# 머리말

델파이가 소개된 지 어느덧 3년이라는 세월이 흘렀습니다. 그동안 많은 사람들이 델파이에 관심을 보여 주었으며 델파이 또한 그 기대에 어긋나지 않았습니다. 델파이는 C언어에 버금가는 강력한 기능을 가짐과 동시에 배우기 쉽고 편리한 환경을 통합한 이상적인 개발 툴이며 현재는 4.0 버전으로 업그레이드하여 한층 더 중무장을 하고 있습니다. 델파이는 일반 응용 프로그램 개발 툴이면서 동시에 데이터 베이스 프로그래밍까지 가능한 윈도우즈 시대의 가장 미래 지향적인 개발 툴이며 앞으로 무한한 발전을 할 것이라 확신합니다.

이 책은 델파이에 관한 모든 것을 다 담고자 노력하였으며 나름대로 델파이 학습의 체계를 세웠습니다. 전반부에는 기초적인 사용법을 순서에 맞게 배치하여 초보자가 읽기 편리하도록 구성하였으며 중반부에서는 파스칼 문법에 대해 상세하게 서술하고 있습니다. 후반부에서는 델파이의 모든 능력을 백분 발휘할 수 있는 고급 기법들에 대해 요약적으로 정리하였습니다. 특히 이 책은 프로그래밍에 처음 입문하는 사람도 쉽게 읽을 수 있도록 내용 배치에 많은 정성을 들여 최대한 쉽게 읽을 수 있도록 하였으며 엄격한 베타 테스트와 많은 사람들의 의견을 수렴하여 내용의 정확성을 기하도록 했습니다. 이 책으로 인해 단 한 사람이라도 좀 더 델파이에 쉽게 다가갈 수 있기를 기대합니다.

지금 우리는 경제적으로 어려운 시기에 있으며 우리의 소프트웨어 산업은 최대의 시련기를 맞고 있습니다. 이 어려운 시련을 극복하는 가장 빠른 지름길은 국민 각자가 자신의 위치에서 자신의 맡은 바에 최선을 다하는 것이라 생각합니다. 프로그래머인 우리들은 부단한 자기 계발과 창조로 소프트웨어 생산 2위국의 명예를 계속 지키며 더 나아가 소프트웨어 선진국으로 도약하는데 기여해야겠습니다.

끝으로 이 책이 나오기까지 끝까지 원고를 꼼꼼히 감수해 준 후배 승우군, 많은 도움을 준 아우 상영군 그리고 여러모로 부족한 책을 기꺼이 출판해 주신 가남사 여러분께 진심으로 감사드립니다.

김 상형

머리말 뒷면은 백지로 남겨 두십시요.

# 차례

## 제 1 장 델파이 소개



## 제 2 장 델파이의 기본

# 제 3 장 컴포넌트



# 제 4 장 폼

## 제 5 장 메뉴

## 제 6 장 파스칼 기본 문법

## 제 7 장 컴포넌트 활용



## 제 8 장 그래픽

## 제 12 장 고급 프로그래밍

## 제 16 장 컴포넌트 제작

## 부록

# 이 책을 읽기 전에

## 1. 독자 선정

이 책은 다음과 같은 독자들을 대상으로 하여 쓰여졌습니다. 자신이 다음의 조건에 적합하지 않다고 생각되시는 분은 먼저 준비를 하고 오시기 바랍니다.

❶ 윈도우즈를 사용해 본 경험이 있어야 합니다.

델파이는 윈도우즈에서 실행되는 개발 툴입니다. 따라서 윈도우즈에 관한 기본적인 조작 방법을 알고 있어야 하며 윈도우즈용 프로그램을 한 두개 정도는 쓸 줄 알아야 합니다. 최소한 클릭이 뭔지, 드래그는 어떻게 하는지는 알아야 하며 파일, 디렉토리 정도는 마음대로 관리할 수 있어야 합니다. 윈도우즈는 반드시 윈도우즈 95나 98 또는 NT를 사용해야 하나 만약 델파이 1.0으로 공부하실 분은 윈도우즈 3.1을 사용하셔도 무방합니다. 시중에 나와있는 윈도우즈 입문서를 참고하시기 바랍니다.

❷ 프로그래밍에 관해서는 기본적인 사항만 알고 있으면 됩니다.

이 책은 프로그래밍의 초보자를 대상으로 문법의 기초부터 설명하고 있으므로 윈도우즈 환경에서 프로그래밍을 한번도 해 보지 않은 사람도 무리없이 읽을 수 있습니다. 그러나 여러가지 이유로 지나치게 상세한 설명은 하지 않고 있으므로 좀 더 기초부터 공부를 하고 싶은 사람은 기초 프로그래밍 서적을 참고하시기 바랍니다. 델파이용 기초 문법 서적보다는 도스용 베이직이나 터보 파스칼이 적당합니다. 부록으로 포함된 CD에는 파스칼 초보자를 위한 매뉴얼이 포함되어 있습니다.

❸ 컴퓨터와 델파이가 반드시 있어야 합니다.

아무리 머리가 좋아도 컴퓨터없이 델파이를 배울 수는 없으며 델파이도 반드시 있어야 실습을 겸할 수 있습니다. 컴퓨터 기종은 최소 486, 권장되는 바는 펜티엄이며 메모리는 16M 정도면 일단은 됩니다. 델파이는 속도가 무척 빠르므로 펜티엄 정도면 충분하며 그 이상의 시스템을 구비할 필요까지는 없습니다.

# 2. 이 책을 읽는 법

이 책의 모든 예제는 배포 CD에 컴파일 가능한 상태로 보관되어 있습니다. 본문 옆에 다음과 같은 아이콘이 있을 경우는 예제가 작성되어 있다는 뜻입니다.


2jang
fruit

이 아이콘의 의미는 DelEx/2jang/fruit 디렉토리에 fruit.dpr 예제를 참고하라는 뜻입니다. 실행 파일도 같이 배포되므로 소스가 필요없을 경우는 실행 파일을 곧바로 실행해 보시기 바랍니다. 다음에 제시하는 순서대로 이 책을 읽으시기 바랍니다.

❶ 가장 초보에 해당하는 부분이 2,3,4장입니다. 1장은 설치 및 소개를 다루고 있으므로 실질적인 내용은 없는 셈입니다. 2,3,4장에 있는 내용은 가장 기초적인 부분이므로 꼼꼼하게 읽으셔야 합니다. 철저하게 시간적 순서에 따라 기술되어 앞에서 언급하지 않은 내용은 뒤에서 언급하지 않고 있으므로 순서대로 읽고 이해하시면 됩니다.

❷ 5장에서 9장까지는 실질적인 프로그래밍에 관해 논하고 있습니다. 특히 6장에서는 델파이의 기본적인 문법에 관해 상세히 다루고 있으므로 주의깊게 읽으시기 바랍니다. 이 부분을 읽으실 때는 부록으로 제공된 오브젝트 레퍼런스를 적절히 활용하여 입체적인 학습을 하시는 것이 좋습니다. 즉 기본 개념은 본문에서 익히되 컴포넌트의 세부적인 사항은 레퍼런스를 통해 정리하시는 것이 좋으며 만약 레퍼런스의 내용이 부족할 경우는 도움말을 참고하십시요.

❸ 10장에서 14장까지가 고급 문법과, 고급 프로그래밍 부분에 해당합니다. 앞부분과는 달리 상세한 설명보다는 요약적으로 이론을 정리하는 형식을 취하고 있습니다. 이 부분은 책을 읽어 이해하기는 힘들며 예제를 분석해 보고 직접 실습해 보아야 합니다.

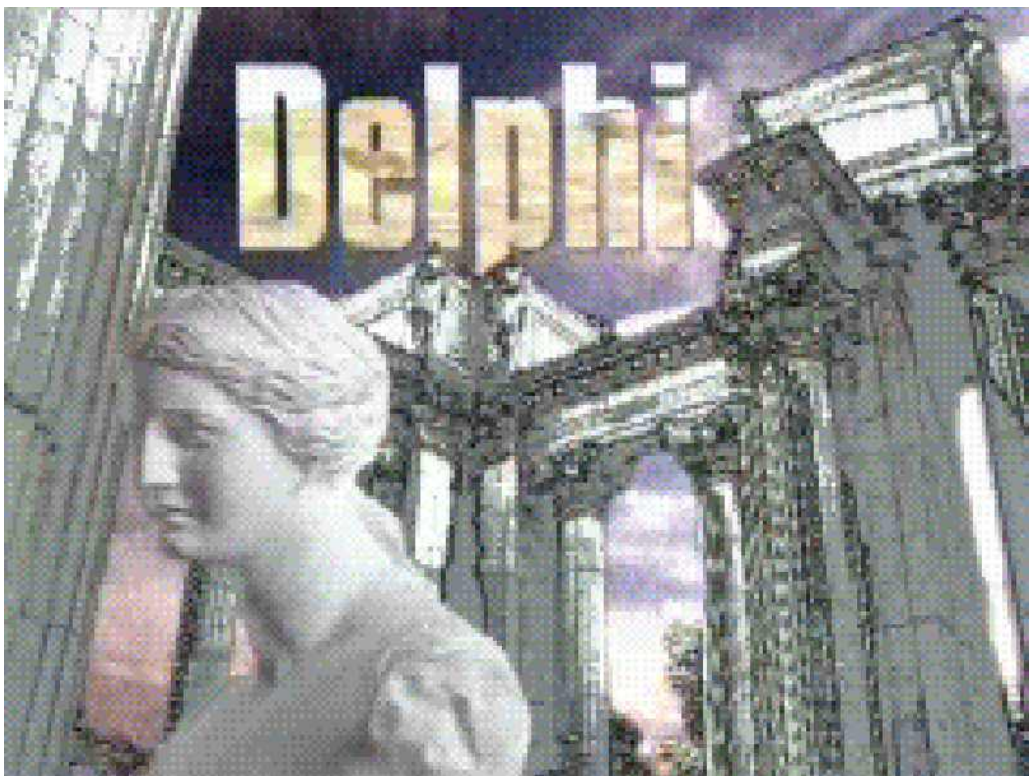
❹ 15,16장은 다소 독립적인 내용을 다루고 있습니다. 15장에서는 데이터 베이스 프로그래밍의 개념적인 면을 중심으로 설명하며 16장은 컴포넌트 제작, ActiveX에 관해 논하고 있습니다. 그러나 여기에 수록된 내용은 어디까지나 개념에 불과하므로 개념을 익히신 후 잡지나 통신망의 자료를 활용하여 더 깊은 부분은 직접 공부하셔야 합니다.

❺ 17장은 델파이의 개발 환경에 대해 설명하고 있습니다. 좀 더 편하고 고급스럽게 델파이를 사용하시고자 하는 분들은 참고하기 바랍니다. 18장에서는 팁 형식으로 여러가지 프로그래밍 기법을 논하고 있습니다. 필요할 때 찾아서 보시기 바랍니다.

❻ 책을 다 읽으시고 난 후에는 여러 가지 예제를 분석해 보시기 바랍니다. 델파이와 함께 제공되는 예제를 먼저 분석해 보신 후 통신망에서 자료를 검색하여 소스를 분석해 보시기 바랍니다. 남의 소스를 분석해 보는 과정에서 책에서 배운 이론을 확립하고 심화할 수 있을 것입니다.

# 델파이 소개

제 1 장



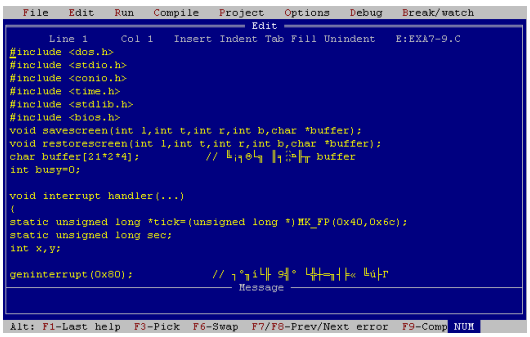
# 1-1 소개

볼랜드 델파이(Borland Delphi)가 세상에 나온지 이미 삼년이 훨씬 넘었으며 지금은 버전 4.0까지 발표되었다. 별로 긴 역사를 가진 것은 아니지만 이 짧은 시간에 많은 사람들이 델파이에 매료되었으며 델파이로 고성능의 실무 프로그램을 작성하고 있다. 과연 델파이란 무엇이며 어떤 특징을 가지고 있는지, 왜 그토록 많은 사람들이 델파이를 배우려고 하는지부터 알아 보도록 하자.

## 가. 비주얼 개발 툴

### ■ 비주얼의 의미

요즘 들어 비주얼 베이직, 비주얼 C++, 비주얼 폭스 프로 등등 부쩍 비주얼(Visual)이라는 말이 유행하고 있으며 점점 이런 비주얼 개발 툴들이 자리 굳힘을 하고 있는 추세이다. 비주얼 툴 이전의 툴을 텍스추얼(textual) 툴이라고 하며 예를 들면 터보 C 2.0이나 MS-C, 클리퍼 등을 들 수 있다. 이런 텍스트 기반의 개발 툴은 소스를 작성하면서 곧바로 결과를 확인할 수 없다는 단점을 가지고 있다.




그림

대표적인 텍스트 기반의 개발 툴인 터보 C 2.0

흔히 그래픽을 그리거나 화면을 디자인할 때 이런 현상을 쉽게 경험할 수 있는데 에디터로 소스를 작성, 저장하고 컴파일시켜 실행 파일로 만든 후 실행시켜 보아야 비로소 화면에 어떤 모양으로 나타나는가를 확인할 수 있었다. 뿐만 아니라 좌표가 잘못 되었거나 배치가 마음에 들지 않으면 마음에 들 때까지 수정, 컴파일, 실행, 확인의 과정을 반복적으로 거쳐야만 했으므로 당연히 개발 속

도도 느리고 비효율적일 수밖에 없었다.

텍스트 기반의 개발 툴은 개발 과정상의 번거로움 외에도 배우기가 어렵다는 치명적인 단점을 가지고 있다. 모든 과정이 소스 작성의 연속이므로 소스의 문법을 거의 완벽하게 이해해야만 비로소 쓸만한 프로그램을 짤 수 있다. C++ 정도의 언어를 제대로 사용하려면 정규 교육 과정을 최소한 6개월 이상 수료해야 할 정도이며 다분히 이론적이기 때문에 중도에 포기하는 사람도 많다. 어렵기도 하지만 따분하기 때문에 솔직히 배우는 재미가 없어 웬만한 의지력을 가진 사람이 아니고서는 끝까지 마스터하기 어렵다.

이런 텍스추얼 환경의 불편함과 비능률을 극복하기 위해 등장한 툴이 비주얼 개발 툴이다. 프로그램 개발 과정을 디자인 과정과 코드 작성 과정으로 분리하여 인터페이스 설계나 화면 배치 등은 컴파일하지 않고도 즉석에서 쉽게 수정할 수 있으며, 일부 코드로 처리해야 할 일도 디자인시에 마우스로 척척 처리할 수 있게 되었다. 또한 개발 과정이 편리해짐과 동시에 배우기도 쉽고 사용하기도 무척이나 용이해졌다. 비주얼(Visual)이라는 말은 “눈에 보인다”는 뜻인데 개발 과정을 시각적으로 쉽게 확인할 수 있으며 디자인시에 만든 프로그램이 실행시에도 그대로 반영된다는 뜻이다.

## ■ 등장 배경

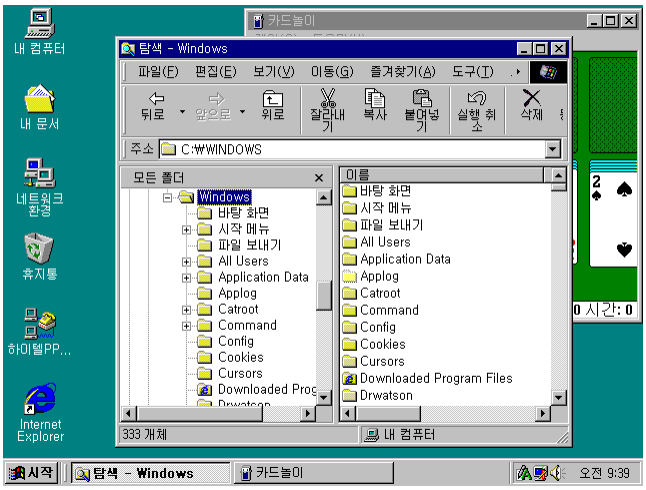
수십년간 사용되어 오던 텍스추얼 툴을 제치고 비주얼 툴이 강세를 보이게 된 이유는 무엇일까? 사실 비주얼 툴은 1960년대부터 연구가 진행되는 등 오랜 역사를 가지고 있지만 최근에 들어서야 다음과 같은 요인에 의해 갑자기 급상승하게 되었다.

GUI 는 철자 그대로 읽거나 “구이”라고 발음한다.

❶ 하드웨어의 발전에 의해 윈도우즈와 같은 GUI(Graphic User Interface) 운영체제가 대중화되었다. 모든 것을 점 단위로 섬세하게 제어할 수 있는 그래픽 기반의 운영체제는 비주얼 개발 툴이 만들어질 수 있는 물적 토대이다. 시커먼 화면에 글자들만 나열되는 도스 환경에서의 비주얼 개발 툴은 꿈과 같은 얘기다.

**그림**

대표적인 GUI OS 인 윈도우즈 98




❷ 윈도우즈의 메시지 구동 시스템은 비주얼 개발 툴의 이벤트 드리븐(Event Driven) 방식과 아주 잘 어울린다. 객체를 배치하는 일은 디자인 과정에서 하고 객체에 어떤 일이 발생할 때 해야 할 일을 코드로 작성하는 방식이어야만 여러 개의 프로그램이 조화롭게 공존할 수 있는 멀티 태스킹 환경에 적합한 프로그램을 작성할 수 있다.

❸ 하드웨어가 보잘 것 없었던 과거에는 프로그램이 만들어지는 과정보다는 만들어진 프로그램이 얼마나 작고 빠르게 실행되는가가 중요했었다. 하지만 현재는 프로그램의 크기나 속도보다는 만드는 사람이 얼마나 편하게 빨리 결과를 낼 수 있는가가 더 중요한 관건이 되었다. 격변하는 시장 상황과 치열한 경쟁에 얼마나 발빠르게 대처하는가가 생존의 요건이 되며 따라서 무엇보다도 생산성이 높은 개발 툴이 필요해진 것이다.

❹ 프로그램에 대한 사용자의 요구 사항 중 예쁜 디자인이나 깔끔한 인터페이스, 직관적이고 통일적인 사용 방법 등이 성능만큼이나 중요해지게 되었다. 이런 예쁜 프로그램을 만들기에는 텍스트 기반의 개발 툴은 아무래도 부적합하다. 비주얼 개발 툴은 한번 만들어진 코드를 재사용하는 조립식 프로그래밍 과정을 사용하므로 디자인에 유리하고 자연스럽게 일관성을 유지할 수 있다.

비주얼 툴이 처음 소개될 때만 해도 많은 사람들이 비주얼 툴의 장래에 대해 의구심을 갖기도 했었다. 실행 속도가 텍스추얼에 비해 뒤지고 몇몇 부분에서 한계가 있었기 때문이다. 그러나 이제는 비주얼 툴이 한때 세인의 입에 오르내리던 단순한 유행으로 그치지 않고 급속도로 발전과 개선을 거듭하여 프로그래밍 툴의 주류를 이루고 있다.

## ■ 비주얼 툴의 종류

현재까지 발표된 비주얼 툴은 종류를 헤아릴 수 없이 많으며 기존의 텍스추얼 툴들에도 비주얼 요소가 가미되고 있거나 아예 비주얼 툴로 바뀌고 있는 중이다. 비주얼 툴은 크게 다음 두 가지 유형으로 나눌 수 있다.

**완전 비주얼 툴**

단 한줄의 코드도 작성하지 않고 마우스로 딸깍거리기만 하면 프로그램이 완성되는 툴이다. 플로우 차트나 함수 아이콘을 연결하는 방식으로 프로그램을 작성한다. 매킨토시 기종에서는 이미 일반화 되었지만 윈도우즈 환경에서는 아직 이런 툴이 많이 보급되어 있지 않다.

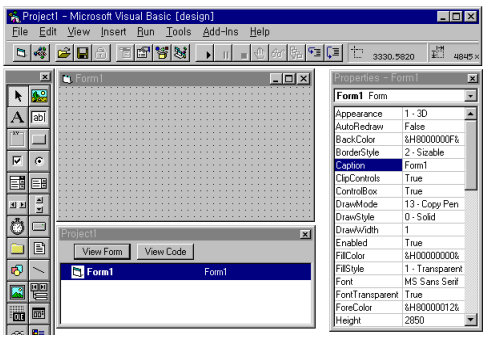
**비주얼 보조 툴**

텍스추얼 언어를 기반으로 하되 화면 설계에 비주얼 방식을 채용한 혼합형의 개발 툴이다. 프로그램 작성 결과는 텍스트 파일로 나타나며 이 소스를 컴파일하여 실행 파일을 만든다. 비주얼적인 요소가 있기는 하지만 코딩은 여전히 해야 하므로 어찌 보면 텍스추얼에 더 가깝다.

현재까지 발표된 대부분의 비주얼 개발 툴은 텍스추얼 기반의 비주얼 보조 툴이며 델파이도 여기에 속한다. 델파이의 유력한 경쟁 제품으로는 비주얼 베이직과 파워 빌더를 들 수 있으며 이 두 제품도 비주얼 보조 툴이다. 완전 비주얼 툴은 여러 가지 한계로 인해 당분간은 비주얼 보조 툴에 비해 약세를 보일 전망이다. 물론 언젠가는 코드를 전혀 몰라도 프로그램을 만들 수 있는 툴이 등장하겠지만 말이다.

**그림**

대표적인 비주얼 개발 툴인 비주얼 베이직



### ■ RAD 툴

비주얼 툴을 다른 이름으로 RAD(Rapid Application Development) 툴이라고도 한다. 문자 그대로 해석해 보면 응용 프로그램을 빨리 개발해 주는 툴이라는 뜻이다. 왜 비주얼 툴을 사용하면 개발 속도가 단축되는지는 앞으로 델파이를 직접 사용해 보면 스스로 느낄 수 있을 것이다.

## 나. 델파이

이 책에서 논할 델파이도 비주얼 개발 툴의 한 부류로서 완전 비주얼 툴은 아니며 파스칼 언어에 기반을 둔 비주얼 보조 툴이다. 델파이의 장점 내지 특징은 다음과 같다.

**완벽한 통합 개발 환경**

델파이는 소스를 번역하여 실행 파일을 만드는 단순한 컴파일러가 아니다. 에디터나 링커는 물론이고 오류를 찾아내는 디버거, 아이콘이나 이미지를 만드는 이미지 에디터, 데이터 베이스 파일을 만들 수 있는 DBD 등 프로그램 개발에 필요한 모든 툴을 내장한 통합 개발 환경(IDE:Integrated Development Environment)을 제공한다. 그래서 프로그래머는 델파이가 제공하는 쾌적한 통합 개발 환경 내에서 필요한 모든 작업을 수행할 수 있다. 오브젝트 브라우저, 메뉴 디자이너, 각종 전문가 등의 개발 도구들을 살펴보면 어지간히도 중무장을 했다는 느낌이 들 것이다.

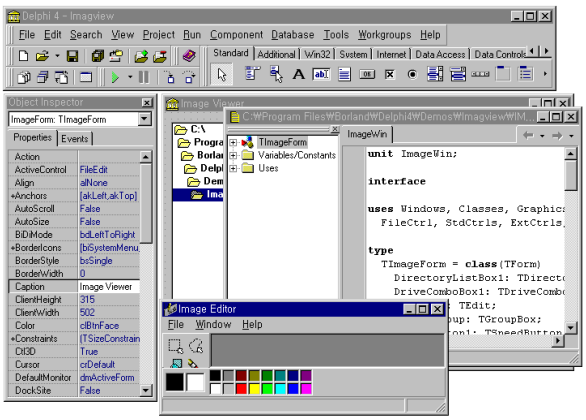



**그림**
델파이의 편리한 통합 개발 환경

**오브젝티브 파스칼 사용**

델파이는 오브젝티브 파스칼 언어를 기반으로 한다. 파스칼은 문법이 엄격하고 깔끔하여 배우기 쉽고 실행 속도가 빨라 C와 함께 가장 대중적인 언어일 뿐만 아니라 최신의 객체 지향 프로그래밍(OOP) 언어다. 경쟁 제품인 비주얼 베이직, 파워 빌더, 여타의 다른 비주얼 툴이 인터프리터나 스크립트 방식인 데 반해 델파이는 완전한 컴파일러이다. 비주얼 개발 툴은 그 특성상 실행 파일의 속도가 느릴 수밖에 없지만 델파이는 컴파일 방식을 채택함으로써 이런 단점을 완벽하게 극복하였다.

**빠른 컴파일 속도**

프로그래머에게 컴파일 시간만큼이나 지겨운 것도 없다. 델파이는 지상에서 가장 빠른 속도로 컴파일을 수행한다고 자랑할만큼 고속의 컴파일러를 내장하고 있다. 펜티엄 급의 컴퓨터에서는 초당 5천 줄 이상을 컴파일해 낸다. 참고로 5천 줄 정도의 소스면 이미 간단한 예제 수준을 넘어선 대형 프로그램이다. 이 책의 예제들은 길어야 2초 정도면 컴파일이 완료되므로 컴파일 시간을 기다려야 하는 일이 거의 없다.

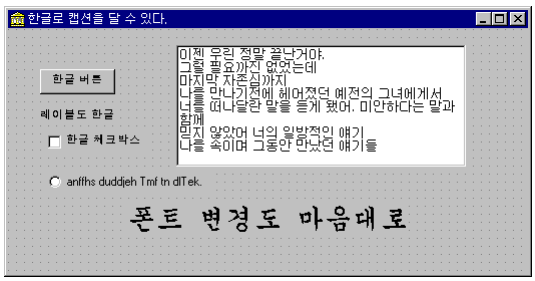
델파이의 빠른 컴파일 속도는 C++ 프로그래머가 가장 부러워하는 장점인데 실제로 C++ 컴파일러와 비교해 보면 적어도 5~10배 정도 차이가 난다. 델파이의 컴파일 속도가 이렇게 환상적으로 빠른 이유는 무엇보다 파스칼 언어의 간단명료함에 기인한다.

**한글 사용에 지장이 없다**

델파이는 미국에서 만든 미제 프로그램이지만 우리말인 한글 사용에 별 제한이 없다. 데이터 베이스 프로그래밍에서 약간의 문제가 있기는 하지만 명쾌한 해결책이 제시되고 있으므로 실질적으로는 아무런 문제가 없는 셈이다. 3.0버전부터 아시아 국가를 위한 IME 지원이 포함되어 있어 한글을 자유롭게 사용할 수 있으며 한글로 만들어진 데이터 베이스 파일도 제어할 수 있다. 그렇다고 해서 변수나 함수 이름에까지 한글을 사용할 수 있다는 얘기는 아니다.

**그림**

폼에서 한글 사용하기



**런타임 라이브러리가 불필요하다**

소스 파일을 컴파일한 결과는 하나의 완벽한 실행 파일(EXE)로 만들어진다. 별도의 DLL이나 라이브러리가 필요없기 때문에 실행 파일만 배포하면 된다. 이는 개발자의 입장에서 볼 때나 사용자의 입장에서 볼 때나 별도의 설치 과정이 없어도 되므로 무척 편리하다. 비주얼 베이직이 VBRUNxx.DLL 등의 라이브러리를 같이 배포해야 하는 단점과 대조되는 장점이다.

런타임 라이브러리를 사용하면 여러 가지 단점이 있지만 반대로 몇 가지 장점도 있다. 델파이 3.0부터는 기호에 따라 런타임 라이브러리를 사용할 수도 있다.

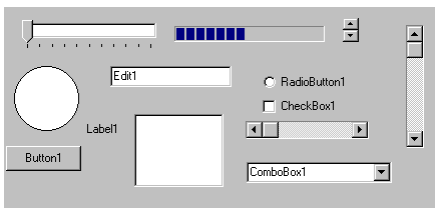
**데이터 베이스 개발을 지원한다**

게임이나 시스템 소프트웨어, 일반 응용 프로그램 외에도 데이터 베이스 프로그래밍을 지원한다. 데이터 베이스 분야는 시장이 넓어 경제적으로 유망한 분야이며 장래성이 있어 안정된 분야이기도 하다. 데이터 베이스 프로그래밍은 그 특성상 비주얼 툴로 만들기에 가장 적합한 분야이다. 델파이를 사용하는 사람들의 주된 용도도 바로 데이터 베이스 프로그램 개발 분야이다.

**조립식 컴포넌트 사용**

마치 프로그램의 부품과도 같은 컴포넌트를 사용하여, 조립하듯이 프로그램을 만든다. 이런 방식은 이미 만들어진 코드를 쉽게 재사용할 수 있다는 점에서 아주 강력할 뿐만 아니라 컴포넌트를 추가하면 개발 환경 자체를 확장할 수 있어서 좋다.

**그림**
여러 가지 컴포넌트




**고급과 저급을 고루 지원한다**

델파이는 초보자가 쉽게 접근할 수 있는 쉬운 개발 툴이다. 뿐만 아니라 윈도우즈 API 함수를 모두 사용할 수 있어 대형 프로젝트를 개발할 때도 C나 어셈블리 못지않은 고난도의 저급 프로그래밍을 구사할 수 있다. 마냥 쉽기만 한 것도 아니고 배우기에 따라서는 얼마든지 고급스럽게 활용할 수 있어 프로그래머를 일정한 틀 안에 가두지 않는다.

이 외에도 당장 초보자에게는 불필요할지 모르겠지만 ①고성능 클라이언트/

서버 프로그래밍 지원 ②타 운영체제로 이식 용이 ③ 데이터 베이스의 확장 용이 ④ 컴포넌트 제작 가능 ⑤ 다른 언어와 혼합 프로그래밍 가능 등의 장점들이 있다. 게다가 “컴파일러의 제 1인자”라고 칭해지는 볼랜드사의 제품이라는 점이 더욱 마음을 든든하게 한다. 볼랜드는 사운을 걸고 델파이에 자사의 모든 최신 기술을 총 집약하였으며 앞으로도 모든 지원을 아끼지 않을 것이라 생각된다.

최근에 볼랜드는 인프라이즈(Inprise)라는 이름으로 사명을 변경하였지만 델파이의 문서 여기저기에 아직도 볼랜드라는 이름이 사용되고 있으며 델파이 4의 설치 디렉토리명도 Borland로 되어 있다. 공식적인 발표에 의하면 회사명은 변경했지만 볼랜드라는 명칭은 브랜드명으로 계속 사용할 것이라고 한다. 델파이 로고 화면을 보면 정식 제품 명칭이 "Borland Delphi 4"라고 되어 있다. 필자 개인적으로 볼랜드라는 명칭이 더 마음에 들고 여러 사람들이 익숙해져 있는 이름이므로 이 책에서는 계속 볼랜드라는 회사명을 사용하기로 한다.


참고하세요

Delphi라는 이름을 처음 영문으로 본 사람은 이걸 델피라고 읽어야 하나 델파이라고 읽어야 하나 혼란스럽기도 할 것이며 Delphi가 무슨 뜻인가도 궁금할 것이다. 필자가 사전을 찾아본 결과 발음기호는 [delfai]이며 강세는 첫 음절에 있고 뜻은 "아폴로 신전이 있는 그리스의 옛 도시"라고 되어 있다. 또 인터넷의 어느 델파이 사이트에 phillaDELPHIa라고 되어 있는데 이 도시명과도 무슨 상관이 있는 듯 하기도 하다. 델파이의 원래 이름은 AppBuilder이며 코드명이 Delphi였는데 코드명이 최종 제품명이 되었다.

## 다. 4.0 에서의 개선점

델파이는 1995 년 봄에 발표된 이래 세 번의 큰 버전업을 하였다. 중간에 버그 패치를 위한 자잘한 버전업도 있었다. 각 버전별 특징과 버전에 따른 추가 사항을 간단하게 정리해 보았다.

| 버전 | 특징 |
|---|---|
| 1.0 | 16비트 개발 툴<br>윈도우즈 3.1용의 프로그램도 개발할 수 있다. |
| 2.0 | 본격적인 32비트 개발 환경<br>긴 문자열 지원 |

| | |
|---|---|
| | variant타입 지원 |
| | 레지스터 호출 관행 추가 |
| | finitialization 섹션 추가 |
| | 오브젝트 창고 |
| 3.0 | ActiveX 컨트롤 사용 및 개발 지원 |
| | 코드 인사이트 |
| | 코드 에디터의 거터 지원 |
| | 멀티 타이어 어플리케이션 개발 지원 |
| | 컴포넌트 템플릿 |
| | 패키지 지원 |
| | IME 지원 |
| | BDE 기능 향상 |
| 4.0 | 아래 참조 |

이 중 델파이 1.0 버전을 제외하고는 모두 32 비트 프로그램을 만드는 툴이다. 그래서 윈도우즈 3.1 버전을 타겟으로 하는 제품은 반드시 1.0 버전을 사용해야 한다. 델파이 1.0 버전은 델파이 CD-ROM 에 같이 포함되어 있으므로 윈도우즈 3.1 용 프로그램을 작성하려는 사람은 1.0 을 설치한 후 사용하면 된다.

버전이 올라감에 따라 컴포넌트의 개수나 지원 가능한 데이터 베이스 드라이버의 수는 꾸준히 증가하고 있다. 현재 최신 버전인 4.0 은 이전 버전에 비해 다음과 같은 여러 가지 사항이 개선, 추가되었다. 델파이를 처음 배우는 사람은 이 개선 사항이 당장 무엇을 의미하는지 몰라도 상관없다. 아래 내용은 본문에서 다시 자세하게 언급할 것이므로 대충 목록만 파악하기 바라며 기존에 델파이 3 를 쓰던 사람들은 어떤 기능이 추가되었는지 구경만 해두기 바란다.

## ■ 동적 배열

같은 타입의 변수 여러 개를 모아 놓는 배열은 선언시에 그 크기를 밝혀주어야 한다. 다음은 1~10 까지의 첨자를 가지는 크기 10 의 정수형 배열을 선언한 것이다.

```
var
  ar: array[1..10] of Integer;
```

이렇게 선언하면 ar[1]~ar[10]까지 10 개의 변수가 생성된다. 델파이 4 에서

는 배열을 선언할 때 미리 그 크기를 밝혀주지 않고 중간에 배열 크기를 마음대로 바꿀 수 있는 동적 배열을 지원한다. 선언할 때는 array of Type 과 같이 선언하며 사용하기 전에 SetLength 함수로 배열의 크기를 지정해 주면 된다. 다음은 정수형 동적 배열을 선언, 사용한 예이다.

```
var
  ar:array of integer;
begin
  SetLength(ar, 5);
  ar[1]:=3;
  label1.Caption:=IntToStr(ar[1]);
end;
```

SetLength 함수는 동적 배열을 위해 요구한 크기만큼의 메모리를 할당해 주며 만약 배열 크기가 늘어났을 경우 메모리를 재할당해 준다. 동적 배열은 항상 0 부터 시작하며 최대 첨자는 SetLength 프로시저로 지정한 크기보다 하나 작다. 즉 SetLeng(ar, 5)로 크기를 지정하면 ar[0]~ar[4]까지 사용할 수 있다. 동적 배열은 내부적으로 포인터이며 긴 문자열과 마찬가지로 참조 회수에 의해 관리된다.

## ■ 메소드 중복 정의

메소드는 이름으로 구분되기 때문에 같은 이름의 메소드가 둘 존재할 수 없었다. 그러나 인수의 개수나 타입이 다르다면 같은 이름으로 여러 개의 메소드를 정의할 수 있는데 이를 메소드 중복 정의(Method Overloading)라고 한다. C++에서는 초기부터 이런 오버로딩을 지원했었는데 델파이는 이번 버전(Ver 4)부터 지원하기 시작했다. 메소드뿐만 아니라 전역 함수나 프로시저도 마찬가지로 중복 정의가 가능하다. 다음은 Proc 이라는 이름으로 두 개의 프로시저를 선언한 것이다. 중복 정의할 때는 메소드 뒤에 overload 라는 키워드를 적어주어야 한다.

```
procedure Proc(i:Integer);overload;
begin
end;

procedure Proc(j:Integer;k:Integer);overload
begin
end;
```

비록 프로시저의 이름이 같지만 인수의 개수가 다르기 때문에 이 두 프로시저는 서로 다른 프로시저로 인식된다. 함수 호출문에서는 인수의 개수를 보고 어떤 함수를 호출할 것인가를 결정하게 된다. 즉 Proc(1)이라고 호출하면 위의 프로시저를 호출한 것이고 Proc(1,2)라고 호출하면 아래쪽의 프로시저를 호출한 것으로 해석된다. 인수의 개수나 타입으로 어떤 메소드를 호출했는지를 구분할 수 있기 때문에 이름이 같은 메소드를 중복해서 정의할 수 있는 것이다.

## ■ 디폴트 인수

디폴트 인수(Default Parameter)란 함수 호출문에서 인수를 생략했을 때 디폴트로 적용되는 인수를 말한다. 함수의 인수 리스트에 "인수:타입=디폴트값" 형태로 적어주면 이 인수는 디폴트 인수가 되어 함수 호출문에서 인수를 생략할 수도 있게 된다. 다음 함수는 문자열 인수를 하나 받아들여 이 문자열로 레이블의 캡션을 변경하되 이 인수의 디폴트값이 'No Title'로 지정되어 있다.

```
procedure ChangeLabel(str:string='No Title');
begin
  Form1.label1.Caption:=str;
end;
```

ChangeLabel() 이라고 호출하면 str 인수는 디폴트값인 'No Title'이 된다. 물론 ChangeLabel('Merong'); 과 같이 인수를 전달해 주면 str 은 전달해 준 값을 대입받는다. 디폴트 인수에서 주의할 것은 디폴트 인수는 반드시 인수 리스트의 끝부분에 있어야 한다는 점이다. 다음처럼 일반 인수가 있고 디폴트 인수가 있을 때 디폴트 인수는 반드시 인수 리스트의 뒤쪽에 있어야 한다.

```
procedure ChangeLabel(i:Integer ;str:string='No Title');
procedure ChangeLabel(i:Integer ;r:real;str:string='No Title');
```

다음과 같이 디폴트 인수가 먼저 오고 일반 인수가 뒤에 와서는 안된다.

```
procedure ChangeLabel(str:string='No Title';i:Integer);
```

이렇게 되면 뒤쪽의 일반 인수 i 값을 반드시 전달해 주어야 하므로 디폴트 인수를 생략할 수 없기 때문이다. 다음처럼 뒤쪽 인수도 디폴트값을 지정해 주면 상관없다.

```
procedure ChangeLabel(str:string='No Title';i:Integer=0);
```

이렇게 하면 둘 다 디폴트 인수가 되었으므로 뒤쪽 인수부터 선택적으로 인수를 생략할 수 있다. 즉 ChangeLabel('Test', 1)로 호출하거나 뒤쪽 인수 하나만 생략하여 ChangeLabel('Test')로 호출해도 되며 둘 다 생략해서 ChangeLabel()이라고 호출할 수도 있다. 그러나 앞쪽 인수만 생략할 수 있는 방법은 없다.

## ■ 추가 데이터 타입

델파이 4 에서는 64 비트 정수형을 나타내는 int64 형이 추가되었다. 이 타입은 최대 2 의 63 승까지 표현할 수 있으며 부호가 있다. 아직까지 32 비트 이상의 정수형이 꼭 필요한 경우는 극히 드물지만 메모리 맵 파일 등 핵심적인 Win32 API 에서 64 비트 정수를 요구하며 곧 선보일 64 비트 운영체제를 지원하기 위한 준비인 듯 하다. 델파이는 64 비트 정수형을 위해 StrToInt64, StrToInt64Def 등의 함수를 추가로 제공하지만 표준 함수들은 대부분 64 비트 정수형을 인식하지 못한다.

또 부호없는 32 비트 정수형인 LongWord 타입이 추가되었는데 이 타입은 기존의 Cardinal 타입과 동일하다고 할 수 있다.

실수형 타입 중 Real 은 구형 볼랜드 파스칼과의 호환을 위해 48 비트 실수를 나타내는 타입이었으나 인텔 CPU 의 실수형이 64 비트인 관계로 Real 형의 속도가 만족스럽지 못했다. 델파이 4 에서는 Real 형을 64 비트로 확장하여 CPU 의 실수 표현과 일치시킴으로써 속도를 개선하였다. 만약 과거와의 호환성을 위해 Real 형을 48 비트로 유지해야 할 필요가 있다면 컴파일러 스위치를 사용하여 강제로 48 비트 실수로 만들 수 있다.

```
{$REALCOMPATIBILITY ON}
```

또는 Real48 이라는 별도의 데이터 타입을 사용해도 된다. 델파이 1.0 부터 Real 형은 지원되었지만 실제로 이 타입은 거의 사용되지 않았으며 앞으로도 사용하지 않는 것이 바람직하다.

## ■ 프로젝트 그룹

프로젝트는 실행 파일 하나를 만들어 내기 위한 구성 파일의 집합이다. 델파이 4 에는 프로젝트보다 한단계 더 상위의 개념으로 프로젝트 그룹이라는 것이 있다. 프로젝트 그룹에는 여러 개의 관련된 프로젝트가 포함될 수 있으므로 동

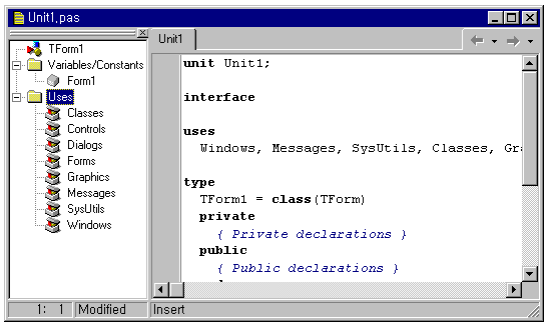
시에 여러 개의 프로젝트를 열어 놓고 작업을 할 수 있게 된다. 예를 들어 제품 하나를 만드는데 이 제품이 DLL 과 실행 파일로 구성된다면 두 개의 프로젝트를 동시에 개발해야 하며 따라서 프로젝트를 계속 번갈아가며 개발을 진행하거나 아니면 델파이를 두 번 실행해야 할 것이다.

프로젝트 그룹을 사용하면 그럴 필요없이 두 프로젝트를 한 그룹에 넣어 두고 작업을 할 수 있어 훨씬 편리하다. 비주얼 C++의 워크 스페이스 개념과 동일하다고 보면 된다.

## ■ 코드 탐색기

델파이 4 를 처음 실행시켜 보면 가장 눈에 먼저 띄는 변화가 바로 코드 탐색기이다. 코드 에디터의 왼쪽에 트리 형식의 조그만 윈도우가 붙어 있는데 이 윈도우를 코드 탐색기(Code Explorer)라고 한다.

그림
코드 에디터 옆에 나타나는 코드 탐색기




코드 탐색기에는 현재 열려진 유닛에 있는 모든 타입, 클래스, 속성, 메소드, 전역 변수 등등이 계층적으로 표시되며 신속한 이동, 검색 등의 기능과 기타 클래스 완성, 모듈 네비게이션 등의 기능이 있다. 코드 탐색기의 자세한 사용법과 기능에 대해서는 차후에 자세히 알아볼 것이다.

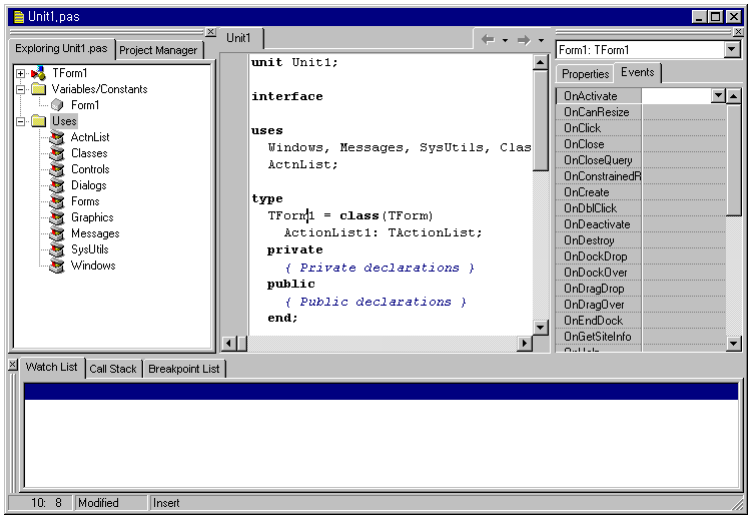
## ■ 도킹 가능한 툴 윈도우

델파이에는 개발중 또는 디버깅중에 사용할 수 있는 여러 가지 툴 윈도우들이 있다. 오브젝트 인스펙터, 정렬 팔레트, 프로젝트 관리자, 브레이크 포인터, 와치 윈도우 등등 줄잡아 열 개가 훨씬 더 넘는다. 이렇게 많은 윈도우가 열리다 보니 웬만한 고해상도 모니터에서도 정신이 없을 정도였다. 비주얼 C++의 개발 환경은 깔끔한 MDI 에 부속 윈도우들이 메인 윈도우에 도킹이 되기 때문에 윈도우를 찾아 뒤지는 불편함이 없어 항상 부러웠었다. 그런데 델파이 4 에 드디어 이 편리한 도킹 윈도우 기능이 포함되어 툴 윈도우를 여러 개 열어 놓고 사용하기가

훨씬 더 편리해졌다.

툴 윈도우끼리 도킹시키는 방법은 간단하다. 단순히 드래그해서 원하는 윈도우에 떨어뜨리기만 하면 된다. 타이틀 바의 높이가 일반 윈도우보다 조금 낮은 툴바형 윈도우들은 죄다 도킹이 된다고 생각하면 된다. 다음은 코드 에디터에 이것 저것 윈도우를 붙여 본 것이다.

**그림**
여러 개의 윈도우가 도킹된 모습




무려 일곱 개나 되는 툴 윈도우가 하나의 윈도우안에 옹기종기 모여있다. 이렇게 툴 윈도우를 도킹시켜 놓고 사용하면 화면 영역이 공평하게 분배되며 윈도우 찾아 뒤적거리지 않아도 되므로 무척 편리하다. 델파이는 기존의 도킹 윈도우에서 한단계 더 발전하여 같은 영역에 여러 윈도우를 배치하고 탭으로 윈도우를 교체하는 기법을 선보였다. 델파이는 또한 이런 도킹 가능한 윈도우를 만드는 방법도 제공한다.

## ■ 추가된 컴포넌트

버전이 바뀔 때마다 여러 개의 컴포넌트가 추가되었었는데 델파이 4 에서도 컴포넌트가 추가되어 무려 172 개나 되는 컴포넌트가 제공된다. 컴포넌트 팔레트에 새로 생긴 페이지는 Midas 페이지뿐이지만 각 페이지에도 하나씩 둘씩 컴포넌트가 추가되어 있다. 어떤 컴포넌트가 추가되어 있으며 각 컴포넌트는 어떤 용도로 사용되는 것인지는 이 책 전체를 통해 살펴볼 것이다.

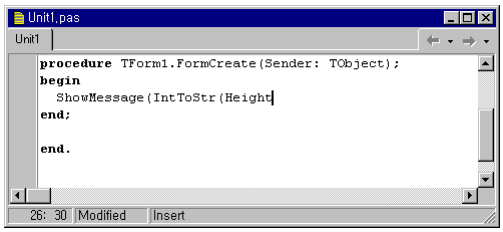
이 외에 델파이 4.0 은 ActiveX 컨트롤 개발에 대한 지원이 대폭 개선되었으

며 원격 디버깅 등 디버깅에 많은 보강이 이루어졌다. 그 외에 추가되고 개선된 기능이야 이루 말할 수 없이 많지만 이에 관해서는 본문에서 하나둘씩 살펴볼 것이다.

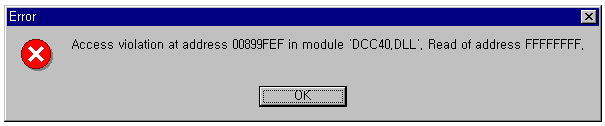
# 라. 델파이 4.01

델파이 4는 많은 부분에서 확장, 개선되었다. 자신이 사용하던 프로그램이 버전업되어 기능이 더 늘어나고 사용하기 편리해 지는 것은 누구나 바라는 일이지만 델파이 4는 늘어난 기능에 못지 않게 버그가 무척이나 많다. 윈도우즈 98에 맞추어 발표하느라 너무 서둘러서 그런지 마치 절인 생선에 덕지 덕지 붙어 있는 소금처럼 많은 버그를 가지고 있다.

대표적으로 간단한 예만 들어 보자. 새 프로젝트를 시작하고 폼을 더블클릭한 후 다음 문장을 입력해 보자.




그러면 당장 다음과 같은 에러 메시지가 나타날 것이다.




만약 자기가 가진 버전이 이런 에러를 출력한다면 버그가 있는 버전이므로 패치를 하도록 하자. www.inprise.com 에 가면 패치 버전이 공개되어 있으므로 다시 설치해 주어야 한다. 이 버그 외에도 상당히 많은 세세한 버그가 있는데 특히 데이터 베이스와 MIDAS 쪽에 버그가 많아 프로젝트에 많은 지장을 준다. 이 책이 나오고 난 후에도 패치가 발표될 지 모르므로 항상 인터넷을 주시하도록 하자.

# 1-2 설치

## 가. 설치 준비

델파이 공부를 해 보고자 마음을 먹었다면 우선 델파이를 자신의 하드 디스크에 설치해야 한다. 물론 이미 델파이가 설치된 컴퓨터를 사용하고 있다면 설치 과정은 생략해도 된다.

델파이를 자신의 시스템에 설치하기 전에 우선 어떤 준비를 해야 하는가를 알아보자. 당연히 제일 중요한 컴퓨터가 있어야 하고 최소한 다음 조건을 만족해야만 한다. 자신의 시스템이 이 조건에 만족하는지 하나씩 살펴보아라. 대체로 윈도우즈가 무리없이 돌아갈 정도면 충분하다.

① 델파이는 32비트 프로그램이므로 CPU는 최소한 386 이상이어야 한다. 권장되는 시스템은 펜티엄 100 정도 수준이지만 램만 충분하다면 486에서도 사용할 수는 있다. 빠르면 빠를수록 좋은 것이 사실이지만 델파이는 컴파일 속도가 상당히 빠른 편이므로 무리해 가면서 굳이 높은 사양의 시스템을 구비할 필요는 없을 것 같다. 필자는 펜티엄 166에 64M를 사용하고 있는데 부족하다고 생각해 본 적은 한번도 없다.

② 최소 8M 이상의 램이 있어야 한다. 8M의 용량은 윈도우즈를 윈도우즈답게 사용하기 위한 최소한의 용량이지만 사실 이 정도 램으로 윈도우즈를 구동시키는 것은 무척 어렵다. 권장되는 바는 16M 이상이지만 가급적이면 32M~64M 정도의 램을 갖추는 것이 좋다. 요즘 램값이 무척 저렴해져서 보통 이 정도의 램은 큰 무리가 아닐 것이다. 32M 정도의 용량이라면 델파이를 쓰는데 조금도 부족함이 없다.

③ 비디오 카드는 흑백은 곤란하고 최소한 VGA이상은 되어야 한다. 표준 VGA도 물론 가능하지만 800*600의 해상도를 사용할 수 있는 슈퍼 VGA가 좋다. 델파이는 여러 개의 윈도우를 동시에 열어서 사용하므로 화면의 크기가 넓을수록 좀 더 쾌적하게 작업을 할 수 있다. 640*480의 해상도에서는 답답함을 느껴야 할 것이고 800*600 정도면 그럭 저럭 쓸만하고 1024*768 정도가 가장 적당하다. 이정도 해상도를 사용하려면 17인치나 15인치의 고해상도 모니터가 필수적이다. 그러나 색상은 많을 필요가 없으며 256색 정도면

충분하다. 물론 그래픽 소프트웨어를 개발한다면 더 많은 색상이 필요할 것이다.

④ 하드 디스크는 최소 500메가 정도는 되어야 한다. 도스, 윈도우즈, 유틸리티 등은 기본으로 설치해야 할 것이고 델파이만 해도 100메가나 차지한다. 게다가 델파이로 프로그램을 만들다 보면 꽤 많은 파일이 생성되므로 웬만한 용량으로는 델파이를 실행시키기 어렵다. 권장되는 바는 1기가 이상이며 크면 클수록 좋다. 현재 120메가나 170메가의 하드 디스크를 쓰고 있다면 이번 기회에 돈이 좀 들더라도 고용량 하드 디스크를 구입하도록 하자. 하드 디스크 가격이 너무 너무 많이 내려 별로 부담이 되지도 않을 것이다.

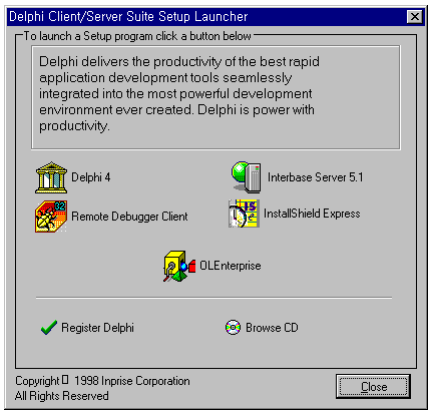
그 외에 별도의 하드웨어가 꼭 필요한 것은 없다. 단 델파이는 CD-ROM 버전으로 출시되므로 설치를 위해 CD-ROM 드라이브는 있어야 한다. 이처럼 델파이를 사용하기 위한 환경은 개발 툴 치고는 대체로 검소한 편이다. 참고로 비주얼 C++의 경우는 펜티엄 II, 64M, 설치 용량 350M 정도 되는 고급 시스템을 요구한다.

# 나. 설치

델파이는 용량이 무척 크기 때문에 CD-ROM만으로 공급되고 있다. 델파이 설치 순서는 다음과 같으며 일반적인 프로그램의 설치 순서와 다를 바가 없으므로 굳이 설명을 읽지 않아도 될 것이다. 설치 프로그램을 실행시킨 후 시키는대로 대답만 해주면 된다. 잘 모르는 사람은 단계를 따라 가며 같이 설치를 해 보도록 하자.

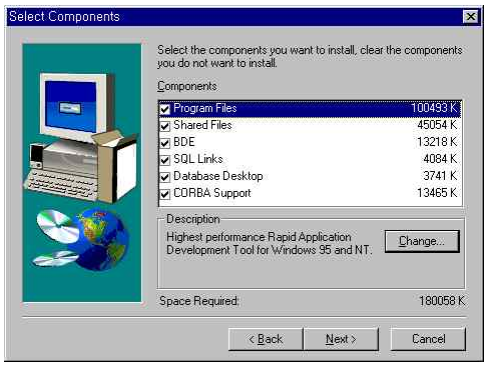
델파이 설치 CD 를 CD-ROM 드라이브에 넣기만 하면 다음과 같이 설치 프로그램이 자동으로 실행된다. 만약 자동 실행이 안된다면 CD-ROM 의 루트 디렉토리에 있는 install.exe 를 실행하면 된다.

그림
델파이 메인 설치 화면

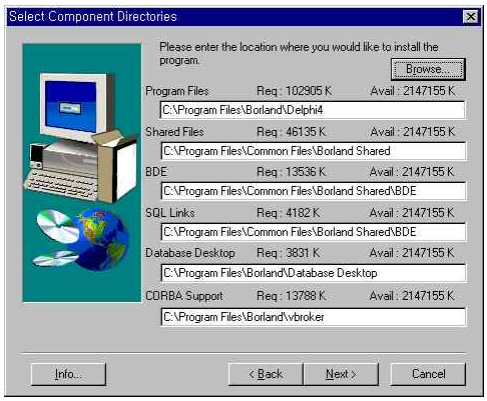



간단한 안내문과 설치 대상 요소들이 있다. 여기서 제일 위에 있는 Delphi 4 를 선택하면 델파이 설치가 시작된다. 나머지 요소들도 이 화면에서 설치할 수 있다. 설치가 시작되면 제일 먼저 환영 메시지가 나타나며 릴리즈 노트, 라이센스 동의문, 시리얼 번호 입력 대화상자등이 차례대로 나타나는데 순서대로 읽어보고 시리얼 번호를 입력해 주면 되며 Next 버튼으로 다음 대화상자로 넘어간다. 셋업 타입 대화상자는 델파이를 어느 수준까지 설치할 것인가를 묻는다.


Typical 은 가장 일반적인 옵션으로 델파이를 설치하는 것이며 Compact 는 필수적으로 필요한 요소만 설치하는 것이다. 설치 과정을 사용자가 일일이 지정해 주려면 Custom 을 선택하면 된다.



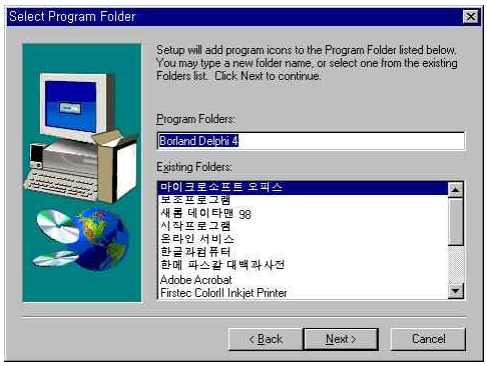

커스텀 셋업 대화상자에서는 델파이의 각 요소를 보여주는데 여기서 설치하고자 하는 요소만 선택해 주면 된다. 일단 Typical 을 선택하도록 하자. 다음 대화상자는 인터베이스 클라이언트를 설치할 것인가를 질문하는데 클라이언트/서버 데이터 베이스 프로그래밍을 하려면 이 옵션을 선택해야 한다. Next 버튼을 누르면 라이센스 동의문을 한번 더 보여주며 설치 경로를 물어온다.




**그림**

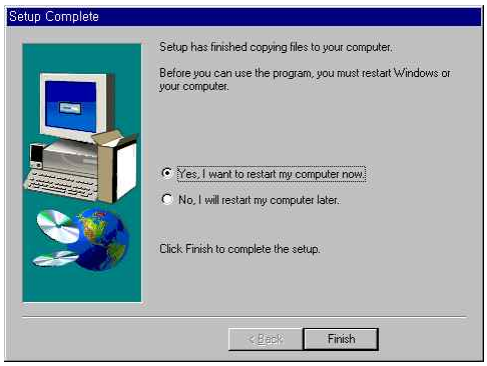
설치할 디렉토리를 선택한다.

디폴트로 C:₩Program Files₩Borland 디렉토리에 설치되는데 Browse 버튼을 눌러 설치 경로를 바꿀 수도 있다. 경로를 설정한 후에는 시작 버튼에 등록될 폴더 이름을 질문한다.



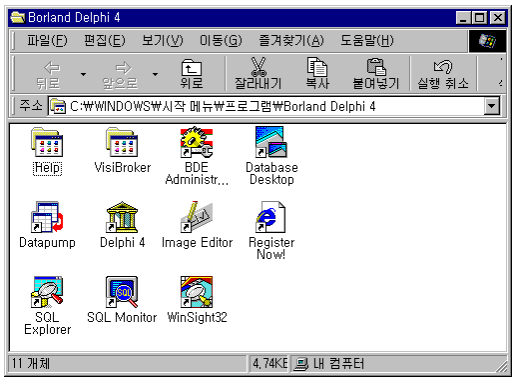

디폴트로 주어진 Borland Delphi 4 를 받아들인다. 마지막으로 간단한 설치 안내문을 보여준다.


Install 버튼을 누르면 파일을 하드 디스크로 복사하며 설치가 진행된다. 설치 중에 델파이 4 의 여러 가지 특징들과 개선점에 대해 보여주는데 주로 자기 제품 자랑이다. 관심있으면 읽어보고 귀찮으면 한 20 분정도 커피 한잔 마시고 있으면 된다. 파일 복사가 끝나면 시스템 레지스트리를 업데이트하며 앞에서 선택한 설치 옵션에 따라 인터베이스 클라이언트 등을 추가로 설치한다. 설치가 완료된 후에는 다음 대화상자를 보여주며 시스템을 재부팅할 것을 요구한다.




Finish 버튼을 누르면 시스템이 재부팅되며 설치가 완료된다. 설치가 완료된 후에 Delphi 4 폴더가 생성되며 이 폴더에 델파이 4와 여러 가지 부속 프로그램이 등록되어 있다.

그림
Delphi4 폴더




이상이 델파이를 설치하는 기본적인 절차이다. 보통 별 무리없이 설치되지만 만약 설치과정에서 문제가 있다면 동봉된 매뉴얼을 참조하여 문제점을 해결하기 바란다.

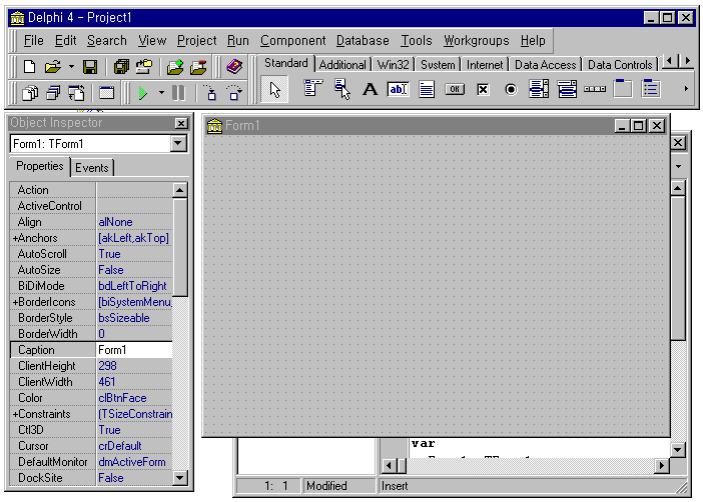
# 다. 실행

이제 설치를 마쳤으므로 델파이를 실행시켜 보자. Borland Delphi 4라는 폴더가 새로 생성되었을 것이며 이 폴더 안에는 델파이에 관련된 여러 가지 프로그램과 문서들이 들어 있다. 이 중 

요렇게 집모양처럼 생긴 아이콘이 델파이 메인 프로그램이다. 이 아이콘을 더블클릭하면 델파이가 실행된다. 아니면 시작 버튼에서 델파이를 찾아 실행시키도록 한다. 앞으로 델파이를 계속 사용할 것이

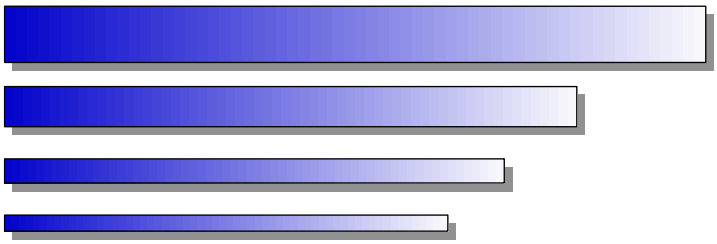
므로 데스크 탑에 쇼트컷 정도는 만들어 두도록 하자. 델파이를 실행시키면 다음과 같이 예쁜 모양의 델파이 개발 환경이 보일 것이다.

**그림**
델파이 개발 환경

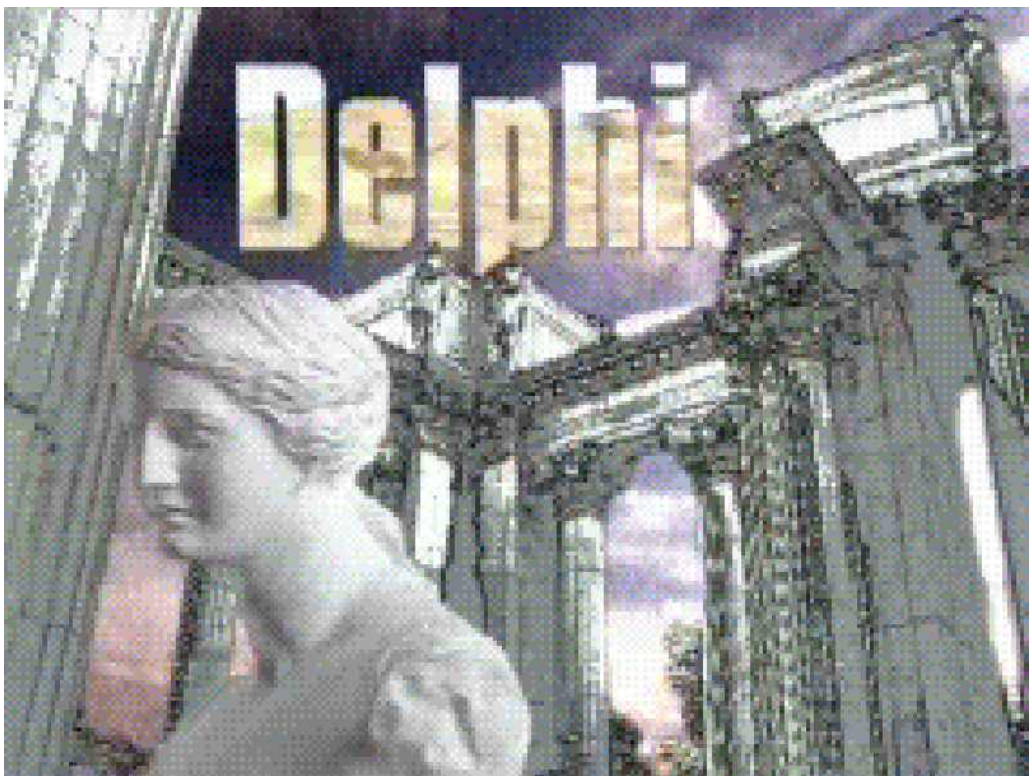



시스템 메뉴를 더블클릭하거나 타이틀 바의 제일 오른쪽에 있는 X 를 클릭하면 종료된다. 이제 다음 장부터 개발 환경 사용법을 익히고 프로그래밍 과정과 문법을 부지런히 공부해 나가도록 하자.

# 델파이의 기본



# 제 2 장

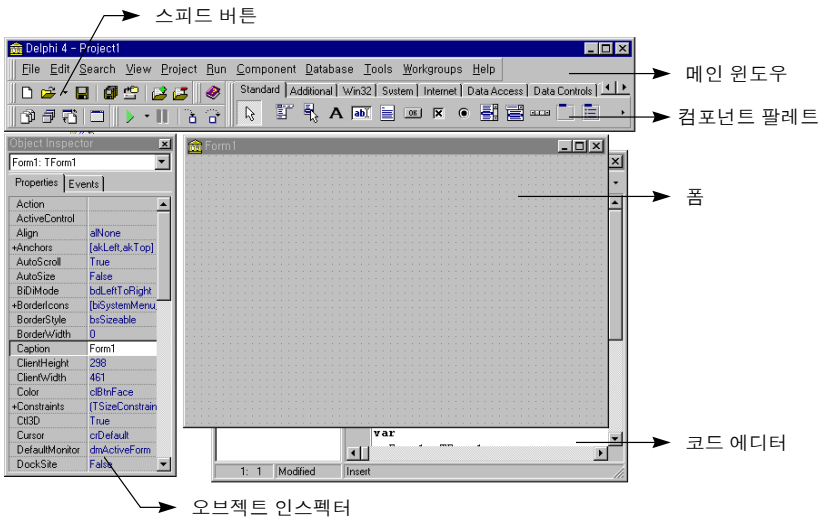


# 2-1 화면 구성

델파이를 무사히 하드 디스크에 설치하고 드디어 델파이의 첫걸음을 내 디딜 단계가 되었다. 어떤 프로그램과 친해지려면 먼저 그 프로그램의 모양새에 익숙해져야 한다. 화면에 펼쳐진 각 부분이 어떤 기능을 가지고 어떻게 조작하는지를 알아야 연습삼아 실습이라도 해 볼 것이 아니겠는가?

델파이를 실행시키면 다음과 같이 여러 개의 윈도우가 화면에 펼쳐질 것이다. 이 윈도우들이 델파이 개발 환경을 이루는 주요 윈도우들이며 이 외에도 당장은 보이지 않지만 개발 과정에 사용되는 숨겨진 여러 개의 윈도우가 준비되어 있다. 숨겨진 윈도우에 관해서는 관련 부분에서 천천히 알아보도록 하고 먼저 기본적으로 나타나는 각 윈도우의 명칭과 역할에 대해 알아보도록 하자.

그림
델파이의 화면 구성




제일 위쪽에 길쭉한 모양의 윈도우가 델파이의 메인 윈도우이며 이 안에 스피드 버튼, 메인 메뉴, 컴포넌트 팔레트가 있다. 나머지 윈도우들은 각각 폼, 코드 에디터, 오브젝트 인스펙터라고 하며 델파이 메인 윈도우에 속한 차일드 윈도우들이다.  메인 윈도우를 닫으면 나머지 윈도우들도 모두 닫혀지며 메인 윈도우를 최소화하면 나머지 윈도우들도 같이 최소화된다. 각각의 윈도우들에 대해 자세히 알아보자.

# 가. 폼

일단, 컴포넌트는 프로그램의 구성 요소라고 생각하기 바란다.

폼(Form)이란 델파이 프로그래밍의 가장 중심적인 재료이며 실행중에 우리 눈에 보이는 윈도우이다. 폼에 버튼이나 레이블, 리스트 박스 등등의 컴포넌트(Component)를 배치하여 프로그램을 만들어 나간다. 델파이로 프로그램을 만드는 것을 그림 그리기에 비유한다면 폼은 곧 도화지에 해당된다. 도화지에 구도를 잡고 스케치를 하고 채색을 하여 그림을 완성시켜 나가듯이 폼 위에 버튼, 레이블을 놓고 메뉴를 만들어 나가면서 프로그램의 겉모양을 완성시켜 나간다.

델파이를 처음 실행했을 때, 즉 프로그램을 만들기 시작했을 때 델파이는 아무것도 놓여져 있지 않은 빈 폼을 하나 만들어 준다. 이 폼은 타이틀 바, 시스템 메뉴, 최소, 최대 버튼, 닫기 버튼, 경계선을 가진 윈도우 모양을 하고 있으며 그 자체로 이미 실행 가능한 윈도우이다. 폼의 색상, 크기, 모양을 바꾸고 폼에 여러 가지 컴포넌트를 놓아 우리가 원하는 프로그램을 만든다. 다음은 폼에 여러 가지 컴포넌트를 놓아 프로그램을 만든 예이다.

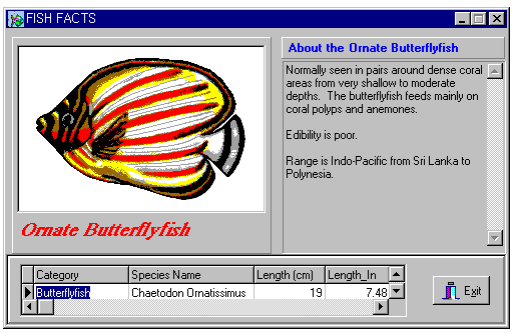
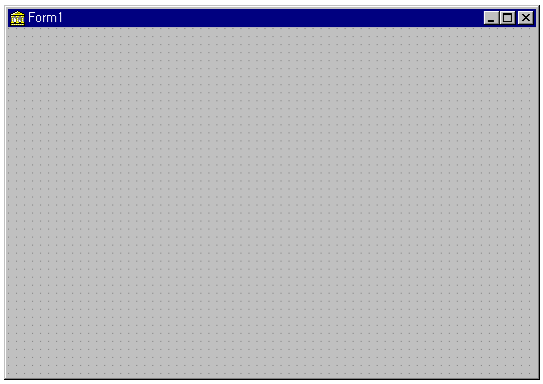



그림
폼에 컴포넌트를 배치한 모양

폼에 컴포넌트를 배치하는 자세한 방법에 대해서는 천천히 알아보기로 하자. 델파이가 디폴트로 만들어 주는 폼을 자세히 들여다 보면 폼 위에 규칙적으로 점들이 놓여져 있는 것을 볼 수 있다. 이 점은 디자인시에 컴포넌트가 배치될 격자를 나타내는 일종의 안내선이라고 할 수 있으며 실행중에는 보이지 않는다.

그림
컴포넌트의 배치를 도와주는 격자




컴포넌트를 점 단위로 배치하면 정렬이 너무 어려워지기 때문에 격자가 있는 곳에만 배치할 수 있도록 되어 있다. 이 격자가 눈에 거슬리는 사람은 Tools 메뉴의 Environment Options 를 선택한 후 Preference 페이지에서 Display Grid 옵션을 해제해 주면 디자인중에도 격자가 나타나지 않는다. 참고로 이 책에서는 깔끔한 인쇄를 위해 격자가 없는 상태에서 그림을 캡처하였다.

# 나. 스피드 버튼

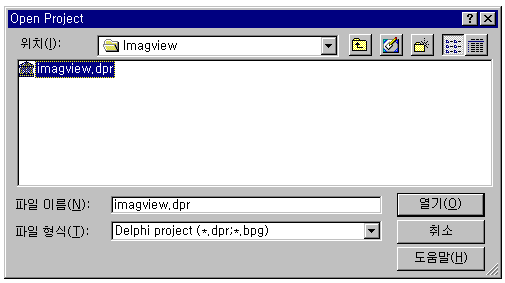
델파이 메인 윈도우의 좌측에 있는 16 개의 정사각형 버튼을 스피드 버튼(Speed Button)이라고 하며 메뉴에 있는 명령을 빠르게 실행시킬 수 있는 방법을 제공한다. 마우스로 클릭하기만 하면 원하는 동작을 즉시 실행할 수 있도록 만든 버튼들이다. 디폴트로 제공되는 16 개의 스피드 버튼은 다음과 같은 기능을 가진다.

그림
스피드 버튼

각 버튼이 어떤 기능을 하는가는 당장 몰라도 상관이 없으므로 스피드 버튼의 정의만 알아두도록 하자. 디폴트로 가장 자주 사용되는 기능에 대해서 16 개의 스피드 버튼이 제공되지만 사용자가 마음대로 버튼을 추가시키거나 변경할 수도 있다.

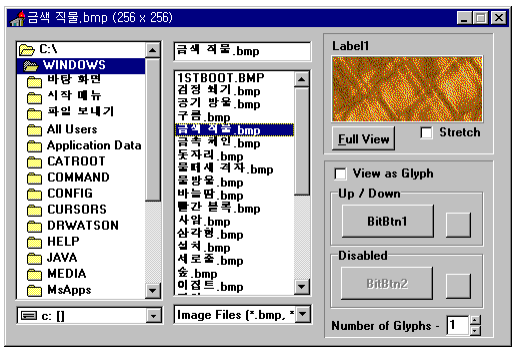
그럼 스피드 버튼을 사용하여 델파이와 함께 제공되는 예제 프로젝트를 읽어와서 실행해 보자. 스피드 버튼에서  버튼(File/Open Project 와 동일한 기능을 가진다)을 눌러 프로젝트를 읽어온다. Open Project 대화상자를 보여줄 것이다.

**그림**
프로젝트 열기 대화 상자




델파이의 DEMOS 디렉토리에 델파이로 만들어진 예제들이 있으며 이 중 IMAGVIEW 디렉토리를 선택하고 imagview.dpr 프로젝트를 선택한다. 프로젝트가 읽혀지고 폼이 나타날 것이다. 이 예제를 실행시키려면 스피드 버튼의  버튼(Run/Run 과 동일한 기능을 가진다)을 누른다. 잠시 컴파일 과정을 거쳐 예제가 실행될 것이다. 실행중의 모습은 다음과 같다.

**그림**
ImagView 예제의 실행 모습



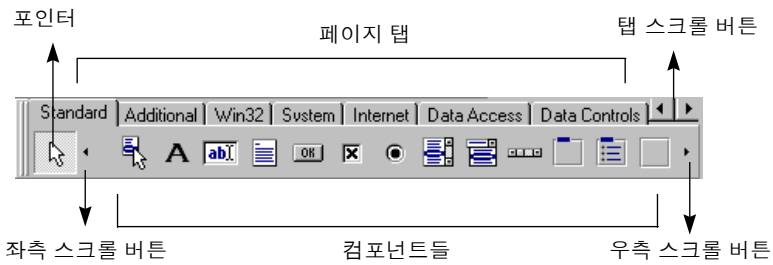

이 예제는 그래픽 파일을 선택하면 그림을 보여주는 예제이다. 예제를 종료시키면 다시 델파이 개발 환경으로 돌아온다. 스피드 버튼 두 개만 사용할 줄 알아도 예제 정도는 쉽게 불러와서 곧바로 실행해 볼 수 있다. 두 버튼을 사용하여 델파이가 어떤 예제를 제공하는지 DEMOS 디렉토리에 있는 예제를 마음껏 실행

해 보도록 하자.

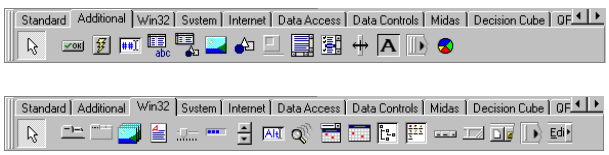
# 다. 컴포넌트 팔레트

델파이 메인 윈도우의 우측에 있는 여러 개의 버튼 그룹이 컴포넌트 팔레트(Component Palette)이며 컴포넌트들을 모아 놓은 것이다. 컴포넌트(Component)란 델파이 프로그램을 이루는 여러 가지 구성 요소를 말한다. 여기서 컴포넌트의 정의를 문장화하여 기술한다는 것은 무리이고 어쨌든 컴포넌트란 버튼, 라디오 버튼, 체크 박스, 리스트 박스 등등 우리가 윈도우즈 프로그램에서 항상 보아오던 것들의 일반 명사라고 이해하기로 하자. 컴포넌트 팔레트는 이런 컴포넌트를 모아놓은 집합소이다. 비슷한 유형끼리 그룹으로 분류되어 있으며 컴포넌트 팔레트 윗쪽에 있는 페이지 탭으로 그룹을 선택한다.

그림
컴포넌트 팔레트

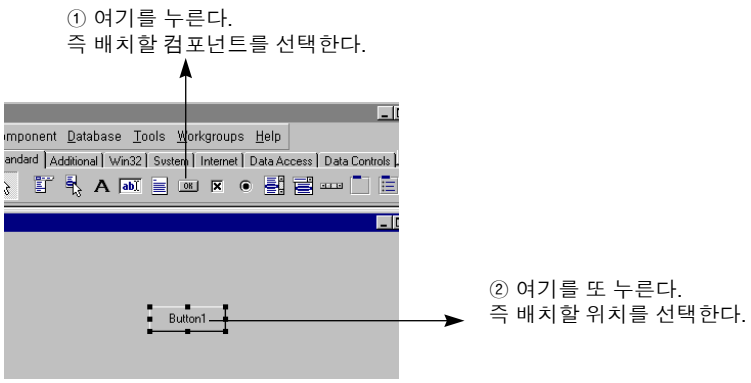



어떤 그룹에 어떤 컴포넌트가 있는지 컴포넌트 팔레트 윗쪽에 있는 페이지 탭을 눌러 구경이라도 해 보도록 하자. 모두 14 개의 페이지가 있는데 주로 많이 사용하는 페이지는 위 그림의 Standard 페이지와 두 번째 Additional, 세 번째 Win32 페이지이다.




컴포넌트를 폼에 배치하는 것은 아주 쉽다. 팔레트에서 배치하고자 하는 컴포넌트 버튼을 누른 후 컴포넌트를 놓고자 하는 폼의 위치를 한번 더 누르기만 하면 된다. 예를 들어 버튼 컴포넌트를 폼에 배치하려면 다음과 같이 한다.

그림
컴포넌트를 배치하는 방법

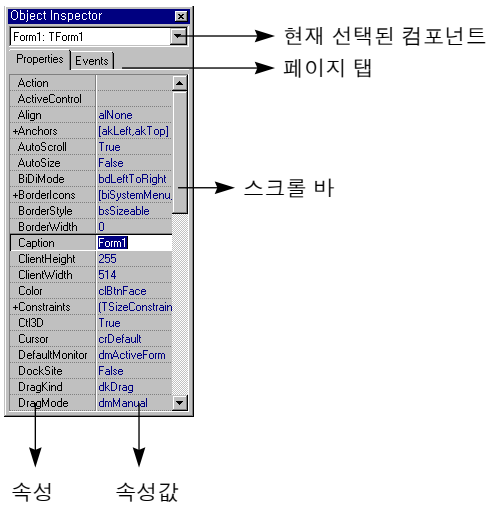



일단 폼에 놓여진 컴포넌트는 위치 이동, 삭제, 크기 조정을 자유롭게 할 수 있다.

# 라. 오브젝트 인스펙터

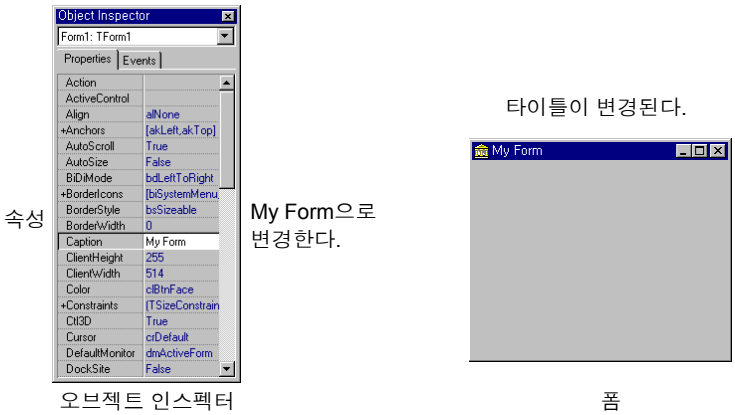
컴포넌트는 모두 속성(Property)이라는 것을 가지고 있다. 속성이란 컴포넌트의 외적인 모양과 내적인 기능을 정의하는 값이다. 예를 들어 폼이라는 컴포넌트를 보면 폼의 폭과 높이를 설정할 수 있고 바탕색은 어떤 색을 쓸 것인가, 타이틀 바에 어떤 제목을 보여줄 것인가에 따라 그 모양이 달라진다. 이렇게 컴포넌트의 개별적인 특성을 정의하는 값을 속성이라고 한다. 오브젝트 인스펙터는 컴포넌트의 속성을 보여주고 수정할 수 있도록 해주는 윈도우이다. 세로로 길게 두 줄로 이루어져 있으며 좌측에는 속성을 나타내고 우측에는 속성값을 나타낸다.

**그림**
오브젝트 인스펙터



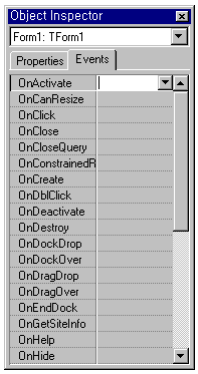

화면에 보이지 않는 속성은 스크롤 바를 이용하여 상하로 이동하며 볼 수 있다. 간단한 실습을 해보자. 오브젝트 인스펙터에서 “Form1”이라고 되어 있는 Caption 속성을 클릭한 후 속성값을 My Form 으로 바꾸어 보자. 폼의 타이틀 바에 있는 제목 문자열이 My Form 으로 바뀔 것이다.

**그림**
오브젝트 인스펙터를 이용한 속성의 변경




Caption 속성은 폼의 타이틀을 설정하는 속성이라는 것을 알 수 있다. 오브젝트 인스펙터는 속성을 설정하는 것 외에도 컴포넌트의 이벤트를 정의하기도 한다. 페이지 탭에 있는 Events 탭을 클릭하면 오브젝트 인스펙터가 다음과 같이 변한다.

그림
오브젝트 인스펙터의 Events 페이지

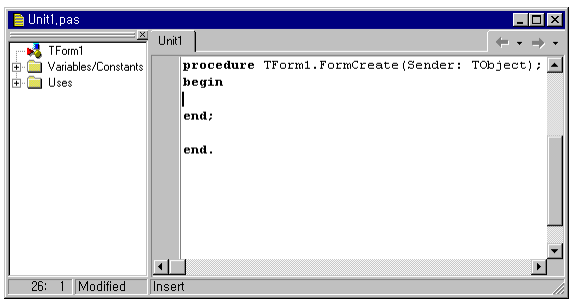



이 상태에서 프로그램의 동작을 기술하는 이벤트 핸들러를 만들어 코드를 작성하는데, 이 부분에 대해서는 잠시 후에 실습해 보기로 하자.

# 마. 코드 에디터

컴포넌트들은 속성이라는 성질을 가지는 것 뿐만 아니라 행동 양식에 해당하는 코드(Code)를 가진다. 컴포넌트의 행동 양식이란 사용자가 컴포넌트를 조작할 때 어떤 식으로 동작할 것인가를 뜻하는 것이다. 예를 들어 버튼을 누를 경우 프로그램을 끝낸다거나 메뉴를 선택할 때 게임을 시작하는 등의 동작을 말한다. 코드 에디터는 폼위에 가려져 있기 때문에 잘 보이지 않지만 코드를 작성할 때는 폼위로 올라온다. 해당 컴포넌트를 더블클릭하면 즉각 코드 에디터가 열린다. 정말 그런지 폼의 아무 곳이나 더블클릭해 보자. 다음과 같이 코드 에디터가 열릴 것이다.

그림
코드 에디터




여기서 작성하는 코드는 폼이 최초 생성될 때의 동작을 기술하는 것이다. 기술하는 방식은 파스칼 문법에 따라 원하는 동작을 가장 잘 지시하도록 간결하고 명료하게 코드를 작성하되 당장 그 방법을 여기서 논할 수는 없고 이 책 전체를

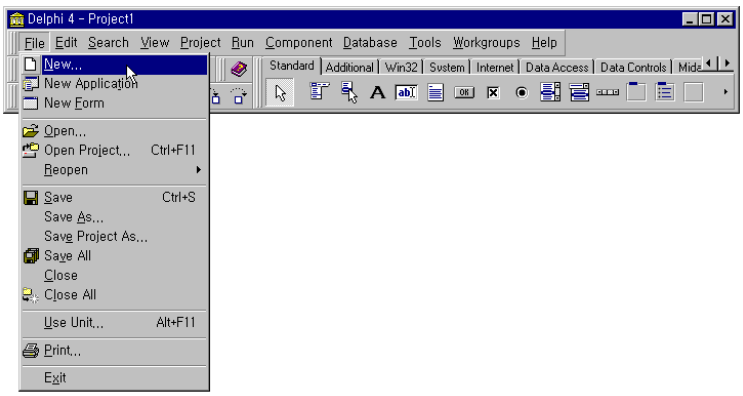
통틀어 순서대로 익혀가도록 하자.

코드 에디터는 프로그래밍에 적합한 여러 가지 고급 기능을 가지고 있지만 일단은 노트 패드나 문서 작성기를 사용하는 방법대로 사용하면 큰 문제없이 편집할 수 있다.

## 바. 메인 메뉴

메인 메뉴는 델파이의 메인 윈도우에 위치하고 있으며 델파이의 거의 모든 기능이 항목별로 잘 정리되어 있다. 사용하는 방법은 표준적인 메뉴 사용법과 전혀 틀리지 않으므로 별도의 설명은 하지 않는다. 개별적인 메뉴 항목에 대해서는 앞으로 관련 부분에서 자세하게 논하게 될 것이다.

그림
델파이의 메인 메뉴




이 책에서는 메뉴 항목을 표시할 때 “File 메뉴의 New 항목”이라고 길게 표현하지 않고 슬래시 기호(/)를 사용하여 “File/New”라고 표현하므로 참고하기 바란다. 이런 메뉴 표기 방법은 이제 거의 모든 문서에서 표준화된 것이므로 모르고 있었다면 지금부터 익숙해지도록 하자.

이상이 델파이 개발 환경의 간략한 구성이다. 그러나 델파이를 실행했을 때 당장 보이는 부분보다는 숨겨져 있는 윈도우가 더 많이 있다. 나머지 윈도우들은 실습을 통해 익히기 바라며 관련된 부분에서 개별적으로 소개하기로 한다.

# 2-2 간단한 예제 제작

2jang
fruit

델파이 프로그래밍을 하는 데 필요한 화면 구성 요소를 죽 살펴보았다. 이제 여기서 아주 간단한 델파이 프로그램을 만들면서 프로그램 개발 과정을 실습해 보도록 하자. 단 여기서 논하고자 하는 것은 개발 과정이지 프로그래밍을 다루려는 것은 아니므로 모르는 내용이 있더라도 그대로 따라만 하기 바란다. "프로그램을 이런 순서로 짜는구나" 하는 정도만 알면 된다. 그럼 컴퓨터를 켜고 윈도우즈를 실행시킨 후 다음 단계를 따라가 보자.

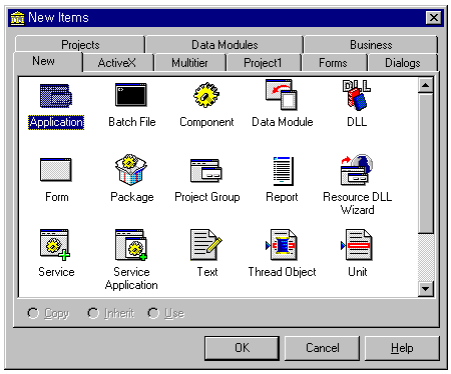
## 1 델파이를 실행한다

델파이 프로그래밍을 하기 위한 첫 번째 과정이다. 시작(  ) 버튼에서 델파이를 찾아 실행시킨다. 멋있는 신전 모양의 그림이 나타나며 잠시 후 델파이가 화면에 모습을 드러낼 것이다. 델파이는 여러 개의 윈도우를 열어놓고 사용하므로 가급적이면 델파이 이외의 윈도우는 닫아버리거나 최소화시켜 깨끗한 화면을 쓰는 것이 좋다.

## 2 새로운 프로젝트를 시작한다

새로 프로그램을 작성하겠다는 의사 표현을 해 주어야 한다. 델파이는 개발중인 프로그램을 프로젝트(Project)로 관리하므로 새로운 프로젝트를 시작하는 것이 프로그램 개발의 첫 단계이다.  델파이를 처음 실행시킨 상태에서는 새 프로젝트가 시작되어 있으므로 바로 프로그래밍을 할 수 있지만 다른 프로그램을 개발중이었다면 다음 절차대로 새 프로젝트를 시작한다. File 메뉴의 New 항목을 선택하면 다음과 같은 대화상자를 보여준다.

그림
New Item 대화상자

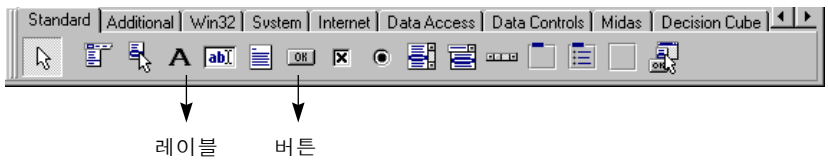



이 대화상자는 프로젝트, 폼, 유닛 중 새로운 것을 작성할 때 나타나며 프로젝트를 새로 만들 때는 New 페이지에서 Application 을 선택하고 OK 버튼을 눌러주면 된다. 이때 다른 프로젝트 작업중이었다면 저장 여부를 물어올 것이다. File/New 명령을 선택하는 대신 File/New Application 메뉴를 바로 선택해도 된다. 이 명령에 의해 새로운 프로젝트가 시작되며 아무것도 작성되지 않은 빈 폼이 나타날 것이다. 이 폼이 우리가 만들어 나갈 프로그램의 재료가 된다.

## 3 폼 위에 컴포넌트를 배치한다

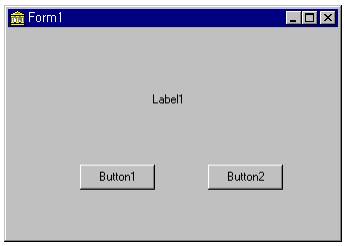
일단 폼의 크기를 적당하게 조절하도록 하자. 우리가 만들 프로그램은 그렇게 넓은 폼을 필요로 하지는 않으므로 디폴트 크기의 절반쯤 되게 폼의 크기를 줄인다. 폼의 크기를 줄이는 방법은 윈도우의 크기를 조절하는 방법과 같다. 폼의 경계선을 드래그하기만 하면 된다.

그리고 컴포넌트를 가져와 폼에 위치시킨다. 원하는 컴포넌트를 컴포넌트 팔레트에서 선택한 후 폼을 클릭하여 놓기만 하면 된다. 레이블 하나와 버튼 두 개를 폼에 배치하도록 하자.




버튼은 윈도우만 시작했다하면 맨날 맨날 보는 버튼이고 레이블이란 폼상에 놓이는 문자열이다. 다음은 세 개의 컴포넌트를 폼에 배치한 모양이다.

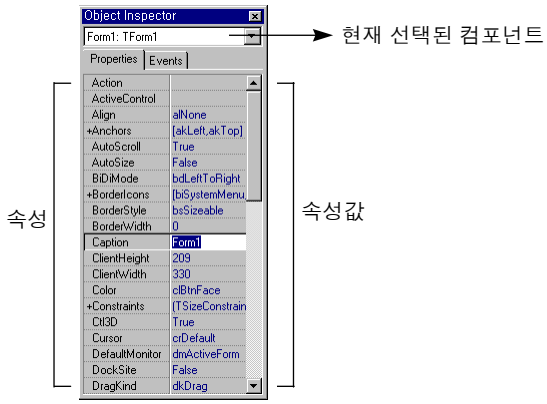
그림
컴포넌트를 폼에 배치한 후의 모양



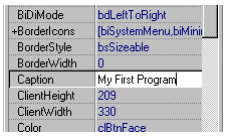

배치한 상태가 이와 다르다면 마우스로 컴포넌트를 드래그하여 적절한 위치로 옮기도록 한다. 정확하게 위치를 정할 필요는 없으며 대충 비슷하기만 하면 된다.

## 4 컴포넌트의 속성을 설정한다

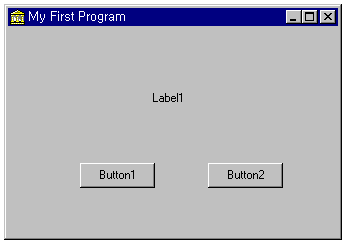
현재 컴포넌트 4 개가 배치되어 있다. 왜 4 개인가 하면 폼 자체도 하나의 컴포넌트이기 때문이다. 배치한 컴포넌트의 속성을 적절히 바꾸어 보자. 먼저 폼의 타이틀이 Form1 으로 되어 있는데 이것을 다른 타이틀로 바꾸어 보자. 폼의 빈 영역을 클릭하여 폼 컴포넌트를 선택하면 오브젝트 인스펙터에 폼의 속성이 나열되며 상단부의 콤보 박스에 현재 선택된 컴포넌트가 Form1 이라는 것이 표시된다.




폼의 타이틀을 바꾸기 위해서는 Caption 속성을 변경해 주면 된다. 속성의 Caption 을 클릭하면 다음과 같이 Caption 속성값을 입력할 수 있도록 캐럿이 나타난다.




디폴트로 설정된 Form1 캡션을 백 스페이스키로 지우고 My First Program 이라는 새로운 타이틀을 입력한다. 속성을 바꾸면 바뀌어진 결과가 즉각적으로 폼에 반영된다. 폼의 타이틀이 벌써 My First Program 으로 바뀌어져 있을 것이다.




특정 컴포넌트의 속성을 변경하는 것은 이처럼 간단하다. 변경하고자 하는 컴포넌트를 선택하고 오브젝트 인스펙터에서 속성값을 바꾸어 주기만 하면 된다. 컴포넌트의 속성 중 제일 중요한 것이 Name 속성이다. 이 속성은 그 컴포넌트의 이름을 뭐라고 칭할 것인가를 지정하며 프로그램내에서 컴포넌트를 참조하는 데 사용한다. 사람에게 영자, 순자, 삼돌이 등의 고유한 이름이 있듯이 컴포넌트들도 고유한 이름을 가진다.

디폴트로 주어지는 Name 속성은 Form1, Form2, Button1, Button2, Label1 등과 같이 컴포넌트의 유형과 숫자 하나로 이루어지지만 보통은 의미를 쉽게 알 수 있는 다른 이름으로 바꾸는 것이 더 편리하다. 그래야 다음에 어떤 컴포넌트가 어떤 의미를 가지는지 쉽게 파악할 수 있다. 컴포넌트가 많아지면 이름이 무척이나 헷갈리므로 요령껏 이름을 잘 정해 주도록 하자. 폼은 디폴트로 주어진 Form1 이름을 그대로 사용하기로 하고 버튼과 레이블의 Name 속성은 다음과 같이 바꾸도록 하자. 물론 오브젝트 인스펙터를 사용한다.

| 원래 이름 | 바꾼 이름 |
|---|---|
| Button1 | BtnApple |
| Button2 | BtnOrange |
| Label1 | Fruit |

Button1 의 이름을 BtnApple 이라고 작성했다. 왜 이런 이름을 붙였는가 하면 레이블의 문자열을 Apple 로 바꾸는 기능을 가지는 버튼이기 때문이다. 좀 길게 이름을 붙인다면 ButtonMakeApple 이라고 할 수 있겠지만 손가락이 너무 고달프기 때문에 간략한 이름을 붙여주는 것이다. 이름을 붙일 때는 간단한 생략형을 쓰지만 대신 버튼 이름을 읽을 때는 "비티엔애플"과 같이 읽지말고 "버튼애플"이라고 읽는 것이 좋다. 레이블은 과일 이름을 출력하는데 사용할 계획이므로 이름을 Fruit 로 주었다.

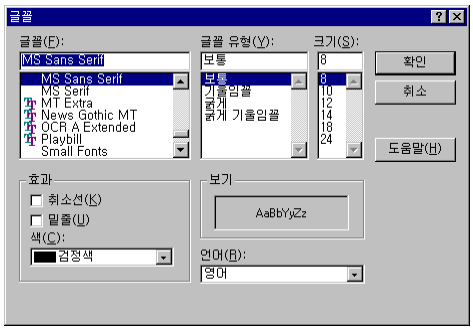
컴포넌트의 이름을 바꾸면 델파이는 Caption 속성도 Name 속성과 같이 변경해 준다. BtnApple 버튼의 캡션을 보면 역시 BtnApple 로 변경되어 있을 것이다. 델파이가 이렇게 해주는 이유는 Name 과 Caption 속성이 일치하는 경우가 많기 때문인데 마음에 들지 않으면 바꿀 수도 있다. 레이블은 Name 과 같은 Caption 을 그냥 쓰기로 하고 버튼의 Caption 속성은 다음과 같이 변경하기로 한다.

| 속성 | 속성값 |
|---|---|
| BtnApple.Caption | Apple |
| BtnOrange.Caption | Orange |

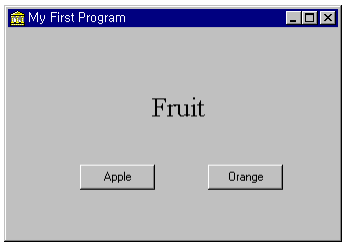
여기서 가운데 삽입된 쩜(.)은 "~의"라는 뜻을 가지며 "~컴포넌트의 ~속성"이라는 표현이다. BtnApple.Caption 은 BtnApple 컴포넌트의 Caption 속성을 의미한다. Caption 속성은 사용자에게 보여지는 컴포넌트의 제목이며 버튼위나 레이블에 문자열로, 또는 폼의 타이틀 바에 나타난다.

마지막으로 Fruit 레이블의 Font 속성을 바꾸어 보자. 글자의 크기가 너무 작아 별로 보기가 시원스럽지 못하므로 큼직한 크기의 글꼴로 바꾼다. Fruit 를 마우스로 클릭한 후 오브젝트 인스펙터의 Font 속성을 클릭한다. 오브젝트 인스펙터내에서 Font 속성을 설정할 수도 있지만 글꼴이란 좀 복잡한 속성이므로 별도의 대화상자를 불러내는 것이 좋다. Font 속성의 (TFont)라고 되어 있는 곳을 더블클릭하거나 오브젝트 인스펙터에 나타난  버튼을 누르면 다음과 같은 대화상자가 나타난다.

그림
폰트 선택 대화상자



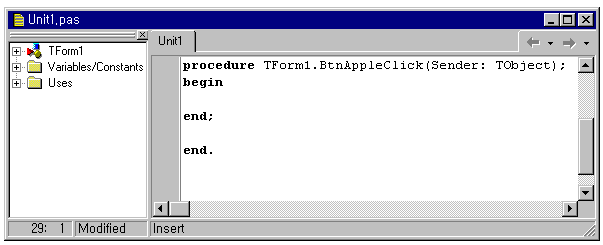

문서 작성기나 파일 관리자 등에서 사용하는 표준적인 대화상자이므로 구체적인 사용법은 생략하기로 한다. 이 대화상자에서 Times New Roman, 20 포인트 크기의 폰트를 선택하도록 하자. 여기까지 속성 설정을 마치면 폼의 모양은 다음과 같아진다.




## 5 코드를 작성한다

컴포넌트를 배치하고 속성을 주었으면 이제 프로그램의 겉모양은 완성된 것이다. 이제 프로그램이 어떤 상황에서 어떤 동작을 할 것인가를 지정해 주어야 하는데 이 과정을 코딩(coding), 즉 "코드를 작성한다"고 표현한다. 코드란 실행중에 컴포넌트가 선택될 경우 컴포넌트의 동작을 정의하는 명령이다. 우선 BtnApple 의 코드를 작성해 보자. 폼 상에 있는 BtnApple 버튼을 더블클릭하면 다음과 같이 코드를 작성할 수 있는 코드 에디터가 열린다.

그림
코드를 작성하는 코드 에디터

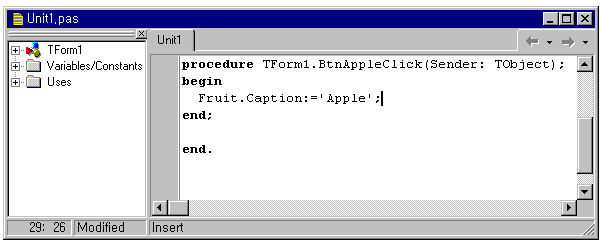



여기에 작성하는 코드는 실행중에 사용자가 BtnApple 버튼을 누를 경우 어

떤 동작을 할 것인가를 지정하는 것이다. 코드 에디터를 열면 벌써 BtnApple 의 코드를 작성할 수 있도록 코드의 뼈대(frame)가 만들어져 있다. 이 뼈대의 begin 과 end; 사이에 우리가 원하는 코드를 삽입해 주기만 하면 된다. 다음과 같이 코드를 작성한다.

```
Fruit.Caption:='Apple';
```

이 코드의 의미는 Fruit.Caption 속성에 'Apple'이라는 문자열을 대입하여 레이블의 Caption 속성을 변경한다는 뜻이다. 입력한 후의 코드 에디터 모양은 다음과 같다.




이 코드의 문법적 의미는 차후 거론하기로 하되 알기 쉽게 말로 풀어보면 다음과 같다.

BtnApple 버튼이 눌러질 경우 Fruit 레이블의 Caption 속성을 'Apple'이라는 문자열로 바꾸어라.

이번에는 BtnOrange 버튼의 코드를 작성해 보자. 똑같은 방법으로 BtnOrange 버튼을 더블클릭하여 코드 에디터를 연 후 다음 코드를 입력한다.

```
Fruit.Caption:='Orange';
```

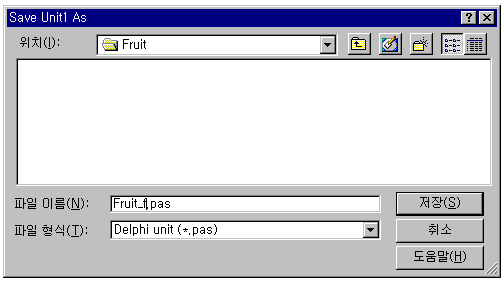
이 코드는 실행중에 BtnOrange 버튼이 눌러질 경우 레이블의 캡션을 'Orange'라는 문자열로 바꾸는 동작을 한다. 여기까지 작업이 완료되었으면 코딩이 끝나고 컴포넌트의 기능도 정의되었다.

## 6 프로젝트를 저장한다.

프로젝트를 다 만들었으면 일단 디스크에 저장해야 한다. 물론 그대로도 실행할 수 있지만 무엇인가 작업을 하고 난 후 저장하는 버릇은 컴퓨터를 사용하는

사람의 기본 자세이므로 일단 저장부터 하자. 그 전에 먼저 실습중에 만드는 프로젝트를 저장할 디렉토리부터 하나 만들어 둔다. C:₩MyPrj, 또는 C:₩DelphiExam 정도면 충분하다. 참고로 이 책의 모든 예제는 C:₩DelEx 디렉토리에 작성되었으므로 여러분들도 특별한 이유가 없는 한 그대로 따라 하기 바란다. 앞으로 이 디렉토리에 실습 예제들을 저장하도록 하자. 지금 만든 예제를 저장하기 위해 Fruit 라는 서브 디렉토리를 만든다.

File 메뉴의 Save All 항목을 선택하거나 스피드 버튼에서  를 누른다. 다음과 같은 대화상자가 출력된다.




방금 만든 폼을 어떤 이름으로 저장할 것인가를 지정하는데 디폴트로 주어진 이름은 Unit1.PAS 이지만 좀 더 기억하기 쉬운 이름으로 바꾸도록 하자. Fruit_f.PAS 란 이름을 준다. 폼을 저장하고 난 다음 프로젝트를 저장한다. 저장하기 대화상자를 보여주며 Project1.DPR 이라는 이름이 디폴트로 주어지는데 이 이름을 Fruit.DPR 로 바꾸도록 하자. 다음에 자세히 살펴보겠지만 델파이 프로젝트는 폼 파일(PAS)과 프로젝트 파일(DPR)로 구성된다. 이제 프로젝트의 저장이 완료되었다.

## 만든 프로그램을 실행해 보자

Run 메뉴의 Run 항목을 선택하거나 스피드 버튼  를 누른다. 그것도 귀찮으면 키보드의 F9 키를 가볍게 눌러주면 된다. 방금 만든 프로그램이 컴파일되고 실행되어 화면에 나타날 것이다.

그림
실행중인 모습


어떤 프로그램이 만들어졌는지 보자. 우리가 배치한 컴포넌트가 폼에 나타나 있다. Apple 이라고 되어 있는 버튼을 눌러 보면 레이블이 Apple 로 바뀌고 Orange 라고 되어 있는 버튼을 누르면 레이블이 Orange 로 바뀐다. 가진 기능이라고는 고작 버튼으로 레이블의 문자열을 바꾸는 것 뿐인 간단한 예제이다. 적당히 가지고 놀다가 시스템 메뉴를 더블클릭하여 종료(또는 타이틀 바의 오른쪽에 있는 닫기 버튼을 누른다)하도록 한다. 프로그램을 종료하면 다시 델파이 개발 환경으로 돌아온다.

## 8 에러가 있을 경우 수정한다

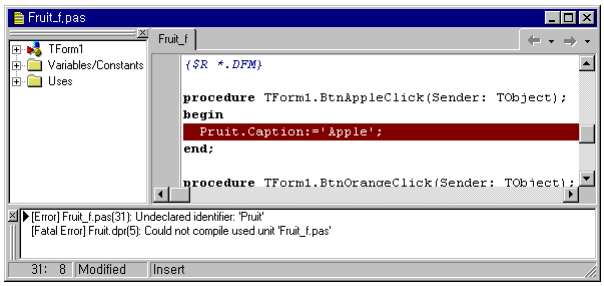
여기까지 프로그램을 만드는 일련의 과정을 실습해 보았다. 그러나 꼭 이런 절차대로 프로그램이 만들어지지만은 않는다. 왜냐하면 사람이란 존재가 워낙 부정확하고 실수가 많은지라 개발 과정 중 한 곳에서 오류를 범할 수 있기 때문이다. 그러나 걱정할 필요는 없다. 오류가 있을 경우 델파이는 어디가 어떻게 잘못되었는지 친절히 가르쳐 주므로 그 부분만 제대로 고쳐주면 된다.

그럼 에러가 발생했을 경우 어떻게 대처해야 하는지 실습해 보기 위해 일부러 실수를 해 보도록 하자. 위 프로그램에서 BtnApple 을 더블클릭하여 코드 에디터를 불러낸 후 코드를 약간 고쳐 보자. 다음과 같이

```
procedure TForm1.BtnAppleClick(Sender: TObject);
begin
  Pruit.Caption:='Apple';
end;
```

Fruit 라고 써야 하는데 실수로 Pruit 라고 했다고 가정하고 프로그램을 실행시켜 보자. 아마 다음과 같은 에러가 발생하고 어디가 잘못되었는지 빨간색으로 반전시켜 보여줄 것이다.

그림
에러 발생




Undeclared identifier 란 에러는 코드에서 칭한 이름 Pruit 가 정의되어 있지 않은 잘못된 이름이라는 뜻이다. "Pruit 란 이름은 컴포넌트 이름도 아니고 폼 이름도 아니고 도대체 뭘 말하는지 잘 모르겠습니다"라는 뜻이다.

이렇게 에러가 발생했을 때는 어디가 잘못되었는가를 살펴보고 에러를 고쳐주기만 하면 된다. 코드 에디터에서 Pruit 를 Fruit 로 다시 고쳐주도록 하자. 이번에는 제대로 실행될 것이다. 만약 이 외에 다른 종류의 에러를 만나더라도 에러를 수정하는 절차는 비슷하다. 에러의 빈도가 가장 높은 것은 오타이므로 에러가 발생하면 제대로 입력했는지 철자를 하나 하나 뜯어 보도록 하자. 에러를 수정하고 난 후에는 다시 저장해야 한다.

여기까지 프로그램 개발을 완료했다. 처음 만들어 본 프로그램이 어떤가? 어렵지도 않고 결과가 곧바로 나타나므로 재미있게 공부했을 것이다. 그런데 프로그램이 너무 간단해서 시시하다는 생각이 들지 않는가? 고작 버튼 두 개만 가지고 노는 이 따위 프로그램이란 생각이 들지도 모르겠다. 그러나 이 예제는 자세히 뜯어 보면 누구말대로 불후의 명작이다. 왜 그런가 하면 눈에 보이는 기능 외에도 다음과 같은 엄청난 기능을 가지고 있기 때문이다.

❶ 타이틀 바를 드래그하여 위치를 옮길 수 있다. 화면의 어느 곳에라도 갈 수 있으며 몇 번이라도 위치를 옮길 수 있다.

❷ 타이틀 바를 더블클릭하면 최대 크기로 확장된다. 다시 한번 더 타이틀 바를 더블클릭하면 원래 크기대로 감쪽같이 돌아온다.

❸ 최소화 버튼을 누르면 아이콘 크기만큼 작아지며 작업 표시줄에 조용히 웅크리고 앉아 있는다.

❹ 시스템 메뉴를 이용하면 키보드로도 조작할 수 있다.

❺ 게다가 경계선을 드래그하여 크기를 마음대로 늘렸다 줄였다 할 수 있다.

❻ 다른 윈도우에 가려졌다 다시 나타나면 원래 모양 그대로 복구된다.

일견 언뜻 보기에는 농담같아 보이고 너무나 당연한 것처럼 생각될지도 모르는 그런 내용들이다. 그러나 이것이 과연 당연하기만 한 것인지 한번 생각해 보라. 도스에서 이런 프로그램을 하나 만들려면 가히 엄청난 노력과 시간이 소요될 것이다. 델파이가 뒷쪽에서 사용자의 의도를 파악하고 철저한 지원을 하고 있기 때문에 이런 프로그램을 짜는 것이 어렵지 않은 것이지 원래부터 간단한 프로그램은 아니다.

이제 여러분은 델파이로 첫 번째 예제를 만들어 보았다. 아마 대부분은 델파이가 뭐하는 프로그램인지, 어떻게 쓰는 것인지를 알았겠지만 일부는 뜻대로 되지 않은 사람도 있을 것이다. 이 예제를 만드는 일에 성공했다면 다음에 필자가 제시하는 예제를 직접 만들어 보아라. 좀 더 델파이와 친숙해지기 위한 방편이다. 아직 문법을 논하지는 않았지만 :=가 대입문이고 문자열을 홑 따옴표로 '요렇게' 표현한다는 것 쯤은 눈치챌 수 있을 것이다.

① 똑같이 버튼 두 개를 배치하고 이번에는 폼의 Caption 속성을 Apple, Orange 로 변경하도록 해보자. 결국 위의 예제와 동일한 예제이다.

② 버튼을 하나 더 늘려 세 개를 배치하고 Apple, Orange, Banana 로 레이블을 변경하도록 해 보자. 조금 더 확장한 형태이지만 방법이나 절차는 동일하다.

워낙 쉬운 문제라 따로 정답을 실지는 않았다. 만약 이 문제를 풀지 못한 사람이 있다면 배포 CD 의 2jang 디렉토리에 ChgCaption, Fruit2 예제를 열어보기 바란다.

# 2-3 개발 환경

간단한 예제 제작을 통해 프로그램 개발 절차에 관해 알아보았다. 여기서는 델파이에서 제공하는 몇 가지 유용한 개발 환경에 관해 알아보자. 프로그래밍을 배우기 전에 우선 델파이라는 컴파일러와 친해질 필요가 있다. 전체적인 개발 환경에 대해서는 나중에 다시 언급하기로 하고 일단 쉽게 배워 요긴하게 쓸 수 있고 특히 이후 학습에 도움이 될만한 몇 가지를 알아본다.

## 가. 풍선 도움말

델파이는 메인 윈도우에 여러 개의 스피드 버튼을 가지고 있고 또 많은 양의 컴포넌트를 보유하고 있다. 스피드 버튼과 컴포넌트 버튼들은 모두 예쁘장한 비트맵으로 되어 있는데 보기에 좋은 것은 사실이지만 처음 대하는 초보자들은 어떤 버튼이 어떤 의미를 가지고 있는지 얼른 익숙해지기 어렵다. 그래서 델파이는 예쁘장한 아이콘 그림 외에도 말로 표현된 풍선 도움말을 제공한다. 마우스 커서를 버튼 위에 살짝 가져다 놓기만 하면 노란색 직사각형 안에 어떤 버튼인지 설명이 나타난다.


**그림** 버튼에 대한 간단한 설명을 보여주는 풍선 도움말

마우스 커서를 다른 버튼으로 옮겨가면 풍선 도움말도 따라 옮겨간다. 설명을 읽어보고 버튼을 선택하므로 초보자가 버튼의 의미를 외우기에 아주 편리하다. 한 가지 아쉬운 점은 도움말이 한글로는 나오지 않는다는 점이다. 풍선 도움말을 참조하여 스피드 버튼과 몇몇 컴포넌트는 일단 이름을 외워 두도록 하자. 어차피 델파이랑 친해지면 모두 외워야 할 버튼들이다.

## 나. 스피드 버튼

스피드 버튼의 정의에 대해서는 앞에서 이미 설명했다. 이제 스피드 버튼이 어떤 동작을 하는지 하나씩 알아보자. 스피드 버튼은 메뉴에 있는 명령과 대체

성이 있어서 스피드 버튼에 있는 기능은 메뉴에도 있다. 각 버튼의 설명 뒤에 동일한 동작을 하는 메뉴 항목을 밝힌다. 메뉴 항목 표기에 사용되는 /(슬래시)기호는 메뉴 항목 사이의 종속 관계를 나타내며 이런 메뉴 표기법은 이 책 전체에 걸쳐 일괄적으로 적용된다.

**표**
스피드 버튼

| 버튼 | 기능 |
|---|---|
|  New | 현재 프로젝트에 새로운 항목을 추가하는 New 대화상자를 보여준다. File/New |
|  Open | 디스크상에 보관해 둔 텍스트 파일을 읽는다. 소스 파일을 읽을 때 사용한다. File/Open |
|  Save | 텍스트 파일(또는 소스 파일)을 디스크에 보관한다. File/Save |
|  Save All | 프로젝트를 디스크에 저장한다. 즉 지금 피곤하니까 내일 계속 작업하기 위해 디스크에 작업 내용을 보관해 두는 것이다. File/Save All |
|  Open Project | 프로젝트를 연다. 즉 어젯밤에 만들다가 디스크에 저장해 둔 프로젝트를 다시 읽어오는 것이다. File/Open Project |
|  Add file to project | 현재 열려진 프로젝트에 새로운 파일을 첨가한다. 첨가된 파일이 코드 에디터에 나타난다. Project/Add to project |
|  Remove file from project | 현재 열려진 프로젝트에 있는 파일을 삭제한다. Project/Remove form project |
|  Help Contents | 델파이 도움말을 보여준다. Help/ Contents |
|  View Unit | 현재 프로젝트에 포함된 유닛의 목록을 보며 그중 하나를 선택한다. 여기서 선택한 유닛이 코드 에디터에 나타난다. View/Units |
|  View Form | 현재 프로젝트에 포함된 폼의 목록을 보이며 그 중 하나를 선택한다. 여기서 선택한 폼이 즉각 열리며 활성화된다. View/Forms |
|  Toggle Form/Unit | 현재 폼에서 작업을 하고 있으면 코드 에디터로 작업을 전환하고 반대로 현재 코드 에디터에서 작업을 하고 있으면 폼으로 작업을 전환한다. View/Toggle Form/Unit |

| | |
|---|---|
|  New Form | 새로운 빈폼을 만들어 현재 프로젝트에 추가한다. File/New Form |
|  Run | 현재 프로젝트를 컴파일시키고 실행한다. Run/Run |
|  Pause | 실행중인 프로그램을 잠시 중단한다. Run/Program Pause |
|  Trace into<br>  Step over | 이 두 개의 버튼은 프로그램의 오류를 검사하는 디버깅에 사용되며 자세한 사용법은 차후 거론하기로 한다. |

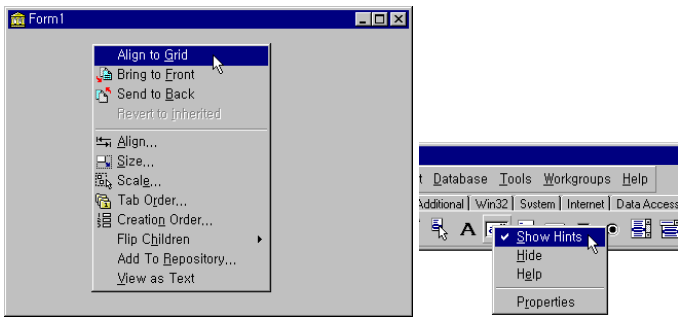
# 다. 스피드 메뉴

스피드 메뉴는 팝업 메뉴, 또는 컨텍스트 메뉴라고도 한다.

스피드 메뉴(Speed Menu)란 마우스의 오른쪽 버튼을 누를 때 나타나는 메뉴이다. 어떤 메뉴가 나타날 것인가는 마우스 커서가 어느 부분에 위치하고 있는가에 따라 달라진다. 컴포넌트 팔레트 위에서 오른쪽 버튼을 누르면 컴포넌트 팔레트에 관한 스피드 메뉴가 나타나고 오브젝트 인스펙터 위에서라면 또 그에 관한 스피드 메뉴가 나타난다. 폼 위에서나 코드 에디터에서도 그에 관련된 스피드 메뉴가 나타나며 이 외에도 델파이의 각 부분에는 고유의 스피드 메뉴가 있다.

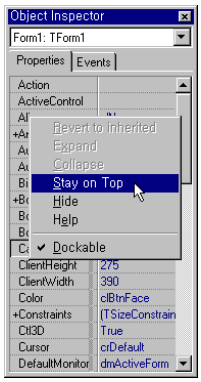
**그림**

델파이의 각 윈도우는 고유의 스피드 메뉴를 가진다.




스피드 메뉴 사용 예를 들어보자. 오브젝트 인스펙터 위에 커서를 놓고 오른쪽 버튼을 누르면 다음과 같은 스피드 메뉴가 나타난다.

그림
오브젝트 인스펙터의 스피드 메뉴




여기서 Stay on Top 이란 항목을 선택해 보자. 그러면 오브젝트 인스펙터는 폼이나 코드 에디터에 의해 가려지지 않고 항상 위에 있게 된다. 이 상태를 해제하려면 다시 한번 더 스피드 메뉴의 Stay on Top 을 선택하면 된다.

# 2-4 도움말

델파이는 규모가 상당히 큰 프로그램이다. 차지하는 디스크 용량이 크고 실행시에 메모리도 많이 사용하고 사용 방법도 그리 간단하지 않다는 뜻이다. 이렇게 규모가 큰 프로그램을 배우려면 당연히 시간도 많이 들고 진지한 노력이 필요하다. 그렇다면 델파이를 배우는 가장 빠르고 정확한 방법은 어떤 것일까?

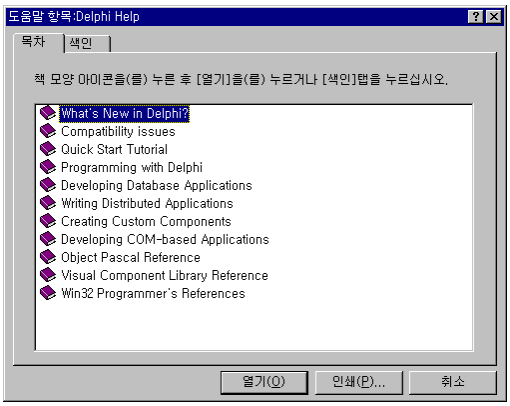
책을 보고 공부하는 방법이 있고 잘 아는 사람에게 배우는 방법이 있지만 그보다는 도움말을 사용하여 델파이에게 델파이를 배우는 것이 가장 확실한 방법이다. 델파이가 제공하는 도움말은 그야말로 가히 엄청날 정도로 상세하며 광범위하다. 도움말 파일을 텍스트로 바꾸어(HLP2DOC.EXE 라는 공개 유틸리티를 사용한다) 인쇄할 경우 무려 수 천 페이지에 달하는 매뉴얼이 만들어진다.

예전에는 도움말을 대충 만들고 도움말에서 "자세한 사항은 매뉴얼을 보세요"라고 했었는데 델파이에서는 오히려 꺼꾸로 되어 매뉴얼을 읽다 보면 "자세한 사항은 도움말을 보세요"라고 되어 있는 문구가 심심찮게 나올 정도다. 델파이 학습에 있어서 도움말은 모든 진리를 담고 있는 바이블과도 같다. 델파이를 배우기 전에 델파이의 도움말을 사용하는 방법을 익혀 두도록 하자. 마치 전쟁에 나가는 군인이 녹슬은 창과 방패를 닦고 갑옷을 준비하는 것과 같다. 도움말을 잘 이용할 줄 알아야 델파이를 좀 더 빠르고 정확하게 배울 수 있다. 델파이가 제공하는 도움말은 크게 2 가지 종류로 나눌 수 있다.

## 가. 메인 도움말

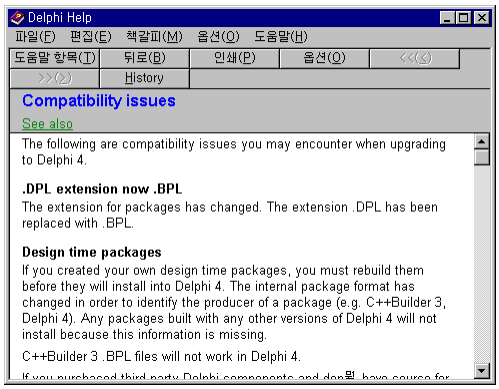
Help/Contents 를 선택하거나 또는 스피드 메뉴에서  버튼을 누르면 메인 도움말을 볼 수 있다.

**그림**
델파이의 메인 도움말 화면




범례별로 11 가지 도움말이 있는데 트리 형식으로 정리되어 있으므로 가지를 확장해 나가다가 원하는 도움말이 나왔을 때 더블클릭하면 도움말을 읽을 수 있다. 다음은 두번째에 있는 Compatibility issues 의 도움말 본문이다.

**그림**
도움말 본문




윈도우즈의 표준 도움말 윈도우이므로 별도의 사용 방법은 설명할 필요가 없을 것이다. 웹 페이지처럼 하이퍼 텍스트로 되어 있으므로 여기 저기 돌아다니면서 메인 도움말에 익숙해지도록 하자. 다 읽어 보지는 않더라도 최소한 어떤 내용이 어디쯤에 있다는 것은 파악해 놓는 것이 좋다. 그래야 의문나는 사항에 대해 빨리 빨리 찾아볼 수 있으며 문제를 신속하게 해결할 수 있다. 다음은 메인 도움말에 있는 도움말 목록이다.

**표**
델파이 도움말의 분류

| 범례 | 설명 |
|---|---|

| | |
|---|---|
| What's New in Delphi | 4.0 버전에 새로 추가된 사항에 대해 설명하고 있다. 기존 사용자들은 이 항목을 읽음으로써 4.0에 신속하게 익숙해질 수 있다. |
| Compatibility issues | 과거 버전에 비해 달라진 점, 주의할 점을 설명하고 있다. |
| Quick Start Tutorial | 간단한 데이터 베이스 프로그램을 단계별로 만들어 봄으로써 처음 델파이를 배우는 사람이 자습하기에 적당한 도움말이다. |
| Programming with Delphi | 델파이에서의 프로그래밍 방법에 대한 도움말이다. 컴포넌트, 파스칼 문법, 패키지 등에 대한 내용이 있다. |
| Developing Database Applications | DB 디자인, DB 연결, 필드, 테이블, 쿼리 조작 등 데이터 베이스 프로그래밍에 대한 내용이 들어있다. |
| Creating Custom Components | 인터넷 서버, 소켓, 코바 등 분산 환경 프로그래밍에 대한 주제를 다룬다. |
| Creating Custom Components | 델파이로 커스텀 컴포넌트를 만드는 방법을 설명하고 있다. 속성, 이벤트, 메소드, 메시지 다루는 방법 등을 자세하게 설명한다. |
| Developing COM-based Applications | COM 기초 이론과 COM 오브젝트 만드는 방법, 오토메이션, ActiveX 컨트롤 만들기 등에 대한 내용이 있다. |
| Object Pascal Reference | 델파이의 언어적 기반인 오브젝트 파스칼 언어의 문법에 대해 설명한다. |
| Visual Component Library Reference | 델파이 시스템 라이브러리인 VCL에 대한 도움말이다. VCL 기초와 유닛, 컴포넌트, 전역 함수에 대한 레퍼런스가 포함되어 있다. |
| Win32 Programmer's Reference | Win32 API 함수들과 OpenGL, OLE, MAPI 등에 대한 레퍼런스이며 마이크로소프트의 지식 베이스(Knowledge Base)도 포함되어 있다. |

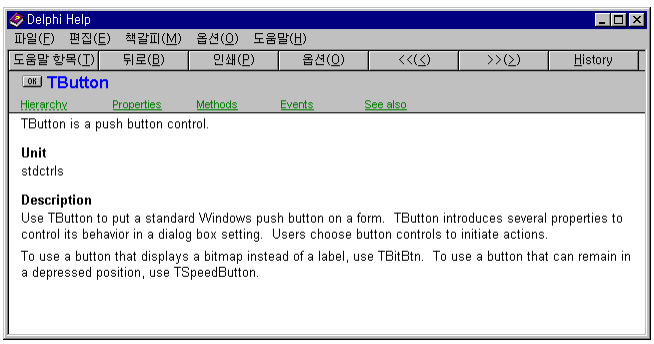
참고로 볼랜드사의 도움말은 경쟁사인 마이크로소프트의 도움말에 비해서는 한 수 아래다. 광범위하기는 하지만 읽기가 굉장히 불편하도록 되어 있으며 도움말 수준도 그리 만족할만하지 못하고 오타나 탈자 심지어 틀린 내용도 있다. 필자 개인적으로는 볼랜드의 도움말은 마이크로소프트의 그것에 비해 해석하기가 무척 어렵고 문장이 깔끔하지 않다고 생각한다. 게다가 볼랜드는 항상 제품을 발표하고 난 후에 도움말을 별도로 다시 패치하는 전과를 몇 번 저지른 적이

있다. 델파이 4.0 의 도움말에도 몇 가지 빠진 항목이 있고 틀린 내용도 꽤 있다. 통신망이나 웹 페이지를 통해 새로운 도움말을 구할 수 있는지 항상 관심을 가지도록 하고 최신 도움말을 구하도록 하자.

## 나. F1 키

F1 키도 물론 도움말이다. 메인 도움말과의 차이점은 원하는 정보를 빠르게 검색해 준다는 점이며 실전에서는 메인 도움말보다는 오히려 F1 키를 더 많이 사용한다. F1 키는 현재 사용자가 어떤 일을 하고 있으며 무엇을 알고 싶은가를 상황에 따라 판단하고 가장 적절한 도움말을 보여준다.

예를 들어 컴포넌트 팔레트에서 컴포넌트를 선택해 둔 채로 F1 키를 누르면 그 컴포넌트에 대한 도움말을 보여주며 오브젝트 인스펙터에서 속성을 편집하다가 F1 키를 누르면 그 속성에 대한 도움말을 보여준다. 특히 개별 컴포넌트에 관한 도움말을 얻을 때 이 키가 무척 유용하다. 다음 화면은 버튼 컴포넌트에 대한 도움말 화면이다.




도움말 본문 자체도 물론 도움이 되지만 위쪽에 속성, 메소드, 이벤트들을 별도로 정리해 놓아 이 도움말만 참조하면 버튼 컴포넌트에 대해 완전히 마스터할 수 있을 정도이다. 다음은 도움말 윈도우의 위쪽에 있는 Properties, Methods 를 눌렀을 때 나타나는 분리된 윈도우이다.

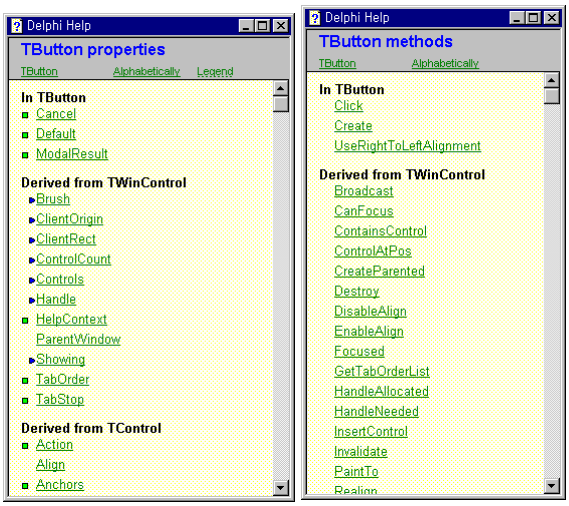
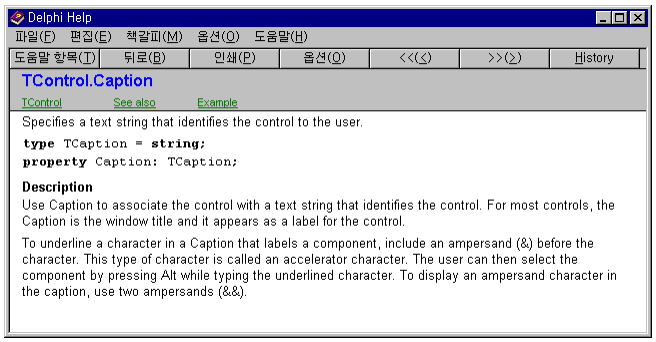



그림
속성, 메소드의 목록

이 윈도우에는 버튼 컴포넌트가 가진 속성, 메소드의 목록이 나타나며 여기서 속성이나 메소드를 선택하면 해당 항목에 대한 도움말을 보여준다. 다음은 버튼의 Caption 속성에 대한 도움말이다.



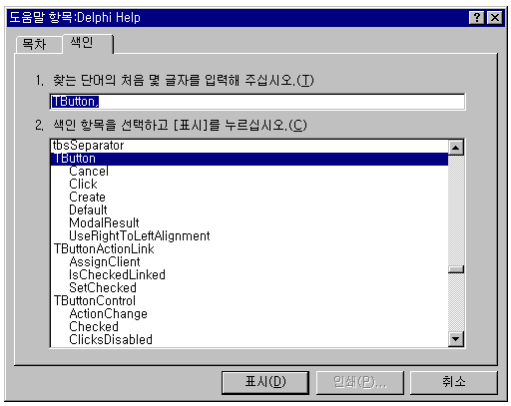

이런식으로 개별 컴포넌트들과 컴포넌트의 속성에 대한 도움말을 참고하면 델파이가 제공하는 모든 컴포넌트를 정복할 수 있을 것이다.

# 다. 색인

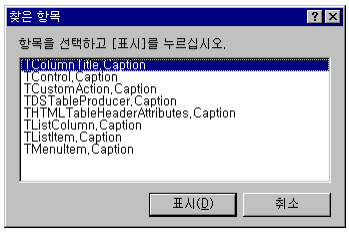
메인 도움말 화면의 색인 탭을 선택하면 알고자 하는 항목에 대해 빠른 속도로 검색을 할 수 있다. 색인 탭의 에디트 박스에서 도움을 얻고자 하는 항목을 입력하면 아래쪽 리스트 박스에 도움말 항목이 나타난다. 이 항목을 더블클릭하면 항목에 대한 도움말을 볼 수 있다. 예를 들어 TButton 이라고 치면 버튼 컴포넌

트에 대한 도움말이 나타난다.

그림
색인 페이지




만약 입력한 검색식에 해당하는 도움말 항목이 여러 개 있을 경우는 해당되는 모든 항목을 보여주는데 이 목록에서 보고자 하는 항목을 선택하면 된다. 예를 들어 Caption 이라고 치면 다음과 같은 목록이 나타난다.



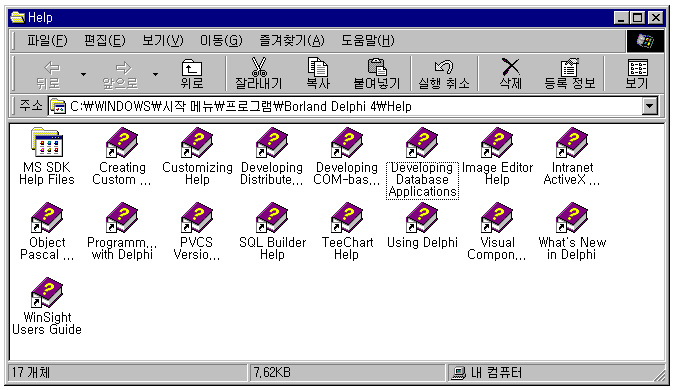

여러 가지 컴포넌트의 Caption 속성들이 있는데 이 중 보고자하는 항목을 선택하면 된다.

# 라. 기타

이외에도 델파이의 Help 폴더에는 몇 가지 독립된 도움말 파일들이 있다. 대부분 메인 도움말의 부속 도움말들이지만 그렇지 않은 것도 몇 가지 있으므로 읽어보기 바란다.

**그림**

델파이의 도움말 폴더

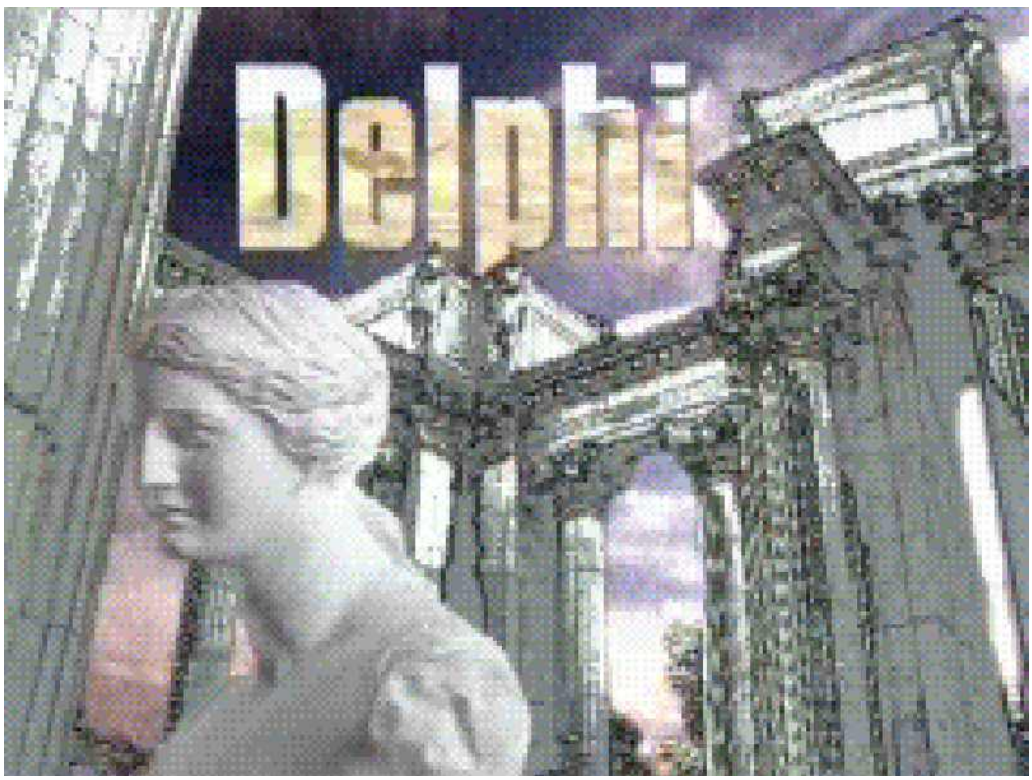



도움말 외에 도움을 얻을 수 있는 곳으로는 국내 대중 통신망의 델파이 동호회(하이텔의 vtool, 나우누리의 xbase 등)나 잡지 등이 있을 수 있고 델파이와 관련된 웹 사이트 등에서도 꽤 많은 도움말을 구할 수 있다. 특히 델파이를 만든 볼랜드의 www.inprise.com 은 자주 들러볼만한 곳이다.

필자가 한마디 잔소리를 하자면 델파이를 잘하고 싶은 생각이 조금이라도 있다면 도움말을 믿고 따라야 한다. 시중에 나와 있는 어떤 책도 델파이가 제공하는 도움말을 따라갈 수는 없다. 델파이를 만든 볼랜드사에서 작성한 도움말이니 만큼 빠진 내용이 없으며 또한 정확하고 신뢰할 만하다. (한 가지 흠이라면 약간의 사소한 오타가 있다.) 델파이에 관한 의문이 생기면 가장 먼저 도움을 줄 수 있는 것도 도움말이고 이책 저책 뒤져도 해결을 하지 못할 때 마지막으로 명쾌한 답을 줄 수 있는 것도 도움말이다.

# 제 3 장

# 컴포넌트



델파이는 여러 개의 컴포넌트를 조립하여 프로그램을 만드는 컴포넌트 기반의 개발 툴이므로 컴포넌트를 아는 것이 곧 델파이를 아는 것이 된다. 많고도 많은 컴포넌트를 한꺼번에 다 공부할 수는 없으므로 일단 이 장에서는 컴포넌트의 정의와 모든 컴포넌트에 두루 적용되는 일반적인 이론에 대해 먼저 공부해 보기로 하자. 개별 컴포넌트에 관해서는 7장에서 세부적으로 알아볼 것이다.

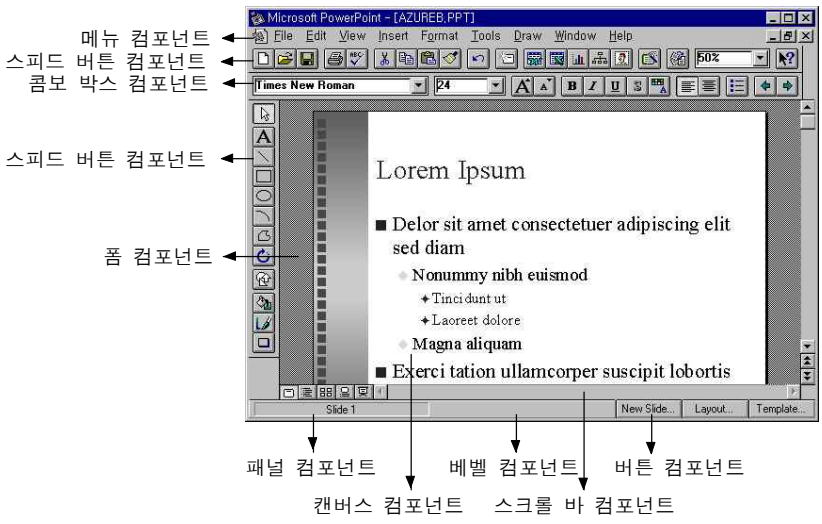
# 3-1 정의

## 가. 컴포넌트란

컴포넌트(Component)란 델파이 프로그램을 이루는 구성 요소(element)이며 컴포넌트를 모아 프로그램을 만든다. 마이크로소프트사에서 만든 프리젠테이션 제작용 프로그램인 파워 포인트를 살펴보면 다음과 같이 많은 컴포넌트들로 이루어져 있음을 알 수 있다.

**그림**

여러 가지 컴포넌트로 이루어진 윈도우즈의 응용 프로그램

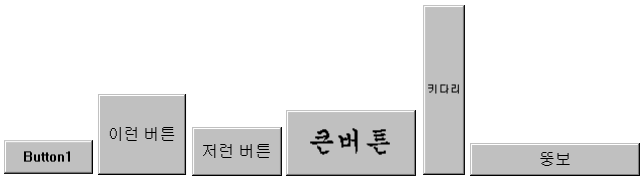



우선 윈도우 자체는 폼 컴포넌트이며 폼 안에는 메뉴, 버튼, 스크롤 바 등등의 컴포넌트들이 들어있다. 윈도우즈의 기본 프로그램인 탐색기, 문서 작성기, 계산기 등을 보아도 모두 비슷 비슷한 요소로 구성되어 있음을 확인할 수 있을 것이다.

컴포넌트는 개별적으로 독특한 특징을 가지며 또한 다른 것과 구별되는 독특한 행동 양식을 갖는다. 버튼 컴포넌트를 예로 든다면 제목, 위치, 크기 등의 속성을 가지며 마우스로 클릭할 경우 눌러진 모양으로 바뀐다는 행동 양식을 가진다.

**그림**

다양한 모양의 버튼 컴포넌트




또 폼은 배경색, 경계선의 모양 등 버튼에는 없는 속성이 있으며 타이틀 바를 드래그하면 위치를 옮기고, 경계선을 드래그하면 크기가 바뀌는 등의 독특한 행동 양식을 가진다.

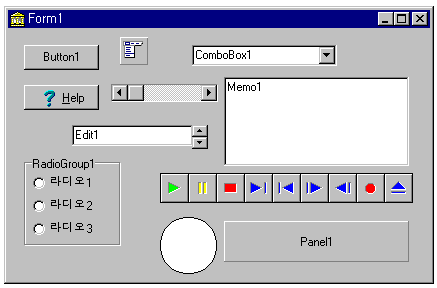
컴포넌트를 한마디로 쉽게 표현하면 "것(thing)"이라고 정의할 수 있다. 무엇인가 우리 눈에 보이고 나름대로의 개성을 가진 "물건(object)"을 뜻하는데 버튼도 물건이고 폼도 물건이고 레이블, 메뉴, 체크 박스 등도 모두 물건이다. 우리가 일상적으로 생각하는 물건과 별반 다를 것이 없다. 이런 여러 가지 구성 요소가 모이면 하나의 프로그램이 완성된다. 델파이가 기본적으로 제공하는 컴포넌트는 프로그래밍에 가장 자주 사용되는 170여 개이지만 새로운 물건이 필요하다면 얼마든지 더 만들어서 추가할 수 있다.

우리가 일상 생활에서 늘상 대하는 볼펜이라는 물건을 생각해 보자. 볼펜은 길이, 모양, 색상 등의 속성을 가지며 볼펜으로 종이를 문지르면 글이 써진다는 행동 양식을 가진다. 이런 일상 생활에서 보는 볼펜이라는 물건을 컴포넌트로 만들어 델파이에서 사용할 수도 있다. 볼펜 컴포넌트의 속성을 설정하여 굵은 볼펜, 긴 볼펜, 검정색 볼펜, 노랑색 볼펜 등을 만들 수 있고 볼펜을 사용하여 폼에 선을 그을 수도 있다.

물론 이 때의 볼펜은 실제 만져지는 볼펜과는 다른 논리적인 볼펜이다. 하지만 컴퓨터 내부에서 이 볼펜이 하는 짓거리나 모양은 실제의 볼펜과 거의 흡사하다. 볼펜뿐만 아니라 세상의 모든 물건들도 컴포넌트가 될 수 있다. 담배 컴포넌트, 돈 컴포넌트, 쓰레기통 컴포넌트, 심지어 사람 컴포넌트까지도 가능하다. 컴포넌트를 한마디로 정의하기는 무척 어려운 일이고 설사 문장화하여 정리하고 그 한마디를 외운다고 하더라도 컴포넌트를 완벽하게 이해할 수는 없다. 컴포넌트에는 버튼, 폼, 타이머, 리스트 박스, 패널,... 등이 있다고 알아두고 그들을 학습하는 과정에서 공통점을 발견하여 컴포넌트의 일반적인 정의를 익히는 것이

멀지만 더 확실한 길인 것 같다.

그림
델파이가 제공하는 다양한 컴포넌트





참고하세요

컴포넌트의 정의를 좀 더 문법적으로 내려보면 다음과 같다. 다음에 구체적으로 배우게 되겠지만 일단 구경이라도 해두자.

Compomennt is any class descended from TComponent.

컴포넌트란 TComponent에서 파생되어진 어떤 클래스이다.

클래스란 데이터 멤버와 멤버 함수를 가지는 데이터(속성)와 코드(행동 양식)의 덩어리이며 그 자체로 독립되어 있는 모듈이며 프로그램의 부품이다. TComponent라는 일반적 특성을 가진 클래스로부터 각 사물의 특수한 특성을 첨가하여 만든 것들이 컴포넌트이다.

# 나. 여러 가지 컴포넌트들

델파이가 제공하는 컴포넌트에는 어떤 것들이 있는지 알아 보자. 컴포넌트들은 컴포넌트 팔레트에 아이콘 형태로 옹기종기 모여 있으며 일부는 팔레트에 없는 숨겨진 것들도 있다. 다음은 가장 많이 사용되고 먼저 알아야 할 기본적인 컴포넌트들이다. 당장 외울 필요까지는 없고 일단 컴포넌트의 이름과 아이콘만 대충 봐 두고 첫 대면이라도 해 두도록 하자.

표
기본적인 컴포넌트

| 페이지 탭 | 아이콘 | 이름 | 기능 |
|---|---|---|---|
| standard |  | MainMenu | 메인 메뉴 |
| standard |  | PopupMenu | 팝업 메뉴 |

| | | | |
|---|---|---|---|
| standard |  | Label | 문자열을 출력한다. |
| standard |  | Edit | 문자열을 입력받는다. |
| standard |  | Memo | 문장을 편집한다. |
| standard |  | Button | 명령을 입력받는다. |
| standard |  | CheckBox | 옵션을 입력받는다. |
| standard |  | RadioButton | 선택을 입력받는다. |
| standard |  | ListBox | 여러 개 중 하나를 선택한다. |
| standard |  | ComboBox | 선택하거나 직접 입력한다. |
| standard |  | ScrollBar | 화면 영역을 이동시킨다. |
| standard |  | GroupBox | 여러 개의 컴포넌트를 묶는다. |
| standard |  | RadiGroup | 라디오 버튼을 만든다. |
| standard |  | Panel | 폼의 영역을 분할한다. |
| standard |  | ActionList | 명령을 한 곳에 모은다. |
| additional |  | BitBtn | 그림이 그려진 버튼 |
| additional |  | SpeedButton | 그림이 그려진 버튼 |
| additional |  | Image | 그래픽을 출력한다. |
| additional |  | Shape | 간단한 도형을 출력한다. |
| system |  | Timer | 시계 |
| Win3.1 |  | FileListBox | 파일의 목록을 보여준다. |
| Win3.1 |  | DirectoryListBox | 디렉토리의 목록을 보여준다. |
| Win3.1 |  | DriveComboBox | 드라이브를 선택한다. |

이 많은 컴포넌트들을 언제 다 공부하느냐고 하겠지만 사실 외워두고 사용할 만한 컴포넌트는 고작 십여 개 정도에 불과하며 나머지는 대충만 알아도 레퍼런스를 참고해 가며 쓸 수 있다. 각각의 컴포넌트는 이 책 전반을 걸쳐 예제와 함께 하나하나 격파해 나가도록 하자.

# 다. 컴포넌트 조작

컴포넌트는 사용자가 마음대로 요리할 수 있는 조작의 대상이다. 폼에 놓을 수도 있고 위치나 크기를 변경할 수도 있고 삭제할 수도 있다. 디자인중에 컴포넌트를 마음대로 변경할 수 있기 때문에 비주얼 툴이라고 하며 그래서 사용하기 쉬운 것이다. File/New Application을 선택해 새 프로젝트를 하나 준비해 놓고 책을 읽으며 컴포넌트를 조작하는 방법을 실습해 보도록 하자. 막상 해보면 아주 쉽고 재미있을 것이다.

## ■ 컴포넌트 배치

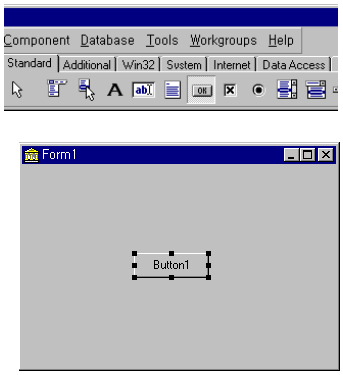
컴포넌트를 폼에 배치하는 방법은 네 가지로 분류할 수 있지만 네 가지 방법이 특별히 다른 것은 아니며 대동소이하다. 어떤 방법을 사용하더라도 효과는 동일하므로 그때 그때의 상황에 따라 가장 편한 방법을 골라 사용하면 된다.

❶ 팔레트에서 원하는 컴포넌트를 클릭한 후 폼을 클릭한다.

이 방법은 2장에서 Fruit 예제를 만들 때 한번 실습해 본 방법이다. 팔레트에서 배치하고자 하는 컴포넌트를 클릭하여 선택한 후 폼에서 컴포넌트를 배치하고자 하는 위치를 한번 더 클릭하는 방법이다. 이 때 폼에 놓여지는 컴포넌트의 크기는 디폴트 크기이며 사용하기에 가장 무난한 수준의 크기이다. 물론 배치하고 난 후에 마음대로 크기나 위치를 바꿀 수 있다.

**그림**

컴포넌트를 폼에 배치하는 방법




배치하고자 하는 컴포넌트를 팔레트에서 선택한다.

배치할 위치에서 클릭한다.

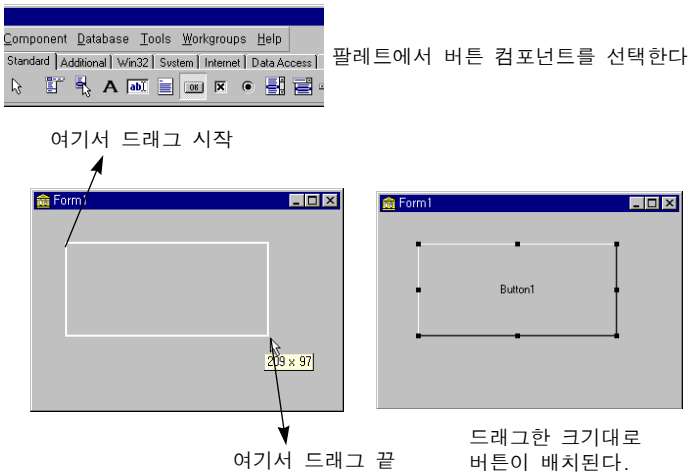
위 그림은 컴포넌트 팔레트에서 버튼을 선택한 후 폼에 배치해 본 것이다. 이것 저것 컴포넌트들을 배치해 보도록 하자.

❷ 팔레트에서 컴포넌트를 클릭한 후 폼에서 드래그한다.

팔레트에서 컴포넌트를 선택하는 것은 앞의 경우와 동일하지만 컴포넌트를 폼에 배치할 때 단순히 클릭하는 것이 아니라 원하는 크기만큼 드래그한다. 드래그할 때 굵은 회색선이 그어지는데 적당한 크기에서 마우스 버튼을 놓으면 드래그한 크기대로 컴포넌트가 배치된다. 이 방법은 컴포넌트를 폼에 놓음과 동시에 크기를 마음대로 설정할 수 있는 방법이다.

**그림**

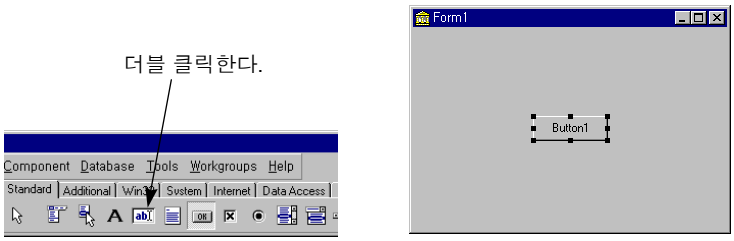
드래그하여 배치와 동시에 크기를 지정하는 방법




❸ 팔레트에서 원하는 컴포넌트를 더블클릭한다.

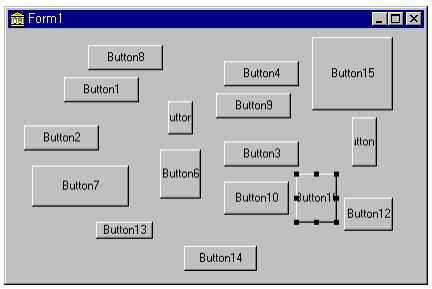
컴포넌트를 배치하는 가장 간단한 방법이다. 더블클릭한 컴포넌트가 폼의 중앙에 디폴트 크기로 배치된다. 무조건 폼의 중앙에 놓이기 때문에 배치한 후 원하는 위치로 이동해야 한다.

**그림**
더블 클릭하여 폼의 중앙에 컴포넌트를 배치하는 방법




❹ 같은 컴포넌트를 여러 번 배치할 때는 컴포넌트 팔레트에서 컴포넌트를 선택할 때 Shift키를 누른 채로 선택한다. 그냥 클릭하여 컴포넌트를 선택하면 컴포넌트를 배치한 후 선택이 해제되지만, Shift키를 누른 채로 클릭하면 선택을 계속 유지해 주므로 같은 컴포넌트를 여러 개 배치할 수 있다. Shift키를 눌러 컴포넌트를 선택했을 때는 팔레트의 컴포넌트 주변에 파란색의 테두리가 쳐진다. 배치가 끝나면 컴포넌트 팔레트의 제일 왼쪽에 있는  를 눌러 선택을 직접 해제해 주어야 한다. 다음 그림은 팔레트에서 버튼 컴포넌트를 Shift 클릭한 후 폼에 여러 번 배치해 본 것이다.

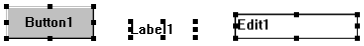
**그림**
같은 컴포넌트를 여러번 배치하는 방법




## ■ 선택

선택이란 작업 대상이 될 컴포넌트를 지정하는 동작이며 워드 프로세서에서 작업 대상이 될 문장을 블럭으로 선택하는 것과 개념적으로 동일하다. 컴포넌트의 속성을 바꾸거나 삭제하고자 할 때 먼저 해당 컴포넌트를 선택해야 한다. 컴포넌트 하나를 선택할 때는 폼에 놓여진 컴포넌트를 클릭하기만 하면 된다. 선택된 컴포넌트 주변에는 8개의 크기 조절 핸들(sizing handle)이 표시되어 현재 작업 대상 컴포넌트임을 나타낸다.

**그림**
크기 조절 핸들



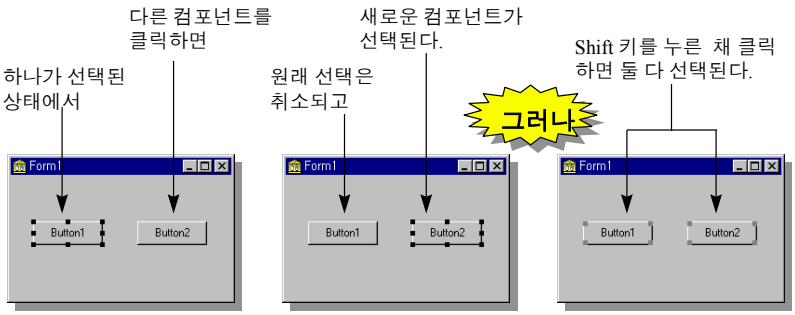

오브젝트 인스펙터는 선택된 컴포넌트의 속성을 보여준다. 버튼, 레이블, 에

디트 등을 배치해 놓고 각 컴포넌트를 선택해 보면 오브젝트 인스펙터에 나타나는 속성 목록이 달라질 것이다.

여러 개의 컴포넌트를 작업 대상으로 선택할 때는 Shift 키를 누른 채로 클릭한다. 그냥 클릭하면 이전에 선택되어 있던 컴포넌트의 선택을 취소시키지만 Shift 클릭은 이전 선택을 유지한 채로 새로운 컴포넌트를 덧붙여 선택한다. 선택된 여러 개의 컴포넌트 주변에는 희미한 핸들이 나타난다.

**그림**

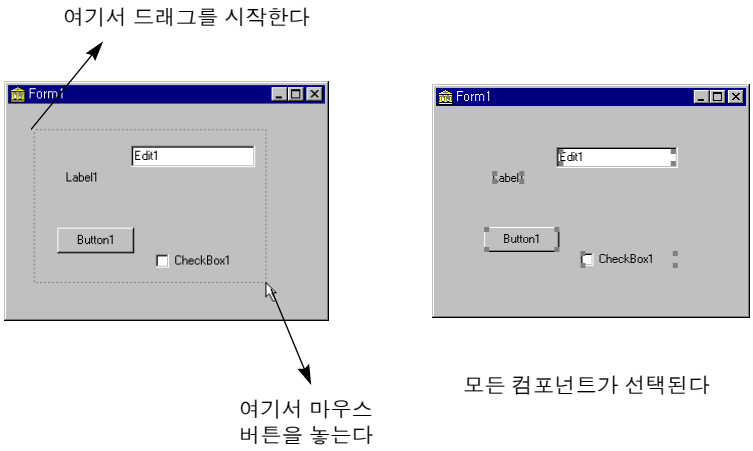
Shift 키를 사용하여 여러 개의 컴포넌트를 선택한다.




여러 개의 컴포넌트를 한꺼번에 선택하려면 폼에서 마우스를 드래그하여 직사각형으로 컴포넌트를 둘러싼다. 마우스로 드래그하여 만든 직사각형 영역에 일부분이라도 포함된 컴포넌트가 모두 선택된다. 이렇게 사각 영역을 드래그하여 영역 안의 모든 요소를 선택하는 방법을 마키 실렉션(marquee selection)이라고 하며 대부분의 윈도우즈용 응용 프로그램에서 선택을 위해 사용하는 표준적인 방법이다.

**그림**

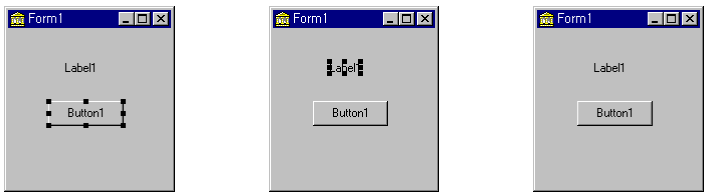
일정 범위의 컴포넌트를 모두 선택하는 마키 셀렉션




선택된 컴포넌트는 모두 작업 대상이 되며 여러 개의 컴포넌트를 선택한 후

한꺼번에 이동시키거나 삭제할 수 있다. 마키 실렉션을 할 때도 Shift 키를 사용하여 기존의 선택을 유지한 채로 다른 컴포넌트를 한꺼번에 선택하는 것이 가능하다. 그 외의 선택 방법으로는 컨테이너 내부에서 Ctrl 키를 사용하여 여러 개를 선택하는 방법과 가려진 컴포넌트를 선택하는 방법이 있는데 이 방법은 차후에 컨테이너를 공부한 후에 별도로 연구해 보도록 하자.

참고로 폼도 하나의 컴포넌트인데 폼을 선택하고자 할 경우는 폼의 아무 곳이나 클릭하여 폼 외의 다른 컴포넌트가 선택되지 않도록 하면 된다. 즉 아무것도 선택되어 있지 않은 상태는 폼이 선택된 것으로 간주하며 오브젝트 인스펙터에는 폼의 속성들이 나타난다. 폼은 경계선으로 크기를 조정하며 타이틀 바를 드래그하여 위치를 옮기므로 선택되더라도 크기 조절 핸들은 보이지 않는다.



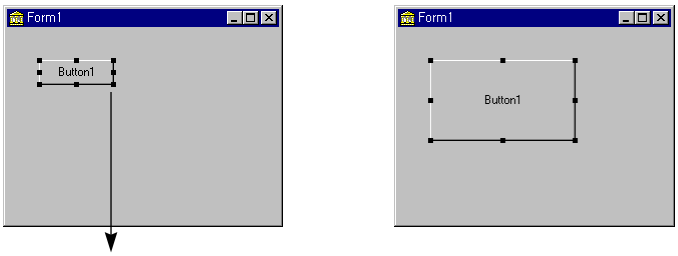

버튼이 선택된 상태 레이블이 선택된 상태 폼이 선택된 상태

## ■ 크기 조절

버튼, 에디트 박스, 레이블 등 대부분의 컴포넌트는 크기를 마음대로 변경할 수 있다. 물론 메뉴나 타이머와 같이 크기 변경이 안되는(크기 조정을 할 필요가 없는) 컴포넌트도 있다. 컴포넌트의 크기를 변경하기 위해서는 먼저 변경의 대상이 되는 컴포넌트를 선택해야 한다. 선택된 컴포넌트 주변에는 8개의 크기 조절 핸들이 표시된다. 이 핸들 위로 마우스 커서를 가져가면 커서가 ↕ ↔ ↘ ↙ 요렇게 변하며 이 핸들을 드래그함으로써 컴포넌트의 크기를 마음대로 조절한다.

**그림**

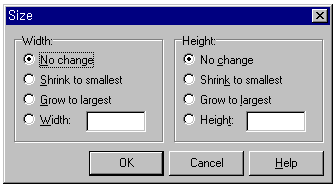
크기 조절 핸들을 드래그하여 크기를 변경한다.



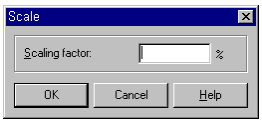

핸들을 드래그한다. 크기가 변경된다.

각 변의 핸들은 한쪽 크기만 조절하며 모서리의 핸들은 수평, 수직 크기를 동시에 조절한다. 윈도우의 경계선을 드래그하여 크기를 조정하는 방법과 동일하다. 단 크기 조절은 여러 개의 컴포넌트를 선택해 두고 한꺼번에 할 수 없다. 마우스로 드래그해야 할 대상이 애매해지기 때문이다. 조금만 생각해 보면 왜 안되는지 쉽게 이해가 갈 것이다. 그러나 대상 컴포넌트의 크기가 모두 같다면 대화상자를 통해 똑같은 크기로 맞출 수는 있다. 복수 개의 컴포넌트를 선택한 후 Edit/Size...를 선택하면 크기 조절 대화상자가 열린다.

그림
크기 조절 대화상자




이 대화상자를 사용하여 선택된 모든 컴포넌트에 대해 폭과 높이를 개별적으로 또는 폭과 높이를 동시에 조절할 수 있다. 폼 컴포넌트의 경우는 별도의 크기 조절 핸들이 나타나지 않으므로 경계선을 드래그하여 크기를 변경하면 된다. 그리고 잘 사용되지는 않지만 Edit/Scale을 선택하여 모든 컴포넌트의 크기를 일정 배율로 증감시키는 것도 가능하다.

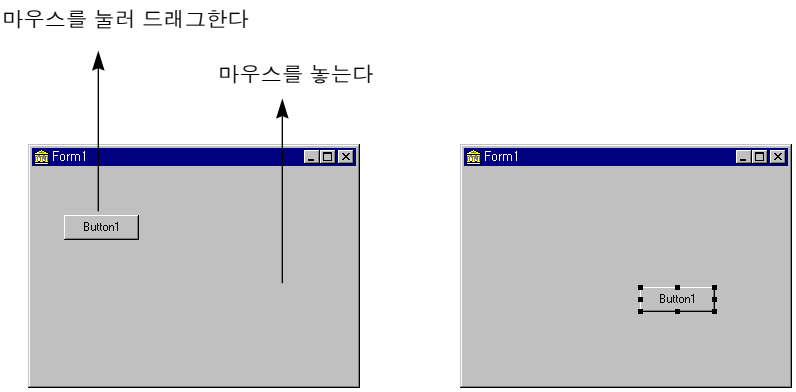



100% 이상의 값을 입력하면 컴포넌트의 크기를 일정 배율로 확대시키며 100% 이하의 값을 입력하면 반대로 축소시킨다.

## ■ 이동

컴포넌트의 위치를 이동시키려면 컴포넌트를 마우스로 드래그한다. 이동시키고자 하는 컴포넌트를 마우스로 끌어다 원하는 위치에 놓기만 하면 된다. 윈도우의 타이틀 바를 드래그하여 위치를 이동시키는 것과 동일하다. 여러 개의 컴포넌트를 선택하여 한꺼번에 이동시키는 것도 물론 가능하다.

그림
컴포넌트 이동 시키기




이동 후의 모습

컴포넌트를 이동시키는 또 다른 방법은 오브젝트 인스펙터를 통하여 Left, Top 속성에 좌표를 직접 입력하는 방법이 있다. 이 속성을 변경하면 컴포넌트의 위치가 변경되며 반대로 컴포넌트를 직접 드래그하여 위치를 옮기면 이 속성도 변한다. 폼은 타이틀 바를 드래그하여 위치를 이동시킨다.

## ■ 삭제

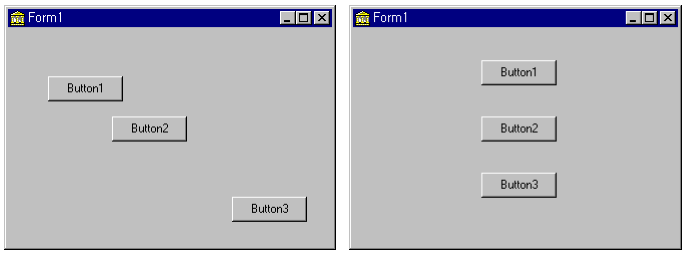
폼에 이미 배치된 컴포넌트를 삭제할 때는 키보드의 Del 키를 사용한다. 물론 Del 키를 누르기 전에 삭제할 컴포넌트가 선택되어 있어야 한다. 컴포넌트를 선택한 후 Del 키를 누르기만 하면 감쪽같이 사라진다. 만약 마우스가 너무 좋아 키보드에 가급적 손 안대고 싶은 사람은 메뉴에서 Edit/Delete 항목을 선택해도 된다. 여러 개의 컴포넌트를 선택한 상태로 Del 키를 누르면 선택된 모든 컴포넌트가 삭제된다.

단축키 Ctrl-Z는 모든 윈도우즈 프로그램에 공통적인 것이다.

만약 실수로 삭제하지 않아야 할 컴포넌트를 삭제했다면 Edit/Undelete 항목을 선택해 복구할 수 있다. 삭제한 컴포넌트를 복구하면 그 컴포넌트에 지정된 속성이나 작성한 코드도 함께 복구된다. 단축키는 Ctrl-Z이므로 혹시 실수할 경우를 대비해 잘 알아 두도록 하자.

## ■ 정렬

좌측과 같이 불규칙한 형태로 배치되어 있는 버튼들은 화면 한가운데로 모으거나 특정한 기준선에 맞추어 주는 것이 훨씬 더 깔끔해 보인다.




물론 컴포넌트를 보기좋게 정렬하는 일은 일종의 이동 조작이므로 마우스로 직접 이동시켜 수동으로 정렬을 할 수 있지만 컴포넌트의 개수가 많을 때는 너무 성가신 일이며 정밀하게 정렬하기가 어렵다. 델파이가 제공하는 정렬 기능의 도움을 받으려면 View 메뉴의 Alignment Palette를 선택한다. 다음과 같이 정렬 팔레트 윈도우가 열릴 것이다.

**그림**
정렬 팔레트


10개의 버튼으로 구성되어 있는데 버튼의 그림이 워낙 설명적이고 풍선 도움말까지 제공되므로 쉽게 이해가 갈 것이다. 버튼을 눌러보면 버튼의 그림이 정렬된 후의 모양으로 바뀌므로 버튼만 보고도 어떤 의미인가를 쉽게 알 수 있도록 되어 있다.

**표**
정렬 팔레트의 버튼

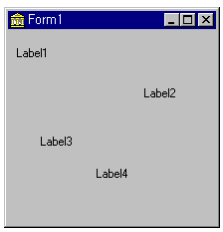
| 버튼 | 누른 후 | 정렬 |
|---|---|---|
|  |  | 선택한 컴포넌트들을 좌측 정렬한다. |
|  |  | 선택한 컴포넌트들을 수평 중앙 정렬한다. |
|  |  | 폼의 수평 중앙에 오도록 한다. |
|  |  | 컴포넌트간의 수평 간격이 일정하도록 한다. |
|  |  | 선택한 컴포넌트들을 우측 정렬한다. |
|  |  | 선택한 컴포넌트들을 상단 정렬한다. |
|  |  | 선택한 컴포넌트들을 수직 중앙 정렬한다. |


폼의 수직 중앙에 오도록 한다.

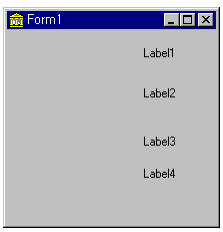

컴포넌트간의 수직 간격이 일정하도록 한다.


선택한 컴포넌트들을 하단 정렬한다.

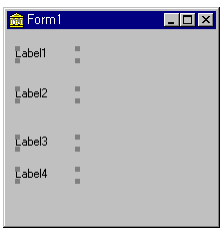
정렬 기능은 선택된 컴포넌트에 대해서만 동작하므로 정렬을 하기 전에 먼저 대상 컴포넌트를 선택해야 한다. 이 때 먼저 선택되는 컴포넌트가 정렬의 기준이 되므로 선택의 순서에 신경을 써야 한다. 네 개의 레이블이 폼에 불규칙하게 배열되어 있을 때 좌측변을 동일하게 맞추려고 한다. 이때 제일 우측에 있는 Label2를 먼저 선택하고 정렬한 결과와 제일 좌측에 있는 Label1을 먼저 선택하고 정렬한 결과가 다르게 나타난다.




원래의 배치 상태




Label2 기준

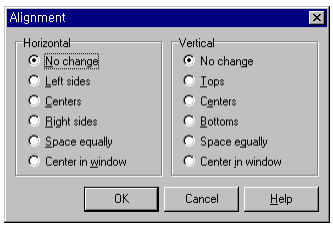



Label1 기준

제일 먼저 선택한 컴포넌트를 기준으로 하여 정렬된다는 것을 알 수 있다. 컴포넌트를 정렬하는 또 다른 방법은 정렬 대화상자를 사용하는 방법이다. Edit 메뉴의 Align...을 선택하면 정렬 대화상자가 나타난다.

**그림**

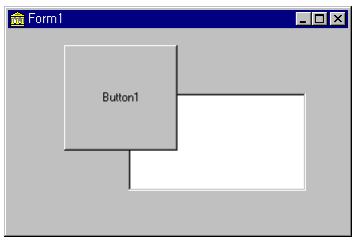
정렬 대화상자




이 대화상자를 사용하면 수평과 수직 정렬을 한꺼번에 수행할 수 있다.

## ■ 앞뒤로 이동

컴포넌트를 폼에 배치하다 보면 때로는 두 개의 컴포넌트가 같은 위치에서 겹쳐지는 경우가 발생한다. 이때는 먼저 배치한 컴포넌트가 나중에 배치한 컴포넌트에 의해 가려지게 된다. 다음 그림을 보면 메모 컴포넌트와 버튼 컴포넌트가 겹쳐져 있는데 메모 컴포넌트의 일부가 버튼에 의해 가려져 있다. 왜냐하면 버튼이 위치적으로 더 위쪽에 있기 때문이다.

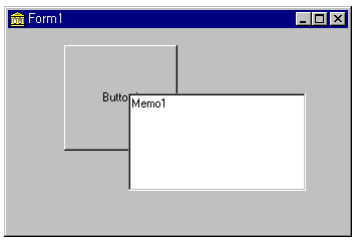



이런 경우는 겹쳐져 있는 컴포넌트들의 우선 순위를 명확하게 정해 주어야 하며 Edit 메뉴의 Send to Back, Bring to Front 두 명령을 사용한다. Bring to Front는 선택된 컴포넌트를 앞쪽으로 이동하여 드러나도록 해 주며 반대로 Send to Back은 컴포넌트를 뒤쪽으로 이동시킨다. 버튼보다 메모 컴포넌트를 더 위쪽에 두고 싶다면 다음 두 가지 방법 중 한 가지를 선택한다.

① 버튼 컴포넌트를 선택한 후 Send to Back 명령을 선택한다. 그러면 버튼이 메모 컴포넌트보다 뒤쪽으로 이동하여 메모 컴포넌트가 드러난다.

② 메모 컴포넌트를 선택한 후 Bring to Front 명령을 선택한다. 메모 컴포넌트가 버튼보다 앞쪽으로 이동한다.

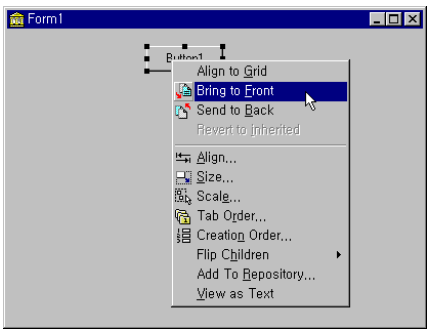
두 명령 중 어떤 방법을 사용하더라도 상관이 없다. 이동 후의 폼은 다음과 같은 모양이 된다.



사실 버튼과 메모 컴포넌트는 이렇게 위치가 겹칠 경우가 거의 없지만 배경 그림을 출력하는 이미지 컴포넌트나 폼의 장식에 사용되는 셰이프, 베벨 컴포넌트의 경우는 다른 컴포넌트와 겹치는 일이 빈번하므로 컴포넌트의 앞뒤를 조정해 주어야 할 필요가 있다.

## ■팝업 메뉴 사용하기

컴포넌트를 배치하고 이동하고 지우는 등의 조작은 델파이의 가장 기본적인 사용 방법이므로 실습을 통해 잘 익혀 두도록 하자. 윈도우즈를 조금이라도 사용해 본 사람이라면 아주 쉽게 익힐 수 있을 것이다. 참고로 앞에서 설명한 컴포넌트 조작 방법은 개별 컴포넌트에서 오른쪽 버튼을 누르면 나타나는 팝업 메뉴로도 할 수 있다. 참고하도록 하자.





① 컴포넌트:프로그램을 이루는 구성 요소
② 팔레트에서 컴포넌트를 선택하고 폼을 클릭하여 컴포넌트를 배치한다.
③ Shift 키를 사용하여 여러 개의 컴포넌트를 선택한다.
④ 핸들을 드래그하여 크기를 조절한다.
⑤ 컴포넌트를 드래그하여 컴포넌트를 이동시킨다.

# 3-2 속성

## 가. 속성

속성(Property)이란 컴포넌트의 특징을 정의하는 여러 가지 값들이다. 디자인 과정에서 속성을 정의하는 것이 보통이지만 실행시에도, 즉 프로그램이 실행되는 동안에도 속성을 바꿀 수 있다. 앞에서 우리가 처음으로 만들어 보았던 Fruit 예제에서 Fruit 레이블의 Caption 속성이 디자인 과정에서 Fruit로 정의되었다가 실행시에 버튼이 눌러지면 Apple이 되기도 하고 Orange가 되기도 하였다.

어떤 속성이 정의되어 있는가는 컴포넌트마다 다르다. 컴포넌트마다 제각각 생긴 모양과 하는 동작이 다르므로 지정해 줄 수 있는 속성도 다양할 수밖에 없다. 특정 컴포넌트에 어떤 속성이 있는지 보고 싶으면 컴포넌트를 선택한 후 오브젝트 인스펙터를 살펴보면 된다. 폼에서 선택한 컴포넌트에 따라 오브젝트 인스펙터에 나타나는 속성 목록이 달라진다. 좌측부터 차례로 버튼, 레이블, 폼, 에디트 컴포넌트의 속성이다. 공통적으로 포함되는 속성도 있지만 특정 컴포넌트에만 있는 속성도 있다.

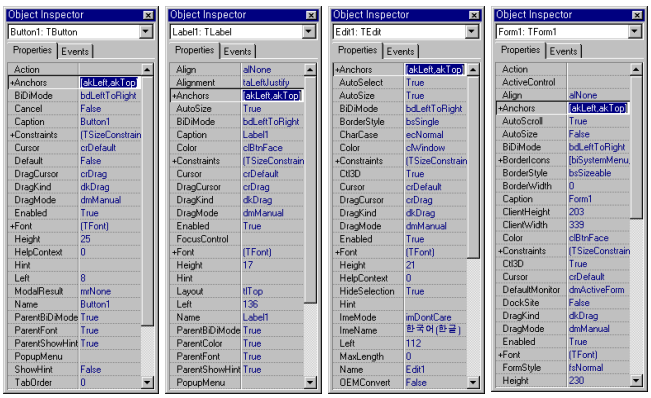



**그림**
컴포넌트별로 갖는 속성이 다르다.

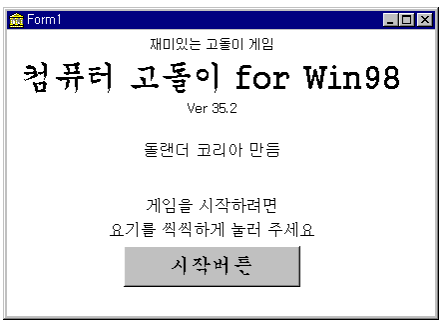
## 나. 레이블 컴포넌트

속성에 대해 연구해 보기 전에 우선 제일 만만한 레이블 컴포넌트 하나를 조

금만 연구해 보자. 그래야 속성에 대해서도 쉽게 이해할 수 있을 것이다. 컴포넌트 팔레트의 Standard 페이지에 A라고 되어 있는 컴포넌트가 바로 레이블이다.


레이블(label) 컴포넌트는 폼상에 문자열을 위치시키고자 할 때 사용하며 레이블 컴포넌트에 표시되는 문자열은 Caption 속성으로 정해진다. 레이블을 폼에 위치시키면 디폴트로 Label1이라는 캡션이 주어지는데 이 값은 디자인 과정에서나 실행 과정에서 마음대로 바꿀 수 있다. 레이블은 다른 컴포넌트에 비해 별다른 기능이 없으며 Caption 속성으로 지정된 문자열을 폼에 보여주는 것이 기능의 거의 전부이다.

그림
레이블의
사용예




위 윈도우에서 시작 버튼을 제외하고 나머지는 모두 레이블이다. 버튼이야 누르는 기능이 있지만 레이블은 그냥 문자열을 보여주기만 할 뿐 거의 아무런 기능도 가지고 있지 않으며 보통 코드를 작성하지 않는다. 그래서 제일 쉽게 접근할 수 있는 컴포넌트이다.

컴포넌트의 속성 중 모든 컴포넌트에 있고 제일 먼저 설정해 주어야 할 속성은 Name 속성이다. Name 속성은 컴포넌트의 이름을 정의하며 이 이름은 프로그램 내의 다른 부분에서 이 컴포넌트를 칭할 때 사용하는 일종의 구분 기호이다. 다른 컴포넌트와 구분하는 것이 Name 속성의 목적이므로 컴포넌트끼리 Name 속성이 절대로 중복되어서는 안되며 델파이가 그렇게 하도록 내버려 두지도 않는다.

디폴트로 주어지는 Name 속성은 Label1, Button1 등과 같이 컴포넌트 유형과 정수 하나로 구성되어 있다. 레이블이 같은 폼안에 계속 만들어지면 Label1, Label2, Label3 등과 같이 일련번호를 사용하여 중복되지 않도록 이름을 붙여 나간다. 이렇게 디폴트로 주어지는 이름을 그대로 사용하는 경우는 별로 없으며 보통은 사용자가 기억하기 쉽고 의미가 있는 이름을 준다. 예를 들어 레이블이

파일 이름을 나타내는 목적에 사용된다면 LabelFileName이라는 이름을 주고 버튼이 어떤 동작의 시작을 지시한다면 Start라는 이름을 주는 것이 좋다. Caption 속성과 Name 속성은 비슷한 것 같지만 확실히 다른 속성이므로 잘 구분하도록 하자.

✍ **Caption**

밖으로 드러나 보이는 제목이다. 레이블의 문자열, 버튼의 제목, 폼의 타이틀 바에 나타나는 제목 등이 모두 Caption 속성이다. 어디까지나 사람에게 보여주기 위한 문자열이므로 그야말로 마음대로 작성해도 된다. 사용할 수 있는 문자가 제한되어 있지도 않으며 한글도 마음대로 쓸 수 있다.

✍ **Name**

프로그램 내부에서 컴포넌트를 칭하는 이름이다. 프로그램이 컴포넌트를 제어할 때 사용하는 이름이므로 다른 컴포넌트의 Name 속성과 절대로 중복되어서는 안되며 코드에서 사용되므로 이름을 붙일 때도 규칙에 맞게 붙여야 한다. 파스칼 예약어를 쓴다거나 중간에 공백을 넣거나 숫자로 시작해서는 안되며 한글을 사용해서도 안된다.

Name과 Caption은 같을 수도 있고 다를 수도 있는데 어디까지나 프로그램 개발자의 편의대로 값을 준다. 컴포넌트가 처음 만들어지면 Name 속성과 같은 이름으로 Caption 속성을 설정해 주는데 이것은 프로그래머가 폼에서 쉽게 컴포넌트를 선택할 수 있도록 하기 위한 델파이의 배려일 뿐이며 사용자가 얼마든지 다른 캡션으로 변경할 수 있다.


① 속성:컴포넌트의 특성을 정의하는 여러 가지 값들이다.
② 오브젝트 인스펙터를 사용하여 속성을 변경한다.
③ 레이블:폼에 문자열을 출력하는 컴포넌트
④ Caption:컴포넌트의 제목을 지정하는 속성
⑤ Name:컴포넌트 고유의 명칭이며 중복되어서는 안된다.

# 다. 일반적인 속성

어떤 속성이 있는가는 컴포넌트에 따라 달라진다. 어떤 속성은 특정한 컴포넌트에만 있는 경우도 있고 Name, Tag 등과 같이 모든 컴포넌트에 다 있는 속성

도 있다. 컴포넌트별로 존재하는 개별적인 속성은 각각의 컴포넌트에만 국한되므로 특정 컴포넌트를 공부할 때 같이 공부하는 것이 정상적인 순서이다. 여기서는 모든 컴포넌트에 있는 가장 일반적인 속성에 관해서만 설명하기로 한다. 델파이를 처음 배우는 사람이면 가급적 외우도록 하는 것이 좋다.

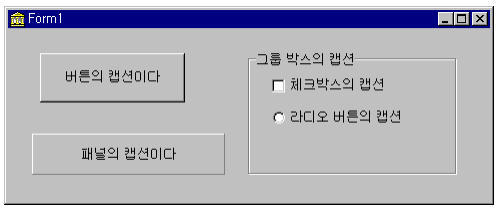
### ✍ Name

컴포넌트의 이름이며 이 이름은 프로그램 내부에서 컴포넌트를 참조하기 위해 사용된다. 앞에서 강조한대로 다른 컴포넌트의 Name 속성과 중복되어서는 안되며 고유의 이름을 가져야 한다.

### ✍ Caption

컨트롤의 제목을 나타내는 문자열이며 컨트롤에 따라 제목이 나타나는 부분이 다르다. 폼의 경우 윈도우의 타이틀 바나 최소화되었을 때의 작업 표시줄에 나타나며 버튼의 경우 버튼 위에 나타난다.

그림
컴포넌트에 따른 캡션의 위치




### ✍ Alignment

컨트롤 내에서 문자열이 정렬되는 방식을 지정한다. 다음과 같은 속성값을 줄 수 있다.

| 속성값 | 의 미 |
|---|---|
| taLeftJustify | 텍스트를 컨트롤의 좌측에 정렬한다. |
| taCenter | 텍스트를 컨트롤의 중앙에 정렬한다. |
| taRightJustify | 텍스트를 컨트롤의 우측에 정렬한다. |

디폴트 정렬은 좌측이다. 다음에 세 가지 정렬 예를 보면 이 속성이 어떤 용도로 사용되는지를 알 수 있다.

좌측 정렬

중앙 정렬

우측 정렬

### ✍ Color

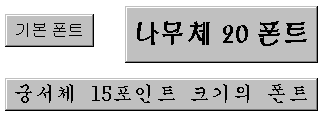
컴포넌트의 배경 색상을 정의한다. 다음 색상 중 한 가지를 선택하거나 시스템에 정의된 색상을 쓸 수도 있다.

| 속성값 | 색 상 |
|---|---|
| clBlack | 검정색 |
| clMaroon | 고동색 |
| clGreen | 초록색 |
| clOlive | 흐린 녹색 |
| clNavy | 짙은 파란색 |
| clPurple | 자주색 |
| clTeal | 감청색 |
| clGray | 회색 |
| clSilver | 은색 |
| clRed | 빨강 |
| clYellow | 노랑색 |
| clLime | 연두색 |
| clBlue | 파랑 |
| clFuchsia | 분홍색 |
| clAqua | 하늘색 |
| clWhite | 흰색 |

Color 속성의 속성값인 clRed, clWhite 등은 프로그램 코드에서도 그대로 사용할 수 있는 색상 이름이다. 색상 이름에 접두어 cl을 붙여서 표현한다는 것을 기억해 두도록 하자. 접두어 cl은 Color의 준말이다.

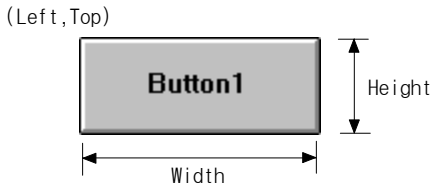
### ✍ Font

컨트롤 상에 표시되는 문자의 글꼴을 지정한다. 주로 Caption의 글꼴을 지정하나 꼭 그런 것은 아니다. 폼의 경우는 Font 속성을 조정한다고 해서 타이틀 바에 나타나는 캡션의 글꼴이 바뀌는 것은 아니며 폼에 출력되는 문자열의 글꼴이 바뀐다.




### ✍ Left,Top,Height,Width

이 속성들은 컴포넌트의 위치와 크기를 지정한다. (Left,Top)은 폼의 좌상단을 기준으로 하여 컴포넌트의 좌표를 지정하며 Height, Width는 컴포넌트의 폭과 높이를 나타낸다. 단 폼의 경우는 전체 화면을 기준으로 하여 위치를 나타낸다. 이 속성들의 단위는 픽셀이다.




크기와 위치를 나타내는 이 속성들은 오브젝트 인스펙터에서 직접 값을 입력하는 경우보다 폼 상에서 컴포넌트를 드래그하여 조정하는 것이 보통이다.

### ✍ Visible

컨트롤이 화면에 보이도록 한다. 이 속성이 False일 경우 컨트롤은 화면에 보이지 않게 된다. 특정 컨트롤을 잠시 숨기고자 하는 목적으로 사용된다. 폼을 제외한 모든 컨트롤의 디폴트 속성은 True이며 만들어진 컨트롤은 일부러 숨기지 않는 한 모두 화면에 보여진다. 숨겨진 컨트롤을 보이게 하려면 Visible 속성을 True로 만들어 주어야 한다.

### ✍ Text

에디트 컨트롤이나 메모 컨트롤에 입력된 문장을 나타내며 실행중에 이 속성을 읽음으로써 사용자가 입력한 문자열을 얻을 수 있다. 또한 이 속성에 문자열을 대입하여 실행중에 에디트 박스의 내용을 바꿀 수도 있다. 에디트 컨트롤, 메모 컨트롤은 Caption 속성이 없으며 대신 Text 속성으로 내용을 표현한다.

### Enabled

컴포넌트의 기능을 잠시 정지시킬 때 사용한다. 예를 들어 버튼 컴포넌트는  와 같은 모양을 가지며 마우스 버튼으로 누르면 쑥 들어가는 것이 본연의 기능이다 그러나 이 속성을 False로 맞추어 놓으면 버튼이  와 같이 희미한 회색이 되며 눌러지지 않게 된다. 메뉴나 에디트 박스 등 대부분의 컴포넌트에도 이 속성이 있으며 컴포넌트를 사용할 수 없다는 것을 사용자에게 알려줄 때 사용한다.

### Tag

특별한 용도가 정해져 있지 않은 속성이며 정수값 하나를 기억할 수 있다. 델파이가 이 속성을 사용하지 않으므로 사용자가 마음대로 의미를 부여하여 사용할 수 있도록 되어 있다. 이 속성으로 컴포넌트에 일련번호를 붙인다거나 컴포넌트의 의미를 나름대로 정의한다.

참고하세요

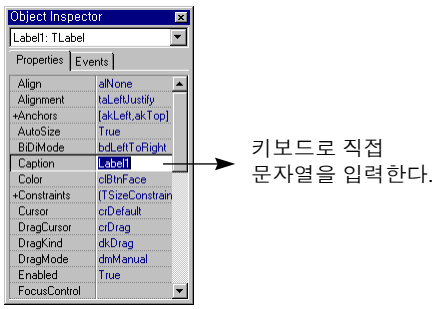
책을 읽다 보면 컴포넌트, 컨트롤 등의 용어를 별 구분없이 섞어서 사용하고 있다는 것을 느낄 것이다. 두 용어는 엄밀히 따진다면 조금 다른 의미를 가지고 있지만 일단은 같은 뜻을 가지고 있다고 생각하기 바란다. 버튼 컴포넌트라는 말과 버튼 컨트롤이라는 말은 거의 같은 뜻이다. 다음에 정확한 의미를 논하겠지만 컴포넌트라는 용어가 컨트롤보다는 조금 더 포괄하는 범위가 넓다.

## 라. 속성 지정

오브젝트 인스펙터에는 선택된 컴포넌트의 속성들이 나열된다. 속성을 변경하기 위해서는 해당 속성을 마우스로 클릭한 후 새로운 값을 입력하는 것이 기본이지만 이 외에도 속성을 변경시키는 방법은 다양하다. 어떤 속성을 어떻게 변경하는지 레이블 컴포넌트를 예로 들어 유형별로 알아보자.

### ■ 문자열 입력형

속성값이 문자열인 경우이며 대표적으로 Caption 속성을 들 수 있다. 오브젝트 인스펙터에서 Caption을 클릭하면 속성값 란에 캐럿이 나타난다.




여기에 새로운 속성값을 입력하되 입력 중 잘못 입력한 사항은 BS, Del, 커서 이동키로 편집할 수 있다. Caption 속성을 변경하면 폼에 있는 레이블의 캡션이 곧바로 변경된다.


오브젝트 인스펙터에서 속성을 선택할 때는 마우스를 사용하는 것이 보통이지만 키보드의 Tab 키와 알파벳 키를 순서대로 눌러 원하는 속성을 빨리 찾을 수도 있다. 예를 들어 Tab 키를 누르고 W를 누르면 W로 시작되는 첫 번째 속성이 곧바로 선택된다. 키보드로도 오브젝트 인스펙터를 사용할 수 있도록 해놓은 세심한 배려가 돋보이기는 하지만 속성이 알파벳 순으로 배열되어 있기 때문에 막상 실무에 사용될 경우는 무척 드물다.

## ■ 숫자 입력형

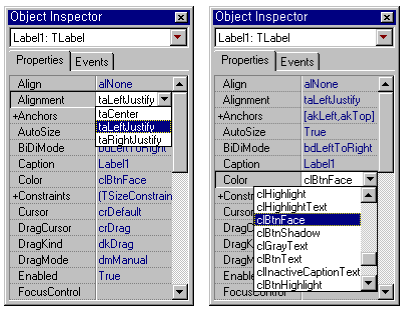
Left, Top, Height, Width 등의 속성이 이에 해당하며 문자열을 입력하듯이 숫자를 입력하면 된다. 그러나 이런 속성은 오브젝트 인스펙터를 통해 조정하는 것보다 폼상에서 마우스로 직접 조정하는 것이 더 쉽다. 폼상에서 컴포넌트의 위치나 크기를 조작하면 이 속성값들이 자동으로 변경된다. 정확한 위치나 크기를 지정해야 할 경우는 오브젝트 인스펙터에서 직접 숫자로 입력하는 것도 물론 가능하다.

## ■ 선택형

몇 가지 선택 가능한 값을 가지며 그 중의 한 값을 선택하는 속성이다. 세 가지 가능한 값을 가지는 Alignment 속성이 이에 해당한다. 이런 속성은 속성값을 더블클릭하면 다음값으로 바꾸어 주므로 원하는 값이 나올 때까지 더블클릭만 해주면 된다. 또는 속성값란의 ▼를 눌러 콤보 박스를 연 다음 목록에서 원하

는 값을 선택한다.

그림
선택형 속성

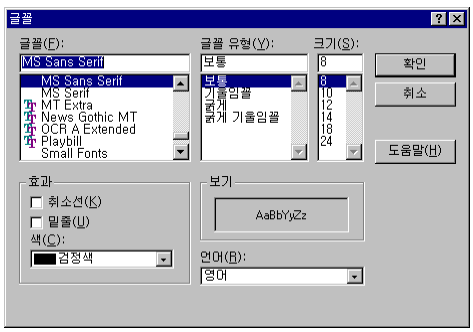



Color, Align, AutoSize 등 대부분의 속성이 이런 선택형이며 속성값으로 True, False의 진위형 값을 가지는 속성들도 모두 선택형이다.

## ■ 대화상자형

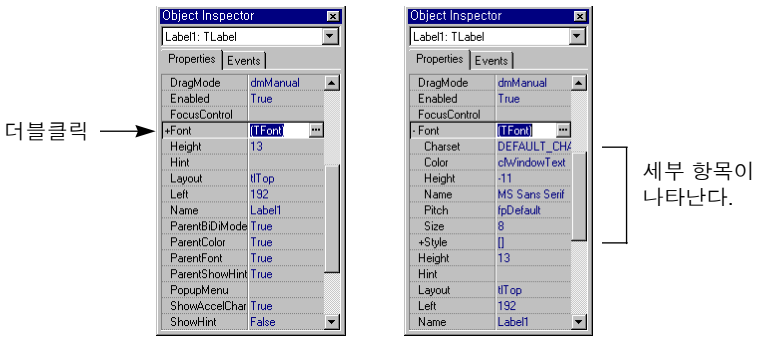
가장 복잡한 유형이며 이것 저것 시시콜콜이 많은 것을 지정해 주어야 하는 경우이다. 글꼴 이름, 크기, 색상, 모양 등을 지정해 주어야 하는 Font 속성이 그 대표적인 경우에 해당된다. 이런 속성은 오브젝트 인스펙터에서 일일이 값을 설정하지 않고 대화상자를 사용한다. Font 속성값을 더블클릭하거나 아니면 속성값 란의 ...를 클릭하면 폰트 속성을 다양하게 설정할 수 있는 대화상자가 열린다.

그림
Font 속성을 설정하는 대화상자




이 대화상자에서 Font 속성을 지정한 후 확인 버튼을 누르면 대화상자가 닫히면서 속성이 변경된다. 오브젝트 인스펙터에서 곧바로 속성을 설정하고자 할 경우는 좌측의 Font 속성 이름을 더블클릭한다. 그러면 Font 속성 아래 세부적인 속성이 나타난다.

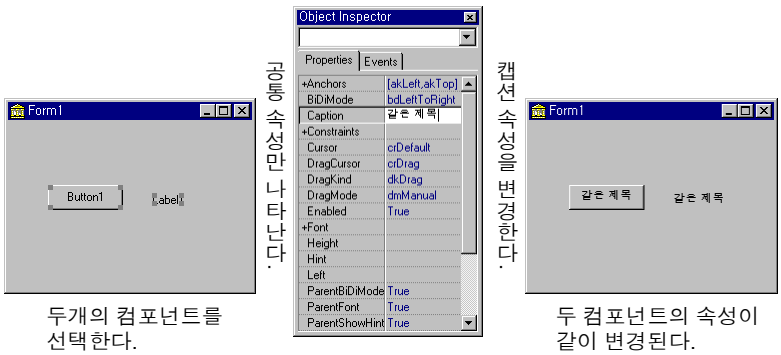
**그림**
세부 속성 열기




Font 속성 옆의 +기호는 이 속성이 확장 가능하다는, 즉 세부 속성이 있다는 뜻이다. Font 속성의 세부 속성중 Style 속성은 또 다른 세부 항목을 가지고 있으며 역시 속성 이름을 더블클릭하면 세부 속성이 열린다. 세부 속성을 닫으려면 다시 한번 더블클릭해 준다.

개별 컴포넌트의 속성을 변경하는 방법은 상술한 바와 같고 복수 개의 컴포넌트가 공통으로 가지는 속성을 한꺼번에 변경하는 것도 가능하다. 폼에서 두 개 이상의 컴포넌트를 선택할 경우 오브젝트 인스펙터에는 선택한 모든 컴포넌트에 공통적으로 존재하는 속성만 나타난다. 단, 예외적으로 Name속성은 모든 컴포넌트에 있지만 컴포넌트별로 유일해야 하므로 나타나지 않는다. 이 상태에서 원하는 속성을 입력하면 두 컴포넌트의 속성이 한꺼번에 변경된다. 다음은 레이블과 버튼의 Caption 속성을 한꺼번에 변경하는 방법을 보인 것이다.

**그림**
공통 속성 변경

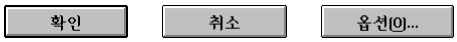



디자인시에는 오브젝트 인스펙터를 사용하여 컴포넌트의 속성을 변경하며 실행중에 컴포넌트의 속성을 변경할 때는 코드를 사용한다. 주로 대입문을 사용하여 “속성:=값”의 형태를 띤다. 2 장에서 Fruit 레이블의 Caption 속성을 변경할 때 := 대입 연산자를 이미 사용해 보았을 것이다.

# 마. 버튼 컴포넌트

버튼 컴포넌트는 마우스를 주 입력 장치로 하는 윈도우즈에서 가장 보편적으로 사용되는 명령 장치이다.


주로 명령을 내리거나 컴퓨터의 질문에 사용자가 응답하는 용도로 사용되며 다음과 같은 모양을 가지고 있다.




델파이가 제공하는 버튼 컴포넌트는 이런 버튼을 폼 상에 위치시킨다. 버튼이 가지는 속성 중 가장 겉으로 잘 드러나는 속성이 버튼 가운데 있는 "확인", "취소" 등의 버튼 제목이며 이 제목은 Caption 속성을 수정함으로써 지정한다. 버튼의 위치는 디자인시에 버튼을 드래그하여 지정하며 크기는 핸들을 드래그하여 조정한다. 가장 흔하게 볼 수 있는 컴포넌트이며 윈도우즈를 조금만 써 본 사람이라면 버튼에 대해서는 잘 알기 때문에 이 책에서 필자가 예제 작성을 위해 아주 열심히 애용하고 있다.

# 3-3 이벤트

## 가. 이벤트와 코드

델파이에서는 마우스를 딸깍거려 컴포넌트를 배치하고 속성을 설정하므로 프로그램을 디자인하는 일이 무척이나 쉬운 편이다. 그러나 컴포넌트를 폼에 배치하는 것은 어디까지나 프로그램의 겉모양을 정의할 뿐이지 프로그램의 동작이나 기능을 정의하는 것은 아니다. 프로그램의 동작을 정의하기 위해서는 이벤트와 코드 작성법에 대해 알아야 한다. 이벤트(event)를 우리말로 간단히 번역하면 "사건"이며 좀 길게 설명하면 다음과 같다.

**이벤트 : 마우스를 클릭하는 사용자의 반응이나 시간이 경과하는 시스템의 변화 등과 같이 인식 가능한 일체의 변화를 의미한다.**

다음과 같은 것들이 모두 하나의 이벤트이며 아무리 사소한 것이든 변경되면 이벤트가 발생한다.

- ❑ 사용자가 버튼을 눌렀다.
- ❑ 시간이 1 초 경과하였다.
- ❑ 폼이 새로 생성되었다.
- ❑ 사용자가 키보드를 눌렀다.

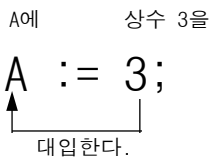
델파이 프로그래밍이란 이런 각각의 이벤트에 대해 어떤 동작을 할 것인가를 코드로 작성하는 과정의 연속이다.  변수에 값을 대입하는 코드, 컴포넌트의 속성을 변화시키는 코드, 화면에 무엇인가를 그리는 코드, 소리를 발생시키는 코드를 작성함으로써 프로그램을 짜 나간다. 이벤트에 대해 어떻게 동작하는가를 지정하는 코드를 이벤트 핸들러(event handler)라고 한다. 2장에서 만든 Fruit 예제의 경우 BtnApple 버튼을 누를 때의 코드를 살펴보자.

```
procedure TForm1.BtnAppleClick(Sender: TObject);
begin
  Fruit.Caption:='Apple';
end;
```

이 코드는 BtnApple 버튼의 OnClick 이벤트를 처리하며 Fruit 레이블의 Caption 속성을 'Apple'로 바꾸는 동작을 한다. 즉 프로그램 실행중에 BtnApple 버튼을 사용자가 마우스로 누를 경우 레이블의 Caption 속성을 바꾸도록 지시하는 이벤트 핸들러이다.

문법에 관해서는 다음에 체계적으로 배우겠지만 일단 가장 기본적인 세 가지는 알고 가도록 하자. 우선 파스칼에서는 대입문에 ':=' 기호를 사용하며 우변의 값을 좌변으로 대입하는 동작을 한다. 예를 들어 변수 A에 값 3을 대입하는 문장은 다음과 같다.

**그림**
변수에 값을 대입하는 대입문




그리고 모든 파스칼 문장은 세미콜론으로 끝난다는 점과 문자열은 홑따옴표로 '요렇게' 싸준다는 점도 알아 두도록 하자. 이 정도만 알면 Fruit.Caption:='Apple'; 코드의 문법적 의미도 파악할 수 있을 것이다.

# 나. 여러 가지 이벤트

각각의 컴포넌트가 모양과 기능이 다르기 때문에 이벤트의 종류도 컴포넌트별로 각각 다르다. 레이블이나 폼의 경우 더블클릭을 할 수 있지만 버튼의 경우는 더블클릭이 없기 때문에 OnDblClick 이벤트가 없다. 속성과 마찬가지로 이벤트도 컴포넌트별로 개별적으로 공부해야 한다. 여기서는 가장 일반적으로 많이 사용되는 이벤트 몇 가지에 관해 우선 알아보기로 한다. 이벤트의 이름은 전치사 On으로 시작하며 이벤트의 의미를 나타낼 수 있는 짧은 이름을 사용한다.

Click의 정확한 의미는 마우스 버튼을 눌렀다 떼는 것이다 단순히 누르는 것과는 구분된다.

☐ OnClick : 컴포넌트 위에서 마우스 버튼을 눌렀다 뗄 경우, 즉 클릭할 경우 발생하는 이벤트이다. 버튼을 누르거나 폼의 빈 공간을 누르는 등의 동작이 OnClick 이벤트를 발생시킨다. 윈도우즈에서는 마우스 클릭을 많이 하므로

이 이벤트가 가장 많이 발생하며 가장 흔한 이벤트이다.

- OnDblClick : 컴포넌트 위에서 마우스를 더블클릭할 때 발생하는 이벤트이다. 폼, 레이블, 에디트 박스 등의 컴포넌트가 이 이벤트를 가진다.
- OnCreate : 폼이 처음 생성될 때 발생하는 이벤트이며 폼이 만들어지기 전에 실행되므로 각종 초기화에 관련된 코드가 기입된다.
- OnShow : 폼이 숨겨져 있다가 나타날 때 발생하는 이벤트이다. 이 이벤트는 폼에만 있으며 다른 컴포넌트에는 없다.
- OnMouseDown : 마우스를 누를 경우 발생하는 이벤트이다. OnClick 이벤트는 마우스 버튼을 눌렀다 떼어야 발생하지만 이 이벤트는 마우스 버튼을 누르기만 해도 발생한다.
- OnKeyDown : 키보드를 누를 때 발생하는 이벤트이다.
- OnChange : 컴포넌트 안의 내용이 바뀔 때 발생하는 이벤트이다. 에디트 박스의 문자열이 변경되었거나 리스트 박스에서 다른 항목을 선택했을 때 발생한다.

이 이벤트들은 가장 자주 발생하며 가장 일반적인 이벤트이므로 대충 외워 두는 것이 좋다. 하긴 실습을 하다보면 자연스럽게 외워지겠지만 말이다. 언제 이 벤트가 발생하는가와 이름이 무엇인가를 기억해 두도록 하고 자세한 사용 방법은 실습을 통해 익히도록 한다.

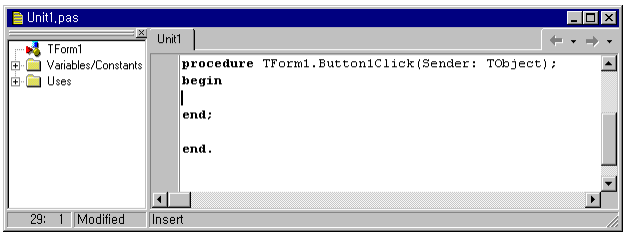
# 다. 코드 작성법

컴포넌트에서 발생 가능한 이벤트에 대한 코드, 즉 이벤트 핸들러를 작성하는 방법에는 두 가지가 있다.

첫째, 컴포넌트를 폼에 배치한 후 컴포넌트를 더블클릭한다. 그러면 그 컴포넌트에서 가장 자주 발생하는 이벤트에 대한 코드를 작성할 수 있도록 코드 에디터가 열린다. Button1 컴포넌트를 폼에 배치한 후 더블클릭했다면 코드 에디터에 다음과 같은 코드가 나타날 것이다.

**그림**

코드 에디터에 나타난 이벤트 핸들러의 뼈대



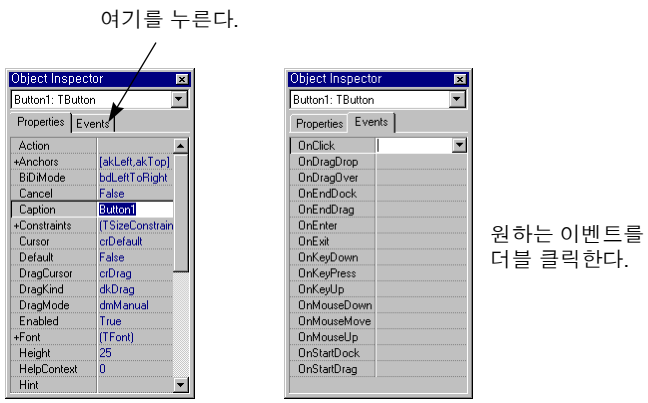

가장 자주 발생하는 이벤트를 디폴트 이벤트라고 한다.

버튼 컴포넌트에서 가장 자주 발생하는 이벤트는 OnClick 이벤트이므로 이 이벤트에 대한 코드를 작성할 수 있도록 해준다. 이 코드는 이벤트 핸들러에 대한 뼈대(Frame)에 해당하며 이벤트 핸들러를 작성할 때 델파이가 자동으로 만들어 준다. 사용자는 이벤트에 대한 처리 사항을 begin과 end; 사이에 기입해 주면 된다.

둘째, 컴포넌트 선택 후 오브젝트 인스펙터에서 이벤트를 선택하여 더블클릭한다. 오브젝트 인스펙터는 평소에 속성값을 보여주지만 Events 페이지 탭을 클릭하면 다음과 같이 이벤트 리스트를 보여준다.

**그림**

Events 페이지




이 상태에서 원하는 이벤트를 더블클릭하면 선택한 이벤트의 핸들러 뼈대를 만들어 코드 에디터에 열어준다. 단 좌측의 이벤트 이름을 클릭하는 것이 아니라 우측의 이벤트값을 입력하는 란을 더블클릭해야 함을 주의하자. 좌측의 이벤트 이름은 더블클릭해도 아무 반응이 없다. 오브젝트 인스펙터를 통해 이벤트 핸들러를 작성할 경우 컴포넌트에서 발생 가능한 모든 이벤트에 대한 핸들러를 만들 수 있다.

반면 폼에서 컴포넌트를 더블클릭하는 방법은 가장 자주 발생하는 이벤트 하나에 대해서만 코드를 만들 수 있다. 즉 이벤트 핸들러를 만드는 정식적인 방법은 오브젝트 인스펙터를 통하는 방법이되 이 방법이 번거로우므로 가장 자주 발

생하는 디폴트 이벤트에 대해서만 폼에서의 더블클릭 방법을 제공하는 것이다. 여기서는 코드를 어떻게 입력하는가만 단계적으로 살펴보았다. 구체적인 코드 작성 문법은 앞으로 하나씩 배워 나가게 될 것이다.


① 이벤트:인식 가능한 일체의 변화, 사건
② 변수나 속성에 값을 대입할 때는 := 대입문을 사용한다.
③ 폼의 컴포넌트를 더블클릭하여 이벤트 핸들러를 작성한다.

# 3-4 메소드

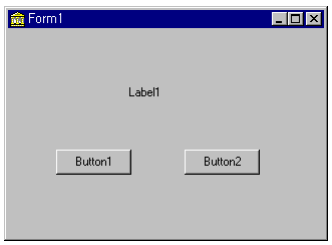
메소드란 특정한 컴포넌트에 연관되어 있는 코드를 말한다. 좀 더 정확하고 유식하게 정의를 내려보면 "특정 오브젝트에 연관된 프로시저나 함수(Procedure or function associated with a particular object)"가 되는데 이 말은 처음 델파이를 배우기 시작한 사람들에게는 너무 난해한 말이다.

속성은 컴포넌트의 모양을 결정한다. 반면 메소드는 컴포넌트의 동작을 유발시킨다. 컴포넌트에 어떤 변화를 주는 것이기 때문에 디자인시에는 메소드를 사용할 수 없으며 코드로 작성하여 실행중에만 사용할 수 있다. 속성과 마찬가지로 메소드도 컴포넌트에 따라 다양하게 존재한다. 레이블 컴포넌트의 경우 수십개의 메소드를 가지는 데 가장 이해하기 쉬운 세 가지 메소드만 예를 들어 보자.

Hide : Visible 속성을 False 로 만들어 레이블이 보이지 않도록 한다.
Show : 레이블의 Visible 속성을 True 로 설정하여 폼에 보이도록 한다.
Repaint : 레이블에 변화가 생겼을 경우 즉각 화면에 다시 그리도록 한다.

3jang
method

메소드는 속성을 지정하는 것과 같은 방법으로 호출한다. "컴포넌트.메소드"의 형식을 가지며 인수가 있을 경우는 메소드 이름 뒤에 인수도 같이 적어 준다. Label1을 화면에서 안보이게 만들려면 Label1.Hide;를 호출하고 보이게 하려면 Label1.Show; 를 호출한다. 실습을 위해 폼에 레이블 하나와 버튼 두 개를 배치하자.




컴포넌트를 배치한 후 Button1을 더블클릭하여 코드 에디터를 연 후 다음 코드를 입력해 보자. Button1의 OnClick 이벤트 핸들러를 작성하는 것이다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Hide;
```

```
end;
```

같은 방법으로 Button2 를 더블클릭하여 다음 코드를 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Label1.Show;
end;
```

F9를 눌러 컴파일시키고 프로그램을 실행시킨 후 Button1을 누르면 레이블이 숨겨지고 Button2를 누르면 레이블이 보일 것이다. Button1을 누르면 OnClick 이벤트가 발생하며 이 이벤트 핸들러에서 Label1의 Hide 메소드를 호출하여 레이블을 숨기며 마찬가지로 Button2의 OnClick 이벤트 핸들러에서 Label1.Show를 호출하여 레이블이 보이도록 만든다.

Show, Hide 메소드는 컴포넌트의 Visible 속성을 변경하여 컴포넌트를 숨기기도 하고 보여주기도 하므로 메소드를 호출하지 않고 직접 Visible 속성을 변경해도 동일한 결과를 얻을 수 있다. 그러나 메소드가 아니면 도저히 해결하지 못하는 경우도 있으며 구체적인 예는 천천히 살펴보기로 하자.

# 3-5 배경색 바꾸기 예제

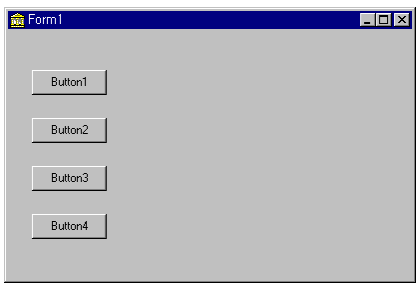
여기까지 여러분들이 배워 온 것들은 델파이의 화면 구성, 컴포넌트, 속성, 이벤트, 코드 작성법 등등 델파이의 가장 기본적인 부분들이었다. 이제 앞에서 터득한 기초를 토대로 예제 하나를 직접 작성해 보도록 하자.

## 가. Fcolor

3jang
fcolor

여기서 우리가 만들어 볼 예제는 폼 상에 네 개의 버튼을 배치해 두고 버튼의 이름을 Red, Blue, Green, Gray 등의 색상 이름으로 준 후 각 버튼이 눌러질 때 폼의 색상을 바꾸어 주는 예제이다. 먼저 직접 작성해 보도록 하되 모르겠으면 아래의 과정을 따라 같이 실습을 해보자.

1. 델파이를 실행한다. 너무 너무 당연한 과정이다.
2. 새로운 프로젝트를 시작한다. File/New Application 을 선택하면 델파이는 깨끗한 폼을 하나 준비해 줄 것이다. 이 폼에 우리의 프로그램을 만들어 나간다.
3. 폼 위에 네 개의 버튼을 다음과 같이 배치한다. 폼의 크기는 버튼 네 개가 들어갈 만한 크기로 조정하도록 한다.




이렇게 버튼을 배치하면 버튼 네 개와 폼 하나, 모두 다섯 개의 컴포넌트가 생성된 것이다.

4. 다음과 같이 속성을 설정한다. 속성 설정은 오브젝트 인스펙터를 사용하되 먼저 폼에서 속성을 변경할 컴포넌트를 선택해야 함을 잊지 말도록 하자.

폼의 경우는 폼의 빈 영역을 클릭하면 선택되는 것이다.

| 컴포넌트 | Name | Caption |
|---|---|---|
| Button1 | Red | Red |
| Button2 | Blue | Blue |
| Button3 | Green | Green |
| Button4 | Gray | Gray |
| Form1 | ColorForm | ColorForm |

Name 속성만 변경하면 Caption 속성도 같이 변경되므로 Name만 입력하면 된다. 간단한 예제이므로 별도의 속성을 지정할 것도 없으며 Name 속성과 Caption 속성도 모두 일치시켰다. 여기까지가 디자인 과정이며 프로그램의 겉모양을 다 만든 것이다. 코드를 작성하여 프로그램의 동작을 지정해 주면 프로그램이 완성된다.

5 코드를 작성한다. Red 버튼을 더블클릭하여 코드 에디터를 연다. 버튼의 경우 가장 자주 발생하는 이벤트가 OnClick 이벤트이므로 버튼을 더블클릭하면 OnClick 이벤트에 대한 이벤트 핸들러를 만들 수 있도록 다음과 같이 뼈대를 만들어 줄 것이다.

```
procedure TForm1.RedClick(Sender: TObject);
begin

end;
```

begin과 end 사이에 ColorForm.Color:=clRed;를 작성한다. 이 코드의 의미는 Red 버튼이 사용자에 의해 눌러질 경우(OnClick 이벤트가 발생할 경우) ColorForm의 Color 속성을 clRed로 바꾼다는 의미이다. 즉 폼의 색상을 빨간색으로 변경한다. 같은 방법으로 나머지 버튼의 OnClick 이벤트에 각각 다음 코드를 작성한다.

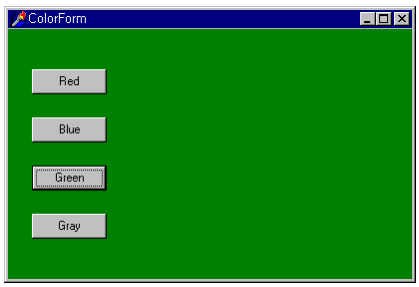
```
procedure TColorForm.RedClick(Sender: TObject);
begin
 ColorForm.Color:=clRed;
end;
```

```
procedure TColorForm.BlueClick(Sender: TObject);
begin
  ColorForm.Color:=clBlue;
end;

procedure TColorForm.GreenClick(Sender: TObject);
begin
  ColorForm.Color:=clGreen;
end;

procedure TColorForm.GrayClick(Sender: TObject);
begin
  ColorForm.Color:=clGray;
end;
```

모두 폼의 색상을 변경하는 코드이다.

6 프로젝트를 저장한다. 소스 파일 이름은 Fcolor_f.pas 로 지정하고 프로젝트 이름은 Fcolor.dpr 로 지정한다. 저장하는 위치는 가급적이면 별도의 디렉토리를 만들어 저장하는 것이 관리하기에 좋다. 프로젝트 이름과 같은 FCOLOR 라는 서브 디렉토리를 만들고 이 디렉토리에 저장하도록 하자.

7 프로그램을 실행한다. F9 키를 누르거나 스피드 버튼의  를 누른다. 에러가 발생하면 십중 팔구는 오타일 가능성이 가장 높으므로 오타를 점검해 보도록 하자. 실행중의 모습은 다음과 같다.

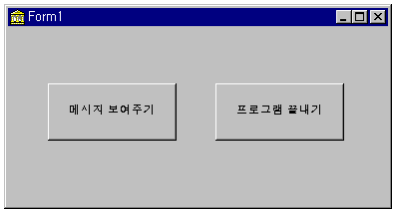



각 버튼을 누르면 폼의 색상이 바뀔 것이다. 각 버튼의 OnClick 이벤트가 발생하면 이벤트 핸들러에 정의된 코드가 실행되어 폼의 Color 속성을 변경하기 때문이다. 예제를 종료하면 다시 델파이 개발 환경으로 돌아온다.

# 나. Button 예제

3jang
Button

실습을 겸해서 비슷한 예제를 하나 더 작성해 보자. 대충의 과정만 보이므로 이번에는 책을 읽고 난 후 직접 만들어 보기 바란다. 다음 예제는 버튼을 사용하여 두 가지 명령을 내린다. 폼에 버튼 두 개를 배치하고 버튼의 Caption 속성에 버튼의 의미를 나타내는 문자열을 다음과 같이 설정하였다.




그리고 버튼을 더블클릭하여 OnClick 이벤트 핸들러를 각각 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('간단한 전달 내용을 보여주는 메시지 상자입니다.');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

ShowMessage는 다음과 같은 메시지 상자를 화면 중앙에 보여주며 Close는 프로그램을 끝내라는 명령이다. 즉 왼쪽 버튼을 누르면 메시지를 보여주며 오른쪽 버튼을 누르면 프로그램을 종료한다.

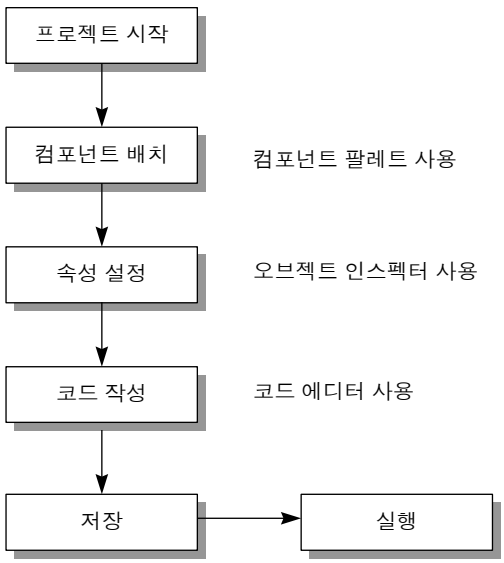

# 다. 델파이 프로그래밍 절차

델파이 프로그래밍은 대체로 다음과 같은 단계를 밟아 만들어진다. 앞에서 실습을 해 보았기 때문에 대충은 알겠지만 머리속에 깔끔하게 정리해 두도록 하

자.

**그림**

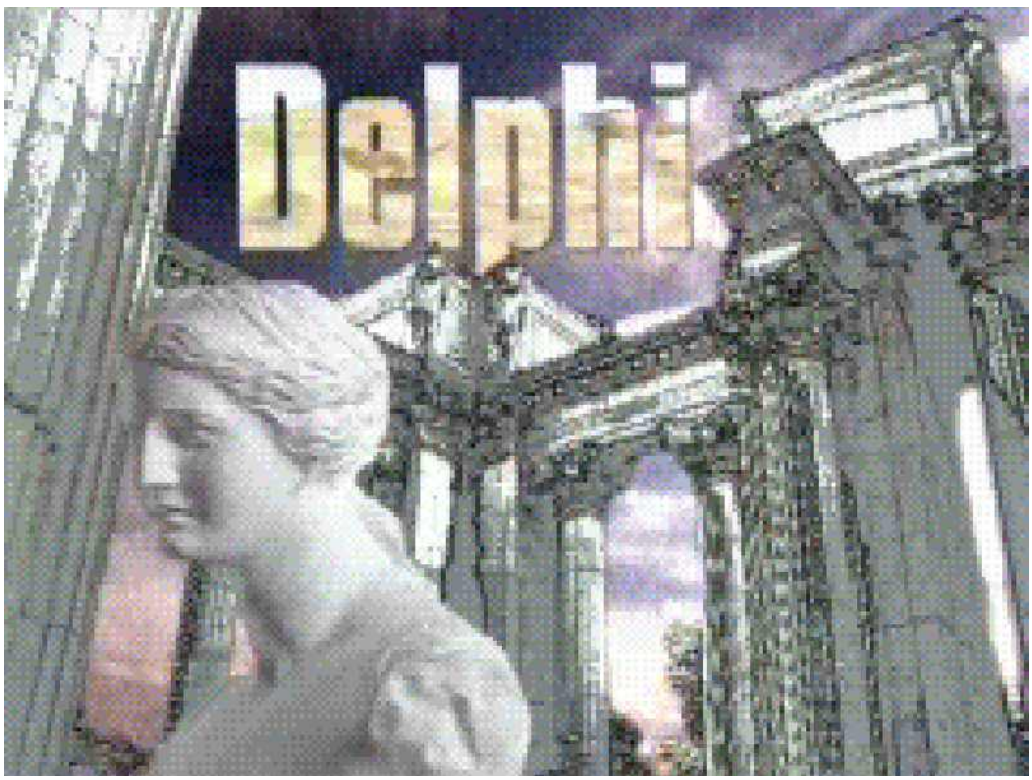
프로그램을 개발하는 전체적인 과정




간단한 프로그램일 경우는 위의 단계를 밟아 가면 되지만 좀 더 규모가 큰 프로그램이라면 프로그램의 구상, 설계 과정이 먼저 이루어져야 하며 만들어지고 난 후에도 테스트, 디버깅, 유지, 보수 등의 과정이 추가된다. 이후부터 이 책에서는 예제를 보일 때 상세한 과정은 보이지 않고 배치할 컴포넌트의 종류와 속성, 코드 리스트만 밝히기로 한다.

# 제 4 장

# 폼



# 4-1 폼

## 가. 폼=윈도우

새로운 프로젝트를 시작하면 제일 먼저 아무 것도 놓여져 있지 않은 빈 폼이 하나 주어진다. 이 폼에 사용자가 원하는 컴포넌트를 배치하면서 프로그램이 점점 완성되어 가게 된다. 델파이가 제공하는 모든 컴포넌트들은 예외없이 폼상에 놓여지게 되므로 폼은 델파이 프로그래밍의 가장 중심이 되는 컴포넌트이다.

**그림**
델파이가 만들어 주는 빈 폼


폼이란 하나의 윈도우를 말한다. 즉 화면 상의 일정한 사각 영역을 차지하고 있고 경계선과 타이틀 바를 가지며 자리를 옮길 수 있는 독립된 개체로서의 윈도우이다. 폼과 윈도우는 같은 대상을 가리키지만 폼은 디자인 과정의 윈도우를 이르는 말이며 윈도우란 폼이 실행되고 있을 때를 이르는 말이다. 폼을 디자인하는 것은 지금 만드는 프로그램이 실행될 때 어떤 모양을 가지는가를 디자인하는 것이다. 폼도 버튼이나 레이블과 마찬가지로 컴포넌트의 일종이되, 다음과 같은 점이 다른 컴포넌트와 다르다.

① 폼은 프로젝트를 시작할 때 델파이가 직접 만들어 주므로 컴포넌트 팔레트에는 없다. 추가로 폼을 더 만들고 싶으면 File/New Form 메뉴를 통해서 만들어야 한다.

② 폼은 다른 컴포넌트를 포함하는 특별난 컴포넌트이다. 모든 컴포넌트가 폼 위에 생성되며 폼이 이동하거나 삭제되면 폼 안에 놓여진 모든 컴포넌트도 이동되거나 삭제된다. 이렇게 다른 컴포넌트를 담는 컴포넌트를 컨테이너 컴포넌트(Container Component)라고 한다.

# 나. 폼의 속성

윈도우즈 환경에서 실행되는 많은 프로그램들을 보면 윈도우의 모양이 가지각색이다. 탐색기나 메모장 같은 표준적인 윈도우도 있고 경계선이 없는 윈도우, 타이틀 바가 없는 윈도우, 메뉴가 있는 윈도우, 배경 색상이 가지각색인 윈도우 등등이 있다.

델파이가 만들어 내는 윈도우의 모양은 폼의 속성을 어떻게 설정하는가에 따라 달라진다. 폼의 모양을 결정하는 몇 가지 중요한 속성에 관해 알아보도록 하자.

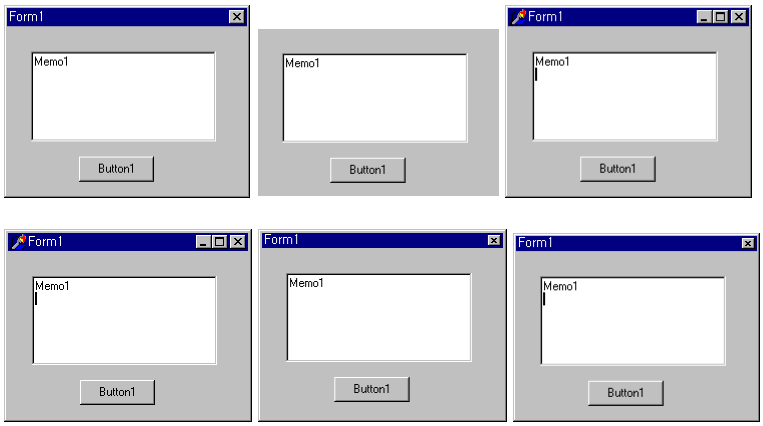
### ✍ BorderStyle

폼의 경계선 모양을 지정한다. 이 속성을 어떻게 지정하는가에 따라 폼의 모양은 물론 폼의 기능에도 변화가 생긴다. 경계선은 단순한 장식이 아니며 경계선의 모양에 따라 폼 전체의 모양과 기능이 달라지기 때문이다. 다음 여섯 가지 값이 가능하다.

| 속성값 | 의 미 |
|---|---|
| bsDialog | 크기 조정이 불가능하며 대화상자 형태를 가진다. |
| bsNone | 경계선을 가지지 않으며 최소, 최대, 조절 메뉴도 가지지 않는다. 크기 조정은 물론 불가능하며 타이틀 바도 없기 때문에 위치를 옮길 수도 없다. |
| bsSingle | 크기 조정이 불가능하며 선 하나로 된 경계선을 가진다. |
| bsSizeable | 크기 조정이 가능한 표준적인 경계선을 가진다. 이 속성이 디폴트이다. |
| bsToolWindow | bsSingle과 같되 타이틀 바의 높이가 좁으며 시스템 메뉴와 닫기 버튼이 없다. |
| bsSizeToolWindow | bsToolWindow와 같되 크기 조정이 가능하다. |

버튼과 메모 컴포넌트만 배치해 두고 경계선 속성을 각각 다르게 설정해 보았다. 모양은 다음과 같으며 실행중에 크기 조정이 가능한 형태는 네 번째 형태인 bsSizeable과 다섯 번째 형태인 bsSizeToolWindow 뿐이다.

그림
BorderStyle 속성에 따른 폼의 모양




폼의 BorderStyle 속성을 변경하더라도 디자인시에는 폼의 속성이 변하지 않으며 실행시켜야 비로소 속성이 변경된다. 왜냐하면 디자인시에는 폼의 크기 조정이나 위치 이동이 항상 가능해야 하기 때문이다. 이 속성을 bsNone으로 설정했다고 해서 경계선과 타이틀 바를 없애버리면 디자인중에 폼의 크기나 위치를 옮기기가 불편해질 것이다. 그래서 디자인시에 폼은 항상 bsSizeable이다.

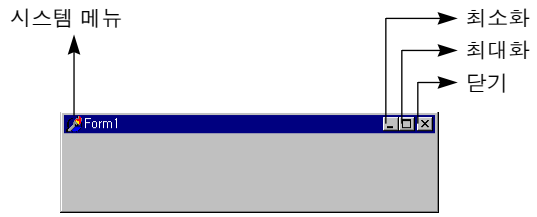
### BorderIcons

폼의 타이틀 바에 나타나는 버튼의 종류를 지정한다. 다음 네 가지 버튼을 개별적으로 선택하여 표시할 수 있다. 디폴트로 세 개의 버튼만 선택되어 있으며 biHelp는 선택되어 있지 않다. 중첩 속성이므로 오브젝트 인스펙터에서 속성 이름을 더블클릭한 후 세부 속성값을 변경하도록 한다.

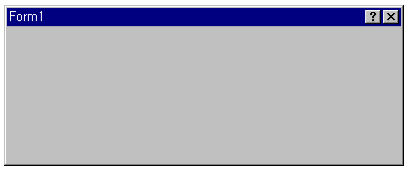
| 속성값 | 의 미 |
|---|---|
| biSystemMenu | 시스템 메뉴 |
| biMinimize | 최소화 버튼 |
| biMaximize | 최대화 버튼 |
| biHelp | 도움말 버튼 |

이 네 가지 아이콘 속성 중 어떤 것들은 BorderStyle 속성과 상호 배타적이다. 예를 들어 BorderStyle 이 bsDialog 일 때는 최소, 최대화 버튼을 쓸 수 없으며 도움말 버튼은 최대, 최소 버튼과 함께 쓸 수 없다. 상식적으로 생각해 보면 당연한 규칙이라고 할 수 있다. 다음은 bsSizeable 윈도우에 세 가지 아이콘을 선택한 것이다. 닫기 버튼은 시스템 메뉴가 있으면 같이 나타난다.

그림
타이틀 바에 나타나는 아이콘




다음은 bsDialog 윈도우에 biHelp 아이콘을 선택한 것이다. 물음표 모양의 아이콘이 나타나는데 이 버튼은 상황별 도움말에 사용된다.




### FormStyle

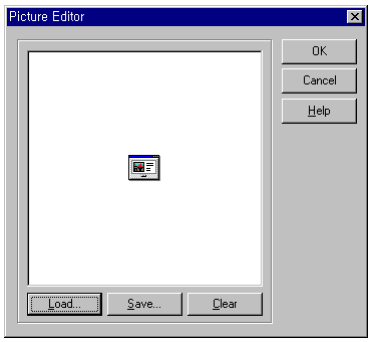
폼의 종류를 지정한다. 마이크로소프트의 워드나 프로그램 관리자처럼 하나의 윈도우 안에 여러 개의 윈도우가 열리는 MDI 프로그램을 만들 때 사용하는 속성이다. 실습은 이 책의 한참 뒤에서나 해 볼 작정이므로 이런 속성도 있다는 것만 알아두자.

| 속성값 | 의미 |
|---|---|
| fsNormal | 표준적인 폼이다. |
| fsMDIChild | MDI 차일드 윈도우이다. |
| fsMDIForm | MDI 페어런트 윈도우이다. |
| fsStayOnTop | 이 속성이 지정된 폼은 다른 폼보다 항상 위에 위치한다. |

### Icon

폼이 최소화되었을 때 나타날 아이콘을 지정한다. 만약 이 속성을 지정하지 않으면 프로그램의 디폴트 아이콘(  요렇게 생겼다)을 사용한다. 폼의 아이콘과 프로그램의 아이콘은 다르다. 폼의 아이콘은 윈도우가 최소화되었을 때 나타나는 아이콘이며 프로그램 아이콘은 프로그램 관리자에 등록될 때 나타나는 아이콘이다. 오브젝트 인스펙터에서 이 속성을 더블클릭하면 아이콘을 읽어올 수 있는 대화상자를 보여준다.

**그림**
그림 읽기 대화상자




Load 버튼을 눌러 *.ICO 파일 중 하나를 읽어오면 폼의 아이콘으로 지정된다. ICO 파일은 델파이에 포함된 이미지 에디터를 사용하여 직접 디자인하거나 공개된 아이콘을 구해 사용하면 된다.

### ✍ Position

폼이 처음 나타날 때의 위치와 크기를 지정하며 특별히 이 속성을 변경하지 않으면 디자인시에 지정한 크기와 위치를 사용한다.

**표**
폼의 Position 속성

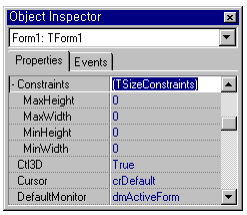
| 속성값 | 의 미 |
|---|---|
| poDesigned | 디자인할 때 지정한 폼의 크기와 위치를 그대로 사용한다. 이 값이 디폴트이다. |
| poDefault | 프로그램이 실행될 때 운영체제가 알아서 크기와 위치를 정해 준다. 화면의 해상도와는 상관없이 좌측 윗부분에 폼을 만들어 준다. |
| poDefaultPosOnly | 폼의 크기는 디자인시에 정한 크기를 사용하지만 위치는 델파이가 알아서 적당한 위치를 선택해 준다. |
| poDefaultSizeOnly | 폼의 위치는 디자인시에 정한 위치를 사용하지만 크기는 델파이가 알아서 적당하게 설정한다. |
| poScreenCenter | 폼의 크기는 디자인시에 정한 크기를 사용하며 위치는 화면의 정 중앙을 사용한다. 화면의 해상도에 상관없이 항상 폼이 정 중앙에 놓이게 된다. |
| pdDesktopCenter | poScreenCenter와 동일한 값이되 여러 개의 모니터를 쓸 때 약간 달라진다. |

### ✍ Constraints

이 속성은 폼의 최대, 최소 크기를 제한하며 오브젝트 인스펙터에서 이 속성

을 더블클릭해 확장해 보면 다음과 같이 4 개의 세부 속성이 나타난다.

그림
Constraints 의 세부 속성




각 속성은 폼의 최대 높이, 최대 폭, 최소 높이, 최소 폭을 지정하는데 디폴트로 이 값들은 모두 0 으로 지정되어 있어 크기 제한을 하지 않는다. 만약 이 속성에 최대, 최소값을 설정하면 폼의 크기는 이 범위내에서만 조정이 가능해진다. 예를 들어 MaxHeight 에 480 을 MaxWidth 에 640 을 설정한 후 폼의 크기를 조정해 보면 이 이상의 크기로 확대되지 않음을 알 수 있다.

### ✍ VertScrollBar, HorzScrollBar

폼에 수평, 수직 스크롤 바를 만들 것인가와 만들 경우 각 스크롤 바에 어떤 속성을 줄 것인가를 지정한다. 스크롤 바에 주어지는 속성이란 최대값, 최소값의 범위, 한 번 누를 때마다 증감될 양 등이다.

이 외에 폼의 배경 색상을 지정하는 Color 속성과 타이틀 바에 나타날 문자열을 지정하는 Caption 속성이 있다. 이런 일반적인 속성에 관해서는 3장에서 이미 논한 바 있으므로 생각이 잘 안나는 사람은 3장을 다시 읽어보기 바란다.

## 다. 폼의 이벤트

폼도 컴포넌트이므로 버튼이나 레이블과 같이 이벤트를 가진다. 사용자의 클릭이나 더블클릭 등의 동작에 의한 변화뿐만 아니라 프로그램 실행 과정에서도 여러 가지 이벤트가 발생한다. OnClick, OnDblClick, OnKeyPress 등 다른 컴포넌트에 있는 가장 일반적인 이벤트 외에도 폼에만 있는 이벤트들도 많다.

### ❑ OnCreate

폼이 가지는 이벤트 중에 가장 중요하며 가장 자주 사용하는 이벤트가 OnCreate 이벤트이다. 이 이벤트는 프로그램이 실행되고 폼이 만들어질 때 제일 먼저 발생한다. 디자인시에 속성을 설정하기가 어려운 경우나 폼에서 사용할

데이터를 초기화하는 등의 동작을 이 이벤트 처리 과정에서 수행한다. 이 이벤트가 폼의 디폴트 이벤트이다. 즉 폼의 빈 영역을 더블클릭하면 코드 에디터에 이 이벤트의 핸들러가 작성된다.

### ❑ OnDestroy

폼이 파괴될 때, 즉 다 사용하고 메모리에서 지워질 때 발생하는 이벤트이다. OnCreate 이벤트에서 초기화를 담당하는 것과는 반대로 OnDestroy 이벤트에서는 마지막 뒷정리를 담당한다. 할당해 놓고 해제하지 않은 메모리, 열어 놓고 닫지 않은 파일들을 이 이벤트에서 처리해 준다. 물론 정리할 내용이 없는 경우에는 이 이벤트를 사용하지 않아도 무방하다.

### ❑ OnShow

폼이 처음 화면에 나타날 때, 또는 가려졌다가 다시 나타날 때 발생하는 이벤트이다. OnCreate는 메모리에서 폼이 만들어질 때 발생하며 OnShow는 화면에 실제로 보이게 될 때 발생한다.

### ❑ OnHide

OnShow와 반대되는 이벤트이며 폼이 숨겨질 때 발생한다.

### ❑ OnClose

폼이 닫혀질 때 발생하는 이벤트이다.

### ❑ OnResize

사용자가 폼의 경계선을 드래그하여 폼의 크기를 변경시킬 때 이 이벤트가 발생한다. 폼의 크기가 너무 작아지거나 너무 커지지 않도록 제한하고 싶다거나 폼의 크기에 맞게 그림을 확대 또는 축소하고자 할 때 이 이벤트를 사용한다.


① 폼:델파이 프로그램의 중심이며 다른 컴포넌트를 담을 수 있는 컨테이너 컴포넌트이다.
② BorderStyle 속성에 따라 경계선의 모양과 기능이 정의된다.
③ OnCreate 이벤트:프로그램 실행과 동시에 발생하는 이벤트이다.

# 4-2 프로젝트의 구성

## 가. 구성 파일

프로그램 하나를 만들려면 여러 가지 구성 파일이 필요하다. 폼의 모양을 기억하는 파일, 코드를 담는 파일, 비트맵이나 커서를 담는 리소스 파일 등이 모여서 하나로 결합되어야 비로소 실행 가능한 프로그램이 만들어진다. 프로젝트(Project)란 하나의 프로그램을 만드는 데 필요한 이런 모든 구성 파일에 관한 정보의 총체를 말한다. 하나의 프로젝트를 만드는 것이 곧 하나의 프로그램을 만드는 것이며 프로그램을 만들기 위해서는 반드시 프로젝트를 구성해야 한다. 델파이의 프로젝트를 구성하는 파일들은 다음과 같은 것들이 있다. 모두 고유의 확장자를 가지므로 확장자별로 알아보자.

### .DPR

DPR 은 Delphi PRoject 의 준말이다.

프로젝트 그 자체이며 프로젝트마다 하나의 DPR 파일이 생성된다. 이 프로젝트에 어떤 폼들이 사용되며 어떤 유닛들이 사용되는가에 관한 정보들이 포함되어 있으며 프로그램을 초기화시키는 코드를 가지고 있다. 텍스트 파일 형태를 가지며 에디터를 사용하여 내용을 보거나 수정할 수도 있다. 하지만 이 파일의 내용은 델파이가 직접 관리하므로 사용자가 건드릴 필요가 없으며 직접 수정하는 것은 바람직하지 않다. 프로젝트에 새로운 폼을 추가하거나 삭제하면 델파이는 DPR 파일을 직접 수정하여 준다.

다음은 2장에서 처음 만들었던 Fruit 예제의 프로젝트 파일이다. 프로젝트의 소스를 보려면 Project/View Source 메뉴 항목을 선택하면 된다.

```
program Fruit;

uses
  Forms,
  Fruit_f in 'FRUIT_F.PAS' {Form1};

{$R *.RES}

begin
```

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

프로젝트 파일은 사용자가 직접 작성하는 것이 아니므로 당장은 그 문법적 의미를 몰라도 상관없다. 물론 고급 프로그래밍 기법을 구사하려면 직접 편집해야 하는 경우도 있다.

### .PAS

이 파일은 델파이의 소스 파일이며 유닛 파일이라고 한다. 보통 하나의 폼에 하나의 유닛 파일이 생성되며(예외도 있다) 이 파일에 여러분들이 만든 변수와 이벤트 핸들러, 그리고 일반 함수들이 기록된다. Fruit 예제의 소스 파일인 Fruit_f.PAS를 보자.

**그림**
유닛 파일의 전체적인 구조

```
unit Fruit_f;

interface

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

type
  TForm1 = class(TForm)
    BtnApple: TButton;
    BtnOrange: TButton;
    Fruit: TLabel;
    procedure BtnAppleClick(Sender: TObject);
    procedure BtnOrangeClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.BtnAppleClick(Sender: TObject);
begin
  Fruit.Caption:='Apple';
end;

procedure TForm1.BtnOrangeClick(Sender: TObject);
begin
  Fruit.Caption:='Orange';
end;

end.
```

델파이가 작성하는 부분

사용자가 만든 이벤트 핸들러

텍스트 파일이므로 코드 에디터에서 직접 수정할 수 있으며 폼에서 컴포넌트를 더블클릭할 경우 이벤트 핸들러를 작성할 수 있도록 코드 에디터를 열어준다. 사용자가 작성하는 부분은 일부일 뿐이며 나머지 윗부분은 대부분 델파이가

자동으로 작성해 준다.

### .DFM

DFM은 Delphi ForM의 준말이다.

폼의 모양을 기억하며 폼 하나에 대해 하나씩 생성된다. 폼 안에 어떤 컴포넌트가 어떤 속성을 가지며 놓여져 있는지가 보관되어 있으며 많은 양의 정보를 가지고 있기 때문에 텍스트 파일이 아닌 이진 파일로 되어 있다. 이 파일을 사용자가 직접 건드릴 필요는 없으며 화면 상에 보이는 폼을 디자인하므로써 간접적으로 편집한다. 즉 컴포넌트를 배치하거나 속성을 변경하면 델파이가 이 파일을 적당히 편집해 준다.

물론 꼭 필요할 경우 텍스트 파일로 바꾸어 편집할 수도 있지만 가급적이면 델파이가 이 파일을 관리하도록 맡겨두는 것이 좋다. DFM 파일을 꼭 보고 싶으면 File/Open 항목을 선택해 코드 에디터에 텍스트 형식으로 열어 볼 수도 있다. 어떤 모양을 가지고 있는지 처음이자 마지막으로 구경이나 해 보도록 하자. 다음은 Fruit_f.DFM 파일의 내용이다.

```
object Form1: TForm1
  Left = 137
  Top = 441
  Width = 301
  Height = 227
  Caption = 'My First Program'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 16
  object Fruit: TLabel
    Left = 98
    Top = 76
    Width = 53
    Height = 31
    Caption = 'Fruit'
    Font.Color = clBlack
    Font.Height = -27
    Font.Name = 'Times New Roman'
    Font.Style = []
    ParentFont = False
  end
  object ButApple: TButton
    Left = 30
```

```
  Top = 124
  Width = 89
  Height = 33
  Caption = 'Apple'
  TabOrder = 0
  OnClick = ButAppleClick
 end
 object ButOrange: TButton
  Left = 158
  Top = 124
  Width = 89
  Height = 33
  Caption = 'Orange'
  TabOrder = 1
  OnClick = ButOrangeClick
 end
end
```

폼에 배치한 컴포넌트들의 속성들이 보관되어 있다. 이 파일은 사용자가 직접 편집해야 할 경우가 전혀 없으므로 델파이에게 맡겨 두도록 하자.

이 세 개의 파일이 델파이 프로그램의 주요 소스 파일이다. 즉 소스 백업을 보관한다거나 소스를 남에게 배포할 경우 이 파일들을 모두 포함해야 하며 이 파일 중 하나라도 빠지면 컴파일이 불가능하다. 이 파일들 외에도 프로젝트의 옵션 선택 상태를 기억하는 .DOF 파일과 프로젝트 외부에 존재하는 리소스에 대한 정보를 담는 .RES 파일, 데스크 탑의 설정 상태를 기억하는 .DSK 파일이 디자인 과정에서 생성된다. 그리고 프로젝트를 저장할 때마다 .~DP, .~PA, .~DF 등의 백업 파일이 만들어진다. 디자인 과정에서 만들어지는 이런 파일 외에도 컴파일 과정에서 다음과 같은 파일들이 만들어진다.

### .DCU

.PAS 파일을 컴파일하여 만든 목적 파일이다. 도스 프로그래밍에 비유한다면 .OBJ 파일에 해당한다고 할 수 있다. 컴파일 중간에 만들어지는 파일이므로 언제든지 삭제해도 되며 소스를 배포할 때 포함할 필요가 없다.

### .EXE

최종적으로 만들어지는 실행 파일이다. 델파이가 만들어 내는 실행 파일은 완

전한 단독 실행 파일이므로 만든 프로그램을 배포할 경우 이 파일만 배포하면 된다. 실행 파일의 이름은 프로젝트의 이름과 동일하다.

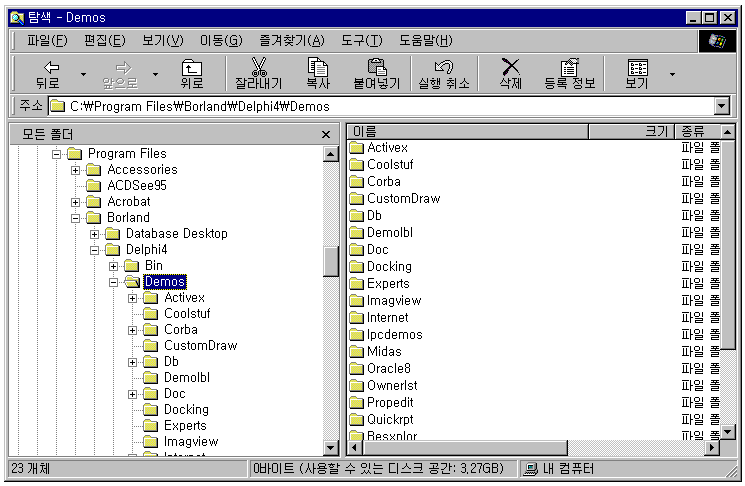
위에서 열거한 파일들이 델파이가 디자인 과정과 컴파일 과정에서 만들어 내는 파일들의 전부이다. 그러나 이 파일들 외에도 프로젝트를 이루는 외부적인 파일들이 있을 수 있다. 대표적으로 BMP 파일이나 WMF 파일, ICO 파일, 그리고 HLP 파일이 여기에 해당한다. 이런 파일들도 프로젝트를 이루는 구성 요소이므로 백업을 할 때 같이 저장해야 한다.

디자인시에 사용한 BMP와 같은 그림 파일은 일단 컴파일되면 실행 파일에 모두 포함되므로 완성된 작품을 배포할 때는 같이 배포하지 않아도 되지만 미래의 수정을 위해 보관해 두는 것이 좋다. 반면 HLP 파일은 실행 파일과는 달리 독립적으로 존재하므로 같이 배포하여야 한다.

## 나. 디렉토리 관리

위에서 보다시피 델파이 프로젝트를 이루는 파일의 종류가 이렇게 많다. 델파이 자체가 만들어 내는 파일도 많지만 그래픽이나 사운드를 삽입하고 데이터 베이스 파일들까지 사용하려면 파일의 종류는 많을 경우 수십 개를 헤아리게 된다. 이렇게 파일이 많아지다보면 관리하는 일도 만만치 않다.

델파이 프로그래밍을 할 때는 프로젝트별로 별도의 디렉토리를 만드는 것이 좋다. 물론 꼭 그래야 하는 것은 아니며 프로젝트를 이루는 구성 파일을 하드 디스크의 여기저기에 흩어 놓고 경로만 제대로 지정해 주어도 상관은 없다. 하지만 백업을 받는다든가 위치를 바꿀 일이 있을 경우를 생각한다면 하나의 디렉토리에 하나의 프로젝트를 보관하는 것이 가장 합리적이며 현명한 방법이다. 델파이도 자신이 제공하는 모든 예제 프로젝트들을 DEMOS 디렉토리 아래에 별도의 서브 디렉토리별로 보관해 놓았다.



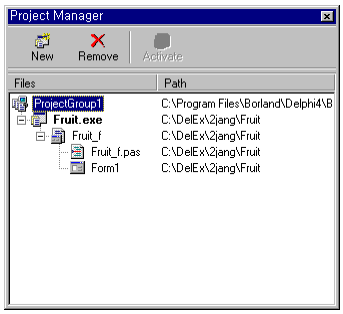

도스용 C의 경우는 프로젝트를 구성하지 않아도 소스만 있으면 컴파일되는 경우도 있지만 델파이는 반드시 프로젝트를 구성하므로 별도의 디렉토리를 작성해야 한다. 이 책의 배포 CD에도 예제 하나당 하나의 디렉토리를 할당하고 있다. 여러분도 실습을 하려면 별도의 디렉토리를 만든 후 프로젝트를 저장하는 습관을 들이도록 하자. 사소한 것 같지만 필자의 조언을 무시하면 언젠가는 낭패를 당하게 될 것이다.

# 다. 프로젝트 관리자

프로젝트를 구성하는 파일들은 가급적이면 한 디렉토리에 모아두는 것이 좋지만 불가피하게 그렇게 하지 못할 경우가 있다. 하나의 폼을 두 개의 프로젝트가 공유하고자 한다거나 다른 프로젝트의 모듈을 잠시 빌려 쓰고자 할 때가 이 경우에 해당된다. 이때 어떤 구성 파일이 어떤 디렉토리에 있는가를 알아 내고 싶을 때는 프로젝트 관리자를 사용한다. 프로젝트 관리자는 프로젝트를 구성하는 파일들의 목록을 보여주며 보관되어 있는 경로를 알려줄 뿐만 아니라 구성 파일을 추가, 삭제, 편집할 수 있는 서비스를 해준다.

프로젝트 관리자는 디폴트로 열려있지 않으므로 화면에 나타나게 하려면 View/ Project Manager 메뉴 항목을 선택해서 열어 주어야 한다. 다음은 Fruit.dpr 예제의 프로젝트 관리자 윈도우이다.

그림
프로젝트 관리자




윗부분에 세 개의 버튼이 있고 아래쪽에는 프로젝트에 속한 구성 파일의 목록이 표시되어 있고 각 구성 파일의 위치가 그 옆에 나타나있다. 목록에서 폼이나 유닛을 선택하면 선택한 항목을 활성화시켜 준다. 예를 들어 Fruit_f.pas를 더블클릭하면 코드 에디터에 이 유닛을 보여주며 Form1을 더블클릭하면 폼을 보여준다. 화면 상단에 있는 버튼들의 기능을 간략하게 정리하였다.

표
프로젝트 관리자의 버튼들

| 버튼 | 설명 |
|---|---|
|   | 프로젝트에 구성 파일을 추가한다. 이 버튼을 누르면 New Items 대화상자가 열리며 이 대화상자에서 폼이나 프로젝트를 추가할 수 있다. |
|   | 프로젝트를 그룹에서 제거한다. 단 프로젝트 자체를 디스크에서 삭제하는 것은 아니다. |
|   | 여러 개의 프로젝트가 그룹에 속해 있을 때 선택한 프로젝트를 활성화시킨다. |

프로젝트 관리자로 여러 개의 폼을 가지는 프로젝트를 만들 수 있다. 그러나 여기서는 버튼의 기능에 대해서만 설명하기로 한다. 왜냐하면 복수 개의 폼을 가지는 프로젝트를 만들려면 프로젝트 관리자를 쓰는 방법 외에 폼끼리 정보를 교환하는 방법, 폼을 호출하는 방법 등의 문법적인 사전 학습이 전제되어야 하기 때문이다. 차후에 실습을 하면서 그때 그때 다시 언급하기로 하고 프로젝트는 프로젝트 관리자로 관리한다는 사실만 알고 가도록 하자.

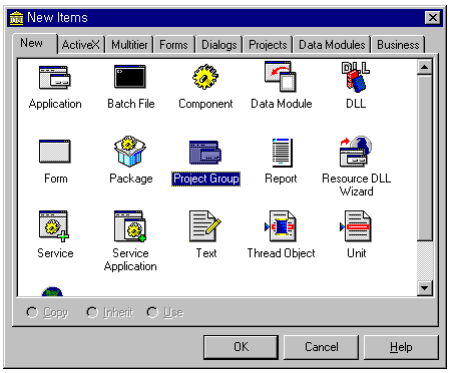
# 라. 프로젝트 그룹

델파이 4에서 새로 추가된 것 중의 하나가 프로젝트 그룹(Project Group)이

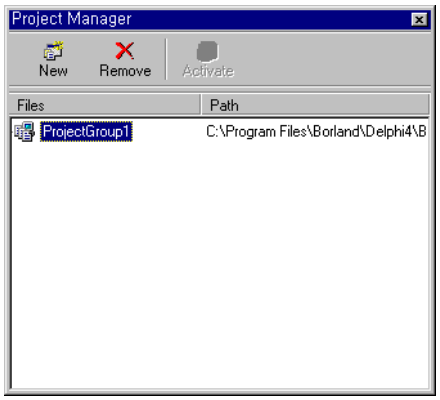
다. 프로젝트 그룹은 프로젝트보다 한단계 더 높은 상위의 개념으로 여러 개의 프로젝트 집합을 프로젝트 그룹이라고 한다. 프로젝트 하나로 하나의 실행 파일을 만들어 내는데 하나의 제품은 여러 개의 실행 파일로 구성되는 것이 보통이다. 지금 여러분들이 쓰고 있는 델파이만 해도 메인 프로그램인 Delphi32.exe 뿐만 아니라 이미지 에디터, DBD, BDE 관리자, WinSight 등의 유틸리티가 포함되어 있다. 가장 전형적인 예는 하나의 실행 파일과 여러 개의 DLL로 구성된 제품을 들 수 있는데 대부분의 프로그램은 이런 형태로 작성된다. 이 때 실행 파일과 DLL은 별도의 프로젝트를 구성해야 하며 그래서 프로젝트를 번갈아가며 개발을 진행해야 한다. 프로젝트를 번갈아 열었다 닫았다 하는 일은 무척 성가신 일이며 또한 작업 효율을 떨어뜨린다.

프로젝트 그룹은 이런 관련성 있는 프로젝트를 한 그룹에 모아 놓고 동시에 열어서 작업을 할 수 있어 대규모 프로젝트 개발에 아주 유용하다. 초보자에게는 아직 별 실용성이 없는 기능이겠지만 프로젝트 관리자를 이해하기 위해 프로젝트 그룹을 만드는 실습을 잠깐 해 보도록 하자. 이 책을 처음 읽고 있다면 이 부분은 당분간 건너 뛰어도 좋다. 3장에서 만들었던 예제 세 개를 한 그룹에 넣어 볼 것이다. 메뉴에서 File/New 항목을 선택하고 New Items 대화상자에서 Project Group을 선택한다.

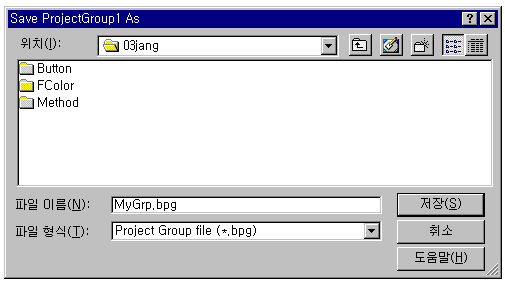
**그림**
프로젝트 그룹 만들기



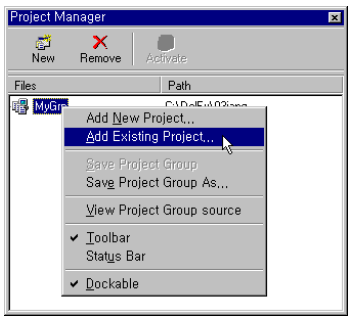

그러면 다음과 같은 빈 프로젝트 그룹을 만들어 줄 것이다. 그룹만 만들어졌기 때문에 프로젝트는 전혀 없으며 프로젝트에 속한 구성 파일들도 당연히 없다.




프로젝트 그룹은 디폴트로 Delphi 4/Bin 디렉토리에 ProjectGroup1이라는 이름으로 만들어지는데 일단 이 그룹을 저장해 보자. 3장의 프로젝트를 담을 것이므로 각 프로젝트의 부모 디렉토리인 03jang 디렉토리에 저장하는 것이 적합할 것 같다. File/Save 를 선택하거나 스피드 버튼의  를 눌러서 저장 대화상자를 불러내 MyGrp.bpg로 저장한다.



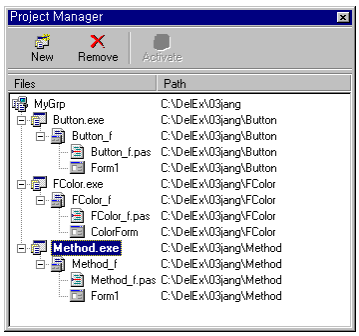

프로젝트 그룹은 확장자 bpg로 저장된다. 이제 이 프로젝트 그룹에 새로운 프로젝트를 포함시켜 보자. MyGrp에서 오른쪽 마우스 버튼을 눌러 팝업 메뉴를 호출한다.



이 메뉴 아래에 Add New Project, Add Existing Project라는 항목이 있는데 각각 새 프로젝트 만들기와 기존 프로젝트 추가해 넣기이다. 우리는 지금 기존에 만들어져 있는 프로젝트를 그룹에 추가할 것이므로 두 번째 항목인 Add Existing Project 항목을 선택한다. 그리고 3장의 Button.dpr을 선택해 이 프로젝트를 그룹에 포함시키고 같은 방법으로 FColor, Method 프로젝트도 포함시킨다. 여기까지 실습을 진행하면 세 개의 프로젝트가 그룹에 포함되며 프로젝트 관리자에는 다음과 같이 나타날 것이다.

**그림**

세 개의 프로젝트를 포함하고 있는 프로젝트 그룹




세 개의 프로젝트가 그룹에 포함되어 있다. 이제 세 프로젝트가 동시에 열려 있으므로 어떤 폼이나 유닛이라도 열어서 수정할 수 있다. 프로젝트 그룹이 열려있는 상태에서 컴파일, 실행의 대상이 되는 프로젝트는 활성화된 프로젝트 하나뿐이며 활성화된 프로젝트는 프로젝트 그룹에서 굵은 글꼴로 나타난다. 다른 프로젝트를 활성화시키려면 프로젝트 이름을 더블클릭하거나 아니면 위쪽의 Activate 버튼을 누르면 된다.

프로젝트 그룹 자체의 소스는 다음과 같다. 이 소스를 직접 편집할 일은 없을 것이고 프로젝트 관리자를 통해 간접적으로 편집하겠지만 모양이 어떤지 구경이나 해 보도록 하자.

```
#------------------------------------------------------------------------
VERSION = BWS.01
#------------------------------------------------------------------------
!ifndef ROOT
ROOT = $(MAKEDIR)₩..
!endif
#------------------------------------------------------------------------
MAKE = $(ROOT)₩bin₩make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)₩bin₩dcc32.exe $**
BRCC = $(ROOT)₩bin₩brcc32.exe $**
#------------------------------------------------------------------------
```

```
PROJECTS = Button.exe FColor.exe Method.exe
#------------------------------------------------------------------------
default: $(PROJECTS)
#------------------------------------------------------------------------

Button.exe: Button₩Button.dpr
  $(DCC)

FColor.exe: FColor₩FColor.dpr
  $(DCC)

Method.exe: Method₩Method.dpr
  $(DCC)
```

몇 가지 간단한 설정 사항과 그룹에 속한 프로젝트들의 경로가 포함되어 있다.

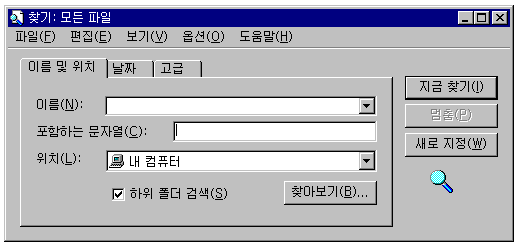

① 프로젝트:하나의 프로그램을 이루는 구성 파일에 대한 정보
② 주요 소스 파일은 DPR, PAS, DFM 세 가지이다.
③ 델파이 프로젝트는 디렉토리 단위로 저장하는 것이 좋다.

# 4-3 에디트 컴포넌트

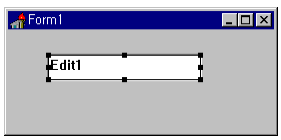
## 가. 에디트 박스

프로그램은 자기 혼자서 실행되는 것이 아니라 사용자에게 명령을 받아서 사용자가 내린 명령을 실행하므로 끊임없이 사용자로부터 명령이나 정보를 입력받아야 할 필요가 있다. 간단한 명령은 버튼 컴포넌트로 해결할 수 있지만 좀 더 복잡한 정보는 마우스로 입력하기가 어렵다. 사람 이름이나 파일 이름, 숫자값, 문장 등을 입력받을 때가 그렇다. 윈도우즈는 실행중에 문자열을 입력받아야 할 경우를 위해 에디트 박스(편집 박스)를 제공한다.

윈 98의 시작 메뉴에서 찾기 명령을 실행해 보면 다음과 같은 대화상자가 나타날 것이다.




찾고자 하는 파일의 이름이나 포함하는 문자열 등을 키보드로 입력받아야 하는데 이때 에디트 박스가 사용된다. 델파이가 제공하는 에디트 컨트롤이 바로 에디트 박스이다. 에디트 컨트롤이 포커스를 가질 경우 컨트롤 내에 캐럿이 나타나며 키보드로부터 문장을 입력받는다. Enter 키를 눌러 입력을 완료하기 전에 BS, Del, 커서 이동키 등으로 제한적인 편집을 할 수도 있다. 폼에 배치하면 다음과 같은 모양을 가지며 크기 조정이나 위치 변경이 자유롭다.




에디트 컨트롤에 입력된 문장은 Text 속성을 읽음으로써 알 수 있으며 반대로 Text 속성을 변경함으로써 에디트 컨트롤의 내용을 바꿀 수 있다. 주요 속성으

로는 AutoSelect, ReadOnly, MaxLength 등이 있다.

✍ **AutoSelect**

실행중에 Tab키를 사용하여 에디트 박스로 포커스를 옮기면 에디트 박스내의 문자열을 자동으로 블럭 선택해 주어 곧바로 에디트 내의 문자열을 다른 문자열로 바꿀 수 있도록 해준다. 블럭 선택이 되어 있는 상태에서 다른 문자열을 입력하면 블럭 부분이 입력된 문자열로 바뀌기 때문이다. 이 속성의 디폴트값은 True로 설정되어 있다.

✍ **ReadOnly**

에디트의 문자열을 읽을 수만 있도록 하며 실행중에 사용자가 문자열을 편집할 수 없도록 한다. 입력을 위해서 에디트를 사용할 경우는 당연히 ReadOnly 속성을 False로 설정해야겠지만 사용자에게 문자열을 보여주기만 할 경우는 ReadOnly 속성을 True로 만들어 읽기 전용의 에디트로 사용할 수 있다.

✍ **MaxLength**

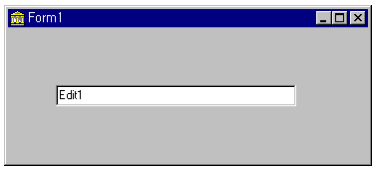
에디트 박스에서 입력받을 수 있는 문자열의 길이를 제한한다. 예를 들어 에디트 컨트롤에서 주소를 입력받아 변수에 대입하고자 하는데 변수의 길이가 30바이트로 정해져 있다면 입력받는 값도 30바이트 이하여야 한다. 그렇지 않으면 30바이트짜리 변수에 30바이트 이상의 값이 입력되어 메모리가 파괴되고 엉뚱한 오동작을 유발할 수 있다. 이럴 때 MaxLength로 입력받을 수 있는 값을 제한한다. 만약 사용자가 실행중에 MaxLength 이상의 문자를 입력하고자 하면 경고음을 내고 입력을 거부한다. 디폴트값은 0이며 0은 입력 한계가 없다는 뜻이다.

# 나. OnChange 이벤트

에디트 컴포넌트의 이벤트 중 가장 자주 발생하는 이벤트는 OnChange 이벤트이며 에디트의 문장, 즉 Text 속성이 변경할 때 발생한다. 이 이벤트가 에디트 컴포넌트의 디폴트 이벤트(가장 자주 발생하는 이벤트)이므로 폼에서 에디트를 더블클릭하면 곧바로 이 이벤트에 대한 핸들러를 작성할 수 있다.

4jang
change

그럼 OnChange 이벤트를 사용하는 간단한 예제를 만들어 보자. 새 프로젝트를 시작하고 폼에 에디트 컴포넌트 하나만 배치한 후 크기를 옆으로 길게 늘여준다. 간단한 예제이므로 속성은 하나도 건드릴 필요가 없다. 에디트를 배치한 후의 모습은 다음과 같다.

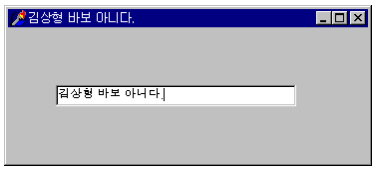



에디트의 OnChange 이벤트 핸들러를 작성하기 위해 에디트 컴포넌트를 더블클릭한다. 코드 에디터가 열리면 다음과 같이 코드를 작성하자.

begin 과 end 사이의 코드만 사용자가 입력하며 나머지는 델파이가 만들어 준다.

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Form1.Caption:=Edit1.Text;
end;
```

이 코드의 의미는 에디트 박스의 내용이 변경되면 폼의 캡션, 즉 타이틀 바에 나타나는 제목을 에디트에 입력된 내용으로 변경한다는 뜻이다. 프로그램을 실행시키고 에디트의 문장을 수정해 보아라. 다음과 같이 폼의 타이틀이 따라서 변경될 것이다.




OnChange 외에도 에디트 컴포넌트가 가지는 이벤트는 여러 가지가 있지만 나머지는 별로 사용되지 않는다.

이 예제를 완전히 이해했다면 약간 변형시켜 보자. 에디트와 버튼을 하나씩 배치하고 버튼을 누를 때 에디트의 문자열로 폼의 캡션을 변경시키도록 하자. 아주 쉬운 문제이므로 직접 풀어보기 바란다. 만약 이 문제를 스스로 풀지 못한다면 아직도 델파이에 대한 기본 이해를 하지 못한 것이라 생각하고 앞부분을 좀 더 자세하게 읽어 보도록 하자.

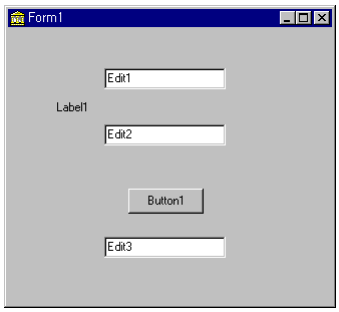

암기사항

① 에디트:문자열을 입력받을 때 사용하는 컴포넌트
② 에디트에 입력된 문자열은 Text 속성으로 읽는다.
③ Text 속성이 변할 때마다 OnChange 이벤트가 발생한다.

# 다. 계산기 예제

4jang
calc

에디트를 이용해 간단한 계산기를 만들어 보자. 에디트 컴포넌트를 쓰는 것은 무척 간단한 일이지만 에디트로부터 입력받은 정보를 가공하여 유용하게 사용하는 것은 쉬운 일이 아니다. 프로그램을 만들어 보기 위해 델파이를 실행하고 새로운 프로젝트를 시작한다. 비어있는 폼에 다음 그림과 같이 컴포넌트를 배치하자.

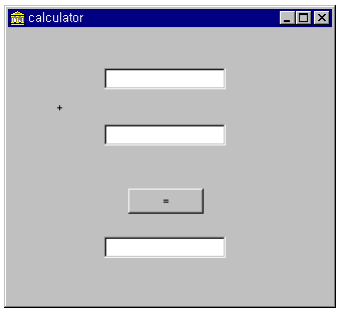



계산기를 만들기 위해 필요한 컴포넌트는 에디트 세 개, 버튼 하나, 레이블 하나이다. 물론 델파이가 기본적으로 만들어 주는 폼 컴포넌트도 포함된다. 컴포넌트의 배치는 꼭 위와 같을 필요는 없고 비슷하게만 만들면 된다. 배치가 끝났으면 각 컴포넌트의 속성을 다음과 같이 설정한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | Name | calculator |
| | Caption | calculator |
| 위쪽 에디트 | Name | num1 |
| | Text | 모두 지움 |
| 가운데 에디트 | Name | num2 |
| | Text | 모두 지움 |
| 아래쪽 에디트 | Name | result |
| | Text | 모두 지움 |
| 버튼 | Name | equal |
| | Caption | = |

| | | |
|---|---|---|
| 레이블 | Name | Label1 |
| | Caption | + |

위의 표는 컴포넌트의 속성을 어떻게 바꿀 것인가를 나타내며 디폴트 속성에서 변경되는 속성만 정리한 것이다. 이렇게 속성을 바꾸었으면 폼의 모양은 다음과 같이 된다.

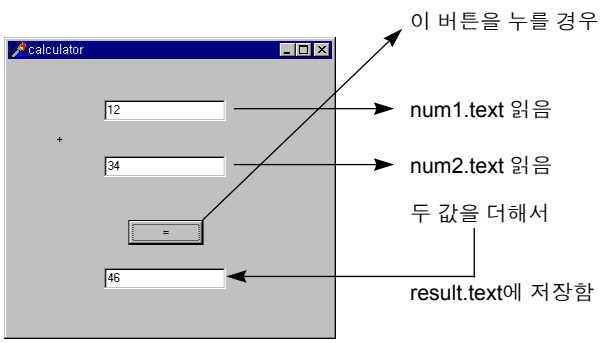



num1, num2 에디트 컨트롤에서 정수값을 입력한 후 equal 버튼을 누르면 두 개의 정수를 더하여 아래쪽의 result에 덧셈의 결과가 나타나도록 만들 계획이다. 에디트 컨트롤 자체에 값을 입력받고 표시하는 기능이 있으므로 입력 및 출력은 별도로 프로그래밍을 하지 않아도 된다.

우리가 프로그래밍을 해 주어야 할 부분은 equal 버튼이 눌러질 때 num1, num2의 값을 읽어 더한 후 result에 표시해 주는 것 뿐이다. 좀 더 문법적으로 표현하자면 equal 버튼에 OnClick 이벤트가 발생할 경우 덧셈을 하여 result로 출력해 주는 이벤트 핸들러를 만드는 것이다. 이벤트 핸들러를 만들기 위해 equal 버튼을 더블클릭하여 코드 에디터를 연 후 코드를 작성한다. 다음과 같이 작성하면 간단하게 해결될 것 같다.

```
procedure Tcalculator.equalClick(Sender: TObject);
begin
 result.text:=num1.text+num2.text;
end;
```

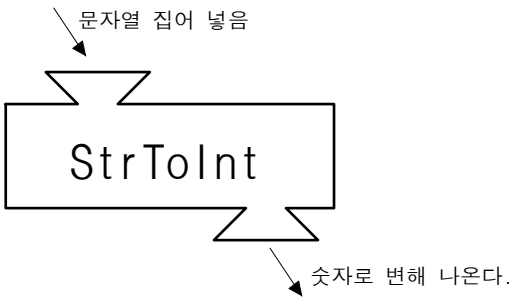
num1에 입력된 숫자값은 num1.text 속성에 저장되어 있고 num2에 입력된 숫자값은 num2.text 속성에 있으므로 이 두 값을 읽어 더한 값으로 result.text 속성을 정의하는 것이다. result.text 속성에 덧셈의 결과가 대입되므로 result 에디트에 계산 결과가 출력될 것 같다.




그러나 불행하게도 이렇게 코딩을 하고 프로그램을 실행시키면 원하던 결과가 나오지 않는다. 왜냐하면 에디트 컴포넌트가 입력받는 것은 문자열이지 숫자가 아니기 때문이다. num1에 입력된 12라는 값은 11보다 1 크고 6의 두 배인 숫자 12가 아니라 1이라는 문자와 2라는 문자가 단순히 연속해 있는 문자열일 뿐이다. 그래서 덧셈을 해도 수학적인 연산이 이루어지지 않고 문자열을 연결하기만 한다. num1에 12, num2에 34가 입력되었다면 계산 결과는 이 두 문자열을 연결한 1234가 되어 버린다.

이 문제를 해결하기 위해서는 에디트 컨트롤에 입력된 문자열을 숫자로 인식하도록 바꾸어 주어야 하는데 이 때 사용되는 함수가 StrToInt이다. StrToInt는 문자열을 숫자로 바꾸어 주어 수학적 연산을 할 수 있도록 해준다. StrToInt(num1.text) 등과 같이 숫자로 바꾸고자 하는 문자열을 괄호 안에 넣어 준다. StrToInt(num1.text) 자체는 숫자값을 가진다.

'12'




12

**그림**
문자열을 숫자로 변경하는 StrToInt 함수

그럼 다음과 같이 코드를 변경해 보자.

```
result.text:=StrToInt(num1.text)+StrToInt(num2.text);
```

숫자로 바꾸어 덧셈을 했으므로 덧셈은 제대로 이루어진다. 그러나 아직도 문제가 있다. 코드를 이렇게 바꾸어 놓고 F9를 눌러 실행시키면 Type mismatch라는 에러가 난다. 대입문의 좌변과 우변의 모양이 다르다는 얘기인데 모양이 다른 값을 대입할 수는 없기 때문이다. 문자열을 담는 변수(또는 속성)에 숫자를 담거나 숫자를 담는 변수(또는 속성)에 문자열을 대입하려고 할 때 이런 에러가 나는데, 왜 그런지 자세히 훑어보자.

```
result.text:=StrToInt(num1.text)+StrToInt(num2.text);
```

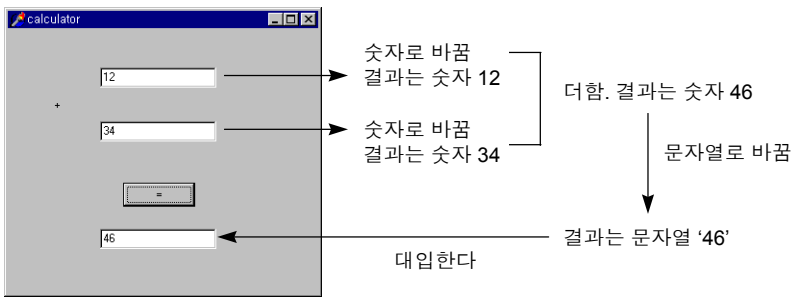
문자 / 문자 / 문자 / 숫자 / 숫자 / 숫자

숫자끼리 더하면 그 결과도 숫자가 되며 그 값을 대입받을 result.text는 문자열을 담는 속성이기 때문에 에러가 발생한다. 그래서 계산한 결과를 문자열로 다시 바꾸어 주어야 하며 코드는 다음과 같이 된다.

```
result.text:=IntToStr(StrToInt(num1.text)+StrToInt(num2.text));
```

IntToStr 함수는 StrToInt와 반대로 정수를 문자열로 바꾸어 준다. 전체 계산식을 괄호로 싸고 IntToStr을 앞에 위치시키면 깔끔하게 문제가 해결된다.

**그림** 에디트로 정수 입력 받기




이렇게 해야 제대로 계산기가 된다. 이런 변수형(type)의 불일치에 대한 문제와 함수에 관한 것은 문법적인 내용이므로 여기서는 IntToStr, StrToInt 함수에 대해서 맛만 보고 다음에 좀 더 자세하게 체계적으로 다루도록 한다. 이제 기본을 만들었으므로 좀 더 완벽하게 만들어 보자. 계산이 잘 되긴 하지만 한 가지 빼놓은 것은 에디트 컨트롤에 숫자가 입력되지 않은 상태에서 equal 버튼이 눌러지는 경우에 대해서이다. 이럴 경우 아무것도 없는 빈 문자열을 숫자로 바꿀 수

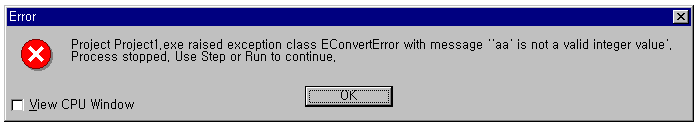
없어 에러가 발생하는데 다음과 같이 코드를 작성하면 문제를 해결할 수 있다.

```
if (num1.text='') or (num2.text='') then exit;
result.text:=IntToStr(StrToInt(num1.text)+StrToInt(num2.text));
```

if 문은 조건에 따라 명령 실행 여부를 결정하는 조건문이다.

num1이나 num2에 빈 문자열이 있을 경우는 계산이고 뭐고 다 집어치우고 함수를 끝내 버리라는 뜻이다. exit는 현재 블럭을 끝내는 기능을 가지며 exit 이후의 명령은 모두 무시되어 실행되지 않는다.

여기까지 계산기 예제 제작을 모두 마친다. 에디트 컨트롤로 문자를 입력받아 사용하는 방법에 대해 실습을 해 보았는데 뜻밖에도 변수의 형 변환과 에러 처리에 대해 논하게 되었다. 참고로 위에서 만든 계산기 예제는 완벽한 것이 아니다. 덧셈밖에 하지 못한다는 기능적인 결함은 둘째 치고 에러 처리가 섬세하지 못하다. 숫자가 입력되지 않았을 경우 그냥 끝내는 것 보다는 "숫자를 입력하십시요" 라는 메시지를 출력해 주는 것이 더 친절한 방법이며 숫자가 아닌 문자가 입력된 경우에  발생하는 다음과 같은 에러에 대해서는 처리를 하지 않았다.




다음에 실력이 더 늘면 직접 이 문제를 풀어보기 바라되 힌트를 준다면 StrToIntDef라는 함수를 사용하면 된다.

# 라. IME

IME(Input Method Editor)는 사용자로부터 키보드 입력을 받아 적합한 문자로 바꾸어 주는 프로세스를 말한다. 키보드의 A 를 누르면 A 가 입력되고 B 를 누르면 B 가 입력되는 것이 너무나 당연하겠지만 꼭 그렇지 않을 수도 있다. 아시아 3 국의 문자는 워낙 특이해서 이런 식으로 키 하나가 문자 하나가 아닐 수도 있는데 예를 들어 우리나라 대한민국의 경우는 A 를 누르면 미음(ㅁ)이 입력되고 B 를 누르면 유(ㅠ)가 입력되어야 한다. IME 는 주로 아시아 3 국의 문자 지원을 위해 존재하는데 영어만 사용한다면 신경쓰지 않아도 된다. 그런데 지금 이 책을 읽고 있는 여러분들은 신경쓰지 않을 수가 없을 것이다.

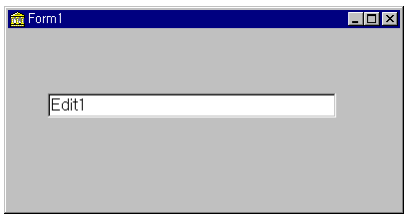
IME 와 관계된 속성으로는 ImeName, ImeMode 속성 두 가지가 있다. 우선 ImeName 은 시스템에 설치된 IME 중 어떤 것을 사용할 것인가를 지정하는데

한글 윈도우즈 98 에서는 '한국어(한글)'밖에 설치되어 있지 않으므로 변경할 필요가 없고 신경쓸 필요도 없다. 물론 일본이나 중국으로 수출될 제품을 개발중이라면 문제가 다르겠지만 말이다.

ImeMode 속성은 현재 설치된 IME 가 어떤 식으로 동작할 것인가를 지정하며 이 속성에 따라 키보드로 입력된 키가 어떻게 처리될 것인가가 결정된다. 디폴트는 imDontCare 이며 사용자가 최후로 설정한 모드대로 동작한다.

4jang
Ime

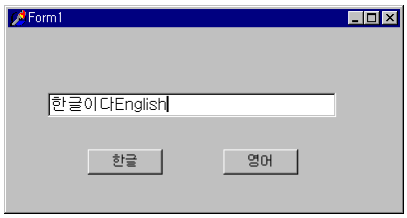
imeMode 를 imSHanguel 로 바꾸면 소프트웨어적으로 한글 입력 상태로 전환할 수 있으며 imSAlpha 로 바꾸면 영문 입력 상태로 전환할 수 있다. 실습을 위해 새 프로젝트를 시작하고 폼에 에디트 하나를 배치해 보자. 입력되는 글자가 잘 보이도록 하기 위해 폰트를 굴림체 10 으로 변경하였다.




이 프로젝트를 실행한 후 에디트에 문자열을 입력하면 영문이 입력될 것이다. 만약 이 에디트가 이름이나 주소를 입력받는 용도로 사용된다면 사용자들은 항상 '한글'키나 Shift+Space 키를 눌러 한글로 전환한 후 입력을 해야 하므로 무척 귀찮을 것이다. 처음부터 한글을 입력받는 용도로 사용하고 싶다면 ImeMode 를 imSHanguel 로 변경해 주면 된다. ImeMode 속성을 사용하면 에디트에 입력되는 문자를 소프트웨어적으로 변경할 수 있다. 테스트를 위해 버튼 두 개를 배치하고 이 버튼들의 이벤트 핸들러를 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.ImeMode:=imSHanguel;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit1.ImeMode:=imSAlpha;
end;
```

Button1 을 누르면 에디트에 한글이 입력될 것이며 Button2 를 누르면 에디트에 영문이 입력될 것이다.




ImeMode 속성에 대입 가능한 값은 imHira, imKata, imChinese 등도 있지만 이 값들은 일본, 중국에서 사용 가능하므로 한글 윈도우즈에서는 나타나지 않는다. 또 같은 한글이라도 imSHanguel 값이 있고 imHanguel이라는 값이 있고 영문도 imSAlpha, imAlpha가 있는데 S가 붙은 값은 영문이 한글의 절반 폭을 가지는 반각이며 S가 붙지 않은 값은 영문과 한글의 폭이 같은 전각이다.


과거 도스 시절에는 전각 문자라는 것이 많이 쓰였던 것 같은데 요즘은 잘 쓰지 않는 것 같다. imDontCare는 사용자가 설정한 대로 따르겠다는 속성인데 Shift+Space나 '한글'키로 사용자가 입력 상태를 설정한 대로 동작한다. 좀 복잡해 보일지도 모르겠는데 기억해 둘만큼 중요한 건 딱 한 가지 뿐이다. 에디트가 처음부터 한글을 입력받아야 한다면 ImeMode 속성을 imSHanguel로 바꾸어 두면 된다.

# 4-4 풍선 도움말

풍선 도움말의 좀 더 공식적인 이름은 툴팁(ToolTip)이다.

델파이는 풍선 도움말을 제공한다. 컴포넌트 팔레트나 스피드 버튼 위에 마우스 커서를 가져다 놓고 잠깐만 기다리면 노란색의 박스를 열어 짧은 도움말을 보여 준다. 별 것 아닌 것 같지만 델파이를 처음 시작하는 사람에게는 무척이나 많은 도움을 주며 머리가 좀 띨띨해도 컴포넌트를 정확하게 선택할 수 있도록 해준다. 이런 식의 도움말 방식을 풍선 도움말이라고 하는데 애초에 네모진 박스보다 타원형을 더 많이 사용했기 때문에 붙여진 이름이다. 다음은 오거나이저라는 프로그램에서 볼 수 있는 전형적인 풍선 도움말이며 우리나라의 한메 소프트에서 만든 파피루스에서도 이런 도움말을 볼 수 있다.

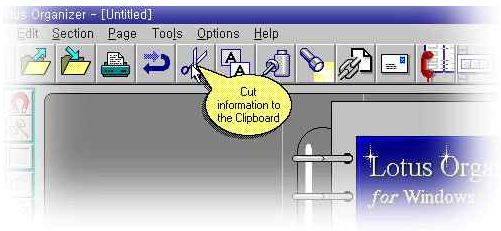



그림
전형적인 모양의 풍선 도움말

이렇게 좋은 풍선 도움말을 델파이의 도움을 받으면 몇 가지 속성만 조정하는 간단한 방법으로 직접 만들 수 있다. 앞에서 만들었던 계산기 예제에 풍선 도움말 기능을 넣어보기로 한다. 우선 풍선 도움말에 관계된 두 가지 속성을 보자. 이 속성들은 거의 모든 컴포넌트에 공통적으로 존재한다.

### ✍ Hint

도움말 문자열이다. 여기에 지정한 문자열이 풍선 안에 나타나게 된다. “번호를 입력하세요”, “검색을 시작합니다”, “알아서 하세요” 등과 같은 문자열을 대입해 주면 된다. 한글을 사용해도 아무런 문제가 없다.

### ✍ ShowHint

컴포넌트가 도움말을 출력할 것인가 아닌가를 지정하며 디폴트값은 False이다. 그래서 풍선 도움말을 설정하려면 Hint 속성에 도움말만 입력한다고 되는 것이 아니라 이 속성을 True로 바꾸어 주어야 한다. 이 속성이 존재하는 이유는 프로그램 실행중에 풍선 도움말을 ON/OFF시킬 수 있도록 하기 위해서이다.

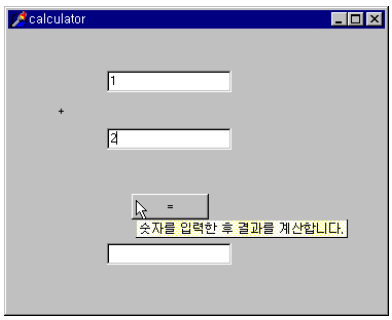
이 두 속성만 알면 일단 풍선 도움말을 만들 수 있다. 도움말을 표시할 컴포넌트의 ShowHint 속성을 True로 바꾸어 주고 Hint 속성에 원하는 도움말을 입력해 주면 된다. 계산기 예제를 다시 불러와서 각 컴포넌트의 Hint 속성에 다음과 같이 도움말을 설정해 보자.

| 컴포넌트 | 도움말(Hint 속성) |
|---|---|
| 위쪽 에디트 | 더할 숫자를 입력하세요. |
| 가운데 에디트 | 더해질 숫자를 입력하세요. |
| 버튼 | 숫자를 입력한 후 결과를 계산합니다. |
| 아래쪽 에디트 | 계산 결과를 보여줍니다. |

그리고 ShowHint 속성을 True로 만들어만 주면 실행중에 풍선 도움말이 출력된다. 실행중의 모습은 다음과 같다. 마우스 커서가 움직이는대로 풍선 도움말이 출력될 것이다.

**그림**

델파이가 제공하는 풍선 도움말




풍선 도움말은 에디트, 버튼은 물론이고 레이블, 체크 박스, 라디오 버튼, 스크롤 바 등등 웬만한 컴포넌트에는 다 사용할 수 있으며 폼 자체에도 풍선 도움말을 사용할 수 있다. 여기서 알아본 풍선 도움말 작성 방법은 아주 간단한 방법이다. 이 외에도 OnHint 이벤트를 사용하여 상태란에 도움말을 별도로 출력한다거나 폼의 도움말을 전체 컴포넌트가 공유하는 좀 더 고급적인 기법들이 있다.

참고하세요

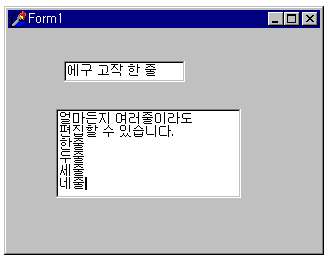
델파이가 기본적으로 제공하는 풍선 도움말은 사용하기에 편리하기는 하지만 별로 예쁘지는 못하다. 꼭 사각형으로만 나오며 글꼴을 바꾸기가 쉽지 않다. 좀 더 잘 만들어진 풍선 도움말 컴포넌트가 공개용으로 배포되고 있으므로 필요한 사람은 그런 컴포넌트를 구해서 사용해 보기 바란다

# 4-5 메모 컴포넌트

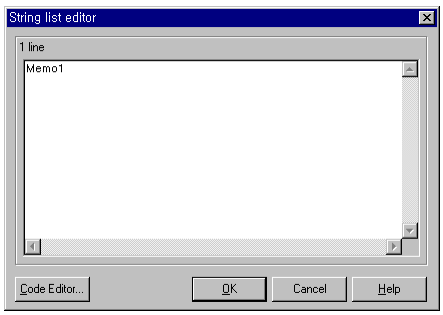
## 가. 메모 컴포넌트

운영체제 차원에서 보면 에디트와 메모는 스타일이 다른 같은 컨트롤이다.

메모 컴포넌트는 에디트의 확장판이다. 에디트의 모든 기능을 그대로 가지면서 몇 가지 추가된 기능들을 더 가지고 있으므로 에디트의 사촌형 정도 된다. 일단 외관상으로 드러나는 차이점이라면 에디트보다는 메모가 크기가 조금 더 크며 기능상으로 가장 두드러지는 차이점은 에디트가 한 줄밖에 입력을 할 수 없는 한계를 가지고 있는데 비해 메모 컴포넌트는 여러 줄의 문장을 입력하거나 편집할 수 있다는 점이다.




메모 컴포넌트에 입력되어 있는 문자열은 에디트 컴포넌트의 경우와 마찬가지로 Text 속성으로 읽거나 쓴다. 메모는 Text 속성 외에도 문자열을 개별적으로 읽을 수 있는 속성이 따로 제공된다. 메모의 Lines 속성을 사용하면 여러 줄의 문장을 입력시킬 수 있으며 각 줄의 문자열을 개별적으로 취급할 수 있다. 오브젝트 인스펙터에서 Lines를 더블클릭하면 여러 줄의 문자열을 입력할 수 있는 문자열 리스트 편집기가 열린다. 여기에 원하는 문자열들을 마치 워드 프로세서를 쓰듯이 입력하고 편집할 수 있다.



스피드 메뉴에서 Load.. 항목을 선택하면 미리 작성되어 있는 텍스트 파일로부터 읽어올 수 있으며 Save.. 항목을 선택하면 텍스트 파일로 저장할 수 있다. Code Editor 버튼은 코드 에디터에서 직접 Lines 속성을 편집할 수 있도록 한다. Lines 속성을 편집할 때마다 별도의 편집기를 열지 않아도 되므로 대단히 편리하다. 코드 에디터에서 문장을 편집한 후 스피드 버튼의 저장 버튼( )을 눌러 주어야 메모의 Lines 속성이 실제로 변경된다.

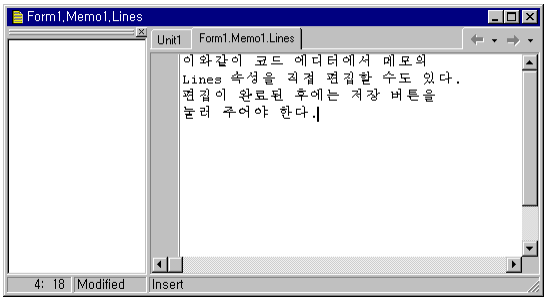



**그림**
코드 에디터에서 문자열 리스트를 직접 편집할 수 있다.

입력된 문자열들을 행단위로 개별적으로 읽어내거나 편집하고 싶을 때는 Lines 속성에 첨자를 붙여준다. Lines 속성의 데이터형은 TStrings이며 일종의 문자열 배열이라고 생각할 수 있다. Lines[0]는 첫 번째 행, Lines[1]은 두 번째 행을 나타내며 Lines[n]은 n+1번째 행을 나타낸다. 배열의 첫 첨자가 0번임을 주의하도록 하자.

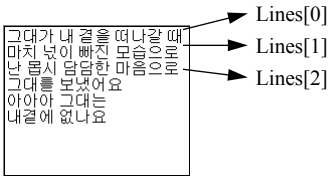
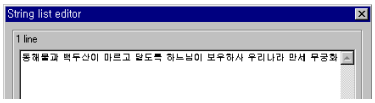



**그림**
Lines 속성으로 개별 행의 문자열을 읽는다.

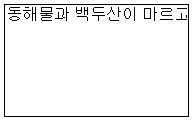
메모와 에디트의 기능이 비슷하므로 AutoSelect, ReadOnly, MaxLength 등 가지는 속성도 거의 동일하다. 에디트에 없고 메모에만 있는 독특한 속성은 여러 줄을 편집할 수 있다는 사실에서 기인되는 것이며 앞에서 살펴본 Lines 속성이 있고 그리고 WordWrap이라는 속성이 있다.

WordWrap 속성은 메모 컴포넌트의 가로폭 길이에 맞게 문자열들을 잘라 다음 줄로 자동 개행시켜 주는 속성이며 디폴트값이 True이다. 문자열을 입력할 때 Enter 키를 직접 넣어주면 당연히 개행되지만 한 줄에 길게 써 넣은 문자열은 WordWrap 속성에 따라 자동 개행되어 다음 줄에 표시되거나 메모 컴포넌트 밖으로 벗어나 보이지 않게 된다. 문자열 리스트에서 애국가 1절을 몽땅 한 줄에

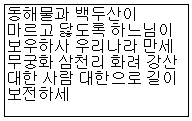
입력한 후 WordWrap 속성을 바꾸어 보면 쉽게 의미를 파악할 수 있을 것이다.




문자열 리스트 편집기

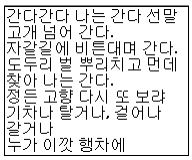



WordWrap=False

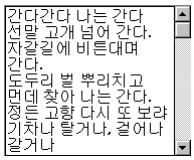



WordWrap=True

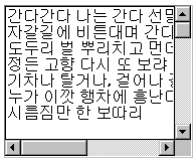
WordWrap 속성이 True일 경우는 문자열이 우측으로 삐져나가 잘리는 경우가 없다. 문자열이 너무 길어 아래쪽으로 잘리는 경우가 있을 수는 있지만 말이다. WordWrap이 False일 때는 잘려나간 문자열을 볼 수 있도록 수평 스크롤바를 달아 주어야 하며 WordWrap이 True이더라도 아래쪽으로 잘려 나간다면 수직 스크롤 바를 달아 주어야 한다.




없음




수직




수평, 수직

스크롤 바는 메모의 Scrollbars 속성으로 설정하며 디폴트값이 ssNone이므로 메모 컴포넌트를 처음 만들면 스크롤바 없이 만들어진다.

# 나. 간단한 에디터 1


4jang
editor1

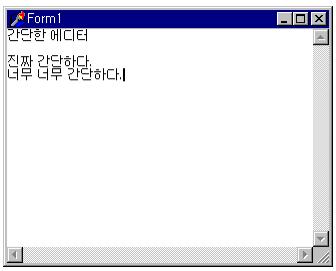
메모 컴포넌트를 사용하여 아주 간단한 에디터를 만들어 보자. 너무 간단하기 때문에 델파이 초보자라도 왼손으로 1분만에 만들 수 있다. 새로운 프로젝트를 시작하고 폼에 메모 컴포넌트를 하나만 배치한다. 그리고 메모 컴포넌트의 속성을 다음과 같이 설정한다.

| 속성 | 속성값 | 설명 |
|---|---|---|
| Align | alClient | 폼 전체 영역을 차지하도록 한다. |

| | | |
|---|---|---|
| BorderStyle | bsNone | 메모의 테두리를 없앤다. |
| ScrollBars | ssBoth | 스크롤 바는 수평, 수직 두 개 다 설치한다. |
| Wordwrap | True | 자동 개행이 되도록 한다. |
| Lines | 모두 지움 | 빈 문서를 작성할 수 있도록 한다. Lines 속성을 더블클릭한 후 문자열 리스트에서 Memo1을 삭제한다. |
| Font | 굴림체 10포인터 | 디폴트 폰트가 영문에 어울리게 되어 있으므로 한글에 적합한 굴림체로 변경한다. |

Align 속성은 컴포넌트가 폼의 일정 위치에 있도록 정렬한다. Align 속성의 디폴트는 alNone이며 이는 정렬을 하지 않고 디자인시에 설정해준 위치를 그대로 사용한다는 뜻이다. Align 속성을 alTop이나 alLeft 등으로 설정하면 컴포넌트의 위치가 항상 폼의 위쪽면이나 왼쪽면에 위치한다. 메모 컴포넌트의 Align 속성을 alClient로 설정해 두면 폼의 크기가 변해도 메모 컴포넌트가 항상 폼 전체 영역을 다 차지한다. 이 속성에 대해서는 7장에서 좀 더 자세하게 다룰 것이다.

메모의 속성만 변경해 주면 벌써 프로젝트가 완료되었다. 이 프로젝트를 저장하고 실행하면 간단한 에디터가 그야말로 간단하게 만들어진다. 실행중의 모습은 다음과 같다.




이 에디터는 문자열 입력은 물론 커서키로 위치를 이동할 수 있으며 BS, Del 키로 편집을 할 수 있으며 Shift+커서이동키로 블럭을 싼 후 Shift+Ins, Shift+Del, Ctrl+Ins 키로 블럭 조작까지도 수행할 수 있다.

이런 고성능의 에디터를 이렇게 짧은 시간에, 더구나 코드 한 줄 작성하지 않고 간단하게 만들 수 있는 이유는 메모 컴포넌트가 에디터를 만드는 데 필요한 모든 기능을 내부적으로 가지고 있기 때문이다. 에디터에 필요한 모든 기능이 미리 프로그래밍되어 메모 컴포넌트에 들어 있으므로 우리는 메모 컴포넌트를

폼에 배치하기만 하면 된다.

여기서 만들어 본 에디터는 정말 기본적인 기능만 가지고 있지만 앞으로 계속 만들어 더욱 더 기능을 첨가시켜 나갈 것이다. 툴 바를 만들고 메뉴를 만들고 클립보드에 복사하는 기능을 넣고 디스크의 파일을 읽어오거나 저장할 수 있도록 할 계획이다. 현재 단계에서는 여기까지밖에 만들 수가 없다.


① 메모:여러 줄의 문장을 입력하거나 편집하는 컴포넌트
② 메모의 Lines 속성으로 각 문자열을 읽을 수 있다.
③ WordWrap:자동 개행 속성

# 4-6 개발 툴

이제 여러분들은 델파이가 무엇인지를 알고 간단한 프로그램 정도는 짤 수 있게 되었다. 앞으로 좀 더 복잡한 문법과 여러 가지 컴포넌트들에 관해 배우게 되겠지만 여기서는 델파이 그 자체에 관해 조금만 더 연구해 보기로 하자. 알아두면 더 편하고 빠르게 작업할 수 있는 방법들이다.

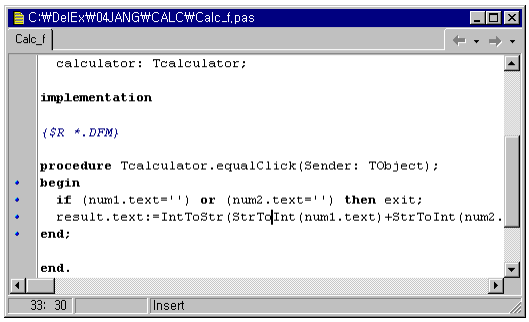
## 가. 코드 에디터

델파이의 코드 에디터는 얼핏 보기에는 일반적인 에디터와 비슷하며 사용법도 그리 어렵지 않아 쉽게 친근해질 수 있다. 하지만 코드 에디터를 좀 더 연구해 보면 작업에 도움을 주는 여러 가지 편리한 기능들이 많이 있다는 것을 발견할 수 있을 것이다.

코드 에디터의 동작 방법에는 Default, IDE, Brief, Epsilon 4가지 종류가 있으며 각 동작 방법에 따라 단축키나 블럭 지정 방법, 커서 이동 방법 등이 다르다. 각 방식은 Options 메뉴를 통해 바꿀 수 있으며 디폴트로 설정된 동작 방식은 Default 방식이다. 나머지 방법은 잘 쓰지도 않지만 혹시 관심있는 사람은 도움말을 참고하도록 하고 여기서는 Default 방식을 중심으로 설명을 하도록 한다.

### ■ 거터(Gutter)

코드 에디터의 왼쪽에 보면 회색의 굵은 띠가 있는데 이 부분을 거터라고 한다. 단순한 장식은 아니며 몇 가지 기능이 있다. 프로젝트를 컴파일해 보면 이 부분에 조그만 점들이 나타나는데 이는 이 줄의 코드가 실제로 컴파일되었다는 것을 나타낸다. 주석, 선언부 등에는 점이 당연히 나타나지 않으며 최적화에 의해 제외된 부분에도 점이 나타나지 않아 이 부분은 컴파일되지 않았음을 알려준다.




또한 북마크, 중단점 표시에도 거터가 사용되는데 관련 부분에서 거터의 사용법에 대해 알아볼 것이다.

## ■ 신택스 하이라이팅(syntax highlighting)

이 기능은 소스 코드 중 각 단어의 의미를 파악하여 의미별로 색상이나 글꼴 모양을 다르게 하여 소스를 쉽게 읽을 수 있도록 해준다. 예약어는 굵은 문자로, 주석은 파란색의 기울임 문자로, 나머지는 보통 문자로 출력해 주며 옵션을 변경하면 자기가 원하는 색상을 마음대로 바꿀 수 있다. 색깔 좀 예쁘게 출력해 주는 정도가 뭐가 그리 대단하냐고 할 지도 모르겠지만 이 기능이 가능하려면 편집과 동시에 편집된 내용을 문법적으로 해석해야 하므로 쉬운 기술은 아니다. 소스의 각 부분에 다른 글꼴을 적용함으로써 좀 더 소스를 읽기 쉽도록 해주며 구문 에러를 미연에 방지해 준다.

## ■ 다단계 Undo

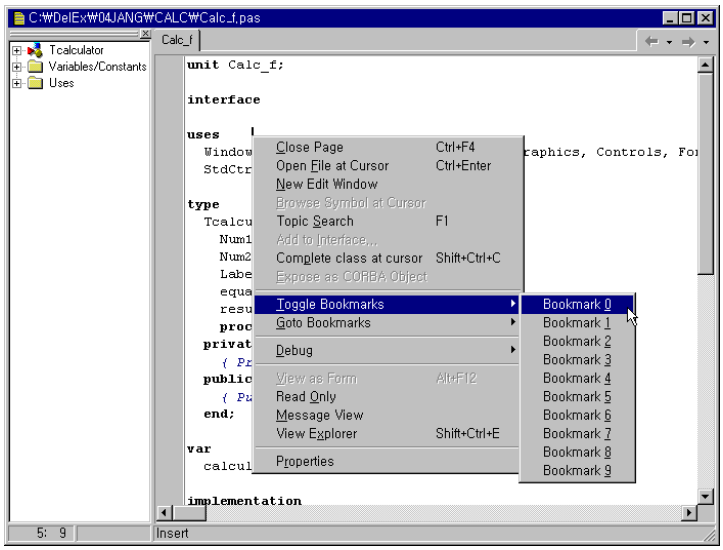
Undo란 편집 취소 기능을 말한다. 코드 에디터는 사용자가 편집한 모든 내용을 순서대로 기억해 두기 때문에 실수했을 경우 Ctrl-Z를 눌러 언제든지 실수를 취소할 수 있다. 여러 번 실수를 해도 Ctrl-Z만 계속 눌러주면 얼마든지 편집 사항을 취소할 수 있으나 단 파일을 저장한 후에는 Undo할 수 없다.

## ■ 북마크 기능

북마크란 일종의 책갈피와도 같으며 소스중의 특정 위치를 기억한다. 길이가 긴 소스를 편집하다 보면 소스 여기저기를 번갈아 돌아다녀야 하며 정확한 위치를 찾기가 어렵지만 북마크를 설정해 놓으면 설정해 놓은 북마크로 정확하게 이동할 수 있다. 북마크를 설정하거나 북마크로 이동하는 명령은 코드 에디터의 스피드 메뉴에 있다. 코드 에디터에서 마우스의 오른쪽 버튼을 누르면 다음과 같은 스피드 메뉴가 나타난다.

**그림**

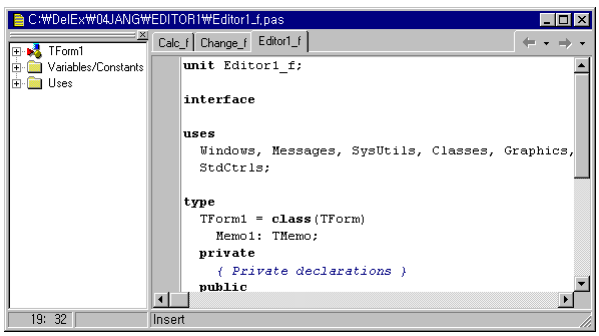
코드 에디터의 특정 위치를 기억하는 마커 기능




Toggle Bookmarks 명령으로 마크를 설정해 놓으면 소스중에 표시가 좌측 거터에 출력되며 언제든지 Goto Bookmarks명령을 사용하여 그 위치로 이동할 수 있다. 북마크는 긴 소스를 편집하던 중에 빨리 빨리 이동하기 위한 장치이므로 마우스를 사용하는 것보다 키보드를 사용하는 것이 훨씬 더 편리하다. 북마크 지정에는 Ctrl-K와 숫자 하나를 누르고, 북마크로 이동시는 Ctrl-Q와 숫자 하나 또는 간단하게 Ctrl키와 숫자 하나를 누른다.

## ■ 여러 개의 소스 편집 가능

코드 에디터는 하나밖에 없지만 위쪽의 페이지 탭을 사용하면 여러 개의 소스 파일을 동시에 열어두고 편집할 수 있다. File/Open 명령으로 새로운 소스 파일을 열면 코드 에디터에 새로운 탭이 생긴다. 다음은 네 개의 소스 파일을 한꺼번에 열어 놓고 편집하는 모양이다.



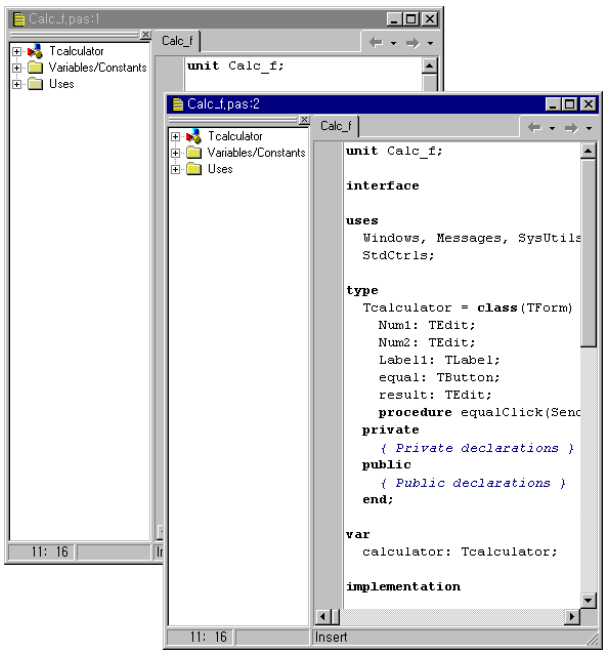
마우스로 탭을 눌러 소스간 전환을 하지만 편집중에는 마우스보다는 키보드를 사용하는 것이 더 편리하다. 페이지 전환 단축키인 Ctrl-Tab을 잘 알아두도록 하자.

## ■ 여러 개의 코드 에디터

코드 에디터가 아무리 한꺼번에 여러 개의 파일을 동시에 편집할 수 있다고는 하지만 탭 방식을 사용하므로 두 개의 소스를 한꺼번에 살펴볼 수는 없다. 두 소스를 상호 비교해가며 편집해야 할 때는 하나의 코드 에디터로는 문제를 해결할 수 없으며 두 개 이상의 코드 에디터를 사용해야 한다. 코드 에디터를 추가로 열려면 View/New Edit Window 항목을 선택한다.

**그림**

두 개의 코드 에디터를 열어 놓고 작업하는 모습



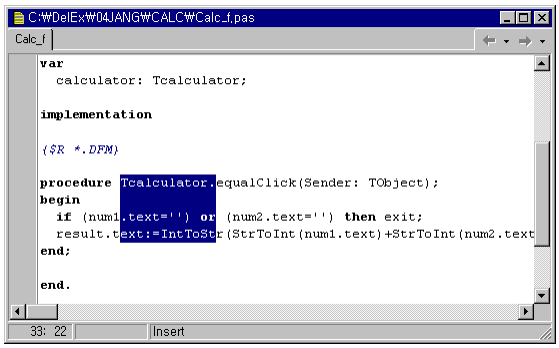

여러 개의 코드 에디터를 열어 놓고 작업하면 소스간의 비교나 이동도 자유로우므로 작업 효율에도 상당한 기여를 한다. 단 코드 에디터를 두 개 정도 열어 놓고 사용하려면 고해상도의 모니터가 필요할 것이다.

## ■ 블럭 선택하기

코드를 작성하다 보면 복사해서 붙이는 작업을 많이 하게 된다. 이때 복사의 대상이 되는 문장을 먼저 블럭으로 선택해야 하는데 마우스로 선택하고자 하는 부분을 드래그하기만 하면 블럭으로 선택된다. 또는 Shift키를 누른채로 커서를

이동하면 고무줄처럼 블럭 영역이 늘어나면서 선택 영역이 계속 확장된다. 이런 선택 방법은 윈도우즈의 표준이므로 더 이상 설명할 필요도 없다. 만약 다음과 같이 선택하고 싶으면 어떻게 할까?

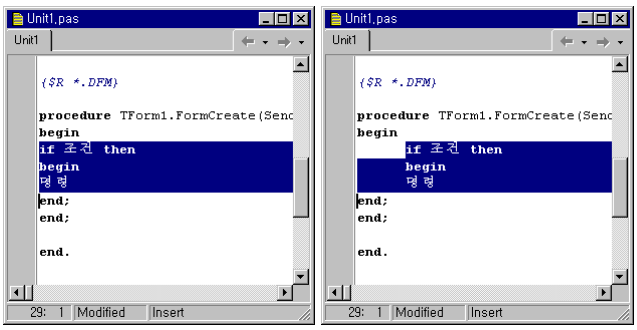
그림
구역으로 선택한 모양




연속된 부분이 아니라 코드의 일부분만 여러 행에 걸쳐 선택하는 방법인데 통상의 방법으로는 이런 선택을 할 수 없지만 Shift+Alt+커서이동키를 사용하면 구역 선택이 가능하다. 구역으로 선택한 영역을 복사한 후 붙이면 구역 복사도 물론 된다.

## ■ 들여쓰기, 내어쓰기

코드는 적당히 들여쓰는 것이 읽기에 좋다. 특히 if, for, begin 등 새로운 단락이 시작될 때는 으레 2칸~4칸 정도 들여써서 한눈에 코드의 분절을 파악할 수 있도록 한다. 그런데 일단 작성한 코드의 들여쓰기를 하려면 일일이 행 앞쪽에 스페이스를 넣어 주어야 하는데 이때는 들여쓰고 싶은 부분을 블럭으로 선택한 후 Ctrl+Shift+I를 누르면 된다. 한번 누를 때마다 두 칸씩 들여쓰기가 된다.




반대로 내어쓰기를 하고 싶으면 블럭을 선택한 후 Ctrl+Shift+U를 누르면 된다. 이 기능을 잘 이용하면 읽기 쉽고 보기 좋은 소스를 유지할 수 있다.

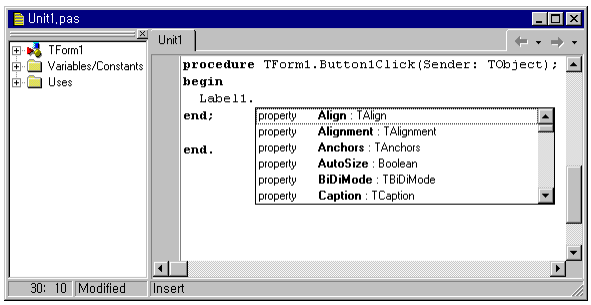
# 나. 코드 인사이트

코드 인사이트(Code Insight) 기능은 코드 작성에 실질적인 도움을 주는 아주 편리한 기능이다. 이 기능들의 도움을 받으면 일일이 타이핑을 하거나 함수의 원형을 외우거나 도움말을 뒤적거릴 필요가 상당히 줄어든다. 추가된 기능들을 테스트해 보기 위해 버튼 두 개, 레이블 하나를 배치해 보자.

## ■ 코드 컴플릿

이 기능은 컴포넌트의 속성, 메소드, 이벤트를 신속하게 선택하도록 해준다. 버튼을 더블클릭하여 OnClick 이벤트 핸들러를 만들고 이 핸들러에서 레이블의 속성을 변경하도록 해보자. Label1.까지만 입력하면 다음과 같이 레이블의 속성들이 팝업 메뉴로 나타나며 이 때 마우스 커서나 아래/위 커서 이동키로 속성을 선택할 수 있다.

그림
컴포넌트의 속성, 메소드가 출력된다.



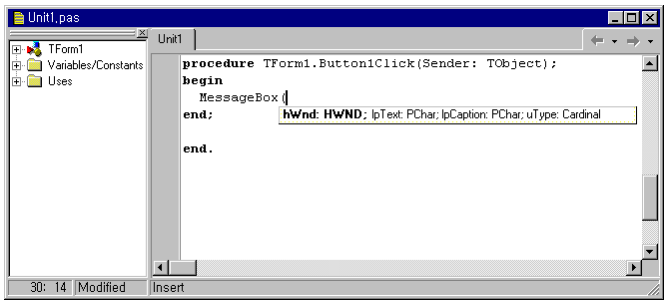

목록에서 원하는 속성을 선택한 후 Enter 키를 누르면 선택한 속성이 코드 에디트에 입력된다. 일일이 타이핑을 하지 않아도 되며 속성 이름을 완전히 외우지 않아도 되며 오타의 가능성도 훨씬 낮아진다.

## ■ 코드 파라미터

델파이에서 호출할 수 있는 함수, 메소드는 1000 여 개가 넘는데 이 모든 함수의 인수들을 모두 외우고 다니는 사람은 없다. 자주 쓰는 몇몇 함수에 대해서만 인수의 개수나 형태를 외워두고(또는 자연스럽게 외워지고) 나머지는 필요할 때마다 도움말을 참조해가며 사용한다. 코드 파라미터 기능은 함수명과 여는 괄호까지만 입력하면 인수의 개수, 타입등을 팝업 윈도우로 보여주는 기능이다. MessageBox(까지만 입력한 후 잠시 기다려 보면 다음과 같이 이 함수의 인수 리스트를 보여줄 것이다.

그림

함수의 인수에 대한 정보를 보여준다.



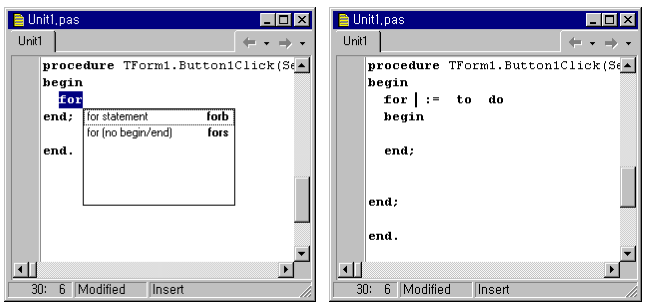

뿐만 아니라 인수를 입력시킴에 따라 다음에 입력해야 할 인수에 대한 정보를 굵은 글꼴로 보여준다. 물론 인수나 리턴값등에 대한 자세한 정보들은 도움말을 읽어보아야겠지만 사실 이정도의 힌트만 해도 대단히 큰 도움이 된다.

## ■ 코드 템플릿

코드 템플릿(Code Template)이란 기본 문법의 틀을 빨리 만들어 주는 기능이다. 예를 들어 for 까지만 입력하고 Ctrl+J 를 누르면 다음과 같이 사용하고자 하는 문장의 종류를 팝업 메뉴를 통해 보여주며 여기서 원하는 문장을 선택하면 기본틀을 소스에 만들어 준다.

그림

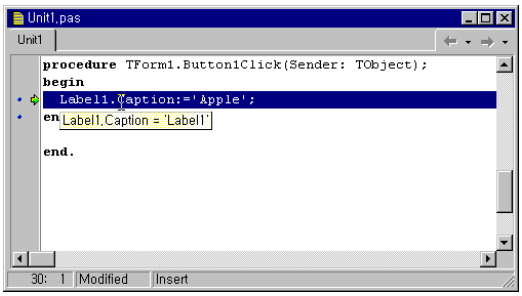
자주 쓰는 코드의 뼈대를 만들어 준다.




여러 가지 언어를 번갈아 사용하는 사람들의 경우 가끔 기본적인 문장 형식을 혼돈하는 경우가 있는데 이 기능을 사용하면 정확한 문법틀을 만든 후 필요한 부분만 추가하는 식으로 코드를 작성할 수 있다.

## ■ 툴팁형 값 평가

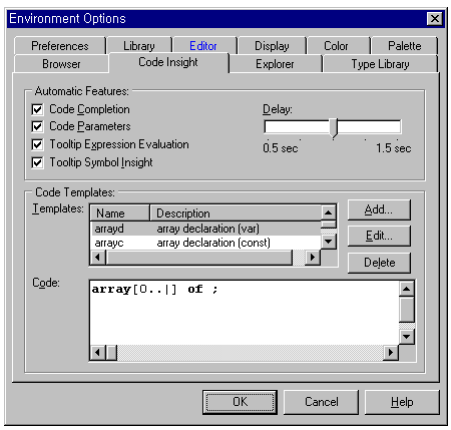
디버깅중에 확인하고자 하는 변수값 위치에 마우스 커스를 갖다대면 툴팁 형태로 해당 변수의 현재값을 보여주는 기능이다. 이 기능에 대해서는 다음에 디버깅을 배울 때 다시 알아 보기로 하자.




## ■ 코드 인사이트 옵션 설정

코드 작성에 도움을 주는 이런 기능들은 무척 편리하고 신선한 것이지만 능숙한 숙련자들에게는 때로는 귀찮을 수도 있다. 그래서 Tools/Environment/Code Insight 대화상자에서 각 기능을 선택적으로 사용할 수 있도록 되어 있다.

그림
코드 인사이트 옵션

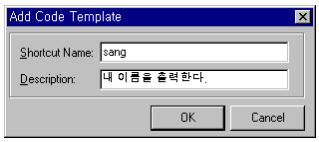



각 기능의 사용 여부와 대기 시간 등을 선택할 수 있으며 아래쪽에서는 코드 템플릿에 새로운 템플릿을 정의할 수 있다. 디폴트로 정의된 몇 가지 템플릿을 보면 다음과 같다.

| 이름 | 정의 |
|---|---|
| arrayd | array[0..|] of ; |
| arrayc | array[0..|] of = (); |
| cases | case | of<br>: ;<br>: ; |

| | |
|---|---|
| | end; |
| classf | T\| = class(T) |
| | private |
| | protected |
| | public |
| | published |
| | end; |
| fors | for \| := to do |
| ifs | if \| then |
| withs | if \| then |

템플릿 정의문에 있는 |문자는 템플릿 출력후에 커서가 위치할 곳을 가리킨다. 예를 들어 ifs 를 입력한 후 Ctrl+J 를 누르면 if then 이 출력되며 커서는 if 다음에 배치되어 조건식을 입력할 수 있도록 해 준다. 새로운 템플릿을 추가하려면 Add 버튼을 누른다. 다음과 같은 대화상자를 보여 주는데 이 대화상자에 템플릿에 대한 간단한 설명을 입력한다.




템플릿의 이름을 입력한 후 OK 버튼을 누르고 Code 란에 다음과 같이 입력해 보자.

```
// Programmer:Kim Sang Hyung
// Tool:Delphi
// Date:|
```

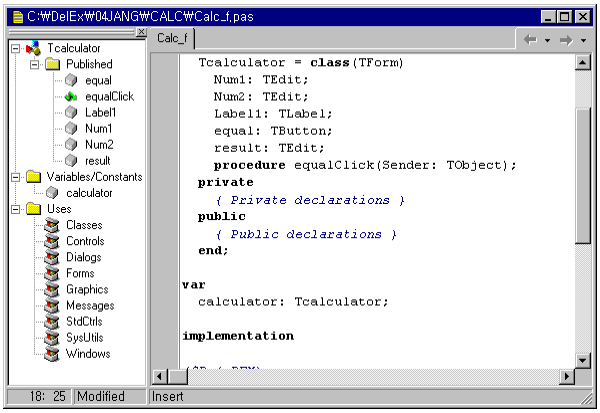
이제 코드 에디터에서 sang 만 입력하고 Ctrl+J 를 누르면 이 문자열이 에디터에 나타나며 Date:다음에 커서가 위치하게 될 것이다. 자주 사용하는 코드는 이런 식으로 템플릿을 정의해 두면 반복적인 키입력을 하지 않아도 될 것이다.

# 다. 코드 탐색기

에디터의 왼쪽을 보면 조그마한 윈도우가 항상 붙어 있는데 이 윈도우를 코드 탐색기(Code Explorer)라고 한다. 코드 탐색기에는 현재 열려 있는 유닛에 정의된 모든 변수, 타입, 속성, 메소드 등이 계층 구조로 나열되어 있다. 앞에서 만든 Calc.dpr 프로젝트를 열어 코드 탐색기를 보도록 하자.

**그림**

코드 에디터 옆에 나타나는 코드 탐색기




코드 탐색기의 가지를 전부 확장해 보면 이 프로젝트에 배치된 컴포넌트 목록과 이벤트 핸들러, 변수 등의 목록이 나타나며 다른 유닛의 목록도 나타난다. 이 목록에서 보고자 하는 명칭을 더블클릭하면 해당 명칭이 정의된 부분으로 즉각 이동한다. 실습삼아 equalClick 이벤트 핸들러를 더블클릭해 보면 이 핸들러가 정의된 곳으로 이동할 것이다. 코드의 길이가 무척 길 때 코드 탐색기를 사용하면 원하는 명칭이 어디에 정의되어 있는지를 빨리 빨리 찾을 수 있게 된다.

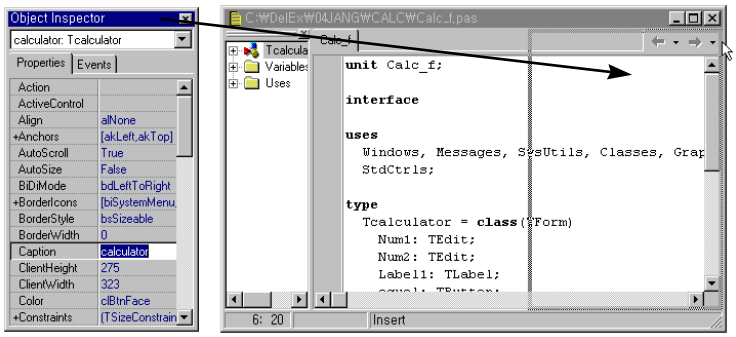
또한 코드 탐색기의 팝업 메뉴를 사용하면 새로운 변수나 컴포넌트 등을 배치할 수 있고 기존의 명칭 이름을 변경할 수도 있다. 이 외에 코드 탐색기는 여러 가지 부가 기능을 가지고 있는데 관련 부분에서 다시 살펴볼 것이다.

# 라. 도킹

델파이 개발 환경은 여러 개의 분리된 윈도우들로 구성되어 있다. 너무 많은 윈도우들이 한꺼번에 열리다 보니 굉장히 혼란스럽고 작업 효율도 떨어지는데 버전 4 부터는 개별 윈도우가 도킹이 되므로 이런 혼란이 많이 감소되었다. 도킹의 예는 코드 에디터를 보면 된다. 코드 에디터 왼쪽에 코드 탐색기가 항상 붙어

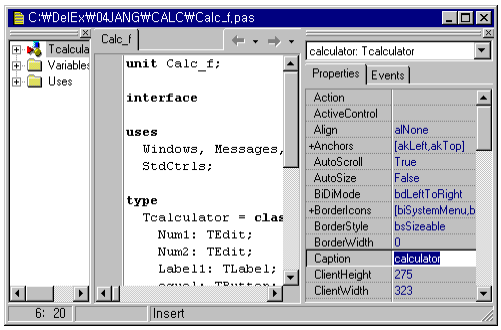
있는데 이 상태가 바로 도킹이 되어 있는 상태다. 코드 탐색기뿐만 아니라 대부분의 윈도우들이 도킹이 된다. 도킹시키는 방법은 드래그해서 갖다 붙이기만 하면 되므로 비교적 간단하고 직관적이다. 예를 들어 오브젝트 인스펙터를 코드 윈도우에 붙이려면 다음과 같이 하면 된다.

**그림**
오브젝트 인스펙터를 코드 에디터에 도킹시키기




오브젝트 인스펙터의 타이틀 바를 드래그해서 코드 에디터의 가장자리에 떨어뜨리면 된다. 이때 드래그되는 회색 음영이 코드 에디터에 척 달라붙는 느낌이 들텐데 그 때 마우스 버튼을 놓으면 된다. 오브젝트 인스펙터가 도킹된 후의 모양은 다음과 같다.

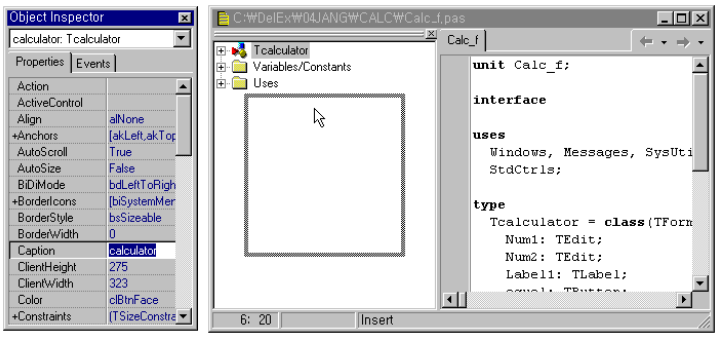
**그림**
도킹된 후의 모양



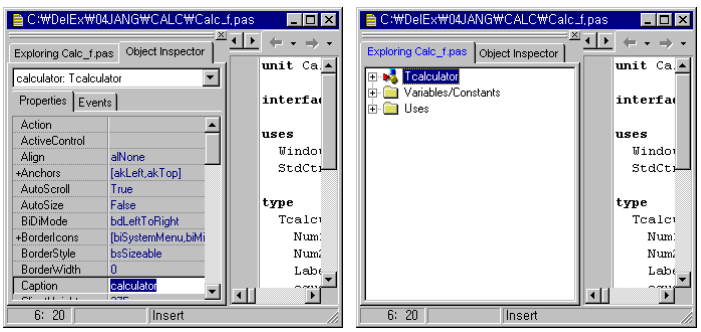

이렇게 도킹을 시켜 놓으면 코드 에디터와 오브젝트 인스펙터가 항상 붙어 다니며 상호의 영역을 침범하지 않게 되므로 훨씬 더 편해진다. 분리할 때는 반대로 도킹된 오브젝트 인스펙터를 드래그해서 코드 에디터 밖으로 끌어내 버리면 된다. 이 방법대로 도킹된 윈도우를 분리하면 코드 탐색기도 코드 에디터에서 분리할 수 있다. 만약 도킹시키지 않고 이동만 하고자 할 경우는 Ctrl 키를 누른 채로 드래그하면 된다.

오브젝트 인스펙터, 코드 탐색기뿐만 아니라 여러 가지 윈도우들이 도킹이 되는데 이런 식으로 코드 에디터의 영역을 사이좋게 나누어 쓰다 보면 화면 영역이

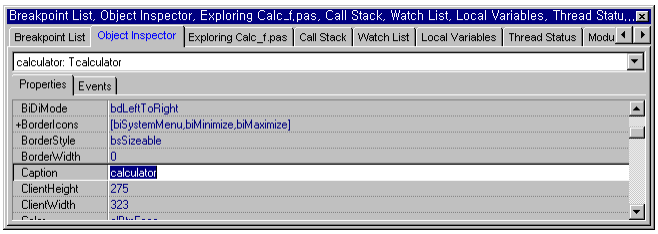
부족할 것이다. 이 때는 여러 개의 윈도우들을 한 영역에 같이 도킹시킬 수가 있다. 다음은 오브젝트 인스펙터를 코드 탐색기 영역에 같이 도킹시키는 예이다.




오브젝트 인스펙터를 코드 탐색기의 중앙으로 드래그하면 회색 음영이 코드 탐색기의 중앙에 나타나게 되는데 이때 마우스 버튼을 놓으면 두 윈도우가 같은 영역에 도킹된 것이다. 두 개 이상의 윈도우가 한 영역에 도킹되면 위쪽에 탭이 나타나며 탭으로 작업 윈도우를 선택한다.



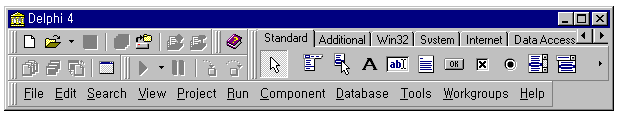

분리할 때는 탭을 드래그해서 밖으로 끄집어 내면 된다. 다음은 필자가 장난으로 10 개의 윈도우를 한 윈도우에 모두 도킹시켜 본 것이다. 하도 많은 윈도우가 붙어 있으니까 탭에도 스크롤 바가 나타난다.



도킹의 대상이 되는 윈도우는 디버깅 윈도우 전체와 오브젝트 인스펙터, 코드 탐색기, 프로젝트 관리자 등이다. 코드 에디터는 다른 윈도우의 도킹을 받아들일 수는 있지만 다른 윈도우에 도킹되지는 못한다. 정렬 팔레트나 컴포넌트 리스트 등의 윈도우는 도킹할 수 없다. 일반적으로 윈도우를 드래그할 때 회색 음영이 드래그되는 윈도우는 도킹이되는 윈도우이며 윈도우 화면이 통째로 드래그되는 윈도우는 도킹이 안되는 윈도우라고 판단하면 된다.

# 마. 메인 메뉴

메인 메뉴에는 델파이의 모든 기능이 총체적으로 모여 있다. 모두 10개의 메뉴 항목이 메뉴 바에 나열되어 있으며 윈도우즈의 표준적인 메뉴와 기능이나 사용 방법이 동일하다. 마우스로는 클릭하면 되고 키보드로 메뉴를 호출할 때는 언제든지 Alt 키를 누르기만 하면 된다. 다만 이때까지 사용해 오던 표준적인 메뉴와는 달리 메뉴가 반드시 타이틀 바 아래에만 있어야 할 필요는 없으며 툴바보다 더 아래쪽에 위치시켜 둘 수도 있다는 점과 메뉴 항목 옆에 비트맵이 나타난다는 점이 조금 다르다.

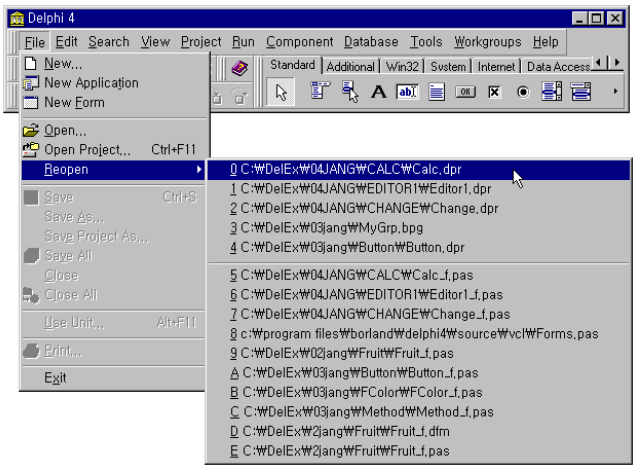



개별적인 메뉴 항목에 대해서는 직관적으로 알 수 있을 것이라 생각하며 설명을 생략하기로 하되 초보자들이 흔히 지나치기 쉬운 유용한 팁 몇 가지를 소개하기로 한다.

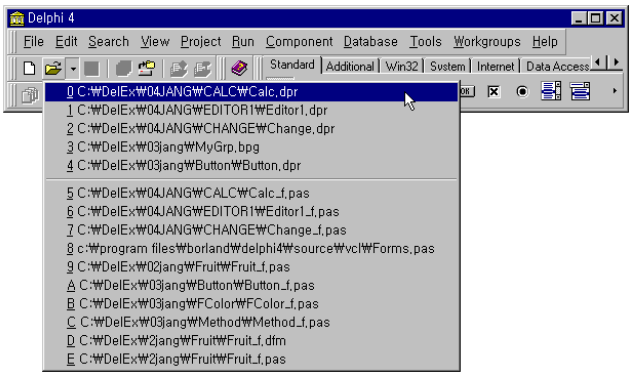
## ■ 프로젝트 다시 열기

File 메뉴를 보면 Reopen이라는 메뉴 항목이 있다. 이 항목의 세부 메뉴에는 최근에 열었던 프로젝트의 목록과 소스 파일 목록이 나타나며 이 목록에서 프로젝트를 선택하면 즉각 그 프로젝트를 열어 준다.

그림
최근에 열었던 프로젝트의 목록



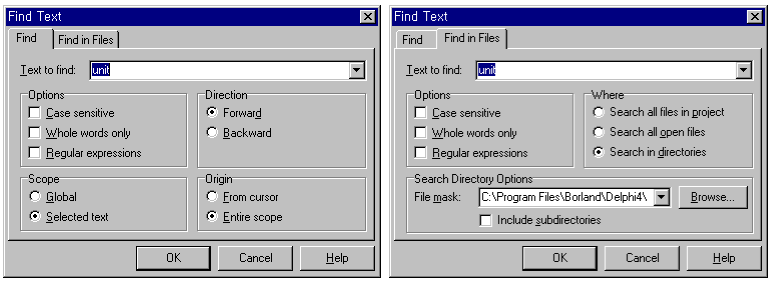

툴바에도 똑같은 기능을 하는 버튼이 있는데  버튼 옆의 아래쪽 화살표 버튼을 누르면 이 메뉴가 똑같이 나타난다.




File/Open을 사용해서 경로를 일일이 찾아가는 방법보다 훨씬 더 빠르고 편하게 프로젝트를 열 수 있다. 특히 여러 개의 프로젝트를 번갈아 가며 동시에 개발하고 있을 때 무척 유용한 기능이다.

## ■ 찾기, 바꾸기

길이가 긴 소스 파일을 편집할 때는 찾기/바꾸기 기능이 필수적이다. 찾기 대화상자는 Search/Find 메뉴를 누르면 나타난다. 두 개의 페이지로 구성되어 있는데 Find 는 현재 열려진 소스내에서 문자열을 찾는 것이고 Find in Files 는 프로젝트내의 모든 소스나 특정 디렉토리의 모든 소스에 대해 검색을 하는 것이다.




검색된 결과는 코드 에디터의 아래쪽에 나타난다. 찾기 대화상자의 아래쪽에는 대소문자 구분, 부분 문자열, 검색 방향 등에 대한 옵션들이 있다.

## ■ 다시 컴파일하기

델파이가 프로젝트를 컴파일할 때는 종속성 점검을 통해 최대한 컴파일 속도를 높인다. 종속성 점검이란 소스 파일의 수정 여부에 따라 다시 컴파일할 필요가 있는 소스만 컴파일하여 실행 파일을 만들어내는 기법이다. 예를 들어 A, B, C 세 가지 소스로 구성된 프로젝트가 있고 C 소스의 일부분을 수정한 후 다시 컴파일한다고 해보자. 이때 A, B는 이미 컴파일되어 있으므로 더 이상 컴파일할 필요가 없고 C만 컴파일한 후 링크하면 실행 파일을 만들 수 있다. 쉽게 말하자면 꼭 필요한 경우만 컴파일을 하도록 하여 컴파일 속도를 향상시키는 기법이다.

그러나 이런 종속성 점검 컴파일 방식이 가끔 문제가 되는 경우가 있다. 구체적인 예를 들기는 어렵지만 가끔 컴파일해야 할 소스를 컴파일하지 않아 제대로 동작하지 않는 예를 볼 수 있는데 이때는 종속성 점검을 하지 않고 무조건 컴파일하도록 해야 하며 이때 사용하는 명령이 Project/Build All Projects이다. 이 명령은 프로젝트에 속한 모든 유닛을 강제로 컴파일하여 완전한 실행 파일을 만들어낸다. 분명히 제대로 했는데 프로그램이 이상하게 돌아간다면 이 메뉴를 한 번 사용해 보기 바란다.

## ■ 문법 검사

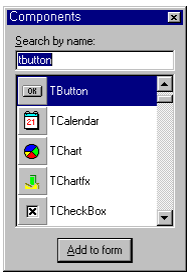
프로그램을 짜다 보면 항상 에러가 나기 마련이다. F9를 눌러 컴파일해 보면 에러 메시지가 출력되며 이 메시지 내용대로 에러를 수정해 주는 것이 통상적인 절차다. 하지만 에러가 발생할 것을 뻔히 알고 있다면 컴파일해 보는 것보다 더 좋은 방법이 있다. Project/Syntax Check 항목을 선택하면 컴파일은 하지 않고 소스의 문법이 어디가 잘못되었는지만 찾아주므로 에러를 찾는 속도가 훨씬 더 빠르다. 단축키 Alt+P, S를 외워두었다가 문법 체크를 먼저 하여 에러를 수정한 후 컴파일하는 것도 권장할만한 일이다.

# 바. 키보드 지원

델파이 작업의 대부분은 마우스로 수행한다. 그러나 키보드를 전혀 무시하지는 않는다. 마우스로 할 수 있는 작업의 대부분은 키보드로도 할 수 있도록 되어 있다. 물론 마우스가 동작하지 않는 극한 상황에서나 사용되겠지만 때로는 유용하게 사용되기도 하므로 잠시 살펴나 보도록 하자. 키보드로 메뉴를 선택하는 방법은 Alt 키와 메뉴 이름의 밑줄친 부분의 문자를 누른다. 예를 들어 View 메뉴는 Alt+V를 눌러 선택한다.

## ■ 컴포넌트 리스트

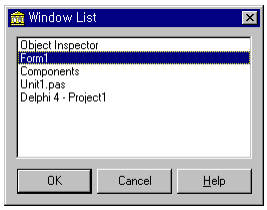
키보드로 컴포넌트를 폼에 배치하고자 할 때 사용한다. View/Component List 항목을 선택하면 다음과 같은 컴포넌트 목록을 보여준다.




위쪽의 에디트 박스에 직접 컴포넌트의 이름을 입력하든가 리스트 박스에서 컴포넌트를 선택한 후 Enter를 누르면 선택한 컴포넌트가 폼의 중앙에 배치된다. 폼의 위치나 크기, 속성을 변경할 때는 F11키를 눌러 오브젝트 인스펙터로 이동한 후 속성을 변경하면 된다.

## ■ 윈도우 리스트

델파이는 여러 개의 윈도우로 구성되어 있다. 마우스를 사용하면 각 윈도우를 전환하는 것이 전혀 문제되지 않지만 키보드를 사용할 경우는 윈도우 리스트를 사용한다. View/Window List 항목을 선택하면 현재 열려 있는 윈도우의 목록이 나타나며 이 목록에서 해당 윈도우를 선택하면 된다.




### ■ 폼/유닛 토글

폼과 유닛 사이를 교체할 때는 View/Toggle Form/Unit 항목을 선택하거나 아니면 단축키인 F12키를 누른다. 오브젝트 인스펙터와 폼 사이를 교체할 때는 F11키를 사용한다.

이 정도만 알면 키보드로도 델파이 프로그램을 그럭 저럭 짤 수 있을 것이다. 물론 마우스가 건재하다면 전혀 도움이 안되는 내용들이므로 깡그리 잊어 먹어도 좋다.

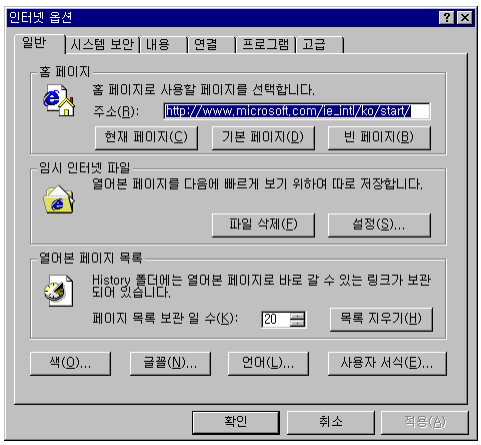
## 사. 탭 순서 지정 대화상자

입력용 대화상자에서는 탭 순서가 중요한 의미를 가진다.

실행중에 폼에 놓여진 컴포넌트 사이로 이동하기 위해서는 Tab키를 사용하며 각 컴포넌트는 Tab키를 누를 때 몇 번째로 포커스를 받을 것인가에 대한 순번이 있다. 윈도우즈의 모든 대화상자를 자세히 관찰해 보면 대화상자의 컨트롤 순번이 아주 잘 정돈되어 있음을 확인할 수 있을 것이다. 다음은 인터넷 익스플로러의 옵션 대화상자이다.

그림

탭 순서가 잘 정돈된 대화상자



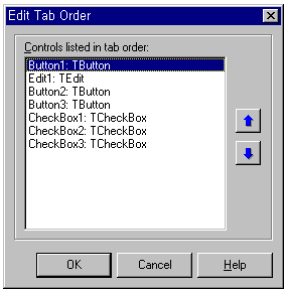

탭 키를 계속 눌러보면 컨트롤의 순번이 왼쪽에서 오른쪽으로 위에서 아래로

매겨져 있기 때문에 키보드로 컨트롤간의 이동이 쉽도록 되어 있다. 너무 당연한 것처럼 생각될지 모르겠지만 일부러 순번을 맞추어주지 않으면 포커스가 개구리 뛰듯 이리 뛰었다 저리 뛰었다 정신 사나운 꼴이 발생하게 된다.

컴포넌트의 순번은 TabOrder 속성으로 지정하며 폼에 배치된 순서대로 순번이 매겨진다. 순번을 변경하고자 할 경우는 개별 컴포넌트의 TabOrder 속성을 직접 변경해도 상관없지만 다음과 같은 탭 순서 지정 대화상자를 사용하면 컴포넌트의 목록을 보며 순번을 편집할 수 있다. 이 대화상자를 불러 내려면 Edit/Tab Order 메뉴 항목을 선택한다.

그림

탭 순서를 조정할 수 있는 탭 순서 지정 대화상자

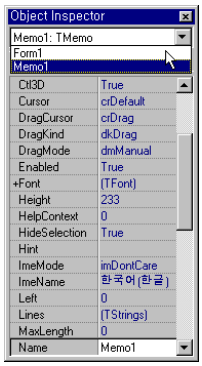



리스트 박스에는 폼에 놓여진 모든 컴포넌트의 목록이 Tab 순서대로 나열되어 있다. Tab 순서를 바꾸려면 컴포넌트를 선택한 후 우측의 아래, 위 버튼을 사용하여 컴포넌트를 이동시키거나 아니면 목록에서 컴포넌트를 직접 드래그하도록 한다.

# 아. 객체 선택기

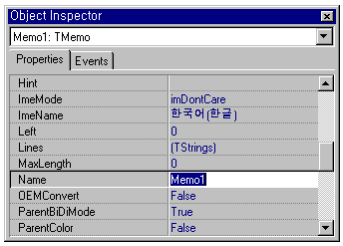
오브젝트 인스펙터는 컴포넌트, 속성, 이벤트만 알면 쉽게 사용할 수 있는 툴이다. 하지만 초보자의 경우 객체 선택기를 몰라 처음에 조금 당황하는 경우가 있다. 어떤 경우냐면 앞에서 만든 에디터 예제와 같은 경우다. 메모 컴포넌트가 폼 전체를 온통 뒤덮고 있기 때문에 마우스로는 폼을 선택할 수가 없으며 따라서 폼의 속성도 바꿀 수 없다. 이런 경우는 컴포넌트가 다른 컴포넌트에 의해 가려져 있는 경우에도 나타난다. 폼에서 컴포넌트를 클릭해서 선택할 수 없을 때 오브젝트 인스펙터의 상단에 있는 객체 선택기를 사용한다.

그림
객체 선택기를 사용하면 폼에 놓여진 어떤 컴포넌트도 선택할 수 있다.




객체 선택기는 오브젝트 인스펙터의 상단에 있으며 폼에 배치된 모든 컴포넌트의 목록이 나타나므로 가려져 있는 컴포넌트도 선택할 수 있고 또한 컴포넌트의 이름을 보고 선택할 수도 있다. 오브젝트 인스펙터는 크기 조절도 자유로우며 다음과 같이 넓게 쓸 수도 있고 경계 부근을 드래그하여 속성란과 속성값 란의 크기를 변경할 수도 있다.

그림
오브젝트 인스펙터는 필요에 따라 크기를 늘릴 수 있다.




# 자. 코드 삭제하기

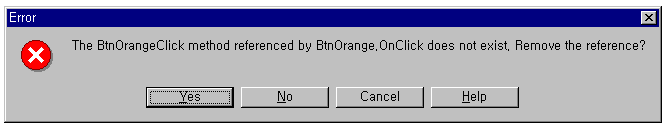
프로그램을 짜다보면 항상 코드를 작성하기만 하는 것이 아니라 때로는 작성한 코드를 삭제해야 할 경우도 있다. 한번 배치한 컴포넌트나 한번 작성한 코드를 삭제할 때는 오히려 작성하는 것보다 조금 더 까다로우므로 약간의 주의를 해야 한다. 우선 컴포넌트를 삭제할 경우를 보자. 다음은 Fruit 예제의 소스인 Fruit_f.pas 중 핵심 부분만을 간추린 것이다.

```
type
  TForm1 = class(TForm)
    BtnApple: TButton;
    Fruit: TLabel;
    BtnOrange: TButton;
    procedure BtnAppleClick(Sender: TObject);
```

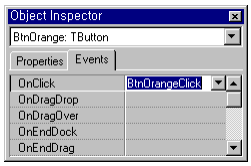
```
    procedure BtnOrangeClick(Sender: TObject);
  end;
procedure TForm1.BtnAppleClick(Sender: TObject);
begin
  Fruit.Caption:='Apple';
end;
procedure TForm1.BtnOrangeClick(Sender: TObject);
begin
  Fruit.Caption:='Orange';
end;
end.
```

만약 폼에서 BtnOrange 버튼을 삭제했다고 한다면 위 리스트에서 밑줄 그은 부분이 자동으로 같이 삭제된다. 델파이는 컴포넌트가 배치되면 코드에도 컴포넌트를 배치하고 삭제하면 코드의 컴포넌트를 같이 삭제해 주므로 컴포넌트를 배치하거나 삭제하는 일은 비교적 자유롭다. 그러나 위 리스트를 다시 자세히 보면 BtnOrange가 삭제된 상황에서 BtnOrange1Click 프로시저는 전혀 필요가 없음에도 불구하고 델파이는 이 코드를 삭제해 주지 않는다. 왜 그런가 하면 이 프로시저는 현재는 BtnOrange만 사용하는 코드이지만 다른 컴포넌트도 이 프로시저를 호출하거나 사용할 수 있기 때문이다. 그래서 델파이가 이 코드를 자동으로 삭제할 수 없으며 불필요한 코드라고 판단되는 코드는 사용자가 직접 삭제해 주어야 한다.

컴포넌트는 남겨둔 채 코드만 삭제할 때는 조금 더 복잡하다. 위 예에서 BtnOrange1Click 프로시저를 삭제해야 할 경우 굵은 이탤릭체로 된 부분을 모두 삭제해야 한다. 이벤트 핸들러 뿐만 아니라 폼의 타입 선언에 포함된 7행도 같이 삭제해 주어야 한다. 둘 중 하나만 삭제해서는 안된다. 프로시저를 삭제한 후 다시 컴파일시키면 다음과 같이 물어온다.




프로시저는 삭제되었지만 BtnOrange 컴포넌트의 OnClick 이벤트는 여전히 삭제된 프로시저를 가리키고 있기 때문이다. 그래서 델파이가 삭제된 프로시저를 완전히 없앨 것인가를 물어오는데 이 질문에 Yes라고 응답해 주면 무사히 코드를 삭제한 것이다. 아니면 아예 프로시저를 없앨 때 오브젝트 인스펙터에서 BtnOrange의 OnClick 이벤트까지 같이 지워버리면 이런 질문을 받지 않는다.




이벤트 핸들러를 삭제하는 또 다른 방법은 이벤트 핸들러 자체는 그대로 두고 코드 부분만 삭제하는 방법이다. 즉 begin~end 사이에 있는 프로시저의 본체 ***Fruit.Caption:='Orange';*** 코드만 삭제한다. 그리고 프로그램을 실행시키거나 저장하면 델파이는 코드가 없는 이벤트 핸들러를 알아서 삭제해 준다. 필자는 이 방법을 많이 애용하는 편이다. 저장이나 실행시키지 않고 삭제하려면 Alt+P,S를 눌러 문법 검사만 해 주어도 된다.

만약 이벤트 핸들러만 만들어 놓고 코드는 아직 작성하고 싶지 않다면 다음과 같이 본체에 세미콜론만 하나 찍어 놓으면 이 핸들러는 실행하거나 저장해도 지워지지 않는다.

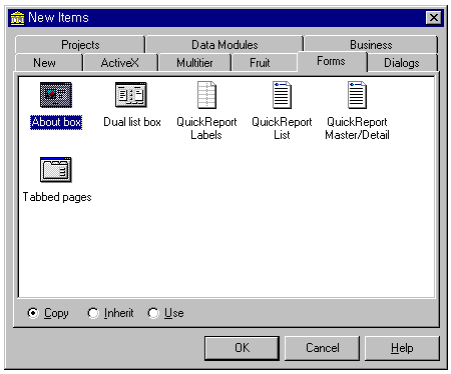
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ;
end;
```

이런 용도로 사용되는 세미콜론은 널 문장이라고 하며 컴파일해 봐야 아무 코드도 만들어내지 않는다. 하지만 문장이 있기는 있는 것이므로 델파이는 이 핸들러가 비어있지 않다고 판단하게 된다.

# 차. 오브젝트 창고

델파이는 사용하기 편리한 비주얼 툴이므로 폼이나 대화상자를 디자인하기 쉽다는 장점을 가진다. 그러나 이런 편리한 개발 환경에서도 매번 같은 폼을 만드는 것은 분명히 짜증나는 일이다. 그래서 델파이는 재사용성을 높이기 위해 오브젝트 창고라는 것을 제공하는데 오브젝트 창고(Object Repository)란 한번 작성한 요소(Item)를 디스크에 저장해 두었다가 다음에 다시 사용할 수 있도록 해주는 장치이다. 여기서 요소(Item)라는 말은 델파이 프로젝트에 사용되는 폼, 프로젝트, 유닛, 대화상자, 데이터 모듈 등을 말한다. 오브젝트 창고를 호출하려면 File/New 메뉴 항목을 선택한다.

그림
재사용 가능한 요소를 담고 있는 오브젝트 창고



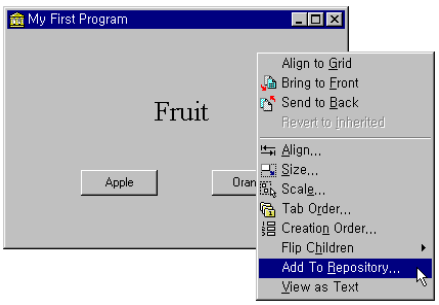

모두 아홉 개의 페이지로 구성되어 있다. 첫 번째 페이지인 New 페이지는 프로젝트에 추가할 수 있는 새로운 요소를 담고 있으며 나머지 페이지는 사용자가 창고에 등록해 놓은 재사용할 수 있는 요소들을 담고 있다.

## ■ 폼 재사용

재사용이 가장 빈번한 요소는 아무래도 폼이다. 잘 설계해 놓은 폼을 한번만 창고에 등록해 놓으면 다음부터는 비슷한 모양의 폼은 다시 만들지 않아도 창고에서 꺼내 오기만 하면 된다. 간단한 예로 폼을 창고에 등록해 두고 재사용하는 예를 들어 보자.

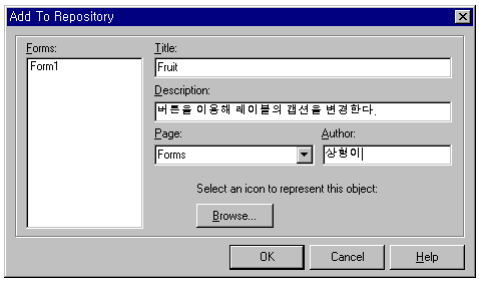
1 창고에 등록하고자 하는 폼을 일단 읽어와야 한다. 2 장에서 처음 만들었던 Fruit 예제의 Fruit_f.pas 폼을 실습 예제로 사용해 보자. 이 프로젝트를 불러온 후 폼의 스피드 메뉴에서 Add to Repository 항목을 선택한다.

그림
폼의 스피드 메뉴



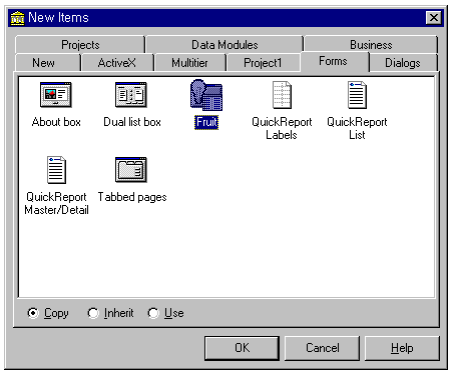

2 다음과 같이 등록할 폼에 관한 정보를 입력하는 대화상자를 보여 줄 것이다.

그림
폼 등록 대화상자




폼의 제목, 간단한 설명 및 등록할 페이지 이름을 지정하고 고유의 아이콘을 할당할 수도 있다. 대충 정보를 입력한 후 OK 버튼을 누르면 등록이 완료된다.

3 그럼 등록한 폼을 다시 한번 더 사용해 보도록 하자. File/New Application 메뉴를 선택해 새로운 프로젝트를 시작한다. 그리고 다시 File/New 를 선택해 오브젝트 창고를 연다.



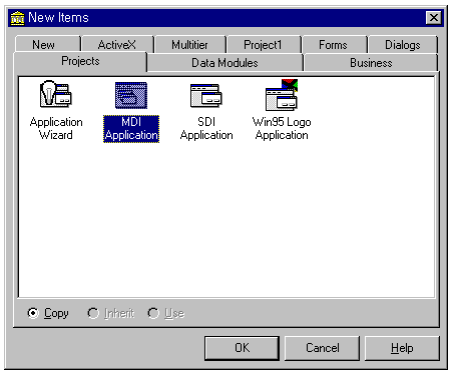

Forms 페이지를 보면 우리가 등록한 Fruit 폼이 위치해 있으며 이 폼을 선택하면 현재 프로젝트에 이 폼이 추가된다. 몇 번의 마우스 클릭만으로 아주 쉽게 폼을 추가하였다.

이 외에도 오브젝트 창고에는 델파이가 기본적으로 제공하는 몇 가지 실용적인 폼이 등록되어 있으므로 필요한 사람은 사용해 보기 바란다.

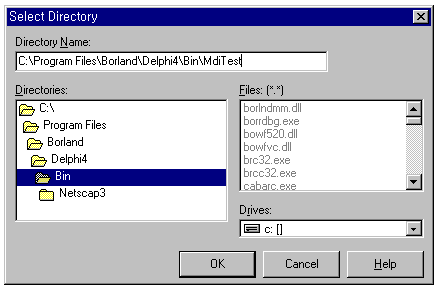
## ■ 프로젝트 재사용

개별적인 폼뿐만 아니라 프로젝트를 통째로 저장해 둘 수도 있다. 이 경우는 재사용을 위해 저장해 둔다기 보다는 비슷한 유형의 프로젝트를 좀 더 빨리 작성하기 위한 모델로서 사용한다고 볼 수 있다. 예를 들어 여러 개의 차일드 윈도우를 가지는 MDI 프로젝트를 만든다고 해보자. MDI는 일단 기본적인 모양이 비슷 비슷하므로 처음부터 수작업으로 만들 필요없이 창고에서 MDI 프로젝트를

읽어와 원하는 대로 변형시키는 것이 훨씬 더 쉽고 빠르다. 오브젝트 창고의 Projects 페이지에 이런 목적으로 만든 프로젝트들이 등록되어 있다. 이 중 MDI Application을 선택해 보자.

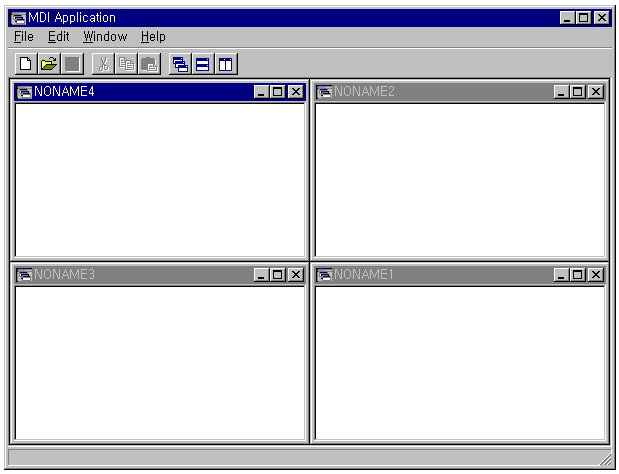



새로운 MDI 프로젝트가 작성되며 이 프로젝트를 저장할 디렉토리명을 물어온다. 적당한 디렉토리를 선택하거나 직접 입력하여 디렉토리를 만들면 새 디렉토리에 작성한 프로젝트를 저장해 줄 것이다.




다음과 같은 모양의 MDI 프로그램이 즉석에서 만들어진다. 아니 사실은 미리 만들어진 프로젝트를 복사해 주는 것이다.

그림
창고에서 꺼내온
MDI 프로젝트



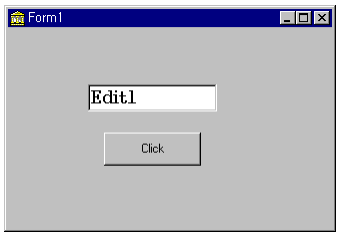

보다시피 MDI의 기본 형태를 다 갖추고 있으며 메뉴, 툴바까지 가지고 있다. 이제 이 프로젝트의 뼈대에 원하는 코드를 추가하면 된다.

델파이가 제공하는 프로젝트 외에도 우리가 만든 프로젝트를 창고에 저장하는 것도 물론 가능하며 Project/Add Project to Repository 항목을 선택하고 몇 가지 정보만 입력해 주면 된다.

## ■ 컴포넌트 템플릿

컴포넌트도 저장해 놓고 재사용할 수 있다. 그런데 폼이나 프로젝트와는 달리 컴포넌트는 오브젝트 창고에 저장되지 않고 컴포넌트 팔레트에 저장된다. 컴포넌트를 배치한 후 속성을 변경하고 이벤트 핸들러까지 작성한 후 이 컴포넌트를 템플릿으로 저장하면 속성, 이벤트 핸들러 등 컴포넌트에 대한 모든 것이 저장된다. 게다가 컴포넌트 템플릿은 두 개 이상의 컴포넌트를 같이 저장할 수도 있다.

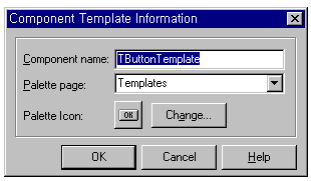
컴포넌트 템플릿의 사용법을 익히기 위해 간단한 실습을 해 보자. 새 프로젝트를 시작하고 빈 폼에 버튼과 에디트를 하나씩 배치한다. 그리고 이 컴포넌트의 속성을 적당히 수정하였다. 버튼은 Name, Caption 속성을 바꾸고 크기를 늘렸으며 에디트는 폰트를 바꾸었다.



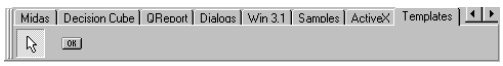

그리고 버튼의 OnClick 이벤트 핸들러까지 작성했다.

```
procedure TForm1.BtnClickClick(Sender: TObject);
begin
  Edit2.Text:='템플릿';
end;
```

버튼이 눌러지면 에디트의 텍스트를 변경하도록 하는 코드이다. 예제를 실행한 후 버튼을 눌러보면 에디트의 텍스트가 바뀔 것이다. 이 두 컴포넌트를 컴포넌트 템플릿에 저장해 보자. 저장의 대상이 되는 컴포넌트를 선택한 후 Component/Create Component Template 항목을 선택하면 다음 대화상자가 나타날 것이다.




이 대화상자에서 등록하고자 하는 컴포넌트 템플릿의 이름과 이 컴포넌트를 저장할 팔레트, 그리고 아이콘을 지정할 수 있다. 컴포넌트의 이름만 TMyCompo 로 변경한 후 OK 버튼을 눌러보자. Template 페이지에 새로운 컴포넌트가 나타날 것이다.




이 컴포넌트가 방금전에 등록한 버튼과 에디트이다. 그럼 제대로 저장이 되었는지 테스트해 보기 위해 새 프로젝트를 시작한다. 그리고 Template 페이지에서 MyCompo 컴포넌트를 가져와 폼에 놓아 보자. 방금전에 저장했던 버튼과 에디트가 폼에 그대로 나타날 뿐만 아니라 속성까지도 저장하기 전 그대로이다. 뿐만 아니라 코드 에디터를 보면 이벤트 핸들러까지 작성되어 있으므로 이대로

프로젝트를 실행할 수도 있다.

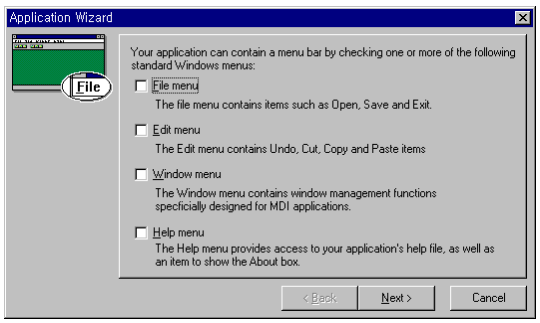
자주 사용하는 컴포넌트의 유형이 있다면 이런 식으로 컴포넌트 템플릿으로 만들어 두고 사용하면 무척 편리하다.

## ■ 마법사

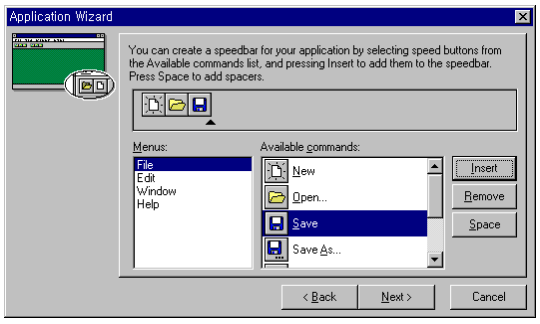
오브젝트 창고가 가지는 또 하나의 기능은 폼이나 프로젝트를 질문 형식으로 만들어주는 마법사(Wizard) 기능이다. 오브젝트 창고에 마법사 기능이 내장되어 있는 것은 아니지만 오브젝트 창고를 통해 마법사를 호출할 수 있다. Projects 페이지의 Application Wizard를 선택해 보자. 다음과 같이 질문을 해온다.

**그림**

질문 형식으로 프로그램의 뼈대를 만들어 주는 어플리케이션 전문가

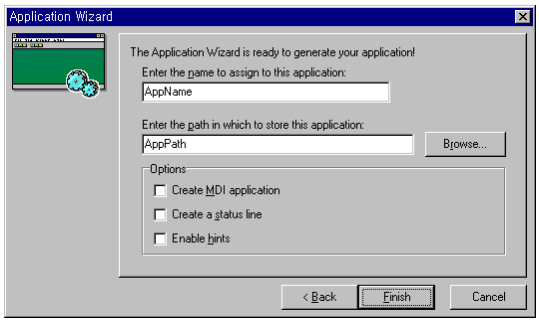



지금 만들고자 하는 프로젝트에 어떤 메뉴를 포함시킬 것인가를 묻고 있으며 이 화면에서 우리가 원하는 메뉴를 선택해 주고 Next 버튼을 누르기만 하면 된다. 다음으로 File 열기 대화상자에서 사용할 확장자명과 툴바에 표시할 아이콘의 종류를 묻는다. 원하는 아이콘을 선택한 후 Insert 버튼을 누르면 툴바에 이 아이콘들이 나타난다.

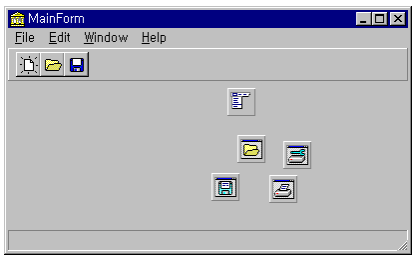



이런 종류의 질문에 계속 대답해 주면서 Next 버튼만 눌러주면 된다. 최종적

으로 묻는 질문은 프로젝트의 이름, 프로젝트를 저장할 디렉토리명, 그리고 프로젝트에 관한 몇 가지 간단한 질문이다.




이런 모든 질문에 응답하면 사용자의 응답에 맞게 프로젝트를 만들어 준다. 마법사에 의해 만들어진 프로젝트는 다음과 같다.




묻는대로 대답만 하면 직접 만들어 주므로 과연 편리하다. 그래서 뭔가 대단한 기술인 것 같지만 사실 알고 보면 별것도 아니다. 마법사는 어디까지나 잡스러운 몇 가지 일을 대신해 줄 뿐이지 스스로 알아서 프로그램을 짜 주는 것은 아니며 결국 마법사가 해 줄 수 있는 일이란 한계가 있을 수밖에 없다. 이 외에 델파이가 제공하는 마법사에는 컴포넌트 마법사, 대화상자 마법사, 데이터 베이스 폼 마법사 등이 있으며 관련 부분에서 별도로 논한다.

## ■ 복사, 상속, 사용

오브젝트 창고에 저장된 요소를 재사용하는 방법에는 세 가지가 있다. 우선 가장 간단한 방법으로는 창고에 있는 요소를 복사(Copy)해서 사용하는 방법이 있다. 복사본을 만든 후 프로젝트에 포함시키므로 복사 후 오브젝트 창고에 있는 요소와 프로젝트에 포함된 요소는 별개의 것이다. 즉 상호 변경되더라도 서로에게 전혀 영향을 미치지 않는다.

상속(Inherit)은 창고의 요소를 파생시켜 새로운 요소를 만든 후 프로젝트에

서 재사용하는 방법이다. 창고의 요소가 변경되면 프로젝트의 요소가 영향을 받지만 그 반대의 경우, 즉 프로젝트의 요소가 변경될 경우 창고의 요소는 영향을 받지 않는다.

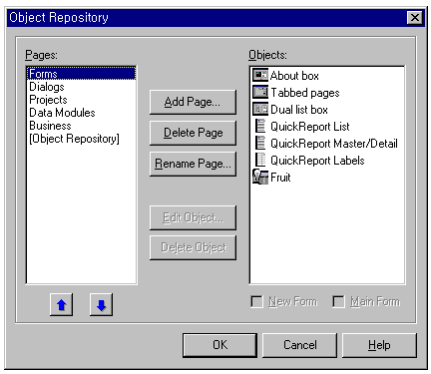
마지막으로 창고의 요소를 마치 프로젝트에 포함된 요소처럼 사용(Use)하는 방법이 있다. 이 방법은 복사본을 만들거나 상속을 하는 것이 아니기 때문에 프로젝트의 요소를 변경하면 창고의 요소도 같이 변경될 뿐만 아니라 그 요소를 사용하는 모든 프로젝트가 같이 변경된다.


세 가지 방법은 오브젝트 창고의 아래쪽에 있는 라디오 버튼으로 선택한다. 물론 필요에 따라 선택하면 되겠지만 상속과 사용은 그 효과를 정확히 알려면 아직 이 책을 좀 더 읽어야 하므로 당분간은 복사만 사용하기 바란다.

## ■ 창고 꾸미기

오브젝트 창고는 아홉 개의 페이지를 기본으로 가지고 있지만 새로운 페이지를 추가하거나 기존 페이지를 통폐합할 수도 있다. Tools/Repository를 선택하거나 아니면 오브젝트 창고의 스피드 메뉴에서 Property를 선택하면 다음과 같은 대화상자를 보여 준다.

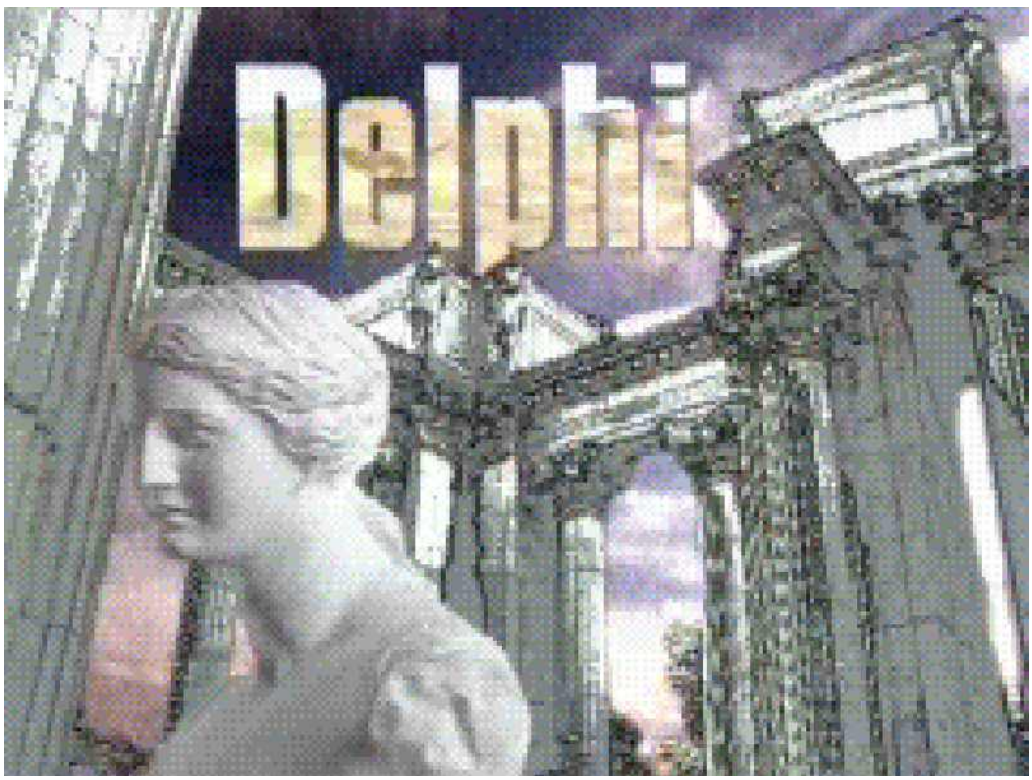
**그림**
오브젝트 창고 관리 윈도우




Add Page, Delete Page 등의 버튼을 사용하여 페이지를 관리하며 아래, 위 버튼으로 페이지의 순서를 변경한다. 자신이 늘상 애용하는 폼이나 대화상자는 자신만의 페이지에 저장하는 것도 권할만한 일이다.

# 제 5 장

# 메뉴



# 5-1 메뉴

## 가. 메뉴란

메뉴(Menu)는 윈도우즈에서 사용자의 명령을 받아들이는 가장 기본적인 유저 인터페이스이다. 프로그램에서 제공하는 모든 기능을 논리적으로 비슷한 유형끼리 모아 두어 사용자가 쉽게 찾아 편리하게 쓸 수 있도록 해준다. 프로그램을 처음 사용하는 초보 사용자라 하더라도 메뉴를 사용하면 프로그램에 어떤 기능들이 있다는 것을 신속하게 파악할 수 있고 메뉴 항목의 이름이 워낙 설명적이므로 쉽게 프로그램에 익숙해지게 된다.

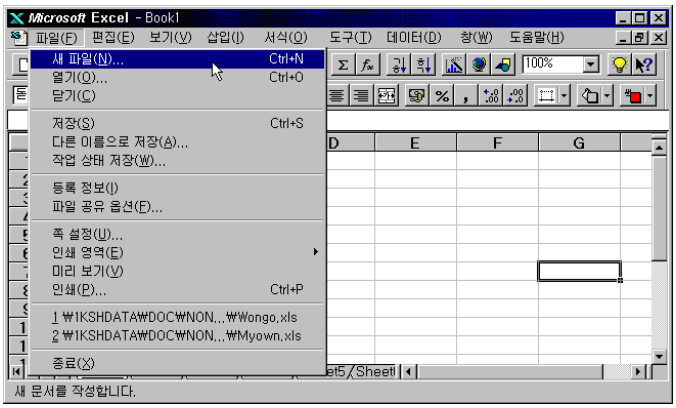
도스 프로그램에서도 메뉴가 사용되고 있으며 요즘은 메뉴가 워낙 일반화되어 있어 메뉴가 없는 프로그램이 없을 정도이다. 윈도우즈에서는 아예 운영체제 차원에서 메뉴를 지원해 주므로 프로그래머도 메뉴를 쉽게 만들 수 있고 사용자도 통일된 방법으로 메뉴를 사용할 수 있다. 델파이도 물론 메뉴 제작을 지원하며 현재까지 나와있는 어떤 메뉴 제작 툴보다 쉽고 빠르게 다양한 형태의 메뉴를 만들 수 있다.

## 나. 메뉴의 종류

윈도우즈에서 사용하는 메뉴에는 크게 두 가지 종류가 있다. 물론 더 다양한 형태의 메뉴를 생각할 수 있고 실제로 별 희한한 형태의 메뉴가 사용되고 있기는 하지만 델파이가 지원하는 메뉴는 일단 다음 두 가지가 있다.

우선 메인 메뉴(Main Menu)가 있다. 윈도우 타이틀 바의 바로 아래에 위치하며 프로그램의 모든 기능을 망라하여 찾기 쉽도록 분류해 놓은 메뉴이다. 메인 메뉴는 상단에 메뉴의 이름들이 나열되어 있으며 메뉴를 클릭하여 호출하면 해당 메뉴가 아래로 펼쳐진다(pull down). 펼쳐진 메뉴 리스트에는 사용자의 명령을 받아들이는 메뉴 항목들이 있다. 다음은 마이크로소프트사가 만든 엑셀의 메뉴이며 항상 보아오던 바이다.

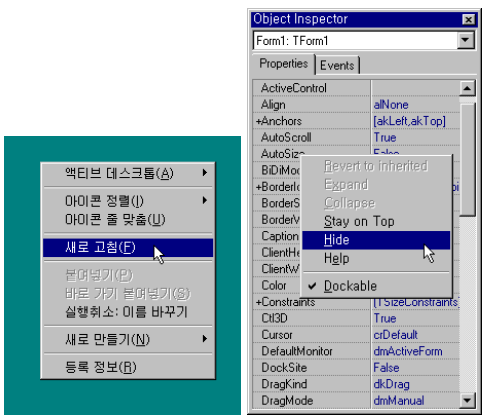
그림
액셀의 메뉴




두 번째로 팝업 메뉴(Popup Menu)가 있다. 마우스의 오른쪽 버튼을 누르면 나타나며 마우스를 누른 위치에서 자주 사용하는 명령들을 모아 놓는다. 델파이에서 제공하는 스피드 메뉴도 팝업 메뉴이며 오브젝트 인스펙터 위에서나 폼 위에서 오른쪽 버튼을 누르면 나타난다. 팝업(popup)이란 불쑥 위로 솟아난다는 뜻인데 언제든지 마우스의 오른쪽 버튼을 누르기만 하면 나타나기 때문에 붙여진 이름이다. 또한 마우스 위치에 따라 메뉴 모양이 달라진다고 해서 컨텍스트 메뉴(Context Menu)라고 하기도 하며 로컬 메뉴(Local Menu)라는 용어도 쓰인다.

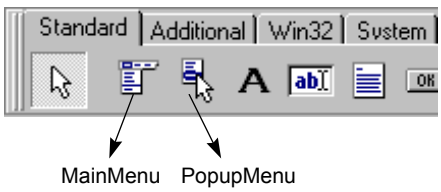
윈도우즈 운영체제도 다양한 팝업 메뉴를 사용하고 있으며 여러 가지 윈도우즈 프로그램에서 팝업 메뉴를 볼 수 있다. 또한 지금 여러분들이 사용하고 있는 델파이도 개발 환경 곳곳에 팝업 메뉴가 정의되어 있는데 델파이에서는 이 메뉴들을 스피드 메뉴라고 부른다.

그림
마우스 오른쪽 버튼을 누르면 나타나는 팝업 메뉴




델파이는 이 두 가지 종류의 메뉴를 컴포넌트로 제공한다. Standard 팔레트

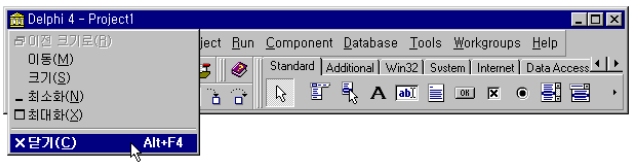
의 제일 처음에 있는 두 개의 컴포넌트가 각각 MainMenu 컴포넌트, PopupMenu 컴포넌트이다.




이 두 개의 컴포넌트를 폼에 배치하고 메뉴 항목의 속성을 설정하여 메뉴를 작성하며 각 메뉴 항목의 이벤트 핸들러를 작성하여 사용자의 명령을 처리하게 된다.

메인 메뉴, 팝업 메뉴 외에도 시스템 메뉴(System Menu)라는 것이 있다. 시스템 메뉴는 메인 윈도우나 대화상자의 좌상단에 있는 아이콘을 누르면 나타나며 키보드 단축키 Alt+Space로 언제든지 호출할 수 있다. 다음은 델파이의 시스템 메뉴이다.

그림
시스템 메뉴

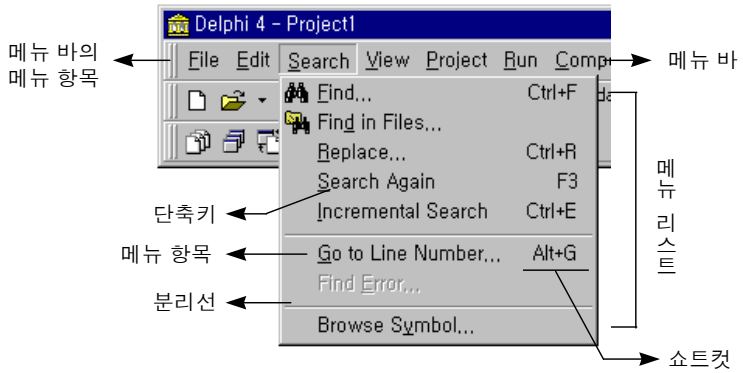



이 메뉴에는 윈도우 크기 조정, 닫기, 이동 등의 기본적인 윈도우 조작 명령들이 있는데 운영체제가 관리하므로 일반적인 프로그래밍 대상은 아니다. 하지만 특수한 경우에는 시스템 메뉴를 수정하여 원하는 항목을 넣을 수도 있다.

# 다. 메뉴에 관한 용어

메뉴 작성에 대해 알아보기 전에 우선 메뉴에 관한 용어부터 간단하게 정리해 보도록 하자. 메뉴는 버튼이나 레이블에 비해서 상당히 복잡한 컴포넌트이기 때문에 사용하는 용어도 다소 복잡하다. 용어를 정확히 알아야 본문이나 도움말을 참고할 때 지장이 없을 것이다. 델파이의 Search 메뉴를 예로 들어 설명하였다.

그림
메뉴를 이루는 여러 가지 구성 요소




이 중에서도 특히 File, Edit 등의 메뉴 바에 있는 메뉴 항목과 아래로 펼쳐져 있는 메뉴 리스트에 있는 Cut, Copy 등의 메뉴 항목을 잘 구분해야 한다. File, Edit 등을 누를 경우는 아래로 메뉴 리스트가 펼쳐질 뿐이며 별다른 동작을 하지는 않는다. 반면 메뉴 리스트의 메뉴 항목은 마우스로 클릭할 경우 명령으로 인식되어 모종의 동작이나 작업을 하게 된다. 그래서 메뉴 바의 메뉴 항목에는 보통 이벤트 핸들러를 정의하지 않으며 메뉴 리스트의 메뉴 항목에만 이벤트 핸들러를 정의한다.

물론 메뉴 바에 있는 메뉴 항목에 이벤트 핸들러를 정의하지 못하는 것은 아니지만 일반적으로 그렇게 하지 않으며 바람직하지도 않다. 사용자들은 보통 프로그램을 설치하고 난 후에 어떤 기능들이 있는지 마우스로 메뉴를 쿡쿡 찍어보는 습성이 있기 때문이다. 이때 메뉴 리스트가 열리는 것은 상관없지만 어떤 명령이 실행된다면 원치않는 결과를 유발할 수도 있다.

# 5-2 메뉴 작성

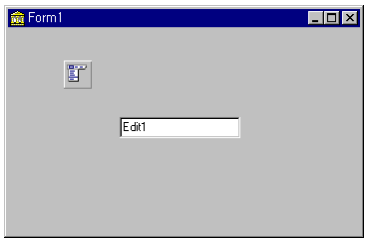
## 가. 전체적인 순서

5jang
menu1

메뉴는 버튼이나 레이블 등에 비한다면 무척이나 복잡한 물건이다. 모양이 정해져 있지도 않고 포함되는 메뉴 항목도 만들기 나름이며 단축키, 쇼트컷, 분리선 등 신경써야 할 많은 것들을 가지고 있다. 그래서 메뉴 컴포넌트를 단순히 폼에 배치만 한다고 해서 메뉴가 만들어지는 것은 아니며 정해진 절차를 따라 만들어 나가야 한다. 개략적인 절차와 방법에 대해 훑어보고 난 후 직접 실습을 해보도록 하자.

### 1 컴포넌트를 배치한다

File/New Application을 선택해 새로운 프로젝트를 시작한 후 빈 폼에 다음과 같이 메인 메뉴 컴포넌트와 에디트 컴포넌트를 배치한다.



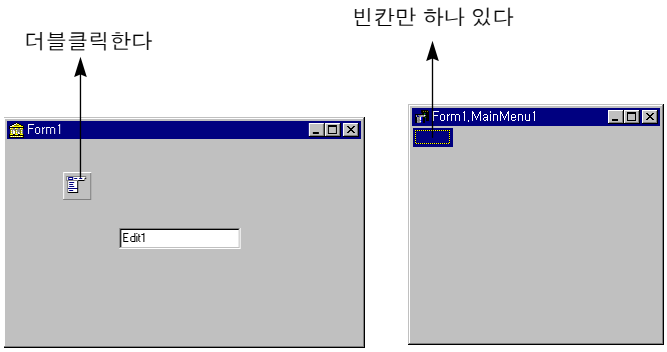

예제를 간단하게 하기 위해 에디트 하나와 메인 메뉴 컴포넌트만 배치하였다. 메인 메뉴 컴포넌트는 아이콘 모양 그대로 폼 상에 나타나며 폼의 어느 위치에 있든지 상관없다. 폼에 있기만 하면 위치나 크기에 상관없이 메뉴가 생성된다. 메뉴 컴포넌트 아이콘은 실행 중에는 폼에서 사라지고 대신 타이틀 바 아래에 실제 메뉴가 나타난다. 간단한 예제이므로 컴포넌트의 이름 및 속성은 그대로 디폴트를 사용하기로 하자.

### 2 메뉴 디자이너를 연다

델파이는 메뉴 항목 디자인을 위해 메뉴를 쉽게 만들 수 있도록 도와주는 메뉴 디자이너(Menu Designer)를 제공한다. 폼 상의 메인 메뉴 아이콘을 더블클

릭하면 다음과 같은 메뉴 디자이너 윈도우가 열릴 것이다.

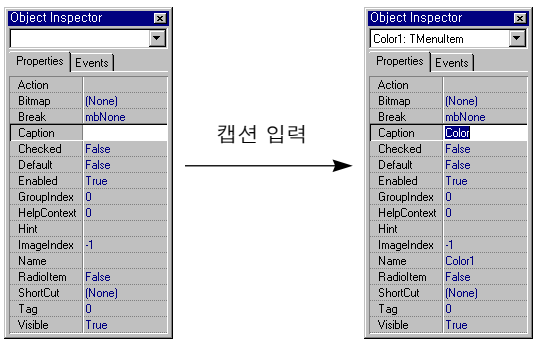
그림
메뉴 디자이너 열기




처음 만드는 메뉴이므로 메뉴 디자이너에는 메뉴 항목이 없고 빈칸(Placeholder)만 하나 있다. 이 빈칸을 시작점으로 해서 메뉴 항목을 만들어 나간다.

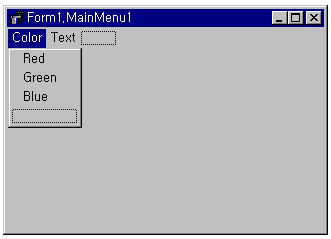
## 3 메뉴 항목을 입력한다

메뉴 디자이너의 빈칸에 파란색의 반전 막대가 위치해 있는데 이 빈칸에 메뉴 항목들을 만들어 나간다. 오브젝트 인스펙터에서 Caption이나 Name 속성을 정의하면 메뉴 항목이 만들어진다. 메뉴 디자이너가 처음 열렸을 때는 아무 항목도 만들어져 있지 않은 상태이며 오브젝트 인스펙터 윗쪽의 오브젝트 선택란이 비어 있다. 일단 오브젝트 인스펙터의 Caption에 Color라고 입력해 보자. Color라는 메뉴 항목이 생성되며 메뉴 디자이너에 생성된 메뉴 항목이 보일 것이다. Caption 속성이 정의되면 Name 속성도 따라서 정의되며 새로운 메뉴 아이템(MenuItem) 컴포넌트가 만들어진다.

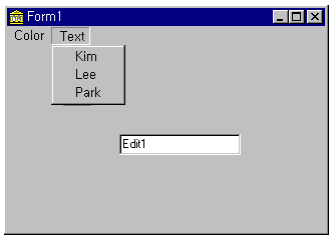



메뉴 아이템은 메뉴 디자이너에 의해 만들어지기 때문에 컴포넌트 팔레트에

는 없다. 하지만 메뉴 아이템도 버튼이나 에디트와 마찬가지로 컴포넌트의 일종이므로 속성이나 이벤트를 가지며 오브젝트 인스펙터를 통해 값을 변경한다. Color 아래에 Red, Green, Blue 항목을 만들어 주고 Text 아래에 Kim, Lee, Park을 만들어준다(아래 그림 참조). 메뉴 바의 오른쪽 끝과 각 메뉴 리스트의 아래쪽에는 항상 빈칸이 하나씩 존재하는데 이 빈칸에 새로운 메뉴 항목을 입력해 주면 된다. 메뉴 항목의 Caption을 수정하려면 파란색 반전 막대를 옮긴 후 오브젝트 인스펙터에 수정할 값을 입력하도록 한다. 여기까지 입력을 마치면 메뉴 디자이너는 다음과 같은 모양을 가지며 폼에도 같은 모양의 메뉴가 나타난다.




메뉴 디자이너




폼

단 폼에는 메뉴 항목 추가를 위한 빈칸(placeholder)이 없다. 메뉴 항목을 입력한 후에도 삭제, 삽입 등의 메뉴 편집을 할 수 있는데 이에 관해서는 잠시 후에 논한다.

## 4 코드를 작성한다

메뉴를 만드는 과정은 메뉴 디자이너를 통해 메뉴 아이템(MenuItem) 컴포넌트를 배치하고 속성을 정의하는 과정의 연속이다. 다음으로 할 일은 프로그램 실행중에 사용자가 메뉴 항목을 선택할 경우 어떤 동작을 할 것인가를 정의하는 것이다. 즉 각각의 메뉴 항목의 OnClick 이벤트에 대한 이벤트 핸들러를 만드는 것이다. 메뉴 항목의 이벤트 핸들러를 작성하는 방법은 다음 두 가지가 있다

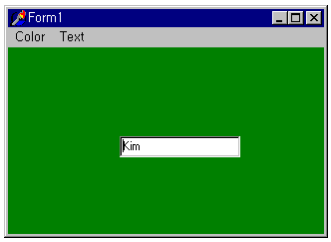
① 메뉴 디자이너에서 메뉴 항목을 더블클릭한다. 해당 메뉴 항목의 OnClick 이벤트 핸들러가 작성되고 코드 에디터가 열린다.

② 폼에서 메뉴 항목을 클릭하여 이벤트 핸들러를 작성한다. 폼에서는 클릭만 하면 코드 에디터가 열린다.

여섯 개의 메뉴 항목에 대해 다음과 같이 이벤트 핸들러를 작성하라. Color 메뉴 아래의 Red, Green, Blue 세 항목은 폼의 색상을 바꾸고 Text 메뉴 아래의 Kim, Lee, Park 세 항목은 에디트의 Text 속성을 바꾼다.

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  Form1.Color:=clRed;
end;

procedure TForm1.Green1Click(Sender: TObject);
begin
  Form1.Color:=clGreen;
end;

procedure TForm1.Blue1Click(Sender: TObject);
begin
  Form1.Color:=clBlue;
end;

procedure TForm1.Kim1Click(Sender: TObject);
begin
  Edit1.Text:='Kim';
end;

procedure TForm1.Lee1Click(Sender: TObject);
begin
  Edit1.Text:='Lee';
end;

procedure TForm1.Park1Click(Sender: TObject);
begin
  Edit1.Text:='Park';
end;
```

이벤트 핸들러까지 만들었으면 메뉴에 관한 모든 처리가 끝났다. 프로그램을 저장하고 직접 실행해 보아라. 메뉴 항목을 선택하면 폼의 색상이 바뀌고 에디트 박스의 텍스트가 변경될 것이다.




메뉴 디자이너 사용법과 메뉴 작성에 관한 자세한 사항은 잠시 후에 알아보기로 하고 일단 메뉴를 작성하는 절차를 머리 속에 정리해 두자.

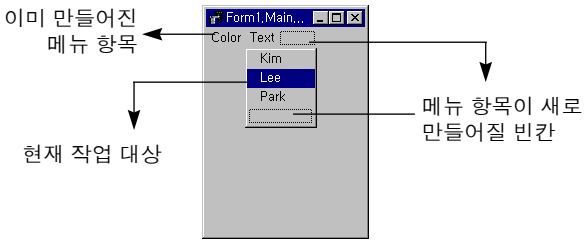
그림
메뉴 작성 과정


그림이 좀 복잡해 보일지 모르겠지만 앞에서 실습해 본 절차를 머리속으로 잘 정리해 두기만 하면 된다. 한번씩 실습해 보고 익숙해지게 되면 메뉴를 만드는 방법이 무척 쉽고 직관적임을 알 수 있을 것이다.

# 나. 메뉴 디자이너

메뉴 작성의 핵심은 메뉴 디자이너를 사용하는 것이며 메뉴 디자이너를 자유자재로 쓸 줄 알아야 메뉴를 마음대로 디자인할 수 있다. 메뉴 디자이너는 무척 간단하게 구성되어 있으며 사용 방법이 아주 직관적이므로 힘들이지 않고 메뉴를 작성할 수 있게 해준다. 우선 메뉴 디자이너를 열려면 폼에서 메인 메뉴 컴포넌트를 더블클릭하면 된다고 앞에서 설명했다. 또는 메인 메뉴 컴포넌트를 선택한 후 오브젝트 인스펙터에서 Items 속성을 더블클릭해도 메뉴 디자이너가 열린다.

이미 만들어진 메뉴 항목은 폼상에 나타나는 것과 같은 모양으로 메뉴 디자이너에 나타나며 우측과 하단에 메뉴 항목을 추가할 수 있는 빈칸(Place holder)이 있다. 물론 이 빈칸은 디자인중에만 사용되므로 실행중에는 보이지 않는다. 파란색의 반전 막대는 현재 작업 대상이 되는 메뉴 항목을 나타내는 일종의 커서이며 이 반전 막대를 움직여 메뉴 항목을 입력시켜 나간다.




메뉴 디자이너에서 편집된 메뉴는 즉각 폼 상에 나타나므로 메뉴 모양을 확인하기 위해 프로그램을 일일이 실행해 보지 않아도 된다. 메뉴를 만들면서 메뉴가 어떤 모양을 가지는가를 폼을 통해 직접 확인할 수 있다.

메뉴 편집이 끝났으면 메뉴 디자이너를 최소화시켜 놓거나 아니면 아예 닫아버려도 메뉴 컴포넌트를 더블클릭하여 언제든지 다시 열 수 있으므로 별도의 저장 명령을 줄 필요는 없다. 메뉴 디자이너는 표준 윈도우이므로 최대화, 최소화, 이동 등이 자유롭다. 작업하기 제일 적당한 위치에 마음에 드는 크기로 배치해 놓고 사용하도록 하자.

## ■ 메뉴 항목의 속성

메뉴 디자이너에 의해 생성되는 메뉴 항목(MenuItem)도 일종의 컴포넌트이며 속성과 이벤트를 가진다. 메뉴 디자이너에서 파란색의 반전 막대가 위치해 있는 메뉴 항목이 현재 선택된 메뉴 항목이며 이 메뉴 항목의 속성이 오브젝트 인스펙터에 나타난다. Caption, Name, Tag 등 모든 컴포넌트가 가지는 속성들 외에 다음과 같은 속성들이 있다.

### ✍ Checked

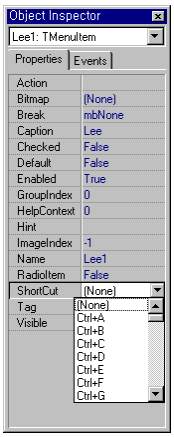
메뉴 항목 옆에 체크 표시를 붙여 선택되었음을 알린다. 디폴트로 False이지만 디자인시나 실행중 언제라도 이 속성을 True로 만들면 좌측에 체크 표시가 나타난다. 참, 거짓의 진위적인 옵션을 설정하거나 두 가지 상태 중 하나를 지정하는 메뉴 항목에 많이 사용된다.

### Enabled

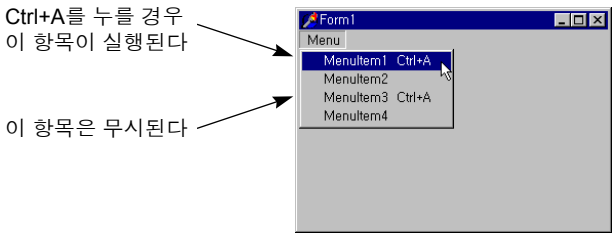
메뉴 항목이 사용 가능한가 아닌가를 지정하며 디폴트는 True이므로 만들어진 모든 메뉴 항목이 사용 가능하다. 실행중에 사용이 불가능한 메뉴 항목, 예를 들어 프린터가 없는 시스템에서의 Print 메뉴나 선택된 문장이 없을 때의 Cut, Copy 등은 Enabled 속성을 False로 만들어 사용할 수 없도록 한다. 사용할 수 없는 메뉴 항목은 회색으로 희미하게 나타나며 마우스로 선택해도 OnClick 이벤트가 발생하지 않는다.

### ShortCut

쇼트컷, 즉 메뉴 항목을 빠르게 실행할 수 있는 키를 정의한다. Ctrl+A, F5 등과 같은 조합키가 사용된다. 새로 만들어지는 메뉴 항목의 디폴트 ShortCut 속성은 None이며 쇼트컷이 정의되어 있지 않다. 메뉴 항목을 선택한 상태에서 오브젝트 인스펙터의 ShortCut 속성을 설정하면 된다. ShortCut 속성을 펼쳐보면 다음과 같은 리스트 박스가 열리는데 이 목록에서 제공하는 쇼트컷 중에 하나를 선택하도록 한다.




쇼트컷을 만들 때 주의할 것은 델파이가 중복 검사를 해주지 않으므로 디자인하는 사람이 알아서 중복되지 않도록 해주어야 한다는 점이다. 두 개의 메뉴 항목이 똑같이 Ctrl+A를 쇼트컷으로 가질 경우 뒷쪽에 있는 쇼트컷은 무시된다.

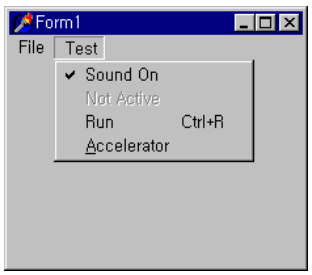



쇼트컷 속성을 지정하면 메뉴 리스트의 우측에 쇼트컷 키에 대한 설명이 같이 출력되어 이 메뉴 항목의 쇼트컷이 어떻게 지정되어 있는지를 금방 알 수 있게 된다. 예를 들어 델파이의 Edit 메뉴를 보면 Cut 항목 옆에 Ctrl+X라고 되어 있어 Ctrl+X를 눌러 Cut 동작을 할 수 있다는 것을 보여준다.

### 단축키

쇼트컷과 단축키는 똑같이 메뉴를 좀 더 빨리 선택할 수 있도록 하기 위해 만들어진 것이다. 차이점이라면 쇼트컷은 메뉴가 떠 있지 않아도 사용할 수 있으며 단축키는 메뉴가 떠 있을 때 Alt키와 함께 사용된다는 점이다. Caption 속성에 &(Ampersand;앰퍼샌드라고 읽음)를 넣어 두면 &다음의 문자가 단축키가 되며 Alt 키와 함께 누를 경우 즉시 메뉴가 실행된다. 예를 들어 &Color이라는 캡션을 정의했을 경우 메뉴에는 Color라고 출력되며 Alt+C를 눌러 메뉴를 빠르게 부를 수 있다.

**그림**
메뉴 항목의 여러 가지 상태




단축키도 쇼트컷과 마찬가지로 중복 검사를 해주지 않으므로 같은 메뉴 리스트에 있는 항목끼리 같은 단축키를 가지지 않도록 해주어야 한다. 메뉴 항목의 앞 글자를 단축키로 쓰는 것이 가장 기억하기 쉽지만 Cut, Copy 등의 항목은 앞 글자가 중복되므로 중간의 한 문자를 단축키로 지정해 주어야 한다. 메뉴 항목이 한글로 되어 있을 때는 파일(&F), 편집(&E) 등과 같이 괄호안에 영문자 하나를 넣어서 단축키를 지정하는 것이 관례다.

### ✍ Visible

메뉴 항목을 보이지 않도록 숨기고자 할 때 이 속성을 사용한다. 디폴트값은 True이며 만들어진 모든 메뉴 항목이 보인다. 실행중에 메뉴 항목을 아예 안보이도록 할 때 이 속성을 False로 만들어 주면 된다. 그러나 당장 사용할 수 없는 메뉴라 하더라도 숨겨버리는 것은 별로 바람직하지 않으며 그보다는 Enabled 속성을 False로 만들어 사용 금지 표시를 하는 것이 좋다. 그래야 사용자들은 현재 상황에서 이 메뉴 항목을 쓰지 못한다는 것을 알 수 있게 된다. 만약 메뉴 항목을 완전히 숨겨버리면 사용자들은 "그놈의 메뉴가 어디 있더라" 하면서 메뉴 리스트를 계속 뒤지게 될 것이다. 이 속성은 실행중에 메뉴를 추가/삭제하는 특수한 용도로만 사용하는 것이 좋다.

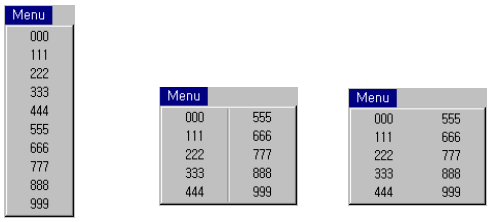
### ✍ Break

하나의 메뉴 리스트에 메뉴 항목이 너무 많을 경우 메뉴 리스트의 길이가 너무 길어지므로 몇 개의 열로 나누도록 한다. 세 가지 속성값 중 하나를 지정한다.

| 속성값 | 의미 |
|---|---|
| mbNone | 열을 나누지 않는다. |
| mbBarBreak | 이 속성이 지정된 항목에서 열을 분리하며 열 사이에 수직의 분리선을 삽입한다. |
| mbBreak | 이 속성이 지정된 항목에서 열을 분리하며 분리선은 삽입하지 않는다. |

Menu라는 메뉴밑에 000~999까지 10개의 메뉴 항목을 만든 후 555 항목의 Break 속성을 변경하여 보았다. Break 속성이 mbNone일 경우는 모든 메뉴가 일렬로 나열되지만 mbBarBreak나 mbBreak 속성을 줄 경우는 다음과 같이 2행으로 메뉴가 나누어진다.

그림
Break 속성에 따른 메뉴의 모양



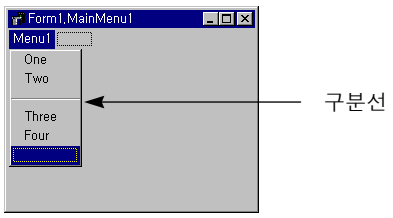

메뉴 항목이 너무 많아 화면에 다 나타내지 못한다거나 보기가 불편할 경우 Break 속성을 사용할 수 있지만 여러 단으로 나누었을 경우는 메뉴를 선택하기가 번거로워지므로 그리 바람직하지는 못하다. 정 메뉴 항목이 많으면 하위 메뉴를 두든가 아니면 두 개의 메뉴 리스트로 분리하는 것이 더 바람직하다. 윈도우즈 프로그램 중에 이단 메뉴를 사용하는 프로그램은 거의 구경하기가 힘들다.

메뉴 항목의 속성들은 디자인시에 지정할 수도 있지만 실행중에도 언제든지 변경할 수 있다. 특히 Checked 속성과 Enabled 속성은 실행시에 자주 변경되는 속성이며 Caption 속성도 가끔 변경하는 경우가 있다.

## ■ 분리선 만들기

메뉴 리스트에는 비슷한 성격의 메뉴 항목을 모아 놓지만 그 중에서도 성격이 다른 메뉴 항목을 구분해야 할 필요가 있다. 이 때 각 메뉴 항목 그룹 사이에 수평선을 그어 구분을 하는데 이 선을 구분선(Separator)이라고 한다. 델파이의 File 메뉴를 보면 구분선이 다섯 개나 삽입되어 있음을 볼 수 있을 것이다. 구분선을 삽입하려면 Caption 속성을 -(하이픈)으로 정의해 주면 된다. Caption 속성이 하이픈으로 정의된 메뉴 항목은 구분선으로 나타날 뿐 실제로 메뉴 항목은 아니므로 OnClick 이벤트를 가지지도 못하며 실행중에 사용자에 의해 선택될 수도 없다. 단지 장식일 뿐이다.

그림
메뉴 항목의 그룹을 구분하는 구분선



## ■ 메뉴 항목에 이름 붙이기

메뉴 항목도 엄연한 컴포넌트이므로 Name 속성이 있고 메뉴 항목의 제목을 정의하는 Caption 속성도 있다. 메뉴 항목은 다른 컴포넌트와는 달리 여러 개가 존재하며 많을 경우 수십 개가 있을 수도 있기 때문에 메뉴 항목 컴포넌트에 일일이 이름(Name 속성)을 정해주기란 무척이나 번거로운 일이다. 그래서 메뉴 항목에 이름을 붙이는 방법은 다른 컴포넌트의 경우와 조금 다르다. 메뉴 항목에 이름을 붙이는 방법은 다음 두 가지를 생각할 수 있다.

① Name 속성에 이름을 직접 지정해 준다. 이름을 정하는 아주 원론적인 방법이다.

② Name 속성은 입력하지 않고 Caption 속성에 제목을 입력한다. 그러면 Name 속성이 Caption 속성에 따라 적절하게 자동으로 작성된다.

통상적으로 두 번째 방법이 많이 쓰이는데 메뉴 항목의 개수가 많기 때문에 개별적으로 이름을 붙이는 것보다 Caption과 Name을 비슷하게 해두는 편이 여러 가지 면에서 편리하기 때문이다. Caption만 입력시켜 주면 델파이는 사용자가 입력한 Caption을 근거로 하여 Name 속성을 작성하는데 같은 Caption을 가진 메뉴 항목이 있을 수도 있으므로 Caption 뒤에 일련번호를 붙인다. Caption이 File이면 Name은 File1, Red이면 Red1이 되며 만약 같은 캡션을 가진 메뉴 항목이 있을 경우는 File2, Red2 등과 같이 일련번호를 증가시킨다.

만약 Caption이 명칭 규칙에 부적절한 이름일 경우, 예를 들어 숫자로 시작하거나 공백이나 특수기호가 있을 경우 델파이가 알아서 명칭 규칙에 적합하게, 다른 메뉴 항목과 이름이 중복되지 않도록 Name을 작성해 준다. Caption에 따라 델파이가 작성한 Name 속성의 예를 보자.

**표** 델파이가 작성해 주는 메뉴 항목의 NAME 속성

| Caption | Name | 설명 |
|---|---|---|
| Blue | Blue1 | 숫자 1만 붙임 |
| File | File1 | 숫자 1만 붙임 |
| 123 | N1231 | 앞에 영문자 N을 추가시킴 |
| ### | N2 | 기호를 영문자와 숫자로 바꾸어 줌 |
| girl friend | girlfriend1 | 공백을 없애줌 |
| - | N3 | 분리선을 영문자와 숫자로 바꾸어 줌 |

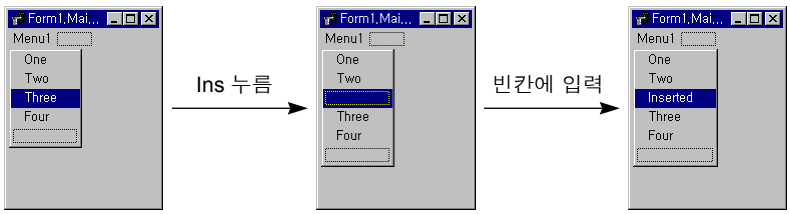
| | | |
|---|---|---|
| File | File2 | 앞에 나온 항목과 중복되므로 숫자 2를 붙임 |

숫자나 기호 등으로 시작되는 캡션의 경우 캡션을 곧바로 Name 속성으로 사용할 수 없으므로 영문 알파벳 N을 붙여 Name 속성을 정의해 준다. 메뉴 항목에 이름을 붙이는 방법은 델파이 내부에서 일어나는 일이므로 크게 신경쓸 필요가 없을지도 모른다. 괜히 복잡하게 설명한 것 같은데 어쨌든 결론은 메뉴 항목을 만들 때는 Name 속성에는 특별히 신경쓸 필요가 없으며 Caption 속성만 알아보기 쉽게 작성하면 된다는 것이다.

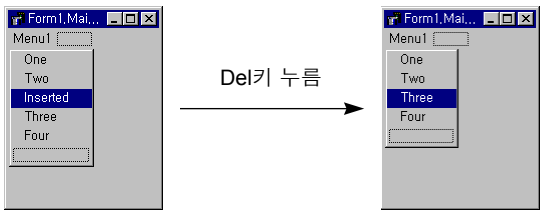
# 다. 메뉴의 편집

이미 메뉴를 만들어 놓고 수정할 때도 메뉴 디자이너를 사용한다. 먼저 오타나 실수에 의해 메뉴 항목의 이름을 변경해야 할 경우를 생각할 수 있는데 이때는 수정하고자 하는 메뉴 항목에 반전 막대를 위치시키고 오브젝트 인스펙터에서 Caption 속성을 편집하면 된다. 예를 들어 Color라는 메뉴 항목을 Calar이라고 잘못 입력했다면 이 메뉴 항목을 선택한 후 오브젝트 인스펙터에서 Caption 속성을 다시 입력해 주면 된다. 아주 쉽다.

메뉴 항목을 중간에 삽입하고자 할 때는 삽입하고자 하는 위치에 반전 막대를 두고 Ins키를 누른다. 그러면 반전 막대가 있는 위치에 빈칸이 하나 삽입되는데 이 빈칸에 원하는 메뉴 항목을 입력하면 된다.



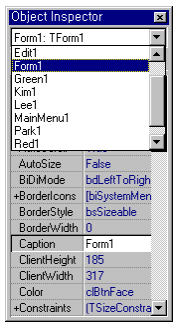

메뉴의 삭제는 더욱 간편하다. 삭제하고자 하는 메뉴 항목에 반전 막대를 두고 Del 키만 누르면 된다.




메뉴 항목을 다른 위치로 옮기거나 순서를 바꿀 때는 마우스로 드래그하여 원하는 위치에 떨어뜨리기만 하면 된다.


하위 메뉴가 있는 항목을 드래그 하면 하위 메뉴까지 모두 이동되는데 이를 이용하면 다단계로 구성된 메뉴를 쉽게 만들 수 있다. 메뉴를 편집하는 일은 비교적 상식과 일치하므로 쉽게 익힐 수 있을 것이다.

메뉴 디자이너를 통하지 않고 메뉴 항목의 속성을 변경할 수도 있다. 메뉴 디자이너가 없으면 메뉴 항목을 선택할 수 없을 것 같지만 오브젝트 인스펙터의 상단에 있는 객체 선택기에는 폼에 놓여진 모든 컴포넌트의 목록이 나타나므로 오브젝트 인스펙터를 통해 속성을 변경하고자 하는 메뉴 항목을 선택하면 된다. 메뉴 항목도 하나의 컴포넌트이며 폼의 구성 요소라는 사실만 알면 쉽게 이해할 수 있을 것이다. 객체 선택기에서 메뉴 항목을 선택한 후 오브젝트 인스펙터에서 원하는 속성을 변경하면 된다.




객체 선택기에서 직접
메뉴 항목을 선택한다.

메뉴 항목이 컴포넌트의 일종이라는 것은 델파이가 작성하는 소스를 보면 쉽게 알 수 있다. 다음은 Menu1_f.pas 파일의 앞부분이다.

```
unit Menu1_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, StdCtrls;

type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    Edit1: TEdit;
    Color1: TMenuItem;
    Red1: TMenuItem;
    Green1: TMenuItem;
    Blue1: TMenuItem;
    Text1: TMenuItem;
    Kim1: TMenuItem;
    Lee1: TMenuItem;
    Park1: TMenuItem;
    procedure Red1Click(Sender: TObject);
    procedure Green1Click(Sender: TObject);
    procedure Blue1Click(Sender: TObject);
    procedure Kim1Click(Sender: TObject);
    procedure Lee1Click(Sender: TObject);
    procedure Park1Click(Sender: TObject);
    procedure Text1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

델파이는 폼에 컴포넌트를 놓을 때마다 소스 파일에 컴포넌트형의 변수를 선언한다. 폼의 타입 선언부를 보면 MainMenu1, Edit1 등의 컴포넌트가 있고 그 아래에 메뉴 디자이너로 만든 TMenuItem 형의 컴포넌트들, 즉 메뉴 항목들이

있다. 보다시피 메뉴 항목도 에디트나 버튼과 같은 컴포넌트의 일종이다.

# 라. 실행중 메뉴 속성 변경

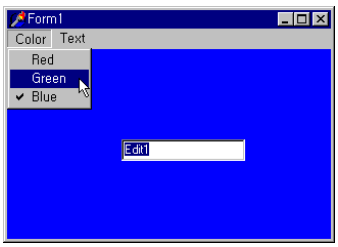
메뉴의 속성은 디자인중에 오브젝트 인스펙터로 변경하는데 실행중에도 속성을 변경할 수 있다. 이때 변경의 대상이 되는 속성은 Enabled, Checked 등이 대표적인데 Enabled 는 메뉴 항목을 사용금지시키고 Checked 는 메뉴 항목 옆에 체크 표시를 달아준다. 앞에서 만든 Menu1 예제에 약간의 코드를 추가하여 현재 선택된 메뉴 항목 옆에 체크 표시를 달아보도록 하자. 즉 Color 메뉴 아래에 현재 선택된 색상 옆에 체크 표시를 달아주고 Text 메뉴 아래에 현재 에디트에 입력된 문자열 옆에 체크 표시를 달아보자.

실행중에 체크 표시를 제어하는 데는 세 가지 방법이 있는데 셋 다 실습해 보자. 우선 가장 간단한 방법으로 메뉴 항목이 선택될 때인 OnClick 이벤트 핸들러에서 메뉴 항목의 Checked 속성을 변경하는 것이다. Color 아래의 세 메뉴 핸들러를 다음과 같이 수정해 보자.

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  Form1.Color:=clRed;
  Red1.Checked:=True;
  Blue1.Checked:=False;
  Green1.Checked:=False;
end;

procedure TForm1.Green1Click(Sender: TObject);
begin
  Form1.Color:=clGreen;
  Red1.Checked:=False;
  Blue1.Checked:=False;
  Green1.Checked:=True;
end;

procedure TForm1.Blue1Click(Sender: TObject);
begin
  Form1.Color:=clBlue;
  Red1.Checked:=False;
  Blue1.Checked:=True;
  Green1.Checked:=False;
```

```
end;
```

세 핸들러의 구조가 동일하므로 대표적으로 Red1Click 이벤트 핸들러만 보도록 하자. Red1 항목이 선택되었을 때 폼의 Color 속성을 변경하고 Red1 의 Checked 속성은 True 로 만들어 이 메뉴 항목 옆에 체크 표시를 달도록 하였다. 그리고 나머지 Blue1, Green1 의 Checked 속성은 False 로 만들어 체크 표시를 없애준다. Green1, Blue1 핸들러의 구조도 이와 동일하되 체크 표시를 하는 메뉴 항목만 다를 뿐이다. 이제 예제를 다시 실행시켜 보자.




파란색이 선택된 상태에서 메뉴 리스트를 열어 보면 Blue 항목 옆에 체크 표시가 나타날 것이며 색상을 변경하면 체크 표시도 변경된 색상으로 이동하게 된다. 메뉴 항목의 OnClick 이벤트에서 메뉴의 체크 상태를 적절하게 변경해 주기 때문이다.

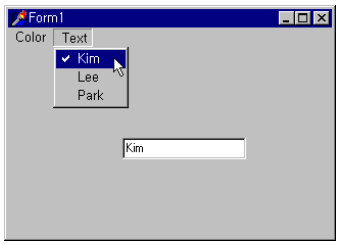
체크 표시를 변경하는 두 번째 방법은 각 메뉴 항목의 위쪽 메뉴의 OnClick 이벤트 핸들러에 코드를 작성하는 방법이다. 개별 메뉴 항목의 OnClick 이벤트에서 체크 표시를 변경하는 것은 메뉴 외에 명령을 내리는 다른 방법이 존재할 경우 정확하지 못할 수도 있다. 예를 들어 폼의 색상을 메뉴로도 바꿀 수 있고 키보드로도 바꿀 수 있을 경우 메뉴로 색상을 바꾸었을 때는 OnClick 이벤트에서 체크 표시를 제대로 변경해 주지만 키보드로 색상을 바꾸었을 때는 체크 표시가 잘못 표시될 수도 있다.

두 번째 방법인 상위 메뉴 항목의 OnClick 이벤트를 사용하면 이런 불일치를 제거할 수 있는데 상위 메뉴의 OnClick 이벤트는 메뉴가 열리기 직전에 발생하기 때문에 여기에 코드를 작성하면 항상 메뉴가 열리기 전에 체크 표시를 제 위치에 옮겨주게 된다. Text 아래의 세 항목에 체크 표시를 달아주기 위해 Text 메뉴 항목의 OnClick 이벤트 핸들러를 작성해 보자. 메뉴 디자이너에서 Text 를 더블클릭한 후 다음 코드를 작성한다.

```
procedure TForm1.Text1Click(Sender: TObject);
begin
```

```
  Kim1.Checked:=False;
  Lee1.Checked:=False;
  Park1.Checked:=False;
  if Edit1.Text='Kim' then Kim1.Checked:=True;
  if Edit1.Text='Lee' then Lee1.Checked:=True;
  if Edit1.Text='Park' then Park1.Checked:=True;
end;
```

세 메뉴 항목의 Checked 속성을 모두 False 로 바꾼 후 Edit1 의 Text 속성을 직접 조사하여 선택된 메뉴 항목의 Checked 속성만 True 로 바꾸어 주었다. 이제 예제를 실행시키면 Text 메뉴가 펼쳐질 때마다 선택된 메뉴 항목 옆에 체크 표시를 달아 줄 것이다.

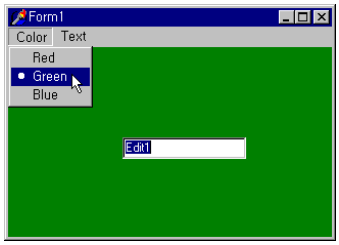



체크 표시를 변경하는 세 번째 방법은 메뉴 항목의 RadioItem 속성과 GroupIndex 속성을 사용하는 것이다. Color 메뉴 아래의 세 항목들은 서로 배타적인 선택을 나타내며 셋 중 하나만 선택될 수 있다. 이럴 때는 세 항목의 GroupIndex 를 같은 값(일단 1 로 설정하자)으로 설정하여 세 항목이 한 그룹임을 지정하고 RadioItem 속성을 True 로 바꾸어 이 항목들이 상호 배타적이라는 것을 지정해 준다. RadioItem 속성이 True 로 지정된 메뉴 항목은 같은 그룹에서 오직 하나의 메뉴 항목만 선택될 수 있으며 한 항목이 선택되면 그룹내의 다른 항목은 자동으로 선택이 취소된다. 이 두 속성을 조정한 후에 코드를 다음과 같이 수정해 보자.

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  Form1.Color:=clRed;
  Red1.Checked:=True;
end;

procedure TForm1.Green1Click(Sender: TObject);
begin
  Form1.Color:=clGreen;
  Green1.Checked:=True;
```

```
end;

procedure TForm1.Blue1Click(Sender: TObject);
begin
  Form1.Color:=clBlue;
  Blue1.Checked:=True;
end;
```

세 메뉴 항목의 GroupIndex 속성이 모두 1 로 설정되어 있으며 RadioItem 속성이 True 로 되어 있으므로 셋 중 하나만 배타적으로 선택될 수 있다. 그래서 각 항목이 선택될 때 한 항목의 Checked 속성을 True 로 변경해 주면 같은 그룹에 속한 나머지 항목의 Checked 속성은 자동으로 False 가 된다. 또한 라디오 그룹은 옆에 체크 표시가 나타나지 않고 둥그런 원 모양으로 이 옵션이 선택되었음이 표시된다. 이런 식으로 체크 상태가 표시되는 메뉴는 탐색기에서도 볼 수 있다.




여기까지 실습을 마친 사람은 Menu1 예제의 Red, Green, Blue 메뉴 항목에 Ctrl+R, Ctrl+G, Ctrl+B 등의 쇼트컷을 달아 보도록 하자. 오브젝트 인스펙터에서 ShortCut 속성만 변경해 주면 된다.


① 메인 메뉴 컴포넌트를 더블클릭하여 메뉴 디자이너를 연다.
② Caption 속성을 입력하면 메뉴 항목 컴포넌트가 생성된다.
③ 메뉴 디자이너에서 Ins, Del키로 메뉴 항목을 추가 삭제하며 드래그하여 위치를 이동시킨다.

# 5-3 팝업 메뉴

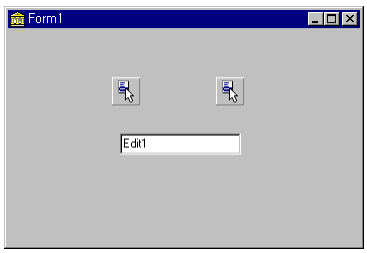
메인 메뉴는 항상 폼의 상단에 위치하고 있기 때문에 언제든지 사용할 수 있으며 모든 명령을 포함하고 있기 때문에 프로그램의 기능을 일목요연하게 알 수 있다는 장점이 있지만 메뉴의 수가 많을 경우 찾아서 사용하기 불편하며 마우스가 이동해야 할 거리가 멀어 비효율적이기도 하다. 이런 메인 메뉴의 단점을 잘 보완해 주는 것이 팝업 메뉴이다.

## 가. 팝업 메뉴의 제작

팝업 메뉴는 버튼, 폼, 에디트 박스 등의 개별적인 컨트롤이 가지는 메뉴이며 컨트롤에 따라 꼭 필요한 명령만을 가지므로 신속하게 작업을 진행할 수 있다. 앞에서 만들어 보았던 메인 메뉴 예제와 똑같은 예제를 만들되 여기서는 메인 메뉴를 없애고 팝업 메뉴로 대체해 보도록 하자.

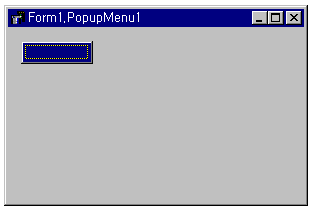
5jang
popup1

우선 다음과 같이 폼 상에 컴포넌트를 배치한다. 에디트 컴포넌트 하나와 팝업 메뉴 컴포넌트 두 개가 필요하다. 메인 메뉴는 하나의 폼에 보통 하나만 있지만 팝업 메뉴는 필요한만큼 얼마든지 만들어 사용할 수 있다.

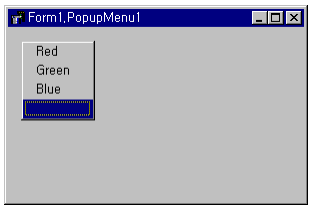



각 컴포넌트의 속성은 모두 디폴트를 사용하기로 한다. 팝업 메뉴를 디자인하는 방법은 메인 메뉴를 디자인하는 방법과 동일하다. 팝업 메뉴 컴포넌트를 더블클릭하여 메뉴 디자이너를 열고 메뉴 디자이너에서 메뉴 항목들을 정의해 준다. 우선 왼쪽에 있는 팝업 메뉴 컴포넌트(PopupMenu1)를 더블클릭하여 메뉴 디자이너를 연다.

그림
팝업 메뉴의 메뉴 디자이너

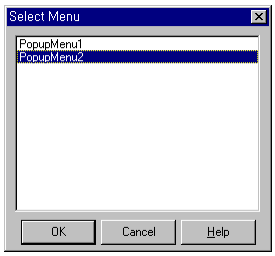



팝업 메뉴 디자이너는 메인 메뉴 디자이너와는 모양이 조금 다르다. 폼의 상단에 나타나는 것이 아니라 화면 상의 어디든지 불쑥 나타나므로 메뉴 바가 없으며 메뉴 리스트만 있다. 메뉴 항목을 만드는 방법은 메인 메뉴와 완전히 동일하다. 다음과 같이 Red, Green, Blue 세 개의 메뉴 항목을 만들도록 하자.

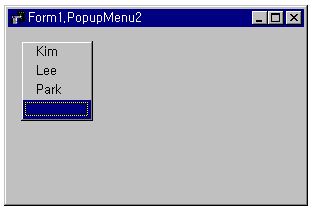



다음으로 오른쪽에 있는 PopupMenu2의 메뉴를 디자인해 보자. 메뉴 디자이너에 다른 메뉴를 불러오려면 폼에서 다른 메뉴 컴포넌트를 더블클릭하면 된다. 폼의 좌측에 있는 PopupMenu2를 더블클릭해 보아라. 아니면 메뉴 디자이너의 스피드 메뉴에서 Select Menu 항목을 선택하여 메뉴 선택 대화상자에서 편집할 메뉴를 선택해도 된다.

그림
메뉴 선택 대화상자




폼이 열려있는 상태라면 PopupMenu2를 더블클릭하는 편이 더 간편하다. PopupMenu2를 더블클릭하여 메뉴 디자이너를 열고 다음과 같이 세 개의 메뉴 항목을 만든다.




다음은 각 메뉴 항목의 코드를 작성한다. 메뉴 항목의 이벤트 핸들러를 작성하는 방법도 메인 메뉴의 경우와 동일하지만 한 가지 차이점은 반드시 메뉴 디자이너를 통해야만 코드 에디터를 열 수 있다는 점이다. 메인 메뉴는 메뉴가 폼 상에도 나타나므로 폼을 통해서 이벤트 핸들러를 만들 수 있지만 팝업 메뉴는 디자인시에 폼에 나타나지 않기 때문이다. 사용하는 코드는 메인 메뉴 예제에서 사용한 코드와 완전히 동일하다. 글자 하나 안틀리고 똑같은 코드이다.

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  Form1.Color:=clRed;
end;

procedure TForm1.Green1Click(Sender: TObject);
begin
  Form1.Color:=clGreen
end;

procedure TForm1.Blue1Click(Sender: TObject);
begin
  Form1.Color:=clBlue
end;

procedure TForm1.Kim1Click(Sender: TObject);
begin
  Edit1.Text:='Kim';
end;

procedure TForm1.Lee1Click(Sender: TObject);
begin
  Edit1.Text:='Lee';
end;

procedure TForm1.Park1Click(Sender: TObject);
begin
  Edit1.Text:='Park';
```

```
end;
```

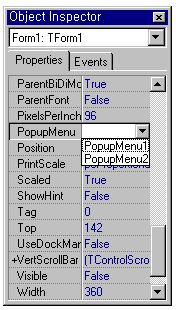
메인 메뉴 예제에서는 메뉴를 만들고 코드를 작성하면 곧바로 실행할 수 있었지만 팝업 메뉴에서는 한 가지를 더해 주어야 한다. 어떤 팝업 메뉴를 어떤 컨트롤이 사용할 것인가를 지정해 주는 일이 남아 있다. 즉 어떤 컨트롤에서 마우스의 오른쪽 버튼을 누를 때 팝업 메뉴가 나타날 것인가를 정해주는 것이다. 컨트롤과 팝업 메뉴를 연결하는 방법은 팝업 메뉴를 사용할 컨트롤의 PopupMenu 속성에 팝업 메뉴의 이름을 지정하는 것이다.

폼, 버튼, 에디트, 레이블, 리스트 박스, 콤보 박스 등등 모든 컨트롤에 PopupMenu 속성이 있어 각각의 컨트롤이 개별적인 팝업 메뉴를 가질 수 있다. Red, Green, Blue 등의 메뉴 항목을 가진 PopupMenu1은 폼의 배경 색상을 변경할 목적으로 만들어졌으므로 Form1의 PopupMenu 속성에 지정하도록 한다.

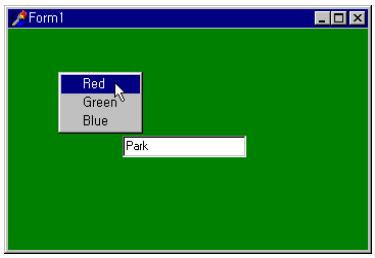
폼을 선택한 후 오브젝트 인스펙터에서 PopupMenu 속성을 마우스로 선택해 보자. 속성값 란의 버튼을 클릭하여 리스트 박스를 열면 사용 가능한 팝업 메뉴의 목록이 나타나는데 이 예제의 경우 두 개의 팝업 메뉴가 나타날 것이다. 여기서 PopupMenu1을 선택하도록 하자.

**그림**

각 컴포넌트의 PopupMenu 속성을 사용하여 컴포넌트에 팝업 메뉴를 연결해 준다.




Form1의 PopupMenu 속성이 PopupMenu1으로 지정되었으므로 실행중에 폼 상의 어디에서나 마우스의 오른쪽 버튼을 누르면 PopupMenu1이 나타나게 된다. 같은 방법으로 Edit1에 PopupMenu2를 연결해 두면 Edit1에서 마우스의 오른쪽 버튼을 누를 때 PopupMenu2가 나타나게 될 것이다. 컨트롤과 팝업 메뉴를 연결하였으면 프로그램이 완전하게 완성되었다. 실행을 시켜 보자. 실행중의 모양은 다음과 같다. 폼에서 마우스의 오른쪽 버튼을 누르면 폼의 색상을 변경할 수 있는 팝업 메뉴가 열리고 에디트 위에서 마우스의 오른쪽 버튼을 누르면 에디트 박스의 문자열을 변경할 수 있는 팝업 메뉴가 열린다.




# 나. AutoPopup 속성

5jang
popup2

팝업 메뉴 컴포넌트의 속성은 대부분 평이하므로 레퍼런스를 참조하도록 하되 AutoPopup 속성 한 가지만 여기서 소개하기로 한다. AutoPopup 속성의 디폴트값은 True이며 따라서 오른쪽 버튼을 누르기만 하면 즉시 팝업 메뉴가 나타난다. AutoPopup 속성이 False일 경우는 오른쪽 버튼을 눌러도 팝업 메뉴가 나타나지 않으며 팝업 메뉴의 Popup 메소드를 사용하여 강제로 나타나게 하여야 한다.

팝업 메뉴는 디폴트로 오른쪽 버튼 클릭에 의해 나타나도록 되어 있는데 만약 키보드나 다른 특수한 방법으로 팝업 메뉴가 나타나도록 하고 싶다면 이 속성을 사용하도록 한다. AutoPopup 속성을 False로 설정한 후 팝업 메뉴를 나타내고 싶은 시점, 예를 들어 키보드 입력시 등에서 Popup 메소드를 사용하여 메뉴를 나타내면 된다.

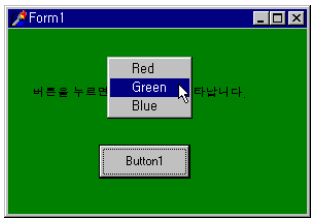
배포 CD의 POPUP2 예제는 화면 상의 버튼을 눌러 팝업 메뉴가 나타나도록 하였다. 그래서 버튼의 OnClick 이벤트에서 팝업 메뉴의 Popup 메소드를 호출하도록 코드가 작성되었다. 응용하기에 따라서는 어떤 상황에서든 팝업 메뉴를 호출하도록 할 수도 있다. 코드는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  PopupMenu1.Popup(Left+100,Top+50);
end;
```

Popup 메소드는 팝업 메뉴가 나타날 화면 상의 좌표를 인수로 사용하며 이 좌표는 화면 전체를 기준으로 한 좌표이므로 폼의 Left, Top 좌표를 더해 주어 폼안에 팝업이 나타나도록 해 주었다. 마우스 오른쪽 버튼에 의해 팝업 메뉴가 나타나는 것이 아니므로 팝업 메뉴가 다른 컴포넌트와 연결될 필요는 없다.

그림

Popup 메소드를 사용하여 버튼을 누를 때 팝업 메뉴가 나타나도록 한다.




# 다. OnPopup 이벤트

델파이에서 메뉴 제작에 사용되는 컴포넌트에는 세 가지 종류가 있다. 우선 컴포넌트 팔레트에 있는 MainMenu 컴포넌트(  )와 PopupMenu 컴포넌트(  )가 있고 컴포넌트 팔레트에는 없지만 메뉴 디자이너에 의해 만들어지는 MenuItem 컴포넌트가 있다. 메인 메뉴 컴포넌트는 메뉴 전체를 대표할 뿐이며 특별한 속성이 없고 항상 폼 상단에 위치하고 있으므로 이벤트를 전혀 가지지 않는다. 즉 메인 메뉴 자체는 어떤 변화가 있을 수 없다는 얘기다. 다만 메뉴 항목 컴포넌트만 사용자가 선택할 경우 OnClick 이벤트를 가진다.

팝업 메뉴 컴포넌트는 메뉴 바에 없으며 사용자가 마우스의 오른쪽 버튼을 누를 때 나타나므로 팝업 메뉴가 화면에 나타날 때 OnPopup 이벤트를 가진다. 팝업 메뉴 컴포넌트를 더블클릭하면 메뉴 디자이너가 열리도록 되어 있으므로 OnPopup 이벤트의 핸들러는 오브젝트 인스펙터를 통해야만 작성할 수 있다. 즉 폼에서 팝업 메뉴 컴포넌트를 더블클릭한다고 해서 코드 에디터에 OnPopup 이벤트 핸들러가 생성되지는 않는다는 얘기다. 팝업 메뉴가 열릴 때 이 이벤트가 발생하므로 팝업 메뉴 내의 메뉴 항목 속성(Enabled, Visible)을 다시 설정해야 할 필요가 있을 때 이 이벤트를 사용한다.

Pupup1 예제를 다시 열어 각 팝업 메뉴의 OnPopup 이벤트를 다음과 같이 작성해 보자. 팝업 메뉴 컴포넌트를 선택한 후 오브젝트 인스펙터의 Events 페이지에서 OnPopup을 더블클릭해야 이벤트 핸들러를 만들 수 있다.

```
procedure TForm1.PopupMenu1Popup(Sender: TObject);
begin
  Red1.Checked:=False;
  Green1.Checked:=False;
  Blue1.Checked:=False;
  if Form1.Color=clRed then Red1.Checked:=True;
  if Form1.Color=clGreen then Green1.Checked:=True;
  if Form1.Color=clBlue then Blue1.Checked:=True;
end;
```

```
procedure TForm1.PopupMenu2Popup(Sender: TObject);
begin
  Kim1.Checked:=False;
  Lee1.Checked:=False;
  Park1.Checked:=False;
  if Edit1.Text='Kim' then Kim1.Checked:=True;
  if Edit1.Text='Lee' then Lee1.Checked:=True;
  if Edit1.Text='Park' then Park1.Checked:=True;
end;
```

현재 선택된 옵션 상태에 따라 적절한 메뉴 항목에 체크 표시를 출력하도록 하였다. 이 코드는 Menu1의 예제에서 사용한 코드와 완전히 동일하다.

# 5-4 메뉴 조작

여기까지 메뉴를 작성하는 아주 기본적인 방법들에 대해서 알아보았다. 기본을 알았으니 기본을 바탕으로 하여 메뉴를 잘 다룰 수 있는 고급적인 방법들에 대해 알아보자. 메뉴를 좀 더 예쁘게, 편리하게 만들고 사용할 수 있는 방법들에 대해 알게 될 것이다.

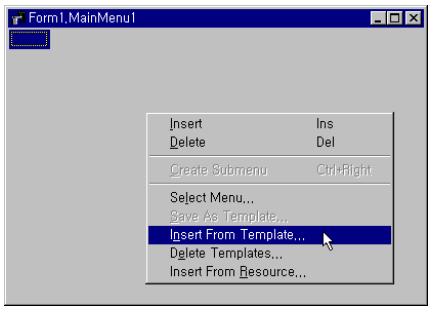
## 가. 메뉴 템플릿

윈도우즈용 프로그램의 메뉴들은 대체로 모양이 비슷비슷하다. File 메뉴나 Edit 메뉴, Help 등의 메뉴는 웬만한 프로그램에는 다 있는 편이며 File 메뉴 안에는 Open File, Save File, Print, Exit 등의 메뉴 항목들이 있게 마련이다. 프로그램이 하는 일에 공통점이 있어서이기도 하지만 사용자에게 일관된 인터페이스를 제공하기 위해 일부러 메뉴를 비슷하게 만들어 메뉴를 쉽게 외울 수 있도록 하기 위함이다. 이러한 메뉴 작성 관행은 강제적으로 지켜야 하는 것은 아니지만 대부분의 프로그래머들이 따르고 있고 사용자들도 이런 관행을 따르는 프로그램을 기대한다. 그래서 윈도우즈 프로그램들의 메뉴가 다 비슷한 모양을 가지는 것이다.

델파이가 제공하는 메뉴 디자이너가 아무리 편리해도 이런 비슷비슷한 메뉴를 매번 만드는 일은 분명이 귀찮은 일 중의 하나이다. 이런 번거로움을 조금이라도 덜어주기 위해 델파이는 많이 사용되는 메뉴들을 미리 만들어서 제공해 주는데 이를 메뉴 템플릿(Menu Template)이라고 한다. 메뉴 템플릿은 BIN 디렉토리에 DELPHI32.DMT라는 파일로 저장되어 있어 언제든지 읽어와 자신의 프로그램에 포함시킬 수 있으며 또한 자신이 만든 메뉴를 템플릿에 저장해 두었다가 다음에 사용할 수도 있다. 메뉴 템플릿을 한마디로 쉽게 정의하자면 메뉴 저장소라고 할 수 있다.

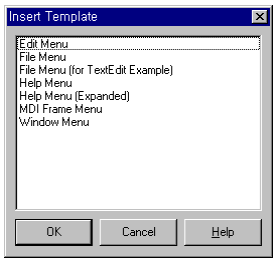
우선 메뉴 템플릿에 있는 메뉴를 읽어와 자신의 프로그램에 사용하는 방법에 대해 알아 보자. 새로운 프로젝트를 시작한 후 메인 메뉴 컴포넌트를 폼에 배치하고  더블클릭하여 메뉴 디자이너를 연다. 메뉴 템플릿을 사용하기 위해서는 메뉴 디자이너의 스피드 메뉴를 사용한다. 메뉴 디자이너 위에서 오른쪽 버튼을 누르면 다음과 같은 스피드 메뉴가 나타날 것이다.

그림
메뉴 디자이너의 스피드 메뉴



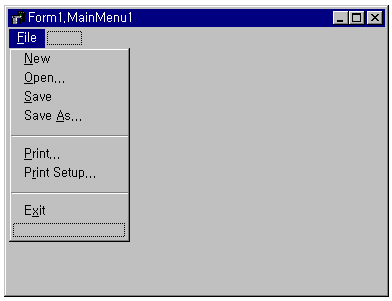

여기서 Insert From Template... 항목을 선택해 보자. 다음과 같이 템플릿에 저장되어 있는 메뉴 항목들의 목록이 나타난다.

그림
이미 작성된 메뉴를 선택하는 메뉴 템플리트




File, Edit, Window, Help 등 가장 일반적인 메뉴들이 있다. 여기서 자신이 삽입하고자 하는 메뉴 항목을 선택한다. File 메뉴를 선택해 보자. 메뉴 목록이 닫히면서 메뉴 디자이너에 File 메뉴가 삽입된다.

그림
메뉴 템플리트에서 불러온 메뉴



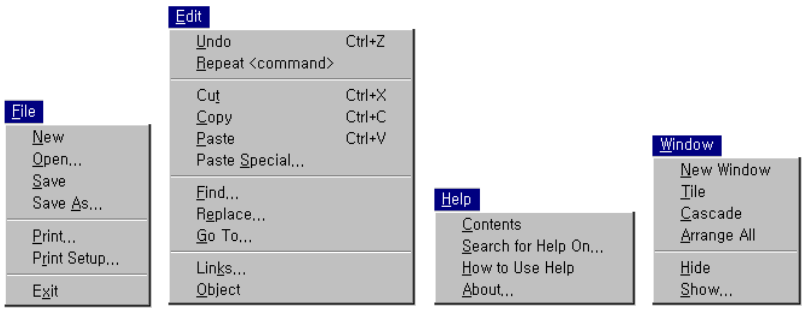

삽입된 File 메뉴에는 New, Open, Save 등 File 메뉴에는 당연히 있어야 할 메뉴 항목들이 있으며 각 메뉴 항목에는 캡션, 단축키는 물론이고 어떤 것들은 쇼트컷까지 이미 정의되어 있다. 하나의 메뉴 항목을 삽입한 후에도 템플릿 목록을 열어 계속 다른 메뉴를 삽입할 수 있다. 삽입되는 위치는 반전 막대가 있는 위치이므로 템플릿에서 메뉴를 삽입하기 전에 삽입할 위치를 지정해 주어야 할 필요가 있다. 예를 들어 Edit 메뉴를 추가로 더 삽입하고자 한다면 File 옆의 빈

칸에 반전 막대를 둔 상태에서 Edit 메뉴를 불러와야 한다.

템플릿에 있는 메뉴는 가장 일반적으로 많이 사용되는 메뉴이지만 아무리 그래도 자기가 만들고자 하는 프로그램에 꼭 맞지는 않을 것이다. 일단 템플릿에서 가져온 메뉴는 메뉴 항목을 삭제하거나 삽입하거나 위치를 옮기거나 이름을 바꾸는 것이 자유롭다. 그래서 대충 비슷한 메뉴를 템플릿에서 읽어온 후 자기 편한대로 수정하도록 한다. 메뉴 템플릿에 미리 저장되어 있는 메뉴들은 다음과 같다.

**그림**

메뉴 템플리트에 저장되어 있는 메뉴




다음은 자기가 만든 메뉴를 템플릿에 저장하는 방법을 알아보자. 우선 빈 메뉴 디자이너를 열고 템플릿에 저장할 메뉴를 하나 만들어 둔다. 메뉴를 템플릿으로 저장할 때도 읽어올 때와 마찬가지로 스피드 메뉴를 사용한다. 메뉴 디자이너에서 오른쪽 버튼을 누른 후 Save As Template...를 선택하면 다음과 같이 템플릿의 메뉴 목록이 나타난다.


위쪽에 있는 에디트 박스에 저장할 템플릿의 이름을 입력해 주면 그 이름으로 자신이 만든 메뉴가 템플릿에 저장된다. 이렇게 메뉴를 한 번 만들어 템플릿에 저장해 두면 다음에 힘들이지 않고 그 메뉴를 계속 사용할 수 있다.

메뉴를 템플릿으로 저장할 경우 델파이는 메뉴 항목의 Name 속성은 저장해 주지 않는다. 왜냐하면 메뉴 항목의 Name 속성은 프로그램 전체를 통틀어 유일

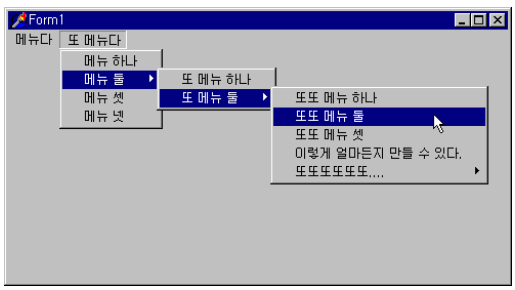
해야 하는데 템플릿에 있는 메뉴 항목의 Name 속성이 다른 메뉴 항목의 Name 속성과 충돌할 위험이 있기 때문이다. 대신 델파이는 템플릿에서 메뉴가 삽입될 때 기존에 있는 메뉴 항목의 이름과 충돌하지 않도록 메뉴 항목의 Name 속성을 재작성해 준다.

비슷한 이유로 메뉴가 템플릿에 저장될 때 OnClick 이벤트 핸들러도 저장되지 않는다. 프로그램마다 상황이 다르므로 코드는 저장해 봐야 그대로 재사용할 수가 없기 때문이다. 템플릿은 어디까지나 메뉴의 모양을 저장하는 것이지 메뉴의 모든 기능을 그대로 저장해 두는 것은 아니다.

# 나. 하위 메뉴 작성

하위 메뉴(Sub Menu)란 메뉴 리스트 안의 메뉴 항목 안에 또 다른 메뉴 리스트가 포함되어 있는 경우를 말한다. 하위 메뉴가 있을 경우 메뉴 항목 옆에 ▶기호가 표시되며 메뉴 항목 선택시 또 다른 메뉴 리스트가 펼쳐진다. 다음과 같이 말이다.

그림
하위 메뉴 작성의 예

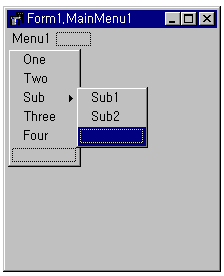



하위 메뉴를 사용하면 당장 필요하지 않은 메뉴 항목은 나타나지 않으므로 화면의 공간이 절약된다는 이점이 있다. 모든 메뉴 항목이 한꺼번에 나타나면 메뉴 리스트가 길어져서 보기에도 좋지 않고 원하는 항목을 찾기도 어려워 사용하기에도 불편할 것이다. 게다가 하위 메뉴는 논리적으로 비슷한 용도를 가진 메뉴 항목들을 한 곳에 깔끔하게 모아 둘 수 있어서 구조적으로도 효율적인 메뉴를 작성할 수 있다. 하위 메뉴를 만드는 방법에는 두 가지가 있다.

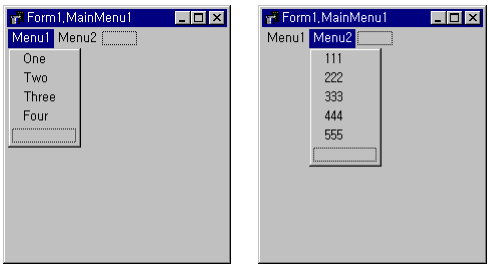
첫째, 메뉴를 디자인할 때 직접 만드는 방법이다. 하위 메뉴를 만들 메뉴 항목을 선택한 후 Ctrl 키와 우측 커서 이동키(→)를 누른다. 그러면 즉시 우측으로 새로운 메뉴를 만들 수 있는 빈칸이 생긴다.

그림
하위 메뉴 작성

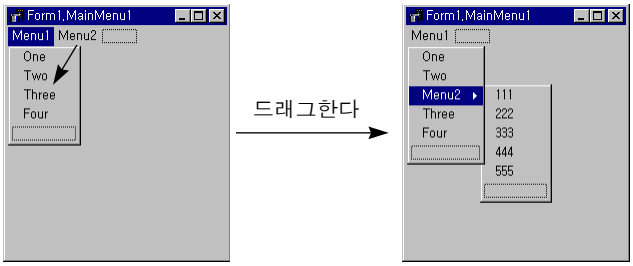

Ctrl - →를 누르는 대신 메뉴 디자이너의 스피드 메뉴에서 Create Submenu를 선택해도 결과는 같다. 일단 서브 메뉴가 만들어지고 서브 메뉴 안에 빈칸이 하나 생기면 이 빈칸을 발판으로 하여 서브 메뉴 안에 메뉴 항목을 채워나가도록 한다. 일단 두 개의 서브 메뉴 항목을 만들어 보았다. 만든 후의 모양은 다음과 같다.




하위 메뉴를 만드는 두 번째 방법으로 메뉴를 만든 후 기존에 있던 메뉴를 하위 메뉴로 이동시키는 방법(demoting)이 있다. 메뉴 항목을 드래그하여 옮길 경우 하위 메뉴가 따라서 이동된다는 것을 이용한 것이다. 실습을 위해 다음과 같이 Menu1과 Menu2를 만들도록 하자.




Menu2를 드래그하여 Menu1의 Three 메뉴 항목에 떨어뜨리면 Menu2가 Three 항목 앞으로 이동하게 되며 Menu2안의 메뉴 항목들은 하위 메뉴에 포함된다.




델파이는 다단계의 하위 메뉴를 지원한다. 즉 하위 메뉴 안에 또 하위 메뉴를 만들 수 있다는 얘기다. 그러나 하위 메뉴를 너무 깊게 만드는 것은 프로그램을 지저분하게 보이도록 하는데 지대한 공헌을 하므로 적당한 수준에서 자재를 하는 편이 좋다. 웬만큼 규모가 큰 프로그램이라도 1차의 하위 메뉴만 만들며 아주 드물게 2차 하위 메뉴까지 사용하기도 한다. 델파이에서는 File/Reopen 등 네 개의 메뉴가 일차까지만 하위 메뉴를 사용하고 있다.

3차, 4차까지 하위 메뉴를 만들면 사용자는 원하는 메뉴 항목을 찾기 위해 숨바꼭질을 해야 할 것이다. 메인 메뉴뿐만 아니라 팝업 메뉴에도 똑같은 방법으로 하위 메뉴를 만들 수 있다. 팝업 메뉴의 경우도 1차 이상의 하위 메뉴는 만들지 않는 것이 좋다.

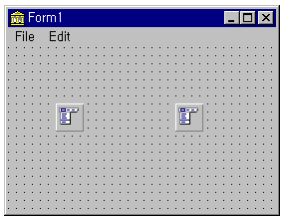
# 다. 실행중 메뉴 교체

하나의 폼은 반드시 하나의 메인 메뉴만을 가질 수 있으며 두 개의 메인 메뉴를 동시에 가질 수는 없다. 팝업 메뉴는 필요에 따라 얼마든지 가질 수 있지만 말이다. 그러나 메인 메뉴가 하나밖에 있을 수 없다고 해서 메인 메뉴 컴포넌트를 하나밖에 가질 수 없는 것은 아니다. 메인 메뉴 컴포넌트도 팝업 메뉴 컴포넌트처럼 여러 개를 만들어 놓을 수 있다. 다만 실행중에는 하나의 메인 메뉴만 선택하여 사용할 수 있을 뿐이다.

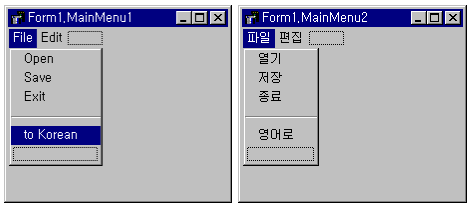
상황에 따라 메뉴가 아주 달라질 경우, 두 개의 메인 메뉴를 미리 만들어 두고 그때 그때 상황에 따라 메인 메뉴를 통째로 교체할 수 있다. 메인 메뉴가 폼에 연결되는 것은 폼의 Menu 속성을 통해서이므로 실행중에 폼의 Menu 속성을 변경해 주면 메뉴를 교체해 가며 사용할 수 있다.


5jang
ChgMenu

메인 메뉴를 교체하는 예제를 만들어 보자. 이 예제는 영문으로 된 메뉴 하나와 한글로 된 메뉴를 가지며 사용자가 두 메뉴를 실행중에 변경할 수 있도록 하였다. 새 프로젝트를 시작한 후 폼에 메인 메뉴 컴포넌트를 두 개 배치한다.




오른쪽 메인 메뉴(MainMenu1:먼저 배치한 메인 메뉴 컴포넌트)에는 영문으로 된 메뉴 항목들을 만들어 주고 왼쪽의 메인 메뉴에는 한글로 된 메뉴 항목들을 만들어 주었다.




File 메뉴의 아래쪽에는 to Korean, 영어로 등의 메뉴 항목을 배치하여 실행중에 다른 메뉴로 교체할 수 있도록 하였다. 이 상태에서 프로그램을 실행시키면 영문으로 된 메뉴가 나타날 것이다. 델파이는 시간상으로 먼저 만들어진 메인 메뉴를 폼의 기본 메뉴로 지정한다. 즉 메인 메뉴 컴포넌트를 배치하는 순간에 폼의 Menu 속성을 이 컴포넌트로 바꾸어 준다. 이 실습에서 영문 메뉴로 작성된 MainMenu1이 먼저 폼에 놓여졌기 때문에 영문 메뉴가 디폴트로 사용되었다.

실행중에 메뉴를 교체하도록 하기 위해 to Korean 메뉴 항목과 영어로 메뉴 항목의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.N7Click(Sender: TObject);
begin
  Menu:=MainMenu1;
end;

procedure TForm1.toKorean1Click(Sender: TObject);
begin
  Menu:=MainMenu2;
end;
```

폼의 Menu 속성을 상대편 메인 메뉴로 변경하도록 하였다. 그래서 영문 메뉴 상태에서 to Korean 항목을 선택하면 한글 메뉴(MainMenu2)로 변경되며

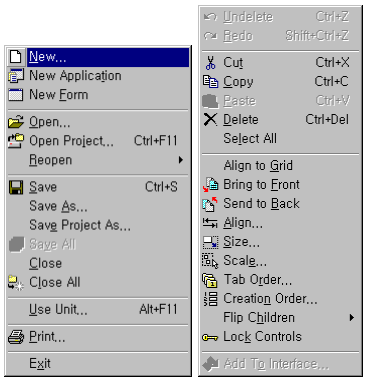
한글 메뉴 상태에서 영어로 항목을 선택하면 영문 메뉴(MainMenu1)로 변경되는 것이다.

이런 식의 메뉴 교체는 두 가지 메뉴뿐만 아니라 여러 개의 메뉴에도 적용된다. 한국어, 영어, 일본어, 중국어 등의 메뉴를 작성해 놓고 사용자가 교체할 수 있도록 한다거나 아니면 프로그램이 실행될 때 국가 코드를 조사해 적절한 메뉴를 사용하도록 한다면 다국어 버전의 프로그램을 따로 만들 필요없이 국제적으로 팔아먹을 수 있는 프로그램을 작성할 수 있을 것이다. 물론 이때 각 메뉴의 항목들은 Caption 만 다르고 기능은 같아야 할 것이다.

# 라. 메뉴에 이미지 사용하기

윈도우즈의 표준 메뉴에는 원래 이미지라는 것이 없었으며 문자열로만 메뉴 항목을 표시했다. 그런데 최근에 와서는 메뉴의 모양도 많이 바뀌어 메뉴 항목의 왼쪽에 조그만 이미지를 그릴 수 있도록 되었다. 델파이의 메뉴에도 조그만 이미지들이 있는데 이는 메뉴 항목이 어떤 툴 버튼과 같은 기능을 가지는지를 쉽게 알 수 있게 해준다.

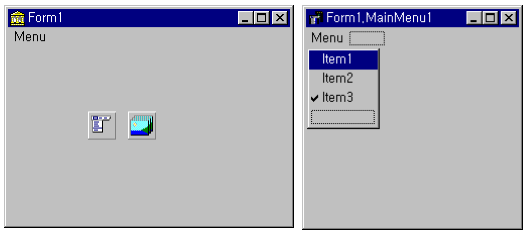
그림
메뉴 항목 왼쪽에 이미지가 나타난다.




메뉴 항목에 이미지를 놓으려면 이미지 리스트 컴포넌트를 사용하면 된다. 이미지 리스트는 Win32 페이지에 있으며 일종의 이미지 배열이라고 생각할 수 있다. 동일한 크기의 이미지를 가지며 메뉴 항목이나 트리, 리스트 컨트롤에 이미지를 공급해 주는 역할을 한다.

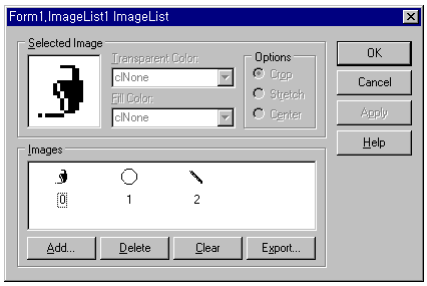

5jang
ImgItem

예제를 만들면서 이미지 리스트의 사용 방법을 익혀 보자. 새 프로젝트를 시작하고 메인 메뉴와 이미지 리스트 컴포넌트를 폼에 배치한다. 메뉴에는 Menu 아래 Item1, Item2, Item3 메뉴 항목을 만들어 놓았으며 이 항목들 옆에 이미지를 출력해 볼 것이다.



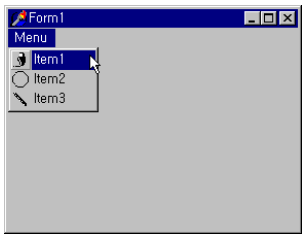

이미지를 불러 오기 위해 이미지 리스트 컴포넌트를 더블클릭하면 이미지 리스트 편집기가 나타난다. 이 편집기는 직접 이미지를 그릴 수는 없으며 이미 만들어져 있는 이미지(BMP 파일 또는 아이콘)들을 불러와 배열로 만들 뿐이다. Add 버튼을 눌러 이미지들을 불러온다.

그림
이미지 리스트 편집기




이미지는 별도의 이미지 에디터로 만들어서 사용해야 하는데 너무 번거로우므로 델파이 예제에 있는 이미지를 훔쳐와 사용해 보자. Demos₩Doc₩Graphex 디렉토리를 보면 몇 개의 BMP 파일이 있으므로 이 파일들을 불러와 보자. Brush.bmp, Ellipse,bmp, Pen.bmp 세 개의 이미지를 불러왔다. 그러면 이 이미지들은 배열을 이루며 각각 첨자 0, 1, 2 가 된다.

메뉴에 이미지를 공급할 때는 두 가지 속성이 사용된다. 우선 메인 메뉴 컴포넌트의 Images 속성에는 이미지 배열에 해당되는 이미지 리스트 컴포넌트를 지정해 준다. MainMenu1.Images 에 ImageList1 컴포넌트를 지정하여 이 메뉴에서 사용할 이미지는 ImageList1 에 있다는 것을 알려준다. 그리고 각 메뉴 항목의 ImageIndex 속성에 이미지의 첨자를 대입해 준다. Item1 은 0 를, Item2 는 1 을, Item3 은 2 를 각각 대입해 준다. 이제 프로그램을 실행시켜 보자. 메뉴 항목 옆에 이미지가 나타날 것이다.




이미지 리스트의 이미지들이 메뉴 항목과 어떻게 대응되었는지를 보기 바란다. 이미지 리스트는 메뉴뿐만 아니라 여러 가지 용도로 사용되는데 더 자세한 사항은 관련 부분에서 다시 논할 것이다. 이미지를 직접 만들 때는 이미지 에디터를 사용하면 된다. 이 역시 관련 부분에서 논한다.


① 메뉴 템플릿:자주 사용되는 메뉴의 형태를 저장해 놓은 것
② Ctrl →키를 사용하여 하위 메뉴를 작성한다.
③ Enabled, Checked 속성을 사용하여 실행중에 메뉴 항목을 조작한다.

# 5-5 이벤트 연결

## 가. 이벤트 핸들러 연결

앞에서 메뉴의 Red, Green, Blue 항목으로 폼의 색상을 바꾸는 예제를 만들어 보았다. 메뉴란 원래 프로그램이 가지는 모든 명령들을 일목 요연하게 정리해 놓기 위해 많이 사용되며 메뉴 항목에서 많이 쓰이는 항목은 별도로 툴바나 스피드 버튼을 사용하는 경우가 많다. 지금 여러분들이 쓰고 있는 델파이만 해도 File/Open Project... 항목과  버튼, Run/Run 항목과  버튼은 완전히 같은 기능을 한다.

메뉴와 같은 기능을 가지는 버튼들이 있을 경우는 코드를 어떻게 작성할까? Red, Green, Blue 메뉴 항목과 똑같은 기능을 하는 BtnRed, BtnGreen, BtnBlue 세 개의 버튼을 만들어 보고자 한다. 메뉴의 Red 항목과 BtnRed 버튼의 기능이 완전히 같으므로 똑같은 코드를 사용해도 되며 그렇게 하는 편이 좋을 것 같다.

그런데 델파이는 메뉴의 Red 항목을 더블클릭할 때와 BtnRed를 더블클릭할 때 각각 다른 이벤트 핸들러를 만들어 준다. 왜냐하면 델파이는 아직까지 Red와 BtnRed가 같은 기능을 한다는 것을 모르기 때문이다. 그래서 두 컴포넌트가 같은 코드를 사용한다는 것을 사용자가 직접 알려주어야 하는데 이 때 사용되는 기법이 이벤트 핸들러 연결이다.

## 나. 이벤트 핸들러의 이름

이벤트 핸들러를 연결하여 하나의 코드를 두 개 이상의 컴포넌트가 사용하려면 먼저 이벤트 핸들러의 이름 붙이는 방법과 오브젝트 인스펙터의 정확한 사용법을 익혀야 한다. 아주 중요한 내용이므로 약간만 정신을 차리고 읽도록 하자. 폼 위에 컴포넌트가 있을 때 더블클릭하면 그 컴포넌트에서 가장 자주 발생하는 이벤트에 대한 코드를 작성할 수 있도록 코드 에디터가 열린다. 예를 들어 버튼의 경우는 OnClick 이벤트가 가장 자주 발생하므로 폼에서 버튼을 더블클릭하면 OnClick 이벤트 핸들러가 즉각 코드 에디터에 작성된다. 이때 그 컴포넌트에 가장 자주 발생하는 이벤트를 디폴트 이벤트라고 하는데 컴포넌트별로 다르다.

델파이는 이벤트 핸들러에 디폴트 이름을 붙여 주는데 이 디폴트 핸들러 이름을 잠시 살펴보자.

**표**

델파이가 만들어 주는 이벤트 핸들러의 디폴트 이름

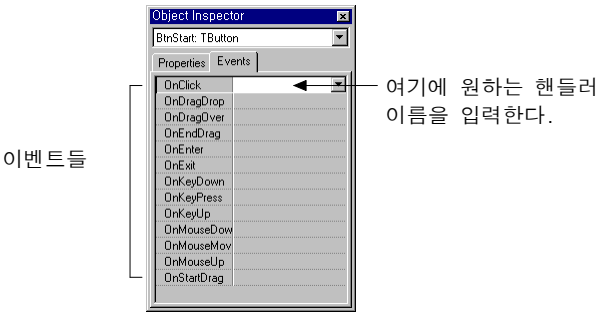
| 컴포넌트 | 핸들러 | 설명 |
|---|---|---|
| 폼 | FormCreate | 폼이 만들어질 때 |
| 레이블 | Label1Click | 레이블을 클릭할 때 |
| 에디트 | Edit1Change | 에디트의 내용이 변할 때 |
| 버튼 | Button1Click | 버튼이 클릭될 때 |

델파이가 디폴트로 붙여 주는 이벤트 핸들러의 이름은 컴포넌트의 이름과 사건의 이름을 조합하여 무척 설명적이다. 위 표에서 보다시피 FormCreate 핸들러는 Form이 Create 될 때 실행되는 이벤트이며 Button1Click 핸들러는 Button1이 Click될 때 실행되는 이벤트라는 것을 쉽게 알 수 있다. 컴포넌트를 더블클릭하면 델파이는 그 컴포넌트에서 가장 빈번하게 발생하는 디폴트 이벤트에 대한 핸들러를 디폴트 이름으로 작성해 주지만 더블클릭하지 않고 다른 방법을 쓴다면 특정 이벤트에 대해서, 사용자 마음에 드는 이름으로 핸들러를 정의할 수도 있다. 즉 Button1의 OnClick 이벤트에 대한 핸들러의 이름이 꼭 Button1Click일 필요는 없으며 MyEvent나 StartGame 등일 수도 있다는 것이다. 직접 실습을 해 보자.

✶ 실습을 위해 새 프로젝트를 시작한 후 빈 폼에 컴포넌트를 배치한다. 예를 들어 게임을 시작하는 버튼 컴포넌트를 BtnStart 라는 이름으로 배치했다고 하자.

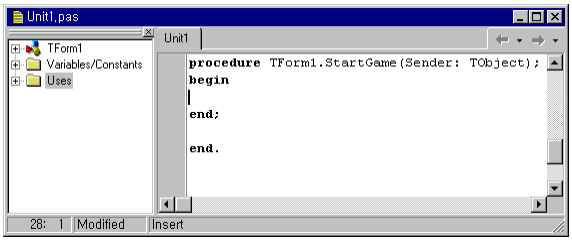
✷ 버튼을 더블클릭하지 말고 클릭만 하여 선택해 둔다. 더블클릭해 버리면 디폴트 핸들러를 작성해 주므로 우리가 원하는 목적을 달성할 수 없게 된다. 클릭하여 선택만 했으므로 오브젝트 인스펙터에는 BtnStart 의 속성들이 나타날 것이다.

✸ 오브젝트 인스펙터의 Events 페이지를 클릭한다. 좌측에 이벤트 리스트가 있고 우측에 이벤트 핸들러를 입력하는 난이 있다.




프로시저란 특정 작업을 담당하는 코드의 한 부분을 말하며 6 장에서 자세히 다루고 있다.

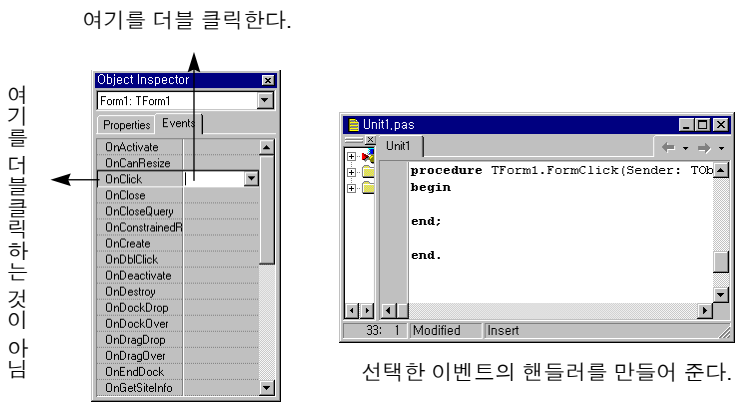
❋ 작성하고자 하는 이벤트를 선택하고 우측에 핸들러의 이름을 직접 입력한다. OnClick 이벤트에 대해 StartGame 이라는 이름을 입력하고 Enter 를 누르면 코드 에디터는 다음과 같이 된다. BtnStart 의 OnClick 이벤트 핸들러로 StartGame 이라는 프로시저가 만들어졌으며 사용자는 여기에 원하는 코드를 작성하면 된다. 물론 프로그램 실행중에 사용자가 BtnStart 버튼을 누르면 이 코드가 실행될 것이다.




이 절차가 이벤트 헨들러를 작성하는 원래의 방법인데 너무 번거로우니까 폼에 있는 컴포넌트를 더블클릭만 하면 델파이가 디폴트 이벤트에 디폴트 이름으로 핸들러를 작성해 주는 것이다. 이벤트 핸들러의 이름은 어떤 컴포넌트의 어떤 이벤트에 대한 처리를 하는지 가급적이면 알아보기 쉽게 작성될 뿐이지 사실 컴포넌트나 이벤트와 논리적인 연관이 있는 것은 아니다. Button1Click 이라고 해서 반드시 Button1 컴포넌트의 OnClick 이벤트를 처리해야 하는 것은 아니며 이 핸들러를 Form 의 OnCreate 이벤트에 대한 핸들러로 사용하고자 한다면 굳이 못할 이유도 없다. Button1Click 이라는 명칭은 어디까지나 프로시저의 명칭(Identifier)일 뿐이기 때문이다.

오브젝트 인스펙터의 Events 페이지의 정확한 기능은 선택된 컴포넌트의 선택된 이벤트에 어떤 핸들러가 연결되어 있는가를 보여주고 핸들러의 이름을 입력할 수 있도록 해주는 것이다. 만약 Form 의 OnClick 이벤트에 대한 핸들러를 작성하고 싶다면 어떻게 해야 할까? 폼을 그냥 더블클릭하면 디폴트로 지정된 OnCreate 이벤트 핸들러가 생성되므로 더블클릭해서는 문제를 해결할 수 없다.

폼을 선택한 후 오브젝트 인스펙터의 OnClick 이벤트를 더블클릭하면 FormClick 이벤트 핸들러를 만들 수 있다. 직접 실습을 해 보도록 하자.



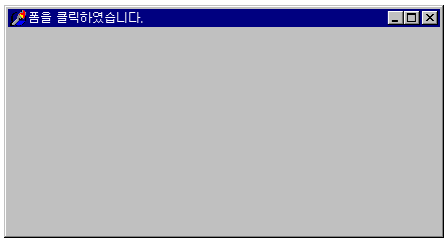

이벤트 핸들러를 작성하는 방법을 표로 요약해 보았다. 표에서 첫 번째 행을 보면 디폴트 이벤트(즉 버튼의 경우 OnClick, 폼의 경우 OnCreate)에 대해 디폴트 이름(Button1Click, FormCreate 등)으로 이벤트 핸들러를 작성하려면 폼에서 컴포넌트를 더블클릭하기만 하면 된다는 뜻이다.

| | 폼에서 선택 | 이벤트 | 핸들러 |
|---|---|---|---|
| 디폴트 이벤트와 디폴트 이름 | 컴포넌트 더블클릭 | 디폴트 | 디폴트 |
| 이벤트 선택과 디폴트 이름 | 클릭 | 더블클릭 | 디폴트 |
| 이벤트 선택과 이름 지정 | 클릭 | 클릭하여 선택 | 입력 |

책을 읽을 때 표나 그림 따위는 무심코 지나치는 사람들이 있는데 참 안좋은 버릇이다. 위 표는 델파이를 이해하는 데 있어 아주 중요한 내용을 담고 있으므로 혹시 지나쳤다면 다시 한번 유심히 봐 두도록 하자.


5jang
fclick

다음 예제는 폼의 OnClick 이벤트 핸들러에 Caption:='폼을 클릭하였습니다.';코드를 작성하여 폼을 클릭할 경우 폼의 캡션을 변경하는 것이다. 배포 CD에 있는 예제만 실행시켜 보지 말고 직접 이 예제를 만들어 보고 이벤트 핸들러를 작성하는 과정을 확실하게 익혀두기 바란다.

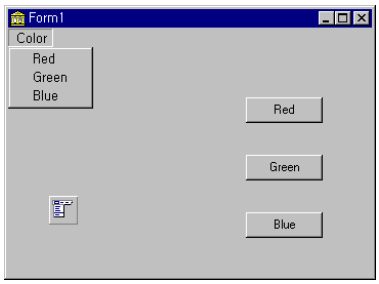



예제를 만드는 방법을 익혀야 하기 때문에 배포 CD 에 있는 예제를 실행해 보는 것은 의미가 없다. 이 예제를 만든 후 폼의 더블클릭 이벤트(OnDblClick)를 처리하는 예제도 만들어 보도록 하되 이벤트 핸들러의 이름을 epfvkdl 이라는 괴상한 이름으로 줘 보도록 하자. 핸들러에서는 ShowMessage 등의 함수를 넣어 제대로 호출되는지 확인하면 된다.

# 다. 메뉴와 같은 기능의 버튼

5jang
event1

이제 두 개의 컴포넌트에 하나의 이벤트 핸들러를 연결하는 실습을 해 보자. 우선 메뉴와 버튼들을 배치한다. 버튼의 이름은 BtnRed, BtnGreen, BtnBlue 로 설정하고 캡션은 Name 속성에서 Btn 을 빼고 Red, Green, Blue 로 설정한다. 메뉴 항목도 같은 이름으로 세 개를 만든다. 물론 캡션만 버튼과 같을 뿐이며 Name 속성은 Red1, Green1 등이 될 것이다.




그리고 메뉴 항목에 대해 코드를 작성한다. 이때는 폼 디자이너에서 메뉴 항목을 더블클릭하여 디폴트 핸들러를 사용하기로 한다.

```
rocedure TForm1.Red1Click(Sender: TObject);
begin
  Form1.Color:=clRed;
end;
```

```
procedure TForm1.Green1Click(Sender: TObject);
begin
  Form1.Color:=clGreen;
end;

procedure TForm1.Blue1Click(Sender: TObject);
begin
  Form1.Color:=clBlue;
end;
```

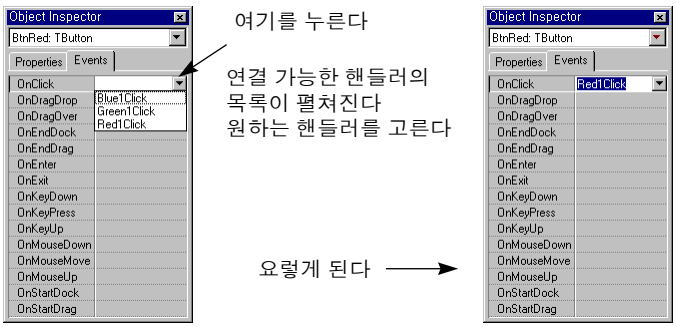
이제 메뉴 항목에 대한 코드는 만들어졌으며 버튼의 OnClick 이벤트를 각 메뉴의 OnClick 이벤트에 연결시켜 준다. 대표적으로 BtnRed와 Red1과의 연결만 해보자.

✶ 폼에서 BtnRed 를 클릭하여 선택한다. 오브젝트 인스펙터에는 BtnRed 의 속성이 나타날 것이다.

✷ 오브젝트 인스펙터의 Events 페이지를 선택하고 OnClick 이벤트를 클릭한다.

✹ 여기서 그냥 더블클릭을 하면 BtnRedClick 이라는 이벤트 핸들러가 만들어지겠지만 그 이름 대신 Red1 메뉴 항목의 OnClick 이벤트 핸들러인 Red1Click 을 입력한다.

❋ 아니면 오브젝트 인스펙터의 ▼를 눌러서 핸들러를 선택한다. ▼를 누르면 현재 이벤트와 연결될 수 있는 핸들러의 목록이 나타난다. 이 상태에서 연결하고자 하는 핸들러를 선택한다.




BtnRed의 OnClick 이벤트 핸들러를 Red1Click으로 지정하자. 이렇게 되면 Red1Click이라는 이벤트 핸들러는 Red1 메뉴 항목의 OnClick에 대해서도 반응하며, BtnRed의 OnClick에 대해서도 호출된다. 하나의 코드를 두 개의 컴포넌트가 공유하게 된 것이다. 같은 방법으로 BtnGreen의 OnClick에

Green1Click, BtnBlue의 OnClick에 Blue1Click을 연결해 주면 프로그램이 완성된다. 실행시켜 보면 메뉴에서 항목을 선택하나 버튼을 누르나 동일한 코드가 실행됨을 확인할 수 있을 것이다. 왜냐하면 두 컴포넌트가 같은 핸들러를 공유하고 있기 때문이다.

참고하세요

특정 이벤트에 어떤 이벤트가 연결될 수 있는가는 이벤트의 타입에 따라 다르다. 버튼의 OnClick 이벤트나 폼의 OnClick 이벤트끼리는 이벤트 타입이 같으므로 컴포넌트에 상관없이 상호 연결될 수 있음은 물론이고 OnClick과 OnCreate, OnShow 등의 이벤트끼리도 같은 타입의 이벤트이므로 연결될 수 있다. 그러나 OnClick과 OnMouseMove, OnKeyDown 등의 이벤트끼리는 이벤트 타입이 맞지 않으므로 연결될 수 없다. 이벤트의 타입은 각 이벤트의 도움말을 참조하거나 이 책을 좀 더 읽다보면 무슨 말인지 알게 될 것이다.

## 라. 하나의 이벤트 핸들러로 통합

5jang
event2

이제 좀 더 고급적인 이벤트 연결에 대해 연구해 보자. 앞에서 만든 예제는 메뉴 항목과 버튼의 이벤트를 연결하기는 했지만 그래도 핸들러가 세 개나 존재한다. 색상을 바꾸는 코드는 같고 바꿀 색상만 다를 뿐이므로 이번에는 세 개의 핸들러까지도 통합시켜 버리도록 하자. 완전히 같은 코드는 아니더라도 비슷한 동작을 하는 두 개의 컴포넌트가 하나의 핸들러를 공유할 수 있다.

앞에서 만든 예제와 같이 컴포넌트를 배치하고 이름을 주도록 하자. 이벤트 핸들러는 ChangeColor라는 이름으로 하나만 만들기로 한다. BtnRed의 OnClick 이벤트 핸들러를 ChangeColor라는 이름으로 작성한다. 앞에서 여러 번 설명했지만 다시 한번 요약하자면 BtnRed 버튼을 선택한 후 오브젝트 인스펙터의 Events 페이지의 OnClick에 ChangeColor를 입력해 주면 된다. 핸들러를 만든 후 코드를 다음과 같이 입력한다.

```
procedure TForm1.ChangeColor(Sender: TObject);
begin
  if (Sender=Red1) or (Sender=BtnRed) then
    Form1.Color:=clRed;
  if (Sender=Green1) or (Sender=BtnGreen) then
    Form1.Color:=clGreen;
  if (Sender=Blue1) or (Sender=BtnBlue) then
```

```
    Form1.Color:=clBlue;
end;
```

그리고 나머지 두 개의 버튼과 세 개의 메뉴 항목의 OnClick 이벤트를 이 코드에 연결시켜 준다. 각 컴포넌트의 OnClick 이벤트에 ChangeColor 프로시저를 선택해 주기만 하면 된다. 프로그램 작성이 끝났다. 얼마나 간단한가. 컴포넌트 배치하고 이벤트 핸들러 하나만 만들고 몽땅 연결시켜 주면 되니까 말이다. 마지막 남은 문제는 코드의 내용을 이해하는 일인데 문법을 배우지 않은 현재 단계에서는 완벽하게 이해하기가 어렵다. 이벤트 핸들러가 호출될 때 어떤 컴포넌트에서 이벤트가 발생했는가에 관한 정보가 Sender라는 인수로 전달되어진다. 이 인수의 값을 조사함으로써 이벤트를 발생시킨 컴포넌트에 따라 처리를 다르게 지정한다. Sender의 값을 평가하는데는 if 조건문이 사용되었다. 위 코드의 내용을 말로 풀어보면

```
만약 Red1 이나 BtnRed 에서 OnClick 이 발생하면 폼을 빨간색으로 칠하고
만약 Green1 이나 BtnGreen 에서 OnClick 이 발생하면 폼을 초록색으로 칠하고
만약 Blue1 이나 BtnBlue 에서 OnClick 이 발생하면 폼을 파란색으로 칠한다.
```

이렇게 된다. 이벤트를 연결시켜 놓았으므로 여섯 개의 컴포넌트 중 하나라도 OnClick이 발생하면 ChangeColor가 호출되고 ChangeColor에서는 Sender 인수를 사용하여 어떤 컴포넌트에서 OnClick이 발생했는지를 조사한다. 비슷한 코드를 한 곳에 모을 수 있고 수정할 때 한군데만 수정하면 되므로 여러 가지 이점이 있다.

이벤트 핸들러를 연결하는 기법은 결코 어렵거나 복잡한 것이 아니다. 초보자에게는 좀 어려워 보일 수도 있겠지만 한번 신경 써서 배워 놓으면 두고 두고 몸을 편하게 할 수 있다. 이 방법을 모르면 매 컴포넌트마다 같은 코드를 작성하는 노가다를 해야 할 것이다. 머리가 딸리면 수족이 고생한다는 말이 이 경우에 딱 어울리지 않을지. 몇 번씩 실습을 반복하여 숙달될 필요가 있는 부분이다.

# 마. 액션 리스트

요즘의 프로그램들은 하나의 동작을 명령하는데 여러 가지 방법을 제공하기 때문에 이벤트 핸들러를 통합하는 경우는 아주 흔하다. 똑같은 명령에 대해 메뉴 항목, 버튼, 키보드 단축키, 툴바, 팝업 메뉴까지 다양한 방법을 사용할 수 있

다. 지금 여러분들이 쓰고 있는 델파이만 해도 프로젝트를 여는데 File/Open Project 메뉴, 툴바, Ctrl+F11 단축키까지 세 가지나 되는 방법을 제공한다. 그래서 델파이는 이런 이벤트 핸들러 연결을 좀 더 쉽게 하기 위해 액션 리스트(ActionList)라는 컴포넌트를 제공한다. 이 컴포넌트는 Standard 페이지의 제일 끝에 있으며 델파이 4.0에서 새로 추가되었다.

액션 리스트 컴포넌트로부터 액션(Action)이라는 내부 컴포넌트를 만든다. 마치 메인 메뉴 컴포넌트로부터 메뉴 항목(MenuItem) 컴포넌트를 만드는 것처럼 말이다. 이런 식으로 컴포넌트 팔레트에는 없지만 다른 컴포넌트로부터 만들어지는 컴포넌트를 내부 컴포넌트(Internal Component)라고 하며 델파이에는 메뉴 항목, 액션 외에도 여러 개의 내부 컴포넌트가 있다. 내부 컴포넌트를 직관적으로 이해하려면 메인 메뉴로부터 만들어지는 메뉴 항목 컴포넌트를 생각하면 된다.

액션 컴포넌트는 화면에는 보이지 않지만 버튼이나 메뉴 항목처럼 사용자의 명령을 받아들이는 장치이다. 사용자의 명령을 받고 사용자에게 프로그램의 현재 상태를 보여주는 이런 장치를 간단히 줄여 UI(User Interface)라고 하며 메뉴 항목, 툴바, 단축키 등이 이에 해당한다. 액션은 액션 리스트 컴포넌트로부터 만들어지며 일종의 컴포넌트이므로 당연히 속성과 이벤트를 가진다. 액션의 속성에는 Caption, Name, Checked, Enabled, Tag, Hint 등등이 있는데 이때까지 사용해 왔던 아주 일반적인 속성들이다. 액션의 이벤트로는 액션이 실행될 때 발생하는 OnExecute가 대표적인데 이 이벤트가 디폴트 이벤트이다. 그외 풍선 도움말을 출력하는데 사용되는 OnHint, 액션의 상태를 변경하는 OnUpdate 이벤트가 있다.

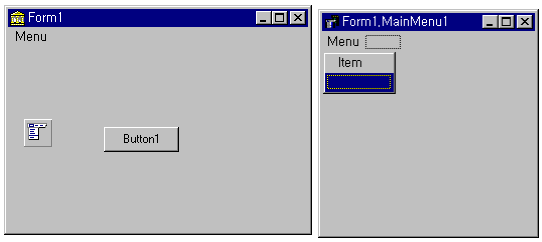
액션은 눈에 보이는 컴포넌트가 아니기 때문에 사용자가 직접 사용할 수 없으며 그래서 버튼이나 메뉴 항목처럼 가시적인 다른 컴포넌트와 연결되어 사용된다. 한 액션에 여러 개의 UI가 연결될 수 있기 때문에 자연스럽게 이벤트 핸들러를 공유할 수 있는 것이다. 게다가 액션의 상태(Checked, Enabled 등)는 곧바로 연결된 컴포넌트의 상태를 변경시키므로 실행 중에 UI의 상태를 바꾸기에도 편리하다. 액션에 대한 이론을 죽 설명해 보았는데 이런식으로 이론만 읽어서는 이해하기 무척 어려운 컴포넌트이므로 간단하나마 예제를 하나 만들어 보도록 하자.

5jang
Action1

지금 만들 예제는 메뉴 항목 하나와 버튼 하나를 폼에 배치하고 두 컴포넌트가 선택되었을 때 같은 메시지를 보여주도록 할 것이다. 앞에서 배운 전통적인 방법대로라면 버튼의 OnClick 이벤트 핸들러와 메뉴 항목의 OnClick 이벤트 핸들러를 일치시켜 주면 된다. 이 예제에서는 액션 리스트 컴포넌트를 사용하여

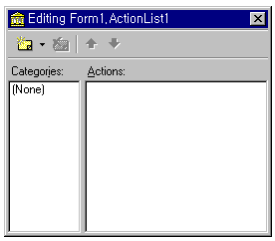
이벤트 핸들러를 공유해 볼 것이다. 다음 단계를 따라 예제를 만들어 보자.

우선 새 프로젝트를 시작하고 폼에 버튼 하나와 메인 메뉴를 하나 배치한다. 버튼의 속성은 그대로 두고 메인 메뉴에는 다음과 같이 Menu 아래 Item이라는 항목만을 만들어 두었다.

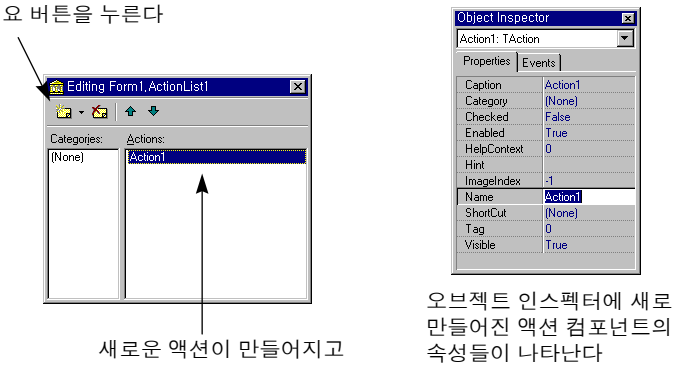



두 UI에 같은 핸들러를 연결하여 동일한 메시지를 출력하도록 할 것이다. 전통적인 방법대로라면 이벤트 핸들러를 작성하고 연결하면 되겠지만 여기서는 액션 리스트를 대신 사용하기로 하자. Standard 페이지의 제일 오른쪽에 있는 액션 리스트 컴포넌트를 폼에 배치한다. 그리고 이 컴포넌트를 씩씩하게 더블클릭해 보자. 다음과 같은 액션 편집기가 열릴 것이다. 마치 메인 메뉴 컴포넌트를 더블클릭하면 메뉴 디자이너가 열리는 것과 같다.

그림
액션 편집기



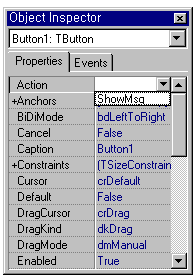

아직까지 작성된 액션이 전혀 없는 상태인데 이 상태에서 액션 편집기의 제일 좌상단에 있는 툴 버튼을 눌러보자. Actions 리스트 박스에 Action1이라는 컴포넌트가 나타나고 오브젝트 인스펙터에는 이 컴포넌트의 속성들이 나열될 것이다.



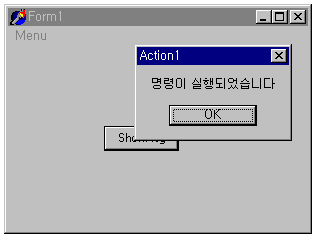

역시 액션도 하나의 컴포넌트라는 것이 확인되었다. 이 과정은 메뉴 디자이너에서 메뉴 항목을 만드는 것과 거의 흡사하다. 이렇게 만들어진 Action1 컴포넌트의 Name 속성을 ShowMsg로 변경한다. 그리고 액션 편집기에서 ShowMsg 컴포넌트를 더블클릭하면 이 컴포넌트의 디폴트 이벤트인 OnExecute의 이벤트 핸들러가 작성될 것이다.

```
procedure TForm1.ShowMsgExecute(Sender: TObject);
begin
  ShowMessage('명령이 실행되었습니다');
end;
```

ShowMessage 프로시저를 호출하여 메시지를 보여주도록 하였다. 이제 사용자가 이 액션을 선택하면 메시지가 출력될 것이다. 그런데 액션은 실행중에 보이지 않기 때문에 사용자가 직접 선택할 수 없다. 액션은 홀로 사용되지 않고 다른 컴포넌트와 함께 사용되는데 이 예제의 경우 액션을 버튼, 메뉴 항목과 연결시켜 주어야 한다. 버튼, 메뉴 항목, 스피드 버튼, 툴 버튼 등의 컴포넌트에는 공통적으로 Action이라는 속성이 있는데 이 속성으로 어떤 액션과 연결될 것인가를 지정한다. 폼에서 버튼을 선택한 후 오브젝트 인스펙터에서 Action 속성을 선택해 보자.

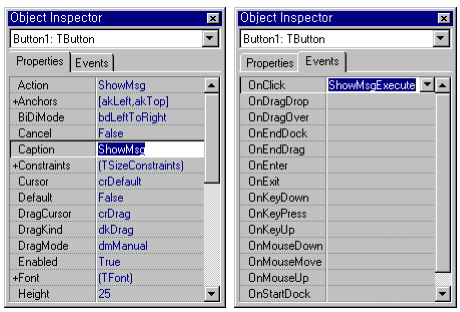


Action 속성의 콤보 박스에는 이 프로젝트에 작성되어 있는 모든 액션의 목록이 나타나는데 우리가 하나의 액션만 만들어 두었기 때문에 ShowMsg 하나밖에 나타나지 않는다. 버튼의 Action 속성에 ShowMsg 액션을 선택하도록 하자. 그러고 같은 방법은 Item1 메뉴 항목의 Action 속성도 ShowMsg로 변경한다. 이제 버튼과 메뉴 항목이 Action 컴포넌트의 OnEcecute 이벤트 핸들러를 공유하게 된 것이다. 과연 제대로 공유가 되었는지 예제를 실행시켜 보자. 버튼을 누르나 메뉴 항목을 선택하나 똑같은 메시지가 출력될 것이다.




이상으로 액션을 사용하여 이벤트 핸들러를 연결하는 간단한 실습을 해 보았다. 좀 복잡하다고 생각되는 사람들은 예제를 처음부터 끝까지 다시 한번 더 작성하면서 복습을 하기 바란다.

예제를 만드는 방법은 알았다 치고 이제 액션과 버튼의 관계에 대해 약간의 관찰을 해보자. 폼에서 버튼을 선택하고 오브젝트 인스펙터를 자세히 관찰해 보면 몇 가지 사항이 자동으로 변경되었음을 발견할 수 있다.




우선 버튼의 Caption 속성이 액션의 Caption 속성과 같아졌다. 우리가 직접 버튼의 캡션을 바꾸지는 않았지만 버튼의 Action 속성을 변경함으로써 버튼의 캡션이 액션의 캡션을 따라간다는 것을 알 수 있다. 또한 Events 페이지를 보면 버튼의 OnClick 이벤트 핸들러가 ShowMsgExecute로 자동 설정되어 있다. 그래서 버튼을 누르면 ShowMsg 액션의 OnExecute 이벤트 핸들러가 실행되었던 것이다. 버튼뿐만 아니라 메뉴 항목의 속성을 살펴보면 역시 마찬가지로 캡

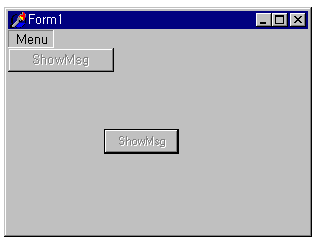
션과 OnClick 이벤트 핸들러가 변경되었다.

이쯤에서 결론을 내리자면 컴포넌트의 Action 속성을 설정하면 컴포넌트의 속성들이 연결된 액션의 속성값을 대입받는다는 것이다. Caption 속성뿐만 아니라 Checked, Enabled 등의 속성들도 마찬가지다. 왜 이렇게 되어 있는가 하면 액션에 연결된 UI들은 모두 같은 목적을 가지고 있으므로 액션만 관리함으로써 연결된 모든 UI들을 한꺼번에 관리할 수 있도록 하기 위해서이다. 예를 들어 액션에 지정된 동작이 현재 사용 불가능 상태라면 액션의 Enabled 속성만 False로 바꾸어 줌으로써 연결된 모든 컴포넌트들도 사용 불가능 상태로 바꿀 수 있게 된다.

UI의 일괄적인 갱신을 위해서 액션은 OnUpdate라는 이벤트를 가지는데 이 이벤트는 프로그램이 아무 일도 하고 있지 않는 한가한 때(OnIdle)에 발생한다. OnUpdate 핸들러에서 프로그램의 현재 상태를 점검해 보고 상황에 따라 적절히 UI를 갱신하면 된다. 액션의 OnUpdate 핸들러를 다음과 같이 작성해 보자.

```
procedure TForm1.ShowMsgUpdate(Sender: TObject);
begin
  if Left < 600 then
    ShowMsg.Enabled:=True
  else
    ShowMsg.Enabled:=False;
end;
```

예제 수준에서 UI 갱신 조건이 적절한 것이 없어 폼의 좌표에 따라 UI를 갱신하도록 하였다. 즉 폼의 X좌표가 600미만일 경우면 ShowMsg를 사용할 수 있고 600이상일 경우는 ShowMsg를 사용금지시켰다. 이 코드에서 사용된 Left < 600이라는 조건문은 실제 프로그램에서는 적절한 다른 조건문으로 바뀌어야 할 것이다. 이제 예제를 실행해 보면 폼이 600좌표 이상으로 이동되면 버튼과 메뉴 항목이 사용 금지될 것이다.




ShowMsg 액션 컴포넌트의 Enabled 속성이 변경되면 연결된 모든 컴포넌

트의 Enabled 속성도 같이 변경되기 때문이다. Enabled 속성뿐만 아니라 Visible, Checked, Caption 등의 속성들도 액션의 것만 변경해 주면 연결된 컴포넌트의 상태들도 모두 바뀌게 된다.

이상으로 액션 리스트와 액션 컴포넌트에 대해 알아 보았다. 여러분들이 아직 명령을 내리는 UI 중 메뉴 항목과 버튼밖에 배우지 않아 두 개의 컴포넌트만 연결해 보았는데 툴바나 스피드 버튼 등까지 합해 한꺼번에 서너 개의 컴포넌트가 같은 명령에 사용될 때는 전통적인 이벤트 연결 방법보다 액션을 사용하는 것이 훨씬 더 유리하다.


① 이벤트 핸들러의 이름은 사용자가 마음대로 설정할 수 있다.
② 오브젝트 인스펙터를 사용하여 이미 작성된 핸들러를 연결한다.
③ Sender 인수로 이벤트가 발생한 컴포넌트가 전달된다.
④ 액션을 사용하면 핸들러를 통합할 수 있다.

# 5-6 간단한 에디터 2

4장에서 메모 컴포넌트를 사용하여 아주 간단한 에디터를 만들어 보았다. 이번에는 메뉴 항목을 넣어 보고 선택 영역을 클립보드로 보내거나 받는 블럭 기능을 추가해 보도록 하자.

## 가. 프로젝트 이름 바꾸기

4장에서 만든 간단한 에디터 예제의 이름이 Editor1이었다. 이 예제를 계속해서 만들어 나가되 만약의 경우를 고려하여 앞에서 만든 예제를 유지한 채로 이름을 바꾸어 새 프로젝트로 복사한 후 계속 만들어 나가는 것이 좋다. Editor1에 기능을 추가하여 Editor2를 만들되 원래의 Editor1도 그대로 두려고 할 경우라든가 프로젝트 이름이 마음에 안들어 바꾸고자 할 경우 프로젝트의 이름을 바꾸어 저장하려면 File 메뉴의 Save As 항목을 사용한다.

우선 새로운 프로젝트를 저장할 디렉토리부터 작성한다. Editor2라는 디렉토리를 만든후 File/Save As를 선택해서 유닛 파일을 editor2f.pas로 바꾸고 File/Save Project As를 선택하여 프로젝트를 editor2.dpr로 바꾸어 준다. 물론 이름만 바꾸는 것이 아니라 디렉토리 위치도 Editor2로 바꾸어 주어야 한다.

혹시, 너무 당연한 얘기를 하는 것이 아니냐고 할 지도 모르겠다. 물론 아주 당연한 얘기를 하고 있다. 그럼에도 불구하고 왜, 지면을 낭비해 가며 이런 얘기를 하는가 하면 디렉토리를 만든 후 원래 프로젝트 파일을 몽땅 복사하고 도스의 ren 명령(또는 탐색기로)으로 직접 파일 이름을 바꾸는 사람들이 있기 때문이다. 그렇게 하면 안된다. 파일 이름을 바꿀 때 델파이에게 바꾸겠다고 신고하고 바꾸어야 델파이가 바꾼 파일 이름을 제대로 인식해 낼 수 있다. 이름을 바꾸거나 위치를 바꾸는 이런 문제에 대해 유형별로 어떻게 해야 하는지 정리해 보면 다음과 같다.

**프로젝트 전체를 다른 디렉토리로 옮길 때**

이때는 단순히 디렉토리 전체를 복사해 주기만 하면 된다. 프로젝트는 보통 디렉토리 단위로 구성되므로 디렉토리 전체를 복사했다면 컴파일하는데 문제가 없다. 단 프로젝트의 구성 파일이 상대 경로로 설정되어 있을 경우에는 디렉토

리 이동 후 경로를 조정해 주어야 한다.

**프로젝트의 이름을 바꿀 때**

델파이가 만들어내는 실행 파일의 이름은 프로젝트의 이름을 따라가므로 생성되는 실행 파일의 이름을 바꾸려면 프로젝트의 이름을 바꾸어야 한다. 처음에 project1.dpr로 저장한 프로젝트를 myutil.dpr로 변경하고자 한다. 파일의 이름을 변경해야 하므로 도스의 ren 명령이나 탐색기를 사용하면 될 것 같지만 그래서는 안된다. 왜냐하면 프로젝트 파일인 DPR 파일의 이름이 프로젝트 선두의 program 선언에 사용되며 파일 이름과 program 선언이 일치해야 하기 때문에 파일 이름만 바꾸어서는 안된다. 프로젝트 이름을 바꿀 때는 프로젝트를 열어놓은 채로 Save Project As... 메뉴를 통해서 이름을 바꾸어야 파일 이름과 코드에 사용된 파일 이름이 동시에 바뀐다. 이렇게 프로젝트의 이름을 바꾼 후 원래 프로젝트 파일을 삭제하면 된다.

**유닛의 이름을 바꿀 때**

프로젝트의 이름을 바꾸는 것과 동일하다. 유닛 파일의 이름이 프로젝트 파일 내의 코드에서 사용되므로 유닛 파일의 이름을 ren 명령으로 변경해서는 안된다. 반드시 Save As... 메뉴를 통해야 한다.

**유닛 파일의 경로를 바꿀 때**

프로젝트의 구성 파일이 여러 디렉토리에 흩어져 있을 경우가 있다. 이때는 가급적 구성 파일을 한 디렉토리에 모으는 것이 바람직하다. 유닛 파일의 경로를 바꿀 때도 유닛 파일만 달랑 이동시켜서는 안된다. 왜냐하면 프로젝트 파일에 유닛 파일의 경로가 기억되어 있기 때문이다. 이때도 반드시 Save As... 메뉴를 통해 프로젝트가 있는 디렉토리에 다시 저장시켜야 한다.

다른 개발 환경에서는 구성 파일의 이름을 바꾸거나 위치를 옮기는 것이 표준 파일 관리 명령으로 가능하지만 델파이의 구성 파일은 반드시 델파이를 통해서 관리해야 한다. 왜 그런가 하면 구성 파일의 이름이나 위치가 코드에서 사용되기 때문이다. 정말로 그런지는 아무 프로젝트나 열어서 확인해 보면 된다. Menu1 프로젝트를 열어 보면 첫 줄에 다음과 같은 선언문이 있을 것이다.

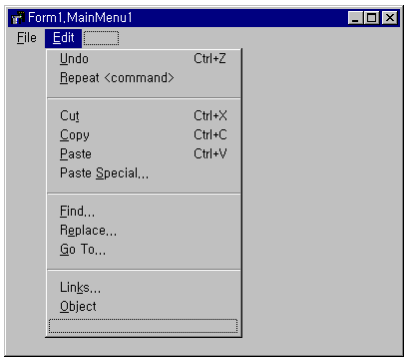
```
unit Menu1_f;
```

이 유닛의 이름이 Menu1_f라는 선언인데 아직까지 유닛이 뭔지는 몰라도 어쨌든 이 이름이 파일명을 따라간다는 것을 알 수 있을 것이다. 이 이름과 파일명이 일치하지 않으면 컴파일이 안된다.

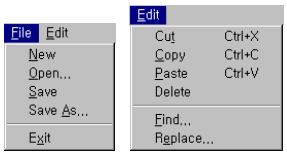
# 나. 메뉴 만들기

5jang
editor2

4장에서 만든 간단한 에디터에 기능을 덧붙여 새로운 에디터를 만들어 보자. Editor2 디렉토리에 Editor2 프로젝트를 새로 복사하되 앞에서 설명한 대로 메뉴를 통해야 한다. 요약하자면 Save As한 후 Save Project As하면 된다. 프로젝트를 새로운 이름으로 복사한 후 기능을 하나씩 첨가해 나간다. 우선 밋밋한 폼에 메뉴를 달아 보자. 제일 먼저 해야 할 일은 메뉴 컴포넌트를 폼에 배치하는 일이며 다음은 폼에 놓은 메뉴 컴포넌트를 더블클릭하여 메뉴 디자이너를 연다. 그 다음 할 일은 메뉴 항목을 만드는 일이겠지만 직접 만들지 말고 메뉴 템플릿을 사용하도록 하자. 메뉴 템플릿의 File, Edit를 폼 디자이너로 불러온다. 템플릿을 불러온 후 폼 디자이너는 다음과 같이 되어 있을 것이다.




템플릿에서 불러온 File, Edit 메뉴는 필요한 거의 대부분의 메뉴 항목을 포함하고 있지만 우리에게 당장 불필요한 것들도 있다. 사용하지 않을 메뉴 항목은 과감히 삭제해 버리고 추가할 항목은 적당히 추가하여 다음 그림과 같이 꼭 필요한 메뉴 항목만 남겨 두도록 하자.




Edit 메뉴의 항목들이 대폭 삭제되었으며 Delete 항목을 추가하였다. 이렇게 만든 메뉴 항목들에 하나 하나씩 기능을 부여해 나갈 것이다.

# 다. 클립보드 사용

메모 컴포넌트는 블럭 선택 기능을 제공하며 선택된 블럭 영역을 관리하는 메소드도 제공한다. 여기서 블럭을 관리한다는 것은 Cut, Copy, Paste를 말하며 메모의 다음 메소드들이 사용된다.

| 메소드 | 동작 |
|---|---|
| CutToClipboard | 선택 영역을 잘라 클립보드로 보낸다. |
| CopyToClipboard | 선택 영역을 복사하여 클립보드로 보낸다. |
| PasteFromClipboard | 클립보드의 내용을 현재 커서 위치에 삽입한다. |
| ClearSelection | 선택 영역을 삭제한다. |

Edit 메뉴의 Cut, Copy, Paste 항목의 이벤트 핸들러에 이 메소드 호출을 기입하여 메뉴 항목이 선택될 때 블럭 동작을 하도록 한다. 단순한 메소드 호출이므로 코드가 모두 한 줄밖에 되지 않는다.

```
procedure TForm1.Cut1Click(Sender: TObject);
begin
  Memo1.CutToClipboard;
end;

procedure TForm1.Copy1Click(Sender: TObject);
begin
  Memo1.CopyToClipboard;
end;

procedure TForm1.Paste1Click(Sender: TObject);
begin
  Memo1.PasteFromClipboard;
end;

procedure TForm1.Delete1Click(Sender: TObject);
begin
  Memo1.ClearSelection;
end;
```

이 코드만 작성하면 사실 블럭 기능을 다 작성한 셈이다. 직접 예제를 실행해서 과연 블럭 기능이 제대로 수행되는지 확인해 보아라.

참고하세요

메모 컴포넌트(에디트도 포함)에는 원래 내장된 블럭 관리 쇼트컷이 있다. 굳이 메소드를 코드로 작성하지 않더라도 Shift-Ins, Shift-Del, Ctrl-Ins키로 블럭 조작을 할 수 있다. 쇼트컷이 메모 컴포넌트에 제공됨에도 불구하고 왜 메소드를 중복하여 제공하는가 하면 메소드와 쇼트컷의 활용 용도가 다르기 때문이다. 메소드는 코드에 삽입되므로 다양한 상황에서 팔방으로 사용할 수 있지만 쇼트컷은 실행중에 사용자가 직접 키보드를 조작하는 것이므로 프로그래밍의 대상이 되지 못한다. 만약 쇼트컷이 있다 하여 메소드를 제공하지 않는다면 툴바를 만드는 것은 불가능해진다. 쇼트컷은 사용자가 키보드를 눌러 주어야만 사용할 수 있으며 마우스로는 사용할 수 없다.

# 라. 메뉴 항목 관리

Edit 메뉴의 Cut, Copy, Paste 항목들은 항상 사용 가능한 것이 아니다. Cut, Copy, Delete 항목은 메모 컴포넌트에 선택된 영역이 있을 때만 사용할 수 있으며 Paste 항목은 클립보드에 복사되어 있는 텍스트가 있을 때만 사용할 수 있다. 사용이 불가능한 메뉴 항목은 흐리게 만들어 사용할 수 없다는 것을 나타내도록 만들어 사용자가 착각을 하지 않도록 해주어야 한다. 이런 메뉴 항목을 관리할 때는 메뉴 항목의 Enabled 속성을 사용하며 그 시점은 Edit 메뉴가 열릴 때이다. Edit1의 OnClick 이벤트에 다음 코드를 작성한다.

```
procedure TForm1.Edit1Click(Sender: TObject);
begin
  Paste1.Enabled:=Clipboard.HasFormat(CF_TEXT);
  if Memo1.SelLength=0 then
    begin
    Cut1.Enabled:=False;
    Copy1.Enabled:=False;
    Delete1.Enabled:=False;
    end
  else
    begin
    Cut1.Enabled:=True;
    Copy1.Enabled:=True;
    Delete1.Enabled:=True;
  end
end;
```

Paste 항목의 사용 가능 여부는 클립보드에 복사된 텍스트가 존재하는가에 따라 달라지며 이 조건 판단에 Clipboard 오브젝트를 사용한다. Clipboard 오브젝트는 윈도우즈의 클립보드를 나타내며 이 오브젝트를 사용하려면 uses절에 Clipbrd 유닛을 추가해 주어야 한다. 유닛에 대해서는 10장에서 자세히 다룰 예정이므로 당장은 몰라도 상관없다.

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

Clipboard 오브젝트의 HasFormat 메소드는 클립보드에 특정 형식의 데이터가 있는지 조사해 주며 인수로 조사할 데이터 형식을 전달해 준다.

| 인수 | 의미 |
|---|---|
| CF_TEXT | 텍스트 |
| CF_BITMAP | 비트맵 |
| CF_METAFILE | 메타 파일 |
| CF_PICTURE | TPicture형의 오브젝트 |
| CF_OBJECT | 기타 오브젝트 |

데이터가 있으면 True를 리턴하고 없으면 False를 리턴한다.

Paste1.Enabled:=Clipboard.HasFormat(CF_TEXT);를 말로 풀어보면 "클립보드에 복사된 텍스트 형식의 데이터가 있으면 Paste1 메뉴 항목을 쓸 수 있도록 하고 없으면 쓸 수 없도록 한다." 는 뜻이다.

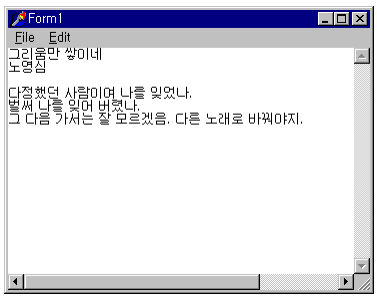
Cut, Copy, Delete 메뉴 항목의 사용 가능 여부는 선택 영역의 유무에 따라 결정된다. 메모 컴포넌트(에디트 컴포넌트도 동일하다)는 다음과 같은 선택 영역과 관련된 속성들을 가진다.

| 속성 | 의미 |
|---|---|
| SelStart | 선택 영역의 처음 위치 |
| SelLength | 선택 영역의 길이 |
| SelText | 선택 영역의 문장 |

메모 컴포넌트의 SelLength 속성이 0이면, 즉 선택된 문장이 없으면 Cut, Copy 메뉴 항목은 사용할 수 없다. 마지막으로 File 메뉴의 Exit 항목에 대한 이벤트 핸들러를 정의해 주면 예제가 완성된다.

```
procedure TForm1.Exit1Click(Sender: TObject);
begin
  close;
end;
```

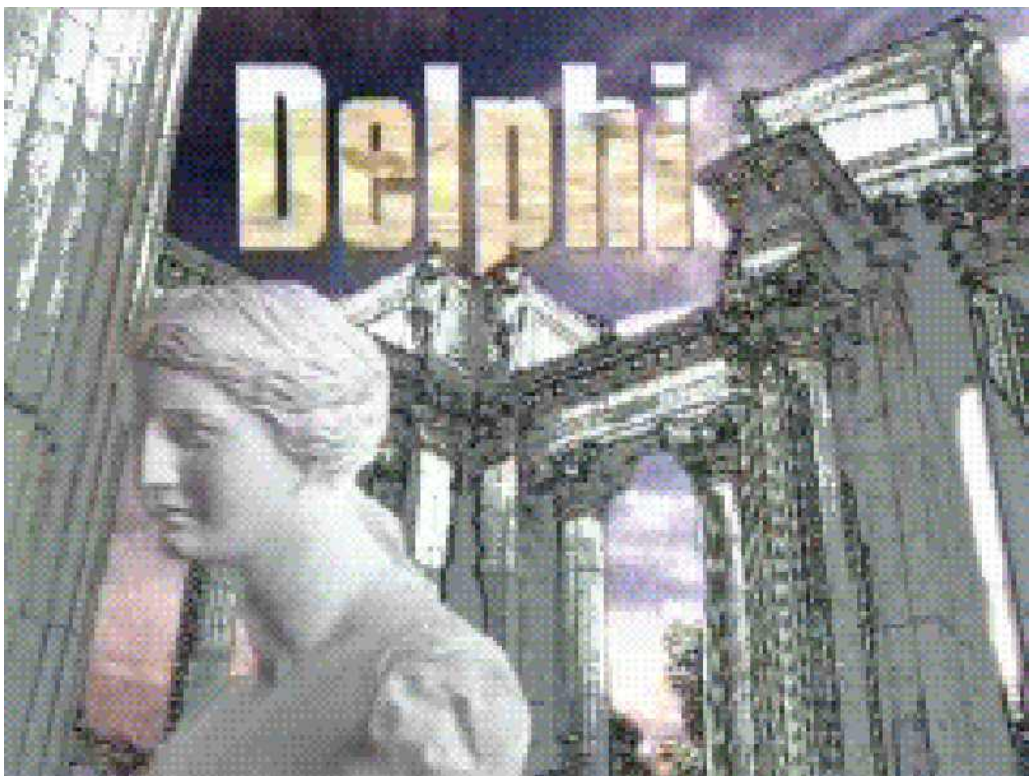
close 명령은 폼의 메소드이며 폼을 닫는 동작을 한다. 폼을 닫음으로써 프로그램을 종료한다. 프로그램 실행중의 모습은 다음과 같다.




메뉴가 추가되었고 블럭 관리 명령들이 추가되었다. Editor2 예제는 여기까지만 작성하고 다음 번에는 툴바가 있는 에디터를 만들어 보기로 하자.

# 제 6 장

# 파스칼 기본 문법



# 6-1 오브젝트 파스칼

## 가. 파스칼

델파이는 비주얼 개발 툴이다. 마우스로 컴포넌트를 끌어와 폼에 배치한 후 속성을 설정하고 컴포넌트의 위치, 크기 등을 모두 마우스로 조작할 수 있다. 하지만 완전 비주얼 툴은 아니기 때문에 마우스만으로 프로그램을 완성시킬 수는 없으며 이벤트 핸들러를 만드는 코딩 과정이 반드시 필요하다.

델파이는 오브젝티브 파스칼(Objective Pascal) 언어를 사용하여 코드를 작성한다. 그래서 델파이에 조금이라도 관심이 있다면 파스칼을 먼저 알아야 하며 프로그래밍의 논리에 대한 개념이 있어야 한다. 델파이가 아무리 쉽다고 하더라도 프로그래밍의 기초 지식이 전혀 없는 상태에서는 많은 어려움을 겪게 된다. 전에 파스칼을 사용해 보았다면 쉽게 델파이에 접근할 수 있겠지만 그렇지 않은 사람은 여기서 파스칼의 기본 문법을 익히도록 하자.

문법을 익히는 것은 다소 따분하고 지루하며 때로는 어렵기도 하다. 그러나 프로그래밍을 하는 사람에게는 반드시 넘어야 하는 피할 수 없는 장벽이므로 충분한 시간적 여유를 가지고 문법 공부에 임하도록 하자. 어려운 부분이기도 하지만 또한 반대로 아주 논리적이기 때문에 관심을 가지고 공부하면 밤새는 줄 모르고 공부하게 되는 부분이기도 하다.

문법이 엄격하다는 말은 난잡하지 않다는 뜻과 같다.

파스칼은 구조적 프로그래밍을 지원하며 엄격한 문법을 사용하므로 프로그래밍에 입문하는 초보자가 배우기에 적합한 언어이다. 델파이가 사용하는 오브젝트 파스칼은 표준 파스칼에 객체 지향 프로그래밍을 할 수 있도록 확장된 문법을 가진다. 그러나 파스칼이 교육용 언어라고 해서 기능이 결코 업무용이나 상업용에 뒤지지는 않는다.

## 나. 전체적인 구조

파스칼 컴파일러는 코드의 형식에는 전혀 제한을 두지 않는다. 즉 공백이 어떻게 띄워져 있는가, 첫 칸 들여쓰기가 어떻게 되어 있는가, 대문자를 쓰는가 소문자를 쓰는가 등을 상관하지 않는다. 괜히 한 줄을 비워 두어도 상관 없으며 두 개의 문장을 하나의 줄에 집어 넣어도 문법에 맞다면 컴파일을 하는데는 지장이

없다.

이런 코드 작성 방법을 프리 포맷(free format)이라 한다. 코드를 반드시 어떤 식으로 작성해야 한다는 강제가 없다는 뜻이다. 다음의 코드는 비록 띄어쓰기가 엉망이지만 아무 문제 없이 컴파일된다. 참고로 베이직 언어는 이런 프리 포맷을 지원하지 않으므로 반드시 형식에 맞게 소스를 작성해 주어야 한다.

```
procedure
TForm1.Button1Click(Sender: TObject);
var i:Integer;j:Real;
begin
i:=5;j:=
3.14;
j:=i;
end;
```

그러나 델파이가 프리 포맷을 지원한다고 해서 너무 난잡하게 소스를 작성해서는 안된다. 델파이 컴파일러가 소스를 읽는데는 아무런 지장이 없지만 사람이 보기에 너무 답답하고 가독성(readability)이 떨어지기 때문이다. 자신이 작성한 코드를 다른 사람이 읽어야 할 경우 너무 읽기에 불편하다면 코드의 의미를 쉽게 파악하기 어려워진다. 다른 사람뿐만 아니라 자기 자신도 일정한 시간이 지나면 이해하기 힘들게 되어 버린다. 그래서 가급적이면 읽기 쉽게 코드를 작성하는 것이 작업의 효율 면에서 유리하다. 위의 코드를 다음과 같이 바꾸면 코드의 의미를 한눈에 파악할 수 있지 않겠는가?

가독성이란 소스의 구조를 얼마나 빨리, 정확하게 파악할 수 있는가를 말한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:Integer;
  j:Real;
begin
  i:=5;
  j:=3.14;
  j:=i;
end;
```

코드에 주석을 달고 싶을 때는 { } 기호를 사용한다. 델파이 컴파일러는 주석을 완전히 무시하므로 어떤 내용이든지 입력해도 된다. 주석이 잘 작성되어 있으면 다음에 시간이 좀 더 지난 후에 봐도 어떤 내용인지, 왜 그렇게 코딩을 했는지 쉽게 파악할 수 있다. 주석에 한글을 사용하는 것도 물론 가능하다.

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

```
var
  i:Integer;   {정수형 변수 i 선언}
  j:Real;      {실수형 변수 j 선언}
begin
  i:=5;        {i 에 5 를 대입한다.}
  j:=3.14;     {j 에 3.14 를 대입한다.}
  j:=i;        {j 에 i 를 대입한다.}
end;
```

{ } 기호 대신에 (* *) 기호를 사용해도 주석으로 처리되며 또한 C++에서 사용하는 // 기호도 한 줄짜리 주석을 위해 사용할 수 있다.

우리가 작성할 코드는 유닛이라는 파일(*.PAS)에 저장되며 유닛 파일은 다음과 같은 구조를 가진다.

```
unit 유닛 이름
interface
  uses ...
var
  전역 변수 선언;

implementation
  uses ...

procedure 이벤트 핸들러 이름;
var
  지역 변수 선언;
begin
코드;
end;

end.
```

유닛의 형태를 선언하는 interface 부와 실제 코드를 정의하는 implementation 부로 크게 나누어진다. 이 중에서 우리가 신경 써야 할 부분은 implementation 이하에 기록되는 이벤트 핸들러의 begin 과 end 사이이다. 나머지는 대부분 델파이가 관리하며 사용자가 직접 코드를 작성해야 할 경우는 드물다. 이벤트 핸들러조차도 뼈대는 델파이가 만들어 주므로 우리는 begin 과 end 사이에 원하는 코드만 작성하면 된다. 유닛의 구조는 위에서 보인 것보다는 좀 더 복잡하지만 여기서는 이정도로만 알아두기로 하고 유닛에 대한 좀 더 상세

한 사항은 10 장에서 다루도록 하자.


파스칼에 대한 기초가 전혀 없는 사람은 도스용 파스칼을 먼저 공부하는 것도 좋다. 배포 CD의 ETC/PASCAL 디렉토리에 터보 파스칼 5.0 매뉴얼이 있으므로 문법에 정 자신이 없으면 이 매뉴얼을 먼저 읽어 보기 바란다. PASCAL.DOC는 한글 워드 5.0 포맷으로 되어 있으며 PASCAL.TXT는 텍스트 파일 포맷으로 되어 있다.

# 6-2 변수

## 가. 변수의 선언

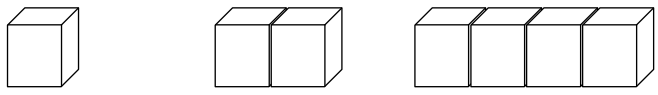
프로그램은 많은 양의 자료(data)를 다룬다. 사용자로부터 입력받은 값, 출력에 사용할 문자열, 계산에 사용되는 중간값들, 프로그램의 현재 상태를 나타내는 값, 이런 값들을 보관하기 위해 변수라는 것이 필요하다. 값을 보관하는 영역은 물론 메모리이며  값 보관을 위해 사용자가 메모리에 이름을 붙여 놓은 것이 변수이다. 또한 지금까지 사용해온 Button1.Caption, Form1.Color 등의 컴포넌트 속성도 값을 기억한다는 면에서는 변수라고 할 수 있다.

델파이에서 변수를 사용하기 위해서는 먼저 이런 이런 변수를 사용하겠다는 선언을 해주어야 한다. 그래야 델파이가 필요한 메모리 영역을 할당해 주고 프로그램에서 사용할 수 있도록 해준다. 변수를 선언하기 위해서는 변수의 형, 변수의 이름 두 가지가 필요하다.

### ■ 변수의 형(type)

변수가 기억할 값의 종류를 지정한다. 정수값을 담을 변수인지, 실수값을 담을 변수인지 아니면 문자열을 담을 변수인지를 델파이에게 알려주는 것이다. 또 같은 정수형이라도 얼마나 큰 정수를 담을 것인지, 부호를 사용하는지 하지 않는지 등을 알려 주어야 한다. 왜 이렇게 변수의 형을 알려 주어야 하는가 하면 어떤 값을 담을 것인지에 따라 필요한 메모리의 양이 달라지기 때문이며 또한 변수형에 따라 연산 방법도 달라지기 때문이다.

그림
변수에 따라 필요한 메모리 양




간단한 문자 하나를 담는 메모리와 복잡한 실수를 담는 메모리의 크기가 어떻게 같을 수 있겠는가? 그래서 델파이에게 메모리가 얼마나 필요하다는 것을 알려주기 위해 변수의 형이 필요하다. 파스칼의 변수형에는 정수형(Integer), 실수형(double), 문자열형(string) 등이 있다. 각각의 변수형에 관해서는 잠시 후에

개별적으로 알아본다.

## ■ 변수 이름(identifier)

하나의 프로그램에 여러 개의 변수가 사용되므로 변수간의 구분을 위해 각각의 변수에 이름을 주어야 한다. 변수의 이름은 사용자가 기억하기 쉬운 이름을 사용하되 명칭 규칙에 적합하게 만들어 주면 된다. 명칭(identifier)을 만드는 규칙에는 다음과 같은 것들이 있다. 이 규칙은 변수의 이름에만 국한되는 것이 아니라 컴포넌트들의 Name 속성, 유닛 이름 등에도 그대로 적용되므로 반드시 숙지하고 있어야 한다.

❶ 최소 1 자에서부터 최대 63 자까지 사용할 수 있다. 변수의 이름이 i 나 j 등과 같이 너무 짧으면 변수가 가지는 의미를 정확하게 나타내기가 어려우며 Screen_Coordinate _of_Snake_Character 등과 같이 길게 작성하면 의미를 정확하게 나타낼 수는 있지만 입력하기가 너무 귀찮아지므로 적당한 길이로 작성하는 것이 좋다. 보통 5~10 자 정도를 사용하는 것이 가장 무난하다.

❷ 영문, 숫자, _(밑줄)로 구성되며 그 외의 문자는 명칭에 사용할 수 없다. $, %, ! 등의 기호가 명칭에 들어가서는 안되며 공백을 넣어서도 안된다. 델파이가 변수, 상수, 연산자 등의 구성 요소를 구분하는 데 공백을 사용하기 때문이다. 만약 꼭 공백이 필요한 경우가 있다면 공백 대신 _(Underscore 라고 하며 우리말로 표현하면 밑줄이다)를 사용하도록 한다. Score, Sum3, Snake_X 등은 모두 적법한 변수 이름이며 Score%, Snake X, Oh!Yes 등은 변수명으로 사용할 수 없다.

❸ 첫 문자에 숫자를 사용해서는 안된다. 숫자를 변수명에 쓸 수는 있지만 변수명의 중간에만 쓸 수 있다. Average3, Me2You 는 가능하지만 3D, 2Way 등은 변수명으로 사용할 수 없다.

❹ 오브젝트 파스칼이 사용하는 예약어(Reserved Word)는 변수명으로 사용할 수 없다. 변수명으로 예약어를 쓰면 컴파일러가 의미를 정확하게 파악하지 못하게 된다. begin, end, for 등이 예약어이다.

❺ 대문자와 소문자는 구분하지 않는다. Score, score, SCORE, ScOrE 등은 모두 같은 변수명이다. 과거에는 입력의 편의를 위해 변수명을 소문자만으로 또는 대문자만으로 작성했지만 요즘은 보기 좋게 첫자만 대문자로 쓰는 경우가 많다.

그리고 너무 당연한 얘기겠지만 변수 이름끼리 중복되어서는 안된다. 이름이 같은 두 변수가 동일 범위에서 사용될 수 없다.

## ■ 변수 선언

var 는 variable, 즉 변수의 약자이다.

변수의 선언(declaration)이란 델파이에게 앞으로 이 변수를 사용하겠다는 의사를 밝히는 것이다. 변수를 선언할 때는 예약어 var를 먼저 쓰고 다음과 같이 쓴다.

**기본 형식**

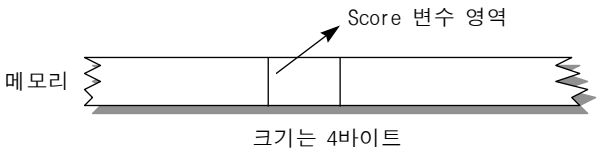
```
var
  변수이름:변수형;
```

예를 들어 정수형 변수를 Score라는 이름으로 선언한다면 다음과 같이 코드를 작성한다.

```
var
  Score:Integer;
```

이 선언에 의해 델파이는 정수형 값을 담을 수 있는 메모리 4바이트를 할당해 주며 앞으로 Score 변수에 값을 쓰면 그 메모리에 값을 기록하고 Score 변수의 값을 읽으면 그 메모리의 값을 조사해 준다.

**그림**
변수에 할당된 메모리 영역




같은 형의 변수를 여러 개 선언한다면 다음과 같이 변수 이름란에 콤마로 끊어 한꺼번에 선언한다.

```
var
  Score, NowTime, Stage:Integer;
```

물론 형이 다른 변수를 한꺼번에 선언하는 것은 불가능하며 다음과 같이 줄을 바꾸어 주어 별도로 선언해야 한다.

```
var
  Score:Integer;
  MyName:string;
```

## ■ 변수의 선언 위치

변수를 선언하는 var문의 위치는 크게 세 군데가 있으며 선언하는 위치에 따라 변수의 성격이 달라진다.

첫째, 함수 내부에서만 사용할 지역 변수(Local)를 선언할 때는 이벤트 핸들러의 begin 이전에 선언한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  {요기}
begin
end;
```

이렇게 선언된 변수는 함수 내부에서만 사용할 수 있으며 그 외의 부분에서는 사용할 수 없다. 함수 내부에서만 국지적으로 사용될 변수를 여기에 선언한다.

둘째, 프로그램 전체에서 사용할 전역 변수(Global)로 선언할 때는 코드 에디터를 직접 열어서 다음 위치에 선언해 준다. var문은 이미 입력되어 있으므로 변수형과 이름만 추가로 적어주면 된다.

```
var
  Form1: TForm1;
  {요기}
implementation
```

프로젝트를 새로 시작하고 코드 에디터를 보면 implementation이라고 되어 있는 부분이 있는데 이 이후에 이벤트 핸들러들이 위치하며 이 앞부분에 변수, 상수, 타입 선언부가 위치한다. implementation 이전에 선언되는 변수가 전역 변수이며 현재 유닛의 모든 함수에서는 물론이고 다른 유닛에서도 마음대로 사용할 수 있다.

셋째, 함수 내부에서 선언하는 것은 아니되 implementation 이후에 선언되는 변수가 있다.

```
var
  Form1: TForm1;
implementation
var
  {요기}
```

현재 유닛의 모든 함수에서 사용할 수 있으나 다른 유닛에서는 이 변수를 사용할 수 없다. 지역 변수, 전역 변수에 대해서는 다음에 유닛을 다룰 때 자세하게 거론하게 될 것이다.

# 나. 변수의 종류

파스칼이 기본적으로 제공하는 변수의 형에는 다음과 같은 것들이 있다. 아주 기초적이면서도 중요한 부분이므로 달달달 외워 버리도록 하자.

## ■ 정수형

정수값을 담는다. 정수란 부호가 있어 음수를 표현할 수는 있지만 소수점 이하의 수는 표현하지 못하는 수이다. 컴퓨터 프로그래밍에서 사용하는 대부분의 수가 정수형 변수이다. 왜 그런가 하면 원래 컴퓨터란 존재가 2진수를 사용하는 정수적인 존재이기 때문이다. 길이와 부호에 따라 다음 6가지 종류가 있다.

**표** 정수형 변수의 종류

| 데이터형 | 바이트 수 | 부호 | 표현 범위 |
|---|---|---|---|
| SmallInt | 2 | 있음 | -32768~32767 |
| ShortInt | 1 | 있음 | -128-127 |
| LongInt | 4 | 있음 | -214748347~2147483647 |
| LongWord | 4 | 없음 | 0~4294967295 |
| int64 | 8 | 있음 | $-2^{63}$~$2^{63}$-1 |
| Byte | 1 | 없음 | 0~255 |

| | | | |
|---|---|---|---|
| Word | 2 | 없음 | 0~65536 |

대부분의 경우 SmallInt형이면 정수형 표현에 큰 지장이 없지만 거리나, 금액 등의 큰 값을 다루어야 할 경우 LongInt형이 사용되며 사람의 키나 몸무게 등의 범위가 작고 절대로 음수가 있을 수 없는 경우는 Byte나 Word 등의 부호가 없는 형이 사용된다. 델파이 4에서 int64라는 64비트 크기의 정수형이 추가되었는데 이 타입은 꼭 필요해서라기보다는 미래의 64비트 운영체제를 위해 추가된 것이다. 델파이 자체가 이 타입을 완벽하게 지원하지 않으므로 사용하려면 여러 주의 사항을 숙지하고 신중하게 사용해야 한다.

위에서 보인 정수형은 고정된 길이를 가지는 정수형이지만 이런 고정된 길이보다는 다음 두 가지 일반적인(Generic) 정수형이 더 많이 사용된다. 특히 Integer형이 가장 많이 사용되는 정수형이다.

**표**

일반적인(Generic) 정수형 변수

| 데이터형 | 바이트 수 | 부호 | 표현 범위 |
|---|---|---|---|
| Integer | 2 | 있음 | -32768~32767 |
| | 4 | | -214748347~2147483647 |
| Cardinal | 2 | 없음 | 0~65536 |
| | 4 | | 0~2147483647 |

이 두 가지 일반형 정수 타입은 그 크기가 고정되어 있지 않으며 운영체제나 CPU의 종류에 따라 길이가 가변적이다. 즉 16비트 CPU나 16비트 운영체제에서 Integer형은 16비트 변수가 되며 32비트 운영체제에서는 32비트 변수가 된다. 델파이 4.0은 윈 95나 NT와 같은 32비트 운영체제에서 실행되므로 Integer형의 크기는 거의 항상 4바이트라고 생각해도 틀림이 없다. 반면 델파이 1.0은 16비트 개발 툴이므로 Integer형의 크기가 2바이트로 고정되어 있다. 혹시 다음에라도 델파이가 16비트 개발 환경으로 이식된다면 Integer형은 16비트 크기가 될 것이다. 또한 64비트 운영체제하에서는 (장담할 수는 없지만)64비트가 될 지도 모른다.

표현 범위에 따라 이렇게 다양한 데이터형을 준비해 놓은 이유는 사용하는 목적에 꼭 맞게 메모리를 알뜰하게 사용하기 위해서이다. 사람의 키를 저장하는 변수를 LongInt형으로 만들어 4바이트를 쓰는 것보다는 Byte형으로 만들어 1

바이트만 쓰는 것이 메모리 효율 면에서나 속도 면에서 유리하기 때문이다. 이 차이가 별로 커 보이지 않을지도 모르겠지만 대규모의 배열이나 데이터 베이스 테이블을 디자인할 때는 엄청난 차이를 보일 수도 있다.

정수형 상수는 -55나 8906299와 같이 우리가 일상 생활에서 늘 표기하는 대로 사용한다. 좀 특별한 형태로는 16진수 표기법이 있는데 숫자 앞에 $기호를 붙여준다. 이미지 데이터나 비트별로 의미를 가지는 데이터는 10진수보다는 16진수로 표기하는 것이 더 편리하다. 10진 상수와 16진 상수는 표기법이 다를 뿐이며 본질적으로 같은 숫자를 표현하므로 별도로 변수형이 구분되지는 않는다. 십진수 12와 16진수 $C는 메모리에 기억될 때는 결국 똑같이 이진수로 1100이다.

참고로 일반적인 프로그래밍 언어에서는 "정수형"이라는 용어를 사용하지만 파스칼에서는 이 용어 대신에 "서수형"이나 "순서형"이라는 용어를 많이 사용한다. 원문으로 Ordinal Type이라고 하는데 이는 각각의 정수끼리 순서가 있어 대소 구분이 명확하고 두 숫자 사이에 몇 개의 숫자가 있는지를 계산할 수 있다는 뜻이다. 즉 2는 3보다도 작다는 것이 명확하고 5와 8사이에는 2개의 요소가 더 있다는 것이 계산 가능하다는 뜻이다. 하지만 이 책에서는 순서형이라는 용어 대신 좀 더 일반적인 정수형이라는 용어를 계속 사용할 것이다. 여러분들은 도움말에서 Ordinal Type이라는 말이 나오면 정수형을 뜻한다고 생각하기 바란다.

## ■ 실수형

소수점 이하의 실수값을 표현할 수 있는 데이터형이다. 정수형과 마찬가지로 길이에 따라 5가지 종류가 준비되어 있는데 단순히 표현할 수 있는 수의 크기만 다른 것이 아니라 정밀도도 차이가 있다. 즉 소수점 몇 자리까지 믿을 수 있는 것인지를 지정하는 유효 자릿수가 데이터형에 따라 다르다.

**표** 실수형 변수의 종류

| 데이터형 | 바이트 수 | 유효 자릿수 | 표현 범위 |
|---|---|---|---|
| Single | 4 | 7-8 | $1.5*10^{-45}..3.4*10^{38}$ |
| Double | 8 | 15-16 | $5.0*10^{-324}..1.7*10^{308}$ |
| Extended | 10 | 19-20 | $3.4*10^{-4932}..1.1*10^{4932}$ |
| Comp | 8 | 19-20 | $-2^{63}+1..2^{63}-1$ |
| Real | 8 | 15-16 | $5.0*10^{-324}..1.7*10^{308}$ |
| Real48 | 6 | 11-12 | $2.9*10^{-39}..1.7*10^{38}$ |

실수형이 필요할 경우는 보통 Single을 사용하며 큰 값을 표현할 경우 Double 정도면 충분하다. 정말로 정확하고 무지막지하게 큰 값이 필요하다면 Extended형을 사용해야겠지만 이 형은 실제로 거의 사용되지 않는다. 표기는 10의 4900승이라고 간단하게 할 수 있지만 이게 어디 인간이 생각할 수 있는 숫자인가 말이다. Real형은 구형의 볼랜드 파스칼 컴파일러와의 호환성을 위해서 제공되는 것인데 델파이 4.0부터 Real형은 Double형과 완전히 같아졌다. 만약 구형 볼랜드 파스칼과의 호환성에 문제가 있다면 Real48타입을 대신 사용하면 되겠으나 이 타입은 가급적 사용하지 않는 것이 좋다. 왜냐 하면 Real48타입은 실수 표기에 관한 국제 표준을 따르지 않기 때문에 잠재적인 문제를 유발할 수 있다.

실수형 상수 표기법에는 두 가지 방법이 있다. 일상 생활에서 사용하는 3.1415 등과 같은 표기법을 고정 소수점 표기법이라 하고 3.14e3 등과 같이 10의 거듭승으로 표기하는 방법을 공학적 표기법 또는 부동 소수점 표기법이라 한다.

## ■ 논리형(Boolean)

참과 거짓의 두 가지 상태 중 하나의 값만을 가지며 참일 경우 True, 거짓일 경우 False의 값을 가진다. 논리형도 메모리에 기억될 때는 정수로서 기억된다. 거짓일 경우가 정수 0이며 참일 경우가 정수 1이지만 일반적으로 0이면 거짓이 되고 0 이외의 다른 값(nonzero)이면 모두 참으로 평가된다. 데이터 형은 Boolean이다.

```
var
  IsDie:Boolean;
```

Boolean형에 대입될 수 있는 값은 True, False 두 가지뿐이다. 기타 논리형 타입에는 길이에 따라 WordBool, LongBool 등이 있지만 다른 언어와의 혼합 프로그래밍에만 사용될 뿐 델파이에서는 거의 사용될 일이 없다.

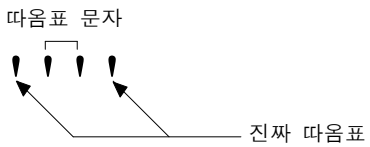
## ■ 문자형

문자 하나를 기억하는 데이터형이다. 'A', 'h', '+' 등의 값들을 기억한다. 데이터형의 이름은 Char형이다.

```
var
  Ch:Char;   {문자형 변수 Char 선언}
begin
  Ch:='T';   {Ch 에 문자형 상수 'T'를 대입}
```

Char형 외에도 1바이트 고정 길이를 가지는 AnsiChar형과 2바이트 고정 길이를 가지는 유니코드용의 WideChar형이 델파이 2.0에서 추가되었다. 그러나 윈95, 윈98에서 아직 유니코드를 제대로 지원하지 않으므로 Char형 외에는 특별히 사용할 일이 없다.

문자 상수는 홑따옴표 하나로 나타내며 홑따옴표 안에 나타내고자 하는 문자 상수를 적어 준다. 만약 홑따옴표 자체를 나타내려면 홑따옴표 두 개를 연속해서 사용한다. '''' 요렇게. 양쪽의 '는 따옴표를 나타내고 가운데 있는 ''는 홑따옴표 문자를 나타낸다.



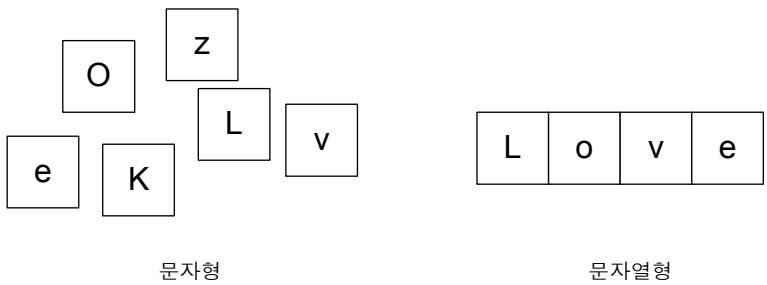

키보드에 없는 제어문자는 #과 제어문자 코드를 사용하여 표기하되 #과 코드 사이에 공백이 없어야 한다. 예를 들어 #13은 Enter 코드를 나타내며 이 코드가 나오면 다음 줄로 개행한다. Label1.Caption:='첫 번째 줄'+#13+'두 번째 줄'; 과 같이 코드를 작성하면 레이블에 여러 줄의 문자열을 출력하는 것도 가능하다.

## ■ 문자열형

문자열(String)이란 문자 여러 개가 모여서 이루어진 것이며 문자열형 변수는 'kim', 'love', 'Good Morning!' 등의 문자열값을 기억한다.

**그림**

문자형 변수 여러 개가 모이면 문자열이 된다.



데이터형 이름은 String이며 다음과 같이 선언한다.

```
var
 MyName:String;
```

이렇게 선언된 문자열형 변수에는 어떤 문자열이든지 대입될 수 있다.

```
MyName:='Kim Sang Hyung';
```

String형 변수의 길이 한계는 2G바이트로서 실질적으로 문자열의 한계가 무한하다고 할 수 있다. 2GB의 메모리 용량을 가진 PC는 적어도 금세기 안(얼마 남지 않았지만)에는 등장하지 않을 것이다. String형은 실행중에 길이를 스스로 변경할 수 있도록 되어 있다. 즉 15자 길이의 문자열이 대입되면 그 길이에 맞게 메모리를 할당하고 100자 길이의 문자열이 대입되면 늘어난만큼 메모리를 자동으로 재할당한다.

또한 다음과 같이 [ ] 괄호 안에 필요한 만큼의 길이를 밝혀주어 메모리를 절약하는 방법도 있다.

```
var
 MyName:String[30];
```

이렇게 선언하면 MyName이라는 문자열 변수는 30문자만 기억할만한 크기로 할당된다. 메모리는 아무리 많아도 부족한 것이므로 최대한 아껴야 한다. 특히 윈도우즈와 같이 여러 개의 프로그램이 동시에 실행되는 멀티태스킹 시스템에서는 리소스를 최대한 아껴 쓰는 것이 프로그램 간의 예의를 지키는 것이다. 문자열 자체가 동적으로 메모리를 할당하는 포인터이므로 이런 방법을 쓸 필요가 별로 없지만 레코드 배열의 경우는 아직도 이런 방식을 사용한다.

문자열형 상수는 문자 상수와 마찬가지로 홑따옴표를 사용한다. 'Korea' , 'Insert Your Diskette' 등이 문자열 상수의 예이다. 문자열 상수 안에 홑따옴표를 쓰고 싶으면 홑따옴표 두 개를 연이어 써 주면 된다. 'Let''s Go' 등과 같이 표기하면 기억되는 문자열은 Let's Go가 된다. 왜 'Let's Go'라고 쓰면 안되는지는 12초만 생각해 보면 알 수 있다.

이 외에도 델파이가 사용하는 데이터형에는 포인터형과 PChar형, 그리고 사용자 정의형이 있다. 일단 여러분은 여기서 소개한 기본형을 먼저 익히고 난 후에 관련 부분에서 나머지 데이터형을 천천히 살펴보기로 하자.

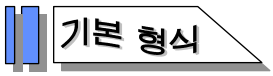
# 다. 상수

상수(Constant)란 변수(Variable)에 반대되는 말이며 한번 값이 정해지면 절대로 바뀔 수 없는 수를 말한다. Score란 변수는 실행중에 값이 증가하기도 하고 다른 값이 대입될 수도 있지만 상수 123은 언제까지고 123일 뿐이다. 상수는 변수에 값을 대입하거나 함수의 인수로 사용된다.

변수에서와 마찬가지로 상수에도 데이터형이 있다. 상수의 데이터형은 선언에 의해 정해지는 것이 아니라 겉모양과 쓰여진 위치의 문맥에 의해 결정된다. 우선 정수형과 실수형은 소수점이 있는가 없는가에 따라 구분한다. -1234나 8906299는 소수점이 없으므로 정수형이며 3.14나 -123.456은 소수점이 있으므로 실수형이다.

123, 3.14 등과 같이 명확하게 상수값으로 표현되는 상수 외에도 명칭으로 표현되는 상수를 선언할 수도 있다. 예약어 const 다음에 상수 명칭을 선언해 준다.

const 문으로 상수를 선언할 때 좌변과 우변의 구분에 =기호를 사용한다. 대입문인 :=이 아니므로 주의하기 바란다.




```
const
  상수이름=상수값;
```

구체적인 예를 들어 보자.

```
const
  Pie=3.14159265;
  Days=365;
```

이 선언에 의해 Pie라는 명칭은 실수값 3.14159265를 가지는 상수로 선언되며 실행중에 값이 변경될 수 없다. 마찬가지로 Days는 365라는 정수 상수값을 가지며 365 이외의 값을 가지지 못한다. 상수를 직접 써야 할 것을 별도의 명칭을 만들어서 사용하는 것 뿐이다. 3.14159265라는 상수가 사용될 곳에 대신 Pie를 사용하면 된다. 그럼 직접 상수를 사용할 것이지 왜 명칭까지 만들어가며 상수를 선언해서 사용할까? 그 이유는 대체로 다음 두 가지 정도를 생각할 수 있다.

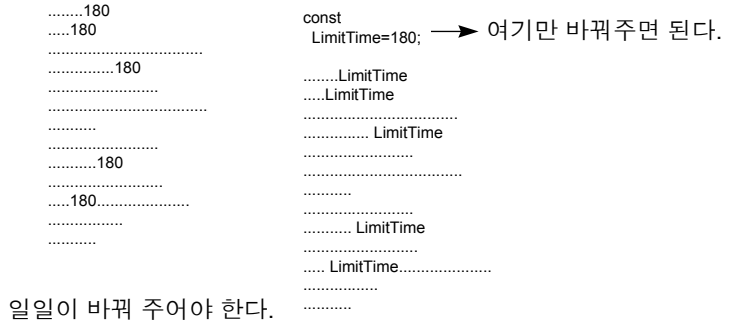
첫째, 개발 과정에서 수정을 용이하게 하기 위해서 상수를 사용한다. 퍼즐 게임을 하나 만들었는데 퍼즐을 푸는 제한 시간을 180초로 설정하고자 한다. 이 게임 소스에는 180이라는 상수가 여러 번 사용될 것이다. 그런데 게임을 만들다가 제한시간을 300초로 늘려주고 싶다고 하자. 이때는 소스를 죄다 뒤져 상수 180을 300으로 다 바꾸어 주어야 하는 성가신 작업을 해야 할 뿐만 아니라 실수로 하나를 발견하지 못하고 그대로 180으로 내버려 두었다가는 치명적인 에러의 원인이 된다. 이럴 때 상수를 선언하여 사용한다.

```
const
 LimitTime=180;
```

애초부터 이렇게 상수를 선언해 두고 소스에는 180이라는 상수를 직접 쓰지 않고 LimitTime이라는 명칭을 사용했다면 중간에 제한 시간을 바꿀 때 소스는 건드릴 필요없이 상수 선언 부분만 바꾸어 주면 된다. 개발 과정에서 자주 변경되며 소스의 여기 저기에서 사용되는 상수는 이렇게 명칭으로 선언해 두고 쓰는 것이 좋다.

**그림**

상수를 사용할 경우의 이점




둘째, 상수를 직접 쓰지 않고 명칭을 사용하면 상수의 의미를 좀더 잘 나타내준다. 그냥 180이라고 써 두면 도대체 이게 왜 180인지, 어떤 의미로 180이라고 썼는지 명확하게 나타내지 못하지만 LimitTime이라는 명칭을 사용하면 제한 시간을 의미한다는 것을 잘 알 수 있다. 성적 처리 프로그램에서 전체 학생 수가 1286명이라면 직접 1286이라는 상수를 쓰는 것보다 명칭을 사용하여 TotalStudent라고 쓰는 것이 더 쉽지 않겠는가? 특히 여러 사람이 공동 작업을 할 때는 자기가 만든 상수의 의미를 다른 사람도 쉽게 알 수 있도록 해 주어야 한다.

상수도 변수와 마찬가지로 데이터형을 가진다. 그러나 명확하게 요건 정수형, 요건 실수형이라고 밝히지는 않으며 상수 선언시 초기화값의 형태로 데이터형을 판단한다. 앞에서 예로 보인 LimitTime은 180이라는 초기화값이 주어졌으므로 정수형 상수가 된다.

# 라. 연산자

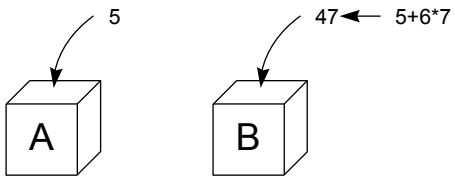
값을 기억하는 변수는 프로그래밍의 주 대상이며 변수를 제어하는 주된 수단은 연산자(Operator)이다. 연산자를 통해 변수가 가진 값을 변경시키고 조작하고 결합해 낸다.

## ■ 대입문

대입문이란 변수의 값을 정의하는 연산문이며 이때까지 기초적인 실습에서 많이 사용해 왔다. 대입문은 "좌변:=우변"의 형식으로 사용하며 좌변에는 값을 담을 수 있는 변수나 컴포넌트의 속성이 온다. 사실 컴포넌트의 속성은 값을 담는다는 면에 있어서는 변수와 거의 같은 개념이다. 물론 문법적으로 엄밀히 따지면 완전히 같지는 않다. 대입문의 우변에는 상수가 곧바로 오거나 하나의 값을 만들어 내는 연산식(Expression)이 온다. 다음이 대입문의 예이다.

```
A:=3;
B:=5+6*7;
Button1.Caption:='누르세요';
```

**그림**
변수에 값을 대입하는 대입 연산자




대입문에서 한 가지 주의할 것은 좌변과 우변의 데이터형이 같아야 한다는 점이다. 정수형 변수에는 정수값을 대입하고 문자열 변수에는 문자열 값을 대입해야 한다. 즉 A가 정수형 변수일 때 A:=5; 나 A:=3+4;는 가능하지만 A:=3.14; 나 A:='korea' 등의 대입문은 사용할 수 없다는 뜻이다. 속성에 대해서도 마찬가지로 Button1.Caption:=23; 따위의 대입은 에러를 유발시킨다.

## ■ 사칙 연산

사칙 연산이란 더하고, 빼고, 곱하고, 나누는 가장 기본적인 연산자이다. 초등학교때부터 배워온 것이므로 간단히 표로 정리하기로 한다.

**표**
사칙 연산자

| 연산자 | 의미 | 예 |
|---|---|---|
| + | 더하기 | A:=1+2; |
| - | 빼기 | A:=2-1; |
| * | 곱하기 | A:=2*3; |
| / | 나누기 | F:=3/2; |
| div | 정수 나누기 | A:=3 div 2; |
| mod | 나머지 | A:=5 mod 2; |

나누기 연산자는 정수 나눗셈을 하는 연산자와 실수 나눗셈을 하는 연산자 두 가지가 분리되어 있다. 정수 나누기란 소수점 이하를 계산하지 않고 정수 부분까지만 나눗셈을 수행한다. 실수 나누기인 3/2는 소수점 이하까지 계산하여 1.5가 되지만 3 div 2는 소수점은 고려하지 않기 때문에 결과는 정수값인 1이 된다.

문자열끼리 연결하는 +연산자는 덧셈에 사용하는 +연산자와 모양만 같은 다른 연산자이다.

+ 연산자는 정수끼리의 덧셈과 실수끼리의 덧셈에 모두 사용할 수 있으며 뿐만 아니라 문자열끼리 연결할 때도 +연산자를 사용한다.

```
Str1:='kanam'+' book';   {결과는 kanam book 이 된다.}
```

## ■ 부호 연산자

+, - 연산자는 부호 연산자로도 사용된다. -는 상수 앞에 쓰여 상수가 음수임을 나타낸다. 다음 연산식을 보자.

```
A:=-B-C;
```

이 식에서 앞에 사용된 -는 음수를 나타내는 부호 연산자이며 뒤의 -는 뺄셈 연산자이다. +도 양수를 나타내는 부호 연산자로 사용할 수 있지만 수학에서와 마찬가지로 보통 생략한다.

## ■ 연산 순위

이 외에도 파스칼에는 논리 연산자, 관계 연산자 등의 여러 가지 연산자가 제공되는데 이 연산자들에 대해서는 이어지는 조건문과 관련 부분에서 따로 알아보기로 한다. 여러 개의 연산자가 한 연산식에 같이 사용될 때는 연산 우선 순위에 따라 연산이 이루어진다. 파스칼은 다음과 같이 4단계의 연산 우선 순위를 정의하고 있다.

**표**
연산 우선 순위

| 순위 | 연산자 |
|---|---|
| 1 | +,-(부호), @, not |
| 2 | *,/,div,mod,as,and,shl,shr |
| 3 | +.-.or, xor |
| 4 | =, <>, <, >, <=, >=, in, is |

그래서 다음 두 식은 연산 순위에 따라 다른 결과를 낸다.

```
A:=1+2*3;   {결과는 7}
A:=1*2+3;   {결과는 5}
```

연산 순위에 의해 항상 곱셈이 덧셈보다 먼저 이루어지기 때문이다. 만약 연산 순위를 강제로 변경하려면 괄호를 사용한다. A:=(1+2)*3;과 같이 괄호로 묶어주면 덧셈이 곱셈보다 먼저 수행되어 결과는 9가 된다.

# 6-3 제어문

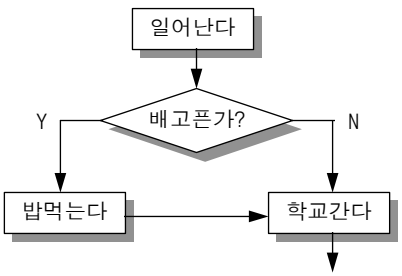
제어문이란 프로그램의 흐름을 통제하는 명령이다. 프로그램은 소스에 나타난 순서대로 물 흐르듯이 위에서 아래로 자연스럽게 실행되지만 제어문에 의해 특정 부분을 반복하거나 선택적으로 실행하기도 한다. 제어문을 잘 사용할 줄 알아야 간결하고 효율적인 프로그램을 짤 수 있다. 제어문은 프로그래밍의 가장 기본이면서도 쉽게 정복되는 것이 아니다. 단순히 암기한다고 해서 되는 것이 아니라 논리적인 사고 능력을 요구하고 창조력이 필요하다. 여러 예제에서 나타나는 패턴을 익혀두고 조금씩 변형하여 응용하다 보면 매끄럽게 프로그램을 작성하는 방법을 터득하게 될 것이다.

## 가. 조건문

조건문이란 특정 조건의 진위에 따라 명령의 실행 여부를 결정하는 제어구조이다. 프로그램의 흐름을 통제하는 가장 기본적인 제어문이며 프로그램에 판단 능력을 부여한다. 일상 생활에서 조건문의 예를 들어보자.

**그림**

일상 생활에서 볼 수 있는 조건 판단




이 경우 "배고픈가?"라는 질문이 조건이 되며 이 조건의 참/거짓 여부에 따라 밥을 먹기도 하고 그냥 식사를 생략하고 학교로 가기도 한다. 델파이 프로그래밍에서 명령 수행 여부를 결정하는데 사용되는 조건이란 주로 변수나 속성값을 평가하는 평가식이다. 델파이에서 조건문은 if문을 사용하며 기본 형식은 다음과 같다.

### 기본 형식

if 조건 then 명령;

6jang
if1

실습을 위해 아주 간단한 예제를 만들어 보자. 새로운 프로젝트를 시작하고 레이블, 에디트, 버튼 컴포넌트를 각각 하나씩 폼에 배치하고 속성은 디폴트를 그대로 사용하기로 한다.


에디트에 문자열을 입력하고 버튼을 누를 때 에디트에 입력된 문자열이 'orange'이면 레이블의 캡션을 'orange'로 바꾸려고 한다. 에디터에 입력된 문자열이 무엇인가의 조건에 따라 레이블의 캡션을 바꿀 것인가 아닌가를 결정한다. 버튼을 누를 때 조건 점검을 해야 하므로 버튼의 OnClick이벤트를 작성해야 한다. 폼에서 버튼을 더블클릭한 후 다음 코드를 작성하도록 하자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if Edit1.Text='orange' Then
   Label1.Caption:='orange';
end;
```

Edit1.Text:='orange' 즉 "에디트에 입력된 문자열이 'orange'이다"가 조건이 되며 이 조건의 진위 여부에 따라 label1.caption:='orange';가 실행되거나 무시된다. 직접 실행해 보아라.

조건식에 사용되는 식은 주로 변수의 값(또는 컴포넌트의 속성)을 비교해 보는 식이며 두 개의 값을 비교하는 관계 연산자가 사용된다. 관계 연산자는 수학에서 사용하는 기호와 유사하다.

**표**
관계 연산자

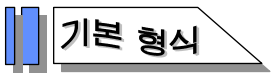
| 연산자 | 의미 |
|---|---|
| A=B | A와 B는 같다. |
| A<>B | A와 B는 다르다. |

| | |
|---|---|
| A<B | A가 B보다 작다. |
| A>B | A가 B보다 크다. |
| A<=B | A가 B보다 작거나 같다. |
| A>=B | A가 B보다 크거나 같다. |

같다를 나타내는 '=' 관계 연산자와 대입문에 사용되는 ':='은 혼돈의 소지가 있으므로 정확하게 구분을 하도록 하자. '=' 관계 연산자는 좌변과 우변이 같다는 형용사이며 ':=' 대입 연산자는 좌변을 우변과 같게 만들라는 동사다.

## ■ Else 문

if문을 조금 더 확장해 보면 다음과 같다.

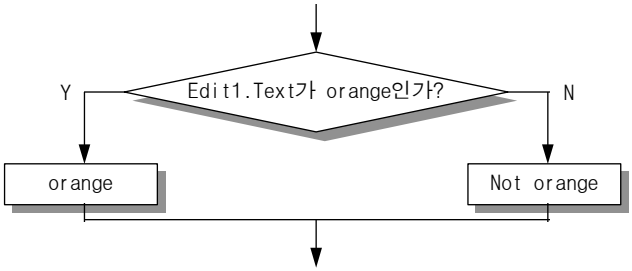



if 조건 then 명령1 else 명령2;

이 형식은 조건문이 거짓일 경우의 처리까지 함께 기술하여 참일 경우와 거짓일 경우 실행되는 명령을 다르게 한다. 조건이 참이면 명령1이 실행되고 거짓일 경우는 명령2를 실행한다. 앞의 예제에서 에디트에 입력된 문자열이 'orange'가 아닐 경우에 레이블의 캡션을 Not orange로 바꾸도록 해보자. 버튼의 이벤트 핸들러를 다음과 같이 수정한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Edit1.Text='orange' Then
    Label1.Caption:='orange'  {여기에 세미콜론이 없다}
  else
    Label1.Caption:='Not orange';
end;
```

else문에서 한 가지 주의할 것은 명령1 다음에 세미콜론이 없다는 점이다. if~else문에서 else문이 완전히 끝나기 전에 세미콜론을 붙여서는 안되며 붙이게 되면 에러가 발생한다. C 프로그래밍을 하던 사람이 흔히 이런 실수를 하므로 주의하도록 하자. 위의 예제를 순서도로 그려보면 다음과 같다.




else문은 if와 짝을 이루어 사용되는 명령이므로 단독으로 사용할 수 없으며 반드시 if 다음에 와야 한다.

## ■ 복합문

6jang
if2

앞의 예제를 다시 수정하여 조건에 따라 폼의 바탕색도 같이 바꾸도록 수정해 보자. 에디트에 'orange'가 입력되면 바탕색을 노란색으로 바꾸고 아니면 빨간색으로 바꾼다. 조건에 따라 레이블의 캡션 변경 명령과 폼의 색상 변경 명령이 동시에 수행되어야 하는데 이때는 begin과 end로 두 명령을 싸 주어야 한다. 왜냐하면 if문의 then 다음에는 하나의 명령만 올 수 있기 때문이다. "if 조건 then 명령"이 기본 형식이지 "if 조건 then 명령들"이 기본 형식이 아니기 때문이다. 만약 다음과 같이 코드를 작성했다고 해보자.

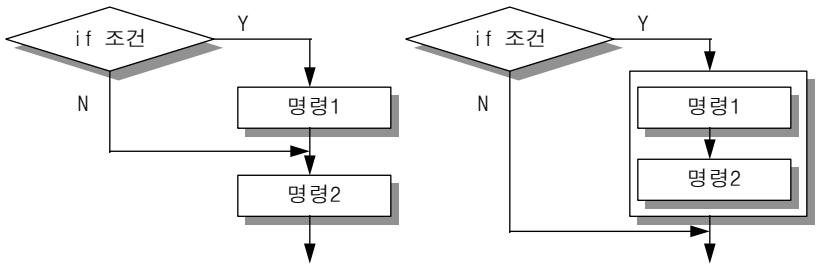
```
procedure TForm1.Button1Click(Sender: TObject);
begin
if Edit1.Text='orange' then
  Label1.Caption:='orange';  여기서 if 문은 끝이 났다.
  Form1.Color:=clYellow;    이 명령은 조건과 상관없이 실행된다.
else                        이 명령은 에러가 된다.
  Label1.Caption:='Not orange';
  Form1.Color:=clRed;
end;
```

if 문은 레이블의 캡션을 'orange'로 바꾸는데서 끝이 나고 폼의 색상을 변경하는 명령에게까지는 영향을 미치지 않는다. 또한 그 다음 줄의 else는 if와는 상관없이 홀로 사용되었으므로 문법적으로 에러가 된다. begin과 end로 이 둘을 묶어주면 델파이는 begin과 end 사이의 모든 명령을 하나의 명령인 것처럼 취급한다. begin과 end로 묶여진 명령의 덩어리를 블럭(block)이라고 하며 파스칼 문법에서 아주 중요한 부분을 차지하므로 잘 알아두도록 하자. 블럭을 사용하여 완성시킨 예제는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if Edit1.Text='orange' then
   begin
   Label1.Caption:='orange';
   Form1.Color:=clYellow;
   end
 else
   begin
   Label1.Caption:='Not orange';
   Form1.Color:=clRed;
   end
end;
```

레이블의 캡션을 바꾸는 명령과 폼의 색상을 바꾸는 명령이 블럭으로 묶여져 한 명령인 것처럼 취급된다. 다음에 블럭으로 묶은 경우와 묶지 않은 경우를 순서도로 비교해 보자.

그림
블럭을 묶은 경우와 그렇지 않은 경우




블럭을 싸지 않은 경우 블럭으로 묶은 경우

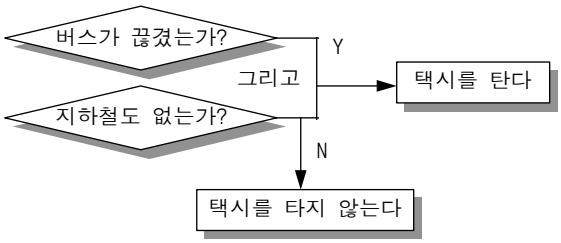
블럭으로 묶은 부분이 하나의 문장으로 취급되며 조건의 진위에 따라 두 명령이 한꺼번에 실행되거나 실행되지 않게 된다. if문 뿐만 아니라 순환문, 선택문에서도 하나의 명령 덩어리로 취급해야 할 명령들은 블럭으로 묶어주어야 한다. 5장의 Editor2 예제에서도 블럭을 사용해 본 적이 있는데 다시 5장으로 돌아가 살펴보기 바란다. 블럭은 파스칼 문법을 이해하는데 큰 비중을 차지하므로 잘 알아두도록 하자.

## ■ 조건 결합

조건문에서 꼭 조건이 하나일 필요는 없다. 동시에 두 개 이상의 조건을 평가하여 명령 수행 여부를 결정하는 것도 물론 가능하다. 일상 생활에서 예를 든다

면 “버스도 끊기고 지하철도 없으면 택시를 탄다” 등의 경우가 여기에 해당된다.

**그림**
두 가지 조건이 동시에 만족해야 하는 AND 연결



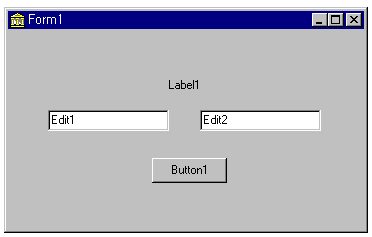

두 개의 조건을 연결하는 데는 여러 가지 방법이 있으며 어떻게 조건을 연결할 것인가를 논리 연산자로 지정한다.

**표**
논리 연산자

| 논리 연산자 | 의미 |
|---|---|
| A and B | A와 B가 동시에 참이어야 참이다. |
| A or B | A나 B중 하나만 참이어도 참이다. |
| A xor B | A, B의 진리값이 달라야 참이다. |
| not A | A가 참이면 거짓, 거짓이면 참이 된다. |

6jang
if3

논리 연산자를 사용해 예제를 변형시켜 보자. 두 개의 조건을 만들기 위해 에디트 컴포넌트를 하나 더 추가시켜 다음과 같이 만든다.




그리고 논리 연산자를 사용하여 코드를 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if (Edit1.Text='orange') and (Edit2.Text='yes') then
   Label1.Caption:='orange'
 else
   Label1.Caption:='Not orange';
```

```
end;
```

실행시켜 보면 edit1에 'orange', edit2에 'yes' 두 조건이 동시에 만족해야 레이블의 캡션이 orange가 되며 둘 중 하나라도 거짓일 경우는 Not orange가 된다. and를 or로 바꾸었을 때 두 조건 중 하나라도 참이 되면 레이블의 캡션은 orange가 된다. 조건을 연결할 때 주의할 점은 논리 연산자 양쪽의 조건을 모두 괄호로 싸 주어야 한다는 점이다. 괄호로 싸 주지 않고 다음과 같이 하면 에러로 처리된다.

```
if edit1.text='orange' and edit2.text='yes' then
  label1.caption:='orange
```

꼭 괄호로 싸 주어야 하는 이유는 논리 연산자가 관계 연산자보다 우선 순위가 높아 식을 평가하기도 전에 조건 결합이 먼저 일어나기 때문이다. 조건식을 괄호로 싸 주어야 식을 평가한 후에 전체적인 결과를 계산해 내게 된다.


① 제어문:프로그램의 흐름을 통제한다.
② if 조건문:조건이 참일 때만 명령을 실행한다.
③ 블럭:begin, end로 싸여진 명령의 집합
④ 논리 연산자:두 개 이상의 조건을 결합한다.

# 나. 반복문

컴퓨터가 하는 일 중 반복되는 부분이 많다. 사실 컴퓨터란 놈이 제일 잘하는 일 중의 하나가 똑같은 일을 몇 번이고 불평없이 지치지도 않고 계속해 대는 일이다. 프로그램을 작성하다 보면 비슷비슷한 부분을 반복해야 할 경우가 무척이나 많다. 1번부터 60번까지 학생의 성적을 똑같은 방법으로 평균을 내는 일, 화면의 처음부터 끝까지 검정색을 모두 흰색으로 바꾸는 일, 수십만 개의 문자열을 비교하여 원하는 문자열을 찾아내는 일 등 반복 투성이다. 델파이는 세 가지 종류의 반복문을 지원한다.

## ■ for문

for 문의 제어 변수로 실수는 사용할 수 없으므로 C를 사용하던 사람은 주의하기 바란다.

for문은 변수의 값 하나를 기준으로 해서 시작값부터 끝값까지 작업을 반복

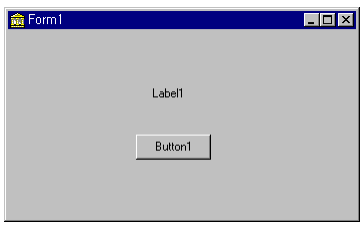
하는 대표적인 반복문이다. 이때 for문의 반복 기준이 되는 변수를 제어 변수라고 하며 정수형 변수가 사용된다. 기본 형식을 익히고 난 후 실습을 해 보자.

### 기본 형식

for 제어변수:=시작값 to 끝값 do 반복할 명령;

6jang
for1

for문이 시작되면 제어 변수는 먼저 시작값으로 설정되며 1씩 증가하면서 계속 명령을 반복하다가 제어 변수가 끝값이 되면 실행을 멈춘다. 반복되는 횟수는 끝값-시작값+1회이다. 실습을 위해 새로운 프로젝트를 시작하고 버튼과 레이블 하나만을 폼에 배치해 두자. 속성은 디폴트를 그대로 사용한다.

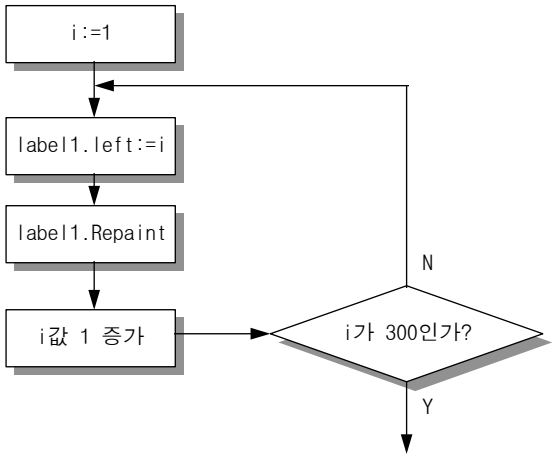



버튼 컴포넌트를 더블클릭하여 다음 코드를 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  for i:=1 to 300 do
    begin
    Label1.Left:=i;
    Label1.Repaint;
    end;
end;
```

여기서 사용된 제어 변수는 i이다. 시작값이 1이며 끝값이 300이므로 i가 1에서부터 1씩 증가하여 300까지 반복한다. 명령의 내부에서는 레이블의 Left속성에 i를 대입하여 레이블이 좌표 1에서부터 300까지 이동한다. repaint 메소드는 레이블을 다시 그리도록 하며 위치가 바뀌면 바뀐 위치에 다시 그려 주어야 한다. 반복해야 할 명령이 두 개이므로 여기서도 블럭이 사용되었다. 이렇게 반복문에 의해 일정 횟수 동안 반복되는 부분을 루프(loop)라고 한다. 순서도로 진행

과정을 나타내 보면 다음과 같다.




만약 레이블을 우측 300에서 좌측의 1로 움직이고 싶으면 초기값을 300으로 주고 종료값을 1로 바꾸어 주면 된다. 이때 시작값과 끝값 사이에 삽입되는 to는 downto로 바꾸어 준다. 값이 증가할 때는 to를 사용하지만 값이 감소할 때는 downto를 사용한다.

```
for i:=300 downto 1 do
  begin
  Label1.Left:=i;
  Label1.Repaint;
  end;
end;
```

## ■ 중첩 루프

6jang
for2

반복이란 한 가지 일을 정해진 횟수만큼 여러 번 실행하는 것이다. 그런데 그 한 가지 일이 또 반복을 하는 일인 경우가 있다. 즉 반복을 반복하는 일인데 예를 들자면 과목1~과목10까지 합산하는 반복 처리를 1번 학생부터 60번 학생까지 또 반복하는 일이다. 이렇게 반복이 겹치는 것을 중첩(nesting)이라고 하며 델파이에서는 for루프를 두 번 연거푸 사용해 주면 된다. 앞의 예제를 변경하여 레이블이 1~300까지 움직이도록 하는 처리를 다섯 번 반복하도록 해보자. 두 개의 for루프를 사용해야 하며 하나의 for루프에는 하나의 제어 변수가 꼭 필요하므로 제어 변수를 하나 더 만들어 주어야 한다. 제어 변수 j를 만들고 j를 1에서 5

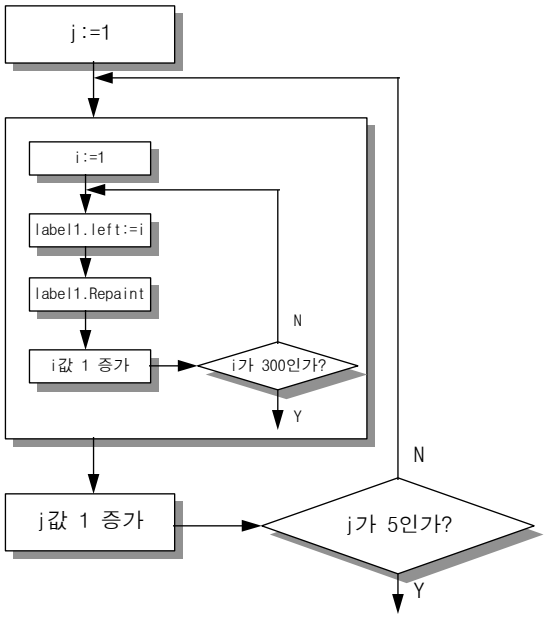
까지 변화시키면서 j가 한 번 변할 때마다 i가 1~300까지 변하도록 해보자. 코드는 다음과 같이 된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i,j:integer;
begin
  for j:=1 to 5 do
    for i:=1 to 300 do
      begin
      Label1.Left:=i;
      Label1.Repaint;
      end;
end;
```

처음 j 루프가 시작되고 j는 시작값 1이 되며 곧이어 i 루프가 시작되고 i는 초기값 1이 된다. i루프가 반복되면서 i가 1에서 300까지 변하고 레이블이 좌에서 우로 이동한다. 레이블이 300까지 완전히 이동하면 i루프가 일단 끝이 난다. i루프가 끝나면 i루프를 싸고 있는 j루프가 계속해서 돌아가며 j는 다음값 2를 가진다. j가 2를 가진 후 i루프는 다시 처음부터 시작되어 1~300까지 변하며 레이블은 다시 좌측에서 우측으로 이동하게 된다.

좀 복잡하니까 순서도로 살펴보자. 프로그램 전체를 j루프가 둘러싸고 있고 j루프의 명령이 i 루프인 구조를 이루고 있다.

**그림**
두 개의 루프가 겹쳐져 있는 이중 루프




중첩 루프에서는 항상 안쪽 루프가 먼저 돌아간다. 즉 i가 1~300까지 완전히 돌아가고 나서야 j가 다음값을 가지게 된다. 바깥쪽 루프의 입장에서 볼 때 안쪽 루프 전체가 하나의 명령이므로 명령이 끝나기 전에 다음값으로 넘어가지 않는다.

루프의 중첩은 몇 번이라도 상관없다. 2중 루프는 기본이며 필요에 따라 3중, 4중, 심할 경우는 7중 루프도 실전에서 사용된다. 과목1~과목10까지의 합산처리를 1번 학생에서 60번 학생까지, 1반에서 12반까지, 1학년에서 3학년까지 반복하여 학교 전체의 성적을 처리한다면 벌써 4중 루프가 되는 셈이다. 그만큼 컴퓨터 프로그래밍에서 반복이 빈번하게 사용된다는 얘기다.

## ■ repeat until

repeat도 for문과 같은 목적으로 사용되는 반복문이다. for문과는 달리 제어변수를 사용하지 않으며 반복 조건을 설정하여 조건이 참이 될 때까지 명령을 실행한다.

### 기본 형식

```
repeat
  명령들
until 종료조건;
```

6jang
repeat1

repeat와 until 사이에 반복의 대상이 되는 명령을 작성하고 until 다음에 반복을 종료할 종료 조건을 기입해 준다. for문은 하나의 명령에 대해서만 반복을 수행하므로 여러 개의 명령을 반복할 경우 begin과 end를 사용하여 묶어주어야 하지만 repeat until은 기본 형식이 원래부터 여러 개의 명령을 반복하도록 되어 있으므로 begin, end를 사용하지 않아도 된다. 레이블을 움직이는 예제를 repeat문으로 바꾸어 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  i:=1;
  repeat
    Label1.Left:=i;
    Label1.Repaint;
    i:=i+1;
  until i=300;
end;
```

repeat문은 제어 변수를 사용하지 않으므로 직접 i값을 초기화시켜 주고 루프를 한 번 돌 때마다 i값을 1씩 증가시켜 주어야 한다. 종료 조건은 "i가 300이다"가 된다. 즉 i가 300이 될 때까지 i값을 1씩 증가시키며 레이블의 좌표를 계속 증가시킨다. 이 예제에서 i변수는 루프 제어에 꼭 필요한 것도 아니며 중간 처리를 위해서만 사용한다. i변수를 사용하지 않고도 똑같은 프로그램을 만들 수 있다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Left:=1;
  repeat
    Label1.Left:=Label1.Left+1;
    Label1.Repaint;
  until Label1.Left=300;
end;
```

루프 내에서 직접 레이블의 Left 속성을 조작하며 루프의 종료 조건도 Left 속

성값을 사용하여 설정하였다. 사실 속성값 자체가 변수이므로 속성값을 제어 변수로 사용한다 해서 전혀 이상할 것이 없다. 어떤 방법을 쓰든지 결과는 같지만 간단한 경우라면 중간 변수를 꼭 사용할 필요가 없다.

## ■ while 문

for나 repeat와 같은 반복문이며 형식만 조금 다르다.

**기본 형식**

```
while 계속조건 do
 명령;
```

6jang
while1

여러 개의 명령을 사용할 경우 begin과 end로 묶어주는 것은 for문의 경우와 같다. repeat문과 가장 큰 차이점이라면 repeat until은 반복을 끝낼 조건을 기입해 주지만 while은 반복을 계속할 조건을 기입해 준다는 점이다. until이 “~일 때까지”이고 while이 “~인 동안”이므로 영어 뜻을 생각하면 자연스럽게 이해가 갈 것이다. 같은 예제를 while문을 사용하여 다시 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Label1.Left:=1;
 while Label1.Left<300 do
  begin
  Label1.Left:=Label1.Left+1;
  Label1.Repaint;
  end;
end;
```

결과는 물론 동일하며 계속 조건이 "Label1.Left가 300이다"가 아니라 "300보다 작다"이다. Label1.Left가 300보다 작은동안 반복하며 300이 되는 즉시 반복을 끝낸다.

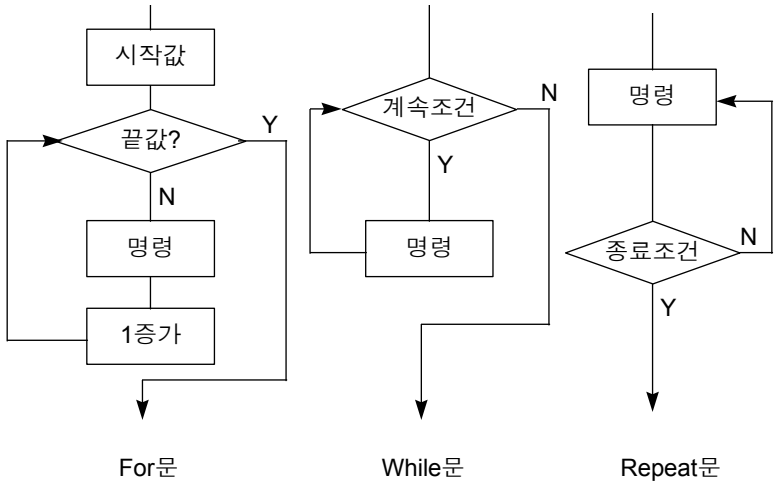
## ■ 루프의 선택

똑같은 반복을 하지만 델파이는 조금씩 형식이 다른 세 개의 반복문을 제공한다. 괜히 기능이 똑같은 반복문을 중복해서 제공하는 것이 아니라 각 제어 구조가 나름대로 특성과 차이를 가지고 있다. 우선 for문은 반복 횟수가 처음부터 정

해진 경우에 가장 적합하며 나머지 둘은 반복 횟수가 가변적인 경우에 적합하다. repeat문은 조건 점검을 루프의 뒤에서 하므로 최소한 한 번은 명령이 실행되지만 while문은 루프가 시작되기도 전에 조건 점검을 하므로 명령을 한 번도 실행하지 않을 수도 있다는 차이점이 있다.

**그림**

세 가지 루프의 진행 과정을 보인 순서도




어떤 목적에 반복문을 사용하는가에 따라 사용해야 할 반복문의 종류가 달라진다. 대개의 경우는 서로 대체성이 있지만 아주 특별한 경우에는 꼭 while문이 필요하다거나 repeat문이 아니면 도저히 해결하지 못하는 경우도 있다. 앞에서 만들어 본 레이블을 이동시키는 예제와 같은 경우라면 셋 중 편한대로 골라서 사용해도 되지만 for문이 가장 속도가 빠르므로 for문을 쓰는 것이 제일 효율적이다.

## ■ Break,Continue

이 명령은 반복문 자체는 아니지만 반복문을 제어하는 데 유용하게 사용되는 명령들이다. Break는 반복문의 나머지 뒷부분을 즉각 중단하며 반복문을 탈출하여 반복문 다음의 명령으로 제어를 옮긴다. 예를 들어 다음과 같은 경우에 i가 5가 되면 명령1과 명령2를 무시하고 명령3을 실행한다.

```
for i:=1 to 10 do
begin
  if i=5 then Break;
  명령 1;
```

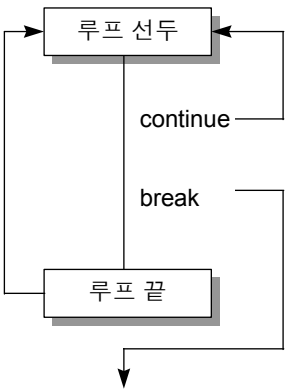
```
 명령 2;
end;
명령 3;
```

더 이상 반복문을 실행할 수 없는 상태이거나 반복할 필요가 없어졌을 때 Break문을 사용하여 반복문을 탈출한다. Continue문은 반복문의 나머지 뒷부분을 무시하고 다음 반복을 계속한다. 예를 들어 다음과 같은 경우에 i가 5가 되면 명령2와 명령3을 무시하고 i를 6으로 만든 후 다음 반복, 즉 명령1을 수행한다.

```
for i:=1 to 10 do
begin
 명령 1;
 if i=5 then Continue;
 명령 2;
 명령 3;
end;
```

반복하는 도중에 특정 조건이 될 때, 즉 반복에서 제외시키고자 할 때 Continue를 사용한다. 위 예에서는 i가 1에서 10까지 변하며 명령1, 2, 3을 실행하되 단 i가 5인 경우는 명령2와 명령3을 실행하지 않는다. 즉 i가 5인 경우에 대해서만 반복 처리를 제외시키는 것이다. 두 명령의 흐름 관계를 그림으로 살펴보자.

그림
루프를 제어하는 명령

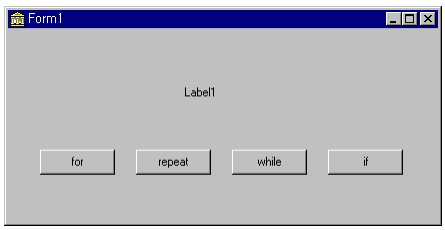



## ■ 1~100 까지 더하는 예제


6jang
sum

필자는 어떤 언어를 제일 먼저 배울 때마다 1~100 까지 더하는 가장 기본적인 예제를 먼저 만들어 보는 아주 좋은 습관을 가지고 있다. 그만큼 이 예제가 그

언어의 특징을 잘 설명해 주고 언어 간의 비교를 할 수 있는 좋은 예제이기 때문이다. 여기서는 델파이로 if 문과 세 개의 순환문을 모두 사용하여 이 예제를 만들어 보았다. 다음과 같이 결과를 출력할 수 있는 레이블 하나와 버튼 네 개를 배치하고 각 버튼에 코드를 작성하였다.



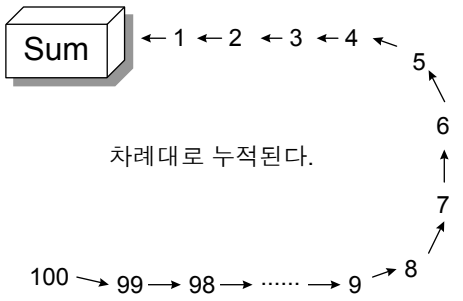

대표적으로 for 문으로 1~100 까지 더하는 예제만 분석해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i,sum:integer;
begin
  sum:=0;           {sum 초기화}
  for i:=1 to 100 do  {1 에서 100 까지}
    sum:=sum+i;      {계속 누적시킴}
  Label1.Caption:=IntToStr(sum);
end;
```

합계를 담을 변수를 sum 으로 선언하고 0 으로 초기화시켰다. 그리고 제어 변수 i 를 1~100 까지 증가시키면서 제어 변수값을 계속 sum 에 누적시켜 나간다. i 가 1 일 때부터 연산 과정을 보면

i=1 ☞ sum:=0+1;
i=2 ☞ sum:=0+1+2;
i=3 ☞ sum:=0+1+2+3;

그림
SUM 변수에 값이 누적되는 과정




이런 식으로 sum 은 자신의 값과 i 를 계속 더해 나가면서 결국 루프가 끝날 때 sum:=0+1+2+3+.....+100; 의 값을 가지게 되며 이 값을 레이블로 출력한다. 나머지 반복문에 의한 합산 과정도 형태만 다를 뿐 논리는 이와 동일하므로 직접 소스를 살펴보기 바란다.

```
{repeat 문}
procedure TForm1.Button2Click(Sender: TObject);
var
 i,sum:integer;
begin
 sum:=0;
 i:=0;
 repeat
 i:=i+1;
 sum:=sum+i;
 until i=100;
 Label1.Caption:=IntToStr(sum);
end;

{while 문}
procedure TForm1.Button3Click(Sender: TObject);
var
 i,sum:integer;
begin
 sum:=0;
 i:=0;
 while i<>100 do
  begin
  i:=i+1;
  sum:=sum+i;
  end;
 Label1.Caption:=IntToStr(sum);
```

```
end;

{if 문}
procedure TForm1.Button4Click(Sender: TObject);
label
  here;   {점프할 목적지인 레이블을 선언한다.}
var
  i,sum:integer;
begin
  sum:=0;
  i:=0;
  here:    {점프처}
  i:=i+1;
  sum:=sum+i;
  if i<>100 then
    goto here;  {here 로 점프}
  Label1.Caption:=IntToStr(sum);
end;
```

이 소스를 주의 깊게 관찰해 보면 세 가지 반복문이 어떻게 다른지 특징을 잘 파악할 수 있을 것이다. 단 if 문의 경우는 label 과 goto 문을 사용하여 점프를 하는 방식을 사용하는데 잘 쓰이지 않는 방식이므로 이런 것도 있다는 것만 보고 넘어가도록 하자.

## ■ 무한 루프

무한 루프란 끊임없이 반복하기만 하는 제어 구조이다. 즉 끝이 없이 명령을 계속 실행하는 구조이며 다음과 같은 형태로 무한 루프를 만든다.

```
while True do
명령;
```

```
repeat
명령;
until False;
```

while 문의 계속 조건이 True 로 되어 있거나 repeat 문의 끝낼 조건이 False 로 되어 있으므로 무한히 반복되는 무한 루프이다. 그렇다고 해서 정말로 무한히 반복하지는 않으며 일정 조건이 되면 루프 내부에서 루프를 탈출한다. 즉 반복문 자체는 무한히 반복하도록 만들어 놓되 루프를 끝낼 시기를 루프 내부에서 결정하도록 한다. 무한 루프를 어떤 경우에 사용하는가 하면 다음과 같은 경우

이다. 두 개의 정수 n 과 m 이 주어졌을 때 이 정수들의 최소 공배수를 찾으려고 한다. 임의의 두 정수가 가지는 최소 공배수는 언제 발견될 지 알 수 없으므로 반복 횟수를 미리 결정할 수 없으며 그래서 무한 루프를 사용해야 한다.

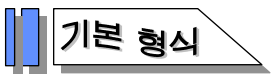
```
procedure TForm1.Button1Click(Sender: TObject);
var
  n,m:integer;
  i:integer;
begin
  n:=9;
  m:=12;
  i:=2;
  while True do
    begin
    if (i mod n=0) and (i mod m=0) then break;
    i:=i+1;
    end;
  Label1.Caption:='최소 공배수는 '+IntToStr(i);
end;
```

임의의 두 정수 n, m 은 사용자로부터 입력되거나 코드의 다른 부분에서 주어지겠지만 일단은 9 와 12 라고 가정한다. 최소 공배수는 항상 2 이상이므로 i 는 2 부터 시작하며 무한 루프에 들어간다. 루프 내부에서는 i 가 두 수의 배수인지를 점검하여 i 가 n 의 배수이고 동시에 m 의 배수일 때 Break 문으로 루프를 탈출하여 그 결과를 레이블로 출력한다. 나머지 연산자 mod 를 사용하며 i 를 n 으로 나눈 나머지가 0 이면 i 를 n 의 배수로 판단한다. 실행중의 모습은 다음과 같다.


참고로 for 문을 사용하여 이 문제를 풀고자 한다면 제어 변수 i 를 2 부터 최소 공배수의 상한값인 n*m 까지 루프를 반복해야 한다. for 문은 반복 횟수가 미리 정해져야 하기 때문에 무한 루프를 만들 수는 없다.

# 다. 선택문

선택문이란 여러 개의 가능한 값이 있고 각각의 값에 따라 처리가 달라야 할 때 사용하는 문장이다. 예를 들어 "돈이 100원 있으면 초코파이를 사고, 200원이 있으면 새우깡을 사먹고, 300원이 있으면 찡오야를 사고 500원이 있으면 포테이토 칩을 사먹고 그보다 더 많으면 은행에 적금을 든다"는 경우가 이에 해당된다. 조건과 명령이 일대일로 대응되고 여러 가지 경우가 있으므로 if문으로 해결하기는 어렵고 선택문을 사용해야 한다. 선택문은 조건과 명령을 깔끔하게 대응시켜 주며 소스를 읽기쉽게 만들어 준다.



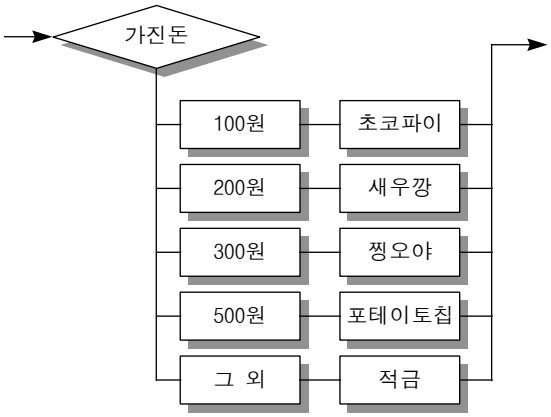

```
case 변수 of
  값1:명령1;
  값2:명령2;
  값3:명령3;
  값n:명령n;
  else 명령4;
end;
```

case 다음의 제어 변수값을 조사해서 제어 변수가 가지는 값에 대응되는 명령을 실행하며 해당되는 값이 없을 경우는 else 뒤의 명령을 수행한다. 위에서 든 예를 case 문으로 바꾸면 다음과 같다.

```
case 가진돈 of
  100:초코파이를 사먹는다;
  200: 새우깡을 사먹는다;
  300:찡오야를 사먹는다;
  500:포테이토 칩을 사먹는다;
  else 은행에 적금을 든다;
end;
```

순서도를 그려 보면 다음과 같다.

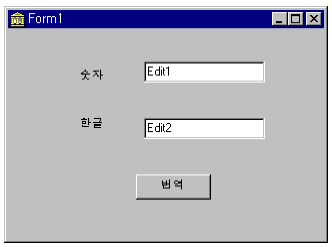
**그림**
경우에 따라 다양한 처리를 지정하는 선택문




제어 변수는 반드시 정수형이어야 하며 값은 다음과 같이 콤마로 끊어 여러 개를 쓸 수도 있다.

```
case Score of
  10,20,30:명령 1;
  40:명령 2;
  50,60:명령 3;
end;
```

그럼 실습을 해 보도록 하자. 다음과 같이 간단하게 폼을 디자인하고 레이블과 버튼의 캡션만 변경한다.





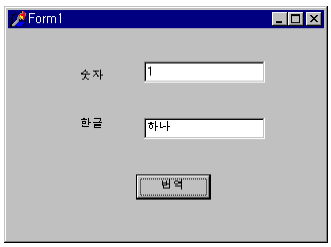
6jang
case1

이 프로그램은 숫자로 1을 넣으면 한글로 '하나'를 출력하고 2를 넣으면 '둘', 3은 '셋', 4는 '넷'을 출력하며 그 외의 값이면 '그 외'라고 출력하는 프로그램이다. 버튼을 더블클릭하여 코드 에디터를 열고 다음과 같이 코드를 작성하자.

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

```
case StrToInt(Edit1.Text) of
  1:Edit2.Text:='하나';
  2:Edit2.Text:='둘';
  3:Edit2.Text:='셋';
  4:Edit2.Text:='넷';
  else
    Edit2.Text:='그 외';
  end;
end;
```

case문의 제어 변수에는 반드시 정수만 사용할 수 있으므로 StrToInt 함수로 Edit1.Text에 입력된 문자열을 정수로 바꾸어 주어야 한다. 실행중의 모습은 다음과 같다.




윗쪽 에디트에 숫자를 입력한 후 번역 버튼을 누르면 아래쪽 에디트에 숫자를 말로 바꾸어 보여준다.


① 반복문:비슷한 부분을 정해진 횟수만큼 실행한다.
② 반복문에는 for, repeat, while 세 가지가 있다.
③ 선택문:각각 값에 따라 다른 처리를 한다.

# 6-4 서브루틴

## 가. 반복 처리

서브루틴(Subroutine)이란 프로그램에서 특정한 일을 담당하는 한 부분을 말하며 이런 서브루틴들을 여러 개 조립하여 프로그램을 만든다. 서브루틴은 프로그램의 부품이 되며 복잡하고 방대한 프로그램을 조직적으로 짤 수 있도록 하는 도구로 사용된다. 프로그램을 짜다 보면 한번 작성한 코드를 다른 곳에서 똑같이 사용해야 할 경우가 많다. 그럴 때마다 코드를 다시 작성하는 것보다는 한 번만 작성해 두고 계속 사용하는 편이 훨씬 더 효율적이다.

의사 코드(Pseudo Code)란 말로 만든 가짜 코드를 말하며 주로 알거리듬 설명을 위해 사용된다.

사용자에게 대화상자를 통해 질문을 하고 사용자의 입력을 받아들이는 코드를 생각해 보자. 하나의 질문이 아니라 여러 개의 질문을 해야 한다면 이 코드는 계속 반복된다. 이름, 생일, 성별 세 가지를 입력받을 경우의 의사 코드를 작성해 보면 다음과 같다.

```
대화상자 그림
질문"이름을 입력하시요"를 출력한다.
키보드로 입력받음
대화상자 닫음
입력받은 값을 이름에 대입

대화상자 그림
질문"생일을 입력하시요"를 출력한다.
키보드로 입력받음
대화상자 닫음
입력받은 값을 생일에 대입

대화상자 그림
질문"성별을 입력하시요"를 출력한다.
키보드로 입력받음
대화상자 닫음
입력받은 값을 성별에 대입
```

벌써 똑같은 코드가 세 번이나 반복되었다. 이렇게 반복될 경우 반복되는 부분을 서브루틴으로 만들고 서브루틴을 불러서 사용한다.

```
서브루틴 질문루틴(질문)              ← 서브루틴을 정의한다.
begin
 대화상자 그림
 질문을 출력한다.
 키보드로 입력받음
 대화상자 닫음
 입력받은 값을 변수에 대입
ebd;

이름:=질문루틴('이름을 입력하시요');    ←서브루틴 호출
생일:=질문루틴('생일을 입력하시요');
성별:=질문루틴('성별을 입력하시요');
```

이런 식으로 반복되는 코드를 별도의 서브루틴 이름으로 등록한 후 필요할 때는 서브루틴을 불러서 사용한다. 서브루틴은 조건문, 반복문과 마찬가지로 프로그래밍의 가장 기초적인 이론에 속한다. 서브루틴을 사용할 경우 다음과 같은 여러 가지 장점이 있다.

❶ 코드의 반복을 최소화한다. 똑같은 부분을 여러 번 입력하는 것보다는 시간이 절약되며 프로그램의 크기가 작아진다.

❷ 수정이 용이하다. 반복되는 부분을 일일이 입력했을 경우 코드를 변경하려면 반복된 모든 부분을 변경해 주어야 하지만 서브루틴으로 만들어 놓으면 서브루틴만 변경하면 된다.

❸ 분할 작업이 가능하다. 각각의 서브루틴을 만들고 난 후 이런 서브루틴을 모아 프로그램을 만들기 때문에 여러 사람이 동시에 작업을 할 수 있다. 각자 책임 맡은 서브루틴을 작성하고 난 후 이를 합치면 하나의 프로그램이 된다.

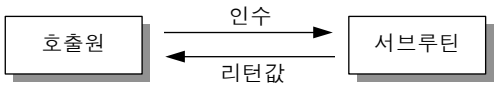
델파이는 프로시저(procedure)와 함수(function) 두 가지 종류의 서브루틴을 제공한다. 기본적인 개념은 위에서 설명한 서브루틴의 그것과 동일하지만 프로시저는 특정한 작업을 분담하고 함수는 특정한 계산을 분담하는 용도로 사용되며 외형적으로는 돌려주는 값이 있는가, 없는가의 차이점이 있다. 프로시저는

돌려주는 리턴값이 없으며 함수는 리턴값이 있다.

서브루틴에서 먼저 이해해야 할 것은 인수와 리턴값의 개념이다. 인수(parameter)란 서브루틴을 호출하는 호출원에서 서브루틴으로 넘겨주는 값을 말하며 서브루틴에게 작업거리를 제공해 준다. 질문을 하는 서브루틴을 호출할 경우 어떤 질문을 할 것인가를 가르쳐 주어야 하며 화면에 그림을 그리는 서브루틴을 호출할 경우 어떤 그림을 어느 위치에 그릴지를 가르쳐 주어야 한다. 이때 호출원에게 넘겨주는 질문, 그림, 위치가 인수이며 서브루틴 호출문에서 서브루틴 이름 뒤의 괄호 안에 표기한다. 리턴값이란 인수와는 반대로 서브루틴에서 작업의 결과를 호출원으로 돌려주는 값을 말한다. 위에서 예로 든 서브루틴에서 사용자에게 입력받은 값이 리턴값이다.

**그림**

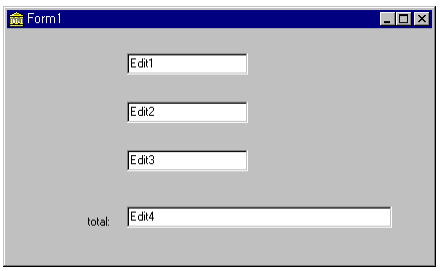
호출원과 서브루틴은 인수와 리턴값으로 값을 교환한다.




인수와 리턴값은 있을 수도 있고 없을 수도 있다. 인수는 꼭 필요할 경우만 사용하며 개수의 제한은 없다. 리턴값은 함수의 경우에 한해 꼭 하나만 있을 수 있으며 프로시저는 리턴값을 가지지 않는다.

# 나. 프로시저

"프로시저는 특정 작업을 담당한다"라고 말하면 무슨 말인지 얼른 이해가 안 되므로 구체적인 예제를 제작해 보면서 프로시저의 필요성과 구현 문법에 대해 알아보자. 다음과 같이 레이블 하나와 에디트 네 개를 배치하자.




6jang
total

레이블의 캡션을 'total:'로 바꾸고 에디트의 Text는 모두 지운 후 제일 아래의 Edit4는 길이를 조금 길게 만든다. 이 예제는 위쪽 세 개의 에디트에 입력된 내

용을 모두 연결하여 아래쪽의 긴 에디트에 출력한다. 문자열끼리 연결하는 것은 단순히 +연산자로 더해 주기만 하면 되며 세 개의 에디트 중 하나라도 변경이 되면 다시 연결을 해야 하므로 세 에디트의 OnChange 이벤트에 코드를 작성해 준다. 문자열만 연결하면 너무 싱거우니까 에디트의 내용이 변경될 때마다 소리를 내도록 하였다. 소리를 내는 데는 MessageBeep를  사용한다. 각 에디트의 OnChange 이벤트를 다음과 같이 작성하였다.

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  MessageBeep(MB_OK);
  Edit4.Text:=Edit1.Text+Edit2.Text+Edit3.Text;
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
  MessageBeep(MB_OK);
  Edit4.Text:=Edit1.Text+Edit2.Text+Edit3.Text;
end;

procedure TForm1.Edit3Change(Sender: TObject);
begin
  MessageBeep(MB_OK);
  Edit4.Text:=Edit1.Text+Edit2.Text+Edit3.Text;
end;
```

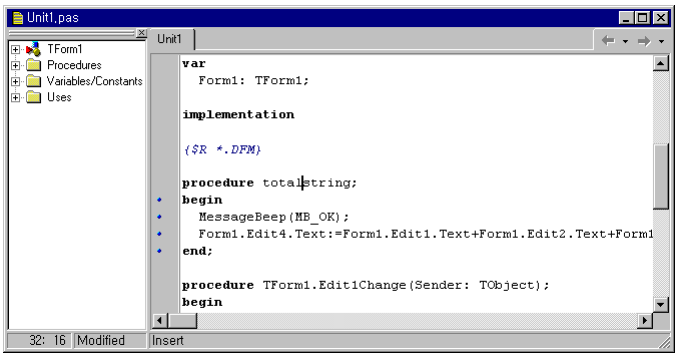
이벤트 핸들러를 살펴보면 모두 같은 내용의 코드를 반복하여 처리하고 있음을 알 수 있다. 이 반복되는 내용을 totalstring이라는 프로시저로 등록해 두고 이벤트 핸들러에서는 이 프로시저를 불러서 사용하도록 변경해 보자.

```
procedure totalstring;
begin
  MessageBeep(MB_OK);
  Form1.Edit4.Text:=Form1.Edit1.Text+Form1.Edit2.Text+Form1.Edit3.Text;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  totalstring;
end;
```

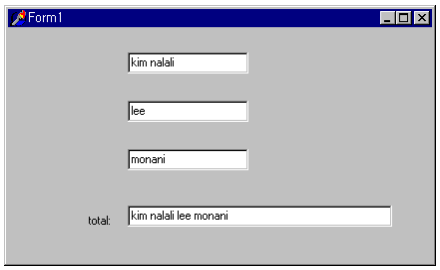
```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  totalstring;
end;

procedure TForm1.Edit3Change(Sender: TObject);
begin
  totalstring;
end;
```

사용자가 프로시저를 입력할 때는 코드 에디터에서 직접 입력해 주어야 한다. 이벤트 핸들러를 만들 때처럼 델파이가 프로시저의 틀을 만들어 주지 않으므로 프로시저의 틀도 직접 만들어야 한다.

입력할 위치는 implementation 아래 {$R *.DFM} 바로 밑부분, 즉 이 프로시저를 사용하는 이벤트 핸들러보다 앞에 있어야 한다. 그렇지 않으면 이벤트 핸들러에서 totalstring을 호출할 수 없다.




totalstring 내에서 에디트 컴포넌트를 칭할 때는 Edit1이라고 부를 수 없으며 어떤 폼에 있는지를 밝혀 주어야 하기 때문에 Form1.Edit1이라고 해야 한다. 왜 그런가 하면 totalstring이 폼에 소속된 이벤트 핸들러가 아니며 사용자가 직접 만든 프로시저이기 때문이다. totalstring을 폼의 타입 선언에 포함시키는 것도 가능하지만 현재 단계에서는 이렇게 만드는 편이 더 이해하기 쉬울 것이다. 실행 결과는 다음과 같다. 세 에디트의 내용중 하나라도 변경되면 totalstring 프로시저가 호출된다.




위의 예제에서는 프로시저의 내용이 짧고 사용되는 곳도 세 군데밖에 없기 때문에 프로시저를 사용하는 것과 그냥 이벤트 핸들러 안에 코드를 쓰는 것과 큰 차이는 없지만 프로시저의 길이가 아주 길고 사용빈도가 잦다면 프로시저를 사용하는 것이 훨씬 더 유리하다. 프로시저를 선언하는 기본 형식은 다음과 같다.

### 기본 형식

```
procedure 프로시저이름(인수:인수형;인수:인수형...);
begin
  코드
end;
```

예약어 procedure로 시작되며 다른 프로시저나 함수, 변수 등과 구분해야 하므로 반드시 고유한 이름을 가져야 한다. 프로시저의 이름은 사용자가 기억하기 쉽게 마음대로 정할 수 있으며 명칭을 만드는 규칙에만 맞게 만들면 된다. 가급적이면 프로시저의 동작을 요약적으로 설명해 주는 이름을 사용하는 것이 좋다. 인수가 있을 경우에는 프로시저 이름 뒤의 괄호 안에 인수의 이름과 인수의 데이터형을 밝혀 주며 여러 개의 인수를 가질 수 있다. 다음은 정수형 인수 age와 문자열 인수 myname을 사용하는, 이름이 printname인 프로시저이다.

```
procedure printname(age:integer;myname:string);
begin
  코드
end;
```

begin과 end 사이에 프로시저의 본체인 코드가 위치한다. 호출원에서 프로시저를 호출할 때는 프로시저의 이름과 인수를 사용한다.

```
printname(23,'Kim Sang Hyung');
```

호출원에서 전달한 인수 23이 age로 전달되고 'Kim Sang Hyung'이 myname으로 전달되며 이 두 인수가 printname 프로시저의 작업거리이다.

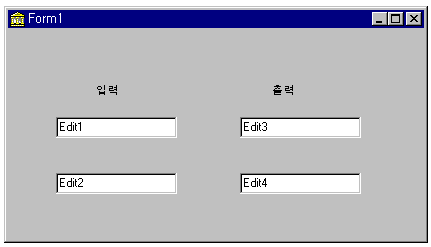
# 다. 함수

함수란 정해진 계산을 하여 계산 결과를 호출원으로 돌려주는 서브루틴이다. 프로시저와 비슷한 형식을 가지지만 리턴값이 있다는 점이 다르다.

### 기본 형식

```
function 함수이름(인수:인수형; 인수:인수형,...):리턴값형;
begin
  코드
end;
```

procedure 대신 예약어 function을 사용하며 프로시저가 이름이 필요한 것과 마찬가지로 함수도 고유의 이름을 가져야 하고 인수를 선언하는 것도 프로시저와 동일하다. 인수 선언 다음에는 호출원으로 돌려줄 값이 어떤 데이터형을 가지는가를 밝혀주며 Integer, Single, String 등의 데이터형 이름을 기입해 준다. 직접 예제를 작성해 보자. 여기서 작성해 볼 예제는 에디트에 입력된 문자열을 뒤집어서 다시 출력하는 예제이다. 즉 'abcd'를 입력하면 'dcba'가 출력된다. 다음과 같이 4개의 에디트를 폼에 배치한다.





6jang
reverse

폼에 컴포넌트를 배치한 후 이번에는 함수를 먼저 입력해 보자. 코드 에디터로 이동한 후 implementation 아래에 다음 코드를 입력해 넣는다.

```
function reversestring(str:string):string;
```

```
var
  i,len:integer;
  s:string;
begin
  len:=length(str);
  s:=str;
  for i:=1 to len do
    s[len-i+1]:=str[i];
  reversestring:=s;
end;
```

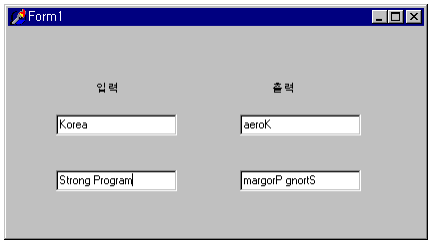
이 코드에서는 문자열 str을 인수로 입력받아서 거꾸로 뒤집은 후 다시 돌려주는 동작을 하는 reversestring이라는 이름의 함수를 정의하고 있다. 함수에서 리턴값을 호출원으로 돌려줄 때는 함수의 이름에 리턴할 값을 대입해 준다. 또는 Result라는 예약어에 돌려줄 값을 대입해도 된다. 즉 다음 두 문장은 결과값을 호출원으로 돌려주는 동일한 문장이다.

```
reversestring:=s;
Result:=s;
```

주의할 것은 리턴값을 돌려준다고 해서 곧바로 함수가 끝나는 것은 아니며 함수는 end; 바로 앞까지 실행된 후 Result나 함수명에 대입된 값을 반환하면서 종료된다. reversestring 함수는 전달되어 온 문자열을 str 인수로 받아 이 문자열의 뒷부분에서부터 한 문자씩 읽어 임시 문자열 변수 s의 앞에서부터 차례로 대입함으로써 문자열을 뒤집는다. 문자열을 뒤집은 후 그 결과값인 s를 다시 호출원으로 돌려준다.  함수 reversestring을 만들었으면 이제 이 함수를 직접 사용해 보자. Edit1, Edit2의 내용이 변경될 때마다 문자열을 뒤집어 Edit3, Edit4로 출력하므로 Edit1, Edit2의 OnChange 이벤트에서 이 함수를 불러야 한다. 이 함수의 리턴값을 출력 에디트의 Text 속성에 대입하면 된다.

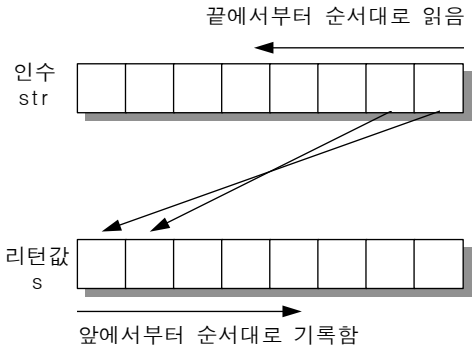
```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Edit3.Text:=reversestring(Edit1.Text);
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
  Edit4.Text:=reversestring(Edit2.Text);
end;
```

이렇게 하면 예제가 완성되었다. 문자열을 뒤집어 주는 함수를 이미 만들어 두었으므로 필요할 때는 뒤집을 재료가 되는 문자열을 인수로 사용하여 함수를 부르기만 하면 된다. 그리고 함수가 돌려주는 리턴값을 대입받으면 원하던 문자열을 얻을 수 있다. 실행중의 모습은 다음과 같다.




함수의 본체에서 문자열을 뒤집는 방법은 인수로 전달된 문자열의 뒷부분부터 읽어서 임시 문자열의 처음부터 써 주기를 문자열 길이만큼 반복하는 것이다. 예상외로 간단하다.

그림
문자열 뒤집기




참고하세요

함수와 프로시저는 코드의 분절을 이룬다는 면에 있어서 동등하며 실제로 C 언어에서는 둘 다 함수로 통합되어 있다. 이 둘을 묶어서 표현할 때는 서브루틴이라고 표현하는 것이 용어 사용 면에서 옳은 표현이지만 보통 함수라고 표현한다. 즉 함수는 좁은 의미로는 리턴값을 돌려줄 수 있는 function을 의미하지만 넓은 의미로는 프로시저를 포함한다고 알아두고 책을 읽는데 오해가 없기 바란다.

6jang
kongfun

반복문에서 우리가 만들었던 최소 공배수를 찾아주는 예제를 이번에는 함수로 만들어 보았다. 코드는 다음과 같다.

```
function MinKong(n,m:integer):integer;
var
 i:integer;
begin
 i:=2;
 while True do
   begin
   if (i mod n=0) and (i mod m=0) then break;
   i:=i+1;
   end;
 MinKong:=i;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 Label1.Caption:='최소 공배수는 '+
   IntToStr(MinKong(9,12));
end;
```

MinKong이라는 이름의 함수를 정의하고 최소 공배수를 찾을 두 정수를 n, m이라는 인수로 전달받는다. 최소 공배수를 찾은 후에 결과를 다시 돌려주며 호출원에서는 함수 호출문을 곧바로 사용한다. 실행 결과는 동일하지만 이렇게 함수로 한번 만들어 놓으면 프로그램 내의 어떤 코드에서도 이 함수를 편리하게 불러서 사용할 수 있게 된다.


① 서브루틴:반복되는 부분을 일정한 형식에 맞게 작성한 코드
② 함수는 리턴값을 가지며 프로시저는 가지지 않는다.

# 라. 참조 호출

## ■ 형식 인수, 실 인수

프로시저의 한 예를 들어 보자. 다음 프로시저는 사람의 이름과 나이를 출력해 주는 프로시저이다.

```
procedure printname(age:integer;myname:string);
```

이 프로시저 호출문에서는 printname(23,'Kim'); 과 같이 두 개의 인수를 넘겨 줄 것이고 이 인수를 프로시저 내부에서는 age 와 myname 으로 대입받아 사용한다. 이 때 실제 호출문에 나타나는 인수를 실(actual) 인수라고 하며 23, 'Kim' 등과 같은 상수나 myage, yourname 과 같은 변수가 올 수 있다. 반면 프로시저 내부에서 실 인수를 대입받는 age 나 myname 을 형식(formal) 인수라고 한다. 형식 인수는 프로시저 호출문에서 전달된 실 인수값을 잠시 보관해 두며 프로시저의 본체에서 사용된다. 형식 인수는 프로시저 호출 과정에서 실 인수의 값을 대입받을 뿐 실 인수와는 물리적으로 상관이 없는 복사본이다. 즉 프로시저 호출 직후에 실 인수와 같은 값을 가질 뿐이지 별도의 메모리를 차지하는 별도의 변수라는 뜻이다. 다음 프로시저를 보자.

```
procedure Make2(num:integer);
begin
  num:=num*2;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  i:=3;
  Make2(i);
  Label1.Caption:=IntToStr(i);
end;
```

Make2 프로시저의 num 이 형식 인수이며 호출문에 나타난 실 인수는 i 이다. 형식 인수 num 은 프로시저가 호출되면 별도의 메모리에 만들어지며 실 인수 i 의 값을 잠시 대입받는다. 문제는 num 이 i 의 복사본일 뿐이므로 프로시저 내에서 num 의 값을 아무리 지지고 볶고 변경해 보아야 실 인수 i 는 아무런 영향을 받지 않는다는 것이다. num 은 프로시저 내에서 만들어졌다가 프로시저가 끝나면 파괴되는 지역 변수에 불과하다.

Make2 프로시저는 인수로 전달된 값을 두 배로 만들어 주는 동작을 하도록 제작된 것이지만 프로시저 내에서 실 인수값을 변경할 수 있는 방법이 없으므로 원하는 목적을 달성할 수 없다. 레이블로 출력되는 값은 원래의 i 값인 3 이다. 프로시저가 호출될 때 프로시저로 전달되는 것은 실 인수의 값이지 실 인수 자체가 아니다. 이렇게 실 인수의 값이 전달되는 호출 방법을 값 호출(call by value)이

라고 한다.

## ■ 참조 호출

6jang
vararg

참조 호출은 값 호출과는 달리 실 인수 자체를 프로시저로 넘겨주는 방법이다. 즉 복사본을 만들지 않고 프로시저 내에서 곧바로 실 인수 자체를 사용하는 방법이다. 참조 호출을 하려면 프로시저 선언문의 인수 리스트에서 var 를 기입해 준다.

```
procedure Make2(var num:integer);
begin
 num:=num*2;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  i:=3;
  Make2(i);
  Label1.Caption:=IntToStr(i);
end;
```

이렇게 되면 형식 인수 num 은 별도의 메모리에 새로 생성되는 복사본이 아니라 실 인수 i 그 자체를 나타내며 num 에 가해지는 어떤 조작은 곧바로 i 의 값을 변화시킨다. 두 변수는 이름만 다를 뿐이지 같은 메모리 영역을 가리키는 같은 변수라고 할 수 있다. 출력되는 결과는 i 값을 두 배로 한 6 이다. 이렇게 실 인수 자체를 인수로 전달하는 방법을 참조 호출(call by reference)이라고 한다. 변수 i 의 값인 3 을 프로시저로 전달하는 것이 아니라 i 그 자체를 전달하기 때문에 붙여진 이름이다.

한 가지 주의할 점은 참조 호출문에 상수는 사용할 수 없다는 점이다. 위의 예에서 Make2(3);과 같이 상수를 넘겨준다면 Make2 의 본체에서는 3:=3*2;라는 말도 안되는 대입이 발생할 것이다. 참조 호출에는 값을 변경시킬 수 있는 변수만을 인수로 넘겨줄 수 있다.

참고하세요

컴포넌트의 속성은 변수와 비슷하여 변수가 쓰이는 곳이면 어느 곳에나 쓰일 수 있다. 하지만 몇몇 부분에서 차이점이 있다고 했었는데 바로 참조 호출 부분에서 차이가 있다. 변수는 참조 호출의 인수로 사용할 수 있지만 속성은 참조 호출의 인수로 사용할 수 없다는

점을 명심할 것. 그 이유는 16장에 가서나 배우게 되겠지만 간단하게 설명하자면 속성은 실제로는 변수가 아니라 변수를 액세스하는 함수이기 때문이다.

## ■ 출력용 참조 인수

인수라는 것은 함수를 호출하는 측에서 함수에게 작업거리를 제공해 주는 값이다. 예를 들어 IntToStr 함수는 문자열을 정수로 바꾸어주는데 이때 변경의 대상이 되는 문자열이 이 함수로 전달되며 앞에서 만든 MinKong 함수는 두 개의 정수를 인수로 전달받아 최소 공배수를 구해주었다. 이렇게 함수의 동작을 지시하기 위해 호출원에서 전달해 주는 인수는 입력용이다.

그런데 참조 호출로 전달되는 인수는 반드시 입력을 위해서만 쓰이는 것이 아니라 함수쪽에서 호출원으로 작업 결과를 보고하기 위한 용도로도 사용된다. 값 호출로 전달된 인수는 호출원으로 돌려줄 방법이 없으므로 함수내에서 쓰이고 버려지지만 참조 호출로 전달된 인수는 호출원으로 돌려줄 수 있기 때문이다. 예를 들어 다음과 같은 프로시저를 상상해 보자.

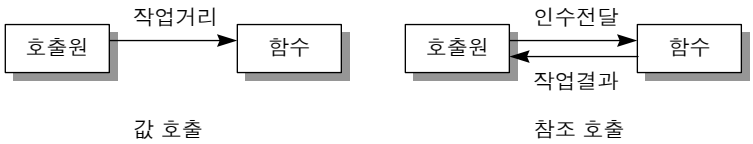
```
procedure CanContinue(var Yes:Boolean);
```

이 프로시저는 계속 진행(게임이나 파일 복사 등 하여튼 어떤 작업이든지)이 가능한가 아닌가를 결정하는데 연산 결과 가능하다면 Yes 에 True 를 대입해 주고 그렇지 않다면 False 를 대입해 주면 된다. 참조 인수로 전달된 Yes 인수를 프로시저 내에서 변경하면 이 값이 호출원으로 그대로 전달되기 때문이다. 호출원에서는 이 프로시저를 다음과 같이 사용할 수 있다.

```
var
  Yes:Boolean
begin
  CanContinue(Yes);
  if Yes=True then
    계속
  else
    안계속;
....
```

참조 인수를 전달해야 하므로 먼저 변수를 선언한 후 이 변수를 CanContinue 프로시저에게 전달해 주어야 한다. 그리고 이 프로시저에서 설정해 주는 Yes 값을 보고 계속 진행을 결정할 수 있다. 이런 식으로 호출원에서 프

로시저에게 작업거리를 제공하기 위해 전달하는 것이 아니라 프로시저의 실행 결과를 돌려받기 위해 전달하는 인수를 출력용 참조 인수라고 한다. 입력용으로 사용되는 인수가 아니므로 CanContinue 프로시저는 이 인수에 어떤 값이 전달되어져 왔는가는 전혀 고려하지 않아도 된다. 그림으로 입력 인수와 출력 인수를 비교해 보면 다음과 같다.




값 호출 참조 호출

출력 인수는 값을 반드시 돌려받기 위해서만 사용하는 것은 아니며 실제로는 입력과 출력을 겸한다. 즉 호출원에서 프로시저로 값을 넘겨주는 용도로도 사용되고 프로시저에서 호출원으로 값을 돌려주기 위한 용도로도 사용된다. 앞에서 만든 Make2 프로시저에서 사용된 참조 인수 num 이 입출력 겸용 인수의 좋은 예이다.

그런데 왜 이런 출력용 인수를 사용할까? 함수의 리턴값을 사용하면 꼭 이런 어려운 방법을 쓰지 않아도 될 것 같다. CanContinue 프로시저의 경우 다음과 같이 함수로 만든다면 똑같이 동작할 수도 있다.

```
function CanContinue();
.....
var
  Yes:Boolean;
begin
  Yes:=CanContinue;
  if Yes=True then 어쩌구 저쩌구.....
```

더 간단하게 한다면 if CanContinue=Ture then ... 처럼 바로 함수의 리턴값을 점검할 수도 있다. 물론 이렇게 해도 된다. 그런데 만약 돌려주어야 할 값이 하나가 아니라 여러 개라면 어떻게 할 것인가? 이런 경우는 함수의 리턴값으로 해결할 수 없다. 그보다 더 중요한 이유는 이벤트 핸들러에서 값을 돌려주고자 할 때이다. 함수는 이벤트 핸들러가 될 수 없고 반드시 프로시저만 이벤트 핸들러가 될 수 있다. 그 이유를 굳이 따지자면 메시지 구동 시스템의 본질적인 문제에 대해 생각해 봐야 하는데 이벤트 핸들러라는 것이 사건에 대한 반응일 뿐 질문에 대한 답이 아니기 때문이다.

조금만 생각해 보면 시스템이나 사용자에 의해 발생하는 메시지는 리턴값을

전혀 요구하지 않는다는 것을 알 수 있다. OnClick 이벤트나 OnChange 이벤트는 "너 지금 눌러졌어", "너 지금 변경됐어"라고 알려주는 것 뿐이지 "너 ~를 어떻게 할래?"라고 묻는 것이 아니다. 컴포넌트는 이런 이벤트에 대해 "나는 이렇게 동작할래"라고 정의하기만 하면 그만이다.

그런데 이벤트중에 의도적으로 보내지는 이벤트는 질문일 수도 있다. 예를 들어 DragOver 이벤트의 경우 "지금 아무개 컴포넌트가 드래그 되고 있는데 드롭해도 돼?"하는 질문을 한다. 그렇다고 프로시저가 리턴값을 줄 수는 없으므로 호출원(이 경우 운영체제)에서 출력용 참조 인수를 보내준다. DragOver 이벤트 핸들러의 원형을 보자.

```
procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer; State: TDragState; var
Accept: Boolean)
```

이 핸들러의 제일 끝에 보면 Accept 라는 인수가 있는데 이 인수가 바로 드롭을 받을 것인지를 묻는 참조 인수인데 드롭을 허가하면 여기에 True 를 대입해 주면 된다. 델파이에는 이런 식의 참조 호출 인수를 사용하는 이벤트가 몇 가지 더 있는데 여기서 개념만 잘 이해하면 당황하지 않을 것이다. 드래그 & 드롭에 대한 자세한 내용은 12 장에서 배울 것이다.

## ■ 함수와 다른 점

위와 같이 어떤 값을 두 배로 만드는 서브루틴이 필요하다면 참조 호출 프로시저를 쓰는 것보다는 리턴값을 돌려 줄 수 있는 함수를 사용하는 것이 더 좋다. 함수로 만든 다음 코드를 위의 코드와 주의 깊게 비교해 보아라.

```
function Make2(num:integer):integer;
begin
  num:=num*2;
  Make2:=num;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  i:=3;
  i:=Make2(i);
  Label1.Caption:=IntToStr(i);
end;
```

결국 결과는 같다. 그러나 함수는 반드시 하나의 리턴값만을 돌려줄 수 있지만 프로시저는 참조 호출을 통해 여러 개의 값을 직접 변경시킬 수 있으므로 실질적으로 다수 개의 리턴값을 반환하기도 한다. 또한 함수 자체는 리턴값으로 평가되기 때문에 수식에 곧바로 사용할 수 있다는 점이 다르다. 위 프로시저는 다음과 같이 쓸 수도 있다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  i:=3;
  Label1.Caption:=IntToStr(Make2(i));
end;
```

델파이에서는 참조 호출이 자주 사용되지는 않으며 리턴값이 필요한 경우는 함수를 사용하므로 어렵다고 생각이 되는 사람은 몰라도 큰 지장은 없을지도 모르겠다. 게다가 여러분이 직접 참조 호출 프로시저를 만들어 쓸 일도 드물다. 하지만 고급 기법에 사용되는 몇 개의 중요한 메소드가 참조 호출을 사용하므로 개념은 꼭 알아 두는 것이 좋다고 생각된다.

# 마. 메시지 상자

프로그램은 혼자서만 작동되는 것이 아니라 사용자와 함께 상호 작용(interaction)해야 한다. 사용자로부터 명령을 받아들여야 하고 작업의 결과나 진행 상황을 사용자에게 보여주어야 한다. 명령을 받아들일 때는 버튼이나 키보드 입력을 사용하며 사용자에게 메시지를 보여줄 때는 주로 메시지 상자를 사용한다. 메시지 상자는 프로그램 윈도우와는 다른 별도의 윈도우이므로 직접 만들려면 폼을 하나 더 만들어야 하는 것이 원칙이지만 델파이가 메시지 상자를 출력해 주는 함수와 프로시저를 제공해 주므로 이 함수들을 사용하면 간단하게 메시지 상자를 출력할 수 있다.

## ■ 메시지 출력

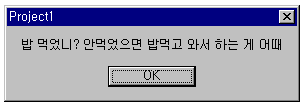
6jang
mes1

사용자에게 전달할 메시지를 메시지 프로시저의 인수로 전달해 주면 윈도우를 만들고 메시지를 보여주고 사용자가 버튼을 누를 때까지 대기했다가 버튼을 누르면 메시지 상자를 닫아주는 일까지 해준다. 가장 간단한 메시지 프로시저는

ShowMessage이다. 인수로는 전달할 메시지 문자열만 기입해 준다. 실습을 위해 빈 폼에 버튼을 만들고 버튼의 OnClick 이벤트에 다음 코드를 작성해 보아라.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('밥 먹었니? 안먹었으면 밥먹고 와서 하는 게 어때');
end;
```

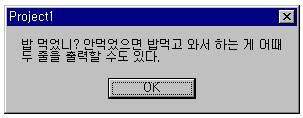
프로그램을 실행시키고 버튼을 누르면 화면 중앙에 메시지 상자를 열어 준다.




인수로 전달한 메시지가 상자 중앙에 출력되며 타이틀 바에는 메시지 상자를 호출한 프로그램의 이름이 출력되어 있다. 그리고 상자 하단에는 OK 버튼을 배치하여 사용자가 OK 버튼을 누를 때까지 대기했다가 버튼을 누르면 메시지 상자를 닫는다.

메시지 상자에 출력할 수 있는 문자열은 원칙적으로 하나밖에 없지만 #13제어 문자를 사용하면 다음과 같이 두 줄을 출력할 수도 있다. #13은 엔터 코드이다.

```
ShowMessage('밥 먹었니? 안먹었으면 밥먹고 와서 하는 게 어때'
      +#13+'두 줄을 출력할 수도 있다.');
```




ShowMessage는 아주 간단한 메시지를 전달할 때만 사용한다. 좀 더 복잡한 형태의 메시지나 사용자에게 어떤 응답을 받아야 할 때는 MessageDlg라는 함수를 사용한다. 이 함수를 사용하는 기본 형식은 다음과 같다.

### 기본 형식

MessageDlg(메시지, 형태, 버튼 종류, 도움말 정보):Word;

**■ 메시지**

ShowMessage의 경우와 같이 메시지 상자에 출력할 문자열이다. 사용자에게 전달할 내용이나 질문할 내용을 여기에 기입해 준다.

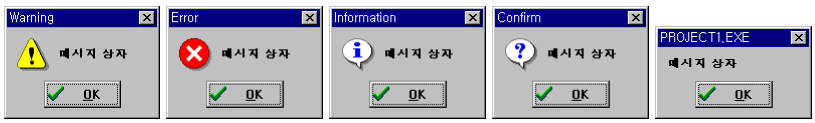
### ■ 형태

형태는 메시지 상자에 어떤 비트맵을 배치할 것인가와 타이틀 바에 나타날 문자열을 지정하며 다음 중 하나의 값을 지정한다.

| 값 | 형태 |
|---|---|
| mtWarning | 노란색 느낌표 비트맵이 나타난다. |
| mtError | 빨간색의 엑스 비트맵이 나타난다. |
| mtInformation | 파란색의 i자 비트맵이 나타난다. |
| mtConfirmation | 물음표 비트맵이 나타난다. |
| mtCustom | 비트맵을 사용하지 않는다. |

아래에 각 메시지 상자가 어떤 모양을 가지는가를 보였다.

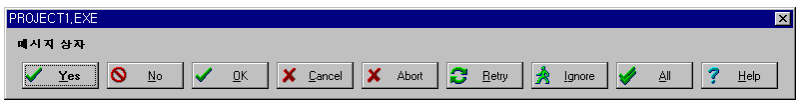
그림
메시지 상자의 종류




### ■ 버튼 종류

메시지 상자에 나타날 버튼의 종류를 지정한다. 집합형이므로 여러 개의 버튼을 [ ] 안에 기입하여 출력할 수 있다. 버튼의 종류에는 mbYes, mbNo, mbOk, mbCancel, mbHelp, mbAbort, mbRetry, mbIgnore, mbAll 등이 있으며 각 버튼의 모양은 다음과 같다. 일일이 버튼의 모양을 보여주기 번거로와서 한 대화상자에 다 모아 보았다.

그림
메시지 상자에 나타나는 버튼들



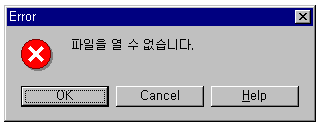

사용 예를 보자.

6jang
mes2

MessageDlg('파일을 열 수 없습니다.',

```
mtError,[mbOk,mbCancel,mbHelp],0);
```

이렇게 호출했다면 메시지 상자 중앙에 에러 메시지를 출력하고 세 개의 버튼을 메시지 상자 하단에 배치해 준다.




일일이 버튼 종류를 기입해 주는 것이 귀찮으면 미리 정의되어 있는 세 가지 형태의 버튼 집합을 사용해도 된다.

| 집합명 | 버튼 |
|---|---|
| mbYesNoCancel | Yes, No, Cancel 버튼 |
| mbOkCancel | Ok, Cancel 버튼 |
| mbAbortRetryIgnore | Abort, Retry, Ignore 버튼 |

이 집합명은 그 자체가 집합형이므로 [ ]괄호를 쓰지 않아도 된다.

**■ 도움말 정보**

메시지 박스에 도움말이 제공될 경우 도움말의 인덱스를 지정한다. 도움말을 사용하지 않더라도 이 인수는 반드시 있어야 하며 어떤 값을 쓰더라도 상관은 없지만 보통 0의 값을 넘겨준다. 물론 도움말을 제공한다면 적절한 도움말 인덱스를 전달해 주어야 할 것이다.

**■ 리턴값**

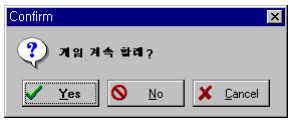
ShowMessage 프로시저와 MessageDlg 함수는 인수의 개수가 다르다는 점 외에도 아주 큰 차이점이 있다. ShowMessage는 단순히 메시지를 보여주기만 하는 프로시저이며 MessageDlg는 메시지를 보여준 후 사용자로부터 응답 결과를 받아 올 수 있는 함수이다. MessageDlg 함수가 리턴하는 값은 사용자가 메시지 박스에서 선택한 버튼이며 mrNone, mrOk, mrCancel, mrAbort, mrRetry, mrIgnore, mrYes, mrNo, mrAbort, mrRetry, mrIgnore, mrAll 중 하나의 값이 된다. 리턴값의 이름 자체가 워낙 설명적이므로 별도의 도표를 그리지 않아도 될 것이다.

이 리턴값을 사용하면 메시지 상자를 보여준 후 사용자의 응답에 따라 별도의

처리를 수행할 수 있다.

```
if MessageDlg('게임 계속 할래?', mtConfirmation,
   mbYesNoCancel,0)=mrYes
 then  게임 계속 처리
else 게임 끝 처리;
```

MessageDlg 함수로 게임을 계속 할 것인가를 물어 보며 이 질문에 대답할 수 있도록 세 개의 버튼을 제공한다.

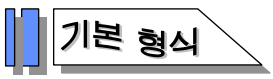



메시지 박스가 닫혀지면 사용자가 선택한 버튼의 종류를 판단하여 게임을 계속 실행시키든가 아니면 게임을 중지한다.

MessageDlg 프로시저 외에 이와 유사한 MessageBox 함수가 있는데 타이틀 바에 나타날 문자열도 직접 제어할 수 있다는 차이가 있을 뿐 MessageDlg와 사용하는 방법은 거의 동일하다. 이 함수에 대한 자세한 정보는 도움말을 참고하기 바란다.

## ■ 입력용 대화상자

사용자에게 내용을 전달하는 메시지 상자와는 반대로 사용자에게 입력을 받아야 하는 입력용 대화상자도 있다. 프로그램 실행중에 문자열을 입력받아야 하거나 질문을 해야 할 때 이런 대화상자를 사용하며 델파이가 제공하는 프로시저에 의해 구현된다.




InputBox(캡션, 질문, 디폴트):string;

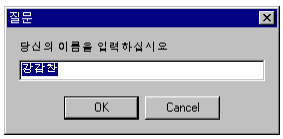

6jang
mes3

캡션은 대화상자의 타이틀 바에 나타날 문자열이며 질문은 대화상자에 나타날 질문 내용이다. 디폴트는 질문 대화상자가 열릴 때 미리 주어지는 문자열이며 사용자가 입력을 하지 않을 경우에 리턴되는 값이다. 사용자가 입력한 문자

열이 InputBox의 리턴값으로 전달된다. 더 길게 설명할 필요도 없이 간단히 예를 들어 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Response:string;
begin
  Response:=InputBox('질문','당신의 이름을 입력하십시요','강감찬');
end;
```

사람의 이름을 입력받는 대화상자를 만든다. 입력된 결과를 대입받기 위해 문자열 변수 Response를 선언하고 InputBox의 리턴값을 대입받았다. 대화상자의 모양은 다음과 같다.




디폴트 문자열이 '강감찬'으로 주어졌으므로 대화상자가 처음 열릴 때 대화상자 내의 에디트 박스에 '강감찬'이라는 문자열이 나타난다. 사용자는 OK 버튼을 눌러 이 값을 그대로 받아 들이거나 아니면 수정한 후 OK 버튼을 누른다.

이 함수외에 문자열을 입력받는 함수도 있다. InputBox 프로시저는 사용자가 어떤 버튼을 눌렀는지를 알 수 없지만 이 함수는 리턴값으로 사용자가 값을 입력했는지, 취소했는지를 알려준다.

```
function InputQuery(const ACaption, APrompt: string; var Value: string): Boolean;
```

리턴값이 True이면 사용자가 입력을 한 것이고 리턴값이 False이면 입력을 취소한 것이다. 입력받을 문자열 변수를 선언하고 이 변수를 세 번째 인수로 전달해 주면 이 문자열에 사용자가 입력한 문자열을 대입해 준다. 간단한 코드 예를 보이면 다음과 같다.

```
var
  str:String;
begin
  if InputQuery('질문','당신의 이름을 입력하십시요',str)=True then
    입력받은 이름 사용
  else
    사용하지 않음;
```

```
end;
```

# 바. 표준 함수들

함수와 프로시저는 반복되는 부분을 정의하여 여러 번 사용할 수 있도록 해주며 프로그램의 구조를 더욱 더 논리적으로 구성할 수 있도록 해준다. 사용자는 필요에 따라 얼마든지 함수와 프로시저를 만들어 쓸 수 있을 뿐만 아니라 델파이가 제공하는 표준 함수들도 사용할 수 있다. 몇 가지 대표적인 표준 함수들에 대해 정리해 보자.

**표** 델파이가 제공하는 표준 함수

| 함수 | 설명 |
|---|---|
| IntToStr(X) | 정수 X를 문자열로 바꾼다. |
| Abs(X) | X의 절대값을 구한다. |
| Time | 현재 시간을 구해준다. |
| Sin(X) | X의 사인값을 구한다. |
| Length(S) | 문자열 S의 길이를 구한다. |
| FileSize(F) | 파일 F의 크기를 구한다. |
| Str(X,S) | 숫자 X를 문자열로 바꾸어 S에 저장한다. |
| Random(R) | 0~R 사이의 난수를 만든다. |

이 중에서 정수를 문자열로 바꾸는 IntToStr 함수는 이미 앞에서 여러 번 사용해 본 적이 있다. 레이블의 Caption 속성은 문자열형이므로 정수값을 곧바로 대입할 수 없다. 그래서 IntToStr 함수로 정수값을 문자열형으로 바꾸어 준 후 대입하는 것이다.

여기서는 간단히 정리만 해 두기로 하고 나머지는 부록의 함수 레퍼런스를 참고하기 바란다. 함수가 어디 한두 개도 아니고 그렇다고 세 개도 아니고 여러 수백 개나 되기 때문에 본문에서 개별 함수들을 일일이 다룰 수가 없다. 레퍼런스에는 다음과 같이 함수의 문법과 설명 및 예제에 관해 참 친절하게 정성스럽게 정리되어 있다.

### IntToStr

SysUtils Unit

**문법** : function IntToStr(Value: Longint): string;

정수를 문자열로 바꾸어 리턴한다. 정수값을 레이블이나 에디트 박스에 출력하고자 할 경우 직접 그 값을 출력할 수는 없다. 왜냐하면 레이블의 Caption 속성이나 에디트의 Text 속성은 문자열형이므로 정수형값을 대입받을 수 없기 때문이다. 정수값을 문자열로 바꾸고자 할 경우에는 이 함수를 사용한다. 다음은 Label1에 정수형 변수 Age의 값을 출력한 예이다.

```
Label1.Caption:=IntToStr(Age);
```

단 본문 중간 중간에 꼭 필요한 함수에 대해서는 개별적으로 설명한다. 함수를 공부할 때는 항상 레퍼런스를 기준으로 공부하되 레퍼런스에서 부족한 면은 델파이가 제공하는 도움말을 직접 참조하기 바란다. 영어로 되어 있기는 하지만 델파이 표준 함수에 대해 광범위하고 자세한 도움말이 제공되며 어떤 함수의 경우는 친절하게 예제까지 제공해 준다.

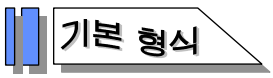

① ShowMessage:간단한 메시지를 전달하는 대화상자
② InputBox:입력용 대화상자

# 6-5 사용자 정의형

델파이가 제공하는 기본적인 데이터형은 정수형, 실수형, 문자열형 등이 있다. 이런 기본적인 데이터형만 사용해도 웬만한 프로그램을 다 작성할 수 있지만 좀 더 프로그램을 효율적으로 짜기 위해 자기만의 데이터형을 만들 수도 있다. 그래서 델파이에서 지원하는 데이터형은 사실 무한하다고 할 수 있다.

## 가. 열거형

열거형(Enumerated type)은 개수가 정해져 있는 집합에서 가능한 몇 가지 값들을 모아 새로운 데이터형을 만드는 것이다. 예를 들어 요일 이름은 월, 화, 수, 목, 금, 토, 일밖에 없고 성별은 남 아니면 여뿐이며 대륙은 아프리카, 남아메리카, 북아메리카, 아시아, 유럽, 오세아니아 6개이다. 개수가 정해져 있고 그 외의 경우가 없을 때 정해진 요소만을 포함하는 새로운 데이터형을 열거형으로 만든다.



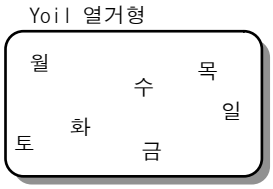

```
type
열거형 이름=(열거요소1,열거요소2,...);
```

열거형 이름과 열거 요소의 이름도 명칭이므로 사용자가 마음대로 정할 수 있다. 그러나 첫 문자에 숫자를 사용하거나 예약어를 사용해서는 안되며 반드시 명칭 규칙에 맞게 만들어야 한다. 몇 가지 열거형의 예를 보자. 변수 선언은 var로 시작되지만 데이터형 선언은 예약어 type으로 시작된다. type 뒤에 4개의 열거형을 선언하였다.

```
type
  Yoil=(Mon,Tue,Wed,Thu,Fri,Sat,Sun);
  JiBang=(KyungSang,  JunLa, KyungKi, KangWon, ChoongChung, JeJu);
  Depart=(Study,Manage,Govern,Publish);
  Sex=(Mail,Femail);
```

**그림**

가능한 값을 모아 열거형을 만든다.




가능한 값이 이 일곱가지 밖에 없다.

김 판 형 상

상요일이나 판요일은 절대로 있을 수 없다.

월~일까지 7개의 가능한 값만을 가지는 Yoil 열거형, 우리나라 8도를 나타내는 JiBang 열거형, 부서를 나타내는 Depart, 그리고 성별을 나타내는 S어쩌구 열거형 등 4개의 열거형을 선언하였다. 사용자가 새롭게 정의한 열거형도 데이터 타입의 일종이며 정수형이나 문자열형 등과 마찬가지로 변수를 선언하고 사용한다. var 선언부에서 기본 데이터형 변수를 선언하는 것과 같은 방법으로 사용자 정의형의 변수를 선언한다.

```
var
  i:integer;      {정수형 변수 i 선언}
  Home:JiBang;    {JiBang 형 변수 Home 선언}
  SoSok:Depart;   {Depart 형 변수 SoSok 선언}
```

만들어진 변수를 사용하는 방법도 기본형과 동일하다.

```
Home:=JeJu;           {대입}
if Depart=Study ...   {비교}
case Yoil of          {선택}
  Mon:...
  Tue:...
```

열거형은 내부적으로 정수로 기억된다. 열거형 선언에서 나타나는 순서대로 열거 요소에 번호가 매겨진다. Yoil형을 예로 들어 Mon이 0의 값을 가지며 Tue가 1, Wed가 2 등등 순서대로 정수값을 가진다. 사실 일부러 열거형을 만들어 사용하지 않고 정수형을 사용해도 똑같은 프로그램을 작성할 수 있다. 그럼에도 불구하고 왜 이런 열거형을 굳이 만들어가며 사용할까? 그 이유는 사람의 두뇌 구조가 숫자를 기억하는 것보다 문자를 기억하는데 더 익숙해져 있기 때문이다.

게임을 하나 만드는데 주인공이 가질 수 있는 아이템이 창, 총, 방패, 칼, 수류탄, 마술봉 이렇게 여섯 가지가 있고 무슨 아이템을 가지고 있는지를 변수 nowitem에 보관한다고 해보자. nowitem이 정수형이라면 창은 0, 총은 1 등과 같이 각각의 아이템을 정수와 대응시켜 주어야 한다. 코드가 다음과 같이 작성

될 것이다.

```
nowitem:=2;       {현재 방패를 가짐}
if nowitem=5 ...  {현재 마술봉을 가지고 있으면}
case nowitem of    {현재 가지고 있는 아이템에 따라 개별 처리}
  0:...
  1:...
```

이렇게 정수를 사용하면 어떤 수가 어떤 아이템과 대응되는지 쉽게 파악하기가 어려워지며 실수할 가능성도 농후해진다.

```
nowitem:=8;
```

8이라는 아이템이 없는데도 엉뚱한 값을 대입하는 실수를 하면 그 다음부터 프로그램은 정신을 못차리고 미로로 빠져 버릴 것이다. 이런 경우 열거형을 선언하여 사용하면 좀 더 기억하기 쉽게 코드를 작성, 관리할 수 있으며 실수의 위험도 막아준다.

```
type
  Titem=(chang,chong,bang,kal,soo,masool);
var
  nowitem:Titem;

nowitem:=bang;      {현재 방패를 가짐}
if nowitem=masool ... {현재 마술봉을 가지고 있으면}
case nowitem of     {현재 가지고 있는 아이템에 따라 개별 처리}
  chang:...
  chong:...
```

주인공이 가질 수 있는 아이템의 종류가 제한적으로 정해져 있으므로 Titem이라는 열거형을 선언하였다. 그리고 Titem형의 변수 nowitem을 선언하고 실제 코드에서 열거 요소를 사용하여 nowitem에 값을 대입하거나 조사한다. 숫자가 아닌 문자로 된 열거 요소를 사용하므로 훨씬 더 기억하기도 쉽고 읽기도 쉽다.

만약 열거형 변수에 열거 요소 이외의 값을 대입하려고 하면 컴파일이 안되므로 실수를 미연에 방지해 주기도 한다. 열거형은 기억의 용이함, 실수의 방지를 위해 만들어진 데이터형이며 숫자로 써야 할 값을 열거 요소로 대신할 뿐이다. 초보자가 흔히 잘못 생각하는 것 중 하나가 열거 요소의 이름을 실행중에도 사용할 수 있다고 생각하는 것이다.

```
Edit1.Text:=nowitem;
```

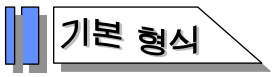
이렇게 쓰면 에디트 컴포넌트에 현재 가지고 있는 아이템의 이름이 나타날 것 같지만 실행중에 nowitem은 정수형이며 Text 속성은 문자열형이므로 상호 데이터형이 맞지 않아 대입할 수 없다. 열거 요소의 이름은 컴파일러가 컴파일할 때만 사용할 뿐이며 실행 파일에는 모두 정수로 번역된다. 만약 꼭 위와 같이 코드를 작성하고 싶다면 Case문을 사용해야 한다.

```
case nowitem of
  chang:Edit1.Text:='chang';
  chong:Edit1.Text:='chong';
  ...
```

어쩌면 이런 코드가 무척 무식해 보일지 모르겠다. 열거 요소의 이름을 꼭 사용하고자 한다면 배열에 열거 요소 이름을 기억시켜 놓고 배열 첨자로 열거형을 사용하는 것이 가장 현명한 방법이다.

# 나. 부분 범위형

부분 범위형(Subrange type)은 정수형, 문자형, 열거형의 값 중 연속된 일부분의 값만을 취할 수 있는 데이터형이다. 변수가 가질 수 있는 값의 범위를 일정 범위에만 제한하여 실수를 방지하려고 할 때 부분 범위형을 사용한다. 예를 들어 날짜를 기억하는 변수라면 정수형을 사용하되 최소값 1에서 최대값 31까지의 값만 가지며 그 외의 값이 대입되면 논리적으로 에러가 된다.




부분 범위형 이름=최소값..최대값;

부분 범위형 변수가 가질 수 있는 최소값과 최대값을 ".."의 양쪽에 적어주면 된다. 실제의 예를 보자.

```
type
  Day=0..31;
  angle=0..359;
```

```
Sigan=1..12;
Height=160..185;
HexaDigit='A'..'F';
```

날짜나 각도, 시간 등과 같이 제한된 범위를 가지는 값과 사람의 키를 160~185로 의도적으로 제한하고자 할 경우 부분 범위형이 사용된다. 열거형을 정의한 후 그 열거 요소 중의 일부분을 부분 범위형으로 만들어 사용할 수도 있다.

```
type
 Cigar=(Sol, ChungJa, PakJa, HansanDo, PalPal, Glory, Hanaro);
 Mycigar=PalPal..Hanaro;
```

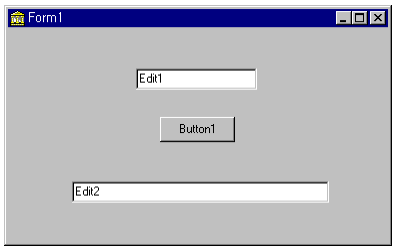
열거형에 여러 개의 담배 이름이 있지만 그 중 일부분을 택해 또다른 데이터형을 만든 것이다. 부분 범위형도 변수를 선언하거나 사용하는 방법이 일반 변수와 동일하다. 단 범위에 속하지 않는 값이 대입될 경우 Constant out of range 에러가 발생하여 실수를 방지해 준다.

```
type
 Day=0..31;
var
 Toady:Day;
 Today:=12;     적법하다
 Today:=33;     에러 발생
```

# 다. 집합형

6jang
set1

집합형(Set type)은 하나의 데이터형에 속한 원소들 중 일부를 모아 놓은 것이다. 정수형, 문자형, 열거형, 부분 범위형의 데이터 타입 중 일부 원소의 집합으로 정의되며 특정값이 그 집합에 속하는가 아닌가를 체크할 때 사용한다. 실수형이나 문자열형은 개수가 정해져 있지 않은 무한집합이므로 집합형에 사용할 수 없다. 집합형을 사용하여 영어의 모음을 가려내는 예제를 작성해 보자. 폼에 다음 그림과 같이 에디트 두 개, 버튼 하나를 배치한다.




정확한 선언위치를 알고 싶으면 배포 CD 의 소스 파일을 참고하기 바란다.

문자형(Char) 원소의 집합형인 TMoem을 다음과 같이 선언한다. 선언하는 위치는 소스의 var 바로 윗부분이며 코드 에디터를 열어 직접 입력해 주어야 한다.

```
type
  TMoem = set of Char;
```

TMoem 형의 변수는 문자형 상수의 집합을 가지게 된다. set of 다음에는 정수형, 문자형, 열거형 또는 부분 범위형 중 하나의 타입이 올 수 있다. 버튼의 OnClick 이벤트를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Moem:TMoem;
begin
  Moem:=['A','E','I','O','U','a','e','i','o','u'];
  if Edit1.Text[1] in Moem then
    Edit2.Text:='첫자가 모음입니다.'
  else
    Edit2.Text:='첫자가 자음입니다.';
end;
```

TMoem을 문자형의 원소를 가지는 집합형으로 정의하고 TMoem형의 변수 Moem을 만든다. 집합형의 원소를 정의할 때는 [ ] 괄호 안에 원소들을 나열해 주면 된다. Moem에 영어의 모음에 해당되는 다섯 글자의 대문자와 소문자를 모아두고 in 연산자를 사용하여 Edit1에 입력된 첫 문자가 모음인지를 조사해 보고 결과를 Edit2에 출력한다. in 연산자는 A in B 형식으로 사용되며 변수 A가 집합 B에 속하는지 살펴보고 속하면 참(True)을 리턴하고 그렇지 않으면 거짓(False)을 리턴한다.

# 라. 배열형

프로그램을 짜다보면 같은 의미의 값을 담는 변수가 여러 개 필요한 경우가 있다. 성적 처리 프로그램의 성적을 기억하는 변수가 대표적인 경우라 할 수 있는데 열 명의 학생이 있고 각 학생의 국어 성적을 기억하는 변수 10개를 K라는 접두어를 붙여 다음과 같이 선언하였다.

```
var
  K1,K2,K3,K4,K5,K6,K7,K8,K9,K10:Integer;
```

K1이 1번 학생의 국어 성적, K2가 2번 학생의 국어 성적, ... K10이 10번 학생의 국어 성적을 보관하는 정수형 변수이며 이렇게 하더라도 값을 기억하는 기능상의 문제는 없다. 하지만 개별 변수가 많아지다 보면 여러 가지 복잡한 문제가 발생한다. 10명의 국어 성적을 모두 합산하여 HAP이란 변수에 대입하려면 다음과 같은 코드를 쓸 수밖에 없다.

```
HAP:=K1+K2+K3+K4+K5+K6+K7+K8+K9+K10;
```

또한 성적을 프린터로 출력해 주는 함수 PRINTK가 있다면 PRINTK(K1) 에서 PRINTK(K10)까지 일일이 호출해 주어야 한다. 배열은 이런 불편한 점을 개선하여 논리적으로 같은 의미를 가지는 변수군(群)을 하나의 이름으로 통합하여 한꺼번에 다룰 수 있도록 해준다. 배열의 정의를 문장화해 보자.

**배열 : 같은 형의 변수들을 하나의 이름으로 모아 놓은 유한집합.**

이 정의에서 가장 중요한 말은 "같은 형"이라는 말이다. 정수형이면 정수형끼리만 모으고 실수형이면 실수형끼리만 모으지 정수형, 실수형, 문자형들을 한꺼번에 같은 배열에 모을 수는 없다. 배열을 선언하는 기본 형식은 다음과 같다.

### 기본 형식

```
var
  배열이름:array [처음첨자..끝첨자] of 배열요소형;
```

array와 of는 미리 정해진 예약어이다. 배열 이름은 명칭이므로 변수나 함수의 이름과 마찬가지로 사용자가 마음대로 정할 수 있다. 첨자란 배열 요소의 번호를 말하며 일반적으로 정수가 사용된다. 배열 요소형이란 배열을 이루는 원소들의 데이터형을 말하며 Integer, Single, Char 등이 사용된다. 배열을 사용하여 성적 처리의 예를 다시 작성해 보자. 일단 변수의 선언은 배열 하나만 선언하면 되므로 아주 간단하다.

```
var
  K:arrary [1..10] of Integer;
```

이 선언에 의해 K라는 배열이 만들어지고 이 배열은 K[1]~K[10]까지 10개의 정수형 변수를 한꺼번에 포함한다. 단 변수만 만든 것이기 때문에 아직 초기화는 되지 않았다. 어떤 값을 대입해 주기 전에는 배열 요소들이 담고 있는 값은 쓰레기값(초기화가 안된 알 수 없는 값)이다. K[1]~K[10]까지 10개의 변수는 일반적인 정수형 변수와 같은 성격을 가지며 개별적으로도 취급할 수 있다. 배열 요소값을 참조하려면 배열의 이름과 꺽쇠 괄호(bracket), 그리고 첨자(index)를 같이 사용해야 한다. 합산 루틴은 다음과 같이 된다.

```
for i:=1 to 10 do
  HAP:=HAP+K[i];
```

배열 요소를 지정하는 첨자에는 K[1], K[5] 등과 같이 상수를 쓸 수 있음은 물론이고 K[i]와 같이 다른 변수를 사용할 수도 있다. 심지어는 K[i*2+j] 등과 같은 수식도 첨자로 쓸 수 있다. 그래서 학생이 10명이건 100명이건 상관없이 for루프를 사용하여 반복적인 처리가 가능하다.  성적을 출력하는 함수 호출도 루프의 제어 변수를 첨자로 사용하여 반복 호출하면 된다.

```
for i:=1 to 10 do
  PRINTK(K[i]);
```

파스칼의 배열이 가지는 일반적인 속성은 다음과 같다.

❶ 반드시 같은 형의 데이터를 모아야 배열이 될 수 있다. 만약 다른 형의 변수들을 하나의 이름으로 묶어서 사용하고 싶다면 잠시 후에 배우게 될 레코드형을 사용해야 한다.

❷ 배열 요소로는 어떤 데이터형도 가능하다. 정수형, 실수형, 문자형 등의 파스칼이 제공하는 기본형은 물론이고 집합형, 열거형, 레코드형 등 사용자가 정의한 데이터형도 배열 요소로 사용할 수 있다. 다음은 열거형의 변수 다섯 개를 모아 열거형 배열 MyDay 를 선언한 것이다.

```
type
  Yoil=(Mon,Tue,Wed,Thu,Fri,Sat,Sun);
var
  MyDay:array [1..5] of Yoil;
```

심지어는 배열 요소로 배열 자체가 사용될 수 있다. 배열형도 하나의 사용자 정의형이므로 배열 요소로 사용될 수 있다. 이런 특수한 경우를 다차원 배열이라고 하는데 잠시 후에 알아본다.

❸ 범위값은 사용자 마음대로 설정할 수 있다. C 언어의 경우 배열의 처음이 반드시 0 부터 시작해야 한다는 제약이 있지만 파스칼에서는 시작값으로 0 은 물론 1, 2 등의 자연수, 심지어 음수까지도 가능하다.

```
var
  MyArray1:array [0..10] of Integer;
  MyArray1:array [5..10] of Integer;
  MyArray1:array [-10..10] of Integer;
```

5 번 학생부터 10 번까지의 학생만 다루고자 할 경우, 배열의 첨자와 학생의 번호를 완전히 일치시킬 수 있어서 좀 더 직관적이고 분명한 관계를 표현할 수 있다.

❹ 배열의 첨자로는 서수적(Ordinal) 데이터가 모두 사용 가능하다. 즉 첨자로 사용되는 값이 정수와 호환되는 데이터이기만 하면 된다는 뜻이며 열거형, 부분 범위형을 첨자로 사용할 수 있다. 부서별 판매 실적을 배열에 기억시키고자 한다면 다음과 같은 배열을 만들어 준다.

```
type
```

```
  Depart=(Study,Manage,Publish,Govern);
var
  SilJuk:array [Study..Govern] of Integer;
```

Study의 실제값이 0이고 Govern의 실제값이 3이므로 생성되는 배열 변수는 [0..3]까지 4개이다. 그러나 배열 첨자로 실수형은 절대로 안된다. 왜냐하면 배열 요소의 개수를 정할 수가 없으며 배열 전체의 크기가 무한대가 되어 버리기 때문이다.

## ■ 다차원 배열

다차원 배열이란 배열의 배열, 즉 배열의 원소인 배열 요소가 또 배열인 형태이다. 배열이 가지는 첨자의 개수에 따라 배열의 차원이 결정되며 첨자가 두 개면 이차원 배열, 첨자가 세 개면 삼차원 배열이라 하고 이차 이상의 배열을 한꺼번에 뭉떵거려 다차원 배열이라고 한다. 다차원 배열의 선언은 일차원 배열 형식을 조금만 변형하면 된다.

```
Sing:=array [1..3] of Integer;
```

이런 일차원 배열을 배열 요소로 가지는 이차원 배열을 다시 만든다면

```
Multi:=array[1..5] of Sing;
```

이렇게 될 것이고 Sing을 원래 선언문으로 바꾸면 다음과 같이 된다.

```
Multi:=array[1..5] of array [1..3] of Integer;
```

정수형 변수를 3개 가지는 배열을 5개 가지는 이차 배열 Multi를 선언한 것이다. 말 그대로 배열의 배열인 셈인데 좀 더 간단하게 표현하여 첨자를 콤마로 끊어 표현할 수도 있다.

```
Multi:=array[1..5,1..3] of Integer;
```

훨씬 더 간단해 보이며 직관적이다. 이렇게 선언된 2차원 배열을 참조하려면 당연히 첨자가 두 개 필요하다. Multi[1,3]:=5; 또는 Num:=Multi[4,1]; 등과 같이 첨자를 콤마로 끊어 사용할 수도 있으며 Multi[1][3]:=5; 와 같이 아예 꺽쇠 괄호를 분리해도 된다. 일단 이렇게 이차원 배열을 선언하고 배열에 값을 대입해 주면 배열을 사용하여 여러 가지 편리한 계산을 할 수 있다.

참고하세요

배열의 배열이 곧 이차원 배열이다. 표현 형식이 다를 뿐 사용하는 방법은 동일하다. 그러나 엄격하게 따지면 차이가 전혀 없는 것은 아니다. 어떤 점이 다른가 하면 첨자가 주어질 때 배열 요소의 위치를 계산하는 첨자 연산 방법이 약간 다르다. 배열의 배열은 일차 배열을 먼저 계산하고 하위 배열 요소를 찾지만 이차원 배열은 두 첨자를 한꺼번에 계산하여 배열 요소를 찾는다.

C는 배열의 배열 형식을 사용하며 BASIC은 이차원 배열을 사용하고 파스칼은 둘 다 사용한다. 어셈블러에서는 프로그래머가 직접 계산을 하므로 정해진 형식은 없지만 대개의 경우 이차원 배열 형식을 사용한다. 이런 언어 비교 이론은 언어 내부의 문제일 뿐이므로 사용자가 굳이 신경쓰지 않아도 된다. 외부적인 사용 방법은 전혀 차이가 없다.

이차원 배열을 사용하면 좀 더 복잡한 자료 형식을 표현할 수 있다. 성적처리도 여러 학생에 대해, 여러 과목에 대해 하나의 변수로 기억시키고 계산할 수 있다.

```
Sung:array [1..5,1..10] of Integer;
```

이 선언에 의해 50개의 정수형 변수가 만들어지며 다섯 과목, 학생 10명의 성적을 기억시킬 수 있다. 메모리에는 다음과 같이 기억된다.

그림
이차원 배열이 메모리에 기억된 모양

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 78 | 82 | 84 | 86 | 92 | 79 | 91 | 90 | 83 | 88 |
| 2 | 95 | 96 | 95 | 90 | 89 | 88 | 94 | 95 | 100 | 99 |
| 3 | 75 | 77 | 77 | 74 | 76 | 72 | 73 | 65 | 71 | 70 |
| 4 | 58 | 59 | 45 | 62 | 52 | 78 | 70 | 62 | 63 | 65 |
| 5 | 75 | 79 | 85 | 85 | 88 | 88 | 89 | 92 | 91 | 93 |

일단 이렇게 기억시켜 놓기만 하면 가로로 더해 과목별 합계나 평균을 구하고 세로로 더해 학생 개인별 합계, 평균을 구할 수 있으며 이중 루프를 사용하면 한꺼번에 학급 전체 총점을 구한다.

```
for i:=1 to 5 do
```

```
for j:=1 to 10 do
 sum:=sum+sung[i,j];
```

이차원 배열은 바둑, 장기, 체스 등의 게임에서 판에 놓여진 말의 종류를 기억하기 위해서도 사용되며 메뉴의 모양을 기억하기 위해서도 사용된다. 또한 복잡한 정보를 기억시키기 위해 사용되기도 한다. 그야말로 프로그래밍에 없어서는 안될 재주 많은 팔방미인이다.

첨자의 개수를 늘려주면 이차원 배열뿐만 아니라 삼차, 사차 얼마든지 차원을 높일 수 있다. 단 배열은  규모가 큰 변수 덩어리이므로 메모리를 많이 소비한다는 것을 잊어서는 안된다. array [1..100,1..100] of Integer; 는 벌써 변수 하나로 4만바이트의 메모리를 차지해 버린다. 꼭 필요한 만큼의 크기를 정해 아껴서 써야 하며 특히 문자열 배열일 경우는 문자열의 길이를 최소화하여야 한다. 삼차원 배열이나 사차원 이상의 배열도 가끔 사용된다. 삼차 이상의 배열을 사용하는 방법도 이차원 배열을 사용하는 방법과 동일하다.

## ■ 문자열형

파스칼은 String이라는 문자열형을 지원한다. 파스칼보다 더 많이 사용되고 있는 C에서는 문자열형이라는 데이터형이 없다. 왜 그런가하면 문자열이라는 것이 문자 여러 개를 모아 놓은 문자 배열로 구현되기 때문에 굳이 컴파일러 차원에서 지원하지 않아도 프로그래머가 직접 만들어서 사용할 수 있으며 오히려 문자열 가공에 더 많은 자유를 부여해 주기 때문이다. 파스칼은 문자열형을 지원하기도 하지만 문자 배열 형태의 문자열도 지원한다.

```
type
 MyString=array [0..128] of Char;
var
 MyName:Mystring;
```

이렇게 선언하면 MyName은 문자 배열형 변수가 된다. 문자열이 문자의 배열임을 이용하면 배열 요소를 접근하는 것과 동일한 방법으로 문자열의 개별 문자를 읽거나 변경시키는 것이 가능하다. MyStr이 string형이며 'Korea'라는 문자열을 대입했을 때 다음과 같이 개별 문자를 읽을 수 있다.

```
MyStr[1] → 'K'
MyStr[2] → 'o'
MyStr[3] → 'r'
```

## ■ 두 가지 형태의 문자열

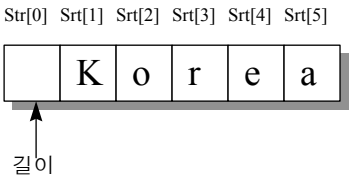
델파이에서는 두 가지 형태의 문자열을 사용한다. 하나는 전통적인 파스칼 언어에서 사용하던 ShortString 문자열이며 나머지 하나는 델파이 2.0버전부터 지원되는 ANSIString 형이다. 두 문자열 형태의 가장 큰 차이점이라 한다면 문자열의 길이를 나타내는 방법이 다르다는 점이며 또한 문자열 길이의 상한값도 다르다. 두 문자열 형태 중 어떤 문자열이 사용되는가는 $H 컴파일 지시자의 설정에 따라 달라지는데 디폴트로 이 옵션은 $H+로 되어 있으므로 ANSIString이 사용된다. 옵션에 따라 사용되는 문자열이 다를 뿐 두 문자열형의 데이터 형은 모두 String이다.

### ShortString

문자열의 끝을 별도로 표시하지 않으며 문자 배열의 선두에 문자열의 길이를 정수값으로 기억시켜 둔다. 문자열을 읽을 때는 선두를 먼저 읽어 길이를 구한 후 길이만큼 읽어들인다. 문자열 중간에 어떤 문자라도 포함할 수 있지만 문자열의 길이를 1 바이트로 기억하므로 255 문자 이상을 넘을 수는 없다. 고전적인 파스칼에서 사용하는 문자열 형식이며 델파이 1.0 에서 디폴트로 지원했던 문자열 형식이다. Str 이 파스칼형의 문자열이며 'Korea'라는 문자열이 기억되어 있을 때 Str[1]은 'K'이며 Str[2]는 'o'이다. Str[0]는 문자열의 문자 길이를 기억하는 특별한 용도로 사용된다. 곧바로 읽으면 문자로 읽혀지기 때문에 Ord(Str[0]);로 읽어야 문자열의 길이를 정수값으로 구할 수 있다.

**그림**

파스칼형 문자열 첫 칸에 길이가 기억된다.




이 문자열으로 사용하려면 {$H-} 컴파일러 지시자를 주면 되는데 사실 이 형태의 문자열을 사용해야 할 이유는 거의 없다.

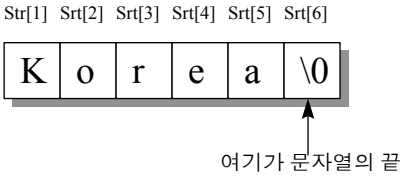
### ANSIString

문자열의 끝에 0 을 넣어 끝임을 표시한다. 따라서 문자열의 길이는 문자열의 처음에서 0 이 나타날 때까지가 되며 255 이상의 길이를 가질 수도 있다. 반면 0 이 문자열의 끝 표시로 사용되므로 문자열 중간에 사용될 수는 없다. 여기서 0 이란 정수 0 을 말하는 것이지 문자 '0'을 말하는 것이 아니므로 혼돈하지 말기

바란다. MS-DOS 에서부터 이 형식을 지원하기 시작하여 윈도우즈도 이 형식의 문자열을 사용하며 C 언어에서 사용하는 문자열 형식이기도 하다. 'Korea'라는 문자열을 가질 경우 메모리에는 다음과 같이 기억된다.

**그림**

널 종료 문자열 끝에 0 을 붙여 문자열 끝을 표시한다.




ANSIString 형의 가장 큰 특징으로는 동적으로 메모리를 할당한다는 점이다. 대입된 문자열 길이만큼 메모리를 사용하되 만약 문자열 길이가 변경되면 알아서 메모리를 늘려 할당한다. 예를 들어 문자열에 최초 'Korea'가 대입되었다면 이 문자열은 5 자분만큼의 메모리를 할당할 것이다. 그런데 이 상태에서 'Beautiful Korea'로 문자열을 바꾸어 대입하면 15 자로 그 길이를 알아서 늘린다. 그래서 꼭 필요한만큼의 메모리만을 사용하게 되며 길이 한계도 없다.

ANSIString 형을 사용하려면 {$H+} 컴파일러 지시자를 사용하되 이 옵션은 디폴트로 선택되어 있으므로 그냥 String 형의 변수를 만들어 사용하기만 하면 된다.

**문자열 변환**

앞에서 보인 두 문자열 형태외에도 델파이에는 제 3 의 문자열 표현 방법인 문자 배열이라는 것이 있다. 문자 배열형은 윈도우즈 API 함수가 사용하는 전형적인 널 종료 문자열이며 API 함수의 인수로 곧바로 사용할 수 있다. 만약 String 형의 문자열을 API 함수에 대입하고자 한다면 이 문자열을 PChar 로 타입 캐스팅해서 사용하면 된다.

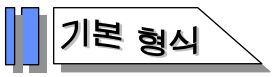
```
var
str:string;
begin
 str:='korea';
 SetWindowText(handle, PChar(str));
end;
```

# 마. 레코드형

배열은 여러 개의 변수를 하나의 이름으로 통합하여 관리, 제어할 수 있다는 점에 있어서 상당히 유용한 테이터형이지만 반드시 같은 데이터형끼리만 모을 수 있다는 특징이 있다. 배열의 이런 단점에 비해 여러 가지 다른 형태의 변수들을 묶을 수 있는 것이 레코드형이다. 그렇다고 해서 배열이 레코드보다 열등하다는 것은 아니다. 레코드(Record)의 정확한 정의를 내려보자.

**레코드 : 논리적으로 관련이 있는 서로 다른 형태의 변수들을 모아 놓은 집합**

배열과 가장 두드러진 차이점은 다른 형태의 변수들을 묶을 수 있다는 점이다. 그래서 레코드형을 선언하는 기본 형식이 다소 복잡하다.




```
type
 레코드이름=record
  필드1:데이터형;
  필드2:데이터형;
  필드3:데이터형;
  필드4:데이터형;
 end;
```

필드란 레코드에 속한 개별 변수들을 일컫는 말이며 레코드의 구성 요소이다. 필드는 어떠한 데이터형이라도 상관이 없으며 개수 제한도 없다. 정수, 실수, 문자열 등의 기본형과 배열, 열거형 등도 가능하다. 심지어는 레코드가 다른 레코드의 필드로 사용되기도 한다. 즉 레코드끼리 중첩도 가능하다는 얘기다. 레코드를 사용하는 가장 좋은 예는 주소록이다. 한 사람의 신상을 기억하는 주소, 전화번호, 나이, 성별 등은 한 사람의 신상에 관한 정보라는 면에서 유기적인 관련성을 가지며 서로 다른 데이터형이므로 레코드형으로 정의할 수 있다. 다음은 사람의 신상을 기억하는 레코드형 JusoRec를 선언한 예이다.

```
type
 JusoRec=record
  Name:String[20];
  Tel:String[15];
  Age:Integer;
  Male:Boolean;
 end;
```

이름을 기억하는 Name, 나이를 기억하는 Age 등의 필드가 있고 보다시피 각 필드의 데이터형이 각각 다르다. 그래서 배열로는 이런 자료를 기억할 수 없으며 반드시 레코드형을 사용해야 한다. 레코드형 변수를 만드는 방법은 일반 변수를 선언하는 것과 동일하다.

```
var
 Juso:JusoRec;
```

이 선언에 의해 JusoRec형의 레코드 변수 Juso가 만들어졌다. 이 레코드의 필드를 참조할 때는 레코드명과 필드 사이에 점을 하나 찍어 "레코드.필드"의 형식으로 표현한다.

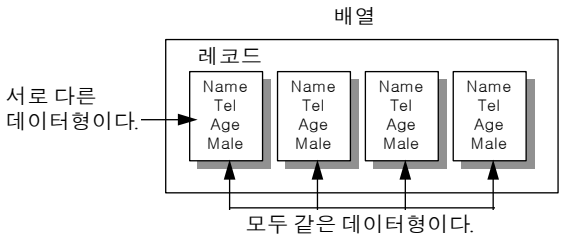
```
Juso.Name:='Kim Sang Hyung';
Juso.Age:=27;
```

여기서 쓰인 점의 의미는 '~레코드에 속한 ~필드'라는 뜻이다. 즉 Juso.Age라는 표현은 Juso레코드에 속한 Age필드라는 뜻이다. 필드는 레코드에 속해있을 뿐 일반 변수와 다를 바가 전혀 없다. Juso.Age는 정수형의 변수이며 정수값을 담고 정수가 사용되는 곳이면 어디든지 사용될 수 있다.

## ■ 레코드 배열

레코드에 속한 필드의 데이터형은 어떤 것이라도 상관없다. 배열이 레코드의 필드가 될 수 있으며 레코드가 레코드의 필드가 될 수도 있다. 또한 배열의 배열 요소도 어떤 데이터형이든지 가능하다. 배열이 레코드를 담을 수 있고 레코드도 배열을 담을 수 있으므로 억지로 만들면 "레코드 배열을 필드로 가지는 레코드의 배열" 따위도 만들 수 있다. 레코드와 배열을 조합하여 만드는 방법 중에 레코드 배열은 실용적으로 많이 사용되는 형태이다. JusoRec의 경우 한 사람의 신상을 담도록 정의되었는데 주소록이란 여러 사람의 신상이 모여 있는 것이므로 레코드 배열이 된다.

그림
레코드 배열의 모양




10명의 주소록을 만들려면 JusoRok 변수를 JusoRok[1]~JusoRok[10]까지 만들어 주면 된다. 다음과 같이 선언한다.

```
var
  JusoRok:array [1..10] of JusoRec;
```

레코드 배열의 경우 레코드의 선택은 첨자로 지정하며 레코드 내에서 개별 필드로의 접근은 "."을 사용한다. JusoRok[3].Name은 세 번째 사람의 이름, JusoRok[5].Age는 다섯 번째 사람의 나이이다.

레코드 배열 내에서 레코드의 선택

↓

JusoRok[3].Name

레코드내에서 필드의 선택

레코드 배열은 크기가 큰 레코드를 크기가 큰 배열로 묶은 것이므로 메모리를 많이 소모한다. 레코드 하나의 크기를 필요한 만큼만 최소로 설계하는 것이 좋다. 특히 문자열 필드의 경우 String[15] 등과 같이 최대한 크기를 제한해야 한다.

## ■ With

레코드 내의 필드를 참조하려면 필드 이름 앞에 일일이 레코드의 이름을 밝혀 주어야 한다.

```
JusoRok[1].Name:='Kim Sang Hyung';
JusoRok[1].Tel:='123-4567';
JusoRok[1].Age:=26;
JusoRok[1].Male:=True;
```

한두 번이면 몰라도 계속 이런 필드 참조문이 반복되면 짜증도 날 뿐더러 작업 효율도 떨어지게 된다. 이럴 때 with로 레코드 이름을 한 번만 밝혀주면 레코드 이름을 일일이 써주지 않아도 된다.

### 기본 형식

```
With 레코드 do
 begin
  필드 참조
 end;
```

with 다음의 레코드 이름이 begin~end 블럭 내의 필드 앞에 일일이 붙여지게 된다. 위의 코드를 With로 다시 작성하면 다음과 같다.

```
With JusoRok[1] do
begin
  Name:='Kim Sang Hyung';
  Tel:='123-4567';
  Age:=26;
  Male:=True;
end;
```

with 블럭 내에서 필드 이름이 일반 변수 이름과 중복될 경우는 필드 이름이 우선 순위가 높다. With 블럭 내에서 Name은 폼의 Name 속성을 말하는 것도 아니며 일반 변수 Name도 아니며 JusoRok[1]의 Name 필드를 말한다. with문은 레코드뿐만 아니라 레코드를 한 단계 더 확장한 오브젝트에도 그대로 사용되므로 잘 알아 두어야 불필요하게 비싼 노동력을 낭비하지 않게 된다.

참고하세요

어느 언어에나 레코드에 해당하는 데이터 타입이 존재한다. 서로 다른 형태의 변수들을 묶는다는 면에서 기능적으로 동일하며 다만 이름만 다를 뿐이다. 다음에 C언어와 파스칼 언어를 비교해 보았다. 두 언어에서 각 용어의 뜻은 완전히 동일하므로 C언어를 먼저 공부한 사람은 참고하기 바란다.

| C 언어 | 구조체 | 멤버 | 멤버 함수 |
|---|---|---|---|

| 파스칼 | 레코드 | 필드 | 메소드 |
|---|---|---|---|

# 바. 타입 상수

델파이에서는 변수를 선언하면서 초기화할 수 없었다. 아니 델파이가 그렇다기보다는 파스칼 언어의 특성이 그렇다.

```
var
  I:Integer=5;
```

이렇게 할 수 있다면 얼마나 좋겠는가만 하고 싶어도 할 수 없었다. 그래서 변수 선언을 따로 하고 변수 정의는 FormCreate 등의 이벤트 코드나 아니면 유닛의 initialization부에서 별도로 해 주어야 한다. 실제 전통적인 파스칼에서는 이런 문제 때문에 변수 초기화를 위해 수백 줄의 코드를 작성해야만 했으며 델파이 1.0까지도 그랬다.

그러나 델파이 2.0 버전부터는 이 문제를 부분적으로나마 해결하였다. 즉 다음과 같이 변수를 선언하면서 초기값을 대입해 줄 수 있다.

```
var
  i:integer=3;
  s:string='korea';
  ar:array [1..3] of string=('dog','cat','cow');
```

그러나 2.0 버전에서 확장된 이 방법도 전역 변수에 한해 초기값을 줄 수 있을 뿐 지역 변수에서는 초기값을 대입할 수 없다. 이런 초기화가 불가능했던 1.0 버전에서는 타입 상수를 사용하여 변수 초기화 문제를 해결하였다.

typed constant: Identifier whose value and type are defined on entering the declaring block

델파이를 만든 볼랜드사에서는 변수를 선언하면서 초기화를 할 수 있도록 하기 위한 대안으로 타입 상수라는 것을 만들었다. 타입 상수(Typed Constant)는 프로그램이 컴파일 되면서 값이 정해지는 상수이기는 하되 메모리를 차지하면서 번지가 변하지 않으며 실행중에 값이 변경될 수 있는 상수이다. 실행중에 변경되면서 무슨 상수냐고 하겠지만 어쨌든 이름은 타입 상수라고 되어 있으며 좀 더 현실적인 이름을 붙여 준다면 "초기화가 가능한 변수"라고 할 수 있다.

타입 상수를 정의할 때는 그래도 문법적으로 상수이므로 const절에서 정의해 주어야 한다. 다음과 같이 상수 이름과 상수의 데이터형, 그리고 초기값을 준다.

```
const
 I:Integer=5;
```

이 선언에 의해 I라는 정수형 상수가 초기값 5로 만들어진다. 실행중에 I:=588; 등과 같이 값을 대입하여 바꿀 수 있다.

사실 정수는 꼭 이렇게 초기값을 주면서 정의해야 할 경우가 드물다. 초기화하는데 큰 어려움이 없기 때문이다. 타입 상수가 정말로 필요한 때는 배열이나 레코드 또는 레코드 배열을 초기화해야 할 경우이다. 배열형 타입 상수를 정의하는 것도 정수형 타입 상수를 정의하는 것과 별로 다를 것이 없다. 길게 설명할 필요없이 예를 보도록 하자.

```
type
 TMyArray=array [1..10] of Integer;
const
 MyArray:TMyArray=(88,96,78,65,87,82,80,94,96,78);
```

배열 요소의 값을 (요렇게, 요렇게) 괄호 안에 집어넣고 초기값을 순서대로 콤마로 끊어 나열해 주면 된다. 레코드형 타입 상수도 비슷한 형식으로 초기화한다.

```
type
 TMyRec=record
  Name:string;
  Age:Integer;
  Male:Boolean;
 end;
const
 MyRec:TMyRec=(Name:'Lee Sun Hee';Age:29;Male:False);
```

"필드:값"의 형식으로 각 필드의 초기값을 정의해 주며 사이 사이에 세미콜론을 삽입해 주어야 한다. 필드는 레코드에 정의되어 있는 순서를 반드시 지켜 주어야 한다. 마지막으로 레코드 배열의 초기값을 정의하는 예를 보자.

```
type
 TMyRec=record
  Name:string;
```

```
    Age:Integer;
    Male:Boolean;
  end;
const
  MyRec:array [1..4] of TMyRec=(
    (Name:'Cho Hye Jin';Age:27;Male:True),
    (Name:'Ha Yong Ja';Age:27;Male:True),
    (Name:'Kim Sang Hyung';Age:22;Male:False),
    (Name:'Kim Sool Tong';Age:35;Male:False)
    );
```

레코드 하나를 ( ) 안에 정의해 주고 각 레코드를 콤마로 구분한 후 전체를 괄호로 한 번 더 싸 준다. 레코드 배열은 간단한 데이터 베이스를 실행 파일에 직접 삽입하고자 할 때 많이 사용된다. 위의 소스에서 const를 var로 바꾸어도 별다른 차이가 없지만 지역 변수일 경우는 var는 불가능하며 반드시 const를 사용해야 한다.

# 사. 대입 호환성

앞에서 살펴본 바와 같이 델파이에는 무수히 많은 데이터형이 존재한다. 이 많은 데이터형들 사이에 어떤 경우는 대입이 가능하고 어떤 경우는 대입이 불가능한 경우가 있는데 대입의 가능성 여부를 대입 호환성(Assignment compatibility)이라고 한다. 대입 호환성이 왜 중요한가하면 실제 프로그래밍에서 반드시 같은 데이터형끼리만 대입하는 것이 아니기 때문이다. 정수형을 문자형 변수에 대입해야 할 경우도 있고 실수형을 정수로 만들어야 할 경우도 있다. 어떤 경우가 대입 가능한가를 알아보자.

❶ 양변의 데이터형이 완전히 일치할 때이다. L 이 Integer 형이고 R 이 Integer 일 때 L:=R 이 가능하며 R:=L 이 가능하다는 얘기다. 너무 당연하다.

❷ 크기가 다른 동일한 데이터형끼리도 대입이 가능하다. 정수형의 경우 표현할 수 있는 크기에 따라 Integer, LongInt 등의 데이터형이 있고 부호없는 Word, Byte 형들이 있는데 크기나 형태가 조금 다르기는 해도 같은 정수형이기 때문에 상호 대입이 가능하다. 실수형의 경우도 Single, Double 등의 데이터형끼리 대입할 수 있다. 단 작은 데이터형 변수에 큰 데이터형 변수값을 대입할 때 일단 대입은 되지만 값이 온전하게 대입되지 않을 수 있다.

```
var
  I:SmallInt;
  L:LongInt;
begin
  L:=100000;
  I:=L;
```

이렇게 대입했을 때 I 와 L 은 상호 대입이 가능하다. 그러나 I 는 십만이라는 큰 숫자를 기억할만큼의 크기를 가지지 못하기 때문에 -31072 라는 엉뚱한 값을 가지게 된다. 이 값은 자리 넘침이 발생하고 남은 값이며 수학적으로는 65536-(100000-65536)이다.

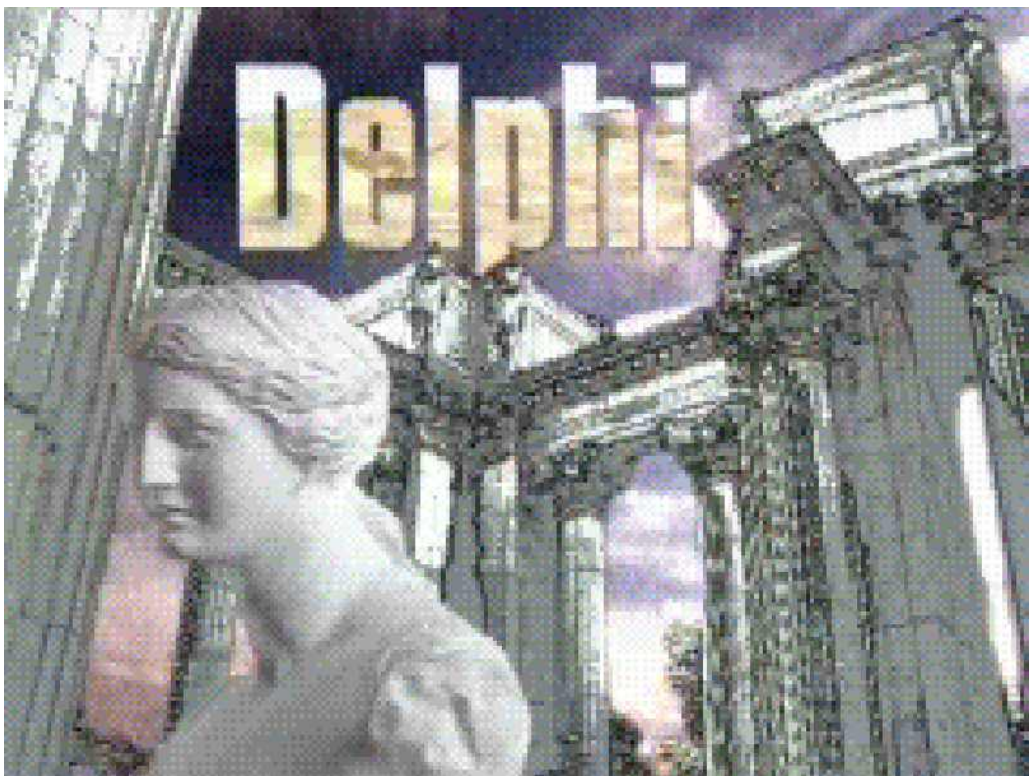
❸ 우변이 좌변의 부분집합일 때 대입 가능하다. R 이 실수형이고 I 가 정수형일 때 정수가 실수의 부분집합이므로 R:=I 와 같이 대입할 수 있다. 그러나 반대로 I:=R 과 같은 대입은 안된다. 실수형 변수는 정수를 기억할 수 있지만 정수형 변수가 실수를 기억할 수 없기 때문이다.

이 외에도 복잡한 대입 규칙이 있지만 이 책 범위를 벗어나는 내용들이므로 생략한다. 이런 문제는 규칙을 일일이 따지는 것보다는 직관적으로 이해하는 것이 훨씬 더 정확하다. 대입 호환성이 조금 복잡하기는 해도 수학적인 상식을 벗어나지는 않기 때문이다. 대입을 꼭 해야 하는데 대입이 안되는 경우는 별도의 해결 방안이 있다. 대표적으로 정수와 문자열을 대입해야 할 경우인데 StrToInt, IntToStr 등의 함수로 우변의 데이터 형을 바꾼 후 대입하면 된다. 실수와 문자열의 경우에도 FloatToStr, StrToFloat 등의 함수가 존재한다.


① 열거형:유한 개수의 원소만으로 이루어진 데이터형
② 부분 범위형:연속된 범위의 값을 원소로 가지는 데이터형
③ 집합형:원소들 중의 일부를 모아 만든 새로운 데이터형
④ 배열:동일한 형의 변수를 모아 놓은 데이터형
⑤ 레코드형:서로 다른 형의 변수를 모아 놓은 데이터형

# 제 7 장

# 컴포넌트 활용



이제 여러분들은 델파이의 기본적인 부분을 어느 정도 알게 되었다. 델파이를 실행시키고 프로젝트를 시작하고 컴파일하고 오류가 있을 경우 수정하는 방법을 알게 되었으며 변수, 함수, 제어 구조에 대한 초보적인 내용을 알았다. 이제 기본을 바탕으로 더 넓은 부분을 연구해 보고 지식을 확장해 나갈 때이다. 델파이는 컴포넌트를 조립하여 프로그램을 만들어 나가므로 컴포넌트 하나하나에 대해 잘 알아야 한다. 이 장에서는 컴포넌트를 개별적으로 하나씩 정복해 나갈 예정이다.

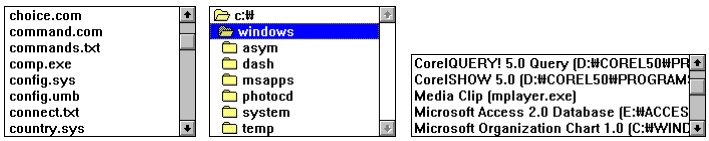
먼저 이 장에서는 standard 페이지에 있는 컴포넌트를 주 대상으로 하여 쉽다고 생각되는 것부터 분석해 보기로 하고 나머지 컴포넌트에 대해서는 이 책의 나머지 부분에서 관련 이론과 함께 알아 볼 것이다. 이 장에서는 개별적인 컴포넌트에 대한 이해와 활용 방법에 대해서도 학습해야겠지만 그보다 더 중요한 것은 컴포넌트에 대한 일반적인 개념 파악을 확실히 하는 것이다.

# 7-1 리스트 박스

## 가. 리스트 박스

리스트 박스(List Box)는 윈도우즈가 제공하는 표준 컨트롤의 하나이며 사용자가 선택할 수 있는 항목(아이템)들을 여러 개 나열해 두고 선택할 수 있도록 해준다. 선택해야 할 대상을 키보드로 직접 입력해 주어야 하는 전통적인 방법보다 선택 대상을 보여주고 마우스로 간단히 선택할 수 있도록 하는 더욱 편리한 방법을 제공해 준다. 다음은 윈도우즈의 표준 프로그램에서 사용하는 리스트 박스이다.

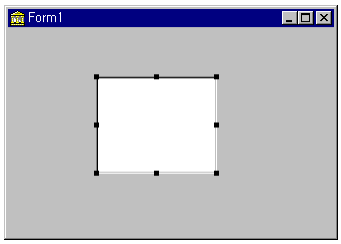
**그림**
여러 가지 모양의 리스트 박스




델파이가 제공하는 리스트 박스 컴포넌트는 윈도우즈의 표준 리스트 박스와 동일한 리스트 박스이다. 컴포넌트 팔레트에서 리스트 박스 컴포넌트를 가져와 폼에 배치하면 아이템이 없는 빈 리스트 박스 하나를 만들어 준다. 아이템이 하

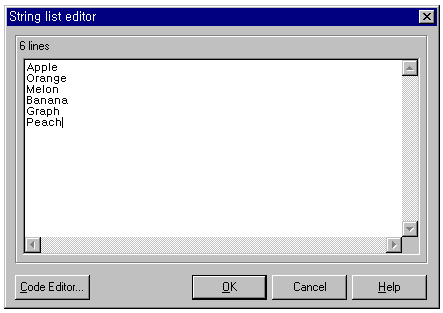
나도 없는 상태에서는 다음 그림과 같이 단순한 직사각형 모양이며 아이템을 담다보면 아이템도 생기고 스크롤 바도 생기고 그림도 넣을 수 있다.

**그림**
리스트 박스 컴포넌트를 폼에 처음 배치한 모양



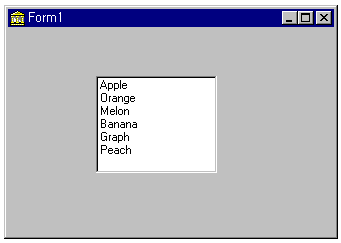

리스트 박스에 담겨지는 아이템은 거의 예외없이 문자열이며 리스트 박스의 Items 속성에 보관된다. 리스트 박스를 배치한 후 아이템을 포함시키려면 Items 속성을 편집하면 된다. 오브젝트 인스펙터에서 Items 속성을 살펴보면 속성값란에 (TStrings)라고 되어 있으며 대화상자형 속성임을 알 수 있다. (TStrings)를 더블클릭하면 다음과 같은 문자열 리스트 편집기가 열린다. 에디터를 사용하듯이 항목들을 입력해 보자.

**그림**
문자열 리스트 편집기



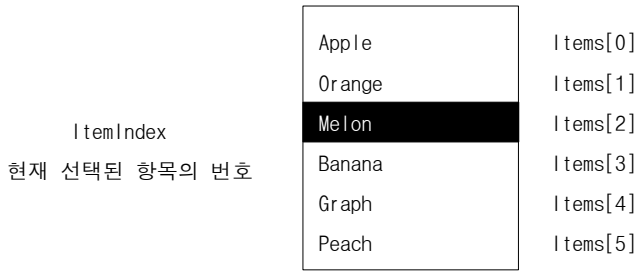

여섯 개의 문자열을 입력한 후 문자열 리스트 편집기의 OK 버튼을 누르면 입력한 문자열이 폼의 리스트 박스에 나타난다.

**그림**
문자열 리스트에 항목이 입력된 모양



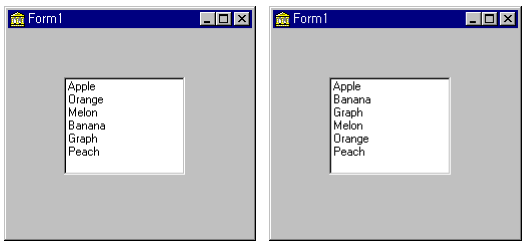

문자열 리스트 편집기는 디자인시에 미리 리스트 박스에 아이템을 포함시키고자 할 때 사용하며 스피드 메뉴의 Load 항목을 사용하여 미리 작성되어 있는

텍스트 파일에서 아이템을 읽어올 수도 있다. 실행시에 리스트 박스에 항목을 추가하거나 삭제할 때는 Add, Delete, Insert 등의 메소드를 사용한다. 리스트 박스에 포함된 항목들은 Items 오브젝트에 배열 형태로 저장되므로 배열을 읽듯이 첨자를 사용하여 항목 문자열을 읽을 수 있다. 예를 들어 ListBox1.Items[0]을 읽으면 리스트 박스의 첫 번째 항목인 'Apple'이 읽혀지며 ListBox1.Items[3]을 읽으면 'Banana'가 읽혀진다.




현재 선택된 아이템의 번호는 리스트 박스의 ItemIndex 속성에 저장된다. 실행중에 사용자가 선택한 항목을 읽으려면 ListBox1.Items[ListBox1.ItemIndex]를 읽으면 된다.

리스트 박스의 속성 중 가장 눈여겨 볼만한 속성은 Sorted 속성이며 이 속성이 False일 경우 아이템이 추가된 순서대로 리스트 박스에 출력되지만 True일 경우 추가된 순서에는 상관없이 알파벳 순으로 정렬해 준다.




리스트 박스가 포함하는 항목이 많을 경우 원하는 항목을 찾아내는 데 시간이 많이 걸리므로 Sorted 속성을 True로 설정하여 찾기 쉽게 해주는 것이 좋다. 불규칙하게 아이템이 널려있는 것 보다는 알파벳 순으로 정렬되어 있는 것이 찾기가 더 쉬울 것이다. 또한 Sorted 속성이 True인 상태에서 새로운 항목이 추가되면 새로 추가되는 항목도 정렬 순서에 맞는 위치에 삽입된다.

Columns 속성을 사용하면 다중 컬럼의 리스트 박스를 만들 수도 있고 Style 속성을 사용하여 리스트 박스에 아이콘을 표시할 수도 있다. 다음은 Columns

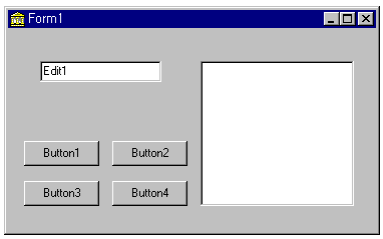
속성을 2로 설정하여 2단으로 구성된 리스트 박스를 만든 모양이다.

**그림**
다중 컬럼의 리스트 박스



7jang
list

에디트와 리스트 박스를 사용하여 리스트 박스에 항목을 추가하거나 삭제할 수 있는 예제를 만들어 보자. 폼에 컴포넌트를 다음과 같이 배치하고 속성을 설정한다.




| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | Name, Caption | ListForm |
| 에디트 | Text | 모두 지움 |
| 위 왼쪽 버튼 | Name | BtnAdd |
| | Caption | Add Item |
| | Default | True |
| 위 오른쪽 버튼 | Name | BtnInsert |
| | Caption | Insert Item |
| 아래 왼쪽 버튼 | Name | BtnDelete |
| | Caption | Delete Item |
| 아래 오른쪽 버튼 | Name | BtnRead |
| | Caption | Read Item |
| 리스트 박스 | | 디폴트 속성 사용 |

에디트 박스에서 입력한 문자열을 리스트 박스에 추가하거나 삽입하는 기능과 리스트 박스에서 선택한 항목을 에디트 박스로 읽어오는 기능, 리스트 박스에서 선택한 항목을 삭제하는 기능을 가지고 있다. 버튼 4개로만 동작하므로 버

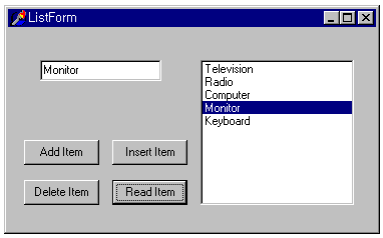
튼의 OnClick 이벤트 핸들러만 작성하면 된다. 일단 작성된 코드를 보자.

```
procedure TListForm.BtnAddClick(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
  Edit1.Text:='';
end;

procedure TListForm.BtnDeleteClick(Sender: TObject);
begin
  ListBox1.Items.Delete(ListBox1.ItemIndex);
end;

procedure TListForm.BtnInsertClick(Sender: TObject);
begin
  ListBox1.Items.Insert(ListBox1.ItemIndex,Edit1.Text);
  Edit1.Text:='';
end;

procedure TListForm.BtnReadClick(Sender: TObject);
begin
  if ListBox1.ItemIndex=-1 then exit;
  Edit1.Text:=ListBox1.Items[ListBox1.ItemIndex];
end;
```

추가, 삽입, 삭제 기능은 모두 Items 오브젝트의 메소드를 사용했다. 메소드 이름이 설명적이므로 기본 형식만 보인다.

```
Add(추가할 문자열);
Insert(삽입할 위치, 삽입할 문자열);
Delete(삭제할 아이템의 위치);
```

Read Item 버튼을 누르면 리스트 박스에서 선택된 항목의 인덱스인 ListBox1.ItemIndex를 조사하여 Items 오브젝트의 아이템을 읽어 에디트 박스에 대입한다. 단 선택이 되어 있지 않은 경우를 고려해야 하므로 읽기 전에 먼저 ItemIndex 속성이 -1이 아닌가 점검해 보아야 한다. ItemIndex 속성은 리스트 박스에서 선택된 아이템의 위치를 나타내되 선택이 되어 있지 않을 경우 -1의 값을 갖는다. 실행중의 모습은 다음과 같다.

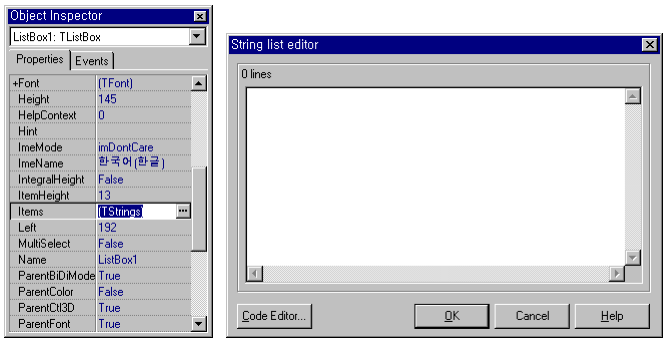



# 나. 문자열 리스트

문자열 리스트는 여러 개의 문자열을 가지고 있는 일종의 문자열 배열이다. 리스트 박스 예제에서 사용한 Items 속성이 문자열 리스트이며 리스트 박스가 포함하고 있는 항목들의 목록을 가지고 있다. 이 외에도 메모 컴포넌트의 Lines 속성, 아웃라인 컴포넌트의 Lines 속성에서 문자열 리스트를 사용한다. 문자열 리스트를 사용하는 속성은 오브젝트 인스펙터에 (TStrings)라고 나타나며 더블 클릭하면 문자열을 입력할 수 있는 문자열 리스트 편집기가 나타난다.

**그림**

오브젝트 인스펙터에 Tstrings 라고 나타나는 속성




디자인시에는 문자열 리스트 편집기를 사용하여 문자열을 편집할 수 있으며 실행시에는 문자열 리스트로 다음과 같은 여러 가지 일을 할 수 있다. 관련된 속성과 메소드 이름을 잘 보아 두도록 하자.

① 리스트에 몇 개의 문자열이 있는지 Count 속성을 검사한다.

② 리스트 내의 특정 문자열을 읽거나 변경한다. 문자열 리스트는 일종의 배열이므로 첨자를 사용하면 원하는 문자열을 얼마든지 읽거나 변경할 수 있다.

③ 문자열의 위치를 검색한다. IndexOf(문자열) 메소드를 사용하면 해당 문자열의 위치를 나타내는 인덱스를 구해 준다.

④ 리스트의 끝에 문자열을 추가(Add 메소드)한다.

⑤ 리스트의 중간에 문자열을 삽입(Insert 메소드)한다.

⑥ 리스트에서 문자열을 이동(Move 메소드)시킨다.

⑦ 리스트에서 문자열을 지운다(Delete 메소드).

⑧ 문자열끼리 복사한다.

일단은 이 정도만 알아 두면 문자열 리스트로 웬만한 작업은 다 할 수 있을 것이다. 좀 더 자세한 정보가 필요한 사람은 도움말에서 TStrings 를 찾아 보기 바란다. 속성, 메소드와 기타 문자열 리스트와 관련된 여러 가지 정보를 얻을 수 있을 것이다.

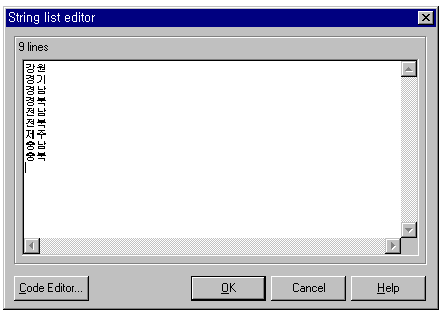
## 다. 전화번호부

7jang
post

리스트 박스를 사용하여 간단한 DDD 목록을 작성해 보자. 우리나라 DDD번호는 시군별로 이루어져 있으며 시, 군 위의 행정 단위가 도이므로 도와 군 두 개의 리스트 박스만 배치해 두고 사용자가 찾고자 하는 지명을 선택할 수 있도록 해주는 것이 좋다. 만약 리스트 박스가 없다면 사용자가 직접 키보드로 "경남 진양군", "강원도 철원군"과 같이 찾고자 하는 지명을 입력해 주어야겠지만 리스트 박스를 사용하면 목록중에서 원하는 항목을 선택하기만 하면 되므로 훨씬 더 사용하기 쉬울 것이다.

폼에 다음과 같이 두 개의 리스트 박스와 네 개의 레이블을 배치하고 레이블을 그림과 같이 수정한다. 레이블의 캡션을 각각 '도명', '군명', '찾으시는 DDD번호는'으로 설정하였다.

제일 아래의 레이블은 ResultStr로 Name 속성을 설정하며 Caption은 모두 지운다. 이 레이블은 찾은 DDD 번호를 출력해 주기 위해 배치하였으며 캡션이 없으므로 디자인시에는 보이지 않는다. 리스트 박스는 모두 디폴트 속성을 사용하되 Name만 각각 DoList, KunList로 변경한다. 좌측에 있는 리스트 박스(DoList)에는 도를 출력할 것이고 우측(KunList)에는 군을 출력할 것이되 단 편의상 특별시, 광역시는 모두 지리적으로 속한 도에 포함시키도록 한다. 즉 서울시는 경기도에 포함시키고 광주시는 전라남도에 포함시킨다. 도명은 9개밖에 없고 실행중에 변하지도 않으므로 디자인시에 미리 입력시켜 놓아도 상관이 없다. DoList의 Items 속성을 더블클릭하여 문자열 리스트 편집기를 불러낸 후 도명을 입력해 둔다.




좌측의 DoList에서 선택된 도에 따라 우측의 KunList에 군명을 출력할 것이다. 그러기 위해서는 어떤 도에 어떤 군이 있는가와 각 군의 DDD 번호는 무엇인가를 입력해야 하며 이 정보는 실행중에 교체되므로 리스트 박스의 Items 속성에 미리 입력시켜 둘 수는 없다. 그래서 군 하나의 정보를 가지는 TDDDNum 레코드를 선언하고 이 레코드를 배열 요소로 가지는 DDDNum 배열을 타입 상수로 선언하여 초기값에 군명과 DDD 번호를 입력해 둔다.

```
type
  TDDDNum=record
    j:string[10];    {군의 이름}
    p:integer;       {DDD 번호}
    end;

const
  DDDNum:array [0..8,0..47] of TDDDNum=(
```

TDDDNum 하나에 특정 군의 정보가 기억되며 이 레코드가 [8,47]의 크기로 배열을 이룬다. 배열의 1차 크기가 8인 이유는 도가 8개이기 때문이다. 2차 크

기가 47인 이유는 군이 제일 많은 경기도에 47개의 군이 있기 때문이다. 이렇게 자료를 구성해 놓으면 x도의 y군에 관한 정보를 다음과 같이 쉽게 구할 수가 있다.

```
DDDNum[x,y].j     {x 도 y 군의 이름}
DDDNum[x,y].p     {x 도 y 군의 DDD 번호}
```

타입 상수에 군 정보를 초기화시켜 주는 부분은 다음과 같다. 리스트가 무척이나 길어서 일부만 보인다.

```
const
 DDDNum:array [0..8,0..47] of TDDDNum=(
  {강원}
  ((j:'강릉';p:0391),(j:'고성';p:0392),(j:'근덕';p:0397),(j:'대화';p:0374),
  (j:'도계';p:0395),(j:'동송';p:0353),(j:'동해';p:0394),(j:'명주';p:0391),
  (j:'북평';p:0394),(j:'삼척';p:0397),(j:'상동';p:0373),(j:'설악';p:0392),
  (j:'속초';p:0392),(j:'양구';p:0364),(j:'양양';p:0396),(j:'영월';p:0373),
  (j:'원주';p:0371),(j:'인제';p:0365),(j:'정선';p:0398),(j:'주문진';p:0391),
  (j:'철원';p:0353),(j:'춘성';p:0361),(j:'춘천';p:0361),(j:'태백';p:0395),
  (j:'평창';p:0374),(j:'홍천';p:0366),(j:'화광';p:0395),(j:'화천';p:0363),
  (j:'횡계';p:0373),(j:'횡성';p:0372),(j:'';p:0),(j:'';p:0),
  (j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0),
  (j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0),
  (j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0),
  (j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0)),

  {경기}
  ((j:'가평';p:0356),(j:'강화';p:0349),(j:'고양';p:0344),(j:'과천';p:02),
  (j:'광명';p:02),(j:'광주';p:0347),(j:'구리';p:0346),(j:'군포';p:0343),
  (j:'김포';p:0341),(j:'남양';p:0339),(j:'남양주';p:0346),(j:'동두천';p:0351),
  (j:'문산';p:0348),(j:'미금';p:0346),(j:'발안';p:0339),(j:'부천';p:032),
  (j:'송탄';p:0333),(j:'수원';p:0331),(j:'시흥';p:0345),(j:'성남';p:0342),
  (j:'안산';p:0345),(j:'안성';p:0334),(j:'안양';p:0343),(j:'안중';p:0333),
  (j:'양주';p:0351),(j:'양평';p:0338),(j:'여주';p:0337),(j:'연천';p:0355),
  (j:'오산';p:0339),(j:'옹진';p:032),(j:'용문';p:0338),(j:'용인';p:0335),
  (j:'원당';p:0344),(j:'의왕';p:0343),(j:'의정부';p:0351),(j:'이천';p:0336),
  (j:'일산';p:0344),(j:'장호원';p:0336),(j:'전곡';p:0355),(j:'청평';p:0356),
  (j:'통진';p:0341),(j:'평택';p:0333),(j:'파주';p:0348),(j:'포천';p:0357),
  (j:'하남';p:0347),(j:'화성';p:0339),(j:'서울';p:02),(j:'인천';p:032)),

  {경남}
```

```
((j:'거제';p:0558),(j:'거창';p:0598),(j:'고성';p:0556),(j:'고현';p:0558),
(j:'김해';p:0525),(j:'남지';p:0559),(j:'남해';p:0594),(j:'마산';p:0551),
(j:'밀양';p:0527),(j:'사천';p:0593),(j:'산청';p:0596),(j:'삼랑진';p:0527),
(j:'삼천포';p:0593),(j:'수산';p:0527),(j:'양산';p:0523),(j:'온산';p:0522),
(j:'울산';p:0522),(j:'울주';p:0522),(j:'의령';p:0555),(j:'의창';p:0551),
(j:'장승포';p:0558),(j:'지족';p:0594),(j:'진양';p:0591),(j:'진영';p:0525),
(j:'진주';p:0591),(j:'진해';p:0553),(j:'창녕';p:0559),(j:'충무';p:0557),
(j:'통영';p:0557),(j:'하동';p:0595),(j:'함안';p:0552),(j:'함양';p:0597),
(j:'합천';p:0599),(j:'';p:0),(j:'';p:0),(j:'';p:0),
(j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0),
(j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0),
(j:'';p:0),(j:'';p:0),(j:'';p:0),(j:'';p:0)),
```

보다시피 이 프로그램은 머리보다는 손이 더 고달픈 예제이다. 그래서 필자도 DDD 목록을 다 입력하지 못하고 몇 개 도만 입력하였다. 시간 많은 사람은 나머지도 직접 입력해 보기 바란다. 작성된 코드는 단 두 개밖에 없다. DoListClick은 도명이 선택될 때 선택된 도에 속한 군 목록을 KunList에 뿌려주는 역할을 한다.

```
procedure TForm1.DoListClick(Sender: TObject);
var
  idx,i:Integer;
begin
  idx:=DoList.ItemIndex;
  KunList.Items.Clear;
  for i:=0 to 47 do
    if DDDNum[idx,i].j◇'' then
      KunList.Items.Add(DDDNum[idx,i].j);
end;
```

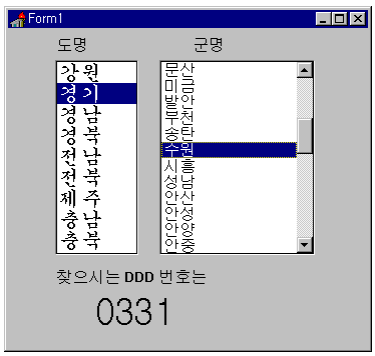
일단 Clear 메소드로 KunList를 깨끗하게 지운다. DoList에 선택된 도의 번호는 DoList의 ItemIndex 속성으로 구할 수 있으며 도의 번호를 DDDNum의 1차 첨자로 사용하여 해당 도의 군을 모두 읽어낸다. 읽어내는 개수는 군 이름 중 ' ', 즉 공백으로 입력된 군 이름이 있을 때까지이다. 경기도가 47개의 군을 가지므로 배열의 2차 첨자가 47이며 나머지 도는 47개 이하이므로 47개를 전부 출력할 필요가 없다. 읽어낸 군 이름을 KunList에 추가할 때는 Add 메소드를 사용한다.

KunListClick은 군명이 선택될 때 선택한 군의 DDD 번호를 ResultStr 레이블에 출력해 준다. 간단한 문자열 조립 예이므로 별다른 설명을 할 필요는 없을 것이다.

```
procedure TForm1.KunListClick(Sender: TObject);
begin
  ResultStr.Caption:='0'+IntToStr(DDDNum[DoList.ItemIndex,KunList.ItemIndex].p);
end;
```

문자열 앞에 '0'을 붙여주는 이유는 문자열로 바꾼 정수의 선행 제로가 잘려 나가기 때문이다. 즉 DDD 번호가 0522라도 숫자로는 522로 기억되기 때문에 앞에 0을 강제로 붙여 주어야 한다. 프로그램 실행시의 모습은 다음과 같다.

**그림**
리스트 박스를 이용해 만든 전화 번호부




여기서는 입력의 편의를 위해 DDD 목록을 작성했지만 똑같은 방법으로 우편번호부를 만들 수도 있다. 우편번호는 시/구/동의 3차원적 구조를 가지므로 세 개의 리스트 박스가 필요하며 입력해야 할 데이터 양도 많을 것이다. 꼭 우편번호부를 만들려면 리스트 박스를 사용하는 것보다 데이터 베이스 프로그래밍을 하는 것이 더 효율적이다.

# 라. 콤보 박스

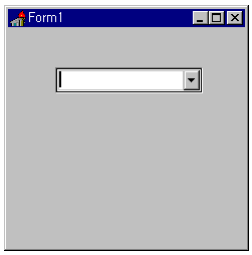
리스트 박스는 선택 사항을 직접 입력하지 않고 여러 개의 항목 중 하나를 마우스로 선택한다는 점에서 무척 편리한 컨트롤이지만 다음과 같은 몇 가지 단점이 있다.

첫째로 리스트 박스에 있는 아이템 중 하나를 선택할 수는 있지만 없는 아이템을 새로 입력할 수는 없다. 반드시 리스트 박스에 미리 입력되어 있는 항목 중의 하나를 선택해야 한다.

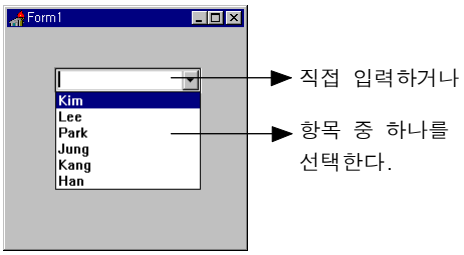
둘째 항상 아이템의 목록을 표시하고 있어야 하므로 크기가 커서 화면을 넓게

차지한다. 리스트 박스 몇 개만 사용하면 대화상자가 너무 커져서 보기에도 좋지 못하고 다른 컨트롤을 배치할 자리도 없어진다.

리스트 박스의 이런 두 가지 단점을 해결한 것이 콤보 박스이다. 콤보 박스는 에디트와 리스트 박스를 합쳐놓은 컴포넌트이다. 선택할 사항은 리스트 박스에서 선택하고 직접 입력해야 할 사항은 에디트에 입력할 수 있으며 리스트 박스는 필요할 경우에만 열어서 사용하고 평소에는 닫아 둠으로써 화면 면적을 절약할 수 있다.




평소에는 자리를 좁게 차지한다.

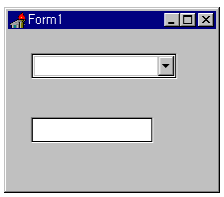



필요할 경우 리스트 박스를 열어서 사용한다.

콤보 박스는 리스트 박스와 에디트 박스의 결합이므로 리스트 박스의 Items 속성과 에디트의 Text 속성을 모두 가진다. 콤보 박스의 밑에 있는 리스트 박스에서 선택한 내용은 윗쪽의 에디트 박스에 나타나고 Text 속성에 대입되므로 콤보 박스에서 선택한 사항은 Text 속성을 읽으면 된다. 콤보 박스 아래쪽의 리스트 박스에서 아이템을 읽거나 삽입, 삭제하는 방법은 리스트 박스의 경우와 같다.


7jang
combo

예제를 작성해 보자. 여기서는 에디트 박스와 콤보 박스 컴포넌트만을 사용한다. 콤보 박스에서 선택한 사항을 에디트에 출력할 것이다.

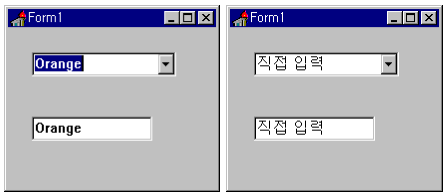



다른 속성은 그대로 두고 콤보 박스의 Items 속성에 Apple, Orange, Banana 세 개의 문자열만을 입력해 둔다. 콤보 박스에 뭔가 변화가 있을 경우에만 에디트에 변경을 가하면 되므로 콤보 박스의 OnChange 이벤트 핸들러를 작

성한다.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  Edit1.Text:=ComboBox1.Text;
end;
```

실행해 보면 콤보 박스에서 세 개의 항목 중에 하나를 선택하거나 콤보 박스의 에디트에서 직접 문자열을 입력하여도 아래쪽의 에디트 컴포넌트로 출력된다. 즉 목록에서 Apple, Orange, Banana 중 하나를 선택할 수 있으며 콤보 박스에는 없는 Meron, Grape 등의 문자열도 에디트에 직접 입력해 넣을 수 있다.




콤보 박스의 Style 속성을 변경하면 약간 형태가 다른 콤보 박스를 만들 수도 있다.

**표** 콤보 박스의 스타일

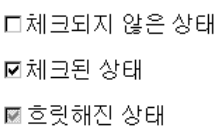
| Style | 형태 |
|---|---|
| csDropDown | 에디트 박스와 리스트 박스를 모두 가지는 전형적인 콤보 박스이며 이 값이 디폴트이다. 목록에서 고를 수도 있고 에디트 박스에 직접 입력할 수도 있다. |
| csSimple | 리스트 박스없이 에디트만 가진다. 일반적으로 잘 사용되지 않으며 데이터 베이스 프로그램에서 가끔 사용된다. |
| csDropDownList | 에디트없이 리스트 박스만 가진다. 그래서 리스트 박스에 있는 항목 중 하나를 선택할 수는 있지만 직접 입력할 수는 없다. 콤보 박스라기보다는 일종의 변형된 리스트 박스이다. |

디폴트는 csDropDown으로 되어 있는데 Style을 csDropDownList로 변경하면 에디트에서 입력할 수는 없으며 목록에서만 선택할 수 있는 콤보 박스가 만들어진다. 직접 속성을 변경해 보고 뭐가 다른지 살펴 보기 바란다. 이 외에도 콤보 박스에 그래픽 항목을 입력할 수 있는 OwnerDraw형 콤보 박스도 있다.

# 7-2 체크 박스

## 가. 체크 박스

체크 박스는 사용자로부터 아주 간단한 옵션을 입력받을 때 사용하는 컨트롤이며 질문에 대한 답이 예 또는 아니오 두 가지(또는 세 가지) 밖에 없을 경우에 적당하다. 조그만 사각 박스가 있고 그 옵션이 의미하는 바를 나타내는 설명이 우측에 위치하는 간단한 모양을 가지며 예일 경우 박스 안에 V 표시가 나타나고 아니오일 경우 박스 안에 V 표시가 나타나지 않는다.




체크 박스의 선택 상태는 State 속성에 나타나며 실행중에 이 속성을 읽어 체크 박스의 상태를 조사한다. State 속성은 다음 세 가지 값 중 하나의 값을 가진다.

**표**

체크 박스의 세 가지 상태

| 속성 | 의미 |
|---|---|
| cbUnchecked | 체크되어 있지 않다. |
| cbChecked | 체크되어 있다. |
| cbGrayed | 흐리게 표시되어 있다. |

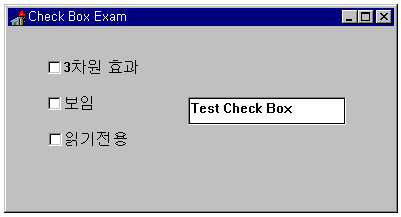
실행중에 사용자가 체크 박스를 클릭하면 OnClick 이벤트가 발생한다. 체크 박스가 바뀔 때마다 특별한 처리를 해야 한다면 OnClick 이벤트를 작성하고 이벤트 핸들러에서 처리를 해준다. 예를 들어 풍선 도움말(Hint) 표시 여부 체크 박스를 클릭할 때마다 도움말을 표시하거나 지우는 코드를 작성하고 싶다면 OnClick 이벤트 핸들러에 이 코드를 위치시킨다.

체크 박스가 선택될 때마다 일일이 코드를 실행시키는 것이 귀찮거나 그럴 필요가 없다면 OnClick 이벤트 핸들러를 작성할 필요없이 일정한 때에 한꺼번에 처리를 하도록 한다. 예를 들어 옵션 설정 대화상자 내에 있는 체크 박스의 경우

체크 박스를 클릭할 때는 동작을 하지 않고 대화상자가 닫히는 시점(FormClose 정도가 적당하다)에서 한꺼번에 처리를 하도록 한다.

7jang
check

체크 박스를 사용한 간단한 예제를 만들어 보자. 체크 박스 세 개와 에디트 박스 하나를 폼에 배치한다. 체크 박스에서 선택한 옵션에 따라 에디트 박스의 속성을 변경하도록 해보자.




컴포넌트 배치 후 Caption 속성을 모두 다음과 같이 바꾼다. 체크 박스의 Caption 속성은 체크 박스의 오른쪽에 나타나는 문자열이며 옵션의 의미를 간단하게 나타낼 수 있도록 작성한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | Caption | Check Box Exam |
| 위 체크 박스 | Caption | 3차원 효과 |
| 가운데 체크 박스 | Caption | 보임 |
| 아래 체크 박스 | Caption | 읽기 전용 |
| 에디트 | Text | Test Check Box |

위쪽의 3차원 효과 체크 박스를 더블클릭하여 다음과 같은 이벤트 핸들러를 작성한다.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.State=cbChecked then
    Edit1.Ctl3D:=True
  else
    Edit1.Ctl3D:=False;
end;
```

이 체크 박스의 상태인 State 속성을 조사하여 체크되어 있으면 에디트 컴포

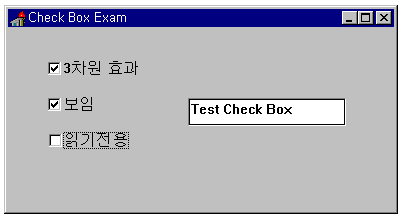
넌트의 Ctl3d 속성을 True로 만들고 체크되어 있지 않으면 False로 만든다. 즉 위쪽 체크 박스는 에디트 박스에 입체감을 줄 것인가 아닌가를 선택한다. 가운데와 아래쪽의 체크 박스도 동일한 구조를 가지며 각각 에디트의 Visible 속성과 ReadOnly 속성을 조작한다.

```
procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  if CheckBox2.State=cbChecked then
    Edit1.Visible:=True
  else
    Edit1.Visible:=False;
end;

procedure TForm1.CheckBox3Click(Sender: TObject);
begin
  if CheckBox3.State=cbChecked then
    Edit1.ReadOnly:=True
  else
    Edit1.ReadOnly:=False;
end;
```

체크 박스의 State 속성은 값을 읽어 체크 박스의 상태를 조사하기도 하지만 State 속성에 값을 대입하여 실행중에 체크 박스의 상태를 강제로 변경할 수도 있다. 프로그램이 처음 실행될 때의 체크 박스 상태를 초기화시키기 위해 폼의 OnCreate 이벤트를 사용한다. 폼의 빈 곳을 더블클릭하여 다음 코드를 입력하도록 하자.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CheckBox1.State:=cbChecked;
  CheckBox2.State:=cbChecked;
  CheckBox3.State:=cbUnChecked;
end;
```

프로그램 실행 중의 모습은 다음과 같다. 체크 박스를 클릭하여 체크 상태를 바꿀 때마다 에디트 박스의 속성이 변경될 것이다.

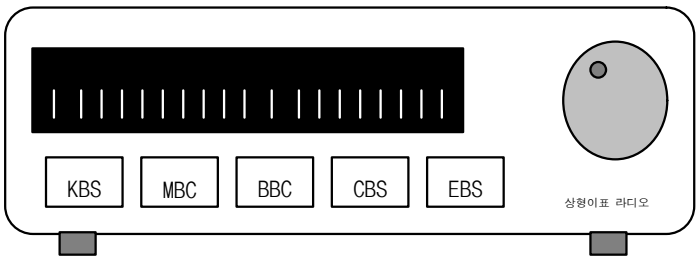



체크 박스는 보편적으로 두 가지 상태 중 한 가지를 결정하기 위해 사용하는데 어떤 경우에는 세 가지 상태를 가질 수도 있다. 체크된 상태, 체크되지 않은 상태 외에 추가로 가질 수 있는 상태는 흐리게 된 상태(Grayed)이다. 이는 옵션이 선택되었는지 선택되지 않았는지를 결정할 수 없는 상태(Intermediate)를 의미한다. Grayed 상태를 가질 수 있도록 하려면 체크 박스의 AllowGrayed 속성을 True 로 변경해 주면 된다.

# 나. 라디오 버튼

라디오 버튼(Radio Button)은 상호 배타적인 옵션을 선택할 때 사용한다. 여러 개의 선택 사항이 있으며 그 중 꼭 하나만 선택될 수 있고 동시에 두 개가 선택될 수 없다는 특징이 있다. 버튼으로 채널을 선택하는 라디오에 이런 형태의 버튼이 사용되기 때문에 붙여진 이름인데 MBC면 MBC, KBS면 KBS 꼭 하나만 선택할 수 있으며 둘을 동시에 선택할 수는 없다. 그래서 라디오 버튼은 다른 버튼을 누르면 앞에 선택되어 있던 버튼은 다시 튀어 올라와 버리도록 되어 있다.

**그림**
수신 방송국을 선택하는 라디오 버튼

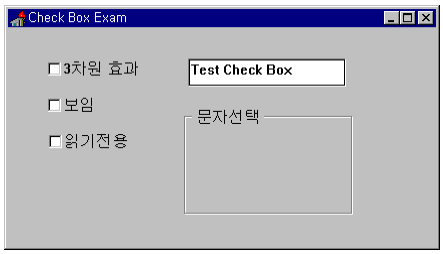



앞에서 만들었던 체크 박스 예제에 라디오 버튼도 추가로 달아 보자. 에디트의 CharCase 속성을 라디오 버튼으로 실행중에 변경시키고자 한다. CharCase 속성은 에디트에 입력될 문자의 대, 소문자 구성을 지정하는데 소문자 모드, 대문자 모드, 혼합 모드 세 가지가 있으며 세 가지 중의 한 가지만 선택될 수 있다.

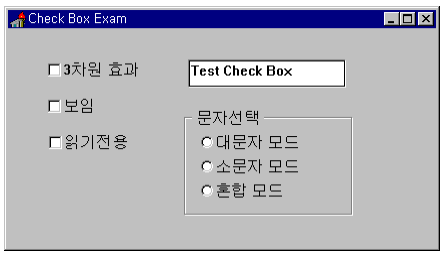
라디오 버튼은 속해 있는 그룹 내에서 하나만 선택되므로 배타적으로 선택되도록 하기 위해 먼저 그룹 박스 컴포넌트를 배치하고 그 안에 라디오 버튼을 배

7jang
radio1

치해야 한다. 그룹 박스 컴포넌트는 Standard 페이지의 끝에서 세 번째에 있다. 폼의 크기를 조금 늘리고 그룹 박스 컴포넌트를 폼의 적당한 위치에 배치해 두고 캡션 속성을 "문자 선택"으로 바꾸어 둔다.




라디오 버튼 세 개를 그룹 박스 안에 배치하고 캡션 속성만 다음과 같이 변경하자.




세 개의 라디오 버튼이 하나의 그룹에 속해 있으므로 세 버튼 중 하나만 배타적으로 선택된다. 세 개의 버튼에 다음과 같이 이벤트 핸들러를 작성한다. 선택된 라디오 버튼에 따라 에디트의 CharCase 속성을 변경한다.

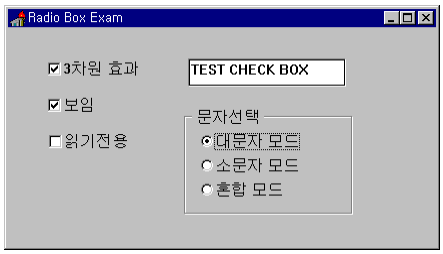
```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
Edit1.CharCase:=ecUpperCase;
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
Edit1.CharCase:=ecLowerCase;
end;

procedure TForm1.RadioButton3Click(Sender: TObject);
begin
Edit1.CharCase:=ecNormal;
```

```
end;
```

그리고 혼합 모드 라디오 버튼이 디폴트로 선택되도록 하기 위해 FormCreate 이벤트에 다음 코드를 추가하고 폼의 캡션을 Radio Box Exam으로 바꾼다.

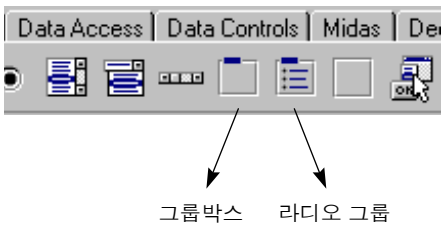
```
RadioButton3.Checked:=True;
```

버튼이 선택될 때 Edit1의 CharCase 속성을 바꾸도록 하였다. 실행해 보면 라디오 버튼을 변경함에 따라 에디트의 문자들이 바뀔 것이며 라디오 버튼 하나를 선택하면 이미 선택되어 있는 옵션은 선택 해제될 것이다. 실행중의 모습은 다음과 같다.

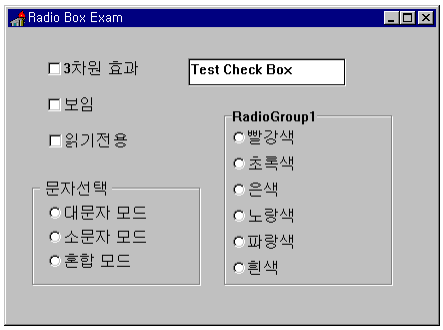



7jang
radio2

라디오 버튼을 만드는 좀 더 쉬운 방법은 라디오 그룹 컴포넌트를 사용하는 것이다. 라디오 그룹은 라디오 버튼을 일일이 배치하지 않고 Items 속성에 문자열 리스트로 라디오 버튼을 간단하게 만들 수 있다. 앞의 예제에 라디오 버튼을 하나 더 만들어 보되 이번에는 에디트의 색상을 선택하도록 해보자. 색상도 여러 개 중에 하나만 선택해야 하므로 라디오 버튼을 사용하는 것이 적합하다.




폼의 크기를 조금 늘리고 라디오 그룹 컴포넌트를 폼에 배치하고 Items 속성을 더블클릭하여 문자열 리스트 편집기를 연다. 라디오 버튼에 사용될 캡션들을 문자열로 입력해 주면 이 문자열들이 라디오 그룹 컴포넌트 내에 라디오 버튼으로 나타난다. 다음과 같이 색상의 이름을 라디오 그룹의 Items 속성에 대입한다.




라디오 그룹은 통째로 하나의 컴포넌트이므로 이벤트도 개별 라디오 버튼이 받는 것이 아니라 라디오 그룹 자체가 받는다. 라디오 그룹에서 어떤 라디오 버튼이 선택되었는가는 ItemIndex 속성을 조사하여 알 수 있다. 라디오 그룹의 OnClick 이벤트를 다음과 같이 작성한다.

```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
case RadioGroup1.ItemIndex of
 0:Edit1.Color:=clRed;
 1:Edit1.Color:=clGreen;
 2:Edit1.Color:=clSilver;
 3:Edit1.Color:=clYellow;
 4:Edit1.Color:=clBlue;
 5:Edit1.Color:=clWhite;
end;
end;
```

라디오 그룹에서 선택된 라디오 버튼의 번호에 따라 에디트의 색상을 변경시키는 코드이다. 폼이 만들어질 때는 은색이 디폴트로 선택되도록 FormCreate 이벤트에 다음 코드를 추가한다.

```
RadioGroup1.ItemIndex:=2;
```

라디오 버튼을 만드는 방법에는 위에서 본 바와 같이 두 가지 방법이 있다. 라디오 버튼만 만들려면 라디오 그룹 컴포넌트를 사용하는 것이 여러모로 편리하다. 라디오 버튼을 일일이 만들어 주지 않아도 될 뿐만 아니라 버튼간의 간격을 자동으로 맞추어 주며 코드를 작성할 때도 통합하여 한번만 작성해 주므로 다음에 수정하기에도 편리하다. 또한 라디오 그룹은 하나의 컴포넌트로 여러 개의 라디오 버튼을 보여줄 수 있으므로 시스템 자원 절약에도 유리하다.

# 다. 체크 리스트

체크 리스트 박스는 체크 박스와 리스트 박스를 합쳐 놓은 컨트롤이다. 형태상으로 볼 때는 리스트 박스에 더 가깝지만 용도상으로 볼 때는 체크 박스의 집합이라고 할 수 있다. Additional 페이지의 10 번째에 위치하고 있다.

보통의 리스트 박스와 유사하되 리스트 박스안의 항목이 문자열이 아니라 체크 박스라는 점이 다르다. 복수 개의 진위 옵션을 입력받아야 할 경우 체크 박스를 한곳에 모을 수 있도록 해준다. 체크 리스트 박스를 폼에 배치한 후 Items 속성에 체크 박스의 캡션들만 입력해 주면 리스트 박스안에 체크 박스가 생성된다. 리스트 박스(TListBox)의 변형이므로 리스트 박스가 가지는 모든 속성과 이벤트, 메소드를 가짐은 물론 세 개의 속성과 한 개의 이벤트를 추가로 가진다.

✍ **AllowGrayed**

체크 리스트 박스안의 체크 박스는 체크/안체크(Checked/Unchecked) 두 가지 상태를 가지는데 이 속성을 True 로 설정해 두면 이 외에 Grayed 라는 제 3 의 상태를 추가로 가진다. 참/거짓의 상태 외에 결정되지 않은 상태라든가 결정할 필요가 없는 상태등이 필요하다면 이 속성을 True 로 설정하도록 한다. 단 이 속성은 개별 체크 박스에 대해 설정할 수 없다.

✍ **Checked**

각 체크 박스의 체크 상태를 가지는 진위형 배열이며 이 배열값을 읽음으로써 체크 리스트 박스내의 체크 박스 상태를 알 수 있다. 즉 n 번째 체크 박스의 체크 상태를 알고 싶으면 Checked[n]값을 읽어보면 된다. 이 배열값이 True 이면 체크된 것이고 그렇지 않으면 안체크된 것이다. 배열이므로 베이스는 당연히 0 이다. 즉 첫 번째 체크 박스의 상태는 Checked[0]를 읽어야지 Checked[1]을 읽는 것이 아니다.

✍ **State**

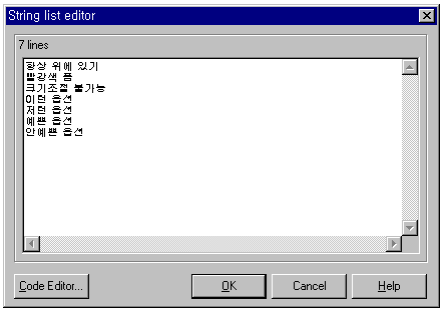
Checked 속성과 유사하되 체크/안체크뿐만 아니라 Grayed 상태까지 조사할 수 있다. 이 배열값을 읽은 결과는 Checked, Unchecked, Grayed 중 한 값이 된다.

OnClickCheck 이벤트는 리스트 박스에서 체크 박스를 클릭할 때 발생한다. 단 체크 박스 자체를 클릭하여 체크 상태를 변경할 때만 발생하며 체크 박스 문

자열을 클릭할 때는 이 이벤트가 발생하지 않는다. 이 이벤트의 핸들러에서 체크 박스의 상태를 조사하여 옵션 변경에 따른 특별한 동작을 해주면 된다.

7jang
ChkList

체크 리스트 박스를 사용하는 간단한 예제를 만들어 보자. 새 프로젝트를 시작하고 폼에 체크 리스트 박스를 하나 배치한 후 Items 속성에 다음과 같이 문자열을 입력한다.




Items 속성에 입력된 각 문자열은 하나의 체크 박스가 되어 리스트 박스에 나타나게 된다. 체크 리스트 박스의 OnClickCheck 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.CheckListBox1ClickCheck(Sender: TObject);
begin
   if CheckListBox1.Checked[0] = TRUE then
     Form1.FormStyle := fsStayOnTop
   else
      Form1.FormStyle := fsNormal;

   if CheckListBox1.Checked[2] = TRUE then
      Form1.BorderStyle := bsSingle
   else
      Form1.BorderStyle := bsSizeable;

   if CheckListBox1.Checked[1] = TRUE then
      Form1.Color := clRed
   else
      Form1.Color := clBtnFace;

end;
```

이벤트 핸들러로 어떤 버튼이 체크되었는지에 대한 정보가 전달되지는 않으므로 일일이 체크 박스의 상태를 조사해 보고 옵션 변경에 따른 처리를 해주었다. 실행중의 모습은 다음과 같다.

# 7-3 스크롤 바

스크롤 바는 긴 막대 모양을 가지고 있으며 윈도우가 전체 작업 영역을 표시하지 못할 때 작업 영역을 이동시키거나 값을 대충 대충 신속하게 조정할 필요가 있을 때 사용한다. 스크롤 바의 어느 부분을 클릭하는가에 따라 값의 변화 정도가 다르며 스크롤 바 중앙의 사각 썸(thumb)을 드래그하여 값을 조절할 수도 있다. Kind 속성으로 모양을 결정하며 수평 스크롤 바와 수직 스크롤 바가 있다.

Kind=sbHorizontal Kind=sbVertical

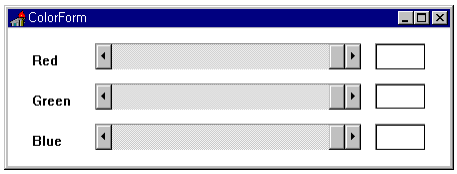
스크롤 바의 주요 속성은 다음과 같다. 스크롤 바가 가지는 값의 범위와 증감 정도를 지정하는 속성이다.

| 속성 | 의미 |
|---|---|
| Min | 스크롤 바가 가질 수 있는 최소값을 지정한다. 디폴트는 0이다. |
| Max | 스크롤 바가 가질 수 있는 최대값을 지정한다. 디폴트는 100이다. |
| SmallChange | 스크롤 바 양끝의 화살표를 클릭할 때 증감될 양을 지정한다. 디폴트는 1이며 섬세하게 값을 조정할 수 있도록 하는 것이 좋다. |
| LargeChange | 스크롤 바의 몸통 부분을 클릭할 때 증감될 양을 지정한다. 디폴트 값은 1이며 신속하게 이동할 수 있도록 큰 값을 주는 것이 좋다. |
| Position | 스크롤 바의 썸의 위치를 가지며 이 값을 읽음으로써 스크롤 바가 가진 값을 알 수 있다. 또한 이 값을 변경함으로써 썸을 이동시킬 수도 있다. |

디자인시에 이 속성들을 설정해 주거나 아니면 실행중에 SetParams 메소드

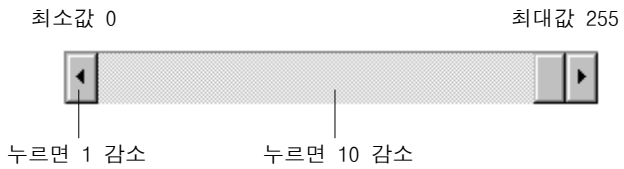
7jang
scrbar

를 사용하여 Min, Max, Position을 한꺼번에 변경할 수도 있다. 스크롤 바를 응용하여 윈도우의 색상을 변경시키는 예제를 작성해 보자. 스크롤 바, 레이블, 에디트 컴포넌트를 각각 세 개씩 폼에 배치하고 속성을 설정한다.




| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 레이블 | Caption | Red, Green, Blue 로 각각 설정 |
| 위쪽 스크롤 바 | Name | Redbar |
| 가운데 스크롤 바 | Name | Greenbar |
| 아래쪽 스크롤 바 | Name | Bluebar |
| 에디트 전부 | Text | 모두 지움 |
| | ReadOnly | True |
| 위쪽 에디트 | Name | RedEdit |
| 가운데 에디트 | Name | GreenEdit |
| 아래쪽 에디트 | Name | BlueEdit |
| 스크롤 바 전부 | Min | 0 |
| | Max | 255 |
| | LargeChange | 10 |
| | SmallChange | 1 |
| 폼 | Name,Caption | ColorForm |

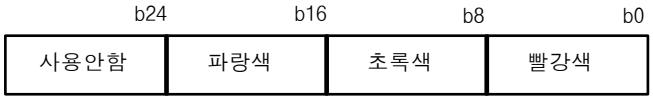
세 개의 스크롤 바에서 입력된 R, G, B값을 읽어서 세 가지 색을 혼합한 색상으로 폼의 색상을 변경하고자 한다. 각 색상 요소값이 최소 0~최대 255까지 가능하므로 스크롤 바의 Min, Max 속성이 0, 255이며 스크롤 바의 허리를 클릭할 때 10씩 증감하고 화살표를 클릭할 때 1씩 증감하도록 하였다.




세 개의 스크롤 바 중 하나라도 변경되면 폼의 색상이 바뀌어야 하므로 이벤트 핸들러 내에서 일일이 처리를 하는 것보다는 프로시저를 하나 만들어 두고 이벤트 핸들러에서 불러서 쓰는 것이 더 효율적이다. ChangeColor라는 이름으로 프로시저를 만든다. 이 함수의 위치는 소스의 implementation 바로 아래 부분이다.

```
procedure changecolor;
begin
 ColorForm.color:=ColorForm.Redbar.Position+
          ColorForm.Greenbar.Position*$100+
          ColorForm.Bluebar.Position*$10000;
end;
```

이 함수는 세 개의 스크롤 바가 가지는 색상값을 조립하여 하나의 색상값을 만들어 폼의 색상에 대입한다. 색상값은 32비트의 길이를 가지며 하위 8비트에서부터 빨강색, 초록색, 파랑색의 농도를 가진다.




초록색값이 상위 8비트로 이동하도록 하기 위해 256(16진수로 $100)을 곱해 주어야 하며 파랑색은 65536(16진수로 $10000))을 곱해주어야 한다. 빨강색값은 제일 하위 바이트이므로 곱하는 값 없이 더해주기만 한다. 이렇게 조립한 색상값을 폼의 Color 속성에 대입하여 폼의 색상을 변경한다.

각 스크롤 바의 OnChange 이벤트 핸들러를 입력한다.

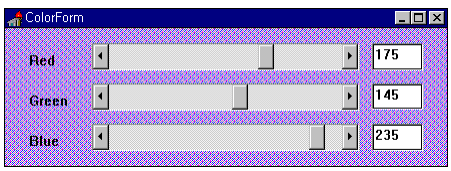
```
procedure TColorForm.RedbarChange(Sender: TObject);
begin
 RedEdit.Text:=IntToStr(Redbar.Position);
 ChangeColor;
end;

procedure TColorForm.GreenbarChange(Sender: TObject);
```

```
begin
  GreenEdit.Text:=IntToStr(Greenbar.Position);
  ChangeColor;
end;

procedure TColorForm.BluebarChange(Sender: TObject);
begin
  BlueEdit.Text:=IntToStr(Bluebar.Position);
  ChangeColor;
end;
```

모두 같은 구조를 가지며 스크롤 바의 Position 속성이 어떤 값을 가지는가를 오른쪽의 에디트에 출력한 후 ChangeColor 프로시저를 호출하여 스크롤 바의 변화가 즉각적으로 폼에 나타나도록 해준다. 마지막으로 FormCreate 이벤트 핸들러에서 스크롤 바와 에디트의 초기값을 설정해 줌으로써 예제를 완성시킨다.

```
procedure TColorForm.FormCreate(Sender: TObject);
begin
  Redbar.Position:=255;
  Greenbar.Position:=255;
  Bluebar.Position:=255;
  RedEdit.Text:='255';
  GreenEdit.Text:='255';
  BlueEdit.Text:='255';
  ChangeColor;
end;
```

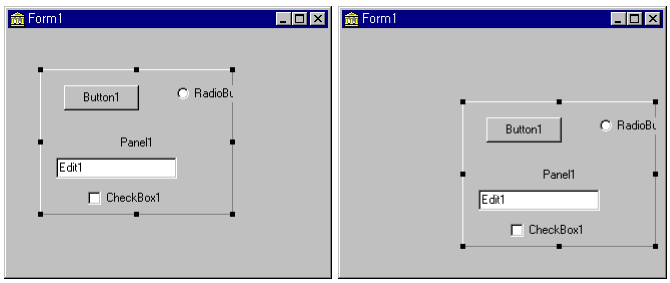
직접 실행해 보면 스크롤 바를 클릭하거나 드래그함으로써 폼의 색상이 다양하게 변하는 것을 알 수 있다.



# 7-4 패널

## 가. 패널

패널 컴포넌트는 다른 컴포넌트가 놓이는 자리를 제공해 주며 패널 위에 다른 컴포넌트를 배치할 수 있다. 패널을 폼에 배치하면 널찍한 양각 사각형이 만들어지며 중앙에 Panel1이라는 캡션만 표시되어 있는데 이 위에 컴포넌트를 가져다 놓으면 된다. 패널 위에 놓여진 컴포넌트들은 디자인 시에 하나의 단위처럼 동작하여 패널의 위치를 변경하면 패널 위의 모든 컴포넌트들도 따라서 이동한다. 이런 식으로 다른 컴포넌트를 포함할 수 있는 컴포넌트를 컨테이너 컴포넌트(Container Component)라고 한다. 컨테이너 컴포넌트의 가장 대표적인 예는 폼이며 패널, 그룹 박스 등 몇 개의 컴포넌트들도 컨테이너로 동작한다.




패널 위에는 주로 버튼, 비트맵 버튼, 스피드 버튼 등이 놓여져 폼 상단의 툴바를 구성하거나 레이블이 놓여져 폼 하단의 상태란을 만들 때 사용된다. 물론 그 외의 다른 용도로도 얼마든지 사용할 수 있다.

패널 중앙의 문자열은 물론 Caption 속성으로 설정한다. 그런데 패널은 홀로 사용되는 경우가 드물며 보통 다른 컴포넌트를 담기 때문에 패널의 Caption은 별로 의미가 없으며 삭제해 버리는 것이 일반적이다. 패널의 모양은 Bevel로 시작되는 다섯 개의 속성으로 설정한다.

| 속성 | 의미 |
|---|---|
| BevelInner | 안쪽 베벨의 모양을 지정한다. bvNone은 베벨을 만들지 않으며 BevelLowered는 음각, BevelRaised는 양각의 표현한다. |

| | |
|---|---|
| BevelOuter | 바깥쪽 베벨의 모양을 지정한다. 속성값은 BevelInner속성과 같다. |
| BevelWidth | 안쪽, 바깥쪽 베벨의 두께를 지정한다. 음각, 양각을 어느 정도로 줄 것인가를 정한다. |
| BevelStyle | 패널 바깥쪽에 검정색 테두리를 그릴 것인가 아닌가를 지정한다. bsNone이면 테두리를 그리지 않고 bsSingle이면 테두리를 그린다. |
| BorderWidth | 안쪽 베벨과 바깥쪽 베벨의 간격을 설정한다. |

다음에 여러 가지 패널 예를 보인다. 직접 속성값을 변경해 보면 패널의 모양을 어떻게 디자인하는지 터득할 수 있을 것이다.

**그림**
속성 변경에 의한 패널 모양

| 패널 | 속성 설정 |
|---|---|
|  | 디폴트 패널 |
|  | BevelOuter를 bsRaised로 설정한 경우 |
|  | BevelInner를 bsLowered로 설정한 경우 |
|  | BorderWidth를 5로 설정한 경우 |
|  | BevelInner를 bsRaised로 설정한 경우 |
|  | BevelWidth를 3으로 설정하고 BorderStyle을 bsSingle로 설정한 경우 |

패널도 물론 이벤트를 가진다. 그러나 패널은 직접 사용되기보다는 다른 컴포넌트가 놓일 자리를 제공해 주는 용도로 많이 사용되므로 패널의 이벤트를 사용할 일은 별로 없다. 패널을 사용하는 예는 차후에 보이기로 한다.

# 나. Align 속성

패널이 가지는 속성중 가장 흥미로운 속성은 Align 속성이다. 이 속성은 패널이 항상 폼의 일정한 위치에 있도록 한다. Align 속성의 디폴트값은 alNone이며 정렬을 전혀 하지 않기 때문에 디자인시에 지정해 준 위치에 정해진 크기대로 항

상 그대로 있다. Align 속성을 바꾸면 패널의 위치는 다음과 같이 변한다.

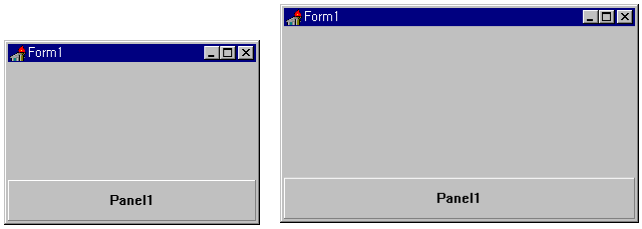
**표**
Align 속성

| 속성값 | 정렬 |
|---|---|
| alNone | 정렬되지 않으며 디자인시에 정해준 위치와 크기를 실행중에도 그대로 사용한다. |
| alTop | 폼의 상단에 정렬되며 패널의 가로 폭은 폼의 가로 폭과 같아진다. 패널의 높이는 디자인시에 결정된 값을 사용한다. |
| alBottom | 폼의 하단에 정렬되며 패널의 가로 폭은 폼의 가로 폭과 같아진다. 패널의 높이는 디자인시에 결정된 값을 사용한다. |
| alLeft | 폼의 좌측변에 정렬되며 패널의 높이는 폼의 높이와 같아진다. 패널의 폭은 디자인시에 결정된 값을 사용한다. |
| alRight | 폼의 우측변에 정렬되며 패널의 높이는 폼의 높이와 같아진다. 패널의 폭은 디자인시에 결정된 값을 사용한다. |
| alClient | 패널을 폼의 작업 영역과 완전히 일치시킨다. 패널이 전체 폼을 가득 채우며 폼의 크기가 조정되면 패널의 크기도 따라서 조정된다. |

패널의 Align 속성이 alBottom인 경우 패널은 항상 폼의 하단부에 위치하며 폼의 크기가 변하면 패널의 위치와 크기도 따라서 변한다. 패널의 이런 속성을 이용하면 항상 폼의 하단에 있는 상태란을 만들 수 있다. 윈도우 크기가 어떻게 변하든 상태란은 항상 윈도우의 제일 아래쪽에 고정된다.

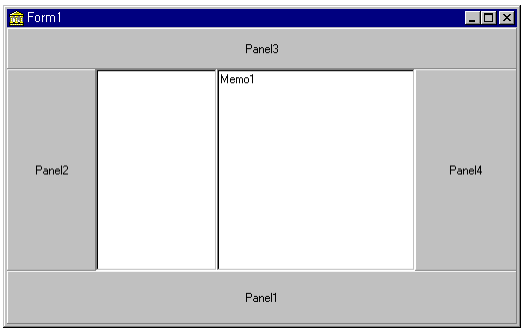
이 속성을 테스트해 보기 위해 새 프로젝트를 시작한 후 패널 컴포넌트를 하나 배치해 보자. 그리고 이 컴포넌트의 Align 속성을 alBottom으로 설정하면 패널이 폼의 하단부에 밀착될 것이다. 이 상태에서는 패널의 위치를 옮길 수는 없으며 높이만 조정 가능하다. 프로그램을 실행한 후 폼의 크기를 변경해 보면 폼의 크기에 상관없이 패널은 항상 폼의 아래쪽에 위치한다. 물론 오른쪽이나 위쪽으로 정렬했다면 해당 위치에 정렬될 것이다.

**그림**
폼의 바닥에 정렬된 패널

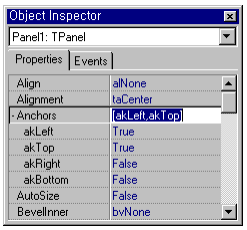



Align 속성은 패널뿐만 아니라 메모, 리스트 박스 등 비교적 면적이 넓은 컴

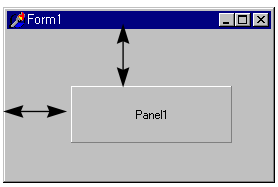
포넌트에 공통적으로 존재한다. 이 속성을 잘 사용하면 여러 가지 컴포넌트가 폼을 적절히 분할해서 사용할 수 있다. 다음 그림은 4개의 패널이 각각 상하좌우에 정렬되어 있고 리스트 박스가 왼쪽으로 그리고 메모가 나머지 영역을 모두 차지하도록 정렬된 것이다. 이렇게 정렬해 놓으면 실행중에 폼의 크기나 모양이 어떻게 바뀌더라도 항상 이 배치 상태를 유지하게 된다.

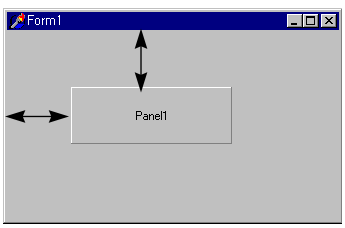



Align 속성과 함께 사용되는 속성으로 Anchors 속성이 있다. 이 속성은 다음과 같은 네 개의 세부 속성을 가지며 모두 진위형이다.



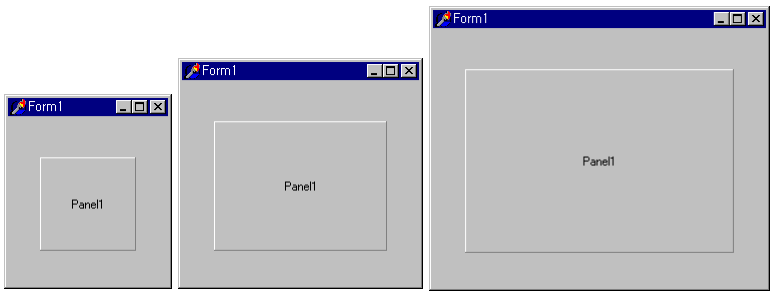

각 속성은 컴포넌트의 한쪽 변이 폼에 대해 일정한 위치에 고정되도록 한다. 예를 들어 akLeft 속성은 폼과 컴포넌트의 좌측변이 항상 일정한 거리를 유지하도록 한다. 디폴트로 akLeft, akTop이 선택되어 있기 때문에 컴포넌트는 폼의 좌측변에 대해 항상 일정한 거리를 유지한다. 새 폼에 패널 하나를 배치한 후 실행해 보자.



폼의 크기가 어떻게 변하든지 패널의 좌측 좌표와 위쪽 좌표는 변함없이 일정한 거리를 유지한다. 왜냐하면 Anchors.akLeft, Anchors.akTop이 True로 설정되어 있기 때문이다. 반면 akRight, akBottom은 False로 설정되어 있으므로 폼과 패널의 오른쪽 간격과 아래쪽 간격은 폼의 크기가 변경되면 같이 변경된다. 이 속성들까지 True로 바꾸어 주면 폼의 크기와 상관없이 항상 일정한 거리를 유지하게 된다. 프로젝트를 실행한 후 직접 확인해 보도록 하자. 폼의 크기를 늘리면 패널의 크기도 같이 늘어나게 될 것이다.

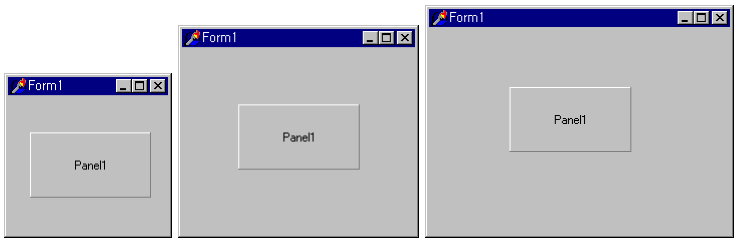
그림
Anchors 속성으로 컴포넌트 고정하기



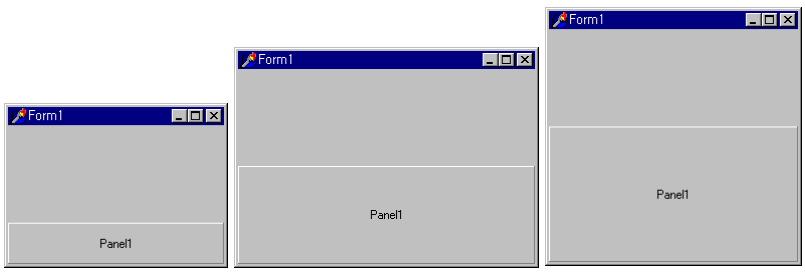

그럼 Anchors 속성과 Align 속성은 어떤 관계가 있을까. 이를 확인하려면 Anchors 세부 속성을 열어 놓은 채로 Align 속성을 변경해 보면 된다. 이를 표로 정리해 보면 다음과 같다.

| Align | akLeft | akTop | akRight | akBottom |
|---|---|---|---|---|
| alNone | True | True | False | False |
| alLeft | True | True | False | True |
| alTop | True | True | True | False |
| alRight | False | True | True | True |
| alBottom | True | False | True | True |
| alClient | True | True | True | True |

표를 자세히 들여다 보면 어렵지 않게 이해할 수 있을 것이다. Align 속성이 alLeft일 경우는 akRight만 False이기 때문에 오른쪽 변을 제외하고는 항상 일정한 거리(보통 0)를 유지할 것이고 그래서 좌측으로 정렬되는 것이다. 이 속성을 잘 이용하면 아주 재미있는 형태의 폼을 만들 수 있다. 패널을 폼의 정 중앙에 배치해 두고 Anchors 속성들을 모두 False로 변경하면 이 패널은 항상 폼의 중앙에 위치하게 된다.




또한 Align 속성을 alBottom으로 두고 Anchors 속성들을 모두 True로 변경하면 패널을 제외한 폼의 높이가 일정해질 것이다.




선뜻 이해가 되지는 않아도 좀 연구해 보면 재미있는 속성이다. 그런데 꼭 필요한 경우가 있기야 하겠지만 실용성이 그리 많은 것 같지는 않다.

# 다. 베벨

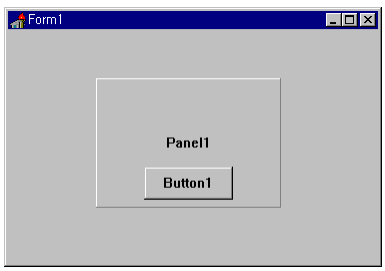
베벨(Bevel) 컴포넌트는 패널과 비슷한 외양을 가지고 있다. 그러나 베벨은 패널과는 달리 다른 컴포넌트를 담을 수 있는 컨테이너 컴포넌트는 아니며 단지 화면에 장식을 더할 뿐이다. 특수한 기능을 가지지 않는 단순한 도형일 뿐이다. 베벨의 모양은 Shape 속성과 Style 속성을 사용하여 설정한다. Shape 속성은 베벨의 모양을 사각 박스, 틀, 또는 단순한 직선으로 설정하며 Style 속성은 베벨이 음각의 모양을 가질 것인가 양각의 모양을 가질 것인가를 설정한다.

디폴트로 Shape 속성은 bsBox, Style 속성은 bsLowered이며  요런 모양을 가지며 Shape를 bsFrame으로 바꾸면  요런 모양이 된다. 베벨이 가질 수 있는 모양은 패널과 비슷하며, 단지 속성을 설정하는 방법만 조금 색다를 뿐이다. 베벨 컴포넌트는 이벤트도 전혀 가지고 있지 않다. 어디까지나 화면의 일정 위치에 있기만 할 뿐이며 단순한 장식에 사용되므로 프로그래밍 대상은 아니다.

# 라. 페어런트

패널에 버튼이나 에디트 등의 다른 컴포넌트를 놓으면 패널과 컴포넌트 사이에 부자 관계가 성립하여 패널이 부모(Parent)가 되고 버튼이나 에디트들이 자식(Child)이 된다. 부자 관계가 성립하면 자식 컴포넌트는 부모에 종속되며 부모가 이동하면 자식들도 같이 이동한다. 그리고 부모를 숨기면 자식도 숨겨지며 폰트나 도움말 등 여러 가지 속성을 부모로부터 영향을 받게 된다.

페어런트와 유사한 개념으로 오너(Owner)라는 개념이 있다. 오너란 컴포넌트들을 소유한 컴포넌트를 말하며 유일하게 폼만 오너가 될 수 있다. 오너가 파괴되면 오너에 소속된 모든 컴포넌트들도 파괴된다. 폼에 패널이 있고 패널 안에 버튼이 있을 경우를 보자.




이렇게 될 경우 버튼은 패널의 자식이 된다. 또한 버튼과 패널의 오너는 폼이 된다. 패널 안에 버튼을 배치하면 버튼의 페어런트는 패널이 되지만 오너는 여전히 폼이다.

부자 관계를 만들려면 컴포넌트를 최초 배치할 때 컨테이너 컴포넌트 안에 배치해야 한다. 만약 그렇게 하지 않고 컴포넌트를 폼에 배치했을 경우, 예를 들어 패널 밖에 버튼을 배치했을 경우 부자 관계를 만들려면 어떻게 할까? 단순히 드래그하여 버튼을 패널 위에 위치시킨다고 해서 부자 관계가 성립하는 것은 아니며 다음과 같이 해야 한다.

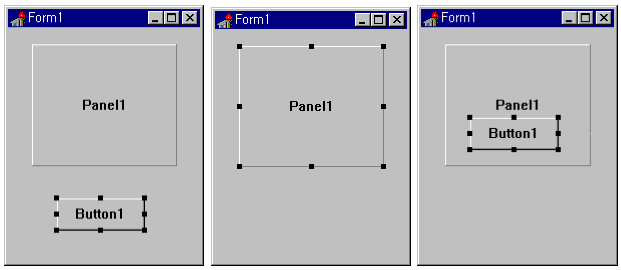
✶ 버튼을 선택한다.

✸ Edit 메뉴의 Cut 항목을 선택하여 버튼을 오려 낸다.

✹ 패널을 선택한다.

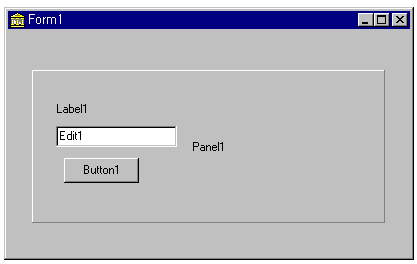
❋ 패널을 선택한 채로 Edit 메뉴의 Paste 항목을 선택하여 버튼을 패널 안으로 붙인다.

그림
부자 관계를 재편집하는 방법




물론 이렇게 하지 않고 버튼을 지워버리고 패널 안에 다시 만들어도 결과는 같지만 설정해 놓은 모든 속성은 다시 디폴트가 되어 버리므로 완전한 해결책이 되지는 못한다. 버튼을 패널에 집어 넣는 방법을 반대로 하면 패널과 버튼의 부자 관계를 끊어 버리는 것도 가능하다. 즉 패널 안에 있는 버튼을 잘라내고 폼을 선택한 후 폼에 버튼을 붙이기 하면 된다. 패널과 버튼뿐만 아니라 그룹 박스와 라디오 버튼 등, 부자 관계가 성립하는 모든 경우에 Cut, Paste로 컴포넌트를 재배치할 수 있다.

패널에 배치된 컴포넌트들을 선택할 때는 좀 특별한 방법을 사용한다. 여러 개의 컴포넌트를 같이 선택하고자 할 때는 Shift 클릭하거나 아니면 마우스로 선택하고자 하는 컴포넌트들을 감싸는 마키 실렉션을 사용하는데 패널에 놓인 컴포넌트들은 마키 실렉션을 할 수가 없다. 왜냐하면 패널에서 마우스를 드래그하면 패널이 이동되기 때문이다. 예를 들어 패널위에 레이블, 에디트, 버튼을 배치했다고 해보자.




이 상태에서 레이블, 에디트, 버튼을 선택하기 위해 마우스로 이 세 컴포넌트의 영역을 감싸 보자. 그러면 선택이 되는 것이 아니라 패널만 움직일 것이다. 컨테이너 컴포넌트에 속해 있는 자식 컴포넌트들을 마키 실렉션으로 선택하려면 Ctrl키를 사용해야 한다. 마우스로 사각 영역을 드래그하고자 하면 패널이 움직여질 뿐 자식 컴포넌트가 선택되지 않으므로 반드시 Ctrl키를 누른 채로 드래그해야 한다. 한번씩 실습해 보면 쉽게 방법을 터득할 수 있을 것이다.

# 마. 비트맵 버튼

비트맵 버튼은 버튼과 모든 면에서 동일한 기능을 가진다. 다만 한 가지 다른 점이라면 버튼 위에 예쁜 비트맵 그림을 놓을 수 있다는 점이다. 글자로서 버튼의 기능을 나타내는 것보다 그림이 첨가되면 훨씬 더 보기 좋은 버튼이 된다. 비트맵 버튼 위에 놓여질 그림은 Glyph 속성으로 사용자가 마음대로 지정하지만 Kind 속성을 사용하면 미리 정의된 자주 쓰이는 몇 가지 비트맵 버튼을 쉽게 만들 수 있다.

표 비트맵 버튼의 Kind 속성

| Kind | 버튼 모양 | 설명 |
|---|---|---|
| bkCustom | BitBtn1 | 사용자가 Glyph 속성을 사용하여 비트맵을 선택하는 버튼. 비트맵이 놓여지지 않는다. |
| bkOK | OK | Default 속성이 True가 되며 대화상자를 닫는 기능을 가진다. |
| bkCancel | Cancel | Cancel 속성이 True가 되며 대화상자를 닫는 기능을 가진다. |
| bkYes | Yes | Default 속성이 True가 되며 대화상자내에서의 변경 사항이 받아들여지며 대화상자가 닫힌다. |
| bkNO | No | Cancel 속성이 True가 되며 대화상자내에서의 변경 사항이 무시되며 대화상자가 닫힌다. |
| bkHelp | Help | 프로그램의 HelpFile 속성에 지정된 Help 화면이 나타난다. 버튼이 지정하는 HelpContext 속성이 사용된다. |
| bkClose | Close | Default 속성이 True가 되며 폼을 닫는다. |
| bkAbort | Abort | Cancel 속성이 True이다. |
| bkRetry | Retry | 재실행 버튼 |
| bkIgnore | Ignore | 무시 버튼 |
| bkAll | All | Default 속성이 True이다. |

Kind 속성을 변경하면 버튼의 Caption, Glyph 속성이 같이 변경되며 ModalResult 속성도 버튼의 기능에 맞게 변경된다. 예를 들어 Kind 속성을 bkOk로 바꾸어 주면 캡션은 Ok가 되며 Glyph도 체크 모양의 비트맵으로 바뀌며 ModalResult는 mrOk로 변경되어 누르는 즉시 대화상자를 닫을 수 있도록 기능을 부여한다. 그래서 Kind 속성을 변경하여 비트맵 버튼을 디자인할 때는

Kind 속성만 바꾸어 주면 되며 별다른 추가 작업을 해 줄 필요가 없다. 물론 Kind 속성을 변경한 후에 위치나 크기 조절이 가능하며 Glyph, Caption도 변경할 수 있다.

## ■ 비트맵 버튼의 모양 조정

우선 버튼의 모양에 가장 큰 영향을 미치는 요소로 버튼의 크기를 조정할 수 있으며 이 외에도 다음 세 가지 속성으로 비트맵과 캡션의 상호 위치 조정을 할 수 있다.

### ✍ LayOut

비트맵과 캡션의 배치상태를 지정하며 디폴트는 비트맵이 버튼의 좌측에 나타나지만 이 속성을 바꾸면 비트맵을 우측이나 위, 아래로 배치할 수 있다.


blGlyphLeft blGlyphRight blGlyphTop blGlyphBottom

### ✍ Margin

비트맵과 버튼과의 거리를 픽셀 단위로 지정한다. 디폴트값은 -1이며 이는 비트맵과 버튼의 거리를 지정하는 것이 아니라 버튼의 중앙에 비트맵과 캡션이 위치하도록 한다는 의미이다. Margin을 -1이 아닌 다른 값으로 바꾸면 버튼 위에서 비트맵의 위치를 세밀하게 조절할 수 있다.


Margin=-1 margin=3 Margin=30

### ✍ Spacing

비트맵과 캡션간의 간격을 지정한다. 디폴트는 4이며 이 값이 -1일 경우 비트맵과 캡션이 버튼의 가장자리로부터 적당한 간격만큼 떨어진다.

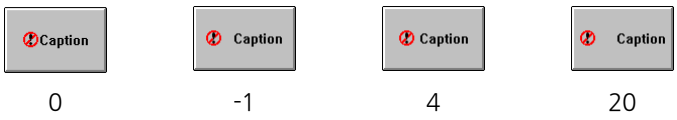
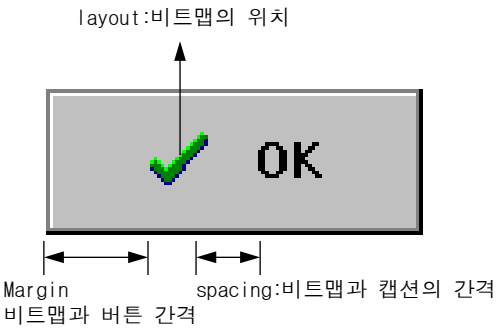
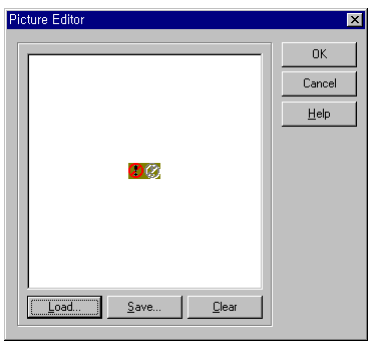



**그림**
비트맵 버튼의 모양을 결정하는 여러 가지 속성




## ■ 비트맵 배치

Kind 속성을 사용하지 않고 비트맵에 직접 그림을 배치하려면 Glyph 속성을 사용한다. Glyph 속성을 더블클릭하면 비트맵 파일을 읽어올 수 있는 다음과 같은 대화상자를 보여준다. 이 대화상자에서 Load 버튼을 눌러 버튼위에 놓을 비트맵 파일을 읽어오되 확장자가 BMP인 파일만 불러올 수 있다. 델파이 설치 CD에 델파이가 제공하는 버튼용 비트맵들이 많이 있으므로 이 중 한 파일을 선택하도록 한다. 제일 처음에 있는 ABORT.BMP를 선택하면 대화상자에 비트맵이 나타난다.

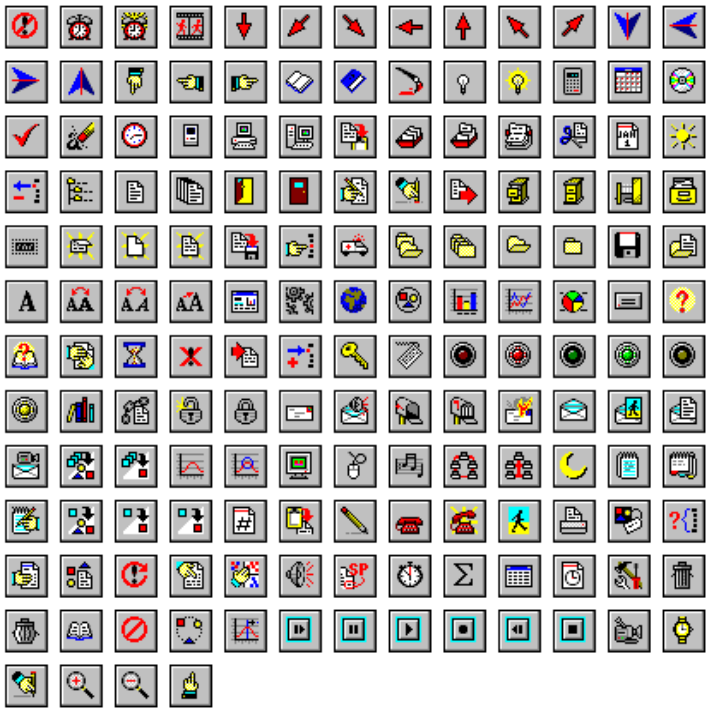



비트맵이 두 장으로 되어 있는데 왜 두장인가는 잠시 후에 알아보고 OK 버튼

을 눌러 대화상자를 닫도록 하자. 선택한 비트맵이 버튼 위에 나타난다. 버튼 위에 비트맵을 배치하는 것은 BMP 파일을 읽어와 Glyph 속성에 대입해 주기만 하면 된다. 델파이가 제공하는 비트맵 외에도 자기가 직접 비트맵을 작성할 수도 있다. 다음에 델파이가 제공하는 비트맵 샘플을 정리해 놓았다. 이 비트맵은 델파이 설치 CD의 Runimage₩Borland Shared₩Imageds₩Buttons 디렉토리에 있으며 디폴트 설치 옵션으로 설치하면 하드 디스크로 복사되지 않으므로 탐색기로 직접 복사해 놓도록 하자. 지면 절약을 위해 파일 이름은 생략하고 Buttons 디렉토리에 있는 버튼들을 알파벳순으로 정리한 것이므로 원하는 비트맵을 쉽게 찾을 수 있을 것이다.

**그림**

델파이가 제공하는 버튼용 이미지



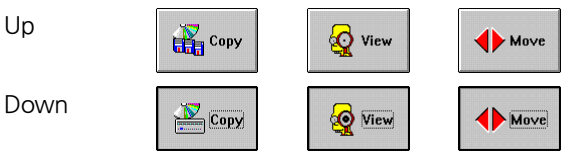
## ■ 여러 개의 비트맵

비트맵 버튼 위의 비트맵은 한꺼번에 4가지가 지정될 수 있으며 버튼의 상태에 따라 비트맵이 교체된다. 버튼이 가질 수 있는 상태는 다음 4가지가 있다.

| 상태 | 의미 |
|---|---|
| Up | 버튼이 눌러지지 않고 있는 상태이다. |

| | |
|---|---|
| Disable | 버튼을 선택할 수 없는 상태이며 보통 희미한 그림자 모양으로 표현한다. |
| Down | 버튼이 마우스에 의해 눌려진 상태이다. |
| StayDown | 버튼이 눌려져 있는 상태이며 마우스 버튼을 놓아도 다시 올라오지 않는다. 비트맵 버튼은 이 상태가 될 수 없으며 스피드 버튼만 가능하다. |

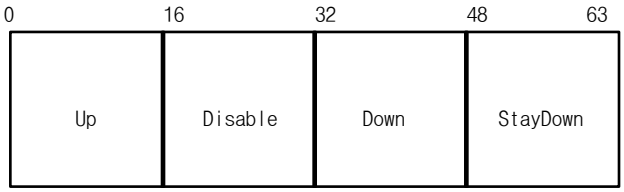
이 4가지 상태에 따라 비트맵을 개별적으로 지정하되 만약 Up 상태의 비트맵 하나만 정의되어 있으면 이 비트맵 하나로 4가지 상태를 모두 표현한다. 하지만 버튼을 좀 더 예쁘게 만들려면 버튼을 놓았을 때와 눌렀을 때의 그림을 바꾸어 주는 것이 좋다. 예를 들어 폴더 그림을 그려놓고 누를 경우 폴더가 열리는 버튼이라든가 사람의 눈동자를 그려놓고 누를 경우 커지는 버튼을 생각해 볼 수 있다.




버튼에 몇 개의 비트맵을 정의하여 사용할 것인가는 NumGlyphs 속성으로 지정한다. 이 속성값만큼 비트맵을 가로로 잘라서 각 버튼 상태에 대응시키게 된다. 델파이에서 제공하는 모든 비트맵은 크기가 가로 32, 세로 16이며 NumGlyphs가 2이며 2가지 모양의 비트맵을 제공한다.

Up Disable

4가지 상태에 별도의 비트맵을 제공하고 싶다면 비트맵의 가로 크기를 세로 크기의 4배만큼(예:가로 64 세로 16) 만들어 주고 NumGlyphs를 4로 설정한 후 비트맵에 4개의 그림을 그려 Glyph 속성에 대입해 준다.

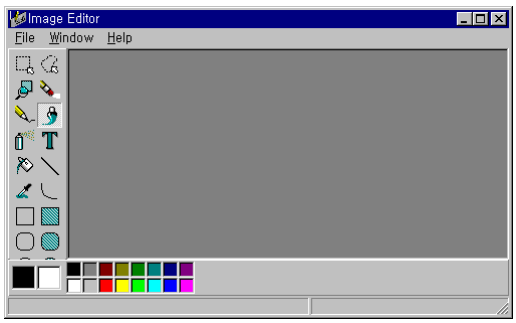



버튼의 상태에 따라 비트맵을 교체하는 일은 델파이가 자동으로 수행해 주므로 별도의 코드를 작성할 필요는 없다.

## ■ 이미지 에디터

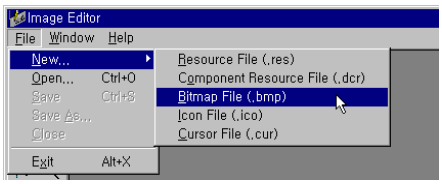
버튼에 놓을 비트맵을 작성하는 방법에는 여러 가지가 있다. 제일 간단한 방법으로는 윈도우즈의 표준 프로그램인 페인트 브러시를 사용하는 방법이 있고 그보다 좀 더 편리하고 많은 기능을 쓰고 싶다면 포토샵이나 페인트샵을 사용할 수도 있다. 델파이는 비트맵 제작을 위해 이미지 에디터라는 별도의 프로그램을 제공하는데 이 프로그램을 사용해도 비트맵을 만들 수 있다. 아무래도 전문 제작 툴인 이미지 에디터를 쓰는 편이 편리할 것이다. 이미지 에디터는 델파이 컴파일러 자체와는 다른 별도의 프로그램이므로 탐색기에서 IMAGEDIT.EXE를 찾아 직접 실행시켜야 한다. 아니면 델파이의 Tools/Image Editor 메뉴 항목을 선택하여 부를 수도 있다. 이미지 에디터의 모양은 다음과 같다.

그림
이미지 에디터



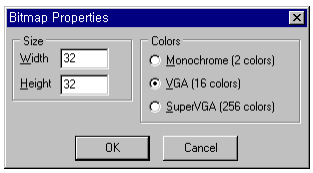

좌측에 도구 모음이 있고 하단에 색상을 선택할 수 있는 색상 그리드가 있다. 이미지 에디터의 사용 방법에 관해서 일일이 논할 수는 없고 여기서는 비트맵을 제작하는 순서만 알아본다.

File/New 를 선택하면 어떤 이미지를 만들 것인가를 선택하는 하위 메뉴가 열린다.



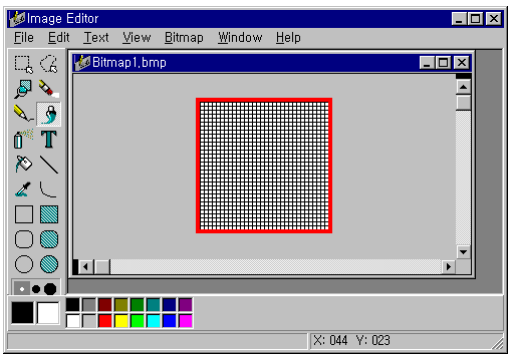

커서, 아이콘, 비트맵 등을 만들 수 있으며 통합적인 리소스를 만들 수도 있다. 세 번째 항목인 Bitmap File을 선택한다.

2 비트맵의 크기와 색상을 물어온다.




비트맵의 크기는 마음대로 지정할 수 있지만 버튼에 사용될 비트맵이라면 Width=Height*NumGlyphs가 만족하도록 만들어야 한다. 버튼에 쓰일 비트맵은 최하 16*16의 크기를 많이 사용하며 좀 크게 만들고 싶으면 24*24나 32*32를 사용한다. 색상은 아주 특별한 경우가 아니라면 16색을 쓰는 것이 좋다. 256색상을 사용하면 팔레트가 섞여 별로 보기에 좋지 못하다.

3 아무것도 그려져 있지 않은 비트맵과 도구 상자, 색상 선택 팔레트가 열린다. 도구 상자의 도구를 사용하여 비트맵을 작성한다. 비트맵을 확대하고 싶으면 View/Zoom In 명령을 선택한다.




**그림**

비트맵을 확대하면 점단위로 편집할 수 있다.

4 비트맵 디자인을 마친 후 File/Save 명령으로 비트맵 파일을 저장한다. 이렇게 저장한 파일을 비트맵 버튼의 Glyph 속성에 사용한다.

이미지 에디터 사용법은 대단히 직관적이며 사용하기 쉽다. 이미지 에디터에서 자체적으로 도움말을 제공하므로 자세한 사용 방법은 도움말을 참조하기 바라며 그렇지 않더라도 페인트 브러시를 사용하는 방법처럼 사용하면 큰 무리없이 사용할 수 있을 것이다.

참고하세요

비트맵 버튼에 사용된 비트맵은 실행 파일에 직접 포함된다. 따라서 비트맵을 실행 파일과 같이 배포할 필요는 없으며 원본 비트맵은 한번 Glyph 속성에 대입하면 지워도 상관없다. 하지만 차후의 수정을 위해서는 가급적 남겨두는 것이 좋을 것이다. 한 가지 주의할 것은 비트맵을 수정할 경우 비트맵만 바꾼다고 해서 실행 파일의 비트맵이 교체되는 것은 아니라는 점이다. 비트맵을 수정했으면 반드시 Glyph 속성에 다시 대입해 주어야 수정한 비트맵이 버튼 위에 나타나게 된다.

# 바. 스피드 버튼

스피드 버튼은 비트맵 버튼과 비슷하다. 버튼 위에 비트맵을 배치할 수 있고 캡션을 줄 수 있으며 Layout, Spacing, Margin 속성으로 모양을 조정하며 여러 개의 비트맵을 쓸 수도 있다. 하지만 두 버튼이 완전히 같지는 않으며 다음과 같은 차이점이 있다.

첫째, 한번 눌러 놓으면 마우스 버튼을 떼어도 다시 버튼이 올라오지 않으며 눌러진 채로 있을 수 있다는 점이다. 버튼은 누른 후 마우스를 떼면 다시 올라오지만 스피드 버튼은 버튼을 누른 채로 그대로 둘 수도 있다. 물론 보통의 버튼처럼 누르면 다시 올라오도록 할 수도 있다. 예를 들어 윈도우즈용 워드 프로세서에서 글꼴 모양을 지정하는 버튼들이 이런 형태를 가진다.

둘째, 홀로 쓰일 수도 있지만 그룹으로도 사용될 수 있다. 예를 들어 문단 모양이나 탭 종류를 지정하는 버튼이 그룹으로 사용된다.

그림
그룹으로 사용되는 스피드 버튼

GroupIndex 속성이 같은 스피드 버튼은 모두 하나의 그룹을 이루며 그룹내에서는 하나의 스피드 버튼만 선택된다. 버튼 형식의 라디오 버튼이라고 할 수 있다. 단 GroupIndex가 0인 경우는 그룹으로 인정되지 않는다. 0 이외의 GroupIndex를 가져야 그룹으로 인정된다. 그룹으로 사용하지 않고 홀로 사용하려면 스피드 버튼의 GroupIndex 속성을 0으로 설정한다.

다음과 같이 4개의 스피드 버튼을 배치하고 Glyph 속성에 아무 비트맵이나 넣어 주고 GroupIndex를 모두 1로 바꾸어 보자.


그리고 실행을 시켜보면 4개 중의 한 개만 선택된다. 라디오 버튼과 같이 하나가 선택되면 그 이전의 버튼은 다시 튀어 올라와 버리므로 상호 배타적인 옵션을 선택할 때 사용된다. GroupIndex가 0이면 스피드 버튼은 비트맵 버튼과 같이 완전히 개별적으로 동작하며 StayDown 상태를 가지지 않는다. GroupIndex가 0 이외의 값이고 다른 스피드 버튼과 GroupIndex가 중복되지 않으면 혼자서 그룹을 이루어 StayDown 상태를 토글시킬 수 있다. 스피드 버튼의 주요한 네 가지 속성을 정리해 보자.

### GroupIndex

0 이 아니면 같은 그룹으로 동작한다. 디폴트값은 0 이며 따라서 처음 만든 스피드 버튼은 그룹을 이루지 않는다. 그룹을 이루고자 하는 스피드 버튼의 GroupIndex 속성을 0 이외의 다른 값으로 일치시켜 주면 같은 그룹 안의 스피드 버튼이 상호 배타적으로 선택된다.

### AllowAllUp

그룹 내의 모든 버튼이 Up 상태가 될 수 있는가를 지정한다. 이 옵션이 False 이면 그룹 내의 한 버튼은 반드시 Down 되어 있어야 하지만 True 일 경우는 모든 버튼이 Up 상태일 수도 있다. 한 버튼의 이 속성을 바꾸면 그룹 내 모든 버튼의 AllowAllUp 속성이 변경된다.

### Down

버튼의 처음 상태를 지정하며 True 일 경우 처음 프로그램을 실행시킬 때부터 눌러진 상태가 된다.

✍ Flat

경계선이 없는 납작한 모양의 버튼을 만든다. 실행중에 경계선이 보이지 않으며 마우스 커서가 버튼 위로 이동하면 경계선이 나타난다.

스피드 버튼도 다른 버튼처럼 OnClick 이벤트를 가지므로 버튼 선택시의 처리는 OnClick 이벤트 핸들러에 작성한다.

# 사. 패널을 이용한 툴바

7jang speed

툴바는 보통 메뉴 밑에 조그만 버튼들을 모아놓아 명령이나 옵션을 쉽게 선택할 수 있도록 해준다. 델파이는 별도의 툴바 컴포넌트를 제공하기도 하지만 여기서는 실습을 위해 패널과 스피드 버튼을 사용해 보기로 하자. 다음 과정을 따라 툴바를 만든다.

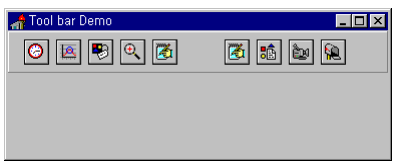
① 패널을 먼저 배치한다. 패널 위에 스피드 버튼을 배치할 것이므로 Caption 속성은 지우는 것이 좋다.

② 툴바는 보통 폼의 상단에 위치하므로 패널의 Align 속성을 alTop 으로 설정하여 항상 폼의 상단에 위치하도록 해준다. 물론 툴바를 왼쪽이나 오른쪽에 두려면 적당히 Align 속성을 설정해 주면 된다. 폼의 위쪽에 정렬시킨 후 적당히 높이를 조정한다.

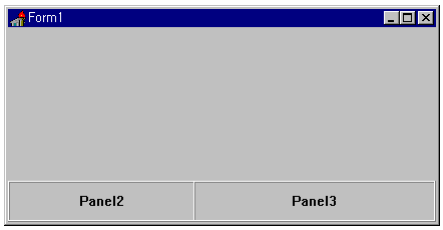
③ 패널 안에 스피드 버튼을 필요한만큼 배치한다. 물론 스피드 버튼뿐만 아니라 콤보 박스나, 체크 박스 등의 다른 컴포넌트들도 배치할 수 있다.

④ 스피드 버튼의 코드를 작성한다.

기본적인 절차는 대충 이렇고 필요에 따라 스피드 버튼을 그룹으로 엮는다거나 툴바를 이중으로 만든다거나 별도의 처리를 첨가할 수 있다. 다음이 툴바를 만들어 본 예이다.



비슷한 방법으로 패널을 사용하면 상태란도 만들 수 있는데 툴바와는 달리 주로 폼의 하단에 놓여진다는 점이 다르며 스피드 버튼 대신 문자열이 위치한다. 상태란 내에 여러 가지 정보를 기록하기 위해 구획을 나눌 필요가 있으면 패널을 중첩시키면 된다. 패널을 중첩시킨다는 말은 패널 안에 패널을 배치한다는 말이다.




물론 패널에는 여러 가지 다른 컴포넌트들도 얼마든지 배치할 수 있다. 그런데 델파이에서는 별도의 상태란 컴포넌트를 따로 제공하므로 이런 식으로 상태란을 만들 필요는 거의 없으며 상태란에 콤보 박스나 특수한 컴포넌트를 배치하고자 하는 경우 정도에나 이런 기법이 사용된다. 이런 식으로도 패널을 사용할 수 있다는 것만 알아 두도록 하자.

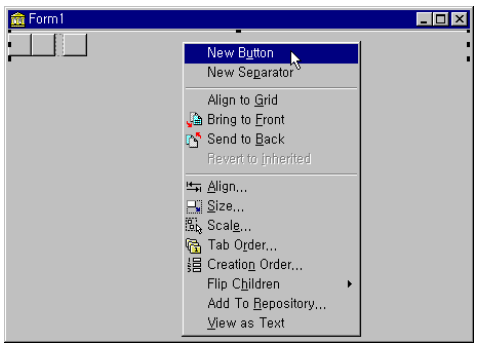
# 7-5 툴바 만들기

## 가. TToolBar 컴포넌트

7jang
ToolBar

툴바를 만들기 위해서는 패널을 사용할 수도 있고 또는 델파이가 별도로 제공하는 TToolBar 라는 컴포넌트를 사용할 수도 있다. 이 컴포넌트를 폼에 배치하면 디폴트로 폼의 상단(alTop)에 밀착되며 툴 버튼을 포함하여 콤보 박스나 체크 박스를 배치할 수 있는 컨테이너가 된다. 툴 버튼을 배치할 때는 툴바의 팝업 메뉴에서 New Button 항목을 선택하며 버튼 사이에 여백을 두고자 할 경우 New Separator 를 선택하면 된다.

그림
툴바의 스피드 메뉴

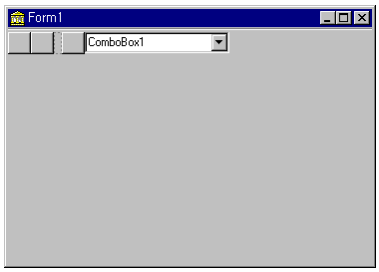



툴바에 놓여지는 버튼은 TToolButton 컴포넌트이며 컴포넌트 팔레트에는 없으므로 툴바의 팝업 메뉴를 통해 추가해야 한다. 마치 TMenuItem 컴포넌트를 메뉴 디자이너로 만드는 것과 같다. 툴 버튼의 속성을 변경하거나 이벤트 핸들러를 작성할 때는 여타의 컴포넌트와 마찬가지로 오브젝트 인스펙터를 사용하면 된다.

New Button 명령으로 추가된 버튼은 비트맵을 가지지 않은 밋밋한 모양이 되 각 버튼에 개별적으로 비트맵을 지정할 수는 없으며 이미지 리스트를 사용해야 한다. 툴바를 사용하는 간단한 예제를 만들어 보자.

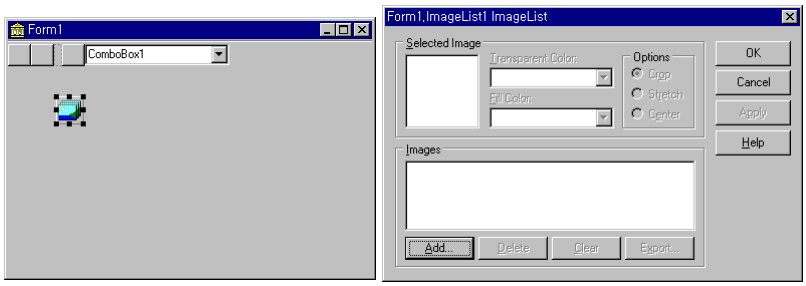
새 프로젝트를 시작하고 툴바 컴포넌트를 폼에 배치한다. 폼 상단에 밋밋한 사각형 모양으로 배치될 것이다. 그리고 툴바의 팝업 메뉴에서 New Button 명령을 사용하여 다음과 같이 적당히 버튼을 배치하고 콤보 박스도 같이

배치해 보자. 버튼 두 개, 여백 하나, 버튼 한 개, 콤보 박스 하나를 각각 배치하였다.

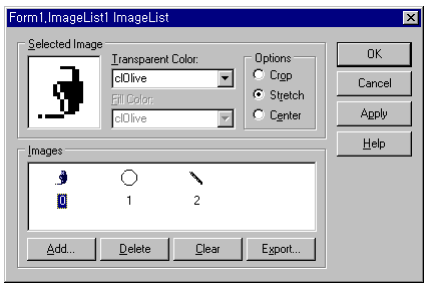



2 버튼에 비트맵을 주기 위해 이미지 리스트 컴포넌트를 폼에 하나 배치하고 팝업 메뉴에서 ImageList Editor 항목을 선택하거나 아니면 이미지 리스트 컴포넌트를 더블클릭한다. 왼쪽과 같은 이미지 리스트 편집기가 열릴 것이다.

그림
이미지 리스트 편집기



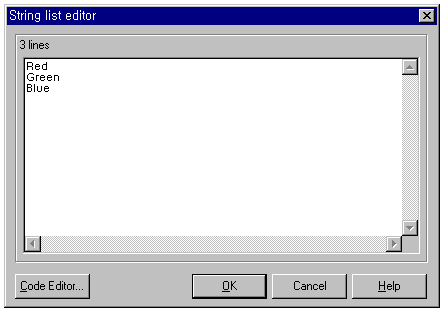

3 Add 버튼을 눌러 미리 만들어져 있는 이미지를 불러오되 여기서는 직접 만들지 말고 만들어져 있는 이미지를 사용하도록 하자. Demos/Doc/GraphEx 디렉토리에 있는 세 가지 이미지를 가져온다. 세 개의 이미지를 불러온 후의 이미지 리스트 편집기 모양은 다음과 같다.




4 툴바의 Images 속성에 ImageList1 을 선택하여 이미지 리스트에 저장된 비트맵이 툴 버튼의 표면에 나타나도록 해준다. 툴 버튼의 ImageIndex 속성

은 이미지 리스트의 몇 번째 이미지를 사용할 것인가를 지정하는데 생성된 순서대로 0,1,2 와 같이 되어 있다. 이 속성값을 변경하면 이미지 리스트내의 다른 이미지도 사용가능하다.

5 콤보 박스의 Items 속성에 다음 세 항목을 추가한다.




6 이제 코드를 작성해 보도록 하자. 콤보 박스의 OnChange 이벤트 핸들러를 다음과 같이 작성한다. 콤보 박스에서 선택한 색상으로 폼의 색상을 변경하도록 하였다.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  if ComboBox1.Text = 'Red' then Form1.Color := clRed;
  if ComboBox1.Text = 'Green' then Form1.Color := clGreen;
  if ComboBox1.Text = 'Blue' then Form1.Color := clBlue;
end;
```

그리고 첫 번째 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
  ShowMessage('첫 번째 버튼을 눌렀습니다');
end;
```

코드의 내용상 콤보 박스에서 항목을 변경하면 폼의 색상이 변경되며 첫번째 툴 버튼을 누르면 메시지를 보여주기만 하도록 되어 있다. 그런데 폼의 색상이 변경되면 툴바의 색상까지 한꺼번에 변경되는 문제가 있다. 왜 그런가 하면 툴바의 ParentColor 속성이 디폴트로 True 이기 때문에 폼의 색상을 따라가기 때문이다. ParentColor 속성을 False 로 바꿔주면 이 문제가 해결된다. 나머지 버

튼에도 자유롭게 기능을 줄 수 있다. 툴 버튼을 추가하고 이미지 리스트를 사용하는 것 외에 특별히 어려운 내용은 없으므로 몇 가지 속성만 간단하게 정리하고 넘어가도록 하자.

✍ **Images**

각 툴 버튼들의 이미지를 가지는 이미지 리스트를 지정한다. 툴 버튼의 ImageIndex 속성으로 이미지 리스트의 몇 번째 이미지를 사용할 것인가가 지정된다.

✍ **Disabled Image**

Images 이미지 리스트가 평상시의 버튼 이미지를 가지는데 비해 이 이미지 리스트는 버튼이 disable 상태가 되었을 때의 이미지를 가진다. 이 속성이 지정되어 있지 않으면 Images 의 이미지를 그레이 상태로 만든 이미지가 대신 사용된다.

✍ **Hot Images**

마우스 커서가 버튼위에 위치할 때의 버튼 이미지를 가지는 이미지 리스트이다.

✍ **AutoSize**

툴바의 높이를 툴바안에 배치된 버튼의 높이에 맞게 자동으로 맞춘다.

✍ **ButtonWidth, ButtonHeight**

툴바안에 배치되는 툴 버튼의 폭과 높이를 지정한다. 디폴트는 22*23 으로 되어 있다.

✍ **EdgeBorder**

툴바의 경계선을 설정하는데 상하좌우 어느쪽에 경계선을 표시할 것인가를 지정하며 ebLeft, ebTop, ebRight, ebBottom 네 개의 불린값을 개별적으로 선택한다. 디폴트로 ebTop 만 선택되어 있어 상단에만 경계선이 나타나는데 ebBottom 도 True 로 바꿔주면 툴바 하단에도 경계선이 나타난다.

✍ **Flat**

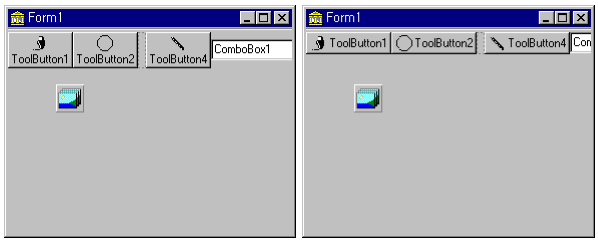
툴 버튼의 경계선을 없애며 마우스 커서가 버튼 위치에 있을 때만 경계선이 나타나도록 한다.

✍ **Indent**

첫 번째 버튼과 툴바와의 거리를 픽셀 단위로 지정한다.

✍ Show Caption

툴 버튼 아래에 버튼의 캡션도 같이 표기한다. 이때 캡션의 방향은 List 속성이 True 이면 오른쪽에 False 이면 아래쪽에 나타난다. 다음은 아래쪽과 오른쪽에 캡션을 나타내 본 것이다.



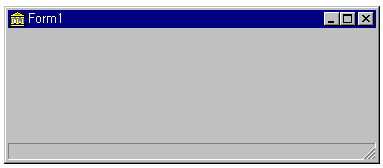

이 외에도 툴바 컴포넌트의 모양을 정의하는 여러 가지 다양한 속성들이 있는데 자세한 사항은 레퍼런스를 참고하기 바란다.

# 나. 상태란

앞에서 패널로 상태란을 만드는 방법에 대해 잠시 언급을 한 적이 있는데 윈도우즈 95 에서는 시스템 차원에서 상태란이 지원되므로 이보다는 별도의 상태란 컴포넌트를 사용하는 것이 더 편리하다. 상태란은 Win32 페이지에 있는 StatusBar 컴포넌트로 쉽게 만들 수 있다. 이 컴포넌트는 상태란을 여러 칸으로 분리해서 각 칸에 원하는 정보를 출력할 수 있도록 해준다. 설명을 하는 것보다는 간단하나마 예제를 하나 만들어 보는 것이 더 나을 것 같다.


7jang
Status

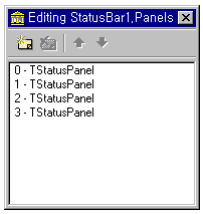
새 프로젝트를 시작하고 Win32 페이지에서 StatusBar 컴포넌트를 가져다가 폼에 배치해 보자. 툴바가 폼의 상단에 정렬되는 것과는 대조적으로 상태란은 폼의 하단에 정렬된다. 처음 배치되었을 때는 다음 그림처럼 아무것도 없는 밋밋한 사각형 뿐이다.




이 상태로도 물론 사용할 수 있지만 보통 상태란은 여러 칸으로 나누어 각 칸에 정보들을 출력하는 것이 보통이다. 이 컴포넌트를 더블클릭하거나 또는 오브

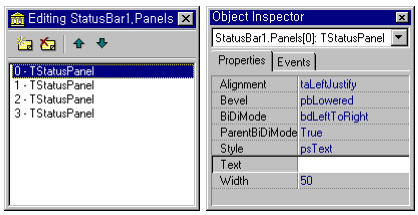
젝트 인스펙터에서 Panels 속성을 더블클릭하거나 아니면 팝업 메뉴에서 Panels Editors... 항목을 선택하면 다음과 같은 패널 편집기가 열린다.

그림
패널 편집기




이 편집기에서 원하는 패널 수만큼 상단 툴바의 Add New 버튼을 눌러주면 세부 패널이 생성되는데 이 때 생성되는 패널들은 TStatusPanel 이라는 컴포넌트이며 상태란의 Panels 배열 요소가 된다. 일단 Add New 버튼을 4 번 눌러 4 개의 패널을 생성해 보자. 패널을 생성한 후 패널 편집기에서 패널을 선택하면 오브젝트 인스펙터에는 각 패널의 속성들이 출력되는데 이 속성들을 변경하여 개별 패널의 모양이나 텍스트를 편집한다.

그림
패널의 속성 변경




Text 속성은 패널에 나타날 문자열이며 Width 는 패널의 폭, Bevel 은 패널의 모양을 정의하는 속성이다. 일단은 모두 디폴트 속성을 사용하도록 하자. 4 개의 패널을 배치한 후의 상태란은 다음과 같아진다.


4 개의 패널로 구획이 나누어져 있다. 이중 앞쪽 두 개는 현재 마우스 커서의 위치를 출력하도록 하고 세 번째 패널에는 Num Lock 키의 상태를 출력하도록 해보자. 현재 마우스 커서를 얻기 위해서는 폼의 OnMouseMove 이벤트의 핸들러를 작성하면 된다.

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
```

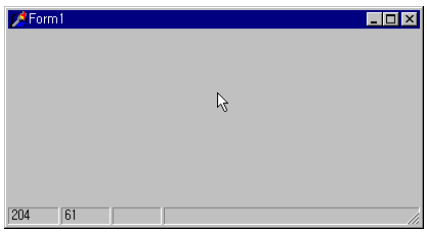
```
  Y: Integer);
begin
  StatusBar1.Panels[0].Text:=IntToStr(X);
  StatusBar1.Panels[1].Text:=IntToStr(Y);
end;
```

인수로 전달된 마우스 위치를 문자열로 변경하여 첫 번째 패널과 두 번째 패널의 Text 속성에 대입하였다. 코드를 보다시피 패널은 상태란의 Panels 배열로 접근할 수 있으며 첨자 0 번이 첫 번째 패널, 첨자 1 번이 두 번째 패널이 된다.

NumLock 키의 상태를 알아내기 위해서는 GetKeyState API 함수를 사용하며 폼의 OnKeyDown 이벤트를 사용하면 적절할 것 같다. GetKeyState 함수로 VK_NUMLOCK 키값을 조사한 후 이 키가 눌러져 있으면 세 번째 패널에 NUM 이라는 텍스트를 주고 그렇지 않으면 텍스트를 지워버린다.

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
if (GetKeyState(VK_NUMLOCK) and 1)=0 then
   StatusBar1.Panels[2].Text:=''
else
    StatusBar1.Panels[2].Text:='NUM';
end;
```

GetKeyState 함수에 대해서는 도움말을 참고하기 바란다. 실행중의 모양은 다음과 같다.




여기서는 패널 세 개만 사용했는데 출력해야 할 정보가 많다면 얼마든지 많은 패널을 만들어 보여줄 수도 있다.

상태란은 이런식으로 패널을 나누어 사용할 수도 있지만 통째로 하나의 긴 문자열을 출력할 수도 있다. 이 때는 상태란의 SimplePanel 속성을 True 로 바꾸어 주고 출력하고자 하는 문자열을 SimpleText 속성에 대입해 주면 된다. 이 속성은 실행중에도 언제든지 변경될 수 있기 때문에 상황에 따라 패널을 나누어 쓰

기도 하고 긴 문자열을 출력하기도 한다.

상태란의 또 다른 기능으로 오른쪽에 있는 크기 조절 핸들(Size Grip)이 있다. 세 줄의 빗금 모양으로 되어 있는데 단순한 장식같아 보이지만 장식효과보다는 윈도우의 크기를 쉽게 조절할 수 있도록 한다. 마우스 커서가 꼭 경계선에 있지 않아도 크기 조절 핸들위에 있으면 윈도우 크기 조절이 가능해진다. 이 기능은 상태란의 SizeGrip 속성이 True 일 때만 사용할 수 있는데 디폴트로 이 속성은 True 이다. 크기 조절 핸들을 사용하지 않을 때는 SizeGrip 을 False 로 변경한다.

# 다. 쿨바

7jang Coolbar

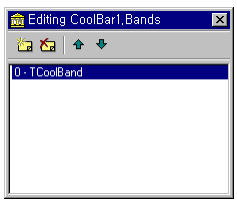
쿨바는 개별적으로 위치를 옮길 수 있고 크기를 조절할 수 있는 차일드 컨트롤을 담는 컨테이너이다. 쿨바에는 쿨 밴드가 들어가며 쿨 밴드에는 개별 컨트롤이 들어간다. 주로 TWinControl 의 후손들이 쿨 밴드에 들어가지만 TWinControl 의 후손이 아닌 컨트롤들도 들어갈 수 있다. 쿨바를 사용하는 단계는 비교적 복잡하므로 아주 극단적으로 간단한 쿨바를 단계를 따라 만들어 보도록 하자.

1 새 프로젝트를 만들고 쿨바를 폼에 배치한다. 디폴트로 alTop 정렬을 가지므로 폼 위쪽에 밀착되어 배치될 것이다.

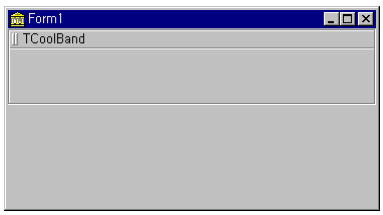
2 쿨 밴드를 만든다. 쿨바의 Bands 속성을 더블클릭하거나 쿨바의 팝업 메뉴에서 Bands Editor 를 선택하면 쿨 밴드 편집기가 열릴 것이다.

그림

밴드 편집기

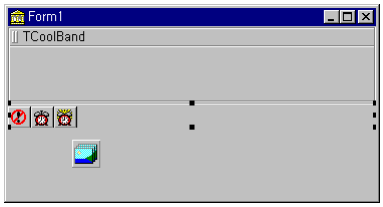



상단의 Add New 버튼을 눌러 쿨 밴드 하나를 만든다. 0 번 쿨 밴드가 생성되며 쿨바의 위쪽에 쿨 밴드가 만들어져 있을 것이다.



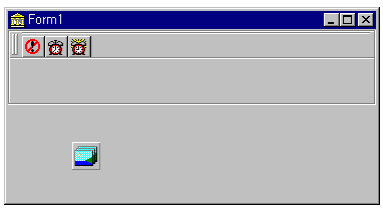
이렇게 만들어진 쿨 밴드에 툴바 등의 컨트롤이 배치된다.

3 간단하게 툴바 하나를 만든다. 툴바 컴포넌트를 폼에 배치한 후 버튼 세 개를 만든다. 물론 이미지 리스트를 사용하여 버튼에 이미지도 그려 넣어야 한다. 여기까지 작업한 후의 폼 모습은 다음과 같다.

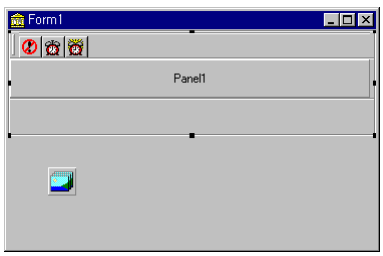



쿨바안에 쿨 밴드가 있고 툴바는 독립적으로 배치되어 있다.

4 툴바를 쿨바에 집어넣기 위해 쿨 밴드의 Controls 속성에 Toolbar1을 대입한다. 쿨 밴드의 속성을 변경하려면 쿨 밴드 에디터에서 쿨 밴드를 선택한 후 오브젝트 인스펙터를 사용해야 한다. 여기까지 작업하면 쿨 밴드가 있는 자리에 툴바가 배치된다.




만약 이 상태에서 툴바를 더 배치하려면 원하는만큼 쿨 밴드와 툴바를 만들어 주면 된다. 툴바뿐만 아니라 패널이나 콤보 박스, 에디트 등도 얼마든지 배치할 수 있는데 이번에는 패널을 하나 더 배치해 보도록 하자. 밴드 편집기에서 Add 버튼을 눌러 새 밴드를 하나 만들어 둔다. 그리고 폼에 패널을 배치해 두고 밴드의 Controls 속성에 Panel 을 대입해 주면 패널이 밴드 안으로 들어가게 된다.



이렇게 배치된 패널에는 다른 컴포넌트를 얼마든지 배치할 수 있다.

# 라. 간단한 에디터 3

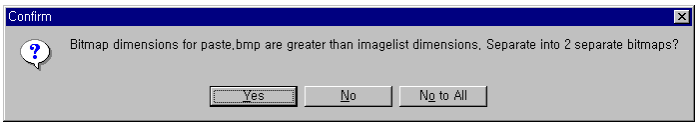
7jang
editor3

5장에서 메뉴와 클립보드 기능을 첨가하여 Editor2를 만들었다. 이제 툴바를 만드는 법까지 알았으므로 에디터에 툴바를 추가해 보자. 툴바 제작에 관한 사항은 앞에서 설명했으므로 자세한 설명은 생략하도록 한다. 먼저 Editor3 디렉토리를 만들고 Editor2 프로젝트를 이 디렉토리로 옮겨오도록 한 후 Editor3를 수정해 보자.

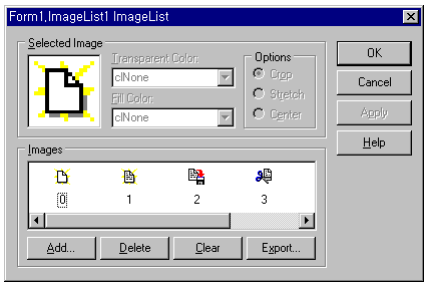
Editor3에 필요한 툴 버튼은 새파일, 열기, 저장, Cut, Copy, Paste 모두 여섯 개이다. 파일 관련 툴 버튼은 당장 사용하지 않지만 차후 확장을 고려하여 미리 만들어 놓기로 한다. 툴바에 있는 Cut, Copy, Paste 기능은 메뉴에 있는 항목들과 일치하므로 통일적인 UI관리를 위해 액션 리스트를 사용해 보도록 하자.

툴바 컴포넌트를 배치하면 폼 상단에 밀착될 것이다. 이 툴바에 여섯 개의 툴 버튼을 배치하되 파일 관련 명령 세 개와 클립보드 관련 명령 세 개를 분리하기 위해 중간에 여백을 하나 둔다. 그리고 각 버튼에 이미지를 주기 위해 이미지 리스트를 배치하고 델파이가 제공하는 이미지를 적절히 불러온다.

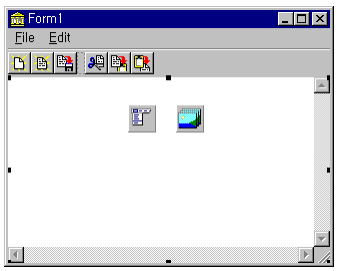
델파이가 제공하는 이미지 중에 filenew, fileopen, filesave, cut, copy, paste 등 우리가 사용하고자 하는 이미지들이 모두 제공되므로 불러와 이미지 리스트에 포함시키기만 하면 된다. 이때 이미지 리스트는 이미지를 불러올 때마다 다음과 같은 질문을 해 올 것이다.



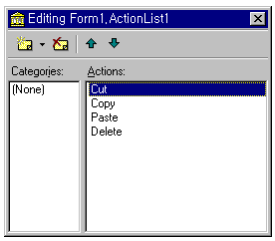

델파이가 제공하는 기본 이미지들은 모두 32*16의 크기를 가지며 Up, Disable 상태를 모두 가지고 있다. 이 두 이미지를 분리할 것인가를 물어오는 질문인데 일단 Yes라고 대답하여 두 개의 이미지를 불러온 후 Disable 이미지는 삭제하도록 하자. 여섯 개의 이미지를 다 읽어들이면 이미지 리스트는 다음과 같이 된다.




이제 툴바의 Images 속성에 ImageList1을 대입하면 툴 버튼에 이미지들이 나타날 것이다.




툴바를 만들었으면 작동할 수 있도록 코드를 작성해 보자. 그런데 기존의 클립보드 동작이 메뉴 항목에 다 정의되어 있기 때문에 메뉴 항목의 이벤트 핸들러에 툴바의 OnClick 이벤트 핸들러만 연결해 주어도 된다. 그러나 그렇게 하면 툴바의 UI 상태를 변경할 수 없으므로 좀 더 발전된 방법인 액션 리스트를 사용해 보자. 액션 리스트 컴포넌트를 폼에 배치하고 액션 리스트를 더블클릭하여 액션 편집기를 연다. 그리고 다음과 같이 4개의 액션을 만든 후 Name 속성을 변경한다.




그리고 각 액션의 OnExecute 이벤트 핸들러를 다음과 같이 작성한다. 기존의 메뉴 항목 핸들러에 있던 코드들을 액션의 핸들러로 옮기기만 하면 된다.

```
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;
```

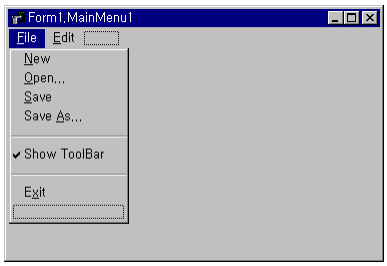
그리고 메뉴 항목의 OnClick 이벤트 핸들러는 이제 필요가 없으므로 삭제한다. 각 핸들러의 본체 코드만 삭제해 두면 된다. 이렇게 만들어진 액션을 메뉴 항목과 툴 버튼의 Action 속성에 대입해 준다. 즉 Cut 액션을 Cut1 메뉴 항목과 ToolButton5의 액션 속성에 대입해 준다. 단 이렇게 되면 툴 버튼의 ImageIndex 속성이 액션의 값인 -1을 따라가기 때문에 이미지가 사라지게 되는데 툴 버튼과 메뉴의 ImageIndex 속성을 수동으로 각각 3,4,5로 다시 설정해 주면 된다.

액션을 만들었으니 액션의 OnUpdate 이벤트 핸들러도 만들어 보자. 네 액션의 OnUpdate 핸들러를 모두 UpdateUI라는 이름으로 만든다. 그리고 기존의 OnEdit1Click 이벤트 핸들러에 있던 코드를 잘라내서 액션의 UpdateUI 이벤트 핸들러에 붙이고 Cut1, Paste1 등 메뉴 항목의 Action 속성을 Cut, Paste 등의 액션으로 변경해 주면 된다. 물론 이렇게 되면 OnEdit1Click 이벤트 핸들러는 지워도 상관없다.

```
procedure TForm1.UpdateUI(Sender: TObject);
begin
  Paste.Enabled:=Clipboard.HasFormat(CF_TEXT);
  if Memo1.SelLength=0 then
    begin
    Cut.Enabled:=False;
    Copy.Enabled:=False;
    Delete.Enabled:=False;
```

```
    end
   else
    begin
    Cut.Enabled:=True;
    Copy.Enabled:=True;
    Delete.Enabled:=True;
   end
  end;
```

이제 메뉴 항목을 선택하는 대신 툴바를 눌러 더 빠른 조작을 할 수 있어 여러모로 편리할 것이다. 뿐만 아니라 메뉴와 툴바의 UI 상태까지 한꺼번에 갱신되는 추가적인 이점까지 생겼다. 여기에 메뉴 항목을 하나 더 추가해 보기로 하자. 툴바를 표시하거나 숨길 수 있는 Show ToolBar라는 메뉴 항목을 File 메뉴 아래에 다음과 같이 추가한다.

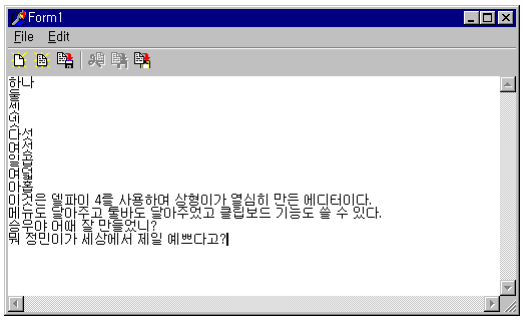



이 메뉴 항목은 툴바의 숨김/보임을 토글하는 용도로 사용되는데 처음 시작할 때 툴바가 보이므로 Checked 속성을 True로 변경하였다. 이 메뉴 항목의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.ShowToolBar1Click(Sender: TObject);
begin
if ShowToolbar1.Checked=False then
  begin
  ShowToolbar1.Checked:=True;
  ToolBar1.Show;
  end
else
  begin
  ShowToolbar1.Checked:=False;
  ToolBar1.Hide;
  end;
end;
```

이 메뉴 항목을 선택할 때마다 툴바의 표시, 숨김을 토글하도록 하였으며 자

기 자신의 체크 표시도 함께 토글시킨다. 그 외 툴 버튼에 힌트를 주어 풍선 도움말도 나타나도록 하였다.

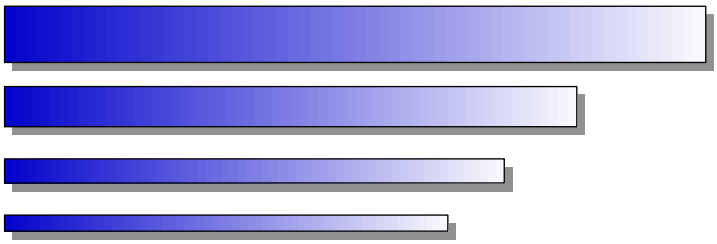



이제 예제를 실행시켜 보자. 툴바와 메뉴 모두 정상적으로 동작할 뿐만 아니라 메뉴 옆에도 이미지가 나타날 것이다. 이 예제는 만드는 과정이 워낙 복잡해 보여 혹시 책대로 따라했는데 잘 안되는 경우도 있을 것이다. 만약 그렇다면 배포 CD의 프로젝트를 읽어와서 분석해 보기 바란다.

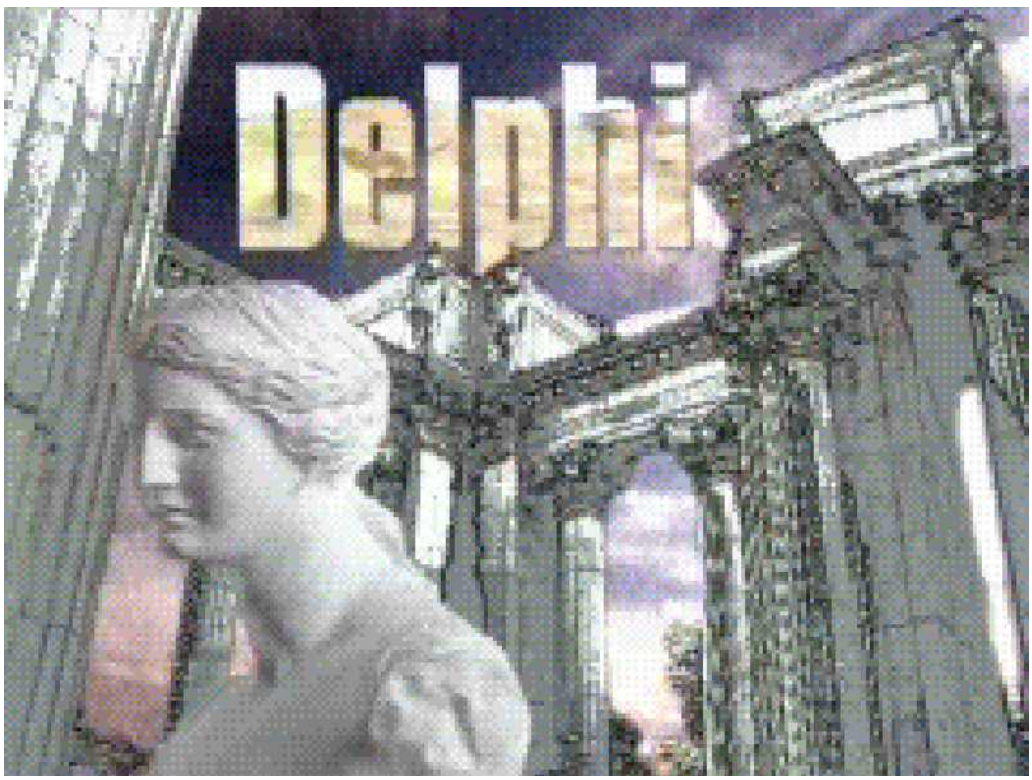
툴바를 다는 일은 무척 간단하면서도 프로그램의 격을 한층 더 높여 준다. 툴바까지 만들었으니 제법 에디터 모양이 나기는 하지만 아직까지 이 에디터는 기본적인 기능이 빠져있다. 에디터 본래의 기능인 편집은 할 수 있지만 편집한 결과를 파일에 저장하는 기능이 없다. 파일 입출력은 11장에서 실습해 보고 11장에서 실용적으로 쓸만한 에디터를 만들어 보자.

이 장에서 알아본 컴포넌트들은 기본적인 용도로 사용되는 쉬운 것들이다. 개별 컴포넌트의 세세한 면을 일일이 익히려고 하기 보다는 컴포넌트의 일반적인 특성에 대해 숙지하도록 하고 도움말을 사용하여 필요한 사항을 빨리 찾을 수 있는 능력을 기르는 것이 더 중요하다고 생각한다. 나머지 컴포넌트는 다음 장부터 관련 부분에서 순서대로 소개하기로 한다.

# 그래픽

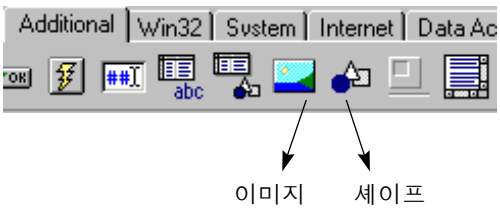


# 제 8 장



# 8-1 그래픽 컴포넌트

델파이가 기본적인 그래픽 출력을 위해 제공하는 컴포넌트는 셰이프와 이미지 두 가지이며 둘 다 Additional 페이지에 있다. 두 개의 컴포넌트로 기본적인 그래픽 표현을 할 수 있으며 또한 폼에는 그래픽 출력에 사용되는 많은 메소드들이 준비되어 있다.

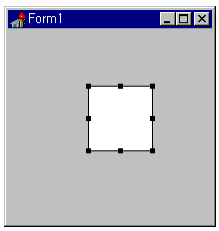



## 가. 셰이프 컴포넌트

폼에 간단한 도형을 배치할 때 사용하는 컴포넌트이다. Additional 페이지의 7번째에  이런 모양으로 위치해 있다. 일단 새로운 프로젝트를 시작하고 폼에 셰이프 컴포넌트를 하나 배치해 보자. 흰색 모양의 직사각형이 폼에 나타날 것이다.

**그림**

셰이프 컴포넌트를 폼에 처음 배치한 모양




위 그림과 같이 직사각형의 도형을 폼에 배치하는 것이 셰이프 컴포넌트의 주된 용도이다. 도형을 출력하여 간단한 장식에 사용될 뿐이며 다른 특별한 용도는 거의 없다. 셰이프의 모양을 결정하는 주요 속성은 다음과 같다.

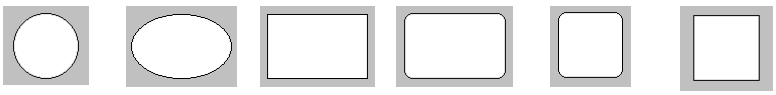
### ■ Shape 속성

셰이프 컴포넌트가 어떤 도형을 출력하는가는 Shape 속성으로 지정하며 이

속성값에 따른 도형의 모양은 다음과 같다. 크기는 마우스로 자유자재로 변경시킬 수 있다.

**표**

Shape 속성으로 여러가지 도형 중 하나를 선택한다.



| stCircle | stEllipse | seRectangle | stRoundRect | stRoundSquare | stSquare |
|---|---|---|---|---|---|
| 원 | 타원 | 직사각형 | 둥근 직사각형 | 둥근 정사각형 | 정사각형 |

원, 사각형 등의 기본적인 도형들이다. 디자인시뿐만 아니라 실행중에도 Shape 속성을 변경할 수 있다.

## ■ Pen 속성

펜은 선을 그을 때 사용하며 어떤 종류의 펜을 사용하는가에 따라 도형의 선 모양이 바뀐다. Pen 속성은 다음과 같은 세부 속성을 가지고 있다. 오브젝트 인스펙터에서 Pen 속성을 더블클릭하여 세부 속성을 조절하도록 한다.

### ✍ Color

펜의 색상을 설정하며 이는 곧 어떤 색으로 선을 그을 것인가를 지정하는 것이다. 디폴트 색상은 검정색(clBlack)이지만 이 속성을 clRed나 clGreen으로 바꾸면 셰이프의 외곽선 색상이 변한다. 오브젝트 인스펙터에서 이 속성을 클릭하면 선택할 수 있는 색상의 목록이 열리므로 원하는 색상을 쉽게 선택할 수 있다.

### ✍ Mode

선을 어떻게 그을 것인가의 방법을 지정하며 펜 색상과 배경 색상의 조합방식을 결정한다.

**표**

펜의 Mode 속성

| 속성 | 의미 |
|---|---|
| pmBlack | 항상 검정색 |
| pmWhite | 항상 흰색 |
| pmCopy | Color 속성에서 지정한 색으로 그려진다. |
| pmMerge | 펜 색상과 배경 색상이 섞인다. |

| | |
|---|---|
| pmNot | 배경 색상을 반전시킨다. |
| pmXor | 펜 색상과 배경 색상을 배타적으로 합한다. |

선을 긋는 동작이 단순한 출력일 수도 있고 원래의 배경색과 조합일 수도 있다. pmCopy 모드를 사용하면 펜의 색상으로 배경을 덮어버리므로 원래의 배경색상이 지워지지만 pmMerge를 사용하면 이미지가 마치 셀로판지에 그려있는 것처럼 살짝 덮히는 효과가 나타나며, pmXor을 사용하면 이미지가 있는 부분이 반전되어 나타난다. 특별한 경우가 아니라면 pmCopy를 사용하는 경우가 가장 많다.

### ✍ Style

펜이 긋는 선의 모양을 설정한다. 실선, 점선 등 다양한 종류가 있다.

**표** 펜의 Style 속성에 따른 선 모양

| 속성 | 설명 | 선 |
|---|---|---|
| psSolid | 실선 | ———— |
| psDash | 굵은 점선 | – — — — — |
| psDot | 가는 점선 | ---------- |
| psDashDot | 일점 쇄선 | – · — · — · — · |
| psDashDotDot | 이점 쇄선 | ··—··—·—··—·· |
| psClear | 투명한 선 | 안보임 |
| psInsideFrame | 외곽에 밀착한 실선 | ———— |

### ✍ Width

선의 굵기를 지정한다. 디폴트는 1이며 가는 선으로 출력되지만 이 속성값을 늘리면 더 굵은 선이 된다. 단 Width를 1이상으로 설정하면 Style 속성은 무시되며 실선만 나타난다. 그래서 굵기 2의 점선은 표현할 수 없다.

## ■ Brush 속성

셰이프 내부의 채움 속성을 지정한다. Pen은 외곽선 속성을 지정하는 반면 Brush는 내부 면의 속성을 지정한다. 두 개의 세부 속성이 있다.

### Color

면의 색상을 지정한다. 사용할 수 있는 색상값은 펜의 경우와 동일하다.

### Style

면의 모양을 지정한다. 면을 꽉 채울 것인가 무늬를 넣을 것인가를 지정한다. 이 속성값에 따른 무늬는 다음과 같다.

**표** 브러시의 스타일

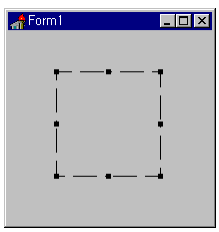
| 속성 | 무늬 | 속성 | 무늬 |
|---|---|---|---|
| bsSolid | | bsCross | |
| bsClear | | bsDiagCross | |
| bsBDiagonal | | bsHorizontal | |
| bsFDiagonal | | bsVertical | |

셰이프 컴포넌트는 주로 폼을 예쁘게 장식하는 용도로 사용되기 때문에 코드를 작성할 필요는 거의 없다. 그래서 drag & drop과 마우스에 관계된 몇 가지 이벤트를 가지기는 하지만 OnClick 이벤트는 가지지 않는다.

# 나. 이미지 컴포넌트

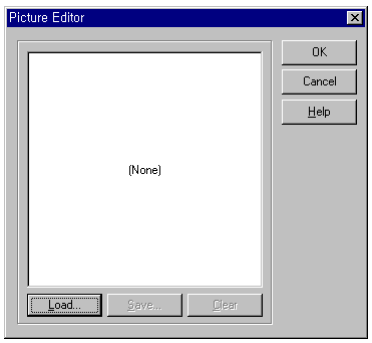
셰이프 컴포넌트가 제한된 몇 가지 도형만을 표현하는데 비해 이미지 컴포넌트는 그래픽 파일을 읽어들일 수 있기 때문에 거의 제한없이 그래픽을 표현할 수 있다. 이미지 컴포넌트를 배치하면 폼에 점선으로 된 사각형이 나타난다.

**그림** 이미지 컴포넌트를 폼에 처음 배치한 모양



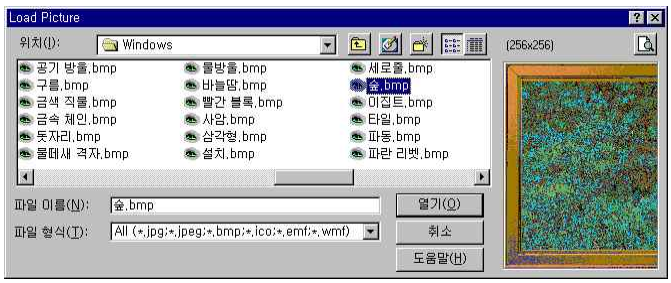

이 사각형은 디자인시에 이미지 컴포넌트의 크기와 위치를 조절하기 위해 보여주는 것이므로 실행중에는 보이지 않으며 다만 이미지 컴포넌트가 가지는 그

래픽만 보인다. 이미지 컴포넌트에 그래픽을 배치하려면 Picture 속성에 그래픽 파일을 지정한다. 오브젝트 인스펙터에서 Picture 속성을 더블클릭하면 그래픽 파일을 읽어올 수 있는 그림 편집기 화면을 보여준다. 아니면 폼에서 이미지 컴포넌트를 직접 더블클릭해도 된다. 다른 컴포넌트는 더블클릭하면 이벤트 핸들러를 작성할 수 있도록 코드 에디터를 열어 주지만 이미지 컴포넌트는 특이하게 그림을 불러 올 수 있는 대화상자를 보여준다.




대화상자 중앙에 현재 설정된 그림이 나타나는데 아직 선택된 그림이 없다. 이 대화상자에서 Load 버튼을 누르면 파일 열기 대화상자와 비슷한 모양의 Load Picture 대화상자를 보여준다. 이 대화상자의 오른쪽에는 미리 보기창이 있어 그림을 쉽게 선택할 수 있도록 해준다.

**그림**
미리 보기 대화상자




이미지 컴포넌트에 읽어올 수 있는 그림은 JPG, BMP, WMF, ICO 가 있고 확장된 메타 파일인 EMF(Enhanced Meta File)도 읽을 수 있다.  JPG 파일을 읽으려면 uses 절에 JPEG 유닛을 적어 주어야 한다.

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

Windows 디렉토리의 숲.bmp 파일 또는 구하기 쉬운 아무 그래픽 파일이나 읽어와 보자. 그림이 이미지 컴포넌트보다 더 커서 일부가 가려져 있다면 이미지 컴포넌트의 크기를 그림의 크기에 맞게 늘려주면 된다. 그래픽을 배치한 후의 폼은 다음과 같다.

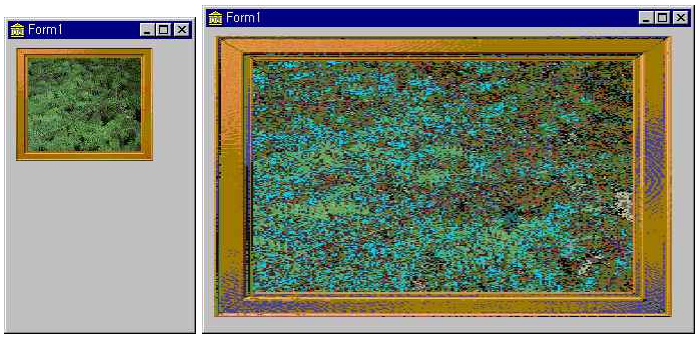

## ■ 이미지 컴포넌트의 속성

### AutoSize

이미지 컴포넌트의 크기를 그래픽의 크기에 맞게 자동 조절한다. 이 속성을 True로 설정하면 이미지 컴포넌트와 그래픽의 크기가 같으므로 가려지는 부분 없이 그래픽 전부가 보이게 된다. AutoSize를 True로 설정한 후에도 이미지 컴포넌트의 크기를 변경할 수 있으며 다시 그래픽 크기와 같게 맞추려면 AutoSize를 False로 바꾼 후 다시 True로 바꾸어 준다.

### Stretch

AutoSize와는 반대로 그래픽의 크기를 이미지 컴포넌트의 크기에 맞게 축소 또는 확장한다. 이 속성을 True로 설정하고 이미지 컴포넌트의 크기를 조절하면 이미지의 크기도 같이 조절된다.



JPG, BMP와 WMF는 Stretch할 수 있지만 ICO는 Stretch할 수 없다.

**Center**

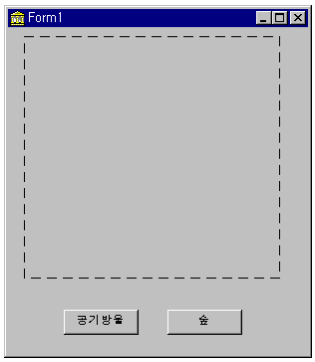
그래픽은 이미지 컴포넌트의 좌상단을 기준으로 놓여진다. Center 속성을 True로 바꾸어 주면 중앙을 기준으로 놓여진다.


이 세 가지 속성을 사용하여 이미지 컴포넌트에 놓이는 그림의 크기와 위치를 조절한다. 이 외에 중요한 속성으로는 이미지 컴포넌트 그 자체의 배치를 조정하는 Align 속성이 있다. 패널과 마찬가지로 폼의 상단(alTop)에 밀착시켜 놓는다거나 폼에 꽉 차도록(alClient) 배치하여 폼 전체에 배경 그림을 넣을 수도 있다.

## ■ 실행중에 그림 바꾸기

8jang
chgbmp

이미지 컴포넌트에 어떤 그림을 놓을 것인가는 Picture 속성으로 지정한다. 이 지정은 디자인시에 가능한 것이고 실행시 그림을 바꾸려면 메소드를 통해야 한다. Picture 오브젝트의 LoadFromFile 메소드를 사용하여 실행중에 그래픽을 읽어오며 인수로 읽어올 그래픽 파일의 이름을 준다. 물론 디렉토리명과 드라이브명도 한꺼번에 지정할 수 있다. 새 폼에 버튼 두 개와 이미지 컴포넌트를 사용하여 실행중에 그래픽을 변경하는 예제를 작성해 보자.




이미지 컴포넌트의 크기를 적당히 조절하고 Stretch 속성을 True로 만들어 이미지가 바뀌어도 보기 싫지 않도록 해준다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 이미지 | Stretch | True |
| 좌측 버튼 | Caption | 공기방울 |
| 우측 버튼 | Caption | 숲 |

두 버튼의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('c:₩windows₩공기 방울.bmp');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('c:₩windows₩숲.bmp');
end;
```

BMP 파일을 인수로 사용한 LoadFromFile 메소드의 호출이다. 만약 윈도우즈가 다른 디렉토리에 설치되어 있다면 디렉토리 이름을 변경해 주도록 하고 그래픽 파일이 없다면 임의의 다른 그래픽 파일을 사용해도 무방하다. 실행중에 버튼을 누르면 그래픽 파일을 읽어와서 그림을 바꾸어 준다.


이때 주의할 점은 그래픽 파일의 관리이다. 델파이는 DLL이나 부가적인 파일이 필요없는 완전한 실행 파일을 만들어 내지만 위의 경우와 같이 프로그램에서 사용하는 파일이 있을 경우 완성작을 배포할 때 같이 배포해야 한다. 그래픽 파일은 실행중에만 사용되므로 실행 파일에는 포함되지 않기 때문이다. 부속 파일은 보통 프로그램 파일과 같은 디렉토리에 위치하므로 실제 읽어올 때는 완전 경로를 주어서는 안된다. 여기서는 예제를 만들기 때문에 완전 경로를 사용했지만 좀 더 완벽한 프로그램이라면 그래픽 파일이 있는 경로를 조사해서 경로를 지정해 주어야 한다.

그러나 디자인중에 읽어들인 그래픽 파일은 폼 파일인 DFM 파일에 포함되며 최종적으로 실행 파일에 직접 포함되므로 같이 배포하지 않아도 된다.

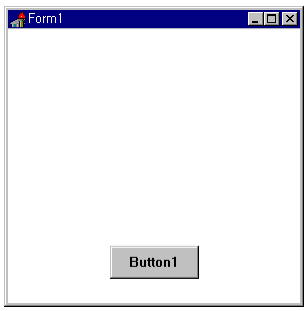
# 8-2 폼에 직접 그리기

셰이프나 이미지 컴포넌트를 사용하여 그래픽을 표현하는 것은 어디까지나 폼을 예쁘게 장식하기 위해서일 뿐이다. 컴포넌트만으로 실행중에 다양한 그래픽을 표현하기에는 부족한 점이 많다. 여기서는 폼에 그림을 직접 그리는 방법에 대해 알아본다.

## 가. 캔버스

델파이로 폼에 그림을 그릴 때 그림이 곧바로 폼에 그려지는 것은 아니며 폼의 속성(오브젝트) 중의 하나인 캔버스(Canvas)에 그려진다. 캔버스는 폼 전체 영역에 걸쳐 있으며 그림이 그려지는 표면이다. 실행중에만 사용할 수 있기 때문에 오브젝트 인스펙터에는 나타나지 않는다. 캔버스는 펜, 브러시, 폰트 등의 다양한 속성을 가지고 있으며 이 속성들을 사용하여 그려지는 선과 채색되는 면, 출력되는 문자의 모양을 조절한다. 델파이가 제공하는 그래픽 함수들은 모두 폼의 메소드가 아니라 캔버스 오브젝트의 메소드이다. 그 이유는 그림이 그려지는 장소가 폼이 아니라 캔버스이기 때문이다.

그래픽 함수들을 직접 사용해 보기 위해 간단한 실습 예제를 준비해 두자. 새로운 프로젝트를 시작하고 그래픽 시작을 명령할 버튼 하나만 배치한다. 폼의 크기는 300*300으로, 또는 그 이상으로 조정하고 그려지는 그래픽이 잘 보이도록 하기 위해 바탕색(Color 속성)을 흰색으로 만들어 놓는다.



# 나. 점

8jang
fdraw1

캔버스에 점을 출력하는 함수는 따로 없다. 캔버스는 그림이 그려지는 널따란 영역이며 마치 모눈종이처럼 수많은 점들로 구성되어 있다. 캔버스의 속성인 Pixels는 캔버스의 모든 점으로 구성된 이차 배열이며 각 점이 어떤 색상을 가지고 있는지를 보관한다. Pixels는 캔버스의 표면과도 같으며 Pixels 배열이 곧 캔버스이다. Pixels 배열의 한 배열 요소에 값을 써 넣으면 점을 찍는 것이고 값을 읽으면 점의 색상을 조사하는 것이다. 캔버스의 100,120 좌표에 빨강색의 점을 찍고 싶으면 Pixels 배열의 100행, 120열 배열 요소에 clRed를 대입하기만 하면 된다.

```
Canvas.Pixels[100,120]:=clRed;
```

Pixels 배열의 배열 첨자가 점을 찍을 좌표가 되며 대입해 주는 값이 점의 색상이다. 점의 색상을 조사할 때는 다음과 같이 한다.

```
nowcolor:=Canvas.Pixels[200,300];
```

배열 요소의 값을 읽는 것이 곧 점의 색상을 읽는 것이다 점찍는 실습을 해보자. 앞에서 만들어 두었던 예제에서 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하고 실행해 보아라.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pixels[100,100]:=clBlack;
end;
```

화면 100,100 좌표에 검정색의 점이 찍힌다. 물론 아주 작은 점이기 때문에 눈을 황소눈만하게 치켜 떠야만 겨우 보일 것이다. 앞으로 그래픽 함수들을 학습하면서 여기(버튼의 OnClick 이벤트 핸들러)에 계속 코드를 추가해 나갈 계획이다. 여러분들도 실습을 할 때 여기에 코드를 작성하면 된다.

# 나. 직선 긋기

이번에는 캔버스에 직선을 그어 보자. 선을 그을 때는 다음 두 함수가 사용된

다.

| 함수 | 기능 |
|---|---|
| MoveTo(X,Y) | 현재 위치를 X,Y로 옮긴다. |
| LineTo(X,Y) | 현재 위치에서 X,Y로 선을 긋는다. |

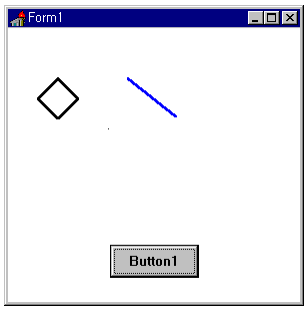
MoveTo로 선의 시작점을 지정하고 LineTo로 선의 끝점을 지정하여 선을 긋는다. 일단 선이 한번 그어지면 LineTo로 계속해서 선을 연결하면서 그릴 수 있다. 델파이는 선을 긋는 펜의 현재 좌표를 항상 기억하고 있다가 다음 LineTo 함수 호출이 있으면 기억하고 있던 현재 위치를 선의 시작 좌표로 사용한다. 또한 선을 긋고난 후에 선의 끝점으로 현재 위치를 옮겨주어 선이 계속 이어질 수 있도록 해준다. 펜의 현재 위치를 설정할 때 사용하는 함수가 MoveTo 함수이며 현재 위치를 알고 싶을 때는 캔버스의 PenPos 속성을 읽으면 된다.

캔버스에 그려지는 선의 모양은 캔버스의 Pen 속성에 영향을 받는다. 펜의 속성은 셰이프 컴포넌트의 Pen 속성과 동일하며 Color로 색상을 설정하고 Width로 폭을 조정한다. 점을 찍는 코드 다음에 직선을 긋는 코드와 펜의 속성을 설정하는 코드를 추가해 보면 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pixels[100,100]:=clBlack;
  Canvas.Pen.Color:=clBlue;
  Canvas.Pen.Width:=2;
  Canvas.MoveTo(120,50);
  Canvas.LineTo(168,88);
  Canvas.PolyLine([Point(50,50),Point(70,70),Point(50,90),
    Point(30,70),Point(50,50)]);
end;
```

선을 그리기 전에 어떤 선을 그을 것인가 펜의 속성부터 정의해 주어야 한다. 선의 굵기를 2로, 색상은 파랑색으로 설정한 후 MoveTo와 LineTo로 (120,50)에서 (168,88)까지 선을 그었다.

PolyLine 함수는 선을 연속해서 긋는다. [ ] 안에 점배열을 넘겨주어 다각형을 만든다. 위 예에서는 마름모를 이루는 4개의 점을 연결하였다. PolyLine 함수의 자세한 사용법은 별도로 설명하지 않아도 소스를 보면 알 수 있을 것이다. 실행 결과는 다음과 같다.

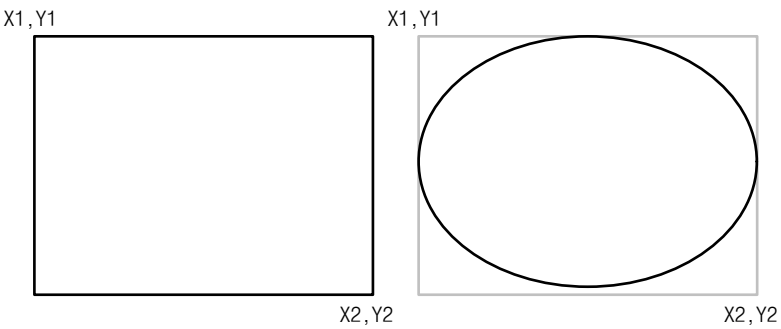



# 라. 도형 그리기

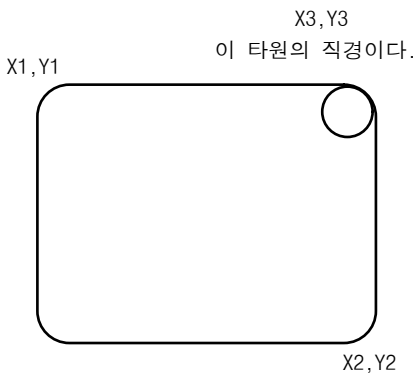
도형을 그리는 함수는 여러 가지가 있지만 일단 기본적으로 다음 세 가지를 알아보자.

| 함수 | 기능 |
|---|---|
| Rectangle(X1,Y1,X2,Y2) | 직 사각형 |
| Ellipse(X1,Y1,X2,Y2) | 타원 |
| RoundRect(X1,Y1,X2,Y2,X3,Y3) | 모서리가 둥근 사각형 |

X1,Y1이 좌상단의 좌표이며 X2,Y2가 우하단의 좌표이며 이 두 점을 대각선으로 하는 직사각형이 그려진다. 타원의 경우 지정한 좌표의 직사각형에 내접하는 타원이 그려진다.




RoundRect의 X3,Y3는 모서리에 놓일 타원의 지름을 지정한다. 모서리에 놓이는 타원의 지름이 클수록 사각형의 모서리가 둥글둥글해질 것이다.

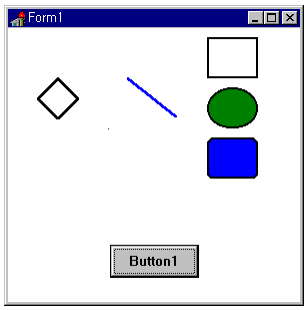



선이 펜의 영향을 받듯이 면은 브러시의 영향을 받는다. 브러시가 가지는 속성대로 면이 그려지며 따라서 면을 그리기 전에 브러시의 속성을 먼저 설정해야 한다. 브러시는 셰이프 컴포넌트에서 설명한 것과 같이 Style과 Color 속성을 가진다. 이 속성들을 사용하여 면을 그려보자. 버튼의 OnClick 이벤트 핸들러에 다음과 같이 코드를 추가한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pixels[100,100]:=clBlack;
  Canvas.Pen.Color:=clBlue;
  Canvas.Pen.Width:=2;
  Canvas.MoveTo(120,50);
  Canvas.LineTo(168,88);
  Canvas.Pen.Color:=clBlack;
  Canvas.Pen.Width:=2;
  Canvas.PolyLine([Point(50,50),Point(70,70),Point(50,90),
    Point(30,70),Point(50,50)]);
  Canvas.Rectangle(200,10,250,50);
  Canvas.Brush.Color:=clGreen;
  Canvas.Ellipse(200,60,250,100);
  Canvas.Brush.Style:=bsDiagCross;
  Canvas.Brush.Color:=clBlue;
  Canvas.RoundRect(200,110,250,150,10,10);
  Canvas.Brush.Style:=bsSolid;
  Canvas.Brush.Color:=clWhite;
  Canvas.Pen.Width:=1;
end;
```

코드의 마지막 부분에서 펜과 브러시의 속성을 디폴트로 다시 바꾸어 주고 있

다. 이 과정을 생략해 버리면 다음에 그림을 그릴 때 여기서 지정한 펜과 브러시가 그대로 사용되므로 뜻하지 않은 결과를 초래할 수도 있다. 여기까지 작성한 코드를 실행하면 다음과 같은 그림이 그려질 것이다.




이게 웬 인상파 화가의 그림이냐고 할지 모르겠지만 점, 선, 사각, 원의 기본 도형을 잘 조합하면 웬만한 그림은 다 그릴 수 있을 것이다. 필자는 원래 초등학교 2학년 때부터 그림과는 담쌓고 살아서 이런 그림밖에 그릴 수가 없다. 도형을 그리는 함수는 이 외에도 Pie, Arc, Chord, FloodFill 등의 함수가 있다. 이 함수들은 원호나 부채꼴을 그릴 때 사용되는데, 인수가 조금 다르기는 하지만 원이나 사각형과 크게 틀리지 않으므로 설명은 생략한다. 부록으로 제공되는 컴포넌트 레퍼런스의 Canvas 오브젝트에 관한 설명이나 델파이가 제공하는 도움말을 참조하기 바란다.

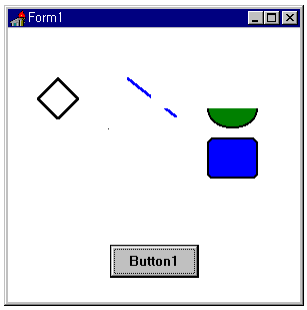
# 마. OnPaint 이벤트


8jang
fdraw2

앞에서 만들어 본 그래픽 예제는 아주 심각한 문제를 하나 가지고 있다. 그림의 내용이야 어쨌건 그림을 그리는 데는 분명히 이상이 없지만 그려진 그림을 계속 유지하는 데 문제가 있다. 도스에서는 화면에 한번 그려 놓으면 일부러 지우지 않는 한은 그림이 화면에 그대로 있지만 윈도우즈에서는 여러 개의 프로그램이 한꺼번에 실행되기 때문에 프로그램이 일부러 지우지 않아도 다른 프로그램에 의해 지워지기도 한다.

화면이 겹쳐져서 가려졌다가 다시 나타난다거나 윈도우가 스크린 밖을 잠시 벗어났다가 다시 스크린 안으로 들어왔을 때, 또는 윈도우의 크기가 조정되었을 때 그려진 그림이 지워진다. 실제로 과연 그런가 직접 살펴 보아라. 위의 예제를 실행시키고 다른 윈도우로 일부를 살짝 가린 후 다시 윈도우를 나타나도록 해보면 가려졌던 부분이 지워지고 없다.

그림
폼의 일부가 다른 윈도우에 의해 지워진 모양




이럴 때는 어떻게 해야 하는가 하면 다시 그려야 한다. 버튼을 눌러 다시 그리기만 하면 문제가 깔끔히 해결된다. 그럼 이렇게 지워질 때마다 계속 버튼을 눌러주어야 하는가 하면 그렇지는 않다. 윈도우즈는 모든 윈도우들을 관리하면서 일부가 지워져 다시 그려져야 할 필요가 있을 때 이 사실을 해당 윈도우에게 통보해 준다.

어떻게 통보해 주는가 하면 윈도우에게 다시 그려져야 함을 알리는 메시지를 보내주며 델파이에서 이 메시지(WM_PAINT)는 이벤트 형태로 나타난다. 폼이 다시 그려져야 할 필요가 있을 때 발생하는 이벤트가 OnPaint 이벤트이다. 윈도우즈는 가려졌다가 다시 나타나는 윈도우를 복구해주는 책임을 지지는 않지만 대신 어떤 윈도우가 다시 그려져야 하는지를 철저히 알려준다.

그래서 폼에 그림을 그리는 코드는 버튼의 OnClick 이벤트에 작성해서는 안되며 폼의 OnPaint 이벤트로 옮겨 주어야 한다. 그럼 코드를 블럭으로 싸서 잘라낸 후 폼의 OnPaint 이벤트로 옮겨 보자.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
with Canvas do
  begin
  Pixels[100,100]:=clBlack;
  Pen.Color:=clBlue;
  Pen.Width:=2;
  MoveTo(120,50);
  LineTo(168,88);
  Pen.Color:=clBlack;
  Pen.Width:=2;
  PolyLine([Point(50,50),Point(70,70),Point(50,90),
    Point(30,70),Point(50,50)]);
  Rectangle(200,10,250,50);
  Brush.Color:=clGreen;
  Ellipse(200,60,250,100);
```

```
  Brush.Style:=bsDiagCross;
  Brush.Color:=clBlue;
  RoundRect(200,110,250,150,10,10);
  Brush.Style:=bsSolid;
  Brush.Color:=clWhite;
  end;
end;
```

이왕 옮기는 김에 좀 더 깔끔한 코드를 작성하기 위해 with Canvas를 밖으로 묶어 내고 들여쓰기를 하였다. 그래픽 함수가 모두 Canvas의 메소드 또는 속성이므로 그래픽을 할 때는 with Canvas do begin ~ end; 사이에 코드를 배치하는 것이 더 간편하다.

이렇게 OnPaint로 코드를 옮겨 놓으면 그림이 가려졌다 다시 나타나더라도 매번 OnPaint 이벤트가 발생하고 이벤트가 발생할 때마다 그림을 다시 그림으로써 어떤 경우에라도 그림이 지워지지 않게 된다.

윈도우즈는 운영체제 차원에서 윈도우에 그려진 그림을 보호해 주지 않는다. 수십 개의 윈도우가 열리며 이중 삼중으로 겹쳐지는 윈도우의 그림을 일일이 다 보호해 주기가 너무 어렵기 때문이다. 물론 꼭 필요하다면 그렇게 할 수도 있다. 하지만 운영체제에서 각 프로그램이 그려놓은 그림까지 다 책임지고자 한다면 속도의 감소와 메모리의 낭비라는 막대한 반대급부가 생기므로 그림의 보존 책임을 운영체제가 맡지 않고 개별 프로그램이 맡는다. 대신 운영체제는 언제 그림이 다시 그려져야 하는지를 개별 프로그램에 철저히 통보해 줌으로써 프로그램이 그림을 항상 보존할 수 있도록 도움을 주고 있다.

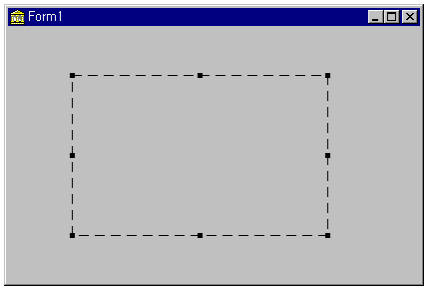
윈도우즈가 프로그램에 보내주는 이런 통보 정보를 메시지라고 하며 프로그램들은 윈도우즈가 보내주는 메시지를 참조하여 다른 프로그램과 완벽한 조화를 이루며 실행된다. 그래서 윈도우즈를 메시지 구동 시스템이라고 하는 것이다.

# 바. 페인트 박스

8jang
PaintBox

앞에서 보았다시피 폼에는 직접 그림을 그릴 수 있는데 이 때는 폼의 세부 속성인 Canvas 오브젝트를 사용하였다. 폼 외에 Canvas 를 제공하는 컴포넌트로는 페인트 박스가 있다. 폼은 여러 가지 컴포넌트를 포함하는 컨테이너로서 동작하는데 여기에 직접 그림을 그리는 것은 사실 약간 부담스러우며 별로 바람직

하지도 않다. 작도 함수로 직접 그림을 그려야 할 필요가 있을 때 폼에 직접 그리는 것보다는 페인트 박스 컴포넌트에 그리는 것이 더 좋다. 페인트 박스 컴포넌트는 System 팔레트의 두 번째에 있다. 페인트 박스를 폼에 배치하면 다음과 같은 사각형만 나타난다.




이 페인트 박스의 안쪽은 모두 Canvas이며 그대로 그리기 표면이 된다. 이 오브젝트를 사용하면 폼에 그림을 그리듯이 페인트 박스에 그림을 그릴 수 있다. 페인트 박스의 OnPaint에 그리기 코드를 작성해 보자.

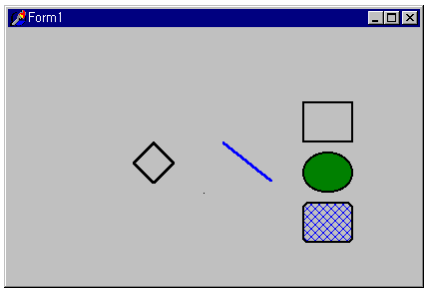
```
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
with PaintBox1.Canvas do
 begin
 Pixels[100,100]:=clBlack;
 Pen.Color:=clBlue;
 Pen.Width:=2;
 MoveTo(120,50);
 LineTo(168,88);
 Pen.Color:=clBlack;
 Pen.Width:=2;
 PolyLine([Point(50,50),Point(70,70),Point(50,90),
  Point(30,70),Point(50,50)]);
 Rectangle(200,10,250,50);
 Brush.Color:=clGreen;
 Ellipse(200,60,250,100);
 Brush.Style:=bsDiagCross;
 Brush.Color:=clBlue;
 RoundRect(200,110,250,150,10,10);
 Brush.Style:=bsSolid;
 Brush.Color:=clWhite;
 end;
end;
```

폼에 직접 그리기를 할 때와 동일한 코드이되 with 문의 Canvas 가 Paint1.Canvas 로 변경되었다. 즉 폼의 캔버스에 그리는 것이 아니라 페인트 박스의 캔버스에 그리기를 하였다. 이렇게 별도의 캔버스를 사용하면 어떤 점이 좋으며 왜 이런 컴포넌트가 필요할까.

우선은 폼의 그리기 코드가 페인트 박스로 옮겨지므로 폼의 부담이 적어져서 좋다. 안그래도 폼은 이런 저런 컴포넌트를 관리하는 일로 바쁜데 그리기까지 하자면 더 느려지고 난잡해지기 때문이다.

두 번째로 그리기에 사용되는 표면을 최소한으로 유지할 수 있어 페인팅 속도가 빨라진다. 폼에 직접 그리기 하면 폼 전체가 다시 그려져야 하므로 느리지만 페인트 박스에 그리기를 하면 페인트 박스만 그리면 되므로 더 빠르다.

세 번째로 페인트 박스의 캔버스에 그린 그림은 페인트 박스의 좌표를 사용하므로 그림 전체를 폼의 다른 위치로 옮기기가 편리해진다. 그리기 함수의 인수를 직접 변경하지 않아도 페인트 박스만 옮겨주면 되기 때문이다. 페인트 박스를 이동하면 다음과 같이 그림도 통째로 이동된다.



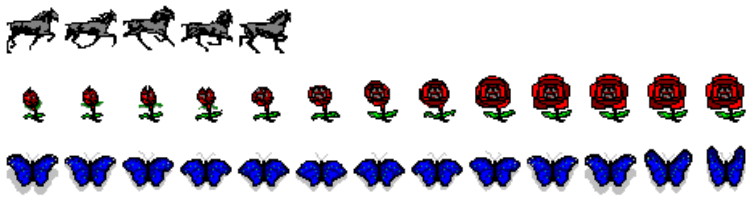

페인트 박스는 폼의 기능중에 그리기만 담당하는 부분을 따로 떼어 놓은 컴포넌트라고 할 수 있다. 그런데 델파이와 같은 비주얼 툴은 폼에 직접 그리기를 할 경우가 극히 드물기 때문에 사실 페인트 박스를 사용할 일은 그다지 많지 않다.

# 8-3 애니메이션

애니메이션(Animation)이란 그림을 움직이는 기법을 말한다. 델파이에서 애니메이션을 구현하려면 이미지 컴포넌트를 사용해야 하며 다음 두 가지 준비가 필요하다.

첫째, 일단 애니메이션을 구현하기 위해 여러 컷의 그림을 만들어 두어야 한다. 움직이는 그림의 각 장면에 해당하는 그림을 계속 교체하여 애니메이션을 구현함으로써 연속적인 동작을 표현하며 이어지는 그림이 된다. 애니메이션에 사용되는 그림은 모두 같은 크기와 같은 색상수를 사용해야 하며 그림이 많을수록 섬세한 애니메이션을 만들 수 있다. 말이 달려가는 애니메이션, 장미꽃이 피는 애니메이션, 나비가 날아가는 애니메이션을 만들려면 다음과 같은 연속적인 그림들이 필요하다.

그림
애니메이션에 사용되는 그래픽



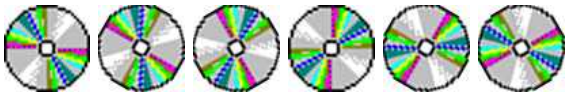
이 그림들을 순서대로 바꾸어 가며 보여주면 애니메이션이 된다. 애니메이션의 방법은 이렇게 간단하지만 이런 그림을 예쁘게 만드는 일은 쉬운 일이 아니다.

둘째, 연속적인 그림을 주기적으로 바꾸어 주어야 한다. 주기적으로 어떤 동작을 하기 위해서는 타이머 컴포넌트를 배치하고 타이머 이벤트에서 애니메이션을 실행하는 것이 가장 효율적이다. 타이머 이벤트의 호출 주기를 설정하는 타이머의 Interval 속성에 따라 애니메이션의 속도가 결정된다.

## 가. 파일에서 읽어오는 방법

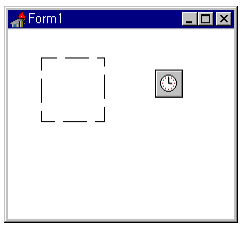

8jang
anim1

애니메이션 실습에 사용할 그림은 다음 여섯 장이다. CD가 회전하는 모양을 6컷의 그림으로 준비해 보았다.



비슷비슷하게 생긴 것 같지만 자세히 살펴보면 CD의 각도가 조금씩 다르다. C-1.BMP에서부터 C-6.BMP까지의 파일 이름을 가지며 크기는 모두 64*64이다. 이 파일들을 같은 디렉토리 안에 두고 애니메이션 주기마다 그림을 교체해 주면 된다.

실습을 위해 폼에 이미지 컴포넌트 하나와 타이머를 배치해 둔다. 폼의 배경색은 흰색으로 바꾸어 잘 보이도록 하고 이미지의 AutoSize 속성을 True로 바꾼다. 타이머의 Interval 속성을 100으로 설정하여 1초에 10번씩 애니메이션이 발생하도록 한다.




애니메이션 실행을 위해서는 현재의 컷이 몇 번째 컷인가를 기억하는 변수가 하나 필요하다. 매 번 타이머 이벤트가 발생할 때마다 이 값을 참조해야 하며 값을 계속 유지해야 하므로 전역 변수로 선언한다. 변수의 이름을 animation으로 설정하고 정수형으로 선언한다.

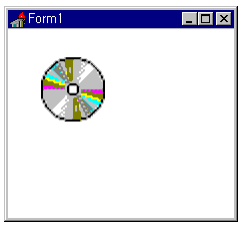
```
implementation
var
 animation:integer;
```

OnTimer 이벤트에서는 실제로 애니메이션을 실행하는 코드를 배치한다. animation 변수의 값을 참조하여 현재 컷 바로 다음 컷의 그림을 이미지 컴포넌트의 Picture 속성에 대입해 주면 된다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
animation:=animation+1;
if animation=7 then animation:=1;
if animation=1 then image1.picture.LoadFromFile('c-1.bmp');
if animation=2 then image1.picture.LoadFromFile('c-2.bmp');
if animation=3 then image1.picture.LoadFromFile('c-3.bmp');
```

```
if animation=4 then image1.picture.LoadFromFile('c-4.bmp');
if animation=5 then image1.picture.LoadFromFile('c-5.bmp');
if animation=6 then image1.picture.LoadFromFile('c-6.bmp');
end;
```

타이머 이벤트가 한번 호출될 때마다 animation 변수는 계속 1씩 증가하며 6다음에는 다시 1로 돌아온다. animation 변수값에 따라 적당한 그림 파일을 읽어와 이미지 컴포넌트에 출력해 주면 애니메이션이 완성된다. 그림 파일을 읽어올 때는 LoadFromFile 메소드를 사용했다. 실행중의 모습은 다음과 같다.




물론 위의 그림과 같이 정지해 있지는 않고 CD가 계속 회전한다. 타이머 이벤트로 주기적으로 동작하므로 일정한 속도를 유지하며 심지어 포커스를 가지고 있지 않아도 CD는 계속해서 돌아간다.

# 나. 미리 읽어 놓는 방법

앞에서 만들어 본 애니메이션 예제는 큰 문제를 하나 가지고 있다. 그림 파일이 디스크에 있기 때문에 애니메이션을 한 번 수행할 때마다 계속 디스크를 읽어야 한다는 문제이다. 알다시피 프로그램 실행중에 하드 디스크를 액세스한다는 것은 무지막지한 실행속도의 감소를 초래한다. 멋있는 장식 한번 넣어 보려다가 프로그램이 기어가는 수가 있다.

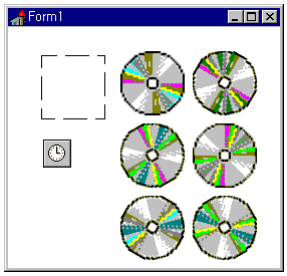
아마 여러분의 프로그램에서 위의 예제를 실행시키면 일단은 비교적 부드럽게 돌아갈 것이며 하드 디스크를 계속해서 읽어대지는 않을 것이다. 언뜻 보기에는 이 문제가 그리 큰 문제같아 보이지 않는다. 디스크를 1초에 열번씩 읽고도 부드럽게 실행되는 이유는 캐시 프로그램이 한번 읽은 그림 파일을 다시 읽지 않도록 해주기 때문이다. 만약 이 프로그램을 캐시가 설치되지 않은 시스템에서 실행하면 그야말로 하드 디스크를 벅벅 긁어대면서 실행될 것이다.

그래서 뭔가 좀 더 효율적인 방법을 찾아보아야 하는데 어떤 방법인가 하면 실행 파일에 미리 필요한 그림 파일을 모두 포함시키는 것이다. 이미지 컴포넌


8jang
anim2

트를 미리 만들어 두고 필요한 그림을 모두 출력시켜 놓은 후 이미지 컴포넌트끼리 그림 파일을 교체하면 된다. 실습을 해보자. 폼에 이미지 컴포넌트를 Image1~Image7까지 모두 7개를 배치해 놓고 Image2~Image7까지 애니메이션의 각 컷을 미리 출력해 놓는다.

**그림**
애니메이션 컷을 미리 읽어 놓은 모양




그리고 타이머 이벤트에서는 다음과 같이 코드를 작성한다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
animation:=animation+1;
if animation=7 then animation:=1;
if animation=1 then image1.picture:=image2.picture;
if animation=2 then image1.picture:=image3.picture;
if animation=3 then image1.picture:=image4.picture;
if animation=4 then image1.picture:=image5.picture;
if animation=5 then image1.picture:=image6.picture;
if animation=6 then image1.picture:=image7.picture;
end;
```

애니메이션이 발생할 때마다 필요한 그림을 담고 있는 이미지 컴포넌트의 Picture 속성을 직접 대입한다. 디자인시에 이미 다른 이미지 컴포넌트에 읽어 놓은 그림들이므로 하드 디스크에서 그림 파일을 읽어올 필요가 없어진다.

그럼 Image2~Image7까지의 이미지 컴포넌트는 어떻게 처리하는가? 괜히 쓸데없이 자리를 차지하고 있어 보기 싫을 것 같다. 이럴 때 사용하는 속성이 Visible 속성이며 이 속성을 False로 설정하여 실행중에는 숨겨 버리면 된다. 결국 보이는 것은 Image1뿐이며 그림은 계속 교체되고 애니메이션이 실행된다. 실행중의 모습은 앞의 경우와 완전히 같다. 이 방법이 뭔가 비정상적이고 대충 끼워 맞추는 듯한 느낌이 들겠지만 간단한 애니메이션이라면 이런 방법을 쓰는 것도 그리 나쁘지는 않으며 실전에서 자주 사용되는 방법이다. 참고로 여기서 사용하지는 않았지만 그래픽 파일을 실행 파일에 포함시키는 가장 합리적인 방법은 리소스 파일에 그래픽을 포함시키는 것이다.

# 다. 가상 화면

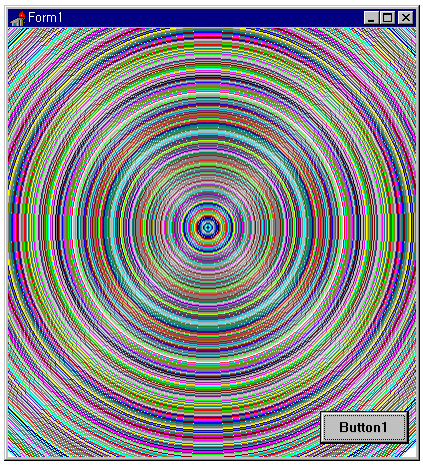
애니메이션을 제대로 구현하려면 가상 화면에 대한 논의를 빼 놓을 수 없다. 각 장면을 순서대로 바꾸어 주어야 하는 애니메이션 기법에서 가장 문제가 되는 것은 장면이 바뀔 때의 화면 깜박거림을 최소화하는 것이다. 컴퓨터의 속도가 무한정 빠르다면 몰라도 현실은 그렇지가 못해서 장면이 바뀔 때의 깜박거림이 사람의 눈에 보이게 된다. 더구나 그래픽을 교체하는 작업은 대규모의 데이터 이동이기 때문에 특히 심하다.

이런 깜박거림을 해결하려면 가상 화면을 사용해야 한다. 초보자에게는 당장 불필요할 지 모르겠지만 그래픽을 처리하는 게임을 만든다든가 그래픽 컴포넌트를 만들 때는 반드시 필요하며 소스 분석중에 심심치 않게 가상 화면을 사용하는 것을 볼 수 있을 것이다. 가상 화면이란 실제의 화면은 아니지만 메모리에 구현된 논리적인 화면이다. 화면에 곧바로 그림을 그리지 않고 가상 화면에서 미리 그림을 그린 후에 실제 화면으로 고속 전송하면 그림을 그리는 과정이 사용자의 눈에는 보이지 않게 된다.

## ■ 가상 화면을 사용하지 않는 경우

8jang
vscr1

가상 화면의 개념적인 면들을 살펴보기 위해 두 개의 예제를 만들어 보았다. 첫 번째 예제인 Vscr1.dpr은 버튼을 누르면 색색의 동심원 300개를 화면에 그린다.




동심원을 그리는 코드는 버튼의 OnClick 이벤트에 다음과 같이 작성되어 있다. 난수 함수와 단순한 그래픽 명령을 사용한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i:integer;
 j:integer;
begin
randomize;
Canvas.Brush.Style:=bsClear;
for i:=1 to 300 do
 begin
 Canvas.Pen.Color:=
  RGB(random(255),random(255),random(255));
 Canvas.Ellipse(200+i,200+i,200-i,200-i);
 end;
end;
```

이 코드는 과연 제대로 300개의 동심원을 그려 내기는 한다. 그러나 그리는 과정이 워낙 느리기 때문에 사용자가 직접 그리는 과정을 확인할 수가 있으며 때로는 이런 모습이 사용자에게 별로 깔끔한 느낌을 주지 못한다. 시간이 걸리는 일이야 어쩔 수 없다 하더라도 다 그린 후에 보여주는 것이 더 깔끔한 그래픽 처리가 아니겠는가?

## ■ 가상 화면을 사용하는 경우

8jang
vscr2

그래서 이 예제를 가상 화면을 사용하여 가상 화면에 미리 그린 후에 한꺼번에 보여주도록 Vscr2.dpr을 작성하였다. 실행 결과는 물론 앞에서 만든 예제와 동일하지만 버튼을 누르면 잠시 기다렸다가 한꺼번에 결과를 보여준다. 버튼의 OnClick 이벤트는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i:integer;
 j:integer;
 vbitmap:TBitmap;
begin
randomize;
vbitmap:=TBitmap.Create; {가상화면 만듬}
vbitmap.Width:=400;
vbitmap.Height:=400;
vbitmap.Canvas.Brush.Style:=bsClear;
for i:=1 to 300 do
 begin
```

```
    vbitmap.Canvas.Pen.Color:=
      RGB(random(255),random(255),random(255));
    vbitmap.Canvas.Ellipse(200+i,200+i,200-i,200-i);
    end;
  Canvas.Draw(0,0,vbitmap);
  vbitmap.Free;
  end;
```

먼저 화면의 크기와 동일한 비트맵 오브젝트를 하나 만든다. 비트맵은 화면과 마찬가지로 Canvas 오브젝트를 가지며 그 표면에 그림을 그릴 수 있는데, 이런 용도로 사용되는 비트맵을 off screen bitmap이라 한다. 그래서 화면에 직접 출력하지 않고 비트맵의 표면에 그림을 그린 후 그림이 완성되면 비트맵을 폼의 캔버스로 고속 전송하는 것이다. 사용자의 눈에는 그림이 그려지는 과정이 보이지 않으며 그린 후의 결과만 보이게 된다. 비트맵을 사용하여 가상 화면 처리를 하는 과정을 요약하면 다음과 같다.

```
var VS:TBitmap;            {비트맵 선언}
VS:=TBitmap.Create;        {비트맵 생성}
...                        {비트맵에 그림을 그린다.}
Canvas.Draw(0,0,VS);       {폼으로 복사}
VS.Free;                   {비트맵 해제}
```

비트맵에 그림을 그리는 방법은 비트맵의 Canvas 오브젝트를 사용하므로 폼의 Canvas 오브젝트를 사용하는 방법과 동일하다. 즉 폼에 직접 그릴 수 있는 그림이라면 비트맵에도 같은 그림을 모두 그릴 수 있다는 얘기다.

여기서 보인 예제는 가상 화면의 개념만을 설명하기 위해 제작한 것이므로 그림을 그리는 속도 차는 사실 별로 느낄 수가 없다. 하지만 실전에서 사용될 때는 이보다 훨씬 더 복잡한 형태를 띠며 더 고도의 기법을 사용하여 속도를 증가시킨다. 예를 들어 백그라운드에서 미리 가상 화면에 비트맵을 작성해 놓는다든지, 컴퓨터가 놀 때 틈틈이 그림을 그린다든지, 변화된 일부분만 고속 복사하는 등의 기법이 쓰이고 있다. 아무튼 그런 고급 기법은 소스를 통해 직접 익히기 바라며 여기서는 개념만 소중히 익히도록 하자.

## 라. 애니메이트

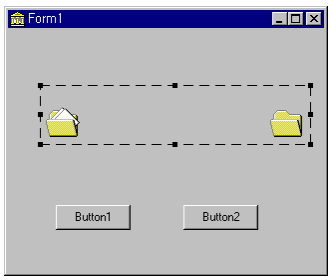
8jang
Anim3

애니메이션은 사실 원리상으로 전혀 복잡한 것이 아니다. 연속적인 그림을 일

정한 시간 간격으로 계속 교체해 주기만 하면 되므로 이미지 컴포넌트만 사용해도 애니메이션을 간단하게 구현할 수 있다. 이런 원론적인 방법외에 애니메이션을 위한 별도의 컴포넌트를 사용하는 방법도 있다. 윈도우즈 98은 애니메이션을 위해 애니메이트라는 컨트롤을 제공하며 델파이는 이 컨트롤을 캡슐화한 애니메이트 컴포넌트를 제공한다.

이 컴포넌트는 이름 그대로 애니메이션을 위한 전용 컴포넌트이며 미리 준비된 AVI 파일을 백그라운드에서 재생해 준다. Win32 컨트롤이므로 Win32 페이지에 이 컴포넌트가 있다. 예제 하나를 만들면서 이 컴포넌트를 사용해 보도록 하자.

새 프로젝트를 시작하고 폼에 애니메이트 하나와 버튼 두 개를 배치한다. 그리고 애니메이트의 commonAVI 속성을 aviCopyFile로 변경해 보자. 여기까지 작업하면 폼의 모양은 다음과 같아진다.




폼에 벌써 파일 복사 애니메이션이 표시되어 있다. 이제 버튼 두 개의 OnClick 이벤트 핸들러에 동영상을 재생/정지하는 코드를 다음과 같이 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Animate1.Active:=True;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Animate1.Active:=False;
end;
```

코드를 보다시피 애니메이트의 Active 속성을 True로 변경해 주기만 하면 재생이 시작된다. Active 속성은 디자인중에도 바꿀 수 있으며 이 속성이 True가 되기만 하면 디자인중에라도 애니메이션을 실행할 수 있다. 프로그램을 실행하

고 버튼을 눌러보면 재생이 시작되거나 중지될 것이다.

예제를 보다시피 애니메이트 컨트롤을 사용하는 것은 이렇게 쉽다. commonAVI 속성은 윈도우즈가 제공하는 AVI 데이터를 선택하도록 하는데 이 속성이 곧 재생할 애니메이션을 지정한다. 이 속성으로 지정할 수 있는 AVI 데이터는 다음과 같다. 윈도우즈를 조금이라도 사용해 본 사람들은 아주 익숙한 애니메이션일 것이다.

| 값 | 그림 |
|---|---|
| aviCopyFile |  |
| aviCopyFiles |  |
| aviDeleteFile |  |
| aviEmptyRecyle |  |
| aviFindComputer |  |
| aviFindFile |  |
| aviFindFolder |  |
| aviNone | 재생하지 않음 |

자기가 직접 AVI 파일을 만들어서 사용할 수도 있는데 이 때는 FileName 속성에 AVI 파일을 지정해 주면 된다. 이런 속성들 외에도 애니메이트에는 일부 구간만 재생하거나 특정 프레임으로 장면을 옮기는 메소드들이 있는데 자세한 사항은 레퍼런스를 참조하기 바란다.

# 8-4 마우스로 그림 그리기

페인트 브러시와 같이 실행중에 사용자가 마우스로 직접 그림을 그리는 프로그램을 작성해 보자. 프로그램의 코드로 그리는 것과는 달리 실행중에 사용자의 입력을 받아서 입력된 내용에 따라 그림을 그린다. 그림 그리기에 사용되는 장치는 아무래도 키보드보다는 자유롭게 움직일 수 있는 마우스가 제격이다.

## 가. 자유 곡선 그리기

8jang
mdraw1

마우스 이벤트로 그림을 그리는 방법은 생각보다 간단하다. 마우스 버튼이 눌러진 위치에서 MoveTo로 현재 좌표를 옮긴 후 커서가 움직이는대로 LineTo로 계속 연결해 주기를 마우스 버튼이 떨어질 때까지 반복하면 된다. 커서의 새 위치는 마우스 이벤트의 X,Y 인수로 전달되므로 이 인수를 사용하면 구할 수 있다.

단 이때 마우스 버튼이 눌러지지 않은 상태에서는 커서가 움직여도 그림이 그려지지 않아야 한다. 마우스 버튼의 현재 상태를 NowDraw라는 별도의 변수를 만들어 MouseDown에서 True로 만들고 MouseUp에서 False로 만들어 주면 된다. 일단 프로젝트를 시작해 보자. 컴포넌트를 배치할 것도 없으며 폼의 속성도 모두 디폴트를 사용하되 폼의 Color 속성만 clWhite로 변경하여 도화지처럼 만든다. NowDraw 변수는 세 개의 이벤트에서 공통적으로 사용해야 하므로 다음과 같이 전역 변수로 선언한다. 참, 거짓의 값만 가지므로 Boolean형이 적합하다.

```
var
  Form1: TForm1;
  NowDraw: Boolean;
```

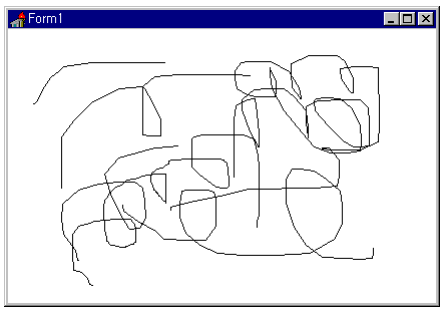
전역 변수로 선언해야 하므로 interface부의 var 선언에 추가하여 NowDraw 변수를 선언하였다. 마우스 이벤트에 대한 코드는 다음과 같이 작성한다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  NowDraw:=True;
```

```
  Canvas.MoveTo(X,Y);
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  NowDraw:=False;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if NowDraw then
    Canvas.LineTo(X,Y);
end;
```

NowDraw 변수의 값을 버튼의 상태에 따라 바꿔주는 것 외에는 특별한 기교가 없다. 프로그램 실행중의 모습은 다음과 같다.

**그림**
마우스의 이동에 따라 선을 그린다.




그림을 그리는 데는 아무런 문제가 없다. 그러나 OnPaint 이벤트에서 그린 그림이 아니기 때문에 가려졌다 다시 나타나거나, 최소화시킨 후 다시 최대화시키면 그림이 지워진다. 이런 문제를 해결하기 위해서는 그려놓은 그림을 일일이 메모리에 보관시켜 두어야 하는데 쉽지 않은 문제이다.

## 나. 선의 색상 변경

8jang
mdraw2

마우스로 정해진 곡선만 그리다 보니 너무 단조로와 선의 색상을 변경시켜 보기로 한다. 색상 선택에는 ColorDialogBox를 사용한다. 이 대화상자는 윈도우즈가 제공하는 공통 다이얼로그 박스이며 자세한 내용은 11장에서 언급하므로

당장 읽지 말고 11장을 읽고 난 후에 다시 읽어보기 바란다.

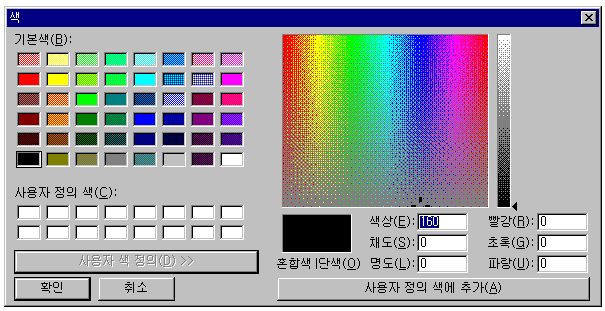
Dialogs 페이지의 6번째에 있는 ColorDialog를 선택하고 이 컴포넌트를 폼에 배치한 후 MouseDown 이벤트를 다음과 같이 변경한다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
if Button=mbLeft then
  begin
  NowDraw:=True;
  Canvas.MoveTo(X,Y);
  end
else
  if ColorDialog1.Execute then
    Canvas.Pen.Color:=ColorDialog1.Color;
end;
```

마우스의 오른쪽 버튼을 누르면 색상 변경 대화상자가 나타나며 여기서 선택한 색상으로 Pen의 Color 속성을 변경해 준다. 실행중에 마우스 오른쪽 버튼을 누르면 다음과 같은 색상 선택 대화상자가 나타나며 이 대화상자에서 선택한 색상으로 선이 그려진다.

**그림**

색상 선택 공통 대화상자




여기서는 선의 색상만 변경해 보았지만 Pen의 여러 가지 속성을 바꾸어 주어 선의 두께, 모양도 변경할 수 있다.

# 다. 키보드로 그리기

8jang
KDraw

이번에는 키보드로 그림을 그리는 실습을 해 보고 키보드 이벤트에 대해서도

연구해 보자. 마우스로 그림을 그릴 때는 OnMouseMove 등의 마우스 이벤트를 사용했지만 키보드로 그림을 그리려면 키보드 이벤트를 사용해야 한다. 키보드 이벤트에는 다음 세 가지 종류가 있다. 일단 키가 눌러질 때와 떨어질 때 발생하는 두 개의 이벤트부터 보자.

```
OnKeyDown:TKeyEvent;
OnKeyUp:TKeyEvent;
TKeyEvent = procedure (Sender: TObject; var Key: Word; Shift: TShiftState) of object;
```

둘 다 같은 타입의 이벤트인데 인수로 어떤 키가 눌러졌는지를 나타내는 Key 인수와 Shift, Alt, Ctrl 키값을 나타내는 Shift 인수가 전달된다. 어떤 키가 눌러졌는지를 전달하는 Key는 가상 키 코드로 나타내는데 다음 표와 같다.

**표** 가상 키 코드 테이블

| 가상키 코드 | 값 | 키 |
|---|---|---|
| VK_LBUTTON | 01 | |
| VK_RBUTTON | 02 | |
| VK_CANCEL | 03 | Ctrl-Break |
| VK_MBUTTON | 04 | |
| VK_BACK | 08 | Backspace |
| VK_TAB | 09 | Tab |
| VK_CLEAR | 0C | NumLock가 꺼져 있을 때의 5 |
| VK_RETURN | 0D | Enter |
| VK_SHIFT | 10 | Shift |
| VK_CONTROL | 11 | Ctrl |
| VK_MENU | 12 | Alt |
| VK_PAUSE | 13 | Pause |
| VK_CAPITAL | 14 | Caps Lock |
| VK_ESCAPE | 1B | Esc |
| VK_SPACE | 20 | 스페이스 |
| VK_PRIOR | 21 | PgUp |
| VK_NEXT | 22 | PgDn |
| VK_END | 23 | End |
| VK_HOME | 24 | Home |

| | | |
|---|---|---|
| VK_LEFT | 25 | 좌측 커서 이동키 |
| VK_UP | 26 | 위쪽 커서 이동키 |
| VK_RIGHT | 27 | 오른쪽 커서 이동키 |
| VK_DOWN | 28 | 아래쪽 커서 이동키 |
| VK_SELECT | 29 | |
| VK_PRINT | 2A | |
| VK_EXECUTE | 2B | |
| VK_SNAPSHOT | 2C | Print Screen |
| VK_INSERT | 2D | Insert |
| VK_DELETE | 2E | Delete |
| VK_HELP | 2F | |
| | 30~39 | 숫자키 0~9 |
| | 41~5A | 영문자 A~Z |
| VK_NUMPAD0~ VK_NUMPAD9 | 60~69 | 숫자 패드의 0~9 |
| VK_MULTIPLY | 6A | 숫자 패드의 * |
| VK_ADD | 6B | 숫자 패드의 + |
| VK_SEPARATOR | 6C | |
| VK_SUBTRACT | 6D | 숫자 패드의 - |
| VK_DECIMAL | 6E | 숫자 패드의 . |
| VK_DIVIDE | 6F | 숫자 패드의 / |
| VK_F1~VK_F16 | 70~7F | 펑션키 F1~F16 |
| VK_NUMLOCK | 90 | Num Lock |
| VK_SCROLL | 91 | Scroll Lock |

가상 키 코드란 키보드의 종류에 상관없이 적용할 수 있는 키 코드이며 일반적으로 키의 스캔 코드값으로 정의되어 있다. Shift 인수는 (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble)의 집합형이며 해당 키(또는 마우스 버튼)이 눌러져 있으면 집합 원소가 이 인수로 전달된다. Key와 Shift의 두 상태를 조사하면 키보드의 어떤 키가 눌러졌는지를 알 수 있다.

```
if  Key=Vk_Left then     // 왼쪽 커서 이동키가 눌러졌다.
if (Key=Vk_Insert) and (ssCtrl in Shift) then  // Ctrl + Ins 누름
```

세 번째 키보드 이벤트는 OnKeyDown 바로 다음에 발생하는 OnKeyPress 이벤트이며 이 이벤트는 다음과 같이 정의되어 있다.

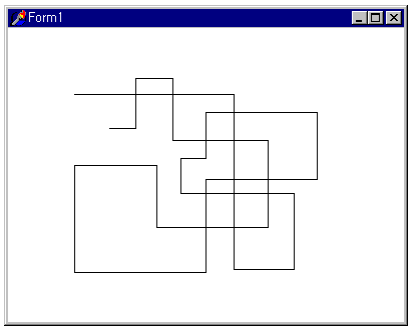
```
type TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
```

인수로 Key값 하나만 전달되는데 이 값은 가상 키 코드가 아니라 입력된 문자값이다. OnKeyDown은 모든 키에 대해 발생하는데 비해 OnKeyPress는 문자키에 대해서만 발생한다는 차이점이 있다. 즉 A, 7 등의 키는 OnKeyPress 이벤트를 발생시키지만 펑션키, 커서 이동키 등은 OnKeyPress 이벤트를 발생시키지 않는다.

키보드로 선을 긋는 예제를 만들려면 커서 이동키의 입력을 받을 수 있는 OnKeyDown 이벤트를 사용하는 것이 가장 적합하다. 새 프로젝트를 시작하고 그림이 잘 보이도록 폼을 흰색으로 만들자. 그리고 다음 코드를 작성한다.

```
implementation

{$R *.DFM}
var
  x,y:Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  x:=100;
  y:=100;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
    Vk_Left:
      x:=x-1;
    Vk_Right:
      x:=x+1;
    Vk_Up:
      y:=y-1;
    Vk_Down:
      y:=y+1;
  end;
  Canvas.Pixels[x,y]:=clBlack;
end;
```

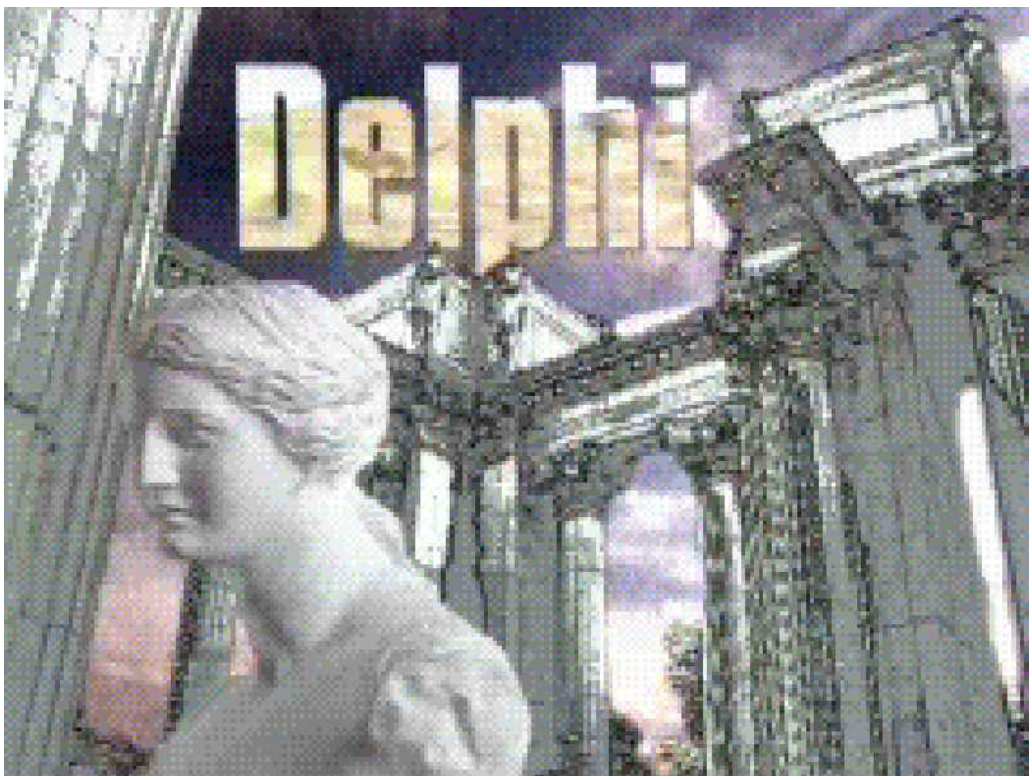
implementation 바로 아래에 현재 좌표값을 기억할 전역 변수 x,y를 선언하고 FormCreate에서 이 값을 (100,100)으로 초기화하였다. FormKeyDown에서는 키가 눌러질 때 Key 인수로 어떤 키가 눌러졌는지를 검사해 보고 x,y값을 적절하게 증감시킨 후 x,y위치에 점을 찍었다. 그래서 커서 이동키를 누르고 있으면 그 방향으로 계속해서 선이 그어질 것이다.




어릴 때 신나게 가지고 놀던 다이얼 스케치라는 장난감처럼 동작한다.

# 제 9 장

# 타이머



# 9-1 시계

## 가. 타이머 컴포넌트

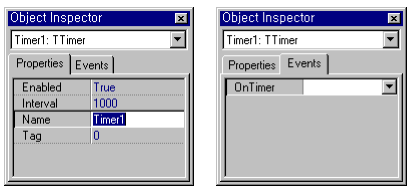
System 팔레트의 첫 번째 칸에 시계 모양의 아이콘을 가진 타이머 컴포넌트(  )가 있다. 타이머 컴포넌트가 하는 일은 일정한 주기로 OnTimer 이벤트를 발생시키는 일이며 프로그램에서는 OnTimer 이벤트 코드에서 주기적으로 계속 반복해야 할 일을 처리한다. 타이머 컴포넌트를 폼에 배치하면  요렇게 생긴 아이콘만 하나 나타난다. 이 아이콘이 의미하는 바는 이 프로그램에서 OnTimer 이벤트를 사용하겠다는 의사표시이며 아이콘 자체는 실행중에 보이지 않는다. 메뉴 컴포넌트와 비슷하며 실행중에는 보이지 않는 이런 컴포넌트를 비가시적(nonvisual) 컴포넌트라고 하며 폼에 놓는 위치나 크기는 의미가 없다.

타이머가 가지는 거의 유일한 속성은 Interval이며 이 속성은 OnTimer 이벤트의 주기를 1000분의 1초 단위로 지정한다. 디폴트값이 1000이므로 1초에 한 번 OnTimer 이벤트를 발생시킨다. 좀 더 빠른 주기가 필요하면 Interval을 500이나 300으로 낮게 설정하도록 하고 반대로 주기를 늦추려면 Interval 속성에 1000보다 더 큰 값을 준다.

Interval 속성 외에도 타이머의 동작 여부를 통제하는 Enabled 속성이 있다. 이 속성이 True이면 타이머는 OnTimer 이벤트를 발생시키며 False이면 OnTimer 이벤트를 발생시키지 않는다. Enabled 속성의 디폴트값은 True이며 항상 타이머가 사용 가능하지만 타이머의 동작을 잠시 정지시킬 필요가 있을 때는 이 속성을 False로 바꾸어 둔다. 오브젝트 인스펙터를 살펴보면 타이머는 Name, Tag 등 다른 모든 컴포넌트가 가지는 속성 두 가지를 포함하여 단 4개의 속성만 가짐을 알 수 있다.

**그림**

타이머의 속성과 이벤트




타이머 컴포넌트가 가지는 유일한 이벤트는 OnTimer뿐이며 이 이벤트를 발생시키는 것이 또한 타이머가 해야 할 유일한 일이다. 타이머 컴포넌트 자체는

이렇게 간단하지만 타이머 이벤트를 사용하기란 쉽지 않을 것이다.

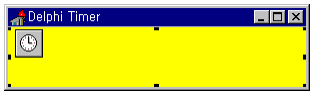
# 나. 시계

9jang
timer1

타이머를 사용하여 아주 간단한 시계를 만들어 보자. 시계는 1초에 한번씩 주기적으로 바뀌며 시간이 바뀔 때마다 화면에 시간을 다시 출력해 주어야 한다. 이런 주기적인 작업은 주기적으로 발생하는 OnTimer 이벤트를 사용하는 것이 제일 적당하다. 타이머 컴포넌트를 폼에 배치하여 1초에 한번씩 OnTimer 이벤트를 발생하도록 해두고 OnTimer 이벤트 핸들러에서 화면에 시간을 출력해 주도록 설계한다. 새로운 프로젝트를 시작한 후 타이머 컴포넌트와 시간을 출력할 레이블을 배치하고 속성을 설정한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 타이머 | Interval | 디폴트값 1000을 그대로 사용 |
| 폼 | Caption | Delphi Timer |
| | Color | clYellow |
| | BorderStyle | bsSingle |
| | BorderIcons | 아래 설명 참조 |
| | Width | 250 |
| | Height | 70 |
| 레이블 | Caption | 모두 지움 |
| | Align | alClient |
| | Alignmenu | taCenter |
| | Font | Arial 32 |

시계가 1초 단위로 움직이므로 타이머를 배치한 후 Interval 속성은 디폴트 그대로 사용한다. 폼은 크기를 대폭 줄이고 경계선을 단일선으로 하여 실행중에 크기가 조정되지 않도록 한다. 시간을 출력할 레이블이 화면 전체를 차지할 수 있도록 Align 속성을 alClient로 설정하였다. 속성 조정을 마친 후의 폼 모양은 다음과 같다.

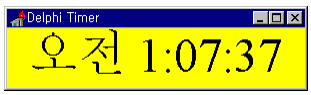


BorderIcons는 타이틀 바에 배치한 버튼을 설정하는 속성인데 크기 조정이 불가능하므로 biMaximize 세부 속성을 False로 만들어 최대화 버튼을 없애고 조절 메뉴와 최소화 버튼은 남겨 둔다. 작성해야 할 코드는 다음 한 줄 뿐이다. 타이머를 더블클릭하여 OnTimer 이벤트를 작성한다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 label1.caption:=TimeToStr(Time);
end;
```

이 코드에서 사용한 Time 함수는 현재 시간을 조사해 준다. 조사한 값은 TDateTime이라는 실수값(Float)으로 리턴된다. 왜 날짜와 시간을 저장하는 데이터형이 실수형인가 하면 델파이가 날짜, 시간을 저장할 때 날짜는 정수부에 저장하고 시간은 소수부에 저장하기 때문이다. 날짜의 소수부가 시간인 것은 논리적으로 맞는 얘기다. 12시간을 0.5일이라고 표현할 수도 있지 않은가?

TDateTime은 실수값이며 델파이 고유의 방식으로 날짜와 시간을 기억시키는 데이터형이므로 이 값을 곧바로 화면에 출력할 수는 없고 문자열로 바꾸어 주어야 한다. 시간을 문자열로 바꾸어 주는 TimeToStr이라는 편리한 함수가 있으므로 이 함수를 사용하면 조사한 시간을 레이블에 출력할 수 있다. 참고로 날짜를 문자열로 바꾸는 DateToStr과 날짜와 시간을 한꺼번에 문자열로 바꾸어 주는 DateTimeToStr 함수도 있으니 많이 애용해 주기 바란다. 거꾸로 문자열을 날짜, 시간으로 바꾸어 주는 StrToDateTime 함수도 있다. OnTimer 이벤트 코드만 작성하면 프로그램은 완성되고 실행시키면 시간이 출력되고 정확하게 동작한다.




현재 시간이 문자열로 나타나는데 단 이 문자열은 현재 설치된 윈도우즈의 버전에 따라 달라질 수 있다. 한글 윈 95에서는 "시:분:초 오전" 으로 나타나며 영문 윈도우즈에서는 오전/오후 대신 AM/PM으로 나타날 것이다. TimeToStr 함수가 윈도우즈에 설정된 시간 포맷에 맞게 문자열을 변경해 주기 때문이다.

아주 작은 크기로 만들었으므로 화면 한귀퉁이에 모셔둘 법도 한 그런 프로그램이다. 한 가지 아쉬운 점이라면 프로그램이 처음 실행되고 1초가 지나야 시간이 출력된다는 점이다. 이 문제를 해결하려면 OnTimer 이벤트 핸들러를 폼의 OnCreate 이벤트에 연결시켜 주면 된다. 폼이 만들어지자 마자 시간이 출력될

9jang
chgcol

것이다.

타이머 컴포넌트를 사용하면 시계 정도는 이렇게 간단하게 만들 수 있다. 그만큼 타이머가 사용하기 쉽다는 얘기며 이용하기에 따라서는 무궁 무진하게 활용할 수 있다. 다음 예제는 타이머를 사용하며 폼의 색상을 주기적으로 바꾸어 준다. 폼에 타이머를 배치한 후 OnTimer 이벤트를 다음과 같이 작성해 주면 된다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
if Form1.Color=clRed then
  Form1.Color:=clBlue
else
  Form1.Color:=clRed;
end;
```

폼의 현재 색상이 빨간색이면 파란색으로 바꾸고 파란색이면 빨간색으로 바꾸어 줌으로써 폼의 배경 색상을 빨간색, 파란색으로 교대로 바꾸어 준다. 타이머의 주기를 빠르게 하면 번쩍거리는 폼을 만들 수 있을 것이고 사용자의 눈길을 끌고자 할 때 아주 좋은 효과를 나타낸다. 실행중의 모습은 다음과 같다.


이 예제에서는 Interval 속성을 300으로 설정하여 0.3초 주기로 폼의 색상을 변경한다.

# 다. 시간 함수

## ■ TDateTime 형

델파이는 시간과 날짜를 보관하기 위해 TDateTime형을 사용한다. TDateTime형은 서기 1년 1월 1일을 기점으로 하여 경과한 날 수로 날짜를 저장하며 시간은 TDateTime형의 소수부에 저장한다. 즉 TDateTime형은 실수 형태를 띠고 있으며 실제로 형 정의가 Float형으로 되어 있다.

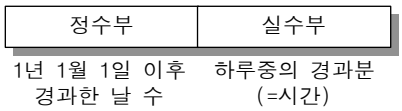



**그림** TDateTime 형의 구조

C 언어의 경우는 시간을 보관하는 데이터형과 날짜를 보관하는 데이터형이 별도의 구조체(파스칼의 레코드)로 정의되어 있어 시간과 날짜를 이루는 각 요소(년, 월, 일, 시, 분, 초)가 별도의 변수에 보관된다. 하지만 파스칼은 TDateTime형의 변수 하나만으로 날짜와 시간을 한꺼번에 보관한다. 각 요소를 일일이 보관하는 것이 아니라 하나의 값으로 통합하여 보관하는 것이다. 예를 들어 C가 "오전 2시 22분 34초"와 같이 시간을 기억한다면 파스칼은 7366초로 기억한다.

시간과 날짜를 이렇게 하나의 변수에 보관하더라도 각 요소를 분리하기 위해 별도의 코드를 작성하지 않아도 된다. 왜냐하면 TDateTime형의 변수를 마음대로 요리할 수 있는 여러 가지 함수가 제공되기 때문이다.

## ■ 시간 조사 함수

현재의 시간을 조사하는 함수에는 다음 세 가지가 있다.

**표** 시간과 날짜를 조사하는 함수

| 함수 | 의미 |
|---|---|
| Date | 현재의 날짜를 조사한다. |
| Time | 현재의 시간을 조사한다. |
| Now | 현재 시간과 날짜를 한꺼번에 조사한다. |

이 함수들을 인수없이 호출해 주기만 하면 원하는 정보를 TDateTime형으로 넘겨준다. 앞에서 만들어본 시계 예제에서 이미 Time 함수를 사용해 보았다. 날짜를 조사하여 레이블로 출력하려면 Date 함수를 사용하되 문자열로 바꾸어 주어야 하므로 DateToStr(Date)라고 쓰면 된다.

## ■ 요소 분리 함수

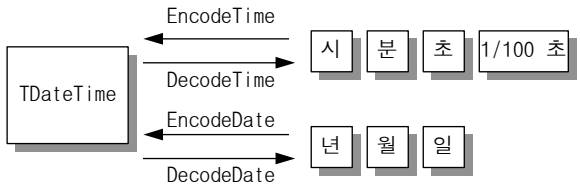
시간과 날짜의 각 요소를 분리하거나 결합시키는 함수들이다. TDateTime형의 변수 하나에서 날짜를 년, 월, 일로 분리할 수 있고 마찬가지로 시간을 시, 분, 초로 분리할 수 있으며 반대로 합치는 것도 가능하다.

**표** 시간과 날짜의 각 요소를 분리/결합하는 함수들

| 함수 | 의미 |
|---|---|
| EncodeTime | 각 요소로 분리된 시간을 합쳐 하나의 변수로 만든다. |
| DecodeTime | 시간을 각 요소로 분리한다. |
| EncodeDate | 각 요소로 분리된 날짜를 합쳐 하나의 변수로 만든다. |
| DecodeDate | 날짜를 각 요소로 분리한다. |

그림으로 설명해 보면 다음과 같다.

**그림** 각 요소의 분리 및 결합




시간이나 날짜를 각각 분리해서 다루어야 할 필요가 있다면 이 함수들을 사용한다. 대표적으로 시간을 분리하는 DecodeTime 함수만 살펴보자.

```
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);
```

첫 번째 인수로 현재 시간을 넘겨주며 나머지 인수는 분리한 요소를 돌려받기 위한 출력용 인수이다. 그래서 두 번째 이후의 인수는 앞에 var가 붙어 참조 호출을 사용한다. 네 개의 Word형 변수를 만든 후 이 함수를 호출하면 각 변수에 원하는 시간 요소가 분리되어 돌아올 것이다.

## ■ 문자열 변환 함수

Time이나 Date 함수에 의해 조사된 시간은 TDateTime형의 변수이므로 이 값을 곧바로 출력할 수는 없다. 에디트나 레이블로 시간을 출력하려면 문자열 형태로 바꾸어야 하며 다음과 같은 함수들이 사용된다.

**그림** DateTime 형을 문자열로 변환하는 함수들

| 함수 | 의미 |
|---|---|
| DateToStr | 날짜를 문자열로 바꾼다. |

| | |
|---|---|
| TimeToStr | 시간을 문자열로 바꾼다. |
| DateTimeToStr | 시간과 날짜를 문자열로 바꾼다. |
| StrToDate | 문자열을 날짜로 바꾼다. |
| StrToTime | 문자열을 시간으로 바꾼다. |
| StrToDateTime | 문자열을 시간과 날짜로 바꾼다. |

함수의 이름 자체가 워낙 설명적이므로 사실 별도의 설명이 필요없을 정도며 외우기도 아주 쉽다.

이 외에도 시간과 날짜의 출력 형태를 정밀하게 조정하는 몇 가지 함수가 더 준비되어 있다. 자세한 사항은 부록으로 제공되는 함수 레퍼런스를 참조하기 바란다.

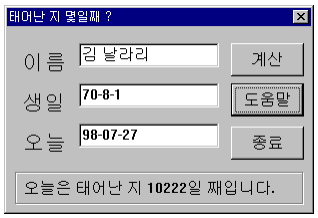
# 9-2 타이머 예제

## 가. 날짜 계산 예제

9jang
birth

사람이 살다 보면 태어난 지 몇일째 살고 있는지 궁금해지는 경우가 있다. 필자만 그런지 모르겠지만 아무튼 이런 계산을 해 보는 것은 참 흥미있는 일이 아닌가 하는 생각이 마구 마구 들어서 날짜를 계산하는 예제를 만들어 보았다. 태어난 날짜와 오늘 날짜를 입력해 주면 오늘이 몇일째인지 계산해 준다. 필자는 성격이 하도 희한해서 이런 일을 아주 재미있게 생각하고 즐기고 있다. 일기장에 매일 그날이 몇일째인가를 적어 놓기도 하고 기념이 될만한 날짜는 일일이 기억해 두기도 한다. 예를 들어 대학 입학을 한 날짜는 6788일째, 첫키스를 한 날짜는 7298일째, 입대한 날은 7478일째 등등.

지면 사정으로 인해 구체적인 프로그램 제작 과정과 소스는 여기서 밝히지 않기로 한다. 배포 CD에 예제 소스가 있으므로 직접 분석해 보기 바라며 몇 가지 중요한 부분만 분석을 해보자. 실행중 화면은 다음과 같다.




### ■ 날짜 계산

두 날짜가 주어질 때 경과한 날짜를 조사하는 방법은 의외로 간단하다. 년끼리 빼서 365를 곱하고 또 월, 일끼리 빼서 더해주고 윤년 계산해 주고 아주 복잡할 것 같지만 델파이가 날짜를 기억하는 방식이 단순하기 때문에 원터치로 경과 날짜를 조사할 수 있다.

델파이가 날짜와 시간 기억에 사용하는 TDateTime 데이터형은 년, 월, 일을 구분하여 저장하는 것이 아니라 서기 1년 1월 1일을 기점으로 하여 몇일이나 경과했는지를 기억한다. 기억될 때부터 년, 월의 단위가 날짜로 바뀌어 기억되므로 두 날짜간의 차이는 단순한 뺄셈을 사용하면 된다.

에디터에 입력된 날짜는 문자열이므로 먼저 StrToDate로 날짜로 바꾼 후 뺄

셈을 하고 출력을 할 때는 다시 FloatToStr 함수로 문자열로 바꾸어 주어야 한다. 날짜를 계산하여 출력하는 코드는 다음과 같으며 "계산" 버튼의 OnClick 이벤트 핸들러에 작성되어 있다.

```
procedure Tbirthcalc.execalcClick(Sender: TObject);
begin
 birthday:=strtodate(birthedit.text);
 today:=strtodate(todayedit.text);
 elapseday:=today-birthday;
 result.caption:='오늘은 태어난 지 '+floattostr(elapseday)+
  '일 째입니다.';
end;
```

오늘 날짜에서 생일 날짜를 빼주기만 하면 된다. 계산한 결과인 문자열은 +연산자를 사용하여 앞 뒤에 설명문을 붙여 읽기 쉽도록 가공한다.

## ■ 정보 기억

프로그램을 매번 실행할 때마다 이름과 생일을 입력해 주는 것은 확실히 번거로운 일이다. 처음 실행할 때 이름과 생일을 한번만 입력해 주면 이 정보를 기억해 두었다가 다음 실행할 때부터는 입력해 주지 않아도 되게끔 하면 훨씬 더 편리할 것이다. 또한 오늘 날짜를 자동으로 입력해 주어 실행하기만 하면 오늘이 몇일째인지를 계산해 주도록 만들고자 한다.

무엇인가 정보를 기억하기 위해서는 INI 파일을 사용해야 한다. 메모리는 빠르고 정확하기는 하지만 전기가 없으면 아무것도 기억하지 못하므로 하드 디스크에 파일의 형태로 정보를 남기는 수밖에 없다. 폼이 열릴 때 INI 파일에서 정보를 읽어 에디트에 입력해 주고 프로그램을 끝낼 때 변경된 정보를 다시 INI 파일에 기록하여 다음 실행에 사용할 수 있도록 해준다. 오늘 날짜를 알아 보려면 Date 함수를 쓴다.

정보를 기억할 INI 파일은 크게 두 가지 종류로 나누는데 첫째는 윈도우즈가 사용하는 WIN.INI에 정보를 기입하는 방법이고 두 번째는 프로그램 고유의 INI 파일을 만들어 사용하는 것이다. 여기서는 BIRTH.INI라는 고유의 파일을 만들어 사용한다. 프로그램을 종료할 때 INI 파일에 기록을 하고 프로그램을 시작할 때 보관해 놓은 정보를 읽는다. 프로그램 종료시점은 FormDestroy 이벤트가 발생할 때이므로 이 이벤트에 정보 보관 코드를 작성한다.

```
procedure Tbirthcalc.FormDestroy(Sender: TObject);
begin
  writeprivateprofilestring('INFO','NAME',
    PChar(nameedit.text),'BIRTH.INI');
  writeprivateprofilestring('INFO','BIRTH',
    PChar(birthedit.text),'BIRTH.INI');
end;
```

이렇게 저장해 놓은 정보는 폼이 처음 열릴 때인 FormShow 이벤트에서 변수로 읽어들인다.

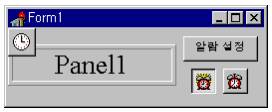
```
procedure Tbirthcalc.FormShow(Sender: TObject);
var
  pst:array [0..128] of char;
begin
  getprivateprofilestring('INFO','NAME','김 날라리',
    pst,128,'BIRTH.INI');
  nameedit.text:=pst;
  getprivateprofilestring('INFO','BIRTH','70-8-1',
    pst,128,'BIRTH.INI');
  birthedit.text:=pst;
  todayedit.text:=datetostr(date);
  birthcalc.execalcClick(sender);
  todayedit.setfocus;
end;
```

INI 파일을 제어하려면 윈도우즈의 API 함수를 사용해야 한다. 델파이는 윈도우즈 API 함수를 훌륭히 지원하지만 당장은 설명할 단계가 아니라 생각되므로 11장으로 설명을 보류한다.

# 나. 알람 시계

9jang
alarm

이번에는 정해진 시간이 되었을 경우 사용자에게 시간을 알려주는 경보 기능을 가진 알람 시계를 만들어 보자. 패널, 버튼, 타이머 각각 하나씩, 그리고 스피드 버튼 두 개가 필요하다. 폼에 컴포넌트를 배치한 후의 모양은 다음과 같다.

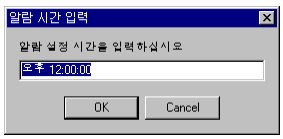



패널은 현재 시간을 출력할 목적으로 사용된다. 어차피 시간을 출력할 목적이라면 문자열을 나타낼 수만 있으면 되므로 레이블, 에디트, 메모 등등 어떤 컴포넌트를 사용해도 상관없다. 패널은 문자열 출력 외에도 약간의 장식 기능이 내장되어 있으므로 멋부리기에 적당하다. 패널의 베벨을 적당히 조정하도록 하고 폰트는 큼직한 크기로 설정하자.

버튼은 알람 시간을 입력받기 위해 사용한다. 알람 시간은 AlarmTime이라는 전역 변수로 선언되어 있으며 버튼의 OnClick 이벤트에서 InputBox를 사용하여 입력받는다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
AlarmTime:=StrToTime(InputBox('알람 시간 입력',
    '알람 설정 시간을 입력하십시요',
    TimeToStr(AlarmTime)));
```

시간을 입력받을 수 있는 함수는 없으므로 일단 문자열로 시간을 입력받은 후 StrToTime 함수를 사용하여 입력받은 문자열을 TDateTime형으로 바꾸어 AlarmTime 변수에 대입한다.




스피드 버튼은 알람 기능을 활성화시키거나 잠시 중지시키며 두 개가 한쌍이 되어 라디오 버튼처럼 동작한다. 두 버튼의 GroupIndex 속성을 1로 설정하여 그룹을 이루도록 하였다. 알람 가동 여부는 AlarmOn이라는 변수에 저장되며 스피드 버튼의 OnClick 이벤트에서 이 값을 변경한다.

```
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  AlarmOn:=True;
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
```

```
  AlarmOn:=False;
end;
```

타이머의 OnTimer 이벤트에서는 현재 시간을 출력하는 일 외에도 현재 시간과 알람 시간을 비교하여 두 시간이 일치할 경우 사용자에게 시간 경보를 해주어야 한다. 현재 시간인 NowTime과 알람 시간인 AlarmTime을 비교하고자 한다면 다음과 같이 코드를 작성하면 될 것 같다.

```
if AlarmTime=NowTime then ....
```

그러나 이렇게 비교하면 항상 두 시간이 틀린 것으로 오판을 한다. 왜냐하면 시간의 최하 단위는 초가 아니라 1/100초이기 때문에 시, 분, 초까지 맞더라도 1/100초 단위가 틀리면 경보 장치가 가동되지 않기 때문이다. 그래서 이런 간단한 비교를 할 수 없으며 다음과 같이 시간 요소를 분리한 후 1/100초 단위는 비교 대상에서 제외시켜야 한다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  NowTime:TDateTime;
  NH,NM,NSec:Word;
  AH,AM,ASec:Word;
  dummy:Word;
begin
  NowTime:=Time;       {현재 시간을 조사하여 출력한다.}
  Panel1.Caption:=TimeToStr(NowTime);
  {시간 비교를 위해 요소를 분리한다.}
  DecodeTime(NowTime,NH,NM,NSec,dummy);
  DecodeTime(AlarmTime,AH,AM,ASec,dummy);
  {시,분,초까지만 비교한다.}
  if (NH=AH) and (NM=AM) and (NSec=ASec) and AlarmOn Then
    begin
    MessageBeep(0);
    ShowMessage('약속 시간입니다.');
    end;
end;
```

시간 요소를 DecodeTime 함수를 사용하여 분리한 후 두 시간의 시, 분, 초가 모두 같고 AlarmOn이 True일 경우 경보장치를 가동한다. 이 예제에서 사용하는 경보장치는 비프음을 내고 대화상자를 보여주는 정도지만 작성하기에 따라

서는 프로그램을 실행시키거나 음악을 연주하는 것도 가능하다. 마지막으로 FormCreate 이벤트 핸들러에서 변수를 초기화 해주면 예제가 완성된다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 AlarmTime:=StrToTime('12:00:00');
 AlarmOn:=True;
 Panel1.Caption:=TimeToStr(Time);
end;
```

# 다. 아날로그 시계

9jang
analog

디지털 시계는 만들기는 편리하지만 웬지 딱딱한 느낌이 든다. 숫자로 나타나는 시계보다는 바늘로 되어 있는 시계가 훨씬 더 예쁘고 우아한 맛이 있다. 여기서는 바늘로 되어 있는 아날로그 시계를 만들어 보기로 한다. 시간을 조사하는 방법은 디지털 시계와 동일하지만 출력 방법이 훨씬 더 복잡하다. 윈도우즈가 제공하는 clock.exe를 보면 그래픽적으로 시간을 표시해 주며 윈도우의 크기가 바뀌면 시계의 크기도 따라 변경된다는 특징을 가지고 있다. 이런 기능을 프로그래밍하려면 시간 루틴보다는 그래픽에 더 많은 신경을 써야 하며 수학적으로도 꽤 복잡한 계산을 해야 한다. 예제를 만드는 과정은 직접 보이지 않고 전체 소스를 보인 후 분석을 하도록 한다. 주석을 참조하여 스스로 분석해 보기 바란다.

```
{간단한 아날로그 시계. 폼 크기에 따라 시계의 크기도 변한다.}
unit Analog_f;

interface

uses
 SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
 Forms, Dialogs, ExtCtrls;

type
 TForm1 = class(TForm)
  Timer1: TTimer;
  procedure FormPaint(Sender: TObject);
  procedure FormResize(Sender: TObject);
  procedure Timer1Timer(Sender: TObject);
 private
  { Private declarations }
 public
```

```
  { Public declarations }
 end;

{pi2 는 각도 계산에 사용되며 원주율의 두배, 즉 360 도를 의미한다.}
const
 pi2=6.283;
var
 Form1: TForm1;
 cx,cy,shortax:integer;   {중심 좌표 및 단축의 길이}
 handrad,hourrad:integer;  {바늘 길이와 시간 표시 길이}

implementation

{$R *.DFM}

{폼이 다시 그려져야 할 때마다 폼 크기에 따라 바늘의 크기
와 시간 표시의 크기를 다시 계산하고 시간 표시를 알맞은 크기
로 출력한다.}
procedure TForm1.FormPaint(Sender: TObject);
var
 r:single;  {시간 표시의 반지름}
 i:integer; {루프 제어 변수}
 angle:single;  {각도}
 x,y:integer;   {좌표}
begin
 cx:=ClientWidth div 2;   {폼의 중심 좌표 구함}
 cy:=ClientHeight div 2;
 if cx>cy then      {두 중심중 짧은 단축을 구한다.}
  shortax:=cy
 else
  shortax:=cx;
 {시계 바늘의 길이는 단축의 4 분 3 으로 한다.}
 handrad:=round(shortax*(3/4));
 {단축의 8 분의 7 위치에 시간 표시를 한다.}
 hourrad:=round(shortax*(7/8));
 {팬은 검정색을 사용하고 면은 흰색을 사용한다.}
 Canvas.Pen.Color:=clBlack;
 Canvas.Brush.Color:=clGreen;
 {1~12 시까지 각 시간 위치에 둥근 원을 그린다.}
 for i:=0 to 11 do
 begin
  {시간 표시의 반지름은 단축의 20 분의 1}
  r:=shortax/20;
  {3,6,9,12 시는 조금 더 크게 만든다}
  if (i mod 3)=0 then r:=round(r*(6/5));
  angle:=i*pi2/12;
```

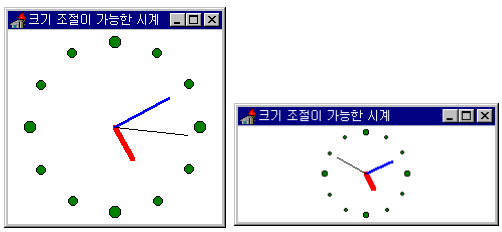
```
   x:=round(hourrad*sin(angle)+cx);
   y:=round(-hourrad*cos(angle)+cy);
 Canvas.Ellipse(x-round(r),y-round(r),x+round(r),y+round(r));
  end;
end;

{폼의 크기가 변하면 변한 크기에 맞도록 전체 폼을 다시
그린다.}
procedure TForm1.FormResize(Sender: TObject);
begin
Refresh;
end;

{1 초에 한번씩 시간을 갱신한다.}
procedure TForm1.Timer1Timer(Sender: TObject);
var
  Hour,Minute,Second,temp:Word;  {시간의 각 요소}
  hang,mang,sang:single;        {각 요소의 각도}
  hx,hy,mx,my,sx,sy:integer;    {각 요소의 좌표}
begin
{현재 시간을 구해 시,분,초의 각 요소로 분리한다.}
DecodeTime(Time,Hour,Minute,Second,temp);
{각 바늘의 각도를 구해낸다.}
sang:=Second*pi2/60;
mang:=Minute*pi2/60+sang/60;
hang:=(Hour mod 12)*pi2/12+mang/60;
{각 바늘의 위치를 구해 낸다.}
hx:=round(handrad/2*sin(hang)+cx);   {시침은 길이를 반으로}
hy:=round(-handrad/2*cos(hang)+cy);
mx:=round(handrad*5/6*sin(mang)+cx);  {분침은 길이를 5/6 로}
my:=round(-handrad*5/6*cos(mang)+cy);
sx:=round(handrad*sin(sang)+cx);     {초침이 가장 길다}
sy:=round(-handrad*cos(sang)+cy);

with Canvas do
begin
 {바늘 전체를 덮을 수 있는 큰 원을 그림으로써 이전 시간에
 출력된 바늘을 지운다. 가장 간단한 방법이지만 속도상으로는
 그리 좋은 효율을 발휘하지 못한다.}
 Pen.Color:=ClWhite;
 Brush.Color:=clWhite;
 Ellipse(cx-handrad,cy-handrad,cx+handrad,cy+handrad);
 {시침은 붉은색 굵기 4}
 Pen.Color:=clRed;
 Pen.Width:=4;
 MoveTo(cx,cy);
```

```
  LineTo(hx,hy);
  {분침은 파란색 굵기 2}
  Pen.Color:=clBlue;
  Pen.Width:=2;
  MoveTo(cx,cy);
  LineTo(mx,my);
  {초침은 검정색 굵기 1}
  Pen.Color:=clBlack;
  Pen.Width:=1;
  MoveTo(cx,cy);
  LineTo(sx,sy);
end;
end;

end.
```

그래픽은 폼 위에 직접 그려지므로 타이머 컴포넌트 외에는 별도의 컴포넌트가 필요없다. 폼의 속성도 특별히 조정해 줄 필요가 없으며 바탕색인 Color 속성만 clWhite로 설정하여 그림이 잘 보이도록만 해두면 된다. 프로그램 실행중의 모습은 다음과 같다.

**그림**

아날로그 시계




보다시피 윈도우의 크기에 따라 시계의 크기도 같이 변하도록 되어있다. 이 프로그램을 구성하는 이벤트 핸들러는 다음 세 개뿐이다. 대부분의 코드가 FormPaint와 Timer1Timer에 집중되어 있으므로 이 두 개의 이벤트 핸들러만 제대로 분석하면 이 예제를 이해할 수 있다.

| 이벤트 핸들러 | 동작 |
|---|---|
| FormPaint | 폼이 다시 그려져야 할 때 호출되며 폼 크기에 따라 시계의 크기를 다시 계산하여 적당한 크기로 그린다. |

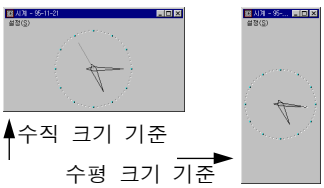
| | |
|---|---|
| FormResize | 폼의 크기가 변하면 시계의 크기가 변해야 하며 변한 크기대로 다시 그려져야 하므로 Refresh 메소드를 호출한다. 이 메소드는 폼을 지운 후 FormPaint를 호출하여 다시 그리도록 한다. |
| Timer1Timer | 1초에 한번씩 호출되며 시간을 갱신한다. |

## ■ 시계의 크기 계산

시계의 크기는 폼의 크기에 비례하여 증가하며 FormPaint 핸들러에서 시계를 그리기 전에 폼 크기에 맞게 다시 계산된다. cx,cy는 폼의 폭과 높이를 2로 나누어 폼의 중앙 좌표로 계산되며 이 값은 시계 바늘의 시작점이 된다. shortax는 cx,cy 중 작은 값을 가지는데 시계의 크기를 계산하는 기준이 된다. 윈도우즈의 시계를 봐도 시계는 항상 폼의 짧은 축을 기준으로 크기가 결정된다는 것을 알 수 있다.

**그림**

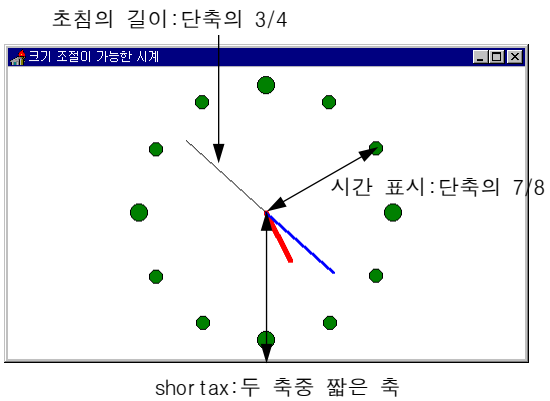
윈도우의 짧은 축을 기준으로 시계의 크기가 결정된다.




shortax를 기준으로 시계의 크기를 정하는 두 개의 변수가 정의된다. hourrad는 시간 표시를 나타내는 12개의 원과 중심과의 거리이며 handrad는 초침의 길이이다.

**그림**

시계 바늘 및 시간 표시의 반지름



시간 표시는 1, 2, 3...12와 같이 숫자로 하는 것이 더 알아보기가 좋겠지만 폰트를 예쁘게 구현하기는 무척 어려우므로 조그만 원 12개로 나타낸다. 시간 표시는 폼보다 조금 더 안쪽에 있어야 하므로 shortax보다 조금 더 짧게 만들고 초침의 크기는 그보다 조금 더 짧아져야 한다. 이 두 값(handrad, hourrad)이 구해지면 시계의 크기가 정해진 것이다.

## ■ 삼각함수 사용

시계 바늘의 초침은 시계 원주를 60등분하여 1초에 한번씩 움직인다. 초침의 한쪽 끝은 시계의 중심부이므로 이미 cx,cy로 정해져 있으며 나머지 한쪽 끝만 찾아 선을 연결해 주기만 하면 된다. 그렇다면 초침의 나머지 한쪽 끝 좌표는 어떻게 구하는가? 이 문제는 현재 초로부터 일차적인 함수 계산으로 간단하게 구할 수 없으며 초침의 각도를 계산하고 계산한 각도로부터 일정거리의 점을 구해내야 하는 어려운 문제이다. 초침을 구하는 문제뿐만 아니라 12개의 시간 표시를 하는 문제도 이와 동일하다. FormPaint 핸들러에서는 시간 표시의 좌표를 삼각함수를 사용해 구해낸다. 이 식의 의미를 알면 초침, 분침, 시침을 그리는 문제도 쉽게 이해할 수 있을 것이다. 일단 삼각함수에 대해 먼저 알아야 할 것은 sin, cos 함수가 취하는 인수는 360분법의 각도가 아니라 호도(radian)값이라는 점이다. 1호도란 반지름과 원주가 같아지는 각도를 말하며 호도와 각도 사이는 다음과 같은 관계가 있다.

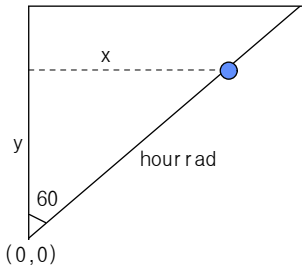
$$\text{호도} = \text{각도} * \frac{\pi}{180}$$

0시~11시까지 각각의 사간 표시점은 0도, 30도, 60도, ...330도의 각도를 이루므로 i시의 각도는 i*30도와 같이 일반화할 수 있다. 이 각도를 호도로 바꾸면 다음과 같이 된다.

$$i * 30 * \frac{\pi}{180} = i * \frac{\pi}{6} = i * \frac{2\pi}{12}$$

여기서 2π(상수 pi2로 정의)란 360도를 의미하며 i시의 시간 표시는 360도를 12등분하여 i를 곱한 각도에 위치한다. 각 시간의 각도가 구해지면 각도와 시간 표시점까지의 거리인 hourrad로부터 x,y 좌표를 구할 수 있다. 좌표를 구하는 식의 일반식을 유도하기 위해 i가 2인 특수한 경우를 보자. i가 2이면 각도는 60도가 된다.

그림
각도와 좌표와의 관계




여기서 우리가 알고 있는 값은 반지름인 hourrad와 각도 60도이며 알고자 하는 값은 x,y 좌표값이다. 각도와 반지름, 그리고 x,y는 삼각함수식으로 표현되며 삼각함수식에서 미지의 값 x,y를 구해낸다. 수학적으로 표현하면 다음과 같다. 별로 어려운 식이 아니므로 조금만 눈여겨 보도록 하자.

$$\sin(60°) = \frac{x}{hourrad}$$
$$\therefore x = \sin(60°) * hourrad$$
$$\cos(60°) = \frac{y}{hourrad}$$
$$\therefore y = \cos(60°) * hourrad$$


참고하세요

위에 보인 식에서는 0시 즉 12시 방향을 각도 0으로 간주하며 시계 방향으로 각도가 증가한다. 통상적으로 수학에서 사용하는 3시 방향 원점, 반시계 방향 증가와는 각도 체계가 조금 다르므로 주의하기 바란다. 각도 체계를 그렇게 마음대로 해도 되냐고 묻겠지만 누가 3시 방향이 꼭 0도여야 한다고 정해놓은 바는 없지 않는가? 프로그래머가 자신의 프로그램을 위해 나름대로의 각도 체계를 만들어 쓰더라도 문제될 것은 없다.

x,y 좌표를 구한 후에 x,y에 cx,cy를 더해 각 시간 표시를 cx,cy만큼 평행이동시킨다. 시계의 중심이 cx,cy이므로 시간 표시도 cx,cy만큼 이동을 해야 한다. 시간 표시 12개의 좌표를 구하는 식은 다음과 같다. 앞에서 보인 일반식을 델파이의 문법에 맞게 고친 것 뿐이다.

```
for i:=0 to 11 do
begin
 angle:=i*pi2/12;
```

```
  x:=round(hourrad*sin(angle)+cx);
  y:=round(-hourrad*cos(angle)+cy);
end;
```

계산 결과인 실수가 정수형 변수인 x,y에 곧바로 대입되지 못하므로 round 함수를 사용하여 정수로 바꾸어 주었다. 이렇게 좌표만 구하고 나면 Ellipse 메소드로 간단하게 타원을 그려 시간 표시를 출력한다. FormPaint에서는 시계의 크기에 맞게 시간 표시만 출력하며 시계 바늘은 그리지 않는다.

### ■ 바늘의 출력

바늘의 좌표를 구하는 방법도 시간 표시의 좌표를 구하는 방법과 거의 유사하되 시간이 변할 때마다 바늘을 그려야 하므로 Timer1Timer 이벤트 핸들러를 사용한다. 물론 타이머의 주기는 1초이다. 일단 현재 시간을 Time 함수로 구해내되 각 바늘의 위치를 계산하기 위해 시, 분, 초를 각각 분리해야 하며 그래서 DecodeTime 함수가 사용된다. 각 요소의 각도는 360도를 요소 범위로 나누어 구한다. 즉 초는 360도(pi2)를 60으로 나누고 시는 12로 나눈다. 분과 시는 그 자체의 각도 외에도 초나 분의 각도가 조금씩 더해진다. 각도로부터 바늘 끝의 좌표를 구해내는 방법은 앞에서 보인 삼각함수 사용법과 동일하다.

좌표가 구해졌으므로 이제 남은 일은 바늘을 그리는 일 뿐이다. 바늘을 그리는 루틴은 단순한 그래픽문 호출이므로 설명을 할 필요가 없을 것이다. 바늘을 그리기 전에, 이전에 그려져 있는 바늘을 지워주어야 하는데 XOR 방법, 바탕색으로 바늘을 다시 그리는 방법 등이 있지만 여기서는 간단하게 바늘 전체를 원으로 덮어버리는 무식한 방법을 사용하고 있다. 물론 XOR 방법에 비해 속도나 효율은 별로 좋지 못하지만 예제 수준에서는 괜찮은 효율을 보이는 것 같다. 깔끔한 출력을 위해서라면 8장에서 배운 가상 화면을 사용해 봄직하다.

## 라. 타이머 활용

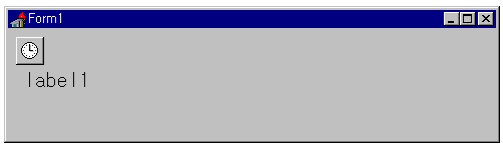
이 외에도 타이머를 활용할 수 있는 용도는 무궁 무진하다. 메시지를 일정 시간 간격으로 계속 화면에 뿌려주는 프로그램을 만든다고 해보자. 메시지를 출력하는 일은 레이블을 사용하면 어렵지 않게 구현할 수 있지만 사용자가 메시지를 읽을 시간을 주려면 타이머를 사용해야 한다. 전통적인 도스 프로그램에서는 다음과 같은 코드를 사용하여 이 문제를 해결하였다.

메시지 1 출력
3 초간 대기
메시지 2 출력
3 초간 대기
.....

이런 식으로 메시지 출력 후 시간을 지연시켜 주는 함수(delay 함수)를 사용하면 간단하게 문제를 해결할 수 있다. 그러나 윈도우즈에서는 보편적으로 이런 방법을 사용하지 않는다. 왜냐하면 윈도우즈는 멀티 태스킹 시스템이고 여러 프로그램이 시간을 쪼개어 공동으로 사용해야 하기 때문이다. 그래서 대기 루틴 동안 CPU 가 아무 일도 하지 못하도록 묶어버리는 지연 코드는 사용할 수 없으며 반드시 타이머를 사용하여 꼭 필요한만큼만 CPU 를 사용해야 한다.

9jang
timelbl

메시지를 화면으로 출력해 주는 예제를 작성해 보자. 폼에는 타이머와 레이블 하나를 배치하고 레이블의 Font 속성을 적당하게 조절한다.




타이머의 Interval 속성은 문자열 출력 후 대기할 시간만큼 설정해 주며 이 프로그램에서는 3000, 즉 3 초 간격을 사용하고 있다. 완성된 소스 코드는 다음과 같다.

```
var
  Form1: TForm1;
  t:integer;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  t:=0;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
```

```
Inc(t);
case t of
  1:label1.caption:='3 초 시간 간격으로 문자열을 출력합니다.';
  2:label1.caption:='타이머를 사용하여 t 변수를 계속 1 씩 증가';
  3:label1.caption:='시키며 t 변수에 따라 출력하는 메시지를';
  4:label1.caption:='바꾸어 줍니다.';
  5:label1.caption:='프로그램을 종료합니다.';
  6:close;
  end;
end;
```

OnTimer 이벤트를 사용하는 것은 시계의 경우와 동일하되 이벤트마다 출력해 줄 문자열이 다르므로 이벤트간의 순서를 정하기 위해 정수형 변수 t 를 전역으로 선언하였다. 그리고 이 변수를 OnTimer 이벤트가 발생할 때마다 1 씩 증가시키며 변수값을 참조하여 적당한 문자열을 출력한다. 특별히 어려운 부분은 없을 것이다.

이 예제는 앞서 만들어 본 시계 예제와는 다른 각도에서 타이머를 이용하고 있다. 시계는 어떤 일을 규칙적으로 반복하기 위해 타이머의 주기를 이용하고 있지만 이 예제는 주기 외의 시간을 다른 프로그램에 양보하기 위해 타이머를 사용하고 있다.

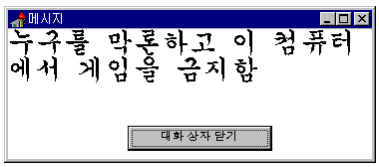
9jang timemes

다음 예제는 일정 시간동안 메시지를 보여 준 후 프로그램을 종료한다. 타이머를 적절하게 사용하여 잠시동안 메시지를 보여 주고 알아서 없어지는 대화상자를 만들고 있다. 사용하는 논리는 앞의 예제와 완전히 동일하다. 이 프로그램의 경우 시간에 상관없이 대화상자를 닫을 수 있도록 별도의 닫기 버튼을 제공한다.

```
var
  Form1: TForm1;
  t:integer;

implementation

{$R *.DFM}

procedure TForm1.Timer1Timer(Sender: TObject);
begin
Inc(t);
if t>5 then close;
```

```
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
t:=0;
end;

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

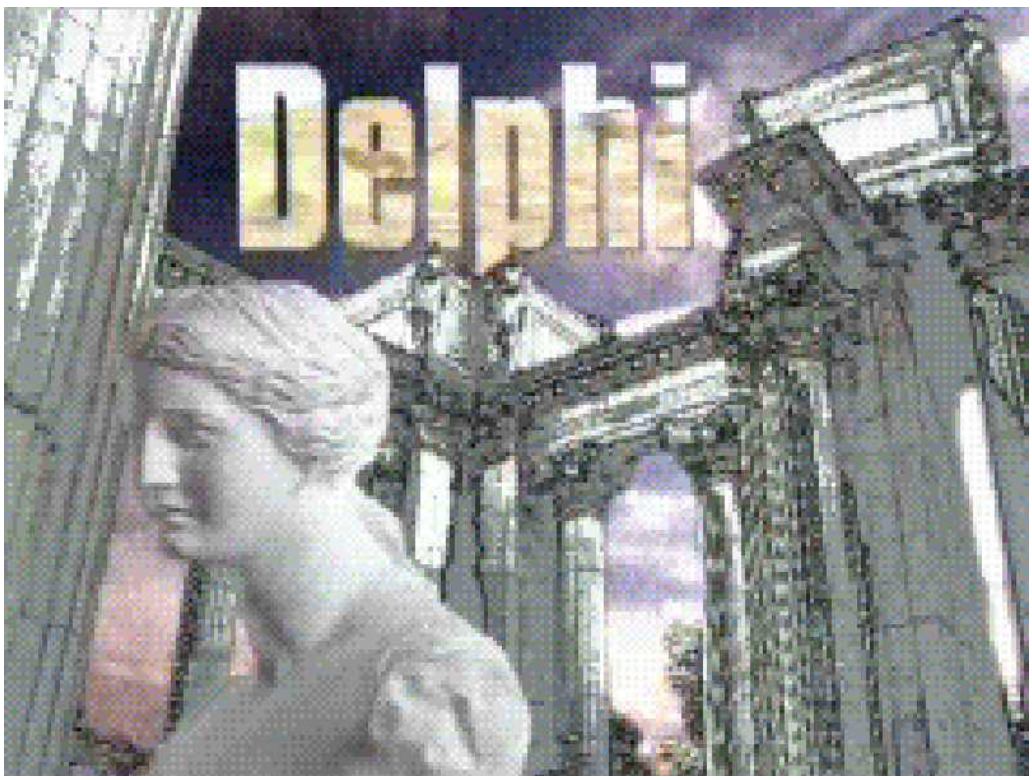
프로그램 실행중의 모습은 다음과 같다. 자신의 컴퓨터를 사용하는 다른 사람에게 알려줄 내용이나 주의사항이 있으면 이런 간단한 프로그램을 작성하여 시작 프로그램 그룹에 등록시켜 두면 좋을 것 같다.




이 외에도 타이머를 사용하는 예는 여러 가지가 있다. 어떤 특정한 조건을 계속적으로 검사해야 할 경우, 그래픽 애니메이션, 시스템 속도에 무관한 실행 속도를 유지하고자 할 경우 등이 있으며 이 중 몇 가지는 앞으로의 예제에서 직접 실습해 볼 것이다.

# 제 10 장

# 파스칼 고급 문법



6장에서는 아주 기본적인 파스칼 문법을 살펴보았다. 6장에서 논한 문법은 아주 기초적인 문법이기 때문에 프로그래밍을 조금이라도 해 본 사람은 대부분 쉽게 이해할 수 있는 내용이다. 그러나 그 정도의 문법만 가지고 델파이를 제대로 사용하기란 어렵다. 여기서는 좀 더 고급적이고 저수준적인 델파이의 문법에 관해 알아본다. 다소 따분하고 어려운 내용일 수도 있지만 고급스러운 프로그래밍 기법을 구사하고 싶다면 여기서 논하는 문법에 대하여 상세히 알고 있어야 한다.

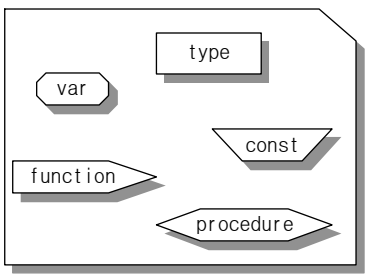
# 10-1 유닛

델파이는 파스칼에 기반을 둔 언어이며 파스칼 프로그램은 유닛으로 구성된다. 고급적인 프로그래밍 기법을 쓰고 싶다면 유닛을 피해갈 수는 없다.

## 가. 정의

일단 외형적으로 구분되는 유닛은 폼 하나에 하나씩 생성되는 확장자가 PAS인 파일을 말한다. 유닛(Unit)을 간단히 정의하여 문장화 한다면 다음과 같이 정의할 수 있다.

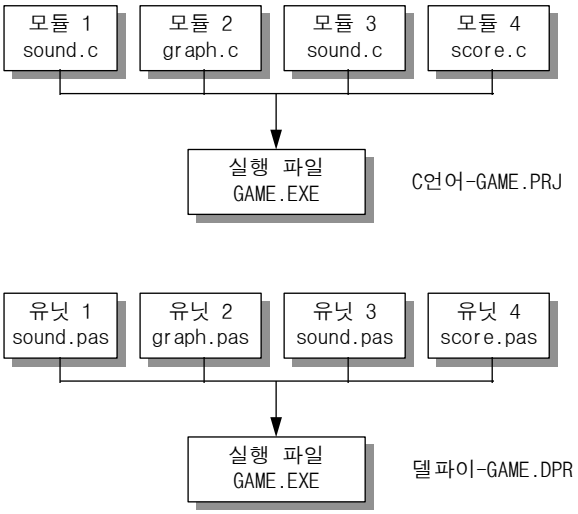
**유닛 : 상수, 변수, 타입, 서브루틴의 집합체**




파스칼 프로그램이 사용하는 여러 요소들을 포함하여 하나의 단위를 이루는 것이 유닛이다. 유닛은 다른 언어에서 말하는 모듈(Module)과 거의 동일한 개념이다. 규모가 큰 프로그램을 개발할 때 한 소스 파일에 모든 코드와 데이터를

다 포함하려면 소스 파일이 지나치게 커질 뿐만 아니라 컴파일 속도도 느려지고 메모리도 효율적으로 활용할 수 없다. 그래서 프로그램을 여러 개의 소스로 분할하여 따로 따로 컴파일한 후 링크 과정에서 합친다.

**그림**
모듈을 결합하여 프로그램을 작성한다.




델파이도 다른 언어와 마찬가지로 여러 개의 유닛을 작성한 후 유닛들을 모아 완전한 실행 파일을 만든다. 유닛 단위로 프로그램을 개발하면 다음과 같은 면에 있어서 여러 가지 이점이 있다.

**❶ 프로그램이 논리적으로 구성된다**

관련된 코드와 데이터를 하나의 유닛에 모아둠으로써 개발 과정에서 분업을 할 수 있다. 즉 그래픽을 잘 하는 사람이 graph.pas 에 그래픽과 관련된 코드만 작성하고 사운드 프로그래머는 sound.pas 만 작성한 후 합쳐서 하나의 프로그램을 완성할 수 있다.

**❷ 컴파일 속도가 빠르다**

소스 중 한 부분이 변경될 경우 소스 전체를 컴파일할 필요없이 변경된 유닛만 다시 컴파일하면 되므로 개발 속도가 빨라진다. 예를 들어 그래픽 부분만 변경되었다면 다른 유닛은 그대로 두고 graph.pas 만 다시 컴파일하면 된다. 만약 하나의 소스 파일로만 구성되어 있다면 일부만 변해도 전체를 다시 컴파일해야 할 것이다.

**❸ 코드의 재사용성이 높아진다**

유닛은 논리적으로 다른 유닛에 대해 독립성이 있다. 예를 들어 사운드를 처리하는 sound.pas 는 그래픽을 처리하는 graph.pas 의 코드와 특별한 연관성을 가지지 않는다. 그래서 일단 만들어진 유닛은 수정없이 또는 약간의 수정을 통하여 다른 프로그램에서 재사용할 수 있으므로 한번 작성한 코드를 재 사용하기가 쉽다.

델파이는 하나의 폼에 대해 개별적인 유닛을 생성하여 폼에 관련된 코드를 작성한다. 새로운 프로젝트를 시작하면 Form1을 생성하며 이 폼의 코드는 Unit1에 작성한다. 추가로 Form2를 작성하면 Unit2가 생성되며 폼의 개수만큼 유닛이 생성된다. 폼의 모양 및 폼에 포함된 컴포넌트의 속성 정의는 폼 파일(DFM)에 저장되며 폼과 관련된 코드(이벤트 핸들러)는 유닛 파일(PAS)에 보관된다. 그러나 유닛이 반드시 폼과 연관될 필요는 없으며 계산만 하는 수학적인 서브루틴이나 내부적인 메모리 처리를 하는 코드만 담은, 폼과는 전혀 상관없는 유닛을 만들 수도 있다.

# 나. 구성

유닛은 키워드 Unit과 유닛의 이름으로 시작된다. 유닛 이름은 변수나 서브루틴의 이름과 마찬가지로 명칭이므로 명칭 규칙에 적합하게 사용자 마음대로 정의할 수 있다. 영문자와 숫자로 구성되며 숫자를 첫자로 쓸 수 없고 파스칼 키워드를 쓸 수 없다. 게다가 여기서 정해준 유닛 이름은 유닛이 디스크에 보관될 때의 파일 이름이 되기 때문에 파일 시스템의 명칭 규칙에도 규제를 받게 된다. 윈도우즈 3.1에서 실행되는 델파이 1.0에서는 8자 이상의 유닛 이름을 사용할 수 없다는 규칙이 있었으나 32비트 델파이에서는 이 규정은 의미가 없어지게 되었다. 만약 16비트 버전과의 호환성을 유지해야 한다면 유닛 이름을 8자 이하로 유지해야 한다.

유닛이 디스크에 보관될 때의 확장자는 PAS이다. 사용자가 직접 코드를 편집하여 유닛 이름을 정하는 경우는 매우 드물며 유닛 파일을 디스크에 저장할 때 파일 이름을 지정함으로써 유닛 이름이 자동으로 결정된다. 유닛을 만들고 이 파일을 MyUnit.PAS라는 이름으로 저장했다면 이 유닛의 이름은 MyUnit이 된다. 유닛의 전체적인 구조는 다음과 같다.

```
unit 유닛 이름
interface
  uses ...
implementation
  uses ...
initialization
  초기화 코드
end.
```

유닛은 크게 interface부와 implementation부로 나누어지며 잘 사용되지는 않지만 initialization부와 finitialization부가 있다. 그리고 유닛의 끝은 반드시 end.으로 끝이 나야 한다. end 다음에 세미콜론이 아니라 마침표로 끝난다는 점을 유의깊게 보도록 하자.

## ■ interface부

파스칼의 유닛은 재사용성이 우수한 성질을 가지고 있다. 즉 이 프로그램에서 작성한 유닛을 다른 프로그램으로 가져다가 그대로 사용할 수 있다. 이런 재사용성을 실현하는 부분이 interface부이다. interface부에서 변수(var), 상수(const), 타입(type), 서브루틴(procedure, function) 등의 선언을 하면 이 유닛을 사용하는 다른 유닛으로 정보가 공개된다. 즉 interface부에 기입되는 내용은 "나는 이런 변수와 요런 서브루틴을 가지고 있습니다"라고 다른 유닛에게 알려주는 역할을 한다. 그래서 다른 유닛에서 이 유닛에 정의된 변수의 값을 읽거나 서브루틴을 호출할 수 있게 해준다. 바꾸어 말하면 다른 유닛과 공유해야 하는 변수나 서브루틴은 interface부에 선언해야 한다.

interface부에서는 선언만 할 수 있으며 직접 코드를 작성할 수는 없다. 서브루틴의 경우 헤더만 작성하여 유닛 내에 어떤 서브루틴이 있는지 정보만 공개할 뿐 구체적으로 어떤 동작을 하는 서브루틴인지를 밝히지는 않는다. 다만 유닛의 전체적인 구조가 어떠하며 어떤 구성 요소로 이루어져 있는지를 설명해 줄 뿐이다.

## ■ implementation부

interface부에서는 서브루틴의 헤더만 정의하며 실제적인 코드는 implementation부에서 정의한다. implementation부에서도 변수나 타입을 선언할 수 있지만 interface부에서 선언한 경우와는 달리 다른 유닛에게 알려지지

않으며 숨겨진다. 이 유닛에서만 사용할 변수라면 implementation부의 var 선언에서 변수를 선언하도록 한다.

왜 이렇게 정보를 숨기는가 하면 이후의 변경을 자유롭게 하기 위해서이다. interface부를 변경하면 그 유닛을 사용하는 다른 유닛도 영향을 받으므로 다른 유닛도 따라서 변경되어야 하지만 implementation부는 다른 유닛에게 알려지지 않았기 때문에 얼마든지 다른 유닛에 영향을 주지 않고 변경할 수 있다. 좀 더 효율적인 앨거리듬을 개발했다거나 내부적인 오류를 발견했을 때 유닛의 내부를 자유롭게 수정해도 다른 유닛에는 일체의 영향을 미치지 않게 된다.

## ■ initialization 부

유닛이 사용하는 데이터를 초기화하는 부분이다. 예를 들어 유닛에서 변수 score를 사용하고 이 값이 최초 1000으로 초기화되어 있어야 한다면 initialization부에 score:=1000; 이라는 코드를 두어 변수를 초기화한다. 여기에 기입된 코드는 유닛의 다른 어떤 부분보다도 먼저 실행되므로 유닛이 제대로 작동하기 위한 조건을 설정해야 한다면 여기에 그 코드를 기입한다. interface부나 implementation부는 반드시 있어야 하지만 initialization부는 불필요할 경우 없어도 상관이 없으며 보통 생략한다. 델파이에서는 관습적으로 변수의 초기화가 필요할 경우 initialization부를 사용하는 경우보다는 FormCreate 이벤트 핸들러를 더 많이 사용한다. 실제로 initialization부와 FromCreate 핸들러의 차이는 극히 미미하므로 폼이 없는 특수한 유닛을 제외하면 initialization부를 꼭 써야 할 경우는 별로 없다.

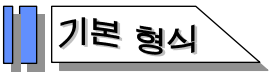
## ■ finitialization 부

원래 전통적인 파스칼 컴파일러에는 없던 것인데 델파이 2.0에서 추가된 부분이다. finitialization부의 코드는 initialization부와는 반대로 프로그램이 종료될 때 마지막 뒷정리를 하는 역할을 한다. 즉 할당해 놓고 해제하지 않은 메모리나 열어 놓고 닫지 않은 파일 등의 처리를 하는 부분이다. 1.0에서도 이런 작업은 ExitProc 등의 함수를 통해 가능하므로 굳이 finitialization부를 사용할 필요는 없다고 생각되며 또한 여기에 작성하는 코드는 세심한 주의를 해야 할 필요가 있기 때문에 함부로 사용하지 않는 것이 좋다.

이상으로 살펴본 바와 같이 델파이의 유닛은 4개의 부로 나누어지는데 실질적으로 마지막 둘은 거의 사용되지 않으며 이 책에도 사용하지 않고 있다. 앞쪽의 interface, implementation부만 신경 써도 된다.

# 다. uses절

한 유닛에서 다른 유닛에 정의되어 있는 변수나 서브루틴을 사용할 수 있다. UnitA와 UnitB가 있을 때 UnitA가 UnitB에 정의되어 있는 변수, 상수, 서브루틴을 마음대로 불러서 사용할 수 있다는 얘기다. 이때 어떤 유닛에 있는 코드와 데이터를 사용하겠다는 선언이 필요하며 이 선언이 uses절이다. uses 키워드 뒤에 사용하고자 하는 유닛의 이름을 콤마로 구분하여 나열해 준다.




uses 유닛1, 유닛2, 유닛3...;

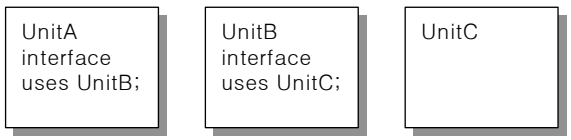
uses절은 유닛에 있는 코드와 데이터가 사용되기 전에 위치해야 하며 대개의 경우 interface부의 첫부분에 적어 주는 것이 제일 무난하다. 새로운 프로젝트를 시작할 때 델파이가 작성해 주는 코드를 보면 세 번째 줄에 다음과 같은 Uses절이 있다.

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

이 유닛들은 델파이가 제공하는 표준 유닛이며 델파이와 함께 제공된다. 이 유닛들 안에 우리가 흔히 사용하는 버튼이나 에디트 등등의 컴포넌트들이 정의되어 있으며 우리가 작성하는 유닛이 이런 표준 유닛을 사용함으로써 별도의 프로그래밍을 하지 않고서도 버튼이나 기타 컴포넌트를 마음대로 사용할 수 있는 것이다. 또한 이때까지 종종 사용해 왔던 IntToStr, ShowMessage 등의 프로시저들도 이 유닛들에 정의되어 있다. uses절은 interface부와 implementation부에 모두 올 수 있지만 의미가 조금 다르다.

## ■ interface 부의 uses 절

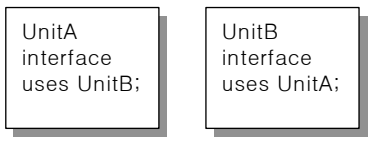
다른 유닛을 사용하고자 할 경우 여기에 유닛 이름을 기술한다. 특징으로는 중첩이 가능하며 상호 참조가 불가능하다. 중첩이 가능하다는 얘기는 UnitA에서 UnitB를 사용하고 UnitB에서 UnitC를 사용할 때 UnitA에서 UnitC도 사용가능하다는 얘기다.




UnitB 사용가능
UnitC 사용가능

UnitC 사용가능

UnitA에서 UnitC를 uses절에 써주지 않고 UnitB만 써주어도 UnitB에서 UnitC를 사용하므로 UnitA까지 UnitC의 함수와 변수를 사용할 수 있다. 어째 말이 좀 꼬이는 듯 하지만 그림을 보면 쉽게 이해가 갈 것이다. 중첩이 가능하기 때문에 다음과 같은 상호 참조는 불가능하다.



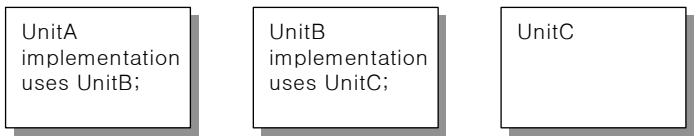

UnitB 사용

UnitA 사용

UnitA에서 UnitB를 사용하고 UnitB에서 또 UnitA를 사용하면 결국 중첩에 의해 UnitA가 자기 자신을 사용하는 꼴이 되고 중첩 횟수는 무한대가 되어 버리기 때문이다. 이런 구문은 컴파일조차 되지 않는다.

## ■ implementation 부의 uses 절

다른 유닛을 사용하고자 할 경우 여기에 유닛 이름을 기술한다는 점은 interface부와 동일하다. interface부의 uses와는 달리 중첩이 불가능하다.

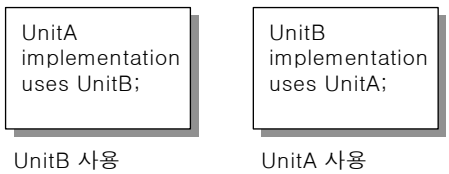



UnitB 사용가능
UnitC 사용불가

UnitC 사용가능

UnitB가 UnitC를 사용하고 있고 UnitB를 UnitA가 사용하고 있지만 UnitA에서는 UnitC를 사용할 수 없다. UnitB에서 uses UnitC 선언을 implementation부에서 했기 때문에 이 정보가 다른 유닛에게는 공개되지 않는다. 즉 UnitC는 UnitB에서 혼자 사용할 뿐이며 UnitA에게까지 UnitC의 정보는 제공되지 않는 것이다. UnitA에서 UnitC를 사용하려면 uses UnitC를 써 주어야 한다. 중첩이

되지 않으므로 상호 참조는 가능하다.




즉 UnitA에서 UnitB를 사용할 수 있고 UnitB에서 UnitA를 사용할 수 있다. 이렇게 되면 상호의 정보를 공유하게 된다.

# 라. 유닛의 예

실제로 유닛이 어떻게 작성되는지 그 예를 보자. 우선 새로운 프로젝트를 시작하면 델파이가 만들어 주는 유닛을 볼 수 있다. 유닛이라기 보다는 유닛을 만들기 위한 뼈대라고 해야 옳을 것이다. 설명을 위해 적당히 한글로 주석을 달아 보았다.

```
unit Unit1;   {유닛의 이름이 Unit1 이다.}

interface  {인터페이스부의 시작}

uses   {이 유닛에서 사용하는 외부 유닛}
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

  Forms, Dialogs;

type  {폼의 타입 선언}
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var   {폼 변수 선언}
  Form1: TForm1;

implementation  {임플리멘테이션부의 시작}
```

```
{$R *.DFM}
        {아직 아무런 코드도 작성되어 있지 않다.}
end.
```

우선 유닛의 이름이 Unit1으로 되어 있다. 이 파일을 다른 이름으로 저장하면 유닛의 이름도 따라 바뀔 것이다. interface부의 uses절에서 많은 유닛 이름을 나열해 놓았는데 이 유닛들은 파스칼 표준 함수와 컴포넌트 코드를 담고 있는 유닛이다. 이런 여러 가지 유닛의 코드와 데이터를 사용하기 때문에 컴포넌트만 척척 배치해도 프로그램을 만들 수 있다. uses절 다음에는 type 선언과 var 선언에 의해 폼을 만든다.

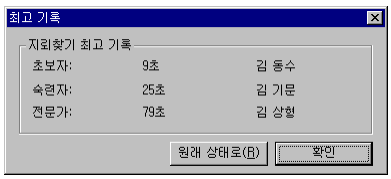
implementation부에는 아직 아무런 코드도 작성되어 있지 않다. {$R *.DFM}은 DFM 파일에 보관된 폼 리소스, 즉 폼과 폼에 포함된 여러 가지 컴포넌트의 속성을 읽어 오라는 컴파일러 지시자이지 코드는 아니다. 폼에 컴포넌트를 배치하고 이벤트 핸들러를 작성해 나가면 이 부분에 코드가 채워지게 된다. 특별히 초기화할 데이터도 없으므로 initialization부는 생략되어 있다. initialization부가 필요하면 직접 코드 에디터에서 end. 앞에 수작업으로 추가해 주어야 한다.


① 유닛:파스칼 프로그램의 모듈, 상수, 변수, 타입, 서브루틴의 집합체
② interface부:외부 유닛으로 공개될 정보가 기입된다.
③ implementation부:서브루틴의 코드를 작성하며 여기서 선언된 변수나 타입은 외부로 공개되지 않는다.
④ uses절:외부 유닛을 사용하겠다는 선언
⑤ interface부의 uses절과 implementation부의 uses절은 의미가 다르다.

# 10-2 프로젝트 옵션

## 가. 여러 개의 폼 사용하기

델파이에서 폼은 곧 윈도우이며 우리 눈에 보이는 실체이다. 새로운 프로젝트를 시작하면 빈 폼을 만들어 주어 윈도우를 디자인할 준비를 해주지만 실제로 하나의 윈도우만을 사용하는 프로그램은 극히 드물다. 보통 여러 개의 윈도우와 대화상자를 사용한다. 간단한 게임인 지뢰 찾기를 봐도 게임 폼 외에도 최고 기록을 출력 해주는 다음과 같은 대화상자가 있다.





10jang
about

그리고 거의 대부분의 프로그램이 간략한 프로그램 소개를 출력하는 About 대화상자를 출력해 준다. 여러 개의 폼을 만드는 기본적인 예제로 여기서는 About 대화상자를 출력하는 예제를 만들어 보자.

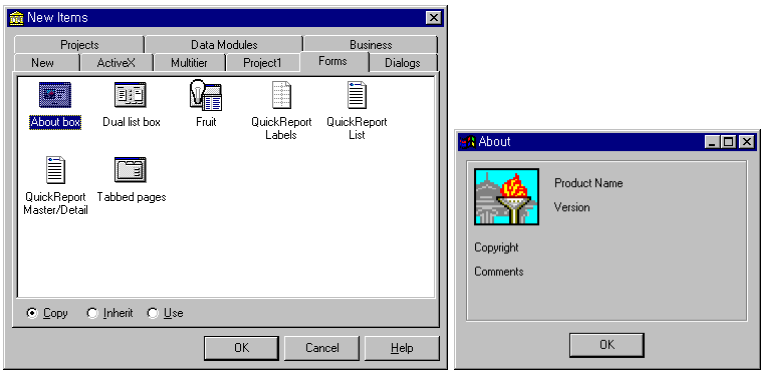
새로운 프로젝트를 시작하면 빈 폼만 열릴 것이다. 폼에 버튼 하나만 배치해 두고 이 버튼을 About 대화상자를 출력하는 명령 버튼으로 사용하기로 한다. 속성은 다음과 같이 설정한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | Name | MainForm |
| | Caption | Main Form |
| 버튼 | Name | BtnAbout |
| | Caption | About |

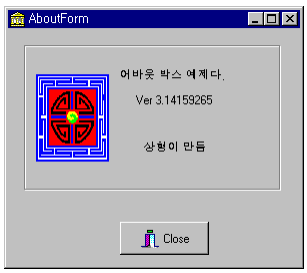
다음으로는 About 대화상자로 사용할 폼을 만들어야 한다. 프로젝트에 새로운 폼을 추가시키는 데는 세 가지 방법이 사용된다.

① File/New 를 선택한 후 오브젝트 창고에서 Form 을 선택한다.
② 스피드 버튼에서 (New Form) 버튼을 누른다.
③ 프로젝트 관리자에서 New 버튼을 누른 후 Form 을 선택한다.

어떤 방법을 선택하든지 결과는 마찬가지다. 좀 더 편한 방법을 사용하려면 File/New 항목을 선택한 후 오브젝트 창고에서 Form 페이지 탭을 선택한 후 미리 만들어진 About 폼을 복사해 오는 방법도 있다.




미리 만들어진 폼을 복사한 후 원하는대로 변경하면 시간을 절약할 수 있지만 여기서는 새로운 폼을 작성하는 실습을 하고 있으므로 빈 폼을 만든 후 직접 디자인해 보도록 하자.




꼭 폼의 모양이 이와 같을 필요는 없으며 자기 마음대로 디자인해도 상관없지만 대화상자를 닫을 수 있는 버튼 하나는 꼭 있어야 한다. 가장 편리한 방법으로는 비트맵 버튼을 배치하고 Kind 속성을 bkClose로 설정하는 방법이 있으며 이렇게 하면 별도의 코드를 작성하지 않아도 된다. 그리고 폼의 이름을 정해 주어야 메인 폼에서 이 폼을 참조할 수 있으므로 Name 속성에 AboutForm이란 이름을 준다.

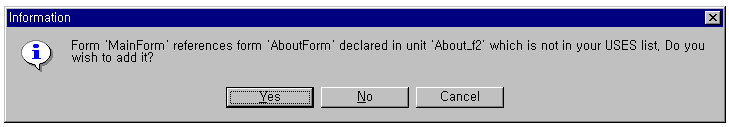
두 개 이상의 폼을 가지는 프로젝트는 우선 저장부터 해야 한다. 왜냐하면 폼

의 유닛 이름이 저장될 때의 파일 이름을 따라가기 때문이다. 유닛 이름이 먼저 정해져야만 uses절에 유닛 이름을 기입할 수 있고 그래야 유닛끼리 정보를 공유할 수 있다. 첫 번째 폼을 About_f.pas로, 두 번째 폼을 About_f2.pas로 저장하고 프로젝트는 About.dpr로 저장한다.

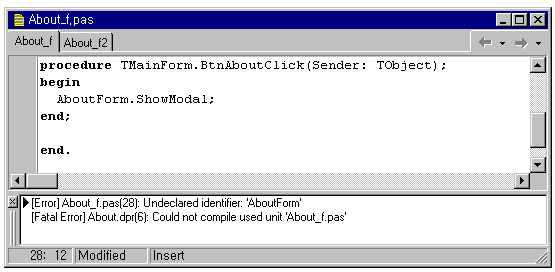
메인 폼에서 버튼을 누르면 About 대화상자가 보이도록 하기 위해 버튼의 OnClick 이벤트를 다음과 같이 작성한다.

```
procedure TMainForm.BtnAboutClick(Sender: TObject);
begin
  AboutForm.ShowModal;
end;
```

위 코드에서 사용된 ShowModal 메소드는 폼을 활성화하여 보이도록 한다. 즉 이 코드는 AboutForm을 활성화하여 화면에 출력한다. 그러나 아직까지 프로그램이 완성된 것은 아니다. F9 키를 눌러 이 예제를 실행시켜 보면 다음과 같은 에러 메시지가 출력될 것이다.




그대로 번역해 보면 "MainForm에서 참조하고 있는 AboutForm은 About_f2 유닛에 정의되어 있는데 uses절에는 이 유닛이 없습니다. 제가 넣어 드릴까요?"라는 질문이다. 사용자의 실수에 대해 델파이가 친절하게 알려주고 에러 교정까지 해주겠다는 것이다. 이 질문에 Yse라고 응답하면 만사 OK겠지만 일단은 뭐가 잘못됐는지 공부를 해야 하므로 No라고 대답해 보자. 그러면 AboutForm.ShowModal;행에서 Unknown Identifier라는 에러가 발생한다.



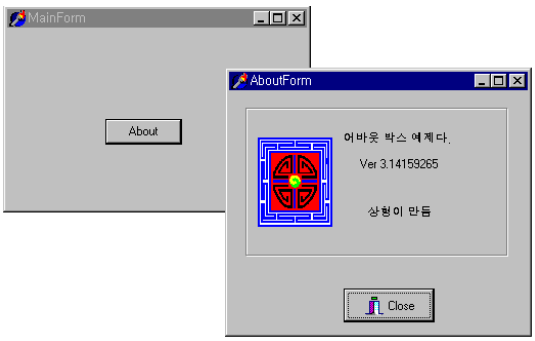

왜냐하면 메인 폼을 담고 있는 About_f 유닛에서는 About_f2 유닛에 정의되어 있는 AboutForm이라는 폼에 대한 정보를 전혀 가지고 있지 않기 때문이

다. 새로운 폼을 프로젝트에 포함시키는 일은 델파이가 자동으로 수행해 주지만 폼끼리 상대방을 인식하도록 하는 일은 사용자가 직접 해주어야 한다. 그래서 코드 에디터를 직접 열어서 About_f 유닛의 uses절에 다음과 같이 About_f2 유닛 이름을 기입해 준다.

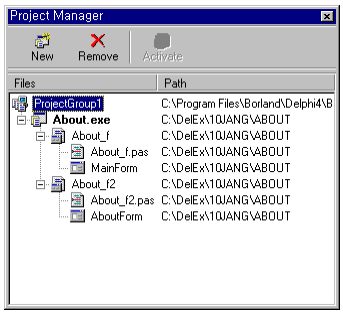
```
unit About_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, About_f2;
```

이로 인해 About_f 유닛에서는 About_f2 유닛에 정의되어 있는 AboutForm을 사용할 수 있다. 두 개 이상의 폼을 가진 프로그램을 만들 때는 폼끼리 상호 인식할 수 있도록 uses절에 참조하고자 하는 유닛의 이름을 기입해 주어야 한다. 이제 프로그램을 실행시키면 제대로 실행되고 버튼을 누르면 About 폼이 나타날 것이다.

그림
메인 폼에서 서브 폼을 호출한 모양




유닛의 이름과 폼의 이름은 다르다는 점을 주의하도록 하자. 폼의 코드는 유닛에 저장되며 폼 파일(DFM)의 이름도 유닛 파일의 이름을 따라가지만 폼의 이름과 유닛의 이름은 엄연히 다르다. About 예제의 프로젝트 관리자를 보면 다음과 같이 되어 있다.




About_f 유닛에 MainForm이라는 폼이 있고 About_f2 유닛에 AboutForm이라는 폼이 있다. 유닛의 이름은 유닛을 저장할 때의 파일 이름이며 폼의 이름은 오브젝트 인스펙터에서 설정한 폼의 Name 속성이다. 이 두 가지를 잘 구분하지 못하면 uses AboutForm;이라든가 About_f2.ShowModal; 등의 엉뚱한 실수를 하게 된다(필자처럼). uses 다음에는 유닛의 이름을 써 주는 것이고 ShowModal은 폼의 메소드임을 기억하자.

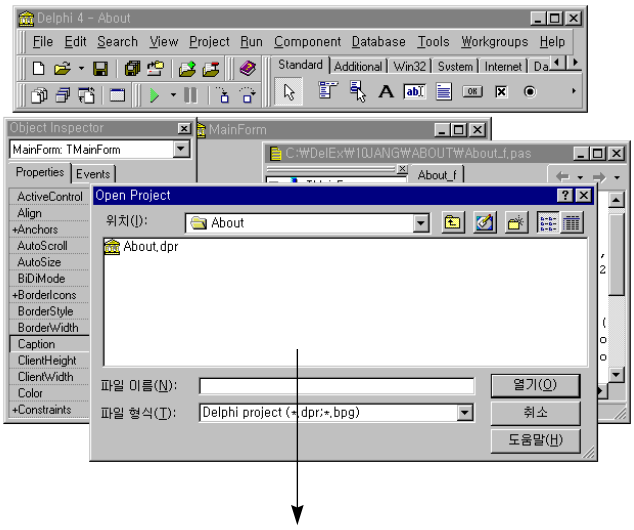
# 나. Modal 대화상자

윈도우즈가 제공하는 대화상자는 대화상자를 닫는 방식에 따라 크게 두 가지로 구분된다. 델파이에서 만드는 폼도 마찬가지로 두 가지 방식으로 호출한다.

■ **Modal 형**

대화상자가 닫히기 전에는 프로젝트 내의 다른 윈도우로 전환하지 못한다. 대화상자의 내용이 반드시 응답을 해야 할 경우나 대화상자가 굳이 열려 있어야 할 필요가 없을 때 Modal형 대화상자가 사용되며 일반적인 대화상자들은 모두 Modal형이다. 대표적으로 델파이의 Open Project 대화상자가 Modal형 대화상자이며 이 대화상자에서 프로젝트를 선택하거나 아니면 취소 버튼을 눌러 대화상자를 닫기 전에는 코드 에디터나 오브젝트 인스펙터 등의 다른 윈도우로 전환할 수 없다. 단 다른 프로그램으로는 전환할 수 있다.

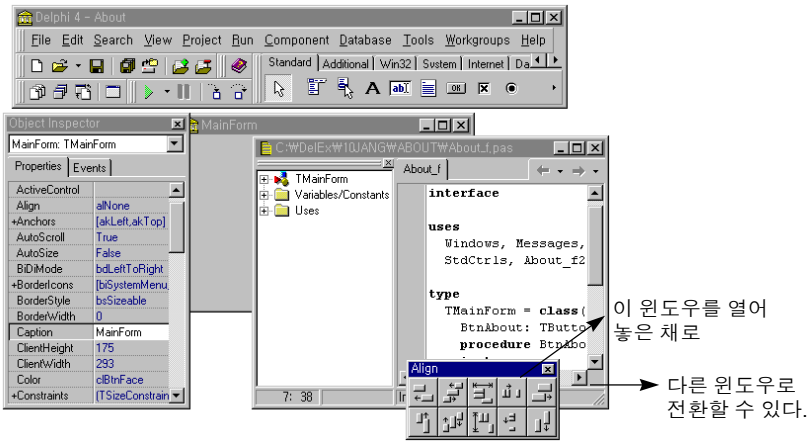
그림
모달형 대화상자




## ■ Modeless 형

대화상자가 열려 있는 채로 다른 윈도우로 전환이 가능한 형태이다. 프로그램의 상태를 나타내거나 작업을 하면서 참조해야 할 여러 가지 정보를 보여주는 윈도우가 Modeless형으로 만들어진다. 델파이의 프로젝트 관리자 윈도우, 정렬 팔레트 윈도우 등이 Modeless형 윈도우이다.

그림
모델리스형 대화상자




대화상자를 어떤 형태로 만들 것인가는 대화상자의 기능에 따라 결정하며 대화상자의 형태를 결정짓는 것은 대화상자를 출력할 때 사용하는 메소드이다.

Show 메소드를 사용하면 Modeless형 대화상자가 되며 ShowModal 메소드를 사용하면 Modal형 대화상자가 된다. 앞에서 작성한 About 예제의 ShowModal 메소드를 Show로 변경하여 실행해 보면 두 메소드의 차이를 확연하게 알 수 있을 것이다.

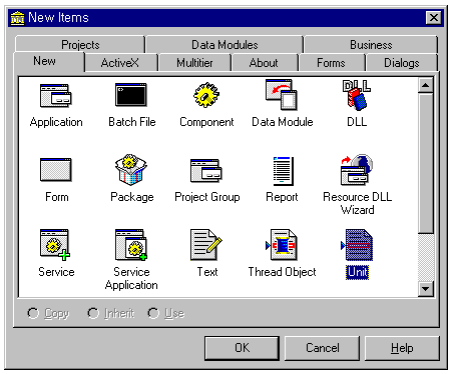
```
AboutForm.Show;     {모델리스형의 대화상자를 연다.}
AboutForm.ShowModal; {모달형의 대화상자를 연다.}
```

모달형으로 대화상자를 호출할 것인가 모델리스형으로 대화상자를 호출할 것인가는 전적으로 프로그래머 마음이지만 되도록이면 관습은 지키는 것이 좋다. 프로그램을 소개하는 About 대화상자를 모델리스형으로 호출했을 때의 모양을 상상해 보아라.

# 다. 폼 없는 유닛

델파이에서는 일반적으로 유닛 하나와 폼 하나가 대응된다. 하나의 유닛에 두 개의 폼이 있을 수 없으며 폼이 있으면 반드시 유닛도 있기 마련이지만 특수한 형태로 폼은 없지만 유닛만 있는 경우가 있을 수 있다. 폼이란 겉으로 드러나는 윈도우이고 유닛은 폼과 관련된 코드를 가지는데 윈도우 없이 코드만 필요한 경우에는 폼 없는 유닛이 사용된다.

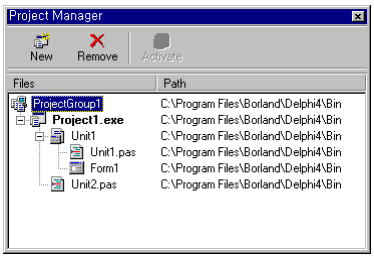
어떤 경우에 폼 없는 유닛이 필요한가 하면 복잡한 수학 처리를 담당해야 한다거나 소트, 정렬, 탐색과 같은 내부적인 데이터 처리를 담당하는 함수들을 담는 유닛이 필요할 때이다. 프로젝트에 폼 없는 유닛을 추가하고 싶으면 File/New 항목을 선택한 후 오브젝트 창고의 제일 마지막에 있는 Unit을 선택한다.



코드 에디터에 새로 만들어지는 유닛의 코드는 다음과 같이 간단한 모양을 가진다.

```
unit Unit2;

interface

implementation

end.
```

유닛 이름과 interface, implementation부의 이름 그리고 유닛의 끝을 나타내는 end. 밖에 없다. 완전히 골조만 갖춘 유닛이다. 델파이가 만들어 주는 유닛의 뼈대 안에 새로운 함수나 변수를 선언하고 코드를 작성하여 사용한다. 디폴트로 initialization부는 만들어지지 않지만 사용자가 직접 만들어 넣을 수도 있다. 폼 없는 유닛을 만들었을 경우 프로젝트 관리자를 보면 Unit2에는 대응되는 폼이 표시되어 있지 않다.




폼 없는 유닛을 사용하는 경우란 그리 흔하지는 않지만 대규모의 예제를 분석하다 보면 가끔 볼 수 있으며 프로젝트 전체에 걸쳐 사용되는 전역 변수나 서브루틴을 담기 위해 사용한다.

## 라. 폼 없는 프로젝트


10jang
beep

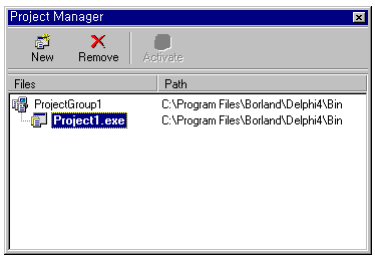
윈도우즈용 프로그램은 모두 윈도우를 가진다. 윈도우가 있어야 사용자에게 무엇인가를 보여줄 것이고 작업 지시를 받을 수 있기 때문에 지극히 당연한 일이다. 하지만 굳이 윈도우가 없는 프로그램을 만들고자 한다면 불가능한 것도 아니다.

## ■ BEEP.DPR

별로 쓸데는 없지만 실습을 해 보고 싶다면 다음 과정을 따라 폼 없는 프로젝트를 만들어 보자.

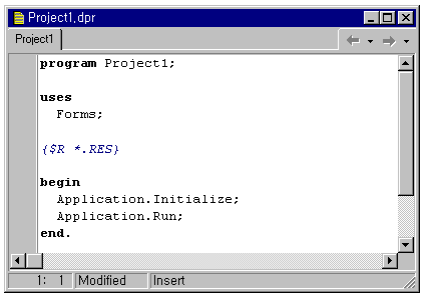
1 늘상 하던대로 File/New 를 선택해 새로운 프로젝트를 시작한다. 프로젝트가 만들어지고 깨끗한 빈 폼이 열릴 것이다.

2 View/Project Manager 항목을 선택해 프로젝트 관리자 윈도우를 연다. 그리고 Unit1 에서 오른쪽 마우스 버튼을 눌러 스피드 메뉴의 Remove From Project 항목을 선택한다. Unit1 을 저장할 것인가 물어보면 과감하게 No 버튼을 눌러 저장하지 않는다. 프로젝트 관리자가 다음과 같이 깔끔해질 것이다.




Project1은 이제 폼도 유닛도 없이 프로젝트만 있게 된다.

3 프로젝트의 소스를 불러온다. 프로젝트 관리자의 스피드 메뉴에서 View Source 를 선택한다. 코드 에디터가 열리고 DPR 파일을 보여줄 것이다.




4 DPR 파일 소스를 블럭으로 싸서 전부 지워 버리고 다음과 같이 입력한다.

```
program Beep;
```

```
uses
  windows;

begin
  MessageBeep(0);
end.
```

이 프로젝트를 BEEP.DPR 이라는 파일 이름으로 저장하고 컴파일 시키면 BEEP.EXE 라는 실행 파일이 만들어진다.

BEEP.EXE는 분명히 윈도우를 가지지 않으며 실행하면 "삑"하는 소리만 내고 종료하는 크기가 15872바이트밖에 안되는 아주 극단적으로 간단한 프로그램이다. 이 예제는 별 의미가 없는 것 같지만 델파이 프로그램의 구조에 관한 다음과 같은 아주 중요한 사항을 가르쳐 준다.

첫째, 델파이로 만든 프로그램의 메인 파일은 유닛 파일이 아니라 프로젝트 파일인 DPR 파일이다. 그래서 유닛이 만들어지기 전에 처리해야 할 일이 있다면 DPR 파일에 직접 코드를 작성해 넣을 수도 있다.

둘째, 델파이 프로그램의 Entry Point, 즉 실행 시작점은 DPR 파일의 begin 바로 다음 부분이다. C에서 프로그램 시작점이 main( ) 함수에서 시작되는 것과 비슷하다. DPR 파일의 begin 다음에서 폼을 생성해 주고 폼을 담고 있는 유닛으로 제어권을 넘겨 준다. 델파이가 생성해 주는 디폴트 프로젝트 파일을 잠깐 살펴보자.

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

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

begin~end 사이의 코드를 보면 OLE 오토메이션 등을 초기화하는 코드, 폼을 생성하는 코드와 폼으로 제어권을 넘기는 세 가지 일을 하고 있다는 것을 알 수 있다. DPR에서 사용하는 Application도 일종의 컴포넌트이며 고급 프로그래밍의 대상이 된다.

셋째, DPR 파일은 처음부터 끝까지 델파이가 책임을 진다. 폼을 추가하면 DPR 파일의 uses절에 유닛을 추가해 주고 begin 아래에 폼을 생성하는 코드도 추가해 준다. 그래서 사용자가 DPR 파일을 직접 건드릴 필요가 없다. 하지만 직접 건드릴 필요가 없을 정도까지 델파이가 서비스를 해 주더라도 직접 건드려야 할 경우도 있으며 직접 DPR 파일을 편집할 경우 몇 가지 고난도의 테크닉을 구사할 수 있다. 폼 없는 프로젝트가 하나의 예이다(비록 별 쓸데는 없지만).

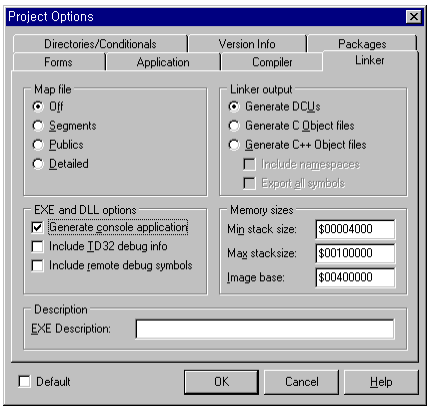
## ■ 콘솔 프로그래밍

10jang
project1

DPR 파일에 직접 프로그램을 작성하면 유닛을 구성하지 않고 도스에서 프로그램을 짜는 기분으로 프로그램을 짤 수도 있다. 물론 윈도우즈 환경에서 이런 프로그램을 짤 일이란 전혀 없겠지만 말이다. 앞에서 실습한 대로 프로젝트의 폼을 없애고 다음과 같은 DPR을 작성해 보자.

```
program Project1;

var
 i,j:Integer;
begin
for j:=1 to 20 do
  begin
  for i:=1 to j do
   Write('#');
  Writeln;
  end;
Readln;
Exit;
end.
```

그리고 Project/Options/Linker 탭에서 Generate console application 옵션을 선택해 준다.




실행해 보면 도스 창이 열리면서 다음과 같은 그림(?)이 그려질 것이다. 고전적인 도스 프로그램에서 루프의 개념을 가르치기 위해 종종 애용되는 예제이다.

```
#
##
###
####
#####
######
#######
########
#########
##########
###########
############
#############
##############
###############
################
#################
##################
###################
```

도스용 파스칼을 해 본 사람은 이런 예제를 보면 마치 고향에 온 듯한 느낌이 들 것이다. 폼 없는 프로젝트를 사용하면 옛날에 보던 향수어린 책들을 꺼내 앨거리듬 실습을 마음대로 할 수 있다. 도스 창에서 실행되는 이런 프로그램을 콘솔 프로그램이라고 하는데 그렇다고 해서 진짜로 도스 프로그램은 아니다. 다만 윈도우를 만들지 않는다 뿐이지 엄연히 Win32 프로그램이며 Win32의 모든

API를 다 사용할 수 있다.

## ■ C 스타일의 프로그래밍

10jang
Sdk

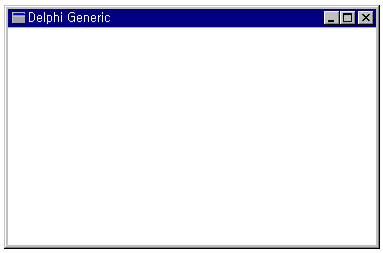
델파이는 윈도우즈 API를 완벽하게 지원한다. 그래서 VCL을 사용하지 않고도 프로그램을 작성하는 것이 가능하다. 새 프로젝트를 시작하고 폼을 제거한 후 프로젝트 파일에 다음 소스를 입력해 보자. 입력하는 것이 귀찮으면 배포 CD에 있는 프로젝트라도 불러와 실행해 보자.

```
program Sdk;

uses
  Windows, Messages;

const
  AppName = 'SDKPRG';

{윈도우 메시지 처리 함수}
function WindowProc(Window: HWnd; Mes, WParam,
  LParam: Longint): Longint; stdcall; export;
var
  AboutProc: TFarProc;
begin
  WindowProc := 0;  {리턴값은 0}
  case Mes of
    wm_LbuttonDown:
      begin
        MessageBox(Window, '마우스 왼쪽 버튼을 눌렀습니다','SDK',MB_OK);
      end;
    wm_Destroy:    {프로그램 종료}
      begin
        PostQuitMessage(0);
        Exit;
      end;
  end;
  WindowProc := DefWindowProc(Window, Mes, WParam, LParam);
end;

{메인 함수. 엔트리 포인트는 아님}
procedure WinMain;
var
  Window: HWnd;
  Mes: TMsg;
```

```
const
  WindowClass: TWndClass = (   {윈도우 클래스 정의}
    style: 0;
    lpfnWndProc: @WindowProc;
    cbClsExtra: 0;
    cbWndExtra: 0;
    hInstance: 0;
    hIcon: 0;
    hCursor: 0;
    hbrBackground: 0;
    lpszMenuName: AppName;
    lpszClassName: AppName);
begin
  if HPrevInst = 0 then  {이전 실행 인스턴스가 없어야 함}
  begin
    WindowClass.hInstance := HInstance;
    WindowClass.hIcon := LoadIcon(0, idi_Application);
    WindowClass.hCursor := LoadCursor(0, idc_Arrow);
    WindowClass.hbrBackground := GetStockObject(white_Brush);
    RegisterClass(WindowClass);
  end;
  Window := CreateWindow(   {윈도우 생성}
    AppName,
    'Delphi Generic',
    ws_OverlappedWindow,
    cw_UseDefault,
    cw_UseDefault,
    cw_UseDefault,
    cw_UseDefault,
    0,
    0,
    HInstance,
    nil);
  ShowWindow(Window, CmdShow);  {윈도우 보임}
  UpdateWindow(Window);
  while GetMessage(Mes, 0, 0, 0) do    {메시지 루프}
  begin
    TranslateMessage(Mes);
    DispatchMessage(Mes);
  end;
  Halt(Mes.wParam);
end;

{엔트리 포인트}
begin
  WinMain;   {메인 함수 호출}
```

```
end.
```

컴파일하고 실행해 보면 빈 윈도우만 하나 나타날 것이다. 가진 기능이라고는 마우스 왼쪽 버튼을 누르면 메시지 상자를 보여주는 것이 고작이다.



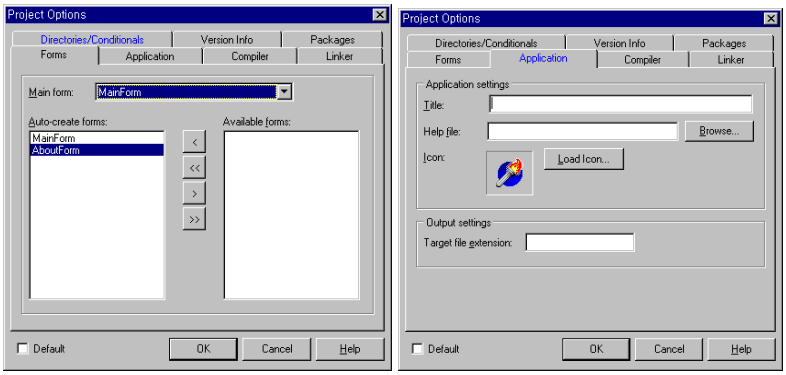

델파이를 처음 배울 때 만들었던 Fruit 예제보다도 더 간단하다. 그런데 이런 간단한 프로그램의 소스가 Fruit 예제보다 10배는 더 길어 보이고 문법도 굉장히 복잡해 보인다. 하지만 실행 파일 크기를 비교해 보면 오히려 거꾸로라는 것을 확인할 수 있을 것이다. Fruit.exe는 무려 284K인데 비해 Sdk.exe는 불과 16K에 불과하기 때문이다. Fruit 예제가 그렇게 간단하게 만들어지는 이유는 VCL이라는 엄청난 후원군이 뒤에 버티고 있기 때문이다.

이 프로젝트에서 사용한 문법은 Win32 API 문법 그대로이며 C 함수들을 델파이에 맞게 약간 손질한 것 뿐이다. API 프로그래밍을 해 본 사람들에게 이 정도 소스는 다 외우고 다닐 정도로 간단한 것이지만 그렇지 못한 사람에게는 무척 어려워 보일 것이다. 아뭏든 델파이로도 이런 저수준 프로그래밍이 가능하다는 것은 델파이가 고수준과 저수준을 얼마나 골고루 지원하는가를 잘 보여 주는 예이다.

# 마. 프로젝트 옵션

프로젝트 전반에 걸친 속성은 Project/Options... 항목을 선택하여 지정한다. 여러 개의 페이지가 있지만 컴파일러, 링커 옵션은 별도로 알아보기로 하고 다음 두 페이지에 관한 사항만 알아본다.

그림
프로젝트 옵션 대화 상자




### ❶ Main Form

메인 폼이란 프로그램을 실행시키면 제일 먼저 보이는 폼을 말한다. 디폴트로 가장 먼저 생성된 폼이 메인 폼이 되지만 드롭다운 리스트에서 메인 폼을 변경할 수 있다. 메인 폼으로 지정된 폼이 닫히면 프로그램을 종료한다.

### ❷ Auto-Created

프로그램을 시작하면 자동으로 생성되는 폼의 목록이다. 물론 여기에 목록이 있다고 해서 프로그램이 시작되자 마자 폼이 보이는 것은 아니다. 생성되는 것과 보이는 것은 다르기 때문이다. 생성한 폼을 보이게 하려면 Show 메소드를 사용해야 한다. 폼의 생성 순서가 중요한 의미를 가질 경우에는 리스트 박스에서 폼의 이름을 드래그하여 폼의 생성 순서를 조정할 수 있다.

### ❸ Available

자동으로 생성되지 않는 폼의 목록이다. 델파이는 프로젝트 내의 모든 폼을 자동으로 생성시키므로 최초 이 리스트 박스는 비어 있을 것이다. 그러나 메모리 문제라든가 속도 문제로 인해 최소한의 필요한 폼만을 생성하고, 그 나머지는 생성하지 않고자 할 때는 해당 폼을 이 목록으로 옮겨 자동 생성을 금지시킬 수도 있다. Auto-Created 리스트 박스에 있는 폼을 Available 리스트 박스로 옮기려면 가운데 있는 [>], [>>] 버튼을 사용한다.

### ❹ Title

폼을 최소화시킬 때 아이콘 밑에 나타날 제목을 지정한다. 255문자까지 가능하다.

### ❺ Help File

프로그램에서 사용할 도움말 파일을 지정한다.

**❻ Icon**

프로그램의 아이콘을 지정한다. 여기서 지정한 아이콘은 프로그램 관리자에 등록되는 아이콘이며 프로그램이 최소화 될 때의 아이콘이기도 하다. 메인 폼의 아이콘을 지정하지 않으면 이 아이콘이 대신 사용된다.

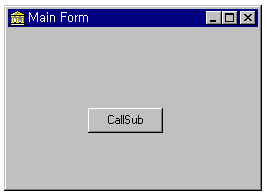
# 바. 폼 간의 정보 교환

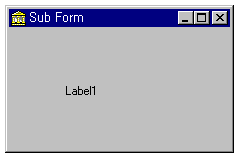
10jang
multi

하나의 프로젝트에 여러 개의 폼이 사용될 경우 문제가 되는 것은 폼 간의 정보를 공유하는 방법이다. 대표적으로 폼이 두 개 있는 경우를 직접 작성해 가며 이 문제를 해결하는 방법을 알아보자. 델파이 문법을 이해하는 데 비중이 큰 예제이므로 좀 귀찮더라도 처음부터 끝까지 예제를 따라 만들어 보고 두 번 더 만들어 보기 바란다. 아주 중요한 예제다. 새 프로젝트를 시작한 후 두 개의 폼을 만든다. 메인 폼에는 버튼을 배치하고 서브 폼에는 레이블 하나를 배치하고 속성을 다음과 같이 설정한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 메인 폼 | Name | MainForm |
| | Caption | Main Form |
| 버튼 | Name | BtnCallSub |
| | Caption | CallSub |
| 서브 폼 | Name | SubForm |
| | Caption | Sub Form |

그리고 폼의 크기는 너무 크면 징그러우니까 귀엽다는 생각이 들만한 크기로 줄인다. 속성을 조정한 후 두 폼의 모양은 다음과 같다.



메인 폼 서브 폼

속성을 지정한 후 메인 폼은 Unit1으로 서브 폼은 Unit2로, 프로젝트는 Multi로 이름을 주어 저장한다. 델파이 프로그램은 되도록이면 먼저 이름을 준 후 작업을 시작하는 것이 여러모로 유리하므로 새 프로젝트는 일단 저장부터 하는 것이 좋다.

메인 폼에 Score라는 정수 변수를 선언하고 5라는 값을 준 후 버튼을 누르면 서브 폼에서 메인 폼의 Score값을 읽어내는 시범을 보일 것이다. 메인 폼에서는 서브 폼을 인식해야 하고 서브 폼에서는 메인 폼의 변수를 읽어내야 하는 문제다. 우선 Unit1의 interface부에 Score라는 정수형 변수를 만든다. 다른 유닛에서도 이 변수를 사용하게 할 것이므로 변수 선언이 interface부에 있어야 하며 implementation부에 있어서는 안된다.

```
var
  MainForm: TMainForm;
  Score:Integer;
```

그리고 BtnCallSub 버튼의 OnClick 이벤트를 다음과 같이 작성한다.

```
procedure TMainForm.BtnCallSubClick(Sender: TObject);
begin
  Score:=5;
  SubForm.Show;
end;
```

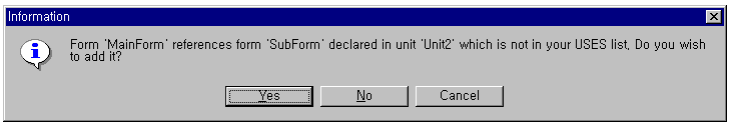
Score라는 변수에 5라는 값을 대입한 후 서브 폼을 호출하는 코드이다. 서브 폼이 호출되면 Score라는 값을 읽어 레이블에 그 값을 출력하고자 한다. 서브 폼의 OnShow 이벤트 코드를 다음과 같이 작성한다.

```
procedure TSubForm.FormShow(Sender: TObject);
begin
  label1.caption:='your score is '+ IntToStr(Unit1.Score);
end;
```

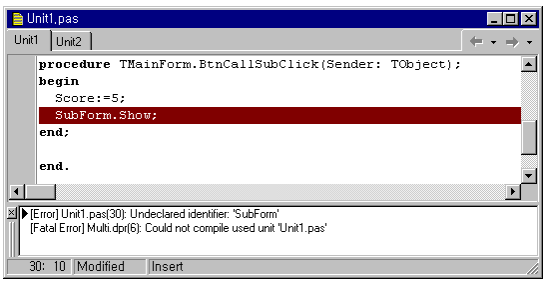

Unit2에서 Unit1에 선언되어 있는 변수 Score를 읽기 위해 Unit1.Score라고 이름을 적어 해당 변수가 어느 유닛에 있는 변수인가를 밝혀주어야 하는 것이 원칙이다. 그러나

uses에 의해 다른 유닛을 사용할 것을 선언했으므로 굳이 소속 유닛을 밝혀주지 않아도 상관없다. 위의 예에서 유닛 이름없이 Score만 기입해도 이 변수를 읽고 쓸 수 있다. 다만 같은 변수가 다른 유닛에도 있을 경우에는 소속 유닛 이름을 밝혀주어야 정확히 변수를 읽을 수 있다.

폼이 나타남과 동시에 Unit1에 정의되어 있는 Score 변수값을 읽어 레이블에 출력해 줄 것이다. Score라는 변수가 숫자이므로 곧바로 레이블로 출력하지는 못하고 IntToStr이라는 함수를 사용하여 문자열로 바꾸어 출력하였다. 과연 이 프로그램이 제대로 동작할 것인가 실행시켜 보자. F9키를 누르면 앞에서 About 예제를 만들 때 봤던 메시지를 보게 된다.




이 메시지는 우리가 해야 할 일을 대신해 주므로 아주 친절하기는 하지만 지금 당장은 공부에 방해가 되므로 당분간 무조건 No라고 대답하도록 하자. 이 질문에 Yes라고 대답해 버리면 델파이 혼자 북치고 장구치고 다 해 버리기 때문에 뭐가 틀렸는지를 알 기회를 뺏기기 때문이다. 컴파일은 고사하고 다음과 같은 에러 메시지가 출력된다.




버튼을 누를 경우 SubForm을 보이도록 할 생각이었는데 에러 메시지를 보면 SubForm이라는 명칭이 선언되어 있지 않다는 것이다. 왜 이런 에러 메시지가 출력되는가 하면 Unit1에서는 Unit2에 있는 폼의 이름을 알지 못하기 때문이다. 프로젝트에 새로운 폼을 추가하면 이 폼의 이름이 프로젝트의 uses절에 자동으로 추가되어 프로젝트는 새로운 폼(이 경우 SubForm)을 인식한다.

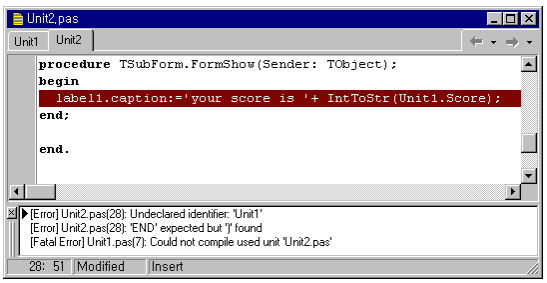
uses

```
Forms,
Unit1 in 'UNIT1.PAS' {MainForm},
Unit2 in 'UNIT2.PAS' {SubForm};
```

그러나 프로젝트만 새로운 폼을 인식할 뿐 개별 유닛들은 새로 생성된 폼을 전혀 인식하지 못한다. 그래서 유닛끼리 서로를 인식하도록 해주기 위해서는 유닛의 uses절에 사용하고자 하는 폼이 있는 유닛 이름을 적어주어야 한다. Unit1의 interface부에 있는 uses절의 끝 부분에 Unit2를 추가해 주고 다시 실행해 보자.

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Unit2;
```

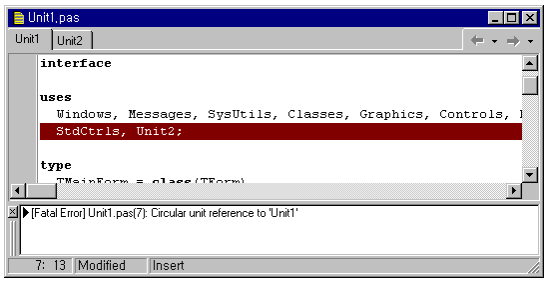
이렇게 적어 주면 Unit1에서 Unit2에 있는 폼을 사용할 수 있다. 그러나 이렇게 해 준다고 문제가 다 해결된 것은 아니며 계속 문제가 발생한다. 직접 실행해 보면 이번에는 Unit2에서 에러가 발생한다.




Unit1에서 Unit2의 SubForm을 인식하기는 하지만 이번에는 반대로 Unit2에서 Unit1에 있는 Score라는 변수를 인식하지 못한다. Unit2의 uses절에는 Unit1의 이름이 등록되어 있지 않기 때문이다. 그럼 이번에는 Unit2의 interface부에 있는 uses절에 Unit1의 이름을 추가해 보자.

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Unit1;
```

이제 Unit2에서도 Unit1을 인식할 수 있게 되었다. 이렇게 하면 될 것 같지만 또 에러가 발생한다.



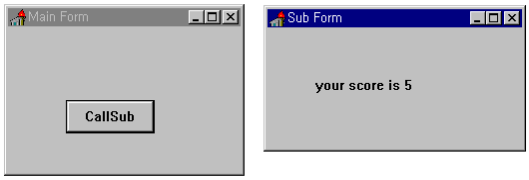

Unit1에서 Unit2를 참조하고 Unit2에서 Unit1을 참조하기 때문에 상호 참조 문제(Circular unit reference)에 걸린 것이다. interface부의 uses절은 중첩이 가능하지만 상호 참조는 불가능하다. 양쪽 유닛의 interface부에 있는 uses절에서 서로를 참조하여 발생한 에러이므로 이 문제를 해결하려면 다음 세 가지 방법 중에 하나를 사용한다.

① Unit1 의 uses Unit2;를 implementation 부로 옮긴다.
② Unit2 의 uses Unit1;을 implementation 부로 옮긴다.
③ Unit1 과 Unit2 의 uses Unit2; uses Unit1 을 모두 implementation 부로 옮긴다.

아무튼 양쪽의 interface부에서 서로 참조하지 않도록만 해 주면 문제가 해결된다. 어떤 방법을 쓰나 마찬가지이지만 ②번 방법을 사용하여 문제를 해결해 보도록 하자. Unit2의 interface부의 uses절에서 Unit1의 이름을 지우고 implementation부에 다음을 추가한다.

```
implementation

uses
 Unit1;
```

이제 프로그램을 실행하면 문제가 깔끔하게 해결되었을 것이다. 프로그램 실행중의 모습은 다음과 같다. CallSub 버튼을 누르면 서브 폼이 나타나며 서브 폼은 메인 폼의 Score 변수를 제대로 읽어낸다. 폼 간의 정보 교환에 성공한 것이다.




폼 간에 정보를 교환하여 사용하는 것은 언뜻 무척이나 복잡해 보이지만 interface부와 implementation부의 정의와 uses문의 정확한 사용 방법만 숙지하면 조금밖에 안 복잡하다. 두 유닛의 구조를 개략적으로 보이면 다음과 같다.

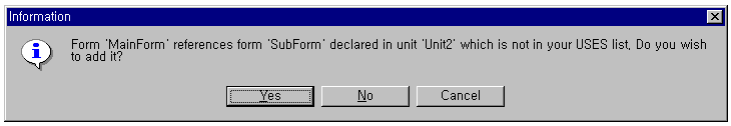
```
Unit Unit1;
interface
uses Unit2;
var
  MainForm:TMainForm;
  Score:Integer;
implementation
SubForm 참조 가능
```

```
Unit Unit2;
interface

var
  SubForm:TSubForm;

implementation
uses Unit1;
Unit1.Score 참조 가능
```

설사 세 개 이상의 유닛이 사용되더라도 이 정도만 알고 있으면 상호 정보를 공유하는데 큰 무리가 없을 것이다.


uses절에 유닛 이름을 많이 적는다고 해서 실행 파일의 크기가 커지는 것은 아니다. 델파이는 프로그램에서 사용하지 않은 함수의 코드는 알아서 실행 파일에 넣지 않기 때문에 사용하는 유닛이 많아도 상관없다. 다만 uses뒤에 유닛 이름이 많을 경우 컴파일 시간이 조금 느려질 뿐이다.

예제는 다 만들었는데 이 예제를 만들면서 보았던 다음 대화상자에 대해 잠깐 논해 보도록 하자.




이 대화상자는 어디가 어떻게 잘못되었는지뿐만 아니라 Yes 버튼을 누르면 직접 잘못을 찾아 고쳐주기까지 한다. 이제 여러분들은 이 메시지가 의미하는 바를 정확하게 이해했을 것이고 Yes 버튼을 눌렀을 때 델파이가 어떤 동작을 할 것이라는 것도 알 것이다. 잘 모르겠다면 이 예제를 다시 만들면서 직접 테스트해 보아라. 다음부터 uses 문에 유닛 이름을 빠뜨렸을 때는 이 질문에 Yes 라고 대답해 주도록 하자. 그러면 델파이가 알아서 빠진 유닛 이름을 넣어줄 것이다.

그러나 이런 자동화된 서비스가 마냥 좋기만 한 것은 아니다. 델파이는 무조건 implementation 부의 Uses 절에 유닛명을 삽입해 주는데 사용자의 의도와는 다를 수도 있으며 Unit2 에 Uses Unit1 이 빠진 경우는 이런 서비스를 베풀어 주지 않는다. 따라서 델파이의 서비스를 받기 전에 왜 이런 서비스를 베풀어 주며 어떤 효과가 있는지를 정확하게 이해하는 것이 중요하다. 잠깐 여담 하나 하자면 필자는 이 대화상자때문에 무척 고민을 많이 했다. "이렇게 하면 이러저러해서 안되니까 요렇게 해 줘야 합니다"라는 내용을 독자들에게 전달해 주고 싶은데 중간에 델파이가 툭 튀어나와 "괜찮아요. 제가 대신 해 줄께요" 하고 쓸데없이 참견을 하니까 말이다. 이럴 때는 정말 책 쓰기가 점점 더 어려워진다는 생각이 들곤 한다.

# 10-3 오브젝트

델파이는 완전한 객체 지향 프로그래밍(OOP:Object Oriented Programming)을 지원하는 언어이다. 사용하기 편리한 컴포넌트 기반의 개발 방식은 OOP 이론의 산물이라 할 수 있으며 이벤트 드리븐 방식도 OOP이론의 지원에 의해 부드럽게 조화되고 있다. 따라서 델파이를 제대로 사용해 보고자 한다면 OOP를 피해갈 수 없는 것이 사실이다. 그러나 OOP는 좋은만큼 개념을 익히기가 어렵고 특히 초보자에게 무척이나 난해하다. 여기서는 OOP 그 자체의 개념을 설명하는 데 초점을 두고 설명을 전개한다.

최대한 소스의 길이의 짧게 하려고 노력했지만 그래도 초보자가 보기에는 다소 이해하기 어렵고 직접 실습을 해 보기에도 웬만한 각오를 해야 할 정도이다. 그리 쉽지 않은 내용이므로 정신집중해서 잘 읽어보기 바라며 꼭 실습을 겸하기 바란다. OOP는 델파이뿐만 아니라 MFC, C++ 빌더, 자바 등 최신 개발 툴의 공통적인 이론적 기반이므로 배워두면 그만큼 가치가 있을 뿐만 아니라 또한 프로그래머를 꿈꾸는 사람이라면 반드시 마스터해야 할 부분이기도 하다. 하루종일(때로는 일주일 이상이 걸릴 수도 있다)을 할애하더라도 이 절의 내용은 반드시 익히기를 간곡하게 권하는 바이다.

## 가. 정의

### ■ 클래스

배열은 여러 개의 변수를 같은 이름으로 선언하고 첨자로 배열 요소를 선택할 수 있다는 점에서 아주 유용한 자료구조이다. 하지만 배열은 각 요소의 데이터형이 같아야만 한다는 제약이 있다. 즉 정수형끼리 모아 정수형 배열을 만들고 실수형끼리 모아 실수형 배열을 만들 수 있을 뿐 실수형과 정수형을 하나의 배열에 담을 수는 없다. 배열의 이런 단점을 극복한 자료구조가 레코드(record)이며 레코드는 다양한 형태의 변수를 하나의 이름으로 묶을 수 있도록 해 준다. 다음 레코드 선언을 보자.

```
type
  people=record
    name:string;
```

```
  age:integer;
  male:boolean;
end;
```

이렇게 데이터형은 다르지만 한 사람의 신상을 표현한다는 점에 있어서 논리적 연관성이 있는 이름, 나이, 성별에 관한 정보가 하나의 레코드로 묶여짐으로써 자료 표현에 있어서 상당한 편리함을 준다. 레코드가 배열에 대해 어떠한 장점을 가지는가는 직관적으로 이해가 될 것이다. 이런 레코드의 특징을 한단계 확장하면 여기서 우리가 연구해 보고자 하는 클래스(Class)가 만들어진다.

클래스란 여러 가지 타입이 다른 변수들을 포함한다는 면에서 레코드와 유사하다. 그러나 클래스는 변수만 포함하는 것이 아니라 서브루틴(function, procedure)도 같이 포함한다는 면에서 레코드보다는 더 확장된 개념이다. 클래스에는 클래스의 특성을 표현하기 위한 변수들이 포함되며 그 변수들을 관리하고 조작할 수 있는 서브루틴이 같이 포함된다. 하나의 자료 안에 데이터와 코드가 같이 포함됨으로 해서 클래스는 독립적으로 사용될 수 있으며 프로그램의 부품으로서 활용된다.

클래스는 일종의 데이터형이라고 할 수 있으며 실제로 문법적으로 데이터형으로 취급된다. 클래스형의 변수를 만들어 메모리 상에 구현시키면 이를 오브젝트(Object)라고 한다. 이를 우리나라 말로 번역하면 객체 또는 개체라고 하는데 객체라는 말은 잡지나 대중통신망을 통해 많이 들어 보았을 것이다. 클래스와 오브젝트, 이 두 용어의 구분에 대해서는 잠시 후에 다시 논해 보도록 하자.

## ■ 클래스의 선언

레코드가 record 키워드에 의해 선언되는 것에 비해 클래스는 class 키워드로 선언하며 클래스를 선언하는 기본 형식은 다음과 같다.

**기본 형식**

```
type
  클래스 이름=class
    필드들
    메소드들
```

```
end;
```

여기서 필드란 클래스에 포함되는 변수를 말하며 메소드(Method)란 클래스에 포함된 서브루틴을 말한다. 클래스를 선언한 예를 보자.

```
type
TMyClass=class
 X,Y:Integer;
 Col:TColor;
 procedure PutDot;
end;
```

X,Y의 정수형 변수 두 개와 Col이라는 색상을 기억하는 변수 그리고 PutDot라는 메소드를 가지는 TMyClass라는 클래스를 선언한 것이다. 이 오브젝트는 화면 상의 위치값과 색상값을 필드로 가지며 화면으로 점을 출력할 수 있는 메소드를 가짐으로써 완전히 독립적으로 존재한다. 독립적이라는 말은 오브젝트가 그 자체로써 어떤 기능을 수행할 수 있다는 얘기다.

만약 위치와 색상을 가지는 점을 레코드로 선언하고 이 레코드가 가진 점을 출력하는 함수를 만들었다면 그 레코드는 혼자서는 독립적이지 못하며 반드시 점을 출력해 주는 함수와 함께 동작해야만 한다. 반면 오브젝트는 내부에 데이터와 코드를 모두 가지기 때문에 혼자서 어떤 대상을 완벽하게 표현할 수 있으며 다른 프로그램으로 가져가 사용하기도 훨씬 수월하다.

**그림**
레코드와 오브젝트의 차이점


레코드 : 자료만 표현할 뿐 동작을 기술하지는 못한다.

오브젝트 : 자료와 동작을 모두 가지고 있다.

오브젝트 타입을 클래스라고 하며 클래스는 데이터 타입과 거의 동일한 말이다. 실제로 오브젝트 변수를 선언할 때도 var 선언에서 일반 변수를 선언하는 것과 동일한 방법으로 선언한다.

```
var
 i:Integer;     {정수형 변수 선언}
```

```
MyObj:TMyClass;  {오브젝트의 선언}
```

변수 i는 정수(Integer)형의 변수이며 오브젝트 MyObj는 TMyClass 클래스형의 변수로 선언되었다. 즉 Integer와 i의 관계가 TMyClass와 MyObj의 관계와 동일하며 곧 이 관계는 클래스와 오브젝트의 관계가 된다. 특정 클래스형의 오브젝트가 메모리에 실제로 만들어졌을 때 이를 클래스의 실체(instance)라고 하며 변수와 동등한 자격을 가진다. 오브젝트와 인스턴스는 실제로 같은 대상을 칭하는 용어이되 오브젝트는 독립적인 개체 자체를 칭할 때 사용하며 인스턴스는 메모리에 구현된 구체적 대상을 칭한다.

클래스의 이름도 일종의 명칭(Identifier)이므로 명칭 규칙에만 맞다면 사용자가 마음대로 작성할 수 있다. 그런데 델파이에서는 관행적으로 클래스 이름은 접두어 T자를 붙여 TCircle, TScreenRect, TBabo 등과 같이 선언한다. 여기서 앞에 붙인 T자는 타입(Type)을 의미하는데 이 말은 곧 클래스가 데이터 타입과 동등이라는 의미이다.

## ■ 클래스의 멤버 참조

오브젝트 변수의 필드를 사용하는 방법은 레코드의 필드를 사용하는 방법과 동일하다. 즉 변수 이름 다음에 점을 찍고 필드 이름을 적어 준다. 오브젝트의 메소드를 호출하는 방법도 필드와 동일하며 오브젝트가 계속 사용될 경우는 with를 사용하는 것도 가능하다.

```
MyObj.X:=45;          X 필드에 45를 대입한다.
MoObj.PutDot;         PutDot 프로시저를 호출한다.
```

MyObj.X는 MyObj 오브젝트에 속한 X 필드를 의미한다. 위 코드는 다음과 같이 쓸 수도 있다.

```
with MyObj do
begin
  X:=45;
  PutDot;
end;
```

## ■ OOP1

10jang
Oop1

그럼 클래스를 선언하여 오브젝트를 사용하는 간단한 예제를 만들어 보자. 문법 공부를 위해 코드를 작성하는 작업이므로 코드 에디터에서 대부분의 작업을 하게 될 것이다. 하지만 일단 결과를 확인해야 하므로 새 프로젝트를 만든 후 다음과 같이 버튼 하나를 배치해 두고 이 버튼의 OnClick 이벤트 핸들러에서 오브젝트를 만들어 보자. 그리고 실행 결과를 명확하게 살펴보기 위해 폼의 색상을 흰색으로 바꾸어 둔다.


작성한 소스 리스트는 다음과 같다. 일단 버튼을 더블클릭하여 OnClick 이벤트 핸들러를 만든 후 다음 코드를 죄다 입력하도록 하자. interface부의 타입 선언부에 있는 내용도 직접 입력해 주어야 한다.

```
unit Oop1_f;

interface

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

type
TMyClass=class   {클래스의 선언}
  X,Y:Integer;      {필드}
  Col:TColor;
  procedure Draw;   {메소드}
  end;
TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
```

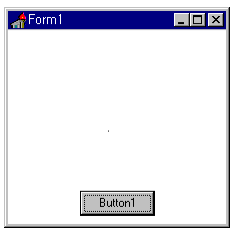
```
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.DFM}

procedure TMyClass.Draw;  {메소드 정의}
begin
Form1.Canvas.Pixels[X,Y]:=Col;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
 MyObj:TMyClass;      {오브젝트 변수 선언}
begin
MyObj:=TMyClass.Create;  {오브젝트 생성}
MyObj.X:=100;
MyObj.Y:=100;
MyObj.Col:=clBlack;
MyObj.Draw;           {메소드 호출}
MyObj.Free;           {오브젝트 파괴}
end;

end.
```

interface부의 type란에서 TMyClass를 선언하고 있는데 세 개의 필드와 하나의 메소드를 가진다. 클래스 선언문은 interface부에 있어도 되며 implementation부에 있어도 상관없는데 interface부에 있으면 다른 유닛에서도 이 클래스를 사용할 수 있으며 implemetation부에 있으면 이 유닛에서만 사용할 수 있다. 클래스의 메소드는 implememtation부에 그 본체를 작성해 주어야 한다. 위 소스에서 선언한 TMyClass는 Draw라는 메소드를 가지고 있는데 이 메소드의 본체가 implementation부에 정의되어 있다. 메소드의 본체를 보면 X,Y 좌표에 Col 색상으로 점을 찍도록 되어 있다.

클래스 선언과 메소드 정의를 해 주면 이제 이 클래스를 사용할 수 있다. 위 소스에서는 Button1Click 핸들러에서 TMyClass형의 오브젝트를 만들고 사용

한다. 클래스를 사용하려면 먼저 그 클래스형의 변수를 선언해야 한다. 선언하는 방법은 var 문에서 일반 변수를 선언하는 방법과 동일하다. Button1Click에서 TMyClass형의 오브젝트 MyObj를 지역 변수로 선언하였다.

클래스형의 변수를 선언했다고 해서 메모리에 실제로 오브젝트가 생성되는 것은 아니며 Create 메소드로 오브젝트가 사용할 메모리를 할당해 주어야 한다. 이때 사용된 Create 메소드는 델파이의 루트 클래스인 TObject로부터 상속받은 것인데 이에 대해서는 잠시 후에 다시 살펴보도록 하자. Create 메소드로 오브젝트를 생성하고, 이 생성된 오브젝트를 사용하면 된다. 이 예제에서는 필드에 값을 대입하고 메소드를 호출하여 화면에 점을 찍었다. 코드의 내용을 보면 좌표 필드는 100,100으로 설정하였고 색상은 검정색으로 설정하였으며 Draw 메소드로 점을 찍었다. 실행 결과는 다음과 같다.



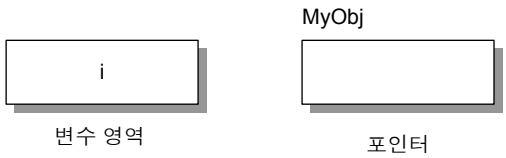

화면의 (100,100)에 검정색 점을 찍었는데 점이 작으므로 잘 봐야 겨우 보일 것이다. 마지막으로 오브젝트 사용 후 Free 메소드를 호출하여 오브젝트가 사용하던 메모리를 해제해 준다. 이로써 오브젝트를 만들고 오브젝트의 필드에 값을 대입한 후 메소드를 불러 사용해 보았다.

## ■ 클래스 레퍼런스 모델

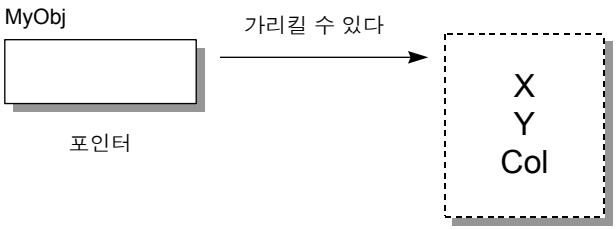
클래스는 데이터형이고 오브젝트는 변수라고 했다. 그런데 델파이에서는 일반 변수와 오브젝트 변수가 메모리에 구현되는 방식이 약간 다르다. 변수를 선언하는 다음 두 문장을 보자.

```
var
  i:Integer;
  MyObj:TMyClass;
```

둘 다 변수를 선언하는 문장이되 하나는 일반 정수형 변수를 선언했으며 하나는 클래스형의 오브젝트 변수를 선언했다. 그런데 이 선언문의 효과가 동일하지 않은데 그림으로 비교해 보자.

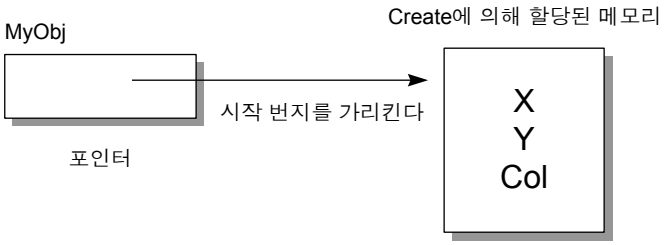



정수형 변수 i 를 선언하면 실제로 정수를 담을 수 있는 변수 영역이 메모리에 할당되며 i 에 값을 대입할 수 있다. 즉 일반 변수는 선언하는 즉시 사용할 수 있다. 그런데 오브젝트형 변수는 오브젝트의 크기만큼 메모리가 할당되는 것이 아니라 오브젝트를 가리킬 수 있는 포인터만 대신 할당된다. 오브젝트 자체가 메모리에 생성되는 것이 아니기 때문에 오브젝트 변수는 선언한다고 해서 곧바로 사용할 수 있는 것이 아니다. MyObj 가 선언된 직후의 상황을 그림으로 그려보면 다음과 같다.




X,Y,Col 필드를 포함하는 오브젝트는 메모리에 아직 없는 상황이며 이 오브젝트를 가리킬 수 있는 포인터만 만들어져 있을 뿐이다. 실제로 오브젝트에 메모리를 할당하기 위해서는 클래스의 Create 메소드를 호출해 주어야 한다. Create 메소드는 오브젝트의 크기만큼 메모리를 할당하며 그 번지를 리턴해 준다. MyObj:=TMyClass.Create 에 의해 다음 상태가 된다.

**그림**
오브젝트의 생성




이 상태에서 X,Y, Col 필드가 실제 메모리에 존재하므로 액세스가 가능하다. OOP1 예제에서는 필드에 값을 대입한 후 Draw 메소드를 호출하였으며 Draw 메소드는 이 필드값을 참조로 하여 점을 찍었다. Create 가 할당하는 메모리 영

역은 힙이기 때문에 사용한 후에 Free 함수를 호출하여 반드시 메모리를 해제해 주어야 한다. 오브젝트의 포인터인 MyObj 는 지역 변수로 선언되었으므로 함수가 종료될 때 자동으로 삭제된다.

이상에서 보다시피 오브젝트형의 변수를 사용할 때는 다음 절차대로 사용해야 한다.

```
var
 변수:클래스;
begin
 변수:=클래스.Create;
 사용;
 변수.Free;
end;
```

또는 오브젝트 변수를 전역으로 선언해 두고 FormCreate 에서 생성하고 FormDestroy 에서 파괴하는 원론적인 방법을 쓸 수도 있다.

델파이가 오브젝트 변수를 관리하는 이런 방법을 오브젝트 레퍼런스 모델(Object Reference Model)이라고 한다. 요약하자면 클래스형의 변수를 선언하면 포인터만 만들어 주며 실제 인스턴스 생성은 Create 메소드를 통하는 방법이다. 최근 세간의 화재가 되고 있는 자바도 이 모델을 사용하며 이 외에도 몇몇 OOP 언어들도 이 모델을 따른다.

반면 좀 더 대중적인 언어인 C++은 이와는 다른 모델을 사용하고 있는데 C++ 코드를 보도록 하자.

```
class MyClass {
 멤버 선언;
};
MyClass MyObj;
MyObj 사용;
```

이렇게 하면 MyObj 는 곧 MyClass 형의 변수(포인터가 아닌)가 되며 변수 선언 후 별도의 생성 절차없이 곧바로 오브젝트 변수를 사용할 수 있다. 델파이의 오브젝트 레퍼런스 모델을 C++로 흉내를 내 본다면 다음과 같이 작성할 수 있다.

```
class MyClass {
 멤버 선언;
};
```

```
MyClass *MyObj;
MyObj=new MyClass;
MyObj 사용;
delete MyClass;
```

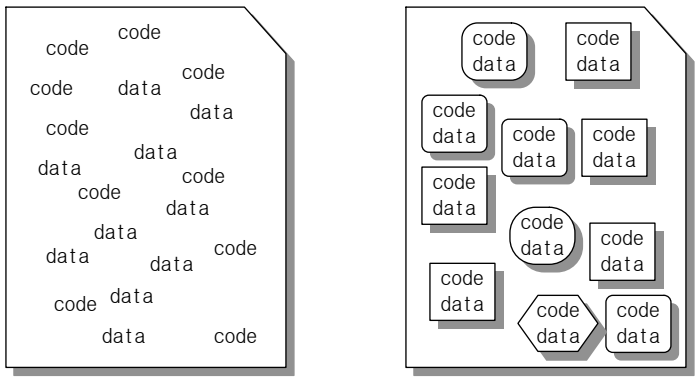
델파이는 오브젝트 변수를 선언할 때 내부적으로 위와 같은 코드를 컴파일해 준다. C++과 델파이의 이런 두 모델은 어떤 것이 더 우월하다고 하기 보다는 각각 장단점을 가지고 있다. 오브젝트 레퍼런스 모델은 메모리 사용면에 있어서 좀 더 유리하고 인수로 전달할 때 속도가 빠른 장점이 있는 반면 생성, 파괴를 직접 해 주어야 하므로 좀 귀찮은 면이 있으며 실수의 가능성이 존재하는 단점이 있다.

# 나. 캡슐화

오브젝트는 코드와 데이터를 한꺼번에 가짐(Encapsulation)으로써 독립적으로 동작할 수 있으며 오브젝트를 모아 프로그램을 완성시킨다. 프로그램은 코드와 데이터로 구성되며 기존 프로그래밍 방식에서는 코드와 데이터를 모아 프로그램을 만들었다. OOP 언어는 이와는 달리 코드와 데이터를 모아 오브젝트를 만들고 오브젝트를 모아 프로그램을 만든다.

**그림**

절차형 프로그래밍과 OOP 프로그래밍의 차이점




기존 프로그래밍 방식 OOP 프로그래밍 방식

일단 오브젝트를 한번 만들어 놓으면 오브젝트만으로 독립성을 가지며 다른 프로그램에도 수정없이 오브젝트를 그대로 재사용할 수 있으므로 프로그램의 생산성을 높여준다. 델파이가 사용하는 버튼, 에디트, 리스트 박스 등의 컴포넌트가 모두 오브젝트이며 이 오브젝트에는 코드와 데이터가 캡슐화되어 있기 때

문에 얼마든지 재사용할 수 있다. 앞에서 우리가 만들어 보았던 TMyClass도 비록 간단하지만 위치와 색상을 가지고 스스로 점을 출력할 능력을 가졌다는 면에서 완전한 독립성을 가진 오브젝트로 인정된다.

## 다. 생성자

10jang
Oop2

Create 메소드 호출에 의해 오브젝트가 메모리를 할당받고 생성되더라도 오브젝트의 필드는 여전히 초기화되어 있지 않다. 그래서 오브젝트 생성 후 필드에 초기값을 대입해 주는 일을 반드시 해 주어야 한다. 오브젝트를 사용하려면 생성 후 어차피 필드에 의미있는 값을 대입해 주는 초기화 과정이 뒤따라야 한다. 그래서 오브젝트 생성과 초기화 과정을 통합하여 필드 초기화를 생성자에서 할 수 있다. 생성자는 오브젝트를 생성하고 필드를 초기화하는 특별한 목적의 메소드이다.

생성자는 예약어 constructor로 시작되는 프로시저(=메소드)이며 필요할 경우 인수를 전달받는다. 생성자 내부에서는 기반 클래스(Inherited)의 생성자를 먼저 호출해 주어 메모리부터 할당받아야 하며 그리고 자신의 필드를 초기화한다. 다음 예제는 필드 대입을 생성자 내부로 옮긴 것이다.

```
unit Oop2_f;

interface

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

type
TMyClass=class
  X,Y:Integer;
  Col:TColor;
  constructor Create(iX,iY:Integer;iCol:TColor);
  procedure Draw;
  end;
TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
```

```
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.DFM}

procedure TMyClass.Draw;
begin
Form1.Canvas.Pixels[X,Y]:=Col;
end;

{생성자:인수를 전달받아 필드를 초기화한다}
constructor TMyClass.Create(iX,iY:Integer;iCol:TColor);
begin
{기반 클래스의 생성자를 먼저 호출해야 한다.}
Inherited Create;
X:=iX;    {자신의 필드를 초기화한다.}
Y:=iY;
Col:=iCol;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyObj:TMyClass;
begin
{생성자에 의해 오브젝트를 생성하면서 동시에
필드를 초기화한다.}
MyObj:=TMyClass.Create(100,100,clBlack);
MyObj.Draw;
MyObj.Free;
end;

end.
```

TMyClass 클래스에 생성자 Create를 정의하며 이 생성자는 세 개의 인수를 전달받아 TMyClass의 필드에 초기값을 대입해 준다. 그래서 오브젝트를 생성시킬 때 Create 뒤에 필드의 초기값을 인수로 나열하여 객체 생성과 동시에 필드에 초기값을 대입하였다. 이렇게 되면 필드에 별도의 값을 대입해 줄 필요없이

생성자의 인수로 초기값을 넘겨주기만 하면 되므로 사용하기가 훨씬 더 편리해진다.

생성자는 필드 초기화를 주로 하지만 이 외에도 오브젝트가 동작할 수 있는 초기 환경을 만들어 주는 역할을 한다. 예를 들어 모뎀을 사용하는 오브젝트가 있다고 할 때 이 오브젝트의 생성자에서 모뎀을 초기화해 두면 오브젝트에서는 모뎀을 자유롭게 사용할 수 있게 된다. 오브젝트에서 필요로 하는 메모리가 있을 때도 생성자에서 메모리를 할당하게 된다.

생성자의 반대되는 함수를 파괴자라고 하며 예약어 destructor로 시작되는 프로시저이다. 생성자에서 오브젝트의 동작을 위해 특별한 초기화나 메모리 할당을 했을 때 종료 처리나 메모리 해제를 해 주는 일을 한다. 모뎀을 사용하는 오브젝트의 경우 생성자에서 모뎀을 초기화했다면 파괴자에서 모뎀을 파괴해 주어야 할 것이다. 생성자에서 메모리나 GDI 오브젝트 등의 자원을 할당했다면 이 자원을 해제하는 일도 파괴자에서 해 주어야 한다. 물론 특별한 종료 처리가 없다면 파괴자는 별도로 정의해 주지 않아도 된다.

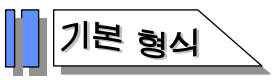
# 라. 상속

## ■ 상속

10jang
Oop3

이미 만들어져 있는 오브젝트를 조금만 수정하여 새로운 오브젝트를 만들어야 할 경우는 상속(Inheritance)이라는 방법을 사용한다. 새로 만들어지는 오브젝트는 이미 만들어져 있는 오브젝트의 필드와 메소드를 그대로 이어 받을 뿐만 아니라 자신만의 필드나 메소드를 추가할 수 있다. 이때 상속되는 오브젝트를 자식(Child)이라고 하며 상속을 해 주는 오브젝트를 부모(Parent)라고 한다.

다른 클래스로부터 상속을 하여 새로운 클래스를 만드는 방법은 class 다음에 괄호로 부모 클래스의 이름을 적어 주면 된다.




```
type
  클래스 이름=class(부모 클래스)
    필드들
    메소드들
```

```
end;
```

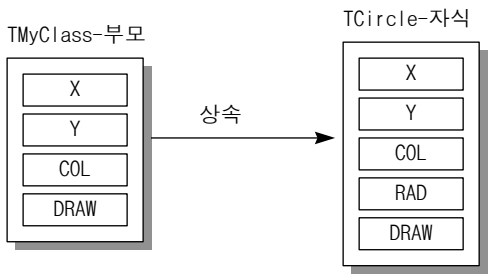
부모 클래스의 이름이 생략된 경우는 TObject를 부모 클래스로 하여 새로운 클래스가 만들어진다. TObject는 필드는 가지고 있지 않으며 몇몇 중요한 메소드(Create가 그 중의 하나이다)만을 가지고 있는 가장 기본이 되는 루트 클래스이다. 별도의 부모 클래스를 지정하지 않으면 자동으로 TObject가 부모 클래스가 되며 TObject의 필드와 메소드를 상속받게 된다. 앞에서 만들었던 OOP1 예제의 TMyClass 정의문을 다시 보면 부모 클래스가 지정되어 있지 않으며 따라서 이 클래스는 TObject로부터 상속을 받았다. 그래서 별도로 메소드를 정의하지 않아도 Create라는 메소드를 사용할 수 있었는데 이 Create 메소드가 TObject로부터 상속받은 메소드이다.

앞에서 만들었던 TMyClass를 상속하여 새로운 클래스 TCircle을 만들어 보자. 코드는 다음과 같다.

```
TCircle=class(TMyClass)
  Rad:Integer;
  constructor Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
  procedure Draw;
  end;
```

클래스 정의문을 보면 괄호안에 TMyClass가 기입되어 있어 TCircle 클래스가 TMyClass로부터 상속받음을 알 수 있다. 그림으로 이런 상속관계를 그려보면 다음과 같다.

**그림**
기존 클래스를 상속하여 파생 클래스를 만든다.




부모 클래스에 정의된 X,Y,Col 필드를 상속받으며 자신만의 필드를 추가로 선언하고 있다. 상속을 사용하는 간단한 예제를 보자. 소스 리스트는 다음과 같다.

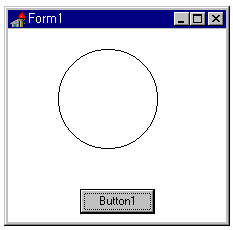
```
unit Oop3_f;

interface

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

type
TMyClass=class
  X,Y:Integer;
  Col:TColor;
  constructor Create(iX,iY:Integer;iCol:TColor);
  procedure Draw;
  end;
TCircle=class(TMyClass)
  Rad:Integer;
  constructor Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
  procedure Draw;
  end;
TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TMyClass.Draw;
begin
Form1.Canvas.Pixels[X,Y]:=Col;
end;

constructor TMyClass.Create(iX,iY:Integer;iCol:TColor);
begin
Inherited Create;
X:=iX;
Y:=iY;
```

```
Col:=iCol;
end;

constructor TCircle.Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
begin
Inherited Create(iX,iY,iCol);
Rad:=iRad;
end;

procedure TCircle.Draw;
var
  TempCol:TColor;
begin
TempCol:=Form1.Canvas.Pen.Color;
Form1.Canvas.Pen.Color:=Col;
Form1.Canvas.Ellipse(X-Rad,Y-Rad,X+Rad,Y+Rad);
Form1.Canvas.Pen.Color:=TempCol;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyCir:TCircle;
begin
MyCir:=TCircle.Create(100,70,clBlack,50);
MyCir.Draw;
MyCir.Free;
end;

end.
```

TCircle은 부모인 TMyClass로부터 필드와 메소드를 상속받고 Rad라는 필드를 추가로 더 정의하였다. TMyClass는 화면 상의 위치와 색상만을 가짐으로써 점을 표현하지만 그 후손인 TCircle은 반지름을 가짐으로써 원을 표현할 수 있다. 이때 Draw라는 메소드는 부모 클래스로부터 상속을 받기도 했지만 자식 클래스에서 다시 정의하고 있다. 점을 찍는 코드와 원을 찍는 코드가 서로 다르기 때문에 Draw 메소드는 그대로 상속을 받을 수 없기 때문이다. 물려 받은 메소드를 변경하고자 할 경우는 이와 같이 같은 이름으로 메소드를 재정의하면 된다.

```
var
  MyCir:TCircle;
```

상속된 클래스의 변수를 위와 같이 선언할 수 있으며 상속에 의해 만들어진 오브젝트도 부모 클래스의 필드와 메소드를 사용할 수 있다. TCircle의 메소드인 Draw에서 원을 그릴 때 TMyClass로부터 상속받은 X,Y,Col 등의 필드를 아무런 제한없이 사용하고 있다. 실행중의 모습은 다음과 같다.




그렇다면 왜 상속이라는 기법을 사용하는가? 그 이유는 무엇보다 편리하기 때문이다. 기존의 프로그래밍 방식이라면 자료구조를 수정해야 할 필요가 있을 때 직접 자료구조를 수정하지만 OOP에서는 기존 자료를 상속한 후 원하는 바대로 필드나 메소드를 추가, 변경할 수 있다. 그래서 이미 만들어진 비슷한 클래스가 존재한다면 처음부터 만들 필요없이 기존 클래스를 상속받아 조금만 수정하면 원하는 클래스를 쉽게 만들 수 있다.

만약 상속이라는 기법이 없다면 버튼, 체크 박스, 라디오 버튼 등을 일일이 처음부터 다시 만들어야 한다. 또한 메인 메뉴, 팝업 메뉴 등도 틀린 부분보다 공통된 부분이 더 많음에도 별도로 컴포넌트를 제작해야 할 것이다. 델파이가 제공하는 컴포넌트의 집합체인 VCL은 이러한 상속에 의해 그 많은 컴포넌트를 무리없이 제공해 줄 수 있는 것이다.

## ■ 이차상속

10jang
Oop4

점을 나타내는 클래스인 TMyClass로부터 원을 나타내는 TCircle 클래스를 상속해 보았다. 이렇게 만들어진 TCircle 클래스에서 원의 시작 각도, 끝 각도를 기억하는 필드와 원을 이동시키는 메소드 등을 추가하여 원호를 나타내는 새로운 클래스를 상속할 수도 있으며 상속의 횟수에 제한이 있지도 않다. 즉 파생된 클래스로부터 새로운 클래스를 또 만들 수 있다는 것이다.

또한 한번 상속에 사용된 부모 클래스도 다른 클래스를 파생하기 위해 얼마든지 재사용할 수 있다. 다음 예제는 TMyClass로부터 TCircle, TLine을 파생시키고 TCircle로부터 TEllipse를 파생시킨 것이다. OOP3 예제를 복사한 후 버튼 두 개를 더 배치하고 코드를 입력해 보자.

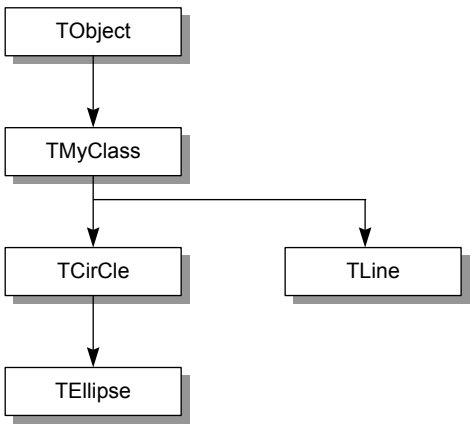
```
unit Oop4_f;

interface

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

type
TMyClass=class
  X,Y:Integer;
  Col:TColor;
  constructor Create(iX,iY:Integer;iCol:TColor);
  procedure Draw;
  end;
TCircle=class(TMyClass)
  Rad:Integer;
  constructor Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
  procedure Draw;
  end;
TLine=class(TMyClass)
  X2,Y2:Integer;
  constructor Create(iX,iY,iX2,iY2:Integer;iCol:TColor);
  procedure Draw;
  end;
TEllipse=class(TCircle)
  RadY:Integer;
  constructor Create(iX,iY:Integer;iCol:TColor;iRad,iRadY:Integer);
  procedure Draw;
  end;
TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TMyClass.Draw;
begin
Form1.Canvas.Pixels[X,Y]:=Col;
end;

constructor TMyClass.Create(iX,iY:Integer;iCol:TColor);
begin
Inherited Create;
X:=iX;
Y:=iY;
Col:=iCol;
end;

constructor TCircle.Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
begin
Inherited Create(iX,iY,iCol);
Rad:=iRad;
end;

procedure TCircle.Draw;
var
  TempCol:TColor;
begin
TempCol:=Form1.Canvas.Pen.Color;
Form1.Canvas.Pen.Color:=Col;
Form1.Canvas.Ellipse(X-Rad,Y-Rad,X+Rad,Y+Rad);
Form1.Canvas.Pen.Color:=TempCol;
end;

constructor TLine.Create(iX,iY,iX2,iY2:Integer;iCol:TColor);
begin
Inherited Create(iX,iY,iCol);
X2:=iX2;
Y2:=iY2;
end;

procedure TLine.Draw;
var
  TempCol:TColor;
begin
TempCol:=Form1.Canvas.Pen.Color;
Form1.Canvas.Pen.Color:=Col;
```

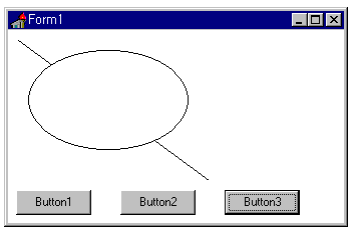
```
Form1.Canvas.MoveTo(X,Y);
Form1.Canvas.LineTo(X2,Y2);
Form1.Canvas.Pen.Color:=TempCol;
end;

constructor TEllipse.Create(iX,iY:Integer;
  iCol:TColor;iRad,iRadY:Integer);
begin
Inherited Create(iX,iY,iCol,iRad);
RadY:=iRadY;
end;

procedure TEllipse.Draw;
var
  TempCol:TColor;
begin
TempCol:=Form1.Canvas.Pen.Color;
Form1.Canvas.Pen.Color:=Col;
Form1.Canvas.Ellipse(X-Rad,Y-RadY,X+Rad,Y+RadY);
Form1.Canvas.Pen.Color:=TempCol;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyCir:TCircle;
begin
MyCir:=TCircle.Create(100,70,clBlack,50);
MyCir.Draw;
MyCir.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  MyLine:TLine;
begin
MyLine:=TLine.Create(10,10,200,150,clBlack);
MyLine.Draw;
MyLine.Free;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  MyEll:TEllipse;
begin
MyEll:=TEllipse.Create(100,70,clBlack,80,50);
MyEll.Draw;
```

```
MyEll.Free;
end;

end.
```

클래스 선언부가 무척 복잡하게 되어 있는 것 같지만 상속 관계는 금방 파악할 수 있을 것이다. 이 예제의 클래스 트리(계층도)를 그려보면 다음과 같다.




TLine 클래스는 선을 나타내는 클래스인데 선은 점에 비해 좌표가 하나 더 필요하므로 TMyClass 로부터 상속받은 후 X2, Y2 필드를 추가로 선언하고 있다. 타원은 원에 비해 X,Y 축 반지름이 구분되어야 하므로 Y 축 반지름 RadY 필드를 추가로 선언하였다. 이때 TCircle 클래스는 TMyClass 의 자식 클래스임과 동시에 TEllipse 의 부모 클래스가 된다. 세 버튼의 OnClick 이벤트 핸들러에는 원, 선, 타원을 각각 생성한 후 그리는 코드가 정의되어 있다. 프로그램 실행중의 모습은 다음과 같다.




클래스는 이런 식으로 얼마든지 상속이 가능하다. 그래서 광범위한 계층 구조를 이루게 된다. 델파이의 VCL 계층도나 비주얼 C++의 MFC 클래스 계층도는

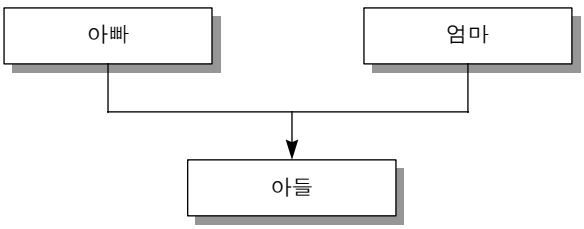
약 200~300 개의 클래스로 이루어져 있다.

### ■ 다중상속

OOP 의 또 다른 상속 기법으로 다중 상속(Multiple Inheritance)이라는 것이 있다. 이는 두 개 이상의 부모 클래스로부터 필드, 메소드를 상속 받는 것이다. 마치 아빠, 엄마의 특성을 빼 닮은 아기가 태어나는 것과 같다.

**그림**

두 개의 부모 클래스로부터 상속받는 다중 상속의 예




C++은 이런 다중 상속을 지원하지만 델파이는 다중 상속을 지원하지 않는다. 강력한 상속 방법이기는 하지만 이해하기 쉽지 않고 여러 가지 문제를 야기할 수 있기 때문에 C++에서도 다중 상속은 자주 사용되지 않는다. 그 예로 비주얼 C++의 MFC 의 경우 다중 상속을 전혀 쓰지 않고 있다. OOP 이론중에 이런 것도 있다는 것만 알아 두도록 하자.

## 마. 정보 은폐

오브젝트의 각 필드와 메소드는 가시성 지정(visibility specifier)을 가지며 이는 각 필드의 액세스 권한을 지정한다. 오브젝트가 만들어지면 오브젝트내에 속한 모든 필드(메소드)들은 프로그램의 어느 곳에서나 액세스 가능하지만 액세스 권한을 제한하면 일부 필드의 액세스를 금지시킬 수도 있다. 액세스 권한 지정자는 다음 네 가지가 있다.

**표**

액세스 권한 지정자

| 지정자 | 액세스 권한 |
|---|---|
| private | 오브젝트가 선언된 유닛 내부에만 알려지며 다른 유닛으로는 알려지지 않는다. 즉 외부 유닛에서는 이 속성을 가진 필드를 액세스할 수 없다. |
| public | 모든 유닛에서 이 필드를 액세스할 수 있는 공개된 필드이다. |

| | |
|---|---|
| protected | private와 마찬가지로 외부 유닛으로 알려지지 않는다. 단 상속을 할 경우 파생 클래스에서는 이 필드를 액세스할 수 있다. |
| published | public과 동일하되 RTTI(Run Time Type Information)을 작성하므로 디자인시에도 사용 가능하다. 속성과 이벤트들이 모두 published 액세스 권한을 가진다. |

액세스 권한 지정자는 오브젝트 타입 선언 내부에 사용되며 지정자 다음의 필드나 메소드의 액세스 권한을 변경한다. 다음 오브젝트 선언 예를 보자.

```
type
TMyClass=class
private
 X,Y:Integer;
public
 Col:TColor;
 procedure Draw;
end;
```

이 선언에서 X,Y 필드는 private로 선언되었으므로 다른 유닛에서 이 필드를 직접 읽거나 쓰는 것이 허용되지 않는다. MyUnit이라는 외부 유닛에서는 다음과 같은 코드를 쓸 수 없다.

```
var
 MyObj:TMyClass;     {오브젝트 선언}
 i:integer;
begin
 MyObj.X:=3;         {값을 쓸 수 없음}
 i:=MyObj.Y;         {값을 읽을 수 없음}
 MyObj.Col:=clRed;   {색상은 변경 가능하다.}
end;
```

TMyClass의 X와 Y는 private 액세스 권한을 가지며 외부 유닛으로는 알려지지 않기 때문에 마치 선언하지 않은 변수를 사용하는 것과 동일하다. 반면 Col 필드는 public으로 선언되었으므로 외부 유닛에서 얼마든지 액세스할 수 있다. 오브젝트가 이렇게 자신의 필드나 메소드 중 일부를 외부로 공개하지 않는 것을 정보 은폐(information hiding)라고 한다. 그렇다면 왜 이렇게 정보를 숨기려고 하며 정보를 숨길 때 어떤 이점이 있을까라는 의문을 가질 수 있다. 여러 가지 이유가 있지만 다음과 같이 요약할 수 있다. 컴퓨터의 키보드를 하나의 오브젝트에 비유하여 설명해 보기로 하자.

첫째, 오브젝트의 동작을 위해 사용되는 내부적인 변수를 보호하기 위해서이다. 만약 사용자가 오브젝트의 내부 변수를 마음대로 건드릴 수 있다면 이는 골치 아픈 버그의 원인이 될 수 있다. 키보드는 외부적으로 키(public 속성)를 가지고 사용자에게는 키만 공개되어 있으며 내부적인 배선이나 칩의 동작은 전혀 알려지지 않는다(private 속성). 그래서 사용자는 공개된 부분인 키만을 건드릴 수 있으며 사용자가 공개된 부분을 어떻게 조작하든 키보드가 이상한 동작을 하지는 않는다. 만약 칩이나 배선까지도 사용자에게 공개되어 있다면 이는 비정상적인 동작을 유발시킬 가능성을 제공하게 된다.

둘째, 사용자의 편의를 위해 불필요한 정보를 공개하지 않는다. 오브젝트를 사용하는데 있어 꼭 필요한 정보만 공개하며 나머지 정보는 사용자가 전혀 신경쓰지 않도록 해 주기 위해서이다. 키보드 사용자는 A 키를 누르면 A 문자가 입력되고 S 키를 누르면 S 키가 입력된다는 정보만 알면 그만이지 키보드 내부의 동작이 어떻게 되며, 왜 그런가에 대해 전혀 신경쓰지 않아도 된다.

세째, 정보를 숨겨두면 오브젝트를 쉽게 개선시킬 수 있다. 사용자는 오브젝트의 공개된 부분만 알고 있으므로 오브젝트를 내부적으로 수정하더라도 외부 사용자는 전혀 영향을 받지 않고 오브젝트를 사용할 수 있다. 오브젝트를 더 빠르고 안정적으로 동작하도록 하는 새로운 앨거리듬을 개발했을 때 숨겨진 부분만을 수정하면 된다. 키보드는 외부적인 기능을 유지한다는 조건하에서 내부적으로 키의 스위치 방식이나 칩, 배선 상태를 바꾸어도 사용자는 키보드 사용법을 다시 배울 필요가 없다. 그래서 정보 은폐는 사용자가 오브젝트에 관해 습득한 기존 지식을 최대한 보호해 주며 이는 곧 프로그래머의 인건비 절약과 생산성 향상으로 직결된다.

물론 정보를 은폐할 필요없이 모두 공개해 두고 사용자가 필요한 정보만 사용하도록 권유할 수도 있다. 하지만 생각해 보라. 정보가 숨겨진 상태에서도 델파이가 제공하는 200여 개의 컴포넌트, 오브젝트의 특성이나 사용법을 모두 알기란 정말 어렵다. 하물며 모든 정보를 다 공개해 놓으면 누가 그 컴포넌트들을 제대로 사용할 것인가? 꼭 필요한 정보만 공개하기 때문에 오브젝트가 독립적일 수 있고 프로그램의 부품이 될 수 있다.

# 바. 다형성

## ■ 포인터와 오브젝트

오브젝트 레퍼런스 모델에 의해 오브젝트 변수는 모두 포인터이다. 이 때 포인터끼리의 호환성 문제, 즉 대입 가능성 문제에 대해 연구해 보기 위해 다음 코드를 보자.

```
var
  MyDot:TMyClass;
  MyCir:TCircle;
begin
  MyDot:=TmyClass.Create(100,100,clBlack);
  MyCir:=TCircle.Create(100,70,clBlack,50);
  MyDot:=MyCir;  {가능}
  MyCir:=MyDot;  {불가능}
```

두 개의 오브젝트 MyDot 와 MyCir 이 생성되었는데 알다시피 둘 다 내부적으로는 포인터 변수이다. 그런데 부모 클래스형인 MyDot 로는 MyCir 오브젝트를 대입받을 수 있다. 왜냐하면 자식 오브젝트는 부모 오브젝트의 필드와 메소드를 모두 가지고 있기 때문이다. 이렇게 대입된 MyDot 로 MyDot.X, MyDot.Y 를 참조해도 MyDot 형인 TMyClass 에 있는 모든 필드는 실제로 MyCir 에 모두 존재하기 때문이며 동작에 아무런 이상이 없다.

반면 자식 클래스형인 MyCir 로는 MyDot 오브젝트를 대입받을 수 없다. 왜냐하면 부모 오브젝트는 자식 오브젝트가 가진 모든 필드와 메소드를 가지고 있지 않기 때문이다. MyCir.X, MyCir.Y 는 가능하지만 MyCir.Rad 를 참조하게 되면 이 필드는 실제로 대입받은 오브젝트인 MyDot 에는 존재하지 않기 때문이다. 이런 잠재적인 문제로 인해 자식 클래스형 변수에 부모 클래스 오브젝트의 대입은 허락하지 않는다.

요약하자면 부모 클래스형의 오브젝트는 자식 클래스형의 오브젝트를 대입받을 수 있지만 그 역은 성립하지 않는다. 이해가 잘 되지 않는다면 아예 외워 버려도 손해 보지 않을 문장이다.

## ■ 가상 메소드

10jang
Virt

가상 메소드에 대한 개념을 익히기 위해 TMyClass, TCircle 클래스가 선언되어 있는 OOP3 예제에 코드를 추가해 보자. 다음과 같이 DrawObj 라는 프로시

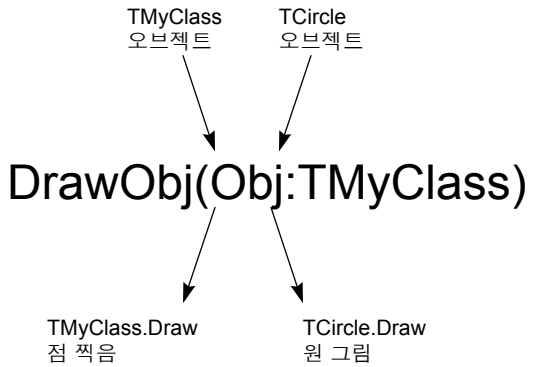
저를 폼에 추가했다.

```
procedure TForm1.DrawObj(Obj:TMyClass);
begin
  Obj.Draw;
end;
```

이 프로시저는 TMyClass 형의 오브젝트를 인수로 전달받아 이 오브젝트의 Draw 메소드를 호출해 준다. 인수 Obj 는 TMyClass 형이므로 이 클래스로부터 파생된 모든 클래스의 오브젝트를 대입받을 수 있다. 앞 항에서 본 대로 부모 클래스형의 오브젝트는 자식 클래스형의 오브젝트를 대입받을 수 있기 때문이다. 이제 DrawObj 로 어떤 오브젝트가 전달되는가에 따라 Obj.Draw 의 실제 동작이 달라질 것이다.

**그림**

인수로 전달되는 오브젝트에 따라 호출되는 메소드가 달라진다.




의도하는 바대로 되는지 테스트해 보기 위해 버튼 두 개를 배치하고 이 버튼의 OnClick 이벤트 핸들러를 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyCir:TCircle;
begin
  MyCir:=TCircle.Create(100,70,clBlack,50);
  DrawObj(MyCir);
  MyCir.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  MyDot:TMyClass;
```

```
  MyCir:TCircle;
begin
  MyDot:=TmyClass.Create(100,100,clBlack);
  DrawObj(MyDot);
  MyDot.Free;
end;
```

TCircle 형의 오브젝트와 TMyClass 형의 오브젝트를 만들어서 이 오브젝트를 DrawObj 함수의 인수로 넘겨 주었다. 이제 실행해 보면 위 그림에서 보인바 대로 Button1 을 누르면 원이 그려질 것이고 Button2 를 누르면 점이 찍혀야 될 것이다.

그런데 실행해 보면 전혀 그렇지가 못하다. Button1 을 누르면 (100,70)에 점을 찍고 Button2 를 누르면 (100,100)에 점을 찍을 뿐이다. 좌표는 분명히 생성자에서 초기화한 X,Y 의 좌표를 사용하지만 웬일인지 Draw 메소드는 점만 찍어댄다.

이렇게 되는 이유는 DrawObj 프로시저의 Obj.Draw 가 TMyClass.Draw 로 컴파일되었기 때문이며 그래서 Form1.Canvas.Pixels[X,Y]:=Col; 코드가 실행되는 것이다. Obj 인수가 TMyClass 형이므로 이 클래스의 후손 오브젝트를 다 대입받을 수는 있지만 Obj.Draw 는 무조건 TMyClass.Draw 가 되는 것이다. 이런 식으로 메소드의 실제 번지가 결정되는 것을 이른 바인딩(early binding) 또는 정적 바인딩(static binding)이라고 한다. 컴파일할 때 이미 Obj.Draw 의 대상 번지가 결정되어 버렸기 때문에 Obj 인수로 어떤 오브젝트가 전달되는가는 전혀 영향을 미치지 못하는 것이다.

그렇다면 이 문제를 해결하기 위해 어떤 방법을 사용해야 할까. 근본 문제는 Obj.Draw 의 번지가 컴파일할 때 결정되어 버리는 정적 바인딩에 있으므로 이를 금지시켜야 하며 이렇게 하려면 Draw 메소드를 가상 메소드(Virtual Method)로 선언하면 된다. 가상으로 만들려는 메소드의 선언문에 virtual 이라고 써주며 이를 상속받은 클래스에서는 override 를 적어주면 된다.

```
type
TMyClass=class
  X,Y:Integer;
  Col:TColor;
  constructor Create(iX,iY:Integer;iCol:TColor);
  procedure Draw;virtual;
  end;
TCircle=class(TMyClass)
  Rad:Integer;
```

```
constructor Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
procedure Draw;override;
end;
```

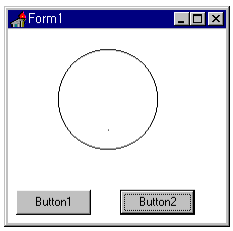
이 선언에 의해 Draw 메소드는 가상 메소드가 되며 실제 번지가 실행 중에 결정되는 동적 바인딩(dynamic binding)을 사용하게 된다. 따라서 Obj.Draw 호출문은 실행중에 Obj 가 어떤 클래스형인지를 파악하여 TMyClass 형이면 TMyClass.Draw 를 호출하여 점을 찍게 되고 TCircle 형이면 TCircle.Draw 를 호출하여 원을 그리게 된다. 이제 전체 소스를 보자.

```
unit Virt_f;

interface

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

type
TMyClass=class
 X,Y:Integer;
 Col:TColor;
 constructor Create(iX,iY:Integer;iCol:TColor);
 procedure Draw;virtual;
 end;
TCircle=class(TMyClass)
 Rad:Integer;
 constructor Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
 procedure Draw;override;
 end;
TForm1 = class(TForm)
   Button1: TButton;
   Button2: TButton;
   procedure Button1Click(Sender: TObject);
   procedure Button2Click(Sender: TObject);
   procedure DrawObj(Obj:TMyClass);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
```

```
 Form1: TForm1;
 MyObj:TMyClass;

implementation

{$R *.DFM}

procedure TMyClass.Draw;
begin
Form1.Canvas.Pixels[X,Y]:=Col;
end;

constructor TMyClass.Create(iX,iY:Integer;iCol:TColor);
begin
Inherited Create;
X:=iX;
Y:=iY;
Col:=iCol;
end;

constructor TCircle.Create(iX,iY:Integer;iCol:TColor;iRad:Integer);
begin
Inherited Create(iX,iY,iCol);
Rad:=iRad;
end;

procedure TCircle.Draw;
var
 TempCol:TColor;
begin
TempCol:=Form1.Canvas.Pen.Color;
Form1.Canvas.Pen.Color:=Col;
Form1.Canvas.Ellipse(X-Rad,Y-Rad,X+Rad,Y+Rad);
Form1.Canvas.Pen.Color:=TempCol;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
 MyCir:TCircle;
begin
MyCir:=TCircle.Create(100,70,clBlack,50);
DrawObj(MyCir);
MyCir.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
var
  MyDot:TMyClass;
begin
MyDot:=TmyClass.Create(100,100,clBlack);
DrawObj(MyDot);
MyDot.Free;
end;

procedure TForm1.DrawObj(Obj:TMyClass);
begin
Obj.Draw;
end;

end.
```

실행 결과는 다음과 같다. Button1을 누르면 원이 그려지며 Button2를 누르면 점이 찍힐 것이다.




## ■ 동적 메소드

컴파일러는 실행중에 가상 메소드의 번지를 찾을 수 있도록 하기 위해 프로그램의 선두에 가상 메소드 테이블(VMT:Virtual Method Table)이라는 일종의 표를 작성하며 가상 메소드 호출문은 이 표를 뒤져 원하는 메소드를 찾아 내게 된다. 표를 통해 함수의 번지를 찾으므로 호출 속도가 빨라진다는 장점이 있지만 반면 가상 메소드를 많이 사용하는 프로그램의 경우는 가상 메소드 테이블 자체의 크기가 무시못할 정도로 커지게 된다.

가상 메소드의 이런 단점을 개선한 것을 동적 메소드라고 하는데 동적 메소드는 테이블을 사용하는 대신 메소드에 고유 번호를 주고 번호로써 메소드의 번지를 찾는 방법을 사용한다. 동적 메소드를 선언하려면 예약어 virtual 대신 dynamic을 사용하면 된다. 동적 메소드는 프로그램의 크기에 있어서는 가상 메소드보다 유리하지만 실행 속도에 있어서는 불리하다. 이런 차이점 외에 가상

메소드와 동적 메소드의 차이점은 전혀 없다.

# 사. C 언어와의 비교

스몰 토크와 같은 순수 OOP 언어는 모든 프로그래밍이 오브젝트 위주이기 때문에 오브젝트에 속하지 않으면 변수 하나, 함수 하나도 만들 수 없다. 이와는 달리 혼성 OOP 언어는 오브젝트를 사용하면서도 구조적 프로그래밍 방식을 허용하는 혼합적인 방식의 언어이며 C++이나 델파이가 사용하는 오브젝티브 파스칼이 여기에 속한다.

C++과 델파이가 사용하는 OOP도 각각 독특한 특징을 가지고 있으며 파스칼보다는 C++이 순수 OOP에 더 가깝다. 두 언어의 가장 큰 차이점은 다중 상속(multiple inheritance)의 지원 여부와 연산자 오버로딩을 들 수 있으며 이 외에도 세부적으로 많은 차이를 들 수 있다. 여기서는 델파이에서 사용되는 OOP 이론에 관해 아주 대충 알아보았지만 아마 OOP를 처음 대하는 사람은 거의 대부분이 제대로 이해하기 힘들 것이다. 이것은 아마도 필자의 무능력함이 근본 원인이겠지만 델파이로 OOP 이론을 설명하기에는 한계가 있기 때문이기도 하다.

우선 델파이와 같은 비주얼 개발 툴에서는 내부적인 코드의 동작을 일일이 독자들에게 보여주기가 참 어렵다. 사용자와 컴퓨터 사이에서 델파이가 너무 많은 일을 해 주고 있기 때문이다. 또한 델파이는 순수 OOP 언어가 아니며 VCL 계층은 객체 지향적으로 설계되어 있지만 사용자가 프로그래밍을 할 때는 객체 지향 이론을 몰라도 되도록 되어 있다. 전역 변수 선언을 허가하며 클래스에 속한 메소드가 아닌 독립적인 서브루틴을 사용할 수 있는 혼성 OOP 언어이므로 OOP 이론을 설명하기가 어렵다.

OOP에 관해 상세하게 알고 싶은 사람은 먼저 도스용 C++을 공부해 보기를 권한다. 도스는 운영체제 자체가 간단하기 때문에 기초적인 예제를 통하여 어떻게(How)뿐만 아니라 왜(Why)에 대해서도 쉽게 이해할 수 있다. C++은 파스칼보다 OOP에 더욱 가깝고 게다가 OOP를 설명하는 대부분의 책, 자료들이 모두 C++을 사용하고 있으므로 관련 자료가 풍부하다는 이점이 있다. 델파이를 공부하는 여러분에게 느닷없이 C++을 연구해 보라고 한다면 너무 무책임하고 비겁한 말일지도 모르지만 그 방법이 멀어도 확실한 길이다. 필자가 OOP 연구를 위해 권하는 컴파일러는 터보 C++ 1.0, 볼랜드 C++ 3.1이며 그 중에서도 특히 볼랜드 C++ 3.1이 가장 좋다. 최신의 비주얼 C++은 OOP를 기반으로 OOP 이론을 십분 활용하는 툴이기는 하지만 OOP를 배우기에는 부적당한 툴이다.

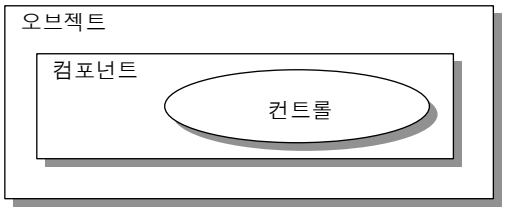
# 아. 오브젝트와 컴포넌트

오브젝트, 컴포넌트, 컨트롤의 세 가지 용어는 혼용되어 쓰이고 있다. 이 책에서도 물론이지만 많은 델파이 서적에서도 물론이고 델파이 도움말에서도 용어의 혼용은 여전하다. 어찌보면 무분별한 혼용이 아니라 문맥에 따라 가장 적절한 단어를 쓰고 있는 것이라 할 수 있지만 독자들에게는 무척 혼란스러울 것이다. 오브젝트와 컴포넌트 그리고 컨트롤은 비슷비슷하지만 분명히 다른 것이므로 어떻게 다른 것인가 구분이나 해 보도록 하자.

- 오브젝트 : 클래스가 메모리에 생성되어 있는 실체를 가리키는 문법적인 용어이다. 레코드가 확장된 개념이며 코드와 데이터가 뭉쳐져 있는 결합체를 오브젝트라고 한다.
- 컴포넌트 : 오브젝트 중 속성, 메소드, 이벤트를 가지며 어떤 특수한 목적에 사용될 수 있는 특성을 가지고 있는 것을 컴포넌트라고 한다. TObject 는 분명히 코드와 데이터를 가진 오브젝트이지만 그 자체로는 사용할 수 없으므로 오브젝트이기는 하지만 컴포넌트는 아니다. 폼이나 버튼, 타이머 등이 컴포넌트이다. 브러시(TBrush)나 펜(TPen)도 오브젝트이기는 하지만 홀로 사용될 수 없으므로 컴포넌트는 아니다. 오브젝트와 컴포넌트를 쉽게 구분하려면 디자인중에 사용할 수 있는가 없는가를 살펴보면 된다. 디자인중에 사용할 수 있으면 컴포넌트이고 코드를 통해서만 사용할 수 있으면 오브젝트라고 대충 구분할 수 있다. 그런데 예외적으로 TApplication 등 몇 가지는 디자인중에 사용할 수 없지만 컴포넌트라고 부른다.
- 컨트롤 : 컴포넌트 중에 화면에 보이는 것을 컨트롤(Control)이라고 하며 주로 사용자의 명령을 받아들이고 출력을 보여주는 역할을 한다. 버튼이나 에디트 박스, 리스트 박스, 레이블 등은 화면에 보이므로 컨트롤이라고 할 수 있다. 그러나 메뉴 컴포넌트, 타이머 컴포넌트는 컴포넌트이기는 하지만 실행중에 화면에 보이지 않으므로 컨트롤은 아니다. 컨트롤은 또 윈도우 컨트롤과 그래픽 컨트롤로 나누어지는데 이 구분은 다음에 알아보도록 하자.

세 가지 개념의 포함 관계를 그림으로 나타내 보면 다음과 같이 그릴 수 있다. 보다시피 오브젝트가 제일 넓은 의미를 포괄하고 있다.

그림
세 가지 용어의 포함관계




컴포넌트는 오브젝트의 일종이지만 오브젝트라고 해서 전부 컴포넌트인 것은 아니며 마찬가지로 컴포넌트라고 해서 꼭 컨트롤인 것은 아니다. 컴포넌트나 컨트롤을 일반적으로 칭할 때는 제일 큰 집합 이름인 오브젝트라고 할 수 있다.

버튼이나 에디트 박스를 칭할 때 어떤 때는 컨트롤이라고도 하고 어떤 때는 컴포넌트라고도 하며 또 어떤 때는 오브젝트라고 칭하기도 하여 상당히 혼란스러운 것 같지만 알고보면 모두 맞는 말이다. 문맥에 따라 사용되는 용어가 다르더라도 읽는 사람이 지혜롭게 판단해야 한다.

# 자. VCL 오브젝트

VCL은 델파이에서 사용되는 모든 오브젝트의 집합체이며 이 안에 우리가 이때까지 사용해 오던 버튼, 에디트, 폼, 레이블 등등의 컴포넌트들이 정의되어 있다. 컴포넌트는 팔레트나 오브젝트 인스펙터를 통해 직접 제어할 수 있지만 이외에도 VCL 내에는 프로그래밍의 대상이 되는 유용한 오브젝트들이 많이 있다. 대표적으로 8장에서 사용한 TBitmap 오브젝트를 들 수 있는데 컴포넌트는 아니므로 디자인중에는 사용할 수 없지만 코드를 통하여 사용한다. 이 외에도 여러분들이 자주 봐오던 오브젝트에는 TPen, TBrush, TCanvas, TStrings 등이 있다. 특히 이런 오브젝트들은 다른 컴포넌트의 속성으로 포함되어 있는 경우가 많다. 다음과 같은 코드를 보자.

```
MyForm.Canvas.Pen.Color:=clRed;
```

VCL 오브젝트도 컴포넌트와 마찬가지로 속성, 이벤트, 메소드 들을 가지고 있다. MyForm의 필드 중 Canvas라는 오브젝트가 포함되어 있고 Canvas는 Pen 오브젝트를 포함하며 Pen의 속성 중 하나인 Color를 clRed로 바꾸는 코드이다. 오브젝트는 변수와 동등한 자격을 가지며 오브젝트의 필드로 또 오브젝트가 사용될 수 있다는 것을 생각하면 전혀 이상한 코드가 아니다.

VCL 오브젝트에 관해서는 관련 부분에서 개별적으로 설명하되 자세한 사항

은 도움말이나 레퍼런스를 참고하기 바란다. VCL 오브젝트들의 계층 구조를 보고 싶으면 브라우저를 사용하면 된다.

# 차. 델파이의 코드 관리

새로운 프로젝트를 시작한 후 코드 에디터에서 소스를 살펴보면 TForm 클래스를 상속하여 TForm1이라는 클래스를 정의한 것을 볼 수 있다. 물론 이 클래스 이름은 폼의 Name 속성을 변경하면 따라서 변경된다. 그리고 TForm1의 실체를 선언하여 폼 오브젝트를 생성한다.

```
type
  TForm1 = class(TForm)    {폼 클래스 상속}
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;          {실체 선언}
```

원래의 TForm을 상속하여 새로운 클래스를 만드는 이유는 사용자가 만드는 폼에는 여러 가지 필드들이 덧붙여지기 때문이다. 예를 들어 하나의 버튼을 폼에 배치하면 이 버튼이 폼의 필드로서 클래스 선언에 포함되며 버튼의 OnClick 이벤트 핸들러를 작성하면 이 서브루틴도 폼의 메소드로 포함된다. 델파이가 직접 코드를 다음과 같이 변경할 것이다.

```
type
  TForm1 = class(TForm)
    Button1: TButton;                  {필드}
    procedure Button1Click(Sender: TObject); {메소드}
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

이렇게 하는 이유는 폼에 배치된 모든 컴포넌트의 오너(Owner)가 폼이 되며 폼이 전체를 관리(=대표)하도록 한다는 의미를 가지고 있다. 이벤트 핸들러가

폼의 메소드가 되어야 하는 이유는 상속의 번거로움을 회피하기 위해서이다. Button1의 OnClick 이벤트 핸들러를 Button1의 메소드로 정의하고자 한다면 TButton 클래스로부터 새로운 클래스를 상속받아야 함이 원칙이다. 그러나 컴포넌트의 이벤트는 메소드 포인터로 존재하며 이 포인터가 폼의 메소드를 가리키도록 변경해주면 상속을 하지 않아도 된다. 그래서 이벤트 핸들러는 폼의 메소드 형태로 만들어지며 Button1의 OnClick 이벤트는 이 번지값만을 가지도록 하여 상속을 하지 않고도 이벤트 핸들러를 정의한다. 이것을 OOP 용어로 대리(Delegation)라고 한다.

사용자가 서브루틴을 직접 정의하려면 가급적이면 TForm1 클래스 선언부의 private나 public 다음에 헤더를 적어 폼의 메소드 형태로 만드는 것이 더 좋다. 물론 헤더만 폼 클래스 선언에 포함시키고 코드는 implementation부에 작성해야 한다. 이렇게 하면 폼 내부의 컴포넌트를 좀 더 편리하게 액세스할 수 있을 뿐만 아니라 사용자가 만든 서브루틴도 정보 은폐를 할 수 있다.

델파이가 코드를 관리하는 이런 규칙은 어찌보면 내부적인 문제이므로 모르더라도 프로그램을 짜는 데는 지장이 없을 것이다. 그러나 알고 나면 더 안심하고 델파이의 서비스를 받을 수 있지 않을까 하는 생각이 든다.


① 오브젝트:변수(필드)와 서브루틴(메소드)을 묶어 하나의 독립된 단위를 이룬다.
② calss 키워드로 오브젝트를 선언한다.
③ 오브젝트는 캡슐화, 상속성, 정보 은폐 등의 특성을 가진다.
④ 오브젝트>컴포넌트>컨트롤

# 10-4 API 호출

윈도우즈가 제공하는 표준 함수들을 API(Application Programming Interface) 함수라고 한다. 도스가 도스에서 돌아가는 프로그램을 위해 도스 펑션콜을 제공하는 것과 마찬가지로 윈도우즈도 윈도우즈용 프로그램을 위해 표준적인 루틴들을 함수로 만들어 제공하며 이것을 윈도우즈 API 함수라 한다.

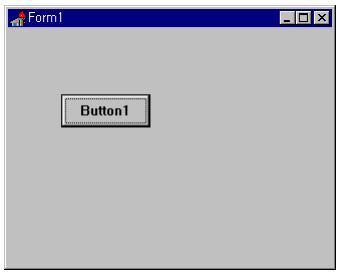
델파이도 윈도우즈용 프로그램이므로 API 함수를 사용하고 델파이로 만들어지는 프로그램도 물론 API 함수를 사용한다. 윈도우를 만들고, 윈도우를 이동시키고, 윈도우에 그림을 그리는 모든 동작이 API 함수로 이루어진다. 이런 API 함수 호출은 대개의 경우 델파이가 대신해 주므로 사용자들은 굳이 API 함수에 관해 몰라도 상관이 없을지도 모른다. 하지만 델파이가 제공하지 않는 기능을 사용해야 할 때, 저수준적인 처리가 필요하다거나 운영체제와 직접 정보를 교환해야 할 필요가 있다면 사용자가 직접 API 함수를 사용할 수 있도록 허락하고 있다. 운영체제가 제공하는 함수를 직접 사용하는 일이므로 다소 고급적인 기법에 사용되며 쉽지가 않다.

물론 API 함수는 안쓰면 그만이지만 꼭 필요한 경우가 있다. 예제를 통해 API 함수가 어떤 경우에 필요하며 어떻게 쓰는가를 알아보자.

## 가. 위치 기억

10jang
api1

일단 버튼 하나를 폼에 배치하고 속성이고 뭐고 하나도 건드리지 말고 버튼의 OnClick 이벤트 코드에 Close 명령만을 입력하고 이 프로그램을 Api1이라는 프로그램으로 저장해 둔다. 실행시키면 화면에 모습을 드러내고 버튼을 누르면 끝내는 기능뿐인 극단적으로 간단한 삼돌이같은 프로그램이다.



이 프로그램을 계속 실행시켜보면 계속 같은 위치에서 시작된다. 폼의 Position 속성이 poDesigned로 되어 있기 때문에 디자인시에 설정한 폼의 위치를 항상 고수한다. 만약 이 프로그램을 윈도우즈의 표준 프로그램인 시계처럼 끝낼 때 설정한 위치에서 다시 실행되도록 하고 싶다면 어떻게 해야 할까. 폼의 위치를 결정하는 Position 속성을 보면 디자인때 지정해 준대로, 항상 화면 중앙에서, 델파이 마음대로 등의 속성값은 있지만 "이전에 끝낸 위치에서"라는 속성값은 없다. 왜냐하면 끝낼 때 위치를 기억해 두는 기법은 속성만으로는 해결할 수 없기 때문이다.

프로그램을 끝낼 때 어딘가에 자신의 위치를 저장해 두어야 다시 시작할 때 저장해 둔 위치값을 읽어 그 위치에서 시작할 수 있다. 이때 좌표값의 저장위치는 현실적으로 디스크의 파일 또는 시스템 레지스트리가 있는데 레지스트리에 대해서는 다음에 알아보고 여기서는 디스크의 파일에 저장해 보도록 하자. 이런 정보 저장 목적으로 사용되는 파일이 INI 파일이다.

윈도우즈가 사용하는 WIN.INI, SYSTEM.INI와 제어판이 사용하는 CONTROL.INI, 그리고 시계가 사용하는 CLOCK.INI 등이 윈도우즈 디렉토리에 있을 것이다. CLOCK.INI 파일의 내용을 보면 과연 이 파일에 시계의 위치, 옵션 설정 상태, 크기, 폰트 등의 정보가 저장되어 있다.

```
[Clock]
Maximized=0
Options=0,0,0,0,0,0
Position=240,179,630,467
sFont=arial
Charset=0
```

CLOCK.EXE는 실행할 때마다 이 파일에서 정보를 읽어와서 위치와 옵션을 다시 설정하고 끝낼 때는 변경된 값을 이 파일에 저장해 두는 것이다. 그렇다면 CLOCK.EXE가 하는대로 우리가 만든 프로그램인 API1.EXE도 자신의 위치를 저장하도록 만들 수 있을 것이다. 그럼 어떻게 저장하는가? 직접 CALLAPI.INI 파일을 만들고 파일 입출력 함수를 사용해서 저장하는 것도 가능하지만 너무 복잡하다. 윈도우즈는 많은 프로그램이 INI 파일 제어를 필요로 함을 알고 이미 그런 API 함수를 제공한다. 다음 함수이다.

### 기본 형식

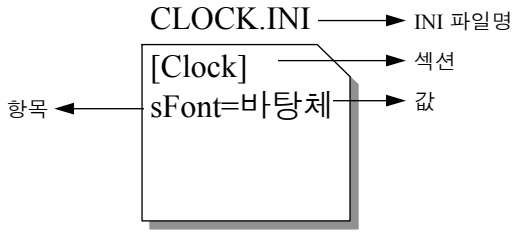
WritePrivateProfileString(섹션, 항목, 출력할 문자열,
INI 파일이름);

함수 이름이 무척이나 길다(표준 API 함수 중에 제일 길다). CLOCK.INI의 [Clock] 섹션의 sFont 항목에 '바탕체'를 쓰고자 한다면 다음과 같이 호출한다.

```
WritePrivateProfileString('clock','sFont','바탕체',
 'CLOCK.INI');
```

INI 파일의 구조만 이해하면 인수도 자연히 이해가 갈 것이다. 물론 순서가 좀 헷갈리겠지만 말이다.

그림
INI 파일의 구조




우리가 만드는 API1.EXE에서 저장할 정보는 폼의 위치(Form.Left, Form.Top)와 폼의 크기(Form.Height, Form.Width) 네 가지이다. 먼저 INI 파일부터 설계해 보자.

```
[Pos]
Left=100
Top=100

[Size]
Width=300
Height=300
```

이런식으로 만들고자 한다. INI 파일에 정보를 저장할 시점은 프로그램이 실행을 끝낼 때, 즉 폼이 닫혀질 때이므로 FormClose(또는 FormDestroy) 이벤트가 발생할 때이다. 이 이벤트에 다음 코드를 기입한다.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 WritePrivateProfileString('Pos','Left',
  PChar(IntToStr(Form1.Left)),'CALLAPI.INI');
 WritePrivateProfileString('Pos','Top',
  PChar(IntToStr(Form1.Top)),'CALLAPI.INI');
 WritePrivateProfileString('Size','Width',
  PChar(IntToStr(Form1.Width)),'CALLAPI.INI');
 WritePrivateProfileString('Size','Height',
  PChar(IntToStr(Form1.Height)),'CALLAPI.INI');
end;
```

Form1.Left값을 출력하는 예를 보면 먼저 Form1.Left가 정수값이므로 문자열로 바꾸기 위해 IntToStr을 호출하고 이 문자열을 PChar로 타입 캐스팅하여 함수의 세 번째 인수로 전달한다. 델파이의 문자열을 API 함수의 인수로 사용할 때는 이처럼 PChar로 타입 캐스팅을 해 주어야 한다. 출력할 값을 문자열로 만든 후 WritePrivateProfileString 함수를 적당한 인수와 함께 호출하여 정보를 기입한다. 나머지 세 개의 정보를 기입하는 과정도 동일하다.

다음 코드는 저장한 정보를 프로그램 실행시에 읽어오는 코드를 작성한것이다. 정보를 읽는 시점은 폼이 만들어질 때이므로 FormCreate 이벤트(또는 FormShow)를 사용한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Form1.Left:=GetPrivateProfileInt('Pos','Left',
  100,'CALLAPI.INI');
 Form1.Top:=GetPrivateProfileInt('Pos','Top',
  100,'CALLAPI.INI');
 Form1.Width:=GetPrivateProfileInt('Size','Width',
  300,'CALLAPI.INI');
 Form1.Height:=GetPrivateProfileInt('Size','Height',
  250,'CALLAPI.INI');
end;
```

WritePrivateProfileString 함수의 반대되는 함수는 두 가지가 있다.

| | |
|---|---|
| GetPrivateProfileInt | 해당 정보를 정수로 읽어들인다. |
| GetPrivateProfileString | 해당 정보를 문자열로 읽어들인다. |

우리가 읽어올 값인 Form.Left와 그 일당들은 모두 정수형이므로 GetPrivateProfileInt 함수를 사용함이 지당하다.

### 기본 형식

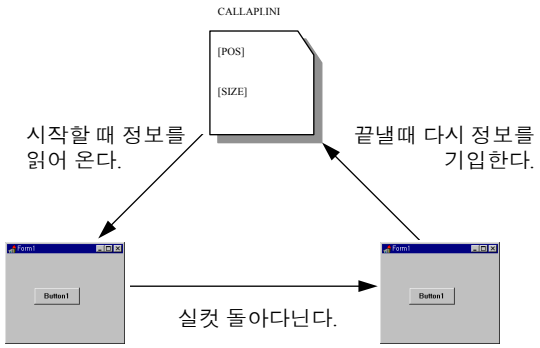
GetPrivateProfileInt(섹션, 항목, 디폴트값, INI 파일);

이런 형식을 가지며 읽은 정수값은 리턴값으로 돌려진다. 디폴트값이란 INI 파일이 없거나 항목이 정의되지 않은 경우에 리턴되는 값이다. 최초 INI파일이 만들어지지 않은 경우를 고려하여 가장 무난한 값을 주는 것이 좋다.

이제 직접 프로그램을 실행해 보자. 프로그램이 시작할 때와 끝날 때 하드 디스크가 돌아가는 소리가 짧게 나며 항상 끝낸 위치에서 다시 시작한다. 윈도우즈 디렉토리를 살펴보면 과연 CALLAPI.INI라는 파일이 예쁘게 만들어져 있을 것이다.

**그림**

항상 같은 위치에서 실행될 수 있는 원리




여기서는 폼의 위치와 크기를 저장하는 목적으로 INI 파일을 사용했지만 응용하기에 따라서는 어떤 형태의 정보도 저장할 수 있다. 게임 프로그램에서 누가 누가 잘했나 점수표, 각종 옵션 설정 상태, 작업 진행 상태 등의 정보를 모두 저장할 수 있다.

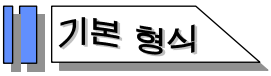
# 나. 해상도 조사

CD

10jang
api2

델파이로 만든 프로그램의 첫 위치는 폼의 Position 속성으로 지정하며 이 속

성값에 따라 디자인시에 설정한 위치에 나타나거나 무조건 화면의 중앙에 나타난다. 또는 앞에서 실습한대로 이전 실행을 끝낸 위치에서 시작하도록 끝낼 때의 좌표를 INI 파일에 저장해 두고 그 위치에 나타나도록 할 수 있다. 그렇다면 무조건 화면의 우하단에 나타나도록 하려면 어떻게 해야 할까? 아주 쉬운 문제인 것 같지만 화면 우하단 좌표는 정해지지 않았기 때문에 쉽게 해결할 수 없다. 화면의 해상도에 따라 우하단의 좌표가 달라지기 때문이다. 그래서 이 문제는 화면 해상도를 알아내면 해결할 수 있다. 이 때 사용하는 함수가 GetSystemMetrics이며 이 함수는 화면의 해상도뿐만 아니라 윈도우즈의 여러 가지 설정 상태를 알려준다.




```
function GetSystemMetrics(Index:Integer):Integer;
```

알고자 하는 정보의 인덱스를 넘겨주면 정보를 조사하여 결과를 리턴해 준다. 인덱스는 다음과 같다. 일부만 보이므로 나머지는 도움말을 참조하기 바란다.

| 인덱스 | 의미 |
|---|---|
| SM_CXSCREEN | 화면의 가로 크기 |
| SM_CYSCREEN | 화면의 세로 크기 |
| SM_MOUSEPRESENT | 마우스의 존재 여부 |
| SM_SWAPBUTTON | 마우스 버튼 교체 여부 |
| SM_CXICON | 아이콘의 가로 크기 |
| SM_CXICONSPACING | 아이콘의 가로 간격 |
| SM_CXFRAME | 윈도우 경계선의 굵기 |

우리가 알고자 하는 정보는 화면의 크기이므로 SM_CXSCREEN과 SM_CYSCREEN 인덱스로 정보를 조사하면 된다. 폼이 처음 생성될 때 조사된 정보를 사용하여 폼의 위치를 재조정하도록 코드를 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Width:=210;
  Height:=100;
```

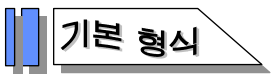
```
  Left:=GetSystemMetrics(SM_CXSCREEN)-Width;
  Top:=GetSystemMetrics(SM_CYSCREEN)-Height;
end;
```

실행시에 해상도를 직접 조사하여 폼의 좌표를 결정하므로 현재 설정된 해상도에 상관없이 폼은 항상 화면의 우하단에 출력된다.

# 다. 비디오 색상 조사

10jang
devcap

그래픽 프로그램의 경우, 특정한 색상 모드에서만 돌아가는 프로그램이 있을 수 있다. 게임의 경우 256색상 모드가 가장 적절하며 그래픽 편집 프로그램은 트루 컬러(True Color) 모드에서 동작되는 것이 좋다. 만약 프로그램이 적절하게 실행될 수 없는 그래픽 모드라면 사용자에게 알려 주는 것이 좋으며 그러기 위해서는 현재의 비디오 모드를 알아야 한다. 현재 프로그램이 실행되는 비디오 모드에서 사용가능한 색상 수나 기타 여러 가지 정보를 알고 싶으면 GetDeviceCaps라는 API 함수를 사용한다.



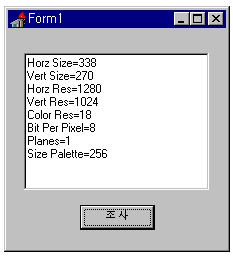

```
function GetDeviceCaps(hdc:HDC;
    Index:Integer):Integer;
```

GetSystemMetrics 함수와 마찬가지로 알고자 하는 정보의 인덱스를 인수로 넘겨주면 정보를 조사해 준다. 첫 번째 인수인 hdc는 조사하고자 하는 장치의 장치 핸들(DC)이며 캔버스 오브젝트의 HANDLE을 넘겨주면 된다. 인덱스는 다음과 같다. 일부만 보이므로 상세한 인덱스 목록은 도움말을 참조하기 바란다.

| 인덱스 | 의미 |
|---|---|
| HORZSIZE | 물리적인 화면의 수평 크기를 밀리미터 단위로 조사한다. |
| VERTSIZE | 물리적인 화면의 수직 크기를 밀리미터 단위로 조사한다. |
| HORZRES | 화면의 수평 해상도를 픽셀 단위로 조사한다. |
| VERTRES | 화면의 수직 해상도를 픽셀 단위로 조사한다. |

| | |
|---|---|
| BITSPIXEL | 각 점의 색상을 기억하는 비트 수이다. |
| PLANES | 컬러면의 수이다. |
| COLORRES | 색상의 해상도를 나타내며 이 값은 표현 가능한 색상 수가 된다. |
| SIZEPALETTE | 시스템 팔레트의 개수이다. |

인덱스가 의미하는 바를 정확하게 이해하려면 아무래도 그래픽 카드와 비디오 메모리에 관한 이해가 선행되어야 할 것 같다. 예제의 실행 화면은 다음과 같다.




버튼을 누르면 단순히 비디오에 관한 여러 가지 정보를 조사해서 메모 컴포넌트로 출력해 준다. 아주 단순한 함수 호출의 나열이다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  t:integer;
begin
  t:=GetDeviceCaps(Canvas.Handle,HORZSIZE);
  Memo1.Lines[0]:='Horz Size='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,VERTSIZE);
  Memo1.Lines[1]:='Vert Size='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,HORZRES);
  Memo1.Lines[2]:='Horz Res='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,VERTRES);
  Memo1.Lines[3]:='Vert Res='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,COLORRES);
  Memo1.Lines[4]:='Color Res='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,BITSPIXEL);
  Memo1.Lines[5]:='Bit Per Pixel='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,PLANES);
  Memo1.Lines[6]:='Planes='+IntToStr(t);
  t:=GetDeviceCaps(Canvas.Handle,NUMCOLORS);
  Memo1.Lines[7]:='Num Colors='+IntToStr(t);
```

```
  t:=GetDeviceCaps(Canvas.Handle,SIZEPALETTE);
  Memo1.Lines[8]:='Size Palette='+IntToStr(t);
end;
```

ColorRes가 18이라는 말은 최대 표현 가능한 색상의 수가 $2^{18}$이라는 뜻이며 BitsPixel이 8이라는 말은 비디오 메모리가 한 점의 색상을 기억할 때 8비트를 사용한다는 뜻이다. Size Palette가 256색상이라는 말은 동시에 출력가능한 색상의 수가 256색상이라는 뜻이다.

# 라. 외부 프로그램 실행

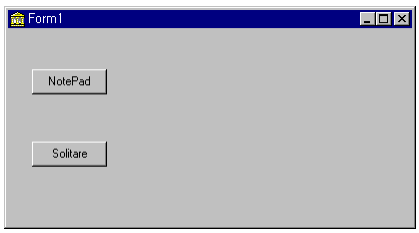
10jang
WinExec

델파이 프로그램에서 외부 프로그램을 실행시키려면 다음 API 함수를 사용한다.

```
UINT WinExec(LPCSTR lpCmdLine,  UINT uCmdShow);
```

첫 번째 인수로 실행시키고자 하는 프로그램 이름을 주되 드라이브명이나 디렉토리명이 포함될 수 있으며 명령행 인수도 전달해 줄 수 있다. 두 번째 인수는 실행 직후에 프로그램이 어떻게 보일 것인가를 지정하는데 통상 이 값은 Sw_ShowNormal 이며 다른 값으로 변경하면 최소화한 상태로 실행시킬 수도 있다.

사용 방법이 쉬우므로 간단한 예제만 하나 만들어 보자. 폼에 버튼 두 개를 배치해 두고 각 버튼을 누르면 메모장과 솔리테어 게임을 실행시켜 볼 것이다.




두 버튼의 이벤트 핸들러를 각각 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  WinExec('Notepad.exe', Sw_ShowNormal);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  WinExec('Sol.exe', Sw_ShowNormal);
end;
```

이제 컴파일한 후 버튼을 눌러보면 메모장이나 솔리테어가 실행될 것이다. 이 함수는 바로 다음 장에서 쉘 프로그램을 만들면서 사용하게 될 것이다. 이 함수 외에 Win32 API 에는 CreateProcess 라는 좀 더 진보된 함수를 제공하며 가급적 이 함수를 사용할 것을 권하고 있다.

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
   );
```

보다시피 인수의 수가 많아 이것 저것 세세하게 많은 것을 지정할 수 있으나 너무 불필요한 것까지 있어 초보자가 사용할만한 함수는 아니다. 비록 WinExec 함수가 16 비트 윈도우즈의 구식 함수지만 이 함수는 호환성을 위해 여전히 제공되고 있으며 CreateProcess 보다 오히려 더 쓰기 쉽다.

API 함수를 쓰는 것은 함수에 대한 개념을 이해하고 함수를 불러서 쓸 줄만 알면 쉽게 사용할 수 있다. 더구나 델파이는 전체 API 함수에 대한 상세한 도움말을 제공하여 철저한 지원태세까지 갖추고 있다. 단 API 함수 중에는 앞에서 본 것처럼 간단한 것도 있지만 CreateWindow와 같은 골때리게 복잡한 함수도 있고 GetDC나 CreateThread와 같이 윈도우즈 시스템에 대한 전반적인 이해가 필요한 것도 있다. 델파이가 직접 하지 못하는 일은 API 함수를 사용해야 하므로 별도의 API 레퍼런스를 항상 참고하기 바란다.

참고하세요

윈도우즈 API에 관심이 있는 사람은 C를 공부해 보기 바란다. 델파이로 API 함수를 공부하는 것도 불가능하지는 않지만 API에 관한 많은 자료와 예제들이 거의 대부분 C로 짜여져 있기 때문에 API를 알고 싶으면 C를 통하는 것이 가장 확실한 방법이다. C로 API 함수를 익히는 데는 많은 시간과 노력이 드는 것이 사실이지만 윈도우즈 시스템의 내부 구조까지 엿볼 수 있어 투자한만큼의 가치가 충분히 있을 것이다.

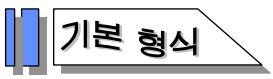
# 10-5 포인터

## 가. 포인터형

포인터란 아주 특수한 형태의 변수이다. 정수형이나 실수형, 문자형 변수는 값을 기억하는데 비해 포인터는 번지를 기억한다. 그래서 포인터는 다른 변수와는 다른 특징들을 가지고 있으며 이런 특수성으로 인해 포인터를 자연스럽게 사용하기란 무척 어렵다. 하지만 델파이에서 포인터를 사용할 일이란 극히 드문 일이므로 크게 걱정할 필요는 없다.

### ■ 타입형 포인터

포인터 변수를 선언할 때는 ^기호를 사용하며 ^기호 다음에 포인터 변수가 가리키고자 하는 데이터의 타입을 적어준다.




```
var
  변수:^데이터형;
```

예를 들어 PINT:^integer; 라고 선언하면 PINT는 정수형 변수의 번지를 기억하는 포인터 변수가 된다. 포인터형의 데이터형이란 포인터 변수가 가리키는 번지의 변수가 가지는 데이터형을 말한다. 말이 조금 꼬이는 듯한 느낌이 들지만 쉽게 예를 들자면 정수형 포인터 변수 PINT는 정수형 변수의 번지를 기억할 수 있다는 얘기다. 대상체의 데이터형을 가지므로 타입형 포인터(typed Pointer)라고 한다. 포인터 변수가 데이터형을 가져야 하는 이유는 다음 두 가지이다.

첫째, 포인터가 가리키는 대상체를 읽어낼 경우 읽어야 할 바이트 수와 읽어낸 값의 비트 해석 방법을 명시화하기 위해서이다. 정수형 포인터는 4바이트를 읽어 정수를 읽어내고 실수형 변수(double)는 8바이트를 읽어 실수를 읽어낸다.

둘째, 포인터 연산에 의해 다른 대상체의 번지로 이동할 때의 이동 거리를 명

시하기 위해서이다. 포인터는 번지를 가리키며 다른 번지값을 가지도록 변경될 수 있다. 특히 배열을 포인트할 때 배열의 다음 요소와의 거리를 계산하기 위해서 포인터의 데이터형이 반드시 필요하다.

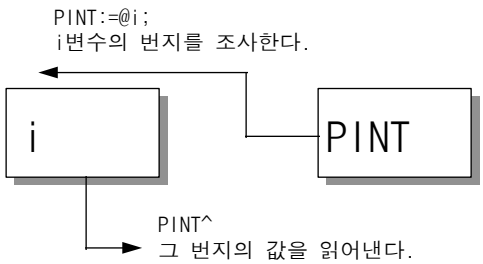
포인터 변수에 값을 대입할 때는 @연산자를 사용한다. @연산자는 피연산자의 번지를 계산해 주며 이 연산자가 계산한 값은 포인터 변수로 대입가능하다. 포인터 변수가 가리키는 값을 읽어낼 때는 ^연산자를 사용하며 포인터 변수 뒤에 기입해 준다. 다음 예제는 포인터 변수를 사용하여 정수형 변수의 값을 간접적으로 읽어낸다.

```
var
  PINT:^integer;
  i:integer;
begin
  i:=5;
  PINT:=@i;
  label1.caption:=IntToStr(PINT^);
end;
```

i는 정수형 변수로 선언되었고 PINT는 정수형 변수의 번지를 가지는 포인터로 선언되었다. i가 5라는 값을 대입받고 PINT는 @i, 즉 i 변수가 기억되어 있는 메모리의 번지값을 대입받았으며 PINT^ 표현에 의해 그 번지에 있는 값을 읽어낸다.

그림

포인터를 사용하여 변수의 값을 간접적으로 읽어낸다.

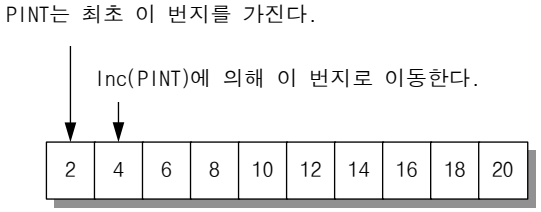



물론 이 경우는 PINT^ 대신에 i 변수값을 곧바로 읽어도 결과는 같다. 차이점이라면 직접 변수를 읽는다는 것과 간접적으로 변수를 읽는다는 점이다. 간접적으로 한 단계를 더 거치면 중간에서 여러 가지 조작을 할 수 있다. 그 예로써 배열값을 읽는 예를 들 수 있다.

```
var
  PINT:^integer;
  i:array [1..10] of integer;
  j:integer;
begin
  for j:=1 to 10 do
    i[j]:=j*2;
  PINT:=@i;
  Label1.Caption:=IntToStr(PINT^);
  Inc(PINT);
  Label1.Caption:=IntToStr(PINT^);
end;
```

크기 10의 정수형 배열 i에 2,4,6,8....의 값들을 기억시켜 두고 이 값들을 순서대로 읽는 예제이다. PINT는 최초 @i, 즉 배열의 시작 번지값(i[1]의 번지)을 가지지만 PINT를 증가시킴으로써 배열의 다음 요소로 정확하게 이동하여 i[2]의 값을 읽어낼 수 있다. PINT가 정수형 포인터 변수이므로 Inc(PINT)에 의해 PINT는 2바이트 증가하게 된다.

**그림**
포인터 연산




이 경우도 물론 i의 첨자로 사용할 정수형 변수를 별도로 하나 더 선언하여 사용하면 똑같은 동작을 하도록 만들 수 있다.

## ■ 타입 없는 포인터

타입 없는 포인터(untyped pointer)란 포인터가 가리키는 대상체가 어떤 데이터형을 가지는지가 정해지지 않은 포인터를 말한다. 메모리 상의 위치값인 번지를 기억하는 기능만 가지고 있다. 그래서 타입 없는 포인터는 어떤 대상체의 번지든지 자유롭게 대입받을 수는 있지만 포인터만으로 메모리의 값을 읽어낼 수는 없다. 타입 없는 포인터를 선언할 때는 Pointer라고 데이터형을 밝혀준다. 다음 예를 보자.

```
var
```

```
  i:Integer;
  P:Pointer;
begin
  i:=5;
  P:=@i;
  Label1.Caption:=IntToStr(P^);
end;
```

P가 타입 없는 포인터(Pointer)형으로 선언되었으며 대상체의 데이터형이 정해져있지 않으므로 어떤 대상체든, 즉 정수형이든 실수형이든 또는 문자열형이든 그 번지를 대입받을 수 있다. 그러나 P^식으로 그 번지에 있는 데이터를 곧바로 읽어낼 수는 없다. 왜냐하면 대상체의 데이터형을 모르므로 몇 바이트를 읽어낼지, 읽어낸 데이터를 어떻게 해석해야 할지를 알 수가 없기 때문이다. 그래서 Pointer형의 포인터가 가리키는 번지의 데이터를 읽으려면 꼭 타입 캐스팅을 해 주어야 한다.

```
var
  i:Integer;
  P:Pointer;
begin
  i:=5;
  P:=@i;
  Label1.Caption:=IntToStr(Integer(P^));
end;
```

이렇게 해야만 P가 가리키는 번지의 데이터를 제대로 읽을 수 있다. Pointer형은 대상체의 데이터형이 미리 정해져 있지 않다뿐이지 말을 바꾸면 캐스팅에 의해 어떤 대상체든지 포인트할 수 있다는 뜻도 된다.

# 나. 동적 메모리 할당

일반적으로 변수는 정적으로 메모리를 할당받는다. 정적으로 메모리를 할당한다는 말은 컴파일시에 변수를 위해 메모리를 확보한다는 얘기이며 정적으로 할당된 메모리는 실행 파일의 크기에 포함된다. var i:integer; 선언에 의해 컴파일러는 i 변수를 위해 4바이트를 할당해 주며 이후부터 i 변수는 4바이트의 메모리를 배타적으로 사용하며 정수값 하나를 기억할 수 있다.

동적으로 메모리를 할당한다는 말은 실행중에 변수를 위해 메모리를 확보하여 이 메모리에 변수를 보관하며 사용이 끝난 후에 메모리를 반납한다는 말이

다. 잠시 임시적으로 사용될 메모리라면 정적으로 할당하는 것보다는 동적으로 할당하여 사용하고 버리는 것이 좋다. 다음은 동적으로 정수형 변수를 위한 메모리를 할당한 후 사용하고 버리는 예이다.

```
var
  PINT:^integer;
begin
  New(PINT);
  PINT^:=5;
  Label1.Caption:=IntToStr(PINT^);
  Dispose(PINT);
end;
```

보다시피 정수형 변수는 선언하지 않고 정수형 포인터 변수 PINT만을 선언하였다. New 프로시저는 동적으로 메모리를 할당하여 인수로 전달된 포인터 변수에 그 번지를 대입해 준다. 할당된 메모리에 값을 읽거나 대입할 때는 PINT^라는 표현을 사용하며 이 경우 PINT^는 정수형 변수와 동일한 문법적 자격을 갖는다. 즉 정수가 사용될 수 있는 위치라면 PINT^도 언제나 사용가능하다는 뜻이다. 위 코드를 보면 PINT^에 값을 대입하기도 하고 PINT^의 값을 읽기도 한다.

동적으로 할당한 변수를 사용하고 난 후에는 반드시 Dispose 프로시저를 사용하여 할당을 해제해 주어야 한다. 그렇지 않으면 그 메모리는 언제까지고 할당된 채로 남아있게 되므로 다시는 다른 목적으로 사용할 수 없게 된다. 이 예제는 포인터를 사용하여 동적으로 메모리를 할당하는 예를 보여줄 뿐이며 실질적으로 꼭 동적으로 메모리를 할당해야 할 필요성은 없다. 하지만 다음과 같은 경우는 포인터가 아니면 문제를 해결할 수 없다.

단 이 예는 어디까지 인위적으로 극단화시킨 예이며 32비트 델파이에서는 약간 부적절한 예일 수도 있다. 이론 설명을 위해 작성되었으므로 여러분들도 그렇게 이해하기 바란다. 크기 100의 문자열 배열이 필요하여 전역 변수로 info라는 문자열 배열 변수를 선언하였다.

```
var
  Form1: TForm1;
  info:array [0..100] of string;
```

각 문자열의 크기는 256이므로 이 배열의 전체 크기는 25000바이트가 넘는 대단히 큰 크기를 가진다. 이런 변수를 가지는 프로그램을 작성하면 당장은 문제가 없지만 프로그램을 확장하다 보면 Too many variables 에러가 발생하게 된다. 왜냐하면 전역 변수는 DS에 보관되며 총 64K를 넘을 수 없기 때문이다. 만약 array[0..300] of string 따위의 배열 변수가 필요할 경우 이런 변수는 아예

선언조차도 할 수 없다. 이 때는 정적으로 변수를 할당해서는 안되며 동적으로 메모리를 할당받아 사용해야 한다. 다음 예를 보자.

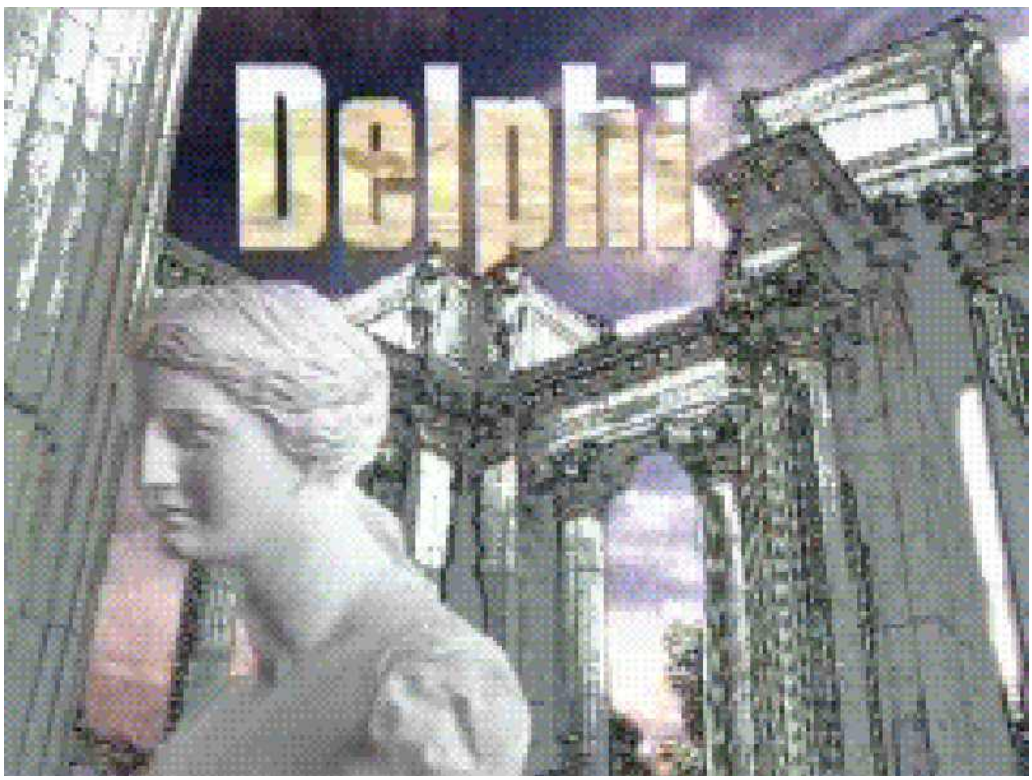
```
type
  infotype=array [0..100] of string;
var
  Form1: TForm1;
  info:^infotype;
implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  new(info);
  info^[1]:='kim sang hyung';
  Label1.Caption:=info^[1];
  dispose(info);
end;
```

info는 infotype, 즉 크기 100의 문자열 배열을 가리키는 포인터 변수로 선언되었으며 FormCreate 이벤트 핸들러에서 New 프로시저에 의해 동적으로 메모리를 할당받았다. 이 배열을 사용할 때는 info^[첨자] 형식으로 사용해야 하며 프로그램이 실행중인 동안은 계속 값을 보관할 수 있다. 메모리를 다 사용하고 난 후에는 반드시 dispose로 메모리 할당을 해제해 주어야 한다. 위의 예제는 FormCreate 이벤트에서 할당과 해제를 모두 하고 있지만 dispose는 FormDestroy로 이동하는 것이 적절하다.


① 포인터:번지를 기억하는 특수한 데이터형
② 포인터 형을 선언할 때는 ^기호를 사용한다.
③ @연산자:피연산자의 번지값을 계산한다.
④ New:실행중에 메모리를 할당한다.

# 제 11 장

# 파일 관리



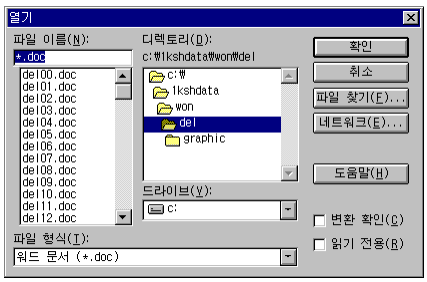
# 11-1 미니쉘

## 가. 파일 관련 컴포넌트

프로그램이 만들어낸 결과를 저장하기 위해서는 하드 디스크의 파일을 이용하는 방법 외에는 아직까지도 별 뾰족한 방법이 없다. 메모리는 아무리 빠르고 정확해도 전기가 없다면 그야말로 아무 힘도 쓰지 못한다. 프로그램이 만들어낸 정보는 파일로 보관되는데, 어찌보면 프로그램의 모든 결과가 파일로 남게 된다. 워드 프로세서가 만들어 내는 문서 파일, 데이터 베이스 프로그램이 만들어 내는 DB 파일, 그래픽 프로그램이 만들어 내는 그림 파일, 그외 사운드 파일, 동영상 파일 등등 대부분의 프로그램이 파일을 사용한다. 물론 게임이나 쉘, 시계 등과 같이 파일과 전혀 무관한 프로그램도 간혹 있기는 하다.

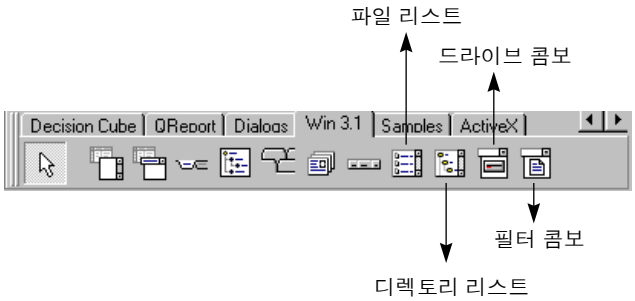
파일을 제어한다는 말은 파일로부터 데이터를 읽고 쓰는 것을 말하며 그전에 작업 대상이 될 파일을 사용자로부터 입력받아야 한다. 도스에서는 사용자가 del myfile.txt 등과 같이 파일 이름을 직접 입력해 주지만 윈도우즈에서는 파일 이름을 입력받을 때도 대화상자를 사용한다. 다음은 윈도우즈 3.1용 워드 6.0의 파일 열기 대화상자이다.

**그림**
파일 열기 대화상자




파일 목록과 디렉토리 목록 등으로 원하는 위치와 파일을 선택하도록 되어 있다. 델파이에서 이런 대화상자를 구성하는 컴포넌트에는 모두 4가지가 있으며 모두 Win 3.1 팔레트에 있다.

그림
파일과 관련된 컴포넌트들




지금은 파일 열기 공통 대화상자 덕분에 이 컴포넌트들을 파일 선택에 사용하는 일이 거의 없어 구닥다리 취급을 받고 있으며 델파이도 예전에는 이 컴포넌트들을 System 페이지에 두었었는데 지금은 Win 3.1 페이지로 이동시켰다. 하지만 이 컴포넌트들은 여전히 아주 쓸만한 컴포넌트들이다.

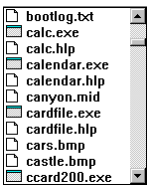
## ■ 파일 리스트 박스

현재 디렉토리의 파일 목록을 나타낸다. 어떤 디렉토리에 있는 파일의 목록을 나타낼 것인가는 Directory 속성으로 지정한다. 이 속성은 실행시에만 사용할 수 있으므로 디자인시에는 디렉토리 목록을 변경할 수 없다. 직접 이 속성에 디렉토리 이름을 대입해 주는 경우는 드물며 주로 디렉토리 리스트 박스에서 간접적으로 대입받는다. 파일 리스트 박스에서 선택된 파일 이름은 FileName 속성에 나타나므로 실행중에 이 속성을 읽어 파일 이름을 입력 받는다.

Directory, FileName 이 두 가지 속성은 실행중에 파일 리스트 박스를 제어하는 중요한 속성이며 나머지 속성은 파일 리스트 박스의 모양을 치장할 때 주로 사용된다.

### ShowGlyphs

파일 이름 옆에 아이콘이 나타나도록 한다. 실행 파일과 데이터 파일의 아이콘을 다르게 보여 줌으로써 프로그램 파일을 좀 더 쉽게 찾을 수 있다.



### ✍ Mask

목록에  어떤 종류의 파일을 나타낼 것인가를 지정한다. *.exe나 *.bmp 등과 같이 와일드 카드식으로 표현한다. 디폴트는 *.*로 설정되어 있으며 모든 파일의 목록을 보여준다.

### ✍ FileType

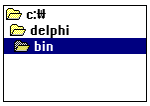
목록에 표시할 파일의 속성을 설정하며 파일 속성에 대응되는 세부 속성을 가지고 있다. 디폴트는 ftNormal만 True로 설정되어 있어 보통 파일만 목록에 나타난다. 숨은 파일이나 시스템 파일까지 목록에 나타내고자 한다면 해당 속성을 True로 바꾸어 주도록 한다.

### ✍ MultiSelect

여러 개의 파일을 선택할 수 있도록 허락한다. 디폴트는 False이며 한번에 하나씩만 선택할 수 있다.

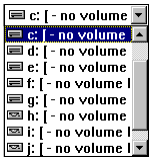
## ■ 디렉토리 리스트 박스

현재 드라이브의 디렉토리 목록을 나타낸다. 단순히 디렉토리를 순서대로 나열하여 보여주기만 하는 것이 아니라 디렉토리 구조를 인식하여 계층적으로 디렉토리 구조를 보여준다. 어떤 드라이브의 목록을 보여줄 것인가는 Drive 속성으로 설정하며 실행중에 선택된 디렉토리명은 Directory 속성에 나타난다.




## ■ 드라이브 콤보 박스

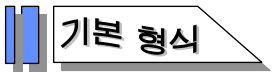
시스템에 장착되어 있는 모든 드라이브의 목록을 보여주며 그 중 하나를 선택하도록 한다. 드라이브 이름과 볼륨 이름을 보여줌은 물론 드라이브명 옆에 어떤 종류의 드라이브인가를 나타내는 조그만 아이콘까지 보여준다. 실행중에 사용자에 의해 선택된 드라이브는 Drive 속성으로 읽는다.



## ■ 필터 콤보 박스

필터란 파일 리스트 박스에 보여줄 파일의 유형을 말하며 보통 *.doc, *.txt 등과 같은 와일드 카드로 표현된다. 필터 콤보 박스는 이런 필터의 목록을 가진다. 보통 여러 개의 필터를 제공하며 그 중 하나를 선택하도록 한다. 대표적으로 워드 패드만 해도 *.WRI, *.DOC, *.TXT, *.* 네 개의 필터를 제공한다.

Filter 속성에 필터들을 등록시키며 사용자에 의해 선택된 필터는 Mask 속성에 나타난다. 필터 콤보 박스는 보통 파일 리스트 박스와 함께 사용되며 필터 콤보에서 선택된 Mask를 파일 리스트 박스의 Mask 속성에 대입해 준다. 필터는 일정한 형식에 따라 정의되며 기본 형식은 다음과 같다.




설명|필터

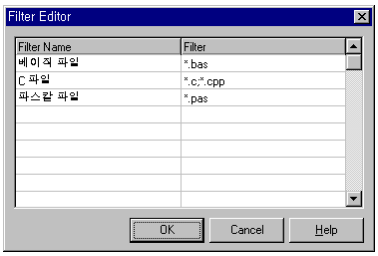
설명은 어떤 종류의 파일인가를 알려주는 문자열이며 필터는 *.exe와 같은 와일드 카드식으로 표현되며 설명과 필터 사이에 파이프 문자(|)가 삽입된다. '워드 파일|*.doc','엑셀 파일|*.xls' 등이 하나의 필터가 된다. 여러 개의 필터를 계속 파이프로 연결하여 작성할 수 있으며 필터 사이사이에 파이프 문자로 구분해 준다. '워드 파일|*.doc|엑셀 파일|*.xls|그림 파일|*.gif' 는 세 개의 필터를 정의한 예이다.

하나의 필터에 두 개 이상의 와일드 카드식을 쓰고 싶으면 '실행 파일|*.exe;*.com;*.bat' 등과 같이 세미콜론으로 구분하여 와일드 카드식을 나열한다. 다음은 필터를 작성한 예와 실행중에 필터 콤보 박스를 보인 것이다.

```
베이직 파일|*.BAS|C 파일|*.C;*.CPP|파스칼 파일|*.PAS
프로젝트 파일|*.DPR|폼 파일|*.DFM|유닛 파일|*.PAS
```


Filter 속성을 이렇게 직접 정의하는 방법 외에도 대화상자를 사용하는 좀 더 편리한 방법이 있다. 오브젝트 인스펙터에서 Filter 속성을 더블클릭하면 필터를 입력할 수 있는 대화상자를 보여준다.

그림
필터 입력 대화상자




필터 이름과 필터가 좌우측에 나누어져 있어서 파이프 문자를 쓰지 않아도 된다는 것 뿐 직접 입력하는 것과 큰 차이는 없다.

# 나. 컴포넌트 간의 연결

파일 선택에 사용되는 컴포넌트들은 대개의 경우 단독으로 사용되지 않으며 두 개 이상이 짝을 지어 사용되는 경우가 많고 그것도 네 개가 한꺼번에 사용되는 경우가 대부분이다. 그 이유는 서로 연관된 정보를 표현하며 실행중에도 상호 영향을 미치기 때문이다. 예를 들어 디렉토리 리스트 박스에서 디렉토리를 변경하면 파일 리스트 박스의 목록이 바뀌어야 하며 드라이브 콤보 박스에서 드라이브를 변경하면 디렉토리, 파일 리스트 박스가 모두 바뀌어야 한다.

컴포넌트끼리 이런 상호 작용을 하기 위해서는 컴포넌트를 연결해 주어야 하며 여기에는 두 가지 방법이 사용된다.

## ■ OnChange 이벤트 사용

한쪽에 변화가 생겼을 때 영향을 받는 다른 쪽의 속성을 직접 변경해 준다. OnChange 이벤트에 영향받는 쪽의 속성을 변경하는 코드를 다음과 같이 작성한다.

```
procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryListBox1.Drive := DriveComboBox1.Drive;
end;
```

드라이브 콤보 박스에서 어떤 변화가 있었을 때, 즉 드라이브가 변경되었을 때 디렉토리 리스트 박스의 Drive 속성을 변경해 주는 코드이다. 마찬가지로 디렉토리가 변경되면 파일 리스트 박스의 Directory 속성을 변경해 준다.

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

또한 필터 콤보 박스에서 어떤 필터를 선택하는가에 따라 파일 리스트 박스의 목록에 출력되는 파일 목록이 달라진다.

```
procedure TForm1.FilterComboBox1Change(Sender: TObject);
begin
FileListBox1.Mask := FilterComboBox1.Mask;
end;
```

OnChange 이벤트를 사용하는 이런 방법은 누구나 생각할 수 있는 고전적인 방법이며 이해하기도 쉬울 것이다.

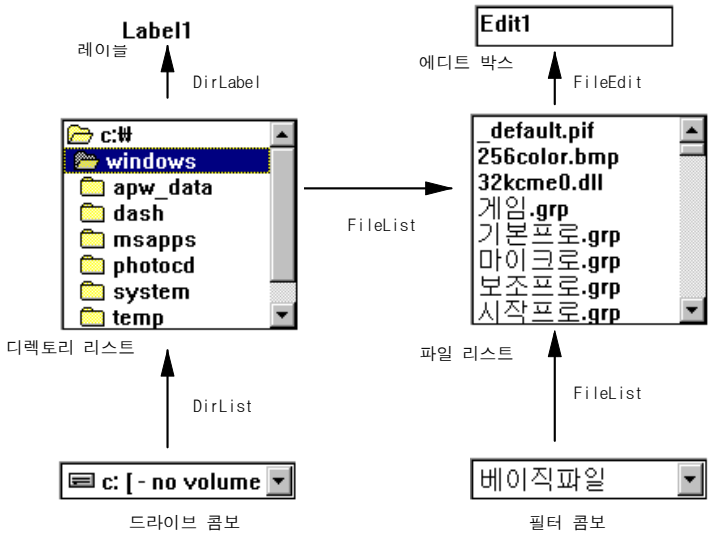
## ■ 속성으로 서로 연결하는 방법

OnChange 이벤트를 사용하는 방법은 너무 뻔하고 당연한 방법이며 이 방법을 사용하면 물론 아무런 문제없이 컴포넌트끼리 연결할 수 있다. 그러나 이보다 좀 더 쉬운 방법으로 코드를 사용하지 않고 컴포넌트의 속성을 사용하여 연결하는 방법이 있다.

드라이브 콤보 박스의 DirList 속성은 이 콤보 박스에 변화가 있을 경우 영향을 받을 디렉토리 리스트 박스를 지정한다. 즉 이 속성에 디렉토리 리스트 박스를 지정해 주기만 하면 드라이브가 변경될 때 디렉토리 리스트 박스의 Drive 속성을 자동으로 바꾸어 준다. 마찬가지로 디렉토리 리스트 박스의 FileList 속성은 디렉토리 변경시 영향을 받을 파일 리스트 박스를 지정하며 필터 콤보 박스의 FileList 속성도 필터 변경시 영향을 받을 파일 리스트 박스를 지정한다.

이 외에도 디렉토리 리스트 박스에서 선택된 디렉토리 이름을 레이블에 출력하고자 할 때 사용하는 DirLabel 속성과 파일 리스트 박스에서 선택한 파일을 에디트 박스에 출력하는 FileEdit 속성이 있다. 속성에 의한 컴포넌트 간의 상호 연결관계는 다음과 같다.

그림
속성에 의한 컴포넌트의 연결



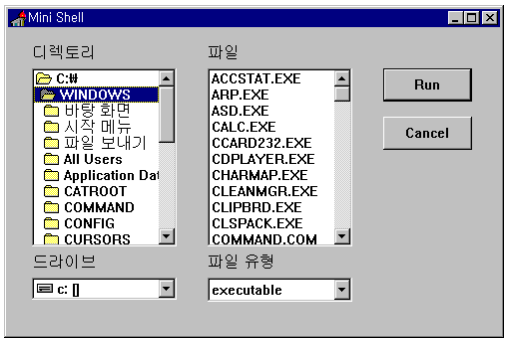

OnChange 이벤트를 사용하는 것 보다는 속성으로 컴포넌트를 연결하는 방법을 사용하는 것이 훨씬 더 빠르고 편리하다. 델파이와 같은 비주얼 개발 툴에서는 가급적이면 코드를 작성하지 않고 속성으로 프로그램을 작성하는 것이 유리하다.

# 다. 미니쉘 작성

11jang
mshell1

파일 관련 컴포넌트를 사용하여 간단한 쉘 프로그램을 만들어 보자. 쉘 프로그램이란 다른 프로그램을 실행시키는 기능을 가지며 실행시킬 프로그램을 선택하는 문제만 해결하면 별 어려움없이 만들 수 있다. 파일 관련 컴포넌트를 사용하여 파일을 선택하도록 하고 더블클릭시 프로그램을 실행시키도록 해 보자.

컴포넌트를 배치한다. 파일 관련 컴포넌트 네 개와 버튼 두 개가 필요하다. 컴포넌트를 배치한 모습보다는 아예 실행시의 모습을 보인다. 컴포넌트의 배치는 물론 꼭 이와 같을 필요는 없으며 적당히 보기 좋은 위치에 배치해도 상관없다.




2 속성을 설정하여 컴포넌트들을 연결해 준다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 드라이브 콤보 | DirList | DirectoryListBox1 |
| 디렉토리 리스트 | FileList | FileListBox1 |
| 필터 콤보 | FileList | FileListBox1 |
| Run 버튼 | Name | BtnRun |
| | Caption | Run |
| | Default | True |
| Cancel 버튼 | Name | BtnCancel |
| | Caption | Cancel |
| | Cancel | True |
| 폼 | Name | ShellForm |
| | Caption | Mini Shell |

Run 버튼은 Default 속성을 True로 만들어 주어 파일을 선택한 후 Enter 키만 누르면 프로그램이 실행될 수 있도록 하였으며 Cancel 버튼은 Cancel 속성을 True로 만들어 Esc 버튼을 누르면 프로그램을 끝낼 수 있도록 하였다.

3 파일 리스트 박스에서 프로그램을 더블클릭할 때 프로그램을 실행하는 코드를 작성한다. 파일 리스트 박스의 OnDblClick 이벤트에 다음 코드를 작성한다.

```
procedure Tshellform.FileListBox1DblClick(Sender: TObject);
begin
```

```
  WinExec(PChar(FileListBox1.FileName),SW_SHOWNORMAL);
end;
```

다른 프로그램을 실행시킬 때는 WinExec 함수를 사용한다. 인수로는 프로그램의 이름과 실행시의 프로그램 상태를 지정해 준다. 파일 리스트 박스에서 선택된 파일 이름은 FileName 속성으로 얻을 수 있다. 단 WinExec 함수는 프로그램 이름을 전달하는 첫 번째 인수로 Null 종료 문자열을 사용하므로, 파스칼형의 문자열을 PChar로 캐스팅해 주어야 한다.

파일을 더블클릭하는 것과 파일을 클릭하여 선택한 후 Run 버튼을 누르는 것이 동일한 효과를 가져오도록 하기 위해 이 이벤트를 Run 버튼의 OnClick 이벤트로 연결시켜 주고 Cancel 버튼의 OnClick 이벤트에는 다음 코드를 작성하여 프로그램을 끝낼 수 있도록 해 준다.

```
procedure Tshellform.BtnCancelClick(Sender: TObject);
begin
  Close;
end;
```

# 11-2 공통 대화상자

델파이가 제공하는 컴포넌트들은 역시 우수하며 사용하기 편리하다. 앞에서 만들었던 파일 선택 대화상자를 도스 상에서 직접 만들려면 아마 몇달은 족히 걸릴 것이다. 특히 그 중에서도 디렉토리 리스트 박스는 스스로 디렉토리 구조를 인식하는 총명함까지 가지고 있으며 컴포넌트끼리의 연결도 무척이나 쉽고 직관적이다.

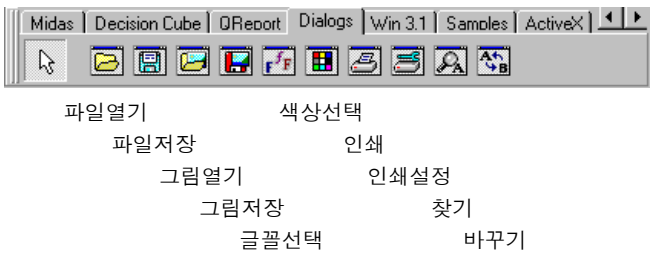
그러나 사람의 욕심이란 정말로 끝이 없는지 이런 편리한 컴포넌트보다 더 편리한 것을 원하게 된다. 파일을 선택해야 할 일은 종종 일어나며 파일 오픈 대화상자란 잘 만들어봐야 다 비슷비슷한 모양을 가지므로 아예 대화상자 자체를 컴포넌트로 제공하는 것이 더 편리하지 않을까 하는 생각이 든다. 그래서 마우스로 두 번만 클릭하면 그런 복잡한 대화상자를 만들어 낼 수 있으면 하고 꿈꿀 것이다.

많은 사람들이 오랫동안 바라던 일은 항상 현실로 나타나는 법, 델파이는 그런 편리한 컴포넌트를 제공한다. 아니 정확하게 말하자면 델파이가 제공하는 것이 아니라 윈도우즈가 제공하는 기능이다. 컴포넌트 팔레트의 열한번째 페이지에 Dialogs 페이지가 있으며 이 페이지에 있는 컴포넌트들이 이런 편리한 대화상자들을 담고 있다.

어떤 종류의 프로그램이든지 윈도우즈에서 돌아가는 모든 프로그램이 항상 사용하는 대화상자라 하여 이런 대화상자를 공통 대화상자(Common DialogBox)라고 한다.

**그림**

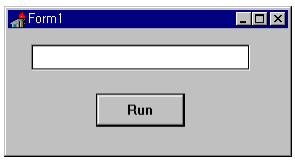
공통 대화상자를 제공하는 컴포넌트




11jang
mshell2

여기서는 이 중에서 가장 대표적인 파일 오픈 대화상자를 사용하여 더욱 간단한 방법으로 쉘 프로그램을 만들어 볼 것이다. 그야 말로 원터치로 프로그램을 만들 수 있다.

1 새 프로젝트를 시작하고 버튼과 에디트 컴포넌트만 배치한 후 폼의 크기를 적당히 축소한다.



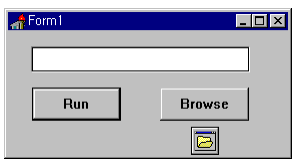

에디트 박스의 Text 속성은 모두 지우고 적당히 크기를 늘려 주자. 버튼은 BtnRun, Run으로 Name과 Caption 속성을 설정하고 Default 속성을 True로 만들어 준다. 에디트 박스에 실행할 프로그램의 이름을 입력한 후 Run 버튼을 누르면 프로그램을 실행하도록 만들 것이다.

2 Run 버튼의 OnClick 이벤트에 다음 코드를 작성한다.

```
procedure TForm1.BtnRunClick(Sender: TObject);
begin
  WinExec(PChar(Edit1.Text ),SW_SHOWNORMAL);
end;
```

에디트 박스에서 입력된 문자열을 Null 종료 문자열로 바꾸어 WinExec 함수로 전달하여 프로그램을 실행시킨다. SOL.EXE나 Explorer.EXE를 입력한 후 Run 버튼을 눌러 직접 확인해 보자. 여기까지만 만들어도 사실 쉘 프로그램이라고 할 수는 있지만 실행시킬 프로그램의 이름을 반드시 알고 있어야 하기 때문에 불편하다. 그래서 실행시킬 프로그램을 별도의 대화상자를 열어서 입력받도록 해 보자.

3 Browse 버튼과 OpenDialog 대화상자 컴포넌트를 배치한다. Browse 버튼을 누르면 파일 오픈 대화상자가 열리고 이 대화상자에서 선택한 파일 이름을 에디트에 복사해 주도록 해 보자.




추가한 두 컴포넌트의 속성은 다음과 같이 설정한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| Browse 버튼 | Name | BtnBrowse |
| | Caption | Browse |
| 파일 오픈 대화상자 | Filter | 실행 파일\|*.exe |

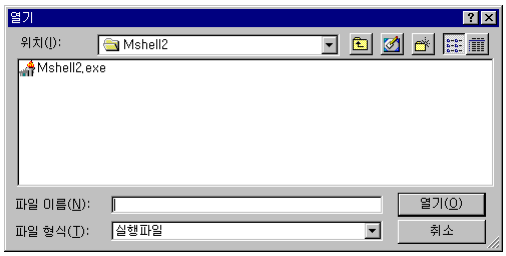
4 Browse 버튼의 OnClick 이벤트 핸들러를 작성한다.

```
procedure TForm1.BtnBrowseClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Edit1.Text:=OpenDialog1.FileName;
end;
```

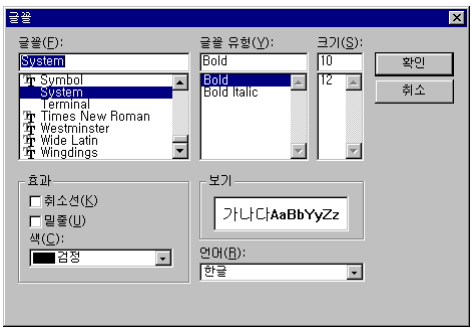
먼저 OpenDialog 컴포넌트의 Execute 메소드로 대화상자를 실행시켜 파일 선택을 하도록 한 후 선택한 파일 이름을 Edit1의 Text 속성에 복사해 준다. 대화상자에서 선택한 파일 이름이 OpenDialog1.FileName 속성에 대입되므로 이 값을 다시 Edit1.Text로 대입해 준다. 여기서 대화상자를 실행시킬 때 조건문을 사용하는 이유는 대화상자에서 파일 이름을 제대로 선택했는가를 점검하기 위해서이다. 파일 오픈 대화상자에서 취소 버튼을 누르면 Execute 메소드의 리턴값이 False가 되어 파일 이름이 복사되지 않도록 한다.

5 다시 컴파일한 후 실행해 보자. 실행중에 Browse 버튼을 누르면 다음과 같은 파일 오픈 대화상자가 열릴 것이다. 여기서 파일 이름을 선택하고 대화상자를 닫으면 에디트에 파일 이름이 나타나게 된다.

그림
파일 열기 대화상자

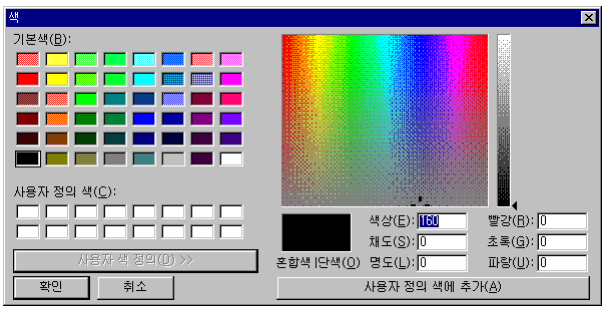



파일을 선택하는 대화상자도 별도의 폼으로 만들 필요가 없으며 컴포넌트를 배치하여 Execute 메소드만 실행하면 된다. 파일 오픈 대화상자 외에도 파일 저장 대화상자, 색상 선택 대화상자, 폰트 선택 대화상자 등등의 컴포넌트가 제공되므로 직접 사용해 보기 바란다. 다음에 몇 가지 공통 대화상자를 보인다.




그림

폰트 선택 공통 대화상자




그림

색상 선택 공통 대화상자

이 중 색상 선택 공통 대화상자는 8장의 MDraw2 예제에서 이미 사용해 본 바 있으며 폰트 선택 대화상자는 이 장의 제일 끝에 있는 Editor4 예제에서 사용해 볼 것이다.

나머지 대화상자의 사용법도 파일 오픈 대화상자의 사용법과 크게 틀리지 않으므로 굳이 설명할 필요는 없을 것 같다. 자세한 사항은 레퍼런스를 참조하기 바란다. 윈도우즈가 운영체제 차원에서 자주 사용되는 대화상자를 제공하기 때문에 윈도우즈는 더욱 직관적이며 사용하기 편리해진다. 프로그램마다 파일을 선택하는 방법이 다 다르다면 얼마나 골치아파지겠는가?

# 11-3 파일 입출력

## 가. 입출력 절차

파일로 데이터를 입출력할 정도의 프로그램이면 이미 작은 규모는 아니다. 파일 입출력이 프로그램을 처음 짜 보는 사람에게는 그만큼 어렵다는 뜻이다. 디스크의 파일로부터 데이터를 입력, 출력하려면 다음 5가지 단계를 반드시 따라야 한다.

### 1 파일 변수를 선언한다

파일 변수란 파일을 제어하기 위해 디스크의 실제적인 파일을 가리키는 변수이며 흔히 파일 핸들(File Handle)이라고 한다. 디스크 상의 파일은 드라이브, 디렉토리, 파일 이름, 확장자 등 여러 가지 구성 요소로 이루어지기 때문에 입출력할 때마다 이런 구성 요소들을 일일이 지정해 주려면 무척 번거로와진다. 게다가 네트웍 환경에서의 UNC(Universal Naming Convention)까지 고려하자면 더욱 복잡해진다. 파일 변수란 이런 복잡한 구성 요소들을 대표하는 변수이며 디스크의 파일과 일단 한번만 연결해 주면 파일 변수를 통해 간단하게 파일 입출력을 할 수 있다. 파일 변수는 TextFile이라는 데이터형을 가지므로 다음과 같이 선언한다.

```
var
  FH:TextFile;
```

물론 FH라는 이름은 변수의 이름이므로 MyFile, GameData 등과 같이 사용자가 마음대로 붙일 수 있다.

### 2 파일과 핸들을 연결한다

파일 핸들과 디스크 상의 실제 파일과 대응시켜 준다. 파일 핸들이 어떤 파일을 가리키는가를 지정해 주는 작업이며 AssignFile 함수를 사용한다.

**기본 형식**

AssignFile(핸들, 파일이름)

핸들은 앞에서 선언한 파일 변수이며 파일 이름은 어느 디렉토리에 있는 어떤 파일인가를 지정하는 문자열이다. 경로를 설정할 수도 있고 현재 디렉토리의 파일이면 파일 이름만 적을 수도 있다.

```
AssignFile(FH,'C:₩AUTOEXEC.BAT');
```

이 명령에 의해 FH라는 파일 변수는 'C:₩AUTOEXEC.BAT' 파일과 연결되며 차후 FH 핸들을 통해 이 파일로부터 데이터를 읽거나, 이 파일에 데이터를 출력한다.

## 3 파일을 오픈한다

다음은 파일을 오픈해야 한다. 파일을 오픈한다는 것은 파일 입출력을 위한 준비를 하는 과정이다. 파일 입출력을 하기 위해서는 읽어온 데이터를 저장할 메모리가 필요하며 또한 파일을 어디쯤 읽고 있는가, 어떤 형태의 파일인가 등의 정보를 기억할 여러 가지 내부적인 변수들이 필요하고 에러 처리를 위한 준비가 필요하다. 이런 일체의 작업을 통틀어 파일 오픈이라고 한다. 파일을 오픈하는 프로시저는 입출력 동작에 따라 세 가지 종류가 있다.

**Rewrite(핸들)**

파일로 데이터를 출력하기 위해 파일을 오픈한다. 만약 파일이 디스크 상에 없으면 파일을 직접 만들고 파일이 있으면 기존 파일을 삭제한 후 다시 만든다.

**Reset(핸들)**

파일로부터 데이터를 입력하기 위해 파일을 오픈한다. 읽기 위해서 오픈한 것이므로 데이터를 출력할 수는 없다. 만약 열고자 하는 파일이 없다면 에러가 발생한다.

**Append(핸들)**

기존에 있는 파일에 데이터를 추가로 출력하기 위해 파일을 오픈한다.

파일로 어떤 작업을 할 것인가에 따라 적당한 오픈 프로시저를 선택하여 사용한다.

## 4 입출력을 한다

파일로부터 데이터를 읽거나 파일에 데이터를 출력하는 실질적인 입출력 과정이다. 여러 가지 방법으로 입출력이 가능하지만 보편적으로 다음 두가지 함수를 사용한다.

```
Readln(핸들, 변수);
Writeln(핸들, 출력할 값);
```

Readln이 데이터를 읽어들이는 함수이고 Writeln이 데이터를 파일로 출력하는 함수이다. 실제 입출력 예는 예제를 통해 실습해 본다.


Readln, Writeln과 비슷한 Read, Write 등의 함수도 있는데 이 함수들은 개행 코드를 삽입하지 않는다는 점이 다르다. Writeln으로 데이터를 출력하면 계속 새로운 행에 출력되지만 Write로 출력하면 한 행에 계속 붙여서 출력된다.

```
Writeln(FH, '우리집 강아지는 미친 강아지');
Writeln(FH, '학교갔다 돌아오면 야옹 야옹 야옹');
```

이렇게 호출하면 각 줄이 개행되어 출력되지만 Write를 쓰면 두 행이 붙어서 출력된다. 대개의 경우 Write보다는 Writeln을 쓸 일이 훨씬 더 많다.

## 5 파일을 닫는다

입출력을 위해 파일을 열었으면 원래대로 닫아 주어야 한다. 파일을 닫는다는 말은 혹시라도 아직 출력이 안되고 메모리에 남아있는 데이터가 있으면 파일로 완전히 출력하고 입출력에 사용하던 메모리를 해제하는 동작을 말한다. 다음 함수를 쓴다.

```
CloseFile(핸들);
```

입출력이 끝난 후에 파일을 닫는 것은 무척 중요한 일이면서도 실수로 빼먹고 안하는 경우가 많은데 반드시 해 주어야 한다. 그렇지 않으면 멀쩡하던 파일이 사라진다거나 깨지는 현상이 일어난다.

이상이 파일로부터 입출력을 하는 다섯 가지 과정이다. 이 중 하나라도 생략할 수 없으며 순서를 어겨서도 안된다. 반드시 이 절차를 따라 주어야 한다. 파일 입출력을 처음 해 보는 사람은 왜 이렇게 복잡할까 하는 불만을 가질지도 모르겠다. 데이터 한번 읽으려면 변수 만들고 열었다, 닫았다 도대체 이게 뭐야?

읽어(이 파일에서, 요런 요런 데이터를, 이 변수에다, 요만큼);

요런 편리한 함수가 있다면 얼마나 좋을까만 현실은 그렇지 못하다. 왜 파일 입출력이 이렇게 복잡하고 까다로운가 하면 조금이라도 속도를 빠르게 하기 위해서이다. 파일로 1바이트씩 출력을 한다고 해보자. 그러면 십만 바이트를 읽는데 이론적으로 디스크가 십만바퀴를 돌아야 한다는 얘기가 되며 삼십만 바이트를 읽는 데 빨라도 한시간은 걸린다. 그래서 입출력을 위한 임시 메모리(buffer라고 한다)를 만들고 얼마정도씩 모았다가 한꺼번에 입출력을 하도록 하였고 그러다 보니 메모리를 준비시키는 과정이 필요하고 입출력이 끝난 후에 메모리를 정리하는 과정도 필요하게 되는 것이다. 파일 입출력 과정이 이렇게 복잡한 이유는 파일이 컴퓨터(=CPU) 외부에 존재하기 때문이다.

# 나. 파일 쓰기

11jang
wrfile

앞에서 배운 절차대로 파일 쓰기를 해 보자. 어쨌든 화면에 보이는 폼은 있어야 하므로 프로젝트를 시작하고 폼에 버튼 하나만 배치한다. 그리고 그 버튼의 OnClick 이벤트에 다음 코드를 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FH:TextFile;      {파일 변수 선언}
begin
  AssignFile(FH,'c:₩testfile.txt');  {할당}
  Rewrite(FH);                {오픈}
  Writeln(FH,'Beautiful Korea!');  {출력}
  CloseFile(FH);              {닫기}
end;
```

여기서는 FH가 파일 변수이다. 변수를 선언하고 이 변수에 c:₩testfile.txt라

는 파일을 할당하고 Rewrite로 파일을 오픈하였다. 파일이 루트 디렉토리에 없다면 새로 생성되며 Writeln에 의해 파일로 문자열을 출력하고 CloseFile로 마지막 뒷정리를 하였다. 과연 파일이 생성되었는가 직접 확인해 보아라.

Writeln문은 문자열 뿐만 아니라 여러 가지 형식의 데이터를 파일로 출력할 수 있으며 한꺼번에 여러 개를 출력하는 것도 가능하다. 출력할 데이터를 콤마로 끊어 죽 나열해 주면 된다. 다음과 같이

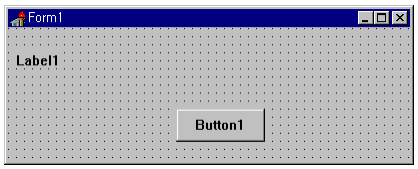
```
Writeln(FH,345,3.14,567,'korea');
```

예제의 Writeln문을 이렇게 바꾸고 난 후 직접 생성된 파일을 확인해 보기 바란다.

# 다. 파일 읽기

11jang
rdfile

이번에는 파일로부터 데이터를 읽어 보자. 쓰기와 마찬가지로 버튼을 누르면 파일로부터 데이터를 읽는 시범을 보이도록 하되 읽은 데이터가 제대로 읽혀졌는지 확인을 해야 하므로 레이블도 하나 배치하도록 하자.

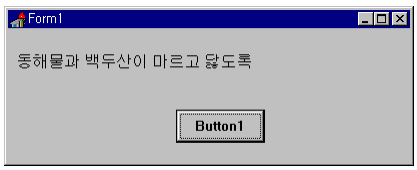



그리고 읽어야 할 파일이 있어야 하므로 간단한 텍스트 파일을 준비해 둔다. 프로젝트를 저장하고 프로젝트가 있는 디렉토리에 LOVE.TXT라는 텍스트 파일을 만든다. 내용은 아무래도 상관없다. 이 예제에서는 LOVE.TXT에 애국가 가사를 입력해 두었다. 버튼의 OnClick 이벤트 코드를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  LOVE:array [1..10] of string;     {배열 준비}
  FH:TextFile;                      {파일 변수}
  i:Integer;                        {루프 제어 변수}
begin
AssignFile(FH,'LOVE.TXT');          {할당}
Reset(FH);                          {오픈}
```

```
for i:=1 to 10 do
  Readln(FH,LOVE[i]);              {읽기}
CloseFile(FH);                     {닫기}
Label1.Caption:=LOVE[1];           {출력}
end;
```

쓰는 것과는 달리 파일로 읽어들인 데이터를 저장할 변수가 필요하므로 문자열 10개를 저장할 수 있는 LOVE라는 문자열 배열을 선언하였다. Readln문에 의해 첫 행부터 차례대로 읽혀져 LOVE[1]~LOVE[10]까지의 변수에 저장한다. 이렇게 저장한 데이터 중 LOVE[1]만 시범적으로 레이블로 출력해 보았다. 실행 결과는 다음과 같다.

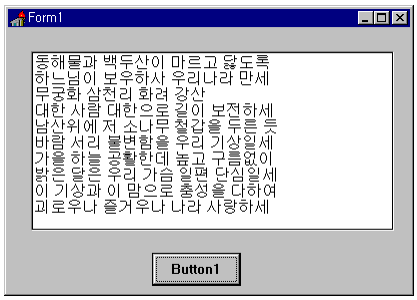



여기서는 텍스트 파일에서 문자열을 읽는 시범만 보이는 간단한 예제를 만들어 보았다. 이 정도 예제를 이해한다면 실전에서 응용하기에 무리가 없을 것이다. 14장의 DelCD 예제에 파일로부터 레코드를 읽어들이는 코드가 있는데 참고하기 바란다.

텍스트 파일로부터 데이터를 읽는 또 다른 방법은 LoadFromFile 메소드를 이용하는 것이다. 이 메소드는 TStrings 오브젝트를 가지는 모든 컴포넌트(예:메모,아웃라인)에 사용할 수 있으며 파일을 열거나 파일 변수를 준비할 필요없이 파일 이름만으로 데이터를 읽어올 수 있다. 또한 원할 경우 디자인시에도 문자열 리스트 편집기에서 파일의 내용을 미리 읽어놓는 것도 가능하다. 실습을 위해 메모 컴포넌트와 버튼 컴포넌트를 폼에 배치한 후 버튼의 OnClick 이벤트를 다음과 같이 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('LOVE.TXT');
end;
```

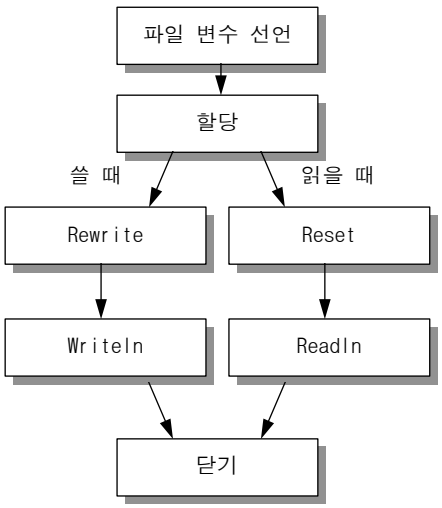
버튼을 누르면 Memo1의 Lines 오브젝트의 LoadFromFile 메소드로 텍스트 파일을 읽어 메모 컴포넌트로 출력하게 된다. 실행 결과는 다음과 같다.




LoadFromFile과 반대되는 메소드는 SaveToFile 메소드이며 텍스트 파일로 문자열을 출력할 때 사용한다. 물론 이런 편리한 메소드를 사용하는 것이 빠르고 간단하고 좋기는 하다. 하지만 직접 텍스트 파일을 읽어들여 변수에 저장해 놓고 사용하는 것은 많은 다양한 변화를 줄 수 있고 변수를 다방면에 사용할 수 있다는 비교할 수 없는 장점이 있다. 텍스트 파일 입출력 과정을 간단하게 그림으로 정리하면 다음과 같다.

**그림**

파일 입출력 절차




# 라. INI 파일

윈도우즈에서 파일 입출력을 하는 또 하나의 예는 초기화 파일인 INI 파일로부터 정보를 읽고 쓰는 작업이다. INI 파일은 일종의 텍스트 파일이므로 텍스트 파일의 읽기 쓰기 함수를 사용하여 액세스할 수도 있지만 그보다 더 편리한 방법이 있다. 델파이는 INI 파일 입출력을 위해 TIniFile 오브젝트를 제공하며 이 오브

젝트에 포함된 메소드를 사용하면 간편하게 INI 파일 입출력을 할 수 있다.

물론 INI 파일 입출력을 위해 윈도우즈 API 함수를 사용하는 방법도 있으며 10장에서 이미 그 예를 직접 만들어 보았다. 하지만 API 함수를 사용하는 것 보다 TIniFile 오브젝트를 사용하는 것이 훨씬 더 쉽다. TIniFile 오브젝트는 IniFiles 유닛에 의해 제공되며 다음과 같은 메소드가 정의되어 있다.

| 메소드 | 동작 |
|---|---|
| ReadBool | 불린값 읽음 |
| ReadInteger | 정수값 읽음 |
| ReadString | 문자열 읽음 |
| WriteBool | 불린값 출력 |
| WriteInteger | 정수값 출력 |
| WriteString | 문자열 출력 |

대표적으로 ReadInteger 메소드만 살펴보면 ReadInteger(Section, Ident, Default); 형태로 선언되어 있으며 인수는 차례대로 섹션, 항목, 디폴트값이다. 나머지도 이와 동일한 형태를 가지고 있다. 이 메소드를 쓰기 위해서는 먼저 TIniFile형의 오브젝트 변수를 선언하고 오브젝트를 생성시켜야 하며 사용 후에는 Free 메소드로 해제해 주어야 한다.

```
var
  변수:TIniFile;
begin
  변수:=TiniFile.Create('파일 이름');
  {정보 입출력}
  변수.Free;
end;
```

11jang
inifile

10장에서 API 함수 호출을 위해 만들었던 예제를 이번에는 TIniFile 오브젝트를 사용하여 만들어 보자. 별다른 컴포넌트는 필요없으며 버튼만 하나 배치해

둔다.


INI 파일 제어를 위해 다음과 같이 TIniFile 오브젝트를 전역 변수로 선언한다.

```
var
  Form1: TForm1;
  POSSIZE:TIniFile;
```

버튼의 OnClick 이벤트 핸들러에서 Close 메소드를 호출하여 폼을 닫도록 하였다. 프로그램의 위치 정보는 폼이 파괴될 때인 FormDestroy에서 저장하고 폼이 다시 만들어질 때인 FormCreate 이벤트에서 정보를 읽어와 폼의 위치를 옮기도록 하였다.

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

procedure TForm1.FormCreate(Sender: TObject);
begin
  POSSIZE:=TIniFile.Create('iniexa.ini');
  Form1.Left:=POSSIZE.ReadInteger('POS','Left',100);
  Form1.Top:=POSSIZE.ReadInteger('POS','Top',100);
  Form1.Width:=POSSIZE.ReadInteger('SIZE','Width',300);
  Form1.Height:=POSSIZE.ReadInteger('SIZE','Height',250);
  POSSIZE.Free;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  POSSIZE:=TIniFile.Create('iniexa.ini');
  POSSIZE.WriteInteger('POS','Left',Form1.Left);
  POSSIZE.WriteInteger('POS','Top',Form1.Top);
  POSSIZE.WriteInteger('SIZE','Width',Form1.Width);
  POSSIZE.WriteInteger('SIZE','Height',Form1.Height);
```

```
  POSSIZE.Free;
end;
```

마지막으로 TIniFile 오브젝트는 Inifiles 유닛에 선언되어 있으므로 이 유닛을 폼의 uses절에 포함시켜 주어야 한다.

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,IniFiles;
```

처음 프로그램이 실행될 때는 디폴트값이 (100,100) 위치에 폭 300, 높이 250으로 되어 있다. 폼을 적당히 이동하고 크기를 변경하고 종료한 후 다시 실행해 보면 종료전의 위치에 그대로 폼이 열릴 것이다. 폼이 종료될 때 INI 파일에 위치를 기억해 두고 시작할 때 다시 읽어오기 때문이다. 윈도우즈 디렉토리를 보면 Iniexa.ini 파일이 있을 것이며 이 파일에는 폼의 최후 위치와 크기가 기록되어 있다.

```
[POS]
Left=790
Top=550

[SIZE]
Width=300
Height=250
```

참고로 이렇게 만들어진 INI 파일은 윈도우즈 디렉토리에 저장되는데 이렇게 되면 윈도우즈 디렉토리가 무수한 INI 파일로 넘쳐나 시스템 속도에 별로 좋지 않은 영향을 끼치게 된다. 다음 방법을 사용하면 INI 파일을 프로그램 실행 파일과 같은 디렉토리에 생성되도록 할 수 있다.

```
var
  Path:String;
begin
  Path:=ExtractFilePath(ParamStr(0))+'IniExa.ini';
  POSSIZE:=TIniFile.Create(Path);
```

ParamStr(0)에는 실행 파일의 전체 경로가 들어 있는데 이 문자열을 ExtractFilePath 함수로 경로 부분만 분리한 후 이 디렉토리에 INI 파일을 생성한 것이다. 응용 프로그램이 개별적으로 사용하는 INI 파일은 이런 식으로 자기

디렉토리에 생성하는 것이 좋다.

기본적인 데이터형 입출력에 사용되는 메소드 외에도 잘 사용되지는 않지만 날짜나 시간, 실수형을 저장할 수 있는 메소드도 있다.

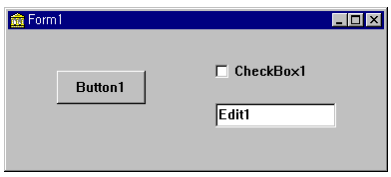
| 메소드 | 대상 |
|---|---|
| ReadFloat, WriteFloat | 실수형 |
| ReadTime, WriteTime | 시간 |
| ReadDate, WriteDate | 날짜 |
| ReadDateTime, WriteDateTime | 날짜, 시간 |

# 마. 레지스트리

11jang
Regi

위에서 보다시피 INI 파일을 사용하는 것은 무척이나 쉽고 직관적이며 기능적으로 문제가 없다. 그런데 윈 95부터는 INI 파일 대신 레지스트리에 정보를 보관할 것을 더 권장하고 있다. INI 파일은 진짜로 디스크에 저장되는 파일이기 때문에 읽고 쓰는 속도가 느리며 윈도우즈 디렉토리를 혼잡스럽게 만드는 주범이기 때문이다.

레지스트리는 운영체제의 모든 설정 상태를 저장하는 시스템 정보 데이터 베이스이며 계층적으로 광범위한 정보를 저장하도록 해 준다. 윈도우즈를 조금이라도 써 본 사람이라면 레지스트리에 대해서는 잘 알 것이다. 앞에서 만들었던 Inifile 예제를 조금 수정하여 이번에는 레지스트리에 정보를 저장하는 예제를 만들어 보도록 하자. 이번에는 폼의 위치뿐만 아니라 프로그램의 설정상태도 같이 저장해 보기 위해 버튼, 체크 박스, 에디트를 각각 배치하였다.




버튼을 누르면 프로그램을 종료하되 체크 박스에 설정된 Boolean형 옵션과 에디트에 설정된 문자열 옵션을 레지스트리에 같이 저장해 볼 것이다. 레지스트리를 사용하는 방법은 INI 파일을 사용하는 방법과 거의 유사하다. 다만 레지스트리를 지원하는 오브젝트가 TRegistryIniFile이라는 것과 이 오브젝트가

Registry 유닛에 선언되어 있다는 점이 다를 뿐이다. 레지스트리의 구조는 상당히 복잡하며 초보자에게는 난해하지만 이 오브젝트를 사용하면 이런 복잡함을 몰라도 마치 INI 파일을 쓰듯이 레지스트리에 정보를 저장하고 읽어올 수 있다. 이 오브젝트를 사용하려면 폼의 uses절에 Registry 유닛 이름을 적어 주어야 한다.

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,Registry;
```

그리고 전역 변수로 TRegistryIniFile 오브젝트형의 POSSIZE를 선언한다.

```
var
  Form1: TForm1;
  POSSIZE:TRegistryIniFile;
```

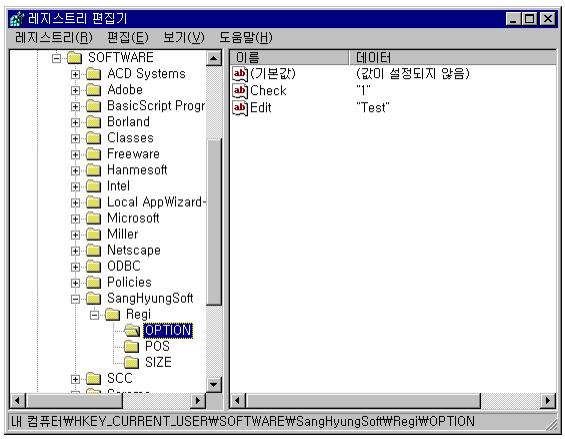
이제 버튼의 OnClick 이벤트 핸들러와 폼의 FormCreate, FormDestroy 핸들러를 작성한다.

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

procedure TForm1.FormCreate(Sender: TObject);
begin
  POSSIZE:=TRegistryIniFile.Create('Software₩SangHyungSoft₩Regi');
  Form1.Left:=POSSIZE.ReadInteger('POS','Left',100);
  Form1.Top:=POSSIZE.ReadInteger('POS','Top',100);
  Form1.Width:=POSSIZE.ReadInteger('SIZE','Width',400);
  Form1.Height:=POSSIZE.ReadInteger('SIZE','Height',250);

  CheckBox1.Checked:=POSSIZE.ReadBool('OPTION', 'Check', False);
  Edit1.Text:=POSSIZE.ReadString('OPTION', 'Edit', 'Nothing');
  POSSIZE.Free;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  POSSIZE:=TRegistryIniFile.Create('Software₩SangHyungSoft₩Regi');
  POSSIZE.WriteInteger('POS','Left',Form1.Left);
  POSSIZE.WriteInteger('POS','Top',Form1.Top);
```

```
  POSSIZE.WriteInteger('SIZE','Width',Form1.Width);
  POSSIZE.WriteInteger('SIZE','Height',Form1.Height);

  POSSIZE.WriteBool('OPTION', 'Check', CheckBox1.Checked);
  POSSIZE.WriteString('OPTION', 'Edit', Edit1.Text);
  POSSIZE.Free;
end;
```

앞에서 만든 Inifile 예제와 코드가 거의 비슷하되 체크 박스의 옵션과 에디트의 문자열을 입출력하는 코드가 덧붙여졌을 뿐이며 TIniFile 대신 TRegistryIniFile 오브젝트를 사용했다. 논리형 옵션을 저장(읽기)할 때는 WriteBool(ReadBool) 메소드를 사용하며 문자열 옵션을 저장(읽기)할 때는 WriteString(ReadString) 메소드를 사용하면 된다.

레지스트리의 경로는 관행상 HKEY_CURRENT_USER/Software 아래에 '회사명/제품명/버전'으로 작성하도록 되어 있는데 이 예제는 이 관행을 따르되 단 버전만 생략하고 있다. 실행해 보면 폼의 위치와 크기뿐만 아니라 옵션의 설정 상태까지도 그대로 복구될 것이다. 레지스트리에 어떤 값이 저장되어 있는지 RegEdit로 직접 확인해 보기 바란다.




SanyHyungsoft₩Regi 아래 POS, SIZE, OPTION 등의 서브 키들이 만들어져 있고 이 키안에 값들이 저장되어 있다.

참고로 TRegistryIniFile 오브젝트는 델파이 4.0버전에 추가된 것이며 델파이 3.0버전에서는 TRegIniFile이라는 오브젝트가 사용되었었다. 두 오브젝트가 기능적으로나 사용방법에 있어 별반 차이가 없지만 가급적이면 최신의 TRegInistryIniFile 오브젝트를 사용하는 것이 좋다고 한다.

# 바. 퀴즈

11jang
quiz

파일 읽기를 배웠으니 여기서는 이 기술을 응용하여 간단한 퀴즈 게임을 작성해 보도록 하자. 퀴즈 프로그램은 많은 문제를 내장하고 있어야 하며 또한 문제는 코드와 분리되는 것이 좋다. 직접 배열로 문제를 정의하는 것보다는 텍스트 파일에 문제를 작성해 두고 프로그램에서는 파일에서 문제를 읽어오는 것이 여러 가지 면에서 좋지 않을까 하고 생각된다.

그래서 이 프로그램에서는 QUIZ.TXT 라는 텍스트 파일을 만들고 이 파일에 문제를 수록한 후 FormCreate 이벤트에서 문제를 읽어 들이도록 하였다. 문제가 수록된 QUIZ.TXT 는 다음과 같은 형식으로 작성되어 있다.

```
한 마지기는 몇 평인가
100 평
200 평
300 평
2
CPU 는 무엇의 약자인가
Central Processing Unit
Computer Processing Unit
Computer Power Unit
1
데미안의 저자는
레오 버스카 글리아
버넷트
헤르만 헤세
3

..... 이하 생략 .....
```

한 문제는 모두 다섯 행을 차지하며 첫 행에 문제를 작성하고 이어지는 행에 보기 세 개와 정답을 기입해 두었다. 프로그램에서는 문제를 다음과 같은 Tmunje 레코드 배열에 저장한다.

```
type
 Tmunje=record
  mun:string[50];
```

```
    b1:string[30];
    b2:string[30];
    b3:string[30];
    ans:integer;
  end;
var
munje:array [1..total] of Tmunje;
```

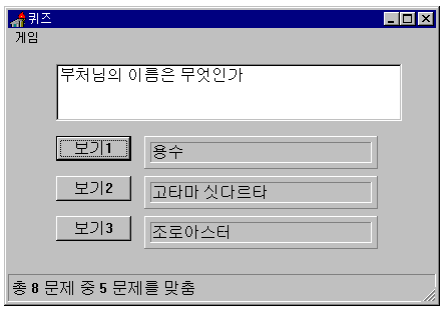
여기서 total 은 문제의 총 개수를 정의하는 상수이며 23 으로 정의되어 있다. 문제의 개수를 늘리려면 이 상수를 늘려주고 QUIZ.TXT 파일에 문제를 추가로 더 작성해 주기만 하면 된다. 텍스트 파일에서 문제를 읽어들이는 코드는 프로그램이 처음 시작되는 FormCreate 이벤트에 다음과 같이 작성되어 있다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  F1:TextFile;
  i:integer;
  st:string;
begin
AssignFile(F1,'Quiz.Txt');  {할당}
Reset(F1);                  {오픈}
{문제를 순서대로 읽어와 배열에 담는다}
for i:=1 to total do
  begin
  readln(F1,munje[i].mun);
  readln(F1,munje[i].b1);
  readln(F1,munje[i].b2);
  readln(F1,munje[i].b3);
  readln(F1,st);
  munje[i].ans:=StrToInt(st);
  end;
CloseFile(F1);
Randomize;
end;
```

앞에서 설명한 대로 파일을 할당하고 오픈한 후 readln 함수로 문자열을 차례대로 읽어들여 munje 레코드의 해당 필드에 대입하고 있다. 이렇게 코드와 데이터를 분리시켜 둠으로 해서 문제를 바꾸거나 문제의 수를 늘리고자 할 때는 코드에 손 댈 필요없이 데이터 파일만 작성하면 되며 또한 프로그램을 짜는 사람과 문제를 만드는 사람이 분업을 할 수 있어서 좋다.

화면 구성은 다음과 같이 간단하게 디자인하였다. 아예 실행중의 화면을 보이도록 한다. 메모 컴포넌트에 문제를 출력하고 패널에 보기를 출력하며 버튼으로 보기를 선택하도록 하였다. 문제를 맞춘 결과는 아래쪽의 상황선에 문자열로 출력된다.

그림
퀴즈 프로그램




게임을 시작하려면 게임/시작 메뉴 항목을 선택하며 이 메뉴에서 Chulje 프로시저를 호출하여 문제를 출제한다.

```
procedure TForm1.Chulje;
begin
 nowmun:=Random(total)+1;
 Memo1.Text:=munje[nowmun].mun;
 Panel1.Caption:=munje[nowmun].b1;
 Panel2.Caption:=munje[nowmun].b2;
 Panel3.Caption:=munje[nowmun].b3;
end;
```

난수로 문제를 선택한 후 munje 레코드에서 문제는 메모로, 보기는 패널로 출력한다. 사용자는 문제를 읽은 후 버튼을 눌러 답을 하며 버튼의 OnClick 이벤트 핸들러에서는 사용자의 응답과 문제의 정답을 비교하여 판정을 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
{답과 버튼의 번호를 비교한다}
if (Sender as TButton).tag=munje[nowmun].ans then
 NO:=NO+1  {맞춘 개수 증가}
else
 NX:=NX+1; {틀린 개수 증가}
StatusBar1.SimpleText:='총 '+ IntToStr(NO+NX)+' 문제 중 '+
 IntToStr(NO) +' 문제를 맞춤';
Chulje;    {다시 문제 출제}
end;
```

세 버튼의 OnClick 이벤트가 모두 이 핸들러로 연결되어 있으며 버튼의 tag 속성을 보기의 번호와 일치시켜 놓아 버튼의 tag 속성을 읽어 사용자가 어떤 번호를 선택했는지 판정한다. 판정한 결과는 NO, NX 변수에 저장되며 상황선에 문자열로 맞춘 개수가 출력된다.

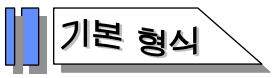
이 게임은 파일 읽기를 보이기 위해 작성되었으므로 세련된 게임 운영은 하지 않고 무조건 문제를 출제하고 답에 대한 판정만 해 준다. 좀 더 완벽한 퀴즈 게임을 만들려면 한번 출제된 문제는 다시 출제되지 않도록 하고, 시간 제한을 둔다거나 일정 개수의 문제를 출력한 후 게임 1회를 끝내도록 하는 것이 좋다. 이 부분은 여러분이 직접 해 보기 바란다.

# 11-4 타입드 파일

텍스트 파일은 입출력이 간단하다는 장점이 있기는 하지만 문장만을 저장할 수 있기 때문에 범용적인 자료를 저장하기에는 부적당하다. 정수나 실수 또는 레코드 등을 파일로 저장하기 위해서는 텍스트 파일을 사용할 수 없으며 타입드 파일(typed file)을 사용해야 한다. 델파이는 파스칼의 모든 변수들을 파일로 저장할 수 있도록 해 주며 그 방법이 바로 타입드 파일이다. 여기서 타입이 있다(typed)는 뜻은 파일로 저장되는 개별 데이터의 데이터형이 정해져 있다는 뜻이다.  적당한 번역이 없어서 그냥 발음대로 읽기로 한다.

## 가. 정수값을 담는 파일

타입드 파일을 사용하는 방법은 텍스트 파일을 사용하는 방법과 거의 동일하다. 단 타입드 파일 변수를 선언하는 형식만 차이가 있다. 타입드 파일 변수는 다음과 같이 선언한다.




```
var
  변수:file of 타입;
```

예를 들어 정수형의 값을 저장할 파일이라면 FH:file of integer; 등과 같이 선언한다. 파일을 열고 닫고 액세스하는 방법은 텍스트 파일의 경우와 동일하다. 정수형 값을 담는 파일을 만들고자 한다면 다음과 같이 코딩한다.

```
var
  FH:file of integer;  {정수형 타입드 파일 변수 선언}
  i:integer;
begin
  AssignFile(FH,'c:₩intfile');  {할당}
  Rewrite(FH);  {오픈}
  for i:=1 to 10000 do
    Write(FH,i);  {정수값 출력}
```

```
 CloseFile(FH); {닫음}
end;
```

FH는 정수형 값을 담는 파일의 핸들로 선언되었으며 c:₩intfile 과 연결되었다. 앞으로 이 파일에는 정수값이 출력되거나 정수값을 읽는 목적으로 사용된다. Write 함수로 정수값 1~10000까지를 파일로 출력하였다. 직접 도스상에서 intfile을 확인해 보면 크기가 정확하게 40000바이트이며 1~10000까지의 값을 저장하고 있을 것이다.

# 나. 레코드형 타입드 파일

정수값을 담는 타입드 파일은 어디까지나 예제를 위해 만든 것일뿐이며 타입드 파일을 제대로 사용하는 예는 레코드를 입출력하는 경우이다. 레코드는 사용자가 정의한 데이터 타입이지만 타입드 파일의 입출력 대상으로도 사용할 수 있다. 다음과 같이 선언된 레코드가 있다고 하자.

```
type
TMyRec=record
 Name:string[20];
 Age:Integer;
 Male:Boolean;
end;
```

이 레코드는 개인의 신상을 담는 세 개의 필드로 구성되어 있다. 만약 이런 레코드 100개를 모아 주소록 프로그램을 만들었다면 주소록은 파일로 저장되어야 하며 TMyRec 타입의 파일 변수를 만듦으로써 TMyRec형의 데이터를 담는 파일을 만든다.

```
var
 MyRec:array [1..100] of TMyRec;   {레코드 배열 선언}
 FH:file of TMyRec;   {레코드 타입의 파일 변수 선언}
 i:integer;
begin
```

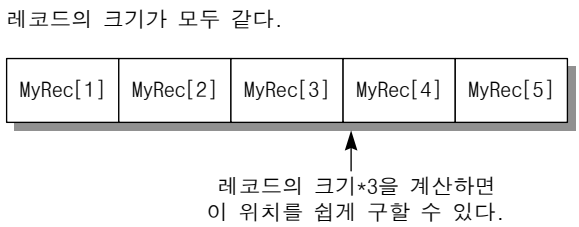
```
AssignFile(FH,'c:₩filerec'); {할당}
Rewrite(FH); {오픈}
for i:=1 to 100 do
  Write(FH,MyRec[i]); {출력}
CloseFile(FH); {닫기}
end;
```

TMyRec형의 배열 MyRec을 크기 100으로 선언하고 MyRec 전체를 c:₩filerec로 출력한다. 물론 위의 코드는 출력 예만 보이기 위해 제작되었기 때문에 MyRec에는 전혀 쓸모없는 값이 들어 있을 것이다. MyRec에 값을 대입해 준 후 파일로 출력하면 주소록 전체가 파일로 저장되며 이 정보는 Read 프로시저에 의해 원래대로 읽혀질 것이다.

# 다. 임의 접근

타입드 파일은 임의 접근(Random Access)이 가능하다. 임의 접근이란 순차 접근(Sequential Access)의 반대되는 말이며 파일에 있는 데이터 중 특정 위치의 데이터를 언제든지 읽을 수 있다. 왜 그런가 하면 타입드 파일에 저장되는 정보가 모두 같은 크기를 가지기 때문에 단순한 곱셈으로 원하는 데이터의 위치를 찾을 수 있기 때문이다.

**그림**
임의 집근




파일 입출력 과정은 항상 현재 위치를 기준으로 수행되며 레코드 하나를 읽으면 현재 위치가 레코드의 길이만큼 증가하여 다음 레코드를 읽을 준비를 한다. 현재 위치를 변경하려면 Seek 프로시저를 사용하며 인수로 파일 핸들과 몇번째 레코드를 읽을 것인가를 넘겨준다. 12번째 레코드를 읽고자 한다면 다음과 같이 코드를 작성하면 된다.

```
Seek(FH,12)        {12 번째 레코드로 이동}
```

```
Read(FH,MyRec[12]);
```

파일의 현재 위치를 구하려면 FilePos(FH) 프로시저를 사용하며 파일의 총 크기를 구하려면 FileSize(FH) 프로시저를 사용한다.

# 11-5 파일 관리

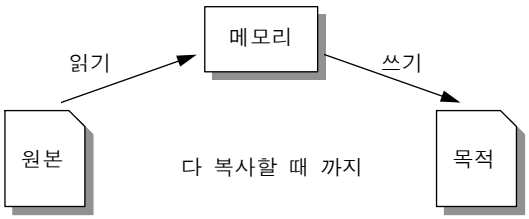
파일로부터 데이터를 입출력하는 것과 파일을 제어하는 것과는 다르다. 전자는 파일에 저장되어 있는 데이터가 그 대상이지만 후자는 파일 자체가 작업의 대상이 된다. 파일 관리에는 파일 복사, 삭제, 이동, 이름 변경, 파일 실행 등이 있다.

## 가. 파일 복사

윈도우즈 95가 나오기 전에는 시스템이 지원하는 파일 복사 함수가 없었다. 그리고 델파이 1.0에서도 파일을 복사하는 함수는 제공하지 않았으며 델파이 4.0버전까지도 델파이는 파일을 복사하는 함수를 제공하지 않는다. 그래서 파일 복사 함수는 직접 만들어서 사용해야만 했었다. 그런데 이런 사정이 윈도우즈 95의 Win32 API 함수에서 파일복사 함수를 제공함에 따라 달라지게 되었다. 일단은 파일 복사 함수를 직접 만들어서 사용해 보도록 하고 그 소스를 분석해 보기 바란다.

파일을 복사하는 원리 자체는 무척 간단하다. 일단 원본 파일에서 데이터를 읽어 잠시 보관해 둘 메모리를 준비한다. 그리고 원본에서 메모리로 읽은 후 목적 파일로 다시 쓰기를 한다. 단 메모리의 크기가 원본 파일의 크기보다 클 수는 없으므로 적당한 크기로 메모리를 할당한 후 읽고 쓰기를 원본 파일을 다 읽을 때까지 반복한다.

그림
파일 복사 과정




윈도우즈에서 메모리를 할당하고 사용하는 방법은 다음과 같다. 도스에서는 할당하고 사용하고 해제만 하면 되지만 윈도우즈에서는 운영체제가 메모리를 마음대로 이동시킬 수 있기 때문에 메모리의 이동을 금지시키는 잠금과 그 반대

동작인 잠금해제 과정이 추가된다.

그림
동적 메모리 할당 및 해제 절차


실제 소스는 다음과 같다. MyCopyFile이라는 함수 이름을 가지며 인수로 원본 파일과 목적 파일의 이름을 준다. 자세한 분석은 주석을 참고하여 직접 해 보기 바란다. MyCopyFile에서 사용하는 대부분의 함수는 델파이의 함수가 아닌 윈도우즈 API 함수들이다.

```
procedure MyCopyFile(src:PChar;dest:PChar);
var
  hsrc,hdest:THandle;  {파일 핸들}
  cbread:word;         {읽은 바이트 수}
  memhandle:Thandle;   {메모리 핸들}
  buffer:PChar;        {메모리 포인터}
begin
  hsrc:=_lopen(src,OF_READ);  {원본 파일을 연다}
  hdest:=_lcreat(dest,0);     {목적 파일을 만든다.}
  cbread:=30000;              {3 만 바이트 단위로 읽는다.}
  memhandle:=LocalAlloc(LMEM_FIXED,30000); {메모리 할당}
  buffer:=locallock(memhandle);  {잠근다.}
  while cbread<>0 do    {더 이상 읽을 데이터가 없을 때까지}
  begin
    cbread:=_lread(hsrc,buffer,30000);   {원본에서 읽어서}
    _lwrite(hdest,buffer,cbread);      {목적 파일로 출력}
  end;

  localunlock(memhandle); {메모리 해제}
  LocalFree(memhandle);
  _lclose(hsrc);          {파일 닫음}
  _lclose(hdest);
end;
```

파일이 없는 경우와 메모리가 부족한 경우의 에러 처리는 하지 않았다. 파일이 없는 경우의 에러 처리는 함수 내에서 해 주는 것보다 이 함수를 호출하는 곳에서 호출 전에 점검해 보도록 하는 것이 타당하다. 이 함수를 사용하여 파일을 복사하는 예제는 잠시 후에 작성해 보기로 한다.

Win32 API에서 제공하는 파일 복사 함수는 다음과 같다. 원형이 C형식으로

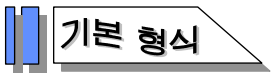
되어 있다.

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName,      // pointer to name of an existing file
  LPCTSTR lpNewFileName,           // pointer to filename to copy to
  BOOL bFailIfExists     // flag for operation if file exists
 );
```

첫 번째 인수는 소스 파일이며 두 번째 인수는 복사로 생성될 파일 이름이다. 세 번째 인수는 복사할 파일이 이미 있을 경우의 처리를 지정하는데 이 값이 False이면 덮어쓰기를 하고 True이면 복사를 하지 않는다.

# 나. 파일 삭제

파일을 삭제하는 함수는 DeleteFile과 Erase 두 가지가 있다.




DeleteFile(파일이름);
Erase(파일핸들)

DeleteFile은 인수로 전달된 파일을 삭제한 후 제대로 파일을 삭제했으면 True를 리턴하며, 파일이 없거나 읽기 전용의 파일인 경우, 또는 그 외의 이유로 파일을 지울 수 없으면 False를 리턴한다. 삭제의 대상이 되는 파일 이름은 DeleteFile('c:₩autoexec.bat') 등과 같이 문자열로 전달된다. Erase는 파일의 이름이 아닌 파일 핸들로써 지울 파일을 입력받는다는 점이 다르다.

파일을 다른 위치로 이동시키는 함수는 MoveFile이라는 API 함수가 있으며 이 함수가 아니더라도 파일 복사와 파일 삭제를 연이어 호출하면 파일을 이동시키는 것과 동일한 효과를 볼 수 있다.

# 다. 이름 변경

이름을 변경하는 함수는 RenameFile이다.

**기본 형식**

RenameFIle(원래이름, 바꿀이름);

DeleteFile과 마찬가지로 제대로 이름을 바꾸었으면 True를 리턴하고 이름을 바꾸는 데 실패했으면 False를 리턴한다. 이 외에도 파일의 속성을 읽거나 바꾸는 다음 함수들이 있다. 자세한 사용 방법은 도움말을 직접 참고하기 바란다.

**기본 형식**

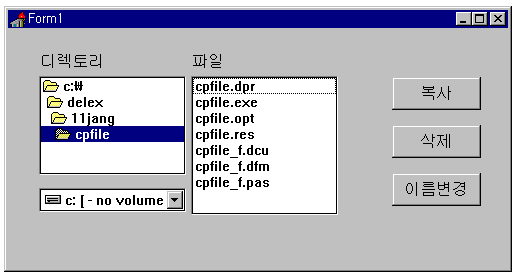
FileGetAttr(파일이름);
FileSetAttr(파일이름, 속성);

파일을 실행시킬 때는 WinExec 함수나 ShellExecute 함수를 사용한다. WinExec는 이미 앞에서 사용해 본 적이 있으므로 설명을 생략하기로 한다.

## 라. 파일 관리 예제

11jang
cpfile

그럼 이제 파일을 제어하는 실제 예제를 작성해 보자. 제대로 만들려면 디렉토리 트리도 두 개 정도는 있어야 하고 Drag & Drop도 지원해야겠지만 어디까지나 예제이므로 여기서는 파일 관리 방법만 보여주기로 한다. 작성 절차는 생략하고 결과만을 보인다. 폼의 모양은 다음과 같다.




드라이브 콤보 박스, 디렉토리 리스트 박스, 파일 리스트 박스를 사용하여 관리의 대상이 되는 파일을 선택한 후 우측의 버튼을 눌러 파일을 관리한다. 복사를 누르면 목적 파일의 이름을 물어보는 대화상자를 출력하도록 했다. 이름 변

경을 누르면 어떤 이름으로 바꿀 것인가를 물어온다.


삭제 버튼을 누르면 한번의 확인 과정을 거친 후 삭제하도록 하였다. 세 버튼의 OnClick 이벤트 코드는 다음과 같다. 복사 버튼에서 MyCopyFile 함수를 호출하므로, 앞에서 보인 MyCopyFile 함수를 미리 입력시켜 놓아야 한다. 배포 CD의 전체 소스를 참고하도록 하자.

```
procedure TForm1.BtnCopyClick(Sender: TObject);
var
  DestFile:String;
begin
  if FileListBox1.FileName='' then exit;
  DestFile:=InputBox('파일 복사',
    '어떤 이름으로 복사하시겠습니까?','temp.bin');
  MyCopyFile(PChar(FileListBox1.FileName),PChar(DestFile));
  FileListBox1.Update;
end;

procedure TForm1.BtnDeleteClick(Sender: TObject);
begin
  if FileListBox1.FileName='' then exit;
  if MessageDlg('삭제하시겠습니까?', mtConfirmation,
    [mbYes, mbNo],0)=mrYes then
  begin
    DeleteFile(PChar(FileListBox1.FileName));
    FileListBox1.Update;
  end;
end;

procedure TForm1.BtnRenameClick(Sender: TObject);
begin
  if FileListBox1.FileName='' then exit;
  RenameFile(FileListBox1.FileName,
     InputBox('이름 변경','바꿀 이름을 입력하십시요','temp.bin'));
  FileListBox1.Update;
end;
```

파일이란 존재는 워낙 믿지 못할 존재이기 때문에 파일이 없을 경우, 디스크에 이상이 있을 경우, 파일 이름이 중복되는 경우 등등 아주 섬세한 에러 처리를 해 주어야 한다. 각종 에러에 대해 일일이 에러 처리를 해 주자면 배보다 배꼽이 더 커지므로 이 예제에서는 파일이 없는 경우에 대해서만 에러 처리를 하였다. 파일을 복사, 삭제, 이름 변경한 후에는 파일 리스트 박스의 Update 메소드를 호출하여 파일 목록에 변경 사항을 반영시켜 주어야 한다. 파일 복사 함수를 Win32 API 함수로 바꾸려면 BtnCopyClick의 코드를 다음과 같이 바꾸어 주면 된다.

```
CopyFile(PChar(FileListBox1.FileName),PChar(DestFile), False);
```

이 외에도 파일을 관리하는 함수에는 다음과 같은 종류가 있다.

**표**
여러 가지 파일 관리 함수

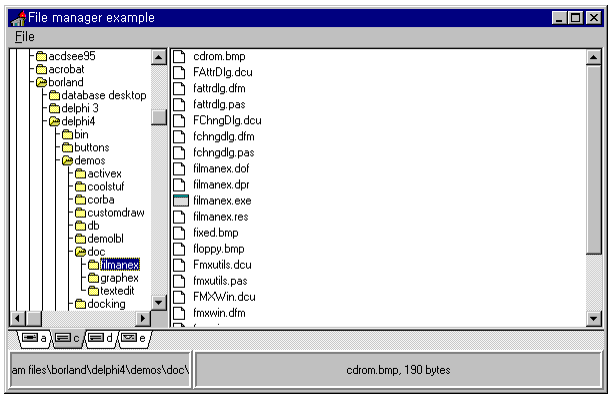
| 함수 | 동작 |
|---|---|
| FileGetAttr | 파일의 속성을 조사한다. |
| FileSetAttr | 파일의 속성을 설정한다. |
| FileGetDate | 파일의 날짜를 조사한다. |
| FileSetDate | 파일의 날짜를 설정한다. |
| FileExist | 파일의 존재 여부를 조사한다. |
| ExtraciFileExt | 파일 경로로부터 확장자만 조사한다. |
| ExtractFileName | 파일 경로로부터 이름만 조사한다. |
| ExtractFilePath | 파일 경로만 조사한다. |
| ChangeFileExt | 확장자를 변경한다. |

자주 사용되는 함수는 아니므로 본문에서는 설명하지 않는다. 필요한 사람은 부록의 레퍼런스를 참조하기 바란다.

# 마. 파일 관리자 예제

DEMOS₩DOC₩FILMANEX 디렉토리를 보면 델파이와 함께 제공되는, 파일 관리자와 비슷한 예제가 있다.

그림
델파이가 제공하는 파일 관리자 예제




이 예제는 예쁜 비트맵이 그려진 탭셋, Drag & Drop 지원, 파일 속성 제어, 파일 실행까지 할 수 있는 실용적으로도 꽤 쓸만한 프로그램이다. 소스도 별로 어렵지 않으므로 직접 분석해 보기 바란다. 여기서 이 큰 예제를 직접 다 분석할 수는 없고 이런 것도 있다는 것만 알려주기로 한다.

끝으로 파일 제어의 가장 고난도에 해당하는 디렉토리 복사 루틴을 간단히 소개만 하기로 한다. 디렉토리 복사는 여러 개의 파일을 일일이 찾아가며 복사해야 하는데다 디렉토리 안에 또 디렉토리가 있을 수 있는 트리 구조로 인해 굉장히 복잡하다. 게다가 함수 호출법 중에 제일 어려운 재귀호출(Recursive Call)까지 사용하기 때문에 문법에 웬만큼 자신이 없으면 이런 루틴을 작성하기가 쉽지 않을 것이다.

재귀 호출이란 함수 내에서 자기 자신을 다시 한 번 더 호출하는 방법이다. 자기가 자기를 호출하면서도 무한 루프에 빠지지 않는 묘한 구조이며 파스칼, C에서만 가능하며 어셈블러에서는 구현하기가 어려우며(전혀 불가능하지는 않다) 전통적인 베이직에서는 꿈도 못꾸는 루틴이다. 자세한 설명을 일일이 다 할 수는 없고 대신 주석을 좀 상세하기 달아 두었다. 이 함수가 제대로 동작하려면 앞에서 소개한 MyCopyFile 함수가 이 함수 이전에 있어야 한다.

```
{ 디렉토리 단위로 복사하는 재귀 호출 함수. 원본 디렉토리와 목적 디렉토리
를 인수로 받아 원본 디렉토리내의 모든 파일과 서브 디렉토리를 목적 디렉
토리로 복사한다. 도스의 XCOPY 명령과 비슷하다. }
procedure copydirectory(srcdir:string;destdir:string);
var
 pst:array [0..128] of char;  { 널 종료 문자열로 바꾸기 위한 버퍼}
 pst2:array [0..128] of char;
 st:string[128];              { 경로 조립을 위한 임시 문자열 }
 st2:string[128];
```

```
 srchmask:TsearchRec;        { 도스의 ffblk 에 해당하는 레코드 }
 result:integer;             { findfirst 의 결과 }
begin
st:=srcdir+'₩*.*';           { 원본의 모든 파일을 복사 }
findfirst(st,faanyfile,srchmask);  { 첫 파일 찾음 }
result:=0;                   { findfirst 에서 에러 없었다는 뜻 }
while result=0 do begin      { 에러가 없는 동안 반복 }
{ 아래 : 현재 디렉토리나 모 디렉토리일 경우는 복사 대상에서 제외한다.
  즉 디렉토리 중에 .과 ..을 복사하지 않도록 한다.
  그렇지 않으면 올바른 재귀호출을 할 수 없다. 이 경우 다음 파일을
  찾아 루프를 계속 반복하도록 한다. }
  if (srchmask.name='.') or (srchmask.name='..') then
    begin
    result:=findnext(srchmask);   {다음 파일 찾음}
    continue;
    end;
{ 아래 : 복사 대상이 서브 디렉토리일 경우 해당 서브 디렉토리를 원본 디렉
토리로 하여 다시 자기 자신을 부른다. 서브 디렉토리의 복사를 마친 후에는
다음 파일을 찾아 루프를 계속 반복하도록 한다. }
  if (srchmask.attr and fadirectory)=fadirectory then
    begin
    st:=srcdir+'₩'+srchmask.name;    { 새로운 원본 디렉토리 }
    st2:=destdir+'₩'+srchmask.name;  { 새로운 목적 디렉토리 }
    mkdir(st2);                  { 목적 디렉토리는 직접 만든다. }
    copydirectory(st,st2);       { 재귀 호출 . 자기 자신을 부른다.}
    result:=findnext(srchmask);      { 다음 파일 찾음 }
    continue;
    end;
{ 찾은 파일이름을 완전 경로로 조립한다. 그래야 파일 복사 함수를
부를 수 있다. }
  st:=srcdir+'₩'+srchmask.name;
  st2:=destdir+'₩'+srchmask.name;
  strpcopy(pst,st);      { 널 종료 문자로 바꾼다. }
  strpcopy(pst2,st2);
  MyCopyFile(pst,pst2);  { 파일 복사 함수를  호출한다. }
  result:=findnext(srchmask);
  end;
end;
```

# 바. 디스크 관리

11jang
FreeDrv

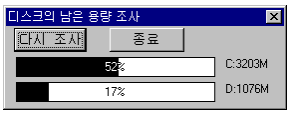
도스(윈도우즈도 물론 마찬가지)에서 데이터를 저장하는 단위는 파일이지만 파일은 디스크와 디렉토리에 저장된다. 따라서 디스크나 디렉토리를 관리하는

함수도 일종의 파일 관리 함수라고 볼 수 있다. 디스크와 디렉토리에 관련된 함수에는 다음과 같은 종류가 있다.

**표**
디스크 관리 함수

| 함수 | 동작 |
|---|---|
| DiskFree | 디스크의 남은 용량을 조사한다. |
| DiskSize | 디스크의 크기를 조사한다. |
| MkDir | 디렉토리를 만든다. |
| RmDir | 디렉토리를 지운다. |
| ChDir | 현재 디렉토리를 변경한다. |
| GetDir | 현재 디렉토리를 조사한다. |
| DirectoryExist | 디렉토리 존재 여부를 확인한다. |

사용법이 무척 쉬우므로 상세한 설명은 생략하므로 이 함수들에 대한 세부적인 사항은 부록의 레퍼런스나 도움말을 참조하기 바란다. 대신 이 함수들을 사용하는 예제만 하나 만들어 보았다. 이 예제는 시스템의 모든 드라이브를 조사하여 사용한 용량과 남은 용량을 보여준다. 동적 객체 생성 기법을 사용하고 있는데 참고할만할 것이다. 실행 중의 모습은 다음과 같다.




필자는 6G 하드 두 개만 사용하고 있기 때문에 게이지가 둘 밖에 나타나지 않았지만 드라이브가 아무리 많아도 그 개수만큼 게이지를 보여준다. 전체 소스는 다음과 같다. 주석이 비교적 상세하게 작성되어 있으므로 분석은 하지 않기로 한다. 주석을 참고하여 직접 연구해 보기 바란다.

```
unit freedrvf;

interface

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

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  DrvLabel:array [1..26] of TLabel;
  DrvGauge:array [1..26] of TGauge;
  TotalDrv:Integer;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
  i:Integer;
  str:string;
  str2:array [0..128] of Char;
begin
  TotalDrv:=0;
  {현재 시스템에 장착된 드라이브의 개수를 구함
  단 플로피나 CD-ROM 은 제외한다.}
  for i:=0 to 25 do
    begin
    str2:='c:₩';
    str2[0]:=Chr(i+Ord('c'));
    if GetDriveType(str2)=DRIVE_FIXED then
      Inc(TotalDrv);
    end;
  {조사된 드라이브의 개수만큼 게이지와 레이블을 생성하여
  폼에 배치한다.}
  for i:=1 to TotalDrv do
    begin
    DrvLabel[i]:=TLabel.Create(Self);
    DrvLabel[i].Parent:=Form1;
    DrvLabel[i].Top:=i*25+5;
```

```
    DrvLabel[i].Left:=220;

    DrvGauge[i]:=TGauge.Create(Self);
    DrvGauge[i].Parent:=Form1;
    DrvGauge[i].Left:=10;
    DrvGauge[i].Top:=i*25+5;
    DrvGauge[i].Width:=200;
    DrvGauge[i].Height:=20;
    end;
  {폼의 높이는 드라이브의 개수에 따라 달라진다.}
  Form1.Height:=TotalDrv*25+55;
  {남은 용량을 조사한다.}
  Button1Click(Sender);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  TotalSpace,FreeSpace:Int64;
  i:Integer;
  str:array [0..128] of Char;
begin
  for i:=0 to 26-2 do
    begin
    str:='c:₩';
    str[0]:=Chr(i+Ord('c'));
    if GetDriveType(str)=DRIVE_FIXED then
      begin
      {총용량을 메가 바이트 단위로 구한다.}
      TotalSpace:=DiskSize(i+3) div 1048576;
      {사용 가능한 용량을 메가 바이트 단위로 구한다.}
      FreeSpace:=DiskFree(i+3) div 1048576;
      {게이지에 남은 용량을 백분율로 표시한다.}
      if TotalSpace◇0 then
         DrvGauge[i+1].Progress:=FreeSpace*100 div TotalSpace;
      {레이블에 남은 용량을 출력한다.}
      DrvLabel[i+1].Caption:=Chr(i+Ord('C'))+
        ':'+IntToStr(FreeSpace)+'M';
      end;
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  close;
end;
```

end.

# 사. 간단한 에디터 4

11jang
editor4

4장, 5장, 7장에서 새로운 기법들을 배울 때마다 실습용으로 에디터를 작성해 보았는데 Editor3까지는 에디터에 가장 기본적으로 있어야 할 파일 입출력 기능이 빠져서 실용적으로 사용할 수가 없었다. 이제 파일 입출력까지 공부를 했으므로 여기서 실제 사용이 가능한 에디터를 만들어 보기로 한다. 여기서 만들 Editor4는 다음 기능들이 추가되었다.

① 파일 열기, 저장하기, 새로 만들기가 가능하다. 공통 대화상자를 사용하여 비교적 쉽게 구현하였다. 액션을 사용하여 툴바와 메뉴 항목 중 편한대로 사용할 수 있다.

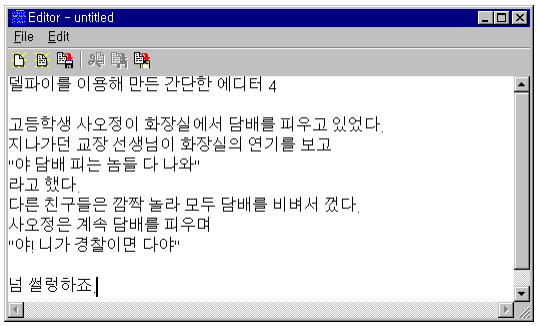
② 폰트 공통 대화상자를 사용하여 메모의 글꼴을 사용자가 마음대로 변경할 수 있도록 하였다.

③ 프로그램에 아이콘을 지정하여 다른 프로그램과 구분이 되도록 하였으며 완성도를 높였다.

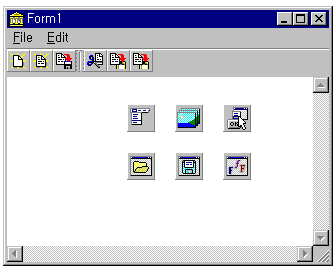
④ 저장되지 않은 문서를 잃지 않도록 문서를 닫기 전, 프로그램을 종료하기 전에 확인하도록 하였다.

⑤ 편집할 파일명을 인수로 받아들이도록 하여 실행과 동시에 파일을 편집할 수 있다.

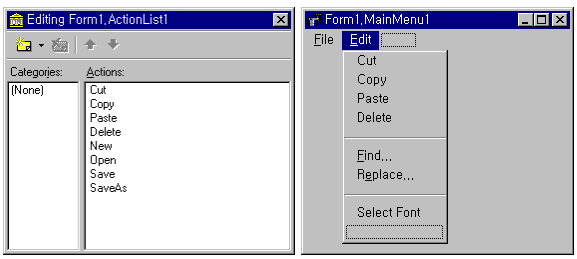
프로그램 실행중의 모습은 다음과 같다. 배포 CD의 프로그램을 실행시켜 보면 에디터에 필요한 대부분의 기능이 다 포함되어 있음을 알 수 있을 것이다.



Editor4의 상세한 제작 절차는 밝히지 않기로 한다. 왜냐하면 새로운 내용을 다룬 것이 아니라 앞에서 이미 학습한 내용을 실습한 것이기 때문이다. 11장까지 읽어온 여러분에게 컴포넌트 배치가 어쩌고, 속성이 어쩌고 일일이 설명한다면 중학생에게 "1 더하기 1은 2지요", "우리나라 국기는 태극기입니다."를 열심히 열강하는 꼴 밖에 되지 않을 것 같다. 대신 프로그램 전체의 구조에 대해서만 살펴 보도록 하자. 일단 디자인중의 폼을 보면 여러 가지 컴포넌트들이 추가되었다.




세 개의 공통 대화상자 컴포넌트가 추가되었는데 각각 파일 열기, 저장하기, 글꼴 선택 공통 대화상자이며 각 대화상자는 툴바/메뉴 항목 등에서 사용한다. 액션 리스트에는 4개의 액션이 추가되었으며 Edit 메뉴의 아래쪽에 Select Font라는 항목이 추가되었다.




Editor4의 전체 소스는 다음과 같다. 상세한 분석은 하지 않으므로 소스를 보고 직접 분석해 보기 바란다. 이때까지 만든 에디터와는 달리 앨거리듬이 식은 죽 먹듯이 쉽게 풀리지만은 않을 것이다.

```
unit Editor4_f;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Menus, Clipbrd, ImgList, ComCtrls, ToolWin, ActnList;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    N2: TMenuItem;
    SaveAs1: TMenuItem;
    Save1: TMenuItem;
    Open1: TMenuItem;
    New1: TMenuItem;
    Edit1: TMenuItem;
    Replace1: TMenuItem;
    Find1: TMenuItem;
    N3: TMenuItem;
    Delete1: TMenuItem;
    Paste1: TMenuItem;
    Copy1: TMenuItem;
    Cut1: TMenuItem;
    ToolBar1: TToolBar;
    ToolButton1: TToolButton;
    ToolButton2: TToolButton;
    ToolButton3: TToolButton;
    ToolButton4: TToolButton;
    ToolButton5: TToolButton;
    ToolButton6: TToolButton;
    ToolButton7: TToolButton;
    ImageList1: TImageList;
    N1: TMenuItem;
    ShowToolBar1: TMenuItem;
    ActionList1: TActionList;
    Cut: TAction;
    Copy: TAction;
    Paste: TAction;
    Delete: TAction;
    New: TAction;
    Open: TAction;
    Save: TAction;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    SaveAs: TAction;
    N4: TMenuItem;
    SelectFont1: TMenuItem;
```

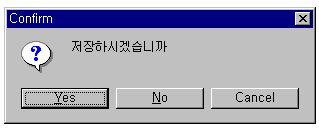
```
    FontDialog1: TFontDialog;
    procedure ShowToolBar1Click(Sender: TObject);
    procedure CutExecute(Sender: TObject);
    procedure CopyExecute(Sender: TObject);
    procedure PasteExecute(Sender: TObject);
    procedure DeleteExecute(Sender: TObject);
    procedure UpdateUI(Sender: TObject);
    procedure NewExecute(Sender: TObject);
    procedure OpenExecute(Sender: TObject);
    procedure SaveExecute(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SelectFont1Click(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  fname:string;   {현재 편집중인 파일의 이름}

implementation

{$R *.DFM}

procedure TForm1.ShowToolBar1Click(Sender: TObject);
begin
if ShowToolbar1.Checked=False then
  begin
  ShowToolbar1.Checked:=True;
  ToolBar1.Show;
  end
else
  begin
  ShowToolbar1.Checked:=False;
  ToolBar1.Hide;
  end;
end;

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;

{툴바, 메뉴 항목의 상태를 변경한다}
procedure TForm1.UpdateUI(Sender: TObject);
begin
  Paste.Enabled:=Clipboard.HasFormat(CF_TEXT);
  {선택 영역이 있어야 Cut, Copy, Delete 를 사용할 수 있다}
  if Memo1.SelLength=0 then
    begin
    Cut.Enabled:=False;
    Copy.Enabled:=False;
    Delete.Enabled:=False;
    end
  else
    begin
    Cut.Enabled:=True;
    Copy.Enabled:=True;
    Delete.Enabled:=True;
  end;

  {메모 내용이 변경되어야 저장 명령을 쓸 수 있다}
  if Memo1.Modified=True then
    begin
    Save.Enabled:=True;
    SaveAs.Enabled:=True;
    end
  else
    begin
    Save.Enabled:=False;
    SaveAs.Enabled:=False;
    end;
end;
```

```
{새 파일}
procedure TForm1.NewExecute(Sender: TObject);
var
  Res:Word;
begin
  {저장 여부를 물어보고 미보관 문서를 저장한다.}
  if Memo1.Modified then
  begin
    Res:=MessageDlg('저장하시겠습니까',mtConfirmation,mbYesNoCancel,0);
    if Res=mrCancel then Exit;
    if Res=mrYes then
      if SaveDialog1.Execute then
        Memo1.Lines.SaveToFile(SaveDialog1.FileName);
  end;
  Memo1.Lines.Clear;
  fname:='untitled';
  Form1.Caption:='Editor - untitled';
end;

{파일 열기}
procedure TForm1.OpenExecute(Sender: TObject);
var
  Res:Word;
begin
  if Memo1.Modified then
  begin
    Res:=MessageDlg('저장하시겠습니까',mtConfirmation,mbYesNoCancel,0);
    if Res=mrCancel then Exit;
    if Res=mrYes then
      if SaveDialog1.Execute then
        Memo1.Lines.SaveToFile(SaveDialog1.FileName);
  end;
  if OpenDialog1.Execute then
    begin
    fname:=OpenDialog1.FileName;
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
    Form1.Caption:='Editor - '+fname;
    end;
end;

{파일 저장하기}
procedure TForm1.SaveExecute(Sender: TObject);
var
  SResult:Boolean;
begin
  SResult:=True;
```

```
  if Memo1.Modified=False then Exit;
  {저장되지 않았거나 Save As 가 선택되었을 경우 파일 이름 재입력}
  if (fname='untitled') or (Sender=SaveAs) then
    SResult:=SaveDialog1.Execute;
  if SResult=True then
    Begin
    fname:=SaveDialog1.FileName;
    Memo1.Lines.SaveToFile(fname);
    Memo1.Modified:=False;
    Form1.Caption:='Editor - '+fname;
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  fname:='untitled';
  Form1.Caption:='Editor - '+fname;
  if ParamCount > 0 then
    Memo1.Lines.LoadFromFile(ParamStr(1));
end;

{메모의 폰트를 변경한다}
procedure TForm1.SelectFont1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
    Memo1.Font:=FontDialog1.Font;
end;

{폼을 닫기 전에 미 보관된 문서가 있으면 저장하도록 한다.}
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
  Res:Word;
begin
  CanClose:=True;
  if Memo1.Modified then
  begin
    Res:=MessageDlg('저장하시겠습니까',mtConfirmation,mbYesNoCancel,0);
    if Res=mrCancel then CanClose:=False;
    if Res=mrYes then
      if SaveDialog1.Execute then
        Memo1.Lines.SaveToFile(SaveDialog1.FileName);
  end;
end;

end.
```

대부분의 코드는 앞에서 배운 것들인데 두 가지 새로운 것이 있어 이에 대해서만 설명을 하도록 한다. 첫 번째는 미보관 문서에 대한 확인 기능이다. 프로그램을 실행한 후 새 문서를 입력하고 Alt+F4로 프로그램을 종료하려고 하면 미저장 문서에 대한 처리를 물어올 것이다.




이 질문에 Yes라고 하면 저장을 하고 No라고 하면 저장을 하지 않고 프로그램을 종료한다. 특이한 것은 Cancel이라고 대답하면 프로그램 종료가 취소된다는 점인데 이런 처리는 폼의 OnCloseQuery 이벤트를 사용하면 된다.

```
procedure(Sender: TObject; var CanClose: Boolean)
```

이 이벤트는 폼이 닫히기 전에 발생하며 참조 인수로 CanClose라는 진위형 참조 인수가 전달되는데 이 인수에 False를 대입하면 프로그램 종료가 취소된다. 이 이벤트의 핸들러에서 프로그램을 종료할 수 있는 상황인지 점검해 본 후 사용자에게 미보관 문서가 있음을 알려주는 것이 좋다.

두 번째는 명령행 인수를 받아들여 프로그램 실행과 동시에 편집할 문서를 읽어올 수 있는 기능이다. notepad readme.txt를 실행하면 메모장이 실행됨과 동시에 readme.txt가 로드되는 바로 그런 기능이다. 이 기능은 FormCreate의 다음 두 줄에 의해 실행된다.

```
if ParamCount > 0 then
  Memo1.Lines.LoadFromFile(ParamStr(1));
```

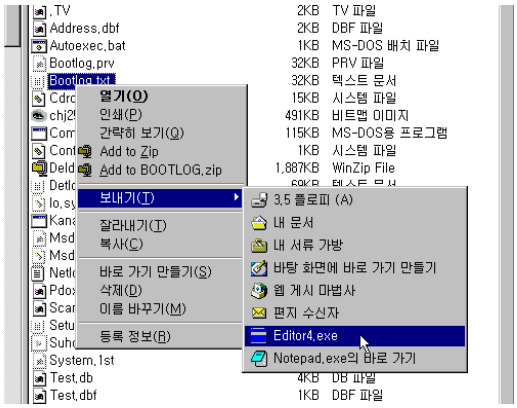
ParamCount 함수는 명령행으로 전달된 인수의 개수를 조사해 주며 ParamStr은 전달된 인수를 조사해 주는 함수이다. ParamStr(0)는 프로그램의 완전 경로명, ParamStr(1)은 첫 번째 인수, ParamStr(2)는 두 번째 인수이다. 예를 들어 Notepad Readme.txt라는 명령의 경우 ParamCount는 1이 되며 ParamStr(0)는 C:₩Windows₩Notepad.exe가 되며 ParamStr(1)은 Readme.txt가 된다. 이 정보들을 참고하여 폼이 생성될 때 인수가 있으면 첫 번째 인수로 전달된 파일을 읽어오도록 하였다.

이 기능이 추가되면 editor4 Humor.txt 등의 명령으로 실행과 동시에 텍스

트 파일을 읽을 수가 있으며 Windows₩Send To 디렉토리에 이 프로그램(또는 단축키)을 복사해 놓으면 탐색기의 보내기 메뉴에도 나타나며 탐색기에서 파일을 선택해 바로 편집할 수 있다.

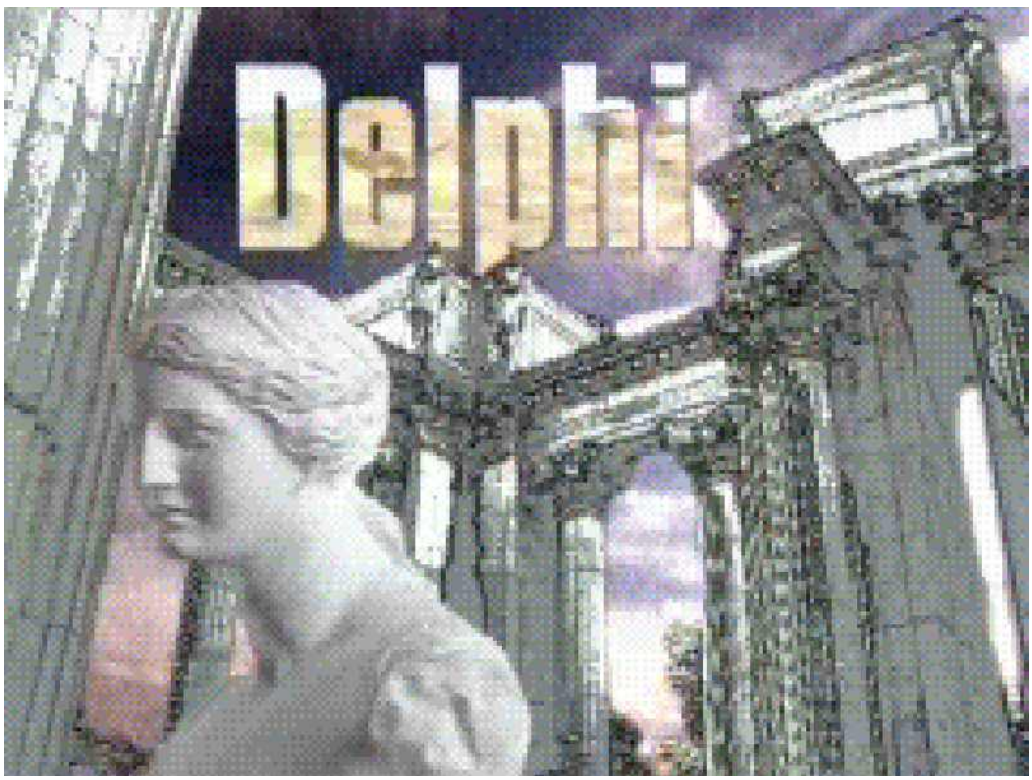
**그림**

탐색기의 보내기 명령



# 고급 프로그래밍

# 제 12 장



# 12-1 드래그 앤 드롭

## 가. 드래그 모드

드래그(Drag ; 끌기)란 마우스 버튼을 누른 채로 마우스를 움직이는 것을 말하며 드롭(Drop ; 떨어뜨리기)이란 드래그를 끝내고 버튼을 놓는 것을 말한다. 이 둘을 묶어 드래그 앤 드롭(Drag & Drop)이라고 하며 윈도우즈에서 윈도우의 이동, 아이콘의 이동 등에 사용되며 탐색기에서 파일을 복사할 때 사용하기도 한다. 델파이에서 드래그 앤 드롭은 이벤트로서 구현되며 버튼, 레이블, 에디트 등 대부분의 컴포넌트가 드래그 앤 드롭을 지원한다. 드래그되는 컨트롤의 행동은 드래그 모드에 의해 통제된다. 두 가지 종류가 있으며 각 컨트롤의 DragMode 속성으로 지정한다.

- dmAutomatic : 사용자가 언제든지 드래그할 수 있는 상태의 드래그 모드이다. 마우스 버튼을 눌러 끌기만 하면 드래그가 시작된다. 자동으로 드래그가 시작되기 때문에 편리하기는 하지만 몇 가지 문제점이 있다.
- dmManual : 프로그램에서 BeginDrag 메소드로 드래그를 허락하기 전에는 드래그를 할 수 없는 상태의 드래그 모드이다. BeginDrag(Immediate) 호출로 드래그를 시작한다. Immediate 인수는 True, False 중 한 값을 가지며 True 일 경우 BeginDrag 호출 즉시 드래그를 시작하고 False 일 경우 마우스 버튼을 누른 상태에서 일정 거리를 움직여야 드래그를 시작한다. BeginDrag(False)로 드래그를 시작하면 마우스를 누른 즉시 드래그가 시작되는 것이 아니므로 클릭할 기회를 남겨 두게 된다.

두 가지 드래그 모드 중에서 자동으로 설정하는 것이 편할 것 같지만 디폴트는 수동 모드이다. 수동 모드로 할 수 있는 일이 더 많으며 자동은 몇 가지 말썽의 소지가 있기 때문이다.

# 나. 드래그 이벤트

드래그 앤 드롭에 사용되는 이벤트는 총 세 가지가 있으며 다음 두 이벤트가 드래그 앤 드롭의 핵심을 이룬다.

■ **DragOver(Sender, Source:TObject;X,Y:Integer;**
**state:TDragstate;var Accept:Boolean);**

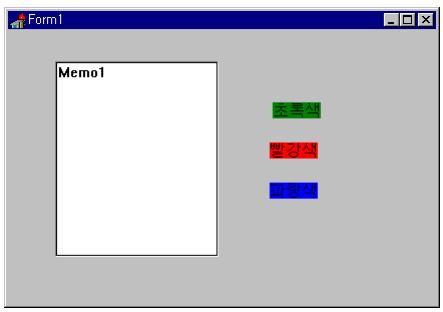
다른 오브젝트가 드래그되고 있을 때 커서 아래의 오브젝트에 발생하는 이벤트이다. 만약 에디트 위로 버튼이 드래그되고 있다면 에디트 컴포넌트가 이 이벤트를 받아들이는 Sender이며 버튼이 드래그되고 있는 Source이다. 이 이벤트에서 처리해야 할 가장 중요한 일은 드래그되고 있는 컨트롤이 드롭이 가능한지를 Accept 참조 인수에 대입해 주는 일이며 Source, X,Y 등의 인수로 전달된 정보를 사용하여 드롭 허가 여부를 판단한다. Accept에 True를 대입해 주면 사용자가 마우스 버튼을 놓을 경우 DragDrop 이벤트가 발생하며 False를 대입해 주면 드래그는 무시된다.

■ **DragDrop(Sender, Source:TObject;X,Y:Integer);**

드래그한 컨트롤을 드롭했을 때 마우스 커서 아래의 컨트롤이 받는 이벤트이다. 이 이벤트가 발생하기 위해서는 드롭을 받는 컨트롤이 DragOver 이벤트에서 Accept 인수에 True를 대입해 주어 드롭을 허가해야 한다. Sender는 드롭을 받는 컨트롤이며 Source는 드래그된 컨트롤이며 X,Y는 드롭이 발생한 화면 좌표이다.

12jang
drag1

드래그 앤 드롭을 사용하는 간단한 예제를 제작해 보자. 메모 컴포넌트 하나와 레이블 세 개를 폼에 배치한다.




레이블의 색상을 각각 다르게 설정하고 레이블을 드래그하여 메모에 드롭하

면 드래그된 레이블의 색상으로 메모의 색상을 변경하도록 할 것이다. 레이블의 속성을 다음과 같이 설정한다.

| 속성 | 위쪽 레이블 | 가운데 레이블 | 아래쪽 레이블 |
|---|---|---|---|
| Name | LabGreen | LabRed | LabBlue |
| Caption | 초록색 | 빨간색 | 파란색 |
| Color | clGreen | clRed | clBlue |
| DragMode | dmAutomatic | dmAutomatic | dmAutomatic |

레이블의 드래그 모드가 모두 자동(dmAutomatic)으로 설정되어 있기 때문에 드래그의 시작에 대해서는 코드를 작성하지 않아도 된다. 작성해야 할 코드는 메모의 DragOver 이벤트와 DragDrop 이벤트이다.

```
procedure TForm1.Memo1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  Memo1.Color:=(Source as TLabel).Color;
end;

procedure TForm1.Memo1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept:=True;
end;
```

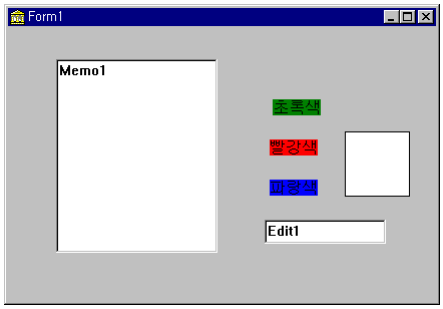
DragOver에서는 Accept에 무조건 True를 대입하여 드래그되는 모든 컨트롤이 드롭되도록 허가하였다. DragDrop에서는 드롭된 컨트롤의 Color 속성을 사용하여 메모 컴포넌트의 색상을 바꾼다. 코드를 작성한 후 예제를 실행시키면 아주 정상적으로 작동한다. 레이블을 드래그하여 메모에 떨어뜨리면 메모의 색상이 레이블의 색상으로 변경될 것이다.

드래그를 시작하면 마우스 커서가  요렇게 변해서 드래그가 시작되었음을 표시하고 드롭이 가능한 지역인 메모 안으로 드래그하면 마우스 커서가  요렇게 변해 드롭 허가를 알린다. 만약 메모에서 Accept에 True를 대입해 주지 않으면 마우스 커서는 계속  요런 모양을 유지하며 DragDrop 이벤트는 발생하지 않는다.

# 다. 수동 모드

12jang
drag2

위의 예제는 아주 간단한 편이지만 사실 드래그 앤 드롭은 이렇게 간단하기만 한 것이 아니다. 위의 예제를 조금만 변형하여 보면 복잡한 문제들이 발생한다. 폼에 에디트 하나와 셰이프 하나를 더 배치하고 이 컨트롤들을 드래그시키기로 하자.




우선 에디트를 드래그하여 메모에 드롭할 경우 에디트에 입력된 문자열을 메모의 Text 속성에 대입하게 해 보자. 에디트의 DragMode를 dmAutomatic으로 설정하고 메모의 DragDrop 이벤트에서 에디트의 Text를 받아들일 수 있도록 다음과 같이 코드를 수정하라

```
procedure TForm1.Memo1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source is TLabel then
    Memo1.Color:=(Source as TLabel).Color
  else
    Memo1.Text:=(Source as TEdit).Text;
end;
```

드래그가 시작된 컨트롤(Source)이 레이블이면 메모의 Color 속성을 변경하도록 하고 에디트이면 Text 속성을 변경하도록 하였다.

여기에 사용된 is 연산자는 실행중에 오브젝트의 타입을 체크한다. obj is cla의 형식으로 사용하며 obj 오브젝트가 cla 클래스나 cla로부터 파생된 클래스의 인스턴스이면 True를 리턴하고 그렇지 않으면 False를 리턴한다. Source is TLabel이라는 조건은 “드래그가 시작된 Source 컨트롤이 레이블형이면”이라는 조건이다. as 연산자는 실행중에 오브젝트의 타입을 변경해 주는 타입 캐스팅(type casting) 연산자이다. obj as cla 형식으로 사용하며 obj 오브젝트를 cla 클래스형으로 변경해 준다. “Source as TLabel”은 드래그가 시작된 컨트롤인

Source가 레이블임을 밝힌다. 만약 타입 캐스팅을 하지 않고 Source.Color라고 사용하면  Source는 TObject 타입으로 간주되며 Color라는 속성을 사용할 수 없게 된다.

여기까지 작성하고 실행시키면 과연 아무런 문제가 없는 것 같아 보인다. 에디트를 드래그하여 메모에 떨어뜨리면 메모의 Text 속성이 Edit1으로 잘 변경되며 레이블을 드래그하면 색상이 제대로 변한다. 그럼 과연 제대로 된 예제인지 검정해 보기 위해 에디트에 Apple이라는 문자열을 입력한 후 드래그하도록 해 보자. 에디트에 문자열을 입력하기 위해 마우스로 에디트를 선택하자마자 드래그가 시작되기 때문에 에디트에 문자열을 입력할 수 없다 (Tab키를 사용하면 가능하기는 하다). 왜 그런가 하면 에디트의 드래그 모드가 마우스를 누르기만 하면 드래그가 시작되는 자동(dmAutomatic)으로 설정되어 있기 때문이다. 이 문제를 해결하려면 에디트의 DragMode 속성을 다시 수동(dmManual)으로 바꾸어 주어야 한다. 그리고 에디트의 MouseDown 이벤트에서 드래그를 시작할 수 있도록 해 준다.

```
procedure TForm1.Edit1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Edit1.BeginDrag(False);
end;
```

BeginDrag 메소드의 인수가 False인 것에 유의하자. 이 인수가 False이기 때문에 마우스를 눌러도 곧바로 드래그가 시작되지 않고 마우스를 일정 거리 이동시켜야 드래그가 시작된다. 드래그 모드에 왜 수동과 자동 두 가지가 필요한가를 이해할 수 있을 것이다. 자동일 경우는 OnClick 이벤트가 드래그에 의해 가려지지만 수동일 경우는 OnClick 이벤트가 온전하게 발생한다. 레이블의 경우 OnClick 이벤트가 별 쓸모가 없으며 포커스를 가지지도 않기 때문에 자동 드래그를 사용하는 것이 여러 면에서 유리하지만 에디트나 버튼은 수동으로 드래그를 해 주어야 한다.

이번에는 셰이프 컴포넌트의 드래그를 생각해 보자. 이 컴포넌트는 드래그를 하기는 하되 메모 컴포넌트로 드래그되지는 않고 다른 컴포넌트로 드래그하기 위해 만들어졌다고 하자. 예제에는 없지만 예를 들어 이미지 컴포넌트로 드래그할 생각이다. 그래서 셰이프를 드래그하여 메모에 떨어뜨릴 경우는 엉뚱한 동작을 하게 될 것이다. 셰이프에는 Text 속성도 없는데 셰이프의 Text 속성을 Memo1.Text에 대입하려고 하기 때문이다. 이럴 때는 메모의 DragOver 이벤트에서 드롭 허가 여부를 조작해 주어야 한다.

```
procedure TForm1.Memo1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if Source is TShape then
    Accept:=False
  else
    Accept:=True;
end;
```

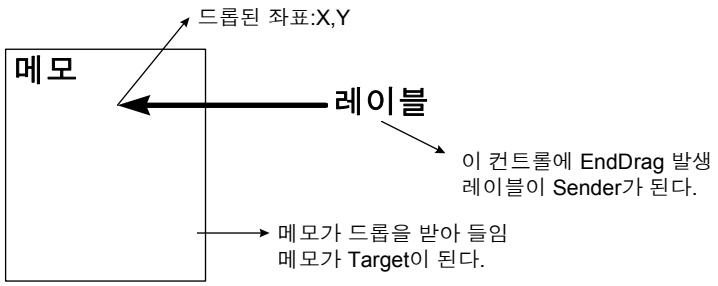
만약 드래그된 컨트롤(Source)이 셰이프(TShape)이면 Accept 인수를 False로 설정하여 드롭되지 못하도록 하고 셰이프 이외의 컨트롤이면 Accept에 True를 설정하여 드롭을 허가해 준다. DragOver 이벤트는 이렇게 엉뚱한 컨트롤이 드롭되지 않도록 방어막을 치는 역할을 한다. 이것이 DragOver 이벤트의 존재 이유이다.

# 라. EndDrag 이벤트

드래그 앤 드롭에 관계된 세 번째 이벤트는 OnEndDrag 이벤트이며 드래그가 끝난 후 드래그를 시작한 컨트롤에 발생한다.

■ **EndDrag(Sender, Target:TObject;X,Y:Integer);**

OnEndDrag 이벤트로는 Sender, Target, X, Y 네 개의 인수가 전달된다. Target 인수는 드롭을 받아들인 컨트롤이며 만약 드롭을 받아들인 컨트롤이 없을 경우, 즉 드롭되지 않았을 경우 Target은 nil이 된다. 예를 들어 레이블이 드래그되어 메모로 드롭되었다면 다음과 같이 이벤트가 발생한다.



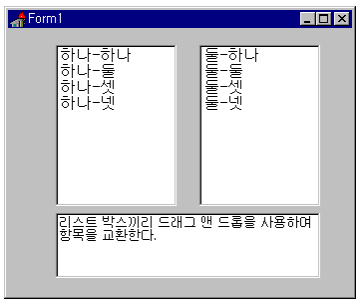

파일 리스트 박스에서 파일을 드래그하여 옮기는 경우를 가정해 보자. 제대로

드롭이 되었다면 드래그된 파일은 파일 리스트 박스에서 지워야 하며 드롭이 되지 않았다면 파일을 지우지 않고 그대로 유지해야 한다. 이런 처리를 해 주는 곳이 OnEndDrag 이벤트이다. 의사 코드를 보이면 다음과 같이 될 것이다.

```
procedure TForm1.Label1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
 if Target = nil then   {드롭이 되지 않았으면}
  파일 이름 그대로 둠
 else                   {제대로 드롭이 되었으면}
  파일 이름 삭제
end;
```

12jang
drag3

그럼 OnEndDrag 이벤트를 사용하는 예제를 작성해 보자. 리스트 박스 두 개를 폼에 배치한 후 각 리스트 박스에 네 개씩 항목을 입력한다.




한 쪽 리스트 박스에서 드래그하여 다른쪽 리스트 박스로 드롭하면 드래그를 시작한 리스트 박스의 항목을 다른쪽 리스트 박스로 옮기기로 한다. 즉 드롭을 받은 리스트 박스에는 드래그된 항목을 추가하고 드래그를 시작한 리스트 박스에서 드래그된 항목은 삭제한다. 사용하는 드래그 모드는 디폴트인 수동 모드이므로 별도로 변경해 주어야 할 속성은 없다. 전체 소스는 다음과 같다.

```
procedure TForm1.ListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
{마우스로 클릭한 부분에 항목이 있으면 드래그를 시작한다.}
IF (Sender as TListBox).ItemAtPos(Point(X,Y),True) <> -1
  then (Sender as TListBox).BeginDrag(False);
end;

{리스트 박스로부터 드래그된 항목이면 드롭을 받아들인다.}
procedure TForm1.ListBox1DragOver(Sender, Source: TObject;
```

```
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
if Source is TListBox then
  Accept:=True;
end;

{드롭을 받은 리스트 박스에 드롭된 항목을 추가한다.}
procedure TForm1.ListBox1DragDrop(Sender, Source: TObject;
  X, Y: Integer);
var
  Src,Dest:TListBox;
begin
Src:=Source as TListBox;
Dest:=Sender as TListBox;
Dest.Items.Add(Src.Items[Src.ItemIndex]);
end;

{드래그를 시작한 리스트 박스에 드롭된 항목을 삭제한다.}
procedure TForm1.ListBox1EndDrag(Sender, Target: TObject;
  X, Y: Integer);
var
  Sen:TListBox;
begin
{드롭을 받은 컨트롤이 리스트 박스일 경우만 삭제한다.}
if Target is TListBox then
  begin
  Sen:=Sender as TListBox;
  Sen.Items.Delete(Sen.ItemIndex);
  end;
end;
```

두 리스트 박스에서 사용하는 코드가 모두 동일하므로 ListBox1에 대한 이벤트 핸들러만 만들고 ListBox2의 이벤트 핸들러는 오브젝트 인스펙터에서 연결해 주기만 하면 된다.

리스트 박스에서 마우스 버튼을 누르면 드래그를 시작하되 그 전에 항목을 제대로 클릭했는지를 점검해 본다. ItemAtPos 메소드는 클릭한 부분에 항목이 있는지 점검해 주며 항목이 없는 빈 영역일 경우는 -1을 리턴해 준다. 항목을 제대로 클릭했으면 BeginDrag를 호출하여 드래그를 시작한다.

DragOver 이벤트  핸들러에서는 드래그를 시작한 컨트롤이 리스트 박스이기만 하면 드롭을 받아들이며 DragDrop 이벤트 핸들러에서는 드롭된 항목을 리스트 박스에 추가시킨다. 마지막으로 EndDrag 이벤트 핸들러는 드래그 앤 드롭

이 완전히 끝난 후의 뒷 정리를 하고 있다. 다른 리스트 박스로 드롭되어진 항목을 Delete 메소드를 사용하여 삭제하되 단 드롭을 받아들인 컨트롤(Target)이 리스트 박스인가 확인한 후에 삭제한다. 이 확인을 생략해 버리면 엉뚱한 곳에 드롭해도 항목이 삭제되어 버릴 것이다.

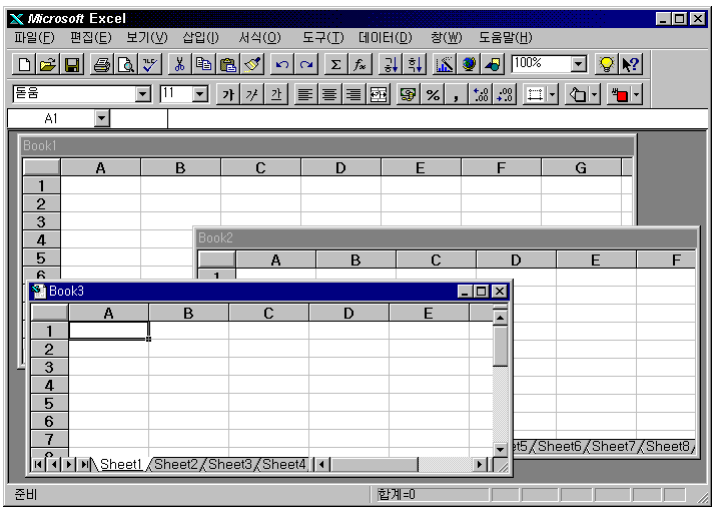
이상 드래그 앤 드롭에 관해 공부해 보았다. 좀 더 완벽하고 규모가 큰 예제를 분석해 보고 싶으면 델파이와 함께 제공하는 파일관리자 예제를 직접 분석해 보기 바란다.

# 12-2 MDI

## 가. MDI의 정의

MDI(Multiple Document Interface)란 하나의 윈도우 안에 여러 개의 다른 윈도우가 동시에 동작하는 프로그램을 말한다. 대표적으로 여러 개의 워크시트를 열 수 있는 마이크로소프트 엑셀을 들 수 있다.

그림
여러 개의 차일드 윈도우를 가지는 MDI 프로그램




워드 프로세서나 스프레드 시트 등 여러 개의 문서를 동시에 열어야 하는 프로그램들이 주로 MDI를 사용한다. 이에 비해 노트패드나 문서 작성기 등과 같이 한번에 하나의 문서만을 편집할 수 있는 프로그램을 SDI(Single Document Interface)라고 하며 이 책에서 지금까지 예제로 만들었던 프로그램들이 모두 SDI이다.

MDI 프로그램은 하나의 프레임 윈도우와 여러 개의 차일드 윈도우로 이루어진다. 프레임 윈도우란 바깥쪽에 있는 프로그램 윈도우이며 차일드 윈도우란 프레임 윈도우 안에서 동작하는 윈도우이다. 위의 엑셀 예에서 바깥쪽의 큰 윈도우가 프레임이며 안쪽의 워크 시트 윈도우들이 차일드 윈도우다. 프레임 폼은 이때까지 늘상 디자인 해 오던 것과 같은 방법으로 디자인하며 차일드 폼은 틀만 만들어 두고 실행중에 생성한다.

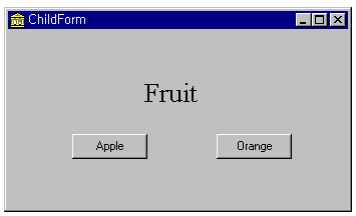
# 나. MDI 예제

12jang
mdi

아주 간단한 MDI 예제를 만들어 보자. MDI는 사실 그리 간단한 프로그램이 아니기 때문에 만드는 과정이 굉장히 복잡하지만 델파이를 사용하면 어렵지 않게 MDI 프로그램을 만들 수 있다. 별로 복잡하지 않으므로 단계를 따라 실습을 해 보도록 하자.

1 File/New Application 항목을 선택하여 새로운 프로젝트를 시작하고 프레임 폼을 만든다. 프로젝트를 시작하면 빈 폼이 주어지는데 이 폼의 FormStyle 속성을 fsMDIForm 으로 설정하기만 하면 이 폼이 프레임 폼이 된다. 프레임 폼의 Name 속성을 FrameForm 으로 정의해 둔다.

2 다음으로 차일드 폼을 만든다. File/New Form 으로 새로운 폼을 추가하고 이 폼을 차일드 폼으로 만든다. FormStyle 속성을 fsMDIChild 로 설정하고 Name 속성을 ChildForm 으로 설정한다. FormStyle 속성만 설정해 주면 차일드 폼이 된다. 이 폼에 다음과 같이 컴포넌트를 배치하자.



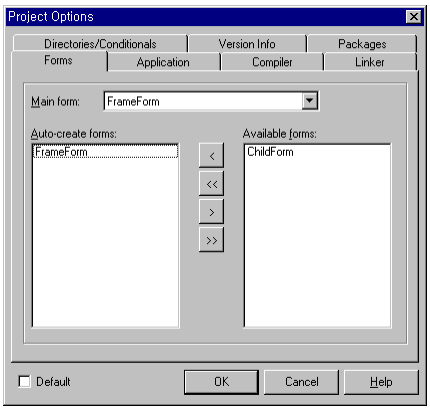

2장에서 제일 처음 만들어 보았던 예제와 같은 폼이다. 이 폼은 실행중에 만들어지는 차일드 윈도우들의 기본 형태(template)가 된다. 컴포넌트의 속성에 대해서는 2장의 예제를 참조하기 바라며 두 버튼의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TChildForm.BtnAppleClick(Sender: TObject);
begin
 Fruit.Caption:='Apple';
end;

procedure TChildForm.BtnOrangeClick(Sender: TObject);
begin
 Fruit.Caption:='Orange';
end;
```

차일드 폼은 실행중에 생성되므로 자동 생성시켜서는 안된다. 델파이는 프로젝트에 포함된 모든 폼을 실행을 시작하기 전에 미리 만들어 두는데 대개의 경우 이 방법이 편하지만 MDI의 경우는 File/New 등의 명령에 의해 윈도우를 만들어야 하므로 자동으로 생성시키지 못하도록 해야 한다. 프로젝트 옵션에서 차일드 폼을 자동 생성시키지 않도록 조정해 준다. Project/Options...를 선택하면 Auto-create forms 리스트 박스에 MainForm과 ChildForm 두개가 다 있는데 왼쪽 리스트 박스에서 차일드 폼을 우측의 리스트 박스(Available forms)로 이동시킨다. ChildForm을 선택한 후 > 버튼을 누르면 된다.

**그림**

프로젝트 옵션 대화상자




이렇게 옵션을 조정하면 델파이는 프로젝트의 소스 파일을 변경해 준다. 다음은 옵션을 변경하기 전의 프로젝트 소스이다.

```
begin
  Application.Initialize;
  Application.CreateForm(TFrameForm, FrameForm);
  Application.CreateForm(TChildForm, ChildForm);
  Application.Run;
end.
```

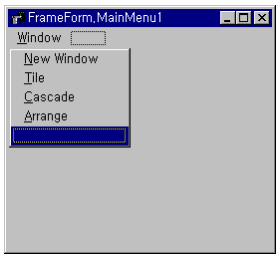
CreateForm 메소드를 호출하여 두 개의 폼을 차례대로 생성시키고 있다. 옵션을 바꾼 후에는 이 소스가 다음과 같이 변경된다.

```
begin
  Application.Initialize;
  Application.CreateForm(TFrameForm, FrameForm);
  Application.Run;
end.
```

차일드 폼을 생성시키는 코드가 삭제되었음을 알 수 있다. 폼을 생성시키는 코드는 프로젝트 소스에 있는데 CreateForm 메소드 호출을 삭제하면 자동생성을 막을 수 있다. 물론 소스를 직접 편집하는 것은 별로 바람직하지 않으므로 프로젝트 옵션 대화상자를 사용하는 것이 좋다. 이 옵션 조정에 의해 델파이가 어떤 조치를 하는지 정도만 이해하고 넘어가도록 하자.

이 예제에서는 차일드 윈도우의 형태가 하나밖에 없다. 즉 실행중에 만들어지는 차일드 폼이 모두 같은 모양을 가진다. 만약 여러 형태의 차일드 윈도우를 사용하는 MDI 프로그램이라면 차일드 폼을 필요한만큼 만들어 두고 실행중에 선택적으로 생성할 수도 있다.

3 프레임 폼의 메뉴를 만든다. 폼에 메인 메뉴 컴포넌트를 배치하고 다음과 같이 메뉴 항목들을 만들어 준다.




일단 여기까지 만들고 프로젝트를 저장한다. 프로젝트를 저장해야 유닛의 이름이 결정되기 때문이다. 프로젝트 이름을 MDI.DPR로 주고 프레임 폼, 차일드 폼의 이름을 각각 MDI_F.PAS, MDI2_F.PAS로 준다.

4 New Window 메뉴 항목에 대한 코드를 작성한다. 이 항목을 선택하면 새로운 차일드 윈도우를 만든다. 코드는 다음과 같다.

```
procedure TFrameForm.NewWindow1Click(Sender: TObject);
var
  CForm:TChildForm;
begin
  CForm:=TChildForm.Create(Self);
end;
```

차일드 윈도우의 Create 메소드를 호출하여 실행중에 새로운 차일드를 만든다. 이 메소드는 차일드 윈도우가 사용할 메모리를 할당하며 필요할 경우 데이터 초기화를 해 준다. 이 메소드 호출이 정상적으로 이루어지기 위해서는 프레임 폼과 차일드 폼이 서로를 인식할 수 있도록 uses 선언을 해 주어야 한다. 프레

임 폼의 interface부에 uses mdi2_f;를 기입해 주도록 하자.

```
unit MDI_f;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 Menus, mdi2_f;
```

5 Tile, Cascade, Arrange 메뉴 항목에 대한 코드를 작성해 준다. 차일드 윈도우를 정렬하는 명령들이다. 다음과 같이 코드를 작성한다.

```
procedure TFrameForm.Tile1Click(Sender: TObject);
begin
 Tile;
end;

procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
 Cascade;
end;

procedure TFrameForm.Arrange1Click(Sender: TObject);
begin
 ArrangeIcons;
end;
```

메소드만 호출하면 된다. 나머지는 델파이가 알아서 다 처리해 준다.

6 New Window 메뉴 항목에 의해 차일드 윈도우를 동적으로 생성하도록 했으므로 차일드가 닫히기 전에 자신을 파괴하도록 해야 한다. 그렇지 않으면 차일드 윈도우가 만들어지기만 하고 파괴되지는 않을 것이다. 차일드 폼의 OnClose 이벤트 핸들러에 다음 코드를 작성하면 된다.

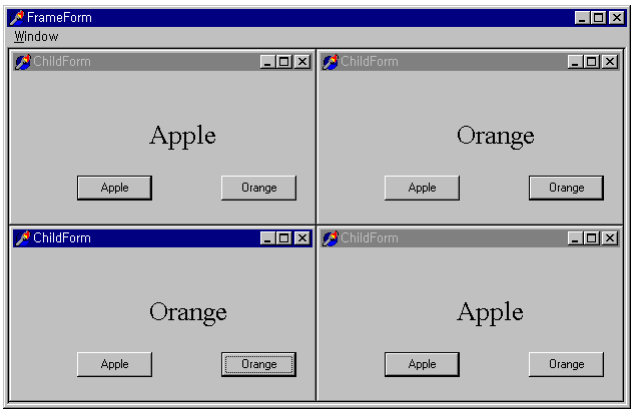
```
procedure TChildForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 Action:=caFree;
end;
```

이 이벤트의 인수로 전달되는 Action 인수는 폼을 정말로 닫을 것인가를 결정하도록 하며 다음과 같은 값을 대입할 수 있다.

| 값 | 설명 |
|---|---|
| caNone | 폼이 닫히는 것을 허락하지 않는다. |
| caHide | 폼을 닫지는 않고 숨기기만 한다. |
| caFree | 폼을 닫고 폼에 의해 할당된 모든 메모리를 해제한다. |
| caMinimize | 폼을 닫지는 않고 최소화시키기만 한다. |

MDI 차일드 폼의 경우 디폴트값이 caMinimize 이므로 이 값을 그대로 사용하면 차일드 폼이 닫히는 것이 아니라 최소화되기만 한다. 그래서 강제로 차일드 폼을 파괴하기 위해 이 값을 caFree 로 변경해 준 것이다.

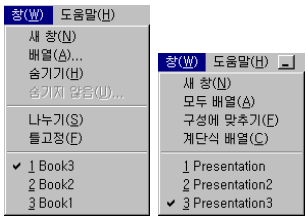
이제 예제를 컴파일한 후 실행해 보자. 다음과 같은 MDI 프로그램을 볼 수 있을 것이다. 처음 실행하면 차일드 윈도우가 없는 빈 프레임 윈도우만 나타난다. New Window 메뉴 항목을 선택할 때마다 똑같은 차일드 윈도우들이 생성된다.



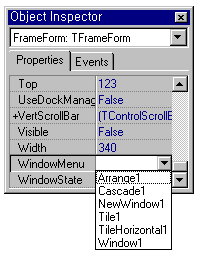

4개의 차일드 윈도우를 열어 놓고 Tile 정렬을 한 모습이다. 만들어진 차일드 윈도우들은 모두 같은 모양을 하고 있지만 별도의 메모리를 사용하는 개체들이다. 프로그램 코드는 차일드 윈도우끼리 공유한다.

이상으로 간단한 MDI 예제를 만들어 보았는데 실습을 따라해 보면 그렇게 어렵지 않다는 것을 알 수 있을 것이다. 여기서 조금 더 발전하여 간단한 MDI 팁 두 가지를 배워 보도록 하자. 먼저 대부분의 MDI 응용 프로그램들을 관찰해 보면 Window(한글 메뉴에서는 '창')라는 메뉴가 있으며 이 메뉴 아래에는 Tile, Cascade 등의 윈도우 관리 명령들이 있다. 뿐만 아니라 이 윈도우의 아래쪽에는 지금 열려진 문서들의 목록이 나타나며 목록중 하나를 선택하면 해당 윈도우

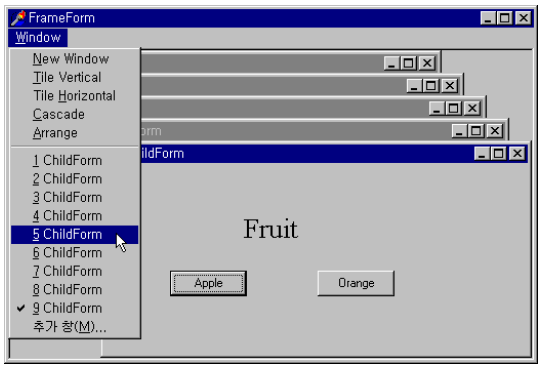
를 활성화시킨다. 다음은 엑셀과 파워 포인트의 윈도우 메뉴이며 열려 있는 워크시트와 프리젠테이션 윈도우의 목록이 메뉴 하단에 나열되어 있다.




이런 메뉴를 만들려면 열려지는 모든 차일드의 일일이 조사해야 하고 메뉴 항목을 실행중에 갱신해야 한다. 또한 메뉴가 선택되면 해당 윈도우로 포커스를 옮겨주는 기능까지도 같이 작성해 주어야 할 것이다. 그런데 이런 기능들은 사실 정형화되어 있기 때문에 사용자가 일일이 작성하지 않아도 윈도우즈 시스템이 제공해줄 수 있으며 실제로 윈도우즈는 MDI에 이런 기능을 이미 프로그래밍해 놓았다. 그래서 사용자들은 어떤 메뉴를 이런 목적에 사용할 것인가를 밝혀주기만 하면 되며 델파이에서는 폼의 WindowMenu 속성으로 이를 지정한다. 프레임 폼을 선택한 후 오브젝트 인스펙터를 보면 WindowMenu라는 속성이 있을 것이다. 이 속성을 열어 보면 다음과 같이 현재 만들어져 있는 메뉴 항목의 목록이 나타나는데 이 중 윈도우 메뉴로 사용할 메뉴를 선택해 주면 된다.



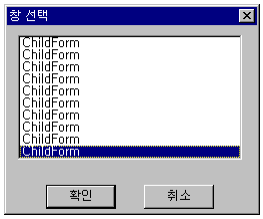

이 예제의 경우 Window1 메뉴 항목이 이런 목적에 적합하므로 이 속성을 선택해 주도록 하자. 그리고 프로그램을 다시 실행시켜 보면 이 메뉴 항목 아래에 차일드 윈도우의 목록이 나타난다.




**그림**

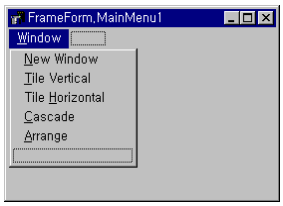
메뉴에 열려져 있는 모든 차일드의 목록이 나타난다.

이 예제의 경우 모든 차일드 윈도우의 캡션이 같아 이름만으로는 구분이 곤란하지만 보통 Child1, Child2 처럼 번호를 붙여 윈도우끼리 구분을 한다. 이 목록 중 하나를 선택하면 선택한 윈도우로 포커스를 옮겨줄 것이다. 목록은 최대 9개까지 나타나는데 만약 9개 이상의 차일드 윈도우가 열리면 제일 아래쪽에 추가 창이라는 메뉴 항목이 나타나며 이 메뉴 항목을 선택하면 모든 차일드 윈도우의 목록을 볼 수 있다.




이 윈도우도 운영체제가 제공해 주는 것이므로 별도로 프로그래밍하지 않고 사용할 수 있다. 보다시피 메인 폼의 WindowMenu 속성만 지정해도 엄청나게 많은 서비스를 공짜로 받을 수 있다.

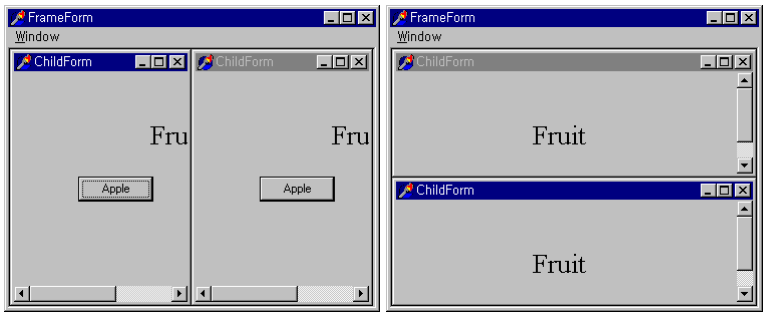
다음은 차일드 윈도우를 타일하는 방법에 대해 알아보자. 차일드 윈도우를 두 개 열어 놓은 상태에서 Tile 명령을 내리면 두 윈도우는 수평으로 정렬된다. 만약 수직으로 정렬하고자 한다면 폼의 TileMode 속성을 변경해 주면 된다. 이 속성을 tbVertical로 바꾼 후 Tile 명령을 내리면 수직으로 정렬되며 tbHorizontal로 바꾼 후 Tile 명령을 내리면 수평으로 정렬되는데 디폴트는 수평 정렬이다. 수직 정렬과 수평 정렬 명령을 따로 만들기 위해 메뉴를 다음과 같이 수정한다.




원래의 Tile 메뉴 항목을 Tile Vertical로 변경하고 그 아래에 Tile Horizontal 항목을 추가하였으며 두 메뉴 항목의 핸들러를 다음과 같이 작성하였다.

```
procedure TFrameForm.Tile1Click(Sender: TObject);
begin
  TileMode:=tbVertical;
  Tile;
end;

procedure TFrameForm.TileHorizontal1Click(Sender: TObject);
begin
  TileMode:=tbHorizontal;
  Tile;
end;
```

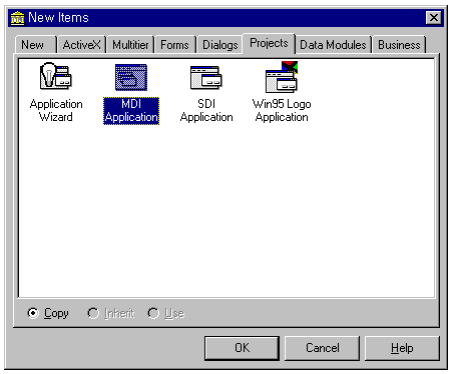
이제 다시 컴파일한 후 테스트해 보자. 다음은 두 개의 차일드를 열어 놓은 후 수평, 수직으로 정렬해 본 것이다.



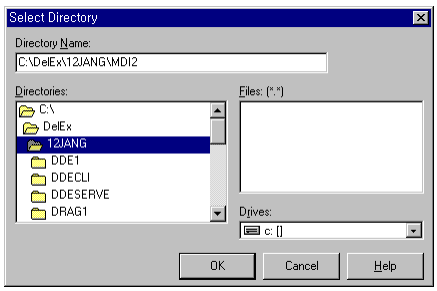

# 다. MDI 템플릿

12jang
Mdi2

MDI 프로그램을 좀 더 편리하게 만들려면 오브젝트 창고의 Projects 페이지에서 MDI Application을 선택하여 MDI 템플릿을 사용한다.

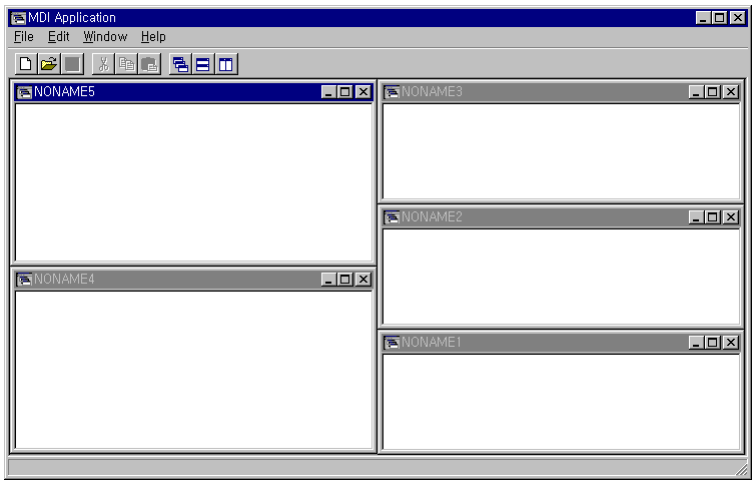



이 예제를 저장할 디렉토리를 물어오는데 적당한 디렉토리를 선택하거나 아니면 새로운 디렉토리를 만든다.

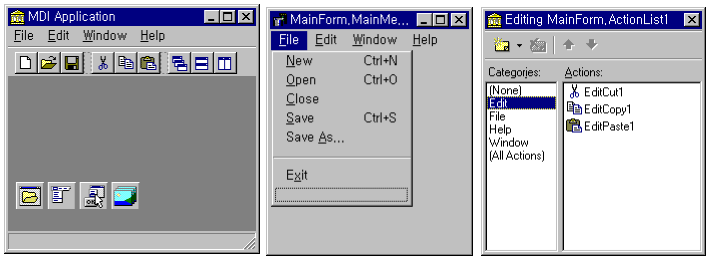



이렇게 해서 만들어진 프로그램은 그 자체로 이미 실행 가능하다. 자식 윈도우를 만드는 코드와 윈도우를 정렬하는 코드를 가지고 있으며 툴바와 상태란, 메뉴까지도 구비하고 있다. 템플릿에서 가져온 프로그램에 자신이 원하는 코드를 덧붙여 나가기만 하면 된다.

그림
MDI 템플릿으로 만든 프로젝트




템플릿을 사용하면 이렇게 만들기가 쉽다. 이 예제를 분석해 보면 MDI 프로그램을 어떻게 만드는가에 대해 배울 것이 많을 것이다. 다음은 템플릿에 의해 만들어진 폼, 메뉴, 액션 리스트이다.




특히 이 예제에서 배울만 한 것은 Edit 관련 명령들과 윈도우 관련 명령들을 모두 디폴트 액션으로 구현하고 있다는 점이다. 그래서 실제 코드는 몇 줄 되지도 않는다.

이상으로 MDI 에 관해 간단하게 살펴 보았다. 더 깊이 들어간다면 MDI 에 대해 다루어야 할 것도 많을 것이고 실제 프로젝트를 만들려면 훨씬 더 상세한 이론에 대해 알아야 하겠지만 별로 그럴 필요까지는 없을 것 같다. 왜냐하면 MDI 는 등장 초기에는 우수한 인터페이스로 인정되었지만 요즘은 MDI 가 사용되는 예가 드물다. 몇 가지 연구 결과 MDI 는 초보자에게 굉장히 혼란을 준다는 결론이 내려졌으며 마이크로소프트에서조차도 꼭 필요한 경우가 아니면 사용을 자제할 것을 권고하고 있다. 대신 탭 방식을 사용하여 여러 개의 문서를 편집한다거나 문서 당 하나의 윈도우를 배정하는 DOI(Document Oriented Interface)가 더 권장되고 있다.  탭 방식을 사용하는 프로그램의 예로는 지금 여러분들이

사용하고 있는 델파이의 코드 에디터가 있으며 DOI 형태로 짜여진 프로그램으로는 아래아 한글이 있다. 연구결과 MDI 보다는 이런 인터페이스가 훨씬 더 편리하다고 한다.

# 12-3 예외 처리

## 가. 예외

인간은 실수의 동물이며 누구나 실수를 하고 프로그래머도 실수를 한다. 그러나 설사 완벽한 프로그래머가 있고 에러가 전혀없는 프로그램이라 하더라도 에러는 여전히 발생한다. 왜 그렇게 될 수밖에 없는가 하면 프로그램이 완전하더라도 프로그램을 사용하는 사람이 불완전하기 때문이다. 어떤 경우에 이런 상황이 발생하는가 하면 다음과 같은 경우이다.

- 디스크에 없는 파일을 참조하려 했다.
- 디스크가 가득차서 더 이상 쓰기를 할 수 없다.
- 범위를 초과한 연산을 하였다.
- 메모리가 부족하다.
- 프린터를 연결하지 않거나 용지가 없는 상황에서 프린트 명령을 내렸다.
- A 드라이브에 디스켓을 넣지 않고 A 드라이브를 읽는다.

이런 경우는 프로그램을 아무리 잘 짜도 어쩔 수 없이 에러가 발생한다. 그렇다고 속수무책 에러가 발생하도록 내버려 둘 수는 없다. 에러가 발생하는 것은 어쩔 수 없지만 발생한 에러를 최대한 바로잡도록 하고 프로그램 실행을 계속할 수 있도록 해 주어야 한다. 예외(exception)란 프로그램 실행중에 어디서 어떤 종류의 에러가 발생했는가의 정보를 말하며 프로그래밍의 대상이다.

전통적인 프로그램에서는 조건문을 사용한 방어적인 예외 처리를 했었다. 예를 들어 파일 입출력 코드의 경우 에러가 발생할 소지가 있는 함수 호출의 리턴값을 철저히 조사한 후 에러가 없을 경우만 처리를 계속하는 식으로 코드를 작성한다.

```
파일 열기
if (에러가 없으며) then
  begin
  파일 읽기
  파일 닫기
  end;
```

이런 방어적인 프로그래밍을 지원하기 위해 에러가 발생할 가능성이 있는 함수들은 대부분 에러 코드를 리턴해 준다. 32비트 윈도우즈에서는 시스템이 별도의 예외 처리 메커니즘을 제공하며 델파이는 이러한 예외 처리 기능을 100% 지원한다. 하지만 예외 처리가 완벽한 해결책은 될 수 없으며 현실적으로 조건문을 사용하는 것이 더 효율적인 경우도 많다.

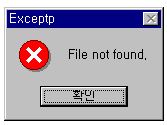
# 나. 예외 처리

12jang except

일부러 에러를 일으키는 프로그램을 작성해 보고 어떻게 예외 상황을 처리하는가 실습해 보자. 새로운 프로젝트를 시작하고 폼에 버튼 하나를 배치한다. 그리고 버튼의 OnClick 이벤트 코드를 다음과 같이 작성해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 FH:TextFile;
 St:String;
begin
  AssignFile(FH,'c:₩autoexec.bat');
  Reset(FH);
  Readln(FH,St);
  CloseFile(FH);
end;
```

루트 디렉토리의 autoexec.bat 파일을 읽는 코드이며 별다른 문제가 없어 보인다. 그러나 만약 루트 디렉토리에 이 파일이 없다면 어떻게 될까? 소스에서 파일 이름을 autoexec.exe로 또는 dogbaby 등과 같은 아주 엉뚱한 이름으로 바꾸어 보자. 그러면 프로그램 실행중에 다음과 같은 에러 메시지가 출력될 것이다.




단 이 에러 메시지를 제대로 보려면 델파이의 통합환경에서 F9를 눌러 실행시키지 말고 탐색기에서 Exceptp.exe를 직접 실행시켜 보아야 한다. 그렇지 않으면 델파이가 예외 처리를 해 주며 예외 발생 즉시 디버거를 가동시켜 버리기 때문이다. Project/Build 명령으로 컴파일만 한 후 탐색기로 직접 실행해 보아

라.

읽고자 하는 파일이 없으므로 에러 메시지가 출력되는 것이 당연하다. 문제는 에러가 발생했다고 해서 에러 메시지가 출력되는 것이 반드시 바람직하지만은 않다는 데 있다. 프로그래머는 에러 메시지를 출력하는 것보다는 에러를 바로 잡고자 한다거나 아니면 파일 이름을 재입력받도록 한다거나 이왕 에러 메시지를 출력하더라도 프로그래머가 직접 출력하도록 하고 싶어 한다. 탐색기를 관찰해 보면 A 드라이브에 디스켓이 없거나 CD-ROM 드라이브에 CD가 없어도 재시도의 기회를 제공하는 등 예외를 우아하고 부드럽게 처리하고 있음을 알 수 있다.

이 메시지는 Reset 함수 실행중에 출력되므로 프로그램 코드에서 정상적인 방법으로는 없애지 못하며 예외 보호 구역(protected block)을 설정하여야 한다. 예외 보호 구역을 설정하면 에러 메시지를 출력하지 않게 되며 프로그래머가 에러에 대한 처리를 직접 해 줄 수 있다. 예외 보호구역의 형태는 다음과 같다.

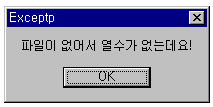
```
try
 {예외를 발생시킬 가능성이 있는 코드}
except
 {예외를 처리하는 코드}
end;
```

try~except 사이의 코드를 실행하다가 에러가 발생하면 실행을 중지하고 except~end 사이의 코드를 실행한다. 여기에 예외에 대한 처리 코드를 기입해 준다. 위의 코드를 다음과 같이 변경해 보자.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 FH:TextFile;
 St:String;
begin
try
  AssignFile(FH,'c:₩autoexec.exe');
  Reset(FH);
  Readln(FH,St);
  CloseFile(FH);
except
  MessageBeep(0);
  ShowMessage('파일이 없어서 열수가 없는데요!');
```

```
end;
end;
```

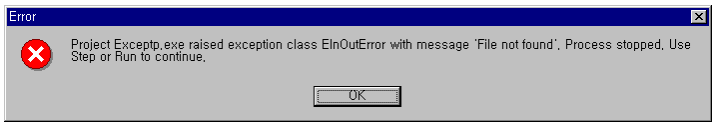
except 이하에 예외 처리 코드를 작성하였다. 여기서는 경고음을 울리고 프로그래머가 정의한 에러 메시지를 출력하도록 하였다.




except 이후에 코드를 작성하지 않으면 에러를 완전히 무시해 버리며 에러 메시지 따위는 출력되지 않는다(Silent Exception). 또는 어떤 상황의 에러가 발생했는가에 관한 정보를 상세하게 출력해 준다거나 에러 상황을 바로 잡는 코드를 기입할 수도 있다.

이 예제를 델파이 통합환경에서 실행하면 다음과 같은 메시지가 먼저 나타날 것이다.

그림
예외 메시지




왜냐하면 통합환경에서의 실행은 프로그램 실행이 아니라 사실은 디버깅이기 때문이다. 그래서 예외가 발생하면 except가 실행되기 전에 델파이가 예외를 먼저 감지하고 벌레잡기 태세에 들어가는 것이다. 델파이 환경 외부에서 프로그램이 실행될 때는 이 메시지가 나타나지 않는다.

예외는 프로그램의 고유 코드와 에러 처리 코드를 분리할 수 있도록 해주며 운영체제의 견고성과 응용 프로그램의 신뢰성을 향상시켜 준다.

# 다. 청소 코드

윈도우즈와 같이 여러 개의 프로그램이 동시에 실행되는 멀티태스킹 시스템에서는 프로그램 간에 자원 관리를 명확히 해야 한다. 한 프로그램에서 자원을 너무 많이 소비하거나 할당받은 자원을 반납하지 않으면 다른 프로그램이 사용할 자원이 부족하게 된다. 제대로 된 프로그램이라면 할당받은 자원을 반드시 해제해 주어야 한다. 반드시 해제해 주어야 하는 자원이란 파일, 메모리, 리소스,

오브젝트 등이되 결국 전부 메모리와 관련되어 있다.

물론 일부러 할당받은 자원을 반납하지 않도록 프로그램을 작성하는 경우란 없다. 하지만 예외 상황이 발생하면 어쩔 수 없이 반납하지 못하는 경우가 발생할 수도 있다. 다음 예를 보자.

```
var
  P:Pointer;
  I,D:Integer;
begin
  D:=0;
  GetMem(P,10000);
  I:=1234 div D;
  FreeMem(P);
end;
```

메모리를 동적으로 할당받은 후 나눗셈에서 예외가 발생하도록 만든 코드이다. 예외가 발생하면 예외가 발생한 블럭을 벗어나 버리므로 할당받은 메모리를 다시 반납할 기회를 박탈당하게 된다. 물론 예외가 발생하지 않는다면 위 코드는 아주 정상적으로 동작하는 아무 문제가 없는 코드이다. 예외 발생시에도 자원을 반드시 반납하도록 하려면 청소 코드(clean up code)를 작성해 주어야 한다. 청소 코드는 보호 구역의 finally 다음에 기입해 준다.

```
try
  {자원 할당}
finally
  {청소 코드:자원 반납}
end;
```

finally 이하의 코드는 예외가 발생하더라도 반드시 실행된다. 그래서 예외 발생과는 무관하게 할당받은 자원을 반드시 반납하게 된다. 위 코드를 보호 구역을 설정하여 다시 작성해 보면 다음과 같이 된다.

```
var
  P:Pointer;
  I,D:Integer;
begin
  D:=0;
try
  GetMem(P,10000);
  I:=1234 div D;
```

```
finally
  FreeMem(P);
end;
end;
```

나눗셈 연산 과정에서 예외가 발생해도 FreeMem 함수가 반드시 실행된다고 보장할 수 있다. 물론 예외가 발생하지 않아도 finally에 있는 청소 코드는 정상적으로 실행된다. 청소 코드는 또한 여러 가지 종류의 자원이 한꺼번에 할당될 때 자원을 한 곳에서 해제하기 위해 사용되기도 한다. 다음 의사 코드를 보자.

```
if (크리티컬 색션 생성 = 성공) then begin
  if (파일 열기 = 성공) then bgein
    if (메모리 할당 = 성공) then begin
      if (오브젝트 생성 = 성공) then begin
        하고 싶은 일
        오브젝트 해제
        end;
      메모리 해제
      end;
    파일 닫기
    end;
  크리티컬 색션 파괴
  end;
```

이렇게 코드를 작성해 놓으니 보기도 싫고 begin ~ end 짝을 찾기도 어렵다. 청소 코드를 사용한다면 다음과 같이 보기 좋게 작성할 수 있다.

```
try
  크리티컬 색션 생성
  파일 열기
  메모리 할당
  오브젝트 생성
  하고 싶은 일
finally
  할당한 자원 해제
end;
```

try 블럭 어디에서건 자원 해제에 실패하면 finally로 이동하여 할당된 자원을 해제할 수 있게 된다. 보기도 좋고 코드를 유지, 보수하기에도 한결 편리해진다.

except와 finally는 비슷한 것 같지만 전혀 다르므로 분명히 구분을 하도록 하자. except는 예외가 발생했을 경우 예외에 관한 처리를 하며 예외가 발생하지

않으면 없는 것처럼 무시된다. 반면 finally는 예외 발생과는 무관하며 예외가 발생해도, 예외가 발생하지 않아도 반드시 실행된다. finally는 예외를 처리하는 것이 아니라 예외와 무관하게 실행되는 코드이다.

예외 블럭과 청소 코드는 필요에 따라 중첩시킬 수도 있다. 예를 들어 청소 코드내에서 예외가 발생할 가능성이 있다면 여기에 예외 블럭을 둘 수 있으며 반대의 경우도 가능하다.

# 라. 예외의 종류

에러가 발생하면 예외 오브젝트(exception object)가 생성되며 에러의 종류에 따라 다음과 같은 예외 오브젝트들이 있다.

**표**
예외 오브젝트

| 예외 오브젝트 | 에러 |
|---|---|
| EInOutError | 일반적인 입출력 에러 |
| EOutOfMemory | 메모리 부족 |
| EDivByZero | 0으로 나누기를 할 때의 에러 |
| ERangeError | 범위를 초과한 수나 수식을 사용 |
| EIntOverFlow | OverFlow 발생 |
| EInvalidCast | as 연산자가 잘못된 타입 변환을 하려고 할 때 |
| EConvertError | IntToStr 등의 함수로 데이터형을 잘못 변환하려 할 때 |

왜 에러의 이름이 오브젝트가 되는지 잘 이해를 못할지도 모르겠다. 예외 오브젝트란 에러에 관한 정보를 담는 일종의 변수라고 생각하기 바란다. 특정한 예외에 응답하는 예외 처리 코드를 작성할 때는 except에 다음과 같이 코드를 작성한다.

```
except
on 예외 이름 do 처리;
on 예외 이름 do 처리;
else 그 외의 처리
```

어떤 종류의 예외가 발생했는가에 따라 처리를 다르게 할 수도 있다.

# 12-4 동적 객체 생성

디자인시에 컴포넌트 팔레트의 컴포넌트를 마우스로 선택하여 폼에 배치하는 방법을 정적 컴포넌트 생성이라 하며 이때까지의 실습을 통해 많이 사용해 본 방법이다. 컴포넌트를 폼에 가져다 놓기만 하면 델파이가 컴포넌트를 생성해 준다. 반면 프로그램 실행중에 컴포넌트를 생성하는 방법을 동적(dynamic) 객체 생성이라 한다.

보통 프로그램을 디자인할 때 필요한 컴포넌트의 종류와 개수를 파악할 수 있지만 가끔 그렇지 못한 경우가 있다. 필요한 컴포넌트의 개수가 가변적이라든가 때에 따라 다른 컴포넌트를 사용해야 할 경우가 있는데 이런 경우는 어떻게 하는가? 우선 가장 쉬운 방법이 실행중에 사용될 가능성이 있는 모든 컴포넌트를 디자인시에 모두 만들어 놓고 당장 쓰이지 않는 컴포넌트의 Visible 속성을 False로 만들어 보이지 않도록 해 두고 필요할 때 Visible 속성을 True로 바꾸는 방법을 생각할 수 있으며 실제로 이런 방법이 사용되기도 한다. 이렇게 하면 처음 실행시에 없던 버튼이 실행중에 생성되는 것처럼 보이게 된다.

하지만 이 방법은 어디까지나 실행중에 만들어지는 것처럼 보일뿐이지 실제로는 정적 객체 생성이며 실행중에 필요한 컴포넌트의 개수가 작고 유한할 경우에나 가능한 방법이다. 실행중에 동적으로 객체를 생성하면 어떤 종류의 컴포넌트든지 개수에 제한없이 만들 수 있다. 예제를 작성해 보자.

## 가. 동적 생성 버튼

12jang
dync

동적 객체 생성의 예를 보기 위해 가장 간단한 컴포넌트인 버튼을 실행중에 만들어 보도록 하자. 새로운 프로젝트를 시작하고 폼에는 전혀 손댈 필요없이 다음과 같이 코드만 입력한다. 동적으로 생성되는 컴포넌트는 디자인시에 마우스로 생성하는 것이 아니라 코드에 의해 만들어진다.

```
{동적으로 버튼을 생성하는 예제.}
unit Dync_f;

interface

uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
{동적으로 생성될 버튼의 OnClick 이벤트 핸들러.
사용자가 직접 입력해 주어야 한다.}
    procedure DynBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
{동적으로 생성할 버튼 컴포넌트 변수}
  DynBtn:TButton;

implementation

{$R *.DFM}

{폼이 생성될 때 버튼 컴포넌트를 같이 생성한다. Create 메소드로
버튼을 생성시킨 후 필수적으로 필요한 속성을 설정하고 OnClick
이벤트 핸들러를 지정해 준다. }
procedure TForm1.FormCreate(Sender: TObject);
begin
  DynBtn:=TButton.Create(Form1);
  DynBtn.Left:=100;
  DynBtn.Top:=100;
  DynBtn.Width:=150;
  DynBtn.Height:=50;
  DynBtn.Caption:='dynamic button';
  DynBtn.Parent:=Form1;
  DynBtn.OnClick:=DynBtnClick;

end;

{동적으로 생성한 버튼의 OnClick 이벤트 핸들러
메시지 상자만 출력한다. }
procedure TForm1.DynBtnClick(Sender: TObject);
begin
  showmessage('동적으로 생성한 버튼이 클릭되었습니다.');
end;
```

```
end.
```

폼에 아무런 컴포넌트도 배치하지 않았지만 이 예제를 실행시키면 폼 중앙에 버튼이 생성된다. 폼이 처음 만들어질 때 버튼 컴포넌트를 만들었기 때문이다. FormCreate의 첫 행에서 TButton 컴포넌트의 Create 메소드를 사용하여 버튼 컴포넌트를 생성시킨다.

```
constructor Create(AOwner:TComponent);
```

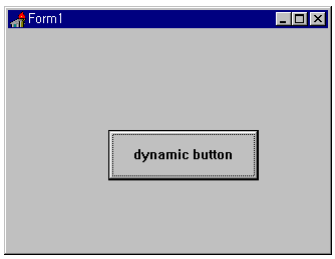
Create 메소드는 메모리를 할당하여 할당한 메모리에 컴포넌트를 만들고 필요할 경우 컴포넌트의 데이터를 초기화한다. 인수로 전달되는 AOWner는 생성될 컴포넌트의 소유주이며 대부분의 컴포넌트는 폼에 소속되므로 이 인수는 폼이 된다. Create 메소드는 생성한 컴포넌트를 리턴하므로 적당한 변수로 이 값을 받아야 한다. 위 예제의 경우는 버튼을 생성했으므로 TButton형의 변수 DynBtn으로 이 값을 대입받았다. Create 메소드는 모든 컴포넌트에 존재하며 동적으로 컴포넌트를 생성할 때 사용한다. 버튼을 만들고자 한다면 TButton.Create를 호출하는 것과 마찬가지로 에디트를 생성하고 싶다면 TEdit.Create를 호출하면 된다.

버튼을 만든 후에는 버튼의 속성을 원하는 대로 직접 대입해 주어야 한다. 디자인시에는 오브젝트 인스펙터를 사용하여 컴포넌트의 속성을 설정하지만 실행중에는 코드로 속성값을 대입할 수밖에 없다. 만약 속성을 설정하지 않으면 생성자에서 설정한 디폴트 속성이 사용되므로 원하는 위치에 원하는 컴포넌트를 배치할 수 없다. 동적으로 컴포넌트를 생성할 때 다른 속성은 디폴트를 사용해도 큰 상관 없이 어쨌든 생성은 되지만 Parent 속성은 반드시 설정해 주어야 한다.

이벤트 핸들러도 실행중에 코드로써 설정할 수 있다. 속성은 원래 실행시에도 변경시키므로 자연스럽게 이해가 되겠지만 이벤트를 실행중에 대입하는 코드가 조금 눈에 낯설 것이다. 나중에 다시 논하게 되겠지만 컴포넌트의 이벤트도 이벤트 핸들러의 포인터 값을 담는 일종의 속성이므로 실행중에 값을 변경할 수 있다. 위의 예에서는 OnClick:=DynBtnClick에 의해 클릭 이벤트가 발생하면 DynBtnClick을 호출하도록 하였다. 이벤트를 설정하려면 먼저 이벤트 핸들러를 만들어 두어야 한다. 디자인시에 존재하지 않는 컴포넌트에 대한 이벤트 핸들러이므로 디자인시에 오브젝트 인스펙터를 통해서 핸들러를 만들 수 없다. 그래서 직접 손으로 코드 에디터에서 이벤트 핸들러를 작성해 주어야 한다.

위의 예제에 있는 DynBtnClick 프로시저는 직접 손으로 입력한 핸들러이다. 이 핸들러를 입력해 주는 일뿐만 아니라 폼의 type 선언에 이 핸들러를 포함시키는 것도 물론 직접 해 주어야 한다. 실행중에 생성된 버튼을 클릭하면 이 핸들러가 호출되어 대화상자를 출력해 준다.

그림
동적으로 생성된 버튼

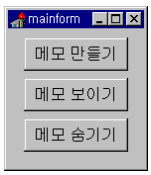



실행중에 동적으로 생성한 컴포넌트는 폼이 파괴될 때 같이 파괴된다. 왜냐하면 컴포넌트를 생성할 때 Parent를 폼으로 설정해 두었기 때문이다. 따라서 생성한 컴포넌트를 일부러 파괴해 줄 필요는 없다.

# 나. 노트 예제


12jang note

앞에서 만들어 본 예제는 어디까지나 동적 객체 생성이 어떤 것인가를 보여주기 위한 것일뿐 실용적인 가치가 없다. 이번에는 동적 객체 생성을 꼭 해야만 하는 예제를 만들어 보자. 만들 예제는 eNote나 WinPost 등과 같이 데스크탑 상에 간단한 메모를 해 둘 수 있는 프로그램이다. 만들어야 할 메모 윈도우의 개수가 가변적이기 때문에 디자인시에 필요한 메모 윈도우를 일일이 만들어 놓을 수가 없다. 그래서 실행중에 필요할 때마다 새로운 메모 폼을 생성해야 한다. 일단 폼에 다음과 같이 세 개의 버튼을 배치한다.




각 버튼의 Name 속성은 New, Show, Hide로 설정하였으며 캡션은 그림에서 보는 바와 같이 변경한다. 동적으로 폼을 생성해야 할 시점은 "메모 만들기" 버튼이 눌러질 때이다. 만드는 과정은 생략하고 아래에 전체 소스를 보인다. 배포 CD의 예제를 불러와 컴파일해 보고 분석하도록 하자.

```
{동적으로 폼을 생성시키는 예제}
unit Note_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons,note_p, StdCtrls;

type
  Tmainform = class(TForm)
    New: TButton;
    Show: TButton;
    Hide: TButton;
    procedure NewClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ShowClick(Sender: TObject);
    procedure HideClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  mainform: Tmainform;
  {동적으로 생성된 폼의 배열}
  notepad:array [1..10] of Tnotepad;
  totalnote:integer;

implementation

{$R *.DFM}

{새로운 노트의 생성. Tnotepad 형의 폼을 하나 만들고 속성을 설정한다.}
procedure Tmainform.NewClick(Sender: TObject);
begin
if totalnote<10 then
begin
  Inc(totalnote);
  Application.CreateForm(Tnotepad, notepad[totalnote]);
  notepad[totalnote].caption:='note'+inttostr(totalnote);
  notepad[totalnote].Left:=totalnote*50;
  notepad[totalnote].Top:=totalnote*30;
```

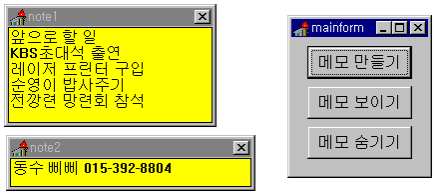
```
  notepad[totalnote].show;
end
else
  ShowMessage('10 개까지만 만들 수 있습니다');
end;

procedure Tmainform.FormCreate(Sender: TObject);
begin
totalnote:=0;
end;

{노트 보이기. 10 개의 노트를 모두 보인다.}
procedure Tmainform.ShowClick(Sender: TObject);
var
  i:integer;
begin
for i:=1 to totalnote do
  notepad[i].show;
end;

{노트 숨기기}
procedure Tmainform.HideClick(Sender: TObject);
var
  i:integer;
begin
for i:=1 to totalnote do
  notepad[i].hide;
end;

end.
```

동적으로 생성되는 폼을 저장하기 위해 Tnotepad형의 notepad 배열을 크기 10으로 정의해 두고 노트 만들기 버튼이 눌러질 때마다 Tnotepad형의 폼을 생성한다. 그렇다면 Tnotepad형이라는 폼은 어떻게 만들어진 것일까? 사용자가 만든 폼이므로 컴포넌트 팔레트에 있지도 않고 델파이가 제공하는 폼도 아니다. 이 폼은 디자인시에 미리 만들어진 폼이다. 프로젝트 관리자를 보면 알겠지만 메인 폼외에도 notepad라는 폼이 정의되어 있다. 이 폼에는 메모 컴포넌트가 하나 배치되어 있으며 alClient 정렬하여 폼 전체를 가득 채우고 있으며 메모의 배경색은 노랑색으로 설정하여 눈에 잘 띄도록 하였다. 메모 컴포넌트로 문자열을 입력받기만 하므로 폼 자체의 코드는 전혀 없다.

디자인시에 폼을 만들어 놓기는 하되 이 폼은 동적으로 생성될 폼이므로 프로

젝트 옵션의 자동생성 리스트 박스에서 제외시켰다. 그리고 메인 폼의 uses절에 이 유닛의 이름인 note_p를 추가해 주면 된다. 프로그램 실행중의 모습은 다음과 같다.

**그림**
동적으로 폼을 생성시킨다.



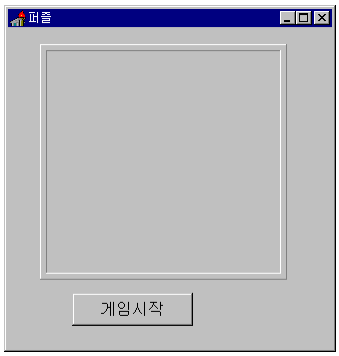

메모 만들기 버튼으로 폼을 동적으로 생성시키며 생성된 폼에 간단한 메모를 끄적거려 놓을 수 있으며 숨기기, 보이기를 자유롭게 할 수 있다. 코드를 죽 읽어보면 그다지 어렵지 않게 이해할 수 있을 것이다. 여기서는 폼을 동적 생성시키는 방법에 대해서만 알아 보았고 좀 더 완벽한 메모 예제는 13장에서 만들어 볼 것이다. 메모 내용을 자동으로 저장하고 폼의 색상, 글꼴 등을 바꿀 수 있도록 해 볼 것이다.

## 다. 퍼즐 게임

12jang
puzzle

동적 객체 생성 기법을 이용하여 퍼즐 게임을 만들어 보자. 퍼즐 게임이란 16개의 격자에 1~15까지의 숫자와 하나의 공백을 두고 격자를 섞은 후 다시 맞추는 게임이다. 숫자 외에도 그림을 조각으로 흩어 놓고 게임을 할 수도 있지만 여기서는 편의상 숫자를 사용하기로 한다. 퍼즐 게임에는 15개의 버튼이 필요하며 이 버튼들을 디자인시에 일일이 만드는 것보다는 실행할 때 동적으로 생성시키는 것이 더 간편하며 효율적이다. 동적으로 버튼을 생성시킬 경우 얻을 수 있는 이점 중에 가장 큰 이점은 버튼의 배열을 만들 수 있다는 점이다. 일단 버튼이 배열에 담겨지면 배열의 첨자(정수값)만으로 버튼을 상호 구분할 수 있으므로 대단히 편리해진다.

새로운 프로젝트를 시작한 후 패널 하나와 버튼 하나를 폼에 다음과 같이 배치하고 속성을 설정한다.




| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | Name | gameform |
| | Caption | 퍼즐 |
| | BorderStyle | bsSingle |
| 패널 | BevelInner | bvLowered |
| | BevelOuter | bvRaised |
| | BevelWidth | 5 |
| 버튼 | Name | Start |
| | Caption | 게임시작 |

게임에 사용되는 폼은 크기가 변할 수 있는 것보다는 고정되어 있는 것이 더 일반적이므로 폼의 BorderStyle 속성을 bsSingle로 설정하여 경계선을 없앤다. 패널의 위치나 크기는 실행시에 버튼의 크기에 맞게 계산하여 조정하므로 디자인시에는 설정해 주지 않아도 된다. 전체 소스 리스트는 다음과 같다.

```
{숫자 퍼즐 게임}
unit Puz_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
```

```
 Tgameform = class(TForm)
  start: TButton;
  Panel1: TPanel;
  procedure FormCreate(Sender: TObject);
  procedure FormKeyPress(Sender: TObject; var Key: Char);
  procedure move(dir:integer);
  procedure startClick(Sender: TObject);
  procedure BtnClick(Sender:TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
 gameform: Tgameform;
 Btn:array [0..14] of TButton;    {15 개의 버튼 배열 }
 puz:array [0..3,0..3] of integer; {버턴위치 보관하는 배열}
 nx,ny:integer;    {공백의 현재 위치}
 moving:boolean;   {퍼즐을 섞고 있는 중인가?}

implementation

{$R *.DFM}

procedure Tgameform.FormCreate(Sender: TObject);
var
 i:integer;
begin
randomize;   {난수 루틴 초기화}
moving:=false;
{15 개의 버튼을 동적으로 생성한다. 버튼을 배열에 넣기 위해서는
동적으로 생성하는 것이 더 쉽다}
for i:=0 to 14 do
 begin
 Btn[i]:=TButton.Create(gameform); {오너는 폼}
 Btn[i].Parent:=panel1;          {페어런트는 패널}
 Btn[i].Left:=(i mod 4)*60+7;
 Btn[i].Top:=(i div 4)*60+7;
 Btn[i].Width:=60;
 Btn[i].Height:=60;
 Btn[i].Caption:=IntToStr(i+1);
 Btn[i].OnClick:=BtnClick;    {OnClick 이벤트 핸들러 설정}
 puz[i mod 4][i div 4]:=i;    {puz 배열의 초기화}
 end;
nx:=3;   {공백의 현재 위치}
```

```
ny:=3;
{폼과 패널의 크기, 위치를 설정한다. 여기서 숫자 14 의 의미는
패널의 양쪽 경계선의 폭을 의미한다.}
gameform.width:=60*4+14;
gameform.height:=60*4+100;
panel1.Left:=0;
panel1.Top:=0;
panel1.Width:=60*4+14;
panel1.Height:=60*4+14;
end;

{키보드의 숫자키 패드에 따라 퍼즐을 움직인다.}
procedure Tgameform.FormKeyPress(Sender: TObject; var Key: Char);
begin
if Key='4' then move(1);
if Key='8' then move(2);
if Key='6' then move(3);
if Key='2' then move(4);
end;

{버튼을 움직인다. 인수로 전달되는 dir 은 움직일 방향이다.
1 이면 좌에서 우로 2 면 위에서 아래로 3 은 우에서 좌료 4 는 아래에서
위로이다. 이때 움직이는 대상은 버튼이 아니라 공백이다. }
procedure Tgameform.move(dir:integer);
var
  i,j,t:integer;
  dx,dy:integer;
  oldx,oldy:integer;
  correct:boolean;
begin
dx:=0;     {움직일 방향을 일단 0 으로 설정한다.}
dy:=0;
oldx:=nx;   {공백의 현재 위치를 저장한다.}
oldy:=ny;
{방향에 따라 공백의 좌표를 변경한다. }
case dir of
  1:begin
    if nx=3 then exit;  {공백이 우측끝 움직일 수 없다.}
    Inc(nx);          {공백을 한칸 우로 움직인다.}
    dx:=-1;           {버튼은 한칸 좌로 이동한다.}
    end;
  2:begin
    if ny=3 then exit;
    Inc(ny);
    dy:=-1;
    end;
```

```
  3:begin
    if nx=0 then exit;
    Dec(nx);
    dx:=1;
    end;
  4:begin
    if ny=0 then exit;
    Dec(ny);
    dy:=1;
    end;
  end;
{공백의 자리에 현재 버튼의 번호를 대입한다.}
puz[oldx][oldy]:=puz[nx][ny];
{움직일 대상이 되는 버튼의 번호를 변수 t 에 대입한다.}
t:=puz[nx][ny];
{새 공백의 자리에 공백임을 나타내는 15 를 기입한다.}
puz[nx][ny]:=15;
{버튼을 움직인다. 움직이는 방향은 dx, dy 에 따라 달라진다.}
for j:=1 to 20 do
  begin
  Btn[t].Left:=Btn[t].Left+dx*3;
  Btn[t].Top :=Btn[t].Top +dy*3;
  Btn[t].Repaint;
end;
{퍼즐을 다 풀었는지를 점검한다. 순서대로 버튼이 배열되었으면
퍼즐을 다 푼 것으로 인정한다.}
correct:=true;
for i:=0 to 3 do
  for j:=0 to 3 do
    if puz[i][j]<>j*4+i then correct:=false;
if ((correct) and (moving=false)) then
  ShowMessage('퍼즐을 모두 푸셨습니다.');
end;

{시작 버튼을 누르면 버튼을 난수로 50 번 움직여 퍼즐을 만든다.}
procedure Tgameform.startClick(Sender: TObject);
var
  i:integer;
  d,od:integer;
begin
moving:=true;
for i:=1 to 50 do
  begin
  d:=random(4)+1;  {되도록 잘 섞이도록 한다.}
  if ((d=1) and (od=3)) or ((d=2) and (od=4)) or
     ((d=3) and (od=1)) or ((d=4) and (od=2)) then
```

```
   begin
   Dec(i);
   continue;
   end;
  od:=d;
  move(d);
  end;
moving:=false;
end;

{버튼을 누르면 눌러진 버튼을 움직인다.}
procedure Tgameform.BtnClick(Sender:TObject);
var
  bn:integer;
  bx,by:integer;
begin
{버튼의 번호를 캡션으로 구한다.}
bn:=StrToInt((Sender as TButton).Caption)-1;
{눌러진 버튼의 현재 위치를 bx,by 에 구한다.}
bx:=Btn[bn].Left/60;
by:=Btn[bn].Top/60;
{버튼의 좌표와 공백의 좌표를 비교하여 버튼이 움직일 방향을
결정한다.}
if (bx=nx+1) and (by=ny+0) then move(1);
if (bx=nx+0) and (by=ny+1) then move(2);
if (bx=nx-1) and (by=ny+0) then move(3);
if (bx=nx+0) and (by=ny-1) then move(4);
end;

end.
```

## ■ 전역 변수

게임 전체를 통제하는 전역 변수는 다섯 개가 있다.

### □ Btn

버튼의 배열이다. TButton의 변수, 즉 동적으로 생성된 버튼을 담을 수 있는 크기 15의 배열이다. 버튼을 배열로 만들어야 하는 이유가 동적으로 버튼을 생성해야 하는 주된 이유이다.

### □ puz

게임판 위의 버튼 배치 상태를 보관하며 가로, 세로 4씩의 크기를 가지는 행렬 형식의 2차 배열이다. puz[x][y]의 값을 조사하면 x,y 격자에 위치하고 있는

버튼의 번호를 조사할 수 있다. 이 배열은 버튼을 움직일 때, 버튼의 위치를 조사할 때 사용하며 퍼즐을 다 맞추었는지를 조사할 때도 사용된다.

### ☐ nx,ny

공백의 위치, 즉 버튼이 없는 격자의 위치를 보관한다. 이 게임은 논리적으로 버튼의 움직임을 처리하기보다는 공백의 움직임을 주로 처리하므로 공백의 위치를 별도의 변수에 저장해 두어야 한다. 최초 공백의 위치는 3,3에 맞추어지며 퍼즐이 섞이면 이 위치도 변한다.

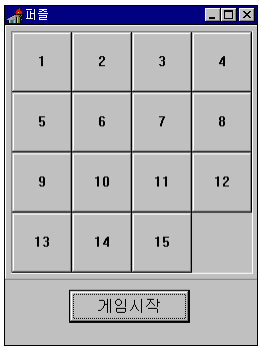
### ☐ moving

퍼즐을 섞고 있는 중이면 True값을 가지며 사용자가 퍼즐을 맞추고 있는 중이면 False값을 가진다. 이 게임에서는 사용자가 퍼즐을 움직일 때 버튼의 이동을 move 프로시저로써 처리하며 컴퓨터가 퍼즐을 섞을 때도 move 프로시저를 사용한다. 사용자가 퍼즐을 움직일 때는 퍼즐을 다 풀었는지의 여부를 매번 점검하지만 컴퓨터가 퍼즐을 섞을 때 우연히 퍼즐이 맞추어지는 경우를 배제하기 위해 이 변수를 사용한다. 즉 컴퓨터가 퍼즐을 섞을 때는 퍼즐을 맞추어도 게임을 끝내지 않고 섞기를 계속한다.

## ■ FormCreate 이벤트 핸들러

게임의 시작 위치인 FormCreate 이벤트에서는 난수 루틴 초기화, 주요 전역변수 초기화, 폼과 패널의 크기 결정 등 몇 가지 초기화와 버튼의 동적 생성을 수행한다. 0번부터 14번까지 15개의 버튼을 만들어 Btn[0]~Btn[14]까지 배열에 저장하며 버튼의 위치는 버튼 번호에 따라 좌에서 우로 위에서 아래로 순서대로 배치된다. 버튼의 크기는 60*60으로 고정되어 있으며 캡션은 버튼의 번호에 1을 더하여 0번 버튼이 1, 1번 버튼이 2, 14번 버튼이 15의 숫자를 가진다. 생성된 15개 버튼의 OnClick 이벤트 핸들러를 BtnClick 프로시저로 설정하여 게임 실행중에 사용자가 마우스로 버튼을 클릭할 수 있도록 해 준다. FormCreate 실행 후, 즉 프로그램 시작 직후에 버튼은 다음과 같이 배치된다.

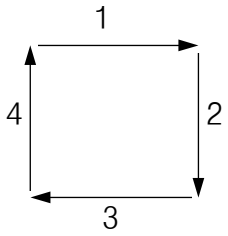
그림
동적으로 생성된 버튼들




아직 게임을 시작하지 않았으므로 최초 퍼즐은 모두 맞추어져 있는 상태이다. 이 상태에서 하단의 게임시작 버튼을 누르면 퍼즐을 무작위로 50번 섞고 게임이 시작된다. 퍼즐을 섞는 방법은 간단하다. 버튼 하나를 움직이는 move 프로시저를 50번 호출하되 움직일 방향을 난수로 생성하여 전달해 주면 퍼즐이 무작위로 섞여진다. 단, 하나의 버튼이 같은 위치를 계속 움직이는 것을 방지하여 퍼즐이 골고루 섞이도록 하는 앨거리듬을 적용하고 있다. 즉 한번 좌측으로 움직였으면 다음에 우측으로 움직이는 것을 금지하여 원위치되지 않도록 한다.

## ■ move 프로시저

이 게임의 핵심이 되는 프로시저이며 버튼 하나를 한 방향으로 움직인다. 버튼이 움직일 방향은 dir 인수로 전달되며 인수값에 따라 움직이는 방향은 다음과 같다.




여기서 한 가지 주의깊게 살펴볼 것은 화면 상으로는 버튼이 움직이지만 내부적으로는 공백이 움직이도록 처리되어 있다는 점이다. 왜냐하면 버튼은 14개나 되지만 공백은 오직 하나밖에 없기 때문에 버튼의 움직임을 처리하는 것보다 공백을 중심으로 움직임을 처리하는 것이 훨씬 더 쉽기 때문이다. 어차피 버튼이 한번 움직이는 것은 공백과 버튼의 위치를 자리 바꿈하는 것이므로 공백을 움직이나 버튼을 움직이나 방향만 잘 신경써 주면 결과는 동일하다.

인수로 전달된 공백의 이동 방향 dir에 따라 공백의 위치인 nx,ny가 조정되며 버튼의 움직일 방향은 dx,dy에 설정된다.

| 이동 방향(dir) | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| nx | 1증가 | 불변 | 1감소 | 불변 |
| ny | 불변 | 1증가 | 불변 | 1감소 |
| dx | -1 | 0 | 1 | 0 |
| dy | 0 | -1 | 0 | 1 |

dir이 1인 경우 즉 공백이 좌에서 우로 움직이는 경우(반대로 생각하면 버튼이 우에서 좌로 움직이는 경우) 공백의 x좌표가 1증가해야 하며 수직 위치는 변하지 않는다. 공백이 우측으로 이동함에 따라 버튼은 반대로 좌측으로 이동해야 하며 그래서 버튼의 위치값에 더해질 dx값이 -1이 된다. 나머지도 같은 논리로 생각해 보면 쉽게 계산이 될 것이다. 단 이동을 하더라도 무조건 이동을 해서는 안된다. 철저하게 조건을 점검해 보고 이동할 수 있는 상황인지를 조사해 본 후 이동해야 하는데 예를 들어 공백이 우측에 붙어 있는 상황에서는 더 이상 우측으로 이동할 수 없다. case문에서는 이런 조건 처리를 하여 이동 불가한 상황인 경우 exit 프로시저를 호출하여 move 프로시저를 벗어나도록 하였다.

공백의 이동 방향이 계산되고 난 후에 puz 배열 상에서 버튼과 공백의 위치를 바꾼다(swap). 그리고 화면 상으로 실제 버튼이 움직이는 모습을 보여준다. 버튼의 이동 처리는 dir 인수에 따라 결정된 dx, dy값을 버튼의 좌표에 계속 더해줌으로써 처리한다. 버튼을 이동시킬 때도 그냥 좌표값만 바꾸는 것보다는 3픽셀씩 천천히 이동하도록 함으로써 애니메이션 효과를 내도록 하였다.

move 프로시저에서 버튼 하나가 움직이고 난 후에는 사용자가 퍼즐을 다 맞추었는지를 점검해 본다. puz 배열을 순서대로 읽어보아 최초 버튼이 생성될 때처럼 버튼이 순서대로 배열되어 있는가를 점검해 보기만 하면 된다. 퍼즐이 맞게 배열되어 있으면 별다른 처리를 할 필요없이 메시지만 보여주면 된다. 좀 더 게임을 재미있게 만들려면 축하 음악을 들려준다거나 최고 기록을 유지하도록 할 수도 있을 것이다.

## ■ 사용자의 입력 처리

사용자가 버튼을 움직이는 방법에는 두 가지가 있다. 키보드를 이용하는 방법

은 극단적으로 간단하며 Form의 OnKeyPress 이벤트 핸들러에서 숫자 키 패드의 방향에 따라 move 함수를 호출해 주기만 하면 된다.

```
procedure Tgameform.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key='4' then move(1);
  if Key='8' then move(2);
  if Key='6' then move(3);
  if Key='2' then move(4);
end;
```

숫자 키 패드의 4번은 왼쪽 방향이지만 move(3)을 호출하는 것이 아니라 move(1)을 호출한다. 앞에서도 말했지만 move 프로시저 내부에서 취하는 이동 방향은 버튼의 이동 방향이 아니라 공백의 이동 방향이기 때문에 사람이 생각하는 방향과는 반대로 되어 있다. 입력된 키의 문자를 직접 비교하는 간단한 방법을 쓰고 있기 때문에 숫자 키 패드를 사용하려면 Num Lock이 켜져 있는 상태여야 한다.

마우스 처리는 키보드에 비해 조금 더 복잡하다. 버튼 15개의 OnClick 이벤트 핸들러가 버튼 생성시에 모두 BtnClick 프로시저로 지정되어 있기 때문에 이 프로시저가 호출되었을 때 어떤 버튼이 이 프로시저를 호출했는지를 먼저 조사해야 한다. 이 과정은 사실 그리 간단한 문제는 아니지만 이 게임의 경우는 우연히 버튼의 캡션 번호가 버튼의 번호보다 1이 많다는 일차 함수 관계가 성립하므로 캡션으로부터 쉽게 BtnClick을 호출한 버튼의 번호를 구할 수 있다. 코드로는 단 한 줄에 불과하다.

```
bn:=StrToInt((Sender as TButton).Caption)-1;
```

만약 이런 방법을 사용할 수 없는 상황이라면, 예를 들어 버튼에 숫자가 아닌 그림이 그려져 있다면 어떻게 해야 할까? 이때는 그림을 일일이 비교할 수 없는데 이럴 때 사용하라고 있는 속성이 바로 tag속성이다. 버튼을 생성할 때 tag속성에 버튼의 번호를 미리 입력해 놓고 OnClick 이벤트 핸들러에서는 Sender.tag를 읽어 어떤 버튼이 이벤트를 발생시켰는지를 알 수 있다.

버튼의 번호를 조사하여 bn 변수에 저장한 후 해당 버튼이 현재 어느 위치에 있는가를 구해낸다. 버튼의 위치는 버튼의 Left 속성과 Top 속성에 버튼의 크기인 60을 나누어 쉽게 구할 수 있다. 버튼이 움직일 방향은 버튼의 위치와 공백의 현재 위치를 비교해 보고 구해낸다. 즉 버튼이 공백보다 한 칸 우측에 있다면 버

튼을 좌측으로 움직이고 버튼이 공백보다 한 칸 위에 있다면 버튼을 아래로 한 칸 움직이면 된다. 공백과 인접해 있지 않은 버튼이라면 아무 동작도 할 필요가 없다.

이상으로 간단하기는 하지만 델파이로 만든 게임을 분석해 보았다. 제작 절차까지 상세하게 밝히며 설명을 하고 싶지만 이런 예제는 독자들이 직접 소스를 인쇄해서 분석해 보는 것이 좋다. 프로그래밍에 갓 입문한 초보자라면 이 예제는 아주 배울 것이 많은 영양가있는 소스가 될 것이다. 잘 이해가 되지 않는 부분은 디버거로 단계 실행을 해 가면서 꼭 분석해 보기 바라며 시간적인 여유가 있다면 이 예제를 이어 짜 보기 바란다. 숫자 대신 그림을 넣을 수도 있을 것이고 4*4배열뿐만 아니라 5*5배열로도 짜 보아라.

# 12-5 DDE

윈도우즈와 같은 멀티 태스킹 시스템에서 프로그램 간의 데이터 교환은 필수적으로 필요하다. 윈도우즈에서 제공하는 데이터 공유 방법은 우선 가장 흔한 방법으로 클립보드를 이용하는 방법이 있고 좀 더 고급 사용자를 위한 OLE, DDE 등 세 가지 방법이 제공되고 있다. DDE(Dynamic Data Exchange)는 동시에 실행되고 있는 두 개의 프로그램이 실행중에 데이터를 주고받을 수 있는 좀 더 발전된 형태의 데이터 공유 방법이다. 데이터뿐만 아니라 한 프로그램에서 다른 프로그램에게 명령을 내리거나 매크로를 전달할 때도 DDE를 사용한다. DDE는 윈도우즈의 아주 특징적인 기능으로서 중요시되고 있지만 여기서는 아주 간단한 예제만 만들어 본다. 나머지는 레퍼런스를 참조하기 바란다.

## 가. DDE 대화

DDE를 사용하여 두 프로그램이 데이터를 주고받는 것을 DDE 대화(Conversation)라고 하며 이는 두 프로그램 간의 끊임없는 메시지 전달에 의해 구현된다. 물론 델파이를 사용하면 메시지에 관해서는 크게 신경쓰지 않아도 된다. DDE 대화는 데이터를 제공하는 서버(Server) 프로그램과 데이터를 요청하는 클라이언트(Client) 프로그램으로 구성되며 델파이로 이 두 가지 유형의 프로그램을 모두 만들 수 있다. DDE 대화를 이루는 요소는 다음 세 가지이다.

① 서비스(Service) : DDE 대화의 이름이라고 할 수 있으며 보통 서버 프로그램의 실행 파일 이름이 사용된다. 필요할 경우 경로를 사용할 수는 있지만 확장자는 생략한다. 만약 서버가 델파이로 만든 프로그램이라면 프로젝트명이 그대로 서비스명이 된다.

② 토픽(Topic) : DDE 대화의 대상이라고 할 수 있으며 보통 서버 프로그램이 사용하는 파일명이 토픽으로 사용된다. 서버가 델파이로 만든 프로그램일 경우 서버폼의 Caption 속성이 토픽이 된다.

③ 아이템(Item) : 교환의 대상이 되는 데이터의 특정 부분을 지정한다. 데이터베이스의 레코드나 필드, 스프레드 시트의 셀 범위 등이 아이템이 된다.

델파이는 DDE를 위해 네 개의 컴포넌트를 제공하며 모두 system 팔레트의 오른쪽에 사이좋게 모여 있다.

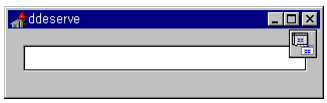

# 나. 문자열 교환

12jang
ddeser

여기서는 아주 간단한 DDE 프로그램을 만들어 보자. 원래 DDE는 기존 프로그램(예를 들어 엑셀이나 워드)과의 데이터 교환을 할 수 있도록 범용적으로 짜여져야겠지만 모든 독자들의 컴퓨터에 이런 프로그램이 다 있는 것은 아니므로 여기서는 서버와 클라이언트를 한꺼번에 만들어 보기로 하자. 두 프로그램은 가장 간단한 데이터인 문자열을 교환하는 시범을 보일 것이다.

## ■ 서버의 제작

서버 제작에 필요한 컴포넌트는 에디터 하나와 TDDEServerItem 컴포넌트 하나뿐이다. 두 컴포넌트는 다음과 같이 배치한다. TDDEServerItem 컴포넌트는 실행중에는 보이지 않는 비가시적 컴포넌트이므로 아무 곳에나 배치해도 상관없다.




Edit1.Text 속성을 깨끗하게 지워 놓고 폼의 Caption 속성을 'Serve Form'으로 바꾸어 준다. 폼의 캡션 속성이 DDE 대화의 토픽이 된다. 필요한 코드는 다음 하나뿐이다.

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  DdeserverItem1.Text:=Edit1.Text;
end;
```

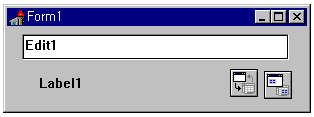
이 코드는 에디트의 내용이 바뀔 때마다 DDEServerItem 컴포넌트의 Text 속성에 변경된 문자열을 대입한다. 이렇게 대입된 문자열은 DDE 대화를 통해 클

라이언트로 자동으로 전달되며 클라이언트는 이 문자열을 자신의 폼의 어딘가에(예를 들어 에디트나 메모에) 출력하거나 다른 용도로 사용하게 될 것이다. 이 프로젝트를 ddeserve.dpr이라는 이름으로 디스크에 저장한다. 프로젝트명은 DDE 대화의 서비스명이 된다. 실행 파일을 만들기 위해 일단 한번 실행해 보자. 아직 클라이언트가 만들어지지 않았으므로 별다른 동작은 하지 않을 것이다.

## ■ 클라이언트의 제작

12jang
ddecli

다음은 서버보다는 조금 더 복잡한 클라이언트를 만들어 보자. 필요한 컴포넌트는 에디트, 레이블, DDEClientConv, DDEClientItem 네 가지이다.




DDEClientItem1.DdeConv 속성에 DDEClientConv 컴포넌트의 이름을 지정하여 두 컴포넌트를 연결해 준다. 서버와 클라이언트를 연결시키는 코드는 폼이 처음 만들어지는 시점인 FormCreate 이벤트에 작성되어 있다.

```
{폼이 처음 만들어질 때 DDE 대화를 연결한다.}
procedure TForm1.FormCreate(Sender: TObject);
begin
  if DdeClientConv1.SetLink('ddeserve','Serve Form') then
    begin
    label1.caption:='연결되었습니다.';
    DdeclientItem1.DdeItem:='DdeServerItem1';
    end
  else
    label1.caption:='연결에 실패했습니다.';
end;
```

대화를 연결시키려면 DDEClientConv 컴포넌트의 DdeService 속성에 서비스의 이름을 지정해 주어야 하며 DdeTopic 속성에 서비스의 토픽을 지정해 주어야 한다. 이 두 가지 연결을 한꺼번에 해 주는 메소드가 SetLink이며 첫 번째 인수로 서비스명을, 두 번째 인수로 토픽명을 지정한다. 서비스명은 서버의 프로그램명이므로 ddeserve가 사용되며 토픽명은 서버폼의 캡션 속성을 지정해 주었다. SetLink 메소드는 연결이 성공하면 True를 리턴하고 연결이 실패하면 False를 리턴한다. 서버가 먼저 실행되었다면 연결은 성공하겠지만 클라이언트를 먼저 실행하면 연결은 실패한다. 성공 여부는 레이블에 문자열로 출력하도록

하였다.

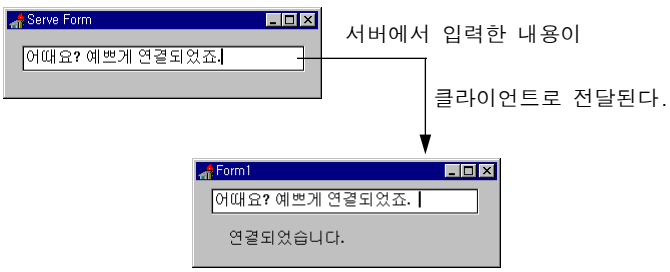
DDE 대화가 연결되면 DDEClientItem 컴포넌트의 DDEItem 속성에 DDE 대화의 아이템을 지정해 주어야 한다. 서버가 델파이로 만든 프로그램일 경우 아이템은 서버의 DDEServerItem 컴포넌트의 Name 속성이 사용된다.

DDE 대화가 성립되면 서버는 자신의 데이터가 변경될 때마다 클라이언트로 변경된 데이터를 보내게 된다. 클라이언트에서 해야 할 일은 서버가 보내준 데이터를 적절한 형태로 사용자에게 보여주는 일이다. 서버가 보낸 데이터는 DDEClientConv 컴포넌트를 거쳐 DDEClientItem 컴포넌트의 Text 속성에 대입되며 이 속성이 변할 때마다 OnChange 이벤트가 발생한다. 이 이벤트에서 Text 속성을 사용(출력, 기억, 인쇄, 계산 등등)하면 된다.

```
{서버에서 데이터가 전달되면 에디트로 출력한다.}
procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
  Edit1.Text:=DdeClientItem1.Text;
end;
```

이 예제에서는 대화가 성립되었음을 보여주기만 하므로, Edit1.Text로 전달받은 문자열을 출력하기만 했다. 코드 작성을 완료했으면 이 프로젝트를 ddecli.dpr이라는 이름으로 디스크에 저장한다. 이제 과연 서버와 클라이언트가 제대로 데이터를 교환하는지 점검을 해 볼 차례다. 두 개의 프로그램을 동시에 실행시켜야 하되 이 예제의 경우는 클라이언트가 실행됨과 동시에 서버를 찾으므로 서버를 먼저 실행시켜야 한다. 물론 클라이언트에서 서버를 찾는 코드가 버튼 클릭이나, 메뉴 항목에 들어 있다면 실행 순서는 아무래도 상관없으며 그렇게 만드는 것이 더 좋다. 실행중의 모습은 다음과 같다.

그림
DDE 에 의한 데이터 교환

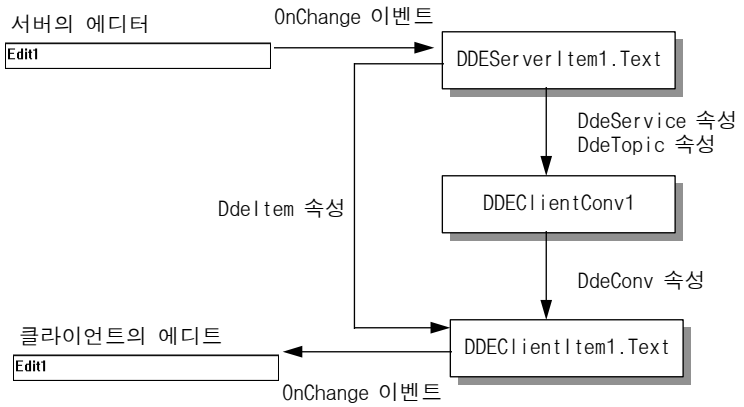



## ■ 연결 관계

예제를 만들어 보면 과연 제대로 실행된다는 것을 직접 확인할 수는 있지만

어떤 경로를 통해 어떻게 데이터가 전달되는지 정리가 잘 안될 것이다. 그래서 그림으로 정리를 해 보았다. 디자인시에 속성으로 연결되기도 하며 실행시에 코드나 이벤트를 통해 연결되기도 한다.

그림
DDE 연결 관계

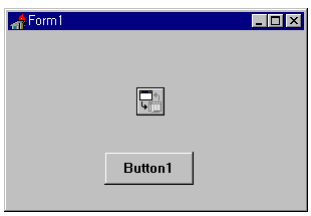



보다시피 비교적 복잡한 경로를 거쳐 두 개의 에디터가 연결되어 있다. 이런 DDE 프로그램을 C++로 짜려면 이보다 훨씬 더 복잡해진다. 그나마 컴포넌트를 사용하니까 이만큼이나마 간단해지는 것이다.

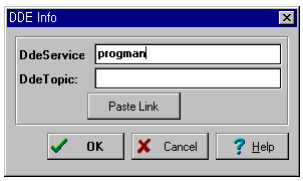
## 다. 매크로 실행


12jang
dde1

여기서 만드는 DDE 예제는 프로그램 관리자에게 매크로를 넘겨주어 프로그램 그룹을 만들도록 하고 그룹 안에 프로그램 항목을 등록하도록 한다. 흔히 설치 프로그램에서 사용하는 방법이다. 새로운 프로젝트를 시작하고 DdeClientConv1 컴포넌트와 버튼 하나를 폼에 배치한다.



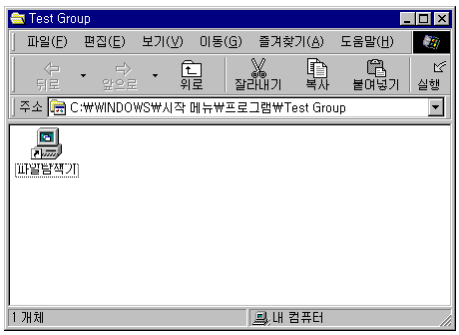
DdeClientConv1 컴포넌트의 DdeService 속성의 ... 버튼을 눌러 DDE 정보 대화상자를 불러낸 후 DdeService 란에 progman을 입력한다. 이로써 우리가 작성하는 예제는 프로그램 관리자와 동적으로 데이터나 매크로를 교환하게 된다. DdeService란에는 서버 프로그램이 될 응용 프로그램의 이름을 완전 경로로 입력해 주되 패스가 설정되어 있다면 프로그램 이름만 입력해도 되며 확장자는 생략 가능하다.




그리고 버튼의 OnClick 이벤트를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Macro: String;
begin
  DDEClientConv1.OpenLink;
  Macro := '[CreateGroup(Test Group)][ShowGroup(Test
    Group,1)] [AddItem(Explorer.EXE,파일탐색기)]'+#13#10;
  DDEClientConv1.ExecuteMacro(PChar(Macro), False);
  DDEClientConv1.CloseLink;
end;
```

프로그램을 실행시키고 버튼을 누르면 Test Group이라는 폴더가 생성되며 이 그룹에 탐색기가 등록된다.




OpenLink 메소드는 DDE 대화를 초기화하며 ExecuteMacro는 서버 프로그램으로 매크로 명령을 전달함으로써 클라이언트가 서버에게 특정 동작을 요청

한다. 전달되는 매크로 내용은 서버가 어떤 프로그램인가에 따라 달라진다. 위의 경우는 프로그램 관리자가 서버이므로 CreateGroup(그룹을 만들어라), ShowGroup(그룹창을 보여라), AddItem(프로그램 항목을 추가하라) 등의 매크로가 전달되었다.

# 12-6 OLE

OLE(Object Linking & Embedding)란 두 개의 프로그램이 OLE 오브젝트라고 불리우는 데이터를 공유하는 방법이며 윈도우즈의 대표적인 기능 중의 하나이다. 적어도 델파이를 공부하는 사람이라면 OLE가 무엇인가를 설명할 필요는 없을 것이다. OLE를 사용하면 문서에 그림을 넣을 수도 있고 도표나 사진을 넣을 수도 있는데 이렇게 여러 개의 오브젝트가 포함된 문서를 복합 문서(Compound Document)라고 한다. 바로 지금 여러분들이 보고 있는 이 책이 OLE의 혜택에 의해 만들어진 책이다.

OLE가 구현되려면 OLE 서버, OLE 컨테이너, 그리고 OLE 오브젝트가 필요하다. OLE 서버란 데이터를 제공하는 프로그램을 말하며 OLE 컨테이너란 서버로부터 데이터를 제공받는 프로그램을 말한다. DDE에서 서버, 클라이언트라고 부르는 것과는 비교가 된다. 두 프로그램이 교환하는 데이터가 OLE 오브젝트이다. OLE 오브젝트는 문서, 그림, 사운드, 사진, 스프레드 시트, 동영상 등 윈도우즈 상에서 사용되는 모든 데이터들이 포함된다.

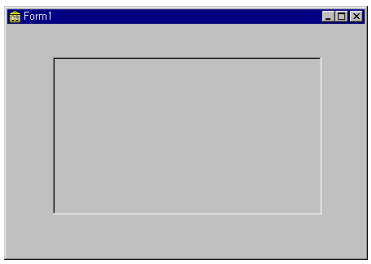
윈도우즈 상에서 실행되는 모든 프로그램이 다 OLE를 지원하는 것은 아니지만 MS-WORD, 파워 포인트, 엑셀, 비지오, 코렐 드로우, 페인트 샵 등의 웬만한 대형 소프트웨어는 모두 OLE를 지원하고 있다. 델파이는 OLE 컨테이너만 지원하며 OLE 서버는 만들지 못한다. 그렇다고 해서 정말로 OLE 서버를 못 만든다는 뜻은 아니며 다만 컴포넌트와 비주얼 툴의 도움을 받지 못한다는 얘기다. 윈도우즈 API를 직접 사용한다면 물론 만들 수도 있다.

OLE도 응용 프로그램과 마찬가지로 버전을 가지며 현재 1.0과 2.0이 동시에 공존하고 있다. OLE 1.0에서는 서버가 별도의 윈도우에서 실행되며 데이터 수정 후 다시 컨테이너로 복귀한다. OLE 2.0에서는 인플레이스 액티베이션(in place activation)을 지원하여 서버가 컨테이너의 작업 영역 상에서 실행되며 메뉴, 상태 바, 툴바를 통합한다. 어떤 버전이 사용될 것인가는 서버가 지원하는 버전에 따라 달라진다. 컨테이너가 아무리 2.0을 지원해도 서버가 1.0만 지원하면 사용되는 OLE 버전은 1.0이다.

12jang
ole1

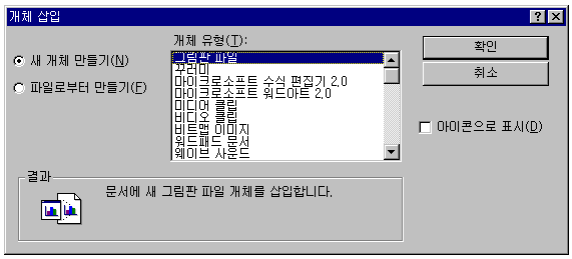
델파이는 디자인시에 OLE를 사용할 수 있으며 실행시에도 OLE를 사용할 수 있다. 디자인시에 생성되는 OLE 오브젝트는 실행 파일에 같이 저장되므로 실행 파일의 크기가 커진다. 디자인시의 OLE 사용 실습을 위해 새로운 프로젝트를 시작하고 System 팔레트의 OLEContainer 컴포넌트를 폼에 배치한다. 컴포넌트

의 크기를 적당히 조절하면 다음과 같은 모양이 될 것이다.

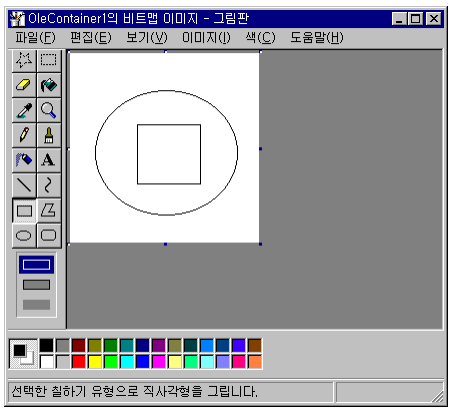



이 컴포넌트의 팝업 메뉴에서 Insert Object 항목을 선택하면 컨테이너에 개체를 삽입할 수 있는 대화상자가 열린다.

그림
개체 삽입 대화상자

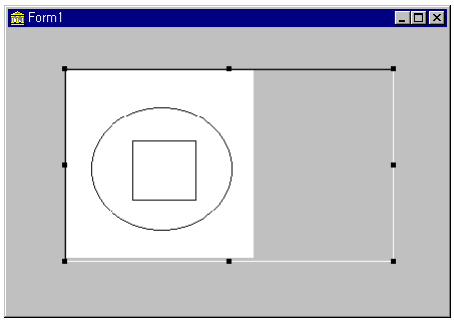



이 대화상자에서 삽입하고자 하는 개체를 선택한다. 가장 간단한 그림판 파일을 선택해 보자. 그러면 "그림판"이라는 프로그램이 실행되면서 그림을 그릴 준비를 한다. 다음과 같이 그림을 그리고 난 후에 그림판을 종료한다.

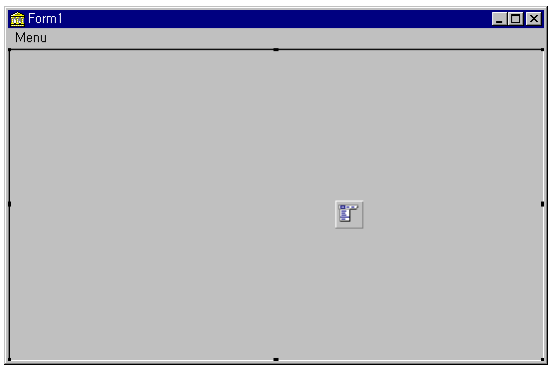



그러면 다음과 같이 그림판에서 그렸던 그림이 폼에 나타나게 된다.

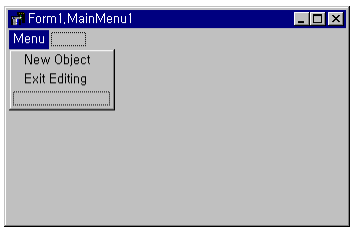
그림
OLE 에 의한 개체 삽입




이렇게 삽입된 오브젝트는 실행 파일에 같이 포함되며 프로그램을 실행시킨 후에도 오브젝트를 더블클릭하여 편집할 수 있다. 그러나 삽입된 개체의 종류가 고정되어 있으며 여러 개의 개체를 포함할 수 없으므로 이 정도 프로그램을 OLE 컨테이너라고 표현하기는 어렵다. 진정한 OLE 컨테이너 프로그램이라면 실행중에 오브젝트를 마음대로 삽입할 수 있어야 한다. 실행중에 개체를 삽입할 수 있는 프로그램을 작성해 보자. 새로운 프로젝트를 시작하고 OLEContainer 컴포넌트와 메인 메뉴를 배치한다.



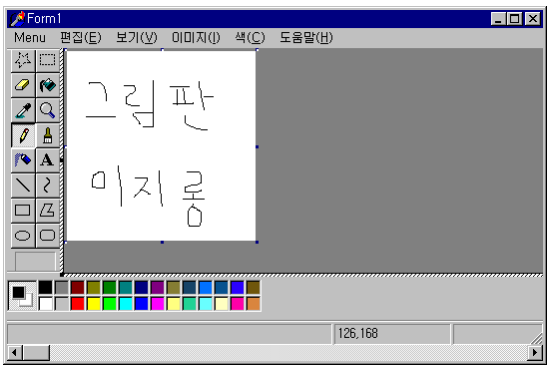

OLE 컨테이너는 폼에 꽉 차도록 alClient 정렬을 하며 메인 메뉴에는 다음과 같이 두 개의 메뉴 항목을 만든다.




그리고 각 메뉴 항목의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.NewObject1Click(Sender: TObject);
begin
  if OleContainer1.InsertObjectDialog then
    OleContainer1.DoVerb(ovPrimary);
end;

procedure TForm1.ExitEditing1Click(Sender: TObject);
begin
    OleContainer1.Close();
end;
```

InsertObjectDialog 메소드는 개체 삽입 대화상자를 호출하며 사용자가 선택한 개체를 컨테이너에 삽입해 준다. DoVerb 메소드는 컨테이너에게 여러 가지 명령을 내리는 메소드인데 ovPrimary 동사는 컨테이너가 편집 모드로 들어가도록 해 준다. 그래서 New Object 메뉴 항목을 선택하면 컨테이너에 새로운 오브젝트가 삽입되며 즉각 편집할 수 있는 상태가 된다. 다음은 그림판 개체를 편집중인 모습이다.




마치 그림판이 실행되고 있는 것 같지만 사실은 Ole1.exe의 내부에서 그림판을 호출한 것이다. 메뉴가 Ole1.exe의 것과 그림판의 것이 병합되어 있음을 알 수 있다. 편집을 종료하려면 컨테이너의 Close 메소드를 호출하면 되며 다시 편집 상태로 들어가려면 오브젝트를 더블클릭하면 된다.

위의 예제를 보면 알겠지만 OLEContainer 컴포넌트 하나가 담을 수 있는 오브젝트는 오직 하나밖에 없다. 여러 개의 오브젝트를 포함시키고 싶다면 그 수만큼 OLEContainer 컴포넌트를 배치해야 한다. 윈도우즈용 워드 프로세서와 같이 오브젝트를 제한없이 포함시키고 싶다면 OLEContainer 컴포넌트를 실행중에 동적으로 생성시켜야 한다.

# 12-7 DLL

## 가. DLL이란

라이브러리(Library)란 함수, 데이터, 타입 등 여러 가지 프로그래밍 요소들의 집합이다. 한번 라이브러리를 만들어 놓기만 하면 다시 만들 필요없이 불러서 사용할 수 있으므로 개발 속도도 빨라지고 신뢰성있는 프로그램을 만들 수도 있다. 파스칼의 유닛(Unit)은 컴파일 단위인 모듈임과 동시에 라이브러리이기도 하다. 그래서 우리는 별도로 함수를 작성하지 않아도 델파이의 표준 유닛에 있는 Writeln, IntToStr, StrCopy 등의 함수들을 사용할 수 있다. 라이브러리에 정의된 함수와 데이터는 링커에 의해 실행 파일에 그대로 옮겨지며 실행 파일의 일부분이 된다. 한글 입출력 유닛, 통신 유닛 등 많은 공개 유닛이 있다.

이런 전통적인 라이브러리 연결 방식을 정적 링크(Static Link)라고 하며 컴파일시에 라이브러리의 코드가 실행 파일에 복사된다. 이에 반해서 여기서 설명하고자 하는 DLL은 동적 연결(Dynamic Link)을 사용한다. 동적 링크란 컴파일시에 함수의 코드가 실행 파일에 복사되는 것이 아니라 실행중에 라이브러리에 있는 함수를 호출하는 방법을 말한다. 프로젝트에 소속된 유닛은 링커에 의해 실행 파일에 같이 포함되지만 DLL은 실행 파일에 포함되지 않고 호출 정보만 포함된다.

DLL을 만들고 사용하는 과정은 일반적으로 유닛을 사용하는 것보다 복잡하고 까다롭다. 하지만 다음과 같은 여러 가지 이점이 있기 때문에 윈도우즈에서는 DLL이 널리 사용되고 있다.

① 같은 코드를 여러 프로그램이 동시에 사용하기 때문에 메모리가 많이 절약된다.

② DLL 을 사용하는 프로그램은 코드가 분리되어 있기 때문에 크기가 작다. 그래서 프로그램 로딩 속도도 훨씬 빨라지며 하드 디스크 용량도 덜 소비하게 된다.

③ DLL 을 교체하여 프로그램의 성능을 향상시키기 쉽다. 이때 DLL 을 사용하는 프로그램은 다시 컴파일하지 않아도 된다.

④ 리소스의 교체가 가능하다. 이런 용도로 DLL 을 사용하면 다국어를 지원하는 프로그램을 만들 수 있다.

⑤ DLL 은 보통 엄격한 테스트를 거친 신뢰성있는 모듈이므로 디버깅이 용이해진다. 그러나 반대로 DLL 에 버그가 있을 경우는 오히려 디버깅이 더 어려워지거나 불가능할 수도 있다.

⑥ DLL 은 개발 툴에 독립적인 모듈이므로 여러 가지 개발 툴을 섞어서 사용하는 혼합 프로그래밍이 가능해진다. 그래서 쉬운 부분은 쉽게 만들고 어렵고 까다로운 부분은 DLL 로 만드는 방법이 흔히 사용된다.

물론 여러 개의 프로그램이 DLL을 동시에 사용하기 때문에 버전 관리를 세심하게 해야 한다는 단점도 있다. 또한 DLL을 사용하는 프로그램은 DLL을 같이 배포해야 하는 번거로움이 있다. 게다가 델파이로 작성된 DLL은 메모리 효율에 그리 큰 공헌을 하지 못한다. 그래서 델파이로는 되도록이면 DLL을 만들지 않는 것이 좋다. 델파이는 DLL이 아니더라도 템플릿 , 유닛, 컴포넌트 등과 같은 프로그램 코드를 분할하는 더 좋은 해결 방법을 제공해 주고 있다.

그럼에도 불구하고 델파이가 DLL 제작을 지원하는 이유는 꼭 DLL을 사용해야 할 경우가 있기 때문이다. 어떤 경우에 DLL이 필요한가 하면 다른 언어나 개발 툴과 혼합 프로그래밍을 할 경우이다. DLL은 언어에 상관없이 쓸 수 있는 모듈이므로 델파이로 만든 DLL을 C에서 부를 수 있고 C에서 만든 DLL을 델파이에서 호출할 수 있다. 그런 경우가 아니라면 DLL을 만들 생각은 하지 않는 것이 좋을 것 같다. 그래도 어떤 어려움을 무릅쓰고라도 DLL을 만들고야 말겠다는 비장한 각오를 가진 사람은 이어지는 내용을 계속 읽어보기 바란다.

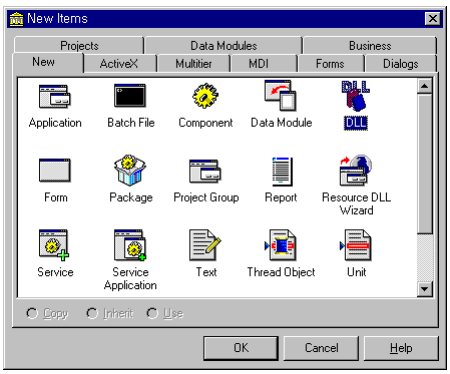
델파이는 DLL을 만들 수 있는 거의 유일한 비주얼 개발 툴이다. 오브젝트 파스칼에 기초한 컴파일러(인터프리터가 아닌)이기 때문에 DLL을 만드는 것이 가능하다. 운영체제는 DLL을 사용하는 프로그램(클라이언트)이 로딩될 때 클라이언트의 메모리 공간으로 DLL을 매칭하여 같은 주소 공간에 둠으로써 클라이언트에서 DLL의 함수를 자유롭게 호출할 수 있도록 한다. 또한 사용 카운트(Usage Count)라는 것을 유지하여 DLL을 사용하는 모든 프로그램이 종료되었을 때 DLL을 메모리에서 해제한다.

# 나. DLL 만들기

DLL은 실행 파일과 거의 동일한 구조(PE 포맷)를 가지므로 실행 파일을 만드는 절차와 동일하게 프로그래밍할 수 있다. 그러나 DLL의 특성상 폼이나 대화상자를 만드는 경우는 드물고 보통 수학적 계산 루틴이나 내부적인 메모리 처리 함수가 DLL에 작성되므로 폼 없이 함수들만 만들어지는 경우가 많다.

12jang
mydll

여기서 만들어 볼 예제는 mybeep라는 이름의 "삑" 소리만 내는 함수 하나를 가지는 DLL이다. 함수 하나만 만들기 때문에 폼은 필요없다. File/New 항목을 선택한 후 New Item 대화상자에서 DLL을 선택한다.




그러면 폼이 없는 프로젝트만을 만들어 줄 것이다. 델파이가 만들어 준 프로젝트의 소스는 다음과 같다.

```
library Project1;

uses
  SysUtils,
  Classes;

begin
end.
```

이 소스를 일단 다음과 같이 수정하도록 하자.

```
library Project1;

uses
```

```
 windows;

procedure mybeep;export;
begin
 MessageBeep(0);
end;

exports
 mybeep;

begin
end.
```

이 파일이 우리가 만들 DLL의 소스 파일이다. 일단 이 프로젝트를 Mydll.dpr로 저장한다. 그러면 제일 첫 줄이 library mydll로 변경될 것이다. 델파이가 만들어 주는 일반적인 프로젝트 파일과 무엇이 다른가 살펴보자.

① 소스의 선두가 program 이 아니라 library 로 바뀌어졌다. 그래서 이 소스를 컴파일하면 EXE 가 생성되는 것이 아니라 DLL 이 생성된다. DLL 의 이름은 library 키워드 뒤에 기입되어 있는 이름을 따라간다.

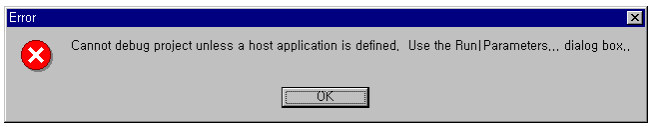
② mybeep 라는 함수가 하나 선언되어 있되 export 라는 키워드가 뒤에 붙어 있다. 이 키워드는 함수가 far 콜을 하도록 하며 외부 프로그램이 쓸 수 있도록 특별한 코드를 함수의 앞뒤에 붙인다. 함수 내부에서는 MessageBeep 함수를 호출하여 비프음을 내며 이 함수를 쓰기 위해 Windows 유닛이 uses 절에 포함되어 있다.

③ exports 선언이 되어 있다. 이 선언은 DLL 에서만 사용되며 함수(procedure와 function)를 외부 프로그램에 공개한다. 공개할 함수의 리스트를 적어주되 몇 가지 옵션을 뒤에 붙일 수 있다. 옵션에 관해서는 도움말을 참조하기 바란다.

④ 프로젝트의 본체는 begin 으로 시작해서 곧바로 end.로 끝이 난다. 즉 이 프로젝트는 함수를 정의하는 것 외에는 아무 일도 하지 않는다. 직접 실행할 실행 파일이 아니기 때문에 프로젝트 자체의 코드는 불필요하다.

이제 이 소스를 컴파일시키면 MyDll.DLL 파일이 생성된다. 실행 파일이 아니므로 F9를 누르거나 Run 명령을 실행하면 다음과 같은 에러 메시지가 시야에

들어올 것이다.

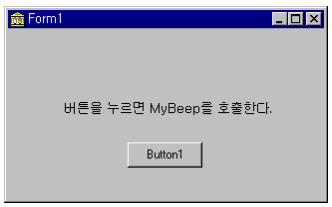



독립된 실행 파일이 아니므로 단독으로 실행할 수 없다는 메시지이다. 그래서 Project 메뉴의 Build 항목을 선택해서 컴파일해야 한다. 하긴 실행을 시켜도 어차피 DLL이 만들어지기는 마찬가지다. 이 예제에서는 예제로서의 간편함과 독자의 손가락을 생각하여 비프음을 내는 함수를 작성했지만 필요에 따라 얼마든지 복잡한 함수를 작성해 넣을 수도 있으며 한 DLL에 여러 개의 함수를 작성하는 것도 물론 가능하다.

# 다. DLL 사용하기

12jang\Mydll
UseDll.dpr

그럼 이제 앞에서 만든 DLL을 사용해 보자. 새 프로젝트를 시작하고 이 프로젝트를 MyDll 프로젝트와 같은 디렉토리에 UseDll이라는 이름으로 저장한다. 왜냐하면 DLL이 같은 디렉토리에 있어야 호출이 가능하기 때문이다. 호출이 제대로 되는지만 확인하면 되므로 버튼 하나만 배치하였다. 레이블은 설명을 위해 배치해 보았다.




버튼의 OnClick 이벤트에서 mybeep 함수를 호출한다.

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

그러나 이렇게 덮어놓고 호출하면 델파이는 mybeep가 유닛 내부 어딘가에 있는 줄 알고 찾아볼 것이며 끝내 찾지 못하고 에러 메시지를 낼 것이다. 그래서 mybeep는 유닛에 있지 않고 DLL에 있다는 것을 다음과 같이 가르쳐 주어야 한

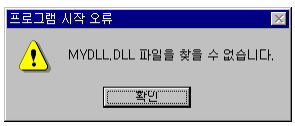
다.

```
procedure mybeep;external 'MYDLL.DLL';
```

이 선언은 mybeep라는 프로시저가 MYDLL.DLL이라는 외부 파일에 있다는 뜻이며 external 뒤에 함수가 있는 DLL 파일명을 적어 주면 된다. 이 선언문은 mybeep가 호출되기 전에 위치해야 하는데 통상적으로 implementation부의 앞부분에 선언해 두면 된다. 프로그램을 실행시킨 후 버튼을 누르면 DLL에 있는 mybeep 함수가 호출되며 "삑"하는 소리가 날 것이다. DLL 만들기와 사용하기에 성공했다. 윈도우즈는 응용 프로그램이 DLL을 호출할 때 다음 순서대로 DLL을 검색해 본다.

① 프로그램의 디폴트 디렉토리
② 프로그램의 현재 디렉토리
③ 윈도우즈의 시스템 디렉토리
④ 윈도우즈 디렉토리
⑤ PATH 환경 변수가 지정하는 모든 디렉토리

DLL은 반드시 이 경로중 하나에 있어야 하는데 보통 윈도우즈의 시스템 디렉토리에 DLL을 배치하며 혼자서 사용하는 DLL이라면 자신의 디렉토리에 두어도 된다. 만약 이 경로에 DLL이 없다면 다음과 같은 에러 메시지를 내고 프로그램 실행은 중단된다.




설치 프로그램을 만들 때는 프로그램이 사용하는 DLL을 위 경로중 하나에 복사하도록 해 주어야 한다.

# 라. 명시적 호출

실행 파일에서 DLL 의 함수를 호출하는 방법에는 다음 두 가지가 있다.

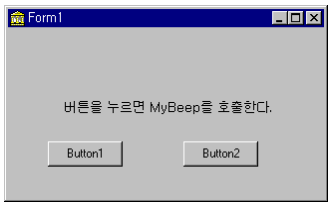
□묵시적(Implicit) 연결:함수가 정의되어 있는 DLL 의 이름과 함수의 형태를 밝히고 본문에서는 함수명으로만 호출하는 방법이다. 실행 파일이 메모리에 로드될 때 DLL 이 같이 로드되며 실행 파일이 종료될 때 DLL 도 같이 종료된다. 실행 파일이 메모리에 있는 동안은 DLL 이 항상 메모리에 있으므로 호출 속도가 빠르지만 메모리를 많이 소모한다는 단점이 있다.

□명시적(Explicit) 연결: 함수를 사용할 때 DLL 을 읽어와 사용하고 사용이 끝난 후 DLL 을 메모리에서 해제하는 방법이다. 호출 속도가 느리기는 하지만 꼭 필요할 때만 DLL 을 사용할 수 있으므로 메모리를 낭비하지 않고 경우에 따라 DLL 을 교체할 수 있다는 장점이 있다.

UseDll 예제에서 보인 DLL 호출 방법은 external 지시자로 DLL 의 이름을 밝혀주고 호출할 때는 함수명으로만 호출하는 묵시적 호출 방법이다. 명시적 연결을 할 때는 다음 세 함수를 사용한다. Win32 API 함수이기 때문에 C 형식으로 원형이 정의되어 있다.

```
HINSTANCE LoadLibrary(LPCTSTR lpLibFileName);
FARPROC GetProcAddress(HMODULE hModule,LPCSTR lpProcName);
BOOL FreeLibrary(HMODULE hLibModule);
```

LoadLibrary 로 DLL 을 메모리로 읽어오고 GetProcAddress 로 DLL 내의 함수번지를 구한다. 그리고 그 번지를 직접 호출하여 함수의 기능을 수행하며 FreeLibrary 로 DLL 을 해제한다. 명시적 연결로 mybeep 함수를 호출해 보기 위해 폼에 버튼을 하나 더 배치하도록 하자.




Button2 의 OnClick 이벤트 핸들러에 명시적 호출 코드를 작성한다.

```
procedure TForm1.Button2Click(Sender: TObject);
type
 TFunc=procedure;StdCall;
var
```

```
  Func:TFunc;
  hInst:THandle;
begin
  hInst:=LoadLibrary('mydll.dll');
  @Func:=GetProcAddress(hInst, 'mybeep');
  Func();
  FreeLibrary(hInst);
end;
```

프로시저 타입 TFunc 를 정의하고 이 타입의 변수 Func 를 선언한다. LoadLibrary 로 mydll.dll 을 읽어와 이 DLL 의 mybeep 함수 번지를 구해 Func 에 대입한다. 함수를 호출할 때는 Func 변수를 바로 사용하면 된다. 사용이 끝난 후에는 FreeLibrary 로 DLL 을 해제해 주었다. 실행 결과는 묵시적으로 연결한 것과 완전히 동일하지만 명시적 연결은 묵시적 연결과는 다른 여러 가지 차이점이 있다.

우선 DLL 을 명시적으로 연결하면 프로그램이 실행될 때 DLL 이 당장 필요한 것은 아니므로 DLL 이 없어도 프로그램은 일단 실행될 수 있다는 점이 다르며 DLL 의 이름이나 함수의 이름을 문자열로 지정하기 때문에 실행중에 조건에 따라 DLL 이나 함수를 교체할 수 있다는 장점도 있다. 그래서 명시적 연결을 사용하면 호환성이 있는 DLL 을 상황에 따라 바꿔가며 사용할 수 있는데 이 기법을 사용하면 마치 윈도우즈가 비디오 드라이버를 교체하는 것과 같은 효과를 흉내낼 수 있게 된다.

단 명시적 연결을 할 때는 DLL 이 제대로 있는지 항상 에러 점검을 해 보도록 해야 한다. DLL 이 분리된 파일로 존재하기 때문에 항상 LoadLibrary 함수가 성공한다고 보장할 수 없기 때문이다.

# 마. 폼 엑스포트하기

DLL이 클라이언트 프로그램에게 공개하는 것은 일반적으로 함수이지만 폼을 공개할 수도 있다. 즉 폼을 DLL로 만들어 놓고 이 폼을 원하는 프로그램에서는 DLL의 함수를 호출함으로써 마치 자신의 폼인 것처럼 사용할 수 있는 것이다. 폼을 DLL로 만드는 방법은 폼을 생성하고 사용, 파괴하는 코드를 하나의 함수(인터페이스 함수)안에 모두 집어 넣고 이 함수를 밖으로 공개하는 원론적인 방법을 사용한다. 그런데 제작 절차가 그리 간단하지 않으므로 다음 단계를 따라 만들어 보도록 하되 큰 의미가 있는 예제는 아니므로 이 기능에 별 관심이 없는

사람은 건너 뛰어도 좋다.

12jang\FormDll
FormDll

우선 새 프로젝트를 시작한 후 엑스포트하고자 하는 폼을 먼저 만든다. 폼에 에디트 하나를 배치하고 이 에디트에 다음과 같이 코드를 작성해 보자. 되도록 간단한 기능만 만들기로 한다.

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Caption:=Edit1.Text;
end;
```

에디트에 입력된 문자열을 폼의 캡션으로 복사해 보았다. 일단 DLL을 만들려면 고정된 디렉토리가 필요하므로 이 프로젝트를 FormDll 디렉토리에 FormDll이라는 이름의 프로젝트로 저장한다. 그리고 컴파일, 실행해 보면 에디트 하나가 있는 폼이 실행될 것이다. 이 폼을 DLL안에 넣어볼 것이되 지금 만든 프로젝트는 실행 파일 프로젝트이므로 이대로는 DLL을 만들 수 없으며 인터페이스 함수를 만든 후 DLL에서 이 함수를 엑스포트하도록 해야 한다. 일단 OutForm이라는 인터페이스 함수를 다음과 같이 작성한다.

```
var
  Form1: TForm1;

procedure OutForm;

implementation

{$R *.DFM}

procedure OutForm;
var
  DllForm:TForm1;
begin
  DllForm:=TForm1.Create(Application);
  DllForm.ShowModal;
  DllForm.Free;
end;
```

interface부에 이 프로시저를 선언하여 외부 모듈로 공개하도록 하였으며 implementation부에서 이 프로시저의 코드를 작성하였다. OutForm에서는

TForm1형의 변수를 선언하고 폼을 생성한 후 ShowModal 메소드로 폼을 실행하였다. 실행이 끝난 후는 Free메소드로 폼을 파괴해 준다. DLL에서 이 함수를 가져다 쓸 수 있도록 예제를 다시 컴파일해 두도록 하자.

12jang\FormDll
CallForm

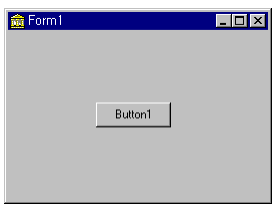
이제 이 폼을 DLL로 만들어 보자. File/New/DLL을 선택하여 새 DLL 프로젝트를 시작한 후 프로젝트 소스를 다음과 같이 수정한다.

```
library Project1;

uses
  FormDll_f in 'FormDll_f.pas';

exports
  OutForm;

begin
end.
```

FormDll_f.pas에 있는 FormDll_f 유닛을 uses절에 적어주면 이 유닛에 있는 TForm1 오브젝트를 사용할 수 있게 된다. 그리고 exports절에 OutForm 인터페이스 함수를 적어 주어 이 함수를 외부로 공개하였다. 이 프로젝트를 CallForm.dpr로 저장하면 첫 줄은 library CallForm;으로 변경될 것이다. 저장하는 위치는 물론 FormDll 프로젝트와 같은 디렉토리여야 한다. 이제 Project/Build 항목을 선택하여 컴파일하면 같은 디렉토리에 CallForm.dll 파일이 생성될 것이며 이 DLL안에 TForm1 폼과 OutForm 함수가 포함되게 된다. DLL안에 폼을 집어 넣은 것이다.

12jang\FormDll
UseFormDll

과연 제대로 만들어졌는지 테스트 프로그램을 작성해 보자. File/New Application으로 새 프로젝트를 시작한 후 폼에 버튼 하나만 배치한다. 이 버튼을 누르면 DLL에 있는 폼을 호출하도록 해 볼 것이다.

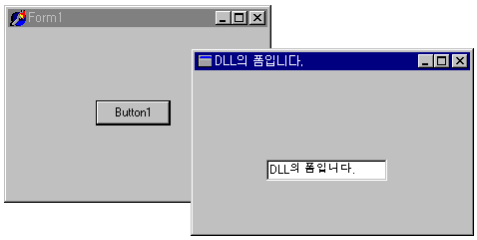


버튼의 OnClick 이벤트 핸들러에서 OutForm 함수를 호출하도록 하였으며 그 전에 OutForm 함수가 DLL에 있는 함수임을 밝혀 주었다.

```
procedure OutForm;external 'CallForm.dll';

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

이 프로젝트를 CallForm.dll 파일이 있는 디렉토리에 UseCallForm.dpr로 저장한 후 실행시켜 보자. 버튼을 누르면 Formdll 프로젝트에 있는 폼이 나타날 것이다.

그림
DLL 에 정의된 폼의 호출




폼을 DLL로 엑스포트하는 간단한 실습을 해 보았는데 무려 세 개나 되는 프로젝트를 만들었다. 다른 툴로는 구현하기 힘든 VCL 기능이 있다면 이런 식으로 DLL을 만들어 사용하면 된다.

여기서 소개한 이론 외에도 DLL을 통한 다국어 버전 작성법, 컴포넌트를 DLL로 만드는 방법, DLL에 아이콘을 왕창 집어 넣는 방법도 있지만 앞에서도 말했듯이 델파이에서 DLL은 그리 권할만한 프로그래밍 방법이 아니므로 지나치게 시간을 투자하지는 말기 바란다.

# 12-8 인쇄

윈도우즈는 화면이나 프린터로 출력을 할 때 곧바로 하드웨어를 액세스하지 않고 GDI(Graphic Device Interface)를 통해 출력한다. 그래서 화면으로 보내는 출력이나 프린터로 보내는 출력이나 모두 GDI로 가게 되며 보내는 방법이 유사할 수밖에 없다. 덕분에 화면에 출력하는 방법만 알고 있으면 프린터로 인쇄하는 방법 정도는 아주 쉽게 터득할 수 있다.

## 가. Printer 오브젝트

델파이는 인쇄에 관한 모든 기능을 Printer 오브젝트로 제공하고 있다. 이 오브젝트는 Canvas를 가지므로 이미지의 Canvas나 폼의 Canvas에 문자열, 그래픽을 인쇄하는 방법과 동일한 방법을 사용하여 프린터로 출력을 보낼 수 있다. 이미지 컴포넌트로 문자열을 출력하기 위해 다음과 같이 코드를 작성해 보자.

```
Image1.Canvas.TextOut(....);  {이미지로 출력}
```

TextOut 메소드를 사용하여 이미지의 표면에 해당하는 Canvas에 문자열을 출력한다. 프린터도 이와 동일한 방법으로 인쇄를 한다. 단 프린터는 인쇄를 하기 전에 인쇄를 준비시키는 BeginDoc 메소드를 호출해 주어야 하며 인쇄가 끝난 후에는 EndDoc 메소드를 호출하여 인쇄 종료를 알려주어야 한다. 다음은 프린터로 인쇄를 하는 코드이며 프린터의 표면(곧 용지의 표면)에 문자열을 출력할 것이다.

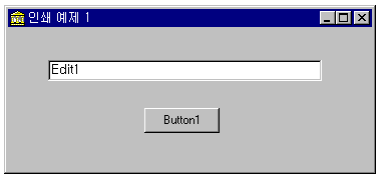
```
Printer.BeginDoc;
Printer.Canvas.TextOut(....);  {프린터로 출력}
Printer.EndDoc;
```

똑같은 TextOut 메소드를 사용하지만 이 메소드를 호출하는 오브젝트에 따라 출력을 어디로 보낼 것인가가 결정된다. Printer 오브젝트는 말 그대로 오브젝트이지만 컴포넌트는 아니다. 그래서 컴포넌트 팔레트에는 없으며 디자인중에도 사용할 수 없다. 오직 코드를 통해 실행중에만 사용할 수 있다. Printer 오브젝트는 Printers 유닛에 선언되어 있으므로 이 오브젝트를 사용하고자 한다면 uses절에 Printers를 기입해 주는 것을 잊지 말아야 한다.

# 나. 문자열 인쇄

12jang
print1

아무래도 그래픽보다는 문자열이 더 취급하기가 쉬우므로 우선 문자열부터 인쇄를 해 보자. 인쇄할 문자열을 입력받을 에디트 박스 하나와 인쇄 명령을 내릴 버튼 하나를 폼에 배치한다.




폼의 캡션을 변경하는 것 외에는 속성을 조절할 필요가 없다. 버튼의 OnClick 이벤트는 다음과 같이 작성되어 있다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Printer.BeginDoc; {인쇄 시작}
  Printer.Canvas.TextOut(10,10,Edit1.Text);
  Printer.EndDoc;  {인쇄 끝}
end;
```

BeginDoc으로 인쇄 시작을 명령하고 TextOut로 Edit1.Text에 입력된 문자열을 보낸 후 EndDoc으로 인쇄를 종료한다. 프린터로 출력을 보내는 명령들을 BeginDoc과 EndDoc 사이에 배치하면 된다. 프린터는 자신의 캔버스로 보내진 문자열(또는 그래픽)을 버퍼에 차곡 차곡 모아 두었다가 EndDoc이 호출될 때 비로소 실제 출력을 한다. 여기서 사용한 10,10의 좌표는 인쇄 용지의 좌상단을 기준으로 10,10픽셀만큼 떨어진 위치를 지정한다. 대충 아무 곳이나 정한 좌표이지 특별한 의미가 있는 것은 아니다. 버튼의 OnClick 이벤트에서 인쇄에 관한 모든 처리를 하고 있다. 단, 이 이벤트를 작성해 주는 일 외에도 uses절에 Printers 유닛을 기입해 주는 것을 잊지 말아야 한다.

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

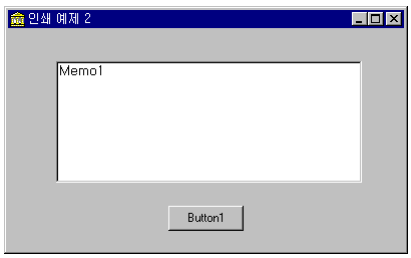
Printer 오브젝트는 컴포넌트가 아니기 때문에 델파이가 해당 유닛을 자동으로 기입해 주지 않는다. 만약 uses절에 유닛 선언을 생략하면 Printer라는 명칭

자체가 없는 것으로 취급되므로 에러 메시지가 발생할 것이다. 앞으로의 예제에서도 uses절에 Printers 유닛을 기입해 주는 것은 사용자가 직접 해야 한다.

출력 결과는 여러분의 프린터로 나와 있을 것이다. 물론 이 예제가 제대로 실행되려면 윈도우즈에 프린터가 설치되어 있어야 하고 프린터가 켜져 있어야 한다. 출력되는 문자열의 폰트는 윈도우즈가 사용하는 시스템 폰트이다. 폰트의 글꼴이나 모양을 바꾸고 싶다면 Canvas 오브젝트의 Font 속성을 원하는대로 변경해 주면 된다.

12jang
print2

이번에는 여러 줄의 문자열을 프린터로 출력하도록 조금만 예제를 변경해 보자. 여러 줄을 입력받아야 하므로 에디트 컴포넌트 대신 메모 컴포넌트를 배치하였다.




그리고 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성하며 uses절에 Printers 유닛을 적어준다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  Printer.BeginDoc;  {인쇄 시작}
  {메모의 각 줄을 읽어 프린터로 보낸다.}
  for i:=0 to memo1.Lines.Count-1 do
    Printer.Canvas.TextOut(10,10+i*80,Memo1.Lines[i]);
  Printer.EndDoc;    {인쇄 끝}
end;
```

메모의 첫 줄부터 끝 줄까지 읽어 프린터로 보내주기를 반복하기만 하면 된다. 메모에 입력된 문자열의 개수는 Count 속성으로 알 수 있으며 첫 줄의 번호가 0번이므로 끝줄의 인덱스는 Count보다 하나가 작다. TextOut문의 Y좌표에 곱해지는 80이라는 상수는 각 줄을 출력할 때마다 띄워야 할 줄간이 되는 셈이다. 그러나 이렇게 단순한 곱셈에 의해 줄간을 띄우면 문제가 발생한다. 출력할 문자열의 정확한 높이는 윈도우즈의 시스템 폰트에 따라 달라지며 프린터의 해

상도에 따라 달라지기 때문이다. 이런 상황을 고려하여 적절한 줄간을 계산해 내기란 쉬운 일이 아니다.

12jang
print3

이 문제는 근본적으로 좌표를 사용하여 문자열을 출력하는 TextOut이라는 함수에 있다. 즉 문제 해결을 위해 좌표를 프로그램에서 지정하지 않고 프린터가 좌표를 계산하도록 맡겨두는 것이 더 좋다는 얘기다. 버튼의 OnClick 이벤트를 다음과 같이 수정하면 문제를 해결할 수 있다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
  PFH:TextFile;
begin
  AssignPrn(PFH);
  Rewrite(PFH);
  for i:=0 to memo1.Lines.Count-1 do
    Writeln(PFH,Memo1.Lines[i]);
  CloseFile(PFH);
end;
```

이 예제는 아주 독특한 방법으로 인쇄를 하고 있다. 이 예제를 이해하기 전에 먼저 프린터는 쓰기 전용의 텍스트 파일이라는 사실부터 이해해야 한다. 도스에서 다음과 같은 명령을 사용해 본 경험이 있을 것이다.

```
C:₩>copy readme.txt prn
C:₩>type readme.txt > prn
```

PRN이란 도스가 사용하는 장치 파일이며 프린터 자체를 의미한다. readme.txt라는 텍스트 파일을 PRN이라는 파일로 복사하면, 즉 프린터로 readme.txt를 출력하면 결과는 프린터의 용지에 인쇄되어 나타난다. 왜 프린터를 파일과 동일하게 취급하는가 하면 파일에 문자열을 출력하는 방법이 프린터로 문자열을 출력하는 방법과 동일하기 때문이다.

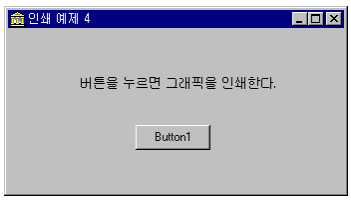
위 예제는 이런 원리를 사용하여 프린터를 파일인 것처럼 취급하며 출력 절차도 파일로 출력하는 방법과 동일하다. 단 파일 변수 PFH에 파일을 할당할 때 Assign 프로시저를 쓰지 않고 AssignPrn 프로시저를 사용한다는 점이 다르다. Rewrite로 파일을 쓰기 모드로 연 후 텍스트를 인쇄한다. 인쇄에 사용하는 프로시저는 텍스트 파일 출력에 사용하는 Writeln 프로시저이다. Writeln 프로시저를 사용하면 줄간을 자동으로 맞추어 주므로 폰트나 프린터의 해상도에 상관없이 적당한 줄간이 띄워질 것이고 프로그램에서는 인쇄할 문자열을 프린터로 보

내주기만 하면 된다.

## 다. 그래픽 인쇄

12jang
print4

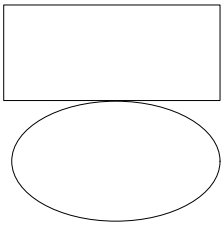
그래픽은 텍스트와는 달라서 인쇄하기가 무척 어려울 것 같아 보이지만 Printer 오브젝트를 사용하면 문자열을 인쇄하는 것처럼 쉽다. 직접 프린터를 제어해야 하는 도스에서의 그래픽 인쇄는 정말이지 아무나 못할 작업이었지만 델파이에서는 Printer 오브젝트만 통하면 화면에 그림을 그리듯 프린터에 그림을 그릴 수 있다. 예제를 보자. 폼에는 그래픽 인쇄를 명령할 버튼만 배치하였다.




버튼의 OnClick 이벤트는 다음과 같다.

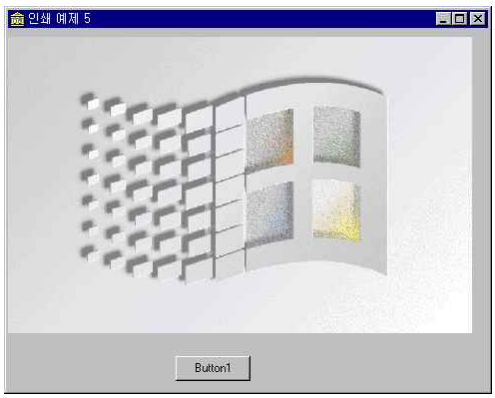
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Printer.BeginDoc;
  with Printer.Canvas do
    begin
    Rectangle(100,100,500,300);
    Ellipse(100,400,500,800);
    end;
  Printer.EndDoc;
end;
```

보다시피 화면에 그림을 그리는 방법과 전혀 틀리지 않아서 더 설명할 필요가 없을 정도다. 만약 위의 코드를 이해하지 못한다면 8장을 제대로 읽지 않았다는 증거이므로 다시 8장을 읽어보기 바란다. 용지에 인쇄되는 결과는 다음과 같다.



12jang
print5

두 개의 메소드를 사용하여 직사각형과 원을 그렸다. 선이나 면의 속성을 바꾸고 싶다면 Canvas의 Brush, Pen 오브젝트의 속성을 변경해 주면 된다. 다음은 비트맵을 출력하는 예제이다.




출력할 비트맵을 이미지 컴포넌트를 사용하여 미리 폼에 배치해 두었으며 버튼을 클릭하면 이 비트맵을 프린터로 보낸다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with Printer do
  begin
  BeginDoc;
  Canvas.Draw(10,10,Form1.Image1.Picture.Graphic);
  EndDoc;
  end;
end;
```

예제의 핵심은 비트맵을 출력하기 위해서는 Draw 메소드를 사용해야 한다는 것이다. 세 번째 인수로 출력하고자 하는 비트맵이나 메타 파일, 아이콘을 지정해 주면 된다. 비트맵을 확대하거나 축소해서 출력하고자 할 때는 Draw 대신 StretchDraw 메소드를 사용하면 된다. 이 외에 폼 자체를 인쇄하는 Print라는

폼의 메소드도 있는데 이 메소드는 폼을 비트맵으로 만들어 그대로 프린터로 출력해 주는 대단한 기능을 가지고 있지만 사실 폼을 인쇄할 경우란 극히 드물기때문에 실용성이 없다.

# 라. 인쇄 대화상자

앞에서 만들어 본 예제들은 버튼의 OnClick 이벤트 핸들러에 인쇄 코드를 작성했지만 제대로 인쇄 루틴을 만든다면 인쇄 대화상자를 보여주고 정말로 인쇄를 할 것인지, 인쇄 범위를 어디까지 설정할 것인지를 물어 보는 것이 원칙이다. 이런 인쇄 대화상자는 직접 만들어도 되겠지만 델파이에서 컴포넌트로 제공하므로 표준적인 대화상자를 사용하는 것이 좋다. 컴포넌트 팔레트의 Dialogs 페이지를 보면 인쇄 대화상자 컴포넌트가 두 개 있다. 하나는  요렇게 생겼고 하나는  이렇게 생겼다.

우선 PrintDialog 대화상자를 보자. 사용하는 방법은 파일 열기 대화상자나 기타 다른 공통 대화상자를 사용하는 방법과 동일하다.

```
if PrintDialog1.Execute then
  인쇄한다;
```

Execute 메소드로 대화상자를 실행시키는데 사용자가 인쇄 취소를 할 수도 있으므로 인쇄를 하기 전에 리턴값을 반드시 살펴보아야 한다. Execute 메소드에 의해 다음과 같은 인쇄 대화상자가 나타나며 대화상자의 구체적인 모양은 설치된 프린터 드라이버에 따라 달라진다.

**그림**
인쇄 공통 대화상자


인쇄 범위, 인쇄 매수, 인쇄할 페이지 등의 옵션이 있으며 사용자가 선택한 옵션은 다음과 같은 속성으로 전달된다.

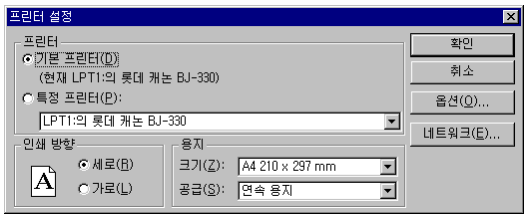
| 속성 | 옵션 |
|---|---|

| | |
|---|---|
| PrintRange | 인쇄 범위 |
| FromPage | 인쇄 시작 페이지 |
| ToPage | 인쇄 끝 페이지 |
| Copies | 인쇄 매수 |

인쇄 대화상자는 사용자로부터 이 옵션들을 입력받아 주기만 할 뿐이지 실제로 옵션에 맞게 인쇄를 해 주는 것은 아니므로 오해하지 말기 바란다. 이 옵션을 적절하게 사용하여 인쇄하는 것은 프로그래머가 직접 해야 할 일이다. PrinterSetupDialog는 다음과 같은 프린터 설정 대화상자를 보여준다.

**그림**

프린터 설정 대화상자




이 대화상자에서 하는 일은 프린터를 교체하거나 용지의 특성, 인쇄 농도 등 프린터에 관한 설정을 하는 것이다. 대화상자 내에 프린터를 선택하거나 옵션을 변경하는 코드가 모두 들어 있으므로 이 대화상자에서 값을 돌려받을 필요는 없다. 그래서 조건 점검도 필요없이 다음과 같이 호출만 해 주면 된다.

```
PrinterSetupDialog1.Execute;
```

프린터는 윈도우즈에 의해 관리되며 프린터 설정 상태도 윈도우즈가 알아서 변경하므로 프로그램에서 프린터 설정에 관해 전혀 신경쓸 필요가 없다. 대화상자를 호출하는 코드를 Print Setup... 등의 메뉴 항목에 연결시켜 두기만 하면 된다.

이상으로 델파이에서 인쇄하는 법을 알아 보았다. 여기서 논한 내용 외에도 제대로 인쇄를 하려면 페이지의 크기와 문서의 크기를 잘 계산하여 페이지 당 줄수, 적당한 줄간을 계산해야 한다. 또한 적당한 위치에서 새로운 페이지로 용지를 교체해 주어야 하며 미리 보기 기능도 제공하면 좋을 것이다. 그러나 이는 인쇄에 관한 기술이라기보다는 내부적인 계산에 관한 기술이므로 여기서는 논하지 않겠다.

# 12-9 멀티 스레드

## 가. 스레드

잘 알다시피 윈도우즈 NT와 윈 98은 멀티 태스킹을 지원하는 운영체제이며 그래서 동시에 여러 개의 프로그램을 실행시킬 수 있어 무척 편리하다. Win32 환경에 와서는 멀티 태스킹뿐만 아니라 멀티 스레드라는 개념이 하나 더 추가되어 좀 더 복잡한 형태의 다중 작업을 지원하게 되었다. 스레드(Thread)란 프로세스내의 실행 경로를 말하는데 이런 스레드가 한 프로세서에 여러 개 있는 구조를 멀티 스레드(Multi Thread)라고 한다. 보통 한 프로그램에는 하나의 주 스레드만 있지만 두 개 이상의 스레드를 가진 프로그램에서는 여러 가지 작업을 동시에 수행할 수 있다.

잘 이해가 가지 않는다면 예를 들어 보자. 워드 프로세서의 경우를 보면 문자열을 정렬하는 작업과 페이지를 계산하는 작업과 사용자로부터 입력을 받아들이는 작업이 동시에 이루어져야 하는데 이럴 때 각 작업을 스레드로 만들어 놓으면 세 가지 작업이 동시에 이루어질 수 있다. 물론 이 정도의 다중 처리는 타이머나 Idle 타임을 사용하는 전통적인 방법으로도 가능하기는 하지만 Win32 에서 제공하는 멀티 스레드를 이용하면 운영체제 차원에서 CPU 시간을 잘게 쪼개어 각 스레드에 분배해 주므로 훨씬 더 안정적이고 매끄러운 다중 처리를 할 수 있다. 스레드에는 두 가지 종류가 있다.

- 작업 스레드(Worker Thread) :특정한 작업만을 분담하는 스레드이다. 예를 들어 복잡한 수학계산이나 그래픽 처리 등만을 전문적으로 수행하며 작업을 다 하면 스레드는 종료되는 것이 보통이다. 정해진 작업을 코드 순서대로 실행할 뿐이므로 메시지 루프가 불필요하다.
- 유저 인터페이스 스레드(User Interface Thread) : 사용자의 명령을 받아들일 수 있는 스레드이다. 사용자의 요구를 분석한 후 요구대로 실행해야 하므로 메시지 루프가 필요하며 별도의 종료 명령을 내려야만 종료된다.

윈도우즈 응용 프로그램은 보통 하나의 유저 인터페이스 스레드를 메인 스레드로 가지며 필요에 따라 작업 스레드를 추가로 만들 수 있다. 하지만 유저 인터

페이스 스레드를 추가로 더 만드는 경우는 거의 없다. 일부러 만들려면 가능은 하지만 오히려 더 복잡한 문제가 많이 발생할 것이다.

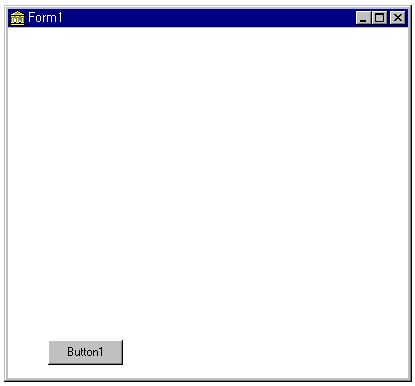
# 나. 단일 스레드

12jang
Thread

여기서는 스레드의 필요성과 효용성을 테스트 해 보기 위해 두 개의 간단한 예제를 만들어 보고 스레드를 쓸 경우와 그렇지 않은 경우를 비교해 본다. 프로그램 실행중에 시간이 아주 오래 걸리는 작업을 해야 할 필요가 있다고 하자. 예를 들어 비트맵에 미리 그림을 그려 놓는다거나 긴 문서의 페이지를 정렬한다거나 하는 등의 작업이 있을 것이다. 이런 작업을 하는 동안에 프로그램은 잠시 중지된 상태가 될 것이며 그동안 사용자는 해당 작업이 끝날 때까지 기다려야만 하는 불편이 있다.

이럴 때 시간을 소모하는 작업을 별도의 스레드로 분리해 두고 따로 실행되도록 한다면 사용자의 대기 시간을 없앨 수 있을 것이다. 여기서 시간이 오래 걸리는 정도는 길게는 10 초나 20 초 정도가 되겠지만 짧게는 1 초 정도가 될 수도 있다. 컴퓨터 사용자는 프로그램이 곧바로 응답하기를 기대하기 때문에 때로는 1~2 초의 시간도 굉장히 길게 느껴질 수도 있다. 또한 아무리 짧은 시간이라도 프로그램이 실행중에 뚝뚝 끊어지는 느낌이 있어 깔끔하지 못하다는 느낌을 받게 될 것이다.

여기서 만들 예제는 화면상에 무작위로 점을 10000 개 찍는데 이는 실전 프로그램에서 "시간이 오래 걸리는 작업"에 비유될 수 있다. 이렇게 시간이 오래 걸리는 작업을 스레드로 분리하는 방법과 분리했을 때의 효과를 보자는 것이다. 새 프로젝트를 시작하고 폼의 아래쪽에 버튼 하나만 배치한다. 폼의 폭(ClientWidth)은 400 으로 설정하고 높이(ClientHeight)는 350 으로 설정하도록 하자. 점을 찍을거니까 잘 보이도록 바탕색은 흰색으로 바꾸는 것이 좋을 것 같다.




버튼을 누르면 점 만개를 찍는 코드를 다음과 같이 작성한다. 시간을 좀 끌기 위해 폼의 캡션도 바꾸도록 했다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i:Integer;
begin
 for i:=0 to 10000 do
  begin
  Caption:=IntToStr(i);
  Canvas.Pixels[random(400),random(300)]:=
   RGB(random(256),random(256),random(256));
  end;
end;
```

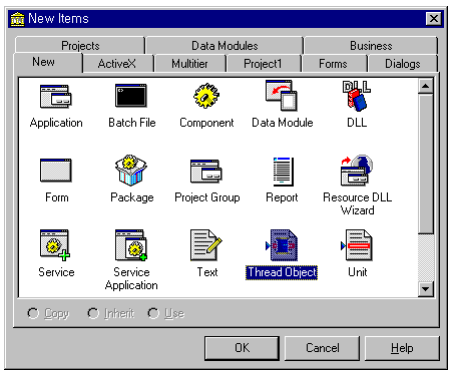
이제 예제를 실행시켜 보자. 웬만큼 컴퓨터가 빠르지 않고서는 이 작업을 다 하는데 꽤 시간이 걸릴 것이다. 필자의 시스템에서는 정확하게 8.33 초가 소요되었다. 어차피 작업량이 많은 일을 시켰으므로 시간이 걸리는 것은 어쩔 수 없다 치더라도 이런 식으로 프로그램을 작성하는 것은 별로 권할만하지 못하다. 왜냐하면 한 프로시저에서 너무 많은 시간을 잡아 먹어버려 그 동안에는 일체의 다른 동작을 할 수 없기 때문이다. 버튼을 눌러놓고 타이틀 바를 드래그해 보면 전혀 반응이 없을뿐만 아니라 시스템 메뉴를 눌러도 아무 동작도 하지 않는다. 왜냐하면 CPU 가 점찍는데만 몰두하고 있기 때문에 그 외의 다른 일은 할 시간이 없기 때문이다.

8 초 정도니까 이렇게 해도 별 상관이 없을지 모르겠지만 만약 10 분 정도 걸리는 작업이라면 이런 식이어서는 곤란하다. 그래서 이런 작업은 한꺼번에 하지 않고 한가할 때 틈틈이 나누어서 하는 방법을 사용하곤 한다. 틈틈이 나누어 하는 방법에는 타이머나 OnIdle 이벤트를 사용하는 방법이 있는데 옛날에는 실제

로 이런 방법이 사용되었으며 현재에도 이 방법이 멀티 스레드보다 더 적합한 경우가 있다. Win32 에서는 이보다 스레드를 사용하는 것이 더 권장되고 있다.

# 다. 멀티 스레드

그럼 이제 똑같은 예제를 스레드를 사용해서 만들어 보고 차이점을 비교해 보도록 하자. 앞에서 만들었던 예제를 계속 이어서 만들어 본다. 프로젝트에 스레드를 추가하기 위해 File/New 메뉴 항목을 선택하고 New Items 대화상자에서 Thread Object 를 선택한다.




그러면 만들고자 하는 스레드 오브젝트의 이름을 물어올 것이다. 여기에 적당한 클래스 이름을 주면 되는데 MyThread 라고 입력해 보자.


새로운 스레드 오브젝트가 생성되었으며 이 스레드를 정의하는 유닛을 만들어 프로젝트에 포함시켜 준다. 이 유닛은 스레드 오브젝트만 정의하며 폼은 가지지 않는다.

```
unit Unit2;

interface

uses
```

```
 Classes;

type
 MyThread = class(TThread)
 private
  { Private declarations }
 protected
  procedure Execute; override;
 end;

implementation

{ Important: Methods and properties of objects in VCL can only be used in a
 method called using Synchronize, for example,

    Synchronize(UpdateCaption);

 and UpdateCaption could look like,

  procedure MyThread.UpdateCaption;
  begin
   Form1.Caption := 'Updated in a thread';
  end; }

{ MyThread }

procedure MyThread.Execute;
begin
 { Place thread code here }
end;

end.
```

유닛을 정의하는 소스인데 이 유닛에는 TThread 클래스로부터 파생된 MyThread 클래스가 정의되어 있다. 별다른 멤버는 없고 Execute 가상 함수만 재정의되어 있다. 중앙의 주석을 읽어보면 VCL 오브젝트는 Synchronize 프로시저 내에서만 쓸 수 있다고 되어 있는데 이 의미는 좀 있다가 따로 자세히 알아보자. 이 주석은 읽어본 후 지워도 상관없다.

일단 두 개의 유닛끼리 서로 참조하기 위해 이름이 필요하므로 이 프로젝트를 저장한다. 유닛은 각각 Thread_f.pas, Thread_f2.pas 로 저장하고 프로젝트명은 Thread.dpr 이라는 이름을 준다.

이제 소스를 뜯어 고쳐 스레드를 사용한 예제를 만들어 보자. 우선 폼에 Button2 버튼을 배치하고 두 유닛 파일의 소스를 다음과 같이 수정한다. 여기저기 많이 손 봐야 하는데 좀 복잡하므로 일단 굵은 글꼴로 된 부분을 직접 입력하도록 한다. 우선 메인 폼의 소스를 보자.

```
unit Thread_f;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 StdCtrls, Thread_f2;

type
 TForm1 = class(TForm)
  Button1: TButton;
  Button2: TButton;
  procedure Button1Click(Sender: TObject);
  procedure Button2Click(Sender: TObject);
  procedure MyOnTerminate(Sender:TObject);
 private
  { Private declarations }
  MyThr:MyThread;
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
 i:Integer;
begin
 for i:=0 to 10000 do
  begin
  Caption:=IntToStr(i);
  Canvas.Pixels[random(400),random(300)]:=
   RGB(random(256),random(256),random(256));
  end;
```

```
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  MyThr:=MyThread.Create(False);
  MyThr.OnTerminate:=MyOnTerminate;
end;

procedure TForm1.MyOnTerminate;
begin
  MyThr.Free;
end;

end.
```

메인 폼에서 MyThread 오브젝트를 사용하기 위해 이 클래스가 정의되어 있는 Thread_f2 유닛을 uses절에 적어주었다. 폼의 타입 선언부에는 MyThread 클래스형의 오브젝트 MyThr과 이 오브젝트의 이벤트 핸들러로 사용될 MyOnTerminate 프로시저를 선언하고 있다. Button2의 OnClick 이벤트 핸들러에서 새 스레드를 Create 메소드로 생성하였으며 동시에 스레드가 종료될 때의 이벤트 핸들러인 OnTerminate를 지정하였다. Button2를 누르면 스레드가 만들어지고 실행될 것이다. OnTerminate에서는 스레드가 종료될 때 이 오브젝트를 파괴하도록 하였다. 다음은 스레드가 정의된 유닛을 보자.

```
unit Thread_f2;

interface

uses
  Classes, SysUtils, Windows;

type
  MyThread = class(TThread)
  private
    { Private declarations }
  protected
    Count:Integer;
    procedure Execute; override;
    procedure PutDot;
```

```
 end;

implementation
uses
  Thread_f;

procedure MyThread.PutDot;
begin
  Form1.Caption:=IntToStr(Count);
  Form1.Canvas.Pixels[random(400),random(300)]:=
    RGB(random(256),random(256),random(256));
end;

procedure MyThread.Execute;
var
  i:Integer;
begin
  for i:=1 to 10000 do
    begin
    Count:=i;
    Synchronize(PutDot);
    end;
end;

end.
```

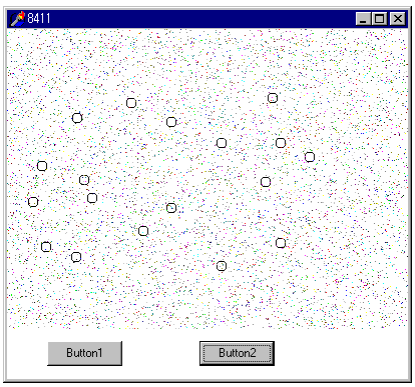
MyThread.Execute 프로시저가 스레드의 시작점(Entry Point)이 되는데 메인 폼에서 스레드를 생성하면 이 프로시저가 호출된다. 여기서 10000개의 점을 찍는 처리를 하고 있다. 내부의 코드는 잠시 후에 분석해 볼 것이다. uses절에는 Windows, SysUtils 유닛을 적어 주었는데 이는 각각 RGB 함수와 IntToStr 프로시저를 사용하기 위해서이다.

이제 예제를 실행하고 Button2를 눌러보면 점을 찍고 있는 동안에도 윈도우를 다른 곳으로 옮길 수 있으며 Button1을 누를 수도 있다. 메인 스레드와 별도의 MyThread가 동시에 실행되고 있으며 CPU는 각 스레드에게 시간을 배분해 주기 때문이다. 과연 다른 동작도 별 무리없이 동작하는지 한 가지 테스트를 더 해 보자. 폼의 OnLButtonDown 메시지 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.Ellipse(X,Y,X+10,y+10);
```

```
end;
```

마우스 왼쪽 버튼을 누르면 그 위치에 조그만 타원을 그리도록 하였다. 스레드가 돌아가고 있는 동안에도 마우스 버튼을 누르면 타원이 잘 그려질 것이다.




# 라. TThread

델파이의 멀티 스레드 지원은 TThread 클래스에 의해 이루어진다. 이 클래스를 직접 사용하지 않아도 Win32 API 함수를 사용하면 스레드를 만들 수 있지만 무척 복잡하다. TThread 클래스에는 스레드의 모든 것이 캡슐화되어 있으므로 이 클래스만 제대로 이해하면 어렵지 않게 멀티 스레드 프로그램을 짤 수 있다.

이 클래스는 순수 가상 함수를 가지고 있는 추상 클래스이기 때문에 직접 사용할 수는 없으며 반드시 자식 클래스를 파생한 후에 파생 클래스를 프로그래밍해야 한다.VCL 소스를 보면 이 클래스는 다음과 같이 정의되어 있다.

```
TThread = class
 private
  FHandle: THandle;
  .....
 protected
  procedure DoTerminate; virtual;
  procedure Execute; virtual; abstract;
  .....
```

보다시피 Execute 메소드가 순수 가상 함수(Pure Virtual Function)로 선언되어 있기 때문에 반드시 파생 클래스에서 이 메소드를 재정의해 주어야 한다. 이 메소드는 실질적으로 스레드의 시작 함수이며 이 메소드에서 어떤 일을 하는

가가 곧 그 스레드가 어떤 작업을 하는지를 결정하게 된다. 이 메소드가 종료되면 스레드도 같이 종료된다. 마치 도스용 C 프로그램의 main 함수와 같다고 할 수 있다.

스레드를 만들 때는 Create 메소드를 사용하는데 이 메소드는 하나의 진위형 인수를 취한다. 이 인수가 False일 경우는 스레드가 생성됨과 동시에 실행되지만 True일 경우는 스레드가 생성되기만 하고 실행되지는 않으며 중지해 있는다. 이렇게 중지된 상태를 서스펜드(Suspend) 상태라고 하는데 이 상태는 Resume 메소드가 호출될 때까지 지속된다. 실행되고 있는 스레드를 잠시 중지하고자 할 때는 Suspend 메소드를 호출해 강제로 서스펜드 상태로 만들어 주면 된다. 스레드를 일단 만들어만 놓고 적당한 시기에 실행시키려면 Create(True)로 생성하며 이 예제의 경우와 같이 만든 후 곧바로 실행하고자 한다면 Create(False)로 생성한다.

TThread의 이벤트로는 유일하게 OnTerminate 가 있는데 이 이벤트는 스레드의 Execute 메소드가 종료되었을 때, 즉 스레드의 실행이 종료되었을 때 보내진다. 이 이벤트의 핸들러에서 스레드를 파괴해 주면 된다. 예제에서는 스레드를 만들 때 이 이벤트의 핸들러를 MyOnTerminate로 지정하여 이 핸들러에서 스레드를 파괴(Free)해 주었다.

TThread의 메소드 중에 가장 이해하기 어려운 것은 Synchronize 메소드이다. 이 메소드는 VCL 오브젝트를 스레드가 단독으로 사용할 수 있도록 해 주는데 조금 더 자세히 설명해 보겠다. 스레드의 메인 메소드는 Execute이며 이 메소드에서 스레드가 하고자 하는 작업을 하면 된다. 그런데 문제는 메인 스레드와 작업 스레드가 동시에 실행되기 때문에 상호 같은 메모리 영역(오브젝트, 변수, 파일 등등)을 참조한다는 점이며 이런 문제로 인해 프로그램이 오동작을 할 수도 있다.

예를 들어 작업 스레드에서 폼의 캡션을 변경하고자 한다. 그런데 우연히 메인 스레드에서도 폼의 캡션을 변경하고자 한다면 이때 문제가 발생한다. 캡션을 변경하는 메소드를 두 코드가 동시에 호출하면 이 메소드가 실행중에 또 호출되는 재진입(Reentrant)이 발생하게 되며 이렇게 되면 캡션 변경 메소드 내에서의 지역 변수가 파괴될 것이다. 그렇게 되면 두 스레드 중 하나는 분명히 제대로 동작할 수 없게 된다. 캡션을 바꾸는 정도야 잘못되도 큰 상관이 없겠지만 전역 변수나 기타 중요한 메모리 영역을 잘못 건드리게 되면 치명적인 결과를 초래하게 될 것이다. 물론 이런 경우는 지극히 확률이 낮지만 그렇다고 해서 발생하지 않는다는 보장이 없다.

그래서 여러 개의 스레드가 실행중일 때는 한쪽이 자원을 액세스하는 동안은 다른 쪽은 그 자원(=VCL 오브젝트)을 건드리지 못하게 해야 하며 자신이 자원

을 다 쓰고 난 후에는 다른 쪽이 쓸 수 있도록 해 주어야 한다. 몇몇 VCL 오브젝트에는 이런 용도로 사용되는 Lock, UnLock 메소드가 존재하는데 Execute 메소드는 자원을 쓰기 전에 Lock하고 다 쓴 후에 UnLock 해 주어야 한다.

```
procedure MyThread.Execute;
begin
 Form1.Canvas.Lock;   // 지금부터 나 혼자 쓸래
 Canvas 사용;         // 실컷 쓴다.
 Form1.Canvas.UnLock;  // 이제 너 차례야.
end;
```

그런데 매번 VCL 오브젝트를 사용할 때마다 이런 처리를 하자면 불편할 것이다. 그래서 이런 처리를 대신 해주는 메소드를 만들었는데 그것이 바로 Synchronize 메소드이다. 이 메소드는 VCL 오브젝트에 대한 배타적인 사용권을 먼저 확보한 후에 VCL 오브젝트를 사용하는 메소드를 호출하고 마지막으로 VCL 오브젝트를 풀어준다. 인수로는 메소드의 포인터를 취하는데 이 메소드내에서는 VCL 오브젝트를 마음대로 접근해도 된다. 예제의 소스를 다시 보도록 하자.

```
procedure MyThread.PutDot;
begin
 Form1.Caption:=IntToStr(Count);
 Form1.Canvas.Pixels[random(400),random(300)]:=
   RGB(random(256),random(256),random(256));
end;

procedure MyThread.Execute;
var
 i:Integer;
begin
 for i:=1 to 10000 do
   begin
   Count:=i;
   Synchronize(PutDot);
   end;
end;
```

Execute 메소드에서는 i루프를 10000번 돌면서 Synchronize 메소드를 호출한다. 이때 Synchronize 메소드의 인수는 PutDot라는 또 다른 메소드이며 이 메소드는 바로 위에 정의되어 있다. Synchronize 메소드가 VCL 오브젝트를 먼저 Lock한 후에 PutDot를 호출해 주므로 PutDot에서는 폼의 캡션도 바꿀 수 있고 캔버스에 무엇인가를 그릴 수도 있다. PutDot가 리턴되면 Synchronize는

VCL 오브젝트를 UnLock하고 리턴하며 Execute는 루프를 계속 돌아갈 것이다. 여기서 또 한 가지 Count라는 필드는 왜 필요할까라는 의문이 생길 것이다. Synchronize 메소드가 인수로 취하는 메소드는 인수를 가지지 않는다. 그래서 PutDot에게 캡션을 어떻게 바꿀 것인가에 대한 정보를 인수로 제공해 줄 수가 없으며 그래서 스레드 오브젝트의 필드를 사용하는 것이다. 필드 대신 전역 변수를 사용해도 상관없다.

TThread 클래스도 몇 가지 속성을 가지고 있는데 그 중 Priority 속성에 대해서만 알아보자. 여러 개의 스레드가 실행중일 때 CPU는 자신의 실행시간을 잘게 쪼개 각 스레드를 교대로 실행한다. 이때 스레드에게 얼마나 많은 시간을 배분할 것인가를 우선 순위라고 하는데 Priority 속성이 이 우선 순위를 지정한다. 스레드의 용도에 따라 우선 순위를 조정해 줄 수 있다.

| 값 | 설명 |
|---|---|
| tpidle | 시스템이 아무 할 일이 없을 때만 실행되는 최하위 우선 순위이다. |
| tpLowest | 매우 낮음 |
| tpLower | 낮음 |
| tpNormal | 보통 |
| tpHigher | 높은 |
| tpHighest | 매우 높음 |
| tpTimeCritical | 가장 높음. 되도록 빨리 처리되어야 할 작업 우선 순위이다. |

우선 순위가 높을수록 스레드는 더 많은 CPU 시간을 받게 되고 따라서 그만큼 더 빨리 실행된다. 디폴트는 tpNormal인데 가급적 빨리 처리해야 할 일이면 우선 순위를 높여줄 수 있다. 그러나 무작정 높게 해 준다고 해서 좋은 것은 아니다. 자기만 급하다고 우선 순위를 높이는 것은 다른 프로그램에 대해 예의에 어긋나며 잘못하면 시스템 전체의 성능이 저하될 수도 있다.

이상으로 다소 복잡한 멀티 스레드에 대해 알아 보았다. 여기까지는 그리 어렵지 않게 이해되겠지만 여기서 다루지 않은 스레드간의 정보교환, 동기화, 스레드 지역 기억장소까지 이해하려면 무척 난해하며 디버깅하기도 쉽지 않아 사용하기도 불편하다. 좋은 기능이기는 하지만 꼭 필요할 때만 사용해야지 남발하게 되면 프로그램만 불안정해지게 된다. 여기서는 스레드에 대한 개념 정도만 알아 보았는데 나머지 이론에 관해서는 Win32 API 서적을 참고하기 바란다. 델

파이와 함께 제공되는 Thrddemo.dpr 예제를 분석해 보면 고급스러운 스레드 사용법에 대해 많은 것을 얻을 수 있을 것이다.


이 예제는 세 개의 스레드를 사용하여 동시에 정렬을 수행하는 시범을 보인다.

# 12-10 인터넷 프로그래밍

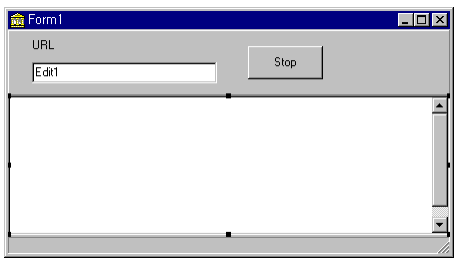
## 가. 웹 브라우저

우리나라에 인터넷이 일반에 소개된 것은 95년부터였다. 그때부터 너나할 것 없이 인터넷에 푹 빠져들게 되었으며 지금은 꽤 큰 시장을 형성하고 있는 정도다 그래서 인터넷과 관련된 기술들이 많이 발표되었고 인터넷 개발을 지원하는 툴들도 급격하고 늘어났다. 델파이도 인터넷 프로그래밍을 지원하며 컴포넌트 팔레트의 Internet 페이지를 보면 인터넷 관련 컴포넌트가 꽤 많이 포함되어 있다.


이 컴포넌트중 일부는 볼랜드에서 만든 것이 아니며 NetManage사에서 만든 것을 볼랜드가 라이센스한 것들이다. 이 많은 컴포넌트들을 다 다루어 본다는 것은 이 책의 범위를 훨씬 뛰어 넘는 것이고 그렇다고 전혀 안 볼 수는 없으니 특징적인 몇 가지만 살펴볼 것이다. 그 중에 그나마 다루기 쉬운 HTML 컴포넌트를 사용해서 웹 브라우저나 한번 만들어 보도록 하자.


12jang
WBro

새 프로젝트를 시작하고 폼 상단에 패널, 하단에 상태란을 배치한다. 그리고 중앙은 모두 HTML 컴포넌트로 가득 채웠다.




패널에는 에디트 하나와 레이블 버튼이 있다. 에디트에 URL을 입력하고 Enter를 누르면 지정한 URL 페이지를 가져와 보여줄 것이다. 작성해야 할 소스는 다음과 같다.

```
procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
```

```
begin
  if Key = VK_return then
    begin
    HTML1.RequestDoc(Edit1.Text);
    end;
end;

procedure TForm1.BtnStopClick(Sender: TObject);
begin
  Html1.Cancel(Null);
end;

procedure TForm1.HTML1BeginRetrieval(Sender: TObject);
begin
  StatusBar1.SimpleText:='Retrieving';
end;

procedure TForm1.HTML1EndRetrieval(Sender: TObject);
begin
  StatusBar1.SimpleText:='Completed';
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  StatusBar1.SimpleText:='Ready';
end;

end.
```

특정 URL의 페이지를 가져 오려면 HTML 컴포넌트의 RequestDoc 메소드를 호출하며 이 메소드의 인수로 요청할 URL만 주면 된다. 그러면 HTML 컴포넌트는 지정한 곳으로 가서 HTML 파일과 이미지 파일 등을 가져와 보여 줄 것이다. 인터넷으로부터 문서를 읽기 시작할 때 OnBeginRetrieval 이벤트가 발생하며 문서를 다 읽었을 때 OnEndRetrieval 이벤트가 발생하는데 이 이벤트에서 사용자에게 현재 상황을 보여주거나 간단한 애니메이션을 보여 줄 수도 있다. 이 예제에서는 상태란에 문서를 읽기 시작했음과 다 읽었음을 메시지로 보여주기만 했다.

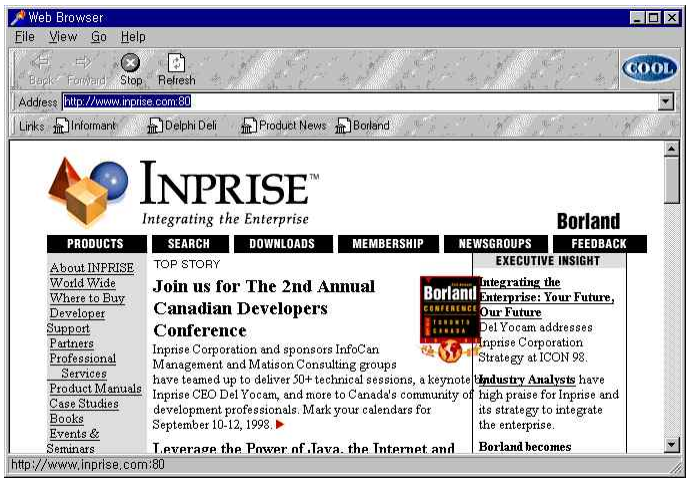
Stop 버튼을 누르면 인터넷 문서 가져 오기를 중지하는데 이 때는 HTML 컴포넌트의 Cancel 메소드를 호출하면 된다. 이 외 한번 간 곳의 URL 목록을 유지해 주는 히스토리 기능이나 사용자가 지정한 곳의 URL을 따로 기억하는 북 마크 기능을 더할 수 있겠지만 이는 문자열 배열을 관리하는 기법 정도면 충분할 것이

다. 실행중의 모습은 다음과 같다.

그림
간단한 웹 브라우저


물론 이 예제가 제대로 실행되려면 인터넷과 접속할 수 있는 환경이 준비되어 있어야 할 것이다. 네트웍에 물려있는 컴퓨터라면 별 준비가 필요없겠지만 그렇지 못한 사람들은 전화접속 네트워킹이나 기타 방법으로 인터넷 접속 환경을 만든 후 이 예제를 실행시켜야 한다. 문서를 가져오는 속도나 이미지 출력질, 정렬 등은 상용 인터넷 탐색기나 네비게이터에 비하면 형편없을 정도라고 할 수 있다. 게다가 ActiveX, 스크립트 등의 최신 기법은 제대로 지원하지 못한다. 한번쯤 흥미삼아 만들어 볼만할지는 몰라도 실무에 쓰기에는 부족함이 많다. 이 컴포넌트보다는 마이크로소프트가 제공하는 인터넷 탐색기 ActiveX 컨트롤을 사용하는 것이 훨씬 더 유리할 것 같다.

참고로 델파이와 함께 제공되는 CoolStuf₩WebBrows 예제를 분석해 보면 이 컴포넌트에 대해 좀 더 많은 정보를 얻을 수 있을 것이다. 히스토리, 에니메이션, HTML 소스 보기 등의 기능이 더 들어있다.




# 나. 웹 서버 프로그래밍

앞에서 만든 웹 브라우저는 웹 클라이언트에서 사용하는 프로그램이다. 인터넷은 기본적으로 클라이언트/서버 모델을 기반으로 동작하며 정보를 제공하는 서버와 제공되는 정보를 사용하는 클라이언트로 구성된다. 서버쪽에서는 클라이언트에게 제공할 정보나 또는 정보를 가공하는 프로그램이 있어야 한다. 서버에서 제공하는 정보가 단순한 HTML 파일이라면 이 페이지는 정적인 페이지가 되며 항상 똑같은 정보를 사용자에게 보여주게 되는데 요즘은 여러 가지 방법으로 웹 페이지에 동적인 요소를 첨가하여 사용자가 원하는 정보만 골라 보여주기도 한다. 웹 서버 프로그래밍이란 사용자의 요구를 받아들여 원하는 정보를 가공하는 그런 프로그램을 만드는 것을 말한다.

웹 서버 프로그래밍을 해 보려면 당연히 웹 서버가 있어야 하며 완벽한 테스트를 위해 인터넷에 접속할 수 있는 환경이 마련되어 있어야 한다. 웹 서버는 NT, 유닉스, 리눅스 등 여러 가지 환경에서 동작하는데 요즘은 NT 4.0 에 IIS 로 웹 서버를 구성하는 것이 가장 일반적이다. 하지만 이 책을 읽는 독자중에 이런 시스템을 가지고 있는 사람은 그리 많지 않으므로 여기서는 윈 95 나 윈 98 에서 웹 서버 환경을 만들어 주는 퍼스널 웹 서버를 사용하기로 한다. 이 프로그램은 크기도 작고 공짜로 사용할 수 있다는 점에서 아주 반가운 존재이다.

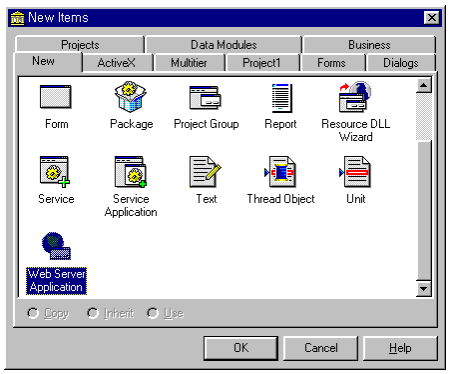
웹 서버 프로그래밍에 관심이 있는 사람은 먼저 퍼스널 웹 서버를 구해 자신의 시스템에 설치해야 한다. 그런데 이 프로그램은 아직까지는 이런 저런 충돌도 많고 어떤 환경에서는 제대로 동작하지 않는 버그가 있어 역시 제대로 프로그

래밍을 해 보려면 NT 와 IIS 3.0 을 준비하는 것이 좋을 듯 하다.

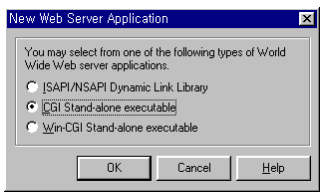
12jang
CGIExam

델파이는 ISAPI, CGI, Win-CGI 등 세 가지 형태의 웹 어플리케이션 개발을 지원하는데 셋 다 결과만 다를 뿐이지 만드는 과정은 동일하다. 여기서는 대표적으로 CGI 예제만 간단하게 만들어 보기로 한다. 단 여기서는 독자들이 인터넷의 기본 개념이나 HTML 문법, 퍼스널 웹 서버, CGI 등에 익숙하다고 가정하고 실습을 진행한다. 웹 서버 어플리케이션 개발은 그 자체만으로 독립된 분야로 취급될 정도로 광범위하기 때문에 여기서 기반 이론까지 논할 수 없음을 양해 바란다. 일단 델파이를 실행하고 다음 단계를 따라 예제를 만들어 보자.

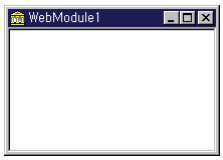
1 여기서 만들어 볼 예제는 문자열을 클라이언트로 보내기만 하는 아주 간단한 예제이다. File/New 메뉴 항목을 선택하여 New Item 대화상자를 불러낸다.




이 대화상자의 제일 아래쪽에 있는 Web Server Application 을 선택한다. 그러면 웹 서버 어플리케이션의 유형을 물어올 것이다.




제일 위에 있는 ISAPI 유형을 선택하면 DLL 이 만들어지고 나머지 두 유형은 실행 파일을 만든다. 실행 파일이 테스트하기 편리하므로 일단 CGI Stand-alone executable 을 선택한다. 그러면 빈 웹 모듈이 나타날 것이다.



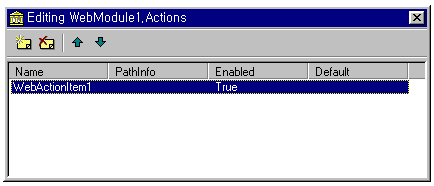

웹 모듈은 비 가시적 컴포넌트를 담는 컨테이너이며 또한 HTTP 프로토콜의 요구에 응답할 수 있는 기능을 가지고 있다. 즉 클라이언트 프로그램의 요구에 따라 원하는 정보를 가공해서 제공할 수 있다는 뜻이다. 이때 클라이언트의 요구를 액션(Action)이라고 하며 웹 모듈은 이런 액션을 담는 집합체라고 할 수 있다. 단 여기서의 액션은 5 장에서 배운 TAction 컴포넌트와는 다른 것이므로 구분하도록 하자.

여기까지 작업을 진행하고 프로젝트의 소스를 보면 다음과 같이 작성되어 있을 것이다.

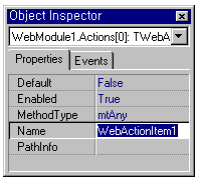
```
program CGIExam;

{$APPTYPE CONSOLE}

uses
  HTTPApp,
  CGIApp,
  CGIExam_f in 'CGIExam_f.pas' {WebModule1: TWebModule};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

program 지시자로 시작되었으므로 실행 파일(EXE)를 만든다는 것을 알 수 있다. 응용 프로그램 타입이 콘솔 형식으로 되어 있는데 웹 서버 어플리케이션의 특성상 자신이 직접 윈도우를 만들어 출력하지 않고 클라이언트인 웹 브라우저로 문자열을 보내기만 하면 되므로 콘솔 형식으로 되어 있다. uses 문에는 두 개의 유닛을 사용하고 있는데 HTTPApp 는 모든 웹 서버 어플리케이션에서 사용하는 유닛이며 CGIApp 는 CGI 유형이 어플리케이션이 사용하는 유닛이다.

2 웹 모듈을 더블클릭하여 웹 모듈 액션 편집기를 호출한다. 최초 아무런 액션도 생성되어 있지 않은데 좌상단의 Add New 버튼을 클릭하면 WebActionItem1 이라는 이름의 새 액션이 만들어진다.




이렇게 만들어진 액션은 TWebActionItem 컴포넌트이며 다른 컴포넌트와 마찬가지로 고유의 속성과 이벤트를 가진다. 웹 모듈 액션 편집기에서 액션을 선택하면 오브젝트 인스펙터에 이 액션의 속성들이 나타날 것이다.




액션 하나는 클라이언트의 요구 하나에 대응된다.

3 속성은 모두 디폴트로 두고 Events 페이지에서 OnAction 을 더블클릭하여 이벤트 핸들러를 만든다. 다음과 같이 복잡한 모양의 핸들러가 만들어질 것이다.

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin

end;
```

Request 인수는 액션을 요청한 클라이언트의 정보를 전달하는 TWebRequest 클래스의 오브젝트이다. TWebRequest 클래스는 추상 클래스이므로 그 자체로는 사용될 수 없으며 이 클래스로부터 파생되는 클래스가 사용된다. Request 오브젝트가 어떤 클래스형인가는 웹 서버 어플리케이션의 유형에 따라 달라지는데 ISAPI 이면 TISAPIRequest, WinCGI 면 TWinCGIRequest 클래스형이 된다. 이 예제의 경우 CGI 프로그램이므로 TCGIRequest 오브젝트가 전달될 것이다. Request 오브젝트로부터 어떤 정보를 얻을 수 있는지는

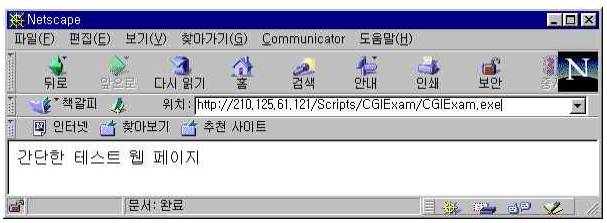
TWebRequest, TCGIRequest 클래스의 도움말에서 속성란을 참조하면 된다.

Response 참조 인수는 서버에서 클라이언트로 출력되는 정보를 대입받는 TWebResponse 클래스형의 오브젝트이다. 마찬가지로 TWebRespose 도 추상 클래스이므로 Response 인수는 웹 서버 어플리케이션의 유형에 따라 TISAPIResponse, TCGIResponse, TWinCGIResponse 클래스 형 중 하나가 될 것이다. Response 의 Content 필드에 출력할 문자열(또는 HTML)을 대입해 주면 이 정보가 클라이언트의 웹 브라우저에 나타나게 된다. 간단한 문자열을 출력해 보자.

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.content:='간단한 테스트 웹 페이지';
end;
```

4 테스트를 위해서는 이름을 가지는 실행 파일이 있어야 하므로 프로젝트를 저장해야 한다. 이때 저장되는 디렉토리는 실행 권한을 가지는 디렉토리여야 한다. WebShare₩Scripts 디렉토리 아래 CGIExam 이라는 디렉토리를 만들고 CGIExam 이라는 이름으로 프로젝트를 저장한다. 또는 이 프로젝트는 아무 디렉토리에 저장해도 상관없는데 그럴 경우는 컴파일한 후 실행 파일만이라도 WebShare₩Scripts 디렉토리로 복사해 주어야 한다.

5 컴파일한 후 실행해 보자. 실행해 봐야 도스 창만 잠깐 열렸다가 닫힐 것이다. 제대로 실행 결과를 보려면 웹 브라우저로 이 파일을 불러와 봐야 한다. 넷스케이프 네비게이터나 인터넷 탐색기를 실행한 후 이 실행 파일이 있는 경로를 URL 로 준다. 필자의 컴퓨터는 실제 인터넷에 접속되어 있기 때문에 IP 주소를 바로 주었는데 만약 여러분의 컴퓨터가 인터넷에 접속되어 있지 않은 상태라면 IP 주소로 127.0.0.1 로 주면 된다.




웹 브라우저에 우리가 출력한 문자열이 나타났다. 이 문자열은 브라우저의 요청에 의해 웹 서버에서 웹 서버 어플리케이션을 실행한 결과를 다시 브라우저로

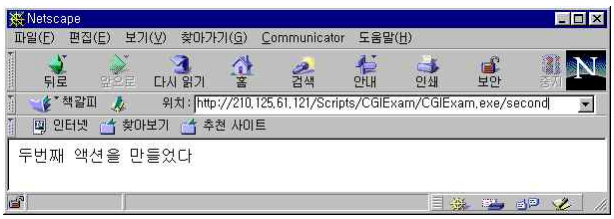
전송하여 나타나는 것이다. 예제를 간단하게 만드느라 고정된 문자열만 출력했지만 프로그래밍하기에 따라서는 얼마든지 복잡한 정보를 전달해 줄 수도 있다. 이 프로그램을 웹 서버 디렉토리에 위치해 두면 세계 어느곳에서나 해당 웹 서버에 접속하여 이 정보를 참조할 수 있게 된다.

한 실행 파일에서 처리할 수 있는 액션의 개수는 제한이 없다. 즉 하나의 웹 서버 어플리케이션으로 브라우저의 여러 가지 요구를 처리할 수 있다는 뜻이다. 계속해서 두 번째 액션을 만들어 보자.

웹 모듈 액션 에디터에서 Add New 버튼을 눌러 액션을 하나 더 만들어 둔다. 그리고 이 액션을 앞에서 만든 액션과 구분하기 위해서 PathInfo 속성에 값을 주어야 한다. PathInfo 속성은 클라이언트로부터 전달된 요구를 구분하기 위해 사용되는 액션간의 구분 기호라고 할 수 있다. 두 번째 액션에 /second 패스 인포를 줘 보자. 그리고 이 액션의 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content:='두번째 액션을 만들었다';
end;
```

이제 이 프로그램을 실행할 때 실행 파일 뒤에 실행하고자 하는 액션의 PathInfo 를 주면 된다.



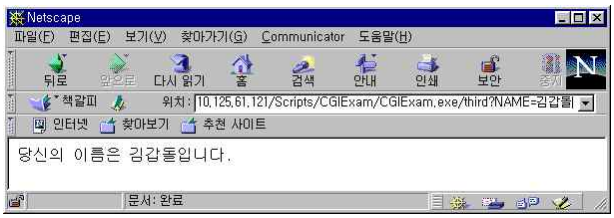

이런 식으로 PathInfo 만 구분해 주면 얼마든지 액션을 만들 수 있다. 이번에는 하나의 액션에 인수를 주어 인수값을 참조하여 액션을 실행시키는 방법에 대해 알아 보자. 웹 브라우저에서 인수는 ? 다음에 “인수명=값”의 형태로 주어지며 이 값이 액션의 인수로 전달된다.

세 번째 액션을 만들고 PathInfo 를 /third 로 준다. 그리고 이 액션의 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
```

```
begin
  Response.Content:='당신의 이름은 '+
    Request.QueryFields.Values['NAME']+'입니다.';
end;
```

코드에서 인수는 Request.QueryFields.Values 다음에 대괄호 안에 인수 이름을 적어 주면 된다. 실행할 때는 PathInfo 다음에 ?를 적어주고 인수를 전달해 준다.




전달된 인수가 액션 내에서 사용되었다. 이렇게 전달할 수 있는 인수의 개수에는 제한이 없으므로 얼마든지 필요한만큼 인수를 만들어 쓸 수 있다. 그런데 사실 웹 서버 어플리케이션이 인수를 요구할 때 사용자들이 직접 인수를 적어주는 경우는 없다. 그렇게 하면 사용자들이 웹 서버 어플리케이션이 어떤 인수를 어떤 이름으로 사용하는지를 일일이 알아야 한다는 얘기인데 현실적으로 불가능한 얘기이다.

그래서 보통 폼(여기서 말하는 폼은 델파이의 TForm 과는 다른 것이다)을 만들고 폼에서 사용자로부터 정보를 입력받은 후에 이 정보를 웹 서버 어플리케이션의 인수로 전달해 준다. 폼을 만들려면 HTML 파일을 작성해야 하는데 필자는 프론트 페이지를 사용하여 간단한 HTML 파일을 작성하였다. 웹 모듈에 PageProducer 컴포넌트를 배치하고 HTMLDoc 속성에 다음 HTML 문장을 작성한다. 또는 이 파일을 별도의 분리된 파일로 저장해 놓고 HTMLFile 속성에 파일명을 대입해 줄 수도 있다.

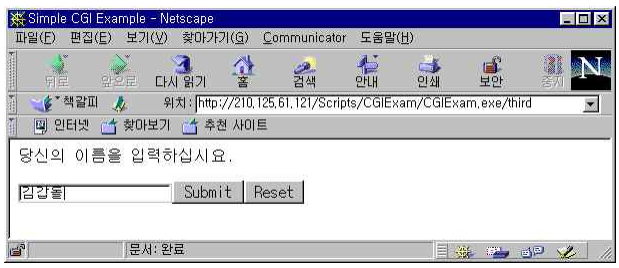
```
<html>

<head>
<title>Simple CGI Example</title>
</head>

<body>

<form action="http://210.125.61.121/Scripts/CGIExam/CGIExam.exe/third">
 <p>당신의 이름을 입력하십시요.</p>
```

```
<p><input type="text" name="NAME" size="20"><input type="submit" value="Submit"
name="B1"><input
 type="reset" value="Reset" name="B2"></p>
</form>
</body>
</html>
```

이 HTML 문은 에디트가 있는 폼을 열어서 사용자로부터 이름을 입력받아 NAME 인수로 전달해 준다. third 액션의 이벤트 핸들러를 다음과 같이 변경한다.

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 if Request.QueryFields.Values['NAME']='' then
 begin
  Response.Content:=PageProducer1.Content;
 end
 else
 begin
 Response.Content:='당신의 이름은 '+
  Request.QueryFields.Values['NAME']+'입니다.';
  end;
end;
```

NAME 인수가 전달되지 않을 때는 폼을 열어 이름을 물어오고 폼에서 NAME 인수를 전달해 주면 이 인수를 출력해 주도록 하였다. 실행중의 모습은 다음과 같다.




에디트에 이름을 입력하고 Submit 버튼을 누르면 이 정보가 웹 서버 어플리케이션으로 전달된다.

# 다. 웹과 데이터 베이스의 연동

12jang
WebDB

웹 서버 어플리케이션에서 클라이언트로 전송되는 정보는 일반적으로 데이터 베이스에서 추출한 레코드들이다. 예를 들어 검색 사이트의 경우 사용자가 입력한 키워드로부터 조건에 맞는 레코드를 검색하여 클라이언트로 보내 준다. 웹과 데이터 베이스를 접합한 예는 검색 사이트, 온라인 쇼핑 몰, 방명록, 질문과 답(Q&A) 등 얼마든지 많이 있다.

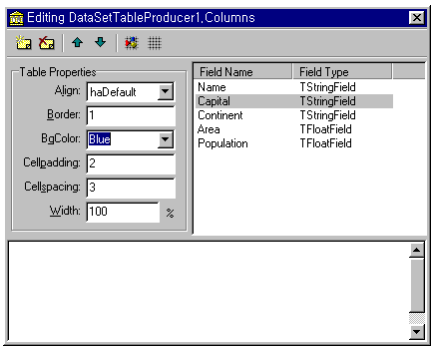
여기서는 데이터 베이스의 테이블을 웹에 출력하는 간단한 예만 보이기로 한다. 단 이 실습을 해 보기 위해서는 델파이에서 데이터 베이스 프로그래밍을 할 수 있어야 하므로 먼저 15 장을 읽어 보아야 한다.

1 File/New 를 선택한 후 Web Module Application 을 선택하고 서버 유형으로 CGI Stand-alone executable 을 선택한다. 앞에서 만든 예제와 완전히 동일하다.

2 데이터 베이스와 접속하기 위해 Table 컴포넌트를 배치하고 이 테이블을 웹 페이지로 보내기 위해 DataSetTableProducer 컴포넌트를 배치한다. 데이터 셋 테이블 프로듀서 컴포넌트는 데이터 셋의 레코드를 HTML 로 변환해 주는 역할을 하는 컴포넌트이다.

3 테이블을 데이터 베이스와 연결한다. DatabaseName 속성에 DBDEMOS 를 주고 TableName 속성에 contry.db(또는 다른 테이블을 사용해도 상관없다)를 지정한다. 테이블의 Active 속성을 True 로 변경하면 데이터 베이스와 연결이 완료되었다.

4 이제 테이블에 들어온 레코드들을 HTML 로 만들어 클라이언트로 전송해 주도록 해야 한다. 데이터 셋 테이블 프로듀서의 DataSet 을 Table1 으로 지정하면 Table1 과 연결된 테이블 내용을 이 컴포넌트에서 인식하게 된다. 테이블의 모양을 설정하기 위해 팝업 메뉴에서 Response Editor 를 선택한다.




이 대화상자에서 정렬, 색상, 간격 등 테이블의 여러 가지 속성을 설정할 수 있는데 일단 배경색만 파랑색으로 변경해 보자.

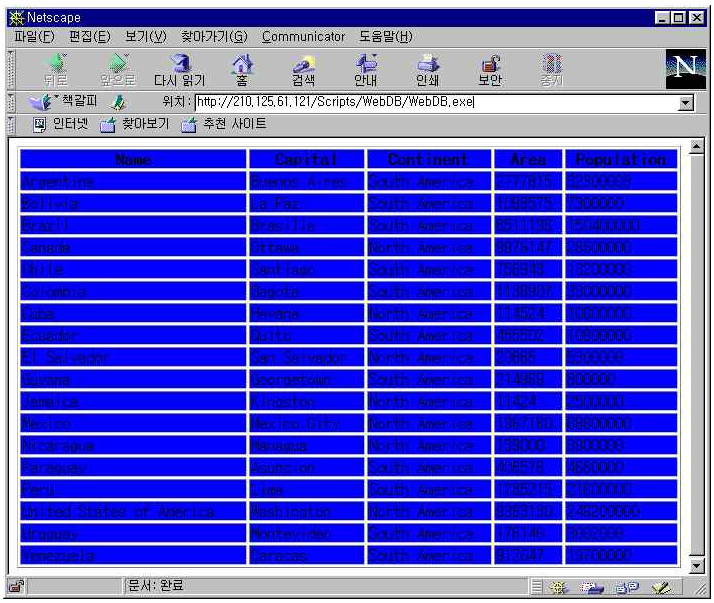
5 액션을 하나 만들고 OnAction 의 이벤트 핸들러를 작성한다.

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content:=DataSetTableProducer1.Content;
end;
```

데이터 셋 테이블 프로듀서 컴포넌트의 Content 속성, 즉 테이블의 내용을 HTML 로 만든 문자열을 Response.Content 로 복사하였다. 따라서 이 문자열이 클라이언트로 전송될 것이며 웹 브라우저에는 테이블 내용이 표로 나타날 것이다.

6 이 프로젝트를 WebDB 라는 이름으로 저장하고 실행 파일을 만들기 위해 컴파일해 준다.

7 제대로 테이블 내용이 나타나는지 테스트해 보자. 네비게이터를 실행하고 이 실행 파일의 경로를 URL 에 입력해 주면 테이블 내용이 웹 브라우저에 나타날 것이다.




웹 브라우저로 전송되는 실제 HTML 파일은 다음과 같다. 이 파일이 데이터셋 테이블 프로듀서 컴포넌트에 의해 만들어진 것이다.

```
<Table Width="100%" Border=1 BgColor="Blue">
<TR><TH>Name</TH><TH>Capital</TH><TH>Continent</TH><TH>Area</TH><TH>Populati
on</TH></TR>
<TR><TD>Argentina</TD><TD>Buenos Aires</TD><TD>South
America</TD><TD>2777815</TD><TD>32300003</TD></TR>
<TR><TD>Bolivia</TD><TD>La Paz</TD><TD>South
America</TD><TD>1098575</TD><TD>7300000</TD></TR>
<TR><TD>Brazil</TD><TD>Brasilia</TD><TD>South
America</TD><TD>8511196</TD><TD>150400000</TD></TR>
<TR><TD>Canada</TD><TD>Ottawa</TD><TD>North
America</TD><TD>9976147</TD><TD>26500000</TD></TR>
<TR><TD>Chile</TD><TD>Santiago</TD><TD>South
America</TD><TD>756943</TD><TD>13200000</TD></TR>
<TR><TD>Colombia</TD><TD>Bagota</TD><TD>South
America</TD><TD>1138907</TD><TD>33000000</TD></TR>
<TR><TD>Cuba</TD><TD>Havana</TD><TD>North
America</TD><TD>114524</TD><TD>10600000</TD></TR>
<TR><TD>Ecuador</TD><TD>Quito</TD><TD>South
America</TD><TD>455502</TD><TD>10600000</TD></TR>
<TR><TD>El Salvador</TD><TD>San Salvador</TD><TD>North
America</TD><TD>20865</TD><TD>5300000</TD></TR>
<TR><TD>Guyana</TD><TD>Georgetown</TD><TD>South
```

```
America</TD><TD>214969</TD><TD>800000</TD></TR>
<TR><TD>Jamaica</TD><TD>Kingston</TD><TD>North
America</TD><TD>11424</TD><TD>2500000</TD></TR>
<TR><TD>Mexico</TD><TD>Mexico City</TD><TD>North
America</TD><TD>1967180</TD><TD>88600000</TD></TR>
<TR><TD>Nicaragua</TD><TD>Managua</TD><TD>North
America</TD><TD>139000</TD><TD>3900000</TD></TR>
<TR><TD>Paraguay</TD><TD>Asuncion</TD><TD>South
America</TD><TD>406576</TD><TD>4660000</TD></TR>
<TR><TD>Peru</TD><TD>Lima</TD><TD>South
America</TD><TD>1285215</TD><TD>21600000</TD></TR>
<TR><TD>United States of America</TD><TD>Washington</TD><TD>North
America</TD><TD>9363130</TD><TD>249200000</TD></TR>
<TR><TD>Uruguay</TD><TD>Montevideo</TD><TD>South
America</TD><TD>176140</TD><TD>3002000</TD></TR>
<TR><TD>Venezuela</TD><TD>Caracas</TD><TD>South
America</TD><TD>912047</TD><TD>19700000</TD></TR>
</Table>
```

이 경우는 물론 테이블 내용을 보여주기만 하며 직접 수정하거나 조건에 맞는 레코드만 검색해 볼 수는 없다. 만약 쿼리를 사용한다면 TDataSetTableProducer 컴포넌트 대신 TQueryTableProducer 컴포넌트를 사용하면 된다.

# 12-11 전역 오브젝트

## 가. Application

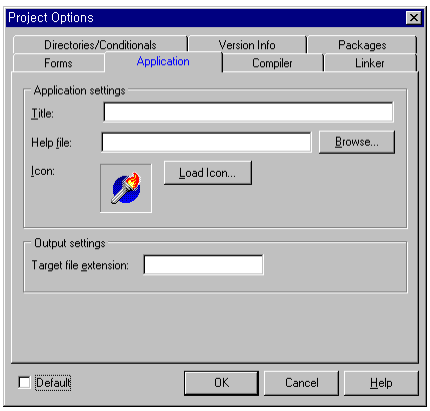
델파이는 몇 개의 전역 오브젝트를 제공한다. 전역 오브젝트(Global Object)란 프로그램이 실행된 직후에 생성되어 프로그램 종료 직전에 파괴되며 실행중에 언제든지 쓸 수 있는 오브젝트를 말한다. 그래서 별도의 선언이나 생성을 할 필요가 없다. 이런 전역 오브젝트의 대표적인 예로 TApplication 컴포넌트가 있다. 이 오브젝트의 존재는 프로젝트의 소스 파일에서 직접 확인할 수 있다.

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

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

이 소스의 begin과 end 사이에 Application이라는 오브젝트가 사용되고 있는데 이 오브젝트가 바로 TAppliation 컴포넌트이다. Application 오브젝트는 윈도우즈 프로그램 그 자체를 캡슐화하고 있으며 이 오브젝트의 속성, 메소드는 프로그램 전체에 영향을 주며 또한 이벤트는 프로그램 자체가 받는 여러 가지 사건을 나타낸다. 그래서 이 오브젝트를 잘 프로그래밍하면 응용 프로그램 자체를 프로그래밍할 수 있다.

Application의 속성에는 Handle, Active 등의 저수준 정보를 나타내는 속성이 있고 Icon, Title, HelpFile 등 응용 프로그램 외형을 정의하는 속성이 있다. Application 컴포넌트는 컴포넌트 팔레트에도 없고 비가시적이기 때문에 오브젝트 인스펙터로 이 속성들을 변경할 수는 없다. 이 속성들은

Project/Options/Applicaion 페이지에서 변경한다.




이외에 관심을 가질 만한 속성으로는 HintColor라는 속성이 있는데 이 속성 값을 변경하면 풍선 도움말의 색상을 마음대로 바꿀 수 있다. 이 속성도 디자인 중에는 변경할 수 없으므로 코드에서(보통 FormCreate 이벤트 핸들러를 사용한다) 변경해 주어야 한다. 다음과 같이

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.HintColor:=clRed;
end;
```

Application의 메소드에는 Initialize, CreateForm, Run 등이 있는데 이 메소드들은 델파이가 디폴트로 호출해 주도록 되어 있으며 사용자가 직접 사용할만한 것은 응용 프로그램을 최소화하는 Minimize나 메시지 상자를 출력하는 MessageBox 등이 있다. 그외의 메소드들은 사용자가 직접 사용할 경우가 극히 드물거나 필요하더라도 굉장히 고급 기법이 동원되어야 하는 것들이다.

12jang
OnActive

Application 오브젝트에 관심을 가져야 하는 가장 큰 이유는 이 오브젝트의 이벤트가 쓸만한 것들이 많기 때문이다. Applicaion은 프로그램 그 자체를 나타내며 다른 어떤 오브젝트보다 먼저 이벤트를 받기 때문에 프로그램 전역적인 처리가 필요한 경우 이 오브젝트의 이벤트가 사용된다. 도움말을 보면 Application의 이벤트에 어떤 것들이 있는지 확인할 수 있겠지만 이중 비교적 간단한 OnActivate 이벤트에 대해 알아보고 간단한 예제를 만들어 보자.

OnActivate 이벤트는 프로그램이 활성화될 때 발생하며 OnDeActivate 이벤트는 프로그램이 포커스를 잃을 때 발생한다. 이 두 이벤트를 활용하면 사용

자가 이 프로그램을 사용할 때만 특정한 동작을 하도록 할 수 있다. 새 프로젝트를 시작하고 폼을 더블클릭하여 FormCreate 이벤트 핸들러를 만들도록 하자. 그리고 다음 코드를 죄다 입력해 넣는다. 굵은 글꼴로 표시한 부분을 모두 직접 입력해야 한다.

```
unit OnActive_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure OnAppActive(Sender: TObject);
    procedure OnAppDeActive(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnActivate:=OnAppActive;
  Application.OnDeActivate:=OnAppDeActive;
end;

procedure TForm1.OnAppActive(Sender: TObject);
begin
  Color:=clBlue;
end;

procedure TForm1.OnAppDeActive(Sender: TObject);
begin
  Color:=clSilver;
```

```
end;

end.
```

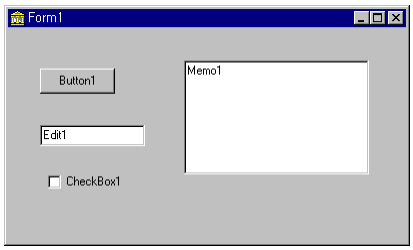
Application 컴포넌트는 디자인중에 사용할 수 없으므로 속성을 바꾸는 것은 물론이고 이벤트 핸들러를 지정하는 것도 코드에서 직접 해야 하며 이벤트 핸들러 함수도 직접 만들어 주어야 한다. 이 예제를 실행시키면 폼이 활성화 중일 때, 즉 사용자가 이 폼에 포커스를 주었을 때는 파란색이 되며 다른 프로그램으로 포커스를 이동했을 때는 회색이 된다. Application의 두 이벤트 핸들러에서 활성/비활성 될 때 폼의 색상을 바꿔주고 있기 때문이다. 활성화되어 있을 때만 애니메이션을 보여준다거나 할 때 이런 이벤트를 사용해 봄 직하다.

12jang
OnMessage

다음은 가장 활용도가 높으며 또한 강력한 기법을 구할 수 있는 OnMessage 이벤트에 대해 알아 보자. 알다시피 윈도우즈는 처음부터 끝까지 메시지에 의해 동작하는 메시지 구동 시스템이다. 응용 프로그램이나 컨트롤이나 메시지를 받아 동작을 하도록 되어 있으며 이런 메시지들은 델파이에서 이벤트 형태로 나타난다. 사용자의 모든 입력은 윈도우즈의 시스템 메시지 큐에 일단 모이며 윈도우즈는 여기에 모인 메시지들을 관련된 응용 프로그램의 메시지 큐로 보내준다. 응용 프로그램은 이렇게 받은 메시지들을 다시 관련된 컨트롤로 보내고 컨트롤들은 자신에게 보내진 메시지를 분석하여 동작한다.

이때 응용 프로그램으로 메시지가 보내질 때 발생하는 이벤트가 바로 OnMessage 이벤트이다. 이 이벤트의 핸들러를 만들면 응용 프로그램에 발생하는 모든 메시지를 검사할 수 있으며 중간에 가로채거나 다른 메시지로 변경할 수도 있다. 즉 OnMessage 이벤트의 핸들러는 메시지에 대한 모든 조작을 가할 수 있는 중앙 메시지 통제 센터라고 할 수 있다. 디폴트로 이 이벤트에 대한 핸들러가 작성되어 있지 않으므로 모든 메시지는 정해진 컨트롤로 가게 되지만 핸들러에서 조작을 하면 그렇지 않을 수도 있다.

그럼 메시지를 조작하는 간단한 예제를 만들어 보자. 새 프로젝트를 시작하고 폼에 몇 가지 컴포넌트를 배치해 보았다.




코드는 다음과 같다.

```
unit OnMessage_f;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
    Edit1: TEdit;
    CheckBox1: TCheckBox;
    procedure FormCreate(Sender: TObject);
    procedure OnAppMessage(var Msg: TMsg; var Handled: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage:=OnAppMessage;
end;
```

```
procedure TForm1.OnAppMessage(var Msg: TMsg; var Handled: Boolean);
begin
 if (Msg.message=Wm_KeyDown) and (Msg.wParam = VK_BACK) then
  begin
  ShowMessage('BS 키를 눌렀습니다');
  Handled:=True;
  end;
end;

end.
```

OnMessage 이벤트 핸들러는 두 개의 인수를 전달받는데 첫 번째 인수는 윈도우즈의 메시지를 담는 TMsg 레코드이다.

```
type

 TMsg = packed record
  hwnd: HWND;
  message: UINT;
  wParam: WPARAM;
  lParam: LPARAM;
  time: DWORD;
  pt: TPoint;

 end;
```

두 번째 인수 Handled는 이 핸들러가 메시지를 처리했는지를 알려주는 출력용 인수이며 이 인수에 True를 대입해 주면 이 메시지는 처리되었으므로 컴포넌트로 전달되지 않는다. 예제를 실행해 보면 BS키가 눌러질 때마다 메시지 박스만 나타나고 BS 기능이 제대로 동작하지 않을 것이다. 왜냐하면 OnMessage 이벤트 핸들러에서 전달되는 모든 메시지를 검사하여 WM_KEYDOWN 메시지이고 눌러진 키가 BS일 경우는 메시지를 가로채 버리기 때문이다. OnMessage 핸들러는 이런 식으로 모든 메시지에 대한 통제를 할 수 있다. 만약 위 예제에서 Handled에 False를 대입해 주면 메시지 상자도 출력되고 BS 기능도 제대로 동작할 것이다.

예제를 간단하게 만들기 위해 실용성 빵점인 예제를 만들었지만 OnMessage 이벤트 핸들러에서 무슨 일을 할 수 있는지는 감을 잡을 수 있을 것이다. 중요한 것은 인수로 전달된 Msg와 Handled가 둘 다 참조 호출 인수라는 점이다. 즉 Msg 레코드의 필드를 마음대로 조작할 수 있으며 그래서 메시지 정

보에 대해 입맛대로 조작을 할 수 있다. 물론 이 정도 기법을 구사하려면 Win32의 메시지에 대한 해박한 지식이 필요하다.

## 나. OnIdle

12jang
OnIdle

Application 오브젝트에 발생하는 이벤트 중에 흥미로운 또 하나의 이벤트는 OnIdle이다. 이 이벤트는 말 그대로 응용 프로그램이 아이들 상태일 때, 즉 별로 할 일이 없을 때 발생한다. 응용 프로그램은 쉴새없이 사용자나 시스템으로부터 메시지를 받아들이지만 그렇다고 해서 항상 바쁜 것은 아니다. 사용자가 키보드를 아무리 빨리 쳐도 컴퓨터는 이를 거뜬히 다 처리하고도 틈틈히 휴식을 취하고 있다. 사실 CPU란 놈은 실제로 무척 바쁘게 돌아가지만 대부분의 시간을 사용자의 입력을 대기하며 빈둥 빈둥 놀고 있다는 것이다. 왜냐하면 사람이 CPU에 비해 형편없이 느리기 때문이다. 이럴 때 발생하는 이벤트가 바로 OnIdle 이벤트이다.

즉 내가 지금 특별히 바쁘지 않은 상태에 있다(=처리할 메시지가 없다)는 신호인 것이다. 응용 프로그램은 우선 순위가 높지 않은 작업을 이 이벤트 핸들러에서 할 수 있으며 그럴 경우 CPU는 남아 도는 시간을 최대한으로 활용할 수 있어 효율적이다. 이 이벤트의 형식을 보면 다음과 같다.

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;
property OnIdle: TIdleEvent
```

Sender 인수는 별 관심을 가질 필요가 없고 두 번째 인수 Done이 아주 중요하다. 앞에 var가 붙었으므로 이 인수는 출력용 참조 인수인데 OnIdle에서 아이들 타임에 해야 할 일을 다 했으면 이 인수에 True를 대입해 주고 아직도 할 일이 남아 있으면 False를 대입해 주면 된다. 운영체제는 응용 프로그램이 처리해야 할 메시지 없이 놀고 있을 때 OnIdle 이벤트를 발생시켜 보고 Done에 True가 대입되면 더 이상 OnIdle을 호출하지 않고 다른 응용 프로그램에게 CPU 시간을 양보한다. 반면 Done에 False가 대입되면 처리해야 할 다른 메시지가 발생할 때까지 OnIdle을 계속 호출해 준다.

앞에서 스레드를 이용해 점을 찍는 예제를 이번에는 OnIdle 이벤트를 사용해서 만들어 보자. 새 프로젝트를 시작하고 컴포넌트는 배치할 필요없고 폼을 흰색으로 바꾼 후 다음 코드를 작성한다. 역시 직접 입력해야 할 부분은 굵은 글꼴로 표시하였다.

```
unit OnIdle_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure OnAppIdle(Sender: TObject; var Done: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Count:Integer;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnIdle:=OnAppIdle;
end;

procedure TForm1.OnAppIdle(Sender: TObject; var Done: Boolean);
begin
  Form1.Caption:=IntToStr(Count);
  Form1.Canvas.Pixels[random(400),random(300)]:=
    RGB(random(256),random(256),random(256));
  Count:=Count+1;
  Done:=False;
end;

end.
```

실행해 보면 폼에 점이 계속 찍히면서도 다른 작업이 가능할 것이다. 스레드

를 사용한 예제와 거의 동일하게 동작하는데 스레드와는 본질적으로 다른 차이점이 있다. 우선 OnIdle 이벤트를 사용한 다중 처리는 실제로 다중 처리가 아니라 다른 일을 하면서 틈틈이 백그라운드 작업을 한다는 점이다. 그래서 응용 프로그램이 바쁘면 OnIdle 처리는 거의 무시되다시피 한다. OnIdle 이벤트는 처리할 다른 메시지가 없을 때에만 보내지기 때문이다.

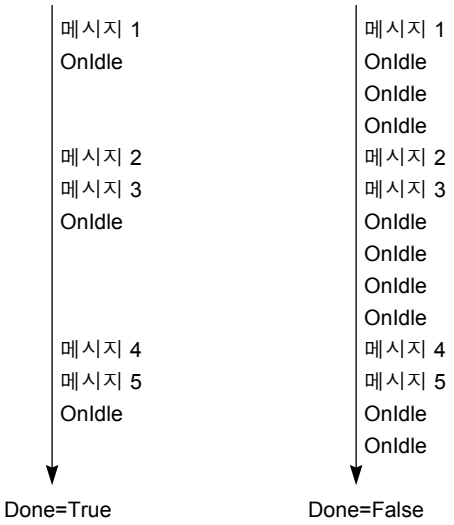
과연 그런지 예제를 실행시켜 놓고 마우스로 폼을 이리저리 흔들어 보거나 다른 프로그램으로 폼을 가렸다 드러냈다 해 보면 점을 찍는 속도가 현저히 느려짐을 느낄 수 있을 것이다. 또는 마우스로 폼의 타이틀 바를 누르면 점 찍는 작업이 일체 중단되는데 왜냐햐면 현재 이 프로그램은 자신의 위치를 바꾸는 일로 무척 바쁘며 따라서 OnIdle 이벤트가 발생하지 않기 때문이다. 반면 Thread 예제는 진짜 다중 처리를 하기 때문에 무슨 일을 하건 스레드가 이상없이 돌아간다.

다중 처리의 질에 있어서는 OnIdle 이벤트를 사용하는 방법이 스레드를 사용하는 방법에 비해 훨씬 더 열등한 존재다. 하지만 CPU 시간을 알뜰하게 사용하는 면에 있어서는 OnIdle 이 훨씬 더 유리하다. 스레드는 다른 프로그램이나 다른 스레드가 바쁘건 말건 일정한 CPU 시간을 사용하지만 OnIdle 이벤트는 한가할 때만 CPU 시간을 사용하기 때문이다. 그래서 별로 급하지 않은 작업은 OnIdle 에서 처리하는 것이 더 좋으며 프로그램의 반응성을 높이는 방법이다.

OnIdle 의 Done 참조 인수에 대해 좀 더 연구해 보자. 이 인수는 OnIdle 이 계속 호출될 것인가 아닌가를 운영체제에게 알려주는 역할을 한다. 이 인수가 False 이면 OnIdle 의 작업이 덜 끝났으므로 계속 OnIdle 을 호출해 주지만 이 인수가 True 이면 다른 메시지를 처리한 후 아이들 타임이 다시 오기 전에는 호출하지 않는다. 그림으로 그려 보자.

그림

Done 인수값에 따라 OnIdle 메시지 전달 방법이 달라진다.



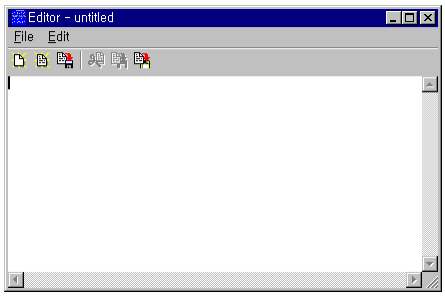

Done 에 True 를 대입하는 왼쪽의 경우를 먼저 보자. 메시지 1 발생 후 다른 메시지가 없으면 OnIdle 이 호출된다. 그리고 이 핸들러에서 True 를 리턴하면 다른 메시지가 들어올 때까지 아무 일도 하지 않고 놀면서 다른 프로그램에게 CPU 시간을 양보한다. 그러다가 메시지 2 가 들어오면 이 메시지를 처리하고 연속해서 들어오는 메시지 3 을 처리한다. 다시 메시지가 들어오지 않으므로 OnIdle 을 호출하고 또 논다. 이 상태는 메시지 4 가 들어올 때까지 지속된다.

Done 에 False 를 대입하는 오른쪽의 경우는 이와 다르다. 메시지 1 발생 후 OnIdle 을 일단 한번 호출한다. 그리고 이 핸들러에서 False 를 리턴하면 계속 이 핸들러를 호출해 주는데 이 상태는 다른 메시지가 들어올 때까지 지속된다. 그러다가 메시지 2, 메시지 3 이 들어오면 처리를 하고 다시 한가한 틈을 타 OnIdle 을 계속 호출해 준다.

그럼 Done 은 어느 경우에 True 가 되고 어느 경우에 False 가 되어야 할까? 일반적으로 두 번 실행하는 것이 의미가 없으면 True 를 주고 계속 실행해야 하면 False 를 준다고 생각하면 된다. 백그라운드 정렬이나 검색의 경우는 계속 실행해야 하므로 False 를 주어야 하고 위와 같이 점을 찍는 작업도 계속 실행해야 하므로 False 를 주어야 한다. Done 에 True 를 대입해 주는 좋은 예는 UI 를 업데이트할 때이다. 메뉴, 툴바 등의 UI 가 변경되는 것은 사용자나 시스템의 변화에 의해 무엇인가가 변경되었을 때 뿐이며 그래서 프로그램이 OnIdle 로 한번 들어왔을 때 딱 한번만 변경해 주면 된다. 두번 세번 검사해 봐야 처리된 메시지가 없으므로 달라질 원인이 전혀 없기 때문이다.

그래서 액션이나 메뉴, 툴바 등의 UI 변경은 메시지를 처리할 때마다 OnIdle

이벤트에서 이루어지며 그렇다고 하더라도 CPU 시간을 전혀 축내지 않는다. 그럼 이쯤에서 퀴즈를 하나 내 보겠는데 해답은 제시하지 않으므로 직접 풀어보기 바란다. 11 장에서 만든 Editor4 예제를 실행시켜 보자.




툴바의 제일 오른쪽에 있는 Paste 버튼은 클립보드에 텍스트가 있으면 Enable 되고 텍스트가 없으면 Disable 된다. Alt+PrtSc 를 눌러 이 폼의 비트맵을 클립보드에 집어 넣으면 이 버튼이 당장 Disable 될 것이다. 그리고 텍스트를 입력한 후 클립보드로 집어 넣으면 다시 이 버튼이 Enable 된다. UI 변경이 제대로 이루어지고 있다. 그런데 이 프로그램 외부에서, 즉 다른 프로그램이 포커스를 가진 상태에서 Alt+PrtSc 를 누르거나 텍스트를 복사하면 이 버튼의 UI 가 전혀 변경되지 않는다. 클립보드에 데이터가 변경되었는데도 말이다. 그런데 또 마우스 커서가 이 윈도우 위를 살짝 지나가면 UI 가 변경된다.

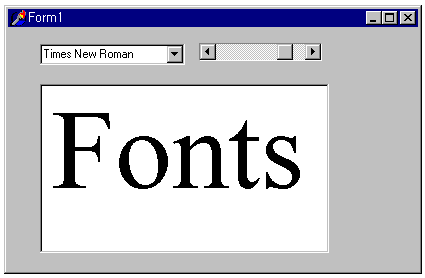
과연 왜 그럴까? 정답을 아는 사람은 mituri@Hitel.net 으로 98 년 12 월 31 일까지 메일을 주면 가장 정확하고 자세한 정답을 제시한 2 명에게 3 만원 상당의 상품을 제공할 예정이다.

# 다. Screen


12jang
Fonts

TScreen 클래스는 응용 프로그램이 실행되고 있는 화면에 대한 정보를 가진다. 메소드나 이벤트도 가지고 있기는 하지만 그보다는 이 오브젝트의 속성을 읽어 프로그램의 실행 환경에 대한 정보를 얻는 용도로 주로 사용된다. Height, Width 속성을 읽으면 현재 화면의 해상도를 알 수 있어 화면 해상도에 따라 프로그램의 크기나 모양이 달라지는 경우에 참고할 만하다. Screen 의 속성 중에 가장 자주 사용되고 실용적인 속성은 Fonts 속성이다. 이 속성은 현재 시스템에 설치된 모든 폰트의 목록을 가지고 있으므로 실행중에 폰트를 변경하고자 할 때 이 속성으로부터 폰트 목록을 구하면 된다.

다음 예제는 메모 컴포넌트의 폰트와 크기를 실행중에 마음대로 바꿀 수 있도록 하였다. 폰트 이름은 콤보 박스에서 선택하고 폰트의 크기는 스크롤 바에서 선택한다.



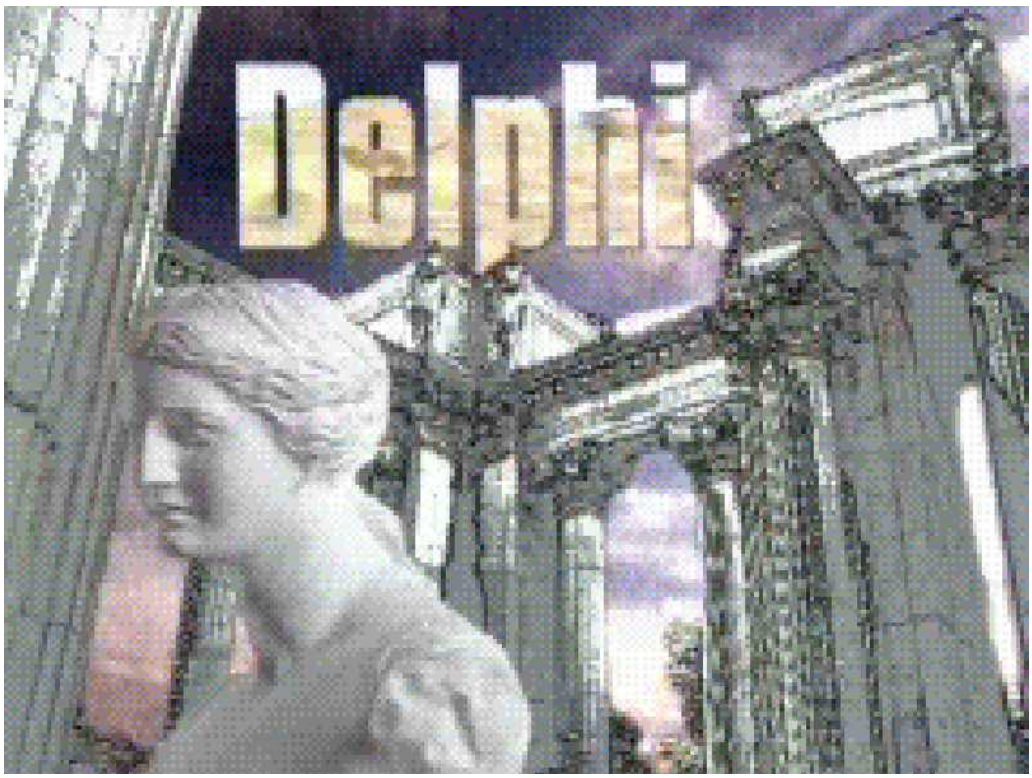

코드는 비교적 간단하다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.Items:=Screen.Fonts;
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  Memo1.Font.Name:=ComboBox1.Items[ComboBox1.ItemIndex];
end;

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Memo1.Font.Size:=ScrollBar1.Position;
end;
```

이 예제의 핵심은 역시 FormCreate에 있는 대입문이다. 콤보 박스에 Screen.Fonts를 대입하여 현재 시스템에 설치된 모든 글꼴의 이름을 대입받았다. 만약 Screen.Fonts 속성이 없다면 이뉴머레이션이라는 복잡한 절차를 통해 폰트 목록을 구해야 한다.

# SDShell 분석

# 제 13 장

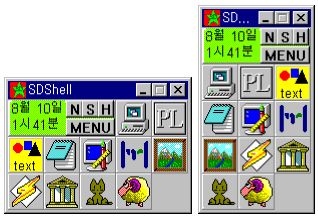


여러분들은 12장까지 델파이의 구석구석을 살펴보고 델파이의 문법을 익히기 위해 여러 가지 예제를 분석해 보고 작성해 보았다. 예제를 분석해 보는 것은 가장 신속하게 언어에 익숙해지는 지름길이기는 하지만 예제는 어디까지나 예제일 수밖에 없다. 부분적인 문법을 익히기 위해 의도적으로 작성된 예제에서는 문법을 익힐 수 있지만 프로그래밍을 익히기란 어렵다. 이 장에서는 하나의 완성된 프로그램을 분석해 봄으로써 델파이 프로그램을 이루는 컴포넌트와 함수, 프로시저 등 여러 가지 부품의 유기적인 연결관계와 프로그램 전체의 구조에 관해 논해 보고자 한다. 초보자는 적당한 길이의 프로그램을 분석하는 과정에서 개별적인 나무가 아닌 전체로서의 숲을 볼 수 있을 것이다.

# 13-1 SDShell이란

## 가. 소개

SDShell은 쉘 프로그램의 일종이다. 쉘이란 다른 프로그램을 실행시킬 수 있는 프로그램을 말하며 프로그램 관리자나 탐색기가 쉘 프로그램이다. SDShell은 좁은 화면에 버튼 형식으로 프로그램을 등록시켜 두고 한 번의 클릭만으로 프로그램을 신속하게 실행시킬 수 있으며 여러 가지 부가 기능들을 제공한다. 폼의 크기를 자유자재로 변경할 수 있어 화면의 상하단이나 좌우 어디든지 배치할 수 있으며 차지하는 공간이 좁아 다른 프로그램이 사용할 화면을 넓게 제공해 준다.




프로그램의 등록은 탐색기에서 간단히 끌어다 떨어뜨리기만 하면 되도록 하여 등록 과정을 최대한 편리하게 구현하였다. 또한 이 프로그램은 홀로 당당히 쉘의 기능을 수행할 수 있다. 옵션에 의해 이 프로그램을 쉘로 지정하면 윈도우즈가 시작되자마자 실행을 시작해서 이 프로그램을 종료하면 윈도우즈가 종료

된다. 쉘로 쓰지 않을 경우는 start up 그룹에 등록해 두고 보조 쉘로 써도 충분히 쓸만한 가치가 있다. Win98의 경우 기본 쉘이 워낙 잘 만들어져 있기 때문에 쉘을 바꿀 필요까지는 없겠지만 보조 쉘은 여전히 필요하다.

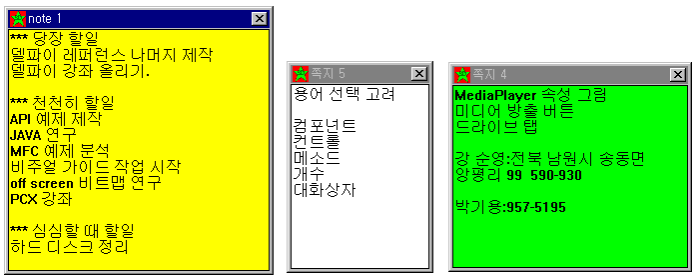
그리고 부가 기능으로 쪽지 기능을 가지고 있어서 데스크탑 상에 임시적인 메모를 해 둘 수 있으며 언제든지 불러와 수정할 수 있다.

# 나. 사용법

별도의 사용법을 설명할 필요도 없이 직관적으로 작성되었으므로 보기만 해도 사용할 수 있을 정도로 사용하기가 쉽다. 프로그램 등록 과정이 드래그 앤 드롭으로 구현되므로 우선 등록 과정이 간단하고 한 번의 클릭만으로 프로그램이 실행되므로 사용 방법도 무척이나 쉬운 편이다. 쪽지 기능은 상단에 배치된 N, S, H 세 개의 버튼을 사용하며 N(New) 버튼이 쪽지 만들기, S(Show) 버튼이 쪽지 보이기, H(Hide) 버튼이 쪽지 숨기기 기능을 가진다. 쪽지 윈도우는 일단 데스크탑 상에 배치되면 위치나 크기를 마음대로 변경할 수 있다.

**그림**

데스크탑 상에 자유롭게 배치할 수 있는 쪽지

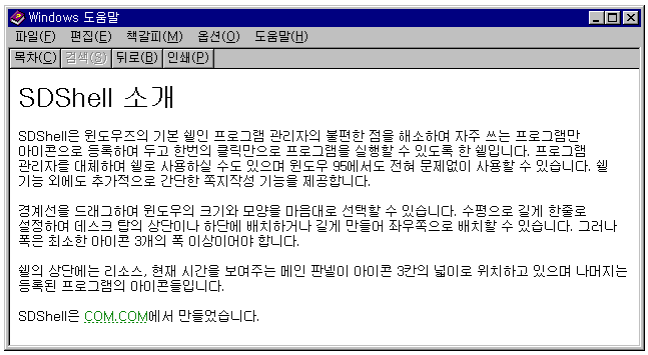



기타 나머지 기능은 팝업 메뉴를 사용한다. 윈도우의 크기를 최소화하기 위해 메인 메뉴는 사용하지 않았다. MENU 버튼을 누르면 나타나는 주 팝업 메뉴와 프로그램 아이콘을 누르면 나타나는 프로그램 팝업 메뉴, 그리고 쪽지 윈도우에 나타나는 쪽지 관리 팝업 메뉴가 있다. 팝업 메뉴 항목의 이름도 척 보면 알 수 있도록 되어 있다.

그림
SDShell 의 스피드 메뉴


그래도 사용법을 잘 모르겠다는 사람을 위해 별도의 도움말도 제공한다. 도움말을 찬찬히 읽어보면 사용법을 더 자세히 알 수 있을 것이다. 도움말은 팝업 메뉴에서 "뭐하는 프로그램일까?"를 선택하면 볼 수 있다.

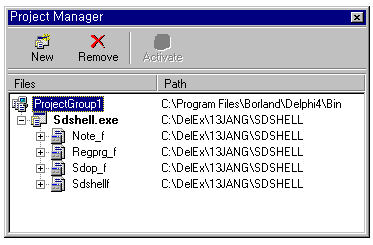
그림
도움말




# 다. 프로젝트의 구성

SDShell은 다음과 같은 네 개의 유닛으로 구성되어 있다.

그림
SDShell 의 프로젝트 구성




별도의 부가 파일이 사용되지 않으므로 유닛과 프로젝트 파일을 같은 디렉토리에 배치해 두고 컴파일하면 된다. 단 도움말 파일은 컴파일과는 상관이 없지만 실행중에 도움말을 보려면 있어야 한다. 결국 완성된 프로그램은 실행 파일

13jang
SDShell

만 있으면 실행할 수 있다.

전체 리스트는 다음과 같다. 필자가 델파이를 처음 배울 때 실습용으로 작성한 것을 4.0버전에 맞게 약간 업그레이드를 했는데 그러다 보니 소스가 좀 지저분하며 16비트 코드의 잔재가 조금 남아있기는 하다. 비교적 자세하게 주석을 달아 놓았으므로 분석이 잘 되지 않는 곳은 주석을 참조하기 바란다. 네 개의 유닛 중에 제일 중요한 메인 폼의 소스만 보이며 나머지 유닛은 본문 중에 필요한 부분만 보인다. 글자가 좀 작아 보기 불편한 사람은 직접 프린터로 출력해서 분석해 보기 바란다.

```
{델파이로 만든 간단한 쉘 프로그램. 버튼에 프로그램을 등록시켜 둔다.
ver 1.21  1996.2.21  김 상형
ver 1,5   1998.8.9 일 수정
}
unit Sdshellf;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons,ShellApi, ExtCtrls,regprg_f, Menus,
  sdop_f, note_f, Registry;

type
  PHWnd = ^HWnd;

{개별적인 버튼의 정보를 기억하는 레코드형}
type
  TArBtn=record
    Panel:Tpanel;   {이미지가 놓일 판넬 }
    Btn:TImage;     {이미지}
    Icon:TIcon;     {아이콘}
    desc:array [0..60] of char;   {프로그램 설명}
    path:array [0..80] of char;   {실행 경로}
    dire:array [0..80] of char;   {시작 디렉토리}
    iconfile:array [0..80] of char; {아이콘을 가지고 있는 파일 }
    iconindex:integer; {파일내에서의 아이콘 번호 }
    MinExec:Boolean;  {최소화 실행}
  end;
{노트에 관한 정보를 기억하는 레코드형}
  TNote=record
    X,Y,W,H:Integer;  {노트의 위치와 크기}
    exist:boolean;    {노트의 존재 여부. 1 이면 있음}
```

```
    notefrm:Tnotepad;  {노트 윈도우. 동적으로 생성됨}
    fontname:string;   {폰트의 이름}
    fontsize:integer;  {폰트의 크기}
    color:TColor      {노트의 색상}
  end;

type
  Tshellform = class(TForm)
    PopupMenu1: TPopupMenu;
    popRun: TMenuItem;
    popDelete: TMenuItem;
    popProperty: TMenuItem;
    restimer: TPanel;
    timergauge: TLabel;
    Timer1: TTimer;
    PopupMenu2: TPopupMenu;
    N1: TMenuItem;
    N2: TMenuItem;
    N3: TMenuItem;
    N4: TMenuItem;
    N5: TMenuItem;
    OpenDialog1: TOpenDialog;
    N6: TMenuItem;
    N7: TMenuItem;
    notenew: TSpeedButton;
    noteshow: TSpeedButton;
    notehide: TSpeedButton;
    menubtn: TSpeedButton;
    Timer2: TTimer;
    N8: TMenuItem;
    N9: TMenuItem;
    N10: TMenuItem;
    N11: TMenuItem;
    DateGauge: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure launch(Sender:TObject);
    procedure btnmousedown (Sender: TObject; Button:TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure btnmouseup (Sender: TObject; Button:TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure makebutton(bn:integer);
    procedure loadini;
    procedure saveini;
    procedure padresize;
    procedure popRunClick(Sender: TObject);
    procedure popDeleteClick(Sender: TObject);
```

```
    procedure FormResize(Sender: TObject);
    procedure popPropertyClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure N2Click(Sender: TObject);
    procedure N1Click(Sender: TObject);
    procedure N4Click(Sender: TObject);
    procedure N5Click(Sender: TObject);
    procedure N7Click(Sender: TObject);
    procedure N6Click(Sender: TObject);
    procedure notenewClick(Sender: TObject);
    procedure noteshowClick(Sender: TObject);
    procedure notehideClick(Sender: TObject);
    procedure loadnote;
    procedure savenote;
    procedure restimerClick(Sender: TObject);
    procedure menubtnClick(Sender: TObject);
    procedure Timer2Timer(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);

  private
    procedure DropFiles(var Msg : TWMDropFiles); message WM_DROPFILES;
    procedure WMNCHitTest(var M:TWMNCHitTest) ; message WM_NCHitTest;
    procedure Syscommand(var M:TWMSYSCommand); message WM_SYSCOMMAND;
    { Private declarations }
  public
    { Public declarations }
  end;

const
  maxprg=64;   {등록가능한 총 프로그램 수 }
  maxnote=20;   {만들 수 있는 최대 노트 갯수}
  HANDCUR=1;

{전역 변수}
var
  shellform: Tshellform;
  ArBtn:array [1..maxprg] of TArBtn;   {프로그램 버튼}
  Note:array [1..maxnote] of TNote;   {노트 }
  totalprg:integer;                {등록된 총 프로그램}
  Icon:TIcon;
  MasterShell:Boolean;   {쉘로 사용되는가를 기억}
  IconPerRow:Integer;   {한줄에 출력될 버튼의 갯수}
  modifiedini:Boolean;   {INI 수정 여부}
  IsDateTimer:Integer;   {시간 날짜 표시기가 있으면 3 없으면 0.현재는 무조건 3}
  IsHint:Boolean;       {풍선 도움말 출력}
```

```
  IsRegWin:Boolean;     {드롭시 등록화면 보여줄 것인가 아닌가}
  IsErrorMsg:Boolean;   {디렉토리 에러 메시지 출력 여부}
  IsRBActivate:Boolean; {마우스 위치에 의한 쉘 활성화}
  IsMinimize:Boolean;   {프로그램 실행시 쉘 최소화}
  IsAlwaysontop:Boolean; {항상 모든 윈도우 위에 위치}
  curdir:string;        {실행파일이 실행되는 디렉토리를 보관할 문자열 변수}
  scrmaxx,scrmaxy:integer;{화면 우하단의 좌표}
  SDReg:TRegIniFile;


implementation

{$R *.DFM}

{조절 메뉴를 더블클릭하여 프로그램을 종료하고자 할 경우.
만약 SDShell 이 쉘로 지정되어 사용되고 있다면 윈도우즈를 종료
하도록 해야 한다. 또한 쉘은 윈도우즈가 종료되지 않는 한은 계속
실행되도록 해 주어야 한다}
procedure Tshellform.Syscommand(var M:TWMSYSCommand);
begin
if (M.CmdType=SC_CLOSE) and (Mastershell) then
  begin
  if MessageDlg('윈도우즈를 끝낼까요?',mtConfirmation,mbOkCancel,0)=mrOk
    Then  begin
    saveini;          {끝내기 전에 정보를 저장한다.}
    ExitWindows(0,0);   {윈도우를 종료한다.}
    end;
  end
else Inherited;
end;

{비 작업 영역을 클릭할 경우 타이틀 바를 드래그 한 것처럼 윈도우즈를
속여 윈도우를 이동시킨다.}
procedure Tshellform.WMNCHitTest(var M:TWMNCHitTest);
begin
inherited;
if M.Result=htClient then
M.Result:=htCaption;
end;

procedure Tshellform.FormCreate(Sender: TObject);
var
  i:integer;
begin
{파일관리자에서 파일을 드래그할 수 있도록 허가한다.}
DragAcceptFiles(Handle, True);
```

```
Screen.Cursors[HANDCUR]:=LoadCursor(hInstance,'HANDCURSOR');
Modifiedini:=False;  {ini 저장할 필요없음}
IsDateTimer:=3;      {날짜, 시계 게이지 있음}
IconPerRow:=4;       {0 으로 나누기 에러 방지}

{화면 우하단의 좌표를 미리 구해 놓음. 여기서 구해 놓은 좌표는
쉘의 위치를 정할 때 사용하며 마우스 위치에 의해 쉘을 활성화할
때도 사용한다.}
scrmaxx:=GetSystemMetrics(SM_CXSCREEN);
scrmaxy:=GetSystemMetrics(SM_CYSCREEN);

{레지스트리에서 쉘의 좌측 및 상단의 좌표를 읽어온다.
비디오 드라이브의 변경에 의해 SDShell 이 현재의 화면 해상도 밖에
위치할 경우는 좌상단으로 위치를 강제로 옮긴다. }
SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDShell');
shellForm.Left:=SDReg.ReadInteger('Pos','Left',100);
shellForm.Top:=SDReg.ReadInteger('Pos','Top',100);
if (shellForm.Left>scrmaxx) or (shellForm.Top>scrmaxy) then
  begin
  shellForm.Left:=100;
  shellForm.Top:=100;
  end;
{레지스트리에서 주요 옵션 설정상태를 읽어온다.ini 파일이 없는 경우를
고려하여 디폴트 옵션을 적절할 상태로 지정하였다. }
  IsRegWin:=SDReg.ReadBool('Pos','IsRegWin',False);
  IsErrorMsg:=SDReg.ReadBool('Pos','IsErrorMsg',False);
  IsHint:=SDReg.ReadBool('Pos','IsHint',True);
  IsRBActivate:=SDReg.ReadBool('Pos','IsRBActivate',True);
  IsMinimize:=SDReg.ReadBool('Pos','IsMinimize',False);
  IsAlwaysontop:=SDReg.ReadBool('Pos','IsAlwaysontop',False);
SDReg.Free;

loadini;   {프로그램 정보를 읽어온다.}

SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDShell');
ClientWidth:=SDReg.ReadInteger('Pos','Width',150);
SDReg.Free;

{쉘이 처음 만들어질 때 시계와 날짜 표시기를 초기화한다.}
Timer1Timer(Sender);
end;

{쉘을 닫을 때 레지스트리에 모든 정보를 저장한다.}
procedure Tshellform.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
```

```
saveini;
CanClose:=True;
end;


{버튼이 클릭될 때 해당 프로그램을 실행시킨다. }
procedure Tshellform.launch(Sender:TObject);
var
  pr:integer;
  i,result:integer;
  myicon:HIcon;
begin
{클릭된 버튼의 번호를 선형 검색으로 구한다.}
for i:=1 to maxprg do
  if Sender=Arbtn[i].Btn then pr:=i;
{프로그램 시작 디렉토리로 위치를 옮긴다. 단 디렉토리가 존재하지 않을
경우의 예외 처리를 해 주어야 에러 메시지를 방지할 수 있다.}
try
chdir(StrPas(ArBtn[pr].dire));
except
if IsErrorMsg Then
  MessageDlg('시작 디렉토리가 없습니다. 그냥 아무데서나 시작할께요',
    mtError,[mbOk],0);
end;
{프로그램 실행시 쉘 최소화}
if IsMinimize Then  Application.Minimize;
{프로그램을 실행한다. 단 이때 실행 파일이 지워져 없을 경우는 winexec 가
32 미만의 값을 리턴하므로 이 경우의 에러처리를 해 준다. 구체적인 에러의
종류는 밝히지 않는다.}
if ArBtn[pr].MinExec then
  result:=winexec(ArBtn[pr].path,SW_MINIMIZE)
else
  result:=winexec(ArBtn[pr].path,SW_SHOWNORMAL);
if result<32 Then MessageDlg('실행 파일 에러로 실행할 수 없습니다.',
    mtError,[mbOk],0);
end;


{ini 파일에서 정보를 읽어와 ArBtn 배열에 저장한다.}
procedure Tshellform.loadini;
var
  pst:array [0..128] of Char;
  pst2:array [0..128] of Char;
  str:string;
  i:integer;
begin
```

```
{만약 레지스트리가 초기화되어 있지 않다면 탐색기만 등록
시킨다. 따라서 이 프로그램은 실행 파일만 있어도 실행할 수 있는 프로그램
이 되며 기동성이 증가한다. 레지스트리 정보의 초기화 점검 여부는 좌측
좌표값이 읽혀지면 초기화된 것으로 간주하고 읽혀지지 않으면 초기화되지
않은 것으로 간주한다.}
SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDShell');
if SDReg.ReadInteger('Pos','Left',10000)=10000 then
 begin
 StrCopy(ArBtn[1].desc,'탐색기');
 GetWindowsDirectory(pst,128);
 StrCopy(ArBtn[1].dire,pst);
 StrCopy(pst2,'₩Explorer.exe');
 StrCopy(ArBtn[1].path,StrCat(pst,pst2));
 StrCopy(ArBtn[1].iconfile,ArBtn[1].path);
 ArBtn[1].iconindex:=0;
 ArBtn[1].MinExec:=False;
 StrCopy(ArBtn[2].path,'null');
 Totalprg:=1;
 modifiedini:=True;
 end

{레지스트리에서 프로그램에 관한 정보를 읽어들인다.
프로그램의 path 가 null 로 설정되어 있으면 그 이하는 정의되지 않은 것으로
간주한다. }
else begin
for i:=1 to maxprg do
 begin
 StrPCopy(ArBtn[i].path,
  SDReg.ReadString('program','path'+inttostr(i),'null'));
 if StrComp(ArBtn[i].path,'null')=0 then break;
 StrPCopy(ArBtn[i].desc,
  SDReg.ReadString('program','desc'+inttostr(i),'null'));
 StrPCopy(ArBtn[i].dire,
  SDReg.ReadString('program','dire'+inttostr(i),'null'));
 StrPCopy(ArBtn[i].iconfile,
  SDReg.ReadString('program','iconfile'+inttostr(i),'null'));
 ArBtn[i].iconindex:=
  SDReg.ReadInteger('program','iconindex'+inttostr(i),0);
 ArBtn[i].MinExec:=
  SDReg.ReadBool('program','MinExec'+inttostr(i),False);
 end;
totalprg:=i-1;
end;

{레지스트리에서 읽어온 갯수만큼 버튼을 초기화 한다}
for i:=1 to totalprg do
```

```
  makebutton(i);
{system.ini 파일에서 shell 의 설정상태를 보고 SDShell 이 쉘로 지정되어 있는가
를 검사하여 Mastershell 변수에 보관한다. 만약 SDShell 의 실행파일 이름을 바꾸어
버리면 제대로 동작하지 못한다.}
getprivateprofilestring('boot','shell','null',pst,128,'system.ini');
str:=ExtractFileName(StrPas(pst));
str:=Copy(Str,1,11);
i:=CompareStr('sdshell.exe',str);
if i=0 Then Mastershell:=True
  else Mastershell:=False;
SDReg.Free;
end;

{레지스트리에 정보를 저장한다.}
procedure Tshellform.saveini;
var
  Pst:String;
  i:integer;
begin
SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDShell');
{쉘의 위치와 폭을 저장한다. 높이는 폭과 등록된 프로그램에 따라
계산 되므로 저장할 필요가 없다.}
SDReg.WriteInteger('Pos','Left',shellForm.Left);
SDReg.WriteInteger('Pos','Top',shellForm.Top);
SDReg.WriteInteger('Pos','Width',shellForm.ClientWidth);

{옵션 설정상태를 저장한다.}
SDReg.WriteBool('Pos','IsHint',IsHint);
SDReg.WriteBool('Pos','IsRegWin',IsRegWin);
SDReg.WriteBool('Pos','IsErrorMsg',IsErrorMsg);
SDReg.WriteBool('Pos','IsRBActivate',IsRBActivate);
SDReg.WriteBool('Pos','IsMinimize',IsMinimize);
SDReg.WriteBool('Pos','IsAlwaysontop',IsAlwaysontop);

savenote;  {노트의 변경된 사항을 저장한다.}

{프로그램 등록 상태가 바뀌어 레지스트리를 수정해야 할 필요가 있을
경우만 프로그램 정보를 기입하도록 한다. 최초 이 코드를 작성한 이유는
프로그램 정보의 양이많아 저장시간을 최대한 절약하기 위한 것이었으나
현재까지는 저장시간이 큰 문제가 되지 않아 무조건 저장하기로 함.
if modifiedini=False then Exit;
}

for i:=1 to totalprg do
  begin
  SDReg.WriteString('program','path'+inttostr(i),ArBtn[i].path);
```

```
  SDReg.WriteString('program','desc'+inttostr(i),ArBtn[i].desc);
  SDReg.WriteString('program','dire'+inttostr(i),ArBtn[i].dire);
  SDReg.WriteString('program','iconfile'+inttostr(i),ArBtn[i].iconfile);
  SDReg.WriteInteger('program','iconindex'+inttostr(i),ArBtn[i].iconindex);
  SDReg.WriteBool('program','MinExec'+inttostr(i),ArBtn[i].MinExec);
  end;
{끝부분에 반드시 null 을 추가하여 몇개의 프로그램이 등록되어 있는가
를 밝혀 주어야 한다. 최초 path 에 null 이 기록된 위치가 등록된
프로그램의 갯수이다. }
SDReg.WriteString('program','path'+inttostr(totalprg+1),'null');
SDReg.WriteString('program','desc'+inttostr(totalprg+1),'null');
SDReg.Free;
end;


{탐색기에서 파일이 드래그되었을 때 드래그된 파일을 등록시킨다.}
procedure Tshellform.DropFiles(var Msg : TWMDropFiles);
var
  FileN:array [0..255] of Char;
  i:integer;
  pst:array [0..128] of char;
  str:string;
begin
if totalprg=maxprg then
  begin
  MessageDlg('최고 64 개까지의 프로그램만 등록할 수 있습니다.',mtError,[mbOK],0);
  DragFinish(Msg.Drop);
  Exit;
end;
{드래그된 파일의 갯수를 조사해 본 후 한개 이상의 파일이 드래그
되었으면 등록을 취소하고 복귀한다. 여러 개의 파일을 한꺼번에 등록할 수는 없다. }
if DragQueryFile(Msg.Drop,Word(-1),FileN,255)>1 then
  begin
  MessageDlg('한번에 하나만 등록할 수 있습니다.',mtError,[mbOK],0);
  DragFinish(Msg.Drop);
  Exit;
  end;
{드래그된 파일의 이름을 얻는다.}
DragQueryFile(Msg.Drop,0,FileN,255);
{등록 정보 윈도우를 호출하여 등록 정보를 입력받는다.
등록되는 정보는 모두 디폴트값이다. }
regprg.editpath.Text:=StrPas(FileN);
regprg.editiconfile.Text:=StrPas(FileN);
regprg.EditIconIndex.Text:='0';
regprg.Editdesc.Text:=ExtractFileName(StrPas(FileN));
str:=ExtractFilePath(StrPas(FileN));
```

```
regprg.Editdire.Text:=Copy(str,1,Length(str)-1);
regprg.CheckBox1.checked:=False;
{등록 정보 보여주기 옵션이 설정되어 있지 않으면 등록 정보를 보여주지
않고 디폴트 옵션을 받아 들인다.}
if IsRegWin then i:=regprg.ShowModal;
if (i<>mrCancel) or (IsRegWin=False) then
 begin
 {프로그램 갯수를 1 증가시킨다.}
 totalprg:=totalprg+1;
 i:=totalprg;
 StrPCopy(ArBtn[i].path,regprg.editpath.Text);
 StrPCopy(ArBtn[i].desc,regprg.Editdesc.Text);
 StrPCopy(ArBtn[i].dire,regprg.Editdire.Text);
 StrPCopy(ArBtn[i].iconfile,regprg.EditIconFile.Text);
 ArBtn[i].IconIndex:=StrToIntDef(regprg.EditIconIndex.Text,0);
 ArBtn[i].MinExec:=regprg.CheckBox1.checked;
 {새 버튼을 하나 만들고 드래그된 버튼을 추가시킨다.}
 makebutton(i);
 modifiedini:=True;
 padresize;
 end;
{드래그 동작이 끝났음을 알린다.}
DragFinish(Msg.Drop);
end;

{새로운 버튼을 하나 만든다. 버튼의 갯수는 등록된 프로그램의 갯수에 따라
가변적이므로 필요할 때마다 동적으로 만든다. }
procedure Tshellform.makebutton(bn:integer);
var
 myicon:HIcon;
 iconfilepath:array [0..128] of char;
begin
 with ArBtn[bn] do begin
  Icon:=TIcon.create;
  {판넬 만듬. 판넬의 위치는 IconPerRow 값과 판넬의 번호에 따라 자동 계산된다.}
  Panel:=TPanel.Create(shellform);
  Panel.Left:=((bn-1+IsDateTimer) mod IconPerRow)*36;
  Panel.Top:=((bn-1+IsDateTimer) div IconPerRow)*36;
  Panel.Width:=36;
  Panel.Height:=36;
  Panel.Parent:=shellform;

  {이미지 만듬. 이 이미지에 프로그램의 아이콘을 출력한다.}
  Btn:=TImage.Create(Panel);
  Btn.Left:=1;
  Btn.Top:=1;
```

```
  Btn.Width:=33;
  Btn.Height:=33;
  Btn.Parent:=Panel;
  Btn.Hint:=Desc;
  Btn.ShowHint:=IsHint;
  Btn.Cursor:=HANDCUR;
  DragAcceptFiles(Btn.canvas.Handle, True);  {버튼도 파일 드롭을 받음}
  with Btn.Canvas do   {버튼에 아이콘을 그리기 위한 준비}
   begin
   Pen.Color:=clSilver;
   Pen.Style:=psSolid;
   Brush.Color:=clSilver;
   Brush.Style:=bsSolid;
   Rectangle(0,0,36,36);
   end;

  {아이콘 파일이 별도로 지정되어 있지 않으면 실행 파일의 아이콘을 사용한다.}
  if StrComp(iconfile,'null')=0 then StrCopy(iconfilepath,path)
   else StrCopy(iconfilepath,iconfile);
  {실행파일에서 아이콘을 추출하되 아이콘이 정의되어 있지 않으면 윈도우즈가
  제공하는 디폴트 아이콘을 사용한다. }
  myIcon:=ExtractIcon(HInstance, iconfilepath,iconindex);
  if myIcon<2 then myIcon:=LoadIcon(0,idi_Question);
  Icon.handle:=myIcon;
  Btn.Canvas.Draw(1,1,Icon);  {이미지에 아이콘 그림}
  {동적으로 생성한 버튼에 팝업 메뉴와 이벤트 핸들러를 지정해 준다. }
  Btn.Popupmenu:=popupmenu1;
  Btn.OnClick:=launch;
  Btn.OnMouseDown:=Btnmousedown;
  Btn.OnMouseUp:=Btnmouseup;
 end;
end;

{팝업 메뉴에서 실행 항목을 선택할 경우 팝업 메뉴를 부른 버튼을
실행시킨다.}
procedure Tshellform.popRunClick(Sender: TObject);
begin
launch(Popupmenu1.PopupComponent);
end;

{프로그램 삭제. 아예 뒷부분을 없앴다가 다시 만든다.}
procedure Tshellform.popDeleteClick(Sender: TObject);
var
 i, pr:integer;
begin
{삭제 대상이 되는 아이콘의 번호를 찾는다.}
```

```
for i:=1 to totalprg do
  if Popupmenu1.PopupComponent=ArBtn[i].Btn then pr:=i;
for i:=pr to totalprg do
  begin
  StrCopy(ArBtn[i].desc,ArBtn[i+1].desc);
  StrCopy(ArBtn[i].path,ArBtn[i+1].path);
  StrCopy(ArBtn[i].dire,ArBtn[i+1].dire);
  StrCopy(ArBtn[i].iconfile,ArBtn[i+1].iconfile);
  ArBtn[i].iconindex:=ArBtn[i+1].iconindex;
  ArBtn[i].MinExec:=ArBtn[i+1].MinExec;
  end;
{오브젝트를 전부 없앴다가 다시 만든다. 이때 오브젝트를 해제하는 순서에
주의해야 한다.Btn 의 페어런트가 Panel 이므로 Btn 을 먼저없앤 후 Panel 을
없애야 한다. 그렇지 않으면 GPF 가 발생한다. }
for i:=pr to totalprg do
  with ArBtn[i] do begin
  Icon.Free;
  Btn.Free;
  Panel.Free;
  end;
{프로그램 갯수를 하나 감소시킨 후 다시 등록한다}
Dec(totalprg);
for i:=pr to totalprg do
  makebutton(i);
modifiedini:=True;
padresize;
end;

{버튼 위에서 마우스를 누를 경우 버튼을 우하단으로 1 픽셀만큼 이동시킨다.}
procedure Tshellform.btnmousedown (Sender: TObject; Button:TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
(Sender as TImage).Left:=(Sender as TImage).Left+1;
(Sender as TImage).Top:=(Sender as TImage).Top+1;
((Sender as TImage).Parent as TPanel).BevelOuter:=BvLowered;
end;
{마우스 버튼을 놓을 경우 원위치시킨다.}
procedure Tshellform.btnmouseup (Sender: TObject; Button:TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
(Sender as TImage).Left:=(Sender as TImage).Left-1;
(Sender as TImage).Top:=(Sender as TImage).Top-1;
((Sender as TImage).Parent as TPanel).BevelOuter:=BvRaised;
end;
```

```
{버튼의 갯수에 따라 폼의 크기를 재 조정한다.}
procedure Tshellform.padresize;
begin
ClientHeight:=(((totalprg-1+IsDateTimer) div IconperRow)+1)*36;
ClientWidth:=36*IconPerRow;
end;

{폼의 크기가 조정될 때 폼의 수평 크기에 맞게 줄당 아이콘이 갯수를
설정한다. IconperRow 를 여기서 결정하면 폼의 크기가 padresize
함수에서 다시 조정된다.}
procedure Tshellform.FormResize(Sender: TObject);
var
  i:integer;
begin
IconPerRow:=ClientWidth div 36;
{총 프로그램 갯수 이하이거나 3 이상이어야 한다.}
if IconPerRow>totalprg+IsDateTimer Then IconPerRow:=totalprg+IsDateTimer;
if IconperRow<3 Then IconperRow:=3;
{아이콘의 위치를 이동시킨다.}
for i:=1 to totalprg do
  with ArBtn[i] do begin
  Panel.Left:=((i-1+IsDateTimer) mod IconPerRow)*36;
  Panel.Top:=((i-1+IsDateTimer) div IconPerRow)*36;
  end;
{폼의 크기를 다시 조정한다.}
padresize;
end;

{팝업 메뉴의 등록 정보 변경 항목}
procedure Tshellform.popPropertyClick(Sender: TObject);
var
  i, pr:integer;
begin
for i:=1 to maxprg do
  if Popupmenu1.PopupComponent=ArBtn[i].Btn then pr:=i;
{등록 정보 윈도우를 호출하여 등록 정보를 입력받는다.}
regprg.editpath.Text:=StrPas(ArBtn[pr].path);
regprg.Editdesc.Text:=StrPas(ArBtn[pr].desc);
regprg.Editdire.Text:=StrPas(ArBtn[pr].dire);
regprg.EditIconFile.Text:=StrPas(ArBtn[pr].IconFile);
regprg.EditIconIndex.Text:=IntToStr(ArBtn[pr].IconIndex);
regprg.CheckBox1.checked:=ArBtn[pr].MinExec;
if regprg.ShowModal<>mrCancel then
{등록 정보가 바뀔 경우 아예 버튼을 다시 만든다.}
  begin
  StrPCopy(ArBtn[pr].path,regprg.editpath.Text);
```

```
  StrPCopy(ArBtn[pr].desc,regprg.Editdesc.Text);
  StrPCopy(ArBtn[pr].dire,regprg.Editdire.Text);
  StrPCopy(ArBtn[pr].IconFile,regprg.EditIconFile.Text);
  ArBtn[pr].MinExec:=regprg.CheckBox1.checked;
  ArBtn[pr].IconIndex:=StrToIntDef(regprg.EditIconIndex.Text,0);
  ArBtn[pr].Icon.Free;
  ArBtn[pr].Btn.Free;
  ArBtn[pr].Panel.Free;
  makebutton(pr);
  modifiedini:=True;
  end;
end;

{5 초 간격으로 시간, 날짜 표시기를 갱신한다.}
procedure Tshellform.Timer1Timer(Sender: TObject);
 var
  H,M,S,MS:Word;
  Mo,D,Y:Word;
begin
DecodeTime(Time,H,M,S,MS);
if H>12 Then H:=H-12;
timergauge.caption:=Format('%d 시%d 분',[H,M]);
DecodeDate(Date,Y,Mo,D);
DateGauge.caption:=Format('%d 월 %d 일',[Mo,D]);
end;

{폼이 출력되기 전에 현재 디렉토리가 어디인가를 구해 놓는다.}
procedure Tshellform.FormShow(Sender: TObject);
begin
curdir:=ParamStr(0);
curdir:=ExtractFilePath(curdir);
loadnote;
end;

{도움말을 보여준다.}
procedure Tshellform.N2Click(Sender: TObject);
begin
WinHelp(handle,'sdshell.hlp',HELP_CONTENTS,0);
end;

{옵션 대화상자를 보여준다.}
procedure Tshellform.N1Click(Sender: TObject);
begin
sdoption.showmodal;
end;
```

```
{SDShell 을 종료한다. 만약 쉘로 지정되어 있으면 윈도우즈를 끝낸다.}
procedure Tshellform.N4Click(Sender: TObject);
begin
if Mastershell then
  begin
  if MessageDlg('윈도우즈를 끝낼까요?',mtConfirmation,mbOkCancel,0)=mrOk
    Then  begin
    saveini;
    ExitWindows(0,0);
    end;
  end
else close;
end;

{직접 프로그램 이름을 입력받아 실행하기}
procedure Tshellform.N5Click(Sender: TObject);
var
  Pst:array [0..128] of Char;
begin
if OpenDialog1.Execute Then
 strPCopy(pst,OpenDialog1.FileName);
 Winexec(pst,SW_SHOWNORMAL);
end;

{새로운 프로그램을 등록한다.}
procedure Tshellform.N7Click(Sender: TObject);
var
  i:Integer;
begin
{등록 정보 윈도우를 호출하여 등록 정보를 입력받는다.}
regprg.editpath.Text:='프로그램의 이름을 입력하세요';
regprg.Editdesc.Text:='간단한 설명을 입력하세요';
regprg.Editdire.Text:='c:₩windows';
if regprg.ShowModal<>mrCancel then
  begin
 {프로그램 갯수를 1 증가시킨다.}
  totalprg:=totalprg+1;
  i:=totalprg;
  StrPCopy(ArBtn[i].path,regprg.editpath.Text);
  StrPCopy(ArBtn[i].desc,regprg.Editdesc.Text);
  StrPCopy(ArBtn[i].dire,regprg.Editdire.Text);
  StrPCopy(ArBtn[i].iconfile,regprg.EditIconFile.Text);
  ArBtn[i].IconIndex:=StrToIntDef(regprg.EditIconIndex.Text,0);
  ArBtn[i].MinExec:=regprg.CheckBox1.checked;
 {새 버튼을 하나 만들고 드래그된 버튼을 추가시킨다.}
  makebutton(i);
```

```
 modifiedini:=True;
 padresize;
 end;
end;

{윈도우 재 시작 명령 }
procedure Tshellform.N6Click(Sender: TObject);
begin
 if MessageDlg('윈도우즈를 다시 시작할까요?',
  mtConfirmation,mbOkCancel,0)=mrOk
  Then  begin
  saveini;
  ExitWindows(EW_RESTARTWINDOWS,0);
  end;
end;

{새로운 노트를 만든다. 노트를 만들 번호는 최초로 존재하지 않는 노트의 번호
를 찾아 덮어쓰는 형식을 취한다. 만약 노트가 20 개 만들어져 있으면 더 이상
노트를 만들 수 없다. 노트에 대해서는 별도의 윈도우를 만들어 주고 속성을 디폴트
로 설정해 준다. }
procedure Tshellform.notenewClick(Sender: TObject);
var i:integer;
begin
{최초의 존재하지 않는 노트 번호를 조사한다.}
for i:=1 to maxnote do
 if Note[i].exist=False then break;
if i=maxnote then begin
 MessageDlg('노트는 최대 20 개까지 만들 수 있습니다.',mtError,[mbOk],0);
 exit;
 end;
note[i].exist:=True;
{실행중에 동적으로 폼을 만든다.}
Application.CreateForm(Tnotepad,note[i].notefrm);
note[i].notefrm.caption:='쪽지 '+IntToStr(i);
note[i].notefrm.Left:=i*30;
note[i].notefrm.Top:=i*20;
note[i].notefrm.Show;
end;

{노트를 보여준다. 모든 노트를 검사하여 존재하는 모든 노트를 보여준다.}
procedure Tshellform.noteshowClick(Sender: TObject);
var i:integer;
begin
for i:=1 to maxnote do
 if note[i].exist then note[i].notefrm.Show;
end;
```

```
{노트를 숨긴다. 모든 노트를 검사하여 존재하는 모든 노트를 숨긴다.}
procedure Tshellform.notehideClick(Sender: TObject);
var i:integer;
begin
for i:=1 to maxnote do
  if note[i].exist then note[i].notefrm.Hide;
end;

{레지스트리에서 노트를 읽어들인다. 먼저 Exist 항목을 점검하여 노트가 존재
하는지 살펴보고 존재하는 노트만 읽어들인다. 읽어들인 노트에 대해서는 별도
의 윈도우를 만들어 준다. }
procedure Tshellform.loadnote;
var
  i:integer;
  Pst:array [0..128] of Char;
  sysdir:array [0..128] of Char;
  str:string;
begin
SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDShell');
for i:=1 to maxnote do begin
  if SDReg.ReadBool('note'+IntToStr(i),'exist',False)=False then begin
    note[i].exist:=False;
    continue;
  end;
  note[i].exist:=True;
  Application.CreateForm(Tnotepad,note[i].notefrm);
  note[i].notefrm.Left:=
    SDReg.ReadInteger('note'+IntToStr(i),'X',10);
  note[i].notefrm.Top:=
    SDReg.ReadInteger('note'+IntToStr(i),'Y',10);
  note[i].notefrm.Width:=
    SDReg.ReadInteger('note'+IntToStr(i),'W',10);
  note[i].notefrm.Height:=
    SDReg.ReadInteger('note'+IntToStr(i),'H',10);

  {노트의 폰트 이름과 크기를 읽어 온다. }
  note[i].notefrm.Memo1.Font.Size:=
    SDReg.ReadInteger('note'+IntToStr(i),'fontsize',10);
  note[i].notefrm.Memo1.Font.Name:=
    SDReg.ReadString('note'+IntToStr(i),'fontname','system');

  {노트의 색상 정보를 읽어 온다.16 진수로 기록된 정수값을 읽어들인다.}
  note[i].notefrm.Memo1.Color:=
    StrToInt('$'+SDReg.ReadString('note'+IntToStr(i),'color','0000FFFF'));
```

```
 {노트의 제목을 읽는다}
 note[i].notefrm.caption:=
  SDReg.ReadString('note'+IntToStr(i),'caption','제목없음');

 {윈도우즈의 시스템 디렉토리에서 노트 파일을 읽어 들이기 위해 시스템
 디렉토리가 어디인가를 조사한다. }
 GetSystemDirectory(sysdir,128);
 str:=StrPas(sysdir);
 str:=str+'₩sdnote.'+IntToStr(i);
 {노트 파일이 없는 경우를 고려하여 에러 메시지를 출력해 준다.}
 try
  note[i].notefrm.Memo1.Lines.LoadFromFile(str);
 except
  MessageDlg('노트 파일이 손상되었습니다.',mtError,[mbOk],0);
 end;
end;
SDReg.Free;
end;

{노트를 레지스트리에 저장한다. 존재하는 노트에 대해서 모든 변화된 사항을
일일이 기록한다.}
procedure Tshellform.savenote;
var
 i:integer;
 Pst:array [0..128] of Char;
 imsimemo:array [0..128] of Char;
 sysdir:array [0..128] of Char;
 str:string;
begin
for i:=1 to maxnote do begin
 if note[i].exist=False then continue;
 SDReg.WriteInteger('note'+IntToStr(i),'X',note[i].notefrm.Left);
 SDReg.WriteInteger('note'+IntToStr(i),'Y',note[i].notefrm.Top);
 SDReg.WriteInteger('note'+IntToStr(i),'W',note[i].notefrm.Width);
 SDReg.WriteInteger('note'+IntToStr(i),'H',note[i].notefrm.Height);
 SDReg.WriteBool('note'+IntToStr(i), 'exist', True);

 {노트의 폰트 이름과 크기를 저장한다.}
 SDReg.WriteInteger('note'+IntToStr(i),
  'fontsize',note[i].notefrm.Memo1.Font.Size);
 SDReg.WriteString('note'+IntToStr(i),
  'fontname',note[i].notefrm.Memo1.Font.Name);

 {노트의 색상값을 16 진수로 바꾸어 기록한다.}
 SDReg.WriteString('note'+IntToStr(i),'color',
  IntToHex(Integer(note[i].notefrm.Memo1.color),8));
```

```
  {노트의 제목을 저장한다.}
  SDReg.WriteString('note'+IntToStr(i),
    'caption',note[i].noteFrm.Caption);

  if not (note[i].notefrm.Memo1.Modified) then continue;
  {윈도우즈의 시스템 디렉토리에 노트 파일을 작성하기 위해 시스템 디렉토리가
  어디인가를 조사한다.}
  GetSystemDirectory(sysdir,128);
  str:=StrPas(sysdir);
  str:=str+'₩sdnote.'+IntToStr(i);
  note[i].notefrm.Memo1.Lines.SaveToFile(str);
end;
end;

{날짜 시간 표시기를 왼쪽 클릭하면 메인 팝업 메뉴를 출력한다.}
procedure Tshellform.restimerClick(Sender: TObject);
begin
popupmenu2.popup(shellform.Left+30,shellform.Top+32);
end;

{ 메뉴 버튼을 누를 경우 팝업 메뉴가 강제로 나타나도록 한다. }
procedure Tshellform.menubtnClick(Sender: TObject);
begin
popupmenu2.popup(shellform.Left+30,shellform.Top+60);
end;

{타이머를 이용하여 이 루틴을 1 초에 한번씩 호출한다.
마우스 커서가 우하단에 위치하고 있을 경우 SDShell 을 활성화시킨다.}
procedure Tshellform.Timer2Timer(Sender: TObject);
var
  CPOS:TPOINT;
begin
GetCursorPos(CPOS);
if (CPOS.x>scrmaxx-4) and (CPOS.y>scrmaxy-4) and IsRBActivate then
  begin
  BringWindowToTop(handle);
  SetActiveWindow(handle);
  SetForegroundWindow(handle);
  end;
end;


end.
```

# 13-2 프로그램 관리

## 가. ArBtn 레코드

SDShell은 프로그램마다 버튼을 하나씩 할당해 주므로 버튼 하나가 곧 프로그램 하나와 대응된다. 등록된 프로그램의 등록 정보와 내부적인 정보 관리를 위해 ArBtn이라는 레코드 배열을 사용한다. TArBtn 레코드는 프로그램 선두에 다음과 같이 정의되어 있다.

```
{개별적인 버튼의 정보를 기억하는 레코드형}
type
 TArBtn=record
  Panel:Tpanel;   {이미지가 놓일 패널 }
  Btn:TImage;     {이미지}
  Icon:TIcon;     {아이콘}
  desc:array [0..60] of char;   {프로그램 설명}
  path:array [0..80] of char;   {실행 경로}
  dire:array [0..80] of char;   {시작 디렉토리}
  iconfile:array [0..80] of char;{아이콘을 가지고 있는 파일}
  iconindex:integer; {파일 내에서의 아이콘 번호 }
  MinExec:Boolean;   {최소화 실행}
 end;
```

각 필드별로 의미를 정리해 보자.

### ☐ Panel

버튼의 아이콘을 저장할 패널이다. 아이콘은 이미지 컴포넌트에 출력되며 이미지 컴포넌트는 패널에 소속된다. 이미지 컴포넌트만으로 버튼을 만들 수 있지만 굳이 패널을 사용하는 이유는 컨테이너 컴포넌트인 패널을 사용하면 버튼의 위치를 옮기기가 쉬우며 버튼의 가장자리에 음각, 양각의 입체 효과를 줄 수 있기 때문이다.

### ☐ Btn

프로그램의 아이콘을 담을 이미지 컴포넌트이다. 이미지의 표면에 실행 파일

의 아이콘을 추출하여 출력한다. 버튼 컴포넌트를 사용하는 것이 더 적당할 것 같지만 버튼에는 그림을 출력할 수 없으며 설사 BitBtn을 사용하더라도 BMP 파일만 출력할 수 있을뿐 아이콘은 출력할 수 없어 이미지를 사용하였다.

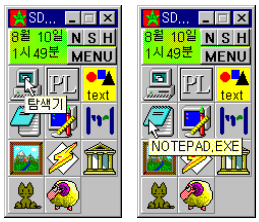
### ☐ Icon

프로그램의 아이콘을 담는다. 실행 파일의 아이콘을 저장해 두었다가 이미지 컴포넌트의 표면에 그릴 때 사용한다.

### ☐ desc

description의 준말이며 프로그램에 관한 짧은 설명 문자열을 담는다. 설명문은 프로그램 등록 과정에서 사용자에 의해 입력되어지며 사용자가 입력하지 않을 경우는 디폴트로 실행 파일의 이름을 설명문으로 사용한다. 설명문은 아이콘 위에서 풍선 도움말로 출력되어 아이콘을 일일이 기억하지 못하는 사람의 기억력을 보조해 준다.

그림
아이콘에 나타나는 풍선 도움말




### ☐ path

프로그램의 실행 파일 이름과 경로를 기억하며 쉘 프로그램이 사용하는 가장 중요한 정보이다. 드라이브명과 디렉토리 정보까지 포함하는 완전 경로(full path)를 가지며 드래그 앤 드롭에 의해 자동으로 입력되므로 사용자가 직접 입력해야 할 경우는 드물다.

### ☐ dire

프로그램 실행을 시작할 디렉토리를 지정한다. 디폴트로 실행 파일이 있는 디렉토리를 사용하지만 별도의 디렉토리를 지정할 수 있다. 워드 프로세서나 스프레드 시트와 같이 데이터 파일이 있는 프로그램은 반드시 시작 디렉토리를 지정할 수 있어야 제대로 된 쉘이라고 할 수 있다.

### ☐ iconfile

프로그램의 아이콘은 보통 실행 파일에 있는 아이콘을 사용하지만 아이콘이 없는 실행 파일의 경우는 별도로 아이콘을 제공받을 실행 파일을 지정할 수 있다. 윈도우즈 프로그램들은 모두 아이콘을 가지지만 도스 프로그램들은 아이콘

이 정의되어 있지 않으므로 다른 실행 파일에 정의된 아이콘을 사용해야 하며 이런 목적으로 프로그램 관리자와 MORICONS.DLL에 다수의 아이콘을 제공한다.

### ☐ iconindex

실행 파일에 정의된 아이콘 중 몇 번째 아이콘을 사용할 것인가를 지정한다. 실행 파일 하나에 보통 하나의 아이콘이 정의되지만 여분으로 여러 개의 아이콘을 가지는 경우도 있으므로 그 중 어떤 아이콘을 사용할 것인가를 선택해야 한다.

### ☐ MinExec

프로그램을 최소화하여 실행할 것인가를 기억한다. 별로 잘 사용되지 않는 필드이지만 모든 쉘이 다 이 기능을 포함하고 있으므로 같이 포함하였다.

ArBtn 레코드의 내용은 프로그램을 등록할 때 작성되고 프로그램을 실행할 때 참조하며 등록 내용을 수정할 때도 수정의 대상이 된다. 즉 이 프로그램 전체를 통제하는 중요한 레코드이므로 이 레코드의 내용을 일단은 다 외워둘 필요가 있다. TArBtn형의 레코드 타입을 정의한 후 전역 변수로 ArBtn배열을 선언한다.

```
var
  shellform: Tshellform;
  ArBtn:array [1..maxprg] of TArBtn;  {프로그램 버튼}
```

ArBtn 배열의 크기는 maxprg로 정의되어 있으며 maxprg는 상수 64로 정의되어 있다. 그래서 SDShell에 등록할 수 있는 프로그램 개수는 64개까지이다. 64개 정도면 웬만한 프로그램은 다 등록할 수 있을 정도로 충분한 개수이다. 만약 부족하다면 maxprg 상수를 수정하면 더 많은 수의 프로그램을 등록할 수 있다. 그런데 64개 이상의 프로그램을 등록해 보면 아이콘 찾기가 장난이 아니므로 별로 권할만한 일은 아니다.

## 나. 등록

프로그램을 등록하는 방법에는 우선 점잖게 대화상자를 이용하여 정보를 일일이 기입해 주는 방법을 생각할 수 있다. 등록 대화상자는 팝업 메뉴에서 프로그램 등록 항목을 선택하여 호출한다.

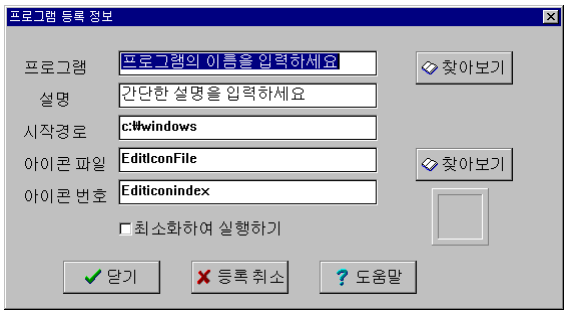
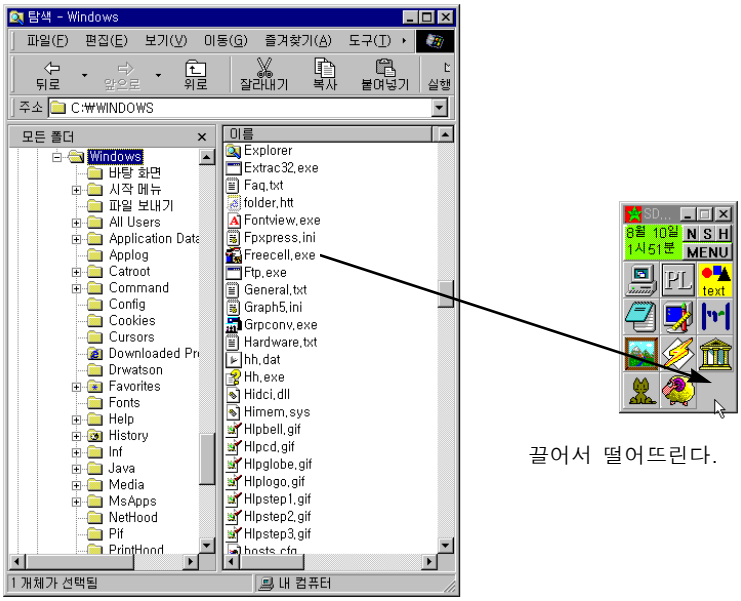



**그림**
프로그램을 등록하는 등록 대화상자

하지만 이 많은 정보를 일일이 기입해 준다는 것은 보통 귀찮은 일이 아니다. 그래서 그 대안으로 드래그 앤 드롭을 제공한다. 탐색기에서 파일을 끌어다 SDShell에 떨어뜨리기만 하면 가장 무난한 디폴트 정보를 직접 생성해서 등록해 준다.




끌어서 떨어뜨린다.

**그림**
탐색기에서 프로그램을 드래그하여 프로그램을 등록한다.

이 기능을 사용하려면 먼저 드래그된 파일을 드롭받겠다는 신고를 해야 한다. 아무 폼이나 파일을 드롭받을 수는 없다. 신고는 DragAcceptFiles 프로시저로 한다.

```
procedure DragAcceptFiles(Wnd: HWnd; Accept: Bool);
```

Wnd가 신고의 대상이 되는 폼의 윈도우 핸들이며 Accept가 True이면 드롭

을 받겠다는 뜻이며 False이면 드롭을 받지 않는다는 뜻이다. 이 함수로 드롭을 받겠다는 신고를 하면 윈도우즈의 탐색기(EXPLORER.EXE)는 파일이 해당 윈도우로 드롭될 때 WM_DROPFILES 메시지를 전달해 준다. SDShell은 FormCreate 이벤트 핸들러에서 폼이 처음 생성될 때 이 함수를 호출해 준다.

```
DragAcceptFiles(Handle, True);
```

그리고 폼뿐만 아니라 폼에 생성되는 개별 버튼도 드롭되는 파일을 받을 수 있도록 makebutton 함수에서 각 이미지 컴포넌트에 대해서도 이 함수를 호출한다.

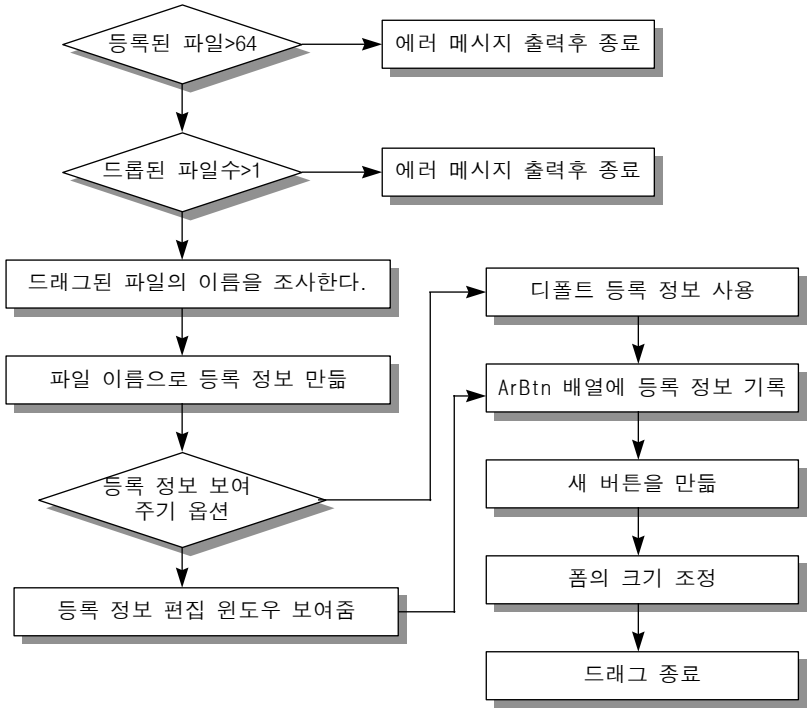
```
DragAcceptFiles(Btn.canvas.Handle, True);
```

이미지의 캔버스도 윈도우이므로 파일을 드롭받을 수 있다. 이제 드롭 허가 신고를 했으므로 사용자가 파일을 드래그하여 드롭할 때마다 SDShell은 WM_DROPFILES 메시지를 받게 될 것이다. 이 메시지에 대한 처리는 DropFiles 함수에서 수행하도록 폼의 타입 선언에서 지정하고 있다.

```
private
  procedure DropFiles(var Msg : TWMDropFiles); message WM_DROPFILES;
```

message 지시자는 윈도우즈가 보내는 메시지에 대한 처리 함수를 지정하는 역할을 한다. 위의 프로시저 선언문은 WM_DROPFILES 메시지가 전달될 경우 이 메시지를 Msg에 저장한 후 DropFiles 프로시저로 보낸다는 의미이다. 그럼 DropFiles 함수에서는 어떤 일이 벌어지는지 순서도를 살펴보자.

그림
드래그에 의한 프로그램 등록 절차




파일이 드래그되면 일단 프로그램이 등록될 수 있는 상태인가 두 가지 조건 점검을 수행한다. 우선 등록 최대 개수인 64개를 초과했는지 점검해 보고 더 이상 등록할 공간이 없으면 종료한다. 그리고 두 번째로 드롭된 파일이 한 개인가를 점검해 본다. 탐색기는 여러 개의 파일을 한꺼번에 드래그할 수 있도록 되어 있지만 프로그램 등록은 한 번에 하나씩만 할 수 있으므로 이 경우도 등록을 할 수 없다. 두 가지 조건이 만족되면 등록을 한다. 우선 드롭된 파일의 개수와 이름을 조사하기 위해 DragQueryFile 함수를 호출한다.

```
function DragQueryFile(Drop: THandle; FileIndex: Word;
    FileName: PChar; cb: Word): Word;
```

각 인수의 의미는 다음과 같다.

### ☐ Drop

윈도우즈로부터 전달된 정보의 ID이며 WM_DROPFILES 메시지가 전달될 때 wParam으로 전달된다. 사용자는 MSG.Drop을 이 인수로 넘겨주기만 하면 된다.

☐ FileIndex

이름을 조사할 파일의 번호이다. 여러 개의 파일이 드롭될 수도 있으므로 몇 번째 파일의 이름을 조사할 것인가를 밝혀 주어야 한다. 이 값이 -1이면 파일의 이름을 조사하는 대신에 드롭된 파일의 개수를 조사해 준다. 이 인수가 0이면 드롭된 첫 번째 파일의 이름을 조사한다.

☐ FileName

조사된 파일 이름을 담을 문자열 포인터이다. 즉 DragQueryFile 함수가 조사한 결과를 돌려주는 출력용 인수이다.

☐ cb

FileName 인수의 길이이다.

드롭된 파일의 이름을 조사했으면 이 이름으로 디폴트 정보를 만든다. 파일 이름은 완전 경로(full path)이므로 실행 파일 이름뿐만 아니라 시작 디렉토리, 설명, 아이콘 파일까지도 파일 이름으로부터 유추해낼 수 있다. 등록 정보를 만든 후 등록 정보 윈도우의 각 에디트 박스에 대입해 주고 사용자에게 수정할 기회를 준다. 옵션 설정에 따라 등록 정보 윈도우를 보여 주지 않고 곧바로 디폴트 값을 취할 수도 있다. 등록 정보가 다 만들어졌으면 ArBtn 배열에 등록 정보를 작성하고 makebutton 함수를 호출하여 버튼을 새로 생성한다. 버튼 개수가 하나 늘어남에 따라 폼의 크기가 다시 조정되어야 하며 그래서 padresize 함수가 호출된다.

# 다. 버튼 만들기

SDShell에 등록되는 프로그램의 개수는 가변적이다. 사용하는 사람에 따라 한 개만 등록해 놓을 수도 있고 64개를 다 등록해 놓고 사용할 수도 있다. 프로그램 하나는 버튼 하나를 차지하며 등록될 프로그램의 개수가 가변적이므로 필요한 버튼의 개수도 가변적이다. 그래서 디자인시에 미리 버튼을 만들어 둘 수가 없으며 실행중에 동적으로 버튼을 생성시켜야 한다. 물론 64개를 몽땅 만들어 두는 아주 무식한 방법을 쓸 수도 있지만 메모리와 리소스를 낭비하는 멍청한 짓이다. 버튼을 동적으로 생성시키는 일은 makebutton 함수에서 수행한다.

makebutton 함수는 생성한 버튼의 번호를 인수 bn으로 전달받으며 bn을 ArBtn 배열의 첨자로 사용하여 등록 정보를 알아내며 이 정보를 참조하여 버튼

을 만든다.

동적으로 생성해야 할 대상은 패널, 이미지 컴포넌트 및 아이콘 오브젝트이며 차례로 이 세 개의 오브젝트를 Create 메소드를 사용하여 생성한다. 그리고 생성한 오브젝트의 속성을 설정한다. 속성 설정 코드 중 눈여겨 볼 것은 패널의 위치, 곧 버튼의 위치이다.

```
Panel.Left:=((bn-1+IsResTimer) mod IconPerRow)*36;
Panel.Top:=((bn-1+IsResTimer) div IconPerRow)*36;
```

패널의 위치는 패널의 번호에 따라 결정되며 한 줄에 몇 개의 아이콘이 배치되는지 IconPerRow 변수값을 참조하여 버튼을 배열한다. 위 식에서 bn에서 1을 빼주는 이유는 버튼 번호가 1부터 시작하기 때문에 처음 버튼 위치를 0으로 맞추기 위해서이며 IsResTimer는 날짜와 시간을 나타내는 버튼 3칸짜리의 제어판을 말하며 이 프로그램의 경우 3으로 고정되어 있다. 계산해 보면 알겠지만 버튼의 번호에 따라 다음과 같이 배치된다.


```
          1  2
3  4  5  6  7
8  9  10 11
```

버튼의 배치 상태는 폼의 폭이 변해도 항상 좌에서 우로 위에서 아래로 배치되도록 되어 있다.  실행 파일에서 아이콘을 추출할 때는 ExtractIcon API 함수를 사용한다.

```
function ExtractIcon(Inst: THandle; ExeFileName: PChar;
  IconIndex: Word): HIcon;
```

인스턴스 핸들과 실행 파일 이름, 그리고 아이콘의 번호를 인수로 넘겨주면 아이콘의 핸들을 리턴해 준다. 아이콘 핸들을 받아 이미지 컴포넌트의 캔버스 표면에 Draw 메소드를 사용하여 그려 주되 아이콘이 정의되지 않은 프로그램의 경우는 디폴트 아이콘을 취하도록 한다.

아이콘까지 그렸으면 버튼은 모두 만들어졌으며 마지막으로 실행중에 버튼이 클릭될 경우 프로그램을 실행시키기 위해 버튼의 이벤트 핸들러를 지정해 준다.

```
Btn.OnClick:=launch;
```

```
Btn.OnMouseDown:=Btnmousedown;
Btn.OnMouseUp:=Btnmouseup;
```

동적으로 생성된 컴포넌트이므로 이벤트 핸들러도 실행중에 동적으로 지정되어야 한다.

# 라. 실행

프로그램의 실행은 launch 함수에서 수행한다. 이 함수는 동적으로 생성된 버튼의 OnClick 이벤트 핸들러로 지정되어 있으므로 아무 버튼이나 클릭되기만 하면 호출된다. 그래서 이 함수에서 제일 먼저 해야 할 일은 많고도 많은 버튼 중에서 자신을 호출한 버튼(=실행의 대상이 되는 프로그램)이 어떤 버튼인가를 조사해 보는 일이며 이 일은 인수로 전달된 Sender와 ArBtn 배열의 Btn 필드를 비교하여 ArBtn 배열의 첨자를 구함으로써 해결하였다. 일단 첨자가 구해지면 이 값을 pr 변수에 대입하여 ArBtn[pr]의 정보를 참조하여 프로그램을 실행한다. 프로그램을 실행하기 전에 시작 디렉토리를 변경하되 시작 디렉토리 지정이 잘못된 경우를 위해 예외 처리를 해 주었다. 그렇지 않으면 귀찮게 에러 메시지가 출력되어 미관상 좋지 못하다. 프로그램 실행은 WinExec 함수로 간단하게 구현하며 옵션 설정 상태에 따라 몇 가지 처리가 추가되어 있다. launch 함수는 비교적 간단하다.

프로그램을 실행시키는 함수는 launch 함수이다. 이 외에 실행과 직접적인 연관은 없지만 미적 효과를 내는 두 함수가 있다. 버튼이 눌러질 때와 떨어질 때의 처리를 수행하는 btnmousedown, btnmouseup 함수이며 이 함수들은 makebutton 함수에서 버튼을 동적으로 생성할 때 버튼의 OnMouseDown, OnMouseUp 이벤트 핸들러로 지정되어 있다. 하는 일은 버튼이 눌러질 때 실제 눌러진 것처럼 쑥 들어간 효과를 내는 것과 버튼이 떨어질 때 원래대로 다시 복구하는 일이다.

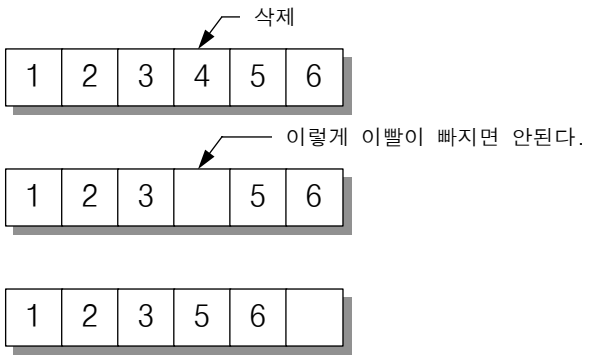
버튼은 당연히 마우스를 누르면 들어가고 놓으면 다시 튀어나오도록 되어 있지만 우리가 만든 버튼은 Button 컴포넌트가 아니라 이미지 컴포넌트이다. 그래서 이렇게 일부러 효과를 내지 않으면 그림만 그대로 있게 되므로 별로 사실감이 없다. Button 컴포넌트에 아이콘을 출력할 수 있다면 이런 귀찮은 처리는 하지 않아도 되겠지만 필자는 그런 방법을 찾지 못해 궁여지책으로 이런 방법을 사용하였다.

# 마. 삭제

프로그램을 등록하는 기능이 있으면 반대로 삭제하는 기능도 있어야 한다. 그런데 등록하는 과정에 비해 삭제 과정은 좀 더 어려운 면이 많다. 왜냐하면 ArBtn 배열은 순서대로 프로그램의 등록 정보를 보관하며 가운데 하나를 비워 둘 수 없기 때문이다. 가운데 하나가 없어지면 선형검색을 할 수도 없으며 버튼의 위치가 번호에 따라 배치되므로 실제 화면상에도 빈칸이 생겨 미관상 치명적인 문제를 발생시킨다.

그래서 삭제된 버튼 이후의 버튼을 모두 앞으로 이동시켜야 한다. 예를 들어 6개의 버튼이 있는데 4번 버튼이 삭제되었다고 한다면 다음과 같이 5번 이후의 버튼이 앞으로 이동해야 한다.

그림
버튼의 삭제




앞으로 한칸씩 이동시켜야 한다.

그래서 삭제할 버튼 이후의 등록 정보를 모두 한칸씩 앞으로 이동시킨다. AtBtn[i]:=ArBtn[i+1]을 총 등록 프로그램의 수만큼 반복한다. ArBtn의 정보는 이런 식으로 복사하되 동적으로 생성한 객체는 복사할 수 없으므로 동적으로 해제한 후 다시 만들어 주었다. 이때 동적으로 생성한 객체가 부자 관계를 가지므로 반드시 자식 컴포넌트를 먼저 해제한 후 부모를 해제해야 한다. 순서를 바꾸면 부모없는 고아가 발생하게 된다. 해제는 Free 메소드로 직접 하고 다시 만들 때는 makebutton 함수를 사용한다.

결국 삭제되는 버튼 이후의 버튼은 모두 없어졌다가 다시 만들어지는 셈이다. 이 방법이 웬지 지저분해 보이고 더 좋은 방법이 있을 것도 같지만 필자는 이 이상의 해결 방법을 찾지 못했다. 하지만 기능적으로 문제가 없고 속도도 생각했던 것보다 훨씬 더 빨라 그냥 두기로 했다.

# 13-3 옵션 설정

## 가. 레지스트리 관리

SDShell은 철저하게 정보를 보관한다. 일단 한 번 등록한 프로그램을 기억해 두어야 하는 것은 너무나 당연한 일이고 폼의 위치, 크기 및 옵션 설정 상태까지도 고스란히 보관하여 다음에 SDShell을 실행할 때 끝내기 전의 상태 그대로 시작할 수 있도록 한다. 정보 보관은 물론 레지스트리를 사용한다. 우선 FormCreate 이벤트 핸들러에서 폼의 위치와 크기, 중요 옵션 설정 상태를 읽어들이며 loadini 함수에서 등록된 프로그램에 관한 정보를 읽어 ArBtn 배열에 기입해 둔다. 프로그램을 끝낼 때 정보를 보관하는 일은 saveini 함수에서 수행한다. 각 함수의 구체적인 코드는 소스를 보면 알겠지만 읽기, 쓰기 함수의 단순한 나열이므로 특별히 분석할 만한 대상은 아니다. 하지만 몇 가지 섬세한 처리를 볼 수 있다. 우선 FormCreate 이벤트 핸들러를 보면 다음과 같은 코드가 있다.

```
if (shellForm.Left>scrmaxx) or (shellForm.Top>scrmaxy) then
  begin
  shellForm.Left:=0;
  shellForm.Top:=0;
  end;
```

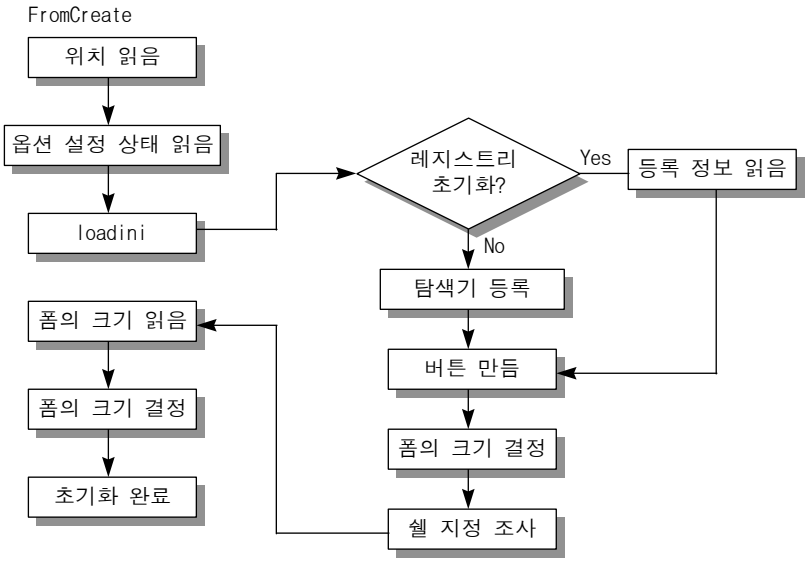
이 코드는 폼의 위치를 읽어들인 후 SDShell이 현재 화면 해상도 안쪽에 있는가를 점검해 보고 그렇지 않으면 화면의 좌상단으로 폼을 옮기는 역할을 한다. 만약 이 코드가 없다면 SDShell이 화면에 아예 나타나지 않는 불상사가 발생할 수도 있다. 예를 들어 1024*768 해상도에서 SDShell을 우측에 배치한 후 윈도우즈를 끝냈다가 해상도를 640*480으로 바꾸어 버린 후 다시 윈도우즈를 시작하면 SDShell은 화면 바깥에 존재하게 된다. 쉘이 화면에 없으면 윈도우즈를 어떻게 사용한다는 말인가. 이런 에러 처리는 일종의 극한 상황에 대한 배려라고 할 수 있다.

그리고 loadini의 처음 부분을 보면 레지스트리가 초기화되지 않은 경우의 처리를 해 주고 있다. 레지스트리에 정보가 전혀 없으면 옵션이나 위치는 둘째 치더라도 등록 프로그램이 하나도 없는 사태가 발생하며 이때 만약 SDShell이 윈도우즈 쉘로 사용되고 있다면 아무 프로그램도 실행할 수 없다. 그래서 INI 파일

이 없더라도 최소한 프로그램 등록부터 할 수 있도록 탐색기를 디폴트로 등록해 준다. 이런 처리가 꼭 필요한 이유는 SDShell이 설치 프로그램을 별도로 제공하지 않으며 실행 파일 하나만 가지고 다녀도 쓸 수 있도록 작성되었기 때문이다. 설치해야만 쓸 수 있는 프로그램이라면 설치할 때 레지스트리의 정보도 같이 초기화해 주겠지만 복사해서 사용할 수도 있기 때문에 레지스트리 초기화도 책임을 져야 한다.

레지스트리의 정보를 읽어들여 버튼을 만들고 폼을 출력하는 초기화 과정을 순서도로 그려보면 다음과 같다.

**그림**
초기화 과정




여기서 어떤 정보를 먼저 읽어들이는가의 순서가 중요한 의미를 가진다. 등록된 프로그램의 개수를 알아야 폼의 크기가 결정되기 때문에 폼의 위치는 등록 정보보다 먼저 읽혀도 상관이 없지만 폼의 크기는 등록 정보를 읽은 후에 읽어야 한다.

이 외에도 INI 파일에 정보를 읽고 쓰는 함수는 쪽지의 설정 상태와 쪽지 내용을 읽고 쓰는 loadnote 함수와 savenote 함수가 있는데 함수 내용은 단순한 입출력문뿐이다.

# 나. 폼의 크기 변경

쉘은 다른 프로그램을 실행시켜야 하는 대단히 중요한 임무를 가지면서 동시에 다른 프로그램을 위해 최대한 화면을 적게 차지하는 미덕을 지녀야 한다. 그래서 대부분의 쉘은 최대한 넓이를 줄이려고 하며 어떤 쉘은 아예 모습을 숨기기도 한다. SDShell도 마찬가지로 그리 넓지 않은 윈도우를 가지며 윈도우의 모양을 마음대로 바꿀 수 있도록 사용자에게 자유를 부여한다. 폼의 크기는 자유자재로 바꿀 수 있되 일단 사용자가 변경한 모양을 유지하면서 버튼을 모두 표시할 수 있는 최소한의 크기를 만들어낸다.

이런 동작은 FormResize 함수에서 폼의 크기가 변할 때마다 수행한다.

```
procedure Tshellform.FormResize(Sender: TObject);
var
  i:integer;
begin
IconPerRow:=ClientWidth div 36;
{총 프로그램 개수 이하이거나 3 이상이어야 한다.}
if IconPerRow>totalprg+IsResTimer Then
   IconPerRow:=totalprg+IsResTimer;
if IconperRow<3 Then IconperRow:=3;
{아이콘의 위치를 이동시킨다.}
for i:=1 to totalprg do
  with ArBtn[i] do begin
  Panel.Left:=((i-1+IsResTimer) mod IconPerRow)*36;
  Panel.Top:=((i-1+IsResTimer) div IconPerRow)*36;
  end;
{폼의 크기를 다시 조정한다.}
padresize;
end;
```

우선 폼의 크기가 변하면 수평 넓이에 따라 한 줄에 몇 개의 아이콘을 배치할 것인가를 결정한다. 이 값은 IconPerRow 변수에 저장되며 이 변수가 폼의 크기와 버튼의 배치에 중요한 기준이 된다. 버튼의 가로폭이 36픽셀이므로 윈도우의 폭을 36으로 나누면 한 줄에 몇 개의 아이콘을 배치할 것인가를 결정할 수 있다. IconPerRow가 일단 결정되면 무조건 이 값을 사용하는 것이 아니라 상한과 하한을 점검한다. 상한은 프로그램 전체의 개수이며 하한은 시간, 날짜 표시기의 폭인 3이다. 이 변수값이 결정되면 버튼의 위치를 다시 조정하여 배치하고

padresize 함수를 불러 폼의 크기도 다시 조정한다. padresize 함수는 다음 두 문장으로 이루어져 있다.

```
ClientHeight:=
  (((totalprg-1+IsResTimer) div IconperRow)+1)*36;
ClientWidth:=36*IconPerRow;
```

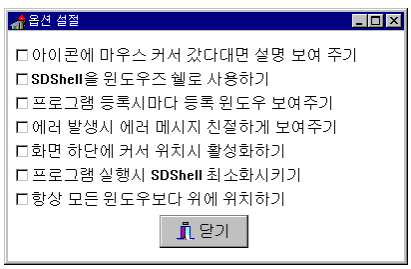
폼의 폭은 IconPerRow에 36을 곱하여 구하는데 어찌 생각하면 쓸데없는 식처럼 보인다. IconPerRow가 ClientWidth div 36이며 ClientWidth는 36*(ClientWidth div 36)=ClientWidth가 될 것 같지만 그렇지는 않다. 왜냐하면 div가 실수 나눗셈이 아닌 정수 나눗셈이기 때문에 ClientWidth는 무조건 36의 배수로 맞추어지게 된다. 폼의 높이를 계산하는 식은 조금 복잡하다. 이런 식은 머리를 굴려 만들어내기 보다는 IconPerRow가 4인 특별한 경우를 가정하여 식을 만든 후 5인 경우와 3인 경우의 검정과 수정을 해 보면 쉽게 일반화하여 얻을 수 있다.

FormResize 이벤트 핸들러와 padresize 함수에 의해 사용자가 폼의 폭을 변경할 때마다 버튼이 폼의 폭에 맞게 다시 배치되고 폼의 크기는 버튼 배치 상태에 따라 다시 조정된다.

# 다. 쉘 기능

SDShell은 윈도우즈 쉘로 사용될 수 있다. 윈도우즈 3.1에서 프로그램 관리자를 대신할 수도 있고 윈 98에서 탐색기를 대신할 수도 있다. SDShell을 쉘로 지정하는 일은 옵션 대화상자에서 두 번째 옵션을 사용하여 이루어진다. 이 옵션을 변경함에 따라 코드에서 어떤 처리를 하는가를 살펴보자. 이 코드는 Sdop_f.pas에 있다.

**그림**
옵션 대화상자




```
if (CheckBox2.State=cbchecked) and (Mastershell=False) then
```

```
if MessageDlg('SDShell 을 윈도우즈용 쉘로 사용하려면 윈도우즈를 다시 시작해야 합니다.'
  ,mtConfirmation,mbOkCancel,0)=mrOk then
Begin
MasterShell:=True;
getprivateprofilestring('boot','shell','null',pst,128,'system.ini');
WritePrivateProfileString('boot','oldshell',pst,'system.ini');
StrPCopy(Pst,curdir+'SDShell.exe');
WritePrivateProfileString('boot','shell',pst,'system.ini');
shellform.saveini;
ExitWindows(EW_RESTARTWINDOWS,0);
end;
```

쉘 지정을 하려면 두 번째 체크 박스가 선택되어 있어야 하며 현재 쉘 지정이 되어 있지 않아야 한다. 이 두 조건을 만족하면 윈도우즈를 다시 시작하겠느냐는 질문을 하며 system.ini에 SDShell을 쉘로 지정한 후 윈도우즈를 다시 기동시킨다. 이때 system.ini의 shell문에 보관되어 있던 기존의 shell을 oldshell로 보관하여 쉘 지정을 취소할 경우를 대비해 두어야 한다. 조건문에 사용된 현재 쉘 지정 여부인 MasterShell 변수는 loadini에서 SDShell이 처음 시작할 때 미리 조사해 둔다.

```
getprivateprofilestring('boot','shell','null',pst,128,'system.ini');
str:=ExtractFileName(StrPas(pst));
str:=Copy(Str,1,11);
i:=CompareStr('SDShell.exe',str);
if i=0 Then Mastershell:=True
 else Mastershell:=False;
```

쉘 지정을 취소하는 코드도 이와 비슷하다.

```
if (CheckBox2.State=cbunchecked) and (Mastershell=True) then
 if MessageDlg('SDShell 을 윈도우즈용 쉘로 사용하지 않으려면 윈도우즈를 다시 시작해야 합니다.'
  ,mtConfirmation,mbOkCancel,0)=mrOk then
 Begin
 MasterShell:=False;
 getprivateprofilestring('boot','oldshell','null',pst,128,'system.ini');
 WritePrivateProfileString('boot','shell',pst,'system.ini');
 shellform.saveini;
 ExitWindows(EW_RESTARTWINDOWS,0);
 end;
```

현재 쉘로 사용하고 있는 상태에서 두 번째 체크 박스가 선택되지 않았을 때 쉘 지정을 취소하며 oldshell에 보관된 원래의 쉘을 원위치시킨 후 윈도우즈를

다시 기동시킨다.

SDShell이 쉘로 사용되고 있는 상황에서 SDShell을 종료하면 윈도우즈도 같이 종료해 주어야 한다. 만약 이 처리를 생략해 버리면 쉘만 끝나고 윈도우즈는 계속 실행중이므로 화면에 마우스 커서만 있는 별 희한한 사태가 발생하게 된다. 그렇다면 어느 시점에서 윈도우즈를 끝내야 할까?

쉘을 종료할 때 윈도우즈도 같이 종료되어야 하므로 FormClose 이벤트 핸들러에서 윈도우즈를 종료해 주는 것이 적당할 것 같다. 그러나 조금만 더 생각해 보면 OnClose 이벤트에서는 이런 처리를 할 수 없다. 윈도우즈를 종료하는 일은 상당히 중대한 일이므로 사용자에게 정말 윈도우즈를 끝낼 것인가에 대한 질문을 한번쯤 하는 것이 상식적으로 자연스럽다. 그래야만 사용자가 실수로 윈도우즈를 종료하려고 했을 때 취소를 할 수 있다. 그러나 OnClose 이벤트는 이미 쉘 종료 결정이 난 후에 발생하는 이벤트이므로 취소를 할 수 없다. FormClose 이벤트 핸들러에 어떤 코드를 작성하더라도 쉘이 끝나는 것은 이미 정해진 일이기 때문이다.

윈도우즈를 종료할 것인가에 관한 질문은 폼이 닫혀지기 전에 결정을 내려야 한다. 폼이 닫혀지기 전에 일어나는 일은 사용자가 조절 메뉴를 더블클릭할 때이며 이때 발생하는 메시지는 WM_SYSCOMMAND이다. 이 메시지를 처리하는 메시지 핸들러를 정의해 두고 이 핸들러에서 질문을 하고 윈도우즈 종료 처리를 해 주면 된다. 폼의 타입 선언을 보면 다음과 같은 메시지 핸들러 지정문이 있다.

```
procedure Syscommand(var M:TWMSYSCommand);
  message WM_SYSCOMMAND;
```

그리고 implementation부에는 이 메소드에 대한 실제의 코드가 작성되어 있다.

```
procedure Tshellform.Syscommand(var M:TWMSYSCommand);
begin
if (M.CmdType=SC_CLOSE) and (Mastershell) then
 begin
 if MessageDlg('윈도우즈를 끝낼까요?'
  ,mtConfirmation,mbOkCancel,0)=mrOk
  Then  begin
  saveini;          {끝내기 전에 정보를 저장한다.}
  ExitWindows(0,0);   {윈도우를 종료한다.}
  end;
```

```
 end
else Inherited;
end;
```

폼을 닫으라는 명령(시스템 메뉴 더블클릭)이 입력되었고 쉘 지정이 되어 있으면 윈도우즈를 끝낼 것인가를 물어보고 OK 버튼을 누르면 모든 정보를 저장하고 윈도우즈를 종료한다. 그렇지 않을 경우는 Inherited를 호출하여 고유의 동작을 할 수 있도록 해 주어야 한다. 이 메소드 외에도 팝업 메뉴의 끝내기 항목에도 비슷한 코드가 있다.

# 라. 프로그램 활성화

쉘은 빈번히 사용되므로 항상 사용할 수 있는 태세를 갖추고 있어야 한다. 그래서 대부분의 쉘 프로그램은 단축키를 가지며 언제든지 단축키를 통해 쉘을 활성화시킬 수 있으며 필요할 때 곧바로 사용할 수 있다. SDShell은 이런 활성화를 위해 단축키는 사용하지 않으며 대신 마우스 커서의 위치를 사용한다. 확률적으로 마우스 커서가 화면의 우하단에 위치하는 일은 거의 없으므로 만약 마우스 커서가 이 위치에 있다면 SDShell을 활성화시키라는 뜻으로 받아들여 즉시 활성화한다. 코드는 다음과 같다.

```
procedure Tshellform.Timer2Timer(Sender: TObject);
var
 CPOS:TPOINT;
begin
GetCursorPos(CPOS);
if (CPOS.x>scrmaxx-4) and (CPOS.y>scrmaxy-4) and IsRBActivate then
 begin
 BringWindowToTop(handle);
 SetActiveWindow(handle);
 SetForegroundWindow(handle);
 end;
end;
```

마우스 커서의 현재 위치를 구한 후 화면 우하단에서 4픽셀 범위 내에 있으면 SDShell을 활성화시킨다. 단 이 기능은 경우에 따라서는 무척 귀찮아질 수도 있으므로 옵션(IsRBActivate)에 의해 사용 여부를 결정할 수 있도록 되어 있다. 이 외에도 다양한 활성화 방법을 생각해 볼 수 있지만 아무래도 단축키보다는 마우스로 활성화를 시키는 방법이 더 우수하다. 실제로 사용해본 결과 이 기능은 꽤

쓸만한 것 같다.

# 13-4 쪽지 기능

## 가. Note 레코드

등록된 프로그램이 ArBtn 배열에 저장되듯이 쪽지는 Note 배열에 저장된다. TNote 레코드는 다음과 같이 정의되어 있다.

```
TNote=record
  X,Y,W,H:Integer;  {노트의 위치와 크기}
  exist:boolean;    {노트의 존재 여부. 1 이면 있음}
  notefrm:Tnotepad; {노트 윈도우. 동적으로 생성됨}
  fontname:string;  {폰트의 이름}
  fontsize:integer; {폰트의 크기}
  color:TColor      {노트의 색상}
end;
```

각 필드의 의미는 다음과 같다.

### ☐ X,Y,W,H

쪽지의 위치와 크기를 저장하는 필드이다.

### ☐ exist

쪽지가 존재하는가 그렇지 않은가를 기억하며 이 값이 1이면 해당 쪽지가 있는 것이다. 쪽지는 별도의 텍스트 파일에 저장되지만 쪽지를 지우더라도 텍스트 파일을 삭제하지는 않으며 이 필드만 0으로 바꾼다. 이 필드는 또한 쪽지를 화면에 나타내거나 숨길 때 조건 점검의 기준이 된다.

### ☐ notefrm

쪽지를 출력할 폼이다. 쪽지 폼은 동적으로 생성되며 생성한 폼이 이 필드에 저장된다.

### ☐ 그외 나머지

쪽지의 폰트와 색상에 대한 옵션을 기억한다.

쪽지를 저장하는 Note 배열의 내용은 프로그램 등록 정보와 마찬가지로

SDSHELL.INI 파일에 같이 저장된다. 그러나 쪽지의 내용 자체는 길이가 너무 길기 때문에 INI 파일에 저장할 수 없으며 system 디렉토리에 별도의 텍스트 파일로 저장된다. 쪽지 파일은 sdnote라는 파일 이름을 가지며 확장자는 쪽지 번호에 따라 1~20까지이다. Note 레코드 배열은 다음과 같이 전역으로 선언되어 있다.

```
var
  Note:array [1..maxnote] of TNote;   {노트}
```

Note 배열의 크기는 maxnote이며 이 상수는 20으로 정의되어 있다. 따라서 동시에 만들 수 있는 쪽지의 개수는 20개까지이다. 만약 쪽지를 더 많이 만들고 싶다면 maxnote를 늘려주면 된다.

# 나. 쪽지 만들기

쪽지 폼은 다음과 같이 디자인되어 있다.

그림
쪽지 폼

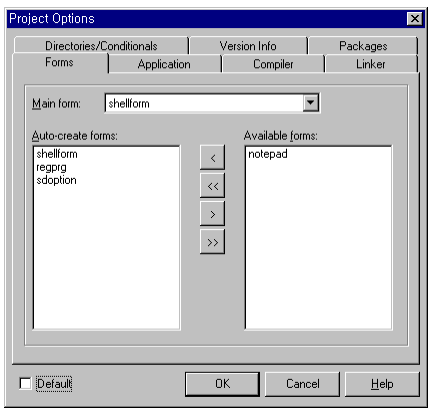

배경색은 노란색을 사용하며 폼 전체가 메모 컴포넌트로 덮여 있으며 옵션 조정을 위해 색상, 폰트 대화상자와 팝업 메뉴 컴포넌트가 있다. 폼을 생성하는 코드는 다음과 같다.

```
procedure Tshellform.notenewClick(Sender: TObject);
var i:integer;
begin
{최초의 존재하지 않는 노트 번호를 조사한다.}
for i:=1 to maxnote do
  if Note[i].exist=False then break;
if i=maxnote then begin
  MessageDlg('노트는 최대 20 개까지 만들 수 있습니다.'
  ,mtError,[mbOk],0);
  exit;
```

```
 end;
note[i].exist:=True;
{실행중에 동적으로 폼을 만든다.}
Application.CreateForm(Tnotepad,note[i].notefrm);
note[i].notefrm.caption:='쪽지 '+IntToStr(i);
note[i].notefrm.Left:=i*30;
note[i].notefrm.Top:=i*20;
note[i].notefrm.Show;
end;
```

20개의 배열을 모두 뒤져보아 exist가 0인, 즉 아직 만들어지지 않은 빈 배열 요소가 있으면 그 첨자를 새로 만들어지는 쪽지로 사용한다. 일단 첨자가 정해지면 동적으로 폼을 생성하고 쪽지 폼의 디폴트 위치를 정해준다. 한번 쪽지를 만든 첨자는 exist가 True이므로 다른 쪽지를 만들 때는 사용되지 않으며 지워진 쪽지는 exist가 False로 되므로 그 자리에 새로운 쪽지가 만들어질 것이다. ArBtn은 비워진 칸을 만들지 않지만 Note는 비워진 칸을 만들 수도 있다.

쪽지 숨기기와 쪽지 보이기 기능은 Note 배열을 모두 뒤져서 exist가 True인 쪽지를 Hide, 또는 Show 메소드로 숨기거나 보이도록 하는 아주 간단한 코드로 되어 있다. 쪽지에 입력된 문장은 메모 컴포넌트가 저장하며 SDShell을 끝낼 때 savenote 함수에 의해 텍스트 파일로 저장되었다가 다음에 SDShell을 실행할 때 loadnote 함수에 의해 다시 복구된다. 쪽지 폼은 동적으로 생성되므로 Options/Project 메뉴의 자동 생성 리스트에서는 삭제해 두어야 한다.




# 다. 쪽지 관리

쪽지 관리는 쪽지 폼의 팝업 메뉴에 다음과 같이 정의되어 있는 메뉴 항목으

로 수행한다.

그림
쪽지 폼의 팝업 메뉴


이 메뉴에 대한 코드는 모두 Note_f.pas에 정의되어 있다. Note_f.pas에 정의된 코드 중 우선 팝업 메뉴가 열릴 때 실행되는 이벤트 핸들러를 살펴보자.

```
procedure Tnotepad.PopupMenu1Popup(Sender: TObject);
var
 i:integer;
begin
 for i:=1 to maxnote do
  {윈도우 핸들과 note 배열상에 보관된 핸들을 비교한다.}
  if note[i].exist then if note[i].noteFrm.handle=handle then break;
 noteindex:=i;
end;
```

쪽지는 하나의 폼을 만들어 두고 같은 폼을 여러 개 만들어 배열에 저장하여 사용하므로 쪽지를 관리하는 함수는 우선 어떤 폼이 작업의 대상이 되는가를 선택해야 한다. 개별 함수에서 이런 선택을 일일이 하는 것 보다는 OnPopup 이벤트에서 팝업 메뉴가 열릴 때 한 번 조사를 해 두고 그 조사 결과를 함수에서 사용하는 것이 더 효율적이다. 이 함수에서는 Note 배열에 저장된 윈도우 핸들과 팝업 메뉴를 부른 핸들을 비교하여 Note 배열의 첨자를 조사해 noteindex라는 변수에 저장한다.

일단 첨자가 구해지면 쪽지를 관리하는 일은 쉽다. 쪽지 숨기기는 Hide 메소드를 한번만 호출해 주면 되며 지우기는 exist 필드를 False로 바꾸어 주면 된다. 쪽지 파일 자체를 지우지 않더라도 exist 필드가 False로 되어 있으면 쪽지가 보이지도 않을 뿐만 아니라 다음에 다른 쪽지를 만들면 덮여서 없어지게 된다. 단 쪽지를 지울 경우는 레지스트리를 직접 수정해서 쪽지가 지워졌음을 기록해야 한다. 왜냐하면 savenote 함수에서 쪽지 정보를 저장할 때 exist 필드가 False인 경우는 아예 저장을 하지 않기 때문이다.

## 라. 쪽지 옵션

쪽지의 색상과 폰트를 마음대로 바꿀 수 있다. 바꾸는 방법은 팝업 메뉴와 대화상자를 사용하는 방법 이상은 아니므로 별도의 설명을 할 필요는 없을 것 같다. 폰트의 경우는 이탤릭체나 굵은체 등의 속성은 지정할 수 없도록 하였다. 물론 할 수도 있지만 선택할 수 있는 옵션이 너무 많으면 레지스트리에 저장해야 할 정보가 너무 커지기 때문에 생략하였다.

쪽지 폼의 캡션 바에 쪽지의 제목을 달 수도 있고 쪽지의 내용을 별도의 텍스트 파일로 저장하거나 반대로 텍스트 파일에서 읽어올 수도 있다. Note 배열의 첨자가 구해지면 이런 일은 무척이나 쉽게 구현할 수 있다.

# 13-5 부가 기능

쉘이라고 해서 꼭 프로그램을 실행시키는 기능만을 가지는 것은 아니다. 일단 쉘은 항상 실행되고 있는 프로그램이므로 쉘에 부가 기능을 넣을 경우 언제든지 사용할 수 있는 편리한 명령이 된다. SDShell도 물론 몇 가지 부가 기능을 제공하고 있다.

## 가. 날짜 및 시계 표시기

윈도우의 좌상단에 연두색의 패널이 있으며 이 패널에는 오늘 날짜와 시간이 항상 출력된다. 둘 다 누구나 관심을 가지는 정보이므로 쉘에 있을 법도 한 기능이다. 날짜와 시간은 항상 변하므로 제대로 된 정보를 보여 주려면 매 시간마다 정보를 갱신해야 하며 그래서 타이머 메시지를 사용한다. 시간은 어차피 분단위로 출력되며 날짜는 거의 고정되어 있는거나 다름없으므로 너무 자주 조사할 필요가 없다. 그래서 이 타이머의 호출 주기는 5초로 설정되어 있다.

```
procedure Tshellform.Timer1Timer(Sender: TObject);
 var
  H,M,S,MS:Word;
  Mo,D,Y:Word;
begin
DecodeTime(Time,H,M,S,MS);
if H>12 Then H:=H-12;
timergauge.caption:=Format('%d 시%d 분',[H,M]);
DecodeDate(Date,Y,Mo,D);
DateGauge.caption:=Format('%d 월 %d 일',[Mo,D]);
end;
```

시간과 날짜를 조사하여 출력하는 방법은 이미 9장에서 많이 다루었던 내용이다.

## 나. 윈도우즈 다시 시작하기

윈도우즈가 불안하다거나 화면이 지저분해진 경우는 윈도우즈를 다시 시작

하는 것이 좋다. 이때는 윈도우즈를 셧다운 시킨 후 다시 시작해야 한다. SDShell은 메뉴 명령 하나만으로 윈도우즈를 다시 시작할 수 있도록 하고 있다. 물론 윈도우즈를 다시 시작한다는 것은 무척 위험한 동작이므로 반드시 한번의 확인을 거친다.

```
procedure Tshellform.N6Click(Sender: TObject);
begin
 if MessageDlg('윈도우즈를 다시 시작할까요?',
  mtConfirmation,mbOkCancel,0)=mrOk
  Then  begin
  saveini;
  ExitWindows(EW_RESTARTWINDOWS,0);
  end;
end;
```

윈도우즈를 끝내거나 다시 시작하는 함수는 ExitWindows 함수이다.

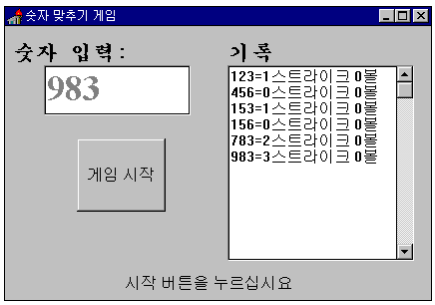
이상으로 SDShell 분석을 모두 마친다. 이 프로그램은 어디까지나 아마추어의 실력으로 아마추어가 아마추어를 위해 만든 것이다. 더구나 델파이를 처음 배울 때 습작으로 만든 것이라 군데군데 미숙한 부분이 보이기도 하며 버그의 가능성도 완전히 배제할 수는 없다. 이미 일정 수준에 도달한 사람에게는 우습게 보일지도 모르겠지만 이제 막 델파이와 친해지기 시작한 사람에게는 배울 것이 많은 예제가 될 것이라고 생각한다.

# 13-6 숫자 맞추기 게임

## 가. 게임 소개

13jang
base

이 게임은 둘이서 세 자리 숫자를 생각한 후 상대방의 숫자를 물어보고 힌트를 통해 숫자를 먼저 맞추는 쪽이 이기는 게임이다. 힌트를 스트라이크, 볼 등의 말로 표현하므로 흔히 야구 게임으로 알려져 있으며 약간의 사고력을 요한다. 두 사람이 하는 게임이지만 컴퓨터가 만든 숫자를 사용자가 맞추는 형식으로 변형시켜 보았다. 실행중의 화면은 다음과 같다.




숫자를 입력하는 에디트 하나와 먼저 입력한 숫자들의 힌트를 기록해 놓는 메모, 그리고 게임을 시작하는 버튼으로 구성되어 있다. 게임이 단순하기는 하지만 더 세련되게 디자인하면 제법 할 만한 게임이다. 전체 소스는 다음과 같으며 문자열 비교 이외에는 특별한 기법이 없는 아주 단순한 게임이다.

```
{컴퓨터가 난수로 만든 세자리 숫자를 맞추는 게임}
unit Base_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
```

```
    Memo1: TMemo;
    BtnStart: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure BtnStartClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  comnum:string;
  mynum:string;
  times:integer;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
randomize;  {난수 루틴을 초기화한다.}
Edit1.Enabled:=False;
label1.caption:='시작 버튼을 누르십시요';
end;

{시작 버튼 누름}
procedure TForm1.BtnStartClick(Sender: TObject);
begin
Memo1.Lines.Clear;  {기록 내용을 지움}
{난수를 만들되 각 자리값이 중복되어서는 안되며 중간
에 0 이 있어도 안된다.}
repeat
comnum:=IntToStr(random(900)+100);
until (comnum[1]<>comnum[2]) and
      (comnum[1]<>comnum[3]) and
      (comnum[2]<>comnum[3]) and
      (comnum[1]<>'0') and
      (comnum[2]<>'0') and
      (comnum[3]<>'0');
times:=0;
```

```
label1.caption:='숫자를 입력하십시요';
Edit1.Enabled:=True;
Edit1.SetFocus;
Edit1.SelectAll;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  S,B,i:integer;
begin
if Key<>#13 then Exit;   {엔터키를 눌러야 함}
{숫자 이외의 문자가 있거나 세자리 숫자가 아닌 경우}
if (StrToIntDef(Edit1.Text,0)=0)
  or (Length(Edit1.Text)<>3) then
  begin
  label1.caption:='세자리 숫자를 입력하십시요';
  Edit1.SetFocus;
  Edit1.SelectAll;
  Exit;
  end;
times:=times+1;
S:=0;   {스트라이크}
B:=0;   {볼}
mynum:=Edit1.Text;
{스트라이크 개수를 센다}
for i:=1 to 3 do
  if comnum[i]=mynum[i] then S:=S+1;
{볼 개수를 센다}
if comnum[1]=mynum[2] then B:=B+1;
if comnum[1]=mynum[3] then B:=B+1;
if comnum[2]=mynum[1] then B:=B+1;
if comnum[2]=mynum[3] then B:=B+1;
if comnum[3]=mynum[1] then B:=B+1;
if comnum[3]=mynum[2] then B:=B+1;
{기록 내용을 메모로 출력한다.}
Memo1.Lines.Add(Format('%s=%d 스트라이크 %d 볼',
        [mynum,S,B]));
{스트라이크가 셋이면 정답을 맞춘 것임}
if S=3 then
  begin
  ShowMessage('정답을 맞추었습니다. 축하합니다.');
  Edit1.Enabled:=False;
  label1.caption:='시작 버튼을 누르십시요';
  end
else
  begin
```

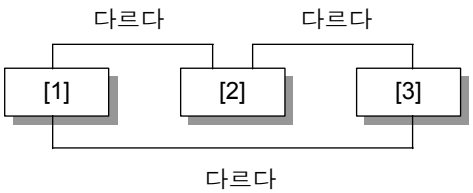
```
  label1.caption:='숫자를 입력하십시요';
  Edit1.SetFocus;
  Edit1.SelectAll;
  end;
end;

end.
```

# 나. 숫자 만들기

사용자가 맞추어야 할 숫자는 난수로 만들어진다. 만들어지는 시점은 게임을 시작할 때이므로 시작 버튼의 OnClick 이벤트를 사용한다. 만들어지는 숫자의 특징은 우선 세 자리가 모두 다른 숫자여야 하며 중간에 0이 있어서도 안된다. random(900)으로 0~899까지의 난수를 만든 후 여기에 100을 더해 100~999까지의 정수 중 하나의 수를 만든다. 만들어진 숫자는 각 자리수를 비교해야 하기 때문에 정수 변수에 저장하지 않고 문자열로 변환한 후 comnum이라는 문자열 변수에 저장된다. 100~999까지의 난수 중 중복을 방지하기 위해 각 자리수를 비교해 보고 0이 중간에 있는지도 검사해 본다.

**그림**

세 가지 조건이 만족하는 숫자를 만들어야 한다.




먼저 숫자를 만든 후 조건에 맞는 숫자인지 점검해야 하므로 선실행 후평가문인 repeat until문이 사용되었다. 즉 조건에 맞는 숫자가 만들어질 때까지 계속 난수를 발생시킨다. 숫자를 만든 후 레이블에 “숫자를 입력하십시요”메시지를 출력하고 에디트가 포커스를 가지도록 하여 사용자가 바로 숫자를 입력할 수 있도록 해 준다.

# 다. 숫자 비교

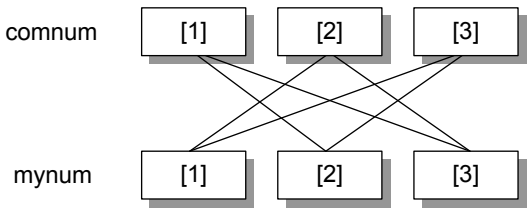
게임이 시작되면 사용자는 컴퓨터가 생각한 숫자인 comnum이 무엇인가 맞추어야 하며 컴퓨터는 사용자가 입력한 숫자와 자신이 생각한 숫자를 비교하여

힌트를 제공해 주어야 한다. 숫자를 비교하는 시점은 에디트 컴포넌트에 숫자를 입력한 후 Enter 키를 누를 때이다. 그래서 KeyPress 이벤트에서 제일 먼저 입력된 키가 Enter 키인 #13인지를 점검해 보고 그렇지 않으면 계속 숫자를 입력받도록 한다.

Enter가 입력되면 우선 사용자가 입력한 숫자가 제대로 비교 가능한 숫자인지부터 점검해 보아야 한다. 23, 8127 등과 같이 자리수가 틀리거나 2nd, 12$ 등과 같이 숫자가 아닌 다른 문자가 있다면 컴퓨터가 만든 난수와 비교를 할 수 없다. 입력한 문자 중에 숫자가 아닌 문자가 있는지를 알고자 할 때는 StrToIntDef 함수를 사용하여 문자열을 숫자로 변경해 보면 된다. 이 함수는 문자열을 숫자로 바꿀 수 없을 경우 두 번째 인수로 전달된 디폴트값을 그대로 리턴해 주는 특성이 있으므로 디폴트를 0으로 넘겨준 후 리턴값이 여전이 0이라면 사용자가 엉뚱한 문자를 입력한 것으로 판단하게 된다.

판단한 결과 입력된 숫자가 세 자리가 아니면 "세 자리 숫자를 입력하십시요"라는 메시지를 보여준다. 스트라이크는 입력받는 mynum과 컴퓨터가 만든 난수 comnum의 대응되는 자리수를 비교하여 일치하는 개수를 구하면 되고 볼은 다음과 같이 6번 비교를 수행하여 일치하는 개수를 구한다.

그림
볼 카운트 계산




계산한 결과는 각각 S, B 변수에 저장되며 이 값은 메모 컴포넌트로 출력되어 사용자가 다음 숫자 입력에 참고할 수 있도록 해 준다. 마지막으로 사용자가 입력한 숫자가 정답인지 즉, S가 3인지 점검해 보기만 하면 된다. 정답일 경우는 대화상자를 열어 축하 메시지를 출력한 후 게임을 중단시키고 그렇지 않으면 계속 다른 숫자를 입력할 수 있도록 에디트 컴포넌트로 포커스를 이동시킨다.

# 라. SDMemo

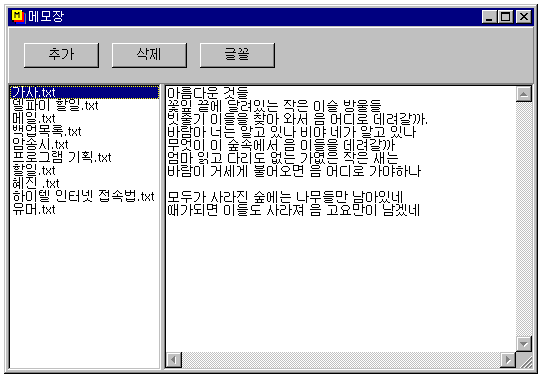
13jang
SDMemo

다음 예제는 간단한 메모를 해 둘 수 있는 메모장이다. 순간적으로 생각나는 정보들을 기록하기 위한 간단한 소프트웨어는 종류를 헤아릴 수 없이 많지만 필자의 경우는 텍스트 파일이 가장 무난하고 편리하다고 생각한다. 마음대로 복사

도 가능하고 편집할 수 있는 프로그램도 많이 있으니까 말이다. 그런데 메모 사항이 많을 경우는 이 파일 저 파일 열었다 닫았다 하는 것이 불편해서 메모의 묶음을 관리하는 프로그램을 만들어 보았다. 사용된 컴포넌트는 메모, 리스트 박스, 스플리터 정도에 불과하다. 실행중의 모습을 보이면 다음과 같다.

**그림**

간단한 메모장




추가 버튼으로 텍스트 파일을 목록에 넣을 수 있으며 목록에서 선택한 파일은 왼쪽의 메모에 나타난다. 텍스트 파일의 목록은 레지스트리에 저장되도록 하였으며 메모에서 텍스트 파일을 직접 편집할 수 있고 별도의 저장 명령을 내리지 않아도 메모가 변경되거나 프로그램이 종료될 때 자동으로 저장하도록 하였다. 전체 소스를 보인다.

```
unit SDMemo_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls,Registry;

type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Panel1: TPanel;
    Memo1: TMemo;
    Splitter1: TSplitter;
    BtnAdd: TButton;
    BtnDelete: TButton;
    OpenDialog1: TOpenDialog;
```

```
    BtnFont: TButton;
    FontDialog1: TFontDialog;
    procedure BtnAddClick(Sender: TObject);
    procedure ListBox1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure BtnDeleteClick(Sender: TObject);
    procedure BtnFontClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  OldItemIndex:Integer;
  FullPath:TStringList;

implementation

{$R *.DFM}

{텍스트 파일을 읽어와 리스트 박스에 추가한다}
procedure TForm1.BtnAddClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    begin
    ListBox1.Items.Add(ExtractFileName(OpenDialog1.Filename));
    FullPath.Add(OpenDialog1.Filename);
    ListBox1.ItemIndex:=ListBox1.Items.Count-1;
    OldItemIndex:=ListBox1.Items.Count-1;
    Memo1.Lines.LoadFromFile(ListBox1.Items[ListBox1.Items.Count-1])
    end;
end;

{삭제 버튼}
procedure TForm1.BtnDeleteClick(Sender: TObject);
var
  NowIndex:Integer;
begin
  if ListBox1.Items.Count = 0 then
    begin
    ShowMessage('지울 메모가 없습니다');
    Exit;
    end;
```

```
  NowIndex:=ListBox1.ItemIndex;
  ListBox1.Items.Delete(NowIndex);
  FullPath.Delete(NowIndex);

  {제일 끝 부분을 지웠을 때의 처리}
  if NowIndex > ListBox1.Items.Count-1 then
    NowIndex := ListBox1.Items.Count-1;
  ListBox1.ItemIndex:=NowIndex;
  OldItemIndex:=NowIndex;

  {최소한 하나 이상 남아 있으면 읽어온다}
  if NowIndex >= 0 then
    Memo1.Lines.LoadFromFile(FullPath[NowIndex])
  else
    Memo1.Text:='추가 버튼을 눌러 텍스트 파일을 추가하십시요';
end;

{리스트 박스를 선택하면 텍스트를 읽어온다}
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  if Memo1.Modified then
    Memo1.Lines.SaveToFile(FullPath[OldItemIndex]);
  Memo1.Lines.LoadFromFile(FullPath[ListBox1.ItemIndex]);
  OldItemIndex:=ListBox1.ItemIndex;
end;

{프로그램이 시작될 때 텍스트 파일 리스트를 읽어온다}
procedure TForm1.FormCreate(Sender: TObject);
var
  i:Integer;
  Count:Integer;
  SDReg:TRegistryIniFile;
  Path:String;
  LastIndex:Integer;
begin
  FullPath:=TStringList.Create;
  SDReg:=TRegistryIniFile.Create('Software₩SangHyungSoft₩SDMemo');

  {저장되어 있는 윈도우 위치를 읽어온다.}
  Left:=SDReg.ReadInteger('Pos','Left',100);
  Top:=SDReg.ReadInteger('Pos','Top',100);
  Width:=SDReg.ReadInteger('Pos','Width',640);
  Height:=SDReg.ReadInteger('Pos','Height',480);

  {저장되어 있는 글꼴 설정을 읽어온다.}
  Memo1.Font.Size:=SDReg.ReadInteger('Pos','fontsize',10);
```

```
  Memo1.Font.Name:=SDReg.ReadString('Pos','fontname','굴림');
  ListBox1.Font:=Memo1.Font;

  {텍스트의 개수와 텍스트 목록을 읽어온다.}
  Count:=SDReg.ReadInteger('Text','Count',0);
  for i:=1 to Count do
    begin
    Path:=SDReg.ReadString('Text',IntToStr(i), '');
    FullPath.Add(Path);
    ListBox1.Items.Add(ExtractFileName(Path));
    end;

  // 최후로 편집하던 텍스트 파일을 미리 읽어 놓는다.
  if Count ◇ 0 then
    begin
    LastIndex:=SDReg.ReadInteger('Text','LastIndex',0);
    Memo1.Lines.LoadFromFile(FullPath[LastIndex]);
    ListBox1.ItemIndex:=LastIndex;
    OldItemIndex:=LastIndex;
    end
  else
    Memo1.Text:='추가 버튼을 눌러 텍스트 파일을 추가하십시요';
  SDReg.Free;
end;

{프로그램이 종료될 때 리스트 박스의 텍스트 리스트를
레지스트리에 저장한다}
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  i:Integer;
  SDReg:TRegistryIniFile;
begin
  SDReg:=TRegistryIniFile.Create('Software₩SangHyungSoft₩SDMemo');
  {마지막 편집하던 텍스트를 저장한다}
  if (Memo1.Modified) and (OldItemIndex ◇ -1) then
    Memo1.Lines.SaveToFile(FullPath[OldItemIndex]);
  {텍스트의 개수부터 저장한다}
  SDReg.WriteInteger('Text','Count',ListBox1.Items.Count);
  for i:=0 to ListBox1.Items.Count-1 do
    begin
    SDReg.WriteString('Text',IntToStr(i+1),FullPath[i]);
    end;

  {마지막 편집하던 파일의 인덱스를 저장한다}
  SDReg.WriteInteger('Text','LastIndex',ListBox1.ItemIndex);
```

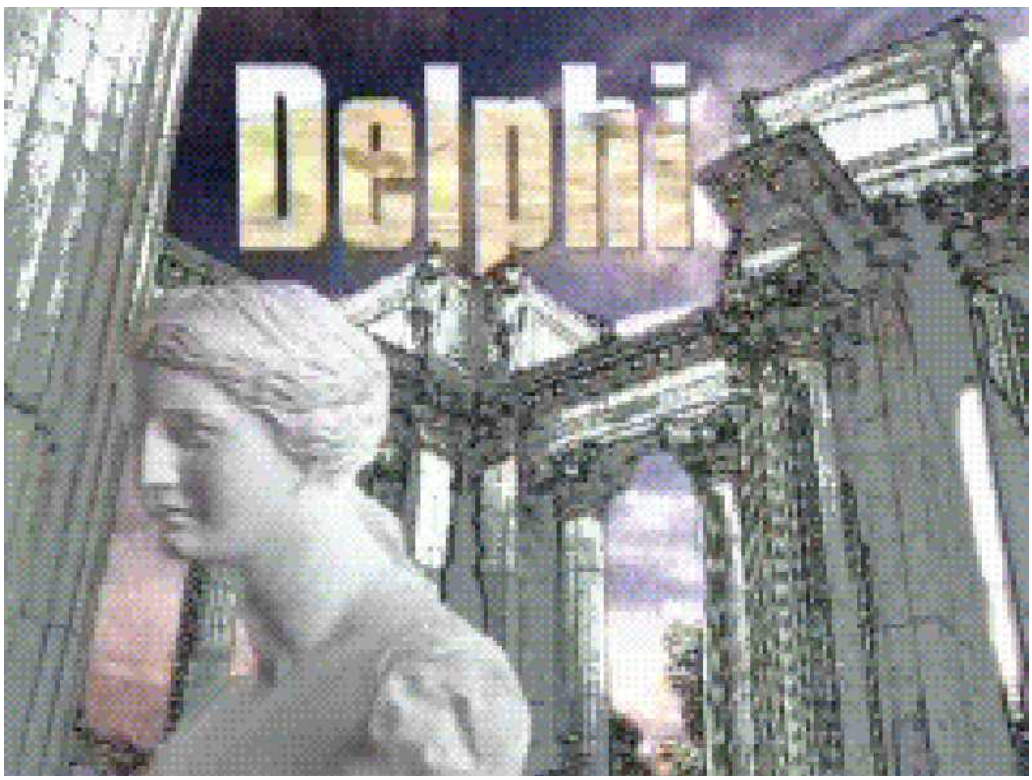
```
  {윈도우 위치를 저장한다}
  SDReg.WriteInteger('Pos','Left',Left);
  SDReg.WriteInteger('Pos','Top',Top);
  SDReg.WriteInteger('Pos','Width',Width);
  SDReg.WriteInteger('Pos','Height',Height);

  {글꼴 설정을 저장한다. 단 글꼴 이름과 크기만 저장된다.}
  SDReg.WriteInteger('Pos','fontsize',Memo1.Font.Size);
  SDReg.WriteString('Pos','fontname',Memo1.Font.Name);
  SDReg.Free;
  FullPath.Free;
end;

procedure TForm1.BtnFontClick(Sender: TObject);
begin
  FontDialog1.Font:=Memo1.Font;
  if FontDialog1.Execute then
    begin
    Memo1.Font:=FontDialog1.Font;
    ListBox1.Font:=FontDialog1.Font;
    end;
end;

end.
```

추가 버튼과 삭제 버튼을 사용하여 텍스트 파일 목록을 관리하며 리스트 박스에서 편집하고자 하는 텍스트 박스를 선택하면 오른쪽 메모에 텍스트의 내용이 나타난다. 메모에 설정한 글꼴이나 윈도우의 위치, 최종 편집 파일의 인덱스 등은 철저하게 레지스트리에 저장되므로 항상성을 유지하도록 하였다. 참고로 새 텍스트 파일 만들기 기능이 제외되어 있는데 이는 파일 열기 대화상자에서 직접 할 수 있으므로 일부러 프로그래밍해 넣지 않았다.

레지스트리를 관리하는 것과 TStringList 오브젝트를 사용하는 것 외에는 특별한 기법이라고 할 만한 것도 없다. 분석해 보면 어렵지 않게 이 소스를 다 이해할 수 있을 것이다. 필자는 실제로 이 프로그램을 활용하고 있는데 아주 간단한 프로그램이지만 이 프로그램에 기능을 추가하면 자기에게 꼭 맞는 PIMS 정도는 만들 수 있을 것 같다.

# 제 14 장

# 여러 가지 컴포넌트

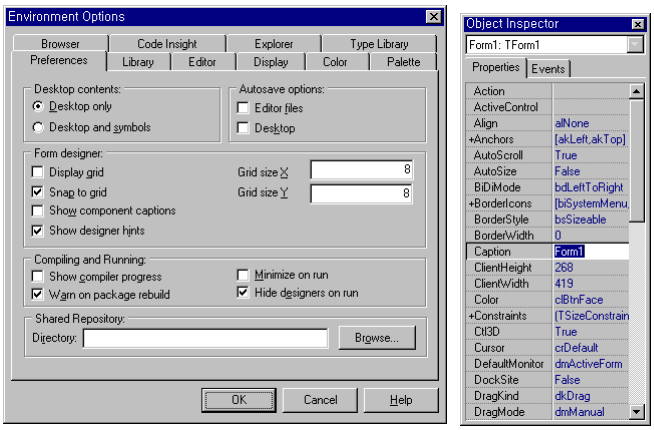


# 14-1 노트북 만들기

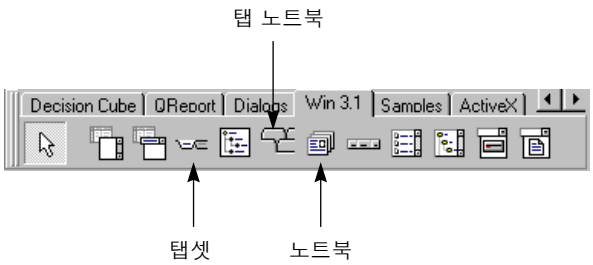
하나의 윈도우가 가지는 면적에는 제한이 있다. 아직까지도 표준 VGA를 사용하는 사람들이 많이 있기 때문에 대화상자의 크기가 640*480 이상이 될 수는 없으며 이 좁은 윈도우에 나타내고자 하는 모든 것을 다 나타내기가 무척이나 곤란한 경우가 있다. 간단한 질문이나 정보 입력 대화상자라면 별 문제가 없겠지만 입력해 주어야 할 사항이 많은 옵션 설정 대화상자의 경우는 윈도우 공간이 무척 비좁게 느껴진다.

그렇다고 대화상자를 두 개, 세 개 만들 수는 없는 노릇이므로 그 해결책으로 보통 하나의 윈도우에 여러 개의 윈도우를 포개어 사용하는 방식의 노트북을 사용한다. 노트북 아래(또는 위)에 페이지를 선택할 수 있는 탭을 달아 다음 페이지를 넘기는 듯한 방법으로 제한된 면적에 많은 것을 담는다. 그 예는 멀리서 찾을 것도 없이 델파이의 옵션 설정 대화상자나 코드 에디터, 오브젝트 인스펙터에서도 볼 수 있다. 요즘은 탭을 다는 것도 부족해 2단, 3단으로 탭을 배치하기도 하고 탭을 스크롤시키는 스크롤 버튼이 있는 것들도 있다(예:델파이의 컴포넌트 팔레트).

**그림**
노트북 사용예




델파이는 이런 노트북을 컴포넌트로 제공하므로 특별한 코드를 작성할 필요 없이 컴포넌트만 배치하여 간단하게 노트북을 디자인할 수 있다. 노트북 제작에 사용되는 컴포넌트는 Win31 페이지에 세 개나 있다.

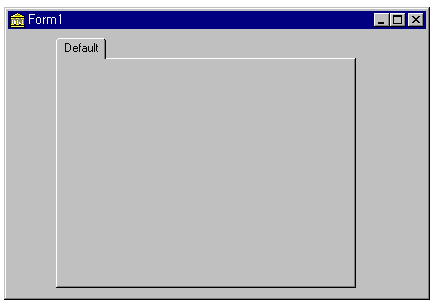



노트북과 탭셋은 주로 두 컴포넌트가 같이 사용되며 탭 노트북은 이 두 가지를 합쳐 놓은 형태이다. 이 중 사용하기가 간단한 탭 노트북부터 살펴보기로 하자.

# 가. 탭 노트북

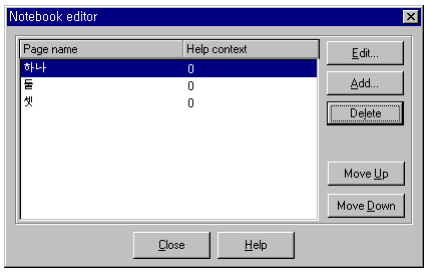
14jang
notebook

탭 노트북은 페이지를 전환할 수 있는 탭과 노트북을 같이 가지고 있는 컴포넌트이다. 폼에 배치하면 다음과 같이 하나의 페이지만을 가진 형태로 나타난다.

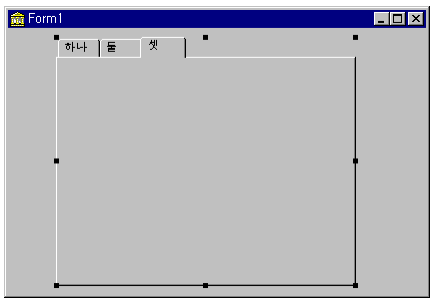



페이지를 추가하려면 Pages 속성에 페이지의 이름을 입력해 준다. 입력된 페이지 수만큼 페이지가 자동으로 생성되며 페이지의 이름이 상단의 페이지 탭에 나타난다. Pages 속성에 하나, 둘, 셋이라는 이름의 페이지를 입력해 보자. 오브젝트 인스펙터에서 Pages 속성값을 더블클릭하거나 ... 를 클릭하면 페이지를 편집할 수 있는 대화상자가 열린다.

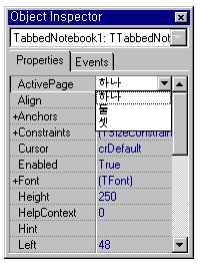
그림
페이지 편집 대화상자




Add 버튼으로 페이지 내용을 추가하며 Edit 버튼으로 이미 입력되어 있는 페이지의 내용을 편집한다. Help Context란은 도움말 정보를 기입하는 난이며 도움말이 없다면 특별히 값을 입력해 줄 필요가 없다. 하나, 둘, 셋을 입력하면 탭 노트북에는 세 개의 페이지가 생성된다.




윗부분의 탭은 프로그램 실행중에 사용자가 마우스 버튼으로 페이지를 교체할 때 사용된다. 프로그래머가 디자인 중에 페이지를 교체하려면 ActivePage 속성이나 PageIndex 속성을 사용한다. ActivePage 속성을 선택하면 오브젝트 인스펙터에 페이지 이름이 나열되며 전환하고자 하는 페이지를 선택하면 된다.




PageIndex 속성은 숫자로써 페이지를 선택할 수 있도록 해주며 첫 번째 페이지를 0번, 두 번째 페이지를 1번 등으로 번호를 매긴다. 페이지 전환을 위해 어떤 속성을 사용하든지 결과는 동일하지만 ActivePage 속성은 이름으로 선택하므로 디자인중에 사용하기 편리하며 PageIndex 속성은 숫자로 선택하므로 코

드에서 사용하기 편리하다.

탭 노트북의 외형을 결정짓는 속성으로는 TabFont 속성이 있다. TabFont 속성은 속성의 이름이 설명하듯이 페이지 탭에 사용할 폰트의 속성이다. 탭 제목이 너무 작게 출력되면 미관상 별로 좋지 못하므로 별도로 폰트를 설정할 수 있도록 되어 있다.

그림
TabsPerRow 속성에 따른 탭 모양

탭 노트북은 다른 컴포넌트를 포함할 수 있는 컨테이너(container) 컴포넌트이며 각 페이지별로 다르게 디자인할 수 있다. 페이지 하나가 마치 독립적인 폼처럼 사용된다. 다음 예는 세 개의 페이지로 구성된 노트북이다.

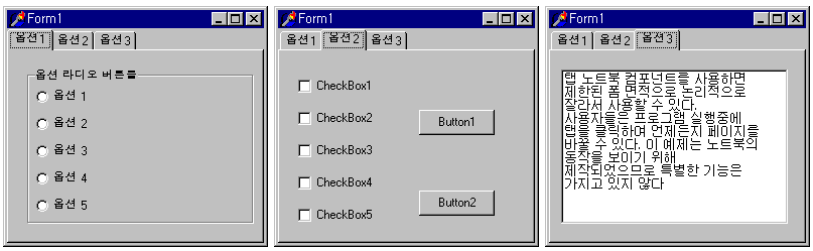


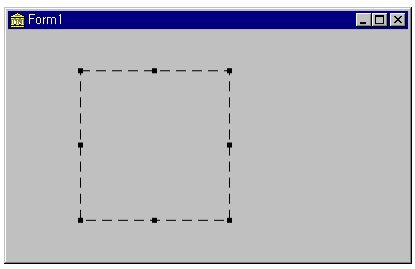

그림
세 개의 다른 페이지가 겹쳐 있다.

노트북의 페이지에 컴포넌트를 배치하고 코드를 작성하는 방법은 폼에서 작업하는 방법과 동일하며 특별한 제한이 없다. 디자인하는 사람의 필요와 능력에 따라 얼마든지 다른 구성을 만들어 낼 수 있다.

## 나. 탭셋

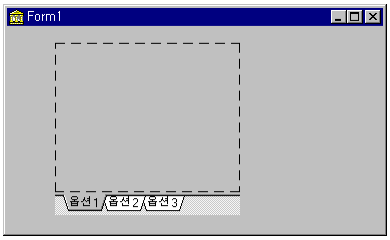

14jang
tabset

TabbedNotebook 컴포넌트는 페이지 탭이 컴포넌트 안에 포함되어 있어서 페이지를 만들거나 관리하기가 무척 쉬운 편이다. 이에 비해 Notebook 컴포넌트는 페이지 탭이 없고 페이지만을 가진다. 이 컴포넌트를 폼에 배치하면 컴포넌트의 모양은 보이지 않고 다음과 같이 점선만 나타난다.




컨테이너 컴포넌트이므로 다른 컴포넌트를 내부에 포함할 수 있으며 탭 노트

북과 마찬가지로 Pages 속성을 통해 여러 개의 페이지를 정의한다. 디자인중에 페이지를 전환하는 방법도 탭 노트북과 마찬가지로 ActivePage 속성이나 PageIndex 속성을 사용한다. 하지만 여러 개의 페이지를 만들어도 페이지 제목이 밖으로 드러나지 않으며 사용자가 실행중에 페이지를 교체할 수 있는 방법도 제공하지 않는다. 그래서 이 컴포넌트는 홀로 사용되는 경우가 드물며 페이지를 교체해 주는 탭셋과 함께 사용되어야 한다. 탭셋을 폼에 배치하면 탭이 없는 사각형 모양만 나타나며 Tabs 속성에 페이지 이름을 입력해 주면 탭이 생성된다.




하지만 탭셋의 Tabs 속성을 이렇게 디자인중에 직접 정의할 필요가 없다. 왜냐하면 탭셋은 노트북과 함께 사용되며 보통 노트북의 페이지와 같은 페이지를 가지기 때문이다. FormCreate 등의 이벤트 핸들러에서 페이지 이름을 노트북으로부터 대입받으면 된다.

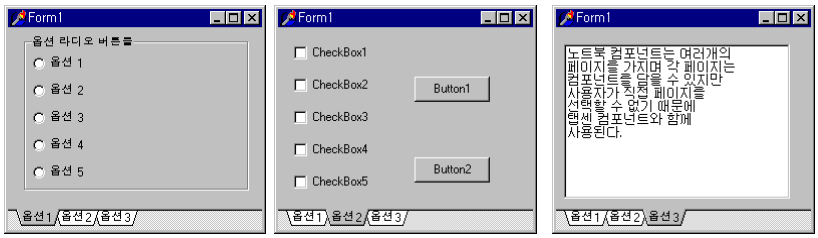
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TabSet1.Tabs:=Notebook1.Pages;
end;
```

그래서 페이지는 노트북에 디자인해 두고 탭셋은 노트북의 페이지를 실행중에 그대로 받아서 사용하기만 한다. 페이지의 교체도 코드를 사용하며 탭셋이 클릭될 때인 OnClick 이벤트에서 선택된 페이지로 노트북의 페이지를 전환해 준다.

```
procedure TForm1.TabSet1Click(Sender: TObject);
begin
  Notebook1.PageIndex := TabSet1.TabIndex;
end;
```

노트북과 탭셋을 사용하면 앞에서 만들어 보았던 탭 노트북과 똑같은 기능의 프로그램을 만들 수 있다. 단 두 개의 컴포넌트가 같이 사용되므로 정렬을 잘 해주어야 한다. 보통 탭셋의 Align 속성은 폼의 바닥(alBottom)으로 설정하고 노

트북이 폼의 나머지 영역을 사용할 수 있도록 Align 속성을 alClient로 설정한다.

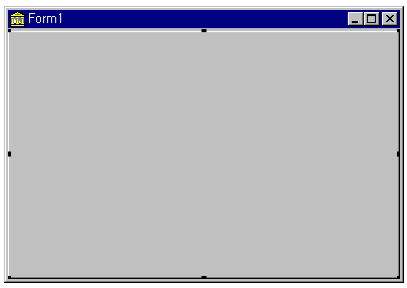



탭 노트북 컴포넌트 하나를 사용하는 경우와 노트북, 탭셋을 사용하는 경우의 차이점이라면 페이지 탭이 윗쪽이 아닌 아래쪽에 있다는 점과 탭셋에 좀 더 많은 장식을 할 수 있다는 점이다. Tabset 컴포넌트는 모양을 꾸미는 많은 속성을 가지고 있다. 탭의 색상, 폰트, 정렬, 크기를 일일이 지정할 수 있으며 심지어 비트맵을 포함시킬 수도 있다. Tabset 컴포넌트의 속성에 관해서는 레퍼런스를 참조하기 바란다.

# 다. 페이지 컨트롤

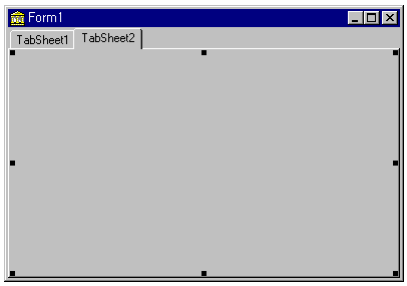
14jang PageControl

중첩된 페이지를 만드는 또 하나의 컨트롤이 있는데 Win32 페이지에 있는 페이지 컨트롤(TPageControl)이다. 이 컴포넌트는 사실 탭 노트북과 기능적으로나 용도상으로도 중복된 컨트롤이지만 탭 노트북은 델파이 고유의 컨트롤이고 페이지 컨트롤은 Win32 API에 정의된 컨트롤이라는 점이 다를 뿐이다. 이 컨트롤도 사용하는 방법은 지극히 간단하다. 일단 새 프로젝트를 시작한 후 페이지 컨트롤을 폼에 배치하고 alClient 정렬하여 폼에 가득 차게 채운다.

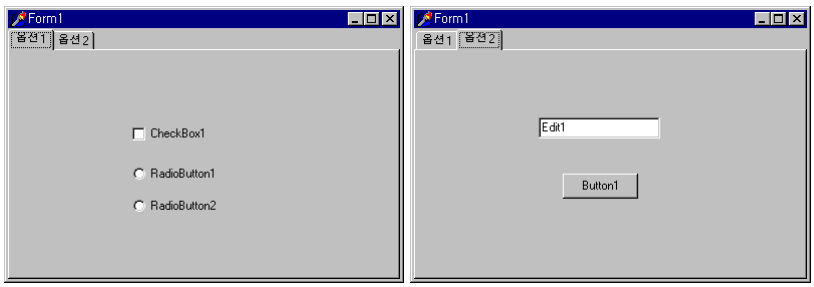



처음 폼에 배치될 때는 페이지가 하나도 정의되어 있지 않으므로 사각형 모양으로 보일 것이다. 새로운 페이지를 만들려면 팝업 메뉴에서 New Page 항목을

선택하면 된다. 이 항목을 두 번 선택하여 두 개의 페이지를 추가해 보자. 그러면 위쪽에 두 개의 탭이 생길 것이다.




새로운 페이지가 생성되면 페이지 컨트롤 내부에 탭 시트 컴포넌트가 배치되는데 이 컴포넌트는 페이지 컨트롤로부터 생성되는 내부 컴포넌트이며 페이지 컨트롤 자체와는 다르다. 마우스로 선택해 보면 탭 시트와 페이지 컨트롤이 각각 선택되며 오브젝트 인스펙터에 나타나는 속성이 다르다는 것을 알 수 있을 것이다. 각 페이지의 캡션은 탭 시트 컴포넌트의 Caption 속성으로 설정한다. 디자인 중에 페이지를 교체할 때는 페이지 컨트롤을 선택한 상태에서 ActivePage 속성에 활성화시킬 탭 시트를 선택하거나 아니면 탭을 직접 눌러도 된다. 두 페이지의 캡션을 변경하고 각각 다른 컨트롤들을 이것 저것 배치해 보도록하자.




탭 노트북의 경우와 마찬가지로 별로 어렵지 않을 것이다.

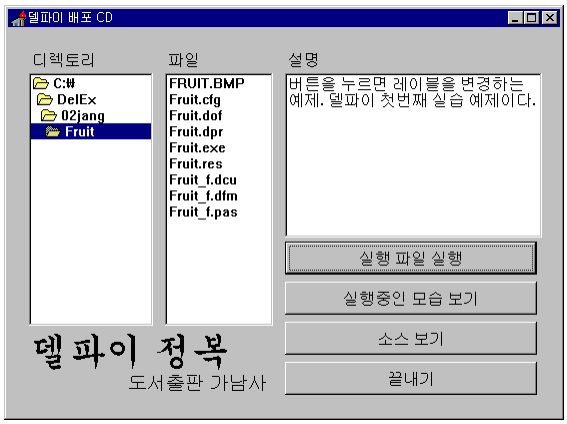
# 라. DelCD 예제


14jang
DelCD

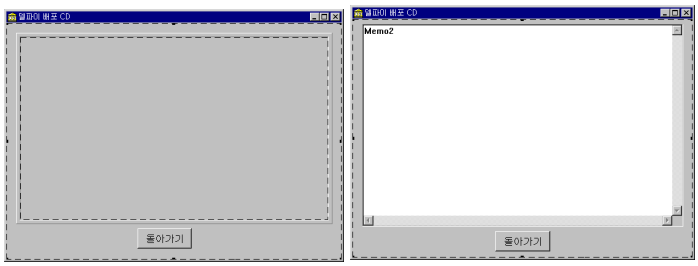
이 책의 모든 예제는 부록으로 포함된 CD-ROM으로 제공되며 CD-ROM의 DelEx 디렉토리 아래에 예제들이 정리되어 있다. 그리고 DelEx 디렉토리에는 DelCD.exe라는 실행 파일이 있어 이 프로그램으로 예제들을 살펴볼 수 있도록 하였다.

**그림**

세 개의 페이지로 구성된 CD 검색 프로그램




실행중에 소스나 실행 모습을 직접 확인해 볼 수 있도록 되어 있는데 이 때 실행중인 모습 보기, 소스 보기 버튼을 누르면 화면이 전환되면서 비트맵과 소스를 보여준다. 이런 페이지 전환 기법은 노트북 컴포넌트를 사용하면 쉽게 구현할 수 있다. 노트북 컴포넌트를 폼 전체에 가득 채워 두고 세 개의 페이지를 만든 후 각 페이지에 이미지, 메모 등을 배치해 두고 페이지를 빠르게 전환하는 방법을 사용하는 것이다. 메인 화면이 page 1이며 그림 보기, 소스 보기는 각각 page 2, page 3로 되어 있다. 노트북을 선택한 후 ActivePage 속성을 변경해 보면 무슨 말인지 알 수 있을 것이다.




이 예제에는 페이지를 교체하는 것 외에 파일로부터 데이터를 읽어들이는 기법 등 배울만한 것들이 꽤 있으므로 분석해 보기 바란다. 전체 소스만 보인다.

```
unit Decd_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, FileCtrl, ExtCtrls;
```

```
type
  TCDForm = class(TForm)
    Notebook1: TNotebook;
    DirectoryListBox1: TDirectoryListBox;
    FileListBox1: TFileListBox;
    Memo1: TMemo;
    BtnExit: TBitBtn;
    BtnExec: TBitBtn;
    BtnBMP: TBitBtn;
    BtnSrc: TBitBtn;
    BitBtn1: TBitBtn;
    Panel1: TPanel;
    Image1: TImage;
    BitBtn2: TBitBtn;
    Memo2: TMemo;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DirectoryListBox1Change(Sender: TObject);
    procedure BtnExecClick(Sender: TObject);
    procedure BtnExitClick(Sender: TObject);
    procedure BtnBMPClick(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure BtnSrcClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

const
  totalprg=140; {프로그램의 총 개수는 140 개}
type
  prg=record
    path:string[30];  {디렉토리 경로}
    exec:string[15];  {실행 파일명}
    src:string[15];   {보여줄 소스 파일}
    desc:string;      {설명}
  end;
type
  infotype=array[0..totalprg] of prg;
```

```
var
  CDForm: TCDForm;
  info:^infotype;     {프로그램 정보 배열}
  nowsel:integer;     {현재 선택된 프로그램}
  cddrive:string[5];  {CD-ROM 드라이브}

implementation

{$R *.DFM}

{선형 검색으로 사용자가 검색한 프로그램을 찾아낸다.}
procedure searchwhat;
var
  i:integer;
  st:string;
begin
for i:=1 to totalprg do
  begin
  st:=info^[i].path;
  st:=cddrive+st;
  if comparetext(st,CDForm.directorylistbox1.directory)=0
    then break;
  end;
nowsel:=i;
end;

procedure TCDForm.FormCreate(Sender: TObject);
var
  F1:TextFile;
  i:integer;
  st:string;
begin
{동적으로 레코드 생성}
new(info);
{파일로부터 정보를 읽어 들임}
AssignFile(F1,'index.txt');
reset(F1);
for i:=1 to totalprg do
  begin
  readln(f1,st);info^[i].path:=st;
  readln(F1,st);info^[i].exec:=st;
  readln(F1,st);info^[i].src:=st;
  readln(F1,st);info^[i].desc:=st;
  end;
CloseFile(F1);
Memo1.Text:='';
```

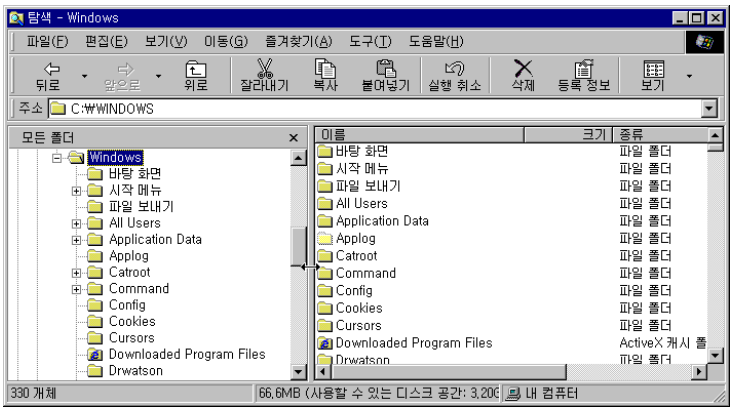
```
cddrive:=Copy(ParamStr(0),0,2);
NoteBook1.PageIndex:=0;
end;

procedure TCDForm.FormDestroy(Sender: TObject);
begin
dispose(info);
end;

procedure TCDForm.DirectoryListBox1Change(Sender: TObject);
begin
searchwhat;
if nowsel=totalprg+1 then
  Memo1.Text:='예제가 있는 디렉토리가 아닙니다. 다른 디렉토리를 선택하십시요.'
else
  Memo1.Text:=info^[nowsel].desc;
end;

procedure TCDForm.BtnExecClick(Sender: TObject);
var
  pst:array [0..128] of Char;
begin
searchwhat;
if nowsel=totalprg+1 then
ShowMessage('예제가 있는 디렉토리가 아닙니다.')
else
  begin
  strPCopy(pst,info^[nowsel].path+'₩'+info^[nowsel].exec);
  WinExec(pst,SW_SHOWNORMAL);
  end;
end;

procedure TCDForm.BtnExitClick(Sender: TObject);
begin
close;
end;

procedure TCDForm.BtnBMPClick(Sender: TObject);
var
  st:string;
begin
searchwhat;
if nowsel=totalprg+1 then
ShowMessage('예제가 있는 디렉토리가 아닙니다.')
else
  begin
```

```
  st:=ChangeFileExt(info^[nowsel].path+'₩'+info^[nowsel].exec,'.BMP');
  image1.picture.loadFromFile(st);
  NoteBook1.PageIndex:=1;
  end;
end;

procedure TCDForm.BitBtn1Click(Sender: TObject);
begin
NoteBook1.PageIndex:=0;
end;

procedure TCDForm.BtnSrcClick(Sender: TObject);
var
  st:string;
begin
searchwhat;
if nowsel=totalprg+1 then
ShowMessage('예제가 있는 디렉토리가 아닙니다.')
else
  begin
  st:=info^[nowsel].path+'₩'+info^[nowsel].src;
  memo2.lines.LoadFromFile(st);
  NoteBook1.PageIndex:=2;
  end;
end;

end.
```

# 14-2 화면 분할

좁은 윈도우 화면을 넓게 사용하는 방법에는 여러 가지가 있다. 앞에서 보인 예와 같이 여러 가지 페이지로 화면을 겹쳐서 쓰는 방법이 있는가 하면 윈도우즈의 탐색기처럼 화면을 분할하여 여러 가지 정보를 분할된 화면에 출력하는 방법도 있다. 탐색기 외에도 워드, 엑셀 등의 많은 상용 프로그램들이 이런 기법을 사용한다.

그림
화면을 분할하여 사용하는 탐색기




화면을 분할했을 때는 분할된 화면의 크기를 사용자가 마음대로 변경할 수 있는 방법을 제공해 주어야 하며 보통 마우스로 경계 부근을 드래그하는 방법이 사용된다.

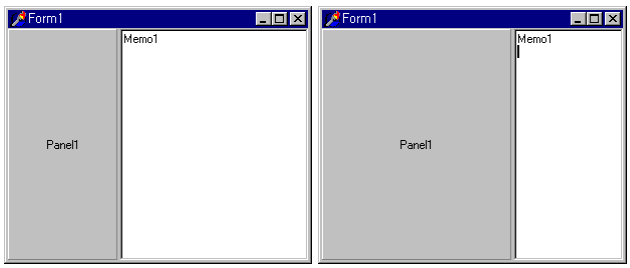
## 가. 스플리터

14jang
Splitter

델파이에서 이런 화면 분할 기법을 구현하려면 여러 가지 방법이 있지만 가장 간단한 방법은 Additional 페이지에 있는 스플리터 컴포넌트를 사용하는 것이다. 스플리터는 미리 배치된 컴포넌트와 같은 정렬을 유지하면서 실행중에 사용자가 마우스를 드래그하여 컴포넌트의 크기를 변경할 수 있도록 해 준다.

사용하는 방법은 아주 간단하다. 컴포넌트를 먼저 배치하고 같은 정렬 속성으로 스플리터를 배치하기만 하면 된다. 간단한 예제를 만들어 이 컴포넌트를 테스트해 보도록 하자. 새 프로젝트를 시작한 후 패널을 하나 배치하고 이 패널의 Align 속성을 alLeft 로 설정하여 폼의 좌측 벽에 밀착시킨다. 그리고 Additonal

페이지에 있는 스플리터 컴포넌트를 폼에 배치하면 스플리터는 자동으로 alLeft 정렬되어 패널의 오른쪽 변에 밀착될 것이다. 폼의 나머지 영역은 메모 컴포넌트를 배치한 후 alClient 정렬을 해 보자. 그리고 예제를 실행하면 패널과 메모가 폼을 분할하며 가운데 스플리터가 배치된다.

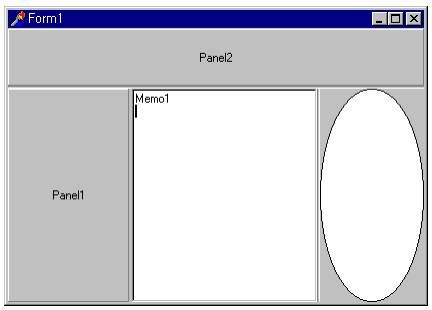



이 상태에서 사용자는 경계 부근의 스플리터를 드래그하여 두 컴포넌트의 화면 배치를 바꿀 수 있다.

Align 속성만 같게 만들어주면 여러 개의 스플리터를 배치하여 사용할 수 있는데 이 때 제일 끝에 배치되는 컨트롤은 alClient 정렬을 지정하여 폼의 나머지 영역을 다 채우도록 해야 논리적으로 문제없이 크기 조정이 된다. 다음 예제는 4 개의 컴포넌트와 3 개의 스플리터를 사용하여 폼을 4 등분해본 것이다. 각 컴포넌트의 정렬 상태만 주의해 주면 이런 분할 폼은 아주 쉽게 만들 수 있다.

**그림**

스플리터를 이용한 화면 분할




스플리터 컴포넌트가 마우스 드래그까지 다 처리해 주므로 화면 분할을 위해 코드를 작성할 필요가 없다. 사용 방법이 비교적 쉬우므로 속성만 간단하게 정리하도록 하자.

### Beveled

푹 파여진 듯한 입체적인 모양을 만들 것인가를 지정하는 옵션이다.

✍ MinSize

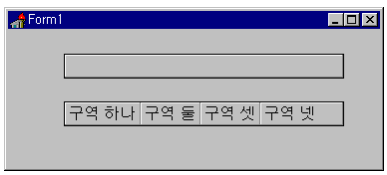
크기 조정이 가능한 최소 크기를 지정하며 바꾸어 말하면 이웃 컨트롤의 크기를 MinSize 이하로는 만들지 않도록 한다. 디폴트로 이 값은 30 이므로 스플리터의 근접 컨트롤은 최소 30 픽셀의 폭(또는 높이)을 확보하게 된다.

✍ Cursor

스플리터 위에 마우스 커서가 위치할 때 보여줄 마우스 커서 모양을 지정한다. 디폴트는 이되 수직선으로 분할된 스플리터의 경우는 로 변경해 주는 것이 자연스럽다.

# 나. 헤더

스플리터 컴포넌트 외에 화면을 분할하는 방법으로는 Header 컴포넌트를 사용하는 방법이 있다. Header 컴포넌트는 크기 조정이 가능한 구역 분할 컴포넌트이며 폼에 배치하면 구역이 정의되어 있지 않기 때문에 패널처럼 보인다. 헤더의 구역은 Sections 속성으로 정의하며 구역 이름을 입력해 주기만 하면 된다. 다음 그림은 헤더를 최초 배치했을 때와 네 개의 구역으로 나누었을 때의 모습이다.



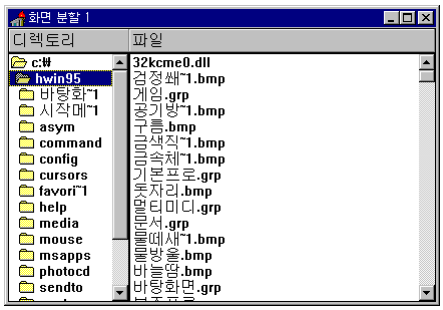

헤더의 각 구역은 다음과 같은 방법으로 크기를 변경할 수 있다.

| 상황 | 변경 방법 |
|---|---|
| 실행시 | 마우스의 왼쪽 버튼으로 헤더의 경계선을 드래그 |
| 디자인시 | 마우스의 오른쪽 버튼으로 헤더의 경계선을 드래그 |
| 코드 | Header.SectionWidth[n] 속성을 변경한다. |

실행시 사용자가 마우스로 경계선을 드래그할 때 OnSizing 이벤트가 발생하

14jang
split1

며 드래그가 종료되고 크기 변경이 완료되면 OnSized 이벤트가 발생한다. 이 두 가지 이벤트를 사용하여 변경된 헤더의 크기에 맞게 화면을 적절히 분할하는 코드를 작성한다. 다음 예제는 파일 리스트 박스와 디렉토리 리스트 박스를 폼에 배치하고 헤더를 사용하여 두 리스트 박스의 크기를 변경한다. 마치 윈도우즈의 파일관리자와 비슷하다.




헤더를 폼에 배치시키고 Align 속성을 alTop으로 설정하여 상단에 밀착시키며 디렉토리 리스트 박스는 좌측(alLeft)에 밀착시키고 파일 리스트 박스는 폼의 나머지 영역을 모두 사용할 수 있도록 alClient로 정렬한다. 디렉토리 리스트 박스의 FileList 속성에 파일 리스트 박스를 지정하여 두 리스트 박스를 연결해 준다. 헤더의 Sections 속성에는 "디렉토리", "파일" 두 개의 문자열을 대입하여 두 개의 구역으로 나눈다. 헤더의 OnSized 이벤트 핸들러 코드를 다음과 같이 작성하면 프로그램이 완성된다.

```
procedure TForm1.Header1Sized(Sender: TObject; ASection, AWidth: Integer);
begin
  DirectoryListBox1.Width:=AWidth;
end;
```

헤더의 크기가 변경되면 변경된 크기대로 디렉토리 리스트 박스의 크기를 조정한다. 파일 리스트 박스는 alClient로 정렬되어 있으므로 별도로 크기를 변경하지 않아도 나머지 영역을 채울 정도로 크기가 자동으로 조정된다.

OnSized 이벤트는 두 개의 인수를 사용한다. 첫 번째 인수인 ASection은 크기가 변경된 구역의 인덱스 번호이며 이 인수가 0이면 첫 번째 구역, 1이면 두 번째 구역이 변경되었음을 알린다. 두 번째 인수 AWidth는 변경된 후의 구역 크기를 픽셀 단위로 나타낸다. 여기까지 코드를 작성하면 헤더를 사용하여 두 리스트 박스의 크기를 마음대로 변경할 수 있지만 크기를 적당히 제한해야 하는 한 가지 문제가 더 남아 있다. 크기를 마음대로 조절하도록 하되 너무 작거나 너

무 크게 만들면 다른 구역이 보이지 않게 되므로 최소한의 크기를 정해 주는 것이 좋으며 이런 동작은 헤더의 크기가 변경되는 중에 발생하는 OnSizing 이벤트 핸들러에서 수행한다. 다음 코드를 보자.

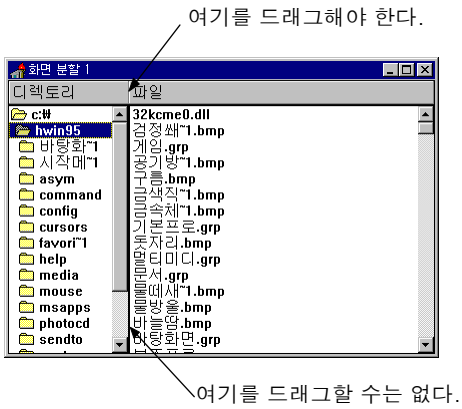
```
procedure TForm1.Header1Sizing(Sender: TObject; ASection, AWidth: Integer);
begin
if AWidth<100 then {최소 100 의 크기는 되어야 한다.}
  begin
  ReleaseCapture;  {마우스 드래그 종료}
  Header1Sized(Sender,ASection,100);
  Header1.SectionWidth[0]:=100;
  end;
if AWidth>ClientWidth-100 then
  begin
  ReleaseCapture;
  Header1Sized(Sender,ASection,ClientWidth-100);
  Header1.SectionWidth[0]:=ClientWidth-100
  end;
end;
```

디렉토리 리스트 박스의 크기가 최소한 100 이상이어야 하며 또 파일 리스트 박스의 크기도 최소한 100은 되어야 한다. 만약 그렇지 못할 경우는 ReleaseCapture 함수를 호출하여 강제로 드래그를 종료시키며 구역의 크기를 100으로 설정해 준다.

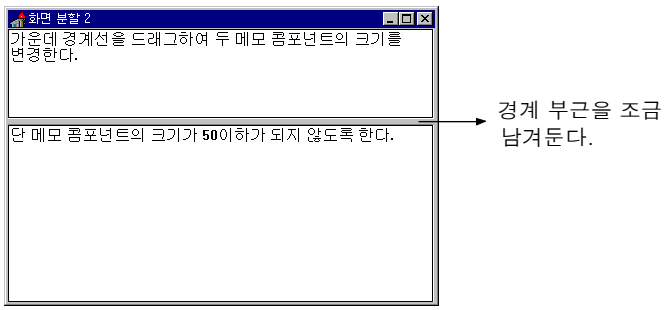
## 다. 수직 분할

14jang
split2

헤더 컴포넌트는 구역 분할 뿐만 아니라 마우스 동작까지 내부에서 지원하므로 화면을 분할하기에는 아주 편리하게 사용할 수 있다. 그러나 단점이 없는 것도 아니다. 우선 반드시 헤더를 드래그해야 하며 화면의 경계선을 드래그할 수 없다는 점이 무척 불편하다.




그리고 수평으로 화면을 분할할 수 있을 뿐이며 수직으로는 분할할 수 없다. 수직으로 화면을 분할하려면 스플리터 컴포넌트를 사용하는 것이 제격이겠지만 폼이나 패널의 마우스 이벤트를 직접 프로그래밍해도 된다. 마우스 이벤트를 사용하여 화면을 수직으로 분할하는 예제를 만들기 위해 새로운 프로젝트를 시작하고 메모 컴포넌트 두 개를 다음과 같이 배치한다.




폼을 드래그하여 경계를 조정하므로 메모의 경계 부근에 폼이 조금이라도 드러나 있어야 하며 드래그를 위해 폼의 속성을 다음과 같이 설정한다.

| 속성 | 속성값 | 설명 |
|---|---|---|
| Caption | 화면 분할 2 | |
| Cursor | crVSplit | 수직으로 드래그하는 커서( ) |
| AutoScroll | False | 메모 컴포넌트의 크기 조정중에 깜박거림 현상을 제거한다. |

메모 컴포넌트 사이에 드러난 폼 영역으로 마우스를 가져가면 커서가  모양으로 바뀌며 경계 영역에서 폼을 드래그하면 변경된 폼의 크기에 따라 아래위의 메모 컴포넌트를 조절한다. 전체 소스는 다음과 같다.

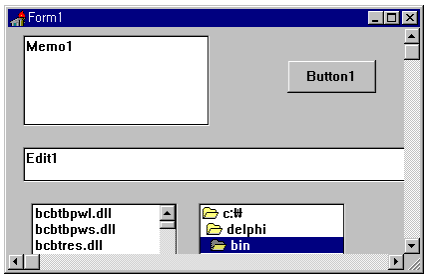
```
unit Split2_f;

interface

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

type
 TForm1 = class(TForm)
   Memo1: TMemo;
   Memo2: TMemo;
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
   procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
     Y: Integer);
   procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
 private
   { Private declarations }
 public
   { Public declarations }
 end;

var
 Form1: TForm1;
 nowdrag:Boolean;    {현재 드래그 중인가?}

implementation

{$R *.DFM}

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
nowdrag:=True;   {드래그 중임을 표시}
end;

{드래그 중일 때 마우스를 움직이면 메모의 크기를 변경한다.}
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
```

```
  Y: Integer);
begin
if (nowdrag) and          {드래그 중이이어야 함}
   (y<ClientHeight-50) and  {아래쪽 메모의 최소 크기는 50}
   (y>50) then            {윗쪽 메모의 최소 크기는 50}
  begin
  {마우스의 수직 위치에 따라 메모의 크기 변경}
  Memo1.Height:=y-2;
  Memo2.Height:=ClientHeight-y-5;
  Memo2.Top:=y+2;
  end;
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
nowdrag:=False;  {드래그 끝}
end;

end.
```

우선 nowdrag라는 변수를 선언하고 마우스를 누른 상태에서만 드래그가 되도록 하기 위해 MouseDown에서 이 값을 True로 만들고 MouseUp에서 이 값을 False로 바꾼다. MouseMove에서 경계를 결정하기 위해 마우스의 수직 좌표인 Y를 사용하며 메모 컴포넌트의 크기가 50이하가 되지 않도록 하고 있다.

# 라. 스크롤 박스

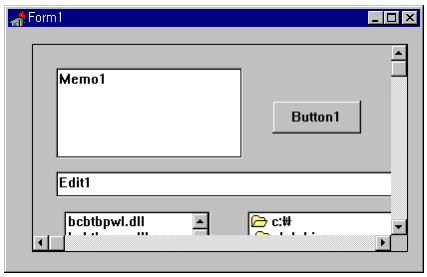
좁은 화면을 넓게 쓰는 전통적인 방법 중 하나가 넓은 영역을 이동해 가며 사용할 수 있는 스크롤 방법이다. 만약 폼이 가지고 있는 컴포넌트가 폼의 면적보다 더 넓게 배치되어 있다면 폼에 자동적으로 스크롤 바가 나타나며 실행중에 스크롤 바를 사용하여 폼의 다른 부분으로 이동할 수 있다.




폼의 스크롤 바를 직접 사용할 경우는 폼 전체가 스크롤되며 폼 중의 일부만 스크롤의 대상이 되도록 할 수 없다. 심지어 스크롤되지 말아야 할 툴바나 상황선까지도 스크롤의 대상이 되어 문제가 되기도 한다. 이런 문제를 해결하려면 스크롤 박스 컴포넌트를 사용하여 폼 내부에 스크롤이 가능한 영역을 만들어 준다. 스크롤 박스는 다른 컴포넌트를 담을 수 있는 컨테이너 컴포넌트이며 스크롤 박스의 면적보다 더 넓게 컴포넌트를 배치하면 스크롤 바가 나타난다.

**그림**

스크롤 박스를 배치한 모습




이렇게 스크롤 박스를 배치해 두고 스크롤 되지 않아야 할 상황선이나 툴바는 스크롤 박스의 바깥쪽에 배치해 두면 스크롤 영역을 일정하게 제한할 수 있다. 스크롤 박스에 관한 좀 더 자세한 사항은 레퍼런스를 참조하기 바란다. 생각보다는 간단한 컴포넌트이다.

# 14-3 아웃 라인

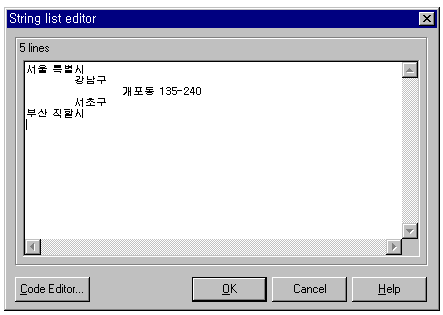
아웃 라인은 다중 레벨(multilevel)의 데이터 구조를 보관하고 보여주며 관리하는 컴포넌트이다. 자주 사용되지는 않지만 복잡한 데이터를 시각적으로 보여준다는 점에 있어서 아주 유용한 컴포넌트이다. 다중 레벨의 데이터 구조란 각각의 자료들이 등급을 이루어 트리 형태로 구성되어 있는 구조를 말하며 가장 흔한 예로 도스가 사용하는 디렉토리 구조를 들 수 있다. 즉 루트가 있고 루트 안에 서브가 있고 서브 안에 또 서브가 있는 식의 계층적인 구조이다.

## 가. 자료의 입력

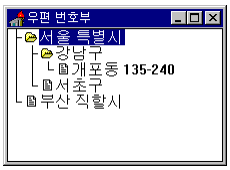
디자인시에 아웃 라인에 데이터를 입력하려면 Lines 속성을 사용한다. Lines 속성은 문자열 리스트(TStrings)형이므로 디자인시에 문자열 리스트 편집기로 편집할 수 있으며 실행시에 문자열 리스트 관리에 사용되는 Add, Insert, Delete 등의 메소드들을 그대로 사용할 수 있다.

14jang
outpost

아웃 라인을 사용하는 가장 적합한 예는 우편번호부를 들 수 있다. 우편번호부는 도(시)-군(구)-면(동)의 3단계 구조를 가지고 있으며 정확하게 트리 구조를 이루고 있으므로 아웃 라인에 기억시킬 수 있고 아웃 라인으로 쉽게 그 내용을 열람할 수 있다. 새로운 프로젝트를 시작하고 폼의 캡션을 “우편 번호부”로 고치고 아웃 라인 컴포넌트를 배치한 후 Align 속성을 alClient로 설정하여 아웃 라인이 전체 폼 영역을 차지하도록 한다. 그리고 오브젝트 인스펙터에서 Lines 속성을 더블클릭하여 문자열 리스트 편집기를 불러낸다. 일단 다음과 같이 입력해 보자.



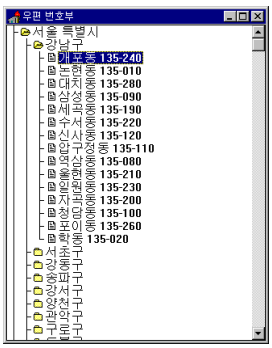
각 항목의 등급은 들여쓰기로 표현하며 들여쓰기에는 Tab 또는 스페이스를 사용하되 가급적이면 Tab을 쓰는 것이 읽기가 쉽다. Tab을 전혀 쓰지 않으면 이 데이터는 1등급이 되며 Tab을 하나 쓰면 2등급, 두 개 쓰면 3등급이 된다. 위에서 입력한 데이터 중 “서울 특별시”는 1등급이며 “강남구”와 “서초구”는 2등급, “개포동”은 3등급이다. 단 문자열 리스트 편집기에서 Tab키를 누르면 Tab이 입력되는 것이 아니라 대화상자 내에서 포커스를 이동시키므로 Tab 입력을 위해서는 반드시 Ctrl+Tab을 눌러야 한다. 이렇게 문자열을 입력시키면 폼의 아웃라인에는 다음과 같이 입력한 정보가 계층적으로 표현될 것이다.




마치 디렉토리 트리 구조와 비슷한 모양을 가지고 있다. 프로그램을 실행시키면 1등급의 항목만 나타나며 항목을 더블클릭하면 하위 항목이 열린다. 윈도우즈의 탐색기나 파일 관리자의 디렉토리 리스트를 이용하는 방법과 동일하다. 아웃 라인을 사용하는 방법은 이렇게 간단하지만 제일 큰 문제는 자료를 입력하는 방법이다. 알다시피 우리나라 우편번호부를 전부 입력해 넣는 일은 쉬운 일이 아니다. 필자도 끝내 다 입력시키지 못하고 일부만 입력해 넣었다. 나머지는 심심한 사람들이 직접 입력해 보기 바란다. 아니면 대중 통신망을 뒤져보면 미리 입력되어져 있는 우편번호부가 있을 것이다. 완성된 우편번호부의 실행 모습은 다음과 같다.

**그림**

아웃 라인으로 만들어본 우편번호부




데이터를 입력하는 일이 좀 번거로울 뿐이지 단 한 줄의 코드도 작성하지 않고 이런 멋있고 사용하기 편리한 우편번호부를 만들었다.

# 나. 아웃 라인의 속성

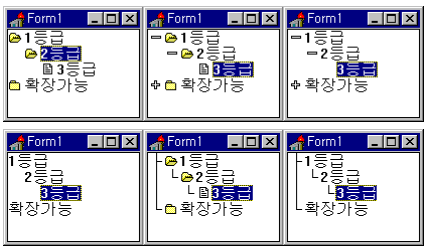
복잡한 자료를 표현하는만큼 아웃 라인은 겉모양을 꾸미기 위한 많은 속성을 가지고 있다.

### ✍ OutlineStyle

계층적으로 자료를 보여주는 방법을 정한다. 다음과 같은 형식이 있다. 속성값 자체가 무척 설명적이므로 굳이 표를 볼 필요도 없다.

| 속성값 | 출력 |
|---|---|
| osPictureText | 그림과 항목의 텍스트 |
| osPlusMinusPictureText | 확장 가능, 축소 가능 표시와 그림과 텍스트 |
| osPlusMinusText | 확장 가능, 축소 가능 표시와 텍스트 |
| osText | 텍스트만 보여준다. |
| osTreePictureText | 트리 구조를 선으로 연결하며 텍스트와 그림을 보여준다. |
| osTreeText | 트리 구조를 선으로 연결하며 텍스트만 보여준다. |

다음에 각 속성값의 예를 간단하게 그림으로 정리하였다. 역시 디폴트가 제일 보기에 좋은 것 같다.




### ✍ 그림 선택

아웃 라인에서 사용하는 비트맵에는 다섯 가지 종류가 있으며 사용자가 마음에 드는 비트맵으로 바꿀 수 있다. 비트맵에 관계된 속성은 다음과 같다. 이 속성을 더블클릭하면 비트맵 버튼의 경우와 마찬가지로 비트맵을 읽어올 수 있는 대화상자가 열린다.

**표**
아웃 라인에서 사용하는 비트맵

| 속성 | 의미 | 비트맵 |
|---|---|---|
| PictureClosed | 닫혀 있는 항목 | |
| PictureLeaf | 최하위 항목 | |
| PictureMinus | 축소 가능 항목 | |
| PictureOpen | 열려 있는 항목 | |
| PicturePlus | 확장 가능 항목 | |

### Options

아웃 라인의 모양이나 행동 방식을 설정하는 속성이다. Options 속성에 속한 다음 세 가지 세부 속성이 있다.

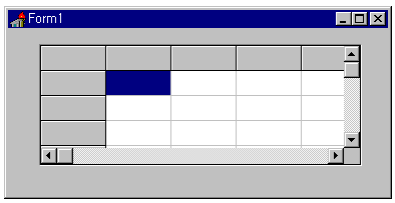
| 속성 | 의미 |
|---|---|
| ooDrawTreeRoot | 첫 번째 항목을 루트 항목으로 연결한다. |
| ooDrawFocusRect | 선택된 항목에 사각형의 테두리를 그린다. |
| ooStretchBitmaps | 그림을 폰트의 크게에 맞게 늘린다. 폰트의 크기가 커지면 그림도 같이 크게 만드는 것이 훨씬 더 보기에 좋다. |

# 14-4 그리드

## 가. 문자열 그리드

그리드는 문자열을 담을 수 있는 바둑판 모양의 격자로 된 컴포넌트이다. Lotus 1-2-3나 Excel 등, 스프레드 시트 프로그램의 워크 시트를 생각하면 그리드의 모양과 기능, 그리고 사용 용도에 관해 쉽게 짐작할 수 있을 것이다. 그리드를 폼에 배치하면 다음과 같이 5*5의 가로 세로 격자를 가진 컴포넌트가 나타난다.

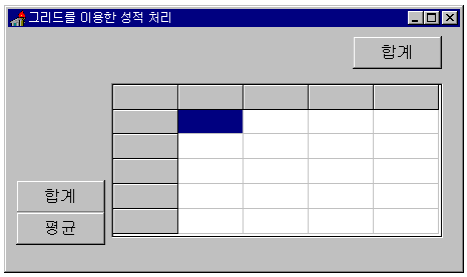
**그림**
문자열 그리드를 폼에 배치한 모양





14jang
sung

그리드의 각 격자를 셀(Cell)이라고 하며 셀 하나에 하나의 문자열을 입력할 수 있다. 각 셀은 Cells 속성으로 액세스할 수 있으나 디자인시에는 Cells 속성에 문자열을 대입할 수 없으므로 반드시 코드를 통해야만 문자열 그리드에 문자열을 입력할 수 있다. 예제를 만들어 보아야만 어떻게 문자열 그리드를 사용하는가를 알 수 있을 것 같다. 여기서 만들 예제는 워크 시트를 흉내낸 간단한 성적 처리 프로그램이다. 다음과 같이 버튼 세 개와 문자열 그리드 하나를 폼에 배치하고 속성을 조정한다.




| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | Caption | 그리드를 이용한 성적 처리 |

| | | |
|---|---|---|
| 그리드 | RowCount | 6 |
| | ScrollBars | ssNone |
| | Options/goEditing | True |
| 버튼들 | Name, Caption | 적당히 조절하세요 |

그리드의 디폴트 크기는 5*5이며 이 크기는 ColCount, RowCount 속성으로 변경한다. 이 예제에서는 디폴트 크기를 그대로 사용하고 있다. Options의 goEditing 세부 속성이 True이면  실행중에 사용자가 그리드에 문자열을 직접 입력 또는 수정할 수 있도록 한다. 전체 소스 코드는 다음과 같다.

```
{문자열 그리드를 이용한 간단한 성적 처리}
unit Sung_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, StdCtrls;

type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    Hap1: TButton;
    ave1: TButton;
    Hap2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Hap1Click(Sender: TObject);
    procedure ave1Click(Sender: TObject);
    procedure Hap2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
type
  TSung=array[1..3,1..3] of Integer;
{그리드에 입력할 성적과 학생 이름, 과목명을 미리 타입 상수로
정의해 둔다. 이 값들은 FormCreate 에서 그리드로 입력된다.}
const
  Sung:TSung=((100,98,96),(23,22,25),(78,81,75));
  Stud:array[1..4] of String=('김상형','김기문','김수동','합계');
```

```
  Subj:array[1..5] of String=('국어','영어','수학','합계','평균');
var
  Form1: TForm1;


implementation

{$R *.DFM}

{폼이 생성될 때 그리드에 문자열을 입력한다.}
procedure TForm1.FormCreate(Sender: TObject);
var
  i,j:integer;
begin
for i:=1 to 4 do   {학생 이름 입력}
  StringGrid1.Cells[i,0]:=Stud[i];
for i:=1 to 5 do   {과목 이름 입력}
  StringGrid1.Cells[0,i]:=Subj[i];
for i:=1 to 3 do   {성적 입력}
  for j:=1 to 3 do
    StringGrid1.Cells[i,j]:=IntToStr(Sung[i,j]);
end;

{개별 학생의 성적 합계를 구한다.}
procedure TForm1.Hap1Click(Sender: TObject);
var
  i,j,sum:integer;
begin
for i:=1 to 3 do
  begin
  sum:=0;
  for j:=1 to 3 do
    sum:=sum+StrToInt(StringGrid1.Cells[i,j]);
  StringGrid1.Cells[i,4]:=IntToStr(sum);
  end;
end;

{개별 학생의 성적 평균을 구한다.}
procedure TForm1.ave1Click(Sender: TObject);
var
  i:integer;
begin
Hap1Click(Sender);
for i:=1 to 3 do
  StringGrid1.Cells[i,5]:=
    IntToStr(StrToInt(StringGrid1.Cells[i,4]) div 3);
```

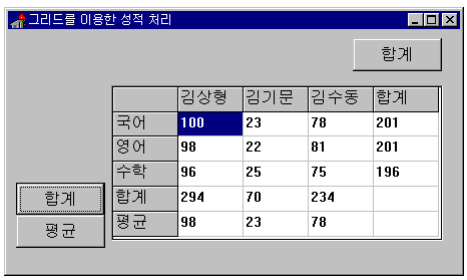
```
end;

{과목 합계를 구한다.}
procedure TForm1.Hap2Click(Sender: TObject);
var
  i,j,sum:integer;
begin
for i:=1 to 3 do
  begin
  sum:=0;
  for j:=1 to 3 do
    sum:=sum+StrToInt(StringGrid1.Cells[j,i]);
  StringGrid1.Cells[4,i]:=IntToStr(sum);
  end;
end;

end.
```

디자인시에 그리드에 문자열을 미리 입력해 둘 수 없으므로 이 예제에서는 타입 상수로 성적을 미리 정의해 두고 FormCreate 이벤트에서 문자열을 그리드로 대입한다. 물론 어디까지나 예제이므로 이런 방법을 쓸 뿐이며 좀 더 제대로 만든다면 파일에서 읽어오거나 데이터 베이스에서 읽어오는 것이 더 좋은 방법이다.

그리드에 성적이 입력되면 합계나 평균 버튼을 사용하여 과목별 합계, 학생별 합계 및 평균을 구할 수 있다. 단 문자열 그리드에 입력되는 모든 데이터는 예외없이 문자열이므로 성적 처리와 같이 수치를 취급하는 경우는 수치와 문자열 변환함수를 적절히 사용해야 한다. 게다가 goEditing 속성을 True로 설정하였으므로 실행중에 성적을 바꾸거나 직접 입력해 넣을 수도 있다. 실행중의 모습은 다음과 같다.

**그림**

문자열 그리드




예상하겠지만 문자열 그리드는 동작이나 모양을 정의하는 속성을 굉장히 많

이 가지고 있다. 본문에서 일일이 속성을 설명할 수는 없으므로 레퍼런스를 참조하기 바란다.

## 나. 드로우 그리드

드로우 그리드는 문자열 그리드와 개념적으로 거의 동일하며 사용하는 속성이나 겉모양도 비슷하다. 다만 차이점이라면 문자열 그리드가 문자열을 담는것에 비해 드로우 그리드는 그림을 담는다는 차이점뿐이다. 드로우 그리드에 관한 나머지 사항은 역시 레퍼런스를 참조하기 바란다.

# 14-5 멀티미디어

윈도우즈는 멀티미디어 시스템이다. 간단한 사운드에서부터 동영상, CD 연주까지 다양한 매체의 재생을 지원한다. 요즘 웬만한 시스템에는 사운드 카드가 설치되어 있으며 소리가 나지 않는 프로그램은 무척 따분하고 적막한 느낌을 준다. 델파이는 다양한 방법으로 멀티미디어를 구현하며 윈도우즈에서 가능한 일은 델파이로도 모두 가능하다.

## 가. 비프음 내기

아주 간단한 사운드를 비프(Beep)음이라고 하며 사용자의 실수, 중요한 대화상자를 출력할 때 주의를 끌기 위해서 주로 사용하거나 게임에서 총알 발사음, 점프음 등의 효과음을 낼 때도 사용한다. 이런 간단한 비프음은 델파이에서 MessageBeep 함수 호출 하나로 간단하게 구현할 수 있다.


14jang
beep1

MessageBeep 함수는 몇 가지 미리 정의되어 있는 비프음 중 하나를 선택해서 연주할 수 있으며 연주할 음은 인수로 전달된다. 실습을 위해 버튼을 하나 배치하고 이 버튼의 OnClick 이벤트에 다음 코드를 추가해 보자.

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

프로그램을 실행시킨 후 버튼을 누르면 비프음이 출력될 것이다. 어떤 음이 출력될 것인가는 어떤 사운드 카드가 설치되어 있는가와 윈도우즈의 사운드 세팅이 어떻게 되어 있는가에 따라 다르다. MessageBeep 함수가 낼 수 있는 음은 지정해 주는 인수에 따라 다음과 같은 종류가 있다.

**표** MessageBeep 함수가 낼 수 있는 소리의 종류

| 인수 | 의미 |
|---|---|
| -1 | PC 스피커에서 나는 기본음 |
| MB_ICONASTERISK | SystemAsterisk |

| | |
|---|---|
| MB_ICONEXCLAMATION | SystemExclamation |
| MB_ICONHAND | SystemHand |
| MB_QUESTION | SystemQeustion |
| MB_OK | SystemDefault |

인수가 -1일 경우는 사운드 카드를 사용하지 않으며 컴퓨터에 달린 스피커를 통해 "삑"하는 음을 발생시킨다. 단 델파이에서는 -1을 $ffff로 표현하므로 이 음을 내고자 할 경우는 MessageBeep($ffff)로 호출해 주어야 한다. 나머지 다섯 개의 음은 윈도우즈의 제어판에서 설정한 사운드 세팅 상태에 따라 실제로 연주될 음이 달라진다. 특별히 고급한 사운드를 쓰지 않고 간단한 사운드를 구현하고자 할 때 적절한 함수이다.

# 나. 웨이브 파일 연주

비프음에 비해 윈도우즈의 표준 사운드 파일인 웨이브 파일(*.WAV)은 훨씬 더 다양한 소리를 출력한다. 웨이브 파일을 연주할 때는 다음 함수를 사용한다.

```
sndPlaySound(Sound, Flag);
```

단 이 함수는 델파이가 제공하는 기본 함수가 아니므로 사용하려면 반드시 uses문에 MMsystem 유닛의 이름을 직접 기입해 주어야 한다. 첫 번째 인수인 Sound에는 연주할 사운드의 이름을 적어주되 WAV 파일 이름이나 윈도우즈에 정의된 비프음을 지정해 줄 수 있지만 비프음은 굳이 이 함수를 쓰지 않아도 연주할 수 있으므로 대개의 경우는 WAV 파일의 이름을 사용한다. 두 번째 인수인 Flag는 연주의 방식을 지정해 주며 다음과 같은 값들이 사용된다.

| 값 | 의미 |
|---|---|
| SND_SYNC | 사운드 연주를 동기화하며 연주가 끝나기 전에는 함수를 끝내지 않는다. 즉 사운드가 연주되는 동안 잠시 멈추어 있는다. |
| SND_ASYNC | 사운드 연주를 비동기화하며 연주가 시작되자마자 함수를 끝내므로 사운드 연주 후 곧바로 다른 일을 할 수 있다. 물론 사운드는 계속 연주된다. |

| | |
|---|---|
| SND_NODEFAULT | sndPlaySound 함수는 사운드 파일이 발견되지 않으면 윈도우즈에 정의된 디폴트 비프음을 내지만 이 플래그가 지정되어 있으면 디폴트 음도 연주하지 않으며 함수 호출을 무시해 버린다. |
| SND_LOOP | 지정한 사운드를 계속 반복적으로 연주한다. 연주를 끝내려면 sndPlaySound(Nil, 0)를 호출해 준다. |

특별한 경우가 아니라면 사운드를 연주한 후 곧바로 다른 일을 할 수 있는 SND_ASYNC 플래그를 사용하는 것이 제일 무난하다.

연주할 사운드 파일은 가급적이면 프로그램 파일과 같은 디렉토리에 있는 것이 좋다. 만약 현재 디렉토리에 사운드 파일이 없을 경우는 윈도우즈 디렉토리, 시스템 디렉토리를 검색해 보고 PATH가 지정된 모든 디렉토리를 검색하여 사운드 파일을 찾는다. 만약 그래도 사운드 파일이 발견되지 않으면 윈도우즈에 정의된 디폴트 비프음을 연주하며 에러는 발생하지 않는다. 설사 사운드 파일이 없거나 사운드 카드가 없는 시스템이라 하더라도 소리가 발생하지 않을 뿐 프로그램의 동작에는 영향을 주지 않으므로 별도의 에러 처리를 할 필요는 없다.

14jang
sound1

다음 예제는 사운드 파일 연주에 관한 몇 가지 기법을 보여 준다. 연주 시작을 명령하는 버튼 하나와 연주할 사운드 파일을 지정하는 라디오 버튼 그룹이 있으며 이 예제와 같은 디렉토리에는 4개의 WAV 파일이 있다.


라디오 그룹에서 연주할 사운드를 선택한 후 Play 버튼을 누르면 해당 사운드가 연주될 것이다. 물론 사운드 카드가 설치되어 있어야 하고 윈도우즈에는 사운드 드라이버가 설치되어 있어야 제대로 동작한다. 전체 소스는 다음과 같다. uses문에 MMsystem 유닛이 기입되어 있음을 주목하자.

```
unit Sound1_f;

interface
```

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    RadioGroup1: TRadioGroup;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0:sndPlaySound('wav1.wav',snd_sync);
    1:sndPlaySound('wav2.wav',snd_async or snd_loop);
    2:sndPlaySound('wav3.wav',snd_async);
    3:sndPlaySound('wav4.wav',snd_sync);
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  RadioGroup1.ItemIndex:=0;
end;

end.
```

버튼의 OnClick 이벤트에서 사운드를 연주하되 Flag의 사용 예를 보이기 위해 각각 다른 방식으로 사운드를 연주한다. 프로그램을 직접 실행해 보면서 Flag 인수에 따라 연주 방식이 어떻게 달라지는가를 확인해 보기 바란다. 우선

SND_ASYNC 플래그를 사용하는 wav3.wav와 SND_SYNC를 사용하는 wav4.wav의 차이점을 확인해 보면 wav3.wav를 연주할 때는 라디오 버튼을 선택하거나 윈도우를 옮기는 등의 동작을 할 수 있지만 wav4.wav를 연주할 때는 그렇지 못하다. SND_ASYNC 플래그를 사용하면 연주를 시작하자마자 프로그램으로 다시 제어권이 돌아오지만 SND_SYNC 플래그를 사용하면 사운드를 연주하는 동안 윈도우즈 전체가 실행을 멈추고 대기한다. 간단한 효과음을 연주한다면 SND_ASYNC 플래그를 사용하는 것이 적절하다.

SND_LOOP 플래그를 사용하는 wav2.wav는 한 번 연주를 시작하면 계속 반복적으로 연주를 하며 다른 사운드를 연주하기 전에는 연주를 멈추지 않는다. 단 이 플래그는 반드시 SND_ASYNC 플래그와 함께 사용해야 하며 그렇지 않으면 연주가 되지 않는다.

# 다. MediaPlayer 컴포넌트

함수를 사용해서 간단한 사운드를 출력하는 것과는 별도로 델파이는 총체적인 멀티미디어 기능을 구현하는 MediaPlayer라는 컴포넌트를 제공하며 System 팔레트의 7번째에 위치하고 있다. 이 컴포넌트를 폼에 배치하면 여러 가지 버튼이 포함되어 있는 패널이 나타난다.

그림

MediaPlayer 컴포넌트

이 버튼들은 개별적으로 동작하는 버튼이 아니라 MediaPlayer 컴포넌트에 소속된 버튼들이며 사용자가 프로그램 실행중에 직접 사용할 수 있도록 기능이 미리 정의되어 있다.

표

MediaPlayer 컴포넌트의 각 버튼 기능

| 버튼 | 기능 |
|---|---|
| Play | 연주를 시작한다. |
| Pause | 잠시 멈춘다거나 연주를 계속한다. |
| Stop | 연주나 녹음을 멈춘다. |
| Next | 다음 트랙으로 이동한다. 트랙이 없는 장비일 경우는 끝으로 이동한다. |
| Prev | 앞 트랙으로 이동한다. 트랙이 없는 장비일 경우는 처음으로 이동한다. |

| | |
|---|---|
| Step | 앞쪽 프레임으로 이동한다. |
| Back | 뒤쪽 프레임으로 이동한다. |
| Record | 녹음을 시작한다. |
| Eject | 미디어를 방출한다. |

이 컴포넌트의 외형을 결정하는 속성에는 다음 세 가지가 있다.

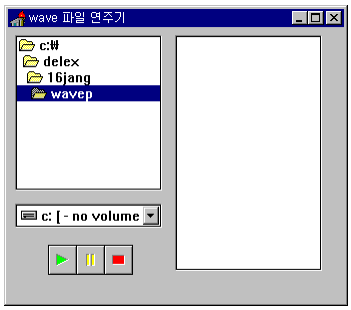
| 속성 | 의미 |
|---|---|
| VisibleButtons | 각 버튼의 출력 여부 |
| EnabledButtons | 각 버튼의 활성화 여부 |
| ColoredButtons | 각 버튼의 색상 사용 여부 |

세 속성 모두 MediaPlayer 컴포넌트에 속한 버튼들을 세부 속성으로 가지며 개별 버튼의 속성을 설정한다. 9개의 버튼 중 불필요한 버튼이 있다면 VisibleButtons 속성을 조정하여 숨길 수 있으며 실행중에 사용 불가능한 버튼은 EnabledButtons 속성으로 사용을 금지시킬 수도 있다.

MediaPlayer 컴포넌트는 멀티미디어에 관한 모든 기능을 제공하며 주로 사운드나 동영상, 오디오 CD의 연주에 사용된다. 어떤 매체를 연주(또는 재생)할 것인가는 DeviceType 속성으로 지정하며 이 속성이 dtAutoSelect일 경우는 매체 파일의 확장자에 따라 연주 형태를 결정한다. 즉 확장자가 WAV이면 사운드를 연주하고 AVI이면 동영상을 재생한다.


14jang
wavep

MediaPlayer 컴포넌트를 사용하여 WAV 파일을 연주하는 예제를 만들어 보자. WAV 파일을 선택하기 위해 드라이브 콤보, 디렉토리 리스트, 파일 리스트를 배치하고 각 컴포넌트를 연결한 후 파일 리스트 박스의 Mask 속성을 *.WAV로 설정한다. 그리고 MediaPlayer 컴포넌트를 배치하고 앞쪽 세 개의 버튼만 남겨둔 후 나머지 버튼은 모두 숨긴다.




작성해야 할 코드는 다음 한 가지 뿐이다. 파일 리스트 박스에서 WAV 파일을 더블클릭하면 WAV 파일을 연주하는 코드이다.

```
procedure TForm1.FileListBox1DblClick(Sender: TObject);
begin
  MediaPlayer1.FileName:=FileListBox1.FileName;
  MediaPlayer1.Open;
  MediaPlayer1.Play;
end;
```

파일 리스트 박스에서 선택된 파일 이름을 MediaPlayer 컴포넌트의 FileName 속성에 대입해 준다. 파일의 확장자가 WAV이므로 사운드 파일을 연주할 준비를 하게 된다. Open 메소드는 지정된 멀티미디어 장치를 연다. 여기서 장치(Device)라는 표현은 물리적인 장치일 수도 있지만 디스크 상의 파일도 가능하며 파일일 경우는 읽어와서 연주를 대비하게 된다. Play 메소드는 장치(파일)를 실제로 연주하게 되며 이 예제에서는 장치가 사운드 파일이므로 사운드를 연주할 것이다. 사용자가 프로그램 실행중에 Play 버튼을 눌러도 Play 메소드를 호출한 것과 동일한 동작을 하게 된다.

# 라. 동영상 재생


14jang
avi

윈도우즈의 표준 동영상 파일인 AVI 파일을 재생하는 일은 너무너무 쉽다. MediaPlayer의 FileName 속성에 AVI 파일을 설정해 주고 Open, Play 메소드를 호출해 주기만 하면 된다. 아니면 AutoOpen 속성을 True로 설정해 놓고 사용자가 Play 버튼을 눌러 동영상을 재생하게 할 수도 있다. 그러면 별도의 윈도우를 열어 AVI 파일을 재생해 줄 것이다. 폼 내에서 AVI 파일을 연주하고 싶다면 패널을 하나 배치한 후 MediaPlayer의 Display 속성에 패널의 이름을 지정해 준다. 그러면 별도의 윈도우를 열지 않고 패널에서 AVI 파일을 재생한다. 실행중

의 모습은 다음과 같다.

그림
폼에서 동영상을 재생하는 모양


이 정도 프로그램을 만드는 데 단 한줄의 코드도 작성하지 않아도 된다니 참 놀라운 일이다. 동영상을 재생하는 코드는 물론이고 잠시 정지, 한 프레임씩 재생, 앞 뒤로 이동하는 코드가 모두 MediaPlayer 컴포넌트에 이미 작성되어 있기 때문이다.

# 마. CD 플레이어

14jang
cdplay

CD 플레이어를 만드는 일도 동영상 재생기를 만드는 일만큼이나 간단하다. MediaPlayer를 폼에 배치하고 앞쪽 다섯 개의 버튼만 남겨두고 모두 제거한다. 오디오 CD를 연주하기 위해 DeviceType 속성을 dtCDAudio로 설정하고 프로그램 시작 직후부터 연주를 할 수 있도록 AutoOpen 속성을 True로 설정해 주기만 하면 된다. 실행중의 모습은 MediaPlayer만 있는 극단적으로 간단한


이런 모양이다.

MediaPlayer의 버튼을 이용하여 연주 시작, 중지, 앞뒤 트랙으로 이동을 할 수 있다. 코드는 작성할 필요가 거의 없으며 FormClose 이벤트 핸들러에서 Stop 메소드를 호출하여 프로그램이 끝나면 연주를 끝낼 수 있도록 만드는 정도만 해 주어도 된다. 이렇게 만들면 무척 간단하기는 하지만 여기에 현재 트랙이나 현재 위치를 나타내고 변경할 수 있는 기능까지 추가하려면 조금 복잡해진다.

MediaPlayer에서 현재 연주 위치를 나타내는 방법에는 여러 가지 형태가 있다. 연주할 수 있는 매체가 다양하므로 당연히 위치 표현 방법도 매체별로 여러 가지가 있을 수밖에 없다. 위치 표현 방법은 TimeFormat 속성으로 지정하며

TimeFormat 속성에 따라 StartPos, EndPos, Length, Position 등 위치에 관계된 속성의 속성값 표현 방식도 달라진다.

| 속성값 | 의미 |
|---|---|
| tfMilliseconds | 1000분의 1초를 나타내며 4바이트 길이를 가진다. |
| tfHMS | 시,분, 초 형태로 나타낸다. |
| tfMSF | 분, 초, 프레임 형태로 나타낸다. |
| tfFrames | 프레임으로만 나타낸다. |
| tfBytes | 바이트 수로 나타낸다. |
| tfTMSF | 트랙, 분, 초, 프레임 형태로 나타낸다. |

오디오 CD의 경우는 트랙과 분, 초 정도의 정보가 필요하므로 TimeFormat 속성을 tfTMSF로 설정하며 프레임 정보는 사용하지 않는다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
MediaPlayer1.TimeFormat:=tfTMSF;
end;
```

CD의 현재 위치는 연주중에 끊임없이 변하므로 타이머를 설치하여 OnTimer 이벤트 핸들러에서 출력해 주는 것이 적당하다. Timer 컴포넌트와 시간을 출력할 labtime 레이블, 트랙을 출력할 labtrack 레이블을 배치하고 OnTimer 이벤트 핸들러에 다음 코드를 작성한다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
 NowPos:LongInt;
begin
NowPos:=MediaPlayer1.Position;
labtime.caption:='Time:'+IntToStr(mci_TMSF_Minute(NowPos))
 +':'+IntToStr(mci_TMSF_Second(NowPos));
labtrack.caption:='Track:'+IntToStr(mci_TMSF_Track(NowPos));
end;
```

tfTMSF 형태의 정보가 4바이트의 크기를 가지므로 정보를 담을 변수 NowPos를 LongInt형으로 선언해 둔다. 현재 위치를 얻을 때는 Position 속성값을 읽으면 되며 이 속성값은 다음과 같은 구조로 되어 있다.

| 프레임 | 초 | 분 | 트랙 |
|---|---|---|---|

※각 격자는 1바이트

NowPos에서 각 바이트를 분리하려면 mci_TMSF_* 매크로 함수를 사용하며 이 매크로 함수들은 mmsystem 유닛에 정의되어 있으므로 uses절에 mmsystem을 반드시 포함시켜 주어야 한다. 조금 더 복잡해진 모습은 다음과 같다.

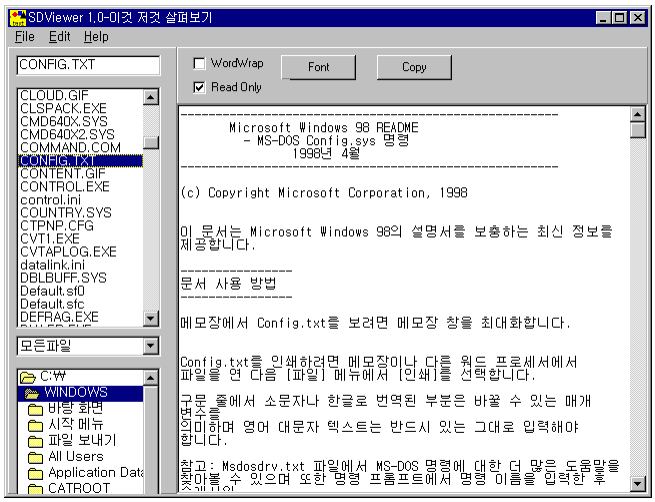

여기에 기능을 더 추가한다면 CD 타이틀의 목록을 작성한다거나 각 제목의 곡명을 입력받아 두었다가 트랙이 바뀔 때마다 곡명을 출력해 주는 기능을 생각할 수 있다. 참고로 델파이 4 초기 버전의 경우 CD 연주에 버그가 있어 이 예제가 제대로 만들어지지 않을 수도 있다.

# 바. SDViewer


14jang
SDViewer

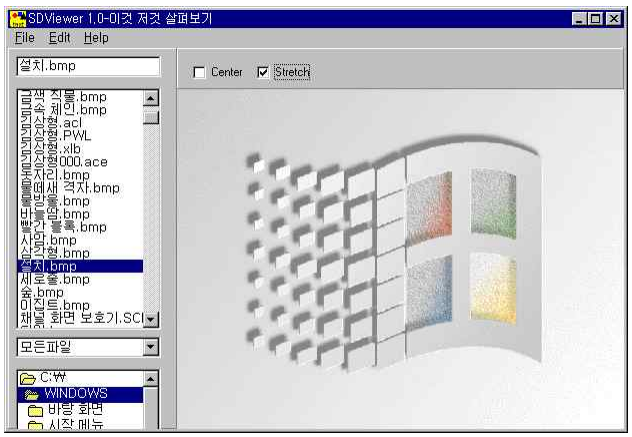
이 예제는 메모, 이미지, Media Player 컴포넌트 세 개를 이용하여 하드 디스크의 여기 저기에 흩어져 있는 텍스트 파일, 그림 파일, 동영상, 웨이브 파일의 내용을 신속하게 살펴 보기 위해 만든 것이다.

그림
텍스트 살펴보기




왼쪽의 디렉토리 리스트 박스와 파일 리스트 박스를 사용하여 보고자 하는 파일을 선택하면 오른쪽에 내용이 나타난다. 왼쪽 영역을 제외한 나머지 영역은 노트북 컴포넌트로 가득 채워져 있으며 노트북은 TEXT, BMP, AVI 세 개의 페이지로 이루어져 있다. 파일 리스트 박스에서 선택된 파일의 확장자를 읽어 어떤 종류의 파일인지를 판별하며 파일의 종류에 따라 노트북의 페이지를 교체한다. 첫 번째 페이지는 위 그림에서 보인 것 처럼 메모 컴포넌트가 배치되어 있고 두 번째 페이지는 그림 파일을 보여 주기 위해 이미지가 배치되어 있다.

그림
비트맵 살펴보기




세 번째 페이지에는 Media Player 컴포넌트만 덩그렇게 배치되어 있으며 WAV, AVI 등의 파일을 연주 또는 재생한다. 특별히 어려운 코드는 없으므로 소스를 보고 직접 분석해 보기 바란다.

```
unit SDViewer_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, FileCtrl, Menus, ComCtrls, MPlayer, Registry, Jpeg;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    FileListBox1: TFileListBox;
    DirectoryListBox1: TDirectoryListBox;
    DriveComboBox1: TDriveComboBox;
    Notebook1: TNotebook;
    MainMenu1: TMainMenu;
    About1: TMenuItem;
    FilterComboBox1: TFilterComboBox;
    Edit1: TEdit;
    RichEdit1: TRichEdit;
    Help1: TMenuItem;
    Edit2: TMenuItem;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Cut1: TMenuItem;
    Copy1: TMenuItem;
    Paste1: TMenuItem;
    Panel2: TPanel;
    CheckBox1: TCheckBox;
    Button1: TButton;
    FontDialog1: TFontDialog;
    Panel3: TPanel;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    MediaPlayer1: TMediaPlayer;
    Panel4: TPanel;
    Panel5: TPanel;
    MediaPlayer2: TMediaPlayer;
    Usage1: TMenuItem;
    Delete1: TMenuItem;
    CheckBox4: TCheckBox;
    Button2: TButton;
    SaveAs1: TMenuItem;
    SaveDialog1: TSaveDialog;
    Image1: TImage;
    procedure About1Click(Sender: TObject);
```

```
    procedure Edit1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure CheckBox1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure CheckBox2Click(Sender: TObject);
    procedure CheckBox3Click(Sender: TObject);
    procedure Usage1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Cut1Click(Sender: TObject);
    procedure Copy1Click(Sender: TObject);
    procedure Paste1Click(Sender: TObject);
    procedure Delete1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure CheckBox4Click(Sender: TObject);
    procedure SaveAs1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FileListBox1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}


{파일 리스트 박스에서 선택이 변경될 때 파일의 확장자에
따라 적절한 뷰어를 호출한다.}
procedure TForm1.FileListBox1Click(Sender: TObject);
var
  str:String;
begin
  {아무것도 선택되어 있지 않으면 리턴한다}
  if FileListBox1.ItemIndex=-1 then Exit;
  str:=ExtractFileExt(FileListBox1.Filename);
  {BMP 파일}
  if (CompareText(str,'.bmp')=0) or (CompareText(str,'.jpg')=0) then
    begin
    notebook1.ActivePage:='bmp';
    Image1.Picture.LoadFromFile(FilelistBox1.FileName);
    end
```

```
  else
  {WAV 파일}
    if (CompareText(str,'.wav')=0) then
    begin
    notebook1.ActivePage:='wav';
    MediaPlayer1.FileName:=FileListBox1.FileName;
    MediaPlayer1.Open;
    MediaPlayer1.Play;
    end
  else
  {동영상 파일}
    if (CompareText(str,'.avi')=0) then
    begin
    notebook1.ActivePage:='avi';
    MediaPlayer2.FileName:=FileListBox1.FileName;
    MediaPlayer2.Open;
    MediaPlayer2.Play;
  end
  else
  {그외의 파일은 모두 텍스트 파일로 간주한다}
  {단 파일이 없을 경우 silent 예외 처리한다.}
    begin
    try
    notebook1.ActivePage:='Text';
    RichEdit1.Lines.LoadFromFile(FilelistBox1.Filename);
    except
  end;
end;
end;

procedure TForm1.About1Click(Sender: TObject);
begin
  ShowMessage('택스트 파일 살펴보기 프로그램이다');
end;

{에디트에 보고자 하는 파일의 마스크를 입력한다}
procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key=13 then
    FileListBox1.mask:=Edit1.Text;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.State=cbChecked then
```

```
   RichEdit1.WordWrap:=True
  else
   RichEdit1.WordWrap:=False;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
   RichEdit1.Font:=FontDialog1.Font;
end;

procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  if CheckBox2.Checked=True then
   Image1.Center:=True
  else
   Image1.Center:=False;
end;

procedure TForm1.CheckBox3Click(Sender: TObject);
begin
  if CheckBox3.Checked=True then
   Image1.Stretch:=True
  else
   Image1.Stretch:=False;
end;

procedure TForm1.Usage1Click(Sender: TObject);
begin
  notebook1.ActivePage:='Text';
  RichEdit1.Lines.Clear;
  RichEdit1.Lines[0]:='그냥 사용하면 되는데요';
end;

procedure TForm1.Cut1Click(Sender: TObject);
begin
  RichEdit1.CutToClipboard;
end;

procedure TForm1.Copy1Click(Sender: TObject);
begin
  RichEdit1.CopyToClipboard;
end;

procedure TForm1.Paste1Click(Sender: TObject);
begin
```

```
  RichEdit1.PasteFromClipboard;
end;

procedure TForm1.Delete1Click(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.CheckBox4Click(Sender: TObject);
begin
  if CheckBox4.State=cbChecked then
    begin
    RichEdit1.ReadOnly:=True;
    end
  else
    begin
    RichEdit1.ReadOnly:=False;
    end;
end;

procedure TForm1.SaveAs1Click(Sender: TObject);
begin
  if RichEdit1.ReadOnly=True then
    begin
    ShowMessage('현재 읽기전용 상태이므로 저장할 수 없습니다.')
    end
  else
    begin
    SaveDialog1.Filename:=FileListBox1.FileName;
    if SaveDialog1.Execute then
      RichEdit1.Lines.SaveToFile(SaveDialog1.Filename);
    end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  Pst:array [0..128] of char;
  SDReg:TRegIniFile;
begin
  SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDViewer');
  SDReg.WriteInteger('Pos','Left',Form1.Left);
```

```
  SDReg.WriteInteger('Pos','Top',Form1.Top);
  SDReg.WriteInteger('Size','Width',Form1.Width);
  SDReg.WriteInteger('Size','Height',Form1.Height);
  SDReg.WriteString('Dir','NowDir',DirectoryListBox1.Directory);
  SDReg.Free;
end;

procedure TForm1.FormShow(Sender: TObject);
var
  SDReg:TRegIniFile;
begin
  SDReg:=TRegIniFile.Create('Software₩SangHyungSoft₩SDViewer');
  notebook1.ActivePage:='Text';
  Form1.Left:=SDReg.ReadInteger('Pos','Left',100);
  Form1.Top :=SDReg.ReadInteger('Pos','Top',100);
  Form1.Width:=SDReg.ReadInteger('Size','Width',500);
  Form1.Height:=SDReg.ReadInteger('Size','Height',400);

  DirectoryListBox1.Directory:=
    SDReg.ReadString('Dir','NowDir','c:₩');
  SDReg.Free;
end;

end.
```

# 14-6 제3자 컴포넌트

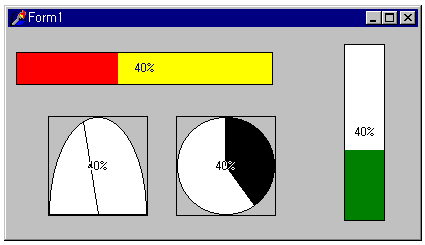
델파이에 사용할 수 있는 컴포넌트의 수에는 제한이 없다. 델파이가 기본적으로 제공하는 컴포넌트 외에도 제3자(third party)들이 만든 많은 컴포넌트를 사용할 수 있다. 여기서 제3자라는 말은 델파이를 만든 볼랜드 이외의 사람들을 말하며 필자나 여러분들도 이에 속한다. 제3자 컴포넌트는 인터넷이나 대중 통신망을 통해 쉽게 구할 수 있으며 자신이 직접 만들 수도 있다. 여기서는 델파이가 기본적으로 제공하는 제3자 컴포넌트를 소개한다. 이 컴포넌트들이 델파이에 포함된 이유는 사용 그 자체보다는 이런 식으로 컴포넌트를 덧붙여 사용할 수 있음을 보여주기 위해서이다. 그래서 이 컴포넌트들은 별도의 도움말이 없으며, 표준 컴포넌트와는 색다른 특징을 가지기도 한다.

도움말이 없어도 오브젝트 인스펙터를 사용하면 직관적으로 그 기능을 알 수 있도록 되어 있으므로 지나치게 상세한 설명은 하지 않기로 한다. 여기서 제공하는 예제는 컴포넌트의 동작을 보이기 위해 제작된 것이므로 제작 과정 또한 상세하게 밝히지 않는다. 예제를 직접 실행해 보면 컴포넌트의 동작을 잘 알 수 있을 것이다.

## 가. 게이지

게이지란 현재 작업의 진행 상태를 나타내며 주로 설치 프로그램에서 설치 작업의 상황을 보여주기 위해 많이 사용한다. 델파이가 제공하는 게이지 컴포넌트는 Samples 페이지의 첫 번째에 있다. 윈 95 컨트롤 중에도 이와 동일한 목적을 가지는 프로그래스 바라는 컨트롤이 있는데 델파이가 제공하는 게이지 컴포넌트는 프로그래스 바에 비해 다양한 속성을 제공한다는 특징이 있다. ForeColor, BackColor 속성으로 전경색, 배경색을 선택할 수 있으며 Kind 속성으로 게이지 모양을 수평, 수직, 파이, 계기판 형태 중 하나를 선택할 수도 있다.

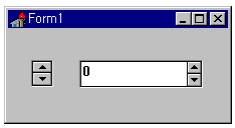
**그림**
여러 가지 모양의 게이지



델파이가 기본 제공하는 게이지 외에도 그래픽을 표현할 수 있는 제3자 게이지들이 많이 발표되어 있다. 심지어 필자조차도 이런 게이지를 만들어 본 적이 있으며 16장에서 컴포넌트 제작 과정에서 볼 수 있다.

# 나. 스핀 버튼

스핀 버튼은 값의 증감을 마우스로 조절할 수 있는 아래, 위 두 개의 작은 버튼을 가진 컴포넌트이며 값을 세밀하게 조절할 필요가 있을 때 사용한다. 스핀 버튼만 가진 형태와 에디트 버튼과 스핀 버튼을 결합한 형태가 있다.

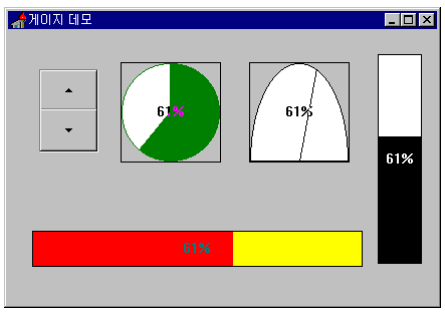



스핀 에디트 버튼은 키보드로 값을 직접 입력할 수도 있고 마우스로 값을 조정할 수 있다는 점에 있어서 다양한 입력 방법을 제공해 주며 스핀 버튼은 버튼에 비트맵을 삽입할 수 있어 장식 효과가 뛰어나다. 값의 범위를 지정하는 MinValue, MaxValue 등의 속성을 가지며 OnDownClick, OnUpClick, OnClick 등의 이벤트를 가진다.


14jang gauge 그림

게이지 컴포넌트 사용예

배포 CD에 제공되는 Gauge1.dpr은 스핀 버튼과 게이지의 동작을 한꺼번에 나타낼 수 있는 예제이다. 이 예제를 실행해 보고 소스를 살펴보면 스핀 버튼과 게이지 컴포넌트를 어떻게 사용하는가를 잘 알 수 있을 것이다.




스핀 버튼 하나와 여러 가지 형태의 게이지를 배치해 두고 스핀 버튼으로 게이지의 진행 상태를 변경한다. 게이지의 모양도 속성에 따라 원, 바늘, 수평, 수직 등 여러 가지가 있음을 알 수 있다.

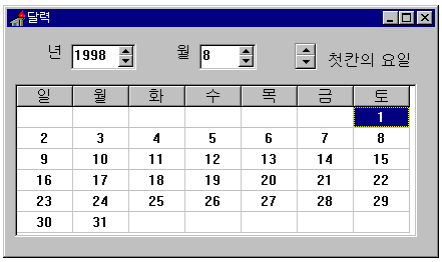
# 다. 달력

14jang
cale

달력 컴포넌트는 말 그대로 달력을 제공해 준다. Calendar 컴포넌트를 폼에 놓기만 하면 달력이 즉시 출력되며 Year, Month 속성을 사용하여 출력할 년, 월을 변경한다. 날짜수나 윤년 계산 및 요일 계산 등도 컴포넌트 내부에서 모두 자동적으로 수행해 주므로 사용자는 출력하고자 하는 년, 월만 지정해 주면 된다. 겉모양을 지정하는 몇 가지 속성과 OnClick, OnChange 등의 아주 일반적인 이벤트를 가진다. 달력을 사용하는 예제는 배포 CD의 cale.dpr이며 실행중의 모습은 다음과 같다.

**그림**

달력 및 스위치 컴포넌트 사용예



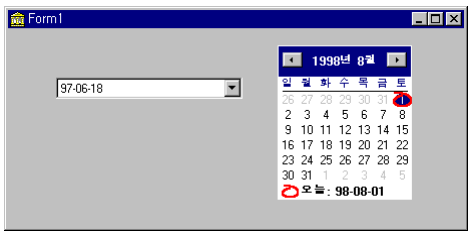

두 개의 스핀 에디트를 사용하여 년, 월을 변경한다. 날짜를 시각적으로 입력받아야 할 필요가 있을 때 아주 유용하게 사용할 만한 컴포넌트이다.

14jang
DTPick

Calendar 컴포넌트 외에 날짜와 관련된 컴포넌트가 Win32 페이지에 두 개가 더 있다. TDateTimePicker 컴포넌트는 실행중에 사용자로부터 시간이나 날짜를 입력받도록 한다. 콤보 박스처럼 생겼지만 펼칠 경우 달력 모양이 펼쳐지며 달력에서 마우스로 원하는 날짜를 선택한다. TMonthCalendar 컴포넌트는 한달치 달력을 보여주는 컴포넌트이다. 이 두 개의 컴포넌트를 폼에 배치해 보았다.

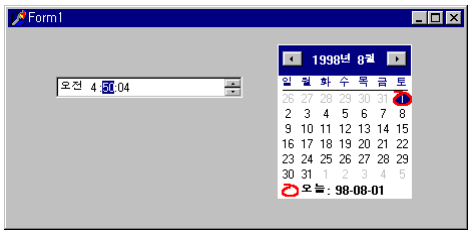
**그림**

날짜 입력 컴포넌트




DateTimePicker 컴포넌트의 콤보 박스를 열면 TMonthCalendar 컴포넌트가 아래로 펼쳐지는데 달력 상단에 있는 < > 버튼을 사용하여 앞달, 다음달로 이동하며 년을 변경하고자 할 경우 년도를 클릭한 후 스핀 버튼을 사용한다. 주요

속성으로는 Kind 속성이 있는데 이 속성이 dtkDate 이면 날짜를 입력받도록 하며 dtkTime 이면 시간을 입력받도록 한다. 시간 입력시 변경하고자 하는 요소(시, 분, 초)를 마우스로 선택한 후 스핀 버튼을 사용하면 된다.



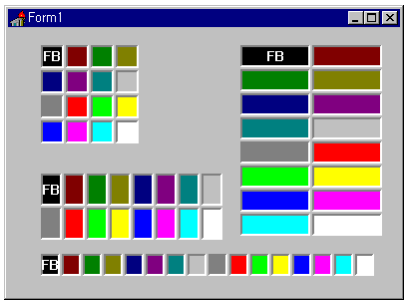

이렇게 입력되어진 날짜/시간은 각각 Time, Date 속성을 읽음으로써 알 수 있으며 날짜나 시간이 변경될 경우 OnChange 이벤트가 발생하므로 이 핸들러에서 날짜/시간 변경시의 동작을 처리하도록 한다. 다음 코드는 날짜 변경시 메시지 상자를 열어 사용자가 선택한 날짜를 보여준다.

```
procedure TForm1.DateTimePicker1Change(Sender: TObject);
var
  str:String;
begin
    ShowMessage(FormatDateTime('y"년" m"월" d"일을 선택하셨습니다."',
    DateTimePicker1.Date));
end;
```

# 라. 컬러 그리드

14jang
colgrid

컬러 그리드는 16개의 색상을 격자에 보여주며 마우스로 색상을 선택하는 컴포넌트이다. 폼에 배치하면 4*4 행렬 형태로 나타나며 크기를 변경하면 내부의 격자들도 크기가 변한다. GridOrdering 속성을 변경시키면 4*4 행렬뿐만 아니라 2*8, 8*2, 16*1의 형태로도 변경할 수 있으므로 폼의 배치 상태에 따라 적당한 모양을 선택한다.




실행중에 사용자가 색상을 선택하면 OnChange 이벤트가 발생하며 이 이벤트 코드에서 선택한 색상을 사용한다. 다음 예제는 메모 컴포넌트와 컬러 그리드를 배치하고 컬러 그리드로 메모 컴포넌트의 색상을 변경한다.


코드는 OnChange 이벤트 핸들러에 다음과 같이 기입되어 있다.

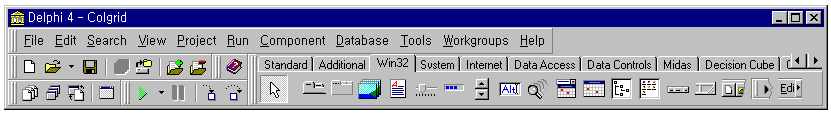
```
procedure TForm1.ColorGrid1Change(Sender: TObject);
begin
  Memo1.Color:=ColorGrid1.BackgroundColor;
  Memo1.Font.Color:=ColorGrid1.ForeGroundColor;
end;
```

컬러 그리드에서 선택된 배경 색상을 메모 컴포넌트의 색상으로 사용하며 전경 색상을 폰트의 색상으로 사용한다.

# 마. Win95 컨트롤

윈도우즈 3.1에서 지원하는 컨트롤은 고작 여섯 가지에 불과했다. 그래서 직접 컨트롤을 만들어서 사용하기도 했으며 제3자가 제공하는 컨트롤을 사서 쓰기도 했다. 그런데 이런 사정은 본격적인 32비트 운영체제인 윈95가 발표되면

서 많이 호전되었다. 윈95는 표준 컨트롤들외에 다양한 컨트롤이 추가하였는데 이 컨트롤들은 윈95부터 지원한다고 하여 윈95 컨트롤이라고 흔히 부른다. 델파이는 Win32 페이지에 윈95 컨트롤들을 모아 놓았다.



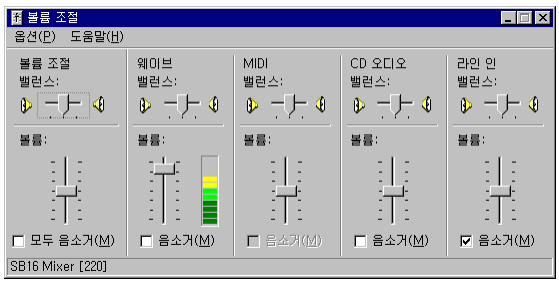

Win95형의 인터페이스를 제공한다고는 하지만 기존의 컴포넌트와 대부분 중복되는 것들이며 사용 방법도 거의 비슷하거나 조금 확장된 것뿐이다. 그래서 표준 컨트롤을 잘 사용하는 사람은 별다른 어려움없이 윈95 컨트롤을 사용할 수 있을 것이다.

## ■트랙 바

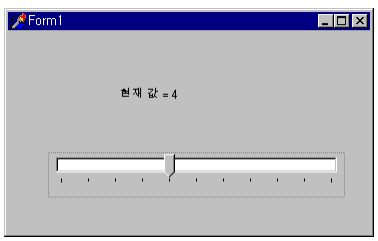
14jang TrackBar 그림

볼륨 조절 대화상자

트랙 바는 본질적으로 스크롤 바와 용도가 거의 동일하다. 연속된 범위를 갖는 값을 신속하게 조정하고자 할 때 트랙 바를 사용한다. 윈98의 볼륨 조절 대화상자를 보면 볼륨 조절용으로 트랙 바를 많이 사용하고 있다.




스크롤 바에 비해 좀 더 현대적이고 세련된 모양을 가지고 있으며 속성들이 많다. Min, Max 속성으로 범위를 지정하며 현재 위치는 Position 속성을 읽어서 구할 수 있다. 다음 예제는 트랙 바와 레이블을 하나 배치해 두고 트랙 바의 값이 변경되면 현재값을 레이블로 출력하도록 하였다.



트랙 바의 OnChange 이벤트의 코드는 다음과 같다.

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
Label1.Caption:=Format('현재 값 = %d', [TrackBar1.Position]);
end;
```

보다시피 하나도 어려울 것이 없는 코드이다. 레퍼런스를 통해 속성만 좀 연구해 보면 쉽게 친숙해질 수 있을 것이다.

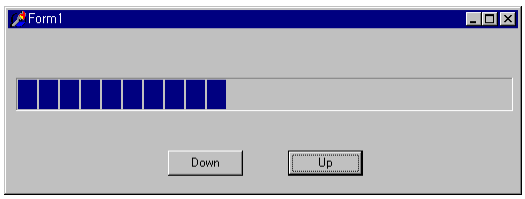
## ■프로그래스 바

14jang
Progress

프로그래스 바는 작업의 진행 상태를 막대 그래프 형식으로 보여주는 컨트롤이며 Samples 페이지의 게이지와 동일한 목적을 가지는 컨트롤이다. 설치 프로그램이나 시간이 오래 걸리는 작업을 하는 프로그램에서 이 컨트롤을 흔하게 볼 수 있다. 사용자의 입력도 받아들이지 않으며 진행 상태만 보여주기 때문에 모든 컨트롤을 통틀어 제일 프로그래밍하기 쉽다. 폼에 프로그래스 하나와 버튼 두 개를 배치하고 버튼의 OnClick 이벤트를 다음과 같이 작성해 보자.

```
procedure TForm1.BtnUpClick(Sender: TObject);
begin
 ProgressBar1.Position:=ProgressBar1.Position+5;
end;

procedure TForm1.BtnDownClick(Sender: TObject);
begin
 ProgressBar1.Position:=ProgressBar1.Position-5;
end;
```

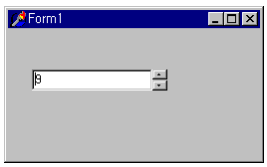
Up 버튼을 누르면 프로그래스 바의 Position 속성을 5 증가시키고 Down 버튼을 누르면 5 감소시키도록 하였다. 물론 실제 프로그래밍에서는 작업의 진행 정도를 출력해야 한다. 실행 중의 모습은 다음과 같다.



## ■업다운

14jang
UpDown

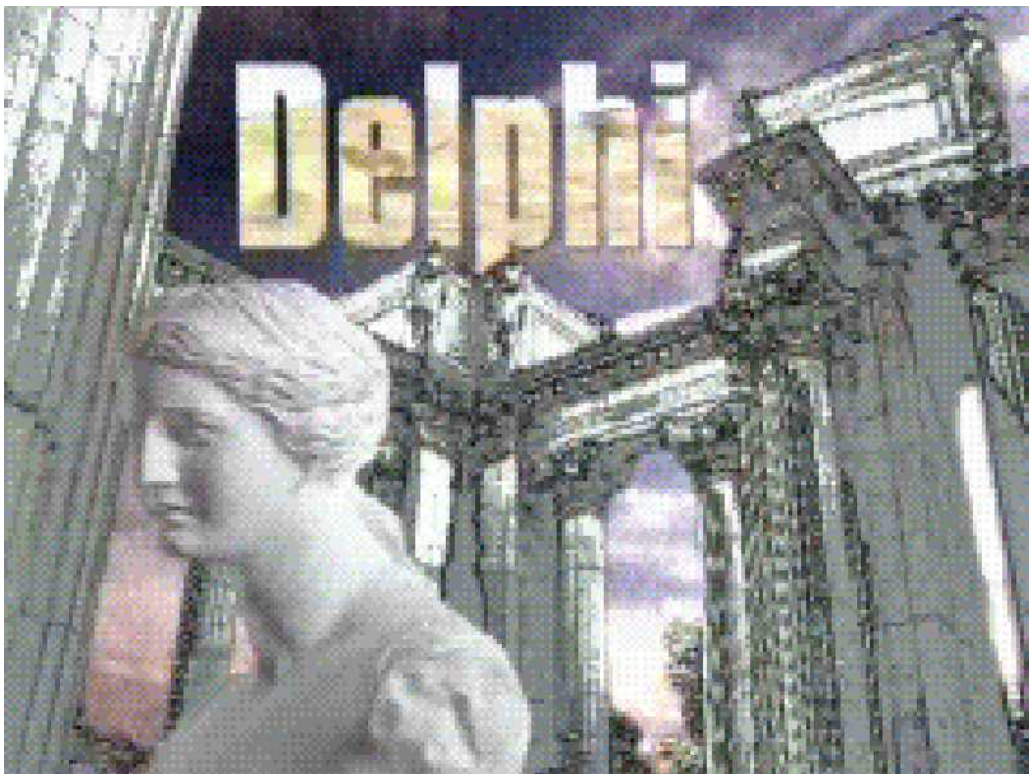
업다운 컨트롤은 주로 에디트 컨트롤과 함께 사용되며 에디트의 값을 증감시켜 주는 용도로 사용된다. Samples 페이지의 SpinButton과 동일한 컴포넌트이다. 새 프로젝트를 만들고 에디트 하나와 업다운 컨트롤을 배치해 보자. 그리고 업다운의 Associate 속성을 Edit1으로 변경하면 업다운이 에디트의 오른쪽에 밀착된다. 프로젝트를 실행시켜 보면 업다운으로 에디트의 값을 직접 변경할 수 있다. 물론 에디트에 직접 값을 입력하는 것도 가능하다.




윈도우즈의 기본 입력 장치는 키보드이지만 실제로는 키보드보다 마우스가 더 자주 사용된다. 그런데 에디트 컨트롤은 그 특성상 마우스로 입력을 할 수 없는데 숫자값을 입력받을 경우 업다운과 함께 쓰면 키보드 없이도 값을 증감시킬 수 있다. 업다운의 용도는 바로 이런 것이다. 가급적이면 키보드에 손대기 싫어하는 사용자들을 위해 마우스로 값을 입력할 수 있도록 해 준다.

# 데이터 베이스 프로그래밍

# 제 15 장



# 15-1 델파이 DB 구조

델파이는 강력한 데이터 베이스 프로그램을 빠르고 쉽게 작성하도록 해준다. 데이터 베이스 프로그램만 전문적으로 개발하는 많은 DB 전용 툴이 있지만 델파이도 그에 못지 않은 성능을 보여준다. 델파이는 일반 어플리케이션 개발 툴인 동시에 또한 데이터 베이스 개발 툴이기도 하다. 혹자는 다음과 같이 델파이를 평하기도 한다. "델파이는 일반 응용 프로그램도 만들 수 있는 데이터 베이스 개발 툴이기도 하고 데이터 베이스 개발도 할 수 있는 일반 응용 프로그램 개발 툴이기도 하다."

즉 요약하자면 델파이는 두 분야에 모두 활용할 수 있는 범용 개발 툴이라는 것이다. 그런데 갈수록 델파이는 데이터 베이스 분야에만 국한되어 활용되는 경향이 짙어지는데 그것은 델파이의 DB지원이 워낙 우수해서 그렇기도 하지만 외부적으로 데이터 베이스 시장이 꾸준히 확장되어 가고 있기 때문이기도 하다. 데이터 베이스는 경영, 회계, 관리, 홈쇼핑, 개인 정보 등에 광범위하게 사용되고 있으며 컴퓨터의 가장 주된 활동분야가 되어 있다.

델파이로 데이터 베이스 프로그램을 개발하는 것은 그렇게 어렵지 않다. 그러나 델파이 DB 프로그래밍은 쉬운 사람에게나 쉽지 아무에게나 쉬운 것은 아니다. 델파이로 데이터 베이스 프로그래밍을 공부해 보려면 최소한 다음 사항은 알고 있어야 한다.

① 델파이에 충분히 익숙해 있어야 한다. 델파이 컴파일러와 개발 환경 그리고 기본적인 파스칼 문법에 대해서 다는 몰라도 뒤적거려서라도 문제를 해결할 수 있을 정도가 되어야 한다. 델파이를 사용하는 것과 델파이로 데이터 베이스 프로그램을 짜는 것은 차원이 다르므로 델파이를 먼저 공부한 후에야 DB 프로그램을 짤 수 있다.

② 데이터 베이스에 대한 기본 개념이 있어야 한다. 데이터 베이스가 뭔지도 모르고 데이터 베이스 프로그램을 짤 수는 없는 노릇이다. 적어도 필드, 레코드, 테이블, 관계형 DB 이런 말들은 알고 있어야 하며 데이터 베이스 프로그램도 사용할 줄 알아야 한다. 도스용 DBase III+ 정도만 쓸 줄 알아도 일단 공부하는데 막히지는 않을 것이며 마이크로소프트의 액세스 정도를 써 봤다면 충분하다.

여기서는 이 두 가지에 대해 어느 정도 알고 있다고 가정하고 설명을 전개하겠다. 그럴리는 없겠지만 위의 두 가지에 대해 전혀 모르는 독자가 델파이로 DB 프로그램을 짜겠다고 한다면 걷지도 못하면서 뛸려고 하는 것과 다를 바 없다. 별로 자신이 없다면 먼저 위 두 가지를 공부하고 오기 바란다. 델파이로 DB 프로그래밍을 하는 것은 정말로 쉽다. 단 처음 시작이 조금 생소하고 어렵게 느껴질 뿐이며 조금만 익숙해지면 쾌적한 프로그래밍 환경에서 편하게 작업을 할 수 있다.

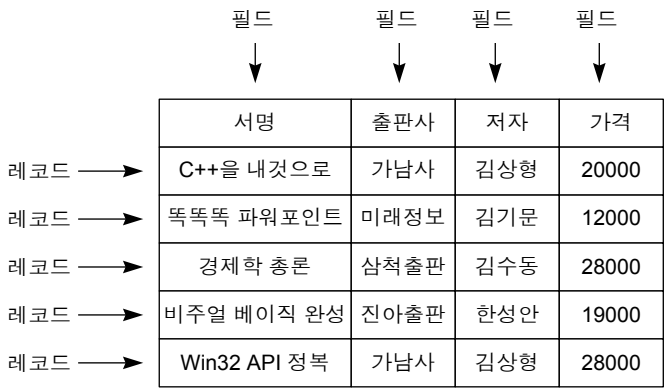
# 가. 데이터 베이스

데이터 베이스 프로그래밍을 해 보기 전에 간단하게 용어부터 알아보도록 하자. 물론 대부분 이 용어들에 대해 친숙하겠지만 그렇지 못한 사람들도 있으므로 간단하게만 정리하기로 한다.

일상생활에서나 또는 업무에서 우리는 평소에 많은 자료들을 접하게 된다. 은행이나 주식사의 예금, 고객에 관한 복잡하고 큰 자료에서부터 시작하여 일상적인 친구 주소록, 도서 목록 등도 일종의 자료들이다. 이런 자료들이 많아지다보면 관리를 해야 할 필요가 생기는데 대부분의 자료는 표 형태로 표현할 수 있다. 예를 들어 도서 목록을 정리한다면 도서 하나에 대해 서명, 출판사, 가격, 저자 등의 정보를 기록할 수 있다. 이때 각 도서의 성질을 정의하는 서명, 출판사, 가격, 저자 등을 필드(Field)라고 하며 한 도서의 정보 묶음을 레코드라고 한다.

예를 들어 컴퓨터 정복이라는 책이 있는데 가남사의 김상형이가 썼고 가격은 18000원이라는 정보가 있다면 이 정보의 묶음이 하나의 레코드가 되고 레코드는 "컴퓨터 정복", "가남사", "김상형" 등의 필드로 구성되어 있다. 이런 레코드와 필드의 집합을 테이블(Table)이라고 하며 다음과 같이 그릴 수 있다.

그림
테이블의 구조




테이블(Table)

이런 테이블이 여러 개 모였을 때 이를 데이터 베이스(Database)라고 한다. 그렇다고 아무 테이블이나 모아 둔다고 해서 제대로 된 데이터 베이스가 되는 것은 아니며 상호 관련성이 있는 테이블이 모여야 한다. 예를 들어 비디오 가게의 데이터 베이스를 생각해 보면 최소한 세 개의 테이블이 있어야 한다. 어떤 비디오 테입을 가지고 있는지 비디오 테입 목록이 있어야 하고 회원을 관리하기 위한 회원 목록이 있어야 하고 또한 누가 무슨 테입을 빌려갔는지를 기록할 대여 목록이 필요하다. 이런 식으로 하나의 목적을 위한 관련 테이블이 모였을 때 이를 데이터 베이스라고 한다.

그럼 데이터 베이스를 잘 정리해 놓으면 무슨 일을 할 수 있을까? 데이터 베이스의 기능 중 으뜸으로 중요한 기능은 역시 정보를 저장하는 기능이다. 일단 정보가 저장되어 있으면 다음과 같은 여러 가지 작업을 손쉽게 효율적으로 할 수 있게 된다.

- 검색:테이블을 뒤져 원하는 정보를 찾아낼 수 있다. 예를 들어 XXX 라는 비디오 테입을 우리 가게가 보유하고 있는지를 단번에 검색해 볼 수 있으며 ZZZ 라는 회원이 가입되어 있는지를 알 수 있다.
- 조회:테이블을 뒤져 현재 상태를 조사할 수 있다. 예를 들어 누가 무슨 테입을 빌려갔는지, 또는 반대로 XXX 비디오 테입을 빌려간 사람은 누구인지 금방 알 수 있다.
- 정렬:테이블의 레코드를 일정한 기준에 따라 정렬할 수 있다. 예를 들어 대여 실적으로 회원을 정렬해 보면 누가 매출에 가장 많은 기여를 했는지를 알 수

있으며 비디오 테입별로도 대여 빈도가 높은 테입을 금방 알 수 있다.

- 계산:테이블의 정보를 바탕으로 하여 원하는 결과를 금방 알 수 있다. 예를 들어 연체료가 10000 원 이상인 불량 사용자를 단번에 조사하여 블랙 리스트를 만들 수 있다.

이상이 데이터 베이스의 주된 기능들이지만 이 외에도 합리적 의사결정, 추론 등등 무한한 활용 가치가 있다. 만약 이런 정보들을 장부에 볼펜으로 기록해 두었다고 해 보자. 일일이 장부를 기록하기도 귀찮은 일이고 어떤 정보를 검색하려면 일일이 장부를 뒤져 봐야만 할 것이다. 또한 가장 많이 대여해간 사람 찾기, 연체료가 많은 사용자 찾기 등의 작업은 정말 끔찍할 정도로 복잡할 것이며 게다가 부정확할 것이다.

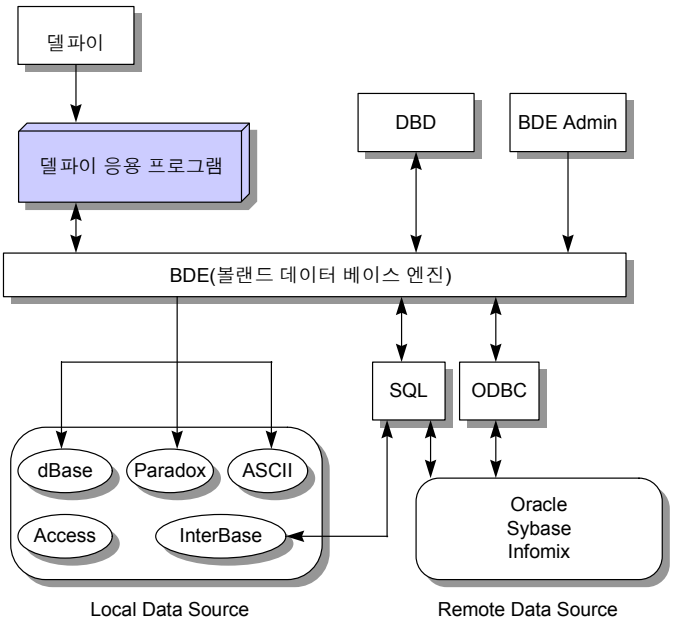
이러한 DB의 기능을 수행하는 체제를 DBMS(Database Management System)라고 하는데 현재 많은 종류의 DBMS들이 나와 있으며 또한 사용되고 있다. 각각의 DBMS들은 나름대로의 장단점을 가지고 있는데 문제는 DBMS끼리 호환이 되지 않는다는 점이다. 즉 A DBMS로 만든 테이블은 일반적으로 B DBMS에서 읽을 수 없다. 그래서 데이터 베이스 개발을 할 때는 어떤 DBMS를 사용할 것인가를 신중하게 결정해야 한다.

델파이는 다양한 종류의 DBMS를 지원하므로 현재까지 발표된 거의 대부분의 DBMS를 사용할 수 있다.

# 나. 전체적인 구조

델파이는 BDE(Borland Database Engine)라는 고성능의 데이터 베이스 엔진을 탑재하고 있으며 이 엔진과 함께 작동하는 많은 컴포넌트들을 제공한다. "엔진"이라고 하니까 괜히 공포스러워 보일지도 모르겠지만 간단하게 생각하면 델파이 프로그램과 함께 실행되는 일종의 DLL이라고 할 수 있다. 이 안에는 데이터 베이스에 관련된 많은 기능들이 미리 작성되어 있으며 그래서 프로그래머는 물리적인 데이터 베이스에 신경쓰지 않고 BDE를 통해 DB 프로그래밍을 하면 된다. BDE는 물리적인 디스크 입출력, 테이블 액세스, 정렬, 다중 사용자간의 충돌 방지, 트랜잭션 등 사용자가 일일이 해야 할 잡일을 대신해 주는 종에 불과하다. 다음 그림은 델파이의 DB 툴과 BDE 엔진, 그리고 델파이로 만든 프로그램간의 유기적인 관계를 나타낸 것이다.

그림
DB 엔진과 응용 프로그램과의 관계

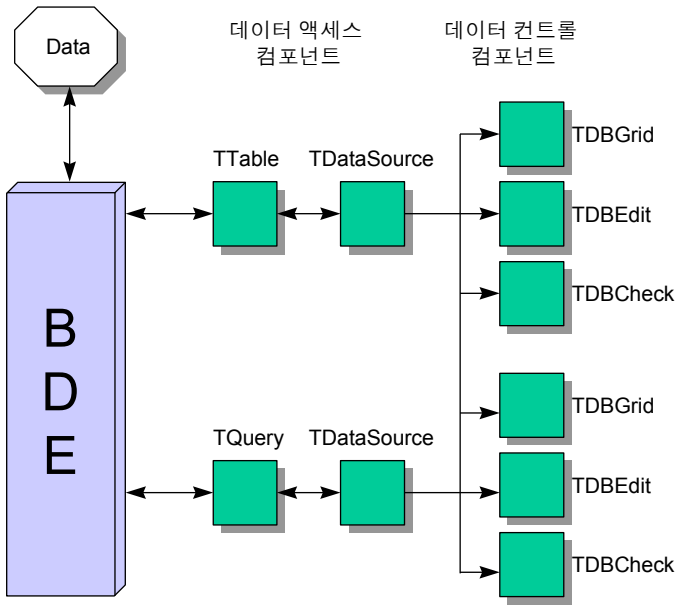



뭔가 대단히 복잡해 보이는데 벌써 이 그림을 다 이해할 필요는 없다. 실습을 하다보면 자연히 알게 될 것이고 다음에 공부한 후에 다시 보면 "그렇군!"이라는 소리가 절로 나올 것이다. 델파이가 제공하는 데이터 베이스 컴포넌트는 크게 두 가지 종류가 있으며 컴포넌트 팔레트에도 별도의 페이지에 위치하고 있다. 물론 두 부류의 컴포넌트는 확연하게 다른 특성을 가지고 있다.

- Data Access 컴포넌트 : 데이터 베이스 소스와의 연결에 관한 정보를 가진다. 즉 어떤 데이터 베이스를 액세스할 것인가를 지정한다.
- Data Control 컴포넌트 : 데이터 베이스에 저장되어 있는 정보를 폼에 출력해 주거나 반대로 입력받는 컴포넌트들이다. 에디트, 리스트 박스, 콤보 박스 등 표준 컴포넌트와 유사한 성격과 모양을 가지지만 데이터 인식(data aware) 능력을 가진다는 특징이 있다.

데이터 액세스 컴포넌트들은 실행중에는 보이지 않지만 BDE와 데이터 컨트롤 컴포넌트와의 중간에서 연결을 해주며 TDataSource 컴포넌트에 의해 데이터 액세스 컴포넌트와 데이터 컨트롤 컴포넌트가 연결된다. 다음 그림은 데이터 액세스, 데이터 컨트롤 컴포넌트와 BDE 그리고 디스크상에 존재하는 실제의 데이터 테이블이 어떻게 연결되는가를 보인 것이다.

그림
데이블, 컴포넌트, 응용 프로그램의 연결 구조




위 그림을 살펴보면 델파이로 데이터 베이스 프로그램을 만들려면 최소한 세 개의 컴포넌트가 필요하다는 것을 알 수 있다. BDE와 연결해 주는 데이터 셋(DateSet), 사용자와 연결해 주는 컨트롤 컴포넌트 그리고 이 둘을 연결해 주는 TDataSource 컴포넌트가 필요하다.

델파이가 직접 제어할 수 있는 데이터 베이스 파일은 Paradox가 만들어 내는 DB 파일과 DBase가 만들어 내는 DBF 파일, 마이크로소프트 액세스 파일등이 있다. 또한 ODBC 드라이버를 사용하면 현재 시스템에 설치된 모든 데이터 베이스에 접근할 수도 있다. Client/ Server 버전을 사용할 경우 Oracle, Sybase, SQL 서버 등의 원격 데이터 베이스 서버에서 돌아가는 프로그램을 만들 수도 있다.

# 다. 데이터 액세스 컴포넌트

Data Access 페이지에는 9개의 컴포넌트가 있으며 데이터 소스와 데이터 컨트롤 컴포넌트를 연결해 주는 역할을 한다. 물론 9개의 컴포넌트에 대해 모두 알아야 하겠지만 당장 필요한 세 가지에 대해서만 알아본 후 개념만 익히도록 하자. 당장 이해가 가지 않더라도 이어지는 실습을 통해 사용해 보면 이해가 갈 것이다.

## ■ Table

데이터를 담고 있는 테이블과 데이터 처리의 주체인 BDE와의 연결을 담당한다. 즉 BDE가 처리할 데이터 테이블이 구체적으로 어떤 종류인지, 좀 더 현실적으로 표현하면 어떤 디렉토리에 있는 어떤 파일인지에 관한 정보를 가진다. 이 컴포넌트를 폼에 배치하면 요런 아이콘 하나만 폼에 나타나며 실행중에는 보이지 않는다. 즉 메뉴나 타이머와 마찬가지로 비가시적(nonvisual) 컴포넌트이다. 후술하는 Query나 DataSource도 실행중에는 보이지 않는다. Table 컴포넌트를 배치한 후 테이블과 BDE를 연결하기 위해 몇 가지 속성을 설정해 주어야 한다.

**① DatabaseName**

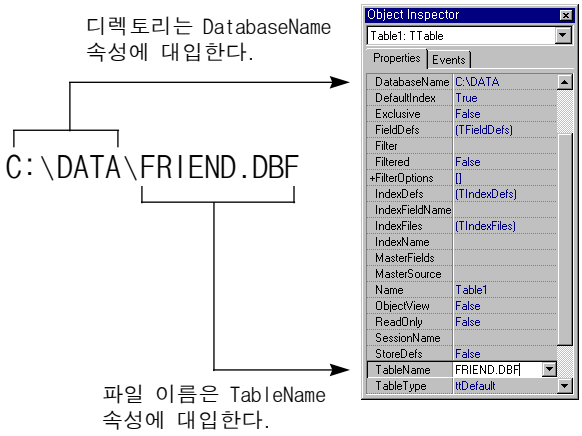
액세스할 데이터 베이스를 지정하며 다음 중 한 가지의 값을 설정한다. 앨리어스를 사용하는 방법이 가장 좋은 방법이지만 실습을 위해서라면 디렉토리 이름을 적어 주어도 좋다. 이 속성에 아무 값도 지정하지 않으면 실행 파일이 있는 현재 디렉토리를 사용한다.

1.BDE 에서 정의한 앨리어스
2.데이터 베이스 파일이 있는 디렉토리
3.Local InterBase 서버 데이터 베이스의 완전 경로
4.TDatabase 컴포넌트가 폼에 있을 경우 이 TDatabase 가 정의한 앨리어스

**② TableName**

데이터 베이스 테이블의 이름을 설정해 준다. DatabaseName 속성에 디렉토리 이름을 입력하고 TableName 속성에 파일 이름을 설정해 주면 된다. 예를 들어 c:₩data₩friend.dbf 파일을 사용하고자 한다면 디렉토리 이름인 c:₩data는 DatabaseName 속성에 대입하고 파일 이름인 friend.dbf는 TableName 속성에 대입한다.

그림
디렉토리와 테이블의 지정




**③ Active**

이 속성이 True여야 테이블이 열리고 테이블의 내용을 살펴볼 수 있다. 디자인시에 이 속성을 True로 설정하면 테이블에 연결된 데이터 컨트롤 컴포넌트에 테이블의 내용이 곧바로 출력된다. 디폴트값은 물론 False이다.

## ■ Query

이 컴포넌트는 SQL 명령을 사용하여 데이터를 액세스할 수 있도록 해준다. SQL(Structured Query Language)이란 데이터 베이스 운용에 사용되는 산업 표준 언어이며 델파이에서는 TQuery 컴포넌트로 구현된다. TQuery 컴포넌트를 폼에 배치한 후 SQL 속성에 사용할 SQL 명령을 기입해 주고 Open이나 ExecSQL 메소드로 SQL 명령을 실행한다. TQuery의 DatabaseName 속성은 TTable과 동일한 의미를 가진다.

이 컴포넌트에 관해서는 여기서 아무리 글을 읽어도 선뜻 이해가 되지 않으므로 잠시 후에 직접 실습을 하면서 좀 더 깊이 연구해 보기로 한다.

## ■ DataSource

데이터 셋은 일단 TQuery 와 TTable 을 합친 말이라고 생각하기 바란다.

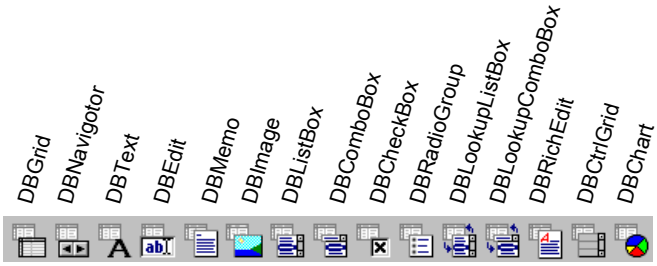
TTable이나 TQuery 등의 데이터 액세스 컴포넌트들은 BDE와 테이블을 연결해 주지만 테이블의 내용을 화면에 출력하지는 못한다. 반면 TDBEdit나 TDBText 등의 데이터 컨트롤 컴포넌트들은 데이터를 화면에 보여줄 수 있는 능력은 있지만 데이터가 들어 있는 테이블의 구조가 어떠한가를 모른다. 이 두 컴포넌트를 연결해 주어 상호 부족한 점을 보충할 수 있도록 해 주는 컴포넌트가 TDataSource 컴포넌트이다. DataSet 속성에 TTable이나 TQuery 컴포넌트의

이름을 지정하여 TDataSource와 데이터 셋을 연결한다. 데이터 컨트롤 컴포넌트들은 DataSource 속성에 TDataSource 컴포넌트의 이름을 지정하여 TTable이나 TQuery와 연결을 유지한다.

# 라. 데이터 컨트롤 컴포넌트

Data Controls 페이지에는 무려 15개나 되는 컴포넌트가 있으며 각 컴포넌트의 이름은 다음과 같다.

**그림**
데이터 컨트롤 컴포넌트




컴포넌트의 이름을 대충 훑어보면 DBEdit, DBListBox, DBCheckBox 등이며 앞에 DB라는 접두어가 붙어 있을 뿐 Standard 페이지에 있는 표준 컨트롤들과 이름이 거의 비슷하다. 실제로 이들의 기능도 표준 컴포넌트와 거의 같으며 속성이나 메소드들도 대부분 같다. 다만 한 가지 추가된 점이라면 테이블과 연결해 주면 테이블의 데이터를 읽어오거나 편집하고 표시할 수 있는 데이터 인식(data aware) 능력을 가진다는 점이다.

DBEdit는 모양과 속성, 그리고 하는 기능이 표준 에디트 컴포넌트와 거의 동일하다. 텍스트의 길이에 따라 크기를 자동 조절하는 AutoSize 속성, 읽기 전용으로 만드는 ReadOnly 속성, 입력 한계를 지정하는 MaxLength 속성 등등, 표준 에디트 컴포넌트의 속성을 그대로 가진다. 데이터 소스와 연결하려면 DataSource 속성에 TDataSource 컴포넌트의 이름을 지정해 주며 이로써 DBEdit가 디스크상의 데이터 베이스 파일에 저장된 정보를 읽어 출력할 수 있게 된다. 테이블 내용 중 어떤 필드의 내용을 읽을 것인가는 DataField 속성으로 지정하며 오브젝트 인스펙터에서 이 속성을 선택하면 테이블에 정의되어 있는 필드 목록이 나타난다.

DBNavigator는 테이블의 레코드를 이동할 수 있는 버튼과 추가, 삭제를 할 수 있는 버튼을 제공한다. DataSource 속성에 TDataSource 컴포넌트의 이름

을 정의하여 테이블과 연결해 놓기만 하면 버튼의 기능들이 자동적으로 구현되도록 되어 있다.

여기까지 여러분들은 델파이로 데이터 베이스 프로그램을 짜기 위한 전반적인 개념 사항에 대해 알아보았다. 전반적인 구조와 몇몇 대표적인 컴포넌트들의 특징, 역할 등에 관해 대충의 학습을 했다. 그러나 여기까지 읽고서 실제로 델파이로 데이터 베이스 프로그램을 짤 수 있는 사람은 아마 없을 것이라 생각되며 도무지 무슨 말을 하는건지 아직까지 개념이 잡히지 않는 사람들이 대부분일 것이다.

델파이로 데이터 베이스 프로그램을 짜는 것은 결코 어려운 일은 아니다. 하지만 처음 개념을 잡기 위해서는 이런 이론들만 나열해서는 알기가 어려우며 간단하더라도 직접 프로그램을 짜 보지 않으면 이해하기가 힘들다. 실습을 해 본 후 대충의 구조를 파악하고 도움말과 레퍼런스를 참조하여 점점 더 깊고 상세하게 학습해 가는 방법이 좋을 것 같다. 처음부터 이론을 다 공부하고 난 후에 실습을 하기에는 너무 복잡하므로 이론을 바탕으로 실습을 하고 실습을 통해 이론을 확립하는 상호 보완적인 학습 방법을 쓰도록 하자.

# 마. DBD의 사용

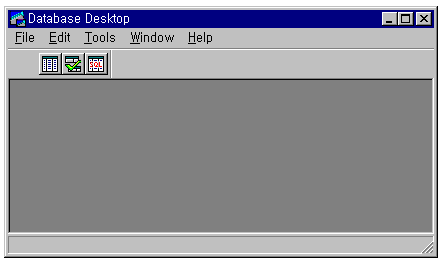
데이터 베이스 프로그램은 데이터를 관리하는 프로그램이다. 따라서 어떤 데이터를 처리하는가에 따라 프로그램의 기능이나 구조면에서 차이가 나며 프로그램 개발 이전에 데이터를 어떤 형식으로 구성할 것인가를 결정해야 한다. 즉 주소록에는 주소록에 맞는 프로그램을 짜야 하고 재고 관리나 비디오 가게 관리에는 그에 걸맞는 프로그램을 만들어야 한다. 즉 코드가 데이터에 종속된다는 얘기다.

그래서 데이터 베이스 프로그램 개발의 첫 단계는 데이터의 성질을 분석하여 적당한 데이터 베이스를 구축하는 것이며 구체적으로 예제 DB 파일을 작성하는 것이 제일 먼저 해야 할 일이다. 델파이는 이런 목적으로 사용하기 위해 DBD(Database Desktop)라는 프로그램을 제공하여 간단한 데이터 베이스 파일을 만들 때 사용하도록 하고 있다. DBD는 Borland Delphi 4 그룹에  이런 아이콘으로 등록되어 있다. DB 프로그래밍을 배우기 전에 먼저 DB 파일을 만들 수 있는 DBD를 배우는 것이 선행되어야 할 것이다. 물론 도스용 DBase III+나 윈도우즈용 Paradox를 쓸 줄 안다면 DBD는 굳이 사용할 필요가 없으므로 바로 다음 절로 점프해도 무관하다.

DBD로 할 수 있는 일은 간단한 DB 파일을 만들고 데이터를 입력하고 기존의

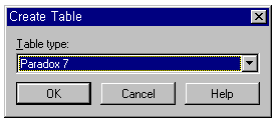
DB 파일을 살펴볼 수 있는 정도이다. 그야말로 미니 데이터 베이스 프로그램이라고 할 수 있다. DBD는 독립적으로 실행되는 프로그램이지만 델파이의 Tools/Database Desktop 메뉴에서도 부를 수 있다. 다음과 같은 수수하고 검소한 외모를 가지고 있다.

그림
미니 데이터 베이스 프로그램인 DBD




## ■ 새로운 테이블 만들기

새로운 DB 테이블을 만들려면 File/New/Table 항목을 선택한다. 그러면 다음과 같은 대화상자를 열어 테이블 타입을 묻는다.




즉 어떤 DB 프로그램으로 작성한 테이블을 만들 것인가를 묻는 것이다. 디폴트는 Paradox가 선택되어 있지만 좀 더 우리에게 친근한 DBase III+를 선택하도록 하자.

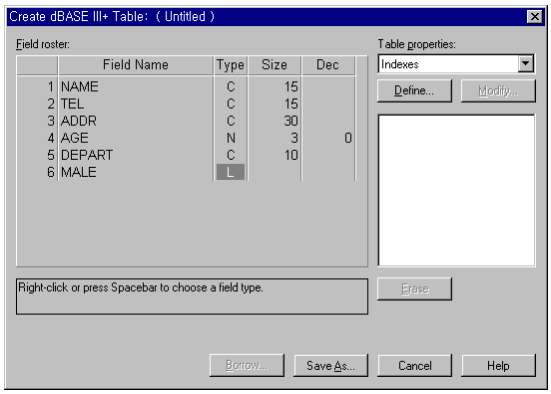
참고하세요

다 알다시피 델파이는 볼랜드사에서 만들었으며 볼랜드사에서 만든 데이터 베이스 프로그램이 파라독스이다. 그러다 보니 델파이와 궁합이 제일 잘 맞는 데이터 베이스 프로그램은 당연히 파라독스이며 파라독스를 사용해야 델파이의 모든 기능을 십분 발휘할 수 있다. 마치 마이크로소프트사의 비주얼 베이직이 마이크로소프트사의 액세스를 가장 잘 지원하는 것과 같다. 그러나 우리나라에서 현실적으로 가장 많이 사용하는 DB 프로그램은 뭐니 뭐니 해도 DBase이며 파라독스를 사용하는 사람은 찾아보기가 힘드므로 이 책에서는 DBase를 주로 사용한다.

그러면 새로운 테이블의 구조를 입력할 수 있는 대화상자가 열린다. 이 대화상자에서 DB에 포함시킬 필드의 이름과 타입, 속성을 설정해 준다. 필자가 입력

한 필드는 다음과 같다.

그림
필드 입력 대화상자

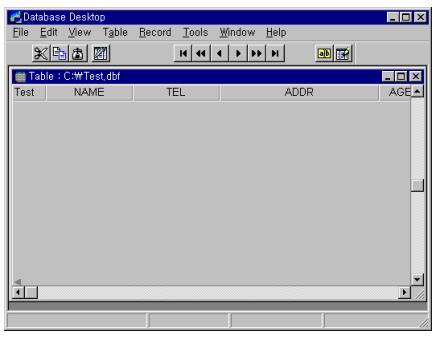



이 대화상자가 무엇을 요구하며 어떻게 사용하는가는 DBase를 조금이라도 공부해 본 사람이라면 다 알 것이므로 지나치게 친절한 설명은 생략한다. 완전히 도스용 DBase III+와 동일한 화면이다. 필드 구조를 입력한 후 Save As 버튼을 눌러 이 테이블에 TEST.DBF라는 파일 이름을 주고 저장한다.

디렉토리는 직접 경로를 선택하거나 아니면 디렉토리의 별명인 앨리어스(잠시 후 설명함)를 선택한다. 테이블 구조를 정의하고 디스크에 저장하면 일단 테이블이 생성되고 DBD는 처음 실행되었을 때와 같은 모양이 된다.

## ■ 데이터 입력하기

새로 만든 테이블에 데이터를 입력하려면 이 테이블을 열어야 한다. File/Open/Table을 선택하거나 툴바에서 버튼을 누른다. 파일 열기 대화상자에서 TEST.DBF를 선택하면 이 테이블의 모습이 DBD에 나타날 것이다.




아직 테이블만 만들어져 있고 데이터는 전혀 입력되어 있지 않다. 데이터를 입력하려면 툴바의 버튼을 누르거나 키보드의 F9를 눌러 편집 상태로 전환한

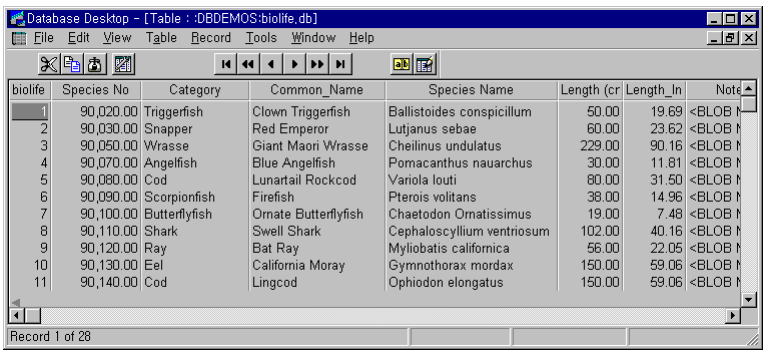
다. 테이블에 커서가 나타나며 입력 대기 상태가 될 것이다. 입력 상태에서 마치 스프레드 시트를 사용하듯이 데이터를 입력하면 된다. 입력중의 모습은 다음과 같다.

그림
DBD 에서 곧바로 데이터를 입력한다.


DBD에는 입력한 데이터를 별도로 저장하는 명령을 내리지 않아도 된다. 왜냐하면 변경한 데이터는 당연히 저장되어야 하며 저장 명령을 내리지 않아도 저장되기 때문이다. 테이블을 닫거나 DBD를 끝내면 알아서 데이터를 저장한다.

## ■ 테이블 보기

DBD로 기존의 테이블을 살펴보거나 수정할 수도 있다. 살펴보는 방법은 File/Open 메뉴 항목을 선택하여 열어 보는 것이다. 편집하는 방법도 앞에서 보인 테이블 만드는 방법과 별로 다르지 않다. 다음은 델파이와 함께 제공되는 BIOLIFE.DB라는 테이블의 내용을 살펴 본 것이다.

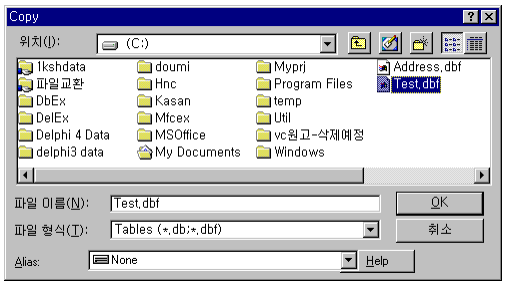



그러나 안타깝게도 그래픽이나 메모같은 크기가 큰 데이터는 보여주지 못한다. DBD는 데이터 베이스 파일이 어떤 구조를 가지는지 살펴 보기에 적당하며

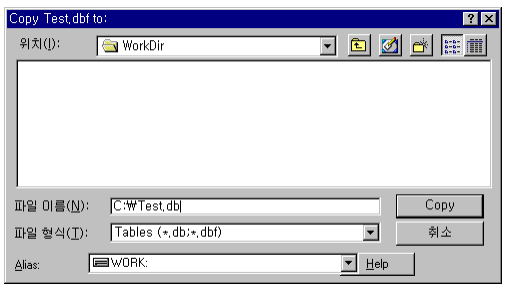
테이블 내용을 완전히 보려면 아무래도 파라독스나 DBase가 있어야 할 것이다.

## ■ 테이블 변환하기

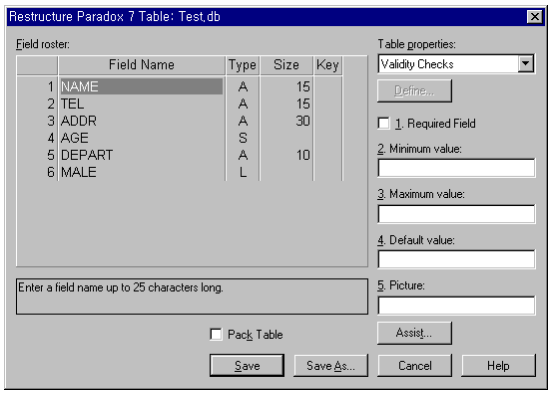
DBase 로 만든 테이블을 파라독스로 변환하거나 또는 파라독스로 만든 테이블을 DBase 테이블로 변환해야 할 경우가 있다. 이때는 Tools/Utility/Copy 명령으로 파일의 확장자만 바꾸어 다른 파일로 저장해 주면 된다. Test.dbf 를 파라독스 포멧으로 변경해 보도록 하자. 테이블을 열 필요없이 Tools/Utility/Copy 명령을 선택하면 입력 파일명을 요구한다.




이 대화상자에서 변환하고자 하는 테이블을 선택하는데 Test.dbf 를 선택해 보자. 그러면 변환후의 파일을 물어온다.




이 대화상자에서 파일의 확장자만 바꾸어주면 된다. Test.db 로 파일의 확장자를 바꾸어주면 이 파일은 파라독스 포맷으로 저장된다. 과연 제대로 저장이 되었는지는 이 테이블을 다시 열어보면 된다. Table/Restructure 명령으로 확인해 보면 필드들이 제대로 변환되어 있을 것이다.



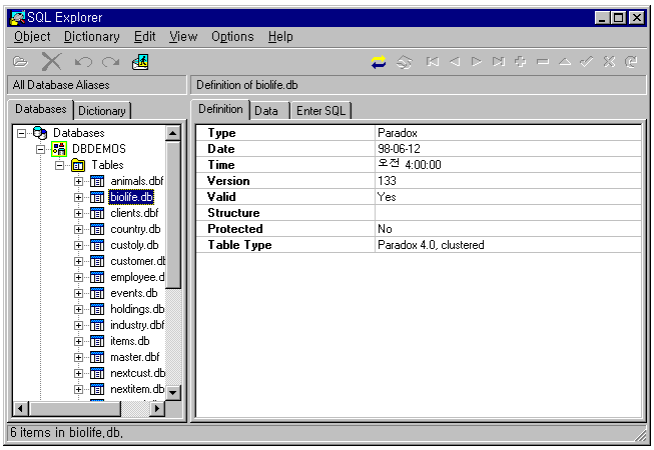

DBase 의 문자열(C)은 A(Alpha)형으로 정수형(N)은 S(Short)형으로 변환되어 있을 것이다.

이 외에도 DBD가 가지는 기능은 많지만 당장 실습에 필요한 테이블 만들기, 보기, 데이터 입력하기만 알아보고 나머지는 직접 도움말을 참조하여 공부해 보기 바란다.

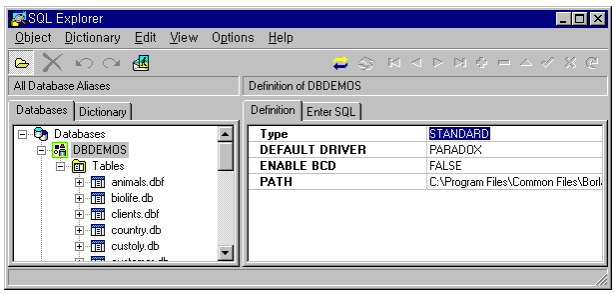
# 바. SQL 탐색기

SQL 탐색기는 데이터 편집 기능과 데이터를 검색해 볼 수 있는 기능을 가진 브라우저이다. DBD 를 대체하기 위해 만들어진 듯 하지만 테이블을 만들 수 있는 기능이 제외되어 있기 때문에 주로 데이터 베이스 파일을 찾거나 관리하는 용도로 사용된다. 델파이 버전별로 기능 제한이 있는데 여기서는 클라이언트/서버 버전을 중심으로 설명한다. 델파이 그룹에서 SQL Explorer 를 선택하거나 델파이에서 직접 Database/Explore 메뉴 항목을 선택하여 SQL 탐색기를 실행시킨다 .

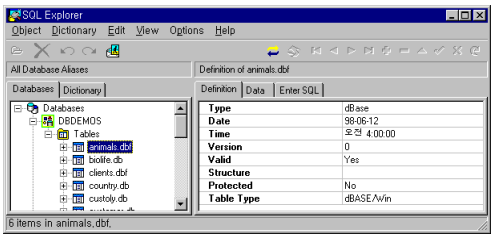
그림
SQL 탐색기



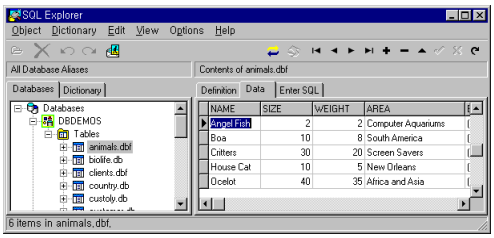

메뉴와 툴바가 있고 작업 영역은 좌, 우 두 개의 영역으로 분할되어 있다. 좌측은 데이터 베이스의 여러 요소를 계층적인 트리 구조로 보여주며 윈도우즈의 파일 탐색기를 사용하는 방법대로 사용하면 된다. 우측은 좌측에 선택된 항목의 세부적인 내용을 보여주되 좌측에 어떤 항목이 선택되어 있는가에 따라 출력되는 내용이 달라진다. 예를 들어 좌측에 앨리어스가 선택되어 있다면 그 앨리어스의 형태, 경로에 대한 정보가 나타나고 테이블이 선택되어 있으면 테이블의 구조, 데이터를 보여준다. 다음은 DBDEMOS 앨리어스를 선택했을 때의 모습이다.



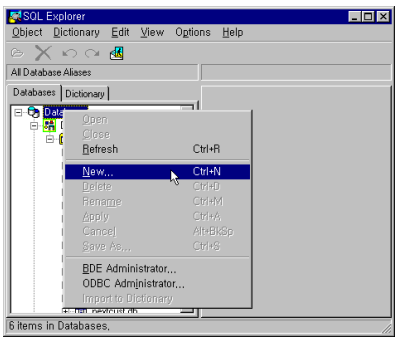

오른쪽에 DBDEMOS 앨리어스에 대한 정보가 나타난다. 다음은 animals.dbf 테이블을 선택했을 때의 모습이다.




오른쪽에 세 개의 탭이 있는데 Definition 탭에는 이 테이블에 대한 정보가 나타나며 Data 탭에는 테이블의 구조와 데이터들이 그리드 형태로 나타난다. Data 탭에서 마치 스프레드 시트를 사용하듯이 테이블의 정보를 직접 수정할 수도 있다.




DBD 와 다른 점은 그래픽 데이터도 보여줄 수 있다는 점이다. 그리드에 (GRAPHIC) 또는 (MEMO)라고 표시된 필드도 더블클릭하면 별도의 윈도우를 열어 데이터를 보여준다. 레코드간의 이동이나 삽입 삭제는 오른쪽 위에 있는 네비게이터를 사용하며 추가로 데이터를 더 입력하고자 할 때는 제일 끝의 레코드로 이동한 후 아래쪽 커서 이동키를 누르면 새로운 빈 레코드를 만들어 준다. 그 외 앨리어스를 새로 만든다거나 수정한다거나 하는 동작은 대부분 마우스 오른쪽 버튼을 누르면 나타나는 스피드 메뉴를 사용한다.



# 15-2 데이터 컨트롤

## 가. 주소록

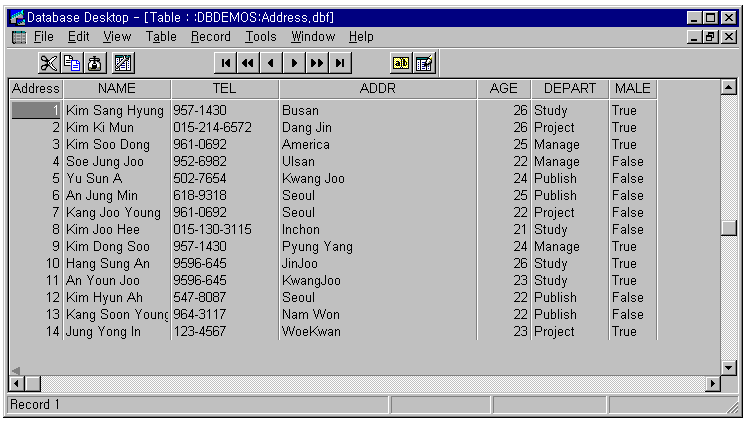
15jang
addr

여기서는 일단 이론을 접어두고 극단적으로 간단한 데이터 베이스 예제를 직접 작성하면서 개발 절차와 개념적인 면들에 관해 정리해 보고자 한다. 철저하게 단계를 따라 설명하였으므로 책만 읽지 말고 컴퓨터 앞에서 실습을 하면서 공부를 하도록 하자. 책만 읽어서는 실력향상에 전혀 도움이 되지 않을뿐만 아니라 오히려 더 헷갈리기만 할 것이다.

### 1 DBF 파일부터 작성한다.

데이터 베이스 프로그램을 작성하려면 일단 재료가 될 데이터 베이스 파일이 필요하다. 델파이에서 직접 사용할 수 있는 데이터 베이스 파일에는 DBase III+ 이상 버전의 DBF 파일과 Paradox의 DB 파일이 있다. 여기서는 가장 대중적인 데이터 베이스 프로그램인 DBase III+를 사용하여 데이터 베이스 파일을 만들었다. 기능적으로 파라독스에 비해 한수 아래이기는 하지만 오래전부터 데이터 베이스를 공부한 사람은 대부분 알고 있기 때문에 DBase III+를 선택했다. 물론 DBase III+는 도스용 프로그램이다. 이외에도 DBase for Windows나 델파이에서 제공하는 간단한 데이터 베이스 제작 프로그램인 DBD.EXE를 사용해도 된다.

DBD를 사용하여 다음과 같이 6개의 필드를 가지는 간단한 구조의 데이터 베이스 파일을 입력하되 입력되는 데이터는 자기 마음대로 친한 친구의 이름을 사용해도 되며 개수는 10개 정도만 입력하도록 하자. 이 파일을 ADDRESS.DBF라는 파일 이름으로 저장해 두되 예제를 작성할 디렉토리에 일단 저장해 두는 것이 좋겠지만 일단은 C 드라이브의 루트 디렉토리에 저장하도록 하자. 반드시 C:₩ADDRESS.DBF로 저장해야 이어지는 실습에 지장이 없다.



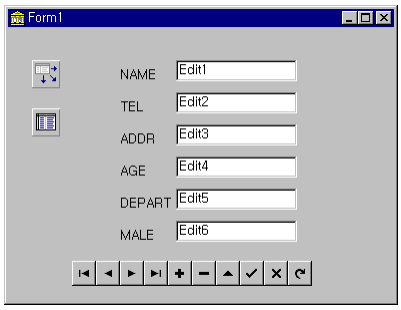

한글도 물론 사용할 수 있으며 사진이나 그림같은 이미지를 사용할 수도 있지만 여기서는 예제의 극단적 단순함을 위해 되도록 간단한 파일을 쓰기로 한다.

## 2 프로젝트를 시작한다.

델파이를 실행시키고 새로운 프로젝트를 시작한다. 여기까지는 늘상 해오던 실습과 동일하다. 빈 폼이 나타날 것이다.

## 3 컴포넌트를 폼에 배치한다.

이 예제에서 필요한 컴포넌트는 DataSource, Table 각각 하나씩, DBEdit와 Label 각각 여섯 개, 그리고 DBNavigator 하나이다. 이 컴포넌트들을 폼에 다음과 같이 배치한다.




예제 제작의 신속함을 위해 모두 디폴트 속성과 이름을 그대로 사용하되 단 레이블의 캡션만 위의 그림대로 설정한다. 폼에 컴포넌트를 배치했으면 일단 프로젝트를 저장하도록 하자. 적당한 디렉토리에 폼을 ADDR_F.PAS라는 이름으로, 프로젝트는 ADDR.DPR이라는 이름으로 저장해 둔다.

## 4 Table1에 DBF 파일을 연결해 준다.

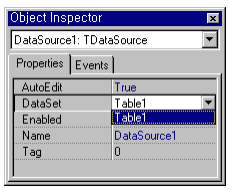
앞에서 우리가 만들었던 DBF 파일을 프로그램에서 사용하기 위해서는 데이터 베이스 파일과 데이터 베이스 엔진을 연결해 주어야 한다. 이 일을 담당하는 컴포넌트가 Table 컴포넌트이며 예제에서의 이름은 Table1이다. Table1 컴포넌트의 속성을 다음과 같이 변경한다.

| 속성 | 속성값 | 의미 |
|---|---|---|
| DatabaseName | C:₩ | DBF 파일이 있는 디렉토리 |
| TableName | ADDRESS.DBF | DBF 파일 이름 |
| Active | True | DBF 파일 활성화 |

Table 컴포넌트와 디스크상의 테이블을 연결하는 방법에는 여러 가지가 있지만 현실적으로 DatabaseName에 디렉토리 이름을, TableName에 데이터 베이스 파일 이름을 지정해 주는 것이 제일 쉽고 간단하다. 이 두 속성을 지정함으로써 Table1은 C:₩ADDRESS.DBF 테이블과 연결된다. Table1 컴포넌트의 속성을 설정하여 BDE와 DBF 파일을 연결해 주어도 아직까지 화면에는 아무런 변화가 없을 것이다. Table1은 내부적 연결만을 담당할 뿐 테이블의 내용을 화면으로 출력할 수는 없기 때문이다.

## 5 Table1을 DataSource1에 연결한다.

Table1의 내용을 화면에 출력하려면 데이터 컨트롤 컴포넌트인 DBEdit가 필요하고 Table1과 DBEdit를 연결해 주는 것은 DataSource1 컴포넌트이다. 그래서 DataSource1 컴포넌트를 Table1에 연결해 주어야 한다. DataSource1의 DataSet 속성에 Table1을 입력해 준다. 오브젝트 인스펙터에서 DataSet 속성을 선택하면 드롭다운 리스트 박스가 열리고 이 리스트에는 벌써 Table1이 입력되어 있으므로 선택만 하면 된다.



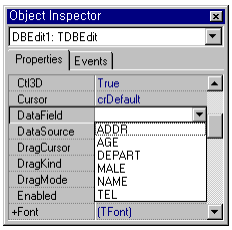

ADDRESS.DBF가 Table1에 연결되었으며 Table1이 DataSource1에 연결

되었다.

## 6 DBEdit에 DataSource1을 연결한다.

테이블의 내용을 화면에 출력하기 위해서는 데이터 컨트롤 컴포넌트를 사용하며 이 예제에서는 DBEdit 컴포넌트 여섯 개를 사용하였다. 데이터 컨트롤 컴포넌트가 어떤 테이블의 데이터를 출력할 것인가는 DataSource 속성으로 설정해 준다. 여섯 개의 DBEdit를 한꺼번에 선택한 후 DataSource 속성에 DataSource1을 입력해 준다. 이 경우도 드롭다운 리스트에 벌써 DataSource1이 입력되어 있으므로 선택만 하면 된다.

그리고 각 DBEdit가 어떤 필드의 내용을 출력할 것인가를 DataField 속성에 설정해 준다. 이 속성을 선택하면 앞에서 작성한 ADDRESS.DBF 파일에서 정의한 필드의 목록이 나타날 것이다. 이 목록은 오브젝트 인스펙터가 DataSource, Table 컴포넌트를 거쳐 C:₩ADDRESS.DBF 파일의 필드를 조사해서 보여주는 것이다.



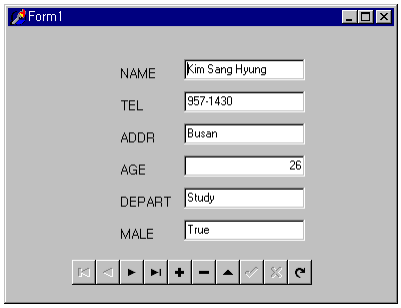

DBEdit1에서부터 순서대로 NAME, TEL, ADDR, AGE, DEPART, MALE을 지정해 준다. 여기까지 지정해 주면 데이터 베이스의 첫 번째 레코드 내용이 벌써 DBEdit에 출력될 것이다. ADDRESS.DBF의 데이터들이 BDE를 거쳐 Table1으로 전달되고 다시 DataSource1을 거쳐 각각의 DBEdit로 전달되기 때문이다.

## 7 DBNavigator를 DataSource1에 연결한다.

프로그램 실행중에 레코드 사이를 이동해 보려면 DBNavigator가 필요하며 이 컴포넌트가 어떤 테이블의 이동을 담당할 것인가를 지정하기 위해 DataSource 속성에 DataSource1을 지정해 준다. 네비게이터에는 아홉 개의 버튼이 있으며 실행중에 사용자는 각 버튼을 사용하여 레코드를 이동하거나 편집한다.

### 8 저장하고 실행한다.

애써 책 한 번 보고 화면 한 번 보고 더듬더듬 만든 프로그램, 혹시 잘못될지도 모르니까 일단 덮어놓고 저장부터 해 두도록 한다. 그리고 이제 실행 버튼을 눌러 프로그램을 실행해 보자. 특별히 잘못한 것이 없다면 화면에 다음과 같은 윈도우가 나타날 것이다.




데이터 베이스의 내용을 화면에 보여주고 아래쪽의 버튼을 사용하여 레코드 사이를 이동할 수 있으며 DBEdit에 직접 입력하여 내용을 변경할 수도 있다.

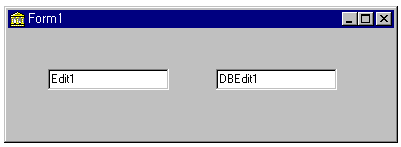
고생해서 만든 프로그램치고는 너무 실망스러울 정도로 간단해 보일 것이다. 그러나 이 프로그램은 다양한 검색, 수정, 열람이 가능하며 그래픽까지 보여줄 수 있는 대형 데이터 베이스 프로그램의 기본 형태를 가진 그야말로 무한한 잠재력을 가진 프로그램이다. 여기에 어떤 기능을 더 덧붙이기란 그야말로 식은 죽 먹기다.

위 예제를 만들면서 우리는 단 한 줄의 코드도 작성하지 않았다. 모든 지정과 설정을 컴포넌트의 배치와 속성 설정으로 해 내었으며 그 배경에는 델파이라는 친절하고 강력한 후원자가 버티고 있다는 것을 알아야 한다. 여기까지의 과정은 기본적이므로 통째로 암기해도 손해 볼 것이 없으며 DB 프로그램에 관심이 있다면 마땅히 암기해야 하고 조금만 노력하면 암기할 수 있다. 책을 덮고 직접 몇 번 실습을 해본 후 앞부분의 이론을 다시 한 번 확인하도록 하자.

## 나. DBEdit

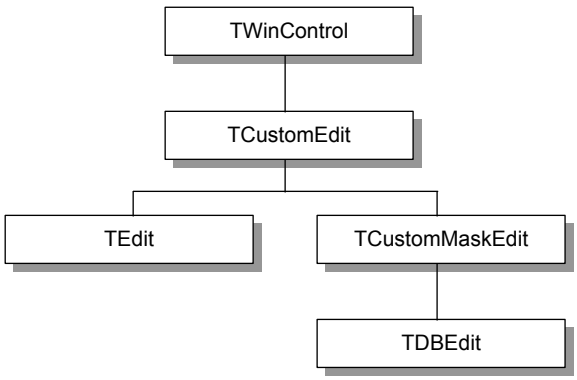
ADDR 예제에서 우리는 테이블의 필드 내용을 보기 위해 DBEdit 컴포넌트를 사용했다. 이름에서 알 수 있듯이 이 컴포넌트는 Edit 컴포넌트에 데이터 베이스 기능이 추가된 것 뿐이다. 폼에 Edit 컴포넌트와 DBEdit 컴포넌트를 각각 배치

해 보자.




이름만 다를 뿐이지 모양은 완전히 똑같다는 것을 알 수 있다. 그럼 두 컴포넌트를 번갈아가며 오브젝트 인스펙터를 살펴보자. 자세히 관찰해 보면DBEdit는 Edit에 비해 DataField, DataSource 속성이 추가되어 있다. 데이터 베이스와 연결을 해야 하므로 이 속성들이 필요한 것이다. 또한 HideSelection, OEMConvert 속성이 없어졌는데 이 두 속성은 데이터 베이스에는 불필요하기 때문이다. 그리고 또 한 가지 차이점은 DBEdit에는 Text 속성이 없는 것처럼 보이는데 사실은 실행중에만 사용할 수 있는 속성으로 변경되었다. 생각해 보면 알겠지만 DBEdit는 DB의 필드 내용을 보여주므로 디자인 중에 Text 속성을 변경하는 것은 무의미하다. 다만 실행중에는 Text 속성을 읽어 필드값을 조사할 수 있다.

속성에 몇 가지 변화가 있지만 이벤트는 전혀 변화가 없으며 메소드는 몇 개 추가되었다. 두 컴포넌트의 계층 구조상의 위치를 그려보면 다음과 같다.




DBEdit와 Edit의 가장 큰 차이점은 데이터 인식 능력이다. 즉 DBEdit는 테이블만 연결해 주면 데이터를 보여주는 능력이 있으며 또한 이 컴포넌트의 텍스트를 편집하면 직접 테이블의 내용을 수정할 수 있다. 과연 그런지 ADDR.EXE를 실행한 후 첫 번째 필드를 변경해 보자. Kim Sang Hyung을 Cha In Pyo로 바꾸고 네비게이터의  버튼을 누른다. 그리고 예제를 종료한 후 다시 실행해 보면 필드 내용이 바뀌어 있을 것이며 DBD로 ADDRESS.DBF를 직접 조사해 봐도

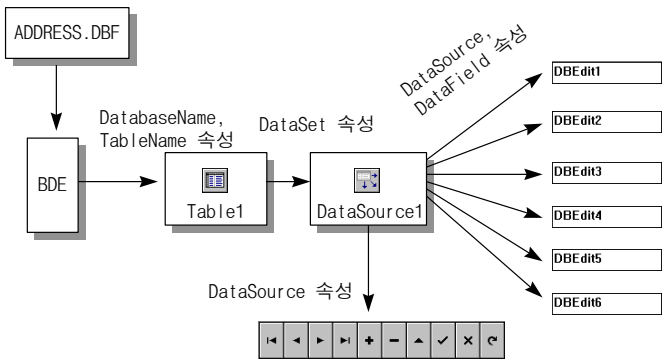
역시 바뀌어져 있을 것이다.

데이터 인식 능력이 있다는 것과 두 가지 속성이 추가되었다는 것 외에는 큰 차이가 없으므로 Edit의 모든 속성과 기법등을 그대로 사용할 수 있다. 즉 여러분의 기존 지식이 보호된다는 뜻이다. 에디트뿐만 아니라 DBListBox, DBRadioGroup 등의 컴포넌트들도 마찬가지로 기본 컴포넌트와 크게 틀리지 않다. 두 컴포넌트가 별로 다르지 않다는 얘기를 너무 길게 한 것 같다.

# 다. 네비게이터

ADDR.DPR 예제는 만들기도 쉽고 이해하기도 어렵지 않을 것이다. DBEdit는 다음과 같은 경로로 데이터 베이스 파일과 연결되며 하나의 DBEdit가 테이블 내의 한 필드에 대응하여 그 필드의 내용을 보여주고 입력을 받는다. 컴포넌트끼리의 연결 관계가 어떻게 되는지를 잘 정리해 둘 필요가 있다.

**그림**
데이터 액세스 컴포넌트와 데이터 컨트롤 컴포넌트의 연결




또한 네비게이터가 어떻게 테이블과 연결되는지도 잘 봐두기 바란다. 네비게이터는 TDataSource 속성으로 테이블 하나와 연결되며 연결된 테이블 내에서 레코드의 이동과 데이터의 편집을 담당한다. 아홉 개의 버튼을 가지며 각 버튼의 기능은 다음과 같다.

**그림**
네비게이터의 버튼

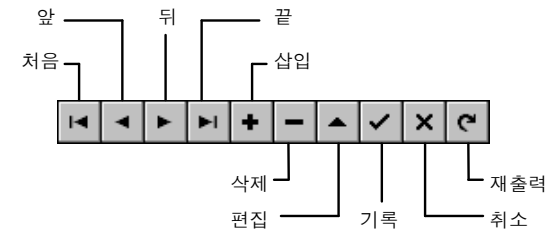
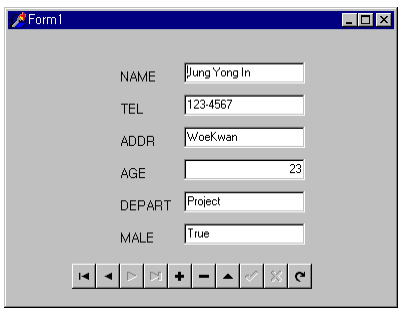


그림
네비게이터 버튼의 기능

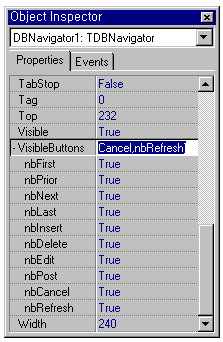
| 버튼 | 기능 |
|---|---|
| 처음 | 테이블 내의 첫 레코드로 이동한다. |
| 앞 | 앞의 레코드로 이동한다. |
| 뒤 | 뒤의 레코드로 이동한다. |
| 끝 | 테이블 내의 마지막 레코드로 이동한다. |
| 삽입 | 현재 레코드 앞에 빈 레코드를 삽입하며 테이블을 편집 상태로 만든다. |
| 삭제 | 현재 레코드를 삭제하며 뒤의 레코드를 현재 레코드로 만든다. |
| 편집 | 테이블을 편집 상태로 만든다. |
| 기록 | 편집한 데이터를 테이블에 기입한다. |
| 취소 | 편집을 취소한다. |
| 재출력 | 테이블로부터 레코드를 다시 읽어들인다. 테이블 하나를 네트웍에서 여러 사람이 사용할 때 사용한다. |

프로그램 실행중에 사용자가 네비게이터의 버튼을 누르면 즉시 레코드가 이동되며 테이블의 내용을 편집할 수도 있다. 뿐만 아니라 네비게이터는 데이터의 끝과 처음을 인식하며 언제 어떤 버튼이 사용가능한지를 알고 있다. 제일 끝 레코드에서는 뒤로 가기 버튼이 사용금지되며 테이블이 읽기 전용으로 열렸을 때는 모든 편집 버튼이 사용금지된다.




단 한 줄의 코드도 작성하지 않고 이런 기능을 제공받는다는 것은 참 놀라운 일이다. 네비게이터는 모든 것이 자동화되어 있기 때문에 특별히 알아둘만한 속성이 없는 셈이다. ShowHint 속성이 디폴트로 False로 되어 있는데 이 속성을 True로 변경하면 풍선 도움말을 볼 수 있고 개별 버튼의 표시 여부를 결정하는 VisibleButtons 속성이 있다. VisibleButtons 속성을 확장해 보면 개별 버튼의 표시 여부를 변경할 수 있는 세부 속성이 열리는데 디폴트로 모두 True로 설정

되어 있으므로 모든 버튼이 표시된다.




별로 필요없는 버튼이 있다면 이 속성을 변경하면 된다. 앞쪽 네 개의 버튼만 남기고 모두 숨겨 버리면  요렇게 될 것이다. 납작한 모양의 버튼을 만들고 싶다면 Flat 속성을 True로 바꿔주면 된다.

# 라. 여러 가지 데이터 컨트롤

DB 테이블에 보관된 데이터가 화면에 출력되는 것은 데이터 컨트롤을 통해서이다. ADDR.DPR에서 사용된 DBEdit는 DataField로 연결된 필드의 데이터를 문자열 형태로 보여준다. DB 테이블의 필드는 대부분 문자열이나 숫자이므로 DBEdit만으로도 웬만한 테이블의 내용을 출력할 수 있다. 그러나 DBEdit는 데이터를 입력받을 때 일일이 키보드를 사용해야 하므로 무척 불편하다. 델파이가 제공하는 나머지 데이터 컨트롤 컴포넌트를 사용하면 데이터를 좀 더 다양하게 보여주고 좀 더 편리하게 입력받을 수 있다.

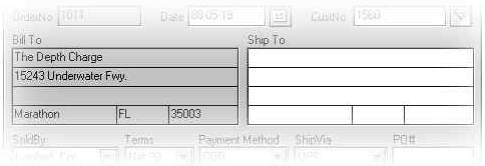
각종 데이터 컨트롤을 테이블에 연결하는 방법은 DBEdit의 경우와 동일하다. 즉 DataSource 속성에 TDataSource 컴포넌트를 기입하고 DataField 속성에 연결할 필드명을 기입해 주면 된다.

## ■ DBText

DBText는 데이터 인식 레이블이라고 할 수 있으며 연결된 필드의 데이터를 화면상에 보여주기만 한다. DBEdit는 데이터를 보여주며 편집까지 할 수 있는데 반해 DBText는 입력을 받지 못하므로 읽기 전용의 DBEdit와 같은 셈이다. 사용자에게 보여주기만 하고 사용자가 변경해서는 안되는 필드(예를 든다면 고객 고유 번호 등)는 DBEdit보다는 DBText에 출력하는 것이 적합하다. 사용하는 속성

이나 모양은 표준 Label 컴포넌트와 동일하다.

그림
데이터를 보여 주기만 하는 DBText




## ■ DBCheckBox

데이터를 인식한다는 점을 제외하고는 표준 체크 박스와 동일하다. 예/ 아니오, True/False 등의 두 가지 상태 중 한 가지를 보여주거나 선택하는 데 사용된다. 주로 데이터 베이스의 논리형(Logical) 필드와 연결되어 사용된다. 예를 들어 남/여를 나타내는 MALE 필드라든가 정회원/준회원을 나타내는 필드의 정보를 표현한다. 물론 DBEdit로도 논리형 필드의 데이터를 출력할 수 있지만 DBCheckBox를 사용할 경우는 사용자가 마우스로 데이터의 내용을 편리하게 바꿀 수 있으며 좀 더 직관적으로 데이터를 출력할 수 있다. 사용하는 속성이나 방법은 표준 체크 박스 컴포넌트와 동일하다.

## ■ DBRadioGroup

데이터 베이스에 사용되는 라디오 그룹이며 내부에 여러 개의 라디오 버튼을 가진다. 필드에 입력되는 값이 일정한 범위 안에 있으며 유한 개수일 때 주로 사용된다. 예를 들어 부서를 나타내는 DEPART 필드의 경우 Study, Project, Manage, Publish 네 가지 경우밖에 없으므로 이 값들을 일일이 키보드로 입력하는 것보다는 라디오 그룹에서 선택하는 것이 훨씬 더 쉽다. 그룹에 라디오 버튼을 포함시키는 방법은 역시 표준 라디오 그룹과 동일하다.

## ■ DBListBox

표준 리스트 박스와 같되 데이터를 인식한다는 점만 다르다. 사용 용도는 DBRadioGroup과 완전히 동일하다. 즉 선택 가능한 값이 정해져 있을 때 그 목록을 보여주고 사용자가 선택하도록 한다. 라디오 그룹과의 차이점이라면 좀 더 많은 수의 목록을 스크롤해가며 선택할 수 있다는 점이다. 목록은 Items 속성에 문자열 리스트로 입력되므로 디자인시에나 실행시에 언제든지 입력할 수 있으며 문자열 리스트에 사용되는 Add, Insert, Delete 등의 메소드도 물론 사용 가능하다.

## ■ DBComboBox

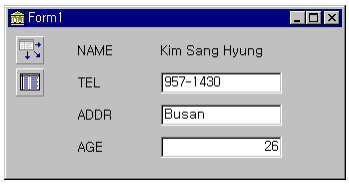
표준 콤보 박스와 같되 데이터를 인식한다는 점만 다르다. 사용 용도는 표준 콤보 박스와 완전히 동일하다. 즉 여러 개의 가능한 값 중에 하나를 선택하거나 아니면 직접 키보드로 입력받는다. 선택 가능한 값이 대개 정해져 있되 아주 특별한 경우 예외가 있다면 리스트 박스보다는 콤보 박스를 사용해야 한다.



15jang
addr2

컴포넌트의 수가 많아 익히기가 어려울 것 같지만 표준 컴포넌트들과 거의 틀리지 않으므로 생각보다 쉽게 익힐 수 있을 것이다. 이 컴포넌트들을 직접 사용해 보기 위해 ADDR.DPR을 수정하여 ADDR2.DPR을 만들어 보자. DBEdit로 모든 필드의 내용을 표시하던 것을 다른 데이터 컨트롤 컴포넌트를 사용하여 적당히 바꾸어 볼 것이다. 앞에서 만들었던 예제와 기능적으로 같은 예제이지만 복습도 해 볼겸 다시 한번 더 만들어 보도록 하자.

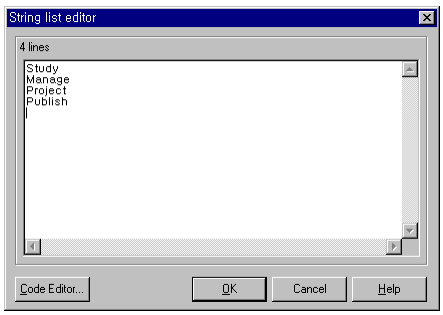
우선 새 프로젝트를 시작하고 Table, DataSource 컴포넌트를 배치한 후 이 컴포넌트의 속성을 적당히 변경하여 테이블과 연결한다. Table의 DatabaseName 속성에 C:₩를 주고 TableName 속성에 ADDRESS.DBF를 주어 DBF 파일과 연결한다. 그리고 DataSource의 DataSet 속성에 Table1을 선택해 준다. 이제 Table의 Active 속성을 True로 바꾸어 주면 테이블과의 연결이 끝났다. 데이터를 표시할 컨트롤들을 배치해 보자. 우선 NAME, TEL, ADDR, AGE라는 캡션을 가지는 레이블들을 먼저 배치하고 TEL, ADDR, AGE 옆에는 DBEdit를 배치하고 NAME 옆에는 DBText를 배치한다. 물론 각 데이터 컨트롤의 DataSource 속성에는 DataSource1을 설정하고 DataField 속성은 레이블과 같은 이름의 필드와 연결해 준다. 여기까지 작업하면 폼의 모양은 다음과 같아질 것이다.



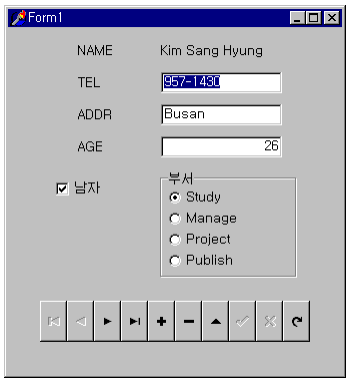
NAME 필드는 DBText 컴포넌트로 출력하도록 하여 사용자가 실행중에 값을 변경하는 것을 금지시켰다. 물론 이 경우는 다분히 억지가 있지만 함부로 수정되어서는 안되는 필드에는 반드시 DBText를 사용해야 한다.

다음으로 남/녀의 성별을 나타내는 필드를 보여 주도록 해 보자. 진위형의 필드이므로 DBCheckBox로 출력하는 것이 적당할 것 같다. 이 컴포넌트를 배치하고 Caption에 "남자"를 설정하고 DataSource, DataField(MALE) 속성을 설정한다. 에디트 박스에 True, False로 출력하는 것보다는 체크 표시를 사용하는 것이 훨씬 더 직관적일 뿐만 아니라 마우스로 값을 변경할 수 있어서 편리하다.

부서는 가능한 값이 네 가지 밖에 없으므로 역시 라디오 그룹 박스가 제격이다. 만약 그렇지 않으면 Manage, Publish 등을 일일이 키보드로 입력해 주어야 하므로 귀찮을 뿐만 아니라 Stuby, Protect 등의 오타가 입력될 가능성까지 가지므로 위험하다. 라디오 그룹을 배치한 후 Items 속성에 다음 문자열들을 입력한다.




그리고 Caption에 "부서"를 설정하고 DataSource, DataField(DEPART) 속성을 설정하면 된다. 라디오 박스 대신 리스트 박스를 사용해도 기능상으로는 동일해진다. 마지막으로 네비게이터를 배치한 후 DataSource와 연결해 주고 디스크에 프로젝트를 저장하면 예제가 완성된다. 실행중의 모습은 다음과 같다.




ADDR 예제와 기능상으로는 동일하지만 다양한 컨트롤을 사용함으로써 훨씬 더 편리하게 사용할 수 있다. 각 컨트롤의 색상이나 폰트 등을 변경한다면 예쁘게 디자인할 수도 있을 것이다.

# 마. DBGrid

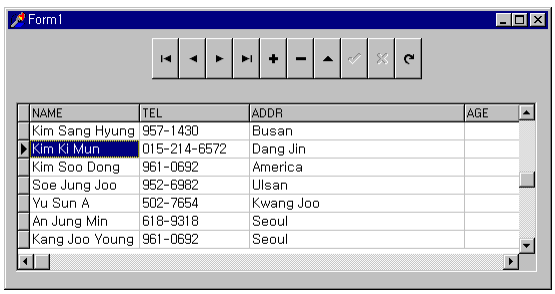
15jang
dbgrid

DBEdit를 비롯해서 이때까지 소개된 데이터 컨트롤들은 테이블 내의 한 필드에 대응하며 한 필드의 데이터만 보여준다. 이에 비해 DBGrid는 테이블 전체를 한눈에 보여줄 뿐만 아니라 편집도 물론 가능하다. 가로, 세로의 표 형식으로 테이블의 내용이 한꺼번에 출력된다. 특정 필드에 대응되지 않기 때문에 DataField 속성은 가지지 않으며 DataSource 속성에 출력하고자 하는 테이블의 이름만 지정해 주면 된다.

새 프로젝트를 시작하고 Table, DataSource, 네비게이터, DBGrid를 배치한다. 그리고 각 컴포넌트의 속성을 앞에서 실습한대로 적당히 연결하고 DBGrid.DataSource에 DataSource1을 연결해 주기만 하면 된다. 실행시의 모습은 다음과 같다.

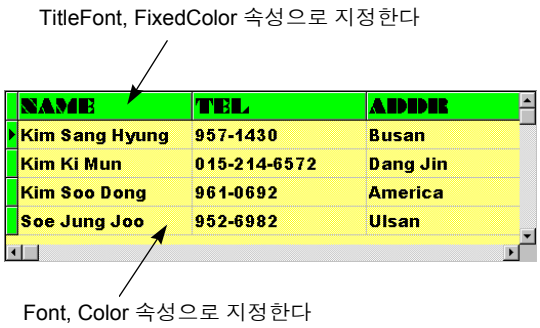
**그림**

테이블 전체를 한눈에 보여주는 DBGrid

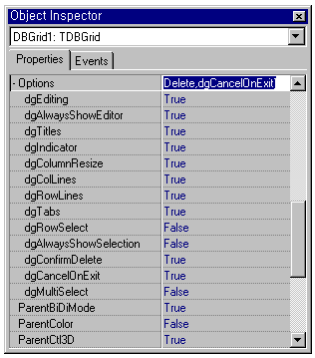


스프레드 시트와 같은 모양을 가지고 있으며 실행중에 마우스로 드래그하여 각 열의 크기나 위치를 옮길 수도 있다. 또한 각 셀을 클릭하여 선택한 후 키보드로 직접 수정할 수 있으며 Ins 키로 레코드를 삽입하거나 Ctrl+Del 키로 레코드를 제거할 수도 있다. 많은 양의 레코드를 한꺼번에 볼 수 있어 데이터를 살펴보기에는 아주 적당한 컨트롤이다. 보다시피 DBGrid는 다른 여타의 컨트롤에 비해 많은 정보를 보여주기 때문에 관심을 가질만한 속성들이 몇 가지 있다.

아주 일반적인 속성으로 Align, BorderStyle, Hint 등의 속성들이 있는데 이 속성들은 다른 컴포넌트의 경우와 동일하다. 또한 모양을 결정하는 두 개의 중요한 속성으로 Font, Color 속성이 있는데 이 또한 아주 일반적이므로 쉽게 이해가 갈 것이다. 그런데 그리드의 Font, Color 속성은 가운데 부분, 즉 데이터를 보여주는 부분에 대해서만 적용됨을 유의하자. 위쪽의 회색 부분인 타이틀 부분과 왼쪽의 인디케이터 부분의 글꼴, 색상은 TitleFont, FixedColor 속성으로 지정한다.




Options 속성에는 13개의 세부 속성이 있는데 이 속성들은 그리드의 외형과 동작에 영향을 주는 진위형 속성들이다.

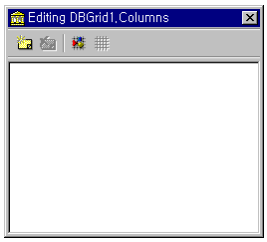


각 세부 속성은 표로 간략하게 요약하였다. 한번씩 속성들을 변경해 보면 의미를 쉽게 알 수 있을 것이다.

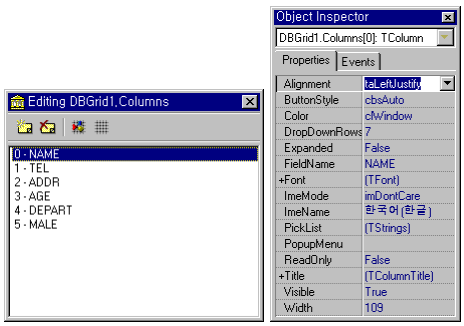
| 속성 | 설명 |
|---|---|
| dgEditing | 그리드를 사용하여 데이터를 직접 편집할 수 있도록 한다. dgRowSelect 속성이 선택되어 있으면 이 속성은 무시된다. |
| dgAlwaysShowEditor | 사용자가 Enter키나 F2키를 누르지 않아도 마우스로 선택하기만 하면 편집할 수 있도록 해 준다. |
| dgTitles | 상단에 필드명을 출력하는 타이틀을 표시한다. |
| dgIndicator | 왼쪽에 현재 레코드를 표시하는 인디케이터를 표시한다. |
| dgColumnResize | 실행중에 컬럼의 크기를 마우스로 변경하거나 위치를 옮길 수 있도록 한다. |
| dgColLine | 컬럼 사이에 수직 구분선을 그린다. |
| dgRowLines | 레코드 사이에 수평 구분선을 그린다. |
| dgTabs | Tab키와 Shift+Tab키를 사용하여 필드 사이를 이동할 수 있도록 한다. |
| dgRowSelect | 한 줄 단위로 선택되도록 한다. 이 속성이 선택되어 있으면 dgEditing의 설정은 무시되며 편집은 금지된다. |
| dgAlwaysShowSelection | 그리드가 포커스를 가지고 있지 않더라도 현재 선택된 셀을 반전하여 보여 주도록 한다. |
| dgConfirmDelete | Ctrl+Del 키를 눌러 레코드를 삭제할 때 확인 메시지를 보여주도록 한다. |
| dgCancelOnExit | 레코드를 삽입만 한 상태에서 수정은 하지 않고 그리드를 빠져나갈 때 삽입을 취소하도록 한다. 이 속성을 선택하면 부적절하게 레코드가 삽입되는 것을 방지할 수 있다. |
| dgMultiSelect | 한번에 여러 개의 줄을 선택할 수 있도록 한다. |

그리드는 여러 개의 컬럼으로 구성되어 있는데 각 컬럼에 개별적으로 속성을 설정할 수도 있다. 사실 그리드의 각 열은 TColumn이라는 별도의 컴포넌트들이며 따라서 별도로 속성을 조정할 수 있다. 과연 그런지 그리드를 더블클릭해 보자. 다음과 같은 컬럼 편집기가 열릴 것이다.

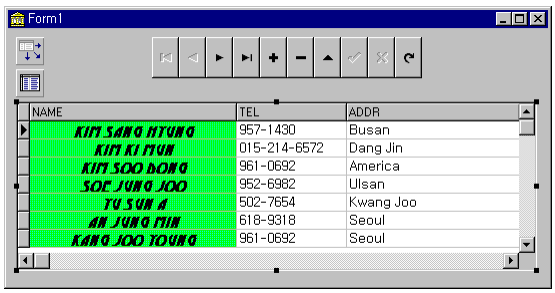
그림
컬럼 편집기




이 화면에서 세 번째 버튼인  를 누르든가 아니면 팝업 메뉴에서 Add All Fields 항목을 선택하면 그리드의 각 컬럼이 나타난다. 이 컬럼들은 TColumn이라는 컴포넌트이며 DBGrid에 Columns라는 배열 형태로 정의되어 있다. 컬럼 편집기에서 컬럼을 선택하면 오브젝트 인스펙터에는 선택된 컬럼의 속성이 나타난다.

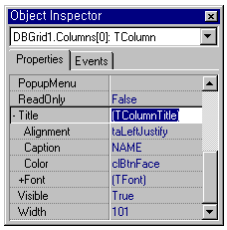



그럼 컬럼의 속성에 대해 알아보자. 일단 직관적으로 이해할 수 있는 속성으로 Width, Alignment, Color, Font 속성이 있는데 이 속성들을 변경하면 개별 컬럼에 다른 폰트나 글꼴을 쓸 수 있으며 정렬 상태나 폭을 디자인중에 마음대로 변경 가능하다. 다음은 NAME 컬럼에 대해서만 글꼴, 폰트를 변경하고 폭을 좀 늘린 후 중앙 정렬해 본 것이다.



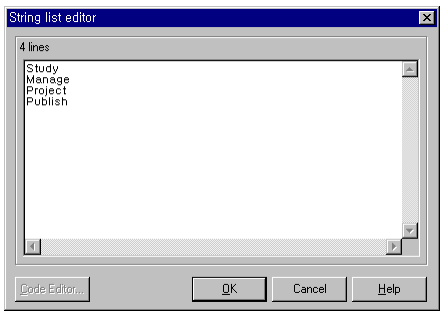

물론 다른 필드들도 얼마든지 속성을 변경하여 다양한 모양으로 만들 수 있

다. 컬럼 에디터에 컬럼을 포함시킨 상태에서는 그리드에서 경계를 드래그하여 폭은 자유롭게 조정 가능하다. Title 속성은 그리드의 타이틀에 대한 속성을 지정하며 세 가지 세부 속성을 가지고 있다.

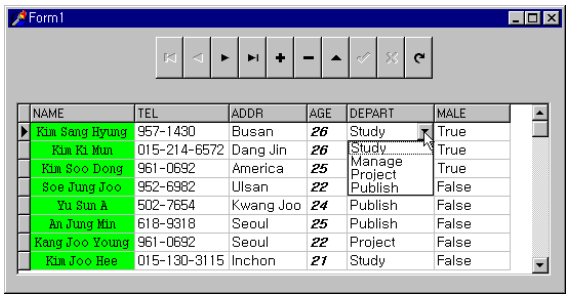



각각 타이틀의 정렬, 캡션, 색상을 지정한다. 디폴트로 타이틀의 캡션은 필드의 이름과 같은 NAME, TEL 등이지만 캡션을 변경하면 좀 더 알아보기 쉽게 '이름', '전화번호' 등과 같이 표기할 수도 있다.

컬럼의 속성중에 재미있는 것은 PickList 속성이다. 이 속성은 사용자가 선택할 수 있는 문자열 목록을 제공해 주어 마치 라디오 그룹에서 값을 선택하는 것처럼 값을 입력할 수 있게 해 준다. 그리드는 모든 필드의 값들을 문자열 형태로 보여주기 때문에 항상 키보드로 입력해야 하지만 이 속성을 사용하면 마우스로도 값을 입력할 수 있게 된다. DEPART 컬럼을 선택한 후 오브젝트 인스펙터에서 PickList를 더블클릭해 보자. 문자열 편집기가 열리는데 여기에 다음과 같이 입력한다.




그리고 예제를 실행해 보면 DEPART 컬럼이 콤보 박스 형식으로 바뀌며 목록중 하나를 선택할 수 있게 된다. 각 컬럼의 속성들을 변경하여 다양한 색상과 글꼴을 사용하도록 해 보았다.




# 바. 이미지 처리

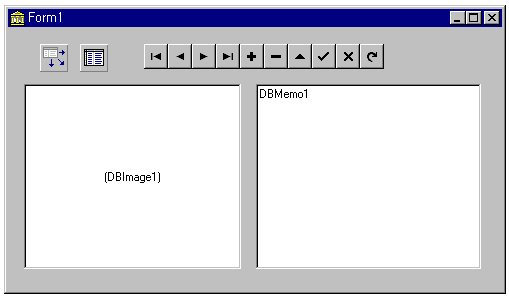
15jang
dbimg

과거의 데이터 베이스 프로그램, 특히 도스상에서 실행되는 데이터 베이스 프로그램들이 다룰 수 있는 자료는 문자열, 날짜, 논리, 수치 등의 짧은 텍스트에 불과했다. 그러나 윈도우즈와 같은 GUI 운영체제의 등장으로 이제는 테이블 안에 사진을 넣거나 길이가 긴 메모를 포함시킬 수 있게 되었으며 심지어는 사람의 목소리나 움직이는 모습까지도 넣을 수 있게 되었다.

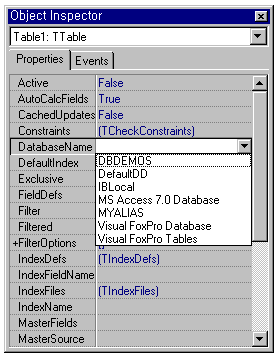
델파이로 테이블 안의 사진(또는 그림)을 출력하려면 DBImage 컴포넌트를 사용하며 이 컴포넌트도 DBEdit나 DBText와 마찬가지 방법으로 테이블에 연결한다. 즉 DataSource 속성과 DataField 속성을 사용한다. DBImage를 폼에 배치하고 적당히 크기를 조정한 후 연결만 해 주면 그림이 출력될 것이다. 표준 이미지 컴포넌트에 데이터 인식 능력만 추가한 것이므로 Stretch, Center 등의 속성도 물론 사용할 수 있다.

길이가 긴 문장을 출력하려면 DBMemo 컴포넌트를 사용한다. 연결하는 방법은 다른 데이터 컨트롤 컴포넌트와 동일하며 그 외 특성은 표준 메모 컴포넌트와 동일하다. 물론 DBImage나 DBMemo를 사용하려면 테이블에 이미지 필드와 메모 필드가 있어야 한다.

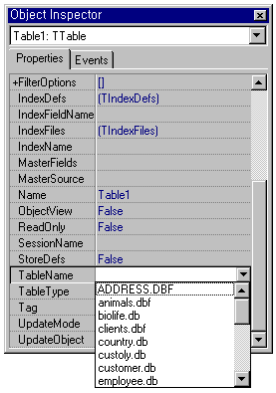
이 두 컴포넌트를 사용하여 간단한 예제를 만들어 보도록 하자. 이미지와 메모가 있는 테이블을 만드는 것은 그리 간단하지 않으므로 델파이와 함께 제공되는 BIOLIFE.DB 파일을 대신 사용하기로 한다. 새 프로젝트를 시작하고 폼에 다음과 같이 컴포넌트를 배치한다.




이제 컴포넌트의 속성들을 사용하여 상호 연결시켜 주는 작업을 해야 하는데 제일 먼저 테이블 컴포넌트를 선택한 후 DatabaseName 속성을 설정하도록 하자. 오브젝트 인스펙터에서 이 속성을 선택한 후 콤보 박스를 열면 다음과 같이 선택 가능한 여러 가지 데이터 베이스 목록이 나타날 것이다.



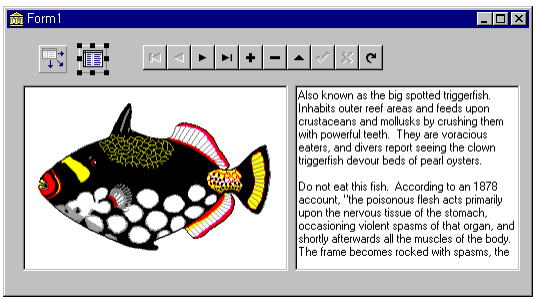

DatabaseName 속성은 디렉토리명 또는 앨리어스명을 지정하는데 ADDR 예제에서는 디렉토리명을 직접 주었지만 이번에는 앨리어스를 사용해 보자. 앨리어스란 디렉토리에 주어지는 일종의 별명을 말하며 잠시 후에 상세히 알아볼 것이다. 일단 DatabaseName 목록의 제일 위에 있는 DBDEMOS를 선택한다. 다음은 TableName 속성을 선택하되 이번에도 직접 입력할 필요없이 목록에서 선택하면 된다.




DBDEMOS에 포함된 테이블의 목록이 나타나는데 여기서 BIOLIFE.DB를 선택한다. 이제 Table1 컴포넌트는 BIOLIFE.DB에 연결된 것이다. 나머지 컴포넌트의 속성을 다음 표대로 설정한다.

| 속성 | 값 |
|---|---|
| DataSource1.DataSet | Table1 |
| DBNavigator1.DataSource | DataSource1 |
| DBImage1.DataSource | DataSource1 |
| DBImage1.DataField | Graphic |
| DBMemo1.DataSource | DataSource1 |
| DBMeno1.DataField | Notes |

마지막으로 Table1을 선택한 후 Active 속성을 True로 바꾸어 보면 이미지와 메모가 나타날 것이다.




실행시켜 보면 여러 가지 모양의 물고기 사진과 설명을 볼 수 있다. 실행중에

데이터를 편집하는 것도 물론 가능하다. 메모의 경우 키보드로 수정하면 되고 이미지는 클립보드를 사용하여 붙이기를 하면 된다. 이미지와 메모를 사용한다 뿐이지 사실 ADDR 예제와 크게 틀리는 바는 없다.

# 사. 테이블 컴포넌트

지금까지의 실습에서 Table 컴포넌트를 계속적으로 사용해 왔다. DatabaseName, TableName 두 속성으로 테이블을 연결해 주는 것 외에는 특별히 신경쓸 것도 없으므로 비교적 간단하게 이해할 수 있을 것이다. 그런데 이 컴포넌트도 좀 연구해 보면 꽤 복잡하며 잘 사용하면 여러 가지 복잡한 일들을 할 수 있다는 것을 알 수 있다. Table 컴포넌트 자체에 대해 깊이있게 연구해 보도록 하자.

## ■ 속성

우선 중요한 속성으로는 테이블과 연결하는데 사용되는 DatabaseName, TableName이 있고 테이블을 활성화시키는 Active 속성이 있다. 이 세 속성에 대해서는 앞에서도 설명했고 실습도 많이 해 보았으므로 이미 이해하고 있을 것이다. 그외 나머지 속성들에 대해 알아보자.

**✍ Exclusive**

이 속성은 네트웍 환경에서 테이블의 배타적 사용권을 통제한다. 로컬 환경에서는 이 속성에 대해 고려하지 않아도 된다. 디폴트값이 False로 되어 있기 때문에 한 테이블을 네트웍상의 여러 프로그램 또는 여러 사람이 동시에 사용할 수 있는데 이 속성을 True로 바꾸어주면 혼자서만 테이블을 사용할 수 있게 된다. 디자인시에는 이 속성을 바꾸지 않는 것이 좋으며 실행중에 Active 속성을 False로 설정한 후 변경하는 것이 좋다.

**✍ ReadOnly**

테이블을 읽기 전용으로 연다. 디폴트로 이 속성은 False이기 때문에 실행중에 테이블 내용을 얼마든지 수정할 수 있지만 이 속성을 True로 변경하면 데이터를 조회해볼 수만 있고 편집할 수는 없다. 과연 그런지 ADDR 예제나 DBGRID 예제를 열어 시험해 보기 바란다. 그런데 이 속성이 False여도, 즉 읽기 전용이 아니라도 편집이 불가능한 경우가 있는데 사용자가 테이블을 수정할 권한이 없는 경우가 이에 해당된다.

### ✍ CanModify

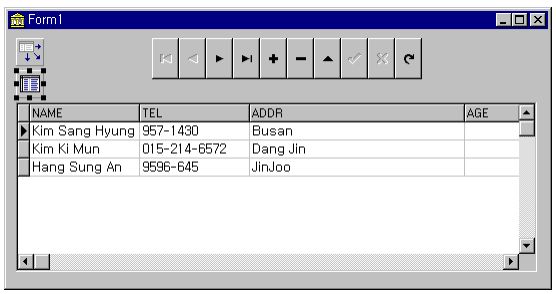
실행중에만 사용할 수 있는 속성이며 사용자가 데이터를 수정할 수 있는 상태인지를 조사한다. 이 값이 True이면 수정 가능하고 False이면 수정이 불가능하다. ReadOnly 속성이 True로 되어 있으면 이 속성은 False가 되며 다른 사용자가 테이블을 배타적으로 열었을 때도 이 속성은 False가 된다.

### ✍ Filtered

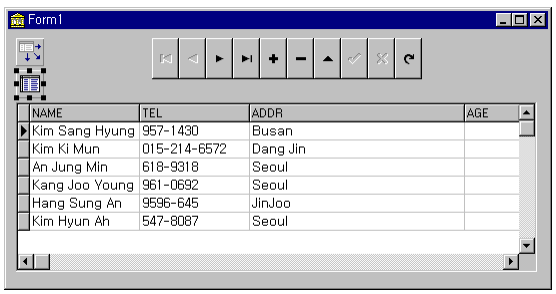
Filter 속성을 사용하여 일정 조건에 맞는 레코드만 추출해 내도록 한다. 이 속성이 True이면 필터링을 하고 False이면 필터링이 무시된다. 디폴트값은 False이므로 Filter 속성의 조건이 무시된다.

### ✍ Filter

레코드를 검색할 조건을 입력한다. 테이블은 이 조건에 맞는 레코드들만 검색해 낸다. 단 이 속성으로 필터링을 하려면 Filtered 속성이 True로 설정되어 있어야 한다. 테스트해 보려면 DBGrid 예제를 열어서 이 속성을 설정해 보면 된다. Filtered 속성을 True로 바꾼 후에 Filter에 AGE=26이라고 조건식을 입력해 보자. 그러면 나이가 26세인 사람들의 레코드만 나타날 것이다.




and, or 등의 논리 연산자를 사용하면 두 개 이상의 조건을 결합시킬 수도 있다. AGE=26 or ADDR='Seoul'이라고 입력하면 나이가 26세이거나 주소가 Seoul인 사람들만 검색된다.



이 속성은 실행 중에도 마음대로 변경할 수 있다.

✍ FilterOptions

필터링 옵션을 지정하는 두 가지 세부 속성을 가지고 있다. 디폴트로 두 옵션은 모두 선택되어 있지 않다.

| 옵션 | 설명 |
|---|---|
| foCaseInsensitive | 문자열의 대소문자를 구분하지 않는다. 즉 ADDR='Seoul', ADDR ='SEOUL', ADDR ='seoul' 어떤 조건을 쓰나 동일한 결과가 나타난다. |
| foNoPartialCompare | 부분 문자열 비교를 허가하지 않는다. 문자열 내부에서 *는 와일드 캐릭터로 해석되므로 NAME='Kim*' 조건을 주면 성이 김씨인 모든 사람이 검색된다. 이 옵션이 선택되어 있으면 *는 문자 그대로 해석된다. |

✍ State

테이블의 현재 상태를 나타내며 이는 테이블로 현재 어떤 작업이 가능한가를 나타낸다. 테이블의 상태는 실행중에 사용자가 데이터를 조작함에 따라 수시로 변경되며 다음 상태중 하나가 된다.

| 상태 | 설명 |
|---|---|
| sdInactive | 테이블이 닫혀진 상태이며 따라서 데이터를 사용할 수 없다. |
| dsBrowse | 조회는 가능하지만 편집은 할 수 없는 상태이며 테이블을 열면 이 상태가 된다. |
| dsEdit | 현재 레코드를 편집할 수 있는 상태. DataSource의 AutoEdit 속성이 True일 경우 자동으로 편집 상태가 되며 Edit 메소드에 의해 강제로 편집 모드로 진입할 수 있다. |
| dsSetKey | 검색이 가능한 상태이며 편집이나 삽입은 불가능하다. |
| dsCalcFields | 필드 계산이 진행중인 상태이다. |
| dsFilter | OnFilterRecord 이벤트에서 필터링을 하고 있는 상태이며 편집은 불가능하다. |
| dsNewValue | TField.CurValue가 액세스되고 있는 중의 임시 상태 |
| dsOldValue | TField.OldValue가 액세스되고 있는 중의 임시 상태 |
| dsCurValue | TField.CurValue가 액세스되고 있는 중의 임시 상태 |

| | |
|---|---|
| dsBlockRead | Next 메소드 호출 후 데이터 인식 컨트롤이 아직 갱신되지 않은 상태 |
| dsInternalCalc | 임시 상태 |

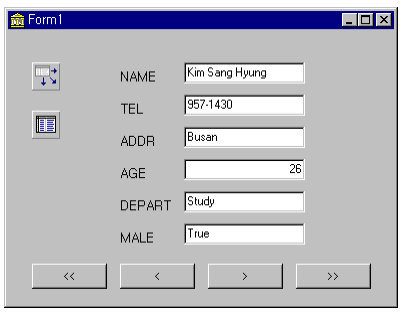
## ■ 이동 메소드

앞서의 예제들에서 테이블의 레코드 사이를 이동하기 위해 네비게이터 컴포넌트를 사용하였다. 코드로 레코드를 이동시킬 때는 다음 메소드를 사용한다.

| 메소드 | 설명 |
|---|---|
| First | 첫 번째 레코드로 이동한다. |
| Prior | 앞의 레코드로 이동한다. |
| Next | 다음 레코드로 이동한다. |
| Last | 마지막 레코드로 이동한다. |
| MoveBy | 지정한 수만큼 이동한다. MoveBy(8)은 현재 레코드로부터 8번째 레코드로 이동한다. 앞쪽으로 이동할 때는 음수값을 주면 된다. |

15jang
DbMove

현재 레코드의 위치와 관련된 두 가지 속성이 있는데 BOF는 현재 레코드가 첫 번째 레코드이면 True가 되며 EOF는 현재 레코드가 마지막 레코드이면 True가 된다. 이 속성과 이동 메소드를 사용하면 네비게이터를 사용하지 않고도 레코드 사이를 이동할 수 있다. ADDR 프로젝트를 복사해 와 DbMove 프로젝트를 만들고 네비게이터 대신 네 개의 버튼을 배치해 보았다.




Name 속성은 디폴트를 사용하고 Caption 속성만 변경해 주었다. 각 버튼의 OnClick 이벤트 핸들러는 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Table1.First;
 Button3.Enabled:=True;
 Button4.Enabled:=True;
 if Table1.BOF then
   begin
   Button1.Enabled:=False;
   Button2.Enabled:=False;
   end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 Table1.Prior;
 Button3.Enabled:=True;
 Button4.Enabled:=True;
 if Table1.BOF then
   begin
   Button1.Enabled:=False;
   Button2.Enabled:=False;
   end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
 Table1.Next;
 Button1.Enabled:=True;
 Button2.Enabled:=True;
 if Table1.EOF then
   begin
   Button3.Enabled:=False;
   Button4.Enabled:=False;
   end;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
 Table1.Last;
 Button1.Enabled:=True;
 Button2.Enabled:=True;
 if Table1.EOF then
   begin
   Button3.Enabled:=False;
   Button4.Enabled:=False;
```

```
    end;
end;
```

First, Prior, Next, Last 메소드를 호출하여 레코드를 이동하였다. 나머지 코드는 현재 레코드의 위치에 따라 버튼을 사용금지시키는 코드들이다. 예제를 실행해 보면 버튼으로 레코드를 변경할 수 있을 것이다.

## ■ 검색 메소드

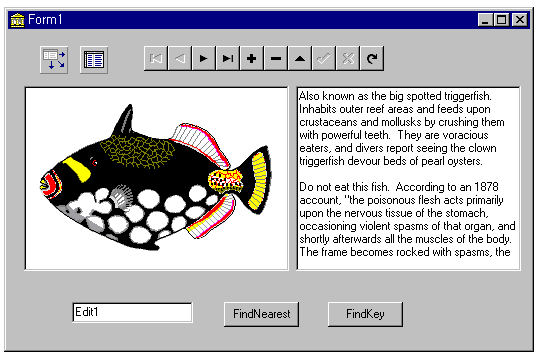
데이터 베이스의 기능 중에 조건에 맞는 특정한 레코드를 찾아내는 검색 기능은 필수적이다. 검색은 주로 쿼리에 의해 이루어지지만 테이블의 검색 메소드로도 가능하다. 테이블로 검색을 하기 위한 전제조건은 반드시 인덱스가 작성되어 있어야 한다는 것이다. 인덱스(Index)란 테이블의 레코드를 액세스하는 순서를 지정하는 보조 정보이다.

```
function FindKey(const KeyValues: array of const): Boolean;
procedure FindNearest(const KeyValues: array of const);
```

두 함수 모두 인덱스 키가 KeyValue로 지정한 값과 일치하는 레코드를 찾아 이동해 준다. 차이점이라면 FindKey 함수는 일치하는 레코드가 발견되지 않을 경우 False를 리턴하지만 FindNearest 함수는 최대한 비슷한 레코드라도 찾아준다는 점이다.

15jang
Find

우리가 만든 ADDRESS.DBF에는 인덱스가 정의되어 있지 않으므로 델파이의 데모 DB인 BIOLIFE.DB를 사용하여 검색 메소드를 실습해 보도록 하자. DbImg 예제를 열고 아래쪽에 에디트 하나 버튼 두 개를 배치하고 캡션을 조정하였다.



그리고 각 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.FindNearest([Edit1.Text]);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  if Table1.FindKey([Edit1.Text])=False then
    ShowMessage('Not Found');
end;
```

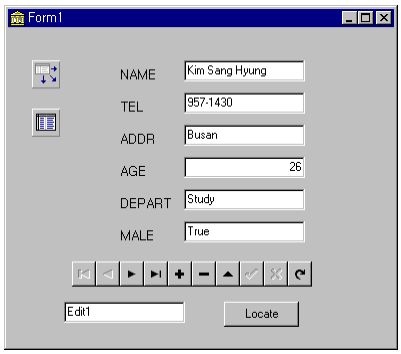
BIOLIFE.DB 파일은 Species No 필드에 인덱스가 설정되어 있다. 프로젝트를 실행한 후 에디트에 90020~90310까지의 값을 넣은 후 버튼을 눌러주면 Species No 필드가 에디트에 입력한 값과 같은 레코드를 찾아 이동해 줄 것이다.예를 들어 90170을 입력하면 상어가 출력된다.

FindKey, FindNearest 메소드는 인덱스가 있는 테이블에만 쓸 수 있는 레코드인데 반해 다음 메소드는 인덱스가 없어도 사용할 수 있다.

```
function Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions):
Boolean;
```

15jang
Locate

첫 번째 인수로 검색 조건이 될 필드명을 주고 두 번째 인수로 원하는 필드값을 주면 된다. 이 메소드는 지정한 필드가 인덱스로 설정되어 있으면 인덱스를 사용하고 그렇지 않으면 순차 검색을 하여 원하는 레코드를 찾아준다. 세 번째 인수는 대소문자 구분, 부분 검색 등의 옵션을 지정한다. ADDR 프로젝트를 수정하여 이 메소드를 사용하는 검색 예제를 만들어 보자. 폼의 아래쪽에 검색 조건을 입력할 에디트 하나와 검색시작을 명령할 버튼을 하나 배치한다.




그리고 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if Table1.Locate('NAME',Edit1.Text,[loCaseInsensitive])
 =False then
   ShowMessage('Not Found');
end;
```

NAME 필드가 에디트에 입력한 문자열과 일치하는 레코드를 찾되 대소문자는 구분하지 않도록 하였다.

# 아. BDE 관리자

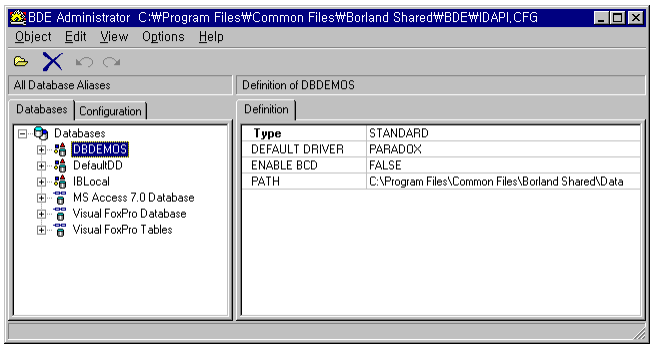
여기까지 실습을 하면서 이 책에서는 한글을 한 번도 사용하지 않았다. 사람 이름도 Kim Sang Hyung으로 표시하고 지명도 Busan, Seoul 등과 같이 영어를 사용했다. 한글을 쓰면 읽기도 쉽고 쓰기도 쉬운데, 왜 한글을 사용하지 못하는가 하면 델파이가 미국에서 만든 미제이므로 한글을 사용할 경우 문제가 발생할 소지가 있기 때문이다. 그렇다고 해서 델파이에서 한글을 전혀 쓰지 못할리는 없고 몇 가지 조정을 해 주어야 하는데 다행히 델파이 4는 디폴트 셋업대로 사용하면 별 문제없이 한글을 사용할 수 있다.

데이터 베이스 프로그램은 또한 디렉토리 문제가 복잡하다. ADDR.EXE는 ADDRESS.DBF라는 데이터 베이스 파일을 사용하며 이 파일을 C:₩ 디렉토리에 두었다. 그리고 Table 컴포넌트의 DatabaseName 속성에는 이 디렉토리 이름을 기입해 주어 DB 파일이 있는 위치를 알려준다. 문제는 DB 파일의 위치는 반드시 절대 경로로 지정되어야 한다는 점이다.

ADDR.EXE를 C:₩DBEX 디렉토리에서 개발을 했고 이 디렉토리에 DB파일을 같이 두었다고 해 보자. 상업적으로 판매되는 프로그램일 경우 이 프로그램을 구입한 사용자가 반드시 C:₩DBEX에 설치한다는 보장이 없으며 디렉토리가 바뀌면 프로그램은 DB 파일을 찾지 못해 제대로 동작할 수가 없다. 그렇다고 소스 파일을 제공하여 사용자가 직접 Table1.DatabaseName 속성을 수정하여 다시 컴파일하라고 할 수도 없다. 이 문제를 해결하기 위해 델파이는 앨리어스(Alias)를 사용한다. 앨리어스란 디렉토리 경로에 붙여진 별명이며 사용자가 마음대로 수정할 수 있다.

앞서 제시한 언어 문제와 앨리어스 지정에 관한 문제들은 델파이와 함께 제공되는 BDE 관리자라는 프로그램을 사용하여 조정한다. 아래아 한글의 CONFIG.EXE나 이야기의 BARAM.EXE와 하는 일이 같다고 보면 틀림없다. 디폴트대로 설치되었다면 이 프로그램은 Program Files₩Common Files₩Borland Shared₩Bde₩Bdeadmin.exe에 있으며 델파이 그룹에  이런 아이콘으로 존재한다. 실행시의 모습은 다음과 같다.

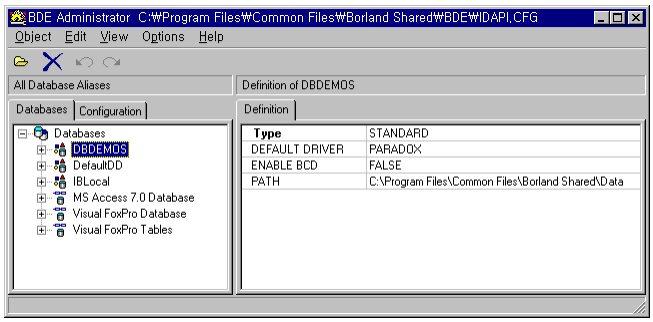
그림
데이터 베이스 환경 설정 프로그램인 BDE Administrator




이 프로그램에서 언어 드라이버를 변경하거나 설치하며 앨리어스를 새로 작성하며 기타 날짜나 시간의 출력 양식을 지정한다. 좌우로 페인이 나누어져 있는데 왼쪽에서 설정 대상을 선택하고 오른쪽에서 설정 대상을 변경한다. 또한 왼쪽에는 두 개의 탭이 있는데 Databases 탭에서는 앨리어스 설정을 하며 Configuration 탭에서는 드라이버와 BDE 환경을 설정한다.

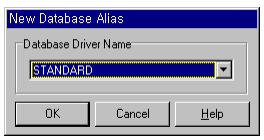
## ■ 앨리어스의 설정

Databases 탭에서 앨리어스 설정을 한다. 이 페이지를 열면 몇 개의 앨리어스가 정의되어 있음을 볼 수 있다.




이중 DBDEMOS 앨리어스는 델파이와 함께 제공되는 데이터 베이스 예제를 위해 설정된 앨리어스인데 실제 디렉토리는 C:₩Program Files₩Common Files₩Borland Shared₩Data로 되어 있다. 이 디렉토리로 이동해 보면 다수의 DB 파일이 있음을 알 수 있다. 델파이가 제공하는 DB 예제의 Table이나 Query 컴포넌트의 DatabaseName 속성이 DBDEMOS로 설정되어 있어 이 테이블들을 사용할 수 있게 된다.

새로운 앨리어스를 정의하려면 Object/New 항목을 선택하거나 팝업 메뉴에서 New를 선택한다. 다음과 같은 대화상자를 보여주며 앨리어스의 유형을 묻는다.




디폴트로 주어진 Standard를 선택하면 STANDARD1이라는 앨리어스를 만들어 줄 것이다.이 이름은 왼쪽 페인의 트리에서 직접 변경할 수 있는데 MYALIAS로 변경해 보도록 하자. 앨리어스의 이름은 다른 앨리어스와 중복되지만 않는다면 마음대로 정할 수 있다. 앨리어스를 만든 후 오른쪽에서 디폴트 드라이버, 경로 등을 지정해 준다. PATH란에 지정하고 싶은 디렉토리(예:C:₩DBEX)를 입력해 주고 프로그래밍할 때는 Table의 DatabaseName 속성에 C:₩DBEX 등과 같이 디렉토리를 바로 쓰지 말고 앨리어스를 사용하도록 한다.

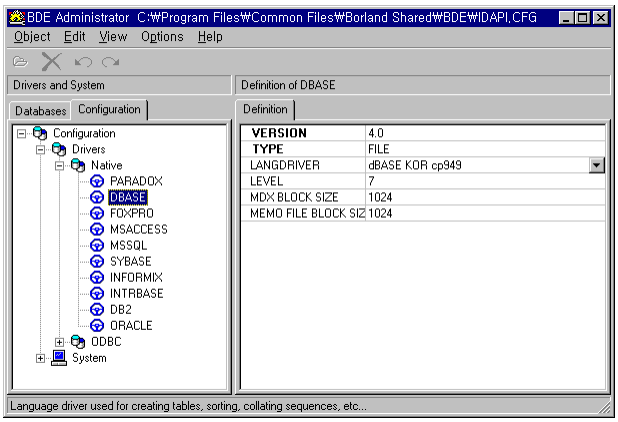
참고하세요

이 책의 모든 예제는 델파이가 예제용으로 만들어 놓은 DBDEMOS 앨리어스를 사용하여 컴파일되어 있다. 따라서 이 예제들이 제대로 컴파일 되게 하려면 예제에서 사용하는 모든 DB 파일을 DBDEMOS가 지정하는 디렉토리로 복사해 두어야 한다. 15jang 디렉토리

아래에 DB 파일을 넣어 두었으므로 이 파일들을 DBDEMOS로 복사하도록 하자. 귀찮다는 생각이 들겠지만 이렇게 하지 않으면 소스를 수정해야만 컴파일해 볼 수 있게 되므로 더 귀찮아진다.

## ■ 드라이버 설정

BDE 관리자의 Configuration 탭에서는 각 데이터 베이스의 드라이버에 대한 설정을 한다. Drivers 가지를 확장하면 Native 드라이버와 ODBC 드라이버가 있으며 이 가지들을 확장하면 현재 BDE에 설치된 드라이버 목록이 나타난다. 목록중 하나를 선택하면 해당 드라이버의 설정 상태가 오른쪽에 나타날 것이다.




설정할 수 있는 항목은 드라이버별로 다르다. 이중 LANGDRIVER 항목은 모든 드라이버에 거의 공통적으로 존재하는데 이 드라이버가 사용할 기본 언어 드라이버이다. 언어 드라이버란 데이터 베이스 운용에 어떤 언어를 사용할 것인가를 지정하며 언어에 따라 정렬의 순서나 사용하는 문자가 달라진다. 최초 델파이를 설치하면 DBASE, PARADOX 등 주요 드라이버의 언어 드라이버가 한국어로 설정되어 있으므로 특별히 조정해 주어야 할 필요는 없다. 그래서 디폴트 설정대로 사용하면 한국어를 아무런 문제없이 사용할 수 있다. 물론 일본어나 프랑스어등을 사용한다면 언어 드라이버를 바꿔 주어야 한다.

Configuration/System에서는 날짜나 시간, 수치값의 출력 포맷을 설정하는데 자세한 방법은 도움말을 참조하기 바란다. 별로 바꿀 일도 없겠지만 설사 바꿀 일이 있다고 해도 어렵지 않게 바꿀 수 있다.

# 15-3 쿼리와 SQL

## 가. Query

아주 간단한 데이터 베이스 프로그램은 Table 컴포넌트를 사용하면 되지만 좀 더 복잡한 조작을 해야 할 경우는 Query 컴포넌트를 사용해야 한다. Query는 Table이 제공하지 못하는 부가적인 몇 가지 기능들을 더 제공하는데 가장 두드러지는 차이로 SQL 언어를 지원한다는 점을 들 수 있다. Query의 DatabaseName 속성에는 SQL 언어로 검색할 데이터 베이스를 지정하며 다음 세 가지 중 한 가지 값을 준다.

① BDE 에서 정의한 앨리어스명
② DB 파일이 있는 디렉토리명
③ Database 컴포넌트가 있을 경우 이 컴포넌트가 정의한 앨리어스명

Query는 TableName이라는 속성이 없어 테이블 이름을 별도로 지정하지 않으며 액세스하고자 하는 테이블의 이름은 SQL 명령 내에서 지정한다. Query에 SQL 명령을 작성할 때는 SQL 속성을 사용하며 실행시와 디자인시의 속성 설정 방법이 다르다.

**■ 디자인시**

오브젝트 인스펙터의 SQL 속성을 더블클릭하면 문자열 리스트 편집기가 열린다. 여기에 SQL 명령들을 기입해 주면 된다. TStrings 속성이므로 여러 줄로 SQL명령을 기술할 수 있다. 디자인시에 SQL 명령을 실행시키려면 Query의 Active 속성을 True로 설정한다.

**■ 실행시**

실행중에 SQL 명령을 작성하려면 다음 절차를 따라야 한다.

① Query 를 Close 한다. 쿼리를 열어놓은 채로 SQL 속성을 변경할 수는 없다.
② 기존의 SQL 명령을 삭제하기 위해 Clear 메소드로 지워준다.
③ SQL 의 Add 메소드로 SQL 명령을 기입한다. 기입되는 명령은 문자열 형태를 가지며

복잡한 명령을 입력해야 할 경우는 문자열 가공 함수로 조립해야 한다.

④ Query 의 Open 메소드로 SQL 명령을 실행한다.

그래서 SQL문을 입력하는 코드는 다음과 같이 순서가 정해져 있다. 모두 Query1의 속성이나 메소드이므로 우측과 같이 with로 Query1을 묶어내는 것이 더 편하다.

```
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add(SQL 명령);
Query1.Open;
```

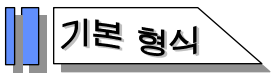
```
with Query1 do
 begin
  Close;
  SQL.Clear;
  SQL.Add(SQL 명령);
  Open;
 end;
```

SQL명령이 수행되면 수행 결과는 DataSource 컴포넌트를 거쳐 데이터 컨트롤 컴포넌트로 전달되어 사용자에게 보여지게 된다.

# 나. SQL 언어

SQL 은 철자 그대로 "에스큐엘"이라고 읽는다.

SQL 언어는 앞에서도 언급했듯이 산업 표준의 데이터 운용 언어이다. 데이터를 선택, 삽입, 삭제하는 데이터 관리 명령(DML)과 테이블을 만들고 변경하는 데이터 정의 명령(DDL)으로 크게 세분되어진다. 우선 테이블에서 데이터를 끄집어내는 SELECT 명령에 대해서 알아보기로 한다.




SELECT 필드 FROM "테이블"

필드는 검색 결과 나타날 필드의 이름이며 *를 사용하면 테이블에 있는 전체 필드를 보여준다. 테이블은 검색 대상이 되는 테이블의 이름을 말하며 구체적으로 DB 파일의 파일명이다. 파일명은 홑따옴표나 겹따옴표로 반드시 싸주어야 하는데 SQL 명령어 자체가 홑따옴표로 표현되는 문자열이므로 파일 이름은 겹따옴표를 쓰는 것이 더 편리하다. 예를 들어 Select * From "Address.dbf" 명령

은 Address.dbf라는 테이블에서 모든 필드의 데이터를 추출한다는 뜻이다.

# 다. 검색 예제

## ■ Where문

SELECT문에 검색 조건을 지정하여 조건에 맞는 필드만을 읽어오고자 할 경우에는 WHERE문을 사용하며 WHERE 뒤에는 검색 조건이 온다. 검색 조건이란 주로 필드값을 상수와 비교하는 문장이며 =, <>, <, > 등의 연산자로 조건을 지정한다. 다음 예를 보자.

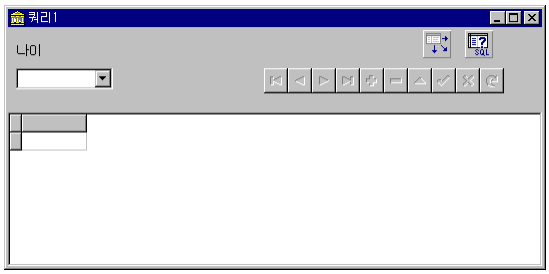
| | |
|---|---|
| WHERE AGE>20 | AGE 가 20 보다 큰 경우 |
| WHERE COUNTRY='KOREA' | Country 가 KOREA 인 경우 |
| WHERE WAGE<50 | WAGE 가 50 보다 작은 경우 |

15jang
query

TQuery와 SQL을 이용한 DB 검색 예제를 순서에 따라 작성해 보자. 사용하는 DB 파일은 앞에서 만들었던 ADDRESS.DBF이며 AGE 필드를 검색 조건으로 사용하기로 하고 조건의 입력은 콤보 박스를 사용한다.

1 일단 프로젝트를 시작한다. 그리고 ADDRESS.DBF 를 DBDEMOS 앨리어스 디렉토리로 복사해 둔다. 그래야 절대 경로를 쓰지 않고도 앨리어스를 통해 테이블에 접근할 수 있게 된다.

2 컴포넌트를 배치한다. DataSource, Query 와 데이터를 출력할 DBGrid, 레코드 이동을 위한 DBNavigator 를 배치한다. 그리고 검색 조건을 입력받을 콤보 박스 하나와 콤보 박스위에 "나이"라는 설명을 달기 위해 레이블을 배치하도록 하자. 단순히 문자열 선택에 사용하므로 Standard 페이지에 있는 콤보 박스면 되며 DBComboBox 가 아니므로 주의하기 바란다. 컴포넌트 배치 후의 폼은 다음과 같다.




3 컴포넌트의 속성을 다음과 같이 설정한다.

| 컴포넌트 | 속성 | 속성값 |
|---|---|---|
| 폼 | 캡션 | 쿼리1 |
| 콤보 박스 | Text | All |
| | Items | All,21,22,24,25,26,27 |
| Query | DatabaseName | DBDEMOS |
| DadaSource | DataSet | TQuery1 |
| DBGrid | DataSource | DataSource1 |
| DBNavigator | DataSource | DataSource1 |

Query1이 DataSource1에 연결되고 DataSource1에 그리드와 네비게이터가 연결되어 있다. 콤보 박스의 Items에 입력되어 있는 값의 목록이 나이를 선택할 조건이 된다. Query1.DataSource 속성에 DataSource1을 지정하지 않으므로 주의하기 바란다.

4 폼이 처음 생성될 때 그리드에 데이터를 공급해 주는 코드를 작성한다. FormCreate 이벤트에 다음 코드를 입력한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with Query1 do
  begin
  Close;
  SQL.Clear;
  SQL.Add('Select * from "address.dbf"');
  Open;
  end;
end;
```

쿼리에 새로운 SQL 명령을 입력하기 전에 이미 입력되어 있던 SQL 명령을 지워야 한다. 먼저 쿼리를 Close하고 SQL을 Clear한다. 그리고 SQL의 Add 메소드를 사용하여 새로운 SQL 명령을 입력하고 Open 메소드로 명령을 실행시킨다. 실행 결과 ADDRESS.DBF에서 모든 레코드를 검색하여 그리드에 출력하게 된다.

5 검색 코드를 작성한다.

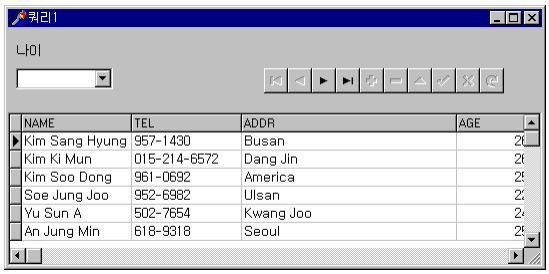
콤보 박스에서 검색 조건을 지정하면 검색을 실행하는 코드를 작성한다. 콤보 박스의 선택이 변경될 때 검색을 시작하므로 코드를 작성할 위치는 콤보 박스의 OnChange 이벤트 핸들러이다.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
with Query1 do
 begin
 Close;
 SQL.Clear;
 SQL.Add('Select * from "address.dbf"');
 if ComboBox1.ItemIndex<>0 then
  SQL.Add(' Where Age='+
  ComboBox1.Items[ComboBox1.ItemIndex]);
 Open;
 end;
end;
```

앞에서 작성한 SQL 명령과 같은 명령을 일단 작성해 넣고 뒤에 Where문을 추가하여 검색 조건을 지정한다. 이 코드에서 지정한 조건은 Age 필드가 콤보 박스에서 선택한 내용과 일치한다는 조건이다. 즉 콤보 박스에서 선택한 Age값과 같은 Age값을 가진 레코드만 그리드에 나타나게 된다.

SQL 명령은 결국 이런 식으로 문자열 조립에 의해 쿼리에 제공된다. 어떤 조건을 입력할 것인가, 조건을 어떤 방식으로 조립하느냐는 프로그래머가 마음대로 결정할 수 있으며 다만 SQL 문법에 맞는 문자열을 만들어 내기만 하면 된다.

6 저장 및 실행한다. 실행 결과는 다음과 같다.




실행하면 FormCreate 이벤트의 Select * From "address.dbf"가 실행되어 모든 필드가 DBGrid에 출력된다. 실행중에 사용자가 콤보 박스에서 나이를 선택하면 Where문의 조건이 변경되므로 조건에 맞는 필드만 검색해 낸다. 콤보 박스에서 선택된 값에 따라 어떤 SQL문이 만들어지는가를 보자.

| ComboBox1 | SQL |
|---|---|
| All | Select * From "address.dbf" |
| 22 | Select * From "address.dbf" where age=22 |
| 23 | Select * From "address.dbf" where age=23 |
| 24 | Select * From "address.dbf" where age=24 |

SQL 문법 자체는 아주 쉽게 이해가 갈 것이다. 조건을 어떤 방식으로 입력받아 문자열을 어떻게 조립하는가가 문제일뿐 SQL을 사용하는 방법 자체는 아주 쉽다.

질의를 통해 만들어진 결과 셋은 통상 읽기 전용이므로 볼 수만 있고 편집할 수는 없다. 왜냐하면 질의의 목적 자체가 조건에 맞는 레코드를 살펴보기 위한 것이기 때문이다. 위 예제를 실행하여 필드를 편집해 보면 편집이 안된다는 것을 알 수 있을 것이다. 만약 굳이 질의의 결과 셋을 편집하고자 한다면 쿼리의 RequestLive 속성을 True로 바꿔주면 된다. 이 속성이 True이면 쿼리는 편집 가능한 결과 셋(Live Data)을 넘겨주게 된다. 그러나 이 속성이 아무리 True이더라도 여러 개의 테이블로 추출한 경우는 편집할 수 없다.

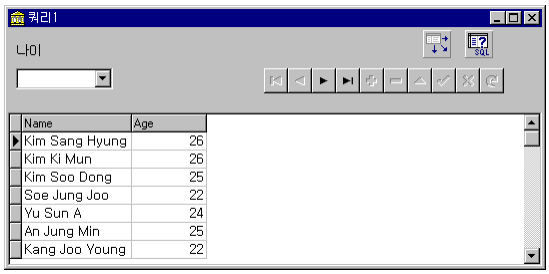
## ■ 일부 필드만 출력

Select문으로 레코드를 출력할 때 출력의 대상이 되는 필드 이름을 Select 문 바로 다음에 지정할 수 있다. 앞의 예에서는 *를 사용하여 모든 필드를 다 출력하도록 했지만 *자리에 원하는 필드 이름을 적어 주면 지정한 필드만 그리드에

나타난다. 다음 예를 보자.

| SQL | 출력되는 필드 |
|---|---|
| Select Name from "address.dbf" | Name 필드만 |
| Select Name,Age from "address.dbf" | Name, Age 필드만 |
| Select * from "address.dbf" | 모든 필드 |

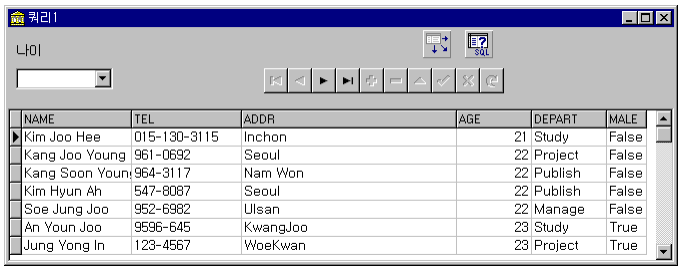
이런 문장은 코드를 작성할 필요도 없이 디자인시에 직접 실습해 볼 수 있다. Query1 컴포넌트를 선택하고 오브젝트 인스펙터에서 SQL 속성을 더블클릭하여 나타난 문자열 리스트 편집기에 Select Name,Age from "address.dbf" 문장을 직접 입력한다. 그리고 Query1의 Active 속성을 True로 바꾸면 즉각 결과를 볼 수 있다.




일일이 코드를 작성하고 컴파일을 시켜 보지 않아도 되므로 SQL을 처음 배울 때 실습하기 편리하다.

## ■ 정렬 순서 지정

그리드에 나타나는 레코드는 아무 지정이 없으면 테이블에 기록되어 있는 순서대로이다. 정렬 순서를 지정하면 기준 필드의 알파벳순으로 레코드가 정렬되어 출력된다. 정렬 순서를 지정하는 SQL문은 Order by문이며 뒤에 기준 필드가 될 필드 이름이 온다. 다음은 Select * From "address.dbf" Order by Age,Name의 결과이다.




Age 필드를 기준으로 오름차순 정렬되며 Age가 같을 경우 Name 필드를 기준으로 오름차순 정렬된다. 내림차순으로 정렬하고자 할 때는 뒤에 desc를 붙여주면 된다.

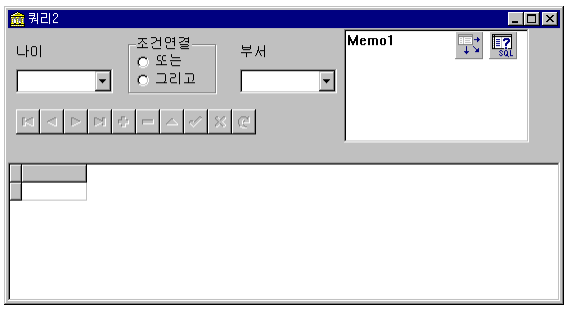
## ■ 조건 결합

15jang
query2

Where문 다음에 여러 개의 조건을 나열할 때는 AND, OR 등의 연산자를 사용한다. 예를 든다면 다음과 같은 Where문을 생각할 수 있다.

| Where문 | 의미 |
|---|---|
| Where Age<25 AND Age>22 | 22살 이상 25살 이하 |
| Where Age<25 OR Addr='Seoul' | 25살 이하이며 고향이 서울 |
| Where Addr='Seoul' AND Depart='Project' | 고향이 서울이고 부서가 총무부 |

SQL문은 이렇게 쉽지만 이런 조건을 실행중에 사용자로부터 입력받아서 문자열을 조립하기란 쉽지가 않다. 배포 CD에 있는 Query2 예제는 나이와 부서, 두 가지 조건을 연결하여 조건에 맞는 레코드를 검색해 낸다. 폼 배치는 다음과 같다.



나이와 부서를 입력받을 콤보 박스가 두 개 배치되어 있으며 콤보 박스의 Items 속성에는 가능한 값들이 이미 입력되어져 있다. 조건은 라디오 그룹을 사용하여 AND, OR 중 선택하도록 되어 있으며 검색 결과는 그리드로 출력된다. 우측의 메모는 실행중에 조립된 SQL문이 어떻게 되어 있는가를 보여주기 위해 일부러 집어 넣었다. 코드는 다음과 같다.

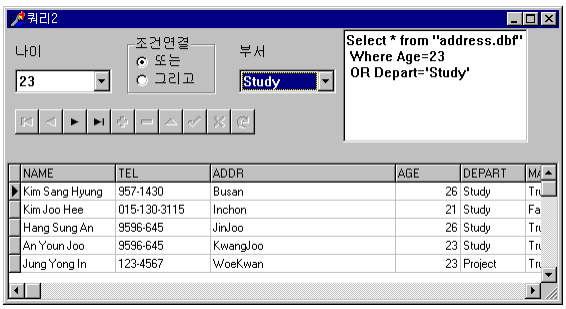
```
{폼이 처음 만들어질 때는 모든 레코드를 출력한다.}
procedure TForm1.FormCreate(Sender: TObject);
begin
with Query1 do
 begin
 Close;
 SQL.Clear;
 SQL.Add('Select * from "address.dbf"');
 Open;
 end;
ComboBox1.ItemIndex:=0;
ComboBox2.ItemIndex:=0;
RadioGroup1.ItemIndex:=0;
end;

{콤보 박스나 라디오 버튼이 바뀔 때마다 검색 조건을
다시 설정한다. 이 핸들러는 ComboBox1, ComboBox2,
RadioGroup1 의 OnChange 이벤트에 연결되어 있다.}
procedure TForm1.ComboBox1Change(Sender: TObject);
var
 andor:string;
begin
with Query1 do
 begin
 Close;
 SQL.Clear;
 SQL.Add('Select * from "address.dbf"');
 {나이 조건이 설정되어 있으면 조건 입력}
 if ComboBox1.ItemIndex<>0 then
  SQL.Add(' Where Age='+
  ComboBox1.Items[ComboBox1.ItemIndex]);

 {두 조건의 유무와 라디오 그룹의 상태에 따라
 적당한 접속사를 선택한다.}
 if RadioGroup1.ItemIndex=0 then andor:=' OR '
  else andor:=' AND ';
 if ComboBox1.ItemIndex=0 then andor:=' Where ';
```

```
{부서 조건이 설정되어 있으면 조건 입력}
if ComboBox2.ItemIndex<>0 then
  begin
  SQL.Add(andor+'Depart='''+
  ComboBox2.Items[ComboBox2.ItemIndex]+'''');
  end;
{조립된 SQL 문을 보여주기 위해 삽입된 문장이다.}
memo1.lines:=query1.sql;
Open;   {SQL 실행}
end;
end;
```

콤보 박스와 라디오 그룹에 입력된 사항을 이용하여 SQL문장을 조립해 낸다. 문자열을 조립하는 방법이 초보자에게는 다소 어려울 수도 있을 것이다. 특히 문자열을 나타내는 홑따옴표가 SQL문에 포함될 경우는 홑따옴표를 두 번 연속써 주어야 함을 주의하도록 하자. 실행중의 모습은 다음과 같다.

**그림**

두 개 이상의 조건을 동시에 설정한다.




선택을 변경하면 그리드에 출력되는 레코드가 바뀔 뿐만 아니라 메모 컴포넌트에 조립된 SQL문을 보여주므로 문자열이 어떻게 조립되는가를 자세히 살펴볼 수 있을 것이다.

## ■ 일부 문자열 검색

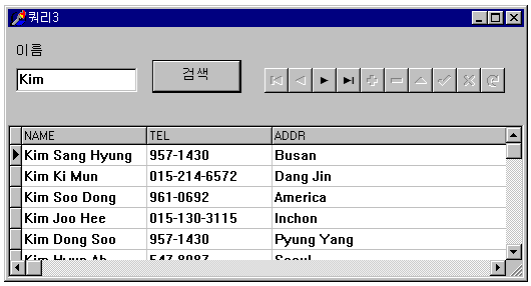
15jang
query3

Where문에 기입되는 조건은 수치나 문자열에 상관없이 모두 제대로 동작한다. 그런데 문자열의 경우는 =연산자를 사용하여 같은가/다른가 외에도 다양한 검색 방법이 필요하며 대표적으로 일부 문자열을 검색해야 할 경우가 있다. 예를 들어 성이 김씨인 사람, 이름 중에 "Joo"가 들어가는 사람, 주소 중 "Masan"이라는 문자열이 있는 경우를 검색해야 할 때이다. 일부 문자열 검색도 Where문을 사용하되 =기호 대신 like 연산자를 사용하며 문자열에는 도스에서 사용하

는 와일드 카드를 이용한다. ?는 한 문자에 대응하며 %는 여러 문자에 대응하므로 이 기호를 문자열 내부에 사용하면 된다. 예를 들면 다음과 같은 SQL문을 작성할 수 있다.

| SQL문 | 의미 |
|---|---|
| Where Name like 'kim%' | 성이 김씨인 사람 |
| Where Name like '%Joo%' | 이름 중에 Joo가 들어가는 사람 |
| Where Addr like '%Masan%' | 고향이 마산인 사람 |

배포 CD의 Query3 예제가 일부 문자열 검색 예를 보여준다. 에디트 박스에서 검색할 문자열을 입력하고 검색 버튼을 누르면 입력한 문자열이 Name 필드에 포함된 모든 레코드를 찾아 검색해 준다.




검색 버튼의 OnClick 이벤트가 다음과 같이 작성되어 있다.

```
{검색 버튼이 눌러지면 에디트에 입력된 문자열을
NAME 필드에 포함한 모든 레코드를 출력한다.}
procedure TForm1.Button1Click(Sender: TObject);
begin
with Query1 do
 begin
 Close;
 SQL.Clear;
 SQL.Add('Select * from "address.dbf"');
 if Edit1.Text<>'' then
  SQL.Add(' Where Name like ''%'+
  Edit1.Text+'%''');
 Open;
 end;
```

```
end;
```

에디트 박스에 kim이 입력되었다면 실행되는 SQL문은 Select * from "address.dbf" Where Name like '%kim%'가 되며 이는 Name 필드에 kim이라는 문자열이 포함되어 있기만 하면 모조리 다 검색해 낸다. 참고로 이렇게 like를 사용하여 검색된 결과는 편집이 불가능한 읽기 전용 상태가 된다.

# 라. SQL 실습

15jang
Query4

앞서의 몇 가지 예제를 만들어 보면서 SQL문에 대해 실습해 보았다. 그런데 SQL문은 그 자체로 하나의 언어이기 때문에 이런 식으로 예제를 일일이 만들어 가며 배우다가는 시간이 너무 많이 든다. 그래서 SQL문을 입력해 넣고 즉각 결과를 확인할 수 있는 SQL 학습용 프로그램을 하나 만들어 보았다. 배포 CD의 Query4.EXE는 SQL문 실습을 위해 의도적으로 작성한 예제이다. 간단한 구조의 ADDRESS.DBF를 사용하며 메모에 SQL문을 직접 입력한 후 버튼을 누르면 즉각 결과를 볼 수 있도록 하였다. 결과는 물론 화면 아래쪽의 그리드에 나타난다. 컴포넌트 배치는 다음과 같다.

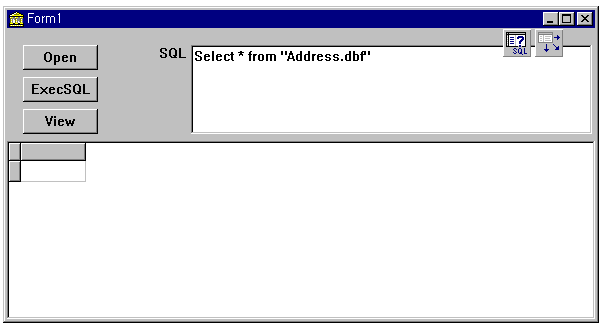



**그림**
SQL 실습을 위한 예제

이 예제는 사용이 목적이므로 제작 과정은 굳이 알 필요가 없다. 단 왼쪽에 있는 세 버튼의 동작을 이해하기 위해 코드를 살펴 보도록 하자. 우선 Open 버튼의 코드는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with Query1 do
  begin
  Close;
  SQL.Clear;
```

```
  SQL.Add(Memo1.Text);
  Open;
  end;
 end;
```

메모에 입력된 SQL문을 실행하는 코드이다. 두 번째 ExecSQL 버튼의 코드는 이와 약간 다르다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
with Query1 do
 begin
 Close;
 SQL.Clear;
 SQL.Add(Memo1.Text);
 ExecSQL;
 end;
end;
```

메모의 SQL문을 실행하기는 하되 마지막에 쿼리를 Open하지 않고 ExecSQL 메소드를 호출하였다. 이 메소드는 SQL 명령을 실행하도록 하는데 Select문 이외의 SQL문은 이 메소드로 실행해야 한다. Select 명령을 실행할 때는 위쪽의 Open 버튼을 사용하고 Insert, Update 등과 같이 테이블을 변경시키는 명령은 ExecSQL 버튼을 사용하면 된다. ExecSQL 명령은 테이블을 변경시키기만 할 뿐 결과를 보여주지 않으므로 테이블의 현재 상황을 보여주는 코드를 작성하였다.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
with Query1 do
 begin
 Close;
 SQL.Clear;
 SQL.Add('Select * from "Address.dbf"');
 Open;
 end;
end;
```

Address.dbf 테이블을 그리드에 뿌려주는 코드일 뿐이다. 이 세 가지 버튼으로 SQL 실습을 해 볼 것이다. 이후 이어지는 SQL 문법에 관한 설명은 이 예제를 사용하여 직접 실습해 보기 바란다. 결과를 금방 금방 확인해 볼 수 있으므로

SQL 실습에 꽤 도움이 될 것이다.

## ■ 중복 필드 제거

Select Age from "Address.dbf"를 실행하면 Age 필드만 나타나며 이 때 나이값이 같은 레코드도 별도의 레코드로 나타난다. 필드값이 중복되는 레코드를 한번만 나타나도록 하려면 Select 다음에 Distinct를 써 준다. 메모에 Select distinct Age from "Address.dbf"를 입력한 후 Open 버튼을 눌러 실행해 보고 어떤 차이가 있는지 살펴보기 바란다.

| AGE |
|---|
| 26 |
| 26 |
| 25 |
| 22 |
| 24 |
| 25 |
| 22 |
| 21 |

Distinct가 없는 경우

| AGE |
|---|
| 21 |
| 22 |
| 23 |
| 24 |
| 25 |
| 26 |

Distinct가 있는 경우

두 개 이상의 필드를 나타낼 경우는 두 필드의 내용이 완전히 중복되는 경우만 제거된다.

## ■ 계산된 필드

Select와 from 사이에는 출력하고자 하는 필드명을 표기하는데 이때 반드시 테이블에 있는 필드만 출력할 수 있는 것은 아니다. 테이블의 필드를 근거로 하여 계산될 수 있는 필드를 출력하도록 할 수 있다. 예를 들어 현재 월급을 10%씩 올려 줄때 각자의 월급이 얼마가 되는지를 살펴 보고자 할 때 월급*1.1 필드를 볼 수 있다. Address.dbf에는 숫자 필드가 Age밖에 없으므로 이 필드로 계산된 필드를 만들어 보자. 나이를 가지고 계산을 한다는 것이 좀 억지스럽기는 하지만 그렇다고 예를 들기 위해 복잡한 테이블을 만들 수는 없으니 여기서의 나이가 월급이나 가격이라고 생각하기 바란다. Select Age, Age+10 from "Address.dbf"이라고 명령을 내리면 다음과 같이 출력된다.

| Age | Age _ 10 |
|---|---|
| 26 | 36 |
| 26 | 36 |
| 25 | 35 |
| 22 | 32 |
| 24 | 34 |
| 25 | 35 |
| 22 | 32 |

Age 필드와 Age에 10을 더한 계산된 필드가 같이 출력되었다. 이 때 계산된 필드의 타이틀은 Age_10이 되는데 이는 계산식으로부터 임의로 주어진 것이다. 이 타이틀을 바꾸고 싶으면 식 다음에 원하는 타이틀을 적어주면 된다. Select Age, Age+10 CalcAge from "Address.dbf"라고 명령을 내리면 Age+10을 계산한 필드가 CalcAge라는 이름으로 출력될 것이다.

## ■ 조건 부정

Where문 뒤에 조건을 반대로 설정하고자 할 때는 not을 사용한다. 예를 들어 Where Age=25는 나이가 25인 경우의 레코드를 추출하지만 Where Not(Age=25)는 나이가 25가 아닌 경우의 레코드를 추출해 낸다. 여러 개의 조건을 논리적으로 연결할 때는 and, or 등의 연산자를 쓸 수 있고 연산 순서를 강제로 변경하고 싶을 때는 괄호를 사용하면 된다.

## ■ 값 정의 여부

필드에 값이 정의되어 있지 않은 레코드를 검색해 내려면 is Null을 사용한다. 예를 들어 Where Depart is Null은 Depart 필드가 공란으로 비어 있는 레코드만을 검색해 내며 반대로 Where Depart is Not Null은 Depart 필드가 정의되어 있는 레코드만 검색해 낸다.

## ■ 범위 점검

정수값을 가지는 필드가 특정 범위에 속해 있는지를 검사할 때는 Between And를 사용한다. Where Age Between 22 and 25는 Age 필드값이 22~25 사이인 레코드를 검색해 낸다. 불연속적인 값에 속한 필드를 검색할 때는 In 연산자와 괄호를 사용한다. Where Age In (21,23,25)는 Age가 21 또는 23 또는 25인 레코드를 검색해 낸다. In 다음에는 반드시 괄호를 사용해야 한다.

## ■ 그룹 짓기

group by 명령은 그룹 단위로 같은 값을 가지는 레코드를 묶어서 추출하도록 한다. 예를 들어 특정한 부서에 속한 사람들의 어떤 필드를 계산한 결과를 보고자 할 때 사용하는데 좀 더 구체적으로 예를 들자면 "총무부에 속한 사원의 평균 키", "삼영 식품으로부터 구입한 제품의 가격 총합" 등을 계산할 때 이 명령을 사용한다. group by 뒤에 그룹을 지을 필드를 지정해 주며 select와 from 사이에

는 그룹으로 지정된 필드와 관계된 값만 와야 한다. 다음 예를 보자.

```
select depart, sum(age)
from "Address.dbf" group by depart
```

이 명령은 depart 필드, 즉 부서별로 그룹을 지어 각 부서의 나이 총합을 계산해 내라는 것이다. 출력되는 필드는 부서와 각 부서별 나이 총합이다.

| depart | SUM OF age |
|---|---|
| Manage | 71 |
| Project | 71 |
| Publish | 93 |
| Study | 117 |

이 경우에도 sum(age) 다음에 타이틀명을 별도로 지정해 줄 수 있다.

## ■ 그룹 함수

앞의 group by 예에서 sum 이라는 함수가 사용되었는데 group by 문과 함께 사용되는 이런 함수를 그룹 함수라고 하며 다음과 같은 종류가 있다.

| 함수 | 설명 |
|---|---|
| Sum | 합계를 구한다. |
| Avg | 평균값을 구한다. |
| Min | 최소값을 구한다. |
| Max | 최대값을 구한다. |
| Count | 개수를 구한다. |

다음 명령은 각 부서별로 최소 나이, 최대 나이 그리고 평균 나이를 구해 준다.

```
select depart, min(age) 최소나이, max(age) 최대나이, avg(age) 평균나이
from "Address.dbf" group by depart
```

실행 결과는 다음과 같다.

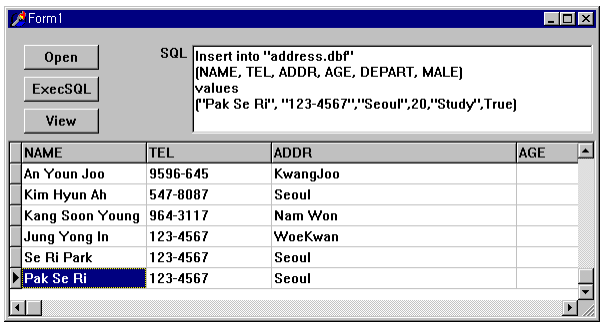
| depart | 최소나이 | 최대나이 | 평균나이 |
|---|---|---|---|
| Manage | 22 | 25 | 23.6666666666667 |
| Project | 22 | 26 | 23.6666666666667 |
| Publish | 22 | 25 | 23.25 |
| Study | 21 | 26 | 23.4 |

## ■ 레코드 삽입

테이블에 레코드를 삽입할 때는 Insert 명령을 사용한다. 예를 직접 보면서 레코드를 삽입해 보도록 하자. 다음 명령을 메모에 입력한다. 이 예를 보면 Insert 명령을 어떻게 사용하는지 이해할 수 있을 것이다.

```
Insert into "address.dbf"
(NAME, TEL, ADDR, AGE, DEPART, MALE)
values
("Pak Se Ri", "123-4567","Seoul",20,"Study",True)
```

단 이 명령은 테이블에 변경을 가하기 때문에 Open 명령으로는 실행할 수 없으며 ExecSQL 메소드를 호출해 주어야 한다. 명령을 입력한 후에 ExecSQL 버튼을 눌러 이 명령을 실행시켜 보자. 그러나 명령만 실행될 뿐 그리드는 텅 비어 버릴 것이다. 테이블에 변경이 가해진 후는 쿼리가 닫혀 버리기 때문인데 View 버튼을 눌러 쿼리를 다시 열어 주도록 하자.




테이블의 제일 끝에 Pak Se Ri 레코드가 추가되었다.

## ■ 레코드 변경

레코드를 변경할 때는 Update 명령을 사용한다. 다음 명령을 내려 보자.

```
Update "address.dbf"
set name="Se Ri Pak", age=21
where name="Pak Se Ri"
```

그리고 그리드를 다시 확인해 보면 Pak Se Ri 레코드가 Se Ri Pak 으로 변경되었을 것이다.

### ■ 레코드 삭제

레코드를 삭제할 때는 delete 명령을 사용한다. 다음은 앞서 삽입한 레코드를 삭제하는 명령이다.

```
delete from "address.dbf" where name="Se Ri Pak"
```

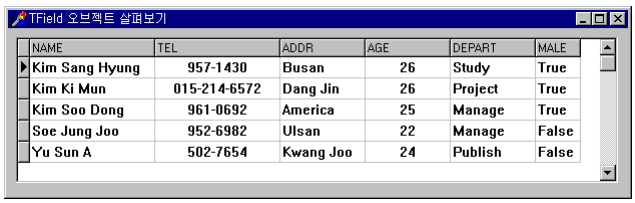
여기서는 SELECT 명령을 중심으로 SQL문을 연구해 보았다. 물론 그 외에도 많은 SQL문이 있지만 여기서는 그 외의 명령에 대해서는 언급하지 않는다. 델파이의 도움말에도 SQL에 관한 도움말은 없으므로 별도의 SQL 관련 서적을 참고하기 바란다.

## 마. TField 오브젝트

DBGrid를 사용하여 DB 파일의 레코드 전체를 한꺼번에 볼 경우 DB 파일에 기록되어 있는 데이터의 모양을 그대로 보여주며 전혀 가공을 하지 않는다. 하지만 DB 파일에 있는 그대로 보여주는 것보다는 우리가 원하는대로 필드를 가공하면 더 보기 좋고 편집하기도 편하도록 만들 수 있다. 일단 실습을 위해 간단한 예제를 하나 만들어 보자. 배포 CD의 Field1 예제는 필드의 속성을 조정하는 예를 보이기 위해 작성되었으며 다음과 같다.


Query1을 배치하고 이 컴포넌트의 SQL 속성을 Select * from "address.dbf"로 설정한 후 Active 속성을 True로 바꾸어 디자인중에 DB 파일을 그리드로 볼 수 있도록 하였다. 물론 DataSource 컴포넌트와 쿼리, 그리드 컴포넌트는 속성으로 연결되어 있다. 이 프로그램은 보다시피 표준 VGA의 화면을 고려하여 폭을 최대한 넓게 잡았지만 DB 파일의 모든 필드를 다 보여주지 못하고 있다. 왜냐하면 ADDR 필드가 지나치게 넓은 공간을 차지하고 있기 때문이며 이는 DB 파일을 설계할 때부터 ADDR 필드가 30의 폭을 가지도록 만들어졌기 때문이다. 우리가 원하는 출력 형태는 다음과 같이 각 필드가 적당한 폭을 가져 한 눈에 레코드 전체가 보이도록 하는 것이다. 설사 일부 필드의 우측이 가

려지더라도 꼭 필요한 만큼만 폼을 아담하게 디자인하는 것이 더 보기 좋다.



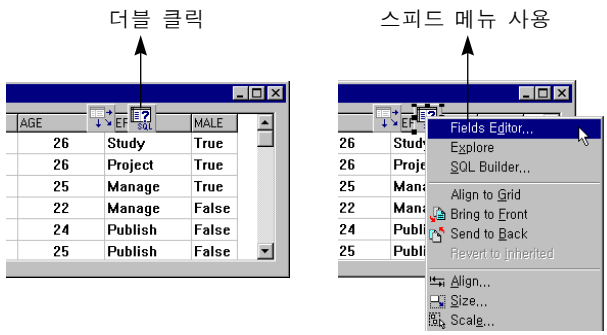

프로그램이 실행중일 때 사용자는 각 필드의 헤더 경계를 마우스로 드래그하여 폭을 조정할 수 있으며 필드를 드래그하여 위치를 바꿀 수도 있다. 그러나 디자인중에는 마우스로 필드의 폭을 변경할 수 없다. 필드의 폭이나 순서를 바꾸려면 각 필드의 속성을 변경해야 한다.

15jang
field1

델파이에서는 레코드에 속한 각각의 필드를 TField 오브젝트로 조정한다. TField 오브젝트는 레코드 내의 한 필드의 속성을 가지며 데이터 셋(Query 또는 Table)이 활성화될 때 레코드 내의 모든 필드에 대해 자동으로 생성된다. TField 오브젝트의 속성을 변경하면 그리드에 출력되는 필드의 모양을 바꿀 수 있으며 TField 오브젝트의 속성을 변경할 때는 필드 편집기(Fields Editor)를 사용한다. 필드 편집기를 여는 방법에는 두 가지가 있다.

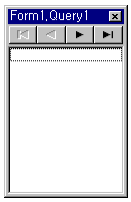
① Query 컴포넌트를 더블클릭한다. Table 컴포넌트에 대해서도 마찬가지다.
② Query 컴포넌트의 스피드 메뉴에서 첫 번째 항목인 Fields Editor 를 선택한다.

**그림**
필드 편집기 열기




필드 편집기의 필드 리스트 박스에는 수동으로 만들 필드의 목록이 나타나며 처음 필드 편집기가 열릴 때는 비어 있다.

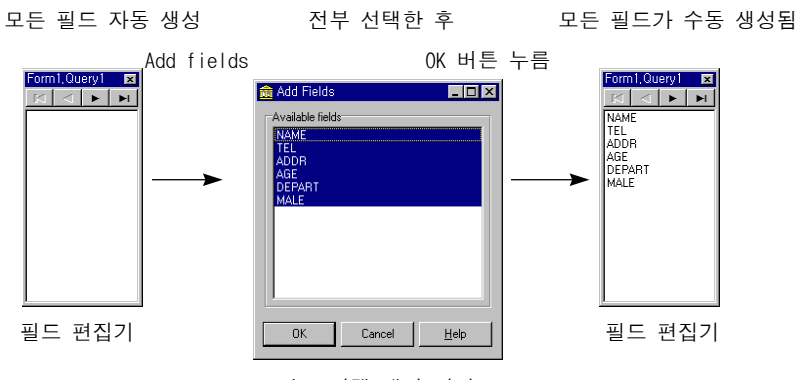
**그림**
필드 편집기




여기서 자동은 dynamic 을 번역한 말이고 수동은 static 을 번역한 말이다.

필드 편집기가 열릴 때 이 리스트 박스에 필드의 목록이 없는 이유는 모든 필드가 자동으로 생성되도록 되어 있기 때문이다. 자동으로 생성되는 필드는 DB 파일에서 설정한 물리적인 속성을 그대로 따른다. 즉 필드의 폭, 정렬 형태, 출력 형태, 필드간의 순서 등등이 데이터 베이스 파일을 설계할 때와 동일하다. 이런 속성을 변경하려면 각 필드가 자동으로 생성되는 것을 금지시키고 수동으로 생성되도록 해야 한다.

**그림**
모든 필드를 수동 생성으로 전환하는 방법




필드 편집기의 스피드 메뉴에서 Add Fields 항목을 선택하면 수동 생성할 필드의 목록을 가진 필드 선택 대화상자가 나타나며 모든 필드가 선택되어 있다. 이 상태로 OK 버튼을 누르면 자동으로 생성되는 필드를 수동으로 생성되도록 모두 바꾸어 주며 필드 편집기의 필드 리스트 박스에 모든 필드가 나타난다.

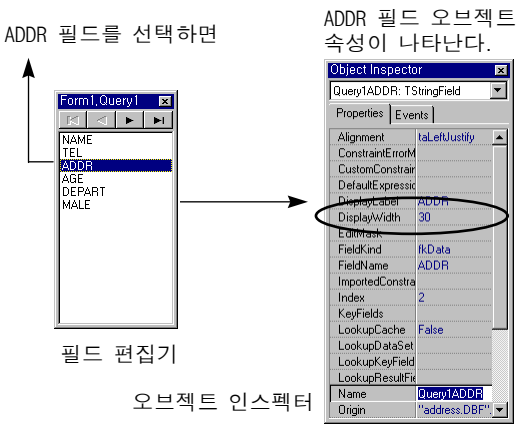
하지만 아직까지 그리드에 나타난 결과에는 아무런 변화가 없다. 수동으로 생성되도록 바꾸어도 자동으로 생성될 때와 동일한 속성을 그대로 가지고 있기 때문이다. 이제부터 각 필드의 속성을 변경하여 여러 가지 변화를 줄 수 있다.

## ■ 폭 지정

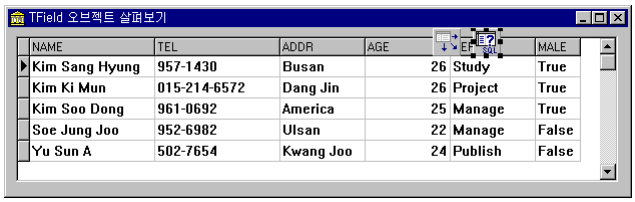
필드 편집기의 수동 생성 필드 리스트에는 그리드로 출력될 필드의 목록이 있으며 이 목록들이 TField 오브젝트들이다(실제로는 그 파생 오브젝트들이다). 필드 편집기에서 이 목록 중 하나를 선택하면 오브젝트 인스펙터에 필드의 속성이

나타나며 우리는 이 속성을 조정하여 그리드에 필드가 출력되는 모양을 마음대로 바꿀 수 있다.

그림
필드 오브젝트의 속성 편집




ADDR 필드의 DisplayWidth 속성을 보면 30으로 설정되어 있으며 이 값은 애초에 DB 파일이 설계될 때 입력된 값이다. 이 값을 변경하면 그리드에 나타나는 ADDR 필드의 폭을 변경할 수 있다. 이 값을 10으로 변경하면 폼의 모양이 다음과 같이 즉각 바뀐다.

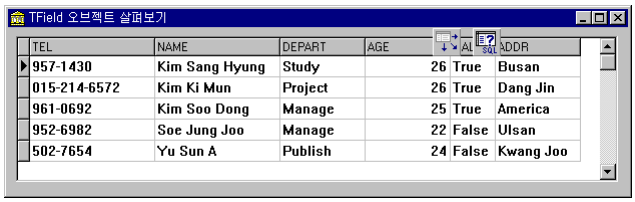



ADDR 필드의 폭이 줄어들었으므로 우측의 필드들도 모두 폼 안으로 들어오게 되었다. 나머지 필드들도 동일한 방법으로 폭을 변경한다. 참고로 그리드의 필드 폭은 그리드의 Columns 속성으로도 변경할 수 있다.

## ■ 순서 변경

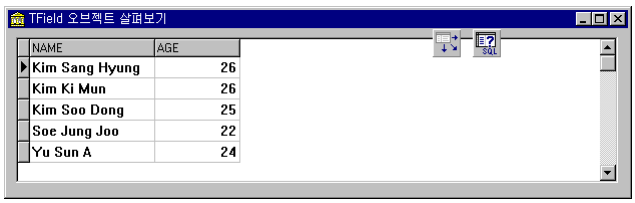
그리드에 나타나는 필드의 순서는 DB 파일에서 정의된 순서대로이다. 이 순서를 바꾸고 싶다면 필드 오브젝트의 Index 속성을 변경한다. Index 속성은 그리드에 나타날 필드의 순서를 지정하며 0이면 첫 번째, 1이면 두 번째 등등이다. AGE 필드의 Index 속성을 1로 설정하면 NAME 필드와 TEL 필드 사이로 AGE 필드가 이동한다.

Index 속성을 이용하는 방법보다 더 편리한 방법은 필드 편집기에서 필드를 드래그하여 위치를 변경하는 방법이다. 너무 쉽지 않은가? 다음은 필자가 마음대로 필드의 순서를 바꾸어 본 예이다.



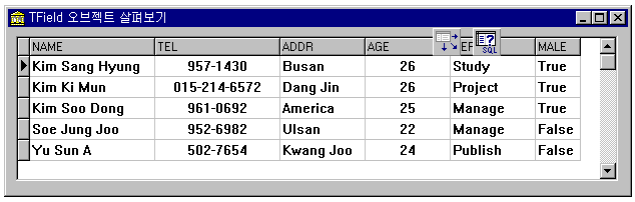

## ■ 일부 숨김

DB 파일에 정의된 모든 필드를 반드시 그리드로 다 출력해야 할 필요는 없다. 꼭 필요한 필드만 출력하고 나머지는 숨기려면 두 가지 방법을 사용한다. 먼저 숨기고자 하는 필드의 Visible 속성을 False로 바꾸는 방법이 있고 두 번째는 아예 필드를 수동 생성 리스트에서 삭제해 버리는 방법이 있다. 수동 생성 리스트에서 삭제하려면 필드 편집기에서 삭제를 원하는 필드를 선택한 후 팝업 메뉴에서 Delete를 선택하거나 아니면 키보드의 Del키를 누른다. 다음은 NAME과 AGE 필드만 남기고 나머지 필드는 모두 삭제한 예이다.



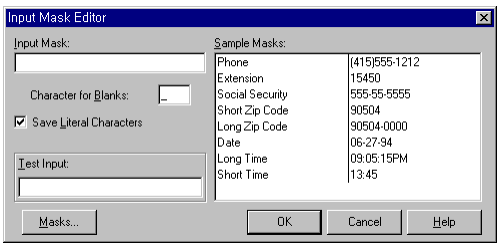

## ■ 정렬

필드의 데이터가 출력될 때 수치값은 우측 정렬되며 문자열은 좌측으로 정렬된다. 정렬 상태를 바꾸려면 TField 오브젝트의 Alignment 속성을 사용한다. 다음은 AGE와 TEL 필드를 중앙 정렬로 바꾸어 본 것이다.



## ■ 최대 최소값 지정

수치 데이터의 경우는 입력을 받을 수 있는 범위가 한정되어 있다. 예를 들어 나이값은 음수가 절대로 올 수 없으며 150이상이 될 리도 없다(설마?). 사용자의 실수를 최대한 방지해 주기 위해 입력 가능한 값의 범위를 지정하려면 MinValue, MaxValue 속성을 사용한다. 문자열 데이터의 경우도 입력상의 실수를 줄이기 위해 EditMask라는 속성을 사용한다. 우편번호의 경우는 NNN-NNN의 형식을 반드시 가져야 하므로 이런 형태로 입력되도록 마스크를 설정해 두는 것이 좋다.

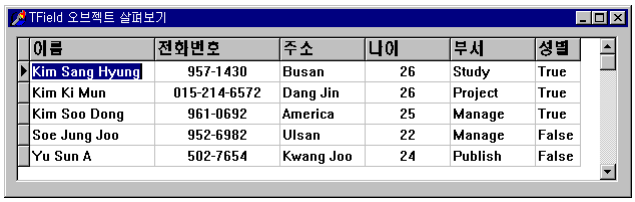



TField 오브젝트를 사용하면 위에서 소개한 간단한 속성 변경 외에도 계산된 필드를 만든다거나 출력 형태, 입력 요구 지정 등 고급 수준의 기법을 구사할 수 있다.

## ■ 필드명 변경

그리드의 상단에는 각 필드명을 나타내는 타이틀이 나타나는데 이때 타이틀은 테이블에 정의된 필드명을 그대로 따라간다. 즉 NAME 필드의 타이틀은 NAME 이 되고 AGE 필드의 타이틀은 AGE 가 된다. 테이블의 필드명은 한글을 사용하는데도 제약이 있으며 DB 의 효율상 길이에도 일정한 한계를 둔다. 게다가 어떤 DBMS 는 소문자를 쓸 수 없으며 공백을 포함할 수도 없도록 되어 있다. 그래서 ADDR, AGE 등과 같이 가급적 짧은 명칭으로 만드는데 프로그래머야 효율상 짧은 이름이 좋다지만 사용자가 보기에는 썩 좋지 못하다.

그래서 그리드의 타이틀을 필드명과 다르게 작성할 수 있는데 이 때는 TField 오브젝트의 DisplayName 속성을 사용한다. 이 속성의 디폴트값은 테이블의 필드명으로 되어 있는데 이를 변경하면 그리드의 타이틀을 변경할 수 있으며 한글이나 공백도 자유롭게 사용할 수 있다. 다음은 각 필드명을 한글로 바꾸고 폰트를 예쁘게 꾸며본 것이다.



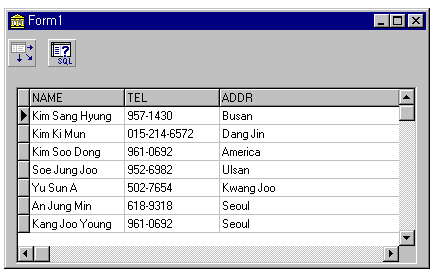

이렇게 해 주면 사용자들이 보기에 훨씬 더 편리할 것이며 각 필드의 의미를 더 직관적으로 이해할 수 있을 것이다. DisplayName은 그리드의 타이틀에 나타나는 제목만 변경할 뿐이지 실제로 필드명을 변경하는 것은 아니다.

## ■ 계산된 필드

15jang
CalcField

테이블이나 쿼리(합쳐서 데이터 셋)는 실제 테이블에 있는 필드들을 읽어와 데이터 컨트롤을 통해 사용자에게 보여준다. 그런데 어떤 경우는 테이블에 있지 않은 필드를 만들어서 보여줘야 하는 경우가 있다. 예를 들자면 점수값을 기준으로 A+, A, B+, B 등의 평점을 보여준다거나 높이, 너비, 폭을 조합하여 부피를 보여줘야 하는 경우등이 있다. 이런 필드는 기존 필드로부터 계산해 낼 수 있기 때문에 보통 테이블에 직접 넣지 않고 실행중에 필드값을 만들어 보여주는데 이런 필드를 계산된 필드(Calculated Field)라고 한다.

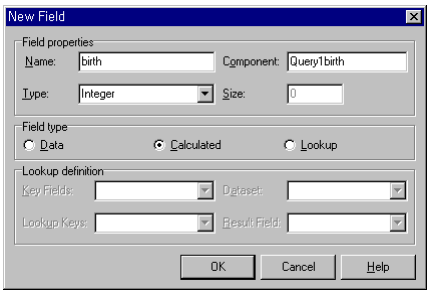
계산된 필드를 만드는 아주 간단한 예제를 만들어 보자. Address.dbf 에서 age 필드로부터 생년을 계산하는 예제를 만들 것이다. 나이와 현재 년도를 알면 생년은 당연히 계산할 수 있으므로 테이블에는 생년 필드가 들어가 있지 않지만 사용자의 편의를 위해 프로그램이 실행중에 계산해 보여줄 수도 있다. 새 프로젝트를 시작하고 DataSource, Query, DBGrid 세 개의 컨트롤을 폼에 배치한다. 그리고 쿼리의 SQL 문에 Select * from address.dbf 질의를 작성하여 DBGrid 에 Address.dbf 의 모든 필드가 나타나도록 해 보자. 여기까지 작업하면 테이블에 있는 필드만 나타난다.




이제 계산된 필드 birth 를 넣어보자. 쿼리 컴포넌트를 더블클릭하여 필드 에

디터를 불러낸 후 Name, Age 만 포함시킨다. 그리고 팝업 메뉴에서 New Field 를 선택한 후 대화상자에 다음과 같이 입력하여 birth 필드를 만든다.

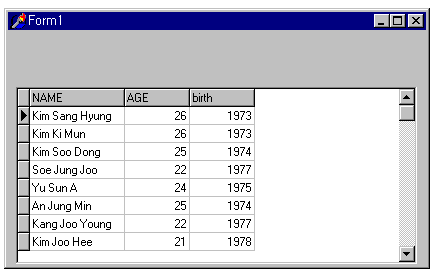
그림
새 필드 만들기




필드의 이름은 birth 로 주었고 타입은 정수형으로 선언하였으며 필드 타입은 Calculated 로 선택했다. 그러면 Query1birth 라는 필드 컴포넌트가 생성된다. 이 필드는 실행중에 계산되는 필드이기 때문에 디자인시에는 값이 나타나지 않는다. 계산된 필드에 값을 대입해 주는 작업은 쿼리(또는 테이블) 컴포넌트의 OnCalcField 이벤트 핸들러에서 해 준다.

```
procedure TForm1.Query1CalcFields(DataSet: TDataSet);
var
  Today:TDateTime;
  Year, Month, Day:Word;
begin
  Today:=Date;
  DecodeDate(Today, Year, Month, Day);
  Query1Birth.Value:=Year-Query1Age.Value+1;
end;
```

올해의 연도를 구한 후 Age 필드값을 빼고 1 을 더하면 생년을 구할 수 있다. 실행해 보면 birth 필드에 생년이 나타날 것이다.

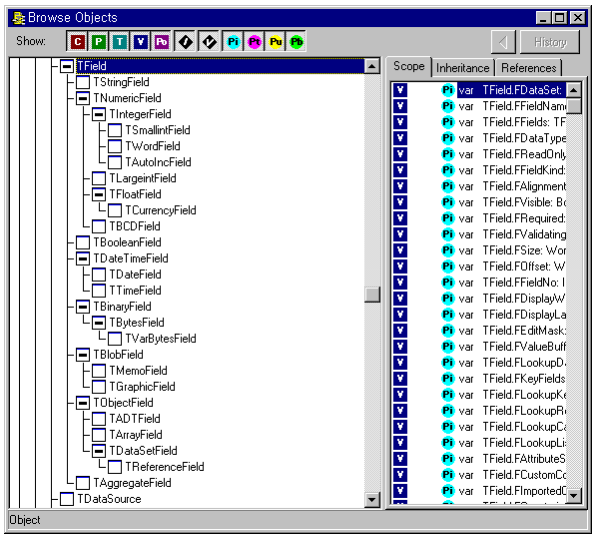


## ■ TField 의 파생 오브젝트

필드 에디터에서 필드 오브젝트를 수동 생성으로 바꾸어 주면 폼의 타입 선언부에 TField형의 오브젝트가 선언된다. 코드 에디터에서 폼의 타입 선언부를 보면 필드 컴포넌트가 생성되어 있을 것이다.

```
type
  TForm1 = class(TForm)
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Query1: TQuery;
    Query1NAME: TStringField;
    Query1TEL: TStringField;
    Query1ADDR: TStringField;
    Query1AGE: TSmallintField;
    Query1DEPART: TStringField;
    Query1MALE: TBooleanField;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

TStringFields, TSmallintField 등의 컴포넌트가 바로 필드 컴포넌트이며 필드 편집기로부터 만들어지는 컴포넌트이다. 이 컴포넌트들은 TField 로부터 파생되며 각 타입별로 가지는 속성이 조금씩 다르다. 예를 들어 값의 범위를 지정하는 MaxValue, MinValue 속성은 정수형 필드에는 있지만 문자열 필드에는 없다. TField 로부터 어떤 필드 컴포넌트가 파생되는지는 오브젝트 브라우저를 보면 한눈에 알아볼 수 있다.




테이블의 필드형에 따라 적절한 필드 컴포넌트가 생성되는데 이는 별도의 설명을 하지 않아도 클래스 이름으로도 쉽게 구분할 수가 있다. 즉 문자열 필드는 TStringField 컴포넌트가 되고 정수형 필드는 TIntegerField 나 TSmallIntField 컴포넌트가 될 것이다.

## ■ 필드 드래그하기

15jang
FieldDrag

필드 에디터에는 아주 재미있는 기능이 하나 있다. 필드 에디터에 나타난 필드명을 폼으로 드래그하면 그 데이터형에 가장 잘 어울리는 데이터 컨트롤 컴포넌트를 배치해 주는 기능이다. 이 기능을 잘 사용하면 아주 빠른 속도로 폼을 디자인할 수 있으며 일일이 반복해 주어야 할 잡일을 원터치로 해 치울 수 있다.

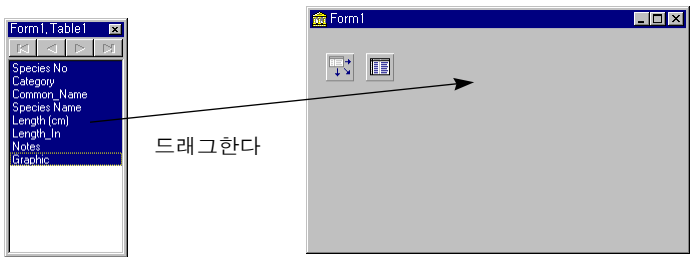
어떤 기능인지 새로운 예제를 하나 만들어서 테스트해 보도록 하자. 빈 폼에 DataSource, Table 컴포넌트만 배치한다. 그리고 Table 의 DatabaseName 속성에 DBDEMOS, TableName 속성에 Biolife.db 를 대입해 주고 DataSource 의 DataSet 속성에 Table1 을 설정한다. 이 상태에서 테이블 컴포넌트를 더블클릭하면 필드 에디터가 열릴 것이다. 필드 에디터의 팝업 메뉴에서 Add Fields 항목을 선택한 후 전체 필드를 선택한다.

이 상태에서 필드 에디터의 팝업 메뉴에서 Select All 을 선택해 모든 필드를 선택하고 필드를 폼으로 드래그해 보자. 신기한 현상이 일어날 것이다.

전부 선택된 상태에서

**그림**

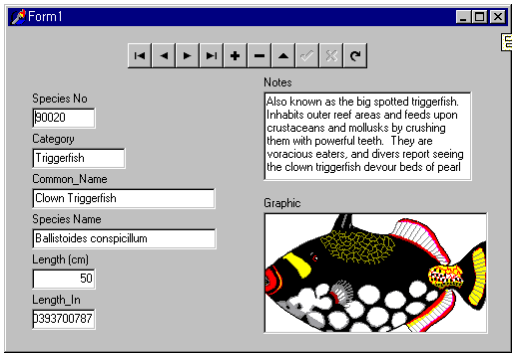
필드 편집기에서 필드를 폼으로 드래그한다.




이렇게 하면 폼에 드래그된 필드들이 적당한 데이터 컨트롤로 변경되어 폼에 배치된다. 뿐만 아니라 각 컨트롤은 이미 테이블의 필드와 연결되어 있으며 컨트롤 상단에는 필드를 설명하는 레이블까지 배치되어 있다. 이렇게 폼에 배치된 후에는 위치나 크기를 자유롭게 조정할 수 있다. 이제 여기에 네비게이터 컴포넌트를 배치하고 테이블의 Active 속성을 True 로 변경해 주기만 하면 간단하게 예제가 완성된다.

**그림**

필드 편집기로부터 만들어진 데이터 컨트롤




정말 너무 쉽고 재미있지 않은가? 이 기능을 잘 활용하면 아무리 복잡한 테이블이라도 데이터 컨트롤을 배치하는 것은 식은 죽 먹기일 것이다.

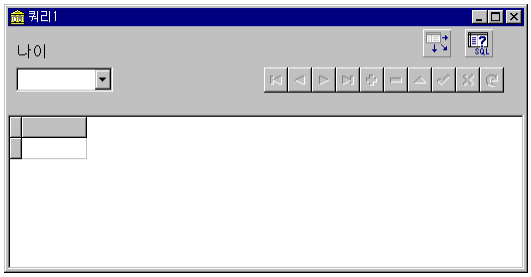
# 바. 파라미터 쿼리

쿼리는 SQL 속성에 문자열로 질의를 작성하여 질의 결과를 얻어내는 컴포넌트이다. 그런데 검색 조건이나 순서, 추출될 필드를 변경하려면 항상 SQL 문을 다시 작성해야 한다. 극히 일부분만 변경하려고 해도 SQL 전체를 다시 작성해야 하는데 이때는 SQL 문을 미리 작성해 놓고 파라미터만 변경하면 된다. 파라미터란 SQL 문 내에 삽입되는 일종의 변수라고 생각할 수 있다. 예제를 만들면서 실

15jang
ParamQuery

습을 해 보도록 하자.

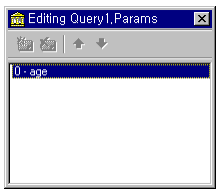
제일 처음 만들었던 쿼리 예제인 Query.dpr 과 똑같이 컴포넌트를 배치한다. 아니면 Query 예제를 Save As 해서 이 프로젝트로 복사한 후 수정해도 된다. 단 콤보 박스의 Items 속성에 All 은 넣지 않기로 하자. 폼의 모양은 Query 예제와 완전히 동일하다.



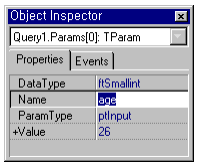

이제 쿼리의 SQL 문을 미리 작성해 놓는다. SQL 속성을 더블클릭하여 문자열 리스트에 다음과 같이 질의를 작성한다.

```
select * from address.dbf where age=:age
```

파라미터는 SQL 문 내에 콜론으로 시작되는데 여기서 :age 가 파라미터이다. 즉 이 질의문은 나이가 :age 파라미터와 같은 레코드만 추출하는 것이되 파라미터는 실행중에 쿼리 컴포넌트의 Params 속성에 대입되는 값으로 대체된다. 쿼리의 Params 속성을 더블클릭하면 파라미터 편집기가 열리고 SQL 속성에 기입된 파라미터들이 나열될 것이다.




age 파라미터가 이미 입력되어 있다. 파라미터는 Params 속성에 배열 형태로 정의되어 있다. 즉 SQL 문 내의 첫 번째 파라미터는 Params[0], 두 번째 파라미터는 Params[1] 등이 된다. 각 파라미터는 TParam 컴포넌트이며 따라서 쿼리의 Params 속성은 TParam 을 요소로 가지는 일차 배열이라고 할 수 있다. 파라미터 편집기에서 파라미터를 선택하면 오브젝트 인스펙터에 이 컴포넌트의 속성이 나타난다. 다음은 age 파라미터의 속성이다.




속성이래봐야 겨우 네 개에 불과한데 DataType 속성은 파리미터의 데이터 타입을 나타내며 ParamType 은 파라미터의 용도를, Value 에는 초기값을 주면 된다. 각각 ftSmallInt, ptInput, 26 으로 설정하도록 하자. 이제 파라미터가 만들어졌으므로 SQL 문은 그대로 두고 파라미터만 변경해 주면 된다. 콤보 박스의 OnChange 이벤트 핸들러를 다음과 같이 작성하자.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  Query1.Close;
  Query1.Params[0].Value:=StrToInt(
    ComboBox1.Items[ComboBox1.ItemIndex]);
  Query1.Open;
end;
```

Params[0].Value 즉 :age 파라미터의 값을 콤보 박스에 선택된 값으로 대입하고 질의를 다시 실행하도록 하는 코드이다. 그래서 사용자가 콤보 박스에서 어떤 값을 선택하는가에 따라 실제 실행되는 질의문이 달라지며 그리드에 나타나는 결과도 달라지게 된다. 마지막으로 폼이 생성될 때의 초기화 처리를 해 준다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.ItemIndex:=5;
  Query1.Open;
end;
```

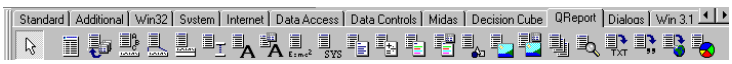
최초 콤보 박스의 인덱스를 5(=26)로 설정하고 질의를 시작하도록 하였다. 실행 결과는 Query 예제와 동일하되 다만 SQL 문은 변경하지 않고 파라미터를 사용한다는 것만 다르다.

# 15-4 보고서 작성

## 가. 퀵 리포트 소개

데이터 베이스는 자료를 체계적으로 정리하고 관리하는 시스템이다. 정리된 데이터 베이스는 화면상으로 볼 수도 있지만 결과적으로 종이에 인쇄되어야 한다. 데이터 베이스에 구축한 데이터를 요약적으로 출력한 것을 보고서(Report)라고 하는데 일종의 인쇄 기술이지만 DB 의 자료를 가져와 인쇄해야 하므로 생각처럼 그리 간단한 것은 아니다.

델파이는 퀵 리포트라는 일련의 보고서 작성 컴포넌트를 제공하는데 이 컴포넌트들을 사용하면 복잡한 보고서 작성을 쉽게 할 수 있다. 델파이 1.0 에는 리포트 스미스(Report Smith)라는 보고서 툴이 있었는데 그다지 사용하기 쉽지가 않았다. 퀵 리포트는 볼랜드사에서 만든 것이 아니라 QuSoft 라는 회사에서 만든 것을 볼랜드사이에서 라이센스하여 델파이에 포함시킨 것이다. 퀵 리포트에 관련된 컴포넌트들은 컴포넌트 팔레트의 QReport 페이지에 모여 있다.




무려 23 개나 되는 컴포넌트가 있는데 컴포넌트의 수로 짐작해 볼 수 있듯이 그리 간단하지 않다.

## 나. QReport 예제

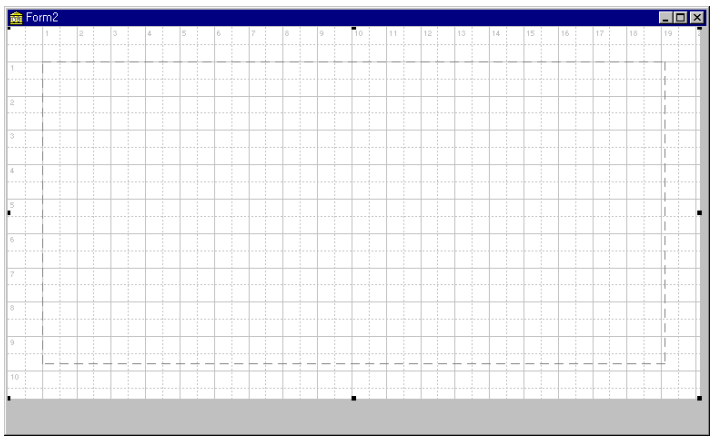

15jang
QReport

퀵 리포트는 지금까지 사용해 오던 컴포넌트들과는 사용 방법이 질적으로 조금 다르기 때문에 처음 대하는 사람에게는 아주 생소하게 느껴질 것이다. 그래서 일단은 아주 간단한 보고서 예제를 하나 만들면서 이 컴포넌트와 친해지도록 해 보자. 여기서 만들 보고서 예제는 이 장에서 제일 처음 만들었던 ADDR 프로젝트에 보고서 출력 기능을 첨가하되 꼭 필요한 기능만 넣어 볼 것이다. 다음 단계를 따라 실습을 해 보자.

1 QREPORT 디렉토리를 만들고 ADDR 예제를 QReport.dpr 로 Save As 한다. ADDR 예제와 똑같은 이름만 다른 프로젝트가 만들어졌을 것이다. 이 예제에 리포트 출력 기능을 넣어 보도록 할 것이다.

2 리포트를 만들려면 보조 폼이 하나 있어야 하므로 File/New Form 명령으로 새 폼을 추가한다. 빈 폼이 주어지는데 여기에 QReport 페이지의 첫 번째에 있는 QuickRep 컴포넌트를 배치한다. 배치하고 나면 모눈종이같이 생긴 것이 배치되는데 이것이 보고서를 출력할 용지가 된다. 디폴트 크기가 너무 크므로 적당하게 크기를 줄이도록 하자. 폭은 가급적 넓게 잡는 것이 좋되 높이는 줄여도 상관없다.

그림
QuickRep 컴포넌트가 배치된 모양




폼을 생성한 후 이 폼을 QReport_f2 로 저장한다. 현재 이 프로젝트에는 메인 폼과 리포트 폼이 있는데 두 폼에서 서로를 인식할 수 있도록 해 주어야 한다. 메인 폼의 Uses 절에 QReport_f2 유닛을 적어주어 메인 폼에서 보고서를 호출할 수 있도록 한다.

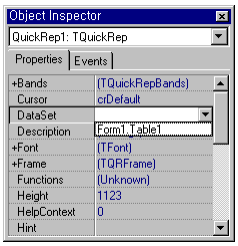
```
unit QReport_f;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, StdCtrls, Db, DBTables, Mask, QReport_f2;
```

보고서 폼에서 테이블을 액세스해야 하므로 메인 폼을 uses 절에 적어주어야 한다.

```
implementation

uses QReport_f;
```

두 개의 폼이 서로 참조하는 상황이되 무한 참조를 방지하기 위해 리포트 폼에서는 implementation 부에 메인 폼을 적어주었다. 이제 메인 폼과 리포트 폼은 서로를 인식하게 된다.

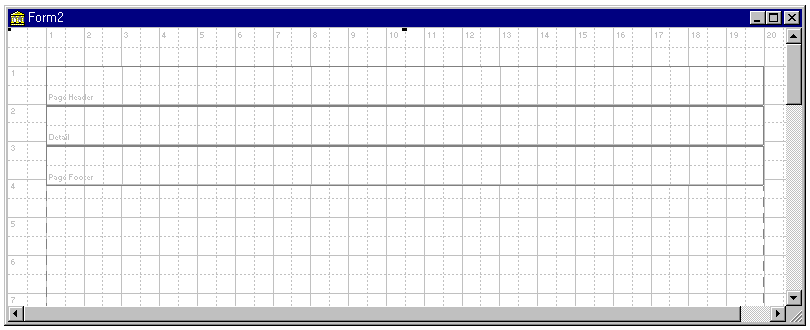
3 보고서 폼은 테이블의 필드 내용을 가져와 인쇄를 해야 하므로 보고서 폼과 테이블을 연결시켜 주어야 한다. QuickRep1 컴포넌트를 선택한 후 오브젝트 인스펙터를 보면 DataSet 이라는 속성이 있을 것이다. 이 속성에 Form1.Table1 을 선택해 준다. 콤보 박스를 열어보면 이 폼이 목록에 있으므로 선택만 해 주면 된다.




DataSet 을 가르쳐 주었으므로 이제 퀵 리포트는 자기가 출력해야 될 데이터가 어느 테이블에 있는지를 알게 될 것이다.

4 퀵 리포트는 여러 개의 밴드로 구성된다. 밴드란 보고서에 출력할 정보의 덩어리를 얘기하는데 예를 들어 머리말, 꼬리말, 본문 따위이다. 밴드를 구성하려면 QRBand 컴포넌트를 QuickRep 컴포넌트 위에 배치하면 된다. 컴포넌트 팔레트의 4 번째에 있는 QRBand 컴포넌트를 가져와 QuickRep 컴포넌트위에 배치해 보자. 마치 툴바가 폼에 배치되는 것처럼 폼의 위쪽에 척척 달라붙을 것이다.

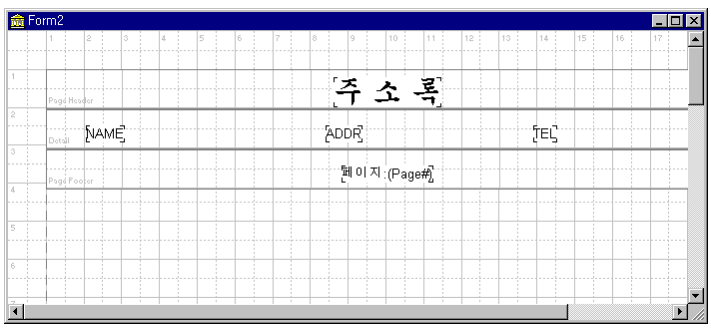
이런 식으로 세 개의 밴드를 폼에 배치하면 세 개의 밴드가 3 층으로 쌓여 있을 것이다. 이 상태에서 각 밴드의 BandType 속성을 각각 rbPageHeader, rbDetail, rbPageFooter 로 수정한다. BandType 속성은 이 밴드에 출력할 정보가 어떤 종류인가를 지정한다. 밴드를 배치한 후의 모양은 다음과 같다.




5 이제 밴드안에 컴포넌트를 배치해 보자. 제일 위에 있는 밴드인 PageHeader 에 QRLabel 을 배치하고 Caption 을 '주소록'으로 변경한다. QRLabel 은 Label 컴포넌트와 마찬가지로 문자열을 출력하는 컴포넌트이며 특별한 기능은 없다. PageHeader 밴드는 페이지의 선두에 출력되는데 여기에 보고서의 제목을 적은 것이다. 폰트를 적당히 크게 해서 제목같이 보이도록 만들어 보자.

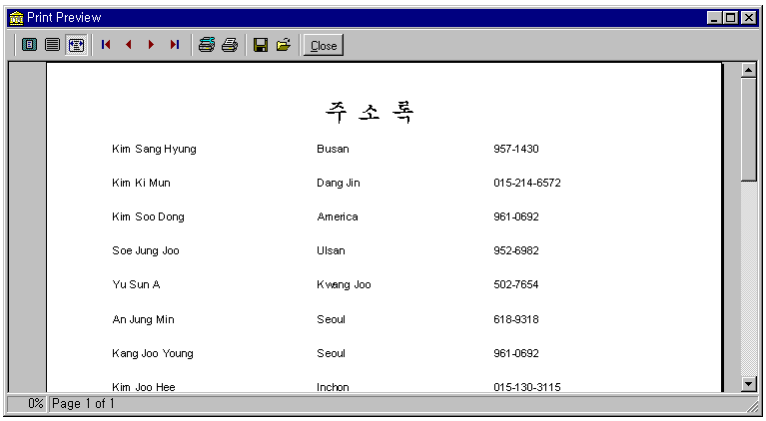
PageFooter 에는 페이지 번호를 출력해 보자. 페이지 번호나 날짜, 시간 등의 정보를 출력할 때는 QRSysData 컴포넌트를 사용하는데 이 컴포넌트의 Text 속성으로 출력 정보의 제목을 지정하고 Data 속성으로 출력할 정보를 선택하면 된다. Text 에 속성 '페이지:'를 주고 Data 에 qrsPageNumber 를 선택하면 출력할 때 'Page:1' 과 같이 출력된다.

중앙의 Detail 밴드는 실제 레코드를 읽어와 출력하는 본문에 해당되는 밴드이다. 여기에 세 개의 QRDBText 컴포넌트를 배치해 보자. 이 컴포넌트는 DB 에서 필드를 읽어와 문자열 형태로 보여준다. 어떤 테이블의 어떤 필드를 가져올 것인가는 두 가지 속성으로 지정해 준다. DataSet 에 Form1.Table1 을 연결해 주고 DataField 에 NAME, ADDR, TEL 을 각각 연결해 준다. 여기까지 작업하면 폼의 모양은 다음과 같이 될 것이다.



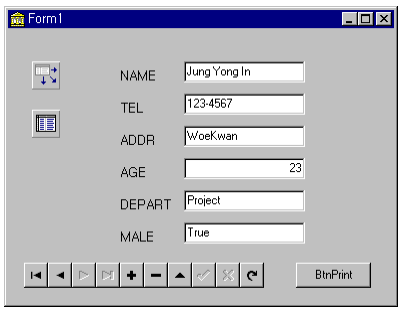
6 이제 간단하나마 리포트가 완성되었다. 제대로 출력되는지 확인해 보려면 종이에 찍어봐야겠지만 미리 보기로도 확인 가능하다. QuickRep 컴포넌트의 팝업 메뉴에서 Preview 를 선택하면 마치 실행중인 것처럼 미리 보기 화면이 나타날 것이다.

그림
미리 보기 화면




이 미리 보기 화면은 물론 우리가 만든 폼이 아니라 퀵 리포트가 제공해 주는 것이며 툴바, 상태란까지 다 갖추고 있다. 미리 보기 화면에서 인쇄 버튼을 누르면 곧바로 인쇄할 수도 있다.

7 마지막으로 메인 폼에서 이 폼을 호출할 수 있도록 해 주어야 한다. 네비게이터를 약간 왼쪽으로 밀고 보고서 출력 명령을 받기 위한 버튼을 하나 배치한다.




버튼의 Name 은 BtnPrint 로 주고 Caption 은 Print 로 설정한다. 그리고 이 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.PrintClick(Sender: TObject);
begin
```

```
  Form2.QuickRep1.Preview;
end;
```

코드의 내용은 아주 지극히 간단하다. Form2 폼에 있는 QuickRep1 컴포넌트의 Preview 메소드를 불러 미리 보기 화면을 호출한 것이다. Print 메소드를 호출하면 직접 인쇄할 수도 있지만 그보다는 미리 보기 화면으로 일단 보여주고 출력하도록 하는 것이 더 나을 것 같다.

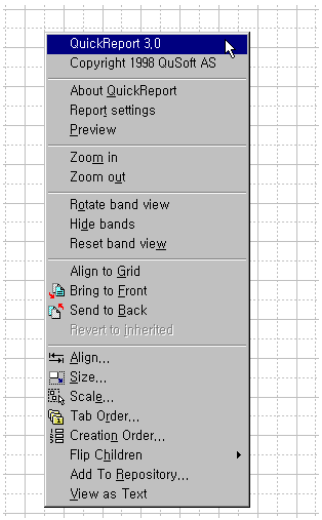
예제가 완성되었다. 실습 과정이 아주 복잡하다고 느껴질지도 모르겠지만 사실 몇 번만 실습해 보면 그리 어렵지는 않다. 보고서라는 복잡한 물건을 다루는데 이 정도라면 사실 그리 복잡하다고 할 수 없다. 위 예제를 몇 번 따라 만들어 보고 익숙해지도록 하자.

# 다. 퀵 리포트 컴포넌트

앞에서 만든 QReport 예제는 보고서 프로그래밍에 대한 개념을 설명하기 위해 아주 간단하게 작성했다. 게다가 만드는 방법만 설명했지 개별 컴포넌트에 대해서도 설명하지 않았다. 이 예제에서 사용한 퀵 리포트 컴포넌트는 다섯 개인데 이 다섯 개가 보고서 작성의 기본 컴포넌트이므로 일단 이 컴포넌트들을 개별적으로 연구해 보고 QReport 예제를 조금씩 개량해 보도록 하자.

## ■ QuickRep

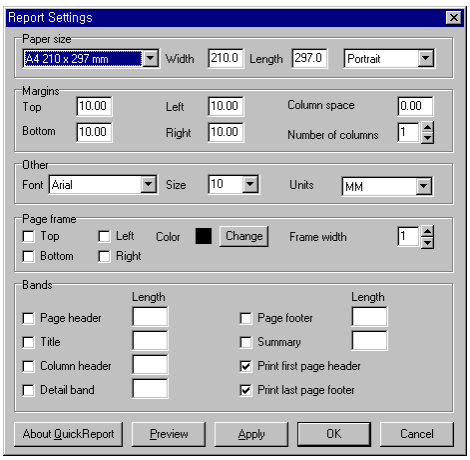
이 컴포넌트는 퀵 리포트로 보고서를 만드는 토대이다. 이 컴포넌트를 폼에 배치하면 엄청난 크기의 출력 용지 모양이 나타나는데 이 용지는 실제로 설정된 용지 크기에 맞추어진 것이다. 이 컴포넌트의 팝업 메뉴에는 아주 많은 기능이 들어 있다.




Preview 는 미리 보기 화면을 보여주는데 미리 보기 화면에서 보고서를 직접 인쇄할 수도 있다. Zoom in, Zoom out 명령은 디자인 중에 보고서 화면을 확대/축소시켜 주며 Hide bands 명령은 잠시 밴드를 안보이도록 해 준다.Report settings 를 선택하거나 아니면 퀵 리포트 컴포넌트를 더블클릭하면 다음과 같은 설정 대화상자가 나타난다.

**그림**

보고서 설정 대화상자




이 대화상자에는 보고서 용지의 크기, 여백, 폰트, 테두리 등을 설정할 수 있다. 컴포넌트가 제공하는 대화상자치고는 참 엄청나다는 생각이 든다.

여러 가지 지정 사항이 있는데 이 중 Font, Frame, Page, Bands 속성은 위의 세팅 대화상자에서 비주얼하게 지정할 수 있으며 직관적이므로 설명할

필요가 없을 것 같다. 이 컴포넌트의 제일 중요한 속성은 어떤 테이블을 인쇄할 것인가를 지정하는 DataSet 속성이다. 이 속성에 Table, Query 컴포넌트명을 지정해 주는데 인쇄시에 이 데이터 셋은 반드시 활성화되어 있어야 한다. 보고서에 인쇄되는 레코드의 순서는 데이터 셋이 정의하는 순서를 그대로 따르므로 만약 순서를 바꾸려면 데이터 셋에서 미리 바꾸어 놓거나 아니면 별도의 쿼리를 만들어 정렬시켜 주면 된다.

## ■ QRBand

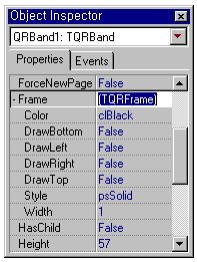
밴드는 보고서의 분절을 이루는 컴포넌트이며 여러 개의 밴드가 모여 완벽한 하나의 보고서를 만들어낸다. 밴드를 폼에 배치하면 띠 모양의 밴드가 배치된다. 밴드의 높이는 자유롭게 조정할 수 있지만 밴드의 폭은 용지 크기에 맞추어지므로 변경할 수 없다. 앞에서 예제를 만들어 봐서 알겠지만 밴드는 QRDBText, QRLabel 등의 출력 컴포넌트를 담는 컨테이너 컴포넌트이다.

밴드의 속성에는 Font, Size, Color 등의 일반적인 속성이 있다. 무엇보다 중요한 속성은 BaneType 속성인데 이 속성은 보고서내의 밴드 역할을 결정하며 이 속성에 따라 보고서내에서의 밴드 위치와 반복 여부가 결정된다.

| 타입 | 설명 |
|---|---|
| rbTitle | 보고서의 제목을 인쇄한다. 첫 번째 페이지에 한 번만 인쇄되며 PageHeader 밴드 바로 다음에 인쇄된다. |
| rbPageHeader | 모든 페이지의 윗부분에 인쇄되는 머리말이다. 첫 페이지에도 머리말을 출력할 것인가는 QuickRep.Option.FirstPageHeader 속성에 따라 달라진다. |
| rbDetail | 테이블의 각 레코드를 출력한다. 이른바 테이블의 내용을 출력하는 본체 부분이라고 할 수 있다. |
| rbPageFooter | 모든 페이지의 끝부분에 인쇄되는 꼬리말이다. |
| rbSummery | 마지막 페이지의 Detail 밴드 다음에 출력된다. |
| rbGroupHeader | TQRGroup, TQRSubDetail에 의해 사용되는 헤더 밴드 |
| rbGroupFooter | TQRGroup, TQRSubDetail에 의해 사용되는 테일 밴드 |
| rbSubDetail | TQRSubDetail에 의해 사용되는 특별한 밴드이며 직접 이 밴드를 설정해서는 안된다. |
| rbColumnHeade | 보고서의 각 컬럼 선두에 출력되는 밴드이며 여러 단으로 구성 |

| | |
|---|---|
| r | 된 보고서에 필드명을 출력하고자 할 때 사용한다. |
| rbOverlay | 사용되지 않음 |
| rbChild | TQRChildBand에 의해 사용되는 특별한 밴드이며 직접 이 밴드를 설정해서는 안된다. |

Frame 속성은 밴드의 상하좌우 테두리 선을 긋도록 하며 다음과 같은 세부 속성을 가지고 있다.




각 변에 줄을 그을 것인가는 Draw* 속성으로 선택하며 Color 는 선의 색상, Width 는 선의 굵기, Style 은 선의 모양을 결정한다.

## ■ QRLabel

퀵 리포트 컴포넌트 중에 극단적으로 간단한 컴포넌트이며 보고서상에 문자열을 출력할 때 사용한다. 사용하는 방법은 표준 레이블 컴포넌트와 거의 동일하다고 할 수 있다. Caption, Font, AutoSize, Color 등의 일반적인 속성을 사용하여 예쁘게 만들어 주면 된다. 레이블과 다른점이라면 Frame 속성이 있어 테두리 장식을 할 수 있다는 정도뿐이다. 특별한 주의사항도 없고 사용하는데 어려울 것도 없는 정말 만만한 컴포넌트이다.

## ■ QRDBText

데이터 베이스의 필드를 읽어와 인쇄해 주는 컴포넌트이다. DBEdit 와 동작이나 모양이 거의 유사하며 문자열은 물론 정수형, 논리형, 메모, 심지어 계산된 필드까지 출력할 수 있다. DataSet 속성에 테이블, 쿼리를 지정해 주고 DataField 에 출력할 필드명을 지정해 주면 된다. 그외 Color, Font, Frame, Alignment 등의 속성을 사용하여 장식을 좀 해 주면 된다. 출력 양

식은 Mask 속성으로 지정할 수 있는데 이 속성이 정의되어 있지 않으면 필드의 DisplayFormat 지정을 따른다.

## ■ QRSysData

이 컴포넌트는 테이블의 필드에서 얻을 수 없는 시스템 정보를 출력해 준다. 예를 들어 보고서 제목, 페이지 번호, 시간, 날짜 등이다. 이 정보를 출력하고자 하는 곳, 보통 머리말이나 꼬리말 정도에 이 컴포넌트를 배치해 두고 Data 속성으로 출력하고자 하는 정보를 선택해 준다.

| 값 | 설명 |
|---|---|
| qrsColumnNumber | 현재 컬럼 번호 |
| qrsDate | 오늘 날짜 |
| qrsDateTime | 현재 날짜와 시간 |
| qrsDetailCount | 레코드 총 수. 여러 개의 테이블이 있을 경우는 마스터 테이블의 레코드 수가 된다. SQL 서버를 사용할 경우는 이 값을 알 수 없는 경우도 있다. |
| qrsDetailNo | 현재 레코드 번호 |
| qrsPageNumber | 현재 페이지 번호 |
| qrsPageCount | 보고서의 총 페이지 수. 단 이 값을 알기 위해서는 이단계 보고서 작성을 해야 하는데 때로는 엄청난 시간이 요구된다. |
| qrsReportTitle | 보고서 제목(TQuickRep.ReportTitle) |
| qrsTime | 현재 시간 |

이 컴포넌트는 Data 속성이 지정하는 정보를 조사해서 출력한다. Text 속성은 이 컴포넌트가 조사한 정보의 앞부분에 붙여질 문자열이며 출력된 정보에 대한 설명을 보여준다. 그냥 12라고 출력되면 왜 12인지 알 수 없으므로 앞에 문자열을 달아 '현재 페이지:12'라고 출력해 주는 것이 좋다. 이 두 속성 외에는 Font, Color, Transparent 등의 속성이 있다.

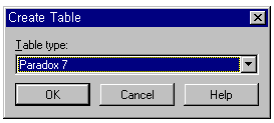
# 15-5 고급 DB

## 가. 회원 관리

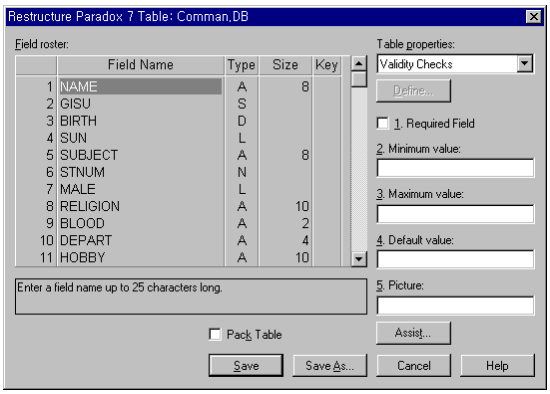
15jang
Hoewon

아무리 이론을 잘 나열해도 이론만 가지고는 실무를 배우기에 부족함이 많다. 이론을 더 잘 이해하려면 완성된 예제를 분석해 보는 것이 좋으며 확실한 자신의 지식으로 만들고 싶다면 직접 실습해 보는 것이 제일 좋다. 국내 대중 통신망이나 인터넷을 통해 공개된 소스를 구해 분석해 본다거나 잡지나 강좌를 통해 여러 가지 기법을 익혀 보기를 권한다. 여기서는 여러분들이 분석해 볼 만한 아주 간단한 회원 관리 프로그램을 작성해 보았다. 대단한 프로그램은 아니지만 DB 프로그램을 어떻게 작성하는가 감을 잡기에는 충분하리라 생각한다. 단계를 따라 실습을 해 보고 배포 CD에 있는 예제를 직접 분석해 보기 바란다.

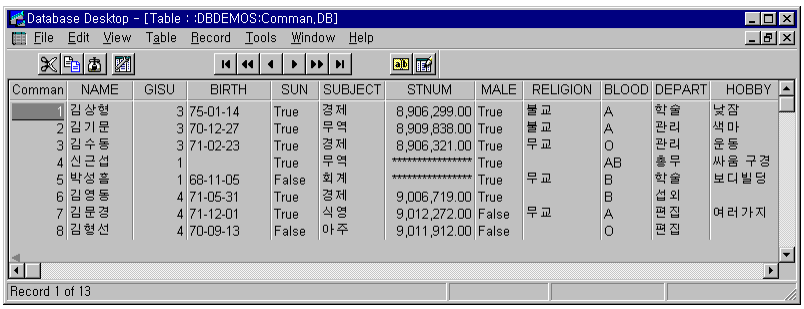
우선 해야 할 일은 회원 관리에 적합한 테이블부터 디자인하는 것이다. 어떤 테이블이 필요하고 각 테이블은 어떤 필드를 포함해야 하는지를 디자인 단계에서 결정해야 한다. 회원 관리의 경우는 회원들의 신상 명세에 대한 테이블만 하나 있으면 된다. DBD를 실행한 후 테이블을 만들어 보자. File/Table/New를 선택한다.




테이블 형식은 Paradox 7으로 선택하고 다음과 같은 필드를 만들어 준다. 포함하는 필드가 무려 20개나 되는데 정확한 필드 목록은 배포 CD의 15jang/Comman.db 파일을 열어 살펴보기 바란다.

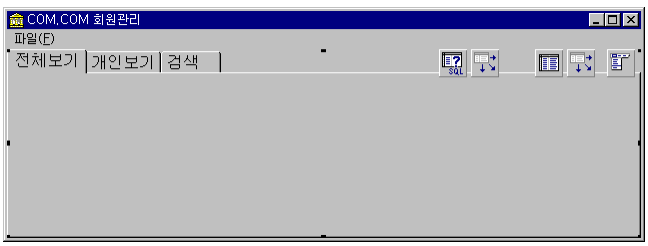



테이블을 만든 후에 이 테이블을 Comman.db로 저장하되 DBDEMOS 디렉토리에 저장하는 것이 좋다. 그리고 데이터를 입력한다.




프로그램을 다 만든 후에도 입력할 수 있으므로 디자인 단계에서 다 입력해 놓을 필요는 없다.

2 새 프로젝트를 시작하고 프로그램의 전체 윤곽을 작성한다. 회원 전체 보기, 개인 보기, 검색 세 개의 페이지를 가지도록 디자인하였으며 탭 노트북으로 세 개의 페이지를 만들었다. 그리고 필요한 컴포넌트들을 배치한다.




메인 메뉴에는 파일/끝내기 메뉴만 정의하였다. DB 검색을 위해 Query1,

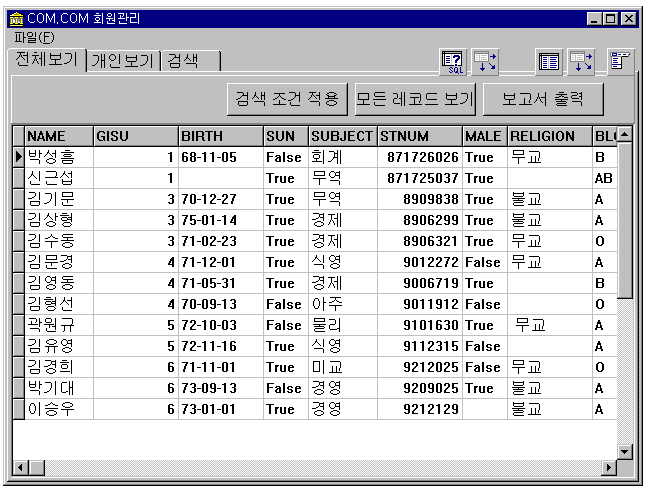
DataSource1을 배치하였고 조회를 위해 Table1, DataSource1을 배치하였다. 그리고 각 데이터 베이스 액세스 컴포넌트는 속성으로 상호 연결해 두었다. 검색식을 기억하기 위해 sqlstr이라는 문자열을 전역 변수로 선언해 두었다.

```
var
  MainForm: TMainForm;
  sqlstr:string;      {SQL 명령}
```

이 문자열은 폼이 생성될 때인 FormCreate에서 테이블 전체 내용을 보여주도록 초기화하였다.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  sqlstr:='select * from comman.db order by GISU,NAME';
end;
```

이제 첫 번째 페이지인 전체 보기를 디자인해 보자. DBGrid를 사용하여 표 형식으로 회원 목록을 보여주도록 하되 검색 결과도 같이 보여주기로 한다. 패널을 alTop으로 배치하고 패널 안에는 세 개의 버튼을 두었으며 나머지 영역은 DBGrid를 alClient 정렬하였다. DBGrid는 쿼리와 연결된 DataSource1과 연결되어 있어 쿼리 결과를 보여준다.




| NAME | GISU | BIRTH | SUN | SUBJECT | STNUM | MALE | RELIGION | BL |
|---|---|---|---|---|---|---|---|---|
| 박성흠 | 1 | 68-11-05 | False | 회계 | 871726026 | True | 무교 | B |
| 신근섭 | 1 | | True | 무역 | 871725037 | True | | AB |
| 김기문 | 3 | 70-12-27 | True | 무역 | 8909838 | True | 불교 | A |
| 김상형 | 3 | 75-01-14 | True | 경제 | 8906299 | True | 불교 | A |
| 김수동 | 3 | 71-02-23 | True | 경제 | 8906321 | True | 무교 | O |
| 김문경 | 4 | 71-12-01 | True | 식영 | 9012272 | False | 무교 | A |
| 김영동 | 4 | 71-05-31 | True | 경제 | 9006719 | True | | B |
| 김형선 | 4 | 70-09-13 | False | 아주 | 9011912 | False | | O |
| 곽원규 | 5 | 72-10-03 | False | 물리 | 9101630 | True | 무교 | A |
| 김유영 | 5 | 72-11-16 | True | 식영 | 9112315 | False | | A |
| 김경희 | 6 | 71-11-01 | True | 미교 | 9212025 | False | 무교 | O |
| 박기대 | 6 | 73-09-13 | False | 경영 | 9209025 | True | 불교 | A |
| 이승우 | 6 | 73-01-01 | True | 경영 | 9212129 | | 불교 | A |

DBGrid에 어떤 레코드가 출력될 것인가는 두 버튼으로 명령한다. 버튼의

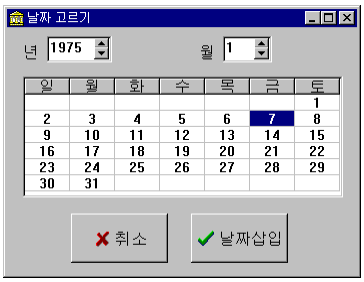
OnClick 이벤트 핸들러는 다음과 같이 작성되어 있다.

```
{전체 보기에서 검색 조건을 적용하여 조건에 맞는
레코드만 보여준다.}
procedure TMainForm.Button1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Sql.Clear;
  Query1.Sql.Add(sqlstr);
  Query1.Open;
end;

{전체 보기에서 검색 조건을 무시하고 모든 레코드를 보여준다.}
procedure TMainForm.Button2Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Sql.Clear;
  Query1.Sql.Add('select * from comman.db order by GISU,NAME');
  Query1.Open;
end;
```

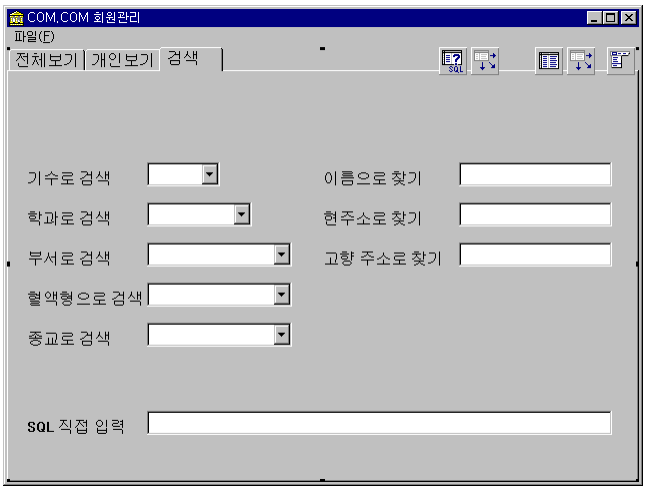
검색 조건 적용 버튼을 누르면 전역 변수 sqlstr 명령을 실행하여 검색식을 적용하였고 모든 레코드 보기 버튼을 누르면 검색식을 무시하고 모든 레코드가 나타나도록 하였다. sqlstr 명령은 세 번째 검색 페이지에서 변경하게 된다.

4 두 번째 페이지는 개인에 대한 정보를 조회해 본다. 한 레코드의 정보를 보여주도록 데이터 컨트롤만 잔뜩 배치하였으며 각 데이터 컨트롤은 Table1에 연결된 DataSource2에 연결되어 있다.


이 페이지에서 네비게이터로 레코드간의 이동이나 편집을 할 수 있다. 데이터 컨트롤에 편집 기능이 내장되어 있으므로 별도로 작성해야 할 코드는 없다. 단 날짜를 편리하게 선택할 수 있도록 날짜 선택 폼을 만들어 스피드 버튼에 연결해 두었다.




5 세 번째 페이지에서 검색 조건에 따라 SQL문을 작성하도록 한다. 다양한 검색 조건을 줄 수 있도록 하였다.




왼쪽의 콤보 박스는 문자열을 직접 비교하는 SQL문을 만들며 오른쪽의 에디트는 일부 분자열을 검색하는 SQL문을 만든다.

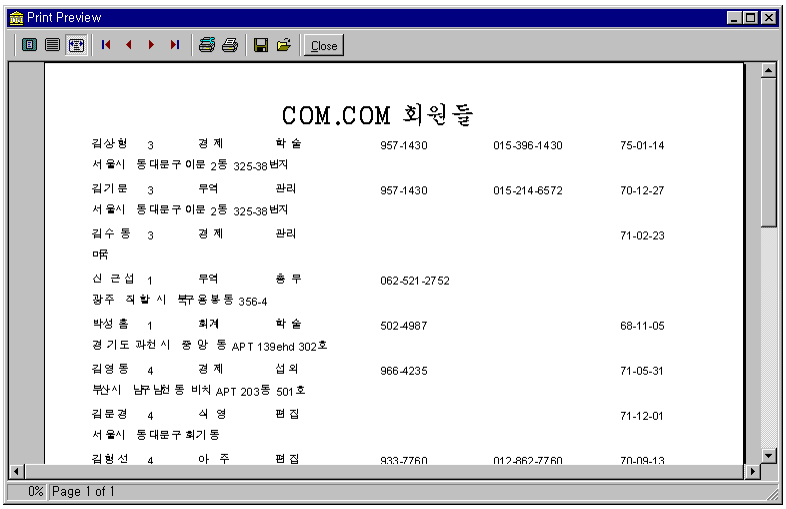
```
{검색 조건이 바뀌면 SQL 명령을 다시 작성한다.
바뀐 검색 조건을 실행하기 위해 전체 보기 페이지로 전환한다.}
{기수로 찾기}
procedure TMainForm.ComboBox1Change(Sender: TObject);
begin
sqlstr:='select * from comman.db where GISU='+
    ComboBox1.Items[ComboBox1.ItemIndex]+
    ' Order By GISU,NAME';
TabbedNotebook1.PageIndex:=0;
end;

{학과로 찾기}
procedure TMainForm.ComboBox2Change(Sender: TObject);
begin
sqlstr:='select * from comman.db where SUBJECT='''+
    ComboBox2.Items[ComboBox2.ItemIndex]+
    ''' Order By GISU,NAME';
TabbedNotebook1.PageIndex:=0;
end;

{부서로 찾기}
procedure TMainForm.ComboBox3Change(Sender: TObject);
begin
sqlstr:='select * from comman.db where DEPART='''+
    ComboBox3.Items[ComboBox3.ItemIndex]+
    ''' Order By GISU,NAME';
```

```
TabbedNotebook1.PageIndex:=0;
end;

{혈액형으로 찾기}
procedure TMainForm.ComboBox4Change(Sender: TObject);
begin
sqlstr:='select * from comman.db where BLOOD='''+
    ComboBox4.Items[ComboBox4.ItemIndex]+
    ''' Order By GISU,NAME';
TabbedNotebook1.PageIndex:=0;
end;

{종교로 찾기}
procedure TMainForm.ComboBox5Change(Sender: TObject);
begin
sqlstr:='select * from comman.db where RELIGION='''+
    ComboBox5.Items[ComboBox5.ItemIndex]+
    ''' Order By GISU,NAME';
TabbedNotebook1.PageIndex:=0;
end;

{일부 문자열 검색 SQL 문을 작성한다. 페이지는 전환하지 않는다.}
{이름으로 찾기}
procedure TMainForm.Edit1Change(Sender: TObject);
begin
  sqlstr:='select * from comman.db where NAME like ''%'+
    Edit1.Text+'%'' Order By GISU,NAME';
end;

{고향으로 찾기}
procedure TMainForm.Edit2Change(Sender: TObject);
begin
  sqlstr:='select * from comman.db where NADDR like ''%'+
    Edit2.Text+'%'' Order By GISU,NAME';
end;

{현주소로 찾기}
procedure TMainForm.Edit3Change(Sender: TObject);
begin
  sqlstr:='select * from comman.db where HADDR like ''%'+
    Edit3.Text+'%'' Order By GISU,NAME';
end;

{SQL 문 직접 입력}
procedure TMainForm.Edit4Change(Sender: TObject);
begin
```

```
  sqlstr:=Edit4.Text;
end;
```

만들어진 SQL문은 전역 변수 sqlstr에 저장되며 첫 번째 페이지로 전환하여 결과를 보여주도록 하였다.

6 보고서 폼을 만든다. File/New Form으로 새 폼을 만든 후 이 폼에 QuickRep 컴포넌트를 배치하고 세 개의 밴드와 QRDbText, QRSysData 등의 컴포넌트를 배치하였다. 보고서는 첫 번째 페이지의 보고서 보기 버튼에 의해 호출된다.




# 나. DB 폼 마법사

15jang DbFormWiz

데이터 베이스 프로그램은 일반 프로그램에 비해 만들기가 무척 까다롭다. 이것저것 연결해 줘야 할 것도 많고 암기해야 할 사항이 많아 초보자가 단기간에 쉽게 배우기에는 무리한 면이 많다. 게다가 익숙한 사용자라 하더라도 매번 데이터 베이스 프로그램을 만들 때 똑같은 과정을 반복해야 하므로 이 또한 무척 짜증나는 일이다. 그래서 델파이에서는 데이터 베이스 폼을 자동으로 만들어 주는 마법사를 제공하며 프로그래머를 위해 다음과 같은 일을 대신해 준다.

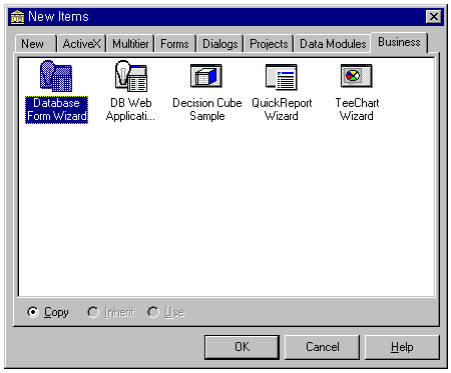
- 데이터 베이스 프로그램을 만드는 데 필요한 컴포넌트를 폼에 배치해 준다.
- Table 이나 Query 를 지정한 데이터 베이스에 연결해 준다.

■ DataSource 를 Dataset 과 연결해 준다.
■ Query 에 SQL 명령을 작성해 준다.

DB 폼 마법사는 폼을 만드는 데 필요한 정보를 질문 형식으로 사용자에게 물어보므로 마우스로 대답만 몇 번 해주면 된다. 앞에서 만들었던 예제를 마법사를 사용해서 만들어 보자.

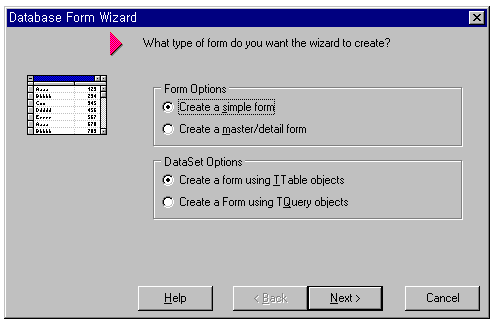
① 일단 모든 프로젝트를 닫는다. 물론 꼭 그럴 필요는 없지만 깨끗한 상태에서 폼을 만든 후 프로젝트를 만들기로 한다. File/Close All 명령을 선택하면 된다.

② File/New 메뉴를 선택한 후 오브젝트 창고 대화상자가 나타나면 Business 페이지에서 Database Form Wizard를 선택한다.




③ 다음과 같은 대화상자를 열어 첫 번째 질문을 해 온다.

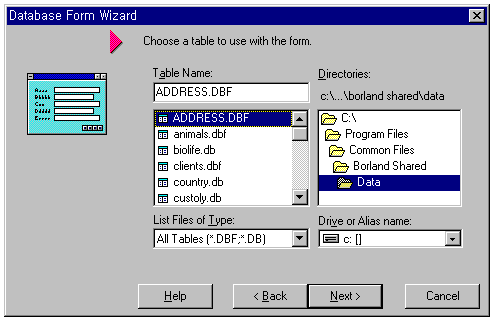
그림
DB 폼 마법사




하나의 테이블을 사용할 것인가 두 개의 테이블을 연결할 것인가를 물어오며 Table을 사용할 것인가 Query를 사용할 것인가를 물어온다. 질문에 대

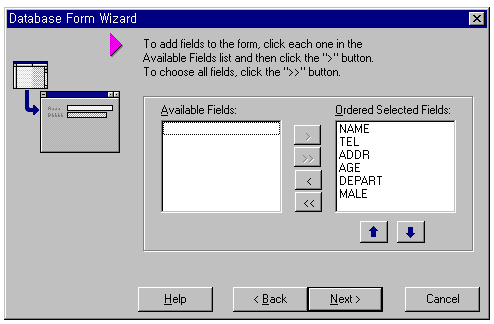
답한 후 Next > 버튼을 누르되 일단 디폴트를 그대로 받아들이도록 하자.

④ 사용할 DBF 파일을 선택해 줄 것을 요구한다.



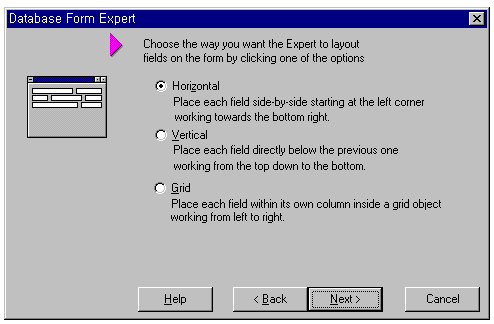

디렉토리를 이동하여 ADDRESS.DBF를 선택하도록 하자.

⑤ 폼에 어떤 필드를 포함시킬 것인가를 물어오는데 특별히 제외시킬 필드가 없으면 >> 버튼을 눌러 모든 필드를 포함시킨다.



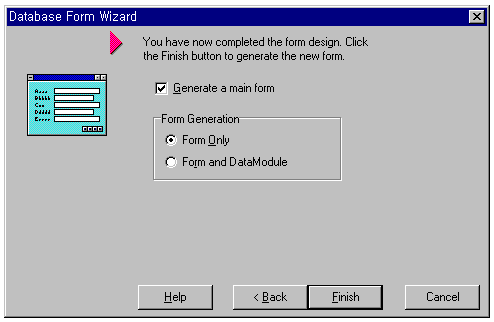

↑,↓ 버튼을 사용하여 필드의 순서를 조정할 수도 있다. 순서 조정이 끝나면 Next > 버튼을 누른다.

⑥ 전체적인 구조를 정해준다.



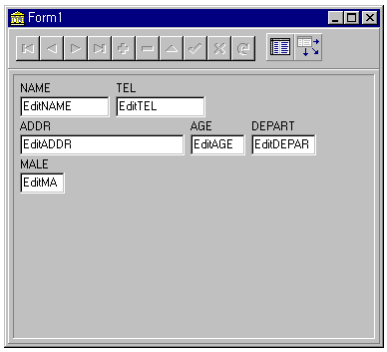
라디오 버튼을 선택함에 따라 좌측에 샘플이 나타나므로 원하는 형태를 쉽게 결정할 수 있다.

⑦ Finish 버튼을 눌러 폼을 만든다.




잠시후 다음과 같은 폼이 나타난다.

그림
마법사가 만들어 낸 폼




마법사가 만들어 내는 것은 폼이지 프로젝트가 아니다. 그래서 이 폼 자체만으로는 실행할 수가 없으며 별도의 프로젝트를 만들어 이 폼을 불러서 사용해야 한다. 그러기 위해서는 폼을 디스크에 일단 저장해야 한다. 물론 프로젝트가 열려 있는 상태에서는 프로젝트를 저장하면 된다. 배포 CD 의 DbFormWiz 디렉토리에 마법사로 만든 프로젝트가 있으니 한번쯤 구경해 보기 바란다.

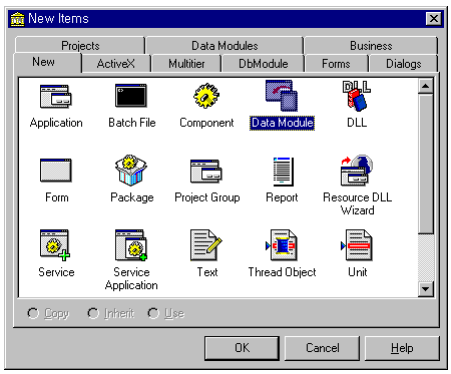
마법사가 만들어 내는 폼은 사람의 손으로 직접 만든 폼과 전혀 다를 바가 없다. 폼이 만들어지고 난 후에 코드를 추가하거나 컴포넌트를 재배치하는 것은 완전히 자유롭다. 마법사에게 의뢰하여 대충의 폼 모양을 만들고 그런 후에 마음에 드는대로 뜯어고치는 것이 지루한 작업을 좀 더 빠르게 하는 한 방법이다.

# 다. 데이터 모듈

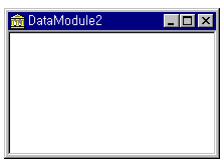
데이터 모듈은 비가시적 컴포넌트를 담는 컨테이너이다. DataSource, Table, Query 등의 데이터 액세스 컴포넌트가 모두 실행중에는 보이지 않는 컴포넌트인데 주로 이런 컴포넌트를 담는다. 폼에 직접 데이터 액세스 컴포넌트를 담을 수도 있지만 다수 개의 폼에서 동시에 데이터 액세스 컴포넌트를 참고하고자 할 경우는 데이터 모듈에 비가시적 컴포넌트를 두는 것이 권장된다.

15jang
DbModule

일단 어떻게 쓰는 컴포넌트인지 알아보기 위해 예제부터 만들어 보자. 일단 새 프로젝트를 시작한다. 데이터 베이스 프로그램을 만들려면 당연히 데이터 소스, 테이블(또는 쿼리), 데이터 컨트롤을 폼에 배치해야 할 것이나 이번에는 그렇게 하지 말고 데이터 모듈을 사용해 보자. 폼은 그대로 두고 File/New 명령을 내린 후 Data Module 을 선택해 보자.




그러면 다음과 같은 빈 데이터 모듈을 만들어 줄 것이다.



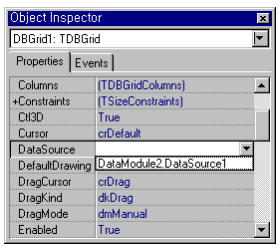

바탕색이 흰색일 뿐이지 사실 폼과 별반 다를바가 없다. 여기에 컴포넌트를 배치해 보자. DataSource 와 Table 을 하나씩 배치한다. 마치 폼에 놓듯이 컴포넌트를 배치할 수 있다. 단 데이터 모듈에는 비가시적 컴포넌트만을 배치할 수 있으며 버튼이나 에디트같은 컨트롤은 배치할 수 없다. Table 의 DatabaseName 에 DBDEMOS 를 지정하고 TableName 에 Address.dbf 를

지정하고 DataSource.DataSet 에 Table1 을 주도록 하자. 그러면 테이블을 연결하는 통상적인 절차가 끝난다.

이제 폼에는 DBGrid 만 하나 배치해 두고 데이터 모듈에 있는 DataSource 를 참조하기만 하면 된다. 단 폼간에 uses 문으로 상호 인식할 수 있어야 하므로 일단 이 프로젝트를 DbModule.dpr 로 저장하되 폼은 DbModule_f.pas 로 데이터 모듈은 DbModule_f2.pas 로 이름을 준다. 그리고 폼의 implementation 부에 다음 두 줄을 적어주면 폼에서 데이터 모듈을 참조할 수 있게 된다.

```
implementation

{$R *.DFM}

uses
  DbModule_f2;
```

이제 폼에 DbGrid 를 배치한 후 Align 속성을 alClient 로 설정하여 폼에 가득 채우도록 하자. 그리고 DBGrid 의 DataSource 속성을 설정하되 콤보박스를 열어보면 이미 DataModule2.DataSource1 이 목록에 있을 것이다. 이 목록을 선택해 주자.




마지막으로 데이터 모듈의 Table1.Active 속성을 True 로 바꾸어 주면 그리드에 테이블 내용이 나타날 것이다. 예제가 완성되었다.

예제는 쉽게 이해가 되는데 왜 이런 데이터 모듈을 사용하며 장점은 무엇일까? 데이터 모듈은 비가시적 컴포넌트들을 되도록이면 한곳에 모아 집중시키기 위해 사용한다. 규모가 큰 데이터 베이스 프로그램의 경우 테이블이 수십 개가 되고 질의도 다양하게 만들어 쓰기 때문에 Table, Query 컴포넌트의 수요가 많다. 이런 컴포넌트들을 폼마다 배치해 놓고 사용한다면 무척 난잡해질 것이며 유지, 보수에도 굉장한 노력이 들어갈 것이다. 이런 컴포넌트들을 데이터 모듈에 모아놓고 폼에서는 데이터 모듈을 참조하도록 하면

훨씬 더 편리하다.

데이터 모듈은 비가시적 컴포넌트를 모아놓는 집합소이기 때문에 그 자체도 실행중에는 보이지 않는다. 뒤에 숨어서 비가시적 컴포넌트를 제공해 주며 또한 비가시적 컴포넌트의 이벤트 핸들러에 대한 대리 소유주가 되어줄 뿐이다. 사실 위와 같은 간단한 예제의 경우는 폼에 직접 데이터 소스, 테이블을 놓는 것이 훨씬 더 편리하겠지만 삼단 분할 프로그램같은 경우의 어플리케이션 계층을 위해서는 데이터 모듈이 필수적이다.

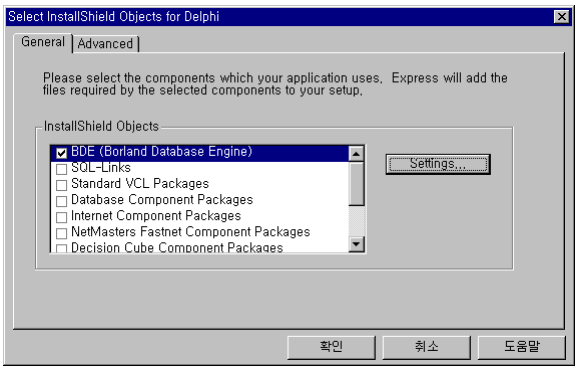
# 라. BDE 의 배포

델파이는 별도의 런타임 라이브러리가 불필요한 독립된 실행 파일을 만들어 낸다. 그래서 설치 프로그램을 만들 필요없이 실행 파일만 복사해서 가져가도 실행할 수 있기 때문에 프로그램을 배포하기가 무척 편리하다. 그런데 데이터 베이스 프로그램은 예외적으로 반드시 설치 프로그램을 작성해야만 한다.

데이터 베이스 프로그램은 아무리 간단한 프로그램이라도 복사해서 다른 컴퓨터로 가져가면 제대로 실행이 되지 않는다. Addr.exe 를 복사해서 테스트 해 보면 이런 사실을 쉽게 확인할 수 있을 것이다. 왜 그런가하면 데이터 베이스 프로그램은 홀로 실행되는 것이 아니라 BDE 와 함께 실행되기 때문에 BDE 없이는 아무 것도 할 수 없기 때문이다. 또한 런타임 패키지를 사용한 프로젝트도 마찬가지로 단독 실행 파일만으로는 실행할 수 없다.

이런 부속 파일들이 있을 때는 으레 설치 프로그램을 만들어서 부속 파일이 제 위치에 복사되도록 해 주어야 하며 시스템 레지스트리도 적절히 수정해 주어야 한다. 이런 작업은 델파이와 함께 제공되는 인스톨 쉴드를 사용하면 어렵지 않게 할 수 있다. 인스톨 쉴드의 두 번째 대화상자에서 BDE 를 선택해 주면 설치 프로그램에 BDE 를 같이 포함해 준다. 또한 BDE 아래쪽에는 런타임 패키지를 선택하는 옵션들이 있는데 이 옵션들도 선택해 주면 설치 프로그램에 패키지도 같이 포함된다.

그림
인스톨 쉴드




인스톨 쉴드의 자세한 사용방법은 도움말을 참고하기 바라며 아뭏든 여기서는 데이터 베이스 프로그램은 단독으로 배포할 수 없다는 것만 꼭 알아두도록 하자.

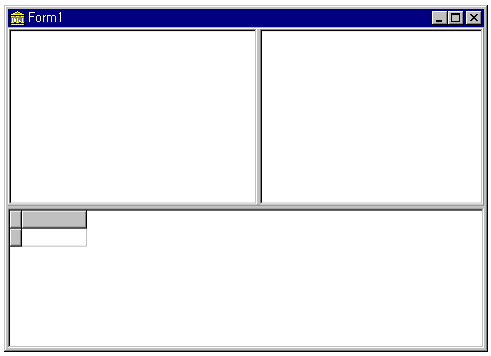
# 마. DB 탐색기

지금까지 데이터 베이스 프로그램을 만들면서 대상 DB 파일을 테이블 컴포넌트의 DatabaseName 속성과 TableName 속성에 대입해 줌으로써 디자인중에 설정해 주었다. 만약 실행중에 대상 DB 를 변경하려면 이 두 속성을 바꾸어주면 된다. 그런데 문제는 어떤 데이터 베이스가 시스템에 있고 각 데이터 베이스에는 또 어떤 테이블이 있는가를 조사해야 한다는 것이다. 이런 목적을 위해 델파이는 Session 이라는 전역 오브젝트를 제공한다. 이 오브젝트는 DBTables 유닛에 정의되어 있으므로 별도의 선언없이도 테이블을 폼에 배치하기만 하면 사용할 수 있다. 이 오브젝트에는 다음과 같은 메소드들이 있어 실행중에 시스템의 데이터 베이스 정보를 구할 수 있게 해 준다.

| 메소드 | 설명 |
|---|---|
| GetDatabaseNames | 데이터 베이스 목록을 얻는다. |
| GetTableNames | 데이터 베이스에 포함된 테이블의 목록을 얻는다. |
| GetDriverNames | 드라이버의 목록을 얻는다. |
| AddAlias | 앨리어스를 추가한다. |
| AddDriver | 드라이버를 추가한다. |

15jang
DbExplorer

이 메소드들만 잘 사용하면 델파이가 제공하는 SQL 탐색기같은 프로그램은 쉽게 만들 수 있다. 우선 새 프로젝트를 시작하고 패널, 리스트 박스, 스플리터, DBGrid 등을 배치하여 다음과 같이 만든다.




폼의 모양이 좀 복잡해 보이지만 다음 순서대로 따라서 배치하기만 하면 된다.

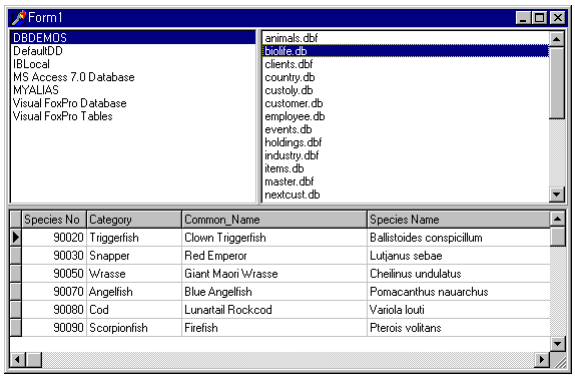
① 패널을 배치하고 alTop 정렬한다.
② 그 위에 리스트 박스를 배치하고 alLeft 정렬한다.
③ 리스트 박스 옆에 스플리터를 배치한다.
④ 그 옆에 리스트 박스를 배치하고 alClient 정렬한다.
⑤ 패널 아래쪽에 스플리터를 배치하고 alTop 정렬한다.
⑥ 폼의 나머지 영역에 DBGrid 를 배치하고 alClient 정렬한다.

그리고 데이터 소스 컴포넌트와 테이블 컴포넌트를 배치한 후 DataSource1.DataSet 에 Table1 을, DBGrid1.DataSource 에 DataSource1 을 대입해 준다. 여기까지 작성하면 테이블 컴포넌트만 데이터 베이스에 연결되어 있지 않을 뿐이며 나머지 컴포넌트들은 상호 연결되었다. 테이블을 실제 DB 와 연결하는 일은 코드에서 수행한다. 다음 세 개의 핸들러를 작성한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.GetDatabaseNames(ListBox1.Items);
end;
```

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Session.GetTableNames(ListBox1.Items[ListBox1.ItemIndex],
    '',True, False, ListBox2.Items);
end;

procedure TForm1.ListBox2Click(Sender: TObject);
begin
  Table1.Active:=False;
  Table1.DatabaseName:=ListBox1.Items[ListBox1.ItemIndex];
  Table1.TableName:=ListBox2.Items[ListBox2.ItemIndex];
  Table1.Active:=True;
end;
```

폼이 생성될 때 GetDataBaseNames 메소드로 시스템에 설치된 데이터베이스의 목록(=앨리어스 목록)을 조사해 ListBox1 에 채우고 ListBox1 에서 앨리어스가 선택되면 GetTableNames 메소드로 테이블의 목록을 조사해 ListBox2 에 채웠다. ListBox2 에서 항목이 선택되면 테이블에 DatabaseName 속성과 TableName 속성을 대입해 주고 테이블을 활성화시켜 테이블을 읽어오도록 하였다.




테이블을 변경하기 전에 Table1 을 반드시 먼저 닫아 주고 변경 후 다시 열어 주어야 함을 유의하자. 읽어온 테이블은 아래쪽의 DBGrid 에 나타난다.

# 15-6 마스터 디테일

## 가. 마스터 디테일 구조

지금까지 작성한 예제에서는 모두 하나의 테이블만을 사용했다. 그런데 실전에서는 테이블 하나로 할 수 있는 일이 사실 별로 없다. 간단한 업무에도 수 개의 테이블이 필요하며 복잡한 대규모 프로젝트에는 수백개의 테이블이 사용되기도 한다. 여러 개의 테이블이 사용되는 가장 간단한 예인 마스터/디테일 관계에 대해 살펴보자.

만약 도서 대출 상황을 관리하는 도서 대여 프로그램을 작성한다고 해 보자. 이 프로그램은 누가 무슨 책을 빌려갔는지를 관리하는 것이 목적이다. 만약 하나의 테이블로 이 정보들을 보관하려면 다음과 같이 테이블을 디자인해야 할 것이다.

| 날짜 | 학번 | 이름 | 학과 | 학년 | 서명 | 반납일 | ..... |
|---|---|---|---|---|---|---|---|
| 98-10-1 | 8906299 | 김상형 | 경제 | 1 | 삼국지 | 98-10-9 | ..... |
| | | | | | | | |

하나의 대출 상황에 대한 모든 것을 다 담도록 레코드를 디자인한 후 책을 빌려갈 때마다 정보를 입력해 넣으면 된다. 그런데 만약 한 학생이 두 번 책을 빌려갈 경우는 똑같은 정보들을 반복해서 기재해야 할 것이다. 이는 논리적으로나 시간적으로나 어느 면으로 보나 낭비가 분명하며 불합리하다. 그래서 하나의 테이블에 모든 정보를 다 담지 말고 둘로 쪼개어 각각 저장한다. 우선 대출해 가는 학생의 정보를 저장하는 다음과 같은 테이블을 만든다.

| 학번 | 이름 | 학과 | 학년 | 성별 | ..... |
|---|---|---|---|---|---|
| | | | | | |

그리고 대출 상황에 대한 테이블을 따로 만들어 준다.

| 대출자 | 서명 | 날짜 | 반납 예정일 | ..... |
|---|---|---|---|---|
| | | | | |

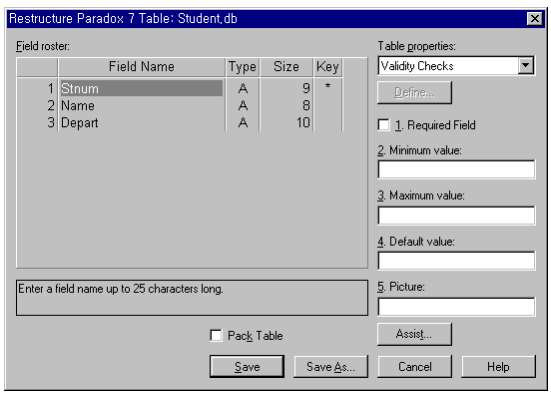
이렇게 테이블을 두 개 만들어 놓고 두 테이블을 연결해서 사용할 수 있다. 예를 들어 대출 상황 테이블에서 대출자에 대해 알고 싶으면 대출자 테이블을 보면 되고 특정 학생이 빌려간 책의 목록을 알고 싶으면 대출 상황 테이블을 보면 될 것이다. 이런 구조를 마스터/디테일 구조라고 하는데 위 예에서는 학생 테이블이 마스터가 되고 대출 상황 테이블이 디테일이 된다.

# 나. BookLoan

15jang
BookLoan

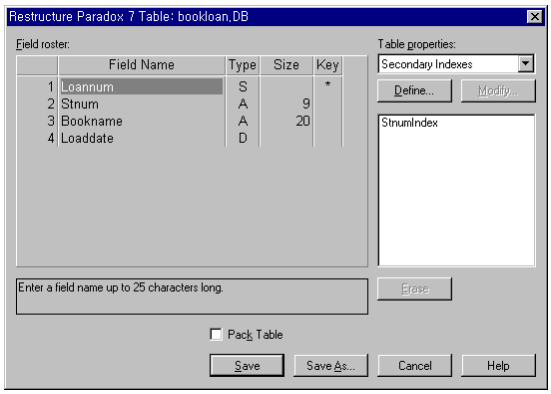
그럼 위에서 예로 든 도서 대출 프로그램을 직접 만들어 보자. 조금만 복잡해지면 실습 과정이 너무 길어져 비효율적이므로 테이블 구조를 최대한 간단하게 디자인하도록 하였다. 하지만 미스터/디테일 구조의 개념을 살펴보는데는 지장이 없을 것이다. 다음 단계를 따라 실습을 같이 진행하거나 아니면 머리속으로라도 제작 과정을 따라 해 보자.

프로젝트를 만들기 전에 일단 테이블부터 디자인한다. 먼저 마스터 테이블에 해당하는 Student.db 를 만들고 여기에 학생 정보를 담도록 하자. DBD 를 실행한 후 새 테이블을 만든다. 테이블 타입은 Paradox 로 선택하고 레코드 구조는 다음과 같이 작성하였다.

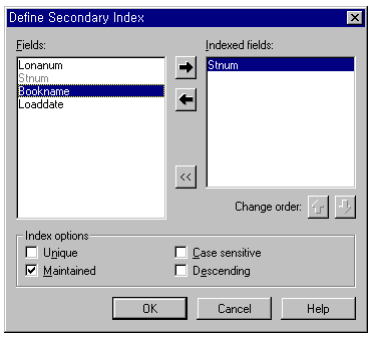



학번, 이름, 학과 세 개의 필드를 만들었으며 Stnum 필드를 Key 필드로

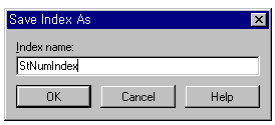
지정하였다. 다음은 책 대출 정보를 담는 Bookloan.db 를 만든다. 역시 Paradox 테이블로 만들었다.



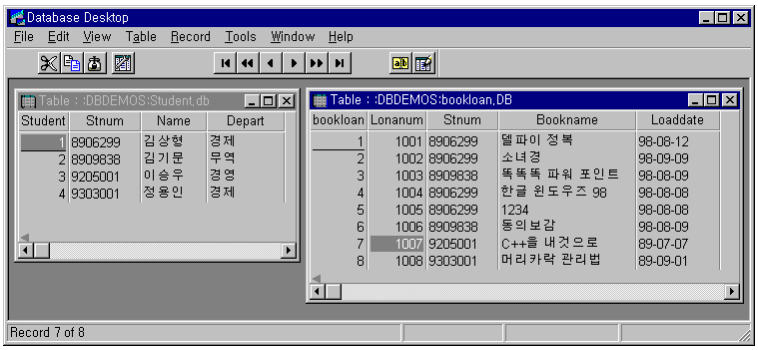

대출번호, 학번(=대출자), 서명, 대출일 네 개의 필드를 만들었다. Loannum 을 키로 지정하고 Stnum 을 세컨드리 인덱스로 만든다. Table properties 에서 Secondary Index 를 선택하고 Define 버튼 누르면 다음 대화상자가 나타날 것이다.




이 대화상자에서 Stnum 을 왼쪽으로 옮기고 OK 버튼을 누르면 인덱스의 이름을 물어올 것이다. 이 인덱스의 이름을 StNumIndex 로 설정하자.

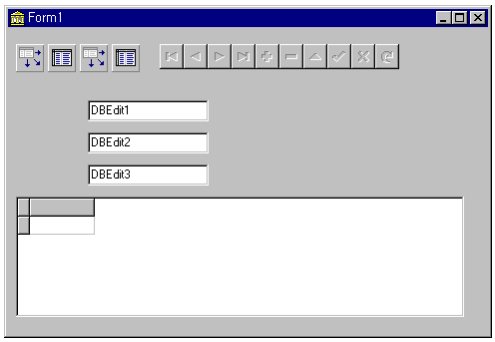



테이블을 다 만들었으면 이 테이블에 적당히 데이터를 입력해 넣는다. 많이 입력해도 나쁠 건 없겠지만 시간이 걸리므로 학생은 4 명만, 대출 정보는 8 개만 입력해 넣었다.



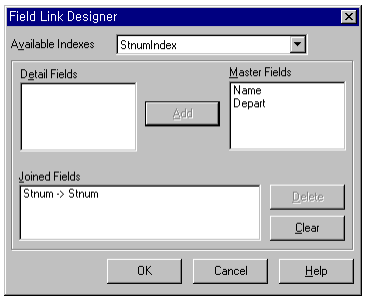

이렇게 준비된 테이블을 DBDEMOS 디렉토리에 저장해 넣는다.

2 새 프로젝트를 만들고 컴포넌트를 연결해 보자. 마스터, 디테일 각각에 대해 데이터 소스, 테이블 컴포넌트가 한 쌍씩 필요하다. 그리고 마스터 테이블인 학생 정보를 출력하기 위해 에디트 세 개, 네비게이터 하나를 배치하였으며 디테일 테이블인 대출 정보를 출력하기 위해 DBGrid 를 배치하였다.




이렇게 배치한 컴포넌트를 속성으로 각각 연결한다. DataSource1.DataSet 에 Table1 을 연결하고 에디트와 네비게이터는 DataSource1 에 연결하며 Table1 은 DBDEMOS 의 Student.db 에 연결해 주었다. DataSource2.DataSet 에 Table2 를 연결하고 DBGrid 는 DataSource2 에 연결한다. 여기까지는 아주 일반적인 설정이다.

이제 마스터 테이블과 디테일 테이블을 연결하기 위해 Table2.MasterSource 에 DataSource1 을 지정한다. 즉 디테일 테이블의 마스터 소스를 DataSource1 으로 설정하여 대출 상황 테이블이 학생 테이블로부터 정보를 받도록 한다. 다음으로 MasterFields 를 더블클릭하면 다음 대화상자가 나타날 것이다.

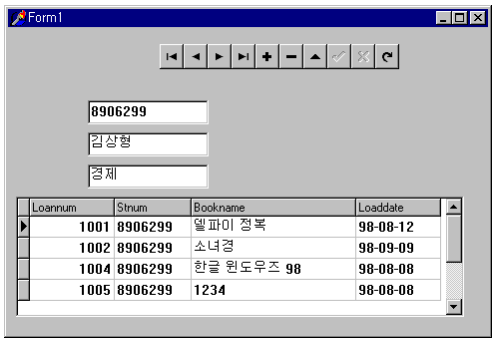



이 대화상자는 마스터 테이블과 디테일 테이블을 연결할 필드를 지정한다. 양쪽에 Stnum 을 선택하여 Add 버튼을 눌러주면 두 테이블은 이 필드를 매개로 하여 연결된다.

3 폼 시작시 두 테이블이 활성화되도록 해 준다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open;
  Table2.Open;
end;
```

이제 프로젝트를 실행해 보자.

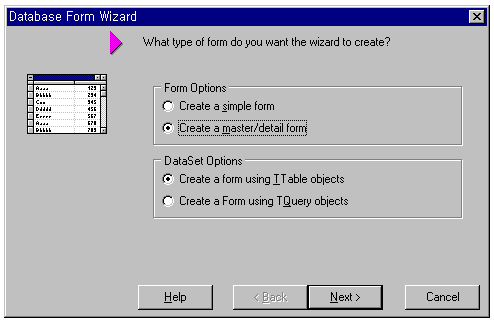



마스터 테이블에서 학생을 선택하면 아래쪽의 그리드에 이 학생이 빌려간 책의 목록이 나타날 것이다.

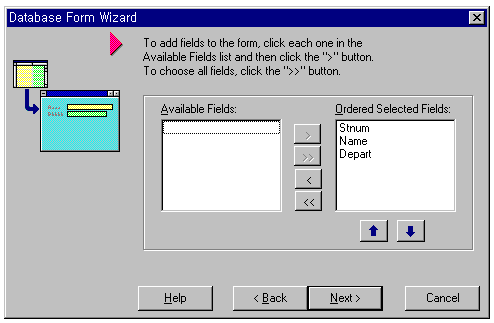
# 다. 폼 위저드로 만들기

15jang
BookLoan2

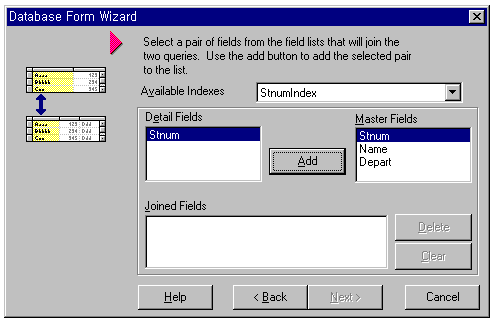
DB 폼 위저드를 사용하면 마스터/디테일 구조의 프로젝트를 좀 더 편리하게 만들 수 있다. 똑같은 예제를 이번에는 마법사를 사용하여 만들어 보자. 새 프로젝트 시작하고 File/New/Business/Database Form Wizard 선택한다.




첫 번째 질문의 Form Options 에서 Create a master/detail form 옵션을 선택하고 Next 를 누른다. 마스터 테이블과 디테일 테이블에 대해 순서대로 물어온다. 먼저 마스터 테이블을 지정해 줄 것을 요구하는데 DBDEMOS 의 Student 테이블을 선택한다. 포함할 필드는 전부 선택한다.

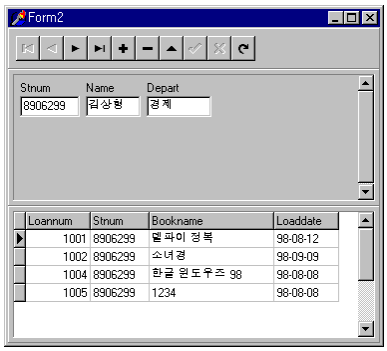



다음은 필드 정렬 방식을 물어오는데 디폴트로 선택되어 있는 수평(Horizontaly) 정렬을 선택하였다. 이어서 디테일 테이블을 물어온다. Bookloan.db 선택하고 필드를 전부 선택하였다. 필드 정렬 방식은 디폴트로 주어진 In a grid 를 선택한다. 다음은 마스터/디테일 구조를 만드는데 있어 가장 중요한 질문인 두 테이블을 연결할 필드에 대한 질문이다.




Available Indexes 에서 StnumIndex 를 선택한 후 양쪽에서 Stnum 을 선택하고 Add 버튼을 누른다. 마지막 질문은 지금 만든 폼을 메인폼으로 만들 것인지와 데이터 모듈을 생성할 것인지를 묻는데 모두 디폴트를 선택한다. Finish 버튼을 누르면 폼이 만들어질 것이다.

마지막으로 프로젝트 관리자를 불러내 Unit1 을 제거함으로써 정리를 하고 이 프로젝트를 BookLoan2 로 저장한다. 실행중의 모습은 다음과 같다.




마법사가 만들어 준 프로젝트를 분석해 보면 컨트롤의 배치 상태만 조금 다를 뿐이며 손으로 만든 프로젝트와 완전히 동일하다는 것을 알 수 있다.

# 15-7 클라이언트/서버

## 가. 클라이언트/서버

지금까지 작성한 모든 예제들은 로컬 컴퓨터 한 대에 데이터와 프로그램이 공존하는 형태였다. 클라이언트/서버(줄여서 C/S 라고 한다) 형태의 데이터 베이스 프로그램은 서버와 클라이언트로 분리한 것을 말한다. 데이터의 관리는 서버 컴퓨터내의 DBMS 가 전담하고 사용자에게 데이터를 보여주고 명령을 받아들이는 일은 클라이언트가 전담하도록 분업화한 것이다. 물론 두 컴퓨터는 네트웍으로 서로 연결되어 있어야 한다. 이런 형태의 장점은 다음과 같다.

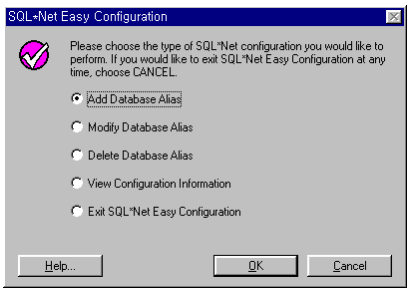
우선 두 대의 컴퓨터에서 서버와 클라이언트가 따로 동작하므로 시스템 효율을 높일 수 있다. 서버는 보통 우리가 사용하는 개인용 컴퓨터에 비해 월등한 성능을 가지므로 데이터 처리 능력이 탁월하다. 게다가 두 대의 컴퓨터를 사용하므로 CPU 가 두개인 셈이며 따라서 처리 속도가 빨라질 수밖에 없다. 또한 하나의 서버에 여러 개의 클라이언트가 동시에 동작할 수 있다. 즉 하나의 데이터 베이스를 여러 명이 동시에 사용할 수 있어 백화점이나 은행 등에서 각 판매대에 클라이언트를 배치해 두고 하나의 데이터 베이스를 참조할 수 있다.

C/S 형태의 데이터 베이스 프로그램을 만드는 것은 로컬 프로그래밍에 비해 다소 복잡한 면이 있으나 그렇다고 해서 질적으로 틀린 것은 아니며 로컬 프로그래밍에서 사용하던 기법은 거의 대부분 C/S 에도 적용된다.

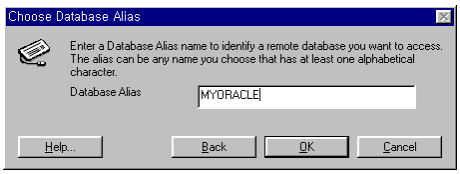
15jang
CSExam

간단한 클라이언트/서버 예제를 만들어 보도록 하되 오라클 7.3 버전이 설치된 NT 서버와 클라이언트를 연결하는 시범을 보일 것이다. 델파이 4 는 오라클 8.0 도 지원한다. 여기서는 오라클을 예로 들었지만 MS-SQL 서버나 기타 다른 DBMS 를 사용하더라도 동일한 방법으로 개발 가능하다. 이 예제를 제대로 실습해 보려면 두 대의 컴퓨터가 네트웍으로 연결되어 있어야 하며 서버 컴퓨터에는 오라클이, 클라이언트에는 SQL 관리 프로그램들이 설치되어 있어야 한다. 또한 사용자는 오라클과 그 관련 툴, 네트웍 등에 대해 이미 알고 있어야 한다. 다음 단계를 따라 실습을 해 보도록 하자.

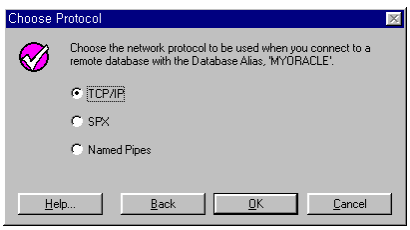
테이블을 만들기 위해 일단 데이터 베이스 서버와 연결해야 한다. 오라클 폴더에서 SQL net Easy Configuration 을 실행하면 다음 대화상자가 나타날 것이다.



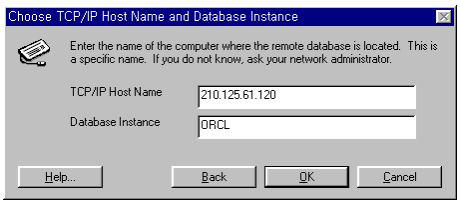

여러 가지 명령이 있는데 여기서 Add Database Alias 를 실행한다. 앨리어스를 묻는 대화상자가 나타는데 MYORACEL 이라고 앨리어스명을 입력해 보자. 물론 앨리어스명은 사용자가 임의대로 작성할 수 있다.



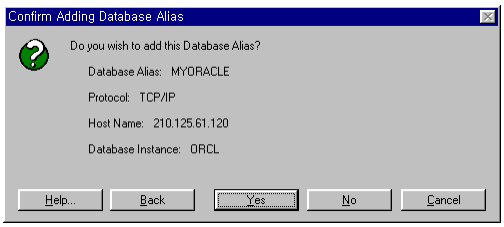

이 앨리어스명은 오라클 클라이언트에서 참조하는 서버명이 된다. 앨리어스명을 입력하면 서버와 어떤 방식으로 연결할 것인지 연결 방식을 묻는다.



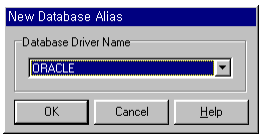

네트웍 상황에 맞게 적절한 프로토콜을 선택하되 대개의 경우 TCP/IP 를 선택하게 될 것이다. TCP/IP 를 선택하면 서버의 위치를 알기 위해 IP 어드레스를 물어온다. 현재 사용하고 있는 서버의 IP 주소를 조사하여 입력해 주도록 하자.




요약 대화상자를 보여주며 서버와의 연결이 완료되었다.




2 앞에서 만든 오라클 앨리어스를 델파이의 BDE 가 인식할 수 있도록 BDE 앨리어스를 만들어 주어야 한다. 델파이에서 BDE Administrator 를 실행하여 새 앨리어스를 만든다. 드라이버명으로 ORACLE 을 선택한다.



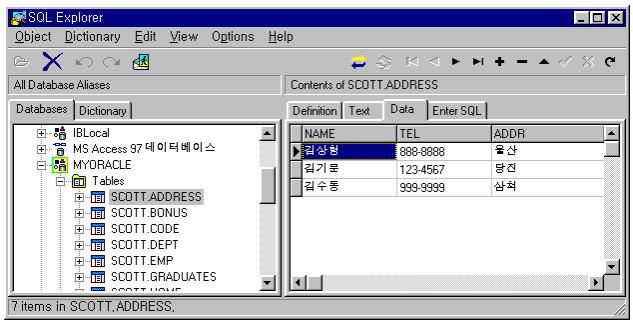

ORACLE1 이라는 이름의 새 앨리어스가 생성되는데 이름을 좀 더 기억하기 쉬운 MYORACLE 로 변경해 주도록 하자.

MYORACLE 앨리어스의 여러 가지 속성을 변경해 준다. Net Protocol 을 TCP/IP 로 변경하고 Server Name 은 앞에서 설정한 오라클 앨리어스 이름 앞에 @를 붙인 @MYORACLE 로 준다. BDE 앨리어스와 구분하기 위해 오라클 앨리어스 이름 앞에는 이런 식으로 @를 붙인다.

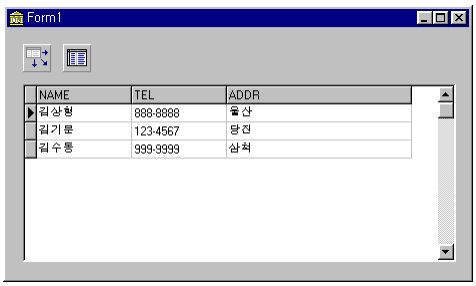
3 SQL Explorer 를 실행하여 서버에 테이블을 만든다. MYORACLE/Table/Enter SQL 에 다음 문장을 입력한다. 이때 암호를 물어볼 수도 있는데 대답해 주면 된다. 세 개의 필드를 가지는 간단한 테이블을 만드는 SQL 문이다.

```
create table address(
name varchar2(15),
tel varchar2(15),
addr varchar2(30))
```

팝업 메뉴에서 Execute 를 선택하면 이 SQL 문이 실행되어 서버에 테이블이 생성될 것이다. 팝업 메뉴에서 Refresh 를 선택하면 테이블이 보이는데 필자 컴퓨터의 경우 SCOTT.ADDRESS 라는 이름으로 테이블이 생성되었다. 이제 Data 탭에서 몇 가지 데이터를 넣어 보자. 물론 데이터는 프로그램을 작성한 후 직접 넣을 수도 있다.



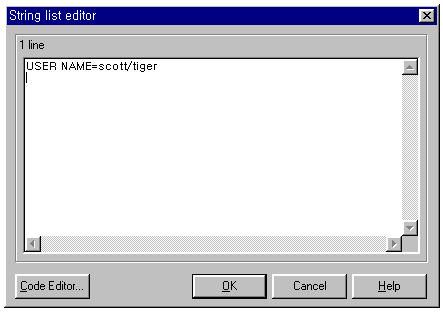

4 이제부터 로컬에서 프로그래밍하는 것은 지금까지 배운대로 하면 된다. 델파이에서 새 프로젝트를 시작하고 폼에 데이터 소스, 테이블, 그리드를 배치하고 속성으로 연결해 준다. 테이블의 DatabaseName 속성에 MYORACLE 로 주고 TableName 속성에 Address.dbf 를 선택해 준다. 이때 연결된 DBMS 에 따라 암호를 물어볼 수도 있는데 적당히 대답해 주도록 한다. 이제 데이터 소스를 테이블에 연결하고 그리드를 데이터 소스에 연결한 후 Table1.Active 를 True 로 변경해 주면 된다.




원격 데이터 베이스에 있는 테이블이 그리드에 나타났다. 실행하면 편집도 물론 가능하다.

5 원격 DBMS 는 보안을 위해 보통 권한이 있다. 권한을 가진 사용자만 사용할 수 있도록 되어 있으며 그래서 한번 로그인할 때마다 암호를

물어본다. 개발중에 암호에 일일이 대답해 주기가 무척 번거로우므로 보통 DataBase 컴포넌트를 배치하여 암호 입력을 생략하도록 한다. Data Access 페이지에 있는 DataBase 컴포넌트를 폼에 배치하고 AliasName 속성에 MYORACLE 을 선택해 주어 이 데이터 베이스가 어떤 DBMS 와 연결될 것인가를 지정한다. DatabaseName 속성은 이 데이터 베이스의 이름을 지정하는데 임의의 이름을 줄 수 있다. 이 속성에 MyAlias 를 입력해주도록 하자. 그리고 Connected 속성을 True 로 변경하면 암호를 한번 물어 오는데 이 암호에 대답해 주면 사용자 이름과 암호를 지정하는 파라미터가 생성된다. Params 속성을 더블클릭해 보면 사용자명과 암호가 설정되어 있을 것이다.




이제 LoginPrompt 속성을 False 로 변경하면 더 이상 암호는 물어보지 않을 것이다. 테이블의 DatabaseName 속성을 MyAlias 로 변경하면 작업이 끝난다. 물론 보안이 중요하다면 개발이 끝난 후에 LoginPrompt 를 True 로 변경하고 Params 를 삭제해 주면 된다.

# 나. MIDAS

델파이는 여러 계층으로 구성된 데이터 베이스 프로그램 개발을 지원하는데 이를 MIDAS(Multi-tiered Distributed Application Service)라고 한다. C/S 형태는 2 계층으로 이루어진 MIDAS 의 특수한 형태라고 할 수 있다. 3 계층, 4 계층은 물론 5 계층 이상도 만들 수 있지만 현실적으로는 다음과 같이 세 개의 부분으로 구성된 3 계층이 가장 일반적이다.

① 원격 데이터 베이스 서버 : 데이터를 관리하는 DBMS 이다. 오라클, 인포믹스, 마이크로소프트의 SQL 서버 등이 있으며 델파이와 함께 제공되는 인터베이스도 사용할 수 있다.

② 어플리케이션 서버 : 원격 데이터 베이스 서버로부터 데이터를 받아 클라이언트에게 제공하는 중계 역할을 담당하는 프로그램이다. 중계자 역할을 하므로 데이터 브로커(Data Broker)라고도 한다.

③ 클라이언트 : 사용자에게 데이터를 보여주고 명령을 받아 처리하는 프로그램이다. 데이터 베이스와의 접속은 어플리케이션 서버에서 담당하므로 클라이언트는 BDE 나 별도의 소프트웨어를 설치하지 않아도 동작할 수 있다.

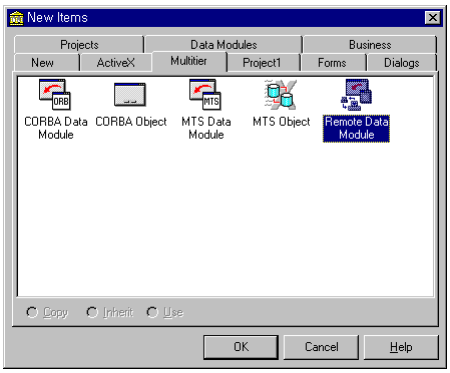
C/S 형태와 비교하자면 어플리케이션 서버가 하나 더 추가되었다. 어플리케이션 서버는 원격 데이터 베이스 서버가 해야 할 일을 대신해 주는 역할을 함으로써 서버의 과중한 부하를 덜어주는 역할을 한다. 데이터 베이스의 규모가 커지면 커질수록 서버가 해야 할 일이 많아지는데 가운데 하나의 계층을 더 둠으로써 전체적으로 데이터를 원활하게 처리할 수 있게 된다.

15jang
MyRemote

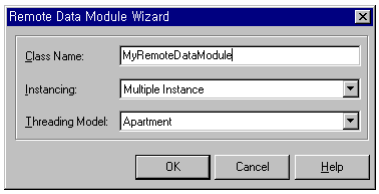
3 계층 데이터 베이스를 개발할 때 원격 데이터 베이스 서버는 기존 상품을 사용하며 개발 대상이 되는 것은 어플리케이션 서버와 클라이언트이다. 먼저 어플리케이션 서버부터 만들어 보자.

File/New Application 메뉴를 선택해 새 프로젝트를 만든다. 빈 폼만 하나 주어질 것이다. 이 폼은 기능적으로는 특별한 용도가 있는 것은 아니지만 그렇다고 해서 삭제해서는 안된다. 레이블이나 패널을 배치하여 이 프로그램이 어플리케이션 서버라는 것만 표시하는 용도로 사용하면 되는데 여기서는 디폴트로 주어진 대로 두기로 하자.

File/New 메뉴 항목을 선택해 Multitier 페이지에서 Remote Data Module 을 선택하여 데이터 모듈을 하나 만든다.



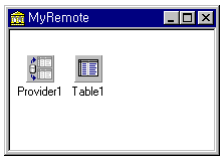
리모트 모듈의 이름을 물어오는데 MyRemoteDataModule 이라고 입력하면 새로 빈 모듈을 만들어 줄 것이다.




리모트 데이터 모듈은 COM 을 기반으로 하는 오토메이션 서버이며 여기에 데이터 액세스 컴포넌트를 배치할 수 있다.

3 빈 모듈에 테이블 하나와 프로바이더 컴포넌트를 각각 하나씩 배치한다.

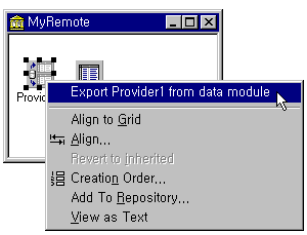
그림
리모트 데이터 모듈




테이블은 원격 데이터 베이스와 접속하며 프로바이더 컴포넌트는 이렇게 접속된 테이블을 클라이언트와 접속시켜 주는 중계자가 된다.

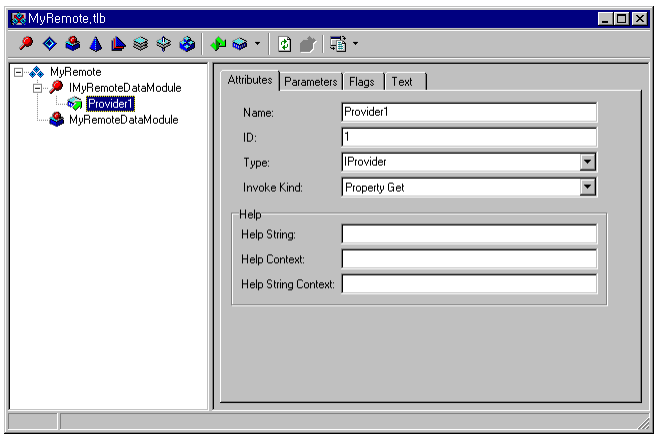
4 Table1.DatabaseName 속성에 DBDEMOS 를 주고 TableName 에 Address.dbf 를 선택해 준다. 여기서는 예제를 간단하게 만들어 보기 위해 DBDEMOS 데이터 베이스를 선택했지만 데이터가 원격 서버에 있으면 적당한 앨리어스를 대입해 주면 된다. Provider1.DataSet 에는 Table1 을 선택해 준다.

5 프로바이더의 팝업 메뉴에서 Export Provier1 from data module 항목을 선택한다.




이 항목을 선택하면 Provider1 이라는 인터페이스가 생성되며 이 인터페

이스는 클라이언트와 자료를 교환하기 위한 접속점이 된다.



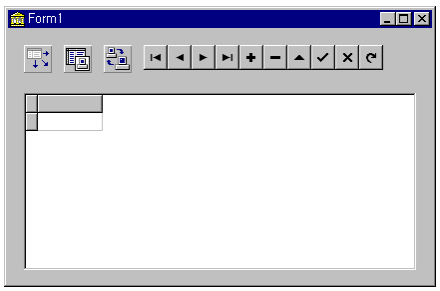

6 예제 작성이 끝났다. 이제 이 프로젝트를 저장하도록 하자. 데이터 모듈은 MyRemote_f2, 폼은 MyRemote_f 로 이름을 주고 프로젝트는 MyRemote 로 이름을 준다. 프로젝트를 저장했으면 일단 한번 실행시켜 레지스트리에 등록해 주어야 한다.

홀로 사용되는 프로젝트는 아니므로 실행해 봐야 빈 폼만 나타날 것이다. 계속해서 클라이언트 프로그램도 만들어 보자.

15jang
MyClient

1 File/New Application 으로 새 프로젝트를 시작한다.

2 폼에 데이터 소스, 클라이언트 데이터 셋, 리모트 서버, 그리드, 네비게이터를 배치한다.

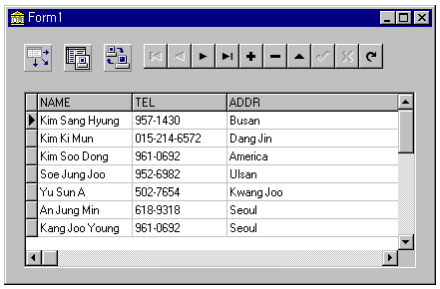



3 이 클라이언트를 어플리케이션 서버와 연결해 준다. 리모트 서버 컴포넌트를 선택하고 오브젝트 인스펙터에서 ServerName 속성의 콤보 박스를 열어 보면 시스템에 등록된 원격 데이터 모듈 이름이 나타나는데 이 중 MyRemote.MyRemoteDataModule 을 선택해 준다. ServerName 속성이

입력되면 ServerGUID 가 자동으로 입력된다. 만약 어플리케이션 서버가 로컬에 있지 않을 경우 ComputerName 속성에 어플리케이션 서버가 있는 위치를 먼저 선택해 주어야 한다.

4 ClientDataSet 의 RemoteServer 속성 목록에서 RemoteServer1 을 선택하고 ProviderName 에 Provider1 을 목록에서 선택한다. 이 시점에서 어플리케이션 서버가 자동으로 실행될 것이다.

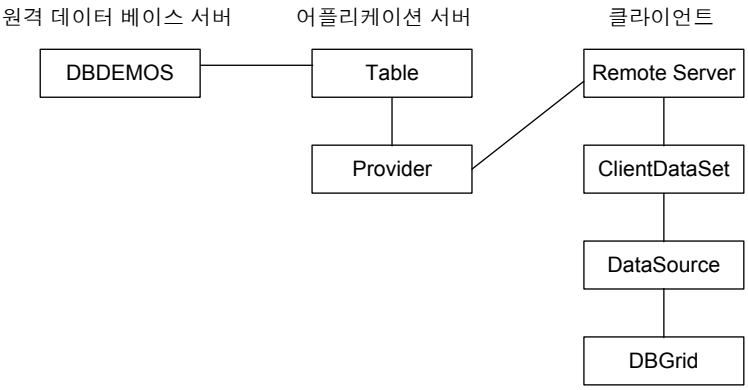
5 데이터 소스의 DataSet 속성에 ClientDataSet1 을 선택하고 그리드와 네비게이터를 데이터 소스에 연결한다. 마지막으로 클라이언트 데이터셋의 Active 속성을 True 로 변경하면 그리드에 테이블 내용이 나타날 것이다.



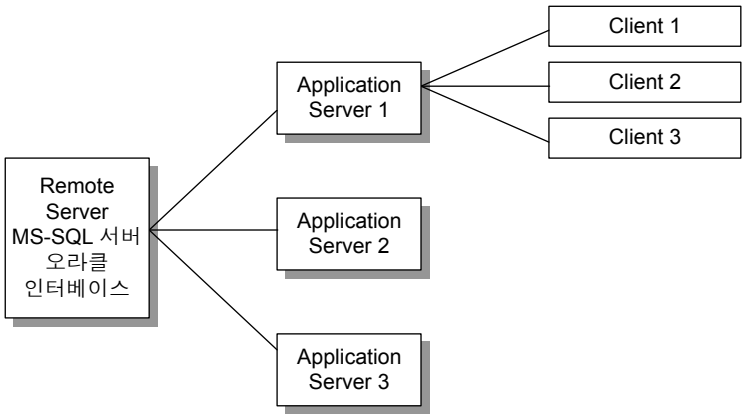

6 MyClient.dpr 이라는 이름으로 프로젝트를 저장한다.

제대로 되었는지 테스트해 보자. 델파이 개발 환경에서 이 프로젝트를 닫고 탐색기에서 직접 MyClient.exe 를 실행해 보자. 클라이언트만 실행하면 어플리케이션 서버도 같이 실행되며 마찬가지로 클라이언트가 종료되면 서버도 같이 종료된다. 데이터의 전달 순서가 무척 복잡해 보이는데 그림으로 정리해 보면 다음과 같다. 이 예제의 경우는 원격 데이터 베이스 서버를 사용하지 않았지만 데이터를 서버로 옮겨주고 앞에서 C/S 개발시에 사용한 방법대로 해 주면 된다.

그림
3 계층 데이터 베이스




이상으로 3 계층의 데이터 베이스 프로그램을 만들어 보았다. 하나의 원격 데이터 서버에는 사용 목적에 따라 여러 개의 어플리케이션 서버를 둘 수 있으며 또한 각 어플리케이션 서버 당 여러 개의 클라이언트를 둘 수 있다.

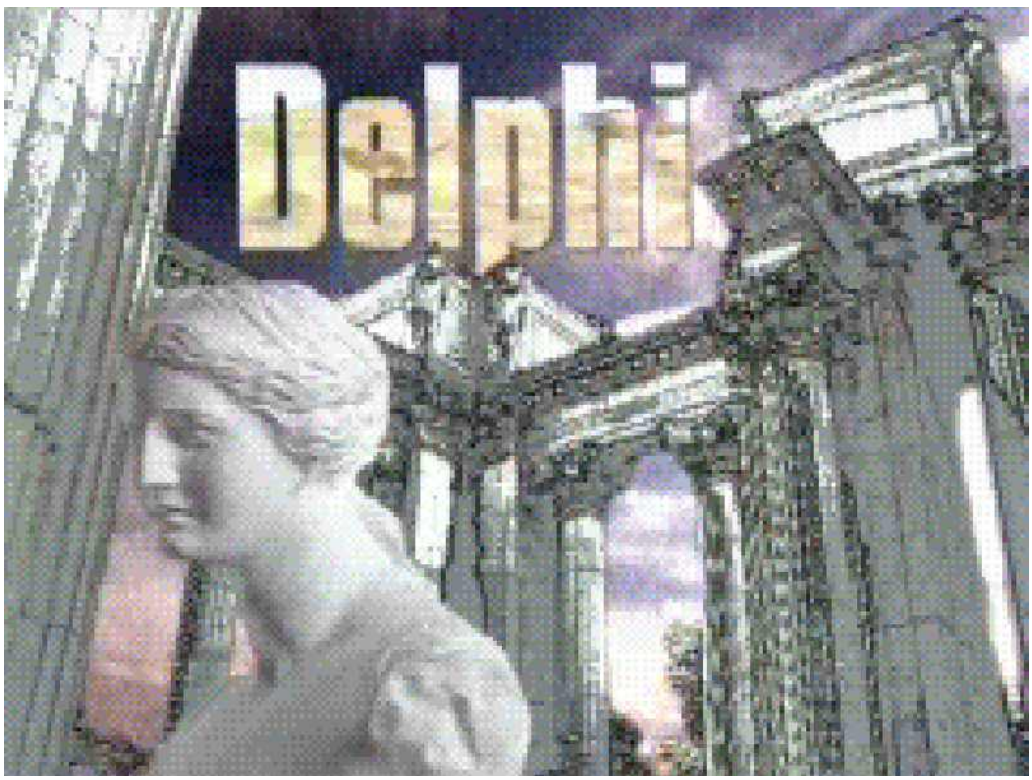



이상 여기까지 델파이의 데이터 베이스 프로그래밍에 관한 여러 가지 사항에 관해 알아 보았다. 독자들도 느끼겠지만 기본적인 개념에 대해서만 설명하고 있을 뿐 실무에 도움을 줄 만한 내용은 많지 않은 것이 사실이다. 그럴 수밖에 없는 것이 데이터 베이스 프로그래밍을 제대로 하려면 델파이만 알아서는 여러 가지 면에서 한계에 부딪히게 되며 알아야 할 내용도 무척이나 많다. 이 책은 델파이적인 데이터 베이스 프로그래밍에 관해 논하고 있을 뿐이지 일반적인 데이터 베이스 이론을 논하지는 못하고 있다. 더 깊은 부분

까지 상세하고 알고 싶은 사람은 좀 더 전문적인 서적을 참고하기 바란다.

# 컴포넌트 제작

제 16 장



# 16-1 컴포넌트 제작의 기초

## 가. 사용과 제작의 차이점

델파이 프로그래밍의 핵심은 컴포넌트다. 컴포넌트를 조립하여 프로그램을 완성시켜 나가므로 개별적인 컴포넌트에 관해 많이 알아야 델파이를 자유롭게 쓸 수 있다. 그래서 컴포넌트를 프로그램의 부품이라고 한다. 델파이는 컴포넌트를 사용할 수 있도록 해줄 뿐만 아니라 컴포넌트를 직접 만들 수 있는 방법도 제공하므로 컴포넌트를 사용하는 툴이면서 동시에 컴포넌트를 만들 수 있는 툴이다. 그래서 델파이를 쉽게 배워 요긴하게 쓰고자 하는 사람은 기존 컴포넌트를 이용하기만 하면 되며 좀 더 고급적이고 전문적인 프로그래밍을 하고자 하는 사람은 직접 컴포넌트를 만들어 쓸 수도 있다.

델파이는 저수준과 고수준을 고루 고루 지원하는 개발 툴이다. 컴포넌트를 만드는 일은 사용하는 것에 비해 무척이나 어려울 것이라는 것은 쉽게 짐작이 갈 것이다. 컴포넌트를 사용하는 것에 비한 컴포넌트 제작의 어려움은 다음과 같다.

**❶ 제작 과정이 비가시적이다**

컴포넌트를 사용할 때는 마우스로 원하는 위치에 배치하고 마음에 들지 않으면 자리를 옮기거나 크기를 마음대로 바꿀 수 있지만 컴포넌트 제작 과정은 처음부터 끝까지 코드 작성 과정의 연속이므로 다 만든 후 설치하여 사용해 보기 전에는 결과를 볼 수 없다. 따라서 제작 시간이 많이 소요될 뿐만 아니라 비능률적이다. 하지만 비주얼 툴(구체적으로 오브젝트 인스펙터를 말한다)을 사용하지 못할 뿐이지 코드 에디터, 디버거, 오브젝트 브라우저 등의 델파이 개발 환경은 모두 사용 가능하다.

**❷ 더 많은 지식을 요구한다**

새로운 컴포넌트는 기존의 컴포넌트를 파생시켜 만들어진다. 파생(derivation)이란 객체 지향 프로그래밍(OOP) 기법의 일종이므로 컴포넌트를 만드는 일은 OOP에 대한 해박한 지식을 요구한다. OOP를 모르면 컴포넌트를 만들어 보겠다는 생각은 꿈이 되고 만다. 뿐만 아니라 컴포넌트를 단순히 사용할 때는 신경쓰지 않아도 되는 컴포넌트의 숨겨진 기능이나 속성(protected

interface)에 대해서도 일일이 신경써야 하며 거의 모든 컴포넌트의 특성과 계통에 대해 숙지하고 있어야 한다.

**❸ 까다로운 규칙이 존재한다**

컴포넌트를 사용하는 일은 무척이나 쉽다. 마우스로 끌어다 놓고 속성만 바꾸어 주면 원하는 대로 컴포넌트가 만들어진다. 뿐만 아니라 사용자가 터무니 없는 실수를 해도 델파이가 알아서 실수를 방지해 준다. 예를 들어 버튼의 Width 속성에 실수로 -20 등의 음수값이 입력되면 델파이가 이 값을 인정해 주지 않으므로 골치아픈 버그를 막아준다. 하지만 컴포넌트 제작시에는 제작자가 코드를 직접 작성하므로 이런 실수도 제작자가 책임을 져야 한다. 뿐만 아니라 제작자는 사용자의 실수까지도 고려하여 어떤 환경에서도 사용할 수 있는 호환성이 좋은 컴포넌트를 만들어야 한다. 컴포넌트를 사용하는 것보다 만드는 것이 더 규칙이 까다로울 것임은 쉽게 수긍이 갈 것이다.

컴포넌트 제작 과정은 이렇게 어렵고 골치가 아프다. 컴포넌트를 만드는 일은 델파이를 사용하는 일과는 다소 질적으로 다른 분야이므로 몰라도 상관없다는 사람은 건너 뛰어도 좋다. 그러나 설사 이해를 못하더라도 한번쯤 봐 두는 것이 좋다고 생각한다. 왜냐하면 컴포넌트를 만드는 과정에서 컴포넌트의 일반적인 특성에 대해 이해할 수 있으며 일단 이해를 하면 백방으로 활용할 수 있는 응용 능력이 생기기 때문이다. 마치 자동차를 정비할 줄 아는 사람이 자동차를 더 잘 관리하고 운전도 잘 하는 이치와 같다. 여기서는 아주 간단한 부분이나마 컴포넌트 제작 과정을 직접 실습해 보고 중요한 규칙들에 관해 알아 본다.

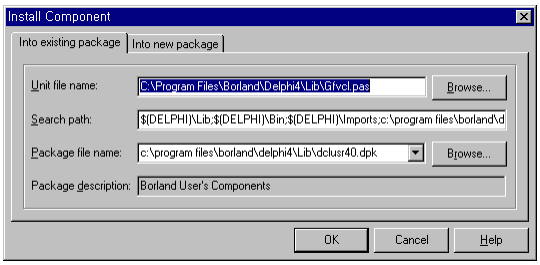
# 나. 컴포넌트의 설치

델파이는 개방적인 구조를 가지고 있어 델파이와 함께 제공되는 기존의 컴포넌트 외에도 별도로 만들어진 컴포넌트를 추가로 설치하여 사용할 수 있다. 추가로 설치되는 컴포넌트는 자신이 직접 만들 수도 있지만 그보다는 통신망을 통해 구하거나 또는 상업적으로 판매되는 컴포넌트를 구입하는 경우가 더 많을 것이다. 통신망에 이미 델파이용으로 공개된 컴포넌트만도 수천 개를 헤아리며 그 중에는 정말 쓸만한 것들이 많이 있다. 새로운 컴포넌트를 구했으면 델파이의 개발 환경에 새 컴포넌트를 설치해 주어야 사용할 수 있다. 설치의 대상이 되는 컴포넌트는 컴파일된 DCU 파일로 제공된다. 또는 소스 파일인 PAS 파일을 곧바로 사용할 수도 있지만 이 경우는 델파이가 PAS를 컴파일하여 DCU로 만든

후 설치하므로 결국 설치의 대상이 되는 파일은 DCU 파일이다.

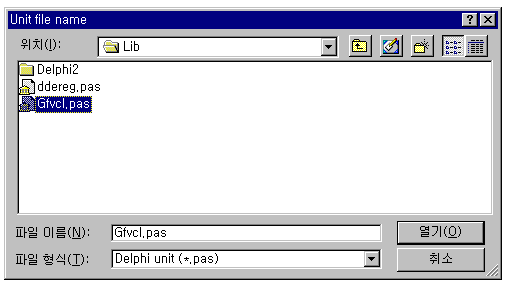
DCU 파일 또는 PAS 파일을 하드 디스크의 임의 위치에 복사하되 가급적이면 delphi 4₩lib 디렉토리에 복사해 두고 설치하는 것이 좋다. 여기서는 그라디언트 효과를 내는 GFVCL 컴포넌트를 예로 들어 설치 과정을 보인다. 배포 CD의 etc₩gradient 디렉토리에 이 컴포넌트의 소스와 DCU 파일이 있는데 Gfvcl.pas와 Gfvcl.dcr 파일을 delphi 4/lib 디렉토리에 복사해 두도록 한다. 이 컴포넌트는 16비트 컴포넌트이기 때문에 DCU 파일을 직접 설치할 수는 없으며 소스 파일을 다시 컴파일해야 한다. Component/Install Component 메뉴 항목을 선택하여 컴포넌트 설치 대화상자를 불러낸다.

그림
컴포넌트 설치 대화상자

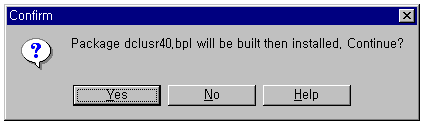



제일 위의 Unit file name에 설치하고자 하는 유닛의 이름을 적어주되 보통 Browse 버튼을 눌러 파일을 선택한다. search path 에디트 박스는 라이브러리 구성 파일의 검색 경로를 지정하며 도스에서 PATH를 지정하는 방법과 마찬가지로 세미콜론으로 끊어 여러 개의 경로를 동시에 지정할 수 있다. 이 에디트 박스의 내용은 직접 입력하지 않아도 Browse 버튼을 사용하여 PAS나 DCU 파일을 선택하면 그 디렉토리가 자동으로 기입된다.

Package file name에는 컴포넌트가 설치될 패키지명을 적어준다. 디폴트로 dclusr40.dpk 패키지가 지정되어 있는데 다른 패키지로 바꿀 수도 있다. Package description에는 선택된 패키지에 대한 간단한 설명이 나타난다. Browser 버튼을 누르면 DCU 파일을 선택할 수 있는 대화상자가 나타난다.



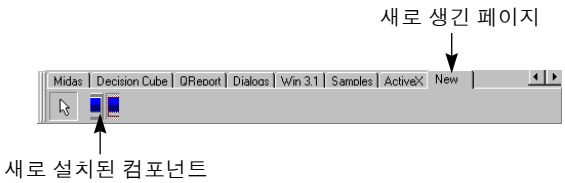
Gfvcl.pas 파일을 선택한 후 OK 버튼을 눌러 설치 대화상자로 돌아간다. 컴포넌트 설치 대화상자에서 다시 OK 버튼을 누른다. 그러면 추가된 컴포넌트를 포함시키기 위해 dclusr40.bpl을 새로 컴파일하겠다고 물어오는데 당연히 Yes라고 대답해야 한다.




델파이는 추가된 컴포넌트를 포함하여 패키지를 다시 컴파일하며 이 과정은 약간 시간이 걸릴 것이다. 잠시 기다리면 컴파일이 무사히 되었다는 메시지가 나타난다.


메시지의 내용인즉 무사히 컴파일되었으며 TGradientFill이라는 컴포넌트가 새로 등록되었다는 뜻이다. OK 버튼을 누르고 컴포넌트 팔레트를 살펴보면 추가된 컴포넌트가 New 페이지에 나타나 있을 것이다.

그림
컴포넌트를 설치한 후의 팔레트




마지막으로 File/Save All을 선택하여 패키지 소스를 저장하면 컴포넌트 설치가 무사히 완료되었다.

컴포넌트를 설치할 페이지에 관한 정보는 DCU 파일이나 PAS 파일에 컴포넌트를 만든 사람에 의해 이미 결정되어 있으므로 사용자가 마음대로 바꿀 수 없다. 하지만 일단 설치하고 난 후에는 Component/Configure Palette...메뉴 항목을 선택하여 마음대로 변경할 수 있다.

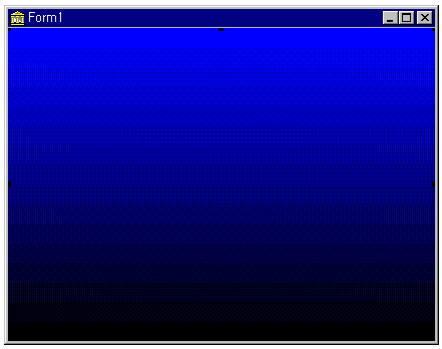
위의 과정대로 하면 정상적으로 설치되어야 하지만 Unsupported 16bit Resource 에러가 발생할 수도 있다. 이 에러의 의미는 16비트 버전용으로 제작된 리소스를 32비트 버전에서 사용할 수 없다는 뜻이다. PAS 소스는 32비트 버전에서 그대로 사용할 수 있지만 컴포넌트의 비트맵을 담고 있는 DCR 파일은

그대로 사용할 수 없다. 이 문제를 해결하려면 16비트의 DCR 파일을 32비트로 바꾸어 주면 되며 이미지 에디터에서 DCR 파일을 읽어 다시 저장하기만 하면 된다.  현재 인터넷에서 구할 수 있는 16비트 델파이용 컴포넌트는 이 과정을 거쳐야만 설치할 수 있다.

추가한 컴포넌트를 사용하는 방법은 기존의 컴포넌트를 사용하는 방법과 동일하다. GFVCL 컴포넌트는 설치 화면에서 흔히 볼 수 있는 색상이 점점 흐려지는 효과를 내는 컴포넌트이다.

**그림**

Gradient 컴포넌트의 사용예



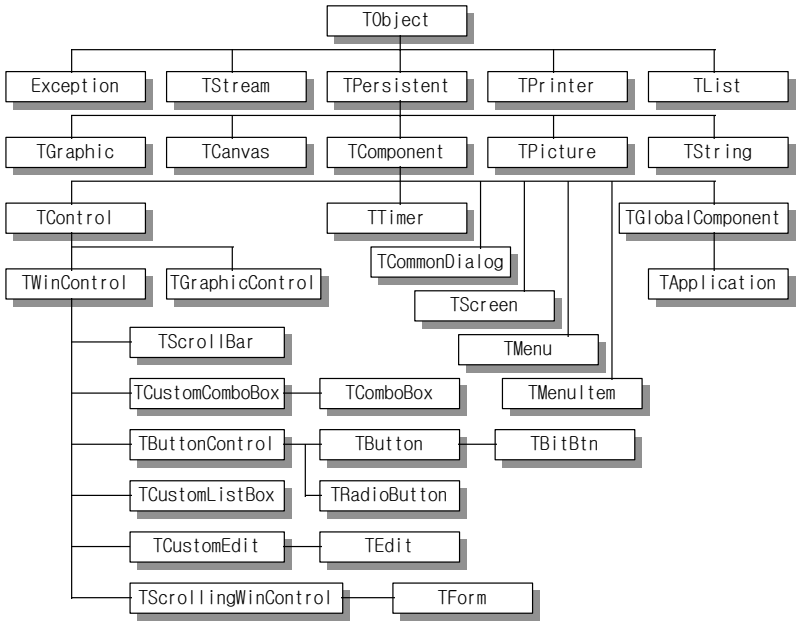

일단 컴포넌트가 팔레트에 설치되고 프로젝트에 사용되면 컴포넌트는 프로젝트의 실행 파일에 포함되므로 DCU 파일을 같이 배포할 필요는 없다.

인터넷이나 우리나라 대중 통신망에도 공개된 컴포넌트들이 많이 있으므로 한번쯤 설치해서 사용해 보기 바란다. 컴포넌트 방식의 프로그래밍 개발 툴이 왜 좋은가를 실감할 수 있을 것이다.

# 다. VCL

델파이는 컴포넌트를 개별적으로 취급하지 않고 하나의 통합된 라이브러리로 묶어 사용한다. 이 라이브러리를 VCL(Visual Component Library)이라고 하며 이 파일 안에 우리가 사용하는 버튼, 레이블, 리스트 박스 등의 컴포넌트가 저장되어 있다. VCL 내의 오브젝트들은 상속을 통해 복잡한 계층 관계를 이루고 있다. 새로운 컴포넌트를 만들려면 VCL 내의 기존 오브젝트를 상속받아야 하므로 VCL 내의 오브젝트 계층을 이해하고 있어야 한다. VCL 계층 구조는 굉장히 복잡하지만 간단히(?) 그려보면 다음과 같다. 물론 이 그림을 이해하려면 OOP에 관한 학습이 선행되어 있어야 한다.

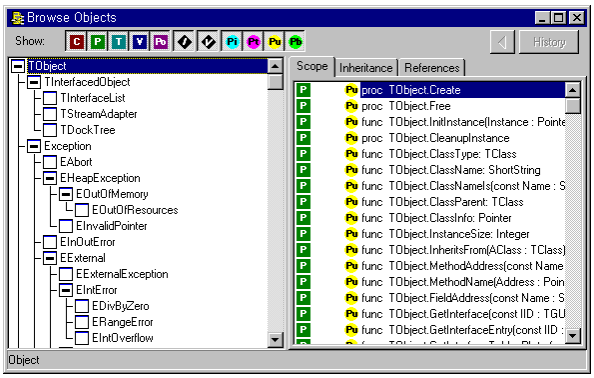
그림
델파이의 VCL 계층도




이 계층도에서 TComponent는 모든 컴포넌트의 부모이며 모든 컴포넌트에 공통적으로 사용되는 속성과 이벤트가 정의되어 있다. TComponent로부터 파생되어 여러 가지 컴포넌트들이 만들어진다. 컴포넌트를 만들 때 제일 먼저 고려해야 할 사항은 어느 컴포넌트로부터 자신의 컴포넌트를 파생시킬 것인가 하는 점이다. 정확한 결정을 내리기 위해서는 델파이가 기본적으로 제공하는 모든 컴포넌트의 특성을 거의 완벽하게 파악하고 있어야 한다.

오브젝트 계층을 좀 더 쉽게, 잘 이해하려면 델파이가 제공하는 오브젝트 브라우저의 도움을 받는 것이 좋다. 오브젝트 브라우저는 VCL 내의 오브젝트 계층은 물론 각 오브젝트에 정의된 필드, 메소드 등을 한눈에 볼 수 있도록 해 준다. 오브젝트 브라우저는 View/Browser 메뉴를 선택하면 볼 수 있다. 단 프로젝트가 실행중일 때만 볼 수 있다.

그림
오브젝트 브라우저




오브젝트 브라우저를 통하여 컴포넌트 간의 계통을 확인할 수 있으며 컴포넌트에 속한 필드, 메소드의 속성, 이름을 살펴볼 수 있다. 오브젝트 브라우저의 정확한 사용 방법은 물론 도움말을 참조해야겠지만 그 보다는 OOP를 이해하면 아주 자연스럽게 사용 방법을 알게 된다. 도움말을 아무리 읽어도 OOP를 모르면 그 좋은 고성능의 오브젝트 브라우저도 그림의 떡이나 다름없다.

참고하세요

컴포넌트를 만드는 과정은 규칙도 많고 알아야 할 것도 많지만 시중에 판매되는 책이나 대중 통신망이나 인터넷을 통해도 관련 자료를 얻기가 쉽지 않을 것이다. 컴포넌트 제작에 관련된 책이라고는 델파이와 함께 제공되는 Component Writer's Guide가 거의 전부이며 이 책도 그다지 상세하게 다루고 있지는 않다. 결국 이 분야에 관심이 있다면 최후의 자료 출처는 도움말이 될 수밖에 없는데 컴포넌트 제작 과정은 델파이의 메인 도움말의 Create Custom Components에 있다. 이 도움말이 컴포넌트 제작의 바이블인 격이므로 참고하기 바란다.

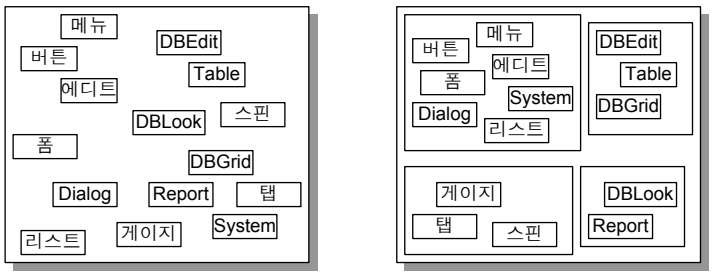
# 16-2 패키지

## 가. 패키지의 정의

델파이 2.0 버전까지는 모든 컴포넌트들이 VCL 이라는 시스템 라이브러리에 포함되었었다. 그러다 보니 VCL 파일인 COMPLIB32.DCL 파일의 크기가 무시 못할만큼 커져 버렸고 관리하기가 쉽지 않게 되었다. 새로운 컴포넌트를 설치하거나 제거할 때마다 VCL 을 전부 다시 컴파일해야만 했으며 만에 하나라도 VCL 이 잘못되면 델파이를 다시 설치해야하는 위험까지 가지고 있었다. 델파이 3.0 버전부터는 이렇게 거대한 크기의 VCL 을 잘게 쪼개서 패키지라는 것을 구성하고 패키지를 모아 VCL 을 만드는 방법을 사용한다.

**그림**

VCL 은 패키지로 구성된다.




2.0버전 이전
모든 컴포넌트가 VCL에 있다.

3.0버전 이후
컴포넌트끼리 모여 패키지를 이루고
패키지가 모여 VCL을 구성한다.

패키지는 컴포넌트들을 담는 특별한 종류의 DLL 이라고 할 수 있으며 패키지 안에는 컴포넌트, 타입, 오브젝트 등등 공유가능한 코드와 데이터가 들어 있다. 델파이로 만든 프로그램은 물론 델파이 개발 환경 자체도 패키지를 사용한다. 비록 확장자로 BPL 을 사용하지만 파일 포맷은 DLL 과 완전히 동일하다. 패키지를 사용하면 다음과 같은 여러 가지 이점들이 생긴다.

① 우선 컴포넌트들이 패키지에 포함되므로 통합환경에 컴포넌트를 개별적으로 설치할 필요없이 패키지를 선택/해제함으로써 쉽게 컴포넌트를 추가/삭제할 수 있으며 추가/삭제하는 시간도 빨라졌다.

②그리고 런타임 패키지를 사용해서 프로젝트를 컴파일하면 패키지를 별도의

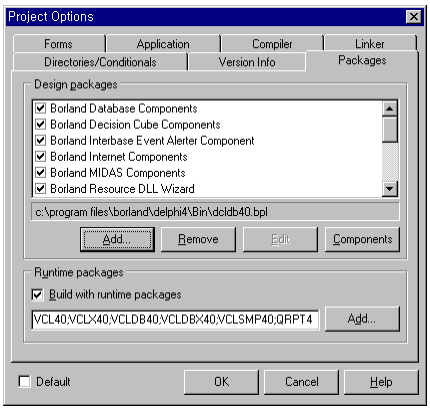
DLL 로 분리해 주므로 실행 파일의 크기가 현격하게 줄어드는 이점이 있으며 복수 개의 프로그램이 패키지 코드를 공유할 수 있어 메모리 효율에도 유리하다.

③ 실행 파일에는 꼭 필요한 고유 코드만 포함되기 때문에 컴파일 시간도 더욱 빨라진다.

④ 프로그램의 고유 코드와 기본 코드가 분리되었으므로 프로그램의 개정판을 배포하기가 훨씬 더 쉽다. 패키지는 그대로 사용할 수 있으므로 실행 파일만 다시 배포하면 된다.

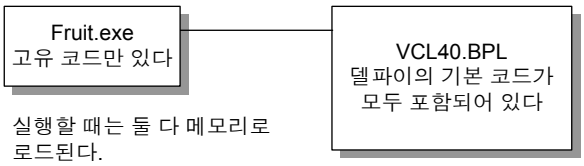
물론 이런 장점만 있는 것은 아니다. 단독 실행 파일이 아니므로 배포하기도 번거롭고 차칫 잘못 사용하면 패키지의 버전이 맞지 않아 프로그램이 실행되지 못할 수도 있다. 게다가 프로그램이 여러 개가 아닐 경우는 패키지의 이점이 나타나지 않는다.

Fruit 예제를 사용하여 패키지를 테스트해 보자. 이 프로젝트를 그냥 컴파일하면 284KB 크기의 실행 파일이 만들어지는데 프로그램의 기능에 비해서는 엄청나게 큰 크기이다. 이 프로젝트를 런타임 패키지를 사용하도록 다시 컴파일해 보자. Project/Options/Packages 에서 Build with runtime packages 옵션을 선택한 후 다시 컴파일하면 된다.




그리고 탐색기로 Fruit.exe 파일의 크기를 살펴보면 놀랍게도 실행 파일의 크기가 14KB 로 줄어들어 있을 것이다. 실행해 보면 런타임 패키지를 쓰지 않았을 때와 기능적으로는 완전히 동일하다. 그렇다면 나머지 270K 바이트는 어디로 간 것일까? 이 코드들이 모두 런타임 패키지로 분리된 것이다. 런타임 패키지에 대부분의 컴포넌트가 들어있기 때문에 실행 파일에는 응용 프로그램 고유의 코

드만 넣으면 된다.




Fruit.exe 는 실행과 동시에 런타임 패키지인 VCL40.BPL 을 자신의 메모리 영역으로 불러들이고 이 패키지에 있는 버튼, 레이블, 폼 등의 코드를 호출하는 것이다. 물론 이렇게 만들어진 실행 파일이 실행되려면 VCL40.BPL 파일이 윈도우즈 시스템 디렉토리 있어야 하며 이 프로그램을 배포할 때도 VCL40.BPL 파일을 같이 배포해 주어야 한다. 비주얼 베이직 프로그램이 VBRunxx.dll 이 필요하고 MFC 로 만든 프로그램이 MFCxx.dll 이 필요한 것과 마찬가지이다.

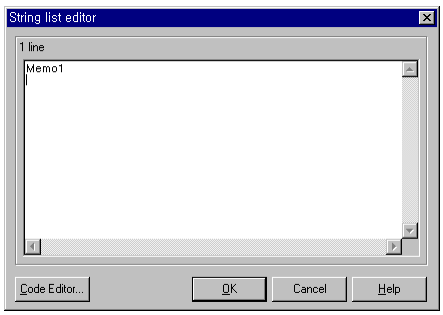
델파이가 시스템에 설치되면 VCL40.BPL 이 윈도우즈의 시스템 디렉토리에 같이 설치되므로 14K 크기의 Fruit.exe 가 아무 이상없이 실행되지만 이 실행 파일만 다른 시스템으로 가져가 실행하면 절대로 실행되지 않는다. 결국 런타임 패키지를 사용한 경우는 실행 파일 14K+패키지 1839KB = 1853KB 용량을 배포해야 하는 것이다. 하나의 프로그램인 경우는 이렇게 불리하지만 수십 개의 프로그램을 동시에 배포해야 할 경우는 오히려 더 유리하다.

# 나. 패키지의 종류

패키지에는 디자인 패키지와 런타임 패키지 두 종류가 있다. 둘 다 컴포넌트를 담는다는 면에서는 동일하지만 디자인 패키지에는 컴포넌트 자체뿐만 아니라 컴포넌트를 편집하는데 필요한 속성 에디터와 패키지 등록 코드 등이 포함되어 있다는 점이 다르다. 런타임 패키지에는 순수하게 컴포넌트의 동작 코드만 들어 있다. 속성 편집기나 등록 코드 따위는 프로그램의 실행과는 무관하기 때문에 굳이 런타임 패키지에 포함시키지 않아도 된다.

예를 들어 메모 컴포넌트를 담고 있는 패키지를 생각해 보자. 메모 컴포넌트는 여러 줄의 문자열을 편집할 수 있는 컨트롤이므로 이 패키지에는 텍스트 편집 코드가 포함되어 있어야 할 것이며 그외 메모의 속성을 관리하는 코드가 필요할 것이다. 이런 코드만 들어 있다면 이는 런타임 패키지가 되며 메모를 사용하는 프로그램과 함께 배포하면 된다. 그런데 메모 컴포넌트를 델파이 개발환경에서 사용하려면 텍스트 편집 코드만 있어서는 안된다. Items 속성을 편집하는 문자

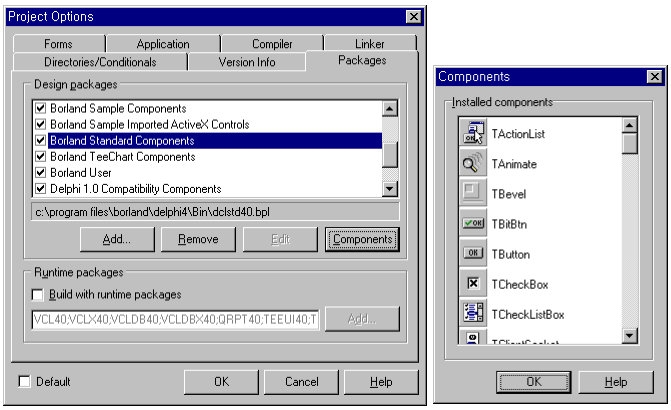
열 리스트 편집기, 폰트 선택 대화상자 등이 같이 필요하다.



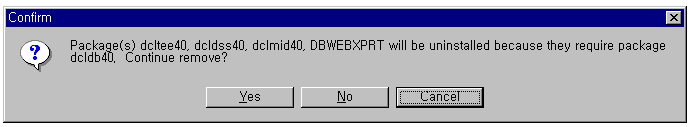

이런 속성을 편집하는 대화상자까지 다 포함하면 디자인 패키지가 된다. 런타임 패키지에는 이런 비주얼 속성 편집 지원까지 같이 포함될 필요가 없다. 그래서 꼭 필요한 코드만을 포함시키기 위해 패키지의 종류가 두 가지로 나누어져 있는 것이다. 디자인 타임 패키지에는 컴포넌트를 관리하는 모든 코드가 들어 있기 때문에 델파이의 통합 개발 환경에 설치될 수 있지만 런타임 패키지는 실행 코드만 가지고 있기 때문에 델파이 환경에 설치될 수 없다. 디자인 타임 패키지는 실행중에 런타임 패키지를 호출하도록 되어 있다.

그럼 이제 패키지의 안쪽을 들여다 보도록 하자. 컴포넌트 팔레트에 설치된 모든 컴포넌트들은 디자인 패키지에 포함되어 있는데 어떤 패키지에 어떤 컴포넌트들이 있는지 보자. Project/Options/Packages 탭의 체크 리스트 박스에는 현재 델파이에 설치된 디자인 패키지가 나열되어 있다. 목록에서 패키지를 선택하면 중앙에 패키지 파일의 이름이 나타나는데 디자인 패키지는 모두 DCL 로 시작되며 BIN 디렉토리에 있다. 각 디자인 패키지에 어떤 컴포넌트들이 포함되어 있는지 알고 싶으면 목록에서 패키지를 선택한 후 Components 버튼을 눌러 보면 된다. 예를 들어 Borland Standard Components 에 어떤 컴포넌트가 있는지 보자.

그림
패키지에 포함된 컴포넌트들




액션 리스트, 버튼, 체크 박스, 메뉴, 이미지 등 가장 일상적으로 사용하는 컴포넌트들이 들어 있다. 이 패키지를 선택 해제하면 이 안에 들어 있는 모든 컴포넌트들을 사용할 수 없게 된다. 즉 컴포넌트 팔레트에서 사라지게 된다. 시험삼아 제일 아래쪽에 있는 QuickReport Components 패키지를 선택 해제하거나 Remove 버튼을 누른 후 OK 버튼을 눌러보자. 그러면 컴포넌트 팔레트에서 퀵리포트 컴포넌트들이 죄다 사라질 것이다. 사용하기 싫은 컴포넌트가 있다면 디자인 패키지 목록에서 제거해 주면 된다. 이때 델파이는 제거할 패키지가 다른 패키지에서 사용되고 있다면 다음과 같이 경고를 한다. 예를 들어 제일 위쪽의 Borland Database Components 를 제거해 보자.




이 패키지가 제거되면 dcltee40, dcldss40 등의 패키지들도 따라서 제거되어야 하는데 정말로 제거하겠느냐는 질문이다. 이 질문에 Yes 라고 대답하면 관련 패키지들도 모두 제거된다. 디자인 패키지를 추가하려면 Add 버튼을 눌러주고 BPL 파일을 선택하면 된다.

다음은 런타임 패키지에 어떤 컴포넌트들이 포함되어 있는지 보자. 런타임 패키지는 모두 VCL 로 시작하며 윈도우즈의 시스템 디렉토리에 위치한다. 간단하게 표로 정리하였다.

| 패키지 | 내용 |
|---|---|
| VCL40.BPL | Windows, StdCtrls, System, SysUtil 등 대부분의 기본 유닛 |

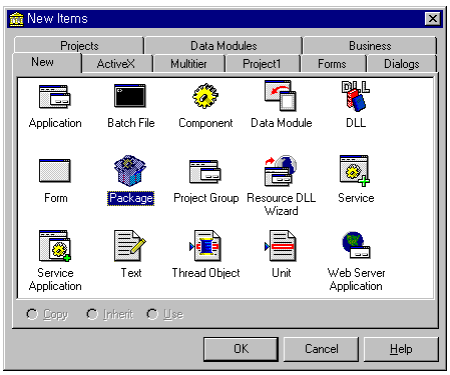
| | |
|---|---|
| VCLX40.BPL | ColorGrid, Outline, Spin, Tab |
| VCLDB40.BPL | 대부분의 기본 데이터 베이스 유닛 |
| VCLDBX40.BPL | DbLookup, Report |
| VCLJPG40.BPL | Jpeg 그래픽 포멧 지원 |

런타임 패키지를 사용했을 경우 이 중 필요한 패키지를 골라 프로그램과 같이 배포해야 한다. 대부분의 경우 VCL40.BPL 만 같이 배포하면 되며 데이터 베이스 프로그램이라면 VCLDB40.BPL 정도를 같이 배포하면 된다.

# 다. 패키지의 제작

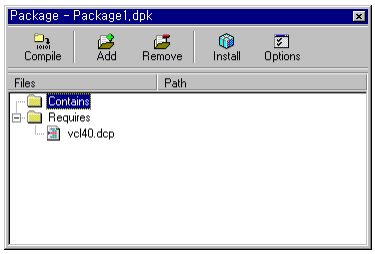
15jang
MyPkg

델파이가 제공하는 패키지를 사용하고 관리하는 방법만 알아도 패키지의 이점을 충분히 활용할 수 있겠지만 원할 경우 자신만의 패키지를 만들 수도 있다. 커스텀 패키지를 만들어 두고 자신이 즐겨쓰는 컴포넌트들을 여기에 모아 둔다거나 프로젝트별로 패키지를 만들어 두고 사용할 수도 있다. 새로운 패키지를 만드는 방법과 패키지의 구조에 대해 간단하게 실습을 해 보자. File/New 를 선택한 후 New 페이지에서 Package 를 선택한다.



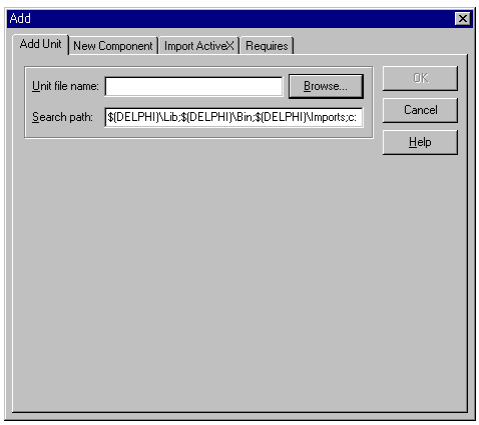

그러면 별다른 추가 질문없이 다음과 같이 생긴 패키지 편집기를 보여 줄 것이다.

그림
패키지 편집기



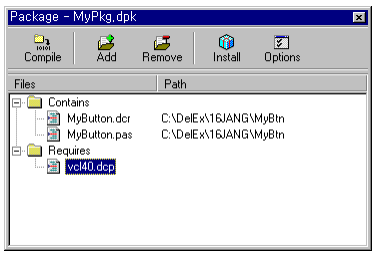

디폴트로 주어지는 패키지 이름은 Package1.dpk 인데 이 파일을 MyPkg.dpk 로 저장하도록 하자. 패키지 편집기 상단에는 몇 개의 툴 버튼이 있고 중앙에는 패키지의 내용을 보여주는 트리가 있다. 트리는 Contains 폴더와 Requires 폴더로 구성된다.

Requires 폴더에는 이 패키지를 컴파일하기 위해 필요한 패키지의 목록이 나타나는데 디폴트로 가장 기본적인 패키지인 vcl40.dcp 만 포함되어 있다. 다른 패키지가 더 필요하다면 상단의 Add 버튼을 눌러 이 폴더에 패키지를 포함해 주면 된다. Contains 폴더에는 이 패키지에 포함되는 유닛 파일의 목록이 나타나는데 새로 만든 패키지이므로 현재는 비어 있다. 이 폴더에 유닛을 추가하려면 역시 Add 버튼을 누르면 된다. 대중 통신망을 통해 구한 컴포넌트나 자신이 직접 만든 유닛을 이 폴더에 넣어주면 된다. 직접 Add 버튼을 눌러 보자.



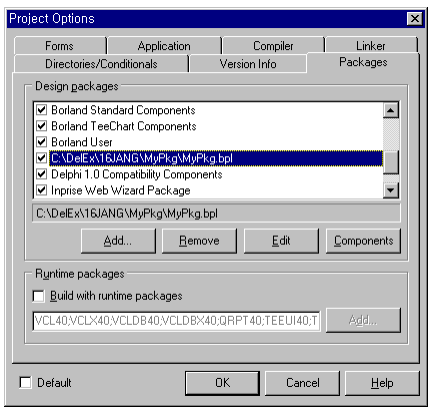

네 개의 탭으로 구성된 대화상자가 나타나는데 각 페이지는 미리 만들어져 있는 유닛, 새 컴포넌트, ActiveX 컨트롤, 그리고 Requires 폴더에 포함될 패키지를 읽어온다. 실습 삼아 New Unit 페이지에서 Browse 버튼을 눌러 CD-ROM의 16jang₩MyBtn₩MyButton.pas 파일을 읽어와 보자. 이 컴포넌트는 잠시 후에 만들어볼 컴포넌트인데 패키지 실습을 위해 사용하기로 한다. 아니면 자신이 구한 다른 컴포넌트를 사용해도 무방하다. 그러면 Contains 폴더에 이 파일

들이 나타날 것이다.




이제 이 패키지는 vcl40 패키지를 사용하며 MyButton 컴포넌트를 포함하게 되었다. Compile 버튼을 누르면 이 패키지가 컴파일되며 디스크에 MyPkg.bpl, MyPkg.dcp 파일이 생성될 것이다. 이 두 파일이 설치 가능한 패키이다. 이제 Install 버튼을 누르면 이 패키지가 델파이 개발 환경에 설치되며 이 패키지에 포함된 컴포넌트들이 팔레트에 나타날 것이다.


우리가 직접 만든 이 패키지도 델파이가 제공하는 패키지와 동일한 자격을 가지며 Project/Options/Packages 페이지의 패키지 리스트에 나타날 것이다.




# 라. 패키지의 구조

패키지 파일의 구조는 아주 간단하다. 소스 파일은 확장자 DPK 를 가지며 이

프로젝트 파일 하나가 패키지 그 자체이다.

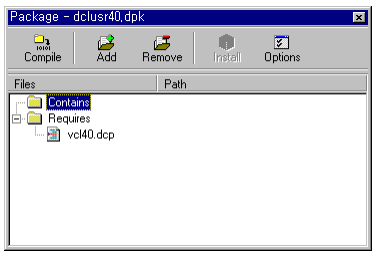
```
package MyPkg;

{$R *.RES}
{$R '..₩MyBtn₩MyButton.dcr'}
{$ALIGN ON}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO ON}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $00400000}
{$IMPLICITBUILD OFF}

requires
  vcl40;

contains
  MyButton in '..₩MyBtn₩MyButton.pas';

end.
```

package 예약어 다음에 패키지명이 있고 컴파일러 옵션이 나열되어 있다. 패키지는 델파이 개발 환경의 컴파일러 설정을 따르지 않고 자신의 컴파일러 설정을 소스 내부에 가지고 있는데 그래야 패키지가 시스템에 독립성을 유지할 수 있기 때문이다. 컴파일러 스위치 다음에는 requires 문과 contains 문이 있는데 이는 패키지 에디터의 requires, contains 폴더와 완전히 동일하다. 이외에 패키지

에 포함되는 유닛의 소스 파일도 패키지의 일부라고 할 수 있겠으나 패키지 자체와는 다른 것이므로 분석할 필요가 없다.

이 패키지를 컴파일하면 두 개의 파일이 생성된다. 확장자 BPL 로 된 파일이 진짜 패키지 파일이며 이 파일안에 패키지에 포함된 컴포넌트의 코드가 들어간다. 이 패키지를 사용하는 프로그램은 이 패키지를 같이 배포해야 한다. 확장자 DCP 로 된 파일은 컴포넌트의 심볼 정보가 들어가는데 이 파일도 역시 패키지와 함께 배포되어야 한다.

패키지는 여러 방면으로 활용할 수 있지만 아마도 가장 주된 용도는 새로운 컴포넌트를 구했을 때 이 컴포넌트를 설치하기 위한 그릇으로 사용하는 것일 것이다. 델파이는 이런 목적으로 dclusr40.dpk 를 제공하고 있으며 이 파일은 Lib 디렉토리에 있다.




디폴트로 이 패키지는 새로 만든 패키지와 완전히 같은 모양을 가지고 있는데 Contains 폴더에 설치하고자 하는 컴포넌트를 포함시켜주면 된다. 이 책에서는 앞으로 컴포넌트 설치를 위해 이 패키지를 사용할 것이며 델파이도 새 컴포넌트를 설치할 때 디폴트 패키지로 dclusr40.dpk 를 추천하고 있다.

# 16-3 기존 컴포넌트의 변경

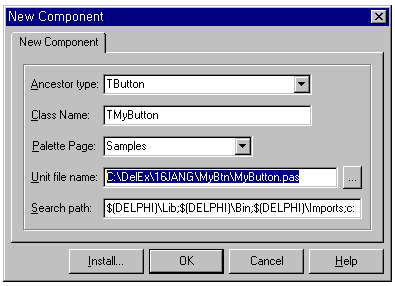
15jang
MyBtn

컴포넌트를 제작하는 과정은 무척 복잡하고 어려우며 알아야 할 규칙도 많다. 그렇다고 그 규칙을 다 공부한 후에나 실습해 볼 수는 없는 노릇이므로 여기서는 덮어놓고 일단 새로운 컴포넌트를 한 번 만들어 보기로 한다. 여기서 제시하는 순서를 따라 한 번 만들어 보면 대충 절차를 익힐 수 있을 것이며 앞으로 어떤 부분을 더 알아야 하는가를 알게 될 것이다. 여기서 만들어 볼 컴포넌트는 TButton을 수정하여 디폴트 속성을 변경한 TMyButton 컴포넌트이다. 델파이를 실행시킨 후 다음 절차를 따라 한다.

## 가. 마법사를 사용하는 방법

1. File/Close All 메뉴를 선택하여 모든 프로젝트를 닫는다. 꼭 그렇게 할 필요는 없지만 프로젝트가 열려 있으면 귀찮게 질문을 해 오기 때문에 프로젝트가 없는 깨끗한 상태에서 작업을 시작하는 것이 좋다.
2. Component/New Component 를 선택하여 컴포넌트 마법사를 불러낸다. 다음과 같은 대화상자가 나타날 것이다.

그림
컴포넌트 마법사




Ancestor type 란에는 기반 클래스의 이름을 기입해 주고 Class Name 란에는 우리가 만들고자 하는 컴포넌트의 클래스 이름을 기입해 준다. 우리는 지금 TButton 을 파생시켜 TMyButton 을 만들고자 하므로 Ancestor type 란에서 TButton 을 선택하고 Calss Name 란에 TMyButton 을 기입한다. Palette Page 란 새로 만든 컴포넌트가 배치될 컴포넌트 팔레트의 페이지이다. 컴포넌트를 만드는 사람이 마음대로 정할 수도 있지만 일단 디폴트로 주어진 Samples 를 그대로 받아들이도록 하자.

Unit file name 은 이 컴포넌트를 저장할 소스 파일의 경로와 이름을 지정하는데 디폴트로 delphi4₩lib 디렉토리에 클래스명과 같은 이름으로 지정되어 있다. 그냥 이 디렉토리에 두어도 되고 다른 별도의 디렉토리에 저장해도 상관없다. search path 는 유닛 파일을 찾을 경로를 지정하되 지금 만드는 유닛의 경로가 같이 포함되므로 특별히 수정해 주지 않아도 된다.

OK 버튼을 누르면 코드 에디터에 새로운 컴포넌트 유닛을 만들어 준다.

3 마법사가 만들어 준 코드에서 생성자를 변경한다. 아래 리스트는 컴포넌트 마법사가 만든 코드이며 이 중 굵은 문자로 된 부분이 우리가 추가로 입력한 부분이다.

```
unit MyButton;

interface

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

type
  TMyButton = class(TButton)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create(AOwner:TComponent);override;
  published
    { Published declarations }
  end;

procedure Register;

implementation

constructor TMyButton.Create(AOWner:TComponent);
begin
  inherited Create(AOwner);
  Caption:='MyBtn';
  Height:=89;
end;
```

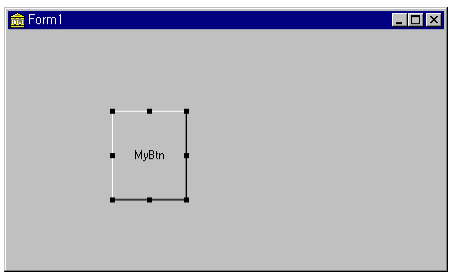
```
procedure Register;
begin
  RegisterComponents('Samples', [TMyButton]);
end;

end.
```

마법사가 만들어 준 코드에는 클래스 상속문과 컴포넌트를 팔레트에 등록시켜 주는 Register 프로시저가 포함되어 있다. 우리가 만든 생성자에서는 기반 클래스인 TButton의 생성자를 먼저 호출하여 필요한 초기화를 할 수 있도록 한 후 Caption 속성과 Height 속성을 변경하였다.

4 일단 만들어진 유닛을 디스크에 저장한다. 마법사 화면에 경로와 파일명을 이미 주었으므로 저장 명령만 내려주면 된다. 이제 MyButton.pas 파일에 우리가 만든 코드가 들어 있는데 이 파일이 우리가 만든 컴포넌트의 소스 파일이며 이 파일을 컴파일한 MyButton.DCU 파일이 설치의 대상이 되는 파일이다. 디스크에 유닛을 저장하면 컴포넌트를 만드는 일은 이미 끝이 났다. File/Close All 로 프로젝트를 닫아도 상관없다.

5 만들어진 컴포넌트를 설치한다. 앞에서 설명한 컴포넌트 설치 과정을 따라 컴포넌트를 라이브러리에 설치해 보자. 설치 중에 에러가 발생하면 오타일 가능성이 가장 크므로 다시 한 번 소스를 훑어본 후 설치한다. 설치한 후 Samples 페이지를 보면 우리가 만든 TMyButton 이 추가되어 있을 것이다.

6 만들어진 컴포넌트가 제대로 동작하는지 테스트해 보자. 새로운 프로젝트를 열어 컴포넌트를 폼에 배치해 보거나 속성을 변경해 봄으로써 테스트해 볼 수 있다.




기반 클래스의 TButton과 동일한 모양을 가지며 동일한 동작을 하지만 크기와 캡션이 변경된 새로운 컴포넌트가 폼에 나타날 것이다. 속성만 변경한 이런

컴포넌트라면 굳이 새로 만들 필요가 없지 않냐는 의문이 생길 것이다. 물론 그렇기는 하지만 똑같은 모양의 컴포넌트를 매번 사용한다면 속성을 설정하는 데 불필요한 시간을 들이지 않아도 되므로 편리한 면도 있다. 물론 이런 용도라면 컴포넌트 템플릿을 사용하는 방법이 더 편리하다.

# 나. 수작업으로 직접 만드는 방법

마법사를 사용하지 않고 직접 손으로 코드 에디터를 사용하여 컴포넌트를 만들 수도 있다. 물론 마법사를 사용하는 방법보다 편리하지는 않지만 제작 과정을 좀 더 자세히 살펴볼 수 있으며 한 유닛에 두 개의 컴포넌트를 제작할 때나, 폼이나 대화상자를 컴포넌트로 만들 때는 마법사를 사용할 수 없으므로 이 방법도 알아 두어야 한다.

1 마법사를 사용하는 경우와 마찬가지로 일단 모든 프로젝트를 닫아두어야 한다. 아무것도 없는 깨끗한 상태에서 새로운 유닛을 작성한다. 유닛이란 오브젝트 파스칼의 컴파일 단위이며 컴포넌트도 유닛 단위로 작성되므로 새 유닛을 만들어야 새 컴포넌트를 만들 수 있다. File/New 메뉴를 선택한 후 오브젝트 창고에서 Unit 을 선택하면 유닛의 뼈대를 만들어 코드 에디터에 편집 가능한 상태로 만들어 준다.

```
unit Unit1;

interface

implementation

end.
```

유닛 선언과 interface, implementation 및 유닛의 끝을 나타내는 end 등 가장 기본적인 구조만을 갖추고 있다. 일단 interface부에 uses절부터 추가하여 기본 유닛을 사용할 수 있도록 만들어 둔다.

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

2 기존 컴포넌트의 클래스에서 새로운 클래스를 파생시킨다. 이 단계에서

새로 만들 클래스의 이름, 그리고 어떤 클래스를 기반 클래스로 하여 파생할 것인가의 두 가지를 결정해야 한다. interface 부에 다음과 같은 형식으로 클래스를 파생시킨다.

```
type
TMyCompo=class(TComponent)
end;
```

이 선언에 의해 TMyCompo 클래스가 만들어졌다. 아직까지는 기반 클래스인 TComponent를 상속만 받았을 뿐 변형되거나 추가되지 않았으므로 두 클래스는 모든 면에서 동일하다.

3. 파생된 클래스를 변형 및 추가한다. 변형이나 추가의 대상이 되는 것은 속성, 메소드, 이벤트, 컴포넌트의 외형 등 컴포넌트의 모든 특성들이며 그 방법에 대해서는 앞으로 논하게 된다.

4. 생성된 클래스를 등록한다. 컴포넌트를 담고 있는 유닛에는 반드시 컴포넌트를 등록하는 Register 라는 프로시저가 있어야 한다. interface 부에 procedure Register;를 기입하고 implementation 부에 코드를 기입한다. register 프로시저에서 해야 할 일은 어떤 컴포넌트를 어떤 페이지에 추가할 것인가를 신고하는 일이며 ResisterComponents 프로시저가 사용된다. 이 프로시저의 첫 번째 인수는 새 컴포넌트가 등록된 컴포넌트 팔레트의 페이지이며 Standard, Additional 등의 기본 팔레트를 사용해도 되며 MyCompo, BABO 등과 같이 자기가 만들어 사용해도 된다. 현재 없는 팔레트 페이지를 지정하면 델파이가 새로운 팔레트 페이지를 만들어 준다. 두 번째 인수는 등록할 컴포넌트의 집합형이므로 여러 개의 컴포넌트를 한꺼번에 등록할 수도 있다.  작성한 유닛을 적당한 이름으로 디스크에 저장한다.

5. 컴포넌트를 설치한다. Component/Install 메뉴로 컴포넌트 설치 대화상자를 불러내며 등록하는 과정은 마법사를 사용하는 방법과 동일하다. 남은 일은 컴포넌트가 제대로 작동하는가 살펴보고 사용하는 일 뿐이다.

마법사를 사용하는 방법과 손으로 직접 만드는 방법은 사실 과정만 다르지 결과는 동일하다. 사람이 직접 해야 할 일을 마법사가 좀 더 편리하고 빠르게 해 준다는 차이점밖에는 없다.

# 다. 도움말과 비트맵의 제작

이 외에 컴포넌트 작성에 관계되는 일에는 컴포넌트에 관한 도움말을 만드는 일과 컴포넌트 비트맵을 만드는 일 등이 있다. 델파이는 여러 개의 도움말을 상호 참조하는 특별한 도움말 엔진을 사용한다. 그래서 사용자가 컴포넌트를 추가할 때 도움말까지도 델파이 도움말에 병합할 수 있도록 되어 있다. 컴포넌트는 도움말이 없으면 그 특성을 파악하기가 쉽지 않다. 그래서 컴포넌트 제작자는 컴포넌트와 함께 컴포넌트의 사용법을 도움말로 작성하여 제공해야 할 의무가 있으며 델파이는 이를 위해 자신의 도움말과 제작자의 도움말을 합칠 수 있도록 준비하고 있다.

컴포넌트 비트맵이란 팔레트에 그려질 컴포넌트 버튼의 모양을 말한다. 생략할 경우는 디폴트를 취하므로 없어도 상관은 없지만 가급적이면 비트맵은 만들어 주는 것이 좋다. 컴포넌트 비트맵은 다음 규칙에 따라 작성한다.

① 윈도우즈의 표준 리소스 형태로 제작하되 확장자는 반드시 DCR(Dynamic Component Resource)이어야 한다.

② 비트맵의 크기는 반드시 24*24 이어야 한다.

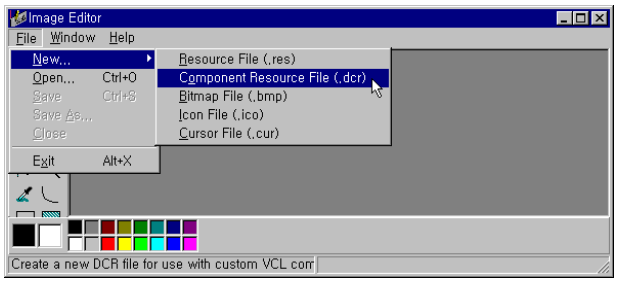
③ 리소스 파일의 이름은 반드시 유닛의 이름과 같아야 하며 유닛 파일과 같은 디렉토리에 있어야 한다.

④ 비트맵의 이름은 반드시 컴포넌트의 이름과 같아야 한다.

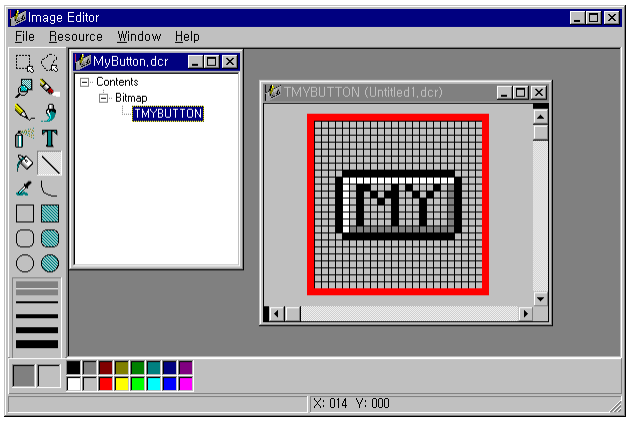
DCR 파일은 델파이가 제공하는 이미지 에디터로도 작성할 수 있다. 앞에서 우리가 만들었던 TMyButton 컴포넌트의 경우 유닛 이름이 MyButton.pas이므로 리소스 파일의 이름은 반드시 MyButton.dcr이어야 하며 컴포넌트의 이름이 TMyButton이므로 비트맵의 이름은 반드시 TMYBUTTON이어야 한다. 비트맵 이름은 대소문자를 구분하지는 않지만 관습적으로 대문자를 많이 쓴다. Tools/Image Editor를 선택하여 이미지 에디터를 실행시키고 File/New 의 하위 항목 중 DCR을 선택한다.

그림

이미지 에디터를 사용하여 DCR 파일을 작성한다.




그러면 DCR 프로젝트 윈도우가 열리며 Resource/New/Bitmap 항목을 선택하여 새 비트맵을 제작할 수 있다. 비트맵의 크기는 24*24를 사용하며 색상은 16색상을 사용하는 것이 좋다. 다음은 TMyButton에 사용된 비트맵 디자인 과정을 보인 것이다.




비트맵의 이름을 TMYBUTTON으로 변경하였으며 리소스 파일은 MyButton.dcr로 저장하였다. TMyButton을 설치할 때 MyButton.DCR을 같은 디렉토리에 위치시켜 두면 이 비트맵이 컴포넌트 팔레트에 나타날 것이다. 만약 새로 만든 컴포넌트의 비트맵이 정의되어 있지 않다면 기반 클래스의 비트맵을 사용한다.

# 16-4 컴포넌트의 세부 요소

## 가. 게이지 컴포넌트

15jang
MyGauge

이번에는 좀 더 복잡한 모양의 컴포넌트를 직접 만들어 보자. 설치 프로그램에서 흔히 사용되는 작업 진행 상황을 나타내는 막대 모양의 게이지를 만들 것이다. 물론 이런 게이지는 ActiveX로도 만들어져 있으며 델파이의 Samples 페이지에도 이미 만들어져 있지만, 예로 들기에 아주 적합하다고 생각되므로 다시 만들어 보았다. 마법사를 사용하여 유닛을 만든 후 비주얼 툴의 도움은 전혀 받지 않고 코드를 직접 작성하는 방법을 사용하였다. 물론 작성하는 과정에서 여러 차례의 시행 착오를 겪어야만 했다. 전체 소스는 다음과 같다.

```
{간단한 눈금을 나타내는 게이지 컴포넌트}
unit MyGauge;

interface

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

{그래픽 컨트롤로부터 상속받아 새로운 타입을 만든다.}
type
  TMyGauge = class(TGraphicControl)
  private
    FGauge:integer;           {눈금을 기억하는 변수}
    FGaugeColor:TColor;
    FOnGaugeChange:TNotifyEvent;  {사용자 정의 이벤트}
    procedure SetGauge(gag:integer);   {눈금 설정 메소드}
    procedure SetGaugeColor(C:TColor);
  protected
    procedure Paint;override;  {게이지를 그린다.}
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
  published
    property OnClick;     {여섯개의 이벤트를 가진다.}
    property OnDblClick;
```

```
    property OnMouseUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnGaugeChange:TNotifyEvent {사용자 정의 이벤트}
      read FOnGaugeChange write FOnGaugeChange;
    {Gauge 속성은 FGauge 를 읽으며 쓸 때는 SetGauge 메소드를 호출한다.}
    property Gauge:integer read FGauge
          write SetGauge default 0;
    property GaugeColor:TColor read FGaugeColor
      write SetGaugeColor default clRed;
    property Width default 100;
    property Height default 25;
    property Caption;
  end;

procedure Register;

implementation

{컴포넌트 팔레트의 Samples 페이지에 이 컴포넌트를 등록한다.}
procedure Register;
begin
  RegisterComponents('Samples', [TMyGauge]);
end;

{게이지의 생성자. 기반 클래스의 생성자를 먼저 호출한 후 높이
와 넓이를 디폴트값으로 맞추어준다.}
constructor TMyGauge.Create(AOWner:TComponent);
begin
  Inherited Create(AOWner);
  Height:=25;
  Width:=100;
  FGaugeColor:=clRed;
end;

{게이지의 파괴자. 기반 클래스의 파괴자를 호출한다.}
destructor TMyGauge.Destroy;
begin
  Inherited Destroy;
end;

{눈금을 설정한다. 인수로 전달된 변수로 FGauge 값을 설정하며
눈금이 0~100 사이에 있는지 점검해서 범위를 벗어나지 않도록 해 준다.
눈금을 다시 설정한 후에는 Paint 메소드를 호출하여 게이지를 다시
그리며 이벤트 핸들러를 호출한다.}
procedure TMyGauge.SetGauge(gag:integer);
```

```
begin
if Fgauge<>gag then
begin
 FGauge:=gag;
 if FGauge<0 then FGauge:=0;
 if FGauge>100 then FGauge:=100;
 {이벤트 핸들러가 지정되어 있으면 호출한다.}
 if Assigned(FOnGaugeChange) then OnGaugeChange(Self);
 Invalidate;   {Paint 메소드 호출}
end;
end;

procedure TMyGauge.SetGaugeColor(C:TColor);
begin
if FgaugeColor<>C then
begin
 FGaugeColor:=C;
 Invalidate;
end;
end;

{게이지를 그린다.}
procedure TMyGauge.Paint;
var
 cap:String;
begin
 with canvas do
 begin
  pen.color:=clgray;  {사각형을 먼저 그린다.}
  moveto(0,height);
  lineto(0,0);      {좌측, 상단은 회색으로}
  lineto(width-1,0);
  pen.color:=clwhite; {우측, 하단은 흰색으로}
  lineto(width-1,height-1);
  lineto(0,height-1);
  pen.color:=clblack; {속을 흰색으로 채운다}
  brush.color:=clWhite;
  pen.Width:=0;
  Rectangle(1,1,Width-1,Height-1);

  Brush.Color:=FGaugeColor; {FGauge 값만큼 막대를 그린다.}
  Pen.Width:=0;
  pen.color:=clblack;
  Rectangle(2,2,2+Round((Width-4)*(FGauge/100)),
   Height-2);
```

```
  cap:=IntToStr(FGauge)+' %'; {게이지 값 출력}
  brush.style:=bsclear;
  TextOut((width-textwidth(cap)) div 2,
    (height-textheight(cap)) div 2,cap);
 end;
end;
end.
```

Name, Caption, Color 등의 기본적인 속성이 정의되어 있으며 OnClick, OnDblClick 등의 기본적인 이벤트가 정의되어 있다. 또한 사용자가 직접 만든 Gauge, GaugeColor 속성과 OnGaugeChange 이벤트를 가진다. 이 소스의 문법은 이 절 전체를 통해 설명되므로 일단 MyGauge.pas라는 이름으로 디스크에 저장한 후 설치해 보자.

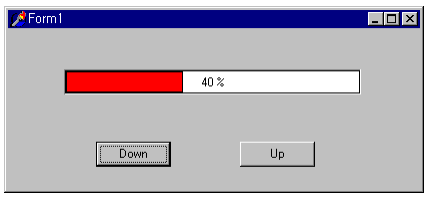
# 나. 게이지 사용

15jang
usegag.dpr

**그림**

게이지 컴포넌트 사용예

TMyGauge를 설치한 후  과연 제대로 동작하는지 살펴보기 위해 예제를 만들어 보았다. Up, Down 두 개의 버튼을 배치하여 버튼을 누르면 게이지의 막대를 증감시키며 OnGaugeChange 이벤트의 존재를 확인하기 위해 게이지 값이 변할 때마다 소리를 내는 예제이다




예제에서 사용된 코드는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MyGauge1.Gauge:=MyGauge1.Gauge-10;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  MyGauge1.Gauge:=MyGauge1.Gauge+10;
end;
```

```
procedure TForm1.MyGauge1GaugeChange(Sender: TObject);
begin
  MessageBeep(0);
end;
```

직접 예제를 보면 알겠지만 사용자가 만든 컴포넌트도 델파이가 기본적으로 제공하는 컴포넌트와 사용하는 방법에 있어서는 전혀 차별을 받지 않는다. 디자인 중에 속성을 마음대로 변경할 수 있으며 이벤트 핸들러도 작성할 수 있다.

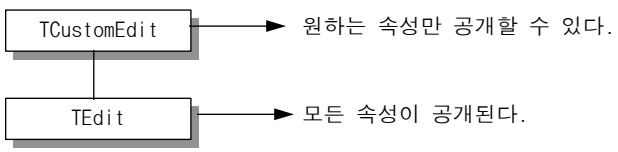
# 다. 기반 클래스 선택

자신이 만들고자 하는 컴포넌트를 어떤 클래스에서 파생시킬 것인가를 결정하려면 VCL 내의 모든 클래스 특징을 대충이나마 파악하고 있어야 한다. VCL의 클래스가 어디 한두 개도 아니고 클래스가 간단한 것도 아니어서 그 구조를 다 파악하기란 쉬운 일이 아니다. 그러나 만들고자 하는 컴포넌트에 따라 몇 가지 기준이 있으므로 이를 따르면 비교적 쉽게 기반 클래스를 선택할 수 있다.

## ■ 기존 컴포넌트를 변경할 경우

TMyButton의 예와 같이 기존 컴포넌트에서 디폴트 속성에 변경을 가하거나 약간만 추가할 경우가 이에 해당되며 이때는 TButton, TEdit, TLabel 등의 변경 대상 컴포넌트가 기반 클래스가 된다. 기존 컴포넌트를 파생시키면 기반 클래스의 모든 속성, 이벤트, 메소드를 그대로 상속받으며 상속받기 싫어도 기반 클래스의 속성을 버릴 수 있는 방법은 없다. 만약 정 상속받기 싫은 속성이 있다면 한 단계 위의 추상 클래스(abstract class)에서 상속받아야 한다.

그림
추상형 클래스




추상형 클래스는 대부분의 속성을 숨겨두어 필요한 속성만 선택적으로 공개할 수 있으며 이런 목적을 위해 존재하는 클래스이다. VCL 계층에서 Custom이라는 단어가 들어가는 클래스들이 모두 추상 클래스이다.

### ■ TWinControl

표준 컨트롤들은 모두 TWinControl에서 파생되며 이런 컨트롤을 윈도우 컨트롤(windowed control)이라 한다. 표준 컨트롤들은 모두 Handle 속성으로 표현되는 윈도우 핸들을 가지며 실행시에 포커스를 받을 수 있다. TButton, TCheckBox, TListBox, TForm 등이 윈도우 컨트롤이며 이런 특성을 가지는 컴포넌트를 만들고 싶다면 TWinControl을 기반 클래스로 사용한다.

### ■ TGraphicControl

비윈도우 컨트롤(nonwindowed control)은 윈도우 핸들을 가지지 않으며 따라서 실행시에 포커스를 받을 수 없다. 윈도우 핸들을 가지지 않으므로 리소스를 소비하지 않으며 다만 화면의 특정 위치에 정해진 모습으로 보이기만 할 뿐이다. 캔버스를 가지며 윈도우즈가 보내주는 WM_PAINT 메시지를 처리하므로 표면에 그림을 그릴 수 있다. TBevel, TShape 등의 컴포넌트가 비윈도우 컨트롤이며 이런 종류의 컴포넌트를 만들고 싶다면 TGraphicControl을 기반 클래스로 사용한다.

### ■ TComponent

비가시적(nonvisual) 컴포넌트를 만들 때 사용하는 기반 클래스이다. 비가시적이란 말은 실행중에 화면으로 보여지지 않는다는 뜻이며 타이머나 파일 오픈 대화상자 컴포넌트가 그 좋은 예이다.

게이지 컴포넌트는 윈도우 핸들을 가지지 않으며 포커스를 받을 필요도 없고 실행중에 그림을 보여주기만 하므로 TGraphicControl을 기반 클래스로 사용한다. 그래서 소스의 클래스 선언문이 다음과 같이 되어 있다.

```
type
  TMyGauge = class(TGraphicControl)
```

# 라. 속성

### ■ 속성과 변수

속성은 디자인시에 드러나는 컴포넌트의 특성일 뿐만 아니라 사용자는 속성

을 변경함으로써 컴포넌트를 마음대로 요리할 수 있으므로 대단한 편리함을 제공해 준다. 컴포넌트를 폼에 배치하기만 하면 오브젝트 인스펙터에 속성이 나열되므로 사용자는 변수에 값을 대입하는 방법처럼 속성값을 변경할 수 있다. 속성은 사실 변수와 거의 동일하며 실행중에 마치 변수인 것처럼 코드에서 속성값을 읽고 쓸 수 있다. 물론 속성과 변수는 완전히 같지는 않다. 우선 가장 큰 차이점이라면 부수적인 효과의 발생 여부를 들 수 있는데 변수는 값만 변하지만 속성값을 변경하면 단순히 값만 변하는 것이 아니라 부수적인 효과(side effect)가 발생한다.

그림
속성 변경에 의한 부수적 효과


Font 속성을 바꾸면 결과가 직접 폼에 나타난다.

예를 들어 버튼의 Width 속성을 변경하면 폼에 그려진 버튼의 크기가 동시에 같이 변한다거나 Name 속성을 바꾸면 Caption 속성도 같이 변하며 TQuery의 Active 속성을 바꾸면 쿼리의 SQL문이 실행되는 것 등이 부수적 효과의 한 예이다. 이는 속성 변경에 의해 단순히 값의 대입만 하는 것이 아니라 속성 변경시에 특정 메소드가 호출된다는 것을 짐작할 수 있게 해 준다. 또한 속성은 함수의 인수로 쓰일 수는 있지만 var를 사용하여 참조 호출에 사용할 수는 없다는 점이 변수와 다른 점이다.

## ■ 속성의 타입

속성이 변수와 동일하다면 속성도 변수와 마찬가지로 타입을 가진다. 정수형, 문자열형의 단순형이 가장 흔하며 진위형, 나열형, 집합형, 오브젝트형도 가능하다. 속성이 어떤 타입을 가지는가에 따라 오브젝트 인스펙터에 나타나는 모양이 달라지며 속성값을 변경하는 방법도 달라진다. 특히 오브젝트가 속성이 되는 경우는 세부 속성을 가지며 Font나 Items 속성과 같은 별도의 속성 편집기(property editor)를 가질 수도 있다.

## ■ 새로운 속성 만들기

새로운 속성을 만들려면 다음과 같은 요소를 만들어 주어야 하며 코드 에디터를 통해 직접 입력해야 한다.

❶ 속성 필드 : 실제 속성값을 기억하는 오브젝트 내부의 기억 장소이다. 관습적으로 이 값에는 접두어 F 를 붙이며 외부에서 함부로 값을 변경하는 것을 방지하기 위해 private 액세스 지정을 가진다. 예를 들어 Color 속성을 기억하는 속성 필드의 이름은 FColor 가 된다.

❷ 속성 읽기 메소드 : 속성 필드를 읽어오는 메소드이다. 속성값의 속성 필드를 읽는 것이 주된 임무이지만 때로는 복잡한 과정을 통해서 속성 필드의 값을 계산해 내기도 한다. 속성 읽기 메소드는 관습적으로 Get 으로 시작하며 읽기만 하므로 인수는 가지지 않는다. 예를 들어 Color 속성을 읽는 메소드라면 GetColor 라는 이름을 가진다.

❸ 속성 쓰기 메소드 : 속성 필드에 값을 대입해 주며 부수적 효과를 처리하기도 한다. 속성 필드와 같은 형의 인수를 하나 취하며 이 값을 속성 필드에 대입한다. 속성 쓰기 메소드는 관습적으로 Set 으로 시작한다. 예를 들어 Color 속성을 쓰는 메소드라면 SetColor 라는 이름을 가진다. 만약 쓰기 메소드가 정의되어 있지 않으면 이 속성은 읽기 전용 속성이다.

❹ 속성 선언 : 속성의 이름과 데이터 타입, 읽기, 쓰기 메소드를 지정한다. 여기서 지정한 속성의 이름이 오브젝트 인스펙터에 나타나는 속성의 이름이다. 읽기, 쓰기 메소드들은 모두 private 액세스 지정을 가지지만 속성 선언은 반드시 published 액세스 지정을 가져야만 한다. 그렇지 않으면 오브젝트 인스펙터에 속성이 나타나지 않는다.

참고하세요

관습은 모든 사람에게 편리함을 제공해 주고 여러모로 시간을 절약해 준다. 사실 관습은 어디까지나 관습이지 규칙은 아니므로 꼭 지키지 않아도 상관없다. 속성필드의 이름은 속성 이름 앞에 F를 붙이는 것이 관습이지만 자기 마음대로 Z를 붙이거나 X를 붙여도 상관없고 아예 연관이 없는 다른 이름을 붙여도 물론 문제가 없다. 쓰기 메소드를 Write로 시작한다거나 읽기 메소드를 Read로 시작해도 델파이가 컴파일하는 데는 지장이 없다. 하지만 관습을 어기면 자기만 불편해진다는 사실을 잊지 말아야 한다. 자기 마음대로 이름을 붙이면 소스는 거의 암호문이 되어 버릴 것이고 시간이 지난 후에는 자기도 이해하지 못하게 된다. 까다로운 규칙을 조금이라도 간단하게 사용하기 위해 만든 것이 관습이므로 자기 머리가 천재가 아니라는 것을 안다면 관습은 지키는 것이 좋으며 거의 대부분의 사람이 관습을 지키고 있다.

속성을 정의하는 예를 보자.

```
type
 TMyCompo = class(TComponent)
 private
  FKorea:integer;          {속성 필드}
  procedure SetKorea(Value:integer);  {속성 쓰기 메소드}
  function GetKorea:integer;         {속성 읽기 메소드}
 published  {속성 선언}
  property Korea:integer read GetKorea write SetKorea default 0;
```

속성의 이름은 Korea이며 이 속성은 오브젝트내의 FKorea라는 필드에 저장되며 읽을 때는 GetKorea 함수를 사용하고 쓸 때는 SetKorea 프로시저를 사용한다. GetKorea와 SetKorea 메소드의 코드는 implementation부에 기입되어 있을 것이다. 속성 읽기, 쓰기 메소드는 사용자가 직접 호출하는 것이 아니라 델파이가 알아서 호출해 준다. Korea 속성에 값을 대입하면 SetKorea가 호출되고 이 속성의 값을 읽으면 GetKorea가 호출된다. 물론 사용자의 눈에는 이런 과정이 전혀 보이지 않는다. 속성을 선언하는 property문의 일반 문법은 다음과 같다.

**기본 형식**

property 속성이름:타입 read 읽기메소드 write 쓰기메소드
  default 디폴트값;

속성 변경에 의한 부수적인 효과가 없을 경우는 read, write에 메소드 이름을 쓰지 않고 다음과 같이 곧바로 속성 필드의 이름이 올 수도 있다. 이 때는 속성을 읽거나 쓰면 메소드를 통하지 않고 곧바로 FKorea 속성 필드값을 읽어주거나 기록한다.

```
property Korea:Integer read FKorea write FKorea;
```

이렇게 속성 읽기, 쓰기 메소드를 통하지 않고 곧바로 속성값을 참조하는 방법을 직접 참조(Direct Access)라고 한다. 직접 참조 방법과 액세스 메소드 방법을 혼용하는 것도 가능하다. 즉 읽는 과정에는 부수적 효과가 없으므로 직접 참조를 하고 쓰는 과정에서는 부수적 효과가 발생하므로 액세스 메소드를 사용하는 방법이다. TMyGauge 컴포넌트의 Gauge 속성이 좋은 예이다. Gauge 속성

이 변하면 게이지를 다시 그려야 하는 부수적 효과가 있지만 값을 읽을 때는 부수적 효과가 발생하지 않는다.

```
property Gauge:integer read FGauge write SetGauge
    default 0;
```

속성 선언문의 default문은 컴포넌트가 처음 폼에 배치될 때 사용되는 디폴트값을 지정한다. 버튼을 폼에 처음 배치하면 디폴트 폭이 89로 되어 있다. 이는 버튼의 Width 속성이 89로 default 설정되어 있기 때문이다. 그러나 property 문에서 지정하는 default값은 실제로 속성값을 디폴트로 설정하지는 않기 때문에 컴포넌트 제작자가 생성자에서 속성의 디폴트값을 직접 설정해 주어야 한다. default문은 속성의 디폴트값을 직접 대입하는 것은 아니며 다만 컴포넌트가 DFM 파일에 저장될 때 그 속성값을 저장할 것인가 아닌가를 결정할 때만 사용한다. DFM 파일에 컴포넌트의 속성이 모두 저장되는 것은 아니며 default문에서 지정한 값과 다른 값을 가지는 속성값만 저장된다.

TMyGauge의 생성자에서는 Height 속성과 Width 속성의 디폴트값을 각각 25,100으로 설정하며 게이지의 초기 색상으로 빨간색을 사용한다. 속성 선언에 default문이 있을 뿐만 아니라 생성자에서 이 속성에 값을 대입해 준다.

```
constructor TMyGauge.Create(AOWner:TComponent);
begin
 Inherited Create(AOWner);
 Height:=25;
 Width:=100;
 FGaugeColor:=clRed;
end;
```

## ■ 속성 쓰기 메소드

속성 쓰기 메소드는 인수로 전달된 값을 속성 필드에 대입하는 일 외에도 부수적인 효과를 처리하는 중요한 일을 한다. 게이지 컴포넌트의 GaugeColor 속성을 변경하는 메소드를 보자.

```
procedure TMyGauge.SetGaugeColor(C:TColor);
begin
if FgaugeColor<>C then
begin
 FGaugeColor:=C;
```

```
  Invalidate;
 end;
end;
```

속성의 실제값을 가지는 FGaugeColor 필드에 인수로 전달된 C를 대입하고 있다. 인수 C의 값은 디자인시에 오브젝트 인스펙터에 입력된 값이거나 GaugeColor:=clRed 등과 같이 작성된 코드의 우변값이며 델파이가 속성 쓰기 메소드로 전달해 준다. 게이지의 색상값이 변하면 게이지가 변경된 색상으로 다시 그려져야 하므로 Invalidate 메소드를 호출하는 부수적 효과 처리를 하고 있다. 쓰기 메소드에서 한 가지 더 눈여겨 볼 것은 가급적이면 화면 출력을 억제하기 위해 현재 설정된 속성값과 새로운 속성값을 비교하여 변화가 있을 때만 속성 필드를 변경한다는 점이다.

## ■ 속성의 상속

새로운 컴포넌트는 기존 컴포넌트를 상속해서 만들어지며 이 때 기반 클래스가 가진 모든 속성을 상속받게 된다. 즉 기반 클래스가 가진 속성을 모두 가진다는 얘기다. 앞에서 만들어 본 TMyButton의 예를 보면 TButton이 가진 모든 속성을 그대로 이어받는다는 것을 확인할 수 있다. 그러나 만약 기반 클래스가 추상형 클래스라면 문제는 달라진다. 추상 클래스는 인스턴스를 생성할 목적으로 만들어져 있는 것이 아니라 상속을 위한 기반 클래스로서 존재하고 있기 때문에 모든 속성을 공개해 놓지 않았다. 대부분의 속성이 protected이거나 public으로 선언되어 있으며 published로 되어 있지 않으므로 상속은 되지만 사용자에게 공개되지는 않는다. 이런 경우 숨겨진 속성을 사용자가 사용할 수 있도록 공개하려면 protected나 public으로 선언되어 있는 속성을 published 속성으로 다시 선언해 주어야 한다. 다시 선언해 줄 때는 파생 클래스 선언에서 속성의 이름만 적어주고 타입은 밝히지 않아도 되며 default문으로 디폴트값을 변경할 수 있다.

예를 들어 다음과 같이 컴포넌트를 상속했다고 하자.

```
type
 TMyControl = class(TWinControl)
```

TWinControl은 Ctl3D라는 속성을 가지고 있으며 이 속성은 protected로 선언되어 있다. 이 클래스로부터 상속된 TMyControl도 물론 이 속성을 가지지만 이 속성은 protected 액세스 지정을 가지므로 디자인중에 오브젝트 인스펙터에 나타나지 않음은 물론 실행중에도 사용할 수 없다. 이 속성을 TMyControl에서

사용하고 싶다면 다음과 같이 다시 선언해 주어야 한다.

```
type
 TMyControl = class(TWinControl)
 published
  property Ctl3D;
```

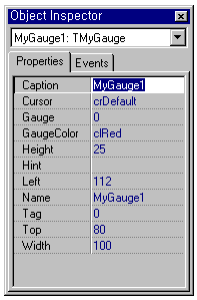
published로 액세스 지정을 변경하면 이 속성은 실행중에는 물론 디자인 시에도 오브젝트 인스펙터를 통해 사용할 수 있다. public으로 변경하면 디자인시에는 사용할 수 없지만 실행중에는 사용할 수 있다. 추상 클래스에서 모든 속성을 공개해 놓지 않은 이유는 꼭 필요한 속성만 선택적으로 사용할 수 있도록 하기 위해서이다.

액세스 지정의 변경은 액세스를 허락하는 방향으로만 변경할 수 있다. 즉 protected를 public으로 public을 published로 변경할 수는 있지만 published를 public이나 protected로 변경할 수는 없다. 따라서 기반 클래스에서 숨겨놓은 속성은 액세스 지정을 변경하여 공개할 수 있지만 일단 공개해 버린 속성은 파생 클래스에서 숨길 방법이 없다.

TMyGauge는 TGraphicControl로부터 Left, Top, Height, Width 등의 위치와 크기를 나타내는 속성과 Name, Tag, Hint, Cursor 등의 기본적인 속성 여덟 가지를 상속받으며 이 속성들은 기반 클래스에서 published 액세스 지정을 가지므로 별도의 처리를 하지 않아도 오브젝트 인스펙터에 나타난다. 그러나 이 일곱 가지 속성 외에도 Align, Font, Visible, Enable 등의 많은 속성도 같이 상속되며 다만 기반 클래스에서 published 속성을 가지지 않기 때문에 오브젝트 인스펙터에는 보이지 않는 것이다. 이런 속성들을 사용하고 싶다면 published로 액세스 지정을 변경해 주면 된다. 직접 소스를 확인해 보자.

```
published
  (중간 생략)
   property Caption;
```

TMyGauge에서는 Caption 속성을 published로 변경하여 오브젝트 인스펙터에 나타나도록 하였다. 그래서 TMyGauge의 오브젝트 인스펙터에 나타나는 속성은 다음과 같다.




### ■ 속성의 속성

델파이가 제공하는 컴포넌트들의 속성을 보면 속성도 여러 가지 성질을 가진다. Caption과 같은 가장 일반적인 속성이 있는가 하면 Canvas와 같이 디자인시에 오브젝트 인스펙터에는 나타나지 않지만 실행중에만 쓸 수 있는 속성이 있다. 또한 Owner와 같이 읽을 수만 있고 값을 변경할 수 없는 읽기 전용의 속성도 있다.

속성의 이런 속성은 액세스 지정과 쓰기 메소드의 존재에 의해 결정된다. 액세스 지정이 published로 되어 있는 속성은 디자인시나 실행시에 모두 사용할 수 있으며 public으로 되어 있는 속성은 실행시에 코드로만 사용할 수 있다. 그 외 private나 protected 액세스 지정을 가지는 속성은 내부로 숨겨지기 때문에 아예 사용할 수 없는 속성들이다.

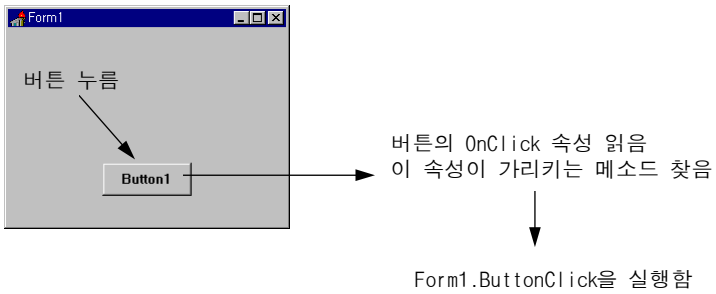
쓰기 메소드가 지정되어 있지 않으면, 즉 속성 선언문에 write문이 없으면 읽기 전용의 속성이 된다. 반대로 읽기 메소드가 지정되어 있지 않으면, 즉 read문이 없으면 이 속성은 쓰기 전용의 속성이 된다. 하긴 쓰기 전용의 메소드가 필요한 경우란 극히 드물겠지만 문법적으로는 분명히 가능하다.

## 마. 이벤트

이벤트는 특정한 사건과 코드(handler)를 연결하여 실행중에 컴포넌트의 행동을 정의한다. 윈도우즈에서는 끊임없이 사건이 발생하며 델파이는 각 사건에 대한 처리를 담당하는 메소드(event handler)를 만들어 이 사건들을 처리한다. 컴포넌트의 이벤트란 사건을 처리하는 메소드에 대한 포인터로서 존재한다. 특정 사건을 처리하는 핸들러를 만들었다면 이벤트는 이 핸들러에 대한 포인터값을 가지며 이 때 사건을 처리하는 핸들러는 오브젝트(=컴포넌트)내에 존재하는 것이 아니라 폼의 메소드 형태로 존재한다. 오브젝트는 다만 폼이 가지고 있는

메소드에 대한 포인터만을 가진다.

그림
이벤트 발생과 호출 과정




컴포넌트의 이벤트는 속성으로서 존재하며 속성을 다루는 방법과 동일하게 선언되고 작성하며 취급된다. 이벤트는 private 액세스 지정을 가지며 속성과 마찬가지로 접두어 F를 붙여 이름을 정한다. 예를 들어 TMyControl의 OnClick 이벤트라면 다음과 같이 정의된다.

```
type
  TMyControl = class(TComponent)
  private
    FOnClick:TNotifyEvent;
  published
    property OnClick:TNotifyEvent read FOnClick write FOnClick;
  end;
```

handler라는 이벤트 핸들러를 정의하면 OnClick 속성이 handler로 정의되며 이 값은 오브젝트 내의 FOnClick 필드에 저장된다. 이벤트 변경시에는 부수적 효과(side effect)가 발생하지 않으므로 이벤트를 정의하는 property문에는 별도의 액세스 메소드가 정의되어 있지 않으며 그래서 OnClick 속성의 read, write문에는 속성의 이름인 FOnClick이 기입되어 있다. 이벤트가 속성과 동일하게 취급되므로 속성과 마찬가지로 실행중에 대입문에 의해 이벤트를 변경하는 것이 가능하다.

## ■ 이벤트의 타입

이벤트도 속성이므로 다른 속성 필드처럼 타입을 가지고 있다. 델파이는 표준 이벤트에 대한 몇 가지 타입을 미리 정의해 두고 있으며 이벤트간의 호환을 위해 같은 타입에 대해서는 인수의 개수와 순서가 통일되어 있다.

표
이벤트 타입

| 메소드 포인터 타입 | 이벤트 |
|---|---|
| TNotifyEvent | OnCreate, OnClick, OnShow, OnPaint, OnResize |
| TMouseEvent | OnMouseDown, OnMouseUp |
| TMouseMoveEvent | OnMouseMove |
| TKeyEvent | OnKeyDown, OnKeyUp |
| TKeyPressEvent | OnKeyPress |

가장 일반적인 TNotifyEvent는 사건이 일어났음을 단순히 알리기만 하므로 특별히 전달되는 인수가 없다. 물론 모든 이벤트에서 이벤트를 발생시킨 컴포넌트를 구분하기 위해 사용하는 Sender 인수는 제외하고 말이다. 마우스 이벤트인 TMouseEvent는 눌러진 마우스의 버튼(Button)과 마우스 이벤트 발생시 키보드 상태(Shift) 그리고 마우스 이벤트가 발생될 때의 커서 위치(X,Y)가 인수로 전달된다.

같은 타입의 메소드끼리는 핸들러를 공유할 수 있다. 예를 들어 버튼의 OnClick 이벤트에서 정의한 코드는 폼의 OnCreate 이벤트에서 같이 사용할 수 있다. 두 이벤트 핸들러가 모두 TNotifyEvent 타입을 가지며 인수의 개수와 순서가 일치하기 때문이다. 반면 폼의 OnMouseMove 이벤트에서 정의한 코드는 버튼의 OnClick 이벤트에서 공유할 수 없다. 인수의 개수가 달라 이벤트 핸들러가 호환되지 않기 때문이며 오브젝트 인스펙터에도 호환되지 않는 이벤트는 나타나지 않는다.

이벤트 속성은 메소드에 대한 포인터형을 가지며 이 속성에 대입되는 핸들러는 모두 프로시저(procedure)이다. 함수(function)는 이벤트 핸들러로 지정될 수 없다. 이벤트에 핸들러가 꼭 지정되어 있어야 할 필요는 없으며 따라서 핸들러가 지정되지 않은 이벤트가 존재할 수 있다. 이 때는 아무 일도 하지 않는 비어 있는(empty) 프로시저를 이벤트 핸들러로 지정함으로써 문제를 해결한다. 그러나 함수는 아무 일도 하지 않는 비어 있는 형태가 없으므로 이벤트 핸들러가 될 수 없다. 이벤트는 모두 프로시저형이므로 인수를 받을 수는 있지만 공식적으로 리턴값을 되돌려 줄 수는 없다. 만약 이벤트에서 리턴값을 호출원으로 되돌려 주어야 한다면 참조 호출 인수를 사용해야 하며 DragOver 이벤트가 그 좋은 예이다.

## ■ 이벤트의 상속

속성과 마찬가지로 이벤트도 기반 클래스의 이벤트를 상속받는다. 상속되는

이벤트의 종류는 기반 클래스가 무엇인가에 따라 두 가지가 있다. TControl에서 상속되는 이벤트에는 다음과 같은 것들이 있다.

```
OnClick, OnDblClick, OnDragOver, OnDragDrop
OnEndDrag, OnMouseDown, OnMouseMove, OnMouseUp
```

TControl에는 이 이벤트들을 처리하는 이벤트 처리 메소드를 가지고 있으며 메소드의 이름은 이벤트의 이름에서 On을 뺀 이름이다. 예를 들어 OnClick 이벤트에 대한 메소드는 Click이다.

TWinControl에서 상속되는 컴포넌트는 TControl에 정의된 이벤트 외에도 다음과 같은 이벤트를 추가로 가진다.

```
OnKeyUp, OnKeyDown, OnKeyPress, OnEnter, OnExit
```

TWinControl은 윈도우 컨트롤을 만드는 시작점이 되며 윈도우 컨트롤은 포커스를 가지므로 키보드 입력을 받을 수 있다. 그래서 TControl이 가지는 이벤트 외에 키보드와 관련된 이벤트를 추가로 가진다.

표준 이벤트는 모두 protected 액세스 지정을 가지며 모두 숨겨져 있다. 상속에 의해 컴포넌트를 만들었을 경우 이 컴포넌트에는 사용자에게 공개된 이벤트가 하나도 없는 셈이 된다. 그래서 사용자에게 이벤트를 공개하기 위해서는 공개하고자 하는 이벤트를 public이나 published 액세스 지정을 가지도록 다시 선언해 주어야 한다. 그래서 TMyGauge 예제에서 다섯 개의 이벤트를 다시 정의하여 published로 액세스 지정을 변경하고 있다.

```
published
  property OnClick;
  property OnDblClick;
  property OnMouseUp;
  property OnMouseDown;
  property OnMouseMove;
```

액세스 지정을 바꾸어 다시 정의할 때는 이벤트의 이름만 밝혀주면 되며 이벤트의 타입이나 read, write를 지정해 줄 필요는 없다.

## ■ 이벤트 핸들러 변경

표준 이벤트에는 모두 숨겨진 이벤트 처리 메소드가 있으며 컴포넌트 제작 과정에서 새로운 메소드를 만들고 이 메소드가 이벤트를 처리하도록 이벤트 핸들러를 바꿀 수 있다. 바꾼 이벤트 핸들러에서는 원래의(Inherited) 이벤트 핸들러를 반드시 호출해 주어 이벤트에 대한 고유의 처리를 할 수 있도록 해 주어야 한다. 예를 들어 OnClick 이벤트에 대한 이벤트 처리 메소드인 Click을 다시 정의한다면 다음과 같이 된다.

```
procedure TMyControl.Click;
begin
 Inherited Click;  {원래의 이벤트 핸들러를 호출한다.}
  여기에 추가로 코드를 작성한다.
end;
```

이 때 기반 클래스의 메소드를 호출하는 순서가 때에 따라서는 중요한 의미를 가진다. 일반적으로 위의 경우와 같이 제작자가 작성한 이벤트 핸들러가 호출되기 전에 기반 클래스의 메소드가 호출되는 것이 보통이지만 반대로 되는 경우도 있다. 기반 클래스에 정의된 메소드의 동작에 영향을 미치는 특정한 조건을 만들어 주어야 한다거나 하는 경우가 이에 해당한다.

## ■ 새로운 이벤트의 작성

필요한 대부분의 이벤트가 기반 클래스에 정의되어 있기 때문에 파생 클래스에서 별도의 이벤트를 정의할 필요는 거의 없다. 물론 꼭 필요하다면 사용자가 직접 자신만의 이벤트를 정의할 수 있다. 게이지 컴포넌트에서는 OnGaugeChange라는 이벤트를 정의하고 있으며 이 이벤트는 Gauge 속성이 변할 때마다 발생한다.

이벤트도 어차피 속성이므로 속성을 만드는 방법과 비슷한 절차로 이벤트를 작성한다. 우선 이벤트값을 기억할 필드가 필요하며 이 필드는 private 액세스 지정을 가진다.

```
private
 FOnGaugeChange:TNotifyEvent;
```

관행에 따라 이벤트 이름 앞에는 F가 붙어 있으며 이벤트의 타입은 단순히 게이지값이 바뀌었음을 알리기만 하므로 TNotifyEvent이다. 이 필드는 이벤트 핸들러로 지정된 메소드의 포인터값을 가지게 된다. 이벤트 속성을 정의하는 문장

은 다음과 같이 되어 있으며 오브젝트 인스펙터에 이벤트가 나타나기 위해 published 액세스 지정을 가짐을 유의하자.

```
published
 property OnGaugeChange:TNotifyEvent
  read FOnGaugeChange write FOnGaugeChange;
```

보다시피 read, write문에 직접 필드 이름이 기입되어 있다. 왜냐하면 이벤트 변경에 의한 부수적 효과란 있을 수 없기 때문이다. 이벤트 속성의 이름은 관행에 의해 이벤트 이름 앞에 On을 붙인다. 여기까지 작성하면 이벤트는 이미 만들어졌으며 마지막으로 적당한 위치에서 이벤트 핸들러를 호출해 주어야 한다. 어느 시점에서 이벤트를 호출할 것인가는 이벤트의 정의에 따라 달라진다. 게이지의 경우는 FGauge 필드가 변할 때마다 이벤트가 발생하므로 FGauge값이 변하는 시점인 SetGauge 프로시저에서 이벤트 핸들러를 호출하도록 해 주는 것이 가장 적절하다.

```
procedure TMyGauge.SetGauge(gag:integer);
begin
if Fgauge<>gag then
begin
 FGauge:=gag;
 if FGauge<0 then FGauge:=0;
 if FGauge>100 then FGauge:=100;
 {이벤트 핸들러가 지정되어 있으면 호출한다.}
 if Assigned(FOnGaugeChange) then OnGaugeChange(Self);
 Invalidate;   {Paint 메소드 호출}
end;
end;
```

이벤트가 메소드의 포인터이므로 이벤트의 이름만 적어주면 이벤트 핸들러를 호출할 수 있다. 그러나 그렇다고 무작정 OnGaugeChange(Self)라고 호출해서는 절대로 안된다. 왜냐하면 이벤트에 핸들러가 반드시 할당되어 있다고 보장할 수 없기 때문이다. 그래서 Assigned 함수를 사용하여 메소드 포인터가 nil이 아닌가 점검해 보고, 즉 핸들러가 할당되어 있는가를 점검해 보고 핸들러가 정의되어 있을 경우에만 호출을 해야 한다.

게이지의 OnGaugeChange 이벤트는 발생 시점이 FGauge 필드가 변할 때이므로 내부적으로 핸들러를 호출할 위치를 쉽게 구할 수 있다. 그러나 좀 더 특수한 이벤트의 경우 컴포넌트 내에서 이벤트가 발생하는 것이 아니라 외부에서

발생하는 경우가 있으며, 이 때는 윈도우즈가 보내주는 메시지와 직접 연결하여 메시지 처리 메소드에서 적당한 조건이 되면 이벤트 핸들러를 호출해 주어야 한다.

# 바. 메소드

컴포넌트의 메소드는 오브젝트의 메소드와 개념적으로 동일하며 강제적으로 꼭 이래야만 한다는 규칙이 없는 셈이다. 그래서 속성이나 이벤트에 비해 메소드는 만들기가 더 쉽다. 하지만 강제 규칙이 없다고 해서 마음대로 만들어도 되는 것은 아니며 좀 더 사용하기 쉽고 직관적이도록 하는 몇 가지 관습이 있다.

## ■ 최소 개수

우선 메소드의 개수는 적을수록 좋다. 메소드는 실행중에만 사용할 수 있는 반면 속성은 디자인시에나 실행시에 모두 사용할 수 있으며 메소드보다는 속성이 훨씬 사용하기 쉬우므로 가급적이면 메소드를 속성으로 바꾸는 것이 좋다. 예를 들어 Left, Top이라는 속성으로 컴포넌트의 위치를 읽고 쓸 수 있는데 굳이 Move라는 메소드를 만들 필요가 없다는 얘기다.

그리고 메소드는 아무때나 호출될 수 있어야 한다. 특정한 전제 조건이 있다거나 먼저 어떤 메소드가 호출되어져야 하는 경우란 바람직하지 못하다. 예를 들어 methodB가 호출되려면 그 전에 methodA가 먼저 호출되어져 methodB가 제대로 동작할 수 있도록 해 주어야 하는 경우가 이에 해당된다. 이때는 methodB 내부에서 자신이 제대로 동작할 수 있는 상황이 되는가 점검해 보고 그렇지 않으면 알아서 methodA를 먼저 호출하도록 해 주는 것이 좋다. 메소드 호출에 이런 전제 조건이 있다면 사용자는 메소드 사용에 대한 주의사항을 숙지해야 할 것이고 이는 좀 더 편한 개발 환경을 만들고자 하는 비주얼 툴 본래의 취지에 어긋난다. 물론 어쩔 수 없이 순서를 지켜가며 메소드를 호출해 주어야 하는 불가피한 경우도 있다. TQuery에 SQL문을 대입해 줄 때가 이에 해당한다. 꼭 필요한 경우를 제외하고는 메소드 사용에 대한 주의사항을 만들지 않도록 해야 한다.

## ■ 메소드의 이름 결정

메소드의 이름은 물론 제작자가 마음대로 붙일 수 있지만 가급적이면 설명적인 것이 좋다. 메소드의 이름만 보고도 어떤 동작을 할 것인가를 쉽게 짐작할 수

있도록 작성해 두어야 한다. 델파이가 제공하는 컴포넌트의 메소드는 BringToFront, LoadFromFile 등과 같이 아주 설명적인 이름을 사용하므로 힘들여 외울 필요도 없이 보기만 해도 곧바로 메소드를 이해하고 사용할 수 있다. 만약 이런 메소드의 이름을 BTF, LFF 또는 method34 등과 같이 붙이면 사용자는 일일이 매뉴얼이나 도움말을 뒤져야 할 것이다. 이름이 좀 길어지더라도 이름 자체가 메소드를 잘 설명할 수 있어야 하며 그래서 메소드의 이름은 대부분 동사를 사용한다. 값을 바꾸는 메소드라면 WhatValue나 Value2보다는 ChangeValue가 적당하다. 함수 메소드의 경우는 함수의 동작을 설명하는 것보다는 리턴값의 의미에 대해 직관적으로 알 수 있도록 이름을 붙이는 것이 좋다.

## ■ 메소드의 액세스 지정

메소드의 액세스 지정은 다음 규칙을 따라 지정하면 별 문제가 없다. 메소드가 published 액세스 지정을 가져야 할 경우는 없다.

① public 이어야 하는 경우 : 메소드는 사용자가 언제든지 호출할 수 있어야 하므로 public 액세스 지정을 가지는 경우가 가장 많다.

② protected 이어야 하는 경우 : protected 속성을 가지는 메소드는 사용자가 직접 호출할 수 없는 메소드이다. 다른 메소드에서 내부적으로 사용하는 메소드들이 보통 protected 액세스 권한을 가진다. 오브젝트 내부에서 사용하는 이런 메소드는 보통 저수준적인 처리를 하기 때문에 사용자가 함부로 호출할 수 있도록 허락해서는 안된다. 언뜻 생각하면 private 액세스 지정을 해야 할 것 같지만 private 로 지정할 경우는 다른 컴포넌트로 상속할 경우 메소드가 상속되지 않게 되므로 문제가 발생하게 된다.

③ privated 이어야 하는 경우 : 속성을 읽고 쓰는 메소드들이 private 액세스 지정을 가진다. 사용자는 속성을 통해 이 메소드를 간접적으로 사용할 뿐 직접 호출할 수는 없다.

# 16-5 여러 가지 컴포넌트

이제 여러분은 컴포넌트 제작에 관한 기본 지식의 일부를 살펴 보았다. 왜 일부인가 하면 이 책에서 논한 내용은 아주 기초적이며 초보적인 단계에 불과하기 때문이다. 나머지는 일일이 이론을 나열하는 것보다 만들어져 있는 컴포넌트의 소스를 분석해 보는 것이 더 효율적이라 생각한다. 여기서는 필자가 만든 것과 제3자들이 만든 컴포넌트 중 소스가 제공되는 몇 가지를 소개하기로 한다. 배포 CD에 소스가 있으므로 직접 분석해 보아라.

## 가. 폼을 컴포넌트로 만들기

15jang
yoil.pas

폼도 사실 알고 보면 컴포넌트이므로 사용자가 디자인한 폼을 컴포넌트로 만들어 팔레트에 등록할 수 있다. 자주 사용되는 폼이라면 매 프로젝트마다 만드는 것 보다는 한 번 컴포넌트로 만들어 두고 편하게 사용하는 것이 더 효율적이다. Dialog 페이지에 있는 파일 열기, 색상 선택 대화상자 컴포넌트들이 그 좋은 예이다. 그러나 폼은 다른 컴포넌트와는 다른 특수함을 가지고 있으므로 컴포넌트화 하기도 제일 어렵다. 단계별로 따라하면서 폼을 컴포넌트로 만들어 보자.

### ■ 폼 디자인

일단 컴포넌트로 만들고자 하는 폼을 먼저 만든다. 이 과정은 보통 폼을 디자인하는 과정과 동일하며 비주얼 툴을 모두 사용할 수 있다. 여기서 보이고자 하는 예제는 요일을 선택하는 대화상자 컴포넌트이다. 물론 이 따위 대화상자를 컴포넌트로 만들어봐야 무슨 소용이 있겠는가만 어디까지나 예제이므로 실용성은 논하지 말자. 일단 버튼 8개를 폼에 배치하고 캡션을 다음 그림과 같이 변경한다.

**그림**
요일 선택 대화상자


각 버튼의 요일값을 버튼의 tag 속성에 순서대로 0~6까지 설정해 두며 버튼의 OnClick 이벤트에서 tag 속성값을 읽어서 요일값을 입력받도록 하자. 폼의

속성을 다음과 같이 변경한다.

| 속성 | 속성값 | 설명 |
|---|---|---|
| Name | yoilform | |
| Caption | 요일을 고르세요 | |
| BorderStyle | bsDialog | 대화상자 모양의 경계선을 가진다. |
| Position | poScreenCenter | 대화상자이므로 화면의 중앙에 나타나는 것이 제일 무난하다. |

폼에서 선택한 요일값을 담기 위해 NowYoil이라는 필드를 폼의 타입 선언부에 private 액세스 지정으로 선언한다.

```
private
  NowYoil:integer;
```

폼의 버튼을 누르면 선택한 요일의 값을 NowYoil 필드에 대입할 것이며 이 값이 곧 폼의 리턴값이 된다. 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure Tyoilform.Button1Click(Sender: TObject);
begin
  NowYoil:=(Sender as TButton).Tag;
  ModalResult:=mrOk;
end;

procedure Tyoilform.Button8Click(Sender: TObject);
begin
  ModalResult:=mrCancel;
end;
```

Button1Click 핸들러를 요일을 선택하는 일곱 개 버튼의 OnClick 이벤트에 모두 연결한다. 핸들러 내에서는 이 핸들러를 호출한 컴포넌트의 tag 속성값을 읽어 요일을 알아내며 이 값을 NowYoil에 대입한다. Cancel 버튼에는 대화상자를 닫는 코드가 작성되어 있다.

## ■ 컴포넌트화

여기까지 폼을 만드는 과정은 모두 끝났다. 문제는 이 폼을 컴포넌트로 만들

어야 하는 지금부터다. 폼이 그대로 컴포넌트가 될 수 없으므로 폼을 컴포넌트 형태로 잘 포장해야 하며 다음과 같은 절차가 필요하다. 여기서부터는 델파이의 그 편리한 비주얼 개발 환경을 전혀 사용할 수 없다. 오로지 키보드로 직접 코드를 작성하는 수밖에 없다.

1 컴포넌트의 타입을 선언한다. 수작업으로 컴포넌트를 만들 때와 같이 기반 클래스로부터 컴포넌트를 파생시키는 과정이다. 우리가 만들 요일 선택 대화상자에 TYoilCompo 라는 이름을 붙여주고 컴포넌트의 원조격인 TComponent 클래스를 기반 클래스로 사용한다.

2 Register 프로시저를 작성한다. 마법사를 사용할 때는 마법사가 등록해 주지만 수작업으로 작성할 때는 직접 등록 프로시저를 만들어야 한다. interface 부에 헤더를 선언하고 implementation 부에 코드를 작성한다. 새 컴포넌트를 배치할 페이지는 Samples 이다.

3 interface 부의 끝 부분에 있는 다음 코드를 삭제한다. 이 코드는 폼을 정적으로 생성시키지만 여기서는 컴포넌트에 의해 동적으로 생성되어야 하므로 마땅히 삭제되어야 한다.

```
var
 YoilForm1:TYoilForm;
```

4 이제 컴포넌트 자체의 속성과 메소드를 작성한다. 어떤 메소드를 작성할 것인가는 컴포넌트의 기능에 따라 달라지겠지만 이 경우는 Execute 메소드 하나만 작성하면 된다. 대화상자 컴포넌트는 보통 Execute 메소드만 제공한다.

이상 위에서 열거한 5단계를 모두 거쳐 만들어진 코드는 다음과 같으며 알아보기 쉽게 주석을 붙여 보았다. 리스트에서 굵게 표시된 코드가 폼을 만든 후 키보드로 직접 입력한 부분이다. 물론 비주얼 툴의 도움은 전혀 받지 않았다.

```
{요일 선택 대화상자를 컴포넌트로 등록한다.}
unit Syoil;
```

```
interface

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

{컴포넌트 자체의 타입 선언}
type
  TYoilCompo=class(TComponent)
    private
      FYoilNum:integer;   {요일 숫자}
      FYoilStr:string;    {요일 문자}
    public
      function Execute:Boolean;
    published
      property YoilNum:integer   {요일 숫자 속성}
        read FYoilNum write FYoilNum;
      property YoilStr:string    {요일 문자 속성}
        read FYoilStr write FYoilStr;
    end;
procedure Register;

{컴포넌트에 포함되는 폼의 타입 선언}
type
  Tyoilform = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Button6: TButton;
    Button7: TButton;
    Button8: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button8Click(Sender: TObject);
  private
    NowYoil:integer;  {요일 값을 잠시 기억한다.}
    { Private declarations }
  public
    { Public declarations }
  end;

implementation
{$R *.DFM}

function TYoilCompo.Execute:Boolean;
```

```
var
  YForm:TYoilForm;
begin
{폼을 동적으로 생성한다.}
YForm:=TYoilForm.Create(Self);
{폼을 실행한 후 결과를 속성에 대입한다.}
if YForm.ShowModal=mrOk then
begin
  FYoilNum:=YForm.NowYoil;
  case FYoilNum of
    0:FYoilStr:='월';
    1:FYoilStr:='화';
    2:FYoilStr:='수';
    3:FYoilStr:='목';
    4:FYoilStr:='금';
    5:FYoilStr:='토';
    6:FYoilStr:='일';
  end;
  Result:=True;
end
else
  Result:=False;
end;

{요일 버튼을 누를 경우 NowYoil 값에 기억시켜둔다.}
procedure Tyoilform.Button1Click(Sender: TObject);
begin
NowYoil:=(Sender as TButton).Tag;
ModalResult:=mrOk;
end;

{취소 버튼을 누르면 mrCancel 을 리턴한다.}
procedure Tyoilform.Button8Click(Sender: TObject);
begin
ModalResult:=mrCancel;
end;

{등록 프로시저}
procedure Register;
begin
  RegisterComponents('Samples',[TYoilCompo]);
end;

end.
```

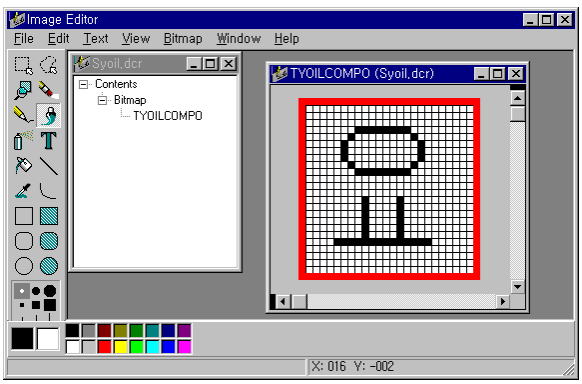
그럼 이제 이 코드를 분석하기만 하면 된다. YoilCompo가 사용하는 속성은 FYoilNum과 FYoilStr 두 가지가 있다. 전자는 요일을 숫자 형태로 기억하며 후자는 요일을 문자 형태로 기억한다. 같은 내용의 정보이지만 사용자의 편의를 위해 두 가지 형태를 모두 지원한다.

Execute 메소드는 TYoilForm 클래스의 Create 메소드를 사용하여 폼을 동적으로(실행중에) 생성시킨 후 ShowModal 메소드로 폼을 호출한다. 폼에서는 사용자가 누른 버튼에 따라 적당한 값을 NowYoil 필드에 대입해 줄 것이며 폼이 닫힌 후 이 값을 다시 FYoilNum, FYoilStr 속성에 대입한다. 결국 이 컴포넌트의 사용자가 최종적으로 얻을 수 있는 값은 FYoilNum, FYoilStr 속성 두 가지이다.

## ■ 설치

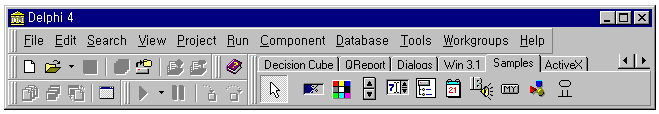
이 파일을 일단 디스크에 저장하되 프로젝트는 저장할 필요가 없으며 유닛 파일인 PAS만 저장하면 된다. 컴포넌트의 실체는 PAS를 컴파일한 DCU 파일이므로 DPR 파일은 필요가 없다. 단 이 경우 폼의 내용을 담고 있는 DFM 파일은 있어야 한다. 유닛만 SYOIL.PAS라는 이름으로 적당한 디렉토리에 저장하자.

마지막 단계는 컴포넌트 페이지에 나타날 비트맵을 만드는 것이다. 비트맵은 꼭 만들지 않아도 상관이 없지만 YoilCompo는 비가시적 컴포넌트이기 때문에 비트맵을 만드는 것이 좋다. 비가시적 컴포넌트는 폼에 배치할 경우 비트맵만 보이기 때문에 비트맵이 정의되어 있지 않으면 디자인시에 구분하기가 어려워진다. 다음과 같이 대충(정말 대충) 비트맵을 디자인 하였다. 필자는 초등학교 때부터 미술에 별다른 재질을 보이지 못했으며 지금도 별 변화가 없는 것 같다.




비트맵의 이름은 반드시 컴포넌트 이름과 같은 YOILCOMPO여야 하며 DCR 파일 이름은 반드시 유닛 파일 이름과 같은 SYOIL.DCR이어야 한다. 또한 DCR

파일은 반드시 SYOIL.PAS 파일과 같은 디렉토리에 있어야 한다. 그럼 이제 앞에서 배운 바대로 컴포넌트를 설치해 보자. 설치가 완료된 후의 모습은 다음과 같다.

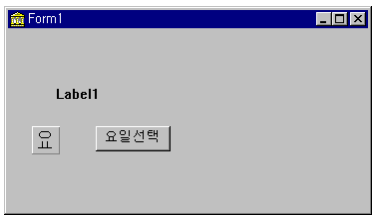



Samples 페이지 제일 오른쪽에 '요'라고 적힌 컴포넌트가 나타난다.

## ■ 요일 컴포넌트 사용


15jang
useyoil

그럼 애써 만든 컴포넌트가 제대로 동작하는지 시운전을 해 보도록 하자. 새로운 프로젝트를 시작한 후 YoilCompo 하나, 레이블, 버튼을 배치하고 버튼의 캡션을 “요일선택”으로 바꾼다.




보다시피 YoilCompo는 비가시적 컴포넌트이기 때문에 폼 상에는 비트맵만 나타나며 실행중에 이 비트맵은 보이지 않는다. 버튼을 누르면 요일 선택 대화상자가 나타나도록 다음과 같이 OnClick의 이벤트 핸들러를 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if YoilCompo1.Execute then
    label1.caption:=YoilCompo1.YoilStr+'요일을 선택했습니다.'
  else
    label1.caption:='요일 선택을 취소했습니다.';
end;
```

사용 방법은 파일 열기 대화상자나, 폰트 선택 대화상자를 사용하는 방법과 완전히 동일하다. 선택한 결과는 레이블에 나타난다.

이상으로 컴포넌트 제작에 관한 간단한 몇 가지 실습을 마친다. 컴포넌트 제작에 관해 더 상세한 내용을 알고 싶은 사람은 CWG.HLP 도움말을 참고하기 바

라며 소스가 필요한 사람은 인터넷을 검색해 보기 바란다. 그러나 학습을 위한 소스라면 그보다는 델파이와 함께 설치되는 VCL 의 소스를 참고하는 편이 가장 효율적이다. 델파이 C/S 버전에는 VCL 의 모든 컴포넌트에 대한 소스가 SOURCE/VCL 디렉토리에 제공되고 있다. 공부해 보고 비슷한 컴포넌트를 만들 수도 있고 아니면 기존 컴포넌트를 마음대로 뜯어 고칠 수도 있다.

# 16-6 ActiveX 컨트롤

## 가. ActiveX 란

ActiveX 는 마이크로소프트의 최신 기술을 총 집약한 용어이다. 역사적으로 볼 때 ActiveX 는 OLE 기법을 개선한 것이라고 할 수 있지만 사실은 그보다 훨씬 더 많은 기술들이 포함되어 있다. ActiveX 의 기술에는 오토메이션, ActiveX 도큐멘트, ActiveX 컨트롤 등이 대표적이지만 이 외에도 여러 가지 기법들이 포함되어 있으며 아직도 계속 새로운 기술들이 발전 개선되어가고 있다.

AcitveX 의 이론적 기반은 COM(Component Object Model)이라는 것이다. 그래서 ActiveX 를 제대로 이해하려면 COM 에 대한 이론적인 학습이 반드시 선행되어야 한다. COM 을 정의하자면 소프트웨어의 부품인 컴포넌트끼리 데이터와 정보를 교환하고 상호작용하는 방법에 대한 명세이다. 소프트웨어끼리 통신하고 상호작용하는 방법은 COM 이전에도 DDE, DLL, VBX 등 여러 가지 이론들이 있었다. 이런 과거의 이론들에 비해 COM 은 다음과 같은 장점을 가진다.

① 언어에 독립적이다. COM 은 어디까지나 사양, 또는 명세(영어로 Specification) 규정일 뿐이며 내부적인 구현까지 정해 놓은 것은 아니다. 그래서 COM 명세만 따른다면 어떤 언어로든지 COM 을 활용할 수 있다. C/C++은 물론이고 비주얼 베이직, 델파이 등의 RAD 툴, 파스칼, 자바, 심지어 어셈블리로도 COM 을 사용할 수 있다. COM 을 위한 언어의 최소 사양은 포인터, 그 중에서도 함수 포인터를 지원하기만 하면 된다.

② 이진(binary) 표준이다. 컴파일된 결과인 DLL 이나 EXE 또는 OCX 파일이 COM 객체가 되며 이진 형태로 배포될 수 있고 재사용될 수 있다. MFC 나 OWL 의 C++ 클래스 라이브러리처럼 소스 코드가 반드시 필요한 것이 아니다. 그래서 훨씬 더 배포하기 쉽고 사용하기도 쉽다.

③ 위치에 투명하게 사용된다. 컴포넌트가 어디에 설치되어 있더라도 사용가능하다. 단 사용하기 전에 레지스트리에 등록되어 있어야 한다. 심지어 네트워상의 다른 컴퓨터에 있어도 상관없다.

COM 이론은 사실 결코 쉽지 않으며 무시못할 정도로 양도 많다. 그래서 이 책에서는 COM 의 기본 이론에 대해서는 다루지 않으며 델파이 개발환경을 이용한 ActiveX 프로그래밍에 대해서만 특화해서 다룬다. 어차피 COM 은 별도로 공부해야 할 과목이며 자료가 많이 공개되어 있으므로 따로 공부해 볼 것을 권한다.

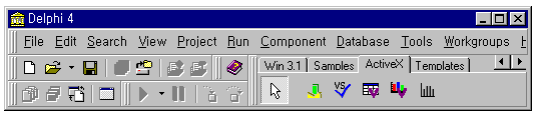
이 책에서는 여러 가지 기술중에서 COM 의 모든 기법이 총동원된 가장 완성된 기술임과 동시에 가장 실용성이 있는 ActiveX 컨트롤에 대해 중점적으로 알아 볼 것이다. 예전에는 OCX 컨트롤이라고 불렀지만 요즘은 ActiveX 컨트롤이라고 부른다. 이 기술은 COM 을 기반으로 하여 프로그램의 부품에 해당하는 컨트롤을 만들고 사용하는 기술이다.

ActiveX 컨트롤은 메소드와 속성, 이벤트를 제공하여 컨테이너에서 컨트롤의 모든 기능을 자유롭게 쓸 수 있도록 해 준다. 델파이의 컴포넌트와 개념적으로 동일하지만 언어에 독립적이며 이진 포멧으로 제공가능하다는 점에 있어서 훨씬 더 우월한 존재이다. 또한 ActiveX 컨트롤은 그 자체로 웹 페이지에 사용할 수 있어 인터넷에도 응용할 수 있다. 자바 애플릿과 마찬가지로 웹 페이지에서 응용 프로그램을 실행시킬 수 있다.

델파이는 ActiveX 컨트롤을 사용할 수 있는 개발 툴임과 동시에 ActiveX 컨트롤을 만들 수 있는 툴이기도 하다. ActiveX 컨트롤을 제작할 수 있는 개발 툴에는 여러 가지 툴이 있고 또한 각 툴들도 여러 가지 방법으로 ActiveX 컨트롤 개발을 지원하고 있는데 저마다 자신이 ActiveX 컨트롤 개발에는 최적의 툴이라고 자랑하고 있다. 델파이도 예외는 아니어서 ActiveX 컨트롤 제작에 많은 지원을 하고 있으며 비주얼 툴답게 빠른 속도로 ActiveX 컨트롤을 만들어 준다.

# 나. 샘플 컨트롤

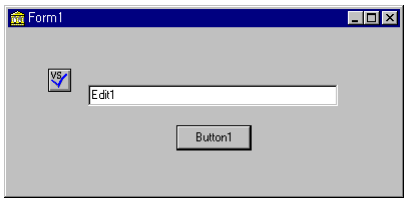
일단 ActvieX 컨트롤을 먼저 사용해 보도록 하자. 델파이는 컴포넌트 팔레트의 ActiveX 페이지에 다음과 같은 컨트롤을 제공한다.




이 컨트롤들은 모두 볼랜드에서 만든 것이 아니며 제 3 자들에 의해 만들어진 것을 볼랜드에서 라이센스한 것들이다. 말하자면 평가판 정도되는 컨트롤인데

15jang
Spell

기능적으로 약간의 제한 사항이 있다. 이 중에 제일 사용하기 쉬운 스펠 체커 컨트롤을 사용해 보자. 새 프로젝트를 시작하고 에디트, 버튼, 그리고 ActiveX 페이지에서 TVSSpell 컴포넌트를 배치한다.



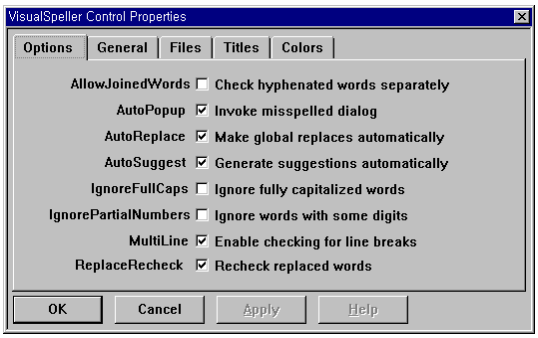

에디트에 단어를 입력하고 버튼을 누르면 사전에 있는 단어인지를 검사하도록 할 것이다. 버튼의 OnClick 이벤트 핸들러에 다음 코드를 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  VSSpell1.CheckText:=Edit1.Text;
  if VSSpell1.ResultCode=0 then
    ShowMessage('맞는 단어입니다')
  else
    ShowMessage('단어가 틀렸습니다');
end;
```

실행해 보면 버튼은 에디트에 입력된 단어가 사전에 있으면 있다는 메시지를 보여주고 없으면 다음과 같이 추천 단어를 보여준다.


이 화면은 워드나 기타 제품들에서 익숙하게 본 적이 있을 것이다. 맞춤법 검사에 대한 여러 가지 옵션은 디자인중에 TVSSpell 컴포넌트를 더블클릭하여 설정한다.

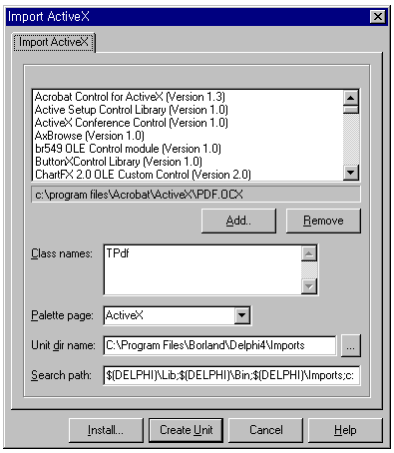
그림
맞춤법 검사 옵션 대화상자




# 다. ActiveX 컨트롤 설치

ActiveX 컨트롤을 델파이 개발환경에 설치하면 마치 컴포넌트를 사용하듯이 ActiveX 컨트롤을 사용할 수 있다. 인터넷이나 대중 통신망에서 쓸만한 ActiveX 컨트롤들을 많이 구할 수 있으며 판매하는 컨트롤이라면 구입할 수도 있다. 이런 컨트롤을 구했을 때 델파이에 설치하는 방법에 대해 알아보자. 여기서 설치할 컨트롤은 간단한 체크 박스 기능을 하는 ActChk 컨트롤이며 김상형이라는 사람이 비주얼 C++ 6.0 으로 만들어 무료로 배포하는 것이다. 다음 방법대로 이 컨트롤을 델파이에 설치해 보자.

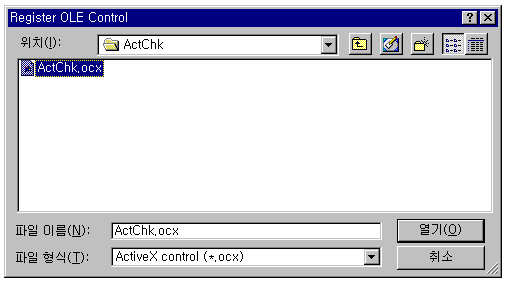
먼저 Component/Import ActiveX Control 메뉴를 선택한다. 그러면 다음 대화상자가 나타나는데 이 대화상자는 시스템 레지스트리의 내용을 몽땅 조사한 후 보여주므로 나타나는데 시간이 좀 걸린다.



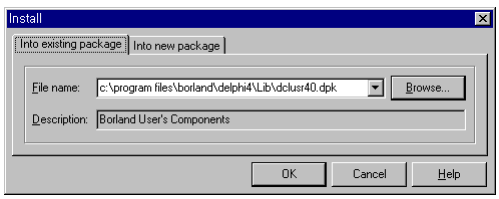

위쪽의 리스트 박스에는 현재 시스템에 등록되어 있는 ActiveX 컨트롤의 목

록이 나타나 있다. 이 목록의 내용은 시스템에 어떤 소프트웨어가 등록되어 있는가에 따라 달라진다. 이 목록에 원하는 컨트롤이 있다면 선택한 후 아래쪽의 Install 버튼을 눌러주면 된다. 새로 컨트롤을 구한 경우라면 이 컨트롤이 목록에 나타나지 않을 것이다. 왜냐하면 ActiveX 컨트롤은 시스템 레지스트리에 등록해야 사용할 수 있기 때문이다. 그래서 ActiveX 컨트롤을 델파이에 설치하기 전에 먼저 시스템 레지스트리에 등록해 주어야 하며 이때는 RegSvr32 등의 유틸리티를 사용하면 된다. 또는 위 대화상자에서 바로 레지스트리에 등록할 수 있다.

지금 설치하고자 하는 ActChk 컨트롤을 레지스트리에 먼저 등록하기 위해 Add 버튼을 누른다. 그러면 이 컨트롤이 있는 경로를 물어오는 파일 열기 대화상자를 보여준다. ActChk 컨트롤은 배포 CD 의 Etc₩ActChk 디렉토리에 있는데 CD-ROM 에서 바로 선택해 주어서는 안되며 이 파일을 윈도우즈의 시스템 디렉토리에 복사해 준 후 설치하는 것이 좋다. 아뭏든 어디에 복사하든 하드 디스크에 복사한 후 ActChk.ocx 를 선택해 준다.

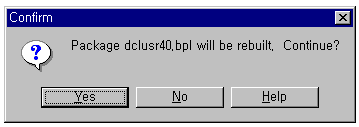



잠시 후면 이 컨트롤이 레지스트리에 등록되며 델파이의 Import ActiveX 대화상자 목록에도 이 컨트롤이 나타날 것이다. 이제 델파이 환경에 이 컨트롤을 등록하기 위해 아래쪽에 있는 Install 버튼을 누르면 다음 대화상자가 나타난다.

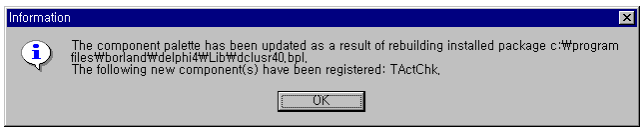



두 개의 페이지로 구성되어 있는데 첫 번째 페이지는 기존의 패키지에 컨트롤을 등록하는 것이고 두 번째 페이지는 새로운 패키지를 만들어 등록하는 것이다. 패키지를 새로 만드는 것은 너무 번거로우니까 이때까지 실습용으로 사용해 오던 dclusr40 패키지에 이 컨트롤을 등록하도록 하자. OK 버튼을 눌러주기만 하

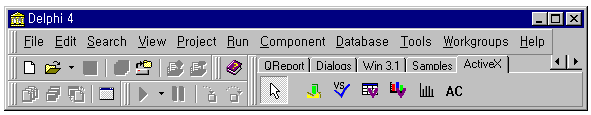
면 된다. 그러면 이 패키지를 컴파일하겠다는 메시지를 보여준다.




Yes 라고 대답해 주면 ActiveX 컨트롤이 포함된 패키지가 다시 컴파일되며 잠시 후에 컴파일이 완료되었다는 메시지가 나타난다.




ActiveX 컨트롤 등록하기가 끝났다. 과연 제대로 등록이 되었는지 확인해 보자. 컴포넌트 팔레트의 ActiveX 페이지를 보면 AC 라고 되어 있는 컴포넌트가 추가되어 있을 것이다.




이 컴포넌트가 바로 지금 우리가 등록한 ActiveX 컨트롤이다. 델파이는 ActiveX 컨트롤을 델파이 환경에 통합하기 위해 이 컨트롤을 포장하는 소스 파일을 만들어 내며 이 파일들은 델파이 디렉토리의 Imports 디렉토리에 위치한다.

```
unit ACTCHKLib_TLB;

// ************************************************************************ //
// WARNING                                                                  //
// -------                                                                  //
// The types declared in this file were generated from data read from a     //
// Type Library. If this type library is explicitly or indirectly (via      //
// another type library referring to this type library) re-imported, or the //
// 'Refresh' command of the Type Library Editor activated while editing the //
// Type Library, the contents of this file will be regenerated and all      //
// manual modifications will be lost.                                       //
// ************************************************************************ //

// PASTLWTR : $Revision:   1.11.1.62  $
```

```
// File generated on 98-08-08 오전 4:15:28 from Type Library described below.

// ************************************************************************ //
// Type Lib: D:₩델파이 CD₩ETC₩ACTCHK₩ACTCHK.OCX
// IID₩LCID: {6E5DA804-F51E-11D0-BCBB-444553540000}₩0
// Helpfile: D:₩델파이 CD₩ETC₩ACTCHK₩ActChk.hlp
// HelpString: ActChk ActiveX Control module
// Version:    1.0
// ************************************************************************ //

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

// *********************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:     //
//   Type Libraries     : LIBID_xxxx                                    //
//   CoClasses          : CLASS_xxxx                                    //
//   DISPInterfaces     : DIID_xxxx                                     //
//   Non-DISP interfaces: IID_xxxx                                      //
// *********************************************************************//
const
  LIBID_ACTCHKLib: TGUID = '{6E5DA804-F51E-11D0-BCBB-444553540000}';
  DIID__DActChk: TGUID = '{6E5DA805-F51E-11D0-BCBB-444553540000}';
  DIID__DActChkEvents: TGUID = '{6E5DA806-F51E-11D0-BCBB-444553540000}';
  CLASS_ActChk: TGUID = '{6E5DA807-F51E-11D0-BCBB-444553540000}';
type

// *********************************************************************//
// Forward declaration of interfaces defined in Type Library           //
// *********************************************************************//
  _DActChk = dispinterface;
  _DActChkEvents = dispinterface;

// *********************************************************************//
// Declaration of CoClasses defined in Type Library                    //
// (NOTE: Here we map each CoClass to its Default Interface)           //
// *********************************************************************//
  ActChk = _DActChk;


// *********************************************************************//
// DispIntf:  _DActChk
// Flags:     (4112) Hidden Dispatchable
// GUID:      {6E5DA805-F51E-11D0-BCBB-444553540000}
// *********************************************************************//
```

```
  _DActChk = dispinterface
    ['{6E5DA805-F51E-11D0-BCBB-444553540000}']
    property Caption: WideString dispid -518;
    property BackColor: OLE_COLOR dispid -501;
    property ChkState: Smallint dispid 1;
    property EllipseRatio: Smallint dispid 2;
    property ForeColor: OLE_COLOR dispid -513;
    function ReverseSelection: Smallint; dispid 3;
    procedure AboutBox; dispid -552;
  end;

// ******************************************************************//
// DispIntf:  _DActChkEvents
// Flags:     (4096) Dispatchable
// GUID:      {6E5DA806-F51E-11D0-BCBB-444553540000}
// ******************************************************************//
  _DActChkEvents = dispinterface
    ['{6E5DA806-F51E-11D0-BCBB-444553540000}']
    procedure ChangeState(state: Smallint); dispid 1;
  end;


// ******************************************************************//
// OLE Control Proxy class declaration
// Control Name     : TActChk
// Help String      : ActChk Control
// Default Interface: _DActChk
// Def. Intf. DISP? : Yes
// Event   Interface: _DActChkEvents
// TypeFlags        : (34) CanCreate Control
// ******************************************************************//
  TActChkChangeState = procedure(Sender: TObject; state: Smallint) of object;

  TActChk = class(TOleControl)
  private
    FOnChangeState: TActChkChangeState;
    FIntf: _DActChk;
    function  GetControlInterface: _DActChk;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    function ReverseSelection: Smallint;
    procedure AboutBox;
    property  ControlInterface: _DActChk read GetControlInterface;
  published
```

```
    property  ParentColor;
    property  ParentFont;
    property  TabStop;
    property  Align;
    property  DragCursor;
    property  DragMode;
    property  ParentShowHint;
    property  PopupMenu;
    property  ShowHint;
    property  TabOrder;
    property  Visible;
    property  OnDragDrop;
    property  OnDragOver;
    property  OnEndDrag;
    property  OnEnter;
    property  OnExit;
    property  OnStartDrag;
    property Caption: WideString index -518 read GetWideStringProp write SetWideStringProp stored False;
    property BackColor: TColor index -501 read GetTColorProp write SetTColorProp stored False;
    property ChkState: Smallint index 1 read GetSmallintProp write SetSmallintProp stored False;
    property EllipseRatio: Smallint index 2 read GetSmallintProp write SetSmallintProp stored False;
    property ForeColor: TColor index -513 read GetTColorProp write SetTColorProp stored False;
    property OnChangeState: TActChkChangeState read FOnChangeState write FOnChangeState;
  end;

procedure Register;

implementation

uses ComObj;

procedure TActChk.InitControlData;
const
  CEventDispIDs: array [0..0] of DWORD = (
    $00000001);
  CControlData: TControlData = (
    ClassID: '{6E5DA807-F51E-11D0-BCBB-444553540000}';
    EventIID: '{6E5DA806-F51E-11D0-BCBB-444553540000}';
    EventCount: 1;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: nil;
    Flags: $00000013;
    Version: 300);
begin
  ControlData := @CControlData;
end;
```

```
procedure TActChk.CreateControl;

  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as _DActChk;
  end;

begin
  if FIntf = nil then DoCreate;
end;

function TActChk.GetControlInterface: _DActChk;
begin
  CreateControl;
  Result := FIntf;
end;

function TActChk.ReverseSelection: Smallint;
begin
  Result := ControlInterface.ReverseSelection;
end;

procedure TActChk.AboutBox;
begin
  ControlInterface.AboutBox;
end;

procedure Register;
begin
  RegisterComponents('ActiveX',[TActChk]);
end;

end.
```

이 파일은 ActiveX 컨트롤을 감싸는 래퍼 클래스(wrapper class)를 정의하고 있다. 델파이에서 ActiveX 컨트롤을 사용하려면 COM 명세에 맞게 속성, 메소드를 호출해 주어야 하는데 가능한 일이기는 하지만 델파이의 문법과는 다소 이질적이므로 직접 사용하기가 무척 어렵다. 그래서 델파이는 ActiveX 컨트롤이 가지고 있는 속성, 메소드 등을 델파이 개발환경에서 편리하게 사용할 수 있도록 파스칼 소스 형태로 만들어주는 것이다. 이 소스는 델파이 패키지에 통합될 수 있으며 이 소스를 포함한 패키지를 컴파일함으로써 ActiveX 가 컴포넌트

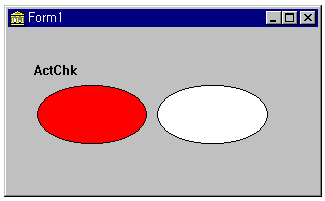
가 되는 것이다.

소스 중간을 보면 TActChk 컨트롤이 TOleControl 클래스로부터 파생되고 있는데 TOleControl 은 ActiveX 컨트롤의 기본적인 기능을 모두 가지고 있는 클래스이다. 또한 이 클래스는 ActiveX 컨트롤의 멤버를 델파이에서 액세스할 수 있도록 인터페이스해 준다. 이 클래스부터 파생함으로써 ActiveX 컨트롤이 델파이 개발환경에 통합되는 것이다. TActChk 컨트롤은 ActChk 컨트롤의 모든 특성을 가지고 있는 ActiveX 컨트롤의 파스칼판 복사본이라고 할 수 있다.

# 라. ActChk 컨트롤

15jang
TestActChk

방금전에 설치한 ActChk 컨트롤이 뭐하는 컨트롤인지 또 과연 델파이 개발환경에 잘 통합되었는지 간단한 예제를 만들어 보자. 새 프로젝트를 시작하고 폼에 이 컨트롤을 배치해 보자. 크기를 좀 늘려준 후 Caption 속성을 'ActChk' 로 바꾸고 ChkState 속성을 1 로 바꾸어준다.



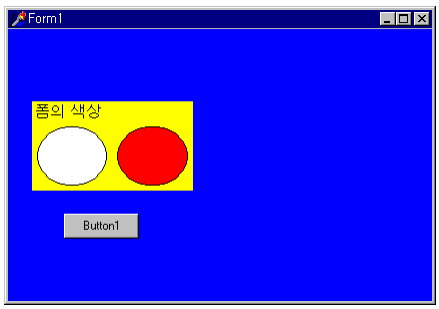

ChkState 속성을 0,2 로 각각 바꾸어 보면 이 값에 따라 빨간색 원이 바뀌며 이 속성이 현재 선택 상태를 나타낸다는 것을 알 수 있을 것이다. Caption 속성은 위쪽에 나타나는 문자열이며 ForeColor 는 캡션의 색상을, BackColor 는 배경 색상을 지정한다. EllipseRatio 속성을 변경해 보면 타원의 크기가 변경된다. 속성 변경이 제대로 되고 있다.

이 상태에서 실행을 해 보고 마우스로 타원을 클릭하면 선택 상태가 변경된다. 마우스 동작에도 제대로 반응하고 있다. 마우스로 클릭하면 OnClick 이벤트가 발생하며 선택 상태가 변경될 때 OnChangeState 이벤트가 발생하도록 되어 있다. 디자인중에 테스트해 볼 수는 없지만 선택 상태를 반전시켜주는 ReverseSelection 이라는 메소드도 가지고 있다.

이 컨트롤은 두 가지 선택 상태중에서 한 가지를 선택할 수 있는 일종의 체크박스와 같은 컨트롤이며 필자가 비주얼 C++을 이용한 ActiveX 컨트롤 제작 예제로 작성한 것이다. 예제이다 보니 사실 실용성은 없지만 동작은 제대로 한다.

이 컨트롤을 사용해 폼의 색깔을 변경하는 예제를 만들어 보았다.

그림
ActChk 컨트롤 사용 예




ActChk 컨트롤의 상태가 변경될 때마다 폼의 색상을 변경하도록 하였으며 Button1 을 누르면 체크 상태를 반전하도록 하였다.

```
procedure TForm1.ActChk1ChangeState(Sender: TObject; state: Smallint);
begin
  case ActChk1.ChkState of
    1: Form1.Color:=clRed;
    2: Form1.Color:=clBlue;
  else Form1.Color:=clSilver;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ActChk1.ReverseSelection;
end;
```

이 컨트롤에 대해서는 더 이상 자세히 알 필요도 없겠지만 어쨌든 이런 식으로 ActiveX 컨트롤을 델파이 개발환경으로 가져와 쓸 수 있다는 것만 알아 두도록 하자. 단 이렇게 만들어진 예제는 배포할 때 OCX 파일을 같이 배포해 주어야 한다. OCX 파일은 형태가 좀 특별한 DLL 일 뿐이므로 메모리를 공유하고 하드디스크를 절약하고 업그레이드를 쉽게 할 수 있다는 DLL 의 모든 장점을 그대로 가진다. 또한 버전에 따른 충돌, 파일의 분리 등과 같은 DLL 의 단점도 그대로 가지며 여기에 반드시 사용전에 등록까지 해 주어야 한다는 부담도 있다.

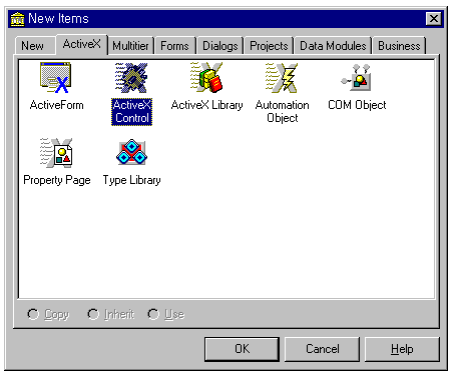
# 16-7 ActiveX 컨트롤 제작

## 가. 캘린더 컨트롤

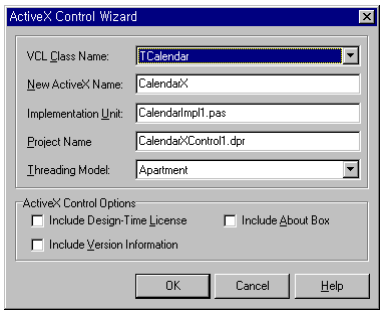
15jang CalendarX

델파이로는 ActiveX 컨트롤을 사용하는 것 뿐만 아니라 만드는 것도 가능하다. 먼저 델파이로 VCL 컨트롤을 만든 후 이 컨트롤을 ActiveX 컨트롤로 변환하기만 하면 된다. 변환의 대상이 되는 컨트롤은 윈도우 핸들을 가지고 있는 TWinControl 파생 클래스여야만 한다. 그래서 버튼, 에디트 등 대부분의 델파이 컨트롤이 ActiveX 컨트롤로 변환될 수 있지만 레이블, 셰이프 등은 TWinControl 로부터 파생되지 않았기 때문에 ActiveX 컨트롤로 변환할 수 없다. 앞에서 우리가 만든 MyGauge 컴포넌트도 TGraphicControl 로부터 파생되었기 때문에 곧바로는 ActiveX 컨트롤이 될 수 없으며 소스를 약간 수정해 주어야 한다.

VCL 컴포넌트를 ActiveX 컨트롤로 만드는 실습을 해 보자. 델파이의 Calendar 컴포넌트를 ActiveX 컨트롤로 변환해 볼 것이다. File/New 메뉴 항목을 선택한 후 ActiveX 페이지에서 ActiveX Control 을 선택한다.

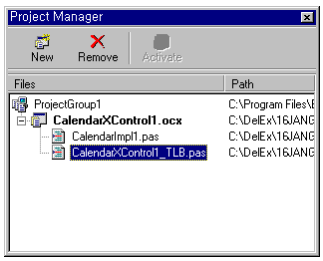



그러면 다음 대화상자가 나타날 것이다.




이 대화상자는 ActiveX 컨트롤을 만들기 위한 여러 가지 옵션을 입력받는데 이 중 제일 중요한 입력 사항은 제일 위의 콤보 박스에서 선택하는 VCL 클래스이며 이 클래스가 ActiveX 컨트롤의 재료가 된다. 나머지 정보들은 만들어질 ActiveX 컨트롤의 이름, 소스 파일명 및 몇 가지 옵션들인데 대개의 경우 디폴트를 받아들이면 된다. 아래쪽에는 라이센스 생성, AboutBox 생성, 버전 정보 생성을 선택하는 체크 박스가 있다.

VCL 클래스 콤보 박스를 열어보면 현재 등록된 컴포넌트중에 ActiveX 로 변환할 수 있는 컴포넌트의 목록이 나타나는데 이 중 TCalendar 컴포넌트를 선택하자. 그리고 OK 버튼을 누르면 이 컴포넌트를 ActiveX 로 변환하기 위한 프로젝트를 만들어 준다. 프로젝트가 어떻게 구성되어 있는지 보자.




프로젝트 파일, 유닛 파일, 그리고 타입 라이브러리로 이루어져 있다. 먼저 프로젝트 파일을 보자.

```
library CalendarXControl1;

uses
  ComServ,
  CalendarXControl1_TLB in 'CalendarXControl1_TLB.pas',
  CalendarImpl1 in 'CalendarImpl1.pas' {CalendarX: CoClass};
```

```
{$E ocx}

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.TLB}

{$R *.RES}

begin
end.
```

마치 DLL 의 프로젝트와 유사한 모양을 가지고 있는데 실제로 OCX 파일은 DLL 이므로 당연하다고 하겠다. 실행 파일을 만드는 프로젝트가 아니므로 begin 과 end 사이에는 아무 코드도 없다.

다음은 이 컴포넌트의 실행 코드를 담고 있는 유닛 파일이다. 소스가 너무 길기 때문에 중간 대부분을 생략하고 중요 부분만 보였다.

```
unit CalendarImpl1;

interface

uses
  Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms, StdCtrls,
  ComServ, StdVCL, AXCtrls, CalendarXControl1_TLB, Calendar, Grids;

type
  TCalendarX = class(TActiveXControl, ICalendarX)
  private
    { Private declarations }
    FDelphiControl: TCalendar;
    FEvents: ICalendarXEvents;
    procedure ChangeEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
  protected
    { Protected declarations }
    procedure DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage); override;
```

```
procedure EventSinkChanged(const EventSink: IUnknown); override;
procedure InitializeControl; override;
function ClassNameIs(const Name: WideString): WordBool; safecall;
function DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
function DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
function Get_BiDiMode: TxBiDiMode; safecall;
function Get_BorderStyle: TxBorderStyle; safecall;
function Get_CalendarDate: Double; safecall;
function Get_Color: OLE_COLOR; safecall;
function Get_Ctl3D: WordBool; safecall;
function Get_Cursor: Smallint; safecall;
function Get_Day: Integer; safecall;
function Get_DoubleBuffered: WordBool; safecall;
function Get_DragCursor: Smallint; safecall;
function Get_DragMode: TxDragMode; safecall;
function Get_Enabled: WordBool; safecall;
function Get_Font: IFontDisp; safecall;
function Get_GridLineWidth: Integer; safecall;
function Get_Month: Integer; safecall;
function Get_ParentColor: WordBool; safecall;
function Get_ParentFont: WordBool; safecall;
function Get_ReadOnly: WordBool; safecall;
function Get_StartOfWeek: Smallint; safecall;
function Get_UseCurrentDate: WordBool; safecall;
function Get_Visible: WordBool; safecall;
function Get_Year: Integer; safecall;
function GetControlsAlignment: TxAlignment; safecall;
function IsRightToLeft: WordBool; safecall;
function UseRightToLeftAlignment: WordBool; safecall;
function UseRightToLeftReading: WordBool; safecall;
function UseRightToLeftScrollBar: WordBool; safecall;
procedure _Set_Font(const Value: IFontDisp); safecall;
procedure FlipChildren(AllLevels: WordBool); safecall;
procedure InitiateAction; safecall;
procedure NextMonth; safecall;
procedure NextYear; safecall;
procedure PrevMonth; safecall;
procedure PrevYear; safecall;
procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
procedure Set_BorderStyle(Value: TxBorderStyle); safecall;
procedure Set_CalendarDate(Value: Double); safecall;
procedure Set_Color(Value: OLE_COLOR); safecall;
procedure Set_Ctl3D(Value: WordBool); safecall;
procedure Set_Cursor(Value: Smallint); safecall;
procedure Set_Day(Value: Integer); safecall;
procedure Set_DoubleBuffered(Value: WordBool); safecall;
```

```
    procedure Set_DragCursor(Value: Smallint); safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    procedure Set_GridLineWidth(Value: Integer); safecall;
    procedure Set_Month(Value: Integer); safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    procedure Set_ParentFont(Value: WordBool); safecall;
    procedure Set_ReadOnly(Value: WordBool); safecall;
    procedure Set_StartOfWeek(Value: Smallint); safecall;
    procedure Set_UseCurrentDate(Value: WordBool); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    procedure Set_Year(Value: Integer); safecall;
    procedure UpdateCalendar; safecall;
  end;

implementation

uses ComObj;

{ TCalendarX }

procedure TCalendarX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  { Define property pages here.  Property pages are defined by calling
    DefinePropertyPage with the class id of the page.  For example,
      DefinePropertyPage(Class_CalendarXPage); }
end;

procedure TCalendarX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as ICalendarXEvents;
end;

procedure TCalendarX.InitializeControl;
begin
  FDelphiControl := Control as TCalendar;
  FDelphiControl.OnChange := ChangeEvent;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnDblClick := DblClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;

function TCalendarX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := FDelphiControl.ClassNameIs(Name);
```

```
end;

function TCalendarX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := FDelphiControl.DrawTextBiDiModeFlags(Flags);
end;

function TCalendarX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := FDelphiControl.DrawTextBiDiModeFlagsReadingOnly;
end;

========== 중간 생략 ==========
function TCalendarX.Get_Day: Integer;
begin
  Result := FDelphiControl.Day;
end;

function TCalendarX.Get_Visible: WordBool;
begin
  Result := FDelphiControl.Visible;
end;

function TCalendarX.Get_Year: Integer;
begin
  Result := FDelphiControl.Year;
end;

procedure TCalendarX.Set_Day(Value: Integer);
begin
  FDelphiControl.Day := Value;
end;

procedure TCalendarX.Set_Visible(Value: WordBool);
begin
  FDelphiControl.Visible := Value;
end;

procedure TCalendarX.Set_Year(Value: Integer);
begin
  FDelphiControl.Year := Value;
end;

procedure TCalendarX.UpdateCalendar;
begin
  FDelphiControl.UpdateCalendar;
```

```
end;

procedure TCalendarX.ChangeEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnChange;
end;

procedure TCalendarX.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;

procedure TCalendarX.DblClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDblClick;
end;

procedure TCalendarX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;

initialization
  TActiveXControlFactory.Create(
    ComServer,
    TCalendarX,
    TCalendar,
    Class_CalendarX,
    1,
    '',
    0,
    tmApartment);
end.
```

TCalendarX 클래스를 TActiveXControl 클래스로부터 상속받아 ICalendarX 인터페이스의 기능을 구현하고 있다. 코드의 대부분은 속성을 읽고 쓰는 메소드들이다. ICalendarX 인터페이스는 타입 라이브러리에 정의되어 있다. 역시 소스가 무척이나 길기 때문에 대부분 생략하고 중요한 부분만 보인다. 대충 모양만 보도록 하고 정확한 리스트는 배포 CD 를 참조하거나 아니면 직접

만들어서 보기 바란다.

```
unit CalendarXControl1_TLB;


interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
  LIBID_CalendarXControl1: TGUID = '{649AF6CD-2E7E-11D2-80C6-00A024463382}';
  IID_ICalendarX: TGUID = '{649AF6CE-2E7E-11D2-80C6-00A024463382}';
  DIID_ICalendarXEvents: TGUID = '{649AF6D0-2E7E-11D2-80C6-00A024463382}';
  CLASS_CalendarX: TGUID = '{649AF6D2-2E7E-11D2-80C6-00A024463382}';

// TxBorderStyle constants
type
  TxBorderStyle = TOleEnum;
const
  bsNone = $00000000;
  bsSingle = $00000001;

// TxDragMode constants
type
  TxDragMode = TOleEnum;
const
  dmManual = $00000000;
  dmAutomatic = $00000001;

// TxAlignment constants
type
  TxAlignment = TOleEnum;
const
  taLeftJustify = $00000000;
  taRightJustify = $00000001;
  taCenter = $00000002;

// TxBiDiMode constants
type
  TxBiDiMode = TOleEnum;
const
  bdLeftToRight = $00000000;
  bdRightToLeft = $00000001;
  bdRightToLeftNoAlign = $00000002;
  bdRightToLeftReadingOnly = $00000003;
```

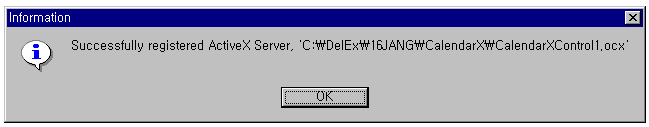
```
type
ICalendarX = interface;
  ICalendarXDisp = dispinterface;
  ICalendarXEvents = dispinterface;

CalendarX = ICalendarX;


// *********************************************************************//
// Interface: ICalendarX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {649AF6CE-2E7E-11D2-80C6-00A024463382}
// *********************************************************************//
  ICalendarX = interface(IDispatch)
    ['{649AF6CE-2E7E-11D2-80C6-00A024463382}']
    function Get_CalendarDate: Double; safecall;
    procedure Set_CalendarDate(Value: Double); safecall;
    procedure NextMonth; safecall;
    procedure NextYear; safecall;
    procedure PrevMonth; safecall;
    procedure PrevYear; safecall;
    procedure UpdateCalendar; safecall;
====================== 중간 생략 ==================
end;

// *********************************************************************//
// DispIntf:  ICalendarXDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {649AF6CE-2E7E-11D2-80C6-00A024463382}
// *********************************************************************//
  ICalendarXDisp = dispinterface
    ['{649AF6CE-2E7E-11D2-80C6-00A024463382}']
    property CalendarDate: Double dispid 1;
    procedure NextMonth; dispid 2;
    procedure NextYear; dispid 3;
    procedure PrevMonth; dispid 4;
    procedure PrevYear; dispid 5;
    procedure UpdateCalendar; dispid 6;
    property BorderStyle: TxBorderStyle dispid 7;
    property Color: OLE_COLOR dispid -501;
====================== 중간 생략 ==================
end;

ICalendarXEvents = dispinterface
    ['{649AF6D0-2E7E-11D2-80C6-00A024463382}']
```

```
    procedure OnClick; dispid 1;
    procedure OnChange; dispid 2;
    procedure OnDblClick; dispid 3;
    procedure OnKeyPress(var Key: Smallint); dispid 9;
  end;


TCalendarXOnKeyPress = procedure(Sender: TObject; var Key: Smallint) of object;

  TCalendarX = class(TOleControl)
  private
    FOnClick: TNotifyEvent;
    FOnChange: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnKeyPress: TCalendarXOnKeyPress;
    FIntf: ICalendarX;
    function  GetControlInterface: ICalendarX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    procedure NextMonth;
    procedure NextYear;
    procedure PrevMonth;
    procedure PrevYear;
    procedure UpdateCalendar;
==================== 중간 생략 ==================
published
    property  TabStop;
    property  Align;
    property  ParentShowHint;
    property  PopupMenu;
    property  ShowHint;
    property  TabOrder;
    property  OnDragDrop;
    property  OnDragOver;
    property  OnEndDrag;
    property  OnEnter;
    property  OnExit;
    property  OnStartDrag;
==================== 중간 생략 ==================

procedure Register;

implementation
```

```
uses ComObj;
====================== 이하 생략 ==================
```

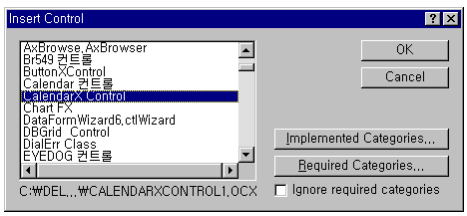
소스를 한번 죽 훑어 보기는 하되 COM 명세에 대해 잘 모르는 사람들은 이 소스 내용을 완벽히 파악하기 힘들 것이다.

이 프로젝트를 적당한 디렉토리에 저장하고 컴파일하면 ActiveX 컨트롤이 만들어진다. 단 실행 파일을 만드는 것이 아니라 단독 실행이 불가능한 컨트롤을 만드는 것이므로 F9 를 누르지 말고 Project/Buid 항목을 선택하여 컴파일만 하도록 한다. 그리고 탐색기로 확인해 보면 CalendarXControl.ocx 파일이 생성되어 있는데 이 파일이 결과적으로 만들어진 ActiveX 컨트롤이다. 컨트롤을 만들었으면 이 컨트롤을 시스템 레지스트리에 먼저 등록해야 한다. Run/Register ActiveX Server 항목을 선택하면 모든 등록 처리를 해 주며 다음 메시지를 보여준다.




무사히 등록되었다는 뜻이다. 그럼 이제 이렇게 만든 ActiveX 컨트롤을 테스트해 보자. 델파이 컴포넌트를 ActiveX 컨트롤로 만든 것이므로 델파이에서 테스트하는 것은 의미가 없다. ActiveX 컨트롤을 사용할 수 있는 어떤 툴이라도 테스트해 볼 수 있는데 여기서는 비주얼 C++ 6.0 과 함께 제공되는 ActiveX 컨트롤 테스트 컨테이너 프로그램을 사용해 보자. 이 프로그램이 없다면 비주얼 베이직이나 액세스 등으로도 테스트 가능하다.

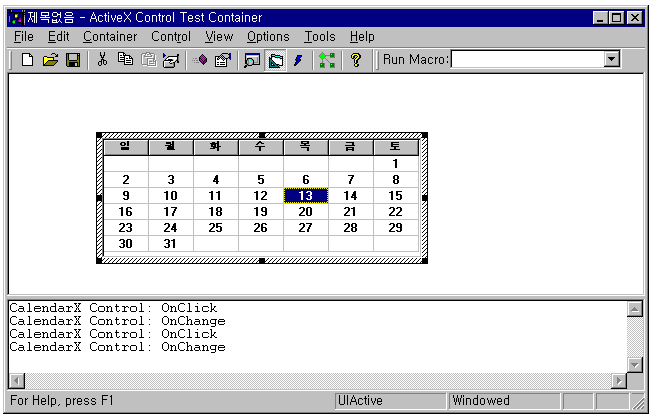
테스트 컨테이너에서 컨트롤을 불러오기 위해 Edit/Insert New Control 명령을 선택한다. 그러면 잠시 후에 시스템 레지스트리에 등록된 모든 컨트롤의 목록이 나타날 것이다.



이 목록에서 CalendarX 컨트롤을 선택하면 델파이의 Calendar 컴포넌트가 테스트 컨테이너에 나타난다.

**그림**

델파이 컴포넌트를 ActiveX 컨트롤로 만든 예



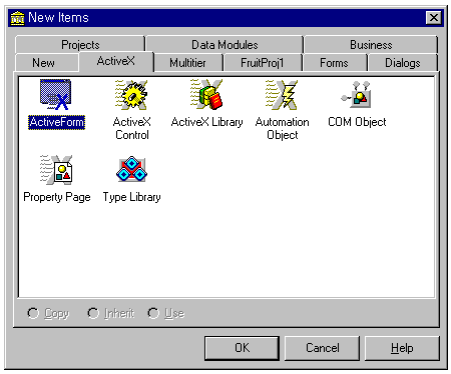

사용하는 방법이나 속성은 델파이에서의 사용 방법과 거의 동일하다. 테스트 컨테이너 내에서 속성을 변경할 수 있으며 메소드도 호출할 수 있고 어떤 이벤트가 발생하는지 살펴볼 수도 있다.

# 나. 액티브 폼

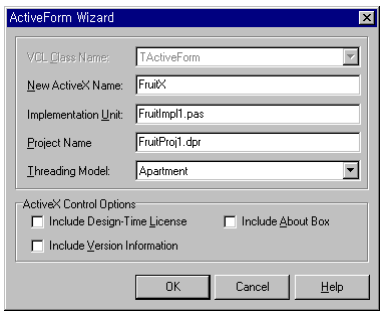
15jang
FruitX

델파이의 컨트롤들은 윈도우에 기반한 것이면 모두 ActiveX 컨트롤로 만들 수 있다. 그 중에 특히 재미있는 것은 폼도 윈도우 핸들을 가지고 있는 컴포넌트이므로 ActiveX 컨트롤이 될 수 있다는 것이다. 폼에는 여러 가지 컴포넌트들이 배치될 수 있으며 또한 그러한 컴포넌트들을 관리하는 코드까지 포함되므로 폼을 ActiveX 로 만들면 프로그램을 통째로 ActiveX 컨트롤로 만들 수 있다. 이렇게 폼을 ActiveX 컨틀로 만들 때 이를 액티브 폼(ActiveForm)이라고 한다.

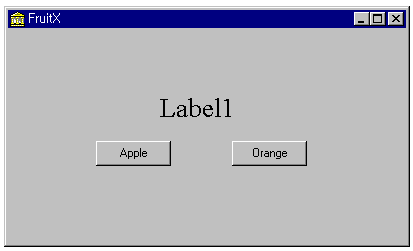
굉장히 복잡할 것 같지만 막상 만들어 보면 컨트롤을 만드는 것보다 더 쉽다. 단계를 따라 아주 간단한 액티브 폼을 만들어 보자. File/New 를 선택하고 ActiveX 페이지에서 ActiveForm 을 선택한다.




다음 대화상자는 만들고자 하는 액티브 폼의 입력 정보를 요구한다.



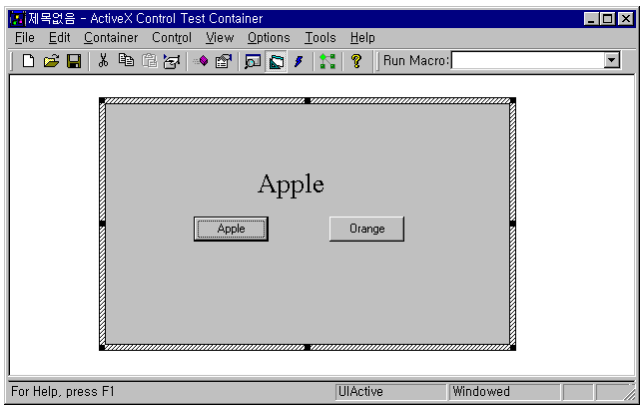

VCL 클래스는 이미 TActiveForm 으로 고정되어 있으며 그 아래쪽의 에디트 박스에 컨트롤의 이름, 소스를 저장할 유닛과 프로젝트명을 입력해 준다. 두 번째 에디트에 FruitX 라고 입력하고 OK 버튼을 누른다. 일반 프로젝트를 만들 때와 마찬가지로 빈 폼이 주어질 것이며 액티브 폼을 만드는데 필요한 소스 코드가 생성된다. 폼을 프로그래밍하는 방법은 일반 프로젝트의 경우와 완전히 동일하다. 이때까지 배운 내용대로 아무 폼이나 만들어도 된다. 2 장에서 처음 만들었던 Fruit 예제와 똑같이 컴포넌트를 배치해 보자.




그리고 두 버튼의 OnClick 이벤트 핸들러를 작성하여 버튼을 누르면 레이블의 캡션이 바뀌도록 해 준다.

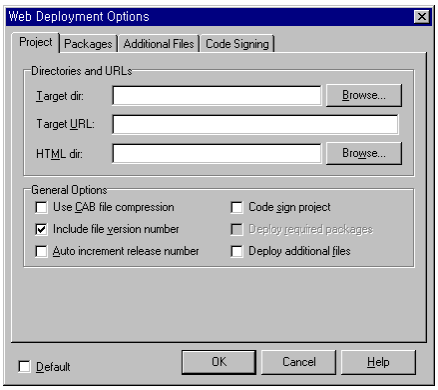
```
procedure TFruitX.BtnAppleClick(Sender: TObject);
begin
  Label1.Caption:='Apple';
end;

procedure TFruitX.BtnOrangeClick(Sender: TObject);
begin
  Label1.Caption:='Orange';
end;
```

예제가 다 만들어졌다. 일단 이 프로젝트를 일정한 디렉토리에 저장한 후 Project/Build 메뉴 항목을 선택하여 프로젝트를 컴파일하면 FruitProj1.ocx 파일이 생성될 것이다. 이 파일안에 방금 작성한 액티브 폼과 코드가 모두 포함되어 있다. Run/Register ActiveX Server 항목을 선택하여 시스템 레지스트리에 이 컨트롤을 등록한 후 테스트 컨테이너로 테스트해 보자.




테스트 컨테이너내에 폼이 열리며 버튼을 누르면 레이블이 바뀐다. 폼 자체가 하나의 컨트롤이 된 것이다. 이제 이 폼은 어떤 개발 툴이나 두루 사용할 수 있다. 게다가 ActiveX 컨트롤의 특성상 이 폼은 인터넷의 웹 사이트에도 그대로 올라갈 수 있다.

액티브 폼을 웹 사이트에 포함시키려면 HTML 파일을 작성해 주어야 한다. 그러나 이런 작업도 직접 할 필요없이 델파이가 대신해 준다. Web Deploy 항목을 선택하면 액티브 폼을 포함하는 HTML 파일을 만들어 주는데 먼저 Web Deploymenr Option으로 몇 가지 설정을 해 주어야 한다.



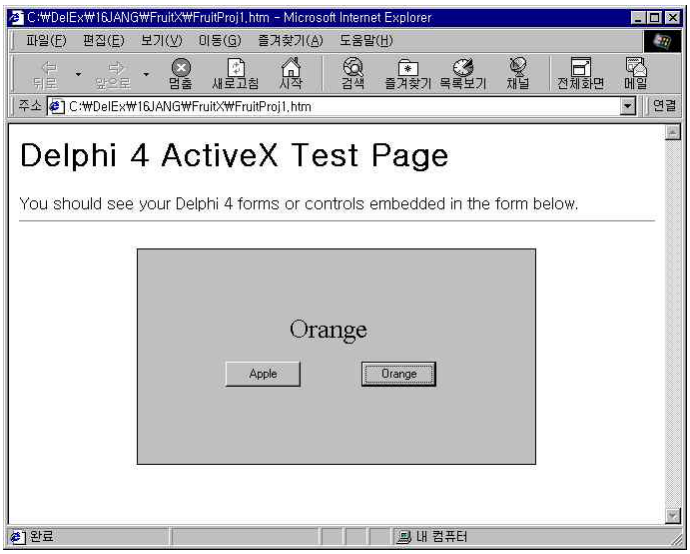

Target dir은 액티브 폼 파일이 복사될 디렉토리이며 Target URL은 OCX 파일의 위치이며 HTML dir은 HTML 파일을 저장할 디렉토리이다. 현재 프로젝트 디렉토리를 설정하려면 모두 점(.)을 적어주면 된다. Web Deploy 명령에 의해 프로젝트 디렉토리에 다음과 같은 HTML 파일이 생성될 것이다.

```
<HTML>
<H1> Delphi 4 ActiveX Test Page </H1><p>
You should see your Delphi 4 forms or controls embedded in the form below.
<HR><center><P>
<OBJECT
        classid="clsid:649AF6DF-2E7E-11D2-80C6-00A024463382"
        codebase="./FruitProj1.ocx"#version=1,0,0,0
        width=400
        height=216
        align=center
        hspace=0
        vspace=0
>
</OBJECT>
</HTML>
```

<OBJECT> 태그안에 우리가 만든 액티브 폼이 포함되어 있다. 이 파일을 더블클릭하면 웹 브라우저가 열리면서 액티브 폼이 웹 브라우저상에 나타난다. 물론 나타날 뿐만 아니라 실제로도 동작한다. 단 이 때 웹 브라우저는 ActiveX 지원 기능이 있어야 한다.

**그림**

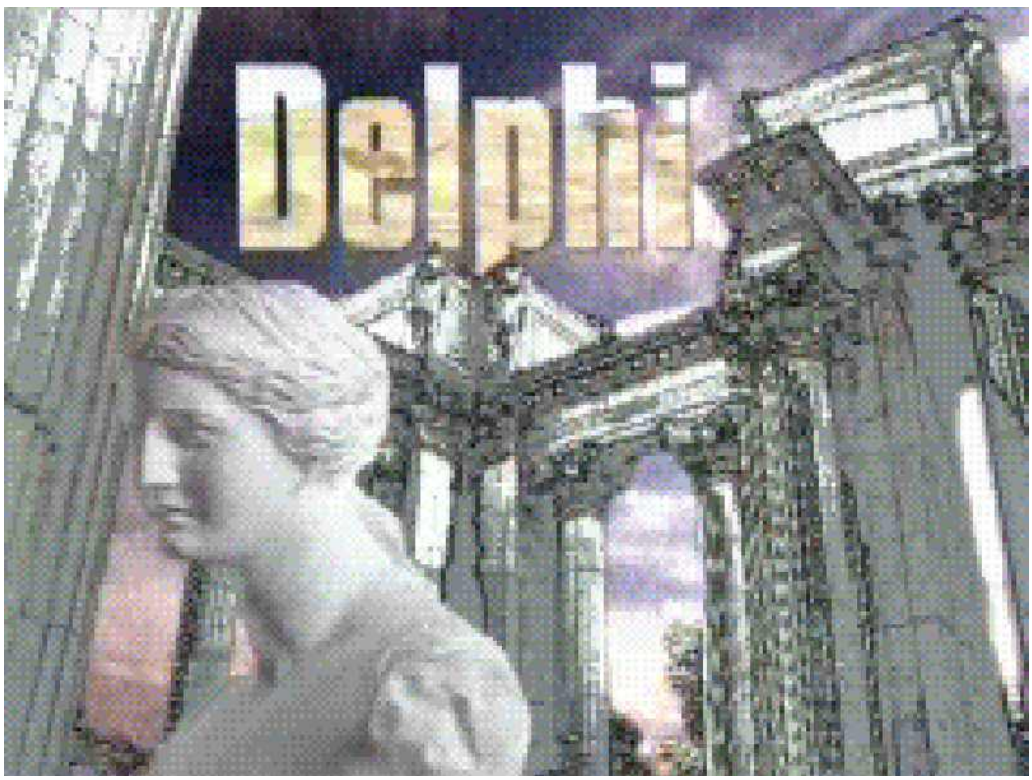
웹 페이지에 나타난 액티브 폼




이런 식으로 액티브 폼을 사용하면 웹 상에서 동작하는 프로그램을 만들 수 있다. 그러나 보안 문제, 파일 크기 문제 등 현실적으로 실용화하기에는 아직 해결해야 할 문제가 많다.

# 델파이 제대로 쓰기

제 17 장



# 17-1 디버거 사용

## 가. 디버깅

프로그램은 프로그래머에 의해 만들어지며 프로그래머는 사람이므로 반드시 실수를 하게 되어 있고 프로그램에는 항상 버그가 있게 마련이다. 물론 버그 없는 프로그램을 만들겠다는 비장한 각오와 성실한 노력이 필요하지만 그렇다고 버그의 존재 자체를 부정할 수는 없다. 버그는 당연히 발생하되 다만 최소화시켜야 한다. 버그의 위치를 찾아내고 버그를 수정하는 작업을 디버깅(Debugging)이라고 하며 프로그램 개발 과정에서 가장 많은 시간과 인내심을 요구한다. 심할 경우 총 개발 기간의 80%가 벌레잡기에 소요되기도 한다.

델파이는 통합환경 디버거를 제공하여 별도의 디버깅 툴을 사용하지 않고 델파이 내부에서 디버깅 작업을 직접 할 수 있다. 통합환경 디버깅이 가능하다는 것은 델파이가 가진 장점 중에서도 아주 엄청난 매력을 지닌 장점이다. 통합환경 디버거가 생산성 향상에 얼마나 큰 기여를 하는가는 디버깅을 하다 보면 절실히 깨닫게 될 것이다.

버그란 사람의 실수에 의해 비롯된다. Integer를 Intergr과 같이 잘못 표기하는 오타가 버그의 가장 주된 원인이기는 하지만 오타는 컴파일 과정에서 즉각적으로 발견되고 보고되기 때문에 디버깅 작업의 대상은 아니다. 디버깅이 요구되는 버그의 종류에는 다음 두 가지가 있다.

### ■ 실행시 에러

문법적으로는 아무 문제가 없으므로 컴파일에는 이상이 없었지만 프로그램 실행중에(Run time) 에러가 발생하는 경우이다. 예를 들어 디스크 상에 없는 파일을 액세스하고자 했다거나 0으로 나누는 동작을 한 경우에 실행시 에러가 발생한다. 실행시 에러는 운영체제가 에러 메시지를 출력하고 어떤 종류의 에러가 발생했는지를 알려주므로 다소 수정하기가 쉬운 편이다. 디버거로 프로그램을 단계 실행해 보아 어느 부분에서 버그가 발생했는지를 찾아 수정해 주면 된다.

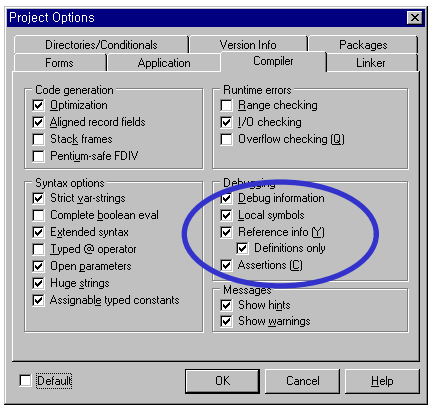
### ■ 논리적 에러

논리적 에러(Logical error)란 문법적으로도 적법하고 무엇인가 합당한 동작

을 하기는 하여 프로그램 실행에는 아무 문제가 없지만 프로그래머가 의도한 대로 결과가 나오지 않는 에러이다. 화면 출력이 제대로 안된다거나 엉뚱한 정보가 화면으로 출력되는 경우, 계산 결과가 틀린 경우 등이 있다. 논리 에러는 대개의 경우 변수에 잘못된 값이 대입되어 발생하는 경우가 가장 많으며 프로그램의 제어구조가 잘못 설계되었거나 자료구조에 결함이 있는 경우 발생한다. 문법에 이상이 없으므로 어디서 에러가 발생했는지 찾아내기가 무척 어렵다. 소스를 처음부터 끝까지 검토해 보아야 하는 것이 원칙이지만 이 경우도 디버거를 사용하면 에러가 발생한 위치와 원인을 쉽게 알아낼 수 있다.

델파이로 디버깅을 하려면 디버깅이 가능한 조건을 만들어 주어야 한다. 디버거는 실행 파일에 삽입한 디버깅 정보를 사용하여 디버깅을 수행하므로 디버깅 정보가 실행 파일에 삽입될 수 있도록 해 주어야 한다. Project Options의 Compiler 페이지에서 Debugging란에 있는 옵션들이 모두 체크되어 있어야 한다. 이 옵션들이 선택되어야 실행 파일을 컴파일할 때 디버깅 정보를 같이 생성한다.

그림
디버거 옵션




이 옵션들이 선택되어 있어야 한다.

이 옵션들은 모두 디폴트로 선택되어 있으므로 델파이를 설치한 후 특별히 옵션을 조정하지 않았다면 통합환경 디버거를 이상없이 사용할 수 있다.


디버깅 정보는 DCU 파일에 삽입되므로 실행 파일의 크기에는 영향을 미치지 않는다. 그러나 통합환경 디버거 외에 TD32.EXE라는 외부 디버깅 툴을 사용할 경우 Linker 페이지의 Include TD32 debug info 옵션을 선택하는데 이 옵션을 on시킬 경우 실행 파일의 크기가 증가한다. 디폴트로 이 옵션은 선택되어 있지 않지만 만약 꼭 필요할 경우는 이 옵션을 선택하되 디버깅이 끝난 후는 다시 이 옵션을 해제해 주는 것이 좋다.

# 나. 단계 실행

디버거가 가지는 주요한 기능 중의 하나는 단계 실행 기능이다. 프로그램을 그냥 실행시키면 어디서 에러가 발생하는지 알 수 없지만 한 행씩 실행시켜 나가면서 프로그램의 상태를 관찰해 보면 에러의 발생 위치를 알 수 있고 그 위치를 검토해 보면 에러가 발생한 원인도 자연히 알게 된다. 사실 버그는 찾아내는 것이 고치는 것보다 훨씬 더 어렵다. 어디서 버그가 발생했는지만 정확하게 알고 있다면 버그를 고치는 일은 그야말로 시간 문제이다.

## ■ 화면 배치

단계 실행 기능을 사용하기 위해서는 화면 배치를 잘하는 것이 중요하다. 우선 코드 에디터와 프로그램 실행 화면이 분리되어 있어야 한다. 만약 코드 에디터와 폼이 겹쳐 있는 상태에서 디버깅을 하면 한 행 실행을 할 때마다 폼과 코드 에디터가 번갈아 가며 깜박이기 때문에 보기 싫은 건 둘째치고 디버깅 속도가 현저히 감소된다. 또한 감시 윈도우나 평가 윈도우 위치도 고려하여 되도록 윈도우끼리 겹치지 않도록 배치하는 것이 좋으며 도킹 기능을 적절히 활용하여 화면을 알뜰하게 사용해야 한다. 그러기 위해서는 1024*768 이상의 고해상도 모니터라는 객관적인 조건이 필요하다.

## ■ 커서 위치까지 실행

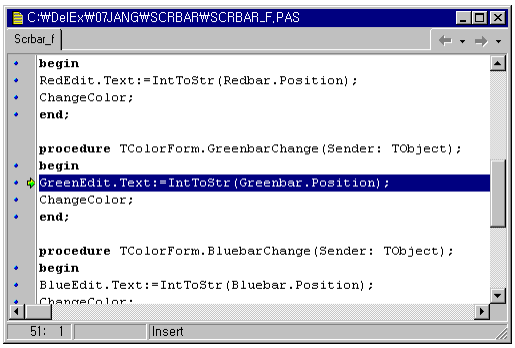
소스의 특정 부분이 의심스러울 경우는 그 부분 직전까지만 실행해 보고 프로그램을 중단시킨 후 상태를 관찰해 본다. 특정 부분까지만 프로그램을 실행시킨 후 중단시키고자 할 때 코드 에디터에서 커서를 원하는 위치에 두고 이 기능을 사용한다. 커서 위치까지만 실행시키려면 다음 방법 중 하나를 사용한다. 물론 먼저 커서를 원하는 곳에 위치시켜 두어야 한다.

① Run/Run to Cursor 메뉴 항목을 선택한다.
② 코드 에디터의 Debug/Run to Cursor 스피드 메뉴 항목을 선택한다.
③ 키보드 단축키인 F4 를 누른다. 현실적으로 가장 빠른 방법이다.

F4를 누르면 커서가 있는 위치까지 실행한 후 멈춘다. 코드 에디터에는 멈춘

위치에 파란색의 반전 막대를 보여 준다. 이 반전 막대는 다음 실행할 코드를 보여주는 것이지 실행한 코드를 보여주는 것은 아니다. 즉 이 반전 막대가 위치한 곳의 코드는 아직 실행되지 않은 상태이다.

그림
단계 실행중의 코드 에디터



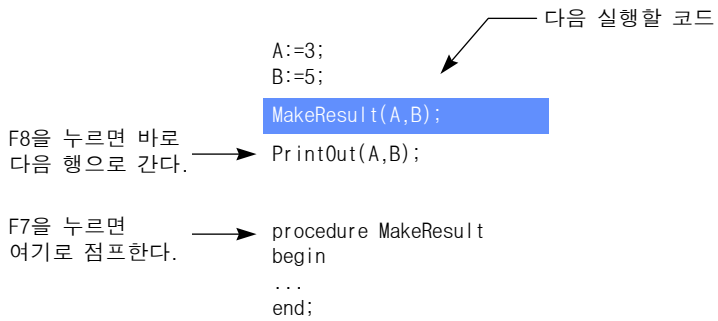

이 상태에서 변수의 값을 확인해 본다거나 변수값을 바꾼다거나 실행중인 프로그램의 상태를 확인할 수 있다. 커서 위치까지 실행 기능은 코드가 생성되는 위치에서만 가능하다. 코드가 생성되지 않는 주석문, 빈칸, 변수 선언문 등에 커서를 위치시켜 두고 F4를 누르면 이 명령은 무시된다. 코드가 생성되지 않는 곳에는 디버깅 정보도 생성되지 않기 때문에 그 위치에서 실행을 멈출 수가 없다. 잠시 후에 설명할 중단점 설정도 마찬가지로 코드가 생성되는 행에만 설정할 수 있다.

## ■ Step Over

프로그램을 한 행씩 실행한다. 한행 한행 실행시켜 가면서 변수값과 프로그램 상태를 확인해 볼 수 있다. 스피드 버튼  를 누르거나 단축키 F8을 누른다. 함수 호출문을 만났을 경우 함수 호출 전체를 하나의 명령으로 간주하고 한꺼번에 함수 호출을 수행한다.

## ■ Trace Into

Step Over와 마찬가지로 한 행씩 실행한다. 차이점이라면 함수 호출문이 있을 경우 함수 내부로까지 추적해 들어간다는 점이다. 함수 내부가 의심스러우면 이 기능을 사용하여 함수까지 검토해 보는 것이 좋지만 함수에 이상이 없다고 판단되면 Step Over를 사용하여 함수를 한번에 실행하도록 하는 것이 더 빠르다. 스피드 버튼  를 누르거나 단축키인 F7을 누른다. 단계 실행에 사용되는 세 명령의 단축키인 F4, F7, F8은 암기해 두는 것이 좋다.




그림

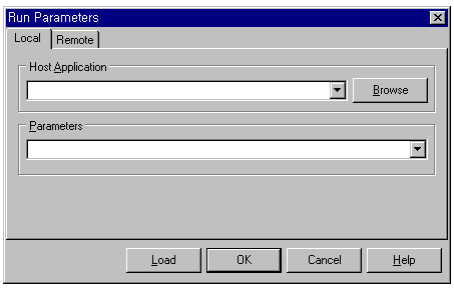
Step Over 와 Trace Into의 차이

## ■ Program Reset

단계 실행을 하던중 처음부터 다시 단계 실행을 해 보고 싶으면 Run/Program Reset을 선택하거나 단축키 Ctrl-F2를 누른다. 그러면 프로그램 실행을 강제로 중단하고 디자인 과정으로 복귀하게 되며 다시 디버깅을 시작하거나 프로그램을 실행시킬 수 있다. 그러나 이 기능은 프로그램을 강제로 중단시키기 때문에 프로그램이 할당한 자원을 잃어버린다는 단점이 있으므로 함부로 사용하지 않는 것이 좋다. 차라리 남은 부분을 실행시켜 프로그램을 정상 종료한 후 다시 디버깅을 하는 것이 더 현명한 방법이다.

## ■ 인수 지정

윈도우즈용 프로그램은 웬만해서는 명령행 인수를 사용하지 않는다. 만약 명령행 인수를 사용하는 프로그램을 만든다면 이 인수도 디버깅의 대상이 된다. Run 메뉴의 Parameters... 항목을 선택하면 인수를 입력할 수 있는 대화상자가 열린다. 여기에 인수를 지정해 주면 프로그램 실행시 이 인수가 프로그램으로 전달된다.

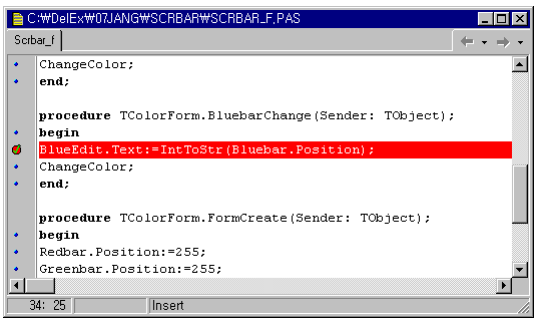


# 다. 중단점

중단점(BreakPoint)이란 프로그램 실행을 강제로 멈추도록 지정한 지점이다. 뭔가 에러가 있을 것 같은 지점에 중단점을 설정해 놓고 프로그램을 실행시키면 정상적으로 프로그램이 실행되다가 중단점에 이르면 실행을 중단하고 디버깅 상태로 돌아온다. 단계 실행으로 원하는 부분을 찾아가는 방법이 너무 느리므로 이런 경우는 중단점을 사용하는 것이 더 빠르다.

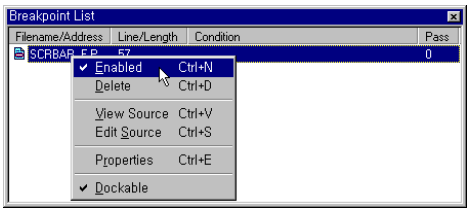
중단점을 설정하려면 코드 에디터에서 중단점을 설정하려는 부분에 커서를 위치시킨 후 F5키를 누르거나 마우스로 코드의 좌측 여백을 클릭한다. 다음과 같이 빨간 반전 막대가 표시되며 중단점으로 지정된다.

그림
중단점이 설정된 모양




F5나 마우스를 사용하는 방법 외에도 스피드 메뉴나 메인 메뉴를 사용하는 방법도 있지만 F5키가 현실적으로 가장 편리한 방법이다. 중단점을 해제하려면 중단점이 설정된 곳을 다시 한 번 중단점으로 지정하면 된다. 중단점은 한꺼번에 여러 개를 지정할 수 있으며 지정된 모든 중단점에서 프로그램 실행을 멈춘다. 또한 프로그램 실행중에 잠시 정지시킨 상태나 단계 실행중에도 중단점을 설정하거나 해제할 수 있다.

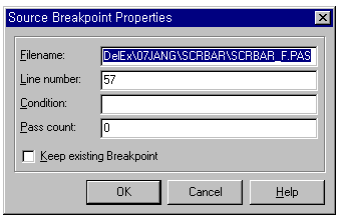
여러 개의 중단점을 설정해 놓고 디버깅을 하고 있을 경우 어떤 중단점이 어디에 위치해 있는지 코드 에디터에 한꺼번에 나타나지 않으므로 이 때는 중단점 리스트 윈도우를 열어 중단점을 관리한다. View/Debug Windows/Breakpoints를 선택하면 다음과 같은 중단점 리스트 윈도우가 열린다. 어떤 파일의 몇 행에 중단점이 설정되어 있는가가 한눈에 보인다. 중단점에 대한 여러 가지 조작은 이 윈도우의 스피드 메뉴를 사용한다.




중단점 삭제, 정지, 중단점 위치로 찾아가기 등등의 메뉴 항목들이 있다. 이 중 Properties 항목을 선택하면 중단점에 관한 정보를 수정하거나 추가적인 정보를 지정해 줄 수 있는 중단점 편집 화면이 열린다.

**그림**

중단점 조건 설정 대화상자




Condition은 중단점의 중단 조건을 지정하며 디폴트로 아무 조건도 지정되어 있지 않기 때문에 중단점은 무조건 중단된다. Condition에 특정 조건을 설정해 놓으면 조건이 만족할 때만 중단점이 기능을 발휘한다. 조건식에는 변수 평가식이 기입된다. 예를 들어 A=100이라는 조건식을 기입해 놓으면 중단점 실행중에 변수 A가 100의 값을 가지는 경우에 한해서만 실행을 중단한다. 소스의 특정 행에 이상이 있어 중단점으로 설정할 때 다른 경우는 이상이 없는데 A가 100이 되기만 하면 에러가 발생한다면 이런 조건을 주어 A가 100일 때 실행을 중단하여 프로그램의 상태를 확인한다.

Pass Count는 중단점을 몇 번까지 그냥 지나칠 것인가를 지정한다. Pass가 5로 지정되어 있다면 최초 4번은 중단점을 무시하고 중단점을 5번째 실행할 때 프로그램 실행을 중단한다. 루프가 몇 번까지는 이상없이 돌아가는데 일정 횟수를 경과하면 이상이 발생할 경우 Pass Count를 사용한다.

# 라. 변수 점검

버그의 원인 중 가장 일반적이고 자주 발생하는 원인이 변수를 잘못 조작하는 것이다. 변수에 잘못된 값을 대입한다거나 또는 함수가 잘못된 리턴값을 넘겨주는 경우에는 프로그램이 원하는 출력을 낼 수 없다. 단계 실행을 하면서 변수값을 살펴보면 어디가 어떻게 잘못되는지를 알 수 있다. 살펴볼 변수란 주요 전역

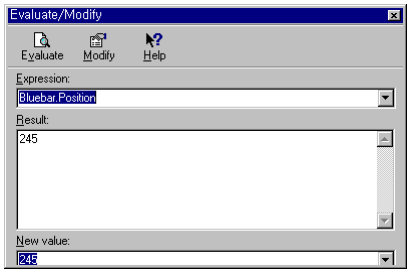
변수, 루프 제어 변수, 함수의 인수 등이다.

## ■ 변수 평가 및 변경

평가 윈도우는 변수값을 평가해 보는 가장 간단한 방법이며 디버깅중에 언제든지 사용할 수 있다. 코드 에디터에서 변수가 있는 위치에 커서를 두고 Ctrl-F7을 누르거나 Run 메뉴의 Evaluate/Modify를 선택한다. 다음과 같은 변수 평가 화면이 열린다.

**그림**

실행중에 변수값을 조사할 수 있는 평가 화면




윈도우가 열리면서 Expression란에는 코드 에디터의 커서 위치에 있는 변수 이름이 기입되는데 다른 변수 이름을 직접 기입해 넣어도 된다. Evaluate 버튼을 누르면(또는 키보드의 Enter 키를 누르면) Result란에 변수를 평가한 결과가 출력된다. New Value란에 새로운 값을 입력하고 Modify 버튼을 누르면 변수값을 실행중에 변경할 수도 있다.

값을 평가할 때 Result란에 출력되는 출력 형식을 변경하려면 변수 뒤에 콤마를 찍고 다음과 같은 출력 형식을 지정한다. 어떤 변수는 10진수 형태로 보는 것이 더 편리한 경우도 있을 것이고 16진수로 보는 것이 더 편리한 경우도 있으므로 보고자 하는 형태를 지정해 준다.

**표**

평가 화면의 변수 출력 형식

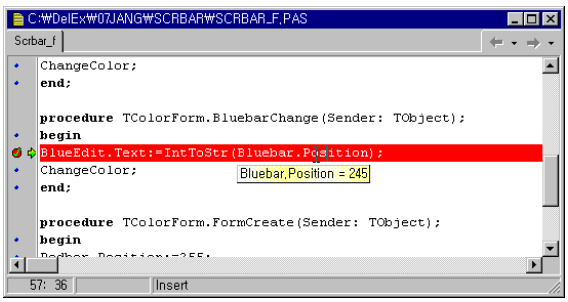
| 형식 | 변수형 | 출력 |
|---|---|---|
| H 또는 X | 정수형 | 16진수 형태로 출력된다. |
| C | 문자형 | 0..31까지 코드를 특별한 문자로 보여줌 |
| D | 정수형 | 10진수로 출력된다. |
| Fn | 실수형 | n유효숫자까지 출력된다. |
| nM | 모든 변수 | 메모리 덤프 형태로 보여주며 n으로 덤프 바이트 수를 지정하며 생략시 데이터형의 길이만큼 출력된다. |
| P | 포인터 | Seg:Off 형식으로 출력된다. |

| | | |
|---|---|---|
| R | 레코드, 클래스 | 필드 이름과 값을 같이 보여준다. |
| S | 문자형 | 출력불가능한 아스키 문자를 #nn형태로 보여준다. |

변수 평가 윈도우에서는 프로그램에서 사용하는 모든 변수값을 평가할 수 있으며 컴포넌트의 속성, 레코드, 배열 등의 데이터형도 모두 평가할 수 있다. 더구나 연산자를 사용한 수식도 평가할 수 있다. I*5+8과 같은 간단한 일차 함수식에서부터 Ar[i*5][j+5]와 같은 첨자 연산, 포인터 연산까지 사용할 수 있다. 코드에서 :=의 우측에 놓일 수 있는 모든 수식을 평가할 수 있다. 단 함수를 호출하는 수식은 사용할 수 없다.

## ■ 툴팁형 값 평가

Evaluate/Modify 기능은 값을 정밀하게 보고 싶다거나 배열 등의 큰 변수일 때 사용하며 간단한 변수라면 이 기능으로 빠르게 값을 조사할 수 있다. 디버깅 중에 확인하고자 하는 변수값 위치에 마우스 커서를 갖다대면 툴팁 형태로 해당 변수의 현재값을 보여준다.



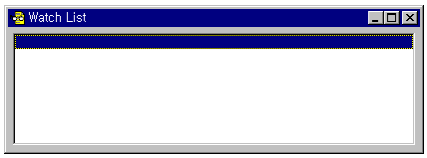

마우스 커서만 갖다대면 값을 바로 확인할 수 있어 무척 편리하다.

## ■ 와치 윈도우

변수를 평가하는 기능은 단계 실행을 할 때마다 매번 검사를 해 보아야 하지만 와치 윈도우에 변수를 등록해 두면 프로그램이 실행되는 단계마다 변수값을 계속 출력해 주어 변수값이 어떻게 변하는가를 살펴볼 수 있도록 해 준다. 와치 윈도우를 열려면 View 메뉴의 Debug Windows/Watches 항목을 선택한다.

그림

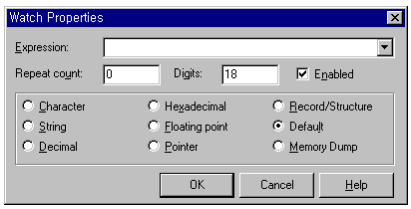
실행중에 항상 변수 값을 감시할 수 있는 와치 윈도우




현재는 아무 변수도 등록되어 있지 않지만 다음 방법으로 변수를 등록시킨다.

① Run 메뉴의 Add Watch 를 선택한다.
② 코드 에디터의 스피드 메뉴에서 Debug/Add Watch at Cursor 를 선택한다.
③ 키보드 단축키인 Ctrl-F5 를 누른다.
④ 와치 윈도우를 더블클릭한다.
⑤ 와치 윈도우의 스피드 메뉴에서 Add Watch 를 선택한다.

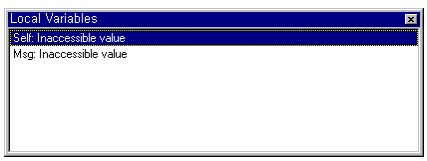
그러면 다음과 같은 변수 등록 윈도우가 열릴 것이다.




와치 윈도우에 등록하고자 하는 변수 이름과 어떤 형식으로 변수값을 출력해 줄 것인가를 지정해 준다. 와치 윈도우의 스피드 메뉴를 사용하여 변수를 등록 취소하거나 잠시 정지시키거나 추가한다.

## ■ 지역 변수 확인

디버깅을 하다보면 이 함수 저 함수로 옮겨 다니게 되는데 그때마다 지역 변수들을 매번 확인하려면 무척 귀찮다. 지역 변수 목록이 함수마다 다르기 때문에 와치 윈도우를 쓸 수도 없는데 이 경우는 지역 변수 윈도우를 사용하면 편리하다. View/Debug Windows/Local Variables 를 선택하면 이 윈도우가 나타난다.

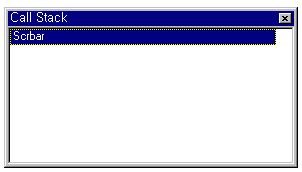


함수 사이를 옮겨 다녀도 항상 현재 함수의 지역 변수들을 보여주므로 지역 변수는 언제나 이 윈도우에서 값을 확인할 수 있다.

# 마. 기타 디버깅 툴

## ■ 콜 스택

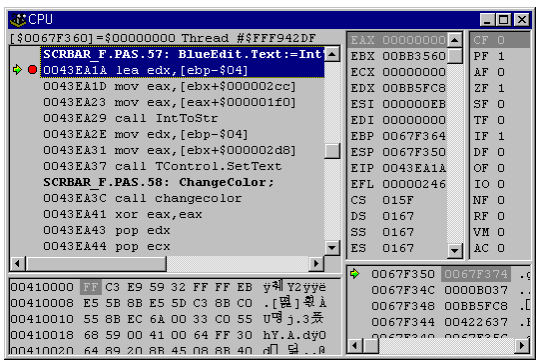
콜 스택 윈도우는 현재의 함수가 호출되기 전에 거쳐온 모든 함수들의 목록을 보여준다.




## ■ CPU 윈도우

이 윈도우는 모든 디버깅 윈도우 중에 제일 복잡하고 제일 강력하면서도 또한 제일 위험한 윈도우이다. CPU 의 내부를 보여주며 직접 수정도 할 수 있기 때문에 어떠한 변경도 가할 수 있지만 차칫 잘못 건드리면 디버깅이고 뭐고 시스템이 죽어버리는 경우도 발생할 수 있다.

그림
CPU 윈도우

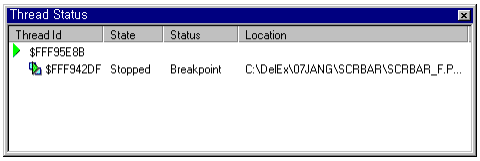



모두 5 개의 영역으로 나누어져 있는데 각각 디스 어셈블리, 레지스터, 플래그, 메모리, 스택이다. 각 영역의 팝업 메뉴에는 다양한 명령어들이 존재하는데 팝업 메뉴 이름 자체가 설명적으로 작성되어 있으므로 별도의 설명을 할 필요는 없을 것 같다. 이 윈도우의 내용을 이해하려면 기본적으로 어셈블리 언어에 대

한 개념이 있어야 하며 또한 어셈블리 프로그램을 디버깅해 본 적이 있어야 한다. 어셈블리만 안다면 이 윈도우의 사용법은 아주 식은 죽 먹기지만 그렇지 못한 사람에게는 별 소용이 없을 것이다.

## ■ 스레드 윈도우

이 윈도우는 현재 실행되고 있는 스레드의 목록과 상황을 보여준다. 싱글 스레드 프로그램에서는 별 필요가 없겠지만 멀티 스레드 프로그램을 디버깅할 때는 유용하게 사용될 것이다. 목록에는 스레드의 ID, 현재 상태, 소스의 위치 등이 표시된다.




여기서 소개한 디버깅 툴 외에도 델파이 4에는 네트웍을 통한 리모트 디버깅 기능이 제공된다. 별도의 디버깅 클라이언트를 설치하고 네트웍 옵션, 권한까지 설정해 주어야 하기 때문에 아무나 사용할 수는 없지만 가능하다면 아주 능률적으로 디버깅을 할 수 있을 것 같다.

델파이가 제공하는 통합환경 디버거는 대단히 우수한 성능과 편리한 기능을 가지고 있다. 프로그램 개발 과정의 대부분이 디버깅 과정임을 감안할 때 디버거를 얼마나 잘 쓰는가는 프로그램을 얼마나 정확하고 빠르게 짤 수 있는가를 좌우한다. 디버거를 사용하는 방법은 단순히 설명서를 읽고 귀에 주워담는다고 해서 터득할 수 있는 것이 아니다. 몸소 디버깅을 해 보고 손에 익고 머리에 익숙해져야 제대로 디버깅을 한다고 할 수 있다. 디버깅의 도사가 되고 싶으면 적극적으로 디버거를 활용하겠다는 자세가 중요하다고 생각한다.

# 17-2 컴파일러 지시자

어느 프로그램에서나 사용자의 취향이나 기호에 따라 프로그램을 다양하게 쓸 수 있도록 해 주는 옵션이라는 것이 제공된다. 대부분의 경우 가장 편리하고 무난한 값으로 디폴트 옵션이 설정되므로 굳이 옵션을 건드리지 않아도 되며 완전히 무시해도 큰 상관은 없다. 그러나 델파이같은 컴파일러에서는 옵션이 무척 중요하다. 옵션을 어떻게 설정했는가에 따라 컴파일된 결과가 달라지기 때문이다.

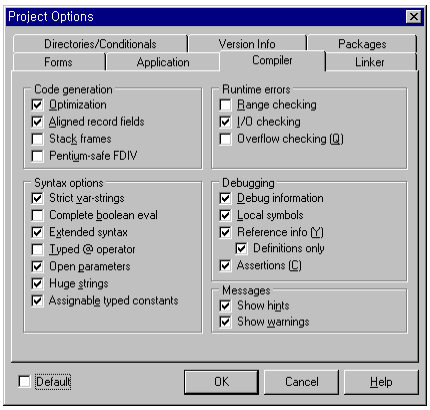
컴파일러 지시자는 컴파일러가 프로그램을 기계어로 번역할 때 어떤 방식을 사용할 것인가를 설정하는 옵션이며 컴파일에 관계된 옵션과 링크에 관계된 옵션이 있다.

## 가. 컴파일러 지시자

델파이에서 컴파일 옵션을 설정하는 방법에는 두 가지가 있다. 첫째는 대화상자를 통해 지정하는 것이고 둘째는 소스 코드에 직접 컴파일러 지시자를 포함시키는 방법이다. 컴파일러 지시자는 소스 코드에 주석의 형태로 표기되며 {$A+} {$D+} 등과 같이 주석 안에 $문자를 넣어 주석이 아닌 컴파일러 지시자임을 나타낸다.  대화상자를 사용하여 지정하는 옵션은 전체 프로젝트에 영향을 미치며 소스 코드에 포함되는 컴파일러 지시자는 국부적인 부분에만 영향을 미친다.

컴파일러 옵션을 지정하는 대화상자는 Project/Options...의 Compiler 페이지에 있다. 모든 옵션은 체크 박스 형태로 되어 있으므로 체크 박스를 선택/비선택하여 옵션을 지정한다.

그림
컴파일러 옵션 설정 대화상자




두 가지 옵션 지정 방법 중에 어떤 것을 쓸 것인가는 사용자의 자유이다. 아무래도 소스 코드에 컴파일러 지시자를 삽입하는 방법보다는 대화상자가 사용하기 편리할 것이다. 그러나 여기서는 소스 코드에 포함되는 컴파일러 지시자를 기준으로 설명하도록 한다. 대화상자의 옵션들도 컴파일러 지시자와 동일하므로 하나만 알면 둘 다 쓸 수 있다. 델파이가 제공하는 컴파일러 지시자는 세 가지 형태가 있다.

✶ 스위치형 : +, - 기호로 On/Off 를 지정한다. {$F+} {$A-} 등이 그 예이다. + 기호가 뒤에 있으면 해당 옵션을 가능하게 한다는 뜻이고 -기호가 뒤에 있으면 해당 옵션을 쓰지 않는다는 뜻이다.

✷ 인수형 : 옵션 뒤에 인수를 넘겨주는 형태이다. {$I 파일이름} {$R 파일이름}등과 같이 사용된다.

✹ 조건부 컴파일 지시자 : 값을 평가하여 그 결과에 따라 컴파일 동작을 결정하는 형이다. {$DEFINE} {$IFDEF} 등이 있다.

## ■ Optimizations {$O}

컴파일 과정에서 최적화를 수행한다.

## ■ Aligned Record fields {$A}

이 옵션이 선택되어 있으면 2바이트의 변수를 워드 정렬한다. 80x86 CPU는 특성상 홀수 번지에 있는 데이터를 읽는 것보다 짝수 번지에 있는 데이터를 읽는 속도가 두 배 빠르다. 워드 정렬이란 홀수 번지에 놓여질 데이터를 한 바이트 건너뛰어 강제로 짝수 번지로 옮겨주는 것을 말한다. 메모리 1바이트가 버려지지

만 대신 속도가 빨라진다. 단 이 옵션은 1바이트 크기의 데이터, 레코드나 오브젝트의 필드 그리고 배열 요소에는 적용되지 않는다. 배열이나 레코드 내부까지 정렬하면 메모리 낭비가 너무 심해지기 때문이다. 전역 변수는 이 옵션에 상관없이 워드 정렬된다.

## ■ Pentium-Safe FDIV {$U}

펜티엄 칩의 나누기 오류를 피할 수 있는 코드를 작성한다. 오류가 있는 펜티엄 시스템에서 프로그램이 실행될 경우 나눗셈 연산은 소프트웨어적으로 실행되며 따라서 실행 속도가 약간 저하된다. 이 옵션은 디폴트로 선택되어 있지만 아주 정확한 나눗셈이 필요하지 않다면 이 옵션을 취소하여 실행 속도를 더 빠르게 할 수도 있다.

## ■ Stack frames {$W}

프로시저와 함수 호출문에 스택 프레임을 생성할 것인가 아닌가를 지정한다. 이 옵션이 선택되어 있으면 스택 프레임이 필요없더라도 항상 스택 프레임을 생성하며 이 옵션이 선택되어 있지 않으면 꼭 필요할 때만 스택 프레임을 생성한다. 일반적으로 함수가 지역 변수를 가지면 스택 프레임이 생성된다. 대개의 경우 이 옵션을 선택할 필요는 거의 없지만 디버깅 툴 중에 모든 함수와 프로시저가 스택 프레임을 가져야만 디버깅이 가능한 경우가 있기 때문에 이 옵션이 제공된다.

## ■ Range Checking {$R}

배열 첨자와 문자열 첨자가 올바른 범위 내에 있는지를 점검하도록 한다. 배열의 범위 밖을 잘못 액세스할 경우 메모리가 파괴되어 심각한 에러가 발생할 수도 있으므로 이 옵션을 선택하여 범위밖을 액세스하지 못하도록 금지한다. 이 옵션은 프로그램을 느리게 만들고 크기를 크게 하기 때문에 디폴트로 선택되어 있지 않지만 개발중에나 디버깅중에는 선택하는 것이 좋다.

## ■ I/O Checking {$I}

디스크로 파일을 읽거나 쓰는 I/O 함수에 입출력 결과를 점검하는 코드를 작성하도록 한다. 이 옵션이 선택되어 있으면 입출력 에러 발생시 예외를 발생시키며 프로그램을 중단시킬 수도 있다. 예외 발생을 금지시키고 프로그램에서 직접 IOResult 함수를 사용하여 에러를 점검할 경우는 이 옵션을 끄도록 한다.

## ■ Overflow checking {$Q}

산술적인 연산의 결과 오버 플로우가 발생할 경우는 변수에 잘못된 값이 대입될 수도 있다. 이 옵션을 선택하면 가감 승제및 Sqr, Abs, Succ, Pred 등의 산술 함수에 오버 플로우를 점검하는 코드를 덧붙인다. 그러나 증감 함수인 Inc, Dec에는 점검 코드를 덧붙이지 않는다. 이 옵션도 속도와 크기에 불리하므로 최종 컴파일시에는 선택하지 않는 것이 좋다.

## ■ Strict Var strings {$V}

이 옵션이 선택되면 참조 호출로 전달된 문자열 형의 형식 인수와 실인수를 비교하여 두 인수가 틀릴 경우 에러 메시지를 출력한다. 문자열 인수의 경우 두 인수의 길이 차이로 인해 자칫 엉뚱한 메모리 영역을 파괴할 수도 있기 때문에 컴파일러가 두 인수를 비교하도록 한다. 단 이 옵션은 짧은 문자열을 사용할 때만 의미가 있으며 긴 문자열을 사용할 때는 전혀 선택할 필요가 없다. 하위 버전과의 호환성 유지를 위해 지원되는 옵션이다.

## ■ Complete boolean evaluation {$B}

이 옵션이 선택되면 두 개 이상의 조건이 결합되어 있는 논리식을 평가할 경우 식 전체의 결과가 이미 계산되었더라도 나머지 식을 모두 평가하도록 한다. 이 옵션은 디폴트로 선택되어 있지 않으므로 불필요한 논리식을 점검하지 않는다. 이를 쇼트 서키트(Short Circuit : 적당한 번역이 없음) 기능이라고 하며 대부분의 고급 컴파일러들이 이 기능을 지원한다. 예를 들어 다음과 같은 조건식을 보자.

```
if (A=3) and (B=4) then
```

첫 번째 식을 평가해 본 결과 A가 3이 아니면 두 번째 식의 평가 결과와는 무관하게 전체 조건식은 거짓이 된다. 이 경우 B가 4인지 아닌지를 점검하는 것은 불필요하므로 아예 점검을 하지 않게 되며 이를 쇼트 서키트 기능이라고 한다. 불필요한 식을 평가하지 않으므로 프로그램의 속도는 당연히 빨라진다. 다음 식도 마찬가지이다.

```
if (A=3) or (B=4) then
```

A값이 3이라면 뒤의 식은 볼 것도 없이 이 식은 참이 된다. 그러나 쇼트 서키

트 기능은 수학적으로 분명히 완벽한 근거를 가지고 있음에도 불구하고 문제가 되는 경우가 있다. 다음과 같이 수식 중에 함수 호출문이 포함된 경우가 그렇다.

```
if (A=3) or (AreYouDie(x,y)) then
```

이 경우 A가 3이라면 뒤의 함수 호출문의 리턴값에 상관없이 전체 조건식은 참이 되며 함수는 호출되지 않는다. 물론 결과적으로 조건 점검은 정확하게 했지만 함수 호출이 되지 않았으므로 프로그래머가 의도한 결과와는 달라질 수 있다. 이런 문제가 발생할 경우는 이 옵션을 선택하지 않아야 하며 디폴트로 선택되어 있지 않다.

최대한 프로그램의 속도를 빠르게 하고 싶다면 이 옵션을 선택하되 위와 같이 문제를 일으킬 수 있는 조건 점검문은 쇼트 서키트가 적용되지 않도록 고쳐 써 주어야 한다. 예를 들어 조건문의 순서를 바꾸든가 아니면 함수 호출문을 조건문 밖으로 끄집어 내 주도록 한다.

## ■ Extended Syntax {$X}

오브젝트 파스칼의 문법을 확장한다. 원래 파스칼의 함수는 단독으로 사용될 수 없고 반드시 수식내에서 사용되어야 하지만 이 옵션을 선택하면 C의 경우처럼 함수만 단독으로 사용할 수 있으며 리턴값이 버려지게 할 수도 있다. 그러나 System 유닛에 있는 표준 함수는 문법을 확장해도 단독으로 사용할 수 없으며 사용자가 정의한 함수의 경우에만 확장 문법을 적용할 수 있다. 또한 원래 파스칼에는 없는 널 종료 문자열(PChar형)을 사용할 수 있도록 해준다.

## ■ Typed @ Operator {$T}

@연산자가 리턴하는 포인터는 타입이 정해져 있지 않기 때문에 다른 포인터 타입과 함께 사용할 수 있다. 그러나 이 옵션을 선택하면 @연산의 결과는 피연산자형의 포인터가 된다. 예를 들어 정수형 변수 i가 있을 때 @i의 결과는 정수형의 포인터가 된다.

## ■ Open Parameters {$P}

string으로 전달된 참조 인수의 의미를 통제하며 이 옵션이 선택되어 있을 경우 문자열 인수가 다양한 크기를 가질 수 있도록 허용한다.

## ■ Huge Strings {$H}

델파이 2.0 버전에서 추가된 컴파일러 지시자다. 이 옵션이 선택되어 있으면 string 타입이 긴 문자열이 되며 선택되어 있지 않으면 1.0 버전에서와 마찬가지로 255자의 길이 한계를 가진다. 디폴트로 이 옵션은 선택되어 있으므로 특별히 옵션을 바꾸지 않으면 긴 문자열을 사용할 수 있도록 되어 있지만 과거 버전과의 호환성에 문제가 있을 경우는 잠시 이 옵션을 취소하면 된다. 델파이의 VCL에 포함된 모든 컴포넌트는 이 옵션이 선택된 상태로 컴파일되었으므로 모두 긴 문자열을 사용한다.

## ■ Assignable typed constants {$J}

타입 상수의 변경 가능 여부를 결정한다. 이 옵션이 선택되어 있으면 타입 상수의 값을 실행중에 변경할 수 있지만 선택되어 있지 않으면 값을 변경할 수 없는 상수가 된다. 1.0 버전에서는 타입 상수의 값을 마음대로 바꿀 수 있었지만 2.0 버전 이상에서는 옵션으로 변경 여부를 선택할 수 있도록 되어 있다. 왜냐하면 2.0 버전에서는 타입 상수를 사용하지 않더라도 전역 변수에 초기값을 설정할 수 있도록 변경되었기 때문이다.

## ■ Debug Information {$D}

DCU 파일에 디버깅 정보를 넣을 것인가 아닌가를 지정한다. 이 옵션을 선택하면 유닛 파일에 프로시저의 행번호를 삽입하여 디버깅에 사용할 수 있도록 해준다. 이 옵션의 선택여부는 실행 파일의 크기나 속도에 전혀 영향을 주지 않는다. 디버깅 정보는 유닛을 컴파일한 DCU 파일에만 들어갈 뿐이며 실행 파일에는 들어가지 않기 때문이다.

## ■ Local Symbols {$L}

implementation부에 정의되어 있는 변수(또는 함수)나 프로시저, 함수 내부에서 사용되는 지역 변수에 관한 정보를 유닛 파일에 삽입한다. 이 옵션을 선택하면 독립형 디버거에서도 지역 변수의 값을 검사하거나 변경할 수 있다. 유닛 파일의 크기는 증가시키지만 실행 파일의 크기나 속도에는 영향을 미치지 않는다.

## ■ Symbol info {$Y}

유닛 파일에 모든 심볼의 행번호를 삽입하도록 한다. 심볼 정보는 오브젝트

브라우즈가 심볼의 정의와 참조를 출력할 때 사용한다. 유닛 파일의 크기는 증가시키지만 실행 파일의 크기나 속도에는 영향을 미치지 않는다.

### ■ Assertion {$C}

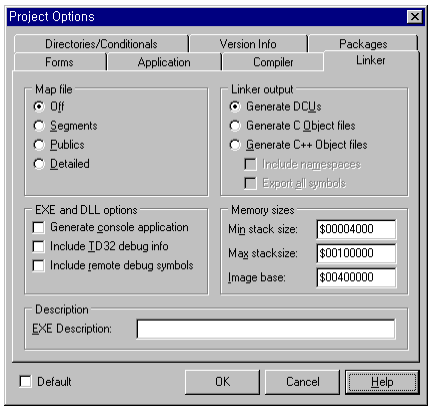
이 옵션을 선택하면 소스내의 확인 구문이 컴파일되어 잘못된 코드를 잡아 내는 용도로 사용된다. 최종 릴리즈 버전에서는 이 옵션을 선택 해제하여 확인 구문을 삭제하는 것이 좋다.

## 나. 링커 옵션

링커 옵션은 컴파일한 모듈을 결합하여 실행 파일을 만드는 방식에 대한 옵션이다. Project/Options 의 Linker 페이지에서 옵션을 설정한다.

**그림**

링커 옵션 설정 대화상자




### ■ Map File Options

맵 파일이란 링크한 결과 생성되는 여러 가지 정보를 담는 파일이며 다른 언어로 작성한 프로그램과 연결한다거나 문제 발생시 참고하는 정보이다. 대규모의 프로젝트가 아니면 맵 파일까지 작성해야 할 경우는 드물다. 맵파일을 어느 정도 수준으로 작성할 것인가를 옵션으로 지정하며 Off이면 맵 파일을 작성하지 않고 Detailed이면 세그먼트 리스트, 프로그램 시작 번지, 링크 도중에 발생한 에러와 경고, 다른 프로그램과 공유하는 심볼들의 이름, 그 외 세그먼트의 주소, 길이, 이름, 그룹 등에 관한 정보가 상세하게 작성된다. Segments, Publics는 Off와 Detailed의 중간 형태이다.

작성된 맵 파일은 프로젝트 파일과 같은 이름을 가지며 프로젝트 디렉토리에

생성된다. 확장자는 MAP이다. 텍스트 파일이므로 메모장같은 에디터로 살펴보면 된다.

## ■ Link Output

링커에 의해 생성될 파일을 지정한다. DCU 파일을 선택하면 델파이 고유의 목적 파일인 DCU 파일을 생성하며 Object file을 선택하면 표준 2진 목적 파일인 OBJ 파일을 생성한다. 다른 언어와 혼합 프로그래밍을 할 경우는 OBJ 파일을 생성하여 OBJ 파일끼리 링크할 수 있도록 해 주어야 한다. C++형식의 오브젝트 파일도 생성할 수 있다.

## ■ Generate console application

실행 파일에 콘솔 모드에서 실행되는 프로그램임을 알리는 플래그를 설정하도록 한다.

## ■ Include TD32 Debug Info

실행 파일에 디버깅 정보를 삽입하여 윈도우즈용 터보 디버거로 디버깅을 할 수 있도록 해 준다. 디버깅 정보가 추가로 들어가므로 실행 파일 크기가 늘어나지만 프로그램의 속도나 메모리 요구량에는 영향을 주지 않는다.

## ■ Include remote dubug symbols

네트웍을 통한 원격 디버깅을 하고자 할 때 이 옵션을 선택한다.

## ■ Memory Size

스택 세그먼트의 크기를 지정해 준다. 스택을 많이 사용하는 프로그램을 작성할 때는 최대 크기를 늘려 준다. Image base란 컴파일한 프로그램을 메모리로 읽어올 번지를 지정하되 DLL을 만들 경우에만 사용된다.

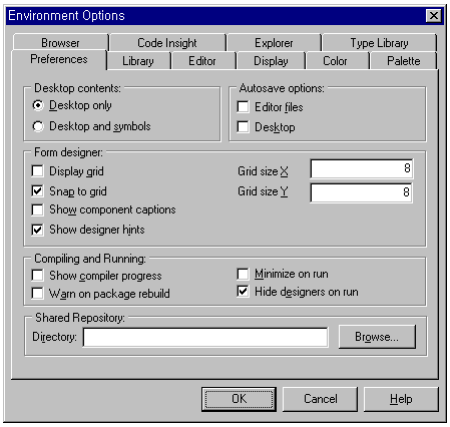
## ■ Exe Description

실행 파일에 대한 간단한 설명을 255 자까지 작성할 수 있으며 여기서 작성한 문자열은 실행 파일에 포함된다. 보통 프로그램의 저작권자 등을 표기한다.

# 17-3 환경 설정

델파이의 개발 환경은 디폴트로만 사용해도 상당히 편리하도록 구성되어 있지만 개성에 따라 개발 환경을 바꾸어 사용할 수도 있다. 개발 환경에 대한 전반적인 설정 상태를 지정하려면 Tools/Environment Options...항목을 선택하여 환경 설정에 대한 대화상자에서 옵션을 변경한다.

그림
환경 설정 대화상자




설정하는 옵션의 종류에 따라 여러 개의 페이지를 가지고 있으며 각 페이지는 위쪽의 페이지 탭을 클릭하여 교체한다. 각 페이지 별로 옵션의 의미를 알아보자.

## 가. Preference

### ■ Desktop Contents

델파이를 끝낼 때 어떤 정보를 저장할 것인가를 지정한다. Desktop Only를 선택하면 디렉토리 정보, 에디터에 열려 있는 파일과 열려 있는 윈도우에 관한 정보를 저장하고 Desktop and Symbols를 선택하면 최후 컴파일에 성공한 브라우저 심볼 정보도 함께 저장한다.

### ■ AutoSave

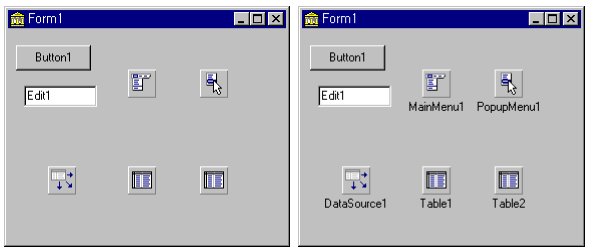
자동으로 저장할 정보를 선택한다. Editor Files 체크 박스를 선택하면 프로그

램을 실행시키거나 델파이를 끝낼 때 변경된 모든 파일을 저장한다. 혹시라도 시스템이 다운될 경우에 대비해서 이 속성을 선택해 놓는 것이 좋다. Desktop 체크 박스를 선택하면 델파이를 끝내거나 프로젝트를 닫을 때의 데스크탑 상황을 저장한다. 그래서 다시 델파이를 시작할 때 열려 있던 모든 파일을 다시 열어 주어 작업의 연속성을 확보해 준다.

## ■ Form Designer

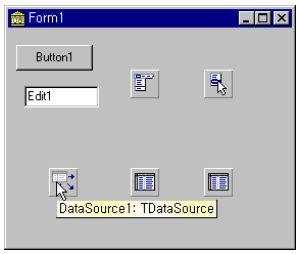
디자인시에 폼에는 컴포넌트 배치를 도와주는 격자점들이 배열되어 있다. 여기서 이 격자점에 관한 설정을 한다. Display Grid는 격자점을 표시하도록 해 주며 Snap to Grid를 선택하면 컴포넌트를 이동하거나 배치할 때 격자점에 달라붙도록 한다. 이 옵션을 선택한 상태에서는 마우스로 드래그하여 격자점 사이에 컴포넌트를 배치할 수는 없으며 오브젝트 인스펙터에서 직접 Left, Top 속성을 조정해야만 점 단위로 컴포넌트를 배치할 수 있다. 오른쪽에 있는 두 개의 옵션은 격자의 크기를 설정하며 2~128까지의 범위에서 설정한다. 디폴트 격자 크기는 8이다.

Show Component Captions 옵션을 선택하면 폼에 배치된 컴포넌트 아래쪽에 컴포넌트의 이름을 출력해 준다. 다음 그림의 왼쪽은 이 옵션을 선택하지 않았을 때이며 오른쪽은 이 옵션을 선택했을 때이다.




메뉴나 데이터 액세스 컴포넌트, 타이머 등의 비가시적 컴포넌트는 아이콘만 표시되기 때문에 여러 개가 있을 경우 잘 구분이 되지 않는데 이 옵션을 사용하면 디자인중에 캡션이 나타나므로 컴포넌트간의 구분을 쉽게 할 수 있다. 디폴트로 이 옵션은 선택되어 있다.

Show designer hints 옵션은 폼상의 컴포넌트에 마우스 커서를 갖다대면 그 컴포넌트의 캡션을 보여준다. 역시 컴포넌트간의 구분을 쉽게 하기 위해 제공하는 옵션이다.

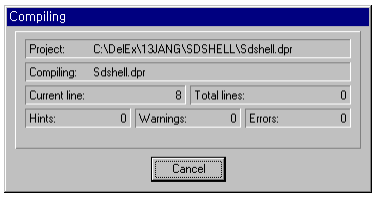



## ■ Compiling and Running

컴파일과 실행에 대한 몇 가지 세부 옵션들이 있다.

Show Compiler Progress 옵션을 선택하면 컴파일시 컴파일 진행 상황을 별도의 윈도우를 열어서 보여준다.

그림
컴파일 상황 윈도우




디폴트로 이 속성은 선택되어 있지 않기 때문에 컴파일을 시켜도 컴파일 상황이 전혀 화면에 나타나지 않는다. 델파이의 컴파일 속도가 워낙 빠르기 때문에 이런 대화상자가 불필요하기도 하지만 대규모의 프로젝트를 컴파일 할 때는 컴파일 상황을 보여주는 것이 기다리기에 지루하지 않다. 이 옵션이 디폴트로 선택되어 있지 않은 것을 보면 볼랜드가 델파이의 컴파일 속도에 대해 얼마나 자신만만하게 생각하는가를 알 수 있으며 실제로 이 대화상자가 필요없을 정도로 델파이는 충분히 빠르다.

Warn on package rebuild 옵션은 말 그대로 package를 rebuild할 때 Warn을 해 주는 옵션이다. Minimize on Run 옵션은 델파이로 작성한 프로그램을 실행시킬 때 델파이를 최소화시키며 프로그램 실행이 끝나면 델파이를 다시 원래의 크기대로 복구한다. 디폴트로 이 옵션은 선택되어 있지 않은데 화면이 좁아 실행시키기에 걸리적거리면 이 옵션을 선택하면 된다. Hide Designer On Run 옵션을 선택하면 델파이로 작성한 프로그램을 실행시킬 때 오브젝트 인스펙터와 폼 디자인 화면을 최소화시켜 숨긴다.

## ■ Shared Repository

오브젝트 창고 파일인 DELPHI32.DRO 파일을 찾을 경로를 지정한다. 이 란

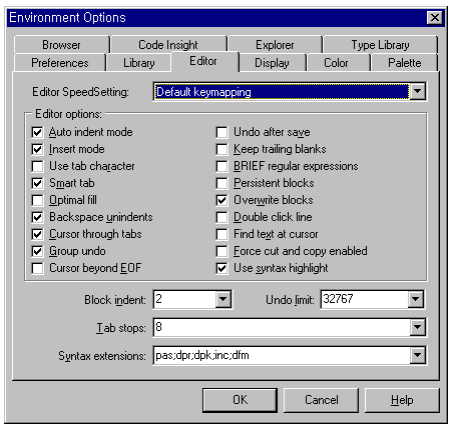
이 비어있으면 BIN 디렉토리에서 찾게 된다.

# 나. 에디터 옵션

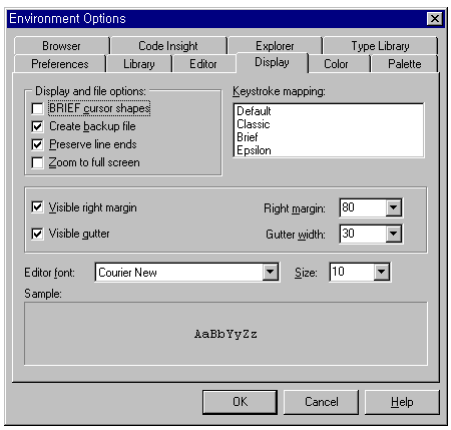
코드 에디터의 여러 가지 옵션들이며 전체 3개의 페이지로 구성되어 있다. Editor 페이지는 에디터의 편집 기능을 정의하는 대화상자이다. 디폴트가 워낙 무난하게 설정되어 있으므로 특별히 조정해 줄 필요는 없지만 조금이라도 에디터를 능률적으로 쓰고 싶은 사람은 이 옵션을 연구해 보기 바란다.

**그림**

에디터 옵션 대화상자

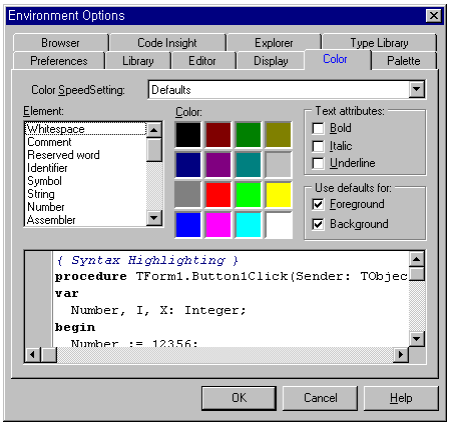



컴파일러용의 에디터를 웬만큼 써 본 사람은 직관적으로 옵션의 의미를 알 수 있으므로 자세한 설명은 생략한다. 두 번째 대화상자는 파일 관리 체계와 코드 에디터에서 사용할 폰트의 종류와 크기를 선택한다.



커서의 모양을 바꾼다거나 백업 파일의 생성 유무, 코드 에디터를 최대화할 경우의 처리 등을 지정할 수 있다. 선택할 수 있는 폰트는 모두 고정 폭을 가진 폰트뿐이며 바꾸어 봐야 별 차이를 느끼지 못할 것이다. 세 번째 대화상자는 코드 에디터의 코드 출력에 사용할 폰트 색상과 모양을 지정한다.

그림
에디터 색상 설정 대화상자




델파이의 코드 에디터는 작성된 코드의 의미를 즉시 해석하여 코드의 문법적 의미에 따라 색상이나 폰트의 모양을 다르게 보여줌으로써 코드를 시각적으로 작성할 수 있도록 돕는다. 이 대화상자에서는 각종 요소의 색상을 설정해 준다. 설정하고자 하는 요소를 Element 리스트 박스에서 선택한 후 컬러 그리드에서 전경색과 배경색을 선택하고 문자의 속성을 정의해 준다. 신텍스 하이라이팅 기능은 단순히 보기 좋게 하려고 제공되는 기능이 아니므로 최대한 작업 능률을 향상시킬 수 있도록 눈에 잘 띄는 색상을 배치하는 것이 좋다.

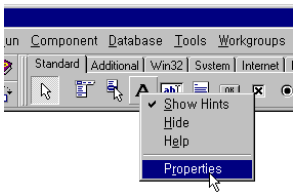
색상은 자기 기호에 맞게 마음대로 변경할 수 있지만 가급적이면 오래 봐도 피곤하지 않은 색상을 선택하는 것이 좋다. 당장 보기 좋다고 노랗게, 빨갛게 해 놓다가는 조만간 안과를 찾거나 정신병원을 찾게 될 것이다. 색상 배치에 별로 자신이 없는 사람은 디폴트 색상을 사용하는 것이 제일 무난하다.

# 다. 팔레트 설정

컴포넌트 팔레트도 옵션 설정의 대상이며 컴포넌트의 배치 상태도 마음대로 바꿀 수 있다. 기존의 컴포넌트 배치를 바꾸는 것은 별로 바람직하지 않지만 추가로 설치한 컴포넌트는 자신의 용도에 맞게 한 곳에 모아 둔다거나 용도별로 분류해 놓는 것이 좋을 것이다. 팔레트를 재설정하고자 할 때는 팔레트의 스피드

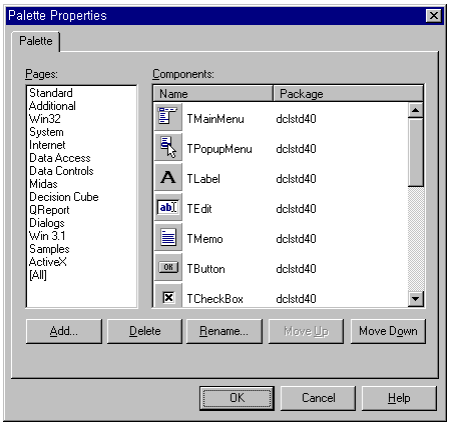
메뉴에서 Properties 항목을 선택하거나 아니면 Option 대화상자에서 Palette 페이지를 선택한다.

**그림**
컴포넌트 팔레트의 스피드 메뉴




다음과 같은 대화상자가 나타난다.

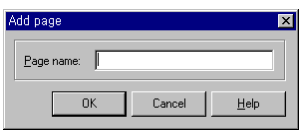
**그림**
팔레트 설정 대화상자




좌측에 Pages 리스트 박스에는 팔레트의 페이지 목록이 나열되어 있으며 우측의 Components 리스트 박스에는 팔레트에 포함된 컴포넌트의 목록이 있고 아래쪽에는 몇 가지 명령 버튼이 위치하고 있다.

## ■ 페이지의 추가 및 삭제

기본적으로 나열된 페이지는 현재 설치된 페이지의 목록이다. 추가로 페이지를 만들고자 할 때는 Add 버튼을 사용한다. 다음과 같이 추가할 페이지의 이름을 묻는다.




페이지를 삭제하려면 삭제할 페이지를 선택한 후 Delete 버튼을 누르되 단

컴포넌트가 포함된 페이지는 삭제할 수 없다. 포함된 컴포넌트를 다른 페이지로 옮기거나 삭제한 후 페이지를 삭제해야 한다. 페이지의 이름을 변경하려면 Rename 버튼을 사용한다.

### ■ 컴포넌트의 이동

다른 페이지로 컴포넌트를 이동시키려면 컴포넌트 목록의 컴포넌트를 드래그하여 원하는 페이지에 떨어뜨리기만 하면 된다. 예를 들어 Standard 페이지의 MainMenu 컴포넌트를 Additional 페이지로 이동하려면 MainMenu 컴포넌트를 드래그하여 Additional 페이지에 떨어뜨린다.

### ■ 순서 변경

Move Up, Move Down버튼은 컴포넌트나 페이지의 순서를 변경할 때 사용한다. 원하는 페이지나 컴포넌트를 선택한 후 Move Up, Move Down 버튼을 누르면 아래 위로 이동할 것이다. 또는 직접 드래그를 해도 된다. 그런데 델파이 4.0의 초기 버전에는 이 기능에 약간의 버그가 있으며 4.01 패치에서 수정되었다.

### ■ 리셋하기

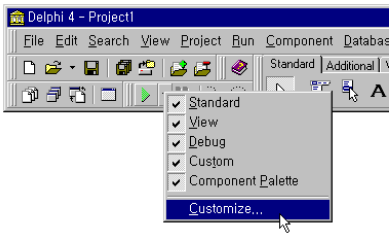
라이브러리에 정의된 배열 상태로 컴포넌트를 재배치하려면 Pages 리스트 박스의 제일 아래에 있는 [All]을 선택한 후 Default Pages 버튼을 누르면 된다.

## 라. 툴바 설정

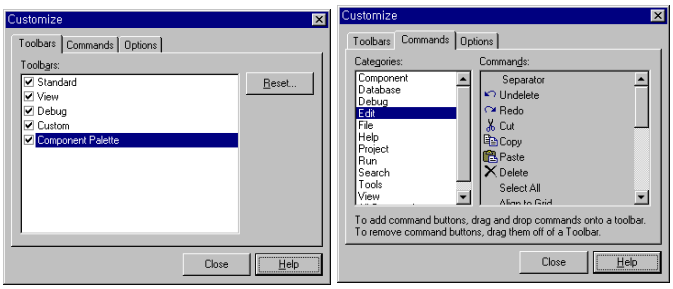
델파이가 제공하는 스피드 버튼은 원터치로 명령을 내릴 수 있기 때문에 대단히 편리하다. 기본적으로 16개의 스피드 버튼을 제공하는데 사용자가 필요에 따라 버튼을 추가하거나 삭제할 수 있도록 되어 있다. 새로운 스피드 버튼을 추가하려면 스피드 버튼의 스피드 메뉴를 사용한다. 스피드 버튼 위에 마우스 커서를 두고 오른쪽 버튼을 눌러 스피드 메뉴가 나타나도록 한다.

**그림**

스피드 버튼의 스피드 메뉴



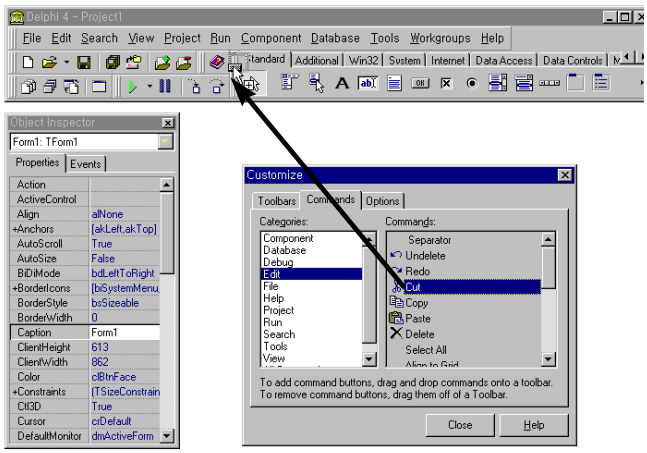

스피드 메뉴의 Customize 항목을 누르면 스피드 버튼을 최적화할 수 있도록 해 주는 대화상자가 나타난다. 이 대화상자는 세 개의 페이지로 구성되어 있다.



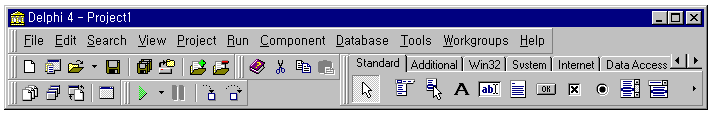

첫 번째 페이지에서는 다섯 가지 툴바의 숨김/보임을 선택하며 두 번째 페이지에는 툴바에 추가할 수 있는 명령의 목록들이 나열되어 있다. 좌측에 메뉴 이름을 담고 있는 리스트 박스가 있고 우측에는 메뉴에 있는 명령어들의 목록이 담겨 있는 리스트 박스가 있다. 우측의 명령 리스트 박스에서 스피드 버튼에 배치하고자 하는 명령을 선택하여 스피드 버튼을 드래그하면 해당 명령에 대한 스피드 버튼이 생성된다.

**그림**

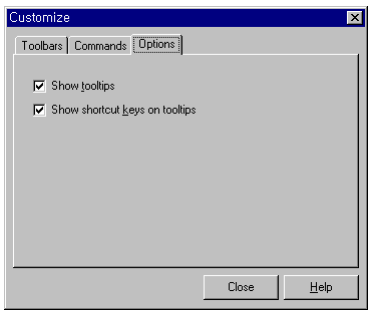
원하는 메뉴 항목을 드래그하여 스피드 버튼으로 등록한다.



디폴트로 설정된 스피드 버튼에 Cut, Copy, Paste 명령이 빠져 있는데 이 항목들을 스피드 버튼에 배치해 놓고 사용하면 코드를 편집할 때 무척 편리하다. 또한 New Application 명령에 대한 툴 버튼이 없어 항상 불편했었는데 이 명령도 툴바에 추가해 보았다. 몇 개의 스피드 버튼을 추가로 배치하고 난 후의 모습은 다음과 같다.

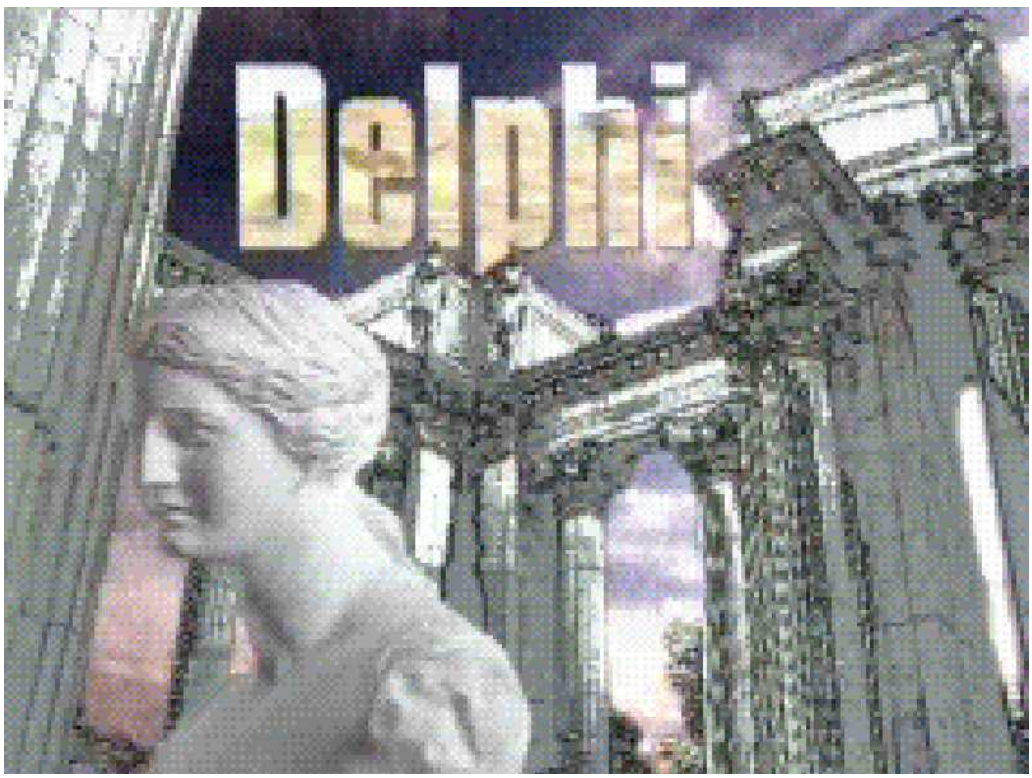



스피드 버튼을 제거하려면 대화상자가 열려 있는 상태에서 스피드 버튼을 드래그하여 패널 밖으로 버리면 된다. 첫 번째 페이지의 Reset 버튼은 델파이를 처음 설치했을 때의 스피드 버튼 상태로 만든다. 마지막 세 번째 페이지에는 툴 팁에 대한 두 가지 옵션이 있다.



# 제 18 장

# 팁 모음



이 장에는 델파이의 각종 팁을 모아 두었다. 팁이란 일종의 토막 지식이며 알아도 그만 몰라도 그만이거나 체계적인 문법에 다소 벗어난 내용을 의미하지만 이 장에 수록된 내용은 그런 의미보다는 좀 더 고급적인 기법들을 담고 있다. 본문에서 설명하기에는 내용이 너무 어려워지는 것 같다거나 책을 읽는 순서상 따로 공부하는 것이 더 효율적이라고 판단되는 내용을 모았다. 어떤 내용이 있는지 미리 살펴 본 후 필요할 때 찾아보는 것이 좋은 학습 방법이라 생각한다.

# 1. 사용자 정의 커서

18jang
cuscur

마우스의 현재 위치를 나타내는 커서의 모양에는 여러 가지가 있으며 화면 영역별로 다른 커서를 사용할 수 있다. 특정 컴포넌트 위에 나타날 커서는 컴포넌트의 Cursor 속성을 사용하여 지정한다. 예를 들어 버튼의 커서를 십자 모양으로 변경하려면 다음과 같이 코드를 작성한다.

```
Button1.Cursor:=crCross;
```

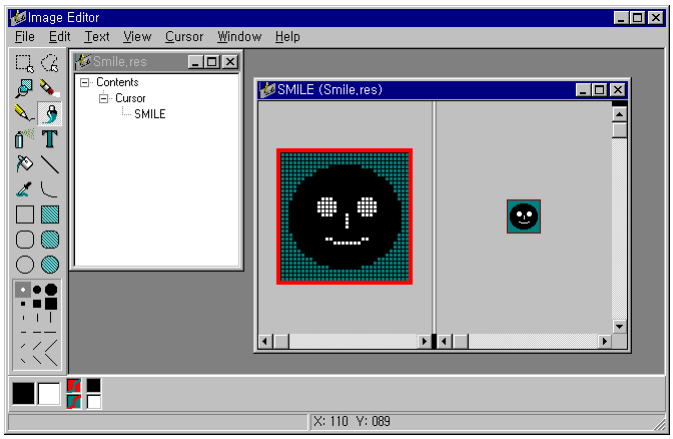
또는 디자인시에 오브젝트 인스펙터를 통해서도 Cursor 속성을 디자인할 수 있다. crCross라는 값의 의미는 Screen 컴포넌트의 Cursors 배열에 저장된 커서의 인덱스, 즉 커서의 번호이며 상수로 -3의 값을 가지는 정수값이다. Screen 컴포넌트는 델파이 프로그램에서 사용 가능한 커서의 목록을 Cursors라는 배열에 저장하며 이 배열에 자신이 만든 커서를 등록함으로써 사용자 정의 커서를 만든다. 다음 단계를 따라 직접 커서를 디자인해 보자.

Screen.Cursors 배열 이름 끝에 s가 있음을 주의해야 한다.

1 새로운 프로젝트를 시작하고 일단 저장부터 한다. 일정한 디렉토리에 배치되어야 리소스 파일을 같은 디렉토리에 저장할 수 있기 때문이다. 프로젝트 이름을 CUSCUR.DPR 로 저장하고 폼을 CUSCUR_F.PAS 로 저장한다.

2 커서를 담은 리소스를 제작한다. 이미지 에디터를 열어 RES 파일을 만들어 프로젝트와 같은 디렉토리에 보관해 두어야 커서가 실행 파일에 같이 포함된다.

그림
사용자 정의 커서 제작




RES 파일을 SMILE.RES 라는 이름으로 저장하고 이 리소스 파일 안에 SMILE 이라는 이름으로 커서를 만들어 둔다. 커서를 어떻게 디자인 할 것인가는 물론 사용자 마음대로이며 사용자의 능력껏 예쁜 커서를 만들도록 한다. 이 예제에서 사용하는 커서는 사람의 웃는 얼굴이다.

3 이제 소스를 작성하여 커서를 실행 파일에서 읽어들이도록 한다. 먼저 커서의 번호로 사용될 상수를 정의한다. 물론 상수를 정의하지 않고 곧바로 정수를 사용할 수도 있지만 커서를 통일적으로 관리하기 위해 표준 커서 인덱스 형식의 상수를 정의하는 것이 좋다. 다음과 같이 crSmile 이라는 상수를 정수 1 로 정의한다.

```
var
  Form1: TForm1;
const
  crSmile=1;
```

표준 커서는 모두 0 이하의 값을 가지는 음수 번호를 가지므로 사용자가 정의하는 커서는 양수여야 한다. 두 개 이상의 커서를 만든다면 2 번, 3 번 상수도 정의해 주어야 한다.

4 리소스 파일로부터 커서를 읽어들인다. 먼저 실행 파일에 같이 포함될 리소스 파일을 {$R SMILE.RES} 지시문으로 지정해 준다.

```
implementation

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

읽어들이는 위치는 폼이 처음 시작될 때인 FormCreate 이벤트가 적당하며 LoadCursor API 함수를 사용한다.

```
Screen.Cursors [crSmile] := LoadCursor (HInstance, 'SMILE');
```

SMILE이라는 커서를 리소스 파일에서 읽어들여 커서 핸들을 Screen 컴포넌트의 Cursors 배열의 번호 1(crSmile)에 대입한다.

5 커서를 사용할 컴포넌트의 Cursor 속성에 방금 읽어들인 커서의 번호를 대입한다. 여기서는 폼이 사용자 정의 커서를 사용하도록 하였다.

```
Form1.Cursor := crSmile;
```

이렇게 코드를 작성하고 프로그램을 실행시킨다. 폼 영역에 마우스가 위치하면 사용자가 정의한 커서가 나타난다. 참고로 CD-ROM의 ₩IMAGES ₩CURORS 디렉토리에 델파이가 제공하는 몇 가지 사용자 정의형 커서가 있으므로 이 커서를 직접 사용해도 된다.

# 2. 커서 애니메이션

위에서 살펴 봤다시피 사용자 정의 커서를 만드는 방법은 그리 어렵지 않다. 사용자 정의 커서를 활용하면 시시각각으로 모양이 변하는 애니메이션 커서를 만드는 것도 가능하다. 윈 95 에서는 커서가 움직이는 것이 너무나도 당연해 보이겠지만 윈도우즈 3.1 에서는 움직이는 커서를 사용하는 프로그램이 그리 많지 않았다. 그 중 대표적인 프로그램으로 코렐 드로우를 꼽을 수 있는데 장시간을 요하는 작업을 할 때 회전하는 모래 시계를 보여줌으로써 기다리는 사람을 덜 지루하게 해 주었다.

18jang
curani

여기서는 코렐 드로우를 흉내내어 움직이는 커서를 만들어 보자. 원리는 아주 간단하다. 각 애니메이션 컷에 해당하는 커서를 리소스로 정의해 두고 작업을 하는 동안 일정 시간 간격으로 계속 커서를 바꾸어 주는 것이다. 커서를 디자인하는 것이 어려울 뿐이지 애니메이션 자체는 무척 간단하다. 이 예제에서 사용하는 커서는 다음 10 가지이다.

이런 식으로 순서대로 변하는 그림 10 장을 리소스 파일에 커서로 작성해 놓고 코드에서 불러들인다. 그리고 커서를 계속 교체하는 것이다. 전체 소스는 다음과 같다.

```
{커서 애니메이션 예제
인터넷에서 구한 소스중 불필요한 부분을 제거하고 꼭 필요한
부분만 간략하게 정리하였다}
unit curani_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
{$R hourgzpd.res}

procedure TForm1.FormCreate(Sender: TObject);
begin
  {회전하는 모래시계 커서를 읽어들인다}
  screen.cursors[1] := LoadCursor(hInstance, pChar('CURSOR_1'));
  screen.cursors[2] := LoadCursor(hInstance, pChar('CURSOR_2'));
  screen.cursors[3] := LoadCursor(hInstance, pChar('CURSOR_3'));
  screen.cursors[4] := LoadCursor(hInstance, pChar('CURSOR_4'));
  screen.cursors[5] := LoadCursor(hInstance, pChar('CURSOR_5'));
  screen.cursors[6] := LoadCursor(hInstance, pChar('CURSOR_6'));
  screen.cursors[7] := LoadCursor(hInstance, pChar('CURSOR_7'));
```

```
  screen.cursors[8] := LoadCursor(hInstance, pChar('CURSOR_8'));
  screen.cursors[9] := LoadCursor(hInstance, pChar('CURSOR_9'));
  screen.cursors[10]:= LoadCursor(hInstance, pChar('CURSOR_10'));
end;

{버튼을 누르면 애니메이션 시작}
procedure TForm1.Button1Click(Sender: TObject);
var
  HGstate:integer;
  i,j,k:longint;
begin
  HGstate:=1;
  for i:= 1 to 50 do
  begin
    inc(HGstate);  {커서 숫자를 증가시킴}
    {10 번째 커서 이후 다시 첫 번째로 돌아온다}
    if HGstate>10 then HGstate:=1;
    screen.cursor:=HGstate;
    for j:= 1 to 2000000 do
    begin
      {이 위치에 작업할 코드를 작성한다.}
      k:=k+1;
      k:=k-1;
    end;
  end;
  {애니메이션이 끝난 후 기본 커서로 변경함}
  screen.cursor:=crDefault;
  Application.processMessages;
end;

end.
```

컴파일러 지시자 {$R}을 사용하여 hourgzpd.res 리소스 파일을 읽어들이며 사용자 정의 커서 10 개를 정의한다. 버튼을 누르면 모종의 작업을 하되 작업 중간중간에 Screen.cursor 를 변경해 주며 작업이 끝나면 커서 모양을 다시 crDefault 로 돌려 준다.

# 3. 그래픽 리스트 박스

18jang
Owner

리스트 박스에는 문자열의 목록이 들어가는 것이 보통이지만 때로는 그래픽

이 리스트 박스에 포함되기도 한다. 그래픽을 포함하는 리스트 박스를 Owner-Draw 리스트 박스라고 하며 이 말은 리스트 박스의 소유주(Owner)인 폼이 직접 리스트 박스 안의 내용물을 그린다(Draw)는 뜻이다. 포함되는 그래픽의 높이가 일정한가 가변적인가에 따라 두 가지 형태가 있다.

## 고정 높이 리스트 박스

리스트 박스에 포함되는 항목들이 모두 같은 높이를 가진다. 제작 과정은 생략하고 배포 CD의 예제를 분석해 보자. 실행중의 모습은 다음과 같다.

**그림**

그래픽 목록을 담고 있는 리스트 박스


비트맵 파일 여섯 개를 준비한 후 이 비트맵들을 리스트 박스 안에 넣어 두었다. 오너 드로우 리스트 박스를 만들려면 일단 리스트 박스의 Style 속성을 lbOwnerDrawFixed로 변경하여 항목을 직접 그리되 각 항목의 높이가 모두 같다는 것을 알려주며 항목의 높이는 ItemHeight 속성으로 설정한다.

리스트 박스에 그릴 비트맵은 FormCreate 이벤트에서 읽어들여 리스트 박스에 추가한다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
 i:integer;
begin
for i:=1 to 6 do
 begin
 bitmaps[i]:=TBitmap.Create;
 bitmaps[i].LoadFromFile('bmp'+IntToStr(i)+'.bmp');
 ListBox1.Items.AddObject('nomean',bitmaps[i]);
 end;
end;
```

이 예제에서는 Bmp1.bmp~Bmp6.bmp까지 여섯 개의 비트맵을 미리 만들어 두고 이 비트맵을 리스트 박스로 출력한다. AddObject 메소드는 리스트 박스에 문자열 항목과 오브젝트를 동시에 추가한다. 이 예제에서 문자열은 사용하지 않는다.

이렇게 읽혀진 비트맵은 리스트 박스의 OnDrawItem 이벤트에서 그려진다. OnDrawItem 이벤트는 리스트 박스가 처음 나타날 때, 스크롤을 할 때 등 항목이 다시 그려져야 할 경우에 발생한다. 이 이벤트에 리스트 박스로 그래픽을 출력하는 코드를 작성하면 된다.

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;  Rect: TRect; State:
TOwnerDrawState);
var
  Bmp:TBitmap;
  ListBox:TListBox;
begin
ListBox:=Control as TListBox;
bmp:=TBitmap(ListBox.Items.Objects[Index]);
ListBox.Canvas.FillRect(Rect);
ListBox.Canvas.Draw(Rect.Left,Rect.Top+3,bmp);
end;
```

4개의 인수가 전달되는데 Control은 이 이벤트가 발생된 리스트 박스이며 Index는 그려져야 할 항목의 번호, Rect는 항목이 그려질 위치, State는 항목의 현재 상태(선택, 사용 불가능, 포커스 가짐)를 나타낸다. 코드를 살펴보면 Objects 배열에 저장된 비트맵을 읽어 bmp 변수에 대입한 후 Draw 메소드로 리스트 박스에 비트맵을 출력하고 있다.

리스트 박스의 각 항목이 비트맵일 뿐 사용하는 방법은 문자열일 때와 동일하다. 다음 코드는 선택된 항목의 번호를 레이블로 출력한다.

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
label1.caption:=IntToStr(ListBox1.ItemIndex)+'번 항목 선택';
end;
```

## 가변 높이 리스트 박스

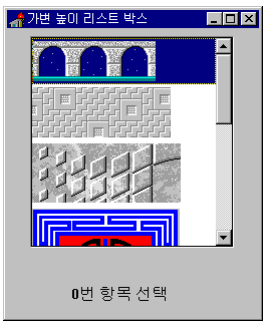
18jang
Owner2

각각의 항목이 서로 다른 높이를 가질 경우는 각 항목의 높이를 계산하는 과정이 추가되며 이 계산은 OnMeasureItem 이벤트에서 수행한다. 이 이벤트는 OnDrawItem 이벤트가 발생하기 직전에 발생하며 계산 결과를 Height 인수로

전달해 주는 역할을 한다. 이 이벤트에서 넘겨준 Height값이 OnDrawItem이벤트의 Rect 인수로 전달되어 항목이 그려질 영역 계산에 사용된다. 코드는 다음과 같다.

**그림**

가변 높이의 그래픽을 담는 리스트 박스




```
procedure TForm1.ListBox1MeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);
begin
Height:=bitmaps[index+1].height+6;
end;
```

리스트 박스의 항목이 bitmaps 배열의 비트맵이므로 이 배열에 저장된 비트맵의 높이를 읽어 Height 인수에 대입하기만 하면 된다. index에 1을 더하는 의미는 리스트 박스의 항목 번호는 0부터 시작하고 bitmaps 배열의 첨자는 1부터 시작하기 때문이다. 높이에 더해진 6은 쉽게 말해서 각 항목 사이에 삽입될 줄간에 해당한다.

# 4. 포커스 잃으면 프로그램 끝내기

18jang
killapp

델파이 프로그램은 TApplication 컴포넌트를 사용하여 Application이라는 변수를 만든다. 이 컴포넌트가 곧 프로그램 그 자체를 나타낸다. 비록 TApplication 컴포넌트는 컴포넌트 팔레트에는 없지만 다른 컴포넌트와 마찬가지로 속성, 이벤트, 메소드를 가진다.

Application 변수의 속성을 사용하면 아이콘, 메인폼, 도움말 파일, 실행 파일 이름 등 프로그램 자체에 관한 여러 가지 정보를 얻을 수 있다. TApplication 컴포넌트의 메소드는 Minimize, Restore 등 프로그램 전체에 영향을 미치며 이 메소드들은 사용자가 실행중에 폼을 조작하면(예를 들어 타이틀 바를 더블클릭

할 때) 자동으로 호출된다. 코드에서 이 메소드들을 의식적으로 사용하면 사용자의 동작없이도 특정 동작을 취할 수 있게 된다. 예를 들어 최소화되어 있다가 일정 시간이 경과하면 자동으로 윈도우가 열려진다거나 활성화되면 무조건 특정 작업을 한다거나 할 수 있다.

TApplication 컴포넌트의 이벤트를 사용하면 프로그램 전체에 대해 발생하는 이벤트를 처리할 수 있다. OnActivate, OnDeactivate, OnIdle 등의 이벤트가 있는데 여기서는 OnDeactivate 이벤트를 사용해서 예제를 작성해 보았다. 실행중에 프로그램이 비활성화되면 즉, 포커스를 잃으면 프로그램을 종료하고자 한다. 프로그램이 비활성화되면 Application 컴포넌트에 OnDeactivate 이벤트가 전달되므로 이 이벤트 핸들러에서 프로그램을 종료하도록 하면 된다. 그런데 Application 컴포넌트는 팔레트에도 존재하지 않으며 오브젝트 인스펙터에서도 지정할 수 없기 때문에 디자인시에 이벤트 핸들러를 지정할 수 없다. 그래서 폼이 처음 생성될 때, FormCreate 이벤트 핸들러에서 Application.OnDeactivate 이벤트의 핸들러를 정의해 주어야 한다.

전체 소스는 다음과 같다.

```
unit Killappf;

interface

uses
 SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
 Forms, Dialogs, StdCtrls;

type
 TForm1 = class(TForm)
  Memo1: TMemo;
  procedure KillApp(Sender:TObject);
  procedure FormCreate(Sender: TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation
```

```
{$R *.DFM}

procedure TForm1.KillApp(Sender:TObject);
begin
MessageBeep(0);
Application.Terminate;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
{Application.OnDeactivate:=KillApp;}
end;

end.
```

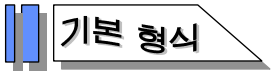
KillApp라는 프로시저를 미리 만들어 두고 프로그램이 비활성화될 때 이 메소드를 호출하도록 이벤트 핸들러로 지정해 놓았다. Killapp 프로시저에서는 Terminate 메소드를 호출하여 프로그램을 종료한다.

# 5. 윈도우 메시지 이용하기

18jang
nchit

윈도우즈는 메시지 구동 시스템이며 델파이 프로그램도 물론 메시지에 의해 운영된다. 그러나 델파이는 메시지를 직접 다루지 않고 이벤트로 바꾸어 취급하므로 사용자가 메시지를 직접 프로그래밍해야 할 경우는 드물다. 하지만 메시지를 직접 프로그래밍할 경우는 이벤트 처리로 구현할 수 없는 색다른 효과를 낼 수 있다.

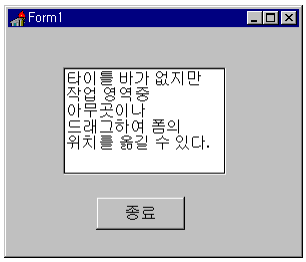
델파이는 표준 윈도우즈 메시지 처리를 위해 메시지 핸들러를 정의하고 핸들러가 메시지를 처리한다. 델파이가 디폴트로 제공하는 메시지 핸들러는 보통 메시지를 무시하며 아무 일도 하지 않도록 되어 있지만 필요할 경우 디폴트 핸들러를 사용자가 정의한 함수로 바꾸어 메시지에 따른 특별한 처리를 할 수 있다. 메시지 핸들러는 procedure 선언 뒤에 message 지시자로 지정한다.




procedure 이름(var 메시지 레코드); message 메시지;

메시지는 메시지 레코드로 전달되며 메시지 유형별로 레코드 타입이 정의되어 있다. 메시지 핸들러는 메시지 레코드를 참조 호출로 전달받아 전달된 메시지의 구체적인 내용을 파악하고 메시지 처리의 결과를 이 인수로 리턴해 준다.

예제를 작성해 보자. 다음과 같이 폼을 만들고 폼의 BorderStyle을 bsNone으로 지정하여 타이틀 바를 없애버린다.




전체 코드는 다음과 같다.

```
unit Nchit_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    procedure WMNCHitTest(var M:TWMNCHitTest) ; message WM_NCHitTest;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.DFM}

procedure TForm1.WMNCHitTest(var M:TWMNCHitTest);
begin
inherited;
if M.Result=htClient then
M.Result:=htCaption;
end;

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

end.
```

WMNCHitTest 프로시저가 WM_NCHITTEST라는 윈도우즈 메시지를 처리하는 메시지 핸들러로 지정되어 있다. WM_NCHITTEST 메시지는 커서가 있는 윈도우(또는 SetCapture에 의해 커서를 잡은 윈도우)의 커서 위치를 커서가 움직일 때마다 조사해 준다. 조사한 결과는 TWMNCHitTest 메시지 레코드의 Result 필드로 전달된다.

```
type
  TWMNCHitTest = record
    Msg: Cardinal;
    Unused: Cardinal;
    case Integer of
      0: (
        XPos: SmallInt;
        YPos: SmallInt);
      1: (
        Pos: TSmallPoint;
        Result: Longint);
  end;
```

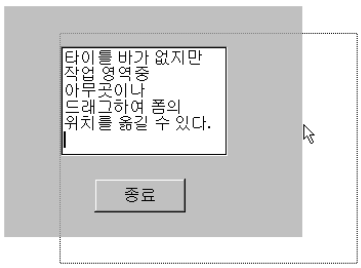
Result 필드로 전달되는 값의 종류는 다음과 같다. 종류가 무척 많으므로 대표적인 몇 개만 보인다. 나머지는 도움말을 참조하기 바란다.

| 값 | 의미 |
|---|---|

| | |
|---|---|
| HTBORDER | 윈도우의 경계선 |
| HTBOTTOM | 아래쪽 경계선 |
| HTLEFT | 왼쪽 경계선 |
| HTRIGHT | 오른쪽 경계선 |
| HTTOP | 윗쪽 경계선 |
| HTMENU | 메뉴 영역 |
| MTCAPTION | 타이틀 바 |
| HTCLIENT | 작업 영역 |
| HTBOTTOMRIGHT | 아래 오른쪽 경계선 |

메시지 핸들러인 WMNCHitTest 프로시저를 보자. 일단 inherited, 즉 디폴트 메시지 핸들러를 호출하여 메시지 고유의 처리를 하도록 한다. 디폴트 핸들러 호출 후에 Result값은 현재 마우스 커서가 위치하고 있는 영역이 저장되어 있을 것이다. 바로 아래 조건문에서는 Result값을 조사하여 만약 현재 커서가 작업 영역에 있다면 이 값을 바꾸어 타이틀 바에 있는 것처럼 만든다. 즉 윈도우즈를 살짝 속이는 것이다. 그래서 이 예제를 실행시키고 작업 영역의 아무 곳이나 클릭하여 드래그하면 마치 타이틀 바를 드래그하는 것처럼 윈도우가 이동하는 것을 볼 수 있을 것이다. 이 예제처럼 타이틀 바가 불필요한 프로그램의 경우는 타이틀 바를 없애는 대신 작업 영역을 드래그하여 위치를 옮길 수 있도록 해 준다. 윈도우즈의 표준 애플릿인 시계가 이런 방법을 사용하는데 이 메시지를 사용하는 것이다.

WMNCHitTest 메소드의 마지막 행을 M.Result:=HtBottomRight로 바꾸면 작업 영역을 드래그하여 윈도우의 크기를 바꿀 수도 있다.



# 6. 한번만 실행되는 프로그램

18jang
once

윈도우즈는 여러 개의 프로그램을 동시에 실행할 수 있을 뿐만 아니라 같은 프로그램을 두 번 실행시킬 수도 있다. 같은 프로그램이라도 두 번 실행되면 다른 인스턴스로 인식되므로 별다른 문제가 없다. 그러나 때에 따라서는 한 프로그램이 두 번 실행되는 것이 바람직하지 못한 경우가 있다. 제어판이나 파일 관리자 또는 쉘 프로그램은 동시에 두 번 실행될 경우 복잡한 문제를 일으킬 수도 있으며 특별히 두 번 실행할 이유도 없다. 원칙적으로 두 번 실행이 가능한 윈도우즈 환경에서 두 번 실행을 금지시키는 것은 약간 복잡한 과정을 거쳐야 한다.

윈도우즈에서 두 번 실행을 금지시키는 방법은 없다. 다만 프로그램이 실행될 때 같은 프로그램이 실행중인가를 살펴보고 이미 실행중이면 종료하는 방법을 사용하여 두 번 실행되지 않도록 할 수 있을 뿐이다. 델파이로 만든 프로그램은 제일 먼저 프로젝트가 제어권을 가지므로, 즉 프로젝트 파일이 엔트리 포인트(entry point)가 된다. 그래서 프로그램이 시작되는 첫 지점인 프로젝트 파일(DPR 파일)을 직접 수정해야만 두 번 실행을 금지시킬 수 있다. 사용자가 DPR 파일을 직접 편집해야 할 일은 극히 드문 일이지만 이 경우는 프로그램 시작의 첫 지점에서 조건 점검을 해야 하므로 DPR 파일을 편집할 수밖에 없다. 다음 리스트에서 굵은 문자로 인쇄된 부분이 사용자가 수정한 부분이다.

```
program Once;

uses
  Windows,
  Forms,
  Once_f in 'Once_f.pas' {Form1};

{$R *.RES}
var
  hPrevWnd:THandle;
begin
  hPrevWnd:=FindWindow(Nil, 'OnceForm');
  if hPrevWnd <> 0 then
    begin
      SetForegroundWindow(hPrevWnd);
    end
  else
    begin
```

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

FindWindow 함수는 첫 번째 인수로 전달된 윈도우 클래스나 두 번째로 전달된 윈도우 타이틀이 같은 윈도우를 찾아 그 핸들을 리턴해 준다. 두 가지 문자열 중 어떤 것이라도 사용할 수 있으나 델파이로 만든 프로그램은 윈도우 클래스가 모두 같으므로 윈도우 클래스는 쓸 수 없고 윈도우의 타이틀을 사용해야 한다. 그래서 이 예제는 폼의 Caption 속성을 OnceForm으로 설정해 두고 이런 캡션을 가진 윈도우가 있는지를 찾도록 하였다.

FindWindow가 OnceForm이라는 타이틀을 가진 윈도우를 발견하면 이 프로그램이 이미 실행되고 있다고 판단한다. 그러면 더 이상 프로그램을 실행시킬 필요없이 이미 실행중인 Once.exe에게 포커스를 넘겨주고 그냥 종료한다. 만약 윈도우가 발견되지 않으면 Application.Initialize 이하의 과정을 거쳐 정상적으로 프로그램을 실행하면 된다.

이 방법은 대충은 동작하지만 사실 완벽한 방법은 아니다. 왜냐하면 OnceForm이라는 타이틀을 가진 윈도우가 우연으로 존재할 가능성도 있으며 또한 이 방법을 사용하면 폼의 캡션을 실행중에 함부로 변경할 수도 없게된다. 좀 더 확실한 방법은 윈도우 클래스명을 특이한 것으로 만들어 윈도우 클래스명으로 실행중인 인스턴스를 찾는 방법이 있으나 이 방법도 역시 제대로 된 해결 방법이라고는 할 수 없다.

완벽한 해결 방법으로는 사용자 정의 메시지를 보내는 방법이 있다. 즉 프로그램이 실행될 때 일단 타이틀이 같은 윈도우를 찾은 후 '너 정말로 내가 찾는 윈도우가 맞니'하고 메시지를 보내본다. 이 메시지를 받은 윈도우가 그렇다고 응답하면 조용히 종료하면 된다.

# 7. 폼의 크기를 제한하기

18jang
fsize

일반적으로 윈도우의 크기나 위치는 사용자가 마음대로 변경할 수 있다. 그러나 특수한 경우에 있어서는 사용자가 마음대로 윈도우의 크기를 변경할 수 없도록 일정한 제약을 가할 필요가 있다. 그 예는 멀리서 찾을 것도 없이 델파이의 메

인 윈도우를 보면 된다. 사용자가 수평 크기를 마음대로 변경할 수는 있지만 수직 높이는 항상 고정되어 있으며 최대화시켰을 때의 크기도 화면 전체를 덮지 않고 화면 위쪽으로 메인 윈도우가 배치되기만 한다.

자유롭게 변경할 수 있는 윈도우의 크기를 일정하게 제한하고자 할 때는 윈도우즈가 보내주는 WM_GETMINMAXINFO 메시지를 사용한다. 이 메시지는 윈도우의 크기나 위치가 변할 때 발생하며 이 메시지를 이용하면 디폴트 최대화 크기와 위치, 최소 크기를 변경할 수 있다. 이 메시지를 가로채려면 다음과 같이 폼의 타입 선언의 private 부에 메시지 처리 프로시저를 등록한다.

```
type
 TForm1 = class(TForm)
  Memo1: TMemo;
 private
  procedure WMGetMinMax(var Message:TWMGetMinMaxInfo);
    message WM_GETMINMAXINFO;
  { Private declarations }
 public
  { Public declarations }
 end;
```

그리고 implemetation 부에 다음과 같이 메시지 처리 프로시저의 코드를 작성한다.

```
procedure TForm1.WMGetMinMax(var Message:TWMGetMinMaxInfo);
var
 MinMax:PMinMaxInfo;
begin
inherited;
MinMax:=Message.MinMaxInfo;
MinMax^.ptMaxSize.X:=630;
MinMax^.ptMaxSize.Y:=470;

MinMax^.ptMaxPosition.X:=5;
MinMax^.ptMaxPosition.Y:=5;

MinMax^.ptMaxTrackSize.X:=630;
MinMax^.ptMaxTrackSize.Y:=470;

MinMax^.ptMinTrackSize.X:=250;
MinMax^.ptMinTrackSize.Y:=150;
end;
```

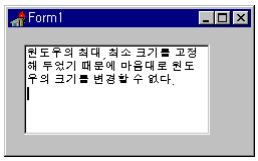
프로시저 선두에서 Inherited 를 호출하여 고유의 처리를 할 수 있도록 해 줌을 유의하도록 하자. 이 메시지가 발생할 때 다음과 같이 선언된 MINMAXINFO 라는 구조체가 인수로 전달된다.

```
typedef struct tagMINMAXINFO { // mmi
  POINT ptReserved;
  POINT ptMaxSize;
  POINT ptMaxPosition;
  POINT ptMinTrackSize;
  POINT ptMaxTrackSize;
} MINMAXINFO;
```

이 구조체의 각 멤버에 원하는 값을 대입해 주면 윈도우의 크기가 대입해 준 값에 따라 일정하게 제한된다. 각 멤버의 의미는 다음 표와 같다.

| 멤버 | 의미 |
|---|---|
| ptReserved | 예약되어 있으며 현재는 사용하지 않는다. |
| ptMaxSize | 최대화되었을 때의 윈도우 크기를 설정한다. |
| ptMaxPosition | 최대화되었을 때의 윈도우 위치를 설정한다. |
| ptMinTrackSize | 사용자가 경계선을 드래그하여 윈도우의 크기를 조정할 때의 최소 크기이다. |
| ptMaxTrackSize | 사용자가 경계선을 드래그하여 윈도우의 크기를 조정할 때의 최대 크기이다. |

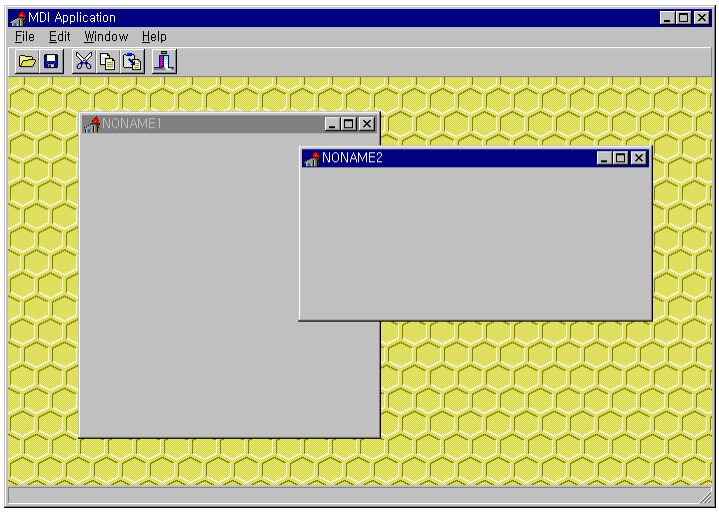
각 멤버는 모두 POINT 형이므로 X,Y 에 적절한 값을 대입해 주도록 한다. 위 예제의 경우 최대화 크기는 630*470 이며 위치는 5,5 로 설정되어 있으므로 타이틀 바를 더블클릭해도 화면 전체 크기만큼 커지지 않는다. 그리고 사용자가 경계선을 드래그하여 윈도우 크기를 조정할 때 최대 630*470 까지 커질 수 있으며 최소 250*150 까지 작게 만들 수 있다. 이 범위를 벗어날 경우는 드래그해도 윈도우의 크기가 변경되지 않는다. 직접 예제를 실행해 보면 무슨 의미인지 쉽게 알 수 있을 것이다.



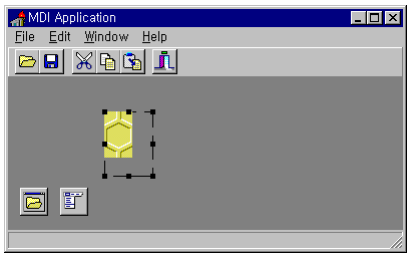

# 8. MDI 폼에 배경 비트맵 넣기

18jang
mdibk

윈도우즈에는 벽지 기능이 있어서 데스크탑의 배경에 원하는 그림을 배치해 놓을 수 있다. 이와 비슷하게 MDI 프로그램의 메인 윈도우 배경에 배경 그림을 그려 넣는 것도 가능하다. 다음이 MDI 메인 윈도우의 배경에 비트맵을 Tile 배치한 것이다.




MDI 예제를 일일이 만들려면 너무 손이 많이 가므로 오브젝트 창고에서 MDI 프로그램을 불러온 후 변경을 가하는 방법을 사용하는 것이 더 좋을 것 같다. MDI 템플릿을 만든 후 제일 먼저 할 일은 메인 폼의 빈 곳에 이미지 컴포넌트를 배치하고 이 컴포넌트에 배경으로 사용할 비트맵을 미리 읽어 놓는 것이다. 이 비트맵이 MDI 메인 폼의 배경에 뿌려질 비트맵이다. 물론 실행중에 비트맵을 변경하는 것도 가능하다.




메인 폼의 타입 선언부에 다음과 같이 변수와 프로시저를 추가한다.

```
private
  { Private declarations }
  FClientInstance:TFarProc;
  FPrevClientProc:TFarProc;
  procedure ClientWndProc(VAR Message:TMessage);
  procedure CreateMDIChild(const Name: string);
  procedure ShowHint(Sender: TObject);
```

그리고 FormCreate 이벤트 핸들러에 다음 코드를 추가한다.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
 Application.OnHint := ShowHint;
 Screen.OnActiveFormChange := UpdateMenuItems;
 FClientInstance:=MakeObjectInstance(ClientWndProc);
 FPrevClientProc:=Pointer(GetWindowLong(ClientHandle,
   GWL_WNDPROC));
 SetWindowLong(ClientHandle,GWL_WNDPROC,
   LongInt(FClientInstance));
end;
```

MakeObjectInstance 함수는 윈도우 프로시저 메소드를 호출해 주는 윈도우 프로시저 함수(function)의 포인터를 리턴해 준다. 윈도우즈는 윈도우 프로시저(window procedure)라 불리우는 함수로 메시지를 전달해 준다. 델파이 컴포넌트는 메시지 처리를 위한 윈도우 프로시저로 메소드(method)를 사용하기 때문에 윈도우즈가 메소드의 포인터를 윈도우 프로시저로 사용할 수 없다. 그래서 MakeObjectInstance 함수로 윈도우즈가 호출할 수 있는 함수를 만들어 주고 그 함수에서 지정한 메소드를 호출하도록 해야 한다. 이런 목적으로 사용되는 변수가 FClientInstance 포인터 변수이며 이 포인터에 ClientWndProc 프로시저를 호출해 주는 함수의 번지가 대입된다. 그리고 SetWindowLong 함수에 의해 이 포인터가 가리키는 ClientWndProc 프로시저가 새로운 윈도우 프로시저

로 지정된다.

```
procedure TMainForm.ClientWndProc(VAR Message: TMessage);
var
 MyDC : hDC;
 Ro, Co : Word;
begin
 with Message do
 case Msg of
  WM_ERASEBKGND:
  begin
   MyDC := TWMEraseBkGnd(Message).DC;
   {비트맵을 배경에 Tile 배치한다.}
   for Ro := 0 TO ClientHeight DIV Image1.Picture.Height DO
   for Co := 0 TO ClientWIDTH DIV Image1.Picture.Width DO
    BitBlt(MyDC, Co*Image1.Picture.Width,

    Ro*Image1.Picture.Height,
    Image1.Picture.Width, Image1.Picture.Height,
    Image1.Picture.Bitmap.Canvas.Handle, 0, 0, SRCCOPY);
    Result := 1;
  end;
  {그 외의 메시지는 원래의 윈도우 프로시저로 처리를 넘긴다.}
  else
   Result := CallWindowProc(FPrevClientProc, ClientHandle, Msg, wParam, lParam);
 end;
end;
```

이 프로시저에서는 모든 처리를 FPrevClientProc, 즉 원래의 윈도우 프로시저로 넘기되 WM_ERASEBKGND 메시지만 가로채서 배경색을 단순히 지우지 않고 비트맵으로 채워 준다. 그래서 MDI 메인 폼의 배경이 지워질 때 비트맵이 배경으로 출력되는 것이다.

# 9. 실행중에 메뉴 항목 추가하기

메뉴는 디자인시에 설정한대로 있지만은 않으며 실행중에 속성을 바꾸거나 메뉴 항목을 추가할 수도 있다. 속성을 바꾸는 것은 너무 쉬운 일이므로 언급할 필요도 없지만 메뉴 항목을 추가하는 경우는 조금 아주 쬐끔 어렵다. 델파이의 File 메뉴 하단에는 최근에 열었던 프로젝트의 이름이 항상 나타난다. 이런 식으로 실행중에만 알 수 있는 정보를 메뉴에 나타내는 방법에 대해 알아보자. 크게

두 가지 방법을 생각할 수 있다.

18jang
dymenu1

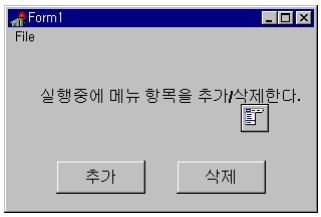
우선 필요한 메뉴를 미리 디자인시에 만들어 두고 당장 사용하지 않을 메뉴 항목의 Visible 속성을 False로 만들어 둔 후 필요할 때 보이도록 하는 방법이다. File 아래에 One, Two, Three, Four 네 개의 메뉴 항목을 만든 후 Four 항목의 Visible 속성을 False로 설정한다. 그리고 두 개의 버튼을 배치한 후 다음과 같이 코드를 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Four1.Visible:=True;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
Four1.Visible:=False;
end;
```

그리고 Four 메뉴 항목을 파일 이름으로 또는 실행시에만 알 수 있는 정보로 변경하면 된다. 이렇게 하면 약간의 눈속임을 사용하여 마치 실행중에 메뉴 항목이 생성된 것처럼 보이기는 하지만 만들 수 있는 개수에 한계가 있다.

18jang
dymenu2

두 번째 방법은 아예 TMenuItem 컴포넌트를 동적으로 생성시키는 것이다. 필요할 때마다 생성시킬 수 있으므로 개수 제한이 없고 삽입되는 위치도 마음대로 정할 수 있다. 폼 디자인은 다음과 같다.




추가 버튼을 누르면 메뉴 항목이 File 메뉴에 추가되며 삭제를 누르면 제일 끝에 추가된 항목이 삭제된다. 코드는 다음과 같다.

```
var
  Form1: TForm1;
  MyItem:array [0..10] of TMenuItem;
  TotalItem:integer;
```

```
implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
if TotalItem=11 then
  begin
  ShowMessage('최고 10 개까지 만들 수 있다');
  Exit;
  end;
MyItem[totalItem]:=TMenuItem.Create(Self);
MyItem[totalItem].Caption:='Item'+IntToStr(TotalItem);
File1.Add(MyItem[totalItem]);
Inc(TotalItem);
label1.Caption:='현재 항목='+IntToStr(TotalItem);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
if TotalItem=0 then
  begin
  ShowMessage('지울 메뉴가 없네요');
  Exit;
  end;
File1.Delete(totalItem+2);
Dec(TotalItem);
label1.Caption:='현재 항목='+IntToStr(TotalItem);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
TotalItem:=0;
end;
```

동적 객체 생성에 관한 내용은 12장을 참고하기 바란다. 메뉴 항목을 실행시에 생성한 후 File1 메뉴 항목의 Add 메소드를 사용하여 메뉴 항목을 추가하며 Delete 메소드를 사용하여 삭제한다. 만약 추가되는 위치를 마음대로 조정하고 싶다면 Add 대신에 Insert 메소드를 사용하면 된다.

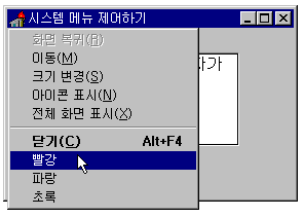
# 10. 시스템 메뉴에 항목 추가하기

18jang
Sysmenu

메뉴는 프로그램의 모든 기능을 모아 놓기에 아주 좋은 장치이지만 화면 크기를 가급적이면 줄여야 하는 프로그램에서는 따로 메뉴를 두기가 거북스러운 경우도 있다. 이럴 때는 팝업 메뉴를 사용하거나 시스템 메뉴에 사용자 항목을 추가하는 방법을 사용한다. 시스템 메뉴는 어느 프로그램에나 있으며 타이틀 바에 위치하므로 추가적인 공간을 요구하지 않는다. 시스템 메뉴는 컴포넌트로 제공되는 것이 아니므로 디자인중에 항목을 편집할 수 없으며 실행중에 코드를 통해서 항목을 삽입할 수 있다. 방법은 앞에서 보인 실행중에 메뉴 항목 추가하기와 동일하다.

**그림**

시스템 메뉴에 추가시킨 사용자 정의 메뉴 항목




메뉴를 추가하는 코드는 프로그램의 시작점인 FormCreate 이벤트에 작성되어 있다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  SysMenu:HMenu;
begin
SysMenu:=GetSystemMenu(Handle,False);
InsertMenu(SysMenu,7,MF_BYPOSITION or MF_STRING,1000,'빨강');
InsertMenu(SysMenu,8,MF_BYPOSITION or MF_STRING,1001,'파랑');
InsertMenu(SysMenu,9,MF_BYPOSITION or MF_STRING,1002,'초록');
Application.OnMessage:=MsgProc;
end;
```

우선 항목 추가의 대상이 되는 시스템 메뉴의 핸들을 GetSystemMenu 함수로 구한다. 이 함수의 첫 번째 인수는 시스템 메뉴를 가지는 윈도우 핸들이다. 여기서는 Handle을 사용하여 폼의 윈도우 핸들만을 사용했지만 최소화할 경우의 시스템 메뉴에도 똑같은 항목을 삽입하려면 Application.Handle에 대해서도 항목을 추가해 주어야 한다. 여기서 구한 메뉴 핸들은 항목 삽입 함수의 인수로 사용된다. 항목 삽입에는 InsertMenu 함수를 사용한다. 두 함수 모두 델파이가

제공하는 함수가 아닌 API 함수이므로 인수의 의미는 API 도움말을 참고하기 바란다.

이렇게 삽입된 메뉴 항목들은 델파이에 의해 만들어진 것이 아니므로 이벤트 핸들러를 가질 수 없다. 그래서 메뉴가 선택될 때 윈도우즈가 보내주는 메시지를 직접 사용하여 코드를 작성해야 한다. 윈도우즈가 응용 프로그램으로 메시지를 보낼 때 OnMessage 이벤트가 발생하므로 이 이벤트 핸들러를 작성하여 코드를 기입하되 이 이벤트도 디자인중에는 지정할 수 없으므로 FormCreate에서 지정한다. 이 예제에서는 Msgproc 프로시저를 사용하고 있다. 폼의 타입 선언에 다음과 같이 프로시저가 선언되어 있다.

```
procedure MsgProc(var Msg:TMsg;var Handled:Boolean);
```

본체는 다음과 같이 작성되어 있다.

```
procedure TForm1.MsgProc(var Msg:TMsg;var Handled:Boolean);
begin
if Msg.message=WM_SYSCOMMAND then
 case Msg.Wparam of
  1000:Color:=clRed;
  1001:Color:=clBlue;
  1002:Color:=clGreen;
 end;
end;
```

WM_SYSCOMMAND는 시스템 메뉴의 항목이 선택될 때의 메시지이며 선택된 항목의 번호는 Wparam으로 전달된다. Wparam값에 따라 각 항목의 처리를 수행하고 있다.

# 11. 메뉴에 비트맵 넣기

18jang
GrMenu

이번에는 메뉴에 비트맵을 삽입하는 방법에 관해 알아보자. 메뉴에는 의례히 말로된 문자열만 들어가지만 비트맵을 넣고 싶다면 그렇게 할 수도 있다. 다음이 그 예이다.

그림
메뉴에 삽입된 비트맵


디자인중에 비트맵을 메뉴에 삽입할 수는 없고 코드에서 API 함수를 사용하여야 한다. 메뉴 항목을 미리 만들어 둔 후 FormCreate 이벤트에서 비트맵을 메뉴에 삽입한다.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hbit:TBitmap;
begin
  hbit:=TBitmap.Create;
  hbit.LoadFromFile('bmp1.bmp');
    InsertMenu(GetSubMenu(GetMenu(Handle),0),2,MF_BYPOSITION
      or MF_BITMAP,1000,PChar(LongInt(hbit.handle)));
end;
```

세 번째 인수에 MF_BITMAP 플래그를 사용하여 삽입되는 항목이 비트맵임을 알리고 있으며 다섯 번째 인수로 비트맵의 핸들을 PChar형으로 변경하여 넘겨주면 된다. 캐스트 연산자를 두 번 연거푸 쓰는 방법이 무척 눈에 낯설어 보이겠지만 C에서는 (LPSTR)(LONG)hbit 이런 모양으로 자주 사용한다.

# 12. OnHint 이벤트 사용하기

18jang
OnHint

풍선 도움말은 컴포넌트 주위에 나타나므로 도움말을 신속하게 제공해 주기는 하지만 때로는 시야를 가려 난잡하게 느껴질 경우도 있다. 그래서 풍선 도움말 대신에 상황선이나 별도로 마련된 도움말 출력 장소에 힌트를 출력하는 방법을 사용하기도 한다. 일정한 장소에 도움말을 출력하려면 OnHint 이벤트를 사용해야 하며 이 이벤트는 Hint 속성을 가진 컴포넌트 위로 마우스 커서가 지나갈 때 발생한다. 이 이벤트에서 패널이나 레이블 등 문자열을 출력할 수 있는 컴포넌트로 해당 컴포넌트의 도움말을 출력하도록 한다.

OnHint 이벤트는 Application 컴포넌트의 이벤트이므로 오브젝트 인스펙터

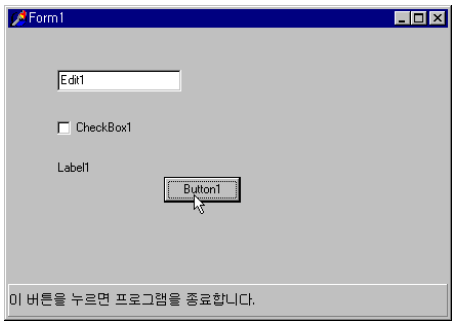
를 통하여 핸들러를 지정할 수 없다. OnHint의 이벤트 핸들러로 사용할 프로시저를 적당한 이름, 예를 들어 DisplayHint 또는 PrintHint 등으로 직접 작성한 후 폼의 OnCreate 이벤트 핸들러에 이 메소드를 Application.OnHint에 대입해 주어야 한다. 예제를 작성해 보자. 코드를 작성하기 전에 도움말을 출력할 패널을 폼에 배치해 두고 Align 속성을 alBottom으로 설정하여 항상 폼의 아래쪽에 있도록 한다. 그리고 Alignment 속성은 taLeftJustify로 설정하여 도움말이 패널의 왼쪽에 나타나도록 하는 것이 좋다. 전체 소스는 다음과 같다. DisplayHint 프로시저의 선언 위치를 주의하도록 하자.

```
unit Onhint_f;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Button1: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    Exit1: TMenuItem;
    Edit1: TMenuItem;
    Cut1: TMenuItem;
    Copy1: TMenuItem;
    Paste1: TMenuItem;
    Edit2: TEdit;
    CheckBox1: TCheckBox;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    procedure DisplayHint(Sender:TObject);
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
Application.OnHint:=DisplayHint;
end;

procedure TForm1.DisplayHint(Sender:TObject);
begin
  Panel1.Caption := Application.Hint;
end;

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

end.
```

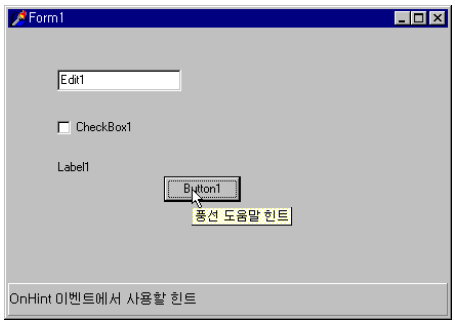
OnHint의 이벤트 핸들러인 DisplayHint 프로시저에서는 패널의 Caption 속성에 Application.Hint 속성을 대입하여 힌트를 출력한다. 여기서 Application.Hint는 현재 마우스 커서 아래쪽에 있는 컴포넌트의 Hint 속성이다. 이렇게 코드를 작성해 놓기만 하면 컴포넌트를 배치하고 Hint 속성에 원하는 도움말을 입력하기만 하면 된다. 컴포넌트의 ShowHint 속성에 상관없이 상황선으로 도움말이 출력될 것이다. ShowHint 속성은 풍선 도움말의 출력 여부만 결정한다. 실행중의 모습은 다음과 같다.

그림
상황선에 출력한 도움말




메뉴 항목과 버튼, 에디트 등의 컴포넌트를 배치해 두고 각 컴포넌트의 Hint 속성에 적당한 도움말을 입력해 두었다. 마우스를 컴포넌트 위로 움직이면 도움말이 상황선에 나타난다. 심지어는 이 프로그램이 포커스를 가지고 있지 않은 상태에서도 마우스 커서만 컴포넌트 위에 있으면 도움말이 출력된다.

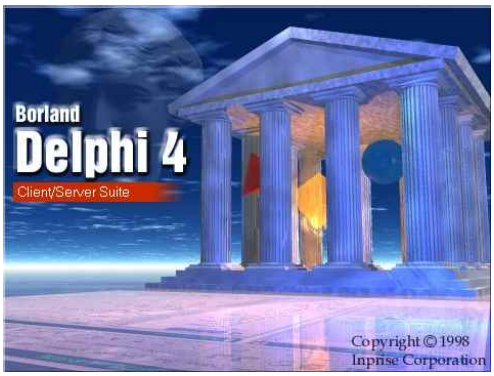
다음은 풍선 도움말과 OnHint 이벤트에서 사용할 도움말을 별도로 정의해 보자. Hint 속성은 풍선 도움말에 사용될 속성과 OnHint 이벤트에서 사용할 도움말을 '|' 문자로 끊어 동시에 정의할 수 있다. 다음 예는 버튼의 Hint 속성이 "풍선 도움말 힌트|OnHint 이벤트에서 사용할 힌트"로 정의된 경우의 도움말 출력 예이다.




Hint 속성에 도움말이 하나만 정의되어 있으면 풍선 도움말과 OnHint 이벤트에서 사용하는 도움말은 같아진다.

# 13. 스플레시 화면

델파이를 실행시키면 화면 정 중앙에 다음과 같은 그림이 열린다. 이 그림은 현재 실행을 준비하고 있는 프로그램이 델파이라는 것을 알려주며 로고 화면 또는 스플레시(Splash) 화면이라고 한다.




이런 그림을 사용하는 이유는 일단은 예쁘다는 미적 효과가 있기 때문이기도 하지만 프로그램이 로드되는 동안 기다리는 사람이 지루하지 않도록 해주는 효과를 가지기도 한다. 웬만큼 덩치가 큰 프로그램은 이런 스플레시 화면을 사용하는 것이 일반적이다. 스플레시 화면을 만드는 것은 생각처럼 그렇게 간단한 일은 아니다. 왜냐하면 프로그램이 실행되기 전에 제일 먼저 나타나는 화면이므로 프로젝트를 직접 편집해야 하기 때문이다.

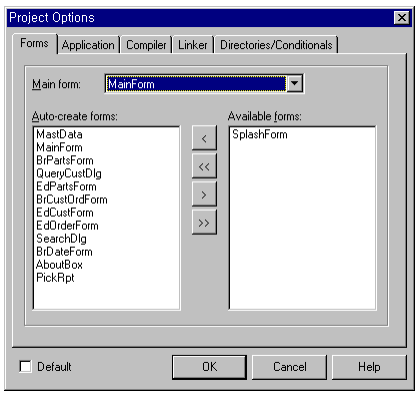
예제는 직접 만들 필요없이 델파이가 제공하는 데모 프로그램 중 MASTAPP라는 예제를 살펴보면 된다. 이 프로그램의 프로젝트 파일을 잠시 살펴보자.

```
begin
  SplashForm := TSplashForm.Create(Application);
  SplashForm.Show;
  SplashForm.Update;
  Application.Title := 'Marine Adventures Order Entry';
  Application.HelpFile := 'MASTAPP.HLP';
  Application.CreateForm(TMainForm, MainForm);
  Application.CreateForm(TBrPartsForm, BrPartsForm);
  Application.CreateForm(TCustQueryDlg, CustQueryDlg);
  Application.CreateForm(TEdPartsForm, EdPartsForm);
  Application.CreateForm(TBrCustOrdForm, BrCustOrdForm);
  Application.CreateForm(TEdCustForm, EdCustForm);
  Application.CreateForm(TOrderForm, OrderForm);
  Application.CreateForm(TPickDlg, PickDlg);
  Application.CreateForm(TBrDateForm, BrDateForm);
  Application.CreateForm(TAboutBox, AboutBox);
  Application.CreateForm(TPickRpt, PickRpt);
  SplashForm.Hide;
  SplashForm.Free;
  Application.Run;
```

```
end.
```

DPR 파일에서는 폼을 생성하고 Application.Run 메소드를 호출하여 메인 폼으로 제어권을 넘기는 일만 하도록 되어 있지만 스플레시 화면을 출력하려면 위와 같이 편집해야 한다. 굵은 글꼴로 된 부분이 사용자가 직접 작성해 넣은 부분이다. 프로젝트가 시작되자마자 제일 먼저 하는 일은 스플레시 폼을 직접 생성하여 보여주는 일이다. 그리고 순서대로 부속 폼을 생성시키고 난 후 스플레시 화면을 파괴시킨 후 메인 폼으로 제어권을 넘기도록 되어 있다. 따라서 스플레시 화면은 DPR에서 폼을 생성하는 동안만 화면에 보이며 폼 생성이 완료되면 사라진다.

스플레시 화면은 프로젝트에서 단 한번 강제적으로 생성되므로 자동으로 생성시킬 필요가 없다. 그래서 Project 옵션의 자동 생성 리스트에서 제외되어 있다.




스플레시 폼은 항상 화면의 중앙에 나타나야 하므로 Position 속성을 PoScreenCenter로 설정해 두어야 하며 다른 폼에 의해 가려지지 않도록 FormStyle 속성을 fsStayOnTop으로 설정해 두는 것이 좋다. MASTAPP 예제에서 사용하는 스플레시 화면은 다음과 같다.

여러분들도 자신이 만든 프로그램의 로고 화면을 직접 디자인하여 사용해 보기 바란다. 뭔가 있어 보이는 느낌이 들지 않는가?

# 14. 폼의 Scaled 속성

18jang
Scaled

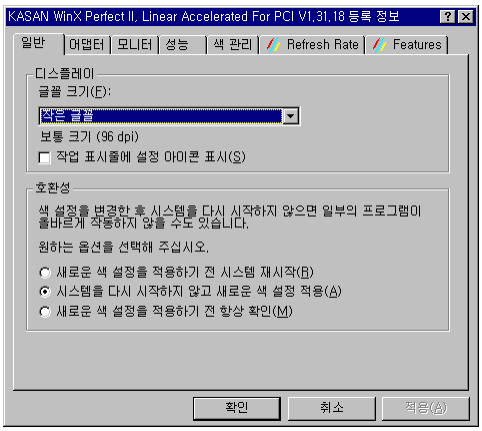
프로그램을 다 만든 후 배포할 때가 되면 마지막 뒷정리를 해야 한다. 이럴 때 개발자들을 곤란하게 만드는 일 중 대표적인 예가 내 컴퓨터에서는 아무 이상이 없었는데 다른 컴퓨터에서는 아주 괴상하게 동작하는 경우다. 그 원인은 사용자마다 시스템 설정 상태가 다르기 때문인데 예를 들어 시스템 색상 설정이 아주 특이하다면 윈도우의 모양이 이상해진다. 그래서 프로그램은 모든 시스템 설정에 무난하게 어울릴 수 있도록 작성되어야 한다.

폼의 Color 속성을 보면 디폴트값이 clBtnFace로 되어 있다. 이 색상은 사용자가 설정한 버튼 색상인데 보통 회색이다. 그렇다고 해서 이 속성을 clSilver나 clGray로 고정해 버리면 사용자의 시스템 설정에 상관없이 회색이 되어 버리기 때문에 부조화의 원인이 된다. 가급적이면 고정된 색상을 사용하지 말고 시스템 색상을 사용하는 것이 좋다.

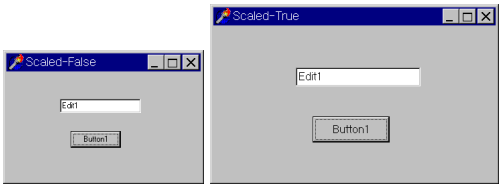
또 하나의 문제가 글꼴의 크기에 따른 문제이다. 보통의 사람들은 윈도우즈가 설치될 때의 디폴트 글꼴 크기를 사용하지만 그렇지 않은 사람도 있다. 시력이 아주 많이 나쁘거나 모니터가 무지막지하게 큰 사람은 좀 더 큰 글꼴을 사용할 수도 있다. 문제는 대화상자에서 컨트롤을 배치하는 기본 단위가 글꼴의 크기와 관련이 있다는 것이다. 그래서 디폴트 크기에서는 대화상자 모양이 썩 볼만했는데 글꼴의 크기가 달라지면 컨트롤도 보이지 않고 서로 겹치는 난리가 나는 것이다.

이런 부조화를 방지해 주는 속성이 바로 폼의 Scaled 속성이다. 이 속성이 True이면 시스템 글꼴 크기에 맞게 폼과 컨트롤의 크기도 같이 늘어나 준다. 이 속성이 False이면 컨트롤의 위치가 글꼴의 크기에 상관없이 무조건 고정되어 버린다. 다행히 Scaled 속성의 디폴트값이 True이므로 이 속성을 특별히 건드리지만 않는다면 이 문제에 대해서는 더 이상 신경쓸 필요가 없다. 이 속성을 변경하면 어떤 효과가 발생하는지 보자.

똑같은 예제를 두 개 만들되 하나는 Scaled 속성을 True로 만들고 하나는 False로 만들었다. 그냥 실행하면 두 프로그램의 모양이 완전히 동일할 것이다. 그러나 시스템의 글꼴 설정을 바꾸어 보면 차이가 생긴다. 바탕 화면의 등록 정보에서 글꼴 크기를 바꾸어 보자.




단, 이 대화상자는 설치된 비디오 카드에 따라 모양이 다를 수도 있다. 글꼴 크기 콤보 박스를 열어 150% 글꼴로 설정하면 시스템이 재부팅된다. 시스템 글꼴이 커졌으므로 탐색기나 바탕화면 등 모든 글꼴이 다 커지며 타이틀 바도 훨씬 더 커졌을 것이다. 글꼴 모양을 바꾼 후 각 프로그램의 모양은 다음과 같다.




Scaled 속성을 False로 설정한 프로그램은 글꼴 크기에 상관없이 컨트롤들이 디자인시의 위치에 그대로 있지만 Scaled 속성을 True로 설정한 프로그램은 컨트롤의 위치가 좀 더 우하단으로 이동해 있다.

## 집필후기

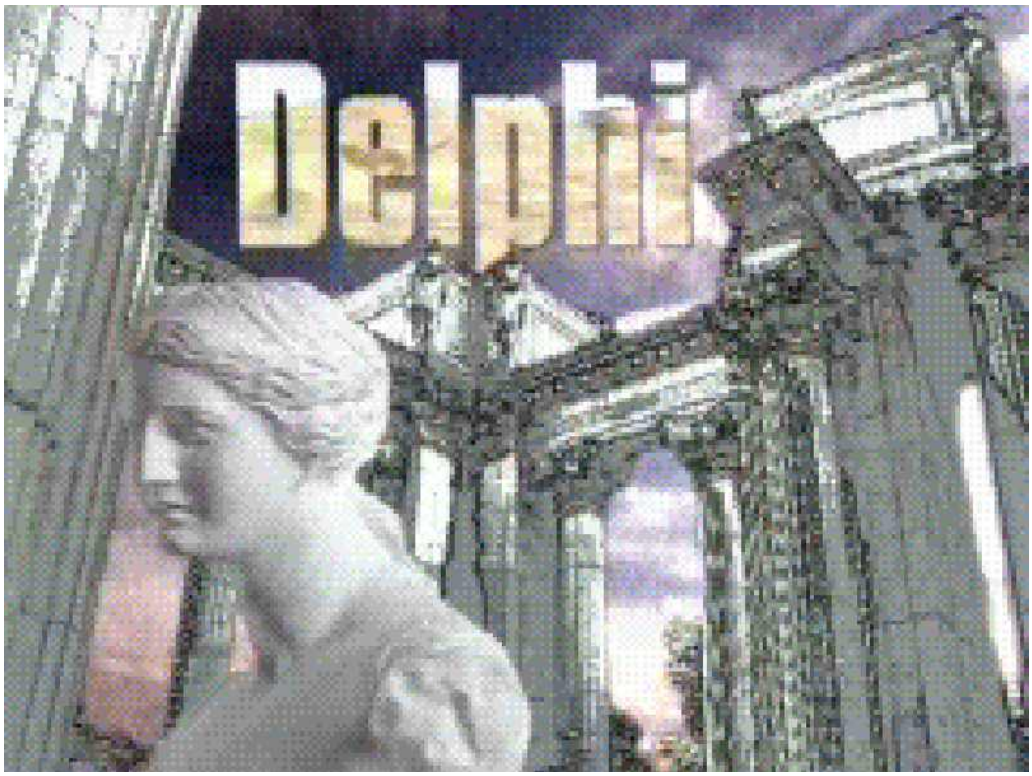
또 하나의 마무리를 하게 되는군요. 스스로 이 책이 많이 부족하고 부끄럽다는 것을 진심으로 잘 알고 있습니다. 시간에 쫓겨 빠진 내용도 있고 때로는 부적절한 설명을 하기도 하고, 오타나 탈자 심지어 틀린 내용도 분명히 있을겁니다. 짧은 시간이었지만 "내가 한시간 노력하면 수천명이 수천시간동안 편하게 공부할 수 있다"는 생각으로 미진한 힘이나마 최선을 다 했다고 자부할 수 있음을 위안으로 삼고자 합니다.

솔직히 말해 델파이 4 는 필자의 능력으로 풀어 쓰기에 어려운 부분들이 많습니다. 델파이 4 를 대하면서 이제 하나의 개발툴에 대한 자습서를 혼자의 힘으로 쓸 수 있는 시대는 지났다는 생각이 듭니다. 요즘의 개발툴은 너무나 거대해서 만능인을 요구하고 있습니다. 기초 프로그래밍, OOP, 데이터 베이스, 인터넷, 네트웍, 멀티 미디어, OLE, COM, ActiveX, CORBA, VCL 의 구조 ..... 등등 홀로 떼어내도 책 한권의 주제가 될만한 것들이 수십개나 모여 있는 지경입니다. 이런 상황은 자연스럽게 테크니컬 라이터(Technical Writer)라는 전문적인 직업인을 요구하고 있습니다. 즉 밥먹고 공부만 하는 사람이 필요하다는 얘기죠.

이제 잔소리 많은 필자는 물러가오며 또 다른 지면을 통해 만나뵐 수 있기 바랍니다. 모든 독자님들 하시는 일에 최선을 다하시고 항상 건강하고 명랑하게 생활하십시요.

# 부록 1

## 배포 CD 사용법



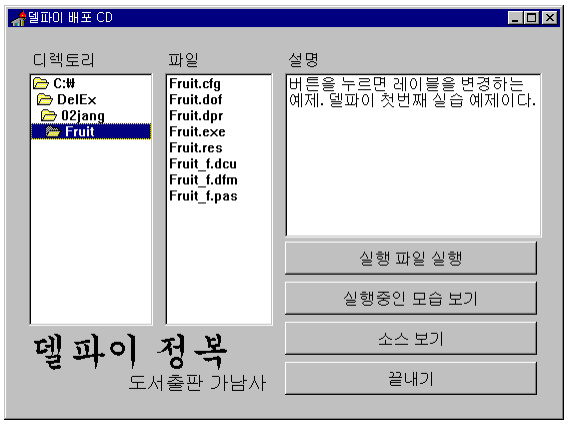
책과 함께 배포되는 CD-ROM 에는 이 책의 모든 예제들이 포함되어 있다. 컴파일 가능한 소스 파일과 바로 실행해 볼 수 있는 실행 파일, 실행 모습을 담은 비트맵 파일들이 포함되어 있다. 전체적으로 두 개의 디렉토리로 구성된다.

# DelEx 디렉토리

이 디렉토리에는 본문에서 작성한 모든 프로젝트의 소스 파일과 실행 파일이 들어있다. 각 장별로 디렉토리를 구성하고 있으며 또한 각 프로젝트별로 서브 디렉토리에 보관되어 있다. CD-ROM 에서 직접 실행 파일을 실행시킬 수는 있지만 예제를 변경해 보고자 할 경우는 해당 예제의 소스 파일을 하드 디스크로 반드시 복사하여야 한다.

또한 Delex 디렉토리에 DELCD.EXE 라는 이름으로 CD 에 포함된 예제를 검색하는 유틸리티가 포함되어 있다. 설치할 필요는 없으며 CD 에서 곧바로 실행할 수 있다. 실행중의 모습은 다음과 같다.

그림
예제 검색 유틸리티

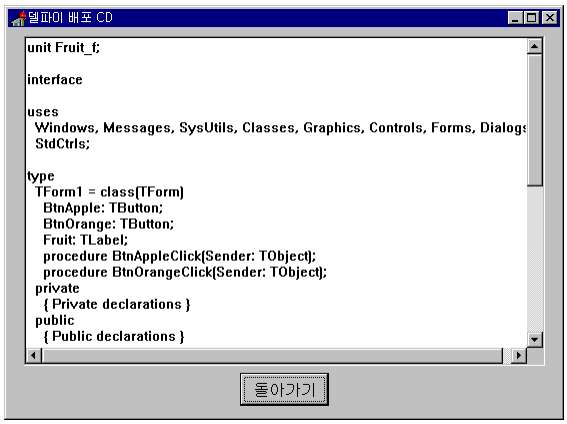



디렉토리 리스트 박스에서 프로젝트가 있는 디렉토리를 선택하면 설명란에 프로젝트에 관한 짧은 설명이 나타난다. DelEx 디렉토리에 대해서만 설명을 제공하므로 다른 디렉토리는 선택해도 아무런 설명이 나타나지 않는다. 파일 리스트 박스는 어떤 파일이 포함되어 있는가를 보여줄 뿐이며 별다른 기능은 가지고 있지 않다. 실행중인 모습보기 버튼을 누르면 다음과 같이 예제의 실행중의 모습을 보여준다.

그림
예제의 실행 모습


실행 파일 실행 버튼을 누르면 예제를 직접 실행해 볼 수도 있다. 소스 보기 버튼을 누르면 프로젝트의 소스를 보여 주므로 델파이를 실행시키지 않아도 소스를 살펴볼 수 있다.

그림
예제의 소스 보기




예제를 하드 디스크로 복사하고자 할 경우는 DelEx 디렉토리를 통째로 하드 디스크로 복사하되 컴파일해 보기 위해서는 Attrib -r/s 명령으로 읽기 전용 속성을 반드시 해제해 주어야 한다. 만약 하드 디스크에 복사한 채로 DelCD 유틸리티를 사용하고자 한다면 디렉토리명은 반드시 루트 아래에 DelEx 라는 이름으로 복사해 주어야 하되 드라이브 명칭에는 영향을 받지 않는다. DelCD 유틸리티를 사용하지 않고 예제만 복사하고자 할 경우는 임의의 디렉토리에 복사할 수 있으며 Bmp 디렉토리를 삭제해도 상관없다.

실행 파일이나 중간 파일(DCU)없이 순수한 예제만 하드 디스크에 설치하고자 할 경우는 DelEx.zip 파일을 풀면 된다. 또한 이 파일은 디스켓에 복사해서 다른 시스템에도 예제를 설치할 수 있으나 단, 몇몇 예제의 데이터(예 AVI 파일)는

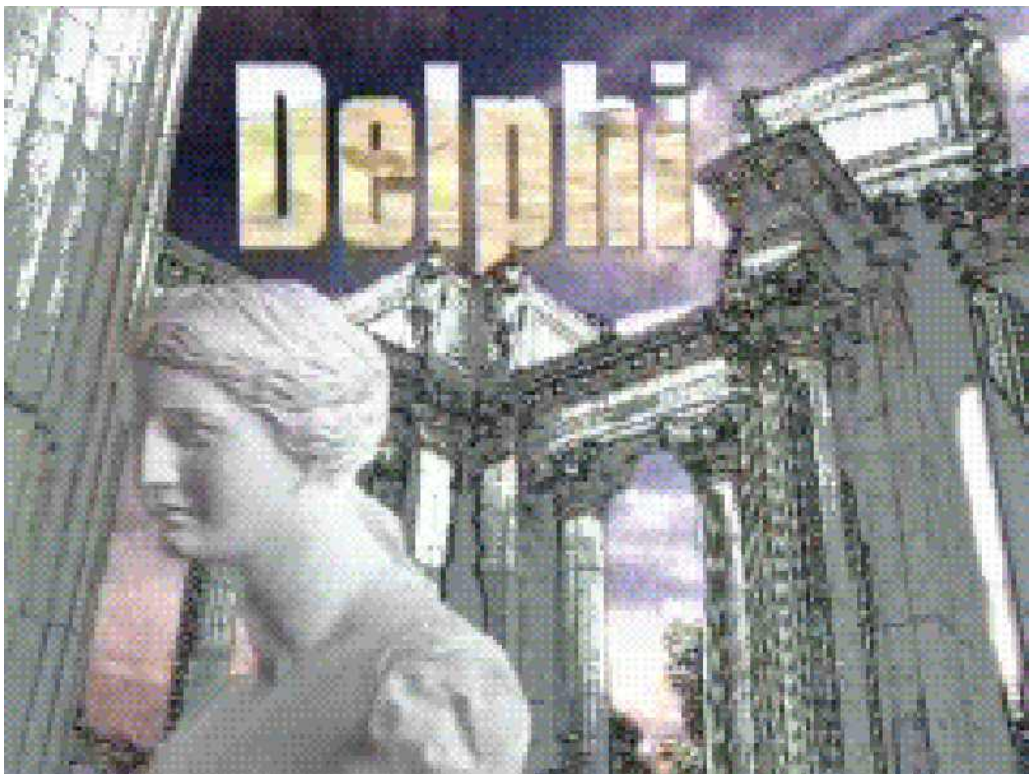
포함되어 있지 않다.

## Etc 디렉토리

이 책 본문에서 사용하는 몇가지 데이터가 포함되어 있다. ActiveX 설치 실습용 OCX 파일, 컴포넌트 설치 실습용 컴포넌트 하나 그리고 도스용 터보 파스칼 매뉴얼 파일이 있다.

# 컴포넌트 레퍼런스

부록 2



델파이가 기본적으로 제공하는 컴포넌트와 오브젝트에 관해 정리하였다. 각 컴포넌트의 정의와 용도를 먼저 밝히고 속성, 메소드, 이벤트를 순서대로 기술하였다. 단 대부분의 컴포넌트에 공통되는 속성, 메소드, 이벤트는 정리하지 않고 각각의 컴포넌트에만 있는 고유한 속성, 메소드, 이벤트만을 논한다. 일반적인 속성에 관한 사항은 이어지는 속성 레퍼런스를 참고하기 바란다. 컴포넌트의 게재 순서는 델파이 4.0 버전의 컴포넌트 팔레트에 있는 순서대로이다. 데이터 베이스 컴포넌트는 기본 컴포넌트와 중복되는 사항이 많거나 레퍼런스가 불필요하다고 판단되어 생략하였으며 샘플 컴포넌트도 싣지 않았다. 속성란에 있어서 읽 기호는 읽기 전용의 속성이라는 뜻이며 실 기호는 실행시에만 사용할 수 있는 속성이라는 뜻이다. 가급적 알파벳 순서대로 게재하려고 했으나 앞뒤로 연관된 속성일 경우는 이 순서를 어긴 경우도 있다.

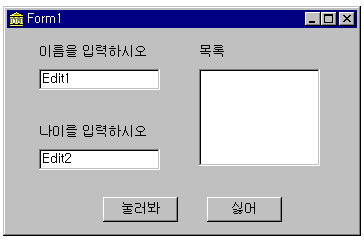
# 1.컴포넌트 레퍼런스

## 1. Form

### ◘ 정의

델파이 프로그램의 기본을 이루는 것이 폼이며 폼은 흔히 말하는 하나의  윈도우이다. 폼 위에 버튼, 레이블, 에디트 박스, 리스트 박스 등의 다양한 컴포넌트를 배치함으로써 윈도우를 디자인한다. 새 프로젝트를 시작할 때 델파이는 아무런 컴포넌트도 가지지 않는 빈 폼을 만들어 준다.  이 폼에 여러 가지 컴포넌트를 배치하고 폼의 속성을 바꾸고 각 컴포넌트에 코드를 작성함으로써 프로그램이  완성되어 간다. 컴포넌트 팔레트에는 없으므로 폼을 추가하려면 메뉴 명령을 사용하여야 한다.


빈 폼




디자인 후

폼의 HorzScrollBar, VertScrollBar 속성을 설정함으로써 스크롤 바를 폼에 부착할 수 있으며, WindowState 속성을 설정함으로써 최초 폼이 화면에 출력될 때의 상태를 최대, 최소, 보통 상태 중 하나로 지정한다.

델파이는 하나의 윈도우 경계 내에 여러 개의 차일드 윈도우가 존재하는 MDI 프로그램을 쉽게 만들 수 있는 속성을 제공한다. 폼의 FormStyle 속성을 사용하여 어떤 폼이 페어런트 폼이 되며 어떤 폼이 차일드 폼이 될 것인가를 지정한다. 프로젝트 내에 여러 개의 폼이 있을 경우 FormStyle 속성이  fsMDIForm인 폼이 페어런트 폼이 되며 FormStyle 속성이 fsMDIChild인 폼이 차일드 폼이 된다. 또한 차일드 폼을 정렬하는 Cascade, Tile, ArrangeIcons 등의 메소드를 제공하여 페어런트 폼 내의 차일드 폼을 정렬하도록 해준다.

폼이 생성될 때 발생하는 OnCreate 이벤트 처리 루틴에서 폼의 기본적인 속성을 변경하거나 프로그램에 필요한 초기화 작업을 해준다. 예를 들어 프로그램에서 사용할 배열을 초기화 하거나 변수를 초기화하는 작업을 폼이 생성될 때 해 주도록 한다.

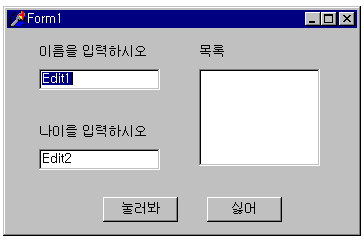
## ◘ 속성

### ActiveControl:TWinControl;

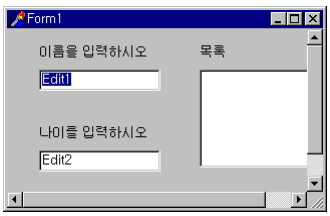
한 프로그램에서 포커스(입력 초점)은 오직 하나의 컨트롤만 가질 수 있다. 이 속성값을 읽으면 현재 포커스를 가진 컨트롤을 조사할 수 있으며 이 속성에 컨트롤을 대입하면 포커스를 이동할 수도 있다. 폼이 현재 포커스를 가지고 있지 않을 때 이 속성은 폼이 포커스를 가질 경우 포커스를 받을 컨트롤을 가리킨다. 컨트롤간에 포커스가 이동될 때 OnExite 이벤트가 일어나기 전에 이 속성이 변경된다.

### AutoScroll:Boolean;

폼에 속한 모든 컴포넌트를 다 나타낼 수 없을 정도의 크기일 때 자동으로 스크롤 바가 나타나도록 설정한다. 사용자가 폼의 크기를 조정하여 폼의 크기가 줄어들 경우 폼의 하단(수평)이나 우측(수직)에 스크롤 바를 만들어 이동할 수 있도록 해 준다. 디폴트로 이 값은 True이지만 만약 스크롤 바를 어떤 경우에라도 나타내고 싶지 않다면 False로 변경하도록 한다.




스크롤 바가 없다




크기를 줄이면
스크롤 바가 생긴다.

### BorderIcons:TBorderIcons;

폼의 타이틀 바에 어떤 아이콘을 나타낼 것인가를 지정한다. 타이틀 바에는 보통 윈도우의 제목이 표시되지만 이 외에도 시스템 메뉴, 최소화 버튼, 최대화 버튼, 도움말 버튼이 있을 수 있다. 다음 네 개의 아이콘이 타이틀 바에 위치할 수 있으며 각각 Boolean형의 세부 속성이다.

| 세부 속성 | 의미 |
|---|---|
| biSystemMenu | 컨트롤 메뉴 |
| biMinimize | 최소화 버튼 |
| biMaxmize | 최대화 버튼 |
| biHelp | 도움말 버튼 |

타이틀 바에서 아이콘을 없애려면 해당 세부 속성을 False로 설정한다.


### BorderStyle:TFormBorderStyle;

폼의 경계선 모양을 지정한다. 이 속성을 어떻게 지정하는가에 따라 폼의 모양은 물론 폼의 기능에도 변화가 생긴다. 경계선은 단순한 장식이 아니며 경계선의 모양에 따라 폼 전체의 모양과 기능이 달라지기 때문이다. 다음 여섯 가지 값이 가능하다.

| 속성값 | 의 미 |
|---|---|
| bsDialog | 크기 조정이 불가능하며 대화상자 형태를 가진다. |
| bsNone | 경계선을 가지지 않으며 최소, 최대, 조절 메뉴도 가지지 않는다. 크기 조정은 물론 불가능하며 타이틀 바도 없기 때문에 위치를 옮길 수도 없다. |
| bsSingle | 크기 조정이 불가능하며 선 하나로 된 경계선을 가진다. |
| bsSizeable | 크기 조정이 가능한 표준적인 경계선을 가진다. 이 속성이 디폴트이다. |
| bsToolWindow | bsSingle 와 같되 타이틀 바의 높이가 좁으며 시스템 메뉴와 닫기 버튼이 없다. |
| bsSzieToolWindow | bsTollWindow 와 같되 크기 조정이 가능하다. |

### ClientHeight:Integer;

폼의 작업 영역 높이를 나타낸다. 작업 영역의 높이는 폼의 높이에서 타이틀 바와 경계선의 높이를 제외한 부분이며 컴포넌트가 배치되는 영역이기도 하다. 폼의 높이를 조정하고자 할 때는 Height 속성을 변경하지만 작업 영역의 높이만 변경하고자 할 때는 이 속성을 변경하는 것이 더 좋다. Height 속성으로 작업 영역의 크기를 변경하려면 타이틀 바와 경계선의 높이를 계산해 줘야 하기 때문이다. 이 속성을 변경하면 Height 속성도 같이 변경된다.

## ClientWidth:Integer;

폼의 작업 영역 너비를 나타낸다. 작업 영역의 너비는 폼의 너비에서 양쪽 경계선의 폭을 제외한 부분이며 컴포넌트가 배치되는 영역이기도 하다. 이 속성을 변경하면 Width 속성도 같이 변경된다.

## 읽 실 Components[i]:TComponent;

폼에 소속된 모든 컴포넌트의 리스트를 가지는 배열 속성이며 이 속성을 사용하면 배열 첨자를 이용하여 각 컴포넌트를 참조할 수 있다. 폼에 소속된 컴포넌트의 개수를 알고자 할 때는 ComponentCount 속성값을 읽으면 된다.

## 읽 실 Controls[i]:TControl;

폼의 차일드인 모든 컨트롤의 목록을 가지는 배열 속성이며 이 속성을 사용하면 배열 첨자를 사용하여 각 컨트롤을 참조할 수 있다. Components 속성은 폼에 소속된 모든 컴포넌트의 목록을 가지는 데 반해 Controls 속성은 폼의 차일드인 컨트롤의 목록만을 가진다. 예를 들어 폼에 패널이 있고 패널 안에 버튼이 있으면 버튼은 폼에 소속된(owned) 컴포넌트이므로 Components 속성의 목록에는 나타나지만 폼의 차일드는 아니므로 Controls 속성의 목록에는 나타나지 않는다. 폼의 차일드 컨트롤의 개수를 알려면 ControlCount 속성을 읽으면 된다.

## FormStyle:TFormStyle;

폼의 형태를 지정한다. 다음 중 하나의 값을 설정한다.

| 속성값 | 의미 |
|---|---|
| fsNormal | 보통의 폼 |
| fsMDIChild | MDI 차일드 윈도우 |
| fsMDIForm | MDI 메인 윈도우 |
| fsStayOnTop | 프로젝트 내에 있는 다른 어떤 폼보다 항상 위에 있는 폼이며 다른 폼에 의해 가려지지 않는다. 단 이 속성을 가진 윈도우끼리는 서로 가려진다. |

디폴트값은 fsNormal이며 MDI 폼이 아닌 보통의 폼이다. MDI 프로그램을 만들 경우 메인 폼의 FormStyle은 fsMDIForm이어야 하며 자식 폼의 FormStyle은 fsMDIChild여야 한다.

## 읽 실 Handle:HWIND;

폼의 윈도우 핸들값이다. 이 값은 델파이에서는 잘 사용하지 않지만 윈도우즈 API 함수를 사용할 때 함수의 인수로 전달된다.

## HorzScrollBar:TControlScrollBar;

폼의 하단에 나타나는 수평 스크롤 바의 속성을 설정하며 다음과 같은 세부 속성을 가진다.

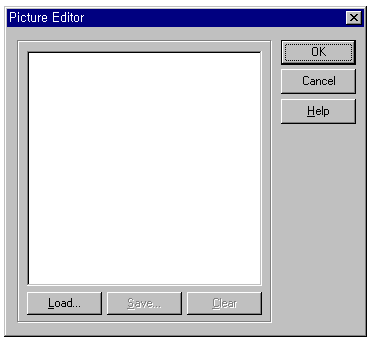
| 세부 속성 | 의미 |
|---|---|
| Increment | 사용자가 스크롤 바 끝의 버튼을 누를 때 이동할 거리이며 디폴트는 8이다. |
| Margin | 폼 가장자리와 컨트롤 간의 최소 픽셀수 |
| Position | 스크롤 바 썸의 위치. |
| Range | 스크롤 바로 이동할 수 있는 폼의 크기를 지정하며 이는 폼의 가상적인 크기가 된다. 예를 들어 폼의 넓이가 100이고 Range가 200이면 우측으로 100만큼의 영역이 더 있다는 뜻이다. |
| Visible | 스크롤 바를 보여줄 것인가를 지정한다. 이 속성이 False이면 스크롤 바는 나타나지 않으며 폼의 숨겨진 영역으로 이동할 수 없다. |

## VertScrollBar:TControlScrollBar;

폼의 우측에 나타나는 수직 스크롤 바의 속성을 설정하며 수평 스크롤 바와 동일한 세부 속성을 가진다.

## Icon:TIcon;

폼이 최소화되었을 때 나타나는 아이콘을 지정한다. 이 속성을 지정하지 않을 경우 프로그램이 사용하는 아이콘을 폼의 아이콘으로 사용한다. 더블클릭하면 아이콘을 읽어올 수 있는 대화상자가 나타난다.




## KeyPreview:Boolean;

실행중에 키보드 입력은 포커스를 가진 컨트롤로 곧바로 전달된다. 그러나 이 속성이 True로 설정되면 모든 키 입력은 일단 폼으로 전달된 후에 포커스를 가진 컨트롤로 전달된다. 키 입력을 폼에서 통제하고자 할 때 이 속성을 사용하며 디폴트는 False로 설정되어 있다.

## Menu:TMainMenu;

이 폼에 연결된 메뉴를 설정한다. 디폴트로 이 속성에는 폼에 제일 먼저 놓은 MainMenu 컴포넌트가 대입되지만 실행중에 이 속성을 변경하면 메뉴를 번갈아가며 사용할 수 있다.

### 실 ModalResult:TModalResult;

이 속성에 0이 아닌 값을 대입하면 Modal형의 폼을 종료하게 되며 이 속성에 대입된 값은 폼을 호출한 ShowModal 함수의 리턴값이 된다. 이 속성은 별도의 코드 없이 속성만으로 폼을 닫고자 할 때 사용된다. 예를 들어 대화상자에 닫기 버튼을 배치하고 이 버튼의 ModalResult 속성을 0이 아닌 값으로 설정하면 버튼을 누르기만 해도 폼이 즉각 종료된다.

### Position:TPosition;

폼이 처음 화면에 나타날 때의 크기와 위치를 지정한다. 디폴트는 poDesigned로 되어 있어 디자인할 때 설정한 크기와 위치를 그대로 사용하지만 속성을 변경하면 특정한 위치와 특정한 크기를 사용할 수 있다.

| 속성값 | 의미 |
|---|---|
| poDesigned | 디자인할 때 설정한 크기와 위치를 그대로 사용한다. |
| poDefault | 델파이가 적절한 위치에 적절한 크기로 폼을 알아서 배치하도록 한다. |
| poDefaultPosOnly | 폼의 크기는 디자인시에 설정한 크기를 사용하며 위치는 델파이가 알아서 설정한다. |
| poDefaultSizeOnly | 폼의 위치는 디자인시에 설정한 크기를 사용하며 위치는 델파이가 알아서 설정한다. |
| poScreenCenter | 폼의 크기는 디자인시에 설정한 크기를 사용하되 위치는 화면 해상도에 상관없이 항상 화면의 중앙에 폼이 나타나도록 한다. 대화상자나 로고 폼에 적당한 속성이다. |
| poDesktopCenter | poScreenCenter와 동일하다. 단 복수 개의 모니터가 설치되어 있을 경우에는 다르다. |

### PrintScale:TPrintScale

Print 메소드로 폼을 인쇄할 때 폼의 크기 비율을 설정한다. 디폴트로 이 값은 poNone 이며 이는 어떠한 크기 조정도 하지 않는다는 뜻이다. 이 속성을 poProportional 로 변경하면 폼이 화면상의 크기와 유사하도록 확대된다. poPrintToFit 로 설정하면 폼이 용지에 가득차도록 인쇄되는데 단 이 때 종횡비는 화면과 같이 유지된다.

### WindowState:TWindowState;

폼이 처음 보여질 때의 상태를 지정하며 다음 세 가지 값 중 하나의 값을 가진다.

| 속성값 | 의미 |
|---|---|
| wsNormal | 원래의 크기대로 나타난다. |
| wsMaximize | 최대화되어 나타난다. |
| wsMinimize | 최소화된 아이콘 모양으로 나타난다. |

디폴트값은 wsNormal이며 작성시에 디자인한 모양대로 나타난다.

## ◘ 메소드

■ procedure ArrangeIcons;

최소화된 폼의 아이콘을 일정한 간격으로 정렬하여 겹치지 않도록 배열한다. 이 메소드는 MDI 페어런트 폼에만 적용할 수 있다.

■ procedure BringToFront;

폼이 전면에 드러나도록 한다. 폼이 다른 윈도우에 의해 가려져 있을 경우 이 메소드를 호출하여 폼을 활성화시킨다.

■ procedure Cascade;

차일드 폼을 겹치도록 정렬한다. 각 폼의 상단부가 보이도록 정렬하므로 원하는 폼을 마우스로 쉽게 선택할 수 있다. 이 메소드는 MDI 페어런트 폼에만 적용할 수 있다.

■ function ClientToScreen(Point: TPoint): TPoint;

■ function ScreenToClient(Point: TPoint): TPoint;

주어진 좌표를 작업 영역 좌표에서 전체 화면 좌표로 또는 그 반대로 변경한다. 작업 영역 좌표란 폼(또는 컨트롤)의 좌상단을 원점으로 하는 좌표이며 전체 화면 좌표는 화면의 좌상단을 원점으로 하는 좌표다. 이 두 메소드를 사용하여 두 좌표 체계를 변환한다.

■ procedure Close;

폼을 종료한다. 이 메소드를 호출하는 것은 사용자가 시스템 메뉴에서 닫기 항목을 선택하는 것과 동일한 효과를 가져온다. 이 메소드가 호출되면 우선 CloseQuery 메소드를 호출하여 폼을 닫아도 되는지를 점검해 본다. 만약 CloseQuery 메소드가 False 를 리턴하면 폼을 닫을 수 없는 상황(예를 들어 미보관 문서가 있는 경우)으로 간주하고 닫기 동작을 중지한다.

■ function GetFormImage: TBitmap;

프린터로 인쇄할 경우의 폼 모양을 비트맵으로 리턴한다.

■ procedure Hide;

Visible 속성을 False 로 변경하여 폼이 보이지 않도록 한다. 폼이 보이지 않는 상태에서도 폼의 속성을 변경하거나 메소드를 호출할 수 있다.

■ procedure Invalidate;

폼을 다시 그리도록 한다. 이 메소드를 호출하면 OnPaint 메시지가 발생하며 다른 윈도우에 의해 지워진 부분이 즉각 다시 그려진다.

■ procedure Next; procedure Previous;
다음(또는 앞) 차일드 폼을 활성화시킨다. 이 메소드는 MDI 페어런트 폼에만 적용할 수 있다.

■ procedure Print;
폼을 프린터로 인쇄한다. 인쇄되는 크기는 PrintScale 속성값에 따라 달라진다.

■ procedure Refresh;
폼에 그려진 모든 이미지를 전부 삭제한 후 다시 그린다. 이 메소드는 Invalidate 와 Update 를 연속하여 호출한다.

■ procedure Release;
폼을 파괴하고 폼이 사용하던 메모리를 해제한다. Free 와 비슷하지만 이벤트 핸들러가 실행을 완전히 마칠 때까지 대기한다는 점이 다르다.

■ procedure Repaint;
폼을 다시 그리되 이미 그려진 이미지를 지우지는 않는다.

■ procedure ScrollBy(DeltaX, DeltaY: Integer);
폼을 강제로 스크롤시킨다. 스크롤 바를 사용하지 않고 코드로 스크롤시킬 때 이 메소드를 사용한다. DeltaX 가 양수이면 오른쪽으로, 음수이면 왼쪽으로 스크롤된다. DeltaY 가 양수이면 아래쪽으로, 음수이면 위쪽으로 스크롤된다.

■ procedure ScrollInView(AControl: TControl);
인수로 주어진 AControl 이 보이도록 폼을 스크롤시킨다.

■ procedure SendToBack;
폼을 다른 윈도우 뒤쪽으로 이동시키며 이 때 폼이 포커스를 가지고 있었다면 포커스를 잃게 된다.

■ procedure Show;
Visible 속성을 True 로 변경하여 폼이 보이도록 해 주며 만약 폼이 다른 윈도우에 가려져 있다면 BringToFront 메소드를 호출하여 폼이 전면으로 드러나도록 해준다. Show 메소드에 의해 보여지는 폼은 모델리스 형이다.

■ function ShowModal: Integer;
Show 와 마찬가지로 폼을 활성화시키지만 모달형으로 폼을 활성화시킨다. 그래서 사용자는 모달형 폼을 닫기 전에는 다른 작업을 할 수 없다. 리턴값은 폼의 ModalResult 속성이다.

■ procedure Tile;
차일드 윈도우가 겹치지 않도록 일정한 크기로 정렬한다. 차일드 윈도우들은 페어런트의 작업 영역을 모두 사용하여 균등하게 배치된다. 이 메소드는 MDI 페어런트 폼에만 적용할 수 있다.

■ procedure Update;
UpdateWindow API 함수를 호출하여 아직 처리되지 않은 Paint 메시지를 즉각 처리하도록 한다.

## ◘ 이벤트

### OnActivate
폼이 활성화되어 포커스를 가지게 될 때 보내진다. 폼이 활성화될 때 해야 할 특별한 작업이 있다면 이 이벤트의 핸들러에서 처리하면 된다. 단 주의할 것은 이 이벤트는 같은 프로그램내의 다른 폼으로부터 포커스가 전달될 때만 발생하며 다른 프로그램으로부터 포커스를 받을 때는 발생하지 않는다는 점이다. 다른 프로그램으로부터 포커스가 옮겨왔을 때는 Application 오브젝트의 OnActivate 이벤트가 발생한다.

### OnDeActivate
폼의 포커스가 다른 폼으로 이동할 때 발생한다. 단 다른 프로그램으로 이동할 때는 이 이벤트가 발생하지 않으며 같은 프로그램내의 다른 폼으로 이동할 때에만 발생한다. 이 경우 대신 Application 오브젝트의 OnDeActivate 이벤트가 발생한다.

### OnClose
폼이 닫히기 직전에 발생하는 이벤트이다. 폼을 닫을 때 특별한 작업, 예를 들어 미 보관 문서를 저장하거나 자원을 반납하는 등의 작업이 필요하다면 이 이벤트 핸들러에서 하도록 한다. 참조 인수로 Action 이라는 인수가 전달되는데 이 인수로 폼을 정말로 닫을 것인가를 지정한다.

| Action | 의미 |
|---|---|
| caNone | 폼이 닫히는 것을 거부하며 아무 일도 일어나지 않는다. |
| caHide | 폼을 닫지는 않고 숨기기만 한다. SDI 차일드의 디폴트값이다. |
| caFree | 폼을 닫고 폼이 사용하던 모든 메모리를 해제한다. |
| caMinimize | 폼을 최소화시킨다. 이 값은 MDI 차일드의 디폴트값이다. |

### OnCloseQuery
폼이 닫히기 직전에 정말로 폼을 닫아도 되는 상황인지를 점검하기 위해 발생하는 이벤트이다. 참조 인수로 CanClose 가 전달되는데 이 인수에 True 를 대입해 주면 폼이 닫히며 False 를 대입해 주면 폼 닫기가 취소된다.

### OnCreate
폼이 처음 생성될 때 발생하는 이벤트이며 폼의 생성자에서 이 이벤트를 발생시킨다. 폼 생성과 함께 해야 할 초기화 작업이 있

다면 이 이벤트 핸들러에서 하거나 아니면 생성자를 재정의(Override)하여 수행하되 두 가지를 다 하는 것은 바람직하지 않다. 이 핸들러에서 생성된 오브젝트는 OnDestroy 이벤트 핸들러에서 파괴해 주어야 한다. 참고로 이 이벤트 이후에 발생하는 이벤트의 순서는 OnCreate-OnShow-OnActivate-OnPaint 순이다.

### OnDestroy
폼이 파괴될 때 발생하는 이벤트이다. OnCreate 이벤트 핸들러에서 생성한 오브젝트를 파괴하기 위한 목적으로 사용된다. 같은 목적으로 파괴자를 재정의(Override)하여 사용할 수도 있지만 두 가지를 다 하는 것은 바람직하지 않다.

### OnHelp
사용자가 F1 키를 누르거나 기타의 방법으로 도움말을 요청했을 때 발생한다. 이 이벤트 핸들러에서 도움말 요청에 대한 특별한 처리를 수행한다.

### OnShow
폼이 나타나기 직전에 발생하는 이벤트이다. OnCreate 와 마찬가지로 초기화에 사용할 수도 있으나 폼이 숨겨졌다 나타난 경우에도 이 이벤트가 발생하므로 여러 개의 폼을 가진 프로젝트에서 오브젝트 생성 등의 초기화에는 적합하지 않다. Visible 속성이 True 로 변경될 때마다 이 이벤트가 발생한다.

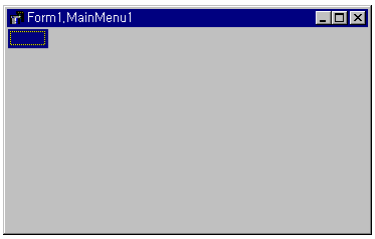
### OnHide
Visible 속성이 False 가 될 때, 즉 폼이 숨겨질 때 발생하는 이벤트이다.

### OnPaint
폼이 다시 그려져야 할 필요가 있을 때 발생하는 이벤트이며 윈도우즈의 WM_PAINT 메시지에 의해 발생한다. 대개의 경우 이 메시지를 처리할 필요가 없지만 폼의 캔버스에 직접 그리기를 했다면 이 이벤트 핸들러에서 그려야 한다. 이 이벤트는 폼 내의 컨트롤이 다시 그려지기 전에 발생한다.

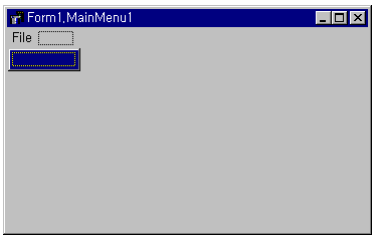
### OnShortKey
사용자가 키를 누를 때 OnKeyDown 이벤트에 선행해서 발생하는 이벤트이며 폼은 이 이벤트의 핸들러에서 쇼트컷을 다른 것으로 변경할 기회를 가진다. 인수로 TWMKey 형의 구조체가 전달되는데 이 구조체에는 어떤 키가 눌러졌는지에 대한 정보가 들어있다.

## 2. MainMenu

### ◘ 정의
폼 상에 메뉴를 만든다. 이 컴포넌트를 폼에 가져다 놓는다고 해서 곧바로 메인 메뉴가 만들어지는 것은 아니며 별도의 메뉴 디자인을 해 주어야 한다. 메뉴를 만들기 위해서는 메인 메뉴 컴포넌트를 폼에 위치시키고 컴포넌트를 더블클릭한다. 메뉴를 디자인할 수 있는 다음과 같은 별도의 윈도우가 열린다. 이 윈도우를 메뉴 디자이너라고 한다.




이 화면에서 메뉴를 만들면 폼에 메뉴가 출력된다. 메뉴를 만드는 방법은 메뉴 디자이너의 좌상단 빈칸(Placeholder)에서부터 아이템을 하나씩 순서대로 입력하는 방식을 사용한다. 메뉴 항목을 입력할 때는 오브젝트 인스펙터의 Caption 속성에 메뉴 항목으로 쓰일 문자열을 입력하면 된다. 예를 들어 File 메뉴를 만들고자 한다면 메뉴 디자이너의 빈칸을 선택해 놓은 후 오브젝트 인스펙터의 Caption란에 File이라는 문자열을 입력하면 된다. 메뉴 항목이 하나 만들어지며 다음 그림과 같이 우측과 아래쪽에 빈칸이 추가된다.




좌상단에서 아래로 이동하면 메뉴 항목을 만드는 것이고 우측으로 이동하면 팝업 항목을 만드는 것이다. 메뉴 디자이너에서 만들어지는 각각의 항목은 TMenuItem 오브젝트이며 고유의 Name과 Caption 속성을 가진다. 그러므로 개별적인 메뉴 항목에 대한 속성 설정은 메뉴 디자이너에서 항목을 선택(반전 막대를 위치시킨다)한 후 오브젝트 인스펙터를 사용한다.

## ◘ 속성

### AutoMerge:Boolean;
MDI 프로그램이 아닌 경우 두 번째 폼의 메뉴가 메인 폼의 메뉴와 병합될 것인지를 지정한다. MDI의 경우는 이 속성에 상관없이 메뉴가 병합된다.

### Images:TImageList;
메뉴 항목의 왼쪽에 나타날 이미지의 리스트를 지정한다. 개별 메뉴 항목의 이미지 번호는 ImageIndex 속성으로 설정한다.

### Items:TMenuItem;
TMenuItem 컴포넌트의 배열이며 이 속성을 더블클릭하면 메뉴 디자이너가 열린다. 이 속성은 직접 편집할 수 없으며 반드시 메뉴 디자이너를 사용하여 편집해야 한다.

### OwnerDraw:Boolean;
각 메뉴 항목을 응용 프로그램이 직접 그릴 것인가를 지정한다. 이 속성이 False이면 메뉴 디자이너로 입력한 대로 메뉴 항목이 그려지지만 이 속성이 True이면 응용 프로그램이 OnDrawItem 이벤트 핸들러에서 메뉴 항목을 직접 그리게 된다. 응용 프로그램은 메뉴를 그리기 위해 OnMeasureItem이벤트와 OnDrawItem 이벤트를 받아 각 메뉴 항목의 크기를 결정하고 메뉴 항목을 직접 그리게 된다.

# 3. PopupMenu

## ◘ 정의
팝업 메뉴 컴포넌트는 폼에 팝업 메뉴를 배치한다. 팝업 메뉴는 컨트롤에서 마우스 오른쪽 버튼을 누르면 나타나며 해당 컨트롤에 관련된 명령만을 담고 있는 메뉴이다. 이 컴포넌트를 배치한 후 더블클릭하여 메뉴 디자이너를 열고 오브젝트 인스펙터를 사용하여 메뉴 항목을 입력한다. 메뉴 디자이너를 사용하는 방법은 메인 메뉴의 경우와 동일하다. 팝업 메뉴는 개별적인 개체에 연결되어 사용되며 연결하고자 하는 개체의 PopupMenu 속성에 팝업 메뉴의 이름을 기입해 준다.

예를 들어 리스트 박스 Listbox1에 Popup1을 연결하려면 Listbox1.PopupMenu 속성을 Popup1로 정의한다. 그러면 실행중에 리스트 박스 위에서 마우스의 오른쪽 버튼을 누르면 Popup1 팝업 메뉴가 나타난다. 팝업 메뉴는 필요에 따라 폼에 여러 개를 배치할 수 있으며 여러 개의 컴포넌트가 하나의 팝업 메뉴를 공유할 수 있다. 팝업 메뉴가 화면에 나타날 때 OnPopup 이벤트가 발생하므로 이 이벤트 핸들러에 팝업 메뉴의 메뉴 항목을 초기화하는 코드를 작성한다.

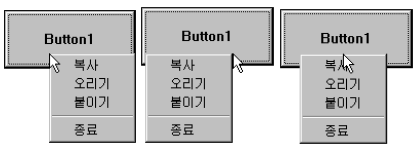
## ◘ 속성

### AutoPopup:Boolean;
팝업 메뉴가 연결된 컴포넌트 위에서 마우스의 오른쪽 버튼을 누를 때 팝업 메뉴가 자동으로 나타날 것인가를 지정한다. 디폴트값이 True이므로 사용자는 오른쪽 버튼을 눌러 언제든지 팝업 메뉴를 불러 쓸 수 있다. 이 속성이 False일 경우는 마우스의 오른쪽 버튼으로 팝업 메뉴를 나타낼 수 없으며 Popup 메소드를 사용하여 강제적으로 팝업 메뉴가 나타나도록 해야 한다. 키보드로 팝업 메뉴를 제어한다거나 특정 조건에서만 팝업 메뉴를 나타나게 하려면 이 속성을 False로 설정한다.

### Alignment:TPopupAllignment;
마우스 오른쪽 버튼을 누를 때 팝업 메뉴가 나타날 위치를 지정한다.

| 속성값 | 의미 |
|---|---|
| paLeft | 마우스 커서의 좌상단에 팝업 메뉴가 나타난다. |
| paRight | 마우스 커서의 우상단에 팝업 메뉴가 나타난다. |
| paCenter | 마우스 커서의 상단 중앙에 팝업 메뉴가 나타난다. |

다음 그림은 버튼에 팝업 메뉴를 지정한 후 이 속성을 좌, 우, 중앙으로 각각 변경해 본 것이다. 마우스 커서의 위치와 팝업 메뉴의 위치가 속성값에 따라 달라진다.




참고로 이 속성값의 매크로명은 팝업 메뉴를 기준으로 한 마우스 커서의 위치이지 마우스 커서의 위치를 기준으로 한 팝업 메뉴 위치가 아님을 유의해야 한다. plLeft는 마우스 커서를 기준으로 팝업 메뉴가 왼쪽에 열리는 것이 아니라 오른쪽에 열린다.

### 실 PopupComponent:TComponent;
팝업 메뉴를 최후로 출력하게 한 컴포넌트의 이름을 담는다. 여러 개의 컴포넌트가 하나의 팝업 메뉴를 공유한 경우 이 속성값을 읽으면 어떤 컴포넌트에 의해 팝업 메뉴가 호출되었는지를 알 수 있다.

### Images:TImageList;
메인 메뉴의 Images 속성과 동일하다.

### Items:TMenuItem;
메인 메뉴의 Items 속성과 동일하다.

### OwnerDraw:Boolean;
메인 메뉴의 OwnerDraw 속성과 동일하다.

## ◘ 메소드

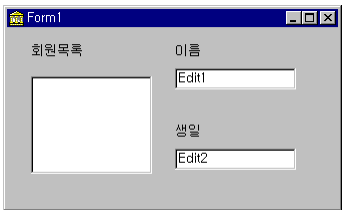
### Popup (X, Y: Integer)
팝업 메뉴를 (X,Y)위치에 강제로 나타나도록 한다. 팝업 메뉴는 보통 사용자가 마우스 오른쪽 버튼을 눌러야만 나타나지만 이 메소드를 사용하면 코드로 팝업 메뉴를 호출할 수 있다. 마우스 오른쪽 버튼 외의 방법으로 팝업 메뉴를 호출하도록 하려면 AutoPopup 속성을 False 로 설정해 두고 이 메소드를 사용하면 된다. 두 개의 인수 (X,Y)는 팝업 메뉴가 출력될 좌표이되 이 좌표는 전체 화면을 기준으로 한 좌표이므로 폼 위에 나타나도록 하려면 폼의 좌상단 좌표인 Left, Top 을 더해 주어야 한다.

## ◘ 이벤트

### OnPopup
팝업 메뉴가 나타나기 직전에 발생하는 이벤트이다. 보통 이 이벤트 핸들러에서 프로그램의 상황을 점검한 후 팝업 메뉴에 속한 메뉴 항목의 Checked, Enabled, Visible 등의 속성을 변경한다.

# 4. Label

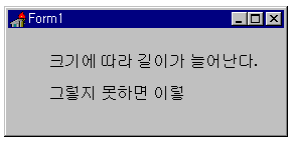
## ◘ 정의
폼 상에 단순히 문자열을 위치시키고자 할 때 사용되는 비 윈도우 컨트롤이다. 비윈도우 컨트롤이라는 뜻은 ①포커스를 가지지 못한다. ②다른 컨트롤을 포함할 수 없다. ③윈도우 핸들을 가지지 않는다는 뜻이다. 레이블 컨트롤에 표시되는 문자열은 Caption 속성으로 입력하며 Alignment 속성에 의해 레이블 내에 문자열이 어떻게 정렬될 것인가를 지정한다. 관련있는 레이블끼리 가지런하게 정렬시키는 것이 보기에 좋다. 레이블에 의해 출력되는 문자열은 사용자가 실행중에 수정하지 못한다는 특징이 있다. 그래서 자신의 캡션을 가지지 못하는 에디트 컨트롤이나 리스트 박스 컨트롤의 제목을 나타내는 용도로 많이 사용한다.



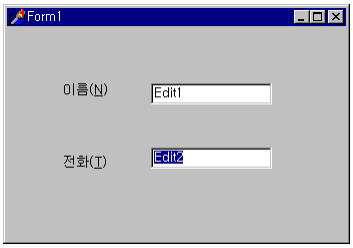

레이블의 캡션에 &를 포함할 경우 & 다음 문자가 레이블의 단축키가 되며 이 단축키가 눌러지면 레이블의 FocusControl 속성이 지정하는 컨트롤이 활성화된다. 레이블이 다른 컨트롤의 제목으로 많이 사용되므로 그 컨트롤의 단축키를 포함한다. 레이블의 AutoSize 속성을 True로 설정하면 캡션의 폰트와 길이에 따라 레이블의 크기가 자동으로 조절된다. transparent 속성은 레이블의 배경색을 투명색으로 만들어 레이블 뒤의 비트맵이나 이미지를 가리지 않도록 해준다.
가장 중요한 속성은 Caption이며 그 외 몇 가지 쓸만한 속성이 있다. 메소드는 있기는 하지만 실용성이 없으며 이벤트는 가지지 않는다.

## ◘ 속성

### AutoSize:Boolean;
True일 경우 레이블의 크기가 캡션의 길이와 폰트의 크기에 따라 자동 조절된다. 즉 캡션 문자열의 길이가 길어지거나 폰트가 커지면 레이블의 크기가 늘어난다. 디폴트값은 True로 설정되어 있으며 이 속성을 False로 변경하면 문자열의 길이에 상관없이 일정한 길이를 가지므로 문자열 일부가 보이지 않을 수도 있다.




### FocusControl:TWinControl;
프로그램 실행중에 Alt 키와 레이블의 &기호 다음 문자가 눌러질 경우 활성화될 컨트롤을 지정한다. 다음 예에서 윗쪽 레이블의 Caption은 "이름(&N)"으로 입력되어 있으며 이 레이블의 FocusControl은 우측의 Edit1으로 설정되어 있다. 그래서 실행중에 Alt-N을 누르면 Edit1으로 포커스가 이동한다. 아래쪽 레이블의 경우도 마찬가지로 Alt-T를 누르면 Edit2로 포커스가 이동하게 된다.




버튼이나 체크 박스의 경우 컴포넌트 내에 Caption 속성이 있어 &기호를 사용하여 단축키를 지정할 수 있지만 에디트나 리스트 박스의 경우는 Caption이 없으므로 레이블 컴포넌트의 단축키를 대신 사용하며 어떤 레이블의 단축키가 어떤 컴포넌트로 연결될 것인가를 FocusControl 속성으로 설정한다.

### ShowAccelChar:Boolean;

레이블의 캡션 속성에 포함되어 있는 &기호의 해석 방식을 지정한다. 이 속성이 True일 경우 &기호는 단축키를 나타내며 밑줄이 그어진 형태로 나타나지만 이 속성이 False일 경우 &기호는 단순히 &문자로만 나타날 뿐만 아니라 단축키 지정도 되지 않는다.

### Transparent:Boolean;

레이블의 배경 색상을 투명색으로 설정한다. 즉 레이블의 배경색상을 없게 만들어 레이블 뒤의 컴포넌트가 보일 수 있도록 지정한다. 비트맵이나 특별한 무늬 위에 레이블이 놓일 경우 이 속성을 True로 설정하여 비트맵이 보일 수 있도록 하는 것이 좋다. 다음 두 가지 경우를 비교해 보아라.

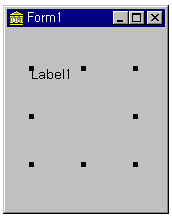

투명색을 사용하지 않을 경우-레이블이 자신만의 배경색을 가지므로 뒤쪽의 이미지가 가려진다.


투명색을 사용할 경우-뒤쪽의 이미지를 가리지 않는다.

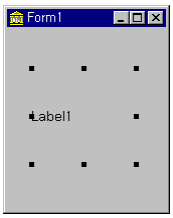
참고로 하도 질문하는 사람이 많아서 밝히는데 그림 속의 귀엽고 깜찍하고 예쁜 아가는 필자다.

### LayOut:TTextLayout;

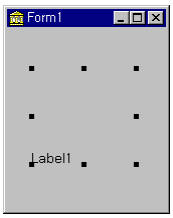
레이블의 수직 정렬을 지정한다. 디폴트값은 tlTop 이며 레이블 영역의 윗쪽에 정렬된다. 이 값을 tlBottom 이나 tlCenter 로 변경하면 수직 중앙이나 바닥에 놓을 수도 있다.




tlTop



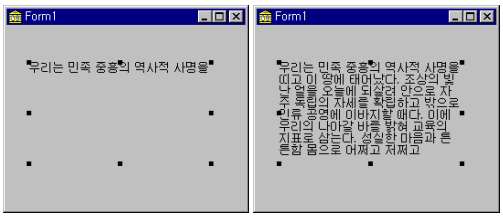

tlCenter




tlBottom

### WordWrap:Boolean;

캡션을 길게 입력했을 경우 캡션이 레이블의 오른쪽 변에 닿았을 때의 처리를 지정한다. 이 속성이 True 이면 오른쪽 변에서 자동 개행되어 다음 줄로 내려가지만 그렇지 않을 경우 뒷부분이 잘려 보이지 않게 된다. 왼쪽은 이 속성을 False 로 설정한 것이고 오른쪽은 이 속성을 True 로 설정한 것이다.

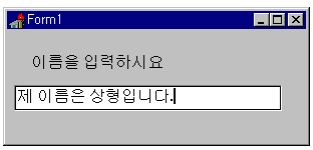



AutoSize 속성이 True 로 설정되어 있을 때 이 속성은 의미가 없다.

# 5. Edit

## ◘ 정의

사용자로부터 간단한 문자열을 입력받거나 보여주며 파일 이름, 사람 이름 등을 입력받고자 할 때 사용한다. 에디트 컨트롤이 포커스를 가질 경우 캐럿이 나타나며 키보드로부터 문장을 입력받는다.



문자열 입력중에 BS나 Del 키로 간단한 편집을 할 수 있으며 커서 이동키로 문자열 사이를 자유롭게 이동할 수 있다. 또한 마우스로 드래그하여 문자열을 선택하며 Ctrl-Ins, Shitft-Ins, Shift-Del 등의 윈도우즈에서 표준적으로 사용하는 클립보드로 복사, 붙이기, 오리기를 수행할 수 있다. 에디트 위에서 마우스 오른쪽 버튼을 누르면 팝업 메뉴가 나타나는데 이 메뉴에도 클립보드 조작 명령이 포함되어 있다.

Text 속성은 에디트 컨트롤에 입력되어진 문장을 나타내며 실행중에 이 속성을 읽으면 사용자가 입력한 문장을 얻을 수 있다. 또한 실행중에 Text 속성에 문자열을 대입하여 에디트 컴포넌트의 문장을 변경하는 것도 가능하다. 에디트 컨트롤이 입력받을 수 있는 문자의 한계는 MaxLength 속성으로 지정하되 디폴트로 설정된 값은 0이며 이는 입력 한계가 없다는 뜻이다. Modified 속성을 조사하면 에디트 컨트롤의 문장이 최후 수정 후 변경되었는가의 여부를 알 수 있다. 에디트 컨트롤의 문장은 사용자가 실행중에 언제든지 변경할 수 있다. 만약 문장을 단순히 보여주기만 하고 입력을 받을 필요가 없을 때는 ReadOnly 속성을 True로 설정하여 사용자가 실행중에 편집할 수 없도록 만든다.

## ◘ 속성

### AutoSelect:Boolean;

에디터 컨트롤이 포커스를 가질 때 텍스트가 선택될 것인가 아닌가를 지정한다. 이 값이 True일 경우 Tab 키를 눌러 에디트 컨트롤 선택시 텍스트가 선택 및 반전되어 곧바로 다른 새로운 텍스트를 입력할 수 있도록 해주며 이 값이 False일 경우 기존의 텍스트를 수정할 수 있도록 해준다. 단 마우스를 이용하여 에디트 컨트롤로 포커스를 이동시켰을 경우는 이 속성이 True이더라도 문장이 선택되지 않는다.

### CharCase:TEditCharCase;

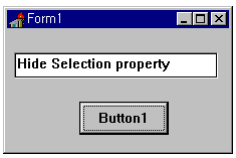
에디트 박스에 입력되는 텍스트의 대소문자 형태를 지정한다. 다음 중 하나의 값을 가진다.

| 속성값 | 의미 |
|---|---|
| ecLowerCase | 소문자로 출력된다. |
| ecNormal | 소문자와 대문자가 혼합되어 출력된다. |
| ecUpperCase | 대문자로 출력된다. |

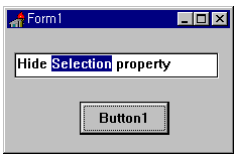
사용자가 입력한 문자가 이 속성과 다를 경우 입력된 문자는 이 속성이 지정한대로 변경되어 출력된다. 예를 들어 이 속성이 ecUpperCase일 때 사용자가 소문자를 입력해도 대문자로 나타난다. 암호 입력, 비밀 번호 입력 등과 같이 대소문자 구분을 확실하게 해야 하는 경우 이 속성을 사용한다.

### HideSelection:Boolean;

에디트 컨트롤에서 선택한 문장은 파란색의 역상으로 나타난다. 이 속성은 역상 표시가 포커스를 잃은 후에도 계속 유지될 것인가 아닌가를 지정한다. 이 속성이 True일 경우 컨트롤이 포커스를 잃으면 역상 표시가 사라지며 컨트롤이 다시 포커스를 가져야 역상 표시가 나타난다.




이 값이 True일 경우 포커스가 버튼으로 이동하면 선택 영역이 보이지 않는다.




이 값이 False일 경우 포커스가 버튼으로 이동해도 역상 표시가 계속 유지된다.

### MaxLength:Integer;

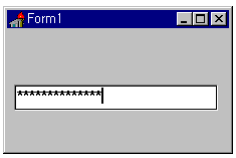
에디트 컨트롤에 입력할 수 있는 최대 문자수를 나타낸다. 디폴트값은 0이며 이는 입력 문자수에 제한이 없다는 뜻이다. 0 이외의 다른 값으로 지정하면 입력할 수 있는 문자수의 한계를 지정하며 입력 한계를 초과할 경우는 비프음을 내고 더 이상의 입력을 받아들이지 않는다. 디렉토리 경로나 파일 이름 등과 같이 길이의 한계가 있는 정보를 입력받을 때는 사용자가 문자열 길이를 초과하여 입력하는 것을 금지시키는 것이 좋다.

### 실 Modified:Boolean;

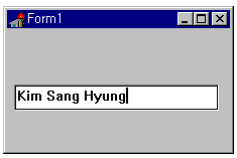
에디트 컨트롤에 입력된 문장이 사용자에 의해 변경되었는지를 조사한다. 이 값이 True이면 최초 컨트롤을 생성한 후, 또는 Modified가 False로 설정된 이후 텍스트가 변경되었다는 뜻이다.

### PasswordChar:Char;

에디트 컨트롤에 문자를 입력받을 때 출력되는 문자의 형태를 지정한다. 이 속성은 디폴트가 0으로 설정되어 있으므로 입력된 문자가 그대로 출력된다. 이 속성에 특별한 다른 문자를 지정하면 입력되는 문자를 모두 이 문자로 바꾸어 출력한다. 암호를 입력할 때 이 속성을 사용한다.




이 속성을 *로 설정한 경우. 입력되는 문자를 모두 *로 바꾼 후 보여준다.




이 속성을 0으로 지정한 경우(디폴트) 입력한 문자를 그대로 보여준다.

단 암호 문자를 설정하더라도 사용자에게 보여주는 문자만 바뀔 뿐이지 입력한 내용이 바뀌는 것은 아니다. 즉 Text 속성을 읽으면 사용자가 입력한 내용을 그대로 읽을 수 있다.

### ReadOnly:Boolean;

사용자가 실행중에 에디트 컨트롤의 문장을 수정할 수 있는지 없는지를 나타낸다. 이 값이 True일 경우 사용자는 에디트 컨트롤의 문장을 편집할 수 없으며 False일 경우 키보드를 사용하여 편집할 수 있다. 디폴트값은 False이다. 사용자가 함부로 변경해서는 안되는 정보를 가지는 에디트 컨트롤은 이 속성을 설정하여 편집을 불가능하게 만들어 두도록 한다.

### 실 SelStart:Integer;

사용자가 마우스 드래그나 Shift-커서 이동키를 사용하여 에디트에 입력된 문자열 중 일부를 선택할 때 선택한 시작 위치를 나타낸다. 예를 들어 첫 번째 문자부터 선택되어 있다면 이 속성값은 0이다. 이 값은 실행중에 변경할 수 있지만 단 에디트 박스가 포커스를 가지고 있을 경우에만 변경할 수 있으며 만약 포커스를 가지고 있지 않은 상태에서 이 속성을 변경할 경우 변경되지 않는다.

### 실 SelLength:Integer;

사용자가 블럭으로 선택한 문자열의 길이를 가진다. SelStart 속성이 변경되면 이 속성도 같이 변한다. 이 값은 실행중에 변경할 수 있지만 단 에디트 박스가 포커스를 가지고 있을 경우에만 변경할 수 있으며 만약 포커스를 가지고 있지 않은 상태에서 이 속성을 변경할 경우 변경되지 않는다.

### 실 SelText:String;

사용자가 실행중에 선택한 문자열을 가진다. 이 속성을 읽음으로써 선택된 문자열을 알 수 있을 뿐만 아니라 이 속성에 새로운 문자열을 대입하여 선택된 문자열을 다른 문자열로 변경할 수도 있다. 현재 선택된 문자열이 없는 상태에서 이 속성에 문자열을 대입하면 커서가 있는 위치에 새로운 문자열이 삽입된다.

### 실 읽 CanUndo:Boolean;

최후 편집 동작을 취소할 수 있는지 없는지를 나타낸다. 이 속성은 주로 Edit/Undo 메뉴 항목의 상태를 변경하기 위해 사용한다.

## ◘ 메소드

■ procedure Clear;
text 속성을 모두 지운다. text:='';과 동일한 동작을 한다.

■ procedure ClearSelection;
선택된 문자열을 삭제한다. 선택된 부분이 없으면 아무런 동작도 하지 않는다.

■ procedure CopyToClipboard; procedure CutToClipboard;
선택된 영역을 클립보드로 복사(또는 잘라내기)한다. 클립보드에 있던 원래 데이터는 파괴된다. 선택된 부분이 없으면 아무런 동작도 하지 않는다.

■ function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;
text 속성을 읽어 Buffer가 가리키는 메모리 영역으로 복사한다. BufSize 인수는 버퍼의 길이를 지정하며 복사한 문자의 수를 리턴해 준다. Buffer 메모리에 있는 문자열은 널 종료 문자열이다.

■ function GetTextLen: Integer;
text 속성의 길이를 리턴한다.

■ function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer;
선택된 영역의 문자열을 Buffer 메모리 영역으로 복사한다. BufSize 인수는 버퍼의 길이를 지정하며 복사한 문자의 수를 리턴해 준다.

■ procedure PasteFromClipboard;
클립보드에 있던 데이터를 현재 삽입점이 있는 위치에 삽입한다.

■ procedure SelectAll;
text 속성의 문자열 전체를 선택 영역으로 지정한다. 문자열 중 일부만 선택 영역으로 지정하려면 SelStart, SelLength 속성을 사용해야 한다.

■ procedure SetFocus;
포커스를 가지도록 활성화시킨다.

■ procedure SetSelTextBuf(Buffer: PChar);
선택된 영역의 문자열을 Buffer 메모리에 있는 널 종료 문자열로 변경한다.

■ procedure SetTextBuf(Buffer: PChar);
Buffer 메모리에 있는 문자열로 text 속성을 변경한다. Buffer에 있는 문자열은 반드시 널 종료 문자열이어야 한다.

## ◘ 이벤트

### OnChange

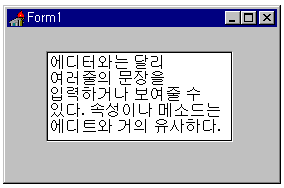
에디트의 Text가 변경될 때 발생하는 이벤트이며 이 이벤트가 발생했을 때 Text는 이미 변경된 상태이다. 실제로 변경되었는

지는 Modified 속성을 읽어 보면 알 수 있다. 에디트의 내용이 변경될 때마다 특정한 작업을 해야 한다면 이 이벤트 핸들러에 원하는 코드를 작성한다.

# 6. Memo

## ◘ 정의

에디트 컴포넌트와 동일한 기능을 가진다. 즉 문장을 보여주며 실행중에 사용자가 문장을 편집할 수 있도록 해준다. 에디트 컴포넌트와 다른 점은 여러 줄을 보여주거나 편집할 수 있다는 점이다.




입력된 문장은 Text 속성으로 나타나며 문자의 변경 여부는 Modified 속성으로 조사하며 최대 입력 한계는 MaxLength 속성으로 지정한다. 사용자는 메모 컴포넌트의 문장을 BS나 Del 키로 간단한 편집을 할 수 있으며 블럭 조작을 수행할 수도 있다. Text 속성 외에도 Lines 속성을 사용하여 개별 문장을 읽거나 변경하는 것도 가능하다. Text 속성은 전체 문장을 읽거나 쓸 때 사용하며 Lines 속성은 개별 행을 읽거나 쓸 때 사용한다. 첫 번째 행을 읽고 싶으면 Lines[0]를 읽으면 되고 두 번째 행을 읽고 싶다면 Lines[1]을 읽으면 된다. Lines 속성은 문자열 리스트이며 Add, Delete, Insert 등의 메소드를 사용하여 개별 행을 삭제하거나 삽입할 수 있다.

다수의 속성이 에디트 컴포넌트와 중복되므로 SelLength, SelText, SelStart, MaxLength, Modified 등의 속성은 에디트를 참조하기 바란다. 메모는 TCustomEdit로부터 파생되므로 에디트에서 가능한 일은 메모에서도 대부분 가능하다. 메모는 에디트가 가지지 않는 속성 몇 가지를 추가로 가지지만 메소드나 이벤트는 에디트의 그것들과 완전히 동일하다.

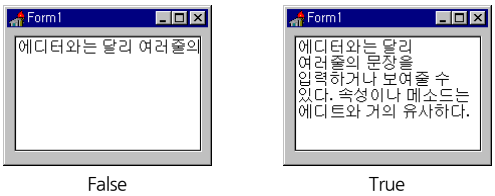
## ◘ 속성

### Scrollbars:TScrollStyle;

메모 컴포넌트가 스크롤 바를 가질 것인지 아닌지를 지정하며 다음 4가지 중 하나의 값이 가능하다. 디폴트는 ssNone이며 스크롤 바가 없다.


### WordWrap:Boolean;

문장이 메모 컴포넌트의 우측 한계를 벗어날 경우 자동 개행할 것인지 아닌지를 지정하며 디폴트는 True이다. WordWrap 속성이 False일 때 문장이 우측으로 벗어날 경우 스크롤 되며 개행하려면 Enter 키를 입력해야 한다. WordWrap 속성이 True일 때 문장이 메모의 우측변에 이르면 자동으로 개행되고 따라서 수평 스크롤 바는 불필요하다.




### WantTabs:Boolean;

메모 컴포넌트의 문장 편집시 Tab 키를 사용할 것인지 아닌지를 지정하며 디폴트는 False이다. Tab 키는 컴포넌트간 포커스를 이동시키는 용도로 사용되므로 문장 편집중에 사용하지 못한다. 이 속성을 True로 설정하면 문장 편집시 Tab 키를 사용할 수는 있지만 다른 컴포넌트로 포커스를 옮길 때는 Tab 키를 사용할 수 없다. 이 속성이 False일 때 Tab 코드를 입력하려면 Ctrl-Tab을 눌러야 한다.

### WantReturns:Boolean;

메모 컴포넌트의 문장 편집시 Enter 키를 사용할 것인지 아닌지를 지정하며 디폴트는 True이다. 만약 이 속성이 False일 경우 Enter 키 입력은 메모 컴포넌트가 받지 않고 폼이 받게 되며 폼의 디폴트 버튼의 OnClick 이벤트가 발생한다. WantReturns 속성이 False인 상태에서 Enter 키를 입력하려면 Ctrl-Enter 키를 눌러야 한다.

### MaxLength:Integer;
메모 컨트롤에 입력할 수 있는 최대 문자수를 나타낸다. 디폴트 값은 0이며 이는 입력 문자수에 제한이 없다는 뜻이다. 0 이외의 다른 값으로 지정하면 입력할 수 있는 문자수의 한계를 지정한다.

### Modified:Boolean;
메모 컨트롤에 입력된 문장이 사용자에 의해 변경되었는지를 조사한다. 실행시에만 사용되며 이 값이 True이면 최초 컨트롤을 생성한 후, 또는 Modified가 False로 설정된 이후 텍스트가 변경되었다는 뜻이다.

## 7. Button

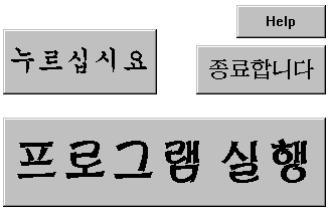
## ◘ 정의
버튼은 사용자가 마우스로 의사 전달을 하는 가장 보편적인 장치이다. 프로그램의 질문에 응답하거나, 새로운 작업을 지시하거나 취소할 때 주로 사용된다.


원하는 위치에 원하는 크기로 배치할 수 있으며 Caption 속성을 사용하여 버튼의 제목을 지정한다. 버튼에서 발생할 수 있는 이벤트 중 가장 보편적인 것이 OnClick 이벤트이며 이는 사용자가 버튼을 마우스로 클릭할 때 발생하므로 이 이벤트에 특정한 코드를 기술하여 버튼의 기능을 정의한다. 버튼의 Default 속성과 Cancel 속성을 이용하면 디폴트 버튼과 취소 버튼을 쉽게 만들 수 있는데 Default 속성이 지정된 버튼은 Enter 키를 누를 때 OnClick 이벤트가 발생하며 Cancel 속성이 지정된 버튼은 Esc 키를 누를 때 OnClick 이벤트가 발생한다.

## ◘ 속성

### Caption:String;
버튼 위에 나타날 문자열이며 이 문자열이 버튼의 제목이 된다. Caption 문자열에 &기호가 있을 경우 & 다음의 문자가 단축키로 지정된다. 한글도 버튼의 캡션으로 사용할 수 있으나 단축키는 반드시 영문으로 지정해야 하므로 괄호 안에 적당한 영문자 하나를 입력해 둔다. 다음 그림의 좌측 버튼 캡션은 &Open이며 우측 버튼 캡션은 열기(&O)이다.


### Font:TFont;
버튼의 Caption 글꼴을 변경한다. 글꼴 이름, 모양, 크기, 색상 등의 세부 속성을 가지며 오브젝트 인스펙터에서 이 속성을 더블클릭하면 글꼴 선택 대화상자가 열린다. Caption과 Font 속성에 따라 다양한 모양의 버튼을 디자인한다.




### ModalResult:TModalResult;
Modal형 폼에 있는 버튼의 ModalResult 속성이 0 이외의 값일 경우 이 버튼이 눌러지면 폼이 종료된다. Ok나 Cancel 등과 같이 폼을 닫는 기능을 가진 버튼은 ModalResult 속성을 MrOk나 MrCancel로 설정함으로써 별도의 코드를 작성할 필요없이 버튼이 눌러지면 대화상자나 폼을 닫도록 한다. 버튼의 ModalResult 속성은 폼을 호출한 ShowModal 함수의 리턴값이 된다. 디자인시와 실행시에 모두 사용할 수 있지만 실행시에 값을 변경할 수는 없다.

### Default:Boolean;
이 속성이 True일 경우 버튼을 디폴트 버튼으로 만든다. 디폴트 버튼이란 Enter 키가 입력될 경우 OnClick 이벤트 핸들러가 자동으로 실행되는 키를 말한다. 디폴트 버튼은 다른 버튼들보다 테두리가 굵게 그려진다. 여러 개의 버튼이 디폴트 버튼으로 지정되어 있을 경우 탭 순서가 제일 빠른 버튼의 코드가 실행되며 나머지는 무시된다.

### Cancel:Boolean;
이 속성이  True일 경우 버튼을 취소 버튼으로 만든다. 취소 버튼이란 Esc 키가 입력될 경우 OnClick 이벤트 핸들러가 자동으로 실행되는 키를 말한다. 예를 들어 다음과 같은 버튼이 폼에 있으며 이 버튼의 Cancel 속성이 True로 설정되어 있다고 하자.

그리고 이 버튼의 OnClick 이벤트에 폼을 종료하는 Close라는 코드가 기입되어 있다.

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

그러면 사용자는 이 버튼을 클릭하여 폼을 종료할 수도 있지만 키보드의 Esc 키를 눌러 폼을 종료할 수도 있다. 여러 개의 버튼이 취소 버튼으로 지정되어 있을 경우 탭 순서가 제일 빠른 버튼의 OnClick 이벤트가 실행된다.

### Enabled:Boolean;

버튼을 사용할 수 있게 할 것인가 사용을 금지시킬 것인가를 지정한다. 이 속성의 디폴트값은 True이므로 생성된 모든 버튼을 사용할 수 있지만 이 속성이 False이면 버튼이 흐릿하게 표시되며 마우스, 키보드 등의 입력을 받아들이지 않는다.


잠시 기능을 정지시켜 두어야 할 버튼에 이 속성을 사용한다.

### Visible:Boolean;

이 속성을 False로 설정하면 아예 버튼이 보이지 않게 된다.

## ◘ 메소드

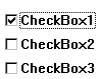
### OnClick

마우스로 버튼을 눌렀을 때와 같이 OnClick 이벤트를 발생시킨다. 코드로 버튼을 누른 효과를 내고자 할 때 이 메소드를 호출한다.

# 8. CheckBox

## ◘ 정의

사용자로부터 옵션을 입력받고자 할 때 사용하며 체크 박스를 클릭하면 선택/비선택이 교체된다.




사용자의 클릭에 의해 체크 박스의 상태가 토글되며 Checked 속성이 변경된다. 선택된 체크 박스에는 V표시가 나타난다. 체크 박스 옆에는 체크 박스의 의미를 나타내는 문장을 기입할 수 있으며 Caption 속성에 문장을 입력한다. 체크 박스의 상태를 읽고자 할 때는 State 속성을 읽으며 이 속성을 변경하여 체크 박스의 상태를 바꾼다. 체크 박스를 클릭하면 OnClick 이벤트가 발생한다.

## ◘ 속성

### AllowGrayed:Boolean;

체크 박스의 상태는 Checked, unChecked 두 가지가 있으며 사용자가 체크 박스를 클릭하면 두 상태가 토글되는 것이 일반적이다. 그러나 AllowGrayed 속성이 True이면 두 가지 상태 외에도 선택도 비선택도 아닌 gray 상태가 존재하며 세 가지 상태가 교체된다.

### State:TCheckState;

체크 박스의 상태를 읽거나 설정하며 다음 세 가지 값을 사용한다.

| 속성값 | 의미 |
|---|---|
| cbUnchecked | 체크되지 않은 상태 |
| cbChecked | 체크된 상태 |
| cbGrayed | 흐리게 표시된 상태이며 이 상태의 의미는 프로그램에서 정의하기 나름이다. |

### Checked:Boolean;

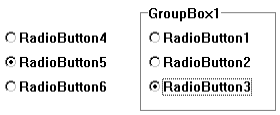
체크 박스의 체크 상태를 나타낸다. True이면 체크 박스가 선택된 상태이며 False이면 체크 박스가 선택되지 않은 상태이다.

# 9. RadioButton

## ◘ 정의

상호 배타적인 옵션을 입력받을 때 사용한다. 여러 개의 라디오 버튼 중에 반드시 하나의 버튼만 선택할 수 있으며 사용자가 라디오 버튼 중 하나를 클릭하여 선택하면 이전에 선택되어 있던 라디오 버튼은 선택 취소된다.


라디오 버튼의 의미를 나타내는 문자열은 라디오 버튼의 우측에 나타나며 Caption 속성에 입력한다. 사용자가 라디오 버튼을 클릭하면 Checked의 속성이 변경되며 OnClick 이벤트가 발생한다. 라디오 버튼은 상호 배타적으로 선택되므로 반드시 그룹을 지어서 사용해야 한다. 폼, 그룹 박스, 패널 등의 동일 컨테이너 컴포넌트에 속해있는 라디오 버튼은 하나의 그룹이 되며 상호 배타적이다. 예를 들어 패널 컴포넌트안에 다섯 개의 라디오 버튼이 있을 경우 이 다섯 개의 라디오 버튼은 상호 배타적으로 선택된다. 반면 패널 컴포넌트에 있는 라디오 버튼과 그룹 컴포넌트에 있는 라디오 버튼끼리는 같은 그룹이 아니므로 상호 선택에 영향을 받지 않는다. 다음 그림은 폼에 있는 라디오 버튼과 그룹 박스에 있는 라디오 버튼을 보인 것이다.




폼에 있는 라디오 버튼과 그룹 박스에 있는 라디오 버튼들은 상호 선택에 영향을 미치지 않는다.

## ◘ 속성

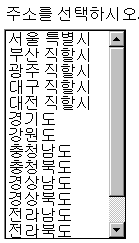
### Checked:Boolean;

라디오 버튼의 선택 상태를 나타낸다. 이 속성이 True이면 라디오 버튼 주위에 검정색 원이 나타나 옵션이 선택되어 있음을 보여준다. False이면 옵션이 선택되지 않은 것이다.

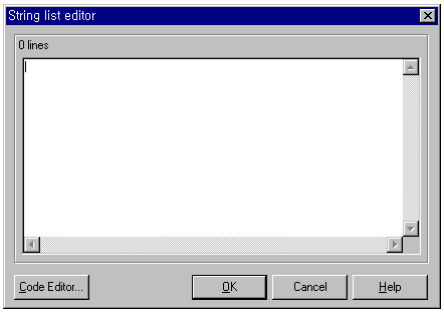
# 10. ListBox

## ◘ 정의

직사각형의 모양을 가지며 사용자가 선택할 수 있는 여러 가지 항목들을 나열해 두고 마우스로 선택할 수 있도록 해준다. 리스트 박스가 없다면 사용자가 직접 키보드로 문자열을 입력해 주어야만 하므로 불편하다.




리스트 박스에 포함되는 항목은 문자열이며 Items 속성에 입력된다. 디자인시에 리스트 박스에 문자열을 입력하려면 오브젝트 인스펙터에서 Items 속성을 더블클릭한다. 다음과 같이 문자열을 입력할 수 있는 문자열 리스트 편집기가 열린다.




Items 속성은 문자열 리스트이므로 실행중에 Add, Insert, Delete 등의 메소드를 사용하여 실행중에 항목들을 추가, 삭제할 수 있다. 사용자에 의해 선택된 항목의 번호는 ItemIndex 속성을 조사하여 알 수 있으며 ListBox1.Items[ListBox1.ItemIndex]로 선택된 항목을 읽어낸다.

## ◘ 속성

### 실 ItemIndex:Integer;

리스트 박스에서 선택된 항목의 번호이다. 첫 번째 항목의 번호가 0번이므로 n번째 항목이 선택되었다면 ItemIndex 속성은 n-1이 된다. 예를 들어 10번째 항목이 선택되었다면 이 속성값은 9가 된다. 선택된 항목이 하나도 없을 경우 이 속성값은 -1이다. 리스트 박스에서 사용자가 선택한 문자열을 읽으려면 ListBox1.Items[ListBox1.ItemIndex]를 읽는다. 참고로 리스트 박스에 포함된 문자열의 개수를 알려주는 속성은 리스트 박스에는 없으며 Items 속성의 Count 속성을 읽어야 한다. 즉 ListBox1.Items.Count를 읽어야 한다.

### Columns:Longint;

리스트 박스는 디폴트로 박스 내의 항목을 1열로 보여준다. 이 속성을 2나 3으로 변경하여 항목을 여러 단으로 보여주도록 할

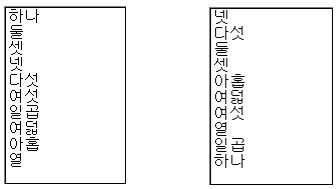
수 있다. 주의할 것은 이 속성이 0인 경우와 1인 경우는 의미가 다르다는 점이다.


0 1 2

이 속성이 0이면 수직으로 스크롤 바를 나타내지만 1이면 리스트 박스의 수직 크기에 맞게 1열씩 보여준다. 디폴트값은 0이다.

## Sorted:Boolean;

리스트 박스 내의 항목들이 알파벳순으로 정렬되도록 한다. 이 속성의 디폴트값이 False이므로 항목들은 리스트 박스에 추가된 순서대로 나타나지만 항목의 개수가 많을 경우는 정렬되도록 하는 것이 검색하기 편리하다. Sorted 속성이 True인 상태에서 새로운 항목이 추가될 경우 추가된 항목은 정렬된 위치에 삽입된다. 다음에 정렬된 경우와 정렬되지 않은 경우를 비교해 보아라.




정렬 전 정렬 후

주의할 것은 Sorted 속성은 항목을 보여주는 순서를 정렬하는 것이 아니라 실제로 문자열 리스트의 배치 상태를 정렬한다. 따라서 일단 Sorted 속성을 True로 설정하면 다시 False로 바꾸어도 원래대로 돌아오지 않는다.

## MultiSelect:Boolean;

사용자는 리스트 박스의 항목 중 하나만을 선택할 수 있다. 그러나 이 속성이 True이면 여러 개의 항목을 선택할 수도 있다.


여러 개의 항목을 선택하는 방법은 ExtendSelect 속성에 따라 달라진다.

## ExtendSelect:Boolean;

MultiSelect 속성과 함께 사용되며 MultiSelect 속성이 False일 경우 이 속성은 의미가 없다. MultiSelect 속성과 이 속성이 동시에 True이면 사용자는 항목들을 선택할 때 Shift 키와 Ctrl 키를 사용할 수 있다. Shift 키는 연속되는 항목들을 선택할 때 사용하며 Ctrl 키는 비 연속적인 항목들을 선택할 때 사용한다. MultiSelect 속성은 True이고 이 속성이 False이면 Shift 키나 Ctrl 키없이 여러 항목들을 선택할 수 있으나 일정 범위의 항목들을 한꺼번에 선택할 수는 없다.

## 실 읽 Selected[i]:Boolean;

여러 개의 항목을 선택할 경우 이 속성을 읽어 어떤 항목이 선택되어 있는지를 조사한다. Boolean형의 배열 형태를 띠고 있으며 조사하고자 하는 항목을 첨자로 사용한다. 예를 들어 세 번째 항목의 선택 여부를 알고 싶으면 Selected[2]의 값을 조사해 보면 된다. 첫 번째 항목의 인덱스가 0이다.

## 실 읽 SelCount:Integer;

여러 개의 항목을 선택할 수 있을 때 선택된 항목의 개수를 가진다.

## Style:TListBoxStyle;

리스트 박스의 형태를 지정한다. 디폴트로 문자열 항목만 담을 수 있지만 Style을 변경하면 그래픽 항목을 포함할 수도 있다.

| 속성값 | 의미 |
|---|---|
| lbStandard | 모든 항목은 문자열이며 같은 높이를 가진다. |
| lbOwnerDrawFixed | 그래픽 항목을 가질 수 있으며 각 항목의 높이는 같다. |
| lbOwnerDrawVariable | 그래픽 항목을 가질 수 있으며 각 항목의 높이는 다양하다. |

## TabWidth:Integer;

리스트 박스내에서 탭 문자의 폭을 설정한다. 이 값이 0 이면 디폴트로 지정된 2DLU 가 적용되며 별도로 지정하면 탭문자가 지정한만큼의 폭을 가지게 된다. 여기서 DLU 란 대화상자에서 사용하는 크기 단위를 말한다.

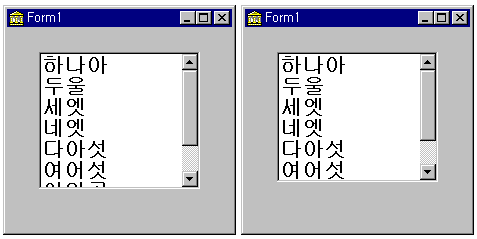
## TopIndex:Integer;

리스트 박스의 제일 위에 보이는 항목의 인덱스를 조사한다. 이 값은 현재의 스크롤 상태에 따라 달라진다. 예를 들어 10 개의 항목이 있는데 이 값이 3 이면 위로 스크롤된 항목이 0,1,2 세 개가 있다는 것을 알 수 있다. 또 이 속성을 변경하면 리스트 박스를 강제로 스크롤시킬 수 있다.

### IntegralHeight:Boolean;

리스트 박스 항목의 일부분이 잘려서 보이도록 할 것인지 아니면 항목의 높이에 맞추어 리스트 박스 크기를 조정할 것인지를 지정한다. 이 속성이 True 이면 리스트 박스의 높이는 개별 항목 높이 배수가 되도록 강제 조정된다. 이 값이 False 일 경우는 리스트 박스의 높이를 변경하지 않으며 대신 제일 아래쪽 항목이 일부만 보일 수도 있다.

예를 들어 하나~열까지 문자열 항목이 입력되어 있는 상태에서 이 속성을 변경해 보자. 왼쪽은 IntegralHeight 를 False 로 설정했기 때문에 아래쪽의 "이일곱" 문자열의 일부분만 보이지만 오른쪽은 이 속성을 True 로 설정하여 리스트 박스 높이를 강제로 조정하여 아래쪽 항목을 아예 보이지 않도록 하였다.




IntegralHeight 속성이 True 인 상태에서 리스트 박스의 크기를 조정해 보면 반드시 항목 높이의 배수에만 크기가 맞추어진다는 것을 알 수 있다.

## ◘ 메소드

■ procedure Clear;

Lines 속성 전체를 지운다.

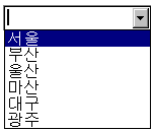
■ function ItemAtPos(Pos: TPoint; Existing: Boolean): Integer;

Pos 위치에 있는 항목의 번호를 조사하여 리턴해 준다. Existing이 True일 경우 Pos가 리스트 박스 외부의 좌표이면 -1을 리턴하며 Existing이 False일 경우 마지막 항목+1을 리턴해 준다.

# 11. ComboBox

## ◘ 정의

에디트 컴포넌트와 리스트 박스를 결합시켜 놓은 컴포넌트이다. 에디트 박스에서 직접 문자열을 입력할 수도 있으며 리스트 박스에서 항목을 선택할 수도 있다.




직접 입력하거나

항목중 하나를 선택한다.

에디트에서 입력한 내용, 또는 리스트 박스에서 선택한 항목은 Text 속성에 나타나며 리스트 박스에 포함된 항목은 Items 문자열 리스트를 사용하여 읽거나 쓸 수 있다. Items 속성은 문자열 리스트이므로 Add, Insert, Delete 등의 메소드를 사용하여 실행중에 항목들을 추가, 삭제할 수 있다. 사용자에 의해 선택된 항목의 번호는 ItemIndex 속성을 조사하여 알 수 있으며 ComboBox1.Items[ComboBox1.ItemIndex]로 읽어낸다. 문자열 리스트를 입력하거나 읽어내는 방법은 리스트 박스와 동일하므로 리스트 박스를 참조하기 바라며 에디트 박스에 입력된 문자열을 관리하는 방법은 에디트 컴포넌트와 동일하므로 에디트 박스를 참조하기 바란다.

## ◘ 속성

### Sorted:Boolean;

콤보 박스 내의 항목들이 알파벳 순으로 정렬되도록 한다. 이 속성의 디폴트값이 False이므로 항목들은 콤보 박스에 추가된 순서대로 나타나지만 항목의 개수가 많을 경우는 정렬되도록 하는 것이 검색하기 편리하다. Sorted 속성이 True인 상태에서 새로운 항목이 추가될 경우 추가된 항목은 정렬된 위치에 삽입된다.

### Style:TComboBoxStyle;

콤보 박스의 형태를 지정한다. 다음중 한 값을 가진다.

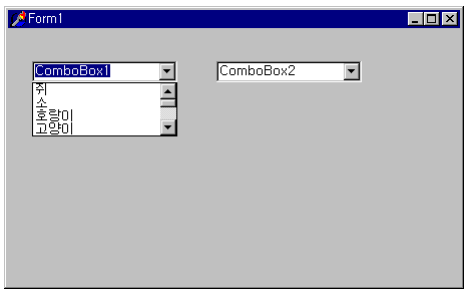
| 속성값 | 의미 |
|---|---|
| csDropDown | 에디트 박스와 리스트 박스를 결합한 형태이며 에디트 박스에 입력하거나 리스트 박스에서 선택한다. 리스트 박스에서 항목을 선택하면 선택한 항목이 에디트 박스로 출력된다. 리스트 박스의 모든 항목은 문자열이다. |
| csSimple | 리스트 박스를 가지지 않으며 에디트 박스만 가진다. 에디트 박스에 직접 입력하는 수밖에 없다. |

| | |
|---|---|
| csDropDownList | 에디트 박스는 가지지 않으며 리스트 박스만을 가진다. 따라서 사용자는 에디트 박스에 새로운 항목을 직접 입력할 수는 없으며 리스트 박스에서 항목을 선택하는 수밖에 없다. 리스트 박스에서 항목을 선택하면 선택한 항목이 에디트 박스로 출력되며 에디트 박스를 클릭하면 리스트 박스가 아래로 펼쳐진다. 리스트 박스와 기능상 동일하지만 자리를 많이 차지하지 않는다는 장점이 있다. |
| csOwnerDrawFixed | 그래픽 항목을 가질 수 있으며 각 항목의 높이는 같다. |
| csOwnerDrawVariable | 그래픽 항목을 가질 수 있으며 각 항목의 높이는 다양하다. |

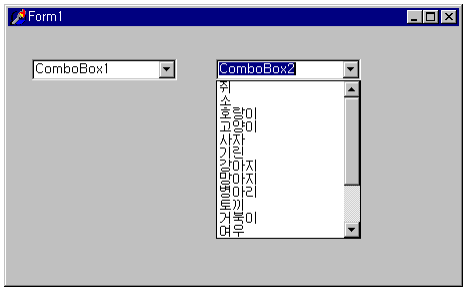
### DropDownCount:Integer;

콤보 박스를 펼쳤을 때 아래로 펼쳐질 목록의 개수를 설정한다. 디폴트로 이 값은 8 이므로 8 개 항목을 표시할만큼의 목록이 열린다. 항목이 아주 많을 경우는 이 값을 충분히 늘려주어 한눈에 많은 항목이 보이도록 해 주는 것이 좋다.

다음 예제는 두 개의 콤보 박스를 배치해 두고 똑같은 Items 를 입력해 준 상태에서 DropDownCount 만 각각 4 와 12 로 설정해 본 것이다.




DropDownCount가 4로 설정된 경우는 펼쳐지는 목록의 높이가 네칸밖에 되지 않아 목록을 선택하기가 불편하며 일일이 스크롤해가며 항목을 봐야 한다. 반면 12로 설정된 경우는 한번에 많은 항목을 볼 수 있기 때문에 다소 편리하다.




단 DropDownCount에 너무 높은 값을 설정하면 화면을 많이 차지하게 된다는 단점이 있다. 만약 이 속성이 항목수보다 더 클 경우는 항목수에 맞추어지며 Style 속성이 csSimple일 경우는 의미가 없다.

### DropedDown:Boolean;

콤보 박스의 목록이 열려 있는지 검사하며 또한 이 속성을 True 로 변경하여 목록이 열리도록 할 수 있다. Style 속성이 csSimple일 경우 이 속성은 의미가 없다.

## ◘ 이벤트

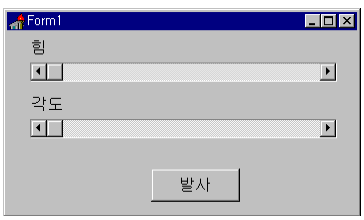
### OnDropDown

콤보 박스의 목록이 열릴 때 이 이벤트가 발생하는데 잘 사용되지 않는 이벤트이다.

# 12. ScrollBar

## ◘ 정의

윈도우의 작업 영역을 스크롤시키거나 정수값을 신속하게 입력받을 때 사용한다.




사용자는 스크롤 바의 몸통 부분을 클릭하거나 양쪽 끝의 화살표를 클릭하여 값을 증감시킬 수 있으며 또는 썸을 드래그하여 값을 신속하게 변경시킬 수도 있다. 스크롤 바가 클릭되면 OnScroll 이벤트가 발생한다. 스크롤 바가 가지는 값의 범위나 증감의 정도는 속성값에 의해 지정되며 스크롤 바가 가지는 값

도 Position 속성으로 읽을 수 있다.

## ◘ 속성

### Position:Integer;

스크롤 바의 썸 위치를 나타내며 이 값이 곧 스크롤 바가 가지는 값이다. 사용자가 스크롤 바를 드래그하거나 클릭하면 이 값이 변하게 된다. 또는 이 값을 실행중에 변경하여 스크롤 바의 썸을 강제로 이동시킬 수도 있다.

### Kind:TScrollBarKind;

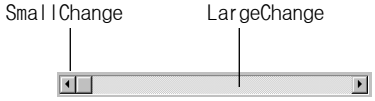
스크롤 바의 모양을 지정하며 수평, 수직 두 가지 중 하나의 값을 선택한다. 디폴트는 수평이다.


### Min,Max:Integer;

스크롤 바가 가질 값의 범위를 지정한다. Min이 최소값, Max가 최대값이며 디폴트값이 0, 100이므로 스크롤 바가 가지는 값의 범위는 0~100이 된다. 즉 수평 스크롤 바의 경우 썸이 좌측 끝에 있으면 0의 값을 가지고 우측 끝에 있으면 100의 값을 가진다. 스크롤 바가 가지는 의미에 따라 범위를 적당히 조절하도록 한다. 예를 들어 각도를 입력받는다면 0~360까지의 범위를 지정하며 색상값이면 0~255까지의 범위를 지정한다. 실행중에 스크롤 바의 범위를 변경하려면 SetParams 메소드를 사용하여 최대, 최소값을 한꺼번에 변경한다.

### SmallChange,LargeChange:TScrollBarInc;

스크롤 바가 클릭될 경우 변한 값의 크기를 지정한다. SmallChange는 스크롤 바 양쪽의 화살표를 클릭할 때의 증감분이며 LargeChange는 스크롤 바의 몸통 부분을 클릭할 때의 증감분이다.




## ◘ 메소드

■ procedure SetParams(APosition, AMax, AMin: Integer);
Position, Max, Min 속성을 한꺼번에 변경하고자 할 때 이 메소드를 사용한다.

## ◘ 이벤트

### OnChange

Position 속성값이 변경되었을 때, 즉 스크롤 바의 현재 위치가 변경되었을 때 발생한다.

### OnScroll

사용자가 마우스나 키보드로 스크롤 바를 스크롤시킬 때 발생하는 이벤트이다. 이때 ScrollPos 인수로 스크롤 후의 위치값을 대입해 주어야 한다. ScrollCode는 사용자가 스크롤 바의 어디를 건드렸는지를 나타내는 코드가 온다.

| 속성값 | 정렬 |
|---|---|
| scLineUp | 작은값 증가 |
| scLineDown | 작은값 감소 |
| scPageUp | 페이지 증가 |
| scPageDown | 페이지 감소 |
| scPosition | 사용자가 썸을 직접 변경함 |
| scTrack | 썸을 변경하고 있는 중이다. |
| scTop | Max로 이동 |
| scBottom | Min으로 이동 |
| scEndScroll | 조작 완료 |

OnScroll에서는 이 코드값으로부터 ScrollPos의 값을 적절히 수정해 주면 된다. 이 이벤트는 Win32 API 프로그래밍시에 주로 사용하는 방법이나 델파이에서는 이 이벤트를 굳이 사용해야 할 경우가 별로 없다.

# 13. GroupBox

## ◘ 정의

관련있는 여러 개의 컴포넌트를 그룹으로 만든다. 그룹으로 만들어 사용하는 가장 대표적인 예는 라디오 버튼이며 하나의 그룹에 속한 라디오 버튼끼리는 상호 배타적으로 선택된다. 그룹 박스 컴포넌트를 폼에 배치한 후 하나의 그룹으로 만들고자 하는 컴포넌트를 그룹 박스 내에 배치한다. 이렇게 그룹 박스 안에 컴포넌트를 배치하면 그룹 박스와 내부의 컴포넌트 사이에는

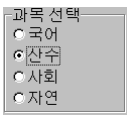
부자 관계가 성립된다.

## ◘ 속성

### Caption:String;
그룹 박스의 이름이며 좌상단에 나타난다. 그룹 박스에 속한 라디오 버튼 등의 의미를 설명할 때 이 속성을 사용한다.

# 14. RadioGroup

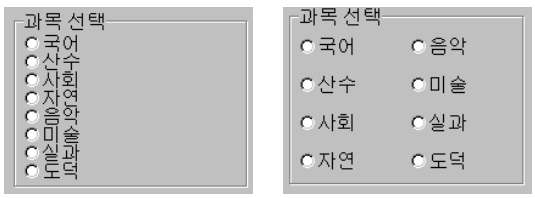
## ◘ 정의
라디오 버튼 전용의 그룹 박스이며 라디오 버튼을 쉽게 만들고 효율적으로 관리할 수 있도록 해준다. Items 속성에 문자열들을 기입해 넣음으로써 라디오 버튼을 쉽게 만들 수 있다.




뿐만 아니라 라디오 버튼 사이의 줄간격을 자동으로 계산해 준다. ItemIndex 속성으로 몇 번째 라디오 버튼이 선택되었는지를 쉽게 알 수 있다.

## ◘ 속성

### Columns:Integer;
디폴트값은 1이며 라디오 버튼이 1열로 생성되지만 이 속성을 변경하여 다중 칼럼의 라디오 버튼을 배치할 수 있다. 라디오 버튼이 많을 경우는 여러 줄로 배치하는 것이 더 보기에 좋다.



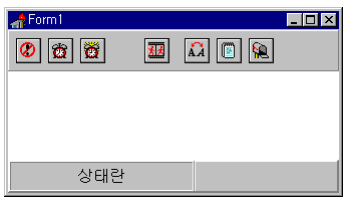

1열 2열

### Caption:String;
라디오 그룹 박스의 이름이며 좌상단에 나타난다. 라디오 그룹에 속한 라디오 버튼 등의 의미를 설명할 때 이 속성을 사용한다.

### ItemIndex:Integer;
실행중에 선택된 라디오 버튼의 번호이다. 라디오 버튼은 상호 배타적으로 선택되므로 하나의 버튼만 선택될 수 있다. 이 속성을 읽으면 선택된 라디오 버튼을 쉽게 알 수 있다. 반면 라디오 그룹을 사용하지 않고 개별적인 라디오 버튼을 사용할 경우 일일이 각 라디오 버튼의 Checked 속성을 검사해 보아야 어떤 라디오 버튼이 선택되었는지를 알 수 있다.

# 15. Panel

## ◘ 정의
패널 컴포넌트는 다른 컴포넌트를 놓을 수 있는 컨테이너 컴포넌트이다. 즉 패널 컴포넌트 위에 다른 여러 개의 컴포넌트를 놓아 일종의 컴포넌트 셋을 구성할 수 있다. 여러 개의 스피드 버튼이나 비트맵 버튼을 패널 위에 놓아 툴바를 만들거나 상태란을 만든다.




패널 위에 컴포넌트를 놓으면 그 컴포넌트는 패널과 함께 움직인다. 패널을 움직이면 패널 위에 놓여진 다른 컴포넌트가 한꺼번에 움직이기 때문에 패널 위에 여러 개의 컴포넌트를 배치한 후 위치를 옮기기가 쉽다. 패널은 패널위에 위치한 모든 컴포넌트의 페어런트이므로 패널을 지우면 패널 위의 컴포넌트도 따라서 지워진다.

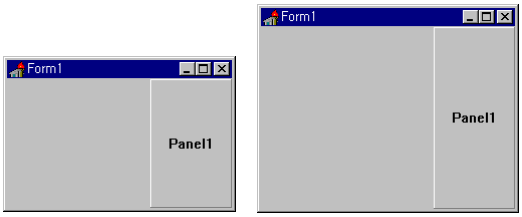
## ◘ 속성

### Align:TAlign;
폼의 상, 하, 좌, 우 특정 위치에 패널을 정렬시키며 일단 정렬된 패널은 폼의 위치나 크기가 변해도 항상 폼의 특정 위치에 위치하게 된다. 가능한 정렬값은 다음과 같다.

| 속성값 | 정렬 |
|---|---|
| alNone | 정렬을 전혀 하지 않으며 디자인시에 지정한 크기와 위치를 사용한다. |

| | |
|---|---|
| alTop | 폼의 위쪽에 밀착 정렬한다. |
| alBottom | 폼의 아래쪽에 밀착 정렬한다. |
| alLeft | 폼의 왼쪽에 밀착 정렬한다. |
| alRight | 폼의 오른쪽에 밀착 정렬한다. |
| alClient | 폼의 전면적으로 모두 덮을만한 크기로 정렬한다. |

다음 예를 보자. 패널의 Align 속성을 alRight로 설정하여 폼의 우측으로 정렬시켰다. 폼의 크기가 변경되어도 패널은 항상 폼의 우측에 위치한다.



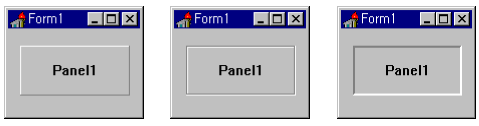

툴바를 만들 경우는 폼의 상단에 밀착시키며 상황선을 만들 경우는 폼의 아래쪽에 밀착시킨다.

## BevelInner,BevelOuter:TPanelBevel;

패널 컴포넌트는 안쪽과 바깥쪽에 각각 하나씩의 베벨을 가진다. 베벨이란 일종의 경계선 표시이며 흰색과 짙은 회색을 사용하여 패널에 입체감을 준다. 두 속성은 다음 세 가지 값 중 하나의 값을 가진다.

| 속성값 | 의미 |
|---|---|
| bvNone | 베벨을 그리지 않는다. |
| bvLowered | 패널이 밑으로 내려간 모양을 만든다. |
| bvRaised | 패널이 위로 올라온 모양을 만든다. |

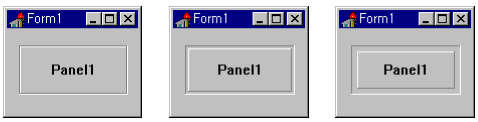
디폴트로 안쪽 베벨은 없으며 바깥쪽 베벨만 bvRaised 속성을 가지므로 패널은 올라온 모양을 가진다. 바깥쪽 베벨을 bvLowered로 만들고 안쪽 베벨을 bvRaised로 만들면 홈이 파진 모양의 베벨이 만들어진다. 또 바깥쪽 베벨은 bvLowered로 설정하면 음각 모양이 된다.




디폴트 패널-양각 홈 모양의 패널 음각 패널

## BorderWidth:Integer;

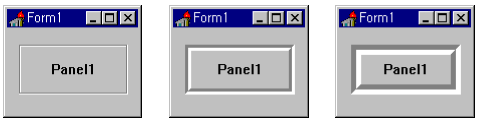
안쪽 베벨과 바깥쪽 베벨의 거리를 지정한다. 디폴트값은 0이며 두 베벨 사이의 간격은 없지만 다음과 같이 간격을 띄울 수도 있다.




0 2 5

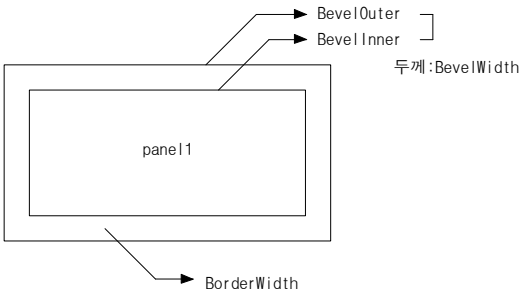
## BevelWidth:Integer;

안쪽 베벨, 바깥쪽 베벨의 선두께를 지정한다. 이 값에 따라 패널이 어느 정도의 양각, 음각을 가질 것인가가 결정된다. 디폴트 값은 1이며 3이상의 값을 주는 것은 시각적으로 좋지못하다. 다음에 이 값을 조정한 여러 가지 패널을 보인다.




1 3 5

네 가지 속성을 조정하여 패널의 모양을 설정한다.




이 네 가지 속성에 따라 패널이 양각이 되기도 하고 음각이 되기도 한다.

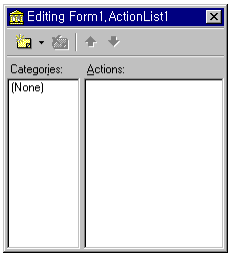
## BorderStyle:TBorderStyle;

패널 전체를 둘러싸는 검정색의 사각형 테두리를 두른다. 이 속성의 디폴트는 bsNone이며 테두리가 없지만 bsSingle로 변경하면 테두리가 생긴다.

# 16.ActionList

## ◘ 정의

툴바, 메뉴, 버튼 등의 컴포넌트에 의해 사용되는 액션의 리스트이다. 액션 리스트내에는 액션(TAction)이라는 컴포넌트가 포함되며 액션 리스트를 더블클릭하면 액션 리스트 편집기가 열린다.




이 편집기를 통해 액션 컴포넌트를 생성하며 개별 액션 컴포넌트의 속성은 오브젝트 인스펙터에서 설정한다. 개별 액션 컴포넌트에 대한 내용은 Action 컴포넌트를 참고하기 바란다.

## ◘ 속성

### 읽 ActionCount:Integer;

액션 리스트에 포함된 액션의 개수를 조사한다.

### Images:TImageList;

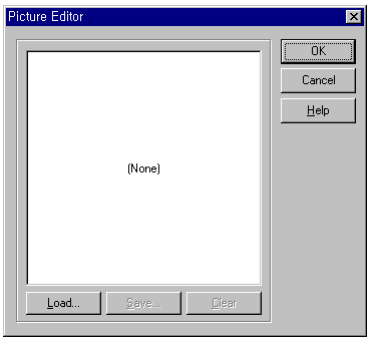
액션이 사용할 이미지의 리스트를 지정한다. 개별 액션이 사용할 이미지의 번호는 액션 컴포넌트의 ImageIndex 속성으로 설정한다.

# 17. BitBtn

## ◘ 정의

비트맵 버튼은 버튼 컴포넌트와 기능 면에서 거의 동일하며 다만 버튼 위에 비트맵을 놓을 수 있다는 점만 다르다. 다음과 같이 버튼 위에 비트맵을 배치함으로써 약간의 장식을 덧붙일 수 있다.


사용 용도는 TButton 컴포넌트와 동일하며 Caption, Default, Cancel 등의 속성이 버튼과 같다. 비트맵 버튼 위에 놓여지는 비트맵은 미리 정의된 몇 가지를 선택해서 사용하거나 직접 그려서 사용한다. 또는 Kind 속성을 사용하여 미리 디자인되어 있는 몇 가지 형태의 비트맵 버튼을 사용할 수 있다. 직접 그린 비트맵을 사용할 경우 Glyph 속성에 사용할 비트맵 이름을 지정해 주어야 하며 오브젝트 인스펙터에서 Glyph 속성을 더블클릭하면 비트맵을 읽을 수 있는 다음과 같은 대화상자를 보여준다.




이 대화상자에서 Load 버튼을 눌러 원하는 비트맵을 선택하면 버튼 위에 비트맵이 출력된다. 물론 비트맵은 이미지 에디터로 미리 만들어 두어야 한다. 아니면 Delphi₩Image 디렉토리에 델파이와 함께 제공되는 100여 개의 비트맵 중 하나를 선택하여 사용할 수 있다. 비트맵 버튼의 여러 가지 상태, 즉 버튼이 눌러졌을 경우, 눌러져 있지 않을 경우, 누를 수 없는 경우 등의 상태에 따라 각각 다른 비트맵을 사용할 수 있으며 이때 몇 개의 비트맵을 사용할 것인가를 NumGlyphs 속성에 정의해 두어야 한다.

## ◘ 속성

### Glyph:TBitmap;

버튼 위에 나타날 비트맵을 설정한다. 버튼의 상태에 따라 최대 네 개의 비트맵을 지정할 수 있다.

| 상태 | 의미 |
|---|---|
| Up | 버튼이 눌러지지 않은 상태. 다른 비트맵이 정의되어 있지 않으면 델파이는 이 비트맵만 사용하여 모든 상태를 나타낸다. |
| Disabled | 버튼을 선택할 수 없는 상태. 보통 희미한 형태의 버튼 모양을 사용한다. |
| Down | 버튼이 눌러진 상태 |
| Stay Down | 버튼이 눌러진 채로 있는 상태를 나타내며 이 상태는 스피드 버튼의 경우에만 가능하다. |

이 네 가지 상태의 비트맵들이 수평적으로 배치되어 하나의 비트맵 안에 들어간다. 예를 들어 델파이가 제공하는 샘플 비트맵 중 ZOOMOUT.BMP는 다음과 같이 되어 있다.

이 비트맵은 Up과 Disable 두 가지 상태의 비트맵을 가지고 있다. 네 가지 비트맵을 모두 사용할 경우는 네 가지 상태의 비트맵을 수평적으로 배치해 주면 된다.

## NumGlyphs:TNumGlyphs;

몇 개의 비트맵을 사용할 것인지를 지정한다. 비트맵 버튼은 버튼의 상태에 따라 최대 4개의 비트맵을 사용할 수 있다. 만약 비트맵이 하나밖에 정의되어 있지 않을 경우는 하나의 비트맵으로 4가지 상태를 모두 표현하게 된다. 각각의 비트맵은 크기가 모두 같아야 한다.

## Kind:TBitBtnKind;

비트맵 버튼의 유형을 지정하며 이 속성을 사용하여 미리 정의된 몇 가지 비트맵 버튼을 사용한다.

| Kind | 버튼 모양 | 설명 |
|---|---|---|
| bkCustom | BitBtn1 | 사용자가 Glyph 속성을 사용하여 비트맵을 선택하는 버튼 |
| bkOK | OK | Default 속성이 True 가 되며 대화상자를 닫는 기능을 가진다. |
| bkCancel | Cancel | Cancel 속성이 True 가 되며 대화상자를 닫는 기능을 가진다. |
| bkYes | Yes | Default 속성이 True 가 되며 대화상자 내에서의 변경 사항이 받아들여지며 대화상자가 닫힌다. |
| bkNO | No | Cancel 속성이 True 가 되며 대화상자 내에서의 변경 사항이 무시되며 대화상자가 닫힌다. |
| bkHelp | Help | 프로그램의 HelpFile 속성에 지정된 Help 화면이 나타난다. 버튼이 지정하는 HelpContext 속성이 사용된다. |
| bkClose | Close | Default 속성이 True 가 되며 폼을 닫는다. |
| bkAbort | Abort | Cancel 속성이 True 이다. |
| bkRetry | Retry | 다시 실행 |
| bkIgnore | Ignore | 무시 |
| bkAll | All | Default 속성이 True 이다. |

Kind 속성을 변경하면 비트맵 버튼의 Caption, Glyph, ModalResult 속성이 적당한 형태로 같이 변경된다.

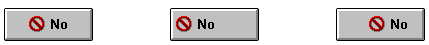
## LayOut:TButtonLayout;

비트맵 버튼 위에 비트맵을 어떻게 정렬할 것인가를 지정하며 다음 4가지 중 하나를 선택한다. 디폴트는 blGlyphLeft이며 비트맵이 버튼의 좌측에 놓인다.

| 속성 | 모양 | 설명 |
|---|---|---|
| blGlyphLeft | All | 이미지가 버튼의 좌측에 나타난다. |
| blGlyphRight | All | 이미지가 버튼의 우측에 나타난다. |
| blGlyphTop | All | 이미지가 버튼의 위에 나타난다. |
| blGlyphBottom | All | 이미지가 버튼의 바닥에 나타난다. |

## Margin:Integer;

비트맵 버튼과 비트맵 사이의 공간 크기를 지정한다. 즉 버튼의 좌측에서 얼만큼 떨어진 거리에 비트맵을 놓을 것인가를 지정한다. 이 값은 Layout 속성에 따라 조금씩 의미가 달라지는데 Layout 이 blGlyphRight라면 버튼의 우측과 비트맵의 거리를 지정하며 blGlyphTop이라면 버튼의 상단과 비트맵의 거리를 지정하게 된다. 디폴트값은 -1이며 이는 비트맵과 문자열이 버튼의 중앙에 오도록 한다는 뜻이다.




Margin=-1 margin=3 Margin=30

## Spacing:Integer;

비트맵 버튼 위의 비트맵과 캡션 간의 간격을 지정한다. 디폴트는 4이며 이는 곧 비트맵과 버튼과의 거리가 4픽셀이라는 뜻이다. 이 값이 -1이면 캡션은 비트맵과 버튼 가장자리의 중앙에 위치한다.

## ModalResult:TModalResult;

Modal형 폼에 있는 버튼의 ModalResult 속성이 0 이외의 값일 경우 이 버튼이 눌러지면 폼이 종료된다. Ok나 Cancel 등과 같이 폼을 닫는 기능을 가진 버튼은 ModalResult 속성을 MrOk나 MrCancel로 설정함으로써 별도의 코드를 작성할 필요없이 버튼이 눌러지면 대화상자나 폼을 닫도록 한다. 버튼의 ModalResult 속성은 폼을 호출한 ShowModal 함수의 리턴값이 된다. 디자인시와 실행시에 모두 사용할 수 있지만 실행시에 값을 변경할 수는 없다. Kind 속성을 변경하면 이 속성도 같이 변경된다.

### Style:TButtonStyle;
비트맵 버튼의 모양을 지정하며 다음 세 가지 속성값을 가진다.

| 속성값 | 의미 |
|---|---|
| bsWin31 | 사용하는 윈도우즈 버전에 상관없이 윈도우즈 3.1 형의 버튼 모양을 가진다. |
| bsNew | 사용하는 윈도우즈 버전에 상관없이 Win95 형의 모양을 가진다. |
| bsAutoDetect | 사용하는 윈도우즈 버전에 따라 적절한 모양의 모양을 가진다. 즉 3.1 에서 실행되면 3.1 의 버튼 모양을 가지며 Win95 에서 실행되면 Win95 의 버튼 모양을 가진다. |

참고로 윈도우즈 3.1에서 사용하는 버튼 모양이 테두리가 조금 더 두껍다. 이 속성은 디폴트로 bsAutoDetect로 되어 있으며 특별한 이유가 아니면 건드리지 않는 것이 좋다.

## ◘ 이벤트

### OnClick
마우스로 비트맵 버튼을 클릭했을 때 발생하며 일반적으로 사용자가 버튼을 선택했을 때의 동작을 이 이벤트의 핸들러에 기록한다. 그러나 Kind 속성으로 버튼의 ModalResult 속성을 미리 정의해 두었다면 이 이벤트는 사용하지 않아도 된다.

# 18. SpeedButton

## ◘ 정의
비트맵 버튼과 유사하나 버튼 표면에 문자를 가지지 않는 경우가 일반적이며 그래픽만을 가진다. 물론 Cpation 속성을 정의해 주면 문자를 가질 수도 있다.


보통 패널과 함께 사용되며 툴바를 구성한다. 스피드 버튼에 사용될 비트맵은 Glyph 속성으로 지정하며 오브젝트 인스펙터에서 Glyph 속성을 더블클릭하여 비트맵을 읽을 수 있는 대화상자를 불러낸다. 스피드 버튼의 상태에 따라 각기 다른 여러 개의 비트맵을 사용할 수 있으며 사용하는 비트맵의 개수는 NumGlyphs 속성으로 지정한다. 스피드 버튼의 주요한 특징으로는 여러 개의 스피드 버튼이 모여 그룹을 구성할 수 있다는 점이다. 스피드 버튼으로 그룹을 구성하기 위해 GroupIndex 속성을 사용한다. Layout, Margin, Spacing 속성의 의미는 비트맵 버튼과 동일하므로 참조하기 바란다.

## ◘ 속성

### Glyph:TBitmap;
버튼 위에 나타날 비트맵을 설정한다. 버튼의 상태에 따라 최대 네 개의 비트맵을 지정할 수 있다.

| 속성 | 의미 |
|---|---|
| Up | 버튼이 눌러지지 않은 상태. 다른 비트맵이 정의되어 있지 않으면 델파이는 이 비트맵만 사용하여 모든 상태를 나타낸다. |
| Disabled | 버튼을 선택할 수 없는 상태. 보통 희미한 형태의 버튼 모양을 사용한다. |
| Down | 버튼이 눌러진 상태 |
| Stay Down | 버튼이 눌러진 채로 있는 상태를 나타내며 이 상태는 스피드 버튼의 경우에만 가능하다. |

이 네 가지 상태의 비트맵들이 수평적으로 배치되어 하나의 비트맵 안에 들어간다.

### NumGlyphs:TNumGlyphs;
몇 개의 비트맵을 사용할 것인가를 지정한다. 스피드 버튼은 버튼의 상태에 따라 최대 4개의 비트맵을 사용하고 있다. 각각의 비트맵은 크기가 모두 같아야 한다.

### GroupIndex:Integer;
그룹을 구성하여 사용할 스피드 버튼을 지정한다. 이 속성이 같은 스피드 버튼은 모두 하나의 그룹이 되며 그룹 내의 한 버튼이 눌러지면 눌러진 상태를 계속 유지하며(StayDown) 이전에 눌러졌던 버튼은 선택 취소된다. 그룹을 이루는 스피드 버튼은 라디오 버튼과 동일하게 작동한다. GroupIndex 속성의 디폴트값은 0이며 이 속성이 0이면 그룹을 구성하지 않고 단독으로 사용된다.

### Down:Boolean;
스피드 버튼은 최초 실행시 선택되지 않은(눌러지지 않은)상태이다. Down 속성이 True이면 눌러진 상태에서 프로그램이 시작된다.

### AllowAllUp:Boolean;
스피드 버튼이 그룹을 구성하여 사용될 경우 그룹 내에 반드시 한 버튼만이 선택될 수 있으며 또한 반드시 한 버튼은 선택되어 있어야 한다. 이 속성이 True이면 그룹 내의 모든 버튼이 선택

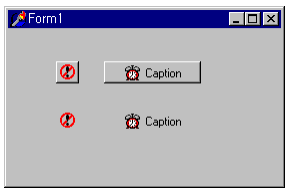
되지 않은(눌러지지 않은) 상태가 될 수 있다. 그룹내 한 버튼의 이 속성을 변경하면 다른 버튼의 이 속성도 모두 변경된다.

### 읽 MouseInControl:Boolean;

마우스 버튼이 스피트 버튼 컨트롤의 내부에 있는지를 조사한다.

### Flat:Boolean;

삼차원의 경계선을 가지지 않도록 한다. 이 속성이 True 이면 인터넷 탐색기 스타일의 버튼이 만들어지며 마우스 커서가 버튼 위로 올라가면 경계선이 보인다. 다음 아래쪽의 버튼이 이 속성을 True 로 설정하여 만든 것이다.




# 19. MaskEdit

## ◘ 정의

표준 에디트 박스와 동일하며 EditMask 속성을 사용하여 특정 문자만 입력받도록 한다. 만일 사용자가 유효하지 않은 문자를 입력하면 입력되지 않는다. 예를 들어 전화 번호의 경우 000-0000이라는 마스크를 지정하여 국번 세 자리와 번호 네 자리가 입력되도록 하며 우편 번호는 000-000으로 마스크를 지정하여 앞, 뒤 세 자리씩 입력되도록 한다. 입력된 문자는 Text 속성으로 읽어내며 Modified, MaxLength, ReadOnly, AutoSelect 등 대부분의 속성이 표준 에디트 박스와 동일하다. EditMask 속성으로 사용자가 입력할 수 있는 문자를 제한하며 여러 가지 마스크 문자가 사용된다.

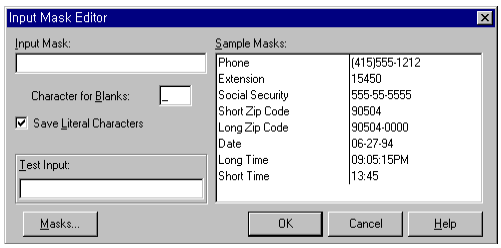
## ◘ 속성

### EditMask:string;

실행중에 마스크 에디트에 입력될 수 있는 문자를 제한하며 다음과 같은 여러 가지 마스크 문자를 사용하여 마스크를 설정한다.

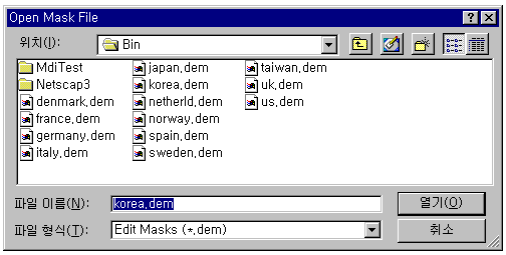
| 문자 | 의미 |
|---|---|
| ! | 이 문자가 마스크에 있으면 선행 공백을 모두 삭제하며 없으면 후행 공백을 모두 삭제한다. |
| > | 입력되는 모든 문자를 대문자로 변경한다. |
| < | 입력되는 모든 문자를 소문자로 변경한다. |
| ◇ | 이 두 문자가 한꺼번에 연이어 오면 대소문자 검사를 하지 않고 입력된 문자를 그대로 출력한다. |
| ₩ | 이 문자 뒤의 한 문자는 마스크 문자로 인정되지 않으며 문자 그대로 출력된다. 즉 C 는 마스크 문자이지만 ₩C 는 일반 문자 C 이다. |
| L | A-Z,a-z 까지 알파벳 문자만 올 수 있다. 그 외의 숫자나 기호는 입력할 수 없다. |
| l | 위와 같으나 생략 가능 |
| A | 알파벳 문자와 숫자만 올 수 있다. A-Z,a-z 와 0-9 까지의 숫자를 입력할 수 있지만 $,%,( 등의 기호는 입력받을 수 없다. |
| a | 위와 같으나 생략 가능 |
| C | 문자가 올 수 있다. 알파벳, 숫자, 기호 등 어떤 문자라도 입력받을 수 있다. |
| c | 위와 같으나 생략 가능 |
| 0 | 숫자만 올 수 있다.0-9 까지의 10 진 숫자만 입력받는다. |
| 9 | 위와 같으나 생략 가능 |
| # | 숫자 또는 부호만 올 수 있다.0-9 까지의 숫자, +,- 등의 기호를 입력할 수 있다. |
| : | 시간 구분에 사용한다. |
| / | 날짜 구분에 사용한다. |
| ; | 마스크 구분에 사용한다. |
| - | 이 문자 자리는 공백의 자리이며 커서가 이 위치를 건너뜀으로써 자동으로 한칸의 공백을 만든다. |

l, a, c, 9 등의 마스크 문자는 L, A, C, 0와 동일한 문자를 입력받되 반드시 해당 문자를 입력해야 할 필요는 없으며 입력하지 않고 공백으로 남겨 두어도 된다. 예를 들어 전화 번호의 경우 (999)000-0000으로 마스크를 설정하면 전화 번호 7자리는 반드시 입력해야 하지만 앞의 DDD 번호 세 자리는 입력하지 않을 수도 있다. 오브젝트 인스펙터에서 EditMask 속성을 더블클릭하면 마스크 편집 대화상자가 열린다.




이 대화상자를 사용하면 마스크 입력 및 입력된 마스크를 테스트해 볼 수 있으며 우측의 Sample Masks에서 미리 작성된 몇 가지 마스크를 간단히 입력할 수도 있다. Save Literal

Characters 체크 박스는 Text 속성에 입력된 내용을 대입할 때 일반 문자를 출력할 것인지 아닌지를 결정하며 이 체크 박스를 변경하면 마스크의 두 번째 필드가 0 또는 1로 자동 변경된다. Character For Blanks는 공백으로 사용할 공백 문자를 설정하며 공백 문자는 사용자가 입력해야 할 문자의 위치를 나타낸다. 디폴트는 '_'로 설정되어 있지만 사용자가 입력해야 할 자리수를 정확히 알고 있다면 스페이스로 대체할 수도 있다. Masks 버튼은 각 국가별로 사용되는 샘플 마스크를 우측의 리스트 박스로 읽어들인다. 여러 국가의 샘플 마스크 중 하나를 선택할 수 있다.




우리나라 대한민국도 있다.

### Text:TCaption;

마스크 에디트에 입력된 문자열이며 Save Literal Character 옵션 설정 상태에 따라 일반 문자가 포함되기도 하고 제외되기도 한다. 예를 들어 마스크가 ₩(999₩)000-0000로 설정되어 있고 사용자가 (123)456-7890을 입력했을 경우 Save Literal Charcter 옵션을 선택하지 않았다면 Text 속성은 ( , ), - 등의 일반 문자는 제외하고 사용자가 직접 입력한 12345678900이 된다.

### 실 EditText:string;

마스크 에디트에 사용자가 입력한 문자열이며 일반 문자를 포함한다. 즉 이 속성값이 실행중에 에디트 박스에 나타나는 문자열이다.

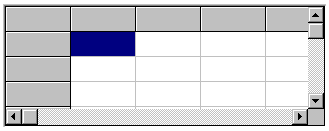
### 실 읽 IsMasked: Boolean;

EditMask 속성이 정의되어 있는가를 조사한다. 이 속성이 False이면 마스크가 정의되어 있지 않은 것이다.

# 20. StringGrid

## ◘ 정의

문자열을 담는 격자를 가지는 컴포넌트이다. 가로 세로로 배열된 격자에는 문자열들이 저장된다. 폼에 배치하면 다음과 같이 5*5 행렬 모양의 빈 그리드가 생성된다.



각 격자(=셀)에 기억되어 있는 문자열을 읽거나 대입할 때는 Cells 속성을 사용한다. 단 디자인중에는 이 속성을 사용할 수 없으므로 실행중에 코드로 문자열을 대입해야 한다. 그리드는 표 형식의 컴포넌트이며 각 셀은 가로 좌표와 세로 좌표로서 지정된다. 예를 들어 [3,3] 위치의 셀을 읽으려면 StringGrid1.Cells[3,3]의 값을 읽으며 값을 대입하려면 StringGrid1.Cells[3,3]:='56'; 과 같이 코딩한다. 셀의 좌표는 좌상단 셀이 [0,0]이며 첫 번째 첨자가 가로줄의 열(Col) 좌표이며 두 번째 첨자가 세로축의 행(Row) 좌표이다.

| | | | | |
|---|---|---|---|---|
| [0,0] | [1,0] | [2,0] | | |
| [0,1] | [1,1] | [2,1] | | |
| [0,2] | [1,2] | [2,2] | | |
| [0,3] | [1,3] | [2,3] | | |
| | | | | |
| | | | | |

문자열 그리드의 가로 세로 크기는 ColCount, RowCount 속성에 의해 설정되며 각 셀의 크기는 ColWidth, RowHeight 속성에 의해 설정된다.

## ◘ 속성

### 실 Cells[i,j]:string;

실행중에만 쓸 수 있는 속성이다. 즉 디자인시에는 문자열 그리드에 문자열을 입력할 수 있는 방법이 없다. Cells[Col,Row]의 형식으로 참조되는 2차원 문자열 배열 형태이며 행, 열의 첨자를 지정함으로써 해당 셀의 값을 읽거나 대입한다. 좌상단의 첫 번째 셀이 Cells[0,0]이다.

### ColCount, RowCount:LongInt;

문자열 그리드에 포함된 셀의 개수를 결정한다. 가로 쪽 셀 수는 ColCount 속성에 의해 지정되며 세로쪽 셀 수는 RowCount 속성에 의해 지정된다. 따라서 문자열 그리드 내의 총 셀수는 ColCount*RowCount개가 되며 디폴트로 이 값은 5*5=25이다.

## 실 Col, Row:LongInt;

문자열 그리드에서 현재 포커스를 가지고 있는 셀의 좌표이다. Col이 가로축 좌표, Row가 세로축 좌표를 나타낸다. 실행중에만 사용할 수 있는 속성이지만 읽기 전용은 아니므로 이 값을 변경하여 포커스를 강제로 옮길 수도 있다.

## 실 Cols, Rows:TStrings;

문자열 그리드의 특정한 행(또는 특정한 열)의 문자열을 가지는 1차원 문자열 배열이다. 예를 들어 Cols[3]은 3열의 문자열 모두를 가지는 문자열 배열이다. Cols, Rows는 문자열 리스트(TString)형의 배열이므로 문자열 리스트에서 사용할 수 있는 모든 메소드를 사용할 수 있다. 예를 들어 StringGrid1.Cols[2].Add('korea');는 2열 문자열 리스트에 'korea'라는 문자열을 추가한다.

## DefaultColWidth, DefaultRowHeight:Integer;

그리드에 생성되는 셀의 디폴트 폭과 높이를 지정한다. 이 속성을 변경하면 그리드 내의 모든 셀들의 크기가 변한다. 개별적인 셀의 폭과 높이를 변경하려면 ColWidths, RowHeights 속성을 사용한다.

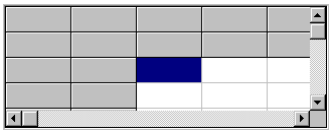
## 실 ColWidths[i]:Integer;

## 실 RowHeights[i]:Integer;:

개별적인 셀의 폭과 높이를 설정한다. 문자열 그리드가 처음 만들어질 때 이 속성값들은 DefaultColWidth, DefaultRowHeight 속성값에 따라 디폴트 설정된다. 개별 셀의 폭, 높이를 바꾸려면 프로그램 실행중에 이 속성값을 변경한다. 이 속성은 각 열과 행의 폭, 높이 값을 가지는 1차원 정수 배열이므로 설정하고자 하는 행과 열의 번호를 첨자로서 지정한다. 예를 들어 ColWidths[3]:=100;은 3열의 폭을 100픽셀로 설정한다. 이 속성들은 디자인시에는 사용할 수 없는 속성이지만 디자인시에도 개별 셀의 크기를 변경할 수 있다. 디자인시에는 마우스로 셀의 경계 부분을 드래그하여 크기를 변경한다.

## FixedCols, FixedRows:Integer;

그리드의 좌측과 상단에 스크롤되지 않고 고정되어 있는 행과 열의 수를 지정한다. 고정된 행과 열은 각 행, 열의 제목을 나타내는 목적으로 사용되며 일반 셀과는 다른 색상을 사용한다. 고정된 행과 열의 수는 총 열의 수, 총 행의 수보다 반드시 1 작아야 한다. 디폴트값은 모두 1이므로 고정된 행과 열이 각각 하나씩 있다.

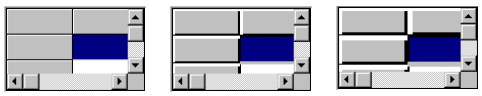


## FixedColor:TColor;

고정된 행과 열의 색상을 설정한다.

## GridLineWidth:Integer;

각 셀의 행과 열을 구분하는 선의 굵기를 지정한다. 디폴트값은 1이므로 얇은 선 하나만 있지만 이 값을 증가시키면 경계선의 색상이 두터워진다. 이 값을 0으로 설정하면 경계선이 없어진다.



1 3 5

## 실 읽 VisibleColCount,VisibleRowCount:Integer;

스크롤 영역으로 숨겨지지 않고 현재 화면에 나타나 있는 셀의 개수를 조사한다. 일부만 보여지는 셀은 개수에서 제외된다.

## Options:TGridOptions

그리드의 외적 모양을 지정하는 여러 가지 세부 속성을 가지고 있다. 각 세부 속성은 모두 진위형이다. 다음과 같은 세부 속성들이 있다.

| 옵션 | 의미 |
|---|---|
| goFixedHorzLine | 비스크롤 영역에 수평선을 그린다. |
| goFixedVertLine | 비스크롤 영역에 수직선을 그린다. |
| goHorzLine | 수평선을 그린다. |
| goVertLine | 수직선을 그린다. |
| goRangeSelect | 실행 중에 사용자가 일정 범위의 셀을 선택할 수 있도록 한다. goEditing이 참으로 설정된 경우는 범위 선택을 할 수 없다. |
| goDrawFocusSelected | 포커스를 가진 셀이 선택 블럭의 다른 셀들과 같은 색상을 가지도록 한다. False일 경우는 포커스를 가진 셀은 선택되지 않은 셀과 같은 색상을 가진다. |
| goRowSizing | 개별적인 행의 크기를 변경할 수 있다. |

| | |
|---|---|
| goColSizing | 개별적인 열의 크기를 변경할 수 있다. |
| goRowMoving | 마우스를 사용하여 한 행을 다른 위치로 이동시킬 수 있다. |
| goColMoving | 마우스를 사용하여 한 열을 다른 위치로 이동시킬 수 있다. |
| goEditing | 실행중에 사용자가 그리드의 문자열을 직접 편집할 수 있다. 편집이 가능한 상태에서 범위 선택은 할 수 없다. |
| goAlwaysShowEditor | goEditing 속성이 참일 경우만 사용 가능한 옵션이다. 이 옵션이 True이면 F2나 Enter를 누르지 않고도 그리드의 문자열을 곧바로 편집 가능하다. |
| goTabs | Tab키를 사용하여 그리드의 열을 이동한다. |
| goRowSelect | 셀하나만 선택할 수 없으며 한 행 전체를 선택하도록 한다. |
| goThumbTracking | 스크롤 바의 썸을 드래그할 경우 그리드의 문자열도 같이 드래그된다. 이 옵션이 False이면 썸을 드래그하여 놓기 전에는 그리드의 문자열이 스크롤 되지 않는다. |

### 실 Objects[i,j]:TObject;

실행중에만 사용할 수 있는 속성이다. 각 셀에 대응되는 오브젝트를 기억한다. 셀에 오브젝트를 연결했을 경우 오브젝트와 셀은 별개로 존재한다. 문자열 그리드가 파괴되어도 오브젝트들은 파괴되지 않으므로 개별적으로 파괴해 주어야 한다. TObject형의 2차원 배열이므로 셀의 번호를 첨자로 사용한다. String1.Objets[1,1]:=MyIcon; 등과 같이 셀과 오브젝트를 연결한다.

### 실 LeftCol, TopRow:LongInt;

실행중에만 사용할 수 있는 속성이다. 어떤 행(열)이 그리드의 최좌측(최상단)에 나타날 것인가를 설정한다.

## 21. DrawGrid

### ◘ 정의

가로, 세로 표 형식으로 데이터를 출력해 주는 컴포넌트이다. DefaultDrawing 속성이 False이면 OnDrawCell 이벤트에서 셀에 직접 데이터를 출력해야 하며 True일 경우 내정된 값으로 자동적으로 셀을 그린다. 그림을 그릴 영역은 CellRect 메소드로 구하며 MouseToCell 메소드로 마우스 커서가 있는 셀의 좌표를 구할 수 있다.

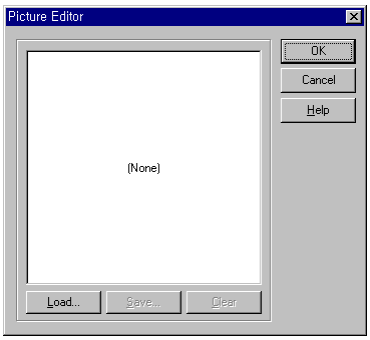
### ◘ 속성

DrawGrid의 속성 대부분은 StringGrid와 완전히 동일하므로 참고하기 바란다.

## 22. Image

### ◘ 정의

이미지 컴포넌트는 폼 상에 그림을 놓고자 할 때 사용된다. 어떤 그림을 놓을 것인가는 이미지 컴포넌트의 Picture 속성에 설정한다. 오브젝트 인스펙터의 Picture 속성을 더블클릭하거나 폼의 이미지 컴포넌트를 더블클릭하면 그림을 읽을 수 있는 다음과 같은 대화상자를 보여준다.




이 대화상자에서 *.BMP, *.ICO, *.WMF, *.JPG 파일을 선택해 읽어올 수 있다. JPG 파일을 읽어오는 코드는 JPEG 유닛에 선언되어 있는데 이미지에 JPG 파일을 선택하면 이 유닛이 자동으로 uses절에 삽입된다. 그러나 만약 실행중에 JPG 파일을 읽어오고자 한다면 uses JPEG문은 직접 넣어 주어야 한다. 이미지는 Canvas를 가지므로 그림 파일을 읽어와 출력하는 것 외에도 LineTo, Ellipse, Rectangle 등 Canvas의 그리기 메소드를 사용하여 그림을 직접 그릴 수도 있다.

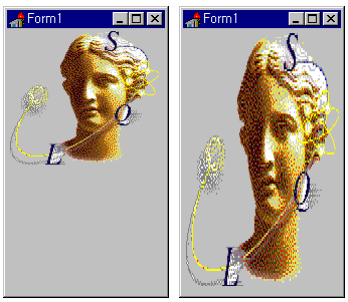
### ◘ 속성

#### Autosize:Boolean;

이 속성이 True이면 이미지 컴포넌트의 크기가 그림의 크기에 맞게 조정된다. 디폴트값은 False이며 그림 크기에 상관없이 이미지 컴포넌트가 일정한 크기를 가진다. 만약 이미지 컴포넌트가 그림보다 작다면 그림의 일부는 잘려서 보이지 않게 된다.

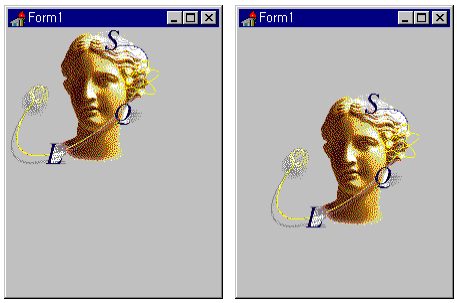
### Stretch:Boolean;

*.BMP, *.WMF 파일의 경우 그림의 크기를 이미지 컴포넌트의 크기에 맞추어 재조정한다. 아이콘 파일은 이 속성의 영향을 받지 않는다. 왼쪽의 그림은 Stretch를 하지 않은 것이고 오른쪽의 그림은 Stretch를 한 것이다.




### Center:Boolean;

그림을 이미지 컴포넌트의 중앙에 놓도록 한다. 디폴트값은 False이며 이 경우 그림은 이미지 컴포넌트의 좌상단에 놓인다. 왼쪽 그림은 Center 속성이 False이며 오른쪽 그림은 Center 속성이 True이다.




### IncrementalDisplay:Boolean;

시간이 오래 걸리는 작업, 예를 들어 크기가 큰 압축 이미지를 풀고 있는 작업 중에 대충의 이미지를 그릴 수 있는가를 나타낸다. 이 속성이 True 이면 그래픽 파일을 완전히 읽지 않아도 출력이 가능하며 파일을 다 읽기 전에 출력을 금지하려면 이 속성을 False 로 설정한다.

## ◘ 이벤트

### OnProgress

그림을 읽어오는 등 시간이 오래 걸리는 작업중에 주기적으로 발생하는 이벤트이다. 사용자에게 현재 상황을 보여주고자 할 때 이 이벤트에서 진행 상태를 보여주면 된다. 이 이벤트 타입은 다음과 같이 정의되어 있다.

```
TProgressStage = (psStarting, psRunning, psEnding);
TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage; PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg: string) of object;

property OnProgress: TProgressEvent
```

State 인수는 작업의 시작, 진행, 종료를 알려주는데 이 값이 psStarting이면 작업의 시작, psRunning이면 진행중, psEnding이면 작업의 종료라는 뜻이다. 만약 프로그래스 바로 진행 상태를 보여주고자 한다면 작업 시작시에 프로그래스 바를 생성하고 진행중에 프로그래스 바의 포지션을 갱신하며 작업 종료 후에 파괴하면 된다. percentDone 인수는 작업의 진행률을 백분율로 알려주되 단 이 값은 대충의 값일 뿐 정확한 값은 아니다. RedrawNow 인수는 현재 그림을 그려도 되는지를 나타낸다. R 인수는 그림이 변경되어 다시 그려져야 할 부분에 대한 사각 영역을 전달해 준다.
Msg 문자열은 현재 작업 상황에 대한 간단한 설명을 담는데 예를 들어 Loading, Storing, Reducing colors 등의 문자열이 전달되며 이 문자열은 상태란이나 기타 문자열을 출력할 수 있는 곳에 바로 출력할 수 있다.

# 23. Shape

## ◘ 정의

폼에 기하학적인 도형을 위치시키는 비윈도우(nonwindowed) 컴포넌트이다. 따라서 윈도우 핸들을 가지지 않으며 실행중에 포커스를 가질 수 없으며 다만 장식에 사용될 뿐이다. Shape 속성으로 원, 사각형 등의 도형을 선택하며 Brush, Pen 등의 속성으로 도형의 모양을 결정한다.

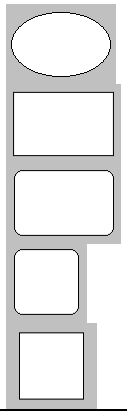
## ◘ 속성

### Shpae:TShapeTyep;

어떤 도형을 배치할 것인가를 선택한다. 다음 6가지가 선택 가능하다.

| 속성 | 도형 | 모양 |
|---|---|---|
| stCircle | 원 |  |

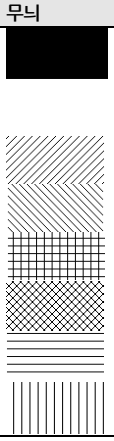
| | |
|---|---|
| stEllipse | 타원 |
| seRectangle | 직사각형 |
| stRoundRect | 둥근 직사각형 |
| stRoundSquare | 둥근 정사각형 |
| stSquare | 정사각형 |



### Brush:TBrush;

도형 내부를 채색할 브러시를 지정하며 Color 세부 속성으로 채색 색상을 선택하고 Style 세부 속성으로 채색 무늬를 선택한다. Style은 다음과 같은 값들이 가능하다.

| 속성 | 무늬 |
|---|---|
| bsSolid | |
| bsClear | |
| bsBDiagonal | |
| bsFDiagonal | |
| bsCross | |
| bsDiagCross | |
| bsHorizontal | |
| bsVertical | |



### Pen:TPen;

도형의 외곽선을 그릴 펜을 지정하며 Color, Mode, Style, Width 세부 속성을 사용한다. Color가 색상 Width가 폭이며 Mode가 ROP 연산법, Style이 선의 모양을 지정한다.

| 속성 | 설명 | 선 |
|---|---|---|
| psSolid | 실선 | |
| psDash | 굵은 점선 | |
| psDot | 가는 점선 | |
| psDashDot | 일점 쇄선 | |
| psDashDotDot | 이점 쇄선 | |
| psClear | 투명한 선 | 안보임 |
| psInsideFrame | 외곽에 밀착한 실선 | |

## 24. Bevel

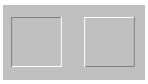
## ◘ 정의

폼 안에 사각 박스를 위치시킨다. 어디까지나 장식에 사용되는 단순한 도형에 불과하며 패널과 같이 다른 컴포넌트를 담을 수 있는 컨테이너는 아니다. Shape, Style 속성으로 베벨의 모양을 결정한다.

## ◘ 속성

### Style:TBevelStyle;

베벨이 양각 모양을 가질 것인가 음각 모양을 가질 것인가를 지정한다. 두 가지 형태의 베벨 모양은 다음과 같다.



### Shape:TBevelShape;

베벨의 모양을 결정하며 다음중 하나의 값을 가진다.

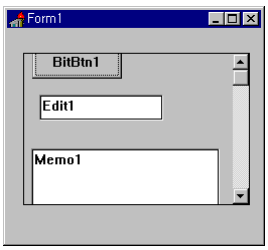
| 속성 | 의미 |
|---|---|
| bsBox | 음각 또는 양각 모양의 사각형 |
| bsFrame | 음각 또는 양각의 홈을 가지는 틀 |
| bsTopLine | 상단에 선 하나만을 가진다. |
| bsBottomLine | 하단에 선 하나만을 가진다. |
| bsLeftLine | 좌측에 선 하나만을 가진다. |
| bsRightLine | 우측에 선 하나만을 가진다. |

각 모양은 다음과 같다.

# 25. ScrollBox

## ◘ 정의

전체 폼보다 작은 스크롤 영역을 만든다. 폼은 폼 내의 컴포넌트가 다 표시되지 못할 때 스크롤 바를 사용하여 작업 영역을 이동할 수 있도록 해 주지만 이때 툴바나 상태바로 사용되는 패널까지도 스크롤되어 버린다. 이런 문제를 해결하기 위해 스크롤 박스를 사용하며 패널이 차지한 영역의 나머지 영역을 스크롤 바가 차지하도록 하여 패널은 스크롤 영역에서 제외시킨다.
또는 폼의 일정 영역에 스크롤되는 박스를 만들고자 한다면 이 컴포넌트를 사용한다. 스크롤 박스는 컨테이너 컴포넌트이며 이 안에 다른 컴포넌트를 배치할 수 있다. 다음 그림은 폼 중앙에 스크롤 박스를 배치하고 스크롤 박스 안에 버튼과 에디트, 메모를 배치한 모양이다.




스크롤 박스 외부는 스크롤에서 제외시킬 수 있다. 실행중에는 물론이며 디자인중에 스크롤 박스의 스크롤 바를 사용하여 스크롤시킬 수 있다.

## ◘ 속성

### HorzScrollBar,VertScrollBar:TControlScrollBar;

가장 중요한 속성이며 수평, 수직 스크롤 바의 모양, 동작을 지정하며 다음과 같은 세부 속성을 가진다.

| 세부 속성 | 의미 |
|---|---|
| Visible | 스크롤 바의 표시 여부를 지정하며 반드시 True이어야 한다. |
| Range | 스크롤 바의 범위를 지정하며 이 값은 반드시 작업 영역의 폭이나 높이보다 커야 한다. |
| Margin | 스크롤 바가 나타날 최소값, Margin+Range가 작업 영역의 폭이나 높이보다 더 클 때 스크롤 바가 나타난다. |
| Position | 스크롤 바의 썸 위치 |
| Increment | 양끝의 화살표를 클릭할 때의 증감분 |

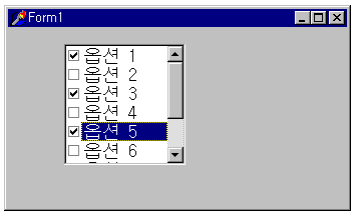
### BorderStyle:TBorderStyle;

스크롤 박스 주위의 경계선 속성을 지정하며 테두리가 없는 bsNone과 검정색의 테두리가 있는 bsSingle 두 가지 속성 값이 있다.

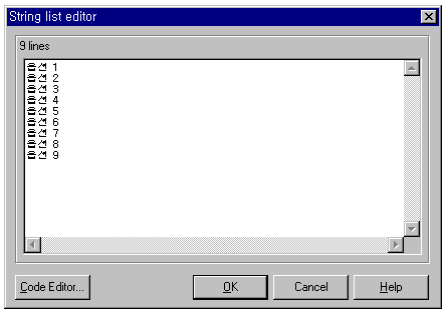
# 26.CheckListBox

## ◘ 정의

리스트 박스내에 여러 개의 체크 박스를 가진다. 선택해야 할 진위형 옵션이 다수 개일 경우 이 컴포넌트를 사용한다.




일종의 오너 드로우 리스트 박스이며 다만 리스트 박스내의 항목이 체크 박스 형태로 나타난다는 것만 다르다. 리스트 박스내의 체크 항목은 Items 속성에 문자열 형태로 입력해 준다.




옵션의 전체 개수가 리스트 박스의 높이보다 많을 경우는 오른쪽에 스크롤 바가 나타난다. 개별 항목의 선택 상태는 리스트 박스내의 체크 박스를 클릭하면 된다. Items, Columns, ItemIndex, Sorted 등의 속성은 모두 리스트 박스의 속성과 완전히 동일하다.

## ◘ 속성

### AllowGrayed:Boolean

체크 박스가 grayed 상태를 가질 수 있는가 아닌가를 설정한다.

디폴트값이 False 이므로 리스트 박스내의 체크는 선택/비선택의 두 가지 상태밖에 가지지 못하지만 이 속성을 True 로 바꾸어주면 선택도 비선택도 아닌 Grayed 라는 상태를 가지게 된다. 이 속성을 변경하면 체크 리스트내의 모든 체크상태가 한꺼번에 변경되며 개별 체크의 AllowGrayed 는 설정할 수 없다.

### 실 Checked:Boolean;

개별 체크 버튼이 선택되어 있는지 아닌지를 나타내는 진위형 배열 속성이다. Checked[2] 값을 읽음으로써 세 번째 체크 버튼의 체크 상태를 조사할 수 있다.

### 실 State:TCheckBoxState;

개별 체크 버튼의 현재 상태를 나타내는 배열 속성이며 cbChecked, cbUnChecked, cbGrayed 중 한 값을 가진다. State[2] 값을 읽음으로써 세 번째 체크 버튼의 선택 상태를 조사할 수 있다.

## ◘ 이벤트

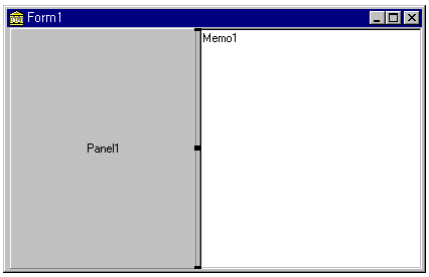
### OnClickCheck

사용자가 체크 리스트 박스내의 체크 박스를 선택하거나 비선택할 때마다 이 이벤트가 발생한다. 체크 박스가 변경될 때마다 해야 할 작업이 있다면 이 이벤트의 핸들러에 코드를 작성한다.

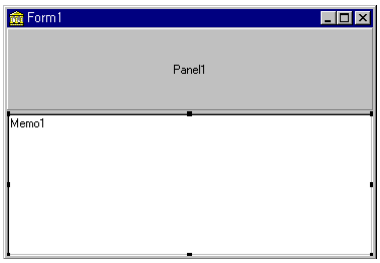
# 27.Splitter

## ◘ 정의

폼의 작업 영역을 크기 조정이 가능한 페인으로 분할하는 컴포넌트이다. 두 개의 정렬된 컴포넌트 사이에 배치되어 두 컴포넌트의 크기를 실행중에 변경할 수 있도록 한다. 이때 스플리터는 반드시 처음 배치된 컴포넌트와 같은 속성은 Align 속성으로 설정되어야 하며 두 번째 배치되는 속성은 폼의 나머지를 채울 수 있도록 alClient 정렬되어야 한다. 다음은 패널, 메뉴를 배치하고 가운데 스플리터를 배치한 것이다.




패널의 Align 속성을 alLeft 로 설정하였으며 스플리터의 Align 속성도 alLeft 로 설정하여 스플리터가 패널의 좌측변에 밀착되도록 하였다. 그리고 메모는 폼의 나머지 영역을 차지할 수 있도록 alClient 정렬되었다. 만약 다음과 같이 패널과 메모를 수직으로 분할하려면 스플리터를 alTop 으로 배치해 주어야 한다.




스플리터의 Cursor 속성은 수평 분할일 경우 crHSplit 가 되며 수직 분할일 경우 crVSplit 가 된다.

## ◘ 속성

### Beveled:Boolean;

이 값이 Ture 이며 스플리터가 3 차원 모양이 된다.

### MinSize:NaturalNumber;

이웃 컴포넌트의 최소 크기를 지정하는데 실행중 크기 변경에 의해 컴포넌트가 완전히 가려지는 것을 방지하기 위해 이 속성을 사용한다. 디폴트값은 30 으로 되어 있다.

### ResizeStyle:TResizeStyle;

스플리터가 드래그되고 있는 동안의 효과를 지정한다. 효과를 지정할 뿐이며 마우스 드래그에 의해 크기가 조절되는 동작은 동일하다.

| 값 | 설명 |
|---|---|
| rsNone | 마우스 버튼을 놓기 전에는 어떠한 동작도 하지 않으며 버튼을 놓는 순간 크기가 변경된다. |
| rdLine | 마우스를 드래그하는 동안 굵은 선을 보여준다. 마우스 버튼을 놓으면 선이 있던 자리가 새로운 경계선이 된다. |
| rsUpdate | 마우스를 드래그하는 동안 실제로 컴포넌트의 크기가 변경된다. |
| rcPatter | 마우스를 드래그하는 동안 회색의 음영을 보여주며 마우스 버튼을 놓으면 음영이 있던 자리가 새로운 경계선이 된다. |

## ◘ 이벤트

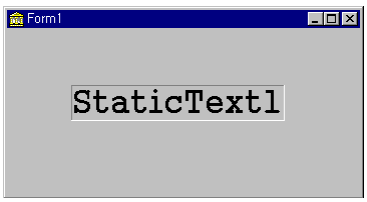
### OnCanResize
사용자가 마우스로 스플리터를 옮기고자 할 때 발생한다. 옮긴 후의 새로운 크기인 NewSize 와 옮기는 것을 허락할 것인지를 질문하는 Accept 참조 인수가 전달되는데 이 이벤트의 핸들러에서 NewSize 값과 프로그램의 상황을 참조하여 옮기는 것을 거부할 수도 있다. Accept 인수에 False 를 대입하면 어떠한 크기 변경도 금지된다.

### OnMoved
스플리터로 크기 변경을 하고 난 후에 발생하는 이벤트이다. 크기가 바뀐 후에 해야 할 작업이 있다면 이 이벤트 핸들러에 코드를 작성하면 된다.

# 28.Static Text

## ◘ 정의
표준 레이블과 용도상으로 완전히 동일한 컴포넌트이다. 즉 폼의 특정 위치에 배치되어 문자열을 보여주는 기능을 한다. 단 이 컴포넌트는 윈도우 핸들을 가지는 윈도우 컨트롤이라는 점이 다를 뿐이다. ActiveX 속성 페이지에서 컴포넌트의 단축키를 가지는 레이블은 반드시 윈도우 핸들을 가져야 하는데 이런 용도로만 사용되며 일반적인 프로그래밍에서는 굳이 이 컴포넌트를 사용해야 할 필요가 없다.

## ◘ 속성
Lable 컴포넌트가 가지는 모든 속성을 가지며 이 외에 Handle 등의 저수준 속성을 추가로 가진다. BorderStyle 속성을 사용하면 컴포넌트 주변에 삼차원 장식을 할 수 있다는 정도의 차이점만 있다.




# 29.ControlBar

## ◘ 정의
툴바 컴포넌트의 도킹 사이트로 사용되는 컴포넌트이다. Picture 속성에 그림을 대입해 주면 툴바 배경에 그림을 넣을 수도 있다.

## ◘ 속성

### AutoDrag:Boolean;
이 속성이 True 이면 컨트롤 밴드가 마우스 드래그에 의해 도킹 해제될 수 있다.

# 30.Chart

## ◘ 정의
이 컴포넌트는 David Berneda 라는 사람이 만든 라이브러리를 볼랜드에서 라이센스하여 델파이에 포함시킨 것이다. 이 컴포넌트에 대한 자세한 사항은 델파이 도움말에 포함되어 있는 TeeChart User Guide 를 참고하기 바란다. 웬만한 책 한 권 분량의 도움말이 게재되어 있으며 컴포넌트의 부피가 패키지 하나의 분량과 맞먹을 정도이다.

# 31. TabControl

## ◘ 정의
탭셋과 개념적으로 동일한 기능을 가지며 모양은 노트북 형태를 띠고 있다. 상단에 각 페이지의 제목이 탭 형태로 나타난다. 주의할 것은 이 컴포넌트는 어디까지나 탭셋일 뿐이지 각 탭에 따른 여러 개의 페이지를 생성해 주지는 않는다는 점이다. 여러 페이지로 구성된 대화상자를 만들려면 PageControl 컴포넌트를 사용해야 한다. 탭셋을 여러 개 만들어도 생성되는 페이지는 하나뿐이다.
새로운 탭을 만들려면 Tabs 속성에 페이지의 문자열을 입력해 주며 디자인중에 탭을 변경하려면 TabIndex 속성을 탭 번호로 바꾸어 준다. Enabled 속성을 False로 변경하면 탭 컨트롤에 속한 모든 탭을 사용할 수 없게 된다.

## ◘ 속성

### Tabs:TStrings;

탭 컨트롤의 상단에 나타날 각 탭의 제목 문자열이며 문자열의 개수만큼 새로운 탭이 생성된다. 실행중에도 이 속성을 변경하여 탭을 추가하거나 삭제할 수 있다.

### TabIndex:Integer;

탭 컨트롤에서 현재 선택된 탭의 번호를 가지며 실행중이나 디자인시 언제라도 이 속성을 변경하여 페이지를 전환한다. 이 속성이 0이면 첫 번째 탭이 활성화되어 있는 것이며 1이면 두 번째 탭, 총 탭수-1이면 제일 우측의 마지막 탭이 선택된 것이다. 이 값이 -1이면 어떤 탭도 선택되지 않은 상태이다. 이 속성을 Tabs 배열의 인덱스로 사용하면 탭의 제목 문자열을 읽을 수도 있다. 즉 현재 선택된 탭의 제목 문자열을 알고 싶으면 Tabs[TabIndex]를 읽으면 된다.

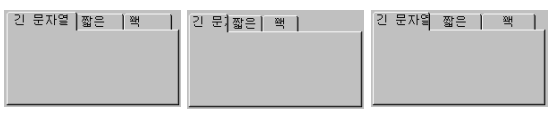
### MultiLine:Boolean;

생성된 모든 탭은 한 줄에 표시되며 한 줄에 다 출력할 수 없을 경우는 우측에 탭을 스크롤할 수 있는 화살표 버튼이 자동으로 나타난다. 그러나 이 속성이 True이면 스크롤을 사용하지 않고 여러 줄을 사용하여 탭을 겹쳐서 표현한다. 탭이 8개 있는 경우 이 속성에 따른 탭 컨트롤의 모양은 다음과 같이 달라진다.


False True

### TabWidth: Smallint;

탭 하나의 가로폭을 픽셀 단위로 설정한다. 디폴트값은 0이며 이는 입력된 제목 문자열의 길이에 맞게 크기를 자동으로 조절한다는 뜻이다. 다음은 이 속성을 여러 가지로 바꾸어 본 것이다.



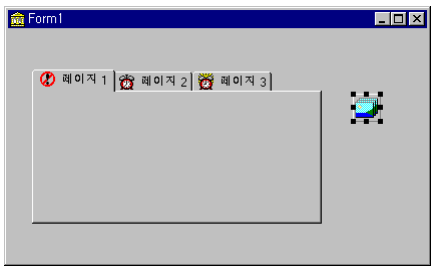

0 40 60

### TabHeight:Smallint;

탭 하나의 높이를 픽셀 단위로 지정한다. 이 값이 0 이면 텍스트 높이에 맞게 자동 설정된다.

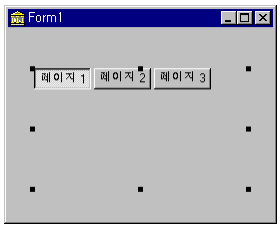
### Images:TCustomImageList;

탭에 같이 출력될 이미지의 리스트를 지정한다. 같은 폼에 이미지 리스트 컴포넌트를 배치해 두고 이미지를 읽어온 후 이 속성에 이미지 리스트의 컴포넌트를 지정해 주면 이미지 리스트상의 이미지가 탭과 순서대로 대응되어 탭 위에 나타나게 된다.

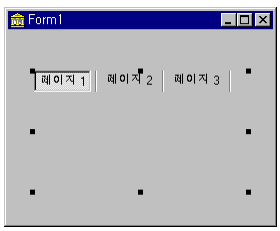



### Style:TTabStyle;

탭 컨트롤의 스타일을 지정하며 세가지 선택 가능한 값이 있다. 이 속성에 fsTabs를 선택하면 표준 탭 모양이 되며 fsButtons를 선택하면 탭 대신 버튼이 나타나며 이 버튼들을 눌러 페이지를 교체한다.




fsFlatButtons를 선택하면 납작한 모양의 버튼이 대신 사용된다.

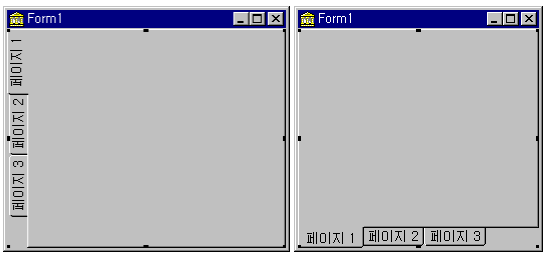



### MultiSelect:Boolean;

여러 개의 탭이 동시에 선택될 수 있는지를 지정한다. 이 속성은 Style이 fsTabs일 경우에는 사용할 수 없다.

### HotTrack:Boolean;
마우스 커서 아래쪽의 탭이 강조될 것인가를 지정한다. 이 속성의 디폴트값은 False이지만 이 속성을 True로 바꾸어주면 마우스 커서 아래쪽의 탭은 파란색으로 강조된다.

### TabPosition:TTabPosition;
탭의 위치를 설정한다. 디폴트로 탭은 위쪽에 배치되지만 아래쪽이나 좌우에도 배치할 수 있다. 가능한 값은 tpTop, tpBottom, tpLeft, tpRight 네 가지가 있다. 다음은 각각 왼쪽과 아래쪽에 탭을 배치해 본 것이다.




탭이 좌우로 배치될 때 탭 컨트롤의 폰트는 반드시 트루타입만 사용해야 한다. 그렇지 않을 경우 탭 제목이 제대로 출력되지 않을 수도 있다.

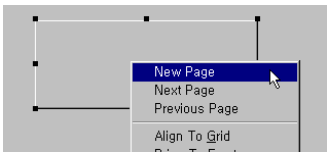
## ◘ 이벤트

### OnChange
탭이 변경되고 난 후에 발생하는 이벤트이다. 이 이벤트에서 페이지를 변경해 주는 처리를 하면 된다.

### OnChanging
탭이 변경되기 직전에 발생하는 이벤트이다. 탭이 변경되기 전에 변경 가능한지를 점검해 보고 그렇지 않을 경우 탭 변경을 거부할 수도 있다. 참조 인수로 전달되는 AllowChange 인수에 False를 대입해 주면 탭 변경이 거부된다. 또한 이 이벤트는 탭이 변경되기 전에 현재 페이지의 정보를 저장하기 위한 용도로도 사용된다.

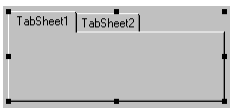
# 32. PageControl

## ◘ 정의
16비트 델파이에서 사용하던 TabbedNoteBook을 대체하는 컴포넌트이며 기능상으로 동일하지만 사용 방법이 조금 다르다. 여러 개의 페이지로 구성된 다중 페이지 대화상자를 구성할 때 사용하며 TTabSheet 오브젝트 여러 개를 겹쳐서 나타낸다. TTabSheet 오브젝트는 각각의 개별적인 페이지에 해당하는 별도의 오브젝트이며 페이지를 추가하면 자동으로 생성된다. 새로운 페이지를 만들려면 스피드 메뉴에서 New Page 메뉴를 사용한다.




New Page 항목을 선택하면 TabSheet라는 페이지가 생성되며 상단에 페이지 탭이 나타난다. 계속해서 New Page를 사용하여 새로운 페이지를 생성할 수 있다. 각 페이지의 제목 문자열은 TabSheet 오브젝트의 Caption 속성으로 지정하며 TabSheet를 선택한 후 오브젝트 인스펙터를 사용하면 된다. 주의할 것은 페이지 컨트롤은 내부에 탭 시트 오브젝트를 포함하는 형태를 띠고 있으며 페이지 컨트롤이 선택된 경우와 탭 시트가 선택된 경우가 다르다는 점이다. 다음 두 그림을 비교해 보아라.

탭 시트 선택



페이지 컨트롤 선택

탭의 제목을 변경하려면 탭 시트가 선택된 상태에서 오브젝트 인스펙터를 사용하며 페이지 컨트롤의 속성을 변경할 때는 페이지 컨트롤이 선택된 상태에서 오브젝트 인스펙터를 사용해야 한다. 이미 작성한 페이지를 삭제하려면 탭 시트가 선택된 상태에서 Del 키를 누른다. 즉 탭 시트 오브젝트를 삭제하면 된다.
활성 페이지는 페이지 컨트롤 선택 상태에서 ActivePage 속성을 변경하거나 마우스로 탭을 클릭하여 선택한다. 마우스로 탭을 클릭하여 활성 페이지를 바꾸는 것은 디자인중에도 가능하다. 스피드 메뉴의 Next Page, Previous Page 항목을 사용해도 활성 페이지를 변경할 수 있다.
TabControl 컴포넌트와 마찬가지로 TCustomTabControl 클래스로부터 파생되었으므로 Style, TabHeight, TabPosition, HotTrack. Images, Multiline 등의 속성은 탭 컨트롤의 그것들과 동일하다. OnChange, OnChanging 이벤트도 마찬가지다.

## ◘ 속성

### ActivePage:TTabSheet;
출력할 페이지의 이름을 지정하며 이는 TTabSheet 오브젝트의 NAME 속성을 사용한다.

실 읽 PageCount:Integer;
페이지 컨트롤에 포함된 페이지의 개수를 조사한다.

실 읽 Pages[I]:TTabSheet;
페이지 컨트롤의 각 페이지에 해당하는 탭 시트 배열형 속성이며 이 속성을 읽어 각 페이지의 탭 시트를 액세스한다.

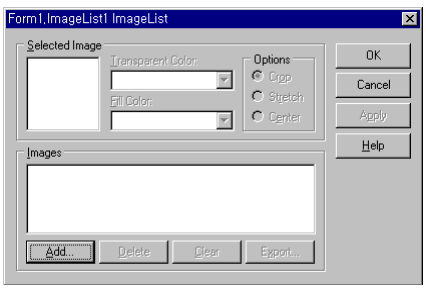
MultiLine:Boolean;
생성된 모든 페이지는 한 줄에 표시되며 한 줄에 다 출력할 수 없을 경우는 우측에 탭을 스크롤할 수 있는 화살표 버튼이 자동으로 나타난다. 그러나 이 속성이 True이면 스크롤을 사용하지 않고 여러 줄을 사용하여 탭을 겹쳐서 표현한다. TabControl의 경우와 동일한 의미를 가진다.

TabWidth:Integer;
탭 하나의 가로폭을 픽셀 단위로 설정한다. 디폴트값은 0이며 입력된 제목 문자열의 길이에 맞게 크기를 자동으로 조절한다는 뜻이다.

# 33. ImageList

## ◘ 정의

그래픽 이미지 여러 장을 담는 컴포넌트이다. 이 컴포넌트를 폼에 배치하면 아이콘만 나타나며 실행중에 아이콘은 보이지 않는다. 그래픽 파일을 읽어오려면 폼에서 이 컴포넌트 아이콘을 더블클릭한다. 그래픽 파일을 불러올 수 있는 다음과 같은 대화상자를 보여줄 것이다.




이 대화상자의 Add 버튼을 사용하여 그래픽 파일을 순서대로 불러온다. 읽을 수 있는 그래픽 파일은 ICO, BMP, WMF 세가지 종류가 있다. 그래픽 파일의 크기는 디폴트로 16x16 의 크기를 가지지만 Width, Height 속성을 변경하면 마음대로 크기를 설정할 수 있다.

## ◘ 속성

Width, Height:Integer;
이미지 리스트에 포함될 그래픽 파일의 폭과 높이를 설정하며 디폴트값은 16x16이다. 이 속성은 그래픽 파일을 읽어들이기 전에 먼저 설정해 주어야 한다. 이 속성을 변경하면 이미지 리스트에 포함된 모든 그래픽 파일이 지워진다.

AllocBy:Integer;
이미지 리스트는 실행중에도 이미지를 불러와 추가할 수 있다. 새 이미지가 추가될 때마다 이미지 리스트는 여분의 메모리를 추가로 할당해 놓는데 이 속성은 한번에 몇 개의 이미지만큼 메모리를 할당할 것인가를 지정한다. 이 값을 1 로 설정해 두면 이미지 하나가 추가될 때마다 메모리가 재할당되기 때문에 속도상으로 불리하며 이 값을 지나치게 큰 값으로 설정해 두면 메모리가 낭비될 수도 있다. 이미지 추가가 얼마나 자주 일어나는가에 따라 이 속성을 적절하게 설정해 주어야 한다. 그러나 이 속성은 이미지 리스트의 내부적인 메모리 관리 방법을 지정하는 것에 불과하므로 기능 자체에는 영향을 주지 않는다.

BkColor:TColor;
이미지를 그릴 때 사용할 배경 색상을 지정한다. 이 속성이 clNone 이면 배경이 투명하게 그려진다. Masked 속성이 False 일 경우 이 속성은 의미가 없다.

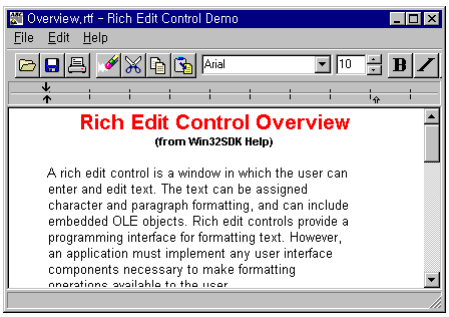
Masked:Boolean;
이미지 리스트가 마스크를 포함할 것인가를 지정한다. 마스크 부분은 투명하게 그려지거나 아니면 bkColor 로 지정한 색상으로 그려진다.

ShareImage:Boolean;
이미지 리스트를 공유할 것인가를 지정하며 디폴트로 공유하지 않는다. 이미지 리스트는 탭 컨트롤, 메뉴, 리스트 뷰, 트리 뷰 등 여러 종류의 컨트롤에 이미지를 제공해 주는데 하나의 이미지 리스트를 여러 개의 컨트롤이 동시에 사용할 수도 있다. 하나의 컨트롤이 이미지 리스트를 독점적으로 사용하고 있다면 이미지 리스트가 파괴될 때 핸들도 파괴하는 것이 정상이지만 공유되고 있다면 그렇지 않아야 한다. 이 경우 이미지 리스트를 최후로 사용한 컨트롤이 파괴될 때 이미지 리스트도 같이 파괴해 주어야 한다.

# 34. RichEdit

## ◘ 정의

메모 컴포넌트와 모든 면에서 동일하다. 메모 컴포넌트가 가지는 대부분의 속성을 가지며 기능도 유사하다. 단 Rich Text 형식의 파일(RTF)를 표시하거나 편집할 수 있다.




다양한 크기와 모양의 문자를 사용할 수 있고 문단 모양까지 조정할 수 있어 간단한 워드 프로세서를 제작할 수도 있다. 실행중에 Print, FindText 등의 메소드를 사용하여 인쇄를 하거나 문자열 검색을 하기도 한다.

## ◘ 속성

대부분의 속성이 메모 컴포넌트와 유사하다.

# 35. TrackBar

## ◘ 정의

스크롤 바와 기능적으로 유사하며 사용 방법도 비슷한 컨트롤이다. 현재 값의 위치를 나타내는 슬라이더를 가지며 이 슬라이더를 드래그하여 트랙 바의 값을 변경한다. 그러나 스크롤 바에 있는 양끝의 화살표는 없다. 값을 대충 입력받을 필요가 있는 볼륨 조절이나 양의 정도를 나타낼 때 이 컨트롤을 사용하며 스크롤 바에 비해 훨씬 더 미적인 장식 효과가 많이 준비되어 있다. 포커스를 가지고 있을 때는 키보드를 사용하여 값을 조절할 수도 있다.

## ◘ 속성

### Orientation:TTrackBarOrientation;

수직, 수평 두 가지 모양 중 하나를 선택할 수 있으며 디폴트는 수평이다.


### Mix,Max:Integer;

트랙 바의 좌측 최소값(또는 상단)과 우측 최대값(또는 하단)을 지정하며 이 값이 트랙 바로 선택가능한 값의 범위이다. 디폴트로 이 값은 0~10으로 되어 있다.

### LineSize:Integer;

키보드의 커서 이동키를 누를 때 증감할 눈금의 양이며 디폴트는 1이다.

### PageSize:Integer;

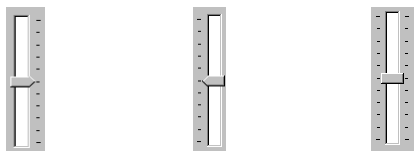
키보드의 PgUp, PgDn 키를 누를 때 증감할 눈금의 양이며 디폴트는 2이다.

### TickMarks:Integer;

트랙 바의 눈금 표시를 어느쪽에 나타낼 것인가를 설정한다.

| 속성값 | 의미 |
|---|---|
| tmBottomRight | 하단이나 오른쪽에 눈금이 나타난다. |
| tmTopLeft | 상단이나 왼쪽에 눈금이 나타난다. |
| tmBoth | 양쪽에 눈금이 나타난다. |

트랙 바의 모양이 수직인 경우 각 속성에 따른 모양은 다음과 같다.

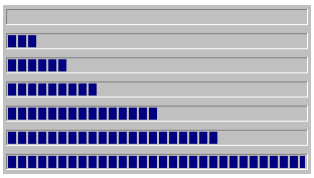


### TickStyle:TTickStyle;
눈금 표시의 표시 형태를 변경하며 다음과 같은 속성값이 있다. 그림도 함께 보인다.

| 속성값 | 의미 |
|---|---|
| tsAuto | 각각의 값에 눈금 표시를 출력한다. |
| tsManual | 트랙 바의 양쪽 끝에 눈금 표시를 출력한다. |
| tsNone | 눈금 표시를 전혀 출력하지 않는다. |

## 36. ProgressBar

### ◘ 정의
게이지와 유사하며 작업의 진행 상태를 막대 그래프 형식으로 보여준다. 윈95의 설치 화면에서 흔히 볼 수 있는 진행 상태 게이지이다.



### ◘ 속성

### Min,Max:Integer;
최소값과 최대값을 지정하며 디폴트는 0~100까지이다.

### Position:Integer;
현재 진행 정도를 나타낸다.

### Step:Integer;
StepInc 메소드로 Position 속성값을 증가시킬 때의 증가값을 지정하며 디폴트값은 10이다.

## 37. UpDown

### ◘ 정의
델파이 1.0에서 제공하는 스핀 버튼과 동일한 기능을 가지며 마우스로 정수값을 정확하게 증감시킬 때 사용한다.

### ◘ 속성

### Min,Max:SmallInt;
최소, 최대값을 지정한다.

### Position:SmallInt;
이 컴포넌트가 현재 가지고 있는 값이다.

### Orientation:TUDOrientation;
증감 화살표를 수직으로 배치할 것인가 수평으로 배치할 것인가를 지정한다.

### Wrap:Boolean;
끝값과 처음값을 연결하여 값이 순환되도록 한다. 예를 들어 끝값 100에서 증가 버튼을 누르면 다시 0이 되도록 한다.

### Associate:TWinControl;
UpDown 컴포넌트와 연결될 컴포넌트를 설정한다. 연결이 설정되면 UpDown 컴포넌트가 연결된 컴포넌트 옆으로 이동하며 UpDown에서 값이 변경될 경우 연결된 컴포넌트의 Text 또는 Caption 속성이 같이 변경된다. 가장 흔한 연결의 예는 UpDown과 에디트를 연결한 경우이다.


이렇게 연결해 놓으면 에디트로 직접 값을 입력할 수도 있고 UpDown으로 값을 조절할 수도 있다.

# 38. HotKey

## ◘ 정의

실행중에 쇼트컷을 입력받고자 할 때 사용한다. 이 컴포넌트가 포커스를 가지고 있을 때 사용자는 Ctrl, Alt, Shift 키와 일반키의 조합으로 쇼트컷을 입력할 수 있으며 이렇게 입력된 쇼트컷은 HotKey 속성에 보관된다. HotKey 속성에 보관된 쇼트컷은 메뉴 항목 컴포넌트의 ShortCut 속성에 곧바로 대입될 수 있으므로 실행중에 사용자가 단축키를 변경시키도록 해준다.

## ◘ 속성

### HotKey:TShortCut;

현재 선택된 쇼트컷이며 디폴트값은 Alt-A이다. 핫키 컴포넌트가 포커스를 가지고 있을 때 사용자가 다른 키를 누르면 이 속성은 다시 설정된다.

# 39.Animate

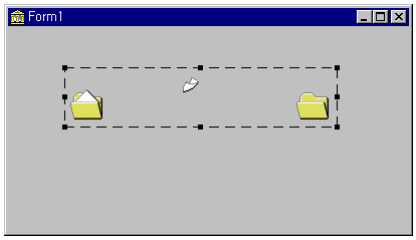
## ◘ 정의

윈도우즈의 애니메이션 컨트롤을 캡슐화한 컴포넌트이다. 간단한 동영상을 재생하고자 할 때 이 컴포넌트를 사용한다. 단 이 컴포넌트와 함께 사용하는 동영상은 소리를 재생할 수 없으며 비디오 스트림이 하나만 있어야 하고 RLE 포멧으로 압축되어 있어야 한다. 재생의 대상이 되는 동영상은 리소스의 형태나 AVI 파일 또는 Shell32.dll 로부터 얻을 수 있다. 각 동영상은 CommonAVI, FileName, ResID, ResName 속성으로 지정하는데 이 속성들은 상호 배타적이므로 한쪽 속성에 값이 대입되면 나머지 속성은 모두 0 가 된다. 즉 파일 이름을 지정하면 리소스에 있는 동영상은 읽어올 수 없다.

## ◘ 속성

### Active:Boolean;

이 속성을 True 로 바꾸면 재생을 시작한다. 디자인 중에도 이 속성을 True 로 변경하여 동영상을 재생할 수 있다. 이 속성을 사용하기 전에 이 컴포넌트는 반드시 열려있는 상태여야 한다.




### Center:Boolean;

컴포넌트의 중앙에서 동영상을 재생하도록 한다. 이 속성이 False 이면 좌상단에서 재생된다.

### CommonAVI:TCommonAVI;

재생의 대상이 되는 윈도우즈 표준 동영상을 지정한다. 다음과 같은 값들이 가능하다.

| 값 | 그림 |
|---|---|
| aviCopyFile | |
| aviCopyFiles | |
| aviDeleteFile | |
| aviEmptyRecyle | |
| aviFindComputer | |
| aviFindFile | |
| aviFindFolder | |
| aviNone | 재생하지 않음 |

### FileName:TFileName;

재생하고자 하는 AVI 파일을 지정한다. 문자열 형태로 파일 이름을 지정하되 드라이브명과 디렉토리명을 포함할 수 있다.

### ResID:Integer;

재생하고자하는 AVI 동영상의 리소스 ID 를 지정한다. 이 속성을 사용하기 전에 리소스를 가진 인스턴스 핸들이 ResHandle 속성에 먼저 대입되어야 한다.

### ResName:string;

재생하고자 하는 AVI 동영상의 리소스 이름을 지정한다. 이 속성을 사용하기 전에 리소스를 가진 인스턴스 핸들이 ResHandle 속성에 먼저 대입되어야 한다.

### ResHandle:TInteger;

리소스에서 AVI 동영상을 가져올 경우 이 리소스를 가진 인스

턴스 핸들을 대입해 준다. 만약 이 속성이 리소스 핸들을 지정하지 않을 경우 에니메이트 컴포넌트는 메인 응용 프로그램의 인스턴스 핸들을 사용한다.

실 읽 FrameCount:Integer;
재생할 동영상의 총 프레임 수를 조사한다.

실 읽 FrameHeight:Integer;
재생할 동영상의 높이를 픽셀 단위로 조사한다.

실 읽 FrameWidth:Integer;
재생할 동영상의 폭을 픽셀 단위로 조사한다.

실 Open:Boolean;
재생할 동영상이 메모리로 읽혀졌는지를 조사한다. 이 속성이 True 이면 첫 프레임을 읽어와 화면으로 출력해 주며 Active 속성을 변경하거나 Open 메소드를 호출하여 동영상을 재생할 수 있다. 이 속성을 False 로 설정하면 AVI 클립이 사용하던 리소스를 해제한다.

Repetitions:Integer;
반복 재생할 횟수를 지정한다. 이 속성이 지정하는 횟수만큼 반복하되 0 일 경우는 무한번 재생한다.

StartFrame:Integer;
재생을 시작할 시작 프레임수를 지정한다.

StopFrame:Integer;
재생을 중지할 끝 프레임수를 지정한다. 이 속성이 0 이면 AVI 클립의 끝까지 재생된다. 동영상의 일부분만 연주하고자 할 때 StartFrame, StopFrame 속성을 사용한다.

Timer:Boolean;
동영상 재생을 타이머 메시지에 맞추어 연주하도록 한다. 이 속성이 False 이면 애니메이트는 별도의 분리된 스레드를 만들어 비동기적으로 재생한다. 동영상을 다른 동작과 함께 동기화시키고자 할 때 이 속성을 사용한다.

TransParent:Boolean;
AVI 파일에 지정된 배경색을 Color 속성 또는 부모 컨트롤의 색상(ParentColor 속성이 True 일 경우)으로 변경한다. 이 속성을 사용하면 투명한 동영상을 재생할 수 있다.

## ◘ 메소드

■ procedure Play(FromFrame, ToFrame: Word; Count: Integer);
동영상을 연주한다. 시작 프레임과 끝 프레임은 FromFrame, ToFrame 인수로 전달하며 재생 횟수는 Count 인수로 전달한다. 이 메소드의 인수는 애니메이트 컨트롤의 StartFrame, StopFrame 속성보다 우선 순위가 높으므로 속성 설정에 상관없이 동영상의 일부분을 재생할 수 있다. 이 메소드가 호출될 때 애니메이트가 열려있지 않은 상태이면 Open 속성을 True 로 강제 설정한다.

■ procedure Stop;
재생을 중지한다.

■ procedure Reset;
시작 프레임, 끝 프레임 지정을 모두 리셋하고 AVI 클립을 디폴트 상태로 돌려놓는다.

■ procedure Seek(Frame: SmallInt);
Frame 인수로 지정한 프레임으로 이동한다.

## ◘ 이벤트

OnOpen
애니메이트 컨트롤이 열릴 때 발생하는 이벤트이다.

OnClose
애니메이트 컨트롤이 닫힐 때 발생하는 이벤트이다.

OnPlay
재생을 시작할 때 발생하는 이벤트이다.

OnStop
재생을 중지할 때 발생하는 이벤트이다.

# 40.DateTimePicker

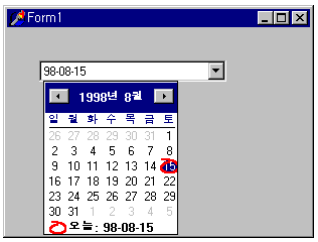
## ◘ 정의

시간과 날짜를 입력받는 컴포넌트이다. 콤보 박스와 비슷한 모양을 하고 있으나 날짜와 시간을 설정할 수 있는 기능이 있다. 날짜를 설정할 경우 콤보 박스를 열면 달력이 나타나며 시간을 설정할 경우 스핀 버튼이 나나탄다.

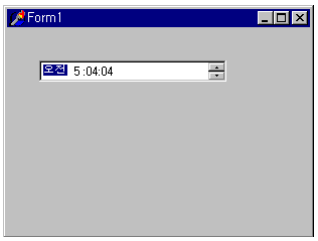
## ◘ 속성

### Kine:TDateTimeKind;

시간을 설정하는 용도로 사용할 것인지 날짜를 설정하는 용도로 사용할 것인지를 지정한다. dtkDate, dtkTime 둘 중 하나를 선택할 수 있다. 다음은 이 속성을 dtkDate 로 설정하여 날짜를 입력받도록 한 것이다.




이렇게 입력된 날짜는 Date 속성으로 읽는다. 다음은 이 속성을 dtkTime 으로 설정하여 시간을 입력받도록 한 것이다. 마우스로 시/분/초의 요소를 선택한 후 오른쪽의 스핀 버튼으로 시간을 설정한다.

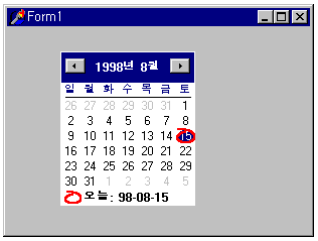



사용자가 설정한 시간은 Time 속성으로 읽는다.

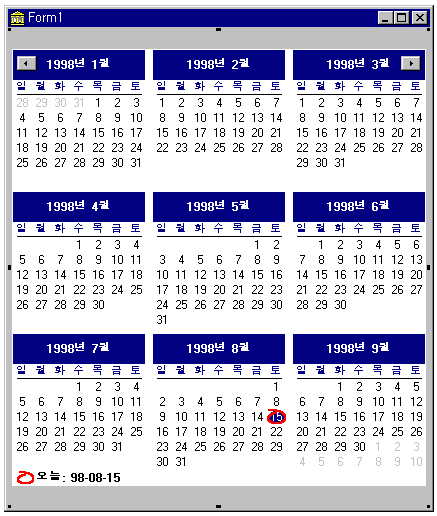
# 41.MonthCalendar

## ◘ 정의

날짜를 선택할 수 있는 달력이다. DateTimePicker 컴포넌트에서 사용하는 달력과 동일한 달력이다.




상단 좌우측에 있는 화살표 버튼을 클릭하여 달을 증감하며 연도를 클릭한 후 상하 화살표 버튼을 클릭하여 연도를 증감한다. 날짜는 달력에서 직접 클릭하여 변경한다. 크기를 크게 만들면 한꺼번에 여러 달을 볼 수도 있다.



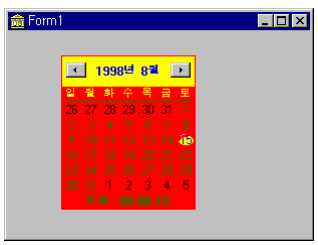

## ◘ 속성

### Date:TDate;

달력에 표시된 날짜를 나타낸다. 이 속성을 읽으면 사용자가 선택한 날짜를 알 수 있으며 이 속성을 변경하면 달력이 열릴 때 보여줄 날짜를 변경할 수 있다.

### CalColors:TMonthCalColors;

달력의 색상을 변경한다. BackColor, MonthBackColor, TextColor, TitleBackColor, TrailingTextColor 등의 세부 속성을 가지고 있으며 모두 TColor 형이다. 각각의 세부 속성은 이름 그대로의 의미를 가지고 있다. 다음은 필자가 이 속성을 사용하여 달력의 모양을 아주 섹시하게 바꿔 본 것이다.



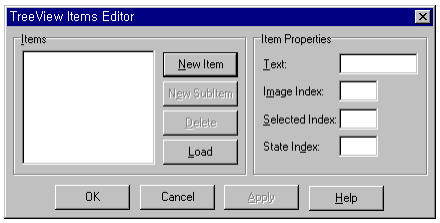
### FirstDayOfWeek:TCalDayOfWeek;

일주일의 시작을 지정한다. 대부분의 경우 일요일이 일주일의 시작이지만 이 속성을 변경하면 월요일이 일주일의 처음에 오도록 할 수도 있다.

# 42. TreeView

## ◘ 정의

탐색기의 좌측에 있는 디렉토리 리스트와 유사한 계층별 트리 구조를 보여준다. 각 항목은 Items 속성에 입력하며 이 속성을 더블클릭하여 항목 편집기를 연다.




이 대화상자에서 New Item, New SubItem 버튼을 사용하여 다중 레벨의 트리 구조를 입력한다. 실행중에 각 노드는 +,- 등의 확장 및 축소 가능 표시를 보여준다.
트리 뷰의 각 노드는 TreeNode 오브젝트이며 이 오브젝트의 AddChild, AddChildObject 등의 메소드를 사용하여 실행중에 노드를 추가할 수도 있다.

## ◘ 속성

### Indent:Integer;

부모 노드와 자식 노드 간의 거리를 픽셀 단위로 지정한다. 디폴트는 적당한 거리만큼 떨어진 19픽셀이다.

19

30

40

### ShowButton:Boolean;

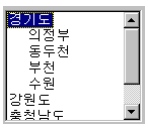
확장 및 축소 가능 표시를 각 노드 옆에 보여준다. 이 속성이 False일 경우 확장, 축소 가능 표시가 나타나지 않으며 노드를 직접 더블클릭하여 확장, 축소해야 한다.

### ShowLines:Boolean;

노드 간의 연결 가지를 보여준다.

버튼 숨김

가지 숨김

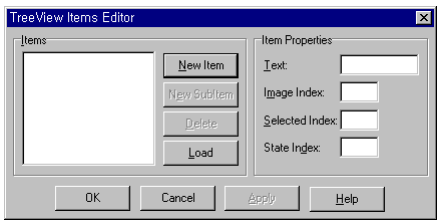


둘 다 숨김

# 43. ListView

## ◘ 정의

여러 가지 항목을 ViewStyle 속성에 따라 다양한 방법으로 보여주는 컴포넌트이다. 오브젝트 인스펙터에서 Items 속성을 더블클릭하여 나타나는 항목 편집기를 사용하여 항목을 입력한다.




## ◘ 속성

### ViewStyle:TViewStyle;

입력된 항목을 어떤 형식으로 보여줄 것인가를 설정하며 다음과 같은 값들 중 하나를 선택한다.

| 속성값 | 의미 |
|---|---|
| vsIcon | 각 항목은 문자열과 큰 아이콘으로 표현되며 실행중에 항목을 드래그하여 위치를 옮긴다. |
| vsSmallIcon | 각 항목은 문자열과 작은 아이콘으로 표현되며 실행중에 항목을 드래그하여 위치를 옮긴다. |
| vsList | 작은 아이콘이 문자열의 좌측에 나타나며 실행중에 드래그하여 항목의 위치를 옮길 수 없다. |
| vsReport | 표 형식으로 항목을 보여준다. |

### Columns:TListColumns;

트리 뷰의 열 제목 문자열을 가지며 열 제목을 출력하기 위해서는 ViewStyle 속성이 반드시 vsReport여야 한다. 오브젝트 인

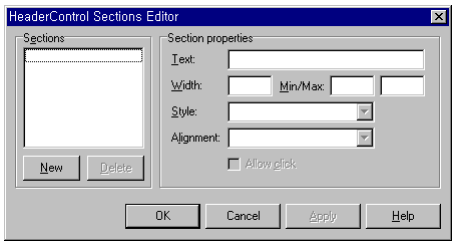
스펙터에서 이 속성을 더블클릭하면 열 제목을 입력할 수 있는 대화상자가 나타난다.

### LargeImage:TImageList;
ViewStyle 속성이 vsIcon일 경우 문자열 옆에 표시할 아이콘 목록을 지정한다.

### SmallIcon:TImageList;
ViewStyle 속성이 SmallIcon일 경우 문자열 옆에 표시할 아이콘 목록을 지정한다.

## 44. HeaderControl

### ◘ 정의
기존의 헤더 컴포넌트와 유사하되 각 섹션이 THeaderSection이라는 오브젝트로 구성되어 있다. 새로운 섹션을 추가하려면 오브젝트 인스펙터에서 Sections 속성을 더블클릭하여 섹션 편집기를 연다.



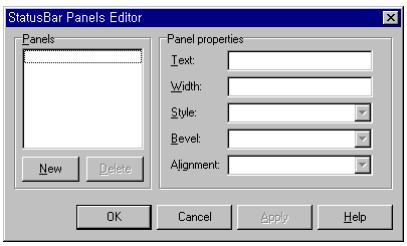

이 대화상자에서 섹션을 생성하고 각 섹션의 속성을 변경한다.

### ◘ 속성

### Sections:THeaderSection;
헤더에 속한 섹션의 목록이다.

## 45. StatusBar

### ◘ 정의
응용 프로그램의 밑 부분에 흔히 배치되는 상태란이다. 이 컴포넌트의 디폴트 Align 속성이 alBottom이므로 폼에 배치하면 밑 부분에 자동 배치된다. 하나의 상태란에 여러 가지 정보를 동시에 표현하고자 할 때는 Panels 속성을 사용한다. 오브젝트 인스펙터에서 이 속성을 더블클릭하면 다음과 같은 패널 편집기가 열린다.

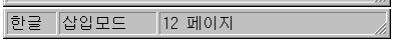



이 대화상자에서 패널의 제목을 정의하며 폭과 정렬 상태, 베벨 모양을 변경한다.
SimplePanel 속성에 따라 하나의 긴 문장을 나타낼 수도 있고 여러 개의 칸으로 구분하여 복합적인 여러 가지 정보를 동시에 나타낼 수도 있다.

### ◘ 속성

### SimplePanel:Boolean;
이 속성이 True이면 칸을 나누지 않고 상황선 전체에 하나의 문자열만 나타내며 False이면 칸을 나누어 사용한다. 실행중에도 이 속성을 바꿀 수 있으므로 긴 메시지를 출력할 때 잠시만 사용해도 된다.

True


False


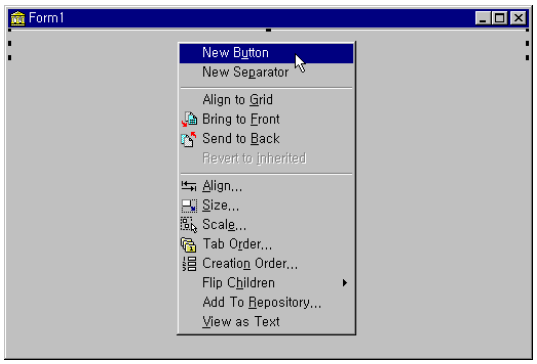

### SimpleText:string;
SimplePanel 속성이 True인 경우 상황선에 나타날 문자열을 입력한다.

### SizeGrip:Boolean;
상황선의 우하단에는 폼의 크기를 조절할 수 있는 삼각형 모양의 크기 조절 핸들이 있다. 이 속성을 False로 바꾸면 이 핸들이 표시되지 않는다. 상황선의 Align 속성이 alBottom일 경우만 이 핸들은 크기 조절용으로 사용되며 그 외의 정렬을 가지면 단순한 장식에 불과하다.

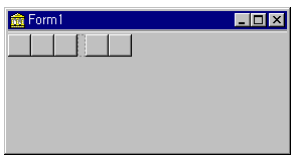
# 46.Toolbar

## ◘ 정의

폼의 상단에 밀착되어 툴바를 구성한다. 툴바에 포함되는 비트맵은 통상 툴 버튼 컴포넌트(TToolButton)나 콤보 박스 등의 다른 컴포넌트가 배치될 수도 있다. 툴바에 배치되는 툴 버튼은 높이와 폭이 모두 같아야 하며 툴 버튼 외의 컨트롤이 배치되더라도 높이는 일정해야 한다. 툴바에 툴 버튼을 배치할 때는 툴바의 팝업 메뉴를 사용한다.




New Button 을 선택하면 새 툴 버튼이 배치되고 New Separator 는 버튼 사이에 공백을 삽입한다. 다음은 버튼 세 개와 공백 하나 그리고 버튼 두 개를 배치해 본 것이다.




버튼에 그려지는 이미지는 Images 속성이 지정하는 이미지 리스트로부터 제공된다.

## ◘ 속성

### Images:TCustomImageList;

툴 버튼에 놓여질 이미지의 집합인 이미지 리스트를 지정한다. 각 툴 버튼의 ImageIndex 속성으로 이미지 리스트 상에 몇번째 이미지를 사용할 것인가를 지정한다.

### HotImages:TCustomImageList;

마우스가 툴 버튼을 가리킬 때 사용될 이미지 리스트를 지정한다. 이 속성이 지정되지 않으면 Images 속성으로 지정한 이미지가 대신 사용된다.

### DisabledImages:TCustomImageList;

툴 버튼이 사용금지되었을 때 나타낼 이미지를 지정한다. 이 속성이 지정되지 않으면 Images 속성으로 지정한 이미지를 흐리게 만든 이미지가 대신 사용된다.

### 실 읽 ButtonCount:Integer;

툴바에 속한 툴 버튼의 개수를 조사한다.

### ButtonHeight:Integer;

툴바에 속한 툴 버튼의 높이를 조사한다.

### 실 읽 Buttons[]:TToolButton;

툴바에 속한 툴 버튼의 배열 속성이다. 읽고자 하는 버튼의 첨자를 넘겨주면 해당 툴 버튼 오브젝트가 리턴된다.

### ButtonWdith:Integer;

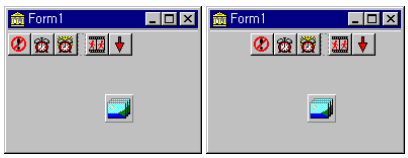
툴바에 속한 툴 버튼의 폭을 조사한다. 공백이나 구분자에게는 적용되지 않으며 버튼에만 적용된다.

### Flat:Boolean;

이 속성이 True 이면 납작한 모양의 툴 버튼을 만들며 툴바가 투명하게 되어 배경의 그림이 보이게 된다. 툴 버튼은 팝업 경계선을 가지며 이 경계선은 마우스가 버튼 위로 올라가면 보이게 된다.이 속성은 COMCTL32.DLL 4.70 이상의 버전을 필요로 한다.

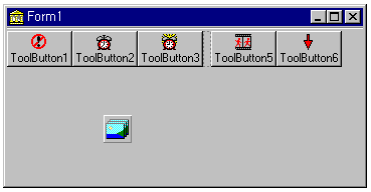
### Indent:Integer;

첫 번째 툴 버튼과 툴바의 좌측변과의 간격을 지정한다. 다음 왼쪽은 디폴트값인 0 을 그대로 사용한 것이며 오른쪽은 Indent 속성을 40 으로 바꾸어 본 것이다.



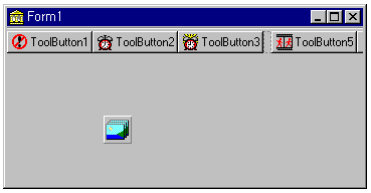

### ShowCaption:Boolean;

툴 버튼에 캡션을 같이 나타내도록 한다.




### List:Boolean;

툴 버튼의 갭션을 이미지의 오른쪽에 나타내도록 한다. 이 속성이 False 이면 캡션이 이미지의 아래쪽에 나타난다.



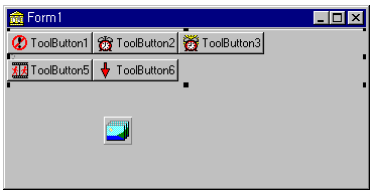

단 이 속성은 ShowCaption 속성이 True 일 때만 의미가 있다.

### Transparent:Boolean;

툴바를 투명하게 만든다. 이 속성은 툴바 그 자체에 적용되는 것이지 툴 버튼에 적용되는 것이 아님을 유의해야 한다. 가급적이면 이 속성은 툴바가 화면에 나타나기 전에 설정하는 것이 좋다.

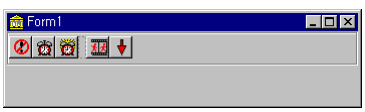
### Wrapable:Boolean;

툴바의 오른쪽 변까지 버튼이 가득 찼을 때 다음 줄로 자동으로 내리도록 한다.




### EdgeBorder:TEdgeBorder;

툴바의 상하좌우에 경계선을 그릴 것인가를 지정한다. ebLeft, ebTop, ebRight, ebBottom 네 개의 진위형 세부 속성을 가지는데 디폴트는 ebTop 만 선택되어 있다. 다음 그림은 네 가지 속성을 모두 선택했을 때의 모양이다.




### EdgeInner:TEdgeStyle;

경계선 안쪽의 장식을 지정한다. esNone 은 장식이 없는 것이며 esRaised 는 양각, esLowered 는 음각이다.

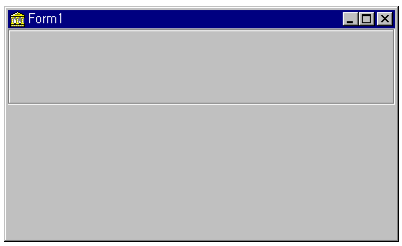
### EdgeOuter:TEdgeStyle;

경계선 바깥쪽의 장식을 지정하며 가능한 값은 EdgeInner 와 동일하다. 두 속성을 잘 조합하면 삼차원 효과를 낼 수 있다. 디폴트로 안쪽은 양각이며 바깥쪽은 음각으로 되어 있다.

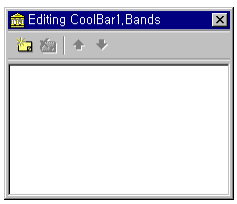
## 47.Coolbar

### ◘ 정의

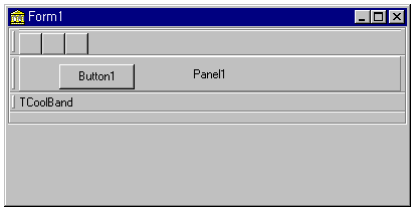
쿨바는 TCoolBand 오브젝트의 컨테이너이며 여러 개의 밴드를 모아두는 장소가 된다. 또한 쿨 밴드는 툴바, 패널 등의 페어런트로 활용되며 툴바는 또한 툴 버튼을 포함할 수 있다. 쿨바는 이런 여러 가지 차일드 컨트롤을 포괄하며 각 차일드 컨트롤이 독립적으로 움직이거나 크기 변경이 가능하도록 해 준다. 쿨바를 폼에 배치하면 폼 상단에 밀착되며 사각 영역만 나타난다.




이 컴포넌트를 더블클릭하면 밴드 편집기가 열리며 이 편집기에서 원하는만큼 쿨 밴드를 생성할 수 있다.




각각의 밴드는 TCoolBand 라는 개별 컴포넌트이며 Control 속성에 툴바, 패널 등의 컴포넌트를 연결하면 쿨 밴드안에 연결된 컴포넌트가 배치된다.

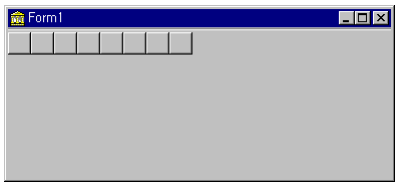



## ◘ 속성

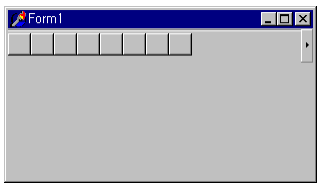
### Bands:TCoolBands;
쿨 밴드 오브젝트의 집합 속성이다. 디자인중에 밴드 편집기로 밴드를 생성하거나 수정한다. 이 속성을 더블클릭하면 밴드 편집기가 열린다.

# 48.PageScroller

## ◘ 정의
툴바와 같이 폭이 좁은 윈도우의 출력 영역을 제공한다. 만약 툴바가 출력 영역보다 더 클 경우 윈도우의 가장자리에 화살표 버튼을 붙여 주며 이 버튼을 사용하여 출력 영역을 스크롤할 수 있다. 스크롤 바와 기능적으로 유사한 컨트롤이다. 다음 그림은 페이지 스크롤러 안에 툴바를 배치한 모양이다.




실행중에 폼의 크기가 줄어들면 툴바를 스크롤할 수 있는 화살표 버튼이 나타난다.




## ◘ 속성

### AutoScroll:Boolean;
마우스 버튼을 누르지 않아도 마우스 커서가 화살표 위에 있기만 해도 스크롤되도록 한다.

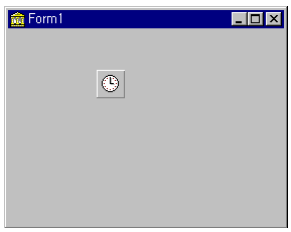
### ButtonSize:Integer;
화살표 버튼의 크기를 지정한다. 디폴트값은 12 이다.

### Control:TWinControl;
페이지 스크롤러 안에 포함된 컨트롤을 지정한다.

### Margin:Integer;
페이지 스크롤러의 가장자리 변과 이 컴포넌트에 포함된 컨트롤과의 간격을 지정한다.

# 49. Timer

## ◘ 정의
일정한 시간 주기로 OnTimer 이벤트를 발생시킨다. 컴포넌트를 폼에 배치하면 시계 모양의 아이콘으로 표시되며 실행중에는 보이지 않는다.




일정 주기로 규칙적인 일을 반복하고자 할 때 이 컴포넌트를 사용한다. 예를 들어 시계를 움직인다거나 규칙적으로 반복되는 이미지의 교체 처리에 OnTimer 이벤트를 사용한다. 이 이벤트가 발생하는 주기는 Interval 속성으로 설정한다. 하나의 폼에 여러 개의 타이머를 배치하여 각기 다른 시간 간격으로 OnTimer 이벤트를 발생시키는 것도 가능하다.

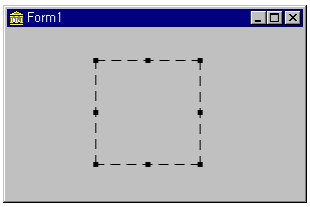
## ◘ 속성

### Interval:Word;
OnTimer 이벤트가 발생할 주기를 지정하며 1/1000초 단위를 사용한다. 디폴트값이 1000이므로 1초에 한번씩 OnTimer 이벤트가 발생하는 셈이다. 지정할 수 있는 시간 간격은 0~65536까지이며 간격이 0일 경우는 OnTimer 이벤트가 발생하지 않는다.

# 50. PaintBox

## ◘ 정의

폼에 LineTo나 Ellipse 등의 그래픽 메소드를 사용하여 그림을 그릴 경우 그림이 출력되는 위치는 폼 전체가 된다. 페인트 박스는 폼과 마찬가지로 그림이 그려지는 영역을 제공하지만 페인트 박스 내부에서만 그리기를 하도록 하여 영역을 제한한다. 이 컴포넌트를 배치하면 다음과 같이 페인트 박스의 영역만 나타나며 실행중에 이 영역 구분은 보이지 않는다.




일단 페인트 박스가 폼에 놓여지면 페인트 박스의 OnPaint 이벤트 핸들러를 사용하여 페인트 박스의 캔버스에만 그림을 그릴 수 있다. 폼에 그림을 그리고자 한다면 폼의 OnPaint 이벤트를 사용해야 한다. 폼과 마찬가지로 그림이 그려지는 캔버스를 가지며 캔버스의 그림 그리기 메소드를 사용한다. 예를 들어 PaintBox1.Canvas.LineTo(100,100); 등과 같이 쓸 수 있다.

# 51. MediaPlayer

## ◘ 정의

멀티미디어 장비를 통제한다. 여기서 말하는 장비란 CD-ROM 드라이브나 VCR과 같은 하드웨어일 수도 있고 WAV나 AVI와 같은 소프트웨어일 수도 있다. 이 컴포넌트를 폼에 놓으면 다음과 같은 모양의 여러 가지 버튼을 가진 패널이 배치된다.

이 버튼들을 분리된 버튼이 아니며 하나의 컴포넌트에 속하는 버튼이다. 각 버튼의 기능은 다음과 같다.

| 버튼 | 기능 |
|---|---|
| Play | 연주를 시작한다. |
| Pause | 잠시 멈춘다거나 연주를 계속한다. |
| Stop | 연주나 녹음을 멈춘다. |
| Next | 다음 트랙으로 이동한다. 트랙이 없는 장비일 경우는 끝으로 이동한다. |
| Prev | 앞 트랙으로 이동한다. 트랙이 없는 장비일 경우는 처음으로 이동한다. |
| Step | 앞쪽 프레임으로 이동한다. |
| Back | 뒤쪽 프레임으로 이동한다. |
| Record | 녹음을 시작한다. |
| Eject | 미디어를 방출한다. |

실행중에 사용자의 버튼 조작에 의해 장비가 동작되거나 멈추기도 하며 또는 코드의 메소드에 의해 통제되기도 한다. 장비의 형태는 DeviceType 속성에 의해 선택되며 이 속성이 dtAutoSelect일 경우 FileName 속성의 파일 확장자에 의해 장비의 종류를 결정한다. 소프트웨어 장비(WAV, AVI 등)일 경우 FileName 속성에 파일 이름을 지정한다. 파일(또는 장비)을 오픈할 때는 Open 메소드를 사용하며 연주할 때는 Play 메소드를 사용한다.

## ◘ 속성

### DeviceType:TMPDeviiceTypes;

멀티미디어 장비의 종류를 결정한다. 선택 가능한 장비의 종류에는 dtAutoSelect, dtAVIVideo, dtCDAudio, dtDAT, dtDigitalVideo, dtMMMovie, dtOther, dtOverlay, dtScanner, dtSequencer, dtVCR, dtVideodisc, or dtWaveAudio 등이 있으며 디폴트는 dtAutoSelect이다. dtAutoSelect일 경우 FileName 속성의 파일 확장자에 따라 장비의 종류를 자동으로 결정한다. 만약 확장자가 없거나 정확하게 되어 있지 않을 경우 이 속성을 직접 설정해 주어야 한다.

### AutoOpen:Boolean;

프로그램이 시작될 때 멀티미디어 장비를 자동으로 오픈시킨다. 이 속성이 False일 경우 Open 메소드를 이용하여 강제로 Open시켜 주어야 한다.

### EnableButtons:TButtonSet;

멀티미디어 컴포넌트에 소속된 버튼들의 활성화 여부를 지정하며 각 버튼 이름의 세부 속성으로 지정한다.

### VisibleButtons:TButtonSet;

멀티미디어 컴포넌트에 소속된 버튼들의 출력 여부를 지정하며 각 버튼 이름의 세부 속성으로 지정한다.

### ColorButtons:TButtonSet;

멀티미디어 컴포넌트에 소속된 버튼들을 컬러로 출력할 것인가 흑백으로 출력할 것인가를 지정한다.

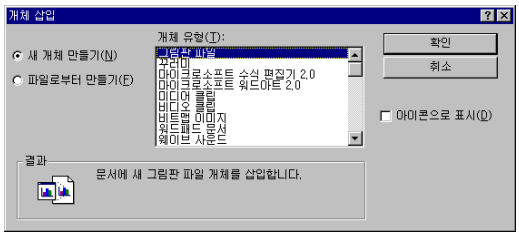
### Wait:Boolean;

멀티미디어 조작 메소드 실행 후에 제어를 프로그램으로 돌릴 것인가를 결정한다. 이 속성이 True이면 메소드의 동작이 완전히 종료되기 전에는 프로그램으로 제어권을 돌려주지 않으며 False이면 메소드 동작이 계속되면서 프로그램으로 제어권을 돌려준다. 디폴트로 Play와 StartRecording 메소드에 대해서는 False이며 나머지는 True이다. 이 속성은 속성을 설정한 후 한 번만 사용할 수 있다. 만약 메소드를 계속 호출할 경우 이 속성을 계속 조정해 주어야 한다.

## 52. OleContainer

### ◘ 정의

연결 또는 포함된 OLE 오브젝트를 담는 컴포넌트이다. 이 컴포넌트를 폼에 배치하고 개체 삽입 대화상자를 통해 개체를 삽입하면 폼에 삽입된 개체가 출력된다. 디자인시나 실행시에 OLE 서버를 호출하여 즉시 개체를 편집할 수 있으며 편집된 개체는 폼에 나타나게 된다. 디자인시에 팝업 메뉴에서 Insert Object 항목을 선택하면 다음과 같은 오브젝트 삽입 대화상자가 나타난다.




이 대화상자에서 OLE 서버를 선택하여 디자인중에 오브젝트를 폼에 배치할 수도 있다.

## 53. DdeClientConv

### ◘ 정의

DDE 서버와 함께 DDE 대화를 구성하려면 이 컴포넌트를 DDEClientItem 컴포넌트와 연결해 주어야 한다. 대화가 성립되려면 DDEService 속성에 서버 프로그램의 이름을 지정하고 DDETopic 속성에 토픽을 지정해 주어야 한다. 오브젝트 인스펙터에서 DDEService나 DDETopic 속성을 더블클릭하면 DDE 연결 대화상자가 나타난다.


이 대화상자에서 서비스와 토픽을 직접 기입하거나 실행중에 SetLink 메소드를 사용하여 서버와 연결한다. PokeData 메소드는 문자열을 연결된 DDE 서버로 보내며 ExecuteMacro 메소드는 연결된 DDE 서버로 매크로를 보낸다.

### ◘ 속성

#### ConnectMode:TDataMode;

DDE 대화를 구성할 방법을 지정한다. DdeAutomatic이면 클라이언트는 실행시에 자동으로 대화를 구성하며 DdeManual이면 사용자가 직접 OpenLink 메소드로 연결해 주어야 한다.

#### DdeService:string;

서비스 이름이며 보통 서버 프로그램의 실행 파일 이름이 사용된다. 필요할 경우 경로가 사용될 수 있으며 확장자 EXE는 붙이지 않는다. 서버가 델파이로 만든 프로그램이라면 서비스는 프로젝트의 이름이 된다.

#### DDETopic:string;

DDE대화의 대상이며 보통 서버 프로그램이 사용하는 파일명이다. 서버가 델파이로 만든 프로그램일 경우 연결된 컴포넌트가 포함된 폼의 이름이(Caption 속성) 토픽으로 사용된다. 그러나 클라이언트가 TDDEServerConv 컴포넌트와 연결되어 있다면 폼의 캡션 대신 이 컴포넌트의 이름이 토픽으로 사용된다.

#### FormatChars:Boolean;

이 속성이 True로 설정되어 있으면 ASCII 코드 8(BS), 9(Tab), 10(LF), 13(CR) 코드가 걸러지며 클라이언트 데이터에 나타나지 않는다. 디폴트는 False로 설정되어 있다.

## 54. DdeClientItem

### ◘ 정의

DDE 대화의 아이템을 정의한다. DDEConv 속성에 TDDEClientConv 컴포넌트를 지정하며 DDE 대화에 연결된다. 또한 대화가 성립되기 위해서는 DDEItem 속성에 DDE 항목의 이름을 반드시 설정해야 한다. 대화가 연결되어 있는 동안 서버

는 끊임없이 클라이언트의 데이터를 수정하며 실제로 전달된 데이터는 Text 속성으로 읽을 수 있다. Text 속성이 변할 때마다 OnChange 이벤트가 발생하므로 이 이벤트에서 변경된 데이터를 에디트나 메모로 보내는 코드를 실행하도록 해야 한다.

## ◘ 속성

### DDEConv:TDdeClientConv;
DDE 대화에 사용되는 TDDEClientConv 컴포넌트의 이름을 기입해준다. 이 속성을 정의해 주어야 서버로부터 전달된 데이터를 전달받을 수 있다.

### DDEItem:string;
DDE 대화의 항목을 지정하며 이 값은 연결된 서버 프로그램에 따라 달라진다. 서버 프로그램이 델파이로 만든 프로그램일 경우 DDEServerItem 컴포넌트의 Name 속성을 사용하면 된다.

### Text:string;
서버로부터 전달된 문자열이 나타난다. 255문자까지 기억한다.

### Lines:TStrings;
서버로부터 전달된 문자열이 나타난다. Text와는 달리 여러 줄의 문자열을 가질 수 있다.

# 55. DdeServerConv

## ◘ 정의
DDE 클라이언트와 함께 DDE 대화를 구성하며 TDDEServerItem 컴포넌트와 연결되어 있어야 한다. 클라이언트가 서버로 매크로를 전달하면 OnExecuteMacro 이벤트가 발생하며 이 이벤트에 매크로를 실행시키는 코드를 작성해야 한다. TDDEServerConv 컴포넌트는 꼭 필요한 것은 아니며 이 컴포넌트가 없어도 클라이언트는 TDDEServeItem 컴포넌트로 곧바로 데이터를 요청할 수 있다. 단 이 컴포넌트를 쓰지 않으면 DDE 토픽명이 폼의 캡션이어야 한다. 만약 폼의 캡션이 가변적이거나 너부 복잡(예를 들어 한글이나 한자)할 경우는 이 컴포넌트를 사용해야 한다.

# 56. DdeServerItem

## ◘ 정의
DDE 대화의 아이템을 지정한다. 홀로 사용될 수도 있으며 TDDEServeConv 컴포넌트와 함께 사용될 수도 있다. 두 컴포넌트가 함께 사용될 경우 ServeConv 속성에 TDDEServeConv 컴포넌트의 이름을 지정하여 연결해 주어야 한다. Text 속성에는 클라이언트로 보낼 문자열이 기입된다.

## ◘ 속성

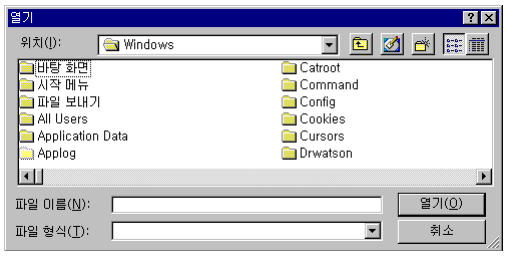
### ServeConv:TDdeServerConv;
이 컴포넌트와 연결된 TDDEServeConv 컴포넌트의 이름을 지정한다.

### Text:string;
클라이언트로 전달될 문자열이며 255문자까지 기억한다.

### Lines:TStrings;
클라이언트로 전달될 문자열이며 여러 줄의 문자열을 기억한다.

# 57. OpenDialog

## ◘ 정의
파일 열기 대화상자를 제공한다. 사용자가 프로그램 실행중에 파일 이름을 마우스로 선택할 수 있도록 대화상자를 열어주고 입력받은 파일을 프로그램으로 전달해 준다.




이 컴포넌트를 폼에 배치하면 아이콘만 폼에 나타나며 실행중에 이 아이콘은 보이지 않는다. 대화상자를 실행하려면 Execute 메소드를 호출하며 사용자가 파일을 선택한 후 OK버튼을 누르면 FileName 속성에 선택한 파일 이름을 대입해준다.

Execute 메소드는 대화상자가 제대로 출력되고 사용자에 의해 파일이 선택되었으면 True를 리턴하며 에러가 발생했거나 사용자가 Cancel 버튼을 눌러 선택을 취소했으면 False를 리턴한다. 이 대화상자를 이용하고자 할 때는 반드시 Execute 메소드의 리턴값을 점검해 본 후 사용해야 한다. 이 컴포넌트를 사용하는 일반적인 코드는 다음과 같다.

```
if OpenDialog1.Execute then
  Edit1.Text:=OpenDialog1.FileName;
```

대화상자를 실행한 후 에러가 없으면 선택된 파일 이름을 에디트 박스로 출력한다. 물론 선택된 파일을 실행시키거나 삭제하는 등의 동작을 하는 것도 가능하다. 대화상자의 동작이나 모양을 지정하는 여러 가지 속성이 있다. 메소드는 Execute 하나만 있으며 이벤트는 가지지 않는다.

## ◘ 속성

### FileName:TFileName;

대화상자가 열릴 때 대화상자의 에디트 박스에 출력될 파일 이름을 지정한다. 또한 대화상자가 닫힐 때 사용자에 의해 선택된 파일 이름이 저장되기도 하므로 이 속성이 곧 파일 열기 대화상자의 리턴값이 된다. 선택한 파일 이름은 곧바로 파일 실행, 삭제, 읽기 등의 함수에 사용될 수 있다.

### DefaultExt:TFileExt;

사용자가 직접 에디트 박스에 파일을 입력할 경우 확장자를 생략하면 이 속성이 지정하는 확장자가 자동으로 파일 이름에 추가된다. 물론 확장자를 명시했을 경우는 이 속성이 무시된다. 최초 세 문자까지만 확장자로 사용되며 파일 이름과 확장자를 구분하는 '.'은 포함시키지 않아도 된다.

### FileEditStyle:TFileEditStyle;

대화상자에 사용자가 직접 파일을 입력하는 에디트 박스의 형태를 지정한다. 이 속성이 FsEdit이면 에디트 박스를 사용하며 fsComboBox이면 드롭다운 콤보 박스를 사용한다. 디폴트는 fsEdit이다. 콤보 박스를 사용할 경우 사용자는 실행중에 콤보 박스의 List 속성에 선택 가능한 파일 목록을 입력해 둔 후 대화상자를 불러야 한다. 콤보 박스에 입력될 파일 목록은 HistoryList 속성에 입력해 준다.

### HistoryList:TStrings ;

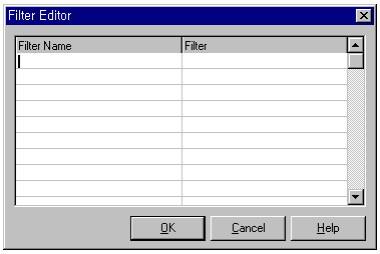
대화상자에 콤보 박스를 배치했을 경우 콤보 박스에 입력할 파일 이름 목록을 지정해 준다.

### Files:TStrings;

Options 속성의 ofAllowMultiSelect 속성이 True로 설정되어 여러 개의 파일을 선택할 수 있도록 했을 때 선택된 파일들의 목록을 담는다. 여러 개의 파일 이름을 저장하는 문자열 리스트형이다.

### Filter:string;

필터를 설정하며 "설명|필터" 형식으로 필터를 지정한다. 설명은 어떤 종류의 파일인가를 알려주는 문자열이며 필터는 *.exe와 같은 와일드 카드 식으로 표현되며 가운데 파이프 문자(|)가 삽입된다. '워드 파일|*.doc','엑셀 파일|*.xls' 등이 하나의 필터가 된다. 여러 개의 필터를 계속 파이프로 연결하여 작성할 수 있으며 필터 사이사이에 파이프 문자로 구분해준다. '워드 파일|*.doc|엑셀 파일|*.xls|그림 파일|*.gif' 는 세 개의 필터를 정의한 예이다. 하나의 필터에 두 개 이상의 와일드 카드식을 쓰고 싶으면 '실행 파일|*.exe;*.com;*.bat' 등과 같이 세미콜론으로 구분하여 와일드 카드 식을 나열한다. 이 속성을 더블클릭하여 필터 편집기를 불러낸 후 필터를 편집할 수도 있다.




### FilterIndex:Integer;

Filter 속성에 설정된 필터 중에 어떤 필터를 사용할 것인가를 지정한다. 디폴트값은 1이며 첫 번째 필터가 사용된다.

### InitialDir:string;

대화상자가 열릴 때 목록을 보여줄 시작 디렉토리를 지정한다. 이 속성을 설정하지 않으면 현재 디렉토리의 파일 목록이 출력된다.

### Options:TOpenOptions;

대화상자의 모양과 기능을 정의하는 여러 가지 세부 속성을 가진다.

| 속성 | 의미 |
|---|---|
| ofAllowMultiSelect | 파일 리스트 박스에서 여러 개의 파일을 선택할 수 있도록 해준다. |

| | |
|---|---|
| ofCreatePrompt | 이 옵션이 True이며 사용자가 존재하지 않는 파일 이름을 직접 에디트 박스에 입력했을 경우 경고를 출력한다. 현재 존재하지 않는 파일을 만들 것인가 질문한다. |
| ofExtensionDifferent | 파일 열기 대화상자에서 리턴한 파일의 확장자가 DefaultExt 속성과 다른 확장자를 가지고 있을 경우 이 속성이 True로 설정된다. 오브젝트 인스펙터에 이 속성을 직접 설정하는 것은 의미가 없다. |
| ofFileMustExist | 존재하지 않는 파일 이름을 에디트 박스에서 직접 입력했을 경우 경로와 파일 이름이 맞게 입력되었는지 질문한다. |
| ofHideReadOnly | 읽기 전용으로 파일을 열 수 있는 ReadOnly 체크 박스를 보여주지 않는다. |
| ofNoChangeDir | 디렉토리를 변경할 수 없도록 한다. |
| ofNoReadOnlyReturn | 사용자가 읽기 전용의 파일을 선택했을 경우 이 사실을 별도의 대화상자를 열어 알려 준다. |
| ofNoTestFileCreate | 쓰기 금지, 디스크 용량 부족, 드라이브의 준비 상황 등을 체크하지 않도록 한다. |
| ofNoValidate | 파일 에디트 박스에 파일 이름으로 사용할 수 없는 문자를 입력할 경우 에러 메시지를 출력하지만 이 속성이 True일 경우는 에러 메시지를 출력하지 않는다. |
| ofOverwritePrompt | 이미 존재하고 있는 파일을 다시 저장하고자 할 경우 경고 메시지를 출력한다. |
| ofReadOnly | 읽기 전용 체크 박스를 체크한 상태로 대화상자를 연다. |
| ofPathMustExist | 존재하는 경로 이름만 입력할 수 있도록 한다. 디스크 상에 없는 디렉토리명을 입력할 경우는 에러 메시지를 출력한다. |
| ofShareAware | 파일 공유 에러가 발생할 경우 모든 에러를 무시하고 파일 이름을 리턴한다. |
| ofShowHelp | 대화상자에 도움말 버튼을 출력한다. |

Title:string;
대화상자의 타이틀 바에 나타날 문자열을 지정한다.

## ◘ 메소드

■ Execute
파일 열기 대화상자를 모달로 실행시킨다. 사용자가 파일을 선택하거나 OK 버튼을 누르면 True 를 리턴하며 파일 선택을 취소할 경우 False 를 리턴한다. 보통 다음과 같은 형식으로 사용된다.

```
if OpenDialog1.Execute then
  파일 읽기;
```

# 58. SaveDialog

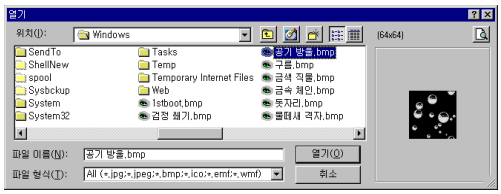
## ◘ 정의

파일 열기 대화상자와 완전히 동일한 기능을 수행한다. 다만 파일 열기 대화상자는 파일을 읽어들일 때 사용하며 파일 저장 대화상자는 파일을 기록할 때 사용한다는 점이 다르다. 두 대화상자의 공통된 목적은 파일의 이름을 실행중에 마우스로 입력받는 것이다. 사용하는 속성, 메소드도 동일하다.

# 59.OpenPictureDialog

## ◘ 정의

OpenDialog 와 모든 면에 있어서 동일하나 대화상자의 오른쪽에서 그림 미리보기를 제공한다는 차이점만 있다.




# 60.SavePictureDialog

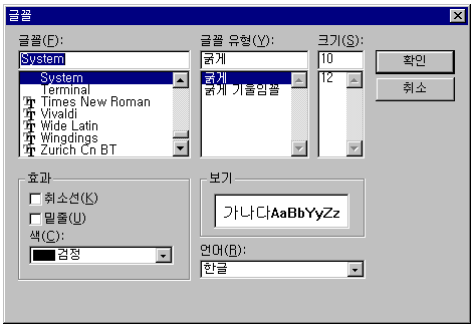
## ◘ 정의

그림 미리 보기를 제공하는 것 외에 SaveDialog 와 모든 면에 있어서 동일하다.

# 61. FontDialog

## ◘ 정의

폰트를 선택할 수 있는 대화상자를 보여주며 폰트의 여러 가지 속성을 선택하도록 해준다.




이 컴포넌트를 폼에 배치하면 아이콘만 폼에 나타나며 실행중에 이 아이콘은 보이지 않는다. 대화상자를 실행하려면 Execute 메소드를 호출하며 사용자가 대화상자에서 선택한 속성은 Font 속성에 대입된다. Execute 메소드는 대화상자가 제대로 출력되고 사용자에 의해 폰트가 선택되었으면 True를 리턴하며 에러가 발생했거나 사용자가 Cancel 버튼을 눌러 선택을 취소했으면 False를 리턴한다. 이 대화상자를 이용하고자 할 때는 반드시 Execute 메소드의 리턴값을 점검해 본 후 사용해야 한다. 이 컴포넌트를 사용하는 일반적인 코드는 다음과 같다.

```
if FontDialog1.Execute then
  Form1.Font:=FontDialog1.Font;
```

대화상자의 동작이나 모양을 지정하는 여러 가지 속성이 있는데 메소드는 Execute 메소드 하나만 있으며 이벤트는 가지지 않는다.

## ◘ 속성

### Device :TFontDialogDevice;

선택된 폰트가 영향을 미칠 장치를 선택한다. fdScreen은 화면에 영향을 주며 fdPrinter는 프린터에 영향을 준다.

### Font:TFont;

폰트 대화상자에서 사용자가 선택한 폰트에 관한 특성을 가진다. 대화상자 호출 후에 이 속성을 읽어 폰트를 선택하므로 이 속성이 폰트 선택 대화상자의 리턴값이 된다.

### MinFontSize, MaxFontSize :Integer;

선택할 폰트의 최대 크기와 최소 크기를 지정한다. 두 속성의 디폴트값은 모두 0이며 이는 최대, 최소 크기의 제한이 없다는 뜻이다.

### Options :TFontDialogOptions;

폰트 선택 대화상자의 동작과 모양을 정의하는 여러 가지 세부 속성을 가진다.

| 속성 | 의미 |
|---|---|
| fdAnsiOnly | 윈도우즈의 문자셋을 가진 폰트만 보여준다. 즉 심벌이나 그림 따위만 가진 폰트는 보여주지 않는다. |
| fdEffects | 효과를 설정할 수 있는 체크 박스와 색상을 변경할 수 있는 리스트 박스를 대화상자에 출력한다. |
| fdFixedPitchOnly | 고정 폭 폰트만 선택할 수 있도록 한다. |
| fdForceFontExist | 설치되어 있지 않는 폰트는 선택할 수 없도록 한다. |
| fdLimitSize | 선택할 수 있는 폰트의 크기를 제한한다. |
| fdNoFaceSel | 대화상자가 처음 나타날 때 아무 폰트도 선택되어 있지 않도록 한다. |
| fdNoOEMFonts | 벡터 폰트가 아닌 폰트만 폰트 콤보 박스에 나타나도록 한다. |
| fdScalableOnly | 크기를 변경할 수 있는 폰트만 폰트 콤보 박스에 나타나도록 한다. |
| fdNoSimulations | GDI 폰트 시뮬레이션을 하지 않는 폰트만 폰트 콤보 박스에 나타나도록 한다. |
| fdNoSizeSel | 대화상자가 처음 나타날 때 폰트 크기가 선택되지 않은 상태로 나타난다. |
| fdNoStyleSel | 대화상자가 나타날 때 Style이 선택되지 않은 상태로 나타난다. |
| fdNoVectorFonts | fdNoOEMFonts 속성과 동일하다. |
| fdShowHelp | 도움말 버튼을 보여준다. |
| fdTrueTypeOnly | 트루타입 폰트만 폰트 콤보 박스에 나타난다. |
| fdWysiwyg | 프린터와 화면에 모두 사용할 수 있는 폰트만 나타난다. |

# 62. ColorDialog

## ◘ 정의

색상 선택 대화상자를 제공한다. 사용자가 프로그램 실행중에 색상을 마우스로 선택할 수 있도록 대화상자를 열어 주고 사용자가 선택한 색상을 프로그램으로 전달해준다.


이 컴포넌트를 폼에 배치하면 아이콘만 폼에 나타나며 실행중에 이 아이콘은 보이지 않는다. 대화상자를 실행하려면 Execute 메소드를 호출하며 사용자가 색상을 선택한 후 OK버튼을 누르면 Color 속성에 선택한 색상을 대입해준다. Execute 메소드는 대화상자가 제대로 출력되고 사용자에 의해 파일이 선택되었으면 True를 리턴하며 에러가 발생했거나 사용자가 Cancel 버튼을 눌러 선택을 취소했으면 False를 리턴한다. 이 대화상자를 이용하고자 할 때는 반드시 Execute 메소드의 리턴값을 점검해 본 후 사용해야 한다. 이 컴포넌트를 사용하는 일반적인 코드는 다음과 같다.

```
if ColorDialog1.Execute then
  Form1.Color:=ColorDialog1.Color;
```

대화상자를 실행한 후 에러가 없으면 선택한 색상으로 폼의 색상을 변경한다. 대화상자의 동작이나 모양을 지정하는 여러 가지 속성이 있는데 메소드는 Execute 하나만 있으며 이벤트는 가지지 않는다.

## ◘ 속성

### Color:TColor;

색상 선택 대화상자에서 사용자가 선택한 색상을 가진다. 이 속성을 읽어 사용자가 선택한 색상을 알 수 있으므로 이 속성이 곧 색상 선택 대화상자의 리턴값이다.

### CustomColors:TStrings;

대화상자에서 사용자에게 보여줄 사용자 정의형 색상을 정의한다. 사용자 정의형 색상은 6자리의 16진수로 구성된 문자열 형태로 넘겨지며 이 16진수가 색상값을 나타낸다. 각 색상값은 ColorA=00FF80 등과 같은 형식으로 이루어져 있어야 한다. ColorA ~ ColorP까지 16개의 사용자 정의형 색상을 정의할 수 있다.

### Options:TColorDialogOptions;

다음과 같은 세부 속성을 가진다. 디폴트로 이 속성들은 모두 선택되어 있지 않다.

| 속성 | 의미 |
|---|---|
| cdFullOpen | 사용자 정의형 색상을 선택할 수 있도록 대화상자를 연다. |
| cdPreventFullOpen | 사용자 정의형 색상을 선택하지 못하도록 한다. |
| cdShowHelp | 도움말 버튼을 대화상자에 표시한다. |

# 63. PrintDialog

## ◘ 정의

인쇄 대화상자를 보여준다. 이 컴포넌트를 폼에 배치하면 아이콘만 폼에 나타나며 실행중에 아이콘은 보이지 않는다. 인쇄 대화상자는 다음과 같다.


물론 이 대화상자는 현재 시스템에 설치되어 있는 프린터에 따라 달라진다. 대화상자를 실행하려면 Execute 메소드를 호출한다. Execute 메소드는 대화상자가 제대로 출력되고 OK 버튼이 선택되었으면 True를 리턴하며 에러가 발생했거나 사용자가 Cancel 버튼을 눌러 선택을 취소했으면 False를 리턴한다. 이 대화상자를 이용하고자 할 때는 반드시 Execute 메소드의 리턴값을 점검해 본 후 사용해야 한다. 이 컴포넌트를 사용하는 일반적인 코드는 다음과 같다.

```
if PrintDialog1.Execute then
  프린트에 관계된 코드
```

대화상자에서 사용자가 선택한 각종 옵션은 대화상자의 속성을 통해 프로그램으로 전달된다. 대화상자의 모양이나 동작을 정의하는 여러 가지 속성이 있으며 메소드는 Execute 하나만 가지며 이벤트는 가지지 않는다.

## ◘ 속성

### Collate:Boolean;
사용자가 Collate 체크 박스를 선택했는지를 리턴한다.

### Copies:Integer;
여러장 인쇄를 선택했을 경우 인쇄 매수를 리턴한다. 디폴트값은 0이다.

### MaxPage, MinPage:Integer;
인쇄할 문서의 페이지 범위를 설정한다. 사용자가 이 범위 밖의 페이지를 지정하면 경고 대화상자가 출력된다.

### FromPage, ToPage:Integer;
인쇄할 페이지의 범위를 리턴한다.

### PrintRage:TPrintRange;
인쇄 범위를 리턴한다. 다음 세 가지 값 중 하나가 된다.

| 인쇄 범위 | 의미 |
|---|---|
| prAllpages | 전체 페이지 인쇄 |
| prSelection | 선택된 문장만 인쇄 |
| prPageNums | 일정 범위만 인쇄 |

### PrintToFile :Boolean;
사용자가 대화상자에서 파일로 인쇄 옵션을 선택했는가를 리턴한다.

### Options:TPrintDialogOptions;
인쇄 대화상자의 동작을 정의하는 세부 속성을 가진다.

| 속성 | 의미 |
|---|---|
| poHelp | 도움말 버튼을 보여준다. |
| poPageNums | 페이지 선택 라디오 버튼을 사용 가능하게 하여 인쇄 범위를 선택할 수 있도록 한다. |
| poPrintToFile | 파일로 출력할 수 있도록 체크 박스를 보여준다. |
| poSelection | 선택된 영역만 인쇄할 수 있도록 라디오 버튼을 사용가능하게 한다. |
| poWarning | 프린터가 설치되어 있지 않을 경우 경고 메시지를 출력한다. |
| poDisablePrintToFile | 파일로 인쇄 체크 박스를 보여준다. |

# 64. PrintSetupDialog

## ◘ 정의
프린터 설정 대화상자를 제공한다. 이 컴포넌트를 폼에 배치하면 아이콘만 폼에 나타나며 실행중에 아이콘은 보이지 않는다. 대화상자를 실행하려면 Execute 메소드를 호출한다. 이 컴포넌트를 사용하여 프린터 설정 대화상자를 직접 호출할 수도 있지만 인쇄 대화상자의 설정 버튼에 의해 호출되기도 한다. 프린터 설정 대화상자에서 사용자가 변경한 사항은 프로그램으로 전달되지 않고 프린터 드라이버를 직접 변경시키므로 프로그램에서 별도로 해 주어야 할 일은 없다. 프린터 설정 대화상자는 다음과 같다.


물론 이 대화상자는 현재 윈도우즈에 설치된 프린터의 종류에 따라 달라진다.

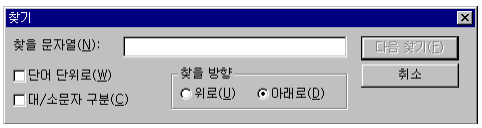
# 65. FindDialog

## ◘ 정의
문자열 찾기 대화상자를 제공한다. 이 컴포넌트를 배치하면 폼에 아이콘만 나타나며 실행중에는 보이지 않는다. 대화상자를 실행하기 위해서는 Execute 메소드를 호출한다. Execute 메소드는 대화상자가 제대로 출력되고 사용자에 의해 찾을 문자열이 입력되었으면 True를 리턴하며 에러가 발생했거나 사용자가 Cancel 버튼을 눌러 찾기를 취소했으면 False를 리턴한다. 이 대화상자를 이용하고자 할 때는 반드시 Execute 메소드의 리턴값을 점검해 본 후 사용해야 한다. 이 컴포넌트를 사용하는 일반적인 코드는 다음과 같다.

```
if FindDialog1.Execute then
  찾기 코드
```

문자열 찾기 대화상자가 실행되면 사용자는 에디트 박스에 검색하고자 하는 문자열을 입력하고 다음 찾기 버튼을 눌러 검색 시작을 지시한다. 그러나 이 대화상자는 검색 문자열을 입력받기만 할 뿐 실제로 검색을 수행하는 것은 아니다. 사용자가 다음

찾기 버튼을 누르면 OnFind 이벤트를 발생시키며 프로그래머는 OnFind 이벤트 핸드러에서 검색 코드를 작성해야 한다. 찾기 대화상자의 모습은 다음과 같다.




## ◘ 속성

### FindText:string;

검색하고자 하는 문자열이다. 대화상자 호출 전에 문자열을 미리 입력시켜 두면 대화상자의 에디트 박스에 이 문자열이 나타난다.

### Position:TPoint;

대화상자가 출력될 위치를 화면 좌표로 지정한다. 검색한 문자열의 위치를 보여주기 위해 대화상자의 위치를 잘 조정해 주어야 한다.

### Options:TFindOptions;

찾기 대화상자의 동작과 모양을 지정하는 여러 가지 세부 속성들이다.

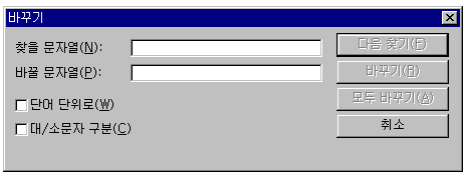
| 속성 | 의미 |
|---|---|
| frDisableMatchCase | 대/소문자 구분 체크 박스를 사용할 수 없도록 한다. |
| frDisableUpDown | 위로/아래로 라디오 버튼을 사용할 수 없도록 한다. |
| frDisableWholeWord | 단어 단위로 체크 박스를 사용할 수 없도록 한다. |
| frDown | 아래로 찾기 라디오 버튼을 체크하여 검색을 아래로 수행하도록 한다. 이 속성이 False이면 검색은 위쪽으로 수행된다. 물론 사용자는 이 속성에 상관없이 실행중에 옵션을 변경할 수 있다. |
| frFindNext | 다음 찾기 버튼을 누르면 이 속성이 True로 설정된다. |
| frHideMatchCase | 대소문자 구분 체크 박스를 숨긴다. |
| frHideWholeWord | 단어 단위로 체크 박스를 숨긴다. |
| frHideUpDown | 위로/아래로 라디오 버튼을 숨긴다. |
| frMatchCase | 대/소문자 구분 체크 박스가 선택된다. |
| frReplace | 바꾸기 버튼을 누를 경우 이 속성이 True로 설정된다. |
| frReplaceAll | 모두 바꾸기 버튼을 누를 경우 이 속성이 True로 설정된다. |
| frShowHelp | 대화상자에 도움말 버튼을 보여준다. |
| frWholeWord | 단어 단위로 체크 박스가 선택된다. |

# 66. ReplaceDialog

## ◘ 정의

문자열 바꾸기 대화상자를 제공한다. 문자열 바꾸기 대화상자는 문자열 찾기 대화상자의 모든 기능을 가지며 문자열을 바꿀 수 있는 기능까지도 제공한다. 이 컴포넌트를 폼에 배치하면 아이콘만 나타나며 실행중에는 보이지 않는다. 대화상자를 실행하려면 Execute 메소드를 사용한다. 찾고자 하는 문자열은 FindText 속성으로 지정하며 바꾸고자 하는 문자열은 ReplaceText 속성으로 지정한다.

문자열 바꾸기 대화상자가 실행되면 사용자는 두 개의 에디트 박스에 찾고자 하는 문자열과 바꾸고자 하는 문자열을 기입한 후 다음 찾기 또는 바꾸기 버튼을 선택하여 찾기, 바꾸기 작업을 지시한다. 그러나 이 대화상자는 찾을 문자열과 바꿀 문자열을 입력받기만 할 뿐 실제로 찾기 바꾸기를 수행하는 것은 아니다. 사용자가 다음 찾기 버튼을 누르면 OnFind 이벤트가 발생하며 바꾸기 버튼을 누르면 OnReplace 이벤트가 발생한다. 프로그래머는 이러한 이벤트 핸들러에 찾기, 바꾸기 코드를 작성해야 한다. 문자열 바꾸기 대화상자의 모습은 다음과 같다.




## ◘ 속성

### FindText:string;

검색하고자 하는 문자열이다. 대화상자 호출 전에 문자열을 미리 입력시켜 두면 대화상자의 에디트 박스에 이 문자열이 나타난다.

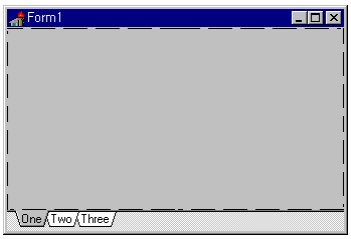
### ReplaceText:string;

바꾸고자 하는 문자열이다.

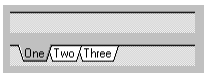
### Options:
문자열 바꾸기 대화상자의 모양과 동작을 지정하는 여러 가지 세부 속성을 가지며 찾기 대화상자와 동일하다.

# 67. TabSet

## ◘ 정의
수평으로 늘어서 있는 탭을 표현하며 주로 NoteBook 컴포넌트와 함께 사용하여 NoteBook의 페이지를 바꾸어 주는 역할을 한다. 다음 그림은 윗쪽에 노트북을 아래쪽에는 탭셋을 배치한 모양이다.




이렇게 배치하려면 노트북은 alClient로 정렬하고 탭셋은 alBottom으로 정렬해야 한다.
Tabs 속성에 탭셋에 나타날 탭 제목의 문자열들을 입력해 주면 입력한 제목의 수만큼 탭이 나타난다. 최초 탭셋을 폼에 배치하면 Tabs 속성에 아무 문자열도 정의되어 있지 않으므로 비어있는 탭셋 모양을 가지지만 One, Two, Three 세 개의 탭 제목을 입력했을 때의 탭셋의 모양은 다음과 같다.




그러나 탭셋의 Tabs 속성은 디자인시에 직접 입력하지 않으며 실행시에 다음 코드로 노트북의 페이지와 동일한 탭 제목을 대입받는 방법이 더 효율적이다.

```
TabSet1.Tabs:=NoteBook1.Page;
```

탭셋이 단독으로 사용되는 경우보다는 노트북과 함께 사용하는 경우가 더 많고 노트북의 페이지를 표현하는 목적으로 사용되므로 노트북의 페이지 이름이 그대로 탭셋의 제목이 된다. 탭셋의 OnClick 이벤트에서 다음과 같은 코드를 작성하여 탭셋의 내용이 바뀌면 노트북도 대응되는 페이지로 바뀔 수 있도록 해주어야 한다.

```
procedure TForm1.TabSet1Click(Sender: TObject);
begin
  Notebook1.PageIndex := TabSet1.TabIndex;
end;
```

탭셋에서 선택된 탭의 번호를 노트북의 페이지 번호에 대입해 준다.

## ◘ 속성
주로 탭셋의 외부적인 모양과 색상을 지정하는 속성들이다.

### SelectColor:TColor
선택된 탭의 색상을 지정한다. 되도록 눈에 띄는 분명한 색상을 사용하는 것이 좋다. 디폴트는 clBtnFace, 즉 버튼의 색상과 같은 색으로 되어 있다.

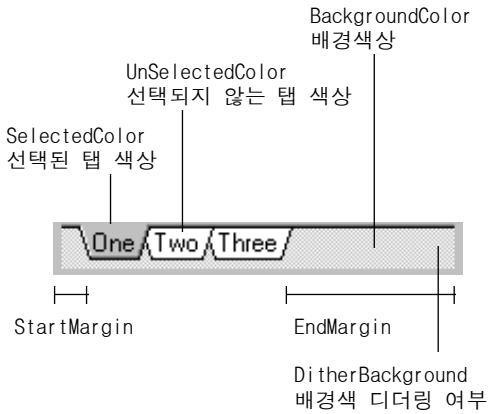
### UnSelectedColor:TColor
선택된 탭 외의 선택되지 않은 나머지 탭의 색상을 설정한다. 선택되지 않은 탭의 색상을 지정하는 것이므로 되도록 흐리고 눈에 띄지 않은 색상을 사용하는 것이 좋다. 디폴트는 윈도우의 배경색, 즉 흰색으로 되어 있다.

### BackGroundColor:TColor
탭 뒤쪽의 배경 색상을 지정한다. 탭 자체의 색상보다는 밝은 색을 사용하여 탭이 분명히 보이도록 해 주는 것이 좋으며 폼의 배경 색과도 조화를 이루게 하는 것이 좋다.

### DitherBackground:Boolean
배경 색상을 디더링할 것인가를 지정한다. 디더링을 한다는 의미는 배경 색상에 흰색을 반쯤 섞어 50% 정도 배경을 밝게 작성하도록 한다는 뜻이며 배경 색상이 밝으면 탭 이름이 좀 더 분명하게 보인다. 노란색이나 연두색 등의 밝은 색상을 사용한다면 이 속성을 False로 설정해도 상관없다.

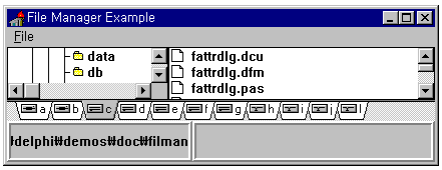
### StartMargin:Integer
탭셋과 첫 번째 탭과의 간격을 픽셀 단위로 지정하며 디폴트값은 5이다.

### EndMargin:Integer
탭셋과 마지막 탭과의 간격을 픽셀 단위로 지정하며 디폴트값은 5이다. StartMargin 속성과 함께 탭셋의 전체적인 모양을 결정하며 탭셋에 몇 개의 탭을 배치할 것인가를 결정하는 중요한 요소가 된다. 주의할 것은 이 속성은 최소값을 지정하는 것이지 강제적인 간격을 의미하는 것은 아니라는 점이다. 즉 탭 우측으로 100픽셀이 남은 상태에서 이 속성을 50으로 한다고 해서 탭을 강제로 우측으로 옮기지는 않는다. 다만 우측에 최소한 50픽셀의 공간을 둔다는 뜻이다.



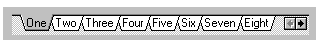

### Style:TTabStyle

탭셋의 모양을 지정한다. tsStandard이면 각 탭에 보통 크기의 문자열로서 제목을 나타내지만 tsOwnerDraw이면 각 탭은 TabHeight 속성에서 지정한만큼의 높이를 가지며 문자열 외에도 그래픽 이미지를 탭에 표시할 수 있다. 다음 그림은 델파이와 함께 제공되는 파일 관리자 예제에서 사용된 탭셋이다. 드라이브의 유형에 따라 비트맵을 탭셋으로 보여준다.




### 실 읽 VisibleTabs:Integer

탭셋에 보이는 탭의 숫자를 가진다. 이 숫자는 Tabs에 입력된 탭의 총 개수와는 다를 수도 있다 왜냐하면 탭셋의 폭이 탭 전체를 다 보여줄 만한 크기가 되지 않을 수도 있기 때문이다. 다음 그림은 10개의 탭 중 8개만 보이는 경우이며 우측에 나머지 두 개의 탭을 볼 수 있는 스크롤 버튼이 나타난다.




이 경우 전체 탭은 10개이지만 보이는 탭은 8개이다.

# 68. Outline

## ◘ 정의

다중 레벨의 데이터 구조를 Tree 형식으로 나타내 준다. 아웃라인의 각 항목은 TOutLineNode 오브젝트이며 아웃라인의 Items 속성으로 참조한다. Items 속성은 TOutLineNode형 오브젝트의 1차원 배열이므로 첨자에 의해 각 노드를 구분한다. 아웃라인의 디폴트 속성이 Items이므로 참조시 Items 속성을 직접 사용하지 않고도 사용할 수 있다. 예를 들어 OutLine1.Items[1]은 OutLine[1]과 같다. 아웃라인에 항목을 관리할 때는 메소드를 사용한다. 항목을 추가할 때는 Add나 AddString, 삽입할 때는 Insert 또는 InsertString, 삭제할 때는 Delete를 사용하며 하위 항목을 추가할 때는 AddChild나 AddChildObject 메소드를 사용한다. 아웃라인에서 현재 선택된 항목은 SelectedItems 속성으로 읽을 수 있다.

## ◘ 속성

### 실 읽 Itemcount:Longint;

실행중에만 사용할 수 있는 읽기 전용의 속성이다. 아웃라인에 포함된 총 항목의 개수를 조사해 준다.

### ItemsSeparator:string;

Text형의 항목들을 구분해 주는 구분 문자를 지정하며 디폴트 값은 '₩'이다.

# 69. TabbedNotebook

## ◘ 정의

하나의 윈도우가 표시할 수 있는 정보의 양에는 한계가 있다. 아무리 컨트롤을 밀도있게 배치하고 콤보 박스, 드롭다운 리스트 등의 절약형 컨트롤을 사용해도 윈도우가 비좁기는 마찬가지이다. 그래서 여러 개의 윈도우를 열거나 하나의 윈도우 안에 새끼 윈도우를 쓰는 MDI가 필요하다. 그 외의 방법으로는 페이지를 겹쳐 표현하는 탭북 형태가 많이 사용된다.

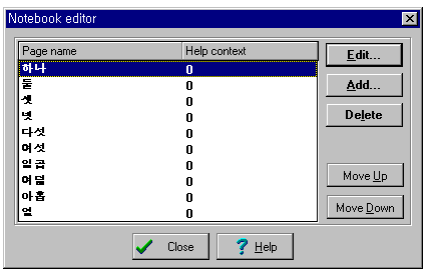

탭북은 여러 개의 페이지를 같은 공간에 배치하고 탭을 클릭하여 해당 페이지를 펼치는 컨트롤이다. TabbedNoteBook을 폼에 배치하면 우선 하나의 탭만을 가지는 모양으로 나타난다. Pages 속성에 탭에 나타날 제목을 문자열 리스트로 입력해 주면 제목의 수만큼 페이지가 생성되며 이 제목이 탭의 상단에 나타난다.

생성된 페이지를 펼치려면 실행시와 디자인시에 ActivePage나 PageIndex 속성으로 펼칠 페이지를 지정해 준다. ActivePage는 페이지의 제목 문자열로 지정하며 PageIndex는 첫 페이지를 0번으로 하는 페이지 번호로 지정한다.

## ◘ 속성

### Pages:TStrings;

탭북의 각 페이지 제목을 가지며 이 속성에 입력된 제목의 수만큼 페이지가 생성된다. 예를 들어 세 개의 각각 다른 페이지를 만들고 싶다면 세 페이지의 이름을 정한 후 이 속성에 세 개의 이름 문자열을 대입해 주어야 한다. 오브젝트 인스펙터에서 이 속성을 더블클릭하면 페이지 이름을 입력할 수 있는 문자열 편집기가 열린다.




이 대화상자에서 탭 제목을 편집, 추가, 삭제하며 아래쪽의 MoveUp, MoveDown 버튼을 사용하여 탭 제목의 순서를 변경한다. 한글도 입력 가능하다.

### ActivePage:String;

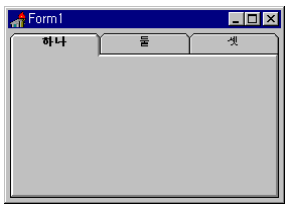
출력할 페이지의 이름을 지정한다. Pages 속성에 등록되어 있는 이름 중에 하나를 사용해야 한다. 문자열이므로 단순히 대입해도 되며 디자인중에는 이 속성을 더블클릭하여 다음 페이지로 전환한다.

### PageIndex:Integer;

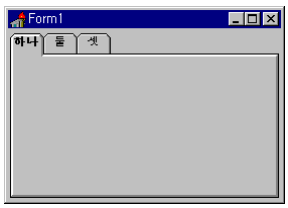
출력할 페이지의 번호를 지정한다. Pages 속성에 등록한 탭의 이름은 등록된 순서대로 0, 1, 2, ...등의 인덱스 번호를 부여받으며 이 번호로 활동 페이지를 지정한다. ActivePage와 똑같은 목적에 사용되는 속성이지만 탭에 이름이 주어지지 않은 경우에는 ActivePage 속성으로 활동 페이지를 지정할 수 없기 때문에 이 속성을 사용해야 한다.

### TabsPerRow:Integer;

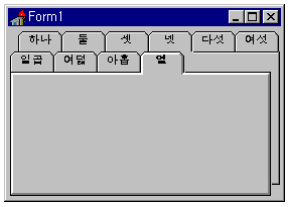
한 줄에 나타날 탭의 수를 설정한다. 이 속성에 실제의 탭수보다 더 많은 값을 주면 탭의 크기가 적당히 줄어들며 좌측으로 정렬되어 보기가 깔끔해진다. 다음 그림은 탭의 수와 이 속성값이 모두 3으로 설정되어 있는 경우이며 탭이 탭북의 상단을 모두 차지하고 있다.



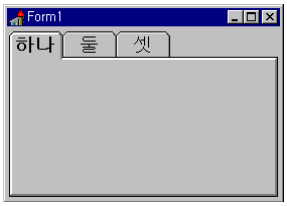

다음 그림은 탭이 세 개이되 TabsPerRow를 8로 설정하여 8개의 탭이 들어갈만한 공간에 세 개의 탭만 배치한 것이다. 탭의 크기가 탭북의 1/8로 줄어든다.




다음 그림은 탭의 개수보다 TabsPerRow 속성을 더 작게 준 경우이다. 탭은 전부 열 개이지만 이 속성에 6을 대입함으로써 탭을 2단으로 배치한다.

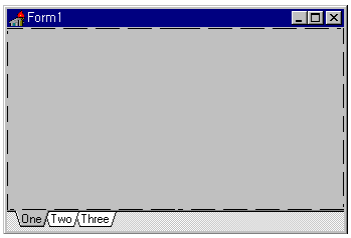


### TabFont:TFont
탭 제목의 폰트를 지정한다. 탭의 제목은 가급적이면 눈에 잘 띄게 크고 깔끔하게 만들어 주는 것이 좋다.




## 70. NoteBook

### ◘ 정의
노트북은 여러 개의 페이지를 가지며 각 페이지는 고유의 컨트롤들을 가진다. 페이지가 같은 공간에 겹쳐져 있기 때문에 좁은 공간에 더 많은 정보를 표현할 수 있다는 장점이 있다. 각 페이지의 제목은 Pages 속성으로 입력해 주며 PageIndex 속성이나 ActivePage 속성으로 활동 페이지를 지정한다. 노트북은 페이지의 제목이 화면으로 출력되지 않으며 사용자가 실행중에 마우스로 직접 페이지를 선택할 수 있는 방법이 제공되지 않기 때문에 보통 탭셋과 함께 사용된다. 다음 그림은 윗쪽에 노트북을 아래쪽에는 탭셋을 배치한 모양이다.



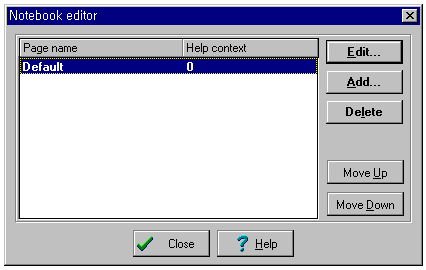

이렇게 배치하려면 노트북은 alClient로 정렬하고 탭셋은 alBottom으로 정렬해야 한다. 일반적으로 노트북은 폼 전체를 덮는 경우가 많으므로 alClient나 다른 정렬 속성을 사용하여 정렬해 주어야 한다.
폼이 처음 생성될 때 노트북의 Pages 속성을 탭셋의 Tabs 속성에 대입해 주어 탭셋이 노트북의 페이지 제목을 가질 수 있도록 해 주며 탭셋의 OnClick 이벤트에 다음 코드를 작성하여 탭셋이 클릭될 때 노트북의 페이지를 변경한다.

```
procedure TForm1.TabSet1Click(Sender: TObject);
begin
  Notebook1.PageIndex := TabSet1.TabIndex;
end;
```

노트북은 내부에 다른 컴포넌트를 가질 수 있는 컨테이너 컴포넌트이다. 페이지를 바꾸어 가며 디자인을 다르게 할 수 있다.

### ◘ 속성

#### Pages:TStrings
노트북의 각 페이지 제목을 가지며 이 속성에 입력된 제목의 수만큼 페이지가 생성된다. 예를 들어 세 개의 각각 다른 페이지를 만들고 싶다면 세 페이지의 이름을 정한 후 이 속성에 세 개의 이름 문자열을 대입해 주어야 한다.  오브젝트 인스펙터에서 이 속성을 더블클릭하면 다음과 같이 페이지 이름을 입력할 수 있는 대화상자가 열린다. 한글도 입력할 수 있다.




#### ActivePage:String
출력할 페이지의 이름을 지정한다. Pages 속성에 등록되어 있는 이름 중에 하나를 사용해야 한다. 문자열이므로 단순히 대입해도 되며 디자인중에는 이 속성을 더블클릭하여 다음 페이지로 전환한다.

#### PageIndex:Integer
출력할 페이지의 번호를 지정한다. Pages 속성에 등록한 탭의 이름은 등록된 순서대로 0, 1, 2, ...등의 인덱스 번호를 부여받으며 이 번호로 활동 페이지를 지정한다. ActivePage와 똑같은 목적에 사용되는 속성이지만 탭에 이름이 주어지지 않은 경우에는 ActivePage 속성으로 활동 페이지를 지정할 수 없기 때문에 이 속성을 사용해야 한다.

## 71. Header

### ◘ 정의
섹션별로 텍스트를 보여준다. Sections 속성에 문자열들을 입력해 놓으면 이 문자열이 각각 하나씩 섹션을 차지한다. 다음은

세 개의 섹션을 입력해 놓은 모습이다.


섹션의 크기는 조정가능하며 디자인시에 오른쪽 마우스를 드래그하여 조정하고 실행시에는 왼쪽 마우스를 드래그하여 조정한다. 섹션의 크기가 변경되면 onSizing 이벤트가 발생하며 크기 변경이 완료된 후에 OnSized 이벤트가 발생한다. 헤더는 리스트 박스나 스크롤 박스 등과 함께 폼을 수직적으로 분할하는 목적에 사용된다. 즉 헤더의 크기를 변경하면 리스트 박스의 크기를 조절한다거나 스크롤 박스의 크기를 조절하는 코드를 작성하여 실행중에 컴포넌트의 크기를 조절할 수 있도록 해준다. 파일 관리자나 탐색기에서 디렉토리 리스트 박스와 파일 리스트 박스의 크기를 조절하는 방법을 생각하면 헤더의 쓰임새를 쉽게 알 수 있을 것이다.

## ◘ 속성

### Sections:TStrings;

헤더에 나타날 각 섹션의 제목 문자열을 가지며 이 속성에 대입되는 문자열의 개수만큼 섹션이 생성된다. 첫 번째 입력된 문자열이 첫 번째 섹션이 되며 헤더의 제일 왼쪽에 나타난다. 오브젝트 인스펙터에서 이 속성을 더블클릭하여 문자열 리스트 편집기를 연 후 문자열들을 입력한다.

### AllowResize:Boolean;

사용자가 실행중에 섹션의 크기를 바꿀 수 있도록 할 것인가 아닌가를 지정한다. 디폴트는 True이므로 사용자가 실행중에 왼쪽 마우스 버튼으로 섹션의 경계 부근을 드래그하여 크기를 변경할 수 있다.

### 실 SectionWidth[i]:Integer;

i번째 섹션의 폭을 픽셀 단위로 조사해 주거나 이 속성에 값을 대입하여 폭을 변경한다.

# 72. FileListBox

## ◘ 정의

현재 디렉토리의 파일 목록을 출력해 주는 특별한 리스트 박스이다. 특정 디렉토리의 파일 목록을 보고자 할 경우 Directory 속성에 디렉토리 경로를 지정해 준다. 단독으로 사용되는 경우는 드물며 디렉토리 리스트 박스, 드라이브 콤보 박스와 함께 사용된다. 이때 드라이브 리스트 박스와 파일 리스트 박스는 연결되어야 하며 코드로 연결하는 방법과 속성으로 연결하는 방법이 있다. 코드로 연결할 경우는 디렉토리 리스트 박스의 OnChange 이벤트에서 파일 리스트 박스의 Directory 속성을 변경하도록 한다.

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

속성으로 연결할 경우는 디렉토리 리스트 박스의 FileList 속성에 연결할 파일 리스트 박스의 이름을 정해준다. 실행중에 사용자는 파일 리스트 박스에서 파일을 마우스로 선택하며 이때 선택된 파일의 이름은 FileName 속성으로 읽을 수 있다. FileName 속성은 경로를 포함한다.

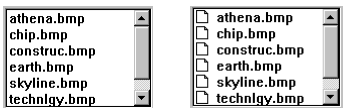
## ◘ 속성

### Mask:string;

파일 리스트 박스에 표시되는 파일의 조건을 와일드 카드로 설정한다. 예를 들어 *.exe로 설정하면 실행 파일의 목록만 보여준다. 여러 가지 조건을 설정하고자 할 경우에는 *.pas;*.c;*.bas 등과 같이 조건식을 세미콜론으로 구분해 주어야 한다. 디폴트 Mask는 *.*이며 모든 파일의 목록을 보여준다.

### ShowGlyph:Boolean;

파일 이름 옆에 파일의 종류를 나타내는 비트맵을 출력하도록 한다. 이 속성의 디폴트값은 False이므로 비트맵이 나타나지 않는다.




### FileType:TFileType;

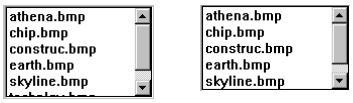
목록에 출력할 파일의 속성을 지정한다. 디폴트값은 보통 파일만 표시하도록 되어 있으나 세부 속성을 조정하면 숨은 파일이나 읽기 전용 파일도 표시해 준다. 세부 속성은 다음과 같다.

| 세부 속성 | 의미 |
|---|---|
| ftReadOnly | 읽기 전용 파일 |
| ftHidden | 숨은 파일 |
| ftSystem | 시스템 파일 |
| ftVolumeID | 디스크 볼륨 |

| | |
|---|---|
| ftDirectory | 디렉토리 |
| ftArchive | 기록 속성 |
| ftNormal | 보통 파일 |

### IntegralHeight:Boolean

파일 리스트 박스의 수직 길이를 조절한다. 이 속성이 True이면 최하단의 항목이 완전히 보일 정도의 수직 크기를 맞추어 주므로 항목의 일부만 보이는 경우가 없도록 해 준다. 이 속성의 디폴트값은 False이므로 최하단의 항목의 일부만 보일 수도 있다.




False인 경우 True인 경우

### ItemHeight:Integer;

리스트 박스의 한 아이템의 높이를 지정한다. 파일 리스트 박스에서 이 값은 파일 이름 간의 줄간이다. 디폴트값은 16이며 픽셀 단위로 지정한다.


16 20 30

### FileEdit:TEdit;

파일 리스트 박스에서 선택된 파일 이름이 출력될 에디트 박스를 지정한다. 만약 선택된 파일이 없을 경우에는 파일 리스트 박스의 Mask 속성이 에디트 박스에 출력된다.

### 실 FileName:string;

파일 리스트 박스에서 선택된 파일의 이름이며 드라이브와 디렉토리 경로를 포함한 완전 경로(full path)이다.

### 실 Directory:string;

파일 리스트 박스가 나타낼 디렉토리를 지정한다. 디자인 중에는 디렉토리를 변경할 수 없으며 실행중에 코드로 변경해 주어야 한다. 디렉토리 리스트 박스의 Directory 속성을 파일 리스트 박스의 Directory 속성에 대입하거나 아니면 디렉토리 리스트 박스의 FileList 속성으로 두 리스트 박스를 연결하여 자동으로 디렉토리가 바뀌도록 해 주어야 한다.

### 실 Items:TStrings;

파일 리스트 박스에 출력된 파일의 이름을 가지는 문자열 리스트이다. ItemIndex 속성으로 현재 선택된 파일의 인덱스를 조사할 수 있다.

### 실 ItemIndex:Integer;

리스트 박스에서 선택된 파일의 인덱스, 즉 번호이다. 첫 번째 파일이 선택되어 있다면 이 값은 0이고 두 번째 파일이 선택되어 있다면 이 값은 1이며 선택된 파일이 없으면 이 값은 -1이다. 선택된 파일의 이름을 조사하려면 FileListBox1.Items[FileListBox1.ItemIndex]를 조사하거나 아니면 FileName 속성을 곧바로 읽으면 된다.

### MultiSelect:Boolean;

한번에 여러 개의 파일이 선택될 수 있도록 한다. 디폴트로 이 값은 False이므로 한 번에 하나의 파일만 선택할 수 있다.

### 실 Selected[i]:Boolean;

리스트 박스에서 어떤 파일이 선택되었는가를 나타내며 진위형의 배열 형태를 가지고 있다. 조사하고자 하는 파일의 인덱스를 첨자로 사용하여 그 파일의 선택 여부를 조사한다. 예를 들어 다섯 번째 파일의 선택 여부를 알고 싶다면 Selected[4]를 조사하면 된다.

### 실 읽 TopIndex:Integer;

리스트 박스의 제일 위에 있는 항목의 번호를 나타낸다. 당연히 0번일 것 같지만 스크롤에 의해 다른 번호의 파일이 제일 윗쪽에 출력되어 있을 수도 있다.

## 73. DirectoryListBox

### ◘ 정의

현재 드라이브의 디렉토리 구조를 보여주며 디렉토리를 선택할 수 있도록 해준다. 다른 드라이브의 디렉토리 구조를 보고 싶으면 Drive 속성을 변경한다. 단독으로 사용되는 경우는 드물며 파일 리스트 박스, 드라이브 콤보 박스와 함께 사용된다. 이때 각각의 컴포넌트끼리 연결해 주어 한쪽의 변화가 다른 컴포넌트에 즉각 반영되도록 해야 하며 그 방법으로는 코드를 사용하는 방법과 속성을 사용하는 방법이 있다. 예를 들어 디렉토리 리스트 박스에서 디렉토리를 변경할 때 파일 리스트 박스의 목록이 변경되도록 하려면 다음과 같이 OnChange 이벤트를 사용한다.

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
```

```
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

속성으로 연결하고자 할 경우, 드라이브 콤보 박스의 DirList 속성에 드라이브 리스트 박스의 이름을 지정하며 디렉토리 리스트 박스의 FileList 속성에 파일 리스트 박스의 이름을 지정한다. 실행중에 사용자는 디렉토리 리스트 박스에서 디렉토리를 마우스로 더블클릭하여 선택할 수 있으며 이때 선택된 디렉토리명은 Directory 속성으로 읽을 수 있다.

## ◘ 속성

### IntegralHeight:Boolean;

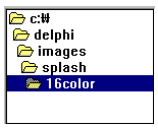
디렉토리  리스트 박스의 수직 길이를 조절한다. 이 속성이 True이면 최하단의 항목이 완전히 보일 정도의 수직 크기를 맞추어 주므로 항목의 일부만 보이는 경우가 없도록 해 준다. 이 속성의 디폴트값은 False이므로 최하단의 항목의 일부만 보일 수도 있다.

False인 경우

True인 경우

### ItemHeight:Integer;

리스트 박스의 한 아이템의 높이를 지정한다. 디렉토리 리스트 박스에서 이 값은 디렉토리 이름 간의 줄간이다. 디폴트값은 16이며 픽셀 단위로 지정한다.



16

30

### DirLabel:TLabel

디렉토리 리스트 박스에서 선택된 디렉토리의 이름을 출력해 주는 레이블을 지정한다.

### FileList:TFileListBox;

디렉토리 리스트 박스와 연결될 파일 리스트 박스를 지정한다. 디렉토리가 변경될 경우 연결된 파일 리스트 박스의 Directory 속성이 변경된 디렉토리로 자동 설정된다.

### 실 Drive:Char;

디렉토리 리스트 박스가 출력할 드라이브를 지정한다. 이 값은 직접 코드로 정의하는 경우보다는 드라이브 콤보 박스의 Drive 속성을 대입받거나 아니면 DirList 속성으로 두 컴포넌트를 연결하여 자동으로 변경되도록 한다.

### 실 Directory:string;

사용자가 디렉토리 리스트 박스에서 선택한 디렉토리명을 나타낸다. 이 값을 파일 리스트 박스의 Directory 속성에 대입하거나 FileList 속성으로 두 컴포넌트를 연결하여 자동으로 변경되도록 한다.

# 74. DriveComboBox

## ◘ 정의

시스템에 설치되어 있는 모든 드라이브의 목록을 가지는 특별한 콤보 박스이다. 사용자는 프로그램 실행중에 이 콤보 박스를 사용하여 드라이브를 선택할 수 있다. 이때 선택된 드라이브는 Drive 속성으로 읽을 수 있다. 드라이브 콤보 박스는 단독으로 사용되는 경우가 드물며 보통 파일 리스트 박스, 디렉토리 리스트 박스와 함께 사용된다. 이때 각각의 컴포넌트를 연결하여 한쪽의 변화가 다른 컴포넌트에 즉각 반영되도록 해야 하며 그 방법으로는 코드를 사용하는 방법과 속성을 사용하는 방법이 있다.  예를 들어 드라이브 콤보 박스에서 드라이브를 변경할 때 디렉토리 리스트 박스와 파일 리스트 박스의 내용을 변경시키려면 다음과 같이 OnChange 이벤트를 사용한다.

```
procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryListBox1.Drive := DriveComboBox1.Drive;
end;
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

드라이브 콤보 박스의 변화가 디렉토리 리스트 박스를 변화시키며 따라서 파일 리스트 박스의 내용도 변경된다. 속성으로 연결하고자 할 경우, 드라이브 콤보 박스의 DirList 속성에 디렉토리 리스트 박스의 이름을 지정하며 디렉토리 리스트 박스의 FileList 속성에 파일 리스트 박스의 이름을 지정한다.

## ◘ 속성

### DirList:TDirectoryListBox;
드라이브 콤보 박스와 연결될 디렉토리 리스트 박스를 지정한다. 드라이브가 변경될 경우 연결된 디렉토리 리스트 박스의 Drive 속성이 변경된 드라이브로 자동 설정된다.

### 실 Drive:Char;
사용자가 드라이브 콤보 박스에서 선택한 드라이브를 나타낸다. 이 속성값을 디렉토리 리스트 박스의 Drive 속성에 대입해 준다.

# 75. FilterComboBox

## ◘ 정의
필터를 선택할 수 있도록 해 주는 콤보 박스이다. 필터는 파일 리스트 박스에 표시될 파일의 조건을 지정한다. Filter 속성에 여러 가지 필터를 설정하며 사용자는 필터 콤보 박스에서 필터 중의 하나를 선택한다. 이때 선택된 필터는 Mask 속성으로 읽을 수 있다. 필터 콤보 박스는 필연적으로 파일 리스트 박스와 함께 사용되며 필터의 변화가 파일 리스트 박스의 목록에 즉각적인 영향을 줄 수 있도록 두 개의 컴포넌트가 연결되어 있어야 한다. 그 방법으로는 코드를 사용하는 방법과 속성을 사용하는 방법이 있다. 코드를 사용할 경우는 다음과 같이 필터 콤보 박스의 OnChange 이벤트를 사용한다.

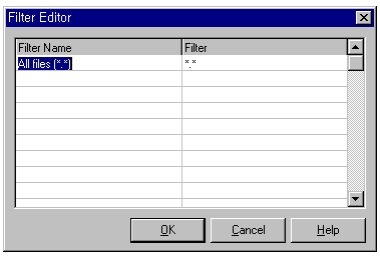
```
procedure TForm1.FilterComboBox1Change(Sender: TObject);
begin
  FileListBox1.Mask := FilterComboBox1.Mask;
end;
```

속성으로 연결할 경우는 필터 콤보 박스의 FileList 속성에 파일 리스트 박스의 이름을 지정해 준다.

## ◘ 속성

### FileList:TFileListBox;
필터 콤보 박스와 연결될 파일 리스트 박스를 지정한다. 필터가 변경될 경우 연결된 파일 리스트 박스의 Mask 속성이 변경되며 파일의 목록도 변경된다.

### Filter:string;
필터를 설정하며 "설명|필터" 형식으로 필터를 지정한다. 설명은 어떤 종류의 파일인가를 알려주는 문자열이며 필터는 *.exe와 같은 와일드 카드식으로 표현되며 가운데 파이프 문자(|)가 삽입된다. '워드 파일|*.doc','엑셀 파일|*.xls' 등이 하나의 필터가 된다. 여러 개의 필터를 계속 파이프로 연결하여 작성할 수 있으며 필터 사이사이에 파이프 문자로 구분해 준다. '워드 파일|*.doc|엑셀 파일|*.xls|그림 파일|*.gif'는 세 개의 필터를 정의한 예이다. 하나의 필터에 두 개 이상의 와일드 카드식을 쓰고 싶으면 '실행 파일|*.exe;*.com;*.bat' 등과 같이 세미콜론으로 구분하여 와일드 카드식을 나열한다.
필터를 입력하는 다른 방법은 오브젝트 인스펙터에서 Fileter 속성을 더블클릭하여 필터 편집기를 사용하는 방법이다.




왼쪽에 필터의 설명을 입력하고 오른쪽에 필터를 입력한다.

### 실 읽 Mask:string;
사용자가 실행중에 선택한 필터값을 가진다.

# 76. Application

## ◘ 정의
실행중인 프로그램 그 자체를 대표하는 컴포넌트이다. 컴포넌트 팔레트에는 없지만 델파이는 이 컴포넌트를 사용하여 프로그램을 작성한다. 즉 TApplication 형의 변수 Application을 생성하여 프로그램 인스턴스를 만들며 Run 메소드로 프로그램을 실행하고 Terminate 메소드로 프로그램을 종료한다. 이 컴포넌트의 속성, 이벤트, 메소드를 사용하면 프로그램 전체의 속성을 변경하거나 프로그램 전체에 영향을 미치는 이벤트를 작성할 수 있다. 팔레트에 이 컴포넌트가 없으므로 오브젝트 인스펙터를 통해 속성이나 이벤트를 대입할 수는 없으며 실행중에 코드로만 속성, 이벤트 값을 대입할 수 있다. OnActivate, OnDeactivate 이벤트는 프로그램이 활성화되거나 비활성화될 때 발생하며 OnHint 이벤트를 사용하여 도움말을 출력한다. OnMessage 이벤트는 윈도우즈가 프로그램으로 메시지를 보

내올 때 발생한다.

## ◘ 속성

### 읽 실 Components[i]:TComponent;

프로그램에 소속된 모든 컴포넌트의 리스트를 가지는 배열 속성이며 이 속성을 사용하면 배열 첨자를 사용하여 각 컴포넌트를 참조할 수 있다. 소속된 컴포넌트의 개수를 알고자 할 때는 ComponentCount 속성값을 읽으면 된다.

### 읽 실 Controls[i]:TControl;

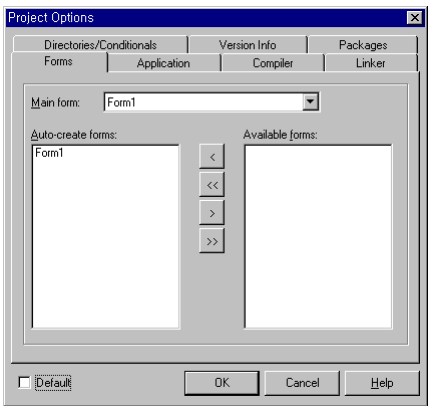
프로그램의 차일드인 모든 컨트롤의 목록을 가지는 배열 속성이며 이 속성을 사용하면 배열 첨자를 사용하여 각 컨트롤을 참조할 수 있다. Components 속성은 소속된 모든 컴포넌트의 목록을 가지는데 반해 Controls 속성은 폼의 차일드인 컨트롤의 목록만을 가진다. 예를 들어 패널이 있고 패널 안에 버튼이 있으면 버튼은 프로그램에 소속된(owned) 컴포넌트이므로 Components 속성의 목록에는 나타나지만 차일드는 아니므로 Controls 속성의 목록에는 나타나지 않는다. 차일드 컨트롤의 개수를 알려면 ControlCount 속성을 읽으면 된다.

### 읽 실 ExeName:String;

실행 파일의 이름을 가지며 이는 프로젝트 파일의 이름과 동일하다. 실행 파일이 속한 디렉토리 경로를 포함하는 완전 경로(full path)이다.

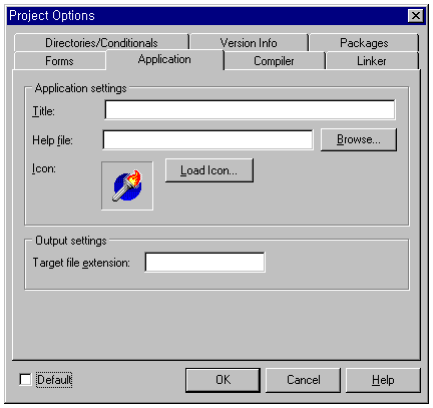
### 읽 실 MainForm: TForm;

프로그램 실행시 제일 먼저 생성되는 메인 폼을 지정한다. 메인 폼을 닫으면 프로그램이 끝난다. 이 속성을 변경하려면 Options/Project 옵션의 Forms 페이지를 변경한다.




### 읽 실 Icon: TIcon;

프로그램이 최소화될 때 나타날 아이콘을 지정하며 이 아이콘은 프로그램 관리자에 등록될 아이콘이다. Options/Project의 Application 페이지에서 아이콘을 변경한다.




### 실 HelpFile: string;

프로그램이 사용할 도움말 파일을 지정한다.

### 실 Title: string;

프로그램의 제목에 해당되며 프로그램이 최소화될 때 프로그램 아이콘 밑에 나타날 제목이다. 실행중에 변경하거나 Options/Project 메뉴 항목의 Application 페이지에서 선택할 수 있다.

### 실 HintColor: TColor

풍선 도움말의 색상을 설정한다.

# 77. Screen

## ◘ 정의

현재 화면 상태에 관한 정보를 가지는 컴포넌트이며 이 컴포넌트는 자동으로 생성되므로 속성값을 읽어 정보를 얻기만 하면 된다. 폼의 목록, 활동 폼, 활동 컨트롤, 화면 해상도, 폰트의 목록, 커서의 목록 등을 가진다.

## ◘ 속성

### 읽 실 Fonts: TStrings;

화면에 나타낼 수 있는 폰트의 목록을 가진다. 참고로 프린터로 출력될 수 있는 폰트의 목록은 TPrinter 오브젝트의 Fonts 속성으로 읽을 수 있다.

### 실 Cursors[Index: Integer]: HCursor;

프로그램에서 사용할 수 있는 커서의 목록을 가진다. 이 커서 목록을 변경하면 사용자 정의 커서를 만들어 사용할 수 있다. 다음은 기본적으로 제공되는 커서의 목록이다.

| 커서 | 값 |
|---|---|
| crDefault | 0 |
| crNone | -1 |
| crArrow | -2 |
| crCross | -3 |
| crIBeam | -4 |
| crSize | -6 |
| crSizeNESW | -7 |
| crSizeNS | -8 |
| crSizeNWSE | -9 |
| crSizeWE | -8 |
| crUpArrow | -10 |
| crHourGlass | -11 |
| crDrag | -12 |
| crNoDrop | -13 |
| crHSplit | -14 |
| crVSplit | -15 |
| crMultiDrag | -16 |

### 읽 실 Forms[Index: Integer]: TForm;

화면상의 폼 목록을 가지며 인덱스를 사용하여 개별 폼을 참조할 수 있도록 해준다. 첫 번째 폼의 인덱스가 0이며 두 번째 폼의 인덱스가 1, 세 번째 폼의 인덱스가 2 ...

### 읽 실 Width,Height: Integer;

현재 설정된 비디오의 해상도를 픽셀 단위로 조사한다.

### 읽 실 PixelsPerInch: Integer;

현재 설치된 비디오 드라이버에서 사용하는 인치 당 픽셀수를 조사한다. 이 값은 델파이가 로드될 때 윈도우즈로부터 조사해 본 값이다.

# 78. Session

## ◘ 정의

데이터 베이스와 프로그램 간의 전반적인 연결을 담당하는 컴포넌트이다. 이 컴포넌트가 만들어지는 과정을 명확히 볼 수는 없지만 메소드나 속성을 사용하여 프로그램에 광범위한 영향을 미친다. Database 속성은 활동중인 모든 DB의 목록을 가지는 배열이며 DataCount는 그 개수를 가진다.

# 79. MenuItem

## ◘ 정의

컴포넌트 팔레트에는 없으며 메뉴 디자이너에서 메뉴 항목을 만들 때 자동으로 생성되는 컴포넌트이다. 각각의 메뉴 항목에 해당되며 메뉴 항목의 여러 가지 속성을 지정한다. 실행중에 사용자가 메뉴 항목을 선택하면 OnClick 이벤트가 발생한다.

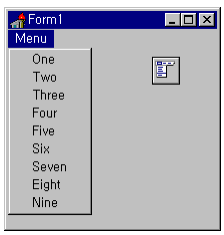
## ◘ 속성

### Break:TMenuBreak;

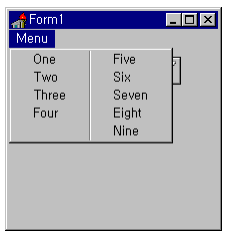
메뉴 항목이 많을 경우 여러 개의 칼럼으로 메뉴를 나눈다. 다음 세 가지 값 중 하나를 선택한다.

| 속성 | 의미 |
|---|---|
| mbNone | 칼럼을 나누지 않는다(디폴트). |
| mbBarBreak | 다음 칼럼으로 항목을 나눈다. 칼럼 사이에 세로 구분선을 넣어준다. |
| mbBreak | 다음 칼럼으로 항목을 나눈다. 칼럼 사이에 세로 구분선을 넣지 않는다. |

다음 예는 five 항목의 속성을 mbBarBreak로 설정한 경우와 mbNone으로 설정한 경우이다.



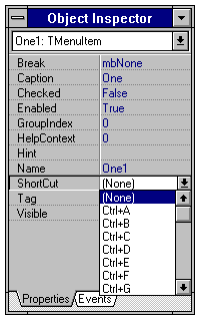
mbNone



mbBarBreak

### ShortCut:TShortCut;

단축키(Short Cut)를 지정한다. 이때의 단축키는 Caption에 &를 넣어 지정하는 단축키(Accelerator)와는 다르다. 액셀러레이터는 Alt 키와 함께 쓰며 메뉴가 활성화되었을 때만 유효하나 쇼트컷은 폼이 활성화되어 있으면 항상 유효하다는 차이가 있다. 오브젝트 인스펙터의 Shortcut을 선택하면 다음과 같은 단축키 리스트 박스를 보여준다. 이렇게 지정된 단축키는 실행시 메뉴 항목 옆에 같이 출력된다.




# 80. Field

## ◘ 정의

레코드에 속한 개별 필드를 대표하는 컴포넌트이다. 컴포넌트 팔레트에 없으므로 오브젝트 인스펙터를 통하여 이 컴포넌트의 속성을 설정할 수는 없으며 데이터 셋(TTable, TQuery)이 활성화될 때 자동으로 생성된다. 자동으로 생성된 TField 컴포넌트는 테이블의 모든 필드를 가진다. 디자인 중에 이 컴포넌트를 정적으로 생성하려면 필드 에디터를 사용하며 필드 에디터는 TTable이나 TQuery를 더블클릭하여 연다. 이 컴포넌트의 속성을 조절하면 데이터 베이스의 필드 속성을 일부 변경할 수 있다. 예를 들어 출력하고자 하는 필드를 선택하거나 필드의 폭을 조절하고 출력 형식을 변경할 수 있다. TField 컴포넌트는 추상형의 오브젝트이므로 직접 인스턴스를 생성할 수 있으며 이 컴포넌트로 파생된 여러 가지 컴포넌트로 필드 인스턴스를 생성한다. 이 컴포넌트의 속성을 알게 되면 나머지 파생 컴포넌트의 속성도 쉽게 알 수 있다.

## ◘ 속성

### Alignment:TAlignment;

데이터 컨트롤에 나타날 데이터의 정렬 방식을 설정한다. 디폴트로 문자는 좌측 정렬되며 수치는 우측 정렬되지만 이 속성으로 필드의 정렬 방식을 강제 설정할 수 있다.

### Visible:Boolean;

필드의 출력 여부를 설정한다. 디폴트는 True이지만 이 속성을 False로 바꾸면 특정 필드를 숨길 수 있다.

### DisplayWidth:Integer;

그리드로 출력될 경우 필드가 차지할 폭을 문자 단위로 설정한다. 그리드에서 필드가 차지하는 폭의 길이는 데이터 베이스에 정의된 필드의 폭과 같지만 이 속성을 조정하여 출력시의 폭을 설정한다.

### ReadOnly:Boolean;

읽기 전용의 필드로 만든다. 데이터를 편집할 수 없으며 그리드에서 Tab을 누를 경우 읽기전용 속성을 가진 필드는 건너뛴다.

### 실 AsInteger: Longint;

필드값을 Longint형으로 읽거나 쓰고자 할 때 이 속성을 사용한다. 문자열형 필드일 경우 이 속성을 사용하면 정수값을 문자열로 바꾸어 값을 교환할 수 있다. 이 속성 외에도 AsString, AsFloat 등 동일한 목적으로 사용되는 속성들이 있다.

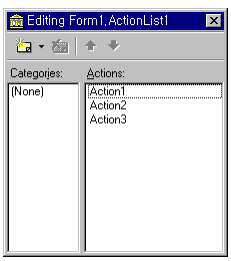
### 실 읽 DataType: TFieldType;

필드의 데이터 타입을 나타낸다. 가능한 데이터 타입은 ftBoolean, ftDateTime, ftInteger, ftBlob, ftString 등등 데이터 베이스의 필드로 사용되는 모든 타입들이다.

# 81.Action

## ◘ 정의

복수 개의 컨트롤로부터의 명령을 집중화하기 위한 오브젝트이다. 컴포넌트 팔레트에는 없으며 액션 리스트의 액션 편집기로부터 생성된다.




이 대화상자의 New 버튼을 눌러 액션을 생성시키며 일단 생성

된 액션은 이 편집기에서 선택한 후 오브젝트 인스펙터에서 속성을 변경할 수 있다. 액션과 연결하려는 개별 컴포넌트의 Action 속성에 액션 컴포넌트를 지정해 주면 컴포넌트 선택시 액션의 OnExecute 이벤트가 발생한다. Caption, Checked, Enabled, Hint, ImageIndex, ShortCut, Visible 등의 일반적인 속성은 액션과 연결되는 컴포넌트의 속성으로 복사되며 실행중에도 액션의 속성을 변경하면 연결된 컴포넌트의 속성이 같이 변경된다.

## ◘ 속성

### 실 DisableIfNoHandler:Boolean;

핸들러가 지정되어 있지 않으면 액션을 사용 금지시킨다. 이 값은 디폴트로 True 이지만 False 로 변경해 주면 핸들러가 없어도 선택은 가능하도록 할 수 있다.

## ◘ 이벤트

### OnExecute

실행중에 사용자에 의해 액션이 선택되었을 때 발생하는 이벤트이다. 액션은 사용자가 직접 사용하는 컴포넌트가 아니므로 액션에 의해 이 이벤트가 발생하지는 않으면 액션과 연결된 컴포넌트에 의해 발생한다. 예를 들어 액션이 버튼과 연결되어 있으면 버튼의 OnClick 이벤트 핸들러가 이 이벤트의 핸들러와 자동으로 연결된다.

### OnUpdate

액션의 상태를 변경시키기 위해 발생하는 이벤트이다. 응용 프로그램이 처리할 메시지가 없는 아이들 타임에 이 이벤트를 발생시키며 보통 이 이벤트 핸들러에서는 액션의 Enabled, Checked 속성을 조건에 따라 변경시킨다. 이 핸들러에서 액션의 속성을 변경하면 연결된 모든 컴포넌트의 속성이 한꺼번에 변경되는 효과가 발생한다.

# 2.오브젝트 레퍼런스

오브젝트의 속성은 모두 실행시에만 사용할 수 있는 실행시 전용 속성이다.

## 1. BitMap

### ◘ 정의

BMP 파일 포맷의 비트맵 그래픽을 담는 오브젝트이다. 비트맵 오브젝트에는 윈도우즈의 비트맵(HBITMAP)과 팔레트(HPALETTE)가 같이 기억되며 팔레트를 자동으로 조절하는 능력을 가지고 있다.
비트맵의 크기는 Height, Width 속성으로 지정하며 Monochrome 속성으로 흑백 비트맵인지 컬러 비트맵인지를 알아낸다. 외부 비트맵 파일을 읽어들일 때는 LoadFromFile 메소드를 사용하며 파일로 저장할 때는 SaveToFile 메소드를 사용한다.
Canvas 속성에 캔버스 오브젝트를 가지므로 TCanvas가 가지는 모든 메소드를 사용하여 비트맵의 표면에 그림을 그릴 수 있으며 Draw나 StretchDraw를 사용하여 다른 캔버스 오브젝트의 비트맵을 복사해 올 수도 있다. 비트맵의 내용이 변경되면 OnChange 이벤트가 발생한다.

### ◘ 속성

#### 읽 Empty:Boolean

비트맵에 그래픽이 들어 있는지의 여부를 나타낸다.

#### Palette:HPalette

비트맵의 색상 배치 상태인 팔레트 정보를 가진다.

#### Height, Width:Integer;

비트맵의 폭과 높이를 픽셀 단위로 나타낸다.

#### Monochrome:Boolean

이 속성이 True이면 흑백의 비트맵이다.

### ◘ 메소드

■ procedure LoadFromFile(const FileName: string);
파일로부터 비트맵을 읽어들인다.

■ SaveToFile(const FileName: string);
파일로 비트맵을 출력한다.

## 2. Brush

### ◘ 정의

브러시는 직사각형이나 타원형 등의 닫혀진 폐곡선 영역을 채울 때 사용되며 브러시 오브젝트의 속성을 변경하여 그려지는 그림의 채움 속성을 변경한다. 단일 색으로 채색을 할 수도 있으며 무늬를 가지거나 비트맵으로 채울 수도 있다. 브러시의 속성이 변경되면 OnChange 이벤트가 발생한다.

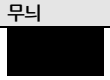
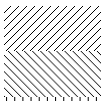
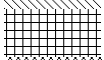
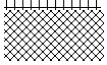
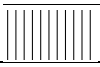
### ◘ 속성

#### Color:TColor

브러시의 색상이다. 사용할 수 있는 색상은 속성 레퍼런스의 Color 속성을 참고하기 바란다.

#### Style:TBrushStyle;

채울 면의 패턴을 정의한다. 다음과 같은 채움 패턴이 있다.

| 속성값 | 무늬 |
|---|---|
| bsSolid |  |
| bsClear | |
| bsBDiagonal |  |
| bsFDiagonal | |
| bsCross |  |
| bsDiagCross |  |
| bsHorizontal |  |
| bsVertical |  |

#### Bitmap:TBitmap

실행중에만 사용할 수 있다. 8*8 크기의 비트맵을 사용하여 채움 패턴을 만든다.

# 3. Canvas

## ◘ 정의

그림이 그려지는 표면이며 프로그램은 캔버스에 그림을 그린다. 윈도우즈의 Display Context에 해당한다. Brush, Pen, Font 등의 속성으로 그림을 그리는 데 필요한 브러시, 펜, 폰트의 속성을 설정하며 LineTo, Circle 등의 메소드를 사용하여 캔버스에 그림을 그린다. 캔버스는 다음 그림이 그려지는 위치(CP)를 항상 기억하며 LineTo 함수로 그림을 그릴 때 이 위치를 선의 시작점으로 사용한다.

## ◘ 속성

### PenPos: TPoint;

펜의 현재 위치를 나타내며 선을 그을 때 이 위치를 시작점으로 사용한다. 현재 위치를 옮길 때는 MoveTo 메소드를 사용한다.

### Pixels[X, Y: Longint]: TColor;

캔버스의 전체 표면을 점 단위로 읽거나 쓸 수 있는 2차원 점 배열이다. X,Y 좌표를 첨자로 사용하여 색상값을 대입하면 점을 찍는 것이며 속성값을 읽어내면 점의 색상을 조사하는 것이다.

### 읽 ClipRect: TRect;

그림이 그려지는 영역을 정의하는 클리핑 영역이다. 이 영역밖에 그려지는 그림은 출력되지 않으며 화면으로도 보이지 않게 된다.

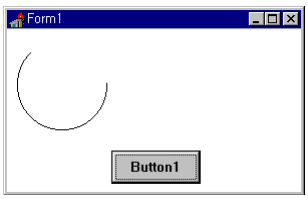
### CopyMode: TCopyMode;

한 캔버스에서 다른 캔버스로 이미지가 복사될 때 복사되는 방법을 정의한다. 이 속성은 디폴트가 cmSrcCopy로 되어 있으며 이 모드는 복사되는 이미지가 원래의 이미지를 덮어버리는 방법이다. 복사 모드를 변경하면 이미지끼리 겹치거나 반전되는 효과를 가져올 수 있다.

## ◘ 메소드

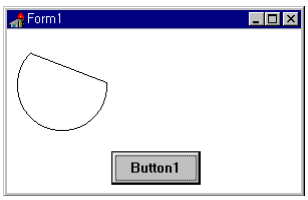
■ procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);

원호를 그린다. (X1,Y1)-(X2,Y2)는 원호에 외접하는 직사각형이며 (X3,Y3),(X4,Y4)는 시작점, 끝점을 지정한다. 시작점과 중심을 연결한 선이 원과 만나는 점부터 원호가 그려진다. 도스의 각도로 시작각, 끝각을 지정하는 방법보다 상당히 지저분한 방법을 사용하고 있다. 다음 그림은 arc(10,10,100,100,50,50,150,50);을 사용하여 그린 그림이다.



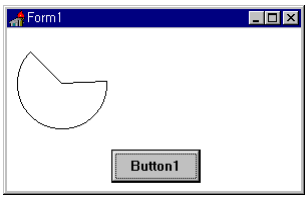

■ procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);

원호를 그리고 시작점과 끝점을 선분으로 잇는다. 인수의 의미는 Arc 메소드와 동일하다. 같은 인수를 사용하여 그려본 그림은 다음과 같다.



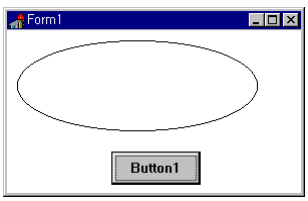

■ procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Longint);

Chord나 Arc와 사용하는 인수는 같으며 끝점을 중심과 잇는다. 출력 결과만 보면 어떤 동작을 하는지 쉽게 알 수 있을 것이다.



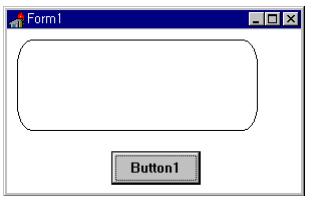

■ procedure Ellipse(X1, Y1, X2, Y2: Integer);

타원을 그린다. (X1,Y1)-(X2,Y2)는 타원에 외접하는 직사각형이다. 다음 그림은 Ellipse(10,10,250,100);로 그린 그림이다.



■ Rectangle(X1, Y1, X2, Y2: Integer);
(X1,Y1)을 시작점으로 하며 (X2,Y2)를 끝점으로 하여 두 점을 잇는 선분을 대각선으로 하는 직사각형을 그린다.

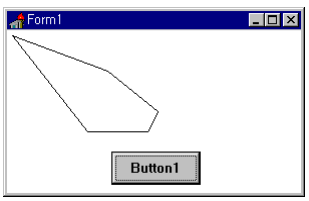
■ RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);
모서리가 둥근 직사각형을 그린다. 모서리의 둥근 정도는 X3,Y3 인수가 지정하는 타원의 반지름으로 정해진다. 타원이 모서리에 접한다고 생각하면 편리하다. 다음 그림은 RoundRect(10,10,250,100,10,10);로 그렸다.




■ FillRect(const Rect: TRect);
지정한 사각 영역을 현재의 브러시를 사용하여 채운다.

■ procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);
X,Y 좌표를 시작점으로 하여 Color 경계선으로 둘러싸인 영역을 현재 브러시를 사용하여 전부 채색한다. 채색하는 방법은 FillStyle 인수로 지정하며 이 인수가 FillBorder일 경우 Color 경계선을 만날 때까지 내부를 모두 칠하며 FsSurface일 경우 Color 색상을 가진 영역을 모두 채색한다.

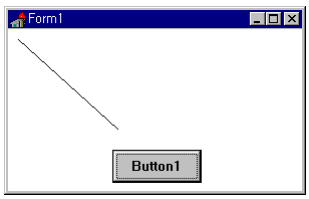
■ procedure Polyline(Points: array of TPoint);
현재 설정된 펜으로 다각형을 그린다. 다각형의 각 꼭지점을 이루는 점의 좌표 배열을 인수로 넘겨준다. 다음 그림은 PolyLine([Point(5, 5), Point(100, 40), Point(150, 80), Point(140, 100), Point(80, 100), Point(5, 5)]); 에 의해 그려진 것이다. 불규칙한 모양의 다각형을 그리고자 할 때 사용한다.




■ procedure Polygon(Points: array of TPoint);
PolyLine 메소드와 사용하는 방법은 동일하며 동작도 비슷하지만 다각형을 그리고 난 후에 다각형의 내부를 현재 선택된 브러시로 채색한다는 점이 다르다.

■ procedure MoveTo(X, Y: Integer);
펜의 현재 위치를 X,Y로 변경한다. 이 위치는 다음 LineTo 메소드에 의해 선이 그려질 때 선의 시작점이 된다.

■ procedure LineTo(X, Y: Integer);
펜의 현재 위치에서 X,Y 위치로 선을 긋는다. 다음은 MoveTo와 LineTo 함수로 (10,10)-(110,100)으로 선을 그은 것이다.




■ procedure FrameRect(const Rect: TRect);
Rect가 지정하는 사각 영역의 테두리를 그리되 내부는 채우지 않는다.

■ procedure DrawFocusRect(const Rect: TRect);
사각형을 그리되 사각 영역에 포커스를 가진 것을 나타내는 듯한 모양으로 그린다. XOR 연산에 의해 그림을 그리므로 이 메소드를 두 번 호출하면 원래 그림을 손상시키지 않고 사각형을 다시 지울 수 있다.

■ procedure TextOut(X, Y: Integer; const Text: string);
X,Y 위치에 문자열을 출력한다.

■ function TextHeight(const Text: string): Integer;
문자열의 높이를 픽셀 단위로 구한다.

■ function TextWidth(const Text: string): Integer;
문자열의 총 폭을 픽셀 단위로 구한다.

■ procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);
사각 영역을 설정하고 이 영역 안에서만 문자열을 출력한다. 사각 영역 밖으로 벗어나는 문자열은 출력되지 않는다.

■ procedure Draw(X, Y: Integer; Graphic: TGraphic);
X,Y 위치에 그래픽을 출력한다. 그래픽은 비트맵, 아이콘, 메타파일 중의 하나이다.

■ procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);
Draw 메소드와 동작은 동일하되 사각 영역을 설정하고 사각영역에 꽉 차도록 그래픽을 확대하거나 축소한다.

■ procedure CopyRect(Dest: TRect; Canvas:

TCanvas; Source: TRect);
다른 캔버스의 특정 사각 영역을 복사한다. Dest는 복사한 이미지가 출력될 사각 영역이며 Canvas는 복사 원본이 될 캔버스이며 Source는 원본 캔버스의 복사 대상 영역이다. Canvas 인수로 자기 자신을 지정하면 같은 캔버스 내에서 복사를 수행할 수도 있다.

# 4. Clipboard

## ◘ 정의

윈도우즈의 클립보드 기능을 수행하는 오브젝트이다. cut, copy, paste 등의 기능을 사용할 때 이 오브젝트가 사용된다. Clipbrd 유닛 내에 이미 Clipboard 오브젝트를 선언해 두었으므로 직접 오브젝트를 만들 필요없이 만들어진 Clipboard 오브젝트를 사용하기만 하면 된다. AsText 속성을 사용하여 텍스트를 클립보드로 보내거나 읽는다.

## ◘ 속성

### AsText: String;

이 속성을 읽으면 클립보드에 보관된 문자열을 읽을 수 있다. 단 클립보드에 문자열 이 외의 데이터가 들어있을 경우는 에러가 발생한다. 또한 이 속성에 문자열을 대입하면 클립보드로 문자열을 복사하게 된다.

### 읽 Formats[Index: Integer]: Word;

클립보드에 포함되어 있는 포맷의 목록을 가지는 배열형 속성이다. 포맷의 개수는 FormatCount 속성으로 읽는다.

## ◘ 메소드

■ procedure Assign(Source: TPersistent);
Source 오브젝트를 클립보드로 복사한다. 오브젝트가 그래픽이면 그래픽 이미지가 적절한 포맷으로 클립보드로 복사된다. 예를 들어 MyBitmap이라는 비트맵을 클립보드로 복사할 경우 다음과 같이 한다.

```
Clipboard.Assign(MyBitmap);
```

클립보드에 있는 비트맵을 다시 읽으려면 다음과 같이 비트맵 오브젝트의 Assign 메소드를 사용한다.

```
MyBitmap.Assign(Clipboard);
```

■ function HasFormat(Format: Word): Boolean;
클립보드에 특정 형태의 데이터가 들어있는지를 점검한다. Format 인수로는 다음과 같은 값들이 사용된다.

| 포맷 | 의미 |
|---|---|
| CF_TEXT | 문자열 |
| CF_BITMAP | 비트맵 |
| CF_METAFILE | 메타 파일 |
| CF_PICTURE | TPicture.오브젝트 |
| CF_OBJECT | 그외 오브젝트(컴포넌트 등) |

# 5. Font

## ◘ 정의

텍스트 출력에 사용되는 글꼴을 정의하는 오브젝트이며 윈도우즈의 HFONT에 대응된다. 여러 가지 속성을 설정하며 글꼴의 모양을 변경한다.

## ◘ 속성

### Name: TFontName;

글꼴의 이름이며 글꼴의 모양을 결정짓는 가장 중요한 요소이다. 쉽게 말해서 명조체, 고딕체, Times, Arial 등의 폰트명이다.

### Color: TColor;

글꼴의 색상이다.

### Style: TFontStyles;

글꼴의 속성이라고 할 수 있으며 다음과 같은 진위형의 세부 속성을 가진다. 각각의 세부 속성을 조합하여 글꼴의 모양을 장식할 수 있다.

| 속성 | 의미 |
|---|---|
| fsBold | 굵은 문자 |
| fsItalic | 기울임체 |
| fsUnderline | 밑줄 |
| fsStrikeout | 가운데줄 |

### Size: Integer;

폰트의 크기이다. internal leading을 포함하는 Height와는 조금 다른 의미를 가지고 있으며 사용자들은 보통 Size로 폰트의 크기를 조절한다.

Height: Integer;
폰트의 높이를 픽셀 단위로 나타낸다. Size 속성에 internal leading을 더한 값이며 일반적으로 사용자는 이 속성보다는 Size 속성으로 폰트의 크기를 조절하는 것이 좋다.

## 6. Graphic

### ◘ 정의
TBitmap, TIcon, TMetaFile 오브젝트의 기반 클래스이다. 어떤 형태의 그래픽을 사용할 것인가가 정해져 있다면 특정 형태의 오브젝트(TBitmap, TIcon, TMetaFile)를 직접 사용하는 것이 좋지만 세 가지 형태의 속성을 동시에 사용하고자 할 경우는 TPicture 오브젝트를 사용하는 것이 좋다.

### ◘ 속성

읽 Empty:Boolean
그래픽이 들어 있는지의 여부를 나타낸다.

Height, Width:Integer;
그래픽의 폭과 높이를 픽셀 단위로 나타낸다.

### ◘ 메소드

■ procedure LoadFromFile(const FileName: string);
파일로부터 그래픽을 읽어들인다.

■ SaveToFile(const FileName: string);
파일로 그래픽을 출력한다.

## 7. Icon

### ◘ 정의
아이콘 그래픽을 담으며 윈도우즈의 TICON에 대응된다.

### ◘ 속성

Height, Width:Integer;
아이콘의 폭과 높이를 픽셀 단위로 나타낸다.

## 8. IniFile

### ◘ 정의
윈도우즈의 초기화 파일인 INI 파일을 제어할 수 있도록 해주는 오브젝트이다. INI 파일로 데이터를 출력하거나 읽을 수 있는 여러 가지 메소드를 제공한다. INI 파일을 사용하는 예는 다음과 같다.

```
var
  MyIni: TIniFile;
begin
  MyIni := TIniFile.Create('Delphi.Ini');
  {데이터 입출력 메소드}
  MyIni.Free;
end;
```

먼저 TIniFile 타입의 변수를 선언하고 Create 메소드로 객체를 생성한다. 이때 Create 메소드의 인수로 입출력의 대상이 될 INI 파일의 이름을 기입해 준다. INI 파일로 데이터를 입출력하는 메소드를 사용한 후 Free 메소드로 객체를 해제해준다.
데이터 입출력 메소드는 공통적으로 Section과 Ident, Default 등의 인수를 가지고 있다. Section은 [ ] 괄호 안의 문자열을 의미하며 Ident는 =의 좌변에 놓인 읽고자 하는 정보의 이름이며 Default는 읽고자 하는 항목이 발견되지 않을 경우 취하는 값이다.

### ◘ 속성

읽 FileName: string;
TIniFile 오브젝트가 사용하고 있는 INI 파일의 이름이며 이 이름은 Create 메소드의 인수로 전달된 파일의 이름이다.

### ◘ 메소드

■ constructor Create(const FileName: string);
TIniFile 객체를 생성한다. 인수로는 INI 파일의 이름을 넘겨주며 디렉토리 경로 없이 파일의 이름만 주어질 경우 윈도우즈 디렉토리에서 파일을 찾는다.

■ procedure Free;
TIniFile 객체를 파괴한다. 사용하고 난 후에는 반드시 이 메소드를 호출하여 객체를 해제해 주어야 한다.

■ function ReadBool(const Section, Ident: string; Default: Boolean): Boolean;

INI 파일로부터 진위형의 값을 읽는다.

■ function ReadInteger(const Section, Ident: string; Default: Longint): Longint;
INI 파일로부터 정수형의 값을 읽는다.

■ function ReadString(const Section, Ident, Default: string): string;
INI 파일로부터 문자열을 읽는다.

■ procedure ReadSection (const Section: string; Strings: TStrings);
INI 파일로부터 섹션 전체를 문자열 리스트로 읽어들인다.

■ procedure WriteBool(const Section, Ident: string; Value: Boolean);
INI 파일에 진위형의 값을 출력한다.

■ procedure WriteInteger(const Section, Ident: string; Value: Longint);
INI 파일에 정수형의 값을 출력한다.

■ procedure WriteString(const Section, Ident, Value: string);
INI 파일에 문자열을 출력한다.

■ procedure EraseSection(const Section: string);
INI 파일의 섹션 전체를 삭제한다.

■ procedure ReadSectionValues(const Section: string; Strings: TStrings);
INI 파일로부터 전체 섹션을 문자열 리스트로 읽어들인다. 문자열 리스트의 Values 속성을 사용하여 문자열 리스트 내의 문자열을 읽는다.

# 9. Metafile

## ◘ 정의

메타 파일 그래픽(WMF 확장자)을 담는다.

## ◘ 속성

### Height, Width:Integer;

메타 파일의 폭과 높이를 픽셀 단위로 나타낸다.

# 10. Pen

## ◘ 정의

캔버스에 선을 그을 때 사용되며 펜 오브젝트의 속성에 따라 그려지는 선의 모양이 달라진다. Color 속성으로 펜의 색깔을 설정하며 Width 속성으로 펜의 굵기를 지정한다.

## ◘ 속성

### Color:TColor

펜의 색상을 지정한다. 디폴트값은 검정색(clBlack)이다.

### Width:Integer

펜의 굵기를 픽셀 단위로 설정한다. 디폴트값은 1이다.

### Style:TPenStylr

펜이 그리는 선의 모양을 설정한다. 속성에 따른 선의 모양은 다음과 같다.

| 속성값 | 선 모양 |
|---|---|
| psSolid | 실선 |
| psDash | 긴 점선 |
| psDot | 점선 |
| psDashDot | 일점 쇄선 |
| psDashDotDot | 이점 쇄선 |
| psClear | 그리지 않음 |
| psInsideFramd | 도형의 안쪽으로 밀착되는 선 |

### Mode:TPenMode;

펜의 색상과 선이 그어질 캔버스의 배경 색상과의 관계를 정의한다.

# 11. Picture

## ◘ 정의

비트맵, 아이콘, 메타 파일 그래픽을 담는 오브젝트이며 어떤 종류의 그래픽을 담을 것인가는 Graphic 속성으로 지정한다. 비트맵을 담고 있다면 Bitmap 속성으로 그래픽을 읽고 아이콘을 담고 있다면 Icon 속성, 메타 파일을 담고 있다면 metafile 속성을 읽는다.

## ◘ 속성

### Graphic: TGraphic;
어떤 종류의 그래픽을 담고 있는가를 나타내며 bitmap, icon, metafile 중 하나이다.

### Height, Width:Integer;
그래픽의 폭과 높이를 픽셀 단위로 나타낸다.

# 12. Printer

## ◘ 정의
Printer 오브젝트는 윈도우즈에서의 프린터 작업에 사용되며 프린트에 관한 모든 기능을 내장하고 있다. 프린터로 문자열이나 그래픽을 출력하는 방법은 화면으로 출력하는 방법과 동일하되 다만 Printer 오브젝트로 출력을 보낸다는 점이 다르다. 인쇄를 하려면 BeginDoc 메소드를 호출하여 인쇄 내용을 보내며 EndDoc로 인쇄 작업을 종료한다. 실제로 프린터가 인쇄를 시작하는 시점은 EndDoc 메소드가 호출되었을 때이다.
인쇄 작업에 문제가 있을 경우 Abort 메소드로 인쇄를 중단할 수 있다. 프린터가 인쇄를 하고 있는지 에러가 발생했는지는 Printing, Aborted 속성으로 조사하며 두 속성 모두 Boolean 형이다. Printer 오브젝트는 이미지나 폼과 마찬가지로 인쇄 표면에 해당하는 Canvas 속성을 가진다. Canvas 오브젝트의 출력 메소드를 사용하면 폼으로 출력하는 방법으로 프린터로 출력할 수 있고 Brush, Font, Pen 속성으로 그래픽의 속성을 조정한다.
Canvas를 사용하여 출력하는 방법 외에 AssignPrn 메소드로 프린터를 텍스트 파일과 연결한 후 Writeln으로 문자열을 출력하는 방법도 있다. NewPage 메소드는 현재 인쇄중인 용지를 새 용지로 교체한다.

## ◘ 속성

### 읽 Aborted:Boolean;
사용자가 Abort 메소드를 사용하여 인쇄를 중지했는지를 조사한다.

### 읽 Printing:Boolean;
BeginDoc를 호출하면 이 속성이 True가 되며 EndDoc나 Abort를 호출하면 False가 되어 현재 인쇄 진행중인지를 나타낸다.

### 읽 Fonts:TStrings;
프린터가 지원하는 폰트의 목록을 가진다. 프린터가 그래픽을 인쇄할 수만 있다면 트루타입 폰트 전체를 사용할 수 있다.

### 읽 PageWidth, PageHeight:Integer;
인쇄 지면의 폭과 높이를 픽셀 단위로 나타낸다.

### Orientation:TPrinterOrientation;
인쇄 방향이 가로(Landscape)인지 세로(Portrait)인지를 나타낸다.

### 읽 PageNumber:Integer;
현재 인쇄하고 있는 페이지의 번호이다.

### 읽 Printers:TStrings;

### PrinterIndex:Integer;
현재 윈도즈에 설치되어 있는 프린터의 목록을 가지며 PrinterIndex는 설치된 프린터 중 선택된 프린터의 번호이다.

### Title:string;
인쇄 작업중에 인쇄 관리자에 나타날 문자열을 지정한다.

## ◘ 메소드

■ procedure BeginDoc;
프린터로 인쇄 작업을 보내기 시작한다. 이 메소드 호출 후에 실제 프린터로 데이터를 출력하는 문장이 온다.

■ procedure EndDoc;
인쇄 작업을 종료한다. 이 메소드가 호출되어야 프린터는 실제 인쇄 작업을 시작한다.

# 13. StringList

## ◘ 정의
컨트롤에 연결되지 않는 독립적인 문자열 리스트이다. Strings 오브젝트와 모든 면에서 동일하므로 자세한 사항은 Strings 오브젝트를 참조하기 바란다.

## 14. Strings

### ◘ 정의

문자열 리스트이며 일종의 문자열 배열을 이루는 오브젝트이다. 문자열 배열을 사용하는 많은 컴포넌트들이 이 오브젝트를 사용한다. 문자열들은 Strings 오브젝트에 직접 저장되지 않으며 Strings 오브젝트를 사용하는 컨트롤에 저장된다. 예를 들어 리스트 박스의 Items 속성에 문자열들을 입력하면 이 문자열들은 TStrings로부터 파생된 TListBoxStrings 오브젝트에 저장된다. 컨트롤의 외부에 저장되는 독립적인 문자열 리스트를 사용하려면 TStringList 오브젝트를 사용하도록 한다.
문자열을 첨가, 삭제, 삽입, 이동, 교환하는 여러 가지 메소드들을 제공하고 있다.

### ◘ 속성

#### Strings[Index: Integer]: string;

문자열 리스트에 저장된 문자열들을 가지는 문자열 배열이며 읽고자 하는 문자열 번호를 첨자로 주어 원하는 문자열을 얻을 수 있다. 단 문자열 번호는 0번이 제일 처음이다. 이 속성은 Strings 오브젝트의 디폴트 속성이므로 속성을 명시하지 않고 곧바로 오브젝트의 이름과 첨자를 사용하면 된다. 예를 들어 메모의 Lines 속성에 저장된 문자열을 읽으려면 Lines.Strings[n]과 같이 읽을 필요 없이 곧바로 Lines[n]과 같이 읽으면 된다.

#### 읽 Count: Integer;

문자열 리스트에 포함된 문자열의 개수를 조사한다.

#### Objects[Index: Integer]: TObject;

문자열 리스트의 각 문자열 항목에 연결된 오브젝트의 목록을 가지는 오브젝트 배열이다. 즉 각 문자열마다 하나씩 대응되는 오브젝트를 가질 수 있다는 얘기다. 이 속성을 사용하면 문자열과 비트맵을 결합하여 비트맵 리스트 박스를 만들 수 있다.

### ◘ 메소드

■ function Add(const S: string): Integer;
문자열 리스트에 새로운 문자열 항목을 더한다. 인수 S가 문자열 리스트에 추가될 문자열이며 문자열 추가후 추가된 문자열의 인덱스를 리턴한다. Sorted 속성이 설정되어 있으면 정렬된 위치로 삽입되며 그렇지 않으면 문자열 리스트의 끝에 추가된다.

■ procedure Delete(Index: Integer);
인덱스로 지정한 번호의 문자열을 문자열 리스트에서 삭제한다.

■ procedure Clear;
문자열 리스트의 모든 항목을 삭제한다.

■ procedure Insert(Index: Integer; const S: string);
Index 인수가 지정하는 위치에 문자열 S를 문자열 리스트에 삽입한다.

■ procedure Move(CurIndex, NewIndex: Integer);
CurIndex번째의 문자열을 NewIndex번째로 이동시킨다. 문자열이 연결된 오브젝트를 가질 경우 오브젝트도 문자열과 함께 이동된다.

■ procedure Exchange(Index1, Index2: Integer);
두 개의 문자열을 교체한다. 문자열이 연결된 오브젝트를 가질 경우 오브젝트도 문자열과 함께 교체된다.

■ function IndexOf(const S: string): Integer;
문자열 리스트에서 문자열 S를 찾아 그 인덱스를 리턴해 준다. 문자열이 발견되지 않을 경우 -1을 리턴한다.

■ procedure AddStrings(Strings: TStrings);
문자열 리스트에 다른 문자열 리스트를 추가한다. 이 메소드에 의해 두 개의 문자열 리스트가 결합된다.

# 3.속성 레퍼런스

컴포넌트의 가장 일반적인 속성들과 초보자가 가장 먼저 알아야 할 속성들만 모아 보았으며 알파벳 순으로 정렬되어 있다.

## Action:TBasicAction;

이 컴포넌트와 연결될 액션 컴포넌트를 지정한다. 액션은 사용자에 반응을 집중화하기 위한 장치이다. 실행중에 이 컴포넌트가 선택되면 연결된 액션의 OnExecute 이벤트가 발생하며 액션의 상태가 변경되면 이 컴포넌트의 Caption, Enabled, Visible 등의 속성이 같이 변경된다.

## Align:TAlign;

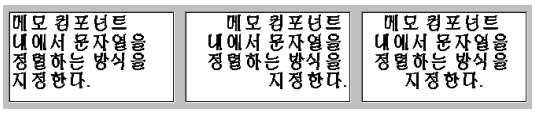
컨트롤이 폼 상에(또는 컨트롤을 포함하는 다른 컨트롤에) 정렬되는 방식을 지정한다. 디폴트값은 alNone이며 아무런 정렬도 수행하지 않으므로 배치한 상태대로 폼에 놓여진다. 가능한 정렬 형태는 다음과 같다.

| 속성 | 의미 |
|---|---|
| alNone | 정렬하지 않는다. 사용자가 지정한 위치에 컨트롤이 위치한다. 이 값이 디폴트값이다. |
| alTop | 컨트롤을 폼의 상단에 위치시키고 폼의 가로 폭에 맞게 크기가 조절된다. |
| alBottom | 컨트롤을 폼의 하단에 위치시키고 폼의 가로 폭에 맞게 크기가 조절된다. |
| alLeft | 컨트롤을 폼의 좌측에 위치시키고 폼의 세로 폭에 맞게 크기가 조절된다. |
| alRight | 컨트롤을 폼의 우측에 위치시키고 폼의 세로 폭에 맞게 크기가 조절된다. |
| alClient | 컨트롤이 폼 전체 크기와 같은 크기로 조정된다. |

일단 Align 속성이 주어지면 폼의 크기나 위치가 변해도 정렬된 컴포넌트는 폼 내에서의 절대적인 위치를 유지한다. 예를 들어 패널을 alTop으로 정렬시키면 폼의 크기가 변해도 항상 폼의 위쪽에 위치하며 폼의 가로 폭에 맞게 수평 크기가 재조정된다. 참고로 Alignment 속성은 컨트롤 내에서 텍스트의 정렬 방식을 지정한다.

## Alignment: TAlignment;

컨트롤 내에서 텍스트가 어떻게 정렬된 것인가를 지정한다. 다음은 Memo 컨트롤에서의 텍스트 정렬 예이며 좌측부터 좌측 정렬, 우측 정렬, 중앙 정렬이다.




가능한 정렬값에는 다음과 같은 것들이 있다.

| 속성 | 의미 |
|---|---|
| taLeftJustify | 텍스트를 컨트롤의 좌측에 정렬한다. |
| taCenter | 텍스트를 컨트롤의 중앙에 정렬한다. |
| taRightJustify | 텍스트를 컨트롤의 우측에 정렬한다. |

참고로 Align 속성은 컨트롤 그 자체의 정렬 방식을 나타낸다.

## Anchors:TAnchors;

컴포넌트와 부모의 위치 관계를 정의하는 4개의 세부 속성을 가지고 있다. 세부 속성이 True로 설정되면 부모의 크기가 변경되어도 항상 부모와 일정한 폭을 유지한다. Align 속성이 변경되면 이 속성도 같이 변경된다.

## AutoSelect:Boolean;

에디트 컨트롤이나 콤보 박스 컨트롤이 포커스를 가질 때 텍스트가 선택될 것인지 아닌지를 지정한다. 이 값이 True일 경우 에디트 컨트롤이나 콤보 박스 컨트롤 선택시 텍스트가 선택 및 반전되어 곧바로 다른 텍스트를 입력할 수 있도록 해주며 이 값이 False일 경우 기존의 텍스트를 수정할 수 있도록 해준다. 이 속성의 디폴트값은 True이다.

## Autosize:Boolean;

True일 경우 컴포넌트의 크기가 캡션(또는 그래픽)의 길이와 폰트의 크기에 따라 자동 조절된다. 즉 캡션 문자열의 길이가 커지면 레이블의 크기가 늘어나며 비트맵의 크기에 따라 이미지 컴포넌트의 크기가 자동으로 조절된다.

## BorderStyle

에디트, 리스트 박스, 메모 컨트롤 등이 경계선을 가질 것인지를 지정한다. 이 값이 bsNone이면 경계선이 없으며 bsSingle이면 경계선이 있다. 에디트 컨트롤의 경우 경계선이 없으면 AutoSize 속성을 True로 설정해 놓아도 효과를 발휘하지 못한다.

bsSingle

bsNone

폼의 경우는 다음 네 가지 값 중 하나의 값을 가지며 경계선의 유무에 따라 크기 조절이 가능해지기도 하고 불가능해지기도 한다.

| 속성값 | 의미 |
|---|---|
| bsDialog | 크기 조정이 불가능하며 대화상자 형태를 가진다. |
| bsSingle | 크기 조정이 불가능하며 선 하나로 된 경계선을 가진다. |
| bsNone | 경계선을 가지지 않으며 최소, 최대, 조절 메뉴도 가지지 않는다. 크기 조정은 물론 불가능하며 타이틀바도 없기 때문에 위치를 옮길 수도 없다. |
| bsSizeable | 크기 조정이 가능한 표준적인 경계선을 가진다. 이 속성이 디폴트이다. |

## 실 읽 BoundRect:TRect;

컨트롤을 둘러싸고 있는 사각형의 좌표를 조사하며 이 때 좌표의 기준은 컨트롤의 페어런트이다.
R:=Control.BoundRect; 명령으로 사각형의 경계를 조사하는 것은 다음 대입문과 동일하다.

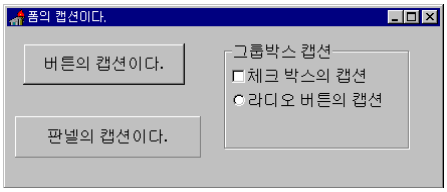
```
R.Left:=Control.Left;
R.Top:=Control.Top;
R.Right:=Control.Left+Control.Width;
R.Bottom:=Control.Top+Control.Height;
```

## Cancel:Boolean;

버튼이 Cancel 버튼인지 아닌지를 지정한다. Cancel 버튼으로 지정한 버튼은 키보드의 Esc 키를 누를 경우 이 버튼을 누른 것으로 간주된다. Cancel 버튼으로 지정된 버튼이 여러 개 있을 경우 탭 순서가 제일 먼저인 버튼이 눌러진 것으로 취급된다.

## Caption:String;

컨트롤의 제목을 나타내는 문자열이며 컨트롤에 따라 제목이 나타나는 부분이 다르다. 폼의 경우 윈도우의 타이틀 바나 최소화되었을 때의 아이콘 밑에 나타나며 버튼의 경우 버튼 위에 나타난다.




## CharCase:TEditCharCase;

에디트 박스에 입력되는 텍스트의 대소문자 형태를 지정한다. 다음 중 하나의 값을 가진다.

| 속성 | 의미 |
|---|---|
| ecLowerCase | 소문자로 출력된다. |
| ecNormal | 소문자와 대문자가 혼합되어 출력된다. |
| exUpperCase | 대문자로 출력된다. |

사용자가 입력한 문자가 이 속성과 다를 경우 입력된 문자는 이 속성이 지정한대로 변경되어 출력된다. 예를 들어 이 속성이 ecUpperCase일 때 사용자가 소문자를 입력해도 대문자로 나타난다. 암호 입력, 비밀 번호 입력 등과 같이 대소문자 구분을 확실하게 해야 하는 경우 이 속성을 사용한다.

## Color:TColor;

폼, 컨트롤의 배경 색상을 정의한다. ParentColor 속성이 True일 경우 컨트롤의 배경 색상은 페어런트의 색상으로 정의되며 페어런트의 색상을 바꾸면 컨트롤의 Color 속성도 따라서 바뀐다. 컨트롤의 Color 속성을 별도로 정의하면 ParentColor 속성은 자동으로 False가 된다. 다음은 Color 속성에 사용할 수 있는 색상이다. 색상을 말로 표현하기가 어려우므로 직접 색상을 바꾸어 가면 선택하기 바란다.

| 속성 | 색상 |
|---|---|
| clBlack | 검정색 |
| clMaroon | 짙은 갈색 |
| clGreen | 초록색 |
| clOlive | 짙은 녹색 |
| clNavy | 짙은 파랑색 |
| clPurple | 자주색 |
| clTeal | 푸른색 |
| clGray | 회색 |
| clSilver | 은색 |
| clRed | 빨간색 |
| clLime | 연두색 |
| clYellow | 노란색 |
| clBlue | 파란색 |
| clFuchsia | 분홍색 |
| clAqua | 하늘색 |
| clWhite | 흰색 |
| clBackground | 윈도우즈의 배경색 |
| clActiveCaption | 활성 윈도우의 타이틀 바 |
| clInactiveCaption | 비활성 윈도우의 타이틀 바 |
| clMenu | 메뉴의 배경색 |
| clWindow | 윈도우의 배경색 |
| clWindowFrame | 윈도우 경계 색상 |
| clMenuText | 메뉴의 텍스트 색상 |
| clWindowText | 윈도우 내의 텍스트 색상 |
| clCaptionText | 활성 윈도우의 타이틀 텍스트 |
| clAtciveBorder | 활성 윈도우의 경계 색상 |
| clInactiveBorder | 비활성 윈도우의 경계 색상 |
| clAppWorkSpace | 작업 영역의 색상 |
| clHighlight | 블럭으로 싸여진 텍스트의 색상 |
| clHighlightText | 버튼 색상 |
| clBtnShadow | 버튼 그림자 색상 |
| clGrayText | 흐려진 텍스트의 색상 |
| clBtnText | 버튼 위의 텍스트 색상 |

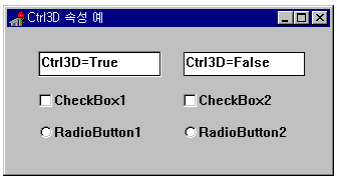
이 표에서 clBackground 이후의 색상은 윈도우즈가 정의한 시스템 색상이며 윈도우즈의 제어판에서 색상표를 어떻게 정의했는가에 따라 실제 색상이 달라진다. 위 표의 속성값을 사용하는 방법 외에도 RGB 등의 함수를 사용하여 트루 컬러를 표현하는 것도 가능하다.

## Constraints:TSizeConstraints;

컨트롤의 최소, 최대 크기를 지정하며 MaxHeight, MaxWidth, MinHeight, MinWidth 네 개의 세부 속성을 가진다. 디폴트로 이들 세부 속성은 모두 0의 값을 가지는데 이는 크기에 제한을 두지 않는다는 뜻이다. 만약 이 세부 속성에 원하는 크기값을 설정하면 사용자는 이 속성으로 지정한 크기 이상(또는 이하)로 컨트롤의 크기를 변경할 수 없다. 예를 들어 폼의 MaxHeight, MaxWidth를 480, 640으로 각각 지정하면 사용자는 폼을 640*480 이상의 크기로 확장하지 못한다.

## Ctl3d:Boolean;

컨트롤이 3차원의 입체 모양으로 출력될 것인가 2차원의 평면 모양으로 출력될 것인가를 지정한다. 디폴트값은 True이며 모든 컨트롤이 입체 모양으로 나타난다. 대화상자의 경우 대화상자의 Ctl3D 속성이 대화상자 내의 모든 컨트롤의 형태를 결정짓는다. ParentCtl3D 속성이 Ture일 경우 페어런트의 Ctl3D 속성에 따라 컨트롤의 Ctl3D 속성이 결정된다. 컨트롤의 Ctl3D 속성만 별도로 변경할 경우 ParentCtl3D 속성은 자동으로 False가 된다.




## Cursor:TCursor;

마우스 커서가 컨트롤 위로 이동할 때 출력되는 이미지 모양을 정의하며 다음과 같은 커서가 사용 가능하다.

| 속성 | 커서 |
|---|---|
| crDefault | |
| crArrow | |
| crCross | |
| crBeam | |
| crSize | |
| crSizeNESW | |
| crSizeNS | |
| crSizeNWSE | |
| crUpArrow | |
| crHourglass | |
| crDrag | |
| crNoDrop | |
| crHSplit | |
| crVsplit | |

사용자가 직접 커서 모양을 디자인하려면 이미지 에디터로 커서를 디자인한 후 Screen 오브젝트의 Cursors 배열에 등록하여 사용한다.

## Default:Boolean;

버튼이 Default 버튼인지 아닌지를 지정한다. Default 버튼으로 지정한 버튼은 키보드의 Enter 키를 누를 경우 이 버튼을 누른 것으로 간주한다. Default 버튼으로 지정된 버튼이 여러 개 있을 경우 탭 순서가 제일 먼저인 버튼이 눌러진 것으로 취급한다.

## DragCursor:TCursor;

컨트롤이 드래그될 때의 커서를 지정한다. 이 속성에 대입될 수 있는 값은 Cursor 속성과 완전히 동일하므로 Cursor 속성을 참조하기 바란다.

## DragKind:TDragKind;

컨트롤이 드래그 & 드롭에 사용될 것인가 아니면 드래그 & 도킹에 사용될 것인가를 지정한다. dkDrag, dkDock 두 가지 값 중 한 가지를 설정한다.

## DragMode:TDragMode;

컨트롤의 드래그 모드를 지정하며 자동(dmAutomatic), 수동(dmManual)이 있다. 자동으로 설정할 경우 사용자가 컨트롤을 누르기만 하면 드래그가 시작되며 수동으로 설정할 경우

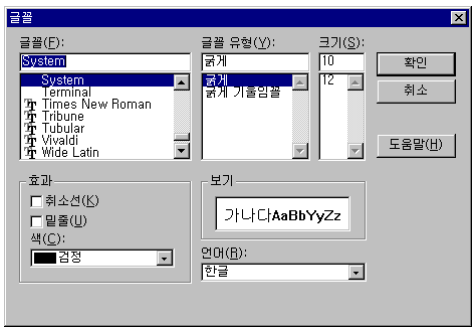
BeginDraw 메소드를 호출해 주어야 드래그가 시작된다.

## Enabled:Boolean;

컨트롤이 마우스나 키보드, 타이머 이벤트에 반응할 것인지 아닌지를 지정한다. True이면 정상적으로 동작하지만 False일 경우 컨트롤은 모든 이벤트를 무시하며 회색으로 표시된다. 특정한 버튼의 기능을 잠시 중단시킬 때 이 속성을 False로 바꾸어 둔다.

## Font:TFont;

컨트롤 상에 표시되는 문자의 폰트를 지정한다. 이 속성을 변경할 때 다음과 같은 폰트 변경 대화상자를 보여준다.




이 대화상자에서 폰트의 이름, 크기, 색상, 종류를 지정한다.

## 설읽Handle: HWND;

컨트롤의 윈도우 핸들을 조사한다. 버튼, 리스트 박스, 에디트 컨트롤 등도 모두 윈도우이며 각각의 컨트롤은 윈도우 핸들을 가진다. 델파이에서 이 속성을 사용하는 경우는 없으며 윈도우즈의 API 함수를 사용할 때 이 속성이 필요하다. 윈도우즈의 API 함수 중 윈도우 핸들을 인수로 요구하는 함수가 많기 때문이다.

## Height:Integer;

컨트롤의 수직 높이를 픽셀 단위로 설정한다. 직접 이 값을 변경하기보다는 주로 마우스로 크기 조절 핸들을 드래그하여 폼 상에서 직접 변경하는 것이 보통이다.

## HideSelection:Boolean;

에디트 컨트롤이나 메모 컨트롤에서 선택한 문장은 파란색의 역상으로 나타난다. 이 속성은 역상 표시가 포커스를 잃은 후에도 계속 유지될 것인지 아닌지를 지정한다. 이 속성이 True일 경우 컨트롤이 포커스를 잃으면 역상 표시가 사라지며 컨트롤이 다시 포커스를 가져야 역상 표시가 나타난다.

## Hint:string;

풍선 도움말에 들어갈 문자열을 입력하며 한글도 입력할 수 있다. 풍선 도움말을 사용하려면 이 속성에 도움말을 입력해 주고 ShowHint 속성을 True로 만들어 주어야 한다.

## ImeMode:TImeMode;

IME란 사용자의 입력을 받아들여 언어 설정에 맞게 바꾸어 주는 프로세스를 말한다. 이 속성값을 변경해 주면 소프트웨어적으로 한글/영문 입력 상태를 전환할 수 있다. 한글로 입력되도록 하려면 imSHanguel값을 대입해 주며 영문으로 입력되도록 하려면 imSAlpha값을 대입해 주면 된다. 그외 일본어나 중국어를 위한 값이 있기는 하지만 한글 윈도우즈에서는 사용할 수 없다.

## Left:Integer;

컨트롤의 좌상단 X 좌표를 설정한다. 이 좌표는 폼의 좌상단을 기준으로 하며 픽셀 단위를 사용한다. 폼의 경우 이 값은 전체 화면을 기준으로 하여 설정한다. 이 값을 직접 지정하는 것보다는 마우스로 컨트롤을 드래그하여 설정하는 것이 일반적이다.

## MaxLength:Integer;

에디트 컨트롤이나 메모 컨트롤에 입력할 수 있는 최대 문자수를 나타낸다. 디폴트값은 0이며 이는 입력 문자수에 제한이 없다는 뜻이다. 0 이외의 다른 값으로 지정하면 입력할 수 있는 문자수의 한계를 지정한다.

## Min,Max:Integer;

스크롤 바가 가지는 최소값과 최대값을 지정한다. 스크롤 바의 좌측값(수직일 경우 상단값)이 Min값이며 우측값(수직일 경우 하단값)이 Max이다. 이 두 값의 범위가 곧 스크롤 바의 범위가 된다.
Min의 디폴트값은 0이며 Max의 디폴트값은 100이다. 두 값은

스크롤 바가 나타내는 의미에 따라 적절하게 조절하도록 한다. 예를 들어 성적을 나타낸다면 0~100, 각도를 나타낸다면 0~360으로 범위를 설정한다. 스크롤 바 외에도 범위를 나타내는 UpDown 컴포넌트에서도 이 속성을 사용하여 범위를 지정한다.

## Modified:Boolean;

에디트 컨트롤이나 메모 컨트롤에 입력된 문장이 사용자에 의해 변경되었는지를 조사한다. 실행시에만 사용되며 이 값이 True이면 최초 컨트롤을 생성한 후, 또는 Modified가 False로 설정된 이후 텍스트가 변경되었다는 뜻이다.

## Name:TComponentName;

컴포넌트의 이름이며 이 이름은 다른 컴포넌트나 프로그램의 다른 부분에서 이 컴포넌트를 참조하기 위해 사용된다. 컴포넌트의 이름은 다른 컴포넌트와 중복되어서는 안되므로 고유의 이름을 가져야 한다. 델파이는 새로운 컨트롤이 생성될 때 컨트롤의 형태와 일련 번호를 사용하여 임시적인 이름을 지어준다. 예를 들어 버튼 컨트롤을 생성할 경우 BUTTON1, BUTTON2, ... 등과 같이 이름을 붙인다. 이 이름을 그대로 사용해도 큰 문제는 없지만 되도록이면 의미를 쉽게 알 수 있는 이름을 주는 것이 좋다.
"'시작" 버튼이라면 buttonstart, "취소" 버튼이라면 buttoncancel등과 같이 이름을 붙이도록 한다.

## 실 읽 Owner:TComponent;

컴포넌트를 소유한 컴포넌트를 명시한다. 소유주가 파괴되면 그 컴포넌트에 속한 모든 컴포넌트도 같이 파괴된다. 폼 상의 모든 컴포넌트는 폼이 소유하고 있으며 폼은 Application 컴포넌트가 소유한다. 그래서 프로그램이 종료되면 폼의 메모리가 해제되고 따라서 폼이 소유한 모든 컴포넌트의 메모리도 해제된다.

## 실 Parent:TWinControl;

컨트롤의 부모 컴포넌트를 명시한다. 부모 컴포넌트는 윈도우 컨트롤이며 내부에 다른 컴포넌트를 포함할 수 있다. 패널, 그룹박스 등이 페어런트가 되며 페어런트를 조작하면 자식 컨트롤도 같이 영향을 받는다. 즉 패널을 이동시키면 패널 내부에 놓여진 모든 컴포넌트도 같이 이동된다.

## ParentColor:Boolean;

각 컨트롤의 배경 색상을 어떻게 설정할 것인가를 지정한다. 이 속성이 True일 경우 컨트롤의 배경색은 페어런트의 Color 속성을 사용하며 False일 경우 컨트롤 자체의 Color 속성을 사용한다. 라디오 그룹, 레이블 등의 몇몇 컨트롤을 제외하고 대부분의 경우 이 속성은 False이다. 이 속성을 사용하면 폼의 색상에 따라 컨트롤의 색상을 일괄적으로 변경할 수 있다. 즉 폼의 색상이 바뀌면 컨트롤의 색상도 폼의 색상에 따라 바뀌어진다. 폼의 색상과 다른 색상을 사용할 경우 컨트롤의 Color 속성을 별도로 변경시키면 ParentColor 속성은 자동으로 False가 된다.

## PasswordChar:Char;

에디트 컨트롤에 문자를 입력받을 때 출력되는 문자의 형태를 지정한다. 디폴트로 이 속성은 0으로 설정되어 있으므로 입력된 문자가 그대로 출력된다. 이 속성에 특별한 다른 문자를 지정하면 입력되는 문자를 모두 이 문자로 바꾸어 출력한다. 암호를 입력할 때 이 속성을 사용한다.

## Popupmenu:TPopupMenu;

사용자가 컨트롤 위에 마우스 커서를 두고 오른쪽 버튼을 누를 경우 나타나는 팝업 메뉴를 지정한다.

## ScrollBars:TScrollStyle;

메모 컨트롤이나 그리드 컨트롤이 스크롤 바를 가질 것인지 아닌지를 지정하며 다음 중 하나의 값을 가진다.

| 속성 | 의미 |
|---|---|
| ssNone | 스크롤 바가 없다. |
| ssHorizontal | 오른쪽에 수직 스크롤 바를 만든다. |
| ssVertical | 아래쪽에 수평 스크롤 바를 만든다. |
| ssBoth | 수평, 수직 두 개의 스크롤 바를 만든다. |

## ShowAccelChar:Boolean;

레이블의 캡션 속성에 포함되어 있는 &기호의 해석 방식을 지정한다. 이 속성이 True일 경우 &기호는 단축키를 나타내며 밑줄이 그어진 형태로 나타나지만 이 속성이 False일 경우 &기호는 단순히 &문자로만 나타난다.

## Sorted:Boolean;

리스트 박스나 콤보 박스에서 아이템들을 알파벳순으로 정렬할 것인지 아닌지를 지정한다. 이 속성이 True이면 모든 아이템이 알파벳 순으로 정렬되며 삽입, 추가되는 아이템도 정렬되어 적당한 위치에 더해진다.

## Taborder:TTabOrder;

페어런트에 속한 컨트롤 사이를 이동할 때는 Tab 키를 사용한다. Tab 키를 누를 경우 포커스가 이동하는 순서를 탭 순서라고 하며 Taborder 속성은 컨트롤이 몇 번째 Tab을 누를 때 포커스를 가질 것인가를 지정한다. 초기값으로 폼 상에 생성된 순서대로 탭이 이동하되 사용자가 이 속성을 바꿈으로써 변경할 수 있다. 탭 순서는 하나의 폼에 있는 각각의 컨트롤에게 고유해야 하며 중복되어서는 안된다. 제일 먼저 생성된 컨트롤이 탭 순서 0을 가지며 그 다음 생성된 컨트롤이 탭 순서 1, 그리고 차례대로 2, 3, ...의 값을 가진다. 사용자가 탭 순서를 중복되게 바꾸면 델파이가 적절히 탭 순서를 조정해 준다. 예를 들어 여섯 개의 컨트롤이 있고 각각 0, 1, 2, 3, 4, 5의 탭 순서를 가질 때 5번 컨트롤의 탭 순서를 2로 바꿀 경우 2번 이후 컨트롤의 탭 순서는 하나씩 뒤로 밀려난다. 또 전체 컨트롤 수보다 더 큰 탭 순서를 지정하면 자동으로 끝 탭 순서로 맞추어 준다. 탭 순서가 0인 컨트롤이 최우선 순위를 가지며 폼이 최초 생성되어 나타날 때 포커스를 가진 컨트롤이 된다.

## Tabstop:Boolean;

사용자가 Tab 키를 누를 때 이 컨트롤이 포커스를 가질 수 있는지 없는지를 지정한다. Tabstop 속성이 False라면 탭 순서를 가지지 않게 되고 따라서 사용자는 Tab 키를 눌러 이 컨트롤로 이동할 수 없다.

## Tag:Longint;

컴포넌트에 별도로 정수값을 저장한다. 델파이에서는 이 정수값에 특별한 의미를 두지 않으며 사용하지도 않는다. 하지만 사용자가 자신의 목적에 따라 특별한 의미를 부여하여 사용할 수도 있다.

## Text:TCaption;

에디트 컨트롤이나 메모 컨트롤에 입력된 문장을 나타내며 실행중에 이 속성을 읽음으로써 사용자가 입력한 문자열을 얻을 수 있으며 또한 이 속성에 문자열을 대입하여 실행중에 에디트 박스의 내용을 바꿀 수도 있다.

## Top:Integer;

컨트롤 좌상단의 Y 좌표를 설정한다. 이 좌표는 폼의 좌상단을 기준으로 하며 픽셀 단위를 사용한다. 폼의 경우 이 값은 전체 화면을 기준으로 하여 설정한다. 이 값을 직접 지정하는 것보다는 마우스로 컨트롤을 드래그하여 설정하는 것이 일반적이다.

## Transparent:Boolean;

레이블의 배경 색상을 투명색으로 설정한다. 즉 레이블의 배경색상을 없게 만들어 레이블 뒤의 컴포넌트가 보일 수 있도록 지정한다. 비트맵이나 특별한 무늬 위에 레이블이 놓일 경우 이 속성을 True로 설정하여 비트맵이 보일 수 있도록 하는 것이 좋다.

## Visible:Boolean;

컨트롤이 화면에 보이도록 한다. 이 속성이 False일 경우 컨트롤은 화면에 보이지 않게 된다. 특정 컨트롤을 잠시 숨기고자 하는 목적으로 사용된다. 폼을 제외한 모든 컨트롤의 디폴트 속성은 True이며 만들어진 컨트롤은 모두 화면에 보여진다. 숨겨진 컨트롤을 보이게 하려면 Show 메소드를 호출하여 Visible 속성을 True로 만들어 주어야 한다.

## Width:Integer;

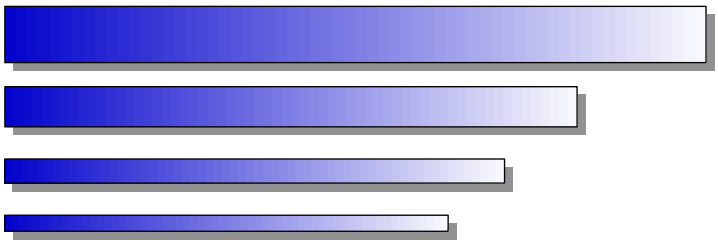
컨트롤의 수평 넓이를 픽셀 단위로 설정한다. 직접 이 값을 변경하기 보다는 주로 마우스로 크기 조절 핸들을 드래그하여 폼 상에서 직접 변경하는 것이 보통이다.

## Wordwrap:Boolean;

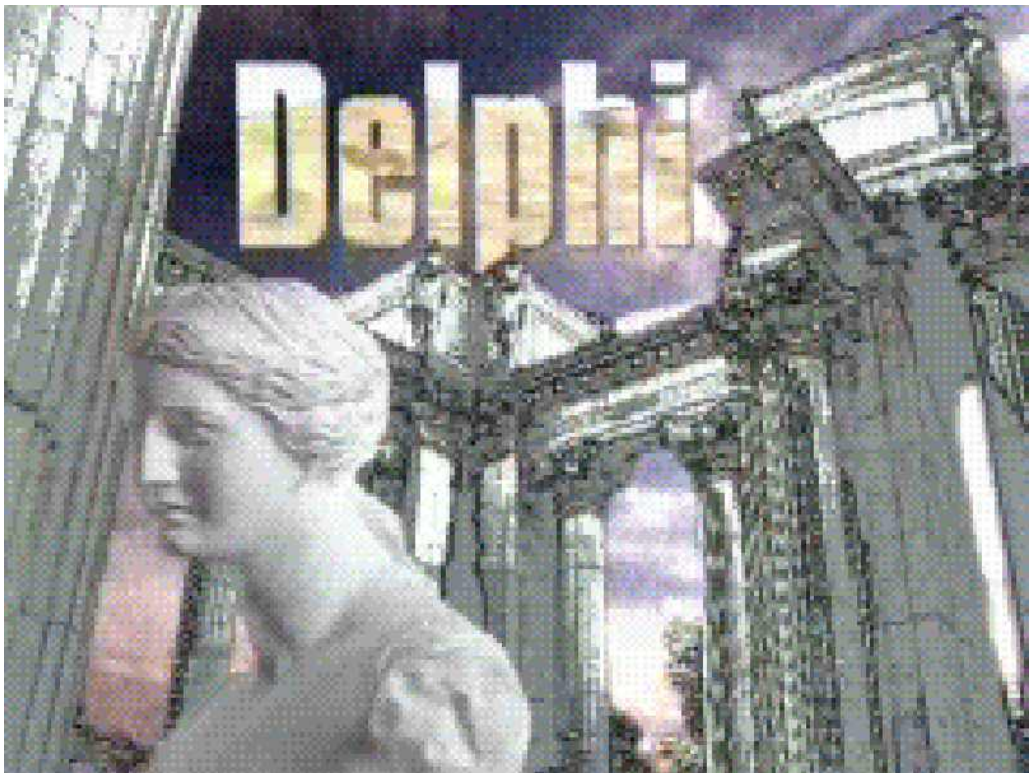
레이블이나 메모 컨트롤에 텍스트가 출력될 때 컨트롤의 다른 쪽 끝에서 단어를 잘라 다음 줄에 출력되도록 하여 텍스트가 컨트롤 밖으로 나가지 않도록 해준다. Wordwrap 기능을 사용할 경우 메모 컨트롤 밖으로 텍스트가 출력되지 않으므로 수평 스크롤 바는 불필요해진다. 수직으로 메모 컨트롤의 경계를 벗어난 경우 수직 스크롤 바를 설치(Scrollbars 속성)하여 이동하며 볼 수 있다. 메모 컨트롤의 경우 최소한 한 줄의 텍스트라도 출력할 수 있을 정도의 길이가 되어야 사용자가 텍스트를 편집할 수 있다. 디폴트값은 False 이며 Wordwrap 기능을 수행하지

않는다.

# 함수 레퍼런스



# 부록 3



델파이의 유닛에 포함되어 있는 개별 함수들에 관해 정리하였다. 이 함수들은 VCL 오브젝트의 메소드가 아니므로 선언된 유닛만 포함해 주면 언제든지 사용할 수 있다. 함수의 수가 워낙 많아 전부 다 정리하지는 못하였으며 내부적으로 사용되는 함수, 호환성을 위해 존재하는 함수, 동일 함수가 존재하는 함수 등은 레퍼런스에서 제외하였다. 그러나 기본적으로 사용되는 함수들에 관해서는 상세한 예를 들어 설명하고 있으므로 부족함이 없을 것이다. 함수 게재 순서는 알파벳순이며 함수명 우측에 함수가 정의되어 있는 유닛을 명시하였으므로 이 레퍼런스에서 부족한 면은 유닛을 직접 참조하기 바란다.

## Abort SysUtils Unit

**문법** : procedure Abort;

에러를 수정하지 않고 현재의 루틴을 종료한다. 에러 메시지도 출력하지 않는다.

## Abs system Unit

**문법** : function Abs(X);

X의 절대값을 구한다. 절대값이란 값은 같으며 부호가 양수인 수를 말한다. X는 정수 또는 실수값이다.

```
Abs(-3.14); { 결과는 3.14 }
Abs(3.14);  { 결과는 3.14 }
```

## AddExitProc system Unit

**문법** : procedure AddExitProc(Proc:TRpocedure);

종료 프로시저를 등록한다. 종료 프로시저란 런타임 라이브러리의 exit 프로시저에 등록되는 함수이며 프로그램이 끝날 때 자동으로 호출되는 함수를 말한다. 프로그램을 끝내기 전에 해야 할 특별한 일, 예를 들어 메모리 해제나 자원 해제 등의 일이 있다면 종료 프로시저에 코드를 작성하도록 한다. 프로그램이 종료될 때 AddExitProc로 등록된 역순으로 호출된다. 즉 처음 등록된 함수가 제일 끝에 호출된다.

## Addr System Unit

**문법** : function Addr(X):Pointer;

X의 번지를 구한다. X는 변수나 함수의 명칭이며 결과는 X의 메모리 번지를 나타내는 포인터 값이며 Pointer형이다. Addr 함수의 리턴값이 Pointer형이므로 Pointer형의 변수에 곧바로 대입될 수 있지만 이 함수의 리턴값으로 메모리를 직접 참조할 수는 없다. 메모리를 참조하기 위해서는 타입 캐스트를 해 주어야 한다.

```
var
 i:Integer;
begin
 P := Addr(i);  { i의 번지를 구한다. }
end.
```

## AdjustLineBreak SysUtils Unit

**문법** : function AdjustLineBreaks(const S: string): string;

주어진 문자열 S에 있는 모든 개행 코드를 CR/LF로 변경한다. CR만 단독으로 있는 문장, LF만 단독으로 있는 문장 또는 LF/CR로 되어 있는 코드가 모두 CR/LF로 변경된다. 유닉스의 텍스트를 PC형태로 바꾸고자 할 때 이 함수를 사용한다.

## AllocMem SysUtils Unit

**문법** : function AllocMem(Size:Cardinal):Pointer;

인수로 주어진 Size 바이트만큼 힙에서 메모리를 할당하며 할당된 메모리는 모두 0으로 초기화된다. 할당한 메모리를 해제하려면 FreeMem 함수를 사용한다.

## AnsiCompareStr SysUtils Unit

**문법** : function AnsiCompareStr(const S1,S2:string):Integer;

두 개의 문자열 S1과 S2를 비교하되 대소문자를 구분하여 비교한다. S1과 S2 문자열이 같으면 0을 리턴하며 S1<S2이면 음수의 값을 리턴하며 S1>S2이면 양수를 리턴한다. 문자열끼리의 비교는 문자열을 이루는 각 문자의 문자 코드를 비교하여 수행된다.

## AnsiCompareText SysUtils Unit

**문법** : function AnsiCompareStr(const S1,S2:string):Integer;

두 개의 문자열 S1과 S2를 비교하되 대소문자를 구분하지 않고

비교한다. S1과 S2 문자열이 같으면 0을 리턴하며 S1<S2이면 음수의 값을 리턴하며 S1>S2이면 양수를 리턴한다. 문자열끼리의 비교는 문자열을 이루는 각 문자의 문자 코드를 비교하여 수행된다.

## AnsiLowerCase SysUtils Unit

**문법** : function AnsiLowerCase(const S: string): string;

문자열 내부의 대문자를 모두 소문자로 바꾼다. 영문자 이외의 글자는 영향을 받지 않는다. 문자 변환은 현재 설치된 언어 드라이버의 영향을 받는다.

## AnsiUpperCase SysUtils Unit

**문법** : function AnsiUpperCase(const S: string): string;

문자열 내부의 소문자를 모두 대문자로 바꾼다. 영문자 이외의 글자는 영향을 받지 않는다. 문자 변환은 현재 설치된 언어 드라이버의 영향을 받는다.

## Append System Unit

**문법** : procedure Append(var f:Text);

파일을 추가 기록용으로 오픈한다. 파일 변수 F는 Assign으로 외부 파일과 연결되어 있어야 한다. 외부 파일이 없으면 에러가 발생하며 이미 열려 있는 경우는 파일을 닫은 후 다시 연다. 파일을 오픈한 후 FP는 파일 끝(EOF)에 위치시키므로 추가적인 데이터를 파일 끝에 추가할 수 있다. 다음 예는 루트 디렉토리의 AUTOEXEC.BAT 파일 끝에 PROMPT문을 추가한다.

```
var
  F: TextFile;
begin
  AssignFile(F, 'c:₩autoexec.bat');
    Append(F); { 파일 오픈 }
    Writeln(F, 'prompt $p$g');
    CloseFile(F);                { 파일 닫음 }
  end;
end.
```

## AppendStr SysUtils Unit

**문법** : procedure AppendStr(var Dest: string; const S: string);

두 개의 문자열을 결합한다. Dest 문자열 뒷부분에 Src 문자열이 추가된다. Dest:=Dest+S와 문법적으로 동일한 기능을 한다.

```
Src:='Apple';
Dest:='Orange';
AppendStr(Dest,Src);
```

Dest 문자열은 OrangeApple가 된다.

## ArcCos Math Unit

**문법** : function ArcCos(X: Extended): Extended;

X의 아크 코사인값을 구한다. 인수 X는 360분법의 각도가 아닌 호도값이며 리턴값은 0~Pi사이의 상수이다.

## ArcSin Math Unit

**문법** : function ArcSin(X: Extended): Extended;

X의 아크 사인값을 구한다. 인수 X는 360분법의 각도가 아닌 호도값이며 리턴값은 -Pi/2~Pi/2사이의 상수이다.

## ArcTan system Unit

**문법** : function ArcTan(X:Real):Real;

삼각함수 아크 탄젠트를 구한다. 인수 X는 360분법의 각도가 아닌 호도값이다.

## Assert Sysstem Unit

**문법** : procedure Assert(expr : Boolean [; const msg: string]);

주어진 진위문을 평가하는 디버깅 프로시저이다. 주어진 문장이 참이 아닐 경우 EAssertionFailed 예외를 발생시키며 인수로 주어진 문자열 메시지와 예외를 발생한 파일명, 예외가 발생한 줄번호를 같이 출력한다.

## Assigned system Unit

**문법** : procedure Assigned(var P):Boolean;

포인터 변수가 값을 가지고 있는지 가지고 있지 않은지(즉 nil)를 검사해 본다. P◇nil 과 문법적으로 동일한 조건식이며 P가 nil이면 False를 리턴하며 nil이 아니면 True를 리턴한다. 이 프로시저는 이벤트 핸들러 호출 전에 이벤트가 이벤트 메소드를 가지고 있는지 아닌지를 점검할 때 사용된다.

## AssignFile system Unit

**문법** : procedure AssignFile(var f,String);

텍스트 파일 변수와 외부 파일을 연결한다. 즉 파일 핸들에 파일을 할당한다. 일단 파일과 핸들이 연결되면 핸들에 대한 모든 조

작은 파일로 입출력된다. 이 프로시저로 핸들에 파일을 할당한 후 Reset이나 Rewrite 등을 사용하여 파일을 오픈하며 Writeln이나 Readln으로 파일 입출력을 수행한다. 파일 이름은 문자열 형태로 전달하되 필요할 경우 드라이브명과 디렉토리명을 사용할 수 있다. 주의할 것은 이미 열려져 있는 파일 핸들을 재할당해서는 안된다는 점이다. 다음 예는 test.txt 파일에서 첫 줄을 읽어 에디트 박스로 출력한다.

```
var
  F: TextFile;
  S: String;
begin
  AssignFile(F, 'test.txt');
  Reset(F);
  Readln(F, S);
  Edit1.Text := S;
  CloseFile(F);
end.
```

## AssignPrn Printers Unit

**문법** : procedure AssignPrn(var F: Text);

텍스트 파일 변수 F를 프린터로 할당한다. 프린터는 쓰기 전용의 텍스트 파일이므로 이 프로시저로 할당을 한 후 Rewrite 프로시저로 파일을 오픈해야 한다. 일단 파일 변수가 프린터로 할당되고 오픈되면 이 파일 변수로 보내는 모든 출력은 프린터로 인쇄된다. 파일로 문자열을 출력하는 Writeln 프로시저로 문자열을 프린터로 전송한다.

## AssignStr system Unit

**문법** : procedure AssignStr(var P: PString; const S: String);

동적으로 할당된 문자열에 새로운 문자열을 대입한다. AssignStr(P,S)는 DisposeStr(P); P:=NewStr(S)와 동일하다. AssignStr은 문자열 포인터를 초기화하지 않으므로 문자열 포인터 P는 nil이거나 아니면 유효한 문자열 포인터이어야 한다.

```
var
  P: PString;
begin
  P := NewStr('First string');  { 문자열 할당 }
  AssignStr(P, 'Second string'); { 새문자열 할당 }
  DisposeStr(P);   { 할당 취소 }
end.
```

## Beep SysUtils Unit

**문법** : procedure Beep;

MessageBeep API 함수를 호출하여 짧은 비프음을 낸다. 사용자에게 단순한 경고를 주거나 할 때 사용하며 디버깅 용도로도 사용된다.

## BlockRead system Unit

**문법** : procedure BlockRead(var f:File; var Buf; Count:Word [;var Result: Word]);

파일로부터 한 개 또는 다수 개의 레코드를 Buf 메모리 영역으로 읽어들인다. F는 언타입드 파일 변수이며 Buf는 어떤 형의 변수든지 상관없다. 읽혀지는 레코드의 개수는 Count인수로 설정하며 실제로 읽혀진 레코드 수는 Result 인수로 리턴된다. 파일로부터 읽혀진 레코드는 Buf 변수가 차지하고 있는 메모리의 시작 위치에 기록된다. 리턴값인 Result가 Count 인수와 같은 값을 가지면 요구한 레코드가 전부 읽혀진 것이며 Result가 Count보다 더 작은 값을 가지면 파일을 읽는 중에 EOF를 만난 것이다.

## BlockWrite system Unit

**문법** : procedure BlockWrite(var f:File; var Buf; Count:Word [;var Result: Word]);

파일로 한 개 또는 다수 개의 레코드를 출력한다. F는 언타입드 파일 변수이며 Buf는 어떤 형의 변수든지 상관없다. 파일로 출력되는 레코드의 개수는 Count 인수로 설정하며 실제로 출력된 레코드 수는 Result 인수로 리턴된다. 파일로 출력되는 레코드는 Buf 변수가 차지하고 있는 메모리의 시작 위치에 있어야 한다. 리턴값인 Reulst가 Count 인수와 같은 값을 가지면 요구한 레코드가 이상없이 출력된 것이며 Result가 Count보다 더 작은 값을 가지면 디스크의 용량이 부족하여 레코드를 모두 출력하지 못한 것이다.

## Bounds Classes Unit

**문법** : function Bounds(ALeft, ATop, AWidth, AHeight: Integer): TRect;

주어진 좌상단 좌표와 높이, 넓이를 사용하여 TRect형의 사각 영역을 구한다. Rect는 좌상단, 우하단 두 점의 좌표로 사각 영역을 구하는 반면 이 함수는 좌상단과 크기를 사용하여 사각 영역을 구한다는 점이 다르다.

```
R := Bounds(X, Y, W, H);
```

즉 위와 같이 사각 영역을 구하는 식은 다음과 동일한 결과를 가

져온다.

```
R := Rect(X, Y, X + W, Y + H);
```

## Break

system Unit

**문법** : procedure Break;

for, while, repeat 등의 반복문의 중간에서 사용되며 루프를 끝내고 루프 다음의 명령으로 프로그램의 흐름을 옮기도록 한다. 만약 반복 루프의 외부에서 이 프로시저가 사용되면 에러이다. 다음 예는 While문으로 무한 루프를 구성하고 문자열을 입력받아 다시 출력하기를 무한 반복하되 아무 문자도 입력되지 않으면 루프를 종료한다.

```
var
  S: String;
begin
  while True do {무한 루프}
  begin
    ReadLn(S);
    if S = '' then Break; {빈 행이면 루프 종료}
    WriteLn(S);
  end;
end.
```

이 루프는 다음과 같이 작성해도 동일하다.

```
var
  S: String;
begin
  S:='temp'
  while S<>'' do
  begin
    ReadLn(S);
    WriteLn(S);
  end;
end.
```

S가 널 스트링일 때까지 반복을 하도록 변경하였다. 루프 시작 시에 조건 점검을 하므로 루프 진입 전에 S가 널이 아니도록 아무 문자열이나 대입해 주어야 한다. 이렇게 하지 않으려면 선 실행 후 평가문인 repeat를 사용하면 된다.

## Ceil

SysUtils Unit

**문법** : function Ceil(X: Extended):Integer;

실수를 정수로 만들되 주어진 실수보다 큰 정수중 가장 작은 값을 리턴한다. 양수일 경우는 단순한 올림이지만 음수일 경우는 올림한 값에서 1 감소한 값이 된다.

```
Ceil(3.14) = 4
Cedil(-3.14) = -3
```

## ChangeFileExt

system Unit

**문법** : function ChangeFileExt(const FileName, Extension: string):string;

첫 번째 인수 FileName으로 주어진 파일의 이름 중 파일의 확장자만 두 번째 인수 Extension으로 변경한다. 다음 예는 인수로 주어진 파일의 확장자를 무조건 INI로 변경한다.

```
ChangeFileExt(ParamStr(0), '.INI');
```

## Chdir

system Unit

**문법** : procedure ChDir(S: String);

인수 S가 지정하는 경로로 현재 디렉토리를 변경한다. 만약 S가 드라이브 문자를 포함하고 있을 경우 현재 드라이브도 변경된다.

## Chr

system Unit

**문법** : function Chr(X:Byte):Char;

서수값 X가 지정하는 문자를 구한다. 아스키 코드값을 X로 전달하면 해당하는 문자를 얻을 수 있다. Chr(65)는 알파벳 문자 A이다. 이 함수를 사용하면 키보드로 직접 입력할 수 없는 문자를 입력할 수 있다.

## Close

system Unit

**문법** : procedure Close(var F);

파일 핸들과 외부 파일과의 연결을 끊고 파일 핸들을 닫는다. 이 함수는 구 버전의 볼랜드 파스칼 컴파일러와의 호환성을 위해 제공되며 델파이에서는 사용되지 않는다. 델파이에서는 핸들을 닫는 CloseFile이라는 별도의 함수를 제공하므로 이 함수를 사용하도록 한다.

## CloseFile

system Unit

**문법** : procedure CloseFile(var F);

파일 핸들과 외부 파일과의 연결을 끊고 파일 핸들을 닫는다. 파일 변수 F는 Reset, Rewrite, Append 등에 의하여 오픈된 파일 핸들이며 파일 핸들은 사용 후에 반드시 닫아주어야 한다.

## ColorToIdent Graphics Unit

**문법** : function ColorToIdent(Color: Longint; var Ident: string): Boolean;

32비트로 표현된 Color 색상값을 파스칼이 사용하는 컬러 명칭 문자열로 변경한다. Color에 대응되는 명칭이 있으면 Ident 문자열에 색상의 명칭을 대입하고 True를 리턴하지만 대응하는 명칭이 없을 경우 Ident는 변경되지 않으며 False를 리턴한다. 다음 예는 $ffffff라는 색상값에 대응하는 색상 명칭을 찾는다. 색상값이 흰색이므로 레이블로 출력되는 문자열은 clWhite가 된다.

```
var
  ID: string;
  ans:Boolean;
begin
ans:=ColorToIdent($ffffff,ID);
label1.caption:=ID;
end;
```

## ColorToRGB Graphics Unit

**문법** : function (Color: TColor): Longint;

델파이가 사용하는 TColor형의 색상값을 32비트의 RGB형태로 변경한다. 색상값이 시스템에 정의된 색상일 경우 어떤 색상으로 정의되어 있는지 RGB값을 조사할 수 있다. 다음 예는 폼의 색상을 RGB로 조사한다.

```
var
  L : Longint;
begin
  L := ColorToRGB(Form1.Color);
end;
```

## ColorToString Graphics Unit

**문법** : function (Color: TColor): string;

TColor형의 값을 문자열로 변경한다. 다음 예는 폼의 색상을 문자열로 변경한 후 레이블로 출력한다. 디폴트 색상을 사용하고 있다면 레이블로 출력되는 값은 clBtnFace일 것이다.

```
var
  ID: string;
begin
ID:=ColorToString(Form1.Color);
label1.caption:=ID;
end;
```

## CompareStr system Unit

**문법** : function CompareStr(const S1, S2: string): Integer;

두 개의 문자열 S1과 S2를 비교하되 대소문자를 구분하여 비교한다. S1과 S2 문자열이 같으면 0을 리턴하며 S1<S2이면 음수의 값을 리턴하며 S1>S2이면 양수를 리턴한다. 문자열끼리의 비교는 문자열을 이루는 각 문자의 문자 코드를 비교하여 수행되며 현재 설치된 언어 드라이버에는 영향을 받지 않는다.

## CompareText system Unit

**문법** : function CompareText(const S1, S2: string): Integer;

두 개의 문자열 S1과 S2를 비교하되 대소문자를 구분하지 않고 비교한다. S1과 S2 문자열이 같으면 0을 리턴하며 S1<S2이면 음수의 값을 리턴하며 S1>S2이면 양수를 리턴한다. 문자열끼리의 비교는 문자열을 이루는 각 문자의 문자 코드를 비교하여 수행된다. "Apple"와 "APPLE"를 CompareText 함수로 비교하면 결과는 0이며 두 문자열이 같은 것으로 비교되지만 CompareStr 함수로 비교하면 결과는 두 문자열이 서로 다른 것으로 비교된다.

## Concat system Unit

**문법** : function Concat(s1 [, s2,..., sn]: String): String;

두 개 이상의 문자열들을 결합하여 새로운 긴 문자열을 만든다. 만약 문자열을 연결한 결과가 255문자 이상일 경우 255번째 이후의 문자는 잘려 나간다. 이 함수 대신 +연산자를 사용하여 문자열을 결합해도 결과는 동일하다. Concat('Korea','China')와 'Korea'+'China'는 같은 결과를 만들어 낸다. 다음 예는 Edit1에 입력된 문자열과 Edit2에 입력된 문자열을 합해 Edit3에 대입한다.

```
Edit3.Text:=ConCat(Edit1.Text,Edit2.Text);
```

## Continue system Unit

**문법** : procedure Continue;

for, while, repeat 등의 반복문에서 사용되며 다음 반복 부분으로 흐름을 옮기도록 한다. 루프의 남은 뒷부분은 무시되며 루프의 조건 점검부로 점프한다. 만약 반복 루프 외부에서 이 프로시저가 사용되면 에러이다. 다음 예는 i가 1~100까지 증가하며 모종의 처리를 하되 i가 10인 경우만 특별히 처리를 생략한다.

```
for i:=1 to 100 do
begin
```

```
 if i=10 then continue;
 ....
end;
```

## Copy

system Unit

**문법** : procedure Copy(S:String; Index, Count:Integer):String;

한 문자열의 부분 문자열을 추출해 낸다. S 문자열의 Index 위치에서부터 Count 문자분의 부분 문자열이 추출된다. Index가 문자열의 전체 길이보다 길 경우 빈 문자열을 리턴하며 Count가 문자열의 남은 부분보다 클 경우 문자열의 끝까지 추출해 낸다. Dest 문자열이 'Orange'일 경우 Copy(Dest,2,3)은 Dest 문자열의 두 번째 문자에서부터 3문자분의 부분 문자열인 'ran'을 추출해낸다.

## Cos

system Unit

**문법** : function Cos(X:Real):Real;

코사인 삼각함수값을 구한다. 여기서 사용되는 인수 X는 360분법의 각도가 아닌 호도값이다.

## CreateDir

SysUtils Unit

**문법** : function CreateDir(const Dir: string): Boolean;

인수로 주어진 문자열 경로에 새 디렉토리를 만든다. 디렉토리 생성에 성공하면 True를 리턴하며 실패하면 False를 리턴한다.

## Date

system Unit

**문법** : fun;

시스템의 시계를 참조하여 오늘 날짜를 구한다. 날짜는 TDateTime형이므로 곧바로 문자열로 출력할 수 없으며 DateToStr 함수를 사용하여 문자열로 바꾸어 주어야 한다. Label1.Caption:=DateToStr(Date) 명령에 의해 Label1에 오늘 날짜가 출력된다.

## DateTimeToFileDate

SysUtils Unit

**문법** : function DateTimeToFileDate(DateTime: TDateTime): Longint;

TDateTime형의 날짜, 시간 값을 DOS 형식의 날짜, 시간 형식으로 바꾼다. 도스 형식의 날짜는 4바이트의 정수형이며 각 비트에 날짜, 시간 요소를 포함하고 있다. 다음 예는 도스 형식의 날짜, 시간 형식을 레이블로 출력한다.

```
label1.caption:=IntToStr(DateTimeToFileDate(Now));
```

## DateTimeToStr

system Unit

**문법** : function DateTimeToStr(DateTime: TDateTime): String;

날짜와 시간을 담은 TDateTime형의 변수를 문자열로 바꾼다. 만약 DateTime 인수가 날짜를 포함하지 않으면 날짜는 00/00/00이 되며 시간을 포함하지 않으면 00:00:00 AM이 된다. 다음 예는 현재 시간과 날짜를 레이블로 출력한다.

```
Label1.Caption := DateTimeToStr(Now);
```

## DateTimeToString

system Unit

**문법** : procedure DateTimeToString(var Result: string; const Format: string; DateTime: TDateTime);

날짜, 시간을 담고 있는 DateTime을 Format에 주어진 형식대로 문자열로 변환하여 Result 문자열에 기입한다. Format 인수에 사용할 수 있는 지정자에 관한 사항은 FormatDateTime 함수를 참조하기 바란다.

## DateToStr

system Unit

**문법** : function DateToStr(Date: TDateTime): String;

날짜를 담는 TDateTime형의 변수에서 날짜를 문자열로 바꾼다. 다음 예는 오늘 날짜를 레이블로 출력한다.

```
Label1.Caption := DateToStr(Date);
```

## DayOfWeek

system Unit

**문법** : function DayOfWeek(Date: TDateTime): Integer;

특정 날짜의 요일을 계산한다. 리턴되는 값은 1~7까지의 정수이며 1이 일요일, 7이 토요일이다. 리턴되는 값이 정수형이므로 월, 화, 수, 목 등의 실제 요일 이름으로 바꾸어 주어야 한다. 다음 예는 오늘이 무슨 요일인지 조사해서 요일 이름을 레이블로 출력해준다.

```
var
  YO:string;
  ONUL:TDateTime;
begin
  ONUL:=Now;
  case DayOfWeek(ONUL) of
    1:YO:='일';
    2:YO:='월';
```

```
    3:YO:='화';
    4:YO:='수';
    5:YO:='목';
    6:YO:='금';
    7:YO:='토';
  end;
  label1.caption:='오늘은 '+YO+'요일입니다.';
end;
```

## Dec system Unit

**문법** : procedure Dec(var X[; N:Longint]);

변수의 값을 1 감소시킨다. Dec(X)는 X:=X-1과 동일한 동작을 한다. 두 번째 인수를 사용하면 1 이상의 값을 감소시킬 수도 있다. Dec(X,5)는 X:=X-5와 동일한 동작을 한다. 첫 번째 인수는 서수형의 변수이며 확장 문법이 적용될 경우 PChar형도 가능하다. N은 정수형의 상수 또는 변수여야 한다. Dec 함수는 가장 최적화된 코드를 생성해 내므로 루프 내부에서 제어 변수값을 감소시키는 용도로 사용하기에 적합하다.

## DecodeDate system Unit

**문법** : procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);

날짜를 담는 TDateTime형의 변수에서 년, 월, 일의 값을 분리시킨다. 분리된 값들은 각각 Year, Month, Day 등의 정수형 변수에 대입된다. 날짜값은 DateToStr 함수로, 문자열로 바꾼 후 한꺼번에 출력할 수 있지만 개별적인 요소를 가공한 후 출력하고자 할 경우는 이 함수를 사용한다. 이 함수의 반대 함수는 EncodeDate 함수이다. 다음 예는 오늘 날짜와 현재 시간을 조사한 후 문자열 조립을 통해 시간과 날짜를 알려 준다.

```
var
  Present: TDateTime;
  Year, Month, Day, Hour, Min, Sec, MSec: Word;
 begin
  Present:= Now;
  DecodeDate(Present, Year, Month, Day);
  Label1.Caption := '오늘은' + IntToStr(Year) + '년'+
IntToStr(Month) + ' 월' + IntToStr(Day)+'일입니다.';
  DecodeTime(Present, Hour, Min, Sec, MSec);
  Label2.Caption := '지금은' + IntToStr(Hour) + '시'+
IntToStr(Min)+'분입니다.';
end;
```

## DecodeTime system Unit

**문법** : fun;

시간을 담는 TDateTime형의 변수에서 시, 분, 초의 값을 분리시킨다. 분리된 값들은 각각 Hour, Min. Sec, MSec 등의 정수형 변수에 대입된다. 시간값은 TimeToStr 함수로 문자열로 바꾼 후 한꺼번에 출력할 수 있지만 개별적인 요소를 가공한 후 출력하고자 할 경우는 이 함수를 사용한다. 이 함수의 반대 함수는 EncodeTime 함수이다. DecodeDate 함수의 예제를 참조하기 바란다.

## Delete system Unit

**문법** : procedure Delete (var S:String; Index, Count:Integer);

한 문자열에서 부분 문자열을 삭제한다. S 문자열의 Index 위치에서부터 Count 문자분의 부분 문자열이 삭제된다. Index가 문자열의 전체 길이보다 길 경우 삭제는 이루어지지 않으며 Count가 문자열의 남은 부분보다 클 경우 문자열의 끝까지 삭제한다. Dest 문자열이 'Orange'일 경우 Delete(Dest,2,3)은 Dest 문자열의 두 번째 문자에서부터 3문자분의 부분 문자열인 'ran'을 삭제하며 Dest 문자열은 'Oge'가 된다.

## DeleteFile SysUtils Unit

**문법** : function DeleteFile(const FileName: string): Boolean;

디스크 상의 파일을 지운다. 파일이 없거나 읽기 전용 속성을 가질 경우 False를 리턴하며 파일은 지워지지 않는다. 그러나 파일을 지울 수 없는 경우라도 예외는 발생하지 않는다. 파일은 경로를 포함할 수 있다. 다음 예는 C 드라이브의 루트에 있는 COMMAND.COM 파일을 지운다. 물론 이 명령을 실행한 후부터 도스는 부팅되지 않는다.

```
DeleteFile('c:₩command.com');
```

## DirectoryExists FileCtrl Unit

**문법** : function DirectoryExists(Name: string): Boolean;

인수로 전달된 디렉토리가 존재하는지 검사하며 존재할 경우 True를 리턴하고 존재하지 않을 경우 False를 리턴한다.

## DiskFree SysUtils Unit

**문법** : function DiskFree(Drive: Byte): Longint;

인수로 지정한 Drive에 사용 가능한 용량을 조사해 준다. 드라

이브 번호는 0이 현재 드라이브, 1이 A, 2가 B 등이다. 드라이브 번호가 잘못되었을 경우는 -1을 리턴한다.

## DiskSize SysUtils Unit

**문법** : function DiskSize(Drive: Byte): Longint;

인수로 지정한 Drive의 총 용량을 조사해 준다. 드라이브 번호는 0이 현재 드라이브, 1이 A, 2가 B 등이다. 드라이브 번호가 잘못되었을 경우는 -1을 리턴한다.

## Dispose system Unit

**문법** : procedure Dispose(var P:Pointer);

동적으로 할당된 포인터의 메모리를 할당 해제한다. 할당이 해제된 포인터 값은 정의되어 있지 않으며 이 포인터를 사용하여 메모리를 참조할 수 없다. 다음 예제는 문자열 변수를 동적으로 생성하여 사용한 후 해제한다.

```
var
  P: ^string;
begin
  New(P);
  P^ := 'Now you see it...';
  Dispose(P);
end.
```

## DisposeStr SysUtils Unit

**문법** : fun;

NewStr 함수에 의해 힙에 할당한 메모리를 해제한다. 주어진 문자열 포인터가 nil이거나 빈 문자열이면 아무 동작도 하지 않는다.

## EncodeDate SysUtils Unit

**문법** : function EncodeDate(Year, Month, Day: Word): TDateTime;

정수형으로 주어진 년, 월, 일의 값을 사용하여 날짜를 담은 TDateTime형의 변수 하나를 만든다. Year는 1~9999까지의 값을 가지며 Month는 1~12, Day는 1~31까지 가능하다. 단 Day 값의 범위는 Month 값의 영향을 받으며 또는 Year가 윤년인 경우의 영향도 받는다. 만약 날짜값이 무효일 경우, 예를 들어 2월 30일 등의 값이 주어지면 EConvertError 예외가 발생한다. 년, 월, 일을 개별적으로 입력받아 하나의 날짜값을 만들고자 할 때 이 함수를 사용한다.

## EncdoeTime SysUtils Unit

**문법** : function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;

정수형으로 주어진 시, 분, 초의 값을 사용하여 시간을 담은 TDateTime형의 변수 하나를 만든다. Hour는 0~12까지 가능하며 만약 Time24Hour가 True일 경우 0~23까지의 24시간을 표현할 수 있다. Min, Sec은 0~59까지 가능하며 MSec은 0~999까지 가능하다. 시간값이 무효일 경우, 예를 들어 8시 86분 등의 값이 주어지면 EConvertError 예외가 발생한다. 시, 분, 초를 개별적으로 입력받아 하나의 시간값을 만들고자 할 때 이 함수를 사용한다.

## Eof system Unit

**문법** : function Eof [ (var F:Text) ]:Boolean;

파일 변수 F의 현재 위치(FP)가 파일의 끝일 경우 True를 리턴한다. 다음 예제는 input.dat 파일을 output.dat 파일로 복사한다. 두 파일을 열어놓고 입력 파일에서 읽어 출력 파일로 쓰기를 입력 파일의 끝까지 반복한다. 이 때 입력 파일의 끝인가를 점검하기 위해 Eof 함수가 사용되었다. 파일의 처음부터 끝까지 한 바이트씩 순서대로 작업을 해야 할 경우에 사용한다.

```
var
  F1, F2: TextFile;
  Ch: Char;
begin
  AssignFile(F1, 'input.dat');
  Reset(F1);
  AssignFile(F2, 'output.dat');
  Rewrite(F2);
  while not Eof(F1) do
    begin
    Read(F1, Ch);
    Write(F2, Ch);
    end;
  CloseFile(F2);
  CloseFile(F1);
end.
```

## Eoln system Unit

**문법** : function Eoln [ {var F:Text) ]:Boolean;

파일 변수 F의 현재 위치(FP)가 한 줄의 끝인지를 점검하고 줄 끝일 경우 True를 리턴한다. 한 줄씩 처리하고자 할 때 이 함수를 사용한다.

## Erase system Unit

**문법** : procedure Erase(var F);

외부 파일을 삭제한다. DeleteFile은 파일명으로 삭제하는 반면 이 프로시저는 파일 핸들로 파일을 삭제한다. 파일을 삭제한 후 파일 핸들은 자동으로 닫혀진다.

## Exclude system Unit

**문법** : procedure Exclude(var S:Set of T;I:T);

집합형 변수 S로부터 요소 I를 삭제한다. 이 프로시저는 S:=S-(I);와 동일한 동작을 하지만 Exclude가 더 효율적인 코드를 만든다.

## Exit system Unit

**문법** : procedure Exit;

현재 블럭을 종료하며 현재 블럭이 main 블럭이면 프로그램을 종료한다. 중첩 루프 내에서 Exit 프로시저는 현재 블럭의 바로 다음 블럭으로 제어를 옮기도록 한다. 함수나 프로시저의 내부에서 Exit는 함수 호출문 바로 다음 문장을 수행하도록 한다.

## Exp system Unit

**문법** : function Exp(X:Real):Real;

자연대수 exp를 계산한다. 이는 수학적으로 e의 x승을 의미하며 e는 2.718281...의 무리수이다.

## ExpandFileName SysUtils Unit

**문법** : function ExpandFileName(const FileName: string): string;

파일 이름에 드라이브명과 디렉토리명을 붙여 완전 경로(full path)를 만들어낸다. 덧붙여지는 경로는 현재 드라이브와 현재 디렉토리이다.

## ExtractFileExt SysUtils Unit

**문법** : function ExtractFileExt(const FileName string): string;

완전 경로에서 파일의 확장자만 분리시킨다. 특정 파일의 확장자를 알고자 할 때 이 함수를 사용하여 확장자만 분리한 후 문자열 비교를 수행하면 된다.

## ExtractFileName SysUtils Unit

**문법** : function ExtractFileName(const FileName: string): string;

완전 경로에서 파일 이름만 분리해 낸다. 파일의 확장자도 같이 분리된다. 즉 완전 경로에서 드라이브명과 디렉토리명이 제외된다.

## ExtractFilePath SysUtils Unit

**문법** : function ExtractFilePath(const FileName: string): string;

완전 경로에서 파일 이름을 제외하고 드라이브명과 디렉토리명만 분리해 낸다. 다음 예는 완전 경로를 가진 FileName으로부터 경로만 분리해낸 후 파일이 있는 디렉토리로 이동한다.

```
ChDir(ExtractFilePath(FileName));
```

## FindControl Controls Unit

**문법** : function FindControl(Handle: HWnd): TWinControl;

윈도우 핸들에 대응되는 윈도우 컨트롤을 찾아준다. 핸들에 대응되는 윈도우 컨트롤이 없거나 핸들이 0이면 nil을 리턴한다.

## FileAge SysUtils Unit

**문법** : function FileAge(const FileName: string): Longint;

파일의 날짜와 시간을 구해낸다.

## FileClose SysUtils Unit

**문법** : procedure FileClose(Handle: Integer);

파일 핸들을 닫는다. 파일 입출력을 위해 파일 변수에 외부 파일을 연결한 경우 핸들을 다 사용한 후에 반드시 파일 핸들을 닫아주어야 한다.

## FileDateToDateTime SysUtils Unit

**문법** : function FileDateToDateTime(FileDate: Longint): TDateTime;

도스에서 사용하는 날짜, 시간 형식의 변수를 TDateTime형으로 바꾼다. 도스 형식의 날짜는 4바이트의 정수형이며 각 비트에 날짜, 시간 요소를 포함하고 있다.

## FileExists SysUtils Unit

**문법** : function FileExists(const FileName: string): Boolean;

파일이 디스크 상에 존재하는지를 조사하며 파일이 없을 경우 False를 리턴한다. 존재하지 않는 파일을 사용하고자 할 경우 예외가 발생하므로 먼저 이 함수를 사용하여 파일이 있는지를 먼저 확인해 보아야 한다. 다음 예는 파일의 존재 여부를 점검해 본 후 파일을 삭제한다.

```
if FileExists(FileName) then
        DeleteFile(FileName);
```

## FileGetAttr SysUtils Unit

**문법** : function FileGetAttr(const FileName: string): Integer;

파일의 속성을 조사한다. 조사되는 파일의 속성은 다음과 같다. 속성을 나타내는 이 상수들과 FileGetAttr이 리턴하는 값을 AND 연산시켜 특정 속성의 유무를 조사한다. 속성 변경에 실패하면 음수의 에러 코드를 리턴한다.

| 상수 | 값 | 의미 |
|---|---|---|
| faReadOnly | $01 | 읽기전용 |
| faHidden | $02 | 숨은 파일 |
| faSysFile | $04 | 시스템 파일 |
| faVolumeID | $08 | 디스크 볼륨 |
| faDirectory | $10 | 디렉토리 |
| faArchive | $20 | 기록 속성 |
| faAnyFile | $3F | 모든 파일 |

## FileGetDate SysUtils Unit

**문법** : function FileGetDate(Handle: Integer): Longint;

파일이 처음 생성된, 또는 최후 수정된 날짜와 시간을 구한다. 이 날짜는 도스의 dir 명령시 나타나는 날짜이며 도스 고유의 포맷을 사용하므로 델파이에서 이 날짜 정보를 사용하려면 FileDataToDateTime 등의 포맷 변환 함수를 사용해야 한다.

## FileOpen SysUtils Unit

**문법** : function FileOpen(const FileName: string; Mode: Word): Integer;

델파이가 사용하는 내부적인 함수이므로 사용자가 직접 사용하는 경우는 드물다. 파일을 오픈할 때는 델파이가 제공하는 Reset, Rewrite, Append 등의 프로시저를 사용한다.

## FilePos system Unit

**문법** : function FilePos(var F):Longint;

파일의 현재 위치(FP)를 구한다. 이 함수는 텍스트 파일에는 사용할 수 없으며 파일이 열려 있어야 한다.

## FileRead SysUtils Unit

**문법** : function FileRead(Handle: Integer; var Buffer; Count: Longint): Longint;

핸들이 지시하는 파일로부터 Count 바이트 분만큼을 버퍼로 읽어들인다. 파일로부터 실제 읽은 바이트 수가 리턴된다.

## FileSearch SysUtils Unit

**문법** : function FileSearch(const Name, DirList: string): string;

DirList에 주어진 디렉토리의 경로에 Name 파일이 있는지 조사해 준다. DirList는 도스의 PATH문에서와 같이 여러 개의 디렉토리명을 세미콜론으로 끊어 기입한다. 파일을 찾을 경우 파일의 완전 경로를 리턴하며 파일을 찾지 못할 경우 널 스트링을 리턴한다.

## FileSeek SysUtils Unit

**문법** : function FileSeek(Handle: Integer; Offset: Longint; Origin: Integer): Longint;

파일의 현재 위치(FP)를 특정 위치로 옮긴다. 옮겨지는 위치는 Origin이 정하는 기준에서 Offset바이트만큼 떨어진 위치이다. Origin의 값은 다음과 같다.

| Origin | 의미 |
|---|---|
| 0 | 파일의 처음을 기준으로 한다. |
| 1 | 파일의 현재 위치를 기준으로 한다. |
| 2 | 파일의 끝을 기준으로 한다. |

## FileSetAttr SysUtils Unit

**문법** : function FileSetAttr(const FileName: string; Attr: Integer): Integer;

파일의 속성을 설정한다. 설정할 수 있는 속성에는 다음과 같은 종류가 있다. 여러 개의 속성을 동시에 설정하려면 속성 상수를 OR 연산으로 연결하여 사용한다. 속성 변경이 정상적으로 수행되면 0을 리턴하며 에러 발생시 음수의 에러 코드를 리턴한다.

| 상수 | 값 | 의미 |
|---|---|---|

| | | |
|---|---|---|
| faReadOnly | $01 | 읽기 전용 |
| faHidden | $02 | 숨은 파일 |
| faSysFile | $04 | 시스템 파일 |
| faVolumeID | $08 | 디스크 볼륨 |
| faDirectory | $10 | 디렉토리 |
| faArchive | $20 | 기록 속성 |
| faAnyFile | $3F | 모든 파일 |

## FileSetDate SysUtils Unit

**문법** : procedure FileSetDate(Handle: Integer; Age: Longint);

파일의 날짜를 변경한다. 이 날짜는 도스의 dir 명령시 나타나는 날짜이며 도스 고유의 포맷을 사용한다.

## FileSize system Unit

**문법** : function FileSize(var F);Longint;

파일 핸들 F가 지정하는 파일의 크기를 바이트 단위로 조사한다. 단 F가 레코드 파일일 경우는 레코드의 개수를 조사한다.

## FillChar system Unit

**문법** : procedure FillChar(var X:Count:Word;Value);

Count수만큼 값 Value를 채운다. Value는 Byte형이거나 Char형이어야 한다. 이 프로시저는 범위 체크를 전혀 하지 않으므로 사용에 주의를 기울여야 한다. 다음 예는 문자열 S에 문자열의 길이만큼 모두 문자 'T'로 가득 채운다.

```
FillChar(S, SizeOf(S), 'T');
```

## FindClose SysUtils Unit

**문법** : procedure FindClose(var SearchRec: TSearchRec);

FindFirst, FindNext 함수를 사용한 파일 검색을 종료한다. 굳이 파일 검색 작업을 종료해 주어야 할 필요는 없으며 실제로 16비트 버전의 윈도우즈에서 이 함수는 아무런 일도 하지 않는다. 하지만 32비트 버전에서는 반드시 검색 종료를 해 주어야 하므로 호환성을 위해 검색 후 종료해 주는 것이 좋다.

## FindFirst SysUtils Unit

**문법** : function FindFirst(const Path: string; Attr: Word; var F: TSearchRec): Integer;

주어진 검색 조건과 파일의 속성을 사용하여 첫 번째 일치하는 파일을 검색해 낸다. Path는 검색하고자 하는 디렉토리와 파일의 조건이며 와일드 카드식으로 표현된다. 예를 들어 'c:₩windows₩*.exe'는 windows 디렉토리의 확장자가 EXE인 첫 번째 파일을 검색한다. Attr 인수는 검색 대상이 되는 파일의 속성을 지정하며 다음과 같은 속성 상수를 사용한다.

| 상수 | 값 | 의미 |
|---|---|---|
| faReadOnly | $01 | 읽기 전용 |
| faHidden | $02 | 숨은 파일 |
| faSysFile | $04 | 시스템 파일 |
| faVolumeID | $08 | 디스크 볼륨 |
| faDirectory | $10 | 디렉토리 |
| faArchive | $20 | 기록 속성 |
| faAnyFile | $3F | 모든 속성 |

여러 가지 속성을 사용할 경우 or 연산자로 속성을 연결한다. 예를 들어 읽기 전용이면서 숨은 파일을 검색하고 싶다면 (faReadOnly + faHidden)과 같이 Attr 인수를 설정한다. 검색한 결과는 TSearchRec 레코드형의 F 변수에 저장되며 다음과 같은 정보를 가지고 있다.

```
TSearchRec = record
  Fill: array[1..21] of Byte;
  Attr: Byte;
  Time: Longint;
  Size: Longint;
  Name: string[12];   {파일 이름}
end;
```

이 정보는 FindNext 함수에 의해 다시 사용되며 계속 검색을 수행한다.

## FindNext SysUtils Unit

**문법** : function FindNext(var F: TSearchRec): Integer;

FindFirst에 이어 계속 검색을 수행한다. FindFirst가 첫 번째 검색한 결과를 저장한 F를 다시 인수로 넘겨주면 계속해서 일치하는 조건을 가진 파일을 검색해 내며 검색의 결과는 역시 F로 저장된다. 검색이 계속 이루어지고 있다면 0을 리턴하며 검색 중 에러가 발생했거나 더 이상 조건이 일치하는 파일이 없으면 음수의 에러 코드를 리턴한다. FindFirst와 FindNext는 조건에 맞는 모든 파일을 검색해 낼 때 사용한다.

## FloatToDecimal SysUtils Unit

**문법** : procedure FloatToDecimal(var Result: TFloatRec; Value: Extended; Precision, Decimals: Integer);

부동 소수점 형태의 수를 문자열과 소수점 위치값 등을 가진 TFloatRec형의 레코드로 변환한다. 이 레코드는 실수값을 적당히 포맷하여 출력하는 용도로 사용된다. Precision은 유효숫자를 지정하며 1~18까지의 값을 가진다. Decimal은 소수점 좌측의 정수부를 몇 자리까지 나타낼 것인가를 지정한다. 변환된 결과는 TFloatRec 형의 구조체에 저장되어 리턴되며 이 구조체는 다음과 같이 정의되어 있다.

```
TFloatRec = record
  Exponent: Integer;
  Negative: Boolean;
  Digits: array[0..18] of Char;
end;
```

각 필드는 다음과 같은 의미를 가진다.

| 필드 | 의미 |
|---|---|
| Exponent | 10의 거듭승을 나타내며 이는 실수값에 있는 정수부의 자리수이다. 실수값이 1보다 더 작다면 이 필드값은 음수가 된다. 실수값이 수치가 아니라면(NAN) -32768, 무한대(INF)라면 32767의 값을 가진다. |
| Negative | 음수이면 True, 양수이면 False |
| Digits | 18자리수의 유효숫자를 가지는 널 종료 문자열이다. 소수점은 문자열에 포함되어 있지 않으며 후행 제로는 모두 제거된다. |

다음 예는 이 함수의 동작을 살펴보기 위해 제작한 것이다. 폼의 레이블에 어떻게 출력되는지 값을 변경해 가며 살펴보도록 하자.

```
var
  FR:TFloatRec;
  Value:double;
begin
Value:=314.159265;
FloatToDecimal(FR,Value,5,10);
Label1.Caption:='Exponent='+IntToStr(FR.Exponent);
Label2.Caption:='Digits='+FR.Digits;
end;
```

이 코드를 실행하면 Exponent는 3, Digits는 31416이 출력된다.

## FloatToStrF

SysUtils Unit

**문법** : function FloatToStrF(Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): string;

인수로 주어진 포맷과 정밀도를 사용하여 실수값을 문자열로 바꾼다. Precision은 유효자리수를 지정하며 실수값이 Single형일 경우는 7이하, Double일 경우는 15이하, Extended형일 경우는 18이하여야 한다. Digits의 의미는 Format 인수에 따라 달라진다. Format 인수에는 다음과 같은 종류가 있다.

| 값 | 의미 |
|---|---|
| ffGeneral | 가장 일반적인 숫자 포맷을 사용한다. 변환된 결과는 고정 소수점 형식, 공학적 표기법 중 길이가 짧은 쪽이 선택되며 후행 제로는 모두 제거된다. 소수점은 꼭 필요한 경우에만 나타난다. |
| ffExponent | 공학적 표기법, 즉 부동 소수점 형태의 문자열로 변환된다. |
| ffFixed | 고정 소수점 형태의 문자열로 변환된다.후행 제로는 제거되지 않는다. |
| ffNumber | ffFixed와 마찬가지로 고정 소수점 포맷 형식을 사용하나 천단위마다 콤마가 삽입된다. |
| ffCurrency | 화폐 단위 형태의 문자열로 변환된다. 화폐 단위 표현의 형태는 Currency, CurrencyFormat 등의 전역 변수에 영향을 받으며 우리나라의 경우 수치 앞에 원(W)기호가 첨가된다. |

주어진 실수가 수치가 아닐 경우 변환된 문자열은 'NAN'이 되며 양의 무한대 수이면 'INF', 음의 무한대 수이면 '-INF'가 된다. 다음 예제는 이 함수를 사용하여 실수값을 문자열로 바꾼다.

```
var
  F:double;
  st:string;
begin
F:=3141.59265;
st:=FloatToStrF(F,ffGeneral,7,10);
label1.Caption:=st;
end;
```

변환된 결과는 3141.593이다. Format 인수를 ffExponent로 바꾸면 3.131593E+3으로 변환되며 ffNumber로 바꾸면 3,141.5930000000로 변환된다. ffCurrency로 바꾸면 W4,131.5930000000로 변환된다.

## FloatToStr

SysUtils Unit

**문법** : function FloatToStr(Value: Extended): string;

FloatToStrF 함수와 동일한 기능을 가지되 일반적인 수치 포맷(ffGeneral)을 사용하며 15자리의 유효 숫자를 사용한다. 즉 이

함수는 변환 포맷이 미리 정의되어 있는 FloatToStrF 함수라고 할 수 있다.

## FloatToText SysUtils Unit

**문법** : function FloatToText(Buffer: PChar; Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): Integer;

실수를 문자열로 변경하며 변경된 결과 버퍼에 저장된 문자의 개수를 리턴한다. 각 인수의 의미는 FloatToStrF 함수와 동일하므로 참고하기 바란다.

## FloatToTextFmt SysUtils Unit

**문법** : function FloatToTextFmt(Buffer: PChar; Value: Extended; Format: PChar): Integer;

FloatToText 함수와 동일한 기능을 하며 실수값을 문자열로 변경하여 버퍼에 저장한다.

## Flush system Unit

**문법** : procedure Flush(var F:Text);

출력용으로 오픈된 텍스트 파일의 버퍼를 비운다. 버퍼를 비운다는 말은 파일로 출력되기 위해 버퍼에 대기하고 있던 데이터를 외부 파일로 강제 전송한다는 뜻이다. 입력용으로 열려진 파일에는 아무런 동작도 하지 않는다.

## FmtLoadStr SysUtils Unit

**문법** : function FmtLoadStr(Ident: Word; const Args: array of const): string;

실행 파일의 리소스 문자열 테이블에서 문자열을 읽어들인다. Ident 인수는 리소스 파일 내의 문자열 리소스 ID이다.

## FmtStr SysUtils Unit

**문법** : procedure FmtStr(var Result: string; const Format: string; const Args: array of const);

Args 배열에 있는 항목들을 적절한 형태로 포맷하여 Result문자열에 대입해 준다. 포맷 형식은 Format 인수의 서식 문자열에 따라 달라진다. 자세한 포맷 형식에 대해서는 Format 함수를 참고하기 바란다.

## ForceDirectories FileCtrl Unit

**문법** : procedure ForceDirectories(Dir: string);

완전 경로를 주면 경로에 이르는 모든 디렉토리를 생성한다. 일반적으로 디렉토리는 한 번에 하나만 만들 수 있지만 이 함수를 쓰면 여러 개의 디렉토리를 한꺼번에 만들 수 있다. 예를 들어 C:₩a₩b₩c 디렉토리를 만들고자 할 경우 MkDir을 사용하면 a부터 만들고 b를 만든 후 c를 만들어야 하며 한꺼번에 세 개의 디렉토리를 만들 수 없다. 이 프로시저를 사용하면 한꺼번에 디렉토리를 생성할 수 있으므로 만들고자 하는 디렉토리의 완전 경로만 입력해 주면 된다. 다음 예는 C 드라이브에 한꺼번에 12개의 디렉토리를 계층적으로 생성한다.

```
ForceDirectories('c:₩a₩b₩c₩d₩e₩f₩g₩h₩i₩j₩k₩l');
```

## Format SysUtils Unit

**문법** : function Format(const Format: string; const Args: array of const): string;

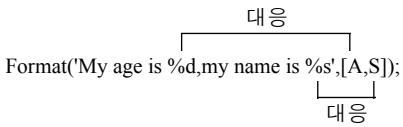
서식 문자열과 뒤따라오는 변수에 의해 문자열을 포맷한다. C의 printf 함수와 사용 방법이 유사하며 문자열 내부에 변수값을 삽입할 수 있는 아주 편리한 방법을 제공한다. 서식 문자열에는 일반 문자열과 서식이 삽입되며 일반 문자열은 그대로 출력되지만 서식은 대응되는 인수값으로 채워진다. 서식은 %기호와 변수의 타입으로 이루어지며 %와 타입 사이에 인덱스, 좌측 정렬 지정, 폭, 정밀도 지정이 삽입된다.

%[인덱스:][-][폭][.정밀도]타입

일단 Format 함수를 사용하는 예를 보자.

```
var
  A:integer;
  S,st:string;
begin
A:=정수값;
S:=문자열값;
st:=Format('My age is %d,my name is %s',[A,S]);
end;
```

서식 문자열에 포함된 %d 자리에 정수형 변수 A의 값이 삽입되며 %s 자리에 문자열 변수 S의 값이 삽입된다. A가 26, S가 '홍길동'이라면 이 문자열은 'My age is 26,my name is 홍길동'이 된다.



서식 문자열 내의 서식은 인수 리스트의 변수들과 순서대로 대응되며 서식의 개수와 인수의 개수가 일치해야 할 뿐만 아니라 대응되는 서식과 변수의 타입도 일치해야 한다. 만약 개수나 타입이 일치하지 않을 경우 컴파일은 가능하지만 실행중에 에러가 발생한다.

폭 지정이란 변수값이 문자열 내부에서 차지하는 자리수를 지정하며 생략시는 변수값의 길이만큼만 자리를 차지한다. 강제로 자리를 늘리고자 할 경우 서식과 타입사이에 폭을 정수로 지정한다. 예를 들어 %5d라고 하면 이 서식에 대입되는 정수가 5자리가 안되더라도 5자리를 강제로 차지한다. 단 여기서 지정하는 폭 지정은 최소한의 폭을 의미할 뿐이며 폭이 값의 실제 자리보다 작더라도 강제로 폭에 맞추지는 않는다. 예를 들어 I가 12345라는 다섯 자리 값을 가지는데 %2d와 대응될 경우 폭 지정 2를 지키기 위해 45만 출력하지 않으며 이 경우는 폭 지정이 무시된다.

정밀도 지정이란 실수에 사용되며 소수점 이하 몇 자리까지를 문자열로 바꿀 것인가를 지정한다. %10.5f 서식은 10자리를 차지하며 소수점 이하 5자리까지 문자열로 출력한다.

좌측 정렬 지정은 폭이 변수값보다 길 경우 공백을 우측에 배치하도록 하며 폭 지정 앞에 '-'기호를 붙여 표현한다. %5d에 대응되는 정수값이 12일 경우 출력은 '□□□12'가 되지만 서식을 %-5d'로 변경하면 '12□□□'가 된다. 단 여기서 □표시는 공백을 의미한다.

인덱스 지정자는 인수 리스트의 몇 번째 인수를 사용할 것인가를 지정하며 정수 하나와 :을 사용한다. 인덱스 지정자가 없으면 인수 리스트에 나타나는 순서대로 서식과 대응되지만 인덱스를 밝힘으로써 중복되는 인수 리스트를 재사용할 수 있다. 예를 들어 '%d%d%d',1,2,3은 앞쪽 서식부터 순서대로 정수 1,2,3에 대응되지만 '%d%d%0:d'는 앞쪽 둘은 순서대로 1,2에 대응되고 세 번째 서식은 인덱스 지정에 의해 첫 번째 인수 1에 대응된다.

타입 문자는 다음과 같다. C의 printf문에서 사용하는 타입 문자와 거의 동일하되 종류가 조금 적다.

| 문자 | 의미 |
|---|---|
| d | 정수값 하나에 대응된다. 대응되는 변수는 반드시 정수이어야 한다. |
| e | 실수값을 공학적 표기법, 즉 부동 소수점 형식으로 변환한다. |
| f | 실수값을 부동 소수점 형식으로 변환한다. |
| g | 실수값과 대응하되 고정 소수점 형식과 부동 소수점 형식 중 길이가 짧은 쪽으로 변환한다. |
| n | 실수값에 대응되며 천단위로 콤마를 삽입해준다. |
| p | 포인터 값에 대응된다. 출력되는 문자열은 SSSS:OOOO 형태의 세그먼트, 오프셋 형태를 띠며 이때 세그먼트, 오프셋은 모두 네 자리의 16진수를 사용한다. |
| m | 실수값에 대응되며 화폐 형태의 문자열로 변환한다. |
| s | 문자열 변수, 문자 변수에 대응된다. |
| x | 정수에 대응되며 16진수 형태의 문자열로 변환된다. |

타입 문자들은 대문자와 소문자를 구분하지 않는다. 서식에 삽입되는 폭 지정, 정밀도 지정대신 *를 사용하면 이 위치의 값을 인수 리스트로 부터 읽는다. 예를 들어

```
Format('%*d',[5,128]);
```

은 *자리에 첫 번째 인수 5를 읽어 값을 취하며 이 값이 폭 지정이 된다. 그래서 위 Format문은 다음과 동일하다.

```
Format('%5d',[128]);
```

서식 내에 *를 사용할 경우는 서식 내의 폭, 정밀도 지정까지도 인수 리스트의 변수값으로 지정할 수 있다.

다음 예는 Format 함수를 사용하여 문자열 내에 정수, 실수값을 삽입한 것이다.

```
var
  I,J:integer;
  F:double;
  st:string;
begin
I:=34;
j:=128;
F:=3.14159265;
st:=Format('I is %5d,J is %-7d,F is %8.6F',[I,J,F]);
label1.Caption:=st;
end;
```

실행 결과 레이블로 출력되는 문자열은 다음과 같다.

```
I is    34,J is 128  ,F is 3.141593
```

이 예제의 서식과 폭, 정밀도, 좌측 정렬을 다양하게 변경해 보면 서식을 어떻게 사용하는지 알 수 있을 것이다.

## FormatBuf

SysUtils Unit

**문법** : procedure FormatBuf(var Buffer; BufLen: Word; const Format; FmtLen: Word; const Args: array of const): Word;

서식에 의해 문자열을 작성한다. 자세한 문법은 Format 함수와 같으므로 참고하기 바란다. Format과 이 함수와의 차이점은 버퍼의 길이가 지정되어 있다는 점이다.

## FormatDateTime SysUtils Unit

**문법** : function FormatDateTime(const Format: string; DateTime: TDateTime): string;

TDateTime형의 변수에 저장된 시간과 날짜 정보를 특정한 형태의 문자열 포맷으로 변환한다. 문자열로 출력되는 형식은 다음과 같은 지정자에 의해 결정되며 각 지정자는 한 칸 이상의 공백으로 구분되어야 한다. 포맷을 지정하는 Format인수에는 지정자와 일반 문자열들이 올 수 있으며 일반 문자열들은 겹 따옴표로 싸 주어야 한다. 지정자의 대소문자는 구분하지 않는다.

| 지정자 | 의미 |
|---|---|
| c | 전역 변수 ShortDateFormat, LongTimeFormat이 지정하는 형식대로 출력되며 지정자가 생략될 경우의 디폴트 지정자이다. |
| d | 날짜값을 출력하되 선행 제로를 생략한다. (1~31) |
| dd | 날짜값을 출력하되 선행 제로를 출력한다. (01~31) |
| ddd | 요일을 생략형으로 출력한다. Sun, Mon, Sat 등과 같이 출력된다. 이 생략형 요일 이름은 ShortDayNames 변수에 의해 지정된다. |
| dddd | 요일 이름을 완전한 형식으로 출력한다. Sunday, Monday 등과 같이 출력된다. 완전한 요일 이름은 LondDayNames 변수에 의해 지정된다. |
| ddddd | ShortDateFormat 전역 변수가 지정하는 형식으로 날짜가 출력된다. |
| dddddd | LongDateFormat 전역 변수가 지정하는 형식으로 날짜가 출력된다. |
| m | 월을 출력하되 선행 제로를 생략한다. (1~12) |
| mm | 월을 출력하되 선행 제로를 출력한다. (01~12) |
| mmm | 월 이름을 생략형으로 출력한다. Jan, Dec 등과 같이 출력된다. 이 생략형 월 이름은 ShortMonthName 변수에 의해 지정된다. |
| mmmm | 월 이름은 완전한 형식으로 출력한다. January, December 등과 같이 출력된다. 완전한 월 이름은 LongMonthName 변수에 의해 지정된다. |
| yy | 년도를 두 자리 숫자로 출력한다. (00~99) |
| yyyy | 년도를 네 자리 숫자로 출력한다. (0000~9999) |
| h | 시간을 출력하되 선행 제로를 생략한다. (1~12) |
| hh | 시간을 출력하되 선행 제로를 출력한다. (01~12) |
| n | 분을 출력하되 선행 제로를 생략한다. (1~59) |
| nn | 분을 출력하되 선행 제로를 출력한다. (01~59) |
| s | 초를 출력하되 선행 제로를 생략한다. (1~59) |
| ss | 초를 출력하되 선행 제로를 출력한다. (01~59) |
| t | 시간값을 ShortTimeFormat 전역 변수가 지정하는 형식으로 출력한다. |
| tt | 시간값을 LongTimeFormat 전역 변수가 지정하는 형식으로 출력한다. |
| am/pm | 오전과 오후를 출력하며 h나 hh 지정자의 시간을 12시간제로 출력한다. 오전이면 am이 출력되며 오후면 pm이 출력된다. 대문자 소문자 모두 쓸 수 있으며 혼합해서 쓸 수도 있다. 출력되는 결과는 이 지정자의 대소문자 구성을 따른다. 즉 Am/Pm 지정자를 쓰면 Am이나 Pm이 출력되며 AM/PM 지정자를 쓰면 AM이나 PM이 출력된다. |
| a/p | 오전과 오후를 출력하며 h나 hh 지정자의 시간을 12시간제로 출력한다. 오전이면 a가 출력되며 오후면 p가 출력된다. 대문자 소문자 모두 쓸 수 있으며 혼합해서 쓸 수도 있다. 출력되는 결과는 이 지정자의 대소문자 구성을 따른다. |
| ampm | 12시간제로 시간을 출력하며 오전일 경우 전역 변수 TimeAmString에 저장된 문자열을 출력하며 오후일 경우 TimePmString에 저장된 문자열을 출력한다. |
| / | DateSeparatoer 지정자가 지정하는 날짜 구분 기호를 출력한다. |
| : | TimeSeparatoer 지정자가 지정하는 시간 구분 기호를 출력한다. |

출력 예를 보이면 다음과 같다.

```
FormatDateTime('dddd mmmm d yyyy hh:mm AM/PM', Now);
Friday November 24 1995 12:25 PM

FormatDateTime('"Now Time is" ddd mmm d yyyy hh:mm AM/PM', Now);
Now Time is Fri Nov 24 1995 12:26 PM
```

다양한 출력 포맷을 변경하여 직접 실습해 보기 바란다.

## FormatFloat SysUtils Unit

**문법** : function FormatFloat(const Format: string; Value: Extended): string;

인수로 주어진 Value 실수값을 Format이 지정하는 포맷 형식에 따라 문자열로 변환한다.

## Frac system Unit

**문법** : function Frac(X:Real):Real;

실수 X의 소수부를 구한다. Frac(X)는 X-Int(X)와 같은 결과를 반납한다. 예를 들어 Frac(3.1415)는 정수부 3의 값을 빼고 남은 소수부 0.1415의 값을 계산한다.

## Free

system Unit

**문법** : procedure Free;

이 프로시저를 호출한 인스턴스가 nil인지를 점검해 보고 nil이 아닐 경우 Destroy를 호출하여 객체를 메모리에서 해제한다. 인스턴스가 nil일 경우 Free 호출은 무시된다.

## FreeMem

system Unit

**문법** : procedure FreeMem(var P:Pointer;Size:Word);

GetMem 프로시저에 의해 동적으로 할당된 메모리를 해제한다. size 인수는 할당 해제할 메모리의 크기이며 이 크기는 GetMem 프로시저에 의해 할당된 크기와 일치해야 한다. 다음 예는 정수형 변수를 위한 메모리를 할당한 후 해제한다.

```
var
  p: pointer;
begin
  { 메모리를 할당한다. }
  GetMem(p, SizeOf(integer));
  { 할당한 메모리를 사용한다.}
  { 할당을 해제한다. }
  FreeMem(p, SizeOf(integer));
end;
```

## GetDir

system Unit

**문법** : procedure GetDir(D:Byte;var S:String);

지정한 드라이브의 현재 디렉토리를 조사한다. D로 드라이브의 번호를 지정하면 현재 디렉토리가 문자열 S에 대입된다. 드라이브 번호 D는 0일 경우, 디폴트 드라이브 1일 경우 A, 2일 경우 B 등과 같이 지정한다. 이 프로시저는 일체의 에러 체크를 하지 않는다. 만약 드라이브 번호로 주어진 D가 존재하지 않을 경우 조사되는 디렉토리는 X:₩가 된다. 다음 예는 현재 드라이브의 디렉토리를 조사하여 레이블에 출력한다.

```
var
  dir:string;
begin
  GetDir(0,dir);
  label1.caption:=dir;
end;
```

## GetMem

system Unit

**문법** : procedure GetMem(var P:Pointer; Size:Word);

동적으로 메모리를 할당하며 할당된 번지를 포인터형의 변수 P에 대입한다. Size는 동적으로 할당할 메모리의 크기를 바이트 수로 나타낸다. 할당된 메모리의 변수는 P^로 읽거나 쓸 수 있다. 다음 예는 정수형 변수를 위한 메모리를 할당한 후 사용하며 사용 후 FreeMem으로 할당을 해제한다.

```
var
  p: pointer;
begin
  GetMem(p, SizeOf(integer));
  integer(P^):=2;
  label1.caption:=IntToStr(integer(P^));
  FreeMem(p, SizeOf(integer));
end;
```

## Halt

system Unit

**문법** : procedure Halt [ (ExitCode:Word) ];

프로그램을 강제로 종료하고 운영체제로 제어권을 넘긴다. Exitcode는 프로그램의 탈출 코드이다.

## Hi

system Unit

**문법** : function Hi(X):Byte;

상위 바이트 값을 부호없는 8비트의 정수로 읽어낸다. 인수로 주어진 X는 Integer 또는 Word형의 16비트 정수값이다.

```
var
  B: Byte;
begin
  B := Hi($1234);  { $12가 대입된다. }
end.
```

## High

system Unit

**문법** : function High(X);

인수로 주어진 X의 값 중 가장 큰값을 찾는다. X가 integer형이라면 결과는 32767이 되며 LongInt형이라면 결과는 21474836647이 된다. 결과값은 인수로 주어진 X와 동일한 형이다. 다음 예는 열거형 중 가장 큰 값을 찾는다. 레이블로 출력되는 값은 6이다.

```
type
  Day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
var
  i:integer;
begin
  i:=integer(High(Day));
```

```
 label1.caption:=IntToStr(i);
end;
```

## IdentToColor Graphics Unit

**문법** : function IdentToColor(const Ident: string; var Color: Longint): Boolean;

색상 명칭 문자열을 32비트로 표현된 Color 색상값으로 변경한다. Ident 문자열이 파스칼이 사용하는 색상 명칭이 아닐 경우 Color 인수는 변경되지 않으며 False를 리턴한다.

```
ans:=IdentToColor('clYellow',Col);
```

## Inc system Unit

**문법** : procedure Inc(var X [ ; N:Longint ] );

변수의 값을 1 증가시킨다. Inc(X)는 X:=X+1과 동일한 동작을 한다. 두 번째 인수를 사용하면 1 이상의 값을 증가시킬 수도 있다. Inc(X,5)는 X:=X+5와 동일한 동작을 한다. 첫 번째 인수는 서수형의 변수이며 확장 문법이 적용될 경우 PChar형도 가능하다. N은 정수형의 상수 또는 변수여야 한다. Inc 함수는 가장 최적화된 코드를 생성해 내므로 루프 내부에서 사용하기에 적합하다.

## Include system Unit

**문법** : procedure Include(var S:Set of T; I:T);

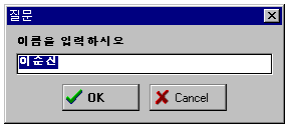
집합형 S에 요소 I를 추가한다. Include(S,I)는 S := S + (I)와 동일한 동작을 하지만 좀 더 효율적인 코드를 생성해 낸다.

## InputBox Dialogs Unit

**문법** : function InputBox(const ACaption, APrompt, ADefault: string): string;

문자열을 입력할 수 있는 대화상자를 보여주고 사용자에게 문자열을 입력받아 리턴한다. 인수의 의미는 인수의 이름을 참조하기 바라며 다음에 간단한 예를 보인다.

```
var
 str:string;
begin
str:=InputBox('질문','이름을 입력하시요','이순신');
end;
```



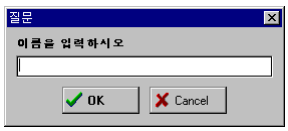

사용자가 문자열 입력 후 Cancel 버튼을 누르면 디폴트 문자열이 리턴되며 OK를 누르면 에디트 박스에 입력한 문자열이 입력된다.

## InputQeury Dialogs Unit

**문법** : function InputQuery(const ACaption, APrompt: string; var Value: string): Boolean;

문자열을 입력할 수 있는 대화상자를 보여주고 사용자에게 문자열을 입력받아 리턴한다. 대화상자가 열리면 아무것도 입력되어 있지 않은 빈 에디트를 보여주며 이 에디트에 사용자가 문자열을 입력한다. 입력된 결과는 리턴값으로 전달되는 것이 아니라 참조 호출로 전달된 value로 전달되며 리턴값은 사용자가 누른 버튼의 종류이다. 다음에 예를 보인다.

```
var
 str:string;
 ans:Boolean;
begin
ans:=InputQuery('질문','이름을 입력하시오',str);
end;
```




사용자가 대화상자에서 어떤 버튼을 눌렀는가를 꼭 알아야 할 경우는 InputQuery를 사용하며 단순히 문자열만 입력받고자 할 경우는 InputBox를 사용한다.

## Insert system Unit

**문법** : procedure Insert(Source:String;var S:String;Index:Integer);

한 문자열의 중간에 다른 문자열을 삽입한다. 삽입되는 위치는 Index 인수가 지정하는 위치이다. 문자열을 삽입한 후의 문자열 길이가 255자를 넘을 경우 255문자 이후의 문자는 잘려진다. Dest 문자열이 'Orange'이고 Src 문자열이 'Apple'인 경우 Insert(Src,Dest,3);는 Dest 문자열의 3번째 문자인 a위치에 Src 문자열을 삽입하며 Dest 문자열은 'OrAppleange'가 된다.

```
var
  S: String;
begin
  S := 'abcdefghijkl';
  Insert(' hotdog ', S, 5);
  label1.caption:=S;
end;
```

이 예를 실행시키면 레이블로 abcd hotdog efghijkl 문자열이 출력된다.

## Int

system Unit

**문법** : function Int(X:Real):Real;

실수 X의 정수부를 구한다. 소수부는 반올림되지 않으며 버려진다. Int(3.14)는 3의 값을 반납한다. 주의할 것은 Int 함수의 리턴값은 정수값을 담지만 데이터형은 여전히 실수형이라는 점이다.

```
R:=int(3.1415);    {R은 3이 된다.}
```

## IntToHex

SysUtils Unit

**문법** : function IntToHex(Value: Longint; Digits: Integer): string;

10진 정수값을 16진수 형태의 문자열로 바꾸어 문자열을 리턴한다. 예를 들어 IntToHex(100,2) 명령에 의해 십진수 100을 16진수로 바꾼 문자열 '64'가 리턴된다. Digits 인수는 만들어질 16진수의 자리수를 지정한다. 그러나 이 값은 최소한의 자리수를 지정할 뿐이며 자리수보다 16진수의 크기가 더 많을 경우는 이 인수의 값이 무시된다. 즉 IntToHex(100,3)에 의해 최소한 세 자리를 할당하여 '064'가 리턴되지만 IntToHex(100,1)로 한다고 하여 두 자리의 16진수를 강제로 한 자리 숫자로 만들지는 않는다. 다음 예는 십진수 1234를 4자리수의 16진수 04D2로 바꾸어 레이블로 출력한다.

```
label1.caption:=IntToHex(1234,4);
```

## IntToStr

SysUtils Unit

**문법** : function IntToStr(Value: Longint): string;

정수를 문자열로 바꾸어 리턴한다. 정수값을 레이블이나 에디트 박스에 출력하고자 할 경우 직접 그 값을 출력할 수는 없다. 왜냐하면 레이블의 Caption 속성이나 에디트의 Text 속성은 문자열형이므로 정수형 값을 대입받을 수 없기 때문이다. 정수값을 문자열로 바꾸고자 할 경우에는 이 함수를 사용한다. 다음은 Label1에 정수형 변수 Age의 값을 출력한 예이다.

```
Label1.Caption:=IntToStr(Age);
```

## IOResult

system Unit

**문법** : function IOResult:Integer;

마지막 입출력 동작에서 발생한 상태값을 조사한다. 이 함수를 사용하여 입출력 에러를 검사하려면 {$I-}옵션을 반드시 꺼 주어야 한다. IOResult 함수를 호출하면 내부적인 에러값은 리셋된다. 이 함수가 리턴하는 값이 0이면 에러가 없는 것이다.

## IsValidIdent

SysUtils Unit

**문법** : function IsValidIdent(const Ident: string): Boolean;

문자열의 내용이 명칭으로서 유효한가 아닌가를 점검해 준다. 명칭은 알파벳, 숫자, 밑줄로 구성되며 첫 자에 숫자가 올 수 없다. 명칭으로서 유효한 문자열이면 True를 리턴하며 명칭으로서 유효하지 않은 문자가 있으면 False를 리턴한다.

## Length

system Unit

**문법** : function Length(S:String):Integer;

주어진 문자열의 길이를 구한다. Length('Kora')는 5의 값을 리턴한다.

## Ln

system Unit

**문법** : function Ln(X:Real):Real;

자연 로그값을 구한다.

## Lo

system Unit

**문법** : function Lo(X):Byte;

하위 바이트값을 부호없는 8비트의 정수로 읽어낸다. 인수로 주어진 X는 Integer 또는 Word형의 16비트 정수값이다.

```
var
  B: Byte;
begin
  B := Lo($1234);  { $34가 대입된다. }
end.
```

## LoadStr

SysUtils Unit

**문법** : function LoadStr(Ident: Word): string;

실행 파일에서 인수 Ident가 지정하는 문자열 리소스를 읽어들인다. 문자열 리소스가 정의되어 있지 않으면 빈 문자열이 리턴된다. 문자열을 리소스로 분리시키면 프로그램의 유지 및 보수에 많은 이점이 있다.

## Log10 Math Unit

**문법** : function Log10(X: Extended): Extended;

10의 로그를 구한다.

## Log2 Math Unit

**문법** : function Log2(X: Extended): Extended;

2의 로그를 구한다.

## LogN Math Unit

**문법** : function LogN(Base, X: Extended): Extended;

임의 베이스에 대한 로그를 구한다.

## Low system Unit

**문법** : function Low(X);

인수로 주어진 X의 값 중 가장 작은값을 찾는다. X가 Integer형이라면 결과는 -32768이 되며 LongInt형이라면 결과는 -21474836648이 된다. 결과값은 인수로 주어진 X와 동일한 형이다. 다음 예는 열거형 중 가장 작은 값을 찾는다. 레이블로 출력되는 값은 0이다.

```
type
  Day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
var
  i:integer;
begin
  i:=integer(Low(Day));
  label1.caption:=IntToStr(i);
end;
```

## LowerCase SysUtils Unit

**문법** : function LowerCase(const S: string): string;

문자열 내부의 대문자를 모두 소문자로 바꾼다. 문자 변환은 7비트의 아스키 코드 내에서 이루어지며 문자 'A'~'Z'까지가 변환의 영향을 받는다. 나머지 기호나 숫자, 한글은 변환되지 않는다.

## Max Math Unit

**문법** :

```
function Max(A,B: Integer): Integer; overload;
function Max(A,B: Int64): Int64; overload;
function Max(A,B: Single): Single; overload;
function Max(A,B: Double): Double; overload;
function Max(A,B: Extended): Extended; overload;
```

두 값중 큰 값을 구한다. 여러 가지 데이터 타입에 대한 함수가 중복정의되어 있다.

## MaxValue Math Unit

**문법** : function MaxValue(const Data: array of Double): Double;

주어진 배열에서 가장 큰 값을 구한다.

## MaxIntValue Math Unit

**문법** : function MaxIntValue(const Data: array of Integer): Integer;

주어진 정수 배열에서 가장 큰 값을 구한다.

## MaxAvail system Unit

**문법** : function MaxAvail:Longint;

힙에서 사용 가능한 메모리 블럭 중 연속적으로 배치되어 있는 가장 큰 메모리의 크기를 조사한다. 이 함수가 조사하는 크기는 한 번에 생성할 수 있는 가장 크기가 큰 동적 변수의 크기가 된다. 즉 한 번에 할당할 수 있는 메모리 크기이다. 이 양은 남아 있는 메모리양과는 의미가 다르다. 설사 남아 있는 메모리의 총량이 500이더라도 제일 큰 덩어리가 100밖에 되지 않으면 크기 200의 메모리를 할당할 수 없다. 왜냐하면 하나의 변수는 연속된 메모리에 있어야 하기 때문이다. 그래서 레코드나 배열 등의 크기가 큰 메모리를 할당할 때는 이 함수로 사용 가능한 연속된 블럭의 크기를 먼저 조사해 보아야 하며 메모리가 없을 경우는 에러 메시지를 출력해 주어야 한다.

```
  if MaxAvail < SizeOf(변수형) then
    MessageDlg('메모리가 없네요',
    mtWarning, [mbOk], 0);
  else
 {메모리를 할당한다.}
```

## Mean Math Unit

**문법** : function Mean(const Data: array of Double):

Extended;

주어진 배열 요소의 산술 평균값을 규한다.

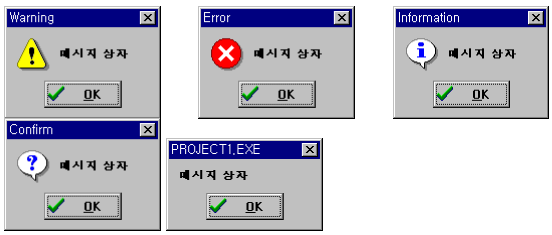
## MessageDlg Dialogs Unit

**문법** : function MessageDlg(const Msg: string; AType: TMsgDlgType; AButtons: TMsgDlgButtons; HelpCtx: Longint): Word;

메시지를 전달하는 대화상자를 보여주며 사용자로부터 응답을 받아들여 리턴한다. Msg 문자열이 메시지의 내용이며 AType에 의해 메시지 상자의 형태와 캡션에 출력된 문자열이 결정된다.

| 값 | 형태 |
|---|---|
| mtWarning | 노란색 느낌표 비트맵이 나타난다. |
| mtError | 빨간색의 엑스 비트맵이 나타난다. |
| mtInformation | 파란색의 i자 비트맵이 나타난다. |
| mtConfirmation | 물음표 비트맵이 나타난다. |
| mtCustom | 비트맵을 사용하지 않는다. |

AType에 따른 메시지 상자의 모양은 다음과 같다.




AButton 인수는 메시지 상자에 나타날 버튼의 종류를 지정하며 여러 개의 버튼을 집합형으로 전달한다. 리턴값은 사용자가 대화상자에 나타난 버튼 중 어떤 버튼을 눌렀는가를 나타내는 값이다.

| 버튼 | 모양 | 리턴값 |
|---|---|---|
| mbYes | Yes | mrYes |
| mbNo | No | mrNo |
| mbOK | OK | mrOk |
| mbCancel | Cancel | mrCancel |
| mbHelp | Help | |
| mdAbort | Abort | mrAbort |
| mbRetry | Retry | mrRetry |
| mbIgnore | Ignore | mrIgnore |
| mbAll | All | mrAll |

버튼 이름을 일일이 지정해 주는 것이 번거로우면 미리 정의되어 있는 다음과 같은 집합명을 사용한다.

| 집합명 | 버튼 |
|---|---|
| mbYesNoCancel | Yes, No, Cancel 버튼 |
| mbOkCancel | Ok, Cancel 버튼 |
| mbAbortRetryIgnore | Abort, Retry, Ignore 버튼 |

이 집합명은 그 자체가 집합형이므로 [ ] 괄호를 쓰지 않아도 된다.

## MessageDlgPos Dialogs Unit

**문법** : function MessageDlgPos(const Msg: string; AType: TMsgDlgType; AButtons: TMsgDlgButtons; HelpCtx: Longint; X, Y: Integer): Word;;

MessageDlg 함수와 동일한 동작을 하는 함수이다. 단 MessageDlg는 대화상자를 무조건 화면 중앙에 열어주지만 이 함수는 X,Y 인수를 사용하여 대화상자가 출력될 화면 좌표를 지정할 수 있다.

## Min Math Unit

**문법** :
function Min(A,B: Integer): Integer; overload;
function Min(A,B: Int64): Int64; overload;
function Min(A,B: Single): Single; overload;
function Min(A,B: Double): Double; overload;
function Min(A,B: Extended): Extended; overload;

두 값중 작은 값을 구한다. 여러 가지 데이터 타입에 대한 함수가 중복 정의되어 있다.

## MinValue Math Unit

**문법** : function MinValue(const Data: array of Double): Double;

주어진 배열에서 가장 작은 값을 구한다.

## MinIntValue Math Unit

**문법** : function MinIntValue(const Data: array of Integer): Integer;

주어진 정수 배열에서 가장 작은 값을 구한다.

## MkDir system Unit

**문법** : procedure MkDir(S:String);

새로운 서브 디렉토리를 생성한다. 문자열 S에 생성하고자하는 서브 디렉토리의 경로를 준다. 디렉토리를 생성하는 규칙은 도스에서와 동일하다. 즉 같은 이름의 디렉토리나 파일이 있어서는 안된다.

## Move system Unit

**문법** : procedure Move(var Source, Dest;Count:Word);

Source에서 Dest로 Count 바이트만큼 메모리 단위로 복사를 수행하며 범위 체크는 하지 않는다. 두 변수가 같은 세그먼트 영역에 있을 때 설사 두 변수의 복사 대상 번지가 겹쳐 있더라도 Move는 원본을 파괴하지 않고 정상적인 복사를 수행한다. 그러나 세그먼트가 다를 경우는 원본이 덮여져 복사가 제대로 수행되지 않을 수도 있다. 그러나 델파이는 겹쳐진 영역에 변수를 생성하지 않으므로 이 문제는 그리 큰 문제는 아니다. 다만 직접 변수의 메모리를 지정했거나(비디오 램의 경우), 번지를 변경하였을 경우에는 이런 문제가 발생한다. 메모리 단위로 복사를 수행하므로 Source, Dest의 데이터 타입은 어떤 형이라도 상관없으며 심지어 두 데이터 타입이 달라도 복사는 수행된다. 다음 예제는 문자열 데이터를 Longint형의 데이터로 복사한다.

```
var
  A: array[1..4] of Char;
  B: Longint;
begin
  Move(A, B, SizeOf(A));
  label1.caption:=IntToStr(B);
end;
```

물론 문자열이 초기화되지 않았으므로 B에 대입되는 값은 이 경우 알 수 없는 쓰레기 값이다.

## New system Unit

**문법** : procedure New(var P: Pointer);
function New( <pointer type> ): Pointer;

함수와 프로시저 두 가지 형태가 있으며 포인터 변수를 곧바로 인수로 사용할 수도 있고 포인터 타입을 인수로 사용할 수도 있다. 동적 변수를 힙에 생성하며 생성된 변수를 포인트하는 포인터 값을 리턴한다. 생성된 변수는 P^에 의해 참조된다. 메모리가 부족할 경우는 실행시 에러를 발생시킨다. 동적으로 할당한 메모리는 사용 후에 반드시 Dispose로 할당을 해제해 주어야 한다. 다음 예제는 문자열 형의 변수를 동적으로 할당하여 사용한 후 할당 해제한다.

```
type
  Str = String[50];
var
  P: ^Str;
begin
  New(P);
  P^ := 'dynamic string';
  label1.caption:=P^;
  Dispose(P);
end;
```

위 예제는 포인터 변수를 직접 인수로 사용하는 New 프로시저를 사용하였다. 다음과 같이 포인터 타입을 만든 후 이 타입을 인수로 사용하여 New 함수를 사용할 수도 있다. 두 예제의 실행 결과는 동일하다.

```
type
  Str = String[50];
  PStr = ^Str;
var
  P: PStr;
begin
  P := New(PStr);
  P^ := 'dynamic string';
  label1.caption:=P^;
  Dispose(P);
end;
```

## NewStr SysUtils Unit

**문법** : function NewStr(const S: string): PString;

문자열 S의 복사본을 힙에 생성하며 그 시작 번지를 리턴해준다. NewStr 함수로 문자열을 할당했다면 DisposeStr 함수로 반드시 해제해 주어야 한다. 문자열을 힙에 생성한 후 길이를 바꾸는 일이 없어야 한다. 만약 길이가 길어지면 힙의 다른 변수를 파괴할 가능성이 있으며 길이가 짧아지면 완전한 해제가 불가능해진다.

## Now SysUtils Unit

**문법** : function Now: TDateTime;

현재 날짜와 시간을 한꺼번에 구한다. 날짜와 시간은 TDateTime형이므로 곧바로 문자열로 출력할 수 없으며 DateTimeToStr 함수를 사용하여 문자열로 바꾸어 주어야 한다. Label1.Caption:=DateTimeToStr(Now) 명령에 의해 Lable1에 오늘 날짜와 시간이 출력된다.
StrToDate: 문자열을 날짜 형태로 바꾸어 TDateTime형의 변수에 저장해 준다. 문자열에 저장된 날짜는 반드시 유효한 날짜

정보를 가지고 있어야 하며 만약 날짜가 무효할 경우 EConverError 예외를 발생시킨다. 문자열은 년, 월, 일을 나타내는 두 개 또는 세 개의 숫자를 담고 있어야 하며 각 요소는 DateSeparator 변수가 지정하는 구분문자(보통 슬래시 기호)에 의해 구분되어 있어야 한다. 년, 월, 일의 순서는 ShortDateTime 변수에 따라 달라지며 mm/dd/yy 또는 dd/mm/yy 또는 yy/mm/dd 중 하나가 된다. 만약 문자열에 두 개의 숫자만 있으면 이는 각각 월, 일의 정보로 인식되며 년은 올해의 년도가 된다. 년의 값은 0~99 또는 1900~1999 둘 다 가능하다.

## Odd

system Unit

**문법** : function Odd(X:Longint):Boolean;

인수로 주어진 정수가 홀수인지를 검사한다. 홀수이면 True를 리턴하고 짝수이면 False를 리턴한다. 다음 예는 i값이 홀수인지를 점검하여 결과를 레이블로 출력한다. 물론 i에 3을 대입한 후 점검하므로 이 예제의 결과는 당연히 홀수이다.

```
var
  i:Integer;
begin
i:=3;
if Odd(i) then
  label1.caption:='홀수입니다.'
else
  label1.caption:='홀수가 아닙니다.';
end;
```

## Ord

system Unit

**문법** : function Ord(X):Longint;

순서형 값의 순서값을 조사한다. 다음 예제는 열거형의 열거요소 중 Soo가 몇 번째 값인가를 조사한다.

```
type
   Yoil = (Wol,Hwa,Soo,Mok,Gum,TTo,Il);
begin
label1.caption:='Soo member is '+IntToStr(Ord(Soo));
end;
```

## ParamCount

system Unit

**문법** : function ParamCount:Word;

명령행에서 프로그램으로 전달된 인수의 개수를 전달한다. 각 인수는 공백이나 탭으로 분리되어 있다. 인수가 전달되지 않을 경우 이 값은 0이다.

## ParamStr

system Unit

**문법** : function ParamStr(Index):String;

명령행으로 전달된 인수를 리턴하며 Index로 인수의 번호를 지정한다. 이 함수는 특히 실행 파일이 실행된 디렉토리 경로를 알고자 할 때 유용하며 ParamStr(0)로 디렉토리를 읽는다. 또는 인수를 받아들이는 프로그램일 경우는 인수를 읽을 때 이 함수를 사용한다.

## Pi

system Unit

**문법** : function Pi:Real;

원주율을 구한다. 원주율은 3.14159265358979323846264338327950288419716939937510582097494459230781640628620 8.....

## Point

Classes Unit

**문법** : function Point(AX, AY: Integer): TPoint;

인수로 전달된 X좌표와 Y좌표를 사용하여 TPoint형의 레코드를 생성한다. TPoint형은 화면 상의 한 점을 기억하는 레코드이며 다음과 같이 정의되어 있다.

```
TPoint = record
  X: Integer;
  Y: Integer;
end;
```

TPoint형의 인수를 요구하는 Polygon 등의 함수에 사용된다.

## Pos

system Unit

**문법** : function Pos(Substr:String; S:String);Byte;

문자열 내의 부분 문자열을 검색한다. 부분 문자열이 검색된 위치를 리턴해 준다. 만약 부분 문자열이 발견되지 않으면 리턴값은 0이다.

## Pred

system Unit

**문법** : function Pred(X);

순서형 변수의 바로 앞 값을 읽는다.

## Ptr system Unit

**문법** : function Ptr(Seg, Ofs:Word):Pointer;

세그먼트와 오프셋 값을 사용하여 포인터 값을 만들며 리턴되는 값은 Seg:Ofs 번지를 포인터하는 포인터이다. Ptr 함수로 만들어진 포인터를 사용하여 메모리 값을 읽으려면 다음과 같이 타입 캐스팅을 해 주어야 한다.

```
if Byte(Ptr(Seg0040, $49)^) = 7 then
  Writeln('Video mode = mono');
```

세그먼트와 오프셋을 사용하여 특정 메모리를 직접 읽어야 할 경우는 극히 드물지만 비디오 램이나 BIOS 데이터 영역을 읽을 때 가끔 사용한다.

## Random system Unit

**문법** : function Random [ (Range:Word) ];

0보다 크거나 같고 Range보다 작은 범위에서 난수를 만든다. Range 인수가 생략될 경우는 0~1사이의 실수 난수가 생성된다.

## Randomize system Unit

**문법** : procedure Randomize;

난수 루틴을 초기화한다. Random 함수는 예측 불가능한 난수를 만들기는 하지만 생성되는 난수가 매번 동일하므로 Randomize로 난수 생성기를 시스템 클럭을 사용하여 초기화 시켜주어야 완전한 난수를 발생시킨다.

## Read system Unit

**문법** : 타입드 파일 function Read(F, V1 [,V2...Vn] );

텍스트 파일 function Read([var F:Text; ]
V1 [,V2,....,Vn] );

타입드 파일일 경우 파일로부터 한 요소(예를 들어 레코드 하나)를 읽어들여 변수에 대입해 주며 텍스트 파일일 경우 파일로부터 변수값을 읽어들인다. 문자열을 읽을 경우 읽고난 후에 다음 줄로 개행을 하지 않으며 읽은 문자열이 변수의 길이보다 길 경우 문자열이 잘려나간다. 정수나 실수를 읽을 때 일체의 공백은 무시되며 수치로 변경될 수 없는 문자가 있을 경우는 에러가 발생된다.

## Readln system Unit

**문법** : procedure Readln ( [var F;Text; ] V1 [,V2,...Vn]);

파일로부터 문장 한 줄을 읽어 변수 V1이하에 대입하며 다음 문장으로 이동한다. Readln(F)와 같이 읽어들일 변수를 지정하지 않으면 다음 줄로 이동하기만 한다. 파일 변수가 생략될 경우는 표준 입력 장치인 키보드로부터 문자열을 입력 받아 V1이하의 변수에 대입한다. 인수없이 Readln만 사용되면 Enter 키가 입력될 때까지 대기한다. 다음 예제는 키보드로부터 문장을 입력받아 다시 레이블로 출력한다.

```
var
  s : String;
begin
Write('문장을 입력하시요: ');
Readln(s);
label1.caption:=s;
end;
```

## ReAllocMem SysUtils Unit

**문법** : function ReAllocMem(P: Pointer; CurSize, NewSize: Cardinal): Pointer;

힙에 할당되어 있는 메모리 블럭을 재할당한다. 현재 크기와 변경하고자 하는 새로운 크기를 인수로 지정해 준다. 현재 크기보다 더 큰 크기로 할당하고자 할 경우, 즉 할당된 메모리를 확장할 경우 확장되는 메모리 영역은 0으로 채워진다. 리턴되는 값은 크기가 조정된 메모리 블럭의 포인터이며 이 값은 원래의 포인터 값과는 항상 다르다.

## Rect Classes Unit

**문법** : function Rect(ALeft, ATop, ARight, ABottom: Integer): TRect;

TRect형 구조체는 두 점의 좌표를 사용하여 사각형 영역을 정의한다. 좌상단, 우하단의 각 X,Y 좌표로도 표현할 수 있으며 TPoint형의 레코드 두 개로 표현할 수도 있다.

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

Rect 함수는 네 개의 좌표를 입력받아 TRect형의 레코드를 만들면 TRect형의 인수를 사용하는 함수나 TRect형의 속성을 정의할 때 사용한다.

## Rename system Unit

**문법** : procedure Rename(var F:Newname);

파일의 이름을 변경한다. 변경하고자 하는 이름을 문자열로 전달해 주면 된다. 파일명을 변경하려면 AssignFile 함수로 파일 핸들에 외부 파일명을 할당해 주고 이 함수를 호출하면 된다. 파일을 오픈할 필요는 없다.

## Reset system Unit

**문법** : procedure Reset(var F [:File;RecSize:Word]);

파일을 읽기 전용으로 오픈한다. F는 AssignFile 함수로 외부 파일과 연결된 파일 핸들이며 RecSize는 F가 untyped 파일일 경우 한 번에 읽을 레코드의 크기를 지정한다. RecSize가 생략될 경우 이 값은 128바이트로 간주된다. 외부 파일이 존재하지 않을 경우는 에러가 발생하며 파일이 이미 열려져 있는 상태이면 파일을 닫은 후에 다시 오픈한다. 이 함수로 파일을 오픈한 후에 Readln 등의 함수로 파일로부터 데이터를 읽는다.

## Rewrite system Unit

**문법** : procedure Rewrite(var F;File [;Recsize:Word]);

파일을 쓰기 전용으로 오픈한다. F는 AssignFile 함수로 외부 파일과 연결된 파일 핸들이며 RecSize는 F가 untyped 파일일 경우 한번에 출력할 레코드의 크기를 지정한다. RecSize가 생략될 경우 이 값은 128바이트로 간주된다. 파일이 없을 경우에는 파일을 생성하고 파일이 이미 존재할 경우는 원래의 파일을 지운 후 다시 만든다. 파일이 이미 열려져 있으면 파일을 닫은 후에 다시 생성하며 파일 위치(FP)는 파일의 선두에 맞추어진다. 이 함수로 파일을 오픈한 후에 Writeln 등의 함수로 데이터를 파일로 출력한다. 다음 예는 test.txt라는 간단한 파일을 만든 후 문자열 하나를 출력한다.

```
var
  F:TextFile
begin
  AssignFile(F, 'test.txt');
  Rewrite(F);
  Writeln(F, '출력할 내용');
  CloseFile(F);
end;
```

파일 입출력이 끝난 후에 파일 핸들은 반드시 CloseFile 프로시저로 닫아 주어야 한다.

## RmDir system Unit

**문법** : procedure RmDir(S: String);

서브 디렉토리를 삭제한다. 삭제할 디렉토리명을 문자열로 넘겨준다. 도스의 RD 명령과 거의 동일한 규칙을 사용하며 디렉토리가 비어있지 않은 경우, 현재 디렉토리인 경우, 루트 디렉토리인 경우는 디렉토리를 삭제할 수 없다.

## Round system Unit

**문법** : function Round(X:Real):Lonint);

실수값 X를 정수값으로 만든다. 즉 실수의 소수 부분을 버리고 정수 부분만 결과로 취한다. 이때 소수부 첫째 자리에서 반올림이 일어나며 반올림의 결과는 부호에 상관없이 일어난다. 반올림의 결과와 Longint형의 범위를 벗어날 경우는 실행시 에러가 발생한다. 다음 예제는 실수 3.1415를 반올림하여 정수로 만든다.

```
var
  F:double;
begin
F:=3.1415;
label1.caption:=IntToStr(Round(F));
end;
```

레이블로 출력되는 결과는 소수부가 잘려나간 3이다. 실수값을 3.6415로 변경하면 반올림에 의해 4가 출력된다.

## RunError system Unit

**문법** : procedure RunError [(Errorcode:Byte)];

실행시 에러를 발생시키고 프로그램을 종료한다. 치명적인 에러가 발생했을 경우 이 함수를 사용한다.

## Seek system Unit

**문법** : procedure Seek(var F;N:Longint);

파일의 현재 위치(FP)를 N위치로 옮긴다. untyped 파일일 경우 N의 의미는 파일 선두에서의 바이트 단위 거리이지만 typed 파일일 경우 N은 N번째 요소의 위치를 의미한다. 예를 들어 레코드형 파일일 경우 N이 5이면 여섯 번째 레코드의 위치로 이동한다. 첫 번째 레코드의 번호가 0번이다. Seek(F, FileSize(F))는 파일의 끝으로 현재 위치를 이동시킨다.

## SeekEof system Unit

**문법** : function SeekEof[(var F:Text)]:Boolean;

파일의 현재 위치가 끝(EOF)인지 점검한다.

## SeekEoln system Unit

**문법** : function SeekEoln[(var F:Text)]:Boolean;

파일의 현재 위치가 줄의 끝(EOL)인지 점검한다.

## SelectDirectory FileCtrl Unit

**문법** : function SelectDirectory(var Directory: string; Options: TSelectDirOpts; HelpCtx: Longint): Boolean;

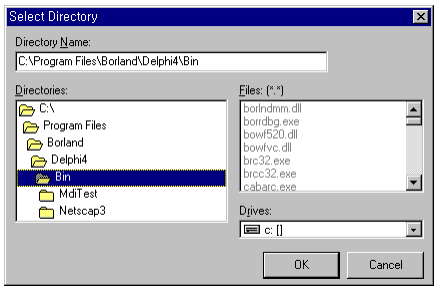
디렉토리 선택 대화상자를 보여주고 사용자로 하여금 디렉토리를 선택하도록 한다. 드라이브 콤보 박스와 디렉토리 리스트 박스를 사용하여 디렉토리를 선택하며 파일 리스트 박스는 어떤 파일이 있는지만 보여준다. 디렉토리를 선택하는 것 뿐만 아니라 에디트 박스에 디렉토리명을 입력하여 없는 디렉토리를 직접 생성할 수도 있다. 인수로 주어지는 Dierectory 문자열은 대화상자가 처음 열릴 때 선택될 디렉토리이며 사용자가 대화상자에서 선택한 디렉토리명이 이 인수로 리턴된다. 사용자가 OK 버튼을 누르면 True를 리턴하고 Cancel 버튼을 누르면 False를 리턴한다.
두 번째 인수 Options는 대화상자의 모양과 기능을 정의하는 옵션이며 다음과 같은 값의 집합형이다.

| 옵션 | 의미 |
|---|---|
| [] | 사용자는 존재하는 디렉토리를 선택할 수 있을 뿐이며 없는 디렉토리를 만들 수 없다. |
| sdAllowCreate | 에디트 박스에 사용자가 직접 존재하지 않는 디렉토리를 입력하는 것을 허용한다. 그러나 이 옵션을 사용한다고 해서 디렉토리를 생성해 주는 것은 아니며 사용자가 직접 Directory 인수를 참조하여 생성해 주어야 한다. |
| sdPerformCreate | 이 옵션을 sdAllowCreate 옵션과 함께 사용할 경우 존재하지 않는 디렉토리가 입력되면 직접 디렉토리를 만들어 준다. |
| sdPrompt | 디렉토리를 새로 만들기 전에 사용자로부터 디렉토리를 정말로 만들 것인가를 물어본다. 위의 두 옵션이 선택되었을 때만 의미가 있는 옵션이다. |

다음 예는 이 함수를 사용하는 간단한 예제이다.

```
var
  Dir: string;
begin
  Dir := 'C:₩';
  if SelectDirectory(Dir, [sdAllowCreate, sdPerformCreate, sdPrompt],0) then
    Label1.Caption := Dir;
end;
```

디렉토리 선택 대화상자의 모습은 다음과 같다.




## SetTextBuf system Unit

**문법** : procedure SetTextBuf(var F:Text;var Buf[;Size:Word]);

텍스트 파일의 입출력에 사용될 버퍼를 지정한다. 디폴트로 주어진 128바이트의 버퍼는 대부분의 경우는 충분하지만 입출력이 잦을 경우 버퍼를 늘려주면 속도를 조금 더 향상시킬 수 있다. 텍스트 파일 F가 사용하는 버퍼를 내부 버퍼를 사용하지 않고 Buf로 변경하며 Size 인수는 Buf의 크기를 지정한다. Size가 생략될 경우 이 값은 SizeOf(Buf)가 되며 이는 Buf가 가진 모든 버퍼를 다 사용한다는 뜻이다. 이 함수는 Reset, Rewrite 등의 오픈 함수가 호출된 직후에 사용한다. 그러나 일단 입출력을 수행한 파일에는 이 함수를 사용하여 버퍼를 변경하지 않는 것이 좋다. 왜냐하면 버퍼를 변경하면 기존 버퍼에 있던 내용을 잃어버리기 때문이다.

## SetLength Sysstem Unit

**문법** : procedure SetLength(var S; NewLength: Integer);

문자열이나 동적 배열 변수의 길이를 정한다. 이때 문자열의 길이가 늘어난다면 메모리가 재할당되며 기존의 문자열 내용은 보존된다. 그러나 새로 할당된 메모리는 초기화되지 않는다. 메모리가 부족할 경우 EOutOfMemory 예외가 발생한다.

## Sin system Unit

**문법** : function Sin(X:Real):Real;

삼각함수 Sin을 계산한다. 인수로 전달된 X는 360분법의 각도가 아닌 라디안 값이다.

## SizeOf

system Unit

**문법** : function SizeOf(X):Word;

X의 크기를 바이트 단위로 리턴한다. X는 변수이거나 데이터 타입이다. SizeOf(integer)는 2이며 SizeOf(double)은 8이다. 주로 레코드나 오브젝트 등 크기가 큰 데이터의 크기를 계산할 때 사용한다. 가상 메소드 테이블을 가진 오브젝트의 인스턴스 크기를 조사하면 VMT에 저장된 크기가 리턴된다.

## Sqr

system Unit

**문법** : function Sqr(X:Real):Real;

X의 제곱값을 구한다. Sqr(X)는 X*X와 동일한 결과를 계산해 낸다.

## Sqrt

system Unit

**문법** : function Sqrt(X:Real):Real;

X의 제곱근을 구한다. Sqrt(2)는1.414213이다. 이때 계산되는 제곱근은 양의 제곱근이다.

## Str

system Unit

**문법** : procedure Str(X [:Width [:Decimals]];var S);

숫자값 X를 문자열 S로 출력한다. 이때 출력되는 값의 폭은 Width 인수가 지정하는 폭을 사용하며 Decimals 인수가 지정하는 정밀도를 사용한다. 숫자값 X는 정수형이거나 실수형 모두 가능하다. 다음 예는 3.1415를 폭 5, 정밀도 3으로 문자열로 변환한다. 레이블로 출력되는 문자열은 3.142이다.

```
var
  F:double;
  S:string;
begin
F:=3.1415;
Str(F:5:3,S);
label1.caption:=S;
end;
```

## StrAlloc

SysUtils Unit

**문법** : function StrAlloc(Size: Word): PChar;

널 종료 문자열을 담기 위한 버퍼를 할당하며 버퍼의 시작 번지를 리턴한다. 버퍼의 크기는 인수로 주어진 Size 바이트이며 최대 문자열 길이는 종료 표시를 나타내는 NULL 값을 제외한 Size-1 문자이다. Size의 최대 크기는 65526까지 가능하다. StrAlloc로 문자열을 위해 할당한 메모리는 반드시 StrDispose로 할당 해제해 주어야 한다.

## StrBufSize

SysUtils Unit

**문법** : function StrBufSize(Str: PChar): Word;

StrAlloc에 의해 동적으로 할당된 버퍼의 크기를 구한다. 이 함수가 리턴하는 값을 널 종료 문자를 포함한 값이다. 만약 Str이 동적으로 할당된 메모리를 가리키고 있지 않을 경우 에러 메시지는 발생하지 않지만 결과는 예측할 수 없다.

## StrCat

SysUtils Unit

**문법** : function StrCat(Dest,Source:PChar):PChar;

두 개의 문자열을 연결한다. Source 문자열을 Dest 문자열에 추가하며 연결된 문자열을 리턴한다. 문자열의 길이 점검은 하지 않으므로 Dest 문자열이 Source의 문자열을 충분히 수용할 만한 크기를 가지도록 해야 한다. 다음 예제는 두 개의 문자열을 결합한다. 레이블로 출력되는 결과는 made in Korea이다.

```
var
  S1:array [0..128] of Char;
  S2:array [0..128] of Char;
begin
StrCopy(S1,'made in ');
StrCopy(S2,'Korea');
StrCat(S1,S2);
label1.caption:=StrPas(S1);
end;
```

## StrComp

SysUtils Unit

**문법** : function StrlComp(Str1, Str2:PChar): Integer;

두 개의 문자열을 비교한다. S1과 S2 문자열이 같으면 0을 리턴하며 S1<S2이면 음수의 값을 리턴하며 S1>S2이면 양수를 리턴한다. 문자열끼리의 비교는 문자열을 이루는 각 문자의 문자 코드를 비교하여 수행된다.

## StrCopy

SysUtils Unit

**문법** : function StrCopy(Dest, Source: PChar): PChar;

널 종료 문자열을 복사한다. Source 문자열을 Dest 문자열에 복사하며 Dest 문자열을 리턴한다. 이 함수는 문자열의 길이 점검을 하지 않으므로 Dest 문자열의 길이가 Source 문자열의 길이보다 짧지 않도록 주의해야 한다. 만약 Dest 문자열의 길이가 Source 문자열을 충분히 수용하지 못할 경우 Dest 뒷부분의 다른 변수값이 파괴된다. 널 종료 문자열끼리는 :=연산자에 의해

직접 대입할 수 없으므로 이 함수를 사용하여 문자열끼리 복사해 주어야 한다. 예를 들어 Source가 'Korea'이고 이 값을 Dest 문자열로 대입하고 싶다고 해서 Dest:=Source와 같은 문장을 쓸 수 없다. 이때는 StrCopy(Dest,Source)와 같이 함수를 사용해야 한다.

## StrDispose SysUtils Unit

**문법** : function StrDispose(Str: PChar;

StrAlloc에 의해 할당된 문자열을 위한 메모리를 해제한다. StrAlloc에 의해 할당된 메모리는 반드시 할당 해제되어야 한다.

## StrEcopy SysUtils Unit

**문법** : function StrECopy(Dest, Source: Pchar): PChar;

문자열끼리 복사하되 복사가 끝난 지점의 번지를 리턴해 준다. 따라서 이 함수를 연속적으로 사용하면 여러 개의 문자열을 한 문자열에 합할 수 있다. 이 함수는 문자열의 길이 점검을 하지 않으므로 Dest 문자열의 길이가 Source 문자열의 길이보다 작지 않아야 한다.

## StrEnd SysUtils Unit

**문법** : function StrEnd(Str: PChar): Pchar;

문자열의 끝 번지를 찾아준다. 문자열의 끝 번지란 곧 문자열의 끝을 나타내는 NULL 문자의 번지를 말한다.

## StrFmt SysUtils Unit

**문법** : function StrFmt(Buffer, Format: PChar; const Args: array of const): PChar;

문자열을 포맷하여 Buffer에 저장한다. 문자열 포맷에 관한 자세한 사항은 Format을 참조하기 바란다.

## StrIComp SysUtils Unit

**문법** : function StrLComp(Str1, Str2: PChar; MaxLen: Word): Integer;

두 개의 문자열을 비교하되 대문자와 소문자를 구분하지 않는다. S1과 S2 문자열이 같으면 0을 리턴하며 S1<S2이면 음수의 값을 리턴하며 S1>S2이면 양수를 리턴한다. 문자열끼리의 비교는 문자열을 이루는 각 문자의 문자 코드를 비교하여 수행된다.

## StrLCat SysUtils Unit

**문법** : function StrLCat(Dest, Source: PChar; MaxLen: Word): PChar;

StrCat와 마찬가지로 문자열을 연결한다. 차이점은 문자열을 연결한 결과 생성되는 문자열이 MaxLen이상의 길이가 되지 않도록 길이 점검을 해준다는 점이다.

## StrLComp SysUtils Unit

**문법** : function StrLIComp(Str1, Str2: PChar; MaxLen: Word): Integer;

StrComp 함수와 마찬가지로 두 개의 문자열을 비교한다. 차이점이라면 인수로 주어진 MaxLen 문자분까지만 비교한다는 점이다.

## StrLCopy SysUtils Unit

**문법** : function StrLCopy(Dest, Source: PChar; MaxLen: Cardinal): PChar;

StrCopy 함수와 마찬가지로 문자열을 복사한다. 차이점이라면 Dest 문자열의 길이를 점검하여 MaxLen이상의 문자가 복사되지 않도록 해준다는 점이다.

## SetLen SysUtils Unit

**문법** : function StrLen(Str: PChar): Cardinal;

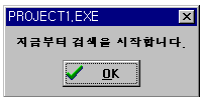
주어진 문자열의 길이를 조사한다. 이 길이에는 널종료 문자열이 제외된다.

## ShowMessage Dialogs Unit

**문법** : procedure ShowMessage(const Msg: string);

대화상자를 열어 Msg 문자열을 보여주기만 하며 OK 버튼을 누르면 대화상자를 닫는다. 캡션에는 이 대화상자를 호출한 프로그램의 실행 파일 이름이 출력된다.

ShowMessage('지금부터 검색을 시작합니다.');




간단한 메시지만 전달하고 사용자의 입력을 받아야 할 필요가 없을 때 이 프로시저를 사용한다.

## ShowMessagePos Dialogs Unit

**문법** : procedure ShowMessagePos(const Msg: string; X, Y: Integer);

ShowMessage 프로시저와 동일한 동작을 하는 프로시저이다. 단 ShowMessage는 대화상자를 무조건 화면 중앙에 열어주지만 이 함수는 X,Y 인수를 사용하여 대화상자가 출력될 화면 좌표를 지정할 수 있다.

## StringToColor Graphics Unit

**문법** : function (S: string): TColor;

문자열로 전달된 색상의 명칭을 TColor형의 색상값으로 변경한다.

## StrLFmt SysUtils Unit

**문법** : function StrLFmt(Buffer: PChar; MaxLen: Word; Format: PChar; const Args: array of const): PChar;

문자열을 포맷한다. StrFmt 함수와 동일하되 버퍼의 길이를 지정한다는 점만 다르다.

## StrLIComp SysUtils Unit

**문법** : function StrLIComp(Str1, Str2: PChar; MaxLen: Word): Integer;

StrIComp 함수와 마찬가지로 대소문자 구분없이 두 개의 문자열을 비교한다. 차이점이라면 인수로 주어진 MaxLen 문자분까지만 비교한다는 점이다.

## StrLower SysUtils Unit

**문법** : function StrLower(Str: PChar): PChar;

주어진 문자열을 소문자로 바꾼다.

## StrMove SysUtils Unit

**문법** : function StrMove(Dest, Source: PChar; Count: Cardinal): PChar;

Source 문자열에서 Count 문자분만큼 Dest 문자열로 복사한다. 이 때 두 문자열의 복사 영역이 상호 겹치더라도 정확하게 복사를 수행한다.

## StrNew SysUtils Unit

**문법** : function StrNew(Str: PChar): PChar;;

Str 문자열의 복사본을 힙에 생성한다. Str이 NIL이거나 빈 문자열일 경우는 아무런 동작도 수행하지 않는다. 똑같은 문자열의 복사본을 만들어 준다.

## StrPas SysUtils Unit

**문법** : function StrPas(Str: PChar): String;

널종료 문자열을 파스칼형 문자열로 바꾸어 주며 바꾸어진 파스칼형 문자열을 리턴한다. 델파이에서 기본적으로 사용하는 문자열의 형태는 파스칼형이다. 에디트 박스의 Text 속성, 각종 컴포넌트의 Caption 속성이 모두 파스칼형 문자열이다. 그러나 윈도우즈 API 함수에서 사용하는 함수는 널 종료 문자열이어서 두 문자열 간의 대입이 불가능하다. API 함수에서 얻은 널 종료 문자열을 컴포넌트의 Caption 속성에 대입하거나 파스칼형 문자열 변수에 대입하고자 할 경우는 반드시 이 함수를 사용하여 문자열 형식을 바꾸어 주어야 한다.

## StrPCopy SysUtils Unit

**문법** : function StrPCopy(Dest: PChar; Source: String): PChar;

파스칼형 문자열을 널 종료 문자열로 바꾸어 복사해 준다. 델파이에서 기본적으로 사용하는 문자열의 형태는 파스칼형이다. 에디트 박스의 Text 속성, 각종 컴포넌트의 Caption 속성이 모두 파스칼형 문자열이다. 그러나 윈도우즈 API 함수에서 사용하는 함수는 널 종료 문자열이어서 두 문자열 간의 대입이 불가능하다. 컴포넌트의 Caption 속성이나 파스칼형 문자열 변수의 문자열을 윈도우즈 API 함수에 사용하고자 할 경우는 반드시 이 함수를 사용하여 문자열 형식을 바꾸어 주어야 한다. 이 함수는 길이 점검을 하지 않으므로 Dest의 길이가 Source의 길이보다 짧지 않도록 유의해야 한다.

## StrPLCopy SysUtils Unit

**문법** : function StrPLCopy(Dest: PChar; const Source: string; MaxLen: Word): PChar;

StrPCopy 함수와 동일한 동작을 하되 복사되는 문자열의 길이가 MaxLen 문자분 이상이 되지 않도록 제한해 준다.

## StrPos SysUtils Unit

**문법** : function StrPos(Str1, Str2: PChar): PChar;

널 종료 문자열인 Str1에서 부분 문자열 Str2를 찾아 그 번지를 리턴해 준다. 만약 부분 문자열이 발견되지 않을 경우 NIL 값을 리턴해 준다.

## StrRScan SysUtils Unit

**문법** : function StrRScan(Str: PChar; Chr: Char): PChar;

StrScan 함수와 마찬가지로 문자열에서 문자 하나를 찾아준다. 차이점이라면 StrScan 함수는 문자열의 좌측에서부터 문자를 찾아 나가지만 StrRScan은 문자열의 우측에서부터 문자를 찾아 나간다. 'Father and mother'라는 문자열에서 문자 a를 찾을 때 좌측에서 찾을 경우와 우측에서 찾을 경우 그 결과가 다르다.

## StrScan SysUtils Unit

**문법** : function StrScan(Str: PChar; Chr: Char): PChar;

널종료 문자열인 Str에서 문자 Chr을 찾아 그 번지를 리턴해준다. 만약 Chr 문자가 발견되지 않을 경우 NIL 값을 리턴해준다.

## StrToDate SysUtils Unit

**문법** : function StrToDate(const S: string): TDateTime;

문자열을 날짜 형식으로 바꾼다. 문자열은 날짜 형식으로 되어 있어야 한다. 즉 문자열 안에는 날짜를 이루는 세 가지 요소가 순서에 맞게 배치되어 있어야 하며 각 요소는 날짜 구분 문자(/)로 분리되어 있어야 한다. 문자열에 날짜로 변경할 수 없는 문자가 있을 경우는 EConvertError 예외가 발생한다.

## StrToDateTime SysUtils Unit

**문법** : function StrToDateTime(const S: string): TDateTime;

문자열은 날짜와 시간을 담는 TDateTime형의 변수로 바꾼다. 문자열에는 MM/DD/YY HH:MM:SS 형태로 날짜와 시간이 담겨 있어야 하며 시간값은 24시간제로 표현하거나 AM, PM을 추가로 뒤에 붙여 주어야 한다.

## StrToFloat SysUtils Unit

**문법** : function StrToFloat(const S: string): Extended;

문자열의 실수를 실수값으로 변경한다. 문자열에는 부호를 나타내는 +/-와 숫자, 그리고 소수점만 있어야 하며 지수를 나타내는 E가 올 수 있다. 그 외의 문자가 있을 경우는 EConvertError 예외가 발생한다.

## StrToInt SysUtils Unit

**문법** : function StrToInt(const S: string): Longint;

문자열을 정수값으로 바꾼다. 이때 문자열에는 숫자로 바꿀 수 없는 문자가 있어서는 안되며 만약 무효한 문자가 있을 경우 EConvertError 예외를 발생시킨다. 에디트 박스에 사용자가 입력한 값을 정수형 변수에 대입하고자 할 때 에디트 박스의 Text 속성과 정수형 변수의 데이터형이 일치하지 않아 대입이 불가능하다. 문자열에 입력된 정수를 정수형 값으로 바꾸어 주어야 정수형 변수에 대입할 수 있다. 다음은 Edit1에 입력한 정수값을 정수형 변수 Age에 대입하는 예이다.

```
Age:=StrToInt(Edit1.Text);
```

## StrToIntDef SysUtils Unit

**문법** : function StrToIntDef(const S: string; Default: Longint): Longint;

문자열을 정수로 변환한다. StrToInt 함수는 변환이 불가능할 경우 예외를 발생시키지만 이 함수는 변환이 불가능할 경우 Default 값을 사용한다.

## StrToTime SysUtils Unit

**문법** : function StrToTime(const S: string): TDateTime;

문자열을 시간 형태로 바꾸어 TDateTime형의 변수에 저장해 둔다. 문자열에는 HH:MM:SS 형태로 시, 분, 초가 담겨 있어야 하며 AM, PM은 있어도 되고 없어도 된다. 시간은 반드시 24시간 형태로 표시해 주어야 한다. 예를 들어 오후 8:26은 20:26으로 표현한다. 만약 문자열에 무효한 시간값이 들어 있을 경우 EConvertError 예외를 발생한다.

## StrUpper SysUtils Unit

**문법** : function StrUpper(Str: PChar): PChar;

주어진 문자열을 대문자로 바꾼다.

## Succ system Unit

**문법** : function Succ(X);

순서형 변수의 바로 다음 값을 읽는다.

## Sum Math Unit

**문법** : function Sum(const Data: array of Double): Extended register;

주어진 배열의 총 합계를 구한다.

## SumInt Math Unit

**문법** : function SumInt(const Data: array of Integer): Integer register;

주어진 징수 배열의 총 합계를 구한다.

## Swap system Unit

**문법** : function Swap(X);

X의 상위 워드와 하위 워드를 교체한다.

```
var
  X: Word;
begin
  X := Swap($1234);  { $3412가 된다. }
end.
```

## TextToFloat SysUtils Unit

**문법** : function TextToFloat(Buffer: PChar; var Value: Extended): Boolean;

Buffer에 문자열 형태로 보관된 실수를 실수값으로 변경한다. 변경에 성공하면 True를 리턴하고 실패하면 False를 리턴한다.

## Time SysUtils Unit

**문법** : function Time: TDateTime;

시스템의 시계를 참조하여 현재 시간을 구한다. 시간은 TDateTime형이므로 곧바로 문자열로 출력할 수 없으며 TimeToStr 함수를 사용하여 문자열로 바꾸어 주어야 한다. Label1.Caption:=TimeToStr(Time) 명령에 의해 Label1에 현재 시간이 출력된다.

## TimeToStr system Unit

**문법** : function TimeToStr(Time: TDateTime): String;

시간을 나타내는 Time을 문자열 형태로 변경한다. 시간값을 곧바로 레이블이나 에디트 등으로 출력할 수 없기 때문에 이 함수를 사용하여 문자열로 변경해 주어야 한다. 다음 예는 현재 시간을 레이블로 출력한다.

```
label1.caption:=TimeToStr(Time);
```

## Trunc system Unit

**문법** : function Trunc(X: Real): Longint;

실수의 소수점 이하를 버리고 정수부만을 취한다. Round 함수는 소수점 첫 째 자리에서 반올림을 하지만 이 함수는 반올림없이 무조건 소수점을 버림한다. 다음 예는 3.6415를 버림하여 3을 레이블로 출력한다.

```
var
  F:double;
begin
F:=3.6415;
label1.caption:=IntToStr(Trunc(F));
end;
```

Trunc 함수 대신 Round 함수를 사용한다면 반올림되어 4가 출력된다.

## Truncate system Unit

**문법** : procedure Truncate(var F);

현재 파일 위치 이후의 모든 레코드를 잘라버리며 파일 위치는 EOF가 된다. 텍스트 파일에서는 이 프로시저를 사용할 수 없다.

## UpCase system Unit

**문법** : function UpCase(Ch: Char): Char;

문자 Ch를 대문자로 바꾼다. 소문자 a에서 대문자 z까지의 문자가 대문자 A~Z로 변경되며 그 외 나머지 문자일 경우는 전혀 영향을 받지 않는다.

## UpperCase system Unit

**문법** : fun;

문자열 내부의 소문자를 모두 대문자로 바꾼다. 문자 변환은 7비트의 아스키 코드 내에서 이루어지며 문자 'A'~'Z'까지가 변환의 영향을 받는다. 나머지 기호나 숫자, 한글은 변환되지 않는다.

## Val system Unit

**문법** : fun;

문자열 변수 S를 숫자형 변수로 바꾼다. 문자열 변수 S에는 숫자가 담겨 있어야 한다. 문자열을 바꾼 결과를 받기 위한 인수 V는 정수형이거나 실수형 변수 모두 가능하다. S에 숫자로 바꿀 수 없는 무효 문자가 있을 경우 Code에는 무효 문자의 위치가 들어가며 만약 무효 문자가 없다면 Code 값은 0가 된다. 널 종료 문자열의 경우 Code에 리턴되는 에러 위치는 실제 문자의 위치보다 하나가 더 많은 값이 된다.

## Write system Unit

**문법** : 텍스트 파일 : procedure Write( [ var F: Text; ] P1 [,P2,...,Pn ] );

타입드 파일 : procedure Write(F, V1 [V2,...Vn]);

변수값을 파일로 출력한다. F는 출력용으로 오픈된 텍스트 파일이어야 하며 파일이 생략될 경우 표준 출력 장치인 화면으로 출력된다. 파일 변수 다음의 인수 P1이하는 파일로 실제 출력될 값들이며 이 값에 필드 폭, 정밀도를 지정할 수 있다. 출력할 수 있는 타입은 문자형, 정수형, 실수형, 문자열, 진위형 등이 있다. 타입드 파일일 경우 출력되는 변수 V1이하는 파일의 요소와 같은 형이어야 한다. 즉 레코드형 파일이라면 출력되는 값도 같은 형의 레코드이어야 한다. 파일의 현재 위치가 파일의 끝이라면 파일의 크기는 출력된 데이터의 길이만큼 늘어난다.

## Writeln system Unit

**문법** : procedure Writeln([ var F: Text; ] P1 [, P2, ...,Pn ] );

Write 함수의 기능을 확장한 함수이다. Write를 호출하여 변수를 파일로 출력하되 변수값 다음에 EOL을 출력하여 자동으로 개행되도록 한다.