GENERICS

Generics reduce the need to rewrite algorithms for each data type. **You can create generic classes, delegates, interfaces, and methods.** With each of these, you identify where data will change in the code segment by putting a placeholder in the code for the type parameters. A generic **class** might use placeholder(s) for the data type of its instance data members or placeholders for return types of one or more of its methods. After the generic **class** is defined, it could be instantiated using several different types of data. A generic method might use a generic type for one of its formal parameters and/or use a generic type as the type of its return value. Generic methods can be defined both as part of generic and other classes. First examine what is involved in defining a generic class.

Generic Classes

Prior to the introduction of generics, one way to write reusable code was to use the **object** data type for instance data members. You will recall that **object** is the base class from which all other classes are derived. After the data type is defined as an object, then through casting and unboxing the data type can be temporarily made to act like any of the other types. Example 11-25 illustrates defining a Stack **class** that could be used to store any type of data items. A stack represents a simple last-in-first-out (LIFO) collection. You can think of stacks as analogous to a pile of trays. The Push( ) method places a tray on the top of the others; the Pop() method retrieves the one on top. One way stacks are used by applications during execution is to store the address of calling methods so the application will know where to return when it finishes the called method.

Example 1 contains a simplified programmer-defined Stack **class**. The instance data member is an array of **object** items.

Example 1

```
public class Stack
{
    private object [ ] items;
    private int stackPointer = 0;
    public Stack(int size)
    {
         items = new object[size];
    }

    public object Pop( )
    {
         return items[--stackPointer];
    }
    public void Push (object anItem)
    {
         items[stackPointer] = anItem;
         stackPointer++;
    }
}
```

The Push( ) method is used to place items in the stack. You will notice that the **class** is implemented as an array. The first item added is placed at index location 0.The stackPointer is incremented by one after each item is placed in the stack. The Pop( ) method retrieves the last one placed in the array. Neither of these methods includes any testing to ensure that reference is not made beyond the array boundaries. Example 2

illustrates an application that instantiates an **object** of the Stack **class** and then pushes data of differing types onto the data structure. Finally, the items are retrieved using the Pop( ) method.

Example 2

```
public class TestOfStack
{
    static void Main(string[ ] args)
    {
        Stack stack = new Stack(5);
        stack.Push("test");
        stack.Push(100);
        stack.Push(200);
        stack.Push(3.6);
        Console.WriteLine( "Values in the Stack are: " +
            (double)stack.Pop( ) + ", " +
            (int)stack.Pop( ) + ", " +
            (int)stack.Pop( ) + ", " +
            (string)stack.Pop( ));
    }
}
```

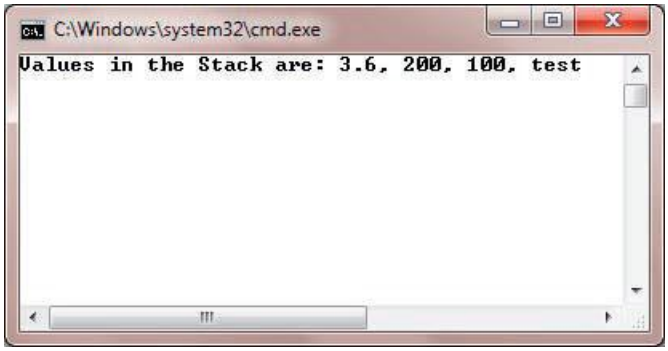

Figure 1

As illustrated in Examples 1 and 2, without generics you could define the data type as an object because it can hold any type. To retrieve the items, the last four lines in Example 2 show that casts had to be performed to unbox or extract the values from the **object** type. In Example 2, because the last item placed on the Stack was a **string**, the first call to Pop( ) retrieved the string. The program would have terminated abnormally if the order of the casting was changed.

This object-based approach illustrated is not type-safe. Data of **string** type could be pushed onto the same stack that also holds integers. When the data is retrieved, if an attempt is made to do arithmetic with the **string** items, the program will terminate abnormally. To avoid this problem, you could define three separate stack classes, one for integers, another for doubles, and a third one for strings. This could be done by copying and pasting the code. But, if a simple modification is needed, corrections would need to be made to all three classes. This is where generics come into play. They allow you to define type-safe, compiler-generated code that can be defined and consumed with any type. The internal algorithms remain the same, but the **class** can be defined once and reused. Example 3 illustrates defining a simple generic **class**.

Example 3

```
public class GenericClass<T>
  {
        public T dataMember;
  }
```

Notice that a generic **class** is defined by inserting an identifier between left and right brackets on the class definition line. In Example 3, T is used as the identifier. T is the standard identifier used to define a generic type in most of Microsoft's documentation; however, you could use any valid identifier as long as you adhered to the rules for creating an identifier. In the body of the method, the identifier is placed where the data type should be inserted. It can be as a return type in a method, in the parameter list of a method, or as shown in Example 3 as the data type for instance data members. When you create an instance of the class, you specify the actual type to substitute for the type parameters. If the dataMember was defined as **public** (or if a **public** property was available for the dataMember), the **class** could be used as shown in Example 4.

Example 4

```
GenericClass<string> anIdentifer = new GenericClass<string>();
anIdentifer.dataMember = "A string";
```

You can take advantage of this new feature and redefine the Stack **class** as a generic **class**. This is illustrated in Example 5.

Example 5

```
public class Stack<T>
{
        private T[ ] items;
        private int stackPointer = 0;
        public Stack(int size)
        {
            items = new T [size];
        }
        public T Pop( )
        {
            return items[– –stackPointer];
    }
        public void Push(T anItem)
        {
       items[stackPointer] = anItem;
            stackPointer++;
        }
}
```

Example 6 illustrates the changes needed to implement the **class**. Notice three separate objects are instantiated. The only values that can be placed in the **int** Stack are integers. Thus, a type-safe data structure has been created.

Example 6

```
public class TestOfGenericStack
    {
        static void Main(string[ ] args)
        {
            Stack<int> intStack = new Stack<int>( 5 );
            Stack<double> doubleStack = new Stack<double>( 5 );
            Stack<string> stringStack = new Stack<string>( 5 );
            stringStack.Push("test");
            intStack.Push(100);
            intStack.Push(200);
            doubleStack.Push(3.6);
            Console.WriteLine("Values in the 3 Stacks are: " +
                doubleStack.Pop( ) + ", " +
                intStack.Pop( ) + ", " +
                intStack.Pop( ) + ", " +
                stringStack.Pop( ) );
            }
}
```

Notice no casting was necessary in Example 6. The output produced from Examples 5 and 6 is the same as what was shown in Figure 1.

Generic Methods

You can define generic methods that are not part of a generic **class**. This way, the method defers the specification of one or more types until the method is declared and instantiated by client code. Defining a generic method is similar to defining a generic class. You insert an identifier between the left and right brackets on the method definition line to indicate it is a generic method. Then place that identifier either in the parameter list or as a return type or in both places. This is illustrated in Example 7.

Example 7

```
public static void SwapData<T> (ref T first, ref T second)
{
         T temp;
         temp = first;
         first = second;
         second = temp;

}
```

A call to the SwapData( ) method is shown in Example 8.

Example 8

```
public static void TestSwap( )
{
      string firstValue = "Programming";
      string secondValue = "C#";
      SwapData<string>(ref firstValue, ref secondValue);
      System.Console.WriteLine(firstValue + " " + secondValue);
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GenericDelegate
{
    // This generic delegate can call any method
    // returning void and taking a single type parameter.
    public delegate void MyGenericDelegate<T>( T arg );

    class Program
    {
        static void Main( string[] args )
        {
            Console.WriteLine("***** Generic Delegates *****\n");

            // Register targets.
            MyGenericDelegate<string> strTarget =
              new MyGenericDelegate<string>(StringTarget);
            strTarget("Some string data");

            MyGenericDelegate<int> intTarget =
              new MyGenericDelegate<int>(IntTarget);
            intTarget(9);
            Console.ReadLine();
        }

        static void StringTarget( string arg )
        {
            Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
        }

        static void IntTarget( int arg )
        {
            Console.WriteLine("++arg is: {0}", ++arg);
        }
    }
}
```