오브젝트 파스칼의 깊은 곳

(Advanced Concepts of Object Pascal)

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

Sender 파라미터의 이용

서로 다른 컴포넌트 들에 대한 서로 다른 동작을 하나의 이벤트 핸들러로 처리하고 싶을 때가 있다. 이럴 때에는 Sender 파라미터를 이용하면 도움이 된다.
Sender 파라미터는 델파이에게 이벤트를 실제로 받은 컴포넌트가 무엇인지를 알려주는 역할을 한다. 다음의 코드는 버튼에 따라서 폼의 캡션에 타이틀을 표시하기도 하고, 아무것도 보여주지 않기도 한다.

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

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

팀 개발 환경에서의 객체 공유

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

1. Tools|Environment Options 메뉴를 선택한다.

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

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

예외 처리 (Handling exceptions)

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

- 코드 블록의 보호 (Protecting blocks of code)

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

- 예외 처리 방법

에러가 발생하면, 어플리케이션은 예외 객체를 생성한다. 예외를 처리하는 방법은 크게 나누어 클린업(clean up) 코드를 작성하는 방법과 직접 예외를 처리해 주는 방법으로 나눌 수 있다.
클린업 코드를 실행하는 것이 가장 간단한 방법이다. 이 경우 에러를 발생시킨 조건을 해결하지는 못하지만, 어플리케이션이 불안정한 상태로 가는 것을 막는 역할을 한다. 즉, 할당된 리소스를 해제하는 것을 예로 들 수 있다. 그에 비해 직접 예외를 처리하는 방법은 실제 에러 상황을 처리하여 예외 객체를 파괴한 후, 어플리케이션이 계속 실행되도록 하는 것이다.

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

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

이 코드에서는 Font 객체의 이름과 크기를 설정할 때 에러가 발생하면 '삑' 소리를 내도록 처리하였다.

- 리소스 할당의 보호 (Protecting resource allocations)

견고한 어플리케이션을 제작함에 있어서 리소스를 할당하고, 이를 해제하는 것은 매우 중요한 부분이다. 예를 들어, 어플리케이션에서 메모리를 할당했으면 이를 반드시 해제해야 하며, 파일을 열었으면 이것은 반드시 닫아주어야 한다.
그러므로, 메모리를 할당하는 등의 리소스를 할당하는 문장이 있을 때에는 이를 해제하는 루틴을 블록으로 처리해 주는 것이 바람직하다.
보호해야 할 리소스로는 파일, 메모리, 윈도우 리소스, 델파이 객체 등이 있으며 이들을 보호하기 위해서는 try ... finally 구문을 사용한다.
그러면, 실제 예를 살펴 보자. 다음의 코드는 메모리를 할당하고 해제하는 루틴이 포함되어 있지만 에러가 발생할 경우 메모리의 해제가 되지 않는다.

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

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

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

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

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

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

- RTL 예외의 처리

수학 함수나 파일 처리 프로시저 등의 RTL(run-time library)의 루틴을 호출하는 코드를 사용할 때 RTL 은 에러가 발생할 때 어플리케이션에 예외를 넘겨준다. 디폴트로 RTL 예외는 어플리케이션에서 사용자에게 메시지를 보여주는데, 이때 RTL 예외를 처리하기 위해 자신의 예외 처리 루틴을 정의해서 쓸 수 있다.
RTL 예외는 SysUtils.pas 유닛에 정의되어 있다. 이들은 Exception 객체에서 상속된 것으로 이 객체는 디스플레이할 메시지를 문자열로 제공한다. RTL 에 의해 발생하는 예외에는 다음의 7 가지가 있다.

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

- 예외처리 구문의 작성

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

```
try
```

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

어플리케이션은 try 파트의 구문을 실행하다가 예외가 발생하면 except 파트의 구문을 실행하게 된다. 이때 try 파트에서 다른 루틴을 호출할 경우 호출된 루틴에서 적절한 예외처리가 되지 않으면 역시 except 파트의 구문이 실행된다.
이때 예외가 발생하면 바로 except 파트로 넘어가므로, 그 뒤의 구문들은 실행되지 않는다.
일단 어플리케이션이 예외처리 구문으로 넘어가면 예외 객체는 자동으로 파괴된다.
예외 처리 구문의 기본적인 문법은 다음과 같다.

```
on <예외의 종류> do <처리 구문>;
```

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

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

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

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

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

- 예외 인스턴스의 사용 (Using the exception instance)

대부분의 경우 예외처리 구문은 예외의 형을 제외한 다른 정보는 필요로 하지 않는다. 그런데, 가끔 예외 인스턴스에 담긴 정보를 필요로 하는 경우가 있다. 대부분은 예외 인스턴스의 특정 정보를 읽어서 이를 예외처리에 활용하는 경우이다.
예를 들어, 하나의 폼에 스크롤 바와 버튼을 하나씩 추가한 후 버튼의 이벤트 핸들러에 다음과 같이 작성했다고 하자.

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

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

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

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

- 예외처리 구문의 범위와 디폴트 예외처리 구문

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

블록이 특정 예외를 처리하지 않는다면, 예외는 블록을 벗어나게 되며 블록을 호출한 코드나 블록이 포함된 바깥 블록으로 제어가 넘어간다. 여기서도 예외 처리가 없으면 예외 객체는 계속 없어지지 않고, 예외 처리가 있을 때까지 계속 바깥 블록으로 이동한다.
예외 처리에 있어서 해당되는 예외가 없을 때에는 디폴트 예외처리 구문을 작성하여 사용하는 경우가 있다. 이럴 때에는 다음과 같이 예외처리 블록의 except 파트에 else 파트를 추가하여 작성하면 된다.

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

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

- 예외의 raise

예외를 로컬에서 처리할 때, 예외 처리 블록이 끝나면 예외 객체가 해제된다는 것은 이미 언급한 바 있다. 그런데, 가끔은 예외 처리 블록이 끝난 이후에 이 예외를 다시 처리하고 싶을 때가 있다. 이럴 때에는 예외 객체가 해제되는 것을 막을 필요가 있는데, 이럴 때에 사용하는 키워드가 raise 이다.
예를 들어, 예외가 발생하면 메시지를 사용자에게 보여주고 그 이후에 표준적인 처리를 하고 싶다고 하자. 이를 위해서는 로컬 예외처리 구문을 선언하여 메시지를 보여주고, raise를 호출하여 다음에 다시 이를 처리할 수 있도록 해야 한다. 다음은 이를 위한 pseudo-code 이다.

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

```
      {특별한 경우에만 처리하는 구문}
      raise;                    {예외 객체가 해제되지 않도록 한다}
    end;
  end;
except
  on ESomething do ...;       {이곳에서 다시 예외 처리를 할 수 있다}
end;
```

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

● 사용자 정의 예외의 정의

델파이에서는 자신이 예외 객체를 선언해 놓고, 이를 어기는 경우에 예외처리를 하는 식으로 프로그래밍할 수 있다.
예외는 객체이기 때문에, 새로운 종류의 예외는 새로운 객체 형을 선언하는 것과 마찬가지로 간단하다. 기본적으로 예외 객체의 조상은 Exception 이다. 그러므로, 다음과 같이 Exception 객체 또는 Exception 객체에서 상속 받은 객체를 상속해서 선언해야 한다.

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

여기서 개발자가 EMyException 예외를 발생시키되, EMyException 에 대한 핸들러를 제공하지 않으면 Exception 객체에 대한 디폴트 예외처리 구문이 실행된다.
특정 예외 객체를 일으키기 위해서는 앞에서도 잠시 살펴본 바 있는 raise 키워드를 이용한다. 이때 raise 뒤에 일으키게될 예외 객체를 적어준다는 점이 다르다. 예외처리 구문이 예외를 실제로 잘 처리하게 되면, 예외 객체가 파괴된다.
실제로 사용하는 예를 살펴보자. 다음의 코드는 패스워드를 잘못 입력한 경우에 사용자가 정의한 EPasswordInvalid 예외를 발생시킨다.

```
type
```

```
  EPasswordInvalid = class(Exception);

if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('패스워드가 잘못 되었습니다 !');
```

참고: at 키워드

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

```
raise object at location;
```

실제로 SysUtils.pas 소스 코드에는 다음과 같은 문장이 있다.

```
rasie OutOfMemory at ReturnAddr;
```

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

문자열 포맷하기 (Formatting Strings)

델파이는 기본적인 텍스트로 된 문자열, 서식 문자, 각각의 서식 문자에 대한 배열을 그 파라미터로 이용하여 다양한 가공을 할 수 있는 Format 함수를 제공한다.
Format 함수의 선언부는 다음과 같다.

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

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

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

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

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

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

이와 같은 형식을 사용하면 정수 i 의 8 자리가 오른쪽 정렬 형태로 출력하게 되고, 왼쪽은 공백으로 채워진다.
포맷 지정자는 다음과 같은 형태를 가진다.

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

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

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

변환 문자로 쓰일 수 있는 것으로는 다음과 같은 것들이 있다.

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

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

부동 소수점 값의 경우 1000 단위를 나타내는 문자열이나 10 진수를 나타내는 문자열은 각각 ThousandSeparator, DecimalSeparator 전역 변수에 의해 결정된다.
인덱스, 너비, 정밀도 지정자(Index, width, and precision specifiers)는 '%10d'와 같이 직접적으로 지정할 수도 있고, '*' 문자를 이용하여 '%*.*f'와 같이 간접적으로 지정할 수도 있다. 예를 들어, 다음의 문장을 살펴 보자.

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

이것은 결국 다음 문장과 같은 역할을 하게 된다.

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

너비 지정자는 변환에 필요한 가장 작은 필드의 너비를 결정한다. 만약 결과 문자열이 이보다 작으면, 공백이 채워지게 된다. 디폴트로는 우측 정렬에 맞도록 좌측에 공백이 채워지지만, '-' 문자열로 좌측 정렬을 지정하면 우측에 공백이 채워진다.
인덱스 지정자는 현재의 argument 리스트 인덱스를 지정된 값으로 설정한다. 첫번째 argument 의 인덱스 값이 0 이며, 이를 이용하여 같은 argument 를 여러 차례 사용하는 것이 가능하다. 예를 들어, 다음의 코드는 '10 20 10 20'이라는 문자열을 만들어 낸다.

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

인터페이스의 활용 (Using interfaces)

델파이 3 부터 추가된 인터페이스 개념은 오브젝트 인터페이스에 의해 구현되는데, 이는 클라이언트 코드가 특정 객체의 구현된 내용에 대해 알 필요가 없이 객체에 대해 접근할 수 있는 방법을 제공한다. 인터페이스의 선언은 실제적인 인터페이스의 구현을 포함하지는 않는다. 인터페이스는 다만 선언일 뿐이며, 나중에 클래스의 한 부분으로 구현되게 된다.
델파이의 인터페이스는 DCOM 뿐만 아니라 델파이 4 에서부터는 CORBA 기술의 스펙과 완벽하게 호환되므로, 나중에 쉽게 COM 이나 CORBA 객체를 생성하고, 사용하게 되는 근간이 된다.

- 기본 문법

다음의 코드는 인터페이스 선언의 예이다.

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

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

메소드, 프로퍼티 등의 멤버들을 선언하게 된다.

참고: 인터페이스의 일반적인 규칙

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

1. 인터페이스 메소드는 추상적이다.

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

2. 인터페이스는 포인터로 접근한다.

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

3. 인터페이스는 특정한 기능을 캡슐화한다.

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

4. 인터페이스는 유일한 ID 를 가진다.

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

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

5. 인터페이스는 변하지 않는다.

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

6. 인터페이스는 그들의 궁극적인 조상으로부터 기본적인 기능을 상속받는다.

모든 인터페이스는 IUnknown 인터페이스로부터 직간접으로 상속 받는다. 이 인터페이스는 인터페이스의 기본적인 기능을 정의한다. 그 내용은 인터페이스를 어떻게 부르며, 생성, 파괴할 것인지에 관한 것이다. 이를 정의한 메소드를 가상함수 테이블의 순서대로 나열하면 QueryInterface, AddRef, Release 이다. 인터페이스를 구현한 어떤 클래스도 반드시 위의 세가지 메소드와 조상 인터페이스의 메소드, 자신이 선언한 메소드는 반드시 구현해야 한다.
QueryInterface 는 클라이언트가 주어진 객체를 질의하고, 지원하는 인터페이스의 포인터에 접근하는 방법을 제공한다. AddRef 와 Release 는 간단한 참조계수(reference count)를 관리하는 메소드로, 객체가 적절할 때에 제거될 수 있도록 해준다. 참조계수가 0 이 아니면 객체는 메모리에 남아 있게 되며, 0 이 되면 인터페이스가 안전하게 객체들을 제거할 수 있다.

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

- 인터페이스를 제작하고 접근하기

인터페이스는 기본적으로 COM, OLE, 서드파티 라이브러리에서 많이 제공하고 있다. 여기에 자신의 인터페이스를 추가할 수도 있는데, 이렇게 하려면 인터페이스 정의를 쓰고, 이 인터페이스를 구현하는 클래스를 제작하면 된다.
델파이에서 인터페이스를 제작하는 과정을 나열해 보면 다음과 같다.

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

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

● 인터페이스 메소드의 구현

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

1. 인터페이스 메소드의 매핑

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

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

디폴트로 각각의 인터페이스 메소드는 클래스내의 같은 이름의 메소드에 매핑된다.

2. 이름이 다른 메소드의 매핑

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

type

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

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

```
type

  IWindow = interface
    procedure Draw;
    ...
  end;

  ICanvas = interface
    procedure Draw;
    ...
  end;

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

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

- 인터페이스 데이터 형 호환성과 변환

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

래스와의 호환성을 '대입호환(assignment compatible)'이라고 한다. 예를 들어 클래스 T 가 인터페이스 I1, I2 를 구현한다고 하면, T 는 I1, I2 와 대입호환된다.
또한, 어떤 인터페이스도 가변형(Variant type)과는 대입호환된다. 만약 인터페이스가 IDispatch type 이거나 그 자손일 경우, 결과의 가변형 값은 varDispatch 라는 type 코드를 가지게 된다. IDispatch type 이 아니면 varUnknown 이라는 코드를 가지게 된다.
인터페이스의 형변환(type casting)은 클래스에서와 비슷한 규칙을 따른다. 클래스는 IntfType(X)의 형태로 인터페이스로의 형변환이 가능하다. 이 때, X 는 클래스 형이며, IntfType 은 인터페이스 형이다.
인터페이스 형의 값은 Variant(X)의 형태로 가변형변수(Variant)로 형변환될 수 있는데, 이 때 X 는 인터페이스 형이다. X 가 IDispatch 이거나 그 자손일 경우 결과가 되는 가변형 코드는 IDispatch 가 되며, 그렇지 않으면 IUnknown 이 된다.
각 가변형 변수도 IUnknown 이나 IDispatch type 의 인터페이스로의 형변환이 가능한데, 이렇게 하려면 IUnknown(X), IDispatch(X)와 같이 쓰면 된다. 이 때 X 는 가변형 변수 이며, 이 때의 type 코드는 IUnknown 으로 형변환될 경우에는 varEmpty, varUnknown, varDispatch 중의 하나이어야 하며, IDispatch 로 형변환될 경우에는 varEmpty, varDispatch 중의 하나이어야 한다.

- 인터페이스 참조하는 법

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

```
type

  IWindow = interface
    ...
  end;

  IDragDrop = interface
    ...
  end;

  TEditWindow = class(TObject, IWindow, IDragDrop)
    ...
  end;
```

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

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

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

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

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

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

- 인터페이스 질의

인터페이스 참조는 위에서 설명한 객체 참조(object reference)에도 as 연산자를 써서 참조할 수도 있는데 이를 인터페이스 질의(interface querying)이라고 한다. 인터페이스 질의 연산은 Reference as Interface 의 형태를 가진다. 여기서 Reference 는 IUnknown 인터페이스나 그 자손 인터페이스를 구현한 클래스이며, Interface 는 인터페이스의 type 결정자이다. 이 연산의 결과로 주어진 인터페이스 type 을 참조할 수 있게 된다. 이 때 질의된 객체나 인터페이스가 주어진 인터페이스를 구현하지 못하면 예외가 발생한다.
객체의 경우 as 연산자는 객체가 IUnknown 인터페이스를 구현하고 있을 경우에만 사용할 수 있다. 쓰는 형태는 Reference as Interface 로 동일한데, Reference 가 객체이면

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

- 대리(delegation)를 통한 인터페이스의 구현

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

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

이 코드는 MyInterface 라는 프로퍼티를 선언하고, 이 프로퍼티는 ImyInterface 인터페이스를 구현한다는 의미이다. implements 지시어는 선언부의 마지막에 위치하여야 하며, 하나 이상의 인터페이스를 나열할 경우에는 콤마로 구별한다.
대리(delegate) 프로퍼티는 다음과 같은 조건을 충족시켜야 한다.

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

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

```
type

  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

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

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

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

대리 프로퍼티가 클래스 형일 경우의 사용 예는 다음과 같다.

```
type

  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyImplClass = class
    procedure P1;
    procedure P2;
  end;

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

procedure TMyImplClass.P1;
...

procedure TMyImplClass.P2;
...

procedure TMyClass.MyP1;
...
```

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

메소드 포인터

메소드 포인터에 대해서는 4 장에서 프로시저 형과 함께 언급한 바 있지만, 델파이에서는 그 중요성이 크기 때문에 다시 한번 알아보고 넘어가도록 하자.
메소드 포인터란 메소드 코드의 주소와 객체 인스턴스의 주소를 모두 가지게 되는 포인터로, 객체 인스턴스의 주소는 메소드 내부에서 Self 로 표현한다. 메소드 포인터 형의 선언은 프로시저 형의 선언과 동일하지만 맨 마지막에 of object 라는 키워드가 추가된다는 점이 다르다.
그러면 메소드 포인터를 하나 선언해 보자.

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

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

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

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

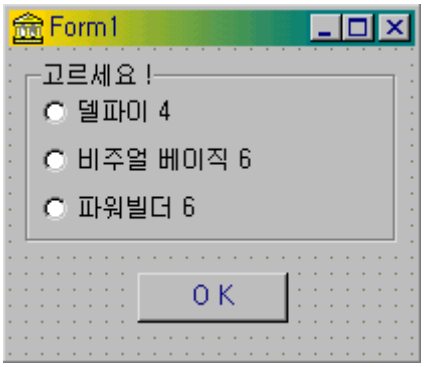
다른 클래스가 있다고 하자.

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

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

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

이것이 델파이 컴포넌트에서 가장 중요한 기술의 하나라고 할 수 있는 대리(delegation) 기법이다. 메소드 포인터가 있는 객체가 있으면, 새로운 메소드를 그 객체에 대입하면 자유롭게 객체의 동작을 바꿀 수가 있는 것이다.
다소 복잡하다고 생각될 수 있겠지만, 델파이 컴포넌트의 대부분의 이벤트 핸들러 등에서 이런 기법은 자주 사용된다. 예를 들어, 버튼에 대해 OnClick 이벤트 핸들러를 추가하면 델파이는 버튼의 OnClick 이라는 이름의 메소드 포인터를 사용자가 제작한 이벤트 핸들러를 가리키도록 하여, 버튼이 클릭될 때 이벤트 핸들러가 호출되는 것이다.
그러면, 메소드 포인터를 확실하게 이해할 수 있는 예제를 하나 만들어 보자.
이번에 만들 예제는 동적으로 이벤트 핸들러를 바꾸어 주는 방법을 소개하는 것으로, 이것이 가능한 이유가 바로 메소드 포인터를 활용할 수 있기 때문이다.
새로운 어플리케이션을 시작하고 폼에 TRadioGroup, TButton 컴포넌트를 하나씩 올려 놓고, 각각의 캡션을 '고르세요 !', 'OK'로 설정하고 RadioGroup1 의 Items 프로퍼티를 다음 그림과 같이 '델파이 4, 비주얼 베이직 6, 파워빌더 6'로 설정한다.



이제 코드 에디터에서 메소드 포인터를 선언하도록 하자. 지금 하려고 하는 작업은 Button1 의 OnClick 이벤트 핸들러를 전혀 작성하지 않고, 순전히 메소드 포인터의 대입을 통해서 이를 구현하려는 것으로 폼이 시작될 때 기본적인 이벤트 핸들러의 역할을 할 Start 라는 프로시저를 OnClick 메소드 포인터에 대입하고, 라디오 버튼을 클릭할 때마다 서로 다른 메소드 포인터를 대입하여 여러 프로시저를 실행하게 할 것이다.
메소드 포인터를 대입할 수 있으려면, Button1 의 OnClick 메소드 포인터와 파라미터의 수와 데이터 형이 일치하여야 한다. OnClick 메소드 포인터는 TNotifyEvent 라는 메소드 포인터 형으로 선언되어 있는데, 델파이 컴포넌트의 많은 이벤트 들이 이 메소드 포인터 형이다. TNotifyEvent 메소드 포인터는 Classes.pas 유닛에 다음과 같이 선언되어 있다.

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

그러므로, 우리가 작성할 이벤트 핸들러 들은 파라미터의 수와 데이터 형을 여기에 맞게 작성하면 된다. 그러면 폼이 처음 시작할 때의 이벤트 핸들러를 Start 라고 하고 델파이 4, 비주얼 베이직 6, 파워빌더 6 라디오 버튼이 선택된 경우의 이벤트 핸들러를 각각 Delphi4, VisualBasic6, PowerBuilder6 라고 하고 이들을 TForm1 의 선언부에 다음과 같이 추가한다.

```
TForm1 = class(TForm)
    RadioGroup1: TRadioGroup;
    Button1: TButton;
    procedure Start(Sender: TObject);
    procedure Delphi4(Sender: TObject);
    procedure VisualBasic6(Sender: TObject);
    procedure PowerBuilder6(Sender: TObject);
    ... (후략)
```

그리고, 메소드 포인터를 대입하는 코드를 작성하기 전에 이들 각각을 다음과 같이 구현하도록 하자.

```
procedure TForm1.Start(Sender: TObject);
begin
  ShowMessage('선택하신 것이 없군요 !');
end;
```

```
procedure TForm1.Delphi4(Sender: TObject);
begin
  ShowMessage('당신의 선택은 100 점 !');
end;

procedure TForm1.VisualBasic6(Sender: TObject);
begin
  ShowMessage('조금만 더 생각해 보세요 !');
end;

procedure TForm1.PowerBuilder6(Sender: TObject);
begin
  ShowMessage('옳은 선택일까요 ?');
end;
```

무슨 내용일까 ? 판단은 자유에 맡기기로 한다. 이 글을 읽고 있는 사람들은 델파이를 사랑하는 사람 들일 것이므로 다소 장난기가 있는 코드이지만 모두 이해할 것으로 믿는다.
그러면, 이제 실제로 메소드 포인터를 대입하도록 하자. 먼저 폼이 생성될 때에는 Start 를 대입해야 하므로 Form1 의 OnCreate 이벤트 핸들러를 다음과 같이 작성한다.

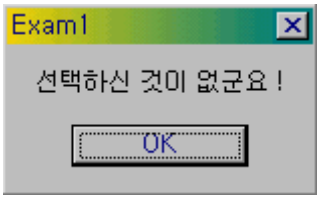
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.OnClick := Start;
end;
```

그리고, 각각의 라디오 버튼을 선택했을 때의 변화를 위해 RadioGroup1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

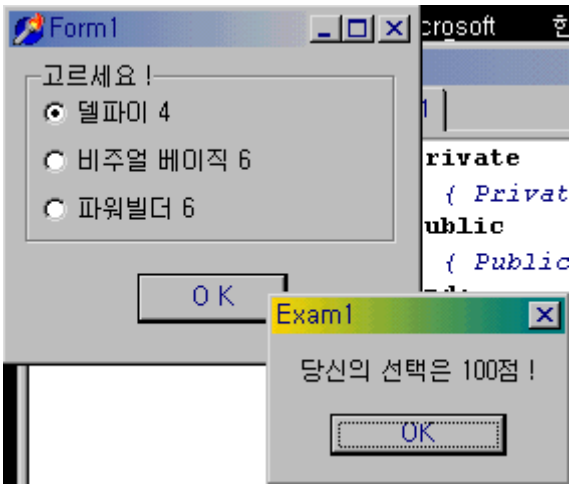
```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0: Button1.OnClick := Delphi4;
    1: Button1.OnClick := VisualBasic6;
    2: Button1.OnClick := PowerBuilder6;
  end;
```

```
end;
```

대입 호환이 보장되므로, 이렇게 프로시저의 이름을 대입하는 것으로 모든 작업은 간단히 끝난다. 그러면 이를 실행해 보자. 실행된 직후에 버튼을 클릭하면 다음과 같은 메시지를 볼 수 있을 것이다.



그러면, 델파이 4 을 선택하고 버튼을 클릭해보자. 다음과 같은 메시지를 볼 수 있다면 제대로 실행된 것이다.



클래스 참조 (Class Reference)

클래스 참조란 클래스 형에 대한 참조(reference)를 일컫는 말이다.
클래스 참조는 클래스의 인스턴스에 대한 조작을 하기 보다는, 클래스 자체에 대한 작업을 하려고 할 때 사용된다. 이럴 때에는 변수나 파라미터를 클래스 자체를 값으로 가지게 할 필요가 있는데, 클래스 참조형(class reference type)을 사용하면 된다.
클래스 참조형은 다른 말로 메타 클래스라고도 하며, 다음과 같이 클래스에 대한 클래스를 선언하면 된다.

```
class of class1;
```

참고로 델파이의 TClass 형은 다음과 같이 선언된 클래스 참조형이다.

```
type
  TClass = class of TObject;

var
  AnyObj: TClass;
```

AnyObj 변수는 어떤 클래스도 참조할 수 있다. 클래스 참조형에 대해 가장 전형적인 사용 예를 든다면 TCollection 클래스의 constructor 에 대한 선언부를 들 수 있다. 다음의 코드를 살펴 보자.

```
type
  TCollectionItemClass = class of TCollectionItem;
  ... (중략)

constructor Create(ItemClass: TCollectionItemClass);
```

이 선언부의 의미는 TCollection 인스턴스 객체를 생성하려면, 반드시 constructor 에 TCollectionItem 에서 상속받은 클래스의 이름을 파라미터로 넘겨주어야 한다는 것이다.
클래스 참조는 다른 OOP 언어에서 사용하는 메타 클래스 개념과 비슷하지만, 오브젝트 파스칼에서 클래스 참조는 클래스가 아니라 일종의 데이터 형에 대한 포인터일 뿐이다.
클래스 참조의 가장 큰 장점은 클래스 데이터 형을 실행 도중에 조작할 수 있다는 점과 선언된 클래스 이외에 어떤 서브 클래스이든 대입할 수 있다는 것이다. 이런 측면에서 보면 모든 클래스는 TObject 에서 파생되므로, 앞에서 예를 든 TClass 클래스 참조형은 델파이로 작성하는 어느 클래스의 참조를 저장할 때에도 사용할 수 있다. 예를 들어, Application 객체의 CreateForm 메소드는 만들어 낼 폼의 클래스를 파라미터로 요구한다.

```
Application.CreateForm(TForm1, Form1);
```

이 코드에서 첫번째 파라미터는 클래스 참조이고, 두번째는 실제 객체이다.
그렇다면 델파이에서의 클래스 참조의 활용 방법에는 어떤 것이 있을까 ? 델파이에서 새로운 컴포넌트를 폼에 추가하면, 데이터 형을 선택하고 그 데이터 형에 대한 객체를 만들어 낸다. 이 작업은 클래스 참조를 활용하여 델파이가 해주는 작업이다.

정 리 (Summary)

이번 장에서는 예외 처리, 인터페이스, 메소드 포인터와 같은 비교적 고급스러운 오브젝트 파스칼의 면면을 살펴 보았다. 이런 사항 들은 몰라도 대부분의 어플리케이션을 작성할 수 있지만, 보다 효율적이고 잘된 프로그램을 개발하려면 알아두면 많은 도움이 될 것이다.
다음 장에서는 VCL 계층 구조와 의미, 일부 컴포넌트의 사용 방법 등에 대해서 알아볼 것이다.