Generics

Source : [BOOK] How to programming C# 2012:
Chapter 20

```
// Fig. 20.1: overloadedMethods.cs
```

```
private static void DisplayArray(int[] inputArray)
{
      foreach (var element in inputArray)
        Console.Write($"{element} ");
      Console.WriteLine();
}

private static void DisplayArray(double[] inputArray)
{
      foreach (var element in inputArray)
        Console.Write($"{element} ");
      Console.WriteLine();
}
private static void DisplayArray(char[] inputArray)
{
      foreach (var element in inputArray)
        Console.Write($"{element} ");
      Console.WriteLine();
}
```

```
// Fig. 20.3: GenericMethod.cs

public static void Main(string[] args)
{
      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.Write("Array intArray contains: ");
      DisplayArray(intArray); // pass an int array argument
      Console.Write("Array doubleArray contains: ");
      DisplayArray(doubleArray); // pass a double array argument
      Console.Write("Array charArray contains: ");
      DisplayArray(charArray); // pass a char array argument
}
private static void DisplayArray<T>(T[] inputArray)
{
      foreach (var element in inputArray)
        Console.Write($"{element} ");
      Console.WriteLine();
}
```

Generic Methods

<T>: **Type parameter list**. It introduces **type names that are generic place holders** for real data types that will be **substituted at compile time**.

<T> tells the compiler this is a **generic type.**

Use T instead of var

ToString() method for all data types

Value VS. Reference Types

For value types, **compiler make different versions** of that method for each type

All reference types use the **same version** of the generic method

```
                    // Fig. 20.4
```

```
private static T Maximum<T>(T x, T y, T z) where T : IComparable<T>
{
      var max = x; // assume x is initially the largest
      // compare y with max
      if (y.CompareTo(max) > 0)
        max = y;


      if (z.CompareTo(max) > 0)
        max = z;

      return max; // return largest object
}
```

Type Constraints

Type parameter **represents any object** by default

- Not all objects have overloaded > operators
- Cannot do expressions like: Object1 > Object2
- Cannot call a method or access a property that’s not available for all objects

Icomparable<T>

Generic interface

Simple types all implements this interface as does **string**

Can compare objects of classes that **implement Icomparable<T>**

Many collections can sort and search for objects of such types

CompareTo method

```
private static T Maximum<T>(T x, T y, T z) where T : IComparable<T>
```

if T implements IComparable<T> => has CompareTo method

a.CompareTo(b)

- 0 if a and b are equal
- Negative if a is less than b
- Positive if b is less than a

Different type constraints

Class constraints: (**where T : class_name**)

- Type argument must be an **object of a specific base class** or one of its **subclasses**

Interface constraint: (**where T : interface_name**)

- Type argument's type must **implement** a specific interface

Reference-type constraint: class (`where T : class`)

- Type argument must be a reference type

Value type constraint: struct (`where T : struct, IComparable<T>`)

- Type argument must be a value type

Overloading generic methods

A generic method may be **overloaded**

Each overloaded method must have a **unique signature**

we could provide a second version of generic method DisplayArray with the additional parameters **lowIndex and highIndex**

A generic method can be **overloaded by non-generic methods** with the same method name

- best matched method

Generic Classes

Describing a class in a **type-independent manner**

We can then **instantiate type-specific versions of the generic class ->** software reusability

```
                    // Generic Stack Class
```

```
public class Stack<T>
{
   private int top; // location of the top element
   private T[] elements; // array that stores stack elements
   public Stack(int stackSize)
   {
      elements = new T[stackSize]; // create stackSize elements
      top = -1; // stack initially empty
   }
   public void Push(T pushValue)
   {
      if (top == elements.Length - 1) // stack is full
         Console.WriteLine("The stack is Full");
      else{
         ++top; // increment top
         elements[top] = pushValue;} // place pushValue on stack
   }
   public T Pop()
   {
      if (top == -1) // stack is empty
         Console.WriteLine("The stack is Empty, Cannot pop");
      --top; // decrement top
      return elements[top + 1]; // return top value
   }
}
```

```
                    // Stack Class Test
```

```
class StackTest
{
   private static double[] doubleElements =
   {5.5 ,4.4 ,3.3 ,2.2 ,1.1};
   private static int[] intElements =
   {10 ,9 ,8 ,7 ,6 ,5 ,4 ,3 ,2 ,1};
   private static Stack<double> doubleStack; // stack stores doubles
   private static Stack<int> intStack; // stack stores ints
   static void Main()
   {
      doubleStack = new Stack<double>(5); // stack of doubles
      intStack = new Stack<int>(10); // stack of ints
      TestPushDouble(); // push doubles onto doubleStack
      TestPopDouble(); // pop doubles from doubleStack
      TestPushInt(); // push ints onto intStack
      TestPopInt(); // pop ints from intStack
   }
   private static void TestPushDouble()
   {
         Console.WriteLine("\nPushing elements onto doubleStack");
         foreach (var element in doubleElements){
            Console.Write($"{element:F1} ");
            doubleStack.Push(element);} // push onto doubleStack
   }
```

```
                    // Stack Class Test
-------------------------------------------------------------------------
    private static void TestPopDouble()
    {
       Console.WriteLine("\nPopping elements from doubleStack");
       double popValue; // store element removed from stack
       while (doubleStack.top!=-1){
          popValue = doubleStack.Pop(); // pop from doubleStack
          Console.Write($"{popValue:F1} ");}
    }
    private static void TestPushInt()
    {
       Console.WriteLine("\nPushing elements onto intStack");
       foreach (var element in intElements){
          Console.Write($"{element} ");
          intStack.Push(element);} // push onto intStack
    }
    private static void TestPopInt()
    {
       Console.WriteLine("\nPopping elements from intStack");
       int popValue; // store element removed from stack
       // remove all elements from stack
       while (intStack.top!=-1){
          popValue = intStack.Pop(); // pop from intStack
          Console.Write($"{popValue:F1} ");}
    }
}
```