CHAPTER _6

클래스와 객체

6장은 델파이 언어의 가장 OOP적인 요소인 클래스와 객체에 대해 살펴봅니다. 상속과 가시성, 오버로드, 생성자/소멸자, 속성, 중첩된 타입 등의 클래스의 특성들과 예외에 대해서도 알아봅니다.

■클래스 타입 ■필드 ■메소드 ■속성 ■중첩된 타입
■클래스 참조 ■예외(Exception)

클래스 또는 클래스 타입은 필드, 메소드, 그리고 속성으로 구성된 구조를 정의합니다. 클래스 타입의 인스턴스를 객체라고 합니다. 클래스의 필드, 메소드, 속성을 클래스의 구성요소 또는 멤버라고 합니다.

- 필드(field)는 본질적으로 객체의 일부인 변수입니다. 레코드의 필드와 마찬가지로 클래스 필드는 클래스의 각 인스턴스에 있는 데이터 항목을 나타냅니다.
- 메소드(method)는 클래스와 연결된 프로시저나 함수입니다. 대부분의 메소드는 클래스의 인스턴스인 객체에 대해 동작합니다. 일부 메소드(클래스 메소드)는 클래스 타입 자체에 대해 작동합니다.
- 속성(property)은 객체와 연결된 데이터(보통 필드에 저장)에 대한 인터페이스입니다. 속성에는 데이터를 읽고 수정하는 방법을 결정하는 액세스 지정자가 있습니다. 객체 외부에 있는 프로그램의 다른 부분에서 속성은 여러 관점에서 필드처럼 나타납니다.

객체는 해당 클래스 타입에 의해 구조가 결정되는 동적으로 할당된 메모리 블럭입니다. 각 객체에는 클래스에서 정의된 모든 필드의 고유한 복사본이 있지만 클래스의 모든 인스턴스

는 같은 메소드를 공유합니다. 객체는 생성자(constructor)와 소멸자(destructor)라는 특수 메소드에 의해 생성 및 소멸됩니다.
클래스 타입의 변수는 실제로는 객체를 참조하는 포인터입니다. 따라서 둘 이상의 변수들이 같은 객체를 참조할 수 있습니다. 다른 포인터와 마찬가지로 클래스 타입 변수는 nil 값을 가질 수 있습니다. 하지만 가리키는 객체에 액세스하기 위해 클래스 타입 변수를 명시적으로 역참조할 필요는 없습니다. 예를 들어, SomeObject.Size := 100은 SomeObject가 참조하는 객체의 Size 속성에 100이라는 값을 지정합니다. 그러나 이것을 SomeObject^.Size := 100 처럼 쓰지는 않습니다.

클래스 타입

클래스 타입은 인스턴스화하기 전에 선언하고 이름을 지정해야 합니다. (변수 선언 내에서 클래스 타입을 정의할 수 없습니다.) 프로그램이나 유닛의 가장 외부 유효 범위(scope)에서만 클래스를 선언할 수 있으며, 프로시저나 함수 선언에서는 안됩니다.
클래스 타입 선언은 다음과 같은 형태를 가집니다.

```
type className = class [abstract | sealed] (ancestorClass)
  memberList
end;
```

여기서 *className*은 유효한 식별자이고, sealed나 abstract는 옵션이며, (*ancestorClass*)도 옵션이며, *memberList*는 클래스의 멤버들(필드, 메소드, 속성)을 선언합니다. (*ancestorClass*)를 생략하면 새 클래스는 이미 정의된 TObject 클래스에서 직접 상속됩니다. (*ancestorClass*)가 있고 *memberList*가 비어 있는 경우 end를 생략할 수 있습니다. 클래스 타입 선언에 클래스에 의해 구현하는 인터페이스들을 포함할 수도 있습니다. 11장의 "인터페이스의 구현"을 참조하십시오.
클래스가 sealed로 표시되면 상속을 통해 확장할 수 없습니다. 클래스가 abstract로 표시되면 Create 생성자로 직접 인스턴스화할 수 없습니다. abstract virtual 메소드를 전혀 포함하고 있지 않은 경우라도 전체 클래스로 abstract로 선언할 수 있습니다. 클래스는 abstract이면서 동시에 sealed일 수는 없습니다.
클래스 선언에서 메소드는 바디 없이 함수나 프로시저의 헤더로 나타납니다. 각 메소드에 대한 정의적 선언(defining declaration)은 프로그램 내의 어디에나 위치시킬 수 있습니다.

예를 들어, Classes 유닛에 있는 TMemoryStream 클래스 선언은 다음과 같습니다.

```
type
  TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
  end;
```

TMemoryStream은 Classes 유닛에 있는 TCustomMemoryStream의 자손으로, 대부분의 멤버를 상속합니다. 하지만 소멸자 메소드인 Destroy를 비롯한 몇몇 메소드와 속성들을 재정의하거나 새로 정의합니다. 생성자인 Create는 변경되지 않고 TObject에서 상속하고 재선언되지 않습니다. 각 멤버는 private, protected 또는 public으로 선언되어 있습니다. (이 클래스에는 published 멤버는 없습니다.) 이들 용어에 대한 설명은 아래에서 할 것입니다. 위와 같이 선언했다면, 다음과 같이 TMemoryStream의 인스턴스를 만들 수 있습니다.

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

상속 및 유효 범위(scope)

클래스를 선언할 때 그 클래스의 직접 조상(ancestor)을 지정할 수 있습니다. 예를 들면,

```
type TSomeControl = class(TControl);
```

위 문법은 TControl의 자손으로 TSomeControl이라는 클래스를 선언합니다. 클래스 타입은 해당 직접 조상으로부터 자동으로 모든 멤버를 상속받습니다. 각 클래스는 새 멤버를 선언하거나 상속된 멤버를 재정의할 수 있지만, 조상에서 정의된 멤버를 제거할 수는 없습니다. 따라서 TSomeControl에는 TControl과 TControl의 각 조상들에서 정의된 멤버가 모두

포함되어 있습니다.
멤버의 식별자 유효 범위(scope)는 멤버가 선언된 위치에서 시작하여 클래스 선언 끝까지 계속되며 클래스의 모든 자손과 클래스 및 클래스 자손에서 정의된 모든 메소드 블럭까지 확장됩니다.

■ TObject 및 TClass

System 유닛에서 선언된 TObject 클래스는 다른 모든 클래스의 최고 조상입니다. TObject는 기본 생성자 및 소멸자를 비롯한 최소한의 메소드만을 정의합니다. TObject와 함께, System 유닛에는 클래스 참조 타입인 TClass도 선언되어 있습니다.

```
TClass = class of TObject;
```

TObject에 대한 자세한 내용은 온라인 헬프를 참조하십시오. 클래스 참조 타입에 대한 자세한 내용은 "클래스 참조" 절을 참조하십시오.
클래스 타입 선언에서 조상을 지정하지 않으면 클래스는 TObject를 직접 상속받습니다. 따라서,

```
type TMyClass = class
  ...
end;
```

위 선언은 아래의 선언과 같습니다.

```
type TMyClass = class(TObject)
  ...
end;
```

가독성을 위해 후자 형태를 사용하는 것이 좋습니다.

■ 클래스 타입의 호환성

클래스 타입은 자신의 조상 클래스 타입과 대입 호환 가능합니다. 따라서 클래스 타입의 변수는 모든 자손 타입의 인스턴스를 참조할 수 있습니다. 예를 들어, 다음과 같이 선언했다면,

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure)
  TSquare = class(TRectangle);
var
  Fig: TFigure;
```

Fig 변수에는 TFigure, TRectangle, TSquare 타입의 값을 대입할 수 있습니다.

■ object 타입

델파이 컴파일러는 클래스 타입을 대신할 수 있는 object 타입을 사용할 수 있습니다. 문법은 아래와 같습니다.

```
type objectTypeName = object (ancestorObjectType)
  memberList
end;
```

여기서 *objectTypeName*은 유효한 식별자이고, (*ancestorObjectType*)은 옵션이며, memberList는 필드, 메소드, 속성들을 선언합니다. (*ancestorObjectType*)을 생략하면 새 타입은 조상이 없게 됩니다. object타입은 published 멤버를 가질 수 없습니다.
object 타입은 TObject의 자손이 아니므로, 기본으로 제공되는 생성자, 소멸자, 기타 메소드가 없습니다. New 프로시저를 사용하여 object 타입의 인스턴스를 만들 수 있고 만든 인스턴스를 Dispose 프로시저로 소멸시키거나, 레코드에서처럼 object 타입 변수를 레코드처럼 선언할 수 있습니다. object 타입은 하위 호환성을 위해서만 지원되므로 사용하지 않는 것이 좋습니다.

클래스 멤버의 가시성

클래스의 모든 멤버는 가시성(visibility)이라는 성질을 가지는데, 이는 예약어 private, protected, public, published 또는 automated 중의 하나입니다. 예를 들면,

```
published property Color: TColor read GetColor write SetColor;
```

위 코드는 Color라는 published 속성을 선언합니다. 가시성은 멤버가 액세스될 수 있는 범위 및 방법을 결정하는 것으로, private는 최소한의 액세스, protected는 중간 수준으로 액세스, public, published, automated는 최대한의 액세스를 지정합니다.

멤버의 선언이 자신만의 가시성 지정 없이 나타나는 경우 그 멤버는 바로 앞 멤버와 같은 가시성을 가집니다. 클래스가 {$M+} 상태에서 컴파일되거나 {$M+} 상태에서 컴파일된 클래스로부터 파생된 경우, 클래스 선언의 맨 처음에 나타나는 가시성이 지정되지 않은 멤버는 기본적으로 published를 갖게 되고, {$M+} 없이 컴파일된 경우에는 public을 갖게 됩니다. 가독성을 위해 가시성에 따라 클래스 선언을 정리해두는 것이 좋습니다. 모든 private 멤버를 모아 두고, 그 뒤에는 protected 멤버를 모아 두고, 이런 식입니다. 그러면 각 가시성 예약어는 한 번만 나타나며 선언의 새 "섹션" 시작을 표시합니다. 일반적인 클래스 선언은 다음과 같습니다.

```
type
  TMyClass = class(TControl)
  private
    ... { private declarations here}
  protected
    ... { protected declarations here }
  public
    ... { public declarations here }
  published
    ... { published declarations here }
  end;
```

자손 클래스를 재선언하면 자손 클래스 멤버의 가시성을 높일 수 있지만, 가시성을 낮출 수는 없습니다. 예를 들어, protected 속성은 자손 클래스에서 public으로 재선언될 수 있지만 private이 될 수는 없습니다. 또한, published 멤버는 자손 클래스에서 public이 될 수 없습니다. 자세한 내용은 6장의 "속성 오버라이드 및 재선언"을 참조하십시오.

■ private, protected 및 public 멤버

private 멤버는 해당 클래스가 선언된 유닛이나 프로그램 외부에서 보이지 않습니다. 즉, 다른 모듈에서 private 메소드를 호출할 수 없고 private 필드나 속성을 다른 모듈에서 읽거나 쓸 수 없습니다. 관계가 있는 클래스의 선언을 같은 모듈에 두면, private 멤버에 대한 접근성을 넓히지 않고도 private 멤버 간에 서로 액세스할 수 있도록 할 수 있습니다.

protected 멤버는 해당 클래스가 선언된 모듈과 해당 클래스의 모든 자손 클래스에서 볼 수 있으며, 자손 클래스가 선언된 모듈에 제한을 받지 않습니다. 즉, protected 멤버가 선언된 클래스로부터 파생된 클래스에 속한 모든 메소드의 구현 바디에서 protected 메소드를 호출할 수 있고 또 protected 필드나 속성을 읽거나 쓸 수 있습니다. 일반적으로 파생 클래스의

구현에서만 사용할 수 있도록 하려는 멤버들을 protected로 선언합니다.
public 멤버는 해당 클래스를 참조할 수 있는 모든 곳에서 보여집니다.

■ strict 가시성 지정자

가시성 지정자 private 및 protected에 더하여, 델파이 컴파일러는 더 강하게 액세스를 제한할 수 있는 추가 가시성 설정들을 지원합니다. 이 설정들은 strict private와 strict protected 가시성입니다.

strict private가시성으로 지정된 클래스 멤버는 선언된 클래스 내에서만 액세스 가능합니다. 이 가시성으로 지정된 멤버들은 같은 유닛 내의 프로시저나 함수에게도 보여지지 않습니다. strict protected 가시성으로 지정된 클래스 멤버는 선언된 곳과는 무관하게 자신이 선언된 클래스 안에서, 그리고 모든 파생 클래스 안에서 보여집니다. 나아가서, 인스턴스 멤버들(class나 class var 키워드 없이 선언된 멤버들)이 strict private나 strict protected로 선언되면 자신이 속한 클래스의 인스턴스 바깥에서는 액세스가 불가능합니다. 한 클래스의 인스턴스는 동일 클래스의 다른 인스턴스의 strict private나 strict protected 인스턴스 멤버들을 액세스할 수 없습니다.

델파이의 전통적인 private 가시성 지정자 는 CLR의 assembly 가시성에 해당합니다. 델파이의 protected 가시성 지정자 는 CLR의 assembly 혹은 family 가시성에 해당합니다.

Note

'strict' 라는 단어는 클래스 선언의 문맥 내에서 지시어로 취급됩니다. 클래스 선언 안에서 strict라는 이름으로 멤버를 선언할 수 없지만, 클래스 선언의 바깥에서는 가능합니다.

■ published 멤버

published 멤버는 public 멤버와 같은 가시성을 가집니다. 차이점은, published 멤버에 대해서는 런타임 타입 정보(RTTI)가 생성된다는 것입니다. RTTI를 사용하여 응용 프로그램에서 객체의 필드 및 속성을 동적으로 조회하고 해당 메소드를 찾을 수 있습니다. RTTI는 폼 파일을 저장하고 로드할 때 속성의 값을 액세스하고, Object Inspector에서 속성을 표시하며, 이벤트 핸들러라는 특정 메소드를 이벤트라는 특정 속성과 연결할 수 있습니다.

published 속성은 특정 데이터 타입에 제한됩니다. 서수(ordinal), 문자열, 클래스, 인터페이스, variant, 메소드 포인터 타입은 published로 선언될 수 있습니다. 집합 타입의 경우에도 그 시작 및 마지막의 서수 값이 0과 31 사이라면 가능합니다. (다시 말해, 집합 타입은 바이트, 워드 또는 더블 워드에 맞아야 합니다.) Real48을 제외한 실수 타입들도 published로 선언될 수 있습니다. 배열 타입의 속성(아래에서 설명할 배열 속성과 다름)은 published로 선언될 수 없습니다.

일부 속성들은 published로 선언할 수 있더라도 스트리밍 시스템에서 완벽하게 지원되지 않습니다. 이러한 속성에는 레코드 타입의 속성, published 가능한 모든 배열 속성("속성" 절

의 "배열 속성" 참조), 익명 값을 포함하는 열거 타입(4장의 "순서값이 지정된 열거 타입" 참조)의 속성이 포함됩니다. 이러한 종류의 속성을 published로 선언하는 경우 Object Inspector에서 해당 속성이 제대로 표시되지 않거나 객체가 디스크에 스트리밍될 때 속성 값이 유지되지 않습니다.
모든 메소드는 published가 될 수 있지만 클래스는 같은 이름으로 오버로드된 두 개 이상의 메소드를 published로 만들 수 없습니다. 필드는 클래스나 인터페이스 타입인 경우에만 published가 될 수 있습니다.
클래스는 {$M+} 상태에서 컴파일되거나 {$M+} 상태에서 컴파일된 클래스의 자손이 아니면 published 멤버를 가질 수 없습니다. published 멤버가 있는 대부분의 클래스는 {$M+} 상태에서 컴파일된 TPersistent에서 파생되므로 $M 지시어를 사용할 필요는 거의 없습니다.

■ automated 멤버

automated 멤버는 public 멤버와 같은 가시성을 가집니다. 차이점은, automated 멤버의 경우 오토메이션 타입 정보(오토메이션 서버에 필요)가 생성된다는 것입니다. automated 예약어는 하위 호환성을 위해 유지됩니다. ComObj 유닛의 TAutoObject 클래스는 automated를 사용하지 않습니다.
automated로 선언된 메소드와 속성에는 다음과 같은 제한 사항이 따릅니다.

- 모든 속성, 배열 속성 파라미터, 메소드 파라미터, 함수 결과의 타입은 automated 가능해야 합니다. automated 가능 타입은 Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool, 그리고 모든 인터페이스 타입들입니다.
- 메소드 선언에서는 기본인 register 호출 규칙을 사용해야 합니다. 메소드 선언은 가상(virtual)일 수 있지만 동적(dynamic)일 수는 없습니다.
- 속성 선언은 액세스 지정자(read 및 write)를 포함할 수 있지만 다른 지정자(index, stored, default 및 nodefault)는 허용되지 않습니다. 액세스 지정자 는 기본인 register 호출 규칙을 사용하는 메소드 식별자를 써야 하며, 필드 식별자는 허용되지 않습니다.
- 속성 선언에서는 타입을 지정해야 합니다. 속성 오버라이드는 허용되지 않습니다.

automated 메소드나 속성의 선언에는 dispid 지시어를 포함할 수 있습니다. dispid 지시어에 이미 사용된 ID를 지정하면 에러가 발생합니다.
dispid 지시어는 멤버에 대한 오토메이션 디스패치 ID를 지정하는 정수 상수의 앞에 있어야

합니다. 그렇지 않으면 컴파일러는 클래스 및 해당 조상의 메소드나 속성으로 사용된 최대 디스패치 ID보다 큰 디스패치 ID를 멤버에게 지정합니다. 오토메이션에 대한 자세한 내용은 11장의 "오토메이션 객체"를 참조하십시오.

forward 선언과 상호 종속 클래스

클래스 타입 선언이 class와 세미콜론으로 끝나는 경우, 즉,

```
type className = class;
```

이 문법처럼 class 다음에 조상이나 클래스 멤버가 없는 경우 forward 선언이라고 합니다. forward 선언은 동일한 타입 선언 섹션 내에서 같은 클래스를 정의적 선언(defining declaration)하여 완료되어야 합니다. 즉, forward 선언과 정의적 선언 사이는 다른 타입 선언들 외에는 올 수 없습니다.

forward 선언을 이용하면 상호 종속적인 클래스들을 선언할 수 있습니다. 예를 들면, 다음과 같습니다.

```
type
  TFigure = class; // forward 선언
  TDrawing = class
    Figure: TFigure;
    ...
  end;
  TFigure = class // 정의적 선언
    Drawing: TDrawing;
    ...
  end;
```

클래스 멤버가 없는 TObject의 자손 타입의 완전한 선언과 forward 선언을 혼동하지 마십시오.

```
type
  TFirstClass = class; // 이것은 forward 선언입니다
  TSecondClass = class // 이것은 완전한 class 선언입니다
  end;
  TThirdClass = class(TObject); // 이것은 완전한 class 선언입니다
```

필드

필드는 객체에 속하는 변수와 같습니다. 필드는 클래스 타입을 비롯한 어떤 타입도 될 수 있습니다. (즉, 필드는 객체 참조를 가질 수 있습니다.) 필드는 일반적으로 private입니다.
클래스의 필드 멤버를 정의하려면, 단순히 변수를 선언하는 것처럼 필드를 선언하면 됩니다. 예를 들어, 다음의 선언은 TObject에서 상속 받은 이외의 유일한 멤버가 Int라는 정수 필드인 TNumber라는 클래스를 만듭니다.

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

var 예약어는 옵션입니다. 하지만, var 예약어가 사용되지 않으면 모든 필드 선언들은 모든 속성 및 메소드 선언보다 먼저 위치해야 합니다. 모든 속성 및 메소드 선언의 뒤에서 var를 사용하여 추가로 필드 선언을 할 수 있습니다.
필드는 정적으로 바인드됩니다. 즉, 이 필드에 대한 참조는 컴파일 시에 결정됩니다. 이것의 의미를 알아보기 위해 다음 코드를 살펴봅시다.

```
type
  TAncestor = class
    Value: Integer;
end;

  TDescendant = class(TAncestor)
    Value: string; // 상속된 Value 필드를 숨겨버림
end;

var
  MyObject: TAncestor;
begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!'; // 에러
  TDescendant(MyObject).Value := 'Hello!'; // 동작함!
end;
```

MyObject는 TDescendant의 인스턴스를 가지고 있지만 TAncestor로 선언되었습니다. 따라서 컴파일러는 MyObject.Value를 TAncestor에 선언된 정수 필드에 대해 참조하는 것으로 해석합니다. 하지만 TDescendant 객체에 두 필드 모두 존재하며, 상속된 Value는 새로

운 Value에 의해 숨겨졌으며, 타입 캐스트를 통해 액세스할 수 있습니다.

클래스 필드

클래스 필드는 객체 참조 없이(위에서 살펴봤던 일반적인 "인스턴스 필드"와 다름) 액세스 가능한 클래스의 데이터 필드입니다. 클래스 필드에 저장된 데이터는 클래스의 모든 인스턴스들이 공유하며, 클래스를 참조하거나 클래스의 인스턴스를 참조하는 변수를 통해 액세스할 수 있습니다.

클래스 선언 내에서 class var 블록 선언으로 클래스 필드의 블록을 만들 수 있습니다. class var 이후에 선언된 모든 필드들은 정적인 저장 특성을 가지게 됩니다. class var 블록은 다음과 같은 것들을 만나면 끝나게 됩니다.

1. 다른 class var 혹은 var 선언
2. 프로시저나 함수 선언 (클래스 프로시저와 클래스 함수 포함)
3. 속성 선언 (클래스 속성 포함)
4. 생성자 혹은 소멸자 선언
5. 가시성 지정 (public, private, protected, published, strict private, strict protected)

예를 들면 다음과 같습니다.

```
type
    TMyClass = class
      public
        class var          // 클래스 정적 필드들의 블록을 시작
          Red: Integer;
          Green: Integer;
          Blue: Integer;
        var                // class var 블록의 끝
          InstanceField: Integer;
    end;
```

클래스 필드 Red, Green, Blue는 다음과 같은 코드로 액세스할 수 있습니다.

```
TMyClass.Red := 1;
TMyClass.Green := 2;
TMyClass.Blue := 3;
```

클래스 필드는 클래스의 인스턴스를 통해서도 액세스가 가능합니다. 아래와 같이 선언했다면,

```
var
  myObject: TMyClass;
```

아래 코드는 위의 Red, Green, Blue 대입문과 같은 효과를 갖습니다.

```
myObject.Red := 1;
myObject.Green := 2;
myObject.Blue := 3;
```

메소드

메소드는 클래스와 연관된 프로시저나 함수입니다. 메소드를 호출하려면 메소드가 작동하는 객체(클래스 메소드의 경우에는 클래스)를 지정해야 합니다. 예를 들어 SomeObject.Free 호출은 SomeObject 객체의 Free 메소드를 호출합니다.

메소드 선언과 구현

클래스 선언 내에서 메소드는 프로시저나 함수의 헤더로 나타나며, forward 선언처럼 동작합니다. 각메소드는 클래스 선언 이후의 같은 모듈 내에서 정의적 선언으로 구현되어야 합니다. 예를 들어, TMyClass의 선언이 DoSomething이라는 메소드를 포함한다고 가정해 봅시다.

```
type
  TMyClass = class(TObject)
    ...
    procedure DoSomething;
    ...
  end;
```

DoSomething의 선언 정의는 모듈 뒷 부분에 나타나야 합니다.

```
procedure TMyClass.DoSomething;
begin
  ...
end;
```

클래스 선언은 유닛의 인터페이스나 임플먼테이션 섹션 중 어디에나 있을 수 있지만, 클래스 메소드의 정의적 선언은 임플먼테이션 섹션에 있어야 합니다.
정의적 선언의 헤더에서 메소드 이름은 항상 해당 메소드가 속한 클래스 이름으로 한정되어야 합니다. 헤더는 클래스 선언에 있는 헤더의 파라미터 목록을 반복할 수 있으며, 그런 경우 파라미터의 순서, 타입 및 이름이 정확하게 일치해야 합니다. 메소드가 함수인 경우에는 리턴값의 타입도 반드시 일치해야 합니다.
메소드 선언은 다른 함수나 프로시저에는 사용되지 않는 특수한 지시어를 포함할 수 있습니다. 지시어는 정의적 선언이 아닌 클래스 선언에만 나타나야 하며, 다음과 같은 순서가 되어야 합니다.

```
reintroduce; overload; binding; calling convention; abstract; warning
```

여기서 *binding*은 virtual, dynamic 또는 override이고, *calling convention*은 register, pascal, cdecl, stdcall 또는 safecall이며, *warning*은 platform, deprecated 또는 library입니다.

■ inherited

예약어 inherited는 다형적(polymorphic) 동작을 구현하는 데 특수한 역할을 수행합니다. inherited는 메소드 정의문에 나타나며, 뒤에 식별자가 따라오거나 따라오지 않을 수 있습니다. inherited 다음에 메소드 멤버의 이름이 나오면, 이것은 일반적인 메소드 호출 또는 속성이나 필드의 참조를 나타내는데, 한 가지 다른 점은 이 메소드를 찾을 때, 메소드가 속한 클래스의 바로 위 조상 클래스부터 찾기 시작한다는 점입니다. 예를 들면,

```
inherited Create(...);
```

이것은 메소드의 정의에 나타나며, 상속된 Create를 호출합니다.
inherited 다음에 식별자가 따라오지 않으면, 이것은 자신을 포함하는 메소드와 같은 이름을 가지는 상속된 메소드를 나타내거나, 메시지 핸들러인 경우 동일 메시지에 대해 상속된 메시지 핸들러를 나타냅니다. 이러한 경우, inherited에는 명시적인 파라미터가 없지만, 해당 메소드에 전달된 파라미터와 같은 파라미터를 상속된 메소드로 전달합니다. 예를 들면,

```
inherited;
```

이 문법은 생성자의 구현에서 자주 나타납니다. 상속된 생성자를 자손에게 전달한 파라미터와 같은 파라미터와 함께 호출합니다.

■ Self

메소드의 구현에서 Self 식별자는 메소드가 호출된 객체를 참조합니다. 예를 들어, Classes 유닛에서 TCollection의 Add 메소드 구현은 다음과 같습니다.

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

Add 메소드는 항상 FItemClass 필드가 참조하는 클래스에서 Create 메소드를 호출합니다. 여기서 FItemClass 필드는 TCollectionItem의 자손입니다. TCollectionItem.Create는 TCollection 타입의 파라미터 하나만을 가지므로, Add는 Add가 호출된 TCollection 인스턴스에 파라미터를 전달합니다. 이것은 다음 코드에서 설명합니다.

```
var MyCollection: TCollection;
...
MyCollection.Add; // TCollectionItem.Create 메소드로 MyCollection이 전달됨
```

Self는 여러가지 이유로 유용합니다. 예를 들어, 클래스 타입에서 선언된 멤버 식별자는 클래스 메소드 중 하나의 블럭에서 다시 선언될 수 있습니다. 이러한 경우, 원래의 멤버 식별자를 Self.Identifier로 액세스할 수 있습니다.

클래스 메소드의 Self에 대한 내용은 "클래스 메소드"를 참조하십시오.

메소드 바인딩

메소드는 정적(static), 가상(virtual) 또는 동적(dynamic)일 수 있습니다. 기본값은 static입니다. 가상 메소드와 동적 메소드는 오버라이드될 수 있고 추상화(abstract)될 수 있습니다. 이런 지정은 한 클래스 타입의 변수가 자손 클래스 타입의 값을 가지게 되면 역할을 하게 됩니다. 이들 지시어는 어떤 메소드가 호출되었을 때 어떤 구현이 선택될지를 결정합니다.

■ 정적 메소드

메소드는 기본적으로 정적입니다. 정적 메소드가 호출되면, 메소드가 호출된 클래스나 객체

변수의 (컴파일 시점에서) 선언된 타입에 따라 선택될 구현이 결정됩니다. 다음 예제에서 Draw는 정적 메소드입니다.

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

위와 같이 선언했다면, 다음 코드는 정적 메소드를 호출했을 때의 동작을 보여줍니다. Figure.Draw에 대한 두 번째 호출에서, Figure 변수는 TRectangle 클래스의 객체를 참조하지만 Figure 변수의 선언된 타입이 TFigure이기 때문에 TFigure의 Draw의 구현이 호출됩니다.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;              // TFigure.Draw 호출
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;              // TFigure.Draw 호출

  TRectangle(Figure).Draw; // TRectangle.Draw 호출

  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;           // TRectangle.Draw 호출
  Rectangle.Destroy;
end;
```

■ 가상 메소드 및 동적 메소드

메소드를 가상(virtual) 또는 동적(dynamic)으로 만들려면 해당 선언에 virtual이나 dynamic 지시어를 추가하면 됩니다. 정적 메소드와는 달리, 가상 메소드와 동적 메소드는 자손 클래스에서 오버라이드될 수 있습니다. 오버라이드된 메소드가 호출되면, (변수의 선언된 타입이 아닌) 메소드가 호출된 클래스나 객체의 실제(런타임) 타입에 따라 실행할 구현이 결정됩니다.

메소드를 오버라이드하려면 override 지시어를 사용하여 해당 메소드를 재선언합니다. override 선언은 조상 선언과 파라미터들의 순서와 타입, 그리고 결과 타입(있는 경우)이 일치해야 합니다.
다음 예제에서 TFigure에서 선언된 Draw 메소드는 두 자손 클래스에서 오버라이드됩니다.

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

위와 같이 선언했다면, 다음 코드는 실제 객체의 타입이 런타임 시 변하는 변수를 통해 가상 메소드를 호출할 경우의 동작을 보여줍니다.

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw; // TRectangle.Draw 호출
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw; // TEllipse.Draw 호출
  Figure.Destroy;
end;
```

오버라이드가 가능한 것은 가상 메소드와 동적 메소드 뿐입니다. 그러나 모든 메소드를 오버로드할 수는 있습니다. "메소드 오버로드"를 참조하십시오.
델파이 컴파일러는 final 가상 메소드라는 개념도 지원합니다. 가상 메소드에 final 예약어가 사용되면, 자손 클래스들은 그 메소드를 오버라이드할 수 없게 됩니다. final 예약어를 사용하는 것은 클래스가 어떻게 사용되도록 의도되었는지를 기록해둘 수 있는 중요한 설계상의 결정입니다. 또한 컴파일러에게 생성할 코드를 더 최적화할 수 있는 힌트를 주기도 합니다.

Note

확실하고 눈에 띄는 이점이 있을 때만 동적 메소드를 사용하십시오. 일반적으로는 가상 메소드를 사용하면 됩니다.

■ 가상 메소드와 동적 메소드 비교

델파이에서 가상 메소드와 동적 메소드는 의미적으로 유사합니다. 하지만 가상 메소드와 동적 메소드는 런타임 시 메소드 호출을 디스패치(dispatch)하는 구현에서만 서로 다릅니다. 가상 메소드는 속도 관점에서 최적화하는 반면, 동적 메소드는 코드 크기의 관점에서 최적화합니다.일반적으로, 가상 메소드는 다형적인 동작을 구현하기 위한 가장 효율적인 방법입니다. 응용 프로그램에서 기반 클래스가 많은 자손 클래스들에서 상속되는 여러 오버라이드 가능한 메소드들을 선언할 경우 동적 메소드가 유용합니다.

■ 오버라이드와 숨김

동일 메소드 식별자와 파라미터 시그너처(signature)를 지정한 상속된 메소드에서 override를 포함하지 않을 경우, 새로운 선언은 오버라이드 대신 단순히 상속된 메소드를 숨기기만 합니다(hide). 두 메소드 모두 자손 클래스에 존재하며, 메소드 이름은 정적으로 바인드됩니다. 아래의 예를 참고하십시오.

```
type
  T1 = class(TObject)
    procedure Act; virtual
  end;
  T2 = class(T1)
    procedure Act; // Act가 재선언되었지만 오버라이드된 것은 아님
  end;
var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act; // T1.Act 호출
end;
```

■ reintroduce

reintroduce 지시어는 이전에 선언된 가상 메소드를 숨기게 되는 상황에 대해 컴파일러 경고를 표시하지 않도록 강제합니다. 예를 들면 다음과 같습니다.

```
procedure DoSomething; reintroduce; // 조상 클래스도 DoSomething 메소드를 가지고 있음
```

상속받은 가상 메소드를 새로운 메소드로 숨기려고 할 때 reintroduce를 사용하십시오.

■ 추상 메소드

추상(abstract) 메소드는 가상 메소드 혹은 동적 메소드로서, 선언된 클래스에 구현이 없는 경우입니다. 추상 메소드는 자손 클래스에서 구현합니다. 추상 메소드는 virtual 또는 dynamic 다음에 abstract 지시어를 추가하여 선언해야 합니다. 예를 들면 다음과 같습니다.

```
procedure DoSomething; virtual; abstract;
```

해당 메소드가 오버라이드된 클래스 혹은 클래스의 인스턴스에서만 추상 메소드를 호출할 수 있습니다.

클래스 메소드

대부분의 메소드는 인스턴스 메소드라고 불리는데, 각각의 객체 인스턴스에 대해 동작하기 때문입니다. 클래스 메소드는 객체 대신 클래스에 대해 동작하는 메소드입니다(생성자 제외). 클래스 메소드에는 일반적인 클래스 메소드와 클래스 정적 메소드의 두 종류가 있습니다.

■ 일반적인 클래스 메소드

클래스 메소드의 정의는 예약어 class로 시작되어야 합니다. 아래 예를 보십시오.

```
type
  TFigure = class
  public
     class function Supports(Operation: string): Boolean; virtual;
     class procedure GetInfo(var Info: TFigureInfo); virtual;
     ...
  end;
```

클래스 메소드의 정의적 선언도 역시 class로 시작해야 합니다. 아래 예를 봅시다.

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
    ...
end;
```

클래스 메소드의 정의적 선언에서, 식별자 Self는 메소드가 호출된 클래스를 의미합니다(자신이 정의된 클래스의 자손일 수도 있습니다). 클래스 C에서 메소드가 호출되면, Self는 C

클래스 타입입니다. 따라서 Self를 이용하여 인스턴스 필드, 인스턴스 속성, 일반적인 (객체) 메소드를 액세스할 수가 없습니다. 하지만 생성자나 다른 클래스 메소드, 혹은 클래스 속성이나 클래스 필드를 호출하기 위해 Self를 사용할 수는 있습니다.
클래스 메소드는 클래스 참조나 객체 레퍼런스를 통해 호출할 수 있습니다. 객체 레퍼런스를 통해 호출했을 때는, 객체의 클래스는 Self의 값이 됩니다.

■ 클래스 정적 메소드

일반적인 클래스 메소드처럼, 클래스 정적 메소드(class static method)는 객체 레퍼런스 없이 액세스 가능합니다. 일반적인 클래스 메소드와 다른 점은, 클래스 정적 메소드는 Self 파라미터가 아예 존재하지 않는다는 것입니다. 인스턴스 멤버를 액세스할 수 없다는 점도 동일합니다. (역시 클래스 필드, 클래스 속성, 클래스 메소드를 액세스할 수 있습니다) 일반 클래스 메소드와 다른 또 한가지는, 클래스 정적 메소드는 가상 메소드로 선언될 수 없다는 것입니다. 메소드를 클래스 정적 메소드로 만들려면 선언의 뒤에 static이라는 예약어를 추가하면 됩니다. 아래의 예를 봅시다.

```
type
   TMyClass = class
     strict private
       class var
       FX: Integer;

     strict protected
       // 노트: 클래스 속성의 액세스 지정자는 class static으로 선언되어야 합니다.
       class function GetX: Integer; static;
       class procedure SetX(val: Integer); static;

     public
       class property X: Integer read GetX write SetX;
       class procedure StatProc(s: String); static;
   end;
```

일반 클래스 메소드처럼, 클래스 정적 메소드를 호출하려면 객체 참조 없이 클래스 타입을 통해서 호출하면 됩니다. 아래의 예를 참고하십시오.

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

메소드 오버로드

메소드는 overload 지시어를 사용하여 재선언할 수 있습니다. 재선언된 메소드가 조상과 다른 파라미터 시그너처를 가지는 경우 재선언된 메소드를 숨기지 않고 상속된 메소드를 오버로드합니다. 자손 클래스의 메소드를 호출하면 호출한 루틴의 파라미터와 일치하는 구현이 선택됩니다.

가상 메소드를 오버로드하는 경우 자손 클래스에서 가상 메소드를 재선언할 때 reintroduce 지시어를 사용합니다. 아래 예를 참고하십시오.

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;
  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
  ...
  SomeObject := T2.Create;
  SomeObject.Test('Hello!');  // T2.Test 호출
  SomeObject.Test(7);         // T1.Test 호출
```

클래스 내에서 이름이 같은 여러 오버로드된 메소드를 published로 사용할 수 없습니다. 런타임 타입 정보(RTTI)를 유지하기 위해 각각의 published 멤버에 대한 고유 이름이 필요합니다.

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer; // 에러
    ...
```

---

read나 write 속성 지정자 역할을 하는 메소드는 오버로드할 수 없습니다.

오버로드된 메소드의 구현의 파라미터 리스트는 클래스 선언에서 지정했던 파라미터 리스트와 같아야 합니다.오버로드에 대한 자세한 내용은 5장의 "프로시저와 함수 오버로드"를 참조하십시오.

생성자

생성자(constructor)는 인스턴스 객체를 만들고 초기화하는 특수한 메소드입니다. 생성자 선언은 프로시저 선언과 비슷하지만 constructor로 시작합니다. 예를 들면 다음과 같습니다.

```
constructor Create;
constructor Create(AOwner: Tcomponent);
```

생성자는 기본인 register 호출 규칙을 사용해야 합니다. 선언에서는 리턴 값을 지정하지 않지만, 생성자는 자신이 생성한 객체 혹은 호출된 객체에 대한 참조를 리턴합니다.
클래스는 두 개 이상의 생성자를 가질 수 있지만, 대부분의 클래스는 하나만 있습니다. 일반적으로 Create 생성자를 호출하는 것이 일반적입니다.
객체를 만들려면 클래스 타입에 대한 생성자 메소드를 호출합니다. 예를 들면, 다음과 같습니다.

```
MyObject := TMyClass.Create;
```

이 문법은 힙(heap)에 새 객체를 위한 메모리를 할당하고, 모든 서수 필드 값을 영(0)으로 설정하며, 모든 포인터와 클래스 타입 필드에 nil을 지정하고, 모든 문자열 필드를 빈(empty) 문자열로 초기화합니다.
그 다음으로 생성자의 구현에서 지정한 다른 동작들이 실행됩니다. 일반적으로 생성자에 전달된 파라미터로 객체가 초기화됩니다. 마지막으로 생성자는 새로 할당 및 초기화된 객체에 대한 참조를 리턴합니다. 리턴 값 타입은 생성자를 호출하면서 지정한 클래스 타입과 동일합니다.
클래스 참조에서 호출된 생성자의 실행 중에 예외가 발생하면, 자동으로 Destroy 소멸자 정상적으로 생성되지 못한 객체를 소멸하기 위해 자동으로 호출됩니다.
클래스 참조가 아닌 객체 참조를 사용하여 생성자를 호출하면 생성자는 객체를 생성하지 않습니다. 그 대신, 생성자는 지정한 객체에 대해 생성자 구현에 있는 문장들만 실행한 다음 객체에 대한 참조를 리턴합니다. 생성자는 일반적으로 상속된 생성자를 실행하기 위해 예약어 inherited와 함께 호출됩니다.
클래스 타입과 클래스 타입 생성자에 대한 예제는 다음과 같습니다.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ...
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner); // 상속된 부분들을 초기화
  Width := 65;             // 상속된 속성을 변경
  Height := 65;
  FPen := TPen.Create;     // 새 필드들을 초기화
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

생성자의 첫 번째 동작은 일반적으로 상속된 생성자를 호출하여 객체의 상속된 필드들을 초기화하는 것입니다. 그런 다음 생성자는 자손 클래스에서 추가한 필드들을 초기화합니다. 생성자는 항상 새로운 객체에 할당된 메모리를 지우기 때문에, 모든 필드 값은 처음에 0(서수 타입), nil(포인터 타입 및 클래스 타입), 빈 문자열(문자열 타입) 또는 Unassigned(variant 타입)이 됩니다. 따라서 0이 아닌 값이나 빈 값이 아닌 값이 아니라면 생성자에서 필드를 초기화할 필요가 없습니다.

클래스 타입 식별자를 통해 호출된 경우, virtual로 선언된 생성자는 정적(static) 생성자와 동일합니다. 그러나 클래스 참조 타입과 함께 사용하면 가상 생성자는 다형적인 객체 생성, 즉 타입이 컴파일 시에 알려지지 않는 객체 생성을 허용합니다. 6장의 "클래스 참조"를 참조하십시오.

소멸자

소멸자는 호출한 객체를 소멸하고 그 메모리를 해제하는 특수한 메소드입니다. 소멸자 선언은 프로시저 선언과 비슷하지만 destructor로 시작합니다. 예를 들면 다음과 같습니다.

```
destructor Destroy;
destructor Destroy; override;
```

소멸자는 기본인 register 호출 규칙을 사용해야 합니다. 클래스는 두 개 이상의 소멸자를 가질 수 있지만, 각 클래스는 상속된 Destroy 메소드를 오버라이드하고 다른 소멸자는 선언하지 않는 것이 좋습니다.
소멸자를 호출하려면 인스턴스 객체를 참조해야 합니다. 예를 들면 다음과 같습니다.

```
MyObject.Destroy;
```

소멸자가 호출되면 가장 먼저 소멸자 구현에서 지정된 동작들이 실행됩니다. 일반적으로 포함된 모든 객체를 소멸하고 객체에 의해 할당되었던 리소스를 해제하는 작업들입니다. 다음으로 객체에 할당되었던 메모리가 해제됩니다.
소멸자 구현의 예는 다음과 같습니다.

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

소멸자 구현의 마지막 동작은 일반적으로 상속된 소멸자를 호출하여 객체의 상속된 필드를 소멸하는 것입니다.
객체를 생성하는 중에 예외가 발생하는 경우 Destroy가 자동으로 호출되어 정상적으로 생성되지 않은 객체를 제거합니다. 이는 불완전하게 생성된 객체를 제거하는 코드가 Destroy에 구현되어 있어야 한다는 것을 의미합니다. 생성자는 다른 동작을 실행하기 전에 새 객체의 필드를 0 또는 빈 값으로 설정하기 때문에, 불완전하게 생성된 객체의 클래스 타입 및 포인터 타입 필드는 항상 nil입니다. 그러므로 소멸자는 클래스 타입 또는 포인터 타입 필드를 작업하기 전에 nil인지 확인해야 합니다. Destroy 대신 Free 메소드(TObject에서 정의)를 호출하면 객체를 소멸하기 전에 내부적으로 자동으로 nil 값인지 확인하므로 편리합니다.

메시지 메소드

메시지 메소드는 동적으로 디스패치된 메시지에 대한 응답 동작을 구현합니다. 메시지 메소드 문법은 모든 플랫폼에서 지원됩니다. VCL은 Windows 메시지에 응답하기 위해 메시지 메소드를 사용합니다. 메시지 메소드를 만들려면 메소드 선언에 message 지시어를 추가하고, 그 뒤에 메시지 ID를 지정하는 1과 49151 사이의 정수 상수를 덧붙이면 됩니다. Win32

메시지의 경우에는 이 정수 상수(Win32 메시지 ID)들은 VCL 컨트롤의 메시지 메소드들을 위해 이미 정의되어 있으며, 해당하는 레코드 타입들과 함께 Messages 유닛에 정의되어 있습니다. 메시지 메소드는 var 파라미터 하나만을 갖는 프로시저여야 합니다.
예를 들면, 다음과 같습니다.

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ...
  end;
```

메시지 메소드는 상속된 메시지 메소드를 오버라이드하기 위해 override 지시어를 포함할 필요가 없습니다. 사실, 메시지 메소드는 오버라이드하는 메소드와 같은 메소드 이름을 가질 필요도 없습니다. 오직 메시지 ID에 따라 메소드가 응답할 메시지와 오버라이드하는 것인지의 여부가 결정됩니다.

메시지 메소드의 구현

메시지 메소드의 구현은 다음 예제에서와 같이 상속된 메시지 메소드를 호출할 수 있습니다.

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Message.CharCode = Ord(#13) then
    ProcessEnter
  else
    inherited;
end;
```

메시지 메소드에서의 inherited 문은 클래스 계층에서 역으로 검색하여 현재 메소드와 같은 ID를 갖는 첫 메시지 메소드를 호출하고, 자동으로 메시지 레코드를 전달합니다. 조상 클래스들이 모두 해당 메시지 ID에 대한 메시지 메소드를 구현하지 않은 경우, inherited는 TObject에서 정의된 DefaultHandler 메소드를 호출합니다.
TObject에 있는 DefaultHandler의 구현에서는 아무런 동작도 실행하지 않고 그냥 리턴됩니다. DefaultHandler를 오버라이드하면 클래스는 자체적으로 기본 메시지 처리를 구현할 수 있습니다. 컨트롤의 DefaultHandler 메소드는 Win32 API의 DefWindowProc 함수를 호출합니다.

메시지 디스패칭

메시지 핸들러는 직접 호출되는 일은 거의 없습니다. 대신, TObject에서 상속된 Dispatch 메소드를 사용하여 메시지가 객체에 디스패치됩니다.

```
procedure Dispatch(var Message);
```

Dispatch에 전달된 Message 파라미터는 레코드로서, 첫 필드가 메시지 ID를 가진 Word 타입이어야 합니다.
Dispatch는 클래스 계층에서 역으로 검색하여(호출된 객체의 클래스에서 시작) 전달된 ID에 해당하는 처음으로 발견된 메시지 메소드를 호출합니다. 지정 ID에 대한 메시지 메소드가 발견되지 않으면 Dispatch는 DefaultHandler를 호출합니다.

속성

속성(property)은 필드와 마찬가지로 객체의 속성(attribute)을 정의합니다. 그러나 필드가 단순히 내용을 읽거나 변경할 수 있는 메모리 위치인 반면, 속성은 해당 데이터를 읽거나 쓰는 작업에 특정 동작을 연결합니다. 속성은 객체의 특성에 대한 액세스 과정을 관리할 수 있는 방법이며, 또한 객체의 속성을 (정적인 값이 아닌) 계산 결과로 지정할 있게 해줍니다.
속성의 선언에서는 이름과 타입을 지정하며, 하나 이상의 액세스 지정자(access specifier)를 포함합니다. 속성 선언의 문법은 다음과 같습니다.

```
property propertyName[indexes]: type index integerConstant specifiers;
```

여기서,

- *propertyName*은 유효한 식별자입니다.
- [indexes]는 옵션이며, 세미콜론으로 구분되는 일련의 파라미터 선언들을 나타냅니다. 각 파라미터 선언은 $identifier_1$, ..., $identifier_n$: *type*과 같은 형태를 가집니다. 자세한 내용은 아래의 "배열 속성"을 참조하십시오.
- *type*은 이미 정의되었거나 이전에 선언된 데이터 타입이어야 합니다.
  따라서, property Num: 0..9 ...와 같은 속성 선언은 잘못되었습니다.

- index *integerConstant* 절은 옵션입니다. 자세한 내용은 아래의 "Index 지정자"를 참조하십시오.
- *specifiers*는 read, write, stored, default(또는 nodefault), implements 지정자의 리스트입니다. 모든 속성 선언에는 적어도 하나의 read 또는 write 지정자가 있습니다. implements에 대한 내용은 11장의 "위임(delegation)으로 인터페이스 구현"을 참조하십시오.

속성은 속성의 액세스 지정자에 의해 정의됩니다. 필드와 달리, 속성은 var 파라미터로 전달할 수 없으며, 속성에 @ 연산자를 사용할 수도 없습니다. 이는 속성이 반드시 메모리에 있는 것이 아니기 때문입니다. 예를 들면, 데이터베이스에서 값을 가져오거나 임의(random)의 값을 생성하는 read 메소드를 지정할 수 있습니다.

속성 액세스

모든 속성은 read 지정자를 가지거나, write 지정자를 가지거나, 또는 둘 다 가집니다. 이러한 지정자를 액세스 지정자(access specifier)라고 하며 다음과 같은 형태를 가집니다.

```
read fieldOrMethod
write fieldOrMethod
```

여기서 *fieldOrMethod*는 속성이 선언된 클래스나 조상 클래스에서 선언된 필드 혹은 메소드의 이름입니다.

- *fieldOrMethod*가 같은 클래스에서 선언되는 경우 속성의 선언보다 앞에 선언해야 합니다. 조상 클래스에서 선언되는 경우에는 *fieldOrMethod*를 자손 클래스에서 볼 수 있어야 합니다. 즉, 다른 유닛에서 선언된 조상 클래스의 private 필드나 메소드여서는 안됩니다.
- *fieldOrMethod*가 필드인 경우 속성과 같은 타입이어야 합니다.
- fieldOrMethod가 메소드인 경우, dynamic이어서는 안되며, virtual인 경우 오버로드될 수 없습니다. 또한, published 속성의 액세스 메소드는 기본인 register 호출 규칙을 사용해야 합니다.
- read 지정자에서 *fieldOrMethod*가 메소드인 경우, 파라미터가 없고 리턴 타입이 속성의 타입과 동일한 함수이어야 합니다. (인덱스 속성이나 배열 속성의 액세스 메소드는 예외입니다)
- write 지정자에서 *fieldOrMethod*가 메소드인 경우, 속성의 타입과 동일한 타입의 단 하나의 값 파라미터 또는 상수 파라미터만을 가지는 프로시저여야 합니다. (배열 속성이나 인덱스 속성의 경우 더 많아집니다)

예를 들어, 다음과 같이 선언했다고 가정하면,

```
property Color: TColor read GetColor write SetColor;
```

GetColor 메소드는 다음과 같이 선언해야 합니다.

```
function GetColor: TColor;
```

또, SetColor 메소드는 다음 중 하나로 선언해야 합니다.

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: Tcolor);
```

물론 SetColor의 파라미터 이름이 Value일 필요는 없습니다.
속성이 표현식에서 사용되면, read 지정자에서 지정한 필드 혹은 메소드를 사용하여 속성 값을 읽습니다. 속성이 대입문의 왼쪽에서 사용되면, write 지정자에서 지정한 필드 혹은 메소드를 사용하여 속성 값을 씁니다.
아래 예제에서는 Heading이라는 published 속성을 가진 TCompass라는 클래스를 선언합니다. Heading의 값은 FHeading 필드를 통해 읽고 SetHeading 프로시저를 통해 씁니다.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    ...
  end;
```

위와 같이 선언했다면, 아래의 문장은,

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

아래의 문장과 동일한 의미입니다.

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

TCompass 클래스에서 Heading 속성의 읽기 작업과 관련한 다른 동작이 없습니다. read 작업은 오직 FHeading 필드에 저장된 값을 읽어 오는 것 뿐입니다. 반면, Heading 속성에 값을 대입하면 SetHeading 메소드에 대한 호출로 해석되며, SetHeading 메소드에서는 FHeading 필드에 새 값을 저장하는 작업 뿐만 아니라 다른 동작을 수행할 수 있습니다. 예를 들어, SetHeading은 다음과 같이 구현될 수 있습니다.

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint; // 새 값을 반영하기 위해 유저 인터페이스를 업데이트합니다.
  end;
end;
```

선언에서 read 지정자만을 가지는 속성은 읽기 전용(read-only) 속성이고, 선언에 write 지정자만을 가지는 속성은 쓰기 전용(write-only) 속성입니다. 읽기 전용 속성에 값을 대입하려 하거나 표현식에 쓰기 전용 속성을 사용하면 에러가 발생합니다.

배열 속성

배열 속성은 인덱스를 가지는 속성입니다. 배열 속성은 리스트의 항목들, 컨트롤의 자식 컨트롤들, 비트맵의 픽셀 등을 나타낼 수 있습니다.
배열 속성의 선언은 인덱스의 이름과 타입을 지정하는 파라미터 리스트가 추가됩니다. 예를 들면, 다음과 같습니다.

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string): string read GetValue write SetValue;
```

파라미터 선언을 괄호 대신 대괄호로 묶는 것을 제외하면, 인덱스 파라미터 리스트의 형식은 프로시저나 함수의 파라미터 리스트의 형식과 같습니다. 배열에서는 서수(ordinal) 타입 인

덱스만을 사용할 수 있는 것과 달리, 배열 속성에서는 모든 타입의 인덱스를 허용합니다.
배열 속성에서 액세스 지정자는 필드가 아닌 메소드여야 합니다. read 지정자의 메소드는 속성의 인덱스 파라미터 리스트와 동일한(파라미터의 수와 타입, 순서 모두 동일) 파라미터 리스트와 해당 속성의 타입과 일치하는 리턴 타입을 가지는 함수여야 합니다. write 지정자의 메소드는 속성의 인덱스 파라미터 리스트와 동일한(파라미터의 수와 타입, 순서 모두 동일) 파라미터 리스트에 추가로 해당 속성의 타입과 동일한 타입의 값 파라미터 또는 상수 파라미터를 포함하는 프로시저여야 합니다.
예를 들어, 위의 배열 속성의 액세스 메소드들을 다음과 같이 선언할 수 있습니다.

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

배열 속성을 액세스하려면 속성 식별자에 인덱스를 지정하면 됩니다. 예를 들어, 다음의 문장들은,

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

아래의 문장들과 동일한 의미입니다.

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

배열 속성의 정의 뒤에 default 지시어를 추가할 수 있습니다. 이런 경우, 배열 속성은 클래스의 기본 속성(default property)이 됩니다. 예를 들면, 다음과 같습니다.

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ...
  end;
```

클래스가 기본 속성을 가지는 경우, object.property[index]를 object[index]로 줄여서 기본 속성을 액세스할 수 있습니다. 예를 들어, 위와 같은 선언에서 StringArray.Strings[7]은 StringArray[7]로 줄여 쓸 수 있습니다. 클래스는 주어진 시그니처(배열 파라미터 리스트)에 대해서 기본 속성을 하나만 가질 수 있지만, 기본 속성을 오버로드하는 것은 가능합니다. 컴파일러는 속성을 항상 정적으로 바인드하므로, 자손 클래스에서 기본 속성을 변경하거나 숨기면 예기치 않은 동작이 일어날 수 있습니다.

index 지정자

index 지정자를 사용하면 여러 속성에서 같은 액세스 메소드를 공유하면서 다른 값을 나타낼 수 있습니다. index 지정자는 index 지시어와 -2147483647과 2147483647 사이의 정수 상수로 구성됩니다. 속성이 index 지정자를 가지는 경우, 해당 read 및 write 지정자는 필드가 아닌 메소드를 지정해야 합니다. 예를 들면, 다음과 같습니다.

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Char;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
    property Left:Longint index 0 read GetCoordinate write SetCoordinate;
    property Top:Longint index 1 read GetCoordinate write SetCoordinate;
    property Right:Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom:Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint read GetCoordinate write
  SetCoordinate;
    ...
  end;
```

index 지정자가 있는 속성의 액세스 메소드는 추가로 정수 타입의 값 파라미터를 가져야 합니다. 이 추가 파라미터는 read 함수에서는 맨 마지막 파라미터여야 하고, write 프로시저에서는 끝에서 두 번째 파라미터(속성 값을 지정하는 파라미터의 앞)여야 합니다. 프로그램에서 속성에 액세스하면 속성의 index 지정자에서 지정했던 정수 상수가 액세스 메소드에 자동으로 전달됩니다.

위와 같이 선언하고 TRectangle 타입의 Rectangle 객체가 있다면,

```
Rectangle.Right := Rectangle.Left + 100;
```

위 문장은 아래 문장과 같은 의미가 됩니다.

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

■ 저장소 지정자

옵션인 stored, default, nodefault 지시어를 저장소 지정자(storage specifier)라고 합니다. 저장소 지정자는 프로그램 동작에 영향을 주지는 않지만, published 속성의 값들을 폼 파일에 저장할 것인지 여부를 지정합니다.

stored 지시어 뒤에는 True/False 값이나, 부울 타입의 필드 이름이나, 파라미터가 없고 부울 값을 리턴하는 메소드 이름이 따라와야 합니다. 예를 들면, 다음과 같습니다.

```
property Name: TComponentName read FName write SetName stored False;
```

속성이 stored 지시어를 갖지 않는 경우 stored True가 지정된 것처럼 처리됩니다.

default 지시어 뒤에는 해당 속성과 같은 타입의 상수가 따라와야 합니다. 예를 들면, 다음과 같습니다.

```
property Tag: Longint read FTag write FTag default 0;
```

Note

서수 값 2147483648를 default 값으로 사용할 수 없습니다. 이 값은 내부적으로 nodefault를 나타내기 위해 사용됩니다.

상속된 default 값을 오버라이드하면서 값을 새로 지정하지 않으려면 nodefault 지시어를 사용합니다. default 및 nodefault 지시어는 서수 타입과 집합 타입에 대해서만 지원됩니다(집합 타입은 집합 기반 타입의 범위가 0과 31 사이의 순서 값을 가지는 경우에만 지원). 서수 타입 및 집합 타입의 속성이 default 또는 nodefault 없이 선언되면 nodefault가 지정된 것으로 간주됩니다. 실수, 포인터, 문자열은 암시적으로 각각 0, nil 및 '' (빈 문자열)을 default 값으로 가집니다.

Note

속성의 값은 자동으로 default 값으로 초기화되지 않습니다. 다시 말해, default 지시어는 속성 값이 폼 파일에 저장될 때만 영향을 미치며, 새로 생성된 인스턴스의 속성의 초기 값에는 영향을 미치지 않는다는 것입니다.

컴포넌트의 상태를 저장할 때 컴포넌트의 published 속성에 대한 저장소 지정자를 확인합니다. 속성의 현재 값이 해당 default 값과 다르고(또는 default 값이 없는 경우) stored 지정자가 True이면 속성 값이 저장됩니다. 그렇지 않으면 속성 값이 저장되지 않습니다.

노트: 속성의 값은 자동으로 default 값으로 초기화되지 않습니다. 다시 말해, default 지시어는 속성 값이 폼 파일에 저장될 때만 영향을 미치며, 새로 생성된 인스턴스의 속성의 초기 값에는 영향을 미치지 않는다는 것입니다.

저장소 지정자는 배열 속성에 대해 지원되지 않습니다. default 지시어는 배열 속성 선언에

서 사용될 때 다른 의미를 가집니다. "속성" 절의 "배열 속성"을 참조하십시오.

속성 오버라이드 및 재선언

속성을 선언할 때 타입을 지정하지 않는 경우를 속성 오버라이드(property override)라고 합니다. 속성 오버라이드에서 기존의 속성의 상속된 가시성이나 지정자를 변경할 수 있습니다. 속성 오버라이드의 가장 간단한 형태는 property 예약어와 바로 뒤이은 상속된 속성 식별자 만으로 구성되며, 이 형태는 속성의 가시성을 변경하는 데 사용됩니다. 예를 들어, 조상 클래스에서 속성을 protected로 선언했다면 파생된 클래스에서는 해당 속성을 클래스의 public 또는 published 섹션에서 재선언할 수 있습니다.

속성 오버라이드는 read, write, stored, default, nodefault 지시어를 포함할 수 있으며 이러한 지시어는 각각 상속된 속성의 해당하는 지시어를 오버라이드합니다. 오버라이드는 상속된 액세스 지정자를 교체하고 필요한 지정자를 추가하거나 속성의 가시성을 높일 수 있지만, 액세스 지정자를 제거하거나 속성의 가시성을 낮출 수는 없습니다. 오버라이드는 implements 지시어를 포함할 수도 있는데, 이것은 상속된 인터페이스들을 제거하지 않고 구현된 인터페이스 리스트에 추가합니다.

다음 선언은 속성 오버라이드의 사용법을 보여줍니다.

```
type
  TAncestor = class
    ...
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    ...
  end;
type
  TDerived = class(TAncestor)
    ...
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    ...
  end;
```

속성 Size의 오버라이드에서는 write 지정자를 추가하여 속성 값을 수정할 수 있게 합니다. Text와 Color의 오버라이드에서는 속성의 가시성을 protected에서 published로 변경합니

다. 또한 Color의 속성 오버라이드에서는 또한 속성 값이 clBlue가 아닌 경우 속성이 저장되도록 지정합니다.

속성을 재선언할 때 타입 식별자를 포함시키면 상속된 속성을 오버라이드하는 것이 아니라 숨깁니다. 이는 상속된 속성과 같은 이름의 새로운 속성이 만들어진다는 것을 의미합니다. 타입이 지정된 모든 속성 선언은 완전한 선언이어야 하고, 따라서 최소 하나의 액세스 지정자는 포함해야 합니다.

파생된 클래스에서 속성이 숨겨지거나 오버라이드되거나 여부에 관계 없이, 속성에 대한 검색은 항상 정적입니다. 즉, 객체를 식별하는 데 사용된 변수(객체 포인터를 가진 변수)가 선언된 타입에 따라(컴파일 타임) 해당 속성 식별자의 해석이 결정됩니다. 따라서, 다음 코드를 실행한 후에 MyObject.Value의 값을 읽거나 대입하면, MyObject 변수는 TDescendant의 인스턴스를 가지고 있음에도 불구하고, Method1이나 Method2가 호출됩니다. 그러나 MyObject를 TDescendant로 타입 캐스트하면 자손 클래스의 속성과 해당 액세스 지정자를 액세스할 수 있습니다.

```
type
  TAncestor = class
    ...
    property Value: Integer read Method1 write Method2;
  end;
  TDescendant = class(TAncestor)
    ...
    property Value: Integer read Method3 write Method4;
  end;

var MyObject: TAncestor;
...
MyObject := TDescendant.Create;
```

클래스 속성

클래스 속성(class property)은 객체 참조 없이 액세스가 가능합니다. 클래스 속성의 지정자는 클래스 정적 메소드나 클래스 필드로 선언되어야 합니다. 클래스 속성은 class property 예약어로 선언됩니다. 클래스 속성은 published가 될 수 없으며, stored나 default 지정을 할 수도 없습니다.

class var 블록 선언을 사용하여 클래스 선언에 클래스 정적 필드의 블록을 만들 수 있습니다. class var 이후에 선언된 모든 필드들은 정적인 저장소 특성을 갖습니다. class var 블록은 다음과 같은 것들을 만나면 종료됩니다.

- 다른 class var 선언
- 프로시저나 함수(즉 메소드) 선언 (클래스 프로시저나 클래스 함수도 포함)
- 속성 선언 (클래스 속성 선언 포함)
- 생성자나 소멸자 선언
- 가시성 유효범위 지정자 (public, private, protected, published, strict private, strict protected)

예를 들면 아래와 같습니다.

```
type
  TMyClass = class
  strict private
    class var          // 필드들이 class 필드로 선언되어야 한다는 점에 주의하십시오
      FRed: Integer;
      FGreen: Integer;
      FBlue: Integer;
  public               // class var 블록의 끝
    class property Red: Integer read FRed write FRed;
    class property Green: Integer read FGreen write FGreen;
    class property Blue: Integer read FBlue write FBlue;
  end;
```

위의 클래스 속성들은 다음과 같은 코드로 액세스할 수 있습니다.

```
TMyClass.Red := 0;
TMyClass.Blue := 0;
TMyClass.Green := 0;
```

중첩된 타입

클래스 선언 안에서 타입 선언을 중첩하여 선언할 수 있습니다. 중첩된 타입(nested type)은 일반적으로 객체 지향 프로그래밍에서 전반적으로 사용됩니다. 중첩된 타입은 개념적으로 연관된 타입들을 함께 유지할 수 있는 방법을 제공하며, 이름 충돌을 피하기 위해서도 좋습니다.

중첩된 타입의 선언

중첩된 타입의 선언은 4장의 "타입 선언" 절에서 정의된 타입 선언 문법을 따릅니다.

```
type
  className = class [abstract | sealed] (ancestorType)
    memberList

    type
      nestedTypeDeclaration

    memberList
end;
```

중첩된 타입의 선언의 끝은 식별자가 아닌 토큰(procedure, class, type, 모든 가시성 유효 범위 지정자)이 처음 나타나는 곳입니다.

중첩된 타입과 그 포함된 타입에 대해서도 일반적인 접근성 규칙이 적용됩니다. 중첩된 타입은 자신을 포함하는 클래스(containing class)의 인스턴스 변수(필드, 속성, 메소드)를 액세스할 수 있지만, 그러기 위해서는 객체 참조를 가지고 있어야 합니다. 중첩된 타입은 클래스 필드, 클래스 속성, 클래스 정적 메소드를 객체 참조 없이 액세스할 수 있지만, 일반적인 델파이 가시성 규칙이 적용됩니다.

중첩된 타입은 자신을 포함하는 클래스의 크기를 증가시키지 않습니다. 포함하는 클래스의 인스턴스를 생성하더라도 중첩된 타입의 인스턴스도 함께 생성되는 것은 아닙니다. 중첩된 타입은 자신의 선언 문맥에 의해서만 자신을 포함하는 클래스와 연관됩니다.

중첩된 클래스의 선언과 액세스

다음의 예는 중첩 클래스의 필드와 메소드를 선언하고 액세스하는 방법을 보여줍니다.

```
type
  TOuterClass = class
  strict private
    myField: Integer;

  public
    type
      TInnerClass = class
      public
        myInnerField: Integer;
        procedure innerProc;
      end;

    procedure outerProc;
  end;
```

안쪽 클래스의 innerProc 메소드를 구현하려면, 메소드 이름을 바깥 클래스의 이름으로 한정해야 합니다. 예를 들면 다음과 같습니다.

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

중첩된 타입의 멤버를 액세스하려면 일반적인 클래스 멤버를 액세스할 때처럼 마침표(.) 표기를 사용합니다. 예를 들면 다음과 같습니다.

```
var
   x: TOuterClass;
   y: TOuterClass.TInnerClass;

begin
   x := TOuterClass.Create;
   x.outerProc;
   ...
   y := TOuterClass.TInnerClass.Create;
   y.innerProc;
```

중첩된 상수

중첩된 타입 섹션과 같은 방식으로 상수도 클래스 타입 안에 선언할 수 있습니다. 상수 영역의 끝은 중첩된 타입 영역과 같은 토큰(예약어나 가시성 지정자)이 처음 나타나는 곳입니다. 타입 지정 상수는 지원되지 않으므로, Currency나 TDateTime 같은 값 타입의 중첩 상수를 선언할 수 없습니다.

중첩 상수는 모든 단순 타입이 될 수 있습니다. 서수, 서수 부분범위(subrange), 열거형, 문자열, 실수 타입이 가능합니다.

다음의 코드는 중첩 상수의 선언을 보여줍니다.

```
type
  TMyClass = class
    const
      x = 12;
      y = TMyClass.x + 23;
    procedure Hello;
  private
```

```
    const
      s = 'A string constant';
  end;

begin
   writeln(TMyClass.y);    // y의 값, 35를 출력함
end.
```

클래스 참조

클래스의 인스턴스인 객체가 아닌 클래스 자체에서 작업이 실행되는 경우가 종종 있습니다. 예를 들어, 클래스 참조를 사용하여 생성자 메소드를 호출할 때 이런 경우가 발생합니다. 클래스 이름을 사용하면 항상 특정 클래스를 참조할 수 있지만 클래스를 값으로 갖는 변수나 파라미터를 선언해야 하는 경우에는 클래스 참조 타입이 필요합니다.

클래스 참조 타입

클래스 참조 타입(class-reference type)은 종종 메타클래스(metaclass)라고도 불리며, 그 문법은 다음과 같습니다.

```
class of type
```

여기서 *type*은 임의의 클래스 타입입니다. *type* 식별자 자체는 타입이 class of *type*인 값을 나타냅니다. type1이 type2의 조상인 경우, class of type2를 class of type1에 대입할 수 있습니다(대입 호환). 따라서,

```
type TClass = class of TObject;
var AnyObj: TClass;
```

이 문법은 모든 클래스에 대한 참조를 가질 수 있는 AnyObj라는 변수를 선언합니다. (클래스 참조 타입의 정의는 변수 선언이나 파라미터 목록에 직접 나타날 수 없습니다.) 모든 클래스 참조 타입의 변수에 nil 값을 지정할 수 있습니다.
클래스 참조 타입 사용의 사용 방법을 알려면 Classes 유닛에 있는 TCollection 생성자 선언을 참고하십시오.

```
type TCollectionItemClass = class of TCollectionItem;
  ...
constructor Create(ItemClass: TcollectionItemClass);
```

이 선언에서는 TCollection 인스턴스 객체를 만들려면 TCollectionItem에서 상속받는 클래스의 이름을 생성자에 전달해야 하도록 강제하고 있습니다.
클래스 참조 타입은 컴파일 시 실제 타입을 모르는 클래스 또는 객체에서 클래스 메소드나 가상 생성자를 호출할 때 유용합니다.

■ 생성자와 클래스 참조

클래스 참조 타입의 변수를 사용하여 생성자를 호출할 수 있습니다. 이 기능으로 컴파일 시에 타입을 모르는 객체를 생성하는 문법을 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

CreateControl 함수는 생성할 컨트롤 종류를 지정하는 클래스 참조 파라미터를 필요로 합니다. CreateControl 함수는 이 파라미터를 해당 클래스의 생성자를 호출하기 위해 사용합니다.
클래스 타입 식별자는 클래스 참조 값을 나타내므로, CreateControl에 대한 호출에서 인스턴스를 만들 클래스의 식별자를 지정할 수 있습니다. 예를 들면, 다음과 같습니다.

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

클래스 참조를 사용하여 호출되는 생성자는 보통 가상(virtual) 생성자입니다. 호출에 의해

실행되는 생성자의 구현은 클래스 참조의 런타임 타입 정보(RTTI)에 따라 결정됩니다.

클래스 연산자

클래스 메소드는 클래스 참조에 대해 동작합니다. 모든 클래스는 TObject로부터 ClassType 및 ClassParent라는 두 클래스 메소드를 상속합니다. 이 메소드들은 각각 한 객체의 클래스에 대한 참조와 한 객체의 직접 조상 클래스에 대한 참조를 리턴합니다. 두 메소드는 TClass 타입의 값을 리턴하는데(여기서 `TClass = class of TObject`), 이 리턴되는 TClass 타입 값은 더 특정한 타입으로 타입 캐스트될 수 있습니다. 또한 모든 클래스는 객체가 지정된 클래스의 자손인지 여부를 확인하는 InheritsFrom이라는 메소드도 상속받습니다. 이러한 메소드는 is 및 as 연산자에 의해 사용되며, 이들 메소드들을 직접 호출하는 경우는 거의 없습니다.

■ is 연산자

is 연산자는 동적 타입 검사를 수행하며, 객체의 실제 런타임 클래스를 확인하는 데 사용됩니다. 다음 표현식에서,

```
object is class
```

이 문법은 *object*가 *class*나 그 자손 클래스의 인스턴스인 경우 True를 리턴하고 그렇지 않으면 False를 리턴합니다. *object*가 *nil*이면 결과는 *False*가 됩니다. *object*의 선언된 타입이 *class*와 관련이 없으면, 즉 타입이 다르고 *object*가 *class*의 자손 또는 조상이 아니면, 컴파일 에러가 발생합니다. 예를 들면, 다음과 같습니다.

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

이 문장은 변수가 참조하는 객체가 TEdit나 그 자손의 인스턴스인지를 먼저 확인한 후, 변수를 TEdit로 타입 캐스트합니다.

■ as 연산자

as 연산자는 확인된 타입 캐스트(checked typecast)를 실행합니다. 다음 표현식에서,

```
object as class
```

이 문법은 *object*와 동일한 객체에 대한 참조를 리턴하지만, *class*에서 제공한 타입으로 리턴합니다. 런타임 시 *object*는 *class*나 그 자손 클래스의 인스턴스 혹은 nil이어야 합니다. 그렇지 않을 경우 예외가 발생합니다. *object*의 선언된 타입이 *class*와 관련이 없으면, 즉 타입이 다르고 *object*가 *class*의 자손 또는 조상이 아니면, 컴파일 에러가 발생합니다. 예를 들면, 다음과 같습니다.

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

연산자 우선 순위 규칙상 as 타입 캐스트에 괄호를 사용해야 하는 경우가 많습니다. 예를 들면, 다음과 같습니다.

```
(Sender as TButton).Caption := '&Ok';
```

예외(Exception)

에러나 다른 사건이 프로그램의 정상적인 실행을 방해하면 예외(exception)가 발생합니다. 예외는 예외 핸들러로 제어를 넘김으로써 정상적인 프로그램 로직과 에러 처리를 분리할 수 있도록 해줍니다. 예외는 객체이므로 상속을 통해 계층적으로 그룹화할 수 있으며, 기존 코드에 영향을 주지 않고 새로운 예외를 만들 수 있습니다. 예외는 에러 메시지와 같은 정보를 예외가 발생한 위치로부터 처리된 위치로 전달할 수 있습니다.
애플리케이션에서 SysUtils 유닛을 uses 하면 대부분의 런타임 에러는 자동으로 예외로 변환됩니다. 메모리 부족, 0으로 나누기, 일반 보호 에러 등 처리하지 않으면 애플리케이션을 종료시키는 많은 에러들을 잡아 처리할 수 있습니다.

예외를 사용하는 시기

예외는 프로그램을 중단시키지 않고 불편한 조건문 없이 런타임 에러를 우아하게 처리할 수 있게 해줍니다. 예외 처리를 이용하는 방식은 코드/데이터 크기 및 런타임 성능 면에서는 불

리합니다.

거의 모든 이유로 예외를 발생시킬 수 있으며 어떤 코드 블럭이든 try…except 또는 try...finally 문을 사용하여 보호할 수 있지만, 실제로 특별한 경우에만 예외 처리를 사용하는 것이 가장 좋습니다.

예외 처리는 드물게 발생하거나 판단하기 어렵지만 그 결과는 대단히 심각한(애플리케이션의 갑작스런 중지 등) 에러에 적합합니다. if...then 문으로 테스트하기 복잡하거나 어려운 경우, 운영 체제에서 발생한 예외를 처리해야 하는 경우, 소스 코드를 직접 관리할 수 없는 루틴에서 발생하는 예외를 처리해야 하는 경우 등입니다. 예외는 하드웨어, 메모리, I/O, 운영 체제 에러에서 흔히 사용됩니다.

조건문이 에러를 테스트하기 위해 가장 좋은 방법인 경우가 종종 있습니다. 예를 들어, 파일을 열기 전에 파일이 존재하는지 확인하고 싶은 경우를 가정해 보십시오. 다음과 같은 방법으로 할 수도 있습니다.

```
try
  AssignFile(F, FileName);
  Reset(F); // 파일이 발견되지 않으면 EInOutError 예외를 발생
except
  on Exception do ...
end;
```

그러나 다음의 코드를 사용하여 예외 처리의 오버헤드를 피할 수도 있습니다.

```
if FileExists(FileName) then // 파일이 발견되지 않으면 False를 리턴: 예외를 발생시키지 않음
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

어서션(assertion)은 소스 코드의 어느 곳에서나 부울 조건을 테스트할 수 있는 또다른 방법입니다. Assert 문이 실패하면 런타임 에러를 내고 프로그램이 중단되거나(SysUtils 유닛을 uses 하는 경우) EAssertionFailed 예외가 발생합니다. 어서션은 발생할 것으로 생각하지 못하는 조건을 테스트하기 위해서만 사용되어야 합니다. 자세한 내용은 Assert 표준 프로시저의 온라인 헬프를 참조하십시오.

예외 타입의 선언

예외 타입은 다른 클래스와 마찬가지로 선언됩니다. 실제로 어떤 클래스의 인스턴스든 예외

로 사용할 수 있지만, SysUtils에 정의된 Exception 클래스에서 예외 클래스를 파생하는 것이 바람직합니다.
상속을 이용하여 예외를 패밀리로 그룹화할 수 있습니다. 예를 들어, SysUtils에 있는 다음의 선언은 수학적인 에러의 예외 타입 패밀리를 정의합니다.

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

위와 같이 선언한 후에는 EInvalidOp, EZeroDivide, EOverflow 및 EUnderflow를 처리하는 단일 EMathError 예외 핸들러를 정의할 수 있습니다.
예외 클래스는 경우에 따라 에러에 대한 추가 정보를 전달하는 필드, 메소드, 속성을 정의합니다. 예를 들면, 다음과 같습니다.

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

예외의 발생과 처리

예외 객체를 발생(raise)시키려면 raise 문에서 예외 클래스의 인스턴스를 사용합니다. 예를 들면, 다음과 같습니다.

```
raise EMathError.Create;
```

일반적으로 raise 문의 형태는 다음과 같습니다.

```
raise object at address
```

여기서 *object*와 at *address*는 모두 옵션입니다. *address*가 지정될 경우 그것은 포인터 타입으로 리턴되는 모든 형태의 표현식일 수 있지만, 일반적으로는 프로시저나 함수에 대한 포인터입니다. 예를 들면 다음과 같습니다.

```
raise Exception.Create('Missing parameter') at @MyFunction;
```

에러가 실제로 발생한 위치보다 앞선 스택의 위치에서 예외를 발생시키려면 이 옵션을 사용합니다. 발생된 예외(raise 문에서 참조된 예외)는 특정 예외 처리 로직에 의해 처리됩니다. raise 문은 일반적인 방법으로 제어를 돌려주지 않습니다. 대신, 해당 종류의 예외를 처리할 수 있는 가장 안쪽의 예외 핸들러로 제어를 넘깁니다. (가장 안쪽의 핸들러란 가장 최근에 들어왔지만 아직 빠져나가지 않은 try...except 블럭 내에 있는 핸들러입니다.)
예를 들어, 아래 함수는 문자열을 정수로 변환하는데, 결과 값이 지정된 범위를 벗어나는 경우 ERangeError 예외를 발생시킵니다.

Note
유닛의 initialization 섹션에서 예외가 발생하면 의도한 결과가 생기지 않을 수도 있습니다. 일반적인 예외 지원은 SysUtils 유닛에서 제공되는데, 그 지원들이 사용 가능하려면 초기화되어야만 합니다. 초기화 중에 예외가 발생하게 되면, SysUtils를 비롯한 초기화되었던 모든 유닛이 finalize 되고 예외가 재발생합니다. 그런 다음 예외가 포착되고 처리하는데, 일반적으로 프로그램은 중단되게 됩니다. 비슷하게, 유닛의 finalization 섹션에서 예외를 발생시키면, 예외가 발생했을 때 SysUtils가 이미 finalize 된 상태이면 의도했던 결과가 되지 않을 수도 있습니다.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S); // StrToInt는 SysUtils에 선언되어 있음
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt('%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

raise 문에서 CreateFmt 메소드가 호출되었다는 점에 유의하십시오. Exception 클래스와 그 자손은 예외 메시지와 컨텍스트 ID를 만들 수 있는 다른 방법을 제공하는 특별한 생성자들을 가지고 있습니다. 자세한 내용은 온라인 헬프를 참조하십시오.
발생한 예외는 처리되고 난 후에는 자동으로 소멸됩니다. 발생한 예외를 절대 수동으로 파괴(Free) 하지 마십시오.

■ try...except 문

예외는 try...except 문 내에서 처리됩니다. 예를 들면, 다음과 같습니다.

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

이 문에서는 Y를 Z로 나눕니다. 그러나 EZeroDivide(0으로 나누기) 예외가 발생하게 되면

HandleZeroDivide 라는 루틴을 호출합니다.
try...except 문의 문법은 다음과 같습니다.

```
try statements except exceptionBlock end
```

여기서 *statements*는 세미콜론으로 구분되는 일련의 문장들이며, *exceptionBlock*은 다음 중 하나입니다.

- 일련의 다른 문장들
- 일련의 예외 핸들러들. 옵션으로 다음의 절이 따라올 수 있습니다.

```
else statements
```

예외 핸들러는 다음과 같은 형태를 가집니다.

```
on identifier: type do statement
```

여기서 *identifier*:은 옵션으로서 *identifier*는 유효한 식별자이고, *type*은 예외를 나타내는데 사용되는 타입이며, *statement*는 임의의 문장입니다.
try...except 문은 try 다음에 있는 첫 *statements* 리스트의 문장들을 실행합니다. 예외가 발생하지 않으면 예외 블럭(*exceptionBlock*)은 무시되고 프로그램의 다음 부분으로 제어가 넘어갑니다.
첫번째 *statements* 리스트에 포함된 *raise* 문, 혹은 *statements* 리스트에서 호출된 프로시저나 함수에 의해 첫번째 *statements* 리스트의 실행 중 예외가 발생하면, 예외를 "처리"하려는 시도가 일어납니다.

- *exceptionBlock*의 핸들러 중 해당 예외와 일치하는 것이 있을 경우, 그중 첫 번째 핸들러로 제어가 넘어갑니다. 핸들러에 있는 타입이 해당 예외나 그 조상일 때, 그 예외 핸들러와 예외가 "일치"하는 것입니다.
- 일치하는 핸들러가 없으면, else 절이 있으면 그 절의 *statements*로 제어가 넘어갑니다.
- 예외 블럭이 예외 핸들러가 없는 일련의 문장들인 경우, 그 리스트의 첫 번째 문장으로 제어가 넘어갑니다.

위 조건 중 해당하는 것이 없는 경우, 가장 최근에 진입해서 아직 빠져나가지 않은 try...except 문의 예외 블럭에서 찾기를 계속합니다. 적절한 핸들러, else 절 또는 문장 목록이 없는 경우에는 그 다음의 가장 최근에 진입해서 아직 빠져나가지 않은 try...except 문을 검색해나가고, 이런 식으로 계속 검색해 나갑니다. 가장 외부 try...except 문에 도달해도 예외가 처리되지 않는 경우, 프로그램이 종료됩니다.

예외가 처리될 때, 처리가 일어나는 try...except 문이 포함된 프로시저나 함수로 스택을 역으로 추적하며, 실행될 예외 핸들러나 else 절, 문장 목록으로 제어가 전달됩니다. 이러한 과정에서 예외가 처리되는 try...except 문에 들어간 이후의 모든 프로시저와 함수 호출은 취소됩니다. 그런 다음, 예외 객체는 Destroy 소멸자가 호출되어 자동으로 소멸되며, 제어는 try...except 문의 다음 문장으로 전달됩니다. (Exit, Break, Continue 표준 프로시저를 호출하면 제어가 예외 핸들러를 빠져나가게 되며, 그런 경우에도 예외 객체는 자동으로 소멸됩니다.)

아래 예제에서, 첫 번째 예외 핸들러는 0으로 나누기 예외를 처리하고, 두 번째 예외 핸들러는 오버플로우 예외를 처리하며, 마지막 예외 핸들러는 다른 모든 수학 예외를 처리합니다. EMathError를 마지막에 배치한 것은, 이 클래스가 다른 두 예외 클래스의 조상이기 때문입니다. 이 클래스를 먼저 배치할 경우 다른 두 핸들러는 어떤 경우에도 호출되지 않습니다.

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

예외 핸들러는 예외 클래스의 이름 앞에 식별자를 지정할 수 있습니다. 이는 식별자가 on...do에 뒤의 문장을 실행하는 동안 예외 객체를 나타내는 식별자를 선언합니다. 이 식별자의 유효 범위(scope)는 해당 문장으로 제한됩니다. 예를 들면, 다음과 같습니다.

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

```
try
  ...
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

예외 블록에 else 절이 있는 경우, else 절은 해당 블럭의 예외 핸들러들에 의해 처리되지 않은 예외를 처리합니다. 예를 들면, 다음과 같습니다.

```
try
  ...
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

여기서 else 절은 EMathError가 아닌 모든 예외를 처리합니다.
예외 핸들러가 없이 문장 리스트로만 구성된 예외 블럭은 모든 예외를 처리합니다. 예를 들면, 다음과 같습니다.

```
try
  ...
except
  HandleException;
end;
```

여기서 HandleException 루틴은 try와 except 사이의 문을 실행한 결과로 발생하는 모든 예외를 처리합니다.

■ 예외의 재발생

예외 블록에서 예약어 raise가 뒤에 예외 객체 참조 없이 나오는 경우, 블럭에 의해 처리되고 있던 예외를 발생합니다. 이를 통해 예외 핸들러가 최소한의 방법으로 에러에 대응한 후, 예외를 재발생(re-raise)시킬 수 있습니다. 예외 재발생은 프로시저나 함수가 예외 발생 후 완전히 예외를 처리할 수 없지만 클린업 작업을 해야 하는 경우 유용합니다.
예를 들어, GetFileList 함수는 TStringList 객체를 생성한 다음 해당 객체를 지정한 서치

패스(Search Path)에 맞는 파일 이름들을 담습니다.

```
function GetFileList(const Path: string): TStringList;
var
  I:Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
    begin
      Result.Add(SearchRec.Name);
      I := FindNext(SearchRec);
    end;
  except
    Result.Free;
    raise;
  end;
end;
```

GetFileList는 TStringList 객체를 만든 다음, FindFirst와 FindNext 함수(SysUtils에서 정의됨)를 사용하여 TStringList 객체에 데이터를 넣습니다. 서치 패스가 잘못되었거나 TStringList에 추가할 메모리가 부족하거나 해서 이 작업이 실패하는 경우, 호출자는 TStringList의 존재 여부를 알지 못하므로 GetFileList 가 TStringList를 파괴해야 합니다. 이러한 이유로 TStringList에 데이터를 추가하는 작업을 try...except 문 안에서 실행되도록 한 것입니다. 예외가 발생하게 되면 문장의 예외 블럭은 TStringList를 파괴한 후 예외를 재발생시킵니다.

■ 중첩된 예외

예외 핸들러에서 실행된 코드에서도 자체적으로 예외를 발생시키고 처리할 수 있습니다. 이러한 예외가 예외 핸들러 내에서 처리된다면 원래의 예외에 영향을 주지 않습니다. 그러나 일단 예외 핸들러에서 발생한 새로운 예외가 그 핸들러를 벗어나게 되면 원래의 예외는 소실됩니다. 아래의 Tan 함수가 이를 보여줍니다.

```
type
  ETrigError = class(EMathError);

function Tan(X: Extended): Extended;
```

```
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
  end;
end;
```

Tan의 실행 중에 EMathError 예외가 발생하면 예외 핸들러는 ETrigError를 발생시킵니다. Tan에서는 ETrigError에 대한 핸들러를 제공하지 않으므로, 원래 핸들러를 벗어게 되고 따라서 EMathError 예외는 소멸됩니다. 호출자에게는 Tan 함수가 ETrigError 예외를 발생시킨 것처럼 보이게 됩니다.

■ try...finally 문

때로는 동작의 특정 부분이 완료되었는지, 동작이 예외에 의해 중단되었는지 여부를 확인하고자 할 경우가 있습니다. 예를 들어, 한 루틴이 어떤 리소스에 대한 관리를 맡게 되면, 루틴이 정상적으로 종료되었든 아니든 관계 없이 리소스가 확실히 해제되도록 하는 것이 중요합니다. 이러한 상황에서 try...finally 문을 사용할 수 있습니다.

다음 예제는 파일을 열어 처리하는 코드가 실행 중에 에러가 발생하더라도 어떻게 파일을 최종적으로 닫을 수 있는지를 보여줍니다.

```
Reset(F);
try
  ...  // F 파일 관련 작업
finally
  CloseFile(F);
end;
```

try...finally 문의 문법은 다음과 같습니다.

```
try statementList1 finally statementList2 end
```

여기서 각 *statementList*는 세미콜론으로 구분된 일련의 문장입니다. 먼저 try...finally 문은 *statementList1* (try 절)의 문장을 실행합니다. *statementList1*이 예외없이 종료되면 *statementList2* (finally 절)가 실행됩니다. *statementList1*의 실행 중 예외가 발생하면

*statementList2*로 제어가 넘어가고, *statementList2*가 실행을 끝내면 예외가 재발생합니다. Exit, Break 또는 Continue 프로시저를 호출하여 제어가 statementList1에서 빠져나오더라도 *statementList2*는 자동으로 실행됩니다. 따라서 finally 절은 try 절이 어떻게 종료되든지 상관없이 항상 실행됩니다.

예외가 발생했지만 finally 절에서 처리되지 않으면 try...finally 문 외부로 예외가 전달되며 try 절에서 발생했던 예외는 소실됩니다. 그러므로 finally 절은 지역적으로 발생한 예외를 모두 처리하여 다른 예외의 전달을 방해하지 않아야 합니다.

■ 표준 예외 클래스 및 루틴

SysUtils 유닛과 System 유닛에는 ExceptObject, ExceptAddr, ShowException 등을 포함한 예외 처리를 위한 몇몇 표준 루틴을 선언되어 있습니다. SysUtils, System 및 다른 유닛들에는 또한 Exception에서 파생된(OutlineError 제외) 수십 개의 예외 클래스들이 포함되어 있습니다.

Exception 클래스는 에러 설명을 전달하는 Message 속성과 온라인 헬프 연동을 위한 컨텍스트 ID를 전달하는 HelpContext라는 속성을 가지고 있습니다. 이 클래스는 다른 방법으로 에러 설명과 컨텍스트 ID를 지정할 수 있도록 해주는 다양한 생성자 메소드도 정의합니다.