제너릭

제너릭이라는 단어는 자바등의 언어에서는 들어보거나 사용해 보신 분들이계시지만 델파이 개발자들은 아직도 생소하거나 코드를 재대로 구현해 본적이 없는분들도 계십니다. 또한 기존의 소스를 수정하시는 것이 번거롭다고 생각하실 수 있습니다.

그래서 이번 자료에서는 델파이 2009버전부터 추가되어 사용되는 제너릭의 전반적인 개념과 델파이에서 사용되는 제너릭 클래스등을 소개해 보도록 하겠습니다.

제너릭이 무엇인지 설명하기 전에 델파이에서 저네릭을 사용하지 않았던 시점의 코드를 한 번 살펴 보도록 하겠습니다.

우리가 프로시저와 함수들을 선언 할 때 우리는 항상 프로시저, 함수 등의 매개변수 타입을 미리 선언하고 사용하였습니다.

|  |
| --- |
| function Add(x,y:integer):integer;  function Divide(x,y:real):real;  function Test(x,y:string):string; |

특히 dll 같은 외부 프로세스의 루틴을 호출하기 위해 프로시저, 함수 변수들을 따로 선언했었습니다.

|  |
| --- |
| Type  TAddFunc = function(x,y:integer):integer;  TDivFunc = function(x,y:real):real;  // TTestFunc = function(x,y:string):string;  Var  AddFunc:TAddFunc;  DivFunc:DivFunc;  TestFunc:function(x,y:string):string; //타입 선언하지 않고 변수를 직접 선언 할 수도 있음 |

또는 클래스를 정의할 때 다음과 같이 선언 합니다.

|  |
| --- |
| TSIPare = class  Private  FKey :String;  FValue :Integer;  Public  Function GetKey:String;  Procedure Setkey(Key:string);  Function GetValue:Integer;  Procedure SetValue(Value:Integer);  Property Key:string read GetKey write SetKey;  Property Value:integer read GetValue write SetValue;  end; |

만일string,string 또는 integer, integer, integer,string 을 타입으로 갖는 클래스가 각각 필요하다면

우리는 코드를 복사해서 TSSPair, TIIPair. TISPare 클래스들은 선언할 것입니다.

배열을 사용하고 데이터를 입력할 경우 다음과 같이 코드를 구현하였습니다.

|  |
| --- |
| Type  TCountry = array[0..2] of String; //정적배열  Var  Countires:Country =(‘한국’, ‘일본’, ‘미국’)  TestLists:array of string; //동적배열  IA:array of integer;  Procedure TForm1.Button1Click(Sender:TObject);  Var  I:byte;  Begin  SetLength(TestLists,3);  TestLists[0] := ‘a’;  TestLists[0] := ‘b’;  TestLists[0] := ‘c’;  SetLength(IA,10);  For i:= 0 to 9 do  Begin  IA[i] := i;  End;  End; |

위의 예제에서 다른 타입의 데이터를 처리하는 배열이 필요하다면 여러분들은 다시 배열타입이나 변수를 선언하고 데이터를 처리 할 것 입니다.

**제너릭이란?**

제네릭(generic)이란 데이터의 타입(data type)을 일반화한다(generalize)는 것을 의미합니다.

클래스나 메소드를 정의할 때, 구체적인 타입(type)을 적지 않고 변수형태로 적어 놓는 것이다. 클래스 생성이나 메소드 호출 시점에 정함으로써 자료 형에 대한 안정성을 높이는 기능을 말합니다. 이때 클래스,메소드에서의 자료 형은 사용자가 스스로 선택할 수 있도록 범용적인 상태로 정하게 됩니다.

앞에서 제너릭을 사용하지 않은 예제를 제너릭을 사용하여 수정 해 보도록 하겠습니다.

* 제너릭을 사용한 메소드 호출

|  |
| --- |
| Type  TCalcFunc<T> = function(x,y:T):T;    Procedure TForm1.Button1Click(Sender:TObject);  Var  CalcFunc:TCalcFunc<integer>;  Begin  CalcFunc := Add;  ShowMessage(IntToStr(CalcFunc(2,3));  end; |

* 제너릭 클래스

|  |
| --- |
| Type  TPair<TKey,TValue> = class  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;  TSIPair = TPair<String,Integer>;  TSSPair = TPair<String,String>;  TIIPair = TPair<Integer,Integer>;  TISPair = TPair<Integer,String>; |

제네릭 클래스(Genenric class)에서는 타입을 변수로 표시한다. 이것을 “타입 매개변수(type parameter)”라고 하며, 타입 매개변수는 개체 생성 시에 프로그래머에 의하여 결정 됩니다.

* 제네릭 컬렉션 클래스

언어에 제네릭 타입(들)이 추가되면서 델파이에는 제네릭 컬렉션(들) 기본 세트가 추가되었습니다. 이 세트는 System.Generics.Collections 유닛에 있으며, 여러분이 담으려고 하는 특정 데이터 타입에 연결된 컨테이너(들)을 위한 제네릭한 청사진을 제시합니다. 제네릭 컬렉션(들)에는 다음과 같은 것들이 있습니다:

* TList<T>, 기본 제네릭 리스트로 정렬과 열거(enumeration) 지원합니다.
* TThreadList<T>, 쓰래드에 안전한 리스트로서 잠금(locking)을 지원합니다.
* TQueue<T>, TStack<T>
* TArray<T>
* TDictionary<TKey,TValue>, 상당히 막강한 딕셔너리이며, 키(key)와 값(value)의 타입을 필요에 맞게 지정할 수 있습니다.
* TObjectList<T: class> 객체 타입을 담을 수 있으며 소유권을 지원합니다.(아래의 컨테이너들도 동일)
* TObjectQueue<T: class>, TObjectStack<T: class>
* TObjectDictionary<TKey,TValue>
* TThreadedQueue<T>

아래 예제는 앞에 배열 예제를 TArray를 사용하여 수정하여 보겠습니다.

|  |
| --- |
| Type  TArrayEx = class(TArray)  public  class function CloneArray<T>(const A: array of T): TArray<T>;  end;  { TArrayEx }  Implementation  class function TArrayEx.CloneArray<T>(const A: array of T): TArray<T>;  var  Idx: Integer;  begin  SetLength(Result, Length(A));  for Idx := Low(A) to High(A) do  Result[Idx - Low(A)] := A[Idx];  end;  procedure TForm1.Button2Click(Sender: TObject);  var  IA: TArray<Integer>;  SA: TArray<string>;  FA:TArray<real>;  begin  IA := TArrayEx.CloneArray<Integer>([0,1,2,3,5,6,7,8,9]);  SA := TArrayEx.CloneArray<string>(['a', 'b','c']);  FA := TArrayEx.CloneArray<real>([10.1,22.3]);  ShowMessage(IntToStr(IA[3])); |

* TDictionary 사용 예제

|  |
| --- |
| procedure TForm1.Button2Click(Sender: TObject);  begin  var MyDictionary := TDictionary<string, Integer>.Create;  MyDictionary.Add ('one', 1);  MyDictionary.Add ('two', 2);  var APair := MyDictionary.ExtractPair('two');  Button2.Caption := APair.Value.ToString;  end; |

* TList<t> 사용예제

|  |
| --- |
| procedure TForm1.Button3Click(Sender: TObject);  var  List: TList<Integer>;  FoundIndex: Integer;  begin  { Create a new List. }  List := TList<Integer>.Create;  { 몇 개의 데이터를 추가 }  List.AddRange([5, 1, 8, 2, 9, 14, 4, 5, 1]);  ShowMessage('첫번째1의 인덱스는 ' + IntToStr(List.IndexOf(1)));  ShowMessage('마지막1의 인덱스는 ' + IntToStr(List.LastIndexOf(1)));  ShowMessage('데이터에 100의 유무는? ' + BoolToStr(List.Contains(100)));  {데이터 추가 }  List.Add(100);  ShowMessage(IntToStr(List.Count) + '개의 데이터가 있습니다.');  { 첫번쨰 1 데이터 삭제 }  List.Remove(1);  { 첫번째 위치의 데이터 삭제 }  List.Delete(0);  List.DeleteRange(0, 2);  { 리스트에서 남아있는 1을 지움 }  List.Extract(1);  {실제 길이 구함 }  List.TrimExcess;  ShowMessage('리스트 항목 갯수는 ' + IntToStr(List.Capacity));  { 리스트 클리어 }  List.Clear;  { Insert some elements. }  List.Insert(0, 2);  List.Insert(1, 1);  List.InsertRange(0, [6, 3, 8, 10, 11]);  { 리스트 내용 소트 }  List.Sort;  { Binary search for the required element. }  if List.BinarySearch(6, FoundIndex) then  ShowMessage('6 데이터의 인덱스는 ' + IntToStr(FoundIndex));  { Reverse the list. }  List.Reverse;  ShowMessage('첫번째 데이터는 ' + IntToStr(List.Items[0]));  List.Free;  end; |

* TObjectList<T: class>

|  |
| --- |
| type  { Declare a new object type. }  TNewObject = class  private  FName: String;  public  constructor Create(const AName: String);  destructor Destroy; override;  end;  constructor TNewObject.Create(const AName: String);  begin  FName := AName;  end;  destructor TNewObject.Destroy;  begin  ShowMessage('오브젝트"' + FName + '" 해제되었습니다!');  inherited;  end;  procedure TForm1.Button4Click(Sender: TObject);  { TNewObject }  var  List: TObjectList<TNewObject>;  Obj: TNewObject;  begin  { Create a new List. }  { The OwnsObjects property is set by default to true -- the list will free the owned objects automatically. }  List := TObjectList<TNewObject>.Create;  { Add some items to the List. }  List.Add(TNewObject.Create('One'));  List.Add(TNewObject.Create('Two'));  { Add a new item, but keep the reference. }  Obj := TNewObject.Create('Three');  List.Add(Obj);  {  Remove an instance of the TNewObject class. The destructor  is called for the owned objects, because you have set the OwnsObjects  to true.  }  List.Delete(0);  List.Extract(Obj);  { Destroy the List completely -- more message boxes will be shown. }  List.Free;  end. |

위와 같이 제너릭을 사용하면 다음과 같은 장점이 있습니다.

제너릭 장점

제네릭스 컬렉션(들)을 사용하면 여러분이 사용하고자 하는 데이터 타입을 담을 수 있는 특정 컨테이너(들)을 정의할 수 있습니다. 그러면 해당 데이터 타입에 부합하는 지를 컴파일러가 확인하므로 실행 중에 데이터 타입을 확인하지 않아도 됩니다. 그 결과, 코드가 더 깔끔하고 읽기도 더 쉽습니다. 애플리케이션은 더 견고해지고, 실행 시점에 확인할 필요가 줄어들기 때문에 실행 속도도 빨라집니다:

참고:기존의 컬렉션과 제너릭 컬렉션은 정리해놓은 자료입니다 아래 주소를 참조하십시오.

<https://tech.devgear.co.kr/index.php?mid=delphi_news&search_keyword=%EC%A0%9C%EB%84%88%EB%A6%AD&search_target=title_content&doc>