



솔루션 문서

---

# 새 **Delphi** 코딩 스타일과 아키텍처

Delphi 2009의 언어 기능 리뷰

2008년 12월

---

## **Corporate Headquarters**

100 California Street, 12th Floor  
San Francisco, California 94111

## **Asia-Pacific Headquarters**

L7. 313 La Trobe Street  
Melbourne VIC 3000  
Australia

## **DEVGEAR**

서울특별시 서초구  
반포동 743-14  
데브기어 4층

## 소개: DELPHI 언어

Object Pascal 로 더 잘 알려진 Delphi 언어는 단일 상속 및 개체 참조 모델이 특징인 강력한 최신 타입 확인 및 개체 지향 언어입니다. 최근 몇 년간 Delphi 언어는 레코드 메소드, 레코드를 위한 운영자 오버로드, 클래스 데이터, 중첩 타입, 봉인된 클래스, 최종 메소드 및 기타 관련 기능 등을 사용하여 향상되었습니다. 가장 놀라운 확장은 새로운 메소드를 기존 클래스에 추가하거나 일부 기존 메소드를 대체하는 데 사용하는 기술인 클래스 도우미의 도입일 것입니다.

그러나 Delphi 2009 에서 컴파일러에 추가된 새 기능은 훨씬 더 중요합니다. Unicode 를 지원하기 위한 문자열 타입에 대한 확장 이외에도 Delphi 의 최근 버전에는 제네릭 데이터 타입, 익명 메소드 및 여러 개의 기타 “ 사소하지만 ” 흥미로운 기능이 적용되었습니다.

## 제네릭 소개

제네릭 클래스의 첫 예로서 저는 키-값 쌍 데이터 구조를 구현했습니다. 아래에 표시된 첫 번째 코드 조각은 값을 보유하는 데 사용되는 개체와 함께, 일반적으로 작성되는 데이터 구조를 보여줍니다.

```
type
  TKeyVal ue = class
  private
    FKey: string; FVal ue: TObject;
    procedure SetKey(const Val ue: string);
    procedure SetVal ue(const Val ue: TObject);
  public
    property Key: string read FKey write SetKey;
    property Val ue: TObject read FVal ue write SetVal ue;
  end;
```

이 클래스를 사용하려면 다음 조각에 표시된 대로 개체를 생성하고, 해당 개체의 키와 값을 설정하여 사용할 수 있습니다.

```
// FormCreate
kv := TKeyVal ue.Create;

// Button1Cl ick kv.Key := 'mykey'; kv.Val ue := Sender;

// Button2Cl ick
kv.Val ue := sel f; // 폼

// Button3Cl ick
ShowMessage ('[' + kv.Key + ', ' +
kv.Val ue.Cl assName + ']');
```

제네릭을 사용하면 값에 대한 폭넓은 정의를 사용할 수 있지만 그것이 핵심은 아닙니다. 앞으로 다루겠지만 큰 차이점은 키-값 제네릭 클래스를 인스턴트화하면 해당 클래스가 지정된 데이터 타입에 연결된 특정 클래스가 된다는 것입니다. 이렇게 하면 코드 타입을 더 안전하게 만들 수 있지만 개인적으로는 기존에 사용하던 방식을 그대로 고수합니다. 이제 제네릭 클래스를 정의하는

데 사용되는 구문에 대해 알아보도록 하겠습니다.

```

type
  TKeyVal ue<T> = class
  private
    FKey: string; FVal ue: T;
    procedure SetKey(const Val ue: string);
    procedure SetVal ue(const Val ue: T);
  public
    property Key: string read FKey write SetKey;
    property Val ue: T read FVal ue write SetVal ue;
  end;
    
```

이 클래스 정의에는 자리표시자 T에 의해 표시되는 한 개의 지정되지 않은 타입이 있습니다. 제네릭 TKeyVal ue<T> 클래스는 지정되지 않은 해당 타입을 속성 값 필드 타입과 setter 메소드 매개 변수로 사용합니다. 그러나 메소드는 일반적인 방법으로 정의되며, 메소드가 제네릭 타입과 관련이 있다 하더라도 해당 정의에는 제네릭 타입을 포함한 클래스의 전체 이름이 포함됩니다.

```

procedure TKeyVal ue<T>. SetKey(const Val ue: string);
begin
  FKey := Val ue;
end;

procedure TKeyVal ue<T>. SetVal ue(const Val ue: T);
begin
  FVal ue := Val ue;
end;
    
```

대신, 클래스를 사용하려면 제네릭 타입의 실제 값을 제공하여 전체 클래스를 정규화해야 합니다. 예를 들어 이제 아래와 같이 작성하여 키-값 개체 호스팅 버튼을 값으로 선언할 수 있습니다.

```

kv: TKeyVal ue<TButton>;
    
```

또한 전체 이름이 실제 타입 이름이기 때문에 인스턴스 작성 시 해당 전체 이름도 필요합니다. (인스턴스화되지 않은 제네릭 타입 이름은 타입 구성 메커니즘과 유사합니다.)

키-값 쌍에 대해 특정 타입의 값을 사용하면, TButton 개체 또는 파생 개체를 키 값 쌍에 추가할 수 있고 추출된 개체의 다양한 메소드를 사용할 수 있기 때문에 코드가 훨씬 강력해집니다. 다음은 일부 코드 조각의 예입니다.

```

// FormCreate
kv := TKeyVal ue<TButton>. Create;

// Button1Click kv.Key := 'mykey';
kv.Val ue := Sender as TButton;

// Button2Click
kv.Val ue := Sender as TButton; // was "sel f"

// Button3Click
ShowMessage ('[' + kv.Key + ', ' + kv.Val ue.Name + ']');
    
```

코드의 이전 버전에서는 제네릭 개체 할당 시 버튼 또는 양식을 추가할 수 있었습니다. 이제는 버튼만 추가할 수 있으며, 이것은 컴파일러에 의해 강제로 적용된 규칙입니다. 마찬가지로, 출력에는 제네릭 kv.Val ue.ClassName이 아니라 구성 요소 이름 또는 TButton의 다른 속성을 사용할 수 있습니다.

물론 키-값 쌍을 다음과 같이 선언하여 원본 프로그램을 모방할 수도 있습니다.

```
    kvo: TKeyVal ue<TObject>;
```

제네릭 키-값 쌍 클래스의 이번 버전에서는 개체를 값으로 추가할 수 있습니다. 그러나 추출한 개체를 특정 타입에 캐스트하지 않으면 추출한 개체를 많이 활용할 수 없습니다. 균형을 잘 맞추려면 다음과 같이 특정 버튼과 개체 중간의 어떤 것을 찾고 값이 구성 요소가 되도록 요청할 수 있습니다.

```
    kvc: TKeyVal ue<TComponent>;
```

마지막으로 다음과 같이 개체 값이 아니라 보다 일반적인 정수 값을 저장하는 제네릭 키-값 쌍 클래스의 인스턴스를 생성할 수 있습니다.

```
    kvi: TKeyVal ue<Integer>;
```

## 제네릭의 타입 규칙

제네릭 타입의 인스턴스를 선언하면 이 타입은 수반되는 모든 작업에서 컴파일러가 강제로 적용하는 특정 버전을 사용하게 됩니다. 따라서, 다음과 같은 제네릭 클래스를 가지는 경우

```
type
  TSimpleGeneric<T> = class
    Value: T;
  end;
```

지정된 타입으로 특정 개체를 선언하기 때문에 다른 타입을 값 필드에 할당할 수 없습니다. 다음 두 개의 개체가 지정되면 아래의 일부 할당은 틀릴 수 있습니다.

```
var
  sg1: TSimpleGeneric<string>;
  sg2: TSimpleGeneric<Integer>;
begin
  sg1 := TSimpleGeneric<string>.Create;
  sg2 := TSimpleGeneric<Integer>.Create;

  sg1.Value := 'foo';
  sg1.Value := 10; // Error
  // E2010 Incompatible types: 'string' and 'Integer'

  sg2.Value := 'foo'; // Error
  // E2010 Incompatible types: 'Integer' and 'string'
  sg2.Value := 10;
```

특정 타입을 제네릭 선언으로 정의하면 **Object Pascal**과 같이 강력한 타입의 언어에서 기대하는 바처럼 컴파일러가 강제로 적용하게 됩니다. 타입 확인도 전체적으로 제네릭 개체에 적용할 수 있습니다. 개체에 대한 제네릭 매개 변수를 지정하면 다른 호환되지 않는 타입 인스턴스에 기반한 유사 제네릭 타입을 개체에 할당할 수 없습니다. 잘 이해가 안되는 경우 아래 예를 보면 명확하게 이해할 수 있습니다.

```
sg1 := TSimpleGeneric<Integer>.Create; // 에러
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

타입 호환성 규칙은 타입 이름이 아니라 구조별로 정해지지만 제네릭 타입 인스턴스에 다른

호환되지 않는 타입 인스턴스를 할당할 수는 없습니다.

## DELPHI의 제네릭

앞의 예에서는 Delphi 3이 인터페이스를 도입한 이후 Object Pascal 언어로의 가장 중요한 확장 중 하나인 제네릭 클래스를 정의하고 사용하는 방법을 살펴보았습니다. 이제 꽤 복잡하지만 매우 중요한 그 기술을 심도 있게 다루기 전에 예를 통해 그 기능을 설명하도록 하겠습니다. 언어적인 측면에서 제네릭을 설명한 다음 이 기술을 언어에 추가한 주요 이유 중 하나인 제네릭 컨테이너 클래스의 사용과 정의를 포함한 추가 예로 다시 돌아가도록 하겠습니다.

Delphi 2009에서 클래스를 정의할 때 아래와 같이 꺾쇠 괄호 내의 기타 “매개 변수”를 추가하여 이후에 제공되는 타입의 위치를 유지할 수 있다는 것을 이미 살펴보았습니다.

```
type
  TMyClass <T> = class
    ...
  end;
```

제네릭 타입은 필드 타입(앞의 예에서 사용함), 속성의 타입, 함수의 매개 변수 또는 반환 값의 타입 및 기타 타입으로 사용할 수 있습니다.

제네릭 타입이 결과 즉 매개 변수로서만 사용되거나 클래스의 선언이 아닌 일부 메소드의 정의에만 사용되는 경우가 있으므로 로컬 필드(또는 배열)에 대한 타입을 반드시 사용해야 하는 것은 아닙니다.

확장 또는 제네릭 타입 선언의 이 형식을 클래스 및 레코드에 사용할 수 있습니다. (최신 버전의 Delphi에서는 레코드도 메소드와 오버로드된 연산자를 포함할 수 있습니다.) C++과 달리 제네릭 글로벌 함수를 선언할 수는 없지만 단일 클래스 메소드로 제네릭 클래스를 선언할 수는 있으며 이는 거의 같습니다.

다른 정적 언어와 마찬가지로 Delphi의 제네릭 구현은 런타임 프레임워크에 기반하지 않고 컴파일러와 링커로 처리되기 때문에 런타임 메커니즘과는 거의 무관합니다. 런타임 시 바인딩되는 가상 함수 호출과 달리 템플릿 메소드는 사용자가 인스턴스화하는 각 템플릿 타입별로 한 번 생성되며 컴파일 시간 동안 생성됩니다. 이러한 접근 방법에는 단점도 있지만 긍정적인 측면에서 보면 제네릭 클래스가 일반 클래스만큼 또는 런타임 범주에 대한 필요성을 줄일 수 있을 정도로 보다 효율적이라는 것을 의미합니다.

## 제네릭 타입 함수

지금까지 살펴본 제네릭 타입 정의의 가장 큰 문제는 제네릭 타입의 개체에 대해 할 수 있는 작업이 거의 없다는 것입니다. 이러한 한계를 극복하기 위해 사용할 수 있는 두 가지의 기술이 있습니다. 첫 번째는 제네릭 타입을 특별히 지원하는 런타임 라이브러리의 일부 특수 함수를 활용하는 것입니다. 두 번째는 훨씬 더 강력한 방법으로서 사용할 수 있는 타입에 제약 조건을 포함한 제네릭 클래스를 정의하는 것입니다.

이 섹션에서는 첫 번째 부분에, 다음 섹션에서는 제약 조건에 초점을 맞춰 설명하도록 하겠습니다.

이미 설명한 바와 같이 제네릭 타입 정의의 매개 변수 타입(T)에서 동작하도록 특별히 수정된 2개의 기존 함수와 1개의 새로운 함수가 있습니다.

- **Default(T)**는 새로운 함수로 현재 타입에 대해 빈 또는 “ 0(영)인 값” 이나 **null** 값을 반환하며, 0(영), 빈 문자열, **nil** 등의 값이 될 수 있습니다.
- **TypeInfo(T)** 는 제네릭 타입의 현재 버전에 대한 런타임 정보에 포인터를 반환합니다.
- **SizeOf(T)**는 타입의 메모리 크기를 바이트로 반환합니다.

다음 예는 활성 중인 3개의 제네릭 타입 함수를 표시하는 제네릭 클래스입니다.

```

type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure Zero;
    function GetDataSize: Integer;
    function GetDataName: string;
  end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf (T);
end;

function TSampleClass<T>.GetDataName: string;
begin
  Result := GetTypeName (TypeInfo (T));
end;

procedure TSampleClass<T>.Zero;
begin
  data := Default (T);
end;
    
```

**GetDataName** method에서 저는 직접 데이터 구조에 액세스하지 않고 **GetTypeName** function(또는 **TypeInfo** 단위)를 사용했습니다. 이 방법을 사용하면 타입 이름이 있는 인코딩된 **ShortString** 값에서 적합한 **UTF-8** 변환을 실행할 수 있습니다.

위에 나온 선언이 제공된 경우 3개의 다른 제네릭 타입 인스턴스에서 자체적으로 3번 반복되는 다음의 테스트 코드를 컴파일할 수 있습니다. 여기서는 반복된 코드를 생략했지만 데이터 필드에 액세스하는 데 사용되는 명령문은 실제 타입에 따라 달라지므로 남겨두었습니다.

```

var
  t1: TSampleClass<Integer>;
  t2: TSampleClass<string>;
  t3: TSampleClass<double>;
begin
  t1 := TSampleClass<Integer>.Create;
  t1.Zero;
  Log (' TSampleClass<Integer> ');
  Log (' data: ' + IntToStr (t1.data)); Log (' type: ' + t1.GetDataName);
  Log (' si ze: ' + IntToStr (t1.GetDataSi ze));

  t2 := TSampleClass<string>.Create;
    
```

```

...
Log (' data: ' + t2. data);

t3 := TSampI eCl ass<doubl e>. Create;
...
Log (' data: ' + Fl oatToStr (t3. data));

```

이 코드를 실행하면 다음과 같은 출력이 생성됩니다.

```

TSampI eCl ass<I nteger>
data: 0
type: I nteger si ze: 4
TSampI eCl ass<stri ng>
data:
type: stri ng si ze: 4
TSampI eCl ass<doubl e>
data: 0
type: Doubl e si ze: 8

```

이상하게 들리겠지만 제네릭 클래스 내용 이외의 특정 타입에서도 제네릭 타입 함수를 사용할 수 있습니다. 예를 들어 다음과 같이 작성할 수 있습니다.

```

var
  I: I nteger;
  s: stri ng;
begin
  I := Defaul t (I nteger);
  Log (' Defaul t I nteger' : + I ntToStr (I));

  s := Defaul t (stri ng);
  Log (' Defaul t Stri ng' : + s);

  Log (' TypeI nfo Stri ng' : + GetTypeI nfo (stri ng));

```

Default are에 대한 호출은 Delphi 2009의 새로운 기능이지만(템플릿 외에서는 전혀 유용하지 않음) 마지막의 TypeInfo에 대한 호출은 이전 버전의 Delphi에서 이미 가능했습니다. 다음은 호출에 대한 간단한 출력입니다.

```

Defaul t I nteger: 0
Defaul t Stri ng:
TypeI nfo Stri ng: stri ng

```

## 제네릭 제약 조건

위에서 살펴본 바와 같이 제네릭 타입 값을 통해 제네릭 클래스의 메소드로 할 수 있는 작업은 매우 제한되어 있습니다. 제네릭 타입 값을 차례로 분배(즉, 할당)하고 위에서 방금 전에 설명한 제네릭 타입 함수가 허용하는 제한된 작업을 수행할 수 있습니다.

제네릭 타입 클래스의 실제 작업 중 일부를 수행하기 위해서 일반적으로 제약 조건을 배치합니다. 예를 들어 제네릭 타입을 클래스로 제한하면 컴파일러를 사용하여 클래스의 모든 TObject 메소드를 호출할 수 있습니다. 클래스를 지정된 계층에 속하도록 하거나 특정 인터페이스를 실행하도록 제약할 수 있습니다.

## 클래스 제약 조건

사용할 수 있는 가장 간단한 제약 조건은 클래스 제약 조건입니다. 클래스 제약 조건을 사용하려면 다음과 같은 제네릭 타입을 선언해야 합니다.

```
type
  TSampI eCl ass <T: cl ass> = cl ass
```

클래스 제약 조건을 지정하여 개체 타입만 제네릭 타입으로 사용할 수 있음을 지정합니다.

다음의 선언을 사용하여 클래스 제약 조건을 실행할 수 있습니다.

```
type
  TSampI eCl ass <T: cl ass> = cl ass
private
  data: T;
public
  procedure One;
  function ReadT: T;
  procedure SetT (t: T);
end;
```

다음에 표시된 처음 2개의 인스턴스(3번째는 아님)를 생성할 수 있습니다.

```
sampI e1: TSampI eCl ass<TButton>;
sampI e2: TSampI eCl ass<TStrings>;
sampI e3: TSampI eCl ass<Integer>; // Error
```

이 마지막 선언으로 발생하는 컴파일러 에러는 다음과 같습니다.

```
E2511 Type parameter 'T' must be a class type
```

이 제약 조건을 지정하면 어떤 이점이 있을까요? 제네릭 클래스 메소드에서 이제 가상 메소드를 포함한 TObject 메소드를 호출할 수 있습니다! 다음은 TSampleClass클래스에 대한 One 메소드입니다.

```
procedure TSampI eCl ass<T>. One;
begin
  if Assigned (data) then
    begin
      Form30. Log(' Cl assName: ' + data. Cl assName);
      Form30. Log(' Si ze: ' + IntToStr (data. InstanceSi ze));
      Form30. Log(' ToStri ng: ' + data. ToStri ng);
    end;
  end;
```

프로그램을 사용하여 다음 코드 조각에서처럼 One 메소드가 제네릭 타입의 몇몇 인스턴스를 정의하고 사용할 때의 실제 효과를 확인할 수 있습니다.

```
var
  sampI e1: TSampI eCl ass<TButton>;
begin
  sampI e1 := TSampI eCl ass<TButton>. Create;
  try
    sampI e1. SetT (Sender as TButton);
    sampI e1. One;
  finally
    sampI e1. Free;
```

**end;**

사용자 정의된 ToString 메소드를 포함한 클래스를 선언하면 제네릭 타입에 제공된 실제 타입과 상관 없이 데이터 개체가 특정 타입에 속할 경우 이 사용자 정의 버전이 호출됩니다. 다시 말하면 다음과 같은 TButton descendant가 있는 경우,

```
type
  TMyButton = class(TButton)
public
  function ToString: string; override;
end;
```

이 개체를 TSampleClass<TButton> 값으로 전달하거나 제네릭 타입의 특정 인스턴스를 정의할 수 있으며, 두 경우 모두 One을 호출하면 다음과 같이 ToString의 특정 버전을 실행하게 됩니다.

```
var
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TMyButton>;
  mb: TMyButton;
begin
  ...
  sample1.SetT(mb);
  sample1.One;
  sample2.SetT(mb);
  sample2.One;
```

클래스 제약 조건과 유사하게 다음과 같이 선언한 레코드 제약 조건을 사용할 수 있습니다.

```
type
  TSampleRec <T: record> = class
```

그러나 일반적으로 서로 다른 레코드가 공통으로 가지는 것이 거의 없기 때문에(공통 조상이 없음), 이 선언은 다소 제한됩니다.

## 특정 클래스 제약 조건

제네릭 클래스가 클래스의 특정 하위 세트(특정 계층)와 함께 동작해야 하는 경우 지정된 기본 클래스에 기반한 제약 조건을 지정해야 할 수도 있습니다. 예를 들어 다음과 같이 선언하는 경우

```
type
  TCompClass <T: TComponent> = class
```

이 제네릭 클래스의 인스턴스는 구성 요소 클래스, 즉 TComponent 하위 항목 클래스에만 적용할 수 있습니다. 이렇게 하면 매우 특정한 제네릭 타입(다소 이상하게 들리겠지만 실제로 그러함)을 사용할 수 있으며 컴파일러를 사용하여 해당 제네릭 타입에서 작업하는 동안 TComponent 클래스의 모든 메소드를 사용할 수 있습니다.

이 메소드가 매우 강력해 보인다면 다시 한 번 생각해봐야 합니다. 상속과 타입 호환 규칙으로 할 수 있는 작업에 대해 고려한다면 제네릭 클래스를 사용하지 않고도 일반적인 개체 지향 기술을 사용하여 동일한 문제를 해결할 수 있습니다. 특정 클래스 제약 조건이 전혀 유용하지 않다고 말하는 것은 아니지만 높은 수준의 클래스 제약 조건이나 개인적으로 매우 흥미롭게 생각한 인터페이스 기반 제약 조건만큼 강력하지는 않습니다.

## 인터페이스 제약 조건

제네릭 클래스를 지정된 클래스로 제약하는 대신 지정된 인터페이스를 타입 매개 변수로서 구현하는 클래스만 수락하는 것이 일반적으로 보다 나은 유연성을 제공합니다. 그렇게 하면 제네릭 타입의 인스턴스에서 인터페이스를 호출할 수 있습니다.

이미 말했듯이 제네릭에 대한 인터페이스 제약 조건의 사용은 .NET 프레임워크에서 매우 일반적입니다. 예를 통해 인터페이스 제약 조건의 사용을 알아보도록 하겠습니다. 먼저 다음과 같이 인터페이스를 선언해야 합니다.

```
type
  IGetVal ue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    property Value: Integer read GetValue write SetValue;
end;
```

그런 다음 인터페이스를 구현하는 클래스를 정의할 수 있습니다.

```
type
  TGetVal ue = class(TSi ngl etonI mplementati on, IGetVal ue)
  private
    fValue: Integer;
  public
    constructor Create (Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
end;
```

다음과 같이 제네릭 클래스를 지정된 인터페이스를 구현하는 타입으로 정의하면 흥미로운 일이 일어납니다.

```
type
  TInftCl ass <T: IGetVal ue> = class
  private
    val 1, val 2: T; // or IGetVal ue
  public
    procedure Set1 (val : T);
    procedure Set2 (val : T);
    function GetMi n: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
end;
```

이 클래스의 제네릭 메소드에 대한 코드에 다음과 같이 작성할 수 있습니다.

```
function TInftCl ass<T>. GetMi n: Integer;
begin
  Result := mi n(val 1. GetVal ue, val 2. GetVal ue);
end;

procedure TInftCl ass<T>. IncreaseByTen;
begin
  val 1. SetVal ue(val 1. GetVal ue + 10);
  val 2. Val ue := val 2. Val ue + 10;
```

**end;**

이제 이러한 모든 정의를 사용하여 다음과 같이 제네릭 클래스를 사용할 수 있습니다.

```
procedure TForm1.nftConstraint.btnValueClick(Sender: TObject);
var
    iClass: TInftClass<TGetValue>;
begin
    iClass := TInftClass<TGetValue>.Create;
    iClass.Set1 (TGetValue.Create (5));
    iClass.Set2 (TGetValue.Create (25));
    Log (' Average: ' + IntToStr (iClass.GetAverage));
    iClass.IncreaseByTen;
    Log (' Min: ' + IntToStr (iClass.GetMin));
end;
```

이 제네릭 클래스의 유연성을 보여주기 위해 저는 인터페이스에 대해 완전히 다른 구현을 생성했습니다.

```
TButtonValue = class(TButton, IGetValue)
public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
        Parent: TWinControl): TButtonValue;
end;

{ TButtonValue }

function TButtonValue.GetValue: Integer;
begin
    Result := Left;
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
    Left := Value;
end;
```

클래스 함수는 임의 위치의 부모 컨트롤 내에 버튼을 생성하여 다음과 같은 샘플 코드에서 사용됩니다.

```
procedure TForm1.nftConstraint.btnValueButtonClick(Sender: TObject);
var
    iClass: TInftClass<TButtonValue>;
begin
    iClass := TInftClass<TButtonValue>.Create;
    iClass.Set1 (TButtonValue.MakeTButtonValue (self, ScrollBox1));
    iClass.Set2 (TButtonValue.MakeTButtonValue (self, ScrollBox1));
    Log (' Average: ' + IntToStr (iClass.GetAverage));
    Log (' Min: ' + IntToStr (iClass.GetMin)); iClass.IncreaseByTen;
    Log (' New Average: ' + IntToStr (iClass.GetAverage));
end;
```

## 인터페이스 참조와 제네릭 인터페이스 제약 조건 비교

마지막 예에서 저는 지정된 인터페이스를 구현하는 개체와 함께 동작하는 제네릭 클래스를 정의했습니다. 인터페이스 참조에 기반한 표준(비제네릭) 클래스를 생성한다면 유사한 효과를 거둘 수 있습니다. 사실상 다음과 같은 클래스를 정의할 수 있습니다.

```

type
  TPlainInftClass = class
  private
    val 1, val 2: IGetValue;
  public
    procedure Set1 (val : IGetValue);
    procedure Set2 (val : IGetValue);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
end;
    
```

이 두 접근 방법 간의 차이는 무엇일까요? 차이점은 위와 같은 클래스에서 다른 타입의 두 개체 클래스가 모두 지정된 인터페이스를 구현하는 경우 해당 두 개체를 **setter** 메소드에 전달할 수 있다는 것입니다. 반면 제네릭 버전에서는 지정된 타입의 개체만 제네릭 클래스의 지정된 인스턴스에 전달할 수 있습니다. 따라서 제네릭 버전은 타입 확인 측면에서 더 보수적이고 엄격합니다.

중요한 차이점은 인터페이스 기반 버전을 사용하는 것은 **Delphi** 참조 카운트 메커니즘을 사용하는 것을 의미하지만 제네릭 버전을 사용하게 되면 클래스가 지정된 타입의 일반 개체를 처리하며 참조 카운트가 되지 않는다는 것입니다. 뿐만 아니라 제네릭 버전은 생성자 제약 조건과 같은 여러 개의 제약 조건을 가질 수 있으며 개발자는 제네릭 버전을 통해 제네릭 타입의 실제 타입을 요청하는 것과 같은 다양한 제네릭 함수를 사용할 수 있습니다. 이는 인터페이스를 사용하는 경우에는 수행할 수 없습니다. (인터페이스로 작업 시 사실상 기본 **TObject** 메소드에 액세스할 수 없습니다).

다시 말해 인터페이스 제약 조건을 포함한 제네릭 클래스를 사용하면 인터페이스의 단점 없이 이점을 활용할 수 있습니다. 대부분의 경우 두 가지 접근 방법이 유사하지만 어떤 경우에는 인터페이스 기반 솔루션이 더 유연하다는 점에 주목하십시오.

## 미리 정의된 제네릭 컨테이너 사용

**C++** 언어의 초기 템플릿이 배포된 이후 제네릭 클래스가 가장 확실하게 사용되어온 방법 중 하나는 제네릭 컨테이너, 목록 또는 컨테이너에 대한 정의였습니다. **Delphi**의 자체 **TObjectList**와 같은 개체 목록을 정의하면 사실상 어떤 종류의 객체라도 잠재적으로 보유할 수 있는 목록이 표시됩니다. 상속 또는 컴퍼지션을 사용하면 특정 타입에 대한 사용자 정의 컨테이너를 정의할 수 있지만 이 방법은 진부할 뿐만 아니라 에러를 자주 일으킬 수 있는 접근 방법입니다.

**Delphi 2009**는 새 **Generics.Collections** 단위에서 볼 수 있는 제네릭 컨테이너 클래스의 작은 세트를 정의합니다. 4개의 핵심 컨테이너 클래스는 모두 독립적인 방법(다른 클래스에서 상속받지 않음)으로 구현됩니다. 모두 동적 배열을 사용하여 비슷한 방법으로 구현되며 컨테이너 단위의 해당 비제네릭 컨테이너 클래스에 매핑됩니다.

```

type
  TList<T> = class
  TQueue<T> = class
  TStack<T> = class
  TDictionary<TKey, TValue> = class

```

클래스 이름을 고려해 보면 이러한 클래스 간의 논리적 차이는 명백합니다. 이러한 클래스를 테스트하는 좋은 방법은 비제네릭 컨테이너 클래스를 사용하는 기존 코드에서 얼마나 많은 변경을 수행해야 하는지 알아보는 것입니다. 그 예로 저는 **Mastering Delphi 2005**라는 책의 실제 샘플 프로그램을 사용하여 제네릭을 사용하도록 변환했습니다.

## TList<T> 사용

샘플 프로그램에는 TDate 클래스를 정의하는 단위가 있으며 주 형식은 날짜의 TList를 참조하는 데 사용됩니다. 먼저 Generics.Collections를 참조하는 사용 절을 추가한 다음 주 형식 필드의 선언을 다음과 같이 변경했습니다.

```

private
  ListDate: TList<TDate>;

```

물론 목록을 생성하는 주 형식 OnCreate 이벤트 처리기도 업데이트해야 하므로 다음과 같은 구문을 작성해야 합니다.

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  ListDate := TList<TDate>.Create;
end;

```

이제 나머지 코드를 있는 그대로 컴파일해 볼 수 있습니다. 프로그램에는 TButton 개체를 목록에 추가하도록 시도하는 “의도한” 버그가 있습니다. 컴파일하는 데 사용되는 해당 코드는 이제 다음과 같이 실패합니다.

```

procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  ListDate.Add(Sender); // Error:
  // E2010 Incompatible types: 'TDate' and 'TObject'
end;

```

날짜의 새 목록은 타입 확인 측면에서 원래의 제네릭 목록 포인터보다 더 강력합니다. 해당 줄을 제거하고 프로그램에서 컴파일 및 작동합니다. 여전히 프로그램을 개선할 수 있습니다.

다음은 ListBox 컨트롤에서 목록의 모든 날짜를 표시하는 데 사용되는 원본 코드입니다.

```

var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    ListBox1.Items.Add((TObject(ListDate[I]) as TDate).Text);

```

프로그램이 개체 목록(TObjectList)이 아닌 포인터 목록(TList)을 사용하고 있기 때문에 캐스트가 매끄럽지 못합니다. 원본 데모 날짜가 TObjectList 클래스 날짜보다 앞서기 때문일 수 있습니다! 다음을 작성하면 프로그램을 쉽게 개선할 수 있습니다.

```
for I := 0 to ListDate.Count - 1 do
  Listbox1.Items.Add(ListDate[I].Text);
```

이 코드에 대한 또 다른 개선 방법은 일반 for loop 대신 열거(미리 정의된 제네릭 목록이 완벽히 지원)를 사용하는 것입니다.

```
var
  aDate: TDate;
begin
  for aDate in ListDate do
    begin
      Listbox1.Items.Add(aDate.Text);
    end;
```

마지막으로 TDate 개체를 가진 제네릭 TObjectList를 사용하여 프로그램을 개선할 수 있지만 이것에 대해서는 다음 섹션에서 설명하겠습니다.

이미 설명했듯이 TList<T> 제네릭 클래스는 높은 수준의 호환성을 제공합니다. Add, Insert, Remove 및 IndexOf와 같은 모든 클래스 메소드가 있습니다. Capacity 및 Count 속성도 있습니다. 이상하게도 Items가 Item이 되지만 기본 속성이 되면 거의 참조를 하지 않게 됩니다.

## TList<T> 정렬

정렬이 작동하는 방식은 아주 흥미롭습니다. (여기에서의 목표는 정렬 지원을 앞의 예에 추가하는 것입니다.) Sort 메소드는 다음과 같이 정의됩니다.

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

IComparer<T>는 Generic.Defaults 단위 내에 선언됩니다. 첫 번째 버전의 프로그램을 호출하면 프로그램은 TList<T>의 기본 생성자로 초기화된 기본 비교자를 사용합니다. 이 경우에는 쓸모가 없습니다.

대신에 IComparer<T> 인터페이스의 적합한 구현을 정의해야 합니다. 타입 호환성을 위해 특정 TDate 클래스에서 작동하는 구현을 정의해야 합니다. 다음 섹션에서 설명할 익명 메소드를 포함하여 이를 수행할 수 있는 방법은 여러 가지가 있습니다.

이 흥미로운 기술은 몇 가지의 제네릭 사용 패턴을 표시하고 TComparer라고 불리는 Generics.Defaults 단위의 일부인 구조 클래스를 활용할 수 있는 기회를 제공합니다. 클래스는 다음과 같이 인터페이스의 추상 및 제네릭 구현으로 정의됩니다.

```
type
  TComparer<T> = class(InterfacedObject, IComparer<T>)
  public
    class function Default: IComparer<T>;
    class function Construct(
      const Comparison: TComparison<T>): IComparer<T>;
    function Compare(const Left, Right: T): Integer; virtual; abstract;
  end;
```

여기서 해야 하는 작업은 특정 데이터 타입에 대한 제네릭 클래스(예의 경우 TData)를 인스턴트화하고 특정 타입에 대한 Compare 메소드를 구현하는 구체적인 클래스를 상속하는 것입니다. 잠시만 배울 수 있는 코딩 관용구를 사용하여 다음과 같이 두 개의 작업을 한 번에

수행할 수 있습니다.

```
type
  TDateComparer = class(TComparer<TDate>)
    function Compare(const Left, Right: TDate): Integer; override;
  end;
```

이 코드를 보면 많은 사람들이 이상하게 생각할 것입니다. 새 클래스가 제네릭 클래스의 특정 인스턴스에서 상속되면 다음과 같이 두 개의 개별 단계로 표현할 수 있습니다.

```
type
  TAnyDateComparer = TComparer<TDate>;
  TMyDateComparer = class(TAnyDateComparer)
    function Compare(const Left, Right: TDate): Integer; override;
  end;
```

여기에서 강조하려는 핵심 사항은 아니지만 소스 코드에서 **Compare** 함수의 실제 구현을 볼 수 있습니다. 그렇지만 목록을 정렬하더라도 **IndexOf** 메소드는 **TStringList** 클래스와 달리 목록을 활용하지는 않습니다.

## 익명 메소드를 사용한 정렬

이전 섹션에 나온 정렬 코드는 꽤 복잡해 보이며 실제로도 복잡합니다. 정렬 함수를 **Sort** 메소드로 직접 전달하는 것이 훨씬 쉬우면서도 깨끗합니다. 과거에는 일반적으로 함수 포인터를 전달하여 정렬을 수행했습니다. Delphi 2009에서는 익명 메소드를 전달하여 이를 수행할 수 있습니다.

사실상 **TList<T>** 클래스에 있는 **Sort** 메소드의 **IComparer<T>** 매개 변수는 다음과 같이 정의된 익명 메소드를 매개 변수로 전달하는 **TComparer<T>**의 **Construct** 메소드를 호출하여 사용할 수 있습니다.

```
type
  TComparison<T> = reference to function(const Left, Right: T): Integer;
```

실제로 타입-호환 함수를 작성하고 다음과 같이 해당 함수를 매개 변수로 전달할 수 있습니다.

```
function DoCompare(const Left, Right: TDate): Integer;
var
  lDate, rDate: TDateTime;
begin
  lDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  rDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if lDate = rDate then
    Result := 0
  else if lDate < rDate then
    Result := -1
  else
    Result := 1;
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

이 방법이 너무 일반적인 것으로 생각되는 경우 다음과 같이 개별 함수의 선언을 피하고 개별

함수(해당 소스 코드)를 매개 변수로서 **Construct** 메소드에 보내는 것을 고려해볼 수 있습니다.

```

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort(TComparer<TDate>.Construct(
    function (const Left, Right: TDate): Integer
    var
      lDate, rDate: TDateTime;
    begin
      lDate := EncodeDate(Left.Year, Left.Month, Left.Day);
      rDate := EncodeDate(Right.Year, Right.Month, Right.Day);
      if lDate = rDate then
        Result := 0
      else if lDate < rDate then
        Result := -1
      else
        Result := 1;
    end));
end;
  
```

이 예는 익명 메소드를 보여 주는 간단한 예로서 자세한 내용은 나중에 학습하게 될 것입니다. 이번 최종 버전은 이전 섹션에서 다룬 원본보다 훨씬 쉽게 작성할 수 있습니다. 그렇지만 파생 클래스를 포함하는 것이 대부분의 Delphi 개발자에게 보다 깨끗하게 보이고 이해하기 쉬울 수 있습니다.

## 익명 메소드(클로저)

Delphi 언어에서는 프로시저 타입(포인터를 프로시저 및 함수에 선언하는 타입)과 메소드 포인터(포인터를 메소드에 선언하는 타입)를 오랫동안 사용해왔습니다. 프로시저 타입과 포인터를 직접적으로 자주 사용하지는 않더라도 이는 모든 개발자들이 작업에 사용하는 Delphi의 주요 기능입니다. 사실 메소드 포인터 타입은 VCL의 이벤트 처리기에 대한 기초가 됩니다. 이벤트 처리기를 선언할 때마다, 그것이 순수 **Button1Click** 이라 하더라도 개발자는 메소드 포인터를 사용하는 이벤트(이 경우 **OnClick** 이벤트)에 연결되는 메소드를 선언하게 됩니다.

익명 메소드는 다른 위치에서 정의한 메소드의 이름이 아니라 메소드의 실제 코드를 매개 변수로 전달함으로써 이 기능을 확장합니다. 그렇지만 이것이 유일한 차이점은 아닙니다. 익명 메소드가 다른 기술과 매우 다른 부분은 로컬 변수의 수명을 관리하는 방법입니다.

익명 메소드는 Delphi에서는 새 기능이지만 수년간 다른 프로그래밍 언어(특히 동적 언어)에서 다른 형식으로 또한 다른 이름으로 사용되어져 왔습니다. 제 경우에는 **JavaScript**에서의 마감, 특히 **jQuery** ([www.jquery.org](http://www.jquery.org)) 라이브러리 및 **AJAX** 호출 작업을 많이 해보았습니다. **C#**에서의 해당 기능은 익명 제어입니다.

여기서는 마감과 다양한 프로그래밍 언어에서의 관련 기술을 비교하지 않고 대신 Delphi 2009에서 마감이 어떻게 작동되는지에 대한 정보를 기술하도록 하겠습니다.

## 익명 메소드의 구문 및 의미

Delphi의 익명 메소드는 표현식 컨텍스트에서 메소드 값을 생성하는 메커니즘입니다. 다소 복잡한 정의이지만 메소드 포인터와의 핵심적인 차이를 나타내주는 것은 표현식 컨텍스트라는 말입니다. 익명 메소드를 설명하기 전에 매우 간단한 코드 예부터 시작하도록 하겠습니다.

다음은 익명 메소드 타입 선언으로, 이를 통해 Delphi가 강력한 타입의 언어로 계속 사용될 수 있게 됩니다.

```
type
  TIntProc = reference to procedure(n: Integer);
```

선언에 사용되는 키워드에서만 참조 메소드와 차이를 보입니다.

```
type
  TIntMethod = procedure(n: Integer) of object;
```

## 익명 메소드 변수

익명 메소드 타입이 있으면 다음과 같이 이 타입의 변수를 선언하고 타입-호환 익명 메소드를 할당하고 변수를 통해 해당 메소드를 호출할 수 있습니다.

```
procedure TFormAnonymFirst.btnSimpleVarClick(Sender: TObject);
var
  anIntProc: TIntProc;
begin
  anIntProc :=
    procedure(n: Integer)
      begin
        Memo1.Lines.Add(IntToStr(n));
      end;
  anIntProc(22);
end;
```

현재 위치 코드를 포함한 실제 프로시저를 변수에 할당하는 데 사용되는 구문을 눈여겨 보십시오. 이 구문은 과거의 Pascal에서 볼 수 없었던 구문입니다.

## 익명 메소드 매개 변수

보다 놀라운 구문을 포함하는 좀 더 흥미로운 예를 들자면, 익명 메소드를 매개 변수로서 함수에 전달할 수 있다는 것입니다. 다음과 같이 익명 메소드 매개 변수를 사용하는 함수가 있다고 가정해 보겠습니다.

```
procedure CallTwice(value: Integer; anIntProc: TIntProc);
begin
  anIntProc(value);
  Inc(value);
  anIntProc(value);
end;
```

함수는 두 개의 연속 정수 값으로 두 번에 걸쳐 매개 변수로서 전달된 메소드를 호출합니다. 한

번은 매개 변수로서 전달되며 다른 한 번은 그 뒤에 수반됩니다. 실제 익명 메소드를 함수에 전달하여 다음과 같이 직접적인 현재 위치 코드를 포함한 함수를 호출합니다.

```

procedure TFormAnonymFirst.btnProcParamClick(Sender: TObject);
begin
  CallTwice(48,
    procedure(n: Integer)
    begin
      Memo1.Lines.Add(IntToHex(n, 4));
    end);
  CallTwice(100,
    procedure(n: Integer)
    begin
      Memo1.Lines.Add(FloatToStr(Sqrt(n)));
    end);
end;

```

구문의 측면에서 프로시저가 괄호와 함께 매개 변수로서 전달되고 세미콜론(;)으로 종료되지 않는 것을 눈여겨 보십시오. 코드의 실제 결과는 48 및 49를 지정하여 IntToHex를, 100 및 101 제공근으로 FloatToStr을 호출한 값이며 다음과 같이 출력됩니다.

```

0030
0031
10
10.0498756211209

```

## 로컬 변수 사용

사용하는 구문이 “그다지 훌륭하지” 않다 하더라도 메소드 포인터를 사용하면 동일한 효과를 얻을 수 있습니다. 익명 메소드를 명확히 차별시켜주는 것은 익명 메소드가 호출 메소드의 로컬 변수를 참조할 수 있는 방법입니다. 다음 코드를 고려해 보십시오.

```

procedure TFormAnonymFirst.btnLocalValClick(Sender: TObject);
var
  aNumber: Integer;
begin
  aNumber := 0;
  CallTwice(10,
    procedure(n: Integer)
    begin
      Inc(aNumber, n);
    end);
  Memo1.Lines.Add(IntToStr(aNumber));
end;

```

여기에서 여전히 CallTwice 프로시저에 전달되는 메소드는 로컬 매개 변수 n뿐 아니라 호출 컨텍스트의 로컬 변수인 aNumber도 사용합니다. 효과는 무엇일까요? 익명 메소드를 두 번 호출하면 매개 변수가 로컬 변수에 추가되어(첫 번째에는 10, 두 번째에는 11) 해당 로컬 변수가 수정됩니다. aNumber의 최종 값은 21입니다.

## 로컬 변수의 수명 연장

앞의 예는 흥미로운 결과를 보여 주지만 중첩 함수 호출의 시퀀스와 로컬 변수를 사용할 수 있다는

사실은 놀랍지 않습니다. 그러나 익명 메소드의 장점은 로컬 변수를 사용하여 필요에 따라 수명을 연장할 수 있다는 사실에 있습니다. 장황한 설명보다는 예를 통해 그것을 설명하겠습니다.

저는 클래스 완성을 사용하여 다음과 같이 `TFormAnonymFirst` 형식 클래스에 익명 메소드 포인터 타입의 속성을 추가했습니다. (사실 프로젝트의 모든 코드에 사용한 것과 동일한 익명 메소드입니다.)

```
private
  FAnonMeth: TIntegerProc;
  procedure SetAnonMeth(const Value: TIntegerProc);
public
  property AnonMeth: TIntegerProc read FAnonMeth write SetAnonMeth;
```

그 다음 두 개의 추가 버튼을 형식에 추가했습니다. 첫 번째 버튼은 다음과 같이 로컬 변수를 사용하는 익명 메소드를 속성에 저장(이전 `btnLocalValClick` 메소드에서의 방식과 유사)합니다.

```
procedure TFormAnonymFirst.btnStoreClick(Sender: TObject);
var
  aNumber: Integer;
begin
  aNumber := 3;
  AnonMeth :=
    procedure(n: Integer)
    begin
      Inc(aNumber, n);
      Memo1.Lines.Add(IntToStr(aNumber));
    end;
end;
```

이 메소드를 실행하면 익명 메소드는 실행되지 않고 저장만 됩니다. 로컬 변수인 `aNumber` 는 0(영)으로 초기화되어 수정되지 않고 메소드가 종료될 때 로컬 범위를 벗어나 위치가 변경됩니다. 최소한 이것은 표준 Delphi 코드에서 예상되는 작업입니다.

이 단계를 위해 양식에 추가한 두 번째 버튼은 익명 메소드라고 하며, `AnonMeth` 속성에 저장됩니다.

```
procedure TFormAnonymFirst.btnCallClick(Sender: TObject);
begin
  if Assigned(AnonMeth) then
    begin
      CallTwice(2, AnonMeth);
    end;
end;
```

이 코드를 실행하면 코드는 스택에 더 이상 존재하지 않는 메소드의 로컬 변수 `aNumber` 를 사용하는 익명 메소드를 호출합니다. 그러나 익명 메소드는 실행 컨텍스트를 캡처하기 때문에 변수는 그대로 있고 익명 메소드의 해당 지정된 인스턴스(메소드에 대한 참조)가 있는 한 사용될 수 있습니다.

더 자세히 알아보려면 다음을 수행하십시오. 저장 버튼을 한번 누르고 호출 버튼을 두 번 누르면 사용 중인 것과 동일한 캡처된 변수가 다음과 같이 표시됩니다.

```
5
8
10
```

13

이제 저장 버튼을 한 번 더 누른 다음 다시 호출을 누릅니다. 로컬 변수의 값이 재설정되는 이유는 무엇일까요? 새 익명 메소드 인스턴스를 할당하면 자체 실행 컨텍스트와 함께 이전 익명 메소드 인스턴스가 삭제되고 로컬 변수의 새 인스턴스를 포함한 새 실행 컨텍스트가 캡처됩니다. 전체 시퀀스 저장 - 호출 - 호출 - 저장 - 호출을 실행하면 다음과 같은 결과가 생성됩니다.

```
5
8
10
13
5
8
```

다른 일부 언어에서 수행하는 동작과 비슷한 이 동작은 익명 메소드가 과거에 문자 그대로 불가능했던 기능을 실행하는 데 사용할 수 있는 매우 강력한 언어 기능을 가질 수 있다는 것을 보여줍니다.

## 기타 새로운 언어 기능

Object Pascal 언어의 많은 새 중요 기능을 살펴보다 보면 일부 사소한 기능을 놓치기 쉽습니다.

### 주석 처리된 DEPRECATED 지시어

`deprecated` 지시어(기호를 표시하는 데 사용됨)은 호환성이 문제가 되는 경우 계속 사용할 수 있지만 이제 컴파일러 경고의 일부로 표시되는 문자열이 뒤에 나올 수 있습니다. 다음과 같은 코드 조각에서처럼 프로시저를 정의하고 호출하는 경우

```
procedure DoNothing;
  deprecated 'use DoSomething instead';
begin
end;

procedure TFormMinorLang.btnDepracatedClick(Sender: TObject);
begin
  DoNothing;
end;
```

`btnDepracatedClick` 메소드의 호출 위치에 다음과 같은 경고가 표시됩니다.

```
W1000 Symbol 'DoNothing' is deprecated: 'use DoSomething instead'
```

이 방법은 사용하지 않는 기호의 선언에 주석을 추가하는 이전 방법보다 훨씬 낫습니다. 이 방법에서는 에러 메시지에서 클릭하여 기호가 사용되는 소스 코드 줄을 열고 선언 위치로 이동하여 주석을 찾습니다. 말할 필요도 없이 위의 코드는 Delphi 2007에서 컴파일되지 않으며 다음과 같은 에러가 발생합니다.

```
E2029 Declaration expected but string constant found
```

새 `deprecated` 기능은 Delphi 2009 RTL 및 VCL에서는 많이 사용되지만 이전 버전의 컴파일러와 호환되지 않기 때문에 타사 공급업체는 이 기능을 사용하는 데 제약이 있을 것입니다.

## EXIT에서 값 리턴

일반적으로 Pascal 함수는 다음과 같이 함수 이름을 사용하여 결과를 할당하는 데 사용됩니다.

```
function ComputeValue: Integer;
begin
  ...
  ComputeValue := 10;
end;
```

Delphi는 Result 식별자를 사용하여 반환값을 함수에 할당하는 대체 코딩을 제공해 왔습니다.

```
function ComputeValue: Integer;
begin
  ...
  Result := 10;
end;
```

두 개의 접근 방법은 동일하며 코드의 플로우를 변경하지 않습니다. 함수 결과를 할당하고 두 개의 개별 문을 사용할 수 있는 현재 실행을 중지하려면, 결과를 할당한 다음 **Exit**를 호출하십시오. 문자열 목록에 있는 지정된 번호를 포함하는 문자열을 검색하는 다음 코드 조각은 이 접근 방법에 대한 일반적인 예입니다.

```
function FindExit (sl: TStringList; n: Integer): string;
var
  I: Integer;
begin
  for I := 0 to sl.Count do
    if Pos(IntToStr (n), sl [I]) > 0 then
      begin
        Result := sl [I];
        Exit;
      end;
  end;
end;
```

Delphi 2009에서는 두 개의 문을 **Exit**에 대한 새 특정 호출로 대체하여, C언어 반환 문과 비슷한 방법으로 함수의 반환 값을 전달할 수 있습니다. 또한 단일 문을 사용하여 **begin/end**를 지정하지 않아도 되므로 더 간결한 버전으로 위의 코드를 작성할 수 있습니다.

```
function FindExitValue (sl: TStringList; n: Integer): string;
var
  I: Integer;
begin
  for I := 0 to sl.Count do
    if Pos(IntToStr (n), sl [I]) > 0 then
      Exit(sl [I]);
  end;
```

## 새 별칭 정수 타입

엄격히 말해 컴파일러의 변경 사항이 아니라 시스템 단위에 추가된 사항이긴 합니다만, 이제 서명된 정수 데이터 타입 및 서명되지 않은 정수 데이터 타입에 대해 기억하기 쉬운 별칭 세트를 사용할 수 있습니다. 다음은 컴파일러의 미리 정의된 서명된 정의 타입 및 서명되지 않은 정의 타입입니다.

ShortInt	Byte
SmallInt	Word
Integer	Cardi nal
Nati veInt	Nati veUI nt
Int64	UI nt64

이러한 타입은 Delphi 2007 및 이전 버전에 이미 존재하지만 64비트 타입은 일부 버전의 컴파일러에만 존재합니다. 컴파일러 버전(32비트 및 향후 64비트)에 따라 달라지는 NativeInt 및 NativeUInt 타입은 Delphi 2007에 이미 있지만 문서화되지 않았습니다.

CPU 네이티브 정수 크기와 일치하는 데이터 타입이 필요한 경우 이러한 타입을 사용할 수 있습니다. 사실상 정수 타입은 32비트에서 64비트로 이동 시 변경되지 않도록 되어 있습니다.

시스템 단위에 의해 추가된 다음 미리 정의된 별칭 세트는 Delphi 2009의 새로운 기능입니다.

```

type
  Int8    = ShortInt;
  Int16   = SmallInt;
  Int32   = Integer;
  UInt8   = Byte;
  UInt16  = Word;
  UInt32  = Cardi nal ;
    
```

새로운 세트를 추가하지 않았더라도 일반적으로 ShortInt가 SmallInt보다 작은지 기억하기가 어렵고 Int16 또는 Int8의 실제 구현을 기억하기는 쉽기 때문에 이 미리 정의된 별칭 세트가 보다 사용하기 쉬울 것입니다.

## 결론

Delphi 언어에 추가된 몇 가지 흥미로운 기능을 다루긴 했지만 컴파일러의 이번 버전에서 가장 큰 차이점은 제네릭에 대한 지원, 익명 메소드에 대한 지원 및 제네릭과 익명 메소드의 조합입니다. 이러한 기능은 Delphi 언어를 단순히 확장하는 것이 아니라 일반적인 개체 지향 프로그래밍과 Delphi에서 전통적으로 사용되어 온 이벤트 중심 프로그래밍 접근 방법 이외의 새 프로그래밍 패러다임에 대한 가능성을 열었습니다. 클래스를 한 개 이상의 데이터 타입에서 매개 변수화할 수 있는 기능과 루틴을 매개 변수로 전달하는 기능은 새로운 코딩 스타일과 Delphi 응용 프로그램의 새로운 설계 방법을 제시합니다. Delphi 2009의 언어 기능은 바로 여기에 있지만 라이브러리와 구성 요소가 이러한 기능을 완전히 활용할 때까지는 시간이 걸릴 것입니다. Delphi 2009를 사용하면 오늘 바로 새 코딩 기술을 사용하여 작업을 시작할 수 있습니다.

## 필자에 대하여

이 문서는 베스트 셀러 시리즈인 Mastering Delphi의 저작자 Marco Cantù가 Embarcadero Technologies 를 위해 작성하였습니다. 이 문서의 내용은 그의 최근 저서인 “Delphi 2009 핸드북” (<http://www.marcocantu.com/dh2009>)에서 발췌한 것입니다. Marco Cantù에 대한 정보는 그의 개인 블로그(<http://blog.marcocantu.com>)에서 읽을 수 있으며 전자

메일([marco.cantu@gmail.com](mailto:marco.cantu@gmail.com))을 통해 연락할 수 있습니다.



Embarcadero Technologies Inc.는 애플리케이션 개발자 및 데이터베이스 전문가가 자신이 선택한 환경에서 소프트웨어 애플리케이션을 설계, 빌드 및 실행하는 도구를 사용할 수 있도록 합니다. 전 세계 3백만 이상의 커뮤니티와 Fortune지 선정 100대 기업 중 90개 기업이 Embarcadero의 CodeGear™ 및 DatabaseGear™ 제품군을 기반으로 하여 생산성을 향상시키고 개방적인 협업 및 자유로운 혁신을 추구하고 있습니다. Embarcadero는 1993년에 설립되어 캘리포니아 샌프란시스코에 본사가 있으며 전 세계에 사무소를 두고 있습니다. Embarcadero의 온라인 주소는 [www.embarcadero.com](http://www.embarcadero.com)입니다. Embarcadero의 주요 제품인 DatabaseGear의 도구에는 ER/Studio®, DBArtisan®, Rapid SQL® 및 Embarcadero® Change Manager™가 있습니다.



데브기어는 미국 Embarcadero Technologies Inc.와 기존의 코드기어 한국 지사의 협력으로 전략적으로 설립된 엠바카데로 솔루션 전문 공급 기업입니다. 데브기어는 Delphi, C++Builder, JBuilder, Delphi Prism 등 개발툴 제품들과 ER/Studio, PowerSQL, DB Artisan, EA/Studio 등의 데이터베이스 툴 제품들에 대한 한국 시장에 공급은 물론 기술지원 및 교육을 제공합니다. 데브기어 웹 사이트는 <http://www.devgear.co.kr/>이며 제품에 대한 문의는 [ask@embarcadero.kr](mailto:ask@embarcadero.kr) 로 하면 됩니다.