CHAPTER _7

제네릭(Generics)

이 장에서는 델파이 2009에서 추가된 강력한 문법 특성인 제네릭의 개요와 제네릭을 위한 문법 변경 사항 등을 살펴보고, 제네릭의 제한 지정, 그리고 오버로드의 사용에 대해 알아봅니다.

■제네릭의 개요 ■제네릭의 용어들 ■제네릭의 선언
■제네릭에서의 오버로드와 타입 호환성 ■제네릭의 제약조건
■제네릭 내의 클래스 변수 ■표준함수와 문법의 변경 사항

제네릭의 개요

제네릭, 혹은 제네릭 타입이라는 용어들은 타입으로 파라미터화될 수 있는 기반 내의 것들의 집합을 설명합니다. 제네릭이라고 말했을 때 제네릭 타입과 제네릭 메소드(제네릭 프로시저와 제네릭 함수)를 의미하기도 합니다.

제네릭은 알고리즘(프로시저나 함수 등)이나 데이터 구조(클래스, 인터페이스, 레코드 등)를 그 알고리즘 혹은 데이터 구조가 사용하는 하나 이상의 구체적인 타입으로부터 관계를 끊는 것을 허용하는 추상화 방법들의 집합입니다.

그 정의에서 다른 타입들을 사용하는 메소드나 데이터 타입은 하나 이상의 구체적 타입을 타입 파라미터로 대체시킴으로써 더 일반화시킬 수 있습니다. 그런 후 그런 타입 파라미터를 메소드나 데이터 구조 선언의 타입 파라미터 리스트에 추가하면 됩니다. 이것은, 바디에 있는 문자 상수의 인스턴스를 파라미터 이름으로 대체시키고 그 파라미터를 프로시저의 파라미터 리스트에 추가하여 프로시저를 더 일반화시킬 수 있는 것과 비슷합니다.

예를 들어, 객체들(TObject 타입)의 리스트를 유지하는 클래스(TMyList)를, TObject를

타입 파라미터 이름('T' 같은)으로 대체하고 타입 파라미터를 클래스의 타입 파라미터 리스트에 추가하여 TMyList<T>으로 하면, 재사용성이 높고 타입에 안전하도록 만들 수 있습니다.

제네릭 타입이나 메소드를 구체화(인스턴스화)하려면 사용하려는 시점에 제네릭 타입이나 메소드에 타입 인자를 지정하면 됩니다. 타입 인자를 지정하는 동작은 제네릭 정의 내의 타입 파라미터들을 해당 타입 인자로 대체함으로써 새로운 타입이나 메소드를 실질적으로 생성하게 됩니다.

예를 들면, 위의 리스트는 TMyList<Double>와 같이 사용될 수 있습니다. 이 코드는 TMyList<Double> 이라는 새로운 타입을 생성하며, 그 정의는 모든 'T'가 'Double'로 대체되었다는 점 외에는 TMyList<T>와 동일합니다.

서로 다른 특징들이 있기는 하지만, 추상화(abstraction) 메커니즘으로서의 제네릭은 다형성(polymorphism)의 기능들과 많은 부분에서 중복된다는 점을 언급할 필요가 있습니다. 인스턴스화 시점에서 새로운 타입이나 메소드가 만들어지기 때문에 타입 에러를 더 빨리(실행 중이 아닌 컴파일 중에) 찾아낼 수 있습니다. 이런 특징은 또한 최적화의 범위를 넓혀준다는 장점도 있지만, 다른 한편으로는 각각의 인스턴스화마다 최종 실행되는 애플리케이션의 메모리를 사용하기 때문에 성능을 낮출 가능성도 있습니다.

코드 예제

다음의 예에서, TSIPair는 String과 Integer의 두 데이터 타입을 저장하는 클래스입니다.

```
type
  TSIPair = class
  private
    FKey: String;
    FValue: Integer;
  public
    function GetKey: String;
    procedure SetKey(Key: String);
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;
```

클래스를 데이터 타입과 독립적으로 만들려면, 데이터 타입을 타입 파라미터로 대체합니다.

```
type
  TPair<TKey,TValue>= class    // 두 타입 파라미터로 TPair 타입을 선언
  private
    FKey: TKey;
    FValue: TValue;
  public
    function GetKey: TKey;
    procedure SetKey(Key: TKey);
    function GetValue: TValue;
    procedure SetValue(Value: TValue);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;

type
  TSIPair = TPair<String,Integer>; // 인스턴스화된 타입 선언
  TSSPair = TPair<String,String>;  // 다른 데이터 타입으로 선언
  TISPair = TPair<Integer,String>;
  TIIPair = TPair<Integer,Integer>;
  TSXPair = TPair<String,TXMLNode>;
```

플랫폼 요구 사항들과 차이점들

델파이의 제네릭은 .NET 2.0 및 그 이후 버전, 그리고 네이티브 Win32 컴파일러에서 지원됩니다. 이들 플랫폼들에서 지원되는 기능들에는 약간의 차이들이 있습니다.

■ 런타임 타입 정보(RTTI)

Win32에서, 제네릭과 메소드는 RTTI 정보를 갖지 않지만 인스턴스화된 타입은 RTTI 정보를 갖습니다. 인스턴스화된 타입은 제네릭과 파라미터들의 집합의 조합입니다.

■ 인터페이스 GUID

Win32에서, 인스턴스화된 인터페이스 타입은 인터페이스 GUID를 갖지 않습니다.

■ 인터페이스 내의 파라미터화된 메소드

Win32에서, 파라미터화된 메소드(타입 파라미터와 함께 선언된 메소드)는 인터페이스 내에서 선언될 수 없습니다. .NET에서는 이것이 가능합니다.

■ 인스턴스화 시점

Win32에서는, 인스턴스화는 컴파일러에 의해 수행됩니다. 모든 인스턴스화된 객체들이 .obj 파일로 내보내어집니다. .NET에서는, 인스턴스화는 런타임 환경에서 수행됩니다.

■ 동적 인스턴스화

Win32에서는 런타임에 동적 인스턴스화가 지원되지 않습니다.

■ 인터페이스 제약조건

Win32 인터페이스는 "가벼운" 인터페이스가 아닙니다. 이것은 모든 타입 파라미터 constraints가 항상 COM IUnknown 메소드들인 _AddRef, _Release, QueryInterface를 지원하거나 TInterfacedObject로부터 상속받는다는 것을 의미합니다. 레코드 타입은 인터페이스 제약조건 파라미터를 지정할 수 없습니다. .NET에서만 레코드 타입 내의 인터페이스를 지원하기 때문입니다.

제네릭의 용어들

제네릭을 설명하기 위해 사용되는 용어들은 아래와 같습니다.

타입 제네릭(type generic)	실제 타입을 형성하기 위해 제공될 타입 파라미터를 요구하는 타입 선언. 아래 코드에서 'List<Item>'은 제네릭입니다. **type** List<Item> = **class** ... **end;**
제네릭	타입 제네릭과 동일
타입 파라미터	제네릭 선언이나 메소드 바디 내의 다른 선언에서 타입으로 사용되기 위해 제네릭 선언 혹은 메소드 헤더 내에서 선언된 파라미터. 실제 타입 인자에 연결될 것임. 아래 코드에서 Item이 타입 파라미터입니다. **type** List<Item> = **class** ... **end;**
타입 인자 (argument)	인스턴스화된 타입을 만들기 위해 타입 식별자와 함께 사용되는 타입. List<Integer>의 형태에서, List가 제네릭이면 타입 인자는 Integer입니다.
인스턴스화된 타입 (Instantiated type)	제네릭과 파라미터들의 집합의 조합.
구축된 타입 (constructed type)	인스턴스화된 타입과 동일.

닫힌 구축된 타입 (closed constructed type)	모든 파라미터들이 실제 타입으로 결정된 구축된 타입. List<Integer>의 형태에서, Integer가 실제 타입이므로 List<Integer>는 닫힌 구축된 타입입니다.
열린 구축된 타입 (open constructed type)	적어도 하나의 파라미터가 타입 파라미터인 구축된 타입. 포함 클래스에서 T가 타입 파라미터일 경우, List<T>는 열린 구축된 타입입니다.
인스턴스화(instantiation)	컴파일러가 제네릭에 정의된 메소드에 대한 실제 명령 코드와 닫힌 구축된 타입에 대한 실제 가상 메소드 테이블을 생성합니다. 이 과정은 델파이 컴파일드 유닛 파일(.dcu)나 오브젝트 파일(.obj)을 생성해내기 전에 필요합니다. .NET에서는 .NET이 타입을 인스턴스화하기 위한 명령들을 가지고 있기 때문에 필요하지 않습니다.

제네릭의 선언

제네릭의 선언은 일반적인 클래스, 레코드, 인터페이스 타입과 비슷합니다. 차이점은, 제네릭 선언의 타입 식별자 뒤에 각진 괄호(<>)로 둘러싸인 하나 이상의 타입 파라미터의 리스트가 따라온다는 것입니다.
타입 파라미터는 포함하는 타입(container type) 선언과 메소드 바디 내에서 일반적인 타입 식별자처럼 사용될 수 있습니다.
예를 들면 다음과 같습니다.

```
type
  TPair<Tkey,TValue> = class    // TKey와 TValue는 타입 파라미터입니다
    FKey: TKey;
    FValue: TValue;
    function GetValue: TValue;
  end;

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

타입 인자

제네릭 타입은 타입 인자(type argument)를 제공함으로써 인스턴스화됩니다. 델파이에서 어떤 타입이든 타입 인자로 사용할 수 있지만, 정적 배열, 짧은 문자열, 그리고 이들 두가지 타입의 필드를 포함하는(재귀적으로) 레코드 타입의 경우는 예외입니다.

```
type
  TFoo<T> = class
    FData: T;
  end;

var
  F: TFoo<Integer>; // 'Integer가 TFoo<T>의 타입 인자입니다
begin
...
end.
```

중첩된 타입

제네릭 내에 중첩된 타입도 역시 제네릭입니다.

```
type
  TFoo<T> = class
    type
      TBar = class
        X: Integer;
        // ...
      end;
      // ...  TBaz = class
    type
      TQux<T> = class
        X: Integer;
        // ...
      end;
    // ...
  end;
```

중첩된 TBar 타입을 액세스하려면 먼저 TFoo 타입을 구축(construct)해야 합니다.

```
var
  N: TFoo<Double>.TBar;
```

제네릭은 또한 일반 클래스 내에서 중첩된 타입으로도 선언될 수 있습니다.

```
type
  TOuter = class
    type
      TData<T> = class
```

```
      FFoo1: TFoo<Integer>;          // 닫힌 구축된 타입으로 선언됨
      FFoo2: TFoo<T>;                // 열린 구축된 타입으로 선언됨
      FFooBar1: TFoo<Integer>.TBar; // 닫힌 구축된 타입으로 선언됨
      FFooBar2: TFoo<T>.TBar;        // 열린 구축된 타입으로 선언됨
      FBazQux1: TBaz.TQux<Integer>; // 닫힌 구축된 타입으로 선언됨
      FBazQux2: TBaz.TQux<T>;        // 열린 구축된 타입으로 선언됨
      ...
    end;
  var
    FIntegerData: TData<Integer>;
    FStringData: TData<String>;
end;
```

기반 타입

파라미터화된 클래스나 인터페이스의 기반 타입(base type)은 실제 타입이거나 구축된 타입일 수 있습니다. 기반 타입은 타입 파라미터만으로 이루어질 수는 없습니다.

```
type
  TFoo1<T> = class(TBar)             // 실제 타입
  end;

  TFoo2<T> =  class(TBar2<T>)        // 열린 구축된 타입
  end;

  TFoo3<T> = class(TBar3<Integer>)   // 닫힌 구축된 타입
  end;
```

TFoo2〈String〉이 인스턴스화되면, 조상 클래스는 TBar2〈String〉이 되며, TBar2〈String〉은 자동으로 인스턴스화됩니다.

클래스, 인터페이스, 레코드 타입

클래스, 인터페이스, 레코드 타입은 타입 파라미터로 선언될 수 있습니다.
예를 들면 다음과 같습니다.

```
type
  TRecord<T> = record
    FData: T;
  end;

type
```

```
  IAncestor<T> = interface
    function GetRecord: TRecord<T>;
  end;

  IFoo<T> = interface(IAncestor<T>)
    procedure AMethod(Param: T);
  end;

type
  TFoo<T> = class(TObject, IFoo<T>)
    FField: TRecord<T>;
    procedure AMethod(Param: T);
    function GetRecord: TRecord<T>;
  end;
```

프로시저 타입

프로시저 타입과 메소드 포인터는 타입 파라미터로 선언될 수 있습니다. 파라미터 타입과 결과 타입에서도 타입 파라미터를 사용할 수 있습니다.
예를 들면 다음과 같습니다.

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

type
  TFoo = class
    procedure Test;
    procedure MyProc(X, Y: Integer);
  end;

procedure Sample(Param: Integer);
begin
  WriteLn(Param);
end;

procedure TFoo.MyProc(X, Y: Integer);
begin
  WriteLn('X:', X, ', Y:', Y);
end;

procedure TFoo.Test;
var
  X: TMyProc<Integer>;
  Y: TMyProc2<Integer>;
begin
```

```
  X := Sample;
  X(10);
  Y := MyProc;
  Y(20, 30);
end;

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end.

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end.
```

파라미터화된 메소드

메소드를 타입 파라미터와 함께 선언할 수 있습니다. 파라미터 타입과 결과 타입에서 타입 파라미터를 사용할 수 있습니다. 파라미터화된 메소드는 오버로드된 메소드와 비슷합니다. 메소드를 인스턴스화하는 데에는 두 가지 방법이 있습니다.

- 명시적으로 타입 인자를 지정하는 방법
- 인자 타입으로부터 자동으로 추정하는 방법

예를 들면 다음과 같습니다.

```
type
  TMyProc2<Y> = procedure(Param1, Param2: Y) of object;
  TFoo = class
    procedure Test;
    procedure MyProc2<T>(X, Y: T);
  end;

procedure TFoo.MyProc2<T>(X, Y: T);
begin
  Write('MyProc2<T>');
  {$IFDEF CIL}
```

```
  Write(X.ToString);
  Write(', ');
  WriteLn(Y.ToString);
  {$ENDIF}
  WR
end;

procedure TFoo.Test;
var
  P: TMyProc2<Integer>;
begin
  MyProc2<String>('Hello', 'World');     //타입이 지정됨
  MyProc2('Hello', 'World');             //인자 타입으로부터 추정됨
  MyProc2<Integer>(10, 20);
  MyProc2(10, 20);
  P := MyProc2<Integer>;
  P(40, 50);
end;

var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end.
```

타입 파라미터의 유효 범위

타입 파라미터의 유효 범위는 타입 선언과 그 모든 멤버들의 바디를 포함하지만, 자손 타입은 포함되지 않습니다.

예를 들면 다음과 같습니다.

```
type
  TFoo<T> = class
    X: T;
  end;

  TBar<S> = class(TFoo<S>)
    Y: T;  // 에러!  unknown identifier "T"
  end;

var
  F: TFoo<Integer>;
begin
  F.T      // 에러! unknown identifier "T"
end.
```

제네릭에서의 오버로드와 타입 호환성

오버로드

제네릭 메소드도 'overload' 지시어를 사용하여 제네릭이 아닌 메소드들과 함께 오버로딩에 포함될 수 있습니다. 제네릭 메소드와 비 제네릭 메소드 사이의 오버로드 선택 문제가 모호하지 않으면 컴파일러는 비 제네릭 메소드를 선택합니다.
예를 들면 다음과 같습니다.

```
type
  TFoo = class
    procedure Proc<T>(A: T);
    procedure Proc(A: String);
    procedure Test;
  end;

procedure TFoo.Test;
begin
  Proc('Hello');            // Proc(A: String)을 호출
  Proc<String>('Hello');    // Proc<T>(A: T)을 호출
end;
```

타입 호환성

두 인스턴스화되지 않은 제네릭들이 대입 호환되는 경우는 동등하거나 공통 타입에 대한 별칭(alias)인 경우 뿐입니다.
두 인스턴스화된 제네릭들은 기반 타입이 동등 (혹은 공통 타입에 대한 별칭임) 하고 타입 인자가 동등한 경우 대입 호환됩니다.

제네릭의 제약조건

제네릭의 타입 파라미터에 제약조건을 연계시킬 수 있습니다. 제약조건(constraint)은 제네릭 타입의 구축(construction) 시에 파라미터로 넘겨질 모든 구체적 타입들이 지원해야 하는 아이템들을 선언합니다.

제네릭에서 제약조건의 지정

제약조건 항목들은 다음과 같은 것들을 포함합니다.

- 없거나, 하나 이상의 인터페이스 타입
- 없거나 하나의 클래스 타입
- 예약어 "constructor", "class", 혹은 "record"

제약조건으로서 "constructor" 와 "class"를 모두 지정할 수 있습니다. 하지만 "record"는 다른 예약어들과 같이 쓰일 수 없습니다. 여러 제약조건을 함께 사용하면 덧셈 합으로 동작합니다(AND 로직).
여기서 예제들에서는 클래스 타입들만을 보여주지만, 제약조건은 모든 제네릭 형태에 적용됩니다.

■ 제약조건의 선언

제약조건은 일반적인 파라미터 리스트 내의 타입 선언과 비슷한 방식으로 선언됩니다.

```
type
  TFoo<T: ISerializable> = class
    FField: T;
  end;
```

위의 예제에서 'T' 타입 파라미터는 ISerializable 인터페이스를 반드시 지원해야 한다는 것을 명시하고 있습니다. TFoo〈TMyClass〉와 같이 타입을 구축할 때, 컴파일러는 TMyClass가 실질적으로 ISerializable을 지원하는지 컴파일 중에 확인합니다.

■ 복수의 타입 파라미터

제약조건을 지정할 때는 파라미터 리스트 선언에서처럼 세미콜론으로 복수의 타입 파라미터들을 분리합니다.

```
type
   TFoo<T: ISerializable; V: IComparable>
```

파라미터 선언에서처럼 복수의 타입 파라미터들은 같은 제약조건에 대해 콤마 리스트로 그룹을 묶을 수 있습니다.

```
type
   TFoo<S, U: ISerializable> ...
```

위의 예에서, S 및 U는 둘 다 ISerializable 제약조건으로 묶입니다.

■ 복수의 제약조건

단일 타입 파라미터에 콜론(:) 뒤에 콤마 리스트 형식으로 지정하여 복수의 제약조건을 적용할 수 있습니다.

```
type
  TFoo<T: ISerializable, ICloneable; V: IComparable> ...
```

제약조건이 지정된 타입 파라미터를 "자유로운" 타입 파라미터와 혼합하여 사용할 수 있습니다. 다음의 모든 코드는 적합합니다.

```
type
  TFoo<T; C: IComparable> ...
  TBar<T, V> ...
  TTest<S: ISerializable; V> ...
  // T와 V는 자유롭지만 C와 S는 제약조건이 지정되어 있음
```

제약조건의 종류

■ 인터페이스 타입 제약조건

타입 파라미터 제약조건은 0개, 하나 혹은 콤마로 구분된 복수의 인터페이스 타입들의 리스트를 포함할 수 있습니다.

인터페이스 타입으로 타입 파라미터 제약조건을 지정하는 것은, 컴파일러가 컴파일 중에 타입 구축(construction)에서 인자로 넘겨진 구체적 타입이 지정한 인터페이스 타입을 구현하는지 검사한다는 의미입니다.

예를 들면 다음과 같습니다.

```
type
  TFoo<T: ICloneable> ...

  TTest1 = class(TObject, ICloneable)
     ...
  end;

  TError = class
```

```
  end;

var
  X: TFoo<TTest1>;  //  TTest1이 ICloneable을 지원하는지 컴파일 중에 확인됩니다
  Y: TFoo<TError>;  // 문법 에러 ? TError는 ICloneable을 지원하지 않습니다
```

■ 클래스 타입 제약조건

타입 파라미터는 0개 혹은 하나의 클래스 타입의 제약조건을 가질 수 있습니다. 인터페이스 타입 제약조건에서와 마찬가지로, 이 선언은 컴파일러가 컴파일 중에 타입 구축(construction)에서 인자로 넘겨진 구체적 타입이 지정한 제약조건 클래스와 대입 호환되는지 검사한다는 의미입니다. 클래스 타입들의 호환성은 일반적인 OOP 타입 호환성의 규칙을 따릅니다. 조상 타입이 필요한 곳에 자손 타입을 넘길 수 있습니다.

■ 생성자 제약조건

타입 파라미터는 0개 혹은 하나의 예약어 "constructor" 제약조건을 가질 수 있습니다. 이것은 실제 인자 타입이 기본 생성자(파라미터가 없는 public 생성자)를 정의한 클래스여야 한다는 것을 의미하며, 따라서 인자 타입 자체에 대해 아무것도 모르더라도(최소한의 기반 타입 요구사항도 필요하지 않음) 제네릭 타입 내의 메소드가 인자 타입의 기본 생성자를 사용하여 인자 타입의 인스턴스를 구축할 수 있게 됩니다.
제약조건 선언에서는 "constructor"를 인터페이스나 클래스 타입 제약조건과 어떤 순서로든 섞어 사용할 수 있습니다.

■ 클래스 제약조건

타입 파라미터 리스트는 0개 혹은 하나의 예약어 "class" 제약조건을 가질 수 있습니다. 이것은 실제 타입이 참조 타입, 즉 클래스나 인터페이스 타입이어야 한다는 것을 의미합니다.

■ 레코드 제약조건

타입 파라미터는 0개 혹은 하나의 예약어 "record" 제약조건을 가질 수 있습니다. 이것은 실제 타입이 값 타입(참조 타입은 불가)이어야 한다는 것을 의미합니다. "record" 제약조건은 "class"나 "constructor" 제약조건과 같이 쓰일 수 없습니다.

■ 타입 추정

제약조건을 가진 타입 파라미터의 필드나 변수를 사용할 때는 그 필드나 변수를 제약조건을

가진 타입들 중 하나로 타입 캐스트할 필요가 없는 경우가 많습니다. 컴파일러는 해당 타입에 대한 모든 제약조건들에 걸쳐 메소드 이름을 살펴보거나 같은 이름을 가진 여러 메소드들에 대해 오버로드 해석을 변화시켜봄으로써 코드에서 참조하는 것이 어떤 타입인지 추정할 수 있습니다.
예를 들면 다음과 같습니다.

```
type
  TFoo<T: ISerializable, ICloneable> = class
    FData: T;
    procedure Test;
  end;

procedure TFoo<T>.Test;
begin
  FData.Clone;
end;
```

FData가 T 타입이고, 이 T 타입은 두 인터페이스 모두 지원하는 것으로 보장되므로, 컴파일러는 ISerializable 및 ICloneable에서 "Clone" 메소드를 찾아봅니다. 두 인터페이스가 모두 동일한 파라미터 리스트를 가진 "Clone"을 구현할 경우, 컴파일러는 "모호한 메소드 호출 에러(ambiguous method call error)"를 내며, 모호함을 없애기 위해 두 인터페이스 중 하나로 타입 캐스트할 것을 요구합니다.

제네릭 내의 클래스 변수

제네릭 타입 내에서 정의된 클래스 변수는 타입 파라미터에 의해 타입이 인스턴스화될 때마다 인스턴스화됩니다.
다음의 코드는 TFoo〈Integer〉.FCount 및 TFoo〈String〉.FCount가 단 한번 인스턴스화되고 이들은 서로 다른 변수라는 것을 보여줍니다.

```
{$APPTYPE CONSOLE}
type
  TFoo<T> = class
    class var FCount: Integer;
    constructor Create;
  end;
```

```
  constructor TFoo<T>.Create;
begin
  inherited Create;
  Inc(FCount);
end;

procedure Test;
  FI: TFoo<Integer>;
begin
  FI := TFoo<Integer>.Create;
  FI.Free;
end;

var
  FI: TFoo<Integer>;
  FS: TFoo<String>;
begin
  FI := TFoo<Integer>.Create;
  FI.Free;
  FS := TFoo<String>.Create;
  FS.Free;
  Test;
  WriteLn(TFoo<Integer>.FCount);  // 2 출력
  WriteLn(TFoo<String>.FCount);   // 1 출력
end;
```

표준함수와 문법의 변경 사항

다음은 제네릭을 지원하기 위한 표준 함수의 변경 사항들입니다.
예제는 다음과 같은 형식입니다.

```
인스턴스화된 타입 : TFoo<Integer,String>
열린 구축된 타입  : TFoo<Integer,T>
제네릭 타입       : TFoo<,>
타입 파라미터     : T
```

표준 함수들:

```
procedure Initialize(var X);                                    [.NET]
procedure Initialize(var X; Count:Cardinal);                    [Win32]
```

```
procedure Finalize(var X);                                          [Win32]
```
인스턴스화된 타입 : 허용
열린 구축된 타입 : 허용
제네릭 타입 : 허용되지 않음
타입 파라미터 : 허용

```
function High(X:TypeId): Integer|Int64|UInt64;
function Low(X:TypeId): Integer|Int64|UInt64;
```
인스턴스화된 타입 : 허용
열린 구축된 타입 : 허용되지 않음
제네릭 타입 : 허용되지 않음
타입 파라미터 : 허용되지 않음

```
function Default(X:TYPE_ID): valueOfTypeId>;
```
인스턴스화된 타입 : 허용
열린 구축된 타입 : 허용
제네릭 타입 : 허용되지 않음
타입 파라미터 : 허용

```
function New;
// a := New(array[2,3] of Integer);                                  [.NET]
// a := New(array[,], (const array init expr));                      [.NET]
// a := New(dynArrayTypeId, dim1 [, dim2...dimN]);                   [.NET]
// New( PtrVar );                                                   [Win32]
// New( PtrToRecordOrObjectVar,<CtorIdent>  );                      [Win32]
// New( PtrToRecordOrObjectVar, <CtorIdent> ( CtorParams ) );       [Win32]
// P := New( PtrType );                                             [Win32]
// P := New( PtrTypeOfRecordOrObject, <CtorIdent> );                [Win32]
// P := New( PtrTypeOfRecordOrObject, <CtorIdent> ( CtorParams ) ); [Win32]
```
인스턴스화된 타입 : 허용
열린 구축된 타입 : 허용되지 않음
제네릭 타입 : 허용되지 않음
타입 파라미터 : 허용되지 않음

```
function SizeOf(TYPE_ID): PosInt;
```
인스턴스화된 타입 : 허용
열린 구축된 타입 : 허용
제네릭 타입 : 허용되지 않음
타입 파라미터 : 허용

```
function TypeInfo;
function TypeHandle;
function TypeId;
// function TypeHandle(Identifier): System.RuntimeTypeHandle; [.NET]
// function TypeHandle(Identifier): Pointer;                  [Win32]
// function TypeInfo(Identifier): System.Type;                [.NET]
// function TypeInfo(Identifier): Pointer;                    [Win32]
// function TypeOf(Identifier): System.Type;                  [.NET]
// function TypeOf(object): Pointer;                          [Win32]
```

```
인스턴스화된 타입 : 허용
열린 구축된 타입 : 허용
제네릭 타입    : 허용되지 않음
타입 파라미터  : 허용
```

델파이 언어 문법 변경 사항

아래의 변경 사항들은 제네릭 혹은 제네릭 타입을 지원하기 위한 것들입니다.

```
{ Type Declarations }

  TypeDeclaration -> [ CAttrs ] Ident '=' Type
                  -> [ CAttrs ] Ident '=' RecordTypeDecl
                  -> [ CAttrs ] Ident '=' ClassTypeDecl
                  -> [ CAttrs ] Ident '=' InterfaceTypeDecl
                  -> [ CAttrs ] Ident '=' ClassHelperTypeDecl
                  -> [ CAttrs ] Ident '=' RecordHelperTypeDecl
                  -> [ CAttrs ] Ident '=' ObjectTypeDecl            {Win32 only}
                  -> [ CAttrs ] Ident '=' DispatchInterfaceType     {Win32 only}
                  -> [ CAttrs ] Ident '=' TYPE TypeId
                  -> [ CAttrs ] Ident '=' TYPE ClassTypeId           {.NET only}
  {NEW}           -> [ CAttrs ] Ident TypeParams '=' RecordTypeDecl
  {NEW}           -> [ CAttrs ] Ident TypeParams '=' ClassTypeDecl
  {NEW}           -> [ CAttrs ] Ident TypeParams '=' InterfaceTypeDecl
  {NEW}           -> [ CAttrs ] Ident TypeParams '=' Type

  {NEW} TypeParams -> '<' TypeParamDeclList '>'

  {NEW} TypeParamDeclList -> TypeParamDecl/';'...

  {NEW} TypeParamDecl -> TypeParamList [ ':' ConstraintList ]

  {NEW} TypeParamList -> ( [ CAttrs ] [ '+' | '-' [ CAttrs ] ] Ident )/','...

  {NEW} ConstraintList -> Constraint/','...

  {NEW} Constraint -> CONSTRUCTOR
  {NEW}            -> RECORD
  {NEW}            -> CLASS
  {NEW}            -> TypeId

  MethodResolutionClause -> FUNCTION InterfaceIdent '.'
  {OLD}                     Ident               '=' Ident              ';'
  {NEW}                     Ident [ TypeArgs ]  '=' Ident [ TypeArgs ] ';'
                         -> PROCEDURE InterfaceIdent '.'
  {OLD}                     Ident               '=' Ident              ';'
```

```
  {NEW}                     Ident [ TypeArgs ]  '=' Ident [ TypeArgs ] ';'

  FunctionHeading -> [ CLASS ] FUNCTION Ident
  {NEW}              [ FormalTypeParamList ]
                     [ FormalParameterList ] ':' TypeIdStringFile

  ProcedureHeading -> [ CALSS ] PROCEDURE Ident
  {NEW}                [ FormalTypeParamList ]
                       [ FormalParameterList ]

  ClassOperatorHeading -> CLASS OPERATOR OperatorIdent
  {NEW}                     [ FormalTypeParamList ]
                            FormalParameterList : TypeIdStringFile

  ConstructorHeading -> CONSTRUCTOR Ident
  {NEW}                 [ FormalTypeParamList ]
                        [ FormalParameterList ]

  RecordConstructorHeading -> CONSTRUCTOR Ident
  {NEW}                         [ FormalTypeParamList ]
                                FormalParameterList

  DestructorHeading -> DESTRUCTOR Ident
  {NEW}                [ FormalTypeParamList ]
                       [ FormalParameterList ]

  MethodBodyHeading -> [ CLASS ] FUNCTION NSTypeId '.' Ident
  {NEW}                [ FormalTypeParamList ]
                       [ FormalParameterList ] ':' TypeIdStringFile
                    -> [ CLASS ] PROCEDURE NSTypeId '.' Ident
  {NEW}                [ FormalTypeParamList ]
                       [ FormalParameterList ]

  ProcedureTypeHeading -> PROCEDURE
  {NEW}                   [ FormalTypeParamList ]
                          [ FormalParameterList ]

  FunctionTypeHeading -> FUNCTION
  {NEW}                  [ FormalTypeParamList ]
                         [ FormalParameterList ] ':' TypeIdStringFile

  FormalTypeParamList -> '<' TypeParamDeclList >'


{ Types }

  Type -> TypeId
       -> SimpleType
       -> StructualType
       -> PointerType
```

```
      -> StringType
      -> ProcedureType
      -> VariantType
  {Win32 only}
      -> ClassRefType
      -> TypeRefType
  {NEW}-> ClassTypeId TypeArgs
  {NEW}-> RecordTypeId TypeArgs
  {NEW}-> InterfaceIdent TypeArgs

  {NEW} TypeArgs -> '<' ( TypeId | STRING )/','... '>'


{ Attributes }

  CAttrExpr -> ConstExpr
            -> TYPEOF '(' TypeId ')'
  {NEW}     -> TYPEOF '(' TypeId '<' [ ','... ] '>' ')'
  {NEW}     -> TYPEOF '(' TypeId '<' TypeId/','... '>' ')'
```