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 a specific type.
- 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.

20.1 Introduction

20.2 Motivation for Generic Methods

20.3 Generic-Method Implementation

20.4 Type Constraints

20.4.1 `IComparable<T>` Interface

20.4.2 Specifying Type Constraints

20.5 Overloading Generic Methods

20.6 Generic Classes

20.7 Wrap-Up

```
// 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.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
   }
```

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 (var element in inputArray)
      {
         Console.Write($"{element} ");
      }

      Console.WriteLine();
   }

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

      Console.WriteLine();
   }
```

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 (var element in inputArray)
      {
         Console.Write($"{element} ");
      }

      Console.WriteLine();
   }
}
```

```
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.)

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

   Console.WriteLine();
}
```

Fig. 20.2 | `DisplayArray` method in which actual type names have been replaced by convention with the generic name `T`.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

```
// Fig. 20.3: GenericMethod.cs
// Using a generic method to display arrays of different types.
using System;

class GenericMethod
{
   static void Main()
   {
      // 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.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
   }
```

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 (var element in inputArray)
      {
         Console.Write($"{element} ");
      }

      Console.WriteLine();
   }
}
```

```
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.

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.

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

class MaximumTest
{
    static void Main()
    {
        Console.WriteLine($"Maximum of 3, 4 and 5 is {Maximum(3, 4, 5)}\n");
        Console.WriteLine(
            $"Maximum of 6.6, 8.8 and 7.7 is {Maximum(6.6, 8.8, 7.7)}\n");
        Console.WriteLine("Maximum of pear, apple and orange is " +
            $"{Maximum("pear", "apple", "orange")}\n");
    }

```

Fig. 20.4 | Generic method `Maximum` returns the largest of three objects. (Part 1 of 3.)

```
   // generic function determines the
   // largest of the IComparable<T> objects
   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; // 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
   }
}
```

Fig. 20.4 | Generic method `Maximum` returns the largest of three objects. (Part 2 of 3.)

```
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 3 of 3.)

```
 1  // Fig. 20.5: Stack.cs
 2  // Generic class Stack.
 3  using System;
 4
 5  public class Stack<T>
 6  {
 7     private int top; // location of the top element
 8     private T[] elements; // array that stores stack elements
 9
10     // parameterless constructor creates a stack of the default size
11     public Stack()
12        : this(10) // default stack size
13     {
14        // empty constructor; calls constructor at line 18 to perform init
15     }
16
```

Fig. 20.5 | Generic class Stack. (Part 1 of 4.)

```
   // constructor creates a stack of the specified number of elements
   public Stack(int stackSize)
   {
      if (stackSize <= 0) // validate stackSize
      {
         throw new ArgumentException("Stack size must be positive.");
      }

      elements = new T[stackSize]; // create stackSize elements
      top = -1; // stack initially empty
   }

```

Fig. 20.5 | Generic class `Stack`. (Part 2 of 4.)

```
   // push element onto the stack; if unsuccessful,
   // throw FullStackException
   public void Push(T pushValue)
   {
      if (top == elements.Length - 1) // stack is full
      {
         throw new FullStackException(
            $"Stack is full, cannot push {pushValue}");
      }

      ++top; // increment top
      elements[top] = pushValue; // place pushValue on stack
   }
```

Fig. 20.5 | Generic class Stack. (Part 3 of 4.)

```

   // return the top element if not empty,
   // else throw EmptyStackException
   public T Pop()
   {
      if (top == -1) // stack is empty
      {
         throw new EmptyStackException("Stack is empty, cannot pop");
      }

      --top; // decrement top
      return elements[top + 1]; // return top value
   }
}
```

Fig. 20.5 | Generic class `Stack`. (Part 4 of 4.)

```csharp
// Fig. 20.6: FullStackException.cs
// FullStackException indicates a stack is full.
using System;

public class FullStackException : Exception
{
   // parameterless constructor
   public FullStackException() : base("Stack is full")
   {
      // empty constructor
   }

   // one-parameter constructor
   public FullStackException(string exception) : base(exception)
   {
      // empty constructor
   }
```

Fig. 20.6 | `FullStackException` indicates a stack is full. (Part 1 of 2.)

```

   // two-parameter constructor
   public FullStackException(string exception, Exception inner)
      : base(exception, inner)
   {
      // empty constructor
   }
}
```

Fig. 20.6 | `FullStackException` indicates a stack is full. (Part 2 of 2.)

```
// Fig. 20.7: EmptyStackException.cs
// EmptyStackException indicates a stack is empty.
using System;

public class EmptyStackException : Exception
{
   // parameterless constructor
   public EmptyStackException() : base("Stack is empty")
   {
      // empty constructor
   }

   // one-parameter constructor
   public EmptyStackException(string exception) : base(exception)
   {
      // empty constructor
   }

```

Fig. 20.7 | `EmptyStackException` indicates a stack is empty. (Part 1 of 2.)

```
   // two-parameter constructor
   public EmptyStackException(string exception, Exception inner)
      : base(exception, inner)
   {
      // empty constructor
   }
}
```

Fig. 20.7 | `EmptyStackException` indicates a stack is empty. (Part 2 of 2.)

```
// Fig. 20.8: StackTest.cs
// Testing generic class Stack.
using System;

class StackTest
{
   // create arrays of doubles and ints
   private static double[] doubleElements =
      {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
   private static int[] intElements =
      {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

   private static Stack<double> doubleStack; // stack stores doubles
   private static Stack<int> intStack; // stack stores ints

```

Fig. 20.8 | Testing generic class Stack. (Part 1 of 8.)

```
   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
   }

```

Fig. 20.8 | Testing generic class Stack. (Part 2 of 8.)

```
   // test Push method with doubleStack
   private static void TestPushDouble()
   {
      // push elements onto stack
      try
      {
         Console.WriteLine("\nPushing elements onto doubleStack");

         // push elements onto stack
         foreach (var element in doubleElements)
         {
            Console.Write($"{element:F1} ");
            doubleStack.Push(element); // push onto doubleStack
         }
      }
      catch (FullStackException exception)
      {
         Console.Error.WriteLine($"\nMessage: {exception.Message}");
         Console.Error.WriteLine(exception.StackTrace);
      }
   }

```

Fig. 20.8 | Testing generic class Stack. (Part 3 of 8.)

```
      // test Pop method with doubleStack
      private static void TestPopDouble()
      {
         // pop elements from stack
         try
         {
            Console.WriteLine("\nPopping elements from doubleStack");

            double popValue; // store element removed from stack

            // remove all elements from stack
            while (true)
            {
               popValue = doubleStack.Pop(); // pop from doubleStack
               Console.Write($"{popValue:F1} ");
            }
         }
         catch (EmptyStackException exception)
         {
            Console.Error.WriteLine($"\nMessage: {exception.Message}");
            Console.Error.WriteLine(exception.StackTrace);
         }
      }

```

Fig. 20.8 | Testing generic class Stack. (Part 4 of 8.)

```
   // test Push method with intStack
   private static void TestPushInt()
   {
      // push elements onto stack
      try
      {
         Console.WriteLine("\nPushing elements onto intStack");

         // push elements onto stack
         foreach (var element in intElements)
         {
            Console.Write($"{element} ");
            intStack.Push(element); // push onto intStack
         }
      }
      catch (FullStackException exception)
      {
         Console.Error.WriteLine($"\nMessage: {exception.Message}");
         Console.Error.WriteLine(exception.StackTrace);
      }
   }

```

Fig. 20.8 | Testing generic class Stack. (Part 5 of 8.)

```
   // test Pop method with intStack
   private static void TestPopInt()
   {
      // pop elements from stack
      try
      {
         Console.WriteLine("\nPopping elements from intStack");

         int popValue; // store element removed from stack

         // remove all elements from stack
         while (true)
         {
            popValue = intStack.Pop(); // pop from intStack
            Console.Write($"{popValue:F1} ");
         }
      }
      catch (EmptyStackException exception)
      {
         Console.Error.WriteLine($"\nMessage: {exception.Message}");
         Console.Error.WriteLine(exception.StackTrace);
      }
   }
}
```

Fig. 20.8 | Testing generic class Stack. (Part 6 of 8.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
Message: Stack is full, cannot push 6.6
   at Stack`1.Push(T pushValue) in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\Stack.cs:line 35
   at StackTest.TestPushDouble() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\StackTest.cs:line 39

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\Stack.cs:line 49
   at StackTest.TestPopDouble() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\StackTest.cs:line 62
```

Fig. 20.8 | Testing generic class `Stack`. (Part 7 of 8.)

```
Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 11
Message: Stack is full, cannot push 11
   at Stack`1.Push(T pushValue) in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\Stack.cs:line 35
   at StackTest.TestPushInt() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\StackTest.cs:line 85

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\Stack.cs:line 49
   at StackTest.TestPopInt() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_05_08\Stack\Stack\StackTest.cs:line 109
```

Fig. 20.8 | Testing generic class Stack. (Part 8 of 8.)

```
// Fig. 20.9: StackTest.cs
// Testing generic class Stack.
using System;
using System.Collections.Generic;

class StackTest
{
   // create arrays of doubles and ints
   private static double[] doubleElements =
      {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
   private static int[] intElements =
      {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

   private static Stack<double> doubleStack; // stack stores doubles
   private static Stack<int> intStack; // stack stores int objects

```

Fig. 20.9 | Testing generic class Stack. (Part 1 of 6.)

```
   static void Main()
   {
      doubleStack = new Stack<double>(5); // stack of doubles
      intStack = new Stack<int>(10); // stack of ints

      // push doubles onto doubleStack
      TestPush(nameof(doubleStack), doubleStack, doubleElements);
      // pop doubles from doubleStack
      TestPop(nameof(doubleStack), doubleStack);
      // push ints onto intStack
      TestPush(nameof(doubleStack), intStack, intElements);
      // pop ints from intStack
      TestPop(nameof(doubleStack), intStack);
   }

```

Fig. 20.9 | Testing generic class `Stack`. (Part 2 of 6.)

```
// test Push method
private static void TestPush<T>(string name, Stack<T> stack,
   IEnumerable<T> elements)
{
   // push elements onto stack
   try
   {
      Console.WriteLine($"\nPushing elements onto {name}");

      // push elements onto stack
      foreach (var element in elements)
      {
         Console.Write($"{element} ");
         stack.Push(element); // push onto stack
      }
   }
   catch (FullStackException exception)
   {
      Console.Error.WriteLine($"\nMessage: {exception.Message}");
      Console.Error.WriteLine(exception.StackTrace);
   }
}

```

Fig. 20.9 | Testing generic class `Stack`. (Part 3 of 6.)

```
   // test Pop method
   private static void TestPop<T>(string name, Stack<T> stack)
   {
      // pop elements from stack
      try
      {
         Console.WriteLine($"\nPopping elements from {name}");

         T popValue; // store element removed from stack

         // remove all elements from stack
         while (true)
         {
            popValue = stack.Pop(); // pop from stack
            Console.Write($"{popValue} ");
         }
      }
      catch (EmptyStackException exception)
      {
         Console.Error.WriteLine($"\nMessage: {exception.Message}");
         Console.Error.WriteLine(exception.StackTrace);
      }
   }
}
```

Fig. 20.9 | Testing generic class `Stack`. (Part 4 of 6.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
Message: Stack is full, cannot push 6.6
   at Stack`1.Push(T pushValue) in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_09\Stack\Stack\Stack.cs:line 35
   at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1
      elements) in C:\Users\PaulDeitel\Documents\examples\ch20\Fig20_09\
      Stack\Stack\StackTest.cs:line 45

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_09\Stack\Stack\Stack.cs:line 49
   at StackTest.TestPop[T](String name, Stack`1 stack) in
      C:\Users\PaulDeitel\Documents\examples\ch20\Fig20_09\Stack\
      Stack\StackTest.cs:line 68
```

Fig. 20.9 | Testing generic class Stack. (Part 5 of 6.)

```
Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 11
Message: Stack is full, cannot push 11
   at Stack`1.Push(T pushValue) in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_09\Stack\Stack\Stack.cs:line 35
   at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1
      elements) in C:\Users\PaulDeitel\Documents\examples\ch20\Fig20_09\
      Stack\Stack\StackTest.cs:line 45

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in C:\Users\PaulDeitel\Documents\
      examples\ch20\Fig20_09\Stack\Stack\Stack.cs:line 49
   at StackTest.TestPop[T](String name, Stack`1 stack) in
      C:\Users\PaulDeitel\Documents\examples\ch20\Fig20_09\Stack\
      Stack\StackTest.cs:line 68
```

Fig. 20.9 | Testing generic class Stack. (Part 6 of 6.)