Chapter 20 Generics

Visual C# 2012 How to Program

OBJECTIVES

In this chapter you'll:

- Create generic methods that perform identical tasks on arguments of different types.
- Create a generic `Stack` class that can be used to store objects of most types.
- Understand how to overload generic methods with nongeneric methods or with other generic methods.
- Understand the kinds of constraints that can be applied to a type parameter.
- Apply multiple constraints to a type parameter.

20.2 Motivation for Generic Methods

- Overloaded methods are often used to perform similar operations on different types of data.
- To understand the motivation for generic methods, let’s begin with an example (Fig. 20.1) that contains three overloaded `DisplayArray` methods (lines 23–29, lines 32–38 and lines 41–47).
- These methods display the elements of an int array, a `double` array and a `char` array, respectively.
- Soon, we’ll reimplement this program more concisely and elegantly using a single generic method

```
// Fig. 20.1: OverloadedMethods.cs
// Using overloaded methods to display arrays of different types.
using System;

class OverloadedMethods
{
   static void Main( string[] args )
   {
      // create arrays of int, double and char
      int[] intArray = { 1, 2, 3, 4, 5, 6 };
      double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      char[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      Console.WriteLine( "Array intArray contains:" );
      DisplayArray( intArray ); // pass an int array argument
      Console.WriteLine( "Array doubleArray contains:" );
      DisplayArray( doubleArray ); // pass a double array argument
      Console.WriteLine( "Array charArray contains:" );
      DisplayArray( charArray ); // pass a char array argument
   } // end Main

```

Fig. 20.1 | Using overloaded methods to display arrays of different types. (Part 1 of 3.)

```
   // output int array
   private static void DisplayArray( int[] inputArray )
   {
      foreach ( int element in inputArray )
         Console.Write( element + " " );

      Console.WriteLine( "\n" );
   } // end method DisplayArray

   // output double array
   private static void DisplayArray( double[] inputArray )
   {
      foreach ( double element in inputArray )
         Console.Write( element + " " );

      Console.WriteLine( "\n" );
   } // end method DisplayArray
```

Fig. 20.1 | Using overloaded methods to display arrays of different types. (Part 2 of 3.)

```
   // output char array
   private static void DisplayArray( char[] inputArray )
   {
      foreach ( char element in inputArray )
         Console.Write( element + " " );

      Console.WriteLine( "\n" );
   } // end method DisplayArray
} // end class OverloadedMethods
```

```
Array intArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:
H E L L O
```

Fig. 20.1 | Using overloaded methods to display arrays of different types. (Part 3 of 3.)

20.3 Generic-Method Implementation

- Figure 20.3 reimplements the app of Fig. 20.1 using a generic `DisplayArray` method (lines 24–30).
- Note that the `DisplayArray` method calls in lines 16, 18 and 20 are identical to those of Fig. 20.1, the outputs of the two apps are identical and the code in Fig. 20.3 is 17 lines *shorter* than that in Fig. 20.1.
- As illustrated in Fig. 20.3, generics enable us to create and test our code once, then *reuse* it for many different types of data.
- This demonstrates the expressive power of generics.

```
private static void DisplayArray( T[] inputArray )
{
   foreach ( T element in inputArray )
      Console.Write( element + " " );

   Console.WriteLine( "\n" );
} // end method DisplayArray
```

Fig. 20.2 | `DisplayArray` method in which actual type names are replaced by convention with the generic name `T`. Again, this code will *not* compile.

```
// Fig. 20.3: GenericMethod.cs
// Using overloaded methods to display arrays of different types.
using System;
using System.Collections.Generic;

class GenericMethod
{
   public static void Main( string[] args )
   {
      // create arrays of int, double and char
      int[] intArray = { 1, 2, 3, 4, 5, 6 };
      double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      char[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      Console.WriteLine( "Array intArray contains:" );
      DisplayArray( intArray ); // pass an int array argument
      Console.WriteLine( "Array doubleArray contains:" );
      DisplayArray( doubleArray ); // pass a double array argument
      Console.WriteLine( "Array charArray contains:" );
      DisplayArray( charArray ); // pass a char array argument
   } // end Main

```

Fig. 20.3 | Using a generic method to display arrays of different types. (Part 1 of 2.)

```
    // output array of all types
    private static void DisplayArray< T >( T[] inputArray )
    {
       foreach ( T element in inputArray )
          Console.Write( element + " " );

       Console.WriteLine( "\n" );
    } // end method DisplayArray
} // end class GenericMethod
```

```
Array intArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:
H E L L O
```

Fig. 20.3 | Using a generic method to display arrays of different types. (Part 2 of 2.)

Common Programming Error 20.1

If you forget to include the type-parameter list when declaring a generic method, the compiler will not recognize the type-parameter names when they're encountered in the method. This results in compilation errors.

Good Programming Practice 20.1

It’s recommended that type parameters be specified as individual capital letters. Typically, a type parameter that represents the type of an element in an array (or other collection) is named `E` for “element” or `T` for “type.”

Common Programming Error 20.2

If the compiler cannot find a single nongeneric or generic method declaration that’s a best match for a method call, or if there are multiple best matches, a compilation error occurs.

20.4 Type Constraints

`IComparable<T>` Interface

- It's possible to compare two objects of the *same* type if that type implements the generic interface `IComparable<T>` (of namespace `System`).
- A benefit of implementing interface `IComparable<T>` is that `IComparable<T>` objects can be used with the *sorting* and *searching* methods of classes in the `System.Collections.Generic` namespace—we discuss those methods in Chapter 21.
- The structures in the Framework Class Library that correspond to the simple types *all* implement this interface.

20.4 Type Constraints (cont.)

Specifying Type Constraints

- Even though `IComparable` objects can be compared, they cannot be used with generic code by default, because not all types implement interface `IComparable<T>`.
- However, we can restrict the types that can be used with a generic method or class to ensure that they meet certain requirements.
- This feature—known as a type constraint—restricts the type of the argument supplied to a particular type parameter.

20.4 Type Constraints (cont.)

- Figure 20.4 declares method `Maximum` (lines 20–34) with a type constraint that requires each of the method’s arguments to be of type `IComparable<T>`.
- This restriction is important, because not all objects can be compared.
- However, all `IComparable<T>` objects are guaranteed to have a `CompareTo` method that can be used in method `Maximum` to determine the largest of its three arguments.

```
// Fig. 20.4: MaximumTest.cs
// Generic method Maximum returns the largest of three objects.
using System;

class MaximumTest
{
   public static void Main( string[] args )
   {
      Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
         3, 4, 5, Maximum( 3, 4, 5 ) );
      Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
         6.6, 8.8, 7.7, Maximum( 6.6, 8.8, 7.7 ) );
      Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
         "pear", "apple", "orange",
         Maximum( "pear", "apple", "orange" ) );
   } // end Main

   // generic function determines the
   // largest of the IComparable objects
   private static T Maximum< T >( T x, T y, T z )
      where T : IComparable< T >
   {
      T max = x; // assume x is initially the largest
```

Fig. 20.4 | Generic method `Maximum` returns the largest of three objects. (Part 1 of 2.)

```

      // compare y with max
      if ( y.CompareTo( max ) > 0 )
         max = y; // y is the largest so far

      // compare z with max
      if ( z.CompareTo( max ) > 0 )
         max = z; // z is the largest

      return max; // return largest object
   } // end method Maximum
} // end class MaximumTest
```

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Fig. 20.4 | Generic method `Maximum` returns the largest of three objects. (Part 2 of 2.)

20.4 Type Constraints (cont.)

- C# provides several kinds of type constraints.
- A class constraint indicates that the type argument must be an object of a specific base class or one of its subclasses.
- An interface constraint indicates that the type argument’s class must implement a specific interface.
- The type constraint in line 21 is an interface constraint, because `IComparable<T>` is an interface.

20.4 Type Constraints (cont.)

- You can specify that the type argument must be a reference type or a value type by using the reference-type constraint (`class`) or the value-type constraint (`struct`), respectively.
- Finally, you can specify a constructor `constraint—new()`—to indicate that the generic code can use operator new to create new objects of the type represented by the type parameter.

20.4 Type Constraints (cont.)

- If a type parameter is specified with a constructor constraint, the type argument’s class must provide a `public` parameterless or default constructor to ensure that objects of the class can be created without passing constructor arguments; otherwise, a compilation error occurs.
- It’s possible to apply multiple constraints to a type parameter.

20.4 Type Constraints (cont.)

- To do so, simply provide a comma-separated list of constraints in the `where` clause.
- If you have a class constraint, reference-type constraint or value-type constraint, it must be listed first—only one of these types of constraints can be used for each type parameter.
- Interface constraints (if any) are listed next.
- The constructor constraint is listed last (if there is one).