모두 처리하도록 할 수도 있다.

#### ● 트랜잭션 처리 (Transactions)

트랜잭션은 한 번에 처리되어야 할 명령의 그룹이다. 이들이 한 번에 처리되어 완전히 데이터를 업데이트 된 경우 이를 ‘committed’ 라고 하며, 처리에 실패해서 전체 명령이 하나도 실행되지 않은 상태로 돌아가는 것을 ‘rolled back’이라고 한다.

트랜잭션은 데이터베이스 명령이 실행 중에 일어날 수 있는 여러 가지 하드웨어 실패 등에 대한 고려를 하기 위해 디자인 된 것이다. 가장 흔히 예를 드는 것으로는 은행의 온라인 업무를 들 수 있다. 또한, 트랜잭션은 SQL 서버의 일치성(concurrency)을 보장하는 기초가 된다. SQL 서버는 내부적으로 트랜잭션을 스케줄하여 여러 명의 사용자가 데이터를 처리할 때 한 사람이 데이터를 변경할 때 다른 사람이 동시에 접근하여 데이터가 깨지지 않도록 관리한다.

로컬 데이터베이스의 경우 트랜잭션을 지원하지 않는 것이 많지만, BDE 드라이버를 이용하면 제한적이지만 일부에서 트랜잭션을 지원하게 할 수 있다. 또한, SQL 서버와 ODBC 와 호환되는 데이터베이스는 대개 데이터베이스 서버가 트랜잭션을 직접 지원한다.

멀티-tiered 어플리케이션의 경우에는 트랜잭션을 데이터베이스 차원에서 지원하지 않고 어플리케이션 서버에서 지원하도록 하면 여러 개의 데이터베이스에 대한 접근을 포함한 트랜잭션을 처리할 수 있다. 예를 들어, 인적 사항은 오라클이 관리하게 하고, 물건 들에 대한 재고 정보를 인터베이스가 관리하게 하는 것과 같은 분산 트랜잭션 처리가 가능하다.

#### ● 데이터 사전 (Data Dictionary)

어떤 데이터베이스를 사용하든, 어플리케이션은 데이터 사전에 접근할 수 있다. 데이터 사전은 어플리케이션과는 독립적으로 사용자가 조절할 수 있는 저장 영역으로 필드의 속성 세트를 확장하거나 데이터의 내용을 설명하는 등의 일을 할 수 있는 곳이다.

예를 들어, 회계 어플리케이션을 개발할 때 화폐 단위의 포맷을 여러 가지로 표현해야 한다면, 이를 위해 여러 가지 특수한 필드 속성을 생성할 수 있다.

데이터 세트를 디자인 시에 생성하는 경우에는 오브젝트 인스펙터를 이용해서 필드의 내용

을 일일이 설정하기 보다는 데이터 사전에서 확장된 필드 속성을 이용하도록 하는 것이 더욱 편리하고 효율적이다.

클라이언트/서버 환경에서는 데이터 사전의 위치를 추가적인 정보의 공유를 위해 원격 서버에 위치시킬 수 있다. 데이터 사전에 대한 자세한 내용은 다음 장의 SQL 탐색기에 대한 설명을 할 때 다루게 될 것이므로 이를 참고하기 바란다.

데이터 사전에 대한 프로그래밍 인터페이스는 `drintf.pas` 유닛에서 제공하는데, 여기에는 다음과 같은 메소드 들이 있다.

루틴	설명
DictionaryActive	데이터 사전이 활성화되어 있는지 나타낸다.
DictionaryDeactivate	데이터 사전을 비활성화 시킨다.
IsNullID	주어진 ID 가 Null ID 인지 나타낸다.
FindDatabaseID	주어진 앨리어스의 데이터베이스 ID 를 반환한다.
FindTableID	주어진 데이터베이스의 테이블 ID 를 반환한다.
FindFieldID	주어진 테이블의 필드 ID 를 반환한다.
FindAttrID	명명된 속성 세트에 대한 ID 를 반환한다.
GetAttrName	주어진 ID 에 해당되는 속성 세트의 이름을 반환한다.
GetAttrNames	사전에서의 각 속성 세트에 대한 콜백을 실행한다.
GetAttrID	지정된 필드에 대한 속성 세트 ID 를 반환한다.
NewAttr	필드 컴포넌트에서 새로운 속성 세트를 생성한다.
UpdateAttr	필드의 프로퍼티와 맞는 속성 세트를 업데이트한다.
CreateField	저장된 속성에 기초한 필드 컴포넌트를 생성한다.
UpdateField	지정된 속성 세트에 맞는 필드의 프로퍼티를 변경한다.
AssociateAttr	주어진 필드 ID 와 속성 세트를 연관시킨다.
UnassociateAttr	필드 ID 와 속성 세트의 연관을 제거한다.
GetControlClass	지정된 속성 ID 에 대한 컨트롤 클래스를 반환한다.
QualifyTableName	사용자 이름에 적당한 테이블 이름을 반환한다.
HasConstraints	사전에 있는 데이터 세트가 constraints 를 가지고 있는지 나타낸다.
UpdateConstraints	import 된 데이터 세트의 constraints 를 업데이트 한다.
UpdateDataset	데이터 세트를 사전의 constraints 와 현재 설정을 업데이트 한다.

- 참조 무결성, 저장 프로시저와 트리거 (Referential integrity, stored procedures, and triggers)

모든 관계형 데이터베이스는 앞에서 설명한 바와 같이 어플리케이션이 데이터를 저장하고,

처리하는 공통적인 형태를 가지고 있다. 이 밖에도 데이터베이스는 다음과 같은 특징을 가지고 있는데, 이들은 데이터베이스 별로 조금씩 차이가 날 수 있다.

### 1. 참조 무결성 (Referential integrity)

참조 무결성은 테이블 사이의 마스터/디테일 관계를 제거하는 방법을 제시한다. 사용자가 마스터 테이블에서 디테일 레코드가 있는 필드를 삭제하려고 할 때에는 참조 무결성 규칙에 따라서, 삭제를 못하게 하거나 자동으로 디테일 레코드를 삭제하도록 할 수 있다.

### 2. 저장 프로시저 (Stored procedures)

저장 프로시저는 명명되어 저장된 SQL 문장의 세트이다. 저장 프로시저는 보통 서버에서 흔히 행해지는 데이터베이스 작업을 저장했다가 이를 실행하여, 레코드 세트를 반환하게 된다.

### 3. 트리거 (Triggers)

트리거는 특정 명령에 의해 자동으로 실행되는 SQL 문장의 세트이다.

## 데이터베이스의 선택 (Select Database)

텔파이 데이터베이스 어플리케이션 개발자는 BDE 와 BDE 이외의 다른 데이터베이스 접근 방법들 중에서 선택의 기로에 놓이는 경우가 많다. 텔파이의 경우 기본적으로 제공되는 BDE 이외에도 아폴로, 타이탄과 같은 써드 파티에서 제공하는 여러 가지 데이터베이스 엔진들이 있기 때문에, 이를 선택하는 것이 중요한 고려 사항이 되기도 한다.

선택의 기준이 되는 핵심은 간단하다. 어플리케이션의 요구사항이 어느 정도 되며, 이를 해결하기 위한 솔루션이 어떤 것이 있는가 ?, 그리고, 특정 기술의 장점이 있는 경우의 솔루션은 어떤 것이 있는가 ? 하는 것이다.

이를 위해 여러 가지를 고려해보아야 하는데, 특정 데이터베이스 포맷을 지원해야 하는지 ?, 특정 DBMS 에 접근해야 하는지 ?, 또한 어플리케이션에 적합한 데이터베이스의 종류는 어떤 것인지 ? 따위의 것들이다.

이를 단계별로 고려해야 할 점들을 나열해 보면 다음과 같다.

- 1 단계: 사용할 데이터베이스의 종류를 선택한다.

첫 단계로는 어플리케이션에 필요한 데이터베이스의 종류가 어떤 것인지를 결정하는 것이다.

고려해야 할 것으로는 다음과 같은 것들이 있다.

#### 1. 로컬 데이터베이스 (Local Database)

데이터가 데이터를 사용하는 컴퓨터, 그리고 이를 사용하는 사용자와 같은 컴퓨터에만 위치할 경우, 동시에 여러 명의 사용자가 접근하지 않아도 될 때 선택한다.

#### 2. 다중 사용자 파일 서버 시스템 (Multi-user File Server)

파일 서버에 데이터가 위치하고, 모든 처리 과정은 클라이언트에서 일어나는 모델이다. 파일을 열고, 몇 바이트를 읽고 하는 명령과 같이 저수준 요구 사항이 사용자의 컴퓨터와 파일 서버에서 전송되게 된다. 파일 서버 데이터베이스는 데이터의 안정성이 낮다. 만약 사용자의 컴퓨터가 어떤 이유에서든 데이터베이스 업데이트를 완료하지 못하면, 데이터베이스는 불안정한 상황에 놓이게 되고, 이럴 경우 데이터베이스의 수리를 위해서는 모든 사용자가 데이터베이스 어플리케이션에서 빠져나가야 한다.

그러므로, 파일 서버 데이터베이스는 다음과 같이 제한된 경우에만 사용하는 것이 좋다.

- 1) 어플리케이션이 분조를 다루는 상황이 없어야 한다.
- 2) 보안이 그렇게 중요하지 않아야 한다.
- 3) 사용자 수가 15 명을 넘지 않는다.
- 4) 예산이 부족한 등의 클라이언트/서버 솔루션을 선택하지 못할 이유가 있을 경우

#### 3. 다중 사용자 클라이언트/서버 데이터베이스 시스템

클라이언트가 서버에 쿼리 문장과 같이 고수준의 요구를 할 수 있는 시스템이다. 그러므로, 많은 양의 데이터 처리가 서버에서 일어나게 되며, 처리된 데이터가 클라이언트에 전송되는 형태를 가지는 데이터베이스 시스템이다. 클라이언트는 데이터베이스 파일에 직접 접근할 수 없기 때문에, 데이터베이스 파일이 망가질 염려가 없으며, 사용자 수가 많아질수록 효율이 증가한다. 앞서서도 언급했지만, 클라이언트/서버 데이터베이스를 선택해야 하는 경우는 다음과 같다.

- 1) 어플리케이션의 작업이 분조를 다룰 경우
- 2) 보안이 중요한 경우
- 3) 15 명 이상의 사용자가 있는 경우
- 4) 앞으로 1)~3)의 경우가 나타날 가능성이 있는 경우



#### 4. 웹에 기초한 시스템

분산된 데이터 접근을 위해 웹에 기초한 솔루션은 배포와 확장성이란 측면에서 장점이 많은 시스템이다. 그렇지만, 여기에 대해서는 이 책의 범위를 넘기 때문에 자세한 언급을 하지 않는다. 중요한 것은 웹 환경의 경우 동시에 많은 양의 트랜잭션이 일어날 수 있으며, 멀티 쓰레드 환경에 적합한 데이터베이스와 데이터베이스 접근 소프트웨어를 이용해야 한다는 것이다.

#### 5. 멀티-tier 데이터베이스 시스템

클라이언트 소프트웨어와 데이터베이스 서버 사이에 하나 이상의 어플리케이션 서버가 존재하는 모델이다. 볼랜드의 MIDAS 가 대표적인 시스템이다. 썬드 파티에서는 현재 특별히 지원되는 모델이 없다.

##### ● 2 단계: 데이터베이스 포맷/서버의 선택

일단 사용할 데이터베이스의 종류를 결정했으면, 실제로 어떤 데이터베이스를 사용할 것인지 구체적으로 결정해야 한다. 여기에는 특별한 선택의 기준이 존재하기 보다는, 선호도와 친밀도 등의 여러 가지를 고려해 보아야 한다. 파일 서버나 로컬 데이터베이스를 사용하기로 결정하였다면 데이터베이스 포맷을 선택하면 BDE 를 쓸 것인지, 아니면 다른 썬드 파티 도구를 사용할 것인지를 결정하는데 도움이 된다. 예를 들어, DBF 파일에 기초한 도구를 사용하기 원한다면, DBF 포맷을 지원하도록 해야 한다. 파일 서버 어플리케이션의 경우에는 표준적인 데이터베이스 포맷을 이용하는 것이 이후의 확장성을 위해 권장된다. 클라이언트/서버 시스템이라면 데이터베이스 서버를 선택해야 하는데 각 제품마다의 특징이 다르기 때문에, 자신의 요구에 맞는 제품을 선택해야 할 것이다.

##### ● 3 단계: 데이터베이스에 접근할 수 있는 BDE 대체 엔진을 선택한다.

일단 데이터베이스 포맷이나 데이터베이스 서버를 선택하고 나면, 이를 지원하는 썬드 파티 엔진을 고를 수 있다. 같은 데이터베이스를 지원하는 엔진도 그 지원의 정도나 수행 능력에 다소간의 차이를 보이게 되며, 가격에도 차이가 있으므로 이를 잘 고려해서 선택해야 할 것이다. 여기에서 고려해야 할 것들은 다음과 같은 것들이 있다.

##### 1. 포맷과 데이터베이스 서버 종류

##### 2. 수행 능력, vendor 의 지원

3. 배포:

쉬운 배포가 되어야 한다. 어떤 제품은 완전히 실행 파일에 포함되는 것도 있고, DLL 을 요구하는 경우도 있다. 클라이언트/서버 솔루션의 경우에는 데이터베이스에 접근할 수 있는 드라이버를 요구하기도 하며 ODBC, ADO, BDE 와 같은 미들웨어 베이스의 솔루션은 적절한 설치를 요구하는 경우도 있다. 참고로 BDE 는 2~3 MB 정도 크기의 DLL 을 요구한다.

4. 데이터 컨트롤의 사용 여부:

GUI 어플리케이션을 개발할 때, 데이터 컨트롤을 제공하는지 여부는 개발에 상당한 영향을 미친다. 이를 지원하는 레벨은 데이터 컨트롤을 지원하지 않는 경우와 자신의 데이터 컨트롤을 제공하는 경우, 그리고 데이터 컨트롤을 바로 사용할 수 있는 경우로 나눌 수 있다. vendor 에서 제공되는 컨트롤만 써야 하는 경우는 없는 것보다는 낫지만, 그리 좋은 방법은 못된다. 델파이 2 까지만 해도 표준 데이터 컨트롤을 이용하도록 개발하는 것이 어려웠지만, 델파이 3 의 가상 데이터 세트를 이용하면 비교적 쉽게 데이터 컨트롤 들을 지원할 수 있으므로 최근의 BDE 대체 엔진들은 대부분 데이터 컨트롤 을 사용할 수 있다.

5. 리포트 인쇄:

데이터 컨트롤과 마찬가지로 QuickReports, Piparti 등의 다른 리포팅 도구를 지원하는 지 여부도 중요한 고려 대상이 된다.

6. 데이터베이스 서버의 접근도:

BDE 대체 엔진 중에 단순히 데이터베이스 서버에 접속하는 것만 고려할 것이 아니라, 어떤 제품들은 그 이상을 지원하므로 이를 고려한다. 예를 들어, Interbase Objects 나 Direct Oracle Access 와 같은 제품은 데이터베이스 서버의 API 를 직접 이용할 수도 있다. 이것이 그렇게 중요한 것은 아니지만, 특정 서버를 사용하는 경우라면 이런 제품을 고려해 보는 것도 괜찮다. 하지만 반드시 생각해야 하는 것은, 이렇게 특정 서버에 적합한 제품을 이용해 프로그램을 개발한 경우 차후에 데이터베이스 서버를 바꿀 경우가 생긴다면 고쳐야 할 사항이 많아진다는 점을 염두에 두어야 할 것이다.

7. Vendor 의 지원 정책과 버그 수정:

아무래도 같은 나라에 있거나, 지원이 쉬운 회사의 제품을 선택하는 것이 좋을 것이다. 또한, 쉽게 E 메일을 보낼 수 있고, 뉴스 그룹 등의 사용자에게 대한 지원이 많은 회사의 제품에 장점이 있다. 또한 모든 소프트웨어에는 버그가 있기 마련이므로, 얼마나 제품에 성의를 가지고 버그를 고쳐주는지 여부도 중요하다.

## 델파이 데이터베이스 컴포넌트

델파이의 컴포넌트 팔레트의 Data Access 페이지에는 데이터베이스에 대해 작업을 하는 컴포넌트 들이 들어 있다. 여기에 있는 것은 대개 데이터베이스 연결, 테이블과 쿼리 등의 요소들을 캡슐화 한 것이다. 또한, Data Controls 페이지에는 데이터베이스를 눈으로 보고, 수정할 수 있는 데이터 컨트롤 들이 위치하고 있다.

델파이에서 데이터베이스에 접근하려면, DataSource 컴포넌트에 의해 연결할 수 있는 데이터 소스가 있어야 한다. 데이터 소스로는 테이블, 쿼리, 또는 저장 프로시저(stored procedure) 등이 될 수 있으며, 이들은 Data Access 페이지의 TTable, TQuery, TStoredProcedure 등으로 캡슐화 되어 있다. 그러므로, 데이터베이스를 연결할 때에는 데이터 컨트롤은 TDataSource 와 연결하고, TDataSource 의 DataSet 프로퍼티로 데이터 소스로 사용되는 데이터 세트 컴포넌트를 연결하면 기본적인 연결이 끝난다.

참고로, 멀티-tier 데이터베이스 어플리케이션을 사용하거나 플랫폼 파일을 사용할 때에는 데이터 소스로 TClientDataSet 컴포넌트를 이용해야 한다.

### ● 테이블과 쿼리, 저장 프로시저

가장 흔히 사용되는 데이터 세트 컴포넌트는 Table 이다. 테이블 객체는 실제 데이터베이스 테이블을 가리킨다. Table 컴포넌트를 사용할 때에는 DatabaseName 프로퍼티에 사용하고자 하는 데이터베이스의 이름을 지시해 주어야 한다. 여기에는 앨리어스 또는 테이블 파일 등이 들어있는 디렉토리 패스를 입력하거나, 사용된 TDatabase 컴포넌트로 설정해야 한다. 일단 DatabaseName 이 설정되면, 오브젝트 인스펙터에 현재 데이터베이스에서 사용 가능한 테이블의 이름을 목록으로 보여주며 여기에서 테이블을 선택하여 TableName 프로퍼티를 선택해야 한다.

테이블에 못지 않게 많이 사용되는 것이 Query 컴포넌트이다. 쿼리는 SQL 언어 명령으로 이루어져 있다. Query 컴포넌트에서의 테이블은 SQL 프로퍼티 안에 저장되어 있는 SQL 문장의 실행 결과에 의해서 사용된다.

저장 프로시저는 TStoredProc 컴포넌트에 의해 지원된다. 이것은 SQL 서버 데이터베이스의 로컬 프로시저들을 가리키는 것으로, 이러한 프로시저를 실행시켜서 데이터베이스 테이블의 폼 안에 결과를 얻을 수 있다.

### ● 기타 데이터 접근 컴포넌트

Table, Query, StoredProc, DataSource 컴포넌트 외에도 몇 가지 컴포넌트 들이 있다. Database 컴포넌트는 트랜잭션 제어, 보안, 그리고 연결을 제어하는 목적으로 사용된다.

이 컴포넌트는 일반적으로 클라이언트/서버 어플리케이션 안에서 원격 데이터베이스에 연결하기 위해서만 사용되거나, 여러 개의 폼에서 같은 데이터베이스에 연결되는 일을 방지하기 위해 사용된다.

Session 컨트롤은 기존의 데이터베이스와 앨리어스의 목록과 데이터베이스 로그인을 수정하기 위한 이벤트를 포함해서 어플리케이션의 데이터베이스 연결에 대한 제어 방법을 제공하는 컴포넌트이다. BatchMove 컴포넌트는 복사, 붙이기, 업데이트, 값 지우기 등의 배치 작업을 하나 이상의 데이터베이스에 대해 수행할 때 사용된다.

UpdateSQL 컴포넌트는 SQL 문장을 작성해서, 읽기 전용 쿼리를 사용하고 있을 때 다양한 업데이트 작업을 수행할 수 있게 해준다. 이 컴포넌트는 테이블이나 쿼리의 UpdateObject 프로퍼티의 값으로 사용된다.

## 데이터베이스와 앨리어스의 이해 (Understanding Database and Alias)

데이터베이스는 정보를 테이블에 저장한다. 그 밖에도, 데이터베이스에 포함된 정보에 대한 테이블과 인덱스처럼 테이블에 의해 사용되는 객체, 저장 프로시저와 같은 SQL 객체 등이 포함되어 있다.

각각의 BDE 를 이용한 데이터 세트 컴포넌트에는 DatabaseName 이라는 프로퍼티가 있다. 이 프로퍼티는 데이터 세트에 있는 정보를 담고 있는 테이블을 포함한 데이터베이스를 지정한다. 어플리케이션을 설정할 때에는 데이터 세트를 바인드하기 전에 반드시 이 프로퍼티를 이용해서 데이터베이스를 지정해 주어야 한다.

데이터베이스는 BDE 앨리어스(alias)를 가지고 있다. 보통 이런 BDE 앨리어스를 DatabaseName 프로퍼티의 값으로 지정하게 된다. BDE 앨리어스는 데이터베이스에다가 데이터베이스의 환경 정보를 합쳐 놓은 것이라고 생각하면 된다. 앨리어스와 연관된 환경 정보는 데이터베이스 종류에 따라 다르다.

예를 들어, 파라독스 데이터베이스에서는 데이터베이스를 어떤 디렉토리에 저장하고 있는지를 알기만 하면 되지만, 사이베이스에서는 서버에 대한 네트워크 주소, 데이터베이스 이름과 사용자 ID, 패스워드 등의 정보를 알고 있어야만 한다.

BDE Administration 도구나 SQL 탐색기를 이용해서 이런 각각의 BDE 앨리어스를 생성하고, 관리할 수 있다.

데이터베이스에 대해서는 독립된 컴포넌트인 TDatabase 를 사용할 수도 있다. 데이터베이스 컴포넌트를 어플리케이션에 추가하지 않으면, 데이터 세트의 DatabaseName 프로퍼티를 바탕으로 임시 컴포넌트가 하나 생성되어 사용된다.

## 데이터베이스에 접속하기 (Connecting to databases)

텔파이 어플리케이션이 데이터베이스에 접속할 때, 이런 접속 과정은 앞에서도 언급한

TDatabase 컴포넌트가 권장하게 된다. 데이터베이스 컴포넌트는 BDE 세션을 이용해서 데이터베이스 서버에 접속한다. 데이터베이스 컴포넌트는 이 밖에도 BDE 에 기초한 어플리케이션에서의 트랙잭션 관리와 연관된 테이블의 캐쉬 업데이트(cached update)를 할 때 사용된다.

## ● 데이터베이스 컴포넌트

어플리케이션의 데이터베이스 접속은 데이터베이스 컴포넌트에 의해 관리된다. 비록 명시적으로 데이터베이스 컴포넌트를 어플리케이션에 추가하지 않았다고 하더라도 런타임에서 임시로 생성되어 접속을 관리한다. 임시 데이터베이스 컴포넌트 들은 필요할 때마다 생성되어 데이터베이스 접속의 세세한 부분을 개발자가 제어하지 않더라도, 대개의 전형적인 데스크탑 데이터베이스 어플리케이션을 지원한다. 그렇지만, 클라이언트/서버 어플리케이션을 제작할 경우에는 개발자가 데이터베이스 컴포넌트를 이용해서 관리하는 것이 좋다.

### 1. 임시 데이터베이스 컴포넌트의 사용

임시 데이터베이스 컴포넌트는 필요에 따라 자동으로 생성된다. 예를 들어, TTable 컴포넌트를 폼에 올려 놓고, 적당히 프로퍼티를 설정해서 테이블을 열면, 델파이는 임시 데이터베이스 컴포넌트를 생성한다.

임시 데이터베이스 컴포넌트의 주요 프로퍼티 들은 현재의 세션에 의해서 결정된다. 예를 들어, 세션의 KeepConnections 프로퍼티가 True(디폴트) 데이터베이스 접속이 데이터 세트가 닫혀도 계속 유지된다. 이 경우에 DropConnections 메소드를 호출해야 접속이 종료된다. 또한, 세션의 디폴트 OnPassword 이벤트는 어플리케이션이 데이터베이스에 접속하고자 할 때 패스워드를 요구하면 표준 패스워드 입력 상자를 보여준다

임시 데이터베이스 컴포넌트에 의해 설정되는 디폴트 프로퍼티는 일반적이고, 합리적인 값으로 설정되어 있기 때문에, 복잡한 클라이언트/서버 어플리케이션과 같이 많은 사용자와 데이터베이스 접속에 서로 다른 요구 사항을 필요로 하는 데에는 적합하지 않다.

### 2. 디자인 시의 데이터베이스 컴포넌트 생성

데이터베이스 컴포넌트를 디자인 시에 생성하면, 초기 프로퍼티를 쉽게 설정할 수 있고, OnLogin 이벤트를 이용하여 데이터베이스 컴포넌트가 서버에 처음 접속할 때의 접속 문제를 컨트롤 할 수 있다.

### 3. 런타임에서의 데이터베이스 컴포넌트 생성

데이터베이스 컴포넌트를 런타임에서 생성하는 경우는 얼마나 많은 수의 데이터베이스 컴포넌트가 필요한 지 모르지만 어플리케이션이 데이터베이스 연결을 관리하기 위한 데이터베이스 컴포넌트가 반드시 필요할 때를 들 수 있다. 데이터베이스 컴포넌트를 런타임에서 생성할 때에는 항상 유일한 이름을 부여해야 하며, 세션과 연관시켜야 한다.

컴포넌트는 TDatabase.Create constructor 를 호출하면 생성된다. 이때 데이터베이스의 이름과 세션의 이름을 모두 제시해야 한다. 다음의 함수는 런타임에서 데이터베이스 컴포넌트를 생성해주는 함수이다.

```
function RunTimeDbCreate(const DatabaseName, SessionName: string): TDatabase;
var
    TempDatabase: TDatabase;
begin
    TempDatabase := nil;
    try
        //세션이 존재하면 이를 활성화하고, 아니면 새로운 세션을 생성한다.
        Sessions.OpenSession(SessionName);
        with Sessions do
            with FindSession(SessionName) do
                Result := FindDatabase(DatabaseName);
                if Result = nil then
                    begin
                        //새로운 데이터베이스 컴포넌트를 생성한다.
                        TempDatabase := TDatabase.Create(Self);
                        TempDatabase.DatabaseName := DatabaseName;
                        TempDatabase.SessionName := SessionName;
                        TempDatabase.KeepConnection := True;
                    end;
                    Result := OpenDatabase(DatabaseName);
                end;
            end;
        end;
    except
        TempDatabase.Free;
        raise;
    end;
end;
```

다음 코드는 이 함수를 이용하여 디폴트 세션에 대한 데이터베이스 컴포넌트를 생성하는 예를 보여준다.

```
var
  MyDatabase: array[1..10] of TDatabase;
  MyDbCount: Integer;
begin
  MyDbCount := 1;
  ...
  //데이터베이스 컴포넌트를 런타임에서 생성한다.
  begin
    MyDatabase[MyDbCount] := RunTimeDbCreate('MyDb' + IntToStr(MyDbCount), '');
    Inc(MyDbCount);
  end;
  ...
end;
```

데이터 컨트롤의 활용 (Using data controls)

대부분의 데이터 인식 컴포넌트 들은 데이터 세트에 저장된 정보를 대표한다. 이들의 종류로는 하나의 필드를 보여 주는 여러 컨트롤 들과 DBGrid 와 같이 여러 레코드를 한번에 보여주는 컨트롤, 그리고 레코드 사이의 이동과 여러가지 작업을 하도록 제공되는 시각적 도구인 DBNavigator 등이 있다.

● 공통적인 데이터 컨트롤의 특징

데이터 컨트롤은 데이터 세트의 현재 레코드의 필드의 내용을 보여주고, 편집할 수 있도록 해준다. 다음에 데이터 컨트롤의 종류와 이들의 특징을 나열해 보았으니, 참고하기 바란다.

데이터 컨트롤	설 명
TDBGrid	정보를 테이블 형태로 보여 준다. 그리드의 열은 데이터 세트의 컬럼, 행은 레코드를 나타낸다.
TDBNavigator	데이터 세트의 데이터 레코드 사이를 이동, 레코드 업데이트, 삭제 등의 작업을 할 수 있는 비주얼 도구
TDBText	데이터를 라벨의 형태로 보여 준다.
TDBEdit	데이터를 에디트 박스로 나타낸다.

TDBMemo	메모나 BLOB 필드에 저장된 데이터를 메모 형태로 보여준다.
TDBImage	데이터 필드의 내용을 그래픽으로 보여 준다.
TDBListBox	필드에 업데이트할 수 있는 아이템의 리스트를 보여 준다..
TDBComboBox	TDBListBox 와 같으나 직접 텍스트를 입력할 수 있다.
TDBCheckBox	Boolean 필드의 값을 나타내는 체크 박스
TDBRadioGroup	여러 값 중에서 하나의 값만 선택할 수 있는 라디오 그룹 형태의 데이터
TDBLookupListBox	필드의 값에 기초한 다른 데이터 세트의 값을 찾을 수 있다.
TDBLookupComboBox	TDBLookupListBox 와 같으나 직접 텍스트를 입력할 수 있다.
TDBCtrlGrid	여러 데이터 컨트롤을 설정할 수 있는 그리드이다.
TDBRichEdit	데이터 필드를 포맷된 메모의 형태로 보여 준다.

데이터 컨트롤은 디자인 시부터 데이터를 인식할 수 있다. 이렇게 하려면, 데이터 컨트롤의 DataSource 프로퍼티를 활성화된 데이터 소스로 설정하면 되는데, 프로퍼티를 설정하는 것과 동시에 데이터 세트의 내용을 볼 수 있다.

## 1. 데이터 세트와 데이터 컨트롤의 연결

데이터 컨트롤을 데이터 세트와 연결할 때에는 데이터 소스 컴포넌트를 이용한다. 데이터 소스 컴포넌트는 데이터 세트와 데이터 컨트롤의 통로 역할을 한다.

일단 DataSource 프로퍼티를 설정했으면, DataField 프로퍼티를 설정할 수 있다. 여기서 컨트롤에 보여줄 데이터 필드를 선택한다. DataField 프로퍼티는 TDBGrid, TDBCtrlGrid, TDBNavigator 와 같은 컴포넌트에는 적용되지 않는다.

그리고, 데이터 세트의 Active 프로퍼티를 True 로 설정하면 컨트롤에서 데이터를 볼 수 있을 것이다.

## 2. 데이터 편집과 업데이트

데이터 컨트롤은 데이터를 보여주는 것 뿐만 아니라, 데이터를 편집하고 변경된 사항을 업데이트 하는 용도로 사용할 수 있다. 이를 위해서는 데이터 세트의 상태가 dsEdit 로 설정되어 있어야 한다.

데이터 컨트롤과 연결된 데이터 소스의 AutoEdit 프로퍼티는 데이터 컨트롤에 키보드 입력이나 마우스 이벤트가 있을 때, 자동으로 데이터 세트의 상태를 dsEdit 로 바꿔줄 것인지를 결정한다. AutoEdit 프로퍼티의 디폴트 값이 True 이기 때문에, 데이터 컨트롤에 작업을 하면 데이터 세트의 상태는 dsEdit 로 변경된다. AutoEdit 프로퍼티가 False 일 경우에는 TDBNavigator 컨트롤을 사용할 경우에는 Edit 버튼을 클릭해야 편집 상태로 들어갈 수 있으며, 아니면 런타임에서 데이터 세트의 Edit 메소드 등을 호출해서 상태를 변경시켜야 한



다.

데이터 컨트롤의 ReadOnly 프로퍼티는 컨트롤에 표시되는 데이터를 편집할 수 있는지 여부를 결정하게 된다. 이 프로퍼티의 디폴트 값은 False 로 사용자는 데이터를 편집할 수 있다. 사용자가 데이터를 컨트롤에서 편집하지 못하도록 하려면, 이 값을 True 로 설정해야 한다. 데이터 소스 컴포넌트의 Enabled 프로퍼티도 편집 여부를 결정하는데 중요하다. 이 프로퍼티는 데이터 소스와 연결된 데이터 컨트롤에 값을 표시할 것인지를 결정한다. 이 값이 False 이면 사용자가 값을 편집할 수가 없다. 디폴트 값은 True 이다. 또한, 데이터 세트의 ReadOnly 프로퍼티는 사용자가 데이터 세트의 데이터를 편집할 수 있는지 여부를 결정한다. 디폴트 값은 False 이다.

참고:

데이터 세트에는 읽기 전용, 런타임 프로퍼티인 CanModify 라는 프로퍼티가 있다. 이 값은 편집을 할 수 있을 때 True 로 설정된다.

이렇게 데이터 컨트롤에서 데이터를 편집하면, TDBGrid 를 제외하고는 탭(Tab)키를 누를 때 변경된 내용이 필드 컴포넌트에 복사된다. 탭키를 누르기 전에 Esc 키를 누르면, 데이터 컨트롤은 변경된 내용을 원상복귀시킨다. TDBGrid 컴포넌트는 변경된 내용을 사용자가 다른 레코드로 옮겨갈 때에만 복사한다. 그러므로, 레코드를 변경하기 전에 Esc 키를 누르면 레코드의 모든 변경 사항을 원상복귀한다.

### 3. 데이터 디스플레이

어플리케이션에서 여러 레코드를 처리하는 경우에, 레코드가 변경될 때마다 이런 내용을 데이터 컨트롤에 표시하려면 화면이 깜빡이면서 많은 시간이 소요된다. 이럴 때에는 데이터 세트의 DisableControls 메소드를 호출하여 데이터 인식 컨트롤에 레코드 내용을 표시하지 않도록 하는 것이 현명하다. 일단 레코드를 검사하는 부분이 모두 종료되면 다시 EnableControls 메소드를 호출하여 컨트롤에 데이터 세트의 내용을 표시하도록 한다.

지금까지, 설명만 해서 지루할 지 모르겠다. 그러면, 이쯤에서 간단한 예제를 이용하여 데이터 컨트롤을 이용하는 기본적인 방법과 EnableControls 와 DisableControls 메소드를 이용할 때와 이를 그대로 사용할 때의 시간 차이를 재도록 해보자.

먼저 폼에다가 TDBGrid, TDataSource, TTable 컴포넌트를 하나씩 올려 놓고, TButton 컴포넌트를 2 개 올려 놓는다.

OrderNo	ItemNo	PartNo	Qty	Disc
1104	15	2612	19	
1104	285	2619	1	
1104	286	2630	1	
1104	287	2632	15	

Button1, Button2 의 Caption 프로퍼티를 ‘시작 1’, ‘시작 2’로 설정한다. 여기에서 Button1 을 선택하면 아무런 조치 없이 처음부터 끝까지 레코드를 변경할 것이다. Button2 를 선택하면 데이터 세트(Table1)의 DisableControls, EnableControls 메소드를 사용해서 각각의 시간을 GetTickCount 함수를 이용해서 측정할 것이다.

먼저 테이블과 데이터 컨트롤을 연결하도록 하자. 방법은 간단하다 DBGrid1 의 DataSource 프로퍼티를 DataSource1 으로 설정하고, DataSource1 의 DataSet 프로퍼티를 Table1 으로 설정한다. 실제 테이블을 지정할 차례인데, Table1 의 DatabaseName 에는 DBDEMOS 앨리어스를 선택하여 데모 데이터베이스 테이블을 사용할 수 있도록 한다. TableName 프로퍼티의 드롭-다운 버튼을 클릭하면 여러 테이블 들을 나열하게 되는데, 여기서는 비교적 레코드의 수가 많은 테이블을 선택하는 것이 시간의 차이를 느끼기 좋으므로 레코드 수가 많은 Items.db 테이블을 선택하도록 한다. 이제 Table1 의 Active 프로퍼티를 True 로 설정하면, DBGrid1 컨트롤에 앞의 그림과 같이 레코드가 나타날 것이다.

이제 Button1 과 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Start, Finish: Integer;
begin
    Start := GetTickCount;
    Table1.First;
    while not Table1.Eof do
        Table1.Next;
    Finish := GetTickCount;
    ShowMessage(IntToStr(Finish - Start) + ' msec !');
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

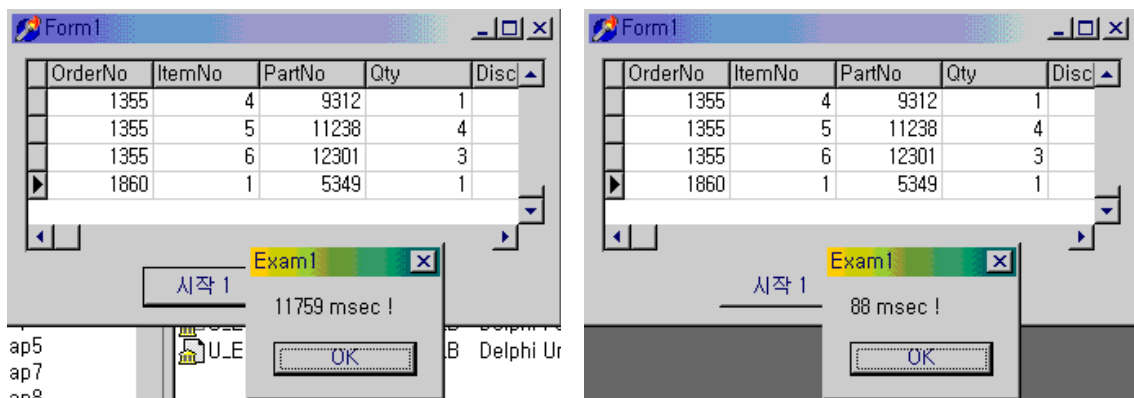
```

var
    Start, Finish: Integer;
begin
    Start := GetTickCount;
    Table1.DisableControls;
    Table1.First;
    while not Table1.Eof do
        Table1.Next;
    Finish := GetTickCount;
    Table1.EnableControls;
    ShowMessage(IntToStr(Finish - Start) + ' msec !');
end;

```

이와 같이 데이터 세트의 레코드들 사이를 이동할 때에는 데이터 세트의 First, Next, Previous, Last 메소드를 사용하며 데이터 세트의 레코드가 첫번째 레코드의 이전으로 이동했다면 Bof, 마지막 레코드의 다음으로 이동했다면 Eof 가 설정되므로 이를 이용하여 검색을 하게 된다.

앞의 이벤트 핸들러 코드에서 대부분의 내용이 비슷하지만, Button2 이벤트 핸들러에서 DisableControls, EnableControls 메소드를 호출한 것만 다르다. 어플리케이션을 호출하여 이들 메소드의 효과를 알아보도록 하자. 필자의 컴퓨터에서 실행한 결과는 다음과 같다.



결과는 무려 100 배가 넘는 속도의 차이가 난다. 이는 데이터 세트 자체의 처리 속도보다, 화면에 내용을 뿌리는 데에 소요되는 시간이 월등히 많다는 것을 의미한다. 그러므로, 많은 양의 레코드를 동시에 처리할 때에는 반드시 DisableControls, EnableControls 메소드를 사용할 것을 권하고 싶다.

#### 4. 데이터 리프레쉬 (Refreshing data)

데이터 세트의 Refresh 메소드는 로컬 버퍼를 비우고, 데이터 세트에서 데이터를 다시 가져오는 메소드이다. 보통 이 메소드는 데이터가 변경되었을 것으로 생각될 때, 컨트롤에 표시되는 내용을 새롭게 하기 위해 사용한다. 흔히 여러 명의 사용자가 동시에 데이터를 편집할 경우에 이런 변경 사항을 빠르게 갱신해줄 필요성이 있다.

**참고:**

리프레쉬가 간혹 예상치 못한 결과를 가져올 때가 있다. 예를 들어, 사용자가 다른 어플리케이션이 삭제한 레코드를 보고 있다면, refresh 메소드를 호출하면 순간적으로 컨트롤에 보이던 데이터가 모두 사라져 버린다.

● Lookup 리스트와 콤보 박스를 이용한 데이터 표시와 편집

TDBLookupListBox 와 TDBLookupComboBox 컴포넌트는 보통 마스터/디테일 테이블의 관계나, 특정 필드 값에 대한 내용을 찾거나 할 때 유용하게 사용된다.

예를 들어, OrderTable 을 보여주는 order 폼이 있다고 하자. OrdersTable 에는 고객의 ID 에 해당되는 CustNo 필드가 있지만(FK, Foreign Key), 회사 이름, 회사 주소, 집 주소, 전화 번호 등의 고객에 대한 정보는 CustomerTable 에 존재하게 된다. 이때 order 폼에서 고객에 대한 정보를 찾을 때, CustNo 필드가 아닌 회사 이름과 같은 고객 정보를 이용해서 검색을 하고자 한다면 어떻게 해야 할까 ? 이럴 때에는 TDBLookupListBox 를 이용해서 CustomerTable 의 모든 회사 이름을 보여 주게 하고, 사용자가 회사 이름을 선택하면 CustNo 의 값을 이용해서 지정된 Order 를 보여주면 된다.

1. Lookup 필드에 기초한 리스트의 지정

Lookup 필드로 사용할 리스트 박스 아이템을 지정하려면 컨트롤에 연결될 데이터 세트에는 반드시 lookup 필드가 정의되어 있어야 한다. 정의되어 있다면, 리스트 박스의 DataSource 프로퍼티를 lookup 필드를 가지고 있는 데이터 세트의 데이터 소스 컴포넌트로 설정하고, DataField 프로퍼티에서 사용할 lookup 필드를 선택하면 된다.

이렇게 하면, lookup 리스트 박스 컨트롤과 연결된 테이블이 활성화될 때, 컨트롤은 데이터 필드가 lookup 필드라는 사실을 알게 되고, 적절한 값을 표시한다.

2. 다른 데이터 소스에 기초한 리스트의 지정

데이터 세트에 대한 lookup 필드를 정의하지 않은 경우에는 다른 데이터 소스 컴포넌트를

이용해서 이를 처리해야 한다. 일단 리스트 박스의 DataSource 프로퍼티를 컨트롤에 대한 데이터 소스로 설정한다. Lookup 값들을 삽입하여 리스트로 보여줄 필드를 DataField 프로퍼티로 지정하는데, 이 필드는 lookup 필드가 아니다. 그리고, ListSource 프로퍼티를 look up 하기 원하는 값들이 있는 필드를 포함한 데이터 세트에 대한 데이터 소스로 설정한다. Lookup 키로 사용하고자 하는 필드를 KeyField 프로퍼티에 설정한다. 이때 KeyField 프로퍼티로 설정한 필드에 인덱스가 되어 있으면 수행속도가 빠르다. 마지막으로 반환하길 원하는 값을 가진 필드를 ListField 로 설정한다.

## DBGrid 의 활용

TDBGrid 컴포넌트는 테이블 형태의 그리드에 레코드를 보여주고, 편집할 수 있는 데이터 컨트롤이다. 데이터 컨트롤 중에서도 가장 흔히 사용되는 컨트롤이지만, 과거 델파이 3 까지는 기본으로 제공되는 그리드의 기능이 뛰어나지 않아서 많은 수의 공개, 셰어 컴포넌트가 소개된 바 있다. 그렇지만, 델파이 4 에서 제공되는 DBGrid 는 비교적 뛰어난 성능을 보여주며 그 동안 필요로 해왔던 기능을 많이 추가하였다.

그리드 컨트롤의 Columns 프로퍼티는 TDBGridColumns 객체의 wrapper 이다.

TDBGridColumns 객체는 그리드 컨트롤의 모든 컬럼을 대표하는 TColumn 객체의 컬렉션으로, 개발자는 컬럼 에디터(Columns editor)를 이용하여 디자인 시에 컬럼의 속성을 설정할 수 있다.

Columns.State 프로퍼티는 그리드에 의해 자동으로 설정되는 런타임 프로퍼티로, 디폴트 값은 그리드에 persistent 컬럼 객체가 존재하지 않거나 데이터 세트에 persistent 필드 컴포넌트가 없다는 의미인 csDefault 이다.

### ● 디폴트 state 에서 그리드 컨트롤 이용하기

그리드의 Columns.State 프로퍼티의 값이 csDefault 이면, 레코드 들은 그리드의 데이터 세트의 각 필드 들의 프로퍼티에 의해서 나타나게 된다. 그리고, 그리드의 컬럼의 순서는 데이터 세트의 필드 순서와 일치한다. 그리드 컨트롤을 이용해서 동적으로 컬럼을 생성해서 테이블의 내용을 보여주면 보다 다양하고, 다양한 형태로 그리드를 활용할 수 있다. 예를 들어, 과라독스 테이블을 처음에는 보여주다가, SQL 쿼리의 결과를 DataSource 프로퍼티를 변경해서 보여주게 하는 등의 변화를 줄 수 있다.

#### 참고:

그리드의 Columns.State 프로퍼티를 런타임에서 csDefault 로 변경할 경우, 그리드에 있는 모든 컬럼 객체가 삭제된다. 그리고, 동적 컬럼을 생성한다.

## ● Persistent 컬럼의 이해

그리드에 사용하기 위한 persistent 컬럼 객체를 생성하면, 이들은 그리드의 데이터 세트와 느슨하게 연결되어 있다. 꼭 알아두어야 하는 것은 persistent 컬럼은 연결된 필드 컴포넌트와는 독립적이라는 것이다. 물론 디폴트 값을 필드 컴포넌트에서 가져오지만, 독자적인 형태를 가지고 있다. 만약 persistent 컬럼의 FieldName 프로퍼티가 비어 있거나, 필드의 이름이 현재 연결된 데이터 세트의 필드 이름과 맞지 않는 경우 Field 프로퍼티의 값은 NULL 이 되고, 셀의 내용은 비게 된다. 이렇게 비어 있는 컬럼에 비트맵을 그려 넣거나, 다른 문자를 채우는 등의 작업을 셀의 디폴트 드로잉 메소드를 오버라이드하여 구현할 수도 있다.

어떤 경우에는 2 개 이상의 persistent 컬럼을 같은 필드와 연결할 수도 있다. 예를 들어, 넓은 그리드에 특정 부품의 번호를 좌측과 우측 끝에 같이 보여주고 싶을 경우에는 이들 컬럼의 필드를 같은 부품의 번호를 나타내도록 설정하면 된다.

### 참고:

Persistent 컬럼은 데이터 세트의 필드와 반드시 연결될 필요가 없고, 여러 개의 컬럼이 하나의 필드를 나타낼 수도 있기 때문에 그리드의 FieldCount 프로퍼티는 그리드의 컬럼 수보다 작을 수 있다. 또한, 현재 선택된 컬럼이 필드와 연결되어 있지 않은 경우에는 SelectedField 프로퍼티가 NULL 로 설정된다.

그리드의 형태를 디자인시에 변경하려면, 일단 컬럼 에디터(Column Editor)를 실행하고, 여기에서 persistent 컬럼 객체의 세트를 생성해야 한다. 이렇게 하면, 그리드의 State 프로퍼티는 자동으로 csCustomized 로 설정된다.

컬럼 에디터에서는 컬럼을 추가, 편집, 삭제할 수 있게 되어 있다. 이런 식으로 컬럼 에디터에서 추가된 컬럼의 이름은 기본적인 디폴트 이름과 순서를 가지게 된다 (예: 0-TColumn). 이렇게 새로운 컬럼에 대한 필드를 설정하기 위해 FieldName 프로퍼티를 나타내고자 하는 필드로 지정한다. 또한, 새로운 컬럼의 타이틀을 설정할 때에는 Title 프로퍼티의 Caption 옵션의 값을 지정하면 된다. 컬럼 에디터에 나타나는 컬럼의 순서는 그리드에 나타나는 컬럼의 순서가 되기 때문에, 컬럼을 드래그-드롭하여 컬럼의 순서를 재배치할 수 있다.

런타임에서 컬럼을 추가할 때에는 다음과 같이 하면 된다.

```
DBGrid1.Columns.Add;
```

또한, persistent 컬럼을 삭제하려면 단순히 컬럼 객체를 해제하면 된다.

```
DBGrid1.Columns[5].Free;
```

#### 1. Lookup 리스트 컬럼의 정의

분리된 lookup 테이블에서 값들의 드롭-다운 리스트를 보여주도록 컬럼을 설정하려면 먼저 데이터 세트에서 lookup 필드 객체를 정의해야 한다. 일단 lookup 필드가 정의되면 컬럼의 FieldName 프로퍼티를 lookup 필드로 설정하고, ButtonStyle 프로퍼티를 cbsAuto 로 설정한다. 이렇게 하면, 그리드는 자동으로 콤보 박스와 같은 형태의 드롭-다운 버튼을 이 컬럼의 셀이 에디트 모드로 들어갈 때 보여주게 되며, 이 버튼을 누르면 lookup 필드에 정의된 테이블의 값들을 드롭-다운 리스트에 보여주게 된다.

#### 2. Pick 리스트 컬럼의 정의

Pick 리스트 컬럼은 컬럼 필드가 보통 필드이고, 드롭-다운 리스트가 lookup 테이블이 아닌 PickList 프로퍼티 값들에 의해 마치 lookup 리스트 컬럼처럼 동작한다는 점을 제외하면 lookup 리스트 컬럼과 비슷하다. Pick 리스트 컬럼을 정의하려면, 먼저 컬럼 리스트 박스에서 컬럼을 선택하고, ButtonStyle 프로퍼티를 cbsAuto 로 설정한다. 오브젝트 인스턴스에서 Picklist 프로퍼티를 더블 클릭하여 문자열 리스트 에디터(string list editor)를 불러낸 후, 드롭-다운 리스트에 보여줄 값들을 리스트를 입력한다.

#### 3. 컬럼에 버튼 올려 놓기

컬럼의 정상적인 셀 에디터 우측에 ellipsis 버튼(...)을 올려 놓을 수 있다. Ctrl+Enter 를 누르거나, 이 버튼을 마우스로 클릭하면 OnEditButtonClick 이벤트가 발생한다. 이 버튼을 이용하면 컬럼의 데이터에 대해 보다 자세한 내용을 볼 수 있도록 할 수 있다. 예를 들어, 주문장의 요약 내용을 보여주다가, 함께 컬럼의 ellipsis 버튼을 클릭하면 각 아이템의 내역과 세금 계산 방법 등을 보여주게 할 수 있다. 그래픽 필드의 경우에는 이 버튼을 누르면 이미지를 보여주도록 할 수 있다.

Ellipsis 버튼을 컬럼에 올려 놓으려면, 먼저 컬럼 리스트 박스에서 컬럼을 선택하고 ButtonStyle 프로퍼티를 cbsEllipsis 로 설정한다. 그리고, 마지막으로 OnEditButtonClick 이벤트 핸들러를 작성하면 된다.

#### 4. 주요 컬럼 프로퍼티

컬럼의 프로퍼티는 컬럼의 셀에 데이터를 어떤 식으로 보여줄 것인지를 결정한다. 다음에 컬럼 객체에 대한 주요 프로퍼티에 대해서 설명하였다.

프로퍼티	설 명
Alignment	데이터의 정렬 방법을 결정한다. 연결 필드의 Alignment 프로퍼티가 디폴트 값
ButtonStyle	디폴트 값은 cbsAuto 로 연결된 필드가 lookup 필드이거나, PickList 프로퍼티에 데이터가 있으면 드롭-다운 리스트를 보여준다. cbsEllipsis 로 설정하면, 셀의 우측에 ellipsis 버튼을 보여주는데, 이 버튼을 클릭하면 OnEditButtonClick 이벤트가 발생한다. cbsNone 으로 설정하면 정상적인 에디트 컨트롤로 사용된다.
Color	셀의 배경 색을 결정한다. DBGrid 의 Color 프로퍼티가 디폴트 값이다.
DropDownRows	드롭-다운 리스트에 보여줄 텍스트의 줄 수. 디폴트 값은 7 이다.
Expanded	컬럼이 확장될 수 있는지 결정한다. ADT 나 배열 필드의 컬럼에 적용된다.
FieldName	컬럼과 연결된 필드의 이름을 지정한다.
ReadOnly	True 이면 사용자가 데이터를 편집할 수 없다. 디폴트 값은 False 이다.
Width	컬럼의 폭을 결정한다. 디폴트 값은 연결 필드의 DisplayWidth 값이다.
Font	컬럼 텍스트의 폰트를 지정한다. DBGrid 의 Font 프로퍼티가 디폴트 값이다.
PickList	드롭-다운 리스트에 보여줄 값의 리스트를 결정한다.
Title	선택된 컬럼의 타이틀에 대한 프로퍼티를 설정한다.

#### ● ADT 와 배열 필드 나타내기

데이터 세트의 ObjectView 프로퍼티 값에 따라, 그리드는 ADT 나 배열 필드를 직접 보여주거나, 필드를 펼치거나 접어서 보여줄 수 있다. ObjectView 프로퍼티의 값이 True 이면 객체 필드를 펼치거나 접어서 보여줄 수 있는데, 필드가 펼쳐지면 각각의 자식 필드가 컬럼의 타이틀 바 아래에 나열된다. 필드가 접히면 하나의 컬럼이 자식 필드 들을 포함한 문자열로 나타난다. 필드의 내용을 펼치거나 접는 것은 필드의 타이틀 바에서 화살표를 클릭해서 실행할 수 있다. ObjectView 프로퍼티 값이 False 이면, 각각의 자식 필드가 분리된 컬럼으로 나타난다.

프로퍼티	클래스	설 명
Expandable	TColumn	값이 True 이면 컬럼이 확장되어 분리된 자식 필드를 보여줄 수 있다.
Expanded	TColumn	컬럼이 확장되었는지를 나타낸다.
MaxTitleRows	TDBGrid	그리드에 나타날 수 있는 타이틀 행의 총 수를 결정한다.
ObjectView	TDataSet	필드가 분리된 컬럼으로 보이게 할 지, 객체 필드를 확장하거나 접을 수 있게 할 지를 결정한다.



ParentColumn	TColumn	자식 필드 컬럼을 소유할 TColumn 객체를 지정한다.
--------------	---------	---------------------------------

## ● 그리드 옵션의 설정

그리드의 Options 프로퍼티를 이용하면, 그리드의 기본적인 행동양식과 형태를 결정할 수 있다. Options 프로퍼티는 여러 가지 프로퍼티로 구성된 세트이다. 다음에 Options 프로퍼티에서 설정할 수 있는 내용 들을 나열하였다.

옵 션	목 적
dgEditing	디폴트 값은 True 로이다. 그리드에서 레코드를 편집, 삽입, 삭제가 가능하게 할 것인지 여부를 결정한다.
dgAlwaysShowEditor	필드가 선택되면, 자동으로 Edit state 로 될 것인지 여부를 결정한다. 디폴트 값은 False 이다.
dgTitles	그리드 위에 필드 이름을 표시할 것인지 여부를 결정한다. 디폴트 값은 True 이다.
dgIndicator	그리드 좌측에 indicator 컬럼을 표시할 지를 결정한다. 레코드를 삽입하면, 화살표가 별표(*)로 변하고, 편집 시에는 '!'의 형태로 변한다.
dgColumnResize	컬럼 크기를 변경할 수 있는지 여부를 결정한다. 디폴트 값은 True 이다.
dgColLines	컬럼 사이에 라인을 그릴 것인지 여부를 결정한다. 디폴트 값은 True 이다.
dgRowLines	레코드 사이에 라인을 그릴 것인지 결정한다. 디폴트 값은 True 이다.
dgTabs	True(디폴트)이면 필드 사이를 옮길 때 탭키를 사용하며, False 이면 탭키를 누를 때 그리드 컨트롤에서 빠져나간다.
dgRowSelect	True 이면 선택을 할 때 그리드의 한 레코드 전체가 선택되며, False(디폴트)이면 레코드의 특정 필드만 선택된다.
dgAlwaysShowSelection	True(디폴트)이면 다른 컨트롤에 포커스가 있더라도 선택된 부분이 보이게 하며, False 이면 그리드에 포커스가 있을 때에만 보인다.
dgConfirmDelete	레코드를 삭제할 때 물어볼 지 여부를 결정한다. 디폴트 값은 True 이다.
dgCancelOnExit	포커스가 그리드에서 벗어날 때 레코드를 삽입하는 도중 이었다면, 이 레코드의 삽입을 취소할 지 여부를 결정한다. 디폴트 값은 True 이다.
dgMultiSelect	사용자가 그리드에서 인접하지 않은 레코드 들을 여러 개 선택할 수 있는지 여부를 결정한다. 디폴트 값은 False 이다.

## ● 그리드에서 편집하기

런타임에서 데이터를 수정하고, 삽입할 수 있게 하려면, 데이터 세트의 CanModify 프로퍼티는 True 이어야 하며, 그리드의 ReadOnly 프로퍼티가 False 이어야 한다.

사용자가 레코드를 편집할 때, 각 필드의 변경된 내용은 내부의 레코드 버퍼에 저장된다. 그렇지만, 다른 레코드로 넘어가지 전에는 내용이 post 되지 않는다. 포커스가 그리드에서 다른 폼으로 넘어가더라도 이 내용은 레코드가 변경되기 전에는 post 되지 않는데, 레코드가 post 되기 전에 Esc 를 누르면 변경된 내용을 취소할 수 있다.

# ● 런타임에서 사용자 행동에 반응하기

그리드를 이용하여 작업을 할 때, 사용되는 이벤트에는 어떤 것들이 있을까 ? 그리드에서 사용되는 이벤트도 꽤나 많을 것이다. 다음에 그리드에서 흔히 사용되는 이벤트에 대해서 정리해 보았다.

이벤트	설 명
OnCellClick	그리드에서 셀을 클릭하면 발생한다.
OnColEnter	그리드의 컬럼으로 이동할 때 발생한다.
OnColExit	그리드의 컬럼에서 떠날 때 발생한다.
OnColumnMoved	사용자가 컬럼을 새로운 위치로 이동할 때 발생한다.
OnDbClick	그리드를 더블 클릭하면 발생한다.
OnDragDrop	그리드에서 드래그-드롭을 할 때 발생한다.
OnDragOver	그리드에서 드래그를 할 때 발생한다.
OnDrawColumnCell	어플리케이션이 개개의 셀을 그릴 필요가 있을 때 발생한다.
OnDrawDataCell	이전 버전에서 사용하던 이벤트로, 어플리케이션이 State 가 csDefault 일 때 개개의 셀을 그릴 필요가 있을 때 발생한다.
OnEditButtonClick	컬럼에서 ellipsis 버튼을 클릭할 때 발생한다.
OnEndDrag	그리드에서 드래깅을 중지할 때 발생한다.
OnEnter	그리드가 포커스를 가질 때 발생한다.
OnExit	그리드가 포커스를 잃을 때 발생한다.
OnStartDrag	그리드에서 드래깅을 시작할 때 발생한다.
OnTitleClick	컬럼의 타이틀을 클릭할 때 발생한다.

## 알아두면 유용하다 !

데이터베이스 어플리케이션을 텔파이를 이용하여 작성할 경우에 가장 많은 고려 대상이 되는 부분이 프로그램의 사용자 인터페이스이다. 개발기간이 짧고 간단한 어플리케이션의 경우 텔파이의 기본적인 데이터 접근 컴포넌트 들과 각종 데이터 인식 컨트롤을 사용하여 프로그램을 디자인하게 된다.

그렇지만, 대형 SQL 을 사용하기 위한 세션의 처리 사용자 로그인 등의 작업을 고려해야

한다면, 이런 데이터 인식 컨트롤을 사용하지 않는 것을 권한다.

델파이 초보자들은 TTable 을 사용하여 데이터 접근을 유지한 상태에서 데이터 인식 컨트롤을 사용하여 프로그램을 작성하는 경우가 많다. 필자가 주변에 확인해본 바로는 대다수의 전문 프로그래머들 역시 데이터 모듈에 TDatabase, TQuery, TDataSource 컴포넌트를 각각 하나씩 올려놓고 데이터 인식 컴포넌트는 직접 사용하지 않는 경우가 많다.

물론, 이런 컨트롤을 잘 사용하면 문제가 없겠지만, 많은 프로그래머 들이 자신들이 직접 각종 컨트롤에 데이터를 표시하도록 코딩하는 이유는 내부적인 기능에 대한 철저한 연구와 윈도우의 고질적인 문제점인 리소스 문제를 해결하기 위함이고, 사용자가 원하는 시점에서 데이터를 조절하기 쉽도록 하기 위해서이다.

## 대량의 레코드 관리 요령

간단한 형태의 레코드를 다룰 때에는 고려하지 않아도 될 내용을, 레코드의 크기가 커질 때에는 이를 효율적으로 사용하기 위해 조금은 특별하게 처리할 필요가 있다. 이때 많이 사용되는 기법으로는 다음의 두가지가 있다.

- 마스터/디테일 형태

이 형태는 주가 되는 마스터 테이블의 세부적인 내용을 포함하는 디테일 테이블을 연결해서 사용하는 것이다. 예를 들어, 고객관리 프로그램의 경우 고객명단을 확인하면서 각 고객에 대한 정보나, 물건과 돈의 입출금 내역 등을 확인하는 형태를 말한다. 보통 DBGrid 를 많이 이용한다.

- 드릴-다운(Drill-down) 형태

이 형태는 마스터/디테일 연결 방법에서 한단계 발전한 것으로 요즈음 각광 받는 의사결정 시스템의 기초적인 단계로 활용할 수 있다. 마스터/디테일 연결에서는 사용자가 데이터를 바라보는 시점이 개발자가 정해진 시점으로 밖에 볼 수 없으며 다양한 폼으로 구성된 개별적인 데이터로서만 확인할 수 있다.

그렇지만 드릴-다운 형태를 이용하면 사용자가 원하는 조건과 형식으로 데이터를 바라볼 수 있다. 델파이에서는 Decision Cube 라는 컴포넌트 세트가 이런 역할을 담당한다.

## 데이터 분석과 문서 작성

델파이에서는 좀더 통계적인 자료를 시각화 시켜줄 수 있는 TChart 컴포넌트가 제공된다. 이 컴포넌트는 DBAware 를 지원하는 TDBChart 로 확장이 되어있는데, 이 차트를 사용하

면 여러가지 자료들을 각종 그래프로 분석하여 자료를 제시하여 줄 수 있기 때문에 시각적인 효과를 높일 수 있다.

또한, 데이터베이스에 저장된 내용을 문서로 표시하기 위해서는 델파이 초기 버전에서는 쉘 어웨어로 팔리던 쿼리 리포트를 델파이 3 에서부터 정식 컴포넌트로 등록을 하여, 델파이에서 간단한 문서부터 다양한 문서의 형태까지 지원한다. 또한, 따로 쿼리 리포트를 구입하면 여러가지 다양한 리포트 디자이너를 사용할 수 있다.

필자는 소규모의 데이터베이스를 사용하는 경우에는 쿼리 리포트를 사용하지만 국내의 현실상 이것 만으로는 만들어 내기가 힘든 문서가 많다고 생각한다. 그렇기 때문에, 대규모의 문서와 일반문서와 똑같은 형태로 구성되는 문서를 만드는 경우에는 Crystal Report 를 유용하게 사용하고 있다.

## 정 리 (Summary)

이번 장에서는 델파이가 지원하는 데이터베이스 모델과 여러가지 컴포넌트의 사용 방법에 대해서 간단하게 설명하였다. 개념적인 내용을 이해한다면 이번 장의 목적은 달성한 것이라고 보겠다. 다음에 이어지는 여러 장에서는 구체적인 예제를 가지고, 데이터베이스 어플리케이션을 제작하는 기법들을 알아볼 것이다.

## 데이터베이스 도구의 활용

## (Using Delphi's Database Tools)

텔파이는 텔파이를 이용해서 데이터베이스 프로그래밍을 할 때에 사용할 수 있는 몇 가지 유틸리티 프로그램을 제공한다. 데이터베이스 데스크탑(Database Desktop), BDE Administrator (BDE 관리자), SQL 탐색기(SQL Explorer), SQL 빌더(SQL Builder), SQL 모니터(SQL Monitor) 등의 프로그램이 그것이다.

이번 장에서는 텔파이에서 제공되는 여러 유틸리티 프로그램의 간단한 사용 방법과 활용 방법에 대해서 알아보도록 한다. SQL 빌더에 대해서는 15 장에서 알아볼 것이다.

## 데이터베이스 데스크탑 사용하기

가장 흔하게 사용하게 되는 유틸리티 프로그램이 바로 데이터베이스 데스크탑이다. 데이터 데스크탑의 역할은 다양한 데이터 소스의 데이터 세트를 열어서 볼 수 있고, 여기에 필드를 추가하거나 데이터의 수정을 하는 등의 여러가지 작업을 할 수 있게 하는 것이다.

다음에 데이터베이스 데스크탑의 기본 스피드 메뉴와 Restrucure 기능에 대해 설명하였다. 그 밖에도 여러가지 유틸리티 기능과 앨리어스 관리자(Alias Manager...) 등의 기능이 있으나, 여기에 대해서는 도움말을 참고하기 바란다.

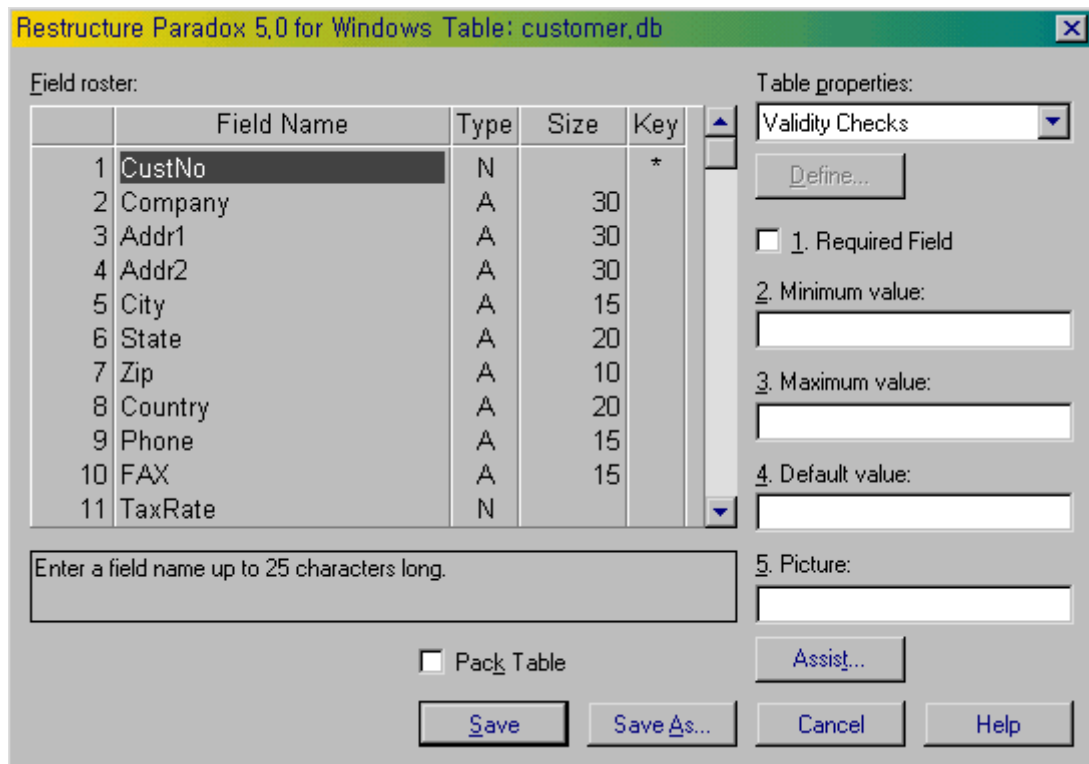
● 데이터베이스 데스크탑의 기본 스피드 메뉴

데이터베이스 데스크탑을 실행하면 3 개의 스피드 버튼을 볼 수 있다. 제일 처음에 나온 아이콘을 이용하면, 텔파이에서 지원되는 모든 테이블을 열어볼 수 있다. 두번째 아이콘은 쿼리를 동일한 데이터 소스로 연다. 세번째 아이콘을 이용하면 다양한 SQL 지정 소스를 열 수 있다.

데이터베이스 데스크탑을 이용하여 실행할 수 있는 간단한 작업을 시도해 보자. 먼저 데이터베이스 데스크탑을 실행하고, 스피드 버튼 중에서 Open Table 아이콘을 선택한다. 그러면, 파일 열기 대화상자가 나타나는데 파일 유형을 열고자 하는 파일 유형으로 지정하고 데이터베이스 파일이 있는 디렉토리에 가서 파일을 열면 된다. 또한, 파일 열기 대화 상자의 제일 마지막에 있는 콤보 박스를 클릭하면 사용가능한 앨리어스를 선택할 수 있다. 여기서 DBDEMOS 를 선택하고, 보이는 데이터베이스 파일 중에서 ‘Customer.db’ 파일을 선택하도록 하자. 그러면, 테이블의 내용을 보여주면서 다음과 같은 스피드 바가 생성될 것이다.



앞쪽에 있는 3 가지 스피드 버튼은 데이터의 자르기(cut), 복사(copy) 및 붙여넣기(paste) 기능을 수행한다. 네번째 버튼은 테이블의 재구성(restructure)을 하는 역할을 한다. 'Customer.db' 파일에 대해서 Restructure 를 선택하면, 다음과 같은 화면이 나타난다. 파라독스인 경우에는 이렇지만, 디베이스의 경우는 상당히 다르다.



이 대화상자의 오른쪽에 보면, Table Properties 콤보 박스를 이용하여 테이블 등록 정보를 선택할 수 있다. 그 밑에 있는 편집 필드 들은 프로그래머가 선택하는 등록 정보(파라독스의 경우에는 Validity Check, 디베이스의 경우 Indexes 가 기본 등록 정보이다)로 지정된다. 콤보 박스를 선택하고, 등록 정보 목록을 통해 화면을 이동하면, 이 대화 상자에 포함되어 있는 모든 테이블과 데이터의 형태가 변한다.

좌측에 있는 필드를 정의하고, 편집하는 부분을 Field Roaster 라고 한다. 여기에서 테이블에 필요한 필드를 추가할 수도 있고, 기존의 필드에 대한 속성을 변경할 수도 있다. 필드를 추가하는 방법은 간단히 Insert 키를 누르기만 하면 된다. 자리가 확보되면 필드 이름을 기록하고, Type 탭에 가서 스페이스바를 누르면 사용 가능한 필드 데이터 형이 나열되며, 여기에서 적당한 데이터 형을 고르면 된다. 가장 우측의 Key 탭에는 필드가 Primary Key 가 된다면 더블 클릭하여 '\*' 표시를 해 준다.

스피드 바에서 Restructure 버튼 옆에 있는 여러 버튼 들은 레코드 사이를 이동하는 역할을 한다. 제일 우측에서 2 번째 버튼은 Field View 버튼으로 이 메뉴를 이용하여 디스플레이

이 영역에 나타나지 않는 메모 필드와 같은 필드의 내용을 볼 수 있다. 필드 사이를 이동할 수 있지만, 다른 키는 무시된다. 이 기능을 이용하면 내용이 변할 염려 없이 특정 필드의 데이터를 제대로 볼 수 있다.

제일 오른쪽에 있는 Edit Data 버튼은 선택한 필드에 어떤 내용을 입력하면, 데이터베이스 데스크탑이 그 내용을 변경한다. 이때 Field View 및 Edit Data 기능은 해제된다. 또한, 필드에 입력을 위한 삽입 필드가 나타난다.

## ● Restructure 대화 상자 이해하기

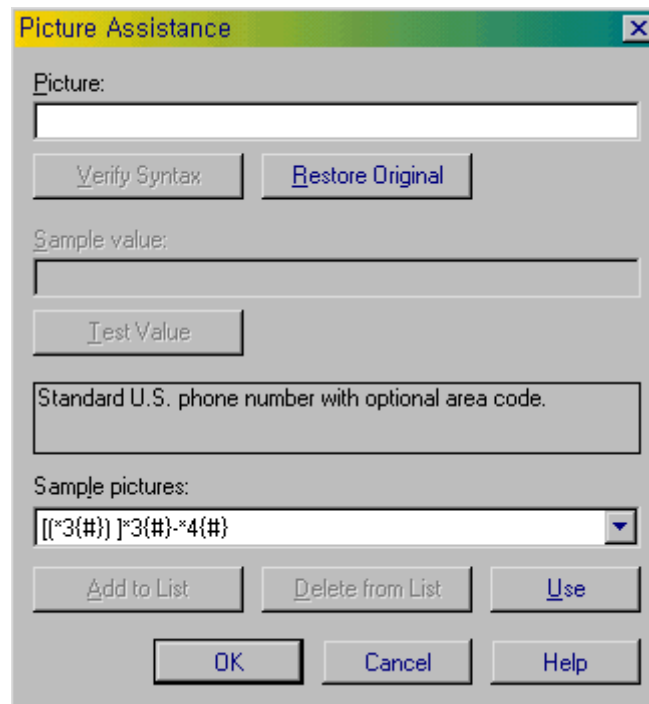
데이터베이스 데스크탑의 가장 큰 기능을 든다면 테이블의 필드를 추가, 삭제, 편집을 하게 되는 restructure 기능을 들 수 있을 것이다. 물론, 파라독스와 디베이스가 다른 면이 많다. 그 밖에 DBMS는 각기 다른 유형의 테이블 등록 정보를 사용할 것이다.

여기에서 관심을 가지고 보아야 할 부분은 Table Properties 콤보 상자에 나타나는 리스트이다. 이 정보를 이용하면, 텔파이가 특정 테이블이나 전체로서의 데이터베이스와 상호 작용하는 방법을 변경할 수 있다. 다음은 파라독스와 디베이스에 대한 테이블 등록 정보와 그 기능을 설명한 것이다. 다른 XBase 언어도 디베이스에 대한 기능과 유사한 등록 정보를 이용할 것이다.

### 1. Validity Checks

테이블의 각 필드에 대한 최소 및 최대값과 같은 데이터 입력의 제한을 할 수 있다. 디폴트 (Default Value) 필드는 무시하기 쉽지만, 올바른 데이터 입력 방법을 보여준다. 그리고, Picture 필드에는 실제로 복잡한 데이터를 입력할 때 도움을 주는 것으로, 테이블에 입력하려는 데이터의 문자맵에 의해 문자를 정의할 수 있다.

Validity Checks 대화 상자의 아래 쪽에 보면 Assist 버튼이 있는데, 이 버튼을 클릭하면 다음과 같은 대화 상자가 나타난다.



여기서 알아야 할 것은 Picture 박스 작성 방법과 나타낼 문자 리스트의 길이이다. Picture 문자 세트에는 다음과 같은 것들이 있다.

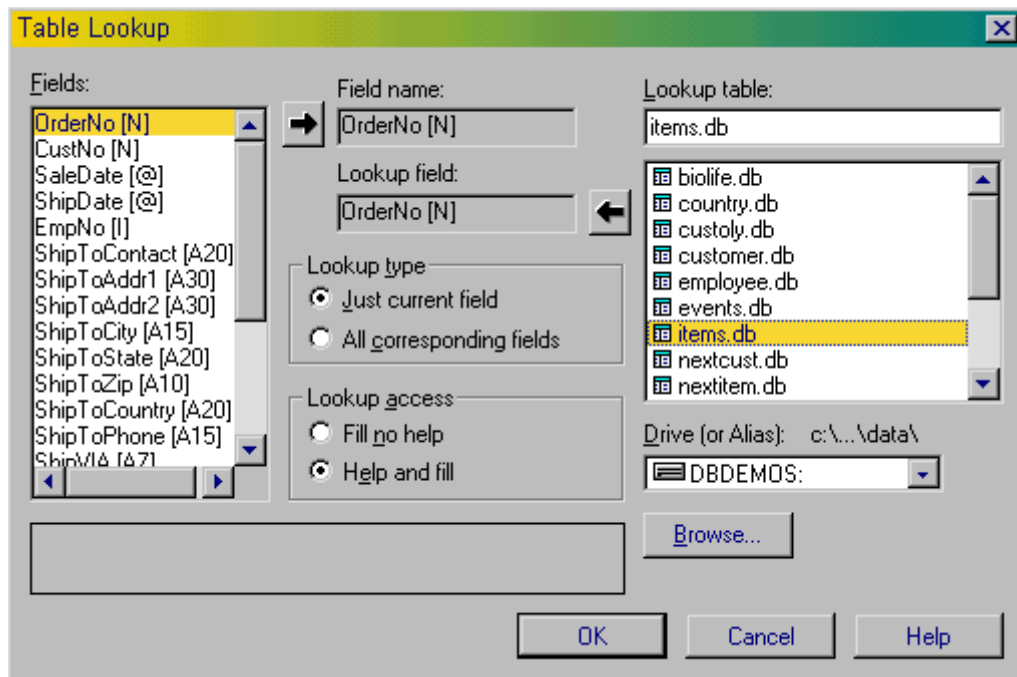
문자	설 명	문자	설 명
#	수치	?	모든 영문자 (대문자, 소문자)
&	모든 영문자 (대문자)	~	모든 영문자 (소문자)
@	모든 문자	!	모든 문자 (대문자로 변환)
;	다음 문자를 일반 문자로 처리	*	뒤에 반복될 회수를 숫자로 붙이고, 중괄호와 함께 문자유형을 붙임
[abc]	선택적인 문자를 나타냄	{a,b,c}	대괄호 사용 방법과 의미가 같다.

예를 들어 설명하면, 앞의 그림의 \*3{#}/\*3{#}-\*4{#}의 의미는 처음에 숫자 3 개를 치면 ‘/’가 표시되고 그 다음에 숫자 3 개를 입력하면 ‘-’가 입력되며, 그 이후에 숫자 4 개를 입력할 수 있다는 의미가 된다. 일단 picture 문장이 작성되면 Verify Syntax 버튼을 클릭하여 오류가 있는지 검사하고, Test Value 버튼을 클릭하여 실제 값이 어떻게 입력되는지 확인하도록 한다. 기존에 저장된 picture 를 가져올 수도 있고, 이렇게 작성한 picture 를 Add to List 버튼을 눌러서 저장할 수도 있다.

## 2. Table Lookup



자동적으로 조회 테이블을 지정할 수 있는 것으로, Define 버튼을 누르고 조회 테이블을 지정하면 된다. 조회 테이블은 보통 광범위한 데이터 입력 및 확인을 위해 사용하게 된다. 보통 분류를 할 필요가 있는 데이터가 있는 경우에 많이 사용된다. 조회 테이블을 연결하는 방법은 먼저 Table Lookup 을 선택하고, Define 버튼을 클릭하면 다음과 같은 대화 상자가 나타난다.



먼저 해야 할 것은 데이터 소스를 선택하는 것으로, 일단은 디렉토리나 앨리어스를 선택한다. 이렇게 하면 우측의 리스트 박스에 선택 가능한 테이블의 이름이 나타나는데, 적당한 테이블을 선택한다. 그리고 조회 필드를 선택할 때에는 Lookup 필드로 사용할 마스터 테이블의 필드를 Field Name 으로 선택하고, 해당되는 조회 테이블을 더블 클릭하면 조회 필드가 선택된다. 이때 조회 테이블에 조회가 가능한 필드의 데이터 형이 맞지 않으면 경고 메시지가 나온다. 앞의 그림은 Orders 테이블을 마스터로 Items 테이블을 조회 테이블로 설정한 것이다.

Lookup type 에서 디폴트 값인 Just Current Field 값을 이용하면, 선택한 데이터를 완전히 관리할 수 있게 된다. All Corresponding Fields 옵션은 조회 테이블의 필드와 일치하는 마스터 테이블의 모든 필드를 선택하게 되는데, 두 테이블의 구조를 확실히 알지 못하면 엉뚱한 결과가 나타날 수 있기 때문에 잘 사용되지 않는다.

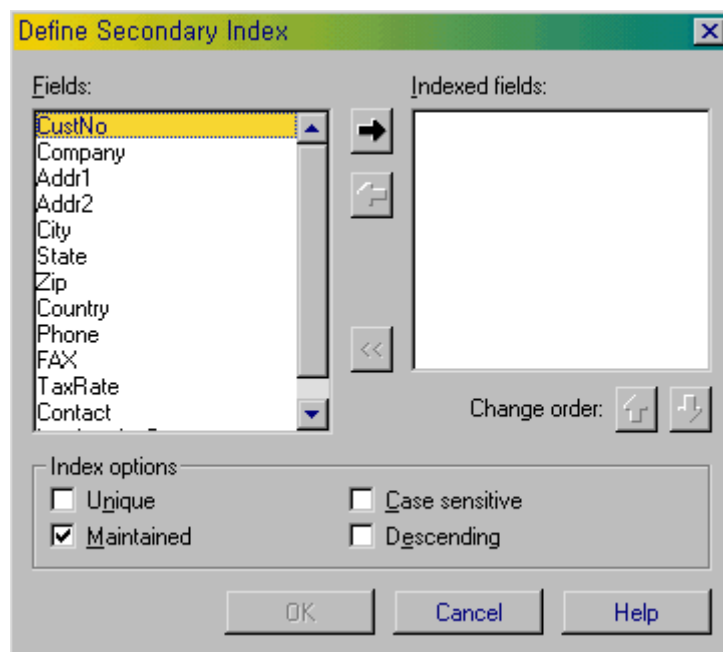
Lookup Access 옵션에서 디폴트 값인 Help and Fill 은 조회 테이블의 정보를 이용하여 선택한 마스터 테이블 필드를 자동으로 채우고, 사용자를 위한 도움말로 받아들인 입력 유형에 대한 메시지를 제공할 수 있다는 의미이다. Fill no Help 옵션은 마스터 테이블만 채우고, 사용자에게 메시지를 전혀 제공하지 않는다.

이제 OK 를 선택하면 조회 테이블이 설정되어, 조회 테이블이 연결된 필드로 Field Roaster 를 위치시키면 연결된 조회 테이블의 이름이 나타난다.

### 3. Secondary Indexes

파라독스는 primary index 와 secondary index 로 인덱스의 종류를 구별한다. Primary index 는 Field Roaster 의 Key 필드를 선택하면 설정된다. 그렇지만, 프로그램 사용자가 데이터를 여러 가지 방법으로 검색할 수 있도록 하려면, secondary index 를 정의해 주는 것이 좋다. 모든 테이블에는 기본적으로 primary index 를 가지고 있지만, secondary index 를 이용해서 조회 및 인쇄 루틴을 추가적으로 지원하는 것이 보통이다.

Secondary index 를 선택하고, Define 버튼을 클릭하면 데이터베이스 종류에 따라 다른 대화 상자가 나타나는데, 파라독스의 경우를 바탕으로 설명하겠다. 파라독스 테이블의 경우 다음과 같은 대화 상자가 나타난다.



Indexed fields 리스트 박스에는 secondary index 가 정의된 필드가 나타난다. 인덱스를 정의하는 방법은 좌측의 리스트 박스에서 인덱스를 설정할 필드를 선택한 후, 우측 화살표 버튼을 누르면 된다.

Index options 박스에서 인덱스에 다양한 효과를 줄 수 있는데, Unique 옵션은 사용자가 이 필드에 중복된 값을 가질 수 없을 때 사용한다. Case sensitive 는 영문자의 경우 대소문자를 가리지 않는 의미이며, Maintained 옵션은 인덱스를 자동으로 관리할 것인지 여부를 결정하는 것이다. 보통 디폴트로 체크되어 있다. 마지막으로, 인덱스는 기본적으로 오름차순으로 정렬되도록 설정되어 있는데, Descending 을 선택하면 내림차순으로 정렬한다.

일단 인덱스를 정의하고 나면, 대화 상자를 이용하여 인덱스를 저장할 수 있으며, 저장할 이름을 지정하면 된다.

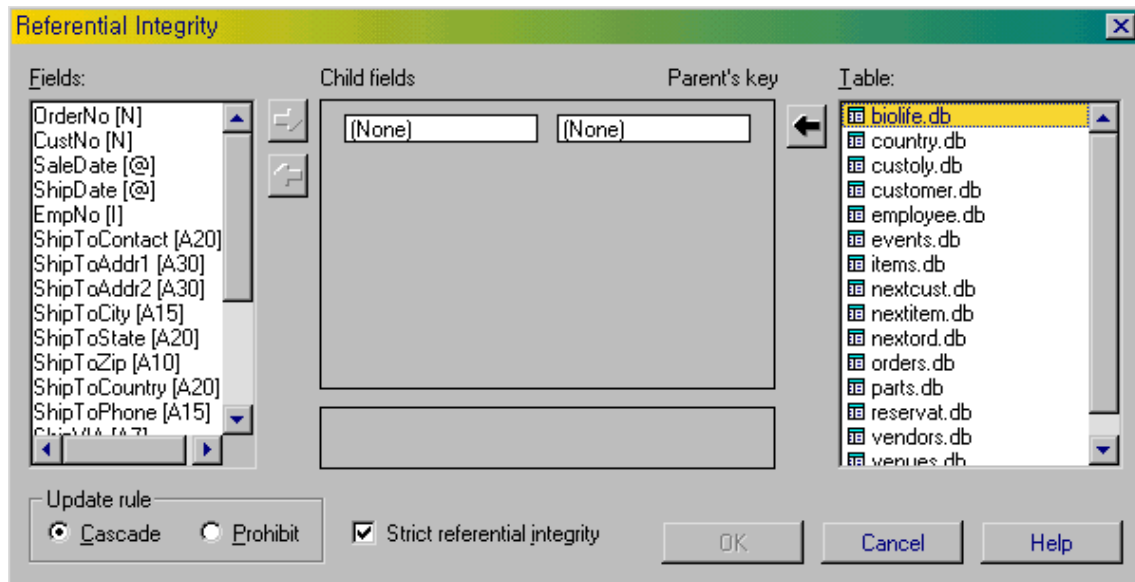
디베이스나 다른 데이터베이스의 경우 사용 방법이 다르지만, 여기에 대한 내용은 도움말을 참고하기 바란다.

#### 4. Referential Integrity

다수의 사용자에게 대한 다양한 관계를 설정하는 것으로, 참조 무결성의 방법을 정한다. 마스터 테이블과 디테일 테이블의 경우 상응하는 레코드가 마스터 테이블에 존재하지 않으면, 디테일 테이블이 orphan 테이블이 되는 문제를 해결하기 위한 방법을 지정한다.

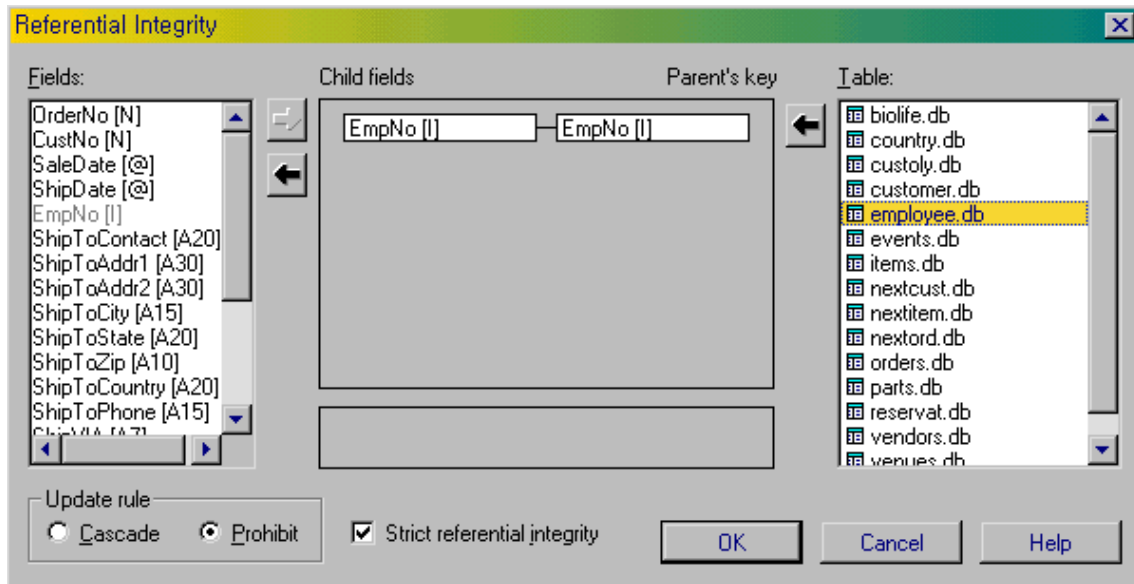
그러면, 마스터/디테일 테이블 관계를 설정하고 참조 무결성 옵션을 선택하는 방법을 알아보자. 예를 들어, orders.db 를 마스터로 하고, employee.db 를 디테일로 설정해서 정의해보자.

먼저 마스터 테이블을 열고 Referential Integrity 를 선택한 후, Define 버튼을 누르면 다음과 같은 대화 상자가 나타난다. 참고로, 이런 작업을 할 때에는 작업을 하려는 테이블을 같은 디렉토리에 위치시키고, File|Working Directory 메뉴에서 작업을 하고자 하는 디렉토리를 지정하는 것이 좋다.



여기에서, 두 테이블을 연결한 필드인 EmpNo 필드를 더블 클릭한다. 그리고, 연결한 디테일 테이블인 employee.db 를 더블 클릭하면 다음과 같이 키들이 연결된다.

그리고, OK 단추를 누르면 Save Referential Integrity As 대화 상자가 나타나는데 여기에 참조 무결성 규칙의 이름을 지정하면 된다.



이때 중요한 것은 Update rule 을 결정하는 것이다. Cascade 는 해당 디테일 레코드를 같이 삭제하는 것이며, Prohibit 은 마스터 레코드를 삭제할 수 없도록 한다.

## 5. Password Security

다수의 사용자가 SQL 서버를 통해 데이터베이스에 접속하여 데이터를 공유한다면 보안이 문제가 될 수 있다. 그러므로, 모든 SQL DBMS 제품들은 암호 화면을 제공한다. 파라독스에서도 암호 보안을 옵션으로 제공하는데, 이를 선택하고 Define 버튼을 클릭하면 Password security 대화 상자가 나타난다. 이 대화 상자에서 제공되는 두 필드에 암호를 입력하고 OK 버튼을 클릭하면 테이블에 암호가 지정된다.

## 6. Table Language

Modify 버튼을 클릭하면, 테이블의 데이터를 화면에 표시하는데 사용하는 문자 세트(코드 페이지)의 변경이 가능하다.

## 7. Dependent Tables

참조적 무결성의 현재 테이블에 의존하는 모든 테이블을 화면에 표시하는 기능으로, 등록 정보와 파일명 목록을 보고서 마스터/디테일 관계를 알 수 있다.

## 8. Indexes

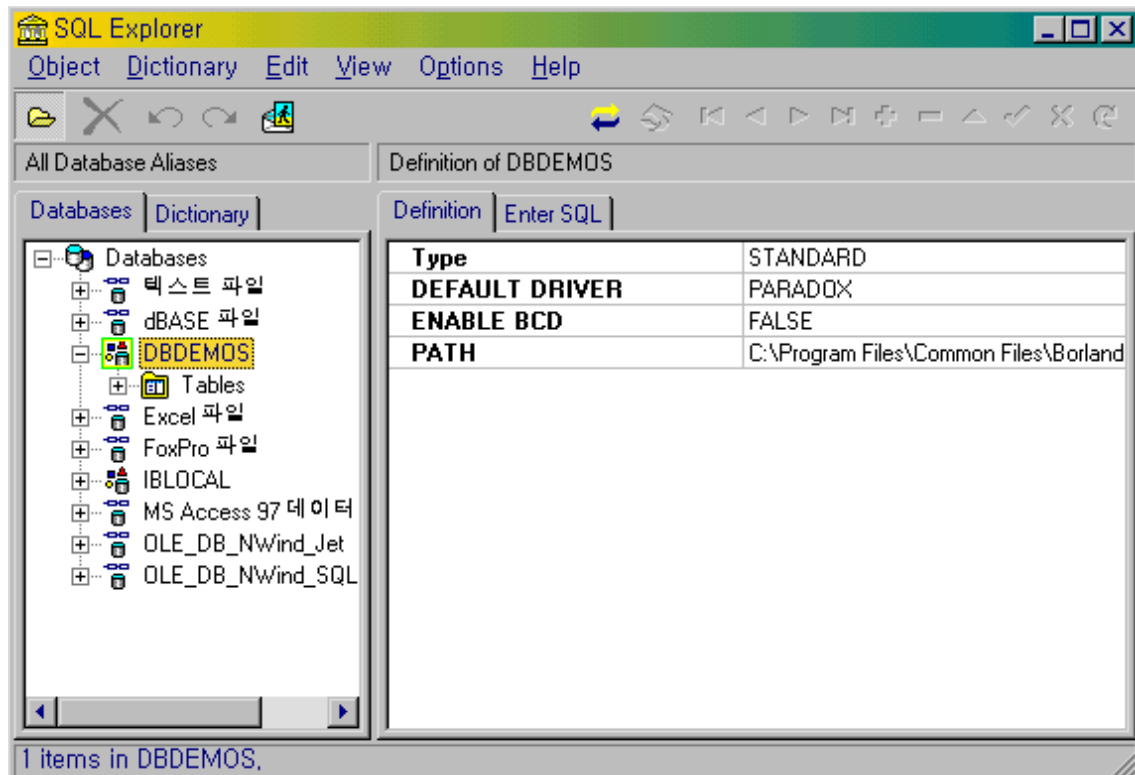
데이터베이스에서는 primary index 와 secondary index 를 구별하지 않는다. 여기에서는 필요한 색인 간의 전환이 가능하다.

## SQL 탐색기(SQL Explorer)와 BDE 관리자(BDE Administrator)

텔파이에서는 데이터베이스의 정보를 보고, 편집할 수 있도록 도와주는 유틸리티인 데이터베이스 탐색기(Database Explorer)가 제공된다. 이 유틸리티 프로그램의 이름은 C/S 버전인 경우에는 SQL 탐색기라고 하며, SQL 링크를 통해 SQL 데이터베이스에 직접 접근이 가능하다. BDE 관리자는 기본적으로 SQL 탐색기의 기능의 일부와 동일하기 때문에, 같이 설명하기로 한다. SQL 탐색기를 사용할 수 있으면, BDE 관리자는 쉽게 사용할 수 있다. 기본적인 SQL 탐색기의 기능을 설명하면 다음과 같다.

1. BDE 앨리어스 관리
2. 테이블, 뷰, 트리거, 저장 프로시저 등의 메타 데이터 객체 관리
3. 사용자와 서버의 보안 정보 관리

SQL 탐색기를 처음 시작하면 다음과 같은 화면을 볼 수 있다.



Databases pane 에는 가능한 앨리어스를 모두 보여주며, 앨리어스를 선택하고

Object|Open 메뉴를 선택하거나, Open 스피드 버튼을 클릭하면 데이터베이스에 연결되는데, 일단 연결되면 앞 그림과 같이 좌측 앨리어스의 아이콘이 녹색 박스로 둘러 싸인다.

SQL 탐색기는 크게 나누어 탐색 패널(browsing panel)과 정보 패널(information panel)의 2 부분으로 구성되어 있다. 좌측의 탐색 패널은 Databases, Dictionary 의 2 가지 pane 이 있으며 정보 패널에는 데이터베이스에 따라 여러가지 정보가 표시된다.

### ● 탐색 패널 (Browsing Panel)

탐색 패널에는 모든 사용가능한 BDE 앨리어스가 트리뷰의 형태로 표시된다. 탐색 패널에서는 새로운 앨리어스를 생성하거나, 데이터베이스의 로그인, 데이터베이스 검사 등을 할 수 있다.

#### 1. 새로운 앨리어스의 생성

새로운 앨리어스를 생성하려면, 탐색 패널에서 오른쪽 버튼을 클릭하고, 팝업 메뉴에서 New 메뉴를 선택하면 된다. 이 메뉴를 선택하면, 앨리어스에 사용할 드라이버 종류를 묻는 대화상자가 나타나며, 일단 드라이버 종류가 선택되면 접속 파라미터에 대한 디폴트 값이 정보 패널에 나타난다. 새로운 앨리어스를 저장하려면, 팝업 메뉴 또는 툴바에서 Apply 메뉴를 선택하면 된다.

#### 2. 데이터베이스에 로그인

데이터베이스를 탐색 패널에서 확장하면 데이터베이스에 로그인하게 되는데, 이때 사용자 ID 와 패스워드를 묻는다. 일단 데이터베이스에 접속되면, 탐색기는 데이터베이스 아이콘 주변에 녹색 박스로 둘러 싸인다. 접속을 종료하려면 앨리어스를 선택하고, 팝업 메뉴에서 Close 메뉴를 선택한다.

#### 3. 데이터베이스 검사

탐색기가 데이터베이스를 확장하면, 데이터베이스 종류에 따라 데이터베이스의 정보를 보여 주게 된다. 앞의 그림의 경우 DBDEMOS 데이터베이스에 대한 정보로 4 가지 정보를 보여 준다. 참고로 IBLocal 데이터베이스를 선택하면 Domains, Tables, Views, Procedures, Functions, Generators, Exceptions, Blob Filters 등의 여러가지 정보가 나타난다. 여기에서 구체적인 테이블 들을 탐색해 가면서 테이블의 필드, 인덱스, 참조 무결성 등에 대한 테이블의 자세한 정보와 직접 SQL 문장을 입력하여 실행하는 등의 작업을 수행할 수 있다.

## ● 정보 패널 (Information Panel)

정보 패널에는 탐색 패널에서 선택된 아이템의 물리적 특징을 보여주게 된다. 정보 패널은 페이지 컨트롤로 구성되어 있고, 보통 Definition, Text, Data, Enter SQL 이라는 4 개의 페이지를 보여주게 된다.

### 1. Definition 탭

Definition 탭에 표시되는 정보는 선택된 객체의 종류에 따라 다양하게 표현된다. BDE 앨리어스가 선택된 경우에는 그 앨리어스의 정보에 대한 BDE 환경설정 정보가 표시되며, 테이블의 필드가 선택된 경우에는 Definition 탭에 필드 구조에 대한 정보가 표시된다.

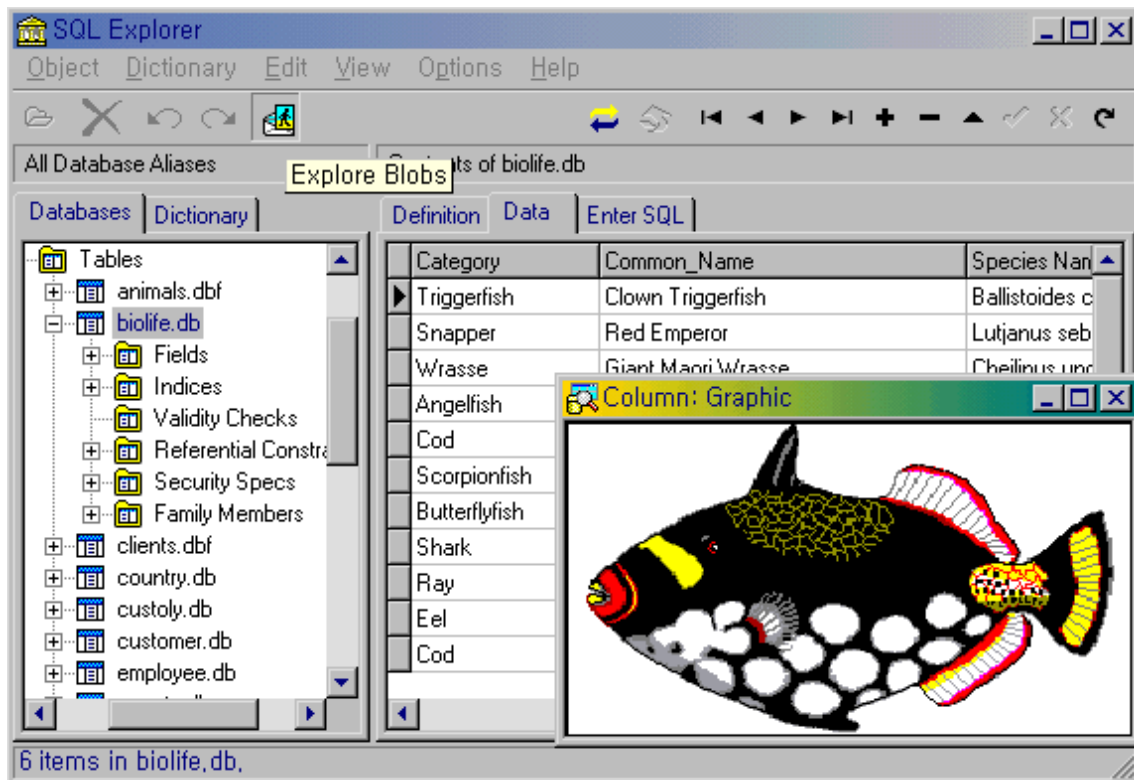
대부분의 경우에 Definition 탭에 표시되는 정보는 편집할 수 없다. 데이터베이스나 테이블, 필드 등을 restructure 하는 경우 Enter SQL 탭 또는 벤더에서 제공하는 소프트웨어를 이용할 수 있다. 데이터베이스에 접속되지 않은 경우에는 BDE 환경설정 정보를 직접 편집할 수 있다.

### 2. Text 탭

Text 탭은 선택된 객체에 대한 메타-데이터 정보를 얻기 위해 데이터베이스 쿼리를 이용할 때 사용된다. 이 정보는 SQL 스크립트로 변경되어 Text 탭에 표시된다. 이 기능은 테이블 등의 객체의 구조를 복제하여 생성할 때 매우 편리하게 사용될 수 있다. 개발자는 단순히 원본 객체를 선택하고, SQL 스크립트를 클립보드로 복사한 후, 이를 Enter SQL 탭에 붙여넣어서 사용하면 된다.

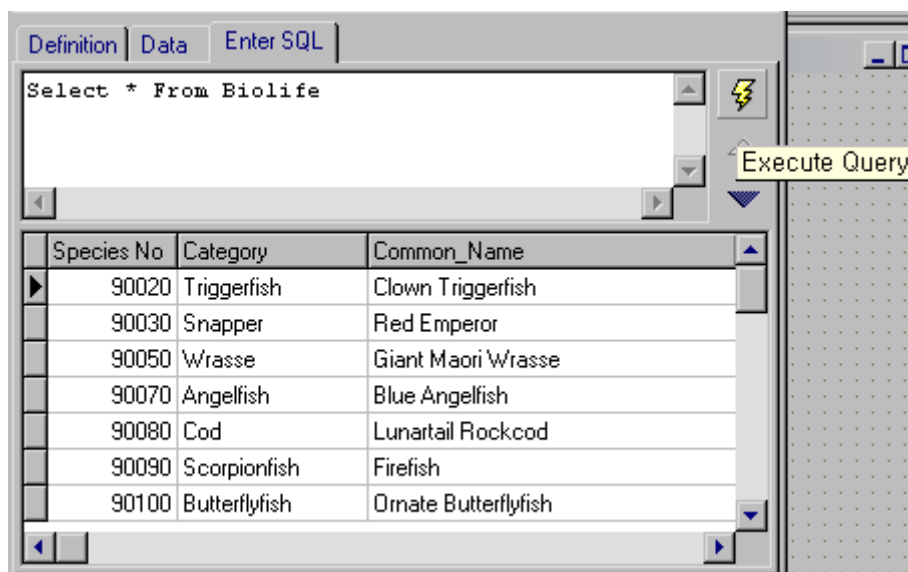
### 3. Data 탭

Data 탭에는 선택된 테이블의 데이터를 탐색하고 편집할 수 있게 한다. 데이터는 그리드의 형태로 표시되며, blob 데이터가 있을 경우에는 Explore Blob 스피드 버튼을 이용해서 그 내용을 볼 수 있다. 데이터베이스 데스크탑과 마찬가지로 실제 데이터의 편집이 가능하므로 편리하게 사용할 수 있다.



#### 4. Enter SQL 탭

Enter SQL 탭에서는 선택된 데이터베이스에 대한 SQL 문장을 실행할 수 있도록 해준다. 여기에 테이블을 생성하거나, 권한 부여, 트리거의 생성과 저장 프로시저를 생성하고 실행하는 등의 여러가지 작업을 할 수 있다.



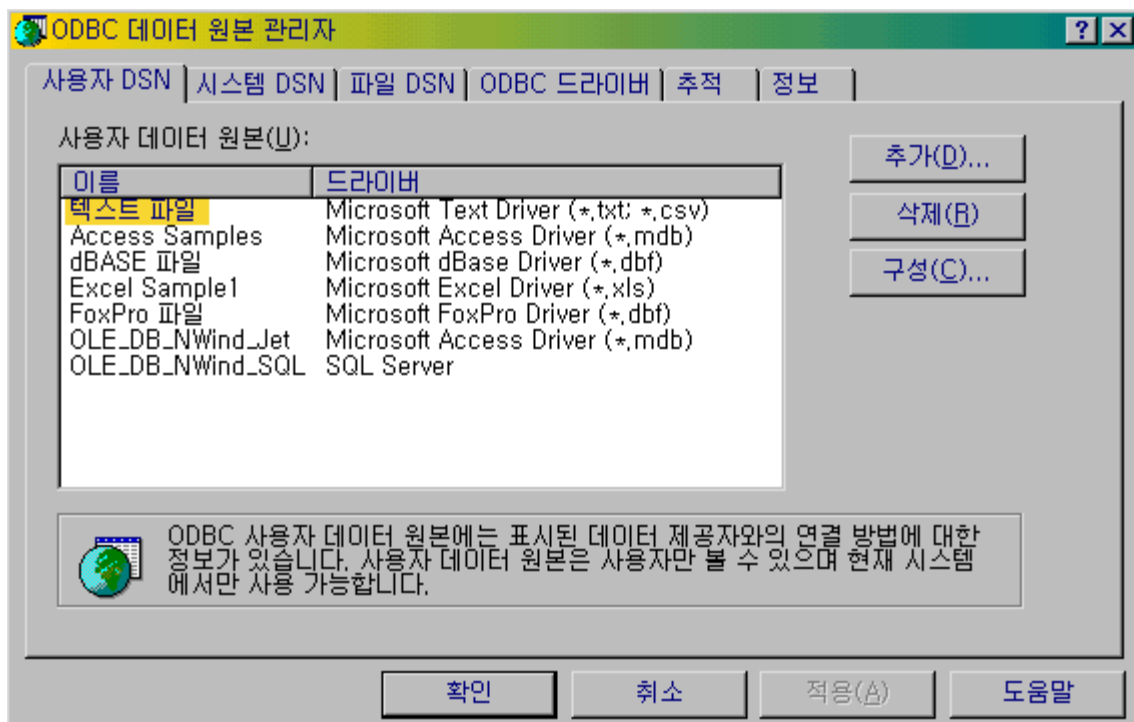


SQL 문장을 실행하려면, 앞의 그림과 같이 SQL 문장을 쳐 넣고 번개 모양의 버튼을 클릭하면 된다. Select 문 등을 이용하여 SQL 문장을 실행하면 결과 세트가 반환되는데, 이 내용은 쿼리 문의 바로 아래에 나타나므로 쿼리 문장을 테스트하는데 대단히 편리한 환경이다.

- ODBC 를 통해 DBMS 에 접근하기

텔파이가 처음 나왔을 때에는 BDE 에 의해 파라독스와 디베이스에 접근하는 것이 가능했지만, 다른 데이터베이스에 접근하기 위해서는 ODBC 를 사용해야 한다. 그런데, 문제는 생각보다 ODBC 드라이버를 설정하고 이를 이용하는 것이 간단하지 않다는 것이었다.

그런데, 텔파이 4 에서는 SQL 탐색기를 이용하여 간단하게 ODBC 드라이버와 BDE 앨리어스가 연동이 된다. ODBC 드라이버가 설정되면 그 내용이 즉시 SQL 탐색기에 앨리어스로 접근이 가능하기 때문에, 대단히 쉽게 ODBC 소스를 설정하고 여기에 접근할 수 있게 되었다. ODBC 를 설정하기 위해서는 SQL 탐색기의 Object|ODBC Administrator 메뉴를 선택한다. 이것에 의해 실행되는 ODBC 설정 유틸리티는, 다음과 같은 제어판의 32 비트 ODBC 설정 애플릿이다.



앞의 그림에는 필자가 나름대로 정의한 ODBC 원본이 포함되어 있다. 그러면, ODBC 원본을 추가하고 이를 텔파이에서 사용할 수 있도록 해보자. 먼저, '추가' 버튼을 눌러서 새로운 ODBC 원본을 추가하도록 한다. 이 버튼을 누르면 윈도우에 추가된 ODBC 드라이버의

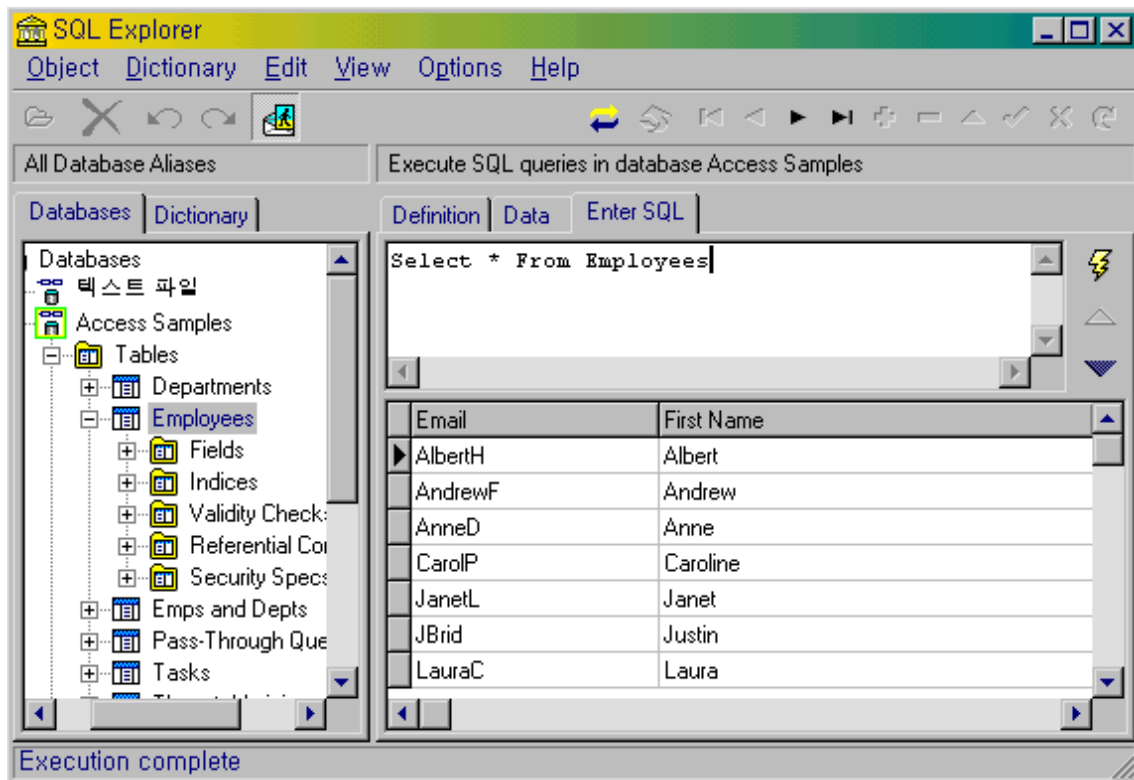
세트가 나열된다. 여기서, 액세스 97 의 ODBC 드라이버를 설정하도록 해보자. 드라이버로 Microsoft Access Driver 를 선택하고, 다음과 같이 이름과 설명을 설정한다.

그리고, ‘선택’ 버튼을 클릭하여 ‘.mdb’ 파일을 선택한다. 필자는 Microsoft Office\Office 디렉토리에 있는 EMPLOYEE.MDB 파일을 선택하였다. 그러면, 이제 확인 버튼을 누르면 설정이 끝난다.

텔과이를 종료하고, SQL 탐색기를 새로 시작하면 Access Samples 라는 새로운 엘리어스가 Databases pane 에 추가되어 있을 것이다. 이 객체를 선택하고 접근하려 하면 사용자 이름과 패스워드를 묻는데, 여기에 ‘Admin’을 사용자 이름에 입력하고 패스워드는 비운 채로 ‘확인’ 버튼을 누르면 Employee.mdb 파일에 접근할 수 있게 된다.

이와 같이 ODBC 를 설정하는 방법은 ODBC 드라이버의 종류가 어떤 것이냐에 따라 달라진다. 이를 위해서는 설치할 ODBC 드라이버의 사용법을 익혀 두어야 한다.

어쨌든, SQL 탐색기를 통해 ODBC 소스에 접근하게 되면 여기에 담겨 있는 테이블에 접근할 수 있으며 앞에서 DBDEMOS 엘리어스에 있는 여러가지 테이블에 접근하는 것과 마찬가지로 다음과 같이 데이터를 직접 탐색하고, SQL 문장을 입력할 수 있다.



## ● 데이터 사전 (Data Dictionary)

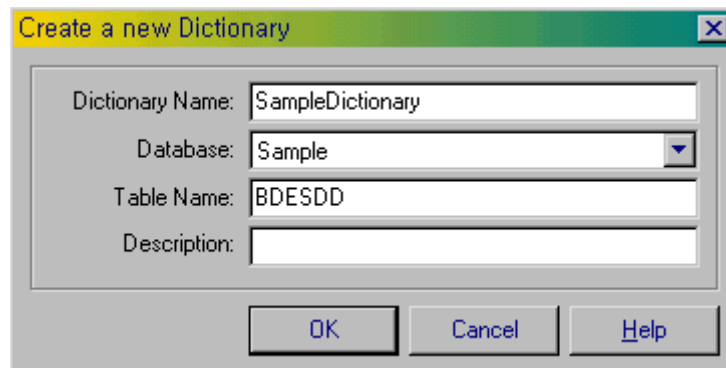
데이터 사전은 일반 언어 사전과 마찬가지로 데이터베이스에 대한 사전의 역할을 한다. 데이터 사전은 여러 데이터베이스를 정의하고, 사용된 용어의 의미를 알려 준다. 데이터베이스 사전에서 단어에 해당되는 것이 데이터베이스이며, 모든 저장 프로시저와 쿼리 뿐만 아니라 테이블 및 인덱스에 대한 정의를 표시한다.

데이터 사전을 작성하려면 우선 SQL 탐색기에서 사용할 데이터베이스를 지정해 주어야 한다. 그리고, 하드 디스크에 데이터 사전 디렉토리를 원하는 디렉토리를 지정하여 작성한다. 이 디렉토리는 이후 데이터에 대한 모든 정의가 저장되는 위치이다. 데이터 사전 폴더에 새로운 데이터베이스 항목을 추가해야 하는데, 표준(Standard) 드라이버는 디렉토리에 패스를 추가하는 것만큼이나 항목을 간단하게 해 준다.

### 1. 데이터 사전의 작성

먼저 Databases pane 에서 New 명령을 선택한 후, 드라이버 type 을 STANDARD 로 설정한다. 그리고, 사용하려는 디렉토리를 지정한다(c:\Wtemp\Wsample 등). 이렇게 지정한 데이터베이스 앨리어스의 이름을 Sample 이라고 지정한다. 이제 SQL 탐색기를 종료한 후, 다시 시작한다. 그리고, Dictionary pane 을 선택한 후 오른쪽 버튼을 클릭하면 나오는 명령 중에 New 를 선택한다. 대화 상자가 나타나면 다음과 같이 Sample 앨리어스를 선택하

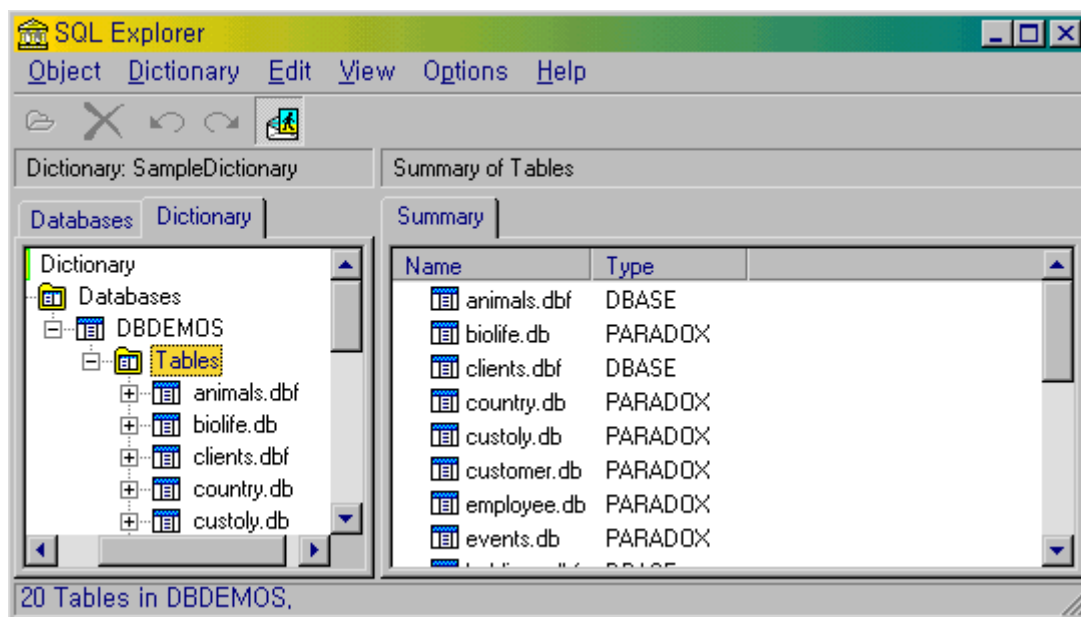
고, 사전의 이름을 SampleDictionary 로 설정한다.



이렇게 하면, 새로운 데이터 사전이 정의된다. 데이터베이스 사전을 작성하는데에는 시간이 많이 걸릴 수도 있다. 특히, 원격 서버의 SQL 데이터베이스에 저장할 경우에는 더욱 그렇다. 일단 데이터베이스 데스크탑에서 이 작업이 완료되면, 정보 패널에 사전 항목이 나타난다.

사전을 정의했으면 실제 데이터를 추가한다. 데이터베이스를 추가할 때에는 Dictionary|Import 명령을 이용한다. 여기서는 Import Database 를 선택하여 데이터 사전을 작성할 것이다. 이 명령을 선택하면 대화 상자가 나타나는데, 여기에서 데이터 사전을 만들 데이터베이스를 선택한다. 여기서는 DBDEMOS 를 선택하고 OK 버튼을 클릭하면, 데이터 사전을 만드는 작업이 시작된다.

데이터 사전이 다 만들어지면 다음과 같이 데이터베이스 목록이 나타난다.



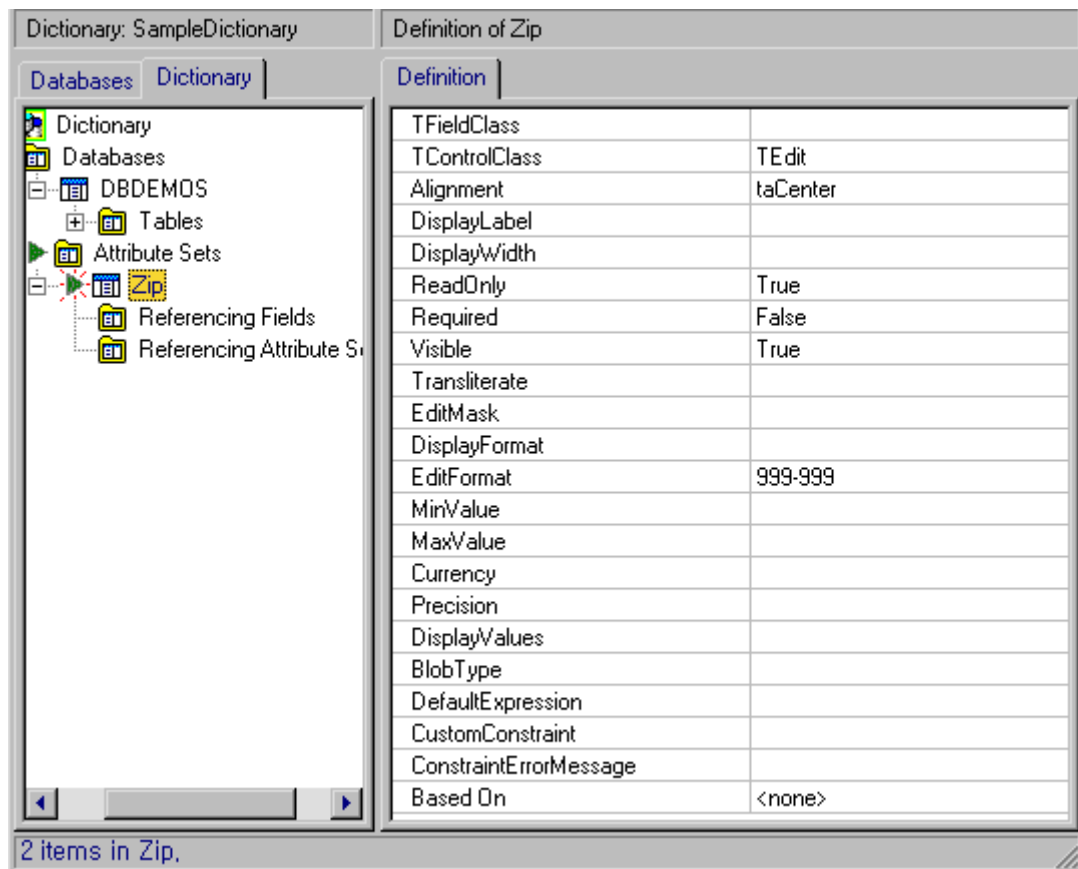
## 2. 데이터 속성 정의하기

데이터 사전을 사용하는 가장 큰 목적은 데이터베이스의 여러 필드에 대한 속성을 직접 관리하는 것이다. 데이터 사전의 Attribute Sets 섹션에는 데이터 사전에서 정의된 모든 속성이 포함되어 있다. 속성(Attribute)이란 데이터베이스 필드에 적용할 수 있는 데이터의 포맷, 최소-최대값, 정렬, 정밀도 등의 정보의 컬렉션이다.

이런 속성을 활용할 수 있는 것은, 여러 어플리케이션에 특정 필드를 같은 방식으로 정의한다고 할 때 그 필드에 대한 내용을 적당한 속성으로 설정하면 관리나, 재사용의 측면에서 바람직하다고 말할 수 있다.

예를 들어 우편 번호나 주소, 전화 번호와 같이 항상 같은 형식으로 사용할 수 있는 필드의 내용을 새로운 속성(Attribute)으로 등록하고, 이런 필드 형을 사용하는 데이터베이스가 있으면 직접 데이터 사전에서 속성 세트를 검색하여 지정하도록 하는 것이다.

다음은 우편 번호에 대한 새로운 속성을 등록하는 실제 예를 보여주는 그림이다.



새로운 속성을 생성하는 방법은 간단하다. Attribute Sets 객체를 선택하고, 오른쪽 버튼을 클릭한 후 New 명령을 선택한다. 새로운 Attribute 가 추가되었으면, 이름을 적당히 변경하고 원하는 형태로 각종 속성들을 설정하면 된다. 참고로 Alignment, ReadOnly, Required, Visible 프로퍼티와 TControlClass 를 잘 설정하면, 필드 에디터에서 폼으로 드

래그-드롭 했을 때 설정한 속성에 대한 컨트롤이 폼위에 자동으로 생성된다.

## 정 리 (Summary)

이번 장에서는 데이터베이스 어플리케이션을 작성하면서 유용하게 사용할 수 있는 각종 유틸리티 프로그램의 사용 방법에 대해서 알아보았다. 사용하기에 따라서는 매우 유용하게 사용하는 사람도 있을 수 있겠고, 어떤 사람은 거의 사용하지 않고 모든 것을 코드로 해결할 수도 있을 것이다.

그렇지만, 구슬이 서말이라도 꿰어야 보배라는 속담도 있지 않은가 ? 사용하라고 제공된 도구를 거들떠 보지 않는다면, 그거야 말로 낭비가 될 것이다.

이번 장의 내용을 다시 한번 살펴 보고, 앞으로 데이터베이스 어플리케이션을 사용할 때 적극적으로 사용하는 지혜를 기르기를 기원한다.

## 데이터베이스 어플리케이션과 SQL

이번 장에서는 델파이에서 지원하는 기본적인 컴포넌트를 이용하여 프로그래밍하는 방법과 SQL 문장을 사용하여 데이터를 조작하는 방법에 대하여 알아보겠다. 앞에서도 간단히 언급한 바 있지만 필자는 데이터 인식 컴포넌트를 그다지 좋아하지는 않는다.

필자가 아는 몇몇 프로그래머들은 기본적으로 델파이에서 지원하는 DBLabel 이나 DBEdit 와 같이 간단한 컨트롤을 사용하는 것조차도 싫어한다.

델파이로 DB 프로그래밍을 하는 방법에는 1, 2, 멀티-tier 환경의 DB 프로그래밍으로 나누어 볼 수 있다. 이들의 차이는 DB 를 어떤 방법으로 접근하느냐에 달려있다. 하지만, 기본적인 비즈니스 규칙을 적용하는 방법은 동일한데, 보통 다음의 5 가지를 고려하면 된다.

1. 자료를 데이터베이스에서 읽어오는 방법
2. 기본 데이터베이스 관련 유틸리티 사용방법
3. 자료 처리시의 에러처리 방법
4. 자료를 화면에 표시하는 방법
5. 사용자의 요구에 따라 프린터로 출력하는 방법

이 내용을 바탕으로 하여 이번 장을 진행하도록 한다. 이 중에서 데이터베이스 관련 유틸리티의 사용방법에 대해서는 앞 장에서 다룬 바 있으므로 이를 참고하기 바란다.

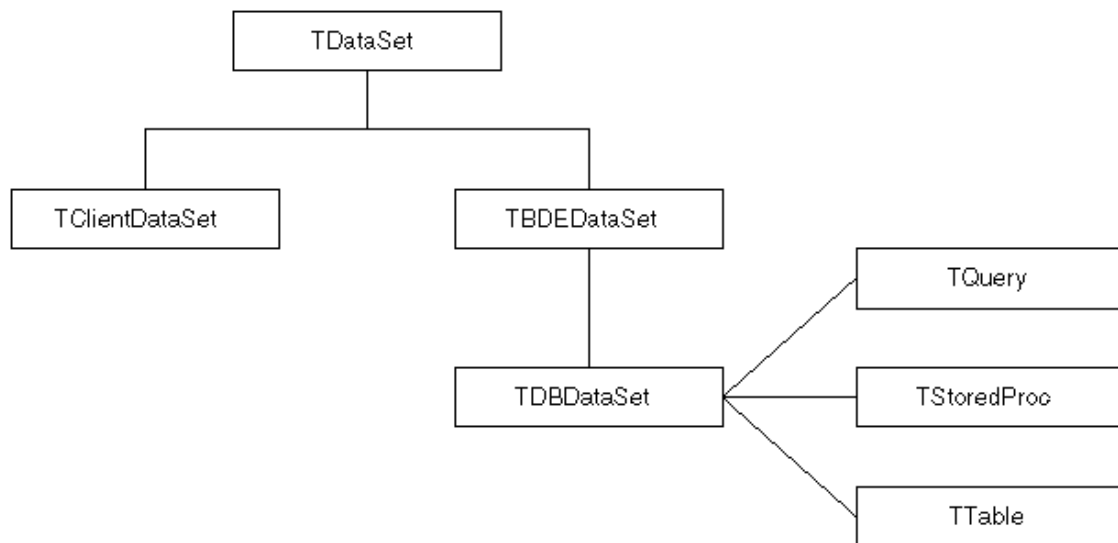
### 자료를 데이터베이스에서 읽어오는 방법

자료를 읽어오는 대상이 되는 것은 기본적으로 TDataSet 의 구조가 기본이 된다. 그러므로, TDataSet 의 내용을 파악할 필요가 있다.

#### ● TDataSet 컴포넌트

데이터 세트란 결과적으로 델파이에서 지원하는 관계형 데이터베이스의 테이블(열과 행으로 구분되는)을 지칭하는 것이다. 델파이의 Data Access 팔레트에서 다양한 데이터 세트 컴포넌트를 만날 수 있는데, 이러한 컴포넌트 들은 결국 관계형 데이터베이스의 테이블을 읽어오기 위한 것이며, 근본적인 차이는 없다.

가장 기본적인 데이터 세트의 계층도를 살펴보면 다음과 같다.



이 계층도를 살펴보면 TDataSet 클래스는 모든 테이블의 상위에 존재한다. 델파이에서 사용되는 모든 데이터 세트 관련 컴포넌트는 다음과 같은 기능을 지원한다.

구 분	프로퍼티/메소드
기본 기능	Active, Open, Close
검색 기능	First, Last, Next, Prior, MoveBy, Refresh, EOF, BOF, and, IsEmpty
편집 기능	Edit, Insert, InsertRecord, Append, AppendRecord, Delete, Post, Cancel, Midufied, State, ChechBrowseMode, SetFields
북마크 기능	BookmarkValid, CompareBookmarks, GotoBookmark, FreeBookmark, GetBookmark, Bookmark
데이터 컨트롤 지원 기능	ControlsDisabled, DlsableControls, EnableControls, ISLinkedTo
이벤트	BeforeOpen, AfterOpen, BeforeClose, AfterClose, BeforsInsert, AfterInsert, BeforeEdit, AfterEdit, BeforePost, AfterPost, BeforeCancel, AfterCancel, BeforeDelete, AfterDelete, BeforsScroll, AfterScroll, onCalcFiels, onDeleteError, onEditError, onNewRecord, onPostError
필드지원 기능	FieldByname, FieldByNumber, FieldDefs, FieldCount, Fields, FieldValue, DefaultFields, FindField, GetFieldList, GetFieldNames, UpdateRecord, ClearFields
하위레벨지원 기능	ActiveBuffer, CursorPosChanged, GetCurrenRecord, Translate, RecordSize

#### ● TDataSet 컴포넌트의 주요 기능



TDataSet 클래스의 주요한 기능에 대해서 그룹을 나누어 알아보도록 하자.

### 1. Active, Open, Close

실제로 `Active := True` 와 `Open` 문장은 동일한 동작을 수행한다. 이렇게 한 이유는 어플리케이션의 디자인 시와 런타임 환경에서 동일한 동작을 수행하기 위한 하나의 방편이다. 그에 비해, `TQuery` 의 형태인 경우에는 `Open` 문으로 수행하지 못하는 몇가지 작업이 있다. `TQuery` 에는 `ExecSQL` 이라는 메소드가 있는데, 이 메소드는 데이터를 조작하는 문장이 `Insert` 문이나 `Delete`, `Update` 문인 경우에 `ExecSQL` 을 사용한다. 이 경우 수행되는 내용을 `Active` 프로퍼티를 통하여 검사할 수 있다.

### 2. TDataSet 의 조작

이번에는 데이터를 검색하고 환경을 조절하는 방법에 대해서 알아보자.

`First`, `Next`, `Prior`, `MoveBy(수치)`는 레코드의 행을 이동하는 메소드이다. 그리고 `BOF` 와 `EOF` 는 레코드의 처음과 끝을 참조하기 위한 Boolean 형태의 프로퍼티이다.

해당 레코드의 필드를 조작하기 위해서는 다음과 같은 프로퍼티를 이용한다.

`테이블.Fields[n].AsString`

`테이블.FieldByName( '필드명' ).AsString;`

`테이블.FieldValues[ '필드명' ]`

`테이블필드명.Value` (이 경우는 필드 에디터에서 필요한 필드를 추가해야 사용할 수 있다)

이런 4 가지 표현방식은 모두 동일하다. 개발자는 어떠한 방식을 사용하여도 좋으나 가장 무난한 방법은 `FieldByName` 을 사용하는 것이다.

`Insert`, `Edit`, `Post`, `Delete`, `Cancel` 메소드 들은 데이터를 추가하거나 삭제하고 취소하는 메소드 들이다. 이들 메소드는 자료를 추가하고 자료를 변경하고 자료를 지우는데 사용되는데, 이 작업은 데이터 세트의 커서의 위치에서 이루어지는데 커서를 이동하는 방법은 위에서 설명한 레코드 행을 이동하는 방법(`Insert`, `Edit`, `Post`, `Delete`, `Cancel` 등)을 사용하면 된다.

### 3. 자료를 읽어오는 방법

이제 데이터 세트의 기본적인 내용을 살펴보았다. 그러면, `TDataSet` 에서 파생된 `TTable` 과 `TQuery`, `TStoredProc` 컴포넌트의 기능에 대해서 알아보자.

`TQuery` 와 `TStoredProc` 컴포넌트는 거의 동일한 기능을 수행하며, `TQuery` 와 `TTable` 의

차이점은 테이블을 보는 관점이 다르다는 것이다. TQuery 컴포넌트에서는 테이블을 바라보는 정의를 SQL 문장으로 기술하며 사용자가 원할 경우에 사용자가 원하는 시점을 사용하기 위해 SQL 문장의 다양한 기능을 이용하여 Join 이나 View 등을 통하여 2 차원적인 테이블을 그때그때 만들어 낼 때 많이 사용한다. TTable 의 경우에는 테이블의 형태가 고정되어 있는 경우가 많거나, 마스터/디테일 기능을 이용할 경우에 많이 사용된다.

델파이에서 데이터베이스를 관리할 때에는 앨리어스를 많이 사용하는데, 이 기능은 여러 RDBMS 서버에서 유래된 것으로, 데이터베이스라고 하는 데이터 공간을 만들어 고유의 명칭을 부여한 다음 해당 데이터공간에 원하는 테이블을 생성하는 방법으로 데이터를 관리하는 것이다. 이 방식은 하나의 목표로 사용되는 테이블 들의 구분과 물리적인 위치를 고정함으로써 데이터베이스의 성능을 향상하는 경우에 사용되었다. MS-SQL 서버인 경우에는 데이터베이스의 크기를 고정하고 단일의 파일을 생성하여 사용한다. 인터베이스의 경우에도 단일 파일로 만들어진 데이터베이스를 사용한다.

그에 비해, 파라독스나 디베이스 등의 경우 많은 테이블이 존재할 수 있고, 이러한 테이블을 단일 형태로 관리할 수 있는 방법으로 이러한 앨리어스를 사용하게 된다. 그에 비해, 액세스의 경우에는 하나의 앨리어스가 하나의 파일로 존재한다.

델파이에서는 이러한 데이터베이스를 관리하기 위한 방법으로 TDatabase 컴포넌트와 TSession 컴포넌트를 지원한다. 이 컴포넌트 들은 앨리어스를 관리하며 해당 데이터베이스에 접근하기 위해 로그인 기능도 부여한다. 일반적으로 TDatabase 컴포넌트를 이용하여 앨리어스에 접근하고 가상 데이터베이스의 이름을 부여한 다음, 이 가상 데이터베이스 컴포넌트의 이름을 통하여 지정된 데이터 세트 컴포넌트의 데이터베이스 테이블에 접근할 수 있다.

#### 4. 데이터의 검색

테이블은 방대한 자료를 가지고 있으며 기본적인 이동명령어인 First, Next, Prior, MoveBy 문장을 사용하여 자료를 이동하며 각각의 필드에 있는 값을 읽어 낼 수 있다. 하지만 테이블에 많은 데이터가 쌓이므로 이러한 이동 명령어만 이용하여 차례로 비교하는 방법을 사용하면 비효율적이 아닐 수 없다.

이럴 때에는 각각의 테이블에 인덱스를 사용하는데, 다음의 방법은 인덱스가 존재한다면 델파이 내에서 데이터를 검색하는 가장 빠른 방법이다.

```
Table1.SetRangerStart;  
Table1.FieldName('필드명').AsString := '검색값';  
Table1.SetRengeEnd;  
Table1.FieldName('필드명').AsStrign := '검색값';  
Table1.ApplyRange;
```

이 문장들에 의해 필드명의 시작 값과 끝 값을 지정하면, 해당 테이블의 인덱스를 참조하여 빠르게 원하는 레코드들을 배열한다. 인덱스가 존재한다면 가장 빠른 방법이다. 그렇지만 이 방법은 TTable에서만 적용되는 방법이다.

다른 방법으로는 Filter 를 사용할 수 있다. 이 방법은 TTable 과 TQuery 에서 모두 사용이 가능하다.

```
Table1.Filtered := True;
```

```
Table1.Filter := '(필드명 >= 값 ) and ( 필드명 <= 값 )';
```

이렇게 필터기능을 사용한 다음 FindFirst, FindLast, FindNext, FindPrior 을 사용하면 원하는 레코드를 찾을 수 있다.

또한, 인덱스가 지원되는 TTable 이나 TClientDataSet 와 같은 컴포넌트에서는 다음과 같이 EditKey, FindKey, FindNearest, GotoKey, GotoNearest, SetKey 메소드를 사용하여 레코드를 검색할 수도 있다.

```
Table1.SetKey;
```

```
Table1.FieldName('필드명').AsString := '검색값';
```

```
if not Table1.GotoKey then ShowMessage('찾지못함');
```

Table1.GotoKey 메소드를 사용하면 해당 값을 가진 레코드로 이동하며, 값이 존재하지 않는다면 False 가 반환된다. 참고로, GotoNearest 메소드를 사용하면 해당 값과 가장 근사치인 값의 레코드로 이동한다. FindKey 와 FindNearest 의 경우는 Setkey 와 GotoKey, Nearest 를 합쳐놓은 것과 동일하다.

TQuery 컴포넌트를 사용할 경우에는 생각외로 간단하다. 해당 SQL 문장을 다음과 같이 기술하면 된다.

```
with Query1 do
```

```
begin
```

```
    Active := False;
```

```
    Sql.Clear;
```

```
    Sql.Add( 'select * from 테이블명 where 필드명 조건 값' );
```

```
    Active := True 또는 Open 또는 ExecSQL;
```

```
end;
```

TQuery 나 기타 SQL 을 사용하는 컴포넌트를 사용할 경우 Active 나 Open/Close 와

ExecSQL 의 차이점은 단순히 데이터를 표시하는 것인지, 아니면 데이터를 조작한 후에 데이터를 리턴하느냐에 달렸다. SQL 문장 중에 데이터를 조작하는 INSERT, DELETE, UPDATE 문장의 경우 ExecSQL 을 사용하여야 한다. 그리고 SELECT 문장의 경우 원하는 데이터를 리턴 받아야 하므로 일반적인 TTable 과 동일하게 Active, Open/Close 를 사용한다.

## 5. Locate 메소드의 사용

Locate 는 커서를 찾는 범주에 맞는 첫번째 행에 위치시킨다. 간단하게 찾을 컬럼의 이름과 비교할 필드 값, 옵션을 넘겨주면 된다. 예를 들어 다음의 코드는 CustTable 의 Company 컬럼의 값이 "Professional Divers, Ltd."인 첫번째 행에 커서를 위치시킨다.

```
var
    LocateSuccess: Boolean;
    SearchOptions: TLocateOptions;
begin
    SearchOptions := [loPartialKey];
    LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
        SearchOptions);
end;
```

Locate 메소드가 해당되는 레코드를 찾으면 그 첫번째 레코드가 현재 레코드가 되며, 리턴값으로 True 가 넘어온다. 해당되는 레코드가 없으면 현재 레코드는 바뀌지 않으며 리턴값으로 False 가 넘어온다.

Locate 메소드는 여러 개의 컬럼과 여러 개의 값들을 비교할 때 유용하다. 검색 값이 가변형(variant)이기 때문에 여러가지 데이터 형을 이용할 수 있다. 여러 개의 컬럼을 지정해서 검색하려면 각각의 아이템들을 세미콜론으로 분리한다.

검색 값이 가변형이기 때문에 여러 개의 값을 넘길 때에는 가변형 배열을 이용하거나, VarArrayOf 함수를 이용해서 가변형 배열을 만들어야 한다. 다음 문장은 여러 컬럼에 여러 값을 이용해 검색을 하는 예이다.

```
with CustTable do
    Locate('Company:Contact:Phone', VarArrayOf(['Sight Diver', 'P']), loPartialKey);
```

## 6. Lookup 메소드의 사용

Lookup 메소드는 검색 조건에 맞는 첫번째 행을 찾아서 데이터 세트와 관련된 lookup 필드와 calculated 필드의 재계산까지 실행하고, 하나 이상의 필드의 값을 가변형으로 돌려준다. Lookup 메소드는 Locate 메소드와 달리 커서를 움직이지 않고, 값만을 돌려준다. 검색할 컬럼과 필드 값을 넘겨주면 되는데, 예를 들어 다음의 코드는 CustTable 의 Company 컬럼의 값이 "Professional Divers, Ltd."인 회사 이름, 담당자, 전화번호를 돌려준다.

```
var
    LookupResults: Variant;
begin
    with CustTable do
        LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company:
            Contact; Phone');
    end;
```

Lookup 메소드는 지정된 필드에서 조건에 맞는 첫번째 레코드의 값을 돌려준다. 하나 이상의 값을 요구할 때에는 가변형 배열(variant array)로 돌려준다. 조건에 맞는 레코드가 없으면 Null 값을 돌려준다.

Lookup 메소드도 Locate 메소드와 마찬가지로 여러 개의 컬럼과 여러 개의 값들을 비교할 수 있다. 다음 문장은 여러 컬럼에 여러 값을 이용해 검색을 하는 예이다.

```
var
    LookupResults: Variant;
begin
    with CustTable do
        LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver',
            'Christiansted']), 'Company; Addr1; Addr2; State; Zip');
    end;
```

## ● 레코드의 표시와 복귀

특정 레코드를 표시하면 원할 때에 그 레코드로 바로 돌아갈 수 있게 된다. 이런 기능을 지원하기 위해 TDataSet 와 그 자손들은 북마크를 지원한다. 북마크 기능은 Bookmark 프로퍼티와 5 개의 메소드로 구현된다.

Bookmark 프로퍼티는 어플리케이션에 있는 북마크 중 현재의 북마크를 가리킨다. 북마크를 하나 추가할 때마다 그것이 현재의 북마크가 된다.

TDataSet 은 가상 북마크 메소드를 구현한다. TBDEDataSet 에서 북마크 메소드를 다시 구현했는데, 이들은 다음과 같다.

1. BookmarkValid: 지정된 북마크가 쓰이고 있는지 알아본다.
2. CompareBookmarks: 2 개의 북마크가 같은 것인지 비교한다.
3. GetBookmark: 현재의 위치를 북마크에 할당한다.
4. GotoBookmark: 이전에 GetBookmark 에 의해 생성된 북마크로 돌아간다.
5. FreeBookmark: 이전에 GetBookmark 에 의해 할당된 북마크를 해제한다.

북마크를 만들려면 먼저 일종의 포인터 변수인 TBookmark 형의 변수를 선언하고, GetBookmark 를 호출하여 데이터 세트의 특정 위치를 값으로 할당하면 된다.

GotoBookmark 를 호출하여 특정 레코드로 이동하기 전에 BookmarkValid 를 호출하여 북마크가 레코드를 가리키고 있는지 확인할 수 있다. 북마크가 레코드를 가리키고 있으면 True 가 돌아온다.

다음은 북마크의 사용 예이다.

```
procedure DoSomething (const Tbl: TTable)
var
    Bookmark: TBookmark;
begin
    Bookmark := Tbl.GetBookmark; {변수에 메모리와 값을 할당}
    Tbl.DisableControls;
    try
        Tbl.First;
        while not Tbl.EOF do
            begin
                {여기에 작업할 코드를 쓴다.}
                ...
                Tbl.Next;
            end;
        finally
            Tbl.GotoBookmark(Bookmark);
            Tbl.EnableControls;
            Tbl.FreeBookmark(Bookmark); {북마크에 할당된 메모리 해제}
        end;
    end;
end;
```

- SQL 에 대하여 ...

세부적인 SQL 문장을 사용하기 전에 델파이에서 지원하는 인터베이스를 살펴보자, 델파이 C/S 버전에서는 C/S 프로그램을 제작하기 편하도록 테스트용 인터베이스 서버가 같이 지원된다. 인터베이스는 다른 RDB 와는 다르게 리모트나 로컬의 DB 를 접근할 경우에 앨리어스에서 해당 데이터베이스를 지정하기 위해서 컴퓨터이름과 해당 디렉토리를 모두 기술하여 주어야 한다. 이것이 불편하게 느껴질 수도 있지만, 물리적인 데이터베이스를 관리하기가 편하므로 유닉스나 윈도우 NT 의 경우 하드가 모자라서 확장할 경우에 데이터베이스를 간단하게 이동을 한 뒤에 해당 BDE 세팅에서 위치만 변경해주면 변경이 쉽게 된다는 장점도 있다.

로컬 인터베이스의 기본적인 User Name 은 SYSDBA, Password 는 masterkey 이다.

인터베이스 그룹의 인터베이스 서버 관리자(Server Manager) 프로그램은 인터베이스의 전반적인 데이터베이스를 다루는 DBA 의 입장에서 인터베이스를 다룰 수 있도록 해준다.

File/Server Login 작업에서는 리모트와 연결 데이터베이스에 접근하는데, 여기에서 앞서 설명한 SYSDBA/masterkey 를 이용한다. 그리고, Tasks/User Security 메뉴에서 새로운 사용자를 추가하는데, 이렇게 추가된 사용자명을 사용하여 프로그램에서 인터베이스에 접근하게 된다.

윈도우 ISQL(Windows ISQL)은 콘솔 SQL(Console SQL)을 지원하는 프로그램으로 데이터베이스나 테이블을 추가/수정/삭제할 수 있는 SQL 문장으로 인터베이스를 조작할 수 있도록 해준다. File/Create Database 메뉴를 선택하면 원하는 위치에 데이터베이스를 생성할 수 있으며, File/Drop Database 메뉴를 선택하면 데이터베이스를 삭제할 수 있다.

참고로, 인터베이스는 기본적으로 gdb 라는 확장자명을 사용하며, 모든 SQL 문장을 이용한 작업을 수행하려면 Commit 명령을 사용해야 한다.

기본적인 SQL 문장의 문법은 다음과 같다.

## 1. 테이블 생성

create table 테이블명(필드명 속성 형태 키형태, ...)으로 기술하면 되는데 다음의 예를 살펴보자.

```
create table data (code char(5) not null primary key, name varchar(20), age integer)
```

이 코드는 data 라는 테이블을 생성하는데 기본키로는 code 라는 char(5)형태의 널(null)값이 존재하지 않는 필드를 사용하고, 그 밖에 name 이라는 varchar 20 자리, age 라는 정수형 필드를 가지는 테이블을 만들려는 SQL 문장이다. 이런 작업은 SQL 문장을 이용하여

수행할 수도 있고, 데이터베이스 데스크탑을 이용할 수도 있다.

## 2. 레코드 추가

insert into 테이블명 (필드값) values (넣는 값)

## 3. 레코드 수정

update from 테이블명 set 필드명 = 값 where 조건

## 4. 레코드 삭제

Delete from 테이블명 where 조건

해당 테이블에 해당 조건에 맞는 레코드를 삭제한다.

## 5. 레코드 검색

Select 선택필드 into 변수명 또는 레코드명 from 테이블 where 조건

선택필드에서 \* 가 기술되면 전체 필드를 지칭하며 필요한 필드명을 나열하면 해당 필드로 구성된 테이블 뷰를 만들게 된다. 이때에 필드의 개수가 적을수록 해당 SQL 문장의 동작 속도는 빨라지게 된다.

참고로 TQuery 컴포넌트에는 Prepare 라는 메소드가 존재하는데, 이 메소드는 SQL 문장을 수행하기 전에 해당 문장을 최적으로 수행할 수 있는 환경을 구성한다. 이 문장은 꼭 사용하는 것이 좋다.

테이블 절에는 여러 개의 테이블을 기술할 수 있는데, 다음과 같이 사용할 수 있다.

Select 테이블명 A.필드명, 테이블명 B.필드명 from 테이블명 A, 테이블명 B

이렇게 하면, 두개 이상의 파일을 Join 하여 하나의 뷰로 볼 수 있도록 하여 준다.

## 6. Like 연산자의 활용

앞에서 언급한 SQL 문장의 조건절에는 다음의 Like 연산자를 사용할 수 있다. Like 연산자는 해당 필드에 속해있는 값들 중에 비슷한 값을 찾아내는 연산자로 도스 명령어를 사용할



때의 \*(와일드카드)와 비슷하다.

SQL 문장 내에 %를 사용하게되면 Like 연산자가 동작하게 된다.

Select \* from 테이블 where 필드 like '신%';

이 문장은 해당 필드에 속해 있는 값들 중에 '신'으로 시작되는 데이터를 찾을 것이다.

## 7. Order By 절의 활용

OrderBy 절은 레코드를 정렬하는 경우에 사용한다.

Select \* from 테이블 order by 필드명

오름차순과 내림차순은 Asce 와 Desc 를 해당 필드명 뒤에 붙여주면 해당 형식으로 정렬된다.

## 8. Group By 절의 활용

선택된 필드를 그룹단위로 묶어서 중복된 것만 보여준다.

Select 필드값, sum( 필드값 ) “합계”, avg( 필드값 ) “평균” from 테이블명 group by 테이블명

이렇게 하면 해당 테이블명이 중복된 값의 합계와 평균을 출력한다.

SQL 문장에 대해서 자세히 알기 위해서는 별도로 RDB 와 SQL 에 대해서 다루고 있는 서적을 한권 정도 참고하기를 권하고 싶다. SQL 문장은 기본적으로 표준화가 되어 있기 때문에, 이를 익혀두면 많은 부분에서 도움이 될 것이다. 그 밖에 몇 가지 유용한 SQL 문장에 대해서는 다음 장에서도 소개할 것이다.

### ● 저장 프로시저(Stored Procedure)의 사용방법

저장 프로시저는 일련의 SQL 문장으로 이루어진 서브루틴과 동일하다. 2-tier 이상의 프로그램을 제작하는 경우에는 많은 테이블을 Open 하여 반복적인 작업을 수행하는 경우가 빈번한데, 해당 테이블마다 SQL 문장을 통하여 Open 한 다음 클라이언트에서 작업을 하게 되면 비효율적인 네트워크 트래픽을 비롯하여 여러가지 자원의 낭비가 발생하게 된다.

이럴 때에는 저장 프로시저를 사용하여 해당 문장을 기술할 수가 있다. 그 밖에도 고정된 SQL 문장으로 클라이언트의 요구를 파라미터로 받아들여서 일반적인 서브루틴처럼 작동하는 모듈을 구성할 경우에도 이와 같은 저장 프로시저를 사용할 수 있습니다.

다음은 저장 프로시저를 만드는 예제 코드이다.

```
Create Procedure Insert_data ( PCode varchar(4), pname varchar(20), pag smallint )
as
begin
    begin
        insert into data values ( :pcode, :pname, :page );
    end
    suspend;
end
```

이런 문장을 수행하면 해당 데이터베이스에 Insert\_data 라는 저장 프로시저가 생성된다. 이제 델파이에서 이를 사용하려면 TStoredProc 컴포넌트를 사용하여 호출하면 된다. 사용하는 방법은 다음과 같다.

```
with StoredProc1 do
begin
    StoredProcName := 'insert_data';
    ParamByName( 'pcode' ).AsString := '값';
    ParamByName( 'pname' ).AsString := '값';
    ParamByName( 'page' ).AsInteger := 값;
    ExecProc;
end;
```

## ● TDatabase 컴포넌트의 활용

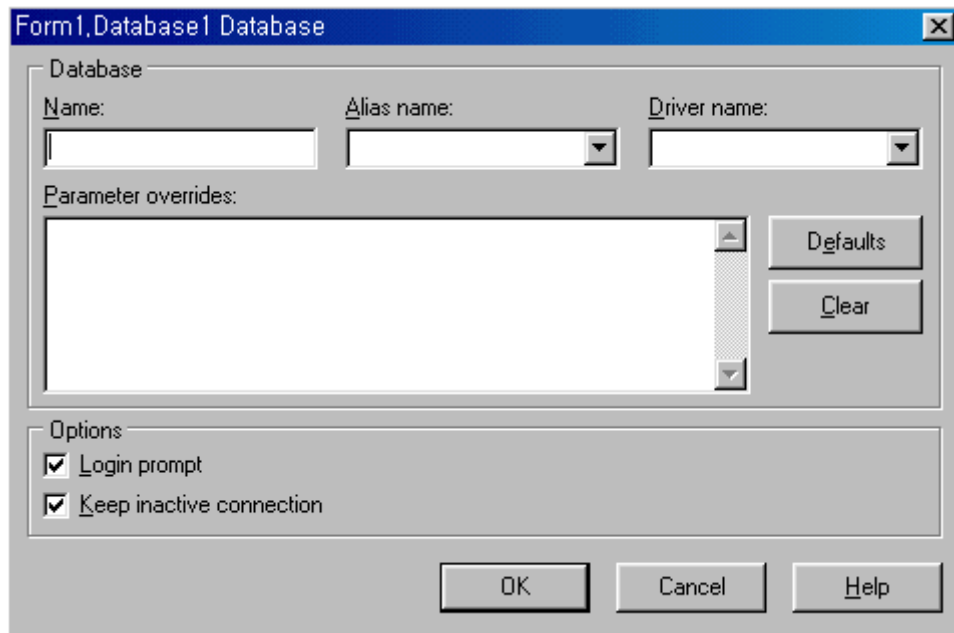
12 장에서도 이 컴포넌트에 대해서는 언급한 바 있지만, 많이 사용되는 컴포넌트인 만큼 이번 장에서도 사용하는 방법에 대해서 간단하게 알아보도록 하자.

### 1. 동적인 엘리어스의 생성

TDatabase 컴포넌트를 사용하면 존재하는 엘리어스를 사용하여 접근하거나, 또는 실행환경에서 하나씩 BDE 에 엘리어스를 생성시켜주는 불편을 사용하지 않고 수행 중에만 존재하는

앨리어스를 생성하여 사용할 수 있게 해준다.

다음 그림은 TDatabase 컴포넌트를 선택하고, 팝업 메뉴에서 Database Editor ... 이라는 메뉴를 선택하면 나타나는 대화상자이다.



여기에서 Parameter overrides 부분에는 해당 데이터베이스에 접근하기 위한 다양한 값들을 설정할 수 있다. Defaults 버튼을 클릭하면 기본적인 정보가 나타난다. Login Prompt 체크 박스를 설정하면, 사용자에게 UserName 과 Password 를 물어보는 대화 상자를 생략하게 할 수 있다. 또한, Keep inactive connection 체크 박스를 설정하면 데이터베이스를 열 때마다 사용자 이름과 암호의 입력작업을 중복할 필요가 없게 해 준다.

## 2. 트랜잭션의 관리

TDatabase 컴포넌트를 사용하는 가장 커다란 잇점이 바로 이것이다. 보통 C/S 프로그램을 제작하는 경우에 초보자들이 가장 어렵게 생각하는 부분이 바로 트랜잭션의 관리이다. 트랜잭션이란 SQL 문장으로 기술된 논리적인 하나의 작업 단위를 뜻하는데, 원격 DB 를 사용하는 경우 중간의 네트워크 때문에 자료의 비정상적인 전송이나 오류가 발생하는 경우가 빈번하다. 특히, 테이블이 하나가 아닌, 2~3 개 이상의 테이블의 값을 수정하거나 추가하는 경우에는 데이터의 무결성에 치명적인 문제가 발생할 수 있기 때문에, 이런 트랜잭션이 필수적이라고 할 수 있다.

3 개의 테이블을 수정하는데, 2 개의 테이블만 수정되고 1 개의 테이블의 값이 수정되지 않는다면 이 데이터는 나중에 치명적인 문제를 발생할 수 있다. 그렇기 때문에, 이럴 때에는 이런 수정작업을 하나로 묶어서 사용해야 하는데, 이를 트랜잭션이라 한다.

이렇게 트랜잭션을 적절하게 사용하면 데이터의 하드웨어적인 문제의 일관성(Consistency)과 다중 사용자 접근에 대한 무결성(Integrity)을 보장해 줍니다.

### 3. 트랜잭션의 관리 방법

트랜잭션의 관리 방법에는 크게 나누어 다음과 같은 2 가지가 있다.

#### - 기본제어

텔파이는 기본적으로 트랜잭션을 기초적인 단계에서 수행한다. Post 나 AppendRecord 등과 같은 메소드에서 자동적으로 해당 트랜잭션을 수행하고 commit 하게 된다. 문제는 이런 기본적인 제어방법은 추가/수정/삭제 작업을 할 때의 문제 발생은 막을 수 있지만, 네트워크 트래픽이나 여러 개의 작업 단위로 이루어진 문장에서는 안전성을 보장할 수 없다.

#### - 개발자가 지정하는 제어방법

메소드	설 명
StartTransation	트랜잭션의 시작 포인트를 설정한다.
Commit	진행된 트랜잭션을 물리적인 데이터베이스에 적용한다.
RollBack	현재 진행된 트랜잭션을 원상태로 복귀시킨다.

참고로, 텔파이 4에서는 로컬 DB 인 파라독스에서도 이러한 트랜잭션을 지원한다.

### 4. 트랜잭션의 옵션과 지원되는 DB에서의 동작

텔파이에서는 다음과 같은 작업을 통하여 작업을 수행한다.

```
TDatabase.StartTransaction;           // 트랜잭션의 시작을 지시한다.  
TDatabase.Commit;                     // 지금까지의 작업을 실제 DB에 반영한다.  
TDatabase.Rollback;                  // 지금까지 작업내용을 취소한다.
```

StartTransations 메소드와 Commit, Rollback 은 try ... finally, try ... except 블록을 활용하여 작업을 수행한 다음의 결과에 대해서 작업을 수행할 수 있도록 하는 것이 좋다.

inTransacion 이라는 Boolean 형의 프로퍼티는 실제 트랜잭션의 작업 시에 True 가 선언되면 StartTransation 과 동일한 역할을 하며, Commint 이나 Rollback 과 같은 기능을 하기 위해서는 False 를 설정하면 된다.

트랜잭션에는 3 가지 방식이 있는데 다음과 같다.

레벨	내용
TiDirtyRead	현재 작업과 관계없는 내용은 다른 사람들이 읽을 수 있다.
TiReadCommotted	현재 작업이 다른 트랜잭션에서의 연산에 참여할 수 있다.
TiRepeatableRead	현재 작업중인 데이터에 대해 다른 트랜잭션이 참여할 수 없다.

약간의 문제는 이러한 트랜잭션이 각각의 RDBMS 마다 다르게 동작한다는 것이다. 다음의 표는 이러한 차이점에 대해서 나열해 보았다.

서버	레벨 설정	실제 동작
Oracle	tiDirtyRead	tiReadCommitted tiReadCommitted
	tiReadCommitted	tiRepeatableRead(ReadOnly)
	tiRepeatableRead	
Sybase, MS-SQL	tiDirtyRead	tiReadCommitted tiReadCommitted
	tiReadCommitted	tiRepeatableRead
	tiRepeatableRead	
DB2	tiDirtyRead	tiDirtyRead
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
Informix	tiDirtyRead	tiDirtyRead
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
인터베이스	tiDirtyRead	tiReadCommitted tiReadCommitted
	tiReadCommitted	tiRepeatableRead
	tiRepeatableRead	
Paradox, dBase, Access, FoxPro	tiDirtyRead	tiDirtyRead
	tiReadCommitted	지원하지 않음
	tiRepeatableRead	지원하지 않음

여기에서 보면 알겠지만 로컬 DB 의 경우 tiDirtyRead 의 옵션 밖에 차이가 나지 않는다. 이 옵션은 ODBC 인 경우에도 동일하다.

## 5. 트랜잭션의 종류

BDE 설정에서도 해당 데이터베이스의 트랜잭션 동작에 대한 기본설정을 할 수 있다. 이것

이 바로 SQLPASSTHRU MODE 인데, 다음의 3 가지로 지원된다.

모 드	설 명
SHARED AUTOCOMMIT	기본적인 설정으로 레코드(행)단위의 모든 작업에 Commit 을 자동 작동시킨다. SQL 과 BDE 메소드가 동일하게 동작한다.
SHARED NOAUTOCOMMIT	프로그램 내부에서 정확하게 트랜잭션이 기술된 시점에서 트랜잭션을 수행한다. SQL 과 BDE 메소드가 동일하게 동작한다.
NOT SHARED	완전히 별개의 트랜잭션을 기술하며 SQL 과 BDE 는 다른 연결방법을 사용한다.

#### ● 로컬 DB 의 원격 DB 로의 이동

일반적으로 작성된 파라독스나 DBase 형태의 DB 를 상용 DBMS 의 데이터로 전송하는 경우에 해당 프로그램을 작성할 필요 없이 간단한 방법으로 이러한 데이터를 전송할 수 있다. 이 경우에는 개발자는 앨리어스를 2 개 준비하여 Source 에서 Target 으로 데이터를 전송하기만 하면 된다.

이 때 사용하는 것이 바로 데이터 이동 위저드(Data Migration Wizard)인데, 주의할 점은 데이터베이스의 형태가 바뀌므로 전송에 앞서 다음과 같은 몇 가지 사항을 고려해야 한다.

1. 각각의 필드의 속성이 Target 에서 지원되는지 살펴 본다.
2. 필드명중에 해당 DB 의 예약어인 경우가 있으므로, 필드명의 적절하게 변경한다.

필자의 경험상 자료를 올리는 경우에는 인덱스를 일단 모두 삭제한 다음 Upsize 하는 것이 성공확률이 높다.

#### ● TDataSource 의 사용 방법

TDataSource 컴포넌트는 단순히 데이터 세트 컴포넌트와 데이터 인식 컨트롤을 연결하는 연결방법으로만 사용하는 것은 않는다. 여기서는 이 컴포넌트의 몇 가지 유용한 기능에 대하여 간단하게 알아보자

##### 1. 마스터/디테일 연결

마스터/디테일 테이블을 TDataSource 컴포넌트를 이용하여 연결할 수 있다. 이때 디테일 테이블은 반드시 TTable 을 사용해야 한다.

## 2. 다양한 이벤트의 활용

이벤트	설 명
onDataChange	해당 DataSet 이 변경될 경우에 발생한다.
onStateChange	해당 DataSet 의 작업형태가 변경될 경우에 발생한다.
onUpdateData	물리적인 데이터가 변경될 경우에 발생한다.

### ● 캐쉬 업데이트(Cached Updates)

트랜잭션과 유사한 기능을 하는 것으로, 개발자가 원하는 테이블에 변경 작업(추가/수정/삭제)작업을 취할 경우에 다중 레코드를 처리하는 것을 허용하여 중간 버퍼에 이러한 작업을 유지한 다음 원하는 시점에 이 작업을 물리적인 데이터베이스에 반영하도록 하는 것을 말한다.

이 기능은 모든 데이터 세트 컴포넌트에서 사용할 수 있는데, CachedUpdates 프로퍼티 값이 True 가 설정되어 있으면 동작이 가능하다. 사용하는 메소드는 다음과 같다.

메소드	설 명
ApplyUpdates	버퍼의 내용을 물리적인 DB 에 저장합니다.
CancelUpdates	현재 버퍼에 저장된 변경내용을 삭제합니다.
RevertRecord	현재 레코드만 원래의 레코드로 변경합니다.

참고로, DataBase1.ApplyUpdates([테이블, 테이블])과 같은 코드를 사용하면 여러 개의 테이블에 있는 캐쉬 업데이트를 하나의 코드로 동작시킬 수 있다. 그러나, CancelUpdates 메소드는 데이터베이스 컴포넌트에서 지원하지 않는다.

### ● TUpdateSQL 컴포넌트

TQuery, TStoredProc 컴포넌트 이외에 기본적으로 TUpdateSQL 이라는 컴포넌트가 지원된다. TQuery 컴포넌트를 이용하면 Select, Insert, Delete, Update 등의 모든 SQL 문장을 사용할 수 있으며, 심지어는 저장 프로시저나 트리거(Trigger)도 만들 수 있다. 그렇지만, 실제로 SQL 문장을 사용하다 보면 자그마한 리소스나 속도 문제에 민감해지는 경우가 많다. TQuery 의 경우 모든 SQL 문장을 동작시킬 수 있도록 많은 내부코드를 가지고 있기 때문에, 일반적인 SQL 문장을 사용하는 경우에 주로 사용되는 Insert, Update, Delete 의 3 가지 기본적인 작업만 사용할 때 이것을 사용하는 것은 다소 낭비라고 할 수 있다. 이런 경우에 보다 빠른 작업과 관리의 편리성을 위하여 TUpdateSQL 컴포넌트를 사용한다.

이 단순화된 컴포넌트는 필요한 3 가지의 행동코드를 가질 수 있어서, 매번 추가/수정/삭제

할 때마다 SQL 프로퍼티의 내용을 지웠다 추가했다 하는 코드를 줄일 수 있으므로 간단한 저장 프로시저처럼 활용할 수 있다.

- 전역 세션(Global Session)의 기능 활용

텔파이에서 현재 사용하고 있는 데이터베이스의 정보나 테이블 정보를 검색해야 할 경우가 있다. 이런 경우에는 세션 컴포넌트를 사용하는데, 기본적으로 세션이 하나 전역으로 선언되어 있고, 이를 디폴트 세션으로 사용하게 된다.

이를 이용하여 Session.GetDriveNames(Strings 객체) 메소드를 사용하면 현재 사용되는 데이터베이스 드라이브 정보를 얻을 수 있으며, Session.GetDataBaseNames(Strings 객체) 메소드를 사용하면 현재 사용되는 데이터베이스명을 얻을 수 있습니다.

마찬가지로, Session.GetTableNames('데이터베이스명', 형태('\*.db'), False, False, Strings 객체) 메소드를 이용하면 테이블의 이름을 얻을 수 있다.

- 북마크(Bookmark) 사용하기

북마크는 책갈피라고 생각하면 된다. 원하는 위치를 지정해 놓고 해당위치로 빠르게 이동할 수 있게 해준다.

```
북마크 := Table1.GetBookMark;           //현재의 위치를 저장한다.
Table1.GotoBookmark(북마크);             // 해당 북마크로 이동
Table1.FreeBookmark(북마크);             // 해당 북마크를 삭제
```

## 자료 처리시의 에러처리 방법

잘 만들어진 데이터베이스 어플리케이션은 예기치 않은 에러 상황이 발생하더라도 이를 고급스럽게 처리하는 방법을 제공해야 한다. 데이터베이스 어플리케이션에서 가장 많이 발생하는 에러는 EDatabaseError 와 EDBEngineError 이다.

- EDatabaseError

실제 DB 프로그래밍을 할 때 만나게 되는 많은 에러 상황은 EDatabaseError 클래스를 사용하여 해결할 수 있다. 이를 처리하는 방법을 pseudo-code 로 나타내면 다음과 같다.

```
try
    ... DB 작업
```



```

except
    on EDatabaseError do
        ...
        raise;
end;

```

즉, DB 작업 중에 문제가 발생할 가능성이 있는 부분을 try...except 블록으로 감싸고 해당 부분에서 EDatabaseError 객체로 발생한 에러를 추적하여 대처할 수 있다.

그러나, EDatabaseError 객체만 가지고는 직접적으로 어떤 오류가 발생하였는지를 알 수 없고 실제 내용은 파생된 EDBEngineError 클래스를 참조해야 알아낼 수 있다.

보통의 경우에는 데이터 세트의 OnPostError, OnEditError, OnDeleteError 의 이벤트를 참조하면 대다수 문제가 해결된다. 해당 이벤트에는 문제가 발생한 데이터 세트의 EDatabaseError 객체가 전달되며 문제를 해결하기 위해 TDataAction 데이터 형의 값을 지정하게 된다. 지정할 수 있는 값에는 daFail(실패), daAbort(중단), daRetry(재시도) 중에서 선택할 수 있다.

#### ● EDBEngineError

EDBEngineError 클래스는 Errors 배열에 BDE 에서 발생하는 모든 오류의 값과 내용을 가지고 있다. 그렇기 때문에, 이를 이용하면 보다 구체적인 에러 처리를 할 수 있게 된다. 이 클래스의 멤버에서 ErrorsCount 에는 발생한 에러의 개수가 담겨 있고, Errors 에는 발생한 에러의 내용이 TDBError 의 배열 형태로 들어 있다. 이들의 값에는 다음과 같은 것들이 있다.

속 성	내 용
Category	오류코드 범주
ErrorCode	오류코드 숫자 (DBIResult 형태)
Message	서버에 연결되어 있으면 서버에서 리턴된 에러메시지, 보통은 BDE 의 에러메시지
NativeError	RDB 에서 발생한 에러인지 체크, 0 이 리턴되면 서버오류가 아닌 BDE 오류
SubCode	하위 오류코드

참고로, Category 와 SubCode 는 Byte 형태로 두 코드는 상위와 하위의 에러코드를 지칭하게 된다.

리턴되는 에러코드는 BDE.INT 파일을 참조하면 다양하게 존재하는 에러의 코드를 구체적으로 알 수 있다. 해당 BDE.INT 에는 Delphi4 의 DOC 디렉토리에 333kB 의 파일로 존재하는데, 실제 에러는 Byte 값으로 존재하며 DBI 상에서의 에러는 ErrorCode 의 Word 타입

의 자료를 참조하여 에러를 찾을 수 있다.

## 자료를 화면에 제시하는 방법

데이터베이스의 내용을 표시하기 위해서 데이터 인식 컴포넌트들을 사용하면 필요한 데이터베이스의 내용을 제시할 수 있으며 쉽게 데이터베이스의 레코드들간의 이동이 가능하다. 이렇게 데이터 인식 컴포넌트를 이용하여 데이터를 표시하는 방법은 간단히 언급한 바 있고, 델파이의 도움말에도 잘 나와 있으므로 자세한 설명은 생략하도록 한다.

필자는 앞에서도 언급했듯이 데이터 인식 컴포넌트들을 거의 사용하지 않는다. 실제 테이블이나 쿼리를 사용할 때 데이터 인식 컨트롤을 사용하면, 소규모 프로젝트에서는 큰 문제가 되지 않지만, 대규모 프로젝트인 경우에는 많은 시행착오를 겪기 쉽다. 엉뚱한 Access Violation Error 를 비롯하여 레코드의 locking 문제 등이 많이 발생하기 때문에, 필자의 경우에는 StringGrid 와 TListView, TListBox 등의 기본적인 화면용 컴포넌트를 주로 사용한다.

데이터는 TQuery 나 TStoredProc, TUpdateSQL 컴포넌트를 주로 사용한다. 물론, 이러한 방식은 델파이 2 까지 사용되던 기법이었고, 델파이 3 부터는 TClientDataSet 을 많이 활용한다. 전장에서도 충분히 이야기 했지만, TClientDataSet 컴포넌트를 사용하면 필요한 데이터만 클라이언트에서 변경하고 이렇게 변경된 내용을 Delta 프로퍼티에 보관하여 RDB 에 반영할 수 있기 때문에, 많은 잇점이 있다.

데이터 인식 컨트롤을 사용하는 방법이나 화면용 기본 컴포넌트를 이용하여 데이터를 표시하는 방법에 대해서는 특별히 언급하지 않아도 여러 곳에서 정보를 얻을 수 있으므로 여기에 대한 설명은 생략하도록 하며, 개발 시에 유용한 몇 가지 팁들은 18 장에서 소개하였다.

### ● TeeChart 의 활용

델파이 3 에서부터 포함된 TeeChart 는 델파이의 native VCL 로 막강한 그래프를 만들어낼 수 있는 유용한 컴포넌트 세트이다. 보통의 경우에는 델파이에서 제공되는 위저드를 이용하면 기본적인 그래프는 간단히 생성할 수 있다. 이 방법에 대해서는 기본으로 제공되는 도움말을 참고하면 쉽게 알 수 있을 것이다.

### ● Decision Cube 의 활용

델파이 3 에서부터 지원되는 Decision Cube 는 의사결정 도구의 결정판이다. 이 도구를 사용하면 드릴-다운(Drilled down)기법을 이용하여 각종 데이터를 조회하여 볼 수 있다. 여기에 대해서는 다음 장에서 자세히 다룰 것이다.

## 사용자의 요구에 따라 프린터로 출력하는 방법

프린터로 데이터를 출력하는 방법에는 여러가지 방법이 있습니다. 이를 크게 나누어 보면 다음과 같이 3 가지로 생각할 수 있다.

1. 일반적인 프린터 메소드를 사용하는 방법
2. QuickReprot 를 사용하는 방법
3. 다른 서드파티를 사용하는 방법

### ● 일반적인 프린터 메소드를 사용하는 방법

WriteLn 함수와 Canvas 를 사용하여 원하는 출력물을 만들어 낼 수 있다. WriteLn 의 사용방법은 다음과 같다.

WriteLn(텍스트화일, 문자열, ... );

참고로, 보통 테이블이나 쿼리를 Open 하여 놓은 상태, 특히 데이터 인식 컨트롤을 사용한 경우에는 화면에서 작업하다가 프린팅과 같은 전체 자료를 사용하여 출력하는 경우에는 Bookmark 를 사용하여 작업하던 장소를 저장하고, 테이블이나 쿼리문을 다시 동작하여 사용하면 간단하게 작업할 수 있다. 또한 데이터 인식 컨트롤을 사용할 경우 전체 내용을 검색하면 DataSource 가 연결된 경우에는 해당 데이터의 Refresh 때문에 수행이 느려진다.

이럴 때에는 DisableControls 와 EnableControls 메소드를 활용하여 활성을 조절하면 프로그램의 수행이 빨라진다.

일반적인 문자열을 출력하는 경우에는 WriteLn 도 무방하지만, 색을 지정하거나 그림이나 도형을 출력하는 경우에는 Canvas 를 사용하여 출력하여야 한다. 다음의 코드를 살펴 보자.

```
Printer.BeginDoc;
```

```
Printer.Canvas.TextOut(x, y, 문자열);
```

```
Printer.EndDoc;
```

이때 주의할 점은 x, y 의 해당 좌표값을 지정하는 것과 좌표값은 Pixel 단위이므로 해당 폰트의 Height 와 Width 를 알아서 계산해 주어야 한다는 점이다. 그리고, 반드시 EndDoc 를 호출하여 끝을 맺어야 한다. 이 부분이 빠지게 되면 무한루프를 돌 우려가 있으니 주의하기 바란다. 이외에도 NewPage, Abort, PageNumber 등을 이용하여 새로운 페이지 출력, 출력 중단, 페이지 번호 등을 알아낼 수 있다.

- 쿼리 리포트(Quick Report)의 활용

쿼리 리포트는 델파이 3 부터 기본적으로 제공되는 출력 컴포넌트로서 강력한 리포팅 기능을 가지고 있어서 기존의 리포트 스미스를 완전히 대체하였다. 쿼리 리포트에 대한 더 자세한 내용은 다음 장에서 다루도록 한다.

## 정 리 (Summary)

이번 장에서는 델파이에서 지원하는 기본적인 컴포넌트를 이용하여 프로그래밍하는 방법과 기본적인 데이터베이스 어플리케이션을 작성할 때 고려해야 할 여러가지에 대해서 다루어 보았다.

다음 장에서는 이번 장에서 간단히 알아본 Decision Cube 와 쿼리 리포트에 대해서 더 자세하게 알아보도록 할 것이다.

## Decision Cube 와 쿼리 리포트의 활용

앞장에서도 간단히 언급한 바 있지만, 델파이 3 부터 기본 컴포넌트 세트에 채용된 Decision Cube 와 쿼리 리포트는 뛰어난 성능과 다양한 기능으로 데이터베이스 어플리케이션의 기능을 한단계 상승시키는 역할을 하고 있다.

Decision Cube 는 의사결정 도구의 결정판이다. 이 도구를 사용하면 드릴-다운(Drilled down)기법을 이용하여 각종 데이터를 조회하여 볼 수 있다. 그리고, 쿼리 리포트는 처음 시작한 쉐어웨어 VCL 컴포넌트로 출발하였지만, 그 성능과 편리함을 인정 받아 기존의 리포트 스미스를 밀어내고 델파이의 기본 출력 컴포넌트로서의 입지를 확실히 하였다.

### Decision Cube 의 활용

Decision Cube 를 사용하는 방법은 아주 간단하지만, 활용되는 예가 드물기 때문에 여기에서는 간단한 예제를 이용하여 그 사용법을 알아보도록 한다.

Decision Cube 에 관여하는 컴포넌트는 모두 6 개로 이들의 역할은 다음과 같다.

컴포넌트	설 명
TDecisionCube	Decision Cube 의 가장 중심이 되는 컴포넌트로서, TDecisionQuery 에서 만들어진 DataSet 을 Decision 패밀리에 연결시켜 준다.
TDecisionQuery	Decision 패밀리에서 사용할 수 있도록 쿼리 DataSet 을 연결시켜 준다.
TDecisionSource	Decision Cube 에서 연결된 DataSet 을 Decision 패밀리가 사용할 수 있는 DataSource 의 역할을 수행한다.
TDecisionPivot	Decision Cube 에서 중심점이나 소트의 중심이 되는 부분을 버튼 형식이나 라디오 버튼 형식으로 나타내어, 사용자의 선택을 Decision 에 반영한다.
TDecisionGrid	Decision Cube 의 결정된 데이터를 그리드 형태로 제시한다.
TDecisionGraph	Decision Cube 의 결정된 데이터를 그래프 형태로 제시한다.

#### 1. Decision Cube 의 구성 방법

먼저 Decision cube 를 이용하여 데이터를 보기 위해서는 TDecisionQuery 컴포넌트를 이용하여 데이터 세트를 만들어 내도록 한다. 폼에 TDecisionQuery 컴포넌트를 하나 올려 놓은 다음 기본적인 DatabaseName 프로퍼티를 DBDEMOS 로 선택하고, 오브젝트 인스펙터에서 SQL 프로퍼티 에디터를 이용하여 SQL 문장을 다음과 같이 설정한다.

```
SELECT PaymentMethod, ShipVIA, Terms, ShipDate, COUNT(AmountPaid), SUM(AmountPaid),
```

```

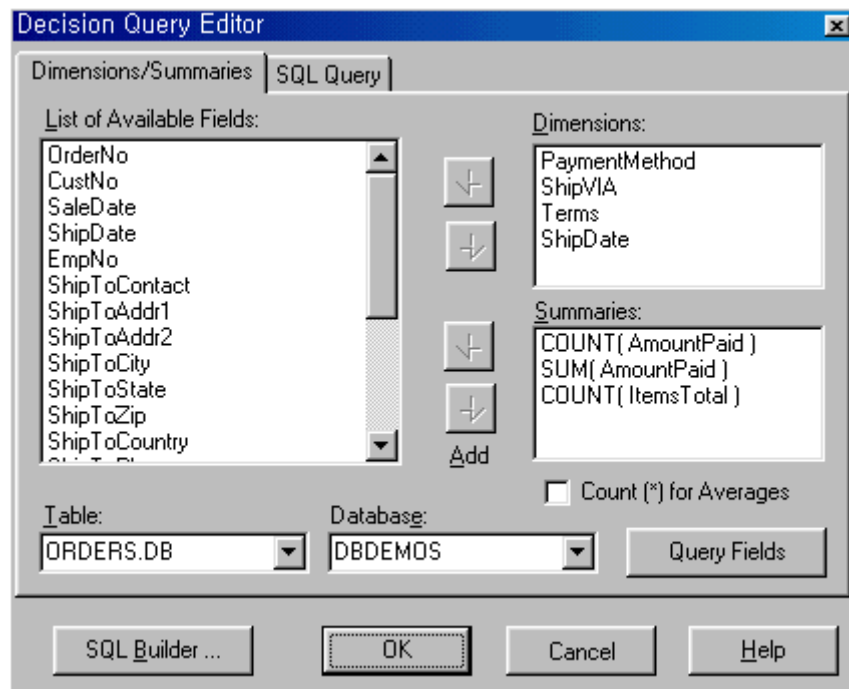
COUNT(ItemsTotal)
FROM "ORDERS.DB" ORDERS
GROUP BY PaymentMethod, ShipVIA, Terms, ShipDate

```

이 쿼리의 내용은 orders.db 파일에 있는 여러 주문 아이템 들에 대한 지불 방법, 선적 방법과 기간, 그리고 선적일과 현재의 지불 상태, 아이템의 수 등에 대한 내용을 지불 방법과 기간 및 선적일, 선적 방법 등의 내용을 참고로 하여 그룹화한 데이터 세트를 뽑아내게 된다. 이제 이 컴포넌트의 Active 프로퍼티를 True 로 하면 해당되는 데이터 세트가 만들어진다. 참고로, Decision Query 의 자세한 내용이 보고 싶으면 TDecisionQuery 컴포넌트를 더블 클릭하면 다음의 폼이 나타날 것이다.

이 화면을 보면 테이블의 내용과 화면에 보이기 위한 필드, 합계 형식으로 나타나는 필드들이 나열되며, 어떤 앨리어스에서 어떤 테이블을 사용하였는지 알 수 있다.

또한, 여기에서 직접 사용할 Dimension 과 Summary 필드를 결정하여 추가할 수도 있으며, 이 내용들이 SQL 프로퍼티에 자동으로 반영된다.



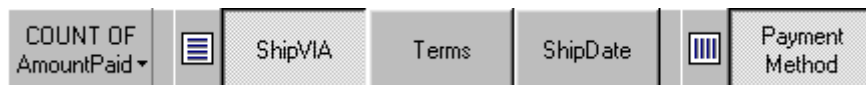
마찬가지로 SQL Query 탭을 선택해보면 앞에서 넣었던 SQL 문장이 나열되어 있으며, 여기서 바로 SQL 문장을 수정할 수도 있다. 이때 SQL 문장을 만드는 것보다 액세스 등에서 사용했던 방법과 비슷하게 비주얼한 환경에서 쿼리를 자동으로 생성하고자 한다면, SQL Builder 버튼을 클릭하여 쿼리를 생성할 수도 있다.

이제는 이렇게 만들어진 데이터 세트를 TDecisionSource 컴포넌트를 통하여 다른 Decision 패밀리에 연결하도록 한다. 이 방법은 기본적으로 TTable 과 TDataSource 를

이용하여 데이터 컨트롤에 연결하던 기존의 방식과 동일하므로 그다지 어렵지 않게 사용할 수 있을 것이다. 먼저 TDecisionCube 와 TDecisionSource 컴포넌트를 폼에 추가하고 TDecisionCube 컴포넌트의 DataSet 프로퍼티를 DecisionQuery1, TDecisionSource 컴포넌트의 DecisionCube 프로퍼티를 DecisionCube1 으로 설정한다.

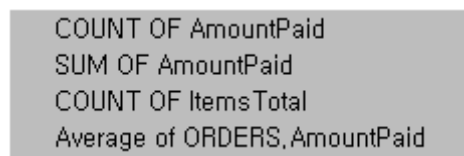
이렇게 연결된 TDecisionSource 컴포넌트를 통하여 TDecisionPivot 과 TDecisionGrid 컴포넌트를 연결하도록 하자. 이때에 Pivot 은 DecisionGrid 에 제시되는 데이터를 조절하는 역할을 한다.

아마도 Pivot 의 내용이 다음과 같이 나타날 것이다.



이 내용을 잘 살펴보면, 앞의 Decision Query Editor 대화 상자에서 Dimensions 와 Summaries 에 나열된 내용으로 이루어져 있다는 것을 알 수 있을 것이다.

Dimensions 에 선택된 필드는 Pivot 에 버튼의 형태로 나타나서 그룹을 동적으로 지정할 수 있게 되며, Summaries 에 지정된 조건들은 다음과 같이 또 다른 형태의 선택을 할 수 있도록 주어진다.



여기에서, 마지막에 나열된 Average of ORDERS.AmountPaid 는 Count 와 SUM 을 사용하여 자연스럽게 자동으로 추가된 내용이다.

마찬가지로 우측의 화면을 보면, ShipVIA 와 ShipDate 등의 버튼에서 해당되는 데이터를 전체 내용을 다 볼수도 있고, 드릴-인(Drill In)하여 조절하여 볼 수도 있다.

여기에서 주의해서 살펴보아야 하는 것은 창살처럼 보이는 비트맵 버튼 들이다. 앞 그림에서 ShipVIA 버튼 좌측에 있는 버튼가 같이 가로로 창살이 있는 비트맵 버튼은 가로열에 영향을 주는 필드를 이 버튼에서 선정한다는 의미이다. 반대로 세로로 창살이 있는 비트맵 버튼은 세로열에 영향을 주는 필드를 선택하는 것이다.

이러한 필드를 조절할 때에는 해당되는 버튼 위에서 마우스의 오른쪽 버튼을 눌러보면 나타나는 팝업 메뉴를 이용하는데, 이때에 Move to Column Area 를 메뉴를 선택하면 Row Area 로 이동하게 되며, 반대로 Row 에서는 Column 으로 이동하는 메뉴가 나타나게 되며 이렇게 동적으로 pivot 의 이동이 가능하다.

그리고, 해당되는 필드의 Drilled In 메뉴를 선택하면, 해당되는 필드의 내용들 중에서 공통적인 부분들만 뽑아서 선택을 할 수 있도록 도와준다. 다음 화면은 ShipDate 버튼에서

Drilled In 메뉴를 선택할 때 나타난다.

Open Dimension All Values
1988
1989
1992
1993
1994
1995

여기에서 Open Dimension 메뉴를 선택하면 처음의 해당 데이터가 모두 DecisionGrid 의 필드에 영향을 끼치며 나타나게 된다. 또한, All Values 메뉴를 선택하면 해당되는 데이터를 전부 나타낸다. 이와 같이 사용자는 마우스의 왼쪽 버튼과 오른쪽 버튼을 번갈아 사용하면서, 해당되는 데이터를 자유자재로 볼 수 있다.

즉, Open Dimension 메뉴를 이용하면 DecisionGrid 에 해당되는 Row 와 Column 에 필드에 해당되는 그룹 형태로 나타나며, Drilled In 메뉴를 선택하면 DecisionGrid 에 나타나는 데이터에 필터를 사용한 것처럼 데이터를 걸러낼 수 있다.

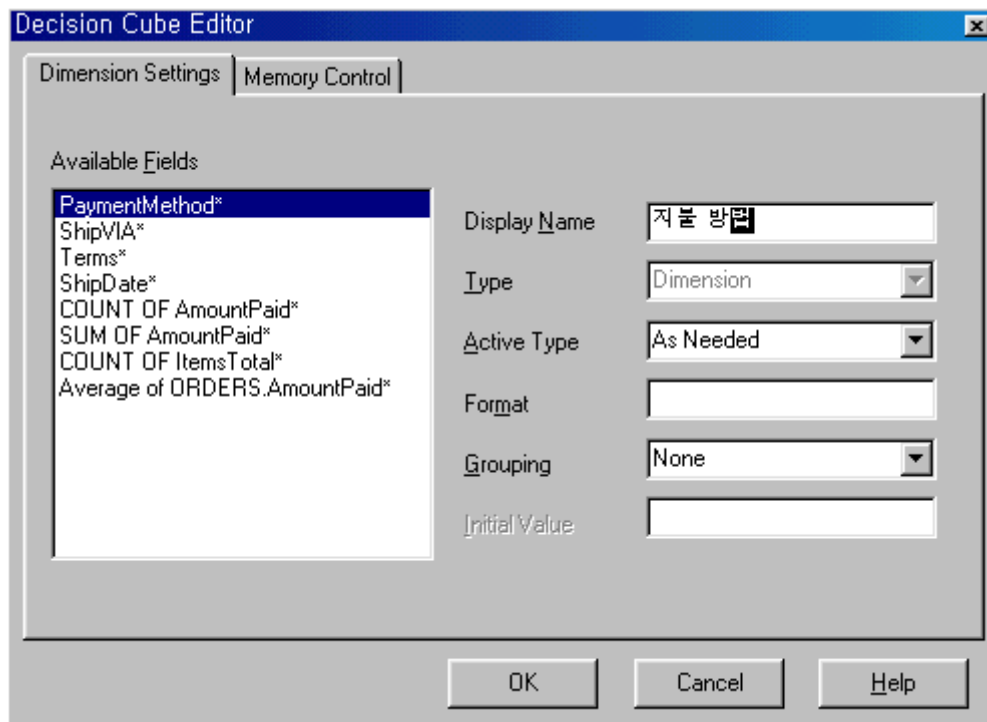
TDecisionGrid 컴포넌트에서 특이한 점은 그리드의 기능에서 특별하게 확장된 부분은 없고, 다만 그리드 내에서 Dimension 의 subtotal(노란색 +, - 그림)부분을 토글할 수 있어서, 해당되는 내용의 중간합계를 자유자재로 볼 수 있다는 점이 다르다.

마지막으로, 이렇게 표로 나타낼 수 있는 데이터를 그래프로 동적으로 표현하기 위해서는 TDecisionGraph 컴포넌트를 이용한다.

이와 같이 Decision Cube 를 이용하면 복잡한 데이터를 동적으로 사용자가 걸러서 볼 수 있는 훌륭한 인터페이스를 제공할 수 있다. 실제 OA 응용 프로그램을 작성할 때에 사용자로 하여금 다양한 관점에서 데이터를 바라다 볼 수 있는 인터페이스를 제공해 준다.

참고로, 필드 이름과 다르게 보기 좋은 이름을 사용하려면 TDecisionCube 컴포넌트의 DimensionMap 프로퍼티의 프로퍼티 에디터를 이용하여 편집하면 된다. 다음 그림은 이를 편집하는 대화 상자이다.

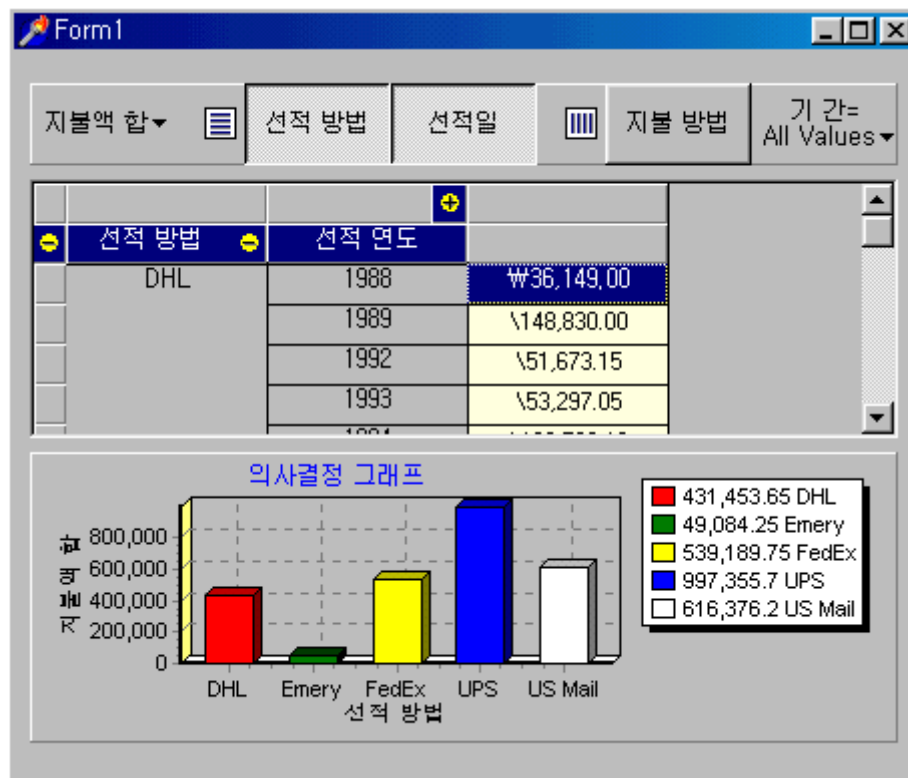




항간에는 이 Decision Cube 의 기능이 느리다고 하는 VB 사용자가 있지만, 실제 VB 에서 지원되는 것은 크리스탈 리포트에서 지원이 되는 부분이며, 사용자가 크리스탈 리포트를 사용해야 한다는 불편함이 있고, 또한 VB 프로그램에서 크리스탈 리포트에게 이러한 값들을 전달하여야 하는 불편함이 있다.

하지만 텔파이를 이용하면 간편하게 프로그램에 포함시킬 수 있으며, 이러한 프로그램도 사용자가 확장하거나 조절할 수 있게 프로그래밍할 수 있는 매우 뛰어난 컴포넌트이다.

이 예제 프로그램의 실행 화면은 다음과 같다.



## 퀵 리포트(Quick Report)의 활용

퀵 리포트는 native VCL 이기 때문에 추가적인 라이브러리나 ocx 파일을 제공하지 않아도 되고, 그 리포팅 기능 자체도 상당히 강력하다. 이를 이용하여 간단한 한 페이지 문서부터 다양한 형태의 그래프를 출력하는 복잡한 리포트까지 만들어낼 수 있다.

퀵 리포트를 이용하여 출력 폼을 디자인하고, 이를 출력하는 코드는 다음과 같이 간단하다.

```
폼.QuickRep.Print;
```

```
폼.QuickRep.Preview; //미리보기
```

그러면, 퀵 리포트의 주요 컴포넌트의 기능과 사용 방법에 대해서 간단히 알아보도록 하자.

### 1. TQuickRep 컴포넌트

이 컴포넌트는 퀵 리포트의 가장 기본적인 컴포넌트로 델파이의 폼을 리포트로 전환하는 역할을 한다. 오른쪽 버튼을 클릭하고 팝업 메뉴에서 Report Setting 을 선택하면 다음과 같이 리포트 용지에 대한 여러가지 기본 설정을 할 수 있다.

참고로 퀵 리포트는 도트 프린터를 사용하거나 A4 이외의 종이를 사용할 때에는 오동작하는 경우가 많기 때문에, 일단 해당 기종에서 테스트를 해서 확인을 하고 출력이 정상적으로 나올 때 사용하는 것이 좋다.

퀵 리포트에서 자주 사용되는 프로퍼티는 다음과 같다.

프로퍼티	내 용
PageCount	출력되는 전체 페이지수
PageNumber	현재 출력되는 페이지번호
RecordNo	사용중인 레코드 번호
PageFrame	페이지의 여백
PaperSize	용지의 크기
ReportTitle	리포트의 제목
Orientation	출력방향 poPortait(세로), poLandscape(가로)
ShowProgress	출력진행상태 Progrees 표시유무
PaperWidth	용지의 폭
PaperLength	용지의 길이

Columns	컬럼의 수
---------	-------

이외에 Band 의 주요 프로퍼티는 다음과 같다.

프로퍼티	내 용
HasColumnHeader	각페이지에 Column 부분을 사용한다
HasDetail	테이블의 정보를 모두 출력
HasPageFooter	아래 출력내용 설정
HasPageHeader	위 출력내용 설정
HasSummery	리포트 끝에 한번만 출력
HasTime	보고서 시작부에 한번만 출력

## 2. TQRBand 컴포넌트

TQRBand 컴포넌트는 쿼리 리포트에서 가장 많이 사용되는 밴드로 BendType 프로퍼티의 값을 조절함으로써 출력되는 곳의 위치를 지정할 수 있도록 한다.

주요 프로퍼티의 내용은 다음과 같다.

프로퍼티	내용
ForceNewColumn	출력 전에 새로운 열을 만든다
ForceNewPage	출력 전에 새로운 페이지를 만든다
LinkBand	여러 개의 밴드를 페이지에 출력 시에 일정한 여백을 갖도록 하고, 공간이 충분하면 링크된 밴드가 이어서 출력되게 한다. 단, 공간이 부족하면 새로운 라인에 출력한다.
HasChild	True 시에 자동으로 Child 가 나타난다.
Frame	테두리의 사각형의 선을 그릴 것인지 결정한다.

BandType 프로퍼티는 다음과 같은 값을 가질 수 있다.

값	내 용
rbChild	다른 밴드가 연결되어 있으면 연결된 밴드를 먼저 출력하고 나중에 출력
rbColumnHeader	각 페이지의 윗 부분에 나오게 한다
rbDetail	테이블 정보가 모두 출력
rbGroupFooter	테이블 정보 맨 끝에 한번 출력
rbGroupHeader	테이블 정보 맨 처음 한번 출력

rbOverlay	페이지 왼쪽 상단부터 출력
rbPageFooter	페이지 아래 출력
rbPageHeader	페이지 위에 출력
rbSubDatail	master/detail 부분의 detail 부분에 사용
rbSummary	맨 마지막에 한 번 합계 낼 경우에 사용
rbTitle	시작부분에 한 번 수행

### 3. TQRSysData 컴포넌트

리포트 출력 시에 시간, 날짜 등의 내용을 출력하는 컴포넌트이다. Data 프로퍼티를 조절하면 다양한 속성을 이용할 수 있다. Data 프로퍼티에 사용할 수 있는 값에는 다음과 같은 것들이 있다.

값	내 용
qrsDate	시스템 날짜
qrsDateTime	시스템 날짜 + 시간 출력
qrsDetailNo	현재의 레코드 번호
qrsPageNumber	현재 출력 페이지 번호
qrsDetailCount	출력될 총 레코드 수
qrsReportTitle	리포트의 제목으로 TQuickRep 컴포넌트의 ReportTitle 프로퍼티 사용
qrsTime	출력 시에 시간 출력

Text 에 들어 있는 내용을 Data 에서 정의된 내용을 출력하기 전에 먼저 찍고, Data 프로퍼티에서 설정된 내용을 출력한다.

### 4. TQRGroup 컴포넌트

리포트에서 마스터/디테일 형태의 리포트를 만드는 경우에 사용한다. 이 컴포넌트에 보면 FooterBand, HasChild, Master 등의 프로퍼티가 있는 것을 알 수 있을 것이다. 이때에 Expression 의 프로퍼티를 이용하면, 원하는 DataSet 의 필드에서 특정 함수를 거쳐서 값을 산출할 수 있다.

### 5. TQRDB, TQRLabel 컴포넌트

TQRLabel 은 출력되는 쿼리 리포트에 원하는 내용을 입력할 수 있는 라벨이라고 생각하면 된다. TQRDB 는 DBLabel 과 거의 동일하다, 해당 밴드 위에 올려놓고 DataSet 의 Fields

를 연결하면 해당 내용을 보여준다.

## 6. TQRPreview 컴포넌트

쿼 리포트는 기본적인 Preview 화면을 지원한다, 하지만 영문으로 나오고 모양이 그리 이쁘지 않다. 이 컴포넌트를 이용하면 이러한 Preview 화면을 조정할 수 있다. 만들어진 Preview 화면을 사용하려면 쿼 리포트에 등록하여 사용할 수 있다.

이 밖에도 많은 수의 쿼 리포트 컴포넌트가 제공되고 있다. 쿼 리포트는 비교적 강력하면서도 효율적으로 사용할 수 있는 우수한 컴포넌트 세트이기 때문에, 유용성이 매우 높다. 하지만, 비교적 버그가 많은 편이어서 한번 그 버그를 경험한 개발자 들이 등을 돌리는 경우도 매우 많은 편이다.

그렇지만, 그 효율성과 장점 역시 탁월하므로 앞에서도 언급했듯이 간단한 종류의 리포트를 작성할 때에는 이를 사용하는 것이 여러모로 유리하다.

## 정 리 (Summary)

이번 장에서는 델파이에서 사용되는 우수한 컴포넌트 세트인 Decision Cube 와 쿼 리포트에 대해서 알아보았다. 이들은 비록 어플리케이션을 작성하는데 있어서 핵심적인 부분을 차지하지는 않는다. 그렇지만, 보기 좋은 떡이 먹기도 좋다는 말이 있듯이 데이터베이스 어플리케이션을 개발할 때에도 출력 화면과 인쇄물을 어떻게 생성해낼 수 있는지 여부는 가장 중요한 어플리케이션의 평가 요소 중의 하나이다.

그러므로, 이들을 효과적으로 사용하면 데이터베이스 어플리케이션의 완성도를 높일 수 있으므로 사용법을 꾸준히 익혀두는 것이 바람직하다.

## 클라이언트/서버 데이터베이스 어플리케이션의 제작 (I)

이번 장에서는 데이터베이스 어플리케이션을 원격 DBMS 를 이용하여 클라이언트/서버의 형태로 개발할 때 고려해야 할 사항 들에 대해서 2 장에 걸쳐서 알아보도록 하겠다.

텔파이 3 부터는 클라이언트 데이터 세트를 이용한 멀티-tier 형태의 데이터베이스 어플리케이션 개발이 가능해졌기 때문에, 2-tier 형태의 클라이언트/서버 어플리케이션을 개발하는 방법에 대해서는 과거와 같은 중요성이 있다고 볼 수는 없다.

하지만, 아직도 많은 개발자 들이 이런 형태로 개발을 실제로 진행하고 있기 때문에 실제 개발에서 도움이 될 수 있는 정보를 제공하고자 한다.

먼저 RDBMS 를 이용하여 어플리케이션을 작성할 때 고려해야 할 주요 내용 들과 텔파이 4 에서 새롭게 제공되는 SQL 빌더(과거의 비주얼 쿼리 빌더)의 사용 방법을 알아보고, 몇 가지 유용한 SQL 문장 들에 대한 소개와 클라이언트/서버 데이터베이스 어플리케이션을 사용할 때 유용하게 사용되는 캐쉬 업데이트(cached updates)에 대한 내용을 알아본다. 그리고, 다음 장에서는 주요 DBMS 의 특징과 데이터베이스 접속을 관리하는 방법에 대해서 알아보도록 할 것이다.

### RDBMS 의 사용

RDBMS 를 사용하여 데이터베이스 어플리케이션을 개발하는 것은 각각의 데이터베이스의 구성 요소와 텔파이의 구성 요소를 결합하는 것으로 서버 데이터베이스의 원하는 테이블과 저장 프로시저 등을 먼저 구성한 다음 텔파이에서 이러한 환경에 접근하여 테이블을 조작하고 내용을 참조하는 작업을 하는 것이다. 이때 개발자가 고려해야할 것으로는 다음과 같은 것들이 있다.

- 1) 데이터베이스에 로그인
- 2) 트랜잭션 컨트롤의 사용
- 3) 특정 SQL 서버의 확장기능 사용
- 4) 만들어진 SQL 의 튜닝작업

물론 이외에도 초기 DB 설계 방법과 데이터웨어하우스 등 많은 작업이 필요하지만, 이 책에서 논의되는 내용은 텔파이로 어플리케이션을 작성하는데 국한되므로 해당 내용만 다루도록 한다.

### RDBMS 에서의 데이터 처리 요령

보통 개발자들은 검색 메소드로 SetRange, Locate, FindKey 등의 다양한 메소드를 사용하게 된다. 그렇지만, RDBMS 를 사용할 경우 이러한 작업은 많은 부하를 RDBMS 에 주게 되며 네트워크 트래픽도 발생한다. 더군다나 여러 개의 실행 모듈로 구분되는 프로그램을 제작할 경우 RDBMS 의 접속을 유지하기 위해 각각의 실행 모듈마다 TDatabase 컴포넌트를 사용하게 된다.

실제 RDBMS 의 접속은 상당한 시간을 소모하므로 이런 로그-인과 로그-아웃 작업은 많지 않은 것이 좋다. 그렇다고 이러한 접속을 계속 유지하는 것 또한 네트워크 트래픽과 서버의 부하에 영향을 주게 된다.

텔파이 4에서는 이를 해결하기 위해 다음과 같은 방법을 지원한다.

- 접속의 유지

원격 데이터 모듈(Remote DataModule)을 사용하여 하나의 클라이언트 시스템에서 하나의 접속을 사용하는 중간 연결 어플리케이션을 사용하고, 필요한 모듈은 해당 원격 리모트 데이터 모듈을 참고하는 TClientDataSet 이나 메소드로 동작하게 하면 접속을 적게 유지할 수 있으며, 작업 속도도 빨라진다. 더구나 TClientDataSet 의 Delta 내용을 적절하게 유지하고 캐쉬 업데이트를 사용하면 네트워크 트래픽을 많이 줄일 수 있다.

- 접속의 리소스 절약

MTS 와 CORBA 의 기능 중에 MTS 의 경우에는 MTS Pooling 과 세션 관리를 통하여 서버의 자원을 절약할 수 있으며, CORBA 의 경우에는 자원관리 기능을 이용하여 많은 사용자가 접근할 때 효과적으로 자원을 배분할 수 있다.

1. 많은 클라이언트의 요구를 처리하기 위한 여러 개의 어플리케이션 서버의 구축

텔파이 3 부터 지원하기 시작한 MIDAS 와 텔파이 4 의 MTS 의 분산객체 지원기능, CORBA 의 ORB 를 사용한 분산환경은 필요한 자원의 적절한 사용과 2 개 이상의 동일한 분산객체의 로드 밸런스(load balance)를 유지하여 전체 시스템의 성능을 보장한다.

2. 기본적인 SQL 의 튜닝작업

기본적인 SQL 문장을 기술하는 것에도 많은 신경을 써야 한다. 특히 테이블의 인덱스 작업과 재인덱스 작업은 중간중간 꼭 필요하다. 텔파이에서는 어플리케이션 서버에서 스레드 등을 통한 모듈로 유휴 시간(idle time)을 적절하게 활용해서 이런 작업을 해 주어야 한다.



### 3. 데이터의 전반적인 연산작업은 RDBMS 에게 ...

RDBMS 를 사용하는 것은 단순히 테이블의 내용을 클라이언트에서 보기위한 것이 아니다. 적절한 저장 프로시저와 트리거를 사용하여 RDBMS 의 기능을 100% 사용하게 하는 것이 중요하다. 예를 들어, 1 년 동안의 전체 통계를 낼 경우에 적절한 저장 프로시저를 사용하면 RDBMS 가 빠른 시간에 결과를 돌려줄 것이다.

## SQL 작업 시의 주의할 내용과 참고사항

SQL 작업을 할 때 조금만 주의를 기울이면 많은 성능의 향상을 가져올 수 있다. 여기에 대해서 보다 자세하게 알아보도록 하자.

- 데이터의 구조와 비즈니스 어플리케이션의 구조, 규칙의 이해는 필수 !

개발자는 데이터베이스 내부의 데이터 크기와 분포를 반드시 알고 SQL 을 작성해야 한다. 또한, SQL 문장을 작성하기 전에 각 테이블의 연산방식과 구조에 대한 데이터 모델을 먼저 이해해야 한다. 이를 위해서 적극적으로 CASE 도구를 활용하여 비즈니스 관계의 데이터 객체의 구조를 문서화하는 작업이 필수적이다.

- 테스트는 실제 데이터를 가지고 할 것 !

많은 개발자가 보통 작업 시에는 많은 로드가 걸리는 실제 데이터 보다는 작게 표본화된 자료를 이용하여 프로그래밍을 하는 경우가 많다. 하지만, 데이터 100 개와 50000 개에서 동작하는 경우에 프로그래밍은 많은 차이를 보일 수 밖에 없다.

물론, 개발과정에서는 표본화된 자료로 작업할 수 있다. 그렇지만, 최소한 검사 단계에 이르면 반드시 실제 데이터 환경보다 더욱 많고 복잡한 상황 하에서 테스트해야 한다. 특히 표본자료에서 각각의 데이터의 분포도는 실제 데이터의 분포도와 유사한 구조를 가지는 것이 좋다.

- RDBMS 고유의 기능에 능숙해야 한다.

SQL 문장은 기본적으로 파싱(parsing)이라는 단계를 거친다. 그러나, identical SQL 문(저장 프로시저 등)은 이러한 단계를 거치지 않으므로 동작 속도가 빨라진다. 쉽게 이야기해서 저장 프로시저 등을 적극적으로 활용해야 한다.

- 인덱스의 사용이 중요하다.

인덱스를 적절히 사용하면 상당한 수행 성능의 향상을 가져올 수 있다. 그렇지만, 지나친 사용은 되려 시스템의 성능을 떨어뜨릴 수도 있다. 인덱스와 SQL 문장을 사용함에 있어서 다음과 같은 것들을 주의해야 한다.

1. 사용자에게 가장 많이 사용되는 컬럼을 찾아내어, 이 컬럼의 인덱스를 생성한다.
2. Join 이 자주 되는 테이블은 뷰(View)를 가지는 것이 좋고, Join 되는 필드도 반드시 인덱스가 필요하다.
3. 각 레코드(행, row)에 적은 비율을 가지고 다양한 분포도를 가지는 컬럼도 인덱스가 필요하다.
4. SQL 문장을 만들 경우에 Where 절에 함수가 사용되는 컬럼에는 인덱스를 만들지 않는 것이 더 낫다.
5. 자주 변경되는 인덱스는 인덱스 재구성으로 인하여 효율성이 떨어지므로, 이 비율이 적절한 것이 좋다.
6. 인덱스는 유일(unique)한 것이 좋다. 특히 Primary Key 부분은 반드시 unique 한 인덱스가 필요하다. 그렇지만, foreign key 와 Where 절에 사용되는 컬럼은 non-unique 인덱스를 사용하는 것이 낫다.
7. 충분히 SQL 문장을 다듬어야 한다. 특히, Where 절에서 작은 크기의 인덱스와 큰 크기의 인덱스, 유일한 인덱스, non-unique 한 인덱스 중에 어느 것을 먼저 수행하느냐에 따라 SQL 문장의 수행속도가 많이 달라진다.
8. 해당 SQL 문장의 최적화 작업을 이해해야 한다. 보통 SQL 은 RULE-BASED 나 COST-BASED 중의 하나로 구동된다. 보통 전통적인 RDBMS 의 경우 RULE-BASED 를 사용하였지만, 새로 출시된 RDB 의 경우에는 COST BASED 방식을 사용한다. 이 방식은 반드시 ANALYZE 스키마를 사용하여 데이터베이스의 이용 통계를 내고, 이 데이터를 데이터 사전 테이블에 기록하고, 이 자료를 COST BASES OPTIMIZER 를 통하여 사용하게 된다.
9. SQL 문장들 간의 여러 상관관계에 대해 전체 SQL 문장을 염두에 두어야 한다. 내가 만든 인덱스와 JOIN 이 다른 SQL 에 많은 영향을 줄 수도 있기 때문이다.
10. WHERE 절을 다시 한번 살펴보자. >, <, = 등의 연산은  $A < B$  의 경우 A 에 인덱스를 만들어 동작하게 된다. 그러나 NULL 값을 가지는 연산인 경우에는  $A \text{ NOT IN (내용 1, 내용 2)}$ ,  $A \neq$  연산식,  $A \text{ LIKE ' \%패턴내용'}$  등을 사용하게 되는데 이때에는 인덱스가 동작하지 않는다. 그러므로, 이러한 연산은 피해야 한다.
11. HAVING 보다는 WHERE 를 적극적으로 사용하는 것이 좋다. 특히 인덱스가 걸려있는 컬럼에는 group by 나 having 절을 사용하지 않는 것이 좋다. 이 경우 인덱스가 동작하지 않기 때문이다. 예를 들어, 'SELECT 필드 A, SUM(필드 B) FROM 테이블 GROUP BY 필드 A HAVING 필드 A=100' 이라는 문장은 'SELECT 필드 A, SUM(필드 B)

FROM 테이블 WHERE 필드 A=100 GROUP BY 필드 A'로 변경하는 것이 낫다. 이렇게 사용하면 WHERE 절에 의하여 먼저 인덱스가 동작하기 때문에 인덱스를 사용할 수 있게 된다.

12. WHERE 절에는 먼저 인덱스가 있는 컬럼을 기술한다. 특히 복합 인덱스인 경우 선행 인덱스를 먼저 사용한다. .
13. 인덱스를 사용한 검색과 보통 검색의 차이를 알아보도록 한다. 테이블의 15% 이상을 검색하는 경우에는 오히려 그냥 검색하는 것이 인덱스를 사용한 검색보다 빠르다는 것을 알 수 있다. 이런 경우일 때에는 SQL 문장을 인덱스를 사용하지 않도록 구성하는 것이 좋다. 꼭, 인덱스를 사용하는 경우가 빠른 것은 아니다. 그 이유는 보통 인덱스를 사용한 검색에서는 검색된 하나의 레코드마다 다중의 논리 연산이 가해지기 때문이다. 그러나, 보통 검색인 경우에는 하나의 논리적인 블록마다 연산되는 모든 행을 사용하기 때문에 테이블에 접근하는 내용이 많은 경우에는 일반적인 보통 검색이 좋다.
14. 인덱스 검색의 경우 ORDER BY 절을 적극 활용하도록 한다. 이 절은 인덱스된 컬럼에 대해 검색을 가능하게 한다.
15. 가장 중요한 것은 사용하는 데이터를 이해하고 있어야 한다는 것이다. 데이터의 분포도의 이해는 다시 한번 말하지만 가장 중요한 점이다. 오라클 7.3 의 경우 이러한 데이터 분포를 이해하기 위한 HOSTOGRAMS 라는 기능이 있다.
16. SQL 문장을 만들 경우에 참조하는 테이블의 수는 작을수록 좋다.
17. 어쩔 수 없이 많은 테이블을 JOIN 하는 경우에는 JOIN 의 순서가 매우 중요하다. 다시 한번 이야기하지만, 적은 수의 행(레코드)를 가진 테이블과 분포도가 풍부한 테이블이 JOIN 의 처음에 기술되어야 한다.
18. SQL 문장은 단순해야 한다. 한 페이지를 넘어가는 SQL 문장은 각 RDBMS 의 SQL 최적화기를 멍청(?)하게 만든다. 가끔은 단순한 것이 복잡한 것보다 빠를 수 있다. 결론은, 각각의 SQL 문장을 테스트 해보아야 한다는 것이다. 임시 테이블의 경우 많은 테이블을 포함하는 경우에는 SQL 의 JOIN 을 나누어서 수행하는 것이 좋다. 복잡한 JOIN 인 경우에는 2~3 개 정도의 SQL 문장이 나올 수도 있다.
19. 하나의 결과를 얻기 위해 여러 형태의 SQL 문장을 기술할 수 있다. 연산자의 사용에 주의해야 하는데, 특히 NOT IN 과 같은 문장은 인덱스가 있다 하더라도 인덱스를 사용하지 않는 보통 검색 작업을 취한다. 그러므로, NOT IN(SELECT)나 NOT EXISTS 보다는 MINUS 산술연산을 사용하는 것이 인덱스를 사용하기 때문에 유리하다.
20. WHERE 절에 OR 보다는 UNION 을 사용하는 것이 좋다.
21. ROWID 나 ROWNUM 을 사용할 수 있다면 적극적으로 사용하는 것이 좋다. 다만 이 값들은 언제나 같지 않기 때문에 이용할 때에는 ROWID 의 값을 리턴하지 않는 것이 좋다. 행은 ROWNUM 을 사용하는데, WHERE 절에 ROWNUM < 100 을 사용하면 100 개 이상의 행을 전달하지 않는다.
22. CURSOR 의 사용에 주의하라. 함축적으로 만들어진 CURSOR 는 여분의 fetch 를 만들

어 낸다. DECLARE, OPEN, FETCH, CLOSE 문을 사용하면 개발자가 명시적으로 만드는 CURSOR 을 사용한다. 보통 DELETE, UPDATE, INSERT, SELECT 문을 사용하면 함축적인 CURSOR 을 사용하게 된다.

23. RDBMS 에 병렬 쿼리 옵션이 있다면 적극적으로 활용하라. 이 병렬 쿼리를 사용하면 보다 빠르게 SQL 문을 수행한다. 오라클 7 의 경우 전체 검색의 경우에만 병렬 쿼리가 동작한다. 그렇지만, 오라클 8 에서는 인덱스가 분할되어 있다면 indexed range 검색을 통한 쿼리도 병렬로 동작한다. 다만, 이 옵션은 다수의 디스크 드라이버나 SMP, MPP 시스템에서만 사용할 수 있다.
24. 네트워크 트래픽에 주의하라! 특히 한번에 처리되는 작업을 크게 하는 것이 빈번한 작업보다는 훨씬 좋다. 오라클의 경우 array processing 과 PL/SQL block 을 사용하면 SQL 문 하나로 많은 row 를 수행한다. INSERT 시에 배열을 사용하면 훨씬 빠른 작업을 할 수 있다.
25. 복잡한 SQL 문은 많은 네트워크 트래픽을 유도할 수 있다. 오라클의 경우 PL/SQL 의 블록내부에 존재한다면 전체 블록이 오라클 서버로 보내져 한번에 수행되고 결과를 빠르게 클라이언트에 전달합니다.

## 새롭게 변한 SQL 빌더의 사용

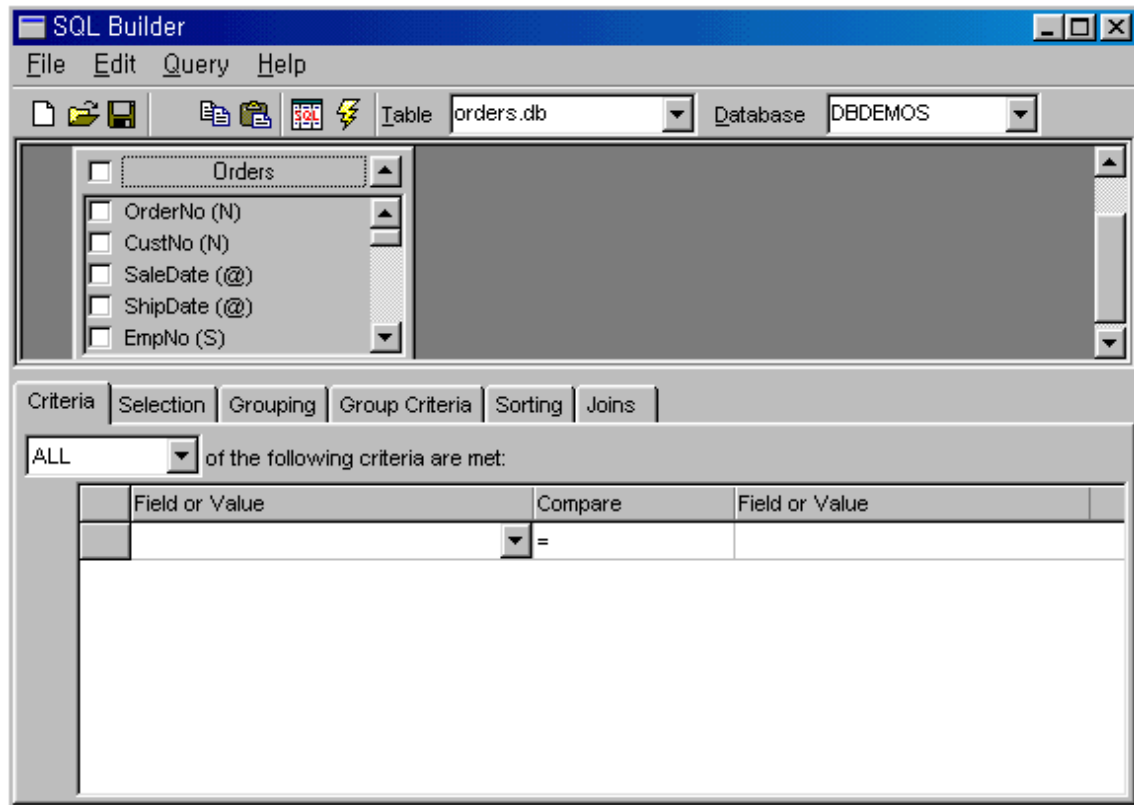
텔파이 4 에서는 텔파이 3 까지의 비주얼 쿼리 빌더(Visual Query Builder)가 한층 업그레이드된 SQL 빌더로 대체되었다. SQL 빌더를 처음 동작시키려면 디자인 시에 폼이나 데이터 모듈에 놓인 TQuery 컴포넌트를 선택하고 오른쪽 버튼을 클릭하면 컨텍스트 메뉴가 뜬다. 여기에서 필드 에디터와 SQL 탐색기, 그리고 SQL 빌더 등을 실행시킬 수 있다.

SQL 빌더를 이용해 시각적으로 대화식의 SQL 쿼리를 생성, 실행할 수 있다. 아주 복잡한 SQL 쿼리도 전문적인 지식 없이 간단하게 작성할 수 있는 것이 SQL 빌더의 장점이다. 심지어는 SQL 구문을 배우기 위한 하나의 도구로 사용할 수도 있을 정도이다. SQL 빌더를 이용해 SQL 쿼리의 결과를 보고, 이를 편집할 수 있다. SQL 빌더가 지원하는 쿼리는 기본적인 select 구문부터 계산 필드가 포함된 복잡한 다중 테이블의 join 을 비롯한 모든 구문을 만들어낼 수 있다.

또한, SQL 빌더를 이용하면 이중의 쿼리들을 사용할 수 있다. 즉, 서로 상이한 데이터베이스에 포함된 테이블들에 대한 쿼리가 가능하다는 말이다. 심지어 데이터베이스 서버가 다른 경우에도 작동한다. 이를 구현하기 위해서, 상이한 데이터베이스에 접근할 때 BDE 가 SQL 구문을 서버로 보내지 않고 로컬 SQL 로 처리하게 된다. 로컬 SQL 은 원래 dBASE 와 파라독스를 위해 지원되는 것인데, 쿼리의 해석과 데이터를 읽어오는 작업을 서버에 맡기지 않고 BDE 내부에서 처리하는 것이다.

- SQL 쿼리 텍스트 진입 윈도우 (SQL Query Text Entry Window)

SQL 쿼리 텍스트 진입 윈도우는 SQL 쿼리 구문을 텍스트로 입력하고 볼 수 있도록 지원되는 윈도우를 말한다. SQL 빌더에서 쿼리를 그래픽 인터페이스로 생성하면 이 윈도우를 통해 SQL 구문이 나타난다. 즉, SQL 빌더와 이 텍스트 입력 윈도우는 서로 연결되어 있다. 만약 텍스트 입력 윈도우에서 쿼리를 변경하면 그것이 SQL 빌더에도 나타난다. 이들을 서로 토글하기 위해서는 상단의 툴바 중에서 SQL 버튼을 클릭하면 된다.



- 쿼리 실행과 쿼리 결과 윈도우(Query Result Window)

F9 를 누르거나 툴바에서 번개 표시가 되어 있는 버튼을 누르면 쿼리가 실행된다.

이와 동시에 쿼리 결과 윈도우(Query Result Window)가 뜨고 그 결과가 그리드에 나타난다. 쿼리 결과 윈도우는 그리드 하나와 DBNavigator 하나로 구성된다.

- 테이블 pane

테이블 pane 은 SQL 구문에 사용된 각 테이블과 그 필드들을 표시하는 판이다. 특정 테이블을 불러오려면 SQL 빌더의 툴바에 나타난 테이블 드롭 다운 박스에서 한 테이블을 선택하면 된다. 물론 그에 앞서 Database 드롭 다운 박스에서 데이터베이스 앨리어스부터 먼

저 선택해야 한다.

테이블 pane 에 올려진 테이블을 변경하는 방법은 오른쪽 버튼을 클릭한 뒤, Edit Table Alias 메뉴를 선택하면 된다. 이렇게 하면, 테이블 이름이 편집가능 상태로 변하는데 여기에서 새로운 테이블 이름을 입력하면 된다.

테이블 왼쪽에 체크박스가 있는데, 이를 통해 어떤 필드를 쿼리의 select 문에 포함시키고, 어떤 필드를 포함시키지 않을 지를 지정할 수 있다. 과란색으로 체크된 필드만 쿼리 구문에 나타난다. 상단에 나타나는 테이블 이름에도 체크 박스가 있는데, 여기에 체크가 될 때에는 테이블 내의 모든 필드를 포함시킨다는 말이다. 이것을 끄면 기본적으로 모든 필드의 체크가 꺼진다. 이 상태에서 필요한 필드만 체크할 수 있다. 필요한 필드만 체크된 경우에는 테이블 이름의 체크박스에는 회색 체크가 나타난다.

테이블 pane 에서는 Join 된 필드끼리의 연결 상태도 보여주는데, 테이블 이름 사이의 가는 선이 연결 관계를 보여준다.

우상단에 있는 Minimize 버튼을 클릭하면 테이블 이름만 나타나고 필드들은 표시되지 않는다. 테이블끼리 조인 시키는 방법은 한 테이블의 필드를 드래그해서 다른 테이블의 필드 위에서 마우스 버튼을 떼는 것이다. 조인이 이루어지면 테이블끼리 굵은 선이 그어지며, from 절 안에 join 이 추가된다. 다음의 구문이 join 이 추가된 예이다.

```
select ... from INNER JOIN "PostCodeMaster.DB" PostcodeMaster
ON (PostcodeMaster.PostCode = Customermaster.PostCode),
"CustomerClass.DB" Customerclass
```

테이블 pane 은 SQL 빌더의 하단에 있는 Selection 탭과 연결해서 동작한다. 테이블 pane 에서 변경된 사항은 Selection 탭에 바로 적용되고, Selection 탭에서 변경된 사항은 테이블 pane 에 바로 적용된다.

#### ● Where 절을 위한 Criteria 페이지

Criteria 페이지는 쿼리 결과 중에서 특정 Row 들만을 가져오기 위하여 where 절을 만드는 페이지이다. 여기에서의 결과를 Where 절에 추가시킨다. Join 문법과는 다르다는 것을 명심하자. 단지 Where 절을 위한 지원이다.

```
select ... from ...
where TableA.Field1 = TableB.Field1
```

여기에서 TableA.Field1 = TableB.Field1 과 같은 문장을 만들어내는 것이 Criteria 페이지의 역할이다. 이런 관계를 필요한 만큼 설정할 수 있다.

이 페이지의 상단에 보면 XXX of the following criteria met: 이라는 부분이 보인다. XXX 는 드롭 다운 콤보박스이다. 콤보박스에는 ALL, ANY, NONE, NOT ALL 중 하나를 선택할 수 있다. 'X = Y' 같은 criterion 이 A, B, C, ... 라고 가정하자. 이때, 각각의 의미는 다음과 같다.

ALL : A and B and C and ...

ANY : A or B or C or ...

NONE : not(A) or B or C or ...

NOTALL : not(A) and B and C and ...

criteria 는 표현식이나 EXISTS 절이 될 수 있다.

- Select 절을 위한 Selection 페이지

어떤 필드를 보여줄 것인가와 그 필드명이 어떻게 출력될 것인가를 지정해준다. 예를 들어,

```
select A, B, C, ... from ...
```

여기에서 A, B, C, ... 의 내용을 채우는 페이지다. Output Name 컬럼에 해당 필드의 출력명을 줄 수 있는데 한글도 사용할 수 있다. 이를 사용하면 DBGrid 등에서 한글 필드명을 나타낼 수 있다. 필드의 추가는 드롭다운을 통해서 선택해도 되고 테이블 pane 에서 해당 필드를 드래그해서 빈 Row 에 놓으면 추가가 이루어진다. 또한 테이블 pane 에서 해당 필드를 드래그해 기존의 Row 에 놓으면 그 내용이 변경된다. 테이블 pane 과 Selection 페이지는 연결되어 있기 때문에, 한쪽의 내용을 변경하면 다른 쪽의 내용에 곧바로 적용된다. 상단에 Remove Duplicates 체크박스를 볼 수 있는데 이것을 클릭하면 동일한 Row 는 쿼리 결과에 나타나지 않는다. 즉, 이것을 SQL 문으로 표현하면,

```
select distinct A, B, C, ... from ...
```

과 같이 select 절에 distinct 키워드가 추가된다.

선택된 필드를 지우고자 하면, 오른쪽 버튼을 클릭한 뒤 Delete Row 메뉴를 선택하면 된다. 함수를 이용하고 싶은 경우에는 Field 컬럼에서 해당 필드를 함수로 묶어주면 된다.

예를 들어,

```
Sum(CustomerMaster.Count)
```

이렇게 하면 Summary 컬럼이 새로 생기면서,

SUM CustomerMaster.Count

와 같은 형태로 바뀐다. 직접 해보기 바란다.

- Group by 절을 위한 Grouping 페이지

group by 는 쿼리 결과에서 특정 그룹의 Row 들을 묶어 하나의 summary Row 를 얻기 위한 SQL 키워드이다. 좌측의 Output Fields 리스트 박스에는 select 절에서 지정된 모든 필드가 올라온다. 이 필드들 중에서 하나를 선택하여 Add 버튼을 누르면 Grouped On 리스트 박스로 옮겨간다. 반대로 Remove 버튼을 누르면 Grouped On 리스트 박스에서 Output Fields 리스트 박스로 옮겨간다.

select .. from ... where ... group by A, B, C, ...

Grouped On 에 추가된 필드는 A, B, C, ... 과 같이 group by 절에 나타난다. 이를 통해 A, B, C, ... 필드에 동일한 값을 가진 Row 는 하나의 Row 로 쿼리 결과에 나타나게 된다. 여기서 주의할 것은 문법상 select 절에는 sum 등의 함수나 group by 절에 이용된 필드만 올 수 있다. 다른 필드의 값은 그룹화된 Row 들마다 각기 다를 수가 있고 그것을 하나의 Row 로 표현하는 것은 불가능하기 때문이다.

- Having 절을 위한 Grouping criteria 페이지

Having 절은 한마디로 어떤 그룹을 선택할 것인지를 지정하는 절이다. 해당 조건을 만족하는 그룹만 group by 를 위해 사용된다. SUM, COUNT 등의 함수를 이용하거나 그냥 SQL 표현식을 사용할 수도 있다. 어떤 형태가 이용될 것인지는 오른쪽 버튼을 클릭하여 SQL Expression, Simple Having Summary Expression, Two Summary Expression 중에서 하나를 선택하는 것으로 지정할 수 있다.

SQL Expression 은 SQL 표현식을 직접 기입하는 것이다. 즉, 다음의 예와 같다.

SUM (Qty \* Price) > 1000

Simple Having Summary Expression 은 두 필드를 비교해 요약하는 것이다. 한 개의 함수와 비교할 두 개의 필드, 비교할 연산자를 선택한다.

Two Summary Expression 은 두 개의 summary 표현식을 비교하는 것이다. 두 개의 함수



와 두 개의 필드 그리고 연산자를 선택한다.

여기서도 Selection 페이지와 마찬가지로 ALL, ANY, NONE, NONE, NOT ALL 중 하나를 선택할 수 있다.

- Order by 절을 위한 Sorting 페이지

Order by 절에 사용될 필드들을 선택하는 페이지이다. Output Fields 에는 select 절에 포함된 필드 명들이 리스트 박스로 나타난다. Add 버튼을 누르면 Sorted By 리스트 박스로 옮겨지며, 반대로 Remove 버튼을 누르면 Sorted By 리스트 박스에서 지워진다.

각 필드 별로 정순정렬과 역순정렬을 지정할 수 있는데, A..Z 를 선택하면 정순, Z..A 를 선택하면 역순정렬이 이루어진다. 만들어진 쿼리의 결과는 다음과 같다.

```
select ... from ... where ... order by A desc, B, C, ...
```

위에서 order by A desc, B, C, ... 부분이 Sorting 페이지를 통해서 추가되는 부분이다. Z..A 를 선택하여, 역순정렬이 사용된 필드이다.

- From 절에서의 join 을 위한 Joins 페이지

다중 테이블 SQL 쿼리를 위해 이용되는 페이지이다. 2 개의 필드와 그 필드의 비교 연산자를 통해 SQL 쿼리에 join 을 추가한다.

상단에 각 필드별로 한 개 씩의 Include Unmatched Records 를 볼 수 있는데 이것이 체크되면 outer join 이 이루어지고, 체크되지 않으면 inner join 이 이루어진다. 첫 번째 것만 체크되면 left outer join 이 쿼리에 추가되고, 두 번째 것만 체크되면 right outer join 이 쿼리에 추가된다. 둘 다 체크된 경우 full outer join 이 쿼리에 추가된다. 아무것도 체크되지 않으면 inner join 으로 처리된다.

SQL1 표준은 outer join 을 지원하지 않는다. 그러므로, 몇몇 구형 SQL 서버들은 outer join 이라는 개념이 아예 없는 경우가 있으므로 주의해야 한다.

테이블 pane 에서 필드를 드래그하여 다른 테이블 pane 의 특정 필드 위에 올려 놓으면 join 이 자동으로 추가된다. 이 페이지에서 새로운 join 을 추가하는 방법은 드롭-다운 콤보 박스 중 <create a new join>을 선택 한 뒤에 필드와 연산자를 선택하면 된다.

Join 을 없애는 방법은 그리드에서 오른쪽 버튼을 눌러 Delete Row 를 Row 가 모두 될 때까지 하면 자동으로 없어진다.

## 유용한 SQL 문장들

여기에 소개하는 몇 가지 SQL 문장 사용에 대한 팁들은 Inprise 에서 제공하고 있는 TI 들을 참고한 것임을 미리 밝혀 둔다.

- 계산된 컬럼의 정렬 (Sorting on a Calculated Column)

데이터 세트의 계산 결과에 대해서, 이를 바탕으로 정렬이 필요한 경우가 있다. 텔과이 어플리케이션에서 SQL 을 사용하는 경우에 어떻게 하면 되는지 간단히 소개한다.

파라독스나 디베이스와 같은 로컬 SQL 의 경우에는 계산 필드(calculated field)가 AS 키워드를 사용한다. 이를 이용하면 계산 필드가 ORDER BY 와 같은 문장을 이용해서 이 필드를 키로 정렬하도록 설정할 수가 있다. 예를 들어, 다음의 SQL 문장을 살펴 보자.

```
SELECT I."PARTNO", I."QTY", (I."QTY" * 100) AS TOTAL  
FROM "ITEMS.DB" I ORDER BY TOTAL
```

이 문장에 의해 계산 필드는 TOTAL 로 명명되며, 이 컬럼의 이름을 이용하여 ORDER BY 구문을 사용하면 정렬이 가능하다.

그런데, Interbase 같은 경우 이런 방법이 적용되지 않는데 이는 이들이 계산 필드를 사용할 때, 이름으로 적용하지 않고 필드의 위치를 이용하기 때문이다. 예를 들어, 다음의 SQL 문장은 월급 순으로 EMPLOYEE 테이블을 정렬하게 된다.

```
SELECT EMP_NO, SALARY, (SALARY / 12) AS MONTHLY  
FROM EMPLOYEE ORDER BY 3
```

이 방법은 로컬 데이터베이스에서도 적용된다. 그러므로, ORDER BY 구문 뒤에 처음부터 필드의 위치를 지정하면 이들을 서로 적용하기가 쉬워질 것이다.

- SUBSTRING 함수의 사용법

SQL 함수인 SUBSTRING 은 대단히 유용하게 사용할 수 있는 함수이다. 많은 수의 SQL 서버와 로컬 데이터베이스에서 지원하지만, Interbase 에서는 지원하지 않는다. 문법은 다음과 같다.

```
SUBSTRING(<column> FROM <start> [, FOR <length>])
```

<column>은 문자열을 추출할 컬럼을 지정하는 것이며, <start>에는 문자를 추출할 시작점을 <length>는 추출할 문자열의 길이를 지정한다.

예를 들어, 다음과 같은 문장은 COMPANY 라는 컬럼의 2 번째 문자에서부터 3 문자를 가져온다.

```
SUBSTRING(COMPANY FROM 2 FOR 3)
```

SUBSTRING 함수는 SELECT 문에 의해 필드 리스트를 지정하는 부분이나, WHERE 절에서 값을 비교할 때 유용하게 사용할 수 있다. 당연한 이야기이지만 SUBSTRING 함수는 문자열 형의 컬럼에서만 사용할 수 있다.

다음 문장은 COMPANY 컬럼에서 앞의 3 자를 추출해서 SS 라는 계산 컬럼(calculated column)으로 지정하게 된다.

```
SELECT (SUBSTRING(C."COMPANY" FROM 1 FOR 3)) AS SS  
FROM "CUSTOMER.DB" C
```

다음 문장은 COMPANY 컬럼의 2~3 번째 문자가 'an'인 모든 레코드를 뽑는다.

```
SELECT C."COMPANY" FROM "CUSTOMER.DB" C  
WHERE SUBSTRING(C."COMPANY" FROM 2 FOR 2) = "an"
```

Interbase 의 경우, SUBSTRING 함수와 비슷한 기능을 하는 SQL 문장을 작성하기 위해서는, LIKE 연산자를 사용할 수 있다. 다음 문장은 EMPLOYEE 테이블의 LAST\_NAME 필드의 값의 2~3 번째 문자가 "an"인 모든 레코드를 뽑는다.

```
SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEE  
WHERE LAST_NAME LIKE "_an%"
```

## 파라미터가 있는 질의의 작성

쿼리 문장에 변화를 줄 수 있도록 하기 위해서는 파라미터를 사용할 수 있는 SQL 문장을 이용해야 한다. 다음의 SQL 문장을 살펴 보자.

```
SELECT TEST."FNAME", TEST."Salary of Employee" FROM TEST  
WHERE TEST."Salary of Employee" > :Val
```

이 문장에서 변수 이름으로 Val 을 설정한 것이다. 이제는 TQuery 객체의 Params 프로퍼티를 이용해서 이들 변수의 내용에 접근할 수 있다. 이 변수의 내용을 TEdit 박스를 이용

해서 입력하게 하려면 다음과 같이 하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    with Query1 do
    begin
        Close;
        ParamByName('Val').AsInteger := StrToInt(Edit1.Text);
        Open;
    end;
end;
```

LIKE 문을 같이 사용하면 더욱 유용하게 사용할 수 있다. 예를 들어 다음과 같은 SQL 문장이 있다고 하자.

```
SELECT * FROM CUSTOMER
WHERE Company LIKE :CompanyName
```

이 문장에서 CompanyName 이 파라미터 변수가 된다. 이를 이용하여 다음과 같이 코딩을 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    with Query1 do
    begin
        Close;
        ParamByName('CompanyName').AsString := Edit1.Text + '%';
        Open;
    end;
end;
```

참고로, ParamByName 을 사용하여 파라미터 변수 이름을 쓰는 것은, Params[Parameter Number]를 사용하는 것과 동일하다. 예를 들어, 앞의 코드에서 CompanyName 이 첫번째 파라미터라면 다음과 같이 쓸 수도 있다.

```
Params[0].AsString := Edit1.Text + '%';
```

## 캐쉬 업데이트(Cached update)의 활용

캐쉬 업데이트는 기본적으로 트랜잭션에 걸리는 시간과 네트워크 트래픽을 줄이려는 목적으로 사용된다. 이런 효과가 있지만, 모든 데이터베이스 어플리케이션에 적용하는 것은 좋지 않다. 캐쉬 업데이트의 사용을 결정할 때 고려해야할 사항은 크게 3 가지가 있다.

1. 로컬에 복사된 데이터를 편집하는 동안 다른 어플리케이션이 서버의 실제 데이터에 접근하여 데이터를 변경할 수 있다.
2. 로컬에 복사된 데이터의 편집된 사항이 서버에 적용되기 전에는 다른 어플리케이션이 변화된 사항을 알 수 없다.
3. 쿼리에 기초한 데이터 세트에 캐쉬 업데이트를 적용할 때에는 업데이트 객체를 필요로 한다.

### ● 캐쉬 업데이트 과정의 이해

캐쉬 업데이트를 수행하는 동안 실제로 다음과 같은 과정을 거치게 된다.

1. 캐쉬 업데이트를 가능하게 한다. 이렇게 하면 서버의 데이터를 읽기전용으로 가져오는 트랜잭션이 시작된다. 이 데이터의 로컬 복사본이 메모리에 저장되며 데이터 컨트롤에 디스플레이되고 편집이 가능해진다.
2. 레코드의 로컬 복사본이 편집된다. 이 때 원본 레코드와 편집된 동작이 메모리에 기록된다.
3. 사용자가 레코드를 스크롤할 때마다 읽기전용 트랜잭션을 이용해 필요한 레코드를 추가로 가져온다.
4. 편집된 레코드를 데이터베이스에 적용하거나 변화를 취소한다. 데이터베이스에 레코드가 기록될 때마다 OnUpdateRecord 이벤트가 발생한다. 이 때 에러가 발생하면 OnUpdateError 이벤트가 발생하여 에러를 수정하고, 업데이트를 계속할 수 있게 하는 방법을 제공한다. 업데이트가 끝나면 로컬 캐쉬에 저장되어 있던 변화사항이 제거된다.

### ● 업데이트 컴포넌트에 대한 SQL 문장 제작

업데이트 컴포넌트에 대한 SQL 문장을 제작하려면, 먼저 TUpdateSQL 컴포넌트를 데이터 모듈이나 폼에 위치시키고, 업데이트 컴포넌트의 이름을 데이터 세트의 UpdateObject 프로퍼티에 기록한다. 그리고, 업데이트 컴포넌트를 더블 클릭하면 업데이트 SQL 에디터를 띄울 수 있는데, 이 에디터에서 SQL 문장을 편집하면 된다.

업데이트 SQL 에디터에는 2 개의 페이지가 있다. 처음 에디터를 띄우면 Options 페이지를 볼 수 있다. Table Name 콤보 박스에서 업데이트할 테이블을 고른 후, Update Fields 리스트 박스에서 업데이트할 컬럼을 선택한다. 또한, Key Fields 리스트 박스에서는 업데이트시 키로 사용할 컬럼을 고른다. 파라독스, 디베이스, 폭스 프로의 경우에는 선택한 컬럼이 반드시 인덱스가 존재하는 컬럼이어야 한다. 만약 데이터베이스 서버가 필드 이름에 따옴표를 요구하면 Quote Field Names 체크 박스를 선택한다. 그리고 나서 Generate SQL 을 클릭하면 업데이트 컴포넌트의 ModifySQL, InsertSQL, DeleteSQL 프로퍼티와 연관된 SQL 문장이 생성된다.

만들어진 SQL 문장을 보고, 편집하려면 SQL 페이지를 선택한다.

### ● 이미 존재하는 레코드의 업데이트

이미 존재하는 레코드를 업데이트할 때에는 업데이트 객체의 ModifySQL 프로퍼티를 이용하여 SQL UPDATE 구문을 생성한다. 이 때 UpdateMode 프로퍼티를 설정하여 델과가 업데이트할 레코드를 찾는 방법을 지정할 수 있다. 다음 테이블에 UpdateMode 프로퍼티에 가능한 값을 나열하였다.

값	의 미
upWhereAll(디폴트)	UPDATE 문장의 WHERE 절에 지정된 값에 정확하게 해당되는 레코드만 업데이트 한다.
upWhereKeyOnly	지정된 키 컬럼에 기초한 레코드를 업데이트 한다.
upWhereChange	키 컬럼과 수정된 컬럼들만 업데이트 한다.

### ● 캐쉬 업데이트의 적용

데이터 세트가 캐쉬 업데이트 모드에 있으면 데이터의 변화가 어플리케이션에서 메소드를 호출하지 않으면 데이터베이스에 기록되지 않는다.

#### 참고:

SQL 쿼리에 의해 가져온 레코드 세트는 라이브 결과 세트(live result set)를 지원하지 않는다. 이 때 업데이트를 하려면 반드시 TUpdateSQL 객체를 사용해야 한다. 업데이트에 테이블의 결합(join)이 포함되면 반드시 각각의 테이블당 하나의 TUpdateSQL 객체를 지원해야 하며, OnUpdateRecord 이벤트를 핸들러를 이용해 이 객체들이 업데이트할 수 있도록 해야 한다.

### ● 데이터 세트 컴포넌트 메소드를 사용한 캐쉬 업데이트의 적용

각각의 데이터 세트에서 ApplyUpdates 와 CommitUpdates 메소드를 사용하여 업데이트가 가능하다. 이렇게 데이터 세트 레벨에서 직접 캐쉬 업데이트를 하면 데이터베이스 트랜잭션을 이용해야 한다. 다음의 코드는 CustomerQuery 데이터 세트의 업데이트를 적용하는 예이다.

```
procedure ApplyButtonClick(Sender: TObject);
begin
    with CustomerQuery do
    begin
        Database1.StartTransaction;
        try
            if not IsSQLBased and (TransIsolation <> tiDirtyRead) then //파라독스, 디베이스 ... ?}
                TransIsolation := tiDirtyRead; //그렇다면 TransIsolation 은 tiDirtyRead!
            ApplyUpdates; //데이터베이스에 업데이트 기록을 시도
            Database1.Commit; //성공하면 변화를 commit!
        except
            Database1.Rollback; //실패하면 롤백!
            raise; //예외 발생으로 CommitUpdates 의 호출을 막는다.
        end;
        CommitUpdates; //성공하면 내부 캐쉬의 내용을 지운다.
    end;
end;
```

ApplyUpdates 를 호출할 때 예외가 발생하면 데이터베이스 트랜잭션은 롤백된다. 위에서 raise 문장을 사용하는 것은 뒤에 나오는 CommitUpdates 문장의 호출을 막기 위한 것인데, CommitUpdates 가 호출되지 않으면 업데이트의 내부 캐쉬가 지워지지 않기 때문에 여러 상황에 대한 조작을 하고 업데이트를 다시 시도할 수 있다.

- 데이터베이스 컴포넌트 메소드를 이용한 캐쉬 업데이트

하나 이상의 데이터 세트에 대해 데이터베이스 컴포넌트의 ApplyUpdates 메소드를 이용해 캐쉬 업데이트의 적용이 가능하다. 다음 코드는 버튼 클릭시 업데이트 내용을 CustomersQuery 데이터 세트에 적용한다.

```
procedure ApplyButtonClick(Sender: TObject);
begin
```

```

if not IsSQLBased and (TransIsolation <> tiDirtyRead) then
    TransIsolation := tiDirtyRead;
    Database1.ApplyUpdates([CustomersQuery]);
end;

```

이 메소드는 트랜잭션을 시작하고, 캐쉬 업데이트 내용을 데이터베이스에 기록한다. 성공적이면 트랜잭션을 commit 하고, 로컬 캐쉬의 업데이트 내용을 지운다. 성공적이지 못하면 롤백하고 캐쉬 업데이트의 내용을 변화시키지 않는다. 이 때 데이터 세트의 OnUpdateError 이벤트를 통해 에러 처리를 할 수 있다.

데이터베이스 컴포넌트의 ApplyUpdates 메소드는 데이터베이스와 연관된 여러 데이터 세트 컴포넌트를 업데이트할 수 있는 잇점이 있다. 다음 코드는 마스터/디테일 형의 2 개의 테이블을 업데이트 한다.

```

if not IsSQLBased and (TransIsolation <> tiDirtyRead) then
    TransIsolation := tiDirtyRead;
    Database1.ApplyUpdates([MasterTable, DetailTable]);

```

#### ● 마스터/디테일 테이블의 업데이트

마스터/디테일 테이블에 업데이트를 적용할 때에는 삭제된 레코드를 다룰 때만 제외하고는 먼저 마스터 테이블을 업데이트하고 디테일 테이블을 업데이트 한다. 복잡한 마스터/디테일 관계를 가지고 있는 경우에도 기본적인 원칙은 같다.

마스터/디테일 테이블을 업데이트할 때 데이터 세트와 데이터베이스 컴포넌트 레벨에서 업데이트하는 방법이 있다. 이때 보다 명확하게 조절을 하기 위해서는 데이터 세트 레벨에서 업데이트하는 것이 좋다. 다음 코드는 2 개의 마스터/디테일 테이블을 캐쉬 업데이트하는 예이다.

```

Database1.StartTransaction;
try
    Master.ApplyUpdates;
    Detail.ApplyUpdates;
    Database1.Commit;
except
    Database1.Rollback;
    raise;
end;

```



Master.CommitUpdates;

Detail.CommitUpdates;

## ● 캐쉬 업데이트의 취소

캐쉬 업데이트를 취소하는 방법은 다음의 3 가지가 있다.

1. 모든 업데이트를 취소하고, 캐쉬 업데이트를 사용하지 않으려면 `CachedUpdates` 프로퍼티를 `False` 로 설정한다.
2. 캐쉬 업데이트를 계속 사용하되, 지금까지의 업데이트를 취소하려면 `CancelUpdates` 메소드를 호출한다.
3. 현재 레코드에 가해진 업데이트를 취소하려면 `RevertRecord` 메소드를 호출한다.

## ● 캐쉬된 레코드의 되살리기

삭제된 레코드는 일단 캐쉬에 저장되기 때문에 이를 되살리는 방법이 있다. 이렇게 하려면 일단 `UpdateRecordTypes` 프로퍼티를 삭제된 레코드를 가리키도록 설정하고 `RevertRecord` 를 호출하면 된다. 다음 코드는 삭제된 레코드를 모두 되살린다.

```
procedure UndeleteAll(DataSet: TDataSet);
begin
    with DataSet do
    begin
        UpdateRecordTypes := [rtDeleted];
        try
            First;
            while not EOF do
                RevertRecord;
        finally
            UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
        end;
    end;
end;
```

## ● 업데이트 상태의 검사

업데이트 상태의 검사는 OnUpdateRecord 와 OnUpdateError 이벤트에서 가장 빈번하게 사용하게 된다. 이때 UpdateStatus 프로퍼티를 사용하게 되는데, 그 값은 다음과 같다.

값	의 미
usUnmodified	레코드의 변화가 없음
usModified	레코드가 수정 되었음
usInserted	새로운 레코드
usDeleted	레코드가 삭제 되었음

다음 코드는 레코드가 수정된 경우 UpdateStatus 프로퍼티를 이용하여 OnCalcFields 이벤트 핸들러에서 필드에 별표('\*')를 보여주는 예제이다.

```
procedure TForm.CalcFields(DataSet: TDataSet);
begin
    if DataSet.UpdateStatus <> usUnmodified then
        Table1ModifiedRow.Value := '*'
    else
        Table1ModifiedRow.Value := Null;
end;
```

주의: 레코드의 UpdateStatus 가 usModified 이면 데이터 세트의 각 필드의 OldValue 프로퍼티를 이용하여 이전 값을 알아낼 수 있다.

### ● 필드의 OldValue, NewValue 프로퍼티의 사용

캐쉬 업데이트를 사용할 때 각 레코드의 원래 값은 TField 객체의 읽기전용 프로퍼티인 OldValue 에 담겨 있다. 바뀐 값은 TField 의 NewValue 프로퍼티에 저장된다. 이 값들은 OnUpdateError 와 OnUpdateRecord 이벤트 핸들러에서 유용하게 쓰일 수 있다.

이 값들을 이용해 에러를 일으킨 원인을 알아내고 이를 수정할 수 있다. 다음 코드는 월급 (salary) 필드를 한번에 25% 이상 인상할 수 없도록 제한한다.

```
var
    SalaryDif: Integer;
    OldSalary: Integer;
begin
    OldSalary := EmpTabSalary.OldValue;
```

```

SalaryDif := EmpTabSalary.NewValue - OldSalary;
if SalaryDif / OldSalary > 0.25 then
begin
    {인상 폭이 너무 크면 25% 까지 감소시킨다.}
    EmpTabSalary.NewValue := OldSalary + OldSalary * 0.25;
    UpdateAction := uaRetry;
end
else
    UpdateAction := uaSkip;
end;

```

앞의 예에서 NewValue 는 25% 인상으로 제한되고 이 값을 가지고 업데이트를 다시하게 된다.

#### ● OnUpdateRecord 이벤트 핸들러의 제작

데이터 세트의 OnUpdateRecord 이벤트 핸들러의 기본 골격은 다음과 같다.

```

procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
    UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    {업데이트 코드...}
end;

```

DataSet 파라미터는 업데이트할 데이터 세트를 가리키며, UpdateKind 파라미터는 업데이트의 type, UpdateAction 파라미터는 업데이트할 것인지를 결정한다. 디폴트 값은 uaFail 이다. 업데이트하는 동안 문제가 없으면 이벤트 핸들러에서 빠져 나가기 전에 이 값을 uaApplied 로 설정하고, 특정 레코드를 업데이트 하지 않으려면 uaSkip 으로 설정하면 캐쉬에 적용되지 않은 변화가 그대로 남아있게 된다. 이 파라미터들 외에 필드 컴포넌트의 OldValue 와 NewValue 프로퍼티가 유용하게 쓰일 수 있다.

주의할 점은, OnUpdateRecord 이벤트에서는 OnUpdateError, OnCalcFields 이벤트 핸들러와 마찬가지로 현재 레코드를 변화시키는 메소드를 호출하면 안된다는 것이다.

OnUpdateRecord 는 필요조건은 아니지만 보통 하나 이상의 업데이트 컴포넌트를 사용한다. 다음의 코드는 업데이트를 위해 테이블 컴포넌트를 이용한다.

```

procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;

```

```

UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    if UpdateKind = ukInsert then
        UpdateTable.AppendRecord([DataSet.Fields[0].NewValue,
DataSet.Fields[1].NewValue])
    else
        if UpdateTable.Locate('KeyField', DataSet.Fields[0].OldValue, []) then
            case UpdateKind of
                ukModify:
                    begin
                        Edit:
                            UpdateTable.Fields[1].Value := DataSet.Fields[1].Value;
                        Post:
                            end;
                ukDelete: DeleteRecord;
            end;
        UpdateAction := uaApplied;
    end;
end;

```

보통은 OnUpdateRecord 이벤트 핸들러에서 2 개 이상의 업데이트 컴포넌트를 사용한다.

```

procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    EmployeeUpdateSQL.Apply(UpdateKind);
    JobUpdateSQL.Appply(UpdateKind);
    UpdateAction := uaApplied;
end;

```

주의:

만약 데이터 세트의 UpdateObject 프로퍼티에 업데이트 컴포넌트를 지정하고, OnUpdateRecord 이벤트 핸들러를 제작하면 지정된 업데이트 컴포넌트는 이벤트 핸들러에서 Apply 메소드를 호출하지 않으면 동작하지 않는다.

- 캐쉬 업데이트 에러 처리

레코드가 처음 캐쉬되어 캐쉬 업데이트를 적용하려고 할 때 시간 차이가 있기 때문에 그동안 다른 어플리케이션에서 데이터베이스의 레코드가 바뀌었을 수가 있다. 이 때 BDE 는 업데이트를 적용할 때 이를 에러로 처리한다.

데이터 세트 컴포넌트의 OnUpdateError 이벤트는 이런 에러를 포함한 에러 처리를 가능하게 해준다. 만약 이 이벤트 핸들러를 작성하지 않으면 업데이트는 실패한다.

이때 주의할 점은 현재 레코드를 변화시키는 메소드(Next, Prior 등)을 호출하면 무한 루프에 빠지게 된다는 점이다.

OnUpdateError 이벤트 핸들러의 코드 골격은 다음과 같다.

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
    UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    {업데이트 코드를 적는다.}
end;
```

#### ● 에러 메시지의 처리

E 파라미터는 보통 EDBEngineError type 이다. 에러 처리에서 여기서 에러 메시지를 알아낼 수 있다. 다음 코드는 라벨에 에러 메시지를 보여준다.

```
ErrorLabel.Caption := E.Message;
```

이 파라미터는 업데이트 에러의 원인을 알아내는 데에도 유용하다. 다음의 코드는 업데이트 에러가 키 위반(key violation)에 의한 것인지 알아보고 UpdateAction 파라미터를 uaSkip 으로 설정한다.

```
{실제 사용시 uses 절에 'Bde'를 추가해야 함}
if E is EDBEngineError then
    with EDBEngineError(E) do
    begin
        if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
            UpdateAction := uaSkip {키 위반이면 이 레코드를 건너 뛴다.}
        else
            UpdateAction := uaAbort {원인을 모르므로 업데이트를 취소 !}
    end;
```

## 정 리 (Summary)

이번 장에서는 클라이언트/서버 모델을 이용하여 데이터베이스 프로그래밍을 할 때 유용하게 사용될 수 있는 SQL 빌더의 사용 방법과 유용한 SQL 문장과 테크닉, 마지막으로 캐쉬 업데이트를 활용하여 프로그래밍을 하는 방법에 대해서 알아보았다.

다음 장에서는 실제 DBMS 의 종류와 그 특징 들에 대해서 알아보고, 이들의 접속 관리를 어떻게 할 것인지에 대한 내용을 알아보도록 할 것이다.

## 클라이언트/서버 데이터베이스 어플리케이션의 제작 (II)

이번 장에서는 앞 장에 이어서 클라이언트/서버 데이터베이스 어플리케이션을 제작할 때 고려해야 할 여러가지에 대해서 설명할 것이다. 그 중에서도 공통적으로 접속을 관리하는 텔파이의 컴포넌트들의 사용법을 소개하고, 실제로 많이 쓰이고 있는 몇몇 DBMS들의 특징과 이들을 다룰 때 주의해야 할 점에 대해서 알아볼 것이다. 지면에 여유가 있다면, 각각의 DBMS에서 제공하는 SQL 문장의 독특한 점에 대해서도 다루면 좋겠지만 이를 언급하는데 따로 1권의 책이 필요할 정도로 많은 분량이기 때문에, 여기서는 주로 서버에 접속하는 부분에 중점을 두고 설명할 것이다.

### 서버 로그인 조절

대부분의 원격 데이터베이스 서버는 데이터베이스에 접근할 때 사용자 이름과 패스워드를 요구한다. 런타임에서 로그인에 대한 서버의 요구를 다루는 방법은 3가지가 있다.

데이터베이스 컴포넌트의 LoginPrompt 프로퍼티를 True(디폴트)로 설정하면, 어플리케이션은 서버가 사용자 이름과 패스워드를 요구할 때 표준 로그인 대화상자를 보여주게 된다. 또 한가지 방법은 LoginPrompt를 False로 설정하고, 데이터베이스 컴포넌트의 Params 프로퍼티에서 사용자 이름과 패스워드를 설정하는 방법이다. 예를 들어,

```
USER NAME=SYSDBA  
PASSWORD=masterkey
```

주의:

Params 프로퍼티는 쉽게 볼 수 있기 때문에, 서버의 보안성이라는 측면에서 추천할 만한 방법이 아니다.

마지막으로 데이터베이스 컴포넌트의 OnLogin 이벤트에서 데이터베이스 Params 프로퍼티에 있던 내용을 이용하는 방법이다. OnLogin의 LoginParams의 Values 프로퍼티를 이용해서 다음과 같이 설정한다.

```
LoginParams.Values['USER NAME'] := UserName;  
LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
```

### 런타임에서 데이터베이스 컴포넌트 만들기

런타임에서 데이터베이스 컴포넌트를 생성하려면 이름을 유일한 값으로 설정해야 하며, 세션과의 연관성을 가지게 해야 한다. 다음의 함수는 TempDatabase 라는 임시 데이터베이스를 이용해서 세션과 데이터베이스가 기존에 존재하지 않을 경우 새로운 세션과 데이터베이스 컴포넌트를 생성해 주는 예제이다.

```
function RunTimeDbCreate(const DatabaseName, SessionName: string): TDatabase;
var
    TempDatabase: TDatabase;
begin
    TempDatabase := nil;
    try
        {세션이 있으면 활성화, 없으면 새로운 세션을 생성}
        Sessions.OpenSession(SessionName);
        with Sessions do
            with FindSession(SessionName) do
                Result := FindDatabase(DatabaseName);
                if Result = nil then
                    begin
                        {새로운 데이터베이스 컴포넌트 생성}
                        TempDatabase := TDatabase.Create(Self);
                        TempDatabase.DatabaseName := 데이터베이스 Name;
                        TempDatabase.SessionName := SessionName;
                        TempDatabase.KeepConnection := True;
                    end;
                    Result := OpenDatabase(DatabaseName);
                end;
            end;
        except
            TempDatabase.Free;
            raise;
        end;
    end;
```

다음의 코드는 이 함수를 이용해서 런타임에서 디폴트 세션의 데이터베이스 컴포넌트를 생성하는 역할을 한다. 이 때 데이터베이스의 이름이 중복되지 않도록 MyDbCount 라는 정수형 변수를 이용한다.



```

var
  MyDatabase: array [1..10] of TDatabase;
  MyDbCount: Integer;
begin
  {MyDbCount 의 초기화}
  MyDbCount := 1;
  ...
  {런타임에서 데이터베이스 컴포넌트 생성}
  begin
    MyDatabase[MyDbCount]:=RunTimeDbCreate('MyDb'+IntToStr(MyDbCount),'');
    Inc(MyDbCount);
  end;
  ...
end;

```

## 원격 서버에 접속할 때 고려할 점

디자인 시에 커넥션 파라미터를 생성하거나 편집하는 방법에는 다음 3 가지 방법이 있다.

1. 데이터베이스 탐색기(Database Explorer)나 BDE 관리 유틸리티 (BDE Administration utility)를 이용하는 방법
2. 오브젝트 인스펙터의 Params 프로퍼티를 더블 클릭하여 문자열 리스트 에디터(String List editor)를 호출하여 이용하는 방법
3. 데이터베이스 컴포넌트를 더블 클릭하여 데이터베이스 프로퍼티 에디터(Database Properties editor)를 호출하여 이용하는 방법

이들 방법은 모두 데이터베이스 컴포넌트의 Params 프로퍼티를 편집하게 된다. 여기에는 패스 정보, 서버 이름, 스키마의 캐쉬 크기, 언어 드라이버, SQL 쿼리 모드 등에 대한 정보가 들어 있다.

런타임에서 엘리어스 파라미터를 설정하려면 Params 프로퍼티를 직접 편집하면 된다.

## 세션의 관리

세션의 현재 상태를 결정하려면, Active 프로퍼티를 검사한다. Active 가 False 면 (디폴트) 세션과 연관된 모든 데이터베이스와 데이터 세트가 닫혀 있는 것이며, True 이면 열려 있는

것이다.

Active 를 True 로 설정하면 세션의 OnStartup 이벤트를 호출하며, NetFileDir, PrivateDir, ConfigMode 프로퍼티의 값을 설정한다. NetFileDir 과 PrivateDir 프로퍼티는 파라독스 테이블에 연결하기 위해 쓰이며, ConfigMode 는 BDE 가 세션에서 생성된 BDE 앨리어스를 어떻게 다룰 것인지를 결정한다.

일단 세션을 활성화한 뒤에는 데이터베이스 연결은 OpenDatabase 메소드를 통해 실시할 수 있다. 데이터베이스나 데이터 세트가 열려 있는 경우 세션 컴포넌트의 Active 프로퍼티를 False 로 설정해야 한다.

- 패스워드가 걸려있는 파라독스 테이블의 이용

세션 컴포넌트는 파라독스 테이블의 패스워드를 다루기 위해 4 개의 메소드와 1 개의 이벤트를 제공한다. 제공하는 메소드는 AddPassword, GetPassword, RemoveAllPasswords, RemovePassword 이며, 이벤트는 OnPassword 이다.

AddPassword 프로시저는 문자열을 파라미터로 넘기면 세션이 파라독스 테이블을 열 때 패스워드를 제공하여 바로 접근이 가능하게 된다. 만약에 패스워드가 걸려있는 파라독스 테이블에 접근할 때 AddPassword 를 호출하지 않고, OnPassword 이벤트 핸들러도 작성하지 않은 경우에는 패스워드를 입력하라는 대화 상자가 뜬다.

RemovePassword 는 삭제할 패스워드 문자열을 파라미터로 넘기면 이전에 메모리에 추가된 패스워드를 제거한다. RemoveAllPasswords 는 이전에 추가된 패스워드를 모두 제거한다. GetPassword 는 OnPassword 이벤트를 일으킨다. OnPassword 이벤트는 어플리케이션이 파라독스 테이블을 처음 열려고 할 때 일어난다. 이 이벤트에서 BDE 에 패스워드를 제공하고, 다루는 코드를 써준다. 이 때 델파이의 디폴트 작업은 대화상자를 띄워서 패스워드를 입력받는 것이다.

- 컨트롤 파일과 임시 파일 위치의 지정

NetFileDir 은 파라독스의 네트워크 컨트롤 파일인 PDOXUSRS.NET 의 위치를 지정한다. 이 파일은 네트워크 드라이브에서 파라독스 테이블을 공유하게 해주는 역할을 한다. 파라독스 테이블을 공유해야 하는 모든 어플리케이션은 반드시 같은 네트워크 컨트롤 파일의 디렉토리를 지정해야 한다 (보통은 네트워크 파일 서버의 디렉토리를 지정한다).

PrivateDir 은 BDE 에서 로컬 SQL 문장 등을 다룰 때 생성하는 임시 테이블을 저장할 디렉토리를 지정한다.

1. 컨트롤 파일 위치의 지정

텔파이는 주어진 데이터베이스 앨리어스의 BDE 환경설정 파일에서 NetFileDir 의 값을 알아낸다. NetFileDir 값을 직접 설정하면, 이 값이 BDE 환경설정으로 기록된다.

다음 코드는 디폴트 세션의 NetFileDir 을 어플리케이션이 실행되는 디렉토리로 지정한다.

```
Session.NetFileDir := ExtractFilePath(ParamStr(0));
```

주의: NetFileDir 은 어플리케이션이 열린 파라독스 파일이 없을 때만 바꿀 수 있다.

## 2. 임시 파일 위치의 저장

PrivateDir 프로퍼티에 값이 지정되어 있지 않으면, BDE 는 현재의 디렉토리를 사용한다. 만약 어플리케이션이 네트워크 파일 서버에서 직접 실행될 경우 데이터베이스를 열기 전에 PrivateDir 을 사용자의 하드 디스크에 있는 디렉토리로 지정함으로써 어플리케이션의 퍼포먼스를 향상시킬 수 있다.

다음의 코드는 디폴트 세션의 PrivateDir 프로퍼티를 사용자의 C:\WTEMP 디렉토리로 지정한다.

```
Session.PrivateDir := 'C:\WTEMP';
```

## ● BDE 앨리어스 관리

### 1. 앨리어스의 visibility 지정

세션의 ConfigMode 프로퍼티는 어떤 BDE 앨리어스가 세션에서 사용가능한지를 결정한다. ConfigMode 는 어떤 형태의 세션들을 쓸 수 있는지를 결정하는 세트(set) 이다. 디폴트 값은 cmAll 로 [cfmVirtual, cfmPersistent]와 같은 의미이다. 이 때에는 세션에서 생성되거나, BDE 환경설정 파일에 있는 모든 앨리어스를 사용할 수 있다.

ConfigMode 프로퍼티의 주된 목적은 어플리케이션이 세션 레벨에서 앨리어스의 보이는 정도를 조절하기 위한 것이다. 예를 들어, ConfigMode 를 [cfmSession]으로 설정하면 세션에서 생성된 앨리어스만 쓸 수 있다.

### 2. 세션 앨리어스를 다른 어플리케이션이나 세션에서 사용하게 하려면 ...

세션에서 생성된 앨리어스는 BDE 에 의해 메모리에 복사되어 저장된다. 기본적으로 이 복사본은 생성된 세션에서만 쓰인다. 이 앨리어스를 같은 어플리케이션의 다른 세션에서 사용하려면 ConfigMode 프로퍼티가 cmAll 이거나 [cfmPersistent]여야 한다.

앨리어스를 모든 세션이나 어플리케이션에서 사용할 수 있게 하려면 세션의 SaveConfigFile 메소드를 사용해야 한다. SaveConfigFile 은 메모리의 앨리어스를 BDE 환경설정 파일에 기록하며, 이 파일을 사용하는 어플리케이션에서 읽을 수 있게 된다.

### 3. 앨리어스, 드라이버, 파라미터의 지정

세션 컴포넌트의 5 개의 메소드가 어플리케이션에서 BDE 앨리어스의 정보를 얻는데 쓰인다. 이들은 다음과 같다.

- GetAliasNames: 세션이 접근하는 앨리어스를 나열한다.
- GetAliasParams: 지정된 앨리어스의 파라미터를 나열한다.
- GetAliasDriverName:  
앨리어스가 사용하는 BDE 드라이버의 이름이 담긴 문자열을 돌려준다.
- GetDriverNames: 세션에서 사용가능한 모든 BDE 드라이버의 이름을 나열한다.
- GetDriverParams: 지정된 드라이버에 대한 드라이버 파라미터들을 돌려준다.

### 4. 앨리어스의 생성, 수정, 삭제

세션은 앨리어스를 생성, 수정, 삭제할 수 있다. AddAlias 메소드는 SQL 데이터베이스 서버에 대한 BDE 앨리어스를 생성하는 메소드이다. AddStandardAlias 는 파라독스, 디베이스, 아스키 테이블에 대한 새로운 BDE 앨리어스를 생성한다.

AddAlias 는 3 개의 파라미터를 가진다. 앨리어스의 이름을 가진 문자열, 사용할 SQL 링크 드라이버를 지정하는 문자열, 앨리어스에 대한 파라미터를 담은 문자열 리스트가 그것이다. 예를 들어, 다음의 문장은 인터베이스 서버에 접근하는 새로운 앨리어스를 추가한다.

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
```

```

        .
        .
        .
    finally
        AliasParams.Free;
    end;
end;

```

AddStandardAlias 는 3 개의 문자열 파라미터를 가진다. 앨리어스의 이름, 파라독스 또는 디베이스 테이블의 패스, 테이블을 열때 사용할 디폴트 드라이버가 그것이다. 예를 들어, 다음 문장은 파라독스 테이블에 접근할 새로운 앨리어스를 생성한다.

```
AddStandardDriver('MYDBDEMOS', 'C:\WTESTINGWDEMOSW', 'Paradox');
```

일단 앨리어스를 생성하면, 파라미터는 ModifyAlias 를 호출하여 수정할 수 있다. ModifyAlias 는 2 개의 파라미터를 가지는데, 하나는 앨리어스의 이름이고 다른 하나는 바뀐 파라미터와 그 값을 나열한 문자열 리스트이다. 다음 예제는 CATS 앨리어스의 OPEN MODE 파라미터를 READ/WRITE 로 바꿔 준다.

```

var
    List: TStringList;
begin
    List := TStringList.Create;
    with List do
        begin
            Clear;
            Add('OPEN MODE=READ/WRITE');
        end;
    Session.ModifyAlias('CATS', List);
    List.Free;
end;

```

세션에서 생성된 앨리어스를 삭제할 때 쓰는 메소드는 DeleteAlias 이다. DeleteAlias 는 삭제할 앨리어스의 이름을 파라미터로 한다.

주의:

DeleteAlias 는 BDE 환경설정 파일에 기록된 앨리어스는 삭제하지 못한다. 만약 환경설정 파일에 기

특된 앨리어스까지 삭제하려면 DeleteAlias 호출 후 SaveConfigFile 메소드를 호출해야 한다.

- 데이터베이스 커넥션의 생성, 열기, 닫기

세션에서 데이터베이스 커넥션을 열려면 OpenDatabase 메소드를 사용한다. 이 메소드는 열어야 하는 데이터베이스의 이름을 파라미터로 넘기는데, 이름은 BDE 앨리어스이거나 데이터베이스 컴포넌트의 이름이어야 한다. 파라독스와 디베이스의 경우에는 이름이 패스를 포함한 이름일 수도 있다. 예를 들어, 다음 문장은 DBDEMOS 앨리어스가 가리키는 데이터베이스와의 커넥션을 연다.

```
var
    DBDemosData: TDatabase;
begin
    Session.OpenDatabase('DBDEMOS');
    ...
```

OpenDatabase 가 호출될 때마다 데이터베이스의 참조계수(reference count)가 1 씩 증가한다. 참조계수가 0 보다 큰 동안 데이터베이스는 열려 있다.

CloseDatabase 메소드를 사용하면 각각의 데이터베이스 커넥션을 닫을 수 있으며, Close 메소드를 사용하면 한 번에 모든 커넥션을 닫을 수 있다. CloseDatabase 를 호출하면 데이터베이스의 참조계수가 1 씩 감소하며, 이 수치가 0 이 되면 데이터베이스가 닫히게 된다. CloseDatabase 는 닫을 데이터베이스의 이름을 파라미터로 받는데, 이 값은 BDE 앨리어스, 데이터베이스 컴포넌트 이름 등이 될 수 있다. 예를 들어, 다음 문장은 DBDEMOS 앨리어스에 지정된 데이터베이스 커넥션을 닫는다.

```
CloseDatabase('DBDEMOS');
```

지정된 데이터베이스 이름이 임시 데이터베이스 컴포넌트와 연관되어 있고, 세션의 KeepConnections 프로퍼티가 False 이면, 임시 데이터베이스 컴포넌트는 메모리에서 해제되며, 커넥션이 종료된다.

또한, 데이터베이스 컴포넌트가 미리 선언되어 있고, 인스턴스화 되어 있으며, 세션의 KeepConnections 프로퍼티가 False 이면, CloseDatabase 는 데이터베이스 컴포넌트의 Close 메소드를 호출하고 커넥션을 종료한다.

모든 데이터베이스 커넥션을 닫는 방법은 세션의 Active 프로퍼티를 False 로 설정하거나, 세션의 Close 메소드를 호출하면 된다.

## 세션의 추가

디자인 시에 세션을 추가하는 것은 쉽다. 단지 데이터 모듈이나 폼에 세션 컴포넌트를 위치시키고, 프로퍼티를 설정하고 이벤트 핸들러를 작성하면 된다. 런타임에서 세션을 생성하고 프로퍼티를 설정, 메소드를 호출하려면 다음과 같이 해야 한다.

1. TSession 변수를 선언한다.
2. Create constructor 를 호출하여 새로운 세션을 인스턴스화 한다. 이렇게 하면 세션에 대한 데이터베이스 컴포넌트들과 BDE 콜백(callback)의 빈 리스트가 작성되고, KeepConnections 프로퍼티가 True 로 설정된다. 어플리케이션의 세션 리스트에 세션이 추가된다.
3. SessionName 프로퍼티에 적절한 이름을 설정한다.
4. 세션을 활성화하고, 프로퍼티를 설정한다.

다음의 코드는 이런 단계를 잘 밝은 샘플 코드이다.

```
var
    SecondSession: TSession;
begin
    SecondSession := TSession.Create;
    with SecondSession do
        try
            SessionName := 'SecondSession';
            KeepConnections := False;
            Open;
        end;
        .
        .
        .
        finally
            SecondSession.Free;
        end;
    end;
```

TSessionList 의 OpenSession 메소드를 이용해서 세션을 생성하고 열 수 있다. 이렇게 OpenSession 을 사용하는 것이 Create 를 호출하는 것보다 안전한데, 이는 OpenSession 이

이미 존재하지 않는 세션만을 생성하기 때문이다.

## 세션의 여러 데이터베이스 컴포넌트에 대한 접근

세션의 Databases, DatabaseCount 프로퍼티는 세션과 연관된 데이터베이스 컴포넌트에 접근할 수 있게 해준다.

Databases 프로퍼티는 세션과 연관된 활성화된 데이터베이스 컴포넌트들의 배열이다. DatabaseCount 프로퍼티와 함께 사용하여 모든 활성화된 데이터베이스 컴포넌트에 영향을 줄 수 있다. DatabaseCount 는 정수형 프로퍼티로 현재 세션과 연관된 활성화된 데이터베이스의 수를 나타낸다.

예를 들어, 다음의 코드는 각각의 활성화된 데이터베이스의 KeepConnection 프로퍼티를 True 로 설정한다.

```
var
    MaxDbCount: Integer;
begin
    with Session do
        if DatabaseCount > 0 then
            for MaxDbCount := 1 to DatabaseCount do
                Databases[MaxDbCount].KeepConnection := True;
            end;
        end;
    end;
```

## 여러 세션의 관리

멀티 스레드를 사용하여 데이터베이스 조작을 하는 어플리케이션을 제작하려면, 각각의 스레드에 추가적인 세션을 생성해야 한다.

주의:

세션 컴포넌트를 추가할 때 반드시 SessionName 프로퍼티는 유일한 값으로 설정해서 디폴트 세션의 SessionName 프로퍼티와 혼동되지 않도록 해야 한다.

세션 컴포넌트를 디자인 시에 배치할 때는 어플리케이션이 동작할 때 몇 개의 스레드를 사용할 지 대강 짐작하고 있어야 한다. 그러므로, 세션을 동적으로 생성하는 것이 바람직한 경우가 많은데, 이렇게 하려면 런타임에서 Sessions.OpenSession 을 호출해야 한다.

Sessions.OpenSession 메소드는 세션의 이름을 파라미터로 받는다. 다음의 코드는 동적으



로 새로운 세션을 생성하고 활성화 시킨다.

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

이 문장에서 새로운 세션이 생성될 때마다 현재 세션의 수를 이름에 추가하여 세션의 이름이 유일한 값을 가지도록 하고 있다.

Sessions 는 TSessionList 의 컴포넌트로 데이터베이스 어플리케이션에 의해 자동으로 인스턴스화 된다. 다음 테이블에 TSessionList 컴포넌트의 프로퍼티와 메소드를 정리해 보았다.

프로퍼티, 메소드	목 적
Count	세션의 수를 돌려 준다 (active, inactive 포함).
FindSession	지정된 이름을 가진 세션을 찾아서 세션 컴포넌트의 포인터를 돌려준다. 찾는 세션이 없으면 nil 을 돌려주며, 파라미터에 아무것도 쓰지 않으면 디폴트 세션의 포인터를 돌려준다.
GetSessionNames	현재 인스턴스화된 세션 컴포넌트의 이름들을 문자열 리스트(string list)에 담아서 돌려준다. 최소한 'Default'는 돌아온다.
List	지정된 세션 이름을 가진 세션 컴포넌트를 돌려주며, 해당되는 것이 없으면 예외가 발생한다.
OpenSession	새로운 세션을 생성하고, 활성화 한다.
Sessions	세션 리스트에 Ordinal 값으로 접근할 수 있다.

## 세션의 정보 이용

세션에는 세션과 데이터베이스 컴포넌트의 정보를 알아낼 수 있는 메소드가 있다. 다음 테이블에 이들 메소드를 정리하였다.

메소드	목 적
GetAliasDriverName	데이터베이스의 지정된 앨리어스의 BDE 드라이버 정보 얻기
GetAliasNames	데이터베이스의 BDE 앨리어스의 리스트 얻기
GetAliasParams	데이터베이스의 지정된 BDE 앨리어스의 파라미터 목록 얻기
GetConfigParams	BDE 환경설정 파일에서 특정 환경설정 정보 얻기
GetDatabaseNames	현재 사용되는 TDatabase 컴포넌트의 이름과 BDE 앨리어스의 목록 얻기
GetDriverNames	현재 인스턴스화된 BDE 드라이버의 이름들 얻기
GetDriverParams	지정된 BDE 드라이버의 파라미터 리스트 얻기

GetStoredProcNames	지정된 데이터베이스의 모든 stored procedure 의 이름 얻기
GetTableNames	지정된 데이터베이스의 지정된 패턴과 일치하는 테이블들의 이름얻기

GetAliasDriverName 을 제외하고, 문자열 리스트의 세트를 리턴값으로 받는다 (GetAliasDriverName 은 단일 문자열이다.). 예를 들어, 다음 코드는 디폴트 세션의 모든 데이터베이스 컴포넌트와 앨리어스의 이름을 얻는다.

```
var
    List: TStringList;
begin
    List := TStringList.Create;
    try
        Session.GetDatabaseNames(List);

        .
        .
        .
    finally
        List.Free;
    end;
end;
```

## 데이터베이스 커백션의 검색

세션의 FindDatabase 메소드를 사용해서 데이터베이스를 검색할 수 있다. 이 메소드는 찾고자 하는 데이터베이스의 이름을 파라미터로 받는다. 다음의 예제에서는 디폴트 세션에서 DBDEMOS 앨리어스를 사용하는 모든 데이터베이스 컴포넌트를 검색하며, 찾지 못하면 이를 생성하고 연다.

```
var
    DB: TDatabase;
begin
    DB := Session.FindDatabase('DBDEMOS');
    if DB = nil then //데이터베이스가 세션에 없다.
        DB := Session.OpenDatabase('DBDEMOS'); //생성하고 open
    if Assigned(DB) and DB.Active then
        begin
```

```

DB.StartTransaction;
.
.
.
end;
end;

```

## 인터베이스와 델파이

가장 먼저 알아볼 DBMS 는 Inprise 의 인터베이스이다. DBMS 자체로도 마켓 쉐어가 그다지 높다고 볼 수는 없지만, 싼 가격과 Inprise 에서 제작했다는 장점 때문에 델파이로 개발되는 프로젝트에서는 채택되는 경우가 많은 서버이다.

일반적인 DBMS 와 달리 인터페이스는 분리된 데이터베이스 접속 요소를 가지고 있지 않기 때문에, 인터페이스 서버에 접근하기 위해서는 이를 따로 정의해야 한다. 원격 인터베이스 서버에 접속하려 하면, 요구되는 접속 정보는 사용하는 프로토콜에 따라 다르다. TCP/IP 를 이용하는 경우 HOSTS 파일이 반드시 서버에 대한 레퍼런스를 포함해야 한다. 그 내용은 '123.33.44.12 marketing'과 같은 형태이면 된다. 또한, TCP SERVICES 파일에 'gds\_db 3050/tcp'와 같이 인터베이스 접근 프로토콜을 지정해야 한다.

이런 작업은 인터베이스를 설치하면 자동적으로 이루어지며, 일단 TCP/IP 접근이 이루어지면 WISQL 등의 인터베이스 도구를 이용하여 접속이 가능하다.

### ● BDE 앨리어스의 설정

데이터베이스 서버에 접속이 가능해지면 BDE 앨리어스를 만들 수 있는데, 일단 BDE Administrator 유틸리티나 델파이 SQL 탐색기를 띄우고 앨리어스 설정 대화 상자를 띄운 뒤에 앨리어스 type 에서 INTRBASE 를 선택한다. 새로운 앨리어스가 생성되면 적당한 이름을 부여하고, Definition 탭에서 파라미터를 설정한다. SERVER NAME 파라미터를 서버의 이름과 접속하고자 하는 데이터베이스 파일 이름으로 설정하는데, 다음과 같은 형태를 가지게 된다.

```
Server:/data/인터베이스/sample.gdb
```

여기에서 Server 는 서버의 이름이며 나머지 부분은 데이터베이스 파일의 패스이다. 옵션으로 USER NAME 파라미터를 데이터베이스 사용자 이름으로 설정할 수 있다. 여기에서 설정한 사용자 이름은 델파이가 로그-인 대화상자의 디폴트 값으로 사용된다.

- 인터베이스 접속에 문제가 있을 때에는 ...

텔파이 어플리케이션을 이용해 인터베이스 서버에 접속할 때 문제가 있을 때에는 다음과 같은 방법으로 문제 해결을 시도하는 것이 좋다.

먼저, 인터베이스에서 제공되는 인터베이스 통신 진단 유틸리티(인터베이스 Communication Diagnostics utility)를 이용하여 데이터베이스 서버에 접속을 시도해 본다. DB 커넥션 페이지에 있는 Test 버튼을 클릭해서 접속이 가능한 경우에는 BDE 앨리어스의 설정이 잘못된 것이 접속이 안되는 원인일 가능성이 높다. 그러므로, BDE 관리자를 이용해서 앨리어스 설정이 잘 되었는지 알아보는 것이 좋다. 접속이 안되는 경우에는 NetBEUI 나 Winsock 페이지에서 Test 버튼을 클릭해서 접속을 시도한다. 이 경우에 접속이 된다면 지정된 내용에 문제가 있다는 의미이다.

다른 방법으로는 인터베이스의 WISQL 유틸리티를 이용하여 서버에 접속을 시도해 본다. 텔파이 어플리케이션은 접속에 문제가 있는데, 여기에서는 접속이 된다면 BDE 앨리어스 설정의 문제로 보면 된다. BDE 앨리어스 문제를 해결할 때에는 보통 SERVER NAME 파라마미터가 잘못된 경우가 많다.

WISQL 도 접속에 실패하는 경우에는 프로토콜 문제일 가능성이 높다. 그러므로, TCP/IP 를 사용한다면 ping 을 이용하여 호스트 컴퓨터에 접속을 시도한다. 그리고, NT 기반의 서버에서 명명된 파이프(named pipe)를 이용하는 경우라면 실행 메뉴를 이용하여 'net view WWservername' 명령을 수행해 본다. 여기에서 servername 은 SQL 서버가 동작하는 NT 기계의 이름이다. 이 명령이 성공적으로 수행되면 'netuse [WWservername\WIPC\\$](#)' 명령을 실행해 본다. 여기서 실패하는 경우라면 NT 시스템 설정에 문제가 있다는 의미이다. TCP/IP 를 사용하고, ping 도 정상적으로 동작하는 경우라면 아마도 HOSTS 파일의 내용이 문제가 있을 것이다. 이 파일의 내용이 제대로 설정되어 있는지 확인하기 바란다.

ping 이 안되는 경우라면 이것은 네트워크 문제로 간주하면 된다. ping 은 되는데, WISQL 로 접속이 안되고 HOSTS 파일에도 문제가 없는 경우에는 telnet 을 이용해서 접속을 시도해 본다. telnet 접속이 되지 않는다면 이것은 서버 기계의 inet 데몬이 제대로 동작하지 않는 것이다. telnet 접속이 되는데 WISQL 접속이 되지 않는 경우에는 인터베이스 서버 설치가 제대로 안되었을 가능성이 높다.

## 오라클과 텔파이

오라클은 현재 전세계적으로 가장 많이 팔린 RDBMS 이다. 최근에 발표된 오라클 8 의 경우에는 ORDBMS 로서의 형태를 가지게 되어, 중첩된 테이블과 ODB 의 장점이 객체 연결 기능 등을 제공한다.

여기서는 오라클 서버에 접속하는 방법을 기준으로 간단하게 설명하고자 한다. 그렇지만, 오라클을 서버로 해서 개발을 하는 경우에는 오라클에 대한 내용을 보다 자세하게 이해하고

개발을 하면 보다 강력한 성능을 이용할 수 있다.

오라클에 접속하기 위해서는 오라클의 SQL Net 접속 소프트웨어를 설치하고 환경 설정을 해야 한다. 이를 설치하기 위해서는 단순히 오라클의 ORAINST 설치 프로그램을 실행하면 된다. 환경 설정을 하기 위해서는 먼저 데이터베이스 앨리어스를 정의해야 한다. 여기서 앨리어스는 데이터베이스 서버에 접속하기 위해 필요한 모든 정보를 포함한다. 이 앨리어스는 오라클 드라이버에 대해서만 동작하는 것으로, 텔파이의 BDE 앨리어스와는 다르다.

오라클 데이터베이스 앨리어스를 SQL Net 환경설정 프로그램을 이용해서 환경 설정을 하려면, SQL Net Easy Configuration 프로그램을 시작하고 Add Database Alias 를 선택한 뒤 OK 버튼을 클릭한다.

다음 대화상자에서는 데이터베이스 앨리어스를 선택할 수 있는데, 여기에서 사용하고자 하는 데이터베이스 앨리어스를 선택한다. 만약 새로운 앨리어스를 추가하고자 할 때에는 새로운 이름을 적어 넣고 OK 버튼을 클릭한다.

그 다음에는 네트워크 프로토콜을 선택할 수 있다. 이렇게 네트워크 프로토콜을 선택한 뒤에는 서버에 대한 추가적인 정보를 주어야 한다. TCP/IP 를 선택한 경우라면 HOSTS 파일에 나타날 서버의 호스트 이름과 IP 주소를 입력해야 한다. SPX 를 선택한 경우에는 SPX 서비스 이름을 대신 사용한다.

마지막 대화상자에서는 데이터베이스 앨리어스에 대한 접속 정보가 표시된다. OK 를 클릭하면 데이터베이스 앨리어스가 추가된다. SQL Net configuration 프로그램의 메인 메뉴로 돌아가면, Exit 를 선택하고 어플리케이션을 닫는다.

데이터베이스 앨리어스가 환경설정된 후에는 이를 이용하여 데이터베이스 서버에 접속을 시도할 수 있다. 이를 위해서는 오라클의 SQL Plus 유틸리티를 이용하여 다음과 같이 접속을 시도할 수 있다.

sqlplus [USERNAME/PASSWORD@ALIASNAME](#) @<file>.<ext>

USERNAME 과 PASSWORD 에는 사용하고자 하는 사용자 이름과 패스워드를 입력하고 ALIASNAME 에는 데이터베이스 앨리어스의 이름을 사용한다.

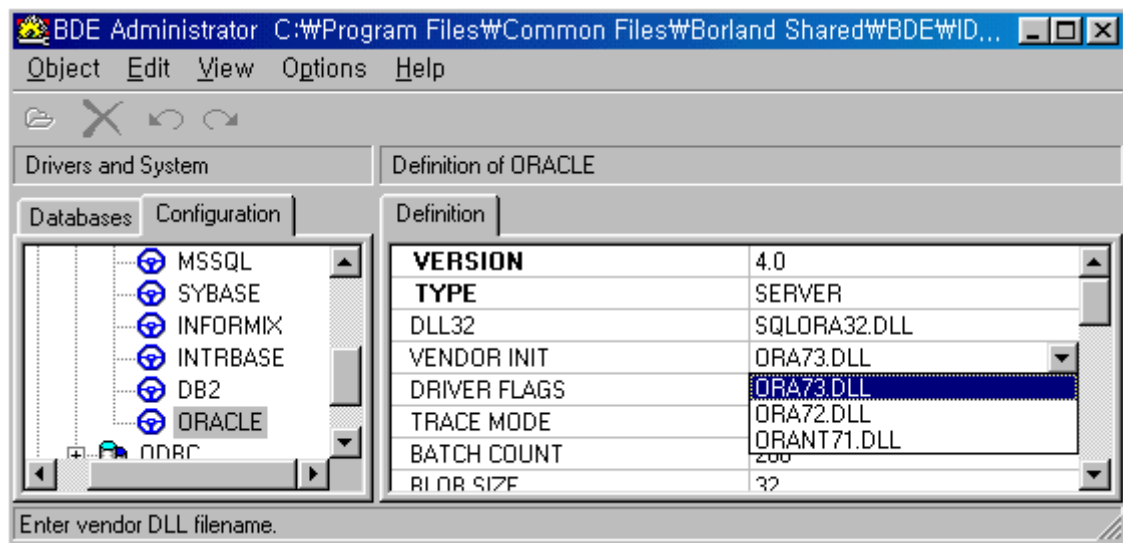
이 밖에도 오라클의 TNSPING 프로그램을 이용하여 서버 접속을 테스트할 수 있다. 이 프로그램은 ping 과 사용법이 비슷하다.

## ● BDE 앨리어스의 설정

서버에 접속이 가능하면, BDE 앨리어스를 만든다. BDE 관리자를 이용하면 되는데, 인터베이스에 비해 다소 복잡한 절차를 거친다.

앨리어스를 만들기 전에 먼저 Configuration 탭의 Drivers|Native|Oracle 을 선택하고, 오른쪽 탭의 VENDOR INIT 항목에서 다음과 같이 적절한 클라이언트를 설정한다. 예를 들

어 서버가 오라클 7.3 이라면 ORA73.DLL 을 선택해야 한다.



그 밖에도 이 곳에서 여러 항목을 변경할 수 있다. 일단 드라이버의 설정이 끝났으면, 데이터베이스 탭에서 앨리어스 type 이 ORACLE 인 새로운 앨리어스를 추가한다. 추가한 앨리어스에 대한 설정만 하면 BDE 앨리어스의 설정이 종료되는데 그 중에서 SERVER NAME 항목과 NET PROTOCOL 항목은 SQL Net configuration 프로그램에서 생성한 앨리어스와 네트워크 프로토콜로 설정한다. 또한, NAME 파라미터를 데이터베이스 서버에 접속하고자 하는 사용자의 이름으로 설정하면 텔파이가 디폴트 로그-인 대화상자를 띄울 때 이 이름을 디폴트로 사용한다.

- 오라클 접속에 문제가 있을 때에는 ...

먼저 오라클의 SQL Plus 유틸리티를 이용하여 서버에 접속을 시도해 본다. 여기서 접속이 되지만 텔파이 어플리케이션에서 접속이 되지 않는다면 이는 BDE 앨리어스 환경 설정의 문제로 봐야 한다. 만약 SQL Plus 유틸리티로 접속이 되지 않으면, 프로토콜의 문제를 의심해야 한다. 서버와의 접속에 TCP/IP 를 사용한다면 ping 을 이용하여 해당 서버와의 접속이 되는지 테스트하고, 이때 호스트 이름으로는 접근이 안되는데 IP 주소로는 접속이 된다면 HOSTS 파일에 문제가 있는 것이므로 이를 수정한다. SPX 프로토콜을 사용하는 경우에는 서버의 SPX 서비스 번호를 제대로 설정해야 한다.

TCP/IP 를 사용하는 경우에 ping 이 제대로 되지 않는다면 네트워크 상의 문제로 봐야 한다. SQL Plus 로 접속은 되지 않는데, ping 이 잘 된다면 오라클 홈 디렉토리가 사용하고자 하는 패스트 지정되었는지 알아본다. 패스가 정확하다면 (홈 디렉토리)\NETWORK\WADMIN\WTNSNAMES.ORA 파일이 제대로 설정되었는지 알아보아야 한다. 이 파일은 텍스트 파일이기 때문에 쉽게 편집할 수 있지만, 직접 편집하기 보다는 SQL Net

configuration 프로그램을 이용하여 편집하는 것이 좋다.

설정에 문제가 없다고 판단되면, 프로토콜을 한 번 변경해보는 것이 좋다.

## ● 그 밖에 ...

오라클은 DBMS 시장에서 마켓 셰어 1 위의 제품인 만큼, 접속 차원에서의 문제를 제외하고도 유용한 여러가지 정보가 있다. 여기에 대해서 설명할 것이다.

보통 RDB 로 시스템을 구성하게 되면 RDB 상에 무수히 많은 데이터베이스가 존재할 수 있기 때문에, 이를 정리하여 주는 관리자의 필요성이 크게 대두된다. 그렇기 때문에, 데이터베이스를 만들 경우에 해당 데이터베이스의 관리자를 정하게 된다. 이때, 단독 관리자도 존재하지만 작업내용에 따라 일을 분담하여 작업하게 되는 복수의 관리자가 존재할 수도 있다. 오라클의 경우 SQL 이나 DBA 가 CREATE DATABASE 명령을 통하여 정상적으로 데이터베이스를 생성할 때, 다음의 SYS 와 SYSTEM 이라는 사용자가 자동적으로 등록된다.

1. SYS: 데이터베이스의 데이터 사전 소유자
2. SYSTEM : SQL Forms 등의 툴을 위한 데이터 사전 소유자

이제 SYS 와 SYSTEM 은 새로운 DBA(관리자)권한을 갖는 사용자를 만들 수 있는 권리를 가지게 된다. 그런데, 왜 이런 식으로 두 개의 관리자가 존재하는 것일까? 그것은 DB 시스템의 관리와 어플리케이션 관리자를 다르게 보기 때문이다. 보통 관리자라 함은 SYSTEM 관리자로써 DBA 작업을 수행하게 된다.

SYS 는 CHANGE\_ON\_INSTALL, SYSTEM 은 MANAGER 라는 디폴트 패스워드를 가지고 있다. 그렇기 때문에, 보안을 위해서 이를 빨리 교체해 주는 것이 좋다.

### 참고:

인터베이스의 경우 SYSDBA 와 MASTERKEY 로 구분되는 기본적인 DBA 의 설정을 갖는다. 그리고 MS-SQL 서버의 경우 SA 로 설정되는 기본적인 DBA 관리자가 존재한다.

SYSTEM 의 패스워드를 변경하기 위한 명령어는 'ALTER USER SYSTEM IDENTIFIED BY 새로운 패스워드' 이다.

참고로 오라클을 설치할 경우에 언어 설정에서 한국어로 한정하면 보통 클라이언트 제품에서 한글을 지원하지 않고, 영어만 지원하는 경우에 충돌의 우려가 있으므로 이 경우에는 영어로 설정하여 사용하는 것이 좋다.

## 마이크로소프트 SQL 서버와 델파이

마이크로소프트 SQL 서버는 저렴한 가격과 윈도우 NT 서버의 보급으로, 최근 들어 가장 가파른 마켓 셰어의 상승을 가져오고 있는 제품이다. 앞으로도 오라클과 함께 시장을 양분할 것으로 예상되는 대표적인 DBMS 이다.

- BDE 앨리어스의 설정

BDE 앨리어스를 새로 만들 때에는 앨리어스 type 으로 MSSQL 을 선택하는 것을 제외하고는 다른 경우와 같다. 생성된 앨리어스의 SERVER NAME 항목에 SQL 서버의 이름을 설정해야 하는데, 이 이름은 SQL Client Configuration 유틸리티에서 사용된 이름과 일치해야 한다. USER NAME 항목을 설정하면 디폴트 로그-인 대화 상자에 설정된 이름을 사용할 수 있다.

그 밖에 SQL 서버에서 사용한 몇 가지 항목들이 있다.

1. APPLICATION NAME

이 항목은 SQL 서버의 sysprocesses 테이블에 사용될 프로세스의 이름을 나타나게 할 수 있다. 이렇게 하면, 어플리케이션 프로세스가 서버에서 다른 프로세스와 구별이 가능하다.

2. BLOB EDIT LOGGING

이 값을 False 로 설정하면 BLOB 필드에 변화가 생긴 내용을 logging 하지 않도록 할 수 있다. 이렇게 함으로써 BLOB 저장 요구를 최소화하고, 수행 성능을 향상시킬 수 있다. 이 방법은, BLOB 데이터를 SQL 서버의 벌크 복사 장치(bulk copy facility)를 이용하여 전송하므로 목적 데이터베이스에 select into/bulk copy 문장을 설정해야 한다. select into/bulk copy 는 sp\_dboption 저장 프로시저를 이용하여 사용할 수 있다.

3. CONNECT TIMEOUT, HOST NAME

CONNECT TIMEOUT 항목은 클라이언트가 접속을 위해 대기하는 시간을 한정할 수 있다. 디폴트는 60 초로 설정되어 있는데, 경우에 따라서 이 값을 변경할 수 있다. 그리고, HOST NAME 항목을 현재의 컴퓨터로 설정하면 SQL 서버의 sysprocesses 테이블에 실행되는 호스트의 이름을 나타나게 할 수 있다. 이렇게 하면, sp\_who 저장 프로시저를 이용해서 접속되어 있는 컴퓨터 들을 확인할 수 있다.

4. DATABASE NAME



이 항목은 접속할 SQL 데이터베이스의 이름을 설정할 때 사용한다. 이 값을 비워두면 서버의 디폴트 데이터베이스가 선택된다. 그렇지만, 이 값을 선택하여 데이터베이스를 명확히 하는 것이 좋다.

## 5. TDS PACKET SIZE

이를 이용하여 Tabular Data Stream(TDS) 패킷의 크기를 지정할 수 있다. TDS 는 SQL 서버가 클라이언트와 데이터를 교환할 때 사용하는 패킷 프로토콜이다. 이 값은 0~65535 까지 가능하지만, 실제로는 512~32767 까지의 값이 사용된다. 디폴트 값은 4096 이며, 보통 4096~8192 사이의 값의 수행 성능이 가장 높다.

SQL 서버의 sp\_configure 저장 프로시저를 이용하면 서버에서 지원하는 현재의 최대 패킷 크기를 결정할 수 있다.

### ● 클라이언트 접속의 설정

SQL 서버의 클라이언트 접속은 SQL Client Configuration 유틸리티를 이용하여 환경 설정을 할 수 있다. 이 유틸리티를 클라이언트에서 실행하고 Advanced 탭을 선택한 뒤, 서버 엔트리 박스에서 서버의 이름을 설정한다. 그리고, Net Library Configuration 에서 사용할 디폴트 네투워크를 선택해야 하는데, 보통 명명된 파이프(named pipe)나 TCP/IP 중 하나를 선택하게 된다.

명명된 파이프를 사용하는 경우에 서버의 이름으로 컴퓨터 이름을 설정하는 것이 좋다. 그리고, TCP/IP 를 사용할 경우에는 IP 주소나 포트 번호 등의 정보를 따로 설정해야 한다.

### ● SQL 서버 접속에 문제가 있을 경우에는 ...

먼저 마이크로소프트의 ISQL/w 유틸리티를 이용하여 서버에 접속을 시도한다. 여기서 접속이 되지만, 텔파이 어플리케이션에서 접속이 되지 않는다면 지금까지와 같이 BDE 엘리어스 설정이 잘못된 경우이다. ISQL/w 을 이용해도 접속이 되지 않으면, ping 을 이용하여 접속을 시도한다. 다른 내용은 모두 오라클의 경우와 같으므로 이를 참고하기 바란다.

## 정 리 (Summary)

이번 장에서는 클라이언트/서버 데이터베이스 어플리케이션을 개발할 때 알아야 할 몇 가지 컴포넌트의 사용 방법과 접속 관리, 주요 DBMS 에 접속할 때 알아야 할 점 등에 대해서 알아보았다. 다음 장에서는 데이터베이스 어플리케이션을 개발할 때 도움이 되는 여러가지 간단한 팁들에 대해서 알아볼 것이다.

## 데이터베이스 어플리케이션 제작의 팁들

데이터베이스 어플리케이션 들을 제작하다 보면 여러가지 작은 난관에 부딪힐 때가 많다. 이럴 때에는 아주 간단한 한 줄의 코딩이나 힌트도 커다란 힘이 된다는 것을 많이 경험하게 된다.

이번 장에서는 텔파이로 데이터베이스 어플리케이션을 제작할 때 유용하게 사용할 수 있는 여러가지 팁들을 소개하고자 한다.

### 기본적인 참고사항

먼저 데이터베이스 어플리케이션을 제작할 때 흔히 만날 수 있는 일반적인 팁에 대해서 알아보도록 하자.

- 동적 앨리어스를 활용하자.

앨리어스는 동적으로 만들어 사용하는 것이 좋다. 다시 말해, BDE 를 이용하여 앨리어스를 등록하는 것보다는 TDatabase 컴포넌트를 사용하여 런타임-환경에서 앨리어스를 만드는 것이 좋다.

- UpdateMode 는 upWhereChanged 로 !

UpdatMode 는 보통 upWhereALL 을 사용하는데 upWhereChanged 를 사용하는 것이 데이터베이스 성능을 향상시키는데 좋다. 다만, 멀티-uer 인 경우에는 처음 레코드에 수정을 가한 사람이 종료하기 전에 데이터의 변경을 가할 경우에 문제가 발생할 수 있다. 이런 경우에는 반드시 upWhereALL 을 사용해야 한다.

- SQL 문장을 수행하기 전에는 꼭 Prepare 문을 수행한다.

Prepare 문장이 중요한 이유는, SQL 문장을 수행하기 위한 최적화 작업을 수행하기 때문이다.

- Join 보다는 View 를 많이 만들어라 !

Join 작업으로 동적으로 View 를 만드는 것보다는, 아예 고정시켜 놓고 사용하는 것이 훨씬 좋다는 것은 설명하지 않아도 쉽게 알 수 있을 것이다.

- CD 타이틀을 만들 경우에 읽기 전용의 파라독스 테이블 사용하기

CD 타이틀의 경우 데이터베이스 파일이 읽기 전용 저장 매체인 CD 에 담겨 있다. 이 경우 데이터베이스 파일을 하드 디스크로 복사한 다음 앨리어스를 생성해서 사용해도 되지만, 하드 디스크 공간을 낭비하는 문제가 있다.

BDE 는 Pdxusers.lck 와 Paradox.lck 파일을 네트워크 등의 멀티유저 환경에 로킹(locking) 정보 파일로 사용한다. 문제는 이 두 파일이 읽기 전용일 경우에는 에러가 발생하며, 오동작 한다는 점이다. 이중에서 PARADOX.LCK 파일은 DOS 용 파라독스 테이블을 지원하기 위한 것이므로 없어도 무방하다. 다만, 비어 있는 PDXUSRS.LCK 파일을 미리 하드디스크에 생성하여 CD 타이틀의 데이터베이스 위치에 넣으면 문제를 해결할 수 있다. 그리고, 다음과 같은 코드를 사용한다.

```
DbiAcqPersistTableLock(Database1.handle, 'PARADOX.DRO','PARADOX' );
```

이때 TDatabase 컴포넌트 파라미터의 패스 값에 해당 CD 타이틀의 테이블 위치 값이 입력되도록 해야 한다.

- 그 밖에 ...

1. 품의 삭제에는 Free 보다 Release 를 사용하시는 것이 좋다.
2. RDBMS 의 성능을 제대로 사용하려면 저장 프로시저를 많이 사용하라 !
3. 캐쉬 업데이트를 적극적으로 사용한다.
4. 필드 에디터(Field Editor)에서 만들어진 고정된 필드 객체는 포인터로 연결되어 프로그램 수행상의 성능이 좋다.
5. 멀티-쓰레드를 적극적으로 활용하여, SQL 문장을 RDB 에 던져 놓고 멍청하게 노는 프로그램이 되지 않도록 한다.

## 오류가 발생된 파라독스 테이블의 복구

델파이에서 기본적으로 지원하는 파라독스 구조의 테이블은 좀처럼 오류가 발생되지 않는 구조의 데이터베이스이다. 이를 활용하면 일반적인 업무용은 물론 LAN 공유 프로그램도 충분히 만들어 낼 수 있다. 이렇게 실무에 적용하다 보면 테이블에 오류가 발생하는 것은 기정사실로 보아야만 한다. 프로그래머는 이런 오류가 발생한 경우를 예측해야 하고, 이의 대처 방법을 준비해야 하는데 이번에는 이 오류를 수정하는 방법을 알아보자.

이렇게 오류를 수정하기 위해서는 DLL 파일이 필요하다. 이 DLL 은 파라독스를 사용하는

어떤 프로그램에서도 사용할 수 있다.

이 DLL 을 받아오려면, <http://www.inprise.com/devsupport/bde/utilities.html>에 접근하면 많은 유틸리티를 얻을 수 있다. 이 곳에서 TUtility v4.01 과, Paradox Table Repair 키트를 얻을 수 있다.

Paradox Table Repair 에는 TUtil32.pas 유닛이 있다. 이 유닛의 내용을 살펴보면. TUnit, TUVerifyTable, TURebuildTable, TUGetCRTblDescCount, TUFillCRTblDesc, TUFillCURProps, TUGetExtTblProps, TUExit, TUGetErrorString 등의 함수들을 지원한다. 이 중에서 가장 중요한 함수가 TUVerifyTable 과 TURebuildTable 인데, 이 함수들은 테이블 검사와 테이블 재복구 기능을 수행한다.

그러면, DLL 파일에 포함된 기능을 사용할 클래스를 설계해 보자. 이 클래스는 테이블을 검사하고 복구할 때 progress bar 를 지원하기 위한 콜백 함수를 만드는 것이 주목적이다. 콜백 함수에 대한 더 자세한 내용은 42 장의 내용을 참고하기 바란다.

```
TBDEUtil = class
```

```
  CbInfo: TUVerifyCallback;
```

```
    //콜백 지원을 위한 레코드형 (작업량, 테이블이름, 실제 Process, 인덱스 정보)
```

```
  TUProps: CURProps;      //파라독스를 지원하기 위한 DBI 함수 중에 DB 커서에 대한 정보
```

```
  hDb: hDBIDb;           //DB 의 핸들 값을 저장
```

```
  vhTSes: hTUSes;        // Word 형(테이블 세션정보 저장)
```

```
  constructor Create;
```

```
  destructor Destroy; override;
```

```
  function GetTCursorProps(szTable: String): Boolean;
```

```
  procedure RegisterCallBack;
```

```
  procedure UnRegisterCallBack;
```

```
end;
```

```
// Progress 를 보여주기 위한 콜백 함수.
```

```
function GenProgressCallBack(ecbType: CBType; Data: LongInt; pcbInfo: Pointer):
```

```
  CBRTYPE; stdcall;
```

```
var
```

```
  CbInfo: TUVerifyCallBack;
```

```
begin
```

```
  CbInfo := TUVerifyCallBack(pcbInfo^);
```

```
  if ecbType = cbGENPROGRESS then
```

```
    case CbInfo.Process of      //현재 검사하고 있는 Process 정보는?
```

```
      //CbInfo.percentdone: 현재 진행된 퍼센트값
```

```

TUVerifyHeader:
begin
    헤더에 해당하는 Progress.Position := CBIInfo.percentdone;
end;

TUVerifyIndex:
begin
    Index에 해당하는 Progress.Position := CBIInfo.percentdone;
end;

TUVerifyData:
begin
    Data에 해당하는 Progress.Position := CBIInfo.percentdone;
end;

TURebuild:
begin
    재구성에 해당하는 Progress.Position := CBIInfo.percentdone;
end;

end;

Result := cbrUSEDEF;      //cbrABORT, cbrCONTINUE 등의 리턴 값, cbrUSEDEF 는 사용자 정의
end;

constructor TBDEUtil.Create;
begin
    TUInit(vhtSes);        //지정 테이블을 사용할 수 있도록 DLL 을 초기화
end;

destructor TBDEUtil.Destroy;
begin
    Check(TUExit(vhtSes)); //사용한 테이블의 정보를 해제
    inherited Destroy;
end;

function TBDEUtil.GetTCursorProps(szTable: String): Boolean;
    //테이블의 정보를 읽어낸다. (세션정보, 테이블 명, 읽어올 정보 )
begin
    if TUFillCURProps(vHtSes, PChar(szTable), TUProps) = DBIERR_NONE then
        Result := True

```

```

    else Result := False;
end;

procedure TBDEUtil.RegisterCallback;
begin
    Check(DbRegisterCallBack(nil, cbGENPROGRESS, 0,
        sizeof(TUVerifyCallBack), @CbInfo, GenProgressCallback));
        //GenProgressCallBack 프로시저를 cbGEBPROGRESS 에 콜백 시킴
        //세션정보, 콜백 타입, 콜백의 버퍼 크기, 버퍼 데이터, 보낼 콜백 함수
end;

procedure TBDEUtil.UnRegisterCallback;
begin
    Check(DbRegisterCallBack(nil, cbGENPROGRESS, 0,
        sizeof(TUVerifyCallBack), @CbInfo, nil));
        //만들어 넣은 콜백 함수를 해제
end;

```

먼저 간단한 예로 이 TUTIL32.dll 을 사용하여 테이블의 정보를 읽어 보자. 만드는 프로시저는 테이블의 정보를 읽어내는 TableInfo 루틴이다.

```

procedure TableInfo;
var
    Buffer, Table: String;
begin
    Table := '원하는 테이블명';
    if BDEUtil.GetTCursorProps(Table) then
        //실제 테이블의 정보를 읽어 낸다
        with BDEUtil.TUProps do
            //읽어온 정보는 TUProps 에 있다
            begin
                IntToStr(iFields);           //필드 갯수
                IntToStr(iRecBufSize);       //레코드 크기
                IntToStr(iIndexes);          //인덱스 수
                InttoStr(iValChecks);        //유효성 검사
                IntToStr(iRefIntChecks);     // 참조무결성 수
            end;
        end;
    end;

```

```

        IntToStr(iRestrVersion);      //재구성 버전
        IntToStr(iPasswords);        //패스워드
        IntToStr(iCodePage);         //코드 페이지
        IntToStr(iBlockSize);        //블록 크기
        IntToStr(iTblLevel);         //테이블 레벨
    end;
end;

```

이제 간단한 구조를 보았으니 본격적으로 테이블의 오류를 검증하는 루틴을 살펴보자.

```

function Trepair_form.Tableverify(FileName: String): Boolean;
    // 테이블의 오류 검증하는 루틴
var
    Msg, L: Integer;
    Table: String;
    Ret: Boolean;
begin
    Screen.Cursor := crHourGlass;
    Ret := False;
    try
        Table := FileName;
        Check(TUExit(BDEUtil.vHtSes));
        Check(TUInit(BDEUtil.vHtSes));
        //DLL 을 사용하기 위해 종료/시작을 반복하여 초기화를 수행한다
        //이 DLL 은 작업을 수행 할 때마다, 재 초기화작업을 거쳐야 한다
        BDEUtil.RegisterCallBack;
        //Progress 를 출력하기 위해 콜백 함수를 설정 한다
    try
        if TUVerifyTable(BDEUtil.vHtSes, PChar(Table), szPARADOX, 'VERIFY.DB',
            nil, 0, Msg) = DBIERR_NONE then
            //실제 TUtil32.dll 의 TUVerifyTable 함수를 호출하여 검증 작업을 실시한다
            //세션 핸들 값, 테이블 명, 드라이버 속성, 에러출력 DB, 패스워드, 옵션, 에러 레벨-오류
            가 발생하면 오류값을 읽어온다
        begin
            //오류가 발생하였을 경우에 메시지를 보여준다
            case Msg of

```

```

0: eLabel.Caption := '테이블 확인 완료. 해당 테이블에 오류가 없습니다.';
1: eLabel.Caption := '테이블 확인 완료. 해당 테이블에 비교된 테이블입니다.';
2: eLabel.Caption := '테이블 확인 완료. 확인이 부정확하게 종료되었습니다.';
3: eLabel.Caption := '테이블 확인 완료. 테이블을 재구성 하십시오.';
4: eLabel.Caption := '테이블 확인 완료. 테이블을 재구성할 수 없습니다!.';

else
begin
    eLabel.Caption := '테이블 오류 확인 실패.';

    //구분할 수 없는 에러가 발생했을 경우..

    Ret := True;

end;

end;

end;

finally
    BDEUtil.UnRegisterCallBack;           //콜백 함수를 원위치 시킨다
end;

finally
    Screen.Cursor := crDefault;
end;

Tableverify := Ret;

end;

```

이제 테이블의 오류를 체크하여 낼 수 있다. 이렇게 체크하여 오류가 발생한 테이블은 다음의 복구 루틴으로 테이블을 재복구할 수 있다.

```

procedure Trepair_form.TableRebuild(FileName: String);    //테이블을 재구성한다
var
    iFld, ildx, iSec, iVal, iRI, iOptP, iOptD: Word;
    szTable, Backup: String;
    Rslt: DBIResult;
    Msg: Integer;
    TblDesc: CRTBIDesc;
begin
    Screen.Cursor := crHourGlass;

    try
        Check(TUExit(BDEUtil.vHtSes));
    
```



```

Check(TUInit(BDEUtil.vHtSes));          //TUtl32.dll 을 재구성한다
szTable := FileName;
BDEUtil.RegisterCallBack;              //콜백을 지정한다
try
    Check(TUVerifyTable(BDEUtil.vHtSes, PChar(szTable), szPARADOX, 'VERIFY.DB',
        nil, 0, Msg));                  //Rebuild 하기 전에 테이블을 한번 검사한다
    Rslt := TUGetCRTblDescCount(BDEUtil.vhTSes, PChar(szTable), iFld,
        idx, iSec, iVal, iRI, iOptP, iOptD);    //재구성하기 위한 정보를 읽어 온다
        //(세션 값, 테이블 명, 필드 수, 인덱스 수, 레코드 수, 유효성 수, 참조 무결성
        수, 파라미터 옵션, 옵션 데이터 크기)
    if Rslt = DBIERR_NONE then          //에러가 없으면, 테이블의 구성 정보를 읽어 낼 수 있으면..
    begin                                // 정보를 제대로 읽어오면.. 재구성이 가능하다
        FillChar(TblDesc, SizeOf(CRTblDesc), 0);
        StrPCopy(TblDesc.szTblName, szTable);
        TblDesc.szTblType := szParadox;
        TblDesc.szErrTblName := 'Rebuild.DB';
        //재구성하는 테이블은 파라독스이며 에러가 발생하면 Rebuild.db 에 저장한다
        TblDesc.iFldCount := iFld;      //필드 수 만큼..
        GetMem(TblDesc.pFldDesc, (iFld * SizeOf(FldDesc)));
        //필드의 수를 읽어서 필요한 만큼의 메모리 할당
        TblDesc.idxCount := idx;
        GetMem(TblDesc.pIdxDesc, (idx * SizeOf(IdxDesc)));
        //인덱스 수를 읽어서 필요한 만큼의 메모리 할당
        TblDesc.iSecRecCount := iSec;
        GetMem(TblDesc.pSecDesc, (iSec * SizeOf(SecDesc)));
        //테이블 정보의 크기만큼의 메모리 할당.
        TblDesc.iValChkCount := iVal;
        GetMem(TblDesc.pvchkDesc, (iVal * SizeOf(VCHKDesc)));
        //유효성 정보를 읽어 들임.
        TblDesc.iRintCount := iRI;
        GetMem(TblDesc.printDesc, (iRI * SizeOf(RINTDesc)));
        //참조 무결성 정보를 읽어들임.
        TblDesc.iOptParams := iOptP;
        GetMem(TblDesc.pFldOptParams, (iOptP * sizeOf(FLDDesc)));
        //파라미터 정보를 읽어들임
        GetMem(TblDesc.pOptData, (iOptD * DBIMAXSCFLDLLEN));

```

```

try
    Rslt := TUFillCRTblDesc(BDEUtil.vhTSes, @TblDesc, PChar(szTable), nil);
    //테이블 정보 읽어 들이기
    if Rslt = DBIERR_NONE then
        begin
            //에러없이 읽어 들이면..
            Backup := 'Backup.Db';
            if TURebuildTable(BDEUtil.vhTSes, PChar(szTable), szPARADOX,
                PChar(Backup), 'KEYVIOL.DB', 'PROBLEM.DB', @TblDesc)
                //테이블의 재복구를 시도한다.
                //(세션 값, 테이블 명, 드라이버 타입, 백업 Db 명, 에러 Db 명,
                문제저장 Db 명, 테이블 구조 정보 값)
                = DBIERR_NONE then runOK.Caption := '재복구 성공 !'
            else runOK.Caption := '재복구 실패 !';
        end
    else
        MessageDlg('테이블의 구조나 내용에 이상이 있음: ', mtError, [mbok], 0);
    finally
        FreeMem(TblDesc.pFldDesc, (iFld * SizeOf(FldDesc)));
        FreeMem(TblDesc.pldxDesc, (ildx * SizeOf(ldxDesc)));
        FreeMem(TblDesc.pSecDesc, (iSec * SizeOf(SecDesc)));
        FreeMem(TblDesc.pvchkDesc, (iVal * SizeOf(VCHKDesc)));
        FreeMem(TblDesc.printDesc, (iRI * SizeOf(RINTDesc)));
        FreeMem(TblDesc.pfldOptParams, (iOptP * sizeOf(FLDDesc)));
        FreeMem(TblDesc.pOptData, (iOptD * DBIMAXSCFLDLEN));
        //할당된 메모리를 해제
    end;
end;
finally
    BDEUtil.UnRegisterCallBack; //콜백 함수를 돌려 놓는다.
end;
finally
    Screen.Cursor := crDefault;
end;
end;

```

이렇게 만들어진 두개의 함수를 사용하면 파라독스로 만들어진 테이블의 오류를 검사하고 복구할 수 있다. 이렇게 만들어진 루틴은 델파이로 만드는 프로그램 내부에 삽입하여 Idle time 에 검사를 수행하거나 오류가 발생되었을 경우에 복구할 수 있는 루틴으로 사용하면 좋을 것이다.

## 데이터베이스 테이블 생성 방법

파라독스 파일을 사용하는 경우 기본적인 필드를 가지는 테이블 파일을 생성할 수 있다. 이제 다음의 파일 구조를 가지는 파라독스 테이블을 프로그램에서 동적으로 생성하여 보자.

NameDB  
Name Char 20 Primary  
Addrress Char 40

NikNameDB  
Name Char 20 Primary  
Nikname Char 20 Primary

TelDB  
Name Char 20 Primary  
Tel1 Char 20 Primary  
Tel2 Char 20 Primary

그리고 데이터베이스 파일을 생성하면서 NikNameDB 에서 NameDB 에 Name 필드로 Table Lookup 을 연결하고, TelDB 에서 NameDB 에 Referential Integrity 로 Name 필드를 사용해서 연결하도록 할 것이다.

먼저 다음은 테이블을 생성하는데 사용되는 변수들에 대한 설명이다.

변 수	데이터 형	설 명
szDirectory	DBIPATH	만들고자 하는 Table 의 위치를 지정하는 변수
TableDesc	CRTblDesc	테이블 정보를 저장하는 변수
FieldsDesc	array[0..n] of FLDDesc	필드 정보를 저장하는 변수
IndexesOP	array[0..n] of CROpType	인덱스의 처리 방법에 대해 지정되는 변수
IndexesDesc	array[0..n] of IDEXDesc	인덱스 정보를 저장하는 변수
RefIntegOp	array[0..n] of CROpType	참조 무결성 처리 방법에 대해 지정되는 변수
RefInteg	array[0..n] of RINTDes	참조 무결성의 정보를 저장하는 변수

ValCheckOP	array[0..n] of CROpType	테이블 록업(Lookup)처리 방법에 대해 지정하는 변수
ValCheckDesc	array[0..n] of VCHKDesc	테이블 록업의 정보를 저장하는 변수

사용할 프로시저를 차례로 알아보자.

- 필드 정보값 넣기

```

procedure DefField (const sName: String; const iAFldType, iASubType, iAFldNum,
    iAUnits1, iAUnits2: Integer);
begin
    with FieldsDesc[iAFldNum] do
        begin
            iFldNum := iAFldNum;
            StrPCopy(szName,sName);
            iFldType := iAFldType;
            iSubType := iASubType;
            iUnits1 := iAUnits1;
            iUnits2 := iAUnits2;
        end;
    end;
end;

```

실제 FLDDesc 의 값은 다음과 같다.

이름	데이터 형	내 용
iFldNum	Word	필드의 번호로 파라독스의 경우 변하지 않는 수
szName	DBINAME	필드의 이름
iFldType	Word	필드의 타입, 변환모드에서 xltNONE 인 경우 해당 드라이버의 물리적인 타입이며 아닌 경우는 BDE 의 논리적인 타입이다
iSubType	Word	필드의 서브타입으로 변환모드의 설정 값에 따라 논리적인 서브 타입이 나 드라이버의 물리적인 서브 타입이다.
iUnits1	Integer	문자, 숫자, 기타 등의 크기를 지정, 수치 타입인 경우 iUnit2 는 배율입니다.
iUnits2	Integer	소수점 자리 수, 기타 등의 정보를 지정한다.
iOffset	Word	레코드 버퍼에서 해당 오프셋을 보고 점프한다. (테이블 생성시에는 참고 하지 않는다)
iLen	Word	바이트로 나타낸 필드의 계산된 길이(테이블 생성 시에는 참고 되지 않

		는다)
iNullOffset	Word	필드의 NULL 을 기록한다. (테이블 생성시에는 참고되지 않는다)
efldvVchk	FLDVchk	유효성 검사의 타입을 검사한다. (테이블 생성시에는 참고되지 않는다)
efldrRights	FLDRights	사용자를 위한 계산된 필드 수준 권한을 체크할 경우에 사용하는 필드이다. (테이블 생성 시에는 참고되지 않는다)

이상의 필드에서 필요한 필드만 선택해서 해당 필드의 정보를 채웁니다.

- 테이블의 인덱스 정보 만들기

```

procedure DefIndex (const sName,sTagName,sFormat,sKeyExp,sKeyCond: string;
                    const aFields: array of integer;
                    const iIndexPos,iIndexID,iAFldsInKey,iKeyLen,
                        iKeyExptype,iABlockSize,iARestrNum: integer;
                    const bAPrimary,bAUnique,bADescending,bAMaintained,bASubSet,
                        bAExpIDX,bAOutOfDate,bACaseInsensitive: boolean);

var
    i: byte;
begin
    IndexesOp[iIndexPos] := crAdd;
    with IndexesDesc[iIndexPos] do begin
        StrPCopy(szName,sName); iIndexId := iIndexId;
        StrPCopy(szFormat,sFormat); StrPCopy(szTagName,sTagName);
        StrPCopy(szKeyExp,sKeyExp); StrPCopy(szKeyCond,sKeyCond);
        iFldsInkey := iAFldsInkey; iKeyLen := iKeyLen; iKeyExpType := iKeyExpType;
        iBlocksize := iABlocksize; iRestrNum := iARestrNum;
        bPrimary := bAPrimary; bUnique := bAUnique; bDescending := bADescending;
        bMaintained := bAMaintained; bSubset := bASubset; bExpIdx := bAExpIdx;
        bOutOfDate := bAOutOfDate; bCaseInsensitive := bACaseInsensitive;
        FillChar(aiKeyFld,SizeOf(aiKeyFld),#0);
        for i := Low(aFields) to High(aFields) do
            aiKeyFld[i] := aFields[i];
        end;
    end;
end;

```

해당소스를 살펴보기 전에 인덱스 정보(IdeDesc)에 들어가는 구조를 알아보자.

이름	데이터 형	내용
szName	DBITBLNAME	인덱스 이름
iIndexId	Word	인덱스 번호
szTagName	DBINAME	태그 명(dBase 전용)
szFormat	DBINAME	선택형 포맷 (BTree, HASH, 기타)
bPrimary	Bool	True 이면 기본 인덱스
bUnique	Bool	True 이면 인덱스는 유일한 키를 포함
bDescending	Bool	True 이면 내림차순
bMaintained	Bool	True 이면 유지 관리되는 인덱스
bSubset	Bool	True 이면 부분집합 인덱스 (dBase 전용)
bExpldx	Bool	True 이면 표현식 인덱스이다 (dBase 전용)
iCost	Word	사용하지 않음..
iFldsInKey	Word	복합 인덱스에서 키 필드의 수를 지정
iKeyLen	Word	인덱스 생성 동안은 지정되지 않고, 바이트로 나타낸 키의 물리적인 길이를 가리킨다.
bOutofDate	Bool	True 이면 인덱스의 날짜가 지난 것이다.
iKeyExpType	Word	키 표현식의 타입을 지정(dBase 전용)
aiKeyFld	DBIKEY	키 안의 키 숫자 배열
szKeyExp	DBIKEYEXP	표현식 인덱스를 위한 키 표현식 지정(dBase 전용)
szKeyCond	DBIKEYEXP	부분집합 조건을 정의한 식을 지정, dBase 식으로 기술
bCaseInsensitive	Bool	True 이면 인덱스는 대소문자를 구별하지 않는다.
iBlockSize	Word	바이트로 나타낸 인덱스의 블록크기
iRestrNum	Word	인덱스 생성시에는 나타나지 않으나, 내부 재구성 정보를 가진다.
iUnused	Array[0..15] of word	사용하지 않음..

## ● 참조 무결성 처리

```

procedure DefRefInt (const iRintPos,iARintNum,iAFldCount: integer;
                    const aiAThisTabFld,aAiOthTabFld: array of integer;
                    const sRintName,sTblName: string; eAType: RINTType;
                    const eAModOP,eADelOP: RINTQual);
var
    i: byte;

```

```

begin
  RefIntegOp[iRintPos] := crAdd;
  with RefInteg[iRintPos] do begin
    iRintNum := iARintNum; StrPCopy(szRintName,sRintName);
    eType := eAType; StrPCopy(szTblName,StrPas(szDirectory)+stblName);
    eModOp := eAModOp; eDelOp := eADelOp; iFldCount := iAFldCount;
    FillChar(aiThisTabFld,SizeOf(aiThisTabFld),#0);
    for i := Low(aiAThisTabFld) to High(aiAThisTabFld) do
      aiThisTabFld[i] := aiAThisTabFld[i];
    FillChar(aiOthTabFld,SizeOf(aiOthTabFld),#0);
    for i := Low(aAiOthTabFld) to High(aAiOthTabFld) do
      aiOthTabFld[i] := aAiOthTabFld[i];
    end;
  end;
end;

```

RINTDesc 는 참조무결성의 구조를 입력하기 위한 데이터 구조이다.

이름	데이터 형	내 용
iRintNum	Word	참조 무결성 번호
szRintName	DBINAME	태그 이름
eType	RINTType	타입(rintMASTER 또는 rintDEPENDANT)
szTblName	DBIPATH	연결되는 테이블 이름
eModOp	RINTQual	수정 한정사 (rintRESTRICT 또는 rintCASCADE)
eDelOp	RINTQual	삭제 한정사 (rintRESTRICT 또는 rintCASCADE)
iFldCount	Word	링크 키의 필드 수
aiThisTabFld	DBIKEY	이 테이블에서 참조 무결성을 구성하는 필드 값
aiOthTabFld	DBIKEY	다른 테이블의 필드 수

#### ● 유효성 검사

```

procedure DefValCheck (const iValPos,iAFldNum: integer;
  const aAMinVal,aAMaxVal,aADefVal: array of Byte;
  const bARequired,bAHasMinVal,bAHasMaxVal,bAHasDefVal: boolean;
  const sPict,sLkupTblName: string;
  const eALKUPTYPE: LKUPTYPE);
var

```

```

i: byte;
begin
  ValCheckOp[IValPos] := crAdd;
  with ValCheckDesc[IValPos] do begin
    iFldNum := iAFldNum; StrPCopy(szPict,sPict);
    bRequired := bARequired; bHasMinVal := bAHasMinVal;
    bHasMaxVal := bAHasMaxVal; bHasDefVal := bAHasDefVal;
    eLKUPType := eALKUPType; StrPCopy(szLkupTblName,sLkupTblName);
    FillChar(aMinVal,SizeOf(aMinVal),#0);
    for i := Low(aAMinVal) to High(aAMinVal) do
      aMinVal[i] := aAMinVal[i];
    FillChar(aMaxVal,SizeOf(aMaxVal),#0);
    for i := Low(aAMaxVal) to High(aAMaxVal) do
      aMaxVal[i] := aAMaxVal[i];
    FillChar(aDefVal,SizeOf(aDefVal),#0);
    for i := Low(aADefVal) to High(aADefVal) do
      aDefVal[i] := aADefVal[i];
    end;
  end;
end;

```

VCHKDesk 유효성 검사에 사용되는 구조는 다음과 같다.

이 름	데이터 형	내 용
iFldNum	Word	필드 수 (1-n)
bRequired	Bool	True 이면 필드 값이 필요하다
bHasMinVal	Bool	True 이며 최소 값을 가진다
bHasMaxVal	Bool	True 이면 최대 값을 가는다.
bHasDefVal	Bool	True 이면 디폴트 값을 가진다.
aMinVal	DBIVCHK	최소 값
aMaxVal	DBIVCHK	최대 값
aDefVal	DBIVCHK	디폴트 값
szPict	DBIPICT	그림 문자열
elkupType	LKUPTYPE	탐색/채움 타입(Paradox 전용)
szLkupTblName	DBIPATH	탐색 테이블 이름(읽는 정보로만 사용)

유효성 LKUPTYPE 의 값은 다음과 같다.



이름	내용
IkupNONE	테이블에 탐색기능이 없다
IkupPRIVATE	현재 필드만 + 전용
IkupALLOCORRESP	모든 해당 필드 + 도움말 없음
IkupHELP	현재 필드만 + 도움말
IkupALLOCORREPSHELP	모든 해당필드 + 도움말

- 테이블 생성에 필요한 정보 입력

다음 DefTable 프로시저는 원하는 테이블을 생성한다.

```

procedure DefTable (const sName,sType,sPassword: string;
  const iAFldCount,iAIDXCount,iAValChkCount,iARintCount: integer);
begin
  FillChar(TableDesc,SizeOf(CRTblDesc),#0);    //TableDesc 에 초기값 채우기..
  with TableDesc do
  begin
    StrPCopy(szTblName,sName); StrPCopy(szTblType,sType);
    // 테이블명과 테이블의 타입을 지정하고
    bProtected := (sPassword <> '');           //패스워드가 있으면 세팅하고..
    if bProtected then
    begin
      StrPCopy(szPassword,sPassword);
      Session.AddPassword(sPassword);           //패스워드가 있으면 세션에 패스워드를 넣는다.
    end;
    bPack := True;
    iFldCount := iAFldCount; pFldDesc := @FieldsDesc;
    iValChkCount := iAValChkCount;
    pcrValChkOp := @ValCheckOp; pvchkDesc := @ValCheckDesc;
    iIDXCount := iAIDXCount;
    pcrIDXOp := @IndexesOp; pIDXDesc := @IndexesDesc;
  end;
end;

```

여기서 사용할 CRTblDesc 의 내용은 다음과 같다.

이름	데이터 형	내 용
szTblName	DBITBLNAME	패스와 확장자를 포함한 테이블 명
szTblType	DBINAME	드라이버 타입
szErrTblName	DBIPATH	에러 테이블 명(선택 값)
szUserName	DBINAME	사용자 명 (있을 경우에..)
szPassword	DBINAME	패스워드(bProtected 가 True 인 경우) 파라독스 전용
szProtected	Bool	True 이면 암호화
szPack	Bool	테이블이 packing 되어야 하는 경우에 True
iFldCount	Word	필드 정의의 수
percFldOp	pCROpType	필드 작업의 배열
pfldDesc	pFLDDesc	필드 디스크립터 배열
iLdxCount	Word	인덱스의 수
perIdxOp	pCROpType	인덱스 작업 배열
pidxDesc	pIDXDesc	인덱스 디스크립터 배열
iSecRecCount	Word	보안 값의 수 (Paradox 전용)
psecDesc	pSECDesc	보안 디스크립터 배열(Paradox 전용)
iValChkCount	Word	유효성 검사의 수(Paradox 및 SQL 전용)
pecrValChkOp	pCROpType	유효성 검사 작업 배열
pvchkDesc	pVCHKDesc	유효성 검사 디스크립터 배열(Paradox 및 SQL 전용)
iRintCount	Word	참조 무결성 수(Paradox 전용)
pecrRintOp	pCROpType	참조 무결성 작업 배열
printDesc	pRINTDesc	참조 무결성 지정시의 수(Paradox 전용)
iOptParams	Word	선택형 파라미터의 수
pfldOptParams	pFLDDesc	선택형 파라미터를 위한 필드 디스크립터의 배열
pOptData	Pointer	선택형 파라미터의 값

- 실제 테이블을 생성한다.

다음의 코드들은 앞에서 만들어진 소스 코드를 통하여 해당 필드에 값을 채우고 인덱스와 참조 무결성, 유효성 검사에 값을 채운 다음 테이블 정보를 넣는다.

```
procedure StoredNamedb;
```

```
begin
```

```
    DefField('Name', fldZSTRING, 0, 0, 20, 0);
```

```

DefField('Address', fldZSTRING, 0, 1, 100, 0);
DefField('Num', fldINT32, 0, 2, 1, 0);
DefIndex("", "", "", "", "", [1],
          0, 0, 1, 20, 0, 4096, 1, True, True, False, True, False, False, False, False);
DefTable('NAMEDB.DB', 'PARADOX', "", 3, 1, 0, 0);
end;

```

```

procedure StoredTeldb:
begin
  DefField('Name', fldZSTRING, 0, 0, 20, 0);
  DefField('Tel1', fldZSTRING, 0, 1, 20, 0);
  DefField('Tel2', fldZSTRING, 0, 2, 20, 0);
  DefIndex("", "", "", "", "", [1, 2, 3],
            0, 0, 3, 60, 0, 4096, 1, True, True, False, True, False, False, False, False);
  DefRefInt(0, 1, 1, [1], [1], 'nameRef', 'NAMEDB.DB',
            rintDEPENDENT, rintCASCADE, rintRESTRICT);
  DefTable('TELDDB.DB', 'PARADOX', "", 3, 1, 0, 1);
end;

```

```

procedure StoredNiknamedb:
begin
  DefField('Name', fldZSTRING, 0, 0, 20, 0);
  DefField('NikName', fldZSTRING, 0, 1, 20, 0);
  DefIndex("", "", "", "", "", [1, 2],
            0, 0, 2, 40, 0, 4096, 1, True, True, False, True, False, False, False, False);
  DefValCheck(0, 1, [0], [0], [0],
              False, False, False, False, "", 'nameDB.DB', lkupHELP);
  DefTable('NIKNAMEDB.DB', 'PARADOX', "", 2, 1, 1, 0);
end;

```

이제 실제 MakeAllTables 프로시저를 통하여 테이블을 만들어 낸다. 이때 개발자는 TDatabase 컴포넌트를 하나 생성하여 원하는 엘리어스를 선택한다.

```

procedure MakeAllTables (dbDatabase: TDatabase);
var
  iTables: StoredTables;

```

```

begin
    dbDatabase.Connected := True;      // 연결한 다음
    Check(DbGetDirectory(dbDatabase.Handle,False,szDirectory));
        //데이터베이스 컴포넌트에서 해당 디렉토리를 읽어 낸다.
        //StoredNamedb 와 StoredTeldb 와 StoredNikNamedb 를 순서대로 호출한 다음
    if DBCreateTable(dbDatabase.Handle,false,TableDesc) = DBIERR_FILEEXISTS then
        //DBCreateTable 을 통하여 TableDesc 의 값으로 테이블을 생성한다.
    begin
        //에러가 발생하였으므로 작업하던 내용을 모두 지운다.
    end;
end;

```

## 다른 테이블의 데이터 import

테이블 컴포넌트의 BatchMove 메소드를 이용하면 다른 테이블의 데이터를 가져올 수 있다. BatchMove 는 테이블 간의 레코드를 복사하거나, 다른 테이블에서 일어난 레코드를 업데이트, 추가, 삭제하는 기능을 할 수 있다.

BatchMove 는 2 개의 파라미터를 가진다. 데이터를 가져올 테이블의 이름과 어떤 동작을 할 것인지에 대한 모드가 그것이다. 다음 테이블은 BatchMove 의 모드에 대한 값이다.

값	의 미
batAppend	소스 테이블의 레코드를 테이블의 끝에 추가
batAppendUpdate	소스 테이블의 모든 레코드를 추가하고, 테이블에 존재하는 같은 레코드를 업데이트 한다.
batCopy	소스 테이블의 모든 레코드를 복사한다.
BatDelete	소스 테이블에 있는 모든 레코드를 삭제한다.
BatUpdate	소스 테이블의 레코드에 기준하여 레코드를 업데이트 한다.

다음 문장은 현재 테이블의 레코드를 Customer 테이블의 레코드로 업데이트 한다.

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

BatchMove 는 리턴값으로 성공적인 동작을 한 레코드의 수를 돌려준다.

주의: batCopy 로 레코드를 복사하면, 과거의 레코드에 덮어쓰게 된다.

## 같은 데이터베이스 테이블에 연결된 테이블들의 동기화

하나 이상의 테이블 컴포넌트가 같은 테이블에 연결되어 있으면서 데이터 소스 컴포넌트를 공유하지 않으면 각각의 테이블은 데이터와 레코드에 대한 자신의 뷰를 가지게 된다. 사용자가 이런 테이블 컴포넌트에 접근하게 되면 컴포넌트의 현재 레코드가 서로 달라지게 된다. 이런 상황에서 현재의 레코드를 같게 유지하는 메소드가 있는데, GotoCurrent 메소드가 그것이다.

다음 문장은 CustomerTableOne 테이블 컴포넌트의 현재 레코드를 CustomerTableTwo 테이블 컴포넌트의 현재 레코드와 같게 설정한다.

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

서로 다른 폼에 있는 테이블 컴포넌트를 동기화할 때에는 다음 문장과 같이 폼의 이름을 앞에 적어주면 된다.

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

## 양방향 커서의 비활성화

BDE의 양방향 커서를 비활성화할 때 UniDirectional 프로퍼티를 사용한다. 디폴트 값은 False로 결과 세트(result set)의 커서를 레코드의 전후로 움직일 수 있다는 것을 의미한다. 양방향 커서를 사용하면 작업을 할 때 약간의 오버헤드를 가중시키기 때문에 쿼리의 속도가 다소 느려지게 된다. 그러므로, 쿼리의 속도를 증진시키려면 UniDirectional 프로퍼티를 True로 설정하여 커서가 앞으로만 움직이도록 제한하는 것이 좋다.

다음 코드는 쿼리를 실행하기 전에 UniDirectional을 설정하는 예이다.

```
if not CustomerQuery.Prepared then
begin
    CustomerQuery.UniDirectional := True;
    CustomerQuery.Prepare;
end;
CustomerQuery.Open;
```

## 이종(heterogeneous) 쿼리의 생성

델파이에는 하나 이상의 데이터베이스의 테이블들을 사용하는 이종 쿼리를 지원한다. 예를

들어 오라클, 사이베이스의 테이블과 로컬 디베이스 테이블을 포함한 쿼리가 가능하다. 이렇게 하려면 각각의 서버에서만 지원하는 확장 SQL 문법을 쓰면 안된다.

이중 쿼리를 생성하려면 로컬 디렉토리를 참조하는 BDE 앨리어스를 정의하고 쿼리 컴포넌트의 데이터베이스 Name 프로퍼티를 그 앨리어스로 설정한다. BDE 앨리어스의 정의는 데이터베이스 탐색기(Database Explorer)를 이용하면 된다. 각각의 데이터베이스에 대해 분리된 BDE 앨리어스를 정의하고, SQL 프로퍼티에서 실행할 SQL 문장을 지정한다. Params 프로퍼티를 설정하고 Prepare, Open, ExecSQL 로 쿼리를 실행하면 된다.

예를 들어, CUSTOMER 테이블을 가지고 있는 오라클 데이터베이스의 앨리어스를 Oracle1, ORDERS 테이블을 가지고 있는 사이베이스 데이터베이스의 앨리어스를 Sybase1 이라고 할 때 두 테이블을 이용한 간단한 쿼리를 아래에 적어 보았다.

```
SELECT CUSTOMER.CUSTNO, ORDERS.ORDERNO
FROM ":Oracle1:CUSTOMER", ":Sybase1:ORDERS"
WHERE CUSTNO = 1503
```

## 그 밖의 유용한 팁들 ...

- TDBNavigator 버튼을 동적으로 제어

TDBNavCracker(DBNavigator1).Buttons[nbEdit].Enabled := true/false 로 조절한다.

- 테이블이 비정상 종료할 때를 대비하여 자료를 강제로 저장하는 방법

function DbSaveChanges(hCursor: hDBICur): DBIResult stdcall;  
를 이용한다.

예) DbSaveChanges(TTable.Handle);

- 파라독스의 Auto Increment 형의 필드작성

TQuery 를 가지고 SQL 의 CREATE TABLE 명령을 사용한다.

```
with Query1 do
begin
  DatabaseName := 'DBDemos';
  with SQL do
```

```

begin
    Clear;
    Add('CREATE TABLE "PDoxTbl.db" (ID AUTOINC,');
    Add('Name CHAR(255),');
    Add('PRIMARY KEY(ID))');
    ExecSQL;
    Clear;
    Add('CREATE INDEX ByName ON "PDoxTbl.db" (Name)');
    ExecSQL;
end;
end;

```

- DBGrid 에서 다중 선택된 것 찾아내기

```

var
    x: word;
    TempBookmark: TBookmark;
begin
    DBGrid1.Datasource.Dataset.DisableControls;
    //SelectedRows 프로퍼티에 DBGrid 에서 다중 선택된 레코드들의
    //북마크(bookmark)를 가지고 있다(TBookmarkList)
    with DBGrid1.SelectedRows do
        if Count > 0 then
            begin
                TempBookmark := DBGrid1.Datasource.Dataset.GetBookmark;
                for x := 0 to Count - 1 do
                    begin
                        if IndexOf(Items[x]) > -1 then
                            begin
                                DBGrid1.Datasource.Dataset.Bookmark := Items[x];
                                Showmessage(DBGrid1.Datasource.Dataset.Fields[0].AsString);
                            end;
                        end;
                    end;
                end;
            end;
        end;
    end;
    DBGrid1.Datasource.Dataset.GotoBookmark(TempBookmark);
    DBGrid1.Datasource.Dataset.FreeBookmark(TempBookmark);

```

```
DBGrid1.Datasource.Dataset.EnableControls;  
end;
```

- DBGrid 에서 선택된 필드의 타이틀을 굵게 표시하기

ColEnter 이벤트과 ColExit 이벤트를 사용한다.

```
procedure TForm1.DBGrid1ColEnter(Sender: TObject);  
begin  
    //선택된 필드의 타이틀 색을 파란색으로 굵게 ...  
    DBGrid1.Columns[DBGrid1.SelectedIndex].Title.Font.Color := clRed;  
    DBGrid1.Columns[DBGrid1.SelectedIndex].Title.Font.Style :=  
        DBGrid1.Columns[DBGrid1.SelectedIndex].Title.Font.Style + [fsBold];  
    DBGrid1.Repaint;  
end;
```

```
procedure TForm1.DBGrid1ColExit(Sender: TObject);  
begin  
    //원래대로...  
    DBGrid1.Columns[DBGrid1.SelectedIndex].Title.Font.Color := clBlack;  
    DBGrid1.Columns[DBGrid1.SelectedIndex].Title.Font.Style :=  
        DBGrid1.Columns[DBGrid1.SelectedIndex].Title.Font.Style - [fsBold];  
    DBGrid1.Repaint;  
end;
```

- DBGrid 내에서 메모 필드의 내용을 보려면 ?

OnDrawCell 이벤트를 사용하면 된다. 물론, 메모 필드를 보여주기 위해서는 TMemoField 객체를 미리 생성해야 한다.

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;  
    Field: TField; State: TGridDrawState);  
var  
    P: array [0..50] of char;  
    BS: TBlobStream;  
    S: String;
```



```

begin
  If Field is TMemoField then
  begin
    with (Sender as TDBGrid).Canvas do
    begin
      BS := TBlobStream.Create(Table1Notes, bmRead);
      FillChar(P, SizeOf(P), #0);
      BS.Read(P, 50);           // 50 크기만큼 읽어 들이기.. P 의 배열 크기만큼..
      BS.Free;
      S := StrPas(P);
      while Pos(#13, S) > 0 do S[Pos(#13, S)] := ' ';    //리턴값을 뺀다.
      while Pos(#10, S) > 0 do S[Pos(#10, S)] := ' ';    //LineFeed 값을 뺀다.
      FillRect(Rect);                                     //출력한 값을 초기화한 다음
      TextOut(Rect.Left, Rect.Top, S);                   //메모의 내용을 출력한다.
    end;
  end;
end;

```

- TBlobField 에서 32KB 밖에 데이터를 읽고 쓰지 못하면 ?

이 경우에는 TQuery 의 RequestLive 가 False 인 경우 발생한다. 이때에 데이터를 올바르게 읽고 쓰기 위해서는 BDE 관리자에서 사용하고 있는 데이터베이스 앨리어스에서 BLOB Size 를 실제로 사용되는 데이터 사이즈보다 크게 설정해야 한다.

문제는 BDE 관리자의 환경 설정에서 이미 BLOB 설정이 되어서 만들어진 기존의 앨리어스는 수정된 BLOB Size 가 변경된다고 반영되지 않는 점을 주의해야 한다.

- DBGrid 에서 스크롤 바를 없애는 방법

```

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
  Field: TField; State: TGridDrawState);
begin
  ShowScrollbar(DBGrid1.handle, SB_VERT, False);
end;

```

- DBGrid 입력시에 수치만 입력받게 하는 방법

```

procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
    if DBGrid1.SelectedField.FieldName = 'NO_FIELD' then
    begin
        if Key > #32 then
            if (not (Key in ['0'..'9'])) then Key := #0;
        end;
    end;
end;

```

- DBGrid 에서 특정 컬럼만 이동하게 하는 방법

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    TDrawGrid(dbgrid1).FixedCols := 3;           //세번째 컬럼을 고정
    TDrawGrid(dbgrid1).FixedRows := 1;          //첫번째 컬럼을 고정
    TDrawGrid(dbgrid1).FixedColor := clYellow;  //고정 컬럼의 색
end;

```

- RecordCount 프로퍼티의 사용

TQuery 를 사용할 경우에는 RecordCount 는 대다수 오동작을 수행한다. 그러므로, 사용을 자제하는 것이 좋다. 차라리, 첫번째 레코드가 EOF 인지 검사하는 것이 훨씬 동작속도가 빠르다.

## 정 리 (Summary)

이번 장에서는 데이터베이스 어플리케이션을 개발하면서 유용하게 사용될 수 있는 유용한 팁들에 대해서 다루어 보았다. 여기에 다루지 못한 내용 들이 매우 많지만, 인터넷 등을 뒤져보면 도움이 되는 내용을 많이 찾을 수 있을 것이다.

## 파라독스 데이터베이스 파일의 이해 (Understanding Paradox Database File)

파라독스는 MS-DOS 가 사용되던 시절에 독자적인 사용자 인터페이스, 프로그래밍 언어, 파일 포맷으로 탄생했던 데이터베이스 프로그램이다. 이를 1993 년에 볼랜드에서 윈도우용 버전을 발표하면서 파일 포맷과 사용자 인터페이스를 분리하고, 이를 연결하기 위해 ODAPI(Object Database API)를 사용하도록 하였으며, ODAPI 가 IDAPI 를 거쳐 현재의 BDE 로 발전하기에 이른다.

이렇게 되는 동안 파라독스의 파일 포맷은 따로 분리되어 데스크 탑 데이터베이스 도구로 사용되었다. 최근에 볼랜드는 파라독스 데스크탑 데이터베이스 부분을 코렐 사에 매각하였고, 이것이 코렐의 오피스의 근간이 되었다. 아직도 파라독스 파일 포맷은 BDE 가 지원하는 가장 중요한 파일 포맷이다.

많은 델파이 개발자는 파라독스 테이블을 매일 사용한다. 그렇지만 여기에 대한 정보는 거의 없다고 해도 과언이 아니다. 아직도 파라독스 파일 포맷은 가장 견고하고 잘 설계된 파일 포맷이며, 데스크 탑 데이터베이스와 파일 서버 형태로 사용하는 데이터베이스로서는 가장 좋은 것으로 알려져 있다.

이번 장에서는 파라독스 테이블의 내부 구조에 대해서 살펴보고, 실제로 테이블에 필드를 추가, 삭제하거나 색인의 생성, 테이블의 pack, 패스워드 정의 등의 여러가지 작업을 할 때 내부적으로 어떻게 동작하는지에 대해서 알아보도록 한다. 이 내용은 인터넷에 공개된 문서를 바탕으로 작성한 것이다.

### 파일의 종류

마이크로소프트의 액세스는 데이터베이스 전체를 하나의 파일로 정의하고 있다. .MDB 파일 안에는 테이블과 인덱스, 여러가지 관계와 리포트, 쿼리 심지어 코드까지 저장되어 있다. 이러한 저장 방식에는 Win32 의 구조화 저장 기법이 사용된다. 구조화 저장 기법에 대해서는 이 책의 다른 장에서 언급하므로 참고하기 바란다.

이에 비해 파라독스는 각 기능에 따라 다른 파일로 저장하고 있다. 데이터베이스는 연관이 있는 테이블의 시리즈를 담고 있는 서브 디렉토리이며, 테이블의 각각의 요소들은 분리된 파일에 저장된다. 이들은 하나의 패밀리(family)를 구성하며, 사용자가 파일의 이름을 정의하면 BDE 가 확장자를 조절한다. 파라독스 파일로 사용되는 것에는 다음과 같은 것들이 있다.

1. .DB: 테이블 정의와 메모/그래픽/BLOB 필드 데이터를 제외한 모든 데이터 포함
2. .MB: 메모/그래픽/BLOB 데이터
3. .PX: 테이블의 Primary 키에 대한 인덱스

4. .Xnn, .Ynn: Secondary 키에 대한 인덱스
5. .VAL: 테이블의 참조 무결성과 타당성 검사 등의 정의

파라독스의 과거 버전과 데이터베이스 데스크 탑에서는 이들 외에 .TV, .FAM, .SET, .F, .R 등의 확장자를 가진 파일 들이 있다.

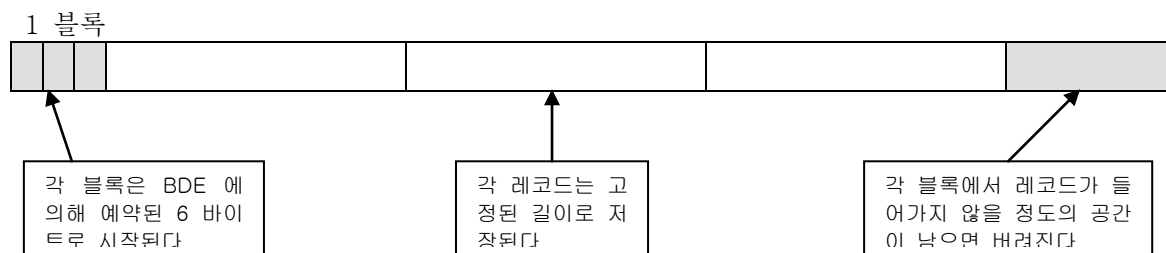
테이블을 복사할 때에는 이들을 모두 같이 복사해 주어야 한다. 그렇지 않으면 문제가 발생할 수도 있다.

## 테이블 포맷

내부적으로 파라독스 파일 포맷은 fixed-length record, VSAM-block, clustered-key 파일 포맷으로 알려져 있다. 이를 쉽게 풀어서 표현하면 다음과 같다.

1. 테이블 내에 고정된 길이를 차지하는 레코드에 저장된 데이터는 길이와 관계 없이 고정되게 저장된다. 예를 들어, 10 자 길이의 alphanumeric 필드에서 이 보다 짧은 길이의 값이 있어도 10 자 길이를 차지한다. (Fixed-length record)
2. 레코드는 물리적인 블록으로 이루어 지는데, 이들은 순차적으로 접근할 수도 있고 랜덤으로 접근할 수도 있다. 각각의 블록에는 하나 이상의 레코드가 존재하며, 레코드는 블록의 크기를 조절하게 만들지 못하므로 각 블록의 뒷 부분에는 레코드 하나가 못들어갈 크기 정도의 빈 공간이 존재하는 경우가 있다. (VSAM-block)
3. 각 블록 내부의 레코드 들은 primary 키 순서에 따라 저장된다. 이렇게 함으로써 primary 키가 사용될 때 접근 속도를 향상시킬 수 있다.

이를 그림으로 정리하면 다음과 같다.



각 테이블은 최소한 2kb 크기의 헤더 파일로 시작되는데, 저장되어야 할 정보량에 따라서 2kb 의 배수의 크기가 될 수도 있다. 헤더에는 다음과 같은 정보 들이 담겨 있다.

1. Structure ID

패밀리 파일의 동기화를 위해 사용되는 것으로, BDE 가 패밀리 파일 중 하나를 열면 파일 헤더의 Structure ID 가 .DB 파일의 ID 와 동일한지 검사하게 된다. 이것이 다르면 .DB 파일이 다른 패밀리 멤버 들이 업데이트 되지 않았을 때, 따로 변경되었다는 의미이다. 그러므로, 테이블의 구조를 재조정하면 그 때마다 모든 패밀리 멤버는 refresh 되어야 한다.

2. 헤더의 크기 (Byte)
3. 레코드의 크기 (Byte)
4. 블록의 크기 (1k, 2k, 4k, 8k, 16k, 32k)
5. 테이블에 키가 있는지 여부
6. 테이블의 필드 수 (1~255)
7. 테이블에 키가 있으면 primary 키가 되는 필드의 번호
8. 필드의 종류, 크기의 배열
9. 필드 이름의 배열
10. 테이블의 데이터 블록의 수 (2 Byte)
11. 첫번째 데이터 블록의 포인터
12. 마지막 데이터 블록의 포인터
13. 테이블 내의 사용 가능한 블록의 수 (2 Byte)
14. 첫번째 사용 가능한 블록에 대한 포인터
15. 문자 번역 방법과 소트 순서 등을 정의하는 테이블 언어
16. AutoIncrement 필드의 최대값에 대한 현재 값
17. 테이블의 데이터를 암호화하는 마스터 패스워드, 헤더는 암호화 되어 있지 않기 때문에 이 패스워드는 안전하다고 말할 수 없다.
18. 테이블에 대한 부가 패스워드의 암호화된 리스트

## 데이터 블록

테이블을 처음 생성하면, 데이터 블록은 할당되지 않는다. 이때에는 테이블이 헤더로만 구성되는데, 첫번째 레코드를 추가함과 동시에 데이터 블록이 .DB 파일에 바로 추가된다.

데이터 블록의 크기는 1k, 2k, 4k, 8k, 16k, 32k 까지 가능하다. 디폴트로 2k 가 설정되어 있지만, BDE Administrator 에 의해서 수정이 가능하다. 블록의 크기는 테이블 내에서 변화되지 않는다. 일단 테이블이 생성되면 BDE 는 블록 크기를 결정하고, 이 크기 만큼 .DB 파일 뒤에 부착시킨다. 데이터 블록의 수는 64k 로 한정되어 있기 때문에, 테이블의 최대 크기는 헤더 크기를 제외하고, 데이터 블록의 크기에 따라 1k 인 경우 64M, 2k 는 128M 이며, 32k 인 경우에는 최대 2048M(2G)까지 가능하다.

그렇지만, 실제로는 이러한 한계에 접근하기는 어렵다. 예를 들어, 100MB 가 넘는 테이블의 구조를 재조정할 경우 경우에 따라서는 이 작업이 몇 시간이 걸릴 수도 있다.

파라독스 파일의 데이터 블록에는 BDE 에 의해 사용되는 6 바이트가 있다. 이들은 3 개의 포인터로 이루어져 있으며, 각 포인터의 크기는 2 바이트이다. 이들의 의미는 다음과 같다.

1. 다음 논리 블록의 번호. 이러한 포인터의 시리즈를 forward chain 이라고 한다. BDE 는 블록 번호를 블록 크기로 곱하고, 여기에 헤더의 크기를 더해서 블록의 물리적인 시작 주소를 계산한다.
2. 이전 논리 블록의 번호. 이러한 포인터의 시리즈를 backward chain 이라고 한다.
3. 블록에 얼마나 많은 활성화된 레코드가 있는지 나타내는 카운터. 데이터 레코드는 언제나 블록의 처음 부분에서 시작하므로, 헤더에 저장되어 있는 레코드 크기를 이용해서 얼마나 많은 블록이 활성화된 데이터를 가지고 있는지 알 수 있다.

데이터 블록이 완전히 비어 있으면, 이 블록은 자유 블록(free block) chain 으로 옮겨 가게 된다. 이 chain 은 BDE 가 테이블에 또 다른 데이터 블록을 추가해야 할 때에 사용된다. 자유 블록에도 다른 블록과 마찬가지로 6 바이트의 포인터가 있으며, 3 번째 포인터가 나타내는 레코드 수는 0 으로 설정되어 있다.

BDE 는 가끔 레코드 크기를 계산할 때 반올림을 적용한다. 예를 들어 블록 크기가 4k(4096)인 경우, 키가 있는 테이블의 최대 레코드 크기를  $(4096 - 6) / 3$  으로 계산해서 1363 바이트로 생각할 수 있다. 그렇지만, BDE 는 1350 바이트를 최대 크기로 결정하고 있다. 만약 레코드의 크기가 1350 바이트가 넘으면 8k 블록을 사용한다.

### 레코드의 삽입과 삭제

레코드의 크기가 204 바이트인 키가 있는 테이블을 생각해 보자. 파라독스 테이블 내에서는 각 블록에 10 개의 레코드를 저장한다고 할 때 2k 크기의 블록을 사용한다. 이 경우 2040 바이트가 10 개의 레코드에 할당되고, 처음의 6 바이트가 블록에 대한 포인터로 사용되므로 단지 2 바이트만 버려진다.

여기서 이 테이블에 4 개의 레코드가 추가되면, BDE 는 블록에 있는 처음 4 개의 레코드 슬롯에 레코드를 저장하고, 레코드 카운터에 4 를 저장한다. 이때 블록의 구조는 다음과 같다.

4	A	B	C	D						
---	---	---	---	---	--	--	--	--	--	--

여기서 C 레코드를 삭제하면 BDE 는 레코드 D 를 C 의 자리로 옮겨 오게 되고, D 가 있었던 공간은 지워지지 않으므로 D 레코드의 데이터는 그대로 남아 있다. 그리고, 레코드 카운터는 3 으로 줄어든다. 이를 그림으로 표현하면 다음과 같다.

3	A	B	D	D						
---	---	---	---	---	--	--	--	--	--	--

여기서 추가되는 레코드가 있으면 과거의 D 레코드 자리는 다른 레코드로 차게 된다. 예를 들어, A 와 B 레코드 사이에 A1 레코드를 삽입하면 다음과 같이 변하게 된다.

4	A	A1	B	D						
---	---	----	---	---	--	--	--	--	--	--

여기에 E, F, G, H, I, J 라는 레코드 6 개를 추가하면 다음과 같이 데이터 블록이 모두 차게 된다.

10	A	A1	B	D	E	F	G	H	I	J
----	---	----	---	---	---	---	---	---	---	---

그런데, 여기서 K 레코드 하나를 더 추가하면 하나의 데이터 블록이 더 할당된다. 이때 BDE 는 이전 블록의 마지막 레코드를 새 블록으로 옮겨 오는데, 이렇게 해서 블록의 마지막 슬롯을 비워 둔다. 이렇게 하는 이유는 A 와 J 사이에 레코드가 삽입될 경우 새로운 블록을 필요없게 하기 위해서이다. K 라는 레코드를 하나 추가할 경우 다음과 같이 배열된다.

9	A	A1	B	D	E	F	G	H	I	J
10	J	K								

여기에 두개의 레코드를 E 와 F 레코드 사이에 추가한다고 가정하자. 이때 첫번째 데이터 블록에 사용 가능한 슬롯이 마지막에 하나 밖에 없으므로, E1 만 E 와 F 사이에 위치하게 되고, 마지막 슬롯에는 I 레코드가 들어가게 된다. 이 상황에서 E2 가 삽입되면 데이터 블록이 어쩔 수 없이 분리되어야 한다. 그렇지만, 블록의 마지막에서 삽입된 것이 아니므로 이러한 분리에는 삽입 지점(insertion point)에서 일어나게 되며, 이 삽입 지점 뒤에 위치하는 모든 레코드는 다음과 같이 새로운 데이터 블록으로 옮겨가게 된다.

6	A	A1	B	D	E	E1	F	G	H	I
2	J	K								
5	E2	F	G	H	I					

이 경우에 새로운 블록은 테이블의 마지막에 위치하게 되므로, BDE 는 이들 블록의 순서를 정확하게 유지하기 위해서 forward, backward chain 번호를 새로 매기게 된다. Forward chain 의 경우 1, 3, 2 의 순서를 가지게 되며, backward chain 의 경우 2, 3, 1 의 순서를 가지게 된다.

데이터베이스 데스크탑에서 테이블을 restructure 하면 BDE 는 빈 레코드 슬롯을 제거하지 않으며, 물리적으로 데이터 블록을 재배치하지도 않는다. 그러나 Pack Table 옵션을 선택

하면 테이블에서 사용되지 않는 공간을 모두 없애버리고 새롭게 테이블을 구성하는데, 이때 각각의 블록은 모두 레코드로 꼭 차게 되고, 자유 블록 chain 은 비워진다. 또한, forward chain 에 입각해서 모든 데이터 블록의 순서를 새롭게 구성한다.

이러한 방식은 키가 결정된 테이블에 적용되는 것이다. 키가 없는 테이블의 경우에는 특정 순서에 맞추어 데이터의 삽입이 결정될 필요가 없으므로, BDE 는 각 슬롯을 사용이 가능한 형태로 버려두지 않는다.

## 필드의 형

각 레코드는 1~255 필드로 구성된다. 파라독스에서는 17 가지 종류의 필드가 존재하는데, 이들에게는 문자열을 나타내는 Alpha, Number, Money, LongInt, Data 등에서 부터 OLE, Binary 등까지 다양하게 존재한다.

이들의 크기는 대체로 필드를 정의할 때 결정되는데, Memo 형의 경우 BDE 는 지정된 처음 n 바이트는 .DB 파일에 저장되며, 나머지 내용은 .MB 파일에 저장된다. Formatted Memo, Graphic, OLE, Binary 형의 경우에도 메모 파일과 마찬가지로 저장되는데, 사실 .DB 파일에 저장되도록 지정하는 n 바이트는 거의 쓸모가 없기 때문에, 0 으로 지정하는 것이 가장 좋다. Memo 형의 경우에는 이와 달리 .DB 파일에 저장된 문자를 이용해서 검색을 할 때 사용될 수 있다.

텔파이는 OLE, Binary, Byte 필드의 내용을 보여줄 수 있는 컴포넌트를 제공하지 않는다. 이들은 OLE 를 사용하는 어플리케이션이나 데이터 스트림을 이용해서 간접적으로 폼에 보여지게 된다. 그러므로, 이들 필드에 대해서는 약간의 코딩이 필요하다. 예를 들어, 실험용 기계에서 RS-232C 포트를 통해서 나온 데이터를 Byte 필드에 저장했다가 이를 스트림으로 읽어서 해석한 후 보여주는 등의 작업을 해 주어야 한다.

## 레코드와 테이블 크기 계산

다음과 같은 구조의 테이블이 있다고 하자. 이 테이블의 레코드의 크기와 테이블의 크기 등을 계산해 보자.

필드 이름	종 류	크 기	비 고
Key	LongInt	4	Primary Key
ID	Alpha 8	8	ID
Password	Alpha 8	8	패스워드
Name	Alpha 10	10	이 름
Address	Alpha 100	100	주 소
BirthDay	Date	4	생년월일



이 경우에 레코드의 크기는  $4 + 8 + 8 + 10 + 100 + 4 = 133$  바이트가 된다. 그런데, 이 테이블은 키를 가지고 있으므로, BDE는 최소한 3개의 레코드를 가질 수 있는 가장 작은 블록 크기(1k 제외, 1k는 과거 버전의 파라독스에서만 사용한다)를 사용한다. 이 경우에는 레코드 3개를 합쳐도 399 바이트에 불과하므로 2k(2048) 바이트 크기의 블록이 데이터 블록으로 사용된다. 각 블록의 처음 6 바이트는 사용할 수 없으므로, 실제로 사용 가능한 공간은 2042 바이트가 된다.

그러므로, BDE가 15개의 레코드를 한 블록에 채워 넣으면 1995 바이트가 사용되고, 블록의 끝에 남는 47 바이트는 버려진다. 참고로, 테이블의 필드 구조를 바꿀 경우 3 바이트 크기까지의 필드를 추가하면 15 레코드가므로 블록에 45 바이트까지 가능하기 때문에, 테이블의 크기를 증가시키지 않아도 되지만, 4 바이트가 넘는 필드를 추가할 경우에는 47 바이트를 넘어버리기 때문에 15 레코드를 한 블록에 넣을 수 없게 되어, 테이블의 크기가 왕창 증가하게 된다.

테이블의 크기는 앞서서도 언급했지만 삽입과 삭제에 의해서 테이블의 크기가 변화할 수가 있다. 그러므로, 다음의 계산은 테이블을 pack 했다고 가정한 것으로 가장 작은 크기라고 말할 수 있다. 계산 방법은 단순히 레코드의 수를 15로 나누어 올림하고, 각 블록 당 2k이므로 2k를 곱하면 얻을 수 있다. 그리고, 여기에 헤더에 해당되는 2k를 더해주면 된다. 여기에 입각해서 계산을 해보면 100 레코드면 16,384 바이트가 되며, 10000 레코드면 1,368,064 바이트가 된다.

## 인덱스의 역할

원래의 관계형 모델에서는 인덱스라는 것이 존재하지 않는다. 인덱스는 파일 포맷의 일부로 수행속도를 향상시키기 위해 디자인 된 것이다.

인덱스란 정렬된 리스트의 일종으로, 파라독스 테이블에 대한 인덱스를 생성하면 BDE는 각 레코드에 대한 필드의 값을 담은 독립된 파일을 생성한다. 이 값들은 문자의 순서나 숫자의 순서에 따라 정렬되어 있고, 테이블에서의 실제 위치를 가리키는 포인터를 가지고 있다.

예를 들어, 다음과 같은 테이블이 있다고 가정하자.

레코드 위치	필드의 내용	레코드 위치	필드의 내용
1	지훈	4	준완
2	현목	5	대옥
3	종상	6	현호

이 테이블의 필드를 지정해서, 인덱스를 만들면 인덱스에는 다음과 같이 저장된다.

대옥	5	준완	4	현목	2
중상	3	지훈	1	현호	6

어플리케이션에서 BDE 에 특정 값을 검색하라고 요청하면, 그 필드에 대한 인덱스가 있으면 BDE 는 정렬되어 있는 인덱스를 이용해서 값을 찾은 뒤, 포인터를 읽어서 실제로 레코드의 위치를 찾아낸다. 이때 정렬이 되어 있기 때문에 BDE 는 이진 검색 기법을 사용할 수가 있으므로, 순차적으로 검색하는 방법에 비해서 훨씬 효율과 수행속도가 빠르다.

## 1 차 인덱스 (Primary Index)

관계형 모델에서는 레코드의 유일성을 보장하기 위해서 하나 이상의 필드를 1 차 키(primary key)로 사용한다. 이 키는 테이블에서 하나만 설정할 수 있으며, 검색 작업 등을 할 때 빈번히 사용된다. 또한, 다른 테이블과의 연결 시에 외부 키(foreign key)로 사용되기도 한다.

1 차 키가 이렇게 중요하기 때문에, BDE 는 1 차 키에 대한 인덱스를 생성하고 이를 1 차 인덱스라고 한다. 1 차 인덱스는 .PX 파일에 저장되며 테이블 내의 모든 레코드에 대한 엔트리를 포함하지 않는다는 측면에서 다른 인덱스와는 다르다.

앞에서도 설명했지만 1 차 키가 있는 모든 파라독스 테이블은 데이터 블록에 최소한 3 개의 레코드를 가지고 있어야 한다. 이 경우에 블록 내부에서는 모든 레코드는 1 차 키의 순서에 따라서 배열된다는 것은 이미 설명했다.

1 차 인덱스를 최소한의 크기로 유지하기 위해서, BDE 는 1 차 인덱스 파일에 각 블록의 첫 번째 레코드 값만을 저장한다. BDE 가 레코드를 검색할 때에는 검색하고자 하는 값을 넘지 않는 가장 큰 값을 가진 블록이 선택되며, 찾고자 하는 레코드는 그 블록에 존재하게 된다. 각 블록 내에서는 1 차 키에 따라 레코드가 순차적으로 배열되어 있으므로 여기에서 검색을 하면 된다. 달리 말하면 2 차레의 이진 트리 검색을 하게 되는 것이다.

이런 형태의 인덱스 방식의 잇점은 인덱스의 크기를 많이 줄일 수 있으며, 검색 속도를 향상시킬 수 있다는 것이다. 또한, 인접한 키 값의 레코드는 실제 물리적으로도 가깝게 위치하므로 디스크 접근 속도도 빠르다.

## 2 차 인덱스 (Secondary index)

1 차 키 이외의 필드에 대한 작업을 많이 할 때에는 검색과 필터를 사용할 때의 수행속도 향상을 위해서 그 필드에 새롭게 인덱스를 걸어주기도 한다. 이러한 인덱스를 2 차 인덱스라고 한다. 2 차 인덱스는 하나 이상의 필드로 구성되며, 오름-내림차순 또는 혼합형을 선택할 수 있고, 대소문자를 가리거나, 중복의 허용 여부 등을 설정할 수 있도록 되어 있다.

이러한 2 차 인덱스는 .Xnn, .Ynn 이라는 확장자를 가지는 2 개의 파일에 저장된다. 이때 하나의 필드에 대해, 오름차순이며 대소문자를 가리는 non-unique 인덱스이면 nn 은 01 부터 FF 까지의 16 진수 문자인데, 테이블이 최고 255 개의 필드를 가질 수 있으므로 각 테이블의 모든 필드를 감당해낼 수 있게 된다.

그런데, 복합 필드를 인덱스로 사용하거나 내림차순이나 혼합형의 정렬 순서를 가지거나 대소문자를 가지는 경우, 또는 unique 인덱스인 경우에는 nn 이 G0 에서부터 하나씩 증가하면서 확장자를 가지게 된다. 예를 들어 이런 새로운 스타일의 인덱스가 처음 16 개까지는 확장자의 nn 이 G0 에서 GF 로 결정되며, 다음 인덱스는 H0 가 된다.

## 2 차 인덱스 파일의 구조

.Xnn 파일은 일종의 파라독스 테이블 구조이다. 이 테이블의 구조는 다음과 같다.

Secondary Key	Primary Key Columns			Hint

Secondary Key 필드는 실제로 인덱스된 값이 들어 있으며, 하나 이상의 필드가 인덱스된 경우에는 두 필드를 합친 값이 들어가 있다. 이때 Secondary Key 필드의 값은 유일하지 않다. 대소문자를 가리지 않는 인덱스의 경우에는 대문자로 변환되어 들어가게 된다. Primary Key Columns 필드에는 각 필드에 대한 인덱스 값을 찾을 수 있는 실제 테이블의 1 차 키가 저장된다. 마지막의 Hint 필드에는 이 1 차 키를 찾을 수 있는 물리적인 블록의 번호가 적혀있다.

.Xnn 파일이 일종의 테이블이므로 .DB 파일과 마찬가지로 1 차 키를 가지고 있다. 이때 Secondary Key 필드는 유일하지 않으므로, 원래 테이블의 1 차 키를 Xnn 테이블의 1 차 키로 사용한다. 그렇다면 이 1 차 키의 1 차 인덱스에 해당되는 파일은 무엇일까? 이것이 바로 .Ynn 파일이다. 즉, .DB 파일과 .PX 파일의 관계와 .Xnn, .Ynn 파일의 관계가 비슷한 것이다.

이렇게 복잡한 구조를 가지고 있으므로 .Xnn 파일은 .PX 파일에 비해서 훨씬 크기가 크다. 심한 경우에는 테이블의 각 레코드에 대한 엔트리를 포함하므로 .DB 파일 보다도 클 수가 있다. 그렇지만, .Ynn 파일의 크기는 작다.

## 2 차 인덱스의 동작

그러면, 예를 들어서 실제로 BDE 가 2 차 인덱스를 어떻게 사용하는지 알아보도록 하자. 설정된 2 차 인덱스는 TTable 컴포넌트에 의한 검색 뿐만 아니라 TQuery 컴포넌트에서 SQL 문장으로 실행해도 효과적으로 사용된다. 다음의 SQL 문장을 보자.

SELECT \* FROM Customer WHERE Country = "Korea"

여기서 Country 필드는 1 차 키가 아니지만, 2 차 인덱스를 걸어놓았다고 하자. 그러면, BDE 는 SQL 문장을 실행하면서 2 차 인덱스를 사용한다. 이때 실제로 BDE 가 수행하는 작업은 다음과 같다.

1. .Ynn 파일을 열고 .PX 파일을 사용할 때처럼 이진 검색 기법을 이용하여 .Xnn 파일에서 "Korea" 엔트리가 들어있는 시작 블록을 찾는다.
2. .Xnn 파일에서 앞에서 찾은 시작 블록으로 가서 순차적으로 "Korea" 엔트리와 일치하는 레코드를 찾는다.
3. 레코드를 찾으면 Secondary Key 필드가 "Korea"가 아닌 레코드가 나올 때까지 순차적으로 레코드를 계속 읽는다.
4. 조건에 맞는 각 레코드의 Primary Key 정보와 Hint 필드에서 실제 테이블의 정보를 가져 온다.
5. BDE 는 테이블의 Hint 필드에 저장된 블록으로 가서 해당 레코드가 나올 때까지 순차적으로 검색한다. 이때 블록 안에서 레코드가 발견되지 않으면 .PX 파일에서 이진 검색을 하게 되고, 해당되는 블록을 찾으면 다시 .DB 파일의 블록으로 가서 레코드를 순차적으로 검색한다.
6. 찾은 레코드를 가져온다

## 타당성 검사 (Validity Check)

타당성 검사는 필드에 값을 추가하거나 수정할 때 BDE 에 의해서 수행되는 간단한 비즈니스 규칙(business rule)을 의미한다. 파라독스 파일 포맷에서 지원하는 타당도 검사 항목에는 다음과 같은 것들이 있다.

1. 값이 요구되는지 여부 (Null 허용 여부) : Required
2. 필드의 최대값 : Maximum
3. 필드의 최소값 : Minimum
4. 레코드가 삽입될 때 디폴트 값 : Default
5. 포맷 마스크 : Picture

필드에 따라서 이러한 타당성 검사를 못하는 경우가 있는데, 많은 필드가 이들 모두를 지원한다. 그런데 AutoIncrement 형의 필드는 Minimum 만을 지원하며, 각종 Blob 필드 들은 Required 만 지원한다. Logical 형의 필드는 Required 와 Default 를 지원한다.

이러한 타당성 검사를 이용하면 어플리케이션 레벨에서가 아니라 데이터베이스 레벨에서 비즈니스 규칙을 정의할 수 있으며, 이로 인해 특정 어플리케이션에 종속적이지 않은 비즈니스 규칙을 활용할 수 있게 된다.

이러한 타당성 검사에 대한 값들은 .VAL 파일에 저장된다. 그런데, 이 파일에 대한 이진 파일 포맷이 공개되지 않았으므로 직접 수정은 불가능하다.

## 참조 무결성 (Referential Integrity)

테이블에서 하나 이상의 필드가 다른 테이블의 1 차 키를 참조하고 있을 때, 이런 필드를 외부 키(foreign key)라고 한다. 외부 키는 데이터베이스 내에 있는 다양한 테이블 간의 데이터 연관성을 유지하는데 매우 중요한 역할을 한다. 참조 무결성 규칙은 두 개의 테이블이 1 차 키와 외부 키의 관계로 연결되어 있을 때, 하나의 테이블의 모든 외부 키의 값은 다른 테이블의 1 차 키의 값과 반드시 일치해야 한다는 것이다.

이러한 참조 무결성은 각 테이블에 작업을 할 때 깨질 수 있는 상황들이 발생한다. 특히 데이터를 삭제하거나 업데이트할 때가 문제가 된다. 이럴 때 참조 무결성을 지키기 위한 방법으로는 크게 세가지가 있다. 첫째 방법은 변화가 일어난 부분을 데이터베이스 내부에서 모두 적용시키는 방법이다. 즉, 필드 값이 삭제된 경우 연결된 모든 테이블의 해당 레코드를 삭제하거나, 데이터 값을 변경한다. 이를 ‘cascade’ 라고 한다. 두번째로 참조가 되고 있는 레코드의 삭제나 업데이트를 금지하는 경우이다. 이를 ‘prohibit’ 이라고 한다. 마지막 방법은 연결된 값을 null 로 설정하는 경우이다. 이를 ‘null’ 이라고 한다.

파라독스 파일 포맷에서는 이들 방법 중에서 몇 가지 방법 만을 사용할 수 있도록 되어 있다. 데이터를 업데이트할 경우에는 cascade 와 prohibit 중에서 하나를 선택할 수 있다. 또한, 데이터를 삭제할 때에는 prohibit 만을 지원한다. 이것의 의미는 개발자가 반드시 연결된 1 차 키가 있는 레코드를 삭제할 때, 이와 연결된 외부 키가 있으면 직접 이들 레코드를 삭제한 뒤에야 삭제가 가능하다는 것이다.

파라독스 파일 포맷의 가장 부족한 한계점을 지적한다면 바로 이와 같은 참조 무결성에 대한 불완전한 지원을 꼽을 수 있다.

이러한 참조 무결성에 대한 정보도 .VAL 파일에 저장된다.

## 암호화된 테이블

파라독스 파일 포맷에는 패스워드를 통하지 않고는 테이블을 보지 못하게 하는 암호화 기능을 가지고 있다. 파라독스 테이블에는 테이블에 조작을 하기 위해서 여러 단계의 패스워드 레벨을 요구한다.

암호화를 하기 위해서는 마스터 패스워드를 정해야 한다. 데이터베이스 데스크탑의 Create/Restructure 대화상자에서 이를 설정할 수 있다. 패스워드는 128 자까지 입력할 수

있으나, 처음 31 자만 의미를 가지는데, 대소문자를 가린다. 이 패스워드는 테이블의 소유자를 정의한다. 이 패스워드를 알아야 부가적인 패스워드를 바꿀 수 있는 권한이 있다. 부가 패스워드(auxiliary password)는 마스터 패스워드를 입력한 후에야 정의할 수 있다. 부가 패스워드를 입력하면 테이블과 필드에 대한 접근 권한을 일일이 정할 수 있는데, 테이블에 대한 접근 권한에는 다음과 같은 것들이 있다.

권 한	설 명
Read Only	테이블을 볼 수는 있지만 데이터나 구조는 수정할 수 없음
Update	테이블을 볼 수 있고, non-key 필드의 수정은 가능하지만, 레코드를 추가, 삭제하거나 키 필드의 수정은 불가능하다.
Data Entry	테이블을 볼 수 있고, non-key 필드의 수정, 레코드 삽입이 가능하지만, 레코드 삭제나 키 필드의 수정은 불가능하다.
Insert & Delete	레코드를 삽입, 삭제, 수정할 수는 있지만, 테이블의 구조를 바꿀 수는 없다.
All	마스터, 부가 패스워드를 바꾸는 것을 제외한 모든 권한을 가짐

필드에 관한 권한의 종류에는 None, Read Only, All 의 세 가지가 있는데, None 은 필드를 볼 수가 없기 때문에, 그 필드의 데이터는 숨겨진다. Read Only 는 필드의 내용을 볼 수는 있지만 수정이 불가능한 것이고, All 은 수정까지 모두 가능한 권한이다.

마스터 패스워드는 암호화된 형식으로 테이블의 헤더에 저장되며, 부가 패스워드는 마스터 패스워드를 이용해서 암호화하여 테이블의 헤더에 저장된다.

실제로 패스워드는 버퍼를 이용해서 저장된다. BDE 를 이용해서 파라독스 테이블을 열게 되면, 디폴트 세션이 생성된다. 이러한 세션에 접근하려면 TSesstion 컴포넌트의 프로퍼티, 메소드, 이벤트를 사용한다. 세션 객체는 하나 이상의 데이터베이스에 가상 연결을 하고, 각 세션은 데이터베이스에 대해 다른 사용자로 취급된다. 개발자는 추가적인 TSession 객체를 추가하고, 이를 이용해서 가상 유저를 관리할 수 있다.

BDE 는 패스워드에 대한 버퍼를 세션에 저장한다. 이 버퍼는 파라독스 테이블에서 사용되는 모든 패스워드를 담을 수 있을 정도로 큰데, 하나의 세션에 모두 25 개까지 담을 수 있다. 만약 사용자가 패스워드를 입력하면, 이것이 버퍼에 추가된다. 마찬가지로 어플리케이션에서 코드로 패스워드를 추가해도 같은 버퍼에 추가된다. 만약 패스워드를 버퍼에 두 차례 추가하면, 두 개가 모두 버퍼에 추가되기 때문에 한 개의 인스턴스가 제거되어도 여전히 암호화된 테이블에 접근할 수 있다.

어플리케이션에서 파라독스 테이블에 접근할 필요가 있으면, BDE 는 버퍼에 있는 패스워드 목록과 테이블의 헤더에 있는 패스워드를 비교해서 권한을 설정하게 된다. 이때 여러 개의 패스워드를 이용할 경우 상위의 권한을 얻게 된다.

암호화 테이블을 사용하면 BDE 의 작업 속도가 약 10~15% 정도 속도가 느려진다. 이는

BDE 가 데이터에 접근할 때 마다 암호화와 해독 알고리즘을 적용하기 때문이다.

또한, 암호화 테이블은 잘 압축이 되지 않는데, 이는 정상적인 파라독스 테이블의 경우에는 반복되는 데이터가 많은데 비해, 암호화 테이블에는 빈 공간과 반복되는 바이트가 거의 없기 때문이다.

## BDE 와 네트워크 잠금 (Network Locking)

BDE 는 파일을 이용해서 파라독스 테이블과 레코드 잠금에 대해서 관리를 하게 된다. 이 파일들은 필요할 때 BDE 에 의해서 생성되며, 자동으로 관리된다.

PDOXUSRS.NET 파일은 파라독스 네트워크 컨트롤 파일로, 네트워크의 사용자에게 대한 여러가지 관리에 관여하게 된다. 파라독스 테이블을 네트워크 상에서 접근하게 되는 모든 어플리케이션에서는 이 파일을 참조하게 되며, 전체 네트워크에는 이 파일이 하나만 존재한다. 이 파일의 위치는 IDAPI.CFG 파일에 저장되어 있으며, 이 파일은 각 사용자에게 의해 관리되거나 네트워크 상에서 공유된다. 이를 설정하려면 BDE Administrator 를 실행하여 Drivers 페이지에 있는 파라독스 드라이버에 대한 설정에서 Net Dir 을 이용해서 정의할 수 있다. 이 디렉토리를 네트워크 컨트롤 디렉토리라고도 한다.

PDOXUSRS.NET 파일이 지정된 위치에서 발견되지 않으면, BDE 에 의해 자동으로 생성된다. 이 파일에서는 최고 300 명의 가상 유저의 관리가 가능하다. 텔과이 어플리케이션에서 PDOXUSRS.NET 파일의 위치는 TSession.NetFileDir 프로퍼티를 이용해서 파악하거나 설정할 수 있다. 만약 다른 PDOXUSRS.NET 파일에 의해 관리하려 하면 새로운 세션 객체를 생성해야 한다.

디렉토리에 있는 테이블들에 대한 접근 권한 관리에는 PDOXUSRS.LCK 파일이 이용된다. 이 파일은 파라독스 테이블이 위치한 각 디렉토리에 BDE 에 의해 첫번째 사용자가 테이블에 접근할 때 생성된다. 이 파일에는 다음과 같은 세 종류의 파일이 있다.

1. 공유된 디렉토리에서는 테이블과 레코드에 대한 잠금(locking) 정보를 가지고 있다. 즉, 어떤 사용자와 세션이 잠금을 걸고 있으며 그 종류가 무엇인지를 저장하고 있다. 사용자가 테이블에 잠금을 걸려고 하면, 이 파일에서 잠금 정보를 확인하고 잠금이 존재하지 않으면 새로운 잠금을 설정하고 이를 파일에 기록한다.
2. 공유되지 않는 디렉토리에서는 이 파일에 특정 플래그가 설정되어 있어서, BDE 가 다른 사용어나 세션이 파라독스 테이블에 접근하지 못하도록 한다. 이러한 디렉토리에는 특정 사용자만 접근하게 하는데, TSession.PrivateDir 프로퍼티에서 이 디렉토리를 설정할 수 있다.
3. 공유가 되지만, 읽기 전용인 디렉토리에서는 일단 잠금이 설정되면 다른 잠금이 설정될 수 없으므로, 이 디렉토리의 모든 테이블에 접근할 수 없게 된다. 이를 잘 활용하면 작업의 수행속도를 증진시킬 수 있다.

디렉토리 잠금은 보통 PARADOX.DRO 파일을 사용하는데, 이 파일을 생성할 때에는 다음과 같은 코드를 사용한다.

```
var
    Lock_Path: Array [0..DbiMaxTblNameLen] of Char;
begin
    StrPCopy(Lock_Path, 'C:\WPath\WPARADOX.DRO');
    Check(DbiAcqPersistTableLock(Database1.Handle, Lock_Path, szPARADOX));
end;
```

DbiAcqPersistTableLock() 함수는 존재하지 않는 테이블에 까지 디렉토리에 대한 잠금을 걸 수 있으며, 이를 해제할 때에는 DbiRelPersisTableLock() 함수를 사용한다.

이와 같이 추가적인 파일을 이용해서 잠금을 제어하게 되면, 추가적인 정보를 여러 상황에서 사용할 수 있다. 예를 들어, 테이블에 접근할 때 잠금이 존재할 경우 BDE 를 이용해서 잠금을 걸고 있는 사용자 등을 알아낼 수 있다.

## 테이블 잠금 (Table Lock)

파라독스 테이블에 대한 잠금에는 Exclusive Lock, Write Lock, Read Lock, Open Lock 의 4 가지 종류가 있다. Exclusive Lock 은 테이블에 대해서 잠금을 건 사용자만 접근할 수 있는 경우로 보통 테이블을 생성하거나, 구조를 바꿀 때 사용된다. Write Lock 은 데이터를 읽을 수는 있지만 이를 수정하지 못하게 하는 잠금으로, 테이블을 복사할 때 쓰는 쪽의 테이블에는 Write Lock 이 걸려 있어야 한다. Write Lock 은 한 테이블에 하나만 걸 수 있다. Read Lock 은 다른 사용자가 Read Lock 을 걸고 있지 않을 경우에 테이블에 쓰기를 할 수 있다. 모든 사용자가 데이터를 읽을 수 있다는 점에서는 Write Lock 과 같지만, 이 경우에는 한 테이블에 대해 여러 개의 Read Lock 이 걸릴 수 있으며, 다른 사용자가 Read Lock 을 걸고 있다면 쓰기를 할 수 없다. 마지막으로 Open Lock 은 최하 레벨의 잠금으로 다른 사용자도 테이블에 대한 여러가지 권한을 가질 수 있다. 단지 Exclusive Lock 이 설정된 테이블에만 접근할 수 없다.

이들 잠금에 대한 상호작용은 다음과 같다. 각 잠금에서 허용되는 잠금에는 \* 표시를 하였다.

	Exclusive Lock	Write Lock	Read Lock	Open Lock
Exclusive Lock				
Write Lock				*



Read Lock			*	*
Open Lock		*	*	*

부연해서 설명하면 Open Lock 은 여러 명의 유저가 동시에 테이블을 볼 때 서로 가질 수 있다. 그리고, Read Lock 역시 여러 명이 동시에 걸 수 있는데, 이 경우에는 각각의 Read Lock 이 다른 사용자가 테이블에 데이터를 쓸 수 없게 한다. Write Lock 은 쓰기 권한을 잠금을 건 사용자에게만 부여하지만, 다른 사람들은 Open Lock 을 통해 데이터를 볼 수 있다.

텔파이에서는 테이블을 공유 모드(Share Mode)나 전용 모드(Exclusive Mode)로 열 수 있으며, 이는 TTable.Exclusive 프로퍼티를 가지고 조절할 수 있다. 이 프로퍼티의 디폴트 값은 False 이며 이 경우 테이블은 Open Lock 으로 열리게 된다. 그리고, 테이블이 열리기 전에 True 로 설정될 경우에는 Exclusive Lock 을 걸게 되므로 다른 어플리케이션에서 테이블에 접근을 할 수 없다.

그리고, TTable.Lock 메소드를 이용해서 잠금을 걸 수 있는데 여기에서 사용할 수 있는 LockType 은 [liReadLock, ltWriteLock] 중의 하나이다. 이미 걸려있는 잠금을 해제할 때에는 TTable.Unlock 메소드를 사용한다.

어떤 경우에는 테이블 잠금을 걸 수 없는 경우가 있는데, 다음과 같은 경우가 있다.

1. 디렉토리가 로컬 하드 디스크에 있으면서 공유되지 않는 경우
2. 사용자의 Private Directory 에 위치하여 다른 사용자가 이 디렉토리에 접근할 수 없는 경우
3. 디렉토리가 CD-ROM 같이 읽기 전용 장치에 있는 경우
4. 테이블의 파일 속성이 읽기 전용일 때
5. PDOXUSRS.LCK 파일이 디렉토리 잠금으로 지정한 경우

## 레코드 잠금 (Record Lock)

사용자가 테이블에 있는 레코드를 편집하기 시작하면, 파라독스는 각 잠금 파일에 레코드 잠금을 걸기 시작한다. 그렇지만 보통의 경우에는 테이블 잠금은 Open Lock 을 계속 유지하고 있게 된다. 사용자가 새로운 레코드나 변경된 레코드를 post 하는 도중에는 테이블의 잠금이 Open Lock 에서 Write Lock 으로 변경된다.

BDE 는 한번에 테이블 당 255 레코드까지 잠금을 걸 수 있다. 이러한 잠금은 모두 exclusive 형태이기 때문에 오직 하나의 유저만 레코드를 편집할 수 있으며, 다른 유저들은 레코드가 편집되는 상태를 볼 수는 있게 된다.

## 정 리 (Summary)

이번 장에서는 텔파이가 기본적으로 사용하는 파라독스 데이터베이스 파일에 대한 포맷과 파라독스의 구조를 살펴 보았다. 직접적으로 쓸 수 있는 정보는 아니지만, 데이터베이스 파일을 이해하고, 이들이 어떻게 관리되는지를 이해하는 데에는 커다란 도움이 되었을 것으로 믿는다.

다음 장에서는 멀티-tier 어플리케이션을 작성하는 방법을 알아볼 것이다.

## 멀티-tier 데이터베이스 어플리케이션의 제작

이번 장에서는 대규모 데이터베이스 어플리케이션을 작성할 수 있는 멀티-tier 데이터베이스 어플리케이션에 대해서 알아본다. 기본적으로 멀티-tier 데이터베이스 어플리케이션은 클라이언트에서 동작하며 데이터를 조작하기 위한 인터페이스를 가지고 있는 클라이언트 프로그램과 실제 DBMS 를 조작하기 위한 리모트 데이터베이스 서버, 그리고 네트워크 상에 연결되어 클라이언트 프로그램의 데이터에 대한 처리를 수행하게 되는 어플리케이션 서버의 3 가지 구조로 나누어볼 수 있다.

이렇게 설명하는 모델이 3-tier 모델이다. 텔과이 4 에서는 이러한 모델을 지원하기 위한 여러가지 컴포넌트를 제공한다. 텔과이 4 에서 멀티-tier 모델을 구현하기 위해서는 사용하는 네트워크 프로토콜에 따라 크게 몇 가지로 세분할 수 있다. TCP/IP 의 기본적인 소켓을 사용해서 이를 구현할 수도 있고, MS 에서 제공하는 DCOM 을 사용할 수 있다. 또한, DCOM 의 트랜잭션과 리소스 문제를 해결하기 위해 제공되는 MTS 를 사용하거나, ORB 를 이용한 CORBA 를 사용할 수도 있다. 그 밖에도 Inprise 에서 제공되는 OLEnterprise 나 Entera 와 같은 미들 웨어를 사용할 수 도 있으며, AS400 용 제품을 이용하여 DB2 를 이용할 수도 있다.

이러한 여러가지 접속 방법은 기본적으로 MIDAS(Multi-tier Distributed Application Services Suite)라는 기술을 이용하게 된다.

### 멀티-tier 데이터베이스 어플리케이션의 필요성

무엇 때문에 개발자는 기존의 1-tier 나 2-tier 환경에 어떠한 점에 만족하지 못하고, 새로운 멀티-tier 환경의 어플리케이션을 만들게 되었는가? 그 이유를 알아보자.

가장 큰 장점으로서는 일을 구분하여 쉽게 개발과 유지보수가 된다는 점을 들 수 있다.

클라이언트 어플리케이션 작성 시에는 사용자의 사용자 인터페이스와 데이터의 제시 방법에 대한 내용을 구축하는데 중심을 맞출 수 있으며, 일의 단위를 구분하기 편리하다. 그리고, 어플리케이션 서버의 작성 시에는 많은 클라이언트의 요구에 따라 비즈니스 규칙이 적용된 데이터를 전송하거나 클라이언트에서 변경된 자료를 역시 비즈니스 규칙이 적용된 데이터를 데이터베이스 서버에 전송하는 내용만 작성하면 된다.

이렇게 일을 구분하여 작업할 수 있으므로, 팀 단위의 프로젝트가 쉬운 것은 두말할 나위 없으며, 모든 클라이언트 프로그램이 적용될 비즈니스 규칙이 구현된 어플리케이션 서버의 변경작업만으로 클라이언트 프로그램을 재배포하거나 재작성할 필요가 없이 해결이 가능하다. 마찬가지로, 데이터베이스 서버의 변경 작업이 발생할 경우에는 해당 계층의 변경만으로 작업을 최소화할 수 있다.

그 이외의 장점으로서는 다음과 같은 것들이 있다.

- Thin 클라이언트

사용자의 PC 에 설치된 클라이언트 프로그램은 1-tier 나 2-tier 어플리케이션에 비해 그 크기가 훨씬 작은 가벼운 프로그램으로 대치가 가능하다. 1-tier 나 2-tier 데이터베이스 어플리케이션에서는 해당되는 DBMS 에 접속하기 위한 무거운 해당 데이터베이스 드라이버와 유틸리티를 가지고 있어야 한다. 하지만 멀티-tier 모델에서는 DB 클라이언트, DLL 이라는 하나의 파일만 있으면 해결이 된다. 아마도 텔파이로 개발된 프로그램을 만들어본 개발자라면 이 고통을 이해할 수 있을 것이다. 필자도 텔파이 1 시절에 개발한 소프트웨어를 보면 간단한 2M 상당의 프로그램을 보급하기 위해 프로그램 디스켓 1 장, BDE 디스켓 2 장, 그 당시 사용하던 크리스탈 리포트 런타임 1 장, 이런 식으로 총 4 장의 디스켓을 사용하였다. 그렇지만, 여기에서 BDE 에 해당되는 내용을 줄일 수 있다. 그리고, 클라이언트에서 작업하는 여러가지 비즈니스 규칙에 관련된 코드를 모두 어플리케이션 서버로 이관시킬 수 있기 때문에 클라이언트 어플리케이션 자체의 크기도 많이 줄일 수 있다.

- 분산 환경

멀티-tier 어플리케이션에서는 어플리케이션 서버로 불리는 중간 계층이 서로 다른 서버나 클라이언트에서 구동될 수 있기 때문에, 해당 작업의 처리를 위해서 여러 대로 분리되어 있는 PC 를 사용할 수 있기 때문에 작업의 효율성을 높일 수 있다.

- 보안

1-tier 나 2-tier 모델에서는 데이터베이스가 노출되거나 데이터베이스의 접속이 쉽게 노출되어 데이터의 보안에 취약했다. 이러한 보안 부분을 어플리케이션 서버가 담당하도록 할 수 있기 때문에, 사용자는 DBMS 의 위치를 알 수 없게 할 수 있다.

- 비용의 절감

많은 개발자들은 DBMS 의 역할과 해당 DBMS 의 사용자 수마다 해당되는 고비용에 대한 비용을 많이 줄일 수 있다. 능력 있는 프로그래머라면, 파라독스나 DBASE 로 구성된 1-tier 데이터베이스 어플리케이션을 응용하여 어플리케이션 서버에서 사용하거나 텔파이에 기본적으로 내장되어 있는 로컬 인터페이스를 사용하여 시스템을 구성할 수 있다.

## 미들웨어

BDE 나 ODBC 도 일종의 미들웨어라고 볼 수 있다. 그 밖에도 TP-monitor 계열의 엔테라 나 텍시도 등의 제품도 미들웨어라 불리운다.

미들웨어를 사용하는 이유는 여러가지가 있으나, 대표적인 것은 대규모 데이터베이스 시스템을 구축하는 경우, 그 규모가 MB 나 GB 단위가 아닌 TB 단위의 대용량 데이터베이스인 경우에 단순한 DBMS 로 이를 감당하기 어렵다.

예를 들어, 여러가지 다양한 DBMS 가 사용되는 경우를 들 수 있는데 대표적으로 IMF 를 맞이하여 은행들 간에 합병을 할 때, 두 은행에서 사용되는 DBMS 가 이 기종일 가능성이 높다. 이때 이 기종 DBMS 에 일어나는 트랜잭션을 모두 처리하거나, 여러 개의 DBMS 를 사용하고 저장 프로시저를 이용한 비즈니스 규칙의 관리 등의 데이터 흐름을 단일화 하는 것이 미들웨어의 역할이다.

BDE 나 ODBC 를 살펴보자, 개발자는 하나의 API 를 호출할 수 만 있게 프로그램을 제작한다면 해당 API 에서 지원하는 DBMS 에는 간단한 접속만 변경함으로써 비즈니스 규칙부분의 코딩을 변경할 필요가 없어지게 된다. 그 밖에도, 한 번에 수십, 수백 개의 트랜잭션이 발생하는 은행과 같은 대규모 사이트의 경우 같은 비즈니스 규칙을 처리하는 여러 개의 어플리케이션 서버가 있을 수 있으며, 시스템의 안정성을 위하여 어플리케이션 서버를 확장할 수도 있다. 이런 경우에 작업을 분배하여 움직이는 스케줄링이 필요한데, 이렇게 재배치를 하는 작업을 로드 밸런싱(load balancing)이라고 하며, 이런 역할도 미들웨어가 담당한다.

## MIDAS 의 기술적인 구조

MIDAS 를 사용하는 경우에 클라이언트는 thin 클라이언트를 지향한다. 클라이언트 PC 에는 BDE 가 필요 없고, DB 클라이언트.DLL 파일만 있으면 된다. MIDAS 는 DCOM, CORBA, TCP/IP 소켓, OLEnterprise 를 지원한다.

MIDAS 컴포넌트 세트는 다음의 4 가지 요소로 구성되어 있다.

컴포넌트	내 용
Remote data modules	COM 자동화 서버나 CORBA 서버를 만들기 위한 특별한 데이터 모듈이다. 이 데이터 모듈에는 TProvider, TSession, TDatabase, TQuery, TTable, TStoredProc, TBatchMove 그리고 TTimer, TimageList 등의 컴포넌트가 올라갈 수 있다. 이것은 듀얼 인터페이스 자동화 서버로 IDataBroker 의 인터페이스를 따르고 있다. 또한, 디자인 작업 시에도 동일한 접속환경을 보여준다.
Provider component	서버에 위치하는 컴포넌트로 TQuery, TTable 의 내용을 클라이언트에서 데이터 패킷 단위로 가져갈 수 있도록 인터페이스를 구성하는 컴포넌트
Client dataset component	DBCLIENT.DLL 을 사용하여 서버의 Provider 와 통신하여 데이터 패킷을 읽어 들일 수 있는 컴포넌트

Connection component

네트워크 상의 서버 위치를 찾고 서버와 연계하며 IProvider 인터페이스를 활성화하여 통신을 할 수 있도록 해주는 통신 프로토콜

### 3-tier 어플리케이션

텔파이로 작성하는 기본적인 데이터베이스 어플리케이션은 1-tier 이거나 2-tier 어플리케이션이다. 하지만 대규모의 데이터베이스 어플리케이션을 작성하다 보면 2-tier 에서 저장 프로시저나 트리거보다 더 많은 일을 하는 비즈니스 규칙을 적용할 어플리케이션을 만들 필요가 있게 된다. 텔파이를 이용하면 이런 비즈니스 규칙을 적용한 어플리케이션 서버를 중간에 여러 계층으로 둘 수 있는 멀티-tier 환경을 쉽게 구성할 수 있다.

보통 멀티-tier C/S 어플리케이션은 논리적인 단계로 구분되는 비즈니스 규칙을 통하여 각각의 컴퓨터 시스템상의 데이터 공유 및 분산 처리와 다른 객체와의 통신을 처리하는 것으로 나누어 볼 수 있다. 이러한 환경을 멀티-tier 환경이라고 부르는데 보통은 클라이언트, 어플리케이션 서버, DBMS 로 구분되는 3-tier 환경이 가장 일반적이다.

이런 형태의 3-tier 환경은 다음과 같은 기본적인 구성요소로 이루어져 있다.

#### 1. 클라이언트 어플리케이션

일반적인 사용자들이 자료의 조회 및 수정, 삭제 등의 데이터에 대한 조작을 취할 수 있는 프로그램이다.

#### 2. 어플리케이션 서버

비즈니스 규칙을 적용하는 부분으로 각각의 작업단위 별로 구분되는 작업으로 일반적인 DBMS 의 저장 프로시저 트리거보다 더 많은 작업과 복잡한 작업을 하는 경우가 많다. 보통은 이러한 어플리케이션 서버는 시스템의 부하가 커짐에 따라 여러 개가 존재할 수 있으며 이들의 분배 작업을 위한 미들웨어의 필요성도 커지게 되었다.

#### 3. 원격 데이터베이스 서버

일반적으로 알고 있는 데이터베이스 서버로서 실제 데이터를 저장하고, 단순하고 반복적인 작업을 수행하는 저장 프로시저나 트리거로 정의된 기본 비즈니스 규칙이 적용되어 있다. DBMS 를 선정하는 기준에는 데이터의 규모도 중요하지만 이런 DBMS 가 작업하는 성능도 중요한 고려 사항이 된다.

이 중에서도 어플리케이션 서버가 가장 중요한 역할을 하게 된다. 어플리케이션 서버는 클

라이언트와 DBMS 간의 자료 흐름을 관장하며 중간 작업을 해준다. 실제 3-tier 서버 시스템의 구축이 잘 이루어진 곳에서는 이러한 비즈니스 규칙을 얼마나 잘 어플리케이션 서버에 적용시키고 분배하느냐에 따라 시스템의 성능이 좌우된다. 이러한 어플리케이션 서버를 데이터 브로커(Data Broker)라고 한다.

이때 경우에 따라서는 여러 층의 어플리케이션 서버가 필요할 수 있으며, 이렇게 되면 3-tier 를 넘어선 멀티-tier 어플리케이션이 되는 것이다. 최근의 추세로 본다면 LAN 상에서 동작하는 구조가 아닌 인터넷이나 원격지에서 저속의 모뎀으로 접속하는 시스템을 구성하는 경우도 많기 때문에 이런 경우에는 다양한 층(layer)으로 구성된 멀티-tier 어플리케이션이 필요할 수 있다. 이럴 때에는 트랜잭션을 처리할 수 있는 중간 계층을 하나 더 만들 수 있고, 하나의 DBMS 가 아닌 이 기종의 DBMS 와 접속하는 부분도 만들어 볼 수 있을 것이다.

## 기본적인 구조

사용자 예제는 2-tier 나 3-tier 나 아무런 차이가 없다. 오히려 3-tier 어플리케이션이 수행속도가 더 떨어질 지도 모른다. 그 이유는 소프트웨어에서 하나의 계층이 더 생긴다는 것은 그만큼의 데이터 전송 속도와 수행 속도를 떨어뜨리는 원인이 될 수 있기 때문이다. 예를 들어, 실제 클라이언트의 개수가 20 여대 미만인 경우에 MIDAS 를 쓴다면 이는 ‘답 잡는데 소 잡는 칼을 쓴 꼴’이 될 것이다.

일반적인 2-tier 에서 사용하던 컴포넌트 들을 그대로 3-tier 에서도 사용해도 상관없다. 다만 TDataSet 에 연결되었던 내용 대신에 TClientDataSet 이라는 계층으로 바뀌는 차이가 가장 큰 차이이다. 그리고, 클라이언트에 위치하는 TRemoteServer 컴포넌트는 어플리케이션 서버의 Provider 와 통신을 하게 된다. Provider 클래스는 TProvider 로 선언되어 있다. 이 RemoteServer 컴포넌트를 사용하는 것은 클라이언트의 TClientDataSet 컴포넌트 인데, 독특한 구조를 가지는 경우에는 개발자가 이런 클라이언트 데이터 세트와 Provider 를 재설계할 수도 있다.

TProvider 컴포넌트는 클라이언트의 데이터 요구를 받아서, 데이터 서버로부터 데이터를 얻어온 뒤에 전송할 내용을 모아서 TClientDataSet 컴포넌트에게 전송하며, 반대로 클라이언트에서 자료의 변경사항을 받게 되면 데이터베이스에 이를 적용시키는 일도 처리한다. 이때 에러가 발생하여 적용에 실패하면 그 내용을 다시 클라이언트에 전송하여 에러에 대응하도록 한다.

클라이언트와 어플리케이션 서버가 연결되면 서버의 Provider 과 클라이언트의 클라이언트 데이터 세트는 정해진 프로토콜에 따라 데이터를 교환한다. 이때 Provider 는 이미 알고 있는 데이터 접근 컴포넌트를 사용하여 자료를 DBMS 나 로컬 데이터베이스에서 읽어온다. 이러한 단계를 하나씩 정리해보자.

- 클라이언트가 어플리케이션 서버에 접속할 때 어플리케이션 서버가 동작 중이 아니라면

해당 서버를 구동시킨다 (이 작업은 각각의 프로토콜을 설정하는 부분에서 해당 서버를 등록하는 작업을 거친다). 그리고, 클라이언트는 해당 어플리케이션 서버로부터 Provider 의 인터페이스를 받아온다.

- 클라이언트는 필요한 자료를 어플리케이션 서버에 요청하고 이때에 자료는 모든 자료를 받거나 세션을 통하여 데이터의 일정 부분만 받아 올 수 있다. 일반적으로 전달 받는 자료의 크기를 조절하여 네트워크 트래픽이나 시스템의 수행속도를 높이는 것이 좋다.
- 어플리케이션 서버는 클라이언트의 요청을 받아 DBMS 에서 데이터를 데이터 접근 컴포넌트를 통하여 데이터를 요청한다. 데이터를 읽어내어 해당되는 비즈니스 규칙을 적용한 데이터를 패킷으로 포장하여 클라이언트에 전송한다.
- 클라이언트는 어플리케이션 서버에서 전송된 데이터를 사용자에게 보여줄 수 있도록 화면에 표시한다. 사용자는 화면에 나타난 데이터를 참고하여 새로운 데이터의 첨삭작업을 가하고 변경된 내용을 클라이언트의 log 에 저장한다. 클라이언트는 변경된 log 자료들을 변경시점에 맞추어서 서버에 전송하기도 하고 사용자의 요구에 따라 어플리케이션 서버에 전송하기도 한다.
- 어플리케이션 서버는 클라이언트에서 전송된 데이터를 데이터베이스 서버를 통하여 데이터의 변경작업을 취한다. 이 시점에서 해당 데이터(레코드)를 다른 사용자가 변경 작업을 하고 있지 않다면 이 자료는 어플리케이션 서버에 의해 데이터베이스 서버에 저장된다. 이 상황을 ‘resolving’이라고 부른다.
- 어플리케이션 서버가 해당 ‘resolving’ 과정 중에 문제가 발생하여 정상적으로 처리하지 못하면, 이 데이터는 데이터베이스 서버를 통해 저장할 수 없다. 해당 데이터는 클라이언트에 오류 신호와 데이터를 돌려 보내, 다시 작업을 취하도록 한다. 클라이언트는 ‘resolving’ 과정 중에 문제가 생긴 데이터를 다시 받아 재처리하게 되는데, 이러한 과정을 재조정(Reconcile)이라고 한다.
- 이러한 재조정 과정은 여러가지가 있지만, 보통은 반송되어온 데이터를 다시 어플리케이션 서버에 전송하여 재처리를 취하는 방법이 있을 것이고, 아예 이 데이터를 취소하여 버리고 다시 데이터를 만들도록 하는 방법이 있을 수 있다.

## MIDAS 어플리케이션 만들기

그러면 실제로 간단한 예제 어플리케이션을 이용하여 사용방법을 익히도록 하자.

- MIDAS 클라이언트 어플리케이션의 구조

사용자들이 사용하는 클라이언트 어플리케이션은 일반적인 2-tier 프로그램과 거의 동일하다. 심지어는 TClientDataSet 을 사용함으로써 플랫-파일을 사용하는 1-tier 어플리케이션이 되기도 한다. 다만 차이가 있다면 IProvider 와 인터페이스하는 부분만 틀릴 뿐이다.



MIDAS 를 사용하여 접속하는 방법에는 다음과 같은 4 가지 방식이 있다.

컴포넌트	프로토콜
TDCOMComponent	DCOM
TSocketConnection	윈도우 소켓 (TCP/IP)
TOLEnterpriseConnection	OLEEnterprise (RPCs)
TCorbaConnection	CORBA (IIOP)

## ● 어플리케이션 서버의 구조

어플리케이션 서버로 사용될 수 있는 원격 데이터 모듈에는 다음의 3 가지가 있다. 이들이 모두 IDataBroker 인터페이스를 사용한다.

### 1. TRemoteDataModule

듀얼 인터페이스 자동화 서버로 구성된 모듈로 DCOM, 소켓, OLEEnterprise 를 사용하여 어플리케이션 서버와 연결한다. 보통 MTS 를 만들지 못할 경우에 이 데이터 모듈을 사용한다.

### 2. TMTSDDataModule

이 모듈은 액티브 라이브러리(.DLL)를 사용하는 MTS 데이터 모듈이다. 이 모듈도 DCOM, 소켓, OLEEnterprise 를 사용하여 어플리케이션 서버와 연결된다.

### 3. TCorbaDataModule

CORBA 서버를 사용할 때 이용되는 데이터 모듈이다.

텔파이 4 에서 어플리케이션 서버를 제작하기 위해서는 텔파이 3 부터 제공되던 DCOM 환경을 사용할 수도 있고, DCOM 의 환경을 분산환경의 이점을 살려 이용할 수 있도록 MTS 를 사용하는 방법, CORBA 를 적용하거나 TCP/IP 의 소켓을 사용하는 등의 여러가지 방법을 사용할 수 있다. 이중에서 CORBA 를 이용하면 윈도우 NT 를 사용해야 한다는 한계를 벗어나, 리눅스나 상용 유닉스에서 만들어진 어플리케이션 서버를 사용할 수 있다. 실제로 윈도우 NT 와 유닉스로 만들어진 프로그램의 성능을 비교해 보면, CORBA 를 사용하는 방법이 얼마나 효율적인지 알 수 있을 것이다.

그 이후에 필요한 클라이언트 어플리케이션을 작성한다. 물론, 클라이언트 어플리케이션을 먼저 작성해도 무방하지만, 작업의 편리함을 위해서는 어플리케이션 서버를 먼저 작성한 다

음에 클라이언트를 구성하는 것이 당연히 쉽다. 그래서 보통은 어플리케이션 서버를 클라이언트보다 먼저 작성하게 된다.

- 어플리케이션 서버의 제작

먼저 어플리케이션 서버의 구성에 대해서 살펴보자. 앞에서도 설명하였듯이 여러가지 방식의 어플리케이션 서버를 작성할 수 있다. 그러나, 개발자들은 이를 굳이 구별할 필요가 없다. 왜냐하면, 어떠한 개발방식을 사용하여도 개발하는 구조가 동일하기 때문이다. 다만, 연결되는 프로토콜만 설정하고 해당되는 컴포넌트만 교체하면 된다.

또한, 이러한 어플리케이션 서버의 작성 방법이 약간의 차이점을 제외하고는 일반적인 1-tier 나 2-tier 어플리케이션을 개발하는 방법과 거의 유사하다. 다만, 앞에서 설명한 Provider 인터페이스를 가져야 한다는 점이 다르다.

이러한 Provider 인터페이스를 지원하기 위해서는 Midas(텔파이 3에서는 Data Access) 컴포넌트 팔레트의 TProvider 컴포넌트를 사용하면 간단하게 해결된다. 이 Provider 인터페이스에 필요한 데이터 세트(TTable, TQuery)를 사용하여 데이터베이스에 접속하면 된다.

File|New 메뉴를 선택하고, 객체 저장소에서 Multitier 탭을 선택하면 사용가능한 여러 종류의 원격 데이터 모듈을 생성할 수 있는 위저드 들이 나타날 것이다. 일단 간단한 원격 데이터 모듈을 만들어 보도록 하자. 참고로 원격 데이터 모듈을 만들려면 일단 어플리케이션이 만들어져 있어야 한다. 이에 비해 CORBA 데이터 모듈이나 MTS 데이터 모듈은 단독으로 만들어 질 수 있다.

원격 데이터 모듈을 만들 수 있는 Remote Data Module 아이템을 더블 클릭하면, 다음과 같은 위저드가 실행될 것이다. 이렇게 해서 작성된 원격 데이터 모듈은 OLE 자동화를 통하여 구동되므로, 일반적인 데이터 모듈과는 큰 차이가 있다.



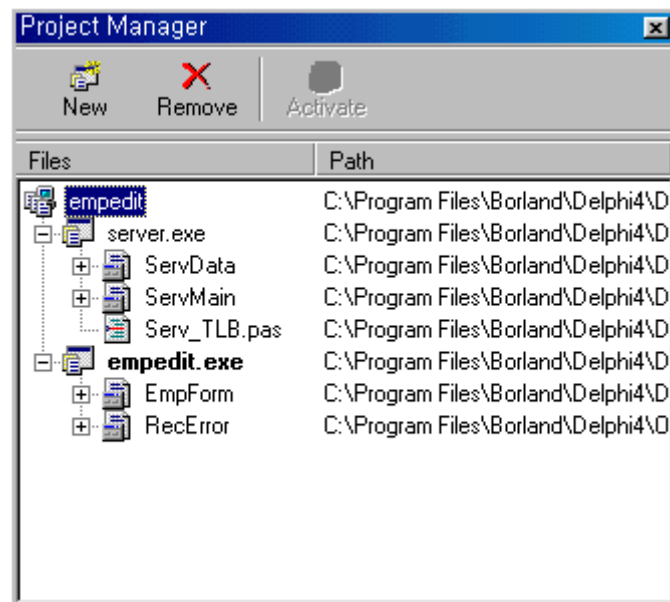
이제 해당 데이터 모듈에 원하는 TDataSet 용 컴포넌트인 TTable, TQuery, TStoredProc 등의 컴포넌트를 사용하여 단독 파일이나 DBMS 에 접속하고, 해당 TProvider 를 사용하면 된다. 또는, TProvider 컴포넌트를 사용하지 않고 해당 데이터 세트의 Provider 프로퍼티

를 사용하여도 무방하다. 단, 이 프로퍼티는 런타임에서만 사용할 수 있다.

TProvider 컴포넌트를 사용한다면 DataSet 프로퍼티에 원하는 데이터 세트를 연결하고, 원하는 이벤트에 코드를 추가한다. TProvider 컴포넌트에는 사용가능한 몇 가지의 이벤트가 있는데 이 이벤트에 대해서는 뒤에서 자세히 설명한다.

해당 어플리케이션 서버 프로젝트를 저장하고 컴파일한 다음 OLE 자동화 서버 정보를 등록해야 한다. 이 방법은 해당 레지스트리에 수동으로 저장하는 방법도 있고, 이 경우와 같이 out-of-process 서버로 작성되는 경우에는 OLE 자동화 서버를 실행하여 등록할 수도 있다. 여기에 대한 더 자세한 내용은 5 부의 내용을 참고하기 바란다. 이렇게 자동화 서버가 등록된 뒤에는 클라이언트에서 해당 서버를 호출할 때 자동적으로 서버가 구동된다.

이제 자세한 Provider 인터페이스를 작성하는 방법을 알아보자. 예제로는 Demos 디렉토리의 MIDAS 서브 디렉토리에 있는 Empedit 프로젝트를 이용한다. .bpg 파일을 로드하면 다음과 같은 화면이 나타날 것이다.



이중에 Serv\_TLB.pas 유닛이 원격 데이터 모듈을 지원하기 위해 자동으로 생성된 파일로 OLE 자동화를 지원하기 위한 각종 내용이 생성되어 있다. 만들어진 클래스 선언부분을 살펴 보자.

```
IEmpServer = interface(IDataBroker)           //데이터 브로커가 선언되어 있다.
[
  '{53BC6561-5B3E-11D0-9FFC-00A0248E4B9A}'
]
function Get_EmpQuery: IProvider; safecall;    //SQL 을 Export 시키면 자동으로 생성
property EmpQuery: IProvider read Get_EmpQuery; //SQL 을 Export 시키면 자동으로 생성
end;
```

```

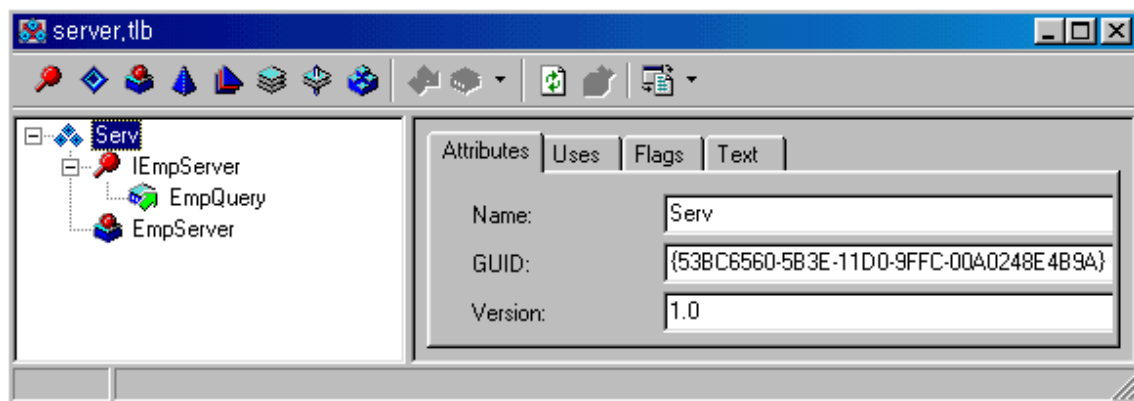
IEmpServerDisp = dispinterface
    ['{53BC6561-5B3E-11D0-9FFC-00A0248E4B9A}']
    function GetProviderNames: OleVariant; dispid 22929905;
                                                //SQL 을 Export 시키면 자동으로 생성
    property EmpQuery: IProvider readonly dispid 1;      //SQL 을 Export 시키면 자동으로 생성
end;

CoEmpServer = class
    class function Create: IEmpServer;
    class function CreateRemote(const MachineName: string): IEmpServer;
end;

```

이제 데이터 세트를 클라이언트에서 접속하여 해당 프로토콜에서 원하는 데이터 세트에 접속할 수 있도록 provider 와 연결하여야 한다. 이때에 연결하는 방법은 간단하다. 데이터 모듈에 데이터 세트를 올려놓은 다음 해당 컴포넌트를 선택하고 마우스의 오른쪽 버튼을 클릭하면, 팝업 메뉴가 나타는데 여기에서 ‘Export 데이터 세트 이름 from data module’ 이나 ‘Export 데이터 세트 이름 from data module’ 메뉴를 선택하면 해당되는 데이터 세트 컴포넌트가 연결된다.

이때 변경되는 내용은 \*.tlb 파일에 적용되는데, 다음 그림을 보면 쿼리 컴포넌트가 IEmpServer 인터페이스의 EmpQuery 프로퍼티로 등록된 것을 알 수 있다.



마찬가지로 \*.tlb 파일을 생성시킨 ServerData.pas 유닛에도 자동적으로 다음과 같은 코드가 추가된다.

```

function TEmpServer.Get_EmpQuery: IProvider;
begin

```

```

    Result := EmpQuery.Provider;
end;

```

이밖에도 다음과 같은 코드도 추가된다.

```

initialization
{ This creates the class factory for us. This code is generated automatically }
TComponentFactory.Create(ComServer, TEmpServer, Class_EmpServer,
    ciMultiInstance);
end.

```

이 코드는 초기화 코드로 해당 폼의 초기화 작업을 할 때 수행된다. 소스 코드를 살펴보면 TComponentFactory 를 이용하여 OLE 자동화 객체를 생성해 주는 코드이다.

이와 같은 작업들이 단순한 export 작업 하나로 이루어진다. 이제 원하는 TDataSet 컴포넌트를 Provider 인터페이스에 포함시키는 작업이 끝났다.

그러면, 데이터 세트의 Provider 프로퍼티를 이용하지 말고 TProvider 컴포넌트를 이용하여 보자. 이 경우에는 해당 데이터 세트의 export 를 하는 것이 아니라 Provider 컴포넌트를 선택한 다음 마우스 오른쪽 버튼을 눌러 나오는 pop-up 메뉴에서 'Export Provider1 from data module'을 선택하면 해당 데이터 모듈의 Provider 역할을 개발자가 선택한 Provider 컴포넌트가 담당하게 된다.

참고로, 이런 방법으로 Provider 컴포넌트를 export 했을 때 \*.tlb 파일에 추가되는 내용은 다음과 같다. 생성한 데이터 모듈의 이름은 Test 이다.

```

ITest = interface(IDataBroker)
    ['{8648718F-5651-11D2-B28C-004F49002780}']
    function Get_Provider1: IProvider; safecall;
    property Provider1: IProvider read Get_Provider1;
end;

ITestDisp = dispinterface
    ['{8648718F-5651-11D2-B28C-004F49002780}']
    property Provider1: IProvider readonly dispid 1;
    function GetProviderNames: OleVariant; dispid 22929905;
end;

```

해당 \*.pas 유닛에 추가되는 내용은 다음과 같다.

```
function TTest.Get_Provider1: IProvider;
begin
    Result := Provider1.Provider;
end;
```

그리고, 해당 \*.tlb 파일을 F12 를 눌러서 에디터를 살펴보면, 해당 프로퍼티로 Provider1 이 선언된 것을 확인할 수 있다. 이 \*.tlb 파일은 OLE 자동화 객체의 선언 내용을 간편하게 설정할 수 있게 해준다. 이곳에서 원하는 프로퍼티나 메소드 등을 선언하고 수정할 수 있다. 자세한 내용은 OLE 자동화에 대해 다른 장을 참고하자.

마찬가지로 컴파일 하고, 한번만 실행하면 해당되는 OLE 자동화 객체가 등록된다. 주의할 점은 한번 등록된 클래스 명으로는 다시 등록할 수 없다는 점이다. 그러므로, 테스트나 수행된 내용이 필요 없으면 UnRegister 하는 습관을 기르도록 한다.

이제 개발자는 OLE 자동화 객체를 사용하여 하나의 어플리케이션을 객체로 사용할 수 있게 되었다. 그리고, 원하는 데이터 세트 컴포넌트를 export 하여 OLE 자동화 객체의 메소드로 사용할 수 있으며 이 부분을 통하여 개발자는 원하는 데이터를 읽어 들이거나 조작할 수 있게 된다. 델파이에서는 이러한 일련의 과정을 통하여 델파이 특유의 VCL 컴포넌트들을 OLE 자동화 객체로 변환할 수 있게 해준다. 이러한 \*.tlb 파일을 이용하면 MTS 나 CORBA 데이터 모듈에도 그대로 적용할 수 있다. 한번 만들어 놓은 소스로 다양한 프로토콜을 지원하는 어플리케이션 서버를 만들 수 있는 것이다.

## ● 클라이언트 어플리케이션의 작성

이제 클라이언트 어플리케이션을 작성해 보자. 앞에서 설명한대로 먼저 간단한 테스트용 어플리케이션 서버를 하나 작성하자. ServerForm 이라는 기본적인 폼과 ServerData 라는 원격 데이터 모듈을 선언한 다음 TTable 컴포넌트를 하나 올려놓고 DBDEMOS 앨리어스의 animals.dbf 테이블을 TableName 프로퍼티의 값으로 설정한 뒤에 export 시킨다. 그리고, 이 프로젝트를 Server.dpr 이라는 이름으로 저장한 뒤에 이를 컴파일 하고, 실행하여 해당 클래스를 등록하도록 한다.

이제부터 만들 클라이언트 어플리케이션은 개발자들이 많이 접했던 1-tier 나 2-tier 방식의 어플리케이션과 크게 다른 점이 없다. 다만 TRemoteServer 와 TClientDataSet 컴포넌트가 새롭게 쓰이게 된다. TRemoteServer 컴포넌트는 클라이언트와 어플리케이션 서버를 해당 프로토콜을 통하여 연결하여 준다. 그리고 TClientDataSet 컴포넌트는 일반적인 TTable, TQuery 등의 DataSet 컴포넌트 들을 대신하여 준다. 간단하게 TDatabase 컴포넌트와 TDataSet 컴포넌트를 두 개의 컴포넌트로 대체한다고 생각하면 비유가 조금 틀리지만 비슷한 내용이 될것이다.

이제 클라이언트를 만들어 보자.

New Application 메뉴를 선택하여 새로운 프로젝트와 데이터 모듈을 추가한다.

만들어진 DataModule 에 TRemoteServer 를 올려놓고 해당 프로퍼티중에 ServerName 을 클릭하면 등록된 서버가 나타날 것이다. 이때 만들어진 서버가 DCOM 을 사용한다면 간단하게 ComputerName 프로퍼티를 먼저 설정하면, 해당 컴퓨터에 설치된 서버들이 나타날 것이다. 이때 원하는 서버를 선택하면 된다. Server.ServerData 는 조금 전에 만들어진 Server.exe 의 원격 데이터 모듈인 ServerData 의 이름을 지칭한다. 선택되어지면 ServerGUID 는 자동으로 설정된다.

참고로 예제로 제공되는 클라이언트 어플리케이션의 경우에 컴퓨터 이름이 독자 들의 컴퓨터 이름과 다를 것이므로 소스를 분석할 때 이를 고려해서 서버가 설치된 컴퓨터 이름을 잘 지정하도록 한다.

그 다음에는 TClientDataSet 컴포넌트를 올려 놓고 RemoteServer 프로퍼티를 RemoteServer1 으로 지정한다. 그리고, ProviderName 을 설정하면 자동적으로 연결된 Server.exe 가 수행되면서 원격 데이터 모듈에 선언된 Table1 이 나타난다. 이 Table1 을 프로퍼티로 설정하면 TClientDataSet 컴포넌트와 Table1 이 연결된 것이다.

이제 이렇게 만들어진 TClientDataSet 을 사용하여 일반 어플리케이션을 제작하는 방법과 동일한 과정으로 제작에 임하면 된다.

여기에서 TRemoterServer 컴포넌트는 서버로 제작된 OLE 자동화 객체의 Provider 에 연결할 수 있도록 해주는 중간 역할을 한다. 엄밀히 말해 해당 Provider 의 DataSet 컴포넌트와 TClientDataSet 컴포넌트를 연결한다.

참고로, TRemoteServer 컴포넌트는 델파이 3 와의 호환성을 위해 남겨진 컴포넌트로 델파이 4 에서는 사용하고자 하는 프로토콜을 직접 지정하여 컴포넌트를 사용해야 한다. 지금과 같이 DCOM 을 사용한다면 TDCOMConnection 컴포넌트를 사용하여 똑 같은 방법으로 데이터 모듈을 연결하여 사용하면 된다. 앞서서도 여러 차례 언급한 TCORBAConnection, TSocketConnection 등도 비슷한 역할을 한다.

MIDASConnection 을 사용하면 앞에서 설명한 3 가지 프로토콜을 혼용할 수 있다. 다시 말해 DCOM, OLEEnterprise 와 소켓 접속을 ConnectType 의 변환에 따라서 간단하게 조절할 수 있다. 그렇다면, OLEEnterprise 와 SimpleObjectBroker 는 무엇을 하는 것일까? SimpleObjectBroker 에는 LoadBalanced 라는 프로퍼티가 있다. SimpleObjectBroker 에서는 서버에 등록된 서버의 어플리케이션 서버를 사용하게 해준다. 여러 어플리케이션 서버에 대한 로드 밸런싱을 수행하는 것이 그 역할이며, 여러 개의 어플리케이션 서버를 두고 클라이언트의 요구를 분산시켜 구동시는 것이다. 이렇게 하기 위해서는 접속 방법으로 TOLEEnterpriseConnection 을 사용하는 것이 좋다. 또는 접속 방법을 마음대로 바꿀 수 있는 TMIDASConnection 컴포넌트를 이용할 수도 있겠다.

DCOM 이나 CORBA 등을 지원할 수 없는 환경에서는 TCP/IP 를 사용하여 소켓 접속을 통해서 접속이 가능하다. 이럴 때에는 TSocketConnection 컴포넌트를 이용한다.

- 멀티-tier 어플리케이션 제작에서 주의할 점

보통 어플리케이션 서버는 로컬 시스템이 아닌 원격 시스템일 경우가 많다. 이 경우에는 해당 시스템의 이름을 지정해 주는데 디자인 타임에서 설정해도 좋고 런타임에 설정하여도 무방하다. 하지만 DCOM 을 이용할 경우에는 시스템 레지스트리에 등록되어 있는 경우에 시스템의 이름을 지정하지 않아도 동작한다. 그 이유는 델파이에서 기본적으로 지원하는 어플리케이션 서버는 OLE 자동화 객체이기 때문이다.

ServerName 프로퍼티에는 어플리케이션 서버의 '이름.exe'에서 이름으로 지정된 것을 사용한다. 이 이름은 OLE 자동화 객체로 사용되는 이름과 동일하다.

보통 해당 접속 컴포넌트의 ServerName 과 ComputerName 을 설정하고 TClientDataSet 컴포넌트의 RemoterServer 와 ProviderName 프로퍼티를 설정하면 Active 프로퍼티가 자동으로 True 가 되지만, 기본적으로 ServerName 과 ComputerName 만 설정하면 데이터를 이용할 수 있다. 이때에 해당 어플리케이션 서버는 이미 동작을 시작하게 되는데, 이는 IProvider 인터페이스를 통하여 GetProvider 메소드가 수행되고 원하는 ProviderName 이 전달되기 때문이다.

Provider 는 OLE 자동화 객체와 데이터 세트 컴포넌트를 연결하기 위한 고리로서 OLE 자동화 객체의 메소드로 쓰일 수 있게 해주는 방법이다. 더구나 Provider 는 기본적으로 1 대 다의 클라이언트가 연결될 수 있는 구조를 가지고 있다는 점이 강점이다. 클라이언트에서 1 개의 어플리케이션 서버가 아닌 몇 개의 어플리케이션 서버의 연결이 필요할 경우에는 해당 되는 수 만큼의 접속 컴포넌트를 올려놓고 이를 사용하면 된다.

이제 필요한 접속을 설정하고 해당 어플리케이션 서버의 Provider 를 연결하여 사용하는 방법을 알아 보았다. 이러한 통신은 어플리케이션 서버와의 접속이 끊기기 전까지 이루는데, 접속 전에 할 일이 있으면 접속 컴포넌트의 BeforeConnect 나 AfterConnect 등의 이벤트를 활용하면 된다.

이제 원하는 작업을 클라이언트에서 수행할 수 있는 프로그램을 만들면 된다. 참고로, 중간에 접속을 중단하는 방법은 Connected 프로퍼티를 False 로 변경하기만 하면 된다. 그 밖에도 ServerName 프로퍼티를 변경하거나 클라이언트 프로그램을 종료하면 자동적으로 어플리케이션 서버는 동작을 멈추고 종료한다.

## TClientDataSet 컴포넌트가 다른 데이터 세트 컴포넌트와 다른 점

멀티-tier 모델을 이용하여 클라이언트 어플리케이션을 작성할 경우에 TClientDataSet 컴포넌트를 사용한다는 점을 이미 언급한 바 있다. 그렇다면, TClientDataSet 컴포넌트는 일반적인 데이터 세트와 어떤 점이 같고 어떤 점이 다른가? 실제 클라이언트 데이터 세트는 델파이 3 에도 존재하였던 일종의 MemoryDataSet 과 거의 유사하다. 이것은 원격 컴퓨터



에 떨어져 있는 어플리케이션 서버의 데이터 세트의 내용을 클라이언트에 있는 것처럼 일반적인 데이터 인식 컴포넌트를 속임으로써 동작한다.

VCL 의 계층도를 봐도 TClientDataSet 컴포넌트가 TDataSet 에서 파생된 클래스임을 알 수 있다. 그러므로, TClientDataSet 컴포넌트는 일반 테이블에서 지원하는 대다수의 기능을 모두 처리하며, 이를 모두 메모리에서 수행하므로 더욱 빠르게 동작한다.

TClientDataSet 컴포넌트의 다른 용도는 플랫-파일 단위의 1-tier 어플리케이션을 작성할 수 있도록 해준다. 이렇게 하려면, TClientDataSet 컴포넌트를 마우스로 선택한 다음 오른쪽 버튼을 클릭한 뒤에 나타나는 팝업 메뉴에서 Assign Local Data 메뉴를 선택하면 해당 테이블이나 쿼리의 테이블 하나를 몽땅 메모리에 적재하고, 이 내용을 플랫 파일로 저장할 수 있다. 그리고, 해당 플랫 파일을 로드하여 BDE 가 없이 DBCLIENT.DLL 파일만으로도 데이터 어플리케이션을 작성할 수 있게 되었다. 물론, 메모리에서 이루어지는 작업이기 때문에 메모리가 적은 시스템에서는 곤란한 경우를 만날 수도 있을 것이다.

이러한 모델을 서류가방 모델(Briefcase model)이라 부르는데, 이 기능을 사용하여 필요할 때에 서버로부터 자료를 받아와서 클라이언트에 저장한 다음 클라이언트에서 작업을 수행하고 나중에 필요한 시기에 데이터를 전송하는 방식을 취할 수 있다. POS 와 같은 시스템에 쓰기에 적당한 방법으로, 특히 네트워크의 성능이 불량하거나 속도가 느린 경우에 유용하게 사용할 수 있는 방법이다.

일반적으로 TClientDataSet 컴포넌트의 사용 방법은 일반 데이터 세트의 CachedUpdates 메소드를 사용하는 것과 유사하다. 특히, 어플리케이션 서버로부터 데이터를 메모리로 가져오고 변경된 자료가 있을 경우에 해당 자료를 어플리케이션 서버로 보내어 변경된 내용을 데이터베이스에 적용하는 과정은 해당 데이터베이스와 동기화를 취한다는 점에서 비슷하다. 다만, OLE 자동화나 CORBA, 소켓 등을 사용한다는 점이 다를 뿐이다.

## 클라이언트 데이터 세트의 레코드 조작

클라이언트에서 어플리케이션 서버에 접속하면 필요한 자료를 전송 받아 작업하게 된다. 이때 사용자가 보는 데이터는 실제 어플리케이션 서버에 존재하는 데이터가 아닌 복제된 자료들이다. 그러므로, 사용자가 데이터 조작을 취하면 클라이언트 데이터 세트의 Delta 프로퍼티에 변경된 내용이 기록되며, 이후에 ApplyUpdates 메소드를 호출할 경우 Delta 프로퍼티에 저장된 데이터가 어플리케이션 서버로 전송되어 데이터 조작을 취하게 된다.

클라이언트에서 TClientDataSet 컴포넌트의 메소드인 ApplyUpdates 를 호출하면 연결된 Provider 에 Delta 에 저장된 데이터 조작 내용을 전송한다. 어플리케이션 서버의 Provider 는 데이터를 전송 받아 자신의 ApplyUpdates 메소드를 수행하고 데이터베이스에 기록한다. 이때에 데이터의 처리는 해당 레코드 단위로 이루어지는데 그 이유는 데이터베이스에 저장하고 조작하는 동안에 오류가 발생하면 해당 레코드 단위의 내용을 되돌려주기 위해서다.

Provider 가 변경된 내용을 데이터베이스에 기록한 다음에는 오류가 발생한 레코드를 클라이언트에 되돌려준다. 이때에 클라이언트는 반송된 자료를 재처리하기 위한 작업을 수행하는데, 변경된 내용을 레코드 단위로 처리하면서 문제가 발생된 내용을 Result 프로퍼티를 통하여 어플리케이션 서버로 전달한다.

이제 클라이언트 데이터 세트와 캐쉬 업데이트가 유사하다고 이야기한 부분이 이해가 되리라 생각한다. ApplyUpdates 메소드가 필요한 시점에서 데이터를 전송함으로써 데이터의 조작이 이루어지므로 사용자에게 원하는 시점에 변경된 내용을 수정할 수 있게 해주거나 자동으로 ApplyUpdates 가 일어나게 해주어야 한다.

그리고, ApplyUpdates 는 MaxErrors 라는 파라미터를 통해서 데이터 조작 시에 일어난 해당 에러의 갯수가 이보다 작을 경우 동작을 계속 수행하게 해주며, MaxErrors 이상의 에러가 발생하면 그 자리에서 수행을 멈춘다. 이 갯수는 에러가 발생하여 변경하지 못한 레코드의 갯수를 의미한다.

해당되는 레코드의 값은 OnReconcileError 이벤트를 통하여 알 수 있다. 이 이벤트는 에러가 발생할 때마다 한번씩 수행되므로 에러에 대한 조작이나 반응을 이 이벤트에 넣어놓으면 에러를 처리할 수 있다.

## 에러가 발생한 레코드의 처리

문제없이 자료가 전송되었다면 좋지만, 멀티-tier 환경에서 가장 믿을 수 없는 것이 네트워크 환경이다. 시시때때로 변화하며 부족한 대역폭을 갖고, 혹시나 패킷을 잃어버리기도 한다면 에러의 발생은 당근(!)이다.

앞에서 설명한대로 OnReconcileError 이벤트를 통하여 일차적으로 재조정 작업을 시도하며, 자동적으로 수행하는 이벤트이다. 문제는 이 이벤트를 작성할 때 주의할 점인데 이 이벤트 내에서는 레코드의 위치를 변경할 수 있는 다른 어떠한 이벤트도 수행하면 안된다는 점이다. 왜냐하면, 다음을 보자.

```
procedure TDataModule1.ClientDataSet1ReconcileError(  
    DataSet: TClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind;  
    var Action: TReconcileAction);
```

이 코드를 보면, 해당 데이터 세트와 에러 내용, 수정/추가/삭제의 변경방법, 그리고 에러에 반응하는 raSkip, raAbort, raMerge, raCorrect, raCancel, raRefresh 와 같이 지정되는 Action 부분이 리턴되어 온다.

그러므로, 잘못하면 무한루프에 빠질 수 있다. 실제 이 이벤트의 역할은 에러가 발생한 레코드의 처리를 어떻게 할 것이냐 ? 라는 점이다. 실제 Action 에 들어갈 수 있는 값은 다음과 같다.

코 드	내 용
raSkip	내용을 건너뛰고 Delta 자료는 남겨두어 다음 ApplyUpdates 에 다시 한번 처리할 수 있도록 한다.
raAbort	재조정을 하지 않는다. 모든 처리를 중단하고 변경작업을 중단한다.
raMerge	변경내용을 서버의 레코드 내용과 합병한다.
raCorrect	변경을 다시 한번 시도한다.
raCancel	변경 사항의 Delta 자료를 삭제한다.

이때 참고할 내용은 UpdateKind 파라미터에 담겨있는데 시도된 데이터가 어떠한 상태인가 하는 점이다. ukModify, ukInsert, ukDelete 의 3 가지 상태는 수정, 추가, 삭제의 3 가지 형태를 말한다. 물론, E 파라미터에는 해당 에러메시지를 전송받는다.

개발자는 이러한 점을 참고로 하여 Action 파라미터에 원하는 값을 넣으면 된다.

그리고, 이 이벤트에서 해당 레코드의 OldValue, NewValue, CurValue 의 프로퍼티를 참고하면 해당 에러의 수정에 도움이 될 것이다.

에러메시지를 보는 방법을 잠깐 살펴보자. 이 에러 메시지는 일반적인 에러 메시지 처리법과 동일하다. E.message 는 메시지의 내용, E.ReconcileError(E).ErrorCode 하면 어떤 에러가 발생하였는지 알 수 있다.

```
with ReconcileError(E) do
begin
    if ErrorCode = DBIERR_KEYVIOL then
        Action := raSkip           //Key violation 이 발생하였다면 레코드는 그냥 건너 뛰어라
    else
        Action := raAbort         //나머지는 수행중지
end;
```

## 플랫-파일 다루기

TClientDataSet 컴포넌트를 이용하여 플랫 파일을 다루는 방법은 앞서도 잠시 설명하였지만, 그 사용 용도가 무궁무진하다. 이 방법은 대입된 로컬 데이터를 Delta 프로퍼티에 몽땅(!) 저장하여 사용하는 방법으로 SaveToFile 과 LoadFromFile 메소드를 통하여 자료를 저장하고 읽어들이 수 있다.

이렇게 하기 위해서는 다음과 같은 프로퍼티와 메소드에 대해서 잘 알아야 한다.

프로퍼티/메소드	내 용
----------	-----

FetchOnDemand	필요할 때에 자동적으로 데이터를 가져올 수 있다. 이 값이 True 면 자동적으로 데이터를 읽어오고, False 인 경우에는 GetNextPacket 메소드를 사용하여 수동으로 읽어 와야 한다.
PacketRecords	한번에 몇 개의 레코드를 가져올 것인가를 결정한다. 보통 -1 를 값으로 설정하여 전체 데이터를 읽어온다.
GetNextPacket	설정된 PacketRecords 수 만큼의 다음 레코드를 읽으라는 메소드

보통은 PacketRecords 프로퍼티가 적당한 값을 가진다. DBGrid 나 StringGrid 를 사용할 경우에 적당한 데이터를 읽어 올 수 있도록 한번에 읽어올 자료의 크기를 지정하는 것이 좋다. PacketRecords 프로퍼티의 값이 0 보다 크다면 GetNextPacket 메소드를 통하여 해당 자료의 크기만큼 자료가 전송되어 오며, 0 일 경우에는 해당 데이터의 정보를 다시 읽어온다. 그 밖의 일반적인 동작은 일반 데이터 세트와 동일하다.

그 밖에 TClientDataSet 컴포넌트의 또다른 중요한 기능은 데이터를 처리하면서 Delta 데이터에 대한 Log 를 관리할 수 있게 해주는 것이다. 이 Log 는 LogChange 프로퍼티를 True 로 설정하는 것으로 이용할 수 있는데, 이 경우 수정/삽입/변경의 모든 내용의 히스토리를 저장한다. 그리고, 이렇게 만들어진 히스토리를 사용하여 변경 내용을 취소할 수 있다. 그 방법은 다음의 3 가지 방법이 있다.

#### 1. 한단계 취소:

UndoLastChange 메소드를 사용하여 바로 전단계의 자료로 돌아가는 방법이다. 이때에 FollowChange 라는 프로퍼티가 True 이면 하면 undo 된 자료의 위치로 이동하고, False 이면 이동하지 않는다. 이 작업이 성공하면 True, 실패하면 False 가 반환된다.

#### 2. 한꺼번에 취소:

히스토리는 하나의 레코드에 대한 여러 번에 걸친 변경 자료를 모두 가지고 있다. 이때에 RevertRecord 메소드를 사용하면 선택된 레코드의 모든 변경을 취소하고 원래 내용으로 돌려놓는다.

#### 3. 전체 레코드의 변경 내용 취소:

변경된 내용을 전부 되돌려 놓고자 할 경우에는 CancelUpdates 메소드를 사용하면, 모든 변경사항이 원상복귀 된다.

#### ● 플랫폼-파일 상태에서의 데이터 저장

플랫-파일 상태에서는 ApplyUpdates 메소드를 사용하여 데이터를 저장하지 않는다. 이 경우에는 변경된 자료가 히스토리인 로그 자료에 쌓여 있으므로, Data 프로퍼티는 원래의 자료를 고스란히 가지고 있다. 이때에 MergeChangeLog 메소드를 수행하면 로그 자료를 Delta 프로퍼티에 적용하여 저장할 수 있도록 해준다.

- 실제 클라이언트 데이터 세트의 저장

실제 클라이언트 데이터 세트의 데이터를 저장하려면 SaveToFile 메소드를 통하여 하나의 파일에 저장하고, 실제 해당 파일이 존재한다면 그 파일을 덮어서 다시 쓰게 된다.

## 정 리 (Summary)

이번 장에서는 멀티-tier 데이터베이스 어플리케이션을 제작하는 방법에 대해서 알아보았다. 멀티-tier 데이터베이스 어플리케이션은 대부분의 컴퓨터가 네트워크에 물리게 되는 환경이 되면 그 중요성이 점점 커질 것이다. 그러므로, 이번 장에서 다룬 내용에 대해서 잘 익혀둘 필요가 있다.

다음 장부터 이어지는 제 4 부에서는 텔과이의 가장 커다란 장점이라고 할 수 있는 컴포넌트 개발에 대한 내용을 다루게 된다.

## 패키지의 활용 (Using Packages)

텔파이에서는 텔파이의 버전 3 에서부터 패키지(package)라는 새로운 개념을 도입함으로써 보다 효율적이고, 분산된 구조를 가질 수 있게 되었다. 패키지란 간단히 말하면 컴포넌트를 모아서 이들을 적절하게 분산할 수도 있고, 어플리케이션을 작게 만들 수도 있는 DLL wrapper 이다.

엄밀히 말하면 전혀 새로운 개념이라고는 할 수 없는 것이 패키지이다. 과거에도 텔파이에서 DLL 을 이용해서 어플리케이션을 여러 개로 분산시킬 수 있었다. 그렇지만 패키지는 이 보다 훨씬 더 편리하고, 진보된 개념이다.

이번 장에서는 패키지 개념에 대해서 알아보고, 이를 실제 어플리케이션에서 어떻게 이용할 수 있는지 알아보도록 한다.

### 패키지(Package)란 ?

패키지란 일종의 DLL 로, DLL 에 보다 많은 장점을 부여한 것으로 생각하면 된다. 패키지 파일의 확장자는 BPL 인데, BPL 파일을 사용할 때의 장점에는 다음과 같은 것들이 있다. 참고로 텔파이 3 버전에서의 BPL 에 해당되는 확장자는 DPL 이다.

- 실행 파일의 크기를 줄여 준다.
- 어플리케이션을 배포하고 업데이트하기 쉽다.
- 여러 개의 어플리케이션에서 리소스를 공통적으로 사용할 수 있다.

DLL 에 비해 확실한 장점으로 꼽을 수 있는 것은 어플리케이션의 조각을 쉽게 합쳐서 패키지 모듈로 재구성할 수 있다는 것이다. 또한, 기본적으로 바뀌지 않는 부분은 내버려 두고 바뀐 부분의 모듈만을 배포할 수도 있다. DLL 과는 달리 패키지를 사용할 때에는 함수를 미리 선언하지 않아도 되고, 명시적으로 로드할 필요도 없다. 단지 필요한 것은 uses 절에 모듈의 이름을 적어주는 것으로 충분하다.

패키지와 관련된 파일 확장자에는 다음과 같은 것들이 있다.

확장자	설 명
DPK	패키지에 담겨 있는 유닛들에 대한 소스 파일
DCP	패키지 헤더와 컴파일러가 요구하는 심볼 정보를 포함한 패키지의 모든 DCU 파일 들이 하나로 합쳐진 바이너리 이미지 파일로 패키지당 하나씩 생성된다.
DCU	패키지에 포함된 유닛 파일의 바이너리 이미지 파일
BPL	런타임 패키지로 일종의 윈도우 DLL 파일이다.

## 패키지의 필요성

기본적으로 델파이의 모든 기본적인 컴포넌트는 패키지로 구성되어 있다. 델파이의 표준 컴포넌트 라이브러리에 패키지를 적용함으로써 얻을 수 있는 효과도 많지만, 실제로 더욱 도움이 된 것은 써드 파티에서 개발한 컴포넌트 들을 패키지를 이용해서 대단히 쉽게 설치, 이용할 수 있게 된 것이다.

개발자가 어플리케이션을 사용할 때 상당히 많은 수의 써드 파티 라이브러리를 사용한다고 가정하자. 이렇게 되면 실행 파일의 크기가 1MB 는 훌쩍 뛰어 넘기가 일수이다. 이렇게 커다란 실행 파일의 대부분을 차지하는 것은 코드에 포함된 정적 라이브러리의 코드 들이다. 그러나 이들 중 빈번히 사용되고, 변경하는 부분은 그다지 많지 않은 경우가 대부분이다. 만약에 이들을 분리된 패키지 파일로 분리해낼 수 있으면 실행 파일의 크기를 많이 줄일 수 있음은 쉽게 미루어 짐작할 수 있을 것이다.

단적인 예를 들어, 폼에 TTable, TDataSource, TDBGrid, TDBNavigator 컴포넌트를 하나씩 넣고, 어플리케이션을 컴파일하면 실행파일의 크기는 400KB 를 넘게 된다. 그렇지만, 이를 패키지를 이용해서 분리해 주면 약 13KB 정도의 아주 작은 실행 파일로 만들어낼 수 있게 된다.

패키지를 활용할 수 있는 또 다른 좋은 상황은 동일한 모듈을 포함한 여러 개의 실행 파일을 만들 때이다. 예를 들어, 동일한 컴포넌트 들을 사용해서 개발한 여러 개의 어플리케이션이 있다면 이들은 동일한 패키지를 공유할 수 있다. 이럴 때에 여러 개의 어플리케이션을 각각 업그레이드 하기 보다, 동일한 부분의 기능을 향상 시켜서 배포할 수도 있게 된다. DLL 과 패키지의 또 다른 점은 패키지 파일은 프로젝트에 정적으로 링크된다는 점이다. 런타임에는 사용된 모든 DLL 파일이 일단 메모리에 적재된다. 그렇기 때문에, 패키지는 DLL 과 같은 메모리 절약 효과는 없다.

패키지를 사용하는 것이 꼭 장점만 있는 것은 아니다. 기본적으로 조각을 내어 업그레이드를 하는 것이므로 버전 컨트롤(version control)에 보다 많은 신경을 써야 한다.

## 패키지의 종류

패키지에는 다음과 같이 세가지가 있다.

- 런타임 패키지 (Runtime Packages)

가장 흔히 사용되는 형태로 어플리케이션을 배포할 때 같이 배포되는 것이다. 여기에는 컴포넌트나 함수 라이브러리가 포함된다.

- 디자인 타임 패키지 (Design-time Packages)

델파이의 IDE 에 컴포넌트를 인스톨하고, 이들 컴포넌트에 대한 프로퍼티 에디터(property editor)를 포함할 때 사용하는 패키지로 IDE 자체가 워낙 커다란 어플리케이션이기 때문에 이들을 유용하게 이용할 수 있다. 그러므로, 실제로 잘 쓰이지 않는 컴포넌트를 패키지로 분리해 두었다가, 필요할 때 사용하는 것도 좋은 아이디어가 될 수 있다. 기본적으로 패키지는 런타임 패키지 이면서 동시에 디자인 타임 패키지일 수도 있다.

#### ● 커스텀 패키지 (Custom Packages)

커스텀 패키지는 개발자가 직접 패키지로 개발한 것을 말한다. 여기에는 각종 컴포넌트와 클래스, 함수 라이브러리가 포함될 수 있으며, 어플리케이션에 쉽게 포함될 수 있다.

### 패키지의 설치

컴포넌트를 설치하기 위해 델파이의 메뉴를 살펴 보면, 과거 델파이 2.0 까지 존재하던 Install Component 메뉴이외에 Install Packages 메뉴가 추가되어 있는 것을 확인할 수 있을 것이다. 모든 볼랜드사의 컴포넌트와 델파이 4 에 번들로 들어온 써드 파티 컴포넌트가 모두 패키지로 구성되어 있다. 과거에는 컴포넌트를 설치하기 위해 유닛 파일(.PAS 또는 .DCU)을 이용했지만, 패키지를 설치할 때에는 BPL, DCP 파일을 사용하게 된다. 하나의 컴포넌트를 설치해도 반드시 패키지 파일을 지정해 주어야 한다.

Component 메뉴에서 Install Packages 를 선택하면 IDE 에 설치될 디자인 타임 패키지를 선택할 수 있는 대화 상자가 나타난다. 여기에서 델파이 4 의 현재 컴포넌트 팔레트에 추가할 패키지를 선택할 수 있다. ‘Add’ 버튼을 클릭하면 추가할 패키지 파일을 선택할 수 있다.

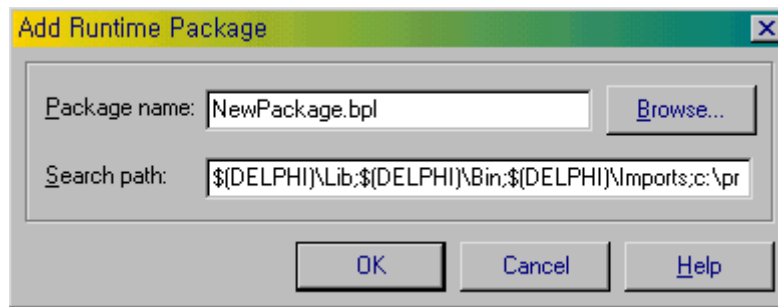
패키지 파일과 함께 .DPC 라는 확장자를 가지는 파일도 발견할 수 있는데, 이 파일의 확장자는 ‘Delphi Package Collection’의 약자로 어플리케이션에 의해 하나 이상의 패키지가 사용될 때 이들을 모아 놓은 파일을 의미한다.

### 어플리케이션에 패키지 사용하기

어플리케이션에 패키지를 사용하려면, 패키지를 사용할 어플리케이션 프로젝트의 Project|Options 메뉴를 선택하고, Packages 탭에서 Build with Runtime Packages 체크박스를 선택하고, 하나 이상의 패키지의 이름을 적어 넣으면 된다. 이때 디자인-타임 패키지와 연결된 런타임 패키지는 이미 에디트 박스에 나열되어 있을 것이다.

이미 존재하는 리스트에 패키지를 추가하려면, Add 버튼을 클릭하고 새로운 패키지의 이름을 다음과 같은 Add Runtime Package 대화 상자에서 입력하면 된다. 이때 Browse 를 선택하여 패키지 파일의 위치를 지정할 수 있다.





Runtime Packages 에디트 박스에 나열된 런타임 패키지 들은 자동적으로 컴파일 시에 어플리케이션과 링크되는 패키지 들이다. 이 내용이 비어있으면, 어플리케이션이 패키지 없이 컴파일 된다.

런타임 패키지는 현재 프로젝트에 대해서만 선택된다. 이때, 선택한 패키지를 앞으로의 프로젝트에 자동으로 디폴트로 사용하게 하려면 Defaults 체크 박스를 선택하면 된다.

#### ● 어떤 런타임 패키지를 사용할 것인가 ?

델파이에는 여러 가지 기본적인 언어와 컴포넌트를 지원하기 위한 런타임 패키지가 지원된다. 가장 흔히 사용되는 것은 VCL40 으로 가장 기본적인 컴포넌트와 시스템 함수, 윈도우 인터페이스를 제공한다. 다음에 델파이에서 제공되는 런타임 패키지에서 지원되는 유닛에 대해서 나열해 보았다.

패키지	포함된 유닛
VCL40.BPL	Ax, Buttons, Classes, Clipbrd, Comctrls, Commctrl, Commdlg, Comobj, Comstrs, Consts, Controls, Ddeml, Dialogs, Dlgs, Dsgnintf, Dsgnwnds, Editintf, Exptintf, Extctrls, Extdlgs, Fileintf, Forms, Graphics, Grids, Imm, IniFiles, Isapi, Isapi2, Istreams, Libhelp, Libintf, Lzexpand, Mapi, Mask, Math, Menu, Messages, Mmsystem, Nsapi, Ole2l, Oleconst, Olecnrs, Olectrls, Oledlg, Penwin, Printers, Proxies, Registry, Regstr, Richedit, Shellapi, Shlobj, Stdctrls, Stdvcl, Sysutils, Tlhelp32, Toolintf, Toolwin, Typinfo, Vclcom, Virtintf, Windows, Wininet, Winsock, Winspool, Winsvc
VCLX40.BPL	Checklst, Colorgrd, Ddeman, Filectrl, Mplayer, Outline, Tabnotbk, Tabs
VCLDB40.BPL	Bde, Bdeconst, Bdeprov, Db, Dbcgrids, Dbclient, Dbcommon, Dbconsts, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables, Dsintf, Provider, SMintf
VCLDBX40.BPL	Dblookup, Report
DSS40.BPL	Mxarrays, Mxbutton, Mxcommon, Mxconsts, Mxdb, Mxdcube, Mxdssqry, Mxgraph, Mxgrid, Mxpivsrc, Mxqedcom, Mxqparse, Mxqryedt, Mxstore, Mxtables, Mxqvb
QRPT40.BPL	Qr2const, Qrabout, Qralias, Qrctrls, Qrdatasu, Qrexpbld, Qrextra, Qrprev,

	Qrprgres, Qrprntr, Qrqred32, Quickrpt
TEE40.BPL	Arrowcha, Bubblech, Chart, Ganttch, Series, Teeconst, Teefunci, Teengine, Teeprocs, Teeshape
TEEDB40.BPL	Dbchart, Qrtee
TEEUI40.BPL	Areaedit, Arrowedi, Axisincr, Axmaxmin, Baredit, Brushdlg, Bubbledi, Custedit, Dbedit, Editchar, Flineedi, Ganttedi, Ieditcha, Pendlg, Pieedit, Shapeedi, Teeabout, Teegally, Teelish, Teeprevi, Teexport
VCLSMP40.BPL	Sampreg, Smpconst

### ● 패키지의 동적 적재

패키지를 동적으로 어플리케이션에 적재하려면 LoadPackage 함수를 사용하면 된다. 예를 들어, 다음의 코드는 파일 열기 대화 상자에서 선택한 파일을 어플리케이션에 적재한다.

```
with OpenFileDialog do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

반대로, 동적으로 패키지를 메모리에서 내리고 싶을 때에는 UnloadPackage 프로시저를 사용하면 된다. 그렇지만, 패키지를 메모리에서 해제하려면 패키지와 연관된 객체와 클래스를 먼저 모두 파괴해야 하는 것을 명심하기 바란다.

### ● 디자인-타임 패키지

디자인-타임 패키지는 IDE 의 컴포넌트 팔레트에 컴포넌트를 추가하고, 컴포넌트에 대한 프로퍼티 에디터를 생성할 때 사용된다.

텔파이 4 에는 다음과 같은 디자인-타임 컴포넌트 패키지가 IDE 에 설치되어 있다.

패키지	컴포넌트 팔레트	패키지	컴포넌트 팔레트
DCLSTD40.BPL	Standard, Additional, System, Win32, Dialogs	DCLTEE40.BPL	Additional (TChart component)
DCLDB40.BPL	Data Access, Data Controls	DCLMID40.BPL	Data Access (MIDAS)
DCL31W40.BPL	Win 3.1	NMFAST.BPL	Internet
DCLSMP40.BPL	Samples	DCLOCX40.BPL	ActiveX
DCLQRT40.BPL	QReport	DCLDSS40.BPL	Decision Cube

IBSMP40.BPL	Samples (IBEventAlerter component)	DCLINT40.BPL	International Tools (Resource DLL wizard)
-------------	---------------------------------------	--------------	--

이들 디자인-타임 패키지는 Requires 절에 참조되어 있는 런타임 패키지를 호출하여 동작한다. 예를 들어, DCLSTD40 은 VCL40 런타임 패키지를 호출한다.

## 패키지 소스 파일

분리된 소스 파일에서 선언된 패키지는 DPK 파일로 저장된다. 패키지 소스 파일은 다음과 같은 형태를 가지고 있다.

```
package packageName;
    requiresClause;
    containsClause;
end.
```

여기서 requiresClause, containsClause 는 옵션이다. 예를 들어, 다음 코드는 VCLDB40 패키지를 선언하는 것이다.

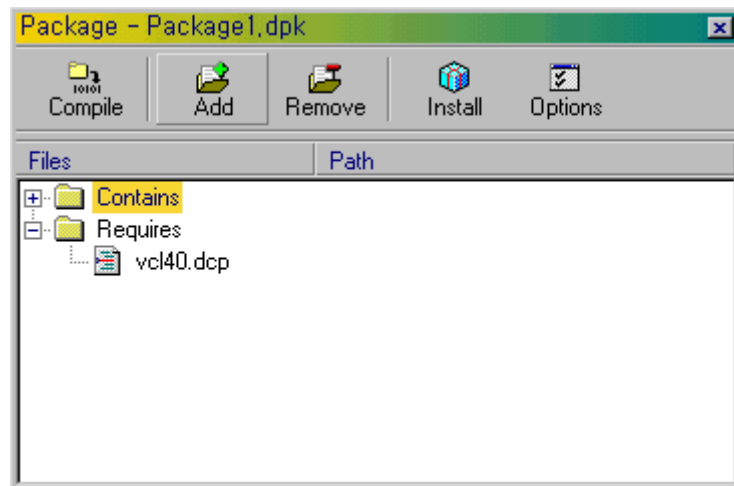
```
package VCLDB40;
    requires VCL40;
    contains Db, Dbcgrids, Dbctrls, Dbgrids, ... ;
end.
```

이와 같이 requires 절에는 패키지에서 사용하는 다른 외부 패키지를 선언한다. 만약 다른 패키지를 참조하지 않으면, require 절이 필요하지 않다. contains 절에는 패키지에 컴파일 될 유닛이 지정된다. 경우에 따라서는 다음과 같이 디렉토리 패스를 지정할 수도 있다.

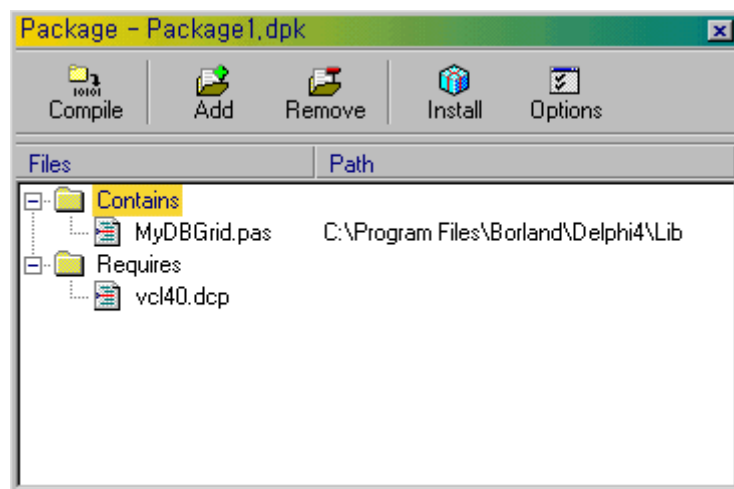
```
contains MyUnit in 'C:\WMyProject\WMyUnit.pas';
```

## 패키지 만들기

패키지를 만들려면, File|New 메뉴를 선택하고 package 를 선택한다. 그러면, 다음 그림과 같이 패키지 에디터가 뜬다.



File|Save As 명령을 선택하여, 적당한 패키지 이름을 정해서 저장한다. 이때 파일은 .DPK 확장자를 가지게 된다. 패키지에 유닛을 추가하려면, Add 버튼을 누르고 추가할 유닛 파일이나 액티브 X 컴포넌트 등을 지정하면 된다. 다음은 필자가 작성한 컴포넌트 유닛을 하나 추가한 화면이다.



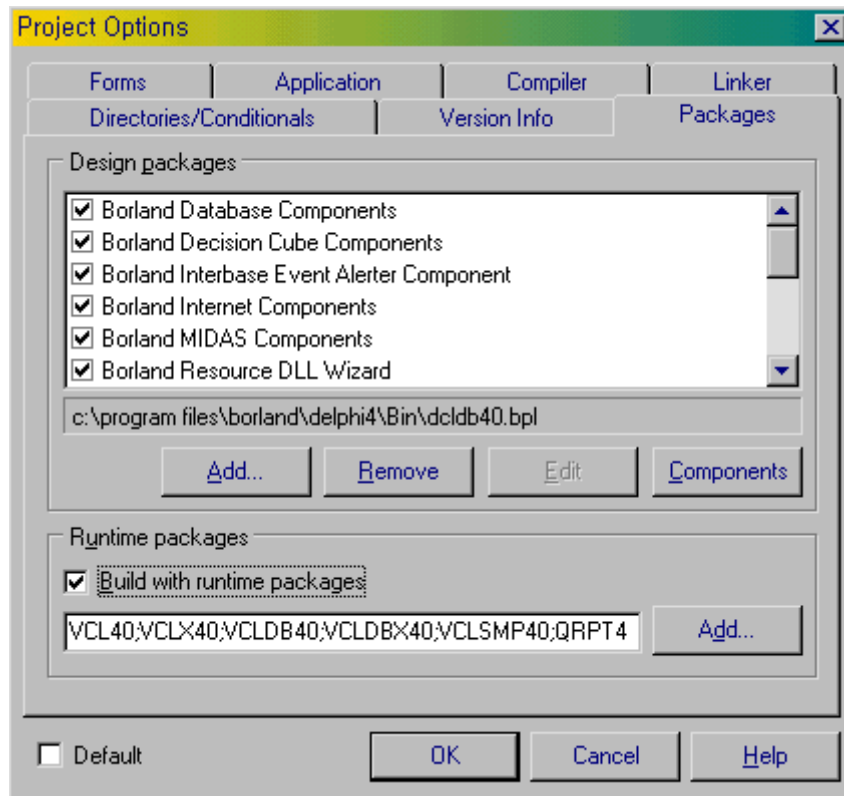
일단 이렇게 패키지에 파일과 컴포넌트를 모두 추가 했으면 컴파일을 한다. 컴파일이 정상적으로 완료되면 BPL 파일이 생성되며 Install 버튼을 선택하면 델파이의 IDE 에서 사용할 수 있게 된다. 패키지의 Options 메뉴에서 몇 가지 옵션을 줄 수 있는데, 예를 들어 컴파일할 때마다 패키지를 rebuild 하게 하거나, build 를 선택했을 때에만 패키지가 컴파일 되게 하는 등의 선택을 할 수 있다.

## 패키지의 배포

기본적으로 델파이 4 는 모든 어플리케이션 제작에 사용된 모든 컴포넌트와 유닛을 하나의

실행파일에 통합하게 된다. 이런 구조에서 패키지를 이용하는 형태로 전환하려면 Project|Options 메뉴 아이템을 선택해서 패키지를 이용하도록 설정해 주어야 한다.

Project|Options 메뉴 아이템을 선택하면 지금 현재 사용 중인 프로젝트에 대한 여러가지 옵션을 선택할 수 있다. 여기에서 Packages 탭을 클릭하면 패키지에 대한 여러가지 설정을 할 수 있게 된다. 이를 선택한 화면은 다음 그림과 같은데, Component|Install Packages 를 선택했을 때와 거의 비슷한 형태임을 알 수 있을 것이다.



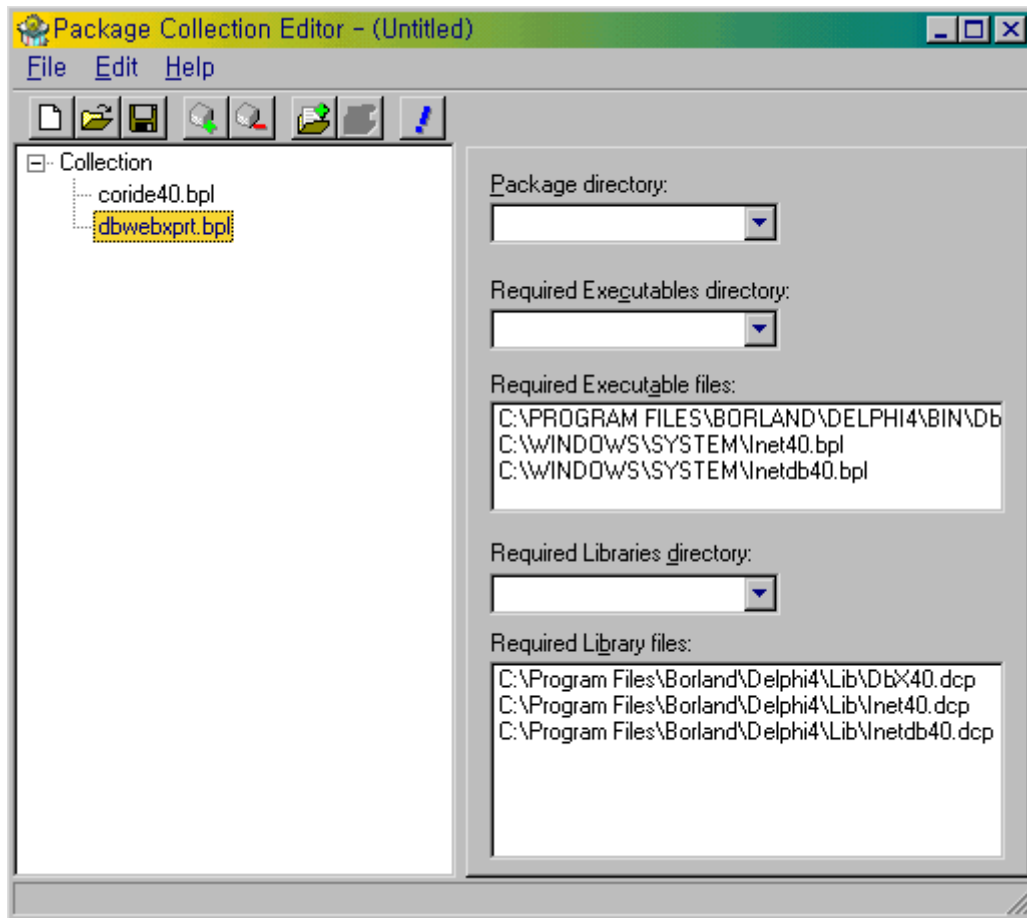
여기에서 보이는 Design Packages 목록에는 컴포넌트 팔레트에 나타나는 디자인 타임 패키지가 선택된다. 그렇지만, 패키지를 배포할 때에는 아래에 있는 런타임 패키지를 선택하는 것이 중요한데, 'Build with runtime packages' 체크 박스를 선택하고, 같이 배포할 런타임 패키지를 여기에 Add 하면 하나의 실행 파일로 만들어질 부분들이 패키지로 분할된다.

## 패키지 컬렉션 (Package Collections)

패키지 컬렉션은 패키지 컬렉션 에디터 유틸리티로 제작한다. 패키지 컬렉션이란 여러 개의 패키지를 하나로 묶어서 편리하게 사용할 수 있게 한 것이다. 필요한 모든 유닛과 패키지를 이를 이용해서 하나로 묶어서 여러 가지의 다른 프로젝트를 하나의 단순한 작업으로 쉽게 작업할 수 있는 장점이 있다.

그러면, 실제로 패키지 컬렉션을 만드는 작업에 대해서 알아보자.

델파이의 Tools 메뉴에서 Package Collection Editor 를 선택하면 다음 그림과 같은 유틸리티 프로그램이 실행된다. 여기에서는 Edit|Add Package 메뉴를 선택해서 두개의 패키지를 추가하였다. 패키지를 모두 추가하고, 패키지 파일의 위치와 필요한 라이브러리의 위치를 지정해 주고 나면 이를 컴파일 한다. 이렇게 하면 DPC 파일이 생성되는데, 이 파일을 이용해서 여러 개의 패키지를 한꺼번에 사용할 수 있게 된다.



## 이 장을 마치며 ...

패키지란 것이 그다지 복잡하지는 않지만, 개발자에게 주는 이득은 상당한 것이다. 지금까지 살펴 보았듯이 패키지를 이용하는 방법은 어플리케이션을 개발할 때 사용하는 컴포넌트를 추가, 삭제, 그리고 지정하는 단순한 작업이다.

디자인 타임 패키지를 잘 이용하면 델파이를 이용해서 개발을 할 때 많은 유연성을 발휘할 수 있다. 런타임 패키지는 어플리케이션의 중복되는 코드 부분을 줄여줄 수 있도록 활용할 수 있으며, 자칫 비대해지기 쉬운 실행 파일을 작게 유지시킬 수도 있다. 뿐만 아니라 적절한 버전 컨트롤이 가능하다면 업그레이드와 유지 보수에도 매우 편리하게 활용할 수 있다.

# 기본적인 컴포넌트의 제작

## (Creating Basic Components)

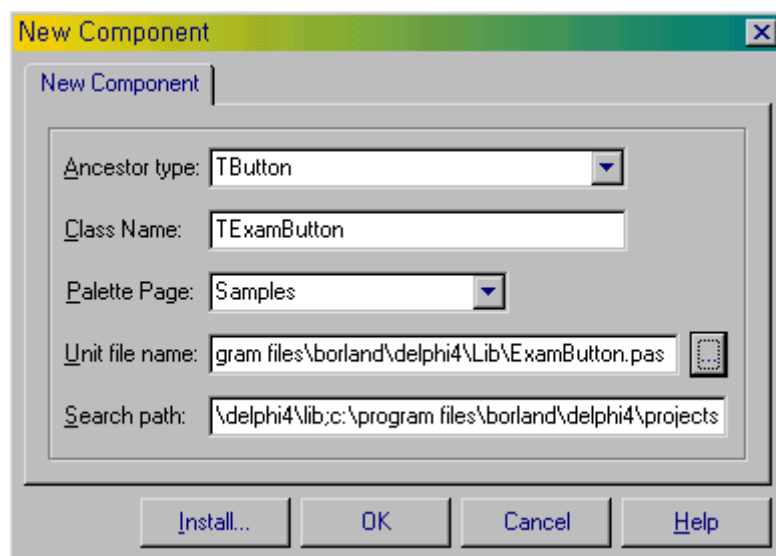
델파이는 가장 잘된 윈도우용 OOP 개발 환경이라고 말할 수 있다. 이러한 델파이의 가장 핵심 부분은 누가 뭐라 해도 델파이의 튼튼한 컴포넌트라고 말할 수 있다. 컴포넌트란 OOP 의 기본 개념을 충실하게 지원하는 델파이의 객체로, 이러한 컴포넌트를 개발하는 방법이야 말로, 델파이 개발자에게는 가장 중요한 기술이라고 말해도 과언이 아니다. 이번 장에서는 컴포넌트가 동작하는 방법을 이해하고 실제로 컴포넌트를 제작하는 방법에 대해서 알아 보기로 한다.

### 컴포넌트의 구조

컴포넌트는 크게 나누어 필드, 메소드, 프로퍼티라는 세가지의 파트로 나뉘어 있다. 필드는 객체 내부의 데이터 변수이며, 메소드는 객체에 속한 프로시저와 함수를 말하고, 프로퍼티란 객체에 속한 데이터와 코드에 접근하는 방법을 제공하는 엔티티이다.

### 간단한 컴포넌트의 제작

컴포넌트를 만드는 가장 간단한 방법은 Component|New Component... 메뉴를 선택해서 Component Expert 를 시작하는 것이다.



여기에서 앞에서와 같이 새로 만들 컴포넌트가 상속할 클래스의 이름과 컴포넌트 클래스의

이름, 그리고 컴포넌트가 위치할 컴포넌트 팔레트 페이지를 지정하면 뼈대가 되는 코드가 만들어진다.

만들어진 코드는 다음과 같다.

```
unit ExamButton;
```

```
interface
```

```
uses
```

```
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
    StdCtrls;
```

```
type
```

```
    TExamButton = class(TButton)
```

```
    private
```

```
        { Private declarations }
```

```
    protected
```

```
        { Protected declarations }
```

```
    public
```

```
        { Public declarations }
```

```
    published
```

```
        { Published declarations }
```

```
end;
```

```
procedure Register;
```

```
implementation
```

```
procedure Register;
```

```
begin
```

```
    RegisterComponents('Samples', [TExamButton]);
```

```
end;
```

```
end.
```

이렇게 뼈대가 만들어지면, 프로퍼티나 메소드 등을 추가해서 간단하게 확장된 새로운 컴포



넌트를 제작할 수 있다. TExamButton 컴포넌트를 클릭하면 간단한 메시지 박스를 표시할 수 있도록 수정해 보자. 이를 구현하려면 protected 섹션에서 Click 메소드를 다음과 같이 오버라이드하면 된다.

```
protected
    procedure Click; override;
```

그리고, 다음과 같이 구현한다.

```
procedure TExamButton.Click;
begin
    inherited Click;
    ShowMessage('클릭 !');
end;
```

inherited 키워드는 오버라이드한 과거의 메소드를 실행하도록 하는 키워드이다. 이렇게 만들어진 컴포넌트는 파일로 저장하고, Component|Install Component 메뉴를 통해서 실제로 인스톨 할 수 있게 된다.

#### 참고: 오버라이드할 메소드 알아내기

컴포넌트를 제작할 때 특정 이벤트에서 다른 동작을 하도록 만들고 싶을 때가 있다. 이럴 때에는 보통 오브젝트 인스펙터에서 관찰할 수 있는 이벤트 이름에서 'On'을 뺀 이름의 메소드를 오버라이드 하면 된다. 앞의 예제에서는 OnClick 이벤트에 대해서 조작을 가하기 위해 'On'을 뺀 Click 메소드를 오버라이드 하였다.

## 상속(Inheritance)의 활용

상속이란 부모 클래스에 속해 있는 메소드와 프로퍼티를 재사용하거나 override 해서 사용할 수 있는 특성을 말하는 것이다. 이를 이용하면 부모 클래스의 기능을 유지한 채, 새로운 기능을 추가해서 향상된 클래스를 쉽게 만들어낼 수 있다.

### ● Protected 프로퍼티의 노출

델파이에는 많은 수의 TCustomXxxx 클래스가 있는데, 이들의 모든 프로퍼티는 protected로 정의되어 있다. 이들은 컴포넌트 팔레트에는 나타나지 않지만 실제로 사용되는 컴포넌트 클래스의 기초가 되는 것들이다. 예를 들어, TCustomEdit 는 TCustimMaskEdit,

TCustomMemo, TDBLookupCombo, TEdit, TSpinEdit 컴포넌트의 공통된 부모 클래스이다. 또한, TCustomMaskEdit 컴포넌트는 TDBEdit, TMaskEdit 의 공통된 부모 클래스가 된다. 이렇게 기초가 되는 부모 TCustomXxxx 클래스를 상속 받아서 이들의 protected 프로퍼티, 이벤트를 오브젝트 인스펙터에 나타나게 하려면 이들을 published 섹션에 재선언만 해주면 된다. 예를 들어, TSpeedButton 클래스는 Align 프로퍼티가 없는데, 이 프로퍼티를 추가하려면 다음과 같이 Align 프로퍼티를 published 섹션에 재선언 해주는 컴포넌트를 만들면 된다.

```
unit AlignBtn;
```

```
interface
```

```
uses
```

```
    Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs, Menus, Buttons;
```

```
type
```

```
    TAlignSpeedButton = class(TSpeedButton)
```

```
    published
```

```
        property Align;           //원래는 protected 섹션에 선언되어 있었음
```

```
    end;
```

```
procedure Register;
```

```
implementation
```

```
begin
```

```
    RegisterComponents('Samples', [TAlignSpeedButton]);
```

```
end;
```

```
end.
```

## ● 상속받은 프로퍼티 감추기

위의 경우와는 반대로, 상속받아 만든 컴포넌트의 프로퍼티나 이벤트를 사용자가 오브젝트 인스펙터를 통해 접근하지 못하도록 하고 싶을 때가 있다. 이럴 때에는 기존의 프로퍼티와 같은 이름의 프로퍼티를 재선언하고, 이를 read-only 로 설정하면 된다. 다음의 코드는 TPanel 컴포넌트에서 Left, Top, Height, Width 프로퍼티를 없앤 컴포넌트 이다.

```

TSnapPanel = class(TPanel)
private
    FDummyProperty: Byte;           //프로퍼티를 숨기기 위해 사용되는 필드

... (중략)

published
    property Height: Byte read FDummyProperty;
    property Left: Byte read FDummyProperty;
    property Top: Byte read FDummyProperty;
    property Width: Byte read FDummyProperty;
end;

```

## ● 가상 메소드의 override

메소드를 override 하는 것이 클래스의 기능을 확장하는 가장 빠른 방법이다.

텔과이로 컴포넌트를 다룰 때에는 각각의 컴포넌트가 publish 하는 다양한 이벤트에 익숙해져야 한다. 새로운 컴포넌트를 상속받을 때에 가장 흔히 하는 실수는 상속받은 컴포넌트의 생성자에서 이벤트 핸들러를 동적으로 생성하고, 여기에 값을 대입하는 것이다. 이렇게 하면, 사용자가 해당 이벤트에 해당되는 핸들러를 사용하게 되면, 생성자에서 만든 이벤트 핸들러는 절대로 호출되지 않는다. 그러므로, 해당 컴포넌트가 이벤트에 반응해야 한다면 이벤트 핸들러를 만들지 말고 처리해야 할 이벤트와 대응되는 가상 메소드를 override 하도록 한다.

문제는 가상 메소드와 연결된 이벤트에 대한 정보를 찾기가 어렵다는 것인데, 이를 알기 위해서는 본래 소스 코드를 봐야 알 수 있겠지만, 일반적으로 VCL에서는 이벤트 이름의 ‘On’ 부분을 뺀 이름이 가상 메소드의 이름이다. 예를 들어, OnClick 이벤트에 해당하는 가상 메소드는 ‘Click’이다. 다음 컴포넌트는 버튼 컴포넌트의 Click 메소드를 override 해서 버튼을 클릭할 때마다 소리를 나게 하는 것이다.

```

unit SountBtn;

```

```

interface

```

```

uses

```

```

    Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs, Menus, StdCtrls;

```

type

TSoundButton = class(TButton)

private

FSoundFile: string;

protected

public

constructor Create(AOwner: TComponent); override;

procedure Click; override;

published

property SoundFile: string read FSoundFile write FSoundFile;

end;

procedure Register;

implementation

uses

MMSystem;

procedure TSoundButton.Click;

begin

sndPlaySound(@FSoundFile, snd\_Async or snd\_NoDefault);

inherited Click;

end;

constructor TSoundButton.Create(AOwner: TComponent);

begin

inherited Create(AOwner);

FSoundFile := '\*.wav';

end;

procedure Register;

begin

RegisterComponents('Samples', [TSoundButton]);

end;

end.

## 윈도우 메시지 핸들러

전통적인 윈도우 프로그래밍에서 가장 중요한 것 중의 하나가 윈도우에 전달된 메시지를 처리하는 것이다. 델파이는 많은 부분을 처리해 준다. 그렇지만, 델파이가 처리하지 못하는 메시지는 개발자가 직접 처리해 주어야 한다.

### ● 메시지-처리 시스템의 이해

모든 델파이 클래스는 메시지를 처리하기 위한 기본적인 방법으로 메시지-처리 메소드나 메시지 핸들러와 같은 방법을 사용한다. 메시지 핸들러의 기본적인 아이디어는 메시지를 받은 클래스가 메시지 종류에 따라 특정 메소드 세트를 호출하는 것이다. 이때, 지정된 메소드가 없을 경우에는 디폴트 핸들러가 실행된다.

다음 그림은 메시지-디스패치(message-dispatch) 시스템에 대한 다이어그램이다.



VCL 은 메시지-디스패치 시스템을 모든 윈도우 메시지를 특정 클래스의 메소드 호출로 처리한다. 개발자가 할 일은 메시지-처리 메소드를 생성하는 것이다.

### 1. 윈도우 메시지

델파이에서의 윈도우 메시지는 몇 개의 필드로 이루어진 데이터 레코드이다. 그 중에서도 가장 중요한 것은 메시지를 확인할 수 있는 정수값이다. 윈도우는 많은 메시지를 정의하고 있다. 그리고, 이런 메시지 들은 Messages.pas 유닛에 선언되어 있으며 이들은 정수값으로 구별된다. 그리고, 메시지 레코드에는 2 개의 파라미터 필드와 1 개의 결과 필드를 포함하고 있다.

하나의 파라미터는 16 비트이며, 다른 하나는 32 비트 값이다. 이를 Win32 에서의 표현 방법을 이용하면 각각 wParam, lParam 에 해당하며, lParam 값의 경우 순서를 나누어 lParamHi, lParamLo 와 같이 접근하는 것이 가능하다.

처음 윈도우를 이용해서 프로그래밍을 할 때에는 API 를 사용할 때, 개발자가 각각의 파라미터의 내용을 찾아봐야 했다. 그런데, 지금은 메시지 크래킹(message cracking)이라고 하는 명명된 파라미터를 사용하기 때문에 이해하는 것이 간단해졌다. 예를 들어, WM\_KEYDOWN 메시지의 파라미터는 nVirtKey, lKeyData 이다.

## 2. 메시지의 디스패칭 (Dispatching Message)

어플리케이션이 윈도우를 생성할 때, 윈도우 프로시저를 윈도우 커널에 등록하게 된다. 이때 윈도우 프로시저는 윈도우로 넘어오는 메시지를 처리하는 루틴을 말한다. 전통적으로 윈도우 프로시저는 각각의 메시지에 대한 커다란 case 문으로 작성했었다. 여기에서 꼭 기억해 두어야 할 것은 윈도우 핸들을 가진 모든 윈도우가 메시지를 처리한다는 것이다. 즉, 새로운 윈도우를 생성할 때마다, 이들은 반드시 완전한 형태의 윈도우 프로시저를 각각 가지고 있어야 하는 것이다.

델파이의 메시지 디스패칭을 다음과 같은 방법으로 단순화 하였다.

- 각각의 컴포넌트는 완전한 메시지-디스패칭 시스템을 상속한다.
- 디스패치 시스템은 디폴트 처리가 된다. 그러므로, 개발자는 디폴트 처리와 다른 부분만 핸들러를 만들어 주면 된다.
- 메시지 처리를 할 때에도, 일부 내용만 수정하고 나머지 처리 부분을 상속해서 처리하면 된다.

## 3. 메시지 흐름의 추적

델파이는 어플리케이션에 있는 각각의 컴포넌트에 대한 윈도우 프로시저로 `MainWndProc` 라는 메소드를 등록한다. `MainWndProc` 메소드에는 예외 처리 블록을 포함하고 있으며, 메시지 구조체를 윈도우에서 `WndProc` 라는 가상 메소드로 넘겨주고, 예외 처리는 클래스의 `HandleException` 메소드를 호출하여 수행된다.

`MainWndProc` 메소드는 특별히 메시지를 처리하지는 않는다. 실제로, 처리하는 부분은 필요에 의해 메소드를 오버라이드할 수 있는 `WndProc` 에서 이루어진다.

`WndProc` 메소드는 메시지를 처리할 때 영향을 미칠 수 있는 특별한 조건 들을 검사해서, 원하지 않는 메시지를 처리할 수 있다. 예를 들어, 드래그를 하고 있을 때 컴포넌트는 키보드 이벤트를 무시한다. 그러므로, `TWinControl` 클래스의 `WndProc` 메소드는 드래그를 하지 않을 때에만 키보드 이벤트를 처리한다. 결국에는 `WndProc` 가 `TObject` 객체에서 상속받은 `Dispatch` 메소드를 호출하게 되며 여기에서 메시지를 처리할 메소드가 어떤 것인지를 결정하게 된다.

`Dispatch` 메소드는 메시지 구조체의 `Msg` 필드를 이용하여, 특정 메시지를 어떻게 디스패치할 것인지를 결정한다. 컴포넌트가 특정 메시지에 대한 핸들러를 정의한다면 `Dispatch` 는 그 메소드를 호출하며, 해당 메시지에 대한 메소드가 없으면 `DefaultHandler` 를 호출한다.

### ● 메시지 처리 방법의 변경

컴포넌트에 대한 메시지 처리 방법을 변경하기 전에, 실제 하려고 하는 것이 무엇인지를 분명히 해야 한다. 델파이는 대부분의 윈도우 메시지를 컴포넌트의 이벤트로 번역해서 처리하므로 많은 경우에는 직접 메시지를 처리하기 보다는 이벤트를 처리하는 것으로 해결이 된다. 메시지 처리 방법을 변경하기 위해서는 메시지를 처리하는 메소드를 오버라이드해야 한다.

### 1. 메시지 핸들러 메소드의 오버라이드

컴포넌트가 특정 메시지를 처리하는 방법을 변경하려면, 메시지를 처리하는 메소드를 오버라이드해야 한다. 컴포넌트에 메시지를 처리하는 메소드가 없는 경우라면, 새로운 메시지 처리 메소드를 선언해야 한다. 메시지 처리 메소드를 오버라이드하기 위해서는, 메소드가 오버라이드하고 있는 것과 같은 메시지 인덱스를 이용해서 새로운 메소드를 선언하면 된다. override 지시어를 사용하는 것이 아니라, 동일한 메시지 인덱스를 이용하여 message 지시어를 사용한다는 것에 주의한다.

예를 들어, WM\_PAINT 메시지를 처리하는 메소드인 WMPaint 메소드를 다음과 같이 선언하여 사용할 수 있다.

```
type
  TMyComponent = class(...)
    ...
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
  end;
```

### 2. 메시지 파라미터의 활용

메시지 핸들러에 넘겨진 파라미터는 var 형이므로, 핸들러에서 파라미터의 값을 변경할 수 있다. Message 파라미터의 데이터 형은 처리할 메시지에 따라 다양하다. 그러므로, 이를 잘 알기 위해서는 윈도우 메시지에 대한 문서를 참고해야 한다. 만약, 과거 스타일로 WParam, LParam 등으로 메시지 파라미터를 표현한 경우에는 이를 TMessage 데이터 형으로 형전환 해서 사용한다.

### 3. 메시지 트래핑

어떤 경우에는 컴포넌트가 메시지를 무시하도록 하고 싶을 때가 있다. 이렇게 메시지를 트랩하도록 하려면 WndProc 메소드를 오버라이드한다. WndProc 메소드는 앞에서 설명했듯

이 Dispatch 메소드가 메시지 처리 메소드를 호출하기 전에 메시지를 처리할 수 있기 때문에, 여기에서 디스패치를 하기 전에 메시지를 거를 수 있다. TWinControl 에서 상속한 컨트롤의 WndProc 를 다음과 같이 오버라이드하여 사용할 수 있다.

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
    inherited WndProc(Message);
end;
```

WndProc 메소드를 오버라이드 하면 메시지의 범위를 거를 수 있고, 메시지를 디스패치하지 않도록 할 수 있기 때문에 핸들러는 호출되지 않는다.

다음의 코드는 TControl 에 대한 WndProc 메소드의 일부이다.

```
procedure TControl.WndProc(var Message: TMessage);
begin
    ...
    if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
        if Dragging then                                //드래그를 하고 있다면,
            DragMouseMsg(TWMMouse(Message))           //마우스 드래그에 해당하는 메시지 처리를 ...
        else
            ...                                          //아니면, 정상적인 처리를 한다.
        end;
    ...
end;
```

## ● 새로운 메시지 핸들러의 작성

텔과이는 대부분의 윈도우 메시지에 대한 핸들러를 제공하기 때문에, 메시지 핸들러를 새로 작성할 필요가 있는 경우는 사용자 정의 메시지를 정의해서 여기에 대한 메시지 핸들러를 작성할 때가 많다.

### 1. 메시지의 정의

많은 수의 표준 컴포넌트 들이 내부적인 사용을 위해 메시지를 정의한다. 이렇게 메시지를 정의하는 이유로 가장 흔한 것이 표준 윈도우 메시지로 처리할 수 없는 정보나, 상태의 변경을 알리기 위한 것이다.



메시지 identifier 는 정수형으로, 윈도우는 특정 번호 이하만 사용하기 때문에 사용자 정의 메시지에 나름대로 번호를 부여해서 사용할 수 있다. WM\_APP 상수는 사용자 정의 메시지의 시작 번호를 대표한다. 그러므로, 메시지를 정의할 때에는 WM\_APP 를 바탕으로 하여 identifier 를 결정해서 사용한다.

그런데, 주의해야 할 것은 표준 윈도우 컨트롤 중에서도 사용자 정의 메시지 범위에 들어가는 메시지를 사용하는 경우가 있다는 점이다. 이런 컴포넌트에는 리스트 박스, 콤보 박스, 에디트 박스와 버튼 등이 있다. 이런 컴포넌트를 이용해서 새로운 메시지를 정의할 때에는 Messages.pas 유닛을 참고하여 컨트롤이 이미 사용하고 있는 윈도우 메시지와 겹치지 않도록 주의해야 한다.

사용자 정의 메시지는 다음과 같이 정의하면 된다.

```
const
```

```
    WM_MYFIRSTMESSAGE = WM_APP + 400;
```

```
    WM_MYSECONDMESSAGE = WM_APP + 401;
```

메시지 레코드는 메시지 처리 메소드로 전송되는 파라미터의 데이터 형이다. 만약 메시지의 파라미터를 사용하지 않거나, 과거와 같이 wParam, lParam 을 사용한 파라미터를 사용할 경우에는 디폴트 메시지 레코드인 TMessage 를 사용하면 된다.

메시지 레코드 데이터 형을 선언하기 위해서는 다음과 같은 규칙을 따라야 한다.

- Msg 레코드의 첫번째 필드는 TMsgParam 데이터 형으로 선언한다.
- 그 다음의 2 바이트를 Word 파라미터로 설정하고 다음 2 바이트는 사용하지 않거나, 4 바이트를 LongInt 파라미터를 설정한다.
- 마지막으로 LongInt 형의 Result 필드를 추가한다.

예를 들어, 모든 마우스 메시지를 처리하는 TWMMouse 메시지 레코드의 선언부를 여기에 소개한다. 가변형 레코드를 사용하여, 같은 파라미터를 2 개의 세트로 정의해서 사용한다.

```
type
```

```
    TWMMouse = record
```

```
        Msg: TMsgParam;           //메시지 ID
```

```
        Keys: Word;               //wParam 에 해당
```

```
        case Integer of           //lParam 을 해석하는 방법이 2 가지 !
```

```
            0: (
```

```
                XPos: Integer;     //x, y 좌표로 접근
```

```
                YPos: Integer);
```

```

1: (
    Pos: TPoint;           //위치
    Result: Longint);      //결과 필드
end;

```

## 2. 새로운 메시지-처리 메소드의 선언

새로운 메시지-처리 메소드는 컴포넌트가 표준 컴포넌트에 의해 처리되지 않는 윈도우 메시지를 처리하거나, 자신의 메시지를 정의해서 사용할 경우에 필요하다.

다음의 코드는 CM\_CHANGE\_COLOR 라는 사용자 정의 메시지에 대한 메시지 핸들러를 선언한 것이다.

```

const
    CM_CHANGE_COLOR = WM_APP + 400;

type
    TMyComponent = class(TControl)
    ...
protected
    procedure CMChangeColor(var Message: TMessage); message CM_CHANGE_COLOR;
end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
    Color := Message lParam;
    inherited;
end;

```

그러면, VCL 에서 TControl 컴포넌트에 OnClick 이벤트를 처리할 수 있도록 하는 부분의 소스 코드를 살펴 보자. 다소 복잡하지만 기본적인 방법은 동일하게 사용되고 있다.

```

TControl = class(TComponent)
private
    FOnClick: TNotifyEvent;
    ... (중략)

```

```

    procedure WMLButtonUp(var Message: TWMLButtonUp); message WM_LBUTTONDOWN;
    ... (중략)

protected
    ... (중략)

    procedure Click; dynamic;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    ... (중략)

end;

... (중략)

procedure TControl.Click;
begin
    if Assigned(FOnClick) then FOnClick(Self);
end;

procedure TControl.WMLButtonUp(var Message: TWMLButtonUp);
begin
    inherited;
    if csCaptureMouse in ControlStyle then
        MouseCapture := False;
    if csClicked in ControlStyle then
        begin
            Exclude(FControlState, csClicked);
            if PtInRect(ClientRect, SmallPointToPoint(Message.Pos)) then
                Click;          //이벤트 핸들러를 호출하게 되는 가상 메소드를 호출한다.
            end;
            DoMouseUp(Message, mbLeft);
        end
    end
end

```

## 컴포넌트에서 그래픽 이용하기

델파이는 윈도우 GDI 를 여러가지 레벨로 캡슐화하고 있다. GDI 함수를 직접 호출할 때에

는 디바이스 컨텍스트에 대한 핸들을 사용하게 된다. 일단 그래픽 이미지를 그리고 나면, 반드시 디바이스 컨텍스트를 원래의 상태로 복귀시키고 이를 처리해야 한다.

10 장에서 이미 그래픽에 대한 내용을 다룬 바 있지만, 컴포넌트를 제작하기 위해서는 여기에 대해서 잘 알고 있어야 한다. 델파이는 컴포넌트에 Canvas 프로퍼티를 제공함으로써 복잡한 GDI 함수를 직접 호출해서 사용하는 것을 대신한다. 캔버스는 리소스를 관리하고, 선택하고, 사용하는 등의 모든 작업을 도맡아 하게 된다.

델파이를 사용할 때의 장점으로는 이밖에도 리소스를 캐쉬하기 때문에 나중에 다시 사용하거나, 반복적인 작업을 할 때 속도의 증진을 기대할 수 있다.

## ● 캔버스의 활용

Canvas 클래스는 윈도우 그래픽을 여러가지 레벨에서 캡슐화 한다. 즉, 도형을 그리거나 라인을 그리고, 텍스트를 그릴 수 있는 고수준 함수에서 부터, 윈도우 GDI 에 접근하는 저수준 함수까지 지원한다. 다음에 캔버스에 대한 내용을 정리하였다.

레 벨	작 업	메소드와 프로퍼티
High	라인과 도형 그리기	MoveTo, LineTo, Rectangle, Ellipse
	텍스트 디스플레이와 측정	TextOut, TextHeight, TextWidth, TextRect
	영역 채우기	FillRect, FloodFill
Intermediate	텍스트와 그래픽 정의	Pen, Brush, Font 프로퍼티
	픽셀 처리	Pixels 프로퍼티
	이미지 복사와 병합	Draw, StretchDraw, BrushCopy, CopyRect 메소드 CopyMode 프로퍼티
Low	GDI 함수 호출	Handle 프로퍼티

## ● Picture 객체 작업 하기

델파이는 캔버스에 직접 그림을 그리는 것 이외에, 비트맵이나 메타 파일, 아이콘 등과 같은 그래픽 이미지를 처리할 수 있는 Picture 객체를 제공한다.

### 1. Picture, graphic, canvas

델파이가 그래픽을 처리하는 클래스에는 3 가지가 있다. 이들을 구별하고 잘 사용하는 것이 중요하다.

앞에서도 간단히 설명했지만, 캔버스는 폼이나, 그래픽 컨트롤, 프린터 또는 비트맵의 표면에 그릴 수 있는 객체이다. 다시 말해서, 실제로 화폭에 그림을 그리는 화가를 연상할 때

화폭에 해당되는 것이 캔버스이다. 보통 캔버스는 독립적인 클래스로 사용하지 않고, 컨트롤의 프로퍼티로 제공된다.

그래픽은 파일이나 리소스의 형태로 접근할 수 있는 그래픽 이미지를 대표한다. 델파이는 TGraphic 클래스를 상속한 TBitmap, TIcon, TMetafile 등의 클래스를 제공한다. TGraphic 클래스는 여러가지 다른 종류의 그래픽에서 공통적으로 사용하는 표준 인터페이스를 정의하고 있다.

Picture 는 그래픽의 컨테이너로, 어떤 그래픽 클래스도 담을 수 있다. 그리고, 어플리케이션은 이들에게 똑같은 방법으로 접근할 수 있는 방법을 제공한다. 예를 들어, Image 컨트롤은 Picture 프로퍼티를 제공하는데 이를 이용하여 여러 종류의 그래픽 이미지를 표시할 수 있다.

## 2. 팔레트 다루기

256 색상의 비디오 모드처럼 팔레트에 기초한 디바이스를 사용할 때 델파이는 자동으로 팔레트의 realization 을 지원한다. 대부분의 컨트롤은 팔레트가 필요없다. 그렇지만, 그래픽 이미지를 표시하는 컨트롤은 이미지를 제대로 표시하기 위해서는 윈도우와 스크린 디바이스 드라이버와 상호 작용해야 한다. 윈도우는 이런 작업을 팔레트의 realization 이라고 한다. 다시 말해, 팔레트를 realize 하는 것은 현재의 윈도우에 모든 팔레트를 사용하도록 하고, 배경 윈도우는 가능한 팔레트를 가장 가까운 색상과 맞도록 표시하는 작업이다. 그러므로, 활성화된 윈도우가 바뀔 경우에 윈도우는 팔레트를 realize 해야 한다.

컨트롤에 대한 팔레트를 지정하기 위해서는 컨트롤의 GetPalette 메소드를 오버라이드하여 팔레트의 핸들을 반환하도록 한다.

컨트롤이 GetPalette 메소드를 오버라이드하여 팔레트를 지정하면, 델파이는 자동으로 윈도우의 팔레트 메시지에 반응한다. 팔레트 메시지를 처리하는 메소드는 PaletteChanged 이다. PaletteChanged 메소드의 주 목적은 컨트롤의 팔레트를 realize 하는 것이다. 윈도우가 팔레트를 realization 할 때에는 활성화된 윈도우에 foreground 팔레트를 가지도록 하고, 다른 background 팔레트를 가지도록 한다. 델파이는 여기에서 추가적으로 윈도우 내에 있는 컨트롤 들의 탭 순서에 따라 팔레트를 realize 한다. 그러므로, 디폴트 핸들러를 오버라이드하여 팔레트를 관리할 필요가 있는 경우는 탭 순서에서 첫번째가 아닌 컨트롤에 foreground 팔레트를 적용하고자 할 경우 이외에는 거의 없다.

### ● Off-screen 비트맵

복잡한 그래픽 이미지를 그릴 때 윈도우 프로그래밍에서 공통적인 테크닉은 off-screen 비트맵을 생성하고, 비트맵에 이미지를 그리고, 스크린에 비트맵을 복사하여 붙여넣는 작업이 있다. Off-screen 이미지를 사용하면, 스크린에 반복적인 그리기 작업을 할 때처럼 깜빡이

는 현상을 줄일 수 있다. 델파이의 비트맵 클래스 역시 off-screen 이미지로 작업할 수 있다.

## ● 변화에 반응하기

모든 그래픽 객체는 객체의 변화에 반응하는 이벤트를 가질 수 있다. 이런 이벤트를 이용하여 컴포넌트가 이미지를 다시 그릴 때 반응하도록 할 수 있는 것이다. 그래픽 객체의 변화를 반영하는 것은 그래픽 객체를 디자인-타임 인터페이스로 사용할 때에 더욱 중요하다. 그래픽 객체의 변화에 반응하려면, 클래스의 OnChange 이벤트에 메소드를 대입해야 한다. 예를 들어, TShape 컴포넌트는 펜, 브러쉬 등의 객체를 프로퍼티로 제공하는데, 컴포넌트의 constructor 에서 메소드를 OnChange 이벤트에 대입하여 컴포넌트가 펜이나 브러쉬가 변경되면 이미지를 refresh 하도록 한다.

type

```
TShape = class(TGraphicControl)
```

```
public
```

```
    procedure StyleChanged(Sender: TObject);
```

```
end;
```

...

implementation

...

```
constructor TShape.Create(AOwner: TComponent);
```

```
begin
```

```
    inherited Create(AOwner);
```

```
    Width := 65;
```

```
    Height := 65;
```

```
    FPen := TPen.Create;
```

```
    FPen.OnChange := StyleChanged;           //OnChange 이벤트에 메소드 대입
```

```
    FBrush := TBrush.Create;
```

```
    FBrush.OnChange := StyleChanged;         //OnChange 이벤트에 메소드 대입
```

```
end;
```

```
procedure TShape.StyleChanged(Sender: TObject);
```

```
begin
```

```
    Invalidate();                           //컴포넌트를 다시 그린다.
```

```
end;
```

## 그래픽 컴포넌트의 제작

순수한 그래픽 컨트롤은 포커스를 가질 수 없기 때문에, 윈도우 핸들을 필요로 하지 않는다. 사용자 들은 컨트롤을 마우스로 조작할 수는 있지만, 키보드 인터페이스는 가질 수 없다. 여기에서는 Additional 페이지에 있는 TShape 와 거의 동일한 TSampleShape 라는 컴포넌트를 만들어 볼 것이다.

### ● 컴포넌트 시작하기

먼저 컴포넌트를 TGraphicControl 에서 상속 받도록 하고, 이름을 TSampleShape 라고 한다. 이 컴포넌트는 Samples 페이지에 등록하도록 한다. 이렇게 설정하면 다음과 같은 뼈대 코드가 만들어질 것이다.

```
unit Shapes;
```

```
interface
```

```
uses
```

```
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

```
type
```

```
    TSampleShape = class(TGraphicControl)
```

```
    private
```

```
    protected
```

```
    public
```

```
    published
```

```
    end;
```

```
procedure Register;
```

```
implementation
```

```
procedure Register;
```

```
begin
```

```
    RegisterComponents('Samples', [TSampleShape]);
```

end;

end.

먼저, TGraphicControl 에서 protected 섹션에 정의된 여러 프로퍼티를 published 섹션에 사용하도록 선언한다. 많은 프로퍼티들은 이미 TGraphicControl 클래스의 published 섹션에 선언되어 있으므로, 마우스 이벤트와 드래그-드롭을 지원하는 이벤트만 publish 하면 된다.

published

```
property DragCursor;
property DragMode;
property OnDragDrop;
property OnDragOver;
property OnEndDrag;
property OnMouseDown;
property OnMouseMove;
property OnMouseUp;
```

end;

## ● 그래픽 기능의 추가

### 1. Shape 프로퍼티의 추가

그래픽 컨트롤은 사용자 입력을 포함한 동적 조건을 반영하여 형태를 바꿀 수 있다. 일반적으로 그래픽 컨트롤의 형태는 이들의 조합이라고 할 수 있다. 예를 들어, TGauge 컨트롤의 경우 형태와 방향을 결정하고, 숫자와 그림을 보여 줄 것인지 여부 등을 프로퍼티에서 결정할 수 있다.

TSampleShape 컨트롤에도 어떤 것을 그릴 것인지 여부를 결정할 수 있도록 Shape 라는 프로퍼티를 제공하도록 한다. 이를 위해 먼저, 프로퍼티로 사용할 열거형(enumerated type)을 정의하도록 하자.

```
TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
```

이제 다음과 같이 프로퍼티를 선언한다.



```

private
    FShape: TSampleShapeType;
    procedure SetShape(Value: TSampleShapeType);
published
    property Shape: TSampleShapeType read FShape write SetShape;
end;

```

그리고, SetShape 메소드를 다음과 같이 구현한다.

```

procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
    if FShape <> Value then
        begin
            FShape := Value;
            Invalidate;
        end;
    end;
end;

```

## 2. 디폴트 프로퍼티의 변경

그래픽 컨트롤의 디폴트 크기는 매우 작다. 그러므로, 이를 수정하려면 다음과 같이 프로퍼티를 선언할 때 default 값을 변경하고 constructor에서 수정할 필요가 있다.

```

public
    constructor Create(AOwner: TComponent); override
end;

published
    property Height default 65;
    property Width default 65;
end;

constructor TSampleShape.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);

```

```

Width := 65;
Height := 65;
end;

```

### 3. 펜과 브러쉬의 publish

펜과 브러쉬를 변경할 수 있게 하려면, 내부적으로 사용할 객체를 private 섹션에 필드로 선언해야 한다.

```

private
  FPen: TPen;
  FBrush: TBrush;
  ...
end;

```

이들을 프로퍼티로 접근할 수 있게 하기 위해 Set 메소드와 프로퍼티를 published 섹션에 추가한다.

```

private
  procedure SetBrush(Value: TBrush);
  procedure SetPen(Value: TPen);
published
  property Brush: TBrush read FBrush write SetBrush;
  property Pen: TPen read FPen write SetPen;
end;

```

그리고, SetBrush 와 SetPen 메소드를 다음과 같이 구현한다.

```

procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);
end;

```

```
end;
```

또한, 이들을 constructor 에서 생성해야 런타임에서 사용할 수 있다. 또한, destructor 에서는 이 객체 들을 해제해야 한다.

```
public
```

```
    destructor Destroy; override;
```

```
end;
```

```
constructor TSampleShape.Create(AOwner: TComponent);
```

```
begin
```

```
    inherited Create(AOwner);
```

```
    Width := 65;
```

```
    Height := 65;
```

```
    FPen := TPen.Create;
```

```
    FBrush := TBrush.Create;
```

```
end;
```

```
destructor TSampleShape.Destroy;
```

```
begin
```

```
    FPen.Free;
```

```
    FBrush.Free;
```

```
    inherited Destroy;
```

```
end;
```

마지막으로 펜과 브러쉬가 변경 되었을 때, 그려진 도형을 다시 그릴 필요가 있다. 펜과 브러쉬 객체는 모두 OnChange 이벤트를 가지고 있으므로, 이들 이벤트를 설정해서 사용하면 된다.

```
published
```

```
    procedure StyleChanged(Sender: TObject);
```

```
end;
```

```
...
```

```
constructor TSampleShape.Create(AOwner: TComponent);
```

```

begin
    inherited Create(AOwner);
    Width := 65;
    Height := 65;
    FPen := TPen.Create;
    FPen.OnChange := StyleChanged;
    FBrush := TBrush.Create;
    FBrush.OnChange := StyleChanged;
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
    Invalidate;
end;

```

## ● 컴포넌트 이미지 그리기

마지막으로 스크린에 그리는 부분을 구현할 차례이다. 그리는 부분은 TGraphicControl 클래스의 추상 메소드인 Paint 를 오버라이드해야 한다. 여기에서 선택된 펜과 브러시를 가지고 선택된 도형을 그린다.

```

protected
    procedure Paint; override;
    ...
end;

```

Paint 메소드는 도형의 형태에 따라 좌표를 사용하는 방법을 달리해서 구현해야 한다. Paint 메소드는 다음과 같이 구현하도록 한다.

```

procedure TSampleShape.Paint;
var
    X, Y, W, H, S: Integer;
begin
    with Canvas do
        begin
            Pen := FPen;

```

```

Brush := FBrush;
W := Width;
H := Height;
if W < H then S := W else S := H;
case FShape of
    sstRectangle, sstRoundRect, sstEllipse:
        begin
            X := 0;
            Y := 0;
        end;
    sstSquare, sstRoundSquare, sstCircle:
        begin
            X := (W - S) div 2;
            Y := (H - S) div 2;
            W := S;
            H := S;
        end;
end;
case FShape of
    sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);
    sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);
    sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);
end;
end;
end;
end;

```

## 객체 결합 (Object Composition)

객체 결합이란 여러 개의 컴포넌트를 하나로 묶어서 새로운 컴포넌트를 만드는 것을 의미한다. 이 기술을 이용하면 각각의 요소가 되는 컴포넌트의 이벤트와 프로퍼티에 사용자가 함부로 접근하지 못하도록 보호할 수 있다. 비주얼 컴포넌트의 결합에는 컨테이너 컴포넌트(TPanel, TCustomPanel, TWinControl) 등이 요구되며, 이들은 서브-컴포넌트의 부모 윈도우가 된다.

이 테크닉은 복잡한 컴포넌트를 그룹화하거나 컴포넌트 간의 유기적인 관계가 필요한 경우에 유용하게 사용될 수 있다.

다음의 예제 컴포넌트는 OK, Cancel 의 2 개의 버튼을 가지고 있으며, 각각의 컴포넌트에 대한 OnClick, Caption 프로퍼티를 제공한다. 참고로, 각각의 버튼이 Caption 프로퍼티를 가지므로 앞에서 설명한 프로퍼티 숨기기를 이용하여 OKCancel 버튼 자체의 Caption 프로퍼티는 없애 버린다 (FDummyProperty 를 이용).

```
unit OKCancel;
```

```
interface
```

```
uses
```

```
Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs, Menus, Buttons;
```

```
type
```

```
TOKCancelButton = class(TWinControl)
```

```
    //여기에 서브-컴포넌트들을 선언한다.
```

```
    OKButton: TBitBtn;
```

```
    CancelButton: TBitBtn;
```

```
private
```

```
    FOnClick_OKButton: TNotifyEvent;
```

```
    FOnClick_CancelButton: TNotifyEvent;
```

```
    FDummyProperty: string;
```

```
    procedure SetCaption_OKButton(Value: TCaption);
```

```
    function GetCaption_OKButton: TCaption;
```

```
    procedure SetCaption_CancelButton(Value: TCaption);
```

```
    function GetCaption_CancelButton: TCaption;
```

```
protected
```

```
    procedure Click_OKButton(Sender: TObject); virtual;
```

```
    procedure Click_CancelButton(Sender: TObject); virtual;
```

```
public
```

```
    constructor Create(AOwner: TComponent); override;
```

```
published
```

```
    property Caption_OKButton: TCaption read GetCaption_OKButton
```

```
        write SetCaption_OKButton;
```

```
    property Caption_CancelButton: TCaption read GetCaption_CancelButton
```

```

        write SetCaption_CancelButton;
property OnClick_OKButton: TNotifyEvent read FOnClick_OKButton
        write FOnClick_OKButton;
property OnClick_CancelButton: TNotifyEvent read FOnClick_CancelButton
        write FOnClick_CancelButton;
property Caption: string read FDummyProperty;
end;

```

procedure Register;

참고로, 여기까지 선언부를 작성하고 Ctrl+Shift+C 키를 누르면 다음과 같은 구현 부분에 대한 뼈대 코드가 모두 자동으로 만들어진다. 이 기능이 바로 클래스 완료(Class Completion)이다. 컴포넌트를 제작하는 사람 들에게는 허드렛일을 많이 줄여줄 수 있는 기능이다.

어쨌든 구현부분은 다음과 같이 구현한다.

implementation

```

procedure TOKCancelButton.SetCaption_OKButton(Value: TCaption);
begin
    OKButton.Caption := Value;
end;

```

```

function TOKCancelButton.GetCaption_OKButton: TCaption;
begin
    Result := OKButton.Caption;
end;

```

```

procedure TOKCancelButton.SetCaption_CancelButton(Value: TCaption);
begin
    CancelButton.Caption := Value;
end;

```

```

function TOKCancelButton.GetCaption_CancelButton: TCaption;
begin
    Result := CancelButton.Caption;
end;

```

end;

procedure TOKCancelButton.Click\_OKButton(Sender: TObject);

begin

if Assigned(FOnClick\_OKButton) then FOnClick\_OKButton(Self);

end;

procedure TOKCancelButton.Click\_CancelButton(Sender: TObject);

begin

if Assigned(FOnClick\_CancelButton) then FOnClick\_CancelButton(Self);

end;

constructor TOKCancelButton.Create(AOwner: TComponent);

begin

inherited Create(AOwner);

Width := 75;

Height := 60;

OKButton := TBitBtn.Create(Self);

with OKButton do

begin

Parent := Self;

Kind := bkOK;

SetBounds(0, 0, 75, 30);

TabOrder := 0;

OnClick := Click\_OKButton;

end;

CancelButton := TBitBtn.Create(Self);

with CancelButton do

begin

Parent := Self;

Kind := bkCancel;

SetBounds(0, 31, 75, 30);

TabOrder := 1;

OnClick := Click\_CancelButton;

end;

FDummyProperty := '';



end;

procedure Register;

begin

RegisterComponents('Samples', [TOKCancelButton]);

end;

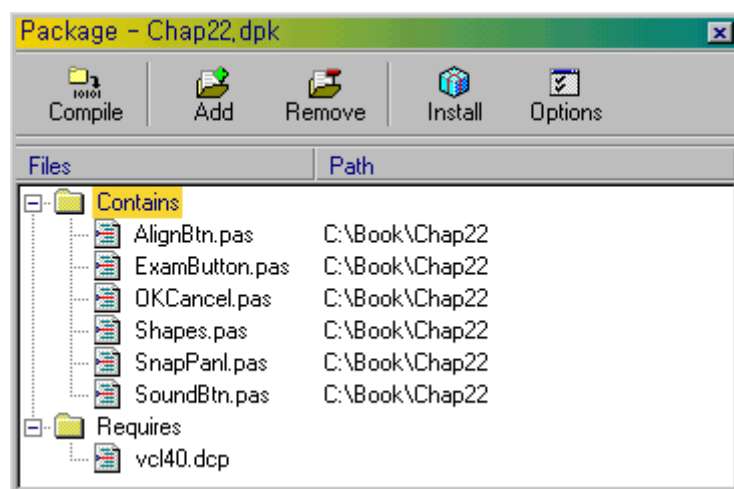
end.

이와 같이 컴포넌트를 제작할 때, 내부에서 제어할 수 있는 컴포넌트를 활용하면 훨씬 강력하고 편리한 활용이 가능하다. 이 밖에도 private 섹션에 FComponentName 과 같은 형태로 내부에서 생성하여 관리할 컴포넌트를 선언해서, 필드 변수로 활용하고 프로퍼티를 선언해서 사용하는 방법이 있다.

## 패키지 제작과 컴포넌트 추가

비록 기본적인 예제 컴포넌트이긴 했지만, 이번 장에서 제작한 컴포넌트는 모두 6 개이다. 이들을 하나의 패키지로 묶어 보도록 하자. 먼저, File|New 메뉴를 선택한 후 객체 저장소에서 Package 를 더블 클릭하여 새로운 패키지를 하나 생성한다. 그리고, File|Save As 명령을 선택하여 적당한 디렉토리에 .DPK 파일을 저장한다. 여기서는 chap22.dpk 라는 이름으로 저장하도록 한다.

패키지 에디터 화면에서 Add 버튼을 클릭하고 다음과 같이 앞에서 제작한 6 개의 컴포넌트를 차례대로 추가한다.



패키지 파일을 저장하고, Install 버튼을 누르면 패키지가 설치될 것이다. 컴파일의 끝

컴포넌트 팔레트의 Samples 페이지에는 6 개의 컴포넌트가 추가될 것이다. 그리고, 이 명령에 의해 .BPL 파일이 생성되는데 이 파일을 따로 배포할 수도 있다.

어떤가 ? 컴포넌트를 만들고, 이들을 자신의 패키지로 엮어서 배포하는 일이 생각보다 너무나 쉽다고 생각될 것이다. 텔파이 4 에서는 이와 같이 컴포넌트를 개발하고, 이를 관리하는 강력한 수단을 제공하고 있다.

## 정 리 (Summary)

이번 장에서는 가장 기본적인 컴포넌트를 만드는 방법과 이들을 이용하여 패키지로 묶는 방법에 대해서 알아보았다. 컴포넌트는 텔파이의 핵심이라고 말할 수 있는 부분이다.

다음 장에서는 기본적인 컴포넌트에 기능을 추가하는 방법과 데이터 인식 컨트롤을 제작하는 방법과 같이 보다 고급스러운 컴포넌트 제작 기법에 대해서 알아볼 것이다.

# 컴포넌트 제작의 깊은 곳

## (Advanced Components Writing Techniques)

### 그리드 컴포넌트의 제작

비교적 복잡하면서도 유용하게 사용할 수 있는 컴포넌트가 그리드 컴포넌트 들이다. 델파이는 그리드를 작성하기 쉽도록 TCustomGrid 라는 기초 컴포넌트를 제공하고 있다. 그러면, 이 컴포넌트를 바탕으로 해서 달력 컴포넌트를 하나 만들어 보도록 하자.

TCustomGrid 컴포넌트를 상속하도록 하고, 컴포넌트의 이름을 TSampleCalendar 라고 하자. 이 컴포넌트 역시 Samples 페이지에 등록하도록 한다.

- 상속된 프로퍼티 publish 와 초기값 변경

TCustomGrid 는 추상적인 그리드 컴포넌트이기 때문에, 많은 수의 protected 프로퍼티를 제공한다. 그러므로, 다음과 같이 필요한 프로퍼티를 published 섹션에 선언해 주도록 한다.

published

property Align;  
property BorderStyle;  
property Color;  
property Ctl3D;  
property Font;  
property GridLineWidth;  
property ParentColor;  
property ParentFont;  
property OnClick;  
property OnDbClick;  
property OnDragDrop;  
property OnDragOver;  
property OnEndDrag;  
property OnKeyDown;  
property OnKeyPress;  
property OnKeyUp;

```
end;
```

달력은 기본적으로 행과 열의 수가 고정되어 있기 때문에, ColCount 나 RowCount 같은 프로퍼티를 publish 할 필요가 없다. 그렇지만, 이런 초기 값을 constructor 에서 설정할 필요가 있다.

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    ColCount := 7;
    RowCount := 7;                                //머릿글 포함
    FixedCols := 0;
    FixedRows := 1;                                //요일 표시 하는 행
    ScrollBars := ssNone;
    Options := Options - [goRangeSelect] + [goDrawFocusSelected]; //범위 선택을 할 수 없다.
end;
```

#### ● 셀 크기 조절과 내부 채우기

사용자나 어플리케이션이 컨트롤의 크기를 변경하면, 윈도우는 WM\_SIZE 메시지를 받게 된다. 컴포넌트는 이 메시지에 맞추어 이미지를 다시 그릴 필요가 있다. 이를 위해서 WM\_SIZE 메시지에 반응하는 메시지 처리 메소드를 추가할 필요가 있다. 먼저 protected 섹션에 다음과 같이 메시지 처리 메소드를 선언한다.

```
procedure WMSize(var Message: TWMSize); message WM_SIZE;
```

그리고, 다음과 같이 구현한다.

```
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
    GridLines: Integer;
begin
    GridLines := 6 * GridLineWidth;                //전체 줄의 크기
    DefaultColWidth := (Message.Width - GridLines) div 7;    //새로운 셀의 폭
    DefaultRowHeight := (Message.Height - GridLines) div 7;    //새로운 셀의 높이
end;
```

그리드 컨트롤은 셀 단위로 내부를 채우게 된다. 그러므로, 달력 컴포넌트라면 각각의 셀마다 날짜를 계산해서 그려야 한다. 그리드 셀은 DrawCell 가상 메소드를 호출하여 그리게 되므로, 이 메소드를 오버라이드해야 한다.

먼저 첫번째 행의 요일을 해당되는 열에 그리도록 한다.

protected 섹션에 DrawCell 메소드를 선언하고, 이를 다음과 같이 구현한다.

```
procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); override;
```

```
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
```

```
    AState: TGridDrawState);
```

```
begin
```

```
    if ARow = 0 then
```

```
        Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);
```

```
end;
```

## ● 날짜 계산

달력 컨트롤을 만들 때 중요한 것은 사용자나 어플리케이션이 연, 월, 일을 설정할 수 있는 기전을 제공해야 한다는 것이다. 델파이는 날짜와 시간은 TDateTime 데이터 형의 변수에 저장한다. 그러므로, 날짜 자체는 TDateTime 데이터 형 변수에 저장하되 Day, Month, Year 프로퍼티를 제공하여 쉽게 날짜를 조정할 수 있도록 해야 한다.

### 1. 날짜의 저장

날짜를 저장하기 위해서는 먼저 TDateTime 데이터 형의 필드 변수를 하나 선언하고, 이를 constructor 에서 현재의 날짜를 얻어서 저장한다.

```
private
```

```
    FDate: TDateTime;
```

```
constructor TSampleCalendar.Create(AOwner: TComponent);
```

```
begin
```

```
    inherited Create(AOwner);
```

```
    ...
```

```
    FDate := Date;
```

```
end;
```

그리고, 런타임에서 접근할 수 있는 프로퍼티를 하나 선언한다.

```
TSampleCalendar = class(TCustomGrid)
private
    procedure SetCalendarDate(Value: TDateTime);
public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
    ...

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
    FDate := Value;
    Refresh;           //화면을 업데이트
end;
```

## 2. Day, Month, Year 프로퍼티 선언

먼저 다음과 같이 프로퍼티를 선언한다. 특이하게 보일지도 모르겠지만 하나의 날짜에 의해서 이들 프로퍼티는 동시에 변경되고, 설정되므로 이들을 각각의 Get, Set 메소드로 구현할 필요가 없다. 이렇게 중복되는 부분을 같은 접근 메소드(access method)를 이용해서 사용할 때에는 index 를 사용하면 유용하다.

```
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;
    ...
```

그러면 GetDateElement, SetDateElement 메소드를 다음과 같이 선언하고 구현하면 된다.

```
type
    TSampleCalendar = class(TCustomGrid)
private
    function GetDateElement(Index: Integer): Integer
```

```

    procedure SetDateElement(Index: Integer; Value: Integer);
    ...

function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
    AYear, AMonth, ADay: Word;
begin
    DecodeDate(FDate, AYear, AMonth, ADay);
    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
        begin
            DecodeDate(FDate, AYear, AMonth, ADay);
            case Index of
                1: AYear := Value;
                2: AMonth := Value;
                3: ADay := Value;
                else Exit;
            end;
            FDate := EncodeDate(AYear, AMonth, ADay);
            Refresh;
        end;
    end;
end;

```

그렇게 어려운 코드는 아니다. EncodeDate 와 DecodeDate 를 이용하면 TDateTime 데이터 형과 Year, Month, Day 의 정수형 사이를 자유롭게 변환할 수 있다는 것을 알아두면 된

다.

### 3. 날짜 표시하기

날짜를 달력에 그릴 때에는 달마다 일수가 다르고, 주어진 연도가 윤년인지 여부를 고려해야 한다. IsLeapYear 함수를 이용하면 해당 연도가 윤년인지 알아볼 수 있으며, SysUtils.pas 유닛의 MonthDays 배열을 이용하면 해당 달의 날짜 수를 알아낼 수 있다. 일단 윤년 정보와 그 달의 날짜 수를 알게 되면, 그리드에서 각 날짜의 위치를 계산할 수 있게 된다.

먼저 그 달에서 첫번째 날짜가 적당한 요일에 위치하는지를 계산할 때 사용할 오프셋을 저장할 필드와 필드 값을 업데이트할 메소드를 선언하고 이를 다음과 같이 구현한다.

```
type
```

```
TSampleCalendar = class(TCustomGrid)
```

```
private
```

```
    FMonthOffset: Integer;
```

```
...
```

```
protected
```

```
    procedure UpdateCalendar; virtual;
```

```
end;
```

```
...
```

```
procedure TSampleCalendar.UpdateCalendar;
```

```
var
```

```
    AYear, AMonth, ADay: Word;
```

```
    FirstDate: TDateTime;           //그 달의 첫번째 날짜
```

```
begin
```

```
    if FDate <> 0 then
```

```
    begin
```

```
        DecodeDate(FDate, AYear, AMonth, ADay);
```

```
        FirstDate := EncodeDate(AYear, AMonth, 1);
```

```
        FMonthOffset := 2 - DayOfWeek(FirstDate); //오프셋을 초기화한다.
```

```
    end;
```

```
    Refresh;           //달력을 다시 그린다.
```

```
end;
```



그리고, constructor 와 SetCalendarDate, SetDateElement 메소드에 UpdateCalendar 메소드를 호출하도록 수정한다.

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    ...
    UpdateCalendar;
end;
```

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
    FDate := Value;
    UpdateCalendar;
end;
```

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ...
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
end;
end;
```

이제는 셀의 행과 열의 번호를 넘겨 주면, 이 위치가 그 달의 몇 번째 날인지 계산하는 메소드를 다음과 같이 구현한다.

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
    Result := FMonthOffset + ACol + (ARow - 1) * 7;
    if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
        Result := -1; //셀의 위치의 날짜가 유효하지 않다.
end;
```

이들 메소드를 이용하면, 날짜의 위치를 알 수 있다. 그러면, DrawCell 메소드를 다음과 같이 구현하여 날짜를 표시하도록 한다.

```

procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
    TheText: string;
    TempDay: Integer;
begin
    if ARow = 0 then                                     //헤더이면 ...
        TheText := ShortDayNames[ACol + 1]              //요일을 표시한다.
    else
        begin
            TheText := '';                                //일단 셀을 비운다.
            TempDay := DayNum(ACol, ARow);                //셀의 위치에 따른 날짜를 구한다.
            if TempDay <> -1 then TheText := IntToStr(TempDay); //결과가 유효할 때만 사용 !
        end;
        with ARect, Canvas do
            TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
                Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText); //셀의 중앙에 날짜 표시
        end;
    end;
end;

```

#### 4. 현재 날짜의 선택

이제 선택된 날짜를 표시하도록 해보자. 이를 위해서는 UpdateCalendar 메소드를 Refresh 메소드를 호출하기 전에 Row, Column 프로퍼티를 설정하도록 수정해야 한다.

```

procedure TSampleCalendar.UpdateCalendar;
begin
    if FDate <> 0 then
        begin
            ...
            Row := (ADay - FMonthOffset) div 7 + 1;
            Col := (ADay - FMonthOffset) mod 7;
        end;
        Refresh;
    end;
end;

```

#### ● 날짜 변경

날짜 사이를 이동할 때에는 화살표 키를 이용하여 이동할 수도 있고, 마우스를 클릭할 수도 있다. 이들을 모두 처리해 주어야 하는 것이 중요하다.

선택된 날짜를 변경해 주어야 할 것이다. 기본적으로 그리드는 화살표 키를 누르거나, 마우스를 클릭할 때 선택된 셀을 옮기는 것을 처리하게 되어 있다. 그렇지만, 달력의 경우 약간의 수정이 필요하다. 이를 위해, 그리드의 Click 메소드를 오버라이드해야 한다.

```
procedure TSampleCalendar.Click;
var
    TempDay: Integer;
begin
    inherited Click;
    TempDay := DayNum(Col, Row);
    if TempDay <> -1 then Day := TempDay;
end;
```

날짜가 변경되었을 때 이벤트를 발생시킬 수 있다면 좋을 것이다. 그러면, OnChange 이벤트를 추가하도록 하자. 먼저 다음과 같이 프로시저 형을 선언하고 이벤트를 추가하자.

```
type
    TSampleCalendar = class(TCustomGrid)
    private
        FOnChange: TNotifyEvent;
    protected
        procedure Change; dynamic;
    ...
    published
        property OnChange: TNotifyEvent read FOnChange write FOnChange;
    ...
```

그리고, Change 메소드를 다음과 같이 구현한다.

```
procedure TSampleCalendar.Change;
begin
    if Assigned(FOnChange) then FOnChange(Self);
end;
```

그리고, SetCalendarDate 와 SetDateElement 메소드에 Change 메소드를 호출하는 부분을 추가한다.

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
```

```
begin
```

```
    FDate := Value;
```

```
    UpdateCalendar;
```

```
    Change;
```

```
end;
```

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
```

```
begin
```

```
    ...
```

```
    FDate := EncodeDate(AYear, AMonth, ADay);
```

```
    UpdateCalendar;
```

```
    Change;
```

```
end;
```

```
end;
```

이렇게 만든 달력 컴포넌트는 비어 있는 셀이 존재하게 되는데, 날짜가 찍혀 있지 않은 셀을 선택할 경우 선택이 되지 않도록 해야 할 것이다. 이를 위해서는 SelectCell 메소드를 오버라이드하여 구현해야 한다.

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
```

```
begin
```

```
    if DayNum(ACol, ARow) = -1 then Result := False
```

```
    else Result := inherited SelectCell(ACol, ARow);
```

```
end;
```

이것으로 간단한 달력 컴포넌트를 그리드를 이용해서 작성해 보았다. 비록 볼품도 없고, 단순한 컴포넌트이지만, 그리드를 이용하여 여러가지 컴포넌트를 개발하는 데에는 참고가 될만한 것이다.

## 데이터 인식 컨트롤의 제작

텔과이는 데이터 소스와 연결해서 데이터를 보여주고, 편집할 수 있는 여러 가지 컨트롤 들을 제공한다. 이런 데이터 인식 컨트롤을 제작하기 위해서는 몇 가지 고려해야 할 공통적인 것들이 있다. 단순한 단일 필드와 연결된 데이터 인식 컨트롤을 만들 수도 있고, 여러 레코드를 관리하는 컨트롤을 만들 수도 있다.

컨트롤과 데이터베이스를 연결하는 것은 데이터 링크 클래스를 통해서 다루어진다. 데이터 링크 클래스 중에서 단일 필드와 연결할 때에는 보통 TFieldDataLink 클래스를 사용한다. 물론 전체 테이블과 연결할 수 있는 데이터 링크 클래스도 있다. 여기에서는 필자가 작성한 TDBTree 컴포넌트를 바탕으로 설명하겠지만, 지면 관계상 570 라인 가까이에 이르는 전체 소스를 설명할 수는 없고 데이터 인식 컨트롤을 구현하는 방법을 중심으로 설명하도록 한다. 그 밖에도 소스를 분석하면 객체를 저장하고, 이를 활용하는 등의 여러가지 테크닉을 배울 수 있을 것이다.

### ● TFieldDataLink 클래스

TFieldDataLink 클래스는 컨트롤과 데이터 소스 사이의 연결을 구성한다. 그러므로, 데이터 인식 컨트롤을 만들 때에는 이 클래스에 대한 이해가 필수적이라고 할 수 있다.

주요 메소드와 프로퍼티를 소개하면 다음과 같다.

메소드/프로퍼티	내 용
Edit	편집 가능한 레코드를 편집 모드로 만든다. 데이터 소스가 읽기 전용이면 False를 반환한다.
Modified	컨트롤이 데이터를 변경할 때 호출해야 한다. 데이터 링크는 레코드가 바뀌지 않았던 데이터 소스를 제공하며, 변경된 데이터를 요구할 때 OnUpdateData 이벤트 핸들러를 호출한다.
Reset	데이터 소스의 데이터를 변경하지 않거나, 새로운 데이터를 검색할 때 호출한다.
CanModify	읽기 전용으로 필드의 값을 수정할 수 있는지 여부를 결정한다.
Control	읽기 전용으로 연결된 데이터 컨트롤을 가리킨다. 데이터 링크가 입력 포커스를 받는 컨트롤을 결정하기 위해 Control 속성을 사용한다.
Editing	읽기 전용으로 데이터 소스의 State 속성이 dsEdit, dsInsert, dsSetKey 중 하나 인지 나타낸다.
Field	읽기 전용으로 연결된 필드 클래스를, 연결된 것이 없으면 nil을 반환한다.
FieldName	데이터베이스 필드 이름을 지정한다.
OnActiveChange	데이터 소스의 Active 프로퍼티가 변경될 때 호출되는 이벤트이다.
OnDataChange	데이터 레코드가 변한 후에 호출되는 이벤트이다. 모든 데이터 인식 컨트롤은

	여기에 대한 핸들러를 구현해야 한다. 이 핸들러가 없으면 컨트롤이 필드로부터 데이터를 검색할 때 알 수 있는 방법이 없다.
OnEditingChange	데이터 소스가 편집모드로 들어가거나 나올 때 호출되는 이벤트이다.
OnUpdateData	컨트롤로부터 변경된 데이터를 소스가 요구할 때 호출되는 이벤트이다. 여기에 대한 핸들러는 연결된 필드 구성요소의 데이터를 저장한다.
UpdateRecord	데이터 소스가 데이터 컨트롤에 의해 수정된 데이터를 가진 현재 레코드의 복사본을 수정하려 할 때 호출된다. 데이터 컨트롤에서 포커스를 잃을 때 호출된다.
BufferCount	사용가능한 레코드 버퍼의 개수를 반환한다. 그리드와 같은 다중 레코드 컨트롤을 구현할 때에 이를 이용하여 사용자가 한 번에 많은 수의 값을 편집할 때 성능을 향상시킬 수 있다. 디폴트 값은 1 이다.
DataSet	읽기 전용으로 데이터 소스로 연결된 데이터 세트를 되돌려 준다.
DataSource	연결된 데이터 소스를 지정한다.
RecordCout	현재 데이터 링크에 의해 버퍼되어진 레코드 수를 반환한다.

데이터 인식 컨트롤은 자체적인 데이터 링크 클래스를 가지고 있으며, 컨트롤이 데이터 링크 객체를 생성하고 파괴하는 책임을 가지고 있다.

type

```
TDBTree = class(TCustomTreeView)
private
    FDataTitleLink: TFieldDataLink;
    ...
end;
```

모든 데이터 인식 컨트롤은 데이터를 컨트롤에 제공하는 데이터 소스를 지정하는 DataSource 프로퍼티를 가지고 있다. 또한, 연결할 데이터 소스의 필드를 지정하는 DataField 프로퍼티를 가지고 있다. 물론, 보통의 컨트롤은 하나의 필드와 데이터 컨트롤을 연결하게 되므로 이렇게 DataField 프로퍼티로 연결하면 되지만, 컨트롤에 따라서는 여러 개의 필드를 설정해야 하는 경우도 있을 것이다. 보통의 경우에는 DataField 프로퍼티를 하나만 가지면 되지만, 트리 구조를 데이터베이스에 저장하기 위해서는 자신의 ID 와 Parent, Index 를 저장할 3 개의 필드와 타이틀의 내용을 저장할 필드가 필요하다. 물론 대표적인 필드는 Title 을 저장할 필드를 하나만 지정하도록 published 프로퍼티로 지정하고 이를 DataField 프로퍼티로 사용하고, 나머지는 런타임에서 접근할 수 있도록 public 섹션에 선언해서 사용하도록 한다. 이들 프로퍼티를 접근하고, 설정할 접근 메소드를 설정하고 구현할 때에는 데이터 링크 클래스를 이용한다.

```

TDBTree = class(TCustomTreeView)
private
    FIDFieldName: string;
    FParentFieldName: string;
    FIndexFieldName: string;
    function GetDataTitleField: string;
    function GetDataSource: TDataSource;
    function GetTitleField: TField;
    function GetIDField: TField;
    function GetParentField: TField;
    function GetIndexField: TField;
    function GetDataSet: TDataSet;
    procedure SetDataTitleField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
    ...
public
    property TreeDataSet: TDataSet read GetDataSet;
    property IDField: TField read GetIDField;
    property ParentField: TField read GetParentField;
    property IndexField: TField read GetIndexField;
    property TitleField: TField read GetTitleField;
    ...
published
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    property DataField: string read GetDataTitleField write SetDataTitleField;
    ...
end;

function TDBTree.GetDataSource: TDataSource;
begin
    Result := FDataTitleLink.DataSource;
end;

procedure TDBTree.SetDataSource(Value: TDataSource);
begin
    FDataTitleLink.DataSource := Value;

```

```

    if Value <> nil then Value.FreeNotification(Self);
end;

function TDBTree.GetDataSet: TDataSet;
begin
    Result := FDataSetLink.DataSet;
end;

function TDBTree.GetDataTitleField: string;
begin
    Result := FDataSetLink.FieldName;
end;

procedure TDBTree.SetDataTitleField(const Value: string);
begin
    FDataSetLink.FieldName := Value;
end;

procedure TDBTree.SetIDField(FieldName: string);
begin
    if TreeDataSet.FindField(FieldName) <> nil then
        FIDFieldName := FieldName;
end;

procedure TDBTree.SetParentField(FieldName: string);
begin
    if TreeDataSet.FindField(FieldName) <> nil then
        FParentFieldName := FieldName;
end;

procedure TDBTree.SetIndexField(FieldName: string);
begin
    if TreeDataSet.FindField(FieldName) <> nil then
        FIndexFieldName := FieldName;
end;

```



```
function TDBTree.GetTitleField: TField;
```

```
begin
```

```
    Result := FDataTitleLink.Field;
```

```
end;
```

```
function TDBTree.GetIDField: TField;
```

```
begin
```

```
    if TreeDataSet.FindField(FIDFieldName) <> nil then
```

```
        Result := TreeDataSet.FieldByName(FIDFieldName)
```

```
    else MessageDlg('해당되는 ID 필드가 없습니다 !', mtError, [mbOK], 0);
```

```
end;
```

```
function TDBTree.GetParentField: TField;
```

```
begin
```

```
    if TreeDataSet.FindField(FParentFieldName) <> nil then
```

```
        Result := TreeDataSet.FieldByName(FParentFieldName)
```

```
    else MessageDlg('해당되는 Parent 필드가 없습니다 !', mtError, [mbOK], 0);
```

```
end;
```

```
function TDBTree.GetIndexField: TField;
```

```
begin
```

```
    if TreeDataSet.FindField(FIndexFieldName) <> nil then
```

```
        Result := TreeDataSet.FieldByName(FIndexFieldName)
```

```
    else MessageDlg('해당되는 Index 필드가 없습니다 !', mtError, [mbOK], 0);
```

```
end;
```

그리고, constructor 와 destructor 에서 데이터 링크 객체를 생성하고, 파괴하는 것이 중요하다.

```
public
```

```
    constructor Create(AOwner: TComponent); override;
```

```
    destructor Destroy; override;
```

```
    ...
```

```
constructor TDBTree.Create(AOwner: TComponent);
```

```
begin
```

```

inherited Create(AOwner);
inherited ReadOnly := True;
ControlStyle := ControlStyle + [csReplicatable];
FDataTitleLink := TFieldDataLink.Create;
FDataTitleLink.Control := Self;
FIDFieldName := 'ID';
FParentFieldName := 'Parent';
FIndexFieldName := 'Index';
FMemorySaving := True;
end;

destructor TDBTree.Destroy;
var
    i: integer;
begin
    i := 0;
    FDataTitleLink.Free;
    inherited Destroy;
end;

```

데이터 링크를 이용하여 필드와 데이터 소스를 설정하는 것까지 했으면, 절반은 끝난 셈이다. 이제는 데이터의 변화를 컨트롤에 반영하도록 구현하는 것이 중요하다. 보통의 경우 이 때에는 데이터 링크 클래스의 `OnChange` 이벤트를 이용한다. 데이터 소스에서 데이터의 변화를 감지하면, 데이터 링크 객체는 `OnChange` 이벤트 핸들러를 호출하게 된다. 그러므로, `DataChange` 메소드를 구현하고, 이 메소드를 `OnChange` 이벤트에 연결하는 것이 중요하다. 그러나, `DBTree`에서는 이를 이용하지 않고 따로 `public` 메소드들을 제공하여 구현하였다. 표준적인 방법이 아니기 때문에 따로 설명하지는 않는다. 대신 일반적인 데이터 인식 컨트롤을 구현할 때에는 앞에서 설명한 방법을 사용하는데, 여기에서는 `TDBEdit` 컨트롤이 어떻게 구현되었는지 소개한다.

```

TDBEdit = class(TCustomMaskEdit)
private
    ...
    procedure DataChange(Sender: TObject);

procedure TDBEdit.DataChange(Sender: TObject);

```

```

begin
  if FDataLink.Field <> nil then
    begin
      if FAlignment <> FDataLink.Field.Alignment then
        begin
          EditText := ''; {forces update}
          FAlignment := FDataLink.Field.Alignment;
        end;
      EditMask := FDataLink.Field.EditMask;
      if not (csDesigning in ComponentState) then
        begin
          if (FDataLink.Field.DataType = ftString) and (MaxLength = 0) then
            MaxLength := FDataLink.Field.Size;
          end;
          if FFocused and FDataLink.CanModify then
            Text := FDataLink.Field.Text
          else
            begin
              EditText := FDataLink.Field.DisplayText;
              if FDataLink.Editing and FDataLink.FModified then
                Modified := True;
              end;
            end else
              begin
                FAlignment := taLeftJustify;
                EditMask := '';
                if csDesigning in ComponentState then
                  EditText := Name else
                  EditText := '';
                end;
              end;
            end;
  end;
end;

```

데이터 인식 컨트롤의 내용을 편집했을 때에는 필드 데이터 링크 객체를 업데이트해야 데이터 세트에 변화를 반영할 수 있다. 데이터가 변했을 때에는 OnDataChange 이벤트에서 처리하듯이, 데이터 컨트롤이 변경 되었을 때에는 OnUpdateData 이벤트에서 처리해야 한다. 그러므로, UpdateData 메소드를 구현하고 OnUpdateData 이벤트에 UpdateData 메소

드를 연결한다. TDBEdit 컨트롤은 다음과 같이 구현되어 있다.

```
TDBEdit = class(TCustomMaskEdit)
private
    ...
    procedure UpdateData(Sender: TObject);

procedure TFieldDataLink.UpdateData;
begin
    if FModified then
    begin
        if (Field <> nil) and Assigned(FOnUpdateData) then FOnUpdateData(Self);
        FModified := False;
    end;
end;
```

DataChange, UpdateData 메소드에서 중요한 것은 이렇게 구현된 메소드를 constructor 에서 이벤트 핸들러를 대입하는 코드를 추가해야 한다는 것이다.

```
constructor TDBEdit.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    inherited ReadOnly := True;
    ControlStyle := ControlStyle + [csReplicatable];
    FDataLink := TFieldDataLink.Create;
    FDataLink.Control := Self;
    FDataLink.OnDataChange := DataChange;
    FDataLink.OnEditingChange := EditingChange;
    FDataLink.OnUpdateData := UpdateData;
end;
```

데이터 인식 컨트롤의 Change 메소드는 새로운 값이 설정될 때마다 호출된다. Change 메소드는 OnChage 이벤트 핸들러가 존재하면 이를 호출한다. 그러므로, 컴포넌트 사용자는 OnChage 이벤트 핸들러를 이용하여 데이터 변화에 대한 여러가지 처리를 하는 코드를 입력할 수 있다. 설정 값이 변경되면 연결된 데이터 세트는 반드시 변화에 대해 반응하도록 해야하는데, 이를 위해서는 Change 메소드를 오버라이드하여야 한다. DBTree 의 경우 다

음과 같이 구현한다.

```
TDBTree = class(TCustomTreeView)
...
protected
    procedure Change(Node: TTreeNode); override;
...

procedure TDBTree.Change(Node: TTreeNode);
begin
    inherited Change(Node);
    GotoID(Node);
end;

procedure TDBTree.GotoID(ANode: TTreeNode);
begin
    if ANode <> nil then
        TreeDataSet.Locate(FIDFieldName, PID(ANode.Data)^, []);
end;
```

여기서 Change 메소드는 사용자가 트리 노드를 선택하여 변경한 경우이므로 해당 트리 노드에 해당되는 레코드로 옮겨가야 한다. 이를 처리하는 메소드가 GotoID이며, GotoID 메소드는 DataSet의 Locate 메소드를 이용하여 구현한다.

이것으로 데이터 인식 컨트롤의 가장 핵심적인 부분에 대한 구현 방법에 대해서 소개하였다. 예제로 보여준 DBTree의 경우 일반적인 데이터 인식 컨트롤과는 달리 여러 개의 레코드를 보여주고, 이들 사이를 옮겨 다니는 처리를 해야 하기 때문에 다소 복잡하다. 그래서, 사실 가장 전형적인 예제로서는 부적절한 면이 있지만, 어떤 식으로 구현하는지에 대해서는 감을 잡았을 것으로 믿는다. 델파이 소스 중에서 DBCtrls.pas 유닛에 보면 좋은 예제가 많으므로 이들을 참고로 할 것을 권하고 싶다.

이번 장에서 같이 제공하는 DBTree 컴포넌트는 인터넷에서 구할 수 있는 여러가지 DBTreeView 컴포넌트와 조금은 다른 방식으로 간단하게 구현한 컴포넌트이다. 물론 인터넷에서 구한 컴포넌트보다 미약한 부분도 많지만, 사용하기에는 더 쉽고 간단하다.

컴포넌트를 제작할 때 아쉬운 점은 필자가 DBTree 컴포넌트를 처음 만들 때에는 인터넷에서 없었기 때문에 96년 가을에 사용을 위해서 제작한 것인데, 몇 달 지나지 않아서 비슷한 컴포넌트들이 공개되었다. 이처럼 기다리면 비슷한 것들이 나오는 것을 보면 누구나 생각하고 필요로 하는 내용은 비슷한가 보다 (^\_^). DBTree 컴포넌트의 소스와 함께 제공되는

Readme.txt 파일을 읽어 보면 DBTree 컴포넌트의 사용법에 대해서 적어 놓았으므로 활용해서 좋은 어플리케이션을 만들어보기 바란다.

## 컴포넌트 제작에 유용한 프로퍼티/메소드

지금까지 언급한 여러가지 컴포넌트 제작에 대한 내용 이외에도 조금은 고급스러운 컴포넌트로 만들기 위해서는 몇 가지 알아야 할 프로퍼티와 메소드가 있다. 여기서 이들에 대해 모두 다룰 수는 없지만 그래도 자주 쓰이는 것들을 중심으로 소개하고자 한다.

### ● 컴포넌트 State 의 검사

TComponent 컴포넌트의 ComponentState 프로퍼티는 컴포넌트의 현재 상태를 파악하는데 사용할 수 있다. 보통 컴포넌트를 셰어웨어로 배포할 때, 디자인 타임과 런타임을 구별해서 제한을 두는 컴포넌트 등을 제작할 때 사용된다.

ComponentState 프로퍼티 값에는 다음과 같은 것들이 있다.

값	설 명
csAncestor	컴포넌트가 조상 품에 나타난 경우 설정된다. csDesigning 이 같이 설정
csDesigning	컴포넌트가 디자인 모드에 있을 경우 설정된다.
csDestroying	컴포넌트가 파괴되려 할 때 설정된다.
csFixups	컴포넌트가 다른 품의 컴포넌트와 연결되어 있으나, 연결된 컴포넌트가 아직 로드되지 않은 경우 설정된다.
csLoading	Filer 객체에서 컴포넌트를 로딩하는 중이면 설정된다.
csReading	스트림에서 프로퍼티를 읽어들이는 도중이면 설정된다.
csUpdating	컴포넌트의 변경 사항을 업데이트하는 도중일 때. csAncestor 가 설정된 경우에만 사용된다.
csWriting	프로퍼티 값을 스트림에 기록하는 도중이면 설정된다.

예를 들어, 디자인 타임에서 등록하라는 메시지를 다음과 같이 보여줄 수 있다.

```
if (csDesigning in ComponentState) then ShowMessage('등록하세요 !');
```

### ● 컨트롤의 State 프로퍼티

TControl 클래스에는 런타임에서의 컨트롤의 상태를 반영하는 ControlState 라는 프로퍼티가 정의되어 있다. 이 프로퍼티는 세트로 정의되어 있으며, 다음과 같은 값을 가질 수 있

다.

값	의 미
csLButtonDown	왼쪽 마우스 버튼을 누르고 아직 놓지않은 상태
csClicked	클릭 이벤트가 발생할 때 설정된다.
csPalette	WM_PALETTECHANGED 메시지를 받았다.
csReadingState	컨트롤이 state 정보를 스트림에서 읽고 있다.
csAlignmentNeeded	컨트롤이 재정렬될 필요가 있을 때
csFocusing	어플리케이션이 컨트롤에 포커스를 주려고 한다.
csCreating	컨트롤이 생성되고 있다.
csPaintCopy	컨트롤이 복제되어 그려지고 있다.

### ● 컨트롤의 Style 설정

ControlStyle 프로퍼티는 컨트롤의 특징을 결정하는 세트 프로퍼티이다. 이 프로퍼티는 주로 컴포넌트의 constructor에서 설정하게 된다. 다음과 같은 값들과 의미를 가질 수 있다.

값	의 미
csAcceptsControls	디자인 타임에서 컨트롤을 드롭하면 parent가 될 수 있다.
csCaptureMouse	마우스를 클릭했을 때 이벤트를 캡처할 수 있다.
csDesignInteractive	디자인 타임에서 오른쪽 마우스 버튼을 클릭하면 왼쪽 버튼을 클릭하는 것으로 매핑하여 컨트롤을 다룬다.
csClickEvents	마우스의 클릭을 받아들인다.
csFramed	컨트롤이 3D 프레임을 가진다.
csSetCaption	Name 프로퍼티가 변경될 때, 자동으로 캡션이 변경된다.
csOpaque	컨트롤이 클라이언트 영역을 완전히 채운다.
csDoubleClicks	더블 클릭 메시지를 처리한다.
csFixedWidth	컨트롤의 폭이 변경되지 않는다.
csFixedHeight	컨트롤의 높이가 변경되지 않는다.
csNoDesignVisible	디자인 타임에 컨트롤이 보이지 않는다.
csReplicatable	컨트롤을 DBCtrlGrid에 드롭할 수 있다.
csNoStdEvents	마우스, 키보드, 클릭과 같은 표준 이벤트를 무시한다.

### ● Loaded 메소드

Loaded 메소드는 컨트롤이 생성될 때 프로퍼티를 DFM 파일에서 읽어온 뒤, 컴포넌트가 보여지기 전에 프로퍼티 값에 대한 생성 프로세스를 처리할 기회를 얻을 수 있다. 보통, 여기에서 저장된 프로퍼티 값에 대해 private, public 데이터 필드의 값을 초기화하거나 제작된 컴포넌트에 대한 여러가지 설정을 할 기회가 된다.

#### ● Notification 메소드

Notification 메소드는 TComponent 클래스에서 제공되는 가상 메소드로, 델파이 IDE 가 컴포넌트가 폼에 드롭되거나 제거될 때 호출하는 메소드이다. Notification 메소드는 다음과 같이 선언되어 있다.

```
procedure Notification(AComponent: TComponent; Operation: TOperation); virtual;
```

Notification 메소드의 첫번째 파라미터는 폼에 추가되거나 삭제되는 컴포넌트를 나타내며, 두번째 파라미터는 opInsert, opRemove 라는 값을 가질 수 있다. 이 메소드는 다른 컴포넌트가 폼에 추가되거나 삭제될 때 특정 작업을 해야할 때 사용된다. 예를 들어, TBatchMove 의 Notification 메소드는 다음과 같이 구현되어 있다.

```
procedure TBatchMove.Notification(AComponent: TComponent;  
    Operation: TOperation);  
begin  
    inherited Notification(AComponent, Operation);  
    if Operation = opRemove then  
    begin  
        if Destination = AComponent then Destination := nil;  
        if Source = AComponent then Source := nil;  
    end;  
end;
```

먼저 상속된 Notification 메소드를 호출하고, 작업이 opRemove 인 경우에 제거되려는 컴포넌트의 종류가 Source, Destination 프로퍼티에 지정된 컴포넌트이면 이들 프로퍼티의 값을 nil 로 설정한다. 이렇게, 다른 컴포넌트와의 관련이 있도록 만들어진 경우에는 컴포넌트의 추가와 삭제에 따른 처리를 해주어야 한다. DBTree 컴포넌트에도 Notification 메소드를 다음과 같이 구현하고 있다.

```
procedure TDBTree.Notification(AComponent: TComponent;
```



```

    Operation: TOperation);
begin
    inherited Notification(AComponent, Operation);
    if (Operation = opRemove) and (FDataTitleLink <> nil) and
        (AComponent = DataSource) then DataSource := nil;
end;

```

#### ● FreeNotification 메소드

FreeNotification 메소드는 Notification 메소드와 관련되어 사용되는 TComponent 의 메소드이다. Notification 메소드가 폼에서 컴포넌트가 추가되고 삭제될 때 호출되지만, Owner 가 다른 컴포넌트의 추가, 삭제에는 반응하지 못한다. 즉, 데이터 인식 컨트롤의 경우 데이터 모듈에 데이터 소스가 있고 폼에 컨트롤이 있는 경우와 같이 Owner 가 다른 경우에는 데이터 소스가 추가, 삭제될 때 Notification 메소드를 호출하지 못한다.

이런 경우에는 연결된 컴포넌트를 FreeNotification 메소드를 이용해서 컴포넌트가 삭제될 때 Notification 메소드를 호출하도록 지정할 수 있다. 사소한 듯하지만 데이터 인식 컨트롤 등과 같이 연결된 컴포넌트가 있어야 하는 컴포넌트를 제작할 때에는 놓치기 쉬운 부분이다. TDBTree 컨트롤의 데이터 소스 컨트롤을 지정하는 SetDataSource 메소드의 구현 부분을 살펴 보자.

```

procedure TDBTree.SetDataSource(Value: TDataSource);
begin
    FDataTitleLink.DataSource := Value;
    if Value <> nil then Value.FreeNotification(Self);
end;

```

이렇게 데이터 소스를 연결할 때 파라미터로 Notification 을 받을 컴포넌트를 지정하면 된다. 여기서 Value 는 데이터 소스를 가리키며, 데이터 소스의 FreeNotification 메소드에 Self(여기서는 TDBTree)를 파라미터로 하여 호출함으로써 데이터 소스가 추가, 삭제될 때 TDBTree 의 Notification 메소드를 호출하게 된다.

#### ● CreateWnd, DestroyWnd 메소드

CreateWnd 메소드는 윈도우 컨트롤이 처음 생성되거나, 윈도우가 프로퍼티의 변경에 따라 파괴되었다가 재생성 되어야 할 필요가 있을 때 호출된다. 그러므로, CreateWnd 메소드는 윈도우가 처음 생성될 때 추가적인 초기화 메시지를 넘겨주기 위해 오버라이드하여 구현한

다. CreateWnd 메소드는 처음에 CreateParams 메소드를 호출하여 윈도우 생성 파라미터를 초기화하고, CreateWindowHandle 을 호출하여 컨트롤에 대한 윈도우 핸들을 생성한다. 그리고, 새로운 크기의 윈도우를 적용하게 되며 WM\_SETFONT 메시지를 이용해 Perform 메소드를 호출하여 컨트롤의 폰트를 설정한다.

DestroyWnd 메소드는 TWinControl 에서 추가한 메소드로, 컴포넌트가 파괴될 때 컨트롤의 윈도우 핸들과 연관된 디바이스 컨텍스트를 해제하기 위한 메소드이다. 이 메소드는 컴포넌트의 Destroy 메소드 이전에 호출된다. 만약 Destroy 메소드에서 윈도우 핸들을 이용한 작업을 하려고 하면, 'Window has no parent' 에러 메시지를 보게 된다. 이는 윈도우 핸들이 없어졌기 때문이다. 그러므로, 컴포넌트가 파괴될 때 윈도우 핸들을 가지고 디바이스 컨텍스트를 이용한 여러가지 작업을 하고자 할 때에는 DestroyWnd 메소드를 오버라이드하여 구현해 주어야 한다. 여기서 꼭 생각해야 할 것은 상속된 DestroyWnd 메소드를 호출하면 핸들이 없어지므로, 상속된 DestroyWnd 메소드는 가장 나중에 호출하는 것이 좋다. VCL 소스 코드 중에서 비교적 간단한 TCustomEdit 의 예를 들어 설명 하겠다.

```
procedure TCustomEdit.CreateWnd;
begin
    FCreating := True;
    try
        inherited CreateWnd;
    finally
        FCreating := False;
    end;
    DoSetMaxLength(FMaxLength);
    Modified := FModified;
    if FPasswordChar <> #0 then
        SendMessage(Handle, EM_SETPASSWORDCHAR, Ord(FPasswordChar), 0);
    UpdateHeight;
end;
```

먼저 상속된 CreateWnd 메소드를 호출하여 초기 작업을 한다. 그리고, FMaxLength 필드의 값을 이용하여 길이를 결정하고 Modified 프로퍼티를 설정한다. 또한, FPasswordChar의 값이 존재하면 패스워드 문자를 보여주기 위해 메시지를 넘겨주고 컨트롤의 높이를 맞추게 된다. 이와 같이 CreateWnd 메소드에서는 윈도우가 생성될 때 고려해야 할 여러가지를 설정한다.

```
procedure TCustomEdit.DestroyWnd;
```

```
begin
```

```
    FModified := Modified;
```

```
    inherited DestroyWnd;
```

```
end;
```

```
function TCustomEdit.GetModified: Boolean;
```

```
begin
```

```
    Result := FModified;
```

```
    if HandleAllocated then Result := SendMessage(Handle, EM_GETMODIFY, 0, 0) <> 0;
```

```
end;
```

여기서는 FModified 필드의 값을 Modified 프로퍼티의 값과 일치시킨다. 자동적으로 접근 메소드인 GetModified 가 호출되는데, 여기에서 Handle 을 이용하여 변경된 내용을 반영하게 된다.

## 정 리 (Summary)

이것으로 컴포넌트 제작에 대한 여러가지 방법 들에 대한 설명을 마치고자 한다. 이번 장에서 제작한 달력과 DBTree 컴포넌트는 Chap23.dpk 패키지 파일에 저장하였으니 직접 설치하고 익혀보기 바란다. 컴포넌트 제작은 델파이로 프로그래밍을 할 때 가장 필수적인 부분이고 중요한 부분이라고 할 수 있다. 나름대로 여러가지 부분을 다루었으나, 컨테이너 컴포넌트를 제작하는 방법이나 프로퍼티 에디터를 제작하는 방법 등의 고급 컴포넌트 제작에 대한 내용을 지면 관계상 모두 다루지 못한 아쉬움이 남는다. 아마도 컴포넌트를 제대로 제작하는 내용을 모두 담으려고 하면 적어도 그것만 가지고 1000 페이지는 넘는 책을 써야 할 것이다. 델파이의 소스를 열심히 분석해보면 얻는 것이 많을 것이라는 정도로 밖에 여기서는 말할 수 없는 것이 무척 아쉽다.

제 4 부에서 다룬 내용은 델파이에서 사용하게 되는 델파이 컴포넌트에 대한 개발 방법에 대해서 알아 보았다. 다음 장에서부터 다루게 되는 제 5 부의 내용은 델파이에 한정되지 않고, 다른 개발 도구에서도 사용할 수 있는 표준적인 개발 방법론인 DLL/DCOM/CORBA에 대해서 알아보도록 할 것이다.

# 델파이 4에서의 DLL 프로그래밍

## (DLL Programming in Delphi 4)

윈도우에서 가장 표준적으로 다루게 되는 파일 형식 중에서 동적으로 사용하는 표준적인 실행 파일 형식이 바로 DLL 파일이다. 이번 장에서는 델파이 4를 이용하여 DLL 파일을 이용하고, 제작하는 방법에 대해서 알아보도록 한다.

### DLL에 대한 기초적인 이해

동적 링킹(dynamic linking)이란 정적 링킹(static linking)과 상대되는 개념으로 실행 가능한 모듈 런타임에서 필요한 정보만을 포함할 수 있도록 하는 것이다. 이에 비해서 정적 링킹이란 라이브러리 함수의 실행 코드가 각각의 어플리케이션에 담겨져 있어야 한다.

DLL은 프로세스가 시작되거나 프로세스의 쓰레드가 LoadLibrary 함수를 호출할 때 프로세스에 연결된다. 일단 DLL이 프로세스에 연결되면 운영체제는 DLL 모듈을 프로세스의 주소 공간에 매핑을 하게 되고, 이 과정을 통해 프로세스가 DLL의 실행 코드에 접근할 수 있다. 반대로 프로세스가 종료되거나 FreeLibrary 함수가 호출되면 DLL 모듈은 프로세스의 주소 공간의 매핑에서 벗어난다.

다른 함수들과 마찬가지로, DLL 함수 역시 쓰레드의 공간에서 실행된다. 그렇기 때문에 다음과 같은 조건을 항상 염두에 두어야 한다.

1. DLL을 호출한 프로세스의 쓰레드는 DLL 함수에 의해 열린 핸들을 사용할 수 있다. 마찬가지로 호출하는 프로세스의 쓰레드가 연 핸들은 DLL 함수가 사용할 수 있다.
2. DLL은 호출 쓰레드의 스택과 호출 프로세스의 주소 공간을 사용한다.
3. DLL에 의해 할당된 메모리는 호출 프로세스의 주소 공간에 있다.
4. DLL을 호출한 프로세스의 쓰레드들은 DLL에서 전역으로 선언된 변수 값을 읽고 쓸 수 있다.

#### ● 동적 링킹의 장단점

동적 링킹은 정적 링킹에 비해 몇 가지 장점을 가지고 있다. 이를 나열하면 다음과 같다.

1. 많은 수의 프로세스들이 하나의 DLL을 동시에 사용할 수 있다. 이렇게 하면 메모리 공간을 절약할 수 있다.
2. DLL에 있는 함수가 변하면 이를 사용하는 어플리케이션은 함수의 형태와 리턴 값이

바뀌지 않는 한에는 재컴파일이나 링크가 필요 없이 기능이 업그레이드될 수 있다.

3. DLL 로 쪼개 놓으면 나중에 업그레이드가 용이하다. 즉, 기능이 향상된 부분의 DLL 만 통신 등으로 보내면 해결이 될 수 있는 경우가 많다.
4. 서로 다른 프로그래밍 언어로 만들어진 프로그램 사이에서 함수의 호출 규칙(calling convention)만 동일하다면 함수의 공유와 통신이 가능하다.

DLL 을 사용할 때 단점 역시 존재한다. 쪼개져 있기 때문에 좋은 점도 있지만, 이들 중 하나만 없어도, 또는 하나만 비정상적으로 동작해도 전체 어플리케이션이 동작하지 않는 불상사를 맞을 수 있다.

#### ● DLL 진입점(Entry-Point) 함수

DllEntryPoint 함수는 운영체제가 호출하는 함수로, 반드시 있을 필요는 없다. DllEntruPoint 는 전형적인 함수가 아니라 링커 커맨드 라인에 의해 지정되는 참조값(reference)이다. 그러므로, DLL 에 항상 진입점 함수가 있을 필요는 없다. 하지만, 쓰레드와 프로세스를 초기화하는 작업 등을 할 때 유용하게 사용할 수 있다.

일단 진입점 함수가 지정되면 운영체제는 프로세스에 DLL 이 연결되거나 연결이 끊길 때, 또는 쓰레드가 DLL 에 연결되거나 연결이 끊길 때 이 함수를 호출한다.

이렇게 호출되면 DLL 은 프로세스의 주소 공간에 메모리를 할당하거나 프로세스가 접근할 수 있는 핸들을 열 수가 있다. 또한, 새로운 쓰레드와 연결될 때에는 그 쓰레드에 대한 지역 저장소(thread local storage, TLS)를 할당할 수 있다.

#### ● 런타임 동적 링킹 (Run-Time Dynamic Linking)

런타임 동적 링킹은 프로세스가 DLL 의 이름을 지정해서 LoadLibrary 함수를 호출하고, DLL 의 함수에 대한 시작 주소를 얻어오는 GetProcAddress 함수를 호출함으로써 이루어진다. LoadLibrary 를 호출할 때 DLL 모듈이 이미 호출 프로세스의 주소 공간에 매핑되어 있는 경우에는 DLL 에 대한 핸들을 반환하고, 모듈의 참조 계수(reference count)를 하나 증가시킨다.

시스템이 DLL 을 찾지 못하거나 진입점 함수가 False 를 반환하면 LoadLibrary 는 NULL 값을 리턴하며, 성공적으로 수행된 경우에는 DLL 모듈의 핸들을 돌려 준다. 프로세스는 이 핸들을 이용해서 GetProcAddress, FreeLibrary 등의 함수를 호출할 수 있게 된다.

GetModuleHandle 함수도 GetProcAddress 나 FreeLibrary 함수가 사용할 수 있는 핸들을 반환하지만, DLL 모듈이 이미 프로세스의 주소 공간에 매핑이 된 후에야 동작하며, 모듈의 참조 계수를 증가시키지 않는다.

LoadLibrary 나 GetModuleHandle 함수에 의해 DLL 의 모듈 핸들을 얻었으면 프로세스는

GetProcAddress 함수를 이용해서 함수의 시작점을 알아낼 수 있다.

더 이상 DLL 모듈이 필요하지 않으면 프로세스는 FreeLibrary 함수를 호출하여 모듈의 참조 계수를 하나 감소시키고, 만약 참조 계수가 0 이 되면 주소 공간의 매핑을 해제한다.

## ● DLL 데이터

윈도우는 다음의 DLL 데이터를 지원한다.

1. DLL 을 사용하는 각각의 프로세스 전용의 전역 또는 정적 변수
2. DLL 을 사용하는 모든 프로세스가 공유하는 전역 또는 정적 변수
3. DLL 을 사용하는 각각의 프로세스 전용의 동적 할당 메모리
4. 여러 프로세스에서 공유할 수 있는 동적 할당 메모리
5. 멀티 쓰레드 프로세스의 각각의 쓰레드 전용의 정적 저장소(static storage)

DLL 변수의 디폴트 범위는 1 의 경우로 각각의 프로세스에 대해 전역, 정적 변수로 설정되어 있다.

DLL 이 메모리 할당 함수(GetMem, New, GlobalAlloc, LocalAlloc, HeapAlloc, VirtualAlloc)를 써서 메모리를 할당할 경우, 메모리는 DLL 을 호출한 프로세스의 주소 공간에 할당되므로 그 프로세스의 쓰레드만 접근할 수 있다. 그러므로, DLL 에서 데이터를 공유하기 위해서는 파일 매핑을 사용해서 메모리를 할당해야 한다.

## ● Win16 과 Win32 DLL

Win32 환경에서의 근본적인 Win16 과의 차이점은 세그먼트로 나뉘어진 메모리 모델을 플랫폼 메모리 프로세스로 변경한다는 것이다. 16 비트 윈도우에서는 DLL 이 운영체제와 통합된 전역으로 접근 가능한 라이브러리로 취급되기 때문에, 여러 어플리케이션에서 공유할 수 있다.

16 비트 DLL 은 어플리케이션을 적재할 때 스택을 사용하기 때문에, 메모리의 64K 데이터 세그먼트 제한을 가질 수 밖에 없다. 이들 DLL 을 사용할 때에는 LoadLibrary 함수를 사용하는데, LoadLibrary 함수가 호출되면 윈도우가 관리하는 참조 계수가 하나 증가한다. 어플리케이션이 DLL 을 사용하고 나면, FreeLibrary 함수를 호출하게 되는데 이 함수에 의해 참조 계수가 하나 감소하며 이 값이 0 이되면 DLL 이 메모리에서 해제된다.

16 비트 DLL 은 자신의 데이터 세그먼트를 따로 가지고 있기 때문에, 전역 변수와 상수 등이 어플리케이션 사이에서 서로 통신을 할 수 있는 수단으로 사용이 가능하다. 어플리케이션은 DLL 에 메모리 블록을 글로벌 힙에 할당하고, 포인터를 다른 어플리케이션에서 사용하도록 넘기는 등의 작업을 할 수 있다.

Win32 모델에서는 어플리케이션이 자신의 메모리 공간에 분리되어 존재한다. 운영체제가 어플리케이션을 적재하게 되면, 프로세스(process)라고 하는 커널 프로세스 객체(kernel process object)를 생성한다. 하나의 프로세스는 최대 4GB 까지의 크기를 가질 수 있으며, 시스템 페이징 파일의 메모리 지역을 주소 관리를 위해 할당된다. 프로세스가 생성되면, 커널 파일 매핑 객체(kernel file-mapping object)가 생성되어 실행 파일을 프로세스의 주소 공간에 매핑하게 된다. 이로 인해 실행 파일은 하드 디스크의 특정 위치에 자리잡게 되므로 전체 이미지를 RAM 으로 적재할 필요가 없다.

실행 파일이 메모리에 매핑되면, 운영체제가 EXE 파일에 의해 요구되는 모든 DLL 의 이름들을 가져오기 위해 이미지의 오프셋으로 이동한 뒤에 요구되는 DLL 을 찾아서 EXE 파일과 같은 프로세스로 매핑한다. 커널 파일 매핑 객체는 각각의 DLL 에 대해서 생성된다. 어플리케이션의 실행이 종료되면 각각의 DLL 의 매핑이 해제되고 파일 매핑 객체가 파괴된다. 이런 프로세스는 EXE 파일의 매핑이 해제되고 커널 프로세스 객체가 파괴될 때까지 지속된다. Win32 환경에서 DLL 은 이와 같이 각각의 어플리케이션에 의해 독자적으로 사용된다. 다시 말해 여러 인스턴스가 생성되어 각각의 DLL 이 독자적인 주소 공간을 가지게 되는 것이다.

## DLL 호출하기

DLL 에 정의되어 있는 루틴을 호출하기 전에, 이들을 반드시 import 해야 한다. 이렇게 DLL 의 함수를 import 하는 방법에는 2 가지가 있다. 첫번째는 외부 프로시저나 함수를 선언하는 것이고, 다른 하나는 윈도우 API 함수를 직접 호출하는 것이다.

어떤 방법을 사용하든 런타임이 될 때까지는 어플리케이션에 링크되지 않는다. 이는 DLL 이 프로그램을 컴파일할 때에는 필요없다는 것이며, 루틴을 import 할 때 컴파일 시에는 특별한 타당성 검사를 하지 않는다는 것을 의미한다.

### ● 정적 로딩 (Static loading)

DLL 함수를 import 하여 사용하는 가장 간단한 방법은 외부 지시어를 이용하여 선언해 사용한 것이다. 예를 들어, 다음과 같이 하면 된다.

```
procedure DoSomething; external 'MYLIB.DLL';
```

이런 선언부를 프로그램에 위치시키면, MYLIB.DLL 이 프로그램이 시작될 때 메모리에 적재된다. 프로그램이 실행되는 동안 DoSomething 을 호출하는 것은 이는 언제나 DLL 의 진입점(entry point)를 호출하는 것을 의미한다.

이런 선언부는 프로그램에 직접 위치시킬 수도 있고, 따로 유닛을 추가해서 선언한 뒤에 이

유닛을 use 절에 추가하여 사용할 수도 있다. 관리의 편리성이라는 측면에서 이렇게 따로 유닛을 선언해서 사용하는 것이 좋은데, 델파이의 Windows.pas 유닛도 이런 대표적인 DLL 의 API 함수를 선언한 유닛이다.

- 동적 로딩 (Dynamic loading)

DLL 에 직접 접근하여 사용하려면 윈도우의 API 함수를 호출해서 사용할 수 있다. 대표적인 함수가 LoadLibrary, FreeLibrary, GetProcAddress 등이다. 이들은 Windows.pas 유닛에 선언되어 있으므로 uses 절에 Windows.pas 유닛을 추가하고 다음과 같은 식으로 사용하면 된다.

```
uses Windows, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ...

begin
  Handle := LoadLibrary('DATETIME.DLL');
  if Handle <> 0 then
    begin
      @GetTime := GetProcAddress(Handle, 'GetTime');
      if @GetTime <> nil then
        begin
          GetTime(Time);
          with Time do
            WriteLn('The time is ', Hour, ':', Minute, ':', Second);
```



```

    end;
    FreeLibrary(Handle);
end;
end;

```

이런 식으로 루틴을 import 하면 DLL 은 LoadLibrary 함수가 실행되기 전까지는 DLL 이 메모리에 적재되지 않는다. 그리고 LoadLibrary 에 의해 적재된 DLL 은 사용이 끝난 뒤 FreeLibrary 를 호출해야 메모리에서 해제된다. 그러므로, 꼭 필요할 때에만 DLL 을 메모리에 올려서 사용하게 되므로 메모리를 절약할 수 있는 장점이 있다.

## DLL 제작하기

델파이를 이용해서 쉽게 DLL 을 만들 수 있다. 새로운 DLL 프로젝트는 하나의 DPR 파일로 이루어지며, 디폴트 폼이나 추가적인 유닛을 필요로 하지 않는다. DLL 을 만들려면 먼저 File|New 메뉴를 선택하여 객체 저장소를 띄우고 DLL 을 더블 클릭하면 껍데기 코드가 만들어진다.

```
library Exam1;
```

```
...
```

```
uses
```

```
    SysUtils,
```

```
    Classes;
```

```
begin
```

```
end.
```

DLL 의 구조는 기본적으로 다른 프로그램과 다를 바가 없다. 다만 DLL 프로젝트는 library 키워드로 시작된다. 다음 코드는 Min, Max 라는 2 개의 export 함수를 가지는 DLL 을 제작한 것이다.

```
library Exam1;
```

```
function Min(X, Y: Integer): Integer; stdcall;
```

```
begin
```

```

    if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
    if X > Y then Max := X else Max := Y;
end;

exports
    Min index 1,
    Max index 2;

begin

end.

```

다른 언어로 제작하는 어플리케이션에 DLL 을 사용할 수 있게 하려면 이와 같이 stdcall 호출 규칙을 사용하는 것이 좋다. 여기에 대해서는 다시 다루게 될 것이다. 이렇게 만든 DLL 을 사용할 때에는 앞서도 설명했듯이 사용할 어플리케이션의 type 선언부나 새로운 유닛을 추가한 뒤에 Min, Max 를 다음과 같이 선언하고 간단히 이 함수를 호출하면 된다.

```

function Min(X, Y: Integer): Integer; stdcall; external 'Exam1.DLL';
function Max(X, Y: Integer): Integer; stdcall; external 'Exam1.DLL';

```

```
ShowMessage(IntToStr(Min(1, 2)));    //결과는 '1'을 메시지 박스로 보여 준다.
```

이때 주의할 점은 Exma1.DLL 이 윈도우, 시스템이나 어플리케이션의 디렉토리와 같이 검색하는 디렉토리 안에 위치해야 한다는 점이다. 아니면 정확한 패스를 external 뒤에 적어 주어야 한다. 또한, 앞서 설명한 것과 같이 LoadLibrary 함수를 이용하여 동적으로 루틴을 호출할 수도 있다.

DLL 은 여러 개의 유닛으로 구성할 수도 있다. 이런 경우에는 라이브러리의 소스 파일은 uses 절에 실제로 루틴을 구현한 유닛을 적어주고, 여기에는 단순히 export 문장과 DLL 초기화 코드만 적어주면 된다. 다음 코드를 살펴 보자.

```

library Editors;
uses EdInit, EdInOut, EdFormat, EdPrint;

```

exports

InitEditors index 1,  
DoneEditors index 2,  
InsertText index 3,  
DeleteSelection index 4,  
FormatSelection index 5,  
PrintSelection index 6,  
...  
SetErrorHandler index 53;

begin

end.

이와 같이 DLL 프로젝트 파일은 명시적으로 루틴을 export 하는 부분 만을 선언하고, 나머지 구현 부분은 uses 절에 추가된 유닛에 위치한다.

#### ● exports 절

루틴을 exports 하는 방법은 다음 형식을 가지고 선언하면 된다.

exports entry1, ..., entryn;

각각의 entry 는 exports 절 이전에 선언된 프로시저나 함수의 이름과 옵션으로 인덱스와 옵션 이름을 추가할 수 있다. 인덱스는 1~2,147,483,647 까지의 값을 가질 수 있다. 그렇지만 효율적인 프로그램이 되려면 낮은 수를 사용하는 것이 좋다. Entry 의 인덱스는 옵션이기 때문에, 이것이 없는 경우에는 DLL export 테이블에 자동으로 인덱스를 부여하게 된다. 또한, 이름을 지정하지 않은 경우에는 유닛에서 선언한 이름을 그대로 사용하게 된다. 그러므로, name 지시어는 다음과 같이 루틴의 원래 이름과 다른 이름으로 export 할 때 사용하게 된다.

exports

DoSomethingABC index 1 name 'DoSomething';

#### ● DLL 초기화 코드

델파이 프로젝트에서 DLL 의 진입점은 처음의 begin ... end. 사이에 위치한다. 로더 코드가 실행될 때 여기에 위치한 모든 문장이 가장 먼저 실행된다. 그러므로, 여기에는 초기화 루틴과 메모리 처리 루틴 등을 위치시키기 좋다. DllEntryPoint 는 DllMain 으로 불리기도 하는데, 옵션으로 제공되며 이 함수가 export 되는 경우에는 프로세스나 스레드가 DLL 을 적재하거나 해제할 때 호출된다. DllEntryPoint 는 C++ 에서 다음과 같이 선언되어 있다.

```
BOOL WINAPI DllEntryPoint(HINSTANCE hInst, DWORD dwReason, LPVOID lpvReserved);
```

델파이는 C++ 에서의 DllMain 처럼 동작하는 DllMain 루틴을 지원하는데, 이를 위해서 다음과 같은 추가적인 코딩이 필요하다.

```
library Exam1;
```

```
uses
```

```
    ShareMem, Windows, SysUtils, Classes;
```

```
procedure DllMain(dwReason: DWORD);
```

```
begin
```

```
    case dwReason of
```

```
        dll_Process_Attach: ;
```

```
        dll_Process_Detach: ;
```

```
        dll_Thread_Attach: ;
```

```
        dll_Thread_Detach: ;
```

```
    end;
```

```
end;
```

```
begin
```

```
    DLLProc := @DllMain;
```

```
    DllMain(dll_Process_Attach);
```

```
end.
```

uses 절의 처음에 ShareMem 유닛을 추가한 것에 주의하라. 델파이에서 사용하는 긴 문자열을 제대로 지원하기 위해서는 이와 같이 DLL 의 첫번째 uses 절에 ShareMem.pas 유닛을 추가해야 하고, 배포할 때 BORLANDMM.DLL 파일을 같이 배포해야 한다. 물론 PChar, ShortString 데이터 형만 사용하는 경우에는 필요가 없다.

uses 절에 Windows.pas 유닛을 추가한 이유는 dll\_ 상수에 대한 선언부가 있기 때문으로 이들의 의미는 다음과 같다.

상 수	값	의 미	용 도
dll_Process_Attach	1	프로세스가 DLL 에 매핑된다. 프로세스당 한번만 호출된다.	전역 객체나 변수를 초기화 하거나, 배치 프로세스를 처리한다.
dll_Thread_Attach	2	프로세스가 자식 쓰레드를 생성 할 때 호출된다.	쓰레드에 대한 코드를 처리할 때 사용 된다.
dll_Thread_Detach	3	쓰레드가 종료될 때 호출된다.	DLL 이 쓰레드 당 자원을 처리할 때 이를 해제하는 역할
dll_Process_Detach	0	프로세스에서 DLL 을 해제할 때 호출된다.	전역 객체나 변수, 파일, 포트 등을 해 제하고 닫는다.

DLLMain 프로시저는 여러 차례 호출될 수 있으나, 이들이 호출되는 원인은 이렇게 4 가지이다. 프로세스는 DLL 을 여러 차례 적재할 수 있는데, 이들은 다중 쓰레드로 실행된다. 그런데 이때 DLL 이 주소에 매핑되는 것은 첫번째 호출에 의해서 결정되며, 그 이후의 호출은 참조 계수를 증가시키게 된다.

델파이는 암시적으로 시스템 유닛에 정의된 모든 기능을 포함하고 있다. 시스템 유닛은 \_StartLib 라는 어셈블리 언어 함수를 정의하고 있다. 이 루틴은 DLLProc 라는 시스템 레벨 변수에 대입된 모든 코드를 실행할 책임이 있다. DLLProc 변수는 일종의 포인터 변수로 디폴트 값은 nil 이다. 여기에 DLLMain 함수의 주소를 대입하면 DLLMain 함수가 자동으로 호출된다. 또한, 델파이 프로젝트 파일의 begin ... end. 사이의 부분은 초기화를 담당하는 부분으로 프로세스가 처음 DLL 을 불러들일 때 한번만 호출되기 때문에 dll\_Process\_Attahc 와 같은 역할을 하게 된다.

DLLMain 함수는 DLL 프로젝트에 꼭 사용해야 하는 것은 아니고, 옵션으로 이용하는 것이다. 그렇지만, 전역 변수를 초기화하고 사용할 객체를 인스턴스화 하는 등의 초기 작업을 하는데 유용하게 사용할 수 있다. 다음과 같이 어플리케이션의 전역 예외 처리를 위한 객체를 생성하여 사용하는 것이 대표적인 예가 된다.

```
library Exam2;
```

```
uses
```

```
    ShareMem, Windows, SysUtils, Classes, Forms;
```

```
type
```

```
    TErrorHandler = class
```

```

    procedure ErrorHandle(Sender: TObject; E: Exception);
end;

procedure TErrorHandler.ErrorHandle(Sender: TObject; E: Exception);
begin
    //전역 에러 처리 루틴
end;

var
    ErrorHandler: TErrorHandler;

procedure DLLMain(dwReason: DWORD);
begin
    case dwReason of
        dll_Process_Attach:
            begin
                ErrorHandler := TErrorHandler.Create;
                Application.OnException := ErrorHandler.ErrorHandle;
            end;
        dll_Process_Detach:
            begin
                Application.OnException := nil;
                ErrorHandler.Free;
            end;
        dll_Thread_Attach: ;
        dll_Thread_Detach: ;
    end;
end;

begin
    DLLProc := @DLLMain;
    DLLMain(dll_Process_Attach);
end.

```

먼저 Application 객체를 사용하기 위해 Forms.pas 유닛을 uses 절에 추가한다. 그리고, Application 객체의 OnException 이벤트 핸들러에 TErrorHandler 클래스의 ErrorHandle

프로시저로 대입한다. 여기에서 ErrorHandler 프로시저에서 에러에 대해 아무런 작업을 하지 않았지만, 에러 메시지를 보여주는 등의 작업을 하도록 정의할 수 있다.

이 밖에도 라이브러리 초기화 코드로 ExitProc 변수를 이용하여 DLL 이 메모리에서 해제될 때 실행되는 Exit 프로시저를 설치할 수도 있다. 다음의 코드를 살펴 보자.

library Test;

var

SaveExit: Pointer;

procedure LibExit;

begin

...

ExitProc := SaveExit; //원래의 내용을 복구 한다.

end;

begin

...

SaveExit := ExitProc; //복구를 위해 위치를 저장한다.

ExitProc := @LibExit; //LibExit 프로시저를 ExitProc 로 저장한다.

end.

이 코드에 의해 DLL 이 메모리에서 해제될 때 라이브러리의 exit 프로시저가 ExitProc 변수가 nil 이 될 때까지 ExitProc 에 저장된 주소를 반복적으로 호출하게 된다.

#### ● 호출 규칙 (Calling Conventions)

호출 규칙은 메소드나 함수에 변수를 넘기는 프로토콜을 정의하는 것이다. 호출 규칙은 함수에 넘겨 줄 파라미터의 순서와, 파라미터를 넘길 때 CPU 레지스터 사용 여부 등을 지정한다. 또한, 스택의 내용을 정리하는 책임을 호출한(caller) 함수가 가지는 지 아니면 호출된(callee) 함수가 가지는 지 결정한다.

델파이에서 사용되는 호출 규칙에는 다음과 같은 것들이 있다.

호출 규칙	파라미터 순서	스택 청소 책임
Fastcall(Register)	좌측에서 우측으로	호출된 루틴
Stdcall	우측에서 좌측으로	호출된 루틴

Pascal	좌측에서 우측으로	호출된 루틴
Cdecl	우측에서 좌측으로	호출하는 루틴
Safecall	우측에서 좌측으로	호출된 루틴

디폴트 호출 규칙은 Fastcall 이다. 가장 효과적인 프로토콜이며, CPU 레지스터를 이용하여 처음 3 개의 파라미터를 넘긴다. 그렇기 때문에, 수행속도가 빠르다. Pascal 호출 규칙은 델파이 1.0 에서 사용하던 호출 규칙으로 윈도우 3.1 의 표준 호출 규칙이다. Win32 에서는 Stdcall 로 표준 호출 규칙을 변경하였다.

그러므로, 작성한 DLL 을 델파이에서 사용할 것이라면 지정한 호출 규칙만 알면 어떤 것을 사용해도 상관이 없다. 그러나, C++ 을 비롯한 다른 언어와의 호환성을 위해서는 Win32 의 표준 호출 규칙인 Stdcall 을 사용하는 것이 좋다.

#### ● DLL 의 예외와 런타임 에러

예외가 발생하지만 DLL 에서 처리하지 못한 경우에는 예외가 DLL 을 호출한 어플리케이션으로 전달된다. 그러므로, DLL 에서 발생한 에러라 할 지라도 이를 호출한 어플리케이션 코드에서 try ... except 문을 사용하여 처리가 가능하다. 만약, DLL 이 오브젝트 파스칼이 아닌 다른 언어에서 사용될 때에는 예외 코드 \$OEEDFACE 를 처리하면 된다. 운영체제의 예외 레코드의 배열인 ExceptionInformation 의 첫번째 entry 에는 예외가 발생한 주소가 담고 있으며, 두번째 entry 에는 오브젝트 파스칼 예외 객체의 레퍼런스를 저장하고 있다.

DLL 이 SysUtils.pas 유닛을 사용하지 않는 경우에는 델파이가 예외 처리를 제대로 하지 못한다. 이런 경우에는 런타임 에러가 DLL 에서 발생한 경우, 이를 호출한 어플리케이션이 중단된다. DLL 은 오브젝트 파스칼 프로그램에서 호출될 지를 알 수 있는 방법이 없기 때문에, 어플리케이션의 exit 프로시저를 호출할 수도 없다. 그러므로, 어플리케이션의 동작이 중지되면서 메모리에서 삭제된다.

## 정 리 (Summary)

이번 장에서는 DLL 에 대한 정보를 알 때, 이를 선언하여 델파이에서 사용하는 방법과 간단한 DLL 파일의 제작 방법에 대해서 알아 보았다.

물론 DLL 을 이용하여 리소스를 저장하거나, 폼을 DLL 로 제작하는 등의 다소 고급스러운 내용에 대해서는 다루지 않았지만 여기에 대해서는 비교적 많은 책들에서 다루고 있고, 또한 Inprise 의 TI, FAQ 등에서도 자주 나오는 내용이기 때문에 생략하였다



# COM 의 기초 개념

## (Basic Concepts of Component Object Model)

COM (Component Object Model)은 OLE 와 액티브 X 기술의 기초가 되는 개념으로 인터페이스라는 미리 정의된 루틴의 세트를 통해 각 객체들간의 상호운용을 가능하게 해주는 객체 기반의 프로그래밍 specification 이다. COM 은 기본적으로 소스 코드 수준의 표준이 아니라 바이너리 표준이다. 이 때문에 여러가지 다른 언어로 객체를 구현할 수 있으며, 서로 다른 플랫폼과 다른 주소 공간에서 실행과 통신이 가능하다. 또한, COM 객체들은 유일한 identifier 가 있어서 생성과 인터페이스로의 접근이 쉬우므로 확장, 수정, 업데이트가 용이하다. COM 에는 COM 객체의 핵심적인 기능을 정의하고, COM 객체들의 생성과 관리를 가능하게 하는 API 함수의 세트를 정의하고 있는 표준 인터페이스들의 세트를 포함한 라이브러리를 가지고 있다.

또한, COM 은 OLE 와 액티브 X 의 기초가 되는데, 이들 기술들은 운영체제의 확장부분처럼 동작하여 이들 type 의 객체들을 생성하고 조작할 수 있는 라이브러리들을 제공한다. COM 을 기반으로 하여 개발자들은 다른 COM 에 기반한 기술들과 상호작용할 수 있는 그들만의 확장부분을 생성할 수 있다.

델파이에는 이런 COM, OLE, 액티브 X 응용프로그램의 핵심적인 요소를 쉽게 생성할 수 있는 클래스들과 마법사들이 준비되어 있다. Delphi object framework 에는 하나의 어플리케이션에서 쓰이는 간단한 COM-호환 클래스들로부터 OLE 자동화 서버와 액티브 X 컨트롤에서 쓰일 수 있는 클래스들까지 지원하고 있다. 델파이를 COM-기반 어플리케이션을 개발하는데 사용하여 어플리케이션의 내부적으로 인터페이스를 기반으로한 세련된 소프트웨어 디자인을 할 수 있으며, 윈도우 95 셸 확장이나 DirectX 와 같은 COM 에 기초한 여러가지 API 객체 들과 상호작용이 가능한 객체들을 생성해낼 수 있다.

델파이 4 가 지원하는 위저드로 생성할 수 있는 것으로는 다음과 같은 것들이 있다.

- 단순한 COM 객체
- 자동화 서버
- 자동화 컨트롤러
- 액티브 X 서버와 액티브 폼
- 마이크로소프트 트랜잭션 서버(MTS) 객체

자주 쓰이는 용어 정의부터 하겠다.

인터페이스: 명확하게 정의된 목적이 있는 메소드 프로토타입들의 세트

COM 객체: CoClass 라고 불리는 클래스의 인스턴스로 COM 인터페이스들의 메소드들을 실제로 구현한다.

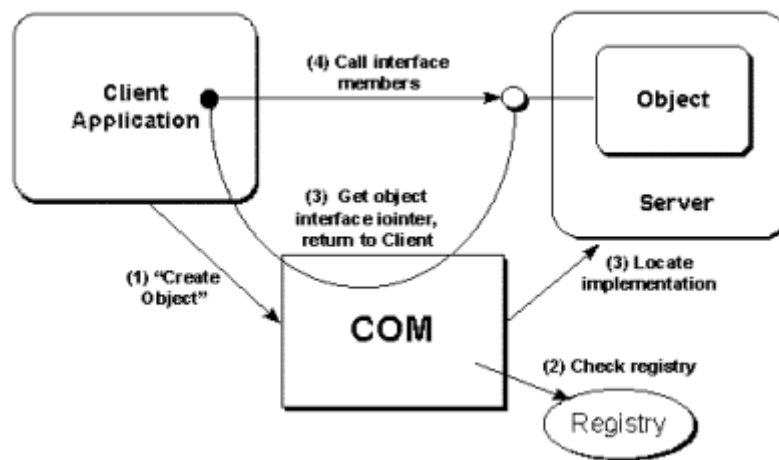
COM/액티브 X 서버: COM 이나 액티브 X 객체들을 포함한 모듈 (EXE, DLL, OCX).

클래스 팩토리: 지정된 CoClass 로부터 COM 객체를 생성할 수 있는 객체

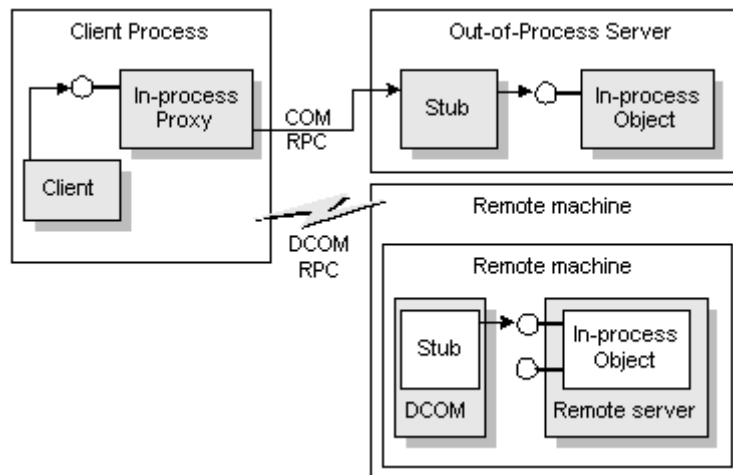
타입 라이브러리: 리소스로 저장이 가능한 이진 심볼 파일로, COM 이나 액티브 X 서버에 대한 데이터 형 정보를 담고 있다.

## COM 아키텍처의 이해

이들 COM 객체가 동작하는 방식은 아래 그림과 같다. 클라이언트 어플리케이션이 ClassID 를 가지고 COM 객체를 생성하는 요구를 하면, OS 에서 제공하는 COM 라이브러리 (윈도우 95 의 경우 COMOBJ.DLL, OLE32.DLL 에 구현되어 있다.)가 시스템 레지스트리에 서 해당되는 CLSID 를 가진 COM 객체를 찾게 된다. 이때 이 COM 객체는 클라이언트 어플리케이션의 요구를 서비스하는 COM 서버로 동작하게 되는 것이다. 이렇게 찾은 COM 객체의 구현 부분을 찾아서 해당 인터페이스 멤버 함수의 포인터를 클라이언트의 어플리케이션의 해당 부분에 매핑 시켜준다.



그런데 이때 COM 객체의 위치는 하나의 프로세스 공간에 있을 수도 있고, 다른 작업 공간에 있을 수도 있으며, 심지어는 다른 컴퓨터 안에 있을 수도 있다. 이때 COM 객체의 인터페이스 포인터의 위치를 하나의 프로세스 공간에 있는 것처럼 포장하는 과정을 마샬링이라고 하며, 이때 COM 서버 측의 포인터를 포장하는 객체를 스텝(stub), 클라이언트 어플리케이션 측의 객체를 프록시(proxy)라고 한다. 이를 그림으로 정리해보면 다음과 같다.



만약 클라이언트 어플리케이션에서 크래쉬가 일어나게 되면(예를 들어 클라이언트 어플리케이션이 동작하는 컴퓨터가 불의의 사고로 다운된 경우) 클라이언트의 프록시 객체는 파괴되며, 이와 연관된 서버 측의 스텝 객체가 커넥션이 단절된 것을 감지하여 클라이언트 객체와 연결된 수만큼 레퍼런스 카운트를 감소시키며, 이것이 0 이 되면 서버 객체가 파괴된다. 반대로, 서버 객체가 비정상 종료되면 클라이언트 어플리케이션은 프록시 객체에서 에러 코드를 받을 수 있게 된다 (DLL 의 경우에는 클라이언트 어플리케이션에도 크래쉬가 일어나면서 비정상 종료하게 된다.).

## COM 어플리케이션의 구성

COM 어플리케이션을 구현할 때에는 다음과 같은 부분을 제공해야 한다.

- COM 인터페이스  
객체가 자신의 서비스를 클라이언트에게 노출시키는 방법이다. COM 객체는 인터페이스를 연관된 메소드와 프로퍼티의 세트로 제공한다.
- COM 서버  
EXE, DLL, OCX 형태의 모듈로 COM 객체를 위한 코드를 포함하고 있다. 객체 들은 이 서버 안에 있으며, COM 서버는 하나 이상의 인터페이스를 구현한다.
- COM 클라이언트  
서버에서 요구한 서비스를 얻기 위해 인터페이스를 호출하는 코드이다. 클라이언트는 서버에서 제공하는 서비스가 어떻게 구체적으로 동작하는지는 모른다. COM 클라이언트 중 가장 흔한 것은 자동화 컨트롤러이다.

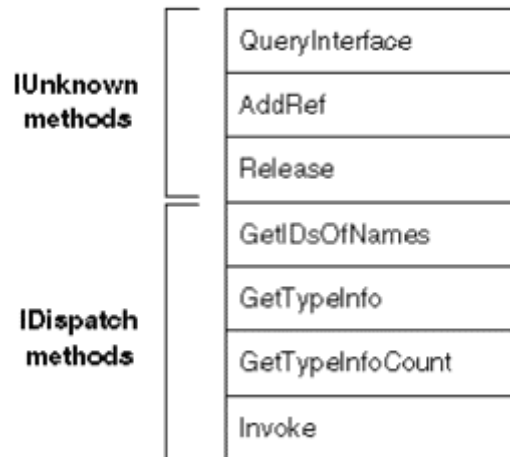
## OLE 와 액티브 X 객체

OLE, 액티브 X 객체들은 비주얼할 수도 논비주얼할 수도 있으며, 클라이언트에서 같은 작업공간에서 돌아갈 수도 있고, 다른 작업공간에서 돌아갈 수도 있다. 다른 작업공간에서 돌아가게 만들거나 remote-procedure call 을 쓸 수 있게 하려면 필요한 인자들을 package 하고, unpackage 할 수 있어야 한다. 위에서도 설명했듯이 이와 같은 과정을 마샬링이라고 하는데, 이렇게 RPC 를 쓰거나 프로세스 공간을 초월한 어플리케이션 작업을 할 때에는 여러가지 방법으로 여러가지 인자들이 마샬링 작업을 거쳐야 하며, 여러가지 종류의 OLE 와 액티브 X 객체들이 서로 다른 방법의 마샬링 작업을 하게 된다.

다음 표에 생성이 가능한 OLE 객체의 type 들을 나열해 보았다.

OLE 객체	비주얼	작업 공간	통 신	타입 라이브러리
OLE/Active 문서	거의	In-process, cross-process	Verbs for marshaling	No
OLE 자동화	일부	In-process, cross-process, or remote	IDispatch auto-marshaling	Recommended
액티브 X 컨트롤	거의	In-process	IDispatch auto-marshaling	Required
사용자 정의 인터페이스 객체	옵션	In-process, cross-process, remote	Manual marshaling	Recommended

OLE 자동화 서버들은 IDispatch 인터페이스를 구현한 객체들이다. IDispatch 는 IUnknown 에서 파생되었기 때문에 IUnknown 인터페이스도 구현하고 있다. IUnknown 인터페이스는 IDispatch 인터페이스의 메소드들이 OLE 자동화 객체의 메소드와 프로퍼티에 접근하는 동안 서버 객체가 지원하는 다른 모든 인터페이스들을 관리한다. 다음 그림은 IUnknown 과 IDispatch 인터페이스를 지원하는 객체의 IDispatch 인터페이스 vtable 의 모식도이다.



OLE 자동화 컨트롤러는 OLE IDispatch 인터페이스를 이용해 이 인터페이스를 구현한 OLE 서버 객체에 접근하는 클라이언트이다. 컨트롤러는 반드시 먼저 객체를 생성하고, 그 IDispatch 인터페이스 객체의 포인터에 해당하는 IUnknown 인터페이스를 질의(query)로 알아낸다. OLE 자동화 컨트롤러는 OLE 자동화 객체에 다음의 두가지 다른 방법으로 접근이 가능하다.

- IDispatch 인터페이스를 이용한다.
- 만약 듀얼 인터페이스 (dual interface)가 정의된 경우라면 객체의 가상함수 테이블 (virtual function table, vtable)이 멤버 함수를 직접 호출한다.

## 인터페이스 라이브러리의 이용

특별한 type 의 객체를 제작하거나 사용할 때에는 그 type 에 대한 해당되는 라이브러리나, 표준 객체에 대한 규칙을 참조해야 한다. 다음에 인터페이스 라이브러리의 type 에 대한 요약과 가능한 확장성을 기술하였다.

### 1. COM 인터페이스

COM 은 인터페이스의 표준 라이브러리를 정의하고 있다. 여기에는 인터페이스의 기본적인 기능을 정의하는 IUnknown, 외부에서 COM 객체를 인스턴스화할 때 필요한 클래스 factory 객체를 정의하는 IClassFactory 인터페이스 등을 포함한다.

COM 인터페이스를 구현하는 클래스의 제작에 관심이 있는 개발자들은 델파이에서 제공하는 TInterfacedObject, TComObject, TTypedComObject, TComObjectFactory, TTypedComObjectFactory 클래스를 참조하면 도움이 된다.

## 2. OLE/액티브 X 확장 부분

OLE 라이브러리 확장 부분은 많은 수의 객체 type 을 지원하며, 각각의 객체는 지원할 수 있거나 반드시 지원해야 하는 인터페이스들의 정의를 가지고 있다. OLE 그 자체는 아주 일반적인 목적의 인터페이스들을 많이 제공하는데, 이런 인터페이스는 IOle 로 시작한다.

OLE 처럼 액티브 X 역시 COM 의 구현 부분이다. 액티브 X 기술을 이용한 공통적인 어플리케이션에는 Active 문서, 액티브 X 컨트롤 등이 있다. 이 기술은 COM 에 대한 확장 부분으로 개발된 것으로 크기와 속도를 최적화한 기술이다.

## 3. 액티브 도큐먼트 (Active document)

액티브 도큐먼트는 문서 객체와 컨테이너에 대한 인터페이스의 라이브러리이다. 이 라이브러리에서는 그 자신의 인터페이스를 이용해서 컨테이너와 객체 어플리케이션간의 링킹과 임베딩을 위한 통신 등의 다양한 서비스를 제공한다. 액티브 도큐먼트 컨테이너를 제작하는데 관심이 있는 개발자는 TOleContainer 컴포넌트를 이용하면 된다.

## 4. OLE 자동화

OLE 자동화는 IDispatch 인터페이스의 구현을 필요로 한다. OLE 자동화는 서로 다른 작업 공간에서의 파라미터의 전송과 패키징, 객체의 저장과 객체 이름의 번역 등을 지원하는 서비스를 포함한다. OLE 자동화에 관심이 있는 개발자는 IDispatch 인터페이스를 구현하고 있는 TAutoObject 클래스를 이용하면 된다.

## 5. 액티브 X 컨트롤

액티브 X 컨트롤은, 과거에 OLE 컨트롤이나 OCX 로 불리던 것으로, 액티브 X 라이브러리에 있는 인터페이스들을 구현한 객체를 가리키는 말이다. 이들 인터페이스들은 객체가 이벤트를 발생시키고, 데이터 소스에 바인딩이 가능하고, 라이선싱을 지원하는 인터페이스를 제공한다. 델파이는 TActiveXControl 과 TActiveForm 클래스를 제공하며 동시에 쉽게 액티브 X 객체를 제작할 수 있도록 위저드를 제공한다.

## 6. 커스텀/서드파티 인터페이스의 구현

자기 자신의 라이브러리와 규칙 들을 정의할 수 있다. 커스텀 또는 서드파티 인터페이스를 구현하는 객체를 제작할 때에는 TComObject, TTypedComObject, TAutoObject 를 기초 클래스로 하면 된다.

## 클래스 팩토리

COM 객체는 하나 이상의 COM 인터페이스를 구현한 CoClass 의 인스턴스이며, COM 객체는 CoClass 인터페이스에 의해 정의된 서비스를 제공한다.

클래스 팩토리는 COM 객체를 제작하고 등록하는데 사용하는 객체이다. 클래스 팩토리는 CoClass 를 인스턴스화하는 표준 메커니즘을 제공한다. 클래스 팩토리를 이용하여 객체를 인스턴스화하면 클라이언트가 유일한 identifier 만 알게 되면 CoClass 에 대해 다른 것들은 알 필요가 없어진다. 클래스 팩토리는 생성자(constructor) 메소드와 argument 등을 다룬다. 각각의 CoClass 정의에는 클래스 팩토리가 있다. 클래스 팩토리는 하나 이상의 CoClass 와 관련이 있을 수 있으며, 이들 각각이 인스턴스화될 수 있다. 어쨌든 일단 클래스 팩토리 자신이 인스턴스화되면 이것은 특정 클래스 identifier(CLSID)와 연관이 되며, 이것이 생성자(constructor)에 넘겨진다. 이 파라미터는 어떤 CoClass 가 클래스 팩토리에 의해 인스턴스화 되는지를 나타낸다. 그래서, 클래스 팩토리 클래스가 하나 이상의 CoClass 에 적용되면 각각의 클래스 팩토리 인스턴스는 오직 하나의 CoClass 의 인스턴스를 생성할 수 있다.

### 1. CoClass 의 인스턴스화

CoClass 는 CoGetClassObject 라는 글로벌 윈도우 API 함수를 호출하여 인스턴스화될 수 있다. 이 함수는 레지스트리에서 CLSID 를 찾고, 서버의 패스를 알아내어 서버를 로드하고 클래스 팩토리 인터페이스(보통은 IClassFactory)에다가 포인터를 넘긴다. IClassFactory 포인터는 CoClass 의 인스턴스를 생성할 때 쓰이는 클래스 팩토리의 CreateInstance 메소드를 호출할 때 쓰인다. 위의 과정을 밟지 않고, API 함수인 CoCreateInstance 를 호출하면 위의 모든 과정을 한 번에 해준다. 일단 서버를 로딩하면 CoCreateInstance 는 자동으로 CoGetClassObject 함수를 호출하고, IClassFactory 포인터를 통해서 CreateInstance 메소드를 호출한다. 어쨌든 클래스의 여러 개의 인스턴스를 생성할 때에는 하나의 클래스 팩토리를 이용해서 CreateInstance 메소드를 직접 호출하는 것이 빠르다. CoGetClassObject 만이 IClassFactory 인터페이스 포인터를 제공할 수 있기 때문에, CoCreateInstance 를 호출하기 보다는 CoGetClassObject 를 호출하는 것이 빠르다.

### 2. 델파이의 클래스 팩토리 클래스

객체가 클래스 팩토리가 되려면 IClassFactory 인터페이스를 지원해야 한다. 델파이의 CoClass 들은 그에 해당하는 클래스 팩토리 클래스들을 가지고 있다. 이들은 IClassFactory 인터페이스를 구현하거나 IClassFactory2 인터페이스를 지원한다.

IClassFactory2 인터페이스는 라이선싱이 필요한 액티브 X 객체를 지원한다.

텔파이의 클래스 팩토리 객체들은 서버를 포함한 유닛의 initialization 섹션에서 생성되어야 한다. 이로 인해 서버가 일단 로드되면 클래스 팩토리는 자동적으로 사용이 가능해진다. 텔파이의 COM 서버 클래스는 COM 또는 액티브 X 서버의 제작에 쓰이는데, 여기에는 클래스 팩토리를 관리하는 메소드들을 포함하고 있다. 클래스 팩토리에는 reference count 를 하는 기능이 있기 때문에 서버는 언제 unload 되어야 하는지 알 수 있다.

## COM, 액티브 X 서버

COM 서버는 클라이언트 어플리케이션이나 라이브러리에 서비스를 제공하는 어플리케이션이나 라이브러리를 말한다. COM 서버는 클라이언트와 같은 작업 공간에서 동작하는 DLL 인 in-process 서버일 수도 있고, 클라이언트와 다른 작업 공간이 되 같은 기계 안에서 동작하는 EXE 파일인 로컬 서버일 수도 있으며, 클라이언트와 다른 기계에서 동작하는 리모트 서버일 수도 있다. COM 서버는 COM 객체가 존재하는 모듈이다. COM 서버는 OLE 자동화 객체, 액티브 X 컨트롤, 액티브 폼에 대한 코드를 담고 있다. 텔파이는 액티브 X 서버가 가지는 기본적인 기능들을 캡슐화한 클래스를 가지고 있는데, 그 기능에는 다음과 같은 것들이 포함되어 있다.

1. 서버의 등록, 클래스의 등록, 서버를 로딩하거나 언로딩하거나 객체의 인스턴스화를 담당하는데 필요한 루틴들을 export 한다.
2. 서버에 객체를 구현하는데 필요한 클래스 factory 의 생성과 관리
3. 서버의 type, help 파일, 서버의 이름, 타입 라이브러리 등의 서버에 대한 핵심적인 정보를 제공한다.

텔파이의 서버 클래스는 타입 라이브러리를 필수로 한다. 여기에는 서버에서 사용이 가능한 객체와 인터페이스에 대한 정보를 담고 있다.

## 마샬링(marshaling) 기전

마샬링은 클라이언트가 다른 프로세스나 기계에 있는 원격 객체의 인터페이스 함수를 호출할 수 있도록 해주는 기전을 말한다. 다른 말로 하면, 서버 프로세스의 인터페이스 포인터를 클라이언트 프로세스에서 사용할 수 있는 포인터로 바꾸고, 클라이언트의 파라미터를 원격 객체의 프로세스 공간으로 옮겨야 한다.

어느 인터페이스 호출에서도 클라이언트는 argument 들을 스택에 밀어넣고, 인터페이스 포인터를 호출하여 함수를 실행한다. 만약 이때 객체가 in-process 가 아니면 호출된 함수는 프록시에 전달된다. 프록시는 argument 들을 마샬링 패킷에 포장하고, 이를 원격 객체에



전달한다. 원격 객체의 스텝은 이 패킷을 풀어서, argument 들을 스택에 집어 넣고, 객체의 구현 부분을 호출한다. 즉, 객체는 클라이언트의 함수 호출하는 방식을 자신의 프로세스에서 자신의 주소를 이용하여 재생성하는 것이다.

어떤 형태의 마샬링을 사용할 것인가 하는 문제는 COM 객체가 구현된 방법에 달려있다. IDispatch 인터페이스가 제공되는 경우에는 표준 마샬링 기법을 이용하게 된다. 이 방법은 시스템 표준 RPC 를 통해 통신하게 된다.

## 자동화 (Automation)

자동화란 어플리케이션이 다른 어플리케이션의 객체를 제어할 수 있는 능력을 말한다. 자동화 객체의 클라이언트는 자동화 컨트롤러(automation controller)라고 하며, 서버 객체를 자동화 객체(automation object)라고 한다. 자동화는 in-process, 로컬, 원격 서버에서 모두 사용이 가능하다.

자동화의 가장 큰 특징은 다음의 2 가지 이다.

1. 자동화 객체는 반드시 객체의 인터페이스와 인터페이스 메소드, 파라미터 등의 정보를 기술해야 한다. 이런 정보는 보통 타입 라이브러리에 기록되며, 이를 이용해 여러가지 작업을 할 수 있게 된다. 델파이 4 에서 제작한 자동화 서버에도 타입 정보가 포함된다.
2. 자동화 객체는 반드시 메소드 들을 다른 어플리케이션이 접근할 수 있도록 해야 한다. 이를 위해 반드시 IDispatch 인터페이스를 구현해야 하며, 이를 통해 인터페이스의 메소드와 프로퍼티에 접근할 수 있게 된다.

## 액티브 X 컨트롤

액티브 X 컨트롤은 in-process 서버에서만 돌아가는 비주얼 컨트롤로 OLE 컨테이너 어플리케이션에 플러그인 될 수 있다. 액티브 X 컨트롤은 자체로 완전한 어플리케이션이 될 수는 없지만, 여러가지 어플리케이션에서 사용될 수 있는 재사용 가능한 컨트롤이다. 액티브 X 컨트롤의 프로퍼티, 메소드, 이벤트를 사용하려면 자동화 기법을 이용해야 한다.

액티브 X 컨트롤은 웹 페이지에서 많이 사용되고 있다. 그렇기 때문에 액티브 X 는 현재 웹에서 interactive 한 콘텐츠를 제공하는 표준으로 점점 자리를 잡아가고 있다.

델파이 4 에서는 이런 액티브 X 컨트롤을 쉽게 제작할 수 있는 위저드를 제공한다.

## 타입 라이브러리 (type libraries)

타입 라이브러리는 COM 객체에 어떤 인터페이스가 존재하는지, 그리고 각 인터페이스 메

소드 argument 들의 수와 데이터 형 등의 정보를 담고 있다. 또한, CoClass 의 identifier(CLSIDs)와 인터페이스의 identifier(IIDs) 그리고, 자동화 인터페이스 메소드에 대한 디스패치 identifier(dispatchIDs) 등도 가진다.

타입 라이브러리에는 그 밖에도 다음과 같은 정보를 담고 있다.

- 사용자 정의 인터페이스와 연관된 사용자 정의 데이터 형 정보
- 인터페이스 메소드가 아니면서도 자동화나 액티브 X 서버에 의해 export 되는 루틴
- 열거형(enumeration), 구조체(structure), 공용체(union), 앨리어스, 모듈 데이터 형 등에 대한 정보
- 다른 타입 라이브러리의 타입 기술에 대한 레퍼런스

#### ● 타입 라이브러리의 생성 방법과 사용

지금까지의 개발 도구에서는 IDL(Interface Description Language) 또는 ODL(Object Description Language)를 이용하여 스크립트를 작성함으로써 타입 라이브러리를 작성했다. 그리고, 이렇게 작성한 스크립트를 컴파일러를 동작시켜 헤더 파일을 만들어 사용해왔다. 델파이 4 는 자동화 서버나 액티브 X 컨트롤 위저드를 사용할 때 타입 라이브러리를 자동으로 만들어 준다. 그리고, 타입 라이브러리 에디터를 이용해서 타입 라이브러리의 내용을 보면서 쉽게 정보를 편집할 수 있게 해주며, 델파이는 해당되는 소스를 자동으로 업데이트 한다.

타입 라이브러리 에디터는 자동으로 표준 타입 라이브러리를 생성하고, 델파이용 인터페이스 파일(.pas 파일)에 오브젝트 파스칼 문법으로 인터페이스 정의를 한다.

외부 사용자에게 노출될 객체 들의 타입 라이브러리를 작성하는 것은 매우 중요한 작업이다. 예를 들면 다음과 같은 것들이 있다.

- 액티브 X 컨트롤은 액티브 X 컨트롤이 포함된 DLL 에 타입 라이브러리가 리소스로 함께 포함되어 있기를 요구한다.
- 자동화 서버를 구현한 어플리케이션은 반드시 타입 라이브러리를 제공해서, 클라이언트가 early 바인딩을 사용할 수 있도록 해야 한다.
- Vtable 바인딩을 지원하는 노출된 객체 들은 반드시 타입 라이브러리 기술되어야 한다. 이는 vtable 레퍼런스가 컴파일 시에 사용되기 때문이다.
- IProvideClassInfo 인터페이스를 지원하는 클래스에서 인스턴스화된 객체들은 반드시 타입 라이브러리를 가져야 하는데, 델파이의 VCL 중에서는 TTypedComObjectClass 를 상속한 클래스 들이 해당된다.
- Drag-and-drop 에는 타입 라이브러리가 반드시 필요한 것은 아니지만, 제공되면 객체를 확인할 때 유용하다.

만약 인터페이스를 단지 어플리케이션 내부에서만 사용할 것이라면 타입 라이브러리는 생성할 필요가 없다.

#### ● 타입 라이브러리의 접근 방법과 활용

이진 타입 라이브러리는 보통 리소스 파일(.res)의 일부이거나 .tlb 파일 확장자를 가진 독자적인 파일로 존재한다. 일단 타입 라이브러리가 생성되면, 오브젝트 브라우저(object browser), 컴파일러 등의 도구가 특별한 타입 인터페이스를 통해 타입 라이브러리에 접근할 수 있다. 이러한 타입 인터페이스에는 다음과 같은 것들이 있다.

인터페이스	설 명
ITypeLib	타입 라이브러리에 접근할 수 있는 메소드를 제공한다.
ITypeInfo	타입 라이브러리에 포함된 각 객체에 대한 설명을 제공한다. 예를 들어, 브라우저는 이 인터페이스를 이용하여 타입 라이브러리에서 각 객체에 대한 정보를 뽑아낸다.
ITypeComp	컴파일러가 인터페이스에 바인드할 때 필요한 접근 정보를 빨리 뽑아낼 수 있다.

델파이는 다른 어플리케이션에서 타입 라이브러리를 import 하거나, 이를 사용할 수 있다. COM 어플리케이션을 위해 사용된 대부분의 VCL 클래스는 타입 라이브러리와 실행 중인 객체의 인스턴스에서 타입 정보를 불러오거나 저장할 때 사용되는 핵심 인터페이스를 제공한다. TTypedComObject VCL 클래스는 타입 정보를 제공하는 인터페이스를 지원하며, 액티브 X 객체 프레임웍에서 사용되는 기반 클래스이다.

어플리케이션에서 타입 라이브러리를 요구하지 않아도, 타입 라이브러리를 사용하면 다음과 같은 잇점을 얻을 수 있다.

- 데이터 형 검사를 컴파일 시에 할 수 있다.
- 자동화에서 early 바인딩을 사용할 수 있으며, vtable 이나 듀얼 인터페이스를 지원하지 않는 자동화 컨트롤러가 dispIDs 를 컴파일 시에 인코드할 수 있으며, 이로 인한 런타임 수행 성능의 향상을 얻을 수 있다.
- 타입 브라우저(Type browsers)로 라이브러리를 스캔할 수 있으므로, 클라이언트가 개발자가 만든 객체의 특징을 볼 수 있게 된다.
- RegisterTypeLib 함수를 이용하여 노출된 객체 들을 레지스트리 데이터베이스에 등록하는데 사용할 수 있다.
- 시스템 레지스트리에서 객체를 제거할 때에도 UnRegisterTypeLib 함수를 사용할 수 있다.
- 자동화 작업이 타입 라이브러리에서 얻은 정보를 이용하여 파라미터를 포장하게 되므로,

로컬 서버에 접근하는 성능이 향상된다.

## 액티브 도큐먼트 (Active Documents)

과거에 OLE 도큐먼트로 불리던 액티브 도큐먼트는 연결(linking)과 임베딩(embedding), drag-and-drop, 비주얼 편집(visual editing)을 지원하는 COM 서비스의 세트이다.

액티브 도큐먼트는 사운드 클립, 스프레드 시트, 텍스트와 비트맵과 같은 다른 포맷의 데이터와 객체를 통합할 수 있다. 액티브 X 컨트롤과는 달리 액티브 도큐먼트는 in-process 서버로 한정되지 않고, 프로세스의 경계를 넘어서 사용될 수 있다. 참고로, 액티브 도큐먼트의 스펙을 살펴보면 프로세스의 경계를 넘어서 사용할 수 있도록 마샬링이 지원되지만, 사실상 액티브 도큐먼트는 동일 기계 상의 다른 프로세스에서는 사용될 수 있지만 원격 서버에서는 동작하지 않는다. 이는 액티브 도큐먼트가 사용하는 데이터 형이 윈도우 핸들, 메뉴 핸들과 같이 주어진 기계 상의 시스템에 특정한 것들을 사용하기 때문이다.

대부분이 비주얼하지 않은 자동화 객체와는 달리, 액티브 도큐먼트 객체는 다른 어플리케이션에서도 비주얼하다. 그러므로, 액티브 도큐먼트 객체는 디스플레이나 출력 장치에 비주얼 하게 보여지는데 사용되는 프리젠테이션 데이터(presentation data)와 객체를 편집할 때 사용되는 원시 데이터(native data)의 2 가지 데이터 종류를 가지게 된다.

액티브 도큐먼트 객체는 도큐먼트 컨테이너이거나 도큐먼트 서버일 수 있다. 텔파이 4 는 액티브 도큐먼트를 생성하는 자동화 위저드를 제공하지 않지만, TOleContainer VCL 클래스를 이용하여 이미 존재하는 액티브 도큐먼트를 연결하거나 임베딩할 수 있다. 또한, TOleContainer 는 액티브 도큐먼트 컨테이너로 사용할 수도 있다. 액티브 도큐먼트 서버에 대한 객체를 생성하려면 COM 기초 클래스 하나를 사용하고, 지원하고자 하는 객체 데이터 형에 대한 적절한 인터페이스를 구현하여야 하며, 위저드는 제공되지 않는다.

## COM 과 VCL 클래스

텔파이에서 COM 기술을 구현한 유닛을 살펴보면, 우선 COM 기술의 API 세트와 인터페이스 정의를 오브젝트 파스칼의 형식에 맞도록 선언해 놓은 부분이 ActiveX.pas 와 Ole2.pas, OleCtl.pas, OldDlg.pas 유닛에 기록되어 있다. ActiveX.pas unit 은 텔파이에서 지원하는 인터페이스 선언문을 사용하여 COM 인터페이스를 재선언하고, COM 라이브러리 DLL 에서 지원하는 API 세트를 정의해 놓은 유닛이다. 그러므로, COM 기술을 위한 모든 VCL 클래스들은 기본적으로 이 유닛을 사용해야만 한다. 이렇게 텔파이가 액티브 X 기술을 지원하기 위해서 제공하는 클래스를 DAX(Delphi ActiveX Framework)라고 한다.

실제로 COM 객체 클래스에서 가장 기초 클래스가 되는 TComObject, TAutoObject, TComClassFactory 등의 클래스는 ComObj.pas unit 에 구현되어 있다. 텔파이의 모든 COM 객체는 TComObject 클래스에서 상속받은 객체라고 할 수 있는데, 그 선언부분은 다

음과 같으며, 이해를 돕기 위해 필자가 약간의 주석을 달았다.

```
TComObject = class(TObject, IUnknown, ISupportErrorInfo)
```

```
{액티브 X.pas unit 에 선언된 IUnknown, ISupportErrorInfo 인터페이스를 구현한 클래스임을 나타내는 코드}
```

```
private
```

```
    FRefCount: Integer;           //내부적인 참조계수로 사용되는 변수
```

```
    FFactory: TComObjectFactory; //COM 객체의 클래스 factory
```

```
    FController: Pointer;        //Aggregation 된 경우의 IUnknown 의 구현부분
```

```
    function GetController: IUnknown;
```

```
protected
```

```
    //아래의 메소드는 IUnknown 자체의 구현이다.
```

```
    function IUnknown.QueryInterface = ObjQueryInterface;
```

```
        //QueryInterface 를 TComObject.ObjQueryInterface 가 구현한다는 의미
```

```
    function IUnknown._AddRef = ObjAddRef;
```

```
    function IUnknown._Release = ObjRelease;
```

```
    //다른 인터페이스들에 대한 IUnknown 메소드
```

```
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
```

```
        //해당 IID 를 받아서 IUnknown 의 QueryInterface 를 실행
```

```
    function _AddRef: Integer; stdcall;
```

```
    function _Release: Integer; stdcall;
```

```
    //아래의 메소드는 ISupportErrorInfo 인터페이스를 구현한다.
```

```
    function InterfaceSupportsErrorInfo(const iid: TIID): HRESULT; stdcall;
```

```
public
```

```
    constructor Create; //Aggregate 의 부분이 아닌 COM 객체를 생성하는 메소드
```

```
    constructor CreateAggregated(const Controller: IUnknown);
```

```
//Aggregate 된 부분으로서의 COM 객체의 생성을 담당하는 메소드
```

```
    constructor CreateFromFactory(Factory: TComObjectFactory;
```

```
        const Controller: IUnknown);
```

```
        //실제 constructor 역할을 하는 메소드로 Create, CreateAggregated 메소드에 의해 호출된다.
```

```
COM 객체에 메모리를 할당하고 프로퍼티 들을 설정한다.
```

```

destructor Destroy: override;           //COM 객체를 파괴하는 메소드
procedure Initialize: virtual;          //가상함수로 초기화에 쓰인다.
function ObjAddRef: Integer; virtual; stdcall;
function ObjQueryInterface(const IID: TGUID; out Obj): Integer; virtual; stdcall;
function ObjRelease: Integer; virtual; stdcall;
function SafeCallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HRESULT; override;
property Controller: IUnknown read GetController;
//컨트롤러가 되는 IUnknown 인터페이스
property Factory: TComObjectFactory read FFactory;
property RefCount: Integer read FRefCount;
end;

```

{COM class }

```
TComClass = class of TComObject; //TComClass 클래스의 정의
```

TComObject 는 클래스 identifier 를 가지는 COM 객체에 대한 기초 클래스이다. 이때 CLSID 를 사용하는데 이것은 클래스를 데이터베이스 레지스트리에 등록하고 클래스 팩토리를 통해 외부에 인스턴스화 하는데 사용된다. TComObject 의 클래스 팩토리는 TComObjectFactory 가 담당하며, 클래스 팩토리의 프로퍼티에서 이름, 설명, CLSID 등의 TComObject 클래스의 여러가지 정보를 제공한다. 또한, TComObjectFactory 메소드를 사용하여 TComObject 클래스를 레지스트리에 등록하고 인스턴스화 할 수 있다.

TComObject 는 하나의 COM 객체로 인스턴스화될 수도 있고, aggregate 의 한 부분으로 인스턴스화될 수도 있다. 만약 인스턴스화된 COM 객체가 aggregate 의 내부 객체라면 그 IUnknown 인터페이스 메소드가 적절한 컨트롤러 인터페이스에 의해 동작하여 aggregation 을 지원하게 된다. 결과적으로 내부 COM 객체에 의해 구현된 모든 인터페이스들은 인터페이스 참조가 생길 때마다 참조계수(reference count)가 직접적으로 영향받지 않고, 컨트롤러 인터페이스에 의해 영향 받는다.

마지막으로 TComObject 는 ISupportErrorInfo, InterfaceSupportsErrorInfo 를 구현한다. 그러므로, OLE 예외처리와 safecall 호출규칙을 지원한다.

델파이의 COM 지원은 이렇게 TComObject 를 기초 클래스로 해서 풍부한 VCL 클래스로 이루어져 있다. 이들 중 가장 중요한 클래스들은 TTypedComObject, TAutoObject, TActiveXControl 등을 들 수 있다. 이들중 TTypedComObject 와 TAutoObject 는 ComObj.pas unit 에 구현되어 있으며, TActiveXControl 의 AxCtrls.pas unit 에 구현되어 있다. AxCtrls.pas unit 에서는 VCL 클래스를 액티브 X 컨트롤로 전환하기 위한 TActiveXControl 클래스의 많은 인터페이스를 구현하고 있다.

## 정 리 (Summary)

이번 장에서는 텔파이 4 와 COM 에 대한 기초적인 개념 설명과 내용을 알아 보았다. COM 과 CORBA 에 대해서는 텔파이가 많은 일을 해 주지만, 기초적인 개념을 이해하고 있어야 제대로 된 어플리케이션을 만들 수 있다.

다음 장에서는 이번 장의 개념을 바탕으로 실제 COM 인터페이스를 활용하고, COM 객체를 만드는 방법에 대해서 알아보도록 할 것이다.

# COM 인터페이스와 COM 객체의 활용

## (Using COM Interfaces and COM Objects)

델파이에는 COM 을 지원하기 위한 여러가지 클래스가 존재한다. 이들을 이용하면 쉽게 가장 원시적인 COM 객체에서부터 액티브 폼과 같이 다소 복잡한 객체까지 구현이 가능하다. 이번 장에서는 COM 객체를 생성하고, 인터페이스를 구현하는 기본적인 방법과 이들의 활용방안에 대해서 알아보도록 한다.

### 인터페이스 활용하기

인터페이스에 대해서는 7 장에서 오브젝트 파스칼의 문법에 대해서 다룬 적이 있지만, COM 기술과 밀접한 관계가 있기 때문에 여기서 다시 한번 알아본다. 인터페이스는 추상 메소드 (abstract method)만을 포함한 클래스와 유사한 구조를 가지고 있다. 엄격하게 말해, 인터페이스 메소드 정의에는 파라미터의 수와 데이터 형, 리턴값의 데이터 형과 기대되는 행동에 대해 기술해야 한다. 이러한 인터페이스 메소드는 의미와 논리적으로 인터페이스의 목적과 부합되어야 한다.

인터페이스의 이름은 항상 I 로 시작하는 것이 규칙이다. 예를 들어, IPersist 인터페이스는 객체의 상태를 storage, stream, file 등에 저장하거나 읽어오는 메소드를 정의하는 인터페이스 들의 기초 인터페이스가 된다. 다음의 코드는 실제 인터페이스를 선언한 것으로 편집에 공통적으로 필요한 메소드를 모아서 IEdit 라는 인터페이스를 선언하였다.

type

IEdit = interface

procedure Copy; stdcall;

procedure Cut; stdcall;

procedure Paste; stdcall;

function Undo: Boolean; stdcall;

end;

추상 클래스와 마찬가지로, 인터페이스 역시 자체적으로 인스턴스화되지 못한다. 이들을 인스턴스화 하여 사용하려면 인터페이스를 반드시 클래스로 구현해야 한다. 다음의 코드를 살펴 보자.

TEditor = class(TInterfacedObject, IEdit)



```

    procedure Copy: stdcall;
    procedure Cut: stdcall;
    procedure Paste: stdcall;
    function Undo: Boolean: stdcall;
end;

```

여기서 TEditor 클래스는 TInterfacedObject 라는 기초 클래스를 상속받은 클래스로, TInterfacedObject 는 가장 기본적인 인터페이스라고 할 수 있는 IUnknown 을 구현한 클래스이다. 그러므로, IUnknown 인터페이스의 메소드는 구현할 필요는 없고 이와 같이 추가적으로 구현하겠다고 선언한 IEdit 인터페이스의 메소드 들을 각각 구현하게 된다.

### ● 클래스간 인터페이스 공유

인터페이스를 이용하면 구현되는 내용과 클래스를 분리해서 디자인할 수 있다. 또한, 2 개의 클래스가 같은 인터페이스를 공유하면 쉽게 다형성(polymorphism)을 구현할 수 있다. 그림을 그리는 기능을 선언한 IPaint 인터페이스와 이 인터페이스를 구현한 2 개의 클래스를 생각해보자.

```

IPaint = interface
    procedure Paint;
end;

```

```

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

```

```

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

```

이들은 IPaint 변수에 의해 서로 대입이 가능하며, 서로 다른 클래스의 메소드를 간단하게 하나의 메소드처럼 호출하는 것이 가능하다.

```

var
    Painter: IPaint;
begin

```

```

Painter := TSquare.Create;
Painter.Paint;
Painter := TCircle.Create;
Painter.Paint;
end;

```

여기에서, IRotate 라는 추가적인 인터페이스를 선언하고 클래스에 구현하도록 수정해보자.

```

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

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

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

```

원은 회전해도 모양이 변하지 않으므로, IRotate 인터페이스를 구현할 필요가 없다.

#### ● 프로시저에 인터페이스 활용하기

인터페이스를 이용하면 특정 기초 클래스에서 상속받은 객체를 요구하지 않고도 객체들을 다룰 수 있는 일반적인 프로시저를 작성할 수 있다. 앞에서 IPaint, IRotate 인터페이스를 가지고 다음과 같은 프로시저의 제작이 가능하다. 여기에서 IPaint 인터페이스를 구현한 여러 객체들의 배열을 파라미터로 사용할 수 있다.

```

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

```

```
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;
```

- as 연산자의 사용

인터페이스를 구현한 클래스는 as 연산자를 사용하여 인터페이스에 동적으로 바인딩할 수 있다. 다음 코드를 살펴 보자.

```
procedure PaintObjects(P: TInterfacedObject)
```

```
var
```

```
    X: IPaint;
```

```
begin
```

```
    X := P as IPaint;
```

```
    ...
```

```
end;
```

여기에서 TInterfacedObject 클래스인 변수 P 는 변수 X 에 대입될 수 있다. 이때 변수 X 는 IPaint 인터페이스의 레퍼런스가 된다. 이런 대입이 가능한 것이 동적 바인딩 때문이다. 이때 컴파일러는 P 가 IPaint 인터페이스를 지원하는지 알아내기 위해서 P 의 IUnknown 인터페이스의 QueryInterface 메소드를 호출하는 코드를 생성한다. 런타임에서 P 가 IPaint 레퍼런스 대입이 되지 않으면 예외가 발생한다. 그렇지만, 컴파일 시에는 최소한 P 가 IUnknown 만 지원한다면 에러는 발생하지 않는다.

- 포함과 대리, 통합 (Containment, Delegation and Aggregation)

COM 에서 인터페이스를 선언한 뒤에도 이런 인터페이스에 어떤 기능을 더 넣어서 확장을 하고 싶은 경우가 있을 것이다. COM 에서는 이런 경우에 포함과 통합을 이용하게 된다. 다시 말해서 포함과 통합은 어떤 객체가 다른 객체를 어떻게 사용할 것인가를 결정하는 기술이다.

## 1. 포함과 대리 (Containment and Delegation)

인터페이스를 이용한 코드 재사용 중에서 대표적인 방법 중의 하나가 다른 객체를 포함하거나, 다른 객체에 포함되는 포함(containment)이다. 포함에서 외부 객체가 내부 객체의 인터페이스 포인터를 포함한다. 즉, 외부 객체가 내부 객체의 인터페이스를 이용해서 자신의 인터페이스를 구현하는 것이다. VCL 은 포함과 코드 재사용을 위해서 프로퍼티를 사용한다. 이를 지원하기 위해서 델파이 4 에서 implements 키워드가 추가 되었다. 이를 이용하면 서브-객체(sub-object)에 대한 인터페이스의 구현의 일부 또는 전부를 대리(delegate)에 대한 코드로 쉽게 작성할 수 있다. Implements 키워드를 이용한 대리에 대한 내용은 7 장을 참고하기 바란다.

## 2. 통합 (Aggregation)

통합(aggregation)은 포함과 대리를 통한 코드 재사용과는 다르다. 즉, 외부 객체가 내부 객체를 포함하는데 이때 내부 객체에 의해 구현된 인터페이스는 외부 객체에 의해서만 노출될 수 있다. 통합은 코드 재사용을 할 때 서브-객체가 포함된 객체의 기능을 정의하지만, 구체적인 구현부는 숨겨지는 형태로 코드 재사용을 하기 때문에, 모듈화가 용이하다는 장점이 있다.

통합에서 외부 객체는 하나 이상의 인터페이스를 구현한다. 내부 객체 역시 하나 이상의 인터페이스를 구현할 수 있다. 그렇지만, 외부 객체만 인터페이스를 노출할 수 있다. 그러므로, 외부 객체에 의해 노출되는 인터페이스 중에는 외부 객체가 구현하는 것도 있고, 내부 객체가 구현하는 것도 존재하게 된다. 그렇지만, 클라이언트는 내부 객체에 대해서는 전혀 알 수가 없다. 그렇기 때문에, 외부 객체 클래스는 내부 객체 클래스 형을 같은 인터페이스를 구현하는 어떤 클래스와도 교체가 가능하다. 즉, 내부 객체 클래스에 대한 코드는 이를 사용하고자 하는 여러 클래스 들에 의해 공유될 수 있는 것이다.

통합에 대한 구현 모델은 대리(delegation)를 이용해서 IUnknown 을 구현하는 규칙을 명시적으로 정의한다. 내부 객체는 반드시 IUnknown 을 구현해야 하며, 내부 객체의 참조 계수(reference count)를 조절해야 한다. 이러한 IUnknown 의 구현은 외부와 내부 객체의 관계를 추적하게 된다. 예를 들어, 내부 객체와 같은 형의 객체가 생성될 때 요구된 인터페이스의 IUnknown 이 성공할 때에만 제대로 객체가 생성된다. 또한, 내부 객체는 구현하는 모든 인터페이스에 대한 2 번째 IUnknown 인터페이스를 구현해야 한다. 이 인터페이스들이 외부 객체에 대한 QueryInterface, AddRef, Release 메소드를 호출하는 대리(delegate) 역할을 하게 된다. 이때 외부에 노출된 IUnknown 을 Controlling Unknown 이라고 한다.

통합 클래스를 작성할 때에는 TComObject 의 IUnknown 인터페이스의 자세한 구현 부분을

참고로 하면 된다. TComObject 는 통합을 지원하는 COM 클래스이기 때문에, COM 어플리케이션을 작성할 때 TComObject 를 기초 클래스로 이용하면 통합을 쉽게 지원할 수 있다.

- 참조 계수에 대한 이해

델파이는 인터페이스 정의와 참조 계수 처리의 대부분을 구현한 클래스를 제공한다. TInterfacedObject 클래스는 이러한 역할을 하는 기초 클래스이다. 참조 계수를 사용하기 위해서는 객체를 인터페이스 레퍼런스로 사용하고, 참조 계수를 적용해야 한다. 다음의 코드를 살펴보자.

```
procedure beep(x: ITest);
function test_func()
var
    y: ITest;
begin
    y := TTest.Create;           //y 가 ITest 변수이므로, 참조 계수는 1 이 된다.
    beep(y);                    //y 를 이용한 호출이므로 참조 계수를 1 증가시킨 후, 다시 감소한다.
    y.something;                //현재 참조 계수는 1 이다.
end;
```

TInterfacedObject 를 이용하면, 이런 참조 계수의 사용이 자동으로 이루어 지게 되지만 다음과 같이 이를 제대로 다루지 않으면 객체가 갑자기 사라질 수도 있으니 주의해야 한다.

```
function test_func()
var
    x: TTest;
begin
    x := TTest.Create;           //인터페이스를 참조하지 않으므로 계수는 0 이다.
    beep(x as ITest);           //참조 계수가 1 증가 했다가 1 감소한다.
    x.something;                //객체가 없어진다 !
end;
```

객체가 VCL 컴포넌트이거나 다른 컴포넌트에 의해 소유된 컨트롤인 경우에는 TComponent 에 기초한 메모리 관리 시스템을 사용하게 된다. 이때 VCL 컴포넌트의 메모리 관리 시스템과 COM 의 참조 계수를 합쳐서 작성하면 안된다. 만약, 인터페이스를 지원

하는 컴포넌트를 만들고 싶다면 IUnknown 의 AddRef, Release 메소드를 COM 참조 계수 기전을 사용하지 않도록 empty 함수로 작성하면 된다.

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

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

QueryInterface 메소드는 객체에 대한 동적 질의를 제공하기 위해 구현해야 한다. QueryInterface 를 구현하면 as 연산자를 이용하여 컴포넌트에서 인터페이스를 사용할 수 있게 된다. 또한, 통합(aggregation)을 이용할 수 있는데 외부 객체가 컴포넌트이면 내부 객체는 참조 계수를 구현해야 한다. 이때 TInterfacedObject 를 기초 클래스로 내부 객체를 구현하고, 컴포넌트를 외부 객체로 사용하여 통합을 구현하는 것이 보통이다.

#### ● 분산 어플리케이션에서의 인터페이스 활용

인터페이스는 COM 이나 CORBA 분산 객체 모델의 핵심이다. 델파이는 이들의 공통적인 기초 클래스로 TInterfacedObject 를 제공한다. TInterfacedObject 는 단순히 IUnknown 인터페이스 메소드를 구현한다.

COM 클래스는 클래스 팩토리와 클래스 ID(CLSIDs)를 이용한 기능성이 추가된다. 클래스 팩토리는 CLSIDs 를 통해 클래스 인스턴스를 생성하게 되고, CLSIDs 는 COM 클래스를 등록하고 관리하는 기초 요소가 된다. 이때 클래스 팩토리와 클래스 ID 를 가진 COM 클래스를 CoClass 라고 한다.

참고로, CORBA 의 경우 클라이언트의 스텝(stub) 클래스와 서버의 스켈레톤(skeleton) 클래스를 통해 인터페이스를 사용하게 된다. 어플리케이션은 반드시 스텝과 스켈레톤 클래스를 사용하거나, 모든 파라미터를 가변형으로 변경하는 DII(Dynamic Invocation Interface)를 이용할 수 있다. 델파이에서는 CORBA 를 클래스 팩토리를 통해서 구현할 수 있도록 하였으며, CoClass 도 사용한다. 이렇게 COM 과 CORBA 를 구현하는 방법을 동일하게 사용함으로써 델파이는 COM/CORBA 클라이언트를 동시에 지원하는 서버를 작성할 수 있게 되었다.

## 델파이 COM 관련 클래스의 이해

델파이의 COM 관련 클래스에 대해서는 25 장에서 간단하게 다루었지만, 여기서는 가장 기초적인 클래스라고 할 수 있는 TInterfacedObject, TComObject, TTypedComObject 에 대해서 알아보도록 한다.

- TInterfacedObject

TInterfacedObject 클래스는 IUnknown 인터페이스를 구현한 기초 클래스이다. 이 클래스는 System.pas 유닛에 다음과 같이 선언되어 있다.

```
type
  TInterfacedObject = class(TObject, IUnknown)
  private
    FRefCount: Integer;
  protected
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    property RefCount: Integer read FRefCount;
  end;
```

사용법은 비교적 간단하다. 그대로 상속받아서 사용하는 것이다. 다음의 코드에서 TDerived 는 IPaint 인터페이스를 구현한 클래스이다.

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

IUnknown 을 구현하고 있기 때문에, 참조 계수 관리와 인터페이스 객체의 메모리 관리를 자동으로 해 주는 역할을 한다.

- TComObject

TComObject 는 IUnknown 이외에 COM 객체를 위한 추가적인 인터페이스를 지원한다. TComObject 는 반드시 클래스 ID 를 가져야 한다. CLSIDs 는 데이터베이스 레지스트리에 클래스를 등록하고, 클래스를 클래스 팩토리를 이용하여 외부에 인스턴스화할 때 필요하다. TComObject 에 대한 클래스 팩토리는 TComObjectFactory 이며, 클래스 팩토리 프로퍼티는 이름, 설명, CLSID 등의 TComObject 클래스에 대한 정보를 제공하며, 클래스 팩토리의 메소드를 이용하여 TComObject 클래스를 레지스트리에 등록하고, 이를 인스턴스화 한다. TComObject 는 단일 COM 객체로 인스턴스화 될 수도 있고, 통합(aggregation)의 한 부분으로서 인스턴스화할 수도 있다. TComObject 의 IUnknown 메소드 들은 통합의 내부 객체로 COM 객체가 인스턴스화할 때 IUnknown 인터페이스를 조절할 수 있도록 구현되어 있다. 결과적으로 내부 COM 객체에 의해 구현된 모든 인터페이스는 내부적인 참조 계수를 가지게 되지만, 이것이 인터페이스 레퍼런스가 생성될 때 직접 영향을 받지 않는다. ISupportErrorInfo 인터페이스는 OLE 자동화 컨트롤러가 에러 객체를 계속 사용할 수 있는지 여부를 질의하고 에러 정보를 호출 chain 에 과급할 수 있도록 한다. 또한, IErrorInfo 를 지원하여 OLE 예외 처리와 safecall 호출 규칙을 지원한다.

#### ● TTypedComObject

TTypedComObject 클래스는 TComObject 클래스에 IProvideClassInfo 인터페이스를 추가하여 객체에 대한 인터페이스와 데이터 형을 설명할 수 있도록 한다. 이를 위해서 타입 라이브러리를 요구하는데, 디폴트로는 듀얼 인터페이스를 지원한다.

IProvideClassInfo 는 GetClassInfo 라는 메소드를 지원하여 타입 정보를 제공할 수 있다. 즉, TTypedComObject 클래스에서 상속받은 객체가 실행될 때 클라이언트가 객체에 대한 정보를 GetClassInfo 메소드를 이용하여 얻을 수 있다.

TTypedComObject 클래스는 타입 라이브러리를 로드하지 않고, 타입 정보를 제공하고자 할 때 기초 클래스로 사용하면 된다.

### TInterfacedObject 클래스를 이용한 예제의 작성

그림 앞에서의 설명을 바탕으로 2 개의 인터페이스를 지원하는 간단한 객체를 하나 제작해 보자. 먼저 ILocation, IDimension 이라는 인터페이스를 type 선언문에 다음과 같이 선언한다. 이 인터페이스 들은 그림을 그리는 객체의 위치와 크기를 각각 지정하는 역할을 한다.

```
ILocation = interface
    function GetLeft: Integer;
    procedure SetLeft(Value: Integer);
    function GetTop: Integer;
```



```

    procedure SetTop( Value: Integer );
    property Left: Integer read GetLeft write SetLeft;
    property Top: Integer read GetTop write SetTop;
end;

```

```

IDimension = interface
    function GetWidth: Integer;
    procedure SetWidth(Value: Integer);
    function GetHeight: Integer;
    procedure SetHeight(Value: Integer);
    property Width: Integer read GetWidth write SetWidth;
    property Height: Integer read GetHeight write SetHeight;
end;

```

그리고, 이 인터페이스를 구현할 객체를 TInterfacedObject 클래스에서 상속받아 다음과 같이 선언한다.

```

TWidget = class(TInterfacedObject, ILocation, IDimension)
private
    FLeft: Integer;
    FTop: Integer;
    FWidth: Integer;
    FHeight: Integer;
public
    constructor Create;
    function GetLeft: Integer;
    procedure SetLeft(Value: Integer);
    function GetTop: Integer;
    procedure SetTop(Value: Integer);
    function GetWidth: Integer;
    procedure SetWidth(Value: Integer);
    function GetHeight: Integer;
    procedure SetHeight(Value: Integer);
    procedure Draw(Canvas: TCanvas);
end;

```

이렇게 선언된 TWidget 컴포넌트에서 ILocation, IDimension 과 공통적인 메소드는 이들 인터페이스를 실제로 구현할 때 사용된다. Draw 메소드는 이들 인터페이스의 메소드가 아니라 TWidget 의 메소드로 실제 도형을 그리는 역할을 한다.

그러면, TWidget 클래스를 다음과 같이 구현하도록 한다.

```
constructor TWidget.Create:
```

```
begin
```

```
    inherited Create;
```

```
    FLeft := 0;
```

```
    FTop := 0;
```

```
    FWidth := 100;
```

```
    FHeight := 100;
```

```
end;
```

```
function TWidget.GetLeft: Integer;
```

```
begin
```

```
    Result := FLeft;
```

```
end;
```

```
procedure TWidget.SetLeft(Value: Integer);
```

```
begin
```

```
    FLeft := Value;
```

```
end;
```

```
function TWidget.GetTop: Integer;
```

```
begin
```

```
    Result := FTop;
```

```
end;
```

```
procedure TWidget.SetTop(Value: Integer);
```

```
begin
```

```
    FTop := Value;
```

```
end;
```

```
function TWidget.GetWidth: Integer;
```

```
begin
```

```
    Result := FWidth;  
end;
```

```
procedure TWidget.SetWidth(Value: Integer);  
begin  
    FWidth := Value;  
end;
```

```
function TWidget.GetHeight: Integer;  
begin  
    Result := FHeight;  
end;
```

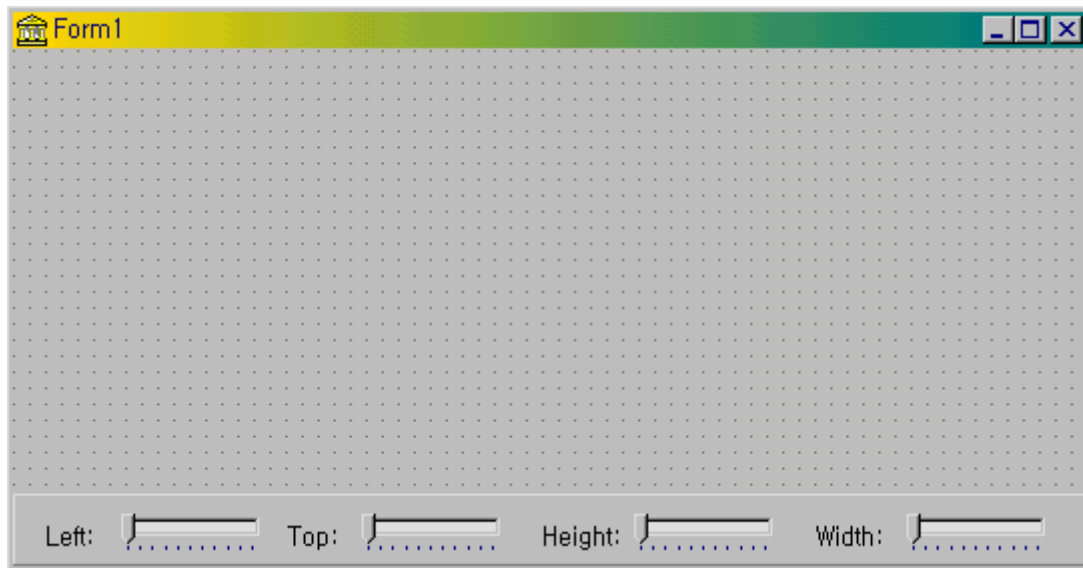
... (중략)

```
procedure TWidget.Draw(Canvas: TCanvas);  
begin  
    with Canvas do  
    begin  
        Brush.Color := clBlue;  
        Pen.Color := clBlack;  
        Rectangle( FLeft, FTop, FLeft + FWidth, FTop + FHeight );  
    end;  
end;
```

그다지 어려운 내용이 아니므로, 설명은 생략하겠다. 이제 이들 인터페이스와 클래스 객체를 사용하려면 이들을 담을 변수를 선언해야 한다. 폼의 public 섹션에 다음과 같이 변수를 선언하도록 한다.

```
AWidget: TWidget;  
LocRef: ILocation;  
DimRef: IDimension;
```

이제 준비는 모두 끝났다. 폼에 패널을 하나 넣고 alBottom 으로 Align 프로퍼티를 설정하고, Caption 을 없앤 뒤, 그 위에 TLabel 과 TTrackBar 컴포넌트를 각각 4 개씩 넣고 다음과 같이 디자인하도록 한다.



이들 트랙바는 각각 폭을 80, 높이는 20, LineSize 프로퍼티는 5 정도로 설정한다. 이들을 조정하면 파란색 바탕, 검은색 라인의 사각형이 폼에 그려지도록 TWidget 클래스와 2 개의 인터페이스를 활용할 것이다. Form 의 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    AWidget := TWidget.Create;
    LocRef := AWidget;
    DimRef := AWidget;
    TrackBar1.Max := Width;
    TrackBar2.Max := Height - Panel1.Height;
    TrackBar3.Max := Width - TrackBar1.Position;
    TrackBar4.Max := Height - TrackBar2.Position - Panel1.Height;
    TrackBar1.Position := 0;
    TrackBar2.Position := 0;
    TrackBar3.Position := 100;
    TrackBar4.Position := 100;
end;
```

여기서, TWidget 클래스를 생성하여 AWidget 변수에 대입하고 이 객체를 ILocation, IDimension 변수인 LocRef, DimRef 에 대입할 수 있다. 이는 AWidget 객체가 ILocation, IDimension 인터페이스를 구현하고 있기 때문에 가능하다. 그 뒤의 코드는 4 개의 트랙 바

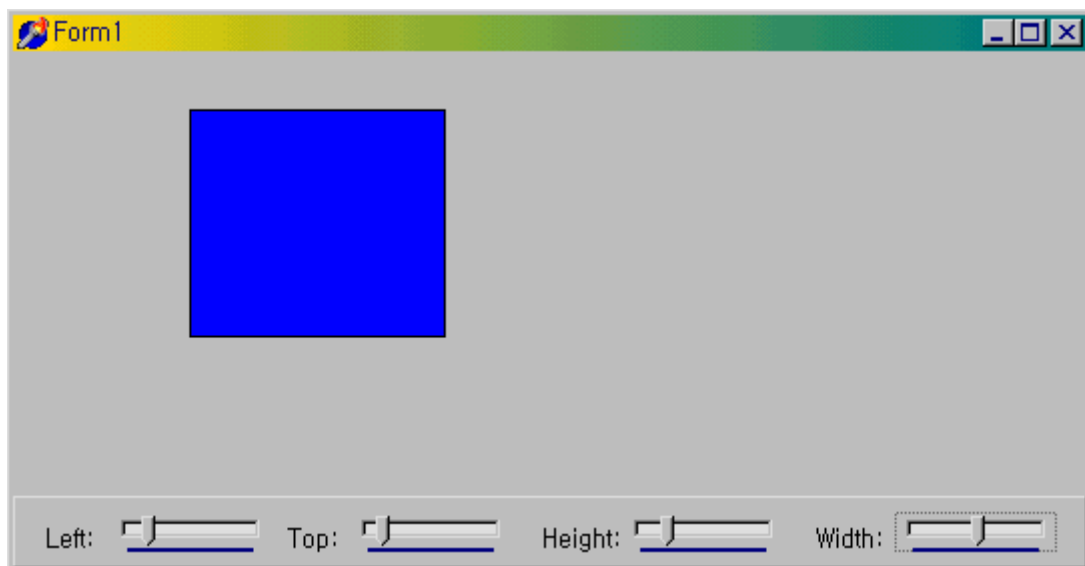
에 대한 초기값을 폼의 크기에 맞추어 설정하는 내용이다.

이제 OnPaint 이벤트 핸들러에서 작성하고, 4 개의 트랙 바를 선택하고 이들 모두의 OnChange 이벤트 핸들러를 작성한다.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
    AWidget.Draw(Canvas);
end;

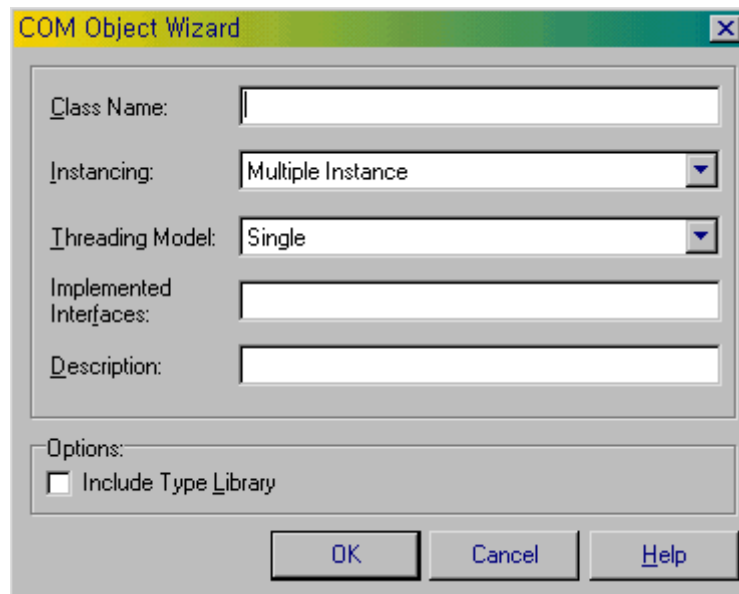
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
    LocRef.Left := TrackBar1.Position;
    LocRef.Top := TrackBar2.Position;
    DimRef.Height := TrackBar3.Position;
    DimRef.Width := TrackBar4.Position;
    Invalidate;
end;
```

이 어플리케이션을 실행하면 다음과 같이 크기와 위치를 트랙 바를 가지고 마음대로 조절할 수 있는 파란 사각형을 볼 수 있을 것이다.



COM 객체 위저드의 사용

텔파이는 여러가지 COM 객체를 생성하는데 도움을 주는 위저드를 많이 제공한다. 이 중에서 COM 객체 위저드를 이용하면 간단하고, 작은 COM 객체를 쉽게 만들 수 있다. COM 객체를 생성하기 위해서는 먼저 File|New 메뉴를 선택하여 객체 저장소를 띄운뒤, ActiveX 탭을 선택하고 여기에서 COM Object 아이콘을 더블 클릭하면 다음과 같은 대화 상자가 나타날 것이다.



COM 객체는 in-process, out-of-process 또는 원격 서버로 구현할 수 있다. COM 객체 위저드는 다음과 같은 작업을 해 준다.

1. 새로운 유닛을 생성한다.
2. TComObject 에서 상속받은 새로운 클래스를 정의하고, 클래스 팩토리 constructor 를 설정한다.

앞의 대화 상자에서 클래스 이름과 구현할 인터페이스를 적어주면 되지만, 그 밖에도 인스턴싱 타입과 쓰레딩 모델을 선택하여야 한다. 이들의 의미는 다음과 같다.

- COM 객체 인스턴싱 타입 (instancing type)

COM 객체가 in-process 서버로만 사용된다면 인스턴싱은 무시된다. 다른 경우에 COM 어플리케이션이 새로운 COM 객체를 생성하게 되면, 다음의 인스턴싱 타입 중 하나를 가질 수 있다.

인스턴싱	의 미
------	-----

Multiple	여러 개의 어플리케이션들이 객체에 접속할 수 있다. 클라이언트가 서비스를 요구하면, 독립된 서버의 인스턴스가 생성된다.
Single	어플리케이션이 객체에 접속하면, 다른 어플리케이션은 접속할 수 없다. 이 옵션은 SDI 어플리케이션에서 자주 사용되는데, 클라이언트가 단일 인스턴스 객체에서 서비스를 요구하면, 모든 요구가 같은 서버에 의해 다루어진다. 예를 들어, 사용자가 워드 프로세서에서 새로운 문서를 열면, 이 문서는 어플리케이션 프로세스와 같은 프로세스에서 열린다.
Internal	이 옵션으로 지정된 객체는 내부적으로 생성된다. 외부의 어플리케이션은 객체의 인스턴스를 직접 생성할 수 없다.

### ● 쓰레딩 모델의 선택

위저드에서 객체를 생성할 때에 객체가 지원하게 될 쓰레딩 모델을 지정해야 한다. 어플리케이션의 클라이언트와 서버 양측에서 COM 을 초기화할 때 쓰레드를 이용하는 규칙을 지정하는데, COM 은 이를 비교하여 같은 규칙을 지원하면 COM 은 양측을 직접 연결하고 규칙을 지킬 것이라고 신뢰할 수 있다. 만약 서로의 규칙이 다르면 COM 은 마샬링을 설정하여 서로가 자신의 규칙을 지킬 수 있도록 해준다. 물론 마샬링은 수행 속도를 감소시키는 것이 단점이다.

위저드에서 선택하게 되는 쓰레딩 모델은 레지스트리에 기록되는 정보이다. 그러므로, 개발자는 쓰레딩 모델에 걸맞게 객체를 구현하여야 한다. 쓰레딩 모델은 in-process 서버에서만 유효하다. Out-of-process 서버는 EXE 로 등록되며, 쓰레딩 모델은 자신이 직접 구현해야 한다.

선택할 수 있는 쓰레딩 모델로는 다음과 같은 것들이 있다.

쓰레딩 모델	설 명
Single	쓰레드를 지원하지 않는다. 클라이언트의 요구는 호출 이전에 의해 나열되고, 클라이언트는 한번에 하나씩 처리하므로 쓰레드 지원이 필요하지 않다. 대신 수행 성능의 향상은 없다.
Apartment (Single-thread apartment)	클라이언트가 객체의 메소드를 객체가 생성된 쓰레드에서만 호출할 수 있는 모델이다. 같은 서버의 서로 다른 객체 들은 서로 다른 쓰레드에서 호출될 수 있으나, 각각의 객체 자체는 하나의 쓰레드에서만 호출될 수 있다. 각 인스턴스의 데이터는 안전하고, 전역 데이터는 반드시 임계 섹션이나 다른 동기화 기법을 이용하여 보호해야 한다. 약간의 수행 성능 향상이 있으며, 객체도 작성하기 쉬운 장점이 있다. 웹 브라우저 컨트롤로 가장 많이 사용되는 모델이다.
Free (Multi-thread apartment)	클라이언트가 어느 때, 어느 쓰레드에서든 객체의 메소드를 호출할 수 있는 모델이다. 객체는 반드시 모든 인스턴스와 전역 데이터를 임계 섹션이나 다른 동기화 기법을 이용하여 보호해야 한다. 클라이언트는 쉽게 작성할 수 있지만, 서버

	객체는 구현하기 어렵다. 분산 DCOM 환경에서 많이 사용되는 모델이다.
Both	객체가 클라이언트를 지원할 때 apartment 와 free 쓰레딩 모델을 모두 지원하는 모델이다. 최고의 수행 능력과 유연함을 가지게 된다.

참고로 지역 변수는 쓰레딩 모델에 관계 없이 언제나 안전하다. 이는 지역 변수가 각 쓰레드의 스택에 변수의 값을 저장하기 때문이다.

#### 1. Free 쓰레딩 모델을 지원하는 객체의 제작

거의 언제나 하나 이상의 쓰레드에서 객체에 접근해야 하는 경우에는 Free 쓰레딩 모델을 사용하는 것이 좋다. 가장 흔한 예로는 원격 기계에 있는 객체에 접속하는 클라이언트 어플리케이션이다. 원격 클라이언트가 객체의 메소드를 호출하면, 서버는 서버 기계의 쓰레드 pool 에서 쓰레드의 호출을 받게 된다. 이 쓰레드는 실제 객체에 로컬로 호출하지만, 객체가 free 쓰레딩 모델을 지원하기 때문에 쓰레드는 객체를 직접 호출할 수 있다.

만약 객체가 apartment 쓰레딩 모델을 지원하면 호출은 객체를 생성한 쓰레드로 이전되어야 하며, 결과가 클라이언트로 반환되기 전에 호출을 받은 쓰레드로 전송해야 한다. 이런 작업을 위해서는 마샬링이 필요하다.

Free 쓰레딩을 지원하려면, 개발자는 반드시 각각의 메소드에 의해 인스턴스의 데이터가 어떻게 접근되는지 고려해야 한다. 만약 메소드가 인스턴스 데이터를 기록한다면, 개발자는 임계 섹션(critical section) 등의 여러가지 동기화 기법을 이용하여 인스턴스 데이터를 보호해야 한다. 그래도, 이런 데이터 보호 기법이 COM 의 마샬링 보다는 훨씬 효율적이다. 만약 데이터가 읽기 전용이면 이런 문제는 생기지 않는다.

#### 2. Apartment 쓰레딩 모델을 지원하는 객체의 제작

Apartment 쓰레딩 모델을 구현하려면 다음의 몇가지 규칙을 따라야 한다.

- 어플리케이션의 첫번째 쓰레드는 COM 의 메인 쓰레드가 된다. 또한, 이 쓰레드는 COM 을 uninitialize 하는 마지막 쓰레드가 된다.
- Apartment 쓰레딩 모델의 각각의 쓰레드는 반드시 메시지 루프를 가져야 하며, 메시지 큐를 자주 검사해야 한다.
- 쓰레드가 COM 인터페이스에 대한 포인터를 얻으면, 인터페이스의 메소드는 그 쓰레드에서만 호출할 수 있다.

이 쓰레딩 모델은 쓰레딩을 지원하지 않는 모델과 Free 쓰레딩 모델의 중간 정도의 모델이므로, 서버는 전역 데이터는 보호해야 하지만 객체의 인스턴스 데이터는 언제나 같은 쓰레



드의 메소드에 의해 호출되므로 안전하다.

보통 웹 브라우저의 컨트롤에 이 모델이 많이 사용되는데, 이는 브라우저 어플리케이션이 각 쓰레드를 apartment 로 초기화하기 때문이다.

## COM 객체의 활용

그러면 이번에는 COM 객체 위저드를 이용해서 COM 객체를 생성해서, 이를 등록하고 클라이언트 어플리케이션에서 COM 객체를 사용하는 방법에 대해서 알아보자.

### ● COM 객체 서버의 제작

먼저 COM 객체를 구현할 프로젝트 파일을 만들어야 한다. 여기서는 DLL 의 형태로 만들 것이다. 일단 File|New 메뉴를 선택하고 여기서 DLL 아이콘을 더블 클릭해서 DLL 프로젝트 파일을 하나 생성한다. 물론, 여기에서 ActiveX 탭에 있는 ActiveX Library 를 더블 클릭하면 더욱 쉽게 구현이 가능하지만 프로젝트 파일을 일반적인 DLL 로 선택한 것은 기초적인 구현 방법을 설명하기 위한 것임을 미리 밝혀 둔다. 그리고, 다시 File|New 메뉴를 선택한 후 객체 저장소에서 ActiveX 탭을 클릭하고 Com Object 아이콘을 더블 클릭하여 앞에서 설명한 구현할 TComObject 의 이름과, 지원할 인터페이스, 인스턴싱 type, 쓰레딩 모델 등을 대화 상자에 적어 넣는다. 인스턴싱 type 과 쓰레딩 모델은 디폴트 값인 Single, Multiple Instance 로 선택한다. 참고로 이들의 종류를 결정할 때에는 앞에서 설명한 대로 COM 객체의 용도에 따라서 결정해야 하는 것임을 명심하도록 하자. 참고로 Description 부분에 기록하는 내용은 나중에 액티브 X 서버에 대한 정보를 레지스트리에 기록하고, 이를 조회할 때 나타나는 설명이 되므로 적당한 설명을 적어넣는 것도 좋을 것이다. 여기서는 COM 객체의 이름으로는 Math, 구현할 인터페이스로는 IMath 라고 적어넣고 OK 버튼을 클릭하면 다음과 같은 뼈대 코드가 생성될 것이다.

```
unit Unit2;
```

```
interface
```

```
uses
```

```
    Windows, ActiveX, ComObj;
```

```
type
```

```
    TMath = class(TComObject, IMath)
```

```
    protected
```

```

    {Declare IMath methods here}
end;

const
    Class_Math: TGUID = '{ED309D88-2732-11D2-9774-0000E838052E}';

implementation

uses ComServ;

initialization
    TComObjectFactory.Create(ComServer, TMath, Class_Math,
        'Math', '', ciMultiInstance, tmSingle);
end.

```

이렇게 COM 객체 위저드는 해당되는 클래스 이름과 인터페이스 이름에 대한 기본적인 선언과 클래스 팩토리의 생성 코드를 자동으로 만들어 준다. initialization 섹션의 클래스 팩토리 생성 코드가 COM 객체를 생성하는 핵심이 된다.

그러면, 실제로 사용할 만한 COM 객체를 생성해보자. IMath 인터페이스는 델파이의 Math.pas 유닛에서 지원되지 않는 몇가지 중요한 수학 함수를 선언하고, 이를 델파이 외에 다른 언어에서도 사용할 수도 있게할 것이다. 앞에서 설명한 TInterfaced 객체에서의 인터페이스와는 달리 COM 객체에서 사용하는 인터페이스에는 반드시 GUID 형식의 IID 를 가져야 한다. 그러므로, 먼저 interface 이름을 적어 넣고 Shift+Ctrl+G 키를 누르면 GUID 형식의 IID 가 추가된다. 그리고, 인터페이스의 메소드를 다음과 같이 선언한다.

```

IMath = interface
    ['{ED309D86-2732-11D2-9774-0000E838052E}']
    function Distance (const X1, Y1, X2, Y2: Extended): Extended; stdcall;
    procedure Polar2XY (const Rho, Theta: Extended; var X, Y: Extended); stdcall;
    procedure XY2Polar (const X, Y: Extended; var Rho, Theta: Extended); stdcall;
    function FactorialX (A: Cardinal): Extended; stdcall;
end;

```

간단히 메소드를 설명하면 Distance 메소드는 두개의 좌표의 거리를 계산하는 함수이며, Polar2XY 와 XY2Polar 메소드는 극좌표계와 XY 좌표계의 좌표를 전환하여 계산하는 함수이다. FactorialX 는 잘 아는 Factorial 함수이다 ( $n! = 1*2*...*(n-1)*n$ ).

이 인터페이스를 구현하기 위해 TMath 클래스를 다음과 같이 변경하고 이들을 구현한다.

```
TMath = class(TComObject, IMath)
private
    function Sgn (const X: Extended): ShortInt;
public
    function Distance (const X1, Y1, X2, Y2: Extended): Extended; stdcall;
    procedure Polar2XY (const Rho, Theta: Extended; var X, Y: Extended); stdcall;
    procedure XY2Polar (const X, Y: Extended; var Rho, Theta: Extended); stdcall;
    function FactorialX (A: Cardinal): Extended; stdcall;
end;
```

여기서 Sgn 메소드는 극좌표 계산을 할 때 사용되는 유틸리티 함수이다.

이들을 다음과 같이 구현한다. 구현 방법 자체는 수학적인 내용이므로 설명을 생략하겠다.

```
function TMath.Sgn(const X: Extended): ShortInt;
begin
    if X < 0.0 then Result := -1
    else if X = 0.0 then Result := 0
    else Result := 1
end;
```

```
function TMath.Distance(const X1, Y1, X2, Y2: Extended): Extended;
var
    X, Y: Extended;
begin
    X := Abs (X1 - X2);
    Y := Abs (Y1 - Y2);
    if X > Y then      Result := X * Sqrt (1 + Sqr (Y / X))
    else if Y <> 0 then Result := Y * Sqrt (1 + Sqr (X / Y))
    else Result := 0
end;
```

```
function TMath.FactorialX(A: Cardinal): Extended;
var
    I: Integer;
```

```

begin
  if A > 1547 then
    begin
      Result := 0.0;
      Exit;
    end;
    Result := 1.0;
    for I := 2 to A do Result := Result * I;
  end;

procedure TMath.Polar2XY(const Rho, Theta: Extended; var X, Y: Extended);
begin
  SinCos (Theta, X, Y);
  X := Rho * X;
  Y := Rho * Y;
end;

procedure TMath.XY2Polar(const X, Y: Extended; var Rho, Theta: Extended);
begin
  Rho := Sqrt (Sqr (X) + Sqr (Y));
  if Abs (X) > 0.00001 then Theta := ArcTan (Y / X)
  else Theta := Sgn (Y) * Pi / 2.0;
end;

```

기본적인 구현은 거의 끝난 셈이다. 이제 File|Save All 메뉴를 선택하여 유닛 파일과 프로젝트 파일을 각각 적당한 이름으로 저장하도록 한다. 이 책의 부록으로 제공되는 CD-ROM 에서는 U\_ExamSvr1.pas, ExamSvr1.dpr 로 저장하였다. 마지막으로, 프로젝트 파일을 다음과 같이 수정한다.

```

library ExamSvr1;

uses
  ComServ,
  U_ExamSvr1 in 'U_ExamSvr1.pas';

exports

```

```
DllGetClassObject, DllCanUnloadNow,  
DllRegisterServer, DllUnregisterServer;  
end.
```

이 코드의 의미는 기본적인 구현 부분은 U\_ExamSvr1 에 위치하고 있으며, 프로젝트 파일에서는 COM 서버를 등록할 때 필요한 4 개의 함수를 export 하게 된다. 이들은 모두 ComServ.pas 유닛에 선언되어 있으므로 uses 절에 ComServ.pas 유닛을 추가한 것이다.

DllGetClassObject 는 액티브 X 객체에 대한 클래스 팩토리를 얻으려고 할 때, OLE 엔진이 액티브 X 서버 DLL 을 메모리에 불러들인 후 호출하는 함수이다 .

DllRegisterServer 는 in-process 액티브 X 서버(DLL)에 의해 export 되는 함수로 타입 라이브러리와 서버 모듈에서 지원하는 모든 클래스에 대한 레지스트리 엔트리를 생성한다.

DllUnregisterServer 는 일반적으로 액티브 X 서버가 사용자의 시스템에 인스톨될 때 호출된다. DllUnregisterServer 은 in-process 자동화 서버(DLL)에 의해 export 되며, 서버가 언인스톨될 때 DLL 이 DllRegisterServer 에 의해 등록된 엔트리를 제거하는 역할을 한다.

DllCanUnloadNow 는 액티브 X 서버(DLL)에 의해 export 되며, Ole 엔진에 의해 호출되어 서버가 더 이상 사용되지 않을 때 메모리에서 언로드할 수 있는지 알아낸다.

이들 4 개의 함수는 액티브 X/COM 서버가 설치될 때에는 기본적으로 export 되어야 하는 것들이다. 이런 작업을 프로젝트 파일을 선택할 때 표준 DLL 대신 ActiveX Library 를 선택하면 하지 않아도 된다.

이제 몇 가지 수학 함수를 지원하는 COM 객체가 완성되었다. 컴파일을 하고, COM 서버를 등록하면 다른 어플리케이션에서 이를 사용할 수 있게 된다.

등록하는 방법은 델파이를 사용하고 있다면 Run|Register ActiveX Server 명령을 선택하면 즉시 등록된다. 델파이가 깔려 있지 않은 컴퓨터에 ExamSvr1.DLL 파일을 액티브 X 서버로 등록하려면, 마이크로소프트의 REGSVR32.EXE 파일을 이용하여 직접 등록하거나 확장자가 REG 인 파일을 텍스트 파일로 작성하여 이를 실행하도록 하면 자동으로 등록되게 할 수 있다. REG 파일을 작성하는 방법과 REGSVR32.EXE 파일의 사용 방법에 대한 것은 MSDN 이나 마이크로소프트의 웹 사이트를 찾아보면 자세한 내용이 있으므로 이를 참고하기 바란다. 그 밖에, 런타임에서 코드로 작성한 액티브 X 서버 DLL 을 등록할 수 있다. 이 방법은 클라이언트 어플리케이션을 작성하는 부분에서 다루게 될 것이므로 이를 참고하기 바란다.

## ● COM 객체 클라이언트의 제작

이제 등록된 COM 객체를 이용하여 이를 활용하는 클라이언트 어플리케이션을 만들어 보자. 일단 새로운 어플리케이션을 시작하고 폼을 다음과 같이 디자인한다. 우리가 사용할 IMath 인터페이스의 Distance, Polar2XY, XY2Polar, FactorialX 등의 계산을 마음대로 할 수 있도록

록 8 개의 TEdit 컴포넌트와 1 개의 TSpinEdit 컴포넌트를 배치하고, 계산을 실행할 때 사용할 버튼도 하나 올려 넣도록 한다. 그리고, 각 컴포넌트의 역할을 설명하기 위해 여러 개의 라벨을 올려 놓고, 결과를 보여주기 위해 라벨 컴포넌트를 위치시킨 뒤 Caption 을 지우도록 한다.

폼의 디자인이 끝났으면, IMath 인터페이스를 사용할 수 있도록 하기 위해 인터페이스 선언부와 클래스 ID 부분을 COM 객체 서버의 소스에서 복사해서 type 선언부에 추가하도록 한다. 그리고, IMath 인터페이스를 저장할 전역 변수를 하나 선언한다.

type

```
IMath = interface
    ['{ED309D86-2732-11D2-9774-0000E838052E}']
    ...
end;
```

var

```
Form1: TForm1;
AMath: IMath;
```

const

```
Class_Math: TGUID = '{ED309D85-2732-11D2-9774-0000E838052E}';
```

uses 절에 ComObj.pas 유닛만 추가하면, COM 객체를 사용할 준비는 모두 끝난 셈이다. 먼저 폼의 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
    AMath := CreateComObject(Class_Math) as IMath;
end;
```

CreateComObject 의 파라미터로는 COM 객체의 CLSID 가 사용된다. 그렇기 때문에, const 절의 내용을 복사해와서 사용한 것이다. 또한 생성된 COM 객체의 인터페이스를 사용하기 위해서 as 연산자를 이용하여 IMath 인터페이스를 AMath 변수에 저장한다.

그런데, 앞에서도 간단히 설명한 바 있지만 액티브 X 서버 DLL 이 등록되어 있지 않다면 여기에서 클래스가 등록되지 않았다는 에러 메시지를 보게 될 것이다. 물론, 액티브 X 서버 DLL 을 컴파일하고 델파이에서 Run|Register ActiveX Server DLL 명령을 선택해서 레지스트리에 등록했거나, REGSVR.EXE 프로그램을 사용하여 등록했다면 문제가 없겠지만 어플리케이션을 배포할 때 이를 모두 처리해주는 건 쉽지 않다. 물론 InstallShield 와 같은 설치 프로그램의 최근 버전에는 이들을 처리할 수 있는 내용이 있지만, 이보다 더 간단하게 처리할 수 있는 방법이 있으면 좋을 것이다.

이를 위해서 런타임에서 직접 등록해서 사용할 수 있도록 코드를 수정하도록 하자. 먼저 uses 절에 ActiveX.pas 유닛을 추가하고, 다음과 같이 OnCreate 이벤트 핸들러를 다시 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
    OCXHandle: THandle;
    RegFunc: TDllRegisterServer;
begin
    try
        AMath := CreateComObject(Class_Math) as IMath;
    except
        OCXHandle
            := LoadLibrary(PChar(ExtractFilePath(Application.ExeName) + 'WExamSvr1.Dll'));
        RegFunc := GetProcAddress(OCXHandle, 'DllRegisterServer');
        if RegFunc > 0 then ShowMessage('등록에 실패했습니다 !')
        else AMath := CreateComObject(Class_Math) as IMath;
        FreeLibrary(OCXHandle);
    end;
end;
```

우선 CreateComObject 메소드를 이용해서 COM 객체의 생성을 시도한다. 이때 레지스트

리에 COM 객체가 등록되어 있지 않았다면, 예외가 발생할 것이다. 그러므로, try...except 구문을 이용하여 우리가 사용하고자 하는 액티브 X 서버 DLL 을 직접 등록한다.

등록할 때에는 액티브 X 서버 DLL 에 구현된 DllRegisterServer 함수의 주소를 이용하게 된다. 즉, TDllRegisterServer 형(ActiveX.pas 유닛에 정의되어 있다)의 변수를 선언하고 이 변수에 DllRegisterServer 함수의 주소를 얻어온 뒤에 COM 객체를 생성하면 된다.

이제 AMath 변수를 이용해서 IMath 인터페이스의 메소드를 호출할 수 있다.

그러면, Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    X1, Y1, X2, Y2: Extended;
begin
    X1 := StrToFloat(Edit1.Text);
    Y1 := StrToFloat(Edit2.Text);
    X2 := StrToFloat(Edit3.Text);
    Y2 := StrToFloat(Edit4.Text);
    Label4.Caption := FloatToStr(AMath.Distance(X1, Y1, X2, Y2));
    AMath.XY2Polar(StrToFloat(Edit5.Text), StrToFloat(Edit6.Text), X1, Y1);
    Label8.Caption := FloatToStr(X1) + ', ' + FloatToStr(Y1);
    AMath.Polar2XY(StrToFloat(Edit7.Text), StrToFloat(Edit8.Text), X1, Y1);
    Label10.Caption := FloatToStr(X1) + ', ' + FloatToStr(Y1);
    Label12.Caption := FloatToStr(AMath.FactorialX(SpinEdit1.Value));
end;
```

내용이 평이하므로, 자세한 설명은 생략하도록 한다. 이것으로 클라이언트 프로그램이 완성되었다. 프로그램을 실행하고 에디트 박스와 SpinEdit 박스에 적당한 값을 입력하고 ‘계산’ 버튼을 클릭하면 다음과 같은 계산 결과를 볼 수 있을 것이다.



- Initialize, Destory 메소드

TComObject 클래스의 constructor 는 가상 함수가 아니다. 그렇지만, constructor 는 항상 가상 메소드인 Initialize 메소드를 호출한다. 그러므로, 객체를 생성할 때 내부 필드 값을 가지고 있고, 그 값을 초기화하는 등의 작업이 필요한 경우에는 Initialize 메소드를 오버라이드해서 사용하면 된다. Destroy 함수는 가상 함수이기 때문에 직접 오버라이드가 가능하다. 여기서는 COM 객체를 구현하기 위해 사용한 여러가지 리소스 등을 해제하는 코드 등을 추가하게 된다. 간단하게 사용방법을 소개하면 다음과 같다.

```
TSampleObject(TComObject, ISampleInterface)
```

```
private
```

```
    FSample: string;
```

```
public
```

```
    function GetSample: string; stdcall;
```

```
    procedure SetSample(Value: string); stdcall;
```

```
    procedure Initialize; override;
```

```
    destructor Destroy; override;
```

```
end;
```

```
...
```

```
procedure TSampleObject.Initialize;
```

```
begin
```

```
    inherited;
```

```
    FSample := 'Default Value';
```

```
end;
```

```
procedure TSampleObject.Destroy;
```

```
begin
```

```
    inherited;
```

```
    ShowMessage('Object Destroyed !');
```

```
end;
```

## 정 리 (Summary)

이번 장에서는 COM 을 활용할 때 가장 기본이라고 할 수 있는 인터페이스의 사용 방법과 가장 원시적인 TInterfacedObject 클래스를 사용한 간단한 예제와 COM 객체 위저드를 이용하여 COM 객체 서버를 제작하고, 이렇게 제작된 서버를 사용하는 클라이언트 프로그램을 만들어서 활용하는 방법을 익혀 보았다.

COM 객체 서버는 기본적인 제작 방법만 알면 제작이 쉽기 때문에, 델파이 VCL 소스 코드로 된 여러가지 내용을 COM 객체 서버로 제작해서 DLL 로 만든 뒤에 이를 등록하여 비주얼 베이직이나 비주얼 C++ 과 같은 다른 언어에서 그대로 활용하게 할 수 있으므로 장점이 무척 많다. 자칫 수박 겉핥기처럼 읽고만 넘어갈 수도 있는 내용이지만, 위저드를 이용해서 액티브 X 컨트롤을 만들어 사용하는 것보다 훨씬 강력하고도 활용도가 높을 수 있는 내용이므로 반드시 자신의 것으로 만들고 넘어가기 바란다.

다음 장에서는 TOleContainer 컴포넌트를 이용하여 액티브 도큐먼트를 활용하는 방법과 OLE 자동화 컨트롤러를 만들어서 DAO, 엑셀 등을 직접 제어하는 방법, 마지막으로 OLE 자동화 서버를 제작하는 방법에 대해서 알아볼 것이다.

## OLE 도큐먼트와 OLE 자동화 컨트롤러

### (OLE Document and OLE Automation Controller)

마이크로소프트에서 처음 OLE 를 발표했을 때 OLE 는 DDE(Dynamic Data Exchange) 모델의 확장판이었다. 즉, 클립보드를 사용하여 데이터를 복사하고 DDE 를 이용하여 두 문서를 연결할 수 있었던 것처럼, OLE 를 이용하여 데이터를 서버 어플리케이션에서 클라이언트 어플리케이션 쪽으로 서버 관련 정보 또는 윈도우 레지스트리에 저장된 정보를 레퍼런스와 함께 복사할 수 있었다.

그러나, 그 이후 발표된 OLE2 는 과거의 OLE 와 같이 복합 문서를 작성하기 위한 도구로서의 역할을 하는 것이 아니라, COM 이라는 기반 기술을 가리키는 단어가 되었다.

여기에서 과거의 OLE 에 해당되는 내용을 OLE 도큐먼트 또는 액티브 도큐먼트라고 하며, OLE2 를 필두로한 최근의 핵심기술은 DCOM/ActiveX 라는 단어로 표현하고 있다.

이번 장에서는 OLE 도큐먼트를 다루기 위한 TOleContainer 클래스의 사용법과 OLE 자동화 컨트롤러를 작성하는 방법에 대해서 알아볼 것이다.

### OLE 도큐먼트와 TOleContainer 컴포넌트

OLE 도큐먼트는 임베딩(embedding)과 연결(linkning)의 2 가지 기능을 가지고 있다.

OLE 도큐먼트에서 객체를 삽입하는 임베딩은 클립보드를 이용해서 객체를 삽입하는 것과 큰 차이는 없으나, 서버 어플리케이션에서 OLE 객체를 복사해서 이를 컨테이너 어플리케이션에 붙일 때 그 데이터와 서버에 대한 정보(GUID)를 같이 복사하게 된다는 것이다. 여기에 비해 객체를 연결하는 것은 서버에 대한 데이터와 정보의 레퍼런스만을 복사하는 것이다. 그러므로, 객체를 임베딩한 후 이 객체를 수정하면 이것은 독자적인 내용을 수정하는 것이 되지만, 연결된 객체를 수정하면 실제로는 별도의 파일에 있는 원래의 데이터를 수정하는 것이 된다. 또한, 임베딩된 객체의 데이터는 컨테이너 어플리케이션에 의해 저장되고 관리된다. 반면에 연결된 객체는 서버에 의해 독립적으로 다루어지는 물리적으로 별도의 파일 안에 저장됩니다.

어플리케이션에 OLE 2.0 서버로부터 객체를 삽입한 경우에는 컨테이너는 비주얼(in-place) 편집을 지원하는데, 이것은 컨테이너의 메인 윈도우 안에서 그 객체를 수정할 수 있게 해준다. 서버와 컨테이너의 어플리케이션 윈도우, 메뉴, 툴바가 자동으로 병합되고 이를 이용하여 사용자는 하나의 컨테이너 어플리케이션에서 OLE 서버를 지원하는 여러 종류의 객체를 작업할 수 있게 된다.

- TOleContainer 컴포넌트 사용하기

TOleContainer 컴포넌트는 델파이 어플리케이션에 OLE 도큐먼트 컨테이너를 쉽게 구현하도록 지원하는 컴포넌트이다. 사용자가 어플리케이션에 객체를 삽입할 수 있도록 지원하려면 InsertObjectDialog 메소드를 호출할 수도 있고, OLE 객체를 TOleContainer 에 연결시킬 때 CreateObject, CreateLinkToFile 메소드를 이용해서 객체를 생성하거나 연결할 수 있다.

TOleContainer 는 자동으로 메뉴 병합을 지원하기 때문에, 데이터를 in-place 활성화 시키면 컨테이너 폼의 메뉴가 해당 OLE 객체 서버 어플리케이션의 메뉴와 합쳐진다. 마찬가지로, in-place 활성화되는 OLE 서버의 툴바 역시 컨테이너 어플리케이션 윈도우에 나타난다. 이때 툴바로 사용되는 패널에 나타나는데, 이를 막기 위해서는 패널의 Locked 프로퍼티를 True 로 설정해주면 된다.

TOleContainer 클래스의 주요한 메소드와 프로퍼티를 소개하면 다음과 같은 것들이 있다.

이 름	설 명
AllowInPlace	In-place 활성화를 지원하는지 여부
AllowActiveDoc	이 값이 True 이면 OLE 컨테이너의 팝업 메뉴에 OLE 객체의 Verbs 를 포함하게 된다.
AutoActivate	컨테이너 안의 객체를 활성화하는 방법을 지정한다. aaManual, aaGetFocus, aaDoubleClick 등의 값이 있다.
Align	In-place 활성화를 제대로 지원하려면 alClient 로 지정하는 것이 좋다.
CanPaste	클립보드의 데이터를 임베딩 객체로 붙여넣을 수 있는지 지정한다.
CopyOnSave	이 값이 True 이면 OLE 객체를 기록할 때 파일을 이용하며, 중복된 데이터를 압축하므로 공간을 절약할 수 있다.
CreateLinkToFile	OLE 객체의 연결을 물리적 파일로 생성한다.
CreateObject	OLE 컨테이너에 새로운 임베딩 객체를 생성한다.
CreateObjectFromFile	임베딩 객체를 지정된 파일에서 생성한다.
CreateObjectFromInfo	TCreateInfo 레코드의 스펙에 기초하여 객체를 생성한다.
DestroyObject	객체와 변경된 사항을 제거한다.
DoVerb	OLE 객체가 특정 행동을 수행할 것을 요구한다.
Iconic	이 값이 True 이면 아이콘이 컨테이너에 표시되며, False 이면 객체의 데이터가 표시된다.
InsertObjectDialog	운영체제에서 제공하는 OLE 삽입을 지원하는 대화상자를 실행한다.
Linked	이 값이 True 이면 객체가 연결된 것이다.
LoadFromFile	지정된 파일에서 OLE 객체를 읽어온다.
LoadFromStream	스트림에서 OLE 객체를 읽어온다.

Modified	이 값이 True 이면 OLE 객체가 변경된 것이다.
ObjectVerbs	OLE 객체가 지원하는 모든 verbs 의 이름을 스트링 리스트로 반환한다.
OleClassName	OLE 객체의 클래스 이름을 지정한다.
OleObjectInterface	OLE 객체에 대한 IOleObject 인터페이스를 반환한다.
SaveToFile	OLE 객체를 지정된 파일에 저장한다.
SaveToStream	OLE 객체를 스트림에 저장한다.
Copy	클립보드로 컨테이너의 객체를 복사한다.
Paste	클립보드에서 컨테이너로 붙여넣는다.
Run	OLE 객체를 ovRunning 상태로 전환한다.
UpdateObject	OLE 객체의 현재 데이터를 반영하기 위해 소스를 다시 읽어들인다.
UpdateVerbs	OLE 객체의 verbs 리스트를 refresh 한다.
SourceDoc	연결된 OLE 객체에 대한 소스 문서의 이름
State	OLE 객체의 state (osEmpty, osLoaded, osRunning, osOpen, osInPlaceActive, osUIActive)
StorageInterace	OLE 객체의 IStorage 인터페이스

TOLeContainer 컴포넌트는 4 가지 이벤트를 지원한다. OnActivate 이벤트는 OLE 객체가 활성화될 때 발생하며, OnDeactivate 이벤트는 OLE 객체가 비활성화될 때 발생한다.

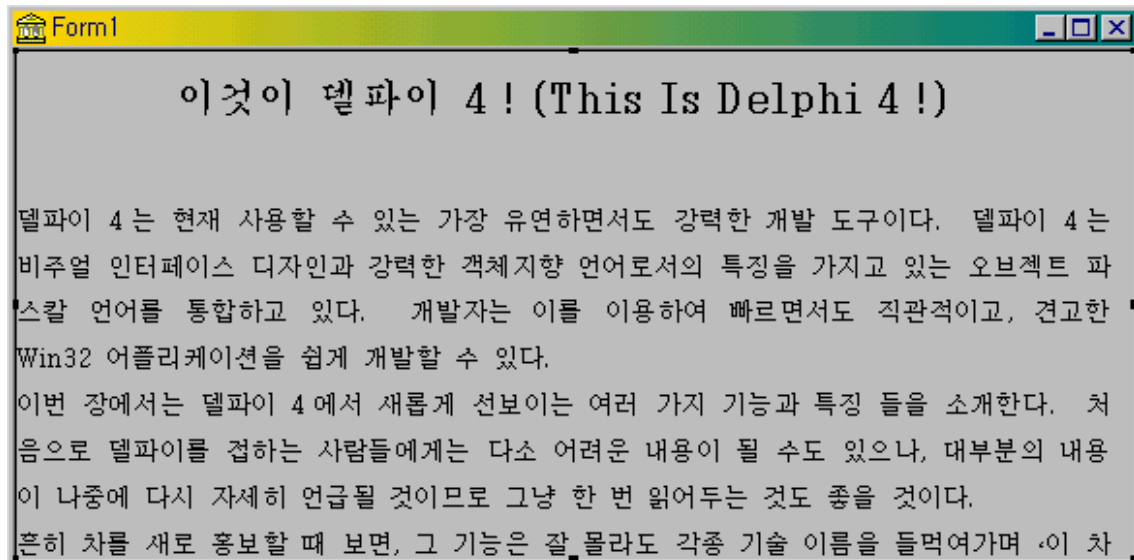
또한, OnObjectMove 이벤트는 OLE 객체의 크기나 위치가 바뀔 때 발생하는 것으로 이벤트의 Bounds 파라미터는 in-place 객체의 사각형 좌표를 지정한다. OnResize 이벤트는 OLE 컨테이너 윈도우의 크기가 변경될 때 발생하는 것으로 SizeMode 프로퍼티의 값이 smAutoSize 일 경우에 컨테이너 윈도우의 크기가 자동으로 변경된다.

간단한 OLE 컨테이너 어플리케이션을 만들기 위해서, 먼저 OleContainer 컴포넌트를 폼에 올려 놓는다. 그리고, 이 컴포넌트를 선택하고 오른쪽 마우스 버튼을 클릭해서 팝업 메뉴를 띄운 후 여기에서 Insert Object 명령을 선택하면 표준 OLE Insert Object 대화 상자가 나타난다. 이 대화 상자에는 레지스트리에 저장된 OLE 서버 어플리케이션이 나열되는데, 여기서 적당한 객체를 선택해서 삽입하면 컨테이너 컴포넌트에 데이터가 표시된다. 이렇게 객체가 컨테이너 안에 삽입되면 새로운 메뉴 항목으로 OLE 객체의 프로퍼티를 변경하거나, 다른 OLE 객체를 삽입하는 메뉴, 객체의 복사 삭제, 해당 객체의 verb(Edit, Open, Play 등)가 메뉴로 추가된다. Verb 란 간단하게 설명하면 OLE 도큐먼트 객체의 특정 기능을 수행하는 명령이라고 생각하면 된다.

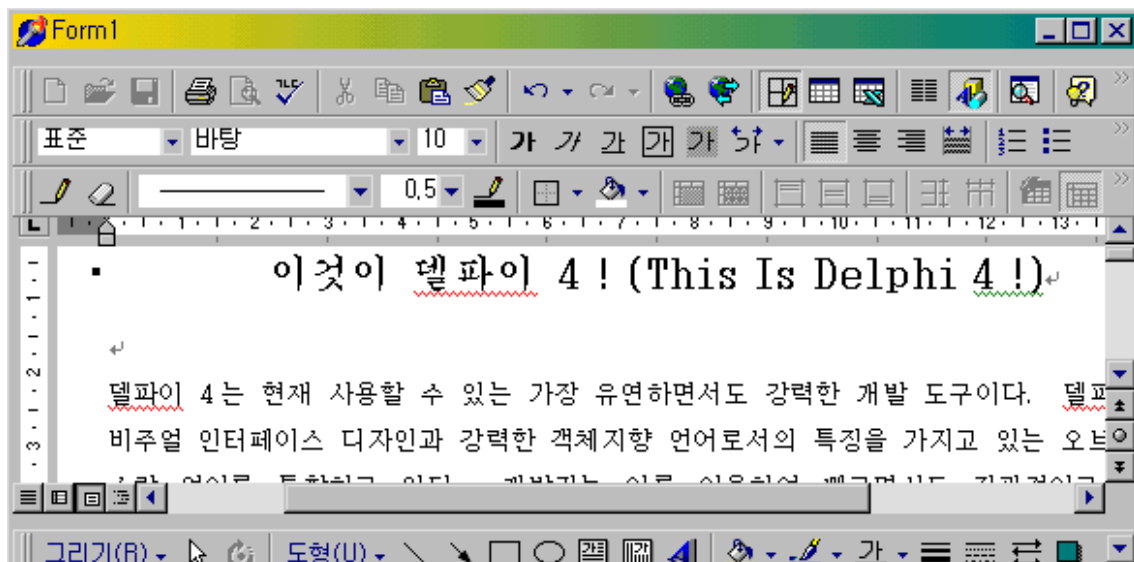
일단 이렇게 OLE 객체를 컨테이너에 추가하면, OLE 서버가 활성화되어 새로운 객체를 편집할 수 있게 된다. 서버 어플리케이션을 종료하면, 델파이가 컨테이너의 객체를 업데이트 해서 표시하게 된다.

간단한 OLE 컨테이너 객체를 제작하는 방법은 매우 간단하다. 그냥 TOLeContainer 컴포넌트를 다음과 같이 올려 놓고, Align 프로퍼티를 alClient 로 설정하기만 하면 된다. 그리

고, 초기에 보여줄 객체를 오른쪽 버튼을 눌러서 Insert Object 메뉴를 선택한 후 적당한 객체를 삽입한다. 여기서는 이 책의 1 장을 선택하였다. MS 워드 문서로 작성되었기 때문에 다음과 같이 삽입된다.



이제 이 예제를 실행하고, 컨테이너 컴포넌트를 더블 클릭하면 워드가 작동하면서 다음과 같이 서버 윈도우가 어플리케이션 내부에서 실행되는 것을 볼 수 있을 것이다.

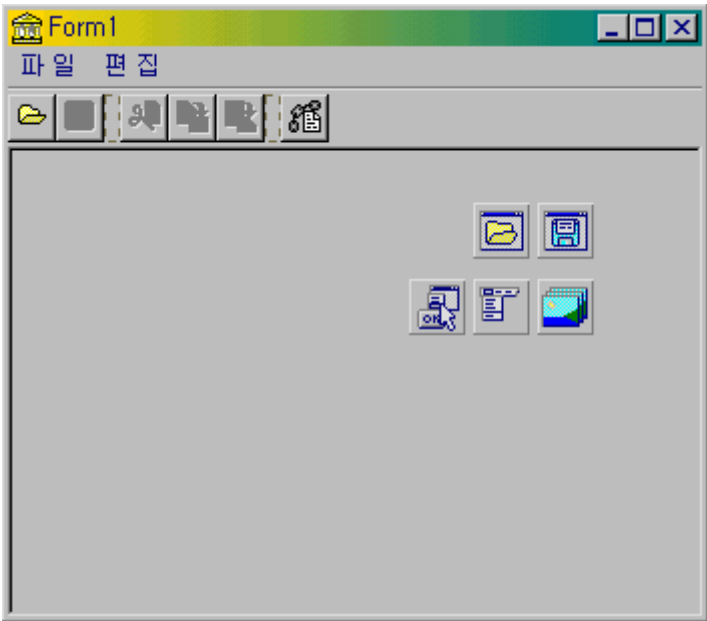


언뜻 보아서는 마치 워드를 실행해서 문서를 불러온 것 같이 보일 것이다. 하지만 잘보면 품의 캡션이 'Form1'으로 이는 델파이 어플리케이션의 컨테이너 내에서 툴바와 함께 in-place 활성화가 된 것이다. 만약 어플리케이션에 메뉴 컴포넌트를 추가했으면, 메뉴도 병합되어 보일 것이다.

이때 AutoActivate 프로퍼티가 aaDoubleClick 으로 설정되어 있기 때문에, 더블 클릭할 경우 객체 편집 모드로 들어가는 것이다. 만약 이 값이 aaManual 이면 Active 속성을 이용해서 코드로 서버를 활성화 시킬 수 있게 되고, aaGetFocus 이면 입력 포커스가 컨테이너에 오기만 하면 서버가 활성화된다.

- OLE 컨테이너 어플리케이션 제작

그러면, 여러가지 OLE 서버 객체를 담을 수 있는 OLE 컨테이너 어플리케이션을 제작해보자. 먼저 폼을 다음과 같이 디자인한다.



폼에 TOleContainer, TImageList, TMainMenu, TToolBar, TActionList, TOpenDialog, TSaveDialog 컴포넌트를 각각 추가한 것이다. 액션 리스트의 사용법에 대해서는 이미 자세히 설명한 바 있으므로, 다시 다루지는 않겠다. ImageList 에 적절한 비트맵을 추가하여 툴바와 메뉴의 비트맵으로 사용할 수 있도록 한다. 즉, MainMenu 와 Toolbar, ActionList 컴포넌트의 Images 프로퍼티를 ImageList 로 선택하고 메인 메뉴와 툴바의 Action 을 적절하게 설정하면 된다.

이 어플리케이션에서 사용되는 액션에는 다음과 같은 것들이 있다.

액션	역할	액션	역할
actNew	새로운 객체 생성	actOpen	파일에서 객체 읽기
actSave, actSaveAs	객체를 파일로 저장	actExit	종료
actCut	객체 오려두기	actCopy	객체 복사하기

actPaste	객체 붙이기	actInsert	객체 삽입
----------	--------	-----------	-------

먼저, 파일의 내용을 저장할 전역 변수를 선언한다.

```
var
  Form1: TForm1;
  ObjectFileName: TFileName;
```

폼의 OnCreate 이벤트 핸들러에서 ObjectFileName 변수를 초기화 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ObjectFileName := '';
end;
```

그리고, 메뉴를 클릭할 때 파일 저장과 복사, 잘라내기, 붙여넣기 등의 액션과 메뉴의 Enabled 프로퍼티를 조절하도록 한다.

```
procedure TForm1.mnFileClick(Sender: TObject);
begin
  with OleContainer1 do
  begin
    actSave.Enabled := Modified;
    mnSaveAs.Enabled := Modified;
  end;
end;
```

```
procedure TForm1.mnEditClick(Sender: TObject);
begin
  with OleContainer1 do
  begin
    actCut.Enabled := State <> osEmpty;
    actCopy.Enabled := State <> osEmpty;
    actPaste.Enabled := CanPaste;
  end;
end;
```



이제, 파일을 읽어오거나 새로운 객체를 생성하는 부분을 구현할 차례이다.

```
procedure TForm1.actOpenExecute(Sender: TObject);
```

```
begin
```

```
  with OpenFileDialog1 do
```

```
    if Execute then
```

```
    begin
```

```
      OleContainer1.CreateObjectFromFile(FileName, False);
```

```
      ObjectFileName := FileName;
```

```
      actCut.Enabled := True;
```

```
      actCopy.Enabled := True;
```

```
      actPaste.Enabled := True;
```

```
    end;
```

```
end;
```

```
procedure TForm1.actNewExecute(Sender: TObject);
```

```
begin
```

```
  if (OleContainer1.State = osEmpty) or
```

```
    (MessageDlg('현재 객체를 삭제할까요?', mtConfirmation, mbOkCancel, 0) = mrOk) then
```

```
  begin
```

```
    with OleContainer1 do
```

```
    begin
```

```
      DestroyObject;
```

```
      actInsertExecute(Sender);
```

```
      DoVerb(PrimaryVerb);
```

```
      ObjectFileName := '';
```

```
    end
```

```
  end;
```

```
end;
```

```
procedure TForm1.actInsertExecute(Sender: TObject);
```

```
begin
```

```
  if (OleContainer1.State = osEmpty) or
```

```
    (MessageDlg('현재 객체를 삭제할까요?', mtConfirmation, mbOkCancel, 0) = mrOk) then
```

```
  if OleContainer1.InsertObjectDialog then
```

```

begin
    actCut.Enabled := True;
    actCopy.Enabled := True;
    actPaste.Enabled := OleContainer1.CanPaste;
    with OleContainer1 do
        DoVerb(PrimaryVerb);
    end;
end;

```

3 가지 액션이 모두 비슷한 역할을 한다. ‘열기’ 메뉴를 선택한 경우에는 일단 OpenFileDialog 를 실행하여 파일 이름을 얻은 후 OleContainer 의 CreateObjectFromFile 메소드를 이용한다. 그에 비해 ‘새 문서’를 선택한 경우에는 먼저 컨테이너의 객체를 DestroyObject 메소드를 이용해서 삭제한 후 ‘객체 삽입’을 수행하게 된다.

‘객체 삽입’ 메뉴에서는 InsertObjectDialog 메뉴를 호출하여 표준 대화상자에서 삽입할 객체를 선택하도록 한다.

Cut, Copy, Paste 에 대한 액션은 다음과 같이 구현한다.

```

procedure TForm1.actCopyExecute(Sender: TObject);
begin
    OleContainer1.Copy;
    actPaste.Enabled := True;
end;

```

```

procedure TForm1.actCutExecute(Sender: TObject);
begin
    if OleContainer1.State <> osEmpty then
        with OleContainer1 do
            begin
                Copy;
                DestroyObject;
                actCopy.Enabled := False;
                actPaste.Enabled := OleContainer1.CanPaste;
                ObjectFilename := '';
            end;
        end;
end;

```

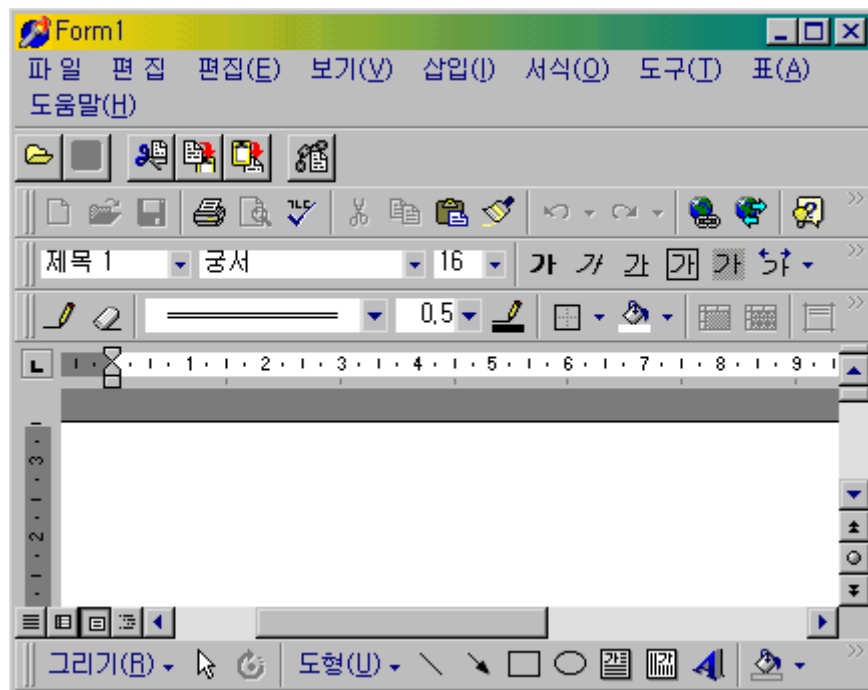
```

procedure TForm1.actPasteExecute(Sender: TObject);
begin
  if (OleContainer1.State = osEmpty) or
    (MessageDlg('현재 객체를 교체할까요?', mtConfirmation, mbOkCancel, 0) = mrOk) then
  begin
    OleContainer1.Paste;
    actCopy.Enabled := True;
    actCut.Enabled := True;
  end;
end;

```

복사는 단순히 Copy 메소드를 호출하면 된다. 그에 비해 Cut 는 컨테이너에 객체가 들어 있다면 DestroyObject 메소드를 호출하여 이를 삭제하고, Copy 메소드를 수행하도록 한다. 또한, 이런 변경 사항에 따른 부대적인 설정이 더 필요하다. Paste 는 Paste 메소드를 이용하여 구현하게 된다.

이것으로 구현이 완료되었다. 이 예제는 Chap27 디렉토리의 Exam1.dpr 프로젝트로 저장되어 있으니 이를 참고하기 바란다. 컴파일하고 실행하면 쉽게 여러가지 OLE 서버 객체를 컨테이너에 담을 수 있다. 또한, 다음과 같이 in-place 활성화를 시켜서 비주얼 편집을 하게 되면 메뉴 병합 등이 실행되는 것을 볼 수 있을 것이다.



## 자동화 컨트롤러(Automation controller)의 제작

자동화 컨트롤러는 자동화 서버를 제어하는 어플리케이션을 말한다. 자동화 서버는 자동화 프로토콜을 통해 접근하여 기능을 할 수 있는 어플리케이션으로 대표적인 자동화 서버로는 마이크로소프트 워드, 엑셀, 인터넷 익스플로러 등을 들 수 있다.

자동화 서버의 타입 라이브러리를 import 하면 자동으로 자동화 서버를 제어할 수 있는 클래스가 만들어진다. 이런 클래스에는 dispatch 인터페이스 wrapper 와 경우에 따라서는 듀얼 인터페이스 wrapper 가 포함되어 있다. 타입 라이브러리를 import 하려면 다음과 같이 하면 된다.

1. Project|Import Type Library 메뉴를 선택한다.
2. 목록에서 타입 라이브러리를 선택한다.

만약 타입 라이브러리가 목록에 없다면, Add 버튼을 누르고 타입 라이브러리 파일(TLB)을 찾아서 이를 선택하고 OK 버튼을 누른다.

## OLE 자동화 기법을 이용한 엑셀과 워드 제어

텔과이를 이용해서 프로그래밍을 하다가 보면 간혹 오피스와의 연계를 통한 프로그래밍이 필요한 경우가 있다. 이럴 때에는 OLE 자동화 기법을 이용해서 오피스 객체를 직접 제어 하면 생각보다 쉽게 문제를 해결할 수가 있다.

기본적으로 OLE 자동화를 이용해서 오피스 객체를 사용하는 것은 그다지 어렵지 않다. 실제로 공부를 해야 하는 것은 텔과이의 문제라기 보다는, 다소 복잡한 엑셀과 워드 객체의 구조를 익히는 것이 문제라고 할 수 있다.

이렇게 엑셀과 워드를 OLE 자동화를 이용해서 다루는 방법에는 크게 2 가지가 있다.

첫 번째 방법은 워드와 엑셀 객체를 가변형 변수와 IDispatch 인터페이스를 이용해서 사용하는 것이며, 두 번째 방법은 표준 COM 인터페이스와 dispinterface 를 이용하는 방법이다. 이 2 가지 방법에는 중요한 차이가 있는데 가변형 변수를 사용하는 방법은 아주 쉽지만 속도가 다소 처지는 방법이며, COM 인터페이스를 사용하는 방법은 다소 어렵지만 속도가 빠르다.

OLE 자동화를 원활하게 사용하려면, 시스템의 RAM 이 많이 필요하다. OLE 자동화는 실제로 CPU 속도보다는 메모리 크기에 수행 속도가 좌우된다고 해도 과언이 아니다. 그러므로, 가능하면 많은 메모리를 확보한 시스템에서 돌릴 수 있도록 하는 것이 중요하다.

- OLE 자동화를 시작하자 !

앞에서도 언급했지만, 델파이에서 OLE 자동화를 사용하는 방법에는 크게 2 가지가 있다. 인터페이스를 직접 이용하거나 IDispatch 인터페이스와 가변형 변수를 사용하는 방법이 그것인데, 인터페이스를 직접 사용하면 클라이언트 측에서 개발자의 코드의 데이터 형 검사를 하기 때문에 비교적 빠른 속도를 낼 수 있다. 이런 방식을 ‘early binding’이라고 한다. 그렇지만, 처음 OLE 자동화를 익히는 사람에게는 이 방식보다는 IDispatch 인터페이스와 가변형 변수를 이용하는 방법이 훨씬 쉽기 때문에, IDispatch 인터페이스를 이용하는 방법에 대해서 알아보도록 하자.

다음의 코드는 엑셀을 시작하는 핵심 부분이다.

```
var
    V: Variant;
begin
    V := CreateOLEObject('Excel.Application');
    V.Visible := True;
end;
```

반대로, 엑셀을 마칠 때에는 다음과 같이 한다.

```
if not VarIsEmpty(V) then V.quit;
```

앞의 코드를 사용하려면, 먼저 델파이 유닛의 uses 절에 ComObj 를 추가해야 한다. ComObj 유닛에는 OLE 자동화 객체를 불러오고, 이들을 호출할 수 있는 루틴들이 포함되어 있다. 보통 가장 많이 사용하는 함수가 앞에서 사용한 CreateOLEObject 함수이다. 그리고 그 밖에도 VarDispatchInvoke, DispatchInvoke, GetIDsOfNames 등의 함수를 사용할 수 있다.

처음에 사용한 CreateOLEObject('Excel.Application') 줄이 실행되면 엑셀이 백 그라운드에서 시작되기 때문에 사용자에게 보이지 않는다. 보통 엑셀 객체의 엔진만 사용하는 경우라면 이렇게 생성한 객체를 직접 사용해서 원하는 작업을 하면 된다. 그런데, 우리가 하는 작업이 엑셀에서 어떻게 이루어지는지를 보여주려면 Visible 프로퍼티를 True 로 설정해야 한다. 보통 코드가 어떻게 작동하는지 눈으로 볼 수 있기 때문에, 디버깅 과정에서는 이를 True 로 설정했다가 디버깅이 끝난 후에는 나타나지 않도록 해서 속도를 증진 시키는 것이 바람직하다.

여기에서 사용한 V 라는 변수는 가변형 변수이며, CreateOLEObject 함수는 내부적으로 여러가지 OLE 함수를 호출한다. 그 결과로 COM 객체를 생성하고 이 객체의 IDispatch 인터페이스를 돌려주는 역할을 한다.

가변형(variant) 변수는 기본적으로 어떠한 데이터 형도 모두 담을 수 있다. 그렇기 때문에,

여기에 COM 객체에 대한 IDispatch 인터페이스도 담을 수 있는 것이다.

그러니까, 'V := CreateOLEObject('Excel.Application')' 이라는 코드의 의미는 엑셀의 Application 이라는 COM 객체에 대한 인스턴스를 생성하고, 이 인스턴스를 가리키는 IDispatch 인터페이스를 V 라는 가변형 변수에 대입하는 것이다. 이렇게 되면, V 라는 가변형 변수는 이때부터 직접 COM 객체를 다루듯이 사용할 수 있게 된다. 그렇기 때문에, 'V.Visible := True' 라는 문장이 동작하게 되는 것이다.

보기에는 간단하지만 'V.Visible := True'라는 문장에 의해 내부적으로 많은 일이 일어나게 되는데, 이를 수행하기 위해 IDispatch 인터페이스의 GetIDsOfNames, Invoke 메소드가 호출된다.

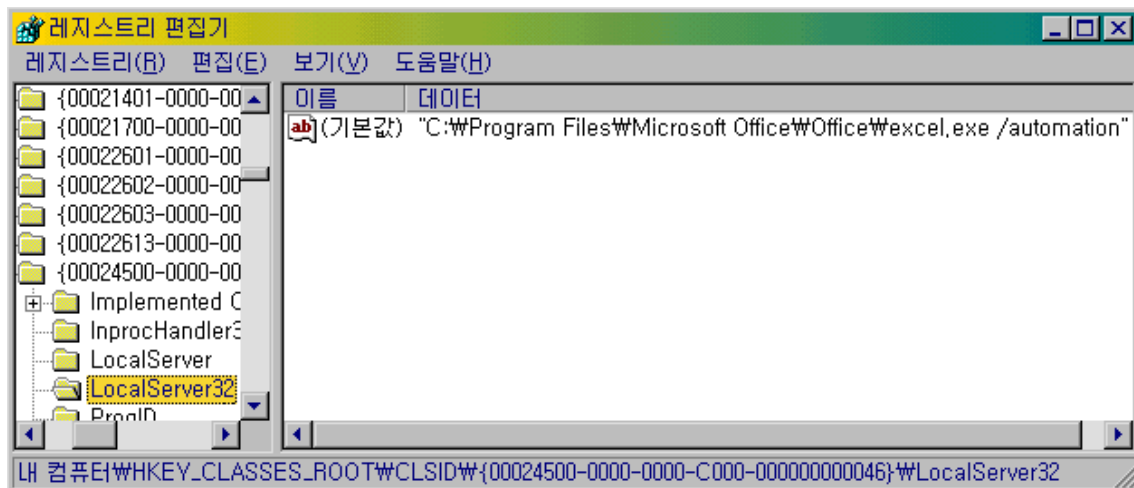
기본적으로 IDispatch 인터페이스는 가변형 변수를 사용해서 쉽게 사용할 수 있도록 디자인되었다. 이를 이용하면 디자인 시에 데이터 형 검사를 하지 않고, 런타임에서 데이터 형을 검사하게 되는데, 이를 'late binding'이라고 한다. 다른 말로 앞의 코드를 예로 들면, 델파이는 엑셀의 Application 객체가 Visible 이라는 프로퍼티가 있는지 전혀 알지 못한다. 그렇기 때문에, 만약에 COM 객체가 지원하지 않는 프로퍼티나 메소드를 사용한다고 해도 델파이의 컴파일러는 아무런 에러도 발생시키지 않는다. 다만, 실제로 이 메소드나 프로퍼티가 호출될 때 예외가 발생하게 될 것이다.

그러면 이제 마칠 때 사용하는 코드를 살펴 보자. VarIsEmpty 함수는 파라미터로 전해지는 가변형 변수가 어떤 객체나 실제 데이터를 가리키는지 알아보게 된다. 그러므로, 'if not VarIsEmpty(V) then ...' 의 의미는 가변형 변수 V 가 실제로 어떤 객체를 가리키고 있다면 then 절의 문장이 실행된다는 것이다.

이 문장에 전혀 문제가 없는 것은 아닌데, 예를 들어 V 에 엑셀의 객체가 아닌 다른 값이 들어 있더라도 then 절이 실행되게 된다.

## ● 엑셀 자동화 객체의 생성

CreateOLEObject 함수를 사용하면 COM 객체를 생성하고, 여기에 대한 IDispatch 인터페이스를 얻을 수 있다는 것을 앞에서 설명했다. 이 함수는 파라미터로 COM 객체를 지정하는 문자열을 받게 되어 있다. 앞에서 예를 든 코드에서는 'Excel.Application'이 사용되었다. 이런 문자열은 시스템의 레지스트리에서 발견할 수 있는데, 좀더 정확히 말하면 이 문자열에 해당되는 CLSID 값을 레지스트리에서 찾아서 레지스트리의 LocalServer 키의 정보를 가지고 COM 객체의 인스턴스를 생성하게 된다. 엑셀의 경우 Local Server 정보는 다음 그림과 같이 엑셀 파일의 패스 경로와 /automation 이라는 스위치로 이루어져 있다.



이렇게 프로그램을 나타내는 문자열을 프로그램 ID 라고 하며, 'ProgIDs'로 표시한다.

#### ● 엑셀 자동화 객체의 이해

엑셀 자동화 객체의 구조에서 가장 상위의 객체가 Application 객체이다. 그 밑에 Workbooks, Worksheets, Charts 등의 객체가 존재한다. 그런데, 이러한 객체의 구조는 델파이의 VCL 객체구조와는 다소 차이가 있다. 델파이의 방식으로 이해를 하면 Workbooks 객체는 Application 객체의 자손이며, Worksheets 객체는 Workbooks 객체의 자손 객체로 생각할 수도 있다. 그러나, OLE 자동화에서 나타나는 객체의 구조는 OOP의 계층 구조와는 전혀 다른 형태를 가지고 있다. OLE 자동화에서의 계층 구조는 단지 어떤 객체에 접근할 때, 어떤 객체를 통해서 접근해야 하는지를 나타내는 것이다. 물론, OOP의 계층 구조로도 이를 나타낼 수는 있지만, 근본적으로는 다른 것임을 잊지 말도록 한다.

예를 들어, OOP의 계층 구조라면 A 밑에 B, B 밑에 C가 있다면, A.B, A.B.C의 형태로는 접근이 가능하지만, A.C의 형태의 접근은 불가능하다. 그러나, OLE 자동화의 계층 구조에서는 이것도 가능한 경우가 많다.

엑셀의 Application 객체는 사용이 가능한 엑셀 자동화 객체들의 세트를 나타내는 가장 일반적이고, 추상적인 객체라고 말할 수 있다. 즉, Application 객체는 엑셀에서 접근이 가능한 모든 기능을 담고있는 최상위 레벨의 컨테이너이다.

Application 객체의 바로 아래 객체인 Workbooks 객체는 Worksheets와 Charts 객체의 집합을 포함하고 있다. Worksheet 객체는 스프레드 시트의 표준 페이지를 나타내며, Chart 객체는 그래프를 나타낸다. 또한, Sheets 객체는 Worksheets와 Charts 객체를 모두 포함한다.

이러한 엑셀의 객체를 사용하는 순서와 방법은, 실제로 엑셀이라는 스프레드 시트 어플리케이션을 사용하는 순서에 입각해서 생각하면 비교적 쉽게 익힐 수 있다. 보통의 경우 엑셀을 사용하는 사용자는 일단 스프레드 시트 파일을 열고, 데이터를 입력한 후, 이를 이용한

계산식을 입력하고, 데이터에 대한 그래프를 그리게 된다. 엑셀 자동화 객체를 이용하는 방법도 여기에 입각해서 생각하면 비교적 간단하게 이해할 수 있다.

그럼 이제 실제 엑셀 객체를 이용하는 프로그램을 작성해 보도록 하자.

새로운 어플리케이션을 시작하고, 폼 위에 버튼 1 개와 리스트 박스를 1 개 없도록 하자.

그리고, OLE 자동화를 사용해야 하므로 폼 클래스 선언문의 private 섹션에 XL 이라는 가변형 변수를 선언하고, uses 절에 ComObj 를 추가한다.

버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

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

```
const
```

```
    {Excel Sheet Type}
```

```
    xlChart = -4109;
```

```
    xlDialogSheet = -4116;
```

```
    xlExcel4IntlMacroSheet = 4;
```

```
    xlExcel4MacroSheet = 3;
```

```
    xlWorksheet = -4167;
```

```
    {Excel WBA Template}
```

```
    xIWBAChart = -4109;
```

```
    xIWBAExcel4IntlMacroSheet = 4;
```

```
    xIWBAExcel4MacroSheet = 3
```

```
    xIWBAWorksheet = -4167;
```

```
var
```

```
    i, k: Integer;
```

```
    Sheets: Variant;
```

```
begin
```

```
    XL := CreateOLEObject('Excel.Application');
```

```
    XL.Visible := True;
```

```
    XL.Workbooks.Add;
```

```
    XL.Workbooks.Add(xIWBAChart);
```

```
    XL.Workbooks.Add(xIWBAWorksheet);
```

```
    XL.Workbooks[2].Sheets.Add(,1, xlChart);
```

```
    XL.Workbooks[3].Sheets.Add(,1, xlWorksheet);
```

```
    for i := 1 to XL.Workbooks.Count do
```

```
    begin
```



```

        ListBox1.Items.Add('워크북 이름: '+XL.Workbooks[i].Name);
    for j := 1 to XL.Workbooks[i].Sheets.Count do
        ListBox1.Items.Add('    시트 이름: '+XL.Workbooks[i].Sheets[j].Name);
    end;
end;

```

Workbooks 는 일종의 컬렉션 객체로 Workbooks 객체의 컬렉션이다. 마찬가지로 Sheets, Charts 역시 각각 Sheet, Chart 의 컬렉션 객체이다. Workbooks 객체의 Add 메소드는 Workbooks 컬렉션에 새로운 Workbook 객체를 추가하는 메소드이다. COM 객체의 메소드를 사용할 때에는 가변 파라미터 리스트(variable parameter list)라는 기법을 사용할 수 있는데, 이 기법은 메소드의 파라미터를 원한다면 건너뛴 수 있는 것이다. 위에서 보면 Workbooks 의 Add 메소드가 처음에는 아무런 파라미터 없이도 쓰였다가, 그 뒤의 두 줄에서는 파라미터를 하나씩 가지고 있다. 이 파라미터로 쓰인 것이 어떤 형태의 Sheet 객체를 추가할 것인가를 결정한 것인데, xlWBATWorksheet 파라미터는 Worksheet 객체를 xlWBATChart 파라미터는 Chart 객체를 추가한다는 것을 의미한다.

이러한 컬렉션 객체는 뒤에 대괄호로 숫자를 넣어서 몇 번째 객체임을 표시할 수 있게 되어 있다. Workbooks[2]란 두 번째 Workbook 객체를 의미한다.

```

XL.Workbooks[2].Sheets.Add(,1, xlChart)

```

앞의 코드에서는 Sheets 객체의 Add 메소드가 사용되었다. Sheets 객체의 Add 메소드는 4 개의 파라미터를 가진다. 첫번째와 두번째 파라미터인 Before, After 는 각각 새로운 sheet 가 추가될 때 그 전후에 위치하는 sheet 객체를 담은 가변형 값을 지정하며, 세번째인 Count 파라미터에는 추가할 시트의 수를, 마지막으로 네번째인 Type 파라미터에는 추가할 시트의 종류를 지정하게 된다. 추가되는 Sheet 의 종류로는 xlWorksheet, xlChart, xlExcel4MacroSheet, xlExcel4IntlMacroSheet 등의 것들이 있는데, 디폴트 값으로는 xlWorksheet 가 지정된다.

앞의 코드의 경우에는 콤마를 처음에 2 개 찍음으로써 처음 2 개의 파라미터는 생략하고, xlChart 형의 Sheet 객체를 하나 추가하게 된다. 이런 식으로 파라미터를 생략하는 것은 IDispatch 를 이용한 'late binding'에서만 허용되며, 우리가 나중에 공부하게 될 COM 인터페이스를 직접 이용하는 방법에서는 사용할 수 없다.

그 뒤에 나오는 for..loop 에서는 Workbook 객체의 이름과 각 Workbook 객체의 Sheet 객체의 이름을 리스트 박스에 표시하게 된다.

그럼, 여기에서 선언해서 사용된 각종 상수 들의 구체적인 값을 알아내는 방법을 알아보자. 기본적으로 이런 상수는 엑셀의 타입 라이브러리(type library)의 값을 읽어서 알아낼 수 있다. 타입 라이브러리의 값을 읽는 방법은 마이크로소프트 SDK 에서 제공되는 OleView 어

플리케이션 등의 도구를 이용하거나, 델파이의 타입 라이브러리 에디터를 사용할 수 있다.

여기서는 델파이에 내장된 타입 라이브러리를 이용하는 방법을 사용하도록 하겠다.

델파이 메뉴에서 Project|Import Type Library 를 선택하고, 엑셀의 타입 라이브러리 파일인 Excel8.olb 파일을 선택한다. 그러면, 많은 수의 경고 메시지가 나오기는 하지만 예러는 없이 타입 라이브러리가 로드될 것이다. 이렇게 많은 경고 메시지가 나오는 이유는 엑셀의 타입 라이브러리에 정의된 많은 수의 이름과 델파이의 예약어가 같을 경우, 이 이름들을 자동으로 수정하면서 나오는 메시지이다. 보통의 경우 이름의 처음이나 끝에 밑줄(\_) 기호가 추가된다. 이렇게 해서 타입 라이브러리를 읽어온 후에 원하는 상수의 실제 값을 찾아보면 된다. 그러나, 매번 엑셀이나 워드 등의 자동화 객체를 사용할 때마다 이들 상수를 선언해서 사용하는 것은 매우 비효율적이다. 그러므로, 엑셀과 워드에서 사용되는 상수들을 선언한 유닛을 따로 만들어 놓고 이를 이용하는 것이 좋을 것이다. 그래서, 앞에서의 설명에는 엑셀과 워드에 대한 상수를 담은 유닛인 XLConst.pas, WordConst.pas 유닛을 만들어 놓고, 이를 사용하도록 하겠다. 이들 유닛은 이달의 디스켓으로 제공되니 유용하게 사용하기 바란다.

#### ● 엑셀 워크시트를 사용하자

이번에는 워크 시트를 이용해서 실제 데이터를 입력하고, 이를 이용해 계산을 하는 방법과 데이터를 저장하는 방법을 알아보도록 하자.

새로운 어플리케이션을 시작하고 폼에 버튼을 하나 없도록 하자. 그리고, 폼 클래스의 private 섹션에 XL 이라는 가변형 변수를 선언하고, uses 절에 ComObj.pas 와 XLConst.pas 유닛을 추가한다. 이제 Button1 에 대한 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i: Integer;
    Sheet, Range, Column: Variant;
begin
    XL := CreateOLEObject('Excel.Application');
    XL.Visible := True;
    XL.Workbooks.Add(xlWBATWorksheet);
    XL.Workbooks[1].Worksheets[1].Name := 'Example1';
    Sheet := XL.Workbooks[1].Worksheets['Example1'];
    for i := 1 to 10 do
        Sheet.Cells[i, 1] := i;
```

```

Range := XL.Workbooks(1).Worksheets('Example1').Range('B1:F10');
Range.Formula := '=RAND()';
Range.Columns.Interior.ColorIndex := 5;
Range.Borders.LineStyle := xlContinuous;
Column := XL.Workbooks[1].Worksheets(1).Columns;
Column.Columns[1].ColumnWidth := 5;
Column.Columns[1].Font.Color := clRed;
end;

```

일견 복잡해 보이지만, 그다지 어렵지는 않을 것이다. 위의 코드에서 사용하는 OLE 자동화 객체를 각각 Sheet, Range, Column 이라는 가변형 변수에 담아서 사용한다. 일단 XL 이라는 가변형 변수에 엑셀의 Application 자동화 객체를 저장하고, Workbooks 통합문서에 워크 시트를 하나 추가한다. 그리고, 추가한 워크 시트의 이름을 'Example1'이라고 명명했다.

그러면 이제 가변형 변수 Sheet 에 우리가 사용할 Worksheet 객체를 대입하도록 하자.

Sheet := XL.Workbooks[1].Worksheets['Example1'] 이라는 문장은 엑셀의 첫번째 통합 문서내의 'Example1' 이라는 이름의 워크시트 객체를 찾아서 대입한다. 이때 Workbooks[1] 통합문서의 첫번째 워크 시트의 이름을 'Example1'으로 지정했으므로, Worksheets['Example1']은 Worksheets[1]과 같은 의미를 가지게 된다.

Sheet 객체를 이용해서 실제 데이터를 입력하려면 Cells 프로퍼티나 Range 객체를 사용해야 한다. Cells 프로퍼티의 경우에는 행과 열을 각각 숫자로 기입해서 셀의 위치를 나타낸다. 그러므로, Cells[1, 1]은 엑셀의 'A1' 셀을 Cells[3, 2]는 'B3' 셀을 나타낸다. Range 객체는 한꺼번에 여러 개의 셀을 선택할 때 사용하는 것으로 앞의 예제의 Range('B1:F10')의 경우에는 'B1' 셀이 가장 좌측 위에 위치하고, 'F10' 셀이 가장 우측 아래라고 할 때 이들에 의해 둘러싸여지는 직사각형 범위의 모든 셀을 가리키게 된다.

이를 이해하면 앞의 코드 들을 쉽게 이해할 수 있을 것이다.

한 가지 재미있는 것은 대괄호와 보통 괄호를 마구 혼용해서 썼는데, 이렇게 사용하는 것이 모두 허용된다. 그리고, 직접 셀이나 Range 객체의 Formula 프로퍼티에 문자열로 엑셀에서 사용하던 함수식을 입력할 수 있다. 그러므로, Range.Formula := '=RAND()' 와 같은 문장을 사용할 수 있다. 이 문장으로 Range 변수에 지정된 모든 셀들에 0~1 사이의 난수가 대입된다.

그 다음 문장은 Range 에 해당되는 모든 셀들의 내부 색상을 5 번 색상으로 선택하였다. ColorIndex 프로퍼티에는 미리 정의되어 있는 색상들이 있는데 1 번부터 8 번까지가 각각 black, white, red, green, blue, yellow, purple, cyan 으로 지정되어 있다. 그러므로, 이 문장에서는 5 번인 blue 컬러가 선택된다.

이렇게 색상이 선택되면 엑셀에서는 셀간을 구분하는 구분선이 없어지는데, 이를 없애지지

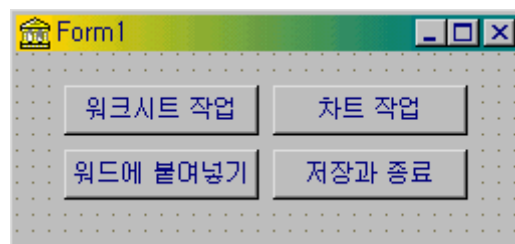
않게 하려면, 그 다음 문장에서처럼 Range 객체의 Borders.LineStyle 프로퍼티를 xlContinuous 로 지정하면 된다.

이런 방식으로 셀의 내부를 지정하는 많은 프로퍼티가 있는데, 예를 들어 Interior.Pattern 프로퍼티는 셀 내부를 그리는 패턴을 지정할 수 있다. 이들 프로퍼티에 대한 자세한 설명은 엑셀 객체에 대해 자세히 설명한 서적이나 엑셀의 도움말을 참고하기 바란다.

다음에 이어지는 문장들은 워크 시트의 Columns 객체를 Column 이라는 가변형 변수에 대입하고 이를 이용해서 Column 의 형태를 결정하는 문장들이다. 여기에서는 첫번째 열의 폭을 5, 폰트의 색상을 clRed 로 설정하였다.

#### ● 엑셀 차트 기능과 워드 객체의 사용

이제 데이터를 입력하고, 이를 이용해서 차트를 그리고 이를 MS 워드 문서로 복사까지 해 줄 수 있도록 확장해보자. 폼에 버튼을 3 개 더 얹고 (합이 4 개이다), 폼 클래스의 private 섹션에 Word 라는 가변형 변수를 하나 더 추가하고, uses 절에 WordConst, ActiveX 를 추가한다. 그리고 각 버튼의 캡션을 다음과 같이 ‘워크시트 작업’, ‘차트 작업’, ‘워드 붙여넣기’, ‘저장과 종료’ 라고 설정한다.



그리고, Button1 의 OnClick 이벤트를 핸들러를 다음과 같이 수정한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Sheet: Variant;
    i: Integer;
begin
    XL := CreateOLEObject('Excel.Application');
    XL.Visible := True;
    XL.Workbooks.Add(XLWBATWorksheet);
    XL.Workbooks[1].Worksheets[1].Name := 'Example1';
    Sheet := XL.Workbooks[1].Worksheets['Example1'];
    for i := 1 to 10 do
```

```

        Sheet.Cells[i, 1] := i;
    end;

```

Button1 의 이벤트 핸들러는 거의 앞에서 설명한 예제와 변함이 없다. 그러면, 이제 이를 이용해서 차트를 그릴 Button2 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button2Click(Sender: TObject);
var
    Range, Sheets: Variant;
begin
    XL.Workbooks[1].Sheets.Add(,1, xlChart);
    XL.Workbooks[1].Sheets[1].Name := 'ExamChart1';
    Sheets := XLApp.Sheets;
    Range := Sheets.Item['Example1'].Range['A1:A10'];
    Sheets.Item['ExamChart1'].SeriesCollection.Item[1].Values := Range;
    Sheets.Item['ExamChart1'].ChartType := xl3DPie;
    Sheets.Item['ExamChart1'].SeriesCollection.Item[1].HasDataLabels := True;
end;

```

차트 객체를 처음 생성할 때에는 Sheets 객체의 Add 메소드를 사용해서 xlChart 형식의 Sheet 를 추가하면 된다. 앞의 코드에서 처음 2 개의 파라미터를 생략했는데, 이것은 앞뒤에 위치할 Sheet 를 지정하지 않은 것이며, 그 뒤의 파라미터는 xlChart 형식의 시트를 1 개 추가한다는 의미이다.

앞의 코드에서는 Sheets 객체를 계속 사용하는데, 앞에서도 설명했지만 Sheets 객체는 Worksheets 와 Charts 객체를 모두 가리킬 수 있기 때문에 꽤 편리하게 사용할 수 있다. 일단 이렇게 차트 객체가 생성되면 실제로 어떤 데이터를 가지고 차트를 그릴 것인지를 결정해야 한다. 이를 위해서 Button1Click 이벤트 핸들러에서 삽입한 셀들을 Range 로 설정해서 이 셀들을 SeriesCollection 의 Values 로 설정한다. 이렇게 Values 프로퍼티에 데이터를 설정하면 이를 이용해서 그래프를 그릴 수 있게 된다.

Series 란 그래프를 그리려고 하는 데이터의 range 를 의미한다. 그리고, SeriesCollection 이란 이러한 Series 의 Collection 이다.

그러므로, Sheets.Item['ExamChart1'].SeriesCollection.Item[1].Values := Range 란 코드의 의미는 'ExamChart1'이라는 이름을 가진 시트의 첫번째 Series 의 데이터 Range 값을 Range 라는 변수에 들어있는 값으로 설정한다는 의미이다.

그 다음 문장은 차트의 형을 3 차원 파이 그래프로 결정한 것이며, 데이터 라벨을 보이도록 하였다.

이제 이렇게 만들어진 차트를 워드에 추가해 보도록 하자. 일단은 데이터를 클립보드에 복사하고, 이 데이터를 다시 워드로 붙여넣기(paste) 하면 된다. Button3 의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.Button3Click(Sender: TObject);
var
    Sheets, Range: Variant;
    i, ParNo: Integer;
begin
    Sheets := XL.Sheets;
    Sheets.Item['ExamChart1'].Select;
    XL.Selection.Copy;
    Word := CreateOLEObject('Word.Application');
    Word.Visible := True;
    Word.Documents.Add;
    Word.Documents.Item(1).Paragraphs.Add;
    Word.Documents.Item(1).Paragraphs.Add;
    ParNo := Word.Documents.Item(1).Paragraphs.Count;
    Range := Word.Documents.Item(1).Range(
        Word.Documents.Item(1).Paragraphs.Item(ParNo).Range.Start,
        Word.Documents.Item(1).Paragraphs.Item(ParNo).Range.End);
    Range.Text := '이것은 그래프 입니다.';
    Word.Documents.Item(1).Paragraphs.Add;
    Range := Word.Documents.Item(1).Range(
        Word.Documents.Item(1).Paragraphs.Item(ParNo + 1).Range.Start,
        Word.Documents.Item(1).Paragraphs.Item(ParNo + 1).Range.End);
    Range.PasteSpecial(,,,wdPasteOleObject);
end;
```

먼저, 복사할 객체를 선택해야 한다. 객체의 Select 메소드를 사용해서 차트 객체를 선택하고, 선택된 객체를 클립보드로 복사하는 Copy 메소드를 호출한다.

그리고, 워드 객체를 사용하기 위해 Word := CreateOLEObject('Word.Application') 문장에서 워드 객체를 생성한다. 오피스 95 의 워드 까지는 워드 객체가 'Word.Basic'으로 표현되었으나, 오피스 97 의 워드 8.0 부터는 Word.Application 으로 바뀌었다. 엑셀에서의 통합 문서 객체인 Workbooks 에 해당하는 것이 워드에서는 Documents 객체이다. 그러므로,

Word.Documents.Add 메소드에 의해 새로운 워드 문서가 추가된다. 그리고, 그 다음 문장에서 사용한 Paragraphs 객체의 Add 메소드는 새로운 문단을 추가하는 역할을 하게 된다. 워드에서는 엔터 키를 누를 때마다 새로운 문단이 시작되므로 이것의 의미는 새로운 줄을 하나 추가하는 것과 같은 의미가 된다. 앞의 경우에는 처음에 두 줄의 빈문단을 삽입하게 된다. 그리고 나서, 범위를 설정하고, 그 범위에 적절한 문장을 삽입한다. Range 변수의 범위를 설정할 때 범위의 시작점과 끝점을 모두 지정하는데, 이들은 경우에 따라서는 생략이 가능하다. .

다음에 이어지는 문장에서 역시 엑셀의 차트를 워드에 삽입하는 루틴 들이 이어지고 있다. 이 문장들도 앞에서 설명한 기본적인 문장에 대한 설명과 동일하므로, 중복되는 설명은 따로 하지 않는다. 차트를 붙여넣을 때에는 Paste 메소드와 PasteSpecial 메소드를 사용할 수 있는데, 여기서는 OLE 객체로 간주하여 붙여넣게 되는 PasteSpecial 메소드를 사용한다. 이렇게 PasteSpecial 메소드를 사용해서 붙여넣은 OLE 객체는 나중에 워드 문서에서 이 객체를 더블 클릭할 때 이를 편집할 수도 있게 된다. 이 메소드는 상당히 많은 파라미터를 가지는데, 여기에 대한 설명은 오피스 97 의 OLE 자동화 객체의 메소드에 대한 책이나 도움말을 참고하기 바란다.

이제 워드에 삽입 되었을 것이다. Button4 를 누르면 워드 문서가 저장이 되고, 워드 객체와 엑셀이 모두 종료되도록 하자.

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Word.Documents.Item(1).SaveAs('c:\WExam.doc');
  if not VarIsEmpty(XL) then
  begin
    XL.DisplayAlerts := False;
    XL.Quit;
  end;
  if not VarIsEmpty(Word) then
  begin
    Word.Documents.Item(1).Close(wdDoNotSaveChanges);
    Word.Quit;
  end;
end;
```

위의 코드에서 많은 부분은 앞에서 설명했다. 참고로 엑셀의 경우에는 빠져나갈 때 DisplayAlerts 프로퍼티를 False 로 설정하면 파일을 저장하지 않겠는지 묻는 메시지가 나타나지 않는다. 워드의 경우에도 Close 메소드에 wdDoNotSaveChanges 를 파라미터로

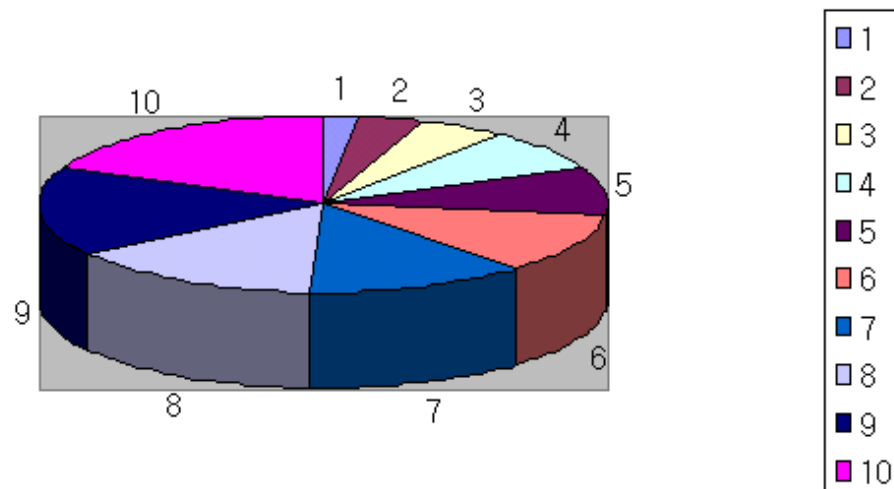
넘겨주면 사용자에게 저장에 대한 메시지를 묻지 않고 종료하게 된다.

이제 완성된 예제를 실행하고, 4 개의 버튼을 차례로 누르면 c:\W 디렉토리에 Exam.doc 이라는 워드 파일이 만들어진다. 이 파일을 워드에서 읽어오면 다음 그림과 같은 문서일 것이다.

↵

↵

이것은 그래프 입니다.↵



간단하게 엑셀과 워드를 제어하는 어플리케이션을 제작해 보았다. 실제로 동작하는 모습을 보여주기 위해서 Visible 프로퍼티를 True 로 설정하는 등의 수행 성능을 떨어뜨리는 코드가 많이 들어가 있으므로, 이점은 꼭 유념해서 살펴보기 바란다.

또한, 차트를 삽입할 때 워드에서 보여지는 크기 등을 지정하지 않았기 때문에, 실제 실행해보면 차트가 너무 커서 잘려서 워드에 삽입되는 등의 모습을 볼 수 있을 것이다. 이런 부분의 세세한 제어는 코드를 추가하면 쉽게 해결할 수 있다.

## Early 바인딩과 Late 바인딩 (IDispatch vs VTable)

앞에서도 간단히 설명한 바 있지만, OLE 자동화를 사용할 때에는 가변형 변수를 사용하는 late 바인딩과 인터페이스를 직접 사용하는 early 바인딩이 있다.

이들은 파라미터를 넘기는 방식이나, 접근 방식에 따라 속도차이가 날 수 밖에 없다. 먼저 기본적인 메모리 사용 방법에 따른 속도를 생각해보면, 스택에 기초한 메모리를 할당하는 경우가 힙에 기초한 메모리 할당 기법에 비해 처리 속도가 빠르다. 그런데, 모든 가변형 파라미터는 기본적으로 가변형 배열로 형변환이 되는데, 이들은 모두 힙에 기초한 메모리를 사용한다.



물론 COM 인터페이스를 사용할 때 가장 빠른 방법은 VTable 의 메소드 주소를 알아내어 직접 호출하는 방법일 것이다. 그러나, 이 방법은 융통성이 적고 사용 방법이 까다로운 단점이 있다. 이를 보완하기 위해서 나온 것이 듀얼 인터페이스(dual interface)이다. 듀얼 인터페이스는 가변형으로 사용하는 IDispatch 의 유연한 동적 디스패치와 VTable 직접 접근 방법의 수행 성능을 모두 지원할 수 있는 인터페이스이다.

기본적으로 모든 COM 인터페이스는 IUnknown 에서 상속받게 되고, OLE 자동화를 지원하는 모든 인터페이스는 IDispatch 를 상속한다. 듀얼 인터페이스는 IDispatch 에서 상속받은 인터페이스로 듀얼 인터페이스는 파라미터를 스택에서 처리하며, 이렇게 하기 위해서 타입 라이브러리를 사용하므로 반드시 듀얼 인터페이스로 처리하기 위해서는 타입 라이브러리를 통해서 데이터 형을 파악할 수 있어야 한다.

- Late 바인딩과 IDispatch

Late 바인딩은 IDispatch 에 의해 구현된다. IDispatch 인터페이스는 IUnknown 인터페이스에 다음과 같은 메소드가 추가된 인터페이스이다.

```
function GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; Dispid: Integer): HRESULT; virtual; stdcall;
function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HRESULT; virtual; stdcall;
function GetTypeInfoCount(out Count: Integer): HRESULT; virtual; stdcall;
function Invoke(Dispid: Integer; const IID: TGUID; LocaleID: Integer; Flags: Word;
    var Params: VarResult, ExcepInfo, ArgErr: Pointer): HRESULT; virtual; stdcall;
```

GetIDsOfNames 메소드는 런타임에서 함수 이름을 가지고 적절한 메소드의 디스패치 ID 를 반환하는 역할을 한다. IDispatch 인터페이스는 Invoke 메소드에 디스패치 ID 를 이용하여 실제 메소드를 호출한다.

Invoke 메소드는 자동화의 핵심이라고 말할 수 있다. 디스패치 ID 를 이용해서 메소드를 호출하게 되는데, 파라미터로 가변형을 사용하게 된다. 함수 호출이 이루어지기 전에, 이러한 파라미터 들은 반드시 가변형 데이터 형으로 변환된다.

이런 방법의 장점은 호출자가 컴파일 시에 자동화 객체 메소드의 수를 알 필요가 없고, 각 함수에 대한 자세한 파라미터의 리스트를 몰라도 되며, 디폴트 파라미터나 옵션 파라미터를 설정하기가 매우 쉽다.

- 수행성능의 문제

자동화 객체를 사용할 때의 가장 근본적인 문제가 되는 것은 Win32 프로세스 공간을 넘어

다니는 것이다. Win32 프로세스 공간은 서로 독립되어 있기 때문에, 서로가 주소를 공유할 수 없는 구조로 되어 있다. 그러므로, 파라미터를 넘길 때에는 일종의 전역 메모리 버퍼나 메시징 시스템을 사용해야 하는데 이런 작업에서 시간을 많이 소요하게 된다.

그러므로, 자동화 객체 호출에서의 속도를 증진시키려면 이렇게 시간을 많이 소요하는 작업에서 조금이라도 효율적인 방법을 써야 한다. 그러므로, Invoke 메소드를 호출하기 전에 모든 파라미터를 가변형으로 변환해야 하고, 이를 함수를 호출한 쪽에서 다시 원래의 데이터 형으로 변경해야 한다면 성능에 막대한 지장을 초래할 것임은 불을 보듯이 뻔한 일이다. 이렇게 가변형으로의 파라미터 변경을 막고 직접 데이터 형을 사용할 수 있게 하려면, 메소드를 호출한 쪽에서 메소드의 이름과 파라미터의 데이터 형을 알 수 있다면 가능한데 이를 위해서 필요한 것이 바로 타입 라이브러리이다.

자동화에서 타입 라이브러리는 객체에 의해 지원되는 인터페이스, 각각의 인터페이스에 의해 지원되는 메소드의 이름과 파라미터, 적절한 도움말 파일 등의 정보를 제공하게 된다.

잘 만들어진 자동화 컨트롤러는 객체의 타입 라이브러리를 이용해서 early 바인딩을 지원하도록 제작해야 한다. 즉, 타입 라이브러리를 이용해서 지원되는 데이터 형을 가변형으로의 변환 과정이 없게 직접 사용할 수 있도록 하는 것이 핵심이다.

Late 바인딩과 early 바인딩을 모두 지원하게 하기 위해서는 듀얼 인터페이스를 사용해야 하는데 이때 파라미터로 사용되는 데이터 형은 반드시 자동화 호환(automation compatible)해야 한다. 이는 다른 말로 레코드와 같이 사용자가 정의한 데이터 형은 쓸 수 없다는 의미이다. 만약, 자동화 객체에 레코드를 파라미터로 사용하려면 데이터 멤버를 몇 개로 분리해야 한다.

## ● 기본적인 구현 방법

그러면, 지금까지의 대략적인 설명을 바탕으로 간단한 코드 예제로 late 바인딩과 early 바인딩을 설명하겠다. 여기서는 ISample 이라는 인터페이스가 있고, 이 인터페이스의 CoClass 가 CoSample, OLE 자동화 객체의 ProgID 를 'Sample.Application'이라고 가정하고 코드를 설명하도록 한다.

### 1. Late 바인딩

가장 느린 방법으로, 앞에서 예제를 통해서 설명했으므로 잘 이해했을 것으로 믿는다. 사용하기가 매우 쉽다는 장점이 있다. 앞의 가정에 의해서 late 바인딩 코드를 작성하면 다음과 같이 하면 된다.

```
var
```

```
MyServer: OleVariant;
```

```

begin
    MyServer := CreateOleObject('Sample.Application');
    MyServer.SampleMethod;
end;

```

이렇게 하면, 내부적으로는 IDispatch 인터페이스를 통해 호출을 하게 되는데 IUnknown 의 QueryInterface 로 위치를 찾기 위해 여러 인터페이스를 검색하게 되고, 인터페이스와 메소드의 위치를 찾게 되면 인덱스를 서버가 클라이언트로 넘겨 준다. 클라이언트는 내부적으로 MyServer.Invoke(인덱스)의 형태로 호출하여 동작한다.

## 2. Early 바인딩 – IDispatch 인터페이스

Early 바인딩에는 듀얼 인터페이스를 사용하는 방법과 IDispatch 인터페이스를 이용한 방법의 2 가지가 있다. 사용하는 방법은 간단하다. 기본적으로 인터페이스에 대한 선언부가 있는 유닛을 uses 절에 추가하고 다음과 같이 코딩하면 된다.

```

var
    MyServer: ISample;
begin
    MyServer := CreateOleObject('Sample.Application') as ISample;
    MyServer.SampleMethod;
end;

```

이렇게 하면, 컴파일할 때 메소드와 클라이언트에 메소드 수 등의 정보를 넘겨줄 수 있으므로 클라이언트는 컴파일 시에 주어진 메소드 인덱스를 이용해서 IDispatch 인터페이스의 Invoke 메소드를 호출하면 된다.

## 3. Early 바인딩 – VTable/듀얼 인터페이스

이 방법이 가장 빠른 방법으로, 델파이에서 타입 라이브러리를 이용해서 직접 지원하고 있는 방법이다. 사실 2 번의 방법과 속도 차이는 거의 없다. 사용 방법은 다음과 같다.

```

var
    MyServer: ISample;
begin
    MyServer := CoSample.Create;

```

```
MyServer.SampleMethod;  
end;
```

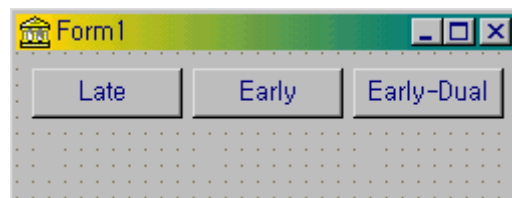
이때, CoSample 이 정의된 타입 라이브러리의 선언부 유닛이 uses 절에 추가되어 있어야 함은 물론이다. 듀얼 인터페이스는 이와 같이 타입 라이브러리 정보를 이용해서 클라이언트가 서버 객체의 주소를 직접 얻을 수 있도록 한다. 즉, 인터페이스를 질의할 필요도 없고, Invoke 메소드도 호출할 필요도 없으므로 효율적인 것이다. 듀얼 인터페이스는 이렇게 직접적인 접근 이외에도 1 번의 방법처럼 전통적인 가변형을 이용한 접근도 가능하다.

### ● 실전 예제

그렇다면, late 바인딩과 early 바인딩을 이용해서 같은 OLE 자동화 컨트롤러를 이용해서 반복적인 작업을 하면 얼마나 차이가 날지 무척 궁금할 것이다.

예제를 통해서 워드 객체에 작업을 앞에서 작성한 방식의 late 바인딩과 early 바인딩을 이용한 방법을 사용하여 이들을 처리하는데 걸리는 시간을 비교해 보도록 하자. 이 예제를 한번 작성하고 나면, 다른 OLE 자동화 컨트롤러 어플리케이션을 만들더라도 쉽게 early 바인딩을 구현할 수 있을 것이다.

먼저 간단하게 다음과 같이 버튼 3 개와 라벨 컴포넌트 3 개를 폼에 올려 놓자. 여기에서 라벨 컴포넌트의 캡션을 지우고, 각 버튼의 캡션을 다음과 같이 설정한다.



그러면, 이들 3 개 버튼의 OnClick 이벤트 핸들러를 작성하자. Late 버튼은 late 바인딩을 이용할 것이고, Early 버튼은 IDispatch 인터페이스의 early 바인딩을 이용할 것이다. 마지막으로 Early-Dual 버튼을 클릭하면 듀얼 인터페이스를 사용한다.

이를 위해서 먼저 타입 라이브러리를 import 할 필요가 있다. Project|Import Type Library 메뉴를 선택하면 나타나는 대화 상자에서 Microsoft Word 8.0 Object Library 를 선택하고 생성될 \_TLB.pas 파일을 저장할 패스를 Unit dir name 에디트 박스에 지정하고 OK 버튼을 클릭하면 Word\_TLB.pas, Office\_TLB.pas, VBIDE\_TLB.pas 파일이 디렉토리에 생성될 것이다.

프로젝트의 uses 절에 Word\_TLB.pas, ComObj.pas 유닛을 추가하고 late 바인딩을 위해 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    MsWordOleVariant: OleVariant;
    i, StartTick, StopTick: Integer;
begin
    MSWordOleVariant := CreateOleObject('Word.Application');
    StartTick := GetTickCount;
    for i := 1 to 10 do
    begin
        MSWordOleVariant.UserName := 'Sample';
        MSWordOleVariant.UserName := '';
        MSWordOleVariant.StatusBar := 'Sample';
        MSWordOleVariant.StatusBar := '';
    end;
    StopTick := GetTickCount;
    Label1.Caption := IntToStr(StopTick - StartTick);
    MSWordOleVariant.Quit(wdDoNotSaveChanges);
    MSWordOleVariant := UnAssigned;
end;

```

Late 바인딩의 방법은 앞에서 소개한 워드, 엑셀 제어 방법과 기본적으로 동일하다. CreateOleObject 메소드를 이용하며, 생성된 객체를 OleVariant(또는 Variant) 형의 변수에 저장한다. 그리고, 이를 이용하여 접근이 가능하다. 시간은 GetTickCount 함수를 이용하여 백만분의 1 초 단위로 측정하며, UserName 과 StatusBar 프로퍼티의 내용을 10 차례에 걸쳐 변경하는 속도를 쥔다.

마지막으로 워드 객체를 닫는데, 원래의 Quit 프로시저에는 3 개의 파라미터를 가지게 되어 있지만 가변형을 사용하므로 이렇게 데이터 형에 연연하지 않고 생략된 파라미터를 이용하여 메소드 호출이 가능하다.

Early 바인딩을 위한 Button2 의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```

procedure TForm1.Button2Click(Sender: TObject);
var
    MsWordInterface: _Application;
    i, StartTick, StopTick: Integer;
    wdSaveOptions, FormatOpt, Routing: OleVariant;
begin

```

```

MSWordInterface := CreateOleObject('Word.Application') as _Application;
wdSaveOptions := wdDoNotSaveChanges;
FormatOpt := 0;
Routing := 0;
StartTick := GetTickCount;
for i := 1 to 10 do
begin
    MSWordInterface.UserName := 'Sample';
    MSWordInterface.UserName := '';
    MSWordInterface.StatusBar := 'Sample';
    MSWordInterface.StatusBar := '';
end;
StopTick := GetTickCount;
Label2.Caption := IntToStr(StopTick - StartTick);
MSWordInterface.Quit(wdSaveOptions, FormatOpt, Routing);
end;

```

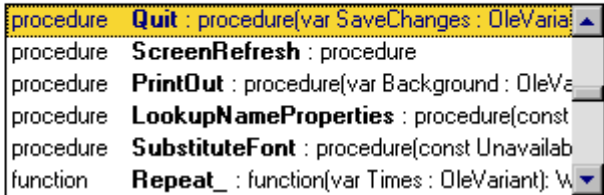
어플리케이션 인터페이스를 저장할 변수를 \_Application 인터페이스로 선언한다. 그리고, 이를 CreateOleObject 함수를 이용하여 OLE 객체를 생성한 뒤 as 연산자를 이용하여 \_Application 으로 형변환하여 변수에 저장하는 것이 중요한 키가 된다.

나머지 부분은 거의 동일한데, 아마도 코딩을 하다 보면 다음과 같이 코드 인사이트(Code Insight) 기능이 동작하는 것을 볼 수 있을 것이다. 이는 런타임이 아닌 컴파일 타임에서 데이터 형에 대한 검사를 하기 때문에 가능한 것이다.

```

    MSWordInterface.UserName := TempName;
    MSWordInterface.StatusBar := TempStatus;
    MSWordInterface.Q
end;
end.

```



또 한가지 눈여겨 보아야 할 것은, 마지막 부분에 Quit 메소드를 호출할 때 데이터 형에 맞도록 OleVariant 타입의 변수를 3 개 선언하고, 이들을 파라미터로 모두 사용해야만 컴파일 이 된다는 점이다. 즉, early 바인딩은 선언된 데이터 형을 그대로 사용해서 호출하지 않으면 안되는 것이다.

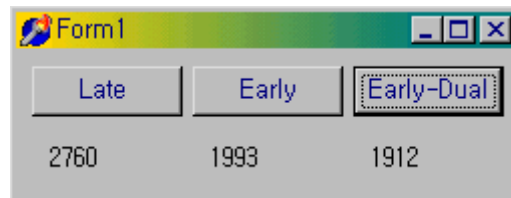
듀얼 인터페이스를 이용하는 경우도 거의 동일하다. 다만, 처음 선언부만 차이가 나는데

Button3 의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button3Click(Sender: TObject);  
var  
    MSWordInterface: _Application;  
    i, StartTick, StopTick: Integer;  
    wdSaveOptions, FormatOpt, Routing: OleVariant;  
begin  
    MSWordInterface := CoApplication_.Create;  
    ... (후략)
```

생략한 부분의 코드는 Button2 의 경우와 거의 동일하다. 참고로 인터페이스의 이름인 \_Application 이나 CoClass 이름인 CoApplication\_과 같이 밑줄이 있는 이름 들은 파스칼의 이름 규칙에 따라 다른 예약어와 겹칠 가능성이 있는 이름을 델파이가 변경한 것이라는 것을 알아두도록 하자.

자 그럼, 이제 결과를 보도록 하자. 이 프로그램을 컴파일하고 실행한 후, 버튼을 차례로 누르면 다음과 같은 결과를 볼 수 있을 것이다.



아마도 이 결과는 시스템에 따라서 조금씩은 차이가 날 것이다. 결과를 어떻게 생각하는가? 솔직히 필자는 이보다 훨씬 더 차이가 날 것으로 생각했는데 의외로 큰 차이가 나지 않은 결과이다. 어쨌든 early 바인딩이 더 빠르고, 듀얼 인터페이스가 미세하게나마 가장 빠르다는 것은 확실하다.

여기서 수행 성능에 큰 차이가 나지 않은 이유는 우리가 수행한 메소드가 단순한 문자열을 이용하여 값을 설정하는 정도의 간단한 것이기 때문이다. 또한, 위드는 기본적으로 OLE 자동화를 비주얼 베이직을 염두에 두고 제작되었기 때문에, 대부분의 파라미터가 OleVariant 데이터 형을 사용하도록 되어 있어 early 바인딩에 의한 성능 향상을 대폭 증가시키지는 못한다. 그렇지만, 만약에 필요한 OLE 자동화 서버를 작성할 때 직접 듀얼 인터페이스로 구현하여 이를 사용할 때 late 바인딩과 early 바인딩을 이용한 클라이언트 어플리케이션을 비교해보면 수행 성능에 엄청난 차이가 존재한다는 것을 쉽게 느낄 수 있을 것이다.

## 정 리 (Summary)

이번 장에서는 OLE 가 처음 나올 때 중시하던 복합 문서를 위한 OLE 도큐먼트(액티브 도큐먼트)를 OleContainer 컴포넌트를 이용해서 지원하는 방법과 OLE 자동화 컨트롤러 어플리케이션을 제작하여 여러가지 OLE 자동화 서버를 제어하는 방법에 대해서 알아보았다.

특히 델파이에서 OLE 자동화를 이용해서 오피스 객체를 사용하는 것은, 실제로 마이크로소프트에서 발표한 오피스 개발자 에디션(Microsoft Office Developer Edition)의 성능과 같거나, 델파이의 성능을 이용하면 더 나은 수행 성능을 보여주는 것으로 알려져 있다. 그러므로, 이를 잘 활용한다면 보다 통합된 어플리케이션을 개발하는데 커다란 도움이 될 것이다. 다음 장에서는 OLE 자동화 서버를 작성하는 방법과 타입 라이브러리와 듀얼 인터페이스에 대해서 보다 심도 있게 다룰 것이다.



# OLE 자동화 서버의 작성

## (Creating OLE Automation Servers)

자동화 서버는 자동화 컨트롤러라고 불리는 클라이언트 어플리케이션에 메소드를 노출시키는 어플리케이션을 말한다. 컨트롤러는 델파이 뿐만 아니라, 비주얼 베이직이나 C++ 과 같은 다른 도구로 작성할 수도 있다.

이미 26 장에서 COM 객체 서버를 작성하고 이를 이용하는 방법을 소개한 바 있지만, 여기서는 IDispatch 인터페이스에 기초를 둔 자동화 서버를 이용하게 된다. 자동화 서버는 프로세스간 통신의 방법으로도 최근에 가장 흔히 사용하는 것이므로 여기에 대해서는 잘 알아두는 것이 좋을 것이다.

### 자동화 객체의 제작

오브젝트 파스칼에서 자동화 객체는 TAutoObject 클래스에서 상속받아서 작성하게 된다. TAutoObject 객체는 자동화 프로토콜을 지원하며, 이를 이용하여 어플리케이션에서 사용할 수 있다. 자동화 객체는 자동화 객체 위저드를 이용해서 쉽게 만들 수 있다.

자동화 객체를 생성하기 전에, 먼저 사용할 프로젝트 객체를 정해야 한다. 여기에서는 File|New 메뉴를 선택하고, ActiveX 탭에서 ActiveX Library 를 이용하도록 한다. 물론 프로젝트는 이런 액티브 X DLL 라이브러리가 아니어도 상관이 없다.

자동화 객체 위저드를 이용하는 방법은 File|New 메뉴를 선택하고, ActiveX 탭에서 Automation Object 를 선택한다. ComObject 위저드와 마찬가지로 Automation Object 위저드에서도 클래스의 이름과 인스턴싱 type, 쓰레딩 모델을 선택해야 한다. 여기에 대해서는 26 장의 내용을 참고하기 바란다.

그 밖에 Generate event support code 옵션을 선택할 수 있는데, 이 옵션을 선택하면 위저드가 자동화 객체의 이벤트를 처리하기 위해 분리된 인터페이스를 구현한다는 뜻이다.

이들 항목을 모두 선택하고 OK 버튼을 클릭하면 현재 프로젝트에 자동화 객체가 추가되면서 타입 라이브러리가 열리는 것을 볼 수 있을 것이다. 타입 라이브러리에서 프로퍼티와 인터페이스의 메소드를 설정하면 된다.

기본적으로 델파이의 자동화 객체는 듀얼 인터페이스를 구현한다. 그러므로, IDispatch 인터페이스를 통한 late(런타임) 바인딩과 VTable 을 통한 early(컴파일-타임) 바인딩을 모두 지원하게 할 수 있다.

### 어플리케이션 프로퍼티, 메소드, 이벤트의 설정

타입 라이브러리에 담겨진 정보는 호스트 어플리케이션이 주어진 객체가 어떤 기능을 할 수 있는지를 알 수 있도록 해준다. 자동화 서버를 작성할 때에는 타입 라이브러리 정보가 어플리케이션에 리소스로서 자동으로 컴파일 된다.

- 프로퍼티 노출

프로퍼티는 객체의 색상이나 폰트와 같이 객체의 상태에 대한 정보를 설정하거나, 반환하는 멤버이다. 예를 들어, 버튼 컨트롤에는 다음과 같은 프로퍼티가 선언되어 있다.

property Caption: WideString;

자동화에 대한 프로퍼티를 노출시키려면 타입 라이브러리 에디터에서 자동화 객체에 대한 인터페이스를 선택하고, 툴바의 Property 버튼을 클릭하거나 이 버튼의 옆의 화살표를 클릭하고 노출한 프로퍼티의 type 을 클릭한다. 그리고, Attributes pane 에서 프로퍼티의 이름을 지정하고, Parameters pane 에서 프로퍼티의 리턴 type 을 지정하고 적당한 파라미터를 추가하면 된다. 그리고, Refresh 버튼을 클릭한다.

이렇게 하면, 프로퍼티에 대한 정의와 구현부의 뼈대 코드가 자동화 객체의 유닛 파일에 추가될 것이다. 마지막으로 만들어진 뼈대 코드에 프로퍼티의 구현을 위한 코드를 추가하면 된다.

- 메소드 선언

메소드는 프로시저나 함수일 수 있다. 메소드를 노출시키려면 타입 라이브러리 에디터에서 인터페이스를 선택하고 Method 버튼을 클릭한다. 그리고, Attributes pane 에서 메소드의 이름을 지정하고 Parameters pane 에서 메소드의 리턴 type 과 적절한 파라미터를 추가한 후 Refresh 버튼을 클릭한다.

이렇게 하면, 프로퍼티와 마찬가지로 자동화 객체 유닛 파일에 메소드에 대한 선언부와 구현부의 뼈대 코드가 추가된다. 마지막으로 여기에 실제 기능을 구현할 코드를 삽입하면 된다.

- 이벤트 노출

자동화 객체에 이벤트를 노출시키려면 처음에 자동화 객체 위저드에서 Generate event support code 옵션을 체크해야 한다. 이렇게 하면, Events 인터페이스를 포함한 자동화 객체를 생성한다.

그리고, 타입 라이브러리 에디터에서 자동화 객체에 대한 Events 인터페이스를 선택한다.

여기에서 Method 버튼을 클릭하고 Attributes pane 에서 이벤트의 적절한 이름을 지정한다. 그리고, Refresh 버튼을 클릭하면 이벤트에 대한 선언부와 뼈대 코드가 자동화 객체 유닛 파일에 삽입될 것이다.

코드 에디터에서 TAutoObject 에서 상속받은 이벤트 핸들러를 다음과 같은 형식으로 추가한다.

```
unit ev;
```

```
interface
```

```
uses
```

```
    ComObj, AxCtrls, ActiveX, Project1_TLB;
```

```
type
```

```
    TMyAutoObject = class (TAutoObject, IConnectionPointContainer, IMyAutoObject)
```

```
private
```

```
    ...
```

```
public
```

```
    procedure Initialize; override;
```

```
    procedure EventHandler; {이벤트 핸들러}
```

그리고, Initialize 메소드의 후반부에 다음과 같이 이벤트를 이벤트 핸들러에 연결한다.

```
procedure TMyAutoObject.Initialize;
```

```
begin
```

```
    inherited Initialize;
```

```
    FConnectionPoints:= TConnectionPoints.Create(Self);
```

```
    if AutoFactory.EventTypeInfo <> nil then
```

```
        FConnectionPoints.CreateConnectionPoint (AUtoFactory.EventIID,  
            ckSingle, EventConnect);
```

```
        OnEvent = EventHandler; {이벤트 핸들러와 이벤트를 연결한다}
```

```
end;
```

VCL 객체를 이용하는 자동화에 대한 이벤트를 노출시키는 것은, 다음과 같이 이벤트를 변경하여 설정한다.

```
procedure TMyAutoObject.EventHandler;
```

```
begin
```

```
    if FEvents <> nil then FEvents.MyEvent;
```

```
    {이벤트 핸들러를 호출시킨다}
```

```
end;
```

## 자동화 객체에서 이벤트 처리하기

텔파이 4에서의 자동화 객체 위저드에서 Generates event code 옵션을 체크하면 기본적인 이벤트 코드를 생성해준다.

이벤트를 처리하기 위해서는 처리할 이벤트의 종류에 따라 다른 인터페이스와 함께 액티브 X IConnectionPoint 와 IConnectionPointContainer 인터페이스를 지원해야 한다.

IConnectionPoint 인터페이스는 객체에 대한 outgoing 포인터를 노출하도록 허용하는 역할을 한다. 이러한 포인터를 이벤트 싱크(event sink)라고 한다. IConnectionPointContainer 인터페이스는 객체에 의해 제공되는 커넥션 포인트(connection point)를 나열하고, 호출자가 여기에서 적절한 커넥션 포인트를 찾을 수 있도록 해준다. 이벤트가 발생하면, 커넥션을 이용해서 이벤트를 처리하는 인터페이스를 호출하게 되는 이벤트 소스와 인터페이스를 실제 구현하는 이벤트 싱크를 연결하는 것이다. 싱크는 이벤트의 세트에 대한 멤버 함수들을 구현한다. 여기에 대해서는 30장에서 더욱 자세히 다루게 될 것이다.

## 자동화 인터페이스

텔파이 위저드는 듀얼 인터페이스를 디폴트로 구현한다. 즉, 자동화 객체가 IDispatch 인터페이스를 통해 런타임에서 late 바인딩을 지원하며, 직접 객체의 VTable 의 멤버 함수를 호출할 수 있도록 하여 컴파일 시의 early 바인딩을 지원한다.

### ● 듀얼 인터페이스 (Dual interface)

듀얼 인터페이스는 동시에 사용자 정의 인터페이스와 dispinterface 를 지원한다.

VTable 인터페이스를 위해서 컴파일러는 형 검사를 실시하고, 보다 정확한 에러 메시지를 제공하며, 데이터 형 정보를 얻을 수 없는 자동화 컨트롤러를 위해 dispinterface 가 객체에 런타임에서 접근할 수 있도록 허용한다.

듀얼 인터페이스 서버를 in-process 서버로 구현할 경우에는 vtable 인터페이스를 통해 보다 빠르게 접근할 수 있도록 하며, out-of-process 서버의 경우에는 COM 이 vtable 인터페이스와 dispinterface 에 대한 데이터를 타입 라이브러리에서 제공하는 정보에 따라 마샬링 해준다.

듀얼 인터페이스 vtable 의 처음 3 개의 엔트리는 IUnknown 인터페이스의 것이며, 그 다음 4 개의 엔트리는 IDispatch 인터페이스를 지원한다. 나머지 내용이 사용자 정의 인터페이

스의 메소드에 직접 접근할 때 사용되는 COM 엔트리이다.

- 디스패치 인터페이스 (Dispatch interface)

자동화 컨트롤러는 COM 의 IDispatch 인터페이스를 이용하여 COM 서버 객체에 접근한다. 컨트롤러는 먼저 객체를 생성하고, 객체의 IUnknown 인터페이스를 이용하여 IDispatch 인터페이스에 대한 포인터를 질의한다. IDispatch 인터페이스는 내부적으로는 디스패치 ID(dispID)를 사용하여 메소드나 프로퍼티를 추적한다. dispID 는 인터페이스 멤버에 대한 유일한 식별자로 쓰인다. IDispatch 를 이용하여 컨트롤러는 객체에 대한 데이터 형 정보를 얻게 되고, 이를 지정된 dispID 에 매핑하여 사용하게 된다. dispID 는 런타임에서 IDispatch 인터페이스의 GetIDsOfNames 메소드를 이용하여 얻을 수 있다.

일단 dispID 를 얻게 되면, 컨트롤러는 IDispatch 인터페이스의 Invoke 메소드를 이용하여 적절한 코드를 실행하게 된다. 이 과정에서 프로퍼티나 메소드의 파라미터 들을 Invoke 메소드의 파라미터로 패키징한다. Invoke 메소드는 고정된 컴파일 타임 signature 를 가지고 있으며 호출하는 인터페이스 메소드의 파라미터의 수와 내용이 어떤 것이든 받아들인다.

- 사용자 정의 인터페이스 (Custom interface)

사용자 정의 인터페이스는 클라이언트가 vtable 의 나열된 순서에 따라서 메소드를 호출하게 되며, 파라미터에 대한 데이터 형에 대한 정보를 호출자가 알고 있다는 기본 과정에서 출발하게 된다. VTable 은 객체의 프로퍼티와 메소드에 대한 주소를 모두 나열하게 되는데, 객체가 IDispatch 인터페이스를 지원하지 않을 경우에는 객체의 IUnknown 인터페이스를 따른다.

클라이언트가 객체에 대한 타입 라이브러리를 얻을 수 있으면, IDispatch 인터페이스의 dispID 를 가지고 직접 vtable 의 오프셋(offset)에 직접 바인딩하게 된다. 이런 컴파일 타임 바인딩은 객체의 메소드나 프로퍼티를 직접 객체의 vtable 로 호출하게 된다. 그러므로, Invoke 나 GetIDsOfNames 메소드를 호출할 필요가 없다.

## 데이터 마샬링 (Marshaling data)

out-of-process 또는 원격 서버인 경우에는 COM 이 어떤 방식으로 데이터를 마샬링하는지 이해하고 있어야 한다. IDispatch 인터페이스를 사용하거나, 서버를 제작할 때 타입 라이브러리를 작성하고 인터페이스를 OLE 자동화 플래그로 설정하면 이를 자동으로 지원할 수 있다. COM 은 자동화 호환 데이터 형에 대해서는 타입 라이브러리를 이용할 경우 프록시와 스텝을 설정하여 데이터를 마샬링할 수 있다.

마샬링을 직접 지원하고자 하면 IMarshal 인터페이스의 모든 메소드를 구현해야 하는데, 이

를 커스텀 마샬링이라고 한다.

- 자동화 호환 데이터 형 (Automation compatible types)

함수의 결과와 파라미터의 데이터 형이 듀얼과 dispatch 인터페이스인 경우에는 반드시 자동화 호환 데이터 형이어야 한다. OLE 자동화 호환 데이터 형으로는 다음과 같은 것들이 있다.

1. SmallInt, Integer, Single, Double, WideString 과 같은 단순 데이터 형. 자세한 것은 도움말을 참고하기 바란다.
2. 타입 라이브러리에 정의된 열거형(Enumeration type). OLE 자동화 호환 열거형은 32 비트 값으로 저장되며, 정수형으로 취급된다.
3. OLE 자동화에 문제가 없는 IDispatch 에서 상속받은 인터페이스 데이터 형
4. 타입 라이브러리에 정의된 Dispinterface 데이터 형
5. IFont, IStrings, IPicture 등은 각각 TFont, TStrings, TPicture 와 같은 helper 객체와 매핑된다.

액티브 X 컨트롤과 액티브폼 위저드는 필요할 때 이런 helper 객체를 생성한다. 이런 helper 객체를 사용하려면 GetOleFont, GetOleStrings, GetOlePicture 와 같은 전역 루틴을 사용한다.

- 자동 마샬링에 대한 데이터 형 제한

인터페이스의 마샬링을 자동으로 지원하게 하기 위해서는 다음과 같은 제한 사항을 지켜야 한다. 자동화 객체를 타입 라이브러리 에디터를 이용해서 편집할 때에는 이런 제한점을 지키도록 도와준다.

1. 데이터 형은 플랫폼간 통신(cross-platform communication)에 호환되어야 한다.
2. 듀얼 인터페이스의 모든 멤버는 반드시 함수의 리턴값으로 HRESULT 를 넘겨야 한다.
3. 다른 값을 반환해야 하는 듀얼 인터페이스의 멤버는 이를 var 또는 out 파라미터로 지정해서 사용해야 한다.

- 커스텀 마샬링 (Custom marshaling)

보통의 경우에는 앞에서 설명한 제한 사항을 지키면서 COM 에서 지원하는 마샬링을 자동으로 사용하게 된다. 어쨌든, 마샬링의 수행 성능을 향상시키기 위해 커스텀 마샬링을 사

용하게 될 수도 있다.

## 자동화 서버의 등록

자동화 서버는 in-process 서버일 수도 있고, out-of-process 서버일 수도 있다.

in-process 서버(DLL, OCX)를 등록하려면, Run|Register ActiveX Server 메뉴를 선택하면 되며 이를 해제하려면 Run|Unregister ActiveX Server 메뉴를 이용하면 된다.

out-of-process 서버를 등록하려면 서버를 실행할 때 /regserver 커맨드 라인 옵션을 사용하면 된다. 커맨드 라인 옵션을 설정할 때에는 Run|Parameters.. 메뉴의 대화 상자를 이용한다. 또한 이를 해제할 때에는 /unregserver 커맨드 라인 옵션을 사용하면 된다.

## 어플리케이션의 테스트와 디버깅

자동화 서버의 테스트와 디버깅을 위해서는 먼저 Project|Options 대화 상자의 Compiler 탭을 이용하여 디버깅 정보를 사용하도록 설정하고, Tools|Debugger Options 대화 상자에서 Integrated Debugging 을 이용하도록 하면 된다.

모든 in-process 서버는 디버깅과 테스트를 위해 Run|Parameters 메뉴를 선택하고 여기에서 Host Application 박스에 자동화 컨트롤러의 이름을 입력하고 OK 를 클릭한 뒤에 이를 실행하면 된다. 디버깅 방법은 일반적인 어플리케이션과 마찬가지로 정지점(break point)를 이용하는 것이 보통이다.

## 타입 라이브러리 에디터의 활용

텔파이 4 에서 제공되는 타입 라이브러리 에디터는 이전 버전의 타입 라이브러리 에디터의 한계점을 많이 극복한 도구로, COM/CORBA 개발에 있어서 텔파이가 가지고 있는 최대의 강점이라고 할 수 있다. 타입 라이브러리는 액티브 X 컨트롤이나 서버에 노출되는 객체 클래스, 멤버 함수, 인터페이스와 데이터 형에 대한 정보를 포함하고 있는 파일을 말한다.

타입 라이브러리는 IDL(Interface Definition Language)나 ODL(Object Description Language)을 이용하여 스크립트를 작성한 후, 이를 컴파일하는 것이 전통적인 개발 방식이다. 그렇지만, 텔파이 4 가 제공하는 타입 라이브러리 에디터를 이용하면 쉽게 타입 라이브러리를 작성할 수 있다.

COM 객체나 액티브 X 컨트롤, 자동화 객체를 텔파이의 위저드를 이용하여 제작했으면, 타입 라이브러리 에디터는 기본적으로 파스칼 문법으로 된 타입 라이브러리의 유닛 파일을 생성해준다. 타입 라이브러리 에디터에서 인터페이스나 메소드, 프로퍼티 등을 추가하거나 업데이트하고 Refresh 버튼을 클릭하면 이런 변화가 파스칼 유닛 파일에 그대로 반영한다.

● 오브젝트 파스칼과 IDL 문법

디폴트로 타입 라이브러리 에디터의 Text 페이지에는 타입 정보에 대한 내용을 오브젝트 파스칼 문법이나 IDL 문법으로 볼 수 있다. 어떤 언어를 디폴트로 할 것인지를 결정하려면 Tools|Environment Options 메뉴의 Type Library 페이지에서 선택할 수 있다.

오브젝트 파스칼 어플리케이션과 마찬가지로 타입 라이브러리의 식별자(identifier) 역시 대소문자를 가리지 않으며, 255 자까지 사용할 수 있다.

또한, 오브젝트 파스칼 문법으로 정의된 타입 라이브러리 정보는 타입 라이브러리 에디터의 가장 우측의 툴 버튼인 Export to IDL 버튼을 클릭하면 쉽게 IDL 문법으로 전환할 수 있다. 이때 우측의 화살표 키를 클릭하면 COM 을 지원하는 MIDL 문법을 따르게 할 것인지, CORBA 를 지원하는 CORBA IDL 문법을 따르게 할 것인지를 결정할 수 있다.

1. 속성 스펙 (Attribute specifications)

오브젝트 파스칼은 타입 라이브러리에 속성 스펙을 포함하도록 확장되었다. 속성 스펙은 쉼표로 분리된 대괄호로 표현되며, 각각의 속성 스펙은 속성 이름과 값으로 구성된다. 다음에 속성의 이름과 사용 예에 대해서 나열하였다.

속성 이름	사용 예	적용 대상
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo
bindable	[bindable]	CoClass 멤버를 제외한 멤버
control	[control]	type library, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	anything
default	[default]	CoClass 멤버
defaultbind	[defaultbind]	CoClass 멤버를 제외한 멤버
defaultcollection	[defaultcollection]	CoClass 멤버를 제외한 멤버
defaultvtbl	[defaultvtbl]	CoClass 멤버
dispid	[dispid]	CoClass 멤버를 제외한 멤버
displaybind	[displaybind]	CoClass 멤버를 제외한 멤버
dllname	[dllname 'Helper.dll']	module typeinfo
dual	[dual]	interface typeinfo
helpfile	[helpfile 'c:\Whelp\Wmyhelp.hlp']	type library
helpstringdll	[helpstringdll 'c:\Whelp\Wmyhelp.dll']	type library



helpcontext	[helpcontext 2005]	CoClass 멤버, 파라미터 제외
helpstring	[helpstring 'payroll interface']	CoClass 멤버, 파라미터 제외
helpstringcontext	[helpstringcontext \$17]	CoClass 멤버, 파라미터 제외
hidden	[hidden]	파라미터 제외
immediatebind	[immediatebind]	CoClass 멤버를 제외한 멤버
lcid	[lcid \$324]	type library
licensed	[licensed]	type library, CoClass typeinfo
nonbrowsable	[nonbrowsable]	CoClass 멤버를 제외한 멤버
nonextensible	[nonextensible]	interface typeinfo
oleautomation	[oleautomation]	interface typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	CoClass 멤버를 제외한 멤버
propput	[propput]	CoClass 멤버를 제외한 멤버
propputref	[propputref]	CoClass 멤버를 제외한 멤버
public	[public]	alias typeinfo
readonly	[readonly]	CoClass 멤버를 제외한 멤버
replaceable	[replaceable]	CoClass 멤버, 파라미터 제외
requestededit	[requestededit]	CoClass 멤버를 제외한 멤버
restricted	[restricted]	파라미터 제외
source	[source]	모든 멤버
uidefault	[uidefault]	CoClass 멤버를 제외한 멤버
usesgetlasterror	[usesgetlasterror]	CoClass 멤버를 제외한 멤버
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' ]	type library, typeinfo (required)
vararg	[vararg]	CoClass 멤버를 제외한 멤버
version	[version 1.1]	type library, typeinfo

## 2. 인터페이스 문법

인터페이스 데이터 형 정보를 오브젝트 파스칼 문법으로 표현하면 다음과 같다.

```

interfacename = interface[(baseinterface)] [attributes]
    functionlist
    [propertymethodlist]
end;
```

예를 들어, 다음의 텍스트는 2 개의 메소드와 1 개의 프로퍼티를 가지고 있는 인터페이스의 선언부이다.

```
Interface1 = interface (IDispatch)
    [uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
    function Calculate(optional seed: Integer = 0): Integer;
    procedure Reset;
    procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
    function GetRange: Integer [propget, dispid $00000005]; stdcall;
end;
```

이를 적절한 IDL 문법으로 변경하면 다음과 같다.

```
[uuid (5FD36EEF-70E5-11D1-AA62-00C04FB16F42), version 1.0]
interface Interface1: IDispatch
{
    longCalculate([in, optional, defaultvalue(0)] long seed);
    void Reset(void);
    [propput, id(0x00000005)] void _stdcallPutRange([in] long Value);
    [propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

### 3. 디스패치 인터페이스 문법 (Dispatch interface syntax)

dispinterface 를 선언하기 위한 오브젝트 파스칼의 문법은 다음과 같다.

```
dispinterfacename = dispinterface [attributes]
    functionlist
    [propertylist]
end;
```

예를 들어, 다음의 텍스트는 앞에서 선언한 인터페이스에 대한 디스패치 인터페이스를 표현한 것이다.

```
MyDispObj = dispinterface
```

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
version 1.0,
helpstring 'MyObj 에 대한 디스패치 인터페이스']
function Calculate(seed: Integer): Integer [dispid 1];
procedure Reset [dispid 2];
property Range: Integer [dispid 3];
end;
```

여기에 해당하는 IDL 문법은 다음과 같다.

```
[uuid (5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
version 1.0,
helpstring("MyObj 에 대한 디스패치 인터페이스")]
dispinterface Interface1
{
    methods:
        [id(1)] intCalculate([in] int seed);
        [id(2)] void Reset(void);
    properties:
        [id(3)] int Value;
};
```

#### 4. CoClass 문법 (CoClass syntax)

CoClass 의 타입 정보에 대한 오브젝트 파스칼의 문법은 다음과 같은 형태를 가진다.

```
classname = coclass(interfacename[interfaceattributes], ...); [attributes];
```

예를 들어, 다음의 텍스트는 IMyInt 라는 인터페이스와 DmyInt 라는 dispinterface 에 대한 CoClass 를 선언한 것이다.

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
version 1.0,
helpstring 'CoClass',
appobject]
```

여기에 해당되는 IDL 문법은 다음과 같다.

```
[uuid (2MD36ABF-90E3-11D1-AA75-02C04FB73F42),  
version 1.0,  
helpstring ("CoClass"),  
appobject]  
coclass myapp  
{  
    methods:  
    [source] interface MyInt;  
    dispinterface DMyInt;  
};
```

#### 4. 그 밖에 ...

그 밖에도 열거형, 앨리어스, 레코드, 모듈 등의 여러가지 데이터 형에 대한 오브젝트 파스칼의 타입 라이브러리 정보에 대한 문법이 존재한다. 여기서 이들 모두에 대해서 다루는 것은 다소 지면 낭비라 생각되므로, 이 정도로 정리하도록 하겠다.

IDL 문법에 대한 내용은 COM/CORBA 에 대해 자세히 접근한 서적에서 참고하기 바라며, 오브젝트 파스칼 문법 부분은 델파이의 도움말에서 제공되는 내용을 참고하면 될 것이다.

#### ● 타입 라이브러리 에디터의 사용법

그러면, 간단한 타입 라이브러리 에디터의 사용방법을 익혀보도록 하자. 앞서 설명한 여러가지 문법 들은 실제로 작성할 필요가 없이, 타입 라이브러리 에디터를 이용하면 자동으로 생성되는 내용들이다.

타입 라이브러리에 인터페이스에 대한 정보를 추가하려면, 툴바에서 인터페이스 아이콘을 클릭한다. 그러면, 인터페이스가 좌측의 객체 리스트 pane 에 추가되면서 이름을 입력할 수 있게 될 것이다. 인터페이스에 적절한 이름을 쳐 넣는다. 이렇게 추가된 인터페이스는 디폴트로 설정된 속성을 가지게 되는데, 여기에서 적절한 속성 값들을 설정할 수 있다.

이제 인터페이스에 멤버 들을 추가할 차례이다. 우선 해당되는 인터페이스를 선택하고, 툴바에서 메소드를 추가할 때에는 메소드 아이콘을 프로퍼티를 추가할 때에는 프로퍼티 아이콘을 클릭한다.

추가된 멤버의 이름을 설정한다. 이들 멤버에는 기본적인 디폴트 설정 값이 Attributes 페이지에 나타나게 되는데, 이 내용을 멤버의 설계에 맞도록 수정한다.

물론, 이런 작업을 툴바를 클릭해서 하나하나 추가할 수도 있지만, Text 페이지에 파스칼 문법으로 직접 입력하는 것으로 이를 대신할 수도 있다.

일단 이런 식으로 멤버를 인터페이스에 추가하고 나면, 이들 멤버가 분리된 아이템으로 객체 리스트 pane(object list pane)에 나타날 것이다.

CoClass 역시 비슷한 방법으로 타입 라이브러리에 추가할 수 있다. 툴바에서 CoClass 아이콘을 클릭하고, 이름을 설정한 뒤에 Attributes 페이지에서 속성을 편집하면 된다. 여기에 멤버를 추가할 때에는 클래스가 지원할 인터페이스를 먼저 선택해야 하는데, Text 페이지에서 오른쪽 버튼을 클릭하면 선택가능한 인터페이스가 나열될 것이다. 여기에서 CoClass가 구현한 인터페이스를 골라서 더블 클릭하면 적절하게 설정이 된다.

그 밖에 열거형이나 앨리어스, 레코드 등을 추가하는 방법에 대해서는 도움말을 참고하기 바란다.

#### ● 타입 라이브러리의 배포

액티브 X 컨트롤 위저드를 이용해서 액티브 X 컨트롤을 제작할 경우, 타입 라이브러리는 자동으로 액티브 X 라이브러리 파일인 OCX 파일에 리소스로 포함되어 컴파일 된다. 그렇지만, 이를 분리된 .tlb 파일로 배포하는 것도 가능하다.

과거에는 자동화 어플리케이션의 경우 대부분 타입 라이브러리를 .tlb 파일에 저장하여 분리하여 배포하는 경우가 많았다. 그렇지만 델파이의 위저드를 사용할 경우 마찬가지로 .ocx, .exe 파일에 리소스로 포함되어 컴파일된다.

참고로 이런 형태로 컴파일 할 경우 액티브 X 라이브러리 하나에 여러 개의 타입 라이브러리가 포함될 수 있다. 독자적인 타입 라이브러리 파일을 배포하고자 한다면, 타입 라이브러리 에디터에서 이진 파일을 직접 저장할 수 있으므로 이를 이용할 수도 있다.

그렇지만, 쉽게 통합된 파일로 배포할 수 있는 것을 과거 방식대로 타입 라이브러리 파일을 분리해서 배포하는 것은 별로 권하고 싶지 않은 방법이다.

### 듀얼 인터페이스를 지원하는 자동화 서버의 제작

그러면, 듀얼 인터페이스를 지원하는 자동화 서버를 만들어 보자. 이번에는 액티브 X 라이브러리 DLL 파일이 아니라, 위드나 엑셀처럼 .exe 실행 파일로 된 out-of-process 서버로 만들어 보자.

먼저 New Application 메뉴를 선택해서 새로운 프로젝트를 연다. 그리고, 적당한 이름으로 저장을 한 뒤에 File|New 메뉴에서 ActiveX 탭의 Automation Object 아이콘을 더블 클릭한다. 나타나는 대화 상자에서 클래스 이름을 Sample 이라고 입력하고, 나머지 필드의 내용은 디폴트 값을 사용한다.

- 타입 라이브러리의 작성

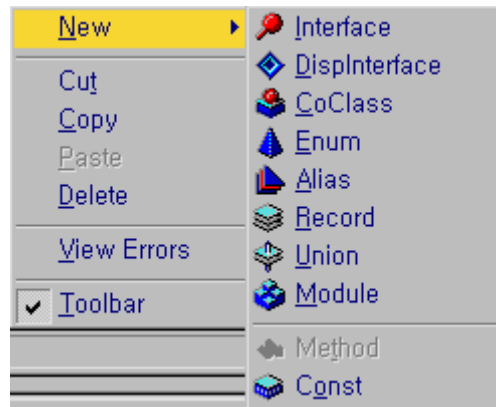
OK 를 선택하면 타입 라이브러리 에디터가 뜨는데, 여기에는 이미 CoClass 인 Sample 과 ISample 인터페이스가 이미 정의되어 있을 것이다.

여기에 우리가 구상한 인터페이스의 멤버를 추가하도록 한다.

이번에 작성할 자동화 서버는 같이 포함된 프로젝트의 품의 Canvas 를 이용하여 품에 지정된 모양의 도형을 크기에 맞추어 그리도록 할 것이다.

그러므로 여기에서는 도형의 모양을 열거형으로 TSampleShape 을 선언하고, 이 데이터 형의 프로퍼티인 Shape 와 도형의 폭과 높이를 결정하는 Width, Height 프로퍼티를 멤버로 가진다. 그리고, 지정된 프로퍼티에 맞추어 그림을 그리는 Paint 메소드와 품의 Canvas 를 지워주는 Clear 메소드로 구성한다.

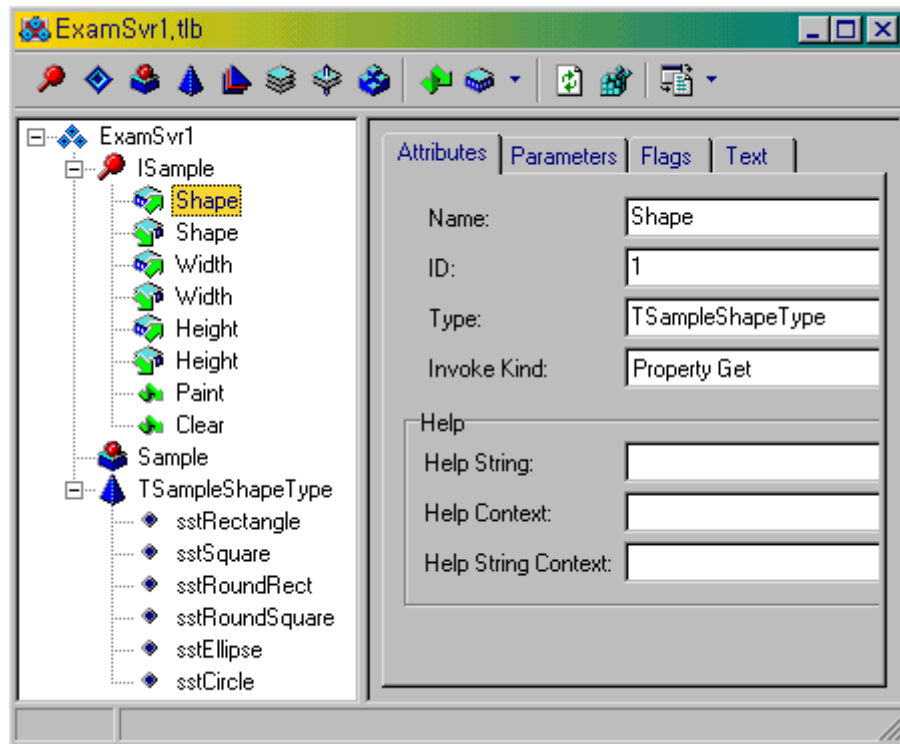
그러면 타입 라이브러리에서 이런 요구 사항에 맞추어 인터페이스를 정의해보자. 먼저 열거형을 정의해야 한다. 타입 라이브러리 에디터에서 삼각뿔 모양의 아이콘인 New Enumeration 메뉴를 클릭한다. 그러면 새로운 열거형이 추가되는데, 열거형의 이름을 TSampleShape 으로 설정한다. 이제 TSampleShape 을 선택하고, 오른쪽 버튼을 클릭하면 팝업 메뉴가 생성되는데, 여기서 New 에 마우스 커서를 가져가면 다음과 같은 보조 메뉴가 나타날 것이다. 여기에서 Const 를 선택하여 멤버를 하나씩 입력하도록 한다.



TSampleShape 의 멤버로 sstRectangle, sstSquare, sstRoundRect, sstRoundSquare, sstEllipse, sstCircle 을 순서대로 설정한다.

이제 프로퍼티를 추가할 차례이다. 이 프로퍼티 들은 읽기와 쓰기가 모두 가능해야 하므로 ISample 인터페이스를 선택한 후 간단히 New Property 아이콘을 클릭하면 된다. 만약 읽기나 쓰기 전용 프로퍼티를 원할 경우에는 New Property 아이콘 옆의 화살표 아이콘을 클릭하여 해당되는 것을 선택할 수 있다. 먼저 Shape 프로퍼티를 추가하고 이름을 설정한 뒤, Type 콤보 박스에 TSampleShape 를 선택한다. 그리고, Width 와 Height 를 각각 추가하고 Type 콤보 박스에 Integer 를 선택한다. 이렇게 함으로써 프로퍼티 설정은 모두 끝났다. 메소드 역시 New Method 아이콘을 클릭하여 간단히 추가할 수 있다. 여기서 추가할

메소드인 Paint 와 Clear 는 모두 파라미터와 반환값이 없는 프로시저 형으로 선언할 것이다. 파라미터나 반환값에 대한 속성을 설정할 때에는 Parameters 탭에서 설정하면 된다. 이렇게 타입 라이브러리를 모두 설정하고 나면 다음 그림과 같은 형태로 타입 라이브러리 에디터가 보일 것이다.



- Out-of-process 자동화 서버의 제작

이제 프로젝트 파일을 저장(ExamSvr1.dpr, U\_ExamSvr1.pas, U\_ExamSvr1Impl.pas)하고 타입 라이브러리 에디터를 종료하면 구현할 유닛의 뼈대 코드가 만들어지는데, 여기에 uses 절에 구현에 필요한 루틴인 SysUtils, Graphics, Dialogs 를 추가하고 프로젝트의 폼을 이용할 것이므로 폼의 유닛 파일(여기서는 U\_ExamSvr1)을 추가한다. 폼의 Width, Height 프로퍼티는 각 250 으로 설정하도록 하자. 그리고, 프로퍼티의 내용을 저장할 필드 변수인 FWidth, FHeight, FShape 를 private 섹션에 다음과 같이 선언하고, protected 섹션에 초기화를 위해 Initialize 메소드를 오버라이드하도록 한다.

```
unit U_ExamSvr1Impl;
```

```
interface
```

```
uses
```

ComObj, ActiveX, SysUtils, Graphics, Dialogs, ExamSvr1\_TLB, U\_ExamSvr1;

type

TSample = class(TAutoObject, ISample)

private

FShape: TSampleShapeType;

FHeight: Integer;

FWidth: Integer;

protected

function Get\_Height: Integer; safecall;

function Get\_Shape: TSampleShapeType; safecall;

function Get\_Width: Integer; safecall;

procedure Paint; safecall;

procedure Set\_Height(Value: Integer); safecall;

procedure Set\_Shape(Value: TSampleShapeType); safecall;

procedure Set\_Width(Value: Integer); safecall;

procedure Initialize; override;

procedure Clear; safecall;

end;

그리고, initialization 섹션의 내용은 이미 다음과 같이 작성되어 있을 것이다.

initialization

TAutoObjectFactory.Create(ComServer, TSample, Class\_Sample,  
ciMultiInstance, tmApartment);

여기서 Ctrl+ Shift+ C 를 클릭하여 클래스 완료 메뉴를 동작시키면 선언된 메소드의 뼈대 코드가 자동으로 생성될 것이다. 이 뼈대 코드에 실제 구현 내용을 코딩하면 된다.

먼저 Shape, Width, Height 와 같은 프로퍼티의 구현 방법은 컴포넌트 제작할 때와 마찬가지로 Get\_, Set\_ 메소드를 이용해서 값을 읽거나 설정하도록 하면 된다.

function TSample.Get\_Height: Integer;

begin

Result := FHeight;

end;



```
function TSample.Get_Shape: TSampleShapeType;
```

```
begin
```

```
    Result := FShape;
```

```
end;
```

... (중략)

```
procedure TSample.Set_Width(Value: Integer);
```

```
begin
```

```
    FWidth := Value;
```

```
end;
```

Paint 와 Clear 메소드는 폼의 Canvas 를 이용하여 일반적인 드로우 메소드로 구현하면 된다. Paint 메소드는 지정된 크기의 도형을 서버 폼에 그려주는 역할을 하며, Clear 메소드는 이를 지우는 역할을 한다.

```
procedure TSample.Paint;
```

```
var
```

```
    X, Y, W, H, S: Integer;
```

```
begin
```

```
    with Form1.Canvas do
```

```
    begin
```

```
        Brush.Style := bsSolid;
```

```
        Brush.Color := clBlue;
```

```
        W := FWidth;
```

```
        H := FHeight;
```

```
        if W < H then S := W else S := H;
```

```
        case FShape of
```

```
            sstRectangle, sstRoundRect, sstEllipse:
```

```
            begin
```

```
                X := 0;
```

```
                Y := 0;
```

```
            end;
```

```
            sstSquare, sstRoundSquare, sstCircle:
```

```
            begin
```

```
                X := (W - S) div 2;
```

```

        Y := (H - S) div 2;
        W := S;
        H := S;
    end;
end;
case FShape of
    sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);
    sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);
    sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);
    end;
end;
end;
end;

```

```

procedure TSample.Clear;
begin
    with Form1.Canvas do
        begin
            Brush.Style := bsSolid;
            Brush.Color := clMenu;
            Rectangle(0, 0, Form1.Width, Form1.Height);
        end;
    end;
end;

```

초기화를 담당하는 Initialize 메소드는 컴포넌트의 constructor 의 역할을 하는 메소드이다. 여기서는 다음과 같이 기본적인 초기값을 설정하도록 한다.

```

procedure TSample.Initialize;
begin
    FWidth := 100;
    FHeight := 100;
    FShape := sstRectangle;
end;

```

이것으로 엑셀이나 액세스와 같이 .exe 형태의 out-of-process 자동화 서버가 완성되었다. Out-of-process 자동화 서버는 그냥 .exe 파일을 배포하고, 사용자가 이를 실행할 때 /regserver 파라미터를 주는 것으로 간단히 레지스트리에 등록된다. 레지스트리에 등록되는 ProgID 는 일반적으로 실행파일의 이름과 CoClass 의 이름으로 이루어진다. 그러므로 여기서 작성한 예제의 경우 ExamSvr1.exe 파일이고, CoClass 는 Sample 이므로 'ExamSvr1.Sample'이 된다.

#### ● In-process 서버의 제작

이렇게 배포와 등록이 간편한 장점이 있는 out-of-proc 자동화 서버의 단점은 in-proc 자동화 서버에 비해 프로세스간 주소의 마샬링 작업이 필요하기 때문에 수행성능이 떨어지는 단점이 있다.

그렇다면, 이와 똑같은 자동화 서버를 in-proc 서버로 작성해서 이들의 성능을 비교하는 클라이언트 어플리케이션을 제작해 보도록 하자.

먼저 File|New 메뉴의 ActiveX 탭에서 ActiveX Library 아이콘을 더블 클릭하여 액티브 X 서버 DLL 을 작성하기 위한 프로젝트 파일을 생성한다. 그리고, 지금까지의 방법과 마찬가지로 자동화 객체 위저드를 이용하여 Sample2 라는 새로운 자동화 서버를 작성하도록 하자. 타입 라이브러리에서 앞서 설명한 ISample 과 똑같은 과정을 거쳐서 멤버를 생성한다. 그런데, 같은 이름을 가지지 않도록 ISample2, TSampleShape2, Sample2 와 같이 뒤쪽에 2 를 붙이도록 한다. ISample 과 똑같이 인터페이스를 작성하고 타입 라이브러리를 종료한 후 적당한 이름으로 프로젝트 파일을 저장(여기서는 ExamSvr2.dpr, U\_ExamSvrImpl2.pas)하면 ExamSvr1 과 마찬가지로 코드가 생성될 것이다.

구현 방법이 거의 똑같기 때문에 자세한 코드 설명은 생략하고, 다르게 구현되는 부분만 중점적으로 설명하겠다.

처음에 DLL 로 작성했기 때문에, ExamSvr1 과는 달리 폼이 따로 존재하지 않으므로 새로 추가해야 한다. New Form 메뉴를 선택하여 프로젝트에 폼을 추가하고, 유닛의 이름을 U\_ExamSvr2.pas 라는 이름으로 저장한다.

그리고, 자동화 서버가 시작할 때 폼이 생성되고 나타날 수 있도록 Initialize 메소드를 다음과 같이 오버라이드하여 작성한다.

```
procedure TSample2.Initialize;  
begin  
    FWidth := 100;  
    FHeight := 100;  
    FShape := sstRectangle;  
    Form1 := TForm1.Create(Application);
```

```

Form1.Width := 250;
Form1.Height := 250;
Form1.Show;
end;

```

이를 위해서는 uses 절에 Forms.pas 유닛을 추가해야 한다. 또한, 자동화 서버가 종료할 때에는 폼이 파괴되어야 하므로, destructor 인 Destroy 프로시저 역시 다음과 같이 오버라이드하여야 한다.

```

destructor TSample2.Destroy;
begin
    Form1.Free;
    inherited Destroy;
end;

```

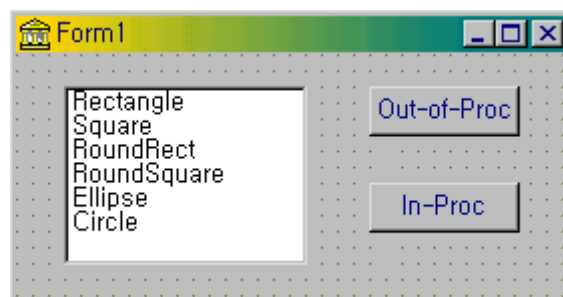
나머지 부분은 이름에 '2'를 추가하는 것 이외에는 모두 동일하므로 설명을 생략하도록 한다. 컴파일하고, 자동화 서버를 등록해야 하므로 Run|Register ActiveX Server 메뉴를 선택하여 자동화 서버를 사용할 수 있는 준비를 완료한다.

## ● 클라이언트 어플리케이션의 제작

이제 2 개의 동일한 역할을 하는 자동화 서버가 out-of-proc, in-proc 으로 모두 준비되었다. 그러면, 이들의 기능을 사용하는 클라이언트를 작성해보도록 하자.

먼저 New Application 메뉴를 선택한 후 다음과 같이 폼에 버튼 2 개와 라벨 2 개, 리스트 박스 1 개를 각각 올려 놓고, 버튼의 캡션을 각각 'Out-of-Proc', 'In-Proc'으로 설정한다. 라벨에는 백만분의 1 초 단위로 걸리는 시간을 나타낼 것이므로 캡션을 지우도록 한다.

또한, 리스트 박스에는 도형의 모양을 설정할 수 있도록 인터페이스의 TSampleShape 열거형의 순서에 맞도록 아이템을 설정한다. 이렇게 순서를 일치시키면 나중에 직접 ItemIndex 를 대입해도 문제가 없다(열거형은 0 부터 시작하는 일종의 정수형이다).



uses 절에 early 바인딩을 사용할 수 있도록 타입 라이브러리 파일(ExamSvr1\_TLB.pas, ExamSvr2\_TLB.pas)을 추가하고, ISample 과 ISample2 인터페이스를 담을 변수를 다음과 같이 전역 변수로 선언한다.

```
var
    Form1: TForm1;
    Sample: ISample;
    Sample2: ISample2;
```

폼의 OnCreate 이벤트 핸들러에서 이들 인터페이스를 생성하여 전역변수에 대입하고, 기본적으로 리스트 박스의 도형을 Rectangle 로 설정한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ListBox1.Selected[0];
    Sample := CoSample.Create;
    Sample2 := CoSample2.Create;
end;
```

자, 그러면 이제 이들을 테스트하는 버튼의 OnClick 이벤트 핸들러를 작성할 차례가 되었다. 지난 장에서와 마찬가지로 GetTickCount 함수를 이용하여 시간을 잴 것이다. 크기를 폭과 높이를 동일하게 1~200 까지 증가시키면서 그릴 때 걸리는 시간을 비교하도록 다음과 같이 코딩한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i, StartTick, StopTick: Integer;
begin
    Sample.Shape := ListBox1.ItemIndex;
    StartTick := GetTickCount;
    for i := 1 to 200 do
    begin
        Sample.Clear;
        Sample.Width := i;
        Sample.Height := i;
```

```

    Sample.Paint;
end;

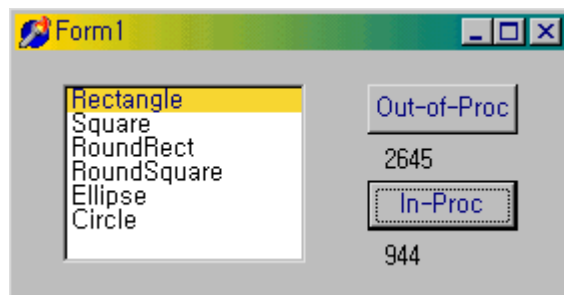
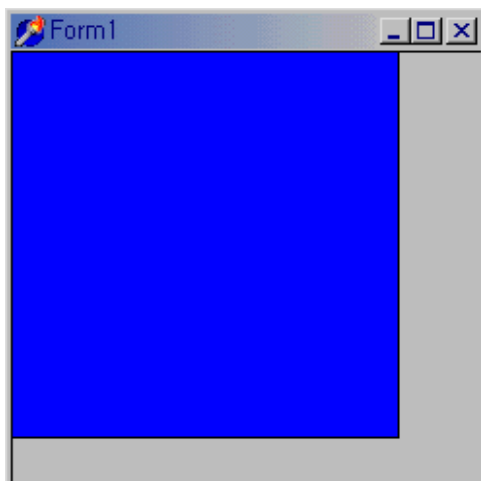
StopTick := GetTickCount;

Label1.Caption := IntToStr(StopTick - StartTick);
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    i, StartTick, StopTick: Integer;
begin
    Sample2.Shape := ListBox1.ItemIndex;
    StartTick := GetTickCount;
    for i := 1 to 200 do
    begin
        Sample2.Clear;
        Sample2.Width := i;
        Sample2.Height := i;
        Sample2.Paint;
    end;
    StopTick := GetTickCount;
    Label2.Caption := IntToStr(StopTick - StartTick);
end;

```

쉬운 내용이므로 따로 설명은 하지 않는다. 그러면, 이제 직접 실행을 해보자. 아마도 실행과 동시에 OnCreate 메소드에 의해 2 개의 서버 폼이 같이 뜰 것이다. 이들을 관찰할 수 있도록 위치를 옮긴 후, 버튼을 클릭하여 시간을 재보자 아마도 다음과 같은 결과를 보여줄 것이다.



결과는 in-proc 서버가 2 배 이상 빠르다. 클라이언트 어플리케이션을 종료하면 자동화 서버 들은 자동으로 종료된다. 이런 속도 차이는 드로우와 같이 느린 작업을 한 결과로 나타난 것이므로, 계산이나 호출이 빈번한 경우에는 이보다 큰 차이가 날 것이다.

## 문자열 리스트, 폰트 등의 객체 활용

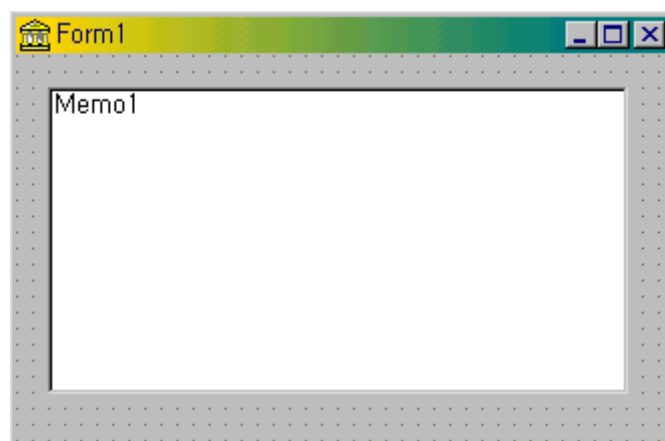
앞에서도 언급한 바 있지만, OLE 자동화를 이용할 때 주의할 점으로는 파라미터나 리턴 값으로 OLE 에 호환되는 데이터 형을 이용한다는 것이다.

그런데, 델파이의 문자열 리스트(string list), 폰트(font)나 그래픽 객체의 경우에는 여기에 해당되는 인터페이스를 이용해서 클라이언트와 서버 간의 통신이 가능하다. 물론 이 경우에는 지금까지 설명한 방법과는 다른 방법을 사용해야 하며, 델파이에서 지원하는 함수를 이용해야 한다.

그러면, 문자열 리스트와 폰트에 대한 정보를 주고받을 수 있는 자동화 서버와 클라이언트를 제작해 보도록 하자.

먼저 인터페이스로 폰트에 대한 인터페이스를 IFontSvr, 문자열 리스트에 대한 인터페이스를 IStringSvr 이라고 하고, 이들에 멤버로 프로퍼티인 Font 와 Items 를 각각 추가한다. 이들의 반환값으로는 IFontDisp 와 IStrings 를 사용할 것이다.

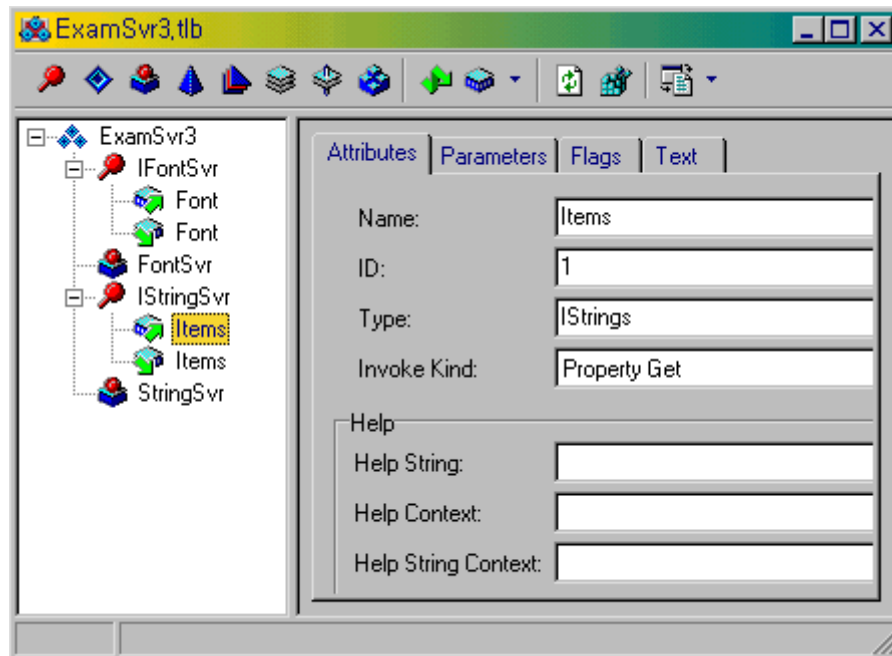
새로운 프로젝트를 시작하고, 폼에 다음과 같이 메모 컴포넌트를 하나 올려 놓는다. 메모 컴포넌트는 문자열 리스트를 클라이언트에서 받아서 보여 주고, 클라이언트로 메모 컴포넌트의 문자열 리스트를 보내는 등의 동작과 폰트의 정보를 반영하여 보여주기 위해서 올려 놓은 것으로 사실 클라이언트의 동작에는 영향을 미치지 않는다.



그리고, 필요한 자동화 객체가 CoFontSvr 과 CoStringSvr 의 2 개 이므로 2 번의 자동화 객체의 생성 작업이 필요하다. 먼저 IFontSvr 에 대한 작업을 하도록 하자. File|New 메뉴의 ActiveX 탭에서 Automation Object 아이콘을 더블 클릭한다. 클래스 이름으로

FontSvr 을 입력하고 OK 버튼을 클릭한다. 타입 라이브러리 에디터에서 IFontSvr 인터페이스를 선택하고, New Properties 버튼을 클릭하고 프로퍼티의 이름으로 Font 를 설정하고, Attributes 탭의 Type 콤보 박스에서 IFontDisp 를 선택한다. IFontDisp 가 TFont 클래스와 호환될 수 있는 역할을 하는 인터페이스이다.

타입 라이브러리 에디터를 일단 닫고 프로젝트 파일을 저장한 후, 다시 한번 File|New 메뉴를 선택한 뒤 Automation Object 아이콘을 더블 클릭하여 StringSvr 클래스를 추가한다. 같은 타입 라이브러리 에디터가 나타나면 IStringSvr 인터페이스를 선택하고, New Properties 버튼을 클릭한 후 프로퍼티의 이름을 Items 로 설정한다. Attributes 탭의 Type 콤보 박스에서 IStrings 를 선택한다. 이렇게 한 뒤의 타입 라이브러리 에디터의 형태는 다음 그림과 같을 것이다.



이렇게 하면 자동화 서버를 구현하는 유닛이 2 개가 추가되었을 것이다. 하나는 FontSvr 에 대한 것이고, 나머지 하나는 StringSvr 에 대한 것이다. 먼저 FontSvr 클래스를 구현하도록 하자. TFontSvr 클래스가 이미 만들어져 있겠지만, 다음과 같이 uses 절에 구현에 필요한 AxCtrls, Graphics, StdVCL 유닛과 ExamSvr3\_TLB 유닛, 그리고 폼을 이용할 것이므로 폼의 유닛인 U\_ExamSvr3 를 추가하고 선언부를 다음과 같이 수정하도록 한다.

```
unit U_ExamSvrImpl3;

interface

uses

    ComObj, ActiveX, StdVCL, AxCtrls, Graphics, ExamSvr3_TLB, U_ExamSvr3;
```



```

type
  TFontSvr = class(TAutoObject, IFontSvr)
  private
    FFont: TFont;
  public
    procedure Initialize; override;
    destructor Destroy; override;
    function Get_Font: IFontDisp; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
  end;

```

FFont 필드에 델파이의 TFont 에 해당되는 내용을 저장하고, 이를 IFontDisp 인터페이스에 맞도록 변환하는 처리를 Get\_Font, Set\_Font 메소드에서 하게 된다.

일단 Initialize 프로시저와 Destroy 프로시저를 다음과 같이 구현하여 폰트 객체를 생성하고, 이를 해제한다.

```

procedure TFontSvr.Initialize;
begin
  inherited Initialize;
  FFont := TFont.Create;
end;

```

```

destructor TFontSvr.Destroy;
begin
  FFont.Free;
  inherited Destroy;
end;

```

그리고, 핵심이 되는 Get\_Font, Set\_Font 메소드를 다음과 같이 구현하면 된다.

```

function TFontSvr.Get_Font: IFontDisp;
begin
  FFont.Assign(Form1.Memo1.Font);
  GetOleFont(FFont, Result);
end;

```

```

procedure TFontSvr.Set_Font(const Value: IFontDisp);
begin
    SetOleFont(FFont, Value);
    Form1.Memo1.Font.Assign(FFont);
end;

```

구현의 핵심이 되는 것이 GetOleFont, SetOleFont 함수이다. 이들 함수는 델파이의 AxCtrls.pas 유닛에 포함된 것으로 델파이의 TFont 와 IFontDisp 인터페이스를 서로 변환할 수 있도록 지원하는 역할을 한다.

마찬가지로 IStrings 나 IPicture 를 지원하는 GetOleStrings, SetOleStrings, GetOlePicture, SetOlePicture 함수를 이용할 수 있다.

TStringSvr 클래스의 구현 방법도 대동소이하다. 먼저, uses 절에서 Graphics 대신에 Classes 유닛을 추가하고 다음과 같이 선언한다.

```

unit U2_ExamSvrImpl3;

interface

uses

    ComObj, ActiveX, StdVCL, Classes, AxCtrls, ExamSvr3_TLB, U_ExamSvr3;

type

    TStringSvr = class(TAutoObject, IStringSvr)
    private
        FItems: TStrings;
    public
        procedure Initialize; override;
        function Get_Items: IStrings; safecall;
        procedure Set_Items(const Value: IStrings); safecall;
    end;

```

Destroy 메소드에 대한 선언부가 빠진 이유는 어떤 이유인지는 잘 모르겠으나, 생성한 TStringList 객체를 파괴하는 메소드를 추가하면, 상속된 Destroy 메소드에서 이를 파괴하기 위해 다시 접근하는 탓에 Access violation 에러가 발생한다. 그래서, 이 부분을 삭제하니 아무런 문제가 발생하지 않았다.

이들의 구현 방법은 다음과 같다.

```

procedure TStringSvr.Initialize;
begin
    inherited Initialize;
    FItems := TStringList.Create;
end;

```

```

function TStringSvr.Get_Items: IStrings;
begin
    FItems := Form1.Memo1.Lines;
    GetOleStrings(FItems, Result);
end;

```

```

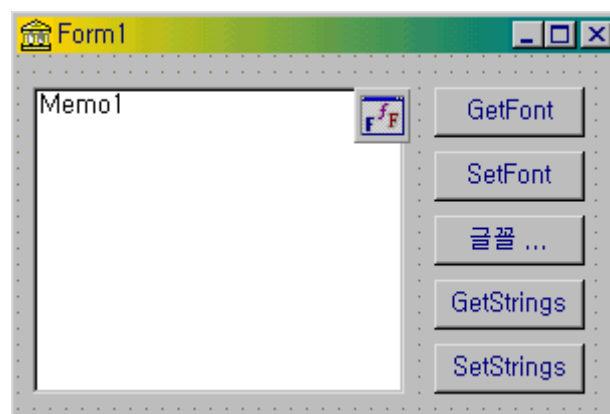
procedure TStringSvr.Set_Items(const Value: IStrings);
begin
    SetOleStrings(FItems, Value);
    Form1.Memo1.Lines.Assign(FItems);
end;

```

역시 핵심은 GetOleStrings 와 SetOleStrings 함수를 이용하여 IStrings 와 TStrings 간의 변환을 하는 부분이다. 이제 프로젝트를 컴파일하고 /regserver 옵션을 주고 실행하면 서버를 사용할 수 있게 된다.

그러면, 이제 이들을 이용하는 클라이언트 어플리케이션을 만들어 보자

새로운 프로젝트를 선택하고, 폼에 메모 컴포넌트 1 개와 버튼을 5 개와 TFontDialog 컴포넌트를 하나 올려 놓고 다음과 같이 캡션을 설정하도록 한다.



여기서 GetFont 버튼은 IFontSvr 의 폰트 정보를 메모 컴포넌트에 적용하고, SetFont 버튼

은 IFontSvr 의 Font 프로퍼티에 메모 컴포넌트의 폰트를 저장한다. ‘글꼴...’ 버튼은 메모 컴포넌트의 폰트를 변경하기 위한 목적으로 사용된다. 마찬가지로 GetStringS 버튼은 서버의 Items 프로퍼티에 저장된 문자열을 메모 컴포넌트로 불러오게 되며, SetStrings 버튼은 메모 컴포넌트의 문자열을 IStringSvr 의 Items 프로퍼티에 저장하는 역할을 한다.

먼저, IStringSvr 인터페이스와 IFontSvr 인터페이스, IFontDisp 와 IStrings 인터페이스를 저장할 전역 변수를 다음과 같이 선언하고, 폼의 OnCreate 이벤트 핸들러를 다음과 같이 작성하여 이들을 early 바인딩을 이용해서 대입한다.

```
var
```

```
Form1: TForm1;
```

```
FontSvr: IFontSvr;
```

```
StringSvr: IStringSvr;
```

```
FFont: IFontDisp;
```

```
FItems: IStrings;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    Memo1.Lines.Clear;
```

```
    FontSvr := CoFontSvr.Create;
```

```
    StringSvr := CoStringSvr.Create;
```

```
end;
```

물론 여기서 FFont 와 FItems 변수의 경우 임시 변수로 사용되는 역할을 하기 때문에 전역 변수로 선언하지 않고, 버튼의 OnClick 이벤트 핸들러에서 지역 변수로 선언해서 사용하는 것이 더 효율적인 사용 방법이다. 그렇지만, 이 예제에서 전역 변수로 선언한 이유는 예제 코드의 중복을 없애서 길이를 줄이고자 한 것이므로 이를 고려하기 바란다.

마지막으로, 앞에서 설명한 기능을 하는 5 개 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하면 된다.

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

```
begin
```

```
    FFont := FontSvr.Font;
```

```
    SetOleFont(Memo1.Font, FFont);
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```

begin
    GetOleFont(Memo1.Font, FFont);
    FontSvr.Font := FFont;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    if FontDialog1.Execute then Memo1.Font := FontDialog1.Font;
end;

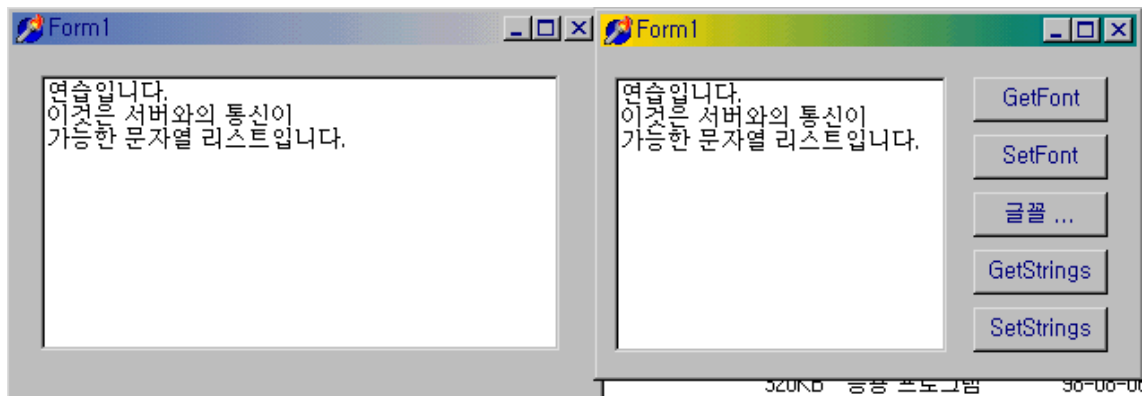
procedure TForm1.Button3Click(Sender: TObject);
begin
    FItems := StringSvr.Items;
    SetOleStrings(Memo1.Lines, FItems);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    GetOleStrings(Memo1.Lines, FItems);
    StringSvr.Items := FItems;
end;

```

당연히 이들 코드를 사용하기 위해서는 GetOleStrings 등의 함수가 선언된 AxCtrls.pas 유닛과 표준 인터페이스의 선언부가 위치한 StdVCL.pas 유닛, 그리고 IStringSvr 과 IFontSvr 인터페이스가 선언되어 있는 ExamSvr3\_TLB.pas 유닛을 uses 절에 추가해야 한다.

그러면, 클라이언트를 실행하고 폰트와 메모 컴포넌트의 내용을 바꿔 가면서 서버와의 통신을 해보기 바란다. 다음과 같이 서버와 클라이언트 간의 문자열 리스트와 폰트의 정보를 쉽게 주고 받는 과정을 눈으로 확인할 수 있을 것이다.



## 정 리 (Summary)

이번 장에서는 기본적인 자동화 서버를 작성하는 방법과 타입 라이브러리 에디터를 이용하는 방법에 대해서 알아보았다. 자동화 서버는 윈도우 95 를 지원하는 어플리케이션을 제작할 때 사용자에게 많은 편리성을 제공할 수 있는 도구가 된다.

또한, 공통적으로 사용하는 기능이 있을 경우 이를 자동화 서버로 구현해 놓으면 개발 도구를 가리지 않고, 어느 곳에서나 사용할 수 있으므로 그 활용도가 매우 높다.

다음 장에서는 액티브 X 컨트롤과 액티브 폼을 제작하고, 이를 활용하는 방법에 대해서 알아볼 것이다.

## 액티브 X 컨트롤, 액티브 폼의 제작

### (Creating ActiveX Controls, ActiveForms)

텔파이의 VCL 컴포넌트와 액티브 X 컴포넌트는 사실 의미 상으로 많은 부분이 통하지만, 실제 구현 방법은 많은 차이가 있기 때문에 VCL 컨트롤을 액티브 X 컨트롤로 전환개발하려면 사실 많은 단순 작업을 해주어야 한다. 텔파이 4 에서는 이런 작업을 단순화 시키는 레이어를 제공하는데 이것이 바로 액티브 X 컨트롤 위저드이다. 마찬가지로 컴포넌트가 추가된 폼을 하나의 액티브 X 컨트롤처럼 사용할 수 있는데, 이를 액티브 폼이라고 하며 액티브 폼 역시 텔파이 4 에서 제공되는 액티브 폼 위저드를 이용해서 쉽게 작성할 수 있다. 액티브 X 컨트롤에 대한 내용의 경우 필자가 가장 많이 참고한 자료는 97 년도에 볼랜드가 개최한 컨퍼런스에서 Conrad Herrman 이 발표한 컨퍼런스 자료이다. Conrad Herrman 은 inprise 에서 운영하는 뉴스 그룹에서도 액티브 X 분야에서 가장 활발한 활동을 하고 있는 사람으로 inprise 와 관련이 없는데에도 불구하고, 어려운 질문에도 즉각 즉각 답변을 해주는 사람이다. 아마도 이 글을 읽을 기회는 없겠지만 이 자리를 빌어 감사의 뜻을 전하고 싶다.

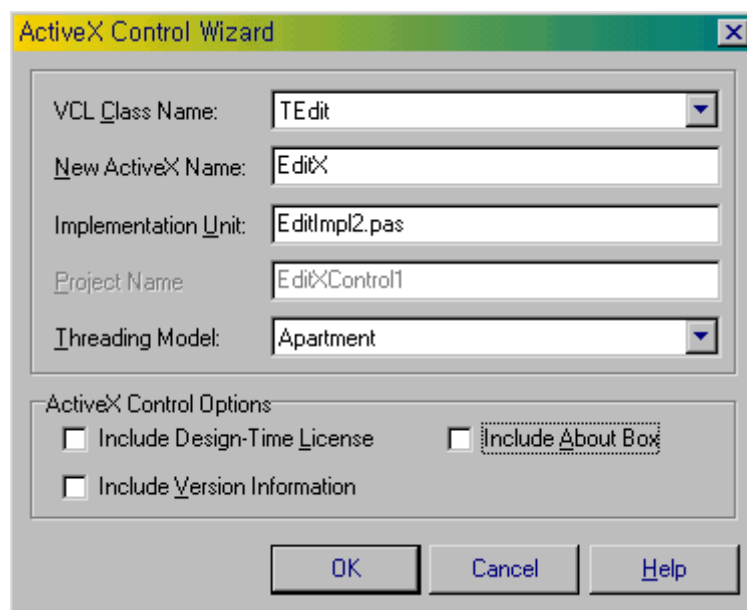
#### 액티브 X 컨트롤 위저드의 이용

사용하는 것은 간단하지만, 위저드가 내부적으로 동작하는 방법을 간단하게 설명하겠다. 먼저 자동화 인터페이스와 이벤트 인터페이스를 지정하고, 구현할 객체의 클래스 ID 를 생성한다. 그리고 나서 이 객체를 액티브 X 서버 라이브러리에서 사용할 수 있도록 포장해 준다. 이 과정에서 VCL 컴포넌트의 프로퍼티, 메소드, 이벤트를 OLE 스타일로 변환시켜주는 짧은 어댑터 루틴이 필요하다. 실제로 액티브 X 컨트롤 위저드를 사용해 보자.

1. TWinControl 에 기초한 정상적인 텔파이 컨트롤을 빌드하고, 이를 컨트롤 팔레트에 인스톨한다. 여기에서는 텔파이 TEdit 를 이용하겠다. 이 컴포넌트를 액티브 X 컨트롤로 만들면 다른 개발도구에서도 유용하게 사용할 수 있을 것이다.
2. File|New... 메뉴를 선택하고, 여기에서 ActiveX 페이지를 선택하면 액티브 X 제작과 관련한 여러가지 마법사를 볼 수 있는데, ActiveX Control 아이템을 선택하면 다음 그림과 같은 대화상자가 나타난다.
3. VCL Class Name 항목에는 액티브 X 컨트롤로 전환시킬 VCL 클래스를 선택하면 자동으로 아래의 3 항목이 채워진다. 이를 그대로 사용해도 좋고, 다른 이름으로 바꿀 수도 있다. 참고로 각각의 항목은 새로운 액티브 X 컨트롤 클래스의 이름, 컨트롤을 구현하는

Unit 파일, 액티브 X 서버 라이브러리 프로젝트 파일을 지정한다.

4. 적절한 스레딩 모델을 선택한다. 보통은 Apartment 모델을 사용한다. 앞 장에서 여기에 대해서는 자세히 설명하였으므로, 이를 참조하기 바란다.
5. 대화상자 하단의 옵션 체크 박스는 나중에 설명하기로 하고, 여기서 OK 를 누르면 위저드는 액티브 X 컨트롤을 구현하는 코드와 이를 담을 액티브 X 서버 라이브러리 프로젝트 파일을 자동으로 생성하거나 수정한다.
6. 프로젝트를 빌드하면 액티브 X 컨트롤이 만들어진다.
7. 마지막으로 만들어진 컨트롤을 시스템에 등록해야 하는데, Run|Register ActiveX Sever 메뉴를 선택하면 등록이 완료된다.



액티브 X 컨트롤 위저드 대화상자

TEdit 의 경우 액티브 X 컨트롤 위저드에 의해 EditXControl1.dpr 파일과 액티브 X 서버를 구현하는 EditXImpl1.pas 파일, 프로젝트에 import 되는 타입 라이브러리 파일(.tlb)과 타입 라이브러리의 파스칼 버전이 생성된다.

## 액티브 X 프로젝트 파일

위저드에 의해 생성되는 프로젝트 파일의 내용을 살펴보자. 소스 코드는 다음과 같다. 이해를 돕기 위해 필자가 주석을 달았다.

```
library EditXControl1;
```

```
    //EditXControl1 이라는 DLL 파일을 생성하는 프로젝트임을 나타냄
```

```
uses
```



```

ComServ,
EditXControl1_TLB in 'EditXControl1_TLB.pas',
EditImpl1 in 'EditImpl1.pas' {EditX: CoClass};

    //EditX 라는 액티브 X 클래스를 구현하는 unit 가 EditXImpl1.pas 파일에 구현되어 있음을 나
    타내는 줄

{$E ocx}          //링커에게 output 파일의 확장자가 '.ocx'임을 알려준다.

exports
    DllGetClassObject,
    DllCanUnloadNow,
    DllRegisterServer,
    DllUnregisterServer;

    //표준 액티브 X 서버 함수를 export.
    이 함수들은 ComServ unit 에 구현되어 있으므로, 따로 구현할 필요가 없다.

{$R *.TLB}        //링커에게 타입 라이브러리 파일을 DLL 의 리소스로 포함할 것을 요구한다.
{$R *.RES}        //링커에게 프로젝트의 리소스를 포함할 것을 요구한다. 여기에는 툴바 비트맵과
                  버전 정보 리소스가 포함된다.

begin
end.

```

## 액티브 X 컨트롤러 객체의 선언

실제로 액티브 X 컨트롤의 구현부분은 'EditXImpl1.pas' 파일에 담겨져 있다. 이 파일에는 액티브 X 컨트롤의 액티브 X 컨트롤러 객체가 구현되어 있으며, 액티브 X 컨트롤러 객체에 의해 자동화 인터페이스가 정의되고, OLE 자동화 스타일의 프로퍼티, 메소드, 이벤트가 구현된다. 주요 소스를 살펴 보자.

```

unit EditImpl1;

interface

uses

    Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms, StdCtrls,
    ComServ, StdVCL, AXCtrls, EditXControl1_TLB;

```

EditXControl1\_TLB.pas unit 은 파스칼 버전의 타입 라이브러리로 여기에 인터페이스가 정의되어 있다. 위저드에 의해 생성된다.

type

```
TEditX = class(TActiveXControl, IEditX)
```

```
//타입 라이브러리에 정의된 IEditX 인터페이스를 구현하는 컨트롤러 객체인
```

```
TEditX 클래스를 TActiveXControl 클래스에 기초하여 선언
```

```
private
```

```
{ Private declarations }
```

```
FDelphiControl: TEdit;
```

```
//VCL 컨트롤을 가리키는 필드. InitializeControl 메소드에 의해 초기화 된다. 아래에 열거되는 멤버들을 이용하여 프로퍼티, 메소드 등에 접근한다.
```

```
FEvents: IEditXEvents;
```

```
//컨테이너의 이벤트 싱커에 대한 포인터이다. 일종의 dispinterface 로 IDispatch 포인터를 저장한다. 이 값은 컨트롤이 컨테이너에 삽입되거나 제거될 때 EventSinkChanged 메소드가 호출되면 설정된다. 이 값이 nil 일 수도 있는데, 이는 컨테이너 애플리케이션이 이벤트에 대한 처리를 하지 않는 경우이다.
```

```
procedure ChangeEvent(Sender: TObject);
```

```
procedure ClickEvent(Sender: TObject);
```

```
procedure DbClickEvent(Sender: TObject);
```

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);
```

이벤트 핸들러 프록시에 대한 선언 부분이다.

```
protected
```

```
{ Protected declarations }
```

```
procedure InitializeControl; override;
```

```
procedure EventSinkChanged(const EventSink: IUnknown); override;
```

```
procedure DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage); override;
```

```
... (중략)
```

앞의 세가지 메소드는 TActiveXControl 에 선언된 가상 메소드를 오버라이드한 것으로, 다음에 자세히 설명한다.

```
function Get_AutoSelect: WordBool; safecall;  
function Get_AutoSize: WordBool; safecall;  
function Get_BevelInner: TxBevelCut; safecall;
```

... (중략)

//이들은 프로퍼티의 getter 메소드로, 이들은 IEditX 인터페이스에 정의되어 있다. 이들은 모두 safecall 호출규칙(calling convention)을 사용하는데, 이것은 오브젝트 파스칼에서 자동화 메소드에 호환되는 듀얼 인터페이스를 선언할 때 사용하는 것으로, 예외가 발생하면 OLE 호출규칙에 의거하여 OLE 에러 값을 반환한다.

```
procedure Set_AutoSelect(Value: WordBool); safecall;  
procedure Set_AutoSize(Value: WordBool); safecall;  
procedure Set_BevelInner(Value: TxBevelCut); safecall;
```

... (중략)

//이들은 프로퍼티의 setter 메소드이다. 이들 역시 IEditX 인터페이스에 정의되어 있다.

end;

//참고로 TEditX 컴포넌트에는 액티브 X 컨트롤로 전환될 때 사용될 수 있는 public 메소드가 없는 관계로, 메소드에 대한 선언이 빠져 있다.

implementation

uses ComObj;

{ TEditX }

procedure TEditX.InitializeControl;

begin

FDelphiControl := Control as TEdit;

//Control 은 TActiveXControl 에 선언되어 있는 프로퍼티로, TWinControl 에서 상속받은 VCL 컨트롤을 지정한다. FDelphiControl 필드에 TEdit VCL 객체의 포인터가 저장된다.

```

FDelphiControl.OnChange := ChangeEvent;
FDelphiControl.OnClick := ClickEvent;
... (중략)
//컨트롤의 VCL 이벤트를 컨트롤러 객체의 이벤트 핸들러 프록시 메소드에 매핑하는 코드이다. 이
    렇게 함으로써 VCL 컨트롤이 이벤트를 발생시키면, 컨트롤러 객체가 이벤트를 받게 된다.
end;

```

InitializeControl 메소드는 컨트롤이 생성되고, 컨테이너에 삽입되기 전에 호출되는 메소드로, COM 컨트롤러 객체와 VCL 객체 간의 커넥션을 만든다. 보다 자세하게 설명하면, 컨트롤러 객체가 VCL 객체의 포인터를 얻은 다음, 이벤트 프록시를 VCL 객체에 연결해 준다.

```

procedure TEditX.EventSinkChanged(const EventSink: IUnknown);
begin
    FEvents := EventSink as IEditXEvents;
end;

```

컨테이너가 제공하는 이벤트 싱크를 저장한다. FEvents 필드는 컨테이너 객체에 이벤트를 발생시킬 때 사용된다. IEditXEvents 는 컨트롤의 이벤트 dispinterface 로 타입 라이브러리에 디폴트 소스 인터페이스로 선언되어 있다.

```

procedure TEditX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin

end;

```

이 메소드는 개발자가 나중에 프로퍼티 페이지를 추가할 때 사용한다. 프로퍼티 페이지를 추가하는 방법에 대해서는 나중에 설명한다.

## 프로퍼티의 Get, Set 메소드의 구현 방법

VCL 컨트롤의 프로퍼티는 위저드에 의해 생성되는 컨트롤러 객체의 Get, Set 메소드에 의해 접근할 수 있다. 이들이 실제로 어떻게 구현되어 있는지 살펴 보자.  
가장 전형적인 Get, Set 메소드는 다음과 같다.

```

function TEditX.Get_AutoSelect: WordBool;
begin

```

```

    Result := FDelphiControl.AutoSelect;
end;

```

```

procedure TEditX.Set_AutoSelect(Value: WordBool);
begin
    FDelphiControl.AutoSelect := Value;
end;

```

대부분의 경우는 이와 같이 간단하게 구현이 되어 있지만, 프로퍼티의 데이터 형이 호환되지 않을 때에는 약간의 조작이 필요하다. 대부분의 경우 위저드가 자동으로 구현해 주지만, 지원하지 않는 데이터 형을 사용하는 public 프로퍼티는 사용할 수 없게 될 수도 있으며, 이를 구현하기 위해서 약간의 코딩이 필요할 수도 있다.

가장 복잡한 경우가 폰트, 그림, string list 등의 경우이다. 예를 들어 폰트의 경우 독립적인 dispatch 인터페이스를 가지는 독립적인 객체이기 때문에, 적절한 접근 방법이 있어야 한다. 이를 해결하려면 Get\_Font 메소드는 폰트의 프로퍼티를 OLE 프로퍼티로 노출(expose)시킬 수 있는 OLE 객체를 생성하고, 이를 반환해야 하며, 반대로 Set\_Font 메소드는 OLE 폰트 객체의 값을 VCL 프로퍼티에 대입할 수 있어야 한다.

다행히 이를 위해, DAX 라이브러리에서는 TFont, TPicture 클래스를 위한 함수를 제공한다. 위저드에서 자동으로 생성한 Get\_Font, Set\_Font 메소드는 다음과 같다.

```

function TEditX.Get_Font: IFontDisp;
begin
    GetOleFont(FDelphiControl.Font, Result);
end;

```

```

procedure TEditX._Set_Font(const Value: IFontDisp);
begin
    SetOleFont(FDelphiControl.Font, Value);
end;

```

위저드에서 일부의 프로퍼티는 생성하지 않는데, 이들의 예를 들면 Height, Left 등과 같이 Extended 프로퍼티를 사용해야 하는 경우나, ParentFont, Hints 등과 같이 액티브 X 컨테이너가 지원하지 않는 경우, PopUpMenu 와 같이 OLE 프로퍼티 데이터 형이 도저히 지원하지 않는 경우 등이 있다.

## 이벤트의 처리

기본적으로 InitializeObject 메소드에 의해 VCL 컨트롤과 이벤트 핸들러가 연결된다는 것은 위에서 설명했다. 그럼 실제 이벤트를 처리하는 부분을 살펴 보자.

```
procedure TEditX.ChangeEvent(Sender: TObject);
begin
    if FEvents <> nil then FEvents.OnChange;
end;
```

이를 분석하면, FEvents dispatch 인터페이스가 설치되어 있으면(즉, IDispatch 포인터가 할당되어 있으면) 단순히 이벤트를 컨테이너의 이벤트 싱크에 넘겨주는 것으로 되어 있다. FEvents 는 위에서 설명한 EventSinkChanged 메소드에 의해 설정된다.

KeyPressEvent 는 OnClick, OnChange 이벤트에 비해 다소 복잡하게 구현되어 있는데, 이는 OLE 이벤트에서 요구하는 파라미터가 델파이 이벤트의 파라미터와 다르기 때문에 이를 변환하는 코드가 생성되기 때문이다. 실제 코드를 살펴보자.

```
procedure TEditX.KeyPressEvent(Sender: TObject; var Key: Char);
var
    TempKey: Smallint;
begin
    TempKey := Smallint(Key);
    if FEvents <> nil then FEvents.OnKeyPress(TempKey);
    Key := Char(TempKey);
end;
```

이 경우 이벤트 핸들러 프록시는 이벤트를 컨테이너에 발생시키기 전에 파라미터를 조작하게 된다. OnKeyPress 이벤트는 SmallInt 에 대한 포인터를 컨테이너에 넘기지만, VCL 컨트롤은 Char 에 대한 포인터를 이벤트 핸들러에게 넘기므로 이를 SmallInt 와 Char 로 타입 캐스팅하는 부분이 필요한 것이다.

참고로, TDateTimePicker 와 같이 다소 복잡한 컨트롤의 경우에는 더 복잡한 처리방법을 가지는 경우도 있다. 이때에는 VCL 컨트롤에서 사용하는 TDateTime, String, Boolean 데이터 type 을 OLE 이벤트의 WideString, WordBool 등으로 타입 캐스팅할 필요가 있다. 이런 부분을 모두 위저드가 자동으로 코딩해준다.

## 클래스 팩토리를 통한 인스턴스의 생성

마지막으로 라이브러리가 메모리 상에 로드되었을 때, 클래스 팩토리를 생성하는 부분을 살펴보자.

initialization

```
TActiveXControlFactory.Create(ComServer, TEditX, TEdit, Class_EditX, 1,
    "", 0, tmApartment);
```

ComServer 는 라이브러리를 나타내는 전역변수로 라이브러리에 의해 생성된 클래스 팩토리의 리스트를 포함하고 있다.

2, 3 번째 파라미터는 각각 액티브 X 를 구현하는 클래스, VCL 컨트롤 클래스이며, Class\_EditX 는 EditXControl1\_TLB.pas unit 에 선언되어 있는 객체의 ClassID 이다. 5, 6, 7 번째 파라미터는 각각 ToolBarBitmapID, LicenseString, MiscControl flag 등을 나타내며, 마지막 파라미터에는 쓰레딩 모델을 지정한다.

## 타입 라이브러리

일단 액티브 X 라이브러리를 컴파일하면 타입 라이브러리는 DLL 에 리소스로 복사된다.

액티브 X 컨트롤 위저드는 처음 액티브 X 컨트롤을 델파이 VCL 에서 생성할 때 타입 라이브러리를 만들며 이를 .TLB 파일로 저장한다. 이때 위저드는 프로퍼티와 파라미터를 OLE 에 호환되는 타입으로 전환한다. 컨트롤에 나열형 프로퍼티나 파라미터가 포함되어 있으면 이에 대한 타입 선언부를 만들어 주며, 데이터 타입이 TFont, TPicture, TStrings 일 경우에는 프로퍼티, 파라미터를 IFont, IPicture, IStrings 로 대입하고, 이들에 대한 어댑터(adapter) 코드를 생성해준다. 만약 VCL 컨트롤에 COM 객체 타입이 될 수 없는 프로퍼티나 파라미터가 있을 경우에는 위저드는 이들을 생략해 버린다.

## 사용자 정의 객체 스트리밍

DAX 객체에 대한 기본적인 스트리밍(streaming)은 VCL 의 포맷을 따르게 된다. 그러나, 개발자가 기본적인 정보 외에 추가적인 정보를 지속적 스트림(persistence stream)에 저장하려 한다면 LoadFromStream, SaveToStream 메소드를 오버라이딩하면 된다. 이때 컨트롤의 프로퍼티를 제대로 읽고, 저장하려면 inherited 메소드를 호출해야 한다. 이들 메소드는 다음과 같이 정의되어 있다.

```
procedure LoadFromStream(const Stream: IStream);
```

```
procedure SaveToStream(const Stream: IStream)
```

텔파이의 표준 스트리밍 클래스는 TStream 이다. 이 클래스에는 Read, Write, Seek 메소드가 있으며, TPersistent 객체에서 TStream 을 이용해 스트리밍을 하게 된다. 대부분의 텔파이 객체가 TPersistent 객체에서 상속받으므로, 객체의 내용을 저장할 때 TStream 을 사용한다. OLE 에서는 스트리밍 객체는 IStream 이라는 인터페이스를 제공하며, 여기에 역시 Read, Write, Seek 메소드가 정의되어 있다. 텔파이 4 에서는 TOleStream 이라는 클래스를 제공하는데 이를 통해 TStream 을 IStream 으로 노출(expose)시킬 수 있다.

TActiveXControl 이 SaveToStream 메소드를 이용해서 객체의 상태를 스트림에 저장할 때 파라미터로 IStream 을 가지는 것이다. 만약 개발자가 추가 데이터를 스트림에 저장하고자 한다면 스트림 어댑터를 이용해 TPersistent 객체가 그 내용을 IStream 에 저장하게 된다. 다음 코드는 TOleStream 을 이용해 추가적인 정보를 문자열 리스트 (string list)에 담아서 저장하는 예이다. 또한, InitializeControl 메소드에서 ExtraInfo 문자열 리스트 객체를 생성하고, destructor 인 Destroy 메소드를 오버라이드하여 사용한 ExtraInfo 문자열 객체를 해제해 주어야 한다.

... (전략)

protected

```
    procedure InitializeControl; override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure LoadFromStream(const Stream: IStream); override; //추가
    procedure SaveToStream(const Stream: IStream); override;    //추가
```

... (중략)

var

```
    ExtraInfo: TStringList;
```

```
procedure TEditX.InitializeControl;
```

```
begin
```

```
    ...
```

```
    ExtraInfo := TStringList.Create;
```

```
end;
```

```
destructor TEditX.Destroy;
```



```

begin
    ExtralInfo.Free;
end;

procedure TEditX.SaveToStream(const Stream: IStream);
var
    TempStream: TStream;
begin
    inherited;
    TempStream := TOleStream.Create(Stream);
    try
        ExtralInfo.SaveToStream(TempStream);
    finally
        TempStream.Free;
    end;
end;

procedure TEditX.LoadFromStream(const Stream: IStream);
var
    TempStream: TStream;
begin
    inherited;
    TempStream := TOleStream.Create(Stream);
    try
        ExtralInfo.LoadFromStream(TempStream);
    finally
        TempStream.Free;
    end;
end;

```

## 컨트롤에 verb 추가하기

Verb 는 사용자가 발생시키는 액션을 말한다. 예를 들어, 메뉴 아이템에서 copy, paste, run 등을 선택하면 객체가 특정한 동작을 한다면 이들이 좋은 verb 의 예가 된다. 그럼 이러한 verb 를 우리가 제작하는 컨트롤에 추가시키는 방법을 배워보자.

Verb 를 컨트롤에 추가할 때 필요한 것은 verb 를 등록하는 코드와 verb 를 실행하는 코드

이다.

Verb 를 등록하는 것은 객체의 클래스 팩토리를 이용해서 라이브러리가 인스톨될 때 verb information 을 시스템 레지스트리에 복사하는 것을 말한다. 이런 작업은 팩토리 객체의 AddVerb 메소드를 이용하면 된다. 다음의 코드를 살펴보자.

```
const
    VERB_TEXT = 100;

initialization
    with TActiveXControlFactory.Create(ComServer, TEditX, TEdit, Class_EditX,
        1, '', 0, tmApartment) do
    begin
        AddVerb(VERB_CLICK, '&Text');
    end;
end.
```

이 코드에 의해서 'Text'라는 verb 가 컨트롤에 추가되며 사용자가 컨테이너의 메뉴 아이템에서 'Text'를 선택하면 에디트 컨트롤의 Text 프로퍼티를 변경하도록 설정하도록 하자. 컨테이너는 객체에 verb 가 있을 때 이를 나타낼 책임을 가지고 있으며, 사용자가 이를 선택했을 때 액티브 X 컨트롤을 호출하여야 한다. DAX 에서는 실제로 verb 를 실행할 때 PerformVerb 메소드를 호출한다.

앞의 Text verb 에 대응하기 위한 PerformVerb 메소드는 다음과 같은 식으로 작성하면 된다. 먼저 interface 의 protected 섹션에 메소드를 선언하고, 이를 구현한다. InputBox 함수는 Dialogs.pas 유닛에 선언되어 있으므로 uses 절에 Dialogs.pas 유닛을 추가할 필요가 있다.

```
procedure PerformVerb(Verb: Integer); override;

procedure TEditX.PerformVerb(Verb: Integer);
var
    InputText: string;
begin
    case Verb of
        VERB_TEXT:
            begin
                InputText := InputBox('텍스트', '문자열을 입력하세요 !', '');
            end;
    end;
end;
```

```

    FDelphiControl.Text := InputText;
end;
else
    Inherited PerformVerb(Verb);
end;
end;

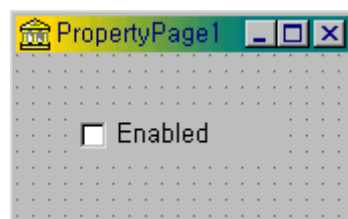
```

## 프로퍼티 페이지의 제작

프로퍼티 페이지는 사용자에게 컨트롤의 프로퍼티를 편집할 수 있게 하기 위해 제공되는 폼이다. 프로퍼티 페이지를 반드시 제공할 필요는 없는데, 이때에는 각 개발 툴의 프로퍼티 인스펙터에서 프로퍼티를 변경하게 된다. 그러나, 프로퍼티를 제공하면 사용자가 보다 쉽고 직관적인 인터페이스를 가지고 프로퍼티를 편집할 수 있게 된다.

프로퍼티 페이지 역시 OLE 객체로 액티브 X 라이브러리에 포장되고 시스템 레지스트리에 등록되어야 한다. 그러나, 이를 사용하는 컨트롤과 같은 라이브러리에 포함되어야 하는 것은 아니기 때문에 다른 컨트롤에서 등록된 같은 프로퍼티 페이지를 사용하는 것도 가능하다. 델파이에서는 폰트와 컬러, 그림과 문자열에 대한 4 가지의 기본적인 프로퍼티 페이지를 제공한다. 이들의 클래스 ID 는 각각 Class\_DColorPropPage, Class\_DFontPropPage, Class\_DPicturePropPage, Class\_DStringPropPage 이다.

프로퍼티 페이지를 추가하려면 객체 저장소나 File|New.. 메뉴의 액티브 X 페이지에서 액티브 X 프로퍼티 페이지 위저드를 실행하면 된다. 위저드를 실행하면 폼하나와 unit 파일 하나가 자동으로 생성되는데, 다음 그림은 TEdit 의 Enabled 프로퍼티를 체크하는 체크 박스를 프로퍼티 페이지에 올려 놓은 그림이며, 생성되는 코드는 다음과 같다.



```

unit EditXProp;

```

```

... (중략)

```

```

type

```

```

    TPropertyPage1 = class(TPropertyPage)

```

```

        CheckBox1: TCheckBox;

```

```

private
protected
public
    procedure UpdatePropertyPage; override;
    procedure UpdateObject; override;
end;

```

여기에서 프로퍼티 페이지 클래스는 TPropertyPage 클래스에서 상속받음을 알 수 있으며, 폼에 삽입한 CheckBox 컨트롤이 있다. TPropertyPage 클래스는 TCustomForm 클래스에서 상속받은 것으로 OleObject 프로퍼티를 이용해 프로퍼티 페이지와 객체의 편집을 연결하며, UpdatePropertyPage, UpdateObject 메소드를 제공하는데, 이를 오버라이드하여 실제로 OLE 객체를 편집하게 된다. 이들 메소드를 다음과 같이 구현하였다.

```

procedure TPropertyPage1.UpdatePropertyPage;
begin
    CheckBox1.Checked := OleObject.Enabled; //이 부분을 실제로 추가해 주어야 함
end;

procedure TPropertyPage1.UpdateObject;
begin
    OleObject.Enabled := CheckBox1.Checked; //이 부분을 실제로 추가해 주어야 함
end;

```

UpdatePropertyPage 메소드는 OLE 객체의 프로퍼티를 프로퍼티 페이지에 반영하는 역할을 하며, UpdateObject 메소드는 프로퍼티 페이지의 내용을 OLE 객체에 반영한다.

```

initialization
    TActiveXPropertyPageFactory.Create(ComServer, TPropertyPage1, Class_PropertyPage1);
end.

```

이 코드는 프로퍼티를 COM 객체로 등록하는 것으로, TActiveXPropertyPageFactory 의 메소드를 사용하게 된다. ComServer 는 액티브 X 라이브러리를 대표하는 전역 변수이며, TpropertyPage1 은 폼의 클래스, Class\_PropertyPage1 은 프로퍼티 페이지 객체의 클래스 ID 이다. 이렇게 제작한 프로퍼티를 실제로 액티브 X 컨트롤에 적용하려면 DefinePropertyPages 메소드를 변경하면 되는데, 처음 만들어진 코드에는 이 부분이 주석으로 채워져 있다. 이를 다음과 같은 코드로 바꾸어 주면 된다.

```

procedure TdateTimePickerX.DefinePropertyPages(DefinePropertyPage: TdefinePropertyPage);
begin
    DefinePropertyPage(Class_PropertyPage1);    //추가된 부분
end;

```

여기에 여러 개의 프로퍼티 페이지를 DefinePropertyPage 메소드를 이용해서 컨트롤이 여러 개의 등록된 프로퍼티 페이지를 이용하도록 할 수 있다.

## Ambient 프로퍼티의 이용

Ambient 프로퍼티는 컨트롤의 컨테이너에 의해 제공되는 프로퍼티를 말한다. 일단 컨트롤이 컨테이너에 삽입되면, 컨테이너의 ambient 프로퍼티의 정보를 물어보게 된다. 그러므로, 컨테이너는 노출시키려는 ambient 프로퍼티를 정의하고 있어야 한다. 액티브 X 는 BackColor, DisplayName 등의 표준 ambient 프로퍼티를 정의하고 있다. 물론 이들을 모두 컨테이너가 노출할 필요는 없지만, 마이크로소프트는 이들을 사용하기 위해 각각에 대한 dispID 를 정의하고 있다. 그러므로, 액티브 X 에서 ambient 프로퍼티에 접근할 때에는 사이트의 IDispatch 인터페이스를 사용한다. 텔파이에서는 IAmbientDispatch 인터페이스를 제공하는데, 이 인터페이스를 이용하여 표준 인터페이스에 접근이 가능하다.

IAmbientDispatch 인터페이스는 어디까지나 dispinterface 이므로 단지 IDispatch 의 포인터일 뿐이며, 다른 dispinterface 로 형변환이 가능하다. IAmbientDispatch 인터페이스의 선언부는 다음과 같다.

```

IAmbientDispatch = dispinterface
    ['{00020400-0000-0000-C000-000000000046}']
    property BackColor: Integer dispid DISPID_AMBIENT_BACKCOLOR;
    property DisplayName: WideString dispid DISPID_AMBIENT_DISPLAYNAME;
    property Font: IFontDisp dispid DISPID_AMBIENT_FONT;
    property ForeColor: Integer dispid DISPID_AMBIENT_FORECOLOR;
    property LocaleID: Integer dispid DISPID_AMBIENT_LOCALEID;
    property MessageReflect: WordBool dispid DISPID_AMBIENT_MESSAGEREFLECT;
    property ScaleUnits: WideString dispid DISPID_AMBIENT_SCALEUNITS;
    property TextAlign: Smallint dispid DISPID_AMBIENT_TEXTALIGN;
    property UserMode: WordBool dispid DISPID_AMBIENT_USERMODE;
    property UIDead: WordBool dispid DISPID_AMBIENT_UIDEAD;
    property ShowGrabHandles: WordBool dispid DISPID_AMBIENT_SHOWGRABHANDLES;

```

```

property ShowHatching: WordBool dispid DISPID_AMBIENT_SHOWHATCHING;
property DisplayAsDefault: WordBool dispid DISPID_AMBIENT_DISPLAYASDEFAULT;
property SupportsMnemonics: WordBool dispid DISPID_AMBIENT_SUPPORTSMNEMONICS;
property AutoClip: WordBool dispid DISPID_AMBIENT_AUTOCLIP;
end;

```

다음의 코드는 TEditX 컨트롤을 클릭할 때 컨테이너에서의 컨트롤의 이름을 메시지 박스로 나타내도록 한 것이다. 이를 위해 DisplayName ambient 프로퍼티를 이용하였다.

```

procedure TEditX.ClickEvent(Sender: TObject);
var
    Site: IOleClientSite;
    Ambients: IDispatch;
begin
    GetClientSite(Site);
    if Site <> nil then
        Site.QueryInterface(IDispatch, Ambients);
        if Ambients <> nil then
            begin
                ShowMessage(IAmbientDispatch(Ambients).DisplayName);
            end;
            if FEvents <> nil then FEvents.OnClick;
        end;
    end;
end;

```

이렇게 하면, 앞으로는 에디트 박스를 클릭할 때마다 폼에서의 에디트 박스의 Name 프로퍼티의 내용이 메시지 박스로 뜰 것이다.

Ambient 프로퍼티를 가장 유용하게 사용되는 경우는 컨테이너의 색상 변화에 따라 컨트롤의 색상 변화를 유도하고 싶거나, 글꼴 변화에 따른 컨트롤의 변화와 같이 사용자 인터페이스 측면에서 반드시 사용해야할 경우 들이 있다. 이럴 때에는 ambient 프로퍼티의 이용이 유일한 해결책이 될 수 있다.

이를 위해서 컨테이너가 ambient 프로퍼티의 값을 변경했을 때, 이를 알아챌 수 있어야 하는데 이때에는 컨테이너 객체의 OnAmbientPropertyChange 메소드를 호출하게 된다. 그러므로 델파이에서 이를 이용하려면 OnAmbientPropertyChange 메소드를 오버라이드하고, IOleControl 인터페이스를 다시 구현하면 된다.

## 사용자 정의 레지스트리 엔트리의 추가

액티브 X 컨트롤을 작성한 뒤에는 레지스트리에 컨트롤에 대한 정보를 추가하고 싶을 경우가 있다. 보통 액티브 X 컨트롤이 등록될 때 이런 작업이 병행된다면 좋을 텐데, 이를 해결하는 방법이 없는지 궁금하지 않은가 ?

이를 위해서는 델파이 COM 팩토리의 UpdateRegistry 메소드를 이용하면 된다. 이 메소드는 라이브러리가 등록되거나 해제될 때 호출되기 때문에 여기에다가 레지스트리에 각종 정보를 추가하는 코드를 삽입하면 된다. 그러므로, UpdateRegistry 메소드를 오버라이드해야 하므로 TActiveXControlFactory 클래스를 상속받은 새로운 클래스 팩토리 클래스를 정의하고, 이 클래스 팩토리를 이용하여 액티브 X 라이브러리를 등록하도록 수정해야 한다.

다음의 클래스는 클래스 팩토리의 constructor 에 레지스트리에 등록할 문자열을 미리 파라미터로 받아서 이를 레지스트리에 등록하도록 수정한 클래스 팩토리 클래스를 구현한 것이다.

type

```
TRegistryFactory = class(TActiveXControlFactory)
public
    constructor Create(ComServer: TComServerObject; ActiveXControlClass: TActiveXControlClass;
        WinControlClass: TWinControlClass; const ClassID: TGUID; ToolboxBitmapID: Integer;
        const LicStr: string; MiscStatus: Integer; ThreadingModel: TThreadingModel = tmSingle;
        SpecialKeyValue: string); override;
    procedure UpdateRegistry(Register: Boolean); override;
protected
    FSpecialKeyValue: string;
end;
```

constructor TRegistryFactory.Create(ComServer: TComServerObject;

```
    ActiveXControlClass: TActiveXControlClass; WinControlClass: TWinControlClass;
    const ClassID: TGUID; ToolboxBitmapID: Integer; const LicStr: string; MiscStatus: Integer;
    ThreadingModel: TThreadingModel = tmSingle; SpecialKeyValue: string);
```

var

```
    TypeAttr: PTypeAttr;
```

begin

```
    FSpecialKeyValue := SpecialKeyValue;
```

```
    inherited Create(ComServer, ActiveXControlClass, WinControlClass, ClassID,
        ToolboxBitmapID, LicStr, MiscStatus, ThreadingModel);
```

```
end;
```

```
procedure TRegistryFactory.UpdateRegistry(Register: Boolean);
```

```
var
```

```
    ClassKey: string;
```

```
begin
```

```
    ClassKey := 'CLSIDW' + GUIDToString(ClassID);
```

```
    if Register then
```

```
    begin
```

```
        inherited UpdateRegistry(Register);
```

```
        CreateRegKey(ClassKey + 'WSpecialKey', '', FSpecialKeyValue);
```

```
    end
```

```
    else
```

```
    begin
```

```
        DeleteRegKey(ClassKey + 'WSpecialKey');
```

```
        inherited UpdateRegistry(Register);
```

```
    end;
```

```
end;
```

이 클래스 팩토리를 이용하기 위해서는 액티브 X 라이브러리의 initialization 섹션을 다음과 같이 수정하면 된다. 다음의 경우 추가적인 레지스트리 키 값을 'Sample'로 설정하는 경우이다.

```
initialization
```

```
    TRegistryFactory.Create(ComServer, TEditX, TEdit, Class_EditX, 1, '', 0, tmApartment, 'Sample');
```

```
end.
```

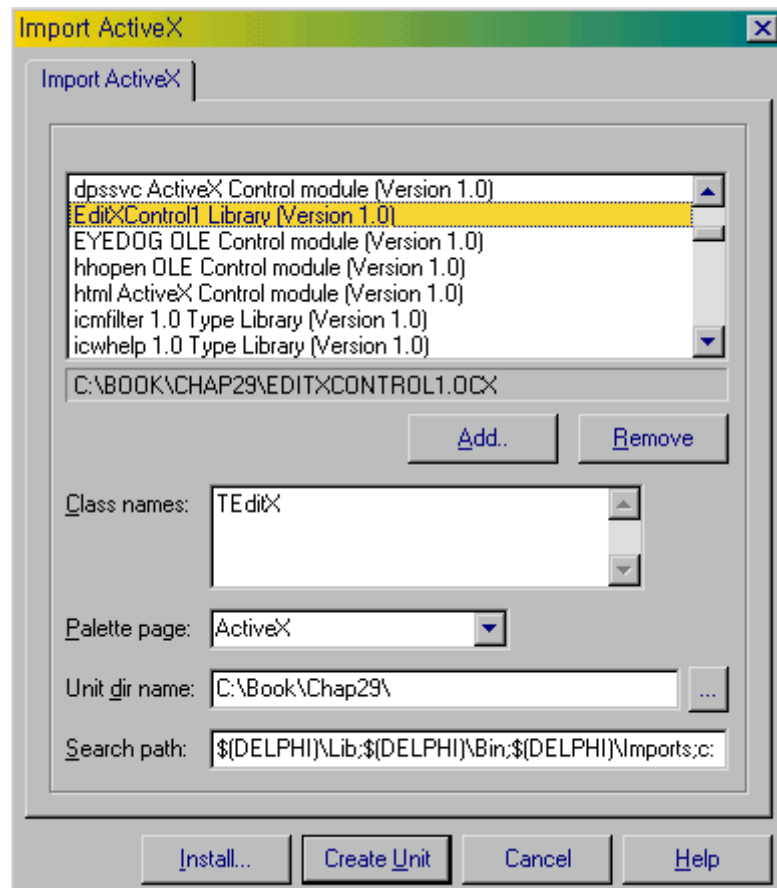
## 액티브 X 컨트롤의 등록과 이용

그러면, 이렇게 작성한 액티브 X 컨트롤을 등록하고 이용하는 방법에 대해서 알아보자. 여기에서 설명하는 내용은 인터넷에서 쉽게 구할 수 있는 여러가지 다른 액티브 X 컨트롤에 대해서도 공통적으로 적용된다고 할 수 있다.

델파이에서 액티브 X 컨트롤을 이용하려면 먼저 .ocx 파일을 레지스트리에 등록시켜야 한다. 델파이를 이용해서 작성한 경우에는 컴파일하고, 간단히 Run|Register ActiveX Server 메뉴를 이용하여 등록이 가능하지만, 인터넷에서 구한 경우에는 RegSvr.EXE 와 같은 등록 프로그램을 이용하거나 .REG 파일을 작성하여 등록해야 한다.



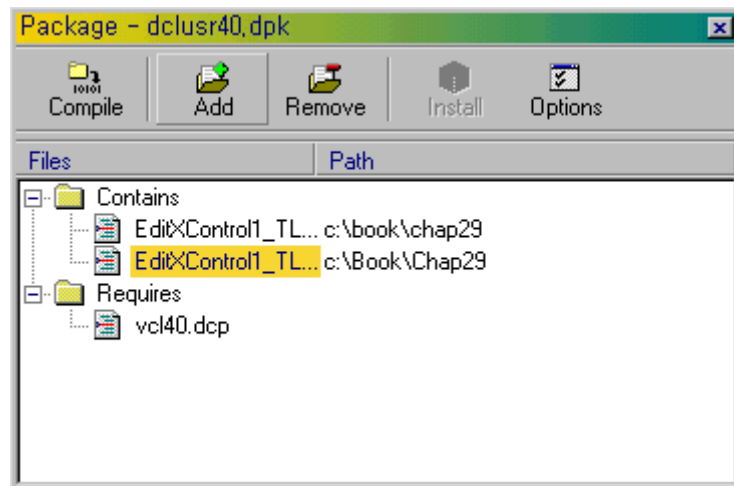
EditX 컨트롤은 간단히 컴파일하고 Run|Register ActiveX Server 메뉴를 선택하여 레지스트리에 등록할 수 있다. 이렇게 등록한 액티브 X 컨트롤을 이용하려면 델파이의 컴포넌트 팔레트에 추가해야 한다. 이를 위해서는 Component|Import ActiveX Control 메뉴를 선택하거나, Component|Install Packages 메뉴를 선택하여 패키지를 먼저 선택한 뒤에 Edit 버튼을 클릭하고 액티브 X 컨트롤을 추가할 수 있다.



이 대화 상자에서 팔레트 페이지의 이름과 생성될 \_TLB.pas 파일의 위치를 지정할 수 있다. Search Path 에디트 박스에는 액티브 X 서버 파일의 위치를 검색할 디폴트 패스를 나열하게 된다. 추가가 가능하며, 여기서 지정한 패스가 아닌 곳에 위치한 액티브 X 서버는 이 대화 상자에 나타나지 않는다. 이런 경우에는 Add 버튼을 클릭하여 액티브 X 서버 파일의 위치를 지정할 수 있다.

Install.. 버튼을 클릭하면 설치할 패키지를 지정하라는 대화 상자가 나타날 것이다. 여기에서 적절한 패키지 파일을 선택하거나, 새로운 패키지를 지정하여 설치할 수 있다.

이렇게 하면, 다음과 같은 패키지 관리자가 나타나면서 설치가 될 것이다.

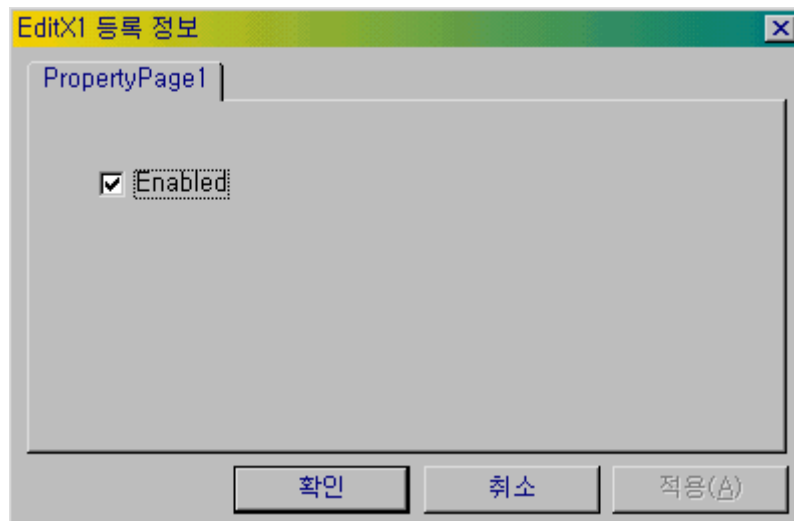


변경 사항이 있는 경우에는 Compile 버튼을 누르면 컴포넌트 팔레트에 변경된 내용이 즉시 반영된다. 여기에서 더 추가하거나 제거하고 싶은 컴포넌트나 액티브 X 컨트롤이 있으면 Add 버튼을 클릭하여 추가나 제거가 가능하다.

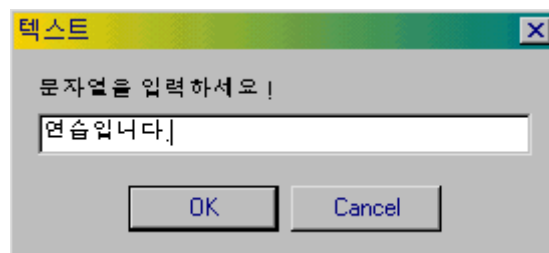
이제 새로운 어플리케이션을 시작하고, EditX 컨트롤을 사용해보자. 컴포넌트 팔레트의 ActiveX 탭에서 EditX 컨트롤을 폼에 올려 놓도록 하자. 아마도 TEdit 컴포넌트를 올려 놓는 것과 거의 같다는 것을 느낄 수 있다. 사용법 역시 동일하다. TEditX 컨트롤 위에서 오른쪽 버튼을 클릭하여 팝업 메뉴를 띄워 보자. 아마도 기본적인 Properties 메뉴와 추가한 Verb 메뉴인 Text 메뉴가 Verb 메뉴로 다음과 같이 나타날 것이다.



여기서 Properties 메뉴를 선택하여 우리가 추가한 프로퍼티 페이지가 다음과 같이 나타날 것이다.



여기서 체크 박스의 내용을 이용하여 Enabled 프로퍼티를 조절할 수 있다. 그러면, 이번에는 Text verb 를 실행해 보도록 하자. 그러면, 우리가 코딩한 대로 다음과 같은 입력 박스가 나타날 것이다. 여기에서 아무 값이나 입력하면 에디트 컨트롤의 내용인 입력한 값으로 변경될 것이다.



나머지 사용 방법은 TEdit 컴포넌트와 거의 유사하게 사용할 수 있다. 다른 액티브 X 컨트롤도 이와 비슷한 방법으로 쉽게 사용할 수 있다.

## 액티브 폼(ActiveForm)의 제작

액티브 폼은 VCL 폼에 기반을 둔 액티브 X 컨트롤이라고 생각하면 된다. 그러니까, 일종의 액티브 X 컨트롤인 셈이다. 그렇지만, 델파이 3 에서 발표된 액티브 폼은 인트라넷 개발 환경을 지원하기 위해서 여러모로 활용되면서 나름대로 독자적인 입지를 굳힌 듯하다. 비록 인터넷에서 범용 액티브 X 컨트롤로 사용하기에는 다소 무리가 따르지만 일정 정도의 네트워크 속도가 보장되는 인트라넷 환경에서는 델파이에서 일반적으로 개발하는 폼에 기반한 개발방법을 그대로 적용해서 하나의 컨트롤로 만들어낼 수 있다는 것은 나름대로의 매력이 있다고 생각할 수 있다.

액티브 폼은 델파이에서 제공하는 액티브 폼 위저드를 이용하여 작성한다. 먼저 File|New

메뉴의 ActiveX 탭에서 ActiveX Library 아이콘을 더블 클릭한다. 액티브 폼 역시 액티브 X 컨트롤이기 때문에 .ocx 파일 형태의 in-process 액티브 X 라이브러리로 등록되어 사용된다.

이렇게 하면, 텔파이에서 비어있는 프로젝트가 하나 생성될 것이다. 그 다음에는 폼을 생성할 차례이다. 마찬가지로 File|New 메뉴의 ActiveX 탭에서 이번에는 ActiveForm 아이콘을 더블 클릭하면 액티브 X 컨트롤 위저드를 생성할 때와 거의 동일한 대화상자가 나타날 것이다. 여기에서 적당한 이름과 모델 들을 선택하고 OK 버튼을 클릭하면, 흔히 보는 비어있는 폼이 하나 만들어질 것이다.

여기까지 별로 한 것이 없어 보이지만, 텔파이가 내부적으로 많은 코딩을 해 놓은 상태이다. 개발자가 할 일은 이 폼을 이용해서 마음대로 사용자 인터페이스를 디자인하고 필요한 코딩을 해주면 된다.

개발하는 방법은 일반적인 폼을 이용해서 개발하는 것과 완전히 동일하다. 컴포넌트를 올려 놓고, 거기에 대한 이벤트 핸들러를 작성한다.

단지 다른 점이 있다면, 컴파일을 하고 나서 이 폼을 실제로 실행할 수 있는 것은 HTML 페이지 위에서만 할 수 있다는 것이다. 다음에 설명하는 웹 배포 방법을 이용해서 액티브 폼을 HTML 파일로 변경하고, 이 파일을 열어 보면 액티브 폼이 제대로 실행되는지 알아볼 수 있다.

그러면 만들어진 빈 폼에 다음과 같이 버튼을 하나 올려 놓고 Caption 프로퍼티를 'OK!'로 설정하고, OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TActiveFormX.Button1Click(Sender: TObject);
begin
    ShowMessage('연습입니다 !');
end;
```

그리고, 컴파일을 하면 프로젝트 파일에 대한 .ocx 파일이 생성될 것이다. 앞으로 할 일은 웹을 이용해 배포할 수 있도록 옵션을 지정하고, 실제로 배포를 하면 완료된다.

## 웹 배포 (Web Deployment)

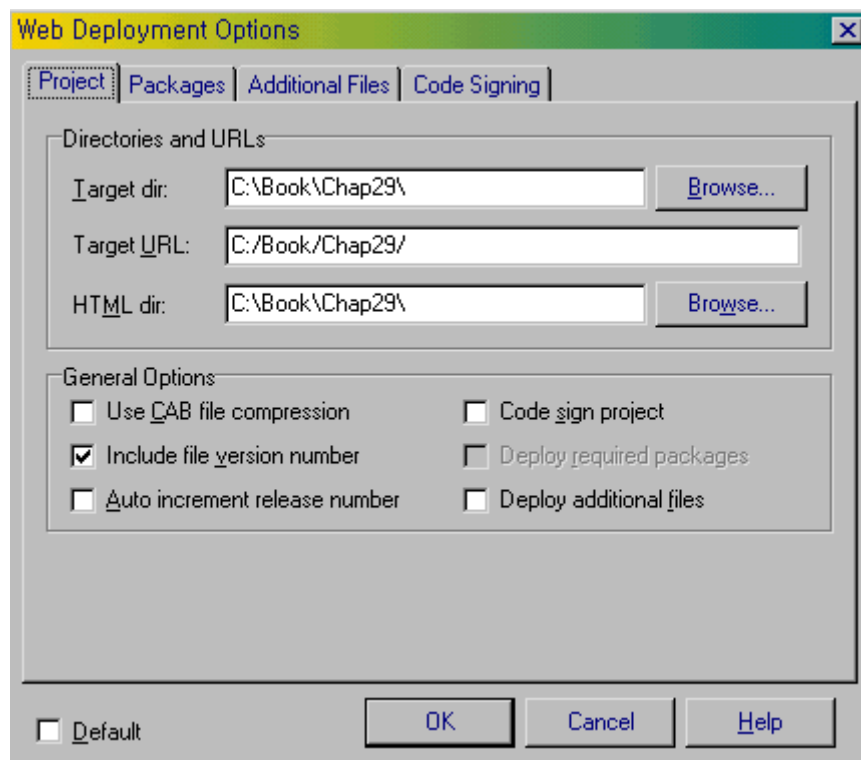
텔파이로 제작한 액티브 X 컨트롤을 웹 사이트에 배포할 때에는 다음과 같은 몇 가지를 지정해 주어야 한다. 웹 배포의 옵션은 Project|Web Deployment Options 메뉴를 선택하면 설정할 수 있다. 이 메뉴를 선택하면 웹 배포 옵션을 위한 대화 상자가 나타나는데, 여기에서 가장 중요한 부분은 다음에 설명하는 Target dir, Target URL, HTML dir 에 대한 내용이다.

1. Target dir: 델파이가 복사할 코드베이스 파일이 위치할 디렉토리를 지정한다.
2. Target URL: 바이너리 코드베이스가 존재하는 서버를 지정한다. 이는 URL 의 형태인 데 예를 들어 'http://www.sample.com/code/Sample.ocx' 등과 같은 형태를 가진다.
3. HTML dir: 델파이가 복사할 HTML 파일이 위치할 디렉토리를 지정한다. HTTP 서버가 로컬 머신에 있다면 'c:\WhttpsWcodebaseW' 과 같은 패스 이름이 된다.

일단 이들 옵션을 지정하면, Project|Web Deploy 메뉴를 선택하여 코드를 서버로 복사할 수 있다. Web Deploy 를 선택하면 델파이는 컨트롤이 포함될 웹 페이지를 HTML 디렉토리에 복사하고, 컨트롤을 코드베이스 디렉토리에 복사한다.

그러면 앞에서 간단하게 만들어본 액티브 폼 .ocx 파일을 배포해 보도록 하자.

Project|Web Deployment Options 메뉴를 선택하고, 대화 상자의 내용을 다음과 같이 설정한다.



물론 여기에서 보여준 디렉토리 정보는 개발자의 컴퓨터의 .ocx 파일의 위치에 해당되는 것이므로 모두들 다를 것이다. 아마도 제공되는 CD-ROM 의 HTML 파일의 내용도 이렇게 설정되어 있으므로 액티브 폼의 디렉토리가 'C:\WBookWChap29'가 아니면 액티브 폼의 내용을 IE 로 볼 수 없을 것이다. 그럴 때에는 HTML 파일을 열어서 자신의 디렉토리에 맞게 변경하기만 하면 된다.

여기서 URL 을 'c:/Book/Chap29'로 설정하였는데, 물론 앞에서 간단히 설명했듯이 웹 서버

의 위치를 알면 웹 서버의 URL 이름을 이용하여 디렉토리까지만 설정하면 된다. 여기서는 개발자가 일단 웹 서버가 없어도 컨트롤을 쉽게 보고 디버깅할 수 있도록 개발자의 컴퓨터의 물리적인 디렉토리를 그대로 지정하였다. 여기서 주의할 것은 URL 명명 규칙상 ‘W’가 아닌 ‘/’을 사용한다는 것이다. 그리고, 만약에 서버가 리모트 웹 서버이면, 로컬 컴퓨터에 HTML 파일과 코드 베이스 파일을 저장하도록 지정한 후, FTP 를 이용해서 웹 페이지에 배포해야 한다. OK 버튼을 클릭하고, Project|Web Deploy 메뉴를 선택하면 지정된 디렉토리에 .htm 파일이 생성될 것이다. 필자의 HTML 파일의 코드는 다음과 같다.

```
<HTML>
```

```
<H1> Delphi 4 ActiveX Test Page </H1><p>
```

You should see your Delphi 4 forms or controls embedded in the form below.

```
<center><P>
```

```
<OBJECT
```

```
    classid="clsid:761B37B2-31E6-11D2-9774-0000E838052E"
```

```
    codebase="C:/Book/Chap29/ActiveFormProj1.ocx"#version=1,0,0,0
```

```
    width=372
```

```
    height=200
```

```
    align=center
```

```
    hspace=0
```

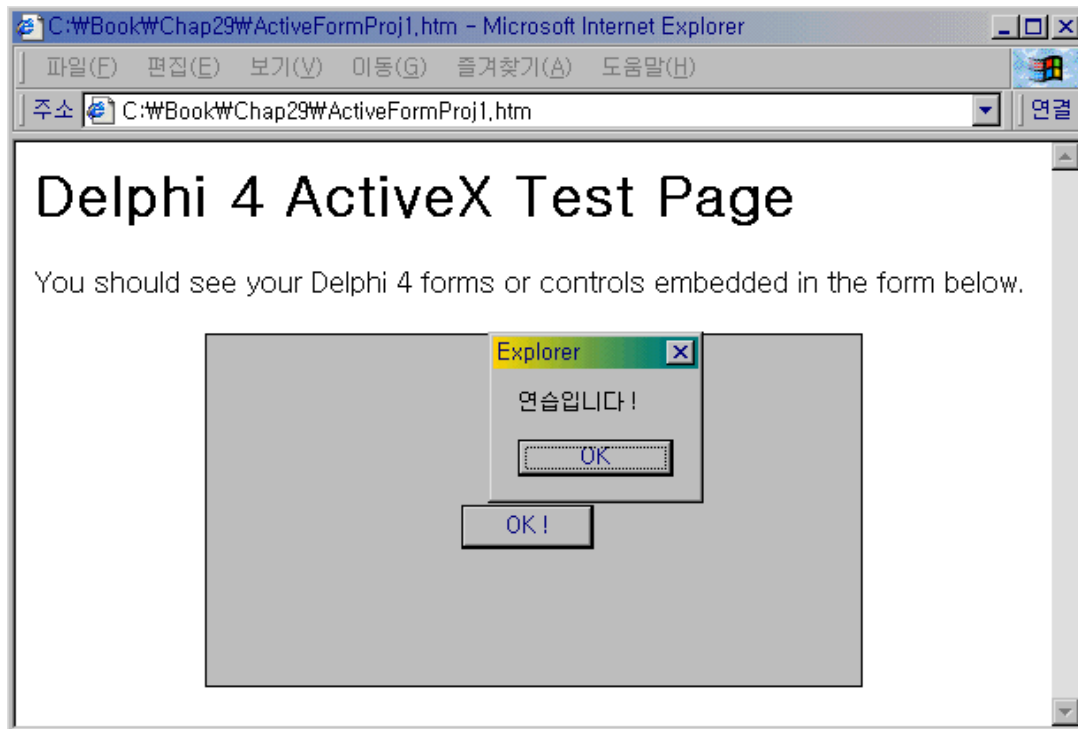
```
    vspace=0
```

```
>
```

```
</OBJECT>
```

```
</HTML>
```

물론, HTML 태그를 이용하여 <OBJECT>, </OBJECT> 사이의 width, height 등의 내용을 변경하여 폼의 크기도 변경할 수 있고, 그 밖의 다른 내용도 에디터를 이용하여 쉽게 변경이 가능하다. 그러면, IE 를 띄우고 만들어진 HTML 파일을 더블 클릭하여 실제 액티브 폼을 띄워 보자. 이때 다음에 설명할 코드 사인을 하지 않은 .ocx 파일이 포함되지 않은 액티브 폼이므로 IE 의 보기|인터넷 옵션 메뉴를 선택하고 보안 레벨을 ‘낮음’으로 선택해야 액티브 폼을 볼 수 있다. 보안 레벨을 ‘보통’ 이상으로 설정한 경우에는 코드 사인이 된 액티브 X 컨트롤만 볼 수 있다.



## 코드 사인 (Code Signing)

코드 사인의 가장 중요한 요소는 인증을 부여할 수 있는 업체에게서 코드키(code key)를 부여 받는 것이다. 마이크로소프트에서는 테스트의 목적으로 키를 생성할 수 있는 방법을 제공하고 있다. 마이크로소프트의 웹 사이트에 가면 이에 대한 정보를 얻을 수 있으며, 'Authenticode 2.0'에 대한 글을 읽어 보면 도움이 될 것이다.

델파이 4 는 프로젝트 옵션 페이지에서 code signature 정보를 지정할 수 있다. Project|Web Deployment Options 대화 상자의 Project 페이지에서 Code Sign Project checkbox 를 체크하고, Code Signing page 에서 배포할 라이선스 파일의 이름과 private key 를 지정하면 끝난다.

Application name, optional company URL 필드는 컨트롤이 다운로드될 때 이곳에 회사의 정보를 적어 넣으면 된다.

또한, 암호화 알고리즘을 선택할 수 있는데 디폴트로는 가장 흔히 사용되는 MD5 가 선택되어 있을 것이다. MD5 는 RSA Data Security 에 의해 개발된 해싱 알고리즘으로 128 비트 해쉬(hash) 값을 만들어 낸다. 다른 것으로 SHA 1 을 선택할 수 있는데, 이것은 NIST(National Institute of Standards and Technology)와 NSA(National Security Agency)에 의해 개발된 해싱 알고리즘으로 160 비트 해쉬 값을 만들어 낸다.

## 웹 보안 (Web Security)

액티브 X 컨트롤은 강력하고 편리하지만 사용자에게 심각한 위험을 초래할 수도 있다. 액티브 X 컨트롤을 만드는 사람이라면 꼭 고려해야 할 사항이 있어서 여기에 몇가지 원칙을 나열하였다.

1. 당연한 말이지만 해로운 액티브 X 컨트롤을 만들면 안된다.
2. 만들어진 컨트롤이 함부로 변형될 수 있으면 안된다.
3. 아무나 함부로 사용할 수 있도록 해서는 안된다. 가능하면 인증을 받고 패스워드에 신경을 쓰도록 한다.
4. 가능하면 어떤 사람이, 어디서 컨트롤을 다운로드 받았는지 파악할 수 있도록 서버를 설정한다.

액티브 X 컨트롤은 인터넷 보다는 사실 인트라넷 환경에 적절하다고 할 수 있다. 인트라넷에 액티브 X 컨트롤을 배포하는 사람은 다음과 같은 사항에 신경을 쓰는 것이 좋다.

1. Code signature 는 완벽한 보안이 되지 못한다. 그러므로 과신은 금물이다.
2. 아주 믿을만하지 못한 사람이 만든 HTML 파일에 포함된 액티브 X 컨트롤을 받아들이지 않는 것이 좋다.

## 추가적인 배포 옵션

그 밖에도 웹 배포를 할 때 지정할 수 있는 옵션 들이 많은데, 여기에 대해서 조금 더 알아보도록 하자.

CAB 파일을 설정하면 윈도우 95 에서 사용되던 캐비넷이라는 압축 파일로 파일 라이브러리를 관리할 수 있도록 할 수 있다. 캐비넷 압축은 다운로드 속도를 많이 줄여 주므로, 인터넷 환경에서는 매우 유용하게 사용될 수 있다. 설치 도중에 브라우저가 캐비넷에 저장된 파일의 압축을 해제해서 저장하므로, 수행 성능의 저하나 비효율성은 거의 없다고 생각해도 된다.

보통 액티브 X 컨트롤이나 액티브 폼을 텔파이에서 개발하여 배포할 때에는 처음에 런타임 패키지를 배포하고, 작아진 컨트롤을 업그레이드로 배포하는 것이 효율적이다. 이렇게 하기 위해서 처음에 패키지에 대한 옵션을 설정할 수 있다. 참고로, 텔파이에 의해 제공되는 모든 패키지는 기본적으로 볼랜드에서 코드 사인한 것이므로 쉽게 CAB 파일로 통합해서 배포할 수 있다.

이렇게 액티브 X 컨트롤을 배포할 때 패키지나 다른 추가적인 파일과 함께 배포되어야 한다면, .INF 파일이 자동으로 생성된다. 이 파일은 다운로드할 파일과 액티브 X 라이브러리를



설정하는 방법 등에 대한 내용을 지정하게 된다.

- 옵션의 조합

다음 테이블은 패키지와 CAB 파일 압축, 코드 사인에 대한 정보를 옵션의 체크 박스에서 선택할 때의 결과에 대해서 정리한 것이다.

패키지/ 추가 파일	CAB 압축	코드 사인	결 과
X	X	X	.OCX 파일 단독
X	X	O	.OCX 파일 단독
X	O	X	.OCX 파일을 포함한 CAB 파일
X	O	O	.OCX 파일을 포함한 CAB 파일
O	X	X	.INF 파일과 .OCX 파일 그리고 추가적인 파일과 패키지
O	X	O	.INF 파일과 .OCX 파일 그리고 추가적인 파일과 패키지
O	O	X	.INF 파일, .OCX 파일을 포함한 .CAB 파일, 그리고 추가적인 파일과 패키지를 포함한 CAB 파일
O	O	O	.INF 파일을 포함한 CAB 파일, .OCX 파일을 포함한 CAB 파일, 추가적인 파일과 패키지를 포함한 CAB 파일

- Packages, Additional Files 탭

웹 배포 옵션의 Packages, Additional Files 탭에서 어떤 패키지를 배포할 것인지 지정할 수 있다. 특정 패키지의 설정을 변경하려면 Packages used by project 리스트 박스에 나열된 패키지 중에서 변경할 패키지를 선택하면 된다. 기본적으로 이들 탭의 옵션 내용은 동일하다.

CAB 옵션에서 Compress in a separate CAB 옵션은 각 패키지 별로 분리된 .CAB 파일을 생성한다. 이 옵션이 디폴트로 설정되어 있다. Compress in a project CAB 옵션은 프로젝트 .CAB 파일에서의 패키지를 포함하게 하는 옵션이다.

Output 옵션은 패키지가 버전 정보를 가지고 있거나 코드 사인에 대한 내용을 설정하는 옵션으로 Use file VersionInfo 옵션은 패키지 파일의 리소스에 저장된 버전 정보를 .INF 파일에서 사용할 수 있도록 한다. Code sign file 옵션은 패키지나 .CAB 파일의 코드 사인을 한다.

Directory and URL options 옵션에서 Target URL 에디트 박스는 URL 형태로서의 웹 서버의 패키지 위치를 지정한다. 이 에디트 박스가 비어 있으면, 목적 기계에 이미 파일이 존재하는 것으로 간주한다. 즉, 패키지가 지정된 위치에 존재하지 않으면, 액티브 X 컨트롤

의 다운로드가 실패한다. Target directory 에디트 박스에는 웹 서버에서 패키지의 패스를 지정한다.

## 정 리 (Summary)

이번 장을 마치기 전에 관심있는 독자들을 위해 델파이와 COM, 액티브 X 컨트롤에 대해 공부할 거리를 제시하고자 한다. 기본적으로 TActiveXControl 이 지원하는 각종 인터페이스의 정의와 설명을 마이크로소프트의 SDK 등의 자료를 이용해서 숙지하는 것이 필요하며, 그 밖에 앰비언트 프로퍼티(컨테이너에서 제공되는 프로퍼티), 커스텀 레지스트리 엔트리를 추가하는 법 등의 정보를 알아보는 것이 좋겠다.

가장 좋은 공부법은 다소 어렵더라도 Inprise 에서 제공하는 뉴스 그룹을 참조하는 것이 가장 큰 도움이 될 것으로 생각되는데, `borland.public.delphi.activex.controls.writing` 과 `borland.public.delphi.oleautomation` 의 2 사이트가 가장 많은 정보를 가지고 있으며, 이를 읽다 보면 좋은 사이트가 많이 소개되어 있으니 한번씩 둘러보는 것이 도움이 될 것이다.

# 고급 COM 기술의 활용 (I)

## (Using Advanced COM Techniques I.)

이번 장에서는 비교적 고급이라고 할 수 있는 COM 에서 컬렉션을 구현하는 방법과 콜백 함수를 이용하여 인터페이스간 통신을 하는 방법, 그리고 연결점(Connection Point) 인터페이스를 사용하여 이벤트를 구현하는 방법을 예제를 통해 익히도록 한다.

### 컬렉션 객체의 구현

델파이의 컴포넌트들 중에는 여러 개의 서브 아이템을 소유하는 클래스들이 많다. 대표적인 것이 TStringList 로 이 클래스는 리스트 박스, 콤보 박스, 메모 컴포넌트 등에서 Lines 또는 Items 프로퍼티로 접근할 수 있도록 되어 있다. TStringList 는 문자열들을 인덱스로 접근할 수 있도록 허용한 일종의 컬렉션 클래스라고 말할 수 있다.

이 밖에도 TList, TTreeNode 등 의 클래스가 컬렉션의 형태로 이루어져 있다.

#### ● IEnumXXXX 인터페이스

그렇다면, COM 에서 이런 컬렉션을 구현하려면 어떻게 하면 될까 ? COM 에서 컬렉션을 구현하기 위해서는 IEnumXXXX 라는 인터페이스를 구현해야 한다.

이런 IEnumXXXX 와 같은 인터페이스를 열거 인터페이스라고 하며, 대표적으로 구현된 예는 이벤트를 구현하기 위해 사용되는 IConnectionPoint 와 IConnectionPointContainer 에서 이용하는 IEnumConnectionPoints, IEnumConnections 인터페이스를 들 수 있다.

IEnumXXXX 인터페이스는 진정한 인터페이스라고는 할 수 없고, 모든 환경 인터페이스의 요청을 지시하기 위한 문서화 도구(documentation device)이다. 기본적으로 이런 열거형 인터페이스는 Next, Skip, Reset, Clone 이라는 4 가지 메소드를 지원한다. 다음의 코드는 예제에서 사용할 IEnumVariant 인터페이스의 선언부이다.

```
IEnumVariant = interface(IUnknown)
    ['{00020404-0000-0000-C000-000000000046}']
    function Next(celt: Longint; out elt:
        pceltFetched: PLongint): HRESULT; stdcall;
    function Skip(celt: Longint): HRESULT; stdcall;
    function Reset: HRESULT; stdcall;
    function Clone(out Enum: IEnumVariant): HRESULT; stdcall;
```

end;

클라이언트는 항목을 가지고 있는 배열을 할당하고, 이것을 배열의 크기와 함께 Next 메소드로 전달한다. 여기에서 배열의 이름이 들어가는 파라미터가 elt 이다. elt 는 아마도 'count of elements of T'의 약자일 것으로 생각되는데, 여기서 T는 데이터 형을 의미하며 배열의 크기를 지정한다. pceltFetched 파라미터는 카운터 변수로 사용된다. 이런 클라이언트의 요구가 있으면 서버는 지정된 항목의 수만큼 배열을 채우고자 시도할 것이고, 카운터에 실제로 들어간 항목의 수를 반환하게 된다.

Reset 메소드는 클라이언트가 첫번째 항목으로부터 다시 열거를 시작하게 할 때 사용되며, 열거하는 도중에 항목을 건너뛰는 때에는 Skip 메소드를 사용한다. 마지막으로 Clone 메소드는 해당되는 IEnumXXXX 인터페이스에게 현재의 열거자 객체의 복사본을 생성해준다. 그러면 실제로 예제를 통해 이를 익히도록 하자.

#### ● 컬렉션을 구현한 자동화 서버

이번에 작성할 예제 자동화 서버는 필자가 인터넷에서 구한 Coll\_Demo 라는 프로그램을 참고하여 작성한 것인데, 아쉽게도 작성자에 대한 정보가 없어서 이를 구체적으로 알리지 못한다는 것을 미리 알려둔다.

이 예제는 파일을 찾아주는 역할을 하는 자동화 서버를 제작하는데, 해당되는 파일들의 정보를 IFileObject 라는 인터페이스에 저장하고 이들의 컬렉션을 관리하는 IFileObjects 인터페이스와 메인 인터페이스 역할을 하는 IFileFinder 인터페이스의 3 가지 인터페이스를 이용한다.

예제를 직접 작성하기 전에, 먼저 이들 인터페이스를 디자인하도록 한다.

```
IFileObject = interface(IDispatch)
    function Get_Name: WideString; safecall;
    function Get_FullName: WideString; safecall;
    function Get_Size: Integer; safecall;
    property Name: WideString read Get_Name;
    property FullName: WideString read Get_FullName;
    property Size: Integer read Get_Size;
end;
```

가장 기본적인 요소가 되는 인터페이스가 IFileObject 이다. 3 가지 프로퍼티를 지원하는데, 이들은 모두 읽기 전용이다. 파일의 전체 경로를 포함한 이름을 저장하는 FullName, 패스 정보를 제외한 파일 이름인 Name, 파일의 크기 정보인 Size 프로퍼티를 가진다.

```

IFileObjects = interface(IDispatch)
    function _NewEnum: IUnknown; safecall;
    function Get_Item(Index: Integer): IFileObject; safecall;
    function Get_Count: Integer; safecall;
    property Item[Index: Integer]: IFileObject read Get_Item;
    property Count: Integer read Get_Count;
end;

```

IFileObjects 인터페이스는 기본적으로 IFileObject 인터페이스를 요소로 한 컬렉션 역할을 하게 된다. \_NewEnum 메소드가 여기서 중요한 역할을 하는데, IEnumVariant 인터페이스를 구현한 클래스를 생성해서 여기에 접근할 수 있도록 IUnknown 인터페이스로 형변환하여 결과값을 리턴한다. 참고로 IEnumVariant 인터페이스는 ActiveX.pas 유닛에 선언되어 있으므로 이를 따로 선언할 필요는 없다.

Item 프로퍼티는 인덱스를 가진 프로퍼티로, 해당 인덱스의 IFileObject 인터페이스를 반환한다. Count 프로퍼티는 전체 IFileObject 인터페이스의 수를 반환한다.

```

IFileFinder = interface(IDispatch)
    function FindFiles(const Spec: WideString): IFileObjects; safecall;
end;

```

IFileFinder 인터페이스는 FindFiles 메소드를 호출할 때, 파라미터로 'c:\W\*.\*)'와 같이 파일을 열거할 조건을 문자열로 넘겨주면 IFileObjects 인터페이스를 반환하는 역할을 한다.

그러면, 이들을 실제로 구현해 보도록 하자.

먼저 File|New 메뉴의 ActiveX 탭에서 ActiveX Library 아이콘을 더블 클릭하여 새로운 DLL 프로젝트를 시작한다. 그리고, 자동화 객체를 선언하도록 한다. File|New 메뉴의 ActiveX 탭에서 Automation Object 아이콘을 더블 클릭하고 클래스 이름으로 FileFinder를 입력하고 OK 버튼을 클릭하면 IFileFinder 인터페이스가 타입 라이브러리 에디터에 추가될 것이다. 마찬가지로 다시 File|New 메뉴의 ActiveX 탭에서 Automation Object 아이콘을 더블 클릭하고 클래스 이름을 FileObjects를 입력하고 OK를 클릭하고, 다시 한번 반복하여 FileObject를 추가한다. 이렇게 하면 IFileFinder, IFileObjects, IFileObject 인터페이스와 이들에 대한 CoClass 들이 선언될 것이다.

그러면, 인터페이스에 메소드와 프로퍼티를 추가해보자.

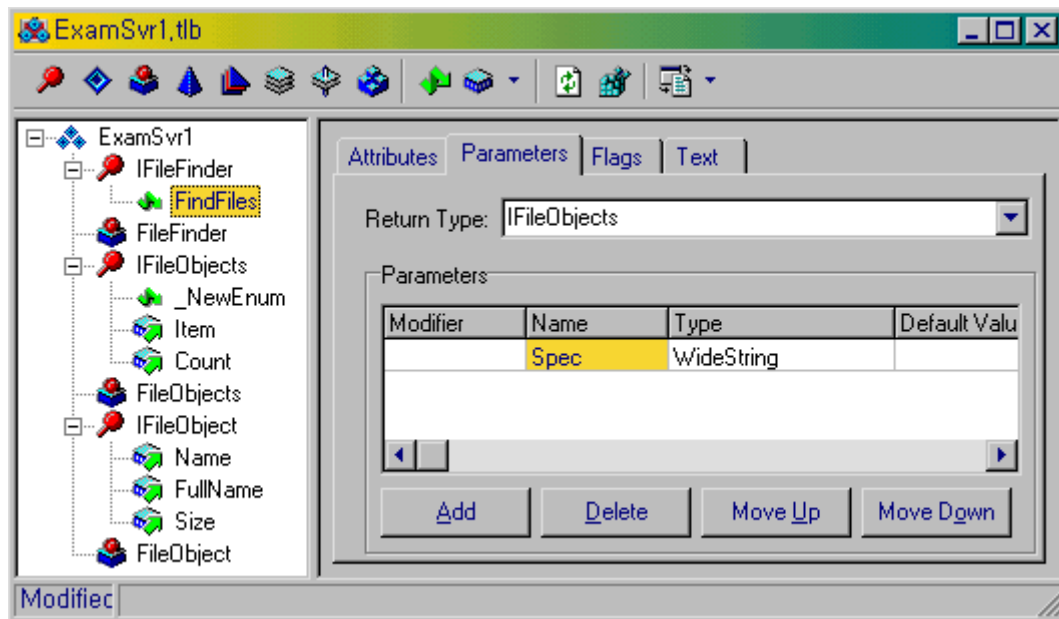
IFileFinder 인터페이스를 선택하고 New Method 버튼을 클릭한다. 메소드의 이름을 FindFiles로 설정하고, Parameters 탭을 선택한다. Return Type 콤보 박스에서 IFileObjects 인터페이스를 선택하고, Add 버튼을 클릭하여 파라미터를 추가한다. 추가된

파라미터의 Name 을 ‘Spec’으로 설정하고, Type 을 콤보 박스에서 WideString 으로 설정한다.

IFileObjects 인터페이스에서는 \_NewEnum 메소드와 Item, Count 프로퍼티를 추가해야 한다. 마찬가지로 방법으로 추가하면 되는데, 이때 Item 과 Count 프로퍼티는 모두 읽기 전용으로 설정해야 하므로 Attributes 탭의 Invoke Kind 콤보 박스에서 property Get 을 선택해야 한다. Item 프로퍼티는 IFileObject, Count 프로퍼티는 Integer 로 Type 을 설정한다.

\_NewEnum 메소드는 Parameters 탭에서 Return Type 으로 IUnknown 을 선택한다.

이제 마지막으로 IFileObject 인터페이스의 프로퍼티를 추가하도록 하자. New Properties 버튼을 클릭하여 Name, FullName, Size 프로퍼티를 추가한다. 이들은 모두 읽기 전용이므로 Attributes 탭의 Invoke Kind 콤보 박스에서 property Get 을 선택해야 한다. Name, FullName 프로퍼티는 WideString, Size 프로퍼티는 Integer 로 Type 을 설정한다. 이렇게 해서 작성된 타입 라이브러리 에디터의 형태는 다음과 같을 것이다.



타입 라이브러리 에디터를 닫으면 자동화 객체에 대한 유닛이 추가되는데, 이들을 모두 적당한 이름으로 저장하도록 하자.

먼저 IFileFinder 인터페이스를 먼저 구현하도록 하자. FindFiles 메소드만 구현하면 되는데, FindFiles 메소드는 Spec 파라미터의 내용을 바탕으로 IFileObjects 인터페이스를 반환하는 역할을 하게 되므로 IFileObjects 인터페이스를 구현한 유닛을 uses 절에 추가해야 한다. 여기서는 U2\_ExamSvr1Impl.pas 유닛을 추가한다.

FindFiles 메소드는 다음과 같이 구현한다.

```
function TFileFinder.FindFiles(const Spec: WideString): IFileObjects;
```

```
begin
    Result := TFileObjects.Create(Spec);
end;
```

IFileObjects 인터페이스의 구현 방법을 알아보기 전에, 먼저 구현하기 쉬운 IFileObject 인터페이스를 먼저 구현하도록 하자.

IFileObject 인터페이스를 구현한 TFileObject 클래스에 프로퍼티의 정보를 저장할 FName, FFullName, FSize 변수를 private 섹션에 추가하고, constructor 인 Create 메소드를 public 섹션에 다음과 같이 추가한다.

```
TFileObject = class(TAutoObject, IFileObject)
private
    FFullName: String;
    FName: String;
    FSize: Integer;
public
    constructor Create(const sr: TSearchRec);
protected
    function Get_FullName: WideString; safecall;
    function Get_Name: WideString; safecall;
    function Get_Size: Integer; safecall;
end;
```

이들 메소드는 다음과 같이 비교적 쉽게 구현할 수 있다. 참고로 TFileObject 클래스의 constructor 는 IFileObjects 인터페이스에 의해서 호출되므로, 파라미터인 sr 등을 조작할 필요는 없다.

```
constructor TFileObject.Create(const sr: TSearchRec);
begin
    inherited Create;
    FFullName := sr.Name;
    FName := ExtractFilename(FFullName);
    FSize := sr.Size;
end;

function TFileObject.Get_FullName: WideString;
```

```
begin
```

```
    Result := FFullName;
```

```
end;
```

```
function TFileObject.Get_Name: WideString;
```

```
begin
```

```
    Result := FName;
```

```
end;
```

```
function TFileObject.Get_Size: Integer;
```

```
begin
```

```
    Result := FSize;
```

```
end;
```

이제 가장 구현하기 어려운 IFileObjects 인터페이스를 구현하도록 한다.

IFileObjects 인터페이스를 구현하기 위해서는 IEnumVariant 인터페이스를 구현해 주어야 한다. 이를 위해서 uses 절에 ActiveX.pas 유닛을 추가한다. 그 밖에도 이들을 구현하기 위해서 여러가지 함수를 사용하게 되는데 여기에 필요한 SysUtils.pas, Windows.pas 유닛과 TList 클래스를 사용하기 위해 Classes.pas 유닛을 uses 절에 추가해야 된다. 그리고, 앞에서 구현한 TFileObject 클래스를 이용하게 되므로 IFileObject 인터페이스를 구현한 유닛을 uses 절에 추가한다 (여기서는 U3\_ExamSvr1Impl.pas).

먼저, IEnumVariant 인터페이스를 구현할 TEnumVariant 클래스를 다음과 같이 선언한다.

```
TEnumVariant = class(TInterfacedObject, IEnumVariant)
```

```
private
```

```
    FIndex: Integer;
```

```
    FList: TList;
```

```
    FParent: IUnknown;
```

```
protected
```

```
    function Next(celt: Longint; out elt: pceltFetched: PLongint): HRESULT; stdcall;
```

```
    function Skip(celt: Longint): HRESULT; stdcall;
```

```
    function Reset: HRESULT; stdcall;
```

```
    function Clone(out enum: IEnumVariant): HRESULT; stdcall;
```

```
public
```

```
    constructor Create(aParent: IUnknown; aList: TList);
```

```
end;
```



여기서 열거자 역할을 하는 메소드는 protected 섹션에 선언된 4 개의 메소드이다. 내부적으로 사용하기 위해 FIndex, FParent, FList 변수를 private 섹션에 추가하고, public 섹션에 constructor 로 Create 메소드를 추가한다.

IFileObjects 인터페이스를 선언하는 TFileObjects 클래스에는 TList 클래스의 객체를 저장할 FList 변수를 private 섹션에 추가하고 constructor 와 destructor 로 사용할 Create, Destroy 메소드를 public 섹션에 다음과 같이 추가한다.

```
TFileObjects = class(TAutoObject, IFileObjects)
private
    FList: TList;
public
    constructor Create(const aFileSpec: String);
    destructor Destroy; override;
protected
    function _NewEnum: IUnknown; safecall;
    function Get_Count: Integer; safecall;
    function Get_Item(Index: Integer): IFileObject; safecall;
end;
```

그러면, 먼저 TEnumVariant 클래스를 구현해 보도록 하자. constructor 인 Create 메소드에서는 TEnumVariant 클래스의 Parent 가 되는 클래스와 열거할 항목을 저장할 TList 클래스 변수의 값을 다음과 같이 설정한다.

```
constructor TEnumVariant.Create(aParent: IUnknown; aList: TList);
begin
    inherited Create;
    FParent := aParent;
    FList := aList;
end;
```

그리고, 가장 중요한 Next 메소드는 다음과 같이 구현한다.

```
function TEnumVariant.Next(celt: Longint; out elt: pceltFetched: PLongint): HRESULT;
type
    TVariantArray = packed array[0..0] of OleVariant;
```

```

var
    i: Integer;
begin
    for i := 0 to celt - 1 do
    begin
        VariantClear(TVariantArray(elt)[i]);
    end;

```

앞에서도 설명한 바 있지만, 여기서 elt 는 배열 이름을 가리키며 celt 는 배열의 항목의 수가 된다. 그러므로, 이 코드는 선언한 TVariantArray 라는 OleVariant 의 배열의 값을 초기화하는 역할을 한다.

```

    i := 0;
    while (celt > 0) and (FIndex < FList.Count) do
    begin
        TVariantArray(elt)[i] := IUnknown(FList[FIndex]) as IDispatch;
        Inc(i);
        Dec(celt);
        Inc(FIndex);
    end;

```

이 코드는 FList 의 마지막 항목까지의 객체를 IDispatch 로 TVariantArray 배열에 저장하는 역할을 한다. 즉, TVariantArray 배열의 내용과 FList 변수의 내용을 동기화하는 코드이다.

```

try
    if Assigned(pceltFetched) then
        pceltFetched^ := i;
except
end;

```

제대로 작업이 끝났으면 pceltFetched 파라미터의 값을 설정한다.

```

if celt = 0 then
    Result := S_OK
else

```

```
    Result := S_FALSE;
end;
```

마지막으로 결과값을 반환하면 된다.

Reset, Skip, Clone 메소드는 구현하기가 비교적 쉽다. 다음과 같이 구현하면 된다.

```
function TEnumVariant.Reset: HRESULT;
begin
    FIndex := 0;
    Result := S_OK;
end;
```

```
function TEnumVariant.Skip(celt: Longint): HRESULT;
begin
    while (celt > 0) and (FIndex < FList.Count) do
    begin
        Dec(celt);
        Inc(FIndex);
    end;
    if celt = 0 then
        Result := S_OK
    else
        Result := S_FALSE;
    end;
end;
```

```
function TEnumVariant.Clone(out enum: IEnumVariant): HRESULT;
var
    r: TEnumVariant;
begin
    r := TEnumVariant.Create(FParent, FList);
    r.FIndex := FIndex;
    enum := r;
    Result := S_OK;
end;
```

이렇게 함으로써 FList 의 아이템에 IDispatch 인터페이스로서 IFileObject 인터페이스를 저

장할 수 있게 되었다. FList 에내용을 저장하고, 이들에 접근할 때 자동으로 IEnumVariant 인터페이스의 메소드를 호출하여 사용하게 된다.

그러면, FList 를 이용하여 IFileObject 인터페이스객체를 관리하는 TFileObjects 클래스를 구현하도록 하자.

가장 중요한 것이 constructor 인 Create 메소드이다. 이 메소드는 IFileFinder 인터페이스의 FindFiles 메소드에 의해서도 호출되며, 실제로 FindFiles 메소드에서 파라미터로 사용된 문자열을 바탕으로 파일을 검색해서 파일의 내용을 바탕으로 TFileObject 클래스를 생성하고, IFileObject 인터페이스를 FList 에 저장한다.

Create 메소드를 다음과 같이 구현한다.

```
constructor TFileObjects.Create(const aFileSpec: String);
```

```
var
```

```
    r: Integer;
```

```
    sr: TSearchRec;
```

```
    Obj: IFileObject;
```

```
begin
```

```
    inherited Create;
```

```
    FList := TList.Create;
```

```
    r := FindFirst(aFileSpec, faAnyFile, sr);
```

```
    try
```

```
        while r = 0 do
```

```
        begin
```

```
            Obj := TFileObject.Create(sr);
```

```
            Obj._AddRef;
```

```
            FList.Add(Pointer(Obj));
```

```
            r := FindNext(sr);
```

```
        end;
```

```
    finally
```

```
        SysUtils.FindClose(sr);
```

```
    end;
```

```
end;
```

그다지 어렵지 않은 코드이므로 자세한 설명은 생략한다. 주의해야할 부분은 Obj 변수를 IFileObject 형으로 선언한 뒤에 이를 TFileObject.Create 메소드에 찾은 파일의 TSearchRec 데이터 형 데이터를 저장한 sr 변수를 파라미터로 사용하여 호출하는 부분과, IFileObject 인터페이스를 사용하기 때문에 \_AddRef 메소드를 호출한 부분이다.

이 부분에서 실수를 하면 COM 의 핵심 부분이라고 할 수 있는 참조 계수관리에 실패하게 된다.

참고: COM 참조계수 관리

델파이는 기본적으로 참조계수 관리를 자동으로 해준다. 그렇지만, 여기에는 일반적인 참조 계수와는 다른 델파이 만의 규칙이 있기 때문에, 이를 잘 숙지하고 있어야 한다.

만약 변수의 데이터 형으로 IUnknown 을 이용한 경우에는 델파이가 자동으로 참조계수 관리를 하므로 AddRef 나 Release 메소드를 사용하면 안된다.

이때 주의할 것은 어떤 식으로 COM 객체를 사용하는지 여부에 따라 참조계수가 증가하기도 하고, 변화를 주지 않을 수도 있다. 다음의 예제 코드를 참고하기 바란다.

var

MyIxxxVariable: ISomeCOMInterface;

....

MyIxxxVariable := TMyCOMObject.Create; //내부적으로 AddRef 가 호출된다.

SomeAPIFunc(MyIxxxVariable);

이 경우에는 내부적으로 ISomeComInterface 에 대한 참조계수가 증가하기 때문에, 함수를 호출하는데 문제가 없다. 그렇지만 다음의 코드는 사정이 다르다.

var

MyDelphiVariable: TMyCOMObject;

....

MyDelphiVariable := TMyCOMObject.Create; //AddRef 가 호출되지 않는다.

SomeAPIFunc(MyDelphiVariable);

이 경우에는 API 함수의 파라미터로 직접 COM 객체를 사용해도 형변환을 하기 때문에 문제가 없지만 AddRef 를 호출하지 않기 때문에, 문제가 된다.

이를 해결하기 위해서는 다음과 같은 코드를 사용하면 된다.

var

MyDelphiVariable: TMyCOMObject;

....

MyDelphiVariable := TMyCOMObject.Create;

SomeAPIFunc(MyDelphiVariable as ISomeCOMInterface);

as 연산자에 의해 AddRef 가 호출되므로 잘 동작하게 된다.

그런데, IFileObject 인터페이스의 \_AddRef 를 호출한 이유는 델파이의 FList 아이템으로 추가할 경우 이들에 의해 인터페이스가 추가, 삭제될 때 참조계수의 변화를 주어야 하기 때문이다. 그러므로, 나중에 IFileObject 인터페이스를 아이템에서 삭제할 경우 \_Release 메소드를 호출해야 한다.

그리고, FList 의 Add 메소드를 호출할 때 IFileObject 인터페이스를 저장한 Obj 변수를 Pointer 형으로 형변환하여 저장하면 된다. 즉, 이렇게 FList 에 아이템을 추가할 때 Pointer 형으로 형변환하는 것으로는 \_AddRef 가 호출되지 않기 때문에, 그 이전에 \_AddRef 메소드를 명시적으로 호출하는 것이다.

TFileObjects 클래스의 \_NewEnum 메소드는 다음과 같이 구현한다.

```
function TFileObjects._NewEnum: IUnknown;  
begin  
    Result := TEnumVariant.Create(Self, FList) as IUnknown;  
end;
```

즉, 열거를 담당하는 aParent 로 Self(TFileObjects)를 넘기고 열거할 항목을 저장할 배열로 FList 변수를 지정하는 것이다.

TFileObjects 클래스의 IFileObject 인터페이스를 항목으로 제공하는 Item 프로퍼티와 항목의 수를 제공하는 Count 프로퍼티는 Get\_Item, Get\_Count 메소드로 다음과 같이 구현할 수 있다.

```
function TFileObjects.Get_Count: Integer;  
begin  
    Result := FList.Count;  
end;
```

```
function TFileObjects.Get_Item(Index: Integer): IFileObject;  
begin  
    Assert((Index > 0) and (Index <= FList.Count));  
    Result := IFileObject(FList[Index - 1]);  
    Result._AddRef;  
end;
```

여기에서도 주의할 것은 IFileObject 인터페이스를 이용하는 클라이언트를 위해서 \_AddRef 를 호출한다는 것이다.

마지막으로 Destroy 메소드를 다음과 같이 구현하면 된다.

```
destructor TFileObjects.Destroy;
var
    i: Integer;
begin
    for i := 0 to FList.Count - 1 do
        IUnknown(FList[i])._Release;
    FList.Free;
    inherited Destroy;
end;
```

Destroy 메소드는 이와 같이 FList 에 저장된 IFileObject 인터페이스를 모두 \_Release 메소드를 호출하여 해제하는 것이 중요하며, 동시에 생성한 FList 클래스를 해제하면 된다.

이것으로 TFileObjects 클래스의 구현이 모두 끝났다. 쉽지 않은 내용이지만, IEnumXXXX 인터페이스에 대해서는 연결점에 대해 설명하면서 다시 다루게 될 것이다.

그러면, 프로젝트를 컴파일하고 Run|Register ActiveX Server 메뉴를 선택하여 자동화 서버를 등록하도록 한다.

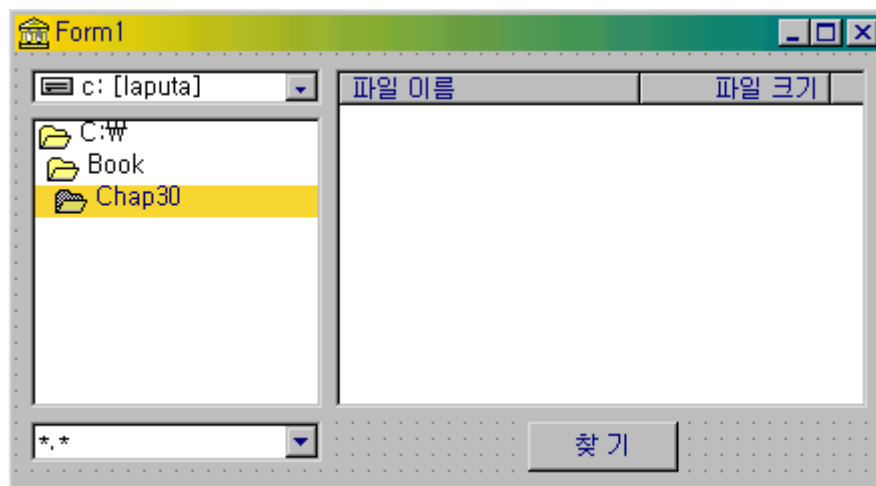
#### ● 클라이언트 어플리케이션의 제작

그러면, 제작한 자동화 서버를 이용해서 디렉토리 브라우저 기능을 하는 클라이언트 어플리케이션을 하나 만들어 보자.

먼저 폼 위에 TDriveComboBox, TDirectoryListBox, TComboBox, TListView, TButton 컴포넌트를 하나씩 올려 놓자.

DriveComboBox1 의 DirList 프로퍼티는 DirectoryListBox1 으로 설정한다. 그리고 ComboBox1 의 Items 프로퍼티 에디터를 이용하여 \*, \*.txt, \*.exe, \*.dll 을 기본적으로 추가하여 이들을 이용하거나 직접 입력이 가능하도록 하자. 버튼 컴포넌트는 Caption 프로퍼티를 ‘찾기’로 설정한다.

ListView1 컴포넌트는 먼저 ViewStyle 프로퍼티를 vsReport 로 설정한다. 그리고, Columns 프로퍼티 에디터를 이용하여 2 개의 새로운 컬럼을 추가하고 이들의 Caption 프로퍼티를 ‘파일 이름’과 ‘파일 크기’로 설정한다. 이들의 Alignment 프로퍼티를 각각 alLeft, alRight 로 설정하고 다음 그림과 같이 적절하게 Width 프로퍼티를 조절한다.



ExamSvr1 의 타입 라이브러리를 사용해야 하므로, uses 절의 ExamSvr1\_TLB.pas 유닛을 추가하도록 한다.

그리고 전역 변수로 IFileFinder 인터페이스 변수를 다음과 같이 선언한다.

```
var
  Form1: TForm1;
  FileFinder: IFileFinder;
```

그리고, 폼의 OnCreate 이벤트 핸들러에서 FileFinder 변수에 early 바인딩을 이용하여 값을 대입한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FileFinder := CoFileFinder.Create;
end;
```

마지막으로 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 구현하면 클라이언트 어플리케이션은 완성된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileObjects: IFileObjects;
  FileObject: IFileObject;
  i: Integer;
  FileItem: TListItem;
```

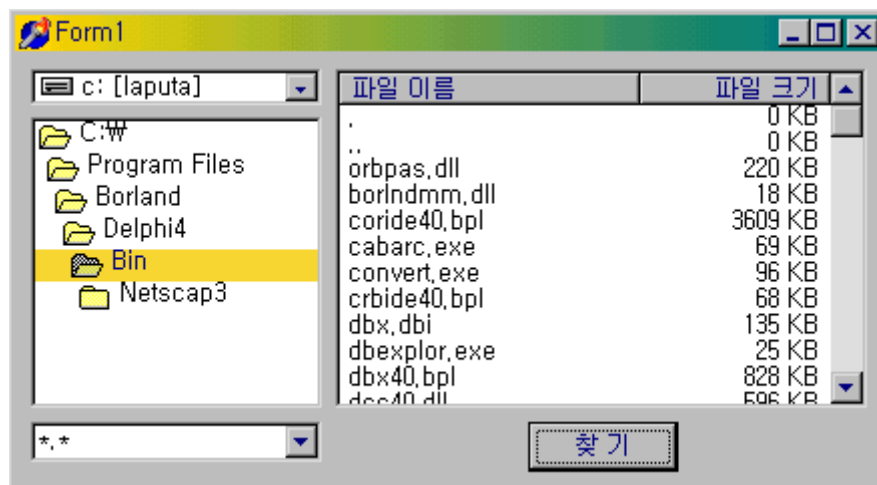


```

begin
  ListView1.Items.Clear;
  FileObjects
    := FileFinder.FindFiles(DirectoryListBox1.Directory + 'W' + ComboBox1.Text);
  for i := 1 to FileObjects.Count do
    begin
      FileObject := FileObjects.Item[i];
      FileItem := ListView1.Items.Add;
      FileItem.Caption := FileObject.Name;
      FileItem.SubItems.Add(IntToStr(FileObject.Size div 1024) + ' KB');
    end;
  end;
end;

```

그러면 클라이언트 어플리케이션을 컴파일하고 실행해보자. 그리고, 디렉토리를 선택한 후 콤보 박스에 파일 찾기에서 입력할 수 있는 여러 가지 옵션을 주고 ‘찾기’ 버튼을 클릭하면 해당되는 파일 들을 리스트 뷰에서 볼 수 있을 것이다.



## COM 에서의 콜백 함수 활용

COM 콜백 인터페이스를 활용하면 COM 서버 컴포넌트가 클라이언트 어플리케이션에 존재하는 객체의 메소드를 호출할 수 있게 된다. 이때 콜백은 이렇게 서버에 중요한 변화가 나타났을 때마다 클라이언트에 이를 알리는 역할을 하게 된다.

콜백에 대해서는 후킹 예제와 함께 제 7 부에서 더 자세하게 다루게 될 것이다.

이러한 콜백을 잘 활용하면 멀티-유저 환경의 클라이언트-서버 데이터베이스 어플리케이션에서 많은 수의 클라이언트에 데이터 변경 등에 대한 여러가지 조작을 할 수 있게 된다.

## 연결점(Connection point) 방법론

실제로 연결점에 대한 개념을 이해하는 것은 그다지 어려운 것이 아니다. 그렇지만 이를 어렵게 느끼게 하는 것은 몇 가지의 기술적인 용어가 많이 나오기 때문이다.

연결점의 기본적인 개념은 클라이언트 객체와 서버 객체가 서로 표준 프로토콜을 이용해서 쉽게 통신을 하게 만들자는 것이다. 이때 프로토콜에서 클라이언트 객체가 서버 객체에게 특정 콜백 인터페이스에 대해서 알고 있는지 묻게 되고, 서버 객체는 여기에 대해 긍정, 또는 부정의 반응을 할 수 있다. 서버 객체가 긍정의 답변을 하게 되면, 클라이언트는 자신의 콜백 인터페이스를 제공하고, 이를 이용해서 통신을 할 수 있도록 요청하게 된다. 이때 부터 서버와 클라이언트는 서로의 메소드를 호출할 수 있게 된다. 이 개념을 이용하면 대단히 유연한 어플리케이션을 개발할 수 있게 된다. 클라이언트는 특정 콜백 인터페이스를 지원하는 특정 서버 객체에 대해 자세히 알 필요가 없으며, 단지 어느 서버 객체이나 자신이 구현할 수 있는 인터페이스를 지원하는지 여부 만을 알아보고 서버가 긍정의 답변을 할 경우에만 통신을 하면 된다.

서버의 관점에서 볼 때 이러한 콜백 인터페이스를 outgoing 인터페이스라고 한다. 이는 인터페이스가 클라이언트에서 구현되며, 서버에 의해 사용되기 때문이다. 반대로 서버에서 구현되고 클라이언트에 의해 사용되는 인터페이스는 incoming 인터페이스라고 한다. 최소한 하나 이상의 outgoing 인터페이스를 지원하는 서버 객체를 연결가능 객체(connectable object), 또는 소스(source)라고 하며, 콜백 인터페이스를 구현하는 클라이언트 객체를 싱크(sink)라고 한다. 클라이언트가 콜백 인터페이스를 이용해서 연결가능 객체에 연결하기 위해서, 연결가능 객체는 반드시 특정 인터페이스에 대한 연결점(connection point)을 구현해야 한다. 서버 객체가 둘 이상의 outgoing 인터페이스를 지원하는 경우도 많은데, 이런 경우에는 여러 종류의 클라이언트에 대한 여러 개의 연결점을 구현해야 한다.

COM 객체는 연결점을 구현하기 위해 IConnectionPointContainer 와 IConnectionPoint 인터페이스를 사용한다. 서버 객체에 연결하고자 하는 클라이언트 객체는 일단 서버 객체에 IConnectionPointContainer 인터페이스를 질의한다. 서버가 유효한 인터페이스를 전달하면, 클라이언트는 이 인터페이스의 FindConnectionPoint 메소드를 호출한다. 이때 파라미터로 클라이언트가 구현하고 있는 outgoing 인터페이스의 인터페이스 ID(IID)를 넘기게 된다. 서버가 넘어온 인터페이스를 지원한다면 IConnectionPoint 인터페이스의 포인터를 cp 파라미터에 담아서 돌려주게 되며, 이 파라미터가 클라이언트가 사용하는 연결점이 된다. 마지막으로 클라이언트가 실제 인터페이스를 구현하는 부분의 포인터를 서버에 넘겨주게 되면 연결이 확립된다. 이를 위해, cp 파라미터를 이용해서 클라이언트는 IConnectionPoint 인터페이스의 Advise 메소드를 호출할 수 있는데, 여기에 지원하는 싱크 인터페이스의 포인터를 unkSink 파라미터에 담아서 보내게 된다. Advise 메소드의 dwCookie 파라미터는 서버가 클라이언트에게 반환하는 것으로, 일종의 ID 와 같은 것이다. 이를 이용해서, 클라

이언트가 연결점과의 연결을 해제할 수 있다. 실제로 연결을 해제할 때에는 IConnectionPoint 인터페이스의 UnAdvise 메소드를 사용한다.

쉽게 말해서, 연결점 컨테이너(IConnectionPointContainer)는 서버 객체가 지원하는 모든 연결점들에 대한 목록이다. 이는 서버 객체가 거의 무한대의 outgoing 인터페이스를 지원할 수 있다는 것으로, 이들 인터페이스는 각각 특정 연결점으로 정의된다. 또한, 하나의 연결점은 무한대의 클라이언트를 지원할 수 있다. 즉, 클라이언트가 IConnectionPoint 인터페이스의 Advise 메소드를 호출할 때, dwCookie 파라미터에 각 클라이언트에 대해서 유일한 ID 를 제공하기 때문에, 하나의 연결점에 연결되는 클라이언트 들이라도 서버는 각각을 dwCookie 를 이용해서 구별할 수 있게 된다.

이들 인터페이스는 다음과 같이 선언되어 있다.

```
IConnectionPointContainer = interface
```

```
function EnumConnectionPoints(out enum: IEnumConnectionPoints): HRESULT;
```

```
function FindConnectionPoint(const iid: TIID; out cp: IConnectionPoint): HRESULT;
```

```
end;
```

```
IConnectionPoint = interface
```

```
function GetConnectionInterface(out iid: TIID): HRESULT;
```

```
function GetConnectionPointContainer(out cpc: IConnectionPointContainer): HRESULT;
```

```
function Advise(const unkSink: IUnknown; out dwCookie: Longint): HRESULT;
```

```
function Unadvise(dwCookie: Longint): HRESULT;
```

```
function EnumConnections(out enum: IEnumConnections): HRESULT;
```

```
end;
```

그러면, 실제로 연결점(ConnectionPoint)을 이용하여 이벤트를 지원하는 COM 객체를 생성하고, 이를 활용하는 클라이언트를 간단하게 작성하도록 하자.

텔파이 4 에서는 이벤트를 지원하는 COM 객체를 쉽게 만들 수 있도록 위저드의 기능이 확장 되었다. 그러므로, COM 객체에 이벤트를 지원하게 확장하는 것은 그리 어렵지 않게 구현할 수 있다.

그런데, 이렇게 위저드의 형태로 확장한 이벤트 지원이 반쪽 밖에 없어서 이벤트를 지원하는 COM 객체는 쉽게 만들 수 있으나, 이를 이용하여 실제 이벤트를 지원하는 클라이언트를 제작하는 것은 꽤 어렵다. 보통은 해당 COM 객체에 대한 이벤트 wrapper 클래스를 오브젝트 파스칼에 맞도록 작성하여, 이를 이용하게 된다.

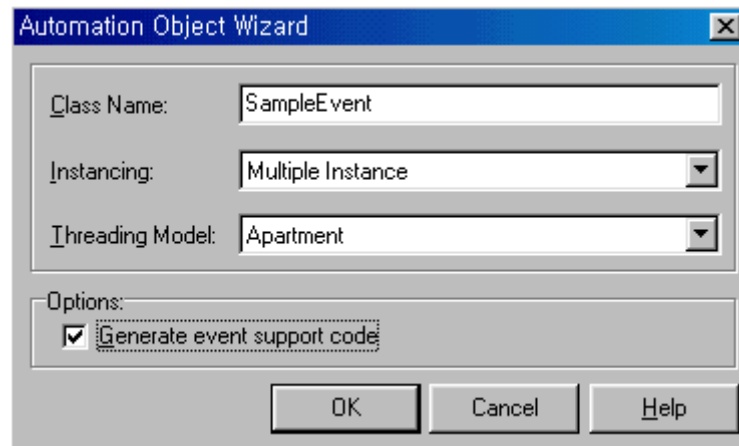
다행히, 이 문제를 쉽게 해결하기 위해 Binh Ly([bly@castle.net](mailto:bly@castle.net))가 이벤트 싱크를 쉽게 처리할 수 있는 유틸리티 유닛과 컴포넌트를 개발하여 프리웨어로 배포하고 있어서 비교적 쉽게 이벤트를 처리할 수 있게 되었다. 그러나, 이런 컴포넌트와 클래스의 이용 방법을 소개

하는 것으로는 이벤트에 대한 명확한 이해가 어렵기 때문에, 다소 복잡하더라도 먼저 간단한 이벤트를 지원하는 COM 객체와 클라이언트를 직접 작성한 뒤에 이들에 대해 따로 설명하도록 하겠다.

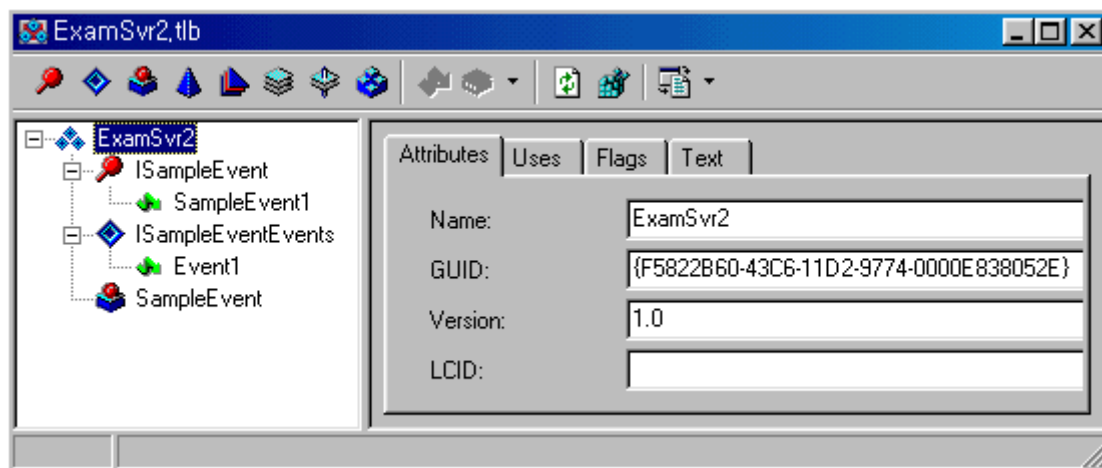
## 이벤트를 지원하는 COM 객체

COM 객체에 이벤트를 지원하게 하는 것은 그리 어렵지 않게 구현할 수 있다.

먼저 File|New 메뉴를 선택한 뒤 ActiveX 탭에서 ActiveX Library 아이콘을 더블 클릭하여 프로젝트 파일을 생성한다. 그리고, File|New 메뉴의 ActiveX 탭에서 Automation Object 아이콘을 더블 클릭하여 자동화 객체를 생성할 대화 상자를 띄우도록 한다. 이 대화 상자에 생성할 클래스 이름을 지정하고, 이벤트를 지원하기 위해서는 다음과 같이 Generate event support code 체크 박스를 선택한다.



OK 버튼을 클릭하면, 타입 라이브러리 에디터가 실행되는데 여기에서 인터페이스의 메소드들을 다음과 같이 설정하도록 한다.



즉, 사용할 메소드를 ISampleEvent 메소드에 추가하고 이벤트는 DispInterface 인 ISampleEventEvents 인터페이스에 Event1 메소드를 추가한다. 이렇게 하고, OK 버튼을 클릭하면 다음과 같은 코드가 자동으로 생성될 것이다.

```
unit U_ExamSvr2;
```

```
interface
```

```
uses
```

```
    ComObj, ActiveX, AxCtrls, ExamSvr2_TLB;
```

```
type
```

```
    TSampleEvent = class(TAutoObject, IConnectionPointContainer, ISampleEvent)
```

```
    private
```

```
        { Private declarations }
```

```
        FConnectionPoints: TConnectionPoints;
```

```
        FEvents: ISampleEventEvents;
```

```
    public
```

```
        procedure Initialize; override;
```

```
    protected
```

```
        { Protected declarations }
```

```
        property ConnectionPoints: TConnectionPoints read FConnectionPoints
```

```
            implements IConnectionPointContainer;
```

```
        procedure EventSinkChanged(const EventSink: IUnknown); override;
```

```
        procedure SampleEvent1; safecall;
```

```
    end;
```

```
implementation
```

```
uses ComServ;
```

```
procedure TSampleEvent.EventSinkChanged(const EventSink: IUnknown);
```

```
begin
```

```
    FEvents := EventSink as ISampleEventEvents;
```

```
end;
```

```

procedure TSampleEvent.Initialize;
begin
    inherited Initialize;
    FConnectionPoints := TConnectionPoints.Create(Self);
    if AutoFactory.EventTypeInfo <> nil then
        FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
            ckSingle, EventConnect);
end;

procedure TSampleEvent.SampleEvent1;
begin

end;

initialization
    TAutoObjectFactory.Create(ComServer, TSampleEvent, Class_SampleEvent,
        ciMultInstance, tmApartment);
end.

```

이 코드를 설명하면, 앞에서도 설명한대로 연결점 컨테이너 인터페이스를 이용하여 이벤트를 구현하게 된다. 클라이언트가 IConnectionPoint 인터페이스의 Advise 메소드를 호출할 때, dwCookie 파라미터에 각 클라이언트에 대해서 유일한 ID 를 제공하기 때문에, 하나의 연결점에 연결되는 클라이언트 들이라도 서버는 각각을 dwCookie 를 이용해서 구별할 수 있게 된다. 그런데, 델파이에서는 TConnectionPoints 클래스에서 이 인터페이스를 구현하기 때문에 이와 같이 간단히 선언하는 것으로 충분하다.

TAutoObject 클래스의 EventSinkChanged 메소드는 다른 인터페이스를 이벤트로 처리할 수 있도록 제공되는 가상 메소드로, 이를 오버라이드하여 파라미터인 EventSink 를 이벤트를 지원하는 Dispinterface 로 타입 캐스팅하여 이를 이용하여 이벤트의 지원이 가능하다.

TAutoObject 클래스의 Initialize 메소드는 객체의 초기화를 할 수 있는 가상 메소드로, 여기에서 TConnectionPoints 클래스의 객체를 생성하고, 이 객체의 CreateConnectionPoint 메소드를 호출하여 파라미터로 지정된 연결점 객체를 생성하여 컨테이너에 포함한다.

개발자가 할 일은 어떤 메소드에서 이벤트를 발생시킬 것인지를 결정하면 된다. 앞에서 델파이가 자동으로 생성한 코드에 의해 FEvents 필드에 ISampleEventEvents 인터페이스에서 정의한 메소드들(이번 예제의 경우 Event)이 포함되므로, 특정 메소드에서 이들 FEvents 의 메소드를 호출하면 특정 메소드를 호출할 때 이벤트가 발생한다.

이번 예제에서는 ISampleEvent 인터페이스의 SampleMethod1 메소드를 호출하면 다른 작업은 하지 않고 Event1 을 호출하여 이벤트만 발생시키도록 다음과 같이 입력한다.

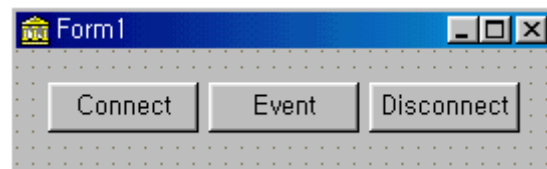
```
procedure TSampleEvent.SampleEvent1;
begin
    FEvents.Event1;
end;
```

이것으로 이벤트를 지원하는 COM 객체 서버의 제작되었다. 실제로 개발자가 한 일은 이벤트 코드를 생성하도록 하는 체크 박스를 선택한 것과 타입 라이브러리에서 인터페이스의 이름 뒤에 Events 가 붙은 Dispinterface 에 이벤트로 사용할 메소드를 추가하는 것, 그리고 인터페이스의 메소드 중에서 이벤트를 발생시킬 곳에서 FEvents 필드에 저장된 이벤트 메소드를 호출한 것 밖에 없다. 나머지는 모두 델파이가 알아서 한다.

이제 이를 컴파일하고, Run|Register ActiveX Server 명령을 선택하여 액티브 X 서버를 등록하도록 한다.

실제로 이렇게 이벤트를 지원하는 COM 객체 서버를 작성하는 것보다, 이벤트를 지원하는 서버를 사용하는 것이 훨씬 더 어렵다. 그러면, 앞에서 작성한 COM 객체 서버를 이용하는 클라이언트 어플리케이션을 만들어 보자.

먼저 폼을 버튼 3 개를 올려 놓고 다음과 같이 디자인한다.



이벤트를 구현한 서버를 이용하기 위해서는 먼저 interface 섹션의 uses 절에 ActiveX.pas 유닛과 ExamSvr2\_TLB.pas 유닛을 추가한다. 또한 implementation 섹션의 uses 절에는 ComObj.pas 유닛을 추가하여 여러 유틸리티 함수를 이용할 수 있도록 한다.

그리고, 이벤트 싱크를 지원하는 클래스와 이를 객체로 사용하여 실제 사용할 수 있도록 하는 클래스를 선언하고 이를 구현해야 한다. 먼저 이벤트 싱크를 지원하는 클래스를 다음과 같이 선언한다.

```
TSampleEventSink = class(TInterfacedObject, IUnknown, IDispatch)
private
    FOwner : TObject;
    FDispatch: IDispatch;
    FDispatchIID: TGUID;
```

```

FConnection: Integer;
FOnEvent: TNotifyEvent;
protected
  //IUnknown
  function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
  //IDispatch
  function GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; Dispid: Pointer): HRESULT; stdcall;
  function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HRESULT; stdcall;
  function GetTypeInfoCount(out Count: Integer): HRESULT; virtual; stdcall;
  function Invoke(dispid: Integer; const IID: TGUID; LocaleID: Integer;
    Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HRESULT; stdcall;
public
  constructor Create(AOwner: TObject; ADispatch: IDispatch; const DisplntfIID: TGUID);
  destructor Destroy; override;
  property OnEvent: TNotifyEvent read FOnEvent write FOnEvent;
end;

```

여기서 TSampleEventSink 클래스는 TInterfacedObject 를 상속하되, 이벤트를 지원하기 위해서 IUnknown 과 IDispatch 인터페이스를 다시 구현하는 클래스이다. private 섹션에 선언된 FOwner 필드는 이벤트를 발생시킬 객체를 지정하게 되고, FDispatch 는 사용할 IDispatch 인터페이스를 그리고, FDisplntfIID 는 이벤트를 지원하는 Dispinterface 의 IID 를 저장한다. FConnection 필드는 이벤트와 연결을 하기 위해 호출하는 함수에서 필요로 하는 필드이다. 그리고, 실제 이벤트로 사용할 필드인 FOnEvent 를 선언하는데 이 필드는 TNotifyEvent 형으로 선언한다.

protected 섹션에는 IUnknown 과 IDispatch 인터페이스를 구현하기 위해 이들 인터페이스의 메소드 들을 선언한다. 그리고, public 섹션에 클래스의 constructor 인 Create 메소드를 새로 정의하고 Destroy 메소드는 오버라이드한다. 마지막으로 OnEvent 라는 이벤트를 프로퍼티로 정의한다.

그러면, 이 클래스를 구현해 보도록 하자. 먼저, constructor 와 destructor 를 다음과 같이 구현한다.

```

constructor TSampleEventSink.Create(AOwner: TObject; ADispatch: IDispatch;
  const DisplntfIID: TGUID);

```



```

begin
    inherited Create;
    FOwner := AOwner;
    FDisplntfIID := DisplntfIID;
    FDispatch := ADispatch;
    InterfaceConnect(FDispatch, FDisplntfIID, Self, FConnection);
end;

```

```

destructor TSampleEventSink.Destroy;
begin
    InterfaceDisconnect(FDispatch, FDisplntfIID, FConnection);
    inherited Destroy;
end;

```

즉, InterfaceConnect 와 InterfaceDisconnect 함수를 호출하기 위해서 필드에 constructor 에 넘어온 파라미터를 저장하고, 이를 이용하여 이벤트와 연결을 한다.

InterfaceConnect 프로시저는 IConnectionPoint 인터페이스를 이용하여 COM 서버에서 이벤트를 지원할 수 있도록 한다. 이 프로시저는 다음과 같이 선언되어 있다. 파라미터로 이벤트의 Source 로 사용될 인터페이스를 처음에, 이벤트 dispInterface 의 IID 를 두번째, 이벤트 Sink 로 사용할 인터페이스를 세번째 파라미터로 사용하며, 마지막에는 연결된 인터페이스의 핸들에 해당되는 값을 넘겨 받게 된다.

```

procedure InterfaceConnect(const Source: IUnknown; const IID: TIID;
    const Sink: IUnknown; var Connection: Longint);

```

마찬가지로, InterfaceDisconnect 프로시저는 지정된 인터페이스의 이벤트 연결을 해제하게 된다.

이렇게 이벤트를 지원하는 인터페이스가 있으면, IUnknown 인터페이스의 QueryInterface 메소드를 다시 구현해서 이벤트 인터페이스에 접근할 수 있도록 해야 한다. 그러므로, QueryInterface 를 다음과 같이 구현한다.

```

function TSampleEventSink.QueryInterface(const IID: TGUID; out Obj): HRESULT;
begin
    Result := E_NOINTERFACE;
    if GetInterface(IID, Obj) then
        Result := S_OK;

```

```

    if IsEqualGUID(IID,FDIsplntfIID) and GetInterface(IDispatch, Obj) then
        Result := S_OK;
    end;

```

즉, Obj 로 넘어온 파라미터만 검사하는 것이 아니라 FDIsplntfIID 필드에 저장된 이벤트 인터페이스의 IID 와도 동일한지 검사하여 이를 모두 허용하도록 하는 것이다.

IUnknown 인터페이스의 다른 메소드인 \_AddRef, \_Release 는 다음과 같이 구현한다. 이 내용은 인터페이스가 자동으로 해제되지 않도록 참조 계수를 지정하는 것이다.

```

function TSampleEventSink._AddRef: Integer;
begin
    Result := 2;
end;

```

```

function TSampleEventSink._Release: Integer;
begin
    Result := 1;
end;

```

이제는 IDispatch 인터페이스의 GetTypeInfo, GetTypeInfoCount, GetIDsOfNames, Invoke 메소드를 구현할 차례인데 이들 중 Invoke 를 제외하고는 그다지 중요하지 않으므로 다음과 같이 간단하게 구현한다.

```

function TSampleEventSink.GetTypeInfoCount(out Count: Integer): HRESULT;
begin
    Count := 0;
    Result := S_OK;
end;

```

```

function TSampleEventSink.GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HRESULT;
begin
    Result := E_NOTIMPL;
end;

```

```

function TSampleEventSink.GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT;

```

```
begin
    Result := E_NOTIMPL;
end;
```

Invoke 메소드에서는 이벤트를 지원하는 인터페이스에서 DispID 파라미터를 이용하여 여러 개의 이벤트 메소드를 매핑하는 역할을 하는데, 이 예제에서는 DispID 가 1 인 메소드 하나만 존재하므로 다음과 같이 간단하게 구현이 가능하다.

```
function TSampleEventSink.Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
    Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HRESULT;
begin
    case DispID of
        1: if Assigned(FOnEvent) then FOnEvent(FOwner);
    end;
    Result := S_OK;
end;
```

이것으로 이벤트 싱크를 지원하는 클래스는 모두 구현하였다. 이번에는 우리가 작성한 자동화 객체 서버와 그 이벤트를 사용할 수 있도록 wrapping 한 클래스를 선언하고 구현할 차례이다. Wrapper 클래스를 다음과 같이 선언한다.

```
TSampleEventObject = class
private
    FSampleEvent: ISampleEvent;
    FEventSink : TSampleEventSink;
    function GetOnEvent: TNotifyEvent;
    procedure SetOnEvent(Value: TNotifyEvent);
public
    constructor Create;
    destructor Destroy; override;
    property SampleEvent: ISampleEvent read FSampleEvent;
    property OnEvent: TNotifyEvent read GetOnEvent write SetOnEvent;
end;
```

private 섹션에 사용할 인터페이스인 ISampleEvent 를 담은 필드 변수인 FSampleEvent 와 이벤트 싱크 객체를 담은 필드 변수인 FSampleEventSink, 그리고 이벤트를 실제로 구현할

접근 메소드(access method)인 GetOnEvent, SetOnEvent 메소드를 선언한다. 그리고, public 섹션에 constructor 와 destructor, 그리고 자동화 객체 서버의 인터페이스를 프로퍼티와 이벤트를 프로퍼티로 제공한다.

이 클래스의 구현은 간단하다. 먼저 constructor 를 다음과 같이 구현한다.

```
constructor TSampleEventObject.Create:
begin
    FSampleEvent := CoSampleEvent.Create;
    FEventSink := TSampleEventSink.Create(Self, FSampleEvent,
        DIID_ISampleEventEvents);
end;
```

즉, ISampleEvent 를 프로퍼티로 접근하여 사용할 수 있도록 필드 변수에 CoClass 를 생성하여 대입하고, 이벤트를 사용할 수 있도록 앞서 작성한 이벤트 싱크 클래스를 인스턴스화하면 된다. 이때 이벤트를 발생시키는 객체를 첫번째 파라미터로 사용하게 된다. 주의할 것은 세번째 파라미터인 DIID\_ISampleEventEvents 인데 이 값을 이용하여 이벤트를 지원하는 인터페이스와 연결하게 된다. 이 값은 서버의 타입 라이브러리의 파스칼 버전인 ExamSvr2\_TLB.pas 유닛에서 인터페이스의 이벤트를 지원하기 위해 선언된 Dispinterface 의 IID 상수를 사용해야 한다.

destructor 와 GetOnEvent, SetOnEvent 메소드는 다음과 같이 간단히 구현할 수 있다.

```
destructor TSampleEventObject.Destroy:
begin
    FEventSink := nil;
    inherited Destroy;
end;

function TSampleEventObject.GetOnEvent: TNotifyEvent;
begin
    Result := FEventSink.OnEvent;
end;

procedure TSampleEventObject.SetOnEvent(Value: TNotifyEvent);
begin
    FEventSink.OnEvent := Value;
end;
```

이것으로 wrapper 클래스의 구현이 끝났다. 이제 이를 이용하여 실제로 이벤트가 동작하는지 알아보도록 하자. 우리가 앞에서 작성한 자동화 서버는 서버의 인터페이스인 ISampleEvent 의 SampleMethod1 메소드를 호출하면 이벤트가 발생하도록 구현하였다. 이를 위해 이벤트인 이벤트 싱크 클래스를 구현하여 델파이의 이벤트 구조와 호환되도록 하였으므로 다음과 같이 TNotifyEvent 형의 메소드를 하나 구현하고, 이 값을 이벤트에 대입하여 이벤트가 발생시 이 메소드가 실행되도록 하면 된다.

먼저, wrapper 클래스를 담을 수 있는 전역 변수를 다음과 같이 선언한다.

```
var
    Form1: TForm1;
    SampleEventObject: TSampleEventObject;
```

그리고, 폼의 private 섹션에 OnEvent 이벤트 핸들러로 사용할 수 있는 메소드를 다음과 같이 추가하고 구현한다.

```
private
    procedure Event(Sender: TObject);
```

... (중략)

```
procedure TForm1.Event(Sender: TObject);
begin
    ShowMessage('Event Fired !');
end;
```

Button1 을 클릭하면 이벤트 싱크 객체를 생성하고, 이벤트 핸들러로 앞서 선언하고 구현한 Event 메소드를 사용하도록 대입하도록 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if not Assigned(SampleEventObject) then
    begin
        SampleEventObject := TSampleEventObject.Create;
        SampleEventObject.OnEvent := Event;
    end;
```

end;

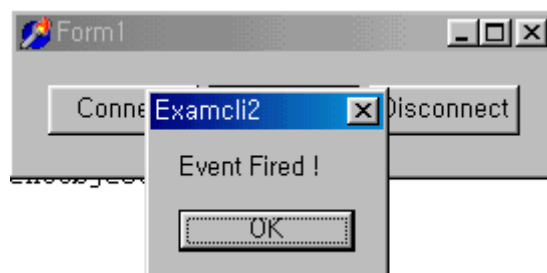
그리고, Button3 를 클릭하면 생성된 wrapper 객체가 있으면 이를 해제하도록 다음과 같이 OnClick 이벤트 핸들러를 작성한다.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    if Assigned(SampleEventObject) then
    begin
        SampleEventObject.Free;
        SampleEventObject := nil;
    end;
end;
```

마지막으로 이벤트를 발생시키는 Button2 의 OnClick 이벤트 핸들러는 다음과 같이 작성한다. 단순히 ISampleEvent 인터페이스의 SampleEvent1 메소드를 호출하는 것으로 이벤트가 발생할 것이다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    if Assigned(SampleEventObject) then
    begin
        SampleEventObject.SampleEvent.SampleEvent1;
    end;
end;
```

프로젝트를 컴파일하고 실행한 뒤에, ‘Connect’ 버튼과 ‘Event’ 버튼을 차례로 클릭하면 다음과 같이 이벤트가 발생하여 이벤트 핸들러가 실행되는 화면을 볼 수 있을 것이다.



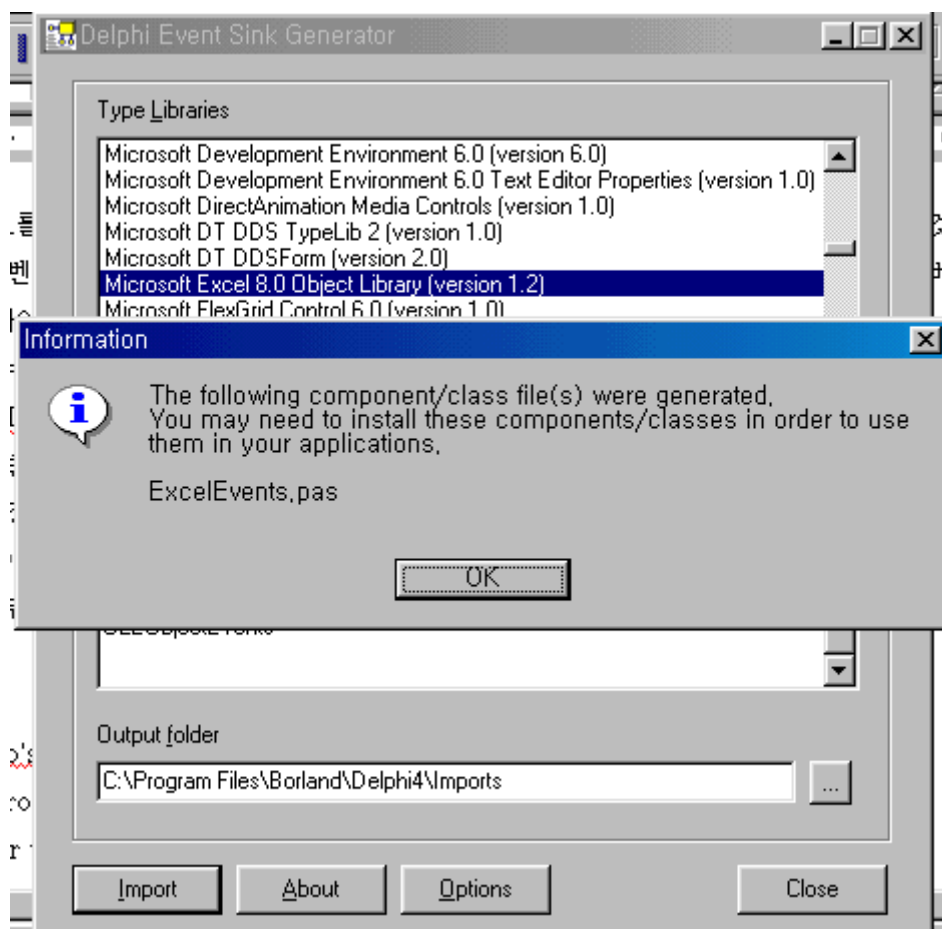
## 이벤트 지원 컴포넌트의 활용

간단한 이벤트를 지원하도록 했지만, 생각보다 작업이 만만치 않다는 것을 알 수 있을 것이다. 특히, 이벤트를 지원하도록 서버를 작성하는 것도 문제지만 이벤트를 지원하는 서버를 사용하는 클라이언트의 제작이 대단히 까다로운 것을 알 수 있다. 그렇다면, 이를 보다 편하게 해결할 수 있는 방법은 없을까?

다행히 Binh Ly([bly@castle.net](mailto:bly@castle.net))가 공개한 컴포넌트와 유틸리티를 이용하면 까다로운 자동화 객체의 이벤트를 쉽게 이용할 수 있다. 이 컴포넌트와 유틸리티의 소스와 데모 어플리케이션이 이 장에 해당되는 디렉토리의 EventSink 서브 디렉토리에 제공되므로 이를 참고하기 바란다.

이해를 돕기 위해 사용 방법을 간단히 소개하면 다음과 같다.

먼저 EventSinkImp 유틸리티를 실행하면, 컴퓨터에 설치된 자동화 서버의 타입 라이브러리에 대한 정보가 나열될 것이다. 이 중에서 사용할 타입 라이브러리를 선택하고 Import 버튼을 클릭하거나 더블 클릭하면 이벤트를 쉽게 사용할 수 있는 파스칼 유닛 파일이 다음과 같이 자동으로 생성된다.



이렇게 생성된 소스 코드는 Output folder 로 지정된 디렉토리에 생성되는데, 이 소스 코드는 컴포넌트 소스 코드이므로 Component|Install Component 메뉴를 이용하여 설치가 가능하다.

컴포넌트를 설치한 뒤에는 이 컴포넌트를 폼에 올려 놓고, 필요한 싱크 메소드를 후킹하여 사용하면 된다.

이렇게 설치된 컴포넌트에는 공통적인 서버 객체에 이벤트 싱크를 연결하고 해제하는 Connect, Disconnect 메소드가 제공되는데 이들은 다음과 같이 early 바인딩과 late 바인딩을 모두 사용하여 이용할 수 있다.

Early 바인딩의 경우에는 다음과 같이 한다.

```
var
  pObject1: IObject1;
begin
  pObject1 := CoObject1.Create;
  SinkComponent.Connect (pObject1 as IUnknown);
end;
```

그리고, late 바인딩의 경우 사용하는 방법은 다음과 같다.

```
var
  vObject1 : OleVariant;
begin
  vObject1 := CreateOleObject ('Server.Object1');
  SinkComponent.Connect (IUnknown (vObject1));
end;
```

이렇게 연결한 서버 객체와의 연결을 해제하려면 Disconnect 메소드를 호출하면 되는데, 싱크 컴포넌트가 파괴되는 경우에는 자동으로 호출되므로 따로 호출할 필요는 없다.

디렉토리에 포함된 데모는 IE 4.0 의 이벤트를 활용하는 예제이다. 이를 분석하면 구체적인 사용방법을 익힐 수 있을 것이다.

## 정 리 (Summary)

이번 장에서는 컬렉션과 이벤트를 구현하는 고급스러운 COM 기술의 구현 방법에 대해서 알아보았다. 간단한 자동화 객체를 만들기 위해서는 이런 어려운 내용을 특별히 익힐 필요가 없겠지만, 실제로 쓸모가 있는 제대로된 객체를 만들어 사용하려면 컬렉션과 이벤트의



구현은 필수적이다.

아무쪼록, 이 장에서 설명한 내용을 바탕으로 국내에서도 훌륭한 자동화 서버 객체나 COM 객체를 작성하여 강력한 어플리케이션을 개발하고, 가능하면 많은 개발자 들이 공유할 수 있도록 범용성을 갖춘 멋진 객체를 개발하고, 이를 공개하여 여러 개발자들에게 도움을 줄 수 있는 개발자들이 많았으면 하는 바람이다.

## 고급 COM 기술의 활용 (II)

### (Using Advanced COM Techniques II.)

이번 장에서는 액티브 X 를 이용하여 여러가지 객체나 어플리케이션을 개발할 때 부딪힐 수 있는 문제점들과 이들에 대한 해결책을 제시할 것이다. 그렇게 방대한 내용은 아니지만, 소프트웨어를 개발하다 보면 실제로 아주 간단한 문제로 골머리를 싸맬 때가 매우 많다. 이번 장에서는 이런 문제들에 대해서 알아본다. 여기에서 소개하는 많은 팁들은 Inprise 의 뉴스 그룹에서 참고하였음을 미리 밝혀둔다.

참고로 텔파이로 액티브 X 기술을 구현할 때의 여러 가지 팁과 FAQ 에 대한 정보는 Conrad Herrman 이 제공하는 홈 페이지에서 많이 얻을 수 있으므로 이를 참고하기 바란다. 이 홈페이지의 URL 은 Inprise 홈 페이지의 개발자 정보에서 찾을 수 있을 것이다.

#### Out of Process 논비주얼 COM 서버

여러가지 COM 서버를 작성하면서 지금까지 작성한 방법을 곰곰히 생각해보면, 결국에는 in-proc DLL 형태의 액티브 X 라이브러리로 만들거나 현재의 어플리케이션에 OLE 자동화 객체를 추가하는 형태로 만들었다. 이런 식으로 어플리케이션 프로젝트에 자동화 객체 위저드를 이용해서 추가한 경우에는 어떤 방식으로든 어플리케이션의 윈도우가 생성되고, 메시징 루프가 윈도우에 의해 사용된다.

예를 들어, out-of-process COM 서버를 만들면 언제나 좋은 싫든 폼이 하나 생성되어 버린다. 이럴 때에 물론 OnCreate 이벤트 핸들러에서 폼을 숨겨버리면 그만이지만, 불필요한 낭비임에는 틀림이 없다.

이런 경우에 윈도우가 생성되지 않고, in-proc 서버가 아닌 out-of-process 서버를 작성해서 사용하고 싶은 경우에는 어떻게 해야 할까 ?

이런 경우에는 Forms 유닛의 Application 객체를 변경해서 사용하되 비주얼 인터페이스를 사용하지 않게 하는 방법이 있다.

이를 위해서는 다음의 유닛을 사용하면 되는데, 이 유닛은 Inprise 의 뉴스 그룹에서 공개된 유닛인데, 작성자에 대한 정보를 기록해두지 않은 탓에 밝히지 못했음을 미리 말해둔다. 이 유닛으로 교체한 다음에 프로젝트 파일의 Application.Initialize 와 Application.Run 구문을 ServerApp.Initialize 와 ServerApp.Run 으로 대체하면 된다. 또한, 어플리케이션에서 Application.ProcessMessage 루틴이 사용되는 부분이 있다면 ServerApp 에 대한 내용으로 변경해야 하는 것은 물론이다.

```
unit ServApp;
```

interface

type

  TServerApp = class(TObject )

  protected

    procedure DoTerminate(var Shutdown: Boolean);

  public

    function ProcessMessages: Boolean;

    procedure Initialize;

    procedure Run;

  end;

var

  ServerApp: TServerApp;

implementation

uses

  ComServ;

function TServerApp.ProcessMessages: Boolean;

var

  Msg: TMsg;

begin

  Result := true;

  while PeekMessage(Msg, 0, 0, 0, PM\_REMOVE) do

    if (Msg.Message = WM\_QUIT) then

      Result := false

    else

      begin

        TranslateMessage(Msg);

        DispatchMessage(Msg);

      end;

end;

```

procedure TServerApp.DoTerminate(var Shutdown: Boolean);
begin
    if Shutdown and not CallTerminateProcs then
        Shutdown := False;
    end;
end;

```

CallTerminateProcs 루틴은 델파이의 COM 시스템을 종료하는 역할을 한다.

```

procedure TServerApp.Initialize;
begin
    if (ComServer.StartMode = smStandalone) then Halt;
    if (InitProc <> nil) then
        TProcedure(InitProc);
    ComServer.OnLastRelease := ServerApp.DoTerminate;
end;

```

InitProc 은 COM 서버 시스템을 초기화하기 위해서 필요하다. 즉, 이 부분이 핵심이라고 할 수 있는데, COM 서버가 stand-alone 으로 시작하지 못하도록 하고 마지막 OnLastRelease 이벤트에 DoTerminate 메소드를 실행하도록 대입하여 COM 서버가 종료할 때 실행되도록 한다.

```

procedure TServerApp.Run;
begin
    while ProcessMessages do;
end;

```

Run 메소드는 간단히 ProcessMessage 루틴을 반복하도록 하면 품이 필요 없이 out-of-process COM 서버를 쉽게 구현할 수 있다.

```

initialization
    ServerApp := TServerApp.Create;

finalization
    ServerApp.Free;

end.

```

구현한 내용을 보면 무척 간단하다는 것을 알 수 있을 것이다. 별거 아닌 것 같지만, 필요에 의해서 이런 유닛을 찾아 다녀 보면 막상 쓸만한 것이 없는 경우가 많고, 그렇다고 직접 작성하려하면 막막한 경우가 많다.

이 주제 역시 해결 방법을 보면 별 것 아니지만, 막상 만들어 보려고 하면 쉽지 않은 것이다. 의외로 많이 사용할 가능성이 있는 해결 방법이므로 꼭 기억했다가 사용하기 바란다.

## 가변형 변수와 스트림간 통신

액티브 X 기술을 이용하여 클라이언트/서버 어플리케이션을 제작하다 보면, 간혹 스트림을 통해 바이너리 파일을 전송할 필요가 있을 때가 있다. 이럴 때에는 DCOM 에서 지원하는 데이터 형에 TStream 클래스가 호환되지 않기 때문에 가변형 변수를 사용하여 전송을 해야 한다.

즉, 다시 말하면 서버 측에서는 파일을 메모리 스트림에 읽어오고, 이를 가변형 변수로 형 변환을 한 뒤에 클라이언트로 전송하고, 클라이언트에서는 이렇게 넘어온 가변형 변수를 메모리 스트림으로 변경하면 된다.

다음에 스트림과 가변형간의 형 변환을 담당하는 루틴을 소개한다.

```
function StreamToVariant(Stream:TMemoryStream): Variant;
```

```
var
```

```
    Data: Pointer;
```

```
begin
```

```
    Result := VarArrayCreate([0, Stream.Size - 1], varByte);
```

```
    Data := VarArrayLock(Result);
```

```
    try
```

```
        Move(Stream.Memory^, Data^, Stream.Size);
```

```
    finally
```

```
        VarArrayUnlock(Result);
```

```
    end;
```

```
end;
```

```
function VariantToStream(V: Variant): TMemoryStream;
```

```
var
```

```
    Data: Pointer;
```

```
begin
```

```
    Result := TMemoryStream.Create;
```

```
    Data := VarArrayLock(V);
```

```

try
    Result.WriteBuffer(Data^, VarArrayHighBound(V, 1) + 1);
finally
    VarArrayUnlock(V);
end;
Result.Seek(0, soFromBeginning);
end;

```

## IOleClientSite 인터페이스의 활용

IOleClientSite 인터페이스를 이용하면 컨테이너에 임베드된 객체의 정보를 얻을 수 있다. 이 인터페이스를 얻기 위한 함수를 다음과 같이 구현할 수 있다.

```

function ClientSite(obj: IUnknown): IOleClientSite;
var
    Site: IOleClientSite;
    OleObj: IOleObject;
begin
    if (obj.QueryInterface(IOleObject, OleObj) = S_OK) and
        (OleObj.GetClientSite(Site) = S_OK) then
        Result := Site
    else
        Result := nil;
    end;
end;

```

obj 파라미터는 액티브 X 컨트롤을 지정한다. 예를 들어, 다음과 같이 사용할 수 있다.

```

type
    TButtonX = class(TActiveXControl)
    ...
    end;

```

... (중략)

```

procedure TButtonX.Click;
var

```

```

Site: IOleClientSite;
begin
    Site := ClientSite(Self);
end;

```

액티브 X 컨트롤을 개발한 뒤에 이를 실제로 사용하게 되면, 실행되는 모드가 런타임인지 아니면 디자인 타임인지를 구별할 필요가 있을 때가 있다. 이를 파악하기 위해서는 컨트롤의 컨테이너의 UserMode 앰비언트 프로퍼티를 이용해야 한다.

이때에도 앞의 ClientSite 함수를 이용하면 컨테이너의 모드를 쉽게 알 수 있다. 다음 함수는 컨테이너가 디자인 모드이면 True 를 반환한다.

```

function IsControllnDesignMode(obj: IUnknown): Boolean;
var
    Mode: Boolean;
begin
    try
        Mode := not ((ClientSite(obj) as IAmbientDispatch).UserMode);
    except
        Mode := False;
    end;
    Result := Mode;
end;

```

## Safe for scripting/safe for initializing 의 지원

‘safe for scripting’과 ‘safe for initializing’이란 코드 사인을 통한 기본적인 보안과 함께 사용될 수 있는 Object Safety 라는 2 차 보안을 나타내는 용어이다.

컨트롤을 HTML 페이지에서 다운로드하면 페이지가 자바 스크립트나 VB 스크립트 등을 이용할 수도 있고, 컨트롤에 대한 프로퍼티 값들이 포함된다. 이때 제작한 컨트롤의 일부 메소드나 프로퍼티를 잘못 사용할 때 클라이언트 컴퓨터에 나쁜 영향을 미칠 수 있다면 이러한 메소드나 프로퍼티가 잘못 사용되지 않도록 해야 할 것이다.

컨트롤이 ‘safe for scripting’으로 표시된다는 의미는 객체가 OLE 자동화를 통해 안전하게 자동화될 수 있다는 의미이다. 즉, 객체를 자동화하는 각종 스크립트에 대해서 클라이언트 컴퓨터에 나쁜 영향을 주지 않는다는 의미이다. 그리고, ‘safe for initializing’으로 표시된다는 의미는 객체의 프로퍼티나 데이터가 어떤 지속성 저장소(persistent storage)에서도 저장되었다가 복원될 수 있다는 의미이다. 이들 자체에 대한 보다 자세한 사항은 마이크로소프트

트에서 제공하는 자료 들을 참고하기 바란다.

그렇다면 컨트롤에 ‘safe for scripting/initializing’을 표시하는 방법에 대해서 알아보도록 하자. 여기에는 크게 2 가지 방법이 존재하는데 첫번째 방법은 컨트롤이 언제나 스크립팅과 초기화에 안전하다고 표시하는 것이고, 다른 하나는 컨트롤이 safe mode 와 unsafe 모드를 전환할 수 있도록 표시하는 방법이다.

#### 1. 컨트롤에 대한 적절한 레지스트리 키를 설치한다.

컨트롤이 설치될 때 레지스트리를 변경하는 방법에 대해서는 클래스 팩토리를 오버라이드하여 구현한다는 것을 이미 앞에서 설명한 바 있다. 그러므로, 해당되는 액티브 X 컨트롤의 클래스 팩토리 클래스를 오버라이드하여 ‘safe for scripting/safe for initialization’을 표시하도록 구현하면 된다.

다음의 유닛에서 이들 클래스 팩토리를 상속하여 구현하였다. 이 유닛은 Conrad Herrman 이 DAX FAQ 를 통해서 공개한 유닛임을 미리 밝혀 두며, 소스 코드에 대한 설명은 이미 29 장에서 UpdateRegistry 메소드를 오버라이드하여 구현하는 방법에 대해 설명한 바 있기 때문에 생략하겠다.

```
unit SafeFactory;
```

```
interface
```

```
uses ComObj, ActiveX, AXCtrls;
```

```
const
```

```
    CATID_SafeForScripting: TGUID = '{7DD95801-9882-11CF-9FA9-00AA006C42C4}';
```

```
    CATID_SafeForInitializing: TGUID = '{7DD95802-9882-11CF-9FA9-00AA006C42C4}';
```

```
type
```

```
    TSafeActiveFormFactory = class(TActiveFormFactory)
```

```
        procedure UpdateRegistry(Register: Boolean); override;
```

```
    end;
```

```
    TSafeActiveXControlFactory = class(TActiveXControlFactory)
```

```
        procedure UpdateRegistry(Register: Boolean); override;
```

```
    end;
```

```
implementation
```



```

procedure AddSafetyKeys(const ClassID: TGUID);
var
    ClassKey: string;
begin
    ClassKey := 'CLSIDW' + GUIDToString(ClassID);
    CreateRegKey(ClassKey + 'WImplemented Categories', '', '');
    CreateRegKey(ClassKey + 'WImplemented CategoriesW' + GUIDToString(
        CATID_SafeForScripting), '', '');
    CreateRegKey(ClassKey + 'WImplemented CategoriesW' + GUIDToString(
        CATID_SafeForInitializing), '', '');
end;

```

```

procedure RemoveSafetyKeys(const ClassID: TGUID);
var
    ClassKey: string;
begin
    ClassKey := 'CLSIDW' + GUIDToString(ClassID);
    DeleteRegKey(ClassKey + 'WImplemented CategoriesW' + GUIDToString(
        CATID_SafeForInitializing));
    DeleteRegKey(ClassKey + 'WImplemented CategoriesW' + GUIDToString(
        CATID_SafeForScripting));
    DeleteRegKey(ClassKey + 'WImplemented Categories');
end;

```

```

{TSafeActiveFormFactory}
procedure TSafeActiveFormFactory.UpdateRegistry(Register: Boolean);
begin
    if Register then
    begin
        AddSafetyKeys(ClassID);
        inherited UpdateRegistry(Register);
    end
    else
    begin
        RemoveSafetyKeys(ClassID);
        inherited UpdateRegistry(Register);
    end
end;

```

```

    end;
end;

{TSafeActiveXControlFactory}
procedure TSafeActiveXControlFactory.UpdateRegistry(Register: Boolean);
begin
    if Register then
    begin
        AddSafetyKeys(ClassID);
        inherited UpdateRegistry(Register);
    end
    else
    begin
        RemoveSafetyKeys(ClassID);
        inherited UpdateRegistry(Register);
    end;
end;
end.

end.

```

이 클래스 팩토리를 이용하기 위해서는 액티브 X 라이브러리의 유닛의 uses 절에 SafeFactory.pas 유닛을 추가하고, initialization 섹션을 다음과 같이 클래스의 이름만 변경 해주면 된다.

```

initialization
    TSafeActiveXControlFactory.Create(ComServer, TSampleActiveX, TSampleControl,
        Class_SampleActiveX, 1, '', 0, tmApartment);
end.

```

## 2. 컨트롤에서 IObjectSafety 인터페이스를 구현하는 방법

IObjectSafety 인터페이스를 구현하면 컨트롤이 safety 를 요구하지 않는 컨테이너에서 동작할 때에는 unsafe 메소드나 프로퍼티를 사용할 수 있게 할 수 있다. 그렇지만, IE 와 같이 보안을 요구하는 컨테이너에서는 IObjectSafety 인터페이스를 이용하여 컨트롤이 safe mode 로 전환하도록 요구하고, 모든 unsafe 메소드와 프로퍼티를 사용 불가능하게 만들 수 있다.

## 델파이 폼을 액티브 폼으로 전환하기

델파이 폼을 액티브 폼으로 변경하는 요령에 대해서 설명하고자 한다. 많은 경우에 있어서 처음부터 액티브 폼으로 개발하는 경우도 있겠지만, 이미 완성된 어플리케이션을 액티브 폼으로 변경하는 작업을 하고자 하는 경우도 많을 것이다. 이럴 때에는 다음과 같은 방법을 이용하여 액티브 폼으로 변경할 수 있다.

- 폼에 있는 모든 컴포넌트를 선택하고 이를 복사했다가 액티브 폼에 붙여넣는 방법

이 방법은 컴포넌트를 폼에 옮기는데에는 큰 문제가 없지만, 컨트롤과 연결된 코드는 복사되지 않기 때문에 이들을 적절하게 설정하는 것이 가장 중요한 작업이 된다.

- 컴포넌트 템플릿을 활용하는 방법

폼에 있는 모든 컴포넌트를 메뉴를 선택하고 Component|Create Component Template 메뉴를 선택하여 템플릿을 생성하면 VCL 컴포넌트 팔레트에 Template 탭이 생성되면서 컴포넌트로 등록될 것이다. 액티브 폼을 열고 이 컴포넌트를 폼에 떨어뜨리면 된다.

이 방법의 장점은 컨트롤과 함께 연결된 코드가 같이 복사된다는 점이다.

- 직접 TForm 클래스를 액티브 폼으로 변경시키는 방법

앞서 설명한 방법과 같이 컨트롤을 복사하지 않고, 코드를 수정함으로써 액티브 폼으로 변경하는 방법이다. 이 방법의 장점은 표준 델파이 어플리케이션으로 일단 개발과 테스트를 마치고 간단히 액티브 폼 버전으로 변경하여 사용할 수 있다는 점이다.

변경 원칙은 다음과 같다. 이 방법은 Conrad Herrman 이 공개한 것으로 꽤 유용하게 적용할 수 있다.

1. 표준 폼 어플리케이션을 디렉토리에 저장한다.
2. 액티브 폼을 저장할 서브 디렉토리를 생성하고, 새로운 액티브 폼 프로젝트를 생성하여 저장한다.
3. 액티브 폼 프로젝트에서 프로젝트 관리자를 선택하고 표준 폼 어플리케이션 프로젝트가 저장된 디렉토리에서 모든 폼과 데이터 모듈을 프로젝트에 추가하도록 한다.
4. 액티브 폼의 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TActiveForm1.FormCreate(Sender: TObject);
```

begin

//이 코드는 정상적인 델파이 TForm 인 자식 폼을 생성한다.

ChildForm := TForm1.Create(Self);

ChildForm.Parent := Self;

ChildForm.Align := alClient;

ChildForm.BorderStyle := bsNone;

ChildForm.Visible := True;

end;

5. uses 절에서 폼의 유닛 파일을 추가하고, 자식 폼을 public 섹션에 추가한다.

type

TActiveForm1 = class ...

....

public

ChildForm: TForm1;

....

end;

폼을 컴파일하고 테스트한다.

## 액티브 폼과 IE 4.0

델파이의 액티브 폼은 일종의 액티브 X 컨트롤이지만, 의외로 IE 4.0 과 충돌하는 경우가 많다. 이 문제는 델파이의 액티브 폼 위저드에도 약간의 문제가 있지만 IE 4.0 자체가 표준과는 벗어난 여러가지 문제점을 많이 안고 있기 때문이기도 하다. 그러므로, 이 문제를 완전히 해결할 수는 없지만 몇 가지 고려할 점들에 대해서 논의하고자 한다.

먼저 델파이 3 에서 문제가 되었던 것 중에서 델파이 4 에서 해결된 것들에 대해서 알아보고, 계속해서 고려해야 할 점을 알아보도록 하자.

### 1. 쓰레딩 모델

델파이 3 는 기본적으로 단일 쓰레드 모델 만을 지원했기 때문에, IE 4.0 의 여러 인스턴스를 메모리에 띄울 때 충돌이 일어나는 경우이다. 이 문제는 델파이 4 에서 쓰레딩 모델로 apartment 쓰레딩 모델을 선택할 수 있게 되면서 많이 해결되었다. 보통 디폴트로 이 모델을 사용하게 되지만, IE 4.0 과의 충돌이 있다면 먼저 쓰레딩 모델을 검토하기 바란다.

## 2. 액티브 폼 내부 컨트롤의 포커스 설정 문제

여러 개의 컨트롤을 가지고 있는 액티브 폼에서 하나의 컨트롤을 클릭하여 포커스를 주고, 다른 어플리케이션을 활성화 시켰다가 다시 IE 4.0 을 활성화하면 이전에 포커스를 가졌던 컨트롤이 포커스를 잃게 된다.

이 문제는 액티브 폼 자체가 UI-active 컨트롤이기 때문이다. 그러므로, 프레임이 활성화 되면 컨트롤이 IInPlaceActiveObject.OnFrameWindowActivate 메소드가 호출되는데 델파이 3 의 위저드에서는 이 메소드가 단지 InPlaceActivate(True) 메소드만을 호출하기 때문에, 컨트롤이 이미 UI-active 한 경우에는 아무런 영향을 미치지 못한다. 그러므로 앞서 설명한 문제를 해결하기 위해서는 폼의 활성화된 자식 컨트롤에 포커스를 설정해야 하는데 델파이 4 에서 이 점이 수정되었다.

## 3. 탭/백스페이스 키가 제대로 동작하지 않을 때

IE 4.0 에서 액티브 폼을 띄운 뒤에 탭/백스페이스 키를 이용할 때 제대로 동작하지 않는 문제가 발생하는 경우가 있다. 이 문제를 해결하기 위해서는 액티브 폼의 타입 라이브러리를 동작시키고 TabsOn 이라는 새로운 메소드를 인터페이스에 추가한다. 이 메소드의 선언은 'procedure TabsOn' 으로 특별한 파라미터나 반환값은 지정할 필요가 없다. 그리고, 이 메소드에 대한 ID 를 부여한다. 그리고 나서 액티브 폼을 구현한 유닛의 protected 섹션에 선언되어 있는 TabsOn 선언부를 다음과 같이 public 섹션으로 옮긴다.

```
public
    procedure TabsOn; safecall;
```

그리고, TabsOn 프로시저의 구현 부분을 다음과 같이 수정한다.

```
procedure TActiveFormX.TabsOn;
begin
    (ComObject As IOleInPlaceActiveObject).OnFrameWindowActivate(True);
end;
```

그리고, 다소 귀찮기는 하지만 폼에 있는 모든 컨트롤의 OnClick 이벤트에 TabsOn 을 호출하도록 추가하여 액티브 폼을 클릭할 때마다 UI 를 활성화 시키도록 한다.

다른 해결 방법으로는 Conrad Herrman 이 제시한 방법이 있는데, 그의 해결책은 폼에 WM\_MOUSEACTIVATE 메시지에 대한 핸들러를 설치해서, 이 메시지가 발생할 때마다 UI

를 활성화하는 것이다. 이렇게 하면, 핸들러가 설치된 자식 컨트롤 들이 클릭될 때마다 활성화되어 제대로 포커스를 가지게 된다.

메시지 핸들러는 다음과 같이 구현한다. 먼저 다음과 같이 핸들러를 선언하고 이를 구현하면 된다.

```
procedure WMMouseActivate(var msg: TWMMouseActivate); message WM_MOUSEACTIVATE;
```

... (중략)

```
procedure TActiveFormX.WMMouseActivate(var msg: TWMMouseActivate);
```

```
begin
```

```
    inherited;
```

```
    if (msg.Result = MA_ACTIVATE) or (msg.Result = MA_ACTIVATEANDEAT) then
```

```
    begin
```

```
        DoUIActivate;
```

```
    end;
```

```
end;
```

```
procedure TActiveFormX.DoUIActivate;
```

```
begin
```

```
    if (ComObject <> nil) then
```

```
    begin
```

```
        (ComObject as IOleObject).DoVerb( OLEIVERB_UIACTIVATE,
```

```
        nil, nil, 0, 0, PRect(nil)^);
```

```
    end;
```

```
end;
```

이 방법은 대부분의 경우에 적용되지만 RichEdit 컨트롤에서는 사용할 수 없다. 이 경우에는 OnClick 메소드를 이용한 방법을 사용해야 한다. 액티브 폼에서 탭 키를 누르면 실제로 컨트롤에서 이동은 일어나지만, 이를 실제로는 제대로 보여주지 못하기 때문이다. 또한, UI가 활성화되어도 자신 컨트롤에 포커스를 제대로 설정하지 못하기 때문에 이런 현상이 나타나는 것이므로 앞서와 같은 방법으로 이 문제를 해결해야 한다.

## 액티브 X 서버 배포에 관한 문제

몇 가지 인터페이스를 구현한 액티브 X 서버를 배포할 때에는 배포할 때 염두에 두어야 할

몇 가지 사항이 있다. IStrings, IProvider, IDataBroker 인터페이스를 사용하거나 델파이 폰트, 색상, 문자열, 그림 프로퍼티 페이지를 사용한 경우에는 볼랜드의 표준 VCL 타입 라이브러리를 같이 배포해야 한다.

이 라이브러리는 STDVCL32.DLL 라이브러리와 STDVCL32.TLB 라고 하는 타입 라이브러리로 존재한다. 이들은 모두 윈도우의 시스템 디렉토리에 위치하며, 모두 시스템 레지스트리에 등록되어야 한다.

## 액티브 X 컨트롤의 코드 다운로드 문제

29 장에서도 설명했듯이 액티브 X 컨트롤을 배포할 때 Web Deployment options 대화 상자에서 적절한 정보를 설정하고, Web Deploy 명령을 선택하면 되는데, 막상 홈 페이지에 올려 놓고 이 홈 페이지를 클라이언트에 가서 브라우저를 띄우고 접근하면 커다란 붉은 X 자만 볼 수 있는 경우가 많다.

제작한 액티브 X 컨트롤을 웹 브라우저에 띄우기 위해서 Web Deploy 명령을 선택하면 HTML 문장에 <OBJECT> 태그가 추가되면서 다음과 같이 액티브 X 컨트롤에 대한 정보가 추가된다.

<OBJECT

```
classid="clsid:29D37F03-F02F-11D0-ACB2-0080C7316F20"
codebase="http://www.SampleSite.com/MyControl.ocx#version=1,0,1,0"
width=350
height=250
align=center
hspace=0
vspace=0
```

>

</OBJECT>

이 태그에서 #version 부분은 옵션인데, 이 내용이 빠지면 버전에 상관이 없다는 의미이다. 실제 액티브 X 컨트롤을 지칭하는 것이 ClassID 이다. 그러므로, IE 가 지정된 ClassID 의 컨트롤이 클라이언트에 설치되어 있으면, 지정된 버전과 일치하는 것인지 알아보고 로컬 버전이 일치하면 컨트롤을 생성하기 위해 로컬 복사본을 이용하게 된다.

만약 HTML 페이지에서 요구하는 버전이 현재 설치된 것보다 새 버전이거나 ClassID 에 지정된 컨트롤이 아직 설치되지 않은 경우에는 IE 가 컨트롤을 생성하기 전에 컨트롤을 다운로드하고 이를 설치하게 된다.

이때 코드 베이스는 로컬 기계의 Windows/OCCache 디렉토리나 서버 디렉토리에 복사하

는데 이때 URI 가 HTTP(디폴트)를 이용하거나, FTP, FILE URI 등을 이용할 수 있다. 여기서 원격 서버를 찾을 수 없거나, 로컬 디스크의 디스크 공간이 부족한 경우, 그리고 codebase 의 이름이 잘못 설정된 경우(서버가 대소문자를 가리는 경우에 혼함)에 이 과정이 실패할 수 있다. .

액티브 X 컨트롤을 다운로드하기 전에 IE 가 코드 signature 를 확인하게 된다. 여기에서 꽤 많은 브라우저의 설정이 잘못된 경우에 액티브 X 컨트롤을 볼 수 없다. 보안 레벨을 High 로 설정한 경우에는 IE 가 컨트롤이 코드 사인되지 않았으면 설치를 하지 않는다. 보안 레벨이 Medium 으로 설정되면 설치할 것인지 물어보는 대화상자를 보여 주고, 컨트롤을 다운로드하게 된다. 보안 레벨이 Low 로 설정되면 코드 사인되지 않은 컨트롤도 바로 설치된다.

이런 과정을 거쳐 다운로드된 codebase 는 Windows/OCCache 디렉토리에 설치된다. 설치되는 codebase 는 Object 태그의 CODEBASE 태그에 의해 그 종류가 결정된다. 만약 파일이 DLL 인 경우(보통 확장자 .OCX) 파일 하나를 가리키며, 파일이 .CAB 인 경우에는 여러 파일이 포함되어 있다. 그리고, codebase 가 .INF 파일인 경우에는 codebase 를 구성하는 파일의 리스트를 설명한다.

DLL 인 경우에는 설치 과정에서 DLL 파일을 로드한 뒤에 DllRegisterServer 함수를 호출하여 컨트롤을 생성하게 된다. 이 과정에서 DLL 파일이 LoadLibrary 함수에 의해 로드되지 않을 수가 있는데 이것은 DLL 이 라이브러리 DLL 이나 델파이 패키지, 시스템 DLL 과 같은 다른 파일에 의존적인데 이 파일들에 접근할 수 없는 경우이다. DLL 이 로드되지만 초기화에 실패하는 경우도 있는데, 이것은 환경 설정에도 문제가 있을 수 있지만 유닛의 초기화 섹션에서 예외가 발생하기 때문일 수도 있다.

또다른 문제로는 컨트롤이 제대로 등록되지 않는 경우에 발생할 수 있다. DLL 이 등록되지 않는 원인으로 가장 흔한 것은 NT 클라이언트를 사용할 경우에 사용자 권한이 레지스트리를 변경할 수 있는 권한이 없는 경우이다.

컨트롤이 이런 과정을 거쳐서 설치되고 생성되면 IE 는 컨트롤의 인스턴스를 생성하려고 시도한다. 이 과정에서 문제가 발생하는 이유로는 HTML 파일에서 지정된 ClassID 가 잘못 지정된 경우와 메모리나 리소스가 부족한 것이 원인일 수 있다.

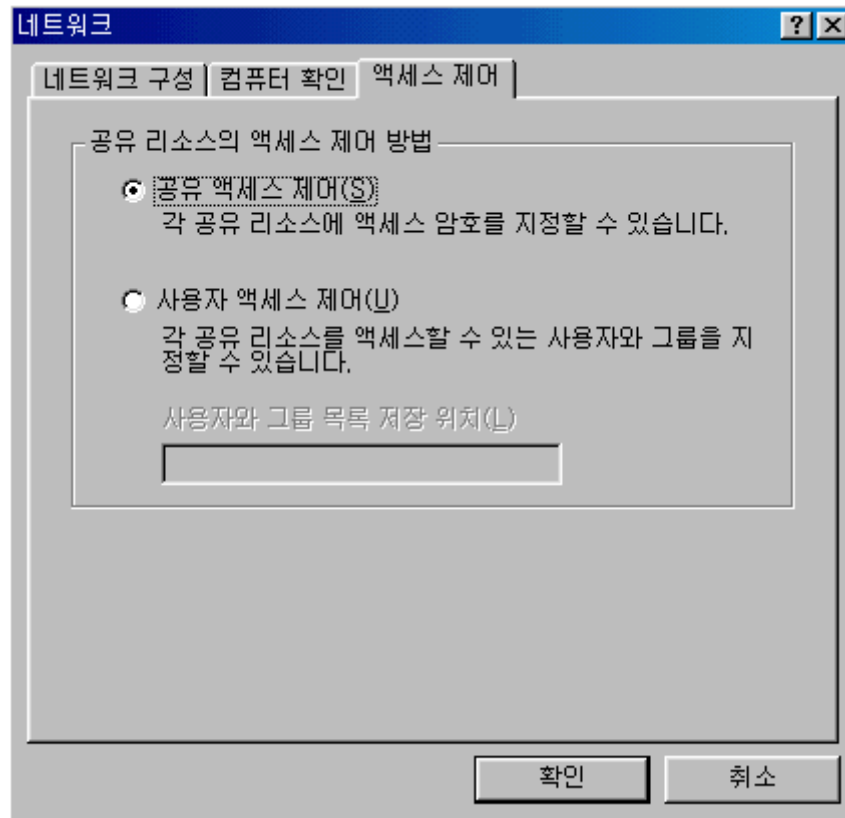
## DCOM 환경 설정

COM 객체와 액티브 X 컨트롤과 액티브 폼 객체를 훌륭하게 생성하고, 이를 이용하여 인터넷이나 인터넷을 통해서 사용하려고 할 때 문제가 되는 이유 중에서 가장 커다란 비중을 차지하는 것은 뜻 밖에도 DCOM 의 환경 설정을 제대로 하지 못했기 때문인 경우가 많다. 그러면, DCOM 클라이언트/서버의 환경 설정에 대해서 중요한 사항을 알아보도록 하자.

### ● 서버의 환경 설정



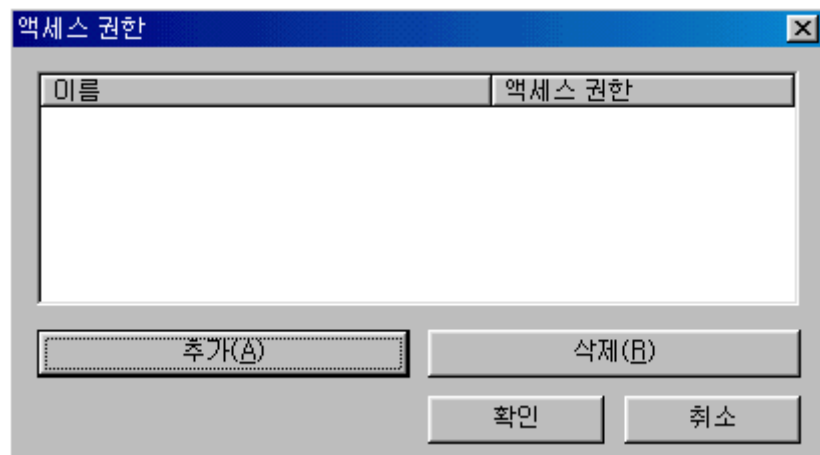
DCOM 의 서버는 사용자 레벨(user level)과 공유 레벨(share level) 접근이 가능하도록 환경설정을 할 수 있다. 이 설정을 변경하기 위해서는 제어판의 네트워크 애플릿을 실행하고, 다음과 같이 액세스 제어 페이지에서 설정을 변경할 수 있다.



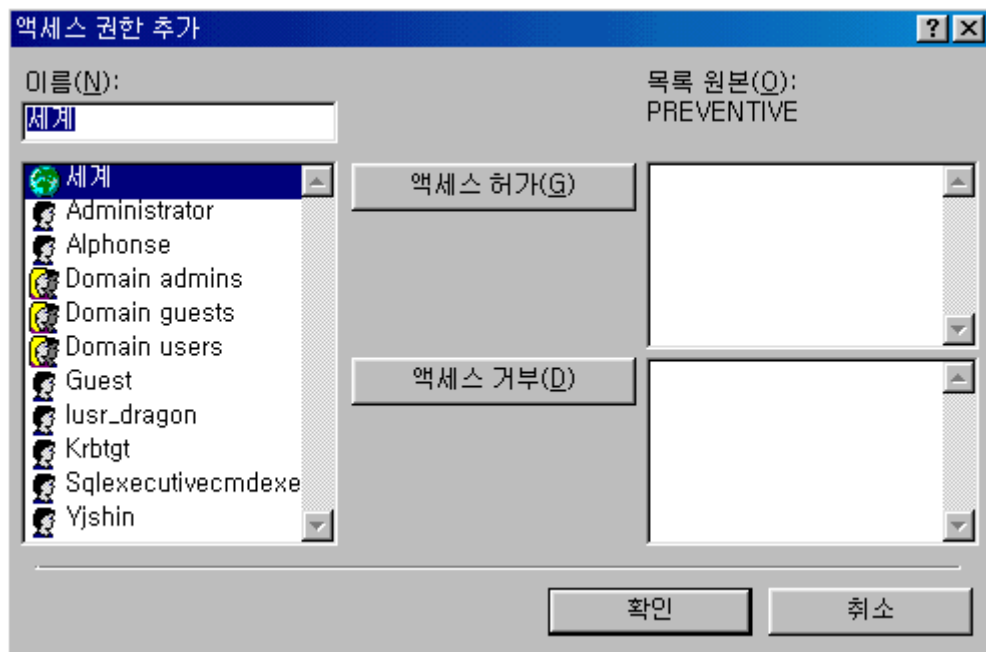
만약 윈도우 NT 서버를 이용한 사용자 인증이 가능하다면, 사용자 액세스 제어를 선택하고, 적절한 NT 서버를 지정하여 사용자와 그룹을 이용하도록 하면 된다. NT 서버와 관계 없이 DCOM 을 이용하기 위해서는 공유 액세스 제어를 선택하도록 한다.

윈도우 95 에서 DCOM 을 사용하기 위해서는 반드시 DCOM for Win95 를 설치해야 하는데 이 설치 파일은 <http://www.microsoft.com/com/dcom95/download.htm> 에서 구할 수 있다. 그렇지만 필자의 생각으로 DCOM 을 사용하기 위해서는 윈도우 NT 4.0 SP 3, 윈도우 98 이상에서 사용하기를 권하고 싶다.

사용자 액세스 제어를 선택한 경우에는 DcomCnfg.exe 유틸리티를 실행하고, 이 유틸리티의 기본 보안(Default Security) 페이지를 선택하고, 기본값 편집(Edit Default) 버튼을 클릭하면 다음과 같이 액세스 권한(Access Permissions) 대화 상자가 나타날 것이다.



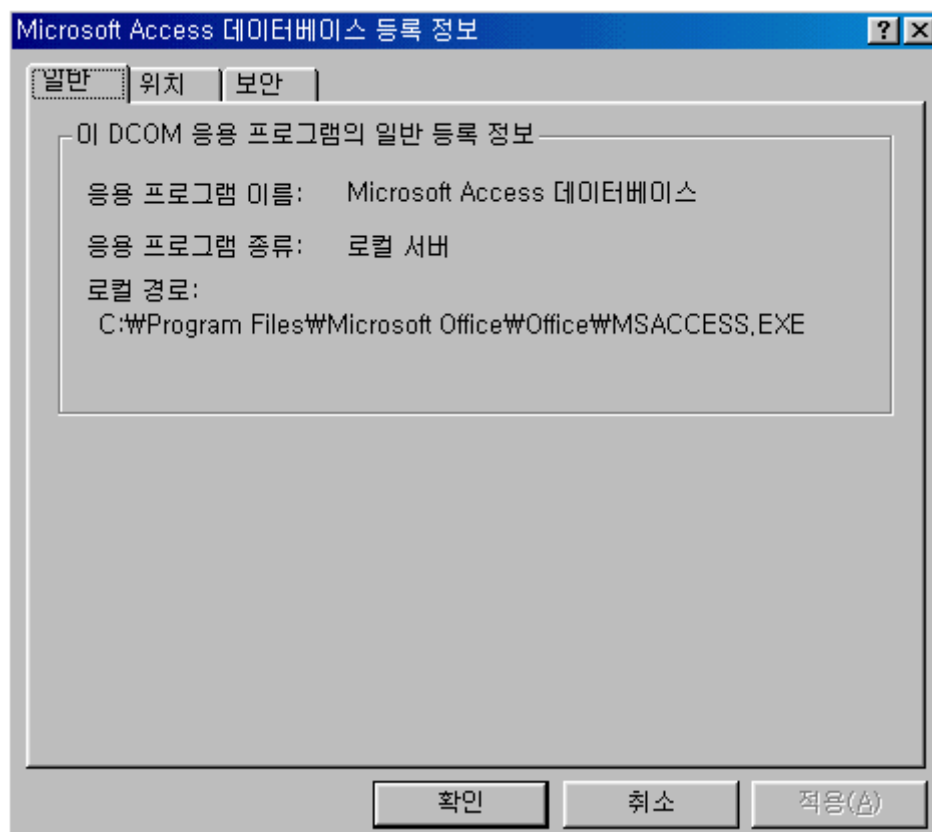
이 대화 상자에서 ‘추가’ 버튼을 클릭하면 다음과 같이 서버에서 관리하고 있는 모든 사용자 목록이 나타나는데, 서버 어플리케이션에 접근해야 하는 모든 사용자들에게 접근을 허용하도록 설정한다.



여기에서 ‘세계’를 선택하면 모든 사용자에게 모두 사용 가능하도록 설정하는 것이다. 윈도우 NT 의 경우에는 다소 화면이 다르게 나타날 수 있는데 ‘세계’에 해당되는 것은 ‘Everyone’이다. 참고로 필자는 윈도우 NT 5.0 영문 베타를 사용하고 있는 관계로 한글 NT 의 해당 이름을 알지 못하므로 한글 NT 를 사용하고 있는 독자들은 한글로 해당되는 이름을 찾아보기 바란다.

이런 방법을 사용하지 않고, 서버 어플리케이션을 설치하고 각 어플리케이션마다 접근 허용을 다르게 설정하는 방법도 있다.

이럴 경우에는 DCOM 구성 등록 정보 윈도우의 응용 프로그램 탭에서 어플리케이션을 선택하고 ‘등록정보’ 버튼을 클릭하면 다음과 같이 이 어플리케이션에 대한 정보를 변경할 수 있는 대화상자가 나타날 것이다.



여기에서 ‘보안’ 탭을 선택하면 앞에서 전체적으로 액세스 권한을 설정했던 것과 동일한 대화상자 들이 나타날 것이다. 이를 이용하여 각 어플리케이션 별로 접근 레벨을 다르게 설정할 수 있다.

그리고, 이 예에서는 나타나지 않았지만 ‘응용프로그램 실행’ 탭이 있는 경우에는 이 탭에서 선택할 수 있는 체크 박스가 ‘데이터가 있는 컴퓨터’, ‘현재 컴퓨터’, ‘컴퓨터 선택’의 3 가지가 나타나는데, 여기서 ‘현재 컴퓨터’ 박스는 선택하면 안된다. 이를 선택하면 현재 컴퓨터에 있는 서버만 실행이 가능하므로 DCOM 으로 설정한 의미가 없다. 그리고, ‘현재 컴퓨터’와 ‘컴퓨터 선택’이 모두 선택된 경우에는 현재 컴퓨터가 우선한다. 그러므로, DCOM 서버로 작성한 서버 프로그램이 서버에서 원격으로 실행되게 하려면 ‘컴퓨터 선택’ 탭을 선택하고 서버의 DNS 주소, IP 주소, 호스트 이름 등을 지정하면 된다.

참고로 윈도우 NT 에서 DComConfig.exe 유틸리티를 사용할 경우에는 Identity, Endpoints, HTTP 탭이 더 있는데 이 중에서 Identity 탭을 선택하고 ‘The interactive user’로 설정하는 것이 좋다.

기본적인 설정이 끝났으면 NT 에서 사용자 관리자(User Manager)를 실행하여, Guest 계정을 선택하고 이 계정의 정보에서 'Account Disabled' 체크 박스가 선택되어 있으면 이를 제거해야 한다.

공유 액세스 제어를 선택한 경우에는 레지스트리 값을 변경할 필요가 있다.

HKLM\Software\Microsoft\WOLE 키의 내용 중에서 EnableRemoteConnect = "Y", LegacyAuthenticationLevel = 1 로 설정하도록 한다. 이 작업은 사용자 액세스 제어를 사용할 때에는 DcomCnfg.exe 유틸리티를 통해서 수정이 가능하기 때문에 매뉴얼로 키를 변경할 필요가 없다. DcomCnfg.exe 유틸리티에서는 기본 등록 정보 탭에서 인증 수준을 '없음'으로 선택하고, 기본 보안 탭에서 '원격 연결 사용' 체크 박스를 선택하면 된다 (디폴트로 선택되어 있다).

그리고, 텔파이를 이용하여 작성한 어플리케이션 중에서 MIDAS 와 같이 필요한 파일이 더 있는 경우에는 해당 파일을 System 또는 System32 디렉토리에 복사하도록 한다. 대표적인 파일로는 DBCLIENT.DLL, STDVCL32.DLL 파일 등이 있다.

이제 서버 어플리케이션을 적당한 로컬 드라이브에 복사하고, 필요에 따라서 BDE 나 SQL Link 등을 설치하고 필요한 앨리어스 등을 생성한다.

서버 어플리케이션을 한번 실행하면 레지스트리에 등록되므로 쉽게 사용할 수 있고, in-process 서버는 DCOM 에서 사용되지 않는다.

만약 DCOM 95 1.0 버전을 이용하고 있다면, RPCSS.EXE 의 단축 아이콘을 시작 폴더에 위치시키는 것이 좋다.

이런 환경 설정의 변경이 효력을 발생하려면 시스템의 재시작이 필수적이다.

이제 서버 어플리케이션을 실행하면 클라이언트와의 연결이 가능하다.

## ● 클라이언트의 환경 설정

서버에 비해 클라이언트의 환경 설정은 더 간단하다. 먼저, 윈도우 95 를 사용하는 독자라면 앞서 소개한 사이트에서 DCOM95 를 다운로드 받아 설치하기 바란다.

기본적인 설정 방법은 서버와 동일하므로 앞의 내용을 참고하기 바란다.

공유 방법으로 공유 액세스 제어를 사용하는 클라이언트인 경우에는 서버에서와 마찬가지로 레지스트리의 내용을 편집할 필요가 있다. HKLM\Software\Microsoft\WOLE 키에서 LegacyAuthenticationLevel = 1 로 설정한다.

설정이 끝났으면, 클라이언트 어플리케이션을 설치하고 실행하면 된다. 텔파이로 작성한 어플리케이션을 사용하는 경우에는 경우에 따라서 DBCLIENT.DLL, STDVCL.DLL 파일을 윈도우 시스템 디렉토리에 복사하거나, BDE 나 SQL Link 를 설치할 필요가 있다.

## ● 인터넷 이용을 위한 DCOM 설정

DCOM 을 인터넷과 방화벽(firewall)을 넘어서 사용할 수 있도록 설정하기 위해서는 다음과 같은 사항을 고려해야 한다.

1. 서버와 클라이언트에서 모두 DcomCnfg.exe 유틸리티를 실행하여 인증 수준을 ‘없음’으로 설정한다.
2. 레지스트리의 내용을 다음과 같이 수정한다.

HKLM/Software/Microsoft/Rpc/Internet

PortsInternetAvailable="Y"

UseInternetPorts="Y"

Ports="3000-4000"

3. 135 번 포트를 열고 방화벽을 넘도록 한다.
4. IP 주소 번역을 Disable 한다.

DCOM 에 대한 HTTP 터널링(tunneling)은 NT SP4 에서부터 사용할 수 있게 되었다. 보다 자세한 내용은 마이크로소프트에서 제공하는 정보를 참고하기 바란다.

#### ● 에러 메시지 정리

에러 메시지	원 인
DCOM not installed	DCOM 이 설치되지 않았다.
Server execution failed	<ol style="list-style-type: none"> <li>1) 레지스트리의 EnableRemoteConnect 값이 N 으로 설정된 경우</li> <li>2) 서버 어플리케이션이 실행되지 않은 경우</li> <li>3) 서버가 로컬 드라이브에 존재하지 않는다.</li> <li>4) 서버 객체에 접근하는 사용자의 권한이 미달된 경우</li> <li>5) 레지스트리의 서버 패스가 지나치게 긴 경우</li> <li>6) DCOMCNFG.exe 에서의 어플리케이션 위치 설정이 틀린 경우</li> </ol>
Class not registered	서버 어플리케이션이 아직 등록되지 않음
RPC server is unavailable	<ol style="list-style-type: none"> <li>1) RPCSS 가 서버에서 실행되지 않음</li> <li>2) 레지스트리의 EnableRemoteConnect 값이 N 으로 설정된 경우</li> <li>3) 잘못된 RemoteServer, ComputerName</li> <li>4) TCP/IP 설정이 잘못된 경우</li> </ol>
Interface not supported	<ol style="list-style-type: none"> <li>1) 사용자 접근 레벨로 설정되지 않은 경우</li> <li>2) Permission 이 Everyone 에게 허용되지 않은 경우</li> <li>3) 레지스트리의 LegacyAuthenticationLevel 값이 설정되지 않은 경우</li> </ol>

	4) Guest 계정이 disable 된 경우 5) 클라이언트가 클라이언트 기계에 인터페이스를 등록하지 않고 vtable 바인딩을 시도하는 경우
Access is denied	1) DCOM 보안 설정이 적절치 못한 경우 2) 서버 어플리케이션이 로컬 드라이브에 없는 경우

## 정 리 (Summary)

이번 장에서는 액티브 X 기술을 이용하여 어플리케이션이나 객체를 개발하면서 만날 수 있는 여러가지 문제점과 그 해결책에 대해서 알아보았다.

여기에 나열한 여러가지 팁 들보다 실제로 구현 과정에 들어가면 예상외로 많은 난관에 부딪히게 된다. MS 에서는 나름대로 분산 환경을 지원하기 위한 표준화 방안으로 DCOM 을 제시한 것이지만, 실제로 이를 활용하여 분산 환경을 구축하는 데에는 아직도 많은 어려움이 있으며, 개선해야 할 부분이 많은 것으로 생각된다.

그렇지만, 앞서 나가는 개발자가 되기 위해서는 이러한 문제점 들을 파악하고 나름대로의 해결책을 찾으려고 노력하는 자세가 필요할 것이다.

## 마이크로소프트 트랜잭션 서버의 이용 (I)

제대로 된 확장성을 고려한 클라이언트/서버 시스템을 구현하는 것은 매우 어렵다. 이런 형태의 시스템을 제작하는 과정에서는 몇 가지 어려운 난관에 봉착하게 된다. 쓰레드가 여러 개 사용되는 형태의 복잡한 어플리케이션에서는 쓰레드를 안전하게 관리하고, 동시에 세션의 통신을 효과적으로 유지하면서 동시에 네트워크 트래픽을 최소한으로 줄이고, 서버와 데이터베이스 리소스를 적게 잡아먹게 작성하여야 한다.

분산 환경의 확장성이 뛰어난 클라이언트/서버 어플리케이션을 제작하는데 있어 이런 문제를 해결하기 위해 제공되는 라이브러리로 대표적인 것을 마이크로소프트 트랜잭션 서버(MTS, Microsoft Transaction Server)를 들 수 있다.

이번 장에서는 마이크로소프트 트랜잭션 서버의 사용 방법과 개념, 그리고 텔파이에서 이를 활용하는 방법에 대한 기초적인 내용에 대해서 다루고자 한다.

### 마이크로소프트 트랜잭션 서버의 기능

- 분산 컴퓨팅 (Distributed Computing)과 멀티 tier 환경

분산 컴퓨팅 환경이 필요한 이유가 뭘까 ? 그것은 최근의 어플리케이션 요구 사항이 하나의 기계로는 감당할 수 없을 만큼 커진 것이 하나의 원인이고, 또 하나는 하드웨어를 교체하기 보다는 추가로 하드웨어를 구입해서 전체적인 성능을 높이는 것이 여러가지로 효율적이기 때문일 것이다.

이렇게 네트워크 상의 각 컴퓨터를 하나의 노드로 보고, 이들 노드에 컴퓨팅 로드를 분산시키는 것이 분산 컴퓨팅 환경이라고 할 수 있다. 여기서 반드시 고려해야 하는 것은 각 노드를 연결하고 있는 네트워크 환경이 컴퓨터 내부의 프로세서와 메모리의 연결을 연결하는 구조에 비해 훨씬 느리다는 점이다.

여기서 확장성의 문제가 가장 많이 발생하게 된다.

전통적인 2 tier 클라이언트/서버 시스템에서 이런 문제를 극복하기 위해서 제안된 것이 3 tier 모델이다. 이 모델에서 클라이언트 어플리케이션은 presentation tier 에 해당되고, 조작하는 데이터가 data tier, 그리고 그 중간에서 전체적인 흐름을 조율하는 workflow tier 로 나누어볼 수 있다.

일반적으로 이런 workflow tier 에 비즈니스 규칙(business rule)을 위치시키고, 여기에서 데이터의 무결성을 강화할 수 있다. 이 부분에 중요한 내용을 위치시키게 되면, 이후에 확장을 하게 될 때 workflow tier 의 수정만으로 쉽게 확장성을 담보할 수 있게 된다.

MTS 에서 제공되는 런타임 환경에서 동작하는 컴포넌트가 바로 이런 workflow tier 가 된다.

- MTS 구성 요소

MTS 에 대한 기능을 알아보기 전에, 먼저 MTS 를 구성하는 주요 요소에 어떤 것들이 있는지에 대해서 알아보도록 하자.

1. 기초 프로세스 (Base Process)

데스크탑 컴퓨터에서 동작하는 클라이언트 응용 프로그램이나 웹 브라우저가 여기에 해당한다. 서버에 어떤 작업을 처리하도록 요청함으로써 트랜잭션을 시작하는 동기를 제공한다.

2. 어플리케이션 컴포넌트 (Application Components), MTS 실행부 (MTS Executive)

비즈니스 로직이 구현되어 있는 부분으로 COM 객체로 작성되어 있다. MTS 에 설치되어 MTS 실행부에 의해 수행된다. MTS 실행부는 응용 프로그램 컴포넌트를 위한 런타임 서비스 환경을 제공한다.

3. 리소스 디스펜서 (Resource Dispenser), 리소스 관리자 (Resource Manager)

리소스 디스펜서는 한 프로세스 내에서 공유되는 상태(데이터베이스의 연결 상태 등) 데이터를 관리한다. 그에 비해 리소스 관리자는 지속적으로 유지되는 데이터를 관리해주는 시스템 서비스를 말하는 것으로 이러한 리소스 관리자로는 마이크로소프트의 SQL 서버, 메시지 큐, 트랜잭션을 지원하는 파일 시스템 등이 있다.

- MTS 서비스

그러면, MTS 에 의해 제공되는 서비스에 대해서 알아보도록 하자.

1. 세션 관리 (Session Management)

언제나 클라이언트와 서버 사이의 연결을 관리하는데에는 세션이 관계하게 된다. 세션은 그 상태를 저장하기 위해 메모리 리소스를 요구하며, 동시에 이를 업데이트하고 관리하기 위해서



는 프로세서 리소스를 요구한다. 그런데, 이렇게 세션을 관리하는데 들어가는 리소스는 보통 서버 측에서 주로 부담하게 된다. 또한 전통적인 2-tier 방식의 경우에 일반적으로  $n$  개의 서버에  $m$  개의 클라이언트가 접속한다고 할 경우에 실제로 필요한 연결의 수는  $m * n$  이 된다. 예를 들어 100,000 개의 클라이언트가 100 개의 서버에 접속한다고 할 때 실제로 관리하게 되는 분리된 세션의 수는 10,000,000 개나 된다.

Workflow tier 의 가장 핵심적인 역할 중의 하나가 이러한 전체 세션을 관리하는 것이다. 예를 들어, 10 개의 새로운 workflow 컨트롤러를 클라이언트와 서버 사이에 위치시키고 각각의 컨트롤러가 클라이언트의 10%를 관리한다고 할 때, 앞의 예에서의 전체 100,000 개의 클라이언트 중 각 컨트롤러가 담당하는 클라이언트는 10,000 개가 되며 세션은  $10,000 * 1$  인 10,000 개를 관리하게 된다. 각각의 workflow 컨트롤러는 또한, 모든 서버들과의 연결을 담당하므로 100 개의 서버에 대해 이들은 100 개의 세션을 관리하게 된다. 그러므로 관리하는 총 세션의 수는 각 컨트롤러 당 10,100 개가 되며, 총  $10 * 10,100 = 101,000$  개의 세션이 된다. 이 숫자는 2-tier 모델에서의 10,000,000 개의 세션에 비해 약 1/100 정도 밖에 되지 않는다.

## 2. 서버 리소스 관리 (Server Resource Management)

데이터베이스 연결은 꽤나 리소스를 많이 잡아 먹는다. 그렇기 때문에 많은 수의 클라이언트 접속이 필요하고, 또한 이 숫자가 더욱 늘어나게 된다면 이들의 연결을 유지하기 위해 많은 양의 서버 리소스가 필요하게 된다.

이때 workflow tier 에 커넥션 풀링(connection pooling)이라는 스키마를 이용해 데이터베이스 연결을 효과적으로 공유하는 로직을 사용하면 리소스를 많이 절약할 수 있다.

MTS 에는 ODBC 리소스 디스펜서(ODBC resource dispenser)라는 컴포넌트가 포함되어 있는데, 이 컴포넌트는 자동으로 ODBC 연결을 풀링한다. 풀링을 이용하면 ODBC 데이터베이스로 작업을 할 때 ODBC 연결을 생성하거나 이를 파괴하는 오버헤드를 줄여줄 수 있기 때문에, 상당한 수행능력의 향상을 기대할 수 있다.

## 3. 데이터 컨텐션 (Data Contention)

다중 사용자 데이터베이스 어플리케이션의 경우, 확장성에 있어 가장 커다란 장애를 들라고 하면 데이터베이스 레코드 들에 대한 컨텐션(contention)을 들 수 있다. 보통 여러 벤더 들은 레코드의 잠금(locking) 기술을 최적화는데 많은 시간과 돈을 투자한다. 그렇지만, 실제로 데이터베이스 어플리케이션을 개발하는 개발자들이 데이터베이스 레코드 들의 컨텐션을 최소화할 책임을 다소는 지고 있다.

개발자는 2 가지 요소를 조절할 수 있다. 어플리케이션에 의해 좌우되는 데이터베이스 잠금의 수와 어플리케이션이 한번 잠금을 할 때 사용하는 시간이 그것이다. 어플리케이션 코드를 다시 디자인하면 어플리케이션에 의해 사용되는 잠금의 수를 최소화할 수 있다. 또한, 데이터를 관리하는 코드를 데이터베이스에 보다 가깝게 위치시킬 수 있다면 아마도 데이터베이스 잠금에 걸리는 시간을 줄일 수 있을 것이다.

전통적인 2-tier 클라이언트/서버 어플리케이션은 SQL 저장 프로시저(SQL stored procedure)를 이용하여 개발자가 잠금을 할 때 걸리는 시간을 최소화한다. 그렇지만, SQL 문장은 기본적으로 OOP 언어와 맞지 않는다.

MTS 객체들은 기본적으로 3-tier 클라이언트/서버 시스템에서 미들 tier 에 위치하며, 제대로 디자인되면 데이터베이스에 접근하는 어플리케이션의 로직을 중앙집중적으로 관리할 수 있도록 해준다. 이렇게 데이터로의 접근을 컴포넌트로 중앙집중함으로써 커넥션 풀링과 데이터베이스 서버로의 연결 속도를 높일 수 있으며, 데이터베이스 잠금에 걸리는 소요 시간을 줄여줄 수 있다.

#### 4. 네트워크 문제

어플리케이션의 확장성에 제한점이 될 수 있는 또 하나의 요소로는 네트워크 트래픽을 들 수 있다. 최근의 OOP 스타일의 클라이언트/서버 어플리케이션의 경우 보통 전형적으로 서버 컴퓨터에 인스턴스를 생성하고, 객체에서 하나 이상의 메소드를 실행하게 하며, 실행이 끝나면 서버 컴퓨터에서 생성된 인스턴스를 파괴하게 된다.

보통 데이터베이스 어플리케이션에서 가장 커다란 비중을 차지하는 것이 데이터베이스 연결과 레코드 잠금이다. 그러므로, 어플리케이션의 확장성을 좋게 하려면 이런 리소스를 보존하는 것이 중요하다. 이를 위해서 클라이언트 어플리케이션 프로그래머는 전통적으로 이들을 필요할 때만 생성해서 사용한 뒤에 이를 파괴하는 형태로 프로그래밍을 하게 된다.

이런 전술은 리소스를 절약하는데에는 도움이 되겠지만, 서버 측의 객체를 생성하고 실제 메소드를 실행하고 파괴하는 등 3 차례 네트워크에 접근해서 작업을 해야하므로 부하를 걸게 만든다.

MTS 는 컨텍스트 wrapper 객체(context wrapper object)라는 것을 이용하여 이런 네트워크 부하를 최소화하는 역할을 한다. 컨텍스트 wrapper 객체는 연결된 컨텍스트 객체와 함께 객체 런타임 환경(object runtime environment)을 형성하는데, 이 환경은 클라이언트 어플리케이션에서 바라볼 때 개념상 하나의 객체로 간주하게 된다.

MTS 에서 클라이언트가 MTS 객체를 생성하고, 파괴할 필요가 없게 하기 위해서, 클라이언트 어플리케이션은 MTS 객체를 그때 그때 생성하고 파괴하는 것이 아니라, 최대한 빨리 MTS 객

체에 대한 레퍼런스를 얻어다가 최대한 늦게 이를 해제한다. 이렇게 함으로써 MTS 는 객체를 사용하는 메소드 호출이 있을 때마다 MTS 객체가 인스턴스를 생성하도록 한다. 객체가 MTS 에 메소드 호출이 끝났음을 알리면 MTS 는 이를 파괴하지만, 컨텍스트 wrapper 는 다음 번 요구에 대비하기 위해서 살아있게 된다. 이렇게 함으로써 MTS 가 객체의 생성과 파괴에 대한 오버헤드를 줄임으로써 전체적인 수행 성능을 높이게 된다.

MTS 의 네트워크 트래픽을 줄이기 위한 이러한 최적화 방법을 just-in-time 활성화(activation)이라고 하는데, 여기에서 중요한 역할을 하는 것은 실제 객체에 의해 구현된 인터페이스에 대한 메소드를 구현한 컨텍스트 wrapper 객체이다. 이 객체는 MTS 에 의해 런타임에서 동적으로 생성되며, 클라이언트 어플리케이션과 실제 MTS 객체 사이에 삽입된다.

## 5. 트랜잭션 (Transactions)

분산 어플리케이션을 제작할 때에는 확장성 이외에도 컴퓨터 시스템에 복잡성이 증가하기 때문에, 여기에 따른 시스템 failure 가능성이 높아질 수 밖에 없어 이를 고려해야 한다. 잘 디자인된 분산 어플리케이션은 확장성도 좋지만, 이런 failure 가 있을 때 이를 자동으로 복구할 수 있는 고려가 들어가 있다.

대부분의 분산 환경에서 이런 자동 복구를 구현하기 위해서 트랜잭션을 사용하고 있다. 트랜잭션이란 여러 가지 기능의 복잡한 세트를 하나의 단위로 간주하여 사용하는 것을 말한다. 트랜잭션은 클라이언트 어플리케이션과 서버에서 실행되는 코드 사이의 일종의 계약과도 같아서, 일단 서버의 코드가 클라이언트에게 트랜잭션이 성공했음을 알리면, 그것은 클라이언트에서 요구한 내용이 완전히 저장되었다는 의미이며, 반대로 실패했음을 알리면 클라이언트에서 요구한 내용이 전혀 수행되지 않는다.

트랜잭션은 에러 처리의 가능성을 많이 줄여 준다. 트랜잭션을 이용하면 전부 실행(commit)되거나, 아예 실행되지 않으므로(rollback) 에러 복구를 위해 일부 실행된 내용들을 undoing 할 필요가 없다.

트랜잭션을 보다 효과적으로 사용하려면, 하나 이상 중첩된 트랜잭션을 사용할 수 있다. 즉, 하나의 커다란 트랜잭션을 마치 트리와 같은 형태로 여러 개의 중첩된 트랜잭션이 포함된 트랜잭션으로 재구성하는 것이다. 이렇게 하면, 중첩된 트랜잭션이 실패할 경우 실패한 트랜잭션만 복구하거나 전체 트랜잭션을 취소하는 것을 선택할 수 있다.

## 6. Data Tier 의 확장

트랜잭션을 분산 컴퓨팅 환경에서 유용하게 사용하려면, 분산 환경을 구축할 때 병목 구간이

될 수 있는 data tier 를 확장할 수 있어야 한다. 즉, 아무리 여러 클라이언트와 workflow tier 에서의 비즈니스 처리를 분산 환경에서 할 수 있더라도, data tier 인 실제 데이터베이스 서버를 한 군데에서 처리해야 한다면 데이터베이스에 접근하는 부분에서 병목이 될 수 밖에 없다. Data tier 를 확장하기 위해서는 하나 이상의 데이터베이스를 MTS 트랜잭션에서 사용할 수 있어야 한다. 즉, 논리적인 데이터베이스를 하나 이상의 데이터베이스 서버 노드에 분산하는 것이다. 이를 위해서 MTS 는 마이크로소프트의 DTC(Distributed Transaction Coordinator)를 이용하여 트랜잭션이 하나 이상의 데이터베이스 서버에서 이루어질 수 있도록 허용하고 있다. 이때 DTC 는 산업 표준인 X/Open XA 프로토콜, IBM 의 LU6.2 프로토콜, 마이크로소프트의 OLE 트랜잭션 프로토콜 등을 이용하게 된다.

## 7. 보안 (Security)

보안 프로그래밍과 연관된 세부적인 내용을 해결하기 위해 MTS 는 role 에 기반을 둔 보안 모델을 제공한다. MTS 의 보안을 이용하기 위해서는 일단 MTS 컴포넌트를 패키지로 그룹화하고, 각각의 패키지에 role 을 부여해야 한다. 예를 들어, Manager 와 Data Entry Clerk 이라는 role 을 미리 정의했다고 하면, 배포할 때 윈도우 NT 관리자가 실제 사용자에게 개발 당시에 미리 정의한 role 을 부여할 수 있다. 런타임에서 사용자들은 자신에게 부여된 role 에 따라 MTS 패키지에서 특정 기능에 접근할 수도 있고, 접근이 거부될 수도 있다.

Role 에 기반을 둔 보안을 이용하여 MTS 는 자동으로 보안 검사를 해주는 셈이다. 클라이언트 어플리케이션이 MTS 객체를 호출하게 되면, MTS 는 클라이언트에 대한 보안 검사를 수행한다. 이런 보안 검사는 인증(authentication)과 접근 제어(access control)와 관련이 있다. 인증은 사용자 자체에 대한 보안 검사인데, MTS 는 기존의 윈도우 NT 보안 인프라를 그대로 유지하면서 이를 보다 쉽게 만들어 준다. 접근 제어는 사용자가 특정 작업을 하고자 할 때 이를 허용할 것인지 여부를 결정하는 것으로 MTS 는 사용자에게 부여된 role 을 가지고 사용자의 요구를 수행할 것인지 여부를 결정한다.

기억해야 할 것은 MTS 패키지 내의 모든 MTS 객체 들은 동일한 보안 컨텍스트를 공유한다는 점이다. 그러므로, 보안 검사는 패키지 범위 내에서 수행되는 것이지 같은 패키지 내의 컴포넌트들 사이의 보안 검사는 모두 동일하다.

### ● 자주 사용되는 용어

MTS 에 대해 더 알아보기 전에, 엔터프라이즈 환경에서 사용되는 몇 가지 용어에 대해 다음에 정리해 보았다.

용 어	설 명
네트워크 리시버 (network receiver)	네트워크를 통한 클라이언트로부터 호출을 받아들이고 병목 현상을 관리하는 역할을 하는 단위
커넥션 관리자 (connection manager)	각 클라이언트에 대한 작업과 시스템 리소스를 추적, 관리한다.
쓰레드 풀 (thread pool)	한 쓰레드가 각 사용자에게 전유되는 것을 막아 시스템 리소스를 효율적으로 사용할 수 있도록 한다.
컨텍스트 관리자 (context manager)	각 사용자에 대해 현재 사용자를 확인하고 그 상태를 추적한다.

## ● MTS 프로그래밍 모델

MTS 는 COM 이 제공하는 컴포넌트 인프라 구조로 사용할 수 있도록 디자인되었다. MTS 는 COM 객체 들에 대한 런타임 인프라 구조를 제공한다. COM 객체가 MTS 객체로 사용되기 위해서는 MTS 런타임 지원을 받기 위한 몇 가지 요구 사항을 지켜 주어야 한다.

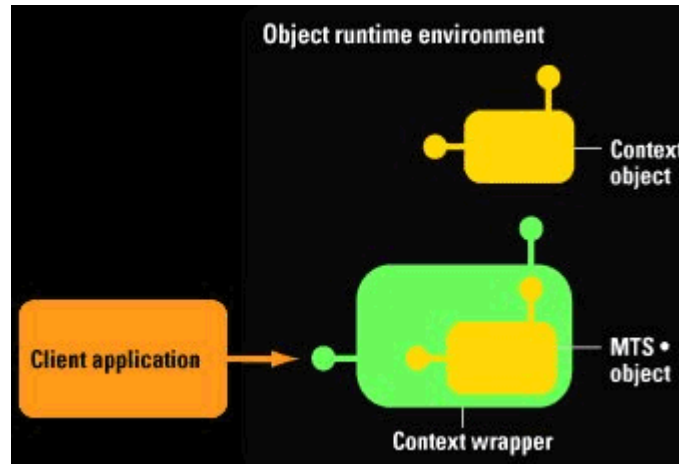
MTS 객체는 반드시 32 비트 in-process COM 서버로 제공되어야 한다. MTS 는 MTS 객체에 의해 구현된 모든 인터페이스의 파라미터 데이터 형과 함수 이름을 알고 있어야 한다. 그러므로, MTS 객체로 구현하기 위해서는 인터페이스 정의에 대한 정보가 제공되어야 한다. 이를 위해서 MTS 는 동적으로 컨텍스트 wrapper 객체를 생성한다. MTS 객체의 인터페이스는 반드시 자동화에 호환되거나 다른 표준 마샬링 인터페이스를 지원해야 한다.

### 1. 컨텍스트 wrapper

컨텍스트 wrapper 객체는 실제 MTS 객체로 외부 클라이언트에서 보일 수 있도록 하는 객체이다. 클라이언트 어플리케이션이 MTS 객체에 대한 레퍼런스를 얻을 때, 실제로는 MTS 에 의해 런타임에서 생성되는 컨텍스트 wrapper 객체를 참조하게 된다. 컨텍스트 wrapper 객체는 just-in-time 활성화를 구현하며, 자동으로 보안 검사를 한다.

클라이언트 어플리케이션이 MTS 객체의 메소드를 호출할 때에는 MTS 가 먼저 앞서 설명한 방법으로 보안 검사를 수행한다. 클라이언트 어플리케이션이 보안 검사를 통과하면 컨텍스트 wrapper 객체는 객체의 인스턴스에 대한 레퍼런스를 가지고 있는지 검사하게 된다. 대부분의 MTS 객체의 경우에는 컨텍스트 wrapper 객체는 반드시 객체의 새로운 인스턴스를 생성하고, 객체의 실제 메소드를 호출하게 된다. 메소드가 컨트롤을 컨텍스트 wrapper 객체에 돌려 주면, SetAbort 나 SetComplete 메소드를 호출했는지 확인하고 그렇다면 컨텍스트 wrapper 는 컨트롤을 클라이언트 어플리케이션에 반환하기 전에 객체를 파괴한다.

MTS 객체와 컨텍스트 wrapper, 그리고 컨텍스트 객체는 모두 MTS 의 객체 런타임 환경 (object runtime environment)에 포함된다. 다음 그림과 같이 MTS 객체는 컨텍스트 wrapper 에 의해 캡슐화되고, 각각의 MTS 객체마다 하나의 컨텍스트 객체를 가지게 된다.



## 2. 컨텍스트 객체 (Context Objects)

MTS 객체의 각각의 인스턴스는 유일한 컨텍스트 객체를 가지게 된다. 이를 사용하기 위해서는 MTS API 함수인 `GetObjectContext` 함수를 이용하여 MTS 객체의 컨텍스트 객체에 대한 레퍼런스를 가져올 수 있다. 컨텍스트 객체는 `IObjectContext` 인터페이스를 구현하는 COM 객체이다.

MTS 프로그래머는 `IObjectContext` 인터페이스의 `SetAbort`, `SetComplete` 메소드를 자주 이용하는데, 이렇게 함으로써 MTS 에게 객체에 대한 처리를 마쳤으므로 함수를 빠져나갈 때 객체를 파괴해도 된다는 것을 알려야 한다.

이 과정을 오브젝트 파스칼 코드로 간단하게 설명하면 다음과 같다.

```

var
  Context: IObjectContext;
begin
  Context := GetObjectContext;
  try
    ... 여러가지 처리
    Context.SetComplete;
  except
  
```

```
Context.SetAbort;  
end;  
end;
```

또한 트랜잭션에 추가적인 MTS 객체 들을 나열하고, 프로그램으로 보안을 구현하기 위해서 컨텍스트 객체의 메소드를 이용할 수도 있다. IsCallerInRole 메소드를 이용하면 호출자의 접근 토큰이 MTS 탐색기(MTS Explorer)에 의해 부여된 role 중에서 일치하는 것이 있는지 검사할 수 있다. 또한, CreateInstance 메소드를 이용하면 MTS 객체의 인스턴스를 새로 생성해서 이를 현재의 트랜잭션 범위에서 사용할 수 있도록 추가하는 것이 가능하다.

#### ● IObjectContext 인터페이스

MTS 어플리케이션은 하나 이상의 MTS 객체 들로 구성되어 있다. 이들은 in-process COM 서버로 컴파일 되어 위치한다. 클라이언트는 MTS 객체에 다른 COM 객체에 접근하는 것과 마찬가지로 접근할 수 있다. MTS 자체는 클라이언트에서 볼 수 없으며, MTS 에 특이한 코드가 따로 존재하는 것은 아니지만 실제로 클라이언트가 객체의 메소드를 호출할 때 MTS 객체에 접근하는 경우에 실제로는 MTS 객체가 중간에 이를 가로채서 관리하게 된다.

MTS 객체가 생성될 때, 일반 COM 객체와 다른 점으로는 MTS 는 항상 컨텍스트 객체를 관리한다는 점이다. 이런 컨텍스트 객체 역시 IObjectContext 인터페이스를 지원하는 COM 객체이다. 이 인터페이스의 역할은 MTS 객체가 MTS 실행부(MTS Executive)와의 통신을 담당한다. 이들을 호출하기 위해서는 MTS 객체가 해당 컨텍스트 객체의 IObjectContext 인터페이스에 대한 포인터를 얻어야 한다. 이를 위해 사용하는 API 함수가 바로 GetObjectContext 함수이다.

IObjectContext 인터페이스는 몇 가지 메소드를 포함하는데, 그 중에서도 가장 중요한 것이 SetComplete 와 SetAbort 메소드이다.

앞에서도 간단히 설명한 바 있지만, 이들 메소드를 MTS 객체가 호출하는 것은 MTS 실행부에 더 이상 관리할 데이터가 없다는 것을 선언하는 것임과 동시에 트랜잭션의 사용 여부와 상관없이 SetComplete 일 경우 객체가 수행한 기능을 commit 하고, SetAbort 인 경우에는 rollback 하는 역할을 한다. 다시 말해 이들 메소드가 호출된다는 것은 더 이상 객체가 존재할 필요가 없다는 것을 선언하는 것이다.

이를 가능하게 하려면, MTS 실행부가 모든 MTS 객체를 잘 포장하고 MTS 객체가 SetComplete 나 SetAbort 메소드를 호출할 때 MTS 는 객체가 가지고 있는 모든 인터페이스를 해제해야 한다. 그렇지만 이때 클라이언트는 객체의 인터페이스 포인터라고 믿는 포인터를 가

지고 있게 되는데, 실제로 이 포인터는 MTS 에서 제공하는 wrapper 에 대한 포인터이다.  
이렇게 함으로써 클라이언트가 MTS 객체를 사용할 때의 지속성을 유지하면서, 동시에 실제 MTS 객체에 의해 사용되던 리소스를 다른 객체 들이 사용할 수 있도록 할 수 있다.  
클라이언트가 MTS 객체의 다른 메소드를 다시 호출하면, MTS 는 클래스 팩토리를 이용해서 그 객체의 새로운 인스턴스를 생성한 후 이를 클라이언트에게 넘겨준다.

#### ● MTS 객체의 사용 예

이 구조의 장점을 예를 들어 설명해보자. Add, Remove, Submit 이라는 3 개의 메소드를 가지는 IOrderEntry 인터페이스를 구현한 MTS 객체가 있다고 하자. 클라이언트는 이를 구현한 MTS 객체를 이용하여 주문서에 제품을 추가, 삭제, 발송한다고 하자. 여기서 인터페이스의 Add 메소드가 다음과 같은 형태로 구현되어 있다.

```
procedure TOrderEntry.Add(Item);  
begin  
    If (NoOrderExists) then CreateOrder;  
    Lookup(Item);  
    If (Available) then  
        begin  
            AddItemToOrder(Item);  
            DecrementInventory(Item);  
        end  
    else  
        ErrorHandler;  
end;
```

다시 말해 현재 주문이 존재하지 않으면 일단 하나를 생성한 뒤에, 데이터베이스에서 요구한 아이템이 있는지 검사한 뒤에 그렇다면 아이템을 현재의 주문에 추가하고, 이 아이템에 대한 재고를 하나 감소시킨다. 이 메소드를 구현함에 있어서 SetComplete 나 SetAbort 를 호출하지 않았기 때문에, 객체는 비활성화되지 않고 메모리에 남아 있는 주문이 생성되며 이를 다른 메소드 들이 이용할 수 있다.

Remove 메소드는 다음과 같은 형태로 구현하면 될 것이다.



```

procedure TOrderEntry.Remove(Item);
begin
    If (ItemInOrder) then
        begin
            RemoveItemFromOrder(Item);
            IncrementInventory(Item);
        end
    else
        ErrorHandler;
    end;
end;

```

이 역시 매우 간단하게 구현 했는데, 먼저 현재 주문이 있는지 확인하고 주문에서 아이템을 삭제한 뒤에 재고를 하나 증가시킨다. 역시 여기에서도 SetComplete 나 SetAbort 를 호출하지 않기 때문에 주문에 대한 상태는 여전히 메모리에 남아 있게 된다. 결국에는 Submit 함수에서 이들을 최종적으로 처리하는 것이다.

```

procedure TOrderEntry.Submit;
begin
    Context = GetObjectContext;
    If (EverythingIsOK) then
        begin
            SubmitOrder;
            Context.SetComplete;
        end
    else
        Context.SetAbort;
    end;
end;

```

실제 MTS 객체에 대한 가장 핵심적인 기능을 하는 메소드이다. 먼저 GetObjectContext API 함수를 호출하여 적절한 컨텍스트 객체의 포인터를 얻는다. 특별한 문제가 없으면 주문을 전송하고 SetComplete 를 호출한다. 그러나, 실패한 부분이 있다면 SetAbort 를 호출하게 된다. 이들 메소드에 의해 MTS 는 객체가 가지고 있는 모든 인터페이스 포인터의 Release 메소드를 호출하며, 객체가 사용하던 모든 리소스와 메모리 상의 데이터는 해제된다. 만약 클라이언트가

다시 이 객체에 대한 Add 메소드를 호출하면 MTS 는 이 객체에 대한 새로운 인스턴스를 생성한다.

이런 식으로 객체를 구현하면 클라이언트가 MTS 객체를 생성, 파괴하는 작업이 없이 마치 계속 객체의 인스턴스를 메모리에 띄워 놓은 것처럼 여러 아이템을 그때 그때 알아서 추가, 삭제하고 전송할 수 있다. 그렇지만, 서버 상에서는 실제로 주문이 진행되는 동안에만 객체를 관리해도 되므로 여기에 들어가는 오버헤드를 많이 줄일 수 있다.

물론 Add, Remove, Submit 메소드를 구현할 때 이들 각각에 대해 SetComplete, SetAbort 를 실행하도록 구현할 수도 있다. 그렇지만, 이렇게 할 경우에는 메모리 상에 있는 데이터의 지속성을 보장할 수 없으므로 주문의 내용을 디스크에 저장하고, 이를 읽는 형태를 가져야 한다. 이렇게 하면 물론 확장성은 더 뛰어나다고 할 수 있지만, 디스크 공간이 더 필요하고 클라이언트의 수에 따라서는 수행 성능도 떨어질 수도 있다.

그러므로, 앞으로의 확장성과 수행 성능의 균형을 고려하여 전체적인 형태를 디자인해야 한다.

#### ● 서버와 리소스

데이터베이스에 접속하기 위해서는 상당한 리소스를 소모하게 된다. 또한, ODBC 연결의 경우 한정된 리소스만 가지고 있기 때문에 계속적인 ODBC 연결이 생성되고, 이 연결이 해제되지 않으면 리소스가 부족하게 된다. 그렇다고, 연결을 필요할 때마다 생성하고 사용한 뒤에 해제하는 것은 수행 속도에 문제를 일으키는 경우가 많다.

MTS 에서는 리소스 디스펜서(resource dispenser)를 이용하여 시스템의 공유할 수 있는 비지속성 리소스(RAM 등)를 효율적으로 관리하는 방법을 제공한다. 가장 중요한 리소스 중의 하나가 ODBC 연결에 대한 관리이다. MTS 객체가 데이터베이스에 연결을 요구할 때, 리소스 디스펜서가 이를 풀(pool)에서 가져다가 이용한 뒤, 객체가 연결을 해제하면 이를 다시 풀로 반환한다. 이때 데이터베이스에 연결하는 것이 실제 리소스를 할당하거나 해제하는 것이 아니기 때문에, 이들을 얻고 해제하는 과정이 훨씬 빠르게 이루어지며 MTS 객체들이 리소스를 공유하기 때문에 리소스도 많이 절약할 수 있다.

#### ● 서버의 역할과 보안

클라이언트가 COM 객체의 메소드를 호출할 때 클라이언트가 메소드를 실행할 권한이 있는지 검사해야 한다. 메소드가 파일 등의 다른 리소스에 접근할 경우에는 그런 리소스에 대해서도 접근 권한이 있는지 검사한다. COM 과 DCOM 서버는 IServerSecutiry 인터페이스의 메소드를 이용하여 클라이언트에 권한 설정을 할 수 있다. 참고로 운영체제가 윈도우 NT 일 경우 각

리소스의 ACL 을 자동으로 검사하기 때문에 리소스에 대한 설정을 따로 해 줄 필요가 없다. 이런 형태의 보안은 권한을 얻는 과정이 복잡하기 때문에 다소 확장성이 떨어진다는 단점이 있다. 그러므로 각 리소스 별로 접근 권한을 설정하는 방법 이외에 role 에 기반을 둔 보안을 MTS 가 지원한다. Role 이란 윈도우 NT 사용자 들과 특정 문자열 이름을 부여 받은 그룹 들의 컬렉션이다. Role 은 MTS 컴포넌트의 모임인 패키지 당 하나씩 부여할 수 있다. MTS 관리자는 MTS 탐색기를 이용하여 각 사용자와 그룹에 role 을 부여할 수 있고, 정확히 어떤 role 들이 각각의 MTS 컴포넌트에 접근할 수 있는지를 지정할 수 있다. MTS 는 인터페이스 호출을 한 호출자의 role 을 검사하여 여기에 접근할 수 없다면 호출을 거부한다. 이런 role 에 기반을 둔 보안은 MTS 관리자가 직접 나름대로 변경해서 이용할 수 있기 때문에 장점이 많다. 물론 프로그램에서도 role 을 검사하는 루틴을 삽입할 수 있는데, IsCallerInRole 메소드가 그 역할을 담당한다. 이 메소드를 이용하면 MTS 가 현재 호출자의 role 을 검사하여 role 에 해당하는 코드를 작성하여 사용하게 구현할 수도 있다.

## ● 서버 트랜잭션

사실상 MTS 의 이름에 들어있는 내용을 보면 마치 MTS 가 트랜잭션을 전담해서 관리하는 서버 처럼 들리는 것이 사실이다. 그렇지만, MTS 가 제공하는 대부분의 기능은 COM 서버를 쉽게 확장할 수 있도록 하는 런타임 환경을 제공하는 것이지 트랜잭션 처리는 주된 기능이 아니다. 트랜잭션 자체는 사실 데이터베이스 서버에서 기본적으로 제공하고 있으며, ODBC 에서도 트랜잭션을 사용할 수 있다. 그렇다면, MTS 가 트랜잭션을 처리하는 것에 대해 지원하는 것은 무엇일까? 문제는 하나 이상의 DBMS 를 이용한 트랜잭션을 처리할 경우에 있다. 이런 경우에는 MTS 를 이용하여 트랜잭션을 관리하게 하면 된다.

전통적인 트랜잭션 시스템과는 달리 MTS 는 클라이언트에게 트랜잭션의 경계를 숨긴다. 이런 방식의 장점은 같은 컴포넌트가 내부 트랜잭션에서 실행될 수도 있고, 다른 컴포넌트와 합쳐서 커다란 트랜잭션을 만들 수 있다는 것이다. MTS 가 자동으로 클라이언트의 요구가 있을 때마다 트랜잭션을 생성하기 때문에 여러 컴포넌트를 다양한 방법으로 결합해서 사용할 수 있다.

예를 들어, 앞에서 설명한 제품을 주문하는 시스템 컴포넌트와 은행 계좌간에 현금을 전송하는 컴포넌트가 각각 존재할 때 이들은 각각의 트랜잭션을 처리하지만, 동시에 주문과 현금 거래를 묶어서 실행하는 더 큰 트랜잭션을 묶어서 사용할 수 있다.

예를 들어 클라이언트가 주문장(order entry) MTS 컴포넌트를 생성하고, MTS 가 이 요구를 가로채서 이 컴포넌트가 트랜잭션을 요구하면 자동으로 트랜잭션을 시작한다. 동시에 이 컴포넌트가 IObjectContext 인터페이스의 CreateInstance 메소드를 이용하여 현금 전송 컴포넌트

의 인스턴스를 생성하고, MTS 가 이 컴포넌트를 로드하면 이 컴포넌트에 의해 트랜잭션이 요구된다. 이때 컴포넌트를 생성한 생성자가 이미 트랜잭션의 일부이기 때문에 현금 전송 컴포넌트의 인스턴스는 트랜잭션의 일부가 된다. 현금 전송 컴포넌트가 작업을 끝내면 SetComplete 또는 SetAbort 를 호출하게 되며, 이것으로 트랜잭션이 끝나는 것이 아니라 주문장 컴포넌트가 SetComplete 또는 SetAbort 를 호출해야 전체 트랜잭션이 완료된다. 여기서 이들 컴포넌트가 모두 SetComplete 를 호출한 경우에는 컴포넌트에 의해 변경된 모든 사항이 commit 되거, 하나라도 SetAbort 가 호출되면 전체 변경 사항이 rollback 된다.

이처럼 MTS 를 이용한 트랜잭션 처리는 지금까지 알려진 트랜잭션 처리 방법에 비해 대단히 유연한 것이 장점이다.

## 트랜잭션이 동작하는 방법

트랜잭션이 동작하는 방법을 알기 위해서 앞에서 간단히 예를 들어 설명한 OrderEntry 를 이용하도록 하겠다. 앞에서 아이템을 주문에 추가하기 위해서는 데이터베이스의 재고 물량을 감소시켜야 했다. 이때 주문에 있는 여러 아이템 들에 대한 재고 물량 정보가 서로 다른 2 개의 데이터베이스에 저장되어 있다고 가정하자. 주문이 완료되면 이들 데이터베이스의 재고 정보는 새롭게 갱신되거나 또는 원래의 상태를 유지하여야 한다.

여기에 대해서 설명하기 전에 몇 가지 설명하고 넘어가야 할 부분이 있다.

보통 데이터베이스 시스템이 TP 모니터를 사용한다면 보통 XA 라고 불리는 X/Open 에서 정의한 인터페이스를 지원한다. MTS 는 XA 를 지원하는 데이터베이스와 함께 사용할 수 있다. 마이크로소프트는 TP 모니터와 OLE 트랜잭션이라고 불리는 데이터베이스 사이의 상호작용을 위한 인터페이스를 정의하였다. MS SQL Server 는 현재 이 인터페이스를 지원하고 있으며, 다른 DBMS 도 이를 지원하게 될 것으로 보인다. 어쨌든 트랜잭션이 수행되는 방법은 XA 와 OLE 트랜잭션 중 어느 것을 사용하느냐 여부에 따라 다르다. 여기서는 XA 를 사용하는 방법을 기준으로 설명하겠다.

MTS 객체가 처음 생성되면 MTS 실행부가 이를 감지하게 되고, 이 컴포넌트의 트랜잭션 속성(transaction attribute)을 검사한다. 컴포넌트가 트랜잭션을 필요로 하므로 이 속성이 설정된 것으로 간주하겠다. 이 사실은 새롭게 생성된 MTS 객체와 연관된 컨텍스트 객체에 트랜잭션에 대한 식별자와 함께 저장된다.

트랜잭션을 시작하기 위해서는 MTS 실행부가 DTC(Distributed Transaction Coordinator)에 접근하여 트랜잭션이 시작된다는 것을 기록한다. DTC 는 분리된 프로세스에서 실행되는 윈도우 NT 서비스로 트랜잭션 들을 조화시킨다.

이제 클라이언트가 Add 메소드를 호출하여 아이템을 주문에 추가하려고 하면, 만약 Add 가 처

음으로 호출된 경우에는 메모리에 주문 상태(order state)가 생성된다. Add 메소드는 ODBC 호출을 이용해 적절한 데이터베이스에 연결하게 된다.

MTS 객체가 데이터베이스와의 연결을 요구하면 ODBC 드라이버는 객체의 컨텍스트 객체를 질의해서 이 객체가 트랜잭션에 속해있는지를 알아보게 된다. 만약 트랜잭션에 속해 있다면, 이 경우에 드라이버는 DTC 에서 해당 데이터베이스를 트랜잭션에 추가하고, 데이터베이스에게는 객체의 요구가 트랜잭션의 일부분이라는 것을 알리게 된다.

이 같은 과정이 MTS 객체가 접근하는 각각의 데이터베이스에 대해서 일어나게 된다.

마지막으로 클라이언트는 주문이 Submit 함수를 호출하여 완료하게 되는데, 컨텍스트 객체가 SetComplete 호출을 받으면 DTC 에게 트랜잭션을 commit 하게 된다. 이때 XA 가 사용되면 DTC 는 ODBC 드라이버와 통신을 해서 트랜잭션을 commit 하도록 각각의 데이터베이스에게 요구하게 된다. 이때 XA 대신 OLE 트랜잭션이 사용된다면 DTC 는 ODBC 를 거치지 않고 직접 데이터베이스에 접근할 수 있다.

지금까지 설명한 내용은 MTS 객체를 생성하고, 이를 개발하는데에는 그렇게 중요한 내용이 아니다. 하지만, 트랜잭션이 실제로 어떻게 처리되는지에 대해서 이해하는 것은 향후의 문제 해결이나 개발하는데 기반 지식으로서 큰 도움을 주게 될 것이다.

## MTS 와 COM

COM 을 분산 어플리케이션 개발 과정에 적용할 경우에는 분산 에러 복구와 concurrency 관리, 로드 밸런싱(load balancing), 데이터 consistency 등의 여러 가지 문제점을 해결해야 한다. 불행하게도 COM 은 객체가 어떻게 이런 난점 들을 해결해야 하는지에 대해서는 특별한 해결 방안을 제시하지 않고 있다.

그러므로, 이를 해결하기 위해서 개발자가 나름대로의 스키마를 이용하여 작성을 해야 했다.

여기에 대해서 공통적인 인프라 구조로서 제시된 것이 바로 MTS 이다. COM 프로그래밍 모델은 전통적인 OOP 모델을 나름대로 표준화하고 확장한 것이라고 한다면, MTS 프로그래밍 모델은 COM 프로그래밍 모델에 기본적인 상태(state)와 행동(behavior)에 대한 여러가지 관계를 추가하여 확장한 것이라고 생각하면 된다.

MTS 의 기본적인 원칙은 객체가 논리적으로 상태와 행동을 모델링하는 것이다. 그렇지만, 물리적인 구현 방법은 2 가지로 다시 나누어 볼 수 있다.

어플리케이션 개발자는 MTS 를 이용하여 객체의 상태를 관리한다면 잠금 관리나 에러 복구, 로드 밸런싱과 데이터 consistency 등을 지원하기 위한 짐을 상당히 덜 수 있다. 이는 객체의 대부분의 상태가 가상 함수 포인터로 대별되는 객체의 행동(behavior)와 같이 저장되어서는 안 된다는 것을 의미한다. 대신 MTS 가 객체의 상태를 RAM 과 같은 비지속성(non-persistent)

저장소나 테이블이나 파일과 같은 지속성(persistent) 저장소에 저장하게 된다. 이렇게 저장된 상태는 MTS 런타임 환경의 제어 하에서 잠금 관리나 데이터 consistency 등에 관계 없이 객체의 메소드에 의해 안전하게 접근이 가능하다. 그러므로, 객체의 상태는 기계가 다운되거나 프로그램이 비정상적으로 종료되더라도 유지될 수 있다.

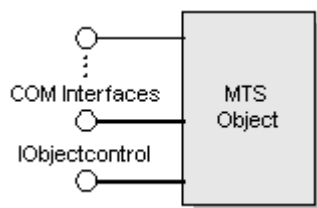
## 델파이 4 와 마이크로소프트 트랜잭션 서버 컴포넌트

MTS 컴포넌트는 DLL 의 형태로 구성되는 COM in-process 서버 컴포넌트이다. 이들은 다른 COM 컴포넌트와는 달리 MTS 런타임 환경에서 실행된다. 이들 컴포넌트를 델파이 4 에서 생성하고 구현할 수 있으며, 다른 액티브 X 호환 개발 툴에서도 사용이 가능하다.

MTS 컴포넌트는 델파이에서 클래스로 구현되었다. 참고로 MTS 에서 일컫는 컴포넌트라는 의미는 COM 객체를 구현하는 코드를 의미하므로 델파이의 컴포넌트와 혼돈하지 않기 바란다. 그러므로, 앞으로 언급하는 MTS 컴포넌트란 MTS 클래스를 의미하는 것으로 받아들이기 바란다.

MTS 서버 객체는 작으면서도 괜찮은 성능을 보여 준다. 예를 들어, MTS 컴포넌트는 비즈니스 물과 어플리케이션 상태에 대한 뷰와 변환 등을 구현할 수 있다. 구체적인 예를 들어, 의학 어플리케이션에서 의료 기록(메디칼 레코드, medical record)은 여러 개의 데이터베이스에 저장되어 관리될 수 있다. 이때 클라이언트 어플리케이션에서는 환자의 그동안의 상태에 대한 정보를 계속 받아본다고 하자. 이럴 때 MTS 컴포넌트는 새로운 환자에 대한 정보, 혈액 검사 결과, 방사선 촬영 결과 등의 상태를 계속 업데이트 해준다.

다음 그림과 같이 MTS 객체는 다른 COM 객체처럼 사용된다. MTS 객체는 COM 인터페이스 이외에 MTS 인터페이스를 지원한다. IUnknown 인 COM 객체에서 공통적인 인터페이스인 것처럼, IObjectControl 은 모든 MTS 객체에서 공통적으로 지원하는 인터페이스 이다. IObjectControl 인터페이스는 MTS 객체를 활성화하거나, 비활성화하는 메소드와 데이터베이스 연결과 같은 리소스를 다루는 메소드를 제공한다,



클라이언트 측에서 보면, MTS 환경 내에 있는 COM 객체는 다른 COM 객체와 별 차이가 없어 보인다. 이럴 때 MTS 는 클라이언트의 요구를 서비스하는 프록시 역할을 하게 된다. 보통

MTS 컴포넌트는 MTS 실행파일(.EXE)에 담겨 있는 in-process 서버이다. MTS 실행파일을 실행하면 MTS 객체는 리소스 관리, 트랜잭션 지원 등의 MTS 런타임 환경의 혜택을 누릴 수 있다.

연결 정보는 MTS 프록시에 의해 관리된다. MTS 클라이언트와 프록시의 연결은 클라이언트가 서버에게 연결을 요구하는 동안 계속 유지되며, 그렇기 때문에 클라이언트는 마치 서버와의 연결이 계속 유지되는 것으로 생각하게 된다.

## MTS 컴포넌트의 요구 사항

MTS 는 COM 에 비해 요구사항이 많다. 우선 MTS 컴포넌트는 반드시 DLL 이어야 한다. 또한, 다음의 몇 가지 요구사항을 반드시 만족시켜야 한다.

1. 컴포넌트는 반드시 표준 클래스 팩토리를 가져야 한다. 텔파이 4 의 MTS 자동화 위저드를 사용하면 이것이 자동으로 제공된다.
2. 모든 컴포넌트 인터페이스와 클래스는 반드시 타입 라이브러리에 기술되어야 한다. 이 역시 텔파이 4 의 위저드를 이용하면 자동으로 실행해 준다. 타입 라이브러리에 메소드와 프로퍼티를 추가할 때에는 타입 라이브러리 에디터를 사용한다. 타입 라이브러리의 정보는 MTS 탐색기(MTS Explorer)에 의해 설치된 컴포넌트에 대한 정보를 런타임에서 조회할 수 있다.
3. 컴포넌트는 반드시 표준 COM 마샬링을 사용하는 인터페이스만 export 해야 한다. 이 역시 위저드에 의해 자동으로 제공된다.
4. 텔파이의 MTS 는 사용자 정의 인터페이스에 대한 수동 마샬링을 지원하지 않는다. 모든 인터페이스는 COM 의 자동 마샬링을 이용하는 듀얼 인터페이스를 사용하여야 한다.
5. 컴포넌트는 반드시 DllRegisterServer 함수를 export 해야 하며, 이를 통해 CLSID, ProgID, 인터페이스와 타입 라이브러리의 Self-registration 을 지원해야 한다. 이것도 마찬가지로 위저드에 의해 제공되어야 한다.

## 리소스 pooling 과 관리

### ● Just-in-time 활성화

클라이언트 레퍼런스를 가지고 있는 동안 객체가 비활성화 되었다가 활성화 되는 것을 just-in-time 활성화라고 한다. 클라이언트 측에서 보면 클라이언트가 객체의 인스턴스를 생성한

시점에서 마지막으로 해제할 때까지 단지 하나의 인스턴스가 존재한다. 그러나, 실제로 객체는 여러 차례 활성화, 비활성화 될 수 있다. 객체가 비활성화 되면 MTS 는 데이터베이스 연결 등의 객체의 모든 리소스를 해제한다.

COM 객체가 MTS 환경의 한 파트로 생성되면, 여기에 해당되는 컨텍스트 객체(context object)도 같이 생성된다. 이 컨텍스트 객체는 하나 이상의 재활성화 사이클(reactivation cycle)에서 계속해서 존재하게 된다. MTS 는 객체 컨텍스트를 이용해서 비활성화된 동안 객체를 추적한다. 이런 컨텍스트 객체는 IObjectContext 인터페이스를 통해 접근할 수 있으며, 트랜잭션을 관리한다. 비활성화 상태에서 생성된 COM 객체는 클라이언트의 요구를 받으면 활성화된다.

MTS 객체는 다음의 경우에 비활성화 된다.

1. SetComplete 나 SetAbort 에 의해 객체가 비활성화를 요구하는 경우:

객체는 객체가 모든 작업을 끝내고, 클라이언트의 다음 호출이 있을 때 객체의 내부 상태(internal object state)를 저장할 필요가 없을 때, IObjectContext 인터페이스의 SetComplete 메소드를 호출한다. 또한, 작업을 정상적으로 끝내지 못하고, 내부 상태를 저장할 필요가 없을 때에는 SetAbort 를 호출한다. 이때 객체의 상태는 트랜잭션이 있기 전의 상태로 돌아간다. 가끔 객체가 상태(state)를 가지지 않도록 디자인 된 경우도 있는데, 이럴 때에는 모든 메소드에서 반환될 때 객체가 비활성화된다.

2. 트랜잭션이 commit 되거나 취소된 경우:

객체의 트랜잭션이 commit 되거나 취소된 경우 객체는 비활성화 된다. 이런 비활성화된 객체들에서 계속 존재하는 것은 트랜잭션의 외부에 있는 클라이언트로부터의 레퍼런스이다. 이들을 재활성화하는 호출은 다음 트랜잭션을 실행하도록 한다.

3. 마지막 클라이언트가 객체를 해제한 경우:

클라이언트가 객체를 해제하면, 객체는 비활성화되며 객체 컨텍스트도 해제된다.

## ● 리소스 pooling

MTS 는 비활성화된 동안 놓고 있는 시스템 리소스를 해제하고, 이렇게 해제된 리소스는 다른 서버 객체에 의해 사용될 수 있게 된다. 예를 들어, 데이터베이스 연결이 서버 객체에 의해 더 이상 사용되지 않으면, 다른 클라이언트에 의해 사용될 수 있다. 이런 것들을 리소스 pooling 이라고 한다.

데이터베이스에 연결을 하고, 끊는 작업은 꽤 시간이 걸리는 작업이다. MTS 는 리소스 디스펜서(resource dispenser)를 이용하여 새로운 데이터베이스 연결을 생성하기 보다는, 현재 존재



하고 있는 데이터베이스 연결을 재사용할 수 있도록 해준다. 리소스 디스펜서는 데이터베이스 연결 등의 리소스를 캐쉬에 저장하기 때문에, 하나의 패키지에 들어있는 컴포넌트 들은 리소스를 공유할 수 있다. 예를 들어, 하나의 어플리케이션에 데이터베이스 검색과 데이터베이스 업데이트 컴포넌트를 같이 가지고 있다고 하자. 이때 이 컴포넌트들을 하나의 패키지로 묶으면 데이터베이스 연결을 공유하게 할 수 있다.

- 객체 pooling

MTS 는 리소스 뿐만 아니라 객체를 관리하는 데에도 유용하게 쓸 수 있다. MTS 가 비활성화 메소드를 호출하고 나면, CanBePooled 메소드를 호출하는데 이 메소드를 통해 재사용을 위해 객체가 pooling 될 수 있는지 여부를 알 수 있다. CanBePooled 가 True 로 설정되면 비활성화시 객체를 파괴하기 보다, 객체 pool 로 객체를 이동한다. 객체 pool 의 객체들은 다른 클라이언트가 이를 요구하면 즉시 사용될 수 있다. 객체 pool 이 비어있을 때에만 MTS 가 새로운 객체 인스턴스를 생성한다.

그런데, 현재까지의 MTS 버전은 객체 pooling 과 recycling 을 지원하지 않는다. MTS 는 CanBePooled 메소드를 호출하지만 실제 pooling 은 일어나지 않는 것이다. 그렇지만 앞으로의 버전에서 이를 지원하게 될 것이다. 델파이 4 는 CanBePooled 를 False 로 초기화하기 때문에, 기본적으로 객체 pooling 이 지원되지 않는다. 앞으로 지원이 된다면 이를 True 로 초기화해서 사용해야 할 것이다.

- 객체 컨텍스트로의 접근

MTS 를 사용하는 COM 객체는 사용되기 전에 생성된다. COM 클라이언트는 COM 라이브러리 함수인 CoCreateInstance 를 호출하여 객체를 생성한다.

MTS 에서 동작하는 COM 객체는 반드시 컨텍스트 객체를 가져야 한다. 컨텍스트 객체는 MTS 에 의해 자동으로 구현되며, MTS 컴포넌트와 트랜잭션을 관리하게 된다. 컨텍스트 객체의 인터페이스는 IObjectContext 이다. 객체 컨텍스트의 대부분의 메소드에 접근하려면 TMtsAutoObject 객체의 IObjectContext 프로퍼티를 이용하거나, TMtsAutoObject 의 메소드를 직접 이용하면 된다.

- 리소스의 해제

개발자는 객체의 리소스를 해제할 의무가 있다. 보통 이럴 때 클라이언트의 요구를 서비스하

고 나서 SetComplete, SetAbort 메소드를 이용한다. 이들 메소드는 MTS 리소스 디스펜서에 의해 할당된 리소스를 해제한다. 동시에 개발자는 MTS 객체나 컨텍스트 객체를 포함한 다른 객체에 대한 레퍼런스를 비롯한, 다른 모든 리소스에 대한 레퍼런스를 해제해야 한다.

이런 메소드를 호출하지 않는 경우는 클라이언트 호출 사이에 상태(state)를 유지하고자 하는 경우이다.

## 리소스 디스펜서 (Resource dispensers)

리소스 디스펜서는 비영속적인(nondurable) 어플리케이션 컴포넌트의 상태(state)를 관리한다. SQL 서버와 같은 리소스 관리자와 비슷하지만, 영속성에 대한 보장을 하지 않는다는 점이 다르다. 델파이 4에서는 MTS에 대해서 2개의 리소스 디스펜서를 제공한다.

- BDE 리소스 디스펜서
- 공유 프로퍼티 관리자 (Shared Property Manager)

BDE 리소스 디스펜서는 표준 데이터베이스 인터페이스를 사용하는 MTS 컴포넌트에 대한 데이터베이스 연결의 pool을 관리한다. 즉, 객체들에 대한 데이터베이스 연결을 빠르면서 효과적으로 할당한다. 원격 MTS 데이터 모듈의 경우 가능한 연결들이 객체의 트랜잭션에 나열되며, 리소스 디스펜서는 자동으로 이들 연결을 재사용한다.

MTS 자동화 객체에 자동으로 객체 트랜잭션을 나열하고, 연결을 재사용하려면 uses 절에 BDEMTS 유닛을 추가한다.

## 기초 클라이언트(Base clients)와 MTS 컴포넌트

MTS 런타임 환경에서는 클라이언트와 객체 사이의 차이점을 이해하는 것이 중요하다. 클라이언트(기초 클라이언트)는 기본적으로는 MTS 하에서 돌아가는 것이 아니다. 기초 클라이언트는 MTS 객체의 주된 사용자이다. 전형적으로 클라이언트는 어플리케이션의 사용자 인터페이스를 제공하거나 최종 사용자의 요구를 MTS 서버 객체에 정의된 비즈니스 함수에 매핑한다. 클라이언트는 트랜잭션 지원이나 리소스 디스펜서 등의 MTS에서 지원하는 여러가지 혜택을 누리지 못한다.

## MTS 자동화 위저드의 이용

MTS 자동화 객체를 만들려면 다음과 같은 과정을 밟는다.

1. File|New 메뉴에서 ActiveX 탭을 선택한다.
2. MTS Automation Object 아이콘을 더블 클릭한다.
3. 자동화 객체의 이름을 적는다.
4. 쓰레딩 모델과 트랜잭션에 대한 옵션을 선택하고 OK 를 클릭한다.

이런 과정을 완료하면, 현재의 프로젝트에 자동화 객체의 정의를 포함한 새로운 유닛에 추가된다. 또한, 위저드는 타입 라이브러리 프로젝트를 추가하게 되는데, 여기에서 자동화 객체에서 사용하게 될 프로퍼티와 메소드를 노출시킨다. 이렇게 생성된 자동화 객체는 듀얼 인터페이스를 지원한다.

MTS 자동화 위저드는 기본적인 IObjectControl 인터페이스의 Activate, Deactivate, CanBePooled 메소드를 구현한다.

## MTS activities

MTS 는 activity 를 통해 concurrency 를 지원한다. 각각의 MTS 객체는 하나의 activity 에 속하며, 이들은 객체의 컨텍스트에 기록된다. 객체와 activity 의 관계는 변할 수 없다. Activity 에는 기초 클라이언트에서 생성된 MTS 객체와 이런 객체에 의해 생성된 MTS 객체와 그 자손들이 포함된다. 이런 객체 들은 하나 이상의 프로세스에 분산되며, 하나 이상의 컴퓨터에서 실행된다.

예를 들어, 의료 어플리케이션에서 여러 종류의 의료 데이터베이스에 레코드를 업데이트하고, 제거하는 MTS 객체가 있다고 하자. 이들은 각각 다른 객체에 의해 작동하게 된다. 이들 객체도 트랜잭션을 기록하기 위한 객체 등의 다른 객체를 사용할 수 있다. 결국 여러 개의 MTS 객체 들이 직간접적으로 기초 클라이언트의 아래에 놓이게 되며, 이들은 모두 동일한 activity 에 속한다.

MTS 는 각각의 activity 를 통해 실행되며, 어플리케이션 상태를 망가뜨릴 수 있는 여러가지 상황을 방지한다. 이런 형태는 결국 하나의 논리적인 쓰레드가 여러 객체 들이 분산되어 있음에도 잘 실행될 수 있도록 하는 것이다. 하나의 논리적 쓰레드를 가진 어플리케이션은 작성하기가 쉽다.

새로운 MTS 객체가 생성되면, 새로운 activity 가 생성된다. MTS 객체가 현재 존재하는 컨텍스트에서 생성된다면 새로운 객체는 같은 activity 의 멤버가 된다. 이럴 때 컨텍스트는 트랜잭션 컨텍스트 객체이거나 MTS 객체 컨텍스트일 수 있다.

MTS 는 하나의 activity 내에는 단지 하나의 논리적 실행 스레드 만을 허용한다. 이것은 객체들이 여러 프로세스에 분산될 수 있다는 것을 제외하면 COM apartment 와 유사하다. 기초 클라이언트가 activity 를 호출하면 첫번째 실행 스레드가 클라이언트로 돌아올 때까지 activity 내의 다른 모든 요구(다른 클라이언트 스레드의 요구 등)는 중지된다.

## 객체 레퍼런스(Object references)와 콜백

객체 레퍼런스를 넘겨줄 때에는 CoCreateInstance, IObjectContext.CreateInstance, ITransactionContext.CreateInstance 등의 객체 생성후 반환되는 레퍼런스를 이용하거나, QueryInterface 의 호출, 객체 레퍼런스를 얻기 위한 SafeRef 호출 등의 방법을 사용하게 된다. 이런 방법으로 획득한 객체 레퍼런스를 세이프 레퍼런스(safe reference)라고 한다. MTS 는 세이프 레퍼런스에 의해 호출되는 메소드는 적절한 컨텍스트에서 실행된다고 간주한다. 또한, 이런 호출은 MTS 런타임 환경을 이용한다. 이렇게 함으로써 MTS 가 컨텍스트 스위치를 관리하고, MTS 객체가 클라이언트 레퍼런스에 독립적으로 존재할 수 있게 된다.

### ● SafeRef 메소드의 이용

객체는 SafeRef 함수를 이용해서 레퍼런스를 얻고, 이를 컨텍스트의 외부로 안전하게 넘겨줄 수 있다. SafeRef 는 다음을 입력으로 받을 수 있다.

- 현재 객체가 다른 객체나 클라이언트에게 넘겨주기를 바라는 인터페이스 ID 의 레퍼런스 (RIID)
- 현재 객체의 인터페이스에 대한 레퍼런스

SafeRef 는 현재 객체의 컨텍스트 외부로 안전하게 넘길 수 있는 RIID 파라미터에 지정된 인터페이스의 포인터를 반환한다. 이때 nil 이 반환되면, 객체가 자신이 아닌 객체에서 세이프 레퍼런스를 요구했거나 RIID 파라미터에 의해 요구된 인터페이스가 구현되지 않은 경우이다.

MTS 객체가 셀프-레퍼런스(self-reference)를 클라이언트나 다른 객체에게 넘기길 원할 때(예를 들어 콜백을 사용할 경우) 언제나 SafeRef 를 먼저 호출하고, 이를 통해 반환된 레퍼런스를 넘겨준다. 객체는 셀프 포인터나 QueryInterface 의 내부 호출을 통해 얻은 셀프 레퍼런스를 클라이언트나 다른 객체에 넘겨서는 안된다. 이런 레퍼런스가 객체 컨텍스트 외부로 전달되면, 더 이상 유효한 레퍼런스가 되지 못한다.

- 콜백 (Callbacks)

객체는 클라이언트나 다른 MTS 컴포넌트에 대해 콜백을 만들 수 있다. 예를 들어 다른 객체를 생성한 객체를 가진 경우, 객체의 생성은 자신의 레퍼런스를 생성된 객체에게 넘겨주게 된다. 생성된 객체는 이 레퍼런스를 이용하여 자신을 생성한 객체를 호출한다.

콜백을 사용할 때에는 다음과 같은 제약이 있다.

- 기초 클라이언트나 다른 패키지로의 콜백은 클라이언트에 접근-레벨 보안(access-level security)을 필요로 한다. 또한, 클라이언트는 반드시 DCOM 서버이어야 한다.
- 방화벽(firewall)을 간섭할 경우 클라이언트로의 콜백이 중단된다.
- 콜백은 같은 트랜잭션이 될 수도 있고, 다른 트랜잭션이 될 수도 있으며 트랜잭션이 아닐 수도 있다.
- 객체의 생성에는 반드시 SafeRef 의 호출이 필요하며, 여기서 반환된 레퍼런스를 콜백을 위해 생성된 객체에 넘겨주어야 한다.

## MTS 객체의 MTS 패키지로의 설치

MTS 어플리케이션은 하나의 MTS 실행파일 인스턴스에서 동작하는 in-process MTS 자동화 객체들 또는 MTS 리모트 데이터 모듈의 그룹으로 구성되어 있다. 같은 프로세스에서 동작하는 COM 객체들의 그룹을 패키지라고 하는데, 하나의 기계는 여러 개의 다른 패키지에서 실행될 수 있으며, 각각의 패키지는 여러 개의 MTS 실행 파일에서 실행된다.

개발자는 어플리케이션 컴포넌트들을 하나의 프로세스에서 실행하기 위해 하나의 패키지로 그룹을 짓는다. 그러나 어떤 경우에는 컴포넌트들을 서로 다른 패키지에 분산시켜, 어플리케이션이 여러 개의 프로세스나 기계에서 실행되기를 원할 수 있다.

MTS 객체들을 하나의 패키지로 설치하려면 다음과 같이 한다.

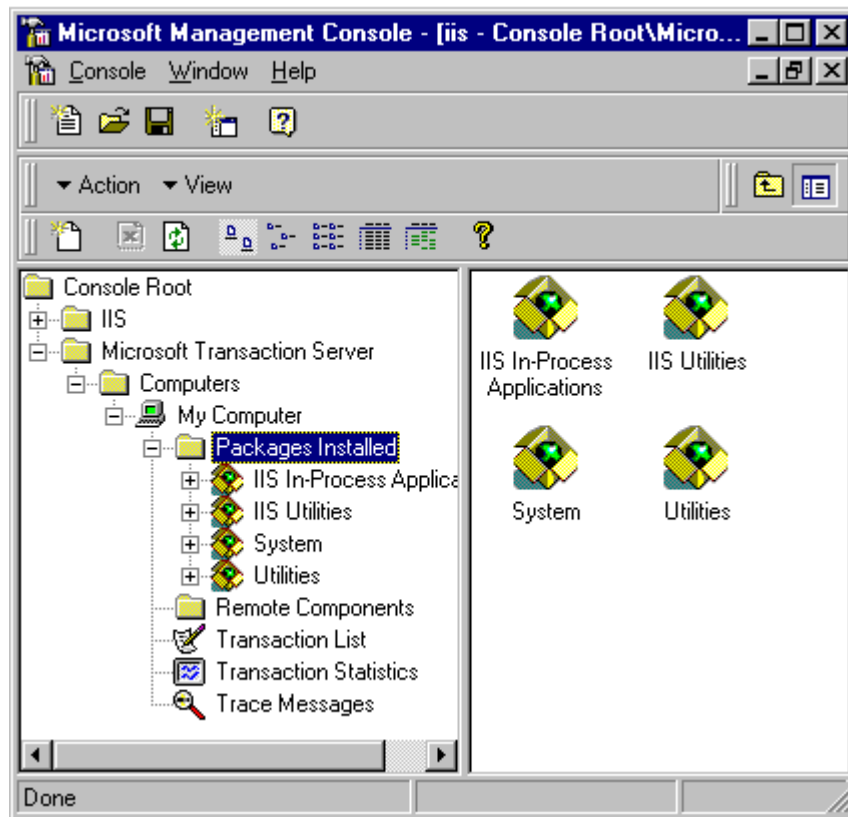
1. Run|Install MTS Objects 메뉴를 선택한다.
2. 설치될 CoClass 들을 체크한다.
3. MTS 카타로그를 refresh 하려면 OK 를 선택한다.
4. 현재 존재하는 패키지에 컴포넌트를 설치하려면 Into Existing Package 탭을 선택하고 컴포넌트를 설치할 패키지의 이름을 고른다. 그렇지 않으면, Into New Package 탭을 선택하고 새롭게 생성할 패키지의 이름을 적는다.

패키지는 여러 DLL 의 컴포넌트를 포함할 수 있으며, 하나의 DLL 의 컴포넌트를 여러 개의 패키지로 나누어 설치할 수도 있다. 그러나, 하나의 컴포넌트는 여러 패키지로 분산될 수 없다.

## MTS 객체들과 MTS 탐색기(Explorer)

일단 MTS 객체들을 MTS 런타임 환경에 설치하면, 이런 런타임 객체 들을 MTS 탐색기를 통해 관리할 수 있다. MTS 탐색기는 MTS 컴포넌트를 관리하고, 배포하는 기능을 담당하는 GUI 이다.

다음 그림은 실제 MTS 탐색기를 띄워서 패키지를 보여주는 화면이다.













MTS 탐색기의 역할은 다음과 같다.

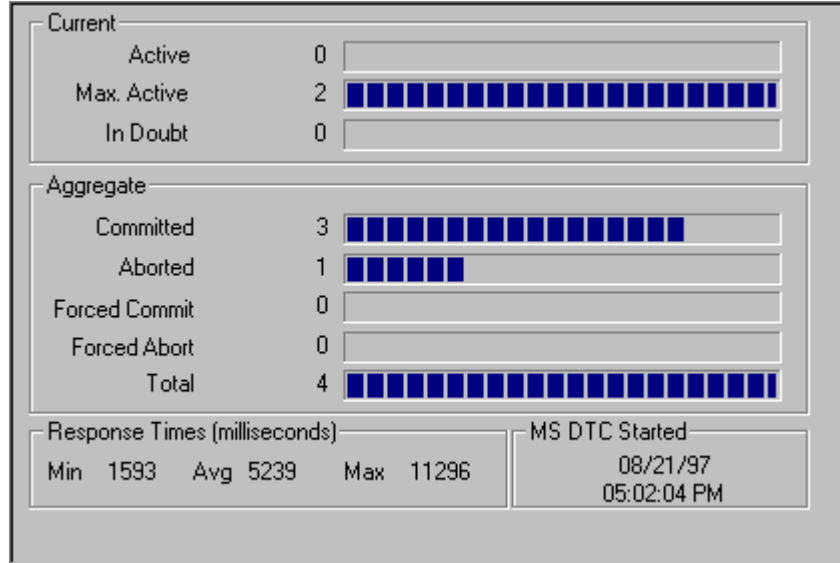
1. MTS 객체, 패키지, role 의 환경 설정
2. 패키지 내의 컴포넌트의 프로퍼티와 컴퓨터에 설치된 패키지를 보여준다.
3. 트랜잭션을 구성하는 MTS 컴포넌트 들에 대한 트랜잭션을 모니터, 관리한다.

4. 컴퓨터 사이에 패키지를 이동한다.
5. 원격 MTS 객체를 로컬 클라이언트에서 쓸 수 있게 한다.

MTS 탐색기는 개발자 뿐만 아니라, 시스템과 웹 관리자 등이 사용하여 패키지를 설치, 배포, 테스트, 관리할 수 있다. 개발자 들은 MTS 탐색기를 이용하여 컴포넌트를 패키지로 모으고, MTS 환경에서 컴포넌트를 분산시키고 이를 테스트하는데 사용하게 되며, 시스템이나 웹 관리자 들은 MTS 탐색기를 이용하여 컴포넌트와 패키지를 설치, 배포, 관리하는데 사용하게 된다. 다음과 같이 MTS 탐색기의 프로퍼티 윈도우(properties window)를 이용하면 패키지에 포함된 컴포넌트의 프로퍼티를 보거나 컴퓨터에 설치된 패키지를 살펴볼 수 있다.

Prog ID	Transaction	DLL	CLSID	Threading	Security
 Bank. Account	Required	C:\Progra...	{5BE6C9DB...	Apartment	Y
 Bank. Account. VC	Required	C:\Progra...	{04CF0B76...	Both	Y
 Bank. Account. VJ	Required	C:\Progra...	{9FAF8612...	Both	Y
 Bank. CreateTable	Requires new	C:\Progra...	{5BE6C9E1...	Apartment	Y
 Bank. GetReceipt	Supported	C:\Progra...	{5BE6C9DF...	Apartment	Y
 Bank. GetReceipt. VC	Requires new	C:\Progra...	{A81260B2...	Both	Y
 Bank. MoveMoney	Required	C:\Progra...	{5BE6C9DD...	Apartment	Y
 Bank. MoveMoney. VC	Required	C:\Progra...	{04CF0B7B...	Both	Y
 Bank. UpdateReceipt	Requires new	C:\Progra...	{5BE6C9E3...	Apartment	Y
 Bank. UpdateReceipt. VC	Requires new	C:\Progra...	{A81260B8...	Both	Y

그리고, 트랜잭션 통계 윈도우(Transaction Statistics window)를 이용하면 현재의 트랜잭션에 대한 요약된 통계를 다음과 같이 볼 수 있다.



MTS 탐색기 윈도우의 좌측 pane 에는 객체의 계층도를 표시하고, 우측 pane 에는 좌측 pane 에서 선택된 아이템을 표시하는 역할을 한다.

각 아이템에 대한 기본적인 정보는 아이템의 프로퍼티 시트(property sheet)를 통해 확인할 수 있는데, 예를 들어 컴퓨터 아이템에는 컴퓨터의 이름과 로그 파일의 위치와 업데이트 설정 등이 담겨 있고, 패키지 아이템에 대해서는 보안과 다른 여러가지 설정에 대한 정보를 담게 된다. 프로퍼티 시트를 보기 위해서는 아이템을 선택하고 Action 메뉴에서 Properties 명령을 선택하거나 오른쪽 버튼을 클릭하고 Properties 메뉴를 선택하면 된다.

MTS 의 런타임 환경 중에서 트랜잭션 처리에 대한 부분은 MS DTC(Distributed Transaction Coordinator)가 주로 담당하게 된다. MS DTC 는 MTS 가 사용하는 윈도우 NT 서비스이다.

MTS 탐색기에 대한 더 자세한 내용은 도움말을 참고하기 바란다.

## 정 리 (Summary)

이번 장에서는 MTS 가 등장하게 된 배경과 MTS 가 지원하는 여러가지 개념에 대한 총론적인 내용과 텔파이 4 에서 지원하는 MTS 관련 기능에 대해 전체적으로 알아 보았다.

다음 장에서는 이를 바탕으로 실제 MTS 패키지를 만들고, 이를 설치하여 실행하는 방법에 대해서 알아볼 것이다.



## 마이크로소프트 트랜잭션 서버의 이용 (II)

이번 장에서는 앞 장에서 설명한 내용을 바탕으로 MTS 를 지원하는 DCOM 객체를 제작하고, 이를 MTS 환경에서 설치하고 실행하는 방법에 대해서 실제 예를 통해 설명할 것이다. 텔파이 4 에서 제공하는 위저드의 사용법은 기존의 COM 에 대한 지원 방법과 유사하지만, 더욱 중요하고도 잘 알려지지 않은 것은 서버에서 어떻게 실제로 MTS 컴포넌트와 패키지를 설치할 것이며, 또한 이렇게 설치한 패키지를 클라이언트에서 사용할 수 있는지에 대한 정보이다.

MTS 를 사용한 실제 적용 방법에 대해서 설명하기 전에, 기본적인 환경과 정보에 대해 알 필요가 있다. MTS 를 잘 활용하기 위해서는 기본적으로 서버 컴퓨터에 MTS 서버가 설치되어 있어야 한다. 윈도우 NT 4.0 의 SP3 버전 이상에서는 IIS 와의 연동을 가능하게 해주는 MTS 서버 버전 2.0 을 사용할 수 있으며, 이를 설치하도록 권하고 싶다. 그리고, 기본적으로 클라이언트 역시 DCOM 환경에서 동작하게 되므로 윈도우 95 의 경우 윈도우 95 용 DCOM 을 설치해야 한다. 또한, 31 장에서 언급한 DCOM 환경 설정이 필요한 것은 물론이다. 이 밖에 더 자세한 MTS 의 설치에 대한 의문점, 업데이트된 파일과 기술적인 내용에 대한 정보는 <http://www.microsoft.com/com/mts-f.htm> 사이트를 방문하여 참고하기 바란다.

그러면, MTS 환경을 이용한 분산 환경의 세계로의 여행을 떠나보자.

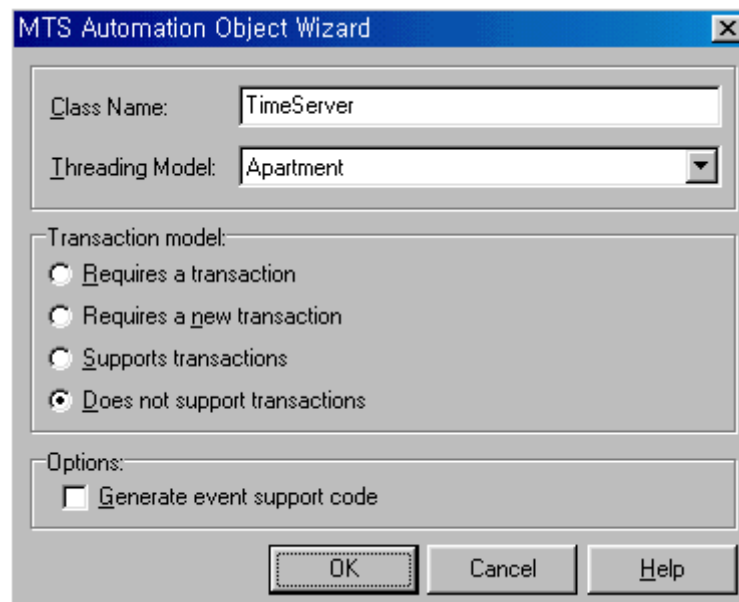
### 간단한 MTS 객체의 제작

MTS 를 이용한 간단한 서버 객체를 제작하고, 이를 이용한 클라이언트 어플리케이션을 작성하는 방법은 서버 객체와 클라이언트 어플리케이션을 제작하는 것보다 이들을 제대로 설치하는 과정이 훨씬 더 복잡하다. 사실 그렇게 어려운 내용이라고 하기는 어렵지만, 국내에 의외로 MTS 를 이용한 개발 사례가 많지 않고, 이를 활용하는 방법에 대해 언급한 문서가 없어서 필자도 이를 제대로 익히는데 많은 시간을 소비하였다.

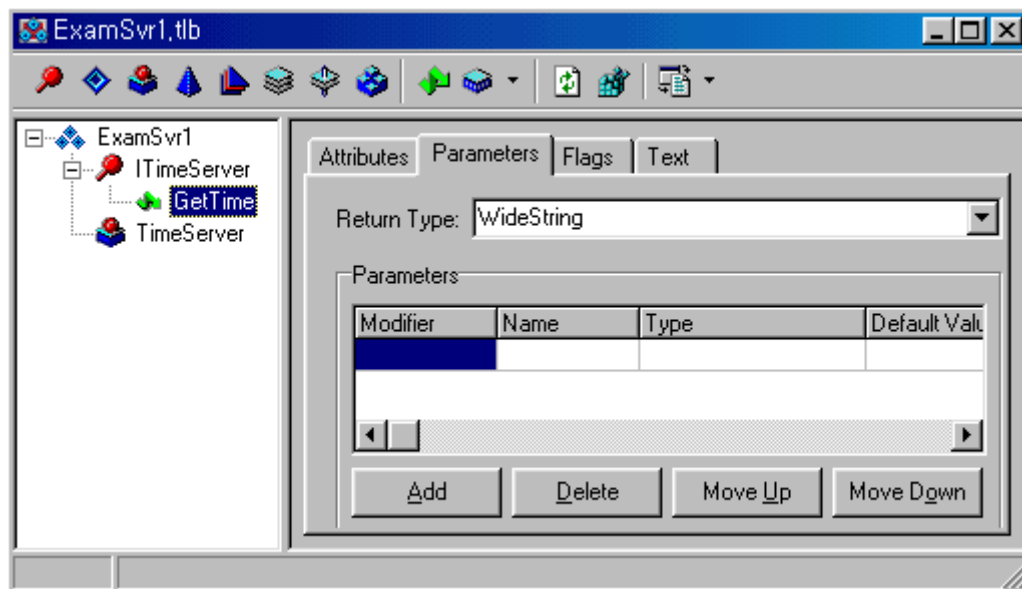
그러면, 실전에 들어가 간단한 MTS 객체를 제작하고 이를 서버와 클라이언트에 제대로 설치하는 방법에 대해서 알아보도록 하자.

먼저 서버 컴포넌트를 작성해야 하는데, 이 과정은 기본적으로 COM 서버 객체를 만드는 과정과 거의 동일하다. 차이점이 있다면 사용하는 위저드가 다르다는 점이다. MTS 객체를 제작하기 위해서는 File|New 메뉴를 선택하고 Multitier 탭에서 MTS Object 아이템을 더블 클릭하여 MTS 자동화 객체 위저드를 실행하도록 하자. 이렇게 하면 다음과 같은 대화상자가 나타나는데, 그 형태가 트랜잭션에 대한 옵션을 선택하는 것을 제외하고는 OLE 자동화 객체 위저드의 대화상자와 거의 같다는 것을 알 수 있을 것이다. 여기에서 다음과 같이 클래스 이름을 적어 넣고, 적당한 옵션을 선택한 후 'OK' 버튼을 클릭한다.

각 옵션의 대한 자세한 설명은 앞 장에서 다룬 바 있으므로 생략하도록 한다.



이렇게 하면, OLE 자동화 서버 위저드와 마찬가지로 타입 라이브러리 에디터가 나타날 것이다. 여기에서 우리가 작성할 MTS 객체 서버는 서버의 시간을 알아낼 수 있는 GetTime 메소드를 지원하는 ITimeServer 인터페이스를 다음과 같이 정의한다.



인터페이스 정의가 끝났으면, 타입 라이브러리 에디터를 종료하면 기본적인 MTS 객체를 생성하기 위한 코드가 자동으로 생성된다. 이들을 적당한 이름으로 저장하도록 하자. 생성된 소스 코드는 다음과 같을 것이다.

```

unit U_ExamSvr1;

interface

uses
    ActiveX, MtsObj, Mtx, ComObj, ExamSvr1_TLB;

type
    TTimeServer = class(TMtsAutoObject, ITimeServer)
    protected
        function GetTime: WideString; safecall;
    { Protected declarations }
    end;

implementation

uses ComServ;

function TTimeServer.GetTime: WideString;
begin
end;

initialization
    TAutoObjectFactory.Create(ComServer, TTimeServer, Class_TimeServer,
        ciMultilInstance, tmApartment);
end.

```

TTimeServer 클래스가 TAutoObject 클래스가 아닌 TMtsAutoObject 클래스에서 상속 받는다는 것을 제외하면 OLE 자동화 서버 위저드에 의해 만들어지는 코드와 완전히 동일한 코드라는 것을 쉽게 알 수 있을 것이다. 그러므로, 필자가 앞에서 소개한 여러 장들의 테크닉 들을 거의 그대로 사용할 수 있다.

구현 부분인 GetTime 메소드만 다음과 같이 구현하여 서버의 시간을 GetTime 메소드를 호출하는 클라이언트에게 문자열 형태로 넘겨주도록 하자. 여기서 TimeToStr 함수를 사용하기 위해 uses 절에 SysUtils.pas 유닛을 추가하도록 한다.

```

function TTimeServer.GetTime: WideString;
begin

```

```
Result := TimeToStr(Now);  
end;
```

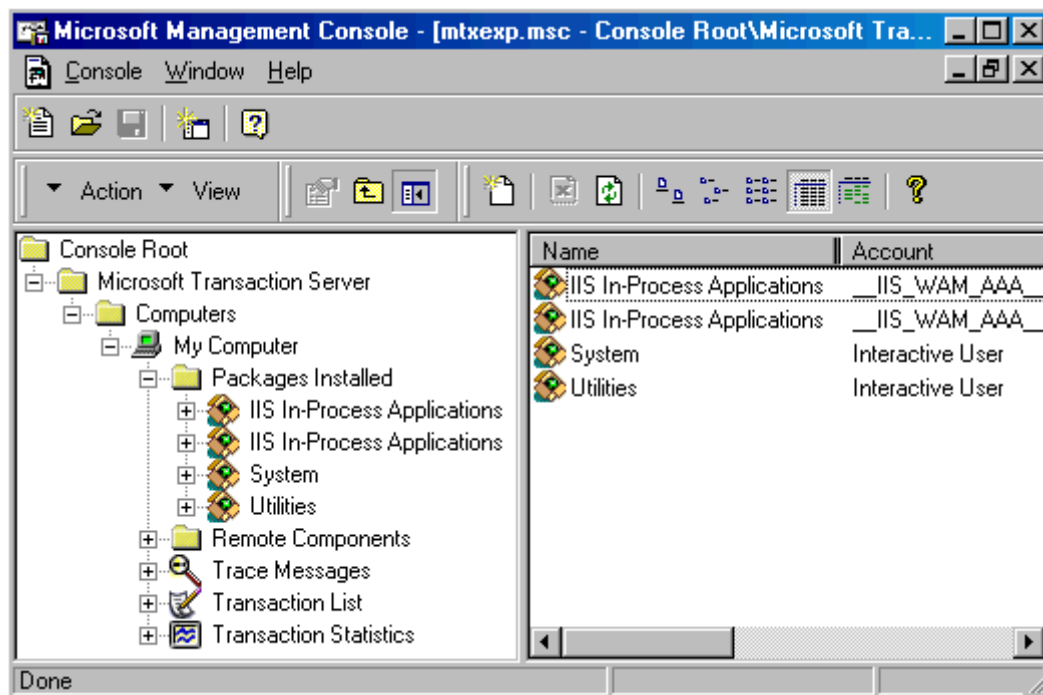
이것으로 간단한 MTS 객체는 완성하였다. 이제 이 프로젝트를 컴파일하면 DLL 파일이 생성될 것이다. 앞 장에서도 설명한 바 있지만, MTS 는 in-process COM 서버를 효과적으로 관리하는 환경이다. DCOM 의 out-of-process 서버를 사용할 때 가질 수 있는 수행 성능의 저하와 리소스 관리의 문제점 등을 효과적으로 관리하도록 해주는 환경으로 MTS 를 이해하면 거의 틀림이 없다.

이렇게 생성된 DLL 을 Run|Register ActiveX Server 메뉴를 이용해 등록하면 개발한 컴퓨터의 in-process COM 서버로 등록할 수 있다. 그렇지만, MTS 환경을 DCOM 을 이용하여 사용하기 위해서는 절대로 이 메뉴를 선택하면 안된다. 그렇게 할 경우 서버 컴퓨터의 in-process DLL 이 실행되는 것이 아니라, 설치한 컴퓨터의 DLL 이 실행되어 버린다.

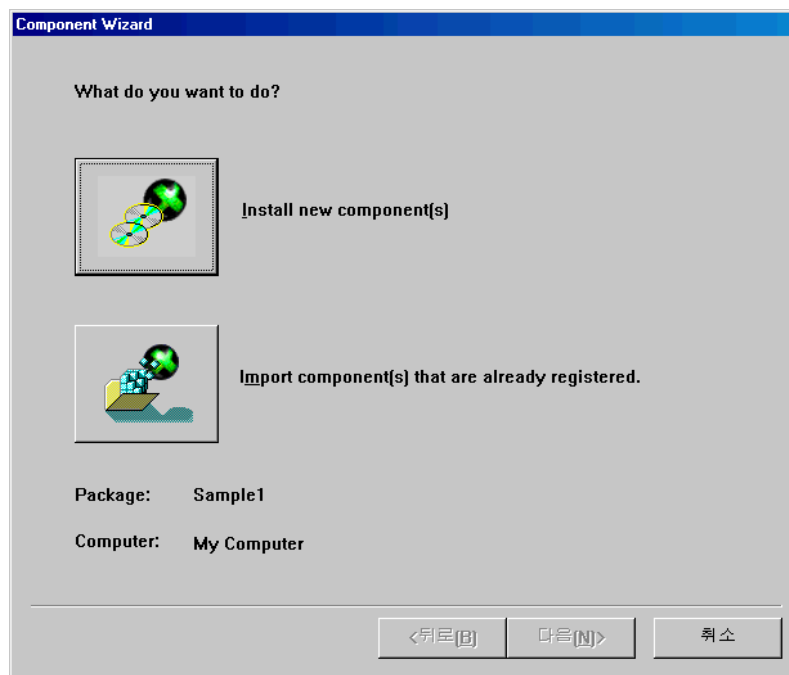
만약, 개발한 컴퓨터가 윈도우 NT 와 MTS 서버를 설치한 환경을 가지고 있으면 Run|Install MTS Objects... 메뉴를 이용하여 MTS 객체를 설치할 수 있지만 필자를 비롯한 대부분의 개발자는 서버에 수시로 접속할 수 있는 클라이언트 개발 환경을 가지고 있을 것으로 사료되므로, 앞 장에서 간단히 설명한 Run|Install MTS Objects... 메뉴를 이용한 객체의 설치방법은 커다란 실효성을 가지지 못할 것이다. 그러므로, 여기에서는 이렇게 만들어진 MTS 객체 DLL 을 서버 컴퓨터에 설치하는 방법을 보다 근본적인 방법을 이용하여 설명하도록 하겠다.

먼저 컴파일을 통해 생성된 DLL 파일을 서버 컴퓨터의 적당한 디렉토리로 복사한다. 필자의 경우 윈도우 NT 서버가 서버 컴퓨터의 D 드라이브에 깔려 있는데, 테스트를 위해 D:\Wtemp 디렉토리로 MTS 객체 서버 DLL 을 복사하였다. 물론 클라이언트 컴퓨터에 그대로 두고 이 위치에 있는 DLL 서버를 패키지로 등록하여 사용할 수도 있지만, 기본적으로는 여러 컴퓨터가 동시에 사용할 DLL 서버이므로 서버 컴퓨터의 적당한 위치로 옮겨서 사용하는 것이 더 합리적일 것이다. 경우에 따라서는 여러 개의 클라이언트에 MTS 컴포넌트를 분산시켜 위치시킨 후 이를 사용하게 할 수도 있다.

이렇게 적당한 위치에 DLL 파일을 배치했으면, 서버 컴퓨터에서 트랜잭션 서버 탐색기(Transaction Server Explorer)를 실행하도록 한다. MTS 탐색기의 좌측 화면에서 'Package Installed' 폴더를 선택하면 다음과 같이 이미 설치된 패키지들을 볼 수 있을 것이다. MTS 2.0 을 설치한 경우 IIS 에 대한 패키지들도 설치되어 있을 것이다. 참고로 여기서 보여주는 화면은 필자의 서버 컴퓨터에 설치된 윈도우 NT 5.0 베타 1 Korean locale 환경에서 실행된 것이기 때문에, 일반적인 윈도우 NT 4.0 정식 버전에서의 실행 모습과는 다소 다르게 보일 수도 있음을 미리 밝혀둔다.



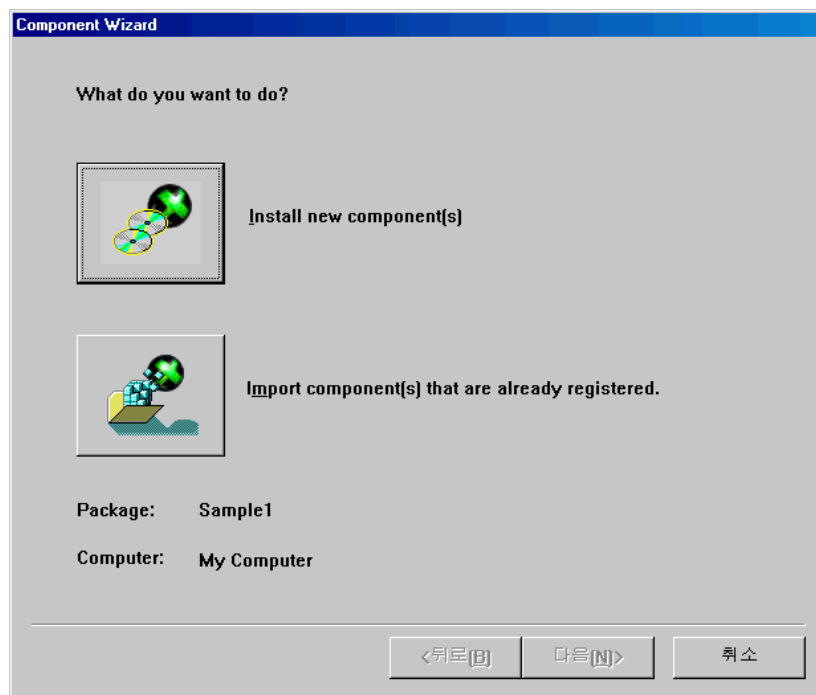
여기에 새롭게 설치할 패키지를 추가해야 한다. 툴바에서 'Create a new object' 버튼을 클릭하거나, 폴더에서 오른쪽 버튼을 클릭하고 New Package 메뉴를 선택하면 다음과 같이 패키지를 새롭게 작성할 것인지 아니면 작성된 패키지를 설치하는 것인지를 물어온다. 여기에서 우리들은 MTS 객체 컴포넌트를 작성한 것이지 패키지를 작성한 것이 아니므로 'Create an empty package' 버튼을 클릭하도록 한다.



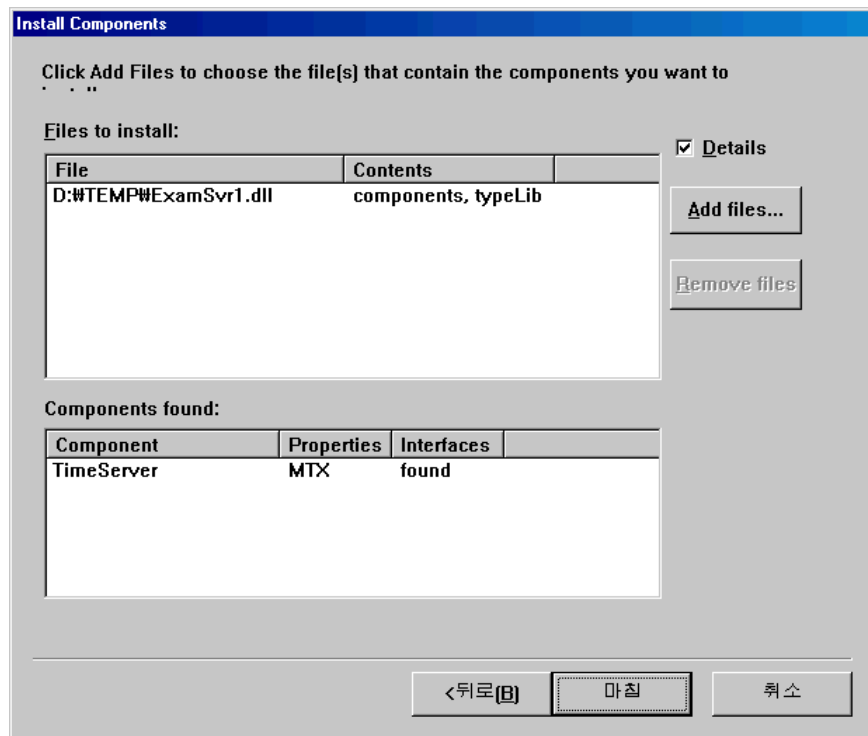
이렇게 하면 새로운 패키지의 이름을 입력하라는 대화상자가 나타나는데, 여기에서 적당한 패키지의 이름을 입력하도록 하자. 필자의 경우는 Sample1 이라는 이름을 부여하였다. 그리고, ‘다음’ 버튼을 클릭하면 이 패키지를 사용하게 될 사용자 계정을 선택할 수 있게 되는데, 여기에서 ‘Interactive user’를 선택하면 현재 접속한 사용자 들이 접속할 수 있도록 하는 것이며 ‘This user’를 선택한 경우에는 사용할 사용자를 윈도우 NT 도메인 유저 리스트를 이용하여 선택하고 이들에게 패스워드를 사용하도록 패스워드를 부여할 수 있다. 특수한 보안이 유지되어야 하는 경우를 제외하고는 ‘Interactive user’를 선택하도록 한다. 그리고, ‘마침’ 버튼을 클릭하면 MTS 탐색기 화면에 Sample1 이라는 새로운 패키지가 설치된 것을 확인할 수 있을 것이다.

이제는 이 패키지에 우리가 작성한 MTS 객체 컴포넌트를 설치할 차례이다. MTS 탐색기의 좌측 pane 에서 Sample1 패키지의 ‘Components’ 폴더를 선택하고 툴바에서 ‘Create a new object’ 버튼을 클릭하거나 오른쪽 버튼을 클릭하고 팝업 메뉴에서 New|Component 메뉴를 선택하여 컴포넌트를 추가한다.

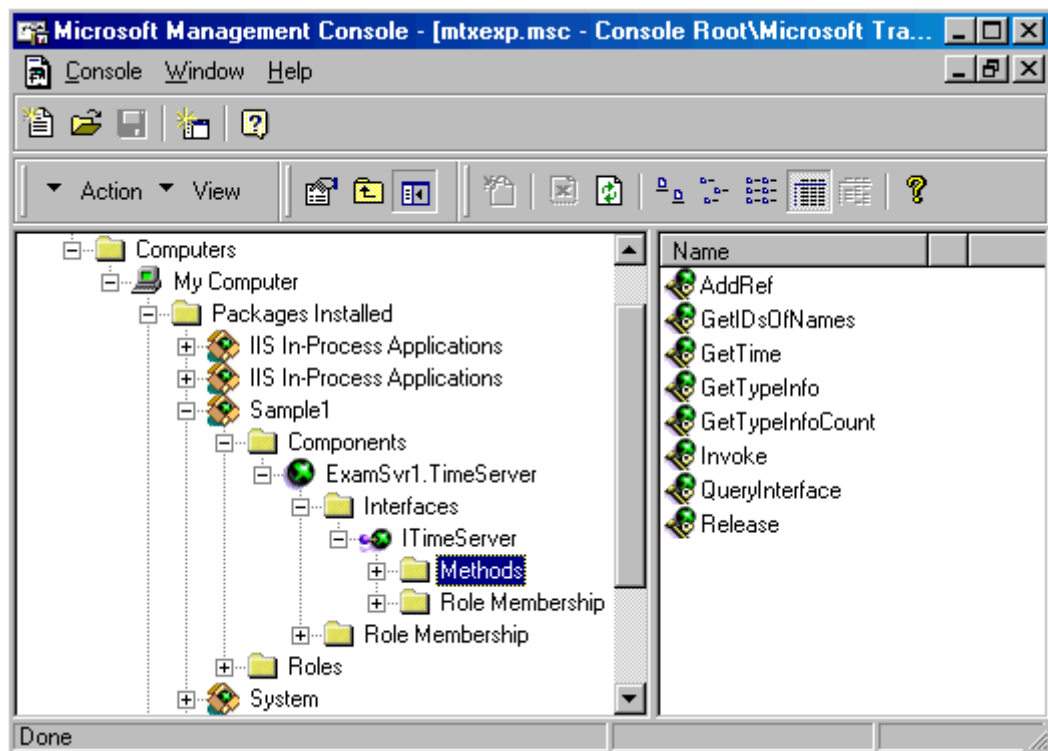
이렇게 하면 다음과 같이 이미 등록된 컴포넌트 중에서 패키지에 등록할 컴포넌트를 고르겠는지, 아니면 새로운 컴포넌트를 선택하는 대화상자가 나타날 것이다. 여기서는 새롭게 파일을 복사해서 등록하는 것이므로 ‘Install new component(s)’ 버튼을 클릭한다.



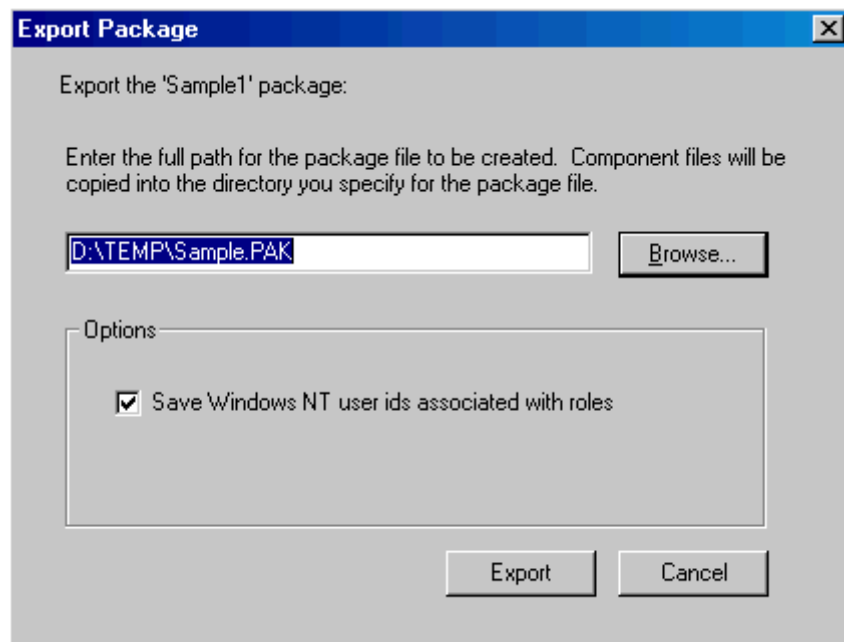
이 버튼을 클릭하면 서버 객체 파일을 추가하는 대화상자가 다음과 같이 나타나게 되는데, 여기에서 우리가 제작한 MTS 컴포넌트 DLL 파일을 ‘Add files’ 버튼을 클릭하여 추가하도록 한다.



여기서 마침 버튼을 클릭하면 패키지에 우리가 작성한 서버 객체가 설치된다.  
다음 화면은 설치된 MTS 컴포넌트를 MTS 탐색기를 이용하여 브라우징한 모습이다.



이제 컴포넌트 설치가 끝났으니, 클라이언트에서 사용할 수 있도록 설정해야 한다. 이렇게 서버에 설치한 MTS 패키지를 클라이언트에서 DCOM 을 이용하여 사용할 수 있게 하기 위해서는 클라이언트로 배포할 실행파일을 MTS 서버가 생성하도록 해야 한다. 이를 위해서 배포할 패키지(여기서는 Sample1)를 선택하고 오른쪽 버튼을 클릭하면 나타나는 팝업 메뉴에서 Export... 메뉴를 선택한다. 이렇게 하면 다음과 같이 패키지 설치 파일을 생성할 패스와 이름을 입력하라는 대화상자가 나타날 것이다. 여기에 클라이언트 등이 접근할 수 있는 공유된 디렉토리를 패스로 지정하고, 적절한 설치 파일 이름을 다음과 같이 적어 넣는다.

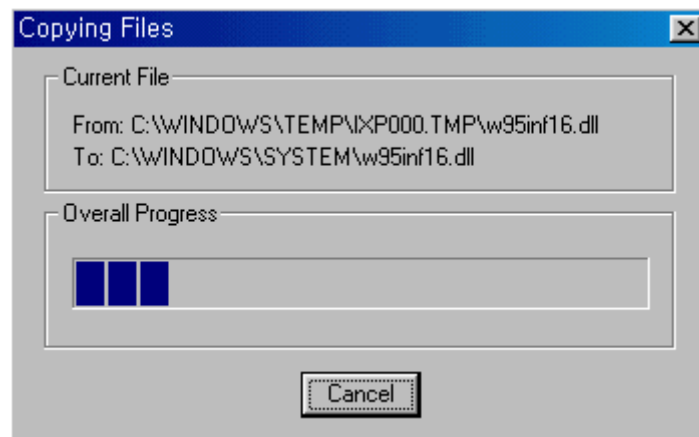


Export 버튼을 클릭하면 해당 디렉토리에 패키지 파일과 함께 패키지 파일의 이름과 동일한 이름을 가진 실행 파일이 담겨 있는 clients 라는 서브 디렉토리가 생성된다.

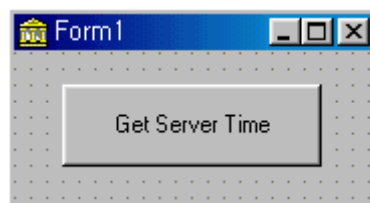
클라이언트에서 MTS 객체를 실행할 수 있게 하려면 이 clients 서브 디렉토리에 접근하여 이 실행파일(여기서는 sample.exe)을 실행해야 한다. 이렇게 클라이언트 설치 실행파일을 클라이언트에서 실행하면 DCOM 에 의해서 요구되는 프록시-스텝 DLL 들과 타입 라이브러리 파일이 클라이언트 컴퓨터에 복사되고, 클라이언트 시스템의 레지스트리 정보가 여기에 맞도록 업데이트 된다. 이 과정을 통해 클라이언트 어플리케이션이 서버 패키지를 사용할 수 있도록 설저오디는 것이다.

다음 화면은 이 실행파일을 클라이언트에서 실행할 때 나타나는 화면이다. 여기서는 이미 설치한 패키지를 다시 설치하는 화면을 잡았기 때문에 처음 설치하는 화면과는 다소 차이가 있을 것이다.





이것으로 MTS 서버 컴포넌트를 제작하고, 이를 사용할 준비가 모두 끝났다. 이제 이를 실제로 사용하는 클라이언트 어플리케이션을 작성해보자. 델파이에서 New Application 메뉴를 선택하여 새로운 메뉴를 선택하고 폼에 다음과 같이 버튼을 하나 없도록 하자.



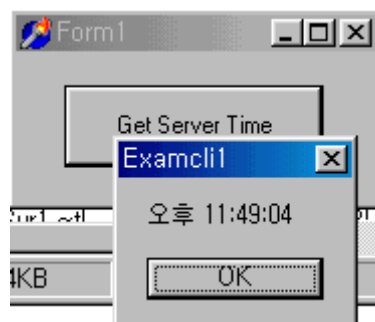
그리고, uses 절에 early 바인딩을 사용할 수 있도록 하기 위해 서버의 타입 라이브러리에 대한 파스칼 유닛(여기서는 ExamSvr1\_TLB.pas)을 추가하도록 한다. 물론 late 바인딩을 이용할 경우에는 이를 추가하는 대신 ComObj.pas 유닛을 추가하고 CreateOLEObject 함수를 이용하면 된다.

마지막으로 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성하면 클라이언트 어플리케이션은 완성된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    TimeServer: ITimeServer;
    Str: WideString;
begin
    TimeServer := CoTimeServer.Create;
    ShowMessage(TimeServer.GetTime);
end;
```

클라이언트 어플리케이션을 실행하고, 버튼을 클릭하여 서버의 시간을 알아보도록 하자.

아마도 첫번째 실행을 할 때 다소의 시간이 걸리겠지만, 이후에는 빠르게 동작하면서 서버의 시간을 알려줄 것이다. 다음은 클라이언트 어플리케이션의 실행 화면이다.



여기서 꼭 짚고 넘어가야 할 부분이 있다.

안타깝게도 필자가 시험해본 결과로는 이렇게 MTS 탐색기를 이용하여 export 한 실행 파일을 실행하여 클라이언트에 MTS 패키지에 대한 정보를 등록하면 윈도우 95 에서는 큰 문제가 없었지만, 윈도우 98 에서는 시스템에 다소 간의 문제가 발생하고 그 다음에 윈도우 98 을 재실행하는 시점부터 DCOM 이 제대로 동작하지 않는 등의 알 수 없는 버그가 존재하였다. 이는 MTS 가 만들어 내는 실행 파일과 윈도우 98 시스템에 호환성이 충분히 고려되지 않았기 때문으로 생각된다.

그러므로, 윈도우 95 를 사용할 때에만 이런 방법을 사용해야 하며, 윈도우 98 을 사용하면 번거롭더라도 제작한 MTS 패키지 컴포넌트에 대한 타입 라이브러리를 직접 클라이언트에서 등록하도록 해야 한다. 아마도, MTS 의 새로운 버전이 나오면 이 문제가 해결될 것으로 생각된다.

## MTS 데이터 모듈의 활용

이번에는 데이터 모듈을 이용하여 델파이의 데이터 접근 컴포넌트를 MTS 환경에서 활용하는 방법에 대해서 알아보도록 하자. 기본적인 예제를 작성하기 전에 데이터 모듈을 활용한 데이터베이스 어플리케이션의 작성 요령에 대해서 먼저 간단히 알아보도록 하자.

보통 기본적인 데이터베이스 어플리케이션을 작성할 때에는 TTable, TQuery 등의 데이터 세트 컴포넌트를 폼에 추가하고, 이와 함께 TDataSource 와 각종 데이터 컨트롤을 연결하여 데이터를 데이터베이스에서 불러와서 보여주고, 데이터 컨트롤에서 변경한 내용을 데이터베이스에 접근하는 형태로 어플리케이션을 작성한다.

그렇지만, 보다 확장성이 좋고 견고한 데이터베이스 어플리케이션을 작성할 때에는 OOP 의 캡슐화 개념을 이용하여 모든 데이터베이스에 관련한 작업을 데이터 모듈로 일원화하여 관리하는 것이 좋다. 그것도 단순히 데이터 모듈을 하나 추가하고, 폼 대신에 데이터 세트 컴포넌트와 데이터 소스 컴포넌트 들을 한 군데 모아 놓고 호출하는 형태로 작업을 하는 것

이 아니라, 데이터 모듈에서 필요로 하는 데이터베이스 작업을 추상화한 뒤에 각각의 데이터베이스 작업을 하나의 메소드로 정의하여 데이터 모듈의 메소드를 호출하면 내부적인 데이터베이스 작업이 이루어지도록 하는 것이 좋다.

이를 달리 해석하면, 데이터 모듈이 데이터베이스 작업을 캡슐화하는 것이다.

MTS 를 활용하여 데이터베이스 작업을 진행하기 위해서는 이러한 개념으로 타입 라이브러리 에디터에서 데이터 모듈에서 수행할 작업들을 인터페이스의 메소드로 추상화하여 정의하고, 이들을 실제로 구현한 뒤에 클라이언트는 인터페이스의 메소드를 호출하여 데이터베이스 작업을 진행하는 구조로 작성해야 한다.

그럼 이런 기본적인 개념에 입각하여 텔파이의 DBDEMOS 앨리어스의 biolife.db 데이터베이스 파일의 데이터를 MTS 환경에서 불러오고, 변경된 내용을 업데이트할 수 있는 서버를 작성해보자.

먼저, File|New 메뉴를 선택하고 ActiveX 페이지에서 ActiveX Library 아이템을 더블 클릭하여 새로운 액티브 X in-process DLL 파일 프로젝트를 시작한다. 여기에 File|New 메뉴의 Multitier 페이지에서 MTS Data Module 아이템을 더블 클릭하여 MTS 데이터 모듈 위저드를 시작한다. 이 위저드의 대화 상자는 MTS 자동화 객체 위저드와 완전히 동일하므로 자세한 설명은 생략한다. 여기에서 적당한 데이터 모듈 클래스의 이름을 적어야 하는데, 필자는 SampleDM 이라는 이름을 사용하였다. OK 버튼을 클릭하면 타입 라이브러리 에디터가 나타날 것이다. 여기에서 클라이언트 들이 사용할 메소드를 먼저 정의해야 한다. 이번에 작성할 예제에서는 데이터베이스 파일의 데이터를 클라이언트에서 얻을 때 사용할 GetData 메소드와 클라이언트에서 업데이트한 데이터를 데이터베이스 파일에 적용할 때 사용할 ApplyUpdates 메소드를 정의하도록 한다.

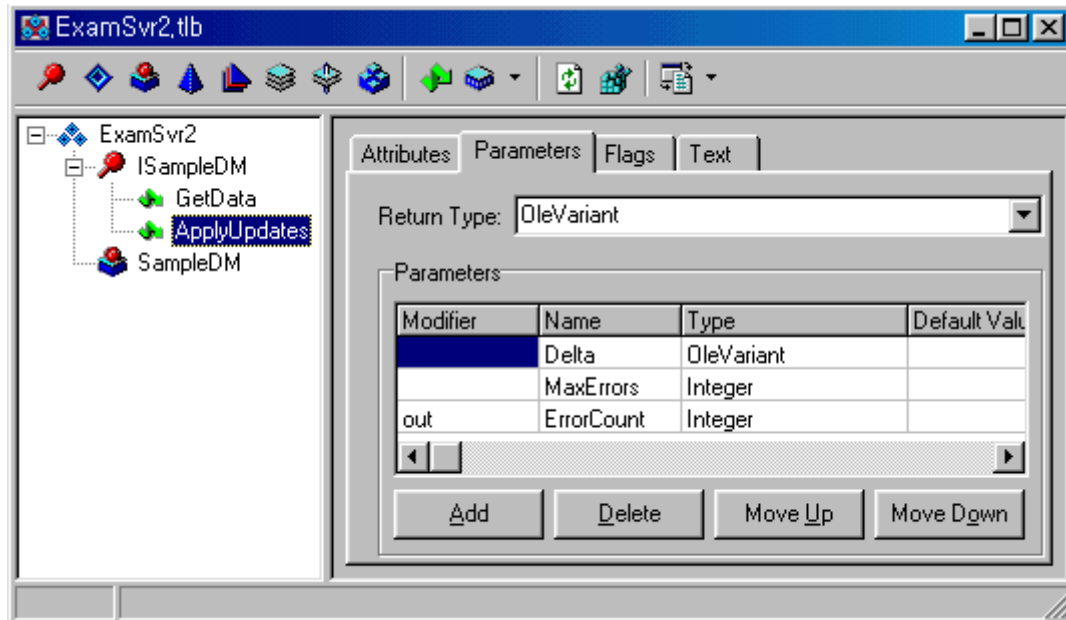
GetData 메소드는 파라미터 없이 호출하면 데이터를 반환해야 하므로, Return Type 으로 OleVariant 를 선택한다. 물론 데이터 모듈에 있는 특정 데이터 필드를 반환할 경우 필드의 데이터 형에 따라 적당한 데이터 형을 선택하면 될 것이다. 이 예제에서 OleVariant 를 선택한 이유는 간단히 TProvider 컴포넌트를 이용하여 데이터베이스 파일의 데이터를 반환할 것이기 때문이다. 그렇지만, 보다 복잡한 데이터베이스 어플리케이션을 작성할 때에는 구체적인 목적을 가진 메소드를 정의하여 해당 데이터 형을 반환하도록 하는 것이 좋다.

예를 들어, 신상정보 데이터베이스에서 특정 ID 를 가진 사람의 이름을 반환하는 메소드를 정의한다고 할 경우에 다음과 같이 메소드를 정의하는 것이 좋을 것이다.

```
GetName(ID: WideString): WideString;
```

이 메소드를 나중에 구현할 때에는 해당 테이블에서 파라미터로 넘어온 ID 값을 이용하여 이름을 알 수 있도록 쿼리를 하거나, 테이블 컴포넌트의 Find 메소드를 이용하면 될 것이다. ApplyUpdates 메소드는 클라이언트에서 변경한 내용을 데이터베이스 파일에 적용할 때 사용하도록 정의하는데, 여기서는 TProvider 컴포넌트의 ApplyUpdates 메소드를 그대로 사

용할 것이기 때문에, 이 메소드의 파라미터와 반환값을 그대로 사용하면 된다.  
이렇게 정의한 타입 라이브러리의 모습은 다음과 같다.



타입 라이브러리 에디터를 닫으면 아마도 다음과 같은 뼈대 코드가 완성되어 있을 것이다.  
이 내용은 데이터베이스와 관련된 유닛들이 uses 절에 추가된 것을 제외하면, 기본적으로  
MTS 자동화 객체 위저드가 생성하는 것과 동일하므로 자세한 설명은 생략한다.

```
unit U_ExamSvr2;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
ComServ, ComObj, VCLCom, StdVcl, BdeProv, BdeMts, DataBkr, DBClient,  
MtsRdm, Mtx, ExamSvr2_TLB, Db, DBTables;
```

```
type
```

```
TSampleDM = class(TMtsDataModule, ISampleDM)
```

```
private
```

```
    { Private declarations }
```

```
public
```

```
    { Public declarations }
```

```
protected
```

```

function ApplyUpdates(Delta: OleVariant; MaxErrors: Integer;
    out ErrorCount: Integer): OleVariant; safecall;
function GetData: OleVariant; safecall;
end;

var
    SampleDM: TSampleDM;

implementation

{$R *.DFM}

function TSampleDM.ApplyUpdates(Delta: OleVariant; MaxErrors: Integer;
    out ErrorCount: Integer): OleVariant;
begin
end;

function TSampleDM.GetData: OleVariant;
begin
end;

initialization
    TComponentFactory.Create(ComServer, TSampleDM,
        Class_SampleDM, ciMultiInstance, tmApartment);
end.

```

이제 사용할 데이터 접근 컴포넌트 들을 데이터 모듈에 추가하고 이들의 프로퍼티를 설정하도록 하자. 여기서는 간단하게 테이블과 TProvider 컴포넌트로 구현할 것이므로 이들 컴포넌트를 각각 하나씩 데이터 모듈에 추가한다.

그리고, Table1 컴포넌트의 Database 프로퍼티를 'DBDEMOS', TableName 프로퍼티를 'biolife.db'로 설정하고, Provider1 컴포넌트의 DataSet 프로퍼티는 Table1 으로 설정한다. 기본적인 프로퍼티 설정이 끝났으면 다음과 같이 간단하게 인터페이스에서 정의한 2 개의 메소드를 구현한다.

```

function TSampleDM.GetData: OleVariant;
begin

```

```

    Result := Provider1.Data;

    SetComplete;
end;

function TSampleDM.ApplyUpdates(Delta: OleVariant; MaxErrors: Integer;
    out ErrorCount: Integer): OleVariant;
begin
    Result := Provider1.ApplyUpdates(Delta, MaxErrors, ErrorCount);
    SetComplete;
end;

```

주의할 점은 각 메소드의 마지막에 SetComplete 메소드를 호출한다는 점이다. 이 메소드는 MTS 자동화 객체에서도 사용할 수 있는데, 그 의미는 현재 작업한 메소드가 완료되었음을 선언하고, MTS 데이터 모듈의 경우 트랜잭션의 중간에 있다면 이 메소드를 호출함으로써 commit 을 하게 된다. MTS 는 클라이언트가 MTS 데이터 모듈의 메소드를 호출할 때 자동으로 트랜잭션을 시작하기 때문에 이 메소드의 호출이 필요한 것이다.

만약 현재까지의 작업이 잘못 되어서 이를 적용하고 싶지 않은 경우에는 SetAbort 나 DisableCommit 메소드를 이용하면 된다.

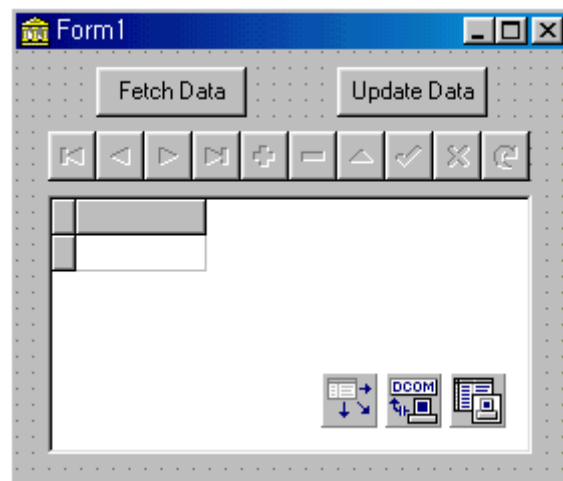
이것으로 간단한 MTS 데이터 모듈 객체의 구현이 끝났다. 컴파일을 하면 DLL 파일이 생성되는데, MTS 를 활용하기 위해서는 앞서 설명한 MTS 자동화 객체와 마찬가지로 현재 작성한 컴포넌트를 MTS 패키지에 설치해야 한다.

패키지를 새로 만들어도 되고, 앞서 작성한 패키지에 이 컴포넌트만 새로 추가해도 된다. 여기에 대해서는 이미 앞에서 충분히 설명하였으므로 자세한 설명은 생략하도록 하겠다.

어쨌든 MTS 를 잘 활용하기 위해서는 서버에서 MTS 탐색기를 이용하여 패키지와 컴포넌트를 등록하고 이들의 설정 값들을 잘 관리하는 것이 핵심이라는 것을 다시 한번 강조하고 싶다. 컴포넌트를 패키지에 등록했으면, 클라이언트에서 이들을 사용할 수 있도록 export 하도록 한다. 그리고, 이 과정에서 생성된 clients 서브 디렉토리의 실행 파일을 MTS 클라이언트를 설치하려는 컴퓨터에서 실행하여 기본적인 설정을 마쳐야 클라이언트 어플리케이션을 동작시킬 수 있다.

여기까지의 과정은 이미 상세하게 설명한 바 있으므로, 독자들이 모두 쉽게 설치했을 것으로 믿고 이번에 작성한 서버를 활용하는 클라이언트 어플리케이션의 제작에 들어가도록 하자.

텔파이에서 새로운 어플리케이션을 시작하고 다음과 같이 폼에 TDBGrid, TDBNavigator, TClientDataSet, TDataSource 와 TDCOMConnection 컴포넌트를 각각 하나씩 넣고, 버튼을 2 개 추가한다.



그리고, DBNavigator1 과 DBGrid1 의 DataSource 프로퍼티는 DataSource1, DataSource1 의 DataSet 프로퍼티는 ClientDataSet1, ClientDataSet1 의 RemoteServer 프로퍼티는 DCOMConnection1 으로 각각 설정한다.

그리고 가장 중요한 DCOMConnection1 의 ServerName 프로퍼티는 오브젝트 인스펙터에서 드롭 다운 리스트의 형태로 현재 등록된 데이터 모듈 서버를 고를 수 있게 되어 있는데, 여기에서 사용할 서버를 선택한다. 앞서 작성한 서버의 예를 들면 'ExamSvr2.SampleDM' 이 된다. 이렇게 서버를 선택하면 자동으로 ServerGUID 프로퍼티는 해당 서버의 GUID 로 설정된다.

기본적인 설정은 끝났고, 'Fetch Data' 버튼을 클릭하면 원격 서버의 데이터를 DBGrid 에 표시하고, 'Update Data' 버튼을 클릭하면 변경된 데이터를 원격 서버의 데이터베이스 서버에 적용시키도록 코딩하면 된다. 이를 위해서는 서버의 인터페이스 메소드인 GetData 와 ApplyUpdates 메소드를 이용하면 간단하게 구현할 수 있다.

먼저 데이터를 가져오는 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    DCOMConnection1.Connected := True;
    ClientDataset1.Data := DCOMConnection1.AppServer.GetData;
    DCOMConnection1.Connected := False;
end;
```

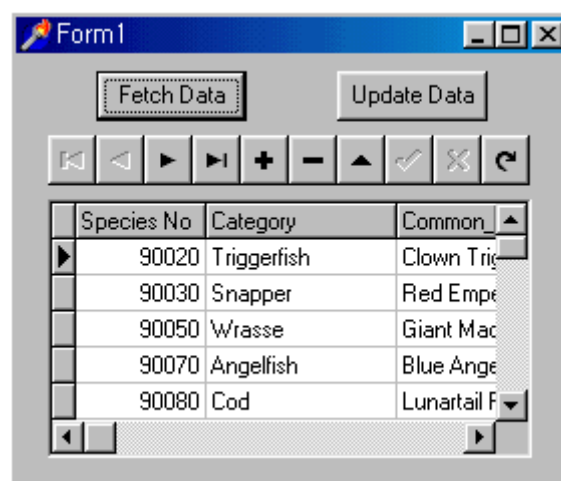
즉, 먼저 커넥션 컴포넌트를 원격 서버에 연결하고 AppServer 프로퍼티를 이용하여 서버 컴포넌트의 인터페이스를 얻은 뒤에 이 인터페이스의 GetData 메소드를 호출하면 된다. 이때 클라이언트 데이터 세트의 Data 프로퍼티 역시 OleVariant 데이터 형이기 때문에 쉽게 적용이 가능하다.

마찬가지로 데이터를 업데이트하는 Button2 의 OnClick 이벤트 핸들러는 다음과 같이 작성하여 구현할 수 있다.

```
procedure TForm1.Button2Click(Sender: TObject);
var
    NumErrors: Integer;
    ErrorPackets: OleVariant;
begin
    DCOMConnection1.Connected := True;
    if ClientDataSet1.ChangeCount > 0 then
        ErrorPackets :=
            DCOMConnection1.AppServer.ApplyUpdates(ClientDataSet1.Delta, 2, NumErrors)
    else
        ShowMessage('업데이트할 레코드가 없습니다 !');
    DCOMConnection1.Connected := False;
end;
```

일단 접속을 한 뒤에 변경된 레코드가 있는지 검사하고, 있다면 서버 인터페이스의 ApplyUpdates 메소드를 호출하여 변경된 내용을 전송하게 된다.

간단한 클라이언트 어플리케이션의 작성이 완료되었다. 이제 이를 컴파일하고 실행하면 다음과 같이 원격 서버와 함께 동작하는 데이터베이스 클라이언트 어플리케이션의 화면을 볼 수 있을 것이다.



정 리 (Summary)



이번 장에서는 구체적인 MTS 패키지와 컴포넌트를 서버와 클라이언트에 설치하고, 이를 활용하는 방법과 텔파이 4 에서 제공하는 위저드를 이용하여 실제로 MTS 컴포넌트를 작성하는 방법에 대해서 알아보았다.

필자의 개인적인 생각으로는 MTS 환경은 CORBA 환경에 비해 가벼우면서도 강력한 기능을 제공하는 우수한 환경으로 볼 수 있을 것 같다. 엔터프라이즈 환경이 아닌 적당한 규모에서, 특히 중소 규모의 네트워크 환경에서 MTS 를 이용하여 분산 환경을 구축한다면 가장 저렴하면서도 적합한 솔루션이 될 것으로 믿는다.

텔파이 4 에서 제공하는 기능 중에서도 가장 실용적이고, 우수한 위저드인 MTS 관련 위저드를 적절하게 활용하는 것이 바로 분산환경에 쉽게 적응하는 지름길이 될 것이므로, 여기에 대한 공부를 소홀히 하지 않기를 바라는 바이다.

## CORBA 의 개념과 활용 (I)

텔파이 4 에서는 텔파이 3 에서 COM 을 완벽하게 지원하게 되는데 이어서 CORBA(Common Object Request Broker Architecture)에 기초한 분산 어플리케이션을 작성하기 쉽도록 여러 가지 위저드와 클래스 들을 제공한다.

CORBA 는 OMG(Object Management Group)에서 채용한 분산 객체 어플리케이션 개발에 대한 표준 스펙으로, 분산 어플리케이션을 제작하는데 객체 지향적인 접근 방법을 제공한다. 이는 인터넷 서버 어플리케이션의 제작에서 HTTP 어플리케이션에 대한 메시지 지향 접근 방법과 비교된다. CORBA 하에서 서버 어플리케이션은 잘 정의된 인터페이스를 기반으로 클라이언트 어플리케이션에 의해 원격으로 사용되는 객체를 구현한다.

CORBA 스펙은 클라이언트 어플리케이션이 어떻게 서버에 구현된 객체와 통신하는지를 정의하는데, 이러한 통신은 ORB(Object Request Broker)에 의해 다루어진다. 텔파이 4 에서는 Inprise 의 VisiBroker ORB 를 CORBA 를 지원하는데 사용한다.

클라이언트와 서버 기계 상의 객체 사이의 통신을 가능하게 하는 기초적인 ORB 기술에 추가하여, CORBA 표준은 수많은 표준 서비스를 정의하고 있다. 이러한 서비스 들은 잘 정의된 인터페이스를 사용하기 때문에, 개발자는 비록 다른 업체에서 만든 서비스라 할 지라도 같은 인터페이스를 구현했으면 같은 방법으로 서비스를 사용할 수 있다.

텔파이 4 의 클라이언트/서버 버전은 기초적인 ORB 기술을 지원하며, 엔터프라이즈 버전은 레코드 형과 같은 추가적인 데이터 형과 SSL 보안과 같은 CORBA 서비스를 제공한다.

이번 장에서는 CORBA 에 대한 전체적인 개념과 텔파이 4 에서 지원하는 CORBA 와 관련된 클래스의 사용 방법에 대해서 알아보도록 할 것이다.

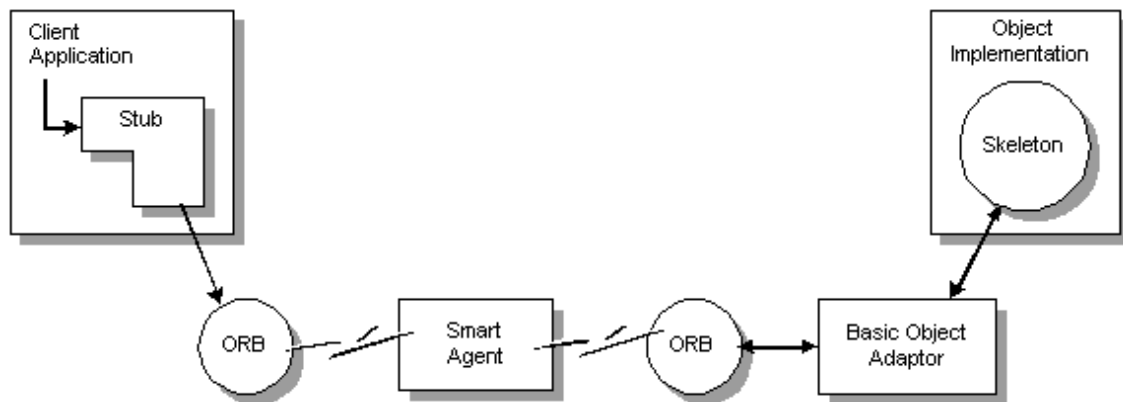
### CORBA 어플리케이션의 개괄

CORBA 어플리케이션의 디자인은 다른 객체 지향 어플리케이션과 별반 다를 것이 없지만, 다른 기계에 존재하는 객체와 네트워크 상에서 통신할 수 있도록 추가적인 계층(layer)를 포함해야 한다는 점이 다르다. 이런 추가적인 계층은 스텝(stub)과 스켈레톤(skeleton)이라고 불리는 특수한 객체에 의해 조절된다.

CORBA 클라이언트에서 스텝(stub)은 서버 기계에 실제로 구현된 객체에 대한 프록시(proxy)로 행동한다. 클라이언트는 인터페이스를 구현한 객체와 직접 상호작용 하듯이 스텝에 접근하게 된다.

어쨌든 인터페이스를 구현한 대부분의 객체와는 달리, 스텝은 클라이언트 기계에 설치된 ORB 소프트웨어를 호출하여 인터페이스 호출을 다루게 된다. ORB 는 LAN 상의 어느 곳에서든 동작하고 있는 스마트 에이전트 (Smart Agent, osagent)를 이용하게 되는데, 스마트 에이전트는 다음 그림과 같이 실제 객체의 구현을 지원하는 가능한 서버에 위치하여 동적이

고 분산된 디렉토리 서비스를 하게 된다.



CORBA 서버에서 ORB 소프트웨어는 인터페이스 호출을 자동으로 생성된 스켈레톤에 넘겨주게 되며, 스켈레톤은 BOA(Basic Object Adaptor)를 통해 ORB 소프트웨어와 통신하게 된다. BOA를 사용하여 스켈레톤은 객체를 스마트 에이전트에 등록하게 되는데, 여기에서 객체의 범위(원격 기계에서 사용될 지 여부 등)와 객체가 인스턴스화 되어 클라이언트에 반응할 수 있게 되는 시기 등을 결정하게 된다.

#### ● 스텝과 스켈레톤의 이해 (Understanding stubs and skeletons)

스텝과 스켈레톤은 CORBA 어플리케이션이 인터페이스 호출을 다음과 같이 마샬링(marshaling)한다.

- 서버 프로세스의 인터페이스 포인터를 가져다가 이 포인터를 클라이언트 프로세스의 코드에서 접근할 수 있도록 해준다.
- 클라이언트로 부터의 인터페이스 호출의 argument 들을 원격 객체의 프로세스 공간에 위치시킨다.

어떤 인터페이스 호출에서도 호출자는 argument 들을 스택에 밀어 넣고, 함수 호출을 인터페이스 포인터를 통해 시도한다. 객체가 인터페이스를 호출한 코드와 같은 프로세스 공간에 있지 않으면, 호출은 같은 프로세스 공간의 스텝을 거쳐 바깥으로 나가게 된다. 스텝은 argument 들을 마샬링 버퍼에 밀어넣고, 원격 객체에 구조체의 형태로 호출을 전송한다. 서버 스켈레톤은 이 구조체를 풀어서 argument 들을 스택에 밀어넣고 객체의 구현부분을 호출한다.

스텝과 스켈레톤은 객체의 인터페이스를 정의할 때 자동으로 생성된다. 개발자가 인터페이스를 정의할 때 이들의 정의는 \_TLB 유닛에 생성된다.

- 스마트 에이전트의 활용 (Using Smart Agents)

스마트 에이전트는 객체를 구현한 서버에 위치한 동적, 분산 디렉토리 서비스이다. 만약 선택할 서버가 많으면, 로드 밸런싱(load balancing)을 지원한다. 또한, 서버에 접속이 실패하면 서버의 재시작을 시도하거나, 다른 호스트 컴퓨터에 위치한 서버를 실행한다. 스마트 에이전트는 클라이언트와 서버 어플리케이션 양측에 완전히 투명하게 접근할 수 있다.

스마트 에이전트는 LAN 상에서 최소한 하나의 호스트에서 시작되어야 하며, ORB 는 스마트 에이전트의 위치를 브로드캐스트 메시지를 보내어 확인한다. 네트워크에 여러 개의 스마트 에이전트가 있으면, ORB 는 먼저 처음으로 반응하는 스마트 에이전트를 사용한다. 일단 스마트 에이전트의 위치가 확인되면 ORB 는 point-to-point UDP 프로토콜을 사용하여 스마트 에이전트와 통신하게 된다. UDP 프로토콜은 TCP 연결에 비해 적은 네트워크 리소스를 사용한다는 장점이 있다.

네트워크에 여러 개의 스마트 에이전트가 있으면, 각각의 스마트 에이전트는 사용가능한 객체를 인식하여 다른 스마트 에이전트의 위치를 직접 연결할 수 있게 되며, 하나의 스마트 에이전트가 비정상적으로 종료될 경우, 객체 들은 자동으로 다른 스마트 에이전트에 재등록되어 사용할 수 있게 된다.

- 서버 어플리케이션의 활성화 (Activating server applications)

서버 어플리케이션은 자신이 시작될 때, 클라이언트의 호출을 받을 수 있는 인터페이스의 ORB 에 BOA 를 통해서 이를 알린다. 이렇게 ORB 를 초기화하고, 서버가 실행되었으며 실행이 가능하다고 알리는 코드는 CORBA 서버 어플리케이션을 시작할 때 사용하는 위저드에 의해 추가된다.

전형적으로 CORBA 서버 어플리케이션은 수동으로 시작하지만, OAD(Object Activation Daemon)를 사용하면 서버를 자동으로 시작하게 하거나 클라이언트가 필요할 때 객체를 인스턴스화 할 수 있다.

OAD 를 사용하기 위해서는 객체를 반드시 등록해야 한다. 객체를 OAD 에 등록할 때에 객체와 객체를 구현한 서버 어플리케이션 사이의 연관 관계를 구현 저장소(Implementation Repository)에 저장한다.

객체가 일단 구현 저장소에 등록되어 엔트리를 가지고 있으면, OAD 는 어플리케이션을 ORB 에 시뮬레이션하게 된다. 그러다가, 클라이언트가 객체를 요구하면 ORB 는 OAD 와 연계하여 마치 서버 어플리케이션이 실행중인 것처럼 반응하고, 동시에 OAD 는 클라이언트의 요구를 실제 서버에 전달하게 된다. 이 과정에서 필요하면 서버 어플리케이션을 시동한다.

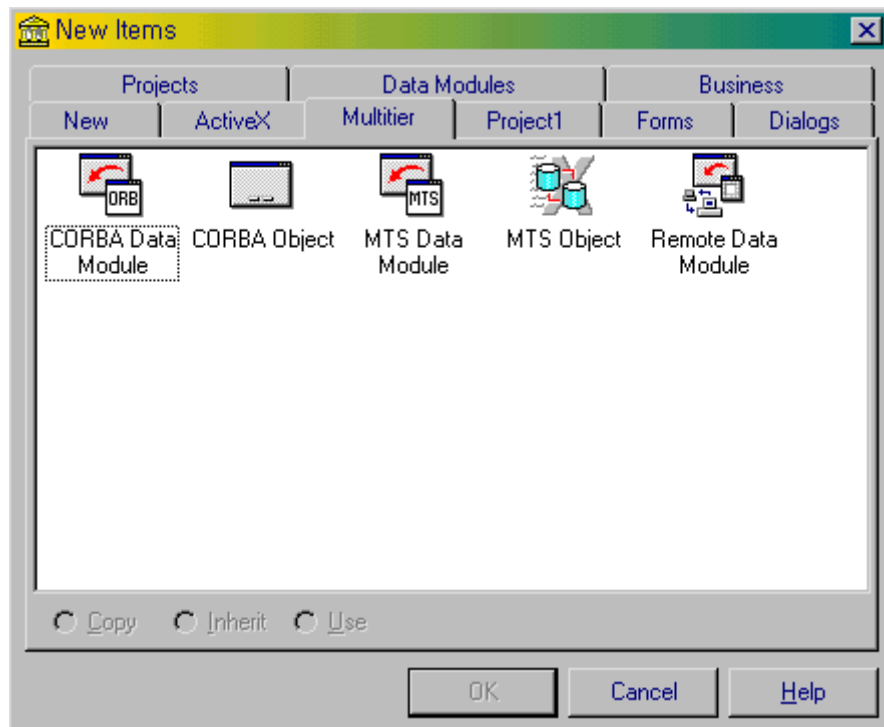
- 인터페이스 호출의 동적 바인딩 (Binding interface calls dynamically)

전형적으로 CORBA 클라이언트는 서버 상의 객체의 인터페이스를 호출할 때 정적 바인딩 (static, early binding)을 사용한다. 이 방법은 빠른 수행성과 컴파일 시 데이터 형 검사 등의 많은 장점을 가지고 있다. 그렇지만, 런타임이 될 때까지 어떤 인터페이스를 사용하기를 원할 지 모를 수도 있는데 이럴 때에는 인터페이스를 런타임에 동적으로 바인딩할 수 있도록 허용하고 있다.

이런 동적 바인딩의 장점을 이용하고자 하면, 반드시 인터페이스를 인터페이스 저장소 (Interface Repository)에 idl2ir 유틸리티를 이용해서 저장해야 한다.

## CORBA 서버의 제작

New Items 대화상자의 Multitier 페이지에 있는 다음과 같은 2 개의 위저드를 사용하여 CORBA 서버를 생성할 수 있다.



- CORBA 데이터 모듈 위저드를 이용하면 멀티-tiered 데이터베이스 어플리케이션에 대한 CORBA 서버를 제작할 수 있다.
- CORBA 객체 위저드는 다양한 CORBA 서버를 생성할 때 사용한다.

또한, 기존에 작성한 OLE 자동화 서버가 있으면 오른쪽 버튼을 클릭하고 Exposing As

CORBA Object 메뉴를 선택하는 것만으로 간단하게 CORBA 서버로 전환할 수 있다. 자동화 서버를 CORBA 객체를 지원하도록 하면, 어플리케이션은 COM 클라이언트와 CORBA 클라이언트를 동시에 서비스할 수 있게 된다.

## ● CORBA 위저드의 활용

위저드를 시작하려면, File|New 메뉴를 선택하여 New Items 대화상자가 나오도록 하고, Multitier 페이지를 선택하고 적절한 위저드를 더블 클릭하면 된다.

이때 CORBA 객체의 클래스 이름을 적어 주면, 이 클래스가 TCorbaDataModule 이나 TCorbaImplementation 클래스를 상속받게 된다. 클래스 이름을 MyObject 라고 하면 위저드는 IMyObject 인터페이스를 구현한 TMyObject 를 선언하는 새로운 유닛을 생성한다. 또한, 위저드에서는 서버 어플리케이션의 인스턴스 모델과 쓰레드 모델을 선택해야 한다. 인스턴스 모델에는 다음의 2 가지가 있으며 적절한 것을 선택한다.

### 1. 인스턴스-per-클라이언트 모델

CORBA 데이터 모듈 인스턴스는 각 클라이언트 연결마다 하나씩 생성되며, 연결이 지속되는 동안에는 지속된다. 클라이언트의 연결이 종료되면 인스턴스가 메모리에서 해제된다. 이 모델은 분리된 클라이언트가 각각의 프로퍼티 설정에 서로 간섭하지 않기 때문에 지속적인 상태 정보(persistent state information)를 사용할 수 있다. 클라이언트 시스템이 객체 인스턴스를 해제하지 않고 종료되는 것을 막기 위해 어플리케이션은 주기적으로 클라이언트가 아직도 실행되고 있는지를 점검하게 되므로 공유 인스턴스 모델에 비해 이런 메시지 만큼의 네트워크 오버헤드를 가지게 된다.

### 2. 공유 인스턴스(shared instance) 모델

CORBA 데이터 모듈의 단일 인스턴스는 모든 클라이언트 요구를 다루게 된다. 이 모델은 전통적인 CORBA 개발 방법에 기초한 것이다. 단일 인스턴스는 모든 클라이언트를 공유해야 하므로, 이 인스턴스는 프로퍼티 설정과 같은 지속적인 상태 정보를 사용할 수 없다. 이는 클라이언트가 CORBA 데이터 모듈에서 프로바이더와 통신하게 되는 IProvider 인터페이스를 이용할 수 없다는 의미가 된다.

인스턴스 모델을 선택했으면, 이번에는 쓰레드 모델을 선택해야 한다. 쓰레드 모델에는 다음의 2 가지가 있다.

#### 1. 싱글 쓰레드(single threaded) 모델

각각의 데이터 모듈 인스턴스는 한 번에 하나씩의 클라이언트 요구만 처리한다. 그렇기 때문에, 프로퍼티나 필드와 같은 인스턴스 데이터는 쓰레드에 안전하다. 그렇지만, 전역 메모리를 사용하는 경우에는 명확하게 이를 보호해야 한다.

## 2. 다중 쓰레드(multi threaded) 모델

각각의 클라이언트 연결은 자신의 쓰레드를 가지게 되지만, 데이터 모듈의 입장에서 보면 여러 클라이언트의 호출을 동시에 받게 되며 각각의 클라이언트는 자신의 쓰레드를 처리하게 된다. 그러므로 전역 메모리를 사용하는 경우 뿐만 아니라 프로퍼티나 필드 등의 인스턴스 데이터가 변화될 가능성에 대비하여 이를 보호할 책임이 있다.

### ● 객체 인터페이스의 정의 (Defining object interfaces)

전통적인 CORBA 도구에서는 객체의 인터페이스를 정의하기 위해서 CORBA IDL(Interface Definition Language)을 사용해야 했다. 그리고 나서, 이들을 스텝과 스켈레톤 코드로 생성해내는 유틸리티를 실행하여 사용하였다. 그렇지만, 델파이에서는 IDL 과 스텝, 스켈레톤을 모두 자동으로 생성해 준다. 개발자가 할 일은 타입 라이브러리 에디터를 이용하여 인터페이스를 편집하면 델파이가 알아서 적절한 소스 파일들을 업데이트 한다.

타입 라이브러리 에디터는 델파이 3 에서 처음 소개 되었는데, 여기서는 COM 기반의 타입 라이브러리를 제작하는 도구로 사용되었다. 그렇기 때문에, CORBA 어플리케이션과는 맞지 않는 옵션과 컨트롤 들을 많이 포함하고 있다. 개발자가 이런 옵션 들을 사용하려고 하면 이들의 설정은 CORBA 에 의해서는 무시된다. 그러므로, COM 자동화 서버를 만들면서 CORBA 서버로도 사용할 수 있도록 할 때에 이런 설정 값들이 자동화 서버를 만들 때 적용된다.

델파이 4 에서는 타입 라이브러리 에디터에서 오브젝트 파스칼 문법이나 COM 객체를 정의할 때 사용되는 마이크로소프트 IDL 문법을 모두 사용할 수 있다. 이때 어느 것을 디폴트로 할 것인지는 Environment Option 대화상자의 Type Library 페이지에서 설정할 수 있다. 주의할 것은 IDL 을 이용하겠다고 선택한 경우, Microsoft IDL 은 CORBA IDL 과 다소 차이가 있으므로, 인터페이스를 디자인할 때 다음에 소개하는 데이터 형만을 사용할 수 있다.

이 름	설 명	이 름	설 명
ShortInt	8 비트 부호 있는 정수	Byte	8 비트 부호 없는 정수
SmallInt	16 비트 부호 있는 정수	Word	16 비트 부호 없는 정수
LongInt	32 비트 부호 있는 정수	Cardinal	32 비트 부호 없는 정수
Single	4 바이트 부동 소수점 실수	Double	8 바이트 부동 소수점 실수

TDateTime	Double 값으로 넘어감	PWideChar	유니코드 문자열
PChar, String	문자열은 반드시 PChar 형태로 넘어가야 함	Variants	CORBA 에서는 Any 로 넘긴다. 배열 등을 넘길 수 있다.
Boolean	CORBA_Boolean 으로 넘어감	Enumeration	정수로 넘어간다.

이 밖에 Object Reference 나 Interface 형은 CORBA 인터페이스 형으로 넘길 수 있다. 참고로, 타입 라이브러리 에디터를 사용하지 않고 코드 에디터에서 오른쪽 버튼을 클릭하고 Add To Interface 를 선택해서 인터페이스를 추가하는 방법도 있다. 그렇지만, 이 경우에도 타입 라이브러리 에디터를 이용해서 .IDL 파일로 저장해야 사용이 가능하다.

파라미터를 사용하는 프로퍼티는 추가할 수 없으며, 이런 프로퍼티를 지원하게 하려면 get, set 메소드를 이용해서 구현해야 한다. 배열이나 Int64, Currency 등의 일부 데이터 형은 Variants 형으로 지정해야 한다. 레코드는 C/S 버전에서는 지원되지 않으며 Enterprise 버전에서 지원된다.

인터페이스에 대한 변경 사항은 자동으로 생성되는 스텝-스켈레톤 유닛에 반영되며, 이 유닛은 타입 라이브러리 에디터에서 Refresh 명령을 선택하거나 Add To Interface 명령을 사용하면 업데이트 된다. 이렇게 자동으로 생성된 유닛은 implementation 유닛의 uses 절에 추가되므로 스텝-스켈레톤 유닛을 직접 편집하는 것은 피하는 것이 좋다.

인터페이스를 편집하면 스텝-스켈레톤 유닛 이외에 서버 구현 유닛에 인터페이스 멤버에 대한 선언과 메소드 구현부에 대한 꺾이기 코드가 자동으로 생성된다. 개발자는 구현부에 생성된 begin ~ end 사이에 적절한 인터페이스를 구현하면 된다.

#### 참고:

CORBA 의 IDL 파일은 타입 라이브러리 에디터의 Export 버튼을 클릭하고 CORBA IDL 을 선택하면 따로 저장할 수 있다. 이렇게 저장한 .IDL 파일을 이용해서 인터페이스를 등록하거나 C++ 등의 다른 언어를 이용하여 스텝과 스켈레톤을 생성하도록 할 수 있다.

#### ● 자동으로 생성된 코드

CORBA 객체 인터페이스를 정의하고 나면, 인터페이스 정의를 반영하기 위해 2 개의 유닛 파일이 자동으로 업데이트 된다.

첫번째 유닛은 스텝-스켈레톤 유닛으로 MyInterface\_TLB.pas 라는 이름을 가지는 파일이다. 이 유닛은 클라이언트 어플리케이션에 의해 사용되는 스텝 클래스를 정의하고, 인터페이스 형과 스켈레톤 클래스에 대한 선언부를 포함한다. 이 파일은 직접 편집하면 안되며, 구현 유닛의 uses 절에 자동으로 추가된다.

스텝-스켈레톤 유닛은 CORBA 서버에 의해 지원되는 인터페이스에 대한 스켈레톤 객체를



정의한다. 스켈레톤 객체는 TCorbaSkeleton 클래스의 자손으로 인터페이스 호출을 마샬링하는 세부적인 동작을 처리하게 된다. 이 객체는 정의한 인터페이스를 직접 구현하는 것은 아니지만, constructor 가 인터페이스 인스턴스를 이용하여 모든 인터페이스 호출을 처리하게 된다.

두번째로 업데이트 되는 유닛은 실제 구현을 담당하는 implementation 유닛이다. 이 유닛의 디폴트 이름은 Unit1.pas 이다. 이 이름은 물론 다른 이름으로 바꿀 수 있다.

정의한 각각의 CORBA 인터페이스에 대해 구현 클래스는 자동으로 정의되어 implementation 유닛에 추가된다. 구현 클래스의 이름은 인터페이스 이름에 기초하는데, 예를 들어, 인터페이스 이름이 IMyInterface 라면 구현 클래스의 이름은 TMyInterface 가 된다. 기본적으로 인터페이스에 선언된 메소드에 대한 구현 부분의 껍데기는 자동으로 생성되므로 이들에 대한 몸체만 코딩하면 된다.

또한, implementation 유닛의 initialization 섹션에는 CORBA 클라이언트 들에 노출된 각각의 객체 인터페이스에 대한 TCorbaFactory 객체를 생성하는 코드가 추가된다. 클라이언트가 CORBA 서버를 호출하면 CORBA 팩토리 객체가 생성되거나 구현 클래스의 인스턴스를 위치시키고 이를 인터페이스로 하여 해당되는 스켈레톤 클래스에 대한 constructor 에 넘겨진다.

## ● 서버 인터페이스의 등록 (Registering server interfaces)

클라이언트 호출에 대해서 서버 객체를 정적 바인딩만 사용하려 한다면 서버 인터페이스를 반드시 등록할 필요는 없지만, 등록시키는 것이 좋다. 인터페이스를 등록하는데 사용할 수 있는 유틸리티는 다음의 2 가지가 있다.

### 1. 인터페이스 저장소 (Interface Repository)

인터페이스 저장소에 인터페이스를 등록하면 클라이언트는 동적 바인딩을 사용할 수 있다. 이를 통해 클라이언트가 DII(dynamic invocation interface)를 사용했다면 델파이가 아닌 다른 언어로 작성된 클라이언트에도 서버가 반응할 수 있다. 인터페이스 저장소를 이용하면 다른 개발자가 인터페이스를 직접 보고, 이를 이용해서 클라이언트 어플리케이션을 작성하도록 할 수 있는 장점이 있다.

### 2. 객체 활성화 데몬 (Object Activation Daemon, OAD)

객체 활성화 데몬에 인터페이스를 등록하면 객체가 클라이언트에 의해 요구를 받을 때까지 인스턴스화 되지 않게 할 수 있다. 이를 통해 서버 시스템의 리소스를 많이 절약할 수 있다.

## CORBA 클라이언트의 제작

CORBA 클라이언트를 작성할 때의 첫번째 단계는 클라이언트 어플리케이션이 클라이언트의 ORB 소프트웨어와 통신할 수 있도록 하는 것이다. 이를 위해서는 유닛 파일의 `uses` 절에 `CorbaInit.pas` 유닛을 추가하면 된다.

그리고 나서는 일반적인 다른 델파이 어플리케이션을 개발하는 방법과 똑같이 개발하면 된다. 서버 어플리케이션에 정의된 객체를 사용하고자 할 때에는 객체 인스턴스를 직접 호출하지 않고, 객체에 대한 인터페이스를 얻어서 이를 이용하여 작업을 하면 된다. 인터페이스를 얻는 방법에는 크게 정적 바인딩(static, early binding)과 동적 바인딩(dynamic, late binding)이 있다.

정적 바인딩을 이용하려면 스텝-스켈레톤 유닛을 클라이언트 어플리케이션에 추가해야 하는데, 이 유닛은 서버 인터페이스를 저장할 때 자동으로 생성된다. 정적 바인딩이 동적 바인딩에 비해 수행속도가 빠르며, 컴파일 시에 데이터 형을 검사하는 장점이 있고 동시에 델파이 3 에서부터 지원하는 코드 완료(code completion)를 사용할 수 있다.

그렇지만, 어떤 객체나 인터페이스를 사용해야 하는지를 런타임이 될 때까지 모를 때에는 동적 바인딩을 사용해야 한다. 동적 바인딩은 스텝 유닛을 요구하지 않지만, 사용하는 모든 원격 객체 인터페이스(remote object interface)가 LAN 상에서 동작하는 인터페이스 저장소에 등록되어 있어야 한다.

### ● 스텝의 활용 (Using stubs)

스텝 클래스는 CORBA 인터페이스를 정의할 때 자동으로 생성된다. 이 클래스는 스텝-스켈레톤 유닛에 정의되어 있다. 스텝-스켈레톤 유닛의 이름은 `BaseName_TLB.pas` 이다. CORBA 클라이언트를 작성할 때, 스텝-스켈레톤 유닛의 코드는 직접 수정하지 않는다. 단지 이 유닛을 `uses` 절에 추가하고, 인터페이스를 호출하면 된다.

각각의 서버 객체에 대해 스텝-스켈레톤 유닛은 해당 스텝 클래스에 대한 인터페이스 정의와 클래스 정의를 포함하고 있다. 예를 들어, 서버가 `TServerObj` 객체 클래스를 정의한다면 스텝-스켈레톤 유닛은 `IServerObj` 인터페이스에 대한 정의와 `TServerObjStub` 클래스에 대한 정의를 포함한다. 스텝 클래스는 `TCorbaDispatchStub` 클래스의 자손으로 해당되는 인터페이스의 마샬링하여 CORBA 서버를 호출하는 역할을 한다. 스텝 클래스 외에도 스텝-스켈레톤 유닛에는 각각의 인터페이스에 대한 스텝 클래스 팩토리를 정의하고 있다. 스텝 클래스 팩토리는 인스턴스화 되지 않으며, 하나의 클래스 메소드만 정의한다.

클라이언트 어플리케이션에서는 CORBA 서버의 객체에 대한 인터페이스를 필요로할 때 스텝 클래스의 인스턴스를 직접 생성하지 않는다. 그 대신에 스텝 팩토리 클래스의 `CreateInstance` 메소드를 호출한다. 이 메소드는 옵션 인스턴스 이름을 하나의 argument

를 가지며, 서버 상의 객체 인스턴스에 대한 인터페이스를 반환한다.  
사용법은 다음과 같다.

```
var
  ObjInterface : IServerObj;
begin
  ObjInterface := TServerObjFactory.CreateInstance("");
  ...
end;
```

CreateInstance 를 호출할 때 이 메소드는 ORB 에서 인터페이스 인스턴스를 얻어오며, 인터페이스를 이용하여 스텝 클래스의 인스턴스를 생성한다. 마지막으로 결과 인터페이스를 반환한다.

#### ● DII(dynamic invocation interface)의 활용

DII 는 클라이언트 어플리케이션이 인터페이스 호출을 마샬링하는 스텝 클래스를 사용하지 않고도 서버 객체를 호출할 수 있도록 해준다. DII 는 컴파일 시에 인터페이스 호출을 바인딩하지 않기 때문에 스텝 클래스를 사용하는 것보다 다소 느리게 동작한다.

DII 를 사용하기 전에 서버 인터페이스는 LAN 상에서 동작하는 인터페이스 저장소에 등록되어 있어야 한다. 클라이언트 어플리케이션에서 DII 를 사용하려면 서버 인터페이스를 얻어서 이를 TAny 데이터 형 변수에 대입한다. TAny 는 CORBA 에서 사용하는 특수한 Variant 데이터 형이다. 그리고 나서 TAny 변수를 인터페이스 인스턴스인 것처럼 사용하여 인터페이스 메소드를 호출하면 된다.

#### ● 인터페이스의 획득 (Obtaining the interface)

DII 를 이용한 동적 바인딩을 사용하기 위해서는 전역 CorbaBind 함수를 호출하면 된다. CorbaBind 함수에는 서버 객체의 Repository ID 나 인터페이스 형을 정해주어야 한다. 이 정보를 이용해서 ORB 에게 인터페이스를 요구하고, 스텝 객체를 생성하게 된다.

CorbaBind 함수를 호출하기 전에 인터페이스 형과 인터페이스의 Repository ID 는 반드시 CorbaInterfaceIDManager 함수를 이용해서 등록해야 한다.

클라이언트 어플리케이션이 인터페이스 형에 대한 스텝 클래스를 등록하면, CorbaBind 함수는 그 클래스의 스텝을 생성하고, 이 함수에 의해 반환된 인터페이스는 'as' 형변환을 통한 정적 바인딩과 동적 바인딩에 모두 사용될 수 있다. 이때 인터페이스 형에 대해 등록된 스텝 클래스가 없다면, CorbaBind 는 기본 스텝 객체(generic stub object)에 대한 인터

페이스를 반환한다. 기본 스텝 객체는 DII 를 이용한 동적 호출을 이용해야만 사용할 수 있다. 실제 사용하는 코드를 살펴 보자.

```
var
    IntToCall: TAny;
begin
    IntToCall := CorbaBind('IDL:MyServer/MyServerObject:1.0');
    ...
```

#### ● DII 를 이용한 인터페이스 호출

인터페이스가 TAny 형 변수에 대입되면, DII 를 이용하여 다음과 같이 간단하게 호출하여 사용할 수 있다.

```
var
    HR, Emp, Payroll, Salary: TAny;
begin
    HR := CorbaBind('IDL:CompanyInfo/HR:1.0');
    Emp := HR.LookupEmployee(Edit1.Text);
    Payroll := CorbaBind('IDL:CompanyInfo/Payroll:1.0');
    Salary := Payroll.GetEmployeeSalary(Emp);
    Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit2.Text) / 100));
end;
```

DII 를 사용할 때에는 모든 인터페이스 메소드가 대소문자를 가리므로 주의하기 바란다. DII 를 이용해서 인터페이스를 호출하면 모든 파라미터가 TAny 데이터 형으로 취급된다. 이 데이터 형을 사용하면 서버가 호출을 받은 후에 데이터 형을 해석하는 것을 허용한다. 파라미터가 항상 TAny 값으로 취급되기 때문에 파라미터 데이터 형을 적절하게 맞추기 위해 먼저 형변환을 할 필요가 없다. 예를 들어, 앞에서의 SetEmployeeSalary 메소드의 경우 문자열 대신 실수를 파라미터로 사용해도 된다.

그렇지만 구조체를 파라미터로 사용하려면 전역 ORB 변수의 변환 메소드를 이용하여 적절한 TAny 데이터 형 값을 만들어 사용해야 한다. 이때 레코드는 MakeStructure, 정적 배열은 MakeArray, 동적 배열은 MakeSequence 라는 메소드를 이용하여 변환을 한다.

이런 함수를 이용할 때에는 생성하고자 하는 레코드, 배열의 데이터 형을 반드시 기술해야 하는데, 이런 데이터 형은 다음과 같이 ORB 의 FindTypeCode 메소드에서 Repository ID 를 이용하면 동적으로 얻을 수 있다.

```

var
    HR, Name, Emp, Payroll, Salary: TAny;
begin
    with ORB do
        begin
            HR := Bind('IDL:CompanyInfo/HR:1.0');
            Name := MakeStructure(FindTypeCode('IDL:CompanyInfo/EmployeeName:1.0',
                                                [Edit1.Text, Edit2.Text]));
            Emp := HR.LookupEmployee(Name);
            Payroll := Bind('IDL:CompanyInfo/Payroll:1.0');
        end;
        Salary := Payroll.GetEmployeeSalary(Emp);
        Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit3.Text) / 100));
    end;
end;

```

## CORBA 어플리케이션 구워 삶기

ORB 와 BOA 라는 2 개의 전역 변수를 이용하면 어플리케이션이 CORBA 소프트웨어와 상호작용하는 방법을 여러 가지로 적용해볼 수 있다.

클라이언트 어플리케이션은 ORB 변수를 사용해서 ORB 소프트웨어의 환경을 설정하고, 서버와의 연결을 해제하며, 인터페이스에 바인드하여 객체에 대한 문자열을 얻는 등의 작업을 하게 된다.

서버 어플리케이션은 BOA 변수를 이용해서 BOA 소프트웨어의 환경을 설정하고, 숨겨진 객체를 노출시키며, 클라이언트 어플리케이션에 의해 주어진 사용자 정의 정보를 가져올 수 있다.

### ● 사용자 인터페이스에 객체 보여주기

CORBA 클라이언트 어플리케이션을 제작할 때, 사용가능한 CORBA 서버 객체의 이름을 사용자에게 보여주고 싶을 때가 있다. 이럴 때에는 객체의 인터페이스를 문자열로 변환해서 보여주어야 하는데, ORB 변수의 ObjectToString 메소드가 이런 역할을 한다. 예를 들어, 다음의 코드는 3 개의 객체를 리스트 박스에 보여준다.

```

var
    Dept1, Dept2, Dept3: IDepartment;

```

```

begin
    Dept1 := TDepartmentFactory.CreateInstance('Sales');
    Dept1.SetDepartmentCode(120);
    Dept2 := TDepartmentFactory.CreateInstance('Marketing');
    Dept2.SetDepartmentCode(98);
    Dept3 := TSecondFactory.CreateInstance('Payroll');
    Dept3.SetDepartmentCode(49);
    ListBox1.Items.Add(ORB.ObjectToString(Dept1));
    ListBox1.Items.Add(ORB.ObjectToString(Dept2));
    ListBox1.Items.Add(ORB.ObjectToString(Dept3));
end;

```

ORB 변수는 이 뿐만 아니라 문자열을 이용해서 객체를 생성할 수 있도록 StringToObject 메소드 역시 제공하고 있다. 다음 코드를 살펴보자.

```

var
    Dept: IDepartment;
begin
    Dept := ORB.StringToObject(ListBox1.Items[ListBox1.ItemIndex]);
    ...      {이 객체를 사용하는 코드}

```

## ● CORBA 객체의 노출과 숨김

CORBA 서버 어플리케이션이 객체 인스턴스를 생성할 때, BOA 변수의 ObjIsReady 메소드를 호출하면 클라이언트에서 이 객체를 사용할 수 있게 된다. ObjIsReady 메소드를 통해 사용가능해진 객체는 서버 어플리케이션이 다시 숨길 수 있다. 이를 위해서는 BOA의 Deactivate 메소드를 사용한다.

서버의 객체를 사용할 수 있는지 여부를 클라이언트 어플리케이션이 알기 위해서는 스텝 객체의 NonExistent 메소드를 사용해서 알아낼 수 있다. 서버 객체가 비활성화된 경우에 이 메소드는 True를 반환하며, 서버 객체를 사용할 수 있으면 False를 반환한다.

## ● 클라이언트 정보를 서버 객체로 넘겨주기

CORBA 클라이언트의 스텝 객체는 연관된 서버 객체에 TCorbaPrincipal을 이용해 정보를 전송할 수 있다. TCorbaPrincipal은 클라이언트 어플리케이션에 대한 정보를 가지고 있는 일종의 배열이다. 스텝 객체는 이 배열의 값을 SetPrincipal 메소드를 이용해서 설정할 수

있다.

일단 CORBA 클라이언트가 데이터를 서버 객체 인스턴스에 기록하면, 서버 객체는 이 정보에 접근할 때 BOA의 GetPrincipal 메소드를 사용하면 된다.

TCorbaPrincipal 이 바이트의 배열이기 때문에, 개발자가 보낼 수 있는 어떤 정보도 전송할 수 있다. 예를 들어, 특정 메소드를 실행가능하게 만들기 전에 서버가 검사할 수 있는 키 값을 클라이언트가 전송하도록 만들 수도 있다,

## CORBA 어플리케이션의 배포

일단 클라이언트나 서버 어플리케이션을 제작하고 이들을 테스트하고 나면 클라이언트 어플리케이션은 사용자의 데스크 탑 컴퓨터에게, 서버 어플리케이션은 서버 기계에 배포해야 한다. 이때 클라이언트와 서버 어플리케이션에는 다음과 같은 파일이 반드시 설치되어 있어야 한다.

1. ORB 라이브러리는 모든 클라이언트와 서버 기계에 설치되어 있어야 한다. 이들 라이브러리는 VisiBroker 를 설치할 때 Bin 서브 디렉토리에 설치되므로 여기에서 찾을 수 있다.
2. 클라이언트가 DII(dynamic invocation interface)를 사용하도록 하였다면, 네트워크 상에 최소한 하나의 호스트에서 인터페이스 저장소(Interface Repository) 서버를 실행해야 한다.
3. 서버가 수동으로 시작되는 것이 아니라 클라이언트의 요구가 있을 때에만 실행하도록 하고 싶다면, OAD(Object Activation Daemon)가 네트워크 상에 최소한 하나의 호스트에서 실행되어야 한다.
4. 네트워크 상에 최소한 하나의 호스트에는 스마트 에이전트(osagent)가 반드시 설치되어야 한다.

그리고, CORBA 어플리케이션을 배포할 때 다음과 같은 환경 변수를 설정할 필요가 있을 수 있다.

변 수	의 미
PATH	ORB 라이브러리가 포함된 디렉토리
VBROKER_ADM	인터페이스 저장소, OAD, 스마트 에이전트에 대한 환경설정 파일을 포함하는 디렉토리
OSAGENT_ADDR	스마트 에이전트가 사용될 호스트 컴퓨터의 IP 주소. 이 변수가 설정되지 않으면 CORBA 어플리케이션은 브로드캐스트 메시지에 응답하는 첫번째 스마트 에이전트를 사용한다.

OSAGENT_PORT	스마트 에이전트가 클라이언트의 요구를 기다릴 포트를 지정한다.
OSAGENT_ADDR_FILE	다른 네트워크 상의 스마트 에이전트 주소를 담고 있는 파일의 패스
OSAGENT_LOCAL_FILE	멀티 호스트에서 동작하는 스마트 에이전트에 대한 네트워크 정보를 담은 파일의 패스
VBROKER_IMPL_PATH	구현 저장소(Implementation Repository)에 대한 디렉토리 지정 (OAD 에 여기에 대한 정보를 담고 있다).
VBROKER_IMPL_NAME	구현 저장소의 디폴트 파일 이름을 지정한다.

## ● 스마트 에이전트의 환경 설정

CORBA 어플리케이션을 배포할 때, 최소한 하나의 스마트 에이전트가 실행되고 있어야 한다. 하나 이상의 스마트 에이전트를 배포할 때에는 현재 실행되는 스마트 에이전트의 실행이 비정상 종료되었을 때 이를 보호하기 위한 방법을 제공해야 한다.

개발자는 네트워크를 ORB 도메인에 따라 스마트 에이전트를 배포한다. 이때 ORB 도메인을 넓히기 위해서 다른 네트워크에 있는 스마트 에이전트를 연결할 수도 있다.

### 1. 스마트 에이전트의 시작

스마트 에이전트를 시작하기 위해서는 osagent 유틸리티를 실행해야 한다. osagent 유틸리티에는 다음과 같은 커맨드 라인 arguments 를 사용할 수 있다.

Argument	설 명
-v	verbose 모드를 활성화한다. 정보와 시스템 진단 메시지가 osagent.log 파일에 기록된다. 이 파일은 VBROKER_ADM 환경 변수에 지정된 디렉토리에서 찾을 수 있다.
-p<n>	스마트 에이전트가 브로드캐스트 메시지를 기다릴 UDP 포트를 지정한다.
-C	NT 서비스로 설치된 경우 스마트 에이전트를 콘솔 모드로 동작하게 한다.

예를 들어, 도스창이나 실행 명령에서 다음과 같이 지정할 경우 UDP 포트는 디폴트로 지정된 14000 을 사용하지 않고, 11000 을 사용하게 된다.

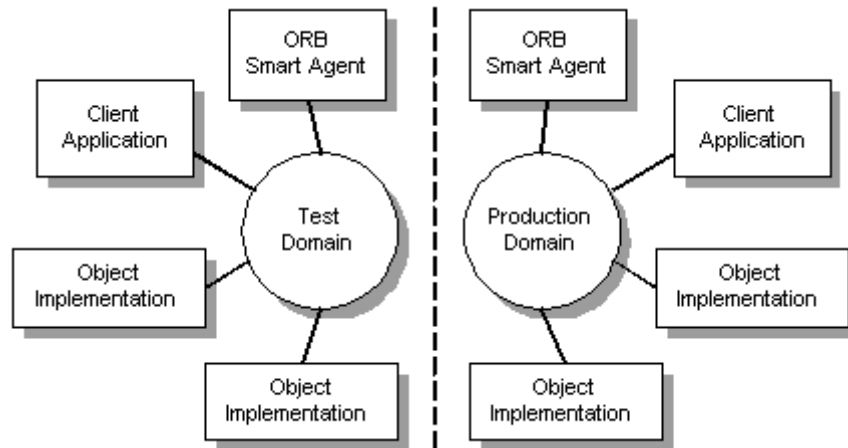
```
osagent -p 11000
```

스마트 에이전트가 브로드캐스트를 기다리는 포트를 바꿀 경우 여러 개의 ORBB 도메인이 생성되는 것을 허용할 수 있다.

### 2. ORB 도메인 환경 설정



2 개 이상의 분리된 ORB 도메인을 실행하는 경우 상당한 장점이 있다. 하나의 도메인에는 클라이언트 어플리케이션의 정식 버전(production version)과 객체를 구현하고, 다른 도메인에는 같은 클라이언트와 객체의 테스트 버전을 포함한다. 이 경우 여러 명의 개발자가 같은 네트워크에서 작업을 할 경우 각각의 개발자는 테스트를 다른 개발자 들을 간섭하지 않고 할 수 있게 된다.



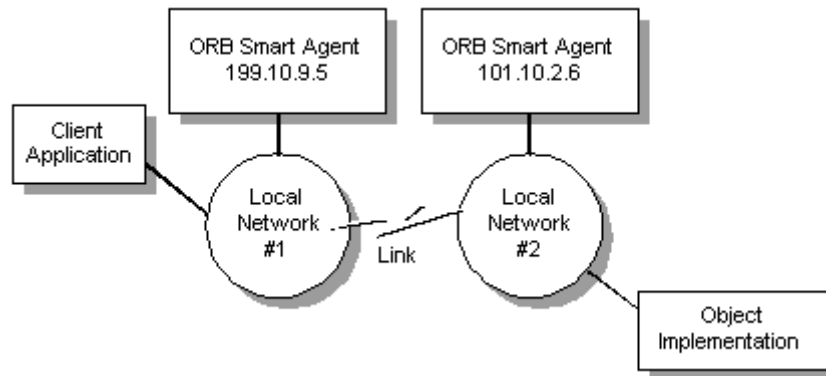
같은 네트워크 상의 2 개 이상의 ORB 도메인은 각 도메인의 osagents 의 유일한 UDP 포트 이름을 이용해서 구별한다. 디폴트 포트 번호는 14000 으로 ORB 가 설치될 때 윈도우 레지스트리에 기록된다. 이 값을 재정의하려면 OSAGENT\_PORT 환경 변수를 다르게 설정하면 된다. 또한, 환경 변수의 값을 또 다시 변경하려면 스마트 에이전트를 시작할 때 -p 옵션을 주면 된다.

### 3. 다른 네트워크 상의 스마트 에이전트와의 접속

네트워크 상에 여러 개의 스마트 에이전트를 시작할 경우, 이들은 UDP 브로드캐스트 메시지를 이용해서 서로를 발견하게 된다. 그러므로, 브로드캐스트 메시지가 전달되는 범위가 결정되는 IP 서브넷 마스크 범위 내에 있어야 한다. 그렇지만, 서로 다른 네트워크 상에 있어도 서로 통신이 가능한데 다음과 같은 방법을 사용할 수 있다.

- agentaddr 파일을 이용하는 방법

다음 그림과 같은 2 개의 스마트 에이전트가 있다고 하자. #1 네트워크 상의 스마트 에이전트는 IP 주소 199.10.9.5 를 사용하고, #2 네트워크 상의 스마트 에이전트는 101.10.2.6 을 IP 주소로 사용한다.



여기서 #1 의 스마트 에이전트는 #2 의 스마트 에이전트와 접속하기 위해서 agentaddr 이라는 파일을 이용하는데, 다음과 같은 줄을 포함하고 있으면 된다.

101.10.2.6

스마트 에이전트는 VBROKER\_ADM 환경 변수에 지정된 디렉토리에서 이 파일을 찾아서 연결하게 된다.

- 멀티-홈 호스트(multi-homed host)의 이용

스마트 에이전트를 하나 이상의 IP 주소를 가진 멀티-홈 호스트에서 실행하는 경우, 다른 네트워크 상에 있는 객체와의 브리징을 할 수 있는 강력한 방법이 있다. 이 경우에는 호스트와 연결된 모든 네트워크는 하나의 스마트 에이전트와 통신할 수 있게 되므로, agentaddr 파일은 필요하지 않다. 어쨌든 멀티 홈 호스트에서는 스마트 에이전트가 적절한 서브넷 마스크와 브로드캐스트 주소 값을 결정할 수 있다. 개발자는 이 값들을 localaddr 파일에 설정해 주어야 한다. 이 값들은 네트워크 환경 설정에서 TCP/IP 프로토콜의 설정 값을 이용하거나, 윈도우 NT 의 경우 ipconfig 명령을 이용하여 얻을 수 있다.

localaddr 파일에는 IP 주소, 서브넷 마스크, 브로드캐스트 주소의 조합을 포함하게 된다. 예를 들어, 다음의 리스트는 2 개의 IP 주소에 대한 스마트 에이전트의 localaddr 파일의 내용이다.

```

216.64.15.10 255.255.255.0 216.64.15.255
214.79.98.88 255.255.255.0 214.79.98.255
  
```

개발자는 OSAGENT\_LOCAL\_FILE 환경 변수에 localaddr 파일의 경로를 설정해 주어야 한다.

## 정 리 (Summary)

이번 장에서는 CORBA 에 대한 전체적인 개념과 텔파이 4 에서 지원하는 여러가지 클래스와 도구 들의 사용방법에 대해서 알아보았다. 새로운 기능이기 때문에 다소 어렵게 느껴질 수도 있지만, 분산 환경을 구현하는데 매우 강력하게 동작하므로 그 활용도는 높다고 할 수 있겠다.

## CORBA 의 개념과 활용 (II)

OMG(Object Management Group)라는 비영리 단체는 1989 년 4 월에 설립되었다. 이 단체는 현재 존재하는 객체지향 기술을 밑바탕으로 하여 프로그램들을 결합하기위한 산업 표준안을 제정하기 위해 600 개 이상의 컴퓨터 단련 단체 및 기업의 연합체로 구성되어 있다. 여기에서 객체지향 기술을 기반으로 이기종의 분산 환경을 지원하기위한 표준 기술을 제정하였는데 이 표준이 OMA(Object Management Architecture)이다.

이 기능 중에서 CORBA 는 컴퓨터 H/W 의 내부 버스처럼 프로그램 사이에서도 서로의 위치에 관계없이 서로를 사용할 수 있는 기능을 제공한다. 그러므로, CORBA 는 OMA 중에서도 가장 중요한 요소이다.

이번 장에서는 CORBA 에 대해 더 깊이 알아볼 것이다.

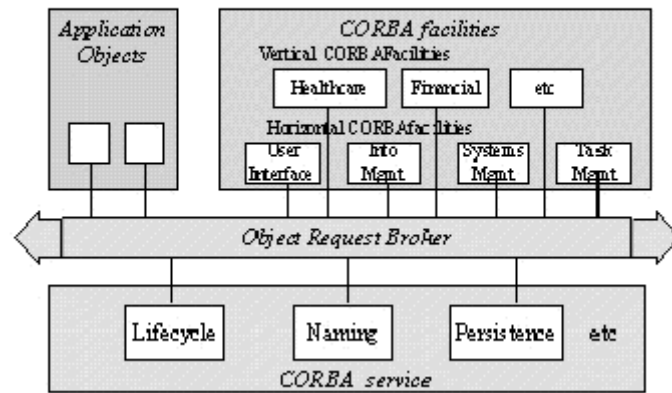
### OMA 와 ORB (Object Request Broker)

CORBA 에서는 ORB 가 가장 중요하다. 바로 이 ORB 때문에 만들어진 CORBA 간의 호환성 문제가 있다. 텔파이 4 에서는 비지제닉에서 제작한 비지 브로커(Visibroker)를 ORB 로 사용한다. 대표적인 ORB 로는 비지제닉의 비지 브로커 이외에도 아이오나의 오빅스가 있다. 그 밖에도 CORBAPLUS, FNORB, OMNI-ORB2, JTRADER, MICO, JOE, JYLU, JACORB, ELECTRA, ILU, OAK, DOME, CHORUS/COOL ORB 등의 여러 회사에서도 ORB 를 제작하고 있다.

OMA 는 객체를 작성하는 구조인데, 이 OMA 에는 분산환경에서 통신을 담당하는 CORBA 와 객체를 조작하는 기본기능을 정의한 COSS(Common Object Service Specification), 그리고 기타 기본 기능과 개발자가 작성한 각종 응용 객체들로 구성된다.

이 중에서 CORBA 서비스는 객체들이 연결하기 위한 기본기능을 제공하는데 이 기능에는 객체의 생명주기 서비스(Object Lifecycle Service), 명명 서비스(Naming Service), 디렉토리 서비스(Directory Service)등이 있고, 객체지향의 접근 방법인 온라인 트랜잭션 처리(On-line Transaction Processing)과 트레이더 서비스(Trader Service)등을 포함하고 있다. 이러한 CORBA 서비스는 객체의 통신과 연결에 필요한 서비스를 제공하고, CORBA Facilities 는 필요한 서비스를 제공한다. 대표적인 복합문서 관리(Compound Document Management) CORBA Facility 는 복합문서의 컴포넌트를 접근할 때 표준화된 방법을 지원한다. 이러한 CORBA Facilities 는 크게 두가지로 구분될 수 있는데, 하나는 Horizontal Facilities 로 복합문서 서비스가 이에 준하는 것으로 전반적인 영역에서 사용할 수 있는 서비스이고, 또 다른 하나는 Vertical Facilities 로 특정한 곳에만 사용하는 서비스를 말한다. 다음의 그림은 OMA 의 구조를 나타내고있다.

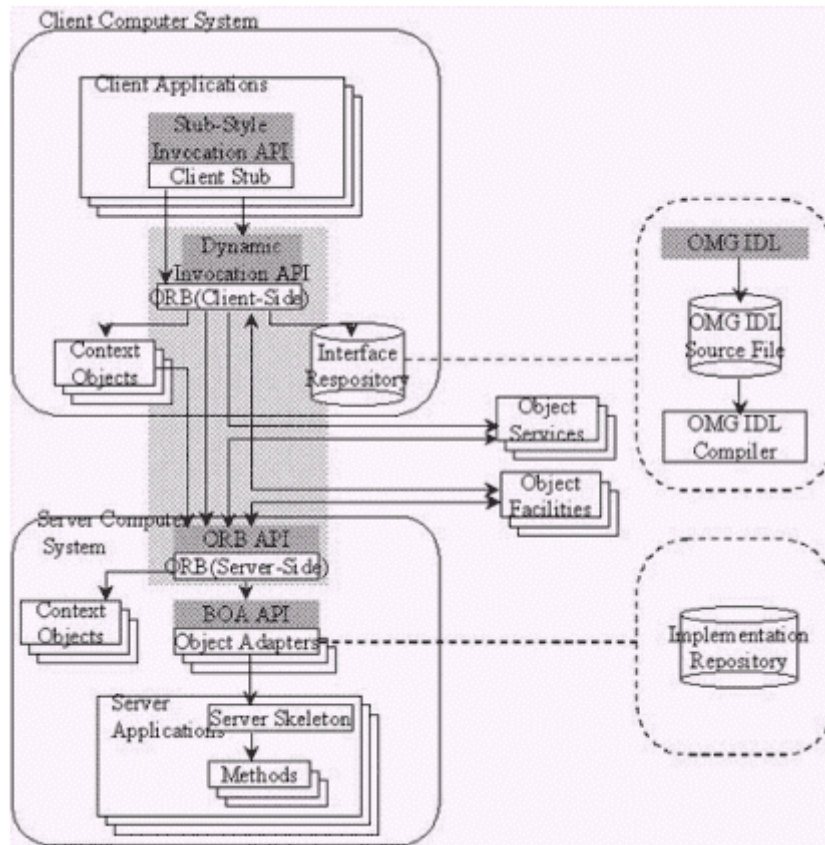
## OMA



OMG 는 1990 년 OMA 를 발표한 이래, 1997 년에 CORBA 스펙 2.1 을 발표했으며 현재 CORBA 3 규격까지 나와있다.

## CORBA 의 동작 원리

이번에는 CORBA 의 동작 원리에 대해서 자세히 살펴보자. 먼저 CORBA 의 구조는 전체적으로 다음 그림과 같이 표현할 수 있다.



해당 그림을 살펴보면서 CORBA 로 구현된 시스템의 호출 순서에 대해 알아보자.

- 클라이언트 프로그램은 OMG IDL 로 정의된 객체들의 동작 들에 대한 요청을 ORB 를 통하여 호출한다. 이때 호출하는 형태는 Stub-style invocation(스텝 호출), Dynamic invocation(동적 호출)이 있는데, 2 가지 방식을 혼용할 수 도 있다.

#### 1. 스텝 호출 (Stub style)

클라이언트 프로그램이 미리 만들어진 인터페이스를 통해 정의된 오퍼레이션에 대하여 링크 하는 방법이다.

#### 2. 동적 호출 (Dynamic style)

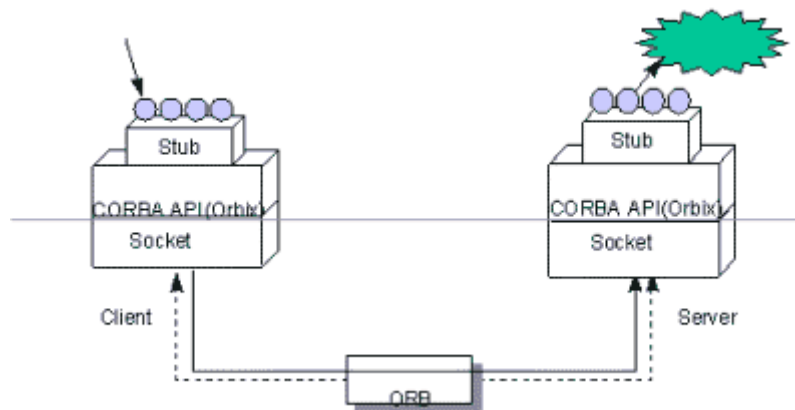
클라이언트 프로그램이 인터페이스 저장소를 통하여 동작하도록 실행 시간에 만들어질 것을 요구한다. 클라이언트와 서버는 ORB(Object Request Broker)를 사용하여 통신하므로 서로간의 정보는 필요가 없게 된다. 클라이언트는 ORB 에서 필요한 요청을 하고 ORB 는 필요한 요청을 서버의 구현에서 선택해서 해당 되는 메소드를 구현부에 보낸다.

컨텍스트 객체(Context Object)는 해당 행위로 전달되지 않는 정보나 서버에 대한 선택, 환경 등의 정보를 가지고 있다. 보통 클라이언트 어플리케이션에서 서버를 호출할 경우에 컨텍스트 객체를 참조하게 된다. 인터페이스 저장소는 네트워크 상에 있는 프로그램들과 데이터 객체에 대한 인터페이스를 모두 가지고 있다. 이 인터페이스 저장소는 동적 호출시에 필요한 정보를 제공한다.

## ● CORBA 의 서버 프로그램 구조

그 구조는 클라이언트 객체에서 요구를 할 경우, 이 요구를 받아들여 동작하기 위한 구현부(Implementation)를 하나 이상 가지고 있다. 이러한 요구를 클라이언트는 ORB 를 통해서 하게 되므로, ORB 는 BOA(Basic Object Adapter)를 사용하여 필요한 구현 내용을 선택하고 해당 구현 내용을 요구한다. 이때 BOA 는 해당 구현부 내부의 메소드 들을 호출하기 위해 서버 스켈레톤(Server Skeleton)을 사용한다.

객체 어댑터(Object Adapter)는 서버의 객체를 관리하는 중요한 도구인데, 이 역할은 구현부를 ORB 에서 호출할 수 있도록 만들어 주는 것이다. 일반적으로 이러한 객체 어댑터를 BOA 라고 부른다. 그러므로, 모든 CORBA 제작사는 그들의 시스템 일부분에서 BOA 를 지원해야 한다. 그러므로, 서버는 이러한 BOA 와 객체의 작업에 대한 메소드 들과의 연결을 제공하며 각각의 연결에 필요한 정보를 가지고 있는 것이다. 이상의 설명을 다음 그림과 같이 표현할 수 있다.



## IDL 에 대하여

IDL 은 Interface Definition Language 의 약자인데, 말 그대로 객체의 인터페이스를 정의하는 언어라는 뜻이다. 쉽게 이야기해서 텔파이에서 만들어진 클래스를 CORBA 에서 호출할 수 있도록 중간에서 연계해주는 언어이다.

텔파이에서는 타입 라이브러리 에디터를 이용해서 이러한 IDL 을 간단하게 만들 수 있다.

다만, 다른 언어에서 사용한다면 이러한 IDL 의 코딩 방법 썸에 대해서는 알고 있어야한다.

## ● IDL 문법

IDL 의 문법은 C++, 자바와 유사하다. 직접 에디터를 사용해서 만들 수 있으며, 이렇게 직접 제작한 텍스트 파일을 이용하여 스텝과 스켈레톤을 생성하려면 텔파이 4 의 BIN 디렉토리의 TLBIMP.EXE 파일을 이용하여 IDL 코드를 컴파일할 수 있다.

### 1. 주석 (Comment)

기본주석은 //, /\* .. \*/로 한 줄 주석과 여러 줄 주석을 사용할 수 있다.

### 2. 예약어

보통은 알고 있는 일반적인 C, 자바 등의 예약어와 동일하지만 약간 다른 부분만 설명하도록 하겠다.

예약어	내 용
any	C 나 C++의 void* 나 오브젝트 파스칼에서의 Variant 데이터 형처럼 데이터 타입만 받아들이는 것이 아니라 객체를 포함한 모든 타입의 변환이 가능한 데이터 형을 말한다.
in	클라이언트에서 구현부로만 전달되는 매개변수를 지칭한다.
out	구현부에서 클라이언트로만 전달되는 매개변수를 지칭한다.
inout	양방향 전달이 가능한 매개 변수를 지칭한다.
oneway	보통 메소드가 동기적으로 수행되는 반면에 비동기적으로 수행할 수 있는 메소드를 지칭하는 것으로 결과를 기다리지 않고 나중에 처리할 수 있는 메소드이다.

## ● 데이터 형 (data type)

IDL 에서 만들어내는 인터페이스에서 해당 작업의 매개 변수나 반환되는 값의 데이터 형태를 지정하는데 사용한다.

- 기본 데이터 타입 (char, short, long, float)
- 구조체 타입 (struct, union, enumeration)
- 템플릿 타입 (sequence, array, string)

물론 앞에서 설명한 any 데이터 형도 가능한데, 이를 사용하면 수행 속도가 느려지는 것은



두말할 나위도 없다.

## 1. 구조체 데이터 형 (Constructed Types)

구조체 데이터 형에는 3 가지가 있다. struct, union, enumeration 의 3 가지가 있다.

### - structure

일반적인 구조체와 동일하다.

```
struct 구조체명 {  
    데이터타입 변수명;  
    데이터타입 변수명;  
}
```

### - union

일반적인 공용체와 동일하다. 서로 다른 데이터 형과 크기를 공유하기 위한 방법이 있을 경우에 사용한다. C(++)와 다르게 각 데이터 멤버는 case 라벨과 함께 선언한다.

```
union token switch (long) {  
    case1: char 변수명;  
    case2: float 변수명;  
    default: long 변수명;  
};
```

### - enumeration

데이터 멤버를 순서적으로 나열할 경우에 사용한다.

```
enum workday {Monday, Tuesday, Wednesday, Thursday, Friday};
```

## 2. 템플릿 데이터 형 (Template types)

enumeration 과 string 두 종류의 템플릿 타입을 제공한다.

- enumeration

일차원 배열을 선언하는 데이터 형이며 최대 크기가 지정되어 있으면 바운드(bounded) 시퀀스라 하고, 최대 크기가 없으면 언바운드(unbounded) 시퀀스라고 한다. 해당 길이는 동적으로 변경되며 선언할 수 있는 데이터 형은 IDL의 모든 데이터 형이 가능하다.

- string

시퀀스와 동일한 방법으로 사용하는데, 시퀀스와 다른점은 문자형 만을 사용할 수 있다는 점이다.

### 3. 배열 (array)

다차원 배열을 선언하는 경우에 사용한다.

### 4. 상수형 (constant type)

일반적으로 사용되는 상수와 동일하다. 사용할 수 있는 데이터 형으로는 boolean, short, char, string, double, float, long, unsigned long, unsigned short 등이 있다.

### 5. 인터페이스 (interface)

클라이언트에서 서버 객체에 서비스를 요청할 때에 처리해주는 속성과 작업을 정의한 세트를 말한다.

#### ● 작업 (operation)

인터페이스의 몸체에 선언되는 것으로 구성은 작업 매개 변수, 예외처리(exception), 반환값의 형태, 매개변수 전달방향 등으로 구성된다.

#### 1. 예외 처리

서비스 요청 시에 예외상황이 발생할 경우 클라이언트에 전달하는 자료구조로 보통은 시스템이 미리 정한 예외처리와 개발자가 설정한 예외처리의 두 가지 종류가 있다. 주의할 점은 이 예외처리에는 사용자가 정한 작업의 매개변수나 데이터 형이 될 수 없다. 그리고 예외처리를 위한 raises 라는 키워드가 있다.

## 2. 컨텍스트

구현 객체의(implementantation object)의 작업에 영향을 줄 수 있는 클라이언트의 환경 요소와 관련된 리스트를 컨텍스트라 한다. 이 컨텍스트를 사용하기 위해서는 raises 문장 바로 뒤에 context(context1, context2, ...) 형태로 선언하여 주기만 하면 된다. 그러나 이 문법은 가능한 사용하지 않는 것이 좋다.

## 3. Oneway 작업(operation)

Oneway 란 작업을 호출하고 결과를 기다리지 않는 것을 선언하는 것으로, 클라이언트에서 호출만 하고 결과에 대해서는 책임지지 않는 호출방법을 의미한다. 다만, 이 Oneway 작업은 다른 작업보다 먼저 선언되어야 하는 점만 주의하자.

## 4. 속성 (Attribute)

속성은 클라이언트가 어떤 변수에 값을 설정하거나 검색할 때에 쉽게 IDL 로 표현하기 위해서 만들어진 형식이다. 속성은 readonly 를 사용할 수 있고, 해당 값을 설정하고 읽어오는 두가지 형태의 일만 수행한다. 간단히 보면 변수 선언하는 것과 비슷하나 실제로는 일종의 작업(operation)이다.

### ● 상속 (Inheritance)

일반적인 객체지향 언어(?)라면 상속이 지원되는 것이 당연할 것이고, 이 상속개념을 사용하면 한번 만들어진 IDL 을 사용하여 새로운 파생 인터페이스를 만들어 낼 수 있다.

이때 파생되는 인터페이스는 간접(indirect)과 직접(direct)기반 인터페이스로 구분된다.

### 1. 간접 기반 인터페이스

만들어진 파생 인터페이스의 기반 인터페이스가 다른 기반 인터페이스에서 파생된 인터페이스일 경우를 말한다.

### 2. 직접 기반 인터페이스

만들어진 파생 인터페이스의 기반 인터페이스가 기본적으로 만들어진 인터페이스일 경우를 말한다.

파생 인터페이스를 만들 경우에 동일한 작업이나 속성 이름을 가지고 있는 기반 인터페이스로부터 상속 받을 수 없고, 또한 상속받은 작업이나 속성의 이름을 재정의 할 수 없다. 이 점이 일반적인 상속과 다른 점이다.

## ● 모듈(Module)

오브젝트 파스칼의 유닛 개념과 동일한 것으로, namespace 를 제공하기 위해 IDL 의 범위를 설정하는 것이다. 이때 외부의 식별자를 사용하기 위해서는 '::'(name resolution operator)앞에 모듈 이름을 붙여 사용한다.

## 비지브로커(Visibroker)

텔파이 4 에서 지원되는 CORBA 의 ORB 는 비지제닉사의 비지브로커이다. 현재 개발된 CORBA 의 ORB 중 다양한 지원과 강력한 성능을 기대할 수 있는 ORB 이다.

현재 지원되는 언어는 자바와 C++, 그리고 텔파이이다. 그리고 자바 표준의 JDBC 와 연결하기 위한 Visichannel for JDBC 도 지원한다. 그리고 여러가지 서버도 지원하는데 VisiBroker Event Service, VisiBroker Naming Service, VisiBroker Productivity Tools 등의 다양한 서비스도 지원하며 새롭게 VisiBroker ITS (트랜잭션 서비스) 도 지원한다.

넷스케이프사에서는 인터넷에서의 강력한 C/S 기반 어플리케이션을 위해 CORBA 자바 기술 이용에 관한 인증 체결에 대한 기술을 지원한다. 그리고, 오라클사에서는 비지제닉의 CORBA 관련 분산객체기술 인증을 체결하여, 이 기술과 함께 객체, 자바 기술, 인트라넷에 적용하여 어플리케이션을 개발하고 있으며, 노벨사에서는 인트라넷웨어 서버 플랫폼에서 비지브로커 사용에 관한 인증을 체결하였다. CORBA, IIOP 관련 개발자 지원과 인트라넷웨어에서 분산 어플리케이션 배치를 가능하게 하고 있다. 사이베이스에서는 비지제닉사의 비지브로커와 관련 기술에 관한 인증을 체결하여, 사이베이스의 새로운 트랜잭션 서버에서 자바, C++ 관련 비지브로커 제품군을 지원한다.

텔파이를 설치하면 다음의 4 가지 유틸리티가 설치 된다.

프로그램 명	내 용
Object Activation Daemon	클라이언트에서 서버를 호출할 경우 자동으로 동작시켜 주는 프로그램
Visibroker Reg-Edit tool	ORB 의 설정 및 자바 머신의 환경 등을 등록한다.
Visibroker SmartAgent	실제 ORB 를 구성하는 메인 모듈, 이 모듈이 동작해야 CORBA 의 ORB 가 동작한다.
Visibroker SmartFinder	연결된 HOST 및 네트워크에 존재하는 CORBA 모듈을 검색한다.

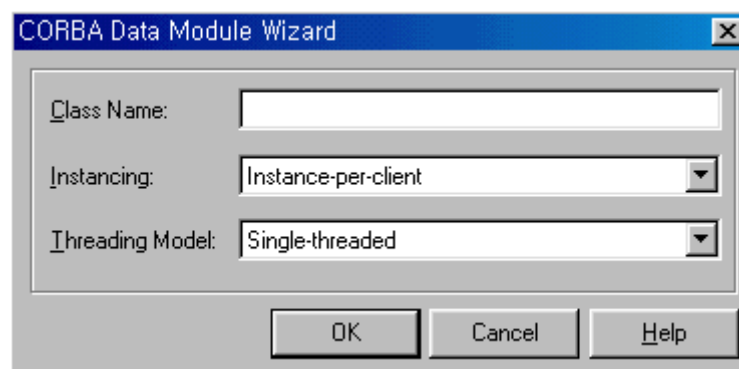
이중에서 가장 중요한 Smart Agents 는 분산환경을 지원하기 위한 ORB 의 기본적인 구성을 지원한다. 실제 구동은 point-to-point 의 UDP 프로토콜을 사용하므로 일반적인 TCP/IP 환경이 지원되는 곳에서 사용한다.

## CORBA 서버의 제작과 구동

ORB 에서 클라이언트가 서버를 호출할 경우에 서버가 자동으로 동작하게 하는 유틸리티는 비지브로커에서 OAD 를 사용하여 구동한다. 이 OAD 는 네트워크의 한 서버에서만 동작하고 있으면 자동적으로 동작한다.

텔파이에서 CORBA 서버를 만드는 방법은 CORBA 데이터 모듈 위저드를 이용하거나, CORBA 객체 위저드를 사용하여 만드는 2 가지 방법이 있다.

File|New 메뉴의 Multitier 탭에서 CORBA DataModule 을 더블 클릭하면 다음의 화면이 나타난다.



여기서 Class Name 에 데이터 모듈의 이름을 입력한다음 ‘OK’를 선택하면 기본적인 CORBA 데이터 모듈을 만들 수 있다. 여기서는 ‘Test’로 이름을 입력하도록 하자. 그리고, 만들어진 데이터 모듈의 유닛을 각각 U\_ExamSvr1.pas, ExamSvr1\_TLB.pas, U\_ExamSvrImpl1.pas 로 저장하고, 프로젝트 파일을 ExamSvr1.dpr 로 저장하도록 하자. 이중에서 U\_ExamSvr1 은 일반적인 어플리케이션을 수행하기 위한 빈 유닛이다. 이것은 별다른 점이 없다.

ExamSvr1\_TLB.pas 유닛은 CORBA IDL 에 해당되는 파스칼 소스 코드이다. 실제 코드를 자동으로 생성시켜 주므로 개발자는 타입 라이브러리 에디터에서 필요한 메소드나 모듈, 속성 등을 만들면 자동적으로 IDL 로 전환되며, 스텝과 스켈레톤을 지원하기 위한 코드를 생성한다. U\_ExamSvrImpl1 는 타입 라이브러리 소스 코드에 해당되는 클래스를 구현하는 유닛이다. 아마도 다음과 같은 유닛이 생성되었을 것이다.

```
unit U_ExamSvrImpl1;
```

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
ComObj, VCLCom, StdVcl, BdeProv, DataBkr, CorbaRdm, CorbaObj,  
ExamSvr1\_TLB;

type

TTest = class(TCorbaDataModule, ITest)

private

{ Private declarations }

public

{ Public declarations }

end;

var

Test: TTest;

implementation

{\$R \*.DFM}

uses Corblnit, CorbaVcl;

initialization

TCorbaVclComponentFactory.Create('TestFactory', 'Test', 'IDL:ExamSvr1/TestFactory:1.0', ITest,  
TTest, iMultInstance, tmSingleThread);

end.

만들어진 TTest 클래스는 TCorbaDataModule 를 상속받아 개발자가 선언한 ITest 인터페이스를 구현한 CORBA 데이터 모듈이다.

이렇게 만들어진 인터페이스는 인터페이스 저장소에 등록되며 보통은 해당 서버 어플리케이션을 한번 실행하면 해당 내용이 자동으로 등록된다.

이때 만들어진 CORBA 서버가 클라이언트에서 호출할 때에 자동으로 동작하게 하려면 OAD 를 사용하여 등록하여야 한다.

## 1. OAD 를 사용하여 등록하기

irep IRname 을 DOS 커맨드에서 실행해 보자. 이 프로그램은 인터페이스 저장소 역할을 한다. 이곳에서 IDL 을 로드하여 필요한 인터페이스를 인터페이스 저장소에 저장할 수 있다. 지원되는 것은 일반 IDL 과 자바, C++ 을 지원한다. 단순히 커맨드 모드에서 사용하려면 irep -console IRname [file.idl]을 사용하여도 무방하다.

텔과이 4 의 Demo 중에 CORBA 서브 디렉토리의 DataModule 서브 디렉토리에 있는 프로젝트 파일을 읽은 다음 CORBAServer\_TLB.pas 를 CORBAServer.idl 로 출력하고, 해당 IDL 을 등록해 보자.

irep IRname CORBAServer.idl 이라고 DOS 명령을 사용하면 인터페이스 저장소에 인터페이스가 등록된다. 이렇게 만들어진 IDL 을 다시 배치하려면 idl2ir 유틸리티를 사용하게 된다. 사용법은 다음과 같다.

```
idl2ir -ir IRname -replace file.idl
```

그러므로 idl2ir -ir IRname CORBAServer.idl 을 수행하면 다음의 화면과 같이 인터페이스 저장소에 등록될 것이다.

```
C:\WINDOWS>idl2ir -ir IRname CORBAServer.idl
Exception in thread "main" org.omg.CORBA.NO_IMPLEMENT[completed=MAYBE, reason=
  Could not locate the following object:
    repository id : IDL:visigenic.com/tools/ir/RepositoryManager:1.0
    object name : IRname
]
    at com.visigenic.vbroker.orb.ORB.bind(ORB.java:1276)
    at com.visigenic.vbroker.orb.ORB.bind(ORB.java:1358)
    at com.visigenic.vbroker.orb.ORB.bind(ORB.java:1166)
    at com.visigenic.vbroker.tools.ir.RepositoryManagerHelper.bind(RepositoryManagerHelper.java:72)
    at com.visigenic.vbroker.tools.ir.RepositoryManagerHelper.bind(RepositoryManagerHelper.java:69)
    at com.visigenic.vbroker.tools.idl2ir.main(idl2ir.java:69)
```

이제 OAD 를 살펴보자. OAD [옵션]의 형태로 수행하는데, 대표적인 옵션으로 -C 옵션을 사용하면 NT 서비스로 등록하여 사용할 수 있다. 다음은 OAD 에서 사용 가능한 옵션이다.

파라미터	내 용
-v	verbose 모드로 동작한다.
-f	다른 호스트로 OAD 가 있는지 확인하여 동작한다.
-t<n>	해당 n 초만큼 지연된 다음에 수행한다.
-C	NT 서비스로 구동된다.
-k	연결된 프로세스 들의 객체를 해제한다.
-?	도움말을 살펴본다.

그러면, OAD 에 해당 서버 인터페이스를 등록하는 방법에 대해서 알아 보자. oadutil 이라는 프로그램을 사용하는데 다음과 같은 형태로 실행하면 된다.

```
oadutil reg -r IDL:내서버/내객체:1.0 -o 내객체 -cpp 내서버.exe -p unshared
```

입력하는 내용은 CORBAServer\_TLB.Pas 의 initialization 부분을 참고하면 된다. 앞서 등록한 CORBA 서브 디렉토리의 DataModule 서브 디렉토리에 있는 CORBAServer 프로젝트의 initialization 섹션은 다음과 같다.

initialization

```
TCorbaVclComponentFactory.Create('DemoCORBAFactory', 'DemoCORBA',  
    'IDL:CORBAServer/DemoCORBAFactory:1.0', IDemoCORBA, TDemoCORBA, iMultilInstance,  
    tmSingleThread);
```

일단 컴파일을 하고 만들어진 CORBAServer.exe 파일을 적당한 디렉토리로 이동한 뒤에, 다음과 같이 등록하도록 하자.

```
oadutil reg -r IDL:CORBAServer/DemoCORBAFactory:1.0 -o DemoCORBAFactory -cpp 파일의 위치  
디렉토리\CORBAServer.exe -p unshared
```

제대로 등록되었으면 다음 화면과 같은 내용이 나타날 것이다. 주의할 점은 이와 같은 등록을 할 때 비지브로커 스마트 에이전트와 OAD 가 동작하는 상태에서 등록하여 주어야 한다.

```
C:\#Tmp>oadutil reg -r IDL:CORBAServer/DemoCORBAFactory:1.0 -o DemoCORBAFactory -  
cpp c:\#Tmp\CORBAS~1.exe -p unshared  
oadutil reg: Executable path 'c:\#Tmp\CORBAS~1.exe' invalid for OAD on host '166  
.104.81.101'  
  
C:\#Tmp>oadutil reg -r IDL:CORBAServer/DemoCORBAFactory:1.0 -o DemoCORBAFactory -  
cpp c:\#Tmp\CORBAS~1.exe -p unshared  
Completed registration of repository_id      = IDL:CORBAServer/DemoCORBAFactory  
:1.0  
object_name      = DemoCORBAFactory  
reference data    =  
path_name        = c:\#Tmp\CORBAS~1.exe  
activation_policy = UNSHARED_SERVER  
args             = NONE  
env              = NONE  
for OAD on host 166.104.81.101
```

CORBA 클라이언트



일단 CORBA 서버를 만들었으면, 이를 인터페이스에 저장하여야 한다. 그리고, 이를 클라이언트에서 사용하는 방법은 CORBA 서버를 만들 때 생성된 \_TLB.pas 파일을 uses 절에 추가해서 사용하면 된다.

예를 들기 위해서 앞서 사용한 CORBA 서버 예제의 클라이언트 프로젝트를 열어보도록 하자. 그러면, CORBAServer\_TLB.pas 유닛이 서버에서와 같이 사용되었다는 것을 알 수 있는데, 개발자는 이와 같이 필요한 인터페이스를 만든 다음 간단하게 해당 프로그램에 등록하여 사용하기만 하면 되는 것이다.

실제 CORBAServer\_TLB.pas 의 initialization 부분을 살펴보자.

initialization

```
CorbaStubManager.RegisterStub(IDemoCORBA, TDemoCORBAStub);  
CorbalInterfaceIDManager.RegisterInterface(IDemoCORBA, 'IDL:DemoCORBA:1.0');  
CorbaSkeletonManager.RegisterSkeleton(IDemoCORBA, TDemoCORBASkeleton);
```

해당 코드는 스텝과 스켈레톤을 정의하여 주며 IDManager 에 해당 CORBA 데이터 모듈을 등록하는 코드를 자동으로 만들어 준다.

그렇다면 CORBA 의 호출 방법중에 동적으로 호출하는 방법에 대해서 알아보자.

## ● 동적호출

DII(dynamic interface invocation)를 사용하는 방법은 해당 서버의 객체를 호출하기 위한 스텝 클래스를 마샬링한 인터페이스에서 찾아와야 한다. 이때 찾을 인터페이스는 앞에서 설명한대로 인터페이스가 인터페이스 저장소에 미리 등록되어 있어야 한다.

보통 호출하는 경우에는 Any 를 사용하여 호출하는 것이 안전하다.

var

```
InCall : TAny;
```

begin

```
InCall := CorbaBind( 'IDL:MyServer/MyServerObject:1.0' );
```

이렇게 코딩을 하면 필요한 인터페이스의 스텝을 알 수 있다.

실제 TAny 형을 찾아보면 델파이의 Variant 데이터 형임을 알 수 있다. 델파이 도움말에서 CorbaBind 에 대한 예제를 살펴보면, CorbaBind 를 사용해서 동적으로 호출하는 DII 를 잘 알 수 있다.

var

```
HR, Emp, Payroll, Salary: TAny;
```

```
begin
    HR := CorbaBind('IDL:CompanyInfo/HR:1.0');
    Emp := HR.LookupEmployee(Edit1.Text);
    Payroll := CorbaBind('IDL:CompanyInfo/Payroll:1.0');
    Salary := Payroll.GetEmployeeSalary(Emp);
    Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit2.Text) / 100));
end;
```

CorbaBind 를 통하여 원하는 IDL 인터페이스를 찾아서 해당 메소드를 사용할 수 있다.

## 정 리 (Summary)

이번 장에서는 CORBA 의 동작 원리에 대해서 더 깊숙히 알아보고, IDL 문법과 실제 CORBA 서버를 제작한 뒤에 어떻게 인터페이스를 등록하고 사용할 수 있는지에 대해서 알아보았다. 이것으로 제 5 부의 내용을 마치게 된다. 다음에 이어지는 제 6 부에서는 인터넷 과 통신에 대해서 알아볼 것이다.

## 웹서버 어플리케이션의 제작

텔파이를 이용하여 실제 인터넷에서 사용할 수 있는 어플리케이션을 개발하는 방법에는 전통적인 CGI 형식의 프로그램을 개발하는 방법과 액티브 X 기술을 기반으로 하는 방법으로 크게 나누어 볼 수 있다. 여기에 CORBA 와 MTS 기반 기술을 미들웨어로 하여 보다 효과적인 CGI 또는 액티브 X 컨트롤을 제작할 수 있을 것이다.

이번 장에서는 이 중에서도 CGI 를 기반으로 한 웹 서버 어플리케이션을 작성하는 방법에 대해서 알아볼 것이다.

### CGI 어플리케이션의 종류

텔파이에서 만들 수 있는 단순한 CGI 어플리케이션은 하드 코딩에 가깝다고 볼 수 있다. 실제로 텔파이의 GUI 디자인을 이용하여 그 폼을 그대로 표현할 수 있다면 좋겠지만, 이는 액티브 폼을 이용하지 않고서는 구현이 불가능하다.

텔파이에서 만들 수 있는 CGI 의 형식에는 다음과 같은 것들이 있다.

어플리케이션 종류	어플리케이션 객체	Request 객체	Response 객체
MS Server DLL (ISAPI)	TISAPIApplication	TISAPIRequest	TISAPIResponse
Netscape Server DLL (NSAPI)	TISAPIApplication	TISAPIRequest	TISAPIResponse
Console CGI Application	TCGIApplication	TCGIRequest	TCGIResponse
Windows CGI Application	TCGIApplication	TWinCGIRequest	TWinCGIResponse

표에서 보듯이 ISAPI 와 NSAPI 를 지원함으로써 양대 웹 서버의 API 를 모두 지원하며, 동시에 Console CGI 나 Windows CGI 도 지원하기 때문에, 텔파이를 CGI 개발 도구로 사용할 수 있다.

### 텔파이로 CGI 어플리케이션을 작성하는 이유

현재 웹 서버 어플리케이션을 제작하는 방법은 다음과 같이 크게 3 가지로 나눌 수 있다.

1. 웹 서버에서 지원되는 기본적인 스크립트를 사용하는 방법이다. 대표적으로 ASP, VB 스크립트와 자바 스크립트, Perl 등이 있다.
2. ISAPI 나 NSAPI 를 통한 CGI 어플리케이션을 제작하는 방법
3. 자바를 사용하여 서블릿이나 애플릿 형태로 개발하거나, 액티브 X 컨트롤의 형태로 클라이언트에서 동작하는 작은 프로그램을 제작하는 방법

물론, 한가지 방법으로 웹 서버의 모든 기능을 만족시킬 수도 있다. 그러나, 프로그램의 성능과 보안문제등 다양한 문제를 해결하기 위해서는 이러한 방법들을 혼용하여야 한다.

여기에서 CGI 로 제작되는 방식은 이미 구세대적인 방식인지는 모르나 현재 시점에서 DB 에 접근하여 작업하는 경우에는 이 방법이 가장 효율적이다.

텔파이는 이러한 CGI 를 제작하는 경우에 매우 효율적인 방법을 제공하는데, 그것은 CGI 의 4 가지 방식을 따로 구분할 필요 없이 한가지 방식으로 개발한 다음 필요한 내용을 링크하여 CGI 를 빌드하기만 하면 된다는 것이다.

## 기본적인 Console CGI 어플리케이션의 제작

텔파이의 Demos 디렉토리의 WebServ 서브 디렉토리에 있는, Iservcgi 프로젝트를 기준으로 설명하도록 한다. 해당 소스를 컴파일한 다음 해당 프로그램을 IIS 의 CGI-Bin 디렉토리에 복사한 다음 해당 내용을 브라우저하여 보면, 텔파이에서 기본적인 데이터를 보여주는 것 뿐만 아니라 테이블의 표를 HTML 형식으로 표시할 수 있도록 지원할 수 있다는 것을 알 수 있을 것이다. 실제 CGI 어플리케이션을 적용한 이 예제를 브라우저를 이용하여 접근할 때, CGI 가 동작할 링크 부분에 커서를 가지고 가면 다음과 같은 형태의 코드가 나타나는 것을 볼 수 있을 것이다.

```
실행파일.exe/runquery?custono=1645
```

이 코드는 CGI 에 파라미터로 runquery 와 custono=1645 라는 코드를 전달한다는 의미이다. CGI 어플리케이션을 구동시키기 위해서는 실행 파일을 단독으로 수행하는 방법과 이렇게 파라미터를 이용하여 전달하는 방법이 있다.

이렇게 파라미터를 처리하는 방법을 Get 메소드라고 하는데, 여기에 대해서 먼저 알아보도록 하자.

실제 해당 소스의 웹 모듈을 살펴보면 여러 테이블과 쿼리가 있다는 것을 알 수 있다. 웹 모듈을 선택하고, Actions 프로퍼티의 ‘...’ 버튼을 클릭하면 다음과 같은 화면을 볼 수 있을 것이다.

Name	PathInfo	Enabled	Default
CustomerList	/customerlist	True	
QueryAction	/runquery	True	
Redirect	/redirect	True	
GetImage	/getimage	True	
BioLife	/biolife	True	
root		True	*

웹 모듈의 파라미터와 해당 Action 의 연결 상태를 알 수 있다. 개발자는 이 Action 에서 필요한 파라미터를 설정할 수 있고, 해당 파라미터와 연결되는 코드를 이벤트 추가하듯이 추가하는 것이 가능하다.

이제 해당 Action 부분의 코드가 어떻게 만들어졌는지 살펴 보자.

기본적으로 수행되는 이벤트는 root 라는 디폴트로 설정된 Action 으로, 이 Action 은 해당 CGI 가 동작할 때에 동작하는 이벤트이고, 여기에서 파라미터에 /runquery 가 전달되면 동작하는 코드는 QueryAction 이벤트이다. 그 밖에 다른 Action 도 많이 있지만, 이 중에서 CustomerList 와 QueryAction 에 대해서 알아보도록 하자.

먼저 WebModule1CustomerListAction 의 코드를 살펴보자.

```
procedure TCustomerInfoModule.WebModule1CustomerListAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := CustomerList.Content;
end;
```

해당 코드는 Response 객체의 Content 부분에 CustomerList.Content 를 전달하는 단순한 코드이다. 그렇다면 CustomerList 는 무엇일까 ? CustomerList 는 다음과 같이 선언되어 있다.

```
CustomerList: TPageProducer;
```

이 PageProducer 의 Content 를 Response 의 Content 에 넣는 단순한 코드로 일견 복잡해 보이는 HTML 화면을 만들어 낸 것이다.

CustomerList 페이지 프로듀서 컴포넌트는 필요한 HTML 파일을 가지고 있다가 해당 내용을 전달한다. 프로퍼티를 살펴보면 Strings 객체의 HTMLDoc 와 HTML 파일의 위치를 가리키는 프로퍼티인 HTMLFile 프로퍼티가 있다. 둘 중 하나의 값으로 내용을 유지한다.

개발자는 해당 내용을 HTMLDoc 프로퍼티에 직접 넣을 수 있으며, 해당 파일을 호출할 수

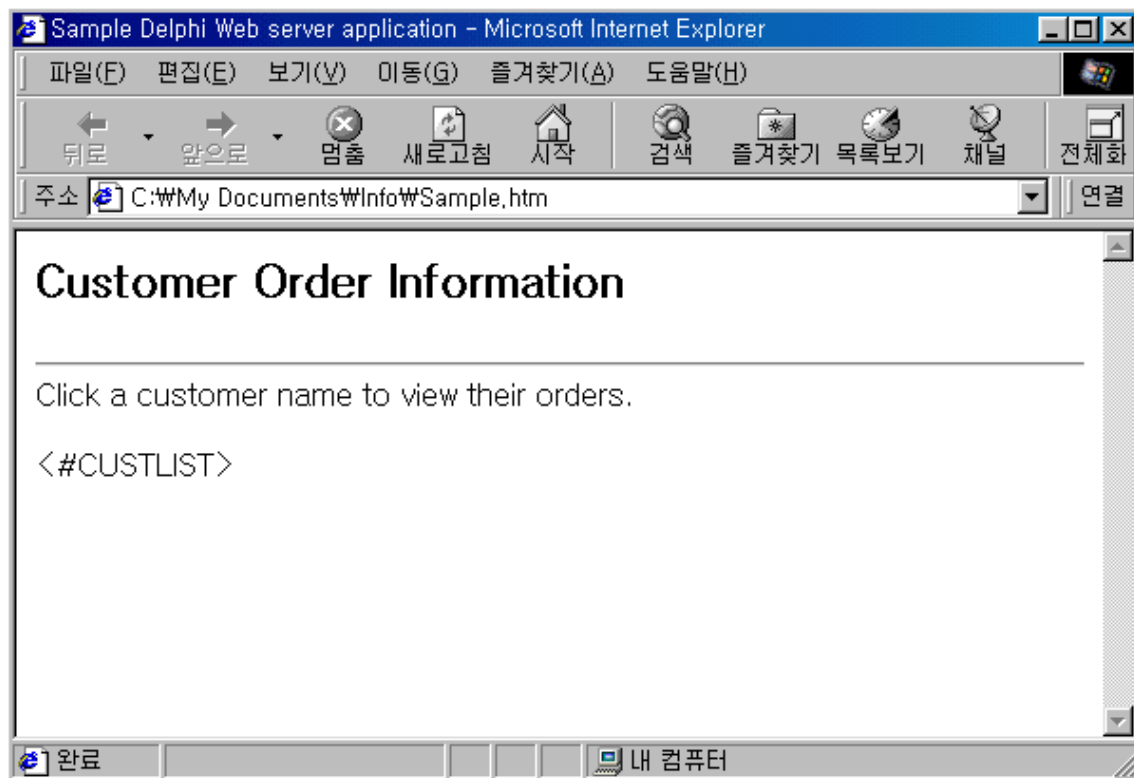
도 있다.

참고로, HTML 파일의 위치를 HTMLFile 프로퍼티에 지정하는 것이 나중에 프로그램을 수정할 때에 좀더 효율적으로 사용할 수 있다. 그러나, 이 방식은 수행 속도를 저하시키는 단점이 있다. 그러므로, 높은 수행 속도를 요구한다면 해당 내용을 직접 HTMLDoc 프로퍼티에 넣어서 사용하는 것이 좋다.

그렇다면, 해당 HTMLDoc 프로퍼티에 들어 있는 내용을 살펴보자.

```
<HTML>
<!------->
<!-- Copyright Inprise Corporation 1998 -->
<!------->
<HEAD>
<TITLE>Sample Delphi Web server application</TITLE>
</HEAD>
<BODY>
<H2>Customer Order Information</H2>
<HR>
Click a customer name to view their orders.<P>
<#CUSTLIST><P>
</BODY>
</HTML>
```

HTML 태그에 대해 조금만 알고 있다면 이 화면이 다음과 같이 보일 것이라는 것을 알 수 있을 것이다. 여기서 주의해서 보아야 하는 것은 중간에 있는 <#CUSTLIST> 부분이다. 이 부분은 일종의 스크립트로서 해당 부분을 페이지 프로듀서가 채우는 것이다.



해당 내용을 채우는 것은 OnHTMLTag 이벤트에서이다. 여기에 해당되는 코드는 다음과 같다.

```

procedure TCustomerInfoModule.CustomerListHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
var
  Customers: String;
begin
  Customers := '';
  Customer.First;
  while not Customer.Eof do
  begin
    Customers :=
      Customers + Format('<A HREF="/scripts/%s/runquery?CustNo=%d">%s</A><BR>',
        [ScriptName, CustomerCustNo.AsInteger, CustomerCompany.AsString]);
    Customer.Next;
  end;
end;

```

```

end;

ReplaceText := Customers;

end;

```

이 코드의 의미는 전달되는 태그의 값에서 해당 부분을 교체하는 것이다. TagString 부분으로 해당 태그를 읽어서 ReplaceText 영역에 해당 내용을 넣으면 해당 부분을 교체하게 된다. 그럼 변환한 내용은 무엇인가 ?

```

Customers :=
    Customers + Format('<A HREF="/scripts/%s/runquery?CustNo=%d">%s</A><BR>',
        [ScriptName, CustomerCustNo.AsInteger, CustomerCompany.AsString]);

```

이 부분은 필요한 부분의 링크를 만드는 코드인데, HTML 의 링크를 만드는 문장을 통하여 해당 파라미터와 코드 값을 주어 문장을 만들어 낸다.

이중에 ScriptName 은 수행시켜야 할 프로그램을 지정한 부분이다. 일반적으로 CGI 어플리케이션이 어느 곳에 위치할 지 모르기 때문에, 이를 고정시키면 매번 프로그램을 수정해야 한다. 그렇기 때문에, 이렇게 웹 모듈을 생성하는 시점에서 해당 프로그램의 위치를 읽어 내는 코드를 만드는 것이 좋다.

소스 코드를 잘 살펴보면 public 섹션에 ‘ScriptName: String;’이라고 선언된 부분을 찾을 수 있을 것이다. 그리고, 이 내용은 웹 모듈의 OnCreate 이벤트에서 해당 위치를 지정하도록 다음과 같이 작성되어 있다.

```

procedure TCustomerInfoModule.DataModule1Create(Sender: TObject);
var
    FN: array[0..MAX_PATH- 1] of char;
begin
    Customer.Open;
    BioLife.Open;
    SetString(ScriptName, FN, GetModuleFileName(hInstance, FN, SizeOf(FN)));
    ScriptName := ExtractFileName(ScriptName);
end;

```

코드는 이와 같다. 먼저 Customer 테이블 파일을 연 뒤에, GetModuleFileName 함수를 이용하여 현재 이 프로그램의 이름을 읽어 낸다. GetModuleFileName API 함수는 현재 프로그램의 인스턴스에 해당되는 프로그램의 시작 패스를 읽어서 FN 문자 배열에 저장한다. 그리고, ‘SetString(var s: string; buffer: PChar; len: Integer);’ 코드는 PChar 로 되어 있는



내용을 문자열 형태로 변환하여 저장하는 함수이다. 그 다음에 ExtractFileName 함수를 이용하여 원하는 패스와 프로그램 명을 분리한다.

이 코드를 이용하면 실제 수행 시의 프로그램의 위치를 저장하고, 링크를 생성하는 코드에서 해당 ScriptName 을 사용하게 된다. 여기서 생성되는 링크를 웹 브라우저에서 클릭하면 QueryAction 이 동작하게 된다.

앞서 CustomerList 페이지 프로듀서에 의해 생성되는 하이퍼 링크 태그는 다음과 같은 형태를 가진다.

```
<A HREF="/scripts/실행파일 패스/runquery?CustNo=고객 번호"> 고객의 회사 </A><BR>',
```

이를 클릭하면 파라미터로 전달 되는 것이 CustNo 필드에 대한 값이다. 그러므로, CustNo 뒤의 번호를 통하여 데이터에 대한 정보를 찾아다가 이를 보여주는 역할을 하는 것이 QueryAction 이다. 그러므로, PathInfo 의 내용이 파라미터로 날아오는 코드이다. 소스 코드는 다음과 같이 작성되어 있다.

```
procedure TCustomerInfoModule.WebModule1.QueryActionAction(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
    if Customer.Locate('CustNo', Request.QueryFields.Values['CustNo'], []) then
    begin
        CustomerOrders.Header.Clear;
        CustomerOrders.Header.Add('The following table was produced using
        a TDataSetTableProducer.<P>');
        CustomerOrders.Header.Add('Orders for: ' + CustomerCompany.AsString);
        Response.Content := CustomerOrders.Content;
    end
    else
        Response.Content := Format('<html><body><b>Customer: %s not found</b></body></html>',
        [Request.QueryFields.Values['CustNo']]);
end;
```

약간 복잡하지만 차근차근 살펴 보자. Customer.Locate 메소드를 이용하여 넘어온 파라미터 번호에 해당되는 레코드를 찾는다. Locate 문의 선언부는 다음과 같다.

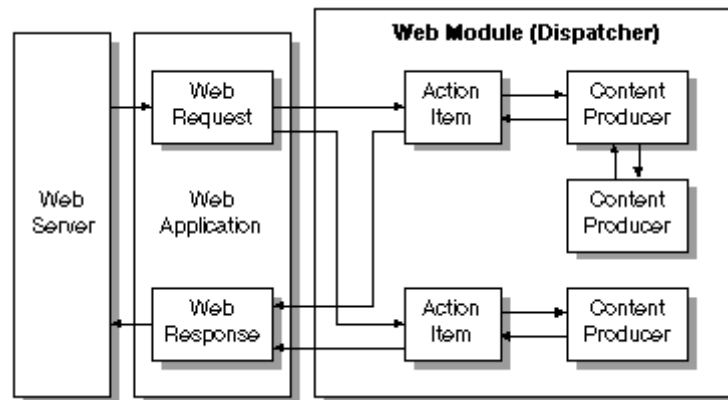
```
function Locate(const KeyFields: string; const KeyValues: Variant; Options:
TLocateOptions): Boolean;
```

여기에서 KeyFields 파라미터에 원하는 키 값과 찾는 옵션이 들어간다.

그러므로, 다음의 코드는 'CustNo'라는 필드의 값 중에서 Request.QueryFields 의 값 중에서 해당되는 레코드를 기본적인 옵션으로 찾는다는 의미이다.

```
Customer.Locate('CustNo', Request.QueryFields.Values['CustNo'], [])
```

여기에서 Request 라는 것은 TWebRequest 클래스 객체이다. TWebRequest 에 대해서 이해하기 위해서는 웹 서버 어플리케이션이 어떤 방식으로 동작하는지 알아야 한다. 이를 가장 잘 표현한 그림이 바로 다음 그림이다.



Action 아이템 이벤트 핸들러의 코드는 Request 정보에 대해 TWebRequest 객체의 프로퍼티를 통해 접근할 수 있다. 그리고, 콘텐츠 프로듀서 컴포넌트 (TPageProducer, TDataSetTableProducer 등)를 통해 Action 아이템들이 동적으로 커스텀 HTML 코드나 다른 MIME 콘텐츠를 만들어 낼 수 있다. 마지막으로, 콘텐츠 프로듀서는 다른 콘텐츠 프로듀서를 이용하거나 HTMLTagAttributes 의 자손을 이용하여 Response 메시지의 콘텐츠를 생성하는 것을 도와준다.

모든 Action 아이템들이 TWebResponse 객체에 기록을 마치고 나면, Dispatcher 가 결과를 웹 어플리케이션에 돌려준다. 어플리케이션은 이런 Response 를 서버를 통해 웹 브라우저로 보낸다.

이 TWebRequest 객체에는 여러가지 프로퍼티와 이벤트가 있으나, 먼저 중요한 TWebRequest.QueryFields 프로퍼티에 대해서 알아보도록 하자. 이 프로퍼티는 'Name=Value' 의 형태로 값을 전달한다.

예제에서 살펴 보면, 다음과 같은 코드가 있다.

```
Request.QueryFields.Values['CustNo']
```

이 코드의 의미는 'CustoNo'라는 Name 으로 들어온 파라미터의 값을 전달하라는 의미이다. 그러므로, 다음의 코드는 Request 로 들어온 파라미터 중에서 'CustNo'의 값을 받아서 Customer 테이블의 Locate 문장을 이용하여 'CustNo'에 해당되는 레코드를 찾게 된다.

```
Customer.Locate('CustNo', Request.QueryFields.Values['CustNo'], [])
```

여기에서 해당되는 레코드가 있을 경우에는 다음 코드가 수행되어 TQueryTableProducer 클래스 객체인 CustomerOrders 객체의 헤더를 적당하게 채우고, 해당 테이블의 내용을 보여주게 된다.

```
CustomerOrders.Header.Clear;  
CustomerOrders.Header.Add('The following table was produced using  
a TDataSetTableProducer.<P>');  
CustomerOrders.Header.Add('Orders for: ' + CustomerCompany.AsString);  
Response.Content := CustomerOrders.Content;
```

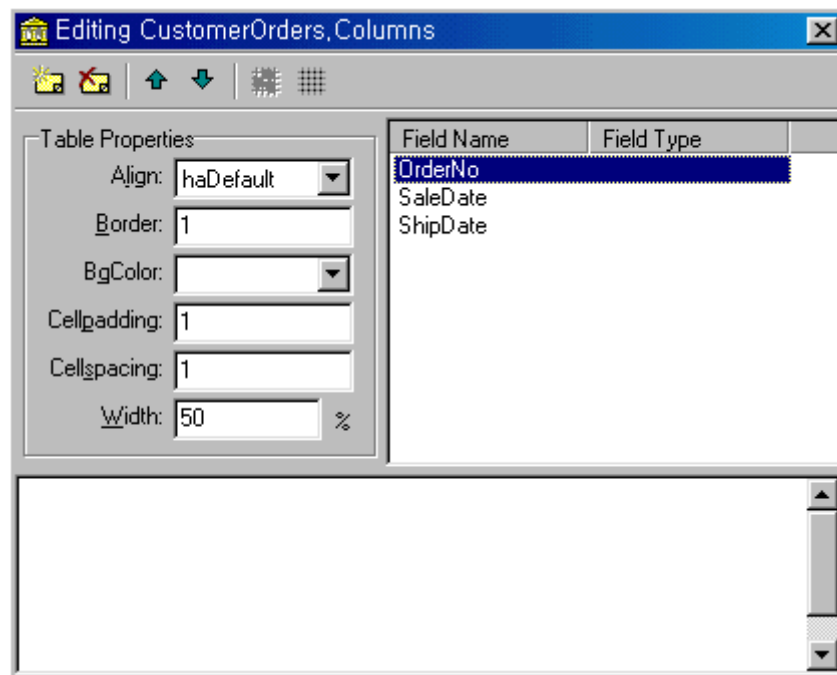
그리고, 레코드가 없는 경우에는 다음의 문장이 수행된다.

```
Response.Content := Format('<html><body><b>Customer: %s not found</b></body></html>',  
[Request.QueryFields.Values['CustNo']]);
```

이 코드의 의미는 Response 의 Content 에 Format 의 함수를 사용하여 'CustNo' 파라미터에 해당되는 레코드를 찾을 수 없다는 메시지를 내보내는 것이다.

앞에서 테이블의 내용을 보여주기 위해서 CustomerOrders 라는 TQueryTableProducer 컴포넌트를 사용하였다. 이 컴포넌트는 무척 다양한 프로퍼티와 이벤트를 가지고 있다. 먼저 Dispatcher 프로퍼티에는 해당 Dispatcher 를 설정해야 하는데, 앞의 예제에서는 웹 모듈의 이름인 CustomerInfoModule 이 설정된다. 그리고, Footer 와 Header 프로퍼티에는 테이블 형태의 HTML 태그를 생성할 때에 처음과 마지막 부분을 장식할 데이터를 지정하게 된다. 또한, RowAttributes 와 TableAttributes 프로퍼티에서는 어떤 쿼리를 통해 데이터를 가져오고, 출력할 내용의 속성을 정의할 수 있다.

그리고, Columns 프로퍼티를 이용하면 출력될 그리드의 형태를 다음 그림과 같이 설정할 수 있다.



이 내용을 살펴보면 OrderNo, SaleDate, ShipDate 의 3 가지 필드영역을 가지는 그리드를 출력함을 알 수 있다. 그리고 Align 을 비롯한 출력에 필요한 다양한 속성을 원하는 형태로 설정할 수 있다.

문장에서 사용한 Header 와 Footer 의 내용은 기본적으로 비어있는데, 이 비어있는 내용을 채우기 위해서 다음과 같은 내용을 사용하였다.

```
CustomerOrders.Header.Clear;
```

```
CustomerOrders.Header.Add('Orders for: ' + CustomerCompany.AsString);
```

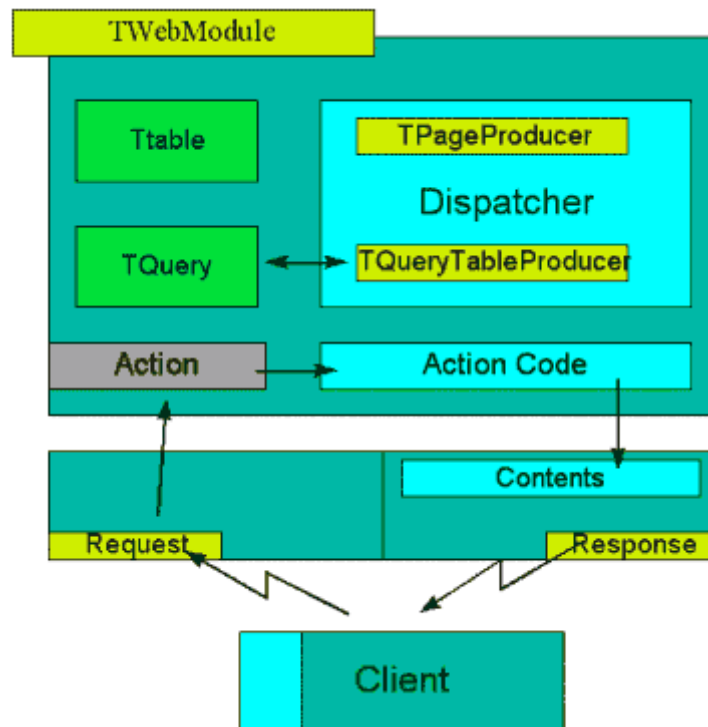
마지막으로, 다음과 같은 코드를 이용하여 Response 객체의 Content 를 CustomerOrders 객체의 HTML 출력 내용으로 채워서 출력을 하도록 한다.

```
Response.Content := CustomerOrders.Content;
```

Response 객체는 Request 객체에 대응되는 것으로 TWebResponse 클래스의 객체이다. 그러면, TWebResponse 클래스에 대해 간단히 알아보자.

TWebResponse 객체는 Response 를 전송하는 두가지 메소드를 제공한다. 이들은 각각 SendResponse 와 SendRedirect 이다. TWebResponse 객체의 모든 헤더 프로퍼티와 지정된 콘텐츠를 사용하는 Response 를 전송할 때는 SendResponse 를 호출한다. 이에 비해 SendRedirect 메소드는 웹 클라이언트에서 다른 URI 로 redirect 하기만 하면 될 때 효과적으로 쓰인다.

만약 Response 를 전송하는 이벤트 핸들러가 없다면, 웹 어플리케이션 객체는 Dispatcher 가 종료된 후 Response 를 전송하게 된다. 어쨌든 Response 를 다루는 Action 아이템이 없으면, 어플리케이션은 Response 를 전송하지 않고 웹 클라이언트와의 접속을 끊는다. 결국, Response 객체는 실제 웹 서버로 전송할 Content 에 해당되는 문장을 담게 된다. 이상의 구조를 단순하게 정리하여 그림으로 표현하면 다음과 같다.



CGI 어플리케이션 개발자는 TWebModule 위에서 원하는 데이터 세트로 사용할 테이블이나 쿼리를 얻는다. 그리고, TPageProducer 나 TQueryTableProducer 컴포넌트를 이용하여 원하는 작업을 Response 의 Contents 프로퍼티에 넣을 수 있고, 이러한 작업들이 바로 TWebModule 의 Action 에 의하여 구동되는 것이다.

이것이 텔파일을 이용한 CGI 어플리케이션을 작성하는 방법과 그 구조이다. CGI 프로그래밍은 이런 이벤트의 동작에만 주의하고 Contents 를 통하여 이루어진다는 것만 주의하면 그렇게 어려운 내용은 아니다.

## 업로드 CGI 의 제작

그러면, 자료를 업로드할 수 있는 CGI 를 하나 만들어 보자. 먼저 업로드를 호출하기 위한 HTML 을 작성하도록 한다.

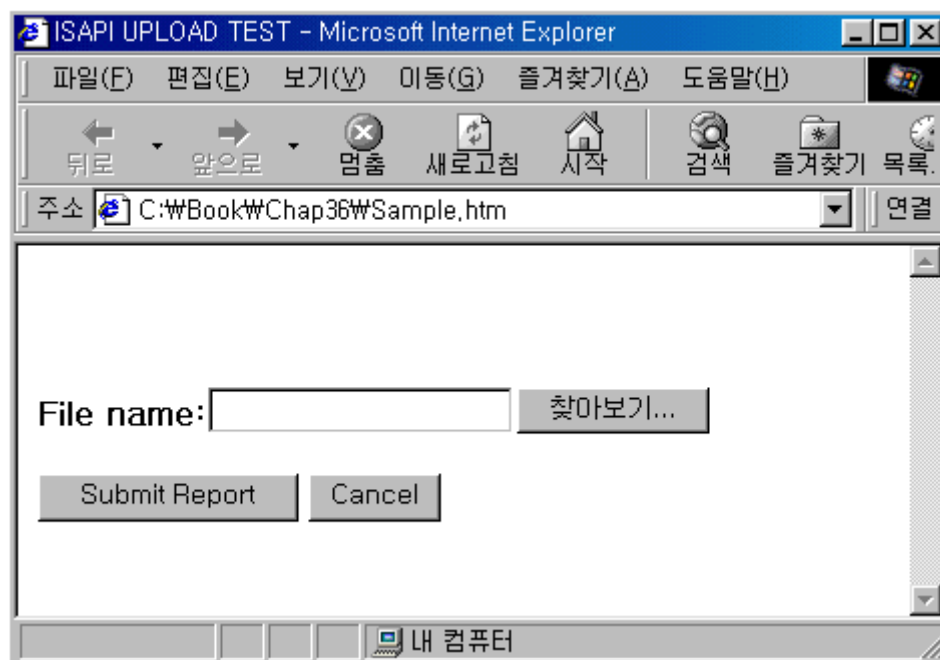
우선 파일 업로드 양식을 만들기 위해서는 fileupload 객체를 이용해서 작성을 해야 하며, CGI 로 보내는 데이터 형도 설정해야 된다. 다음은 웹 페이지의 샘플 소스이다.

```

<html>
<head>
<title>ISAPI UPLOAD TEST</title>
</head>
<body><p>&nbsp;</p>
<p> </p>
<FORM ENCTYPE="multipart/form-data" ACTION="Exam1.dll/upload" METHOD="POST">
<BR><B>File name:</B><INPUT TYPE="file" NAME="File">
<P><INPUT TYPE="submit" VALUE="Submit Report">
<INPUT TYPE="button" VALUE="Cancel" onClick="window.close()">
</FORM>
</body>
</html>

```

이 HTML 파일을 브라우저로 읽어 보면 다음과 같이 나타날 것이다.



이때에 CGI 를 호출하는 부분이 다음과 같다.

```

<FORM ENCTYPE="multipart/form-data" ACTION="Exam1.dll/upload" METHOD="POST">

```

이 내용 중에서 ‘ENCTYPE=”multipart/form-data”’ 부분은 해당 페이지에 있는 데이터 들을 cgi 로 송신할 종류를 나타내며 ”multipart/form-data”는 데이터를 송신하는 내용들 중에서 한가지이다.

METHOD=”POST”는 데이터를 어떠한 프로토콜로 보낼 것인지를 결정하며, “POST”는 표준 입력 방식으로 보낸다는 것을 의미한다.

참고로, 이렇게 파일을 업로드할 때에는 반드시 “POST”로 해야 한다.

그리고, 다음 내용은 fileupload 객체에 해당하는 부분으로, 실제 업로드할 파일을 선택하는 객체이다.

```
<INPUT TYPE="submit" VALUE="Submit Report">
```

cgi 에서 업로드를 실제 수행하는 작업은 의외로 간단하다. 웹 페이지에서 표준 입력으로 보내준 데이터 중에서 파일에 관련된 부분만 골라서 저장하면 된다. 업로드 웹 페이지에서 송신한 데이터는 인터넷 표준 문서인 RFC 문서(rfc1867.txt)에 정의된 데이터 포맷으로 cgi 에서 수신할 수 있다.

수신된 데이터 중에서 파일 정보에 해당하는 부분은 다음과 같은 구조로 되어 있다.

```
Content-disposition: attachment; filename="file1.txt"
```

```
Content-Type: text/plain
```

```
Contents
```

```
--BbC04y
```

수신된 파일의 파일 이름은 filename 이라는 부분에 정의되어 있고, 파일의 내용은 “contents”부분에 정의되어 있으므로, “contents” 부분을 따로 잘라내고 잘라낸 내용을 수신된 파일명으로 저장하면 파일 업로드를 구현할 수 있다.

다음 내용은 표준 입력으로 들어온 데이터를 출력한 파일이다.

```
-----7ce28a3a32c
```

```
Content-Disposition: form-data; name="uploadfilename"; filename="C:\Wtmp\WWIN95WPcmcard.inf"
```

```
Content-Type: text/plain
```

```
; Ethernet PCMCIA Adapter INF file for Windows 95
```

```
;
```

[version]

signature="\$CHICAGO\$"

Class=Net

provider=%V\_MS%

[Manufacturer]

%V\_KINGMAX%=KINGMAX

[KINGMAX]

%KINGMAX.DeviceDesc%=EN10T2.ndi,PCMCIAWKINGMAX-EN10T2T-1BAB

[EN10T2.ndi]

AddReg=EN10T2.ndi.reg,PCM95.ndi.reg,PCM95.params.reg

[EN10T2.ndi.reg]

HKR,Ndi,DeviceID,,"PCMCIAWKINGMAX-EN10T2T"

[PCM95.ndi.reg]

; key,subkey,valuname,type,value

HKR,,DevLoader,,\*ndis

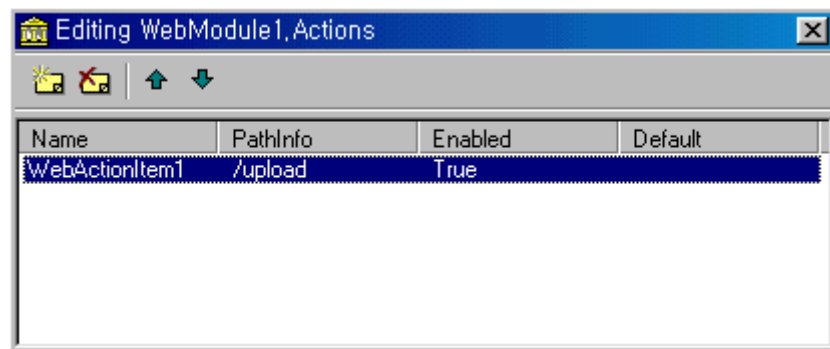
HKR,,DeviceVxDs,,pcm95.sys

HKR,,EnumPropPages,,,"netdi.dll,EnumPropPages"

-----7ce28a3a32c-----

먼저 File|New 메뉴를 선택하고 여기에서 Web Server Application 아이템을 더블 클릭하고, 나타나는 CGI 형태로는 ISAPI/NSAPI 를 선택하도록 하자. 이렇게 하면, 웹 모듈이 생성되는데 오브젝트 인스펙터에서 Actions 프로퍼티를 선택하고, ‘...’ 버튼을 클릭하면 Actions 프로퍼티 에디터가 나타날 것이다. 여기에서 ‘Add New (Ins)’ 버튼을 클릭하고 이 아이템을 선택한 뒤, PathInfo 프로퍼티를 ‘/upload’로 설정하면 다음과 같이 나타날 것이다.





이제 업로드 CGI 를 구현하기 위해서는 다음과 같이 Action 아이템의 이벤트 핸들러를 구현해야 한다. 참고로, 이렇게 원시적인 코딩을 위해서는 isapi2.pas 와 isapiapp.pas 유닛을 uses 절에 추가해 주어야 한다.

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  i: Integer;
  L: LongInt;
  Buffer, Prueba: ^Byte;
  MemoryStream: TMemoryStream;
  Nowecb: PEXTENSION_CONTROL_BLOCK;
  TotalByte: Integer;          //수신될 데이터 크기
begin
  MemoryStream := TMemoryStream.Create;
  GetMem(Prueba, 11856);
  try
    Nowecb := TISAPIRequest(Request).ECB;
    TotalByte := Nowecb.cbTotalBytes;
    Buffer := Nowecb.lpbdata;
    L := Nowecb.cbAvailable;
    MemoryStream.Write(Buffer^, L);
    Dec(TotalByte, L);
    i := L;
    while((TotalByte > 0) and (i <> -1)) do
    begin
      i := Request.ReadClient(Prueba^, 11586);
      MemoryStream.Write(Prueba^, i);
    end;
  end;
end;

```

```

        Dec(TotalByte, i);
    end;

    MemoryStream.SaveToFile('c:\Windows\Temp\Wtmps.txt');
finally
    FreeMem(Prueba);
    MemoryStream.Free;
end;
end;

```

먼저 사용되는 변수 중에 PEXTENSION\_CONTROL\_BLOCK 데이터 형의 Nowecb 라는 변수가 있다. 이 데이터 형은 TISAPIRequest 클래스의 ECB 프로퍼티의 포인터 형으로, 서버 어플리케이션 DLL 이 웹 서버와 통신을 하기 위해 사용하는 확장 컨트롤 블록(extension control block)이다. 그러므로, ECB 프로퍼티는 서버 어플리케이션 DLL 이 클라이언트 요구 메시지를 대표할 때 사용된다. 이와 같이 ECB 프로퍼티를 직접 읽어올 수도 있지만, GetFieldByName 메소드를 이용하여 헤더의 필드 값을 읽어올 수 있다.

GetFieldByName 메소드는 HTTP 헤더 변수의 값을 읽어올 때 사용하는데, 이를 읽어와서 특정 작업을 하기 위해서는 인터넷 표준 문서인 각종 RFC 문서를 참고하면 된다. 텔넷이나 FTP 등의 대표적인 프로토콜 들도 RFC 로 규정되어 있으며, 이를 기초로 하여 원시적인 내용을 모두 구현할 수 있는 것이다. 그리고, 데이터를 읽어와서 저장하기 쉽도록 델파이에서 제공하는 TMemoryStream 클래스를 사용한다.

TotalByte, Buffer, L 이라는 변수는 각각 확장 컨트롤 블록(ECB)의 멤버를 저장하기 위해서 사용된다. TotalByte 변수는 클라이언트에 의해 지정되는 총 바이트 수를 저장하며, Buffer 변수는 해당 데이터에 대한 포인터 변수, 그리고 L 변수는 가능한 데이터의 바이트 수를 저장한다.

그러므로, 일단은 TISAPIRequest 클래스의 ECB 프로퍼티를 이용하여 Nowecb 변수에 Request 객체의 값을 읽어온 후 TotalByte, Buffer, L 변수에 해당 멤버의 값을 저장한다. 그리고, 메모리 스트림 객체에 Buffer 변수에 들어 있는 데이터를 L 변수에 크기 만큼 읽어와서 저장하는 것이 다음의 코드 들이다.

```

Nowecb := TISAPIRequest(Request).ECB;
TotalByte := Nowecb.cbTotalBytes;
Buffer := Nowecb.lpbdata;
L := Nowecb.cbAvailable;
MemoryStream.Write(Buffer^, L);

```

그리고, TotalByte 변수에서 읽은 수 만큼을 계속 감소시키고 이 값이 0 이 되거나 더 이상

읽어올 데이터가 없을 때까지 읽는다.

```
Dec(TotalByte, L);  
i := L;  
while((TotalByte > 0) and (i <> -1)) do  
begin  
    i := Request.ReadClient(Prueba^, 11586);  
    MemoryStream.Write(Prueba^, i);  
    Dec(TotalByte, i);  
end;
```

여기에서 Request 객체의 ReadClient 메소드는 HTTP request 의 콘텐츠 내용을 첫번째 파라미터로 지정된 변수에 두번째 파라미터의 크기 만큼 읽어오는 메소드이다. 즉, Prueba 변수에 11586 크기의 데이터를 읽게 된다. ReadClient 메소드는 서버 어플리케이션이 한 번에 읽어오기에 너무 큰 request 데이터를 읽을 때, 이를 쪼개서 읽어올 수 있도록 해준다. 더 이상 읽어올 데이터가 없을 경우에는 -1 을 반환한다.

이렇게 루프를 돌게 되면, 결국에는 MemoryStream 변수에는 HTTP request 에 의해 넘어온 데이터가 그대로 저장된다. 이를 SaveToFile 메소드를 이용하여 저장하도록 할 수 있다.

## 정 리 (Summary)

이번 장에서는 텔파이를 이용하여 CGI 어플리케이션을 제작하는 방법에 대해서 알아보았다. 현재의 인터넷 환경은 앞서 이야기 했듯이 CGI 를 비롯한 여러가지 방법으로 웹 페이지의 내용과 형태를 풍부하게 만들어볼 수 있다. 하지만, CGI 는 아직도 가장 많이 쓰이는 방법 중의 하나이기 때문에 이를 텔파이를 이용하여 쉽게 작성할 수 있다는 점은 커다란 매력이 아닐 수 없다.

다음 장에서는 텔파이에서 제공하는 소켓 컴포넌트를 이용하여 프로그래밍하는 방법에 대해서 알아볼 것이다.

## 소켓 프로그래밍 기법의 활용

### (Using Socket Programming Techniques)

윈도우에서의 프로세스간 통신 기법으로는 명명된 파이프, DCOM, DDE, 클립 보드와 각종 네트워크 프로그래밍 기법 등을 이용할 수 있다. 이 중에서도 윈도우 95 와 윈도우 NT 3.5 버전부터는 내부적인 통신 프로토콜로 기존의 NetBIEU 와 함께 TCP/IP 를 사실상의 표준으로 인정하고 이를 지원하고 있다. 또한, DCOM 과 윈도우 소켓을 프로세스간 통신의 표준으로 삼고 있으며, 윈도우 NT 4.0 부터는 윈도우 소켓의 2.0 버전(WinSock 2.0)을 사용하여 보다 강화된 소켓 프로그래밍을 지원하게 되었다.

이러한 소켓 프로그래밍을 위해서는 Win32 에서 지원하는 API 를 직접 이용하여 프로그래밍을 할 수도 있겠으나, 델파이에서 지원하는 소켓 컴포넌트를 이용하면 쉽게 소켓을 지원하는 어플리케이션을 지원할 수 있다.

이번 장에서는 소켓 컴포넌트를 이용하는 방법과 소켓 프로그래밍 기법을 익혀보도록 한다.

#### 소켓 프로그래밍의 기초

소켓이란 네트워크 프로토콜을 구현할 때 여기에 저장된 데이터를 포함한 커다란 집합에 대한 핸들이다. 쉽게 말하면 네트워크에 대한 파일 핸들이라고 생각하면 된다. 네트워크 프로그래밍의 기본은 TCP/IP 연결을 하고, 소켓을 생성해서 이를 전송하거나 받는 것이다.

소켓에는 원래 서버용, 클라이언트용 소켓이 분리되어 있는 것은 아니다. 그렇지만 델파이에서는 TServerSocket, TClientSocket 으로 분리된 소켓을 제공하고 있다. 이들은 모두 TCustomSocket 클래스에서 상속받은 것으로 내부적으로는 동일한 소켓을 사용하고, 사용 방법도 거의 비슷하다.

서버가 되는 소켓은 클라이언트측 소켓과는 달리 클라이언트의 연결 요구를 기다리는 listen 이라는 작업을 해야하고, 클라이언트는 서버에 연결(connect)해야 한다. 여기서 양측 소켓의 Active 프로퍼티를 True 로 설정하면 서버와 클라이언트가 접속된다. 또한, TServerSocket 클래스에는 여러 개의 클라이언트가 접속되었을 때 이를 관리할 수 있는 기능을 추가로 가지고 있다.

#### IP 주소와 포트

델파이의 소켓 컴포넌트에는 Address 와 Port 라는 프로퍼티가 있다. Address 프로퍼티는 IP 주소를 나타내며, Port 프로퍼티는 서버로 들어오는 메시지를 통과시키는 번호이다. 포트 번호가 없으면 하나의 서버 컴퓨터에는 하나의 서버 프로그램만 설치할 수 있다. 포트 번호

를 이용해서 동일한 서버 컴퓨터에 여러 개의 서버 프로그램을 사용할 수 있다. 예를 들어 메일 서버, 뉴스그룹 서버, 웹서버, FTP 서버, 텔넷 서버 등의 프로그램들은 모두 다른 포트를 사용한다. 이런 포트 번호 중 일반적으로 사용되는 서비스에 관한 것들이 있는데 SMTP 는 25, NNTP 는 119, Telnet 은 23, FTP 는 21 을 보통 사용한다.

## 소켓과 소켓 연결 (Socket Connection)

소켓은 네트워크 어플리케이션이 네트워크 상의 다른 시스템 사이를 통신할 수 있도록 도와주는 도구가 된다. 각각의 소켓은 하나의 네트워크 연결로 생각할 수 있는데, 여기에는 어플리케이션이 실행되는 시스템, 인터페이스 종류, 연결에 사용된 포트에 대한 주소가 있어야 한다. 그러므로, 소켓 연결에 대해서 충분히 알기 위해서는 각 연결 부분에 대한 소켓의 주소를 반드시 알아야 한다.

이렇게 소켓 연결을 하기 전에 연결 부분을 담당하게 되는 소켓에 대한 정보를 제공해야 하는데, 일부의 정보는 어플리케이션이 실행되고 있는 시스템에서 알아낼 수 있다. 예를 들어, 각 클라이언트 소켓의 로컬 IP 주소에 대한 정보는 운영체제에서 알아낼 수 있으므로 따로 제공할 필요가 없다.

따로 제공해야 하는 정보는 현재 작업하고 있는 소켓의 종류에 따라 달라지는데, 클라이언트 소켓은 연결하고자 하는 서버에 대한 정보를 제공해야 하며 서버 소켓은 제공하는 서비스를 제공하는 포트에 대한 정보를 제공해야 한다. 이러한 소켓 연결에 대한 정보에는 IP 주소와 포트 번호가 모두 포함된다.

## 호스트(Host)란 ?

호스트는 소켓을 포함한 어플리케이션이 동작하는 시스템을 말한다. 이렇게 소켓에 대한 호스트를 지정할 때에는 다음과 같이 표준 인터넷 주소로 사용되는 IP 주소 표기 방식을 많이 사용한다.

123.123.1.2

하나의 시스템은 하나 이상의 IP 주소를 지원하게 된다. IP 주소는 기억하기도 어렵거니와 알아보기도 쉽지 않기 때문에, 호스트의 이름을 지정하는 방식을 같이 사용하여 이러한 단점을 극복한다. 이렇게 이름으로 된 방식의 IP 주소에 대한 앨리어스를 URLs(Uniform Resource Locators) 라고 하며, 다음과 같은 도메인 이름과 서비스를 포함한 형태가 된다.

<http://www.ExamSite.Com>

서버 소켓은 시스템에서 로컬 IP 주소를 알아낼 수 있기 때문에 호스트를 지정할 필요가 없다. 만약 로컬 시스템이 하나 이상의 IP 주소를 가지고 있을 경우에는 서버 소켓은 모든 IP 주소에 대한 클라이언트의 요구를 이용한다. 서버 소켓이 연결되면 클라이언트 소켓은 리모트 IP 주소를 제공하게 된다. 클라이언트 소켓은 반드시 호스트 이름이나 IP 주소를 입력해서 리모트 호스트를 지정해 주어야 한다.

## 연결의 종류

소켓 연결에는 연결의 초기화와 어떤 로컬 소켓이 연결되는지에 따라 기본적으로 다음과 같은 세가지로 나누어 볼 수 있다.

### 1. 클라이언트 연결 (Client connections)

클라이언트 연결은 로컬 시스템의 클라이언트 소켓을 리모트 시스템의 서버 소켓에 연결하는 것을 말한다. 클라이언트 연결은 클라이언트 소켓에 의해 개시되고 초기화된다. 먼저 클라이언트 소켓이 연결하고자 하는 서버 소켓에 대한 정보를 제공하면, 클라이언트 소켓이 서버 소켓을 찾게 되고, 서버의 위치를 파악하게 되면 연결을 요구한다. 서버 소켓은 클라이언트 요구에 대한 큐(queue)를 가지고 있어서 시간이 될 때마다 연결을 시도한다. 일단 서버 소켓이 클라이언트 연결을 받아들이면 클라이언트 소켓에 연결된 서버 소켓에 대한 모든 정보를 전송하게 되며, 클라이언트에 의해 연결이 완료된다

### 2. 리스닝 연결 (Listening connections)

서버 소켓이 활동적으로 클라이언트를 찾아서 연결을 시도하지 않고, 수동적으로 클라이언트의 요구를 기다리는 하프 연결 (half connection) 상태를 유지하는 형태의 연결이다. 서버 소켓은 큐를 리스닝 연결과 연관지어서 관리하며, 큐에는 클라이언트의 연결 요구가 계속 기록된다. 일단 서버 소켓이 클라이언트의 연결 요구를 받아들이면 클라이언트와 연결하기 위한 새로운 소켓을 생성하게 된다.

이렇게 하면 리스닝 연결 자체는 다른 클라이언트 요구를 받아들일 수 있도록 계속 열려있게 할 수 있다.

### 3. 서버 연결 (Server connections)

서버 연결은 서버 소켓에 의해서 이루어지는 것으로, 리스닝 소켓이 클라이언트 요구를 받아들이면 생성된다. 일단 서버가 연결을 받아들이면 서버 소켓의 정보가 클라이언트에게 전송되어 클라이언트 소켓이 이 정보를 받으면 연결이 완료되는 형태이다.

일단 연결이 되면 서버 연결과 클라이언트 연결은 차이가 없는 연결 방식이다. 기본적으로 클라이언트 연결과 서버 연결은 두개의 종료점(endpoint)를 가지며 같은 능력과 같은 종류의 이벤트를 사용한다. 그에 비해 리스닝 연결은 단지 하나의 종료점 만을 가지고 있는 본질적으로 다른 연결방식이다.

## 서비스 프로토콜

네트워크 서버와 클라이언트를 개발하기에 앞서, 먼저 어플리케이션이 제공하거나 사용하게 될 서비스에 대한 이해가 반드시 선행되어야 한다. 많은 서비스 들은 네트워크 어플리케이션이 반드시 지원해야 하는 표준 프로토콜을 가지고 있다. 만약 HTTP, FTP 등의 표준 서비스를 지원하는 네트워크 어플리케이션을 제작한다면, 다른 시스템과 통신하게 되는 프로토콜에 대한 이해가 필요하다.

만약 다른 시스템과 통신하는데 있어서 새로운 형태의 서비스를 제공한다면, 제일 먼저 서비스를 사용하게 될 서버와 클라이언트 사이의 통신 프로토콜을 디자인해야 한다. 이때에는 어떤 메시지가 전달될 것이며, 이런 메시지 들이 어떻게 조화를 이루어야 하며, 정보의 암호화는 어떤 형식으로 할 것인지 등을 결정해야 한다.

가끔 네트워크 서버와 클라이언트 어플리케이션에서 네트워킹 소프트웨어와 서비스를 사용하는 어플리케이션 사이에 레이어(layer)를 제공하는 경우가 있다. 예를 들어, HTTP 서버는 인터넷과 웹서버 어플리케이션 사이에 위치하여 콘텐츠를 제공하고, HTTP 리퀘스트 메시지에 반응하게 된다.

소켓은 네트워크 서버와 클라이언트 어플리케이션, 네트워킹 소프트웨어 사이에 인터페이스를 제공한다. 이때 ISAPI 등의 흔히 사용되는 표준 서버에 대한 API 를 복사해서 사용할 수도 있고, 자신만의 API 를 디자인해서 사용할 수도 있다.

## 서비스와 포트

대부분의 서비스는 특정 포트 번호와 연관되어 있다. 이럴 때에는 포트 번호를 서비스에 대한 번호 코드(numeric code)로 생각할 수 있다. 만약 표준 서비스를 구현한다면 텔넷의 윈도우 소켓 객체의 메소드를 이용하면 그 서비스에 대한 포트 번호를 알아낼 수 있다. 그에 비해 새로운 서비스를 제공하는 경우라면 윈도우 95 나 NT 의 Services 파일에 포트 번호를 연관시켜 지정할 수 있다.

## 소켓 연결과 데이터 송수신

다른 시스템과 소켓 연결을 하는 이유는 연결을 통해서 정보를 송수신할 수 있기 때문이다. 이때 어떤 정보를 주고 받을 지와 언제 어떤 방식을 사용할 지 등은 소켓 연결에 사용된 서

비스에 좌우된다.

이렇게 데이터를 읽고 쓰는 데에는 두가지 방식이 있다. 소켓에 데이터를 읽고 쓸 때 비동기적인 방식을 사용해서 네트워크 어플리케이션에서 다른 코드의 실행을 방해하지 않는 것을 논-블로킹 연결(Non-Blocking connections)이라고 하며, 데이터를 읽고 쓰는 작업을 쓰레드를 이용하여 독립적으로 실행하는 형태의 연결을 블로킹 연결(Blocking connections)이라고 한다.

## 블로킹 연결 (Blocking connections)

클라이언트 소켓에서는 ClientType 프로퍼티를 ctBlocking 으로 설정하면 블로킹 연결이 생성된다. 클라이언트 어플리케이션에 따라서는 읽고 쓰는 데에 새로운 쓰레드를 생성하기를 원할 수도 있는데, 이렇게 하면 어플리케이션은 연결이 완료되어 데이터를 읽고, 쓸 때까지 다른 쓰레드를 실행할 수 있다.

서버 소켓에서는 ServerType 프로퍼티를 stThreadBlocking 으로 설정하면 블로킹 연결이 생성된다. 블로킹 연결은 연결에 의한 데이터 교환이 될 때까지 실행이 되지 않으므로, 다른 클라이언트 연결에 대해서 항상 새로운 쓰레드가 생성된다.

### ● 블로킹 연결과 쓰레드의 이용

클라이언트 소켓은 블로킹 연결이 사용될 때 새로운 쓰레드가 자동으로 생성되지 않는다. 만약, 클라이언트 어플리케이션이 데이터를 읽고, 쓰는 것 이외에 다른 작업이 없다면 이러한 방식도 큰 상관이 없겠지만, 연결이 이루어지지 않은 경우에도 사용자 인터페이스에서 다른 작업을 할 수 있게 하려면 새로운 쓰레드를 생성해야 한다.

그에 비해 서버 소켓은 블로킹 연결이 생성될 때마다 각 클라이언트 연결에 대해 새로운 쓰레드가 생성된다. 그렇게 때문에 연결을 통해 클라이언트가 데이터를 읽고 쓰는 작업을 할 때 다른 클라이언트가 이를 기다리지 않아도 된다. 서버 소켓은 TServerClientThread 객체를 사용해서 각 연결에 대한 쓰레드를 구현한다. 일단 서버 클라이언트 쓰레드가 실행되면, 연결되어 있는 클라이언트 측에서 연결을 통해 데이터를 쓰고 있는지 검사하여 그렇다면 OnClientRead 이벤트를 발생시키며, 그렇지 않으면 OnClientWrite 이벤트를 발생시키고 서버가 데이터를 쓰기 시작한다.

### ● TWinSocketStream 클래스의 활용

논-블로킹 연결과 블로킹 서버 연결 모두 연결에 의해 데이터를 읽고, 쓸 수 있게 되면 특정 이벤트를 받게 된다. 이때 이벤트 핸들러에서는 실제로 데이터를 읽고, 쓰는 작업을 윈도우 소켓 객체의 메소드를 이용해서 실행한다.



블로킹 클라이언트 연결에서는 반드시 자기 자신이 반대편의 연결이 데이터를 읽고, 쓸 준비가 되었는지를 파악해야 한다. 이때 TWInSocketStream 객체를 이용해서 연결을 통한 데이터 읽기, 쓰기를 실행하면, 적절한 시간 간격 등을 조율할 수 있는 메소드를 제공받을 수 있다. 예를 들어, WaitForData 메소드를 호출하면 서버 소켓이 준비될 때까지 기다리게 할 수 있다.

- 클라이언트 쓰레드의 작성

클라이언트 접속을 위한 쓰레드를 작성할 때에는 New Thread Object 대화 상자를 이용하여 새로운 쓰레드 객체를 정의할 수 있다. 새로운 쓰레드 객체의 Execute 메소드는 실제 쓰레드 연결을 통한 데이터 읽기와 쓰레드를 관리한다. 다음의 코드가 전형적인 클라이언트 쓰레드의 예이다.

```
procedure TMyClientThread.Execute;
var
  TheStream: TWInSocketStream;
  buffer: string;
begin
  TheStream := TWInSocketStream.Create(ClientSocket1.Socket, 60000);
  //TWInSocketStream 을 생성한다.
  try
    while (not Terminated) and (ClientSocket1.Active) do
    begin
      try
        GetNextRequest(buffer);
        TheStream.Write(buffer, Length(buffer) + 1);    //버퍼의 내용을 서버에 기록한다.
        ...
      except
        if not(ExceptObject is EAbort) then
          Synchronize(HandleThreadException);
        end;
      end;
    end;
  finally
    TheStream.Free;
  end;
end;
```

쓰레드를 이용하려면 OnConnect 이벤트 핸들러를 작성하면 된다.

- 서버 쓰레드의 작성

서버 접속에 대한 쓰레드는 TServerClientThread 에서 상속받는다.

서버 쓰레드를 구현하기 위해서는 Execute 메소드가 아닌, ClientExecute 메소드를 오버라이드해야 한다. 클라이언트 쓰레드와 비슷하게 구현하지만, 클라이언트 소켓 컴포넌트 대신에 TServerClientWinSocket 객체를 사용한다는 점이 가장 큰 차이점이다. 이 객체는 ClientSocket 프로퍼티를 통해 접근이 가능하다. 그리고, 예외 처리는 HandleException 메소드를 호출하면 된다.

```
procedure TMyServerThread.ClientExecute;
var
  Stream: TWinSocketStream;
  Buffer: array[0 .. 9] of Char;
begin
  while (not Terminated) and ClientSocket.Connected do
    begin
      try
        Stream := TWinSocketStream.Create(ClientSocket, 60000);
        try
          FillChar(Buffer, 10, 0);           //버퍼를 초기화 한다.
          if Stream.WaitForData(60000) then
            begin
              if Stream.Read(Buffer, 10) = 0 then ClientSocket.Close;
              //1 분까지 기다린다.

              ...
            end
          else
            ClientSocket.Close;
          finally
            Stream.Free;
          end;
        except
          HandleException;
```

```
end;  
end;  
end;
```

이 스레드를 사용하려면 OnGetThread 이벤트 핸들러에서 스레드를 생성하면 된다.

## 논-블로킹 연결 (Non-Blocking connections)

클라이언트 소켓에서 ClientType 프로퍼티를 ctNonBlocking 으로 설정하면 논-블로킹 연결이 이루어 진다. 이 경우 반대 편의 서버가 데이터를 읽거나 쓰게 되면 클라이언트 소켓이 이를 알 수 있으며, 이때 OnRead 또는 OnWrite 이벤트 핸들러에 의해서 반응하게 된다. 서버 소켓에서는 ServerType 프로퍼티를 stNonBlocking 으로 설정하면 논-블로킹 연결이 생성되는데, 논-블로킹 클라이언트 연결처럼 클라이언트에서 데이터를 읽고 쓰게 되면 OnClientRead 또는 OnClientWrite 이벤트 핸들러에 의해서 반응하게 된다.

이러한 이벤트에서 소켓 연결과 관련한 윈도우 소켓 객체가 파라미터로 전달되며, 이들 객체를 이용하면 여러가지 메소드를 활용할 수 있다. 소켓 연결에서 데이터를 읽을 때에는 ReceiveBuf, ReceiveText 메소드를 사용할 수 있다. ReceiveBuf 메소드는 사용하기 전에 ReceiveLength 메소드를 사용해서 반대 편 연결에서 전송하려는 데이터의 바이트 수를 결정 한 후에 사용한다.

SendBuf, SendStream, SendText 메소드 등을 이용하면 데이터를 쓸 수 있다. 또한, 데이터를 쓰고 나서 더 이상 소켓 연결을 유지할 필요가 없을 때에는 SendStreamThenDrop 메소드를 사용해서 스트림에서 읽은 데이터를 모두 전송하고 연결을 단도록 할 수 있다.

참고로 SendStream, SendStreamThenDrop 메소드를 사용하면 소켓이 연결이 종료된 후 스트림을 자동으로 메모리에서 해제하므로, 스트림 객체를 직접 해제할 필요가 없다.

## 클라이언트 소켓의 이용

어플리케이션을 TCP/IP 클라이언트로 사용할 때에는 클라이언트 소켓 컴포넌트(TClientSocket)를 폼이나 데이터 모듈에 추가한다. 클라이언트 소켓에는 연결하고자 하는 서버 소켓과 서버에서 제공받을 서비스를 지정하게 된다.

각각의 클라이언트 소켓 컴포넌트는 연결에서의 클라이언트측 종료점을 나타내게 되는 클라이언트 윈도우 소켓 객체(TClientWinSocket)을 사용한다.

### ● 서버의 지정

클라이언트 소켓 컴포넌트는 서버 시스템과 연결하고자 하는 포트를 지정할 때 사용할 수

있는 프로퍼티가 있다. 서버 시스템은 Host 프로퍼티에서 호스트의 이름을 이용해서 지정하거나, Address 프로퍼티에 직접 IP 주소를 적어넣을 수 있다. 만약 둘 다 지정한 경우에는 호스트 이름을 사용한다.

포트 역시 Port 와 Service 프로퍼티를 이용해서 지정할 수 있는데, Port 프로퍼티에는 포트의 번호를 직접 지정하는 것이고 Service 프로퍼티에는 포트 번호와 연관된 표준 서비스의 이름을 지정할 경우 간접적으로 포트가 지정되는 것이다. 둘 다 지정된 경우에는 서비스 이름을 사용한다.

## ● 연결의 생성

연결하고자 하는 서버에 대한 정보를 클라이언트 소켓 컴포넌트에 설정하고 나면, 런타임에서 Open 메소드를 호출하여 연결을 시도하게 된다. 디자인 시에 Active 프로퍼티를 True로 설정하면 어플리케이션이 시작할 때 연결을 시도한다.

## ● 연결에 대한 정보 얻기

서버 소켓과의 연결이 완료되면 클라이언트 윈도우 소켓 객체를 사용해서 연결에 대한 여러 가지 정보를 얻어올 수 있게 된다. 이때 이 객체를 얻어오기 위해 Socket 프로퍼티를 사용한다. 윈도우 소켓 객체에는 클라이언트와 서버 소켓이 사용하는 포트 번호와 주소를 결정할 때 사용하는 프로퍼티가 있으며, SocketHandle 프로퍼티를 이용해서 윈도우 소켓 API를 호출할 때 사용할 소켓 연결에 대한 핸들을 얻을 수도 있다. 또한, Handle 프로퍼티를 이용해서 소켓 연결에서의 메시지를 받는 윈도우에 접근할 수 있으며 ASyncStyles 프로퍼티를 이용해서 윈도우 핸들이 받게 되는 메시지의 종류를 결정할 수도 있다.

## ● 연결 종료

서버 어플리케이션과의 소켓 연결을 통한 통신이 끝나면, Close 메소드를 이용해서 연결을 종료할 수 있다. 이때 서버 측에서 연결을 종료하면 OnDisconnect 이벤트가 발생하므로, 적절한 처리를 해줄 수 있다.

## 서버 소켓의 이용

어플리케이션을 TCP/IP 서버로 둔갑시키려면 먼저 서버 소켓 컴포넌트인 TServerSocket을 폼이나 데이터 모듈에 올려 놓는다. 서버 소켓에서 제공하려는 서비스나 클라이언트의 요구를 기다릴 때 사용할 포트를 지정할 수 있다. 각 서버 소켓 컴포넌트는 서버 윈도우 소켓 객체(TServerWinSocket)를 사용하여 리스닝 연결에서의 서버측 종료점을 이루게 한

다. 또한, 서버가 받아들인 클라이언트 소켓과의 연결에서의 서버 종료점에 대한 클라이언트 윈도우 소켓 객체(TServerClientWinSocket)도 활용한다.

- 포트의 지정

서버 소켓이 클라이언트의 요구를 기다리기 전에 (이런 기다림을 ‘listening’ 이라고 한다.) 서버가 사용할 포트를 지정해 주어야 한다. 이때 Port 프로퍼티를 사용해서 포트를 지정할 수 있다. 서버 어플리케이션이 특정 포트 번호와 연관된 표준 서비스를 제공하는 경우라면 Service 프로퍼티를 지정함으로써 간접적으로 포트를 지정할 수도 있다. 만약에 Port 와 Service 프로퍼티를 모두 지정한 경우라면 서버 소켓은 서비스 이름을 사용하게 된다.

- 클라이언트 요구 대기 (Listening for client request)

일단 서버 소켓 컴포넌트의 포트 번호를 설정하면, 런타임에서 Open 메소드를 사용하여 리스닝 연결(listening connection)을 생성할 수 있게 된다. 만약에 어플리케이션이 시작할 때 리스닝 연결을 자동으로 시작하게 하고 싶으면, 디자인 시에 Active 프로퍼티를 True 로 설정하면 된다.

- 클라이언트에 대한 연결 생성

리스닝 서버 소켓 컴포넌트는 클라이언트 연결 요구를 받는 족족 이를 허용한다. 이렇게 되면 OnClientConnect 이벤트가 발생하며, 적절한 처리를 이벤트 핸들러에서 해주면 된다.

- 연결에 대한 정보 얻기

일단 서버 소켓에서 리스닝 연결을 시작하면, 서버 윈도우 소켓 객체를 사용해서 연결에 대한 정보를 얻을 수 있게 된다. 서버 윈도우 소켓 객체는 Socket 프로퍼티를 이용해서 접근이 가능하다. 이 윈도우 소켓 객체를 이용하면 서버 소켓 컴포넌트가 받아들인 클라이언트 소켓 객체들과의 모든 활성화된 연결을 찾아낼 수 있으며, SocketHandle 프로퍼티를 이용해서 소켓 연결에 대한 핸들을 얻을 수도 있다. 이 핸들을 사용해서 윈도우 소켓 API 를 직접 호출할 수도 있다. 또한, Handle 프로퍼티를 이용하면 소켓 연결에서 날아온 메시지를 받은 윈도우에 접근할 수 있다.

각각의 클라이언트 어플리케이션에 대한 활성화된 연결은 서버 클라이언트 윈도우 소켓 객체(TServerClientWinSocket)에 캡슐화 되어 있다. 이들 모두에게 접근할 때에는 서버 윈도우 소켓 객체의 Connections 프로퍼티를 이용하면 된다. 서버 클라이언트 윈도우 소켓 객체에는 연결의 양쪽 종료점을 형성하는 클라이언트와 소켓 객체에서 사용하는 포트 번호

와 주소를 결정할 수 있는 프로퍼티가 있으며, `SocketHandle` 이라는 프로퍼티를 사용하면 윈도우 소켓 API 호출을 할 때 사용할 수 있는 소켓 연결에 대한 핸들을 얻을 수도 있다. 또한, `Handle` 프로퍼티를 사용하면 소켓 연결에서의 메시지를 받은 윈도우에 접근할 수 있으며, `ASyncStyles` 프로퍼티에서 메시지의 종류를 결정할 수도 있다.

## ● 연결의 종료

리스닝 연결을 종료할 때에는 `Close` 메소드를 사용한다. 이 메소드를 통해 클라이언트 어플리케이션과의 모든 연결을 단절시킬 수 있으며, 리스닝 연결이 종료 되므로 서버 소켓은 더 이상 새로운 연결을 허용하지 않게 된다.

클라이언트가 서버 소켓과의 연결을 종료하면 `OnClientDisconnect` 이벤트가 발생하며, 적절한 이벤트 핸들러를 이용해서 정리를 해준다.

## 소켓 이벤트

앞에서 몇 가지 이벤트에 대하여 이미 언급한 바가 있다. 이를 간단히 먼저 정리하면 논-블로킹 연결이나 블로킹 연결에서 소켓 연결에 대해 언제 데이터를 읽고, 쓸 것인지에 대한 이벤트가 있으며, 서버 측에서 연결을 종료할 때 클라이언트 소켓에서 받게 되는 `OnDisconnect` 이벤트, 그리고 클라이언트에서 연결을 종료할 때 서버 소켓에서 받게 되는 `OnClientDisconnect` 이벤트가 있다.

또한, 클라이언트와 서버 소켓 모두 연결에서 에러 메시지를 받게 되면 `OnError` 이벤트가 발생한다. 그 밖에 연결을 열고, 완료할 때까지 여러가지 이벤트가 발생하게 된다.

## ● 클라이언트 이벤트

클라이언트 소켓이 연결을 열게 되면, 다음과 같은 이벤트 들이 순차적으로 발생한다.

1. 서버 소켓을 찾기 전에 `OnLookup` 이벤트가 발생한다. 이 시점에서는 찾는 서버 소켓을 바꾸기 위해 `Host`, `Address`, `Port`, `Service` 프로퍼티를 변경할 수 없다. `Socket` 프로퍼티를 이용해서 클라이언트 윈도우 소켓 객체에 접근할 수 있으며, 그 객체의 `SocketHandle` 프로퍼티를 이용해서 윈도우 API 를 호출할 수 있다.
2. 윈도우 소켓이 설정되고, 초기화 된다.
3. 일단 서버 소켓을 찾으면 `OnConnecting` 이벤트가 발생한다. 이 시점에서는 윈도우 소켓 객체를 사용해서 서버 소켓에 대한 정보를 얻을 수 있게 되며, 연결에 사용되는 실제 포트와 IP 주소를 알 수 있다.
5. 연결 요구가 서버에 의해 받아들여지면 클라이언트 소켓에서 연결이 완료된다.

6. 연결이 완료되면 OnConnect 이벤트가 발생한다. 연결이 완료되는 즉시 데이터를 읽고, 쓰는 작업을 해야 할 때에는 OnConnect 이벤트 핸들러에 적절한 코드를 작성하면 된다.

## ● 서버 이벤트

서버 소켓 컴포넌트는 리스닝 연결과 클라이언트 연결의 두가지 형태의 연결을 형성할 수 있다. 이들 각각에 따라 발생하는 이벤트가 다르기 때문에, 따로 나누어 설명하겠다.

### - 리스닝 연결 이벤트

리스닝 연결이 생성되기 직전에 OnListen 이벤트가 먼저 발생한다. 이 시점에서 Socket 프로퍼티를 이용해서 서버 윈도우 소켓 객체에 접근할 수 있게 된다. 이 객체의 SocketHandle 프로퍼티를 이용해서 연결이 생성되기 전에 소켓에 대한 여러가지 사항을 바꾸어 볼 수가 있다. 예를 들어, 서버가 리스닝에 사용하는 IP 주소를 제한하는 등의 처리를 할 수가 있다.

### - 클라이언트 연결 이벤트

1. 연결의 서버측 종료점을 형성하는 소켓에 대한 윈도우 소켓 핸들을 넘겨주는 OnGetSocket 이벤트가 클라이언트 측에 소켓 객체를 넘겨줄 때 발생한다. 이때 개발자가 TServerClientWinSocket 클래스 대신에 자신만의 윈도우 소켓 객체를 대신 사용하도록 할 수 있다. 즉, OnGetSocket 이벤트 핸들러에서 자신의 윈도우 소켓 객체를 생성할 수 있다.
2. 이어서 OnAccept 이벤트가 발생하는데, 여기에는 새로운 TServerClientWinSocket 객체를 이벤트 핸들러에 넘겨 준다. 여기에서 처음으로 TServerClientWinSocket 객체를 이용해서 클라이언트에 연결된 서버측 종료점에 대한 정보를 이용할 수 있게 된다.
3. ServerType 이 stThreadBlocking 이면 OnGetThread 이벤트가 발생한다. 여기에서 자신만의 TServerClientThread 객체를 생성할 수 있으며, 이를 대신 사용하게 된다. 이어서 쓰레드가 실행되기 시작하면 OnThreadStart 이벤트가 발생한다. 쓰레드의 초기화나 쓰레드가 연결을 통해 데이터를 읽고, 쓰는 작업을 하려고 윈도우 소켓 API를 호출해야 한다면, 이 이벤트 핸들러에서 작업을 한다.
4. 클라이언트가 연결을 완료하면 OnClientConnect 이벤트가 발생한다. 논-블로킹 서버의 경우에는 이 시점에서 데이터를 읽고, 쓰기를 시작하면 된다.

## 1:1 채팅 예제의 제작

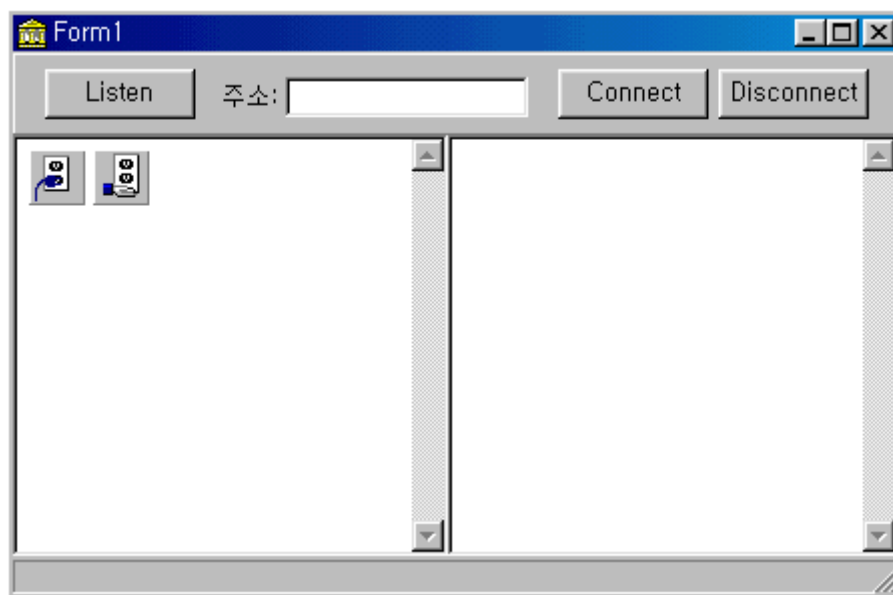
그러면, 실제로 예제를 만들어 나가면서 지금까지 설명한 것들을 익혀 보도록 하자.

이번에 만들 예제는 1:1 채팅을 가능하게 하는 프로그램으로 하나의 어플리케이션에 TClientSocket 과 TServerSocket 을 모두 올려 놓고, 이 프로그램이 경우에 따라서 채팅 서버가 되기도 하고, 클라이언트가 되기도 하는 프로그램이다.

본래 채팅 프로그램을 제대로 만들려면 서버 프로그램에 여러 개의 클라이언트가 접속하는 형태로 제작해야 하지만, 이 예제는 네트워크 프로그래밍의 기본을 이해시키려는 목적으로 제작하는 것이므로 1:1 채팅 만을 지원하도록 하였다.

이런 식으로 클라이언트와 서버의 기능을 모두 갖춘 프로그램은 프로그램을 테스트 하기에 편리하고 실제 메시지를 처리하면서 클라이언트와 서버에서 메시지를 처리하는 방법을 동시에 익힐 수 있는 장점이 있다.

먼저 폼에 메모 컴포넌트를 2 개와 TPanel, TStatusBar 컴포넌트를 하나씩 넣는다. 패널 위에는 버튼 컴포넌트 3 개와 IP 주소를 입력할 에디트 박스를 하나 추가하여 다음과 같이 디자인한다. 물론 1:1 채팅 프로그램을 만들기 위해서 TClientSocket 과 TServerSocket 컴포넌트도 추가해야 할 것이다.



그리고 포트를 결정해야 하는데, 필자는 1001 번으로 결정하였다. ClientSocket1 과 ServerSocket1 컴포넌트의 Port 프로퍼티를 1001 로 설정하였다. 그리고, StatusBar1 컴포넌트를 선택하고 오른쪽 버튼을 클릭한 후 Panels Editor... 메뉴를 선택하여 패널 에디터를 띄운 뒤에 Add New(Ins) 버튼을 클릭하여 StatusPanel 을 하나 추가한다.

이제 이 1:1 채팅 어플리케이션은 클라이언트이면서 동시에 서버로 동작할 수 있도록 준비가 끝난 셈이다. 이를 위해서 전역 변수를 2 개 추가해야 하는데, 하나는 채팅 어플리케이션



선이 클라이언트가 접속하기를 기다리는지 여부를 결정하는 Listening 변수와 현재 서버로 동작하고 있는지를 나타낼 IsServer 변수가 그것이다.

```
var  
    Form1: TForm1;  
    Listening: Boolean;  
    IsServer: Boolean;
```

폼의 OnCreate, OnClose 이벤트 핸들러를 다음과 같이 작성하여 처음 폼이 생성될 때에는 Listening 변수를 초기화 하고, 폼을 닫을 때에는 소켓을 닫도록 한다.

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Listening := False;  
end;  
  
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    ServerSocket1.Close;  
    ClientSocket1.Close;  
end;
```

Listen 버튼의 OnClick 이벤트에서는 현재의 폼을 서버로 대기하도록 하는 기능을 한다. 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Listening := not Listening;  
    if Listening then  
    begin  
        ClientSocket1.Active := False;  
        ServerSocket1.Active := True;  
        StatusBar1.Panels[0].Text := 'Listening...';  
    end  
    else  
    begin
```

```

        if ServerSocket1.Active then
            ServerSocket1.Active := False;
            StatusBar1.Panels[0].Text := '';
        end;
end;

```

이 버튼을 클릭하면 현재의 Listening 변수의 값을 변경한다. 그리고, 이 변수의 값이 True 이면 ServerSocket1 의 Active 프로퍼티를 True 로 설정하고 StatusPanel 의 Text 프로퍼티를 'Listening...'으로 설정한다. 이 값이 False 이면 서버 소켓의 Active 프로퍼티를 False 로 설정한다.

Connect 버튼을 클릭하면 Edit1 의 내용을 ClientSocket1 컴포넌트의 Address 프로퍼티로 설정한다. 참고로 앞서도 설명했듯이 소켓의 Address 와 Host 프로퍼티 중에서 하나를 이용하는데, Address 프로퍼티는 IP 주소를 이용한다. 그리고, 이 주소를 이용하여 서버 소켓에 접속한다.

그리고, Disconnect 버튼을 클릭하면 클라이언트 소켓을 닫고, Listen 버튼의 OnClick 이벤트 핸들러를 호출한다.

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    if ClientSocket1.Active then ClientSocket1.Active := False;
    if Length(Edit1.Text) > 0 then
        begin
            ClientSocket1.Address := Edit1.Text;
            ClientSocket1.Active := True;
        end;
    end;
end;

```

```

procedure TForm1.Button3Click(Sender: TObject);
begin
    ClientSocket1.Close;
    Button1Click(nil);
end;

```

이제 각 버튼의 OnClick 이벤트 핸들러는 모두 작성하였다. 이제부터 서버 소켓과 클라이언트 소켓의 이벤트 핸들러를 하나씩 작성하면서 이들의 역할을 알아보도록 하자.

먼저 서버 소켓의 이벤트 핸들러를 작성하도록 하자. 앞서도 설명했듯이 서버 소켓에 클

라이언트 소켓이 접속을 시도하면 OnGetSocket 이벤트에 이어서 OnAccept 이벤트가 발생한다. 이 이벤트에서는 서버가 클라이언트의 접속을 받아들이기로 할 때 발생하는 이벤트이므로 IsServer 변수를 True 로 설정하여 이제 어플리케이션이 서버로 동작하고 있음을 나타내게 하고, StatusBar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteAddress; OnAccept 이벤트의 Socket 파라미터는 클라이언트 소켓을 나타낸다.

```
procedure TForm1.ServerSocket1Accept(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    IsServer := True;
    StatusBar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteAddress;
end;
```

이어서 클라이언트가 연결을 완료하면 OnClientConnect 이벤트가 발생한다. 지금 작성하고 있는 채팅 어플리케이션과 같은 논-블로킹 서버의 경우에는 이 시점에서 데이터를 읽고, 쓰기를 시작할 수 있다. Memo2 컴포넌트에는 클라이언트 소켓에서 발생한 메시지를 표시할 것이므로, 이 시점에서부터 내용을 보여주도록 메모 컴포넌트의 내용을 지우도록 한다. 그리고, OnRead 이벤트는 서버 소켓이 클라이언트 소켓으로부터 데이터를 전달받을 때 발생하는데 Socket 파라미터의 ReceiveText 프로퍼티에 클라이언트 소켓에서 전송한 텍스트 값이 들어가 있다. 그러므로 이를 Memo2 컴포넌트에 보여주도록 이벤트 핸들러를 작성한다.

```
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Memo2.Lines.Clear;
end;
```

```
procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Memo2.Lines.Add(Socket.ReceiveText);
end;
```

마지막으로 연결이 종료될 때에 발생하는 OnClientDisconnect 이벤트 핸들러에서는 서버 소켓의 Active 프로퍼티를 False 로 설정하고, 처음의 Listening 상태로 들어가기 위해서

Button1Click 이벤트 핸들러를 다시 호출한다. 이때 이를 호출하면 Listening 변수의 값이 변경되므로 먼저 Listening 변수 값을 변경한 뒤에 호출해야 원래의 값이 보존될 것이다.

```
procedure TForm1.ServerSocket1ClientDisconnect(Sender: TObject;  
    Socket: TCustomWinSocket);  
begin  
    ServerSocket1.Active := False;  
    Listening := not Listening;  
    Button1Click(nil);  
end;
```

이번에는 클라이언트 소켓의 이벤트 핸들러를 작성할 차례이다. 클라이언트 소켓도 서버 소켓과 접속이 되었을 때 OnConnect 이벤트가 발생한다. 여기에서도 마찬가지로 접속된 서버 소켓이 위치한 컴퓨터의 이름을 나타내도록 하는데, Socket 파라미터의 RemoteHost 프로퍼티를 사용하면 마이크로소프트의 UNC 이름에 해당되는 컴퓨터 이름이 나타난다.

```
procedure TForm1.ClientSocket1Connect(Sender: TObject;  
    Socket: TCustomWinSocket);  
begin  
    StatusBar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteHost;  
end;
```

이어서 나타날 수 있는 OnRead 이벤트 핸들러에서는 서버 소켓을 Socket 파라미터에서 얻을 수 있다. 서버의 텍스트 내용을 클라이언트로 동작하고 있는 어플리케이션의 메모 컴포넌트에 보여주도록 다음과 같이 이벤트 핸들러를 작성한다.

```
procedure TForm1.ClientSocket1Read(Sender: TObject;  
    Socket: TCustomWinSocket);  
begin  
    Memo2.Lines.Add(Socket.ReceiveText);  
end;
```

클라이언트 소켓의 OnDisconnect 이벤트 핸들러에서는 Button1Click 이벤트 핸들러를 호출하여 서버 소켓으로 동작할 수 있도록 한다.

```
procedure TForm1.ClientSocket1Disconnect(Sender: TObject;
```

```

    Socket: TCustomWinSocket);
begin
    Button1Click(nil);
end;

```

그리고, 이런 접속과정에서 에러가 발생할 경우에는 OnError 이벤트가 발생하는데, 이벤트 핸들러를 다음과 같이 작성하여 발생한 에러 상황을 나타내도록 한다.

```

procedure TForm1.ClientSocket1Error(Sender: TObject;
    Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;
    var ErrorCode: Integer);
begin
    Memo2.Lines.Add('Error connecting to : ' + Edit1.Text);
    ErrorCode := 0;
end;

```

마지막으로 Memo1 컴포넌트의 OnKeyDown 이벤트 핸들러에서 눌러진 키가 리턴 키일 경우에 소켓으로 전송하도록 한다. 이때 IsServer 변수를 검사하여 서버일 경우와 클라이언트일 경우에 서로 다른 소켓에다가 SendText 메소드를 이용하여 텍스트를 전송한다.

```

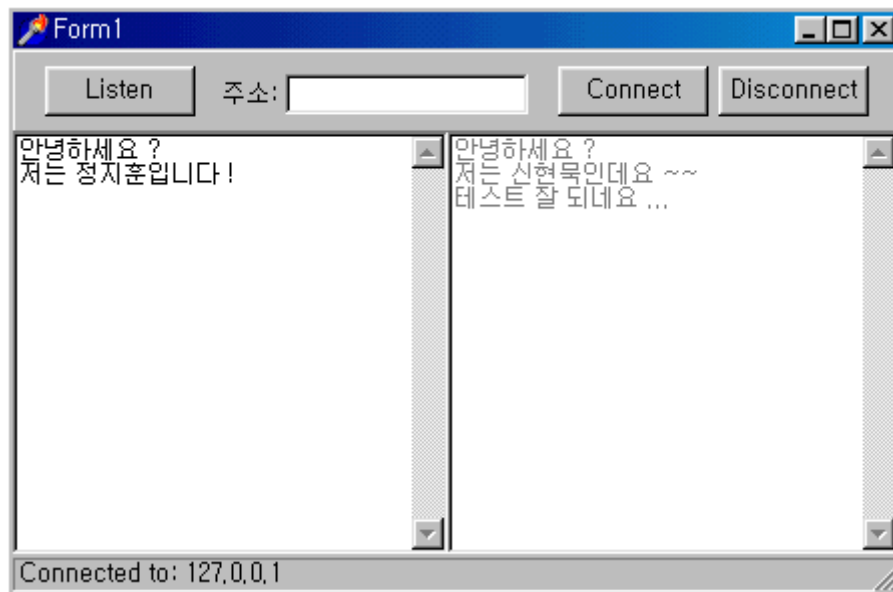
procedure TForm1.Memo1KeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);
begin
    if Key = VK_Return then
        if IsServer then
            ServerSocket1.Socket.Connections[0].SendText(Memo1.Lines[Memo1.Lines.Count - 1])
        else
            ClientSocket1.Socket.SendText(Memo1.Lines[Memo1.Lines.Count - 1]);
end;

```

여기서 클라이언트 소켓의 경우에는 간단하지만, 서버 소켓의 경우 여러 개의 클라이언트 소켓과 물릴 수 있기 때문에 Connections 라는 컬렉션 객체가 포함되어 있다. 그렇지만, 우리가 작성한 어플리케이션은 단지 1:1 통신만 지원하므로 Connections[0]으로 접속된 클라이언트 소켓을 지칭할 수 있다.

이것으로 간단한 1:1 통신을 지원하는 채팅 어플리케이션이 완성되었다. 컴파일하고 실행한 뒤에 서버와 클라이언트 컴퓨터에 각각 띄우도록 한다.

그리고, 서버 측에서는 Listen 버튼을 클릭하여 클라이언트 프로그램의 접속을 대기하도록 하고, 클라이언트 측에서는 에디트 박스에 IP 주소를 적어 넣은 뒤에 Connect 버튼을 클릭하여 서버에 접속하도록 하자. 성공적으로 접속이 되면, 상태 바에 연결된 컴퓨터의 IP 주소(서버 컴퓨터의 경우) 또는 연결된 컴퓨터의 컴퓨터 이름(클라이언트 컴퓨터의 경우)이 나타날 것이다. 이제 메모 컴포넌트에서 통신을 시도하면 다음과 같이 채팅을 할 수 있을 것이다.



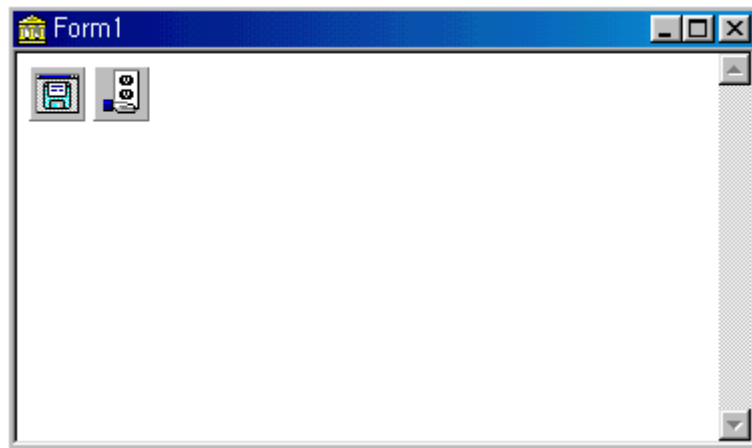
## 블로킹 연결을 이용한 파일 전송 예제

소켓을 이용한 프로그래밍을 할 때 앞서 설명한 채팅 어플리케이션과 같이 연결을 유지하면서 통신을 할 필요가 있을 때에는 중단되지 않는 논-블로킹 연결을 지원하도록 해야 하지만, 경우에 따라서는 전송하는 측과 전송받는 측의 데이터 전송에 있어서 서로 비동기 적으로 처리하는 것이 효율적일 때가 많다. 이럴 때에는 블로킹 연결을 이용하게 되는데, 블로킹 연결을 이용하여 소켓 프로그래밍을 하는 것은 해당되는 연결의 스레드를 생성하여 이를 실행하도록 하는 형태로 제작해야하기 때문에 논-블로킹 연결을 지원하는 어플리케이션에 비해 다소 까다로운 점이 많다.

그러면, 블로킹 연결을 통해 클라이언트에서 서버로 지정된 파일을 전송하는 예제를 작성해보도록 하자. 이 예제는 Stig Johansen 이 공개한 예제를 바탕으로 작성하였다. 예제 프로그램의 구조는 클라이언트 어플리케이션에서 파일 열기 대화 상자에서 지정한 파일을 에디트 박스에 지정한 IP 주소로 전송한다. 그리고, 서버 어플리케이션에서는 클라이언트와 연결되면 서버에 저장할 파일 이름을 지정하면, 여기에 파일을 저장하도록 한다.

먼저, 서버 어플리케이션을 작성하도록 하자. 폼 위에 다음과 같이 메모 컴포넌트와 서버 소켓, 파일 저장 대화상자 컴포넌트를 하나씩 추가하고, 메모 컴포넌트의 Align 프로퍼티는

alClient, ScrollBars 프로퍼티는 ssVertical, ReadOnly 프로퍼티는 True 로 설정한다. 그리고 ServerSocket1 컴포넌트의 ServerType 프로퍼티는 stThreadBlocking, Port 프로퍼티는 2001 로 설정하도록 하자.



블로킹 연결을 지원하는 서버를 작성하기 위해서는 TServerClientThread 에서 상속받은 쓰레드 클래스를 이용하는 것이 핵심이다. 여기서는 파일과 소켓의 스트림을 내부적으로 사용하여 파일의 전송을 구현하기 때문에, private 섹션에 소켓 스트림과 파일 스트림 필드를 추가하고 외부에서 접근할 수 있도록 public 섹션에 프로퍼티로 선언하도록 한다.

```
TServerThread = class(TServerClientThread)
private
    FSocketStream: TWinSocketStream ;
    FFileStream: TFileStream;
protected
    procedure ClientExecute; override;
public
    property SocketStream: TWinSocketStream read FSocketStream write FSocketStream ;
    property FileStream: TFileStream read FFileStream write FFileStream ;
end;
```

쓰레드 클래스에서 꼭 오버라이드해서 구현해야 하는 메소드가 ClientExecute 로, 다중 쓰레드를 지원하는 경우에 Execute 메소드를 오버라이드하는 것과 같은 역할을 한다. 다중 쓰레드 프로그래밍에 대해서는 41 장에서 자세히 다루므로 이를 참고하기 바란다.

폼의 OnCreate 이벤트 핸들러에서 서버 소켓의 Active 프로퍼티를 True 로 설정하여 서버 소켓이 클라이언트 소켓과의 연결을 받아들일 수 있도록 이를 열어놓도록 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ServerSocket1.Active := True;
end;
```

그리고 서버 소켓의 OnAccept, OnListen, OnThreadStart, OnThreadStop 이벤트 핸들러를 다음과 같이 작성하여 클라이언트 소켓과의 연결 상황을 메모 컴포넌트에 나타내도록 한다.

```
procedure TForm1.ServerSocket1Accept(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Memo1.Lines.Add('Accept ' + Socket.RemoteAddress);
end;
```

```
procedure TForm1.ServerSocket1Listen(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Memo1.Lines.Add('Listening ... ');
end;
```

```
procedure TForm1.ServerSocket1ThreadStart(Sender: TObject;
    Thread: TServerClientThread);
begin
    Memo1.Lines.Add('Start Thread of ' + Thread.ClientSocket.LocalAddress);
end;
```

```
procedure TForm1.ServerSocket1ThreadEnd(Sender: TObject;
    Thread: TServerClientThread);
begin
    Memo1.Lines.Add('End Thread');
end;
```

블로킹 연결에 있어서 가장 중요한 것은 OnGetThread 이벤트 핸들러에서 서버 쓰레드를 생성하는 작업이다. 서버 쓰레드를 생성할 때 두번째 파라미터를 False 로 선언하면 생성과 동시에 실행되는 것이지만, 다음과 같이 True 로 설정하면 쓰레드 객체의 프로퍼티를 변경한 뒤에 Resume 메소드로 쓰레드가 실행된다.



```

procedure TForm1.ServerSocket1GetThread(Sender: TObject;
  ClientSocket: TServerClientWinSocket;
  var SocketThread: TServerClientThread);
var
  SocketStream: TWinSocketStream ;
  FileStream: TFileStream ;
  FileName: string;
begin
  FileName := 'Default.file';
  SocketStream := TWinSocketStream.Create(ClientSocket, 20);
  if SaveDialog1.Execute then FileName := SaveDialog1.FileName;
  FileStream := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
  SocketThread := TServerThread.Create(True, ClientSocket);
  (SocketThread as TServerThread).SocketStream := SocketStream;
  (SocketThread as TServerThread).FileStream := FileStream;
  SocketThread.FreeOnTerminate := True ;
  SocketThread.Resume ;
end;

```

여기에서 전송되어온 파일의 이름을 대화 상자에서 선택할 수 있도록 하고, 파일 스트림 객체를 생성하여 쓰레드 객체의 프로퍼티에 대입하고, 마찬가지로 TWinSocketStream 객체를 생성하여 이를 소켓 스트림 프로퍼티에 대입한다.

이제 쓰레드가 실제로 실행되는 ClientExecute 메소드를 구현하면 서버 프로그램이 완성된다. 이를 구현하기에 앞서 소켓 스트림의 전체 내용을 읽어오는 역할을 하는 ReadStream 함수를 다음과 같이 구현한다.

```

function ReadStream (Stream: TWinSocketStream; Buffer: Pointer;
  Count: Integer): Boolean;
var
  P: PChar;
  Total, Delta, TimeOut: Integer;
begin
  if Count = 0 then
    begin
      Result := True;
    end
  else
    begin
      Total := 0;
      Delta := 1;
      TimeOut := 1000;
      while (Total < Count) do
        begin
          Delta := 1;
          while (Delta < Count - Total) do
            begin
              if not ReadStream(Stream, P + Total, Delta) then
                begin
                  Result := False;
                  break;
                end
            end
          end
          Total := Total + Delta;
          Delta := 1;
        end
      end
    end
  end
end;

```

```

Exit;
end;
TimeOut := 0;
Result := True;
Total := 0;
P := Buffer;
while Total < Count do
begin
try
Delta := Stream.Read(P^, Count - Total);
except
Exit;
end;
if Delta = 0 then
begin
Inc(Timeout);
while not Stream.WaitForData(1000) and (TimeOut < 20) do Inc(TimeOut);
if Timeout >= 20 then
begin
Result := False;
Exit;
end;
end
else
TimeOut := 0;
Inc(P, Delta);
Inc(Total, Delta);
end;
end;
end;

```

이 루틴은 꽤 유용하게 사용되므로 잘 익혀두기 바란다. 구현된 내용을 간단히 설명하면 소켓 스트림에서 데이터를 읽어올 때 버퍼와 시간 제한을 이용하여 적절한 버퍼링을 해주는 것이 주된 내용이다. 이런 작업이 필요한 이유는 이런 버퍼링 작업이 없이 송신 측에서는 무조건 데이터를 밀어 넣고, 수신 측에서는 무조건 데이터를 가져올 경우에는 간혹 손실되는 패킷이 생기기 때문이다. 실제로 뉴스 그룹에서도 이런 문제로 어려움을 겪는 많은 개발자들이 있었는데, 이런 문제를 이 루틴으로 해결할 수 있다.

소스를 보면 쉽게 이해할 수 있을 것이나 간단한 설명을 덧붙이자면, 스트림의 Read 메소드에 의해 실제 읽어온 데이터의 바이트 수를 Delta 라는 변수에 대입하게 되는데 이때 이 값이 0 이면 WaitForData 메소드를 이용하여 버퍼에 데이터가 들어오는지 기다려 보고, 이를 여기서는 최대 20 번까지 기다려본 후에 그래도 데이터가 전송되어오지 않으면 연결이 끊어진 것으로 간주하고 실행을 중지하게 된다.

이 루틴을 이용하여 TServerThread 객체의 ClientExecute 메소드는 다음과 같이 구현한다.

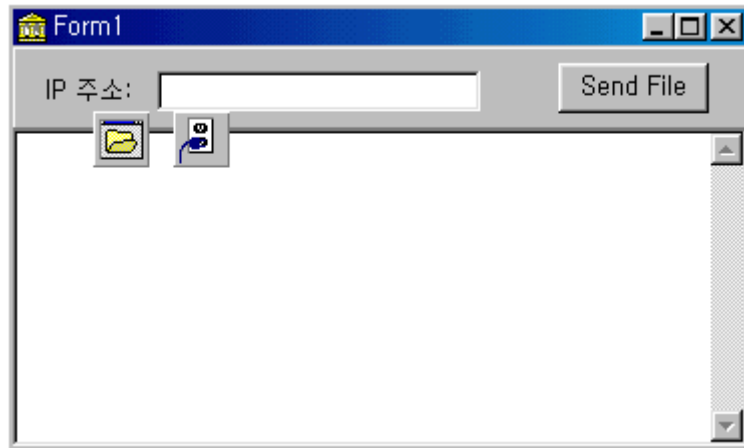
```
procedure TServerThread.ClientExecute;
var
    FileLength: Integer;
    MemoryStream: TMemoryStream;
begin
    if ReadStream(SocketStream, Addr(FileLength), SizeOf(FileLength)) then
    begin
        MemoryStream := TMemoryStream.Create;
        MemoryStream.SetSize(FileLength);
        ReadStream(SocketStream, MemoryStream.Memory, FileLength);
        FFileStream.CopyFrom(MemoryStream, MemoryStream.Size);
        MemoryStream.Free;
    end;
    if Assigned(FSocketStream) then FSocketStream.Free;
    if Assigned(FFileStream) then FFileStream.Free;
    Terminate;
end;
```

여기서 눈여겨 보아야 할 것은 파일 스트림과 소켓 스트림의 원활한 전달을 위해 중간에 TMemoryStream 형의 변수를 이용한다는 점이다. 그리고, 처음에 일단 ReadStream 메소드를 이용하여 전달된 파일의 크기를 먼저 받아본다는 점이 중요하다. 이는 클라이언트에서 파일을 전송할 때 먼저 파일의 크기를 전송하고 나서, 실제 파일을 전송한다는 것을 의미한다. 그리고, 이 값을 이용하여 서버에서 파일을 생성하고 스트림을 복사한다.

이것으로 서버 프로그램이 완성되었다.

이번에는 이 서버 프로그램과 연결해서 사용할 클라이언트 프로그램을 작성해보자.

서버와는 달리 클라이언트 프로그램에는 연결한 서버의 IP 주소를 적어넣을 에디트 박스와 버튼을 하나의 패널에 올려 놓고, 메모 컴포넌트를 다음과 같이 추가하여 디자인하도록 하자.



클라이언트 소켓의 ClientType 프로퍼티는 ctBlocking 으로 설정하고, Port 프로퍼티는 서버와 같은 2001 로 설정한다.

그리고 먼저 클라이언트 소켓의 OnConnect, OnDisconnect, OnError 이벤트 핸들러를 다음과 같이 작성하여 통신 상황을 나타내도록 한다.

```
procedure TForm1.ClientSocket1Connect(Sender: TObject;
```

```
    Socket: TCustomWinSocket);
```

```
begin
```

```
    Memo1.Lines.Add('Connected to ' + Socket.RemoteAddress);
```

```
end;
```

```
procedure TForm1.ClientSocket1Disconnect(Sender: TObject;
```

```
    Socket: TCustomWinSocket);
```

```
begin
```

```
    Memo1.Lines.Add('Disconnected to ' + Socket.RemoteAddress);
```

```
end;
```

```
procedure TForm1.ClientSocket1Error(Sender: TObject;
```

```
    Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;
```

```
    var ErrorCode: Integer);
```

```
begin
```

```
    Memo1.Lines.Add('Error Code is ' + IntToStr(ErrorCode));
```

```
end;
```

그리고, 서버 프로그램에서의 ReadStream 과 마찬가지로 클라이언트에서 파일의 내용을 파

일 스트림에 읽어들이고 후, 이를 소켓 스트림에 기록하는 역할을 하는 WriteStream 함수를 다음과 같이 구현한다.

```
procedure WriteStream(Stream: TWinSocketStream; const Buffer: Pointer;
    Count: Integer);
var
    P: PChar;
    Total, Delta, TimeOut: Integer;
begin
    if Count = 0 then Exit;
    Total := 0;
    TimeOut := 0;
    Delta := 0;
    P := Buffer;
    while Total < Count do
        begin
            try
                Delta := Count - Total;
                if Delta > 16384 then Delta := 16384;    //최대값
                Delta := Stream.Write(P^, Delta);
            except
                Exit;
            end;
            Inc(P, Delta);
            Inc(Total, Delta);
        end;
    end;
```

비교적 간단한 구현 내용이므로 쉽게 이해할 수 있을 것이다. 참고로 여기서는 한번에 최대 16384 바이트를 하나의 패킷으로 스트림에 기록하도록 하였다. 소켓 연결 상태 등에 따라 크기를 변경할 수도 있겠다.

마지막으로 Button1의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    FileStream: TFileStream;
```

```

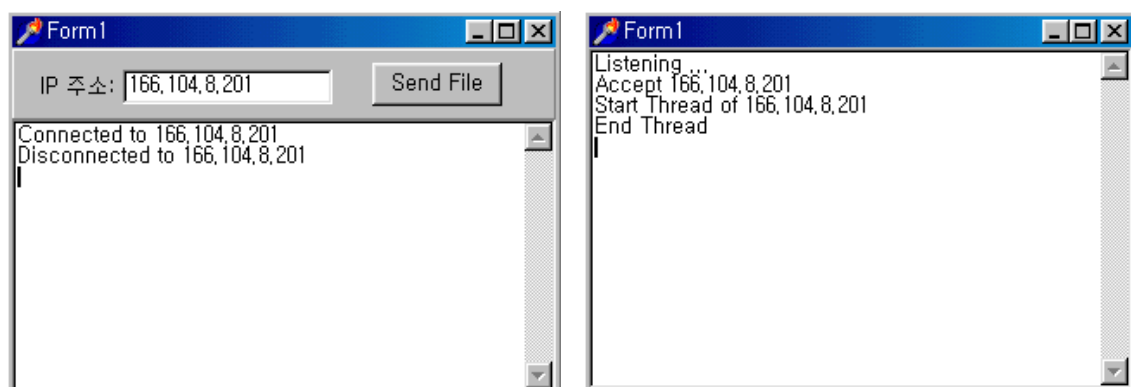
FileLength: Integer;
begin
  if OpenFileDialog1.Execute then
    begin
      FileStream := TFileStream.Create(OpenDialog1.FileName, fmOpenRead
        or fmShareDenyNone);
      FileLength := FileStream.Size;
      if FileLength > 0 then
        begin
          ClientSocket1.Address := Edit1.Text;
          ClientSocket1.Active := True;
          if ClientSocket1.Active then
            begin
              ClientSocket1.Socket.SendBuf(FileLength, SizeOf(FileLength));
              ClientSocket1.Socket.SendStream(FileStream);
            end;
            ClientSocket1.Active := False;
          end;
        end;
      end;
    end;
end;

```

일단 파일 열기 대화상자를 이용하여 전송할 파일을 선택하게 하고, 이 파일에 대한 파일 스트림 객체를 생성한다. 그리고, 파일 스트림의 크기를 먼저 SendBuf 메소드를 이용하여 전송하고, 파일 스트림의 내용을 SendStream 메소드를 이용하여 전송한다.

이와 같이 소켓을 이용하여 스트림을 전송할 때에는 패킷을 나누어 전송하고, 스트림의 크기를 먼저 전송하게 하는 것이 에러를 줄일 수 있는 요령이다. 물론, 독자적인 프로토콜을 정의하여 이를 이용하는 것이 가장 이상적일 것이다.

이것으로 클라이언트 어플리케이션이 완성되었다. 이제 클라이언트와 서버 어플리케이션을 띄우고 파일을 선택하여 전송하도록 해보자. 다음 그림은 서버와 클라이언트 어플리케이션의 실행화면이다.



## 정 리 (Summary)

이번 장에서는 텔파이에서 기본적으로 제공되는 소켓 컴포넌트를 이용하여 프로그래밍을 하는 방법에 대해서 알아보았다. 소켓 컴포넌트는 과거 뉴스그룹에서 텔파이 판매 전략에서 C/S 버전에만 포함된 것이 격렬한 논란거리가 된 컴포넌트이다. 그렇기 때문에, 원속을 지원하는 수많은 프리웨어 컴포넌트 들이 나오게 되었고, 텔파이 4 에서는 프로페셔널 버전에 소켓 컴포넌트만 추가한 새로운 제품군을 등장시켰다. 그만큼 사용 빈도도 높고, 또한 쓸 모도 많다.

그러므로, 여기에서 소개한 소켓 컴포넌트 뿐만 아니라, Francois Piette 가 공개한 프리웨어 컴포넌트 들도 매우 뛰어나고 더 잘된 것들도 많으므로 이런 프리웨어에도 관심을 두고 찾아보기 바란다.

## 직렬 통신 컴포넌트의 제작

### (Creating Serial Communication Components)

이번 장에서는 윈도우 API 함수를 이용해서 시리얼 통신을 제어할 수 있는 방법을 소개하고, 이를 바탕으로 간단한 시리얼 통신 컴포넌트를 제작하도록 할 것이다.

시리얼 통신과 관련해서는 여러가지 프리웨어 컴포넌트를 인터넷에서 찾을 수 있으므로, 자신에게 맞는 컴포넌트를 찾아서 사용하는 것도 하나의 방법이 될 것이다. 그러나, 기본적으로 직렬 통신 포트에 접근해서 이를 사용하는 방법은 익혀두는 것이 좋다.

#### 시리얼 통신의 기초

시리얼 포트는 데이터를 보내고 받는 두가지 일을 한다. 이것이 대단히 간단하게 생각되겠지만, 실제로는 이를 위해서 여러가지 일들이 벌어진다. 시리얼 포트는 컴퓨터의 연산 속도에 비해 훨씬 느리게 동작하기 때문에, 파일 등을 시리얼 포트로 보낸다고 생각할 때 적절한 버퍼의 활용과 흐름 제어(flow control)가 필수적이다.

이러한 흐름 제어의 방법으로 가장 흔히 쓰이는 것이 RTS/CTS와 XON/XOFF 제어이다.

RTS(request to send)와 CTS(clear to send)는 시리얼 포트의 하드웨어적인 흐름 제어 방법이다. 포트의 RTS 라인은 원격 디바이스의 CTS 라인과 연결되어 있고, CTS 라인은 RTS 라인과 연결되어 있다. 원격 디바이스가 데이터를 받을 준비가 되면 RTS 라인을 활성화 시키며, 이렇게 되면 이와 연결된 CTS 라인에서 데이터를 보내기 시작한다. 원격 디바이스가 충분한 양의 데이터를 받으면 RTS 라인을 끄게 되고, 이것이 데이터를 보내지 말라는 신호가 된다. 이러한 사이클이 데이터가 모두 전송될 때까지 반복된다.

XON/XOFF 방법은 소프트웨어적인 흐름 제어 방법으로 XON/XOFF 문자를 보내서 제어하는 방법이다. 일단 시리얼 포트에서 데이터를 전송하기 시작해서 버퍼가 차게 되면 XOFF 문자(ASCII 13h)를 전송한다. 이 문자를 받으면 시리얼 포트는 데이터의 전송을 일단 중단한다. 데이터를 계속 받아서 버퍼에 여유가 생기면 이번에는 XON 문자(ASCII 11h)를 전송해서 시리얼 포트가 데이터 전송을 계속하게 된다.

이 두가지 방법은 서로 다른 방식으로 결국 비슷한 역할을 하게 되는데, 동시에 두가지 모두를 사용할 수도 있다.

일단 데이터가 전송되는 도중에는 보내고, 받은 데이터가 제대로 전송했는지 검사하는 것도 중요하다. 데이터의 전송이 제대로 이루어 졌는지 검사하는 방법에는 여러가지가 있는데, 그중 가장 원시적인 것인 패리티(parity) 에러를 검사하는 것이다. 패리티 에러 검사의 방법에는 even, odd, mark, space 등의 것이 있는데, 실용성이 많이 떨어지기 때문에 거의 사용되지 않는다.



이러한 패리티 검사 방법 외에 실제로 사용되는 것으로 CRC(cyclic redundancy check) 검사를 많이 한다. 이 방법은 전송된 데이터의 순서와 양을 체크 점을 이용해서 검사하는 것으로 비교적 효과적이면서도 정확하다.

흐름 제어 만큼이나 중요한 것이 연결된 두 시리얼 포트가 서로의 상태를 알아보고 이를 제어 하는 방법이다. 여기에 대한 방법에도 하드웨어적인 방법과 소프트웨어적인 방법이 있다. 하드웨어적인 방법은 DSR(data set ready)/DTR(data terminal ready) 라인을 이용하는 것이다. 예를 들어 두 개의 모뎀이 연결되면 각각의 모뎀은 DTR 라인을 활성화 시킨다(이를 ‘hi’ 또는 ‘hot’ 이라고도 한다.). 한 모뎀의 DTR 라인은 다른 모뎀의 DSR 라인과 연결되어 있는데, 양쪽 모뎀이 다른 모뎀으로부터 신호를 받게 되면 바로 하드웨어적인 핸드 셰이킹을 하게 된다. 그렇지만 이 방법은 비교적 오래된 방법으로, 최근에는 소프트웨어적인 방법이 더욱 효과적이고 더 많이 쓰인다. DSR/DTR 방법은 흐름 제어의 한 수단으로 사용되고 있다.

소프트웨어적인 방법은 모뎀이 연결될 때 잡음이 많이 섞이게 되는데 흐름 제어, 압축 레벨, 보드 전송율 등을 조절해서 여기에 대처하게 된다. 간단한 시리얼 통신에서는 이러한 방법이 사용되지 않지만, 파일 전송 등의 비교적 복잡한 작업을 할 때에는 필수적으로 사용된다. Win32 API 에는 이러한 시리얼 통신을 지원하기 위해 비교적 많은 함수가 준비되어 있다. 지금 부터 이들을 하나씩 알아보고, 이를 쉽게 사용할 수 있는 컴포넌트를 하나 만들어 보자.

## 포트 열기

가장 중요한 것이 처음으로 포트를 여는 것이다. 이를 위해서 CreateFile 함수가 사용된다. 이 함수의 선언부는 다음과 같다.

```
function CreateFile(lpFileName: PChar; dwDesiredAccess, dwShareMode: Integer;  
    lpSecurityAttributes: PSecurityAttributes; dwCreationDisposition,  
    dwFlagsAndAttributes: DWORD; hTemplateFile: THandle): THandle;
```

이 함수가 성공적으로 수행되면 시리얼 포트에 대한 핸들을 돌려주게 된다. 첫번째 파라미터로 사용되는 lpFileName 에는 사용할 포트의 이름을 지정한다, 예를 들어 ‘COM1’, ‘COM2’ 등을 지정한다. dwDesiredAccess 파라미터에는 포트에 접근하는 방법을 지정하는데, 포트를 통해서 데이터를 읽고, 쓰기를 모두 한다면 ‘GENERIC\_READ OR GENERIC\_WRITE’로 설정한다. dwShareMode 파라미터에는 지정한 포트를 다른 어플리케이션과 공유할 것인지 여부를 정한다. 보통의 경우에는 0 으로 지정하여 다른 어플리케이션이 접근할 수 없도록 한다.

lpSecurityAttributes 파라미터는 핸들에 대한 보안 레벨을 지정하는 구조체의 포인터를 넘

겨주게 되는데, 보통의 경우 nil 로 설정한다. dwCreationDisposition 파라미터는 윈도우가 파일을 열거나 생성하는 방법을 지정하는데, 시리얼 통신의 경우에는 파일이 언제나 지정되어 있고, 존재하는 상태이므로 'OPEN\_EXISTING'으로 설정된다.

dwFlagsAndAttributes 파라미터는 시리얼 포트가 통신하는 방법을 지정할 수 있는데, 예를 들어 FILE\_FLAG\_OVERLAPPED 로 지정된 경우 시리얼 포트가 동시에 읽기, 쓰기가 가능한 비동기(asynchronous) 통신을 지원한다. 보통의 경우에는 0 으로 지정해서 동기(synchronous) 통신을 지정한다. 마지막으로 hTemplateFile 파라미터는 시리얼 통신과 아무런 관계가 없기 때문에 보통 0 으로 설정한다.

## 장치 제어 블록 (Device Control Block, DCB)

시리얼 포트를 제어하는데 가장 중요한 데이터 구조가 장치 제어 블록(DCB) 구조체이다. 이 구조체에는 시리얼 포트에 적용할 모든 설정 사항이 저장된다. Windows.pas 유닛에 선언되어 있는 DCB 구조체의 선언부는 다음과 같다.

type

```
TDCB = record
    DCBLength: DWORD;           //DCB 구조체의 크기
    BaudRate: DWORD;           //Baud rate
    Flags: LongInt;             //비트 플래그
    wReserved: Word;
    XONLim: Word;               //XON 스위치에 대한 바이트 한계
    XOFLim: Word;               //XOFF 스위치에 대한 바이트 한계
    ByteSize: Byte;             //바이트의 비트 수
    Parity: Byte;               //패리티 종류
    StopBits: Byte;             //스톱 비트
    XONChar: Char;              //XON 문자
    XOFFChar: Char;             //XOFF 문자
    ErrorChar: Char;            //패리티 에러 대체 문자
    EOFChar: Char;              //EOF 문자
    EvtChar: Char;              //이벤트 문자
    wReserved1: Word;
end;
```

DCB 파라미터를 얻어오고, 설정하는데 사용되는 함수가 GetCommState, SetCommState 함수이다. 이들 함수에 포트에 대한 핸들을 넘겨 주면 DCB 구조체의 주소를 얻을 수 있다.

이들 함수의 선언부는 다음과 같다.

```
function GetCommState(hFile: THandle; var lpDCB: TDCB): BOOL; stdcall;  
function SetCommState(hFile: THandle; const lpDCB: TDCB): BOOL; stdcall;
```

## GetCommTimeouts, SetCommTimeouts 함수

데이터를 얻어올 때 몇 가지 문제로 데이터가 전송되지 않을 수가 있다. 예를 들어, 불의의 사고로 선로가 끊어진 경우 시리얼 포트는 데이터를 전송받을 수가 없다. 이때 적절한 시간이 넘게 지나가면 더이상 데이터를 기다리지 않는다. 이러한 시간을 읽거나 설정할 때에 GetCommTimeouts, SetCommTimeouts 함수를 사용한다. 이 함수에 포트의 핸들과 TCommTimeouts 데이터 형의 레코드를 전달하면 제한 시간이 설정된다. 이들의 선언부는 다음과 같다.

```
function GetCommTimeouts(hFile: THandle; var lpCommTimeouts: TCommTimeouts):  
    BOOL; stdcall;  
function SetCommTimeouts(hFile: THandle; const lpCommTimeouts: TCommTimeouts):  
    BOOL; stdcall;
```

```
TCommTimeouts = record  
    ReadIntervalTimeout: DWORD;  
    ReadTotalTimeoutMultiplier: DWORD;  
    ReadTotalTimeoutConstant: DWORD;  
    WriteTotalTimeoutMultiplier: DWORD;  
    WriteTotalTimeoutConstant: DWORD;  
end;
```

보통의 경우에는 마이크로소프트의 디폴트 값을 그대로 사용한다.

## PurgeComm, CloseHandle, ClearCommError 함수

PurgeComm 함수를 이용하면 읽기, 쓰기를 진행하거나 취소할 수 있다. 또한 입력, 출력 버퍼를 비우게 할 수도 있다. 선언부는 다음과 같다.

```
function PurgeComm(hFile: THandle; dwFlags: DWORD): BOOL; stdcall;
```

첫번째 파라미터에는 포트의 핸들을 지정하면 되고, 두번째 파라미터에 플래그를 설정한다. 플래그로 사용할 수 있는 것으로 PURGE\_TXABORT, PURGE\_TXCLEAR, PURGE\_RXABORT, PURGE\_RXCLEAR 등이 있다. 여기서 ‘abort’는 진행 중인 작업을 즉시 중지하라는 의미이며, ‘clear’는 해당 버퍼를 비우라는 의미이다. TX 와 RX 는 각각 ‘transfer’, ‘receive’를 의미한다.

이렇게 사용한 포트를 닫을 때에는 CloseHandle 함수를 사용한다. 이 함수가 성공적으로 수행되면 True 가 반환된다. 선언부는 다음과 같다.

```
function CloseHandle(hFile: THandle): BOOL; stdcall;
```

ClearCommError 함수는 지정된 장치의 현재 상태를 검사해서 에러가 있으면 이를 리포트한다. 또한, 통신 에러가 발생하면 호출되어 장치의 에러 플래그를 지우고 데이터의 읽기, 쓰기 작업을 계속하게 한다. 이 함수의 lpStat 파라미터는 TCommStat 구조체의 포인터로, 이 구조체에 발생한 에러의 내용과 현재 장치의 상태에 대한 내용이 담기게 된다. 함수와 TCommStat 구조체의 선언부는 다음과 같다.

```
function ClearCommError(hFile: THandle; var lpErrors: DWORD; lpStat: PComStat):  
    BOOL; stdcall;  
TComStat = record  
    Flags: TCommStateFlags;  
    Reserved: array[0..2] of Byte;  
    cbInQue: DWORD;  
    cbOutQue: DWORD;  
end;
```

## 데이터 읽기와 쓰기 (ReadFile and WriteFile)

데이터를 읽고 쓰는데 가장 중요한 함수는 ReadFile, WriteFile 함수이다. 이 함수들은 시리얼 포트에서 실제 데이터를 읽고 쓸 때 사용된다. 선언부는 다음과 같다.

```
function WriteFile(hFile: THandle; const Buffer: nNumberOfBytesToWrite: DWORD;  
    var lpNumberOfBytesWritten: DWORD; lpOverlapped: POverlapped): BOOL; stdcall;  
function ReadFile(hFile: THandle; var Buffer: nNumberOfBytesToRead: DWORD;  
    var lpNumberOfBytesRead: DWORD; lpOverlapped: POverlapped): BOOL; stdcall;
```

기본적으로 파라미터가 의미하는 부분은 거의 비슷하다. hFile 파라미터는 사용할 포트의 핸들이고, Buffer 는 데이터를 보내거나 읽어 와야 되는 버퍼의 포인터를 가리킨다. nNumberOfBytesWritten, nNumberOfBytesRead 파라미터는 실제로 포트에 전송되거나 받게 되는 데이터의 바이트 수를 나타낸다. 그리고, lpNumberOfBytesWritten, lpNumberOfBytesRead 파라미터는 실제로 읽고 쓴 바이트 수를 나타낸다. 마지막으로 lpOverlapped 파라미터는 시리얼 포트를 비동기로 사용할 때 사용되는 TOverlapped 구조체에 대한 포인터이다.

## TSerialPort 컴포넌트

그러면, 이러한 지식을 바탕으로 시리얼 통신을 쉽게 사용할 수 있는 컴포넌트를 하나 구현해 보자. 다음부터 설명하는 컴포넌트는 Jason “Wedge” Perry 가 공개한 TSerialPort 컴포넌트에 기초한 것임을 미리 밝혀 둔다.

그러면 이 컴포넌트에서 사용할 프로퍼티를 미리 정의하도록 한다. 사실 시리얼 포트에 대한 프로퍼티로 정의할 만한 것들은 대부분 DCB 구조체의 내용이다. 비교적 중요한 프로퍼티를 나열하면 아래와 같다.

- COM 포트 (COM1 ~ COM8)
- Baud Rate (110 ~ 256,000)
- 패리티 검사의 종류 (None, Even, Odd, Mark, Space)
- 스톱 비트 (1, 1.5, 2)
- 데이터 비트 (4, 5, 6, 7, 8)
- XON 문자 (디폴트 11h)
- XOFF 문자 (디폴트 13h)
- XON Limit (1024k)
- XOFF Limit (2048k)
- 에러 문자 (0)
- 흐름 제어 종류 (RTS/CTS, XON/XOFF, DSR/DTR)

이들을 각각의 프로퍼티로 제어하려면 Set 메소드를 정의해야 한다. 이를 바탕으로 컴포넌트를 다음과 같이 선언한다.

```
unit Serial;
```

```
interface
```

```
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

type

```
TCommPort = (cpCOM1, cpCOM2, cpCOM3, cpCOM4, cpCOM5, cpCOM6,  
    cpCOM7, cpCOM8);  
TBaudRate = (br110, br300, br 600, br1200, br2400, br4800, br9600, br14400, br19200,  
    br38400, br56000, br128000, br256000);  
TParityType = (pcNone, pcEven, pcOdd, pcMark, pcSpace);  
TStopBits = (sbOne, sbOnePtFive, sbTwo);  
TDataBits = (db4, db5, db6, db7, db8);  
TFlowControl = (fcNone, fcXON_XOFF, fcRTS_CTS, fsDSR_DTR);  
TNotifyTXEvent = procedure (Sender: TObject; Data: String) of Object;  
TNotifyRXEvent = procedure (Sender: TObject; Data: String) of Object;
```

const

```
dflt_CommPort = cpCOM2;  
dflt_BaudRate = br14400;  
dflt_ParityType = pcNone;  
dflt_ParityErrorChecking = False;  
dflt_ParityErrorChar = 0;  
dflt_ParityErrorReplacement = False;  
dflt_StopBits = sbNone;  
dflt_DataBits = db8;  
dflt_XONChar = $11;  
dflt_XOFFChar = $13;  
dflt_XONLim = 1024;  
dflt_XOFFLim = 2048;  
dflt_ErrorChar = 0;  
dflt_FlowControl = fcNone;  
dflt_StripNullChars = False;  
dflt_EOFChar = 0;
```

TSerialPort = class(TComponent)

private

```
hCommPort: THandle;  
FCommPort: TCommPort;  
FBaudRate: TBaudRate;  
FParityType: TParityType;
```

FParityErrorChecking: Boolean;  
FParityErrorChar: Byte;  
FParityErrorReplacement: Boolean;  
FStopBits: TStopBits;  
FDataBits: TDataBits;  
FXONChar: Byte;  
XOFFChar: Byte;  
FXONLim: Word;  
XOFFLim: Word;  
FErrorChar: Byte;  
FFlowControl: TFlowControl;  
FStripNullChars: Boolean;  
FEOFChar: Byte;  
FOnTransmit: TNotifyTXEvent;  
FOnReceive: TNotifyRXEvent;  
FAfterTransmit: TNotifyTXEvent;  
FAfterReceive: TNotifyRXEvent;  
FRXBytes: DWORD;  
FTXBytes: DWORD;  
ReadBuffer: String;  
procedure SetCommPort(Value: TCommPort);  
procedure SetBaudRate(Value: TBaudRate);  
procedure SetParityType(Value: TParityType);  
procedure SetParityErrorChecking(Value: Boolean);  
procedure SetParityErrorChar(Value: Byte);  
procedure SetParityErrorReplacement(Value: Boolean);  
procedure SetStopBits(Value: TStopBits);  
procedure SetDataBits(Value: TDataBits);  
procedure SetXONChar(Value: Byte);  
procedure SetXOFFChar(Value: Byte);  
procedure SetXONLim(Value: Word);  
procedure SetXOFFLim(Value: Word);  
procedure SetErrorChar(Value: Byte);  
procedure SetFlowControl(Value: TFlowControl);  
procedure SetStripNullChars(Value: Boolean);  
procedure SetEOFChar(Value: Value);

```

    procedure Initialize_DCB;
protected
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function OpenPort(MyCommPort: TCommport): Boolean;
    function ClosePort: Boolean;
    procedure SendData(Data: PChar; Size: DWORD);
    function GetData: String;
    function PortIsOpen: Boolean;
    procedure FlushTX;
    procedure FlushRX;
published
    property CommPort: TCommPort read FCommPort write SetCommPort default dfilt_CommPort;
    property BaudRate: TBaudRate read FBaudRate write Set FBaudRate default dfilt_BaudRate;
    property ParityType: TParityType read FParityType write SetParityType default dfilt_ParityType;
    property ParityErrorChecking: Boolean read FParityErrorChecking write SetParityErrorChecking
        default dfilt_ParityErrorChecking;
    property ParityErrorChar: Byte read FParityErrorChar write SetParityErrorChar
        default dfilt_ParityErrorChar;
    property StopBits: TStopBits read FStopBits write SetStopBits default dfilt_StopBits;
    property DataBits: TDataBits read FDataBits write SetDataBits default dfilt_DataBits;
    property XONChar: Byte read FXONChar write SetXONChar default dfilt_XONChar;
    property XOFFChar: Byte read FXOFFChar write SetXOFFChar default dfilt_XOFFChar;
    property XONLim: Word read FXONLim write SetXONLim default dfilt_XONLim;
    property XOFFLim: Word read FXOFFLim write SetXOFFLim default dfilt_XOFFLim;
    property ErrorChar: Byte read FErrorChar write SetErrorChar default dfilt_ErrorChar;
    property FlowControl: TFlowControl read FFlowControl write SetFlowControl
        default dfilt_FlowControl;
    property StripNullChars: Boolean read FStripNullChars write SetStripNullChars
        default dfilt_StripNullChars;
    property EOFChar: Byte read FEOFChar write SetEOFChar default dfilt_EOFChar;
    property OnTransmit: TNotifyTXEvent read FOnTransmit write FOnTransmit;
    property OnReceive: TNotifyRXEvent read FOnReceive write FOnReceive;
    property AfterTransmit: TNotifyTXEvent read FAfterTransmit write FAfterTransmit;
    property AfterReceive: TNotifyRXEvent read FAfterReceive write FAfterReceive;

```



```
end;
```

```
procedure Register;
```

이들 각각의 프로퍼티에 대해 앞에서 보다시피 dflt\_ 로 시작하는 디폴트 상수값을 지정하고, Set 메소드를 정의하였다. 이들 Set 메소드의 구현 방법은 기본적으로 다음과 같은 형태를 가진다.

```
procedure TSerialPort.SetProperty(Value: TDefinedType);
```

```
begin
```

```
  if Value <> FProperty then
```

```
  begin
```

```
    FProperty := Value;
```

```
    Initialize_DCB;
```

```
  end;
```

```
end;
```

프로퍼티를 설정하는 방법은 비교적 쉽게 이해할 수 있을 것이다. 이번에는 이벤트를 정의할 차례이다. TSerialPort 클래스에서는 OnTransmit, OnReceive, AfterTransmit, AfterRecieve 라는 4 개의 이벤트를 제공한다. 각각의 이벤트는 Sender 와 Data 라는 파라미터를 가진다.

이제부터 앞에서 선언한 컴포넌트의 메소드를 구현해보자. 먼저 Create, Destroy, PortIsOpen 메소드를 다음과 같이 구현한다.

```
constructor TSerialPort.Create(AOwner: TComponent);
```

```
begin
```

```
  inherited Create(AOwner);
```

```
  hCommPort := INVALID_HANDLE_VALUE;
```

```
  FCommPort := dflt_CommPort;
```

```
  FBaudRate := dflt_BaudRate;
```

```
  FParityCheck := dflt_ParityCheck;
```

```
  FStopBits := dflt_StopBits;
```

```
  FDataBits := dflt_DataBits;
```

```
  FXONChar := dflt_XONChar;
```

```
  FXOFFChar := dflt_XOFFChar;
```

```
  FXONLim := dflt_XONLim;
```

```

FXOFFLim := dflt_XOFFLim;
FErrorChar := dflt_ErrorChar;
FFlowControl := dflt_FlowControl;
FOnTransmit := nil;
FOnReceive := nil;
end;

destructor TSerialPort.Destroy;
begin
    ClosePort;
    inherited Destroy;
end;

function TSerialPort.PortIsOpen: Boolean;
begin
    Result := hCommPort <> INVALID_HANDLE_VALUE;
end;

```

Create 메소드에서는 데이터 필드의 초기값을 설정하게 된다. 여기서 눈여겨 볼 것은 hCommPort 핸들을 INVALID\_HANDLE\_VALUE 로 설정한 부분이다. 이는 시리얼 포트가 성공적으로 열렸는지 테스트하고, 포트에 특정 작업이 이루어지는 것을 막는 등의 역할을 하게 된다. 이와 연관되어서 PortIsOpen 메소드가 사용된다.

Destroy 메소드에서는 열린 포트를 닫는 ClosePort 메소드를 호출한다.

그러면 실제로 포트를 열고, 닫는 OpenPort 와 ClosePort 메소드를 구현하도록 하자. OpenPort 메소드는 앞에서 설명한 CreateFile API 함수를 이용해서 포트에 대한 핸들을 얻어오게 되며, 성공적으로 이 작업이 이루어지면 True 를 반환한다. 이때 포트를 열기 위해서는 이미 열려있는 포트를 닫아야 하므로 ClosePort 메소드를 먼저 호출한다. 그리고, Initialize\_DCB 메소드를 호출해서 포트를 초기화 한다.

ClosePort 메소드는 CloseHandle API 함수를 호출해서 포트에 대한 핸들을 해제하게 되는 데, 이때 PurgeComm API 함수를 사용해서 버퍼의 값을 비우게 되는 FlushTX, FlushRX 메소드를 먼저 호출한다. 그리고, 포트에 대한 핸들에 아무것도 지정되지 않았다는 것을 나타내는 INVALID\_HANDLE\_VALUE 값을 지정한다.

```

function TSerialPort.OpenPort(MyCommPort: TCommPort): Boolean;
var
    MyPort: PChar;

```

```

begin
    ClosePort:
    MyPort := PChar('COM' + IntToStr(Ord(MyCommPort) + 1));
    hCommPort := CreateFile(MyPort, GENERIC_READ or GENERIC_WRITE, 0, nil,
        OPEN_EXISTING, 0, 0);
    Initialize_DCB;
    if hCommPort <> INVALID_HANDLE_VALUE then
    begin
        Result := True;
        FCommPort := MyCommPort;
    end
    else Result := False;
end;

```

```

function TSerialPort.ClosePort: Boolean;
begin
    FlushTX;
    FlushRX;
    Result := CloseHandle(hCommPort);
    hCommPort := INVALID_HANDLE_VALUE;
end;

```

```

function TSerialPort.FlushRX;
begin
    if hCommPort = INVALID_HANDLE_VALUE then Exit;
    PurgeComm(hCommPort, PURGE_RXABORT or PURGE_RXCLEAR);
    ReadBuffer := '';
end;

```

```

function TSerialPort.FlushTX;
begin
    if hCommPort = INVALID_HANDLE_VALUE then Exit;
    PurgeComm(hCommPort, PURGE_TXABORT or PURGE_TXCLEAR);
end;

```

포트를 초기화하는 부분은 Initialize\_DCB 메소드에 의해서 구현된다. 이 메소드는 프로퍼

티가 바뀔 때마다 이 변화를 DCB 구조체에 반영하는 역할을 한다. 실제 구현 방법은 다음 코드를 보면 쉽게 이해할 수 있을 것이다.

```
procedure TSerialPort.Initialize_DCB;
var
    MyDCB: TDCB;
begin
    if hCommPort = INVALID_HANDLE_VALUE then Exit;
    GetCommState(hCommPort, MyDCB);
    case FBaudRate of
        br110: MyDCB.BaudRate := 110;
        br300: MyDCB.BaudRate := 300;
        br600: MyDCB.BaudRate := 600;
        br1200: MyDCB.BaudRate := 1200;
        br2400: MyDCB.BaudRate := 2400;
        br4800: MyDCB.BaudRate := 4800;
        br9600: MyDCB.BaudRate := 9600;
        br14400: MyDCB.BaudRate := 14400;
        br19200: MyDCB.BaudRate := 19200;
        br38400: MyDCB.BaudRate := 38400;
        br56000: MyDCB.BaudRate := 56000;
        br128000: MyDCB.BaudRate := 128000;
        br256000: MyDCB.BaudRate := 256000;
    end;

    case FParityType of
        pcNone: MyDCB.Parity := NOPARITY;
        pcEven: MyDCB.Parity := EVENPARITY;
        pcOdd: MyDCB.Parity := ODDPARITY;
        pcMark: MyDCB.Parity := MARKPARITY;
        pcSpace: MyDCB.Parity := SPACEPARITY;
    end;

    if FParityErrorChecking then inc(MyDCB.Flags, $0002);
    if FParityErrorReplcement then inc(MyDCB.Flags, $0021);
    MyDCB.ErrorChar := Char(FErrorChar);
```

```

case FStopBits of
    sbOne: MyDCB.StopBits := ONESTOPBIT;
    sbOnePtFive: MyDCB.StopBits := ONE5STOPBITS;
    sbTwo: MyDCB.StopBits := TWOSTOPBITS;
end;

case FDataBits of
    db4: MyDCB.ByteSize := 4;
    db5: MyDCB.ByteSize := 5;
    db6: MyDCB.ByteSize := 6;
    db7: MyDCB.ByteSize := 7;
    db8: MyDCB.ByteSize := 8;
end;

case FFlowControl of
    fcXON_XOFF: MyDCB.Flags := MyDCB.Flags or $0020 or $0018;
    fcRTS_CTS: MyDCB.Flags := MyDCB.Flags or $0004 or
        $0024 * RTS_CONTROL_HANDSHAKE;
    fcDSR_DTR: MyDCB.Flags := MyDCB.Flags or $0008 or
        $0010 * DTR_CONTROL_HANDSHAKE;
end;

if FStripNullChars then inc(MyDCB.Flags, $0022);
MyDCB.XONChar := Char(FXONChar);
MyDCB.XOFFChar := Char(FXOFFChar);
MyDCB.XONLim := FXONLim;
MyDCB.XOFFLim := FXOFFLim;
if FEOFChar <> 0 then MyDCB.EOFChar := Char(EOFChar);
SetCommState(hCommPort, MyDCB);
end;

```

이 메소드를 구현하기 위해서 DCB 구조체에 비트 플래그를 사용하였다. 이러한 플래그에는 여러가지 종류가 있는데, 대부분은 흐름 제어와 패리티 검사에 사용된다. 사용 가능한 플래그를 나열하면 다음과 같다.

```

fParity = $0002;           //설정되면 패리티 검사를 한다.
fOutxCtsFlow = $0004;      //CTS 가 high 가 아니면 데이터가 전송되지 않음
fOutxDsrFlow = $0008;      //DSR 이 high 가 아니면 데이터가 전송되지 않음
fDtrControl = $0010;

        //DTR_CONTROL_ENABLE, DTR_CONTROL_DISABLE, DTR_CONTROL_HANDSHAKE
fDsrSensitivity = $0012;    //DSR 이 high 가 아니면 모든 바이트가 무시됨
fTxContinueOnXOff = $0014;

        //이 플래그가 설정되어 있으면 XON 문자가 전송되어도 데이터를 보내며,
        그렇지 않으면 XONLim 이 도달할 때까지 데이터를 보내지 않는다.
fOutX = $0018;             //전송시 XON/XOFF 흐름 제어를 사용
fInX = $0020;              //수신시 XON/XOFF 흐름 제어를 사용
fErrorChar := $0021;       //패리티 에러가 발생하면 이 문자로 교체
fNull = $0022;             //Null 문자를 잘라냄
fRtsControl = $0024;       //RTS 흐름 제어 사용

```

이제 실제로 데이터를 받고, 전송하는 GetData, SetData 메소드를 구현해 보자. SendData 가 보다 쉽게 구현이 가능하므로, 여기에 대해서 먼저 알아본다. 이 메소드가 호출되면 먼저 OnTransmit 이벤트 핸들러를 호출해서 데이터를 전송하기 전에 여러가지 조작을 할 수 있도록 허용한다. 마찬가지로 이 메소드의 마지막에는 AfterTransmit 이벤트 핸들러를 호출해서 데이터 전송 후에 여러가지 작업을 할 수 있도록 한다. 실제로 데이터를 전송하는 작업은 WriteFile API 함수를 사용한다. 파라미터로 포트에 대한 핸들과 전송할 데이터의 포인터, 그리고 전송할 데이터의 크기를 넘겨주면 실제로 전송된 데이터의 크기가 결과값으로 반환된다. 마지막 파라미터에는 전송하는 방법을 지정할 수 있는데, nil 로 설정해서 동시에 읽고, 쓰는 작업을 할 수 없도록 하였다. 이 값을 ‘overlapped’ 스타일의 비동기 통신 모드로 설정하면 동시에 읽고, 쓰는 작업을 할 수 있는데 이를 구현하기 위해서는 버퍼 처리, 메모리 재할당 기법 등의 고급스런 테크닉을 사용해야 한다.

```

procedure TSerialPort.SendData(Data: PChar; Size: DWORD);
var
    NumBytesWritten: DWORD;
begin
    if hCommPort = INVALID_HANDLE_VALUE then Exit;
    if Assigned(FOnTransmit) then FOnTransmit(Self, Data);
    WriteFile(hCommPort, Data^, Size, NumBytesWritten, nil);
    if Assigned(FAfterTransmit) then FAfterTransmit(Self, Data);
end;

```

GetData 메소드에서도 프로시저의 처음과 끝 부분에 OnReceive, AfterReceive 이벤트 핸들러를 호출한다. 그리고, 데이터를 읽어 오는 데에는 ReadFile API 함수를 사용하는데, 이 함수에서 읽어온 데이터를 저장할 버퍼와 데이터의 크기를 지정한다. 이때 데이터의 크기를 적절하게 지정하기 위해 ClearCommError API 함수를 이용해서 TComStat 형의 데이터를 읽어와서, 버퍼의 크기를 결정하는 과정을 거친다.

```
function TSerialPort.GetData: String;
var
    NumBytesRead: DWORD;
    BytesInQueue: LongInt;
    oStatus: TComStat;           //에러 코드를 담기 위해서 사용한다.
    dwErrorCode: DWORD;
begin
    if hCommPort = INVALID_HANDLE_VALUE then Exit;
    if Assigned(FOnReceive) then FOnReceive(Self, ReadBuffer);
    ClearCommError(hCommPort, dwErrorCode, @oStatus);
    BytesInQueue := oStatus.cbInQue;
    if BytesInQueue > 0 then
        begin
            SetLength(ReadBuffer, BytesInQueue + 1);
            ReadFile(hCommPort, PChar(ReadBuffer)^, BytesInQueue, NumBytesRead, nil);
            SetLength(ReadBuffer, StrLen(PChar(ReadBuffer)));
        end;
    if Assigned(FAfterReceive) then FAfterReceive(Self, ReadBuffer);
    Result := ReadBuffer;
end;
```

이렇게 해서 가장 기본적인 작동을 하는 시리얼 포트 컴포넌트를 하나 완성하였다. 이를 바탕으로 보다 기능을 강화해서 좋은 컴포넌트로 만들 수도 있는데, 예를 들어서 CRC 검사를 하고, 비동기 통신이 가능하며, 데이터를 읽고 쓸 때 멀티 쓰레딩 기법을 이용하는 컴포넌트 등을 생각할 수 있겠다.

## TSerialPort 컴포넌트의 활용

이 컴포넌트를 이용하는 방법을 알아보자. TSerialPort 컴포넌트의 기능과 간단한 예제 코

드를 나열하면 다음과 같다.

#### 1. 포트 열기:

```
SerialPort1.OpenPort(cpCOM2);  
or if SerialPort1.OpenPort(cpCOM2) then do something.
```

#### 2. 데이터 전송:

```
var  
    S: PChar;  
begin  
    S := PChar(Edit1.Text);  
    SerialPort1.SendData(S, SizeOf(s));  
    SerialPort1.SendData((chr(13)), 1);           //캐리지 리턴  
end;
```

#### 3. 데이터 읽기:

```
Memo1.Text := SerialPort1.GetData;
```

#### 4. 버퍼 비우기:

```
SerialPort1.FlushRX;  
SerialPort1.FlushTX;
```

#### 5. 포트가 열려 있는지 확인:

```
if SerialPort1.PortIsOpen then do something.
```

#### 6. 포트 닫기

```
SerialPort1.ClosePort;
```

### 정 리 (Summary)



이번 장에서는 직렬 포트를 이용한 통신을 하는 기본적인 방법에 대해서 알아보았다. 최근의 인터넷 환경의 발달로 모뎀과 직렬 포트를 직접 이용하는 사례가 많이 줄어들고 있는 것이 사실이다. 그렇지만, 직렬 포트는 새로운 기계를 제작하여 컴퓨터로 제어를 하거나, 그 밖에도 여러가지 아이디어를 가지고 가장 쉽게 컴퓨터와 연결할 수 있는 수단이다. 그러므로, 직렬 포트를 활용하는 방법을 알아두는 것은 의미가 있다.

# 네트워크 어플리케이션의 제작

## (Creating Network Application)

37 장에서는 소켓을 이용한 어플리케이션 제작 방법에 대해서 알아보았다. 이번 장에서는 텔파이 4 에서 제공되는 NetMaster 의 컴포넌트 들을 소개하고, 이를 이용하여 작성할 수 있는 네트워크 어플리케이션을 소개한다.

### NetMaster 컴포넌트

텔파이의 인터넷 컴포넌트는 2.01 버전의 NetManage 컴포넌트에서 시작되었다. 이 컴포넌트 들은 TCP/IP, SMTP, POP3, Telnet, FTP 등의 인터넷 서비스를 모두 지원했지만 그다지 높은 호응을 얻지는 못했는데, 그 이유는 텔파이의 native VCL 컴포넌트가 아니라 액티브 X 컴포넌트였기 때문이다. 즉, 액티브 X 컨트롤을 같이 배포하고 이를 등록하지 않으면 사용할 수 없는 단점이 있었다. 그렇기 때문에 많은 개발자들이 프리웨어로 배포된 Francois Piette 등의 인터넷 컴포넌트를 사용하였다.

텔파이 4 에서는 이러한 문제점을 시정하여 완전한 native VCL 컴포넌트로 변신한 NetMaster 컴포넌트 들을 만나볼 수 있다. NetMaster 컴포넌트에는 단순한 인터넷 표준 프로토콜을 지원하는 이외에 나름대로 커스터마이징이 가능한 윈시 컴포넌트를 다수 지원하고 있기 때문에 수월하게 네트워크 프로그래밍을 할 수 있다. 그렇지만, 아쉬운 점이 있다면 가장 흔히 사용되는 HTML 컴포넌트는 여전히 액티브 X 컨트롤을 사용한다는 점이다. 그래도 텔파이 3 에서의 HTML 컴포넌트가 버전 1.0 만 지원하여 무척 부족했지만, 텔파이 4 에서는 버전 2.x 를 지원하기 때문에 대부분의 HTML 페이지를 표시할 수 있다.

그러면 NetMaster 컴포넌트 들의 종류와 이들에 대해 개략적으로 알아보도록 하자.

#### ● TNMFTP 컴포넌트

TNMFTP 컴포넌트는 FTP 프로토콜을 이용하여 FTP 서버로 파일을 전송하거나, 파일을 다운로드 받기 위해서 사용된다. 이렇게 TNMFTP 컴포넌트를 이용하여 파일을 전송하기 위해서는 원격지 호스트와의 접속이 선행되어야 한다.

이를 위해서는 먼저 유효한 FTP 서버의 IP 주소를 Host 프로퍼티에 설정하고, Port 프로퍼티를 정확하게 설정해야 한다. 그리고 나서, 서버에서 허용하는 계정의 사용자 ID 와 패스워드를 UserID, Password 프로퍼티에 설정한다. 보통 대부분의 FTP 서버는 사용자 ID 로 Anonymous 와 패스워드로 E-mail 주소를 사용할 수 있도록 되어 있다. 이런 기본적인 프로퍼티 설정이 끝나면 Connect 메소드를 호출하여 서버에 접속한다.

이제 서버의 디렉토리 리스트를 얻어올 차례이다. 이를 위해서는 List 메소드를 호출하고, OnListItem 이벤트 핸들러에서 넘어오는 디렉토리 리스트의 아이템을 처리해 주어야 한다. 디렉토리를 변경하기 위해서는 ChangeDir 메소드를 사용한다.

원격지 호스트에 파일을 업로드하기 위해서는 업로드를 하기 위한 디렉토리에 먼저 위치한 뒤에 Upload 메소드를 호출하면 된다. 이 메소드에 로컬 컴퓨터의 파일의 이름과 원격지 호스트에 위치할 파일 이름을 파라미터로 지정할 수 있다.

파일을 다운로드할 때에는 먼저 List 메소드를 호출하여 다운로드할 파일을 찾을 수 있어야 한다. 그리고 나서 Download 메소드를 호출하는데, 여기에서 파라미터로 다운로드할 파일 이름과 로컬 드라이브에 저장할 파일의 경로와 이름을 지정한다.

디렉토리를 원격 호스트에 만들기 위해서는 MakeDirectory, 디렉토리를 지우기 위해서는 RemoveDir 메소드를 이용한다.

#### ● TNMHTTP 컴포넌트

TNMHTTP 컴포넌트는 인터넷을 통해 HTTP 전송을 수행한다. URL의 특성상 URL에 호스트와 포트의 내용을 포함하기 때문에 Host와 Port 프로퍼티는 설정할 필요가 없다.

문서를 Get하고자 할 때에는 단순히 Get 메소드를 호출하면 된다. 이때 불러올 문서의 URL을 지정하면 되는데, 이렇게 해서 불러온 데이터는 InputFileMode 프로퍼티의 내용에 따라 다르게 저장된다. 이 값이 True이면 문서의 body 부분은 Body 프로퍼티에 지정된 파일에 저장되고, header 부분은 Header 프로퍼티에 지정된 파일에 저장된다. 반면에 이 값이 False이면 Body와 Header 프로퍼티에 직접 저장된다.

데이터를 Post할 때에도 Post 메소드에 URL을 첫번째 파라미터로 지정하고, 두번째 파라미터인 PostData는 OutputFileMode 프로퍼티의 값에 따라 결정된다. OutputFileMode 프로퍼티의 값이 True이면 PostData 파라미터에는 특정 위치에 전달될 데이터가 저장된 파일이 지정된다. 반면 이 값이 False이면 전달된 데이터가 직접 PostData 파라미터에 지정된다. 이 메소드에 의해 전달되어 오는 문서를 저장하는 방법은 InputFileMode에 의해 결정되는데, 그 내용은 Get 메소드와 동일하다.

#### ● TNMNNTP 컴포넌트

TNMNNTP 컴포넌트는 인터넷 뉴스를 NNTP 프로토콜에 의해 인터넷 뉴스 서버에서 읽고, 메시지를 전달하기 위해 사용된다. TNMNNTP 컴포넌트의 핵심 함수를 이용할 때에는 먼저 뉴스 호스트에 접속해야 한다. 이를 위해서 Host 프로퍼티에 뉴스 서버의 주소를 설정하고, Connect 메소드를 이용하여 서버에 접속한다.

뉴스 그룹의 리스트를 얻기 위해서는 GetGroupList 메소드를 이용한다. 그리고, 기사를 읽어올 뉴스 그룹을 선택해야 하는데, SetGroup 메소드에 뉴스그룹의 이름을 파라미터로 넘

겨서 호출하면 된다. 인터넷 뉴스 기사를 읽을 때에는 일단 그룹을 선택한 뒤에 GetArticle 메소드를 호출하면 된다. 이렇게 하면 Body 프로퍼티와 Header 프로퍼티의 값을 이용해서 선택된 기사를 읽을 수 있게 된다.

새로운 뉴스 기사를 전송하기 위해서는 PostHeader, PostBody 프로퍼티에 메시지의 헤더와 내용을 채우고, PostArticle 메소드를 호출하면 된다.

- TNMPOP3 컴포넌트

TNMPOP3 컴포넌트는 인터넷 E-mail 을 POP3 서버로부터 가져오는 역할을 한다. 이를 위해서 먼저 Host 프로퍼티를 E-mail 서버의 주소로 설정하고, UserID 프로퍼티와 Password 프로퍼티에 사용자 이름과 패스워드를 지정하고 Connect 메소드를 호출하여 서버에 접속해야 한다.

인터넷 메일을 가져오기 위해서는 GetMailMessage 메소드를 호출하면 된다. 메일의 body의 파트는 MailMessage 프로퍼티에 저장된다.

- TNMSMTP 컴포넌트

TNMSMTP 컴포넌트는 SMTP 프로토콜을 이용하여 인터넷 메일 서버에 메일을 전송하는 역할을 한다. 다른 컴포넌트와 마찬가지로 Host 와 Post 프로퍼티를 설정하고, Connect 메소드를 호출하여 메일 서버와 접속하고, Disconnect 메소드를 호출하여 접속을 중단한다.

메일을 보내기 위해서는 PostMessage 프로퍼티에 전송할 데이터를 지정하고, SendMail 메소드를 호출하면 된다. 경우에 따라서는 사용자가 연결된 호스트에 있는지 찾아볼 필요가 있는데 그럴 때에는 Verify 메소드를 이용한다. 그리고, 메일링 리스트의 멤버는 ExpandList 메소드를 이용하여 결정하는데, OnMailListReturn 이벤트 핸들러를 작성한다.

- TNMFinger

TNMFinger 컴포넌트는 인터넷 Finger 서버로부터 사용자에게 대한 정보를 얻을 때 사용된다. 먼저 Finger 서버에 접속해야 하므로, Host 프로퍼티에 서버의 주소를 지정하고 Port 프로퍼티는 거의 대부분 79 번을 사용하므로 변경할 필요가 없다. 사용자에게 대한 정보는 FingerStr 프로퍼티에서 얻을 수 있다.

- TNMUDP

TNMUDP 컴포넌트는 UDP 프로토콜을 이용하여 데이터 그램 패킷을 전송하는데 사용한다. 이 컴포넌트 역시 마찬가지로 패킷을 전송하기 전에 원격 호스트와 포트를 알아서

RemoteHost 와 RemotePort 프로퍼티에 값을 대입해야 한다.

실제로 데이터를 전송하기 위해서는 SendBuffer 또는 SendStream 메소드를 이용하는데 SendBuffer 메소드는 원격 호스트에 문자의 배열 형태로 데이터를 전송하며, SendStream 메소드는 데이터의 스트림을 전달한다.

UDP 데이터를 받을 때에는 LocalPort 프로퍼티를 반드시 설정해야 한다. 이 프로퍼티는 반드시 디자인 타임에서 설정해야 하며, 런타임에서 변경할 수 없다. 읽어올 UDP 데이터가 있으면 OnDataAvailable 이벤트가 호출되며 이 이벤트에서 ReadBuffer 메소드를 이용하여 데이터를 버퍼로 읽어들이거나 ReadStream 메소드를 이용하여 스트림으로 데이터를 읽는다.

- TNMDayTime

TNMDayTime 컴포넌트는 인터넷 daytime 서버로부터 RFC 867 에 정의된 날짜와 시간에 대한 정보를 불러온다. 사용방법은 Host 프로퍼티에 서버의 주소를 지정하고, 호스트에 접속하기만 하면 DayTimeStr 프로퍼티에 지정된 호스트의 날짜와 시간이 전송되어 저장되므로 이를 이용하기만 하면 된다.

- TNMTime

TNMTime 컴포넌트는 인터넷 time 서버에서 RFC 868 에 정의된 날짜와 시간에 대한 정보를 불러온다. 사용방법은 Host 프로퍼티에 서버의 주소를 지정하고, 호스트에 접속하기만 하면 TimeStr 프로퍼티에 지정된 호스트의 날짜와 시간이 전송되어 저장되므로 이를 이용하기만 하면 된다.

- TNMEcho

TNMEcho 컴포넌트는 텍스트를 인터넷 에코 서버에 전송하고, 이 내용을 다시 받을 때 사용되는데, RFC 862 에 정의된 프로토콜을 사용한다. 이를 이용해서 네트워크의 무결성과 속도를 측정한다.

서버와의 접속을 위해 Host 프로퍼티와 Port 프로퍼티를 설정하는데, 보통 에코 서버는 7 번 포트를 이용한다. 텍스트를 전송할 때에는 Echo 메소드를 이용한다.

- TNMUUProcessor

TNMUUProcessor 컴포넌트는 인터넷을 통해 전송되는 파일을 인코딩하고 디코딩할 때 사용된다. 사용 방법은 InputFile 프로퍼티를 처리할 파일로 설정하고, Method 프로퍼티에서

인코딩, 디코딩할 방법을 지정하고, `OutputFile` 프로퍼티에 결과로 생성될 파일을 지정한다. 그리고 실제로 실행을 위해서 `Encode`, `Decode` 메소드를 호출하면 된다.

사용하는 인코딩 메소드로는 `MIME/Base 64` 를 지원하는 `uuMIME` 과 `UUEncoding/Decoding` 을 지원하는 `uuCode` 를 사용할 수 있다. 이 컴포넌트를 사용하는 예제를 다음장에서 볼 수 있을 것이다.

- `TNMURL`

`TNMURL` 컴포넌트는 HTTP 전송에 대한 문자열과 URL 포맷 사이를 인코드, 디코드할 때 사용된다. 사용방법은 `InputString` 프로퍼티에 문자열이나 URL 포맷의 값을 설정하면 `Encode` 프로퍼티에 URL 포맷으로 인코드된 문자열을 포함하게 되고, `Decode` 프로퍼티에는 디코드된 문자열이 포함된다. 이때 에러가 발생하면 `OnError` 이벤트가 호출된다.

- `TNMMsg`

`TNMMsg` 컴포넌트는 간단 메시지를 전송하기 위해서 사용하는데, `TNMMsgServ` 컴포넌트와 함께 사용하여야 한다. 메시지를 전송하기 전에 `Host` 프로퍼티와 `Port` 프로퍼티에 해당되는 메시지 서버의 IP 주소와 포트를 설정하고, `FromName` 프로퍼티에서 메시지를 전송한 사람에 대한 정보를 설정하고, `PostIt` 메소드를 호출하면 메시지가 전송된다.

- `TNMMsgServ`

`TNMMsgServ` 컴포넌트는 `TNMMsg` 컴포넌트에서 전송된 메시지를 받는 역할을 한다. 어플리케이션을 디자인할 때 `Port` 프로퍼티에는 메시지 서버가 사용할 포트를 지정하는데, 디폴트 값을 사용해도 무방하다. 메시지 서버에 전송된 메시지를 받아서 처리하기 위해서는 `OnMsg` 이벤트 핸들러를 작성하면 된다,

- `TNMStrm`

`TNMStrm` 컴포넌트는 스트림을 스트림 서버에 전송하기 위해서 `TNMStrmServ` 컴포넌트와 함께 사용된다. 먼저 `Host` 와 `Port` 프로퍼티를 서버에 맞도록 설정하고, `FromName` 프로퍼티에 스트림을 전송하는 사람의 정보를 지정하고, `PostIt` 메소드를 이용하여 스트림을 전송하면 된다.

- `TNMStrmServ`

TNMStrmServ 컴포넌트는 TNMStrm 컴포넌트에서 전송하는 스트림을 처리하는 역할을 한다. TNMMsgServ 컴포넌트와 마찬가지로 Port 프로퍼티를 설정하고, OnMsg 이벤트에서 전송된 스트림을 처리하면 된다.

- TPowersock

TPowersock 컴포넌트는 TCP 통신을 이용하여 다양한 인터넷 프로토콜을 이용할 수 있도록 제공하는 기초 클래스이다. 그러므로 이 컴포넌트는 상속을 받아서 새로이 구성해야 하는 것으로, Connect 메소드를 호출하면 Host 와 Port 프로퍼티에 지정한 원격 호스트의 서비스에 접속하게 된다. 원격 호스트에서 받아온 데이터는 Read, ReadLn, CaptureFile, CaptureStream, CaptureString 메소드를 이용하여 읽을 수 있다. 그리고, 데이터를 원격 호스트에 전송할 때에는 Write, WriteLn, SendFile, SendStream 메소드를 이용한다. 그리고, Transaction 메소드는 원격 호스트에 데이터를 전송하고 서버로부터 응답을 받는다.

- TNMGeneralServ

TNMGeneralServer 컴포넌트는 다중 스레드를 지원하는 인터넷 서버를 개발할 때 RFC 표준을 지원하는 서버나 사용자 정의 서버를 개발할 때 사용되는 기초 클래스이다.

TNMGeneralServer 클래스를 상속하는 서버 클래스를 제작하기 위해서는 Server 메소드를 오버라이드해야 하는데, 여기에서 서버가 클라이언트 연결에 어떻게 반응할 것인지를 결정한다. 클라이언트와의 상호 작용은 TPowersock 의 read, write 메소드를 이용한다.

주의할 점은 Port 프로퍼티를 디자인 타임에서 설정해야 한다는 것이다.

## 메일 클라이언트 어플리케이션의 제작

그러면, 표준 인터넷 프로토콜을 지원하는 컴포넌트 중에서 POP3 와 SMTP 프로토콜을 이용한 메일 클라이언트를 제작해보도록 하자.

먼저 폼을 디자인해야 하는데, 나름대로 성의를 가지고 여러 개의 패널 객체와 Align 프로퍼티를 적절히 설정해서 깔끔한 인터페이스를 구성해보도록 하자. POP3 와 SMTP 클라이언트를 하나의 폼으로 구성하되, 메일을 받는 인터페이스와 보내는 인터페이스는 TPageControl 컴포넌트를 이용하여 다른 페이지로 구성한다.

메일을 보낼 때에는 파일을 Attach 하여 보낼 수 있도록 TOpenDialog 컴포넌트를 추가한다. 그리고, 실제 프로토콜을 관리하는 TNMPOP3, TNMSMTP 컴포넌트를 폼에 추가하도록 하자.

필자가 구성한 폼의 디자인은 다음과 같다. 이와 비슷하게 필요한 형태로 폼을 디자인하면 될 것이다.

‘접 속’ 버튼을 클릭하면 POP3 와 SMTP 라벨 옆에 있는 에디트 박스의 내용을 바탕으로 서버에 접속한다. POP3 의 경우에는 사용자 ID 와 패스워드를 이용하여 메일 서버에 접속하는 과정이 필요하다. 이때 패스워드는 입력할 때 입력하는 문자가 보이지 않거나, 특수 문자로 처리하는데 대개의 경우 ‘\*’ 문자를 이용한다. 이럴 때에는 TMaskEdit 컴포넌트를 이용하는 것이 좋다. MaskEdit1 객체의 PasswordChar 프로퍼티를 ‘\*’로 설정한다. 이 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하여, 메일 서버에 접속하도록 한다.

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

```
begin
```

```
    NMPOP31.AttachFilePath := '.';
```

```
    NMPOP31.DeleteOnRead := FALSE;
```

```
    NMPOP31.ReportLevel := Status_Basic;
```

```
    NMPOP31.TimeOut := 20000;
```

```
    NMPOP31.Host := Edit1.Text;
```

```
    NMPOP31.UserID := Edit3.Text;
```

```
    NMPOP31.Password := MaskEdit1.Text;
```

```
    NMPOP31.Connect;
```

```
    NMSMTP1.TimeOut := 20000;
```

```
    NMSMTP1.ReportLevel := Status_Basic;
```

```
    NMSMTP1.Host := Edit2.Text;
```



```

NMSMTP1.UserID := Edit5.Text;
NMSMTP1.Connect;
end;

```

접속을 하기 위해서는 TNMPOP3, TNMSMTP1 컴포넌트 모두 Connect 메소드를 호출하면 된다. 이때 Host 프로퍼티에 메일 서버에 IP 주소를 대입하고, TimeOut 프로퍼티는 접속을 시도할 시간을 msec 단위로 설정한다. ReportLevel 프로퍼티는 OnStatus 이벤트에서 상태 메시지를 어떻게 표시할 것인지를 결정하는 프로퍼티인데 보통은 Status\_Basic 으로 설정한다. POP3 컴포넌트에서의 DeleteOnRead 프로퍼티는 메시지를 읽을 경우에 메일 서버에서 읽은 메시지를 제거할 것인지 여부를 결정한다. 이 값을 True 로 할 경우에는 메시지를 클라이언트로 다운로드하면 서버에서 메시지가 삭제되며, False 로 설정할 경우에는 메일 클라이언트에서 읽더라도 서버에 메시지가 그대로 남게 된다. 이 경우에는 메시지를 직접 삭제할 수 있도록 해야 한다.

마찬가지로 ‘접속해제’ 버튼을 클릭하면 Disconnect 메소드를 호출하면 된다.

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    NMPOP31.Disconnect;
    NMSMTP1.Disconnect;
end;

```

POP3 페이지에서 ‘List’ 버튼을 클릭하면 POP3 메일 서버에 도착한 메시지를 나열하게 되는데, 이렇게 나열되는 메시지는 OnList 이벤트 핸들러에서 처리할 수 있다. ‘Clear’ 버튼을 클릭하면 메모 컴포넌트의 내용을 지우도록 한다.

```

procedure TForm1.Button3Click(Sender: TObject);
begin
    NMPOP31.List;
end;

```

```

procedure TForm1.Button4Click(Sender: TObject);
begin
    Memo3.Lines.Clear;
end;

```

OnList 이벤트 핸들러는 다음과 같이 작성하여, 전달된 메시지의 번호와 크기를 메모 컴포

년트에 나열하도록 한다.

```
procedure TForm1.NMPOP31List(Msg, Size: Integer);
begin
    Memo3.Lines.Add(IntToStr(Msg) + ' / ' + IntToStr(Size));
end;
```

이렇게 메모 컴포넌트에 나열된 메일 서버의 메시지 중에서 읽어올 메시지 번호를 에디트 박스에 지정하고, ‘Get Message’ 버튼을 클릭하면 해당되는 메시지를 읽어와서 header 는 우측의 위쪽 메모 컴포넌트에, body 는 아래쪽 메모 컴포넌트에 표시한다. ‘GetMessage’ 버튼의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button5Click(Sender: TObject);
begin
    NMPOP31.GetMailMessage(StrToInt(Edit4.Text));
    Memo1.Lines.Assign(NMPOP31.MailMessage.Head);
    Memo2.Lines.Assign(NMPOP31.MailMessage.Body);
    if NMPOP31.MailMessage.Attachments.Text <> '' then
        ShowMessage('Attachments: ' + #10#13 + NMPOP31.MailMessage.Attachments.Text);
end;
```

즉, 메시지를 읽어올 때에는 GetMailMessage 메소드에서 파라미터에 넘겨진 번호의 메시지를 읽어오게 된다. 이렇게 읽어온 메시지는 MailMessage 프로퍼티에 저장되는데, 각각 header 와 body 부분으로 나눌 수 있다. 이들은 모두 TStrings 데이터 형이기 때문에 메모 컴포넌트에 직접 대입할 수 있다. MailMessage 의 Attachment 프로퍼티에는 attach 된 파일을 나타내는데, 파일의 이름을 Text 프로퍼티를 이용하여 얻을 수 있다.

이것으로 버튼의 이벤트 핸들러는 모두 작성하였다. 이제는 TNMPOP3 컴포넌트의 이벤트 핸들러를 작성하도록 하자. 대개의 경우 TStatusBar 컴포넌트에 메일이 전달되는 상황에 대한 정보를 표시하는 정도로 작성한다.

```
procedure TForm1.NMPOP31Connect(Sender: TObject);
begin
    StatusBar1.SimpleText := 'POP3 Server Connected !';
end;

procedure TForm1.NMPOP31Disconnect(Sender: TObject);
```

```

begin
    if StatusBar1 <> nil then
        StatusBar1.SimpleText := 'POP3 Server Disconnected !';
    end;

procedure TForm1.NMPOP31Status(Sender: TComponent; Status: String);
begin
    if StatusBar1 <> nil then
        StatusBar1.SimpleText := 'POP3: ' + Status;
    end;

procedure TForm1.NMPOP31ConnectionFailed(Sender: TObject);
begin
    ShowMessage('POP3 Server Connection Failed');
end;

procedure TForm1.NMPOP31Failure(Sender: TObject);
begin
    ShowMessage('작업이 실패했습니다.');
```

```
procedure TForm1.NMPOP31RetrieveStart(Sender: TObject);
```

```
begin
```

```
    Form1.Cursor := crHourGlass;
```

```
    StatusBar1.SimpleText := 'Retrieval Start';
```

```
end;
```

```
procedure TForm1.NMPOP31RetrieveEnd(Sender: TObject);
```

```
begin
```

```
    Form1.Cursor := crDefault;
```

```
    StatusBar1.SimpleText := 'Retrieval End';
```

```
end;
```

이것으로 POP3 에 대한 부분은 모두 완성되었다.

이번에는 SMTP 클라이언트 기능을 추가할 차례이다. 인터페이스 디자인은 다음과 같다.

SMTP 클라이언트에서는 'Add', 'Remove' 버튼을 클릭하여 attach 될 파일을 파일열기 대화상자를 이용하여 리스트 박스에 추가하도록 'Add', 'Remove' 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button6Click(Sender: TObject);
```

```

begin
    If OpenFileDialog1.Execute then
        ListBox1.Items.Add(OpenDialog1.FileName);
    end;

    procedure TForm1.Button7Click(Sender: TObject);
    begin
        ListBox1.Items.Delete(ListBox1.ItemIndex);
    end;

```

이어서 ‘Send’ 버튼을 클릭하면 이들과 함께 메모 컴포넌트의 내용을 To, CC, BCC 로 지정된 주소에 전송한다. 이때 보내는 사람에 대한 정보를 인터넷 메일 주소와 실제 이름과 함께 전송한다. ‘Send’ 버튼의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```

procedure TForm1.Button8Click(Sender: TObject);
begin
    NMSMTP1.PostMessage.FromAddress := Edit7.Text;
    NMSMTP1.PostMessage.FromName := Edit6.Text;
    NMSMTP1.PostMessage.Subject := Edit11.Text;
    NMSMTP1.PostMessage.ToAddress.Add(Edit8.Text);
    NMSMTP1.PostMessage.ToBlindCarbonCopy.Add(Edit10.Text);
    NMSMTP1.PostMessage.ToCarbonCopy.Add(Edit9.Text);
    NMSMTP1.PostMessage.Attachments.AddStrings(Listbox1.Items);
    NMSMTP1.PostMessage.Body.Assign(Memo4.Lines);
    NMSMTP1.SendMail;
end;

```

POP3 와 마찬가지로 SMTP 컴포넌트에서도 여러가지 이벤트에 대한 상황을 표시할 수 있도록 다음과 같이 이벤트 핸들러들을 작성한다.

```

procedure TForm1.NMSMTP1Connect(Sender: TObject);
begin
    StatusBar1.SimpleText := 'SMTP Server Connected !';
end;

procedure TForm1.NMSMTP1Disconnect(Sender: TObject);

```

```

begin
    If StatusBar1 <> nil then
        StatusBar1.SimpleText := 'SMTP Server Disconnected';
    end;

procedure TForm1.NMSMTP1Status(Sender: TComponent; Status: String);
begin
    if StatusBar1 <> nil then
        StatusBar1.SimpleText := 'SMTP: ' + Status;
    end;

procedure TForm1.NMSMTP1EncodeStart(Filename: String);
begin
    StatusBar1.SimpleText := 'Encoding ' + Filename;
end;

procedure TForm1.NMSMTP1EncodeEnd(Filename: String);
begin
    StatusBar1.SimpleText := 'Finished Encoding ' + Filename;
end;

procedure TForm1.NMSMTP1ConnectionFailed(Sender: TObject);
begin
    ShowMessage('SMTP Server Connection Failed');
end;

procedure TForm1.NMSMTP1Failure(Sender: TObject);
begin
    StatusBar1.SimpleText := 'SMTP Operation Failed';
end;

procedure TForm1.NMSMTP1HostResolved(Sender: TComponent);
begin
    StatusBar1.SimpleText := 'SMTP Host Resolved';
end;

```

```

procedure TForm1.NMSMTP1PacketSent(Sender: TObject);
begin
    StatusBar1.SimpleText := IntToStr(NMSMTP1.BytesSent) + ' Bytes of ' +
        IntToStr(NMSMTP1.BytesTotal) + ' Sent';
end;

procedure TForm1.NMSMTP1RecipientNotFound(Recipient: String);
begin
    ShowMessage('Recipient "' + Recipient + '" not found !');
end;

procedure TForm1.NMSMTP1SendStart(Sender: TObject);
begin
    StatusBar1.SimpleText := 'Sending Message ...';
end;

procedure TForm1.NMSMTP1Success(Sender: TObject);
begin
    StatusBar1.SimpleText := 'Operation Successful !';
end;

```

이것으로 그럴 듯한 인터넷 메일 클라이언트가 작성되었다. 다음 그림들은 필자가 메일 클라이언트를 사용하여 메일을 받고, 메시지를 작성하여 하이텔의 필자 ID 로 전송한 뒤에 이를 하이텔에 접속하여 메일이 도착했는지 확인하는 그림이다.

Form1

POP3 서버:  SMTP 서버:

POP3 (메일 받기) | SMTP (메일 보내기) |

사용자 ID:   
 비밀번호:

메시지 번호/크기

1 / 9637942	Received: from sass165.sandia.gov (mailgate.sandia.gov [132.175.109.11]) by member.medikorea.net (8.6.12h2/8.6.12) with ESMTP id CAA04694 for <alphonse@medikorea.net>; Sat, 29 Aug 1998 02:23:11 +0900 Received: by sass165.sandia.gov (8.9.1a/8.9.1a) id KAA10531  I'm only just making the move from 4.0 to 4.1 so I don't know where this crept in, but the following parse fine in 4.0. I hope the solution (on either my end or Jess's) is simple, because I have a lot of these. and life in 4.0 finally became difficult enough for me to make the leap. so I'd rather not go back!!  --Sidney Bailin  (deffunction extract-pegs
2 / 4893	
3 / 1788	

메시지 번호:

Retrieval End

Form1

POP3:  SMTP:

POP3 (메일 받기) | SMTP (메일 보내기) |

사용자 ID:   
 이름:   
 From:   
 To:   
 CC:   
 BCC:

Subject:

Attachment:

잘되어야 될텐데...

Operation Successful !



```
HiTEL
RMAIL                                편지 읽기                                #1/1

제  목: 연습입니다 !                  형태:TXT                  크기: 270B                  1/ 1
보낸이:인터넷 (alphonse) 98/09/02 19:56 종류:수신

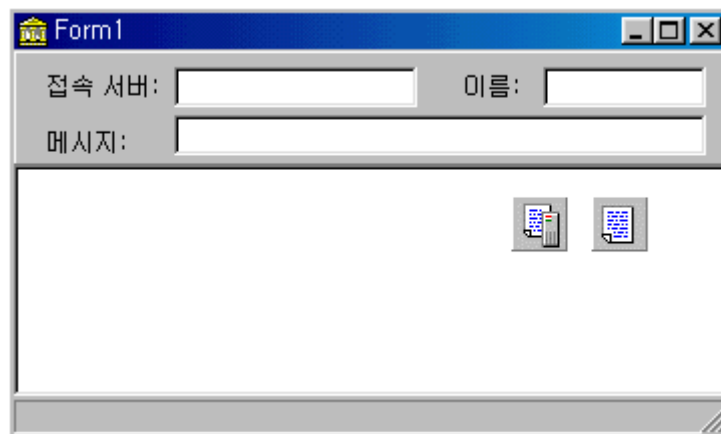
From:alphonse@medikorea.net
Date: Wed, 2 Sep 1998 20:06:54 +0900
Subject: 연습입니다 !
Content-Type: text/plain; charset=us-ascii
HiTEL:ttolttol
/-----/

잘되어야 될텐데...
```

## 메시지 클라이언트/서버 어플리케이션의 제작

이번에는 NetMaster 의 메시지 클라이언트와 서버 컴포넌트를 이용하여 간단한 메시지를 주고받을 수 있는 메시지 클라이언트/서버 어플리케이션을 만들어 보자. 컴포넌트의 사용 방법을 익히고자 하는 것이 목적이므로 1:1 채팅 어플리케이션과 마찬가지로 하나의 어플리케이션이 서버이면서 동시에 클라이언트 역할을 하도록 만들도록 한다.

먼저 폼에 패널을 하나 얹고 패널 위에 메시지 서버의 IP 주소를 지정할 에디트 박스와 보내는 사람의 이름을 설정할 에디트 박스, 그리고 실제 메시지를 입력할 에디트 박스를 추가한다. 그리고, 메모 컴포넌트를 폼에 추가하여 전달되는 메시지를 표시하도록 한다. 그리고 다음과 같이 TStatusBar 컴포넌트를 추가하여 인터넷 메일 클라이언트에서와 마찬가지로 메시지 컴포넌트의 상태를 표시할 수 있도록 한다.



소켓 프로그래밍을 이용한 채팅 프로그램보다 사용방법은 더 간단하다. 다음과 같이 서버의 IP 주소와 보내는 사람의 이름을 각각 Host, FromName 프로퍼티에 지정하고, 보내는 메시지 내용을 PostIt 메소드를 이용하여 전송하는 것으로 끝이다.

```

procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin
    if Key = #13 then
    begin
        NMMsg1.Host := Edit1.Text;
        NMMsg1.FromName := Edit2.Text;
        NMMsg1.PostIt(Edit3.Text);
    end;
end;

```

이제는 메시지 서버 컴포넌트와 메시지 컴포넌트의 각 이벤트를 처리하기만 하면 된다. 먼저 메시지 서버 컴포넌트의 OnClientContact 이벤트는 클라이언트가 접속했을 때 발생하게 된다.

```

procedure TForm1.NMMSGServ1ClientContact(Sender: TObject);
begin
    NMMsgServ1.ReportLevel := Status_Basic;
    NMMsgServ1.TimeOut := 90000;
    StatusBar1.SimpleText := 'Client connected';
end;

```

그리고, 전송된 메시지는 메시지 서버 컴포넌트의 OnMSG 이벤트에서 처리할 수 있다.

```

procedure TForm1.NMMSGServ1MSG(Sender: TComponent; const sFrom,
    sMsg: String);
begin
    Memo1.Lines.Add('(' + sFrom + ') ' + sMsg);
end;

```

메시지 서버 컴포넌트의 상태에 대한 정보는 OnStatus 이벤트 핸들러에서 처리한다.

```

procedure TForm1.NMMSGServ1Status(Sender: TComponent; Status: String);
begin
    if StatusBar1 <> nil then
        StatusBar1.SimpleText := Status;
end;

```

그 다음에는 메시지 클라이언트 컴포넌트의 이벤트를 처리해야 하는데, 이들의 처리 방법은 앞서 설명한 인터넷 메일 컴포넌트 들과 큰 차이가 없으므로 자세한 설명은 생략하도록 하겠다.

```
procedure TForm1.NMMsg1Connect(Sender: TObject);
```

```
begin
```

```
    StatusBar1.SimpleText := 'Connected';
```

```
end;
```

```
procedure TForm1.NMMsg1ConnectionFailed(Sender: TObject);
```

```
begin
```

```
    ShowMessage('Connection Failed');
```

```
end;
```

```
procedure TForm1.NMMsg1Disconnect(Sender: TObject);
```

```
begin
```

```
    if StatusBar1 <> nil then
```

```
        StatusBar1.SimpleText := 'Disconnected';
```

```
end;
```

```
procedure TForm1.NMMsg1HostResolved(Sender: TComponent);
```

```
begin
```

```
    StatusBar1.SimpleText := 'Host Resolved';
```

```
end;
```

```
procedure TForm1.NMMsg1MessageSent(Sender: TObject);
```

```
begin
```

```
    ShowMessage('Message sent!');
```

```
end;
```

```
procedure TForm1.NMMsg1Status(Sender: TComponent; Status: String);
```

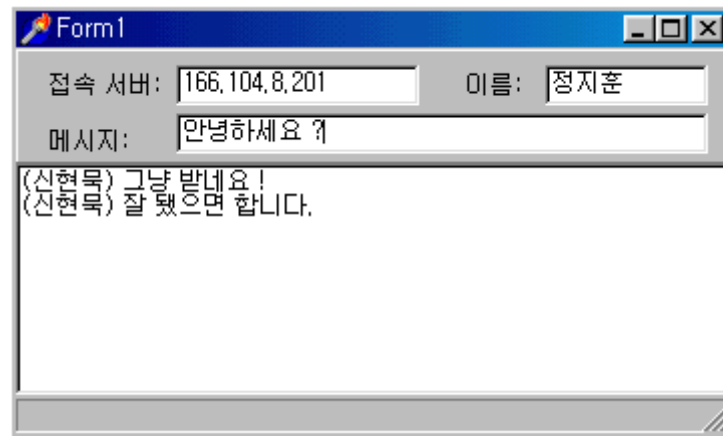
```
begin
```

```
    if StatusBar1 <> nil then
```

```
        StatusBar1.SimpleText := Status;
```

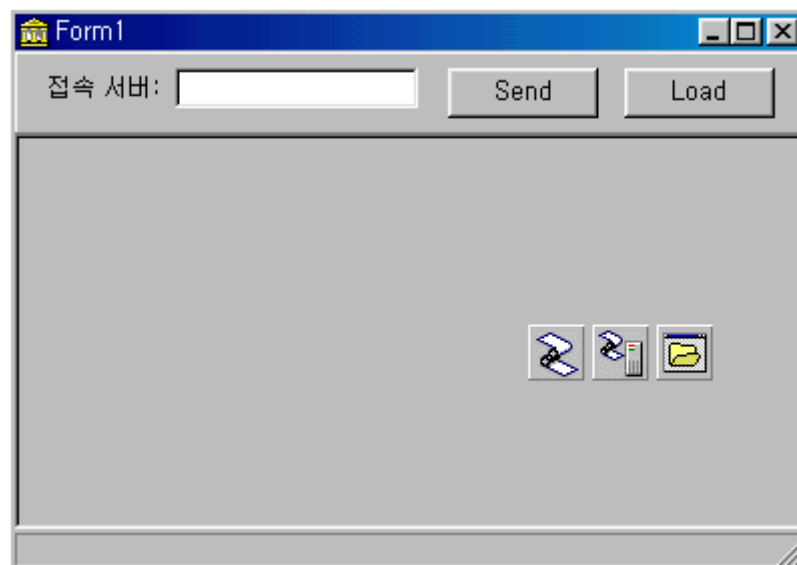
```
end;
```

이제 이 프로그램을 컴파일하고 실행한 뒤에 1:1 채팅을 하듯이 사용해보자. 다음 그림과 같이 무리없이 잘 동작할 것이다.



## 스트림 클라이언트/서버 어플리케이션의 제작

이번에는 스트림 서버와 클라이언트 컴포넌트를 이용하여 파일을 전송하고, 이 파일의 내용을 볼 수 있는 어플리케이션을 만들어 보자. 먼저 서버의 IP 주소를 지정할 수 있는 에디트 박스와 버튼 2 개와 TStatusBar 컴포넌트, 그리고 TOleContainer 컴포넌트와 TOpenDialog, 스트림 클라이언트와 서버 컴포넌트를 각각 하나씩 다음과 같이 폼에 추가하도록 한다.



‘Send’ 버튼을 클릭하면 파일열기 대화상자에서 지정한 파일을 전송하며, ‘Load’ 버튼을 클

릭하면 이 파일을 OLE 컨테이너에서 볼 수 있도록 해준다.

먼저 'Send' 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    MyFStream: TFileStream;
begin
    If OpenFileDialog1.Execute then
    begin
        NMStrm1.Host := Edit1.Text;
        NMStrm1.FromName := ExtractFileName(OpenDialog1.FileName);
        MyFStream := TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
        try
            NMStrm1.PostIt(MyFStream);
        finally
            MyFStream.Free;
        end;
    end;
end;
```

사용 방법은 Host 프로퍼티에 접속할 서버의 IP 주소를 대입하고, 지정된 파일 스트림을 PostIt 메소드를 이용하여 서버로 전송한다. 참고로 FromName 프로퍼티에 파일 이름을 지정하면 파일의 확장자를 알 수 있으므로 이를 활용한다. 즉, 스트림에 대한 정보를 FromName 프로퍼티를 이용하여 문자열로 전송하는 것이다.

이렇게 전송된 파일 스트림은 스트림 서버 컴포넌트의 OnMSG 이벤트에서 받아볼 수 있는데, 이 이벤트 핸들러에서 임시 파일로 저장할 수 있도록 한다.

스트림 서버 컴포넌트의 OnMSG 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.NMStrmServ1MSG(Sender: TComponent; const sFrom: String;
    strm: TStream);
var
    MyFStream: TFileStream;
begin
    Extension := ExtractFileExt(sFrom);
    if FileExists(ExtractFilePath(ParamStr(0)) + 'Wtmp' + Extension) then
        DeleteFile(ExtractFilePath(ParamStr(0)) + 'Wtmp' + Extension);
```

```

MyFStream
:= TFileStream.Create(
    ExtractFilePath(ParamStr(0)) + 'Wtmp' + Extension, fmCreate);
try
    MyFStream.CopyFrom(strm, strm.Size);
finally
    MyFStream.Free;
end;
end;

```

이 이벤트 핸들러에서 파일 이름이 'tmp'이고 파일 확장자는 앞서 스트림 클라이언트 컴포넌트에서 FromName 프로퍼티에 지정한 문자열이 sFrom 파라미터로 넘어오므로, 이 값을 이용하여 확장자를 지정할 수 있다. 여기서는 디렉토리를 현재 어플리케이션이 실행되고 있는 디렉토리에 저장하도록 하였다.

이렇게 저장된 파일을 OLE 컨테이너에서 보여주는 역할을 하는, 'Load' 버튼은 비교적 구현하기 쉽다. 파일이름이 'tmp'이고, 지정된 확장자를 가진 파일을 CreateObjectFromFile 메소드를 이용해서 OLE 컨테이너에 보여준다.

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    if FileExists(ExtractFilePath(ParamStr(0)) + 'Wtmp' + Extension) then
        OleContainer1.CreateObjectFromFile(ExtractFilePath(ParamStr(0)) + 'Wtmp'
            + Extension, False);
end;

```

그리고, 폼의 OnClose 이벤트 핸들러에서 이렇게 생성된 임시 파일을 삭제하도록 한다.

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    if FileExists(ExtractFilePath(ParamStr(0)) + 'Wtmp' + Extension) then
        DeleteFile(ExtractFilePath(ParamStr(0)) + 'Wtmp' + Extension);
end;

```

스트림 컴포넌트의 다른 이벤트 핸들러는 메시지 클라이언트/서버와 마찬가지로 상태 정보를 보여주는 역할을 한다. 각각의 이벤트 핸들러는 다음과 같이 구현하며, 그다지 어려운 내용이 아니므로 자세한 설명은 생략하도록 하겠다.

```
procedure TForm1.NMStrm1Connect(Sender: TObject);
```

```
begin
```

```
    StatusBar1.SimpleText := 'Connected';
```

```
end;
```

```
procedure TForm1.NMStrm1Disconnect(Sender: TObject);
```

```
begin
```

```
    if StatusBar1 <> nil then
```

```
        StatusBar1.SimpleText := 'Disconnected';
```

```
end;
```

```
procedure TForm1.NMStrm1HostResolved(Sender: TComponent);
```

```
begin
```

```
    StatusBar1.SimpleText := 'Host Resolved';
```

```
end;
```

```
procedure TForm1.NMStrm1Status(Sender: TComponent; Status: String);
```

```
begin
```

```
    if StatusBar1 <> nil then
```

```
        StatusBar1.SimpleText := Status;
```

```
end;
```

```
procedure TForm1.NMStrmServ1ClientContact(Sender: TObject);
```

```
begin
```

```
    NMStrmServ1.ReportLevel := Status_Basic;
```

```
    NMStrmServ1.TimeOut := 90000;
```

```
    StatusBar1.SimpleText := 'Client connected';
```

```
end;
```

```
procedure TForm1.NMStrmServ1Status(Sender: TComponent; Status: String);
```

```
begin
```

```
    if StatusBar1 <> nil then
```

```
        StatusBar1.SimpleText := Status;
```

```
end;
```

```
procedure TForm1.NMStrm1ConnectionFailed(Sender: TObject);
```

```
begin
```

```
    ShowMessage('Connection Failed');
```

```
end;
```

```
procedure TForm1.NMStrm1MessageSent(Sender: TObject);
```

```
begin
```

```
    ShowMessage('Stream Sent');
```

```
end;
```

```
procedure TForm1.NMStrm1PacketSent(Sender: TObject);
```

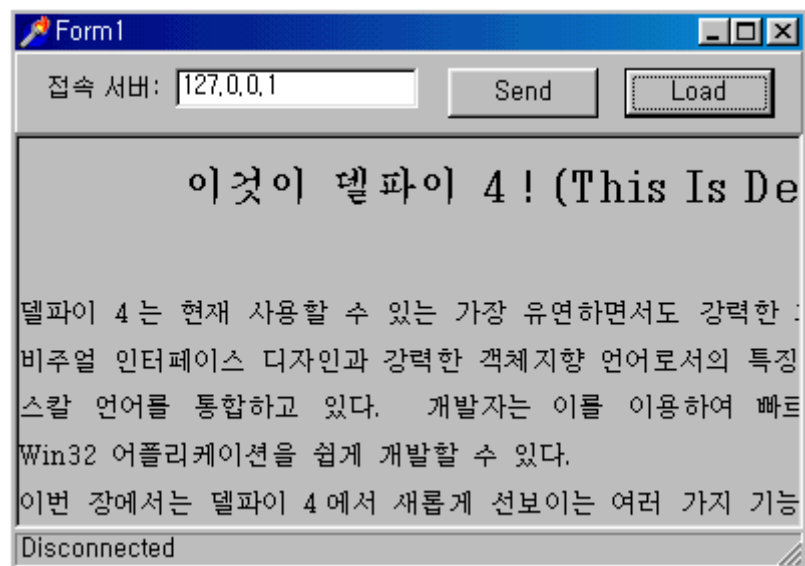
```
begin
```

```
    StatusBar1.SimpleText := IntToStr(NMStrm1.BytesSent) + ' of '
```

```
    + IntToStr(NMStrm1.BytesTotal) + ' Sent';
```

```
end;
```

이것으로 스트림 서버/클라이언트 어플리케이션이 완성되었다. 프로그램을 컴파일하고 실행한 뒤에 Send 버튼을 클릭하여 파일을 지정하면, 서버로 파일 스트림이 전송될 것이다. 다음 그림은 이 책의 1 장에 해당되는 워드 파일을 보여주는 것이다.



## 정 리 (Summary)

델파이는 2.01 버전부터 지원하던 액티브 X 컨트롤 형태의 인터넷 컴포넌트를 버리고, 델파



이 4 부터는 native VCL 형태의 뛰어난 컴포넌트를 제공하여 지금까지의 많은 불평들을 해소시켜 주었다. 또한 TPowersock, TGeneralServer 와 같은 컴포넌트를 이용하면 RFC 에서 제정하는 새로운 프로토콜을 지원하는 컴포넌트를 직접 만들거나, 프로토콜부터 새로 만들어서 사용할 수도 있다.

그렇지만, 중요한 것은 이러한 과정을 제대로 이해하기 위해서는 원속 API 를 이용한 실제 구현 방법에 대해서 알아야 할 것이다. NetMaster 컴포넌트에 대해서 아쉬운 것은 텔파이 4 에 번들되어 있지만, 소스 코드는 공개되지 않았다는 점이다. 이런 측면에서 Francois Piette 의 ICS 컴포넌트 suite 를 추천하고 싶다. 이 컴포넌트는 텔파이 수퍼 페이지나 텔파이 텔리와 같은 사이트에서 쉽게 구할 수 있다.

## 보안과 암호화 기법

### (Security and Cryptograph Techniques)

대부분의 클라이언트/서버 어플리케이션에서는 ID 와 패스워드를 묻는 최소한의 보안을 하고 있다. 그렇지만, 데이터 암호화 등의 다소 높은 수준의 보안이 요구되는 경우도 많다. 이러한 주제를 가지고 정말로 많은 양의 문서와 책들이 쓰여져 왔고, 그리고 많은 연구도 진행 중이다.

그렇지만, 여기에서 다루고자 하는 것은 그렇게 이론적이고 어려운 암호화 기법에 대한 것이 아니다. 실제로 텔파이 어플리케이션을 사용하면서 간단하게 이용할 수 있는 암호화 기법에 대해서 알아보고, 이를 실제로 구현하여 적용하는 방법에 대해서 알아볼 것이다. 그리고, 현재 실용화되어 사용되고 있는 표준 암호화 알고리즘의 종류와 이들을 지원하는 컴포넌트를 소개하고 이를 사용하는 방법에 대해서 알아보도록 한다.

#### 패스워드와 XOR 암호화

보통의 경우 어플리케이션에 패스워드를 이용해서 보안을 하는 경우가 많다. 그런데, 이렇게 사용하는 패스워드를 .INI 파일이나 레지스트리, 일반 텍스트 파일 등에 기록할 경우 아무나 쉽게 패스워드를 간파할 수 있게 된다. 이럴 때에 기본적인 암호화 기법이 필요한데, 가장 쉬우면서도 유용하게 쓸 수 있는 것이 XOR 암호화 기법이다.

이 암호화 기법은 개념적으로도 매우 쉽고, 알고리즘이 간단하기 때문에 수행속도도 매우 빠른 것이 장점이다. 단점은 키를 알게 되면 쉽게 깨질 수 있다는 것이다.

XOR 암호화 기법을 적용하기 위해서는 키가 되는 숫자를 이용하게 된다. 이를 키 번호(키 number)라고 하며 이를 이용해서 메시지를 암호화 하거나, 해독한다. 사용 방법은 암호화하고자 하는 문자나 숫자를 키 번호와 XOR 연산을 행해서 나오는 값을 저장하고, 이를 해독할 때에는 역시 키 번호로 XOR 연산을 하면 된다.

그러면, 실제로 이를 구현하는 코드를 살펴보자.

```
procedure XORBuffer(Buffer: TMemoryStream; 키: String);
```

```
var
```

```
    Len, Place: Word;
```

```
    Character: Char;
```

```
    Size: Integer;
```

```
begin
```

```
    Buffer.Seek(0, 0);
```

```

Len := Length(키);
Place := 1;
Size := Buffer.Size;
while Size > 0 do
begin
    Buffer.Read(Character, 1);
    Character := Char(Byte(Character) xor Byte(키[Place]));
    Buffer.Seek(-1, 1);
    Buffer.Write(Character, 1);
    Inc(Place);
    if Place > Len then Place := 1;
    Dec(Size);
end;
end;

```

키 번호를 문자열로 받아서 그 길이를 이용해서 반복하도록 하고, 암호화가 될 문자열은 버퍼에 담아서 이를 키 번호와 xor 연산을 함으로써 암호화 하는 것이다.

여기에서는 TMemoryStream 클래스를 사용했는데, 문자열을 사용하지 않은 이유는 xor 연산의 결과가 0 일 경우 PChar 문자열의 마지막 문자임을 나타내게 되므로, 이러한 문자열의 규칙에 위배되지 않기 위해서 스트림을 이용하였다. 그렇지만, 이렇게 암호화한 내용을 .INI 파일이나 레지스트리에 저장하고자 하면 문자열로의 변환 과정이 필요하다.

## UUEncoding 과 UUDecoding

유닉스는 8 비트의 데이터 중 7 비트를 사용하는 통신 시스템을 가진 경우가 많은데, 이를 해결하기 위한 방법으로 알려진 것이 UUEncoding, UUDecoding 이라는 것이다. 이 방법을 사용하면 데이터를 ‘!’ 문자인 ASCII 33 부터 ‘z’인 ASCII 122 까지의 문자로 변환하는 것으로, 이를 이용하면 3 개의 문자를 4 개의 바이트에 담게 된다. 이후 4 개의 바이트는 마스크를 거쳐서 32 를 더해서 결과 값으로 출력된다. 이 방법의 단점은 암호화된 데이터가 원래의 이진 데이터 보다 1/4 정도 커진다는 것이지만, 크게 문제가 되지는 않는다/ UUEncoding 을 이용한 암호화 프로시저의 코드는 다음과 같다.

```

procedure UUEncodeBuffer(InBuffer, OutBuffer: PChar; InSize: Word; OutSize: Word);
var
    Chars: Array[0..3] of Byte;
    Hunk: Array[0..2] of Byte;

```

```

i: Integer;
begin
  OutSize := 0;
  repeat
    FillChar(Chars, 4, 0);
    if InSize > 3 then
      begin
        Move(InBuffer^, Hunk, 3);
        Inc(InBuffer, 3);
        Dec(InSize, 3);
      end
    else
      begin
        Move(InBuffer^, Hunk, InSize);
        Inc(InBuffer, InSize);
        Dec(InSize, InSize);
      end;
    Chars[0] := Hunk[0] shr 2;
    Chars[1] := (Hunk[0] shl 4) + (Hunk[1] shr 4);
    Chars[2] := (Hunk[1] shl 2) + (Hunk[2] shr 6);
    Chars[3] := Hunk[2] and $3F;
    for i := 0 to 3 do Chars[i] := (Chars[i] and $3F) + 32;
    Move(Chars, OutBuffer^, 4);
    Inc(OutBuffer, 4);
    Inc(OutSize, 4);
  until InSize <= 0;
end;

```

쉽게 말하자면 암호화할 데이터를 InBuffer, 데이터의 크기를 InSize 에 담아오면 암호화를 거쳐서 암호화된 데이터를 OutBuffer, 크기를 OutSize 로 반환하는 프로시저이다.

이때 3 바이트를 4 바이트로 (3 바이트에 8 비트를 6 비트씩 4 바이트로) 저장한 후, 이를 \$3F 로 마스크 한 후, 32 를 더해서 OutBuffer 에 결과값으로 변환하는 작업을 한다.

이런 UUEncoding 알고리즘을 이용한 암호화 데이터를 풀 때에는 다음과 같은 UUDecoding 프로시저를 사용한다.

```

procedure UUDecodeBuffer(InBuffer, OutBuffer: PChar; InSize: Word; OutSize: Word);

```

```

var
  Chars: Arrays[0..3] of Byte;
  Hunk: Arrays[0..2] of Byte;
  i: Integer;
begin
  OutSize := 0;
  repeat
    FillChar(Hunk, 3, 0);
    if InSize >= 4 then
      begin
        Move(InBuffer^, Chars, 4);
        Inc(InBuffer, 4);
        Dec(InSize, 4);
      end
    else
      begin
        Move(InBuffer^, Chars, InSize);
        Inc(InBuffer, InSize);
        Dec(InSize, InSize);
      end;
    for i := 0 to 3 do
      begin
        if Chars[i] = Ord('') then Chars[i] := Ord(' ');
        Chars[i] := Chars[i] - 32;
      end;
    Hunk[0] := (Chars[0] shl 2) + (Chars[1] shr 4);
    Hunk[1] := (Chars[1] shl 4) + (Chars[2] shr 2);
    Hunk[2] := (Chars[2] shl 6) + Chars[3];
    Move(Hunk, OutBuffer^, 3);
    Inc(OutBuffer, 3);
    Inc(OutSize, 3);
  until InSize <= 0;
end;

```

이런 방식으로 UUEncoding, UUDecoding 을 사용하면 .INI 파일이나 레지스트리에 데이터를 저장할 수 있다. 앞에서 설명한 xor 암호화 기법을 같이 혼용해서 일단 데이터를 xor

로 암호화한 뒤 이를 UUEncoding 알고리즘을 이용해서 변환한 뒤 이를 저장하고, 해독할 때에는 UUDecoding 알고리즘을 이용해서 풀 뒤, 이를 다시 xor 을 이용해서 원래의 값을 얻어내게 된다.

## 암호화 컴포넌트와 표준 알고리즘

암호화라는 것은 앞에서 예를 들어 설명한 것과 같이 특정 알고리즘이나 계산식 등을 통해 데이터를 변경하고, 변경된 데이터를 원할 때 다시 원래의 데이터로 변환할 수 있는 것을 모두 포함한 것이다.

이러한 암호화 알고리즘에는 크게 나누어 private-키와 public-키 알고리즘으로 구분할 수 있다. Private 키 알고리즘을 사용할 때에는 사용자가 암호화와 해독 과정을 모두 같은 키로 할 수 있게 된다. 데이터가 같은 위치에 있을 경우에는 이것이 문제가 되지 않지만, 데이터가 전송되어야 할 경우에는 문제가 될 수 있다. 예를 들어, 보안이 요구되는 메시지를 전송한다고 할 때 전송하는 사람이 private 키를 사용해서 메시지를 암호화한 뒤에 메시지를 보냈다고 하자, 그러면 이를 받는 사람은 암호화하는데 사용한 private 키를 이용해서 메시지를 해독해야 한다. 그런데, 문제는 이렇게 해독할 열쇠가 되는 private 키를 어떻게 안전하게 얻을 수 있는가 하는 문제이다.

이 문제를 해결하기 위해 1976 년 처음 개발된 방법이 바로 public-키 암호화 기법이다. 이 방법은 데이터를 암호화하고 해독하는데 2 개의 연관되었지만 다른 키를 사용하는 것이다. 이때 암호화 키를 public 키라고 하는데, 이렇게 암호화를 하는데 사용된 키가 해독하는데 사용되지 않기 때문에 키가 발견될 염려가 없다. 이때 데이터를 해독할 때에는 private 키가 사용된다. 이제 암호화된 메시지를 전송하고자 하면, 전송하고자 하는 사람이 public 키를 보내고 이를 이용해서 메시지를 암호화한다. 그리고, 암호화된 메시지를 받은 쪽에서는 자신의 private 키를 사용해서 해독하면 된다.

Private-키 암호화는 쉽게 말해서 데이터에 대한 패스워드를 알고 있는 사람들끼리만 이 패스워드를 이용한 해성 함수를 처리함으로써 암호화를 하는 방법이고, public-키 암호화는 패스워드를 걸지 않고도 일반적인 데이터에 대한 public-키를 생성해서 암호화를 하는 방법이다.

Public-키 암호화 알고리즘은 private-키 알고리즘에 비해 암호화와 해독을 할 때 1,000 배가 더 걸린다. 또한, Public-키 암호는 private-키 암호에 비해 같은 정도의 보안 레벨을 제공하기 위해서는 10 배는 더 길어야 한다.

표준화된 암호화 알고리즘으로는 매우 여러 가지가 있지만 널리 쓰이는 것으로는 DES, MD5, RC5, SHA 등이 있다. 이들은 모두 public-키 암호화 기법으로 이들 중에서 DES 와 SHA 에 대해서 알아보자. 그에 비해 앞에서 설명한 XOR 연산자를 이용한 암호화는 일종의 private-키 암호화 기법이라고 말할 수 있다.

- DES

DES 는 Data Encryption Standard 의 약자로 암호학계에서 20 년간의 테스트를 거친 알고리즘이기 때문에 특별한 약점이 없는 표준적인 알고리즘이다. DES 는 암호화와 해독을 위해서 64 비트 블록에 대해서 56 비트의 키를 사용한다. 최근의 경향으로 보아 56 비트의 키는 다소 짧다는 의견이 대두되고 있지만, 보안의 정도가 아주 높아야 하는 경우를 제외하고는 사용하는데 큰 무리가 없다.

- SHA

SHA 는 Secure Hash Algorithm 의 약자로 NIST 와 NSA 에서 개발된 암호화 표준이다. 마이크로소프트에서의 code signing 에서도 MD5 와 함께 표준으로 사용되고 있다. SHA 알고리즘은 160 비트의 해쉬를 만들어내기 때문에, 128 비트를 이용하는 MD5 에 비해 보안의 정도가 높다고 할 수 있다.

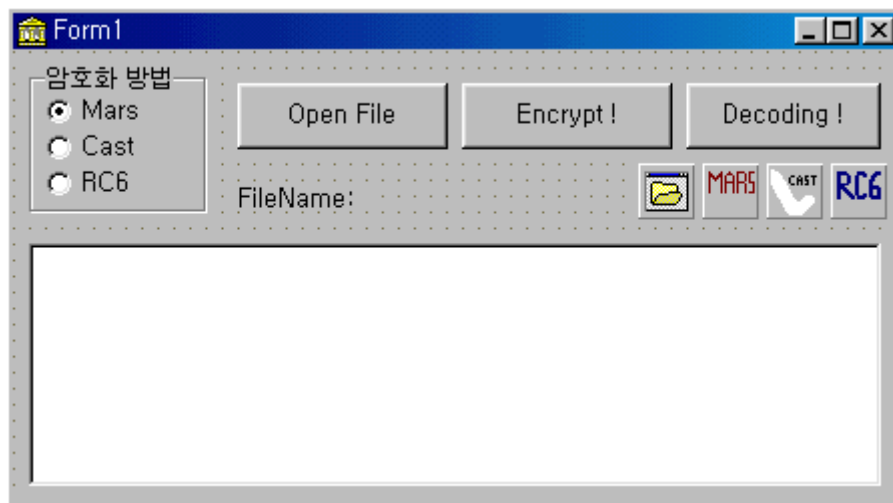
- 암호화 컴포넌트의 활용

인터넷에 보면 여러가지 암호화 컴포넌트가 프리웨어로 공개되고 있다. 그중에서도 표준 알고리즘인 DES, SHA 를 비롯한 여러가지 알고리즘을 구현한 컴포넌트 들을 DSP(Delphi Super Page)나 델파이 델리(Delphi Deli) 등에서 구할 수 있다. CD-ROM 과 함께 제공되는 40 장 디렉토리에 보면, Crypt 라는 서브 디렉토리가 있는데 여기에 앞으로 차세대 암호화 알고리즘의 표준으로 제안되고 있는 Cast, RC6, Mars 알고리즘을 컴포넌트화한 유닛의 소스코드가 포함되어 있다. 이들 알고리즘에 대해서 더 자세히 알고 싶으면 Mars 는 <http://www.research.ibm.com/security/mars/>, Cast 는 <http://www.entrust.com/>, RC6 는 <http://theory.lcs.mit.edu/~rivest/> 홈 페이지를 방문하기 바란다. 이들 컴포넌트는 모두 Duff Neill 에 의해 제작되었다. 이 컴포넌트 들을 이용하여 예제 프로그램을 작성할 것이므로 이들을 컴포넌트 팔레트에 설치하기 바란다.

그 밖에도 David Barton 이 RC5, Blowfish, rmd160, RC5, Misty, IDEA, Skipjack 등의 암호화 기법을 구현한 컴포넌트를 공개하였는데 이를 Others 서브 디렉토리에 같이 제공하고 있다. David Barton 은 델파이를 이용한 암호화 컴포넌트 페이지를 운영하고 있는데, 관심 있는 독자들은 <http://web.ukonline.co.uk/david.w32/delphi.html> 주소에서 운영하는 그의 홈 페이지를 방문해보기 바란다.

그러면, 이 컴포넌트 들을 이용하여 파일을 암호화하고 해독하는 방법을 알아보자.

먼저 폼 위에 라디오 그룹과 라벨 컴포넌트 2 개, 버튼 3 개를 얹고 TOpenDialog, TMars, TCast, TRC6 컴포넌트와 메모 컴포넌트를 다음과 같이 폼에 하나씩 추가한다.



그리고, 각 컴포넌트의 프로퍼티를 앞의 그림에 맞도록 설정한다. 여기서 Open File 버튼을 클릭하면 메모 컴포넌트에 읽어올 파일을 대화상자에서 선택하도록 한다. 그리고, Encrypt 버튼을 클릭하면 라디오 그룹에서 선택한 암호화 방법으로 암호화하고 파일의 확장자가 .enc 인 파일로 저장한다. Decoding 버튼을 클릭하면 확장자가 .enc 인 암호화된 파일의 내용을 해독해서 확장자가 .dec 인 파일로 저장한다.

이때 이런 과정의 내용을 메모 컴포넌트에서 계속 볼 수 있도록 한다.

TRC6, TMars, TCast 컴포넌트의 사용 방법은 완전히 동일하다. 키가 될 문자열을 Key 프로퍼티에 설정하고, InputFile 프로퍼티와 OutputFile 프로퍼티에 각각 읽어올 파일 이름과 암호화나 해독 과정을 거쳐 생성될 출력 파일 이름을 지정한 뒤에 암호화할 경우에는 EncipherFile, 해독할 경우에는 DecipherFile 메소드를 호출하면 된다.

먼저 파일을 읽어오는 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
    begin
        Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
        Label2.Caption := OpenFileDialog1.FileName;
    end;
end;
```

여기서 미리 ‘텍스트 파일’이나 ‘모든 파일’을 선택할 수 있도록 OpenFileDialog 객체의 Filter 프로퍼티를 오브젝트 인스펙터에서 설정하도록 한다. 그리고, Label2 는 암호화와 해독이 진행되는 원래 파일의 이름을 보여주도록 하는 것이다.

암호화와 해독을 진행하도록 하는 Button2 와 Button3 의 OnClick 이벤트 핸들러는 다음과



같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0:
      begin
        Mars1.Key := 'Sample Key';
        Mars1.InputFile := Label2.Caption;
        Mars1.OutputFile := ChangeFileExt(Label2.Caption, '.enc');
        Mars1.EncipherFile;
      end;
    1:
      begin
        Cast1.Key := 'Sample Key';
        Cast1.InputFile := Label2.Caption;
        Cast1.OutputFile := ChangeFileExt(Label2.Caption, '.enc');
        Cast1.EncipherFile;
      end;
    2:
      begin
        RC61.Key := 'Sample Key';
        RC61.InputFile := Label2.Caption;
        RC61.OutputFile := ChangeFileExt(Label2.Caption, '.enc');
        RC61.EncipherFile;
      end;
  end;
  Memo1.Lines.LoadFromFile(ChangeFileExt(Label2.Caption, '.enc'));
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0:
      begin
        Mars1.Key := 'Sample Key';
```

```

    Mars1.InputFile := ChangeFileExt(Label2.Caption, '.enc');
    Mars1.OutputFile := ChangeFileExt(Label2.Caption, '.dec');
    Mars1.DecipherFile;
end;
1:
begin
    Cast1.Key := 'Sample Key';
    Cast1.InputFile := ChangeFileExt(Label2.Caption, '.enc');
    Cast1.OutputFile := ChangeFileExt(Label2.Caption, '.dec');
    Cast1.DecipherFile;
end;
2:
begin
    RC61.Key := 'Sample Key';
    RC61.InputFile := ChangeFileExt(Label2.Caption, '.enc');
    RC61.OutputFile := ChangeFileExt(Label2.Caption, '.dec');
    RC61.DecipherFile;
end;
end;
Memo1.Lines.LoadFromFile(ChangeFileExt(Label2.Caption, '.dec'));
end;

```

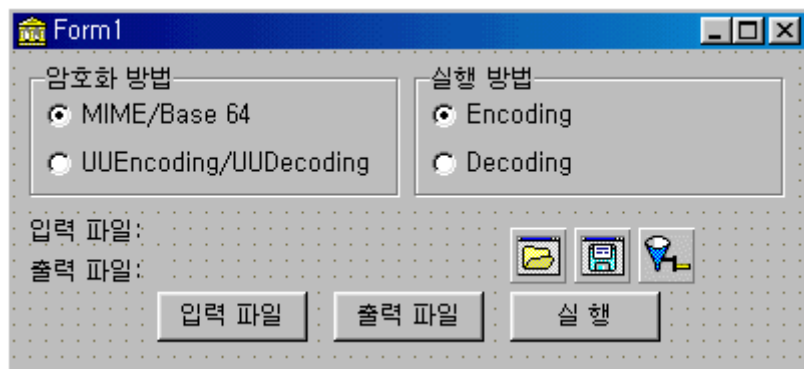
이들 컴포넌트의 사용 방법에 대해서는 앞서도 간단히 설명하였으므로 자세한 설명은 생략한다. 참고로 ChangeFileExt 함수는 파일 이름의 확장자를 2 번째 파라미터에 지정한 문자열로 변경하기 때문에 읽어들인 파일의 확장자를 .enc 와 .dec 로 변경하여 지정할 수 있다.

컴파일을 하고, 이를 실행한 뒤에 Open File 버튼을 클릭하여 암호화할 대상 파일을 선택하도록 하자. 그리고, ‘Encrypt !’ 버튼을 클릭하면 암호화 과정을 거쳐 .enc 파일이 생성되면서 이 파일을 메모에 읽어 온다. 아마도 다음과 같이 알 수 없는 문자열이 보일 것이다. 이제 다시 ‘Decoding !’ 버튼을 클릭하면 해독 과정을 거쳐 .dec 파일이 생성되면서 이 파일을 메모에 읽어 온다. 이 과정을 거치면 원래 텍스트 파일과 같은 내용으로 복원되는 것을 확인할 수 있을 것이다.

## NetMaster 컴포넌트의 활용

39 장에서도 이미 설명한 바 있지만, MIME/Base 64 암호화나 UUEncoding/UUDecoding 암호화는 TNMUUProcessor 컴포넌트를 이용해서 수행할 수 있다. 앞서 UUEncoding 과 UUDecoding 을 하기 위한 루틴을 소개한 바 있으나, TNMUUProcessor 컴포넌트를 이용하여 암호화를 수행하는 예제를 하나 작성해보도록 하자.

먼저 폼에 TRadioGroup 컴포넌트 2 개와 버튼 3 개, 그리고 TTable 컴포넌트 4 개와 TOpenDialog, TSaveDialog, TNMUUProcessor 컴포넌트를 하나씩 올려 놓고 다음과 같이 디자인한다.



‘입력 파일’과 ‘출력 파일’ 버튼을 클릭하면 Label3, Label4 에 입력과 출력으로 사용될 파일의 이름이 표시되도록 다음과 같이 Button2, Button3 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
        Label3.Caption := OpenFileDialog1.FileName;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    if SaveDialog1.Execute then
        Label4.Caption := SaveDialog1.FileName;
end;
```

그리고, ‘실행’ 버튼을 클릭하면 라디오 그룹에서 선택한 방법으로 인코딩 또는 디코딩을 하도록 다음과 같이 Button1 의 OnClick 이벤트 핸들러를 작성하면 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    InStream, OutStream: TFileStream;
begin
    InStream := TFileStream.Create(Label3.Caption, fmOpenRead);
    OutStream := TFileStream.Create(Label4.Caption, fmCreate);
    try
        case RadioGroup1.ItemIndex of
```

```

0: NMUUProcessor1.Method := uuMIME;
1: NMUUProcessor1.Method := uuCode;
end;
NMUUProcessor1.InputStream := InStream;
NMUUProcessor1.OutputStream := OutStream;
case RadioGroup2.ItemIndex of
0: NMUUProcessor1.Encode;
1: NMUUProcessor1.Decode;
end;
finally
InStream.Free;
OutStream.Free;
end;
end;
end;

```

이제 프로그램을 컴파일하고 실행한 뒤에 해당되는 파일들의 이름을 지정하고, ‘실행’ 버튼을 클릭하면 암호화된 파일이 생성되거나, 암호화된 파일이 해독될 것이다.

## 인터넷 보안 표준 (Internet Security Standards)

인터넷이 일반화 되면서, 전자 상거래도 활발하게 적용되고 있다. 이러한 인터넷에서도 보안은 매우 큰 문제라고 할 수 있다. 이를 위해 암호화와 같은 보안 기법이 절실히 요구된다.

마이크로소프트와 넷스케이프 브라우저에서 사용하는 보안 프로토콜은 public-키와 private-키 암호화 방법의 장점을 모두 수용하고 있다. 가장 널리 쓰이는 보안 프로토콜이 바로 SSL(Secure Sockets Layer)이다. SSL 은 네비게이터의 첫번째 버전과 IE 3.0 부터 채용하고 있다. 1995 년에 마이크로소프트는 새로운 보안 프로토콜을 제안했는데, 이것이 바로 PCT(Private Communications Technology)이다. 이것 역시 IE 3.0 에서부터 채용되었다. 그밖에 IETF(Internet Engineering Task Force)에서는 SSL 을 바탕으로 새로운 인터넷 표준 보안 프로토콜을 제안했는데 이것이 TLS(Transport Layer Security)이다.

### ● SSL(Secure Sockets Layer)

네비게이터 3.0 과 4.0 의 좌하단 코너에는 보안 아이콘이 나타난다. 네비게이터 3.0 에서는 아이콘의 형태가 체인의 형태이고, 4.0 에서는 자물쇠 형태이다. 끊어진 체인이 연결되거나, 자물쇠가 잠긴 형태로 바뀌면 이것은 서버에 접근할 때 보안 세션으로 들어간다는 것을 의

미한다.

브라우저가 보안이된 웹 페이지에 처음 접속할 때에 서버는 'hello request' 메시지를 전송한다. 보안 세션을 시작하기 위해서는 브라우저가 'client hello,'라고 불리는 메시지로 반응해야 하며, 서버는 'server hello.'로 응답해야 한다. 이런 초기 과정에서 브라우저와 서버는 handshake 프로토콜을 이용해서 보안 정보를 주고 받는다. 이 과정이 SSL 의 첫번째 파트이다. 클라이언트의 'hello' 메시지는 세션 ID 라는 숫자를 가지고 있는데, 이를 이용해서 브라우저와 서버 간의 세션을 나타낸다. 또한, 메시지는 서버에게 어떤 암호화 알고리즘을 쓸 것인지, 그리고 SSL 의 버전과 브라우저가 지원하는 압축 방법 등을 알려준다. 마지막으로 브라우저가 생성해낸 난수를 포함한다. 서버 'hello' 메시지는 브라우저에 의해 제공되는 것들 중에서 압축 방법과 암호화 알고리즘, 적절한 SSL 버전과 다른 난수, 가능한 세션 ID 등을 선택해서 반응하게 된다.

이 단계에서 클라이언트와 서버는 디지털 certificate 를 서로 교환하게 되는데, 이를 통해서 서로의 내용을 확인하게 된다. 서버의 certificate 에는 handshake 프로토콜에서 선택된 public-키 암호화 알고리즘에 적합한 public 키를 포함하며, 이 키가 짧은 시간 이용된다. 그렇지만, 실제적인 트랜잭션은 private-키 암호화 기법을 이용해서 암호화된다.

이를 구현하기 위해서 서버와 클라이언트는 브라우저에 의해 생성된 하나의 private 키를 가지고, 서버로의 전송을 위한 마스터 키로 public 키만을 사용하는 것이 아니라 브라우저가 premaster secret 키를 대신 보낸다. 이미 정의된 프로토콜에 따라 서버는 premaster secret 키를 이용하여 실제 마스터 키를 결정한다. 이렇게 함으로써 실제 마스터 키를 전송할 필요가 없게 되며, 이런 프로세스가 완료되면 브라우저와 서버는 마스터 키의 복사본을 가지게 된다.

#### ● 인터넷 익스플로러 보안 (Internet Explorer Security)

IE 3.0 은 SSL 과 PCT 를 모두 지원한다. PCT 는 SSL 과 마찬가지로 private 키를 암호화할 때 public 키 암호를 사용하는 방식을 이용한다. SSL 과 PCT 의 가장 큰 차이점은 바로 handshake 프로토콜 단계에 있다. PCT 는 호환 가능한 프로토콜과 접속하기 위해 보다 적은 수의 메시지를 요구하며, 더 많은 암호화 알고리즘을 지원한다. 또한, 인증(authentication)과 암호화에 서로 다른 키를 사용함으로써 보안의 정도가 높다.

IE 4.0 에서는 이 밖에도 자체적인 보안 구역(security zone)을 이용하여 사용자가 자신의 브라우저의 보안 레벨을 조절할 수 있도록 하고 있다. 각각의 보안 구역에 특정 행동만 할 수 있도록 보안 레벨을 부여할 수 있다. 예를 들어, 회사의 인트라넷 사이트를 trusted zone 으로 설정하면 특별한 암호화를 거치지 않고 이 사이트와 통신할 수 있다. 반대로 처음으로 방문하는 인터넷 사이트를 untrusted zone 을 설정하면, 항상 서버에게 브라우저가 정보를 보내기 전에 SSL 인증을 할 것을 요구하게 된다.

- 인터넷 보안의 미래 – TLS (Transport Layer Security)

TLS 프로토콜은 SSL 에 기초한 프로토콜로 앞으로 표준이 될 가능성이 높다.

참고로 신용카드 회사를 중심으로 새로운 보안 표준을 개발하고 있는데, 이것이 SET(Secure Electronic Transaction) 표준이다. SET 는 TLS 와 같은 프로토콜과 함께 사용할 수 있으며, 인증과 신뢰성에 초점을 맞춘 프로토콜이다

## 정 리 (Summary)

이번 장에서는 가장 기본적인 보안을 유지하기 위해 간단한 암호화 방법에서부터 현재 표준적으로 사용되고 있는 몇 가지 알고리즘을 소개하고, 차세대 암호화 표준 알고리즘으로 제안되고 있는 알고리즘을 구현한 컴포넌트를 이용하여 간단한 예제를 작성해 보았다. 그리고, 인터넷에서 제안되고 있는 몇 가지 암호화에 관한 프로토콜을 소개하였다.

최근에 이런 암호화와 보안의 중요성은 날이 갈수록 강조되고 있다. 물론 제대로 된 보안을 위해서는 하드웨어적인 보안과 설비 등이 필요하고 고려해야 될 요소가 많지만, 의외로 앞에서 설명한 간단한 방법으로도 웬만한 수준의 보안은 유지할 수 있으므로 조금만 더 신경을 써서 어플리케이션을 마무리 할 것을 권하는 바이다.

## 다중 쓰레드 프로그래밍 기법 (Multi-Threaded Programming Techniques)

Win32 환경은 기본적으로 쓰레드를 기본적인 실행 단위로 하는 환경이다. 그렇기 때문에, 최근에 멀티 쓰레딩이라는 단어를 심심치 않게 듣게 되며 또한 조금만 복잡한 어플리케이션을 개발할 때에도 멀티 쓰레드를 지원하지 않으면 이는 Win32 어플리케이션으로서의 자격이 없다고 까지 말하고 있다.

그렇다면 멀티 쓰레드를 지원하는 어플리케이션이란 어떤 것을 말하며, 이를 지원하려면 어떤 지식을 알아야 하는가? 이번 장은 여기에 대한 궁금증을 파헤쳐볼 것이다.

마이크로소프트의 Win32 환경 중에서 가장 괄목할 만하게 성장한 기술이라고 생각되는 것 중의 하나가 멀티 태스킹이라고 할 수 있다. 윈도우 95 나 NT에서는 프로세스와 쓰레드란 개념을 가지고 멀티 태스킹을 행한다. 프로세스(Process)란 실행 중인 프로그램의 인스턴스를 가리키는 것으로 예를 들어, 텔파이 4 인스턴스 하나와 MS 워드 인스턴스 하나를 현재 실행하고 있다면 두 개의 프로세스가 시스템에서 실행되고 있는 것이다. 쓰레드(Thread)는 이러한 프로세스 내에서 실행되는 경로(path)를 의미하는 것으로, 프로세스가 운영체제에 의해서 생겨날 때, 쓰레드도 같이 생성된다. 이렇게 프로세스와 같이 생겨나는 쓰레드를 메인 쓰레드라고 한다. 모든 프로세스는 최소한 하나의 쓰레드를 가지고 있으며, 메인 쓰레드가 실행되면 프로세스 내에서 다른 쓰레드를 생성할 수 있게 된다.

다시 말해, 프로세스는 고유한 메모리, 파일 핸들, 그리고 다른 시스템 자원을 갖는 실행 중인 프로그램을 말하며 모든 프로세스는 쓰레드라는 개별적인 실행 경로를 가지는 것이다. 대부분의 경우 프로세스 내의 모든 쓰레드는 프로세스의 모든 코드와 데이터 공간을 공유하는데, 예를 들어 두 쓰레드가 동일한 전역 변수를 공유할 수 있다. 쓰레드는 기본적으로 운영체제에 의해 관리되며 각 쓰레드는 고유한 스택을 가지고 있다.

멀티 쓰레드의 예를 들어보자. 어떤 프로그램이 데이터베이스에 접근해서 정보를 검색하고, 동시에 프린터에 인쇄를 한다고 하자. 이를 하나의 프로세스에서 하나의 쓰레드로 구현하려면 데이터베이스 검색이 끝나고 인쇄를 하던가, 아니면 인쇄를 하고 데이터베이스 검색을 해야 한다. 그렇지만 데이터베이스 검색 쓰레드와 인쇄 쓰레드를 각각 생성해서 동시에 실행시킨다면 이것이 곧 멀티 쓰레드가 실행되는 것이다.

### 멀티 태스킹과 멀티 쓰레드

프로세스는 실행을 위해 메모리에 적재된 프로그램을 말한다. 각각의 프로세스는 가상 주소 공간을 가지고 있으며, 코드와 데이터 그리고 파일, 파이프, 동기화 객체 등의 여러가지 시스템 리소스로 이루어져 있다. 이들은 하나의 쓰레드로 시작하지만, 추가적인 쓰레드를



생성할 수 있다.

쓰레드는 다른 쓰레드에 의해서 실행되는 부분을 포함해서, 프로그램의 어떤 부분의 코드가든 실행할 수 있다. 쓰레드는 운영체제가 CPU의 시간을 할당하는 기본 단위가 되며, 프로세스의 모든 쓰레드들은 가상 주소 공간을 공유하기 때문에 전역 변수와 프로세스의 시스템 리소스에 접근할 수 있다.

멀티 태스킹 운영체제는 가능한 CPU 시간을 쓰레드에 분할해서 사용하게 되는데, Win32 API는 선점형 멀티 태스킹에 적합하도록 디자인되었기 때문에, 현재 실행 중인 쓰레드가 지정된 시간이 지나면 실행이 중단되고 다른 쓰레드가 작동하게 되어 있다. 이때 각 쓰레드에 대한 주소 공간이나 스택 등의 내용을 담고 있는 구조체를 저장했다가 복구하는 작업을 하게 된다.

언제 멀티 태스킹을 사용하는가 ?

다음과 같은 경우에는 멀티 쓰레드 프로그래밍이 매우 유용하다.

- 멀티 윈도우의 입력을 처리할 때
- 여러 개의 통신 장비에서 입력을 처리할 때
- 각각의 태스크들에 우선권(priority)을 부여할 필요가 있을 때

멀티 프로세스를 이용해서 멀티 태스킹을 하는 것보다 하나의 프로세스 공간에서 멀티 쓰레드를 사용하는 것이 좋은 점은 다음과 같다. (멀티 프로세스와 멀티 쓰레드의 차이점은 모두 알고 계시죠 ? 멀티 프로세스는 프로세스 공간이 여러 개인 경우이며, 멀티 쓰레드는 쓰레드가 여러 개)

- 시스템은 프로세스를 생성하고 실행하는 것보다 쓰레드를 생성하고 실행하는 것을 훨씬 빠르게 수행한다. 이는 쓰레드에 대한 코드는 이미 프로세스의 주소 공간에 매핑되어 있는데 비해, 새로운 프로세스에 대한 코드는 반드시 다시 로드되어야 하기 때문이다.
- 프로세스 내의 모든 쓰레드들은 같은 주소 공간을 사용하기 때문에, 프로세스의 전역 변수에 접근할 수 있다. 그러므로, 이를 이용해서 쓰레드 간의 통신이 용이하다.
- 프로세스내의 모든 쓰레드는 파일이나 파이프 등의 프로세스의 리소스에 대한 핸들을 사용할 수 있다

멀티 쓰레드를 사용할 때에 반드시 장점만 있는 것은 아니다. 여러 개의 쓰레드가 같은 리소스에 접근할 경우에는 이들이 충돌하지 않도록 동기화 해주어야 한다. 특히 통신 포트나 디스크 드라이브 등의 시스템 리소스에 접근하는 경우이거나, 파일이나 파이프 핸들 등의

공유 리소스에 대한 핸들, 그리고 프로세스의 전역 변수 등에 대해서는 각별히 주의해서 사용해야 한다. 적절한 동기화를 해주지 못하면 deadlock 등의 심각한 상황에 빠질 수 있다.

## Win32 API 의 스레드에 대한 지원

이 장에서 Win32 API 를 이용해서 직접 멀티 스레드를 구현하는 예제를 하나 제시하고 있다. 그렇지만, 멀티 스레드를 이해하기 위해서는 실제로 Win32 API 에서 스레드에 대해서 어떤 내용을 지원하고 있는지를 아는 것이 큰 도움이 된다. 여기서는 이들에 대해서 알아보기로 한다.

### ● 스케줄 우선권 (Scheduling Priorities)

각 스레드의 스케줄 우선권은 다음의 criteria 에 의해 결정된다.

- 프로세스의 priority class (high, normal, idle)
- 프로세스의 priority class 내의 스레드 priority level (lowest, below normal, normal, above normal, highest)
- 시스템이 스레드의 base priority 에 적용하는 dynamic priority boost

디폴트로 프로세스의 priority class 는 normal 로 설정된다. 그리고, CreateProcess API 함수는 부모 프로세스가 자식 프로세스의 priority class 를 지정하는 것을 허용한다. 프로세스의 priority class 를 변경하기 위해서는 SetPriorityClass 함수를 사용하며, 현재의 priority class 는 GetPriorityClass 를 사용해서 얻을 수 있다.

이와 비슷하게 스레드의 priority level 은 SetThreadPriority 함수와 GetThreadPriority 함수를 이용해서 변경하거나, 얻을 수 있다.

각 스레드의 base priority 가 결정되면, 스케줄러가 이를 이용해서 스레드의 dynamic priority 를 결정하게 된다.

스케줄러는 스레드의 dynamic priority 를 올리거나 내림으로써 스레드에 특정 사건이 생겼을 때의 반응성을 높일 수 있다. 예를 들어, 윈도우가 키보드 입력이나 마우스 움직임 등의 입력 메시지를 받게 되면 스케줄러는 해당 윈도우를 소유한 프로세스에 있는 모든 스레드의 priority 를 상향 조정한다. 이렇게 상향 조절된 dynamic priority 는 스레드가 하나의 time slice 를 완료할 때마다 레벨이 하나씩 낮아져서, 결국에는 base priority 에 도달하게 된다.

이러한, dynamic priority boost 외에, 스케줄러는 foreground window 에 대한 프로세스의 priority class 를 상향 조정한다. Foreground window 는 background 프로세스와 최저 같거나 높은 priority 를 가지게 된다. 사용자는 이렇게 설정된 foreground priority boost 를

ALT+ TAB 이나 ALT+ ESC 키를 같이 누름으로써 해제할 수 있다.

각 priority 클래스의 의미는 다음과 같다.

Priority	의 미
HIGH_PRIORITY_CLASS	프로세스가 정확하게 실행되기 위해서는 즉시 실행되어야 하는 time-critical 한 작업을 하고 있다는 의미로, normal 또는 idle priority class 를 가지는 다른 쓰레드들에 우선한다. 이를 사용하는 예로는 CTRL+ALT+DEL 키를 누르면 실행되는 윈도우의 작업 관리자를 들 수 있다. CPU 를 독차지 하므로 지극히 제한된 용도로만 사용하여야 한다.
IDLE_PRIORITY_CLASS	시스템이 실제로 실행시킬 쓰레드가 없을 때 실행되는 정도의 priority class 이다. 스크린 세이버가 여기에 해당된다.
NORMAL_PRIORITY_CLASS	특별한 스케줄링이 필요 없는 정상적인 프로세스임을 나타낸다.
REALTIME_PRIORITY_CLASS	가장 높은 수준의 priority class 로, 중요한 작업을 수행하는 운영체제의 쓰레드 보다 높은 priority 이다. 이런 쓰레드가 동작할 경우 디스크 캐쉬가 제대로 동작하지 않거나, 마우스 동작이 반응을 하지 않는 등의 현상이 나타날 수 있다.

멀티 쓰레드 프로세스에 대한 priority class 를 선택하고 나서, SetThreadPriority API 함수를 이용해서 상대적인 priority 를 적용시킬 수 있다. 보통은 프로세스의 입력 쓰레드(input thread)에 높은 priority 를 적용하여 사용자의 입력에 대한 반응성을 높인다. 또한, 비교적 CPU 시간을 많이 잡아먹은 쓰레드일 수록 상대적으로 낮은 priority 를 부여하여 필요할 때마다 다른 쓰레드가 실행될 수 있도록 한다. 주의할 점은 높은 priority 를 가진 쓰레드가 낮은 priority 의 쓰레드의 작업이 완료되기를 기다릴 경우에 반드시 wait 함수나 critical section, Sleep 기능을 하는 함수를 이용해야 한다. 만약 여기에서 loop 를 이용할 경우 deadlock 에 들어갈 수 있다.

## ● 쓰레드의 생성

Win32 API 에서는 CreateThread 함수를 이용해서 프로세스에 새로운 쓰레드를 생성하게 된다. 이 때 새로운 쓰레드가 실행할 코드의 주소를 지정해야 한다.

CreateThread 함수에서 사용되는 파라미터는 다음과 같다.

- 새로운 쓰레드의 핸들에 대한 보안 속성(security attributes). 이러한 보안 속성에는 핸들이 자식 프로세스에 의해 상속될 수 있는지 여부를 결정하는 상속 플래그(inheritance flag)가 포함된다. 또한, 실제로 쓰레드 핸들에 접근하기 전에 접근을 허

용할 것인지 여부를 결정할 때 시스템에 의해 사용되는 보안 설명자(security descriptor)도 포함하고 있다.

- 새로운 쓰레드에 대한 초기 스택 크기. 쓰레드의 스택은 프로세스의 메모리 공간에 자동으로 할당된다. 또한 필요에 따라 커지기도 하고, 쓰레드가 종료되면 해제된다.
- 쓰레드가 생성될 때 일시중지 상태(suspended state)로 시작할 것인지 여부를 결정하는 플래그. 일시중지 상태로 설정되면 쓰레드는 ResumeThread 함수가 호출될 때까지는 실행되지 않는다.

#### ● 멀티 쓰레드의 동기화

멀티 쓰레드로 동작하는 프로그램에서 공유된 자원에 접근할 경우나, 독립된 코드가 적절한 순서로 실행되어야 하는 경우에 동기화가 필요하다. Win32 API에서는 이러한 동기화에 사용할 수 있는 다음과 같은 여러가지 객체를 제공한다.

- 동기화 객체: 뮤텝(mutex), 세마포어(semaphore), 이벤트(event)
- 파일 핸들 (File handle)
- 콘솔 입력 버퍼 핸들 (Console input buffer handle)
- 명명된 파이프 핸들 (Named pipe handle)
- 통신 디바이스 핸들 (Communication device handle)
- 프로세스 핸들 (Process handle)
- 쓰레드 핸들 (Thread handle)

이들 객체의 상태가 변할 때까지 실행을 중지하게 할 때에는 WaitForSingleObject, WaitForMultipleObjects, WaitForSingleObjectEx 또는 WaitForMultipleObjectsEx 함수를 사용한다.

이들 객체 중 일부는 특정 이벤트가 발생할 때까지 쓰레드가 실행되지 않도록 하는데 유용하다. 예를 들어, 콘솔 입력 버퍼 핸들의 경우 키보드가 눌리거나, 마우스 버튼이 눌릴 때까지 실행을 중지할 수 있고, 프로세스와 쓰레드 핸들의 경우에는 특정 프로세스나 쓰레드가 종료될 때 실행 여부를 결정하게 할 수 있다.

다른 객체의 경우 공유된 자원을 동시에 접근하지 못하도록 하는데 유용하게 사용할 수 있다. 예를 들어 각각의 쓰레드는 뮤텝 객체에 대한 핸들을 가질 수 있는데, 공유 자원에 접근하기 전에 쓰레드는 반드시 뮤텝의 상태가 signaled 로 변할 때까지 대기하는 함수를 호출해야 한다. 뮤텝이 signaled 되면 대기 중인 쓰레드 중에서 하나의 쓰레드가 자원에 접근할 수 있게 되며, 뮤텝 객체의 상태는 다시 non-signaled 로 변경되어 다른 쓰레드의 접근을 허용하지 않게 된다. 쓰레드가 자원의 사용을 끝내면 다른 쓰레드가 접근할 수 있도록 뮤텝 객체의 상태가 signaled 로 변경된다.

단일 프로세스의 스레드들일 경우에는 임계섹션 객체(critical section)를 사용할 수 있으며 효율이 조금 더 뛰어나다.

- 스레드 로컬 저장소 (Thread Local Storage)

하나의 프로세스에 있는 모든 스레드 들은 가상 메모리 공간과 전역 변수를 공유하게 된다. 그렇지만 각각의 스레드에 대한 정적인 저장소가 필요한 경우가 있는데, 이럴 때에 물론 스레드 함수의 지역 변수를 사용할 수도 있지만, 이러한 지역 변수는 특정 함수 내에서만 값을 유지하기 때문에 그 유용성이 떨어진다. 이와 같이 스레드에 독립적이면서, 전역 변수의 역할을 해줄 수 있는 것이 스레드 로컬 저장소(Thread Local Storage, TLS)이다. TLS를 사용할 때에 스레드는 인덱스를 할당하고, 각 스레드에 대한 값을 저장하거나 불러올 수 있게 된다.

TLS를 사용하는 가장 전형적인 방법은 다음과 같다.

1. TlsAlloc 함수를 사용해서 DLL 이나 프로세스에 대한 TLS 인덱스를 할당한다.
2. TLS 인덱스를 사용할 필요가 있는 각각의 스레드는 동적 저장소(dynamic storage)를 할당하고, TlsSetValue 함수를 사용해서 인덱스와 동적 저장소의 포인터를 연결시킨다.
3. 스레드가 저장소에 접근할 필요가 있으면, TlsGetValue 함수에 TLS 인덱스를 지정해서 포인터 값을 불러온다.
4. 더이상 필요하지 않은 동적 저장소는 각각의 스레드에 의해 해제되고, 모든 스레드가 TLS 인덱스의 사용을 끝내면 TlsFree 함수에 의해 TLS 인덱스를 해제한다.

TLS 는 DLL 을 사용할 때에도 유용하다. DllEntryPoint 함수에서 프로세스나 스레드의 초기 TLS 작업을 수행하고, DLL 이 새로운 프로세스와 함께 동작하게 될 때 entry-point 함수가 TlsAlloc 함수를 호출하여 그 프로세스에 대한 TLS 인덱스를 할당하게 한다. 그리고, DLL 이 그 프로세스의 새로운 스레드와 함께 동작하게 될 때 entry-point 함수에서 그 스레드에 대한 동적 저장소를 할당하고, TlsSetValue 함수를 이용해서 데이터를 저장한다. DLL 은 TLS 인덱스를 각 프로세스의 전역 변수에 저장하고, DLL 의 함수들은 TLS 인덱스를 이용해서 TlsGetValue 함수를 호출하면 스레드에서 데이터에 접근할 수 있게 된다.

## 델파이 스레드의 구조

델파이에서 스레드는 TThread 클래스를 상속받아 만드는 새로운 클래스에서 구현하는 것이 보통이다. 일단 상속을 받고 나서 몇 개의 중요한 함수를 작업에 맞도록 수정해야 하는데, 이런 작업에 필수적으로 포함되는 메소드가 Create, Execute, Destroy 메소드이다. Create 메소드에서는 스레드가 필요로 하는 리소스를 할당해 주어야 하며, Destroy 메소드

에서는 사용한 리소스를 해제해 주는 역할을 해야 한다. Execute 메소드가 실제 멀티 쓰레드 어플리케이션을 만들 때 가장 핵심이 되는데, 일단 쓰레드가 생성되고 나면 보통 Execute 메소드가 호출된다. 그러므로, 실제로 실행되어야 하는 모든 작업에 대한 코드가 Execute 메소드에 위치해야 하는 것이다.

TThread 클래스의 Create 메소드는 CreateSuspended 라는 파라미터를 가진다. 이 파라미터는 Boolean 데이터 형으로 쓰레드가 생성될 때 바로 Execute 메소드를 호출할 것인지 여부를 결정한다. 만약 커스텀 Create 메소드를 사용하는 경우라면 보통 이 파라미터는 True 로 설정한다. 이렇게 함으로써 쓰레드가 사용할 모든 리소스를 할당하기 전에 Execute 메소드가 실행되지 않도록 할 수 있게 된다. 그리고 나서, Resume 메소드가 호출되면 쓰레드가 실행된다. 그러므로, 보통 커스텀 Create 메소드를 사용하는 경우라면 일단 사용할 리소스를 할당하고 나서, 맨 마지막에 Resume 메소드를 호출한다.

TThread 클래스에는 Terminated 라는 프로퍼티가 있는데, 이 프로퍼티는 쓰레드가 현재 실행되고 있는지 여부를 나타내는 프로퍼티이다. Execute 메소드는 실행될 때 이 프로퍼티를 항상 검사를 하게 되어 있어서, 이 프로퍼티가 True 로 설정되면 메소드의 실행을 중단한다. 이 프로퍼티에 맞추어 TThread 클래스는 OnTerminate 라는 이벤트를 제공하는데, 이 이벤트는 쓰레드가 종료되고나서 아직 쓰레드 객체가 메모리에서 해제되기 전에 발생한다. 이 이벤트에 적절한 코드를 대입해서 쓰레드가 끝나기 전에 필요한 여러가지 작업을 하게 할 수 있다. 예를 들어, 쓰레드가 계산한 데이터를 실제로 보여주는 등의 작업을 지시할 수 있겠다.

TThread 클래스의 메소드 중에 Execute 메소드 만큼 중요한 메소드가 또 있는데, Synchronize 메소드가 그것이다. 멀티 쓰레드가 실행되다 보면 여러 개의 쓰레드가 동일한 리소스에 동시에 접근할 수가 있는데 이로 인해 문제가 생길 가능성이 있게 된다. 예를 들어, 두 개의 쓰레드가 폼에 데이터를 동시에 쓰려고 하면 하나의 객체를 그리다가, 다른 쓰레드가 이 동작을 방해할 것이다. 이로 인해 에러가 발생할 소지도 있고, 심할 경우 리소스를 빌리는 과정에서 소위 데드락(Dead Lock)이라고 불리는 리소스 잠김 현상이 나타날 수도 있다. 델파이에서는 이런 문제를 해결하기 위해서 문제가 생길 소지가 있는 리소스를 사용하는 코드에서는 반드시 Synchronize 메소드를 통해서 실행하도록 하고 있다. 보통은 델파이의 VCL 라이브러리에 접근할 때 이 방법을 사용한다.

그렇지만 Synchronize 메소드를 사용할 때에는 기본적으로 메시지를 메인 VCL 쓰레드에서 실행시키게 하는 것이므로, 만약에 쓰레드에서 실행되는 대부분의 코드가 Synchronize 메소드에서 실행된다면 이것은 싱글 쓰레드 어플리케이션을 실행시키는 것과 크게 다르지 않게 된다는 것을 명심해야 한다.

그러므로, 리소스가 공유될 때를 제외하고는 Synchronize 메소드에서 지정한 코드를 과도하게 사용하는 것은 피해야 한다.

## 쓰레드의 활용

보통 TThread 클래스를 이용하는 경우는 몇 가지의 통신 디바이스를 통해서 입력을 받는 경우가 가장 전형적인 경우이며, 그 밖에도 파일을 다루는 경우거나 작업을 할 때 작업에 대한 중요도가 차이가 날 때 보다 중요한 작업에는 높은 priority 를 부여하고, 덜 중요한 작업에는 보다 낮은 priority 를 부여하여 작업하면 보다 효율성을 높일 수 있을 것이다. 그 밖에 멀티 쓰레드 프로그래밍을 해야 할 경우로는 다음과 같은 경우가 있다.

1. 초기화가 오래 걸려서 사용자가 오래 기다려야 하는 상황이 발생할 경우
2. 다른 작업을 할 때 틈틈이 각종 리소스를 정리하는 등의 작은 작업을 백 그라운드에서 실행시키고자 할 경우
3. 어플리케이션의 주 기능을 저하시키지 않으면서도 애니메이션 등의 효과 등을 보여주고자 할 경우
4. 다른 작업을 하면서, 백 그라운드로 커다란 쿼리 등을 실행시켜 전체적인 성능을 향상시키고자 할 경우

이번에는 쓰레드를 사용할 때 반드시 알아야 할 주의사항 몇 가지를 알아보도록 하자.

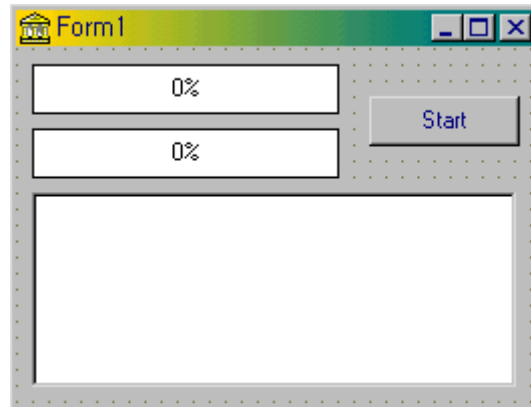
1. 너무 많은 쓰레드를 동시에 작동시키면 시스템의 속도가 심각하게 저하된다. 보통 단일 프로세서 시스템일 경우 프로세스당 16 쓰레드를 넘지 않는 것이 좋다.
2. 여러 개의 쓰레드가 동일한 리소스를 수정할 경우 반드시 동기화를 시켜야 한다.
3. VCL 컴포넌트나 폼을 업데이트하는 메소드는 메인 VCL 쓰레드 내에서 호출되어야 한다.

앞에서도 간단히 설명했지만 TThread 의 주요 프로퍼티로는 쓰레드의 중요도를 결정하는 Priority, 쓰레드의 실행을 Resume 메소드가 실행될 때까지 일시 정지시킬 수 있는 Suspended, 쓰레드의 실행을 중지시킬 때 사용하는 Terminated 프로퍼티 등이 있다. Suspended 프로퍼티와 Terminated 프로퍼티를 설정하는 것은 각각 Suspend, Terminate 메소드를 실행하는 것과 같은 역할을 하게 된다.

## 첫번째 예제

쓰레드 객체를 사용할 때에는 일단 TThread 클래스에서 새로운 클래스를 상속받고, 기본적으로 Execute 메소드를 override 하는 것이 가장 기본적인 방법이다. 또한 꼭 기억해야 할 것은 VCL 객체에 접근할 때에는 Synchronize 메소드를 사용해서 메인 쓰레드에 접근해야 한다는 것이다. 우리의 첫번째 예제는 파일을 복사하는 과정을 TGauge 컴포넌트에 그

려주는 루틴을 쓰레드 객체를 이용해서 제작하는 것이다.  
 일단 다음 그림과 같이 폼을 디자인 하도록 하자.



여기서 Start 버튼(Button1)을 클릭하면 같은 파일을 다른 파일로 복사하는 두 개의 쓰레드가 실행되고, 각각의 쓰레드의 진행과정을 TGauge 컴포넌트에 그려 준다. 그리고, 각각의 쓰레드가 끝나면 TMemo 컴포넌트에 몇 번 쓰레드가 끝났다는 메시지가 나타나도록 해보자.

이렇게 파일을 조작하거나, 인쇄, 통신을 할 때에 멀티 쓰레드 프로그래밍이 많이 사용된다. 먼저 복사를 하는 쓰레드의 이름을 TCopyThread 라고 하자. 그리고, 실제 복사를 할 때 약간의 시간이 지나야 실행시키는 재미가 있으므로, 조금은 커다란 파일을 복사해 보자. 여기서는 텔파이 4 의 Help 디렉토리에 있는 del4vcl.hlp 파일을 선택했다. 이 파일의 크기는 10MB 가 넘기 때문에 이번 어플리케이션에 비교적 적절하게 사용될 수 있을 것이다. 그리고, 각 쓰레드가 끝났을 때 적용할 ThreadDied 프로시저를 다음과 같이 TForm1 의 private 섹션에 선언한다.

```
private
  procedure ThreadDied(Sender: TObject);
```

그리고, Button1 의 OnClick 이벤트를 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  T1, T2: TCopyThread;
begin
  Memo1.Clear;
  T1 := TCopyThread.Create('C:\Program Files\Borland\Delphi4\Help\del4vcl.hlp',
    'C:\Temp\test1.hlp', 1);
```



```

T2 := TCopyThread.Create('C:\Program Files\Borland\Delphi4\Help\del4vcl.hlp',
    'C:\Temp\test2.hlp', 2);
T1.OnTerminate := ThreadDied;
T2.OnTerminate := ThreadDied;
T1.Resume;
T2.Resume;
Button1.Enabled := False;
end;

```

눈치 빠른 독자 들은 TCopyThread 의 Create 메소드를 호출하는 것을 보면 파라미터가 3 개 있는 것으로 보아 커스텀 Create 메소드라는 것을 알 수 있을 것이다. 또한 뒤쪽에 각 쓰레드에 대해 Resume 메소드를 호출한다는 것은 쓰레드가 생성될 때 CreateSuspended 가 True 로 설정되었다는 것을 의미한다.

TCopyThread 의 Create 메소드는 첫번째 파라미터로 읽어 올 파일 이름, 두번째 파라미터로 읽어 쓸 파일 이름을 지정하며, 세번째 파라미터는 실행되는 쓰레드가 몇 번 쓰레드인지 나타내려고 추가한 것이다. 이 값을 이용해서 몇 번 쓰레드가 끝났다는 메시지를 메모에 기록한다.

각 쓰레드의 OnTerminate 이벤트 핸들러로 ThreadDied 프로시저를 대입하였다. 그러면 ThreadDied 프로시저를 구현해보자.

```

procedure TForm1.ThreadDied(Sender: TObject);
begin
    Memo1.Lines.Add(Format('%d 번 쓰레드 종료.', [(Sender as TCopyThread).ReturnValue]));
end;

```

이제는 실제로 TCopyThread 클래스를 선언하고 구현할 차례이다. type 선언문에 다음과 같은 클래스 선언문을 추가한다.

```

TCopyThread = class(TThread)
private
    FSource, FTarget: string;
    FSize, BytesCopied: integer;
protected
    procedure Execute; override;
public
    procedure DisplayProgress;

```

```

    constructor Create(const Source, Target: string; Return: integer);
    property ReturnValue;
end;

```

그다지 어렵지는 않지만 간단하게 선언부를 설명하면, Create 메소드에서 읽어 올 파일과 기록할 파일의 이름과 쓰레드의 번호를 얻어온다. Button1 의 OnClick 이벤트 핸들러를 기준으로 하면 1, 2 가 각각의 쓰레드 번호가 된다. 그리고 Execute 메소드를 override 하는데, 이 메소드에서 private 섹션에서 정의한 필드 값을 이용해서 실제 쓰레드를 수행한다. 이때 TGauge 컴포넌트를 사용하는 메소드가 DisplayProgress 인데 이와 같이 대부분의 VCL 컴포넌트에 접근하는 경우에는 Synchronize 메소드를 사용해서 호출해야 하므로, 따로 선언해 준다.

그럼 먼저 Create 메소드를 구현해 보자

```

constructor TCopyThread.Create(const Source, Target: String; Return: Integer);
begin
    inherited Create(True);
    FreeOnTerminate := True;
    FSource := Source;
    FTarget := Target;
    ReturnValue := Return;
end;

```

단순한 코드 이므로 별 설명이 필요 없을 것으로 생각된다. FreeOnTerminate 프로퍼티는 True 로 설정되었을 때 쓰레드가 종료됨과 동시에 메모리에서 객체가 해제되게 하는 것이다. 보통 True 로 설정하는 경우가 많다.

그러면, TGauge 에 표시하는 DisplayProgress 메소드를 구현하자.

```

procedure TCopyThread.DisplayProgress;
begin
    if ReturnValue = 1 then
        Form1.Gauge1.Progress := Round(BytesCopied/FSize*100.0)
    else
        Form1.Gauge2.Progress := Round(BytesCopied/FSize*100.0);
end;

```

간단히 설명하면 ReturnValue 가 1, 즉 쓰레드를 생성할 때 세번째 파라미터를 1 로 설정한

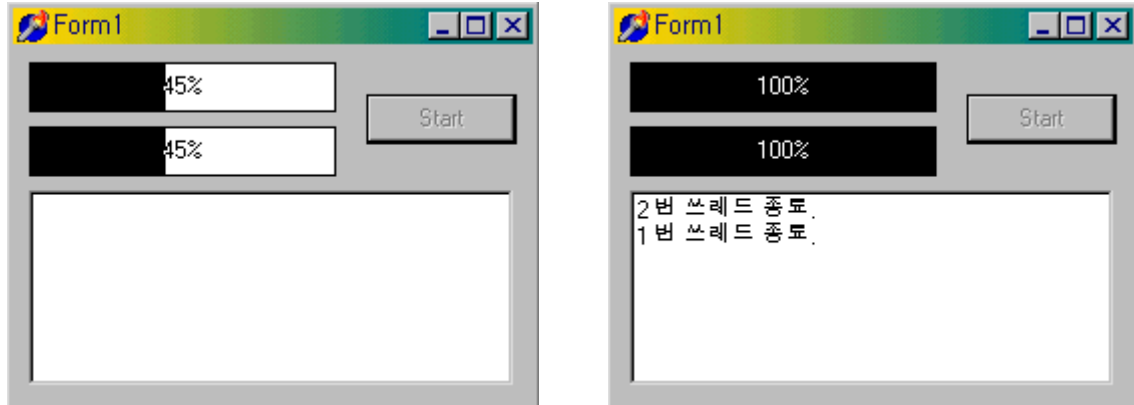
Copy 쓰레드의 진행상황은 Gauge1 에 표시되고, 다른 쓰레드는 Gauge2 에 표시하라는 이야기이다. 이때 진행된 상황의 계산은 파일의 실제 사이즈와 복사된 바이트를 나누어서 표시하면 된다. 마지막으로 Execute 메소드를 구현하면 첫번째 예제가 완성된다.

```
procedure TCopyThread.Execute;
var
    FIn, FOut: TFileStream;
    ChunkSize: Integer;
begin
    FIn := TFileStream.Create(FSource, fmOpenRead or fmShareDenyWrite);
    try
        FSize := FIn.Size;
        BytesCopied := 0;
        FOut := TFileStream.Create(FTarget, fmCreate);
        try
            while BytesCopied < FSize do
                begin
                    ChunkSize := FSize - BytesCopied;
                    if ChunkSize > $8000 then ChunkSize := $800;
                    Inc(BytesCopied, FOut.CopyFrom(FIn, ChunkSize));
                    Synchronize(DisplayProgress);
                end;
            finally
                FOut.Free;
            end;
        finally
            FIn.Free;
        end;
    end;
end;
```

실제 복사하는 루틴은 TFileStream 클래스를 이용하였다. 비교적 사용하기 편리한 루틴이므로 파일을 다루는 어플리케이션을 만들 때 다양하게 응용할 수 있을 것이다. 눈 여겨 보아야 할 부분은 FSize 값에 먼저 FIn 파일 스트림 객체를 생성하면서 원본 파일의 크기를 대입하고, 실제로 복사를 블록 단위로 하면서 BytesCopied 값을 증가시키는 부분이다. 이렇게 한 블록이 복사될 때 마다 Synchronize(DisplayProgress)를 호출하여 복사가 진행되는 상황을 한 눈에 파악할 수 있게 된다. Synchronize 메소드의 파라미터에는 이와 같이

직접 프로시저의 이름을 대입해서 사용하면 된다.

다음 그림들은 이 예제의 실제 실행화면 들이다.



## 델파이 4 에서 향상된 스레드 관리

델파이 4 는 멀티 스레드 어플리케이션의 개발을 돕기 위해 몇 가지 새로운 클래스를 제공하고 있다. TCriticalSection 클래스는 멀티 스레드 어플리케이션에서 하나의 스레드의 실행을 막을 수 있는 클래스이다. 이 클래스는 특정 작업이 끝나기 전에 다른 스레드가 실행되면 안되는 경우, 이를 보호할 수 있다. 그렇지만, TCriticalSection 은 다른 모든 스레드에서 실행되지 않도록 하는 것이므로, 함부로 남용하면 실행 속도를 심각하게 저해할 수도 있다.

이 클래스를 사용하려면 일단 CriticalSection 객체를 생성하고, 보호할 코드의 앞에서 Enter 또는 Acquire 메소드를 호출한다. 그리고, 코드의 끝 부분에 Leave 또는 Release 메소드를 호출한다.

또 하나의 새로운 클래스는 TThreadList 클래스이다. 이 클래스는 멀티 스레드 환경에서도 사용할 수 있는 TList 클래스라고 생각하면 된다. 각각의 TThreadList 객체는 TList 객체를 관리하게 되는데, TList 의 아이템을 사용할 때에는 LockList 메소드를 사용하면 사용할 TList 객체를 반환하며, 사용하고 나서 UnLock 을 호출하면 된다. 간단한 사용 방법은 다음과 같다.

```
with MyThreadList.LockList do
begin
  try
    for X := 0 to Count-1 do
      Something(Items[X]);
    finally
      MyThreadList.UnlockList;
    end;
```

## 멀티 쓰레드 데이터베이스 어플리케이션

멀티 쓰레드 데이터베이스 어플리케이션을 제작하는 데에는 몇가지 고려해야 할 점이 있는데 먼저 여기에 대해서 알아보자.

1. 데이터베이스에 접근하는 각각의 쓰레드는 자신의 세션을 가져야 한다.

BDE 를 통해 데이터베이스에 접근할 때에는 세션을 통해서 연결하게 되어 있다. 각각의 델파이 프로그램은 명시적으로 적어주지 않더라도 기본적인 디폴트 세션이 자동적으로 사용되며, 모든 데이터는 이를 통해서 접근하게 되어 있다. 그런데, 멀티 쓰레드를 사용할 경우 동시에 여러 개의 데이터베이스 작업이 이루어질 수 있다. 이 때 TSession 객체가 하나 밖에 없다면 병목 현상이 일어날 수 밖에 없는 것이다. 이러한 현상의 해결책은 각각의 쓰레드에 자신의 세션을 가지게 하는 것이다. 기본적으로 BDE 는 멀티 쓰레드에 영향을 받지 않도록 디자인 되어 있기 때문에 여러 개의 세션을 통해서 동시에 데이터에 접근하는 것을 허용한다.

2. TQuery 나 TTable 을 사용할 때에는 Synchronize 메소드를 사용하지 않는다.

앞에서 언급했지만 보통 그래픽을 사용하는 VCL 객체를 이용할 때에는 Synchronize 메소드를 사용한다. 그런데, TQuery 객체가 Synchronize 메소드를 통해서 접근될 때에는 쿼리가 실행될 동안 델파이의 메인 쓰레드가 실행을 멈추게 된다. 이렇게 되면 멀티 쓰레드 어플리케이션을 제작한 의미가 거의 없게 되어 버린다.

BDE 는 기본적으로 멀티 쓰레딩을 지원하므로 Synchronize 를 사용하지 않고 TQuery, TTable 에 접근해도 된다.

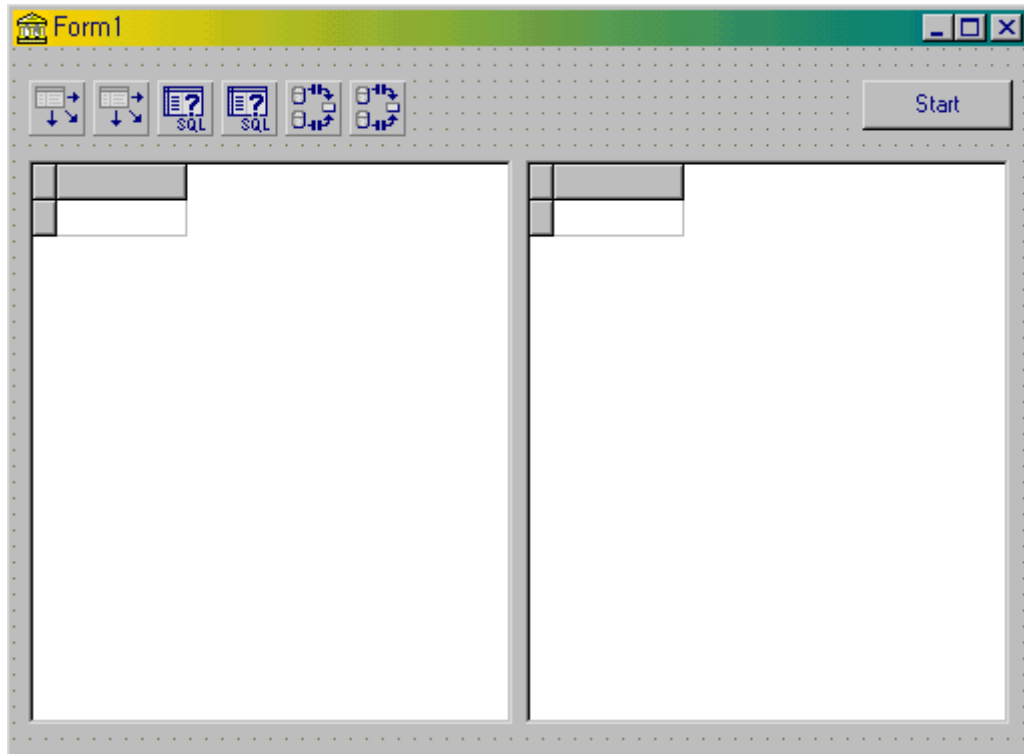
3. TQuery 와 TTable 객체가 Open 되어 있을 때에는 TDataSource 와 연결하지 않는다.

TDataSet 가 Open 되어 있을 때 TDataSource 와 연결되면 델파이가 연결되어 있는 데이터 어웨어 컨트롤에 결과를 보여주려고 시도하게 된다. 그러므로, 이럴 때에는 TDataSet 를 Open 하기 전에 Synchronize 메소드를 사용해서 TDataSet 와 TDataSource 의 연결을 일단 끊어야 한다. 일단 TDataSet 가 Open 되면 TDataSource 와 Synchronize 메소드를 사용해서 연결한다.

## 백그라운드 쿼리 실행 예제

멀티 쓰레드를 이용한 데이터베이스 어플리케이션을 구현하는 아주 간단한 예제를 제작해 보자. 이 예제는 사실 그다지 바람직하지 않은 구현 부분도 있지만, 동시에 두 개의 데이터베이스에 접근해서 쿼리를 실행하는 것을 직접 눈으로 확인할 수 있어서 멀티 쓰레드의 장점을 음미하는데에는 도움이 될 것이다.

먼저, 폼 위에 다음 그림과 같이 TDataSource, TQuery, TSession, TDBGrid 컴포넌트를 각각 두 개씩과 버튼을 하나 올려 놓는다.



그리고, DataSource1, DataSource2 객체의 DataSet 프로퍼티를 각각 Query1, Query2 로 설정하고 DBGrid1, DBGrid2 객체의 DataSource 프로퍼티를 각각 DataSource1, DataSource2 로 설정한다. 또한, Query1 과 Query2 의 SessionName 프로퍼티를 각각 Session1 과 Session2 의 SessionName 프로퍼티 값으로 설정한다. 여기서는 각각 'Sample1', 'Sample2'로 설정하였다. 앞서서도 잠시 설명했지만 BDE 가 멀티 쓰레드로 실행되려면 이렇게 각각의 쓰레드에 대해 다른 TSession 객체가 정의되어 있어야 한다.

마지막으로 실행할 쿼리를 입력할 차례인데, 일단 DatabaseName 프로퍼티를 DBDEMOS 로 설정한다. Query1 의 SQL 문장으로는 Employee.db 에서 레코드를 가져오도록 'SELECT \* FROM "employee.db".', Query2 는 Customer.db 에서 레코드를 가져오도록 'SELECT \* FROM "customer.db".'로 설정한다.

이제 쓰레드 클래스를 선언하도록 하자. type 선언문에 다음과 같이 선언한다.

```

TQThread = class(TThread)
private
    FQuery: TQuery;
protected
    procedure Execute; override;
public
    constructor Create(Query: TQuery);
end;

```

이 쓰레드 클래스의 구현은 아주 간단하게 한다. Create 메소드에서 파라미터로 넘어온 Query 파라미터를 FQuery 필드에 저장하는 정도로 구현하고, Execute 메소드에서 FQuery 필드를 Open 하는 것으로 충분한다.

```

constructor TQThread.Create(Query: TQuery);
begin
    inherited Create(True);
    FQuery := Query;
    FreeOnTerminate := True;
    Resume;
end;

procedure TQThread.Execute;
begin
    FQuery.Open;
end;

```

마지막으로 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    TQThread.Create(Query1);
    TQThread.Create(Query2);
end;

```

이제 프로젝트를 실행시켜 보자. 워낙 빨리 실행되기 때문에 잘 관찰하기는 어렵겠지만 다음 그림과 같이 두 개의 데이터베이스의 레코드가 DBGrid 에 동시에 뿌려진 것을 관찰할

수 있을 것이다.

LastName	FirstName
Nelson	Roberto
Young	Bruce
Lambert	Kim
Johnson	Leslie
Forest	Phil
Weston	K. J.
Lee	Terri
Hall	Stewart
Young	Katherine
Papadopoulos	Chris
Fisher	Pete
Bennet	Ann
De Souza	Roger

Company	Address
Kauai Dive Shoppe	4-97
Unisco	PO E
Sight Diver	1 Ne
Cayman Divers World Unlimited	PO E
Tom Sawyer Diving Centre	632-
Blue Jack Aqua Center	23-7
VIP Divers Club	32 M
Ocean Paradise	PO E
Fantastique Aquatica	Z32
Marmot Divers Club	872
The Depth Charge	1524
Blue Sports	203
Makai SCUBA Club	PO E

## Win32 API 를 이용한 멀티 쓰레드 구현

멀티 쓰레드 프로그래밍을 하면서도 델파이의 TThread 객체를 사용하기가 어려운 경우가 있다. 예를 들어, 아주 간단한 프로시저나 함수를 루프에 넣어서 돌릴 경우에 TThread 객체를 생성해서 쓰기가 좀 아깝다고 생각될 수가 있다. 이럴 때에는 Win32 API 를 직접 호출해서 멀티 쓰레드 프로그래밍을 하는 것이 낫다.

API 를 이용한 멀티 쓰레드 프로그래밍을 할 때에는 먼저 쓰레드 자체를 생성하고, 쓰레드의 엔트리 포인트가 되는 함수를 작성한 후, 이를 쓰레드에게 알려주어 실행하게 하는 절차를 밟는다.

일반적인 함수의 구현과는 달리 멀티 쓰레드를 구현하는 함수에는 다음과 같은 제한점을 가지고 있다. 기본적으로 파라미터를 Pointer 형으로 하나만 가져야 하고, 반드시 LongInt 형의 정수값을 반환해야 한다. 그리고, 표준 윈도우에 의해서 파라미터가 전달되므로 stdcall 로 선언되어야 한다. 전형적인 쓰레드 함수의 선언문은 다음과 같다.

```
function MyThreadFunc(Ptr: Pointer): LongInt; stdcall;
```

그러면, 쓰레드를 생성하는 API 함수인 CreateThread 함수에 대해서 알아보자. 이 함수의

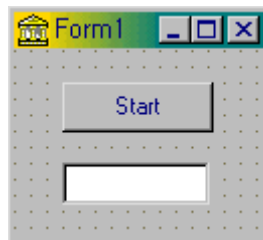


선언문은 다음과 같다.

```
function CreateThread(lpThreadAttributes: Pointer; dwStackSize: DWORD;  
    lpStartAddress: TFNThreadStartRoutine; lpParameter: Pointer; dwCreationFlags:  
    DWORD; var lpThreaded: DWORD): THandle; stdcall;
```

lpThreadAttributes 파라미터는 쓰레드의 보안 특성을 설정하는 주소를 지정하는 것으로 대개 nil 로 설정하게 되며, dwStackSize 파라미터는 쓰레드에 대한 스택의 크기를 지정하는 것으로 보통 0 으로 설정한다. lpStartAddress 파라미터가 가장 중요한 것으로 멀티 쓰레드로 실행될 함수의 주소를 넘겨 준다. lpParameter 파라미터에는 쓰레드에서 사용할 변수의 주소를 넘겨주도록 되어 있는데, 이를 이용해도 되지만 전역 변수를 이용하는 방법이 더 편리하므로 대개는 nil 로 설정한다. dwCreationFlags 파라미터는 대개 0 으로 설정하며, 마지막으로 lpThreaded 파라미터에는 생성한 쓰레드의 ThreadID 의 참조값이 넘어 오게 된다.

그러면, 이를 이용해서 실제로 멀티 쓰레드를 구현하는 예제를 하나 제작해 보도록 하자. 예제는 아주 단순하게 'Start'라는 버튼을 클릭하면 1 부터 1000 까지의 정수를 하나씩 에디트 컨트롤에 표시하는 예제이다. 먼저 다음 그림과 같이 폼을 디자인한다.



그리고, 앞에서 간단히 설명한 데로 쓰레드 함수로 적당한 형태의 함수를 다음과 같이 구현한다.

```
function MyThreadFunc(P: Pointer): LongInt; stdcall;  
  
var  
    Temp: String;  
    i: Integer;  
  
begin  
    EnableWindow(Form1.Button1.Handle, False);  
    for i := 1 to 1000 do  
        begin  
            Temp := IntToStr(i);
```

```

    SendMessage(Form1.Edit1.Handle, WM_SETTEXT, 0, LongInt(PChar(Temp)));
end;
EnableWindow(Form1.Button1.Handle, True);
end;

```

아주 간단한 함수인데, 다시 한번 강조하지만 Pointer 형인 파라미터 하나만 가진 함수로 LongInt 형을 반환하며 stdcall 로 선언한다.

EnableWindow 와 SendMessage 를 쓴 이유는 VCL 코드를 직접 사용하면 VCL 클래스는 멀티 스레드와 충돌을 일으킬 수 있기 때문에, Button1 과 Edit1 컨트롤의 Handle 을 직접 이용해서 API 로 처리한 것이다.

EnableWindow 함수는 핸들을 넘겨준 컨트롤을 사용할 수 있게 한 것으로 VCL 컨트롤의 Enable 프로퍼티를 설정하는 것과 같은 역할을 하게 된다. 1 부터 1000 까지 숫자를 에디트 컨트롤에 표시하고, 그동안 Button1 을 사용할 수 없게 하고 이 작업이 끝나면 Button1 을 사용할 수 있게 하는 것이다.

WM\_SEXTEXT 메시지는 컨트롤에 텍스트를 설정하는 것으로, VCL 컴포넌트의 Text 프로퍼티를 사용하는 것과 동일한 역할을 한다. 이때 SendMessage 함수의 파라미터로는 LongInt 데이터 형을 사용하기 때문에 형변환이 필요하다.

이 함수를 작동시키기 위한 Button1 의 OnClick 이벤트 핸들러는 다음과 같이 작성하면 된다.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    Thread: THandle;
    ThrID: DWORD;
begin
    Thread := CreateThread(nil, 0, @MyThreadFunc, nil, 0, ThrID);
    if Thread = 0 then ShowMessage('스레드가 생성되지 않았습니다 !');
end;

```

어렵지 않은 코드 이므로, 금방 이해할 수 있을 것이다.

프로젝트를 실행해 보면 단순하게 멀티 스레드를 구현할 수 있다는 것을 알 수 있을 것이다.

## 쓰레드 대기 처리

멀티 스레드를 이용해서 어플리케이션을 제작하다 보면 스레드가 모두 끝난 후에 특정 작업을 해야 할 경우가 생긴다.

이를 해결하기 위한 방법으로 가장 손쉬운 것은 Boolean 형의 전역 변수를 하나 이용해서 쓰레드가 끝날 때 이를 True 로 설정하여 쓰레드가 종료했는지 여부를 알아보고, 종료되지 않았다면 Application.ProcessMessages 메소드를 호출하여 다른 작업을 계속할 수 있도록 하는 것이다.

Application.ProcessMessages 메소드를 이용하면 프로그램이 메시지를 받고 처리할 수 있게 되기는 하지만, 상당한 CPU 시간의 소모가 있게 된다. 특히, 여러 개의 쓰레드를 동시에 실행하였는데 이들이 모두 끝났는지를 알아보아야 한다면 문제는 더 심각해진다.

이를 해결하기 위한 가장 좋은 방법은 대기를 수행하는 대기 쓰레드(wait thread)를 따로 작성하는 것이다. 이러한 대기 쓰레드에는 CPU 시간과 시스템 리소스를 거의 소모하지 않는 효과적인 Win32 API 함수를 호출하여 소기의 목적을 달성할 수 있다.

WaitForSingleObject 와 WaitForMultipleObjects API 함수가 이러한 역할을 수행하는 Win32 API 함수이다. 기본적으로 윈도우에서 모든 객체가 생성될 때 이 객체가 현재 사용 중이거나 활성화 되어 있을 때에는 이를 non-signaled 상태에 있다고 하며, 사용 중이 아닐 때에는 signaled 상태에 있다고 한다. 앞의 두가지 API 함수는 지정된 객체 들이 signaled 상태에 들어갈 때까지 대기하는 역할을 해주는 함수이다. 이들 함수의 장점은 CPU 시간을 거의 소모하지 않는 아주 효과적인 대기 상태로 들어간다는 것이다.

WaitForSingleObject 함수는 2 개의 파라미터를 가진다. 대기할 쓰레드의 핸들을 지정하는 THandle 형의 파라미터와 쓰레드가 종료되지 않을 경우에 최고 한도로 몇 millisecond 까지 기다릴 것인지를 설정하게 되어 있다.

보통은 지정한 쓰레드가 종료될 때까지 무조건 기다리는 INFINITE 상수를 설정하게 된다. 간단한 사용 예제 코드는 다음과 같다.

```
WaitForSingleObject(MyThread.Handle, INFINITE);
```

WaitForMultipleObjects 함수는 이보다 조금 더 많은 파라미터를 가진다. 선언문은 다음과 같다.

```
function WaitForMultipleObjects(nCount: DWORD; lpHandles: PWOHandleArray;  
    bWaitAll: BOOL; dwMilliseconds: DWORD): DWORD; stdcall;
```

첫번째 파라미터인 nCount 는 DWORD 형으로 쓰레드의 핸들 배열에 있는 핸들의 수를 지정하며, lpHandles 파라미터는 쓰레드 객체의 핸들 배열의 주소를 설정하도록 되어 있다. bWaitAll 은 모든 객체가 signaled 상태에 들어갈 때까지 대기할 것인지 여부를 설정하는 파라미터이며 마지막으로 dwMilliseconds 파라미터는 대기할 시간을 지정하는 것으로 보통 INFINITE 로 설정한다.

이 함수를 사용할 때 가장 특기할 만한 것은 핸들 배열을 사용하는 것이다. 그러면, 실제

로 간단한 예제를 하나 만들어 보자.

폼위에 TLabel 컴포넌트와 TButton 컴포넌트를 하나씩 올려 놓고, 버튼을 클릭하면 ‘대기’라는 글을 TLabel 에 보여주다가 모든 스레드가 종료되면 ‘종료’를 표시하게 한다.

먼저 사용할 스레드를 TTestThread, 대기 스레드를 TWaitThread 라고 명명하고, type 선언부분에 다음과 같이 스레드를 정의한다.

```
TTestThread = class(TThread)
private
protected
    procedure Execute; override;
end;
```

```
TWaitThread = class(TThread)
private
    procedure UpdateLabel;
protected
    procedure Execute; override;
end;
```

TTestThread 를 구현해 보자. 단지 Sleep 기능을 이용하는 것으로 실행을 대신하도록 한다.

```
procedure TTestThread.Execute;
begin
    FreeOnTerminate := True;
    Sleep(10000);
end;
```

이제 핸들 배열과 WaitForMultipleObjects API 함수를 사용해서 대기 스레드를 구현한다. 이 부분이 이번 섹션의 핵심 부분이라고 할 수 있다.

```
procedure TWaitThread.UpdateLabel;
begin
    Form1.Label1.Caption := '종료';
end;
```

```

procedure TWaitThread.Execute;
var
    HandleArray: array[0..4] of THandle;
    ThreadArray: array[0..4] of TTestThread;
    i: Integer;
begin
    FreeOnTerminate := True;
    for i := 0 to 4 do
        begin
            ThreadArray[i] := TTestThread.Create(False);
            HandleArray[i] := ThreadArray[i].Handle;
            Sleep(1000);    //쓰레드가 동시에 생성되지 않도록 ...
        end;
    WaitForMultipleObjects(5, @HandleArray, True, INFINITE);
    Synchronize(UpdateLabel);
end;

```

그러면 이제 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Caption := '대기';
    TWaitThread.Create(False);
end;

```

비교적 간단한 예제 이지만, 사용법을 익히는 데에는 큰 무리가 없었을 것으로 생각한다.

## 임계 섹션(Critical Sections)의 활용

멀티 쓰레드 어플리케이션을 제작하다 보면 여러 개의 쓰레드가 동시에 파일과 같은 리소스에 접근하는 경우가 있게 된다. 이런 경우에 데이터가 깨지거나, 심할 경우 시스템이 멈추는 등의 부작용이 있을 수가 있으므로 이런 상황을 피하도록 해야 한다. 이를 위해서는 하나의 쓰레드가 점령하고 있는 리소스에는 다른 쓰레드가 접근할 수 없도록 할 필요가 있는데, 이럴 때 사용하게 되는 것이 임계 섹션(Critical Section)이다.

임계 섹션을 간단하게 설명하면 여러 개의 쓰레드가 자신의 차선을 지키면서 동시에 도로를 달려오다가, 1 차선으로 좁아지는 병목 구간이 생긴 것으로 생각하면 쉽게 이해할 수 있다.

즉, 리소스를 사용하는 곳에서는 하나의 스레드 만이 통과가 가능한 것이다. 이러한 구간을 Win32 API 인 EnterCriticalSection 과 LeaveCriticalSection 함수의 블록으로 설정해서 사용하면 된다. 이렇게 임계 섹션에 들어가려고 대기하고 있는 각각의 스레드들은 CPU 시간을 전혀 소모하지 않으므로 효과적이다.

텔파이 3 에서는 이들 API 함수를 SyncObjs.pas 유닛에 TCriticalSection 클래스로 구현해 놓았는데, 실제로 구현한 부분을 살펴 보면 Win32 API 를 직접 사용하는 것과 크게 다르지 않으므로 여기에서는 Win32 API 를 직접 사용하는 방법에 대해서 먼저 알아보고, TCriticalSection 클래스를 사용할 때에는 어떻게 변경 되는지 공부하도록 하자.

임계 섹션을 구현하기 위해서는 먼저 메모리에 임계 섹션을 설정해야 한다. 이를 위해서 사용하는 API 함수가 InitializeCriticalSection 이라는 프로시저로 이 프로시저의 선언부는 다음과 같다.

```
procedure InitializeCriticalSection(var lpCriticalSection: TRTLCriticalSection); stdcall;
```

파라미터로 TRTLCriticalSection 형의 변수를 가지게 되어 있는데, 이 데이터 형은 정의된 임계 섹션에 대한 정보를 가지고 있는 레코드 형이다. 그러나, 실제로 이 레코드에 직접 접근하는 경우는 거의 없다.

이렇게 InitializeCriticalSection 프로시저를 이용해서 임계 섹션을 초기화 했으면 나중에는 결국 DeleteCriticalSection 프로시저를 이용해서 사용한 리소스를 해제해 주어야 한다. 이 프로시저 역시 InitializeCriticalSection 과 마찬가지로 TRTLCriticalSection 데이터 형의 변수를 파라미터로 가진다.

보통 이들 프로시저는 각 유닛의 initialization 과 finalization 섹션에 설정해서 사용하게 된다. TRTLCriticalSection 형의 변수 CritSect 를 사용한다고 가정하면 다음과 같이 각 유닛에 설정해서 사용한다.

initialization

```
InitializeCriticalSection(CritSect);
```

finalization

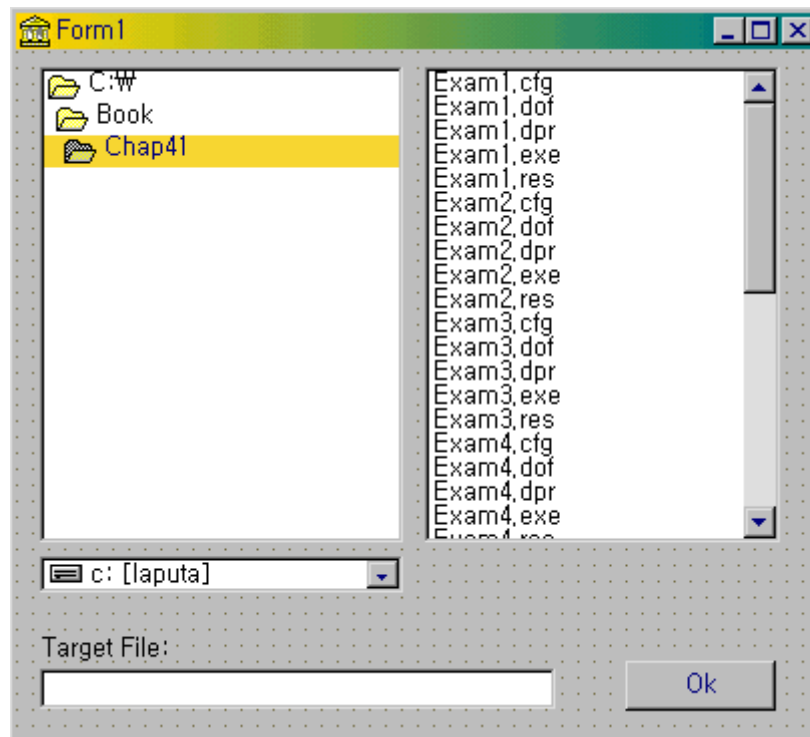
```
DeleteCriticalSection(CritSect);
```

임계 섹션과 스레드 대기 처리 방법을 동시에 익히기 위해서 예제를 하나 만들어 보자. 이번 예제는 두개의 파일을 붙여서 하나의 파일로 합쳐주는 프로그램이다. 예를 들어, 텍스트 파일 같은 경우에 파일을 붙여서 하나로 만들어야할 경우가 생기는데 이를 멀티 스레드 기법과 임계 섹션을 이용해서 구현하는 것이다.

이를 위해서 폼에 드라이브와 디렉토리, 파일을 선택할 수 있도록 TDriveComboBox,

TDirectoryListBox, TFileListBox 컴포넌트를 올려 놓고, TEdit 컨트롤에 목적 파일을 적어넣을 수 있도록 한다. 그리고, 작업을 실행시킬 버튼을 추가한다.

폼의 모습은 다음과 같다.



그러면, 실제로 쓰레드가 어떻게 작동할 것인지 생각해보자. 두개의 쓰레드가 소스 파일에 접근하는 것이 허용된다. 그렇지만 하나의 목적 파일에 쓸 때에는 오직 하나의 쓰레드만 접근할 수 있어야 한다. 그리고, 이렇게 파일에 쓰고 있는 동안에는 그 작업이 끝날 때까지 다른 쓰레드는 대기하도록 해야 한다. 그러므로, 파일에 쓰는 작업을 임계 섹션 블록으로 지정하면 동시에 접근하는 일은 없어진다. 이렇게 멀티 쓰레드를 사용하면 하나의 쓰레드가 파일에 쓰고 있을 때, 다른 쓰레드는 파일의 내용을 읽어오는 등의 작업을 동시에 할 수 있다.

실제로 이 예제에서 사용할 쓰레드는 파일을 복사하는데 사용할 TCopyThread 와 두 개의 TCopyThread 가 실행될 때 이들이 하나의 파일로 복사될 때 사용자 인터페이스가 멈추기 않기 위해서 이들 쓰레드가 완료될 때까지 대기하는 TMasterThread 로 구성된다.

TCopyThread 에서는 임계 섹션을 사용해서 파일에 쓰기 작업을 할 때 하나의 쓰레드만 접근하도록 할 것이며, TMasterThread 에서는 두 개의 TCopyThread 핸들 배열을 이용해서 쓰레드 대기 처리를 할 것이다.

먼저 type 선언문에 TMasterThread 와 TCopyThread 를 다음과 같이 선언하고, 전역변수로 사용할 CritSect, Files, NoFile 변수를 var 절에 추가한다.

```

TMasterThread = class(TThread)
private
    FTarget : String;
protected
    procedure Execute; override;
public
    constructor Create(TargetFile: String);
    function WaitForMultipleThreads(const ThreadArray: array of TThread;
        TimeOutVal: DWORD): Word;
end;

```

```

TCopyThread = class(TThread)
private
    FTarget: String;
protected
    procedure Execute; override;
public
    constructor Create(Target: String);
end;

```

```

var
    Form1: TForm1;
    CritSect: TRTLCriticalSection;
    Files: TStrings;
    NoFile: Boolean;

```

앞에서 NoFile 전역 변수는 파일 리스트 박스에서 파일이 선택되었는지 여부와 쓰레드에서 작업을 수행한 후 이 작업이 완료되었는지 등을 검사할 때 사용하게 된다. 이 변수와 임계 섹션을 초기화하기 위해서 다음과 같이 initialization, finalization 섹션의 코드를 작성한다.

```

initialization
    InitializeCriticalSection(CritSect);
    NoFile := False;

```

```

finalization
    DeleteCriticalSection(CritSect);

```



TMasterThread 를 구현해보자. 이 쓰레드의 Create 메소드를 다음과 같이 구현해서, 두 개의 파일을 합칠 파일의 이름이 TargetFile 파라미터로 넘어오면 이를 FTarget 필드에 저장한다.

```
constructor TMasterThread.Create(TargetFile: String);
begin
    FTarget := TargetFile;
    FreeOnTerminate := True;
    inherited Create(False);
end;
```

그리고, TCopyThread 를 이용해서 작업을 수행하고 이들 작업이 끝날 때까지 기다리는 Execute 메소드는 다음과 같이 구현한다.

```
procedure TMasterThread.Execute;
var
    Thread1, Thread2: TCopyThread;
    i: Integer;
begin
    Thread1 := TCopyThread.Create(FTarget);
    Thread2 := TCopyThread.Create(FTarget);
    WaitForMultipleThreads([Thread1, Thread2], INFINITE);
end;
```

즉, 쓰레드를 생성하고 이들을 직접 내부적으로 정의한 WaitForMultipleThrad s 메소드를 이용해서 파라미터로 넘겨주면 이들 쓰레드의 실행이 모두 완료될 때까지 대기한다. 이를 적당하게 변경하면 두 개의 파일 뿐만 아니라 여러 개의 파일을 붙일 수 있도록 확장할 수 있을 것이다.

TMasterThread 의 핵심적인 부분인 WaitForMultipleThreads 메소드는 다음과 같이 구현된다.

```
function TMasterThread.WaitForMultipleThreads(const ThreadArray: array of TThread;
    TimeOutVal: DWORD): Word;
var
    Handles: TWOHandleArray;
```

```

i: Integer;
begin
  for i := 0 to High(ThreadArray) do
    Handles[i] := ThreadArray[i].Handle;
    Result := WaitForMultipleObjects(High(ThreadArray) + 1, @Handles, True,
      TimeOutVal)
  end;
end;

```

이 메소드는 WaitForMultipleObjects API 함수를 사용하기 쉽도록 변경한 것으로, TThread 클래스의 배열을 직접 파라미터로 받아서, 이를 핸들 배열의 형태로 변환한 후 API 함수를 호출하게 된다.

이번에는 임계 섹션을 사용해서 파일을 읽고, 쓰는 TCopyThread 클래스를 구현해 보자. 먼저 Create 메소드를 다음과 같이 구현한다.

```

constructor TCopyThread.Create(Target: String);
begin
  FTarget := Target;
  FreeOnTerminate := True;
  inherited Create(False);
end;

```

파라미터로 넘어온 문자열을 필드에 저장하는 역할을 하게 된다.

실제로 파일을 읽고, 쓰는 Execute 메소드는 다음과 같이 구현한다. 여기에서는 TFileStream 을 이용해서 파일 조작을 했는데, 자세한 설명은 다른 장에서 다루게 되므로 생략하고 임계 섹션을 다룬 부분을 유심히 살펴보도록 한다. 이 스레드에서는 두 개의 임계 섹션 블록이 사용된다.

```

procedure TCopyThread.Execute;
var
  FileName: String;
  sStream, dStream: TFileStream;
  pBuf: Pointer;
  cnt, bufSize : LongInt;
  FName: String;
  i: Integer;
begin

```

```

FileName := '';
if not NoFile then
repeat
  try
    if Assigned(Files) and (Files.Count > 0) then
    begin
      EnterCriticalSection(CritSect);
      FileName := Files[0];
      Files.Delete(0);
      LeaveCriticalSection(CritSect);
    end
    else Break;
    if (FileName <> '') then
      sStream := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite)
    else Break;
    bufSize := sStream.Size;
    try
      GetMem(pBuf, bufSize);
      cnt := sStream.Read(pBuf^, bufSize);
      EnterCriticalSection(CritSect);
      if FileExists(FTarget) then
        dStream := TFileStream.Create(FDest, fmOpenReadWrite)
      else dStream := TFileStream.Create(FDest, fmCreate);
      dStream.Seek(0, soFromEnd);
      cnt := dStream.Write(pBuf^, cnt);
      LeaveCriticalSection(CritSect);
    finally
      FreeMem(pBuf, bufSize);
      dStream.Free;
    end;
  finally
    sStream.Free;
  end;
until ((not Assigned(Files)) or (Files.Count = 0));
NoFile := True;
end;

```

다소 길지만, 대부분은 파일을 다루기 위한 코드이다. 먼저 앞에서 사용하는 NoFile 전역 변수는 선택된 파일이 있을 경우에 이 쓰레드를 실행하기 위해서 사용되는 Boolean 형의 변수이다. 이 변수가 False 이면 파일이 선택되어 있는 것이다. Files 변수는 합치는 대상이 될 파일 들을 저장하는 TStringList 형의 변수로 이 변수에 있는 모든 파일을 읽고, 쓰면 이 쓰레드가 종료되고 NoFile 변수는 True, Files 변수의 아이템은 모두 삭제 된다.

파일 이름을 설정하고, Files 변수의 아이템을 삭제하는 부분에 첫번째 임계 섹션 블록이 설정되었다. 여기에 임계 섹션이 설정된 이유는 이렇게 하지 않을 경우, 다른 쓰레드가 Files 전역 변수에 접근해서 아이템을 삭제하거나 하면 예기치 않은 결과가 나타나기 때문이다. 이와 같이 파일과 같은 입출력 작업 이외에, 전역 변수에 대한 작업을 할 때에도 임계 섹션을 잘 활용해야 한다.

그 뒤로 나오는 여러 줄을 파일을 읽어와서 목적 파일에 쓰는 작업을 하는 코드로 자세한 설명은 생략하겠다. 두번째 임계 섹션 블록이 설정되어 있는 부분을 주목하자. 이 부분의 코드는 다음과 같다.

```
EnterCriticalSection(CritSect);
  if FileExists(FTarget) then
    dStream := TFileStream.Create(FTarget, fmOpenReadWrite)
  else dStream := TFileStream.Create(FTarget, fmCreate);
  dStream.Seek(0, soFromEnd);
  cnt := dStream.Write(pBuf^, cnt);
LeaveCriticalSection(CritSect);
```

즉, 읽어온 파일을 쓰게 될 파일에 대한 부분을 임계 섹션 블록으로 설정해서 다른 쓰레드의 접근을 막는 것이다.

마지막으로, 버튼을 클릭했을 때 파일 리스트 박스에 선택된 파일 이름 들을 이용해서 TMasterThread 를 호출하는 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender : TObject);
var
  i : Integer;
  Thread : TMasterThread;
begin
  Files := TStringList.Create;
  with FileListBox1 do
    for i := 0 to Items.Count - 1 do
```

```
    if Selected[i] then
        Files.Add(Items[i]);
        Thread := TMasterThread.Create(Edit1.Text);
    end;
```

## 정 리 (Summary)

멀티 쓰레드 환경은 이미 대부분의 어플리케이션이 지원하고 있는 기본 사양이 되어 가고 있는 추세이다. 그럼에도 불구하고 의외로 멀티 쓰레드를 지원하게 할 때 만날 수 있는 어려움이 꽤 많다.

이번 장에서 다룬 임계 섹션 이외에도 뮤텍이나 세마포어 같이 멀티 쓰레드 프로그래밍을 할 때의 동기화 문제가 특히 중요한데, 여기에 대해서는 MSDN 등의 내용을 통해 더욱 깊은 내용을 알아두기를 권하고 싶다.

## 콜백 함수와 후킹 기법

### (Callback functions and Hooking Techniques)

Win32 API 를 이용하여 어플리케이션을 개발하다 보면, 콜백 함수에 대한 내용을 많이 보게 된다. 그렇다면 이렇게 흔히 보게 되는 콜백 함수란 과연 무엇일까 ?

그리고, 이런 콜백 함수를 이용해서 할 수 있는 것 중에서 후킹 기법을 이용하면 윈도우 운영체제에서 사용할 수 있는 여러가지 마우스 작업이나 키보드 작업 등을 중간에서 조작하여 활용하는 것이 가능하다.

이번 장에서는 콜백 함수와 후킹 기법에 대해 알아보도록 한다.

#### 콜백 함수 (Callback functions)

콜백 함수는 대단히 유용함에도 별로 사용되지 않는 것 중의 하나이다. 이를 활용하면 중복된 코드의 양을 줄일 수 있으며, 읽기가 쉬우면서도 직관적인 모듈을 디자인할 수 있다.

콜백이란 프로시저나 함수를 다른 프로시저나 함수의 파라미터로 넘겨서 그 함수에 특정 이벤트가 발생할 때 호출될 수 있도록 하는 함수를 말한다. 콜백 함수의 실행이 완료되면 제어권은 원래의 프로시저(함수)로 넘어온다.

예를 들어, 객체의 배열이 있고 이 객체들의 특정 메소드를 공통으로 실행시키고 싶다고 하자. 이 경우에는 배열을 루프를 돌리면서 특정 메소드를 호출하는 방법을 사용할 수 있다. 그러면, 동시가 아닌 여러 시점에서 여러 개의 다른 메소드를 실행시키고 싶다고 하자. 이때에는 루프와 같은 단순한 방법으로는 해결할 수가 없다. 이럴 때 콜백 함수를 활용하면 유용한 해결책을 구할 수 있다. 즉, 객체들에 적용해야 하는 마스터 프로시저를 작성하고 이들 각각이 각 객체를 파라미터로 넘겨주면서 콜백 프로시저를 호출하면 된다.

또한, 콜백 함수를 달리 말하면 어플리케이션에서 구현된 함수의 주소를 DLL 에 포함되어 있는 함수에 보낼 수 있는 것을 말한다. DLL 의 함수는 어플리케이션의 함수에게 다시 정보를 담아서 보낸다.

#### 콜백 함수와 Win32 API

콜백 함수를 설명할 때 빠지지 않고 단골로 설명하는 윈도우 API 함수에는 EnumFonts() 함수가 있다. EnumFonts() 함수는 주어진 장치 컨텍스트에서 사용이 가능한 폰트를 나열하는 함수이다. 각각의 나열되는 폰트는 EnumFonts() 함수가 어플리케이션의 함수로 콜백을 하게 된다. 즉, 각 폰트에 대한 정보를 돌려주는 것이다. 이런 과정이 나열할 폰트가 모두 나열되거나, 콜백 함수가 더 이상의 나열을 중지하고자 하는 의미의 0 을 반환할 때까

지 계속된다.

대부분의 콜백 함수를 파라미터로 사용할 수 있는 윈도우 API 함수는 lpData 파라미터 역시 사용할 수 있도록 허용하고 있다. 보통 이런 lpData 파라미터는 어플리케이션이 정의한 데이터를 나타내는 역할을 한다. lpData 파라미터가 DLL 에 전달될 때에는 보통 LongInt 형으로 전달된다. 그러므로, LongInt 형의 데이터가 아닌 보다 복잡한 형태의 데이터를 넘기고자 할 경우에는 데이터 구조체의 포인터를 LongInt 로 형변환(typecast)한 뒤에 DLL 의 함수를 호출한다. 그리고, DLL 의 함수에 의해서 처리된 후에 콜백 함수로 넘어오는 LongInt 형의 데이터를 다시 포인터로 형변환하면 데이터 구조체에 접근할 수 있게 된다.

앞에서도 잠시 언급했지만, 콜백 함수가 윈도우 API 함수에 의해 호출될 때에는 윈도우 API 함수에 넘겨준 lpData 파라미터가 다시 콜백 함수가 사용할 수 있도록 넘어오게 된다. 이렇게 함으로써 어플리케이션에서 전역 변수를 선언하지 않아도 프로세스 간의 장벽을 넘을 수 있게 된다. 간단한 예를 들어보면 모든 가능한 폰트를 드롭-다운 리스트 박스에 보여주는 대화 상자를 가정해 보자. 이렇게 하려면, WM\_INITDIALOG 메시지를 처리할 때 EnumFonts() 함수를 호출해야 하는데, 이때 폰트의 정보를 처리할 콜백 함수의 주소를 담아서 호출해야 한다. 콜백 함수는 아마도 넘어온 각 폰트의 이름을 리스트 박스에 추가하는 역할을 하게 된다. 이때 문제가 하나 있는데, 콜백 함수가 리스트 박스에 대한 핸들을 가지고 있지 않기 때문에, 리스트 박스에 대한 핸들을 전역 변수에 담아서 처리해야 한다. 그런데, 이때 lpData 파라미터를 이용해서 리스트 박스의 핸들을 콜백 함수에서 받게 되면 이러한 문제를 해결할 수 있다.

물론 전역 변수를 쓰는 것도 하나의 해결책이 되겠지만, 언제나 전역 변수가 적은 수로 유지하는 것이 좋은 습관이므로 가능한 lpData 파라미터를 활용하는 것이 좋다.

전역 변수를 사용할 때 또 하나의 문제점은 콜백 함수를 16 비트 DLL 에 구현할 경우에는 데이터를 저장한 전역 변수가 콜백을 이용할 때 데이터가 파괴될 가능성이 있다. 예를 들어, 앞의 폰트 대화 상자를 여러 개의 어플리케이션에서 사용한다고 가정하자. 아마도 이런 경우에는 각각의 어플리케이션에서 폰트 대화 상자를 구현하기 보다는, 폰트 대화 상자를 DLL 에 구현하고 이를 사용하기를 원할 것이다. 그런데, 이때 DLL 에서 리스트 박스의 핸들을 저장하기 위해 전역 변수를 사용한다면, 하나 이상의 어플리케이션이 동시에 대화 상자를 로드할 때 문제가 발생할 수 있다. 각각의 폰트 대화 상자의 복사본(copy)은 각기 다른 리스트 박스의 핸들을 담게 되므로, 전역 변수를 사용해서는 문제가 발생하게 된다. 이럴 때에는 리스트 박스의 핸들을 lpData 파라미터를 이용해서 EnumFonts() 함수에 넘겨주면 된다. 이런 문제는 Win32 에서는 DLL 이 독립적인 기억 공간을 확보하기 때문에 문제가 되지 않는다.

## 콜백 함수를 이용한 API 활용 예제

콜백 함수를 이용해서 현재 시스템에서 돌아가는 메인 윈도우의 캡션을 보여주는 예제를 가

지고 콜백 함수의 기본적인 사용 방법을 익혀 보도록 하자.

먼저 사용할 윈도우 API 함수인 EnumWindows API 함수의 정의 부분을 살펴 보자.

```
function EnumWindows(lpEnumFunc: TFNWndEnumProc; lParam: LPARAM): BOOL; stdcall;
```

여기서 lpEnumFunc 의 파라미터는 콜백 함수의 주소를 넘기게 되며, lParam 파라미터는 어플리케이션에서 정의한 데이터를 나타낸다. 이때 콜백 함수의 프로시저 형은 윈도우 도움말을 바탕으로 어플리케이션의 type 선언문에서 선언해 주어야 한다. 그러므로, 다음과 같은 형태의 프로시저 형을 어플리케이션의 인터페이스 섹션에 선언한다. 여기서 파라미터의 데이터 형만 맞으면 콜백 함수로 사용하는데에는 전혀 지장이 없다.

사실 이런 프로시저 형을 직접 type 선언문에 선언하지 않아도 사용할 콜백 함수의 파라미터의 데이터 형과 순서, 수만 맞으면 아무런 상관없이 사용할 수 있다.

type

```
EnumWindowsProc = function(HWnd: THandle; Param: Pointer): Boolean; stdcall;
```

첫 번째 파라미터는 각 메인 윈도우의 핸들이고, 두 번째 파라미터는 EnumWindows 함수를 호출할 때 넘겨주는 값이다. 사실 파스칼에서 TFNWndEnumProc 형은 제대로 정의된 것이 아니고, 단지 포인터일 뿐이다. 즉, 함수에 적절한 파라미터를 넘겨주고 나서는 이를 포인터로 사용해서 호출하는 대신 그 함수의 주소를 이용한다는 것을 의미한다.

새로운 어플리케이션을 시작하고 폼에 리스트 박스와 버튼을 하나씩 없어서 폼을 다음 그림과 같이 디자인 한다.





그러면 콜백 함수로 사용할 함수를 다음과 같이 제작한다.

```
function GetCaption(HWnd: THandle; Param: Pointer): Boolean; stdcall;
var
    Text: String;
begin
    SetLength(Text, 100);
    GetWindowText(HWnd, PChar(Text), 100);
    Form1.ListBox1.Items.Add(IntToStr(HWnd) + ':' + Text);
    Result := True;
end;
```

이런 콜백 함수는 파라미터만 맞으면 사용이 가능하다. 이 콜백 함수의 역할은 윈도우의 캡션을 문자열로 읽어서 리스트 박스에 추가한다.

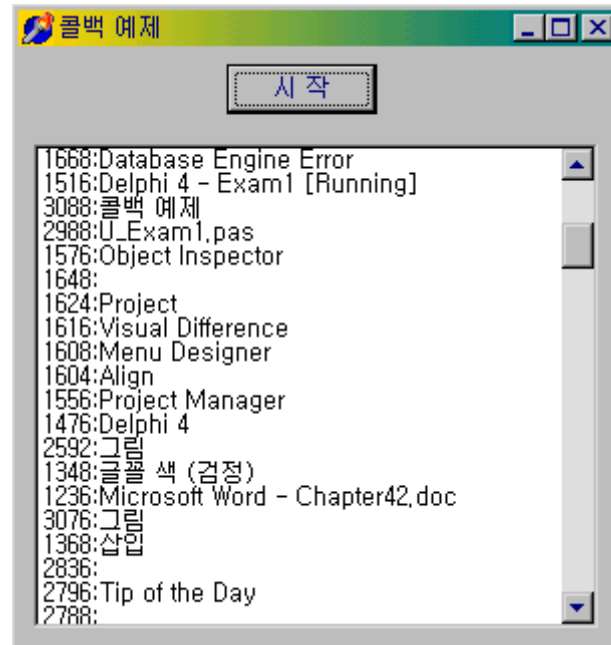
그러면, 이 콜백 함수를 이용하기 위한 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성하도록 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    EnumWin: EnumWindowsProc;
begin
    ListBox1.Items.Clear;
    EnumWin := GetCaption;
    EnumWindows(@EnumWin, 0);
end;
```

앞의 코드는 단지 이렇게 사용할 수도 있다는 것을 보여주기 위한 것으로, 변수로 선언한 EnumWin 의 선언과 이 변수에 GetCaption 함수의 주소를 저장해서 EnumWindows API 함수를 호출하는 과정을 다음과 같이 간단히 처리하는 것과 내용은 같은 것이다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ListBox1.Items.Clear;
    EnumWindows(@GetCaption, 0);
end;
```

이제 이 프로그램을 실행하고 버튼을 클릭하면 다음과 같은 실행 화면을 볼 수 있을 것이다.



## 콜백 함수의 실용적 활용

콜백 함수를 이용하는 방법을 익히는 좋은 예제로 디렉토리를 뒤지면서 파일을 검색하고, 파일이 발견될 때마다 콜백 함수를 호출하여 이를 표시하는 등의 것을 예로 들 수 있겠다. 콜백 함수를 사용하기 위해서는 먼저 프로시저 형을 유닛의 type 섹션에 다음과 같이 선언해야 한다.

```
TFileSearchCallback = procedure(FileName: string) of Object;
```

이와 같이 콜백 함수를 사용하기 위해서는 객체의 메소드로서의 프로시저 형을 선언해서 사용한다. 여기서 파라미터 형이 맞을 경우 얼마든지 콜백으로 사용이 가능하다. 그리고 나서는 실제로 콜백 함수를 호출할 프로시저를 다음과 같이 선언한다.

```
procedure SearchDirectory(Dir, Condition: string; cb: TFileSearchCallback);
```

이 프로시저의 형태를 살펴보면 검색을 할 디렉토리나 검색할 문자열을 나타내는 파라미터인 Dir, Condition 은 다른 프로시저의 형태와 별로 다를 것이 없지만, cb 파라미터에서 앞서 선언한 TFileSearchCallback 프로시저 형을 사용하는 것이 중요한 부분이다. 이 파라미터의 의미는 여기에 콜백 함수를 대입하여 콜백을 이용할 수 있게 된다는 것이다.

이제 실제로 디렉토리를 검색하는 예제 어플리케이션을 작성해보도록 하자.

새로운 어플리케이션을 시작하자.

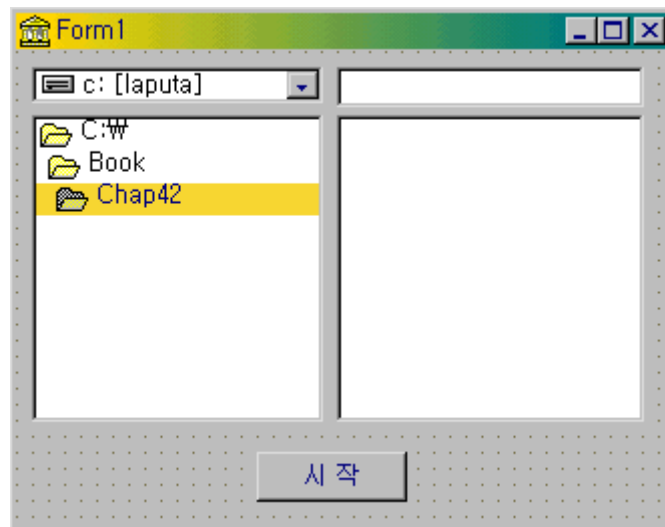
앞에서 설명한 대로 콜백 프로시저 형을 type 섹션에 선언하고, SearchDirectory 프로시저를 private 섹션에 선언한다.

그리고, 다음과 같이 구현한다.

```
procedure TForm1.SearchDirectory(Dir, Condition: string; cb: TFileSearchCallback);
var
    SearchRec: TSearchRec;
    Value: LongInt;
begin
    Value := FindFirst(Dir + 'W' + Condition, faAnyFile, SearchRec);
    while Value = 0 do
    begin
        cb(Dir + 'W' + SearchRec.Name);
        Value := FindNext(SearchRec);
    end;
end;
```

기본적으로 이 프로시저는 Dir 파라미터로 넘어온 디렉토리를 검색해서 파일을 찾게 되면 콜백 함수를 호출하는데, 콜백 함수에게 현재 파일의 경로를 포함한 파일 이름을 파라미터로 넘겨주게 된다. 그러면 이제 실제로 콜백을 수행할 콜백 함수를 작성하고 예제를 눈에 보일 수 있도록 만들어 보자.

폼에 TButton, TListBox, TEdit, TDriveComboBox, TDirectoryListBox 컴포넌트를 하나씩 올려 놓도록 하자. 그리고 Button1 의 Caption 프로퍼티를 ‘시 작’ 으로, Edit1 의 Text 프로퍼티를 ‘’로 다음과 같이 설정한다. 여기서 TEdit 컴포넌트에 검색할 문자열을 입력하고 버튼을 클릭하면 해당되는 파일을 찾아서 파일의 이름일 TListBox 컴포넌트에 추가하도록 하는 것이다. 그리고, TDriveComboBox 와 TDirectoryListBox 를 연결하기 위해 DriveComboBox1 의 DirList 프로퍼티를 DirectoryListBox1 으로 설정한다.



그리고 나서, 다음과 같이 콜백 프로시저를 private 섹션에 선언한다.

```
procedure FileSearchCallback(FileName: string);
```

이 콜백 함수는 디렉토리에서 파일을 찾게 되면, 파일 이름을 TListBox 컴포넌트에 추가하여 보여주는 간단한 함수로 다음과 같이 작성하도록 한다.

```
procedure TForm1.FileSearchCallback(FileName: string);
begin
    ListBox1.Items.Add(ExtractFileName(FileName))
end;
```

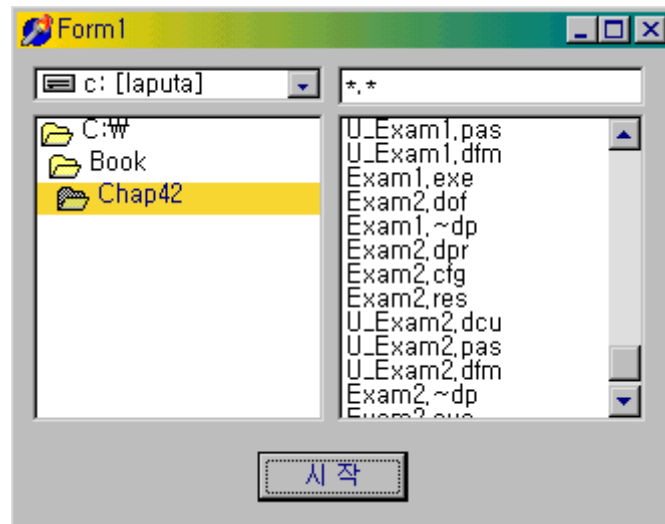
이제 Button1의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ListBox1.Items.Clear;
    SearchDirectory(DirectoryListBox1.Directory, Edit1.Text, FileSearchCallback);
end;
```

여기서 앞에서 예를 든 다른 프로시저와는 달리 콜백 함수의 주소를 @FileSearchCallback으로 넘겨주지 않고 직접 FileSearchCallback으로 넘겨 주는 것에 주목한다. 이것은 델파이가 사용하는 오브젝트 파스칼의 특성에 기인한 것으로, C/C++의 경우에는 기본적으로 객체의 주소를 포인터로 선언하고, 이를 이용해서 프로그래밍을 해야 하므로 명시적으로 함

수의 주소를 가리키는 연산자를 사용해야 하지만, 오브젝트 파스칼은 모든 기본 객체 들이 참조로 사용되기 때문에 주소를 나타내는 '@'를 사용하지 않아도 잘 작동한다.

이제 이 어플리케이션을 컴파일해서 실행하고, 적당한 디렉토리와 검색 문자열을 입력한 후 버튼을 클릭하면 다음과 같은 실행 화면을 얻을 수 있을 것이다.



## 후크 함수 (Hook functions)

후크 함수는 윈도우 메시지 시스템에 삽입되어 메시지에 대한 처리가 일어나기 전에 메시지 스트림에 접근해서 이를 처리할 수 있는 콜백 함수이다. 흔히, 메시징 시스템에 다른 후크가 있는 경우가 있는데, 이 때에는 후크 연쇄(Hook chain)에서의 다음의 후크를 호출해서 이들이 모두 동작할 수 있도록 해야 한다.

후크를 이용해서 시스템에서 메시지 트래픽을 모니터하는 서브 루틴을 제작한다든가, 목적 윈도우 프로시저에 도달하기 전에 일부 메시지를 처리하는 등의 일을 할 수 있다.

후크는 각각의 메시지를 처리할 때 거쳐야 하는 과정을 증가시키기 때문에 시스템의 속도를 다소 저하시키는 경향이 있다. 그러므로, 후크는 꼭 필요할 때 설치했다가 가능한 빨리 제거해 주어야 한다.

후크 함수를 설치할 때에는 SetWindowsHookEx() API 함수를 이용한다. 이 함수를 호출하면 설치된 후크 함수에 대한 32 비트 핸들을 얻게 되며, 이 핸들을 이용해서 후크를 제거하거나 다음의 후크를 호출할 때 이용한다.

후크를 제거할 때에는 UnHookWindowsHookEx() API 함수를, 다음 후크를 호출할 때에는 CallNextHookEx() 함수를 호출한다.

이 때 시스템 전반에 걸친 호크는 후크 함수가 DLL 에 위치해야 하지만, 어플리케이션에 지정된 후크 함수는 어플리케이션이나 DLL 에 모두 위치할 수 있다.

## 후크 연쇄 (Hook Chains)

윈도우는 다른 종류의 많은 후크의 종류를 가지고 있다. 각각의 후크의 형은 윈도우 메시지 처리 메커니즘의 다른 면에 접근한다. 예를 들어, 마우스 메시지에 대한 메시지 트래픽을 모니터할 때에는 WM\_MOUSE 후크를 이용한다.

윈도우는 각각의 후크 종류에 따라 분리된 후크 연쇄(hook chain)를 가지고 있다. 후크 연쇄는 후크 프로시저로 불리는 어플리케이션이 정의한 콜백 함수에 대한 포인터의 리스트이다. 특정한 종류의 후크와 연관된 메시지가 발생하면 윈도우는 메시지를 후크 연쇄에서 참조하는 각각의 후크 프로시저에 넘기게 된다. 후크 프로시저의 동작은 연관된 후크의 종류에 따라 달라진다. 일부의 후크 프로시저는 메시지를 모니터할 뿐이지만, 일부의 경우에는 메시지를 변경하거나 전달을 중지시킬 수 있다.

## 후크 프로시저 설치와 해제할 때 주의점

후크 프로시저를 설치할 때에는 SetWindowsHookEx 함수를 이용한다. 그리고, 이 함수를 호출할 때에는 호출하는 후크의 종류와 프로시저가 모든 쓰레드와 연관되어야 하는지, 아니면 특정 쓰레드와 연관되는지 여부 그리고 프로시저 엔트리 포인터 등을 지정하게 된다.

후크 프로시저를 설치하는 어플리케이션은 반드시 전역 후크 프로시저를 DLL 에 분리해야 한다. 대신 후크 프로시저를 설치하기 전에 DLL 모듈의 핸들을 가지고 있어야 한다. 즉, LoadLibrary 함수를 이용해서 DLL 모듈의 핸들을 얻은 후, 이렇게 핸들을 얻으면 GetProcAddress 함수를 이용해서 후크 프로시저의 주소를 얻을 수 있게 된다. 마지막으로 SetWindowsHookEx 함수를 이용해서 후크 프로시저의 주소를 적절한 후크 연쇄에 설치한다. SetWindowsHookEx 함수에는 모듈의 핸들과 후크 프로시저의 엔트리 포인트, 그리고 쓰레드에 대한 identifier 을 넘겨 주는데, 보통은 이 값으로 0 을 넘겨서 시스템의 모든 쓰레드에 반응할 수 있도록 한다.

특정 쓰레드에 대한 후크 프로시저를 해제(후크 연쇄에서 프로시저의 주소를 삭제하는 것)할 때에는 UnhookWindowsHookEx 함수에 후크 프로시저의 핸들을 지정해서 호출하게 된다. 전역 후크 프로시저를 해제할 때에도 UnhookWindowsHookEx 함수를 호출하기는 마찬가지이다. 그런데, 이 함수는 후크 프로시저를 담고 있는 DLL 을 해제하지 않는다. 이는 전역 후크 프로시저는 모든 윈도우 기반의 어플리케이션에서 호출되기 때문에, 이들이 암시적으로 LoadLibrary 함수를 호출하기 때문으로, 이들이 모두 FreeLibrary 를 호출하게 할 수 있는 방법이 없다. 그러므로, 결국에는 모든 어플리케이션을 중지시키거나 또는 이들이 모두 FreeLibrary 를 호출하게 해야 한다.

이를 해결하기 위해서 전역 후크 프로시저를 설치할 때에는 설치 함수를 DLL 에 후크 프로시저와 함께 제공한다. 이렇게 하면 설치 어플리케이션이 DLL 모듈의 핸들을 가지고 있을 필요가 없다. 각 어플리케이션은 반드시 DLL 과 연결해야만 후크를 설치할 수 있게 되고,

설치 함수는 SetWindowsHookEx 함수의 호출을 통해 DLL 의 모듈 핸들을 제공한다. 어플리케이션은 종료될 때 이 DLL 의 핸들을 통해서 후크를 해제하게 된다.

## 윈도우 후크의 종류

윈도우의 후크에는 다음과 같은 종류 들이 있다.

- WH\_CALLWNDPROC:

SendMessage() 함수가 호출될 때마다 호출되는 윈도우 프로시저 후크

- WH\_CBT:

CBT(computer based training) 후크는 윈도우를 생성, 파괴, 최대화, 이동, 크기 변화 등의 이벤트가 있거나, 마우스나 키보드 이벤트를 제거하기 전에 또는 입력 포커스의 변경이나 시스템 메시지 큐의 동기화가 있기 전에 호출되는 후크이다.

- WH\_GETMESSAGE:

GetMessage() 함수가 어플리케이션 큐에서 메시지를 가져올 때마다 호출되는 후크이다.

- WH\_HARDWARE:

어플리케이션이 GetMessage(), PeekMessage() 함수를 호출하고, 처리할 하드웨어 이벤트 (마우스와 키보드 이벤트를 제외한)가 있을 때 호출되는 후크이다.

- WH\_JOURNALRECORD:

시스템 메시지 큐에서 메시지를 삭제할 때 호출되는 후크이다.

- WH\_JOURNALPLAYBACK:

마우스나 키보드 메시지를 시스템 메시지 큐에 삽입할 때 이용되는 후크이다.

- WH\_KEYBOARD:

어플리케이션이 GetMessage(), PeekMessage() 함수를 호출하고, WM\_KEYUP 또는 WM\_KEYDOWN 키보드 메시지를 처리해야 할 때 호출되는 후크이다.

- WH\_MOUSE:

어플리케이션이 GetMessage(), PeekMessage() 함수를 호출하고, 마우스 메시지를 처리해야 할 때 호출되는 후크이다.

- WH\_MSGFILTER:

대화 상자나 메시지 상자 또는 메뉴가 메시지를 불러올 때, 메시지가 처리되기 전에 호출되는 후크이다.

- WH\_SHELL:

시스템이 만들어 놓은 notification 메시지가 있을 때 호출되는 후크이다.

- WH\_SYSMSGFILTER:

대화 상자나 메시지 상자 또는 메뉴가 메시지를 불러올 때, 메시지가 처리되기 전에 호출되는 시스템 전체에 대한 후크이다.

## 기본적인 후크 예제

후크를 어떻게 설치하고 해제하는지를 코드를 통해서 알아보자.

시스템 후크로 사용할 함수를 DLL 파일에 담고, 이 DLL 을 사용하여 시스템 후크를 설치, 제거하는 어플리케이션을 하나 제작한다.

DLL 파일은 ExamLib3.DLL, 어플리케이션은 Exam3.EXE 로 한다.

먼저, DLL 파일을 만들기 위해 새로운 DLL 프로젝트를 하나 시작한다.

그리고 다음의 코드를 입력하자.

```
library ExamLib3;
```

```
Uses
```

```
    SysUtils, Windows, Dialogs;
```

```
function KeyboardProc(Code, WParam, LParam: Integer): LRESULT; stdcall;
```

```
var
```

```
    Val: Integer;
```

```
begin
```

```
    Val := -1;
```

```
    if (Code >= 0) then Val := CallNextHookEx(0, Code, WParam, LParam);
```

```
    Result := Val;
```

```
end;
```

```
exports
```

```
    KeyboardProc index 1;
```



```
begin  
end.
```

이 DLL 파일은 실제로 아무 동작을 하지 않고, 다음의 후크를 호출하는 역할 만을 한다.  
그러면, DLL 파일을 이용해서 후크를 설치, 해제하는 어플리케이션을 제작하자.  
폼을 생성할 때 후크를 설치하고, 폼을 파괴할 때 후크를 해제하도록 한다.  
후크의 핸들을 public 멤버인 HandleHook 에 저장하고, DLL 의 핸들을 private 멤버인 hInst 에 저장하도록 하자.  
어플리케이션의 소스는 다음과 같다.

```
unit U_Exam3;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
```

```
type
```

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

```
    procedure FormCreate(Sender: TObject);
```

```
    procedure FormDestroy(Sender: TObject);
```

```
private
```

```
    hDLL: HInst;
```

```
public
```

```
    HandleHook: HHook;
```

```
end;
```

```
var
```

```
    Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```

begin
    hDLL := LoadLibrary('ExamLib3');
    if hDLL = 0 then
        ShowMessage('DLL 을 로드하지 못했습니다.');
```

```

    HandleHook := SetWindowsHookEx(WH_KEYBOARD,
        GetProcAddress(hDLL,PChar('KeyboardProc')), hDLL, 0);
    if HandleHook = 0 then
        ShowMessage('후크를 설치하지 못했습니다.');
```

```

end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    if not UnhookWindowsHookEx(HandleHook) then
        ShowMessage('후크를 해제하지 못했습니다.');
```

```

end;

end.
```

즉, 후크를 설치할 때 우선 DLL 파일을 LoadLibrary 함수를 이용해서 메모리에 적재한 후 DLL 파일에 저장되어 있는 함수의 주소를 후크 프로시저로 설정한다. 이때 윈도우 API 함수인 SetWindowsHookEx 를 사용하는데, 후크를 설치하지 못하면 0 이 반환된다.

후크를 제거할 때에는 설치된 후크의 핸들을 이용해서 UnhookWindowsHookEx 함수를 호출하면 된다. 이 함수의 경우에는 후크를 제거하지 못하면 False 를 반환한다.

## 후크를 이용한 매크로 작성기 제작

이번에는 실제 사용 가능하도록 후크를 응용해보자.

아래아 한글이나 워드, 그 밖의 많은 에디터에 보면 대부분의 경우 키보드와 마우스 동작을 기록했다가 저장하고, 이를 다시 재생할 수 있는 매크로 기능을 제공한다. 그러면, 윈도우의 후크 프로시저를 이용해서 키보드와 마우스 입력을 기록했다가 이를 다시 재생할 수 있는 DLL 과 이를 이용한 간단한 예제 어플리케이션을 제작해 보자.

이 예제는 볼랜드에서 제공하는 기술 백서(Technical white paper)에 공개되었던 어플리케이션에 기초한 것임을 미리 밝혀 둔다.

이렇게 메시지를 기록하고, 재생하기 위해서는 시스템 전체에 적용되는(system wide) WH\_JOURNALRECORD, WH\_JOURNALPLAYBACK 윈도우 후크를 이용한 후크 함수를 사용하게 된다. 이때 각각의 윈도우 후크에 대한 후크 프로시저를 JournalRecord()와

JournalPlayback()이라고 하자.

JournalRecord 와 JournalPlayback 후크의 콜백 함수를 설치하기 위해 윈도우 API 함수인 SetWindowsHookEx()에 후크 콜백 함수의 주소를 넘겨 주어야 한다.

이때 JournalRecord 와 JournalPlayback 콜백 함수는 시스템 후크 (system wide hook)이므로, 반드시 DLL 에 위치해야 하며 후크 콜백 함수의 주소는 인스턴스화한 주소를 넘겨 주지 않아도 된다. 이 함수를 호출하면 후크 콜백 함수에 대한 32 비트 핸들을 반환하게 되는데, 이 핸들을 이용해서 다른 후크 함수가 이미 후크 연쇄(hook chain)에 존재하는지 확인하고, 기록이 끝나면 후크를 연쇄에서 제거 하는 등의 일을 할 수 있다.

일단 JournalRecord 콜백 함수의 주소를 가지고 SetWindowsHookEx()를 호출하면, 윈도우는 즉시 메시지의 기록을 시작한다. 이때 JournalRecord 콜백 함수가 호출되는 조건은 다음과 같다.

1. 기록할 키보드나 마우스 이벤트가 존재할 때 (HC\_ACTION)
2. 시스템이 모달 상태에 들어가거나 (HC\_SYSMODALON), 모달 상태에서 빠져 나올 경우 (HC\_SYSMODALOFF).
3. 시스템이 후크 연쇄에서 다음 후크를 호출하기 원할 때

키보드나 마우스 이벤트가 있을 경우에는 HC\_ACTION 후크와 같은 경우인데, 이 이벤트를 기록할 수 있다. 이때 EVENTMSG 형의 구조체에 대한 포인터가 JournalRecord 콜백 함수의 lParam 파라미터에 담겨서 넘어오게 된다. 이벤트가 발생한 시스템 시간은 EVENTMSG 구조체의 time 파라미터에 담겨 있다. 재생을 하기 위해서는 EVENTMSG 구조체를 복사하고, 그 복사본의 time 을 적절하게 조절하면 된다. 즉, 메시지를 기록할 때 시스템 시간을 얻은 후 각 메시지의 시간에서 처음에 얻은 시스템 시간을 빼면, 기록이 시작된 후 얼마의 간격으로 메시지 들이 발생했는지 알아낼 수 있다. 이 값들을 재생이 시작되는 시간에 더하면 기록할 때와 같은 간격으로 메시지들을 발생시킬 수 있다.

시스템이 모달 상태에 들어 갔을 경우에는 HC\_SYSMODALON 후크와 같은 경우로 이 때에는 잠시 기록을 멈추고, 후크 연쇄의 다음 메시지 후크를 호출하게 하여야 하며, 모달 상태에서 빠져 나올 경우의 후크인 HC\_SYSMODALOFF 의 경우에 다시 기록을 재개하도록 하면 된다.

만약 코드 값이 0 보다 작을 경우에는 시스템이 후크 연쇄의 다음 메시지 후크를 호출할 것을 요구하게 되며, 이 때에는 이를 따라야 한다. 그리고, 기록이 끝났을 경우에는 윈도우 API 함수인 UnHookWindowsHookEx()을 SetWindowsHookEx() 함수를 호출했을 때 얻은 32 비트 핸들을 파라미터로 해서 호출하면 후크 콜백 함수를 후크 연쇄에서 제거하게 된다. 기록된 메시지를 재생하고자 할 때에는 또 다시 SetWindowsHookEx() API 함수를 JournalPlayback 콜백 함수의 주소를 파라미터로 해서 호출하면 윈도우는 즉시 재생을 시작하게 된다. 재생을 하는 동안에는 정상적인 마우스와 키보드 입력이 시스템에 의해 중지

된다.

JournalPlayback 콜백 함수는 다음의 조건에 부합될 때 각각 다음과 같은 역할을 하여야 한다.

1. HC\_SKIP:

다음 메시지를 불러온다. 재생할 메시지가 더 이상 없으면 UnHookWindowsHookEx() 함수를 호출해서, 후크를 제거한다.

2. HC\_GETNEXT:

현재의 메시지를 재생한다.

3. HC\_SYSMODALON:

시스템이 모달 상태에 들어간 경우로, 메시지 재생 중에 모달 상태로 들어간다는 것은 시스템에 어떤 문제가 생겼음을 의미한다. 그러므로, 후크 연쇄의 다음 후크를 호출하여 이를 처리할 수 있도록 해준다.

4. HC\_SYSMODALOFF:

시스템이 모달 상태에서 빠져 나온 경우로, 윈도우는 JournalPlayback 콜백 프로시저를 제거하고, 후크 연쇄의 다음 후크를 호출한다.

5. 코드가 0 보다 작은 경우:

시스템이 더 이상의 처리를 원하지 않고, 다음 후크를 호출하라고 요구하는 것이다.

이런 시스템 후크를 구현하기 위해서는 SetWindowsHookEx() 함수를 호출하기 전에, 첫 번째 메시지를 받아서 시스템 시간을 얻어야만 각각의 기록된 메시지 들이 재생시에 동기화되어 동작하게 할 수 있다.

이때 윈도우는 같은 메시지를 한 번 이상 반복할 것인지 묻는다. 윈도우가 현재 메시지를 재생할 것인지를 처음 물어올 때 JournalPlayback 콜백 함수는 현재 시간과 메시지가 작동하도록 되어 있는 시간의 차이 부분을 반환한다. 만약 이 값이 음수라면 0 을 반환하도록 해야 한다. 또한, 같은 메시지가 한번 이상 요구될 경우에도 0 을 반환한다.

## ● 후크 DLL 의 동작 방식

실제 핵심적인 역할을 담당하는 DLL 파일에는 다음과 같은 세 가지 함수와 윈도우가 사용하게 될 두 가지 후크 함수가 포함되어 있다.

1. StartRecording()
2. StopRecording()
3. Playback()
4. JournalRecordProc()
5. JournalPlaybackProc()

StartRecord() 함수는 JournalRecordProc() 후크를 설치하고 이벤트를 기록하기 시작하는 역할을 한다. StartRecord()를 호출하다가 에러가 발생하거나, 이미 매크로를 기록하거나 재생하고 있을 때에는 더 이상의 처리를 하지 않고, 0 을 반환한다. 매크로의 기록이 이루어지는 중간에는 키보드와 마우스 이벤트를 배열에 저장한다. 참고로 지나치게 많은 수의 메모리 소모를 줄이기 위해서 최대치를 설정한다.

StopRecording()가 호출되면, 기록한 이벤트 들을 디스크에 저장한다. 그리고 나서, JournalRecordProc() 후크를 제거한다. 이때 아무것도 기록된 것이 없을 경우에는 -1 을 후크를 제거할 때 에러가 발생하면 -2 를 에러 코드로 반환한다. 에러가 없을 때에는 기록된 메시지의 수를 반환한다.

Playback() 함수는 기록된 매크로를 재생할 때 호출한다. 에러가 발생하거나, 재생할 것이 없으면 더 이상의 처리를 하지 않고 0 을 반환한다. 재생이 끝나면, 매크로를 재생하도록 호출한 어플리케이션을 콜백 한다. 그러므로, 이 함수를 호출하는 어플리케이션은 콜백 함수를 작성하여 재생이 끝났을 때의 처리를 해줄 수 있다.

데이터의 공유를 위해서 몇 개의 전역 변수를 선언했다. 물론 이러한 데이터 공유를 위해서는 메모리 맵 파일 등의 보다 세련된 방법을 선택할 수도 있겠으나, 여기에 대해서는 다음에 다루도록 한다. 일단 EventMsg 구조체 배열에 대한 포인터인 PMsgBuff 라는 전역 포인터를 정의하고, 이를 이용해서 메모리에 이벤트를 기록하고 재생한다. 일단 DLL 이 시작되면 이 포인터를 nil 로 초기화하고, 매크로를 기록하거나 재생할 때에만 이 포인터가 실제로 메모리 블록을 가리키도록 한다. 다른 경우에는 언제나 nil 값을 가지도록 설정함으로써 현재 매크로가 기록되거나, 재생되는지 여부를 확인할 수 있다.

그 밖에 사용하는 전역 변수에는 다음과 같은 내용들을 담게 된다.

1. TheHook: 후크 프로시저의 32 비트 핸들
2. StartTime: 매크로의 기록이나 재생이 시작되는 시간
3. MsgCount: 기록된 메시지의 총 수

4. CurrentMsg: 현재 재생되고 있는 메시지
5. ReportDelayTime: 지체된 시간을 리포트 해야 하는지 여부
6. SysModalOn: 시스템에 현재 모달 상태에 있는지 여부
7. cbPlaybackFinishedProc: 어플리케이션 콜백 함수의 인스턴스 주소
8. cbAppData: Playback() 함수에게 넘겨주는 어플리케이션 데이터의 파라미터

## ● 후크 DLL 의 구현

그러면, 실제로 DLL 을 제작해 보자.

먼저 DLL 의 프로젝트의 이름을 Hook2Lib.dpr 로 저장한다.

그리고, 사용할 데이터 형과 전역 변수를 다음과 같이 선언한다.

```
library Exam4Lib;
```

```
uses
```

```
Windows;
```

```
type
```

```
TwMsg = LongInt;
```

```
TwParam = LongInt;
```

```
TIParam = LongInt;
```

```
const
```

```
MAXMSG = 6500;
```

```
type
```

```
PEventMsg = ^TEventMsg;
```

```
TMsgBuff = Array[0..MAXMSG] of TEventMsg;
```

```
TcbPlaybackFinishedProc = procedure(AppData: Longint); stdcall;
```

```
var
```

```
PMsgBuff: ^TMsgBuff;
```

```
TheHook: HHook;
```

```
StartTime: Longint;
```

```
MsgCount: Longint;
```

```
CurrentMsg: Longint;
```

```

ReportDelayTime: Bool;
SysModalOn: Bool;
cbPlaybackFinishedProc: TcbPlaybackFinishedProc;
cbAppData: Longint;

```

상수로 선언한 MAXMSG 는 메시지를 기록할 때 과도한 메모리 사용을 막기 위해서 한도를 정한 것으로 조절이 가능하다.

TcbPlaybackFinishedProc 프로시저 형은 DLL 의 Playback() 함수를 호출하는 어플리케이션에 대한 콜백을 지원한다. 어플리케이션은 이 프로시저 형에 맞는 콜백 함수를 작성하고, DLL 의 Playback() 함수에 콜백 함수의 주소를 넘겨 주면 Playback() 함수의 종료와 함께 이 콜백 함수를 호출하게 된다.

나머지 전역 변수에 대해서는 앞서도 간략히 언급 했으므로 자세한 설명은 생략 하겠다. 먼저 매크로를 기록할 후크 프로시저인 JournalRecordProc 함수의 구현 부분을 살펴 보자.

```
function JournalRecordProc(Code: Integer; wParam: TwParam; lParam: TlParam):
```

```
    Longint; stdcall;
```

```
begin
```

```
    Result := 0;
```

```
    case Code of
```

```
        HC_ACTION:
```

```
        begin
```

```
            if SysModalOn then Exit;
```

```
            if MsgCount > MAXMSG then Exit;
```

```
            PMsgBuff^[MsgCount] := PEventMsg(lParam)^;
```

```
            Dec(PMsgBuff^[MsgCount].Time, StartTime);
```

```
            Inc(MsgCount);
```

```
            Exit;
```

```
        end;
```

```
        HC_SYSMODALON:
```

```
        begin
```

```
            SysModalOn := True;
```

```
            CallNextHookEx(TheHook, Code, wParam, lParam);
```

```
            Exit;
```

```
        end;
```

```
        HC_SYSMODALOFF:
```

```
        begin
```

```

    SysModalOn := False;
    CallNextHookEx(TheHook, Code, wParam, lParam);
    Exit;
end;
end;
if Code < 0 then
    Result := CallNextHookEx(TheHook, Code, wParam, lParam);
end;

```

일단 이 함수의 리턴 값을 0 으로 설정하고, Code 파라미터의 값에 따른 처리를 해 준다.

이 코드의 값이 HC\_ACTION 인 경우에는 기록할 키보드나 마우스 이벤트가 있는 것이므로 lParam 의 메시지를 버퍼에 저장한다. 이때 주의할 것은 메시지의 시간을 기록하는 것인데, 최초에 StartRecord() 함수가 호출되면 StartTime 전역 변수에 기록 시작 시각이 기록되므로, 이벤트가 발생한 시각에서 StartTime 변수의 값을 빼면 기록 시작 시각에서부터 얼마 뒤에 일어난 이벤트인지 알 수 있다.

시스템이 모달 상태에 들어가거나 빠져 나오는 경우에는 SysModalOn 전역 변수의 값을 설정하고, CallNextHookEx 함수를 호출하여 다음 후크를 실행한다.

또한, 처리할 후크 코드가 아닌 경우에도 다음 후크를 실행한다.

그러면, 이러한 매크로 기록 후크 함수를 설치하고 기록을 시작하게 하는 StartRecording 함수를 구현 하도록 하자.

이 함수의 구현 부분은 다음과 같다.

```

function StartRecording: Integer; stdcall;
begin
    Result := 0;
    if pMsgBuff <> nil then Exit;
    GetMem(PMsgBuff, Sizeof(TMsgBuff));
    if PMsgBuff = nil then Exit;
    SysModalOn := False;
    MsgCount := 0;
    StartTime := GetTickCount;
    TheHook := SetWindowsHookEx(WH_JOURNALRECORD, JournalRecordProc,
        hInstance, 0);
    if TheHook <> 0 then
        begin
            Result := 1;

```



```

Exit;
end
else
begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
end;
end;

```

먼저 기본적인 결과 값을 0 으로 설정하고, 메시지 버퍼를 검사해서 버퍼에 메모리가 할당되어 있으면 이는 메시지가 기록되어 있거나, 현재 기록 중임을 의미하므로 실행을 중지한다. 이상이 없으면 버퍼에 대한 메모리를 할당하고, 초기 전역 변수 값을 설정한다.

특히 StartTime 전역 변수의 값을 설정할 때에는 GetTickCount API 함수를 이용한다. 그리고, WH\_JOURNALRECORD 후크에 대한 후크 함수를 설치한다. 이때 설치가 성공적이면 0 이 아닌 값이 리턴되므로 결과 값으로 1 을 반환하고, 0 이 리턴되면 버퍼에 대한 메모리를 해제한다.

매크로의 기록을 중지하는 StopRecording() 함수는 구현이 다소 복잡하다.

단순히 후크를 해제하는 것만이 아니라 버퍼에 기록된 메시지를 디스크에 저장하는 역할을 해주어야 한다. 그렇기 때문에 파라미터로 기록할 파일 이름을 PChar 형으로 넘겨 받는다. 실행 결과에 따라서 반환되는 결과 값이 다양한데, 성공적으로 메시지가 기록되고 디스크에 저장된 경우에는 저장된 메시지의 수를, 저장할 메시지가 없을 때에는 -1, 후크 함수를 제거하는데 실패한 경우에는 -2 를 반환하며 I/O 에러가 발생한 경우에는 0 을 반환한다.

구현 부분은 다음과 같다.

```

function StopRecording(lpFileName: PChar): LongInt; stdcall;
var
    TheFile: File;
begin
    if PMsgBuff = nil then
    begin
        Result := -1;
        Exit;
    end;
    if UnHookWindowsHookEx(TheHook) = False then
    begin
        Result := -2;
    end;
end;

```

```

Exit;
end;
TheHook := 0;
if MsgCount > 0 then
begin
    Assign(TheFile, lpFileName);
    {$I-}
    Rewrite(TheFile, Sizeof(TEventMsg));
    {$I+}
    if IOResult <> 0 then
    begin
        FreeMem(PMsgBuff, Sizeof(TMsgBuff));
        PMsgBuff := nil;
        Result := 0;
        Exit;
    end;
    {$I-}
    Blockwrite(TheFile, PMsgBuff^, MsgCount);
    {$I+}
    if IOResult <> 0 then
    begin
        FreeMem(PMsgBuff, Sizeof(TMsgBuff));
        PMsgBuff := nil;
        Result := 0;
        {$I-}
        Close(TheFile);
        {$I+}
        if IOResult <> 0 then Exit;
        Exit;
    end;
    {$I-}
    Close(TheFile);
    {$I+}
    if IOResult <> 0 then
    begin
        FreeMem(PMsgBuff, Sizeof(TMsgBuff));

```

```

    PMsgBuff := nil;

    Result := 0;

    Exit;

end;

end;

FreeMem(PMsgBuff, Sizeof(TMsgBuff));

PMsgBuff := nil;

Result := MsgCount;

end;

```

소스가 다소 길지만 내용은 어렵지 않다. 그리고, 중복되는 부분이 있으므로 이를 단축하면 좋을 것이다. 특별히 설명할 만한 부분은 없지만 \$I 컴파일러 지시자(compiler directive)를 통해서 I/O 에러를 처리하는 부분에 대해서 잠시 알아보자.

디폴트로 표준 I/O 프로시저와 함수를 호출하면 자동으로 에러를 검사하게 되어 있다. 만약 에러가 발생하면 예외를 발생시키거나 예외 처리가 안될 경우 프로그램은 실행을 중단한다. 이런 자동 검사를 여부를 결정하는 컴파일러 지시자가 \$I 이다. {\$I-}에 의해 I/O 에러 자동 검사가 꺼지게 되는데, 이때에는 I/O 에러가 발생해도 예외가 발생하지 않는다. 그러므로, I/O 작업을 한 뒤에는 IOResult 함수를 호출해서 에러 여부를 알아보아야 한다. 에러가 발생하지 않은 경우에는 0 을 반환한다.

앞의 StopRecording() 함수 역시 이런 방법을 이용해서 I/O 에러를 처리한다.

이렇게 해서 매크로를 기록하는 부분에 대한 구현이 모두 끝났다.

이번에는 매크로를 재생하는 부분을 구현하도록 하자.

먼저 후크 함수로 사용될 JournalPlayback() 함수를 다음과 같이 구현한다.

기본적인 처리 방침은 Code 값에 따라서, HC\_SKIP 인 경우에는 다음 메시지를 처리하되, 더 이상의 메시지가 없으면 후크 함수를 제거한다. HC\_GENNEXT 는 현재 메시지를 처리한다. 그 밖의 자세한 내용은 앞에서 설명했으므로 생략하도록 하겠다.

눈 여겨 보아야 할 부분은 code 값이 HC\_SKIP 인 경우 후크를 제거할 때와 HC\_SYSMODALOFF 인 경우 메시지 재생을 마치는 경우로 이때에는 어플리케이션에서 넘겨준 콜백 함수를 cbPlaybackFinishedProc(cbAppData)와 같은 형태로 호출한다. 이 콜백 함수는 재생이 끝났음을 어플리케이션에 알리면 동작하게 되는 함수이다.

```

function JournalPlaybackProc(Code: Integer; wParam: TwParam; lParam: TlParam):
    Longint; stdcall;

var
    TimeToFire: Longint;

begin

```

```

Result := 0;
case Code of
  HC_SKIP:
  begin
    Inc(CurrentMsg);
    ReportDelayTime := True;
    if CurrentMsg >= (MsgCount-1) then
      if TheHook <> 0 then
        if UnHookWindowsHookEx(TheHook) = True then
          begin
            TheHook := 0;
            FreeMem(PMsgBuff, Sizeof(TMsgBuff));
            PMsgBuff := nil;
            cbPlaybackFinishedProc(cbAppData);
          end;
        end;
      end;
    Exit;
  end;
  HC_GETNEXT:
  begin
    PEventMsg(IParam)^ := PMsgBuff^[CurrentMsg];
    PEventMsg(IParam)^.Time := StartTime + PMsgBuff^[CurrentMsg].Time;
    if ReportDelayTime then
      begin
        ReportDelayTime := False;
        TimeToFire := PEventMsg(IParam)^.Time - GetTickCount;
        if TimeToFire > 0 then Result := TimeToFire;
      end;
    end;
    Exit;
  end;
  HC_SYSMODALON:
  begin
    SysModalOn := True;
    CallNextHookEx(TheHook, Code, wParam, lParam);
    Exit;
  end;
  HC_SYSMODALOFF:

```

```

begin
    SysModalOn := False;
    TheHook := 0;
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    cbPlaybackFinishedProc(cbAppData);
    CallNextHookEx(TheHook, Code, wParam, lParam);
    Exit;
end;
end;
if Code < 0 then
    Result := CallNextHookEx(TheHook, Code, wParam, lParam);
end;

```

마지막으로 Playback() 함수를 구현하도록 하자.

이 함수는 재생할 매크로 파일의 이름을 PChar 데이터 형인 파라미터로 넘겨 받고, 이와 함께 재생이 끝난 후 호출할 어플리케이션의 콜백 함수와 어플리케이션 데이터를 파라미터로 넘겨 받아 사용한다.

구현 부분은 다음과 같다.

```

function Playback(lpFileName: PChar; EndPlayProc: TcbPlaybackFinishedProc;
    AppData: Longint): Integer; stdcall;
var
    TheFile: File;
begin
    Result := 0;
    if PMsgBuff <> nil then Exit;
    GetMem(PMsgBuff, Sizeof(TMsgBuff));
    if PMsgBuff = nil then Exit;
    Assign(TheFile, lpFileName);
    {$I-}
    Reset(TheFile, Sizeof(TEventMsg));
    {$I+}
    if IOResult <> 0 then
        begin
            FreeMem(PMsgBuff, Sizeof(TMsgBuff));

```

```

    PMsgBuff := nil;
    Exit;
end;
{$I-}
MsgCount := FileSize(TheFile);
{$I+}
if IOResult <> 0 then
begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    {$I-}
    Close(TheFile);
    {$I+}
    if IOResult <> 0 then Exit;
    Exit;
end;
if MsgCount = 0 then
begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    {$I-}
    Close(TheFile);
    {$I+}
    if IOResult <> 0 then Exit;
    Exit;
end;
{$I-}
Blockread(TheFile, PMsgBuff^, MsgCount);
{$I+}
if IOResult <> 0 then
begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    {$I-}
    Close(TheFile);
    {$I+}

```

```

    if IOResult <> 0 then Exit;
    Exit;
end;
{$I-}
Close(TheFile);
{$I+}
if IOResult <> 0 then
begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    Exit;
end;
CurrentMsg := 0;
ReportDelayTime := True;
SysModalOn := False;
cbPlaybackFinishedProc := EndPlayProc;
cbAppData := AppData;
StartTime := GetTickCount;
TheHook := SetWindowsHookEx(WH_JOURNALPLAYBACK, JournalPlayBackProc,
    hInstance, 0);
if TheHook = 0 then
begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    Exit;
end;
Result := 1;
end;

```

소스가 다소 길지만, 많은 부분이 중복되고 StopRecording() 함수와 마찬가지로 I/O 에러 처리를 위해서 길어진 부분이 많기 때문이므로 이해하기는 쉬울 것이다.

일단 재생을 하기 위해 메모리 버퍼를 설정하고, 디스크에 있는 메시지를 버퍼에 읽어들이고 후 콜백 함수와 어플리케이션 데이터를 전역 변수에 저장한다. 그리고, 현재 시각을 GetTickCount API 함수를 통해서 알아낸 후 WH\_JOURNALPLAYBACK 후크에 대한 후크 함수를 설치한다. 재생이 성공적으로 이루어진 경우에는 1 을 반환한다.

이로써 DLL 파일의 구현이 모두 끝났다. 이를 어플리케이션에서 사용하려면 함수를

export 해야 한다. 이 부분은 다음과 같이 구현한다.

exports

```
JournalRecordProc index 1 name 'JOURNALRECORDPROC' resident,  
StartRecording index 2 name 'STARTRECORDING' resident,  
StopRecording index 3 name 'STOPRECORDING' resident,  
JournalPlayBackProc index 4 name 'JOURNALPLAYBACKPROC' resident,  
Playback index 5 name 'PLAYBACK' resident;
```

그리고, DLL 이 처음 시작되면 메시지 버퍼는 메모리 블록을 가리키면 안되므로 다음과 같은 코드를 삽입한다.

begin

```
PMsgBuff := nil;
```

end.

## ● 후크 DLL wrapper 의 작성

앞의 DLL 프로젝트를 컴파일 한 후, 맨 처음 예제와 같은 방식으로 어플리케이션에서 직접 DLL 을 적재해서 사용할 수도 있지만 보다 일반적이고, 쉽게 사용하기 위해서는 wrapper 유닛을 작성해서 사용하면 편리하다. Wrapper 유닛이란 DLL 의 함수를 쉽게 사용하기 위해서 파스칼 문법에 맞게 만든 인터페이스 유닛으로, 이를 이용하면 인터페이스 유닛의 이름을 사용하고자 하는 유닛의 uses 절에 추가하는 것으로 해결할 수 있다.

참고로, 델파이의 Win32 API 지원도 이러한 wrapper 유닛을 통해서 이루어 지는 것으로 windows.pas 등의 유닛이 대표적인 wrapper 유닛이다.

여기에 대해서는 DLL 을 다루는 장에서 더욱 자세하게 다루고 있으므로 이를 참고하기 바란다. 그러면, DLL 의 wrapper 유닛을 다음과 같이 작성한다.

unit HookExam;

interface

type

```
TwMsg = LongInt;
```

```
TwParam = LongInt;
```

```
TIParam = LongInt;
```



```
TcbPlaybackFinishedProc = procedure(AppData: Longint); stdcall;
```

```
function StartRecording: integer; stdcall;
```

```
function StopRecording(lpFileName: PChar): LongInt; stdcall;
```

```
function Playback(lpFileName: PChar; EndPlayProc: TcbPlaybackFinishedProc;  
    AppData: Longint): Integer; stdcall;
```

implementation

```
function StartRecording; external 'ExamLib4' index 2;
```

```
function StopRecording; external 'ExamLib4' index 3;
```

```
function Playback; external 'ExamLib4' index 5;
```

end.

## ● 예제 어플리케이션의 동작 방식

후크 DLL 을 사용하는 어플리케이션에서는 콜백 함수로 PlaybackFinished()를 이용하도록 하자. 이 콜백 함수는 매크로의 재생이 완료되었을 때 호출된다. 즉, DLL 의 Playback() 함수에 PlaybackFinished() 콜백 함수의 주소를 넘겨 주어서 Playback()이 완료되는대로 이를 호출하게 된다. 그런데, PlaybackFinished() 함수의 주소를 저장하기 위해서 프로그램 시작시에 전역 변수를 하나 선언하고 프로그램이 종료될 때 이를 해제하도록 한다.

폼은 매크로의 기록을 시작할 때 사용하는 ‘기록 시작’ 버튼, 기록을 중지할 때 사용하는 ‘기록 중지’ 버튼, 매크로를 재생할 때 사용하는 ‘재 생’ 버튼과 어플리케이션을 종료할 때 사용하는 ‘완 료’의 네 버튼으로 구성한다.

폼이 처음 생성될 때에는 ‘기록 시작’과 ‘완 료’ 버튼을 사용 가능하도록 하고, ‘기록 중지’와 ‘재 생’ 버튼은 기록을 중지시키거나, 재생할 것이 없으므로 이를 사용할 수 없도록 설정한다.

각 버튼의 역할은 다음과 같다.

### 1. 기록 시작:

더 이상의 메시지를 기록하거나 기록 중간에 어플리케이션을 종료하게 하면 안 되므로, ‘기록 시작’과 ‘완 료’ 버튼을 사용 불가능하게 설정하고, 동시에 ‘기록 중지’ 버튼은 사용이 가능하도록 설정한다. 그리고, DLL 의 StartRecording() 함수를 호출해서 메시지 기록을 시작한다. 이때 StartRecording() 함수가 에러 코드를 반환하면 사용자에게 에러 메시지를

보여주고, ‘기록 시작’ 버튼이 눌리기 전의 상태로 재설정한다.

## 2. 기록 중지:

매크로를 저장할 파일 이름을 파라미터로 DLL 의 StopRecording() 함수를 호출한다. 그리고 ‘완 료’와 ‘기록 시작’ 버튼을 사용 가능하도록 설정해서, 세션을 기록하거나 어플리케이션에서 빠져 나갈 수 있도록 허용한다. 또한, 기록이 성공적으로 이루어 졌을 경우에는 ‘재 생’ 버튼도 사용할 수 있도록 설정한다.

## 3. 재 생:

모든 버튼을 선택할 수 없도록 설정하고, 재생할 매크로가 저장된 파일 이름과 PlaybackFinished() 콜백 함수의 인스턴스 주소, 그리고 어플리케이션이 정의한 데이터를 받을 파라미터로 DLL 의 Playback() 함수를 호출한다.

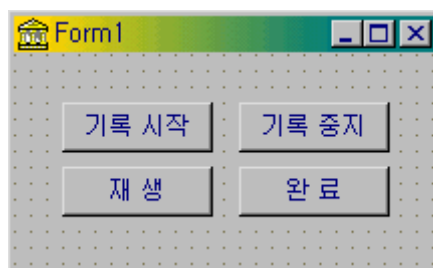
매크로의 재생이 완료되면 후크 함수는 어플리케이션의 PlaybackFinished() 콜백 함수를 호출해서 재생이 완료되었음을 알림과 동시에 메인 윈도우를 어플리케이션 데이터로 넘겨준다. 이때 DLL 의 Playback() 함수가 에러 코드를 반환하면, ‘재 생’ 버튼이 눌리기 이전의 상태로 버튼 들을 재설정한다.

PlaybackFininshed() 콜백 함수가 성공적으로 호출되면 ‘기록 시작’, ‘재 생’, ‘완 료’ 버튼을 사용 가능하도록 설정한다.

## 4. 완 료:

프로그램을 완료한다.

그러면, 예제 어플리케이션의 폼을 다음과 같이 디자인 하자.



폼의 각 버튼의 역할은 앞에서 간단히 설명하였다.

먼저 매크로를 기록, 재생할 파일 이름을 다음과 같이 상수로 선언하고, 앞에서 작성한 후크 DLL 을 사용하기 위해 uses 절에 HookExam 을 추가한다.

```
const
```

```
    FILENAME = 'Hooking.MAC';
```

그리고, 처음 폼이 생성될 때의 버튼 들의 초기 값을 다음과 같이 설정한다.

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    Button1.Enabled := True;
```

```
    Button2.Enabled := False;
```

```
    Button3.Enabled := False;
```

```
    Button4.Enabled := True;
```

```
end;
```

여기서 Button1 은 ‘기록 시작’, Button2 는 ‘기록 중지’, Button3 는 ‘재 생’, Button4 는 ‘완료’ 버튼이다.

각 버튼의 역할 역시 해당되는 DLL 함수를 호출하고, 각 버튼의 Enabled 프로퍼티를 조절하는 것으로 각각의 이벤트 핸들러는 다음과 같다.

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

```
begin
```

```
    Button2.Enabled := True;
```

```
    Button2.SetFocus;
```

```
    Button1.Enabled := False;
```

```
    Button3.Enabled := False;
```

```
    Button4.Enabled := False;
```

```
    if StartRecording = 0 then
```

```
    begin
```

```
        Button1.Enabled := True;
```

```
        Button1.SetFocus;
```

```
        Button2.Enabled := False;
```

```
        Button3.Enabled := False;
```

```
        Button4.Enabled := True;
```

```
        ShowMessage('기록을 시작할 수 없습니다 !');
```

```
    end;
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    if StopRecording(FILENAME) > 0 then
    begin
        Button1.Enabled := True;
        Button3.Enabled := True;
        Button3.SetFocus;
    end
    else
    begin
        Button1.Enabled := True;
        Button1.SetFocus;
        Button3.Enabled := False;
    end;
    Button2.Enabled := False;
    Button4.Enabled := True;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    Button1.Enabled := False;
    Button2.Enabled := False;
    Button3.Enabled := False;
    Button4.Enabled := False;
    if PlayBack(FILENAME, @PlaybackFinished, Handle) = 0 then
    begin
        Button1.Enabled := True;
        Button2.Enabled := False;
        Button3.Enabled := True;
        Button4.Enabled := True;
        Button3.SetFocus;
    end;
end;
```

```
procedure TForm1.Button4Click(Sender: TObject);
```

```
begin
    Close;
end;
```

간단하므로 이해에 별 무리가 없을 것으로 믿는다. 그런데 눈 여겨 보아야 할 부분은 Button3 의 OnClick 이벤트 핸들러에서 Playback() 함수를 호출하는 부분으로, 재생이 끝났을 때 이를 처리할 콜백 함수의 주소를 파라미터로 넘겨 주게 된다. 여기서는 PlaybackFinished 함수의 주소를 넘겨 주었다. 그러므로, 이 콜백 함수를 다음과 같이 작성한다.

```
procedure PlaybackFinished(AppData: LongInt); stdcall;
begin
    Form1.Button1.Enabled := True;
    Form1.Button2.Enabled := False;
    Form1.Button3.Enabled := True;
    Form1.Button4.Enabled := True;
    Form1.Button3.SetFocus;
end;
```

이 때 Form1 을 앞에 붙이는 이유는 이 함수가 TForm1 클래스의 메소드로 선언되지 않았기 때문이다.

어쨌든 후킹 DLL 에 의해 Playback() 함수가 실행되면, 이 함수가 종료되는 데로 PlaybackFinished() 프로시저가 실행되어 버튼 들의 활성화 여부를 결정하게 된다.

이제 이 프로그램을 실행시키면 매크로 기록, 재생을 할 수 있게 된다.

‘기록 시작’ 버튼을 누르고 마음대로 마우스와 키보드를 이용한 작업을 하고, ‘기록 중지’ 버튼을 누르면 이 메시지가 매크로로 기록이 될 것이며, 이를 ‘재 생’ 버튼을 눌러서 재현할 수 있다. 한 번 해보면 얼마나 유용하게 쓰일 수 있을 지 알 수 있을 것이다.

## 정 리 (Summary)

이번 장에서는 Win32 환경에서 자주 사용되는 콜백 함수에 대한 설명과 콜백을 이용하여 구현할 수 있는 후킹 기법에 대해서 알아 보았다. 콜백 함수는 후킹 기법 말고도 다른 곳에서도 자주 사용되므로, 확실하게 이해해 두는 것이 좋다.

## 구조화 저장소 기법

### (Structured Storage Technique)

만약 정해진 포맷의 파일 형식을 써야 하는 것이 아니라, 데이터를 저장할 때 대단히 유연하고도 강력한 방법이 존재한다면 얼마나 편리할까 ?

구조화 저장(structured storage)이라는 새로운 방식으로 이러한 문제를 해결할 수 있다. 구조화 저장은 DocFile 이나 OLE 복합 파일 (OLE compound file)이라는 이름으로도 불리고 있는 새로운 저장방식이다.

이 방식을 이용하면 Load 와 Save 를 할 때 파일의 일부분만을 활용할 수 있다. 약간의 데이터 블록 만이 필요할 때 전체 데이터를 모두 불러오거나, 저장해야 한다면 이는 상당히 비효율적이라고 말할 수 있다. 이를 이용하면 순차적이면서, 점진적인 데이터의 접근이 가능하다. 기본적으로 윈도우에 의해서 지원되는 방식이므로, 쉽게 정보를 얻을 수 있다.

이미 마이크로소프트에서는 이러한 구조화 저장 방식이 차세대 윈도우 제품에서는 디폴트 파일 포맷으로 사용할 것임을 공언하고 있다. 또한, 현재 MS 오피스 제품군에서는 이 기술을 적용하고 있기도 하다.

이 정도의 소개만으로 구조화 저장의 중요성은 충분히 알고도 남음이 있을 것이다. 그러면, 이제 실제로 이 기술에 대한 설명과 델파이에서 이를 어떻게 구현할 것인지에 대해서 알아보도록 하자.

#### 기본적인 이해

구조화 저장(structured storage)이라는 용어는 DocFile 이라는 용어와 혼용되고 있다. 사용의 편의성을 위해 지금 부터는 간단히 DocFile 이라고 지칭하도록 하겠다. DocFile 을 이해하는 가장 좋은 방법은 파일 내에 파일 시스템을 가지고 있는 것이라고 생각하면 된다. 즉, DocFile 에는 디렉토리와 파일 들을 가지고 있는 것이다.

예를 들어, Example.ole 라는 DocFile 이 있을 때 여기에 Version, Files 라는 디렉토리가 있으며 Files 라는 디렉토리 아래에 File1, File2, File3 와 같은 파일 들을 내부적으로 포함한다고 하자. 이때 File1 과 같은 DocFile 내부의 데이터 블록에 다른 데이터 블록에 전혀 영향을 주지 않고 접근하는 것이 가능하다.

참고로 앞으로 사용하는 용어 중에서 Storage 라는 용어는 DocFile 에서의 디렉토리와 동격으로 생각하면 되고, Stream 은 DocFile 에서의 파일로 생각하면 된다.

#### DocFile 생성 함수

DocFile 을 생성하는 함수는 StgCreateDocFile 이다. 이 함수는 델파이 3 의 activex.pas 유닛에 선언되어 있으며 선언부분은 다음과 같다.

```
function StgCreateDocfile(pwcsName: POleStr; grfMode: Longint;
    reserved: Longint; out stgOpen: IStorage): HRESULT; stdcall;
```

첫 번째 파라미터인 pwcsName 은 파일이름을 유니코드로(OLE 에서는 유니코드가 표준으로 사용된다.) 설정하면 되고, grfMode 에는 플래그를 설정하게 된다. 세 번째 파라미터는 현재는 사용되지 않기 때문에 보통 0 으로 설정하게 되며, 실제 사용하게될 IStorage 인터페이스가 네 번째 파라미터에서 넘어오게 된다.

하나의 DocFile 은 그 자체가 Storage 이다. 그렇기 때문에, 이 함수에 의해서 넘어오는 Storage 는 파일의 루트 저장소가 된다. 이 함수가 성공적으로 수행되었는지 여부를 검사할 때에는 SUCCEEDED() 함수를 사용한다. 그 Pseudo Code 를 아래에 들어 보았다.

```
Hr := StgCreateDocFile(...);
if (SUCCEEDED(Hr)) then ...;
```

보통 OLE 를 사용할 때 HRESULT 를 S\_OK 와 비교하는 경우가 많은데, 이 방법은 그다지 좋은 방법이 못된다. 그 이유는 OLE 가 함수가 성공적으로 수행되더라도 미묘하게 차이는 다른 여러가지 반환값을 가질 수 있기 때문이다. 그러므로, SUCCEEDED() 함수를 사용하는 것이 보다 효율적인 방안이 된다.

## 유니코드(Unicode)의 사용

앞의 함수 선언부분에서 언급했듯이 OLE 세계에서는 유니코드가 표준으로 사용된다. 그렇지만 지금까지의 프로그래밍 환경에서는 안시코드(AnsiCode)를 표준으로 사용해 왔기 때문에 다소간의 혼란이 있을 수 있다.

델파이 3 에서는 유니코드와 안시코드를 쉽게 변환할 수 있는 방법을 제공하기 때문에 이런 변화가 커다란 문제가 되지 않는다. WideString 문자열 데이터 형이 유니코드를 지원하게 된다. 그러면, 간단히 안시코드와 유니코드를 변환하는 몇 가지 방법에 대해 알아보도록 하자.

```
var
    s: string;
    ws: WideString;
begin
```

```

s := 'abc';
ws := s;
end;

```

어떤가 ? 너무나 단순하지 않은가 ? 그냥 일반 문자열을 위에서와 같이 WideString 형의 대입하는 것으로 모든 것이 끝난다. 마찬가지로 유니코드 문자열을 안시코드로 변환할 때에도 단순히 다음과 같이 하면 된다.

```

var
  s: string;
  ws: WideString;
begin
  ws := 'abc';
  s := ws;
end;

```

그렇지만, OLE 함수를 사용할 때에는 보통 WideString 데이터 형 보다는 PWideChar 데이터 형을 파라미터로 사용하기 때문에 이를 호출할 때에는 아래와 같은 방식으로 형변환시켜 사용하면 간단히 해결된다.

```

var
  ws: WideString;
begin
  ws := 'abc';
  SomeOLEFunction(PWideChar(ws));
end;

```

텔파이 2에서는 WideString 데이터 형을 지원하지 않고, PWideChar 데이터 형만을 지원하기 때문에 이를 안시코드 문자열과 호환시키기 위해서는 해당하는 문자열의 크기의 두배 만큼의 메모리를 할당받고, 실제로 문자를 유니코드로 바꾸기 위해서는 API 함수인 MultiByteToWideChar 함수를 호출해야 했다. 마찬가지로 유니코드 문자열을 안시코드로 변환시킬 때에도 메모리 할당과 WideCharToMultiByte API 함수를 사용해야 한다. 이 점이 텔파이 4가 얼마나 OLE/COM 환경에 적합한 형태로 바뀌었는지를 보여주는 단적인 예가 될 수 있다.



## Stream, Storage 이름의 제한

DocFile 시스템에서도 약간의 이름에 대한 제한을 가지고 있다. 31 자를 넘는 이름을 가질 수는 없으며, 이름에 '!', ':', '/', 'W' 등의 문자는 사용할 수 없다. 그리고 첫번째 문자는 ordinal 값이 32 이하인 문자가 되면 안된다. 이러한 문자들은 특수한 목적에 사용되게 된다.

## STGM 상수

STGM 상수는 storage, stream 인터페이스에서 객체에 대한 접근 모드나 객체를 실제로 생성, 삭제 등을 하게 되는 조건을 정의하고 있다. 이들 상수에 대해서 알아보도록 하자.

- STGM\_READ, STGM\_WRITE, STGM\_READWRITE

Stream 객체에 대해서는 어떤 메소드를 허용할 것인지를 결정하는 상수이다. 예를 들어, STGM\_READ 는 IStream 의 Read 메소드를 허용하게 된다. Storage 객체에 대해서는 가능한 요소를 나열하고, 이들을 open 한다. STGM\_WRITE 는 객체를 저장할 수 있도록 하며, STGM\_READWRITE 는 STGM\_READ 와 STGM\_WRITE 를 혼합한 것이다.

- STGM\_SHARE\_DENY\_NONE, STGM\_SHARE\_DENY\_READ,  
STGM\_SHARE\_DENY\_WRITE, STGM\_SHARE\_EXCLUSIVE

STGM\_SHARE\_DENY\_NONE 은 어떤 객체를 open 하더라도 이것이 그 객체에 대한 read, write 접근에 대한 제한을 가지지 않는 것을 의미한다.

STGM\_SHARE\_DENY\_READ 는 open 한 객체에 대해서 STGM\_READ 모드로 접근할 수 없도록 제한한다. 주로 root storage 객체에 대해 사용된다. STGM\_SHARE\_DENY\_WRITE 는 STGM\_WRITE 모드로 접근할 수 없도록 제한하는데, 이것은 여러 명의 사용자가 객체에 접근했을 때 생길 수 있는 문제점을 해결할 수 있다.

STGM\_SHARE\_EXCLUSIVE 는 STGM\_READ, STGM\_WRITE 모드 둘 다 접근할 수 없도록 하는 값이다.

- STGM\_DIRECT, STGM\_TRANSACTED

Direct 모드에서는 storage 요소에 대해서 변화가 일어날 경우 이 값이 그대로 반영된다. 이 모드가 디폴트로 되어 있다. Transacted 모드에서는 변화가 일어날 경우 그 값이 버퍼에 저장되었다가 commit 이 호출될 때 객체에 반영된다. 만약 IStream, IStorage 인터페이스

이스에서 Revert 메소드가 호출되면 이러한 변화가 무시된다. 그러나, 이 모드는 현재 OLE에서는 구현되지 않고 있다. 아마도 조만간에는 이것이 지원될 것으로 생각된다.

- STGM\_CREATE, STGM\_CONVERT, STGM\_FAILIFTHHERE

STGM\_CREATE 는 현재 존재하는 storage, stream 객체가 새로운 객체가 생성될 때에는 반드시 제거되어야 한다는 것을 지정한다. 만약 현재의 객체가 성공적으로 제거되지 않으면 새로운 객체가 생성되지 않는다.

STGM\_CONVERT 플래그는 현재 존재하는 stream 의 데이터를 보존하면서 새로운 객체를 생성하는데, 이때 이전 객체의 데이터는 CONTENTS 라는 객체에 보존된다. 이때 과거의 storage 객체에 있던 정보는 stream 의 형태로 변경되어 보존되므로, storage 의 계층 구조 정보는 망실된다.

STGM\_CONVERT 플래그는 디스크에 storage 객체를 생성하려고 하는데, 이미 그런 파일 이름이 존재하거나, Storage 객체 내부에 새로운 stream 을 생성하려고 하는데 같은 이름의 stream 이 있을 경우 등에서 사용하게 된다.

- STGM\_FAILIFTHHERE

이 플래그는 만약 지정된 이름의 객체가 있을 경우에 생성 과정을 취소하는 역할을 해준다. 이 경우에 STG\_E\_FILEALREADYEXISTS 상수가 반환된다.

- STGM\_PRIORITY

이 플래그가 지정되면 현재 우선권을 가진 사용자 만이 객체의 변화를 줄 수 있다. 이를 이용하면 다른 사용자들은 이 객체에 접근해도 이를 변화시킬 수 없게 된다. 이 경우에는 반드시 STGM\_DIRECT, STGM\_READ 가 설정되어 있어야 한다.

- STGM\_DELETEONRELEASE

이 플래그는 임시 파일을 사용할 때 유용하게 쓰이는 것으로, 부모 storage 객체가 해제되면 자동으로 그 아래의 파일 들이 파괴되도록 지정하는 것이다.

## DocFile 을 만들어 보자.

앞에서 설명한 StgCreateDocFile 함수를 사용해서 실제로 DocFile 을 만들어 보기로 하자. 이미 함수 선언과 파라미터에 대해서 간단한 설명을 했지만, 다시 한번 정리해 보자.

함수의 선언부는 다음과 같다.

```
function StgCreateDocfile(pwcsName: POleStr; grfMode: Longint;  
    reserved: Longint; out stgOpen: IStorage): HRESULT; stdcall;
```

그리고, 각 파라미터에는 다음과 같은 내용들을 설정하게 된다.

1. pwcsName: 유니코드 형식의 실제 파일 명. (예) 'c:\WtempWexample.ole'
2. grfMode: STGM 플래그가 설정된다. (예) STGM\_CREATE or STGM\_READWRITE  
STGM\_DIRECT or STGM\_SHARE\_EXCLUSIVE
3. reserved: 0
4. stgOpen: 실제로 storage 를 담게 될 레퍼런스 파라미터

그럼, 이제 실제 DocFile 을 만드는 프로시저를 하나 만들어 보자.

```
procedure Create;  
var  
    Hr: HRESULT;  
    Root: IStorage;  
begin  
    Hr := StgCreateDocFile('c:\WTempWExample1.ole', STGM_CREATE or STGM_READWRITE or  
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, Root);  
    if (SUCCEEDED(Hr)) then begin end else begin end;  
end;
```

아무 것도 하지 않고, Root 라는 IStorage 인터페이스만 받아오는 프로시저가 완성되었다.  
사용법이 그다지 어렵지는 않다는 것을 쉽게 알 수 있을 것이다.

마찬가지로 DocFile 을 여는 것도 그다지 어렵지 않다. 이때에는 StgOpenStorage 라는  
API 함수를 사용하는데, 이는 DocFile 자체가 root storage 이기 때문이다. 이 API 의 사  
용법도 거의 유사하므로 여기에서 간단히 소개하겠다.

```
procedure OpenDocFile;  
var  
    Hr: HRESULT;  
    Root: IStorage;  
begin
```

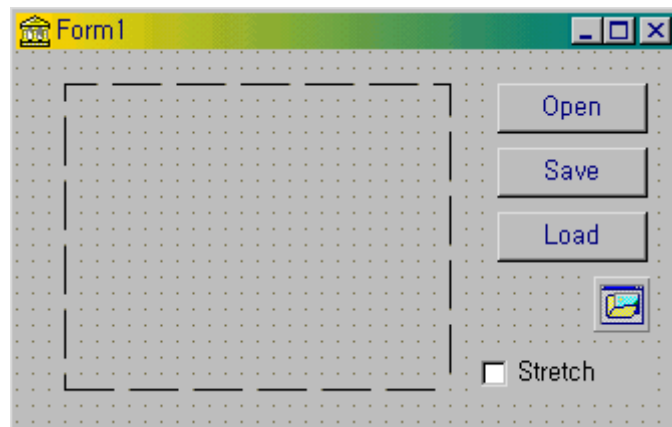
```

Hr := StgOpenStorage('c:\Temp\WExample1.ole', nil, STGM_READWRITE or
    STGM_DIRECT or STGM_SHARE_EXCLUSIVE, nil, 0, Root);
if (SUCCEEDED(HR)) then begin end else begin end;
end;

```

## DocFile 기법을 사용한 최초의 어플리케이션

그럼 이제, DocFile 을 이용해서 그림을 저장하고 불러올 수 있는 어플리케이션을 하나 만들어 보기로 하자. 새로운 어플리케이션을 하나 시작하고, 다음 그림과 같이 폼위에 버튼 3 개와 이미지 컴포넌트, 체크 박스, TOpenPictureDialog 대화상자 컴포넌트를 하나씩 없어서 디자인하도록 하자.



이때 각 버튼의 캡션을 Open, Save, Load 로 정하고, 체크 박스에는 Stretch 라고 캡션을 정하도록 한다. 이제 간단하게 이 어플리케이션에 대해서 설명하면, 구조화 저장 방법을 이용하는 방법을 익히기 위한 어플리케이션으로 Open 버튼을 누르면 대화상자를 띄워서, 아무 그림 파일이나 선택하게 하고, 이를 이미지 컴포넌트에 보여준다. 이때 'Stretch' 체크 박스가 체크되어 있을 경우에는 이미지를 Stretch 해서 보여준다. 그리고, 보여주는 이미지를 'c:\Temp\WExam1.ole' 파일에 저장할 때에는 Save 버튼을 누르고, 저장된 이미지를 불러올 때에는 Load 버튼을 누르게 한다.

유닛의 implementation 섹션에 우리가 사용하게 될 activex.pas 와 AxCtrls.pas 유닛을 uses 문장에 추가한다.

먼저 체크 박스의 OnClick 이벤트 핸들러를 아래와 같이 작성해서, 이미지 컴포넌트의 Stretch 속성에 반영할 수 있도록 하자.

```

procedure TForm1.CheckBox1Click(Sender: TObject);
begin

```

```
Image1.Stretch := CheckBox1.Checked;
end;
```

그리고, Open 버튼의 OnClick 이벤트를 핸들러를 작성하자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if OpenPictureDialog1.Execute then
        Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

이제 실제로 Save 버튼을 클릭하면 그림이 DocFile로 저장되도록 해야 한다. 이렇게 하려면 IStorage에 IStream 스트림을 하나 생성하고 이 스트림에 내용을 기록해야 한다. 스트림을 생성하는 코드는 IStorage 인터페이스의 CreateStream 메소드를 사용한다. 이 메소드의 파라미터로 첫번째 파라미터에 스트림의 이름과 두번째에 STGM 플래그를 설정하고, 마지막 파라미터로 생성된 IStream 인터페이스가 저장될 변수를 지정한다. 즉, 다음과 같은 코드를 사용한다.

```
Hr := Root.CreateStream('ExamStream', STGM_CREATE or STGM_READWRITE or
    STGM_DIRECT or STGM_SHARE_EXCLUSIVE), 0, 0, Stream);
```

그러면, 이렇게 Stream이라는 변수에 IStream 인터페이스를 담아오게 되면 실제 데이터를 여기에 저장해야 한다. 저장하는 방법은 크게 두가지가 있는데, 첫번째 방법은 IStream의 메소드를 직접 이용하는 것이고, 두번째 방법은 AxCtrls.pas 유닛에서 제공하는 TOleStream 클래스를 이용하는 것이다.

텔파이 3에서는 TOleStream 클래스를 이용해서 이 작업을 아주 쉽게 할 수가 있다. 사용법은 아래와 같이 아주 간단하다.

```
OleStream := TOleStream.Create(Stream);
Image1.Picture.SaveToStream(OleStream);
OleStream.Free;
```

이제 Save 버튼의 OnClick 이벤트를 핸들러를 제작해 보자.

```
procedure TForm1.Button2Click(Sender: TObject);
var
```

```

    Hr: HRESULT;
    Stream: IStream;
    OleStream: TOleStream;
    Root: IStorage;
begin
    Hr := StgCreateDocFile( 'c:\Temp\WExample1.ole', STGM_CREATE or STGM_READWRITE or
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, Root);
    if (not SUCCEEDED(Hr)) then Exit;
    Hr := Root.CreateStream('ExampleStream', STGM_CREATE or STGM_READWRITE or
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, 0, Stream);
    if (not SUCCEEDED(Hr)) then Exit;
    OleStream := TOleStream.Create(Stream);
    Image1.Picture.Graphic.SaveToStream(OleStream);
    OleStream.Free;
end;

```

이제는 저장된 이미지를 스트림을 통해서 읽어올 차례이다. 읽을 때에도 쓸 때와 마찬가지로 IStream 의 메소드를 직접 이용하는 방법과 TOleStream 클래스의 메소드를 사용하는 방법이 있다. 먼저 IStream 의 Read 메소드를 사용하는 방법에 대해 알아보자. 이 메소드는 파라미터를 3 개 사용한다. 첫번째 파라미터에는 데이터를 저장할 버퍼를, 두번째 파라미터에는 읽어올 데이터의 크기(바이트), 세번째 파라미터에는 실제로 읽어들이는 데이터의 크기가 넘어온다.

그러므로, 데이터를 읽어 들이기에 앞서 읽어올 데이터의 크기를 알아야 한다. 보통 스트림에 지금과 같이 하나의 데이터를 저장한 경우에는 스트림의 크기가 읽어올 데이터의 크기가 된다. 그러면 스트림의 크기를 알아볼 수 있는 함수에 대해서 알아보자

```

function GetStreamSize(Stream: IStream): LongInt;
var
    Hr: HRESULT;
    StatStg: TStatStg;
begin
    Hr := Stream.Stat(StatStg, STATFLAG_NONAME);
    if (not SUCCEEDED(Hr)) then
    begin
        Result := -1;
        Exit;
    end;
end;

```

```

end;

Result := Round(StatStg.cbSize);

end;

```

IStream 인터페이스의 Stat 메소드를 이용하면 현재의 스트림에 대한 정보를 TStatStg 클래스에 담아 준다. 위에서 Stat 메소드에 대한 파라미터로 사용한 STATFLAG\_NONAME 플래그는 스트림의 이름은 담아오지 말라고 지정한 것인데, 사실 이 경우에는 이름을 사용할 이유가 없기 때문에 이 플래그를 지정함으로써 쓸데 없는 메모리의 낭비를 막을 수가 있다. TStatStg 클래스의 cbSize 멤버에 스트림의 크기가 저장되어 있으므로 이를 정수형으로 바꾸어 그 값을 반환한다.

그리고 나서 IStream 인터페이스를 담고 있는 Stream 이라는 변수가 있다고 하면 아래와 같이 사용하면 된다.

```
Stream.Read(pBuffer, GetStreamSize(Stream), ReadBytes);
```

그러나, 보통의 경우에는 TOleStream 을 이용하는 것이 훨씬 편리하다. 사용법도 데이터를 쓸데와 거의 유사하므로, 아래의 코드를 살펴보면 금방 이해할 수 있을 것이다.

그러면 Load 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하자.

```

procedure TForm1.Button3Click(Sender: TObject);
var
    Hr: HRESULT;
    Stream: IStream;
    OleStream: TOleStream;
    Root: IStorage;
begin
    Hr := StgOpenStorage('c:\Temp\WExample1.ole', nil, STGM_READWRITE or
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, nil, 0, Root);
    if (not SUCCEEDED(Hr)) then Exit;
    Hr := Root.OpenStream('ExampleStream', nil, STGM_READWRITE or
        STGM_DIRECT or STGM_SHARE_EXCLUSIVE, 0, Stream);
    if (not SUCCEEDED(Hr)) then Exit;
    OleStream := TOleStream.Create(Stream);
    if (OleStream.Size > 0) then Image1.Picture.Graphic.LoadFromStream(OleStream);
    OleStream.Free;
end;

```

자 이제 첫번째 구조화 저장기법을 이용한 어플리케이션이 완성되었다. 아직 기능이 많은 것은 아니지만 기본적인 테크닉을 익히는 데에는 유용했을 것으로 생각한다. 그럼 이제 더 기능이 많은 두번째 어플리케이션을 제작해 보도록 하자.

## 파일 뷰어의 제작

이번에는 DocFile 의 내부를 들여다 볼 수 있는 뷰어를 제작해 보자. 이를 구현하기 위해서 IStorage 인터페이스의 EnumElements 메소드를 사용하게 되는데 이 메소드의 속도가 다소 느린 것이 흠이다.

EnumElements 메소드의 선언부는 다음과 같다.

```
function EnumElements(reserved1: Longint; reserved2: Pointer; reserved3: Longint;  
    out enm: IEnumStatStg): HRESULT; stdcall;
```

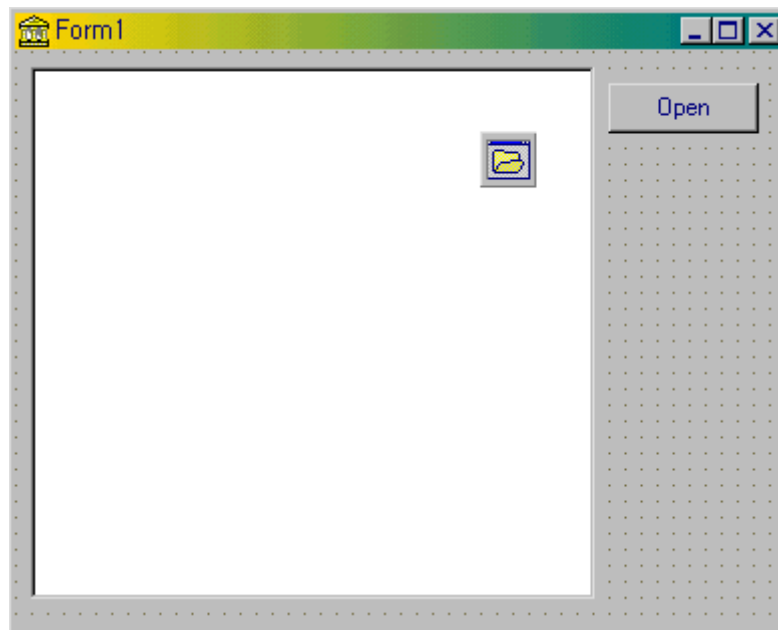
즉, 마지막 파라미터에 적절한 IEnumStatStg 인터페이스 형의 변수를 넣어주면 여기에 정보를 담아서 오게 된다. 이렇게 일단 IEnumStatStg 인터페이스를 받아오면 이 인터페이스의 Next 메소드를 사용해서 각각의 하부 요소 들을 얻을 수 있게 된다. Next 메소드는 아래와 같이 선언되어 있다.

```
function Next(celt: Longint; out elt: pceltFetched: PLongint): HRESULT; stdcall;
```

첫번째 파라미터에는 받아올 아이템의 수, 두번째 파라미터에는 실제로 받아오게 될 TStatStg 클래스 형의 변수를 지정하고, 세번째 파라미터에는 실제로 넘어온 아이템의 수가 반환된다.

이 함수들을 이용해서 워드나 엑셀 등의 내부적인 저장이 어떤 식으로 되어있는지 들여다 볼 수 있는 간단한 뷰어를 제작해보자.





먼저 앞의 그림과 같이 새로운 어플리케이션을 시작하고 폼에 트리뷰 컨트롤 하나와 TOpenDialog 대화상자 하나, 그리고 버튼을 하나 올려 놓자. 버튼의 캡션을 'Open' 으로 설정한다.

그리고, uses 절에 Activex.pas, ComObj.pas 유닛을 추가하고, 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

일단 파일 이름을 유니 코드 형식으로 해야 하기 때문에, WideString 형식으로 선언한 변수에 파일 이름을 집어 넣고, 이를 PWideChar 로 형 변환해서 사용한다.

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

```
var
```

```
    ws: WideString;
```

```
    Hr: HRESULT;
```

```
    Root: IStorage;
```

```
begin
```

```
    if not OpenDialog1.Execute then Exit;
```

```
    TreeView1.Items.Clear;
```

```
    ws := OpenDialog1.FileName;
```

이제는 이 파일에 실제로 DocFile 형식인지 검사해서 그렇다면 그 파일의 정보를 나타내야 한다. 이를 위해서 StgIsStorageFile 이라는 함수를 사용하는데, 이 함수의 파라미터로 유니코드 형식의 파일 이름을 넘겨 주게 되고 만약 DocFile 형식이라면 S\_OK 가 반환된다.

```

if (StgIsStorageFile(PWideChar(ws)) <> S_OK) then
begin
    ShowMessage('DocFile 형식이 아닙니다.');
```

```

    Exit;
```

```

end;
```

지정된 파일이 DocFile 형식이므로 StgOpenStorage 함수를 사용해서 DocFile 을 열고 루트를 앞에서 선언한 Root 라는 IStorage 형의 변수에 담아온다.

```

Hr := StgOpenStorage(PWideChar(ws), nil, STGM_READWRITE or STGM_DIRECT or
    STGM_SHARE_EXCLUSIVE, nil, 0, Root);
```

```

if not SUCCEEDED(Hr) then
```

```

begin
```

```

    ShowMessage('파일을 열 수 없습니다 !');
```

```

    Exit;
```

```

end;
```

성공적으로 파일을 열고, Root 를 받아왔으면 파일의 이름을 트리뷰 컴포넌트의 루트로 추가한다.

```

TreeView1.Items.Add(nil, ws);
```

이제 Enumeration 을 이용해서 트리 뷰에 하부 요소의 이름 들을 추가해 나가자. DocFile 의 구조는 Storage 와 Stream 으로 이루어진 여러 단계의 디렉토리 형태를 가지고 있기 때문에 이를 모두 탐색할 수 있도록 하려면 재귀적 호출을 할 수 있어야 하므로 독자적인 프로시저를 하나 정의해야 한다.

이 프로시저를 Enumeration 이라고 정의하고, 파라미터로 IStorage 인터페이스와 트리 뷰의 해당 아이템을 지정하도록 하자. 그러면 아래와 같이 선언될 것이다.

```

procedure TForm1.Enumeration(Storage: IStorage; ANode: TTreeNode);
```

일단 Button1 의 OnClick 이벤트 핸들러에서는 Root 와 트리 뷰의 첫번째 노드를 파라미터로 해서 이 프로시저를 호출하고, 트리 뷰를 펼쳐 보이면 모든 작업이 끝난다.

```

Enumeration(Root, TreeView1.Items[0]);
```

```

TreeView1.FullExpand;
```

end;

남은 작업은 Enumeration 프로시저를 구현하는 것이다. 일단 다음과 같이 프로시저와 사용할 변수를 선언하고, IEnumStatStg 인터페이스를 Enum 변수에 담아 온다.

```
procedure TForm1.Enumeration(Storage: IStorage; ANode: TTreeNode);
```

```
var
```

```
    Hr: HRESULT;
```

```
    Enum: IEnumStatStg;
```

```
    SubNode: TTreeNode;
```

```
    StatStg: TStatStg;
```

```
    SubStor: IStorage;
```

```
    HrSubStor: HRESULT;
```

```
    NumFetched: integer;
```

```
begin
```

```
    Hr := Storage.EnumElements(0, nil, 0, Enum);
```

```
    OleCheck(Hr);
```

OleCheck 프로시저는 ComObj.pas 유닛에 선언되어 있는데 결과 코드가 에러에 해당되면 EOleSysError 예외를 발생시킨다.

EnumElements 메소드를 이용해서 IEnumStatStg 인터페이스를 Enum 변수에 담아 왔으면 이 인터페이스의 Next 메소드를 사용해서 TStatStg 클래스 형으로 선언된 StatStg 변수에 정보를 담아 오도록 하자. 이때 Next 메소드의 마지막 메소드에는 실제로 넘어온 아이템의 수가 정수의 포인터 형으로 반환되므로 다음과 같이 사용한다.

```
repeat
```

```
    Hr := Enum.Next(1, StatStg, @NumFetched);
```

```
    if Hr <> S_OK then continue;
```

이제는 StatStg 변수의 dwType 정보에 따라서 다르게 대응해야 한다. 하부 요소가 Storage 라면 일단 Storage 의 이름이 담겨 있는 pwcsName 필드를 이용해서 트리뷰 컴포넌트에 노드를 하나 추가한다. 그리고 나서, Storage 를 다시 열어서 그 Storage 의 SubStorage 를 얻고, 이 SubStorage 에 해당되는 아이템과 추가할 트리 노드를 가지고 Enumeration 프로시저를 재귀 호출한다.

```
case StatStg.dwType of
```

```
STGTY_STORAGE:
```

```
begin
```

```
    SubNode := TreeView1.Items.AddChild(ANode, StatStg.pwcsName);
```

```
    HrSubStor := Storage.OpenStorage(StatStg.pwcsName, nil, STGM_READWRITE  
        or STGM_DIRECT or STGM_SHARE_EXCLUSIVE, nil, 0, SubStor);
```

```
    if SUCCEEDED(HrSubStor) then Enumeration(SubStor, SubNode);
```

```
end;
```

하부 요소가 Stream 이라면 스트림의 이름을 트리뷰에 추가하기만 하면 된다.

```
STGTY_STREAM:
```

```
    TreeView1.Items.AddChild(ANode, StatStg.pwcsName);
```

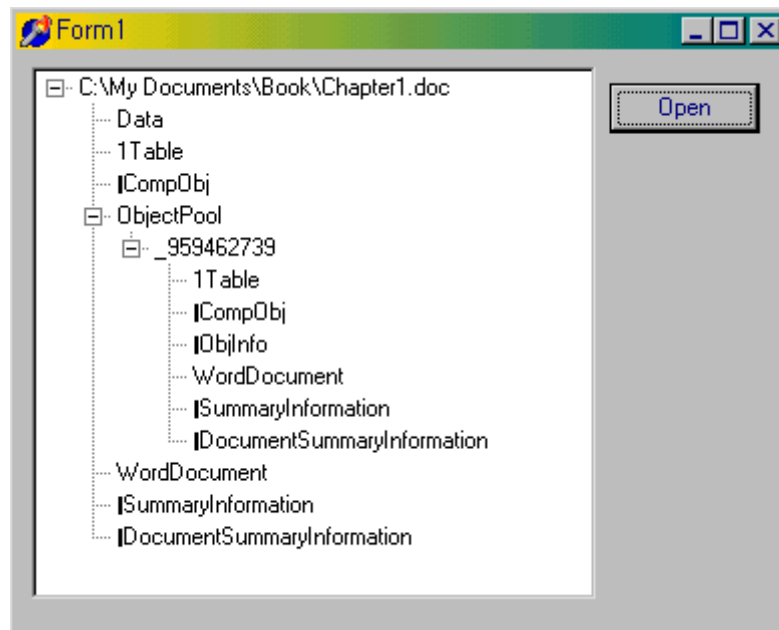
```
end;
```

이를 계속 반복하다가 마지막에 가면 S\_OK 가 반환되지 않으므로, 이때 루프를 종료하면 된다.

```
until (Hr <> S_OK);
```

```
end;
```

이것으로 두번째 어플리케이션이 완성되었다. 이제 실제로 이를 컴파일해서 실행하고, MS 오피스 파일 등을 열어서 하나의 파일에 어떤 형식으로 내용들이 구조적으로 저장되어 있는지 살펴보도록 하자. 아래의 그림은 오피스의 CommonDB.xls 파일을 열어본 것이다. 이를 보면 실제로 저장되는 내용이 하나의 파일에 여러 개의 Stream 과 Storage 로 이루어져 있다는 것을 알 수 있다.



## 정 리 (Summary)

이번 장에서는 새로운 파일 저장 방식으로 사용되고 있는 구조화 저장소 기법에 대해서 알아보았다. 이와 같은 구조화 저장소 기법을 이용해서 파일을 저장하면 파일 하나에 여러 내용을 분류하여 관리할 수 있다.

구조화 저장소 기법을 이용한 파일 저장 방식 역시 이제는 표준으로 정착되어 가고 있다. 그러므로 Stream 과 Storage 를 이용하는 방법에 대해서는 잘 익혀두는 것이 도움이 될 것이다.

## 윈도우 NT 서비스 어플리케이션의 제작

윈도우 NT 서버를 기반으로 한 클라이언트/서버 어플리케이션을 개발하다 보면, 컴퓨터에 관리자가 로그인하지 않은 상황에서도 서버 어플리케이션을 동작할 필요가 있다. 이럴 때에 사용되는 것인 NT 서비스 어플리케이션이다. 시작 프로그램 그룹에 서버 어플리케이션을 등록한 경우에는 사용자의 로그인이 있어야만 시작이 되지만, 서비스 어플리케이션이 윈도우 NT가 기동만 하면 로그인 여부와 관계없이 사용이 가능하다.

델파이 4에서는 이러한 윈도우 NT 서비스 어플리케이션을 쉽게 작성할 수 있는 클래스들과 위저드가 제공된다.

이번 장에서는 델파이 4에서 제공되는 위저드를 이용하여 윈도우 NT 서비스 어플리케이션을 작성하는 방법에 대해서 알아볼 것이다.

### 서비스 어플리케이션 위저드

Win32 서비스를 구현하는 어플리케이션을 제작하려면, File|New 메뉴를 선택하고 객체 저장소의 New 탭에서 Service Application 아이템을 더블 클릭하면 된다. 이렇게 하면 TServiceApplication 클래스 형의 Application 변수가 프로젝트에 추가될 것이다.

일단 이렇게 서비스 어플리케이션을 생성하면 서비스에 해당되는 윈도우가 나타날 것이다. 서비스 자체는 TService 클래스를 상속하여 구현하게 되는데, 오브젝트 인스펙터에서 이 객체의 프로퍼티를 설정하고, 이벤트 핸들러를 작성하면 된다.

서비스 어플리케이션에 추가적인 서비스를 지원하게 하려면 File|New 메뉴의 New 탭에서 Service 아이템을 더블 클릭하여 하나의 어플리케이션에 여러 개의 서비스를 지원하게 할 수도 있다.

### 서비스 관련 클래스

서비스 관련 클래스로 델파이에서 제공되는 것으로는 TServiceApplication, TService, TDependency 클래스가 있다.

#### ● TServiceApplication

TServiceApplication 클래스는 윈도우 NT 서비스 어플리케이션을 캡슐화한 클래스이다. 이 클래스는 서비스 어플리케이션에 가장 기본적인 기능을 제공한다. 각각의 서비스 프로젝트는 자동으로 어플리케이션의 인스턴스로 TServiceApplication 클래스 형인 Application 변수를 선언하게 한다.

TServiceApplication 클래스는 윈도우 NT 서비스를 캡슐화한 TService 객체 들을 포함하는데, 서비스 어플리케이션 객체에는 서비스 객체를 생성, 설치, 등록, 디스패치, 제거하는 메소드를 제공한다.

- TService

TService 클래스는 윈도우 NT 서비스 어플리케이션에서 NT 서비스를 캡슐화한다. Win32 서비스는 SCM(Service Control Manager)에 의해 접근할 수 있으며, 시스템이 시작할 때 자동으로 서비스를 개시하거나, 사용자가 서비스 제어판 애플릿을 이용하여 수동으로 시작하게 할 수 있으며, Win32 기반의 어플리케이션에서 서비스 함수를 사용할 때 시작하게 할 수도 있다.

- TDependency

TDependency 객체는 새로운 서비스나 TService 객체에 의해 구현된 서비스가 시작되기 전에 시작해야 하는 ordering 그룹을 적재하는 역할을 한다.

Dependency 객체 들의 컬렉션은 TDependencies 객체이며, TService 클래스는 Dependencies 와 LoadGroup 프로퍼티를 이용하여 이 컬렉션을 이용할 수 있다. 각각의 dependency 객체의 Name 프로퍼티는 서비스 또는 ordering 그룹의 이름을 결정하며, IsGroup 프로퍼티를 이용하여 이것이 서비스인지 그룹인지 나타내게 된다.

## 서비스 쓰레드

각각의 서비스는 자신의 쓰레드를 가지고 있다. 이런 쓰레드는 TServiceThread 클래스에서 상속받은 객체인데, 서비스 어플리케이션이 하나 이상의 서비스를 구현한 경우에는 서비스가 쓰레드에 안전하도록 작성해야 한다는 것을 잊어서는 안된다.

TServiceThread 클래스는 TService 클래스의 OnExecute 이벤트 핸들러를 작성함으로써 구현될 수 있게 디자인 되었다. 그러므로, 서비스 쓰레드는 새로운 요구를 처리하기 전에 서비스의 OnStart 와 OnExecute 이벤트 핸들러를 호출하는 루프를 포함한 자신의 Execute 메소드를 가지고 있다.

서비스는 처리하는데 비교적 오랜 시간이 걸리며, 동시에 여러 요구를 하나 이상의 클라이언트에서 받을 가능성이 많기 때문에, 이럴 때에는 각각의 요구에 대해서 TThread 클래스에서 상속받은 새로운 쓰레드를 사용하는 것이 효과적이다. 그리고, 이런 경우에는 새로운 쓰레드의 Execute 메소드를 구현해야 한다.

## 서비스 name 프로퍼티

TService, TDependency 클래스와 같이 서비스 어플리케이션을 작성하는데 제공되는 VCL 클래스를 이용할 때에는 name 프로퍼티에 대해서 잘 알아둘 필요가 있다.

서비스는 Service Start names 라고 하는 사용자 이름을 가지고 있는데, 여기에는 패스워드가 연관되어 있다. 이 이름은 서비스 관리자에 표시되며, 에디터 윈도우나 서비스의 실제 이름으로 사용된다.

그에 비해, Dependency 는 서비스일 수도 있고, 그룹으로 적재될 수도 있다. 문제는 dependency 도 이름과 display 이름을 가진다는 점이다. 그 밖에도 서비스 객체 역시 TComponent 클래스에서 상속받은 객체이기 때문에 Name 프로퍼티를 가지고 있다.

이들의 차이점에 대해서 간단히 정리해 보도록 하자.

- TDependency 클래스의 DisplayName 프로퍼티

TDependency 의 DisplayName 프로퍼티는 프로퍼티 에디터 윈도우에 표시되는 dependent 서비스의 이름을 나타낸다. TDependency 클래스의 Name 프로퍼티와 거의 동일한 역할을 한다.

- TService 클래스의 Name 프로퍼티

TService 의 Name 프로퍼티는 TComponent 에서 상속된 프로퍼티로, 컴포넌트의 이름이 되는 동시에 서비스의 이름이 된다. 만약 dependency 가 서비스이면 TDependency 클래스의 Name, DisplayName 프로퍼티와 같은 역할을 한다. 주의할 것은 TService 클래스에도 DisplayName 프로퍼티가 있는데, 이 프로퍼티는 완전히 다른 것이므로 주의하기 바란다.

## 서비스 관련 클래스의 주요 프로퍼티와 메소드

예제 어플리케이션을 작성하기에 앞서 윈도우 NT 서비스 어플리케이션을 작성하기 쉽도록 제공되는 텔파이 클래스의 주요 프로퍼티와 메소드에 대해서 알아보도록 하자.

- TServiceApplication 클래스

TServiceApplication 에서 중요한 프로퍼티로는 ServiceCount, Title 이 있다.

ServiceCount 프로퍼티는 서비스 어플리케이션의 서비스 수를 나타낸다. 각각의 서비스는 TService 인스턴스이며 자신의 서비스 쓰레드를 가지고 있다. Title 프로퍼티는 서비스 어



플리케이션의 이름을 가리키게 된다.

또한 중요한 메소드로는 CreateForm, Initialize, Run 메소드가 있다.

CreateForm 메소드는 서비스 어플리케이션에 포함된 각 서비스에 대한 TService 객체를 생성하는 메소드로, 서비스 객체가 추가될 때마다 자동으로 생성된다. Initialize 메소드는 어플리케이션과 관련된 폼을 숨기기 위한 초기 설정을 한다. 서비스 어플리케이션은 거의 interactive 할 필요가 없기 때문에 폼을 보여줄 필요가 없다. 만약 폼이 어플리케이션에 추가될 경우 Initialize 메소드는 시작할 때 이를 숨기게 된다. Run 메소드는 서비스를 설치, 등록하고 컴포넌트 리스트에 서비스를 추가하며 서비스 어플리케이션의 스레드를 생성하는 역할을 한다.

## ● TService 클래스

### 1. 주요 프로퍼티

AllowPause, AllowStop 프로퍼티는 서비스의 클라이언트가 서비스를 일시 중단하거나 중단할 수 있는지 여부를 결정한다. 이 값이 True 이면 onPause, onContinue 이벤트는 서비스가 일시 중단되거나 다시 시작될 때 발생하며, onStop 이벤트는 서비스가 중지되기 전에 발생한다.

Dependencies 프로퍼티는 TDependency 객체들의 컨테이너로 사용되는 프로퍼티이다. 이 프로퍼티에는 서비스가 시작되기 전에 시작하는 dependent 서비스나 load ordering 그룹이 지정된다. 서비스에서의 Dependency 가 의미하는 바는 이 서비스가 dependency 로 지정된 서비스가 시작되지 않는 한 실행되지 않는다는 의미이다. 그룹에서의 Dependency 가 의미하는 바는 그룹으로 지정된 서비스들 중에서 최소한 하나의 멤버가 시작되어야 실행된다는 의미이다.

DisplayName 프로퍼티는 SCM 에 나타나는 서비스의 이름을 나타낸다. 여기에 대해서는 앞에서 다룬바 있으므로 자세한 설명은 생략한다.

에러 처리를 위한 프로퍼티로는 ErrCode 와 ErrorSeverity 프로퍼티가 있다. ErrCode 프로퍼티는 서비스가 시작되거나 중단되는 동안 발생하는 에러의 코드가 어떤 것인지를 나타낸다. 이 프로퍼티가 정의되지 않은 경우에는 Win32ErrCode 프로퍼티가 대신 사용된다. ErrorSeverity 프로퍼티는 서비스가 기동될 때 실패할 경우에 그 심각한 정도를 나타내는 프로퍼티로 다음과 같은 값들을 가질 수 있다.

값	의 미
esIgnore	에러를 나타내지만 무시한다.
esNormal	에러가 발생하고, 메시지가 표시되지만 실행은 중지되지 않는다.
esSevere	에러가 발생하지만, 마지막(last-known-good) 환경 설정을 사용하면 실행될

	수 있는 경우
esCritical	에러가 발생하고, 과거 설정으로 시작을 시도하되 이 역시 실패하면 실행이 중단된다.

ServiceType 프로퍼티는 서비스의 종류를 나타낸다. 그 값으로는 stWin32(Win32 서비스, 디폴트), stDevice (디바이스 드라이버), stFileSystem (파일 시스템 드라이버) 일 수 있다.

ServiceStartName 프로퍼티는 서비스를 시작할 때 사용되는 이름을 나타낸다. 만약 ServiceType 프로퍼티의 값이 stWin32 이면 ServiceStartName 프로퍼티의 의미는 ‘DomainName\WUserName’ 형태의 계정 이름이다. 만약 계정이 built-in 도메인에 속해 있는 경우에는 ‘.WUserName’ 형태로 지정할 수도 있다. 서비스 어플리케이션이 하나 이상의 서비스를 포함할 때에는 이 프로퍼티와 Password 프로퍼티를 비워둔다.

ServiceType 프로퍼티의 값이 stWin32 가 아니면 ServiceStartName 프로퍼티 값은 디바이스 드라이버를 로드하기 위해 사용되는 입출력 시스템의 드라이버 객체의 이름인 ‘WFileSystemsWRdr 또는 WDriverWXns’ 형태를 가진다.

Password 프로퍼티는 ServiceType 프로퍼티의 값이 stWin32 로 지정되었을 때 ServiceStartName 프로퍼티에 지정된 계정의 패스워드를 나타낸다. 이 프로퍼티의 값이 NULL 이거나 빈 문자열을 가리킬 경우 서비스에는 패스워드가 없는 것이다.

StartType 프로퍼티는 서비스가 어떻게 시작할 것인지 지정한다. 다음과 같은 값을 가질 수 있다.

값	시작되는 방법	비 고
stBoot	운영체제 로더에 의해 시작된다.	ServiceType 이 stWin32 가 아닌 경우
stSystem	부트 시스템이 초기화된 후에 시작된다.	ServiceType 이 stWin32 가 아닌 경우
stAuto	시스템이 시작될 때 자동으로 시작된다.	
stManual	프로세스가 StartService API 함수를 호출할 때 시작된다.	
stDisabled	서비스를 관리자가 수동으로 시작해야 한다.	

Interactive 프로퍼티는 서비스가 윈도우 데스크 탑과 상호작용할 수 있는지 여부를 나타낸다. Interactive 프로퍼티는 ServiceType 프로퍼티가 stWin32 인 경우에만 효과가 있다.

LoadGroup 프로퍼티는 서비스를 포함한 load ordering 그룹의 이름을 나타낸다. 이 프로퍼티는 load ordering 그룹에서 dependency 를 가진 다른 서비스들에 의해 사용된다. 레지스트리는 load ordering 그룹을 HKLM\System\CurrentControlSet\Control\WServiceGroupOrder 키에서 확인할 수 있다.

서비스들과 서비스 그룹이 같은 name space 를 공유하기 때문에, 그룹 이름의 접두어는 반드시 SC\_GROUP\_IDENTIFIER 을 사용해서 서비스 이름과 구별된다. 이 접두어는

TDependency 클래스의 IsGroup 프로퍼티가 True 이면 자동으로 붙는다.

Param 프로퍼티는 서비스에 대한 파라미터 들을 담은 인덱스된 프로퍼티이다. 이들 파라미터는 SCM 의 시작 파라미터 입력 윈도우에 그대로 사용된다. ParamCount 프로퍼티에는 이렇게 설정된 파라미터의 수가 지정된다.

ServiceThread 프로퍼티는 서비스에 대한 스레드로 사용되는 TServiceThread 클래스 인스턴스를 나타낸다.

Status 프로퍼티는 서비스의 현재 상태를 나타내는데, 이 상태 정보는 SCM 에 표시된다. 다음과 같은 값들을 가질 수 있다.

값	의 미
csStopped	서비스가 중단되어 있다.
csStartPending	서비스가 시작하려 하고 있다.
csStopPending	서비스가 중단되고 있다.
csRunning	서비스가 실행되고 있다.
csContinuePending	서비스가 continue 되려 하고 있다.
csPausePending	서비스 pause 되려 하고 있다.
csPaused	서비스가 일시 중단되었다.

Terminated 프로퍼티는 현재 실행되고 있는 서비스의 스레드가 중단되었는지 여부를 가리킨다.

## 2. 주요 메소드

GetServiceController 메소드는 서비스에 대한 핸들러의 포인터를 반환한다. 이 함수는 직접 호출할 필요는 없고, 서비스 객체의 메인 함수가 이를 자동으로 호출한다.

서비스 어플리케이션에서 각각의 서비스는 서비스에 대한 메인 함수의 포인터를 가지고 있다. 요구에 의해 서비스가 시작되면 서비스의 메인 스레드는 RegisterServiceCtrlHandler 함수를 호출하여 이 컨트롤 핸들러 함수를 등록하게 되며, 이 함수에 대한 포인터를 반환한다. 이 함수는 클래스 메소드가 아니기 때문에 서비스 어플리케이션 위저드는 논-멤버 핸들러를 생성하게 되는 것이다.

LogMessage 메소드는 에러 메시지를 이벤트 로그에 전송한다. 이때 디폴트로 EventType 파라미터의 값은 EVENTLOG\_ERROR\_TYPE, ID 와 Category 파라미터는 0 으로 지정되어 있다. 예를 들어, 서비스가 시스템이 부트될 때 로드되지 못하면 에러 이벤트를 로그에 기록한다. EventType 파라미터의 값으로는 다음과 같은 값들을 지정할 수 있다.

값	의 미
EVENTLOG_ERROR_TYPE	Error event
EVENTLOG_WARNING_TYPE	Warning event
EVENTLOG_INFORMATION_TYPE	Information event
EVENTLOG_AUDIT_SUCCESS	Success Audit event
EVENTLOG_AUDIT_FAILURE	Failure Audit event

Category 파라미터는 이벤트 카테고리를 지정하는데, 특정 소스에 대한 정보를 가진다. ID 파라미터는 이벤트 ID 를 지정하는데, 이것은 이벤트 소스와 연관된 메시지 파일에서 엔트리로 사용되는 이벤트이다.

ReportStatus 메소드는 현재의 상태 정보를 전송하는 역할을 한다.

### 3. 주요 이벤트

BeforeInstall 이벤트는 서비스가 처음 등록되기 전에 발생하며, AfterInstall 이벤트 핸들러는 서비스가 등록된 이후에 발생한다. 마찬가지로 BeforeUninstall 이벤트는 서비스가 제거되기 직전에, AfterUninstall 이벤트 핸들러는 서비스가 제거된 직후에 발생한다.

OnStart 이벤트 핸들러는 서비스를 초기화할 때 사용되는데, 예를 들어 서비스 요구가 분리된 쓰레드로 처리된다면 쓰레드에 대한 여러가지 처리 사항은 이 이벤트 핸들러에서 처리하고 Started 파라미터를 True 로 설정하면 서비스를 시작하게 된다. OnStop 이벤트 핸들러는 서비스가 중단될 때 발생하며 Stopped 파라미터를 True 로 설정하면 서비스가 중단된다. OnShutdown 이벤트 핸들러는 서비스가 shutdown 되기 직전에 발생하며, 서비스 어플리케이션은 이 이벤트가 발생한 직후에 종료된다.

OnPause 이벤트 핸들러에서는 서비스가 일시 중단될 때의 처리를 담당하며 Paused 파라미터를 True 로 설정하면 서비스가 일시 중지된다. OnContinue 이벤트 핸들러는 SCM 이 일시 중지된 서비스를 다시 시작할 때 발생하는데 Continued 파라미터를 True 로 설정하면 서비스가 재시작된다.

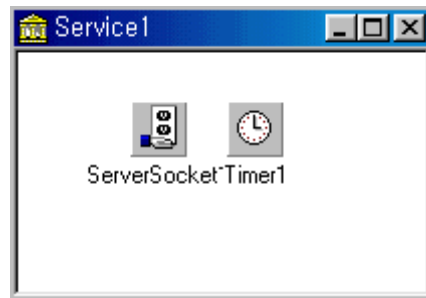
OnExecute 이벤트 핸들러는 개별 쓰레드가 OnStart 이벤트 핸들러에서 요구될 때 서비스를 직접 구현하는 핸들러로 사용된다. OnExecute 이벤트 핸들러가 종료되면, 서비스 쓰레드도 종료된다. 대부분의 OnExecute 이벤트 핸들러는 서비스 쓰레드의 ProcessRequest 메소드를 호출하는 루프를 이용하여 다른 서비스에 대한 요구가 중단되지 않도록 배려하도록 한다.

### 소켓을 이용한 서비스 어플리케이션 예제

그러면, 서비스 어플리케이션 위저드를 이용한 서비스 어플리케이션 예제를 하나 작성해보자. 작성할 서비스 어플리케이션은 TServerSocket 컴포넌트를 사용하여 서버의 현재 시간을 접속된 모든 클라이언트 어플리케이션에게 전송하는 일종의 타임 서버이다.

File|New 메뉴를 선택하고 New 탭에서 Service Application 아이템을 더블 클릭하면 새로운 프로젝트가 생성되면서, 데이터 모듈과 비슷한 윈도우가 하나 열릴 것이다.

여기에 다음과 같이 TTimer 와 TServerSocket 컴포넌트를 떨어뜨리도록 한다.



그리고, 프로젝트와 유닛 파일을 적당한 이름으로 저장하도록 하자. 이렇게 하면 이미 서비스 어플리케이션 위저드에 의해 다음과 같은 코드가 자동으로 생성되었을 것이다.

```
program ExamSvr1;

uses
  SvcMgr,
  U_ExamSvr1 in 'U_ExamSvr1.pas' {Service1: TService};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TService1, Service1);
  Application.Run;
end.

unit U_ExamSvr1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
```

ScktComp, ExtCtrls;

type

TSERVICE1 = class(TSERVICE)

ServerSocket1: TServerSocket;

Timer1: TTimer;

private

public

function GetServiceController: PServiceController; override;

{ Public declarations }

end;

var

Service1: TService1;

implementation

{ \$R \*.DFM }

procedure ServiceController(CtrlCode: DWord); stdcall;

begin

Service1.Controller(CtrlCode);

end;

function TService1.GetServiceController: PServiceController;

begin

Result := @ServiceController;

end;

end.

앞에서 설명했듯이 TService 클래스의 GetServiceController 메소드는 이런 방법으로 컨트롤러 메소드의 포인터를 반환하도록 구현되어 있다.

우리가 작성할 서비스 어플리케이션은 서비스가 시작될 때 서버 소켓을 활성화시키고, 1 초에 한번씩 서버 소켓에 연결된 클라이언트에게 현재 시간을 전송하는 것이다.

서비스 어플리케이션을 작성할 때 핵심이 되는 것은 Service1 객체의 OnExecute 이벤트

핸들러를 작성하는 것이다. 이 이벤트 핸들러가 서비스가 시작되면 실제로 실행되는 부분이다. 여기에서 서버 소켓을 다음과 같이 활성화하고, 다른 서비스가 계속 실행될 수 있도록 ServiceThread 의 ProcessRequest 메소드를 호출하는 것이 중요하다. 이 타임 서비스는 포트 2001 번을 사용하도록 지정하였다.

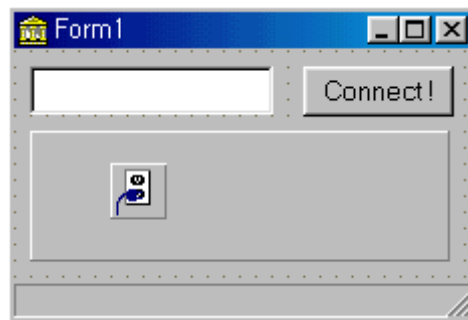
```
procedure TService1.Service1Execute(Sender: TService);
begin
    ServerSocket1.Port := 2001;
    ServerSocket1.Active := True;
    while not Terminated do ServiceThread.ProcessRequests(False);
    ServerSocket1.Active := False;
end;
```

이제는 Timer1 의 OnTimer 이벤트 핸들러를 다음과 같이 작성하여, 시스템의 현재 시각을 연결되어 있는 모든 클라이언트 소켓으로 전송하도록 하면 된다.

```
procedure TService1.Timer1Timer(Sender: TObject);
var
    TimeStr: string;
    i: Integer;
begin
    if ServerSocket1.Active then
    begin
        TimeStr := TimeToStr(Now);
        i := 0;
        while True do
        begin
            try
                ServerSocket1.Socket.Connections[i].SendText(TimeStr);
                Inc(i)
            except
                Break;
            end;
        end;
    end;
end;
```

이것으로 서비스 어플리케이션을 작성하였다. 일반적인 다른 서버 어플리케이션을 작성하는 것과 크게 다른 것이 없다는 것을 느낄 수 있을 것이다.

그러면, 이 서비스 어플리케이션을 서버로 사용하는 클라이언트 어플리케이션을 작성하도록 하자. 새로운 어플리케이션을 시작하고 에디트 박스, 버튼, Status bar, 패널과 클라이언트 소켓 컴포넌트를 다음과 같이 폼에 추가하도록 하자.



소켓 컴포넌트의 Port 프로퍼티는 서비스 어플리케이션의 포트 번호와 같이 2001 로 설정하도록 하자. 그리고, 패널 컴포넌트에는 시간을 표시하도록 할 것이므로 적당한 폰트로 설정하기 바란다.

그리고, Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성하여 서버와 접속하도록 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if ClientSocket1.Active then ClientSocket1.Active := False;
    if Length(Edit1.Text) > 0 then
    begin
        ClientSocket1.Address := Edit1.Text;
        ClientSocket1.Active := True;
    end;
end;
```

서버와 접속하면 서버의 호스트 이름을 표시하도록 클라이언트 소켓의 OnConnect 이벤트 핸들러를 다음과 같이 작성하고, 서버 소켓에서 전송하는 시간 정보를 패널에 표시하도록 OnRead 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.ClientSocket1Connect(Sender: TObject;
```



```

Socket: TCustomWinSocket);

begin
    StatusBar1.SimpleText := 'Connected to ' + Socket.RemoteHost;
end;

procedure TForm1.ClientSocket1Read(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Panel1.Caption := Socket.ReceiveText;
end;

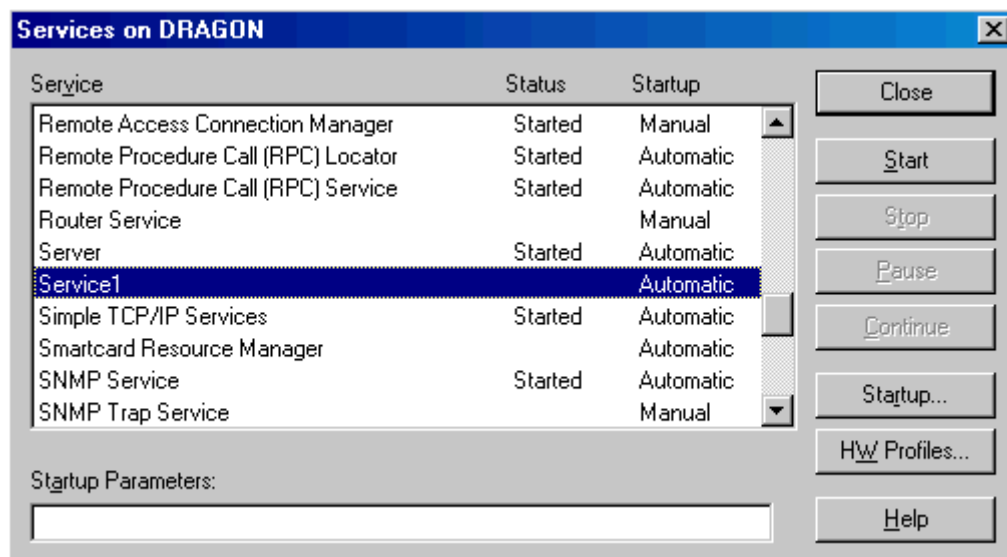
```

이것으로 클라이언트 어플리케이션도 제작하였다. 그러면 이들을 실행하여 실제 동작하는 화면을 보도록 하자.

먼저, 서비스 어플리케이션 .exe 파일을 윈도우 NT 서버 컴퓨터의 적당한 디렉토리에 위치시키고 다음과 같이 실행하여 서비스를 설치하도록 한다.

```
ExamSvr1.exe /install
```

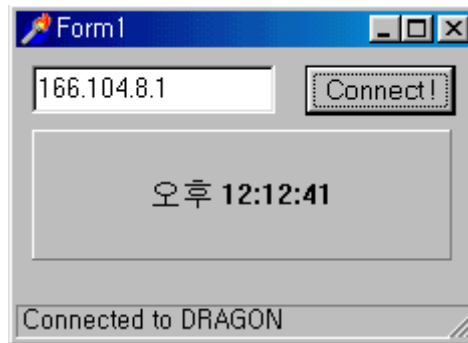
참고로, 이렇게 설치한 서비스 어플리케이션을 제거할 때에는 /uninstall 스위치를 사용하면 된다. 그리고, 서비스 컨트롤 관리자를 띄워서 서비스가 설치되었는지 확인하자. 다음 그림과 같이 서비스가 설치된 것을 확인할 수 있을 것이다.



서비스를 실행하려면 Start 버튼을 클릭하면 된다. 참고로, 윈도우 NT 를 다시 시작하면

자동으로 서비스가 시작되지만, 설치한 직후에는 이와 같이 수동으로 서비스를 시작해야 한다.

이제 클라이언트 컴퓨터에서 클라이언트 어플리케이션을 시작하고, 서버의 IP 주소를 지정한 뒤에 ‘Connect !’ 버튼을 클릭하면 다음과 같이 서버의 시간이 패널에 표시되는 것을 관찰할 수 있을 것이다.



## 정 리 (Summary)

이번 장에서는 텔파이 4 에서 제공되는 서비스 어플리케이션 위저드를 이용하여 윈도우 NT 서비스 어플리케이션을 작성하는 방법에 대해서 알아보았다. 앞서서도 언급했듯이 윈도우 NT 서비스는 나름대로의 잇점이 많다. 그렇다고, 일반적인 클라이언트/서버 방식으로 작성할 수 있는 프로그램을 억지로 서비스로 만드는 것은 낭비이므로 삼가도록 하자.

클라이언트/서버 방식의 어플리케이션을 개발할 때 윈도우 NT 의 서비스로 서버 어플리케이션을 개발하면 여러가지 문제점을 해결할 수 있는 경우가 많다. 그러므로, NT 서비스 자체의 잇점과 단점을 파악하고 실제로 이것이 현재 프로젝트에 있어서 유용한 것인지를 판단하는 것이 중요할 것이다.

# 트레이 아이콘 어플리케이션의 제작

## (Creating Tray Icon Application)

윈도우 95 와 NT 4.0 에는 작업바의 우측에 작업 트레이라는 부분이 있다. 여기에 있는 아이콘은 현재 데스크 탑에서 실행되고 있는 윈도우를 가지고 있는 어플리케이션이라기 보다는 도스 시절에 있었던 램상주 프로그램과 비슷한 역할을 하는 것이 많다.

이번 장에서는 트레이에 상주하는 어플리케이션을 제작하는 방법에 대해서 알아보자.

### Shell\_NotifyIcon

트레이 아이콘을 사용하는 어플리케이션에서는 Shell\_NotifyIcon 이라는 API 함수를 사용하게 된다. 이 함수에는 파라미터가 둘 있는데, TNotifyIconData 구조체에 대한 포인터와 트레이에 작업을 원하는 플래그를 설정한다. 플래그에는 아이콘을 추가, 삭제, 수정 등의 기능을 설정할 수 있다. TNotifyIconData 구조체는 다음과 같이 정의되어 있다.

type

TNotifyIconData = record

cbSize: DWORD;

Wnd: HWND;

uID: UINT;

uFlags: UINT;

uCallbackMessage: UINT;

hIcon: HICON;

szTip: array[0..63] of AnsiChar;

end;

여기서 cbSize 는 구조체의 크기이며, Wnd 는 트레이 아이콘이 메시지를 보낼 윈도우에 대한 핸들이다. uID 는 아이콘을 확인하기 위한 것으로 어플리케이션에 트레이 아이콘이 여러 개 있을 때 사용된다. uFlag 필드에는 세 가지를 사용할 수 있다. NIF\_MESSAGE, NIF\_ICON, NIF\_TIP 이 그것으로 어느 것이 유효한 것인지 결정하는 플래그이다.

uCallbackMessage 필드는 사용자의 아이콘에 대한 동작에 대해 윈도우(hWnd)로 보내어지는 사용자 정의 메시지의 갯수를 나타내며, hIcon 은 트레이 영역 안에 표시할 아이콘의 핸들이다. 마지막으로 szTip 필드에는 작업 표시줄의 아이콘 위로 마우스 커서가 놓일 때 보여주는 툴팁 문자열에 대한 텍스트이다.

이 구조체는 ShellAPI.pas 유닛에 선언되어 있으므로, 실제 사용을 하기 위해서 interface 섹션의 uses 문에 ShellAPI 를 추가해서 사용한다.

## 기본 원칙

트레이 아이콘 어플리케이션이 동작하는 방식에 대해 생각해보자.

제일 먼저 폼이 생성될 때 작업 트레이 부분에 TNotifyIconData 구조체의 내용을 적절히 채워서 아이콘을 등록해야 할 것이다. 그리고, 아이콘이나 툴팁 등의 변화가 있을 경우 이를 적절하게 반영해 주어야 한다.

그리고, 콜백 메시지를 등록해서 일단 트레이 아이콘 어플리케이션에 해당 되는 메시지일 경우에는 이 메시지를 처리해 주는 루틴을 만들어 주어야 한다. 이때 트레이 아이콘을 더블 클릭할 경우에는 디자인한 폼을 보여주도록 해야 할 것이며, 폼이 보여진 후 최소화 버튼을 눌렀을 때에는 다시 트레이 아이콘으로 복귀하도록 해야 할 것이다. 그리고, 트레이 아이콘에서 오른쪽 버튼을 클릭했을 때에는 디자인한 팝업 메뉴를 보여주고 메뉴를 실행할 수 있어야 한다. 또한, 작업이 끝나면 트레이 아이콘을 제거해야 한다.

마지막으로 염두에 두어야 할 것은 어플리케이션이 처음 시작할 때 어플리케이션 폼이 바로 트레이 아이콘으로 최소화해서 위치하게 해야 한다.

이를 간략하게 정리하면 다음과 같다.

1. 트레이 아이콘 등록
2. 트레이 아이콘 변화를 반영
3. 메시지 처리 루틴 제작
4. 팝업 메뉴와 폼의 최소화(minimize), 복귀(restore) 메시지 처리
5. 트레이 아이콘 제거
6. 어플리케이션 시작시 트레이 아이콘으로 시작할 것

먼저 트레이 아이콘을 등록하는 부분에 대해서 알아보자.

다음의 프로시저는 전형적인 트레이 아이콘을 등록하는 코드를 담고 있다.

```
procedure AddTrayIcon;  
var  
    IconData: TNotifyIconData;  
begin  
    with IconData do  
    begin  
        cbSize := SizeOf(IconData);
```

```

Wnd := Handle;
uID := 0;
uFlags := NIF_ICON or NIF_MESSAGE or NIF_TIP;
uCallbackMessage := WM_MyCallBack;
hIcon := LoadIcon(hInstance, 'MyIcon');
szTip := '시계';
end;
Shell_NotifyIcon(NIM_ADD, @IconData);
end;

```

내용은 앞에서 설명한 TNotifyIcon 형의 구조체 변수에 적절한 데이터 필드를 입력하고, 마지막으로 Shell\_NotifyIcon API 함수를 호출하는 것이다.

여기서 주의 깊게 보아야 할 부분이 세군데 있다.

첫째는 uFlags 필드로 여기에서 추가 또는 수정을 할 때 이용할 플래그를 결정한다. NIF\_ICON 은 트레이 아이콘에 대한 정보를, NIF\_TIP 은 트레이 아이콘에 대한 툴팁 힌트에 대한 정보를 나타낸다. 마지막으로 NIF\_MESSAGE 는 사용되는 사용자 정의 메시지에 대한 것이다. 보통 처음 트레이 아이콘을 추가할 때에는 이 세가지 정보를 모두 등록해야 하므로 NIF\_ICON, NIF\_TIP, NIF\_MESSAGE 를 사용하게 된다.

둘째는 uCallbackMessage 필드로 여기에는 메시지 처리를 하게될 사용자 정의 메시지를 대입한다. 보통 처음에 상수 선언을 해서 대입을 하게 된다.

셋째로는 Shell\_NotifyIcon API 함수를 호출할 때 사용하는 파라미터로, 지금과 같이 아이콘을 추가할 때에는 NIM\_ADD, 아이콘에 대한 정보를 수정할 때에는 NIM\_MODIFY, 아이콘을 제거할 때에는 NIM\_DELETE 를 각각 호출한다.

이렇게 등록된 트레이 아이콘에 대한 정보를 수정할 때에는 다음과 같이 하면 된다.

```
Shell_NotifyIcon(NIM_MODIFY, @IconData);
```

즉, 바뀌게 되는 내용을 TNotifyIconData 형 변수의 필드에 대입하고 NIM\_MODIFY 로 Shell\_NotifyIcon API 함수를 호출한다.

트레이 아이콘을 제거할 때에도 다음과 같이 해주면 해결 된다.

```
Shell_NotifyIcon(NIM_DELETE, @IconData);
```

## 메시지 처리

메시지를 처리하기 위해서 먼저 사용할 사용자 정의 메시지와 이를 처리할 콜백 함수를 선

언해야 한다.

```
const
```

```
    WM_CallbackMessage = WM_User + 100;
```

```
procedure WndProc(var Message: TMessage);
```

이 함수는 어플리케이션에 대한 메시지를 전달해서 처리하게 된다. 이 함수를 다음과 같이 사용해서 메시지를 처리한다. 다음의 코드는 필자가 제작한 트레이 아이콘 제작 컴포넌트 (TTrayIcon) 소스의 일부이다.

```
procedure TTrayIcon.WndProc(var Message: TMessage);
```

```
begin
```

```
    try
```

```
        with Message do
```

```
            case Msg of
```

```
                WM_QueryEndSession: Message.Result := 1;
```

```
                WM_EndSession:
```

```
                    if TWmEndSession(Message).EndSession then EndSession;
```

```
                WM_CallbackMessage:
```

```
                    case Message.IParam of
```

```
                        WM_LBUTTONDOWNBLCLK: DbClick;
```

```
                        WM_LBUTTONUP: Click(mbLeft);
```

```
                        WM_RBUTTONUP: Click(mbRight);
```

```
                    end;
```

```
                else Result := DefWindowProc(FHandle, Msg, wParam, lParam);
```

```
            end;
```

```
        except
```

```
            Application.HandleException(Self);
```

```
        end;
```

```
    end;
```

즉, 반드시 처리해 주어야 하는 메시지는 자신이 TNotifyIconData 형의 변수 필드에 등록한 메시지로 여기서는 WM\_CallbackMessage 가 된다. 보통 그 중에서도 마우스 더블 클릭과 좌우 버튼 클릭에 대한 메시지만 처리하면 된다.

그 밖의 메시지 중 WM\_EndSession 은 트레이 아이콘 어플리케이션을 종료하고자 할 때

발생하는 메시지이다. 여기서는 필자가 따로 제작한 EndSession 이라는 프로시저를 호출한다. EndSession 프로시저에서는 다음과 같이 등록된 트레이 아이콘을 제거하는 코드를 사용한다.

```
procedure TTrayIcon.EndSession;
begin
    Shell_NotifyIcon(NIM_DELETE, @FIconData);
end;
```

## 트레이 아이콘 컴포넌트(TTrayIcon)의 제작

트레이 아이콘 어플리케이션에 대한 확실한 이해를 돕기 위해 델파이 폼에 올려 놓고 모든 어플리케이션을 바로 트레이 아이콘 어플리케이션으로 전환시킬 수 있는 컴포넌트를 하나 제작해 보았다.

이 컴포넌트를 사용하면 쉽게 트레이 아이콘 어플리케이션을 만들 수 있다. 지면 관계상 모든 소스를 다 설명할 수는 없지만 중요한 부분을 골라서 설명하겠다.

기본적인 테크닉은 앞에서 설명한 Shell\_NotifyIcon 함수를 이용해서 TNotifyIconData 구조체를 등록, 수정, 삭제하고 메시지를 처리하는 것이다.

먼저 컴포넌트를 설계하도록 하자. 이 컴포넌트에서는 트레이 아이콘의 형태를 지정할 수 있는 Icon, 툴팁을 지정할 수 있는 Hint, 그리고 트레이 아이콘을 오른쪽 버튼 클릭할 때 보여줄 팝업 메뉴를 설정할 때 사용할 PopupMenu 프로퍼티를 제공하도록 하자. 또한 이벤트로는 트레이 아이콘을 클릭할 때 발생하는 OnClick, 더블 클릭할 때 발생하는 OnDbClick, 그리고 폼을 트레이 아이콘에 최소화하거나 정상 형태로 복귀할 때 각각 발생하는 OnHide, OnShow 이벤트를 제공하도록 한다.

다음은 TTrayIcon 컴포넌트의 선언 부분이다.

```
TTrayIcon = class(TComponent)
private
    FHandle: HWND;
    FIconData: TNotifyIconData;
    FIcon: TIcon;
    FHint: string;
    FPopupMenu: TPopupMenu;
    FOnClick: TMouseEvent;
    FOnDbClick: TNotifyEvent;
    FOnHide: TNotifyEvent;
```

```

FOnShow: TNotifyEvent;

procedure SetHint(const Hint: string); virtual;

procedure SetIcon(Icon: TIcon); virtual;

procedure WndProc(var Message: TMessage);

protected

    procedure DoMenu; virtual;

    procedure Click(Button: TMouseButton); virtual;

    procedure DbClick; virtual;

    procedure EndSession; virtual;

    procedure Changed; virtual;

public

    constructor Create(Owner: TComponent); override;

    destructor Destroy; override;

    procedure Hide(Sender: TObject); virtual;

    procedure Show(Sender: TObject); virtual;

published

    property Hint: string read FHint write SetHint;

    property Icon: TIcon read FIcon write SetIcon;

    property PopupMenu: TPopupMenu read FPopupMenu write FPopupMenu;

    property OnClick: TMouseEvent read FOnClick write FOnClick;

    property OnDbClick: TNotifyEvent read FOnDbClick write FOnDbClick;

    property OnHide: TNotifyEvent read FOnHide write FOnHide;

    property OnShow: TNotifyEvent read FOnShow write FOnShow;

end;

```

한번쯤 컴포넌트를 직접 만들어보신 분이라면 그다지 어렵지 않게 이해할 것이다.

그렇지만, 앞에서 설명한 3 개의 프로퍼티와 4 개의 이벤트와 직접 연관되지는 않지만 내부적으로 사용되는 데이터 필드와 메소드를 소개하면 다음과 같다.

FHandle 은 WndProc 콜백 함수가 사용될 어플리케이션 윈도우에 대한 핸들을 저장한다. 또한, FIconData 는 트레이 아이콘을 등록, 수정, 삭제할 때 사용할 TNotifyIconData 구조체 변수이다.

protected 섹션에 정의된 메소드 들중 Click 과 DbClick 은 OnClick, OnDbClick 이벤트를 실제 구현할 메소드 이다. EndSession 은 앞에서 잠깐 설명한 바 있으므로 생략한다.

Changed 메소드는 Set 메소드에서 프로퍼티에 저장된 TNotifyIconData 구조체 필드의 내용을 실제로 반영하는 메소드로 이 메소드와 Set 메소드들은 다음과 같이 구현된다.



```

procedure TTrayIcon.Changed;
begin
    if not (csDesigning in ComponentState) then
        Shell_NotifyIcon(NIM_MODIFY, @FIconData);
end;

procedure TTrayIcon.SetHint(const Hint: string);
begin
    if FHint <> Hint then
    begin
        FHint := Hint;
        StrPLCopy(FIconData.szTip, Hint, SizeOf(FIconData.szTip) - 1);
        if Hint <> '' then
            FIconData.uFlags := FIconData.uFlags or NIF_TIP
        else
            FIconData.uFlags := FIconData.uFlags and not NIF_TIP;
        Changed;
    end;
end;

procedure TTrayIcon.SetIcon(Icon: TIcon);
begin
    if FIcon <> Icon then
    begin
        FIcon.Assign(Icon);
        FIconData.hIcon := Icon.Handle;
        Changed;
    end;
end;

```

즉, Changed 메소드는 어플리케이션이 델파이의 디자인 모드에 있지 않으면 FIconData 변수의 필드 값을 Shell\_Notify 함수를 이용해서 반영하는 것이다.

그리고, 각각의 Set 메소드 들은 먼저 FIconData 변수에 해당 필드와 프로퍼티로 나타내기 위해서 필드 데이터(FIcon, FHint)에 값을 저장하고 Changed 메소드를 호출한다.

DoMenu 메소드는 팝업 메뉴가 선택되었을 때 메뉴를 보여주는 메소드로 Click 메소드에 의해서 호출되며 다음과 같이 구현된다.

```

procedure TTrayIcon.DoMenu:
var
    Pt: TPoint;
begin
    if (FPopupMenu <> nil) and not IsWindowVisible(Application.Handle) then
    begin
        GetCursorPos(Pt);
        FPopupMenu.Popup(Pt.X, Pt.Y);
    end;
end;

```

```

procedure TTrayIcon.Click(Button: TMouseButton);
var
    MousePos: TPoint;
begin
    GetCursorPos(MousePos);
    if (Button = mbRight) then DoMenu;
    if Assigned(FOnClick) then FOnClick(Self, Button, [], MousePos.X, MousePos.Y);
end;

```

그다지 어렵지 않은 코드이므로 자세한 설명은 생략하도록 한다.

컴포넌트를 구현한 다른 부분들은 이달에 디스켓으로 제공되는 소스를 직접 참고하기 바라며, 마지막으로 Create 생성자를 구현하는 코드에 대해서 살펴보자.

```

constructor TTrayIcon.Create(Owner: TComponent);
begin
    inherited Create(Owner);
    FIcon := TIcon.Create;
    FIcon.Assign(Application.Icon);
    FHandle := AllocateHwnd(WndProc);
    if not (csDesigning in ComponentState) then
    begin
        FillChar(FIconData, SizeOf(FIconData), 0);
        with FIconData do
            begin

```

```

    cbSize := SizeOf(FlconData);
    Wnd := FHandle;
    hlcon := Icon.Handle;
    uFlags := NIF_ICON or NIF_MESSAGE;
    uCallbackMessage := WM_CallbackMessage;
end;
StrPLCopy(FlconData.szTip, Application.Title, SizeOf(FlconData.szTip) - 1);
if Application.Title <> '' then
    FlconData.uFlags := FlconData.uFlags or NIF_TIP;
if not Shell_NotifyIcon(NIM_ADD, @FlconData) then
    raise EOutOfResources.Create('트레이 아이콘을 생성할 수 없습니다 !');
Application.OnMinimize := Hide;
Application.OnRestore := Show;
end;
end;
end;

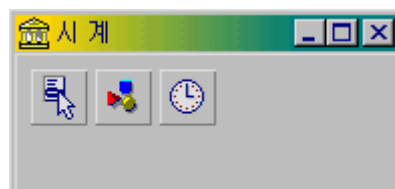
```

즉, 초기 아이콘으로 어플리케이션 객체의 아이콘을 그대로 사용하게 하였다. 또한, 등록하는 윈도우를 `AllocateHwnd` 함수를 이용해서 이 컴포넌트의 `WndProc` 함수가 소속된 윈도우를 찾는다. 그리고, 에러 처리 부분을 추가 했으며 어플리케이션 객체가 최소화 하거나 복귀할 때에 `TTrayIcon` 컴포넌트의 `Hide`, `Show` 메소드에서 처리하도록 한다.

## TTrayIcon 컴포넌트를 이용한 트레이 아이콘 어플리케이션의 제작

그럼 이제 `TTrayIcon` 컴포넌트를 이용해서 실제로 트레이 아이콘 어플리케이션을 제작해 보자. 먼저 이달의 디스켓으로 제공되는 `TrayIcon.zip` 파일의 압축을 풀고 `TrayIcon.pas` 유닛을 이용해서 컴포넌트를 설치한다. 그러면 ‘Samples’ 팔레트에 `TTrayIcon` 컴포넌트가 추가될 것이다.

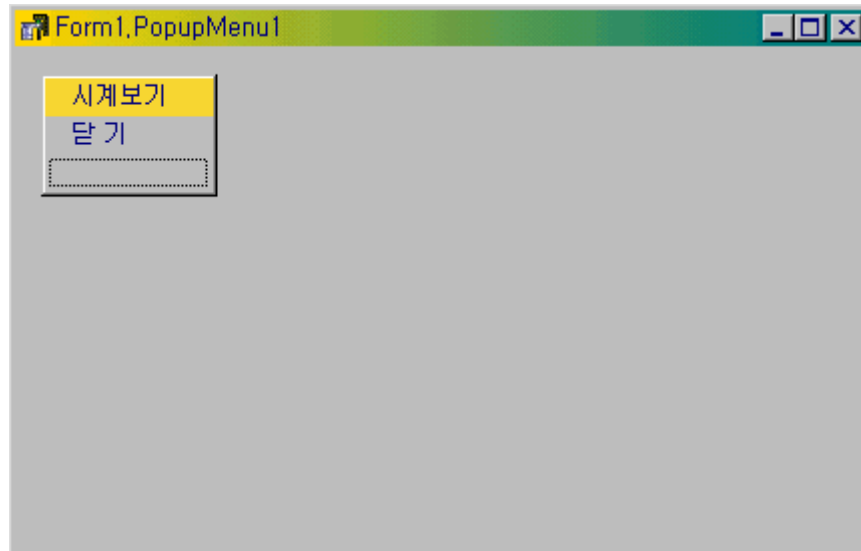
새로운 어플리케이션을 시작하고, 폼을 다음과 같이 디자인 한다.



즉, `TPopupMenu`, `TPanel`, `TTimer`, `TTrayIcon` 컴포넌트를 하나씩 올려 놓는다. 이때 `Panel1` 의 `Align` 프로퍼티를 `alClient` 로 설정해서 폼에 딱 차도록 한다. 또한, `Caption` 프

로퍼티는 시간을 알려주게 할 것이므로 그냥 비워둔다.

그리고 팝업 메뉴에는 ‘시계보기’와 ‘닫 기’를 다음 그림과 같이 설정한다.



이제 TrayIcon1 컴포넌트의 프로퍼티를 설정해 보자. 디자인 시에는 PopupMenu 프로퍼티만 콤보 박스를 이용해서 PopupMenu1 으로 설정하면 된다.

그리고 팝업 메뉴들에 대한 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.N1Click(Sender: TObject);
begin
    Application.Restore;
end;
```

```
procedure TForm1.N2Click(Sender: TObject);
begin
    Form1.Close;
end;
```

즉, ‘시계 보기’ 메뉴일 경우에는 Application 객체의 Restore 메소드를 사용해서 폼을 보여 주게 하고, ‘닫 기’일 경우에는 폼의 Close 메소드를 호출해서 어플리케이션을 종료한다.

이때 Application.Restore 메소드는 TTrayIcon 컴포넌트의 Show 메소드를 호출하도록 변경되어 있으므로 TrayIcon1.Show 와 같은 기능을 하게 된다.

그리고, Timer 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

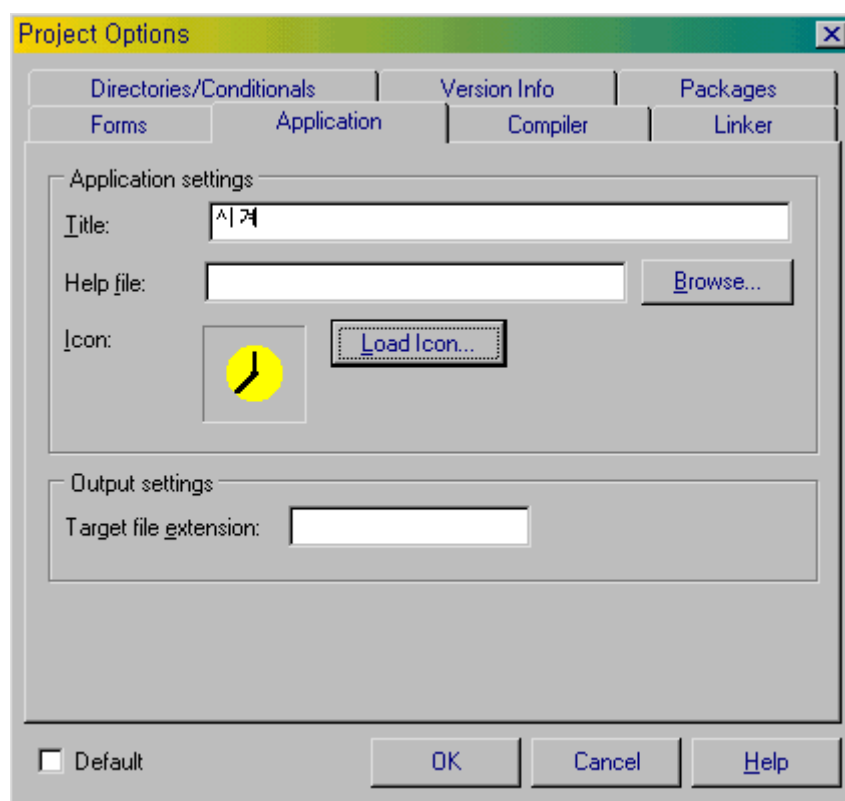
```

begin
    Panel1.Caption := TimeToStr(Time);
    TrayIcon1.Hint := TimeToStr(Time);
end;

```

현재 시각을 문자열로 변환해서 Panel1 과 트레이 아이콘의 툴팁으로 설정한다.

이제 사용할 어플리케이션 아이콘과 초기 툴팁을 결정하기 위해 Project|Options ... 메뉴를 클릭한 후 Application 탭을 선택하고, 다음 그림과 같이 설정한다. 이때 사용한 아이콘 파일은 필자가 이미지 에디터로 급히 그린 것이라 그다지 아름답지 못하므로 독자 여러분의 너그러운 양해를 바란다. 원래 계획으로는 필자가 트레이 아이콘을 이용해서 움직이는 공룡 캐릭터와 타이머를 이용해서 자그마한 다마고치 예제를 제공하려고 했으나, 필자의 조악한 그림 실력으로 인해 이렇게 시계 예제로 전환하였다. 관심 있는 분들은 TTrayIcon 컴포넌트를 이용해서 쉽게 구현할 수 있을 것이다.



이제 마지막으로 처음에도 언급 했듯이 어플리케이션이 시작할 때 트레이 아이콘 상태로 시작하도록 하자. 필자도 이 방법을 찾기 위해 별별 방법을 다 써 보았는데, 해답은 생각보다 가까운 곳에 있었다. 즉, 폼의 OnActivate 이벤트에서 Application.Minimize 를 호출해

주면 된다.

```
procedure TForm1.FormActivate(Sender: TObject);  
begin  
    Application.Minimize;  
end;
```

이제 시계 예제가 완성되었다. 프로그램을 실행하면 다음 그림과 같이 동작하는 화면들을 얻을 수 있을 것이다.



## 정 리 (Summary)

트레이 아이콘 어플리케이션을 만드는 방법은 아마도 윈도우 95 와 윈도우 NT 4.0 셸을 수정해서 사용하는 방법 중에서 가장 유용하고 간단하다고 말할 수 있다. 특히 도스 시절의 램상주 프로그램처럼 간단한 유틸리티를 제작할 때에는 그 활용도가 매우 높다고 할 수 있다. 이미 많은 수의 유틸리티 프로그램들이 제공되고 있는데, 이들 중에는 시스템을 모니터하거나, 경고를 주는 등의 유용한 유틸리티 들이 있으며 대부분의 경우 트레이 아이콘 어플리케이션의 형태로 제공되고 있다.

# Win32 셸 확장 기법의 활용

## (Using Win32 Shell Extensions)

윈도우 95 로 들어오면서, 각종 파일에 대한 접근과 운영체제에서 셸의 역할을 하는 부분이 MS-DOS 시절과 많은 변화를 보여주고 있다.

이번 장에서는 COM 기술에 기반한 Shell 인터페이스의 사용 방법을 익히고, 이를 사용해서 유용한 어플리케이션을 제작해 보도록 한다.

Shell 확장 기법을 사용하기 위해서 델파이에서는 ShlObj.pas 유닛을 제공한다. 여러가지 셸 확장 기법이 존재하지만 대표적인 것으로는 namespace 확장과 컨텍스트 메뉴 핸들러, 아이콘 핸들러와 파일 뷰어에 대한 확장 등을 들 수 있다.

### 폴더와 뷰 (Folder and View)

윈도우 탐색기는 2 개의 pane 으로 나뉘어진 창을 가지고 있다. 각각의 뷰는 객체와 그 내용에 대한 사용자 인터페이스를 표현한다.

좌측 창은 사용자에게 객체의 위치에 대해 Win32 트리뷰 컨트롤을 사용해서 Shell 에 있는 모든 다른 객체에 대한 상대적인 위치를 나타내 보여줌으로써 표현하고 있다. 사용자는 좌측 창을 보면서 현재 선택된 객체가 어떤 객체의 자손인지, 어떤 객체를 포함하고 있는지 한눈에 파악할 수 있다.

과거의 윈도우 3.1 의 파일 관리자를 생각해 보자. 이 때에는 좌측 창이 디렉토리 트리를 나타내는 것이었다. 보기에는 파일 관리자의 좌측 창과 크게 다르지 않아 보이지만, 윈도우 탐색기에서는 디렉토리 이외에 제어판 정보나 프린터 폴더 등의 수 많은 다른 객체 들에 대한 정보를 포함하고 있다.

우측 창에는 좌측 창에서 선택한 아이템에 대한 세부 내용을 표시하게 되어 있다. 현재 디렉토리를 탐색하고 있다면 디렉토리의 내용을 볼 수 있을 것이고, 제어판 등의 객체를 탐색하고 있다면 제어판에 속한 객체를 볼 수 있을 것이다.

이렇게 윈도우 탐색기를 이용해서 탐색을 할 수 있는 폴더 객체에는 몇 가지 종류가 있다. 가장 기본적이고 전통적인 하드 디스크의 디렉토리를 나타내는 폴더 객체가 있을 것이고, 제어판이나 네트워크 정보 등의 가상 폴더가 존재할 것이다. 전통적인 하드 디스크의 디렉토리에 대한 폴더는 셸(shell) 자체에 의해서 구현되며, 다른 종류의 가상 폴더 들은 셸 확장(Shell extension) COM 객체를 통해 구현된다.

이런 가상 폴더 들에 대한 올바른 명칭은 'namespace extensions' 이며, 개발자 들은 이들을 커스텀 셸 확장 COM 객체를 제작함으로써 생성할 수 있다. 이렇게 만든 셸 확장 객체는 반드시 DLL 로 포장해야 하며, 최소한의 IUnknown, IShellExtInit 인터페이스는 구현해

주어야 한다.

이런 namespace extension 객체는 두가지의 메인 컴포넌트로 구성되는데, 이들이 각각 폴더(folder)와 뷰(view) 객체이다. 이들 COM 객체 들은 각각 IShellFolder 와 IShellView 인터페이스를 반드시 구현해야 한다. 그러므로, 결국 커스텀 셸 확장 COM 객체를 만들려면 폴더 객체는 IUnknown, IShellExtInit, IShellFolder 를 구현해야 하며, 뷰 객체는 IUnknown, IShellExtInit, IShellView 인터페이스를 구현해야 한다.

## 아이템 ID (Item Identifiers)

먼저 폴더 객체가 어떻게 윈도우의 셸과 상호작용 하는지 알아 보자. 탐색기의 좌측 창은 셸의 namespace 내에 있는 객체 들의 위치 정보를 대표하는 것이다. 그러므로, 각 객체 들은 자신의 위치를 정확하게 표현해야 하는데, 이를 위해 반드시 필요한 것이 아이템 ID(identifier) 이다.

이때 폴더가 나타내는 것은 디렉토리뿐 아니라 제어판, 네트워크 정보, 프린터 폴더 등일 수도 있다. 그렇기 때문에 폴더의 위치를 나타내는데 디렉토리의 경로를 사용할 수 없다. 마이크로소프트에서는 개발자가 개별적인 컴포넌트에 대해서 독특한 형식을 정의할 수 있도록 했는데, 이러한 구조체를 아이템 ID 라고 한다. 아이템 ID 의 구조는 다음과 같이 정의할 수 있다.

```
PSHItemID = ^TSHItemID;
TSHItemID = packed record
    cb: Word; //ID 의 Size
    abID: array[0..0] of Byte; //아이템 ID (길이는 변화 가능)
end;
```

이 선언부를 살펴 보면 abID 에 대한 구체적인 정의가 없는 것을 알 수 있다. 그렇기 때문에, 자신의 객체에 대해 유일한 경로에 있는 다른 branch 를 적절하게 나타내기 위해서 어떠한 형태의 이진 데이터도 사용할 수 있는 것이다.

이러한 아이템 ID 는 독립적으로 사용하기 보다는, 아이템 ID 리스트(Item Identifier List)라고 불리는 데이터 구조로 사용한다. 보통 하나의 아이템 ID 리스트는 아이템 ID 세트 하나를 포함한다. 이때 목록의 마지막 아이템의 cb 파라미터는 0 을 값으로 가지게 된다. 이런 리스트를 이용해서 특정 객체를 윈도우 셸의 namespace 에서 나타낼 수 있게 된다.

폴더 객체는 셸의 namespace 내에서 어디에 존재하는지를 자신이 직접 추적할 필요가 없다. 객체가 구성되기 전에 셸은 어디에서 이 객체를 찾을 것인지 미리 알아야 하고, 폴더 객체를 생성하고 나면, 셸은 그 객체의 IPersistFolder 인터페이스를 호출해서 위치를 알아내고 이 값을 아이템 ID 리스트에 전달한다.



IShellFolder 인터페이스를 구현한 셸 extension 이 아이템 ID 를 만들어낼 때, 아이템을 확인하기 위한 기본적인 데이터 뿐만 아니라 다른 기능을 구현할 때 도움이 되는 추가적인 정보를 기록한다. 예를 들어, 셸의 파일 시스템 아이템들의 IShellFolder 구현 부분은 기본적으로 아이템 들을 확인하기 위해서 파일과 디렉토리에 대한 긴 이름을 저장하며, 동시에 짧은 이름과 크기, 날짜 등의 데이터도 기록한다.

아이템 ID 리스트가 SHGetPathFromIDList 등의 셸 API 함수의 파라미터로 넘겨질 때에는 언제나 name space 의 root(데스크탑 폴더)에서부터의 경로를 사용해야 한다. 그에 비해서 IShellFolder 멤버 함수에 파라미터로 사용될 때에는 폴더로부터의 상대적 경로를 사용한다.

## 주요 인터페이스와 API

그렇다면 Win32 셸에 의해 지원되는 주요 인터페이스와 API 에 대해서 간단히 알아보도록 하자.

### ● 작업 할당자 (Task allocator) API

모든 셸 extension 은 메모리 객체(보통은 아이템 ID 리스트)를 할당하거나 해제할 때 반드시 작업 할당자를 사용해야 한다. 작업 할당자에 접근하는 방법에는 셸 extension 이 Ole32.dll 과 링크되어 있는지 여부에 따라 두가지 방법이 존재한다.

1. 셸 extension 이 OLE API 를 호출하는 경우(OLE32.DLL 과 링크되는 경우)에는 CoGetMalloc API 를 이용해서 Ole 의 작업 할당자를 호출해야 한다.
2. OLE API 를 호출하지 않는 경우에는 다음에 선언되어 있는 셸 작업 할당자 API 를 호출해야 한다.

```
function SHGetMalloc(var ppMalloc: IMalloc): HRESULT; stdcall;
```

### ● IContextMenu 인터페이스

IContextMenu 인터페이스는 다음과 같은 세가지 경우에 사용된다.

1. 셸이 컨텍스트 메뉴 extension 을 로드할 때

사용자가 셸의 name space(파일, 디렉토리, 프린터, 워크 그룹 등)에서 오른쪽 마우스 버튼을 클릭하면 그 객체 형에 대한 디폴트 컨텍스트 메뉴가 생성되고, 레지스트리에 등록되어 있는 컨텍스트 메뉴 extension 이 로드된다. 여기에 추가적인 메뉴 아이템이 있으면 추가

된다. 이러한 컨텍스트 메뉴 extension 은 HKEY\_CLASSES\_ROOT\beginProgIDend\Wshellex\ContextMenuHandlers 에 등록되어 있다.

## 2. 셸이 확장 name space 에 있는 서브 폴더의 컨텍스트 메뉴를 가져올 때

탐색기의 name space 가 name space extension 에 의해 확장된 경우, 셸은 IShellFolder.GetUIObjectOf 메소드를 호출하여 IContextMenu 객체를 불러온다.

## 3. 셸이 디렉토리에 대한 논-디폴트 drag-and-drop 핸들러를 로드할 때

사용자가 논-디폴트 drag-and-drop 을 파일 시스템 폴더(디렉토리 등)에 수행할 때, 셸은 레지스트리 HKEY\_CLASSES\_ROOT\beginProgIDend\Wshellex\DragDropHandlers 키에 등록된 셸 extension 을 로드한다.

### ● IFileViewer 인터페이스

FileViewer 컴포넌트 객체에 의해 구현되는 인터페이스이다. 파일 이름은 IPersistFile 인터페이스를 통해 뷰어에 전달된다.

### ● IShellBrowser/IShellView/IShellFolder 인터페이스

이들 인터페이스는 셸이 name space extension 과 통신을 할 때 사용된다. 셸은 IShellBrowser 인터페이스를 제공하며, extension 들은 IShellFolder 와 IShellView 인터페이스를 구현한다.

## 1. 커맨드/메뉴 아이템 ID

탐색기는 WM\_COMMAND 메시지가 커맨드/메뉴 아이템 ID 들 범위 안에 있으면 이를 디스패치한다. 모든 메뉴 아이템의 ID 는 FCIDM\_SHVIEWFIRST/LAST 범위에 있어야 하며, 그렇지 않으면 이들을 디스패치하지 못한다.

FCIDM\_SHVIEWFIRST/LAST: IShellView 의 범위 (우측 pane)

FCIDM\_BROWSERFIRST/LAST: 탐색기 프레임의 범위 (IShellBrowser)

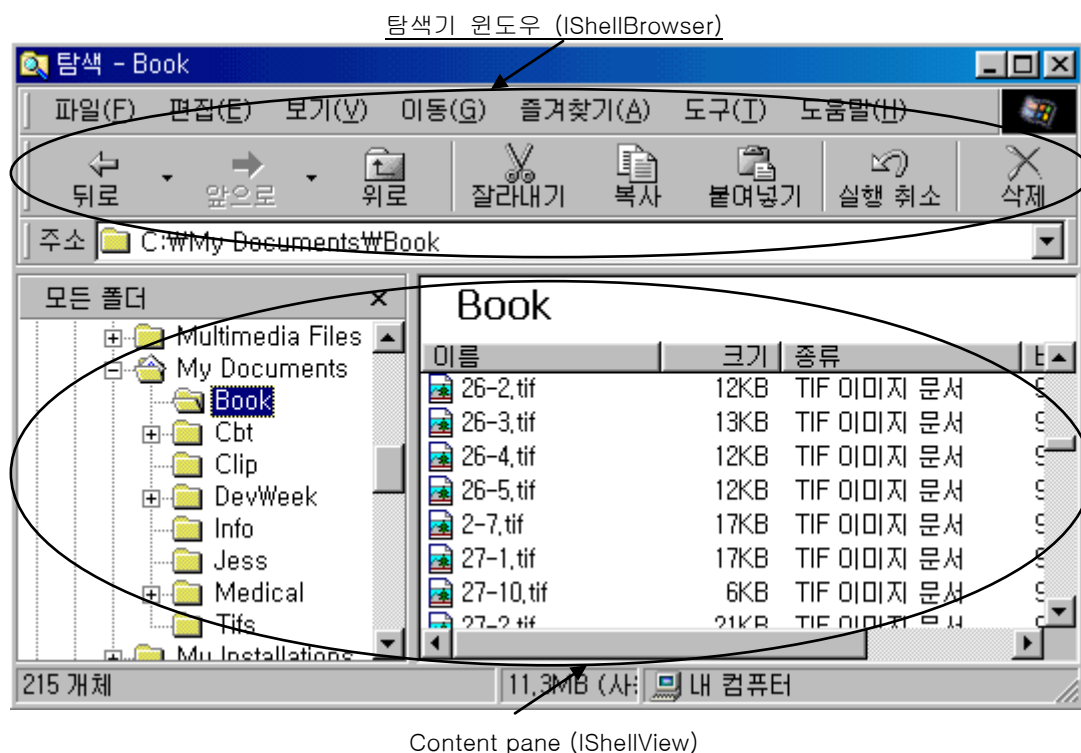
FCIDM\_GLOBAL/LAST: 탐색기 서브 메뉴 IDs

## 2. FOLDERSETTINGS

FOLDERSETTINGS 는 사용자가 브라우즈할 때에 탐색기가 하나의 폴더 뷰에서 다른 폴더 뷰로 넘어갈 때 전달되는 데이터 구조체이다. 현재의 설정을 얻기 위해서 ISV.GetCurrentInfo 메소드를 호출하며, 이를 ISV.CreateViewWindow 메소드에 넘겨서 다음 폴더 뷰가 이를 상속받을 수 있게 한다.

### 3. IShellBrowser 인터페이스

IShellBrowser 인터페이스는 셸 탐색기/폴더 프레임 윈도우에 의해 제공된다. 이 인터페이스가 셸 폴더에 대한 contents pane 을 생성할 때 (이것은 IShellFolder 인터페이스를 제공한다.), IShellView 객체를 생성하기 위해 IShellFolder 의 CreateViewObject 메소드를 호출하게 된다. 그리고 나서, IShellView 의 CreateViewWindow 메소드를 호출하여 contents pane 윈도우를 생성한다. IShellBrowser 인터페이스에 대한 포인터는 CreateViewWindow 를 호출할 때 IShellView 객체에 포인터로 넘겨진다.



## Namespace

Namespace에는 두 가지 종류가 있다. 하나는 셸이 관리하는 표준 namespace 이고, 다른 하나는 개발자가 직접 생성한 커스텀 namespace 이다.

이때 커스텀 namespace 의 가장 상위 레벨의 객체 만이 표준 namespace 에 자동으로 나타난다. 이렇게 하려면 다음의 두 가지 방법 중 하나를 사용해야 한다.

- 표준 namespace 에서 디렉토리를 하나 생성하고 폴더 객체의 CLSID(클래스 ID)를 파일 이름 확장자의 형태로 부여한다. 예를 들면 Custom Namespace.{00000000-0000-0000-0000-000000000000}와 같은 형태가 된다.
- 레지스트리의 다음과 같은 키의 엔트리를 생성한다.  
HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Explorer\Desktop\Namespace.

각각의 namespace 객체는 자신의 프로퍼티와 자식 객체 들이 프로퍼티 값을 어떻게 얻을 것인지에 대해서 알고 있어야 한다. 윈도우 탐색기는 사용자가 해당 namespace 확장자 폴더를 호출하면 (폴더 아이콘을 더블 클릭하는 등) 커스텀 namespace 에 있는 자식 객체를 보여주라고 요구하게 된다.

일단 커스텀 namespace 의 루트를 윈도우 탐색기에 추가하면 탐색기에 보이게 된다. 이 폴더를 탐색하려고 하면 탐색기는 셸의 일반적인 인터페이스에 입각해서 사용자에게 내용을 보여주게 된다. 사용자가 폴더를 더블 클릭하거나 좌측의 ‘+’ 기호를 클릭하면 탐색기는 선택된 폴더의 모든 자식 폴더의 목록을 보여주게 되어 있다. 이를 위해서 탐색기는 폴더 객체의 IShellFolder.EnumObjects 메소드를 호출한다.

또한, 사용자가 특정 폴더를 클릭하면 탐색기는 사용자에게 폴더 객체의 내용을 보여주게 되어 있다. 이를 위해서는 내용을 보여주기 전에 다음의 두 가지가 선행되어야 한다.

- 폴더 객체를 생성한다. 탐색기는 반드시 선택된 폴더 객체의 부모를 먼저 생성하고, 그 객체의 IShellFolder.BindToObject 메소드를 호출한다.
- 뷰 객체를 생성한다. 일단 선택된 폴더 객체가 생성되면 탐색기는 그 객체의 IShellFolder.CreateViewObject 메소드를 호출한다.

## 폴더와 뷰의 상호작용

사용자가 폴더를 클릭하면 탐색기는 폴더에 대한 뷰 객체를 생성해야 한다. 이 작업이 IShellFolder.CreateViewObject 를 호출하여 이루어 진다는 것은 앞에서 간단히 설명했다. 일단 뷰 객체가 생성되면 어떤 종류의 뷰 객체가 생성될 것인지 결정해야 한다.

뷰의 종류에는 크게 두 가지가 있다.

하나는 폴더 내부에 아이템을 포함하는 팝업 뷰 윈도우(popup view window)이다. 이 뷰는 사용자가 팝업 윈도우에서 폴더의 아이템을 더블 클릭 할 때 볼 수 있는 뷰이다. 또한, 사

용자가 탐색기에서 오른쪽 버튼을 클릭해서 해당 폴더에 대해서 ‘열기(O)’를 선택했을 때 볼 수 있다.

다른 하나는 탐색기에 의해서 만들어지는 것으로, 사용자가 탐색기의 좌측 창에서 폴더를 더블 클릭하면 폴더의 내용이 우측 창에 나타나게 되어 있다. 이런 동작은 좌측 창에서 오른쪽 버튼을 클릭해서 ‘탐색(E)’를 선택했을 때와 같은 것이다.

이 두 가지 경우에 있어서, 폴더 객체는 IShellView.CreateViewWindow 메소드를 호출해서 뷰 객체를 생성한다. 즉, 어떤 뷰 모드를 선택할 것인지 여부는 전적으로 뷰 객체가 CreateViewWindow 를 어떻게 구현할 것인지에 달려 있다.

마지막으로 폴더와 뷰의 관계에 있어서 알아야 할 것은 하나의 폴더 객체에 대해서 많은 수의 뷰 객체가 있을 수 있다는 것이다. 그러므로, 각각의 뷰 객체와 폴더 객체는 반드시 분리된 COM 객체로 구현되어야 한다. 이들에 대한 동기화 작업은 윈도우의 셸이 알아서 하게 된다.

셸 확장에 관한 내용 중에서 가장 복잡하고도 어려운 것이 namespace 와 파일 뷰어를 확장해서 구현하는 것이다. 여기에 대해서 구현하는 것은 이 책의 범위를 넘게 되므로 예제를 작성하지는 않는다. 그렇지만, 궁금한 독자들은 인터넷 상에 꽤 많은 컴포넌트와 예제가 소스와 함께 제공되므로 이들을 찾아보기 바란다.

#### 참고: out 키워드

out 키워드는 델파이 3 에서 추가된 것으로 컴파일러에게 함수를 호출하기 전에 몇 가지 추가적인 코드를 생성하게 한다. 다음의 코드를 살펴보자.

```
var  
    malloc: IMalloc;  
begin  
    GetAnotherMallocOut( malloc );  
    GetAnotherMallocVar( malloc );  
end;
```

앞에서 호출한 두가지 함수의 선언부는 다음과 같다.

```
procedure GetAnotherMallocOut( out malloc: IMalloc );  
procedure GetAnotherMallocVar( var malloc: IMalloc );
```

즉, 하나는 out 키워드를 하나는 var 키워드를 사용한 것이다.

앞의 함수 호출을 디버거를 이용해서 컴파일러가 생성해낸 코드를 살펴보면 다음과 같다.

```
GetAnotherMallocOut( malloc );
```

```
    lea     eax,[ebp-04]
```

```
    call    @IntfClear
```

```
    mov     edx,eax
```

```
    mov     eax,ebx
```

```
    call    TForm1.GetAnotherMallocOut
```

```
GetAnotherMallocVar( malloc );
```

```
    lea     edx,[ebp-04]
```

```
    mov     eax,ebx
```

```
    call    TForm1.GetAnotherMallocVar
```

이를 살펴 보면 out 파라미터를 사용한 경우에는 컴파일러가 IntfClear 라는 함수를 호출하는 코드가 추가 되었다는 것을 알 수 있다. 이 함수는 System.pas 유닛에 선언되어 있는 것으로 현재 넘겨지는 변수에 COM 인터페이스가 대입되어 있는지 알아보고, 그렇다면 본 함수를 호출하기 전에 Release 를 호출하게 된다. 이렇게 하면 COM 인터페이스를 사용할 때 흔히 생기기 쉬운 참조 계수(reference count)의 혼란 문제를 해결할 수 있다. 비록 사소해 보이지만, 문제가 생겼을 때에는 대단히 어려울 수도 있는 부분을 해결해 준다.

## 셸 링크 (Shell Link)

셸 확장 인터페이스 중에서 이번에는 IShellLink 인터페이스를 이용해서 어플리케이션의 셸 링크를 생성하는 방법에 대해서 알아보도록 하자.

다른 셸 확장 기법을 사용할 때와는 달리 IShellLink 인터페이스는 윈도우 셸에 의해 구현되므로 따로 구현할 필요는 없고, 단지 다음과 같이 CoCreateInstance 함수를 이용하여 인스턴스를 생성하면 된다.

```
var
```

```
    SL: IShellLink;
```

```
begin
```

```
    OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,  
        IShellLink, SL));
```

```
    ...
```

```
end;
```

참고로 기억해야 할 것은, OLE 함수를 사용하기 전에 CoInitialize 함수를 호출하여 COM 라

이브러리를 초기화할 필요가 있으며, COM 을 모두 사용한 뒤에는 CoUninitialize 함수를 호출하여야 한다.

셸 링크는 데스크 탑에서 오른쪽 버튼을 클릭해서 단축 아이콘을 생성할 수 있는데, 이렇게 생성된 셸 링크는 실제로는 .lnk 확장자를 가지는 파일이다. 윈도우가 시작되면 .lnk 파일에 대한 디렉토리를 검사한다. 셸은 링크와 폴더의 관계를 시스템 레지스트리에 기록하게 된다 (보통 HKEY\_CURRENT\_USER\Software\Microsoft\Windows\Current Version\WExplorer\Shell Folders 키에 위치한다).

쉽게 말해서 셸 링크를 특정 폴더에 생성하는 것은 링크 파일을 특정 디렉토리에 위치시키는 것과 같은 의미이다. 이때 레지스트리에 직접 접근하지 않고, SHGetSpecialFolderPath 함수를 이용하여 특정 폴더에 대한 디렉토리 패스를 얻는 것이 중요하다. 이 함수의 선언부는 다음과 같다.

```
function SHGetSpecialFolderPath(hwndOwner: HWND; lpszPath: PChar;
    nFolder: Integer; fCreate: BOOL): BOOL; stdcall;
```

여기서 hwndOwner 파라미터는 윈도우의 핸들을 나타내며, lpszPath 파라미터에는 패스를 기록할 버퍼의 포인터를 나타낸다. nFolder 파라미터는 패스를 얻고자 하는 특정 폴더를 나타내며, 마지막으로 fCreate 파라미터는 폴더가 존재하지 않을 때 폴더를 생성할 것인지 여부를 결정한다.

여기서 nFolder 파라미터에 지정할 수 있는 값으로는 여러가지가 있는데, 이것에 대한 자세한 내용은 도움말이나 MSDN 등의 내용을 참고하기 바란다.

IShellLink 인터페이스를 구현하기 위해서는 IPersistFile 인터페이스를 지원해야 하는데, 이는 파일 접근이 필요하기 때문이다. IPersistFile 인터페이스는 디스크에 접근할 수 있는 메소드를 제공한다.

IShellLink 를 구현하는 클래스는 IPersistFile 인터페이스를 구현해야 하므로 as 연산자를 사용해서 다음과 같이 인스턴스를 얻을 수 있다.

```
var
    SL: IShellLink;
    PF: IPersistFile;
begin
    OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
        IShellLink, SL));
    PF := SL as IPersistFile;
    ...
end;
```

다음의 코드는 메모장에 대한 데스크탑 셸 링크를 생성한다.

```
procedure MakeNotepad:
const
  AppName = 'c:\Windows\notepad.exe';
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  PF := SL as IPersistFile;
  OleCheck(SL.SetPath(PChar(AppName)));      //링크 패스를 설정
  LnkName := GetFolderLocation('Desktop') + 'W' + ChangeExt(ExtractFileName(AppName), '.lnk');
  PF.Save(PWideChar(LnkName), True);        //링크 파일의 저장
end;
```

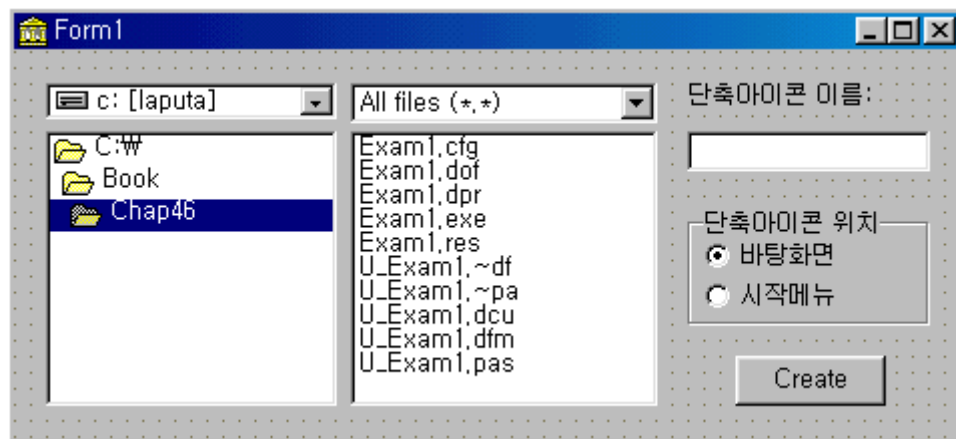
이 프로시저에서 IShellLink 인터페이스의 SetPath 메소드는 실행 파일이나 문서의 링크를 나타낼 때 사용된다. 그리고, 링크의 패스와 파일 이름은 GetFolderLocation 함수를 이용하여 얻어오게 되는데, ChangeFileExt 함수를 이용하여 .exe 확장자를 .lnk 확장자로 변경하여 사용한다. 마지막으로 Save 메소드는 디스크 파일에 대한 링크를 저장한다.

IShellLink 인터페이스의 정의를 보면 GetXXX, SetXXX 메소드가 여러 개 있는데, 이 메소드들은 셸 링크의 여러 필드의 값을 얻거나 설정하는데 사용된다.

그러면, 파일을 선택해서 그 파일을 바탕 화면이나 시작 메뉴에 등록하는 간단한 예제 어플리케이션을 만들어 보자. 시작 메뉴에는 'Sample'이라는 디렉토리를 만들어 여기에 단축 아이콘을 등록할 것이다.

먼저 폼에 TDriveComboBox, TDirectoryListBox, TFileListBox, TFilterComboBox 와 라벨과 에디트박스, 버튼 및 TRadioGroup 컴포넌트를 하나씩 폼에 얹고, 다음과 같이 디자인한다.





여기서 DriveComboBox1 의 DirList 프로퍼티는 DirectoryListBox1, DirectoryListBox1 과 FilterComboBox1 의 FileList 프로퍼티는 FileListBox1 으로 설정한다. 그리고, 필터로는 모든 파일, 실행 파일, DLL, 텍스트 파일의 4 종류로 설정한다.

이제 Button1 의 OnClick 이벤트 핸들러를 작성해서 Edit1 의 내용을 바탕화면이나 시작메뉴에 등록하도록 하면 된다. 이때 uses 절에 ShlObj.pas, ActiveX.pas, ComObj.pas, Registry.pas 유닛을 추가해야 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Unknown: IUnknown;
  ShellLink: IShellLink;
  PersistFile: IPersistFile;
  FileName, Directory: string;
  WideFileName: WideString;
  MyRegistry: TRegIniFile;
begin
  Unknown := CreateComObject(CLSID_ShellLink);
  ShellLink := Unknown as IShellLink;
  PersistFile := Unknown as IPersistFile;
  FileName := FileListBox1.FileName;
  with ShellLink do
  begin
    SetPath(PChar(FileName));
    SetWorkingDirectory(PChar(ExtractFilePath(FileName)));
  end;
  MyRegistry
```

```

:= TRegIniFile.Create('Software\Microsoft\Windows\CurrentVersion\Explorer');
case RadioGroup1.ItemIndex of
  0: Directory := MyRegistry.ReadString('Shell Folders', 'Desktop', '');
  1:
begin
  Directory := MyRegistry.ReadString('Shell Folders', 'Start Menu', '') + '\Sample';
  CreateDir(Directory);
end;
end;
WideFileName := Directory + '\W' + Edit1.Text + '.lnk';
PersistFile.Save(PWChar(WideFileName), False);
MyRegistry.Free;
end;

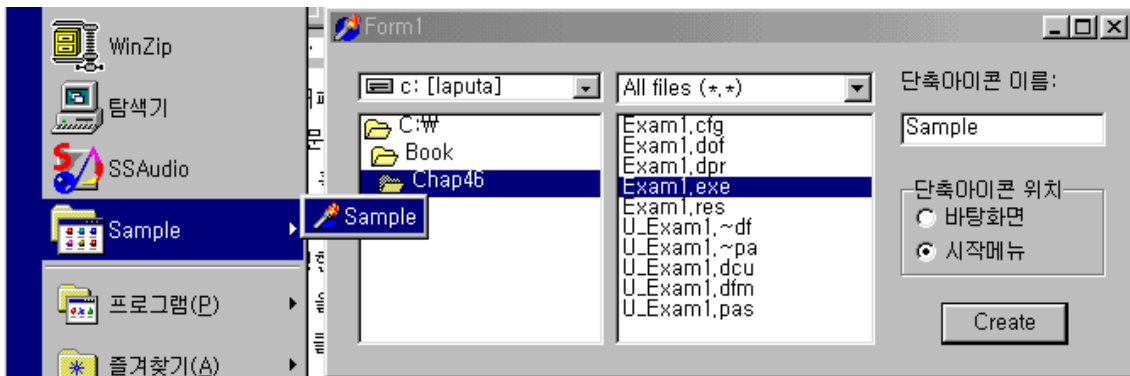
```

IShellLink 인터페이스와 IPersistFile 인터페이스는 이미 윈도우에 의해 구현되어 있으므로 특별히 따로 구현하지 않고, 이렇게 CreateComObject 함수를 이용하여 구현된 객체의 인스턴스를 이용하면 된다. CLSID\_ShellLink 는 IShellLink 인터페이스를 구현한 CoClass 의 CLSID 이다. IShellLink 인터페이스를 구현한 CoClass 는 IPersistFile 인터페이스를 구현하고 있기 때문에 간단히 as 연산자를 이용하여 사용할 수 있다.

단축 아이콘을 지정하기 위해서 반드시 사용해야 하는 메소드로는 SetPath 메소드를 이용하여 단축 아이콘이 가리키게 될 파일의 패스를 지정하면 된다. SetWorkingDirectory 메소드를 이용해서는 단축 아이콘이 실행된 후의 디렉토리를 지정하므로 지정된 파일의 디렉토리로 설정하는 것이 좋다.

그리고, 레지스트리의 Software\Microsoft\Windows\CurrentVersion\Explorer 키의 내용이 중요한데, .lnk 파일을 저장하기 전에 실제로 바탕화면에 이를 저장할 것인지 아니면 시작메뉴에 저장할 것인지 여부는 'Shell Folders' 섹션에서 바탕화면의 경우는 'Desktop', 시작메뉴의 경우는 'Start Menu'의 값을 읽어와서 이를 이용하여 디렉토리를 결정하면 된다. IPersistFile 인터페이스의 Save 메소드는 PWideString 형의 문자열을 이용해야 하기 때문에, WideString 문자열로 선언된 WideFileName 변수에 디렉토리 이름과 에디트 박스의 내용을 이름으로 확장자 '.lnk'를 붙여서 저장한 후 이를 PWChar() 연산을 이용해 PWideString 문자열로 형변환하여 Save 메소드를 호출하면 된다.

컴파일하고, 실행한 뒤에 라디오 그룹에서 '시작메뉴'를 선택하고, 파일 리스트 박스에서 Exam1.exe 파일을 지정하고 단축 아이콘 이름으로 'Sample'을 지정하도록 하자. 그리고 'Create' 버튼을 클릭하면 다음과 같이 단축 아이콘이 만들어진 것을 볼 수 있을 것이다.



## Copy Hook 핸들러

Copy hook 셸 확장을 이용하면 폴더가 복사, 삭제, 이동, 변경될 때 이를 가로챌 수 있는 핸들러를 설치할 수 있다. 이를 이용하여 특정 작업을 할 수 없도록 할 수 있다.

Copy hook 핸들러를 만들기 위해서는 먼저 TComObject 클래스에서 상속한 클래스에 ICopyHook 인터페이스를 구현하도록 해야 한다. ICopyHook 인터페이스는 ShlObj.pas 유닛에 다음과 같이 정의되어 있다.

```
ICopyHookA = interface(IUnknown) { si }
    [SID_IShellCopyHookA]
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT; pszSrcFile: PAnsiChar;
        dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
end;
```

즉, CopyCallback 메소드만 구현하면 되는 것이다. 이 함수는 셸 폴더를 이용하여 작업을 할 때마다 호출되는 콜백 함수의 역할을 한다. 이 함수의 파라미터를 이해하는 것이 중요하다. Wnd 파라미터는 copy hook 핸들러가 사용할 부모 윈도우의 핸들을 지정하며, wFunc 파라미터는 폴더에 수행되는 작업의 종류를 지정한다. 여기에는 다음과 같은 것들이 있다.

값	의미
FO_COPY (\$2)	pszSrcFile 에 지정된 파일을 pszDestFile 에 지정된 위치에 복사한다.
FO_DELETE (\$3)	pszSrcFile 에 지정된 파일을 삭제한다.
FO_MOVE (\$1)	pszSrcFile 에 지정된 파일을 pszDestFile 에 지정된 위치로 이동한다.
FO_RENAME (\$4)	pszSrcFile 에 지정된 파일의 이름을 변경한다.
PO_DELETE (\$13)	pszSrcFile 에 지정된 프린터를 삭제한다.
PO_PORTCHANGE (\$20)	프린터 포트를 변경한다. pszSrcFile 과 pszDestFile 파라미터는 문자열의 목록을 가지는데, 각각의 문자열은 포트와 프린터 이름이 연결되어

	있다. pszSrcFile 파라미터는 현재 프린터 포트를, pszDestFile 파라미터에는 새로운 프린터 포트를 지정한다.
PO_RENAME (\$14)	pszSrcFile 에 지정된 프린터의 이름을 변경한다.
PO_REN_PORT (\$34)	PO_RENAME 과 PO_PORTCHANGE 의 결합

wFlags 파라미터는 작업에 대한 추가적인 내용을 지정한다. 이 파라미터에는 다음 값의 조합으로 지정할 수 있다.

값	의 미
FOF_ALLOWUNDO (\$40)	Undo 정보를 저장한다.
FOF_MULTIDESTFILES (\$1)	SHFileOperation 함수에서 여러 파일을 선택할 수 있다.
FOF_NOCONFIRMATION (\$10)	확인을 하지 않는다.
FOF_NOCONFIRMMKDIR (\$200)	새로운 디렉토리를 생성해야 할 때 확인을 하지 않는다.
FOF_RENAMEONCOLLISION (\$8)	이미 존재하는 파일에 작업을 할 경우 새로운 이름을 이용한다.
FOF_SILENT (\$4)	파일 작업이 진행되는 대화 상자를 보여주지 않는다.
FOF_SIMPLEPROGRESS (\$100)	작업 상황을 간단히 보여주지만, 파일 이름을 표시하지 않는다.

pszSrcFile 파라미터는 소스 폴더를 지정하며, dwSrcAttribs 파라미터는 소스 폴더의 속성을 지정한다. 마찬가지로 pszDestFile 파라미터는 목적 폴더의 이름을 나타내며, dwDestAttribs 파라미터는 목적 폴더의 속성을 나타낸다.

다른 메소드들과는 달리 이 메소드는 OLE 결과 코드를 반환하지 않고, 작업을 허용할 때에는 IDYES, 이 파일에 대한 작업만 허용하지 않을 경우에는 IDNO, 전체 작업을 취소할 때에는 IDCANCEL 을 반환한다.

## 컨텍스트 메뉴 핸들러 (Context Menu Handlers)

컨텍스트 메뉴 핸들러는 셸에서 파일 객체와 연관되어 있는 로컬 메뉴에 아이템을 추가할 수 있도록 해주는 역할을 한다. 컨텍스트 메뉴 셸 확장을 지원하기 위해서는 IShellExtInit 인터페이스와 IContextMenu 인터페이스를 지원해야 한다.

Copy hook 핸들러와 마찬가지로 컨텍스트 메뉴 핸들러를 작성하기 위해서도 TComObject 클래스에서 상속받아서 CoClass 를 작성해야 한다.

IShellExtInit 인터페이스는 셸 확장을 초기화하는데 사용된다. 이 인터페이스에는 다음과 같은 Initialize 메소드 하나로 정의되어 있다.

```
function Initialize(pidlFolder: PItemIDList; lpdoobj: IDataObject; hKeyProgID: HKEY): HRESULT; stdcall;
```

Initialize 메소드는 컨텍스트 메뉴 핸들러를 초기화하는 역할을 한다. pidlFolder 파라미터는 컨텍스트 메뉴를 디스플레이할 아이템을 포함하는 폴더에 대한 PItemIDList 구조체의 포인터이다. PItemIDList 에 대해서는 앞에서 자세히 설명한 바 있으므로 자세한 설명은 생략한다. lpdobj 파라미터는 실행할 객체를 가져오는데 사용할 IDataObject 인터페이스 객체를 지정한다. hkeyProgID 파라미터는 파일 객체나 폴더 형에 대한 레지스트리 키를 포함한다.

Initialize 메소드를 구현하기 위해서는 lpdobj 파라미터에서 IDataObject 인터페이스의 GetData 메소드를 이용하여 데이터를 가져온 뒤에 DragQueryFile 메소드를 이용하여 파일의 수와 파일 이름을 가져와야 한다. 자세한 내용은 예제를 작성하면서 설명하도록 하겠다. IContextMenu 인터페이스는 셸의 파일과 연관되어 있는 팝업 메뉴를 관리하는 기능을 하는 것으로, 다음과 같이 정의되어 있다.

```
IContextMenu = interface(IUnknown)
    [SID_IContextMenu]
    function QueryContextMenu(Menu: HMENU;
        indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HRESULT; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HRESULT; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
        pszName: LPSTR; cchMax: UINT): HRESULT; stdcall;
end;
```

일단 컨텍스트 핸들러가 IShellExtInit 인터페이스에 의해 초기화되면, 그 다음에는 IContextMenu 인터페이스의 QueryContextMenu 메소드가 호출된다. 이 메소드의 파라미터에는 메뉴의 핸들과 삽입될 메뉴 아이템이 첫번째 메뉴 아이템에서 몇 번째에 위치할 것 인지를 지정할 인덱스, 그리고 메뉴 아이템 ID의 최소값과 최대값을 지정한다. 마지막으로 메뉴의 속성을 지정할 플래그를 파라미터로 넘겨주면 된다.

그 다음에는 GetCommandString 메소드가 셸에 의해 호출되는데, 이 메소드는 특정 메뉴 아이템에 대한 언어 독립적인 커맨드 문자열(language-independent command string)이나 도움말 문자열(help string)을 반환하는 역할을 한다. 이 메소드의 파라미터로는 메뉴 아이템의 옵션과 받아들 정보의 종류를 나타내는 플래그, 예약된 파라미터와 문자열 버퍼 그리고 버퍼의 크기를 지정하게 된다.

컨텍스트 메뉴에서 사용자가 새로운 아이템을 클릭하면 셸은 InvokeCommand 메소드를 호출한다. 이 메소드는 TCMInvokeCommandInfo 구조체를 파라미터로 이용하는데, 이 구조체는 ShlObj.pas 유닛에 다음과 같이 정의되어 있다.

```

PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
_CMINVOKECOMMANDINFO = record
    cbSize: DWORD;
    fMask: DWORD;
    hwnd: HWND;
    lpVerb: LPCSTR;
    lpParameters: LPCSTR;
    lpDirectory: LPCSTR;
    nShow: Integer;
    dwHotKey: DWORD;
    hIcon: THandle;
end;
TCMInvokeCommandInfo = _CMINVOKECOMMANDINFO;

```

여기에서 lpVerb 필드의 LoWord 에 선택된 메뉴 아이템의 인덱스가 저장되어 있다.

컨텍스트 메뉴 핸들러는 반드시 시스템 레지스트리의 HKEY\_CLASSES\_ROOT\<File Type>\WshellexWContextMenuHandlers 에 등록되어야 한다. TComObjectFactory 클래스에서 상속받아 레지스트리에 등록하는 과정을 추가한 새로운 클래스 팩토리를 이용하여 이 문제를 해결할 것이다.

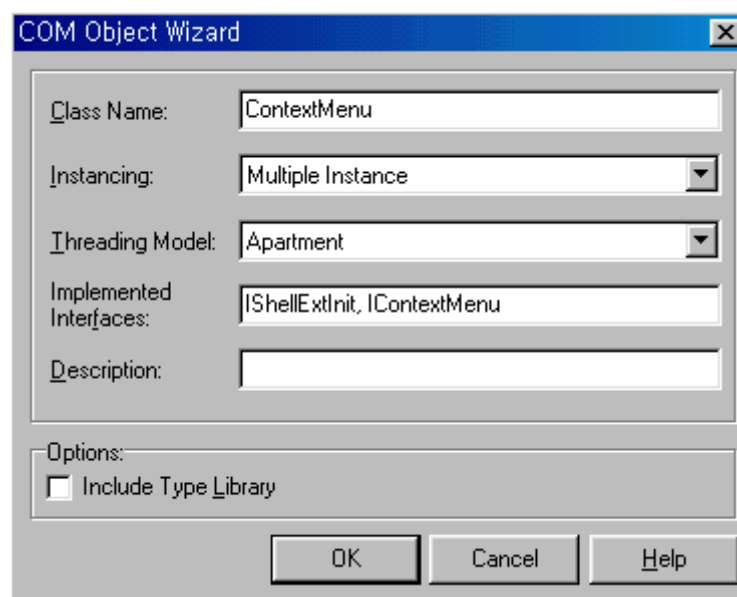
그러면, 꽤 쓸모 있는 예제를 만들어 보자.

텔피언이라면 폴더를 열고 파일을 열어서 팝업 메뉴를 사용할 때, .pas 확장자를 가진 텔파이 유닛의 소스 코드를 보고자 하면 지금까지는 .pas 파일을 더블 클릭하여 덩치 큰 텔파이 IDE 를 띄워서 보거나, 아니면 메모장과 같은 에디터를 일단 띄운 뒤에 ‘열기’ 메뉴를 이용하여 .pas 파일을 읽도록 해야 소스 코드를 볼 수 있었을 것이다. 아마도, 필자를 포함해서 많은 독자들이 .pas 유닛 파일에 커서를 두고 오른쪽 버튼을 클릭한 뒤에, 여기서 파스칼 소스 코드를 표시하고 편집할 수 있는 에디터를 바로 띄워서 사용할 수 있으면 좋겠다는 생각을 모두들 가졌을 것이라고 생각한다. 실제로 텔파이 2, 3 의 (지금쯤은 텔파이 4 버전도 나왔을 지 모르겠다.) Merlin 이라는 셸 확장 프로그램을 설치하면 텔파이 유닛 전용 에디터를 팝업 메뉴에서 직접 띄워서 이를 편집할 수 있도록 제공하였다.

이번에 제작할 컨텍스트 메뉴 핸들러는 .pas 확장자를 가진 텔파이 유닛 파일에서 오른쪽 버튼을 클릭하면 ‘내용 보기’라는 메뉴 아이템이 추가된 팝업 메뉴가 나타나며, ‘내용 보기’를 선택하면 메모장에 해당 텔파이 유닛 파일을 열어서 보여주게 되는 역할을 하는 것이다. 이와 같이 셸 확장 기법의 활용에서도 파일 뷰어(‘간략히 보기’)를 제작하는 것과 함께 컨텍스트 메뉴 핸들러를 만드는 것은 가장 그 유용성이 높은 것 중에 하나이다.

컨텍스트 메뉴 핸들러를 제작하기 위해서 먼저 File|New 메뉴를 선택하고 ActiveX 탭의 ActiveX Library 아이콘을 더블 클릭하여 액티브 X DLL 을 생성하도록 한다. 앞에서도 간

단히 설명했듯이 앞에서 구현한 IShellLink 인터페이스와는 달리 IContextMenu 와 IShellExtInit 인터페이스에 대한 내용은 윈도우가 구현하고 있는 객체를 활용하는 것이 아니기 때문에, 이들 인터페이스를 구현하는 CoClass 를 직접 구현해야 한다. 참고로 여기서 설명하는 용어들을 이해하기 어렵다면 제 4 부의 액티브 X 에 대한 내용들을 보다 정확하게 이해하고 다시 읽어보기를 권장한다. 그러므로, TComObject 에서 상속한 객체에 IContextMenu 와 IShellExtInit 인터페이스를 구현하도록 선언할 것이다. 이를 위해서 텔파이 4 에서 새롭게 제공되는 ComObject 위저드를 이용하도록 하자. File|New 메뉴를 선택하고 ActiveX 탭에서 ComObject 아이콘을 더블 클릭한다. 그리고, 나타나는 대화상자의 내용을 다음과 같이 채운다.



OK 버튼을 클릭하면 아마도 다음과 같은 뼈대 코드가 만들어질 것이다.

```
unit U_Exam2;

interface

uses

    Windows, ActiveX, ComObj;

type

    TContextMenu = class(TComObject, IShellExtInit, IContextMenu )
    protected
        // IShellExtInit methods
        // IContextMenu methods
```

```
end;
```

```
const
```

```
Class_ContextMenu: TGUID = '{B722EC20-3D21-11D2-A345-0000E8364868}';
```

```
implementation
```

```
uses ComServ
```

```
initialization
```

```
TComObjectFactory.Create(ComServer, TContextMenu, Class_ContextMenu,  
  'ContextMenu', '', ciMultInstance, tmApartment);
```

```
end.
```

여기에서 먼저 IShellExtInit, IContextMenu 인터페이스의 선언부가 있는 ShlObj.pas 유닛을 interface 섹션의 uses 절에 추가한다. 그리고 이들 인터페이스의 메소드를 다음과 같이 TContextMenu 클래스의 메소드로 구현하기 위해 선언하도록 한다.

```
uses
```

```
Windows, ActiveX, ComObj, ShlObj;
```

```
type
```

```
TContextMenu = class(TComObject, IShellExtInit, IContextMenu )
```

```
private
```

```
  FFileName: array[0..MAX_PATH] of Char;
```

```
  FMenuIndex: UINT;
```

```
protected
```

```
  // IShellExtInit method
```

```
  function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
```

```
    hKeyProgID: HKEY): HRESULT; reintroduce; stdcall;
```

```
  // IContextMenu methods
```

```
  function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst, idCmdLast,
```

```
    uFlags: UINT): HRESULT; stdcall;
```

```
  function InvokeCommand(var lpici: TCMInvokeCommandInfo): HRESULT; stdcall;
```

```
  function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
```

```
    pszName: LPSTR; cchMax: UINT): HRESULT; stdcall;
```

```
end;
```



먼저, 구현에 필요한 유닛인 SysUtils.pas, ShellAPI.pas, Registry.pas 유닛을 uses 절에 추가한다. 그리고, IShellExtInit 인터페이스의 Initialize 메소드를 다음과 같이 구현한다.

```
function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
    hKeyProgID: HKEY): HRESULT;
var
    StgMedium: TStgMedium;
    FormatEtc: TFormatEtc;
begin
    try
        if lpdobj = nil then
            begin
                Result := E_FAIL;
                Exit;
            end;
        with FormatEtc do
            begin
                cfFormat := CF_HDROP;
                ptd := nil;
                dwAspect := DVASPECT_CONTENT;
                lindex := -1;
                tymed := TYMED_HGLOBAL;
            end;
        Result := lpdobj.GetData(FormatEtc, StgMedium);
        if Failed(Result) then Exit;
    try
        if DragQueryFile(StgMedium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
            begin
                DragQueryFile(StgMedium.hGlobal, 0, FFileName, SizeOf(FFileName));
                Result := NOERROR;
            end
        else
            Result := E_FAIL;
        finally
            ReleaseStgMedium(StgMedium);
        end;
    end;
```

```

        end;
    except
        Result := E_UNEXPECTED;
    end;
end;

```

이 구현 부분에서 핵심이 되는 것은 IDataObject 인터페이스인 lpobj 파라미터이다. 이 파라미터의 GetData 메소드를 이용하여 파일의 데이터를 가져오는 부분이다. 이를 제대로 이해하기 위해서는 TStgMedium과 TFormatEtc 구조체의 내용을 이해해야 한다.

TStgMedium 구조체는 IDataObject, IAdviseSink 인터페이스의 데이터 전송 작업에 사용되는 전역 메모리 핸들로 사용된다. tymed 필드는 저장되는 매체의 종류를 나타내는 것으로 hBitmap(비트맵), hGlobal(전역 메모리 핸들) 등의 값을 가질 수 있다. 이 필드는 공용체(union)와 같은 형태로 사용될 수 있다. 그러므로 StgMedium.hGlobal은 전역 메모리에 대한 핸들을 나타낸다. 더 자세한 내용은 Win32 SDK의 도움말을 참고하기 바란다.

TFormatEtc 구조체는 일반화된 클립보드 포맷이다. 그러므로 목적 디바이스와 데이터의 뷰, 저장 매체 등에 따라 다른 값들을 지정할 수 있다. OLE에서는 이 구조체를 사용하여 클립보드의 정보를 이용한다. cfFormat 멤버는 특정 클립보드 포맷을 나타내는 것으로 OLE에 의해 인식되는 형식은 CF\_TEXT와 같은 표준 포맷과 특정 어플리케이션에 의해서만 지원되는 포맷, 그리고 객체의 연결과 임베딩이 가능한 OLE 포맷 등이 있다. 여기서 사용한 CF\_HDROP 포맷은 TDropFiles 구조체를 포함한 전역 메모리 객체로, 이 객체는 클립보드에서 드래그-드롭 작업을 할 때 같이 복사된다. 어플리케이션은 데이터 객체에 대한 정보를 DragQueryFile이나 DragQueryPoint 함수에 객체의 핸들을 파라미터로 호출하여 얻을 수 있다.

ptd 멤버는 목적 디바이스에 대한 정보를 포함하는 구조체를 가리키는 멤버로 이 값이 nil이면 지정된 데이터 포맷이 목적 디바이스의 종류에 상관없이 적용된다는 의미이다. dwAspect 멤버는 특정 클립보드 포맷의 여러 뷰 중에서 어떤 것을 선택할 것인지를 결정한다. DVASPECT\_CONTENT는 객체를 컨테이너 내부에 임베딩된 형태로 표시한다는 의미이다. linindex 멤버는 데이터가 페이지 경계에 걸렸을 때 어떤 aspect를 보여줄 것인지를 결정하는 것으로 보통 -1을 지정하는데, 이것은 데이터 모두를 보여주라는 의미이다. 마지막으로 tymed 멤버는 TStgMedium의 내용과 같다. 더 자세한 내용은 Win32 SDK의 도움말을 참고하기 바란다.

어쨌든 이 코드의 의미는 드래그-드롭이 가능한 파일의 내용을 초기화하고, IDataObject 인터페이스의 GetData 메소드를 이용하여 StgMedium, FormatEtc 구조체에 의해 지정된 형태로 데이터를 가져오는 것이다. 그리고 나서, DragQueryFile 함수에서 2번째 파라미터를 \$FFFFFFFF로 지정하면 드롭된 파일의 수를 알아낼 수 있다. 여기서는 파일이 하나일 때만 열 수 있으므로 그 값이 1인 경우에만 다음으로 진행한다. 그 다음에는

DragQueryFile 을 다시 호출하여 FFileName 변수에 드롭된 파일의 버퍼를 지정한다.  
이렇게 Initialize 메소드를 구현함으로써 FFileName 변수를 이용해 파일에 접근할 수 있도록 초기화를 완료하였다.

이제 IContextMenu 인터페이스의 QueryContextMenu 메소드를 구현하도록 하자. 이 메소드는 메뉴 인덱스를 지정하고, 메뉴 아이템을 삽입하면 되므로 다음과 같이 간단히 구현이 가능하다.

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
    idCmdLast, uFlags: UINT): HRESULT;
begin
    FMenuIndex := indexMenu;
    InsertMenu(Menu, FMenuIndex, MF_STRING or MF_BYPOSITION, idCmdFirst,
        '내용 보기');
    Result := FMenuIndex + 1;
end;
```

이 메소드에 의해 팝업 메뉴에 ‘내용 보기’라는 메뉴 아이템이 추가된다.  
그러면, 팝업 메뉴에서 이 메뉴 아이템을 선택했을 때 호출되는 InvokeCommand 메소드를 다음과 같이 구현한다.

```
function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo): HRESULT;
begin
    Result := S_OK;
    try
        if HiWord(Integer(lpici.lpVerb)) <> 0 then
        begin
            Result := E_FAIL;
            Exit;
        end;
        if LoWord(lpici.lpVerb) = FMenuIndex then
            WinExec(PChar('Notepad.exe ' + FFileName), SW_SHOW)
        else
            Result := E_INVALIDARG;
    except
        MessageBox(lpici.hwnd, '파일을 볼 수 없습니다 !', 'Error',
            MB_OK or MB_ICONERROR);
    end;
```

```

        Result := E_FAIL;
    end;
end;

```

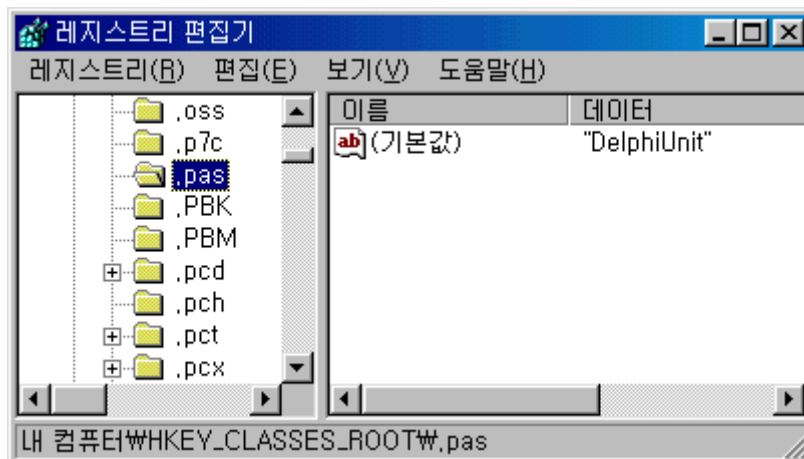
먼저 lpici 구조체의 lpVerb 멤버의 HiWord 를 검사하여 이 값이 0 인 경우에 제대로 선택된 경우이므로 계속 진행한다. 그리고, 이 멤버의 LoWord 값은 선택된 메뉴의 인덱스를 담고 있으므로 이 값이 FMenuIndex 값과 일치한다면 추가한 메뉴 아이템을 선택한 경우이므로 WinExec 함수를 이용하여 메모장에 해당 텔파이 유닛을 열도록 실행한다. 마지막으로 GetCommandString 메소드는 다음과 같이 구현하여 도움말 문자열을 제공하도록 한다.

```

function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
    pszName: LPSTR; cchMax: UINT): HRESULT;
begin
    Result := S_OK;
    try
        if (idCmd = FMenuIndex) and ((uType and GCS_HELPTEXT) <> 0) then
            StrLCopy(pszName, '파스칼 소스 코드를 메모장에 보여줍니다.', cchMax)
        else
            Result := E_INVALIDARG;
        except
            Result := E_UNEXPECTED;
        end;
    end;
end;

```

이것으로 IShellExtInit 인터페이스와 IContextMenu 인터페이스는 모두 구현되었다. 이제 부터는 .pas 파일에 대한 셸 확장 부분이 있음을 레지스트리에 등록해야한다. 이런 등록 작업을 하지 않으면 해당 확장자에 대한 정보가 없기 때문에 핸들러가 작동하지 않는다. 우리가 작업할 확장자는 .pas 이므로 .pas 확장자가 과연 어떤 시스템 객체를 가리키고 있는지를 먼저 알아야 한다. 윈도우 디렉토리의 RegEdit.exe 파일을 실행하면 시스템 레지스트릴 직접 표시하고, 편집할 수 있는데 HKEY\_CLASSES\_ROOT 키를 열면 먼저 파일의 확장자들이 나열될 것이다. 여기서 우리가 셸 확장을 시키려고 하는 .pas 확장자에 대한 키를 선택하면 오른쪽 pane 에 다음과 같이 해당 확장자에 대한 값을 보여줄 것이다. 여기서 ‘DelphiUnit’이 바로 .pas 파일에 대한 객체이다.



참고로 여기에 등록되어 있지 않은 확장자에 대한 컨텍스트 메뉴 핸들러를 제작하는 경우라면 확장자에 대한 키를 새로 등록하고, 등록된 값에 대해서 다음에 설명하는 과정을 같은 방법으로 사용해야 한다.

확장자들의 키들의 아래에 계속해서 나타나는 객체들의 키 중에서 DelphiUnit 키를 찾아보자, 이 키에는 서브 키로 DefaultIcon 과 Shell 이 존재할 것이다. 여기에 셸 확장 부분이 있다면 shellex 서브 키를 추가하면 되고, 컨텍스트 메뉴 핸들러가 존재하면 shellex 서브 키에 ContextMenuHandlers 서브 키를 추가하고 그 값으로 컨텍스트 메뉴 핸들러를 구현한 CoClass 의 CLSID 를 등록하면 된다.

이런 모든 과정은 직접 손으로 RegEdit.exe 프로그램을 이용해서 직접 입력해도 좋지만, 프로그램으로 처리하는 것이 더 좋다는 것은 당연하다.

이를 등록하기 위해서 가장 좋은 방법은 CoClass 가 처음 등록될 때, 레지스트리를 변경하는 과정에서 이들 내용을 같이 변경하도록 하는 것이다. 가장 세련된 형태로 이를 구현하는 것은 TComObjectFactory 클래스를 상속한 새로운 클래스 팩토리의 UpdateRegistry 메소드를 오버라이드하여 이를 구현하는 방법이다. 이때 컨텍스트 메뉴 핸들러는 ProgID 를 가지지 않으므로 GetProgID 메소드를 같이 오버라이드하여 ProgID 를 널 문자열로 지정하도록 하자.

그리고, 또 하나 고려해야 하는 것은 윈도우 NT 의 경우로 모든 셸 확장 부분에 대해 HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved 키에 그 내용을 기록해야 한다. 이를 위해 메소드를 하나 더 추가하도록 한다.

그러면 TContextMenuFactory 클래스를 다음과 같이 선언하고, initializatoin 섹션의 클래스 팩토리를 TContextMenuFactory 로 교체한다.

```
TContextMenuFactory = class(TComObjectFactory)
protected
```

```

function GetProgID: string; override;
procedure ApproveShellExtension(Register: Boolean; const CLSID: string);
    virtual;
public
    procedure UpdateRegistry(Register: Boolean); override;
end;

```

... (중략)

initialization

```

TContextMenuFactory.Create(ComServer, TContextMenu, Class_ContextMenu,
    'ContextMenu', '', ciMultInstance, tmApartment);
end.

```

GetProgID 메소드는 구현하기 쉽다. 단순히 ProgID 로 널 문자열을 넘겨주도록 하면 된다.

```

function TContextMenuFactory.GetProgID: string;
begin
    Result := '';
end;

```

UpdateRegistry 메소드는 앞에서 설명한 바대로 DelphiUnit 키에 대해 새로운 셸 확장에 대한 서브 키를 추가하고, 그 값으로 컨텍스트 메뉴 핸들러 객체의 CLSID 를 추가한다. 여기서 중간에 ApproveShellExtension 메소드를 호출하여 윈도우 NT 의 경우에 추가적인 레지스트리 등록 작업을 수행하도록 하면 된다.

```

procedure TContextMenuFactory.UpdateRegistry(Register: Boolean);
var
    CLSID: string;
begin
    CLSID := GUIDToString(ClassID);
    inherited UpdateRegistry(Register);
    ApproveShellExtension(Register, CLSID);
    if Register then
    begin
        CreateRegKey('DelphiUnitWshellexWContextMenuHandlersW' +

```

```

        ClassName, '', CLSID);
end
else begin
    DeleteRegKey('DelphiUnitWshellexWContextMenuHandlersW' +
        ClassName);
end;
end;

```

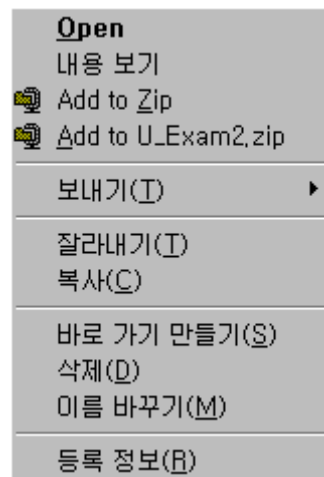
마지막으로 ApproveShellExtension 메소드는 다음과 같이 구현하여 해당 키에 값을 등록하도록 하면 된다.

```

procedure TContextMenuFactory.ApproveShellExtension(Register: Boolean;
    const CLSID: string);
const
    SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved';
begin
    with TRegistry.Create do
        try
            RootKey := HKEY_LOCAL_MACHINE;
            if not OpenKey(SApproveKey, True) then Exit;
            if Register then WriteString(CLSID, Description)
            else DeleteValue(CLSID);
        finally
            Free;
        end;
    end;
end;

```

이것으로 컨텍스트 메뉴 핸들러를 구현하였다. 컴파일하고 Run|Register ActiveX Server 메뉴를 선택하여 이를 등록하도록 하자. 그리고, 탐색기를 실행하고 .pas 확장자를 가진 파일에 커서를 위치시킨후 오른쪽 버튼을 클릭하면 다음과 같이 추가된 메뉴 아이템을 볼 수 있을 것이다.



‘내용 보기’ 메뉴 아이템을 선택하면 다음과 같이 메모장에 해당 파스칼 소스 파일이 열린다.

```

type
  TContextMenu = class(TComObject, IShellExtInit, IContextMenu)
  private
    FFileName: array[0..MAX_PATH] of Char;
    FMenuIndex: UINT;
  protected
    // IShellExtInit method
    function Initialize(pidlFolder: PItemIDList; lpdojb: IDataObject;
      hKeyProgID: HKEY): HRESULT; reintroduce; stdcall;
    // IContextMenu methods
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst, i
      uFlags: UINT): HRESULT; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HRESULT;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT
      pszName: LPSTR; cchMax: UINT): HRESULT; stdcall;
  end;

```

## 정 리 (Summary)

이번 장에서는 텔파이를 이용해 각종 셸 확장 기법을 이용하는 방법에 대해서 몇 가지 알아 보았다. 여기에 다른 내용 이외에도 셸 확장 기법에는 시스템 아이콘을 이용하는 방법도 있고, 파일 뷰어를 구현하고 OLE 객체의 진정한 드래그-드롭을 구현하는 등의 여러가지 기 법을 구현할 수 있다. 이들을 자유자재로 구현하는 좋은 셸 유틸리티를 개발하기 위해서는 COM 과 OLE 인터페이스에 대한 지식이 필수적이라는 것은 두말할 나위도 없다.

아마도, 필자가 나름대로 쉽게 풀어썼음에도 불구하고 내용이 난해하다고 느낀 독자가 많을



것이다. 그런 독자들은 제 4 부의 내용을 다시 한번 읽어보고 COM 에 대한 내용을 보다 확실하게 익힌 후에 다시한번 도전해 보기 바란다. 기본적인 원리만 이해한다면 셸 확장 기법을 이용한 환상적인 유틸리티를 작성하는 것은 이미 누워서 떡먹기와도 다름 없다.

참고로, 잘된 셸 확장 유틸리티를 작성하기 위해서는 Win32 SDK 를 비롯하여 마이크로소프트에서 제공하는 MSDN 과 같은 문서자료를 많이 참고하는 것이 필수적이다.

아무쪼록, 우리나라에서도 외국에서 나오는 윈도우 95/98 전용의 셸 확장 유틸리티를 능가하는 한국형 셸 확장 유틸리티가 나오기를 기대한다. 도스의 MDir 은 그 기능과 편리성에서 세계적인 수준이었음에도 불구하고 윈도우 시장에는 그런 작품이 국내에서 나오지 못한 것이 매우 안타깝다고 생각된다.

필자의 개인적인 생각으로는 비주얼 C++ 보다 델파이가 이런 셸 확장 유틸리티를 작성하는데 훨씬 유리하다고 생각하고 있다. 이번 강의 내용이 한국형 셸 확장 유틸리티 탄생의 초석이 되기를 간절히 바라면서 이번 장을 마무리하고자 한다.

# 델파이 4의 메모리 관리 기법과 메모리 맵 파일

## (Memory Management Techniques in Delphi 4 and Memory Mapped File)

마이크로소프트 Win32 API에서는 각각의 프로세스는 4GB까지 메모리를 가질 수 있는 32비트 가상 주소 공간을 가진다. 하위 2GB 메모리(\$0~\$7FFFFFFF)는 사용자가 사용할 수 있으며, 상위 2GB 메모리(\$80000000~\$FFFFFFFF)는 커널에 의해 예약되어 있다. 프로세스에 의해 사용되는 가상 주소는 메모리에서의 객체의 실제 물리적인 위치를 나타내는 것은 아니다. 커널이 관리하는 각 프로세스는 가상 주소를 물리적 주소로 변환하는 페이지 맵을 관리한다.

### 가상 주소 공간과 물리적 저장 공간

각 프로세스의 가상 주소는 물리적 메모리인 RAM의 전체 크기보다 크다. 물리적 저장 공간을 증가시키기 위해 커널은 디스크 공간을 사용한다. 그러므로, 실행 중인 프로세스들이 사용할 수 있는 저장 공간의 총량은 RAM의 크기에 페이지 파일로 쓸 수 있는 디스크 공간의 크기를 합한 것이 된다.

메모리 관리의 유연성을 극대화하기 위해서 커널은 물리적 메모리의 페이지를 디스크의 페이지 파일로 옮길 수 있도록 허용한다. 물리적 메모리가 옮겨지면 커널은 페이지 맵을 재구성하게 된다. 커널이 물리적 메모리에 공간을 필요로 하면, 가장 오래 전에 사용했던 물리적 메모리 공간을 페이지 파일로 옮기게 된다.

프로세스의 가상 주소 공간의 페이지 들은 다음에 설명되는 상태 중 하나에 있게 된다.

#### 1. Free

현재 접근이 가능하지 않지만, 언제든지 예약 또는 사용 가능한 페이지이다.

#### 2. Reserved

프로세스의 가상 주소 공간의 블록으로 앞으로 사용하기 위해 예약한 페이지이다. 다른 프로세스가 이 페이지에 접근할 수 없으며, 이 페이지와 연결된 물리적 메모리도 없다. 예약된 페이지는 다른 메모리 할당 작업에 의해 접근할 수 없는 가상 주소의 범위를 정하게 된다. 이런 주소 공간을 예약하기 위해서 프로세스는 VirtualAlloc API 함수를 사용하며, 이

를 해제하기 위해서는 VirtualFree 함수를 사용한다.

### 3. Committed

Committed 페이지는 실제 할당된 물리적 저장 공간(RAM 또는 디스크)에 해당되는 페이지이다. 이 페이지는 접근이 불가능하게 하거나, 읽기 전용으로 보호할 수 있으며 경우에 따라서는 읽고 쓰기를 자유롭게 허용할 수도 있다. 이렇게 실제 메모리를 할당하기 위해서 역시 VirtualAlloc 함수를 사용할 수 있다. 동시에 읽고 쓰기 접근을 가능하게 할 경우에는 GlobalAlloc, LocalAlloc API 함수를 사용할 수도 있다. 일단 VirtualAlloc 함수로 할당된 페이지는 VirtualFree 함수로 해제할 수 있으며, 이렇게 되면 페이지의 저장 공간은 해제되지만 그 페이지의 상태는 'reserverd'로 바뀌게 된다.

## 전역과 지역 메모리 관리 함수

프로세스가 메모리를 할당할 때 GlobalAlloc, LocalAlloc 함수를 사용할 수 있다. Win32와 같은 32 비트 선형 메모리 시스템에서는 로컬 힙(heap)과 글로벌 힙이 사실상 구별되지 않기 때문에, 이들 함수에 의해 할당받는 메모리 객체에는 차이가 없다.

이들 함수에 의해 할당받는 메모리 객체는 committed 페이지로, 읽기 쓰기가 모두 가능하다. 이들은 또한 private 메모리로 사용되기 때문에, 다른 프로세스가 접근할 수 없게 된다. GlobalAlloc 함수를 호출할 때에는 GMEM\_DDESHARE 라는 플래그를 설정할 수도 있는데, 실제로 메모리 공유가 되지는 않는다. 다만, 이 플래그는 Win16 API와의 호환성을 위해서 남겨둔 것으로, 일부 어플리케이션에서 DDE 작업의 효율성을 높이기 위해서 사용될 수 있다. 만약에 프로세스 간에 메모리를 공유하고 싶을 경우에는 반드시 파일-매핑 객체를 사용해야 한다.

GlobalAlloc, LocalAlloc 함수를 사용함으로써 32 비트로 표현할 수 있는 어떤 크기의 메모리 블록도 할당받을 수 있다. 다만, 이들이 따로 존재하는 이유는 Win16 과의 호환성을 위해서 이다. 어쨌든 16 비트의 세그먼트로 나뉜 메모리 모델에서 32 비트 가상 메모리 모델로의 변화는 일부의 함수의 기능과 옵션을 의미 없는 것으로 바꾸어 놓았다. 예를 들어, 이제 더 이상 'far'와 같은 예약어를 사용할 필요가 없어졌다.

GlobalAlloc, LocalAlloc 함수 모두 고정된(fixed)거나 이동가능(movable)한 메모리 객체를 할당할 수 있다. 이동가능(movable)한 객체들은 'discardable'로 간주된다. 과거 윈도우 3.1 시절에는 이러한 이동가능한 메모리 객체를 사용하는 것이 메모리 관리 기법으로 중요시 되었다. 이를 이용하면 시스템이 힙의 크기를 절약할 수 있어서, 다른 메모리에 대한 할당 작업을 원활하게 할 수 있었다. 가상 메모리를 이용하면 시스템은 물리적 메모리를 디스크에 있는 페이지 파일로 옮길 수 있는데, 이렇게 옮겨진 부분의 메모리에 대한 가상 메모리 페이지 맵을 수정하는 것으로 쉽게 구현된다. 대신 이동가능한 메모리로 할당하면

시스템이 추가적인 물리적 저장 공간을 필요로 할 때, 가장 오래 전에 사용한, 락(lock)이 걸리지 않은 이동가능한 메모리를 이용하게 할 수 있다. 그러므로, 이동가능한 메모리는 자주 사용되지 않고, 쉽게 다시 생성할 수 있는 메모리의 경우에 사용해야 한다.

고정된 메모리 객체를 할당할 때에는 GlobalAlloc, LocalAlloc 함수가 직접 메모리에 접근할 수 있도록 32 비트 포인터를 반환한다. 그에 비해, 이동가능한 메모리의 경우에는 메모리에 대한 핸들을 반환한다. 만약에 이동가능한 메모리 객체에 대한 포인터를 얻고 싶을 때에는 반드시 GlobalLock, LocalLock 함수를 이용해야 한다. 이들 함수는 메모리를 이동하거나 버릴 수 없도록 고정시키는 역할을 하게 된다. 각 메모리 객체의 내부 데이터 구조에는 0 부터 시작하는 잠금 계수(lock count)가 존재한다. 이동가능한 메모리에서 GlobalLock, LocalLock 함수를 호출하면 이 계수가 하나 증가하며 GlobalUnlock, LocalUnlock 함수를 호출하면 하나 감소한다. 잠긴 메모리는 GlobalReAlloc 이나 LocalReAlloc 함수에 의해 재할당 받지 않는 한 이동과 버림이 불가능하다. 만약 메모리에 대한 UnLock 함수에 의해 잠금 계수가 0 으로 감소하면 그 때부터 이 메모리 객체는 이동과 버림이 가능해진다.

GlobalAlloc 또는 LocalAlloc 함수에 의해 할당받게 되는 메모리의 실제 크기는 요구한 크기보다 클 수도 있다. 이때 실제로 할당받은 메모리의 정확한 바이트 크기를 정할 때 사용되는 API 함수가 GlobalSize, LocalSize 함수이다. 만약에 할당된 크기가 요구한 크기보다 큰 경우에는 프로세스가 이를 모두 사용할 수 있다. GlobalAlloc, LocalAlloc 함수에 의해 할당받은 메모리의 크기와 속성을 변경할 때에는 GlobalReAlloc, LocalReAlloc 함수를 사용할 수 있다. 메모리 객체를 해제할 때에는 공히 GlobalFree, LocalFree 함수가 사용된다. 그밖에 지정된 메모리 객체에 대한 정보를 얻을 때에는 GlobalFlags, LocalFlags 함수가 사용될 수 있으며 지정된 포인터와 연관된 메모리 객체를 알아낼 때에는 GlobalHandle, LocalHandle 함수가 사용된다.

## 가상 메모리 함수

Win32 API 는 가상 주소 공간에 존재하는 메모리 페이지의 상태를 결정하거나, 관리하는 여러가지 가상 메모리 함수를 제공한다. 많은 수의 어플리케이션이 앞에서 설명한 GlobalAlloc, LocalAlloc 함수를 사용해서 Win16 과의 호환성을 제공하지만 가상 메모리 함수는 앞의 함수 들이 제공할 수 없는 기능을 제공한다. 이 함수들을 이용해서 할 수 있는 작업에는 다음과 같은 것들이 있다.

1. 프로세스의 가상 주소 공간의 범위를 예약한다. 이러한 예약 행위는 물리적 저장 공간을 할당하지 않지만, 지정된 주소 공간 범위에 대해 다른 메모리 할당 작업이 접근할 수 없도록 한다. 이 작업은 다른 프로세스의 가상 주소 공간에 영향을 미치지 않는다. 이렇게 사용할 페이지를 예약함으로써 물리적 저장 공간의 불필요한 소모를 막을 수 있

고, 동적인 데이터 구조가 커질 때에 효과적으로 이용할 수 있게 한다.

2. 예약된 프로세스의 가상 주소 공간의 범위를 실제 물리적 저장 공간에서 사용할 수 있도록 할 수 있다. (Commit)
3. 사용하는 페이지의 접근 속성을 읽기-쓰기, 읽기 전용, 접근 불가로 설정할 수 있다. 이는 앞에서 설명한 GlobalAlloc, LocalAlloc 등의 표준 함수들이 읽기-쓰기로만 설정되는 것에 비해 다른 점이다.
4. 예약된 페이지의 범위를 해제할 수 있다.
5. Commit 된 메모리 페이지를 decommit 할 수 있다. 이로 인해 물리적 저장 공간이 해제되며, 다른 프로세스에 의해 이들이 사용될 수 있다
6. 하나 이상의 committed 메모리를 물리적 메모리(RAM)로 lock 해서 시스템이 페이지 파일로 스왑하지 않도록 할 수 있다.
7. 페이지에 대한 정보를 얻을 수 있다.

가상 메모리 함수들은 메모리 페이지를 관리한다. 이 함수들은 현재 컴퓨터에 설정된 페이지 파일의 크기를 이용해서 메모리 페이지의 크기와 주소를 결정하는데, 이를 알아보기 위해서는 GetSystemInfo 함수를 사용한다.

이러한 가상 메모리 함수들에는 VirtualAlloc, VirtualFree, VirtualLock, VirtualUnlock 등의 가장 기본적인 함수들과 메모리 페이지의 정보를 얻을 때 사용하는 VirtualQuery, VirtualQueryEx 함수가 있다.

그 밖에 VirtualProtect, VirtualProtectEx 함수를 이용하면 프로세스의 접근 권한 등을 변경할 수 있다.

## 힙(Heap) 함수

힙 함수는 호출하는 프로세스의 주소 공간에서 하나 이상의 페이지로 된 메모리 블록인 private 힙을 관리할 때 사용된다. 이렇게 할당 받은 private 힙과 일반적인 메모리 할당 함수에 의해 할당 받은 메모리는 근본적으로 같다.

HeapCreate 함수는 HeapAlloc 함수에 의해 할당 받은 private 힙 객체를 생성한다. 이 함수는 힙의 초기 크기와 최대 크기를 지정할 수 있다. 초기 크기는 힙을 처음 할당 받을 때 committed 된 읽기-쓰기가 가능한 페이지를 결정하며, 최대 크기는 예약된 페이지의 총 크기를 결정한다. 이들 페이지 들은 프로세스의 가상 메모리 공간에서 연속된 블록으로 구성되므로 힙이 동적으로 커져갈 수 있다. 만약에 HeapAlloc 함수에 의해 요구된 메모리의 크기가 현재의 committed 페이지의 크기보다 클 경우에는 자동으로 추가적인 페이지가 commit 된다. 일단 이렇게 페이지가 commit 되면, 프로세스가 종료되거나 HeapDestroy 함수에 의해 힙이 파괴되기 전에는 decommit 할 수 없다.

Private 힙 객체의 메모리는 이를 생성한 프로세스에 의해서만 접근이 가능하다. 그 밖에

HeapFree, HeapSize 함수를 사용할 수 있다.

HeapAlloc 함수에 의해 할당된 메모리는 이동할 수 없으며, 분절로 나뉘어 질 수도 있다.

이러한 힙 함수를 사용하는 것은 프로세스가 처음 시작할 때나, 프로세스에 필요한 메모리의 충분한 크기를 지정할 때이며, 만약 HeapCreate 함수 호출이 실패하면 프로세스는 사용자에게 메모리 부족을 알릴 수 있다.

## 공유 메모리

Win32 API 에서 공유 메모리는 파일 매핑에 의해 구현될 수 있다. 다른 메모리 할당 메소드에 의해서 할당된 메모리는 단지 호출한 프로세스에서만 접근할 수 있다.

파일 매핑을 이용하면 쉽게 공유 메모리 블록을 생성할 수 있다. 프로세스는 CreateFileMapping 함수를 사용할 때 파일 매핑 객체를 생성하기 위해 이름을 지정할 수 있으며, 다른 프로세스에서 이 이름을 이용해서 CreateFileMapping, OpenFileMapping 함수를 호출하면 매핑 객체의 핸들을 얻을 수 있게 된다. 이벤트 객체, 세마포어(semaphore) 객체, 뮷텍(mutex) 객체와 파일 매핑 객체는 같은 이름 공간(name space)을 공유한다. 만약 지정된 이름이 이미 존재하고 있는 다른 종류의 객체와 같으면 에러가 발생한다. 그러므로 이런 객체를 생성할 때에는 이름을 유일하게 지정하도록 해서, 중복을 피해야 한다.

각각의 프로세스는 파일 매핑 객체의 핸들을 MapViewOfFile 함수에서 지정하여 자신의 주소 공간에 매핑한다. 이때 모든 프로세스 들은 하나의 파일 매핑 객체를 공유가 가능한 물리적 저장 공간의 같은 위치를 매핑하게 된다. 그렇지만, 이들의 가상 주소는 프로세스들마다 다르다.

매핑된 뷰가 물리적 메모리에서 디스크로 스왑되었을 때에는 파일 매핑 객체가 시스템이 사용하는 디스크 파일과 연관되며, 파일 매핑 객체가 생성될 때 지정된 다른 파일이 된다. 이런 경우에 메모리는 파일의 내용에 따라 초기화될 수 있다. 파일 시스템에서 지정된 파일을 매핑하는 것은 이미 존재하는 파일의 데이터를 공유하거나, 공유 프로세스에 의해 생성되는 데이터를 파일에 저장하고자 할 때 매우 유용하다. 만약 지정된 파일을 매핑하면, 이를 exclusive 접근 권한으로 파일을 열어야 하며, 공유 메모리에 대한 작업이 끝날 때까지 이 파일을 계속 열어 두어야 한다. 이렇게 파일을 열어두면 다른 프로세스가 이 파일에 대해 접근할 수 없게 된다.

이렇게 매핑된 파일은 매핑 객체를 이용하는 마지막 프로세스가 종료되거나, UnmapViewOfFile 함수에 의해 매핑이 제거될 때 업데이트 된다. 이러한 업데이트 작업은 FlushViewOfFile 함수에 의해 강요될 수도 있다. 공유 메모리에 대해서는 더 자세하게 다룰 것이다.

## 델파이의 메모리 관리

델파이에서 사용하는 메모리 관리자(Memory Manager)에 대해 알아보도록 하자. 델파이에서 표준으로 쓰이는 프로시저인 New, Dispose, GetMem, ReallocMem, FreeMem 등이 모두 메모리 관리자를 사용한다. 델파이의 System 유닛에 선언되어 있는 메모리 관리자 레코드의 선언부분을 살펴보자.

```
PMemoryManager = ^TMemoryManager;
TMemoryManager = record
    GetMem: function(Size: Integer): Pointer;
    FreeMem: function(P: Pointer): Integer;
    ReallocMem: function(P: Pointer; Size: Integer): Pointer;
end;
```

사실 이들의 구현 부분을 살펴 보면 위에서 선언된 프로시저 형을 이용해서 실제 구현되는 부분은 인라인 어셈블리로 모두 구현되어 있다. 여기에 대한 자세한 설명은 필자도 정확히 이해하고 있지 못하거니와 이 책에서 다룰 수 있는 범위가 아니라고 생각되므로 생략하도록 한다. 다만 다음 정도를 이해하도록 하자.

델파이가 사용하는 모든 객체와 문자열은 이들 루틴에 의해 내부적으로 관리되고 있다. 이들 루틴의 장점은 small~medium size 의 많은 수의 블록을 관리하는데 최적화 되어 있다는 것이다. 이런 형태의 메모리 관리는 객체를 많이 사용하거나, 문자열을 처리하는 어플리케이션에 적합하다. 사실 이들 외에도 GlobalAlloc, LocalAlloc 등을 이용하여, 윈도우에서 private heap 을 사용하도록 할 수도 있지만, 이들을 사용하면 일반적으로 어플리케이션의 속도가 느려진다.

델파이의 메모리 관리자는 Win32 가상 메모리 API 인 VirtualAlloc, VirtualFree 함수를 직접 이용하기 때문에 매우 효율적이다.

각 메모리 블록은 약 4 바이트 정도의 헤더를 가지고 있는데 여기에 각 블록의 크기를 담고 있다. 그리고, 메모리 관리를 위한 상태 변수로 AllocMemCount, AllocMemSize 가 이용된다. 이들은 각각 현재 할당된 메모리 블록의 수와 할당된 메모리 크기를 담고 있다. 이들을 적절히 이용하면 디버깅을 할 때 현재 어플리케이션에서의 메모리 사용의 문제점을 파악하기 쉬워진다.

System 유닛에서는 이 밖에도 GetMemoryManager, SetMemoryManager 라는 프로시저를 제공하는데, 이들을 이용하면 어플리케이션에서 메모리 관리자를 사용하는 호출을 중간에서 가로챌 수 있다. 또한, GetHeapStatus 라는 함수를 이용하면 메모리 관리자의 상태 정보를 담은 레코드를 넘겨받을 수 있다

GetMemoryManager 는 현재의 메모리 관리 루틴을 반환하며, SetMemoryManager 프로시

저를 호출하여 TMemoryManager 레코드에 새로운 함수 포인터를 설정할 수 있다. 메모리 관리자 레코드의 선언부에서 볼 수 있듯이, 핵심이 되는 메모리 관리자 함수는 GetMem, FreeMem 과 ReallocMem 이다. 그런데, 주의할 점은 FreeMem 의 경우 메모리를 해제할 때 델파이에게 해제될 메모리의 크기를 알려주도록 되어 있는데, 메모리 관리자의 선언부를 보면 크기에 대한 파라미터가 없다. 이는 스스로 할당된 메모리의 크기를 결정할 수 있어야 한다는 것이다. GetMem 함수는 블록 반위로 메모리를 할당하고 포인터를 반환하는데, 더 이상 할당할 메모리가 없다면 nil 을 반환하게 되며 nil 이 돌아오면 델파이 가 예외를 발생시킨다. ReallocMem 함수는 주어진 크기의 메모리를 블록에 재할당하고 할당된 메모리의 포인터를 반환하는 역할을 한다. 만약 수행할 메모리가 충분하지 않은 경우에는 nil 을 반환하며 델파이는 예외를 발생시킨다.

그렇다면, 실제로 메모리 관리자를 변경하여 사용하는 예를 알아보도록 하자. 가장 전형적인 예는 델파이가 DLL 을 사용할 때 VCL 에 접근하게 될 경우 ShareMem.pas 유닛을 가장 처음에 선언해야 한다는 메시지를 접한 적이 있을 것이다. 이 ShareMem.pas 유닛의 내용이 바로 메모리 관리자를 DLL 에서와 같이 메모리를 공유할 때 적합한 함수들로 변경하는 작업을 하는 것이다. 소스를 잠시 살펴보면 다음과 같다.

```
unit ShareMem;
```

```
... (중략)
```

```
const
```

```
    DelphiMM = 'borIndmm.dll';
```

```
function SysGetMem(Size: Integer): Pointer;
```

```
    external DelphiMM name '@BorIndmm@SysGetMem$qqri';
```

```
function SysFreeMem(P: Pointer): Integer;
```

```
    external DelphiMM name '@BorIndmm@SysFreeMem$qqrpv';
```

```
function SysReallocMem(P: Pointer; Size: Integer): Pointer;
```

```
    external DelphiMM name '@BorIndmm@SysReallocMem$qqrpvi';
```

```
function GetHeapStatus: THeapStatus; external DelphiMM;
```

```
function GetAllocMemCount: Integer; external DelphiMM;
```

```
function GetAllocMemSize: Integer; external DelphiMM;
```

```
procedure DumpBlocks; external DelphiMM;
```

```
const
```

```
    SharedMemoryManager: TMemoryManager = (
```



```

GetMem: SysGetMem;
FreeMem: SysFreeMem;
ReallocMem: SysReallocMem);

```

initialization

```

if not IsMemoryManagerSet then
    SetMemoryManager(SharedMemoryManager);

```

end.

즉 SysGetMem, SysFreeMem, SysReallocMem 을 비롯한 핵심 메모리 관리 함수들을 borlndmm.dll 파일의 함수로 선언하고, SetMemoryManager 함수를 이용하여 메모리 관리자로 설정하는 것이다.

마찬가지로 경우에 따라서는 개발자가 자신의 메모리 관리자를 작성하여 이를 이용하도록 할 수도 있다. 그러면 간단한 사용자 정의 메모리 관리자를 다음과 같이 작성해보자. 여기서는 HeapAlloc, HeapFree, HeapRealloc API 함수를 이용하도록 한다.

```

unit SampleMgr;

```

```

interface

```

```

implementation

```

```

uses Windows;

```

```

var

```

```

    Heap: THandle;
    MemMgr: TMemoryManager;

```

```

function ExamGetMem(Size: Integer): Pointer;

```

```

begin

```

```

    Result := HeapAlloc(Heap, 0, Size);

```

```

end;

```

```

function ExamFreeMem(P: Pointer): Integer;

```

```

begin

```

```

    if HeapFree(Heap, 0, P) then Result := 0 else Result := 1;

```

```

end;

```

```

function ExamReallocMem(P: Pointer; Size: Integer): Pointer;
begin
    Result := HeapRealloc(Heap, 0, P, Size);
end;

initialization
begin
    Heap := GetProcessHeap;
    MemMgr.GetMem := ExamGetMem;
    MemMgr.FreeMem := ExamFreeMem;
    MemMgr.ReallocMem := ExamReallocMem;
    SetMemoryManager(MemMgr);
end;

end.

```

그러면, 메모리 관리자를 테스트해 보자. 테스트 폼에는 다음과 같이 TSpinEdit 컴포넌트와 버튼을 하나씩 얹고 SpinEdit1 컴포넌트의 MinValue, MaxValue 프로퍼티를 각각 1, 65536 으로 설정한다. 그리고 Value 프로퍼티를 4 로 설정하여 초기 할당 메모리의 크기를 4KB 로 설정한다.



그리고, Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    P: Pointer;
    i, StartTick, StopTick: Integer;
begin
    StartTick := GetTickCount;
    for i := 1 to 1000 do

```

```

begin
    GetMem(P, SpinEdit1.Value * 1024);
    FreeMem(P);
end;
StopTick := GetTickCount;
Button1.Caption := IntToStr(StopTick - StartTick);
end;

```

그러면, 델파이의 디폴트 메모리 관리자와 방금 작성한 메모리 관리자의 성능을 비교해 보도록 하자. 방금 작성한 프로젝트를 바로 컴파일해서 실행하면 델파이의 디폴트 메모리 관리자를 사용하게 된다. 만약 조금 전에 작성한 SampleMgr.pas 유닛의 메모리 관리 함수를 이용하고자 한다면, 다음과 같이 프로젝트 파일의 uses 절에 제일 앞에 SampleMgr.pas를 추가해야 한다.

```

program ExamTst2;

uses
    SampleMgr,
    Forms,
    U_ExamTst2 in 'U_ExamTst2.pas' {Form1};

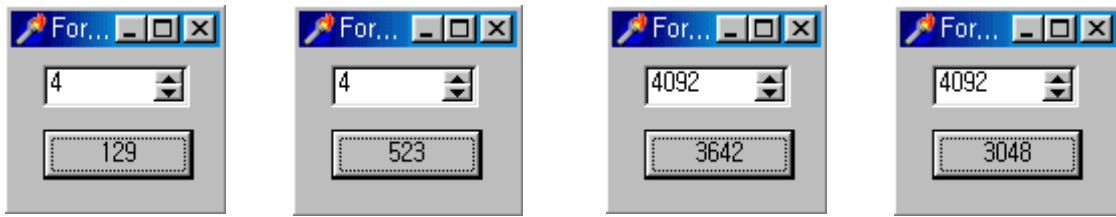
{$R *.RES}

begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.

```

다음 결과는 델파이 디폴트 핸들러를 사용한 경우와 SampleMgr의 메모리 관리 함수를 사용한 경우의 차이를 보여준 경우이다. 여기서는 4KB의 메모리의 할당, 해제를 반복한 것과 4MB의 메모리의 할당, 해제를 반복한 경우의 결과를 보여주고 있다.

결과를 보면 작은 메모리를 이용할 때 델파이의 메모리 관리자가 대단히 효율적으로 동작하지만 커다란 메모리에 대해서는 Win32 API의 HeapAlloc, HeapFree API 함수를 사용하는 것이 더 효율적인 것을 알 수가 있다.



## 파일 매핑

파일 매핑은 파일의 내용과 프로세스의 가상 주소 공간의 일부를 연관시키는 작업이라고 이해하면 된다. 이때 운영체제는 이를 위한 파일 매핑 객체를 생성한다. 파일 뷰(file view)는 파일의 내용에 접근하는데 사용되는 가상 주소 공간의 일부라고 생각하면 된다. 프로세스는 파일 뷰에 데이터를 읽고 쓸 때 일반적으로 동적으로 할당한 메모리를 사용할 때와 마찬가지로 포인터를 이용하면 된다. 또한, VirtualProtect API 함수를 이용하면 다른 가상 메모리 처럼 파일 뷰를 이용할 수 있다.

윈도우 NT에서는 이러한 개념이 비슷하지만, 다소 다르게 사용되는데 이를 간단히 정리하면 다음과 같다.

파일 매핑은 파일의 내용을 프로세스의 가상 주소 공간으로 복사하는 것이다. 파일의 내용에 대한 복사본을 파일 뷰라고 하며, 운영체제가 사용하는 복사본에 대한 내부 구조체를 파일 매핑 객체라고 한다.

윈도우는 데이터를 파일에서 읽어와서 파일 뷰에 기록한, 나중에 이를 다시 파일에 기록한다. 사실 상 파일 매핑은 시스템에게 지정된 프로세스의 가상 주소 공간을 지정된 파일이나 시스템 페이지 파일을 이용하게 하는 방법이라고 이해하면 된다.

파일을 매핑하면 프로세스가 디스크 상에 있는 파일에 동적으로 할당받은 메모리에 접근하듯이 접근이 가능하며, 더 나아가서는 하나 이상의 프로세스가 메모리를 공유할 수 있는 수단이 제공된다.

### ● 파일 매핑의 구현 단계

파일 매핑의 첫번째 단계는 CreateFile 함수를 호출해서 사용할 파일을 여는 것이다. 이 파일은 반드시 다른 프로세스가 접근할 수 없는 상태(exclusive)로 열어야 문제가 생기지 않는다. 가장 쉬운 방법은 CreateFile 함수의 fdwShareMode 파라미터를 0 으로 설정하는 것이다.

CreateFileMapping 함수는 CreateFile 에 의해서 얻은 핸들을 이용해서 파일 매핑 객체를 생성하는 함수이다. 이 함수를 이용해서 파일 매핑 객체의 이름과 파일에서 매핑할 데이터의 크기 (바이트 수), 매핑된 메모리에 접근할 권한 등을 설정하게 된다. 이 함수가 실패하는 경우는 CreateFile 에 의해 열린 파일과 지정된 파일이 일치하되, 접근 권한 플래그에서 충돌이 있게 설정된 경우가 많다. 예를 들어, 파일에 데이터를 읽고 쓰기 위해서는

CreateFile 함수의 fdwAccess 파라미터를 GENERIC\_READ 와 GENERIC\_WRITE 로 설정하고, CreateFileMapping 함수의 fdwProtect 파라미터를 PAGE\_READWRITE 로 설정해야 한다.

파일 매핑은 파일 시스템이 허용하는 한도의 커다란 파일까지 매핑할 수 있다. CreateFileMapping 함수의 dwMaximumSizeHigh, dwMaximumSizeLow 파라미터는 파일에서 매핑될 바이트 수를 지정하는 것으로 파일 매핑의 크기는 실제로 매핑될 파일의 크기와는 독립적이다.

어쨌든 매핑이 파일보다 큰 경우에는 시스템이 CreateFileMapping 함수가 리턴값을 반환하기 전에 파일의 크기를 증가시킨다. 반대로 파일 매핑의 크기가 파일의 크기보다 작으면 파일에서 지정된 크기만큼의 부분만 매핑된다.

어떤 경우에는 파일의 크기를 변경시키고 싶지 않은 경우가 있다 (예를 들어, 읽기 전용 파일을 매핑할 경우). 이럴 때에는 CreateFileMapping 함수의 dwMaximumSizeHigh, dwMaximumSizeLow 파라미터의 값을 모두 0 으로 지정한다. 이렇게 하면 파일 매핑 객체는 파일의 크기와 동일한 크기로 설정된다. 그렇지 않으면, 일단 파일 매핑 객체가 생성되면 매핑 크기가 더 커지거나 작아지지 않으므로, 파일의 크기를 정확하게 계산해서 매핑을 해야 한다.

이제는 파일의 데이터를 실제 메모리로 매핑할 차례이다. 이를 위해서는 앞에서도 잠시 언급한 파일의 뷰를 생성해야 한다. 이때 MapViewOfFile 과 MapViewOfFileEx API 함수를 사용한다. CreateFileMapping 함수를 호출하면 파일 매핑 객체의 핸들을 얻을 수 있는데, 파일의 뷰를 생성하거나 프로세스의 가상 주소 공간 내의 파일의 일부분에 대한 뷰를 생성할 때 이 핸들을 이용한다. MapViewOfFileEx 함수는 기본적으로 MapViewOfFile 함수와 같은 역할을 하는데, 프로세스가 파일의 뷰에 대한 base 주소를 지정할 수 있다는 점이 다르다.

MapViewOfFile 함수는 파일 뷰에 대한 포인터를 반환한다. 어플리케이션은 단순히 이 포인터를 이용해서 동적 메모리에 접근하듯이 데이터를 읽고 쓰면 된다. 실제로 읽고 쓰는 작업은 시스템에 의해 디스크 상의 파일에 반영된다. 그런데, 데이터가 파일 매핑 객체에 기록되면, 이것이 바로 파일로 전달되지는 않는다. 이런 데이터는 보통 캐쉬 메모리에 저장되었다가 마지막에 파일에 기록된다. 어플리케이션에서 만약에 그때 그때 데이터를 디스크에 기록하고 싶은 경우에는 FlushViewOfFile 함수를 이용한다.

어플리케이션은 같은 파일 매핑 객체에서 여러 개의 파일 뷰를 생성할 수 있다. 이들 각각의 파일 뷰는 크기가 다르며, 이들의 옵셋은 MapViewOfFile 함수의 dwOffsetHigh, dwOffsetLow 파라미터에 의해 지정된다. 이때 시스템의 메모리 할당에 대한 정보를 알고 싶은 경우에는 GetSystemInfo 함수를 호출하여 SYSTEM\_INFO 구조체에 정보를 얻어오면 된다.

파일 매핑 객체의 사용이 끝나면, 프로세스는 먼저 모든 파일 뷰를 파괴해야 하는데, 이때 사용하는 함수가 UnmapViewOfFile 함수이다. 참고로 FlushViewOfFile 함수를 이용해서

데이터를 디스크에 기록할 때에도 먼저 `UnmapViewOfFile` 함수를 호출해야 한다.

파일 매핑 객체와 파일은 모두 `CloseHandle` 함수를 호출해서 닫을 수 있다. 먼저 파일 매핑 객체를 닫고, 나중에 파일을 닫는다. 파일 매핑 객체를 닫은 뒤에 필요하면 프로세스가 파일의 크기를 지정할 수 있는데, 이때에는 `SetFilePointer` 와 `SetEndOfFile` 함수를 이용한다. 예를 들어, 매핑이 파일의 크기보다 큰 경우에는 프로세스가 파일 포인터를 요구되는 파일의 크기로 설정할 수 있다.

## ● 파일과 메모리의 공유

파일 매핑은 두 개 이상의 프로세스에 의해 파일이나 메모리가 공유될 때 유용하게 사용될 수 있는 기법이다. 이렇게 하려면, 관련된 모든 프로세스가 같은 파일 매핑 객체와 파일 뷰를 사용하면 된다. 각각의 프로세스의 뷰는 프로세스의 가상 주소 공간에 위치한다. 만약에 하나의 프로세스가 뷰를 업데이트 하면, 다른 프로세스 들도 이러한 업데이트를 즉시 알 수 있다. 파일을 공유하려면 첫번째 프로세스가 `CreateFile` 함수를 이용해서 파일을 생성하거나 연다. 그리고, `CreateFileMapping` 함수에서 파일 매핑 객체에 대한 이름을 지정하여 파일 매핑 객체를 생성한다. 파일과 연관되지 않는 메모리를 공유하기 위해서는 프로세스는 반드시 `CreateFileMapping` 함수를 이용하되 이미 존재하는 파일의 핸들을 지정하는 대신, 파라미터로 `$FFFFFFFF` 를 지정한다. 이렇게 하면 파일 매핑 객체는 시스템 페이지 파일을 이용해서 메모리에 접근하게 된다. 그리고, 이 경우에는 반드시 매핑의 크기를 0 보다 크게 설정해야 한다.

다른 프로세스들이 생성된 파일 매핑 객체의 핸들을 얻는 가장 좋은 방법은 `OpenFileMapping` 함수에 파일 매핑 객체의 이름을 지정해서 호출하는 것이다. 만약 파일 매핑 객체가 이름이 없을 경우에는 프로세스는 상속(`inheritance`)나 복제(`duplication`)을 이용해서 핸들을 얻어와야 한다.

파일이나 메모리를 공유하는 프로세스 들은 반드시 `MapViewOfFile` 을 이용해서 파일 뷰를 생성해야 한다. 그리고, 파일 뷰들은 서로의 데이터를 보호하기 위해서 세마포어(`semaphore`), 뮤텍(`mutex`), 이벤트 등의 동기화 테크닉을 사용해야 한다.

이렇게 공유된 파일 매핑 객체는 모든 프로세스 들이 `CloseHandle`, `UnMapViewOfFile` 을 호출해서 핸들을 닫을 때까지 파괴되지 않는다.

## ● 일치성 (Coherence)

일치성은 하나의 파일 뷰에서 접근이 가능한 데이터는 디스크에 있는 파일의 내용과 동일한 복사본이라는 것을 보장하는 것이다. `CreateFileMapping` 함수는 원격지 파일 (`remote file`)에도 접근할 수 있는데, 이 경우에는 이러한 일치성이 보장되지 않는다. 예를 들어, 두 개의 컴퓨터가 하나의 파일을 쓰기 가능으로 설정하고 파일 매핑 객체를 생성했다고 하자.

이 때 두 컴퓨터에서 같은 페이지의 내용을 변경한 경우 각각의 컴퓨터는 자신이 페이지에 수행한 작업만을 반영해서 볼 수 있다. 데이터가 실제로 디스크에 업데이트될 때에는 이들이 서로 합쳐지지 않는다.

## ● 파일 매핑의 이용

처음에 파일 매핑 객체를 생성할 때에는 다음과 같이 한다.

```
hMapFile = CreateFileMapping(hFile,    //현재 파일의 핸들,
    NULL,                             //디폴트 보안값
    PAGE_READWRITE,                   //읽고, 쓰기가 모두 가능하게 설정
    0,                                //최대 객체의 크기
    0,                                //hFile 의 크기
    'MyFileMappingObject');           //매핑 객체의 이름
if (hMapFile = NULL) then ErrorHandler('파일 매핑 객체를 생성할 수 없다네 ...');
```

다음의 코드는 파일 매핑 객체의 핸들을 이용해서 파일의 뷰를 생성하는 부분이다.

```
lpMapAddress = MapViewOfFile(hMapFile,    //매핑 객체의 핸들
    FILE_MAP_ALL_ACCESS,                  //읽고, 쓰기가 모두 가능하게
    0,                                    //최대 객체의 크기
    0,                                    //hFile 의 크기
    0);                                   //전체 파일을 매핑한다.
if (lpMapAddress = NULL) then ErrorHandler('파일의 뷰를 매핑 못한다네 ...');
```

MapViewOfFile 함수는 파일 뷰에 대한 포인터를 반환한다. 프로세스는 메모리에 접근할 때 이 포인터를 사용한다. 다음의 코드에 의해 두번째 프로세스는 OpenFileMapping 함수를 이용해서 같은 메모리를 공유하게 된다

```
hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS,    //읽고-쓰기 권한
    FALSE,                                         //상속받지 않음
    'MyFileMappingObject');                       //매핑 객체의 이름
if (hMapFile = NULL) then ErrorHandler('파일 매핑 객체를 열수 없다네 ...');
```

```
lpMapAddress = MapViewOfFile(hMapFile,    //매핑 객체의 핸들
    FILE_MAP_ALL_ACCESS,                  //읽고-쓰기 접근 권한
```

```

0,                                //객체의 최대 크기
0,                                //hFile 의 크기
0);                                //전체 파일을 매핑
if (lpMapAddress = NULL) then ErrorHandler('파일의 뷰를 매핑할 수 없다네 ...');

```

다음의 코드는 UnmapViewOfFile 함수를 이용해서 프로세스 주소 공간에서 파일 뷰를 파괴하는 부분이다. 만약에 파일 뷰가 매핑된 후에 바뀐 내용이 있으면, UnmapViewOfFile 함수에 의해 변화된 부분을 디스크 파일에 복사한다.

```

if not UnmapViewOfFile(lpMapAddress) then ErrorHandler('파일 뷰를 해제 못함 ... ');

```

프로세스가 파일 매핑을 모두 사용하고, 파일 뷰의 매핑을 모두 해제 했으면, 파일 매핑 객체를 다음과 같이 닫는다.

```

CloseHandle(hMapFile);

```

FlushViewOfFile 함수는 파일 뷰에서 지정된 바이트 수만큼 물리적 파일에 기록한다.

```

if not FlushViewOfFile(lpMapAddress, dwBytesToFlush) then
    ErrorHandler('메모리를 디스크에 기록할 수 없다네 ... ');

```

## 메모리 맵 파일(memory mapped file)을 이용한 데이터 공유 기법

앞에서 설명한 파일 매핑을 이해한다면, 다음의 루틴 들도 쉽게 이해할 수 있을 것으로 믿는다. 다음에 소개하는 OpenMap 과 CloseMap 은 실제 어플리케이션을 사용할 때 그대로 가져가서 사용할 수도 있는 부분이므로 유용하게 사용하기 바란다.

참고로, 파일 매핑 객체에 대한 핸들과 파일 뷰에 대한 포인터에 대한 전역 변수를 선언해서 써야 한다. 그러므로, 먼저 다음과 같이 변수 선언을 전역으로 한다.

```

var
    HMapping: THandle;
    PMapData: Pointer;

procedure OpenMap(MapFileSize: Integer, MappingObject: String);
begin
    HMapping := CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0, MAPFILESIZE,

```



```

    PChar(MappingObject));
if (hMapping = 0) then
begin
    ShowMessage('메모리 맵을 생성할 수 없습니다.');
```

Application.Terminate;

Exit;

end;

PMapData := MapViewOfFile(HMapping, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

if PMapData = nil then

begin

CloseHandle(HMapping);

ShowMessage('파일 뷰를 매핑할 수 없습니다.');

Application.Terminate;

Exit;

end;

end;

procedure CloseMap;

begin

if PMapData <> nil then UnMapViewOfFile(PMapData);

if HMapping <> 0 then CloseHandle(HMapping);

end;

하나 이상의 어플리케이션이나 DLL 에서 같은 물리적 메모리 블록의 포인터를 얻기 위해서는 이런 방법을 써야 한다. 여기서 PMapData 변수는 MapFileSize 파라미터에 넘겨진 바이트 크기의 데이터에 대한 버퍼를 가리키는 포인터이다. 한가지 생길 수 있는 문제점은 메모리에 접근하는 데이터를 동기화해야 한다는 것이다. 이를 위해서 뮷텍(mutex)을 사용할 수 있는데, 뮷텍을 이용해서 동기화를 할 수 있도록 다음의 LockMap, UnLockMap 프로시저를 적절하게 활용하면 된다. 사용 방법은 메모리 맵 파일에 데이터를 읽거나 쓰기 전에는 LockMap 을 호출한다. 그리고, 자료의 업데이트가 끝나면 바로 UnLockMap 을 호출하면 된다.

그리고, 각 프로젝트에 뮷텍의 핸들을 저장한 전역 변수를 다음과 같이 선언해 주어야 한다.

```

var
    HMapMutex: THandle;
```

```

const
    REQUEST_TIMEOUT = 1000;

function LockMap(MutexName: String): Boolean;
begin
    Result := True;
    HMapMutex := CreateMutex(nil, False, PChar(MutexName));
    if HMapMutex = 0 then
    begin
        ShowMessage('뮷텍을 생성할 수 없습니다.');
```

Result := False;

```

    end
    else
    begin
        if WaitForSingleObject(HMapMutex, REQUEST_TIMEOUT) = WAIT_FAILED then
        begin
            // timeout
            ShowMessage('메모리 맵 파일을 잠글 수 없습니다.');
```

Result := False;

```

        end;
    end;
end;

procedure UnlockMap;
begin
    ReleaseMutex(HMapMutex);
    CloseHandle(HMapMutex);
end;

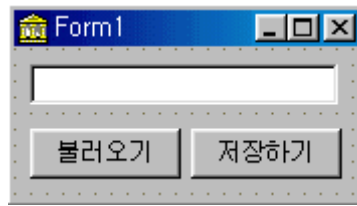
```

여기에 대한 더 자세한 설명은 스레드와 동기화에 대해서 설명한 장을 참고하기 바란다.

## 메모리 맵 파일의 활용

메모리 맵 파일은 대용량의 데이터를 순식간에 처리하는 빠른 처리 속도가 가장 큰 장점이 다. 그렇기 때문에 데이터의 임시 저장 장소로 사용하거나 대용량의 데이터(그래픽 등)를 실시간에 주고 받아야 하는 경우에 활용도가 높다. 그러면, 간단한 예제를 통해서 문자열을 임시 저장할 공간으로 메모리 맵 파일을 활용하는 방법을 익혀보도록 하자. 먼저 폼에

버튼 2개와 에디트 박스를 하나 없어서 다음과 같이 디자인하도록 하자.



그리고, 파일 매핑 핸들로 사용할 변수와 매핑할 데이터로 사용할 변수, 에러 코드 변수를 public 섹션에 선언하고 문자열의 최대 길이를 상수로 다음과 같이 선언한다.

```
public
    hFileMap: THandle;
    MapError: Integer;
    SharedPChar: PChar
end;
```

```
const
    CharLen = 100;
```

이제 OnCreate 이벤트 핸들러에서 파일 매핑 객체를 생성하고, 뷰를 매핑하도록 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    hFileMap := 0;
    SharedPChar := nil;
    try
        hFileMap
            := CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0, CharLen + 1, 'SampleMMF');
        if hFileMap = 0 then
            raise Exception.Create('메모리 맵 파일 생성에 실패했습니다 !')
        else
            MapError := GetLastError();
        SharedPChar
            := PChar(MapViewOfFile(hFileMap, FILE_MAP_READ + FILE_MAP_WRITE, 0, 0, 0));
        if MapError <> ERROR_ALREADY_EXISTS then
            StrPLCopy(SharedPChar, '샘플입니다 !', CharLen);
```

```

    Edit1.Text := SharedPChar;
except
    on E: Exception do begin
        if SharedPChar <> nil then
            UnmapViewOfFile(SharedPChar);
        if hFileMap <> 0 then
            CloseHandle(hFileMap);
        end;
    end;
end;

```

이미 앞에서 CreateFileMapping 과 MapViewOfFile API 함수의 사용법에 대해서는 자세히 설명하였으므로, 이를 참고하기 바란다. 이 코드가 성공적으로 수행되면 SharedPChar 는 공유 데이터를 담는 변수로 사용된다. 그리고, 초기 값으로 ‘샘플입니다 !’ 라는 문자열을 설정한다.

반대로 OnDestroy 이벤트 핸들러에서는 생성된 파일 매핑을 해제해야 한다.

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    UnmapViewOfFile(SharedPChar);
    CloseHandle(hFileMap);
end;

```

일단 이렇게 설정되면, 사용하는 방법은 매우 간단하다. SharedPChar 변수가 공유 데이터 라고 간주하고, 이를 직접 Edit1.Text 프로퍼티를 설정하거나 읽어오도록 다음과 같이 코딩 하면 된다.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.Text := SharedPChar;
end;

```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    StrPLCopy(SharedPChar, Edit1.Text, CharLen);
end;

```

이것으로 간단한 예제가 완성되었다. 컴파일하고 여러 개의 인스턴스를 실행한 뒤에 하나의 인스턴스의 에디트 박스의 내용을 변경하고 ‘저장하기’ 버튼을 클릭한다. 그리고, ‘불러오기’ 버튼을 클릭하면 저장된 내용으로 에디트 박스의 내용이 변경될 것이다.

## 정 리 (Summary)

이번 장에서는 델파이의 메모리를 관리하는 방법과 메모리를 공유하기 위해 메모리 맵 파일을 사용하는 방법에 대해서 알아보았다. 이런 내용은 실제 어플리케이션을 작성할 때 직접적인 영향을 미치지지는 않지만, 전반적인 수행성능을 향상시키거나 기능을 확장하는데 도움이 되는 것들이므로 잘 익혀놓으면 잘 만들어진 어플리케이션을 만드는데 커다란 도움이 될 것이다.

# 하드웨어 제어 기법의 정복

## (Mastering Hardware Control Techniques)

MS-DOS 시절에는 어플리케이션으로 기계를 직접 제어할 수 있었다. 비록 귀찮은 작업들이 많았지만, 프로그래머가 하드웨어에 직접 접근할 수 있었기에 빠른 속도를 낼 수 있었고, 동시에 많은 일들을 마음대로 할 수 있었다.

윈도우 3.1 로 넘어오면서 이러한 자유로움에 제한을 받기 시작했다. 더 이상 하드웨어에 직접 접근하는 것을 자유롭게 허용받지 못했다. 이는 어찌보면 당연하다고 말할 수 있다. 윈도우에서는 사용자가 수많은 어플리케이션을 사용할 수 있으며, 또한 같은 하드웨어를 동시에 접근하지 말라는 보장도 없다. 그리고, 윈도우는 멀티 태스킹 환경이기 때문에 다른 어플리케이션의 동작을 방해해서는 안되고, 이를 위해서 직접적인 하드웨어의 제어는 위험한 발상일 수도 있는 것이다. 그러나, 하드웨어를 직접 제어할 때의 장점도 무시할 수 없는 것이다.

### 하드웨어 포트 직접 제어의 문제점

이제는 Win32 (윈도우 NT, 윈도우 95)의 시대이다. 이들은 진정한 운영체제이며, 기본적으로 선점형(pre-emptive) 멀티 태스킹 환경이다. 그렇기 때문에, 각각의 쓰레드(실행단위)는 프로세서에 일정량의 시간을 할당 받는다. 시간이 되거나, 보다 높은 순위의 쓰레드가 있을 경우 시스템은 다른 쓰레드에 컨트롤을 넘긴다. 이러한 스위칭(switching)은 어떠한 어셈블리 코드 사이에서도 이루어지며, 하나의 쓰레드가 동작하는 도중에 이 쓰레드가 완료되기 전에 컨트롤이 다른 쓰레드로 넘어갔다가 얼마나 오랜 시간이 지난 후에 다음 코드가 진행될 수 있을지는 알 수 없다. 이런 점이 직접적인 하드웨어 제어를 할 때의 가장 큰 문제점이다.

가장 일반적인 형태의 I/O 포트의 입력 부분은 다음과 같은 몇 개의 어셈블리 코드로 이루어진다.

```
mov    dx, AddressPort
mov     al, Address
out     dx, al
jmp     Wait
Wait:
mov     dx, DataPort
in      al, dx
```

쓰레드가 변경될 때 모든 레지스터의 상태는 보존되더라도, I/O 포트의 상태는 보존되지 않는다. 그렇기 때문에, in, out 코드 사이에 자신의 I/O 포트 값을 보존할 수 있는 방법을 강구해야 한다.

## 표준적인 방법

뮤텍(mutex)을 이용하는 방법이다. 문제는 다른 어플리케이션에서도 뮤텍을 무시하지 않아야 한다는 것이다. 그 밖에도 몇 가지 문제점이 있는데, 예를 들어 App1, App2 라는 2 개의 어플리케이션이 있다고 하자. 이들이 모두 포트를 제어하려고 하는데, 이를 만든 제작자의 관점이 달라서 App1 은 일단 AddressPortMutex 을 먼저 얻으려고 하고, App2 는 DataPortMutex 을 먼저 얻으려고 한다고 하자. 그래서, 이들이 각각 뮤텍을 얻고 나서 시스템이 App1 에서 App2 로 쓰레드를 넘겼다고 하자. 그러면, App2 는 address port 를 얻을 수 없는 deadlock 상황이 되버린다. 또한, App1 은 data port 를 얻을 수 없게 되버린다. 이런 문제를 해결하기 위한 가장 좋은 방법은 port/memory area 를 제어하는 디바이스 드라이버를 만드는 것이다. 이때에는 API 를 사용하는데, 가장 전형적인 함수는 다음과 같다.

```
GetIOPortData(AddressPort, DataPort: word): Byte;
```

GetIOPortData 함수는 일단 양쪽 포트에 뮤텍을 걸고, 포트에 접근한다. 그리고, 호출자 (caller)에게 결과 값을 돌려주기 전에 뮤텍을 해제한다. 만약 다른 쓰레드가 이 함수를 동시에 사용하면 한 쪽에서 먼저 동작하고, 다른 쪽에서는 대기하게 되므로 deadlock 은 발생하지 않는다.

문제는 디바이스 드라이버를 제작하는 것이 별로 쉽지 않다는 것이다. 보통 어셈블러나 C 로 제작하게 되는데, 디버깅하기도 어렵거니와 윈도우 NT 의 디바이스 드라이버(VDD, virtual device driver)와 윈도우 95 의 디바이스 드라이버(VxD) 사이의 호환성도 보장할 수 없다. 그러므로, 각각에 대해 다른 디바이스 드라이버를 제작해 주어야 한다.

디바이스 드라이버를 제작하는 방법에 대한 것은 이 책이 다룰 범위를 넘기 때문에, 여기서는 더 이상 다루지 않도록 하겠다.

## 편법적인 방법

보통 하드웨어를 직접 다루려고 하는 것은 특정 하드웨어에 접근하는 어플리케이션을 만들 때이기 때문에, 디바이스 드라이버를 제작한다는 것은 사실 좀 비현실적이고, 낭비라고 할 수 있다.

다행히(?) 윈도우 95 는 윈도우 3.1 과 호환되게 만들어져 있어서 직접적인 I/O 제어가 일부

나마 가능하다 (이는 많은 수의 윈도우 3.1 프로그램이 직접 I/O 포트를 제어하기 때문이다.). 다음과 같은 함수를 이용하면 일단 포트에 접근하여 데이터를 주고 받을 수 있다.

```
function GetPort(p: Word): Byte; stdcall;
```

```
begin
```

```
    asm
```

```
        push    edx
```

```
        push    eax
```

```
        mov     dx, p
```

```
        in      al, dx
```

```
        mov     @result, al
```

```
        pop     eax
```

```
        pop     edx
```

```
    end;
```

```
end;
```

```
procedure SetPort(p: Word; b: Byte); stdcall;
```

```
begin
```

```
    asm
```

```
        push    edx
```

```
        push    eax
```

```
        mov     dx, p
```

```
        mov     al, b
```

```
        out     dx, al
```

```
        pop     eax
```

```
        pop     edx
```

```
    end;
```

```
end;
```

이와 비슷한 코드 들을 이용하게 되는데, 같은 기능을 하는 다른 코드를 살펴보자.

```
function PortIn(IOAddr: Word): Byte;
```

```
begin
```

```
    asm
```

```
        mov     dx, IOAddr
```

```
        in      al, dx
```



```

        mov  result, al
    end;
end;

```

```

procedure PortOut(IOAddr: Word; Data: Byte);
begin
    asm
        mov  dx, IOAddr
        mov  al, Data
        out  dx, al
    end;
end;

```

위의 두 코드를 비교해 보면 알겠지만, 핵심은 dx, al 레지스터에 각각 포트의 주소와 데이터의 값을 저장하고, 이를 이용해서 out, in 을 하면 된다.

이제 조금은 효율적이면서 여러모로 쓰일 수 있는 간단한 유닛을 만들어 보자

```

unit U_Port;

```

```

interface

```

```

function PortReadByte(Addr: Word): Byte;
function PortReadWord(Addr: Word): Word;
function PortReadWordLS(Addr: Word): Word;
procedure PortWriteByte(Addr: Word; Value: Byte);
procedure PortWriteWord(Addr: Word; Value: Word);
procedure PortWriteWordLS(Addr: Word; Value: Word);

```

```

implementation

```

```

function PortReadByte(Addr: Word): Byte; assembler; register;
asm
    mov  dx, ax
    in   al, dx
end;

```

```
function PortReadWord(Addr: Word): Word; assembler; register;
```

```
asm
```

```
    mov  dx, ax
```

```
    in   ax, dx
```

```
end;
```

```
function PortReadWordLS(Addr: Word): Word; assembler; register;
```

```
const
```

```
    Delay = 150;           //CPU 속도와 카드의 속도에 따라 조절
```

```
asm
```

```
    mov  dx, ax
```

```
    in   al, dx             //LSB 포트 값을 읽는다.
```

```
    mov  ecx, Delay
```

```
@1:
```

```
    loop @1                //Delay ...
```

```
    xchg ah, al
```

```
    inc  dx                //포트 + 1
```

```
    in   al, dx            //MSB 포트 값을 읽는다.
```

```
    xchg ah, al            //바이트 순서를 바로 잡음
```

```
end;
```

```
procedure PortWriteByte(Addr: Word; Value: Byte): assembler; register;
```

```
asm
```

```
    xchg ax, dx
```

```
    out  dx, al
```

```
end;
```

```
procedure PortWriteWord(Addr: Word; Value: Word): assembler; register;
```

```
asm
```

```
    xchg ax, dx
```

```
    out  dx, ax
```

```
end;
```

```
procedure PortWriteWordLS(Addr: word; Value: Word): assembler; register;
```

```
const
```

```
    Delay = 150           //CPU, 카드 속도에 따라서...
```

```

asm
    xchg  ax, dx
    out   dx, al           //LSB 포트에 쓰기
    mov   ecx, Delay
@1:
    loop  @1               //Delay...
    xchg  ah, al
    inc   dx               //포트 + 1
    out   dx, al           //MSB 포트에 쓰기
end:

end.

```

## 윈도우 NT 의 경우

앞의 코드는 윈도우 NT 하에서는 동작하지 않는다. 기본적으로 윈도우 NT 는 플랫폼에 구애받지 않기 때문에, 이런 방식의 I/O 포트 제어가 프로세서에 따라 다르게 되어야 하므로 앞의 방식은 통하지 않는다.

그렇다고 전혀 방법이 없는 것은 아니다. NT 에서 윈도우 95/98 의 환경과 다른 점은 사용자 모드에 있는 코드가 직접 하드웨어에 접근할 수 없도록 막는다는 점이다. 즉, 모든 하드웨어 리소스는 운영체제에 의해 관리되는 것이다.

그렇지만, 이러한 기본적인 원칙에도 통하는 방법은 있기 마련이다. NT 커널은 I/O 포트 주소의 맵을 관리하고 있는데, 각각의 프로세스가 접근할 수 있도록 허용한다. 그러므로, 현재의 프로세스가 NT 에게 다른 IOPM(I/O Permission Map)에 접근한다고 알리고 포트에 접근할 수가 있는데, 이런 방법은 OS 의 측면에서 보면 그다지 바람직하지 못한 것이므로 절대로 남발해서는 안된다. 그렇지만, 특정 하드웨어에 접근해서 사용해야 하지만 NT 용 디바이스 드라이버를 새로 작성할 수 없는 특수한 경우에는 이런 방법을 쓸 수 밖에 없을 것이다.

여기서 가장 큰 문제는 앞서서도 언급했듯이 사용자 모드의 코드가 커널 함수를 이용하여 IOPM 을 변경할 수 없다는 점이다. 이를 해결하기 위해서 어플리케이션의 요구에 따라 IOPM 의 내용을 변경할 수 있는 NT 드라이버를 이용할 수 있다. 이 방법이 가능한 이유는 디바이스 드라이버는 IOPM 을 변경할 권한이 있기 때문이다. 즉, 사용자의 코드가 직접 포트에 접근할 수 없다면, 포트에 마음대로 접근할 수 있는 디바이스 드라이버를 거쳐서 이를 제어하는 것이다. 여기에 가장 흔히 사용되었던 디바이스 드라이버가 Dale Roberts 가 작성한 giveio.sys 디바이스 드라이버이다. Giveio.sys 디바이스 드라이버는 문서화되지 않은 커널 함수를 이용하여 모든 I/O 포트에 접근할 수 있도록 하였다. 여기서는 이 디바

이스 드라이버를 사용하지 않고, Graham Wideman 이 선택된 포트에만 접근할 수 있고 몇 가지 진단 함수를 추가하여 새롭게 작성한 gwiopm.sys 디바이스 드라이버를 사용한다. 이 디바이스 드라이버의 소스는 이번 장과 관련한 CD-ROM 디렉토리의 Gwiopm 서브 디렉토리에 C 로 작성된 소스가 담긴 Src 디렉토리에 포함되어 있으므로 관심이 있는 독자는 분석해보기 바란다. 그와 함께 Wideman 이 직접 예제로 제공한 PortTest 디렉토리도 같이 제공된다. gwiopm.sys 드라이버에 대한 텔파이 인터페이스 유닛으로 gwiopm.pas 유닛 역시 Wideman 이 제공하는데, 이를 이용하여 프로그래밍을 하는 방법에 대해서 알아보도록 하자.

디바이스 드라이버 파일이 있다고 해서 이를 바로 사용할 수는 없다. 이를 제대로 사용하려면 드라이버를 설치해야 한다. NT 의 경우에는 레지스트리를 조작해야되는 문제가 있는데, 앞서 46 장에서도 언급했듯이 수동으로 레지스트리를 직접 조작하면 해결할 수 있는 것들이 많지만 의외로 이런 일을 할 수 있는 사람을 극소수이다. 그러므로, 이를 프로그램에서 해결할 수 있도록 해결책을 제시하는 것은 필수이다. Wideman 은 gwiopm.pas 유닛에 이런 설치를 담당하는 함수를 같이 제공하여 쉽게 드라이버를 설치해서 사용할 수 있도록 해주고 있다.

#### ● gwiopm 유닛에 대한 고찰

gwiopm.pas 유닛에는 드라이버를 설치하고 IOPM 을 준비하는 등의 역할을 하는 GWIOPM\_Driver 객체가 포함되어 있다. 하나의 IOPM 은 8KB 의 크기로 구성되어 있는데, 하나의 비트가 x86 의 64K I/O 포트 주소 공간 하나를 제어한다.

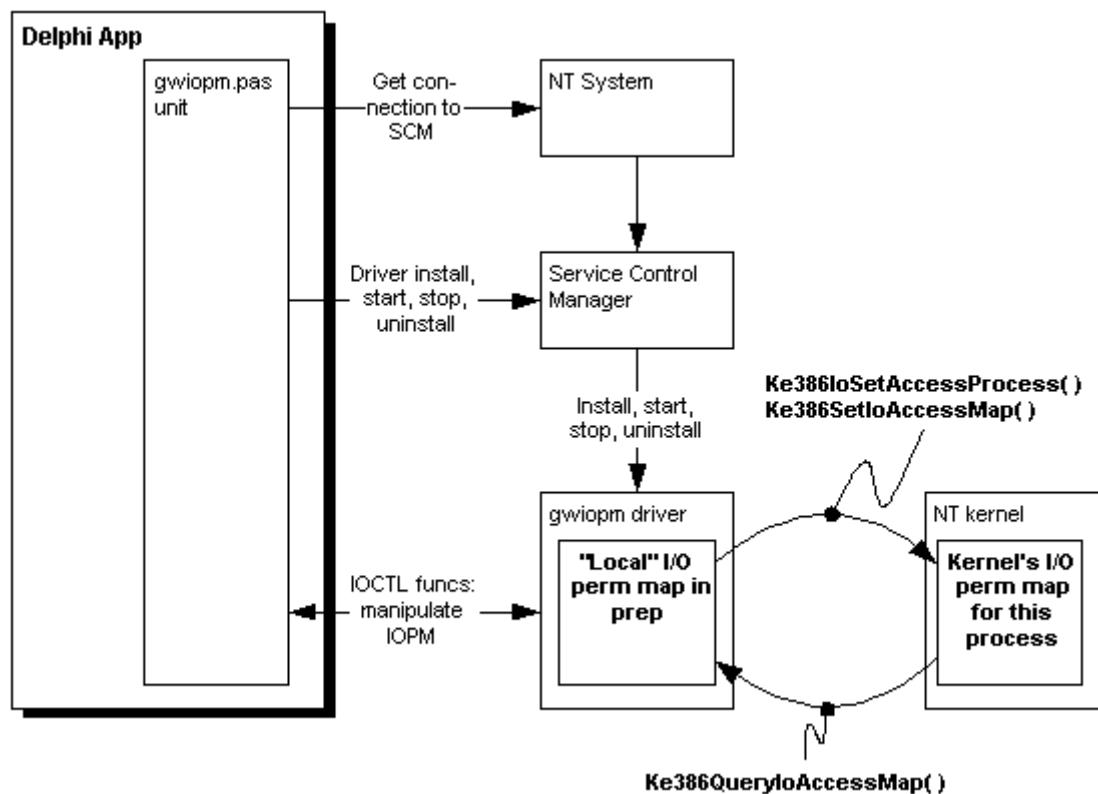
gwiopm.pas 유닛의 역할은 윈도우 NT 의 SCM(Service Control Manager)에 접근하여 접속을 한다. 그리고 드라이버를 설치, 시작, 중지, 제거할 수 있는 방법을 제공하며, IOCTL 함수를 이용하여 IOPM 을 관리할 수 있도록 해준다.

실제로 커널의 IOPM 과 드라이버의 IOPM 을 상호작용할 수 있도록 매핑하는 역할을 하는 것이 gwiopm.sys 디바이스 드라이버의 구현 방법이다. 이를 위해서는 커널의 함수로 Ke386IoSetAccessProcess, Ke386SetIoAccessMap, Ke386QueryIoAccessMap 함수를 사용하는데, 이들의 역할은 다음과 같다고 추정된다 (추정된다는 표현을 쓴 이유는 이들 함수가 모두 문서화되지 않은 함수이기 때문이다). 사용법을 익히고자 하는 독자는 CD-ROM 에 제공되는 C 소스 코드를 참고하기 바란다.

커널 함수 (void 형)	설 명
Ke386IoSetAccessProcess(PEPROCESS, int);	커널에게 현재 프로세스를 제공하여, 이 프로세스가 특정 IOPM 을 사용하기 원한다는 것을 알리는 역할을 한다. 이 함수의 결과로 커널은 프로세스에 대한 주소 공간을 확보한다. int 파라미터의 값이 1 이면 특

	정 맵이 사용 가능한 것이다.
Ke386SetIoAccessMap(int, IOPM *);	드라이버가 8KB 크기의 새로운 맵을 커널에게 전달하는 함수이다. int 파라미터의 값이 1 이면 현재의 맵을 복사하는 것이며, 0 이면 커널의 맵은 \$FF 로 채운다. 참고로 맵의 1 비트가 특정 포트를 담당하므로 int=0 을 파라미터로 사용하면 커널의 맵이 모두 clear 된다.
Ke386QueryIoAccessMap(int, IOPM *);	커널의 맵을 다시 드라이버의 버퍼로 복사하는 역할을 한다. int 파라미터는 1 로 설정한다.

이를 도식화한 그림이 다음과 같다.



그러면 gwiopm.pas 유닛에 의해 제공되는 함수에 대해서 알아보도록 하자. 이들 함수는 모두 GWIOPM\_Driver 객체의 메소드로서 제공된다.

#### 1. SCM 에 접속, 해제를 담당하는 함수

```
function OpenSCM: DWORD;
```

function CloseSCM: DWORD;

이 함수들은 윈도우 NT 의 SCM(Service Control manager)을 열고, 닫는 역할을 한다. OpenSCM 에 의해 SCM 의 핸들을 얻을 수 있으며, 이 핸들은 GWIOPM\_Driver 객체에 의해 사용된다.

## 2. 드라이버 설치에 관련한 함수

function Install(newdriverpath: string): DWORD;

function Start: DWORD;

function Stop: DWORD;

function Remove: DWORD;

이 함수들은 각각 드라이버를 설치, 시작, 중지, 제거하는 역할을 담당한다. Install 함수는 드라이버의 패스를 지정하거나 널 문자열(‘’, 디폴트 값)을 지정한다. 널 문자열이 지정되면 드라이버가 어플리케이션과 같은 디렉토리에 있는 것으로 간주한다.

## 3. 디바이스 함수

function DeviceOpen: DWORD;

function DeviceClose: DWORD;

디바이스를 사용하기 위해서는 먼저 디바이스를 열면 GWIOPM\_Driver 의 다른 드라이버 함수를 사용할 수 있는 디바이스 핸들을 가져올 수 있다. 드라이버 함수를 사용한 뒤에 드라이버를 제거하려면 먼저 디바이스를 닫아야 한다.

## 4. 테스트 함수

function IOCTL\_IOPMD\_READ\_TEST(var RetVal: DWORD): DWORD;

function IOCTL\_IOPMD\_READ\_VERSION(var RetVal: DWORD): DWORD;

드라이버가 잘 동작하는지 알아보기 위한 진단 함수이다. READ\_VERSION 의 RetVal 파라미터 값은 보통 100 을 넘으며, READ\_TEST 의 RetVal 값은 \$123 이다.

## 5. 드라이버의 로컬 IOPM(LIOPM)의 조작 함수

```
function IOCTL_IOPMD_CLEAR_LIOPM: DWORD;
function IOCTL_IOPMD_SET_LIOPM(Addr: Word; B: byte): DWORD;
```

드라이버의 맵 배열을 Clear 하고, 특정 값을 주소에 설정하는 함수이다.

```
function IOCTL_IOPMD_GET_LIOPMB(Addr: Word; var B: byte): DWORD;
function IOCTL_IOPMD_GET_LIOPMA(var A: TIOPM): DWORD;
```

이 함수 들은 특정 바이트 값을 가져오거나, 드라이버에서 맵 전체를 가져오도록 하는 함수이다.

```
function LIOPM_Set_Ports(BeginPort: word; EndPort: word; Enable: Boolean): DWORD;
```

특정 포트에 대한 비트를 Enable/disable 시키는 함수이다.

## 6. 커널 IOPM(KIOPM)과의 상호 작용을 담당하는 함수

```
function IOCTL_IOPMD_ACTIVATE_KIOPM: DWORD;           //드라이버 맵을 커널에 전달
function IOCTL_IOPMD_DEACTIVATE_KIOPM: DWORD;         //커널이 현재의 맵을 무시하도록 지정
function IOCTL_IOPMD_QUERY_KIOPM: DWORD;              //커널 맵을 사용할 수 있게 한다.
```

### ● 예제 어플리케이션

예제로 제공된 PortTest 어플리케이션을 잘 분석해보면 쉽게 사용방법을 익힐 수 있을 것이다. 비교적 자세하게 작성된 좋은 예제이므로 이를 참고하기 바란다.

## 하드웨어 제어와 다이렉트 X

지금까지 포트 제어를 이용한 하드웨어를 접근하는 방법에 대해서 알아보았다. 여기까지 소개한 방법은 비교적 과거의 도스 때와 마찬가지로 low-level 에서 접근하여 하드웨어를 제어하는 방법 들이다.

윈도우 프로그래밍 환경에서 아직도 이런 방식으로 하드웨어에 직접 접근하고자 하는 필요성이 대두되는 것은 아마도 속도의 문제와 윈도우라는 운영체제의 특성을 깰 수 없다는 점이 가장 문제가 될 것이다. 예를 들어, 조이스틱을 제어하려면 MMSystem.pas 유닛을 uses 절에 추가하고 적당한 조이스틱 제어 API 함수를 호출하여 사용할 수 있지만 윈도우라는 제한에 걸리게 되며, 개발자가 원하는 정도로 폭넓은 지원을 API 가 제공하지 않을 경

우 난관에 빠질 수 밖에 없다. 보통 하드웨어를 직접 제어할 때에는 도스에서와 마찬가지로 현재의 프로그램이 모든 권한을 쥐고서 사용할 수 있기를 바라지만, 윈도우 API 는 여러 윈도우의 공유 관계를 우선시 하기 때문에 이런 프로그래머의 욕구와는 배치되는 면이 많다. 그렇지만, 이런 방식으로 직접 하드웨어에 접근하는 것은 그다지 바람직하지 못한 방법이다. 이런 문제를 해결하기 위해서 마이크로소프트에서 해결책으로 제시한 것이 바로 다이렉트 X 이다. 물론 모든 종류의 하드웨어에 다 적용되는 것은 아니지만, 가장 흔히 사용되는 하드웨어를 바탕으로 하여 비교적 성공적으로 정착되어 나가고 있다.

텔파이에서도 게임 프로그래밍을 하는 많은 그룹들이 생겨나고 있으며, 이들을 중심으로 새로운 다이렉트 X 버전이 발표될 때마다 파스칼 버전의 유닛을 번역하여 제공하는 여러 사람들도 생기고 있으며, 다이렉트 X 를 클래스로 잘 포장하여 쉽게 사용할 수 있도록 제공하는 프리웨어 컴포넌트도 많이 등장하고 있다.

그중에서도 DGC(Delphi Game Creator)와 텔파이 X 가 가장 유명하다. CD-ROM 에 가장 최신 버전의 텔파이 X 컴포넌트와 다이렉트 X 유닛을 같이 제공하므로 이를 이용하여 다이렉트 X 의 기능을 즐겨보기 바란다. 좋은 예제도 많이 실려있으므로 이를 이해하는 것은 그다지 어렵지 않을 것이다.

다이렉트 X 에 대한 부분 역시 따로 책을 한 권 구성하여 설명해도 모자랄 정도로 다루고 있는 범위가 광범위하다. 그렇기 때문에, 여기서는 다이렉트 X 에 대한 간단한 길잡이 정도로 다루고 넘어가고자 한다. 기회가 닿는다면 웹이나 하이텔 등의 통신망을 통해 다이렉트 X 강좌를 게시할 계획을 가지고 있으므로 많은 기대 바란다.

다이렉트 X 는 현재 6.0 버전이 출시될 준비를 하고 있으며, 5.0 버전이 가장 일반화되어 많이 사용되고 있다. 다이렉트 X 는 프로그래머가 윈도우 환경에서 하드웨어에 직접 접근할 수 있도록 해주는 API 의 세트이다. 다이렉트 X 의 장점은 윈도우 환경에서의 표준 장치에 접근하는 장점을 채용하여 각각의 하드웨어 별로 따로 프로그래밍이 필요하지 않도록 공통된 API 를 제공하면서도 여기에 최적화된 하드웨어를 제조업체에서 제작하도록 유도함으로써 그 수행 성능의 향상을 같이 유도했다는 점이다.

## ● 다이렉트 X 의 목적

다이렉트 X 가 처음 등장한 이유는 WinG 의 후속으로 윈도우 95 를 게임 플랫폼으로도 확장시키고자하는 의도에서 였다. 돌이켜 보면 게임에 PC 산업에 미친 영향은 막대한데, 그 중에서도 비디오와 사운드를 비롯한 각종 하드웨어의 고급화를 부추긴 결정적인 역할을 했다. 예를 들어, EGA 에서 VGA 이상급으로 비디오 카드가 고급화하고 사운드 블라스터를 위시한 각종 사운드 카드가 일반화한 결정적인 계기가 바로 게임이다. 아마도 IBM PC 의 고전 게임들을 기억하는 독자들은 페르시아 왕자라는 게임이 사운드 블라스터를 전세계적인 히트 상품으로 이끈 장본인이라는 것 정도는 잘 알고 있을 것이다.

그런데, 윈도우 95 의 초기에는 이러한 게임 들의 높은 시스템 요구사항을 따라가기에 윈도



우 95 는 너무나 느린 플랫폼이었다. 이렇게 된 가장 주된 원인이 바로 하드웨어에 대한 직접 접근이 봉쇄된 것이다. 어쩔 수 없이 많은 게임 개발자들은 도스 환경에서의 게임 개발을 계속할 수 밖에 없었고, 불과 1 년전만 해도 대부분의 게임이 도스 환경에서 돌아가도록 제작되었다. 이로 인해 게임을 즐기는 많은 윈도우 95 의 사용자들은 게임을 위해서 도스 환경을 버릴 수 없는 상태가 되었고, 마이크로소프트 역시 향후 운영체제 시장의 발전을 위해서 이러한 형태는 결코 자사에 이익이 되지 않는다는 것을 간파하였다.

다이렉트 X 는 이렇게 탄생했지만, 멀티미디어가 어플리케이션의 주류로 등장하면서 다이렉트 X 의 필요성은 더욱 커져만 간다. 그리고, IE 4.0 으로 대별되는 후기 윈도우 95 와 윈도우 98 에서는 다이렉트 X 가 기본적인 컴포넌트로 운영체제에 설치되면서 다이렉트 X 는 기본 운영체제 환경으로 자리를 잡았다.

현재의 IE 4.0 이상 환경에서 제공되는 다이렉트 X 컴포넌트는 DirectDraw, DirectSound, Direct3D, DirectShow, DirectAnimation 의 5 가지로 구성된다.

#### ● 다이렉트 X 의 기초

다이렉트 X 는 기본적으로 서로 다른 하드웨어 종류에 따라 다른 컴포넌트로 구성되었다. 하드웨어와의 인터페이스는 크게 2 가지인데, 소프트웨어와 하드웨어 사이에 위치하여 개발자가 하드웨어의 종류에 관계없이 사용할 수 있는 중간층 역할을 하는 HAL(Hardware Abstraction Layer)와 하드웨어적으로 지원되지 않는 기능을 소프트웨어적으로 보완하여 구현하는 HEL(Hardware Emulation Layer)로 나누어볼 수 있다. HEL 이 적용되는 가장 흔한 부분은 3D 를 지원하는 부분이다.

또한, 다이렉트 X 는 기본적으로 액티브 X 와 같은 컴포넌트 기술에 기반하고 있기 때문에 쉽게 접근하여 사용할 수 있다.

#### ● DirectDraw

DirectDraw 는 그래픽 카드에 직접 접근하는 방법을 개발자들에게 열어 줌으로써 게임 프로그래밍과 같은 다이렉트 X 를 사용하는 어플리케이션의 활성화를 가져온 가장 대표적인 컴포넌트이다. DirectDraw 를 이용해 그래픽의 속도가 빨라졌으며, 이로 인해 윈도우 95 를 본격적인 게임 플랫폼으로 사용할 수 있는 길이 열렸다.

DirectDraw 는 HAL 과 HEL 을 모두 이용하여 실제로 하드웨어를 직접 접근할 때 HAL 을 사용하고, 비록 지원되지 않는 기능을 사용하더라도 HEL 을 이용하여 소프트웨어 적으로 이를 보완하기 때문에 개발자는 표준적인 개발환경을 이용할 수 있게 되었다.

DirectDraw 의 기능을 단적으로 설명하자면 비디오 메모리 관리자라고 할 수 있다. DirectDraw 는 비디오 메모리를 DirectDraw, DirectDrawSurface, DirectDrawPalette, DirectDrawClipper 라는 4 개의 객체를 통해 인터페이스한다.

DirectDraw 객체는 하드웨어 장치를 대표하는 기본적인 객체로 어플리케이션을 전체화면이나 윈도우 모드로 실행될 수 있도록 설정이 가능하며, 경우에 따라서는 다른 프로그램이 접근하지 못하도록 하드웨어 장치를 배타적으로 사용할 수 있도록 지정할 수도 있다. 또한, 이 객체를 이용하여 비디오 해상도의 조절이 가능하다.

DirectDrawSurface 객체는 개발자가 메모리에 저장된 그래픽에 접근할 때 사용된다. Surface 라는 것은 현재 화면에 나타나는 주 surface(primary surface)와 다음에 화면에 나타날 백 버퍼(back buffer), 큐에 들어가려고 대기하는 off-screen surface 로 나눌 수 있다. 화면이 움직이는 것은 off-screen-surface 들이 백 버퍼로 들어가고, 백 버퍼의 surface 가 주 surface 로 전환되면서 이루어지는 것이다.

DirectDrawPalette 객체는 각 surface 와 surface 사이에 독자적인 256-색상의 팔레트를 적용하거나, 공유 팔레트를 사용하도록 지정할 수 있으며, DirectDrawClipper 객체는 GDI 를 건너뛰어 그래픽을 처리함으로써 그래픽 처리 속도의 향상을 가져올 수 있다.

#### ● DirectSound, DirectInput

DirectSound 는 사운드와 음악 하드웨어에 대한 인터페이스를 담당한다. 보통 .wav 파일이 표준으로 이용되는데 이때 1 차, 2 차, 정적, 스트림 버퍼를 이용하게 된다. 1 차 버퍼(primary buffer)는 현재 컴퓨터에 의해 사용되고 있는 것으로 2 차 버퍼(secondary buffer)에서 사운드 파일을 가져온다. 작은 파일 들은 정적 버퍼(static buffer)에 위치했다가 빠르게 접근이 가능하며, 큰 파일들은 스트림 버퍼(stream buffer)를 이용하여 그때 그때 부분적으로 접근하여 사용하게 된다.

DirectInput 은 조이스틱을 비롯한 각종 게임용 입력 장치를 다룰 수 있는 인터페이스를 제공한다. DirectInput 을 이용하여 최고 16 개의 조이스틱과 32 버튼, 6 가지 축을 지원할 수 있다. 조이스틱 이외에도 그래픽 태블릿과 같은 장치를 입력장치로 사용할 수 있다.

원래 이번 장을 계획하면서 MMSystem.pas 유닛에 정의된 API 를 이용하여 조이스틱을 사용하는 방법에 대해서 다루려고 했으나, 현재의 대체는 다이렉트 X 의 DirectInput 을 이용하여 해결하는 것이기 때문에 따로 다루지 않았다.

#### ● 미디어와 컴포넌트 레이어

앞에서 설명한 DirectDraw, DirectSound, DirectInput 을 기초 레이어라고 한다면, 이 위에는 미디어 레이어가 위치한다. 미디어 레이어에서는 개발자가 기초 레이어에서의 여러 컴포넌트들을 이용하여 진정한 멀티미디어를 구현할 수 있도록 해준다. 미디어 레이어보다 더 상위 레벨의 레이어가 바로 컴포넌트 레이어인데, 컴포넌트 레이어는 미디어 레이어와 기초 레이어의 컴포넌트를 조합하여 실제로 사용될 수 있는 컴포넌트 형태로 제공된다. 가장 대표적인 컴포넌트 레이어로는 NetMeeting, NetShow, ActiveMovie 등을 들 수 있겠다.

미디어 레이어에는 DirectShow, DirectModel, DirectAnimation, DirectPlay, Direct3D Retained Mode, VRML 등이 있다. DirectShow는 쿼타임, avi, wav, mpeg 등의 오디오와 비디오 스트림을 재생할 수 있도록 해준다. 이를 구현하기 위해 많은 필터를 필터 그래프 관리자(filter graph manager)를 통해 데이터 스트림에 연결하는데, 이를 구현하여 새로운 형식의 비디오 스트림을 만들어볼 수도 있다.

DirectModel은 커다란 3차원 그래픽 객체에 대한 렌더링과 상호작용 수단을 제공하며 DirectAnimation은 개발자가 사운드, 동영상, 텍스트를 조합하여 이들을 시간에 맞도록 통합하여 관리할 수 있는 애니메이션 작업을 지원한다. DirectPlay는 온라인 게임과 네트워크에 대한 지원을 강화하여 플레이어 간의 메시지를 주고 받는 등의 상호작용이 가능하다. 최근의 네트워크 지원 상황을 감안할 때 많은 개발자들이 접근하고자 하는 부분 중에 하나이다. 마이크로소프트의 인터넷 게임존([www.zone.com](http://www.zone.com))에서는 DirectPlay를 완벽하게 지원하고 있다.

## 정 리 (Summary)

이번 장에서는 하드웨어에 접근하여 여러가지 작업을 하는 방법에 대해서 알아보았다. 최근의 윈도우 환경에서는 다이렉트 X를 이용하여 하드웨어를 다루는 방법이 거의 표준으로 굳어지고 있으며, 예제도 많고 비교적 향상된 수행 성능을 보여주고 있기 때문에 그 활용 가능성은 밝다고 말할 수 있겠다.

델파이를 이용해서도 다이렉트 X를 마음대로 조작할 수 있으며, 잘 만들어진 클래스도 많이 제공되고 있으므로 많은 개발자들이 여기에 관심을 가지고 다이렉트 X의 발전된 기능을 어플리케이션에 도입하여 사용할 수 있기를 고대한다.

# 어플리케이션의 최적화와 세계화

## (Considering Optimization and Internatinalization of Applications)

도스 시절에 C 를 이용해서 프로그래밍을 해본 경험이 있는 분들은 그 당시에 최적화를 위해서 얼마나 많은 신경을 썼는지 기억할 것이다. 최근의 프로그래밍 환경은 그 당시에 비해 최적화에 대한 고려를 그다지 하지 않아도 좋은 환경으로 바뀌었다. 그렇지만, 나름대로의 최적화는 항상 가능한 법이다.

최적화 기법은 대단히 어렵고 복잡한 것에서부터, 쉽고 단순한 것까지 다양하게 존재한다. 또한, 바야흐로 세계화 시대가 개막되면서 우리가 개발하는 각종 프로그램의 세계화를 고려하는 것도 자연스러운 일이 되었다. 이번 장에서는 델파이에서 손쉽게 고려할 수 있는 간단한 최적화 기법과 세계화를 하기 위해 알아야 할 방법들을 소개하고자 한다.

### 최적화 기법

#### ● 리소스 절약

커다란 어플리케이션을 제작하다보면 리소스 부족으로 인해 에러가 발생하는 경우가 있다. 이럴 때에는 리소스를 절약하는 각종 테크닉을 사용해야 한다. 다음과 같은 방법으로 리소스를 절약할 수 있다.

##### 1. 폼의 자동 생성(autocreation)을 피한다.

프로젝트에서 새로운 폼을 추가하게 되면, 자동적으로 어플리케이션의 시작 코드에 다음과 같은 형태로 폼을 자동 생성하는 코드가 추가 된다.

```
uses
    Form1 in 'Unit1.pas',
    Form2 in 'Unit2.pas',
begin
    Application.CreateForm(TForm1, Form1);
    Application.CreateForm(TForm2, Form2);
    ...(중략)
```

end;

이렇게 하면 처음 어플리케이션이 시작할 때 느려질 뿐 아니라 많은 양의 메모리를 소모하게 된다. 그러므로, 메인 폼만 자동 생성하도록 하고, 다른 폼들은 Project|Options 메뉴에서 자동 생성을 하지 않도록 설정하고, 다음과 같은 형태로 필요할 때 만들어서 사용하도록 한다.

```
procedure MainForm.mnuChildFormClick(Sender: TObject);
begin
    ChildForm1 := TChildForm1.Create(Self);
    ChildForm1.ShowModal;
    ChildForm1.Release;
end;
```

## 2. 불필요한 OLE 지원 부분을 제거한다.

어플리케이션이 200KB 가 되지 않는 작은 크기라도, 델파이의 RTL 에서 기본적으로 OleAut32.dll 을 지원하기 위해서 약 1MB 의 RAM 을 사용하게 된다. 만약 어플리케이션에서 OLE 를 사용하지 않는다면 다음과 같은 코드로 이런 불필요한 메모리 낭비를 줄일 수 있다.

```
FreeLibrary(GetModuleHandle('OleAut32'));
```

메인 폼의 OnCreate 이벤트나 프로젝트 소스에서 이런 코드를 추가하면 OleAut32.dll 과 Ole32.dll 을 메모리에 적재하지 않게 되므로, 약 1MB 정도의 메모리를 절약할 수 있다.

## 3. 각각의 윈도우는 상당히 많은 양의 동적 리소스를 소모하므로 한번에 4~5 개 이상의 윈도우가 동시에 열리지 않도록 조치한다.

## 4. 같은 조건이라면 TPanel 보다는 TBevel 컴포넌트를 사용하는 것이 좋다. TPanel 컴포넌트는 각각의 윈도우에 대한 분리된 핸들이 필요한데 비해 TBevel 은 하나의 핸들만 사용한다. 그리고, TPanel 컴포넌트는 보기보다 많은 수의 프로퍼티와 이벤트가 연결되어 있기 때문에, dfm 파일의 크기를 많이 증가시킨다. 그 밖에 동일한 기능을 하는 컴포넌트 들이 있다면 되도록 적은 수의 프로퍼티와 이벤트를 가지고 있는 컴포넌트를 우선 사용하도록 한다.

5. 폼의 ParentFont 프로퍼티를 TRUE 로 설정하는 것이 좋다. 이렇게 하면 각각의 독립된 컴포넌트에 대해 독립된 폰트 인스턴스를 로드할 필요가 없다.
6. 컴포넌트를 만들 때에는 property 선언부에 default 값을 반드시 설정하도록 한다. default 값이 없으면 어플리케이션에서 dfm 파일에 무조건 값을 저장하게 되지만, default 로 설정된 값이 있으면 변경되지 않는 한 저장되지 않기 때문에 dfm 파일의 크기를 줄일 수 있다.
7. 가능한 델파이의 폼의 자동 생성 기능은 꺼두는 것이 좋다. 메인 폼을 제외한 다른 폼들이 모두 자동 생성될 경우 로딩 시간이 길어지고, 힙 메모리의 낭비를 가져온다. 그러므로, 다소 번거롭더라도 폼은 동적으로 생성하고 해제하도록 한다.
8. 델파이의 폼을 사용하지 않는 프로그램을 개발하는 경우이면, uses 절에서 Forms.pas 유닛을 삭제하도록 한다. Forms.pas 유닛은 실행 파일의 크기를 100KB 이상 커지게 만든다.

#### ● 예외 처리

어플리케이션을 개발할 때 예외 처리 루틴을 이용해서 각종 리소스를 보호하고, 예외가 일어났을 때 사용자에게 보다 세련된 방법으로 이를 알릴 수 있게 만들 수 있다. 그렇지만, 예외 처리 자체가 실행 속도를 저하시키는 단점이 있다. 일단 예외가 발생하면 운영체제와 어플리케이션 코드는 예외를 처리하게 위해 수백 개에 이르는 명령어를 더 처리해야 한다. 다음의 코드를 살펴 보자.

```
function GetSquare(P: PInteger): Integer;
begin
    try
        Result := Sqr(P^);
    except
        ShowMessage('P was nil !');
        Result := -1;
    end;
end;
```

여기에서 P 가 nil 이면 예외가 발생한다. 이때 예외를 처리하는데 수행 속도가 저하되므로,

이를 위해서는 다음과 같이 if 문으로 검사하는 것이 효과적이다.

```
function GetSquare(P: PInteger): Integer;
begin
    if (P = nil) then
    begin
        ShowMessage('P was nil !');
        Result := -1;
    end
    else Result := Sqr(P^);
end;
```

또한, 예외 처리를 사용하면 블록에서 빠져나가기 위해 Exit 문을 사용할 경우가 생기는데, 이 역시 성능을 저하시킨다. 다음의 코드를 살펴보자.

```
var
    P: PChar;
begin
    GetMem(P, 1024);
    try
        ...
        if (P^ = 'Sample') then Exit;
        ...
    finally
        FreeMem(P, 1024);
    end;
end;
```

이렇게 try..finally 블록에서 빠져나가기 위해서 Exit 를 사용하는 것보다는 다음과 같이 하는 것이 더 효과적이다.

```
var
    P: PChar;
begin
    GetMem(P, 1024);
    try
```

```

...
if (P^ <> 'Sample') then
begin
    ...
end;
finally
    FreeMem(P, 1024);
end;
end;

```

## ● 문자열 처리

텔파이 2 이후 부터는 문자열의 255 자 길이 한계가 없어지면서 동적으로 메모리를 할당받는다. 그리고, 문자열 자체도 참조값으로 취급되기 때문에 비교적 빠르게 동작한다. 그렇지만, 아직도 문자열을 처리할 때 몇 가지 고려해야할 것들이 있다.

먼저 문자열에 필요 없는 내용을 가지고 있는 경우이다. 다음 코드를 살펴보자.

```

procedure SampleString;
var
    S: String;
begin
    S := '';
    ...
end;

```

여기서 S 변수에 대한 대입문은 전혀 필요 없는 부분이다.

이 경우는 문자열 처리에 대한 실행속도의 최적화에 해당되는 내용이다.

문자열을 다룰 때에는 코드의 크기를 최적화 하는 방법도 생각해야 한다. 예를 들어 다음과 같이 반복되는 문장에 대한 대입문이 있다고 하자.

```

S1 := '이것은 지구상에서 가장 크다';
S2 := '이것은 지구상에서 가장 무겁다';
S3 := '이것은 지구상에서 가장 빠르다';

```

이 경우에 '이것은 지구상에서 가장 ' 이라는 문자열이 반복되고 있다. 이렇게 해서 만들어진 실행파일을 살펴보면 반복된 문자열이 그대로 실행파일에 들어있는 것을 알 수 있다.



그러므로, 이를 다음과 같이 처리하면 실행파일의 크기를 줄일 수 있다.

```
const
  S = '이것은 지구상에서 가장 ';
begin'
  S1 := S + '크다';
  S2 := S + '무겁다';
  S3 := S + '빠르다';
end;
```

이 경우에는 상수가 컴파일 시에 정해지므로 그다지 큰 효과를 거두기는 어렵다. 그렇지만 상수의 선언을 다음과 같이 해보자.

```
const
  S: String = '이것은 지구상에서 가장 ';
```

이렇게 하면 S 가 문자열로 한번만 할당받고, 뒤쪽에 붙는 문자열 부분은 런타임에서 적절하게 처리된다. 그러므로, 중복되는 부분만큼 코드 크기를 줄여줄 수 있다. 물론 컴파일 시에 처리하는 것보다 속도의 저하는 가져올 수 있다. 그렇지만, 여러 번 중복되는 대입문이 있을 때에는 반드시 고려해야 할テクニック이다.

#### ● 순환문과 판단문의 최적화

어플리케이션의 수행 속도에는 순환문의 처리 방식이 큰 영향을 미친다. 이중에서도 특히 for 와 while 문을 잘못 사용하면 커다란 수행 속도의 손실을 가져올 수 있다. 예를 들어, 다음의 코드를 살펴 보자.

```
while i <= SomeFunction(AParameter) do
begin
  ...
end;
```

여기에서 SomeFunction 함수가 정수값을 돌려준다고 할 때, 이 함수는 루프를 도는 동안 계속해서 실행된다. 그러므로, 이 함수의 실행에 0.1 초가 걸린다고 가정하고 루프를 100 번을 돌 경우 10 초가 걸리는 셈이다. 이런 경우에는 다음과 같이 for 문을 이용하면 한번만 실행되므로 수행 속도를 대폭 향상시킬 수 있다.

```

for I := 1 to SomeFunction(AParameter) do
begin
    ...
end;

```

판단문에 있어서는 가능한 else 문을 사용하지 않는 것이 요령이다. 예를 들어, 다음의 코드는 쉽게 else 를 떼어 버리도록 변경이 가능하다.

```

if Sample = False then A := x else A := y;

```

이 문장은 다음과 같이 변경할 수 있다.

```

A := x;
if Sample = True then A := y;

```

별것 아닌 것 같지만 이렇게 변경하는 것으로 jump, return 이라는 2 개의 기계어 코드를 절약할 수 있다.

최적화를 고려하면 판단문에서 중첩된 if...then...else 문장은 스택에 계속적인 push 를 하게 되므로 스택의 오버 플로우를 유발할 수도 있고, 효과적이지도 않다. 이런 경우에는 가능한 case 문장 등을 동원하여 변경하는 것이 좋다.

예를 들어 다음의 코드를 살펴보자.

```

if bps = 2400 then SomeFunc1
else if bps = 9600 then SomeFunc2
    else if bps = 1400 then SomeFunc3 ...

```

이런 경우에는 다음과 같이 변경하는 것이 바람직하다.

```

case bps of
    2400: SomeFunc1;
    9600: SomeFunc2;
    14400: SomeFunc3;
    ...
end;

```

## ● 간단한 최적화 기법

앞서 설명하지 않은 몇 가지 간단한 최적화 기법을 정리해서 소개하면 다음과 같은 것들이 더 있다.

1. DLL 에 커다란 파라미터를 넘기는 것은 금물이다. DLL 은 호출하는 프로그램과 스택을 공유하기 때문에, 에러가 발생하기 쉽다.
2. 당연한 이야기이지만, 순환 링크(circular link)는 사용하지 않도록 한다. 유지 보수가 어렵고 스마트 링크가 동작하지 않도록 한다.
3. 부득이한 경우가 아니면 Byte 데이터 형은 사용하지 않는다. CPU 의 최소 처리 단위가 Word 이기 때문에 저장 공간의 절약을 위해 Byte 를 사용할 경우 생각보다 많은 절약이 되지 않을 뿐더러, 컴파일러에 의해 이를 처리하기 위한 많은 코드가 생성되어야 하기 때문에 비효율적이다.
4. 마찬가지로 이유로 Boolean 데이터 형도 Integer 로 변경하여 사용해도 되는 경우라면 Integer 형을 사용할 것을 권한다. Boolean 데이터 형도 비효율적인 데이터형 이다.

## ● 테이블과 데이터베이스 최적화 기법

테이블과 데이터베이스를 다룰 때에도 속도를 향상시킬 수 있는 수많은 테크닉들이 있다. 기본적인 테크닉으로는 다음과 같은 것들이 있다.

1. 테이블을 검색을 할 때에는 인덱스를 사용한다.
2. 한 테이블에 너무 많은 필드를 사용하지 않는다. 테이블 하나에 100 개 이상의 필드가 존재하면 메모리를 과도하게 사용하게 되고, 데이터 전송 시 무리가 오게 되므로 이런 경우 테이블을 열기 전에 몇 개의 테이블로 나눈다.
3. 테이블은 꼭 필요할 때에만 열어야 한다. 이는 테이블을 열 때 걸리는 시간이 많기 때문이며, 만약 테이블을 자주 열고 닫게 될 경우에는 테이블을 계속 열어 놓는 것이 좋다. 또한, 여러 개의 파트로 나누어진 어플리케이션을 만들 경우에는 (예를 들어 학생용, 교수용, 통계용 클라이언트가 따로 있는 경우) 테이블을 여는 작업을 폼의 OnCreate, OnClose 이벤트에서 작업하는 것이 좋다. 이렇게 하면, 잠금(locking)이나 업데이트 문제를 예방할 수 있다.

4. 서버 데이터베이스에 Range 를 사용할 때에는 서버가 레코드를 어떤 식으로 접근하는지 알지 못하기 때문에, 특정 범위의 레코드를 변경하고자 할 때에는 SQL 문장의 UPDATE 절을 사용하는 것이 좋다. 이렇게 UPDATE 절을 사용해야 레코드에 대한 모든 작업이 서버에서 작동하며 클라이언트로 변경할 레코드를 넘겨받아서 다시 넘겨주는 불필요한 작업을 거칠 필요가 없기 때문에 효과적이다.
5. 테이블에 대량으로 접근해서 조작을 할 때에는 TTable.DisableControls 를 사용해서 데이터 컨트롤과의 연결을 끊은 후 조작이 끝나고 TTable.EnableControls 를 호출한다.
6. 가능한 OnCalcFields 이벤트는 사용하지 않는다. 속도를 대단히 떨어뜨리는 요인이 된다. 대신 TDataSource.OnDataChange 이벤트를 활용하는 것이 요령이다.
7. DBase 테이블을 SQL 을 사용해서 접근하며 매우 느리다. 이는 서버의 작업이 이루어진 후 결과가 되는 세트 만을 전송해야 하는데, DBase 테이블의 경우에는 네트워크를 통해 데이터가 클라이언트로 모두 넘어온 후 처리되기 때문이다. 그러므로, 네트워크가 느리고 DBase 테이블이 큰 경우에는 심각한 네트워크 트래픽과 클라이언트 부담을 안을 수 밖에 없다. 이럴 때에는 빨리 파라독스로 전환하는 것이 요령이다.

#### ● 실행파일의 크기를 줄이려면 ...

실행 파일의 속도도 중요하지만, 가능한 1 바이트라도 실행 파일의 크기를 줄이는 것도 중요하다. 실행 파일의 크기를 줄이는 기본적인 테크닉으로는 다음과 같은 것들이 있다.

1. 가능한 if..then..else 절보다는 case 문을 사용하는 것이 좋다.
2. 컴파일러 옵션의 활용

프로젝트에 연결되는 모든 .pas 파일과 .dpr 파일의 제일 위에 다음과 같은 줄을 추가한다.

```
{ $D-,L-,O+,Q-,R-,Y-,S- }
```

{ \$D- }는 코드에 디버그 정보를 없애는 것으로 실행 파일을 배포할 때에는 이 옵션을 사용한다. { \$L- }는 로컬 심볼을 코드에 추가하지 않는 것이며, { \$O+ }는 컴파일러에게 불필요한 변수를 제거해서 코드를 최적화하도록 한다.

{ \$Q- }는 정수 오버플로우 검사하는 코드를 제거하며, { \$R- }는 문자열, 배열 등의 범위 검

사 코드를 삭제한다. 또한, {\$S-}는 스택 검사에 대한 코드를 사용하며 {\$Y-}는 심볼 정보를 코드에 추가하지 못하도록 한다.

이렇게 하면 실행 파일의 크기가 약 10~20% 정도가 작아진다.

참고로 이 작업은 실행 파일을 배포할 단계에서 마지막으로 빌드할 때 적용해야 한다.

3. Project|Options|Compiler 페이지에서 ‘Show Hints’, ‘Show Warnings’ 옵션을 체크하고 프로젝트를 빌드하면 사용되지 않는 변수, 프로시저/함수를 보여준다. 이를 이용해서 불필요한 부분을 제거한다.

불필요한 유닛에 대한 uses 절을 제거한다. 델파이의 ‘스마트-링킹(Smart-Linking)’이 우수하지만, 쓰이지 않는 코드를 모두 제거해주지는 않는다는 것을 알아야 한다.

#### ● 옵티마이저(Optimizer)와 RTTI

델파이의 옵티마이저는 그 성능이 뛰어난 것으로 정평이 나있다. 옵티마이저가 메모리에 있는 변수에 대한 할당 부분을 CPU 레지스터를 활용하거나, 공통적인 표현식을 제거하여 최적화하는 등의 여러가지 일을 해준다. 그렇지만, 개발자가 신경을 써주면 옵티마이저의 이러한 특성을 더욱 잘 발휘하게 할 수도 있다. 다음의 코드를 살펴보자.

```
function DummyFunc(i: Integer): Integer;
```

```
begin
```

```
    Result := i;
```

```
end;
```

```
procedure Sample(i: Integer);
```

```
begin
```

```
    if (DummyFunc(i) > 1000) then ShowMessage('Thousands')
```

```
    else if (DummyFunc(i) > 100) then ShowMessage('Hundreds')
```

```
    else if (DummyFunc(i) > 10) then ShowMessage('Tens')
```

```
    else ShowMessage('A few');
```

```
end;
```

이 코드를 다음 코드와 비교해 보자.

```
procedure Sample(i: Integer);
```

```
var
```

```
    j: Integer;
```

```

begin
  j := DummyFunc(i);
  if (DummyFunc(i) > 1000) then ShowMessage('Thousands')
  else if (DummyFunc(i) > 100) then ShowMessage('Hundreds')
  else if (DummyFunc(i) > 10) then ShowMessage('Tens')
  else ShowMessage('A few');
end;

```

언뜻 보기에는 거의 달라진 것이 없어 보이지만, 여기에서는 j 변수의 값이 일단 CPU 레지스터에 저장되도록 최적화되기 때문에 코드가 빠르게 수행된다.

델파이의 RTTI(Run-time Type Information)를 활용할 때에도 최적화를 할 수 있는 경우가 있다. 예를 들어 다음과 같은 코드를 생각해보자.

```

uses MPlayer;
...

function Sample(Button: TMPBtnType): String;
begin
  case Button of
    btPlay: Result := 'btPlay';
    btPause: Result := 'btPause';
    ...
  end;
end;

```

이때 이렇게 나열한 부분이 실행 파일에 포함된다. 그러므로, 이로 인한 실행파일의 크기 증가와 코딩 시간의 낭비 등을 가져올 수 있는 것이다. 앞의 버튼 이름은 이미 RTTI에 기록되어 있기 때문에 다음과 같이 RTTI를 직접 이용하면 이러한 낭비를 막을 수 있다.

```

uses MPlayer, TypInfo;
...

function Sample(Button: TMPBtnType): String;
begin
  Result := GetEnumName(TypeInfo(TMPBtnType), Ord(Button));

```

end;

그 밖에도 수 많은 최적화テクニック이 존재하겠지만 여기서는 이 정도로 줄이고자 한다. 최근과 같이 좋은 개발 환경에서도 신경을 쓰면 작게나마 코드의 크기를 줄이거나, 실행 효율을 향상시키는 최적화 기법은 존재하기 마련이다.

언제나 코딩을 할 때 내가 사용한 코드가 효율적인지를 다시 한 번쯤 되돌아 볼 줄 아는 습관을 들이는 것이 중요할 것이다.

## 어플리케이션의 세계화 (Internationalization)

세계화(internationalization)는 프로그램을 여러 나라에서 사용할 수 있도록 하는 과정이다. 이 과정에서 각 나라의 언어와 문화적인 요소(시간 표기법 등) 등의 사용자 환경을 locale 이라고 한다. 윈도우는 많은 세트의 locale 들을 지원하는데, 각각의 locale 에 맞도록 번역하는 과정을 지역화(localization)이라고 한다.

앞으로 설명할 내용에서는 세계화는 여러나라의 locale 을 쉽게 적용할 수 있도록 어플리케이션을 처음 디자인하고 코딩할 때부터 신경써야 할 내용에 대한 것을 설명하는 것이고, 지역화는 특정 locale 에 적용하기 위해서 어떤 것을 고려해야 되는지에 대해서 설명할 것이다.

### ● 어플리케이션 세계화의 단계

어플리케이션을 세계화하기 위해서는 일반적으로 다음에 설명하는 몇 가지 단계를 거쳐야 한다. 일단 문자 세트를 여러 가지 활용할 수 있도록 문자열을 다루어야 할 것이며, 사용자 인터페이스를 쉽게 변경할 수 있어야 할 것이다. 그리고, 핵심적인 사항은 될 수 있는 한 지역화할 모든 리소스를 분리할 필요가 있다.

#### 1. 어플리케이션 코드의 변경

세계화를 고려할 때 가장 중요한 점은 다양한 locale 에 적용할 수 있도록 문자열을 변경할 수 있어야 한다는 점이다. 각 나라의 문자 세트는 코드 페이지라고 불리는 각 나라 고유의 페이지를 이용한다. 경우에 따라서는 일반적으로 많이 사용되는 ANSI 윈도우 문자 세트를 어플리케이션 사용자 기계의 코드 페이지에 지정된 문자 세트(OEM 문자 세트)로 변환할 필요가 있다.

이때 우리나라를 비롯한 아시아 나라의 문자 세트의 경우 단순한 1:1 매핑으로 해결할 수가 없는 경우가 많다. 그렇기 때문에 2 바이트를 처리할 수 있는 문자열 처리 함수들이 많이 필요하게 되는데, 그렇기 때문에 문자열의 길이를 바이트로 처리하는 내용을 담고 있을

때에는 반드시 이를 고려해서 프로그래밍해야 한다.

이런 어려운 점을 해결하기 위한 방안으로 가장 좋은 것 중의 하나는 유니코드를 이용하는 것이다. 유니코드 문자 세트는 델파이에서 WideChar 데이터 형으로 지원하고 있다. 윈도우 NT 라면 이것으로 거의 문제를 해결할 수 있지만, 윈도우 95 를 사용할 경우에는 API 함수가 지원되는 것이 별로 없다는 단점이 있다. 그렇기 때문에, 델파이의 VCL 소스 코드를 보면 대부분의 문자열을 다루는 루틴이 ANSI 의 단일 바이트 문자 세트와 유니코드를 지원하는 DBCS 문자 세트를 모두 지원하도록 되어 있다. VCL 소스 코드를 자주 들여다 보는 사람이라면 쉽게 알 수 있겠지만 각종 루틴의 말미에 'A'가 붙어 있으면 ANSI 를 지원하는 것이며 'W'가 붙어 있으면 유니코드를 지원하는 것이다.

그 밖에도 IME(input method editor)를 직접 제어하여 사용자의 입력을 제어할 수 있도록 하는 것도 좋은 지역화의 한 방법이다. 델파이 3 부터는 VCL 컴포넌트가 IME 를 이용할 수 있도록 제공되고 있다. 대부분의 윈도우 컨트롤은 텍스트 입력을 받을 때 ImeName 프로퍼티를 이용하여 특정 IME 를 선택할 수 있도록 지원하고 있으며, ImeMode 프로퍼티를 이용하여 IME 가 키보드 입력을 어떤 방식으로 전환할 것인지 지정할 수 있다.

또한, 전역 Screen 변수에서 사용자 시스템에서 이용가능한 IME 들에 대한 정보를 얻을 수 있으며, 그 밖에도 사용자 시스템에 설치된 키보드 매핑에 대한 정보를 얻을 수 있다.

## 2. 사용자 인터페이스 디자인

문자 세트에 대한 고려도 중요하지만, 여러 나라에 어플리케이션을 개발해서 배포하기 위해서는 사용자 인터페이스를 디자인하는 것도 중요하다. 이때 사용자 인터페이스는 각 나라의 환경에 맞도록 쉽게 수정할 수 있도록 해야 한다.

예를 들어, 컨트롤의 캡션 등은 여러나라의 문자로 바뀔 수 있으므로 해당되는 나라의 문자열로 바꾼 뒤에도 캡션이 나타날 수 있도록 넉넉한 공간이 확보되어야 할 것이다.

이렇게 번거로운 문자열 변환 과정을 최소한으로 줄이기 위해서는 이미지를 잘 활용하는 것이 요령이다. 그런데, 이미지에 문자를 같이 사용해야 되는 경우라면 하나의 이미지에 문자를 포함해서 사용하지 말고 투명 배경을 이용해서 텍스트 부분 만을 따로 이미지를 이용할 수 있도록 하는 것이 좋다.

그리고, 각 locale 별로 변경해서 사용해야 할 것으로는 날짜와 시간, 숫자, 화폐 단위 등이 있다. 윈도우 포맷만을 사용한다면 이러한 정보를 윈도우 레지스트리에서 얻을 수 있다.

## 3. 리소스의 분리

어플리케이션을 세계화할 때 코드를 특별히 변경하지 않고 쉽게 여러나라 언어로 지역화하기 위해서는 사용자 인터페이스의 문자열을 독립된 단일 모듈을 분리하여 사용하는 것이 좋다. 델파이는 메뉴, 대화상자, 비트맵 등에 대한 리소스를 포함한 .dfm 파일을 자동으로 생성



성하므로, 여기의 내용을 텍스트 에디터로 직접 편집하는 것도 한 방법이다.

그 밖에 문자열을 따로 분리하여 사용할 수 있는 방법이 있는데 여기에 대해서 더욱 자세히 알아보도록 하자.

## ● 리소스 DLL 의 제작

리소스를 분리하면 지역화를 매우 편하게 진행할 수 있다. 리소스 분리를 할 때 편리하게 사용할 수 있는 방법의 하나가 리소스 DLL 을 생성하여 사용하는 것이다. 리소스 DLL 을 작성하면 리소스 DLL 을 변경하는 것으로 간단하게 번역 작업을 대신할 수 있고, 배포 작업 역시 간단하게 만들 수 있다.

델파이 4 에서 제공되는 리소스 DLL 위저드를 이용하면 쉽게 리소스 DLL 을 생성할 수 있다. 리소스 DLL 위저드는 현재의 저장된 프로젝트에 대해서 RC 파일과 프로젝트에서 사용된 resourcestrings 지시어로 지정된 문자열 테이블을 포함한 RC 파일을 생성한다. 그리고, 적절한 폼과 RES 파일의 내용을 포함한 리소스 DLL 에 대한 프로젝트를 생성한다.

지원하고자 하는 locale 이 있다면, 그 수만큼의 리소스 DLL 을 생성하면 된다. 이때 각각의 리소스 DLL 은 해당 locale 에 대한 확장자를 지정하는데, 보통 처음 2 자는 해당 언어를 가리키며, 마지막 문자는 나라를 지시한다. 한국어의 경우는 언어와 사용하는 나라가 일치하므로 kor 로 표시할 수 있지만, 영어의 경우에는 영국, 미국, 캐나다 등 여러 나라를 지정할 수 있다. 이 경우 미국의 경우 enu(english + US), 캐나다의 경우 enc(english + Canada)가 확장자로 사용된다.

## ● 리소스 DLL 의 활용

리소스 DLL 을 이용하면 실행파일과 다른 DLL, 패키지에 포함된 리소스를 쉽게 지역화된 버전으로 만들 수 있다. 어플리케이션을 시작할 때 로컬 시스템의 locale 을 검사하고, EXE, DLL, BPL 파일과 같은 이름의 리소스 DLL 을 발견하게 되면 확장자를 검사해서 확장자가 시스템 locale 의 나라와 언어에 부합하는지 알아보고, 그렇다면 실행 파일이나 DLL, 패키지에 포함된 리소스 대신 그 리소스 모듈의 리소스를 사용한다. 그리고, 해당되는 언어와 나라에 대한 리소스 모듈이 없을 경우에는 기본적으로 언어에 해당되는 것을 찾고 그 조차도 없을 경우에는 실행 파일이나 DLL, 패키지에 포함되어 컴파일된 리소스를 사용한다. 어떤 경우에는 어플리케이션에서 로컬 시스템의 locale 과 매치되는 리소스 모듈 이외의 리소스 모듈을 사용하고 싶을 때가 있다. 이럴 때에는 윈도우 레지스트리의 엔트리를 설정하면 된다. HKEY\_CURRENT\_USER\Software\Borland\Locales 키에 어플리케이션의 패스와 파일 이름을 문자열 값으로 추가하고, 데이터 값을 리소스 DLL 의 확장자를 데이터 값으로 설정하는 것이 그 방법인데, 어플리케이션이 시작할 때 시스템 locale 에 해당되는 리소스 모듈을 찾기 전에 어플리케이션이 이 키의 값을 이용하여 리소스 모듈을 찾는다.

예를 들어, 다음의 프로시저는 프로그램을 처음 설치하거나 설정할 때 레지스트리 키 값을 설정할 수 있다.

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
    Reg: TRegistry;
begin
    Reg := TRegistry.Create;
    try
        if Reg.OpenKey('Software\Borland\Locales', True) then
            Reg.WriteString(LocaleOverride, FileName);
    finally
        Reg.Free;
    end;
end;
```

이렇게 설정하고 나면, 다음과 같이 어플리케이션에서 FindResourceHInstance 함수를 이용하여 현재의 리소스 모듈의 핸들을 얻을 수 있다.

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

#### ● 리소스 DLL 의 동적 변경

리소스 DLL 은 어플리케이션이 시작될 때 설정하는 것이 보통이지만, 런타임에서 동적으로 이를 변경하는 것도 가능하다. 이를 위해서는 델파이의 데모 어플리케이션과 함께 제공되는 ReInit.pas 유닛을 사용하면 된다. ReInit.pas 유닛은 델파이 4 의 Demos 디렉토리의 RichEdit 디렉토리에 있으므로 이를 복사해서 사용하면 된다.

언어를 변경하려면 LoadNewResourceModule 함수에 새로운 언어에 대한 LCID 를 넘겨주고, ReinitializeForms 프로시저를 호출하면 된다. 예를 들어, 다음의 코드는 인터페이스 언어를 프랑스어로 변경한다.

```
const
    FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;

if LoadNewResourceModule(FRENCH) <> 0 then
    ReinitializeForms;
```

이 방법의 장점은 레지스트리의 설정을 변경하거나, 어플리케이션을 특별히 다시 시작할 필요 없이 리소스를 변경할 수 있다는 점이다.

이때 주의할 점은 런타임에서 폼에 대한 프로퍼티를 변경한 내용이 효력을 잃게 되므로, 일단 새로운 DLL 이 로드되면 디폴트 값이 재설정되지 않는다. 그러므로, 경우에 따라서는 폼 객체의 프로퍼티를 재설정 해주어야 한다.

리소스 DLL 을 동적으로 변경하는 예제는 텔파이 4 의 Demos 디렉토리의 RichEdit 서브 디렉토리에서 제공되는 RichEdit.dpr 프로젝트를 예제를 참고하기 바란다.

## ● 리소스 DLL 위저드

리소스 DLL 프로젝트를 생성하기 위해서 리소스 DLL 위저드를 사용하는 방법에 대해서 알아보자. 리소스 DLL 에는 어플리케이션에서 사용하는 모든 폼과 문자열에 대한 정보를 포함한다. 보통 리소스 DLL 은 어플리케이션을 배포할 준비가 모두 끝났을 때 작성하는 것이 좋다.

리소스 DLL 위저드를 이용하려면 프로젝트를 저장하고 빌드한 뒤에 File|New 메뉴에서 Resource DLL wizard 아이콘을 더블 클릭하면 된다. 이렇게 하면 텔파이는 변환이 필요한 모든 폼과 문자열을 포함한 파일의 리스트를 보여준다. 여기에서 다른 폼을 추가하거나 삭제하려면 Add, Remove 버튼을 이용하면 된다.

폼들이 모두 결정되었으면 Next 버튼을 클릭한다. 이렇게 하면 위저드가 추가적인 DRC 파일을 추가할 것인지 물어본다. 만약에 다른 DRC 파일을 사용하는 프로젝트라면 여기에서 Add 버튼을 클릭하여 추가할 수 있다.

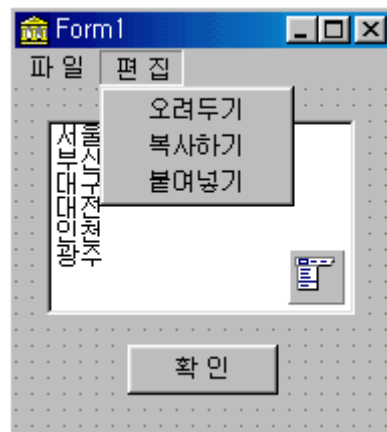
마지막으로 Next 버튼을 클릭하면 리소스 DLL 에 대한 언어를 선택하면 된다.

각각의 윈도우 locale 은 해당되는 파일 확장자를 가진다. 그리고, 리소스 DLL 에 대한 프로젝트는 언어 확장자와 같은 이름의 서브 디렉토리에 저장된다.

어플리케이션을 배포하려면, DLL 을 서브 디렉토리에서 프로젝트 디렉토리로 복사한 뒤에 같이 배포하면 된다.

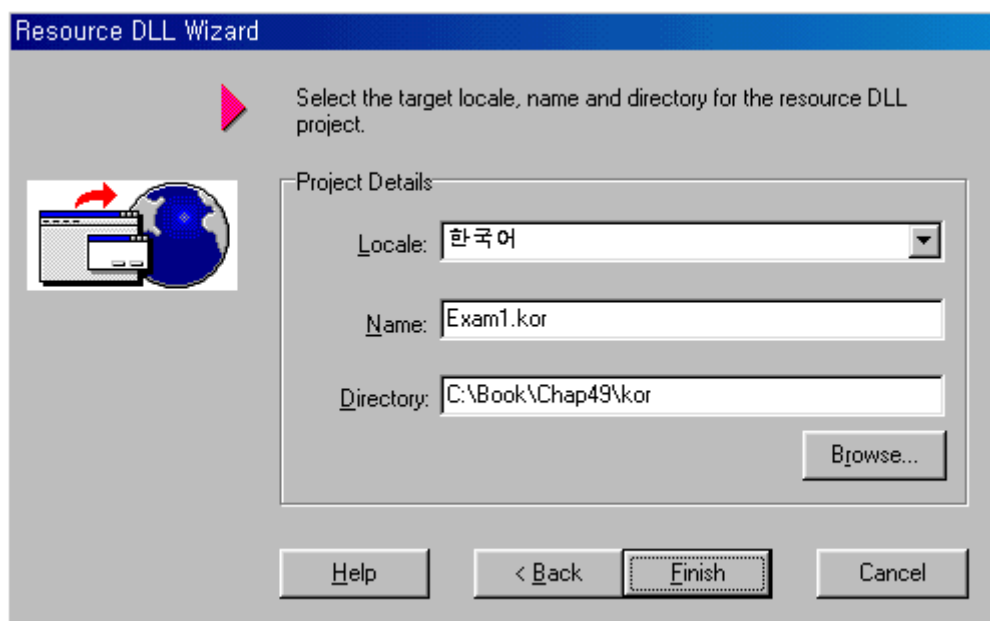
## 예제 어플리케이션의 제작

그러면, 한국어와 영어(미국과 캐나다)를 지원하는(아마도 우리나라 개발자는 이렇게 개발하는 경우가 가장 많을 것이다) 간단한 예제 어플리케이션을 개발해보도록 하자. 먼저 한국어를 지원하도록 다음과 같이 리스트 박스와 버튼, 메뉴 컴포넌트를 하나씩 엮고 폼을 디자인하도록 하자.

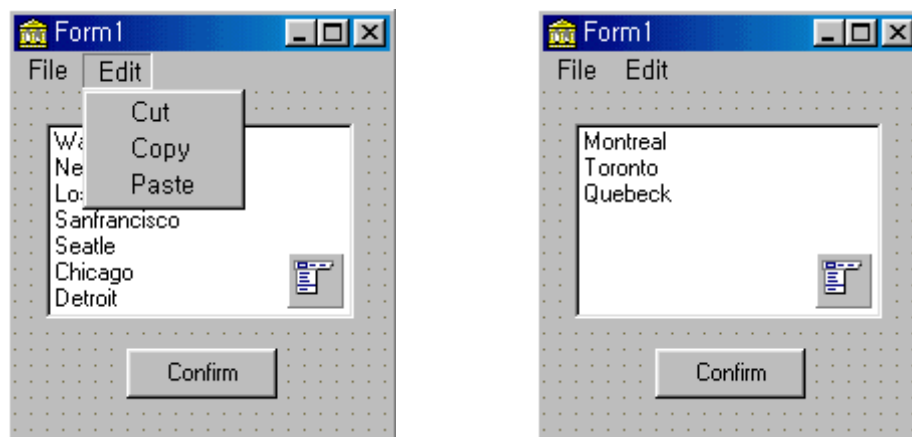


이번에 제작하는 어플리케이션의 목적은 2 개 언어와 3 개 나라를 지원하도록 리소스 DLL 위저드를 이용하는 방법을 익히는 것이므로, 특별한 코딩은 하지 않는다. 리소스 DLL 위저드를 사용하기 위해서는 프로젝트를 저장하고, 컴파일을 일단 해주어야 한다.

저장과 컴파일이 끝났으면 File|New 메뉴를 선택하고 Resource DLL Wizard 아이콘을 더블 클릭하여 리소스 DLL 을 생성하도록 하자. 리소스 DLL 위저드가 처음 실행되면 현재 프로젝트의 폼이 나타날 것이다. 이때 리소스 DLL 에 포함시킬 추가적인 폼이 있다면 Add 버튼을 클릭하여 추가하도록 한다. 그렇지만, 대부분의 경우 프로젝트 내부에 리소스가 포함되도록 디자인하므로 특별한 폼의 추가 없이 Next 버튼을 클릭하면 된다. 그 다음 화면에는 리소스 파일을 추가하도록 하는 화면이 나타나는데, 마찬가지로 추가할 .rc 파일이 있으면 여기에서 추가하도록 한다. 보통은 Next 버튼을 클릭하면 된다. 마지막으로 다음과 같이 Locale 과 DLL 파일의 이름, 디렉토리를 지정하는 대화상자를 완성하고 Finish 버튼을 클릭하면 리소스 DLL 프로젝트가 만들어진다.



리소스 DLL 프로젝트가 만들어 졌으면, 이를 컴파일하면 해당 디렉토리에 확장자가 .kor 인 DLL 파일이 생성된다. 나중에 이를 이용하여 한국어 버전을 배포하는 방법을 설명할 것이다. 다시 이전의 프로젝트 파일을 열고, 미국 버전을 위해 폰트를 MS San Serif 의 8 포인트로 설정한다 (필자는 한국어 어플리케이션을 제작할 때 폰트를 굴림, 9 포인트를 사용한다). 그리고, 폼을 다음과 같이 디자인한다. 참고로 캐나다 버전의 폼 디자인도 옆에 같이 나타내었다. 캐나다 버전을 만드는 방법은 미국 버전과 동일하므로 생략한다.



프로젝트 파일을 저장하고, 컴파일을 한 뒤에 다시 리소스 DLL 위저드를 실행한다. Next 버튼을 2 번 클릭하여 추가할 폼과 리소스 파일이 없다는 것을 나타낸 뒤에 마지막 대화 상자의 Locale 로 ‘영어(미국)’을 선택한다. 이렇게 하면 아마도 Name 은 ‘Exam1.enu’, Directory 는 ‘c:\WBook\WChap49\Wenu’로 설정될 것이다. 참고로 캐나다 버전의 경우에는 Locale 은 ‘영어(캐나다)’, Name 과 Directory 는 ‘Exam1.enc, c:\WBook\WChap49\Wenc’로 각각 지정하면 된다. Finish 버튼을 클릭하면 리소스 DLL 프로젝트가 생성되며, 이를 컴파일해서 .enu, .enc DLL 파일을 생성하도록 한다.

이것으로 리소스 DLL 의 제작이 완료되었다. 그러면, 이들을 직접 만들어 보자. 미국 버전과 캐나다 버전을 나중에 제작했으므로 현재의 Exam1 실행 파일은 영어를 지원할 것이다. 이 실행파일을 한국어를 지원하도록 하려면, Wkor 디렉토리의 Exam1.kor 파일을 실행파일 디렉토리로 옮기면 된다. 실행파일을 실행하면 처음에 제작한 한국어 버전의 폼이 나타나는 것을 볼 수 있다. 배포하는 방법은 이와 같이 각 나라의 로컬 버전에 맞는 리소스 DLL 파일을 실행파일 디렉토리에 포함해서 제공하면 된다.

## 정 리 (Summary)

델파이 4 를 이용하여 각종 프로그램을 제작하는 여러가지 단계가 모두 중요하겠지만, 마무리와 뒷처리도 그에 못지 않게 중요한 부분이다. 이번 장에서 다룬 최적화와 세계화라는

이슈는 이런 마무리에 해당되는 부분인 만큼 최대한 깔끔하고 깨끗하게 처리하는 것이 좋을 것이다.

이제 개발자들도 세계를 무대로 프로그램을 제작하는 시대가 되었다. 텔파이 4 에서 제공하는 리소스 DLL 위저드는 우리나라의 프로그래머들이 외국 시장을 쉽게 공략할 수 있는 터를 열어 놓았지만, 다른 측면에서 보면 미국이나 유럽 또는 인도 등의 소프트웨어 강국에서 쉽게 한국어 버전의 프로그램을 만들어낼 수 있는 장이 열렸다고도 할 수 있다.

그러므로, 개발자들도 우물안 개구리처럼 한국 시장에서만 통할 수 있는 프로그램을 개발하는데 열을 올릴 것이 아니라 바야흐로 시야를 넓혀서 세계 시장을 적극적으로 공략하는 발상의 전환이 필요한 시대이다.

저자소개 : 신현목

저자경력?

현 ONC Korea co.,Ltd. 부설 CORBA 연구소 연구소장

전 SPINTech 대표(<http://www.spintech.co.kr>)

전 하이텔 비주얼 파워툴 소모임 텔마당 2 대 회장(<http://www.delmadang.com>)

현 하이텔 비주얼 파워툴 대표시삽 3, 4 대 대표시삽(<http://www.visualtool.com>)

저자후기

텔파이 1 부터 텔파이 4 까지 이제 4 개의 버전업을 한 텔파이, 처음보았을때 떨리던 가슴을 진정시키던 기억이 아직도 생생하다. 강력한 개발환경, 짜임새있는 VCL, 멋진 인터페이스, 다양한 환경, 간단한 개발, 그리고, 강력한 수행능력. 분명 텔파이는 일류(?)개발툴임에 틀림없습니다.

그동안 공부했던 내용과 다른 분들에게 자그마한 지식이라도 나누고 싶어서 이책을 만드는데 참여했습니다. 기존의 텔파이책들이 너무 컴포넌트의 나열만으로써 이루어진 책들이 많기 때문에 조금은 색다른 모습의 책으로 책을 쓸려고 했는데 잘되었는지는 모르겠습니다. 처음에 많은 내용을 넣었으면 하는 욕심도 있었지만, 결국에는 지훈님과 계획잡았던 내용을 모두 채우지 못한 것이 아쉽습니다.

쓰고나니 조금은 이상한 텔파이 책이 되어버린 것 같지만, 이 책을 보는 모든 사람에게 도움이 되었으면 좋겠습니다. 온라인 상의 좋은 정보는 하이텔의 비주얼툴( go vtool )과, 인터넷의 비주얼 파워툴의 홈페이지(<http://www.visualtool.com>)와 텔마당의 홈페이지(<http://www.delmadang.com>)에도 들려주기 바란다. 아마 이책이 출판되었을때쯤에는 하이텔의 강좌란과 연계한 서비스도 보여줄 수 있을 것이다.

함께 글을 쓴 정지훈님이 너무 고생이 많으셨다. 공저로 작업한다고 해놓고선 바쁘다는 핑계의 제가 쓸 분량을 다 채우지 못해 고생하신 ‘정지훈님’에게 먼저 감사드리고, 글을 쓰는데 지대한 도움을 준 우리 어부인 ‘진숙이’에게도 고맙고, 컴퓨터 전원스위치를 눌러 작업하던 파일을 몇번 날려먹은 우리아가 ‘수철이’에게도 고맙고, 자료를 찾는데 도움을 준 우리 ‘SPINTech 직원들’에게도 감사감사... 그외 비주얼 파워툴의 ‘텔파이 개발자’여러분에게도 감사감사.

## 팁모음집

### 금주가 몇번째 주인지 어떻게 구합니까

function kcIsLeapYear( nYear: Integer ): Boolean; // 윤년을 계산하는 함수

begin

Result := (nYear mod 4 = 0) and ((nYear mod 100 <> 0) or (nYear mod 400 = 0));

end;

function kcMonthDays( nMonth, nYear: Integer ): Integer; // 한달에 몇일이 있는지를 계산하는 함수

const

DaysPerMonth: array[1..12] of Integer = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);

begin

Result := DaysPerMonth[nMonth];

if (nMonth = 2) and kcIsLeapYear(nYear) then Inc(Result);

end;

function kcWeekOfYear( dDate: TDateTime ): Integer; // 위의 두 함수를 써서 몇번째 주인지 계산하는 함수

var

X, nDayCount: Integer;

nMonth, nDay, nYear: Word;

begin

nDayCount := 0;

deCodeDate( dDate, nYear, nMonth, nDay );

For X := 1 to ( nMonth - 1 ) do

nDayCount := nDayCount + kcMonthDays( X, nYear );

nDayCount := nDayCount + nDay;

Result := ( ( nDayCount div 7 ) + 1 );

end;

### 긴 파일명 사용하기

function fileLongName(const aFile: String): String;

var

aInfo: TSHFileInfo;



```

begin
  if SHGetFileInfo(PChar(aFile),0,aInfo,Sizeof(aInfo),SHGFI_DISPLAYNAME)<>0 then
    Result:=StrPas(aInfo.szDisplayName)
  else
    Result:=aFile;
end;

```

## 네트워크 검색

```

connections or persistent (won't normally get here);}

  r:=WNetOpenEnum(ListType,ResourceType,RESOURCEUSAGE_CONTAINER, nil,hEnum);
  { Couldn't enumerate through this container; just make a note of it and continue on: }
  if r<>NO_ERROR then
  begin
    AddShareString(TopContainerIndex,"");
    WNetCloseEnum(hEnum);
    Exit;
  end;

  { We got a valid enumeration handle; walk the resources: }
  while (1=1) do
  begin
    EntryCount:=1;
    NetResLen:=SizeOf(NetRes);
    r:=WNetEnumResource(hEnum,EntryCount,@NetRes,NetResLen);
    case r of
      0: begin
        { Yet another container to enumerate; call this function recursively to handle it: }
        if (NetRes[0].dwUsage=RESOURCEUSAGE_CONTAINER) or (NetRes[0].dwUsage=10)
        then
          DoEnumerationContainer(NetRes[0])
        else
          case NetRes[0].dwDisplayType of
            { Top level type: }
            RESOURCEDISPLAYTYPE_GENERIC,
            RESOURCEDISPLAYTYPE_DOMAIN,

```

```

        RESOURCEDISPLAYTYPE_SERVER: AddContainer(NetRes[0]);
    { Share: }
        RESOURCEDISPLAYTYPE_SHARE: AddShare(TopContainerIndex,NetRes[0]);
    end;
end;
ERROR_NO_MORE_ITEMS: Break;
else begin
    MessageDlg('Error #'+IntToStr(r)+' Walking Resources.',mtError,[mbOK],0);
    Break;
end;
end;
end;
end;

{ Close enumeration handle: }
WNetCloseEnum(hEnum);
end;

procedure TfrmMain.FormShow(Sender: TObject);
begin
    DoEnumeration;
end;

// Add item to tree view; indicate that it is a container:
procedure TfrmMain.AddContainer(NetRes: TNetResource);
var
    ItemName: String;
begin
    ItemName:=Trim(String(NetRes.lpRemoteName));
    if Trim(String(NetRes.lpComment))<>" then
    begin
        if ItemName<>" then ItemName:=ItemName+'  ';
        ItemName:=ItemName+'('+String(NetRes.lpComment)+')';
    end;
    tvResources.Items.Add(tvResources.Selected,ItemName);
end;

```

```
// Add child item to container denoted as current top:
procedure TfrmMain.AddShare(TopContainerIndex: Integer; NetRes:TNetResource);
var
    ItemName: String;
begin
    ItemName:=Trim(String(NetRes.lpRemoteName));
    if Trim(String(NetRes.lpComment))<>" then
    begin
        if ItemName<>" then ItemName:=ItemName+' ';
        ItemName:=ItemName+'('+String(NetRes.lpComment)+)';
    end;

    tvResources.Items.AddChild(tvResources.Items[TopContainerIndex],ItemName);
end;
```

```
{ Add child item to container denoted as current top;
    this just adds a string for purposes such as being unable to enumerate a container. That is, the
    container's shares are not accessible to us.}
procedure TfrmMain.AddShareString(TopContainerIndex: Integer;ItemName: String);
begin
    tvResources.Items.AddChild(tvResources.Items[TopContainerIndex],ItemName);
end;
```

```
{ Add a connection to the tree view.
    Mostly used for persistent and currently connected resources to be displayed.}
procedure TfrmMain.AddConnection(NetRes: TNetResource);
var
    ItemName: String;
begin
    ItemName:=Trim(String(NetRes.lpLocalName));
    if Trim(String(NetRes.lpRemoteName))<>" then
    begin
        if ItemName<>" then ItemName:=ItemName+' ';
        ItemName:=ItemName+'-> '+Trim(String(NetRes.lpRemoteName));
    end;

    tvResources.Items.Add(tvResources.Selected,ItemName);
```

end;

// Expand all containers in the tree view:

procedure TfrmMain.mniExpandAllClick(Sender: TObject);

begin

    tvResources.FullExpand;

end;

// Collapse all containers in the tree view:

procedure TfrmMain.mniCollapseAllClick(Sender: TObject);

begin

    tvResources.FullCollapse;

end;

// Allow saving of tree view to a file:

procedure TfrmMain.mniSaveToFileClick(Sender: TObject);

begin

    if dlgSave.Execute then

        tvResources.SaveToFile(dlgSave.FileName);

end;

// Allow loading of tree view from a file:

procedure TfrmMain.mniLoadFromFileClick(Sender: TObject);

begin

    if dlgOpen.Execute then

        tvResources.LoadFromFile(dlgOpen.FileName);

end;

// Rebrowse:

procedure TfrmMain.btnOKClick(Sender: TObject);

begin

    DoEnumeration;

end;

end.

네트워크 드라이브 등록하기

```
procedure TStartForm.NetBtnClick(Sender: TObject);
var
    OldDrives: TStringList;
    i: Integer;
begin
    OldDrives := TStringList.Create;
    OldDrives.Assign(Drivebox.Items);    // Remember old drive list
    // Show the connection dialog
    if WNetConnectionDialog(Handle, RESOURCETYPE_DISK) = NO_ERROR then
    begin
        DriveBox.TextCase := tcLowerCase;    // Refresh the drive list box
        for i := 0 to DriveBox.Items.Count - 1 do
        begin
            if Olddrives.IndexOf(Drivebox.Items[i]) = -1 then
            begin    // Find new Drive letter
                DriveBox.ItemIndex := i;    // Updates the drive list box to new drive letter
                DriveBox.Drive := DriveBox.Text[1];    // Cascades the update to connected directory lists,
            end;
        end;
        DriveBox.SetFocus;
    end;
end;
```

다른 윈도우에서 선택된 문자열 복사하기

```
procedure TForm1.WMHotkey(Var msg: TWMHotkey);
var
    hOtherWin,
    hFocusWin: THandle;
    OtherThreadID, ProcessID: DWORD;
begin
    hOtherWin := GetForegroundWindow;
    if hOtherWin = 0 then
        Exit;
```

```

OtherThreadID := GetWindowThreadProcessID( hOtherWin, @ProcessID );
if AttachThreadInput( GetCurrentThreadID, OtherThreadID, True ) then
begin
    hFocusWin := GetFocus;
    if hFocusWin <> 0 then
    try
        SendMessage( hFocusWin, WM_COPY, 0, 0 );
    finally
        AttachThreadInput( GetCurrentThreadID, OtherThreadID, False );
    end;
end;

Memo1.Lines.Add( Clipboard.AsText );

if IsIconIC( Application.Handle ) then
    Application.Restore;
end;

```

## 다른 Application 에 Data 전달하기

```

WM_COPYDATA-다른 Application 에 Data 전달
unit other_ap;
{ 다른 Application 을 찾아서 WM_COPYDATA 로 DATA 를 전달 }
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

const WM_COPYDATA = $004A;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Memo1: TMemo;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    end;

```

```

    procedure WMCopyData(var m : TMessage); message WM_COPYDATA;
public
    { Public declarations }
end;

var
    form1: Tform1;

implementation
{$R *.DFM}

type
    PCopyDataStruct = ^TCopyDataStruct;
    TCopyDataStruct = record
        dwData: LongInt;
        cbData: LongInt;
        lpData: Pointer;
    end;

type
    PRecToPass = ^TRecToPass;
    TRecToPass = packed record
        s : string[255];
        i : integer;
    end;

procedure TForm1.WMCopyData(var m : TMessage);
begin
    Memo1.Lines.Add(PRecToPass(PCopyDataStruct(m.LParam)^.lpData)^.s);
    Memo1.Lines.Add(IntToStr(PRecToPass(PCopyDataStruct(m.LParam)^.lpData)^.i));
end;

procedure Tform1.Button1Click(Sender: TObject);
var
    h : THandle;
    cd : TCopyDataStruct;
    rec : TRecToPass;

```

```

begin
  if Form1.Caption = 'My App' then
    begin
      h := FindWindow(nil, 'My Other App');
      rec.s := 'Hello World - From My App';
      rec.i := 1;
    end
  else
    begin
      h := FindWindow(nil, 'My App');
      rec.s := 'Hello World - From My Other App';
      rec.i := 2;
    end;
    cd.dwData := 0;
    cd.cbData := sizeof(rec);
    cd.lpData := @rec;
    if h <> 0 then
      SendMessage(h, WM_CopyData, Form1.Handle, LongInt(@cd));
    end;

  end.

```

델파이 중복실행방지

```

unit PrevInst;

interface

uses
  WinTypes, WinProcs, SysUtils;

type
  PHWND = ^HWND;

  function EnumFunc(Wnd:HWND; TargetWindow:PHWND): bool; export;
  procedure GotoPreviousInstance;

```



implementation

```
function EnumFunc(Wnd:HWND; TargetWindow:PHWND): bool;
var
    ClassName : array[0..30] of char;
begin
    Result := true;
    if GetWindowWord(Wnd,GWW_HINSTANCE) = hPrevInst then
    begin
        GetClassName(Wnd,ClassName,30);
        if StrIComp(ClassName,'TApplication') = 0 then
        begin
            TargetWindow^ := Wnd;
            Result := false;
        end;
    end;
end;

procedure GotoPreviousInstance;
var
    PrevInstWnd : HWND;
begin
    PrevInstWnd := 0;
    EnumWindows(@EnumFunc,longint(@PrevInstWnd));
    if PrevInstWnd <> 0 then
    begin
        if IsIconic(PrevInstWnd) then
            ShowWindow(PrevInstWnd, SW_RESTORE)
        else
            BringWindowToTop(PrevInstWnd);
    end;
end.
```

이러한 유닛을 프로젝트에 추가 하신후 DPR 소스의 BEGIN - END 를 다음과 같

수정해 주세요

```
begin
  if hPrevInst <> 0 then
    GotoPreviousInstance
  else
    begin
      Application.CreateForm(MyForm, MyForm);
      Application.Run;
    end;
  end.
end.
```

### 델파이에서 한글 토글하기

델파이 2.0 이하에서는  
ims.pas 를 이용하여 한영토글을 구현했는데,  
3.0 이상 에서는 한영토글에 대한 간단한 답에 있더군요.  
TEdit 에 ImMode 프라퍼티를 이용합니다.

```
edit1.ImMode:=imHangul; //한글모드
edit2.ImMode:=imAlpha;  //영문모드
```

입력이 한글이 많을 경우,  
입력 초기모드를 한글모드로 바꿔준다면,  
사용자의 한/영키를 누르는 것을 없애줄 수 있겠지요.

### 델파이에서 자동으로 한글입력모드로 변경시키는 소스

uses 절에 Imm 을 추가하세요

그런다음 아래 프로시저를 작성하여 OnEnter 이벤트에서  
한글을 on 하시구요 OnExit 이벤트에서 off 하세요

```
procedure TForm1.SetHangeulMode(SetHangeul: Boolean);
var
  tMode : HIMC;
begin
```

```

tMode := ImmGetContext(handle);
if SetHangeul then    // 한글모드로
                        ImmSetConversionStatus(tMode,      IME_CMODE_HANGEUL,
IME_CMODE_HANGEUL)

                        else                                // 영문모드로
                        ImmSetConversionStatus(tMode, IME_CMODE_ALPHANUMERIC,
IME_CMODE_ALPHANUMERIC);
end;

```

텔파이에서 폼을 사정없이 뜯어내는 방법의 소스

```

var
  WindowRgn,HoleRgn : HRgn;
begin
  WindowRgn := 0;
  GetWindowRgn(handle, WindowRgn);
  DeleteObject(WindowRgn);
  WindowRgn := CreateRectRgn(0,0,Width,Height);
  HoleRgn := CreateRectRgn(16,25,126,236);
  CombineRgn(WindowRgn, WindowRgn, HoleRgn, RGN_DIFF);
  SetWindowRgn(handle, WindowRgn, TRUE);
  DeleteObject(HoleRgn);
end;

```

텔파이에서의 키값

아래에 가상키 값 리스트입니다....

```

vk_LButton   = $01;
vk_RButton   = $02;
vk_Cancel    = $03;
vk_MButton   = $04;  { NOT contiguous with L & RBUTTON }
vk_Back      = $08;
vk_Tab       = $09;
vk_Clear     = $0C;

```

```

vk_Return      = $0D;
vk_Shift       = $10;
vk_Control     = $11;
vk_Menu        = $12;
vk_Pause       = $13;
vk_Capital     = $14;
vk_Escape      = $1B;
vk_Space       = $20;
vk_Prior       = $21;
vk_Next        = $22;

vk_End         = $23;
vk_Home        = $24;
vk_Left        = $25;
vk_Up          = $26;
vk_Right       = $27;
vk_Down        = $28;
vk_Select      = $29;
vk_Print       = $2A;
vk_Execute     = $2B;
vk_SnapShot    = $2C;
{ vk_Copy      = $2C not used by keyboards }
vk_Insert      = $2D;
vk_Delete      = $2E;
vk_Help        = $2F;
{ vk_A thru vk_Z are the same as their ASCII equivalents: 'A' thru 'Z' }
{ vk_0 thru vk_9 are the same as their ASCII equivalents: '0' thru '9' }

vk_Numpad0     = $60;
vk_Numpad1     = $61;
vk_Numpad2     = $62;
vk_Numpad3     = $63;
vk_Numpad4     = $64;
vk_Numpad5     = $65;
vk_Numpad6     = $66;
vk_Numpad7     = $67;

```

vk\_Numpad8 = \$68;  
vk\_Numpad9 = \$69;  
vk\_Multiply = \$6A;  
vk\_Add = \$6B;  
vk\_Separator = \$6C;  
vk\_Subtract = \$6D;  
vk\_Decimal = \$6E;  
vk\_Divide = \$6F;  
vk\_F1 = \$70;  
vk\_F2 = \$71;  
vk\_F3 = \$72;  
vk\_F4 = \$73;  
vk\_F5 = \$74;  
  
vk\_F6 = \$75;  
vk\_F7 = \$76;  
vk\_F8 = \$77;  
vk\_F9 = \$78;  
vk\_F10 = \$79;  
vk\_F11 = \$7A;  
vk\_F12 = \$7B;  
vk\_F13 = \$7C;  
vk\_F14 = \$7D;  
vk\_F15 = \$7E;  
vk\_F16 = \$7F;  
vk\_F17 = \$80;  
vk\_F18 = \$81;  
vk\_F19 = \$82;  
vk\_F20 = \$83;  
vk\_F21 = \$84;  
vk\_F22 = \$85;  
vk\_F23 = \$86;  
vk\_F24 = \$87;  
vk\_NumLock = \$90;  
vk\_Scroll = \$91;

## 디렉토리에 관련된 함수

```
function GetCurrentDir: string; // 현재의 Directory
function ExtractFileDir(const FileName: string): string;
// Directory 만 Return .Filename 빼고
function ExtractFileName(const FileName: string): string;
// 파일 이름만 Return
```

## 동작중인 프로그램 죽이기

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, TlHelp32;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    B_Search: TButton;
    B_Terminate: TButton;
    procedure B_SearchClick(Sender: TObject);
    procedure B_TerminateClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
```

```

// kernel32.dll 을 사용하여 현재 떠있는 process 를 읽어온다
procedure Process32List(Slist: TStrings);
var
    Process32: TProcessEntry32;
    SHandle:   THandle; // the handle of the Windows object
    Next:      BOOL;
begin
    Process32.dwSize := SizeOf(TProcessEntry32);
    SHandle           := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if Process32First(SHandle, Process32) then
    begin
        // 실행파일명과 process object 저장
        Slist.AddObject(Process32.szExeFile, TObject(Process32.th32ProcessID));
        repeat
            Next := Process32Next(SHandle, Process32);
            if Next then
                Slist.AddObject(Process32.szExeFile, TObject(Process32.th32ProcessID));
            until not Next;
        end;
        CloseHandle(SHandle); // closes an open object handle
    end;

procedure TForm1.B_SearchClick(Sender: TObject);
begin
    // 현재 실행중인 process 를 검색
    ListBox1.Items.Clear;
    Process32List(ListBox1.Items);
end;

procedure TForm1.B_TerminateClick(Sender: TObject);
var
    hProcess: THandle;
    ProcId:   DWORD;
    TermSucc: BOOL;

```

```

begin
    // 현재 실행중인 process 를 kill
    if ListBox1.ItemIndex < 0 then System.Exit;
    ProcId := DWORD(ListBox1.Items.Objects[ListBox1.ItemIndex]);
    // 존재하는 process object 의 handle 을 return 한다
    hProcess := OpenProcess(PROCESS_ALL_ACCESS, TRUE, ProcId);
    if hProcess = NULL then
        ShowMessage('OpenProcess error !');
    // 명시한 process 를 강제 종료시킨다
    TermSucc := TerminateProcess(hProcess, 0);
    if TermSucc = FALSE then
        ShowMessage('TerminateProcess error !')
    else
        ShowMessage(Format('Process# %x terminated successfully !', [ProcId]));
end;

end.

```

### 레지스트리를 이용한 모뎀찾기

```

WRegistry := TRegistry.Create;
with WRegistry do
begin
    rootkey := HKEY_LOCAL_MACHINE;
    if OpenKey
        ('\System\CurrentControlSet\Services\Class\Modem\0000',False) then
        Showmessage ('모뎀이 있습니다. ');
    ...
    free..
end;

```

### 마우스의 Enter/Exit Event 사용하기

```

TForm1 = class(TForm)
    Image1 : TImage;
private

```



```

    m_orgProc    : TWndMethod;
    procedure    ImageProc ( var Msg : TMessage ) ;
public
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
end;
    :
    :
procedure TForm1.FormCreate(Sender:TObject);
begin
    m_orgProc := Image1.WindowProc;
    Image1.WindowProc := ImageProc;
end;

procedure TForm1.FormDestroy(Sender:TObject);
begin
    Image1.WindowProc := m_orgProc;
end;

procedure TForm1.ImageProc( var Msg : TMessage );
begin
    case Msg.Msg of
        CM_MOUSELEAVE:
            begin
                // 여기서 컨트롤에 마우스가 들어왔을 때를 처리합니다.
            end;
        CM_MOUSEENTER:
            begin
                // 여기서 컨트롤로부터 마우스가 벗어날때 부분을 처리합니다.
            end;
    end;
    m_orgProc(Msg);
end;

end;

```

## 마우스의 범위 제한하기

다음 예제는 폼에 2 개의 버튼을 두고 첫번째 버튼을 누르면 마우스가 폼 밖으로 못나가게 하고, 두번째 버튼을 누르면 원래대로 바꿔주는 프로그램입니다...

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Rect : TRect;
begin
    Rect := BoundsRect;
    InflateRect(Rect, 0, 0);
    ClipCursor(@Rect);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    ClipCursor(nil);
end;
```

## Message 박스에 두줄출력하기

```
MessageDlg('문자열' + chr(13) + '문자열', mtInformation, [mbOK], 0);
```

참고 : 윈도우에서는 3 줄까지 가능함. 3 줄 이상의 문자열은 자동으로 정렬하지 않으니 개발자가 주의해야 함.

## 바탕화면 바꾸기

```
GetMem( ThePChar , 255 );
StrPCopy( ThePChar , 'wallpaper.bmp');
SystemParametersInfo( SPI_SETDESKWALLPAPER , 0 ,
                                                                ThePChar , SPIF_SENDWININICHANGE );
Freemem( ThePChar , 255 );
```

## 브라우저 동작하기

UrlMon 유닛으로 선언되고 있다 HlinkNavigateString Win32 API 을(를) 씁니다.

호출 예 :

```
HlinkNavigateString(nil,'http://www.borland.co.jp/');
```

만약 액티브 폼의 중(안)에서 불러내고 싶은 경우에는 이하와 같이 지정합니다 :

```
HlinkNavigateString(ComObject,'http://www.borland.co.jp/');
```

ShellApi 유닛으로 선언되고 있다 ShellExecute 을(를) 쓰는 것도 가능합니다.

```
ShellExecute(0, 'open', 'http://www.borland.co.jp/', nil, nil, SW_SHOW)
```

## 사용자가 조합키를 누른것처럼 처리하는 방법

다음 소스를 참고하기 바랍니다. 중요한 부분은 조합키중 키와 키, 키와 같이 홀드(hold) 상태인 키를 확인해서 키값을 포스팅해 주는 것입니다.

완전하다면 더할나위 없이 좋겠지만, 그냥 자신의 프로그램에 덧붙여 사용하거나 외부 참조로 사용해도 무방할 것입니다.

```
procedure PostKeyEx( hWindow: HWND; key: Word; Const shift: TShiftState; Specialkey: Boolean );
```

```
type
```

```
  TBuffers = Array [0..1] of TKeyboardState;
```

```
var
```

```
  pKeyBuffers : ^TBuffers;
```

```
  lparam: LongInt;
```

```
begin
```

```
  if IsWindow( hWindow ) then
```

```
    begin
```

```
      pKeyBuffers := nil;
```

```
      lparam := MakeLong( 0, MapVirtualKey( key, 0 ) );
```

```
      if Specialkey then
```

```

    lparam := lparam or $1000000;
New( pKeyBuffers );
try
    GetKeyboardState( pKeyBuffers^[1] );
    FillChar( pKeyBuffers^[0],Sizeof( TKeyboardState ), 0 );
    if ssShift In shift then
        pKeyBuffers^[0][VK_SHIFT] := $80;
    if ssAlt In shift then
        begin
            pKeyBuffers^[0][VK_MENU] := $80;
            lparam := lparam or $20000000;
        end;
    if ssCtrl in shift then
        pKeyBuffers^[0][VK_CONTROL] := $80;
    if ssLeft in shift then
        pKeyBuffers^[0][VK_LBUTTON] := $80;
    If ssRight in shift then
        pKeyBuffers^[0][VK_RBUTTON] := $80;
    if ssMiddle in shift then
        pKeyBuffers^[0][VK_MBUTTON] := $80;

    SetKeyboardState( pKeyBuffers^[0] );

    if ssAlt in shift then
        begin
            PostMessage( hWindow, WM_SYSKEYDOWN, key, lparam);
            PostMessage( hWindow, WM_SYSKEYUP, key, lparam or $C0000000);
        end
    else
        begin
            PostMessage( hWindow, WM_KEYDOWN, key, lparam);
            PostMessage( hWindow, WM_KEYUP, key, lparam or $C0000000);
        end;

    Application.ProcessMessages;

```

```

        SetKeyboardState( pKeyBuffers^[1] );
    finally
        if pKeyBuffers <> nil then
            Dispose( pKeyBuffers );
        end;
    end;
end; { PostKeyEx }

```

```

procedure TForm1.SpeedButton2Click(Sender: TObject);

```

```

Var

```

```

    W: HWND;

```

```

begin

```

```

    W := Memo1.Handle;

```

```

    PostKeyEx( W, VK_END, [ssCtrl, ssShift], False );

```

```

    // 전체 선택

```

```

    PostKeyEx( W, Ord('C'), [ssCtrl], False );

```

```

    // 클립보드로 복사

```

```

    PostKeyEx( W, Ord('C'), [ssShift], False );

```

```

    // "C"로 치환

```

```

    PostKeyEx( W, VK_RETURN, [], False );

```

```

    // 엔터키(새라인)

```

```

    PostKeyEx( W, VK_END, [], False );

```

```

    // 라인의 끝으로

```

```

    PostKeyEx( W, Ord('V'), [ssCtrl], False );

```

```

    // 붙여넣기

```

```

end;

```

## 시스템 About 사용하기

```

ShellAbout(Self.Handle,

```

```

    PChar(Application.Title),

```

```

    'http://home.t-online.de/home/mirbir.st/#13#10'mailto:mirbir.st@t-online.de',

```

```

    Application.Icon.Handle);

```

Self.Handle 은 현재 동작중인 Application 의 실행영역을 리턴하는 것이고....

PChar( Application.Title )은 Title 의 Caption 을 전달하는 것..

'문서영역'은 이 곳에서 만들었다는 표시...

Application.Icon.Handle 은 About 에서 보일 Icon 의 값을 전달하는 방법

### 시스템 Image 를 사용하는 TListView

```
procedure TDirTreeView.FindAllSubDirectories(pNode: TCTreeNode; ItsTheFirstPass: Boolean);
```

```
var
```

```
    srch: TSearchRec;
```

```
    DOSerr: integer;
```

```
    NewText: String;
```

```
    NewPath: string;
```

```
    tNode: TCTreeNode;
```

```
    cNode: TCTreeNode;
```

```
    ImagesHandleNeeded : boolean;
```

```
    cCursor: HCursor;
```

```
    NewList: TStringList;
```

```
    i: integer;
```

```
    tpath: string;
```

```
function TheImage(FileID: string; Flags: DWord; IconNeeded: Boolean): Integer;
```

```
var
```

```
    SHFileInfo: TSHFileInfo;
```

```
begin
```

```
    Result := SHGetFileInfo(pchar(FileID), 0,
```

```
                           SHFileInfo,    SizeOf(SHFileInfo),
```

```
                           Flags);
```

```
    if IconNeeded then
```

```
        Result := SHFileInfo.iIcon;
```

```
end;
```

```
function ItHasChildren(const fPath: string): Boolean;
```

```
var
```

```
    srch: TSearchrec;
```

```
    found: boolean;
```

```
    DOSerr: integer;
```

```

begin
  chdir(fPath);
  Found := false;
  DOSerr := FindFirst('*.*',faDirectory,srch);
  while (DOSerr=0) and not(Found) do
    begin
      found := ((srch.attr and faDirectory)=faDirectory)
        and ((srch.name<>'.' )
        and (srch.name<>'..'));

      if not(found) then
        DOSerr := FindNext(srch);
      end;
      sysutils.FindClose(srch);
      chdir('..');
      Result := Found;
    end;
  end;
end;

```

```

begin
  tNode := TopItem;
  cCursor := Screen.cursor;
  Screen.cursor := crHourGlass;
  Items.BeginUpdate;
  SortType := stNone;
  tpath := uppercase(fCurrentPath);
  NewList := TStringList.Create;
  getdir(0,NewPath);
  if (NewPath[length(NewPath)]<>'\'') then
    NewPath := NewPath + '\';
  ImagesHandleNeeded := ItsTheFirstPass;
  DOSerr := FindFirst('*.*',faDirectory,srch);
  while DOSerr=0 do
    begin
      if ((srch.attr and faDirectory)=faDirectory) and
        ((srch.name<>'.' ) and (srch.name<>'..')) then
        begin
          NewText := lowercase(srch.name);

```

```

        NewText[1] := Ucase(NewText[1]);
        NewList.AddObject(NewText, pointer(NewStr(NewPath+NewText)));
    end;
    DOSerr := FindNext(srch);
end;
sysutils.FindClose(srch);

NewList.Sorted := true;
with NewList do
for i := 0 to Count-1 do
begin
    cNode := Items.AddChildObject(pNode,Strings[i], PString(Objects[i]));
    with cNode do
    begin
        NewText := PString(Data)^;
        HasChildren := ItHasChildren(NewText);
        if ImagesHandleNeeded then
        begin
            Images.Handle      :=      TheImage(NewText,      SHGFI_SYSICONINDEX      or
SHGFI_SMALLICON, false);
            ImagesHandleNeeded := false;
        end;
        ImageIndex := TheImage(NewText, SHGFI_SYSICONINDEX or SHGFI_SMALLICON,
true);
        SelectedIndex := TheImage(NewText, SHGFI_SYSICONINDEX or SHGFI_SMALLICON or
SHGFI_OPENICON, True);
        if AnsiCompareText(NewText,fCurrentPath)=0 then
        begin
            Expanded := true;
            StateIndex := SelectedIndex;
            Self.Selected := cNode;
        end
    else
        if (pos(uppercase(NewText),tPath)=1) then
        begin
            Expanded := true;

```



```

        tNode := cNode;
    end;
end;
end;
NewList.Free;
Items.EndUpdate;
if Assigned(tNode) then
    TopItem := tNode;
    Screen.cursor := cCursor;
end;

```

## 실행하기

```

function fileExec(const aCmdLine: String; aHide, aWait: Boolean): Boolean;
var
    StartupInfo : TStartupInfo;
    ProcessInfo : TProcessInformation;
begin
    {setup the startup information for the application }
    FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
    with StartupInfo do
    begin
        cb:= SizeOf(TStartupInfo);
        dwFlags:= STARTF_USESHOWWINDOW or STARTF_FORCEONFEEDBACK;
        if aHide then wShowWindow:= SW_HIDE
            else wShowWindow:= SW_SHOWNORMAL;
    end;

    Result := CreateProcess(nil,PChar(aCmdLine), nil, nil, False,
        NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo);
    if aWait then
        if Result then
            begin
                WaitForInputIdle(ProcessInfo.hProcess, INFINITE);
                WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
            end;
        end;
    end;

```

end;

function fileRedirectExec(const aCmdLine: String; Strings: TStrings): Boolean;

var

StartupInfo : TStartupInfo;

ProcessInfo : TProcessInformation;

aOutput : Integer;

aFile : String;

begin

Strings.Clear;

{ Create temp. file for output }

aFile:=FileTemp('.tmp');

aOutput:=FileCreate(aFile);

try

{setup the startup information for the application }

FillChar(StartupInfo, SizeOf(TStartupInfo), 0);

with StartupInfo do

begin

cb:= SizeOf(TStartupInfo);

dwFlags:= STARTF\_USESHOWWINDOW or STARTF\_FORCEONFEEDBACK or  
STARTF\_USESTDHANDLES;

wShowWindow:= SW\_HIDE;

hStdInput:= INVALID\_HANDLE\_VALUE;

hStdOutput:= aOutput;

hStdError:= INVALID\_HANDLE\_VALUE;

end;

Result := CreateProcess(nil,PChar(aCmdLine), nil, nil, False,

NORMAL\_PRIORITY\_CLASS, nil, nil, StartupInfo, ProcessInfo);

if Result then

begin

WaitForInputIdle(ProcessInfo.hProcess, INFINITE);

WaitForSingleObject(ProcessInfo.hProcess, INFINITE);

end;

finally

```

    FileClose(aOutput);
    Strings.LoadFromFile(aFile);
    DeleteFile(aFile);
end;
end;

```

## 외부 Application 의 Window 크기 조절하기

SHOWWINDOW-외부 Application 의 Window 크기 조절

아래 소스는 현재 active 된 window 의 list 를 구한 후 그중 하나를 선택하여 Minimized, Maximized 하는 예제입니다.

```

procedure GetAllWindowsProc(WinHandle: HWND; Slist: TStrings);
var
    P: array[0..256] of Char; {title bar 를 저장 할 buffer}
begin
    P[0] := #0;
    GetWindowText(WinHandle, P, 255); {window's title bar 를 알아낸다}
    if (P[0] <> #0) then
        if IsWindowVisible(WinHandle) then {invisible 한 window 는 제외}
            Slist.AddObject(P, TObject(WinHandle)); {window 의 handle 저장}
        end;
    end;

```

```

procedure GetAllWindows(Slist: TStrings);
var
    WinHandle: HWND;
Begin
    WinHandle := FindWindow(nil, nil);
    GetAllWindowsProc(WinHandle, Slist);
    while (WinHandle <> 0) do {Top level 의 window 부터 순차적으로 handle 을 구한다}
        begin
            WinHandle := GetWindow(WinHandle, GW_HWNDNEXT);
            GetAllWindowsProc(WinHandle, Slist);
        end;
    end;
end;

```

```
procedure TForm1.B_SearchClick(Sender: TObject);
```

```
begin
```

```
    ListBox1.Items.Clear;
```

```
    GetAllWindows(ListBox1.Items);
```

```
end;
```

```
procedure TForm1.B_MaximizeClick(Sender: TObject);
```

```
begin
```

```
    if ListBox1.ItemIndex < 0 then
```

```
        System.Exit;
```

```
    { 선택한 window 를 maximize }
```

```
    ShowWindow(HWND(ListBox1.Items.Objects[ListBox1.ItemIndex]), SW_MAXIMIZE);
```

```
end;
```

```
procedure TForm1.B_minimizeClick(Sender: TObject);
```

```
begin
```

```
    if ListBox1.ItemIndex < 0 then
```

```
        System.Exit;
```

```
    { 선택한 window 를 minimize }
```

```
    ShowWindow(HWND(ListBox1.Items.Objects[ListBox1.ItemIndex]), SW_MINIMIZE);
```

```
end;
```

워크그룹의 호스트네임 읽어내기

```
program ShowSelf;
```

```
{ $apptype console }
```

```
uses    Windows, Winsock, SysUtils;
```

```
function HostIPFromHostEnt( const HostEnt: PHostEnt ): String;
```

```
begin
```

```
    Assert( HostEnt <> nil );
```

```
    // first four bytes are the host address
```

```
    Result := Format( '%d.%d.%d.%d', [Byte(HostEnt^.h_addr^[0]), Byte(HostEnt^.h_addr^[1]),
```

```
        Byte(HostEnt^.h_addr^[2]), Byte(HostEnt^.h_addr^[3])] );
```

```

end;

var
  r: Integer;
  WSAData: TWSAData;
  HostName: array[0..255] of Char;
  HostEnt: PHostEnt;
begin
  // initialize winsock
  r := WSASStartup( MakeLong( 1, 1 ), WSAData );
  if r <> 0 then
    RaiseLastWin32Error;
  try
    Writeln( 'Initialized winsock successfully...' );

    // get the host name (this is the current machine)
    FillChar( HostName, sizeof(HostName), #0 );
    r := gethostname( HostName, sizeof(HostName) );
    if r <> 0 then
      RaiseLastWin32Error;
    Writeln( 'Host name is ', HostName );

    // get host entry (address is contained within)
    HostEnt := gethostbyname( HostName );
    if not Assigned(HostEnt) then
      RaiseLastWin32Error;
    Writeln( 'Got host info...' );

    // dump out the host ip address
    Writeln( 'Host address: ', HostIPFromHostEnt( HostEnt ) );
  finally
    WSACleanup;
  end;
end.

```

윈도우시작메뉴 히스트로에 문서 등록하기

윈도우즈 시작메뉴에 있는 문서 히스토리에 자기가 생성한  
화일을 등록할 수 있는 함수가 있습니다.

먼저 다음과 같은 프로시저를 프로그램에 넣어 주세요.

```
use ShellAPI, ShlObj;
```

```
procedure AddToStartDocument(FilePath: string)
begin
    SHAddToRecentDocs(SHARD_PATH, PChar(FilePath));
end;
```

자 이제 이 함수를 사용해 봅시다. 우린 파라미터로 문서의  
경로를 넘겨주면 됩니다.

예)

```
AddToStartDocument(C:\Test.txt);
=>책에 이렇게 나와 있는데, 미스 프린팅 같군요.
-> 요렇게 해 주세요. AddToStartDocument('C:\Test.txt');
```

## 윈도우 배경그림바꾸기

Window 배경그림 바꾸기

```
procedure ChangeIt;
var
    Reg: TRegIniFile;
begin
    Reg := TRegIniFile.Create('Control Panel');
    Reg.WriteString('desktop','Wallpaper','c:\windows\kim.bmp');
    Reg.WriteString('desktop','TileWallpaper','1');
    Reg.Free;
    SystemParametersInfo(SPI_SETDESKWALLPAPER,0,nil,SPIF_SENDWININICHANGE);
end;
```

## Status 에 색깔 넣기

Status bar 에 색깔 넣기

StatusBar Font 의 색을 바꾸는 방법은 직접 그려주는 수 밖에 없습니다. 익히 아시겠지만 StatusBar 의 Item 이라 할 수 있는 TStatusPanel 에는 Style 이란게 있습니다. 이 값은 psText 나 psOwnerDraw 란 값을 갖는데 psOwnerDraw 일때에는 해당 Panel 을 그릴 때마다 OnDrawPanel event 가 호출됩니다. 이때에 원하는 색으로 직접 그려주시면 됩니다. psOwnerDraw 일때는 그려주지 않게되면 Text 값을 갖고 있다 하더라도 전혀 나오질 않으므로, 반드시 위에 말한 event 에서 그려주셔야 합니다.

다음에 예제를 보여드립니다.

```
procedure TfmMain.m_statusBarDrawPanel(StatusBar:
  TStatusBar; Panel: TStatusPanel; const Rect: TRect);
begin
  with StatusBar.Canvas do begin
    case Panel.ID of
      0 : Font.Color := clBlue;
      2 : if Panel.Text = '한글' then Font.Color := clRed
          else Font.Color := clBlue;
    end;
    FillRect(Rect);
    TextOut(Rect.Left+2,Rect.Top+2,Panel.Text);
  end;
end;
```

위에 ID 란 property 를 사용했는데요, 이것은 index 와는 약간 차이가 있습니다. index property 와 같이 부여되긴 하지만, item 이 추가, 삭제, 삽입되더라도 ID 의 값은 변하지 않습니다.

다시말해 한번 부여된 ID 는 다시 사용되지 않습니다.

## TreeView 프린트하기

TreeView and Print

paintTo can be made to work, you just have to scale the printer.canvas in the ratio of screen to printer resolution.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Printer.BeginDoc;
    try
        printer.canvas.moveto(100,100);
        SetMapMode( printer.canvas.handle, MM_ANISOTROPIC );
        SetWindowExtEx(printer.canvas.handle,
                        GetDeviceCaps(canvas.handle, LOGPIXELSX),
                        GetDeviceCaps(canvas.handle, LOGPIXELSY),
                        Nil);

        SetViewportExtEx(printer.canvas.handle,
                        GetDeviceCaps(printer.canvas.handle, LOGPIXELSX),
                        GetDeviceCaps(printer.canvas.handle, LOGPIXELSY),
                        Nil);

        treeview1.PaintTo( printer.canvas.handle, 100, 100 );
    finally
        printer.enddoc;
    end;
end;
```