Generics allow you to define the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept −

```
using System;
using System.Collections.Generic;

namespace GenericApplication {
   public class MyGenericArray<T> {
      private T[] array;

      public MyGenericArray(int size) {
         array = new T[size + 1];
      }
      public T getItem(int index) {
         return array[index];
      }
      public void setItem(int index, T value) {
         array[index] = value;
      }
   }
   class Tester {
      static void Main(string[] args) {

         //declaring an int array
         MyGenericArray<int> intArray = new MyGenericArray<int>(5);

         //setting values
         for (int c = 0; c < 5; c++) {
            intArray.setItem(c, c*5);
         }

         //retrieving the values
         for (int c = 0; c < 5; c++) {
            Console.Write(intArray.getItem(c) + " ");
         }

         Console.WriteLine();

         //declaring a character array
         MyGenericArray<char> charArray = new
MyGenericArray<char>(5);
```

```
         //setting values
         for (int c = 0; c < 5; c++) {
            charArray.setItem(c, (char)(c+97));
         }
         
         //retrieving the values
         for (int c = 0; c< 5; c++) {
            Console.Write(charArray.getItem(c) + " ");
         }
         Console.WriteLine();
         
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
0 5 10 15 20
a b c d e
```

Features of Generics

Generics is a technique that enriches your programs in the following ways −

- It helps you to maximize code reuse, type safety, and performance.
- You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in the *System.Collections.Generic* namespace. You may use these generic collection classes instead of the collection classes in the *System.Collections* namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- You may create generic classes constrained to enable access to methods on particular data types.
- You may get information on the types used in a generic data type at run-time by means of reflection.

Generic Methods

In the previous example, we have used a generic class; we can declare a generic method with a type parameter. The following program illustrates the concept −

```
using System;
using System.Collections.Generic;
```

```
namespace GenericMethodAppl {
   class Program {
      static void Swap<T>(ref T lhs, ref T rhs) {
         T temp;
         temp = lhs;
         lhs = rhs;
         rhs = temp;
      }
      static void Main(string[] args) {
         int a, b;
         char c, d;
         a = 10;
         b = 20;
         c = 'I';
         d = 'V';

         //display values before swap:
         Console.WriteLine("Int values before calling swap:");
         Console.WriteLine("a = {0}, b = {1}", a, b);
         Console.WriteLine("Char values before calling swap:");
         Console.WriteLine("c = {0}, d = {1}", c, d);

         //call swap
         Swap<int>(ref a, ref b);
         Swap<char>(ref c, ref d);

         //display values after swap:
         Console.WriteLine("Int values after calling swap:");
         Console.WriteLine("a = {0}, b = {1}", a, b);
         Console.WriteLine("Char values after calling swap:");
         Console.WriteLine("c = {0}, d = {1}", c, d);

         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

Generic Delegates

You can define a generic delegate with type parameters. For example −

```
delegate T NumberChanger<T>(T n);
```

The following example shows use of this delegate −

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl {
   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultNum(int q) {
         num *= q;
         return num;
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         //create delegate instances
         NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
         NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);

         //calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());

         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of Num: 35
Value of Num: 175
```

Previous Page Print Page
Next Page