

## 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 `Comparable<T>` Interface

20.4.2 Specifying Type Constraints

**20.5** Overloading Generic Methods

**20.6** Generic Classes

**20.7** Wrap-Up

---

---

```
1  // Fig. 20.1: OverloadedMethods.cs
2  // Using overloaded methods to display arrays of different types.
3  using System;
4
5  class OverloadedMethods
6  {
7      static void Main(string[] args)
8      {
9          // create arrays of int, double and char
10         int[] intArray = {1, 2, 3, 4, 5, 6};
11         double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
12         char[] charArray = {'H', 'E', 'L', 'L', 'O'};
13
14         Console.WriteLine("Array intArray contains: ");
15         DisplayArray(intArray); // pass an int array argument
16         Console.WriteLine("Array doubleArray contains: ");
17         DisplayArray(doubleArray); // pass a double array argument
18         Console.WriteLine("Array charArray contains: ");
19         DisplayArray(charArray); // pass a char array argument
20     }
```

---

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

---

```
21
22 // output int array
23 private static void DisplayArray(int[] inputArray)
24 {
25     foreach (var element in inputArray)
26     {
27         Console.Write($"{element} ");
28     }
29
30     Console.WriteLine();
31 }
32
33 // output double array
34 private static void DisplayArray(double[] inputArray)
35 {
36     foreach (var element in inputArray)
37     {
38         Console.Write($"{element} ");
39     }
40
41     Console.WriteLine();
42 }
```

---

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

```
43
44 // output char array
45 private static void DisplayArray(char[] inputArray)
46 {
47     foreach (var element in inputArray)
48     {
49         Console.Write($"{element} ");
50     }
51
52     Console.WriteLine();
53 }
54 }
```

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.)

---

```
1 private static void DisplayArray(T[] inputArray)
2 {
3     foreach (var element in inputArray)
4     {
5         Console.Write($"{element} ");
6     }
7
8     Console.WriteLine();
9 }
```

---

**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.

---

```
1  // Fig. 20.3: GenericMethod.cs
2  // Using a generic method to display arrays of different types.
3  using System;
4
5  class GenericMethod
6  {
7      static void Main()
8      {
9          // create arrays of int, double and char
10         int[] intArray = {1, 2, 3, 4, 5, 6};
11         double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
12         char[] charArray = {'H', 'E', 'L', 'L', 'O'};
13
14         Console.Write("Array intArray contains: ");
15         DisplayArray(intArray); // pass an int array argument
16         Console.Write("Array doubleArray contains: ");
17         DisplayArray(doubleArray); // pass a double array argument
18         Console.Write("Array charArray contains: ");
19         DisplayArray(charArray); // pass a char array argument
20     }
```

---

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

```
21
22 // output array of all types
23 private static void DisplayArray<T>(T[] inputArray)
24 {
25     foreach (var element in inputArray)
26     {
27         Console.Write($"{element} ");
28     }
29     Console.WriteLine();
30 }
31 }
32 }
```

```
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.*

---

```
1  // Fig. 20.4: MaximumTest.cs
2  // Generic method Maximum returns the largest of three objects.
3  using System;
4
5  class MaximumTest
6  {
7      static void Main()
8      {
9          Console.WriteLine($"Maximum of 3, 4 and 5 is {Maximum(3, 4, 5)}\n");
10         Console.WriteLine(
11             $"Maximum of 6.6, 8.8 and 7.7 is {Maximum(6.6, 8.8, 7.7)}\n");
12         Console.WriteLine("Maximum of pear, apple and orange is " +
13             $"{Maximum("pear", "apple", "orange")}\n");
14     }
15
```

---

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

---

```
16 // generic function determines the
17 // largest of the IComparable<T> objects
18 private static T Maximum<T>(T x, T y, T z) where T : IComparable<T>
19 {
20     var max = x; // assume x is initially the largest
21
22     // compare y with max
23     if (y.CompareTo(max) > 0)
24     {
25         max = y; // y is the largest so far
26     }
27
28     // compare z with max
29     if (z.CompareTo(max) > 0)
30     {
31         max = z; // z is the largest
32     }
33
34     return max; // return largest object
35 }
36 }
```

---

**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.)

---

```
17 // constructor creates a stack of the specified number of elements
18 public Stack(int stackSize)
19 {
20     if (stackSize <= 0) // validate stackSize
21     {
22         throw new ArgumentException("Stack size must be positive.");
23     }
24
25     elements = new T[stackSize]; // create stackSize elements
26     top = -1; // stack initially empty
27 }
28
```

---

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

---

```
29    // push element onto the stack; if unsuccessful,
30    // throw FullStackException
31    public void Push(T pushValue)
32    {
33        if (top == elements.Length - 1) // stack is full
34        {
35            throw new FullStackException(
36                $"Stack is full, cannot push {pushValue}");
37        }
38
39        ++top; // increment top
40        elements[top] = pushValue; // place pushValue on stack
41    }
```

---

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



---

```
42
43 // return the top element if not empty,
44 // else throw EmptyStackException
45 public T Pop()
46 {
47     if (top == -1) // stack is empty
48     {
49         throw new EmptyStackException("Stack is empty, cannot pop");
50     }
51
52     --top; // decrement top
53     return elements[top + 1]; // return top value
54 }
55 }
```

---

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

---

```
1  // Fig. 20.6: FullStackException.cs
2  // FullStackException indicates a stack is full.
3  using System;
4
5  public class FullStackException : Exception
6  {
7      // parameterless constructor
8      public FullStackException() : base("Stack is full")
9      {
10         // empty constructor
11     }
12
13     // one-parameter constructor
14     public FullStackException(string exception) : base(exception)
15     {
16         // empty constructor
17     }
```

---

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

---

```
18
19 // two-parameter constructor
20 public FullStackException(string exception, Exception inner)
21     : base(exception, inner)
22 {
23     // empty constructor
24 }
25 }
```

---

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

---

```
1  // Fig. 20.7: EmptyStackException.cs
2  // EmptyStackException indicates a stack is empty.
3  using System;
4
5  public class EmptyStackException : Exception
6  {
7      // parameterless constructor
8      public EmptyStackException() : base("Stack is empty")
9      {
10         // empty constructor
11     }
12
13     // one-parameter constructor
14     public EmptyStackException(string exception) : base(exception)
15     {
16         // empty constructor
17     }
18
```

---

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

---

```
19    // two-parameter constructor
20    public EmptyStackException(string exception, Exception inner)
21        : base(exception, inner)
22    {
23        // empty constructor
24    }
25 }
```

---

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

---

```
1  // Fig. 20.8: StackTest.cs
2  // Testing generic class Stack.
3  using System;
4
5  class StackTest
6  {
7      // create arrays of doubles and ints
8      private static double[] doubleElements =
9          {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
10     private static int[] intElements =
11         {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
12
13     private static Stack<double> doubleStack; // stack stores doubles
14     private static Stack<int> intStack; // stack stores ints
15
```

---

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

---

```
16    static void Main()
17    {
18        doubleStack = new Stack<double>(5); // stack of doubles
19        intStack = new Stack<int>(10); // stack of ints
20
21        TestPushDouble(); // push doubles onto doubleStack
22        TestPopDouble(); // pop doubles from doubleStack
23        TestPushInt(); // push ints onto intStack
24        TestPopInt(); // pop ints from intStack
25    }
26
```

---

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

---

```
27 // test Push method with doubleStack
28 private static void TestPushDouble()
29 {
30     // push elements onto stack
31     try
32     {
33         Console.WriteLine("\nPushing elements onto doubleStack");
34
35         // push elements onto stack
36         foreach (var element in doubleElements)
37         {
38             Console.Write($"{element:F1} ");
39             doubleStack.Push(element); // push onto doubleStack
40         }
41     }
42     catch (FullStackException exception)
43     {
44         Console.Error.WriteLine($"{exception.Message}");
45         Console.Error.WriteLine(exception.StackTrace);
46     }
47 }
48
```

---

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



---

```
49 // test Pop method with doubleStack
50 private static void TestPopDouble()
51 {
52     // pop elements from stack
53     try
54     {
55         Console.WriteLine("\nPopping elements from doubleStack");
56
57         double popValue; // store element removed from stack
58
59         // remove all elements from stack
60         while (true)
61         {
62             popValue = doubleStack.Pop(); // pop from doubleStack
63             Console.Write($"{popValue:F1} ");
64         }
65     }
66     catch (EmptyStackException exception)
67     {
68         Console.Error.WriteLine($"{\nMessage: {exception.Message}");
69         Console.Error.WriteLine(exception.StackTrace);
70     }
71 }
72
```

---

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

---

```
73 // test Push method with intStack
74 private static void TestPushInt()
75 {
76     // push elements onto stack
77     try
78     {
79         Console.WriteLine("\nPushing elements onto intStack");
80
81         // push elements onto stack
82         foreach (var element in intElements)
83         {
84             Console.Write($"{element} ");
85             intStack.Push(element); // push onto intStack
86         }
87     }
88     catch (FullStackException exception)
89     {
90         Console.Error.WriteLine($"{exception.Message}");
91         Console.Error.WriteLine(exception.StackTrace);
92     }
93 }
94
```

---

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

---

```
95     // test Pop method with intStack
96     private static void TestPopInt()
97     {
98         // pop elements from stack
99         try
100        {
101            Console.WriteLine("\nPopping elements from intStack");
102
103            int popValue; // store element removed from stack
104
105            // remove all elements from stack
106            while (true)
107            {
108                popValue = intStack.Pop(); // pop from intStack
109                Console.Write($"{popValue:F1} ");
110            }
111        }
112        catch (EmptyStackException exception)
113        {
114            Console.Error.WriteLine($"{\nMessage: {exception.Message}");
115            Console.Error.WriteLine(exception.StackTrace);
116        }
117    }
118 }
```

---

**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.)

---

```
1  // Fig. 20.9: StackTest.cs
2  // Testing generic class Stack.
3  using System;
4  using System.Collections.Generic;
5
6  class StackTest
7  {
8      // create arrays of doubles and ints
9      private static double[] doubleElements =
10         {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
11     private static int[] intElements =
12         {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
13
14     private static Stack<double> doubleStack; // stack stores doubles
15     private static Stack<int> intStack; // stack stores int objects
16
```

---

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

---

```
17 static void Main()
18 {
19     doubleStack = new Stack<double>(5); // stack of doubles
20     intStack = new Stack<int>(10); // stack of ints
21
22     // push doubles onto doubleStack
23     TestPush(nameof(doubleStack), doubleStack, doubleElements);
24     // pop doubles from doubleStack
25     TestPop(nameof(doubleStack), doubleStack);
26     // push ints onto intStack
27     TestPush(nameof(doubleStack), intStack, intElements);
28     // pop ints from intStack
29     TestPop(nameof(doubleStack), intStack);
30 }
31
```

---

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

---

```
32 // test Push method
33 private static void TestPush<T>(string name, Stack<T> stack,
34     IEnumerable<T> elements)
35 {
36     // push elements onto stack
37     try
38     {
39         Console.WriteLine($"\\nPushing elements onto {name}");
40
41         // push elements onto stack
42         foreach (var element in elements)
43         {
44             Console.Write($"{element} ");
45             stack.Push(element); // push onto stack
46         }
47     }
48     catch (FullStackException exception)
49     {
50         Console.Error.WriteLine($"\\nMessage: {exception.Message}");
51         Console.Error.WriteLine(exception.StackTrace);
52     }
53 }
54
```

---

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



---

```
55 // test Pop method
56 private static void TestPop<T>(string name, Stack<T> stack)
57 {
58     // pop elements from stack
59     try
60     {
61         Console.WriteLine($"\\nPopping elements from {name}");
62
63         T popValue; // store element removed from stack
64
65         // remove all elements from stack
66         while (true)
67         {
68             popValue = stack.Pop(); // pop from stack
69             Console.Write($"{popValue} ");
70         }
71     }
72     catch (EmptyStackException exception)
73     {
74         Console.Error.WriteLine($"\\nMessage: {exception.Message}");
75         Console.Error.WriteLine(exception.StackTrace);
76     }
77 }
78 }
```

---

**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.)