기본적인 컴포넌트의 제작

(Creating Basic Components)

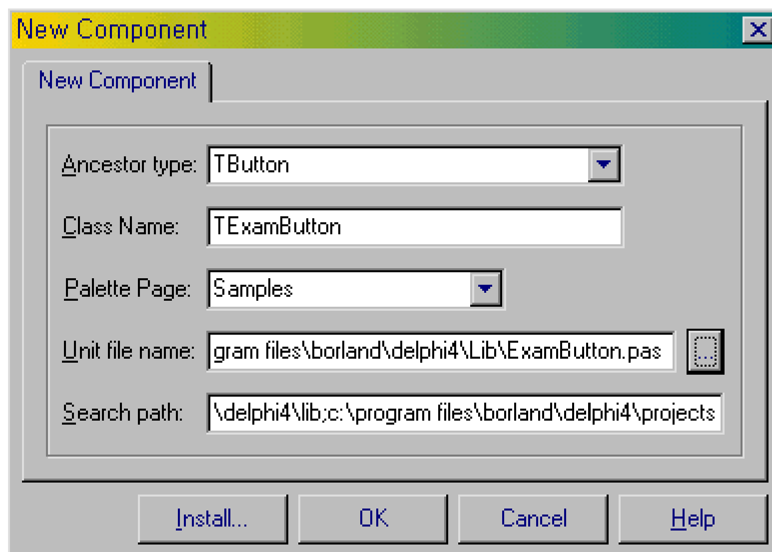
델파이는 가장 잘된 윈도우용 OOP 개발 환경이라고 말할 수 있다. 이러한 델파이의 가장 핵심 부분은 누가 뭐라 해도 델파이의 튼튼한 컴포넌트라고 말할 수 있다. 컴포넌트란 OOP 의 기본 개념을 충실하게 지원하는 델파이의 객체로, 이러한 컴포넌트를 개발하는 방법이야 말로, 델파이 개발자에게는 가장 중요한 기술이라고 말해도 과언이 아니다. 이번 장에서는 컴포넌트가 동작하는 방법을 이해하고 실제로 컴포넌트를 제작하는 방법에 대해서 알아 보기로 한다.

컴포넌트의 구조

컴포넌트는 크게 나누어 필드, 메소드, 프로퍼티라는 세가지의 파트로 나뉘어 있다. 필드는 객체 내부의 데이터 변수이며, 메소드는 객체에 속한 프로시저와 함수를 말하고, 프로퍼티란 객체에 속한 데이터와 코드에 접근하는 방법을 제공하는 엔티티이다.

간단한 컴포넌트의 제작

컴포넌트를 만드는 가장 간단한 방법은 Component|New Component... 메뉴를 선택해서 Component Expert 를 시작하는 것이다.



여기에서 앞에서와 같이 새로 만들 컴포넌트가 상속할 클래스의 이름과 컴포넌트 클래스의

이름, 그리고 컴포넌트가 위치할 컴포넌트 팔레트 페이지를 지정하면 뼈대가 되는 코드가 만들어진다.
만들어진 코드는 다음과 같다.

```
unit ExamButton;

interface

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

type
  TExamButton = class(TButton)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TExamButton]);
end;

end.
```

이렇게 뼈대가 만들어지면, 프로퍼티나 메소드 등을 추가해서 간단하게 확장된 새로운 컴포

넌트를 제작할 수 있다. TExamButton 컴포넌트를 클릭하면 간단한 메시지 박스를 표시할 수 있도록 수정해 보자. 이를 구현하려면 protected 섹션에서 Click 메소드를 다음과 같이 오버라이드하면 된다.

```
protected
  procedure Click; override;
```

그리고, 다음과 같이 구현한다.

```
procedure TExamButton.Click;
begin
  inherited Click;
  ShowMessage('클릭 !');
end;
```

inherited 키워드는 오버라이드한 과거의 메소드를 실행하도록 하는 키워드이다. 이렇게 만들어진 컴포넌트는 파일로 저장하고, Component|Install Component 메뉴를 통해서 실제로 인스톨 할 수 있게 된다.

참고: 오버라이드할 메소드 알아내기
컴포넌트를 제작할 때 특정 이벤트에서 다른 동작을 하도록 만들고 싶을 때가 있다. 이럴 때에는 보통 오브젝트 인스펙터에서 관찰할 수 있는 이벤트 이름에서 'On'을 뺀 이름의 메소드를 오버라이드 하면 된다. 앞의 예제에서는 OnClick 이벤트에 대해서 조작을 가하기 위해 'On'을 뺀 Click 메소드를 오버라이드 하였다.

상속(Inheritance)의 활용

상속이란 부모 클래스에 속해 있는 메소드와 프로퍼티를 재사용하거나 override 해서 사용할 수 있는 특성을 말하는 것이다. 이를 이용하면 부모 클래스의 기능을 유지한 채, 새로운 기능을 추가해서 향상된 클래스를 쉽게 만들어낼 수 있다.

- Protected 프로퍼티의 노출

델파이에는 많은 수의 TCustomXxxx 클래스가 있는데, 이들의 모든 프로퍼티는 protected 로 정의되어 있다. 이들은 컴포넌트 팔레트에는 나타나지 않지만 실제로 사용되는 컴포넌트 클래스의 기초가 되는 것들이다. 예를 들어, TCustomEdit 는 TCustimMaskEdit,

TCustomMemo, TDBLookupCombo, TEdit, TSpinEdit 컴포넌트의 공통된 부모 클래스이다. 또한, TCustomMaskEdit 컴포넌트는 TDBEdit, TMaskEdit 의 공통된 부모 클래스가 된다. 이렇게 기초가 되는 부모 TCustomXxxx 클래스를 상속 받아서 이들의 protected 프로퍼티, 이벤트를 오브젝트 인스펙터에 나타나게 하려면 이들을 published 섹션에 재선언만 해주면 된다. 예를 들어, TSpeedButton 클래스는 Align 프로퍼티가 없는데, 이 프로퍼티를 추가하려면 다음과 같이 Align 프로퍼티를 published 섹션에 재선언 해주는 컴포넌트를 만들면 된다.

```
unit AlignBtn;

inteface

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

type
  TAlignSpeedButton = class(TSpeedButton)
  published
    property Align;           //원래는 protected 섹션에 선언되어 있었음
  end;

procedure Register;

implementation
begin
  RegisterComponents('Samples', [TAlignSpeedButton]);
end;

end.
```

● 상속받은 프로퍼티 감추기

위의 경우와는 반대로, 상속받아 만든 컴포넌트의 프로퍼티나 이벤트를 사용자가 오브젝트 인스펙터를 통해 접근하지 못하도록 하고 싶을 때가 있다. 이럴 때에는 기존의 프로퍼티와 같은 이름의 프로퍼티를 재선언하고, 이를 read-only 로 설정하면 된다. 다음의 코드는 TPanel 컴포넌트에서 Left, Top, Height, Width 프로퍼티를 없앤 컴포넌트 이다.

```
TSnapPanel = class(TPanel)
  private
    FDummyProperty: Byte;            //프로퍼티를 숨기기 위해 사용되는 필드

  ... (중략)

  published
    property Height: Byte read FDummyProperty;
    property Left: Byte read FDummyProperty;
    property Top: Byte read FDummyProperty;
    property Width: Byte read FDummyProperty;
  end;
```

- 가상 메소드의 override

메소드를 override 하는 것이 클래스의 기능을 확장하는 가장 빠른 방법이다.
델파이로 컴포넌트를 다룰 때에는 각각의 컴포넌트가 publish 하는 다양한 이벤트에 익숙해져야 한다. 새로운 컴포넌트를 상속받을 때에 가장 흔히 하는 실수는 상속받은 컴포넌트의 생성자에서 이벤트 핸들러를 동적으로 생성하고, 여기에 값을 대입하는 것이다. 이렇게 하면, 사용자가 해당 이벤트에 해당되는 핸들러를 사용하게 되면, 생성자에서 만든 이벤트 핸들러는 절대로 호출되지 않는다. 그러므로, 해당 컴포넌트가 이벤트에 반응해야 한다면 이벤트 핸들러를 만들지 말고 처리해야할 이벤트와 대응되는 가상 메소드를 override 하도록 한다.
문제는 가상 메소드와 연결된 이벤트에 대한 정보를 찾기가 어렵다는 것인데, 이를 알기 위해서는 본래 소스 코드를 봐야 알 수 있겠지만, 일반적으로 VCL 에서는 이벤트 이름의 'On' 부분을 뺀 이름이 가상 메소드의 이름이다. 예를 들어, OnClick 이벤트에 해당하는 가상 메소드는 'Click'이다. 다음 컴포넌트는 버튼 컴포넌트의 Click 메소드를 override 해서 버튼을 클릭할 때마다 소리를 나게 하는 것이다.

```
unit SountBtn;

interface

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

```
type
  TSountButton = class(TButton)
  private
    FSoundFile: string;
  protected
  public
    constructor Create(AOwner: TComponent); override;
    procedure Click; override;
  published
    property SoundFile: string read FSoundFile write FSoundFile;
  end;

procedure Register;

implementation

uses
  MMSystem;

procedure TSoundButton.Click;
begin
  sndPlaySound(@FSoundFile, snd_Async or snd_NoDefault);
  inherited Click;
end;

constructor TSoundButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSoundFile := '*.wav';
end;

procedure Register;
begin
  RegisterComponents('Samples', [TSoundButton]);
end;
```

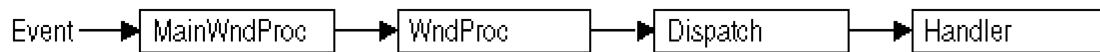
end.

윈도우 메시지 핸들러

전통적인 윈도우 프로그래밍에서 가장 중요한 것 중의 하나가 윈도우에 전달된 메시지를 처리하는 것이다. 델파이는 많은 부분을 처리해 준다. 그렇지만, 델파이가 처리하지 못하는 메시지는 개발자가 직접 처리해 주어야 한다.

- 메시지-처리 시스템의 이해

모든 델파이 클래스는 메시지를 처리하기 위한 기본적인 방법으로 메시지-처리 메소드나 메시지 핸들러와 같은 방법을 사용한다. 메시지 핸들러의 기본적인 아이디어는 메시지를 받은 클래스가 메시지 종류에 따라 특정 메소드 세트를 호출하는 것이다. 이때, 지정된 메소드가 없을 경우에는 디폴트 핸들러가 실행된다.
다음 그림은 메시지-디스패치(message-dispatch) 시스템에 대한 다이어그램이다.



VCL 은 메시지-디스패치 시스템을 모든 윈도우 메시지를 특정 클래스의 메소드 호출로 처리한다. 개발자가 할 일은 메시지-처리 메소드를 생성하는 것이다.

1. 윈도우 메시지

델파이에서의 윈도우 메시지는 몇 개의 필드로 이루어진 데이터 레코드이다. 그 중에서도 가장 중요한 것은 메시지를 확인할 수 있는 정수값이다. 윈도우는 많은 메시지를 정의하고 있다. 그리고, 이런 메시지 들은 Messages.pas 유닛에 선언되어 있으며 이들은 정수값으로 구별된다. 그리고, 메시지 레코드에는 2 개의 파라미터 필드와 1 개의 결과 필드를 포함하고 있다.
하나의 파라미터는 16 비트이며, 다른 하나는 32 비트 값이다. 이를 Win32 에서의 표현 방법을 이용하면 각각 wParam, lParam 에 해당하며, lParam 값의 경우 순서를 나누어 lParamHi, lParamLo 와 같이 접근하는 것이 가능하다.
처음 윈도우를 이용해서 프로그래밍을 할 때에는 API 를 사용할 때, 개발자가 각각의 파라미터의 내용을 찾아봐야 했다. 그런데, 지금은 메시지 크래킹(message cracking)이라고 하는 명명된 파라미터를 사용하기 때문에 이해하는 것이 간단해졌다. 예를 들어, WM_KEYDOWN 메시지의 파라미터는 nVirtKey, lKeyData 이다.

2. 메시지의 디스패칭 (Dispatching Message)

어플리케이션이 윈도우를 생성할 때, 윈도우 프로시저를 윈도우 커널에 등록하게 된다. 이 때 윈도우 프로시저는 윈도우로 넘어오는 메시지를 처리하는 루틴을 말한다. 전통적으로 윈도우 프로시저는 각각의 메시지에 대한 커다란 case 문으로 작성했었다. 여기에서 꼭 기억해 두어야 할 것은 윈도우 핸들을 가진 모든 윈도우가 메시지를 처리한다는 것이다. 즉, 새로운 윈도우를 생성할 때마다, 이들은 반드시 완전한 형태의 윈도우 프로시저를 각각 가지고 있어야 하는 것이다.
델파이는 메시지 디스패칭을 다음과 같은 방법으로 단순화 하였다.

- 각각의 컴포넌트는 완전한 메시지-디스패칭 시스템을 상속한다.
- 디스패치 시스템은 디폴트 처리가 된다. 그러므로, 개발자는 디폴트 처리와 다른 부분만 핸들러를 만들어 주면 된다.
- 메시지 처리를 할 때에도, 일부 내용만 수정하고 나머지 처리 부분을 상속해서 처리하면 된다.

3. 메시지 흐름의 추적

델파이는 어플리케이션에 있는 각각의 컴포넌트에 대한 윈도우 프로시저로 MainWndProc 라는 메소드를 등록한다. MainWndProc 메소드에는 예외 처리 블록을 포함하고 있으며, 메시지 구조체를 윈도우에서 WndProc 라는 가상 메소드로 넘겨주고, 예외 처리는 클래스의 HandleException 메소드를 호출하여 수행된다.
MainWndProc 메소드는 특별히 메시지를 처리하지는 않는다. 실제로, 처리하는 부분은 필요에 의해 메소드를 오버라이드할 수 있는 WndProc 에서 이루어진다.
WndProc 메소드는 메시지를 처리할 때 영향을 미칠 수 있는 특별한 조건 들을 검사해서, 원하지 않는 메시지를 처리할 수 있다. 예를 들어, 드래그를 하고 있을 때 컴포넌트는 키보드 이벤트를 무시한다. 그러므로, TWinControl 클래스의 WndProc 메소드는 드래그를 하지 않을 때에만 키보드 이벤트를 처리한다. 결국에는 WndProc 가 TObject 객체에서 상속받은 Dispatch 메소드를 호출하게 되며 여기에서 메시지를 처리할 메소드가 어떤 것인지를 결정하게 된다.
Dispatch 메소드는 메시지 구조체의 Msg 필드를 이용하여, 특정 메시지를 어떻게 디스패치할 것인지를 결정한다. 컴포넌트가 특정 메시지에 대한 핸들러를 정의한다면 Dispatch 는 그 메소드를 호출하며, 해당 메시지에 대한 메소드가 없으면 DefaultHandler 를 호출한다.

- 메시지 처리 방법의 변경

컴포넌트에 대한 메시지 처리 방법을 변경하기 전에, 실제 하려고 하는 것이 무엇인지를 분명히 해야 한다. 델파이는 대부분의 윈도우 메시지를 컴포넌트의 이벤트로 번역해서 처리하므로 많은 경우에는 직접 메시지를 처리하기 보다는 이벤트를 처리하는 것으로 해결이 된다. 메시지 처리 방법을 변경하기 위해서는 메시지를 처리하는 메소드를 오버라이드해야 한다.

1. 메시지 핸들러 메소드의 오버라이드

컴포넌트가 특정 메시지를 처리하는 방법을 변경하려면, 메시지를 처리하는 메소드를 오버라이드해야 한다. 컴포넌트에 메시지를 처리하는 메소드가 없는 경우라면, 새로운 메시지 처리 메소드를 선언해야 한다. 메시지 처리 메소드를 오버라이드하기 위해서는, 메소드가 오버라이드하고 있는 것과 같은 메시지 인덱스를 이용해서 새로운 메소드를 선언하면 된다. override 지시어를 사용하는 것이 아니라, 동일한 메시지 인덱스를 이용하여 message 지시어를 사용한다는 것에 주의한다.
예를 들어, WM_PAINT 메시지를 처리하는 메소드인 WMPaint 메소드를 다음과 같이 선언하여 사용할 수 있다.

```
type
  TMyComponent = class(...)
  ...
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

2. 메시지 파라미터의 활용

메시지 핸들러에 넘겨진 파라미터는 var 형이므로, 핸들러에서 파라미터의 값을 변경할 수 있다. Message 파라미터의 데이터 형은 처리할 메시지에 따라 다양하다. 그러므로, 이를 잘 알기 위해서는 윈도우 메시지에 대한 문서를 참고해야 한다. 만약, 과거 스타일로 WParam, LParam 등으로 메시지 파라미터를 표현한 경우에는 이를 TMessage 데이터 형으로 형전환 해서 사용한다.

3. 메시지 트래핑

어떤 경우에는 컴포넌트가 메시지를 무시하도록 하고 싶을 때가 있다. 이렇게 메시지를 트랩하도록 하려면 WndProc 메소드를 오버라이드한다. WndProc 메소드는 앞에서 설명했듯

이 Dispatch 메소드가 메시지 처리 메소드를 호출하기 전에 메시지를 처리할 수 있기 때문에, 여기에서 디스패치를 하기 전에 메시지를 거를 수 있다. TWinControl 에서 상속한 컨트롤의 WndProc 를 다음과 같이 오버라이드하여 사용할 수 있다.

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  inherited WndProc(Message);
end;
```

WndProc 메소드를 오버라이드 하면 메시지의 범위를 거를 수 있고, 메시지를 디스패치하지 않도록 할 수 있기 때문에 핸들러는 호출되지 않는다.
다음의 코드는 TControl 에 대한 WndProc 메소드의 일부이다.

```
procedure TControl.WndProc(var Message: TMessage);
begin
  …
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                  //드래그를 하고 있다면,
      DragMouseMsg(TWMMouse(Message))       //마우스 드래그에 해당하는 메시지 처리를 …
    else
      …                                              //아니면, 정상적인 처리를 한다.
    end;
…
end;
```

● 새로운 메시지 핸들러의 작성

델파이는 대부분의 윈도우 메시지에 대한 핸들러를 제공하기 때문에, 메시지 핸들러를 새로 작성할 필요가 있는 경우는 사용자 정의 메시지를 정의해서 여기에 대한 메시지 핸들러를 작성할 때가 많다.

1. 메시지의 정의

많은 수의 표준 컴포넌트 들이 내부적인 사용을 위해 메시지를 정의한다. 이렇게 메시지를 정의하는 이유로 가장 흔한 것이 표준 윈도우 메시지로 처리할 수 없는 정보나, 상태의 변경을 알리기 위한 것이다.

메시지 identifier 는 정수형으로, 윈도우는 특정 번호 이하만 사용하기 때문에 사용자 정의 메시지에 나름대로 번호를 부여해서 사용할 수 있다. WM_APP 상수는 사용자 정의 메시지의 시작 번호를 대표한다. 그러므로, 메시지를 정의할 때에는 WM_APP 를 바탕으로 하여 identifier 를 결정해서 사용한다.
그런데, 주의해야 할 것은 표준 윈도우 컨트롤 중에서도 사용자 정의 메시지 범위에 들어가는 메시지를 사용하는 경우가 있다는 점이다. 이런 컴포넌트에는 리스트 박스, 콤보 박스, 에디트 박스와 버튼 등이 있다. 이런 컴포넌트를 이용해서 새로운 메시지를 정의할 때에는 Messages.pas 유닛을 참고하여 컨트롤이 이미 사용하고 있는 윈도우 메시지와 겹치지 않도록 주의해야 한다.
사용자 정의 메시지는 다음과 같이 정의하면 된다.

```
const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;
```

메시지 레코드는 메시지 처리 메소드로 전송되는 파라미터의 데이터 형이다. 만약 메시지의 파라미터를 사용하지 않거나, 과거와 같이 wParam, lParam 을 사용한 파라미터를 사용할 경우에는 디폴트 메시지 레코드인 TMessage 를 사용하면 된다.
메시지 레코드 데이터 형을 선언하기 위해서는 다음과 같은 규칙을 따라야 한다.

- Msg 레코드의 첫번째 필드는 TMsgParam 데이터 형으로 선언한다.
- 그 다음의 2 바이트를 Word 파라미터로 설정하고 다음 2 바이트는 사용하지 않거나, 4 바이트를 LongInt 파라미터를 설정한다.
- 마지막으로 LongInt 형의 Resut 필드를 추가한다.

예를 들어, 모든 마우스 메시지를 처리하는 TWMMouse 메시지 레코드의 선언부를 여기에 소개한다. 가변형 레코드를 사용하여, 같은 파라미터를 2 개의 세트로 정의해서 사용한다.

```
type
  TWMMouse = record
    Msg: TMsgParam;                 //메시지 ID
    Keys: Word;                     //wParam 에 해당
    case Integer of                 //lParam 을 해석하는 방법이 2 가지 !
      0: (
         XPos: Integer;             //x, y 좌표로 접근
         YPos: Integer);
```

```
    1: (
      Pos: TPoint;              //위치
      Result: Longint);         //결과 필드
end;
```

2. 새로운 메시지-처리 메소드의 선언

새로운 메시지-처리 메소드는 컴포넌트가 표준 컴포넌트에 의해 처리되지 않는 윈도우 메시지를 처리하거나, 자신의 메시지를 정의해서 사용할 경우에 필요하다.
다음의 코드는 CM_CHANGECOLOR 라는 사용자 정의 메시지에 대한 메시지 핸들러를 선언한 것이다.

```
const
  CM_CHANGECOLOR = WM_APP + 400;

type
  TMyComponent = class(TControl)
  …
protected
  procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;
```

그러면, VCL 에서 TControl 컴포넌트에 OnClick 이벤트를 처리할 수 있도록 하는 부분의 소스 코드를 살펴 보자. 다소 복잡하지만 기본적인 방법은 동일하게 사용되고 있다.

```
TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;
    … (중략)
```

```
    procedure WMLButtonUp(var Message: TWMLButtonUp); message WM_LBUTTONUP;
    ... (중략)

  protected
    ... (중략)


    procedure Click; dynamic;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    ... (중략)

end;

... (중략)

procedure TControl.Click;
begin
  if Assigned(FOnClick) then FOnClick(Self);
end;

procedure TControl.WMLButtonUp(var Message: TWMLButtonUp);
begin
  inherited;
  if csCaptureMouse in ControlStyle then
    MouseCapture := False;
  if csClicked in ControlStyle then
  begin
    Exclude(FControlState, csClicked);
    if PtInRect(ClientRect, SmallPointToPoint(Message.Pos)) then
      Click;          //이벤트 핸들러를 호출하게 되는 가상 메소드를 호출한다.
  end;
  DoMouseUp(Message, mbLeft);
end
```

컴포넌트에서 그래픽 이용하기

델파이는 윈도우 GDI 를 여러가지 레벨로 캡슐화하고 있다. GDI 함수를 직접 호출할 때에

는 디바이스 컨텍스트에 대한 핸들을 사용하게 된다. 일단 그래픽 이미지를 그리고 나면, 반드시 디바이스 컨텍스트를 원래의 상태로 복귀시키고 이를 처리해야 한다.
10 장에서 이미 그래픽에 대한 내용을 다룬 바 있지만, 컴포넌트를 제작하기 위해서는 여기에 대해서 잘 알고 있어야 한다. 델파이는 컴포넌트에 Canvas 프로퍼티를 제공함으로써 복잡한 GDI 함수를 직접 호출해서 사용하는 것을 대신한다. 캔버스는 리소스를 관리하고, 선택하고, 사용하는 등의 모든 작업을 도맡아 하게 된다.
델파이를 사용할 때의 장점으로는 이밖에도 리소스를 캐쉬하기 때문에 나중에 다시 사용하거나, 반복적인 작업을 할 때 속도의 증진을 기대할 수 있다.

- 캔버스의 활용

Canvas 클래스는 윈도우 그래픽을 여러가지 레벨에서 캡슐화 한다. 즉, 도형을 그리거나 라인을 그리고, 텍스트를 그릴 수 있는 고수준 함수에서 부터, 윈도우 GDI 에 접근하는 저수준 함수까지 지원한다. 다음에 캔버스에 대한 내용을 정리하였다.

레 벨	작 업	메소드와 프로퍼티
High	라인과 도형 그리기	MoveTo, LineTo, Rectangle, Ellipse
	텍스트 디스플레이와 측정	TextOut, TextHeight, TextWidth, TextRect
	영역 채우기	FillRect, FloodFill
Intermediate	텍스트와 그래픽 정의	Pen, Brush, Font 프로퍼티
	픽셀 처리	Pixels 프로퍼티
	이미지 복사와 병합	Draw, StretchDraw, BrushCopy, CopyRect 메소드 CopyMode 프로퍼티
Low	GDI 함수 호출	Handle 프로퍼티

- Picture 객체 작업 하기

델파이는 캔버스에 직접 그림을 그리는 것 이외에, 비트맵이나 메타 파일, 아이콘 등과 같은 그래픽 이미지를 처리할 수 있는 Picture 객체를 제공한다.

1. Picture, graphic, canvas

델파이가 그래픽을 처리하는 클래스에는 3 가지가 있다. 이들을 구별하고 잘 사용하는 것이 중요하다.
앞에서도 간단히 설명했지만, 캔버스는 폼이나, 그래픽 컨트롤, 프린터 또는 비트맵의 표면에 그릴 수 있는 객체이다. 다시 말해서, 실제로 화폭에 그림을 그리는 화가를 연상할 때

화폭에 해당되는 것이 캔버스이다. 보통 캔버스는 독립적인 클래스로 사용하지 않고, 컨트롤의 프로퍼티로 제공된다.
그래픽은 파일이나 리소스의 형태로 접근할 수 있는 그래픽 이미지를 대표한다. 델파이는 TGraphic 클래스를 상속한 TBitmap, TIcon, TMetafile 등의 클래스를 제공한다. TGraphic 클래스는 여러가지 다른 종류의 그래픽에서 공통적으로 사용하는 표준 인터페이스를 정의하고 있다.
Picture 는 그래픽의 컨테이너로, 어떤 그래픽 클래스도 담을 수 있다. 그리고, 어플리케이션은 이들에게 똑같은 방법으로 접근할 수 있는 방법을 제공한다. 예를 들어, Image 컨트롤은 Picture 프로퍼티를 제공하는데 이를 이용하여 여러 종류의 그래픽 이미지를 표시할 수 있다.

2. 팔레트 다루기

256 색상의 비디오 모드처럼 팔레트에 기초한 디바이스를 사용할 때에 델파이는 자동으로 팔레트의 realization 을 지원한다. 대부분의 컨트롤은 팔레트가 필요없다. 그렇지만, 그래픽 이미지를 표시하는 컨트롤은 이미지를 제대로 표시하기 위해서는 윈도우와 스크린 디바이스 드라이버와 상호 작용해야 한다. 윈도우는 이런 작업을 팔레트의 realization 이라고 한다. 다시 말해, 팔레트를 realize 하는 것은 현재의 윈도우에 모든 팔레트를 사용하도록 하고, 배경 윈도우는 가능한 팔레트를 가장 가까운 색상과 맞도록 표시하는 작업이다. 그러므로, 활성화된 윈도우가 바뀔 경우에 윈도우는 팔레트를 realize 해야 한다.
컨트롤에 대한 팔레트를 지정하기 위해서는 컨트롤의 GetPalette 메소드를 오버라이드하여 팔레트의 핸들을 반환하도록 한다.
컨트롤이 GetPalette 메소드를 오버라이드하여 팔레트를 지정하면, 델파이는 자동으로 윈도우의 팔레트 메시지에 반응한다. 팔레트 메시지를 처리하는 메소드는 PaletteChanged 이다. PaletteChanged 메소드의 주 목적은 컨트롤의 팔레트를 realize 하는 것이다. 윈도우가 팔레트를 realization 할 때에는 활성화된 윈도우에 foreground 팔레트를 가지도록 하고, 다른 background 팔레트를 가지도록 한다. 델파이는 여기에서 추가적으로 윈도우 내에 있는 컨트롤 들의 탭 순서에 따라 팔레트를 realize 한다. 그러므로, 디폴트 핸들러를 오버라이드하여 팔레트를 관리할 필요가 있는 경우는 탭 순서에서 첫번째가 아닌 컨트롤에 foreground 팔레트를 적용하고자 할 경우이외에는 거의 없다.

- Off-screen 비트맵

복잡한 그래픽 이미지를 그릴 때 윈도우 프로그래밍에서 공통적인 테크닉은 off-screen 비트맵을 생성하고, 비트맵에 이미지를 그리고, 스크린에 비트맵을 복사하여 붙여넣는 작업이 있다. Off-screen 이미지를 사용하면, 스크린에 반복적인 그리기 작업을 할 때처럼 깜빡이

는 현상을 줄일 수 있다. 델파이의 비트맵 클래스 역시 off-screen 이미지로 작업할 수 있다.

- 변화에 반응하기

모든 그래픽 객체는 객체의 변화에 반응하는 이벤트를 가질 수 있다. 이런 이벤트를 이용하여 컴포넌트가 이미지를 다시 그릴 때 반응하도록 할 수 있는 것이다. 그래픽 객체의 변화를 반영하는 것은 그래픽 객체를 디자인-타임 인터페이스로 사용할 때에 더욱 중요하다. 그래픽 객체의 변화에 반응하려면, 클래스의 OnChange 이벤트에 메소드를 대입해야 한다. 예를 들어, TShape 컴포넌트는 펜, 브러쉬 등의 객체를 프로퍼티로 제공하는데, 컴포넌트의 constructor 에서 메소드를 OnChange 이벤트에 대입하여 컴포넌트가 펜이나 브러쉬가 변경되면 이미지를 refresh 하도록 한다.

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
…
implementation
…
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Width := 65;
  Height := 65;
  FPen := TPen.Create;
  FPen.OnChange := StyleChanged;                //OnChange 이벤트에 메소드 대입
  FBrush := TBrush.Create;
  FBrush.OnChange := StyleChanged;              //OnChange 이벤트에 메소드 대입
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                                 //컴포넌트를 다시 그린다.
end;
```

그래픽 컴포넌트의 제작

순수한 그래픽 컨트롤은 포커스를 가질 수 없기 때문에, 윈도우 핸들을 필요로 하지 않는다. 사용자 들은 컨트롤을 마우스로 조작할 수는 있지만, 키보드 인터페이스는 가질 수 없다. 여기에서는 Additional 페이지에 있는 TShape 와 거의 동일한 TSampleShape 라는 컴포넌트를 만들어 볼 것이다.

- 컴포넌트 시작하기

먼저 컴포넌트를 TGraphicControl 에서 상속 받도록 하고, 이름을 TSampleShape 라고 한다. 이 컴포넌트는 Samples 페이지에 등록하도록 한다. 이렇게 설정하면 다음과 같은 뼈대 코드가 만들어질 것이다.

```
unit Shapes;

interface

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

type
  TSampleShape = class(TGraphicControl)
  private
  protected
  public
  published
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleShape]);
```

```
end;

end.
```

먼저, TGraphicControl 에서 protected 섹션에 정의된 여러 프로퍼티를 published 섹션에 사용하도록 선언한다. 많은 프로퍼티들은 이미 TGraphicControl 클래스의 published 섹션에 선언되어 있으므로, 마우스 이벤트와 드래그-드롭을 지원하는 이벤트만 publish 하면 된다.

```
published
  property DragCursor;
  property DragMode;
  property OnDragDrop;
  property OnDragOver;
  property OnEndDrag;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
end;
```

- 그래픽 기능의 추가

1. Shape 프로퍼티의 추가

그래픽 컨트롤은 사용자 입력을 포함한 동적 조건을 반영하여 형태를 바꿀 수 있다. 일반적으로 그래픽 컨트롤의 형태는 이들의 조합이라고 할 수 있다. 예를 들어, TGauge 컨트롤의 경우 형태와 방향을 결정하고, 숫자와 그림을 보여 줄 것인지 여부 등을 프로퍼티에서 결정할 수 있다.
TSampleShape 컨트롤에도 어떤 것을 그릴 것인지 여부를 결정할 수 있도록 Shape 라는 프로퍼티를 제공하도록 한다. 이를 위해 먼저, 프로퍼티로 사용할 열거형(enumerated type)을 정의하도록 하자.

```
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
```

이제 다음과 같이 프로퍼티를 선언한다.

```
  private
    FShape: TSampleShapeType;
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

그리고, SetShape 메소드를 다음과 같이 구현한다.

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then
  begin
    FShape := Value;
    Invalidate;
  end;
end;
```

2. 디폴트 프로퍼티의 변경

그래픽 컨트롤의 디폴트 크기는 매우 작다. 그러므로, 이를 수정하려면 다음과 같이 프로퍼티를 선언할 때 default 값을 변경하고 constructor 에서 수정할 필요가 있다.

```
  public
    constructor Create(AOwner: TComponent); override
  end;

  published
    property Height default 65;
    property Width default 65;
  end;

constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
```

```
  Width := 65;
  Height := 65;
end;
```

3. 펜과 브러쉬의 publish

펜과 브러쉬를 변경할 수 있게 하려면, 내부적으로 사용할 객체를 private 섹션에 필드로 선언해야 한다.

```
  private
    FPen: TPen;
    FBrush: TBrush;
    …
  end;
```

이들을 프로퍼티로 접근할 수 있게 하기 위해 Set 메소드와 프로퍼티를 published 섹션에 추가한다.

```
  private
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

그리고, SetBrush 와 SetPen 메소드를 다음과 같이 구현한다.

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);
```

```
end;
```

또한, 이들을 constructor 에서 생성해야 런타임에서 사용할 수 있다. 또한, destructor 에서는 이 객체 들을 해제해야 한다.

```
public
  destructor Destroy; override;
end;

constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Width := 65;
  Height := 65;
  FPen := TPen.Create;
  FBrush := TBrush.Create;
end;

destructor TSampleShape.Destroy;
begin
  FPen.Free;
  FBrush.Free;
  inherited Destroy;
end;
```

마지막으로 펜과 브러쉬가 변경 되었을 때, 그려진 도형을 다시 그릴 필요가 있다. 펜과 브러쉬 객체는 모두 OnChange 이벤트를 가지고 있으므로, 이들 이벤트를 설정해서 사용하면 된다.

```
  published
    procedure StyleChanged(Sender: TObject);
  end;

...

constructor TSampleShape.Create(AOwner: TComponent);
```

```
begin
  inherited Create(AOwner);
  Width := 65;
  Height := 65;
  FPen := TPen.Create;
  FPen.OnChange := StyleChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := StyleChanged;
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate;
end;
```

- 컴포넌트 이미지 그리기

마지막으로 스크린에 그리는 부분을 구현할 차례이다. 그리는 부분은 TGraphicControl 클래스의 추상 메소드인 Paint 를 오버라이드해야 한다. 여기에서 선택된 펜과 브러쉬를 가지고 선택된 도형을 그린다.

```
  protected
    procedure Paint; override;
  …
  end;
```

Paint 메소드는 도형의 형태에 따라 좌표를 사용하는 방법을 달리해서 구현해야 한다. Paint 메소드는 다음과 같이 구현하도록 한다.

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
  begin
    Pen := FPen;
```

```
      Brush := FBrush;
      W := Width;
      H := Height;
      if W < H then S := W else S := H;
      case FShape of
        sstRectangle, sstRoundRect, sstEllipse:
          begin
            X := 0;
            Y := 0;
          end;
        sstSquare, sstRoundSquare, sstCircle:
          begin
            X := (W - S) div 2;
            Y := (H - S) div 2;
            W := S;
            H := S;
          end;
      end;
      case FShape of
        sstRectangle, sstSquare:
          Rectangle(X, Y, X + W, Y + H);
        sstRoundRect, sstRoundSquare:
          RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);
        sstCircle, sstEllipse:
          Ellipse(X, Y, X + W, Y + H);
      end;
    end;
end;
```

객체 결합 (Object Composition)

객체 결합이란 여러 개의 컴포넌트를 하나로 묶어서 새로운 컴포넌트를 만드는 것을 의미한다. 이 테크닉을 이용하면 각각의 요소가 되는 컴포넌트의 이벤트와 프로퍼티에 사용자가 함부로 접근하지 못하도록 보호할 수 있다. 비주얼 컴포넌트의 결합에는 컨테이너 컴포넌트(TPanel, TCustomPanel, TWinControl) 등이 요구되며, 이들은 서브-컴포넌트의 부모 윈도우가 된다.

이 테크닉은 복잡한 컴포넌트를 그룹화하거나 컴포넌트 간의 유기적인 관계가 필요한 경우에 유용하게 사용될 수 있다.
다음의 예제 컴포넌트는 OK, Cancel 의 2 개의 버튼을 가지고 있으며, 각각의 컴포넌트에 대한 OnClick, Caption 프로퍼티를 제공한다. 참고로, 각각의 버튼이 Caption 프로퍼티를 가지므로 앞에서 설명한 프로퍼티 숨기기를 이용하여 OKCancel 버튼 자체의 Caption 프로퍼티는 없애 버린다 (FDummyProperty 를 이용).

```
unit OKCancel;

interface

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

type
  TOKCancelButton = class(TWinControl)
    //여기에 서브-컴포넌트들을 선언한다.
    OKButton: TBitBtn;
    CancelButton: TBitBtn;
  private
    FOnClick_OKButton: TNotifyEvent;
    FOnClick_CancelButton: TNotifyEvent;
    FDummyProperty: string;
    procedure SetCaption_OKButton(Value: TCaption);
    function GetCaption_OKButton: TCaption;
    procedure SetCaption_CancelButton(Value: TCaption);
    function GetCaption_CancelButton: TCaption;
  protected
    procedure Click_OKButton(Sender: TObject); virtual;
    procedure Click_CancelButton(Sender: TObject); virtual;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property Caption_OKButton: TCaption read GetCaption_OKButton
      write SetCaption_OKButton;
    property Caption_CancelButton: TCaption read GetCaption_CancelButton
```

```
      write SetCaption_CancelButton;
    property OnClick_OKButton: TNotifyEvent read FOnClick_OKButton
      write FOnClick_OKButton;
    property OnClick_CancelButton: TNotifyEvent read FOnClick_CancelButton
      write FOnClick_CancelButton;
    property Caption: string read FDummyProperty;
  end;

procedure Register;
```

참고로, 여기까지 선언부를 작성하고 Ctrl+Shift+C 키를 누르면 다음과 같은 구현 부분에 대한 뼈대 코드가 모두 자동으로 만들어진다. 이 기능이 바로 클래스 완료(Class Completion)이다. 컴포넌트를 제작하는 사람 들에게는 허드렛일을 많이 줄여줄 수 있는 기능이다.
어쨌든 구현부분은 다음과 같이 구현한다.

```
implementation

procedure TOKCancelButton.SetCaption_OKButton(Value: TCaption);
begin
  OKButton.Caption := Value;
end;

function TOKCancelButton.GetCaption_OKButton: TCaption;
begin
  Result := OKButton.Caption;
end;

procedure TOKCancelButton.SetCaption_CancelButton(Value: TCaption);
begin
  CancelButton.Caption := Value;
end;

function TOKCancelButton.GetCaption_CancelButton: TCaption;
begin
  Result := CancelButton.Caption;
```

```
end;

procedure TOKCancelButton.Click_OKButton(Sender: TObject);
begin
  if Assigned(FOnClick_OKButton) then FOnClick_OKButton(Self);
end;

procedure TOKCancelButton.Click_CancelButton(Sender: TObject);
begin
  if Assigned(FOnClick_CancelButton) then FOnClick_CancelButton(Self);
end;

constructor TOKCancelButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Width := 75;
  Height := 60;
  OKButton := TBitBtn.Create(Self);
  with OKButton do
  begin
    Parent := Self;
    Kind := bkOK;
    SetBounds(0, 0, 75, 30);
    TabOrder := 0;
    OnClick := Click_OKButton;
  end;
  CancelButton := TBitBtn.Create(Self);
  with CancelButton do
  begin
    Parent := Self;
    Kind := bKCancel;
    SetBounds(0, 31, 75, 30);
    TabOrder := 1;
    OnClick := Click_CancelButton;
  end;
  FDummyProperty := ‘’;
```

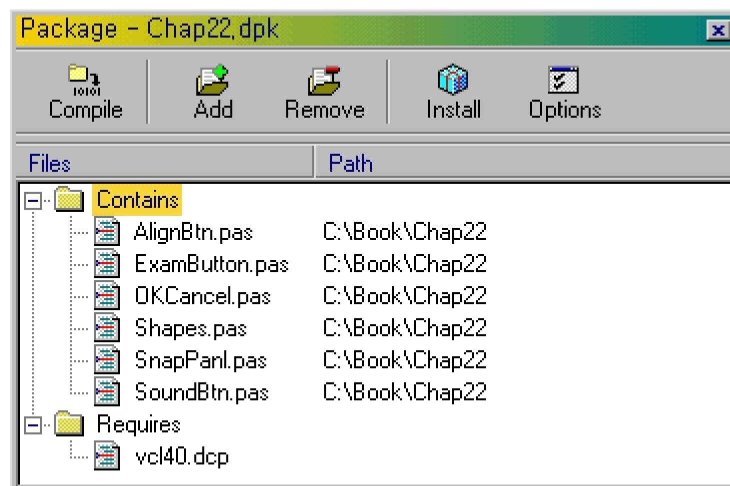
```
end;

procedure Register;
begin
  RegisterComponents('Samples', [TOKCancelButton]);
end;

end.
```

이와 같이 컴포넌트를 제작할 때, 내부에서 제어할 수 있는 컴포넌트를 활용하면 훨씬 강력하고 편리한 활용이 가능하다. 이 밖에도 private 섹션에 FComponetName 과 같은 형태로 내부에서 생성하여 관리할 컴포넌트를 선언해서, 필드 변수로 활용하고 프로퍼티를 선언해서 사용하는 방법이 있다.

패키지 제작과 컴포넌트 추가

비록 기본적인 예제 컴포넌트이긴 했지만, 이번 장에서 제작한 컴포넌트는 모두 6 개이다. 이들을 하나의 패키지로 묶어 보도록 하자. 먼저, File|New 메뉴를 선택한 후 객체 저장소에서 Package 를 더블 클릭하여 새로운 패키지를 하나 생성한다. 그리고, File|Save As 명령을 선택하여 적당한 디렉토리에 .DPK 파일을 저장한다. 여기서는 chap22.dpk 라는 이름으로 저장하도록 한다.

패키지 에디터 화면에서 Add 버튼을 클릭하고 다음과 같이 앞에서 제작한 6 개의 컴포넌트를 차례대로 추가한다.



패키지 파일을 저장하고, Install 버튼을 누르면 패키지가 설치될 것이다. 컴파일이 끝나면

컴포넌트 팔레트의 Samples 페이지에는 6 개의 컴포넌트가 추가될 것이다. 그리고, 이 명령에 의해 .BPL 파일이 생성되는데 이 파일을 따로 배포할 수도 있다.
어떤가 ? 컴포넌트를 만들고, 이들을 자신의 패키지로 엮어서 배포하는 일이 생각보다 너무나 쉽다고 생각될 것이다. 델파이 4 에서는 이와 같이 컴포넌트를 개발하고, 이를 관리하는 강력한 수단을 제공하고 있다.

정 리 (Summary)

이번 장에서는 가장 기본적인 컴포넌트를 만드는 방법과 이들을 이용하여 패키지로 묶는 방법에 대해서 알아보았다. 컴포넌트는 델파이의 핵심이라고 말할 수 있는 부분이다.
다음 장에서는 기본적인 컴포넌트에 기능을 추가하는 방법과 데이터 인식 컨트롤을 제작하는 방법과 같이 보다 고급스러운 컴포넌트 제작 기법에 대해서 알아볼 것이다.