

Part 01

델파이 살펴보기

델파이를 이용한 프로그램 개발에 필요한 기본적인 개념을 소개합니다.
델파이의 구성화면 및 기본환경을 살펴보고 기본문법인 오브젝트 파스칼에 대해 알아봅니다.
클래스와 개체를 이해하고 컴포넌트들의 사용방법을 익혀 어플리케이션을 작성해 봅니다.



1장

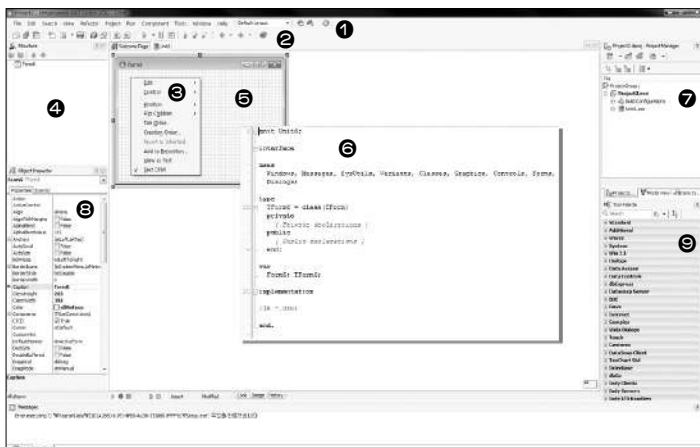
델파이 프로젝트 시작하기

이번장에서는 델파이 개발환경을 소개하고 프로젝트 관리방법과 오브젝트 파스칼 문법 등을 살펴보고 클래스와 개체, 컴포넌트 사용법에 대해 자세히 알아보겠습니다.

1. 델파이 개발환경 둘러보기

첫 장에서는 델파이로 간단한 어플리케이션을 작성해보도록 하겠습니다. 어플리케이션 작성을 통하여 어플리케이션을 구성하는 파일들의 구성요소와 델파이의 기본 환경들이 자연스럽게 소개될 것 입니다. 단, 일반적인 방식인 비주얼한 컴포넌트를 먼저 사용하는 것이 아니라 소스 코드에 직접 코딩 하는 것으로부터 시작 하겠습니다. 델파이는 비주얼한 UI와 풍부한 컴포넌트 지원으로 클릭 몇 번만으로 복잡한 내용도 손쉽게 만들어 낼 수 있습니다.

아래 그림은 델파이가 실행 되는 동안 떠 있는 메인 윈도우 화면입니다. 델파이를 이용한 작업을 할 때 가장 많이 쓰이는 환경으로 기억해 두어야 할 화면입니다.



델파이에서 제공하는 환경들에 대해 간략하게 살펴보겠습니다. 델파이 환경들은 프로그램을 작성하면서 필요한 시점에서 하나하나 자세하게 사용방법에 대해 설명하겠습니다.

❶ 메인 메뉴

프로젝트 열기, 저장하기 등의 File 메뉴와 Edit, 델파이 IDE의 각종 윈도우들을 보이거나 보이지 않게 하기 위한 View 메뉴들로 구성되어 있습니다.

❷ 툴 바

윈도우 메뉴 중 자주 사용하는 메뉴 등을 툴 버튼으로 처리하도록 묶어놓은 바입니다. 사용자 지정이 가능합니다.

❸ 팝업 메뉴(스피드 메뉴)

오른쪽 마우스를 눌렀을 때 표시되는 윈도우 메뉴입니다.

❹ 스트럭처 뷰

프로젝트에서 사용된 컴포넌트, 코드 등의 내용을 트리 형태로 보여줍니다.

❺ 폼 디자이너

폼을 디자인하는 공간입니다. 실행 시, 코드 에디터에 작성한 내용이 실행되어 보여집니다.

❻ 코드 에디터

소스를 입력하는 공간입니다. 단축키 F12로 폼 디자이너와 코드 에디터 화면을 전환할 수 있습니다.

❼ 프로젝트 매니저

현재 프로젝트의 파일과 그룹 관리를 위한 공간입니다. 유닛, 폼 등의 추가 및 삭제 등이 가능합니다.

❽ 오브젝트 인스펙터

컴포넌트의 속성과 이벤트를 설정하기 위하여 사용됩니다. 속성이란 컴포넌트의 위치, 색, 폰트, 정렬 방법들을 말하고, 이벤트는 컴포넌트를 클릭한다거나 사용자 행동에 대해서 어떻게 반응을 하도록 할 것인지를 나타냅니다.

❾ 툴 팔레트

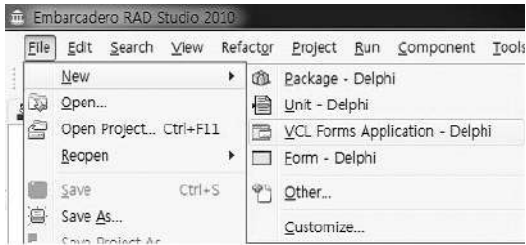
델파이에서 제공하는 모든 컴포넌트를 기능별로 구분하고 있습니다. 컴포넌트 검색 및 선택이 가능합니다.

이제, 델파이를 이용하여 간단한 프로젝트를 만들어 보면서 델파이가 가지고 있는 여러 기능들과 오브젝트 파스칼 언어에 대해 자세히 살펴보겠습니다.

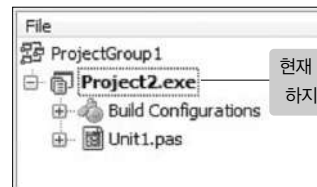
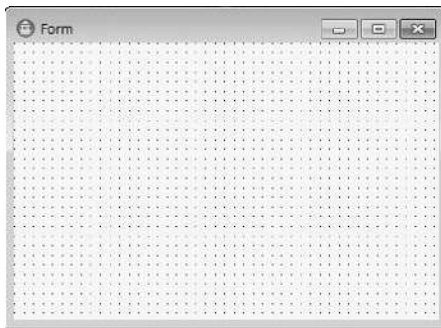
||||||| 딱 락 하 기 |||||||||

프로그램 시작하기

- 01 주 메뉴에서 File → New → VCL Forms Application - Delphi를 이용하여 어플리케이션을 시작합니다.



- 02 처음에 어떤 파일이 보이십니까? 아래와 같이 폼 디자이너 화면에 디폴트로 폼이 생성된 것을 확인할 수 있습니다.

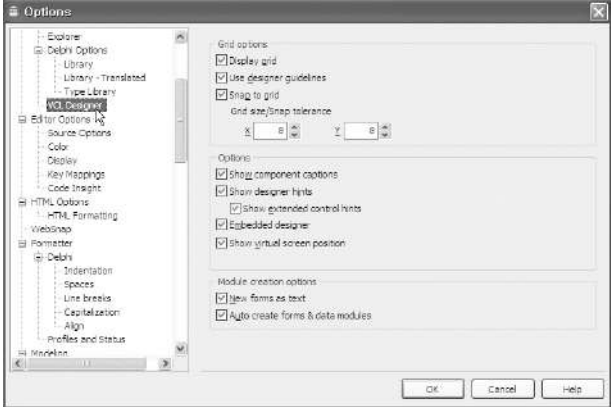


Tip

델파이의 프로젝트 이름은 Project1, Project2.. 소스 이름은 unit1, unit2, unit3 이런식으로 시리얼 번호로 명명됩니다. 현재 디렉토리에 unit1이 있으면 unit2.. 여러분이 프로그램 개발시에는 여러분의 명명 규칙에 따라 Save Project As로 이름을 바꾸어 저장해 주세요!

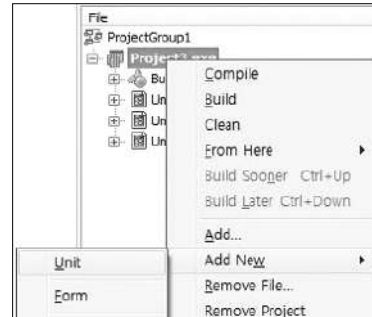
2. 프로젝트 구성요소 살펴보기와 프로젝트 관리

프로젝트 매니저화면에 .exe 파일과 .pas와 .dfm 파일이 생성된 것을 확인할 수 있습니다. 여기에서 프로젝트를 구성하는 파일들의 이름과 특징에 대해서 살펴보겠습니다.

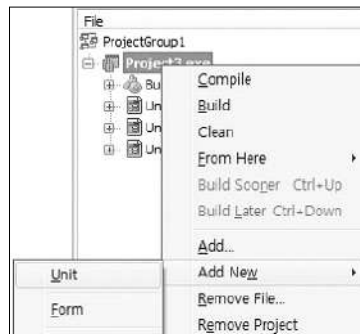
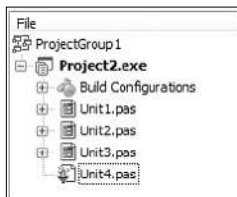
이름	설명
.DPR	프로젝트 파일로 작성한 어플리케이션의 메인 파일로 프로젝트의 총 연결파일입니다. 그러나 이곳에 구체적인 코딩은 많이 하지 않습니다.
.DPROJ	프로젝트 설정 파일의 형식과 확장자는 초기의 INI 파일에서 XML 파일로 그리고 다시 MSBuild 용 XML파일로 변경되었는데 그 설정 파일을 포함한 확장자입니다.
.DFM	<ul style="list-style-type: none"> 폼은 프로젝트의 시작 포인터로 설계 시 작성한 폼의 구체적인 정보가 저장되는 이진파일입니다. (예를 들면 색상, 크기, 위치, 폰트 등등) 델파이 5.0부터 텍스트로 저장되고 열어볼 수 있습니다. Tools → Options 메뉴를 선택하면 Options 대화 상자가 표시되는데 이곳의 VCL Designer 항목에 있는 'New Form as Text' 체크 박스 옵션을 이용해서 바이너리로 저장할지 텍스트 파일로 저장할지 결정할 수 있습니다  <ul style="list-style-type: none"> 이를 위한 보다 간단한 방법은 폼을 선택한 후 마우스 오른쪽 버튼을 클릭하고 컨텍스트 메뉴에서 Text DFM 메뉴 항목을 해제하거나 체크하는 것입니다.
.PAS	유닛 파일로 코딩의 최소 단위 소스이며 유형, 상수, 변수 그리고 프로시저와 함수들의 집합입니다. 유닛 파일을 개별적으로 컴파일 하여 .DCU(Delphi Compiled Unit)를 만듭니다.
.RES	프로젝트에서 사용하는 비트맵, 커서, 아이콘 또는 메시지 등을 포함하는 파일로 바이너리로 저장됩니다. 프로젝트 하나 당 프로젝트 이름과 동일한 리소스 파일이 자동으로 만들어집니다.
.DSK	윈도우가 열려 있는지 또는 그 위치 등의 프로젝트 상태 정보를 텍스트로 저장한 파일입니다.
.DOF	델파이 Project/Option에서 선택된 옵션 정보들이 저장되는 파일입니다

새로운 폼이나 유닛 추가하기

- 01** Project2.exe를 선택하고 마우스 오른쪽 버튼 클릭 → Add New → Form을 선택하거나 주 메뉴의 “File → New → Form”을 선택하여 현재 프로젝트에 새로운 폼을 추가할 수 있습니다. 동일한 방법으로 2개의 화면을 추가합니다.



- 02** 이번에는 현재 프로젝트에 Unit 파일만을 추가해 보겠습니다. 마우스 오른쪽 버튼 → Add New → Unit 을 선택하거나 메뉴의 “File → New → Unit”을 선택합니다. 총 폼 3개, 유닛 4개를 가지는 프로젝트를 확인할 수 있습니다.



- 03** 지금까지의 내용을 모두 저장합니다. File → Save All 혹은 단축키 Shift + Ctrl + S를 이용하여 저장합니다. Unit4.pas는 utest4.pas로 이름을 바꾸어 저장합니다. 각 이름을 utest3, utest2, utest1로 유닛 파일을 저장하고, 프로젝트명은 test.dproj로 저장합니다.





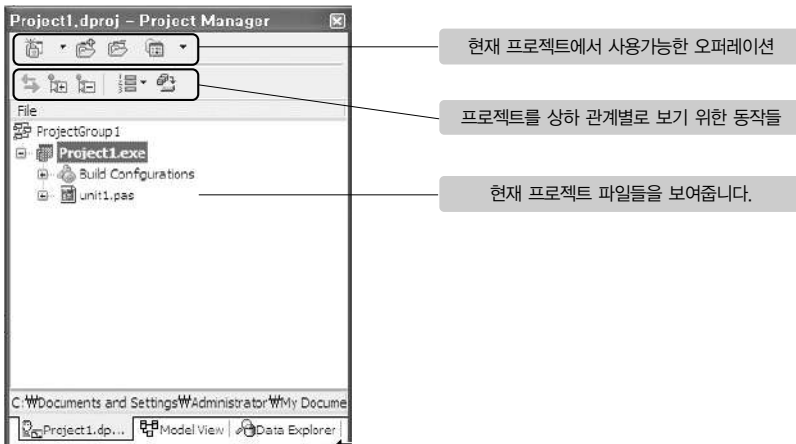
Tip

프로젝트 저장 시 새로운 유닛을 추가할 때 마다 Save Project As..를 사용하여 유닛 이름과 프로젝트를 저장합니다. 이전 버전의 경우는 저장하지 않은 파일들의 이름이 올림차순(unit1,unit2..)으로 표시되기 때문에 혼동이 올 수 있습니다.

이렇게 작성한 프로젝트의 파일들은 어떻게 소스 관리되고 컴파일하고 실행할까요?
전체 프로젝트를 관리하기 위해 프로젝트 매니저를 사용합니다. 간단하게 프로젝트 매니저의 사용방법에 대해서 살펴보겠습니다.

프로젝트 매니저

프로젝트 매니저는 델파이 오른쪽 상단에 위치하고 있는 델파이의 주요 개발 환경으로서 프로젝트를 구성하는 파일들을 나열하고 프로젝트의 유닛과 폼 파일의 추가 삭제가 가능하며 옵션 등을 설정 할 수 있습니다. 복수 개의 프로젝트를 그룹화하여 관리할 수 있으며 프로젝트내의 파일들을 드래그 - 드롭하여 복사 할 수 있으며 파일 탐색기에서 직접 선택하여 프로젝트 매니저로 가져올 수 있습니다.



■ 프로젝트 관리자의 톨바 버튼

메뉴	설명
Activate	경우에 따라서 여러 개의 실행 파일들이 그룹으로 묶일 경우 선택된 프로젝트를 활성화합니다
New	프로젝트에 새 항목을 추가할 수 있도록 델파이 New Item 대화상자가 표시됩니다
Remove	프로젝트로부터 파일을 제거합니다 하지만 디스크에서 없어지는 것은 아닙니다.
View	프로젝트 파일들을 보는 방법을 여러 가지로 제공합니다.
Sync	실제 프로젝트나 프로젝트 그룹이 저장될 때 프로젝트 매니저에 동기화 합니다.
Expand	현재 선택된 노드의 모든 하위(자식)노드들을 펼칩니다
Collapse	현재 선택된 노드의 모든 하위(자식)노드들을 접어 놓습니다.

■ 프로젝트 관리자의 컨텍스트 메뉴들

프로젝트 관리자의 마우스 오른쪽 버튼을 클릭하면 컨텍스트 메뉴가 나타납니다. 이 메뉴들을 이용해서 프로젝트 관리 작업을 간편하게 할 수 있습니다. 현재 프로젝트의 관리자에서 선택되어 있는 항목이 무엇인가에 따라서 나타나는 메뉴의 종류도 달라집니다.

▶ 파일을 선택했을 때의 메뉴

메뉴	설명
Open	파일을 열어 코드 에디터를 보여줍니다.
Show in Explorer	선택한 파일을 윈도우 탐색기를 통해 보여줍니다.
Remove from Project	프로젝트로부터 파일을 제거합니다 하지만 디스크에서 없어지는 것은 아닙니다.
Save	파일을 저장합니다.
Save As	파일을 새 이름으로 저장합니다.
Rename	파일의 이름을 다른 이름으로 변경합니다.

▶ 프로젝트 명을 선택했을 때의 메뉴

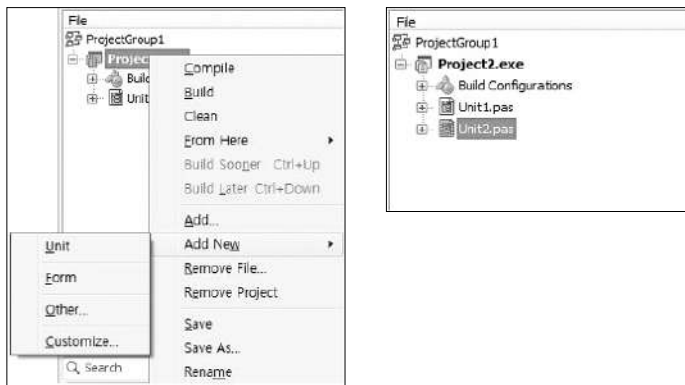
메뉴	설명
Add	프로젝트에 새 항목을 추가할 수 있도록 파일 열기 대화 상자가 나타납니다.
Remove File	프로젝트로부터 파일을 제거합니다. 파일 자체를 삭제하는 것이 아니라 프로젝트 그룹에서 제거하는 것을 의미합니다.
Save	파일을 저장합니다.
Options	프로젝트 옵션을 설정할 수 있도록 Project Options 대화 상자가 표시됩니다.
Activate	경우에 따라서, 여러 개의 실행 파일들이 그룹으로 묶일 경우 선택된 프로젝트를 활성화합니다. 예를 들어 Run을 실행하면 복수개의 프로젝트 중 현재 Active되어 있는 프로젝트가 실행됩니다.
Compile	최근 컴파일 이후 변경된 사항들만 다시 컴파일 합니다.
Build	변경 여부 상관 없이 무조건 컴파일 합니다.
View Source	프로젝트 소스(Dpr)내용을 보여줍니다.
Close	프로젝트를 닫습니다.
Remove Project	프로젝트 그룹으로부터 프로젝트를 제거합니다.
Run	프로젝트를 컴파일 하여 실행합니다.
Run Without Debugging	프로젝트를 디버깅을 사용하지 않고 실행합니다.
Build Sooner	복수 개의 프로젝트 관리 시 선택한 프로젝트를 먼저 컴파일 하도록 합니다.
Build Later	복수 개의 프로젝트 관리 시 선택한 프로젝트를 나중에 컴파일 하도록 합니다.
Build All From Here	프로젝트 그룹에 포함된 모든 프로젝트를 컴파일 합니다. 변경 여부에 상관없이 모두 컴파일 합니다.
Compile All From Here	프로젝트 그룹에 포함된 모든 프로젝트를 컴파일 합니다. 마지막 컴파일 이후 변경된 사항에 대해서만 컴파일 합니다.
Model Support	Together Tool의 Model Support를 지원받을 수 있습니다.

▶ 프로젝트 그룹 명을 선택했을 때의 메뉴

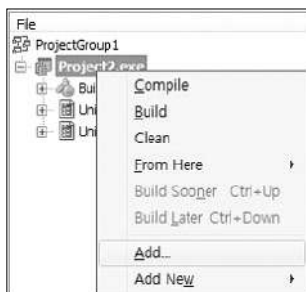
메뉴	설명
Add New Project	프로젝트 그룹에 새 프로젝트를 추가할 수 있도록 오브젝트 리포지토리를 불러옵니다.
Add Existing project	프로젝트 그룹에 기존에 만들어 둔 프로젝트를 추가할 수 있습니다.
Save Project Group	프로젝트 그룹을 저장합니다.(BPG).
Save Project Group As	프로젝트 그룹을 새 이름으로 저장합니다.

■ 새로운 유닛 / 폼 추가

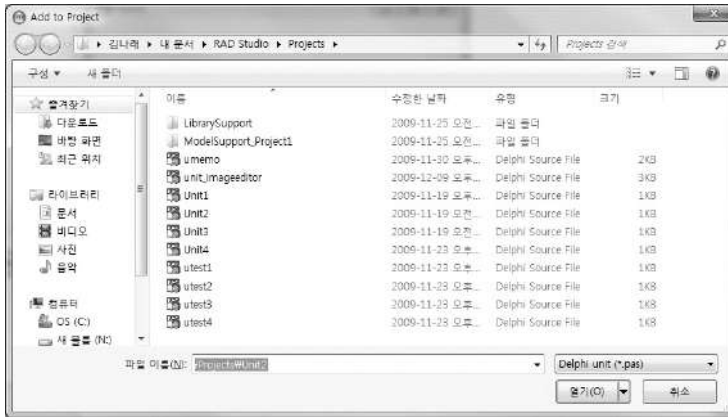
생성한 프로젝트에 새로운 폼이나 유닛 등을 추가할 수 있습니다. 프로젝트 이름을 선택한 후, 마우스 오른쪽 버튼을 이용하여 Add New를 선택합니다. 그리고 추가하기를 원하는 형식을 선택합니다.



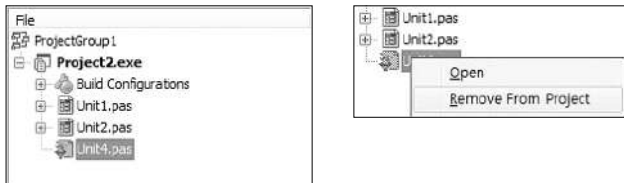
여기에서는 폼을 하나 더 생성하였습니다. Add New → Form을 선택하여 추가할 수 있습니다.



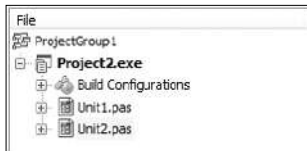
새로운 품 외에도 기존에 작업했던 내용을 추가할 수도 있습니다. 역시, 같은 방법으로 프로젝트 이름을 선택하고 마우스 오른쪽 버튼을 이용하여 Add...를 선택합니다.



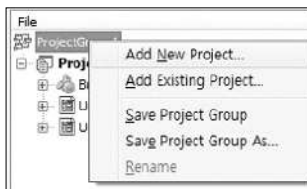
기존에 작업했던 파일 리스트가 표시됩니다. 추가하기를 원하는 파일을 선택합니다. 이전에 작업했던 Unit4라는 이름의 유닛을 추가해 보았습니다. 현재 작업중인 프로젝트 밑에 생성되는 것을 확인할 수 있습니다.



반대로 품을 삭제할 수도 있습니다. 삭제하기를 원하는 품을 선택한 후, 마우스 오른쪽 버튼을 이용하여 Remove From Project를 선택합니다. 추가했던 Unit4를 삭제해 보겠습니다.

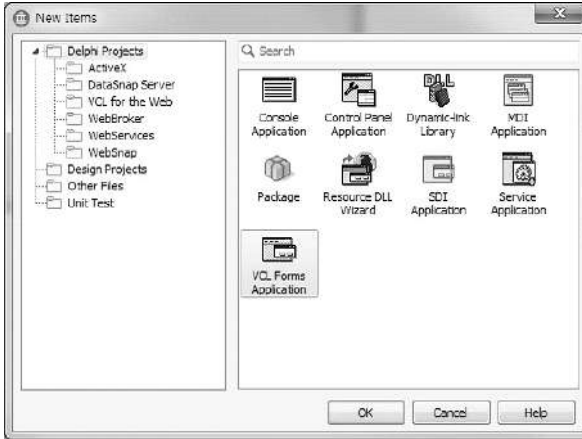


그림에서와 같이 Unit4가 프로젝트에서 제거된 것을 확인할 수 있습니다.

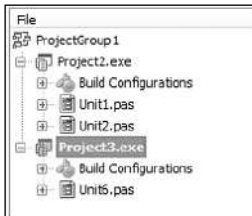


■ 복수 개의 프로젝트 관리

이번에는 프로젝트의 이름이 아닌, 가장 상단의 ProjectGroup을 선택하고 마우스 오른쪽 버튼을 이용하여 Add New Project ...를 선택합니다.

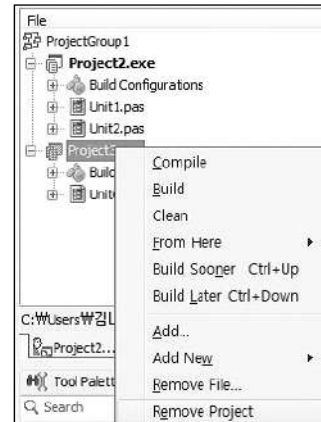


생성할 프로젝트에 대해 작업할 파일 형식을 선택할 수 있는 창이 뜹니다. VCL Forms Application을 추가해 보겠습니다.

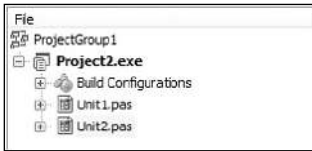


새로운 Project3와 그림에서 선택한 새로운 폼 Unit6가 함께 생성되었습니다.

작업중인 프로젝트가 2가지 이상일 경우, 컴파일 순서가 바뀔 수 있습니다. 먼저 컴파일 하기를 원하는 프로젝트의 팝업 메뉴에서 “Builder Sooner”를 선택합니다. 지금 아래의 상태에서 델파이의 실행 버튼(F9)를 누르면 어떤 프로젝트가 실행될까요? 볼드체로 활성화 되어있는 Project2가 실행됩니다. 해당 프로젝트를 활성화 하려면 프로젝트 이름을 더블-클릭하여 활성화할 수 있습니다.



이번에는 작업했던 프로젝트를 삭제해보겠습니다. 유닛을 삭제했을 때와 마찬가지로, 삭제하기를 원하는 프로젝트의 이름을 선택하고 마우스 오른쪽 버튼을 이용하여 Remove Project를 선택합니다.

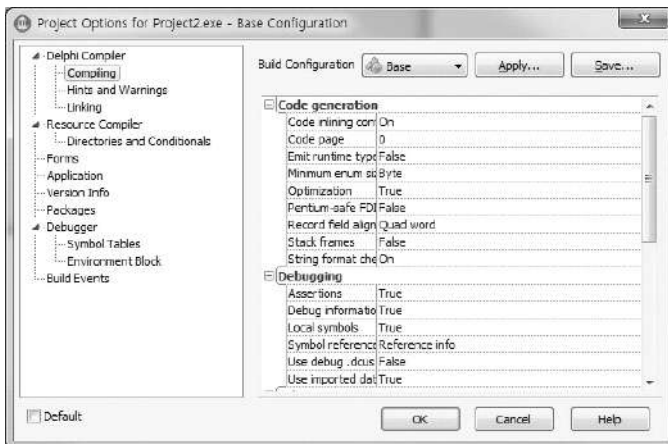


위에서 선택한 Project3 프로젝트가 삭제된 것을 확인할 수 있습니다.

■ 프로젝트 별 옵션 설정

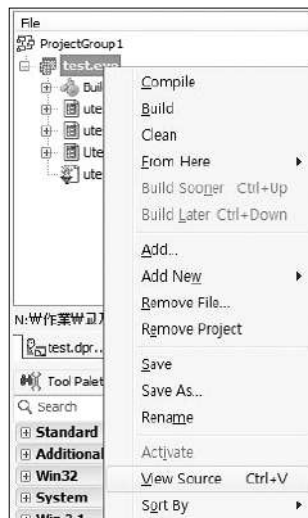
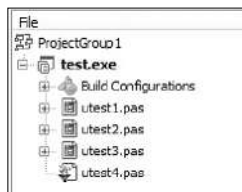
프로젝트 매니저는 여러 가지 기능을 가지고 있습니다. 프로젝트의 정렬 순서를 지정할 수도 있고, 프로젝트의 소스도 볼 수 있습니다. 또한 프로젝트 매니저를 이용하여 델파이의 옵션을 지정할 수도 있습니다.

프로젝트의 이름을 선택하고 마우스 오른쪽 버튼을 이용하여 Options...을 선택합니다.



델파이의 옵션을 설정할 수 있는 옵션 화면을 확인할 수 있습니다. 중요한 것은 이 대화 상자에서 변경하는 사항들은 현재 열려 있는 프로젝트에만 영향을 미친다는 것입니다. 왼쪽 밑에 있는 Default 체크 박스를 체크해야 다음에 시작하는 프로젝트들에 대해서 변경사항이 적용되도록 할 수 있습니다. 작업중인 프로젝트에 관한 옵션들을 원하는 대로 설정할 수 있습니다.

항목	설명
Forms	응용 프로그램의 Main Form을 지정합니다. 보통 프로젝트 시작 시 디폴트로 처음에 올라온 폼이 메인 폼으로 지정됩니다(필요 시 변경합니다). 응용프로그램 생성시 자동 생성될 폼과 원하는 시점에서 사용자에게 의해 생성될 폼 인지를 결정합니다. New Form 선택 시, 디폴트로 Auto Create Form으로 지정됩니다.
Application	<ul style="list-style-type: none"> · Title : 프로그램이 최소화 될 때 프로그램 아이콘 아래 표시될 문자입니다. · Help File : 응용프로그램에서 사용할 도움말을 설정합니다. · Icon : 응용프로그램을 표시하는 이미지를 설정합니다.
Packages	<ul style="list-style-type: none"> · Design Time Packages : 컴포넌트의 분배와 설치를 단순화하기 위해 지정합니다. · Run Time Packages : 프로그램의 실행 파일 사이즈를 줄이기 위해서 에디터 박스에서 컴파일 시에 선택한 패키지를 프로젝트에 자동으로 링크 할 수 있습니다. 패키지는 뒤 부분에서 좀 더 자세하게 설명하겠습니다.
Output settings	<ul style="list-style-type: none"> · Target File Extension : Target 실행 File이 갖는 파일 확장자입니다. 예를 들어, Active X Application인 경우 OCX로 정의되어야 합니다.
Compiler	컴파일러 옵션을 설정합니다.
Resource Compiler	볼랜드 리소스 컴파일러와 다른 리소스 컴파일러인 Microsoft Windows SDK의 컴파일러를 제공하여 선택 가능합니다.
Version Information	프로그램 소스를 버전 별로 관리합니다.
Directories/ Conditionals	델파이가 프로그램을 컴파일, 링크하는데 필요한 파일의 위치와 프로그램이 생성될 디렉토리로 지정합니다.
Build Event	MS Build를 이용할 수 있습니다. 빌드 설정, 빌드 이전, 이후 이벤트, 커맨드 라인 등을 설정합니다.

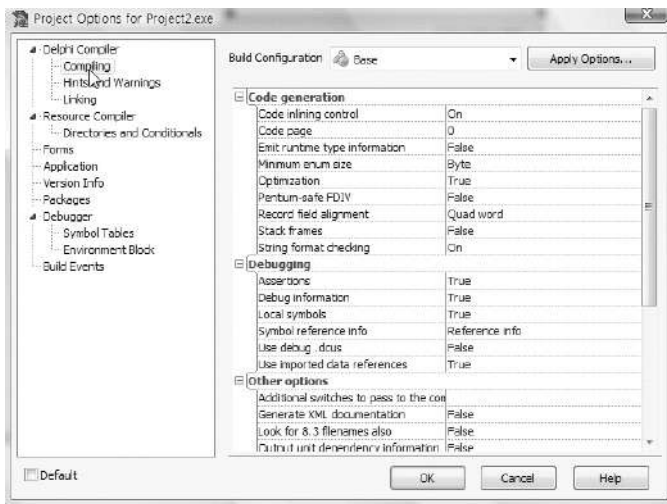


■ 빌드 설정(Build Configuration)

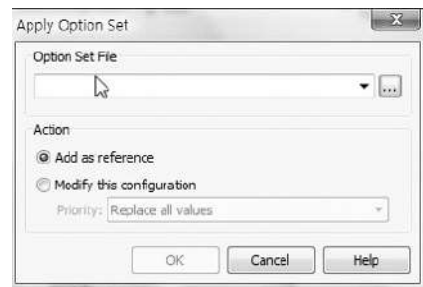
프로젝트 개발 시의 옵션 설정과 프로그램 개발이 끝난 후 릴리즈 버전의 옵션 설정 조건이 같을까요? 당연히 다릅니다. 예를 들면, 개발 시점에서는 디버거 정보들을 포함하는 옵션이 포함된다면 릴리즈 버전에는 디버거 정보가 포함되지 않도록 옵션을 설정해야만 합니다.

텔파이에서는 이런 옵션을 필요할 때마다 수정하는 것이 아니라 빌드 설정을 미리 만들어 놓고 상황에 맞게 적용합니다.

개발자는 프로젝트 최적화, 경로 검색, 기타 옵션 등을 컨트롤 하기 위해 다수의 빌드 설정을 만들 수 있습니다. 빌드 설정은 기본(base) 설정으로부터 옵션을 상속하고 특정 변경된 설정을 적용하여 재정의 될 수 있습니다. 빌드 설정은 프로젝트 매니저를 통해 액세스 될 수 있으며, 컨텍스트 메뉴에서 한번의 빌드 명령으로 프로젝트의 모든 설정을 빌드할 수 있습니다. 프로젝트의 옵션은 옵션 셋(Option Set)으로 저장될 수 있으므로, 동시에 다수의 프로젝트의 빌드 설정간에 공유할 수도 있습니다.



빌드설정매니저(Build Configuration Manager)는 현재 프로젝트 그룹내의 모든 프로젝트에 적용될 수 있는 모든 명명된 설정을 나열합니다. 이름으로 설정을 선택하고 모든 선택(Select All)을 클릭하고, 적용(Apply)하여 전체 프로젝트를 위해 필요한 설정을 쉽게 활성화 할 수 있습니다.



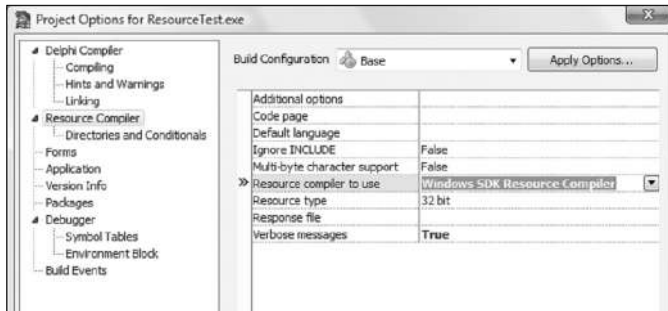
.OPTSET 파일은 .DPROJ의 포맷과 유사한 포맷의 XML 파일로서, 역시 MSBUILD XML 포맷을 기반으로 하며 OptionSet 프로젝트 타입입니다.

■ 리소스 컴파일러

이전 버전의 델파이에서는 Borland Resource Compiler(BRCC32.EXE)를 사용했습니다.

Delphi 2009에서부터 새 리소스 컴파일러, 더 정확히 말하자면 이전과는 다른 리소스 컴파일러인 Microsoft Windows SDK의 컴파일러를 제공합니다. 이는 윈도우가 다루는 모든 새로운 리소스 포맷들을 지원한다는 면에서는 장점이 많습니다.

프로젝트 옵션 다이얼로그의 Resource Compiler에서 해당 옵션을 설정하면 사용할 리소스 컴파일러를 선택하여 지정할 수 있습니다. (프로젝트 옵션 다이얼로그에서는 리소스 컴파일러의 몇 가지 파라미터들을 설정할 수도 있습니다.)



Windows SDK Resource Compiler는 SDK 컴파일러에 대한 프론트 엔드인 새 CodeGear Resource Compiler/Binder를 통해 호출됩니다. 그 외에 리소스 컴파일러 관련 기능의 새로운 점으로는, 이미지(바이너리) 데이터의 인라인 처리, 스트링 리스트에서 문자열 끝의 쉼표에 대한 지원, 문자열 처리 방식의 변경(C-언어 문자열로 취급되므로 파일 이름 내의 \를 두개의 백슬래시로 써야 함), 파일 인클루드 관련으로 폴더를 관리 방법의 변경 등이 포함되었습니다.

리소스 파일을 직접 사용해본 적이 없다면 이러한 변경 사항을 무시할 수도 있습니다. DFM 파일을 리소스로 포함하는 것부터 resourcestring 선언의 사용에 이르기까지, 델파이 환경에서 직접 관리되는 모든 작업들이 이전 버전과 완벽히 호환됩니다. 리소스를 직접 사용하는 경우 주의를 기울여 리소스 파일을 수정해야 합니다.

3. 프로젝트 파일 열어보기

프로젝트를 구성하는 파일의 프로젝트 파일에 대해서 살펴보도록 하겠습니다. 이 파일 안에는 폼과 유닛에 대한 정보와 프로그램을 실행을 위한 간단한 코드가 포함되어 있습니다.

프로젝트의 소스(.Dpr)를 확인해 보도록 하겠습니다. Test.exe 를 클릭한 후, 마우스 오른쪽 버튼 → View Source 혹은 단축키 Ctrl + V를 이용하여 프로젝트의 소스를 확인할 수 있습니다.

```
program Test; ①

uses ②
  Forms,
  utest1 in 'utest1.pas' { Form1 },
  utest2 in 'utest2.pas' { Form2 },
  utest3 in 'utest3.pas' { Form3 },
  utest4 in 'utest4.pas';

{$R *.res} ③

Begin ④
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.CreateForm(TForm3, Form3);
  Application.Run;
end.
```

- ① · Program 이라는 예약어와 함께 프로젝트 이름이 있습니다. 이 이름은 프로젝트 파일의 이름과 동일하며 프로그램 및 유닛은 같은 이름을 가질 수 없습니다.
- ② · Uses절에는 이 프로그램에 포함되어 있는 폼 유닛 파일과 이 유닛에서 필요로 하는 소스 파일들이 나열됩니다.
 - Forms 파일은 이 소스를 컴파일 할 때 필요한 유닛으로 델파이에서 제공하는 유닛입니다. (TApplicaiton 개체가 정의된 소스입니다.)
 - Utest1 in 'utest1.pas' {Form1}에서 utest1은 유닛의 식별자이며 utest1.pas는 유닛의 파일명, {Form1}은 utest1.pas와 연결된 폼 이름을 구분하기 위한 주석문입니다.
- ③ · 컴파일러에게 리소스 파일을 연결해서 컴파일 하라는 컴파일러 지시자입니다.



- ④ · 프로그램을 실행하면 이 부분부터 실행됩니다.
- Application.Initialize : 어플리케이션을 초기화하라는 메소드입니다.
 - Application.CreateForm(TForm1, Form1) : 어플리케이션의 폼을 메모리에 할당합니다.
 - Application.Run : 어플리케이션을 실행합니다.

소스 코드에서 알 수 있듯이 델파이는 오브젝트 파스칼이라는 언어를 사용합니다. 앞으로 각 소스 코드를 입력한 후, 그 코드에 대한 설명은 주석으로 처리하겠습니다.

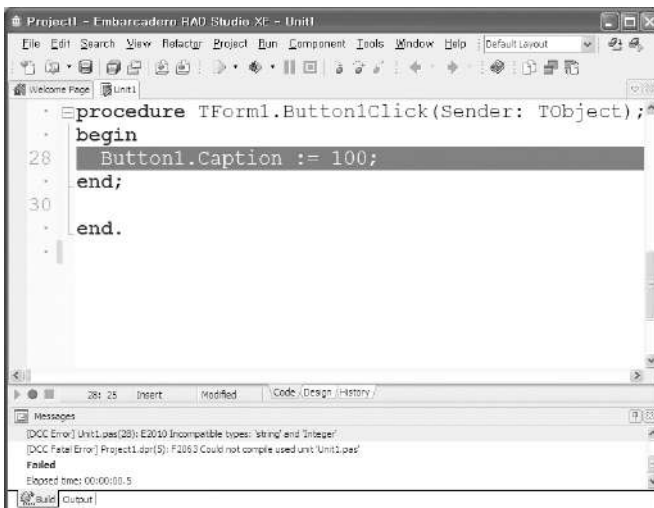
여기서 잠깐!

TApplication은 델파이의 컴포넌트로서 뒷장에서 언급할 개체와 컴포넌트에서 설명하겠지만 Forms.pas에 정의되어 있는 어플리케이션 당 할당되는 개체로서 어플리케이션을 컨트롤할 수 있는 여러가지 속성, 메소드, 이벤트를 제공합니다.

4. 프로그램 실행하기 - 컴파일과 실행

이 프로그램을 컴파일하고 실행하기 위한 메뉴를 설명하겠습니다. 컴파일을 하기 위해서는 Project → Compile 메뉴를 선택하거나 Ctrl + F9키를 누르면 됩니다. 만일 컴파일하고 바로 실행하고자 하는 경우는 Run → Run이나 F9 키 또는 툴 버튼 중에 을 클릭합니다. 만일 디버깅하지 않고 실행하려면 Run → Run Without Debugging이나 Shift + Ctrl + F9 키 또는 툴 버튼 중에 을 클릭합니다.

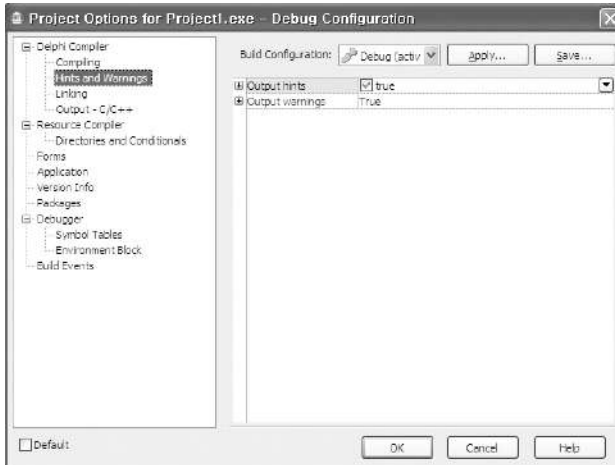
컴파일 하는 도중에 에러가 발생하면 에러가 발생한 코드에서 컴파일이 멈추고 코드 에디터 밑에 조그만 창이 하나 새로 생기면서 에러 내용이 표시됩니다.



좀 더 자세히!

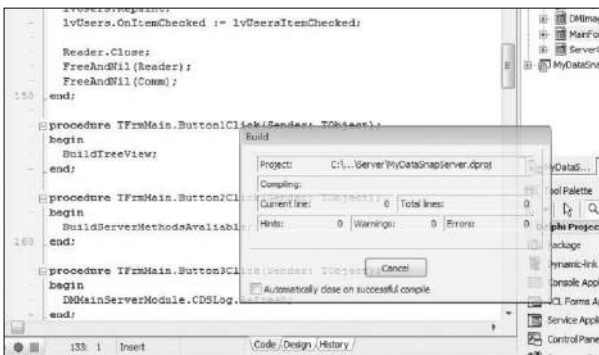
컴파일러 메시지 제어하기

컴파일러 메시지 중에는 에러 메시지 외에 다양한 경고(Warning) 메시지도 포함하고 있습니다. 이러한 경고 메시지 중 몇 가지 종류만 선택해서 나타나도록 설정 할 수 있습니다. Project → Options 메뉴를 선택한 후 Warning이나 Hint 체크를 False 값으로 설정하면 에러 메시지만 표시됩니다.



■ 백그라운드 컴파일

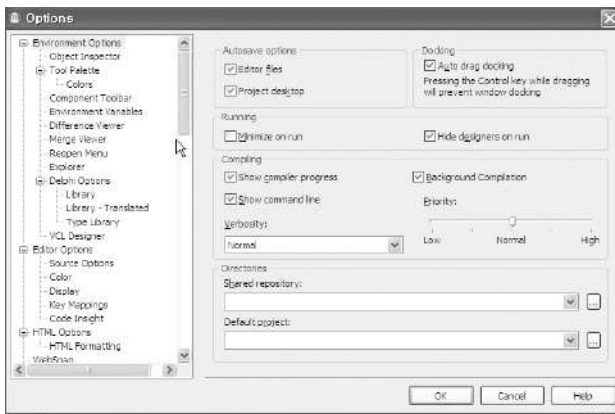
컴파일에 상당한 시간이 걸리는 대형 프로젝트 컴파일 작업을 위해 백그라운드 컴파일 기능이 추가되었습니다. 이 기능을 이용하면 컴파일 하는 동안 코드 에디트, 리뷰 등 다른 작업들을 계속할 수 있습니다. 컴파일 하기 직전의 유닛들의 메모리 스냅샷으로 컴파일 하므로 추가로 코드를 에디트 하더라도 영향을 받지 않습니다.



컴파일 창 보기

델파이로 처음 컴파일을 하게 되면 컴파일이 언제 시작하고 언제 끝났는지 알 수 가 없습니다 따라서 컴파일 하는 과정을 보기 위해서 다음과 같이 하면 됩니다.

- ❶ Tools → Options 메뉴를 선택합니다.
- ❷ Environment Options 항목을 선택합니다.
- ❸ Show Compiler progress 옵션을 체크합니다.
- ❹ 백그라운드 컴파일을 원하는 경우는 Background Compilation을 체크합니다.

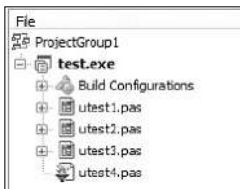


5. 유닛 소스 코딩 하기

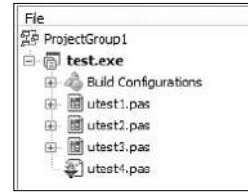
5.1 유닛 소스의 기본 구조

프로젝트 매니저에서 utest1.pas 앞의 ‘+’ 을 선택하면, 그림과 같이 utest1.dfm 파일이 보입니다. 이 파일을 더블-클릭하여 폼 화면으로 전환할 수 있습니다. 단축키 F12를 이용하여 폼 디자인 화면과 코드 에디터 화면을 전환할 수 있습니다.

델파이는 폼에 대한 기본 소스 코드를 제공해줍니다. 때문에 더욱 간편한 코딩이 가능합니다.



각 유닛에 대한 기본 소스 코드를 확인해 보겠습니다. 프로젝트 매니저에서 utest4.pas를 더블-클릭하여 utest4에 대한 소스를 확인할 수 있습니다.



왜 네 개의 유닛 중 굳이 utest4를 선택했을까요?

특별한 코딩이 없는 utest4를 선택하여 유닛의 구조를 살펴보고 직접 오브젝트 파스칼을 코딩해 보도록 하겠습니다.

```
unit utest4;
```

```
interface
```

```
implementation
```

```
Initialization
```

```
Finalization
```

```
end.
```

이 부분은 옵션부분입니다. 처음 소스에는 보이지 않으실 것입니다!

Unit이라는 예약어와 함께 유닛의 이름이 나옵니다. 프로젝트 소스의 경우에는 Program이라는 예약어를 사용하는 대신에 유닛은 unit이라는 용어를 사용합니다.

델파이의 구현부분은 Interface - Implementation - Initialization - Finalization 네 부분으로 나뉩니다. 다음은 각 부분에 대한 설명입니다.

부분	설명
Interface	타입, 상수, 변수, 프로시저, 함수 등을 선언하는 곳입니다. 여기에 선언된 내용은 현재 유닛을 사용하는 모든 프로그램 혹은 다른 유닛에서도 함께 사용할 수 있습니다. (단, Uses절을 사용했을 때)
Implementation	Interface에 선언된 프로시저나 함수를 구현하는 곳입니다. 변수, 상수 등을 선언할 수 있으며, Interface와는 달리 이러한 내용들은 이 내용이 선언된 현재 유닛에서만 사용이 가능합니다.
Initialization	초기처리를 위한 공간입니다. Interface에 정의된 데이터 구조의 초기화 및 필요한 자원 할당을 위한 문장을 기술합니다.
Finalization	Initialization에서 할당된 프로그램에서 사용했던 자원을 되돌리는 문장을 기술합니다.

이제부터 소스에 본격적으로 파스칼 코딩을 해보도록 하겠습니다.

코딩에 앞서 미리 알아야 할 사항들을 몇 가지를 정리하고 시작하겠습니다.

5.2 미리 알아야 할 사항들

■ 주석처리

프로그램 중간 중간에 이 코드가 무슨 일을 하는 것인지 등 코드에 대한 설명을 적어 놓는 것을 주석이라고 합니다. 프로그램 작성시 적당한 주석은 프로그램 관리상 꼭 필요합니다.

주석은 프로그램의 소스 코드에 대한 부연설명으로, 실행에는 영향을 끼치지 않도록 특수기호를 사용해야 합니다. 오브젝트 파스칼에서는 총 3가지의 형태로 주석을 처리할 수 있습니다.

```
{ 중괄호 }  
(* 괄호 *)  
// 한 줄 주석
```

중괄호 '{}' 를 이용한 주석처리는 여러 줄에 걸쳐서 사용할 수 있습니다. 주석 처리를 원하는 시작 지점에서 중괄호를 열고, 끝내기를 원하는 지점에서 중괄호를 닫으면 중괄호 안의 모든 내용을 주석처리 할 수 있습니다.

괄호와 별을 이용한 '(*)' 주석 처리 역시 중괄호와 마찬가지로 여러 줄에 걸쳐서 사용할 수 있습니다. 마지막은 다른 언어와 같은 형태 입니다. '/' 표시가 포함된 그 줄만 주석 처리 됩니다.

■ 세미콜론(;)과 점(.) begin..end;

오브젝트 파스칼에서 세미콜론은 코드 한 줄이 끝났다는 것을 말해주는 기호입니다.

코드가 길어지면 하나의 문장을 여러 라인에 걸쳐서 작성할 수 있습니다.

점(.)은 유닛 혹은 프로그램의 끝이란 것을 표시하기 위해 사용됩니다. 하나의 유닛의 맨 마지막은 end. 으로 되어 있음을 확인 할 수 있습니다.

Begin..end는 코드의 한 블록을 감싸서 다른 블록과 구분하기 위해서 사용합니다.

다시 정리하면 다음과 같습니다.

- 모든 문장은 세미콜론으로 구분됩니다.
- Begin..end; 는 한 쌍으로 사용됩니다.
- end; 앞에서의 세미콜론은 생략 가능합니다.
- 유닛의 마지막 end 뒤에는 점(.)을 추가해서 유닛의 마지막임을 표시합니다.

■ 문장(Statement)

델파이 프로그램 소스 코드 부분은 '실행 가능한 코드 라인' 들로 구성되어 있습니다. 이 실행 가능한 코드 라인을 Statement 라고 합니다. 문장에는 단순문과 복합문이 있는데 단순문은 한 라인으로 이루어진 것을 말하며 복합문은 begin..end사이에 여러 문이 들어가서 하나의 블록으로 만든 것 입니다.

■ 수식(expression)

‘:=’ 기호는 오른쪽에 있는 어떤 값을 왼쪽에 저장 혹은 넣을 때 사용합니다 이를 할당 (Assignment)이라고 합니다.

■ 예약어(Keyword)

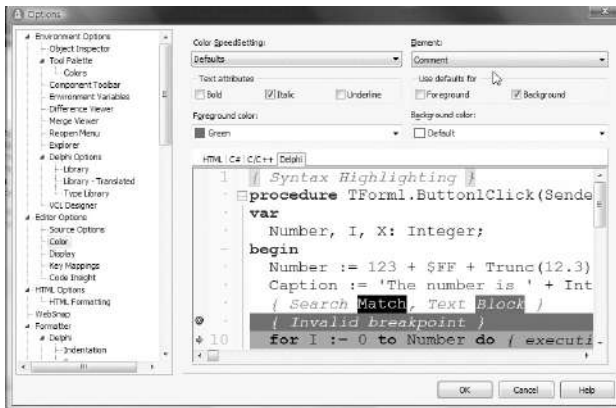
특별한 목적을 위해서 컴파일러가 독립적으로 사용하도록 미리 정해져 있는 단어들을 말합니다. 이러한 예약어들은 여러분 임의로 사용하실 수 없습니다. 이런 예약어들은 코드 에디터에서 굵은 글씨체로 표시되기 때문에 쉽게 알아 볼 수 있습니다.



Tip

키워드 또는 주석 색상 변경하기

- 1 Tools → Options → Editor Option → Color를 선택합니다.
- 2 Element에서 색상을 변경하고 싶은 아이템을 선택합니다.
- 3 기본값으로 설정되어 있는 Foreground Color와 Text Attribute를 변경합니다.
- 4 OK 버튼을 클릭하면 코드에디터에 설정한 값이 적용되어 있음을 확인 할 수 있습니다



5.3 변수 선언

파스칼이 제공하는 다양한 타입의 변수를 선언해 보겠습니다.

변수란, 다루고 싶은 어떤 데이터나 값을 저장하기 위한 메모리 상에 마련되는 공간입니다.

메모리의 일정 부분에 이름을 부여하고, 값을 지정할 때는 이 이름을 이용해서 그 메모리 영역에 값을 넣어줍니다. 이를 ‘할당’이라고 합니다. 변수를 사용하기 위해서는 변수를 선언해야 합니다.

변수를 선언한다는 것은 컴파일러에게 이런 이름과 데이터 타입을 갖는 변수를 사용할 테니 컴퓨터 메모리에 그 만큼 공간을 확보해 달라는 의미입니다.

델파이 유닛 소스에서 변수를 선언할 수 있는 곳은 아래 표시된 세 부분 밖에 없습니다.

```
unit utest4;

interface
{ 변수 선언 }
Var _____ ❶ 이 소스를 Uses하면 사용할 수 있는 변수
  i : integer;

implementation
Var _____ ❷ Implementation 영역에서만 사용할 수 있는 변수
  j :integer;

Procedure aaa;
Var
  K:integer; _____ ❸ 이 루틴에서만 사용할 수 있는 지역변수
Begin
  .....
End;
end.
```

■ 변수 선언 방법

```
변수명 : 자료형;
```

오브젝트 파스칼에서의 변수 선언 형식은 다른 언어와는 다릅니다. 변수가 필요할 때마다 변수를 선언하는 다른 언어와는 달리(C 언어) 오브젝트 파스칼의 경우에는 var 부분에 변수들을 미리 선언해야 합니다.

```
{ 변수를 미리 선언하는 형식 }
x, y : Integer;
s : String;
f : Double;

{ 변수 선언과 동시에 초기화하는 형식 - 프로시저, 함수에서는 사용할 수 없습니다. }
i : Integer = 70;
t : String = 'Hello World';
d : Double = 8.5;
```

```
{ 위에서 미리 선언한 변수의 값을 초기화 }
begin
  x := 10;
  y := 20;
  s := 'Delphi';
  f := 11.23;
end;
```

- 같은 종류의 변수를 나열하는 경우에는 콤마(,)를 사용합니다.
- 변수 이름은 255자를 넘을 수 없습니다.
- 변수 이름은 영어 알파벳이나 숫자, 밑줄(_)만 사용해야 합니다. 대소문자 구별은 없습니다.
- 변수 이름에서 첫 번째 글자는 반드시 문자이어야 합니다.
- 예약어들은 변수 이름이 될 수 없습니다.

여기에서 파스칼의 자료형을 간단히 살펴보도록 하겠습니다.

■ 정수형

▶ **Fundamental Type** : 시스템에 상관없이 크기와 값의 범위가 정해져 있는 형입니다.

타입	범위	포맷
ShortInt	-128..127	Signed 8-bit
SmallInt	-32768..32767	Signed 16-bit
LongInt	-2147483648..2147483647	Signed 32-bit
Int64	$-(2^{63})..(2^{63})-1$	Signed 64-bit
Byte	0..255	Unsigned 8-bit
Word	0..65535	Unsigned 16-bit
LongWord	0..4294967295	Unsigned 32-bit

▶ 16-bit Integer

타입	범위	포맷
Integer	-32768..32767	Signed 16-bit

▶ 32-bit Integer

타입	범위	포맷
Integer	-32768..32767	Signed 16-bit

■ 실수형

▶ Fundamental Type

타입	범위	DIGITS	사이즈/바이트
Real48	$2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$	11-12	6
Single	$1.52 \times 10^{-45} \dots 3.4 \times 10^{38}$	7-8	4
Double	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15-16	8
Extended	$3.4 \times 10^{-4932} \dots 1.1 \times 10^{4932}$	19-20	10
Comp	$-2 + 1.2^{63} - 1$	19-20	8
Currency	-922337203685477.5808.. 922337203685477.5808	19-20	8

▶ Generic Real Type : 운영체제에 따라 크기가 달라질 수 있는 형

타입	범위	DIGITS	사이즈/바이트
Real	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15-16	8

■ 문자열 형

문자열 혹은 스트링이란, 연속적인 문자들의 집합을 말합니다. 문자열형은 어포스트로피(')를 이용해서 문자열형 변수에 저장합니다. 문자열을 서로 더할 때는 '+' 연산자를 사용합니다.

TYPE	초대길이	필요메모리	용도
ShortString	255문자	2 ~ 256바이트	하위호환성
AnsiString	$\sim 2^{31}$ 문자	4바이트 ~ 2GB	8비트(ANSI)문자, DBCS ANSI, MBCS ANSI, 유니코드문자등
WideString	$\sim 2^{30}$ 문자	4바이트 ~ 2GB	유니코드문자, 멀티 유저 서버 및 멀티언어 어플리케이션
UnicodeString	$\sim 2^{30}$ 문자	4바이트 ~ 2GB	유니코드문자, 8비트(ANSI)문자, 멀티 유저 서버 및 멀티언어 어플리케이션

■ 포인터 타입

C 나 C++언어에서 문자열의 끝을 표시하기 위하여 항상 널(null #0)이란 것을 문자열 끝에 추가합니다. 이런 것을 널 종료 문자열이라고 합니다. 널 종료 문자열과의 호환성을 위해서 마련한 것이 PChar형입니다. 즉, 문자열에 대한 포인터를 말합니다.

```
var
    P:PChar;
begin
    P := 'hello';
End;
```

```
Procedure TForm1.Button1Click(Sender:TObject);
Var
    x,y:integer;
    p:^integer;
Begin
    x := 17;
    y := 0;
    p := @x;
    y := p^;
    ShowMessage(IntToStr(y));
end;
```

포인터를 지정할 때 @연산자를 사용했는데, @연산자는 포인터 변수에게 주소를 넘겨 주도록 하는 연산자입니다. 반대로 포인터의 주소가 가리키는 값을 넘겨주는 연산자는 캐럿(^)이라고 합니다.

이 기호는 특정 타입의 변수를 선언할 때도 사용됩니다.

■ 가변형(Variant)

- 일반 변수의 형이 컴파일 시에 확정되는 것과는 달리 Variant는 실행 중 자료형이 바뀔 수 있습니다.
- 정수, 실수, 문자열, Boolean, 날짜, 동적 배열, OLE Automation Object 등을 보관할 수 있습니다.
- 가변형은 Unassigned와 Null이라는 예약어가 있는데, Unassigned는 가변형 변수에 값이 할당되지 않았다는 것을 나타낼 때 사용되고, Null은 이름이 틀리거나 없는 데이터를 가리키는 Variant 값입니다
- 가변형 변수를 선언하면 메모리 16 바이트를 차지하기 때문에 프로그램 실행속도를 저하 시킬 수 있습니다.

```
Var
    V1, V2, V3, V4, V5 : Variant;
    I:integer;
    D:Double;
    S:String;
```

```
begin
    V1 := 1;
    V2 := 1234.5678;
    V3 := 'Hello World'
    S := V3;
    V1 := v1 + v2;
end;
```

■ BOOLEAN 형

불린(BOOLEAN) 형은 참과 거짓을 표시하기 위한 것으로, True 혹은 False 값만을 가질 수 있는 자료형입니다.

Boolean	크기
Boolean(기본)	8bit
ByteBool	8bit
WordBool	16bit
LongBool	32bit

[illegible]

01 Utest4에 아래와 같이 변수를 선언합니다.

```
unit Utest4;

interface

var
    i:integer;
    s:string;

implementation

var
    j:integer;

end.
```

■ 배열

- ### ▶ 1차원 배열

▶ 다차원 배열

```
04) var
    MyArray :[ 0..2,0..3] of string;
    ...
    MyArray[ 0,0] := 'a';
```

||||| **따 라 하 기** |||||

- ```
unit utest4;
interface
{ 정적배열 : 0 부터 2까지의 공간을 가지는 정적인 배열을 지정합니다. }
type
 Country = array[0..2] of string;
```

```

var
 i : integer;
 s : string;
 Countries : Country;
 // type을 선언하지 않고 Countries : array[0..2] of string로도 선언할 수도 있습니다.

implementation
initialization
{ 할당된 정적 배열 Countries에 초기값을 Assign합니다 }
begin
 Countries[0] := 'Korea' ;
 Countries[1] := 'Japan' ;
 Countries[2] := 'Etc' ;
end;
end.

```

#### ▶ 동적 배열

- 동적 배열(Dynamic Array)이란 고정된 크기를 가지고 있지 않은 배열을 말합니다. 앞에서 설명한 배열은 정적 배열(Static Array)이라고 하는데 컴파일 할 때 미리 배열의 크기가 정해집니다.
- 자료형만 정의하고 배열 요소수는 정의하지 않습니다.
- 런타임에 할당하려는 값에 따라 메모리를 할당합니다.
- 동적 배열은 SetLength 프로시저를 이용하여 할당합니다.

```

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

```

## ||||||| **따 라 하 기** ||||||||||||||||||

**01** 변수 부분에 a,b 변수를 추가합니다.

**02** 프로시저 SetLength를 이용하여 각 배열의 공간을 할당합니다.

```

interface

var
 a : array of string;
 b : array of array of string; ┌─ 변수 부분에 추가합니다.

implementation

initialization
{ 할당된 정적 배열 countries에 초기값을 Assign합니다 }
begin
 countries[0] := 'Korea';
 countries[1] := 'Japan';
 countries[2] := 'etc';
 SetLength(a,2); //a값에 대한 공간을 두 칸 생성합니다.
 SetLength(b,2,2); //b값에 대한 공간을 네 칸 생성합니다. (0,0) (0,1) (1,0) (1,1)
 a[0] := 'a';
 b[0,0] := 'b';

end;
end.

```



#### Tip

여기에서 **SetLength**를 사용하지 않고 a,b 변수를 사용하는 경우에는 “Access Violation Error” 오류가 발생합니다. 델파이에서는 메모리에 할당되지 않은 개체나 Type들에 접근하는 경우 이런 오류가 발생합니다.

## 5.5 레코드형 선언

- 다양한 자료형을 연속된 기억공간에 보관합니다.
- 모든 레코드 요소는 다른 자료형을 가질 수 있습니다.
- 크기가 서로 다른 자료형의 레코드 멤버가 연속된 기억공간을 사용합니다.

레코드형 = **Record**

필드명 : 자료형;

필드명 : 자료형;

**end;**

## 01 아래와 같이 레코드를 Type에 추가합니다.

```
unit utest4;

interface

type
Country = array[0..2] of String;

{ 레코드형 자료관리 }
Person = record
 Name : String;
 Age : Integer;
 Address : String;
end;
...
```

Delphi 2007 버전 이후로 레코드에 메소드 포함 기능이 추가되었습니다. 레코드 선언에서 필드 외에도 프로퍼티, 메소드, 클래스 프로퍼티, 클래스 메소드, 클래스 필드, 네스티드(nested) 타임을 가질 수 있습니다.

```
interface

type
TMyRecord = record
 type
 TInnerColorType = Integer;

var
 Red:Integer;
 class var
 Blue:Integer;

procedure printRed();
constructor Create(val:Integer);
property RedProperty: TInnerColorType read Red write Red;
class property BlueProperty:TInnerColorType read Blue write Blue;
end;
```

```

implementation

{ TMyRecord }

constructor TMyRecord.Create(val: Integer);
begin
 Red := val;
end;

procedure TMyRecord.printRed;
begin
 writeln('Red: ', Red);
end;

end.

```

## 5.6 프로시저와 함수 선언

이제 프로시저와 함수를 선언하고 코딩해 보도록 하겠습니다. 프로시저와 함수는 프로그램 안에 포함된 어떤 일을 수행하는 루틴이라는 점에서는 동일하지만 반환 값이 있는 루틴은 함수, 반환 값이 없는 루틴은 프로시저라고 지정합니다. 만일 이 약속이 지켜지지 않으면 컴파일 오류가 생깁니다.

### ■ 프로시저

- Object Pascal에서 가장 일반적인 서브-루틴 유형입니다.
- 프로그램을 기능별로 분리하여 코딩을 더욱 간편하게 합니다.
- 프로시저를 사용하기 위해서는 선언과 구현이 필요합니다.
- 프로시저 선언이란 유닛의 Type 부분에 프로시저 원형을 적어주는 것입니다. 즉 이러한 프로시저를 사용하겠다고 선언하는 것입니다.
- 프로시저 선언은 유닛의 interface 부분에 선언합니다.

```

procedure 프로시저 이름 (<매개변수이름> : <매개변수타입>, <매개변수이름> : <매개변수타입>);

```

- 프로시저 구현이란, 유닛의 implementation 부분에 그 프로시저가 구체적으로 수행할 일을 작성하는 것입니다.



```

procedure 프로시저이름 (<매개변수이름> : <매개변수타입>, <매개변수이름> : <매개변수타입>);
const
 상수명 = 리터럴값 ;
var
 변수명 : 자료형 ;
begin
 Statement 1;
 Statement 2;
end;

```

```

procedure clear (var A : array of Double);
var
 I : Integer;
begin
 for I := 0 to High(A) do A[I] := 0;
end;

```

## ■ 함수

- 리터럴 값을 반환한다는 점을 제외하고는 기본적으로 프로시저와 같습니다.
- 함수도 프로시저와 마찬가지로 먼저 선언하고 구현해야 합니다.
- Procedure 대신에 Function 예약어를 사용합니다.
- 반환값은 지시어 Result를 사용하여 반환합니다.

```

function 함수명 (매개변수이름 : 매개변수타입) : 반환 타입;
const
 상수명 = 리터럴값;
var
 변수명 : 자료형;
begin
 statement1;
 statement2;
end;

```

```

function Add(x, y: integer) : integer;
begin
 Result := x + y ; // Add := x + y; 도 코딩 가능합니다(c적인 코딩)
end;

```

## ■ 매개변수

매개변수 혹은 파라미터는 프로시저와 함수를 호출할 때 값을 전달하기 위한 것입니다.

### [매개변수 전달 방식]

#### ❶ Call by Value

변수를 값으로 전달하는 방식으로, 컴파일러는 값을 복사하여 원래의 값이 아니라 복사한 값이 전달됩니다. 함수 안에서 넘어온 변수값을 변경할 수 없습니다.

```
Procedure Add(x, y : integer);
```

#### ❷ Call by Reference

변수의 메모리 위치를 프로시저에 전달하는 방식으로, 프로시저는 해당 메모리의 위치의 내용을 변경할 수 있습니다. 루틴 안에서 변수값을 변경할 수 있습니다.

```
Procedure ReadData(Var Rec: String);
```

#### ❸ Call by Const

상수로 매개변수를 전달하는 방식으로, 프로시저 내에서 변수값은 변경할 수 없고 참조만 가능합니다.

```
Procedure WriteData(Const A: String);
```

#### ❹ 출력 파라미터

Out 파라미터는 참조에 의해 전달되는 방식입니다. Out 파라미터에서 참조되는 변수의 초기값은 루틴으로 전달될 때 값이 없어집니다. Out 파라미터는 출력용으로만 사용됩니다. 즉 함수나 프로시저 실행 후 어떠한 값을 전달하기 위해 사용되지만 호출시 넘어온 입력 정보는 알려 주지 않습니다.

```
Procedure Writedata(Out a:String);
```

## ■ Overload 루틴

- 하나의 유닛에는 같은 이름의 함수나 프로시저를 선언할 수 없습니다.
- Overload를 사용하면 동일한 범위에서 같은 이름으로 하나 이상의 프로시저와 함수를 선언할 수 있습니다.
- 호출 시 파라미터 리스트의 차이에 의해서 구분됩니다.

```
Function divide(x, y : Integer) : Integer; Overload;
Function divide(x, y : Real) : Real; Overload;
// overload 하지 않으면, 위와 아래에 선언된 함수명(divide)이 같으므로 에러 발생
```

**따 라 하 기**

---

## 01 프로시저 test를 Interface 부분에 선언합니다.

```
interface
var
 i : Integer;
 s : String;
 a : Array of String;
 b : Array of Array of String;
 countries : country;

procedure test;
```

**02** 이 상태에서 컴파일을 하면 어떤 메시지가 표시됩니까? “Unsatisfied Forward or External Declaration :Test”가 표시됩니다.

### 03 이 오류를 해결하기 위해서는 Implementation에 Test 프로시저를 구현해야 합니다.

```
implementation
{ 프로시저 선언 및 프로시저에서 사용할 변수를 선언합니다. }
procedure test;
var
 k : Integer;

begin
 ShowMessage('Hello'); // Hello를 출력합니다.
end;
```

## 04 ShowMessage 실행을 위해 필요한 Dialogs를 uses 절에 추가합니다.

```
interface
uses Dialogs;
var
 i : Integer;
 s : String;
 a : Array of String;
 b : Array of Array of String;
 countries : country;
```

## 05 Divide 라는 함수를 각각 매개변수 타입을 달리하여 선언합니다. 같은 유닛에는 같은 이름의 프로시저나 함수를 선언 할 수 없으므로 Overload라는 지시어를 사용하여 선언합니다.

```
interface
function divide(x , y : Integer) : Integer; Overload;
function divide(x , y : Real) : Real; Overload;
```

## 06 Implementation 부분에 두 개의 함수를 구현합니다.

```
{ result 값에 x와 y의 나눈 값을 넣습니다.(정수형) }
function divide(x , y : Integer) : Integer;
begin
 Result := x div y;
end;

{ result 값에 x와 y의 나눈 값을 넣습니다.(실수형) }
function divide(x , y : Real) : Real;
begin
 Result := x / y;
end;
```

여기에서 델파이에서 사용하는 연산자를 정리 하도록 하겠습니다.

## 5.7 연산자

연산자(Operator)는 변수에 넣은 값을 계산하거나 비교할 때 사용할 수 있는 기호들을 의미합니다.

### ■ 산술연산자

| Operator | Operation        | Operand Types     | Result Type       |
|----------|------------------|-------------------|-------------------|
| +        | Addition         | integer/real type | integer/real type |
| -        | Subtraction      | integer/real type | integer/real type |
| *        | Multiplication   | integer/real type | integer/real type |
| /        | Division         | integer/real type | integer/real type |
| Div      | Integer division | integer type      | integer type      |
| Mod      | Remainder        | integer type      | integer type      |

### ■ 논리연산자

| Operator | Operation           | Operand Types | Result Type |
|----------|---------------------|---------------|-------------|
| Not      | Bitwise negation    | integer type  | Boolean     |
| And      | Bitwise and         | integer type  | Boolean     |
| Or       | Bitwise or          | integer type  | Boolean     |
| Xor      | Bitwise xor         | integer type  | Boolean     |
| Shl      | Bitwise shift left  | integer type  | Boolean     |
| Shr      | Bitwise shift right | integer type  | Boolean     |

### ■ String 연산자

| Operator | Operation     | Operand Types                                       | Result Type |
|----------|---------------|-----------------------------------------------------|-------------|
| +        | Concatenation | String type<br>Character type<br>Packed string type | string type |

### ■ Set 연산자

| Operator Type | Operation    | Result Type      |
|---------------|--------------|------------------|
| +             | Union        | compatible types |
| -             | Difference   | compatible types |
| *             | intersection | compatible types |

## ■ 관계형연산자

| Operator Type | Operation                | Result Type |
|---------------|--------------------------|-------------|
| =             | Equal                    | Boolean     |
| <>            | Not equal                | Boolean     |
| <             | Less than                | Boolean     |
| >             | Greater than             | Boolean     |
| <=            | Less than or equal to    | Boolean     |
| >=            | Greater than or equal to | Boolean     |
| <=            | Subset of                | Boolean     |
| >=            | Superset of              | Boolean     |
| in            | Member of                | Boolean     |

## ■ @연산자

@연산자는 변수, 프로시저, 함수, 또는 메소드의 주소를 계산하기 위해 사용됩니다



### Tip

#### 연산자 우선순위

연산자들의 우선순위에 따라 그리고 결합 순서에 따라 연산이 이루어지기 때문에 주의해야 합니다. 먼저 계산하고자 하는 것이 있다면 괄호를 이용해서 묶어 주셔야 합니다.

| 우선순위 | 연산자들             | 결합순서     |
|------|------------------|----------|
| 1    | *, / , div , mod | 왼쪽에서 오른쪽 |
| 2    | +, -             | 왼쪽에서 오른쪽 |
| 3    | =, <, <=, >, >=  | 왼쪽에서 오른쪽 |
| 4    | And, or, not     | 왼쪽에서 오른쪽 |
| 5    | :=               | 오른쪽에서 왼쪽 |

## 5.8 클래스 작성

이제 델파이에서 조립식 개발을 할 수 있는 로직의 실체인 개체를 직접 코딩 하여 보겠습니다. 지금 개체를 직접 작성하여 보는 것은 델파이의 구조적인 이해를 돕고 앞으로 사용할 컴포넌트를 쉽게 사용할 수 있도록 도와 드릴 것입니다. 그에 앞서, OOP프로그램의 특징을 먼저 살펴 보도록 하겠습니다.

### 5.8.1 OOP (Object-Oriented Programming. 객체지향프로그래밍)

객체지향프로그래밍이란, 데이터를 개체화하여 프로그래밍 하는 방식을 말합니다. 이 방식을 통해 복잡한 문제를 프로그램으로 구축할 수 있습니다. 객체지향프로그래밍의 대표적인 기본 구성 요소는 개체(오브젝트)와 클래스입니다.

#### ■ 클래스

클래스는 속성과 행위를 갖는 레코드형과 비슷한 일종의 자료형(Type)입니다. 클래스 형으로 정의한 변수는 그 자체가 개체(오브젝트)가 되는 것이 아니라 메모리에 자리잡기 위해서는 인스턴스 하는 작업이 필요한데 이를 생성(Create)이라고 합니다. 내부적으로 데이터와 메소드를 가지고 있습니다. 클래스는 Type 절에서 선언합니다. 클래스를 선언할 때는 예약어 class를 사용하며 괄호 안에 계승 받을 선조 클래스를 표시합니다. 조상클래스를 생략하면 가상 상위의 클래스 TObject에서 계승 받는 것을 의미합니다.

```
type
 클래스 이름 = class (조상 클래스)
 멤버 선언
end;
```

TPerson 클래스를 선언한 형태입니다.

```
type
 TPerson = Class { TPerson 클래스의 선언 }
 Name : string;
 Age : byte;
 Address:String;
end;

var
 Person : TPerson; { TPerson 형의 변수 Person선언 }
```



#### Tip

##### Sealed 클래스

sealed로 표시된 클래스는 더 이상 상속이 불가능합니다.

```
TSealedClass = class sealed (TabstractClass)
```

## ■ 개체

메모리에 생성된 오브젝트가 바로 개체 또는 인스턴스입니다. 클래스가 봉어빵을 만드는 틀이라면 개체는 구워져 나온 봉어빵과 같습니다. 개체는 자신만의 데이터를 가지며 클래스의 행위를 수행합니다.

## ■ 필드

클래스의 개체 상태를 나타내는 데이터로서 클래스에 속해있는 일종의 변수들입니다.

## ■ 메소드

개체에서 실행할 수 있는 프로시저나 함수입니다. 클래스에 선언된 메소드가 동작할 때는 오브젝트가 생성된 이후가 됩니다. 메소드를 호출하면 그 메소드가 실제로 참조할 필드들이 위치해 있는 인스턴스 정보가 넘겨져야 합니다. 오브젝트 파스칼에서는 메소드 호출 시에 Self 라는 이름의 매개변수가 묵시적으로 넘겨지게 되는데 실제 그 메소드가 적용될 대상 인스턴스를 가리키는 것입니다.

개체 작성에 앞서 중요한 개체 관련 루틴 3가지에 대해 짚고 넘어가겠습니다. 델파이에서는 개체를 생성하면 꼭 다시 해제해주어야 합니다. 이를 행하는 루틴은 Create, Destroy, Free입니다.

## ■ Create

생성자는 예약어 Constructor로 시작하는 특별한 프로시저로 개체를 생성하고 초기화하는 동작을 수행하는 루틴입니다. 보통 생성자는 Create라는 이름으로 만들며 다음과 같은 형태를 갖습니다.

생성자는 오브젝트가 생성되기 전에 호출되기 때문에 클래스 이름으로 지정하여 호출하도록 되어 있습니다. 새로운 클래스를 작성할 때 필요에 따라 생성자를 작성할 수 있습니다.

```
Constructor TMyobj.Create;
Constructor TComponent.Create(AOwner:TComponent);
```

생성자는 형 이름으로 호출되는데

- ① 오브젝트를 위한 메모리 공간을 확보하고
- ② 선언된 필드들을 초기값으로 만들어 줍니다. 그 후에 프로그래머가 기술한 생성자 코드를 수행하고 끝나면
- ③ 할당된 오브젝트에 대한 주소 값을 돌려주게 됩니다.



### ■ Destroy

파괴자는 예약어 Destructor로 시작하는 루틴으로 생성자와 반대로 작업을 정리하고 메모리를 해제하는 기능을 수행합니다.

```
Destructor TMyobj.Destroy;
```

### ■ Free

할당된 개체의 인스턴스를 체크하여 내부적으로 Destroy를 부르는 루틴입니다.

```
Myobj.Free;
```

Myobj := TMyobj.Create; 개체를 생성합니다.  
그리고 개체를 해제할 때에는 Myobj.Free를 호출합니다.

## 5.8.2 O.O.P 프로그램 특징

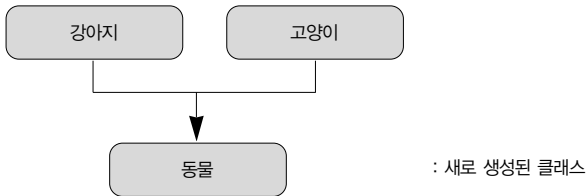
### ■ 상속(Inheritance)

상속이란, 기존의 클래스를 기반으로 새로운 클래스를 정의하는 것을 말합니다. 여기에서 기존의 클래스는 부모 클래스, 새로운 클래스는 자식 혹은 파생 클래스라고 표현합니다. 즉, 부모 클래스가 가지고 있는 상태와 행위를 그대로 가져오는 것을 말합니다. 이 때, 부모의 모든 것을 무조건 그대로 가져와야 합니다. 가져온 후에 다른 특성을 추가하는 등 수정을 하는 것은 가능하지만, 처음부터 받지 않는 것은 불가능합니다. ‘음악’을 예로 들어보겠습니다.



‘음악’이라는 클래스가 있을 때, 음악의 속성을 포함하고 있으며 여기에 자신만의 특성을 가미한 R&B라는 장르가 음악에서 파생되어 나올 수 있습니다. 이 때 ‘음악’은 부모 클래스가 되고, 음악의 한 장르인 ‘R&B’는 자식 클래스가 됩니다.

반대의 경우도 있을 수 있습니다. 이번에는 ‘동물’ 을 예로 들어보겠습니다.



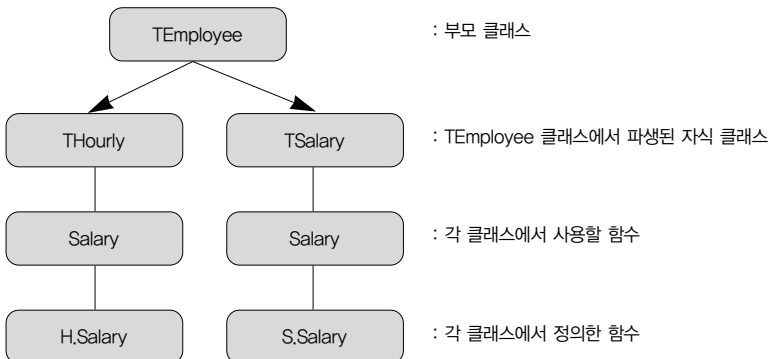
‘강아지’ 와 ‘고양이’ 라는 두 클래스의 공통점을 뽑아서 ‘동물’ 이라는 새로운 클래스를 생성할 수도 있습니다.

OOP에서 ‘상속’ 개념을 사용했을 때의 대표적인 장점은 코드의 중복을 막을 수 있다는 점입니다. 같은 내용의 코드를 계속해서 붙여넣기 할 필요가 없이, 부모 클래스를 상속만 받아와서 사용하면 되기 때문입니다. 또한 복잡한 데이터의 개념을 쉽게 이해할 수 있습니다.

#### ■ 다형성 (Polymorphism)

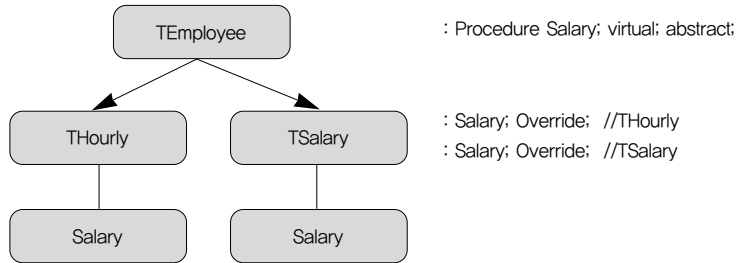
프로그래밍 언어에서 서로 다른 기능들을 하나의 문장으로 수행할 수 있는 능력을 말합니다. 즉, 하나의 루틴 이름으로 각 개체에 맞는 루틴을 수행할 수 있습니다.

예를 들어 TEmployee, THourly, TSalary 3개의 클래스가 있을 때 각각 THourly와 TSalary 클래스에서 Salary라는 함수를 사용한다고 가정해 보겠습니다.



위의 방법은 Salary 메소드 호출 시 정적바인딩을 사용하는데, 컴파일 시에 미리 결정된 클래스 변수의 형에 따라 메소드가 호출됩니다. 이 방법은 자료와 그에 맞는 메소드가 함께 연동되도록 하는 객체 지향의 개념에는 위배되지만 이런 문제를 고려할 필요가 없는 메소드들에서는 사용됩니다.

각 클래스는 Salary라는 동일한 함수를 사용하는데, 각자의 클래스에서 정의하게 되어 똑같은 기능을 하는 변수를 두 번 선언하였습니다. 같은 예를 '다형성'을 사용하는 방식으로 바꾸어 보면,



'다형성'을 이용하여 표현한 그림입니다. 그림상에서 달라진 부분은 'H.Salary'와 'S.Salary'가 사라졌다는 것뿐입니다. 그러나 코딩을 살펴보면 '다형성'을 사용한 경우, 단 3줄로 요약이 되는 것을 확인할 수 있습니다. 지금 예를 든 것은 매우 단순한 예입니다. 실제로 더욱 복잡한 상황을 코딩해야 한다면? 그 차이는 더욱 확연할 것입니다.

```
Procedure Salary; virtual; abstract;
```

이 문장은 TEmployee 하위의 클래스들이 전부 Salary를 사용할 것을 의미합니다.

#### ▶ virtual

virtual 메소드는 하위 클래스에서 같은 이름으로 된 메소드를 만들어 override 시킬 수 있는데 이렇게 override된 메소드를 호출하면 실행 시에 개체형을 조사하여 그에 맞는 메소드가 자동으로 선택되어 호출 됩니다.

virtual은 선언한 Salary를 앞으로 재사용하겠다는 것을 의미합니다.

#### ▶ abstract

abstract 선언은 쉽게 말해 추상화를 의미합니다. Body가 없는 경우에 사용합니다. 부모 클래스에서 선언은 하지만 구현은 상속받은 자식 클래스가 합니다.

#### ▶ override

부모 클래스에게서 상속받은 Salary를 자식 클래스가 재정의해서 사용하겠다는 것을 의미합니다.

virtual로 선언하지 않은 내용은 override할 수 없습니다.

Procedure Salary; override;

#### ▶ Final

오버라이드 한 버추얼 메소드를 final로 표시하여 상속되는 클래스에서 해당 메소드를 더 이상 오버

라이드 하지 못하도록 막을 수 있습니다.

```
Procedure Salary; override; final;
```

## ■ 캡슐화 (Encapsulation)

클래스에서 정의한 개체를 외부에 공개하지 않도록 설정하는 기능입니다.

### ▶ Public

특별한 제한이 없는 필드나 메소드를 나타냅니다. 개체를 정의한 클래스는 물론이고, 다른 곳에서도 참조해서 사용 가능합니다.

### ▶ Private

클래스가 정의된 유닛의 외부에서 접근하거나 알아보지 못하도록 프로시저, 함수, 필드를 정의하기 위해 사용합니다. 식별자의 범위는 선언한 클래스까지입니다. 예외적으로, 같은 유닛에서 정의된 개체는 호출해서 사용 가능합니다.

### ▶ Strict private(2007에서 추가)

Private는 같은 유닛 상에서는 사용이 가능하지만, Strict private는 오직 자신의 하위 개체에게만 허용합니다.

### ▶ Protected

Public과 Private 중간 형태로 하위 개체들에게만 열려 있습니다. 주로 계승된 클래스에서 내부적으로 사용하는 멤버들을 선언합니다.

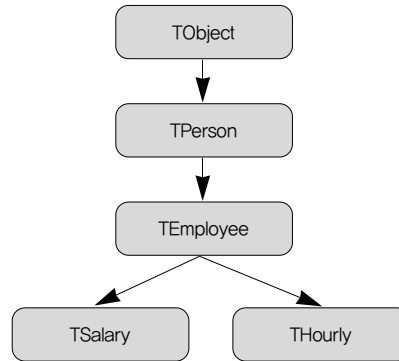
### ▶ Strict protected(2007에서 추가)

개체가 정의된 클래스나 그 클래스에 상속된 하위 클래스에게만 사용을 허가합니다.

### ▶ Published

가시성에서는 Public과 같습니다. 자기 자신은 물론, 다른 곳에서도 호출해서 사용 가능합니다. 단, Object Inspector 부분에 표시되는 부분을 선언하는 영역입니다.

이제 OOP의 특징을 이용하여 아래의 클래스들을 사용하는 프로그램을 작성해 보겠습니다.



**01** utest4.pas의 유닛의 type부분에 위의 개체들을 순서대로 TPerson부터 정의하도록 하겠습니다.

```
unit Utest4;

interface

type
TPerson = class(TObject) // TObject 상속을 받는 TPerson 클래스 생성
private
 ttt : String; // 오직 이 클래스에서만 사용합니다.
public
 // 다음의 내용은 다른 클래스에서도 사용합니다.
 Name : String;
 Age : byte;
 Address : String;
 Constructor Create; virtual; // create의 내용을 상속하여 사용
 function Getname : String;
end;
```

- TObject 상속을 받는 TPerson 클래스를 생성합니다.
- 그 안에 TPerson에서 사용할 내용들을 선언합니다.
- 생성자 루틴 Create를 선언합니다. TPerson의 하위 클래스들에서도 재사용하기 위해 virtual로 선언합니다.

**02** Ctrl + Shift + C를 이용하여 코딩 할 수 있는 공간을 엽니다.

**03** Constructor 루틴에서 초기처리 하도록 기본값을 줍니다.

```
{ TPerson }
{ 사원에 대한 정보 입력 }
Constructor TPerson.Create;
begin
 Name := 'Kim';
 Age := 20;
 Address := 'Anywhere';
end;
```

**04** TPerson의 Getname 루틴을 다음과 같이 코딩합니다.

```
{ 사원 이름을 호출하는 함수 }
function TPerson.Getname: string;
begin
 Result := Name;
end;
```

**05** TPerson을 상속받는 TEmp 클래스 생성합니다.

```
TEmp = class(TPerson)
public
 Office : String;
 Empno : Integer;
 Baserate : real;
 Constructor Create; override; // create를 상속받아와 재정의하여 사용
 function Salary : real; virtual; abstract; // salary 함수를 재사용
end;
```

- TEmp는 TPerson 클래스를 상속받는 클래스 입니다.
- override를 이용하여 TPerson에서 정의했던 create를 상속 받아와서 사용합니다.
- Salary라는 함수를 새로 정의합니다. 이 함수 역시 하위 클래스에서 재사용 할 수 있도록 합니다.

## 06 Ctrl + Shift + C 를 이용하여 빈 공간에 코드를 입력합니다.

```
constructor TEmp.Create; // TPerson에서 정의한 create를 상속받음
begin
 inherited;
 (* 상속받은 create를 재정의합니다 *)
 Office := '데브기어';
 Empno := 1;
 Baserate := 10;
end;
```

좀 더 자세히!

### Inherited 예약어

Inhrited Method 이름을 사용하면 선조 클래스의 해당 이름으로된 메소드를 호출하게 됩니다. 바로 직계 선조에 그런 메소드가 없다면 연속해서 선조의 선조를 차례대로 찾아 호출합니다. 위에서처럼 Inhrited만 사용한다면 그 메소드 이름으로된 선조 클래스의 메소드를 호출하게됩니다. 즉, 위의 코드와 Inherited Create와는 동일한 코딩입니다.

## 07 TEmp를 상속받는 THourly를 선언합니다.

```
THourly = class(TEmp)
 Hrs : Integer;
 Constructor Create ; override; // create를 재정의하여 사용
 function Salary : real; override; // salary를 재정의하여 사용
end;
```

- public으로 선언하여, 다른 곳에서도 호출하여 사용할 수 있도록 정의합니다.
- Create와 Salary를 상속 받아와서 재정의하여 사용합니다.

## 08 상속받은 Create와 Salary에 대해 재정의합니다.

```
{ THourly }

Constructor THourly.Create;
begin
 Inherited;
```

```

 Hrs := 10;
end;
function THourly.Salary: real;
begin
 Result := Baserate * Hrs;
end;

```

**09** TSalary 클래스를 정의합니다.

```

TSalary = class(TEmp)
 Salesamt : real;
 Commissionrate : real;
 Constructor Create; override;
 function Salary : real; override;
end;

```

**10** 상속받은 Create와 Salary에 대해 재정의합니다.

```

{ TSalary }
constructor TSalary.Create;
begin
 Inherited;
 Commissionrate := 1;
 Salesamt := 5;
end;
function TSalary.salary: real;
begin
 result := (Salesamt * Commissionrate) + (Baserate + 40.0);
end;

```

지금까지는 유닛에 직접 코드를 구현해서 클래스를 작성하고 변수나 루틴들을 직접 코드로 추가했습니다. 이번에는 델파이 환경중에 클래스 익스플로러를 사용하여 클래스에 메소드들을 추가해 보겠습니다.



텔파이의 또 다른 특징은 새로운 클래스 익스플로러(Class Explorer)입니다. 클래스 익스플로러는 Delphi의 클래스 모델링 기능을 바탕으로 만들어졌으며 전체 프로젝트 내부의 클래스 구조를 볼 수 있게 해 줍니다. 표시방식은 툴 바의 첫 번째 버튼의 선택에 따라 달라집니다. Base to derived, Derived to base 또는 Container 세가지 중의 하나입니다.

**따 라 하 기**

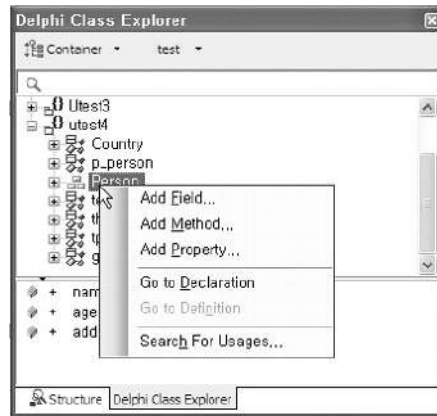
The screenshot shows the Delphi Class Explorer window. The 'Container' dropdown is set to 'test'. The search bar is empty. The list of classes under 'test' includes 'test', 'utest1', 'utest2', 'utest3', and 'utest4'. The 'test' class is currently selected.

The screenshot shows the Delphi IDE's Class Explorer. The project is named 'test'. The class hierarchy is as follows:

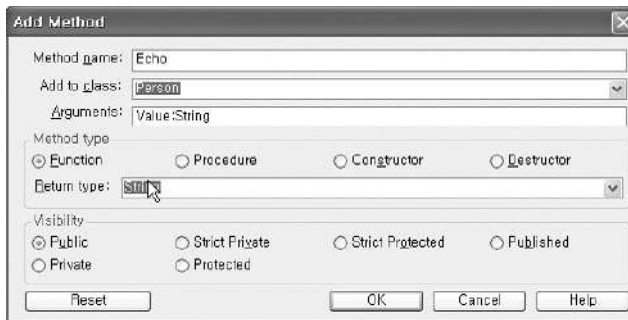
- Utest3
  - utest1
    - Country
      - p.person
        - Person** (highlighted)
        - temp
        - th
        - tperson
        - global's utest4

The bottom of the window shows the 'Structure' tab and the text 'Delphi Class Explorer'.

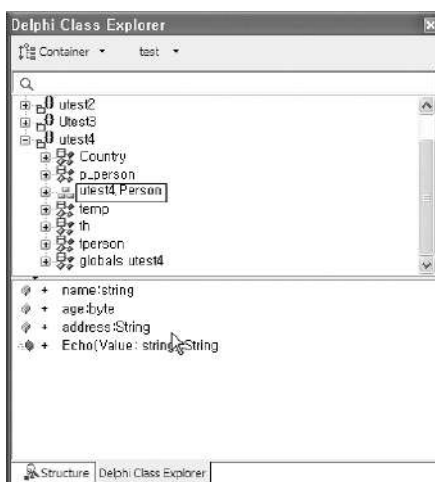
- 03 Tperson 클래스에서 오른쪽 마우스를 클릭하여 “Add Method”를 선택하여 메소드를 추가합니다.



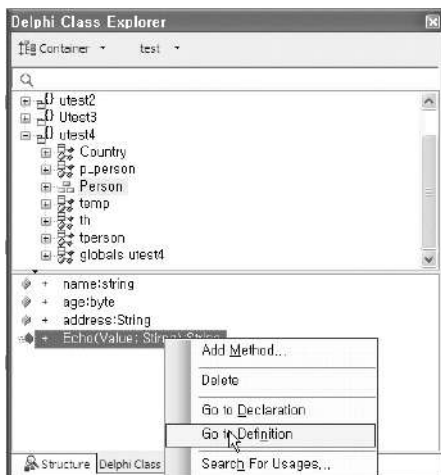
- 04 메소드 이름, 리턴 타입, 매개변수 타입들을 아래와 같이 지정합니다.



- 05 다음과 같이 메소드가 추가된 것을 확인하실 수 있습니다.



06 해당 메소드의 팝업 메뉴에서 “Go to Definition”을 선택하면 소스의 구현부분으로 이동합니다.



07 다음과 같이 코드 부분을 구현합니다.

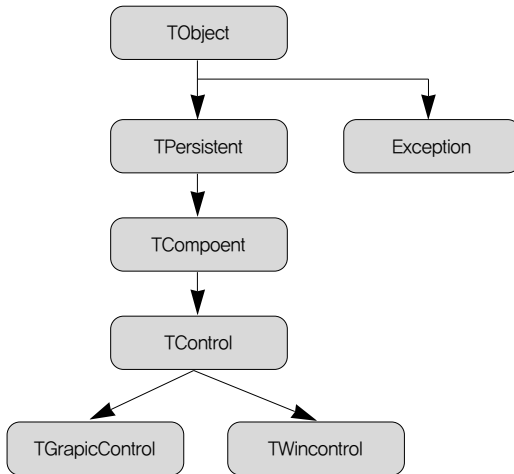
```
function Person.Echo(Value: string): String;
begin
 Result := Value;
end;
```

## 6. 컴포넌트 라이브러리

앞에서 작성한 클래스들처럼 델파이는 VCL(Visual Component Libray)라는 이름으로 윈도우 프로그래밍에 필요한 세부사항을 미리 만들어 준비해 두고 개발자들이 편리하게 사용할 수 있도록 제공합니다.

컴포넌트 라이브러리 내의 컴포넌트들은 모두 상속에 의한 계층적인 구조를 가지고 있습니다. 이것을 VCL 계층구조(VCL Hierarchy)라고 부릅니다.

VCL에서 중요한 위치를 차지하는 기본 클래스들 몇가지에 대해서 알아보겠습니다.



### ■ TObject

델파이의 모든 VCL 클래스들은 이 TObject 클래스로부터 상속됩니다. 모든 클래스의 공통적인 특성과 기능들을 가지고 있습니다.

### ■ TPersistent

SaveToStream과 LoadFromStream이라는 매우 중요한 메소드 2개를 가지고 있습니다.

이 클래스에서 상속받은 모든 클래스들은 자신의 상태 정보를 필요할 때마다 저장하고 다시 불러올 수 있게 됩니다.

### ■ TComponent

이 클래스는 모든 델파이 컴포넌트의 조상클래스입니다. 다음과 같은 기본적인 컴포넌트의 기능들을 가능하게 해줍니다.

- 툴 팔레트에 설치되는 기능
- 디자인 타임에 마우스를 이용하여 폼에 추가 되는 기능
- 오브젝트 인스펙터를 통해 속성이 설정되는 기능

이 클래스에서 모든 컴포넌트들이 사용하는 Name 속성이 처음 도입됩니다.

#### ■ TControl

모든 시각적인 컴포넌트들의 선조 클래스입니다.

#### ■ TGraphicControl

시각적인 컴포넌트이기는 하지만 입력 포커스를 받지 못하는 컴포넌트의 선조 클래스입니다. 예를들면 TImage, TLabel, TBevel등이 그 자손 컴포넌트입니다.

#### ■ TWincontrol

입력 포커스를 받을 수 있는 선조 클래스입니다. TEdit, TComboBox, TButton등의 컴포넌트가 있습니다.

#### ■ Exception

델파이는 에러 상황들마다 클래스를 미리 정의해 놓고 구조적으로 에러 처리를 할 수 있게 해 줍니다. 이런 클래스를 예외 클래스라고 부르는데 모든 예외 클래스의 가장 선조가 되는 클래스입니다. 예외 부분은 별도의 장에서 소개하고 처리하도록 하겠습니다.

또한 이 컴포넌트 라이브러리 내의 컴포넌트들은 다음과 같은 구체화된 상태와 행위를 갖습니다.

##### ① 속성(Property)

컴포넌트가 표시되고 제어되는 방법에대한 Attribute 변수로서 필드가 안전하게 참조 될수 있도록 Read/Write문을 통해 속성값을 사용합니다. Read/Write문에 대해서는 컴포넌트 작성부분에서 자세히 설명하겠습니다. 속성은 값을 읽어들이는 일종의 통로 역할을 합니다.

##### ② 메소드(Method)

컴포넌트 안에 정의되어 있는 프로시저나 함수입니다.

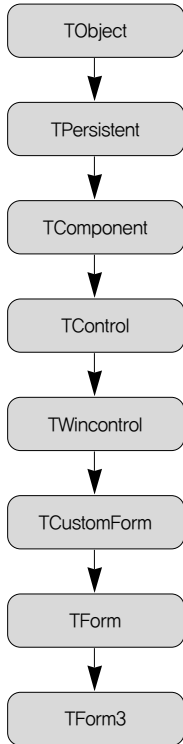
##### ③ 이벤트(Event)

컴포넌트가 인식할 수 있는 사용자 행위나 시스템 동작들입니다.



### Tip

여러분들이 앞에서 프로젝트에 추가해 놓으신 폼의 계층은 어떻게 될까요 ?  
 좌측의 그림은 작성한 폼의 계층 구조를 표시한 것이며 우측은 TForm3의 유닛 소스(utest3.pas) 선언 부분  
 입니다.(자동으로 코딩되어 생성됩니다)



```

unit Utest3;

interface

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

type
 TForm3 = class(TForm) ————— 내가 작성하는 폼 클래스 선언
 Private
 { Private declarations }
 public
 { Public declarations }
 end; ————— 이 부분에 컴포넌트나 프로시저나 함수를 추가합니다.

var
 Form3: TForm3; ————— TForm3 의 변수 선언

implementation

{ $R *.dfm} ————— 컴파일 명령어로 같은 이름의 폼 파일과 같이 합쳐서 컴파일 합니다.

end.

```

이벤트의 종류에는 크게 보면 두 가지 종류가 있습니다. 하나는 사용자가 발생시키는 이벤트인 사용자 이벤트이고 다른 하나는 운영체제가 발생시키는 이벤트인 시스템 이벤트입니다.

델파이는 이런 이벤트들을 폼이나 컴포넌트안에 정의해 두었습니다. 이렇게 정의해 놓은 이벤트들을 오브젝트 인스펙터의 이벤트 탭에서 볼 수 있습니다.

마우스와 키 보드의 동작과 여기에 상응하는 델파이의 이벤트를 정리하면 다음과 같습니다.

| 마우스 동작        | 설명                                                | 델파이 이벤트                               |
|---------------|---------------------------------------------------|---------------------------------------|
| Down          | 마우스 버튼을 누른 상태                                     | OnMouseDown                           |
| Up            | 마우스 버튼이 눌려졌다가 다시 올라온 상태                           | OnMouseUp                             |
| Click         | 마우스 버튼을 눌렀다가 바로 떼는 동작                             | OnClick                               |
| Double Click  | 마우스 버튼을 두 번 연속적으로 클릭하는 동작                         | OnDblClick                            |
| Move          | 마우스를 끌어 이동하는 동작                                   | OnMouseMove                           |
| Drag and Drop | 마우스 버튼을 누른 채 화면상의 어떤 요소를 끌어 이동 시킨 후 마우스 버튼을 놓는 동작 | OnDragDrop<br>OnDragOver<br>OnEndDrag |

| 키보드 동작   | 설명                                                                 | 델파이 이벤트                 |
|----------|--------------------------------------------------------------------|-------------------------|
| Key Down | 키보드에서 하나의 키를 누른 상태<br>어떤 한 컴포넌트가 입력 포커스를 가지고 있을 때<br>키보드의 키를 누른 상태 | OnKeyPress<br>OnKeyDown |
| Key Up   | 눌러 있던 키가 올라올 때 발생                                                  | OnKeyUp                 |

## 7. 개체 사용

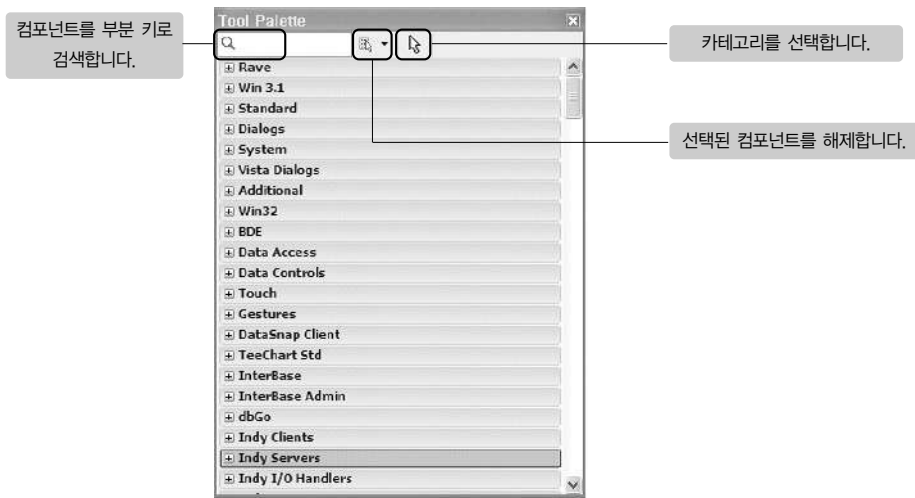
개체 사용의 방법은 다음과 같은 과정으로 이루어 집니다.

- ❶ 사용할 클래스가 정의된 유닛을 `uses`절에 추가합니다.
- ❷ 클래스형의 변수를 선언합니다.
- ❸ 생성자 루틴(Create)을 호출하여 메모리에 할당하고 오브젝트의 주소 값을 변수에 할당합니다.
- ❹ 인스턴스 변수를 통해 해당 개체의 필드와 메소드를 참조 또는 호출합니다.
- ❺ 개체를 다 사용한 후에는 Free 메소드를 호출하여 파과자(Destroy)루틴을 실행하여 메모리에서 해제합니다.

자, 그럼 처음으로 컴포넌트를 사용하여 보겠습니다. 인스톨 되어있는 컴포넌트를 선택하기 위해 툴 팔레트를 사용하여 보겠습니다.

### ■ 툴 팔레트

델파이에서 사용이 가능한 모든 컴포넌트들이 기능별로 분리되어있어 사용이 용이합니다. 사용자 정의 컴포넌트 추가 및 삭제 또한 가능합니다. 검색 및 필터링 기능 제공으로 더욱 손쉽게 컴포넌트를 사용할 수 있습니다. 툴 팔레트에서 검색한 컴포넌트를 클릭하고 폼 위에서 클릭하면 바로 입력됩니다.



## ■ 폼 위에 컴포넌트를 추가하기 !

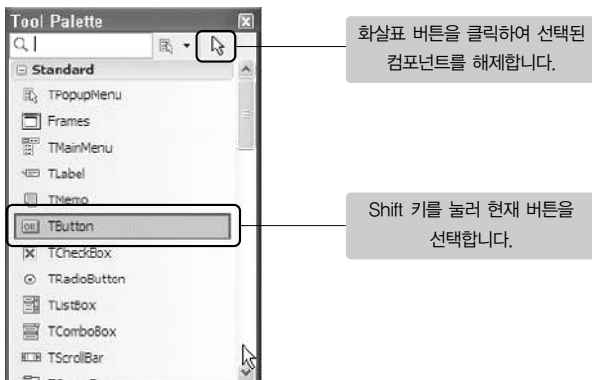
컴포넌트를 폼에 추가하는 방법에는 그 용도에 따라서 몇 가지가 있습니다. 여기에서 설명하는 방법을 알면 실제로 프로그램 작업을 할 때 불필요한 작업을 줄임으로써 프로그램 개발 시간을 단축할 수 있습니다.

### ▶ 폼의 특정 위치에 컴포넌트를 추가하기

- ① 툴 팔레트의 Standard Category의 TButton 컴포넌트를 마우스로 선택합니다.
- ② 폼에 컴포넌트가 놓일 위치를 마우스로 클릭합니다. 이 때 컴포넌트의 크기도 같이 조절하기 위하여 마우스를 클릭한 상태에서 마우스를 움직여 크기를 조절 할 수 있습니다.

### ▶ 동일한 컴포넌트를 여러 개 추가하기

만약 버튼 컴포넌트 10개를 폼에 추가해야 하는 경우 위의 방법을 사용하면 위의 작업을 10번 반복해야 합니다. 따라서 동일한 컴포넌트를 여러 개 폼에 추가하기 위해서는 다음 방법을 사용합니다.

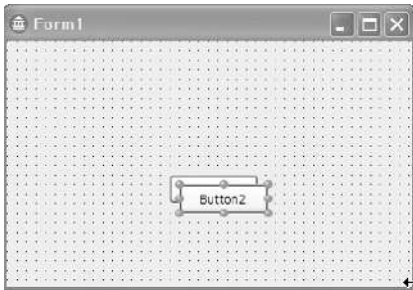




- ❶ Shift 키를 누른 상태에서 버튼 컴포넌트를 선택합니다.
- ❷ 폼의 특정 위치를 마우스로 클릭하고 컴포넌트의 크기를 조절합니다.
- ❸ 원하는 수 만큼 반복해서 폼의 특정 위치에 컴포넌트를 추가합니다.
- ❹ 컴포넌트 추가가 끝나면 툴 팔레트에서 화살표를 클릭하여 해재합니다.

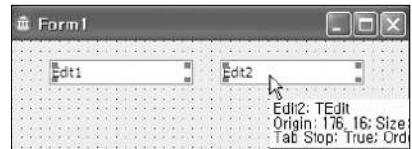
#### ▶ 컴포넌트를 폼에 빠르게 추가하기

컴포넌트를 폼에 빠르게 추가하기 위하여 추가하고자 하는 컴포넌트를 마우스로 더블 클릭합니다. 그러면 폼의 중앙에 위치하게 됩니다.한 번 더 더블-클릭하면 아래 그림과 같이 생성됩니다.

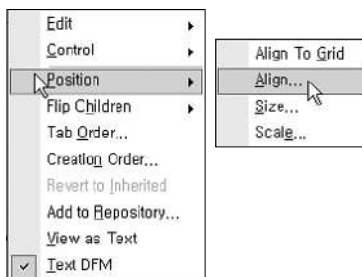


#### ■ 컴포넌트를 정렬해보자!

- ❶ 툴 팔레트에서 TEdit 컴포넌트 2개를 추가합니다.
- ❷ Shift 키를 누른 채로 컴포넌트를 차례로 클릭합니다. 컴포넌트 주변에 회색 점선이 나타납니다.



- ❸ 컴포넌트를 복수개 선택한 후 오른쪽 마우스 버튼을 클릭하면 팝업메뉴가 나타납니다. 여기서 Align 메뉴를 선택하면 정렬창이 표시됩니다.





```

TForm1 = class(TForm)
 Button1: TButton; ————— TButton의 변수를 선언합니다.
private
 { Private declarations }
public
 { Public declarations }
end;
var
 Form1: TForm1;

implementation

{ $R *.dfm}

end.

```

**03** 이 버튼의 생성은 언제 누가 할까요? 이 버튼을 사용하는 화면(TForm)이 생성될 때 이 폼 위에 있는 컴포넌트도 같이 생성해 줍니다. 즉, 이 버튼의 AOwner 컴포넌트는 Form1이 되며 TForm1의 AOwner 컴포넌트는 TApplication 입니다.

**04** 버튼의 속성, 이벤트를 오브젝트 인스펙터를 통해서 설계 시점에 설정해 보겠습니다.



#### Tip

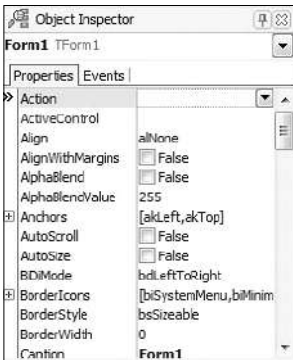
##### 컴포넌트 명명하기

폼에 컴포넌트를 추가하면 컴포넌트 이름에 번호가 자동으로 붙어서 이름이 생성됩니다. 이 이름은 Name이라는 속성에 자동으로 할당됩니다. 예를 들면 버튼 컴포넌트를 폼에 추가하면 Button1, Button2 이런 식으로 이름이 생깁니다. 이런 식으로 계속 컴포넌트 이름을 사용하면 어떤 컴포넌트가 어떤 기능을 수행하는지 혼동이 올 수 있습니다.

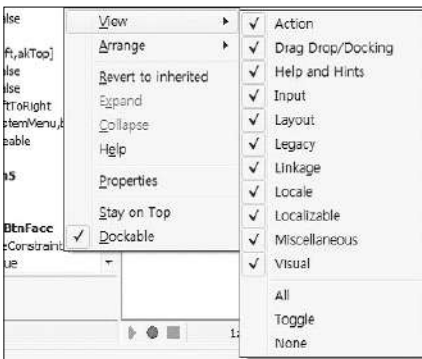
따라서 컴포넌트를 새로 추가하면 컴포넌트 이름을 그 때 그 때 바꿔주는 습관이 필요합니다. 이미 여러분들은 이런 규칙들을 가지고 계실 것입니다. 이것을 명명 규칙이라고 합니다.

## 오브젝트 인스펙터 사용방법

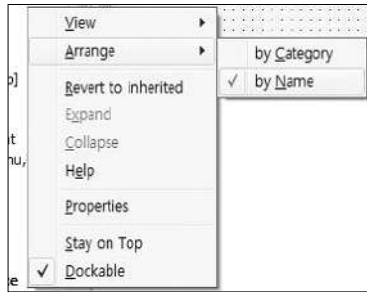
❶ 오브젝트 인스펙터를 이용하여 컴포넌트의 속성을 변경할 수 있습니다. 속성값이 수정된 부분은 굵게 (bold) 처리 됩니다.



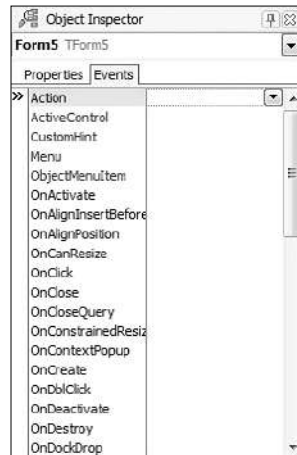
❷ 오브젝트 인스펙터 화면에서 마우스 오른쪽 버튼 → View를 선택하여 오브젝트 인스펙터 화면에 보이는 속성들을 선택할 수 있습니다. 기본적으로 모든 값이 선택되어 있습니다



❸ 동일한 방법으로 Arrange를 선택하여 정렬 방법을 선택할 수 있습니다.



❹ 오브젝트 인스펙터의 상단에 보면, 'Properties'와 'Event' 탭이 있는 것을 볼 수 있습니다. 'Properties' 탭은 앞서 설명한 속성 지정 기능이 있고, 두 번째로 'Events' 탭은 컴포넌트 이벤트 설정 기능이 있습니다. 아래의 그림은 Events 탭을 선택했을 때의 화면입니다.



● 속성의 종류

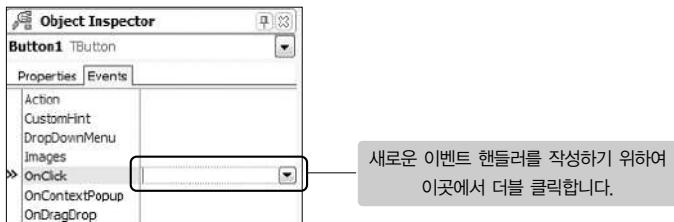
| 종류                 | 특징 및 속성값 변경방법                                                                                                                                                                                                                                                      |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Simple Type        | Name이나 Caption 속성처럼 직접 속성값을 입력합니다.                                                                                                                                                                                                                                 |
| Editor Type        | Font 속성처럼 '...' 표시되며 클릭하면 별도의 에디터 화면이 표시됩니다. <div data-bbox="541 518 739 850" data-label="Image"> </div> <div data-bbox="812 552 1033 620" data-label="Text"> <p>클릭하면 아래와 같은 에디터 창이 표시됩니다.</p> </div> <div data-bbox="812 628 1158 850" data-label="Image"> </div> |
| Sub Property Type  | 속성앞에 '+' 표시가 있으면 하위 속성이 있다는 표시입니다. 클릭하면 하위 속성이 표시됩니다. <div data-bbox="541 966 739 1288" data-label="Image"> </div> <div data-bbox="812 1139 1033 1206" data-label="Text"> <p>Font 속성의 하위 속성들이 표시됩니다.</p> </div>                                                    |
| Enum Property Type | 속성값을 콤보박스 안에 열거된 값 중에서 선택합니다. <div data-bbox="541 1373 739 1701" data-label="Image"> </div>                                                                                                                                                                        |

**05** Button1의 속성(Property) 중 Caption, Name, ParentFont 세 부분의 값을 설정하여 줍니다.

| 컴포넌트    | Property   | 값        |
|---------|------------|----------|
| TButton | Caption    | close    |
|         | Name       | MyButton |
|         | ParentFont | False    |

- **Caption** : 화면에 보여지는 문자입니다.
- **Name** : 컴포넌트가 실제로 갖는 고유이름입니다. 현재 작업을 예로 들면, 코딩을 할 때 Button1에 대해 close 라고 선언하면 오류가 발생합니다. 이유는 Close라는 기존의 메소드와 혼동이 생기기 때문입니다. 여기서는 MyButton으로 선언하겠습니다.
- **ParentFont** : Parent라는 속성은 TControl컴포넌트에 있는 속성으로 어느 컴포넌트 위에 표시할 것인지를 결정하는 속성입니다. 즉 ParentFont는 Parent 컴포넌트의 Font를 따를 것인지를 결정합니다. 이 버튼의 Parent는 폼이므로 ParentFont가 True로 설정할 경우 화면의 폰트가 변경되면 버튼의 폰트도 변경됩니다.

**06** MyButton이 실행할 이벤트를 설정해 줍니다. 오브젝트 인스펙터의 탭페이지를 선택하여 해당 이벤트 "OnClick"를 선택합니다.



**07** OnClick 우측의 빈 공간에 마우스를 놓고 더블-클릭합니다.

**08** 다음과 같이 해당 유닛에 자동으로 코딩이 되면서 이벤트 핸들러 소스 안으로 커서가 놓이게 됩니다.

```
type
 TForm1 = class(TForm)
 MyButton: TButton;
 procedure MyButtonClick(Sender: TObject);
 private
 { Private declarations }
 public
 { Public declarations }
```

```

end;

var
 Form1: TForm1;

implementation
{ $R *.dfm}
procedure TForm1.MyButtonClick(Sender: TObject);
begin
 ← 이곳으로 커서가 위치합니다.
end;

```

위에 선언된 MyButtonClick 루틴은 클래스의 디폴트인 Published에 선언되었습니다. 여기에서 매개변수 Sender의 의미는 이벤트가 발생한 경우 어떤 컴포넌트에서 이 이벤트가 발생했는지 알려줍니다. 즉 이 Sender 매개변수만 있는 이벤트를 TNotifyEvent라고 합니다.

## 09 Close 메소드를 호출하여 아래와 같이 코딩합니다.

```

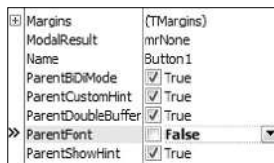
procedure TForm1.MyButtonClick(Sender: TObject);
begin
 { MyButton을 선택하면 화면이 닫힙니다. }
 Close;
end;

```

10 F9를 눌러 실행하면 버튼을 클릭했을 때 메인 폼이 닫히면서 프로그램이 종료됨을 확인합니다.

11 화면에 버튼을 더 추가합니다(Name 은 다시 Button1이 됩니다).

12 버튼의 속성을 아래와 같이 수정합니다.



| 컴포넌트    | Property   | 값       |
|---------|------------|---------|
| TButton | Caption    | 실시간에 변경 |
|         | ParentFont | False   |

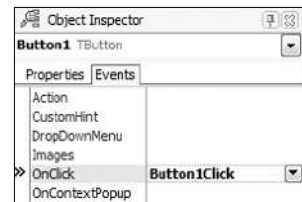


#### Tip

#### Tway-Tool(양방향 도구)

위의 예제는 동일한 속성 Caption을 설계 시점에 변경하거나 버튼을 클릭한 시점 즉, 런타임 시점에 변경했습니다. 이는 앞으로의 델파이 사용 방법을 단적으로 보여줍니다. 설계 시 폼에서 수정하거나 런 타임 시점의 코딩으로 컨트롤합니다.

## 13 MyButton과 마찬가지로 ‘실시간에 변경’ 버튼에 대해 OnClick 이벤트를 설정합니다.



```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Button1.Caption := '닫기';
 // 실시간에 변경 버튼(Button1)을 클릭하면 '닫기' 메시지를 이 버튼 위에 보여줍니다.
end;
```



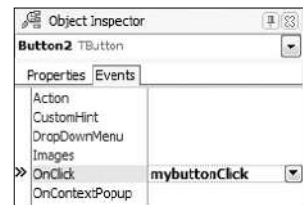
- 14 이번에는 핸들러를 공유하는 방법을 알아보겠습니다. 새로운 버튼 컴포넌트를 폼 위에 올려놓고 아래와 같이 속성값을 변경합니다.



| 컴포넌트    | Property | 값     |
|---------|----------|-------|
| TButton | Caption  | 핸들러공유 |



- 15 동일하게 OnClick 이벤트를 설정합니다. 그러나 앞서 한 것과 달리 MyButton과 핸들러를 공유하기 위해, 우측 아래화살표 버튼을 이용하여 MyButtonClick을 선택합니다.



#### Tip

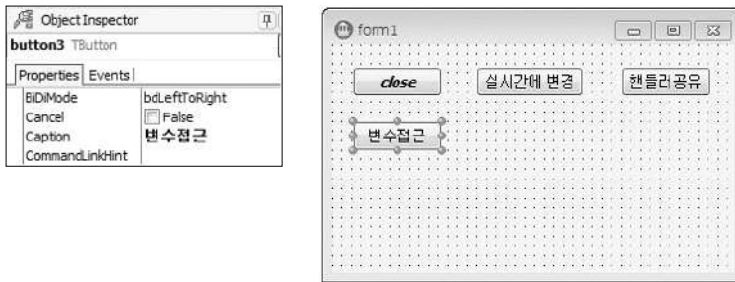
##### 이벤트 핸들러 공유

윈도우에서는 서로 다른 개체의 이벤트 핸들러를 공유하거나 Exchange 할 수 있습니다. 델파이에서도 이벤트 핸들러를 공유할 수 있습니다. 이 때 각 이벤트 핸들러가 호출되었을 때 Sender 매개 변수에는 각각 MyButton 과 Button2가 넘어올 것입니다.

- 16 실행하여 이 버튼을 클릭 했을때도 폼을 닫고 프로그램이 종료하는지 확인합니다.

- 17 이제 앞 부분에서 코딩한 부분들을 확인해 보도록 하겠습니다.

18 폼 위에 버튼을 추가하고 다음과 같이 설정합니다.



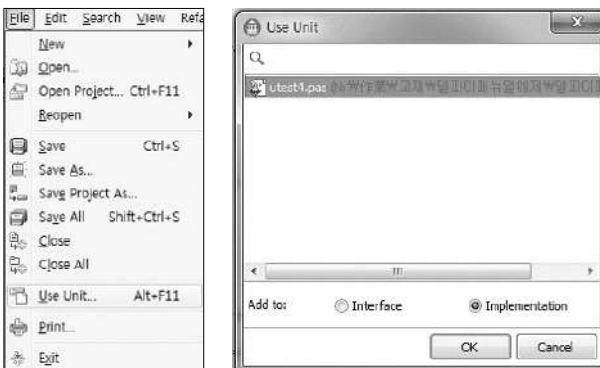
| 컴포넌트    | Property | 값    |
|---------|----------|------|
| TButton | Caption  | 변수접근 |

19 '변수접근' 버튼에 대해 OnClick 이벤트를 설정합니다.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 Button3.Caption := Countries[0];
 //utest4.pas에서 선언했던 Countries[0]의 값을 변수접근버튼(Button3)에 보여줍니다.
end
```

#### 좀 더 자세히!

이 소스를 컴파일하면 어떻게 될까요? 당연히 "Undeclared Identifier"라는 오류가 발생할 것입니다. 연관되어 있지 않은 utest4.pas의 선언 내용을 사용하기 때문입니다. utest4.pas의 내용이 현재 작업중인 utest1.pas와 연결되어 있지 않으므로, utest4.pas의 내용을 utest1.pas에 적용해야 합니다. 이를 위해, File > Use Unit 혹은 단축키 Alt + F11을 이용합니다.



원하는 유닛 화면을 uses 절에 추가할 수 있도록 도와줄 화면이 뜹니다. 추가하기를 원하는 유닛 화면을 선택하고, Interface 부분에 추가할지 Implementation 부분에 추가할지를 정해 추가합니다. 이 예제에서는 utest4.pas를 utest1.pas의 Implementation 부분의 uses 절에 추가합니다.

## 20 새로운 TButton 컴포넌트를 추가한 후, 속성을 '함수호출' 이라고 변경하고 OnClick 이벤트를 설정합니다.

```
procedure TForm1.Button4Click(Sender: TObject);
begin
{ utest4.pas에서 작성했던 Divide에 값을 넣어 결과값을 보여줍니다. }
 Button4.Caption := IntToStr(Divide(10,2));
end;
```



### Tip

런타임에 하나의 자료 형을 다른 자료 형으로 변환하는 형 변환 함수들을 소개합니다.

| 루틴               | 설명                                   |
|------------------|--------------------------------------|
| CurrToStr        | Currency 값을 스트링형으로 변환합니다.            |
| FloatToDecimal   | 실수형 값을 Decimal 값으로 변환합니다.            |
| FloatToStr       | 실수형 값을 스트링형 값으로 변환합니다.               |
| IntToStr         | Integer값을 스트링형 값으로 변환합니다.            |
| DatetimeToStr    | Datetime의 값을 스트링형 값으로 변환합니다.         |
| StringToWideChar | Ansi 스트링 값을 유니코드 스트링형으로 변환합니다.       |
| StrToCurr        | 실수 값을 포함하는 스트링형을 Currency 값으로 변환합니다. |
| StrToFloat       | 스트링 값을 실수형으로 변환합니다.                  |
| StrToInt         | 스트링 값을 정수형으로 변환합니다.                  |
| StrToDatetime    | 스트링 값을 Datetime형으로 변환합니다.            |
| StrPas           | PChar형을 스트링값으로 변환합니다.                |
| StrPCopy         | 스트링값을 PChar형으로 변환합니다.                |

## 21 새로운 TButton 컴포넌트를 추가한 후, 속성을 '프로시저호출' 이라고 변경하고 OnClick 이벤트를 설정합니다.

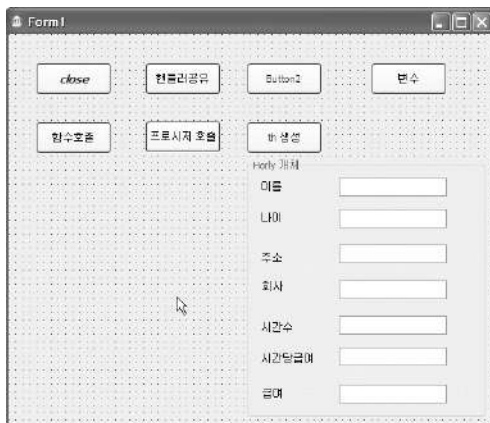
```
procedure TForm1.Button5Click(Sender: TObject);
begin
{ utest4.pas에서 선언한 프로시저 Test가 호출됩니다 }
 Test;
end;
```

**22** 지금까지는 컴포넌트를 마우스 클릭함으로서 자동으로 생성하고 자동으로 해제하는 방법으로 사용해 보았습니다. 하지만 모든 델파이의 컴포넌트 및 오브젝트를 자동으로만 사용할 수 없습니다. 그 이유는

- ① 마우스로 클릭할 수 없는 개체나 컴포넌트를 사용하는 경우
- ② 메모리 사용면에서 효율적으로 사용하기 위해

앞에서 작성한 클래스 TPerson, TEmployee, THourly, TSalary중 THourly개체를 사용해 보겠습니다.

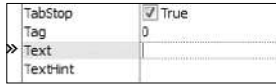
**23** 툴 팔레트에서 GroupBox 컴포넌트를 내려놓고 다음과 같이 설계합니다.



**24** 툴 팔레트에서 TLabel 컴포넌트를 GroupBox 위에 올립니다.

| GroupBox 이름 | 속성      | Label  | 속성값    |
|-------------|---------|--------|--------|
| GroupBox1   | Caption | Label1 | 이름     |
|             |         | Label2 | 나이     |
|             |         | Label3 | 주소     |
|             |         | Label4 | 회사     |
|             |         | Label5 | 시간수    |
|             |         | Label6 | 시간당 급여 |
|             |         | Label7 | 급여     |

## 25 톨 팔레트에서 TEdit 컴포넌트를 선택하여 폼 위에 올립니다.



처음 TEdit를 폼 위에 올렸을 때, Edit 화면에 디폴트 값 'Edit1' 으로 입력이 되어있습니다. 이 값들을 지우기 위해 오브젝트 인스펙터의 Text 속성에 입력되어 있는 값을 지웁니다.

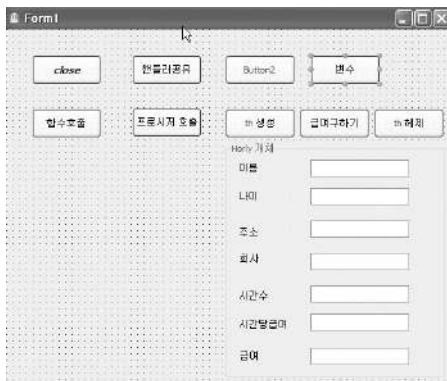


### Tip

동시에 여러 개의 컴포넌트의 속성값을 변경하거나 이벤트 핸들러를 공유하거나 위치를 이동하고 싶을 때는 컴포넌트를 선택하고 다른 컴포넌트를 Shift와 함께 클릭하면 다음과 같이 회색 점선으로 표시됩니다. 선택된 오브젝트의 속성값을 변경하면 동시에 다같이 적용됩니다.



## 26 폼 위에 버튼을 추가합니다. 각각의 속성을 다음과 같이 변경합니다.



| 컴포넌트    | Property | 값      |
|---------|----------|--------|
| TButton | Caption  | Th 생성  |
| TButton | Caption  | 급여 구하기 |
| TButton | Caption  | Th 해제  |

**Tip****개체 사용 방법 기억 하시나요?**

개체 사용 방법에 대해서 한번 더 정리하고 코딩 하겠습니다.

- ❶ 사용 개체의 유닛을 Uses 합니다.
- ❷ 인스턴스 변수를 선언합니다.
- ❸ 생성자 루틴(Create)을 호출하여 메모리에 할당하고 초기처리합니다.
- ❹ Property, Event, Method등을 사용합니다.
- ❺ Free를 호출하여 파괴자 루틴(Destroy)을 불러내 메모리에서 해제합니다.

**27** THourly의 소스 Utest4는 이미 Implemetation의 Uses절에 추가되어 있는 것을 확인하실 수 있습니다.

```
Implementation
Uses
 Utest4;
```

**28** 변수는 어느 부분에 선언 하시겠습니까? 여기서는 Interface의 Var 부분에 선언은 불가능 합니다. Interface 부분에 Uses가 안되어 있기 때문입니다.

```
Implementation
Uses
 Utest4;
Var
 H:THourly;
```

**29** 생성 버튼의 OnClick 이벤트 핸들러를 작성하여 THourly 개체를 생성하고 초기값들을 화면에 표시해 보겠습니다.

```
H := THourly.Create;
Edit1.Text := H.Name;
Edit2.Text := IntToStr(H.Age);
Edit3.Text := H.Address;
Edit4.Text := H.Office;
Edit5.Text := IntToStr(H.Hrs);
Edit6.Text := IntToStr(H.Rate);
```

### 30 실행하여 결과를 확인해 보겠습니다.



#### 좀 더 자세히!

생성자 호출 시 `H := H.Create`하고 코딩하면 결과는 어떻게 될까요? 당연히 “Access Violation Error”가 발생합니다. 결론적으로 클래스이름, 생성자 루틴을 호출해야 합니다. `TH.Name`이나 `TH.Address`와 같이 인스턴스 변수없이 개체가 가지고 있는 변수에 접근할 수 있을까요?

지금 상태에서는 역시 접근 불가능합니다. 하지만 델파이 2007에서 추가된 `Class Var`나 `Class Function`, `Class Const`, `Class Property` 등은 인스턴스 변수없이 액세스 할 수 있습니다. 개체 관련 추가된 문법은 다음에 더 자세히 설명하겠습니다.

화면을 Close하고 프로그램을 종료하면 메모리에 할당된 `Hourly` 개체는 어떻게 될까요?

해제되지 못한 채 메모리에 남아있게 됩니다. 일반 개체는 소유주 개체가 없기 때문에 필요 없을 시에 반드시 코딩을 통해 해제해 주어야 합니다.

### 31 해제 버튼의 OnClick 이벤트 핸들러를 작성하여 THourly 개체를 해제합니다.

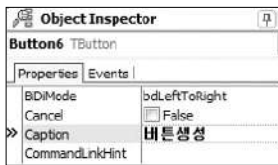
```
H.Free;
```

### 32 “월급구하기” 버튼의 OnClick 이벤트 핸들러를 작성하여 급여를 구해 보겠습니다.

```
Edit7.Text := IntToStr(H.Salary);
```

### 33 이번에는 툴 팔레트에서 마우스 클릭으로 선택했던 TButton 컴포넌트를 수동으로 사용해 보겠습니다.

34 TButton 컴포넌트를 폼 위에 올립니다.



35 버튼 역시 개체 사용 규칙에서 벗어나지 않습니다. 일단 버튼의 소스 StdCtrls.pas는 이미 Uses절에 포함되어 있습니다.

36 이벤트 핸들러 작성에 앞서 변수를 선언합니다.

```
implementation

uses utest4;
var
 Test_btn : TButton;
```

37 OnClick 이벤트를 설정합니다. '버튼생성' 버튼이 할 일은 프로그램 실행 후에 새로운 버튼을 폼 화면에 보여주도록 하는 것입니다.

```
procedure TForm1.Button6Click(Sender: TObject);
begin
 Test_btn := TButton.Create(Self); { '버튼생성' 버튼을 선택하면, 새로운 버튼이
 생성됩니다. }
 Caption := IntToStr(Componentcount); { 프로그램 실행 후 폼 화면 상단에 컴포넌트의
 개수를 보여줍니다. }

 Test_Btn.Parent := Form1; { 설계 시에 화면 위에 올려 놓는 것과 같은 코딩 }
 Test_btn.Top := { '버튼생성' 버튼에 의해 화면에 보여질 'test' 버
 Button5.Top + button5.Height + 24; }
 Test_btn.Height := Button5.Height; {
 Test_btn.Width := Button5.Width; }
 Test_btn.Left := button5.Left;
 Test_btn.Caption := 'test';
 Test_btn.Setfocus; { Test 버튼으로 포커스가 이동합니다. }
End;
```





좀 더 자세히!

### 컴포넌트 생성시 매개변수의 의미는 무엇일까요?

해당 컴포넌트 생성시 소유주 컴포넌트를 설정함으로써 소유주 컴포넌트가 해당 컴포넌트를 관리한다는 의미입니다. ComponentCount를 증가시키고, Components 배열에 인스턴스 변수를 저장해서 소유주 개체가 해제될 때 ComponentCount 만큼 루프를 돌면서 각각의 개체를 해제합니다.  
그러면 어떤 컴포넌트를 주로 소유주 컴포넌트로 사용할까요?

- ❶ Form2 := TForm2.Create(Application);      TApplication 개체를 소유주 컴포넌트로
- ❷ Test\_btn := TTest\_btn.Create(Self);      해당 메소드의 사용 인스턴스 개체를 소유주 컴포넌트로
- ❸ Test\_btn := TTest\_btn.Create(nil);      소유주 컴포넌트가 없으므로 반드시 따로 Free 해야 함

만일 위의 예제에서 Test\_btn을 따로 해제하지 않는다면 화면(TForm1)을 해제할 때 버튼은 해제됩니다.

## 38 이번에는 수동으로 이벤트 핸들러를 선언하고 코딩하며 Test\_Btn의 OnClick 에 Assign하도록 하겠습니다.

```
interface
... (중략) ...
type
 TForm1 = class(TForm)
 ... (중략) ...
 procedure TestHandler(Sender: TObject);
```

interface의 type 부분에 TestHandler 프로시저를 선언합니다. 프로시저를 입력 후, 단축키 Ctrl + Shift + C 를 동시에 누르면 내용을 입력할 수 있는 공간이 자동으로 생성됩니다. 오직 코딩만으로 생성한 'test' 버튼이 실행할 루틴입니다.

```
procedure TForm1.TestHandler(Sender: TObject);
begin
{ 팝업창으로 메시지를 띄워줍니다. }
 ShowMessage('test');
end;
```

### 39 Test\_Btn의 OnClick 이벤트에 작성한 TestHandler을 할당합니다.

```
procedure TForm1.Button9Click(Sender: TObject);
begin
 Test_btn := TButton.Create(Self); — Self는 현재 폴더의 현재오브젝트를 가리킵니다.
 즉 Form1을 의미합니다.

 Caption := inttostr(Componentcount);
 Test_Btn.Parent := Form1;
 Test_btn.Top := Button5.Top + button5.Height + 24;
 Test_btn.Height := Button5.Height;
 Test_btn.Width := Button5.Width;
 Test_btn.Left := button5.left;
 Test_btn.Caption := 'test';
 Test_btn.Onclick := testhandler;
 Test_btn.Setfocus;
 Test_btn.OnClick := Testhandler;
end;
```

### 40 Test\_bnt을 해제하기 위해 버튼을 추가합니다. 버튼에 대해 OnClick 이벤트를 설정합니다.

```
procedure TForm1.Button7Click(Sender: TObject);
begin
 Test_btn.Free; { 'test' 버튼을 폼 화면에서 해제합니다. }
 Caption := IntToStr(ComponentCount); { 컴포넌트의 개수를 보여줍니다. 컴포넌트
end; 개수가 감소합니다 }
```

### 41 단축키 F9를 이용하여 프로그램을 실행합니다.



**Tip****델파이 응용프로그램 작성 방법**

- ❶ File → New → VCL Forms Application으로 새로운 프로젝트를 시작하거나 File → Open Project를 선택하여 기존의 프로젝트를 엽니다.
- ❷ File → New → Form 또는 File → New → Unit을 선택하여 새로운 폼이나 유닛을 추가합니다.
- ❸ 또는 필요 시 프로젝트 매니저를 이용해 작성되어 있는 폼이나 유닛을 추가할 수도 있습니다.
- ❹ 프로젝트로 불러들인 모든 폼은 디폴트로 Auto Create Form으로 설정되어 있습니다.
- ❺ 프로젝트 옵션에서 Available Form으로 이동시키고 Save Project As로 유닛이름과 프로젝트를 저장합니다.
- ❻ 폼 위에 원하는 컴포넌트를 톨 팔레트에서 선택하여 화면을 설계합니다.
- ❼ 각 컴포넌트의 속성을 오브젝트 인스펙터에서 변경합니다.
- ❽ 각 컴포넌트의 원하는 동작 즉, 이벤트를 선택하여 이벤트 핸들러를 작성합니다.
- ❾ 또는 유닛 소스에서 직접 코딩합니다.
- ❿ 각각의 유닛의 인터페이스 부분에 있는 상수, 변수, 개체, 프로시저나 함수등을 Uses하여 마치 자기 소스에 있는 것 처럼 사용합니다.
- ⓫ 프로그램을 실행하여 테스트합니다.

## 8. 클래스 관련 추가된 문법

다음은 클래스와 관련된 추가된 문법들을 정리한 내용입니다.

### Class Helper 개체

델파이 2006에 추가된 개체로 상속을 이용하지 않고 클래스를 확장할 수 있는 방법입니다. 확장되는 클래스를 정상적으로 사용할 수 있는 어떤 곳이든 클래스 헬퍼를 사용할 수 있습니다.

다시 말해서, 변경하고 싶은 내용이 있을 때 코드를 전부 수정할 필요없이 Class Helper를 사용하여 클래스에 기능을 추가하여 변경한 내용을 적용할 수 있습니다.

컴파일러의 식별 영역은 원래의 클래스에 클래스 헬퍼를 더한 만큼이 됩니다. 클래스 헬퍼는 클래스를 확장하는 한 방법을 제공하지만 새로운 코드를 개발할 때 설계 방법으로 취급되어서는 안됩니다.

## 딱 라 하 기

01

```

{ TempClassHelper }

TempClassHelper = class Helper for Temp // Class Helper를 사용
 Function MyEmpno:integer;
end;

Function Tempclasshelper.Myempno: integer;
begin
 Result := Empno;
End;

```

02

```
procedure TForm1.Button10Click(Sender: TObject);
begin
 Emp := TEmp.Create;
 Caption := IntToStr(Emp.MyEmpno);
end;
```

03

## 예 제

다음은 클래스 헬퍼를 이용하여 TListBox에 문자, 숫자, 날짜 등의 데이터를 추가 가능 하도록 하는 예제입니다.

```
TStringsHelper = class helper for TStrings
public
 function Add(const V: Variant): Integer; overload;
 function Add(const Args: array of Variant): Integer; overload;
end;
function TStringsHelper.Add(const V: Variant): Integer;
```

```

begin
 Result := Add([V]);
end;

function TStringsHelper.Add(const Args: array of Variant): Integer;
var
 tmp: string;
 cnt: Integer;
begin
 tmp := '';
 for cnt := Low(Args) to High(Args) do
 tmp := tmp + VarToStr(Args[cnt]);
 Result := Add(tmp);
end;

```

다음은 리스트 박스에 아이템을 추가하는 코드입니다.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
 listbox1.Items.Add('a');
 listbox1.Items.Add(1000);
 listbox1.Items.Add(true);
end;

```

## 클래스 메소드(Class Method)

개체의 동작을 수행하기 위해서는 메소드가 필요합니다. 자동차를 예로 들어 보면, 자동차는 클래스가 됩니다. 그리고 자동차를 구성하는 핸들, 엔진, 바퀴는 개체라고 볼 수 있습니다. 이때 자동차의 움직임인 '달리다' '핸들을 돌리다' 등이 바로 메소드입니다.

개체에 관한 메소드는 보이지 않는 파라미터인 Self(C++의 this와 같은 역할)를 이용하여 개체 자신의 참조자를 전달합니다.

```

Test_btn := TButton.Create(self);

```

위의 예는 Test\_btn이 Self라는 개체 자신의 참조자가 전달되어 생성된 것을 의미합니다. 그러나, 클래스 메소드에는 개체가 없으므로 Self 파라미터는 존재하지 않습니다. 즉, 클래스 메소드란 개체가 아닌 클래스 자체에 대하여 동작을 수행하는 것을 말합니다.

## 클래스 프로퍼티

일반 속성들은 개체 변수를 통해서만 액세스 할 수 있지만 클래스 속성은 클래스 자체에 적용 되는 프로퍼티를 말합니다.

```
{ 클래스 프로퍼티를 액세스 하기 위해서는 정적 클래스로 선언되어야 합니다. }
strict protected
 class function GetX : Integer ; static ;
 class procedure SetX (val : Integer) ; static ;

public
 class property X : Integer read GetX write SetX ; // 클래스 프로퍼티 적용
 class procedure StatProc (s : String) ; static ;
end ;

TMyClass . X := 17 ;
TMyClass . StatProc('Hello') ;
```

## 클래스 정적 메소드

클래스에 정적 클래스 메소드를 추가할 수 있습니다. Delphi 2007 버전부터 추가된 기능입니다. 클래스 정적 메소드는 클래스 타입으로부터 호출할 수 있는 메소드 입니다. 정적 클래스 메소드는 개체에 대한 참조없이도 사용이 가능합니다.

그러나 개체의 멤버들에 접근할 수 없으며, Self 파라미터와 가상 메소드인 virtual로 선언될 수도 없습니다. 이는 다른 클래스 필드, 클래스 프로퍼티, 클래스 메소드와는 다른 차이점 입니다. 즉, 클래스 정적 메소드는 매우 제한적으로 오직 class만이 접근이 가능하고 사용할 수 있습니다.

```
type
 TMyClass = class
 strict private
 class var
 FX : Integer ;
{ 클래스 프로퍼티를 액세스 하기 위해서는 정적 클래스로 선언되어야 합니다. }
 strict protected
 class function GetX : Integer ; static ;
 class procedure SetX (val : Integer) ; static ;
 public
 class property X : Integer read GetX write SetX ; // 클래스 프로퍼티 적용
 class procedure StatProc (s : String) ; static ;
 end ;
 TMyClass . X := 17 ;
 TMyClass . StatProc('Hello') ;
```

## 클래스 변수

클래스 객체가 아닌 클래스 자체에 적용되는 변수를 말합니다.

## 클래스 타입

해당 클래스 내에서만 사용이 가능한 타입 선언을 포함합니다.

```
type
 TClassWithClassType = class
 private // private 선언
 type
 TRecordWithinAClass = record
 SomeField:string;
 end;
 public // public 선언
 class var // 클래스 변수 선언
 RecordWithinAClass:TRecordWithinAClass;
 end;
 . . .
procedure TForm1.FormCreate(Sender:TObject);
begin
 TClassWithClassType.RecordWithinAClass.SomeField := '이것은 클래스 타입으로 선언된 필드입
 니다.';
 ShowMessage(TClassWithClassType.RecordWithinAClass.SomeField);
end;
```

## 중첩된 클래스

클래스 선언 내에서 타입 선언이 포함될 수 있습니다. 이 방법을 이용하여 개념적으로 관계가 있는 타입들을 같이 둘 수 있습니다. 또한 이름 충돌도 피할 수 있습니다.

```
type
 TOuterClass = class // 클래스 선언

 strict private
 MyField:Integer;
 public
 type // TOuterClass 내 타입 선언
 TInnerClass = class // TInnerClass 또 다른 클래스 선언
 public
 MyInnerField:Integer;
 procedure InnerProc;
```

```
end;
procedure OuterProc;
end;

procedure TOuterClass.TInnerClass.InnerProc;
begin
 ...
end;
```

[illegible]

**01** Utest4.pas에 선언한 THourly 클래스에 다음과 같이 클래스 변수와 클래스 프로시저를 추가해 보겠습니다.

```
THourly = class(Temp)
 Hrs, Rate: integer;
 constructor Create; override;
 function Salary: integer; override;
 class procedure Greeting; static;
 class var Etc: string;
end;
```

## 02 구현 부분에 Greeting 함수를 코딩 합니다.

```
class procedure THourly.Greeting;
begin
 ShowMessage('hi');
end;
```

**03** Form1의 THourly 생성 버튼의 코드 부분에 다음과 같이 추가합니다.

```
THourly.Etc := 'Etc';
THourly.Greeting;
H := THourly.Create;
Edit1.Text := H.Name;
Edit2.Text := IntToStr(H.Age);
Edit3.Text := H.Address;
```



```
Edit4.Text := H.Office;
Edit5.Text := IntToStr(H.Hrs);
Edit6.Text := IntToStr(H.Rate);
```

**04** 실행하여 보면 클래스 변수나 클래스 루틴은 개체 인스턴스 없이도 사용가능 하다는 것을 확인할 수 있습니다.



## 제네릭(Generics)

### ■ 제네릭의 개요

제네릭 혹은 제네릭 타입이라는 용어들은 타입으로 파라미터화 될 수 있는 기반내 것들의 집합을 설명합니다. 제네릭이라고 할때, 이는 제네릭 타입과 제네릭 메소드를 의미하기도 합니다. 제네릭은 프로시저나 함수 데이터구조를 루틴이나 데이터 구조가 사용하는 하나 이상의 구체적인 타입으로부터 관계를 끊는 것을 허용하는 추상화 방법들의 집합입니다.

제네릭을 이용하면 명시적 타입을 일일이 명시하지 않고도 타입을 사용하는 코드를 작성할 수 있습니다. 제네릭은 흔히 파라미터 타입이라고 불리는 타입을 구체적으로 명시하지 않는 곳에 적용됩니다. 클래스에서 제네릭을 이용하는 예로는 리스트가 있습니다. 리스트 클래스의 코드를 작성하는 시점에서 리스트에 포함될 아이템들의 타입을 지정할 필요가 없습니다.

### ■ 제네릭 용어들

| 용어                      | 설명                                                                                 |
|-------------------------|------------------------------------------------------------------------------------|
| Type generic            | 실제 타입을 형성하기 위해 제공될 파라미터를 요구하는 타입선언                                                 |
| 제너릭                     | 타입 제너릭과 동일                                                                         |
| Type parameter          | 제너릭 선언이나 바디 내의 다른 선언에서 타입으로 사용되기 위해 제너릭 선언 혹은 메소드 헤더 내에서 선언된 파라미터 실제 타입 인자에 연결될 것임 |
| Type argument           | 인스턴스화 된 타입을 만들기 위해 타입 식별자와 함께 사용되는 타입                                              |
| Closed constructed type | 모든 파라미터들이 실제 타입으로 결정되어있는 구축된 타입                                                    |
| Open constructed type   | 적어도 하나의 파라미터가 타입 파라미터로 구축된 타입                                                      |

아래의 코드는 제네릭 클래스를 구현한 내용입니다.

```

unit Unit_generic1;

interface
type
{ TSIPare = class
 Private
 FKey :String;
 FValue :Integer;
 Public
 Function GetKey:String;
 Procedure Setkey(Key:string);
 Function GetValue:Integer;
 Procedure SetValue(Value:Integer);
 Property Key:string read GetKey write SetKey;
 Property Value:integer read GetValue write SetValue;
end; }
TPair<TKey,TValue> = class
 Private
 FKey :TKey;
 FValue :TValue;
 Public
 Function GetKey:TKey;
 Procedure Setkey(Key:TKey);
 Function GetValue:TValue;
 Procedure SetValue(Value:TValue);
 Property Key:TKey read GetKey write SetKey;
 Property Value:TValue read GetValue write SetValue;
end;

TSIPair = TPair<String,Integer>;
TSSPair = TPair<String,String>;
TIIPair = TPair<Integer,Integer>;
TISPair = TPair<Integer,String>;

TMyProc<T> = Procedure (Param:T);
TMyProc2<Y> = Procedure (Param1,Param2:Y) of object;

TFoo = class
 Procedure Test;
 Procedure MyProc (x,y:Integer);
end;
Procedure Sample (Param:Integer);

implementation
uses

```

```

dialogs,sysutils;

{ TPair<TKey, TValue> }

Procedure Sample(Param:Integer);
begin
 showmessage(inttostr(Param));
end;

function TPair<TKey, TValue>.GetKey: TKey;
begin
 RESULT := FKey;
end;

function TPair<TKey, TValue>.GetValue: TValue;
begin
 Result := FValue;
end;

procedure TPair<TKey, TValue>.Setkey(Key: TKey);
begin
 FKey := Key;
end;

procedure TPair<TKey, TValue>.SetValue(Value: TValue);
begin
 FValue := Value;
end;

procedure TFoo.MyProc(x, y: Integer);
begin
 SHOWmessage(format('X :%d Y: %d', [X,Y]));
end;

procedure TFoo.Test;
Var
 X :TMyProc<Integer>;
 Y :TMyProc2<Integer>;
begin
 X := Sample;
 X(10);
 Y := MyProc;
 Y(20,30);
end;
end.

```

## 익명 메소드(Anonymous Methods)

익명 메소드는 개발자들이 파라미터로 코드 블록을 전달할 수 있게 하는 코드 구조입니다. 이것은 이름이 지정되지 않은 프로시저나 함수를 말합니다. 익명 메소드는 코드 블록을 변수에 할당하거나 다른 메소드의 파라미터로 지정할 수 있는 것으로 취급합니다. 또한 익명 메소드는 해당 메소드가 정의된 컨텍스트에서 변수를 참조하고 값을 바인딩 할 수 있습니다. 따라서 텔 파이의 익명 메소드는 코드 블록이 전달될 때 그 상태를 캡처하는 완전한 클로저 타입입니다.

Type

```
TMethodPointer = Procedure of object;
TStringToInt = Function(x:string):integer of object;

TSimpleProcedure = reference to procedure;
TSimpleFunction = reference to function(x:string):integer;
var
 x: TMethodPointer;
 y: TStringToInt;
 obj:tobj;

 x1: TSimpleProcedure;
 y1: TSimpleFunction;
```

익명 메소드는 다음의 예와 같이 일반적으로 어떤 것에 의해 대입됩니다.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 x1 := procedure
 begin
 showmessage('hello world');
 end;
 x1; // 방금 정의된 루틴 호출

 y1 := function(x:string):integer
 begin
 result := length(x);
 end;
 showmessage(inttostr(y1('aaaa')));
end;
```

익명 메소드는 또한 함수에 의해 리턴 되거나 메소드를 호출할 때 파라미터 값으로 전달될 수도 있습니다.

```
Type
 TFuncOfIntToString = reference to function(x:integer):string;

Procedure AnalyzeFunction(proc: TFuncOfIntToString);
Begin
...
End;

// 익명 메소드를 변수로서 파라미터로 전달하면서 프로시저호출
AnalyzeFunction(myFuncnt);
// 익명 메소드를 직접사용
AnalyzeFunction(function(integer):string;
Begin
 Result := inttostr(x);
End;
```

#### ■ 익명 메소드 변수의 바인딩

익명 메소드의 핵심 기능은 익명 메소드가 정의된 위치에서 보이는 변수들을 참조할 수 있다는 것입니다. 이들 변수들은 익명 메소드에 대한 참조에 바인드 되고 구속됩니다. 따라서 익명 메소드는 상태를 캡처하고 변수의 수명을 연장하게 됩니다.

```
function MakeAdder(y:integer):TFuncofInt;
begin
 //anonymous method
 result := function(x: Integer)
 begin
 result := x + y;
 end;
end;

var
 adder:TFuncofInt;
begin
 adder := MakeAdder(20);
 writeln(adder(22)); //42 출력
end;
```

## ■ 익명 메소드의 유용성

- 변수값들을 연결(bind)할 수 있습니다.
- 메소드를 정의하고 사용하는 편리한 방법입니다.
- 코드를 파라미터화 하는 것이 쉽습니다.

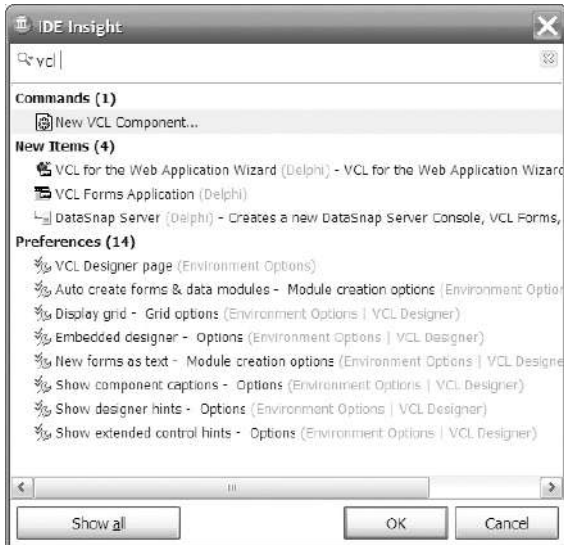
## 9. 기타 환경

### 도움말 사용하기

프로그램 코드를 작성하는 중에 또는 델파이 개발 환경을 사용하는 도중에 잘 모르는 부분이 생겼을 경우에는 언제 어디서든 F1키를 이용하면 온라인 도움말을 볼 수가 있습니다.

### IDE Insight

IDE 인사이트는 개발자가 IDE 내의 모든 기능, 즉 프로젝트, 컴포넌트, 코드 템플릿, 구성 세팅 등 어느 기능이든 빠르게 찾아낼 수 있도록 해줍니다. 간단히 F6 키를 누르기만 하면 IDE 인사이트가 나타나며, 여기서 개발자는 원하는 기능을 타이핑하여 사용 가능한 기능들의 목록을 볼 수 있습니다. 여기서 개발자가 특정 기능을 선택하면 해당 기능이 실행되거나 IDE에서 그 위치로 이동하게 됩니다. IDE 인사이트는 키 입력만으로 전체 IDE 기능들을 호출할 수 있습니다.



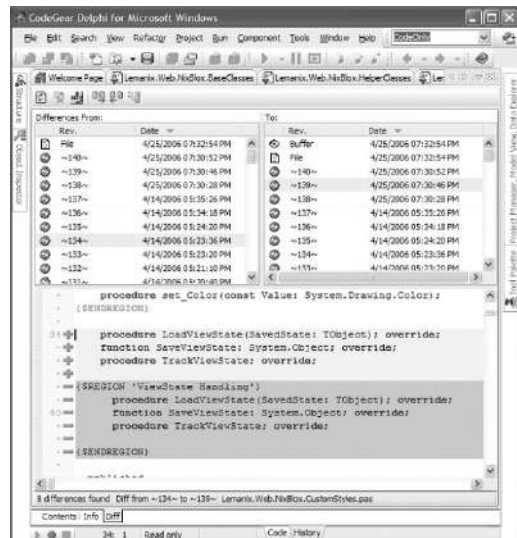
## File Explorer

IDE는 머신 하드 드라이브에 있는 파일들을 액세스할 수 있게 해주는 파일 익스플로러를 가지고 있습니다. 파일들은 프로젝트에 추가되거나 코드 에디터에서 열 수 있습니다. 파일 브라우저는 다른 도킹 가능 윈도우들처럼 IDE에 도킹될 수 있습니다.



## History Tab

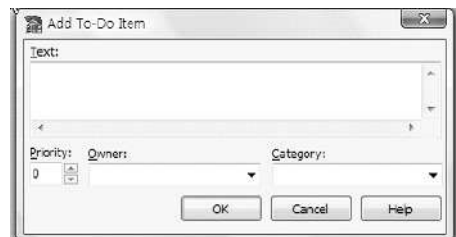
히스토리 탭은 간단한 소스 컨트롤 시스템의 기능을 합니다. 파일이 저장될 때마다 서버 디렉토리에 백업 파일이 만들어집니다. IDE는 diff 엔진을 이용하여 이런 변경들을 추적하여 개발자가 이전 버전들을 살펴볼 수 있게 해줍니다. 파일의 이전 버전이 필요한 경우, 그 버전으로 기존의 파일을 교체할 수 있습니다. 아래의 그림은 히스토리 탭이 한 파일의 두 개의 이전 버전을 비교해서 보여주고 있는 모습입니다. 코드 에디터상에서 아래의 탭 중 History 탭을 선택하시면 됩니다.



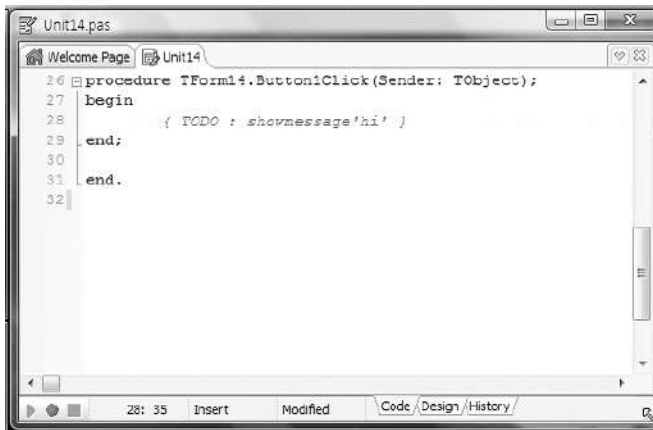
## To Do List

프로젝트 개발 스케줄이나 하고자 하는 일들을 목록에 기록해 두었다가 사용할 수 있습니다. 광범위한 프로젝트 항목들의 관리를 쉽게 해주고 소스 코딩에 직접 이용할 수 있습니다.

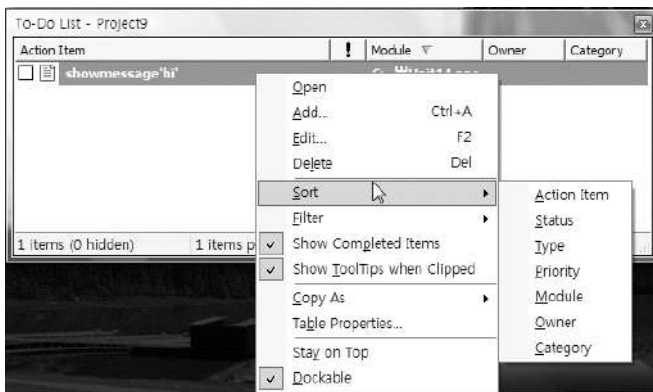
- 1 소스 코딩 하고자 하는 곳에서 오른쪽 마우스 버튼을 눌러 Add To Do Item을 선택하면 화면이 표시됩니다.



- 2 코딩의 내용이나 우선 순위 등을 설정하면 소스 라인에 주석으로 기록이 됩니다.



- 3 작업 시에 View → To Do List를 선택하면 설정되어 있는 목록이 표시됩니다. 목록이 표시 될 때 정렬 순서를 선택할 수 있습니다.



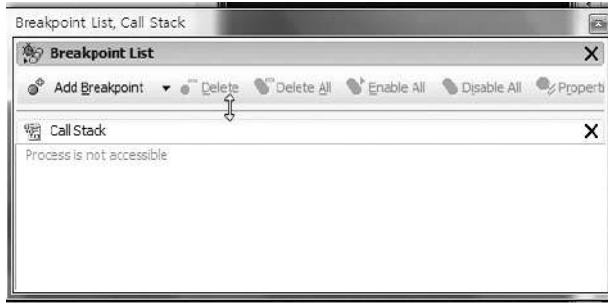
- 4 원하는 Item에서 더블 클릭하면 해당하는 소스로 이동하여 코딩할 수 있습니다.
- 5 코딩 후 항목에 체크 표시를 하여 작업의 상태를 표시합니다.



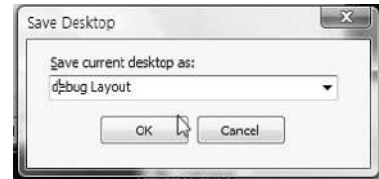
## Desktop 저장

개발 환경을 저장해 두었다가 재사용할 수 있게 해줍니다. 즉 열려있는 창의 상태라든지, 위치 등을 저장하는 것입니다. 특히 개발 환경이 다른 디버그 시의 환경 등을 저장해 두었다가 사용하면 편리합니다.

- 1 디버그 시에 사용할 수 있도록 윈도우 화면 등을 아래와 같이 설정합니다.



- 2 View → Desktops → save Desktop을 선택하면 아래의 화면이 표시되고 특정 이름으로 저장합니다.



- 3 View → Desktop → Save Desktop에서 저장한 Desktop을 선택하면 지정 환경으로 설정됩니다.

