

27

# Generics



# OBJECTIVES

In this chapter you will learn:

- To create generic methods that perform identical tasks on arguments of different types.
- To create a generic `Stack` class that can be used to store objects of any class or interface type.
- To understand how to overload generic methods with nongeneric methods or with other generic methods.
- To understand the `new()` constraint of a type parameter.
- To apply multiple constraints to a type parameter.



## 27.1 Introduction

- **Generic methods** enable you to specify, with a single method declaration, a set of related methods.
- **Generic classes** enable you to specify, with a single class declaration, a set of related classes.
- Similarly, **generic interfaces** enable you to specify, with a single interface declaration, a set of related interfaces.



## Outline

Overl oadedMethods  
.cs

(1 of 3)

```
1 // Fig. 27.1: Overl oadedMethods.cs
2 // Using overl oaded methods to di splay arrays of di fferent types.
3 using System;
4
5 class Overl oadedMethods
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.Wri teLi ne( "Array intArray contai ns:" );
15        Di splayArray( intArray ); // pass an int array argument
16        Console.Wri teLi ne( "Array doubl eArray contai ns:" );
17        Di splayArray( doubl eArray ); // pass a double array argument
18        Console.Wri teLi ne( "Array charArray contai ns:" );
19        Di splayArray( charArray ); // pass a char array argument
20    } // end Mai n
```

**Fig. 27.1** | Using overloaded methods to display arrays of different types. (Part 1 of 3.)



## Outline

OverloadedMethods  
.cs

(2 of 3)

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

**Fig. 27.1** | Using overloaded methods to display arrays of different types. (Part 2 of 3.)



## Outline

OverloadedMethods  
.cs

(3 of 3)

```
40 // output char array
41 private static void DisplayArray( char[] inputArray )
42 {
43     foreach ( var element in inputArray )
44         Console.WriteLine( element + " " );
45
46     Console.WriteLine( "\n" );
47 } // end method DisplayArray
48 } // 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. 27.1** | Using overloaded methods to display arrays of different types. (Part 3 of 3.)



## Outline

- If the operations performed by several overloaded methods are identical for each type, they can be coded using a generic method.

Generi cMethod. cs

```
1 // Fig. 27.3: Generi cMethod. cs
2 // Using overloaded methods to di splay arrays of di fferent types. (1 of 2)
3 using System;
4 using System. Col l ecti ons. Generi c;
5
6 class Generi cMethod
7 {
8     public static void Main( string[] args )
9     {
10         // create arrays of int, double and char
11         int[] intArray = { 1, 2, 3, 4, 5, 6 };
12         doubl e[] doubl eArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
13         char[] charArray = { 'H', 'E', 'L', 'L', 'O' };

```

**Fig. 27.3** | Using overloaded methods to display arrays of different types. (Part 1 of 2.)



## Outline

```

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 } // end Main
21
22 // output array of all types
23 private static void DisplayArray< T >( T[] inputArray )
24 {
25     foreach ( T element in inputArray )
26         Console.WriteLine( element + " " );
27
28     Console.WriteLine( "\n" );
29 } // end method DisplayArray
30 } // end class GenericMethod

```

GenericMethod.cs

(2 of 2)

The compiler sets up a call to DisplayArray with int as the **type argument** for T. This is called **type-inferencing**.

All generic method declarations have a **type-parameter list** delimited by angle brackets that follows the method's name.

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



## 27.3 Generic-Method Implementation

- All generic method declarations have a **type-parameter list** delimited by angle brackets that follows the method's name.
- Each type-parameter list contains one or more **type parameters**, separated by commas.
- A type parameter is an identifier that is used in place of actual type names.
- The type parameters can be used to declare the return type, the parameter types and the local variable types in a generic method declaration.
- A type parameter can be declared only once in the type-parameter list but can appear more than once in the method's parameter list.



## *Specifying Type Constraints*

- 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.

MaximumTest.cs

(1 of 3)

```
1 // Fig. 27.4: MaximumTest.cs
2 // Generic method Maximum returns the largest of three objects.
3 using System;
4
5 class MaximumTest
6 {
7     public static void Main( string[] args )
8     {
9         Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
10             3, 4, 5, Maximum( 3, 4, 5 ) );
11         Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
12             6.6, 8.8, 7.7, Maximum( 6.6, 8.8, 7.7 ) );
13         Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
```

**Fig. 27.4** | Generic method Maximum returns the largest of three objects. (Part 1 of 3.)



## Outline

### MaximumTest.cs

(2 of 3)

```

14     "pear", "apple", "orange",
15     Maximum( "pear", "apple", "orange" ) );
16 } // end Main
17
18 // generic function determines the
19 // largest of the IComparable objects
20 private static T Maximum< T >( T x, T y, T z )
21     where T : IComparable< T >
22 {
23     T max = x; // assume x is initially the largest
24
25     // compare y with max
26     if ( y.CompareTo( max ) > 0 )
27         max = y; // y is the largest so far
28
29     // compare z with max
30     if ( z.CompareTo( max ) > 0 )
31         max = z; // z is the largest
32

```

Use type parameter T as the return type of the method, the type of parameters x, y and z, and the type of local variable max.

The **where** clause specifies the type constraint.

**Fig. 27.4** | Generic method Maximum returns the largest of three objects. (Part 2 of 3.)



## Outline

### MaximumTest.cs

(3 of 3)

```
33     return max; // return largest object
34 } // end method Maximum
35 } // 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. 27.4** | Generic method Maximum returns the largest of three objects. (Part 3 of 3.)



## 27.4 Type Constraints (Cont.)

### *I Comparable< T > Interface*

- It is possible to compare two objects of the same type if that type implements the generic interface `I Comparable< T >`.
- The structures in the Framework Class Library that correspond to the simple types all implement this interface.
- Types that implement `I Comparable< T >` must declare a `CompareTo` method for comparing objects.
- Method `CompareTo` must return:
  - 0 if the objects are equal.
  - A negative integer if the caller is less than the argument.
  - A positive integer if the caller is greater than the argument.



## 27.4 Type Constraints (Cont.)

- The **where** clause specifies the type constraint.
- If no type constraint is specified, the default type constraint is **object**.
- 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.
- You can specify that a type argument must be a reference type or a value type by using the **reference-type (class)** or the **value-type (struct)** constraints.



## 27.6 Generic Classes

- A generic class provides a means for describing a class in a type-independent manner.
- We can then instantiate type-specific objects of the generic class.
- At compilation time, the compiler ensures your code's type safety, and the runtime system replaces type parameters with type arguments.



## Outline

- Figure 27.5 presents a generic Stack class declaration.

### Stack.cs

(1 of 3)

```
1 // Fig. 27.5: Stack.cs
2 // Generic class Stack.
3 using System;
4
5 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    } // end stack constructor
16
```

In a generic class declaration, the class name is followed by a type-parameter list and, optionally, a constraint on its type parameter.

Declare variable elements as an array of type T.

**Fig. 27.5** | Generic class Stack. (Part 1 of 3.)



Outline

## Stack.cs

(2 of 3)

```

17 // constructor creates a stack of the specified number of elements
18 public Stack( int stackSize )
19 {
20     if ( stackSize > 0 ) // validate stackSize
21         elements = new T[ stackSize ]; // create stackSize elements
22     else
23         throw new ArgumentException( "Stack size must be positive." );
24
25     top = -1; // stack initially empty
26 } // end stack constructor
27
28 // push element onto the stack; if unsuccessful,
29 // throw FullStackException
30 public void Push( T pushValue )
31 {
32     if ( top == elements.Length - 1 ) // stack is full
33         throw new FullStackException( string.Format(
34             "Stack is full, cannot push {0}", pushValue ) );

```

Create the array of type T.

**Fig. 27.5** | Generic class Stack. (Part 2 of 3.)



## Outline

### Stack. cs

(3 of 3)

```
35
36     ++top; // increment top
37     elements[ top ] = pushValue; // place pushValue on stack
38 } // end method Push
39
40 // return the top element if not empty,
41 // else throw EmptyStackException
42 public T Pop()
43 {
44     if ( top == -1 ) // stack is empty
45         throw new EmptyStackException( "Stack is empty, cannot pop" );
46
47     --top; // decrement top
48     return elements[ top + 1 ]; // return top value
49 } // end method Pop
50 } // end class Stack
```

**Fig. 27.5** | Generic class Stack. (Part 3 of 3.)



## 27.6 Generic Classes (Cont.)

- In a generic class declaration, the class name is followed by a type-parameter list and, optionally, a constraint on its type parameter.
- As with generic methods, the type-parameter list of a generic class can have one or more type parameters separated by commas.



Outline

- Classes FullStackException (Fig. 27.6) and EmptyStackException (Fig. 27.7) each provide a parameterless constructor, a one-argument constructor for exception classes and a two-argument constructor for creating a new exception using an existing one.

FullStackException  
.CS

(1 of 2)

```

1 // Fig. 27.6: FullStackException.cs
2 // FullStackException indicates a stack is full.
3 using System;
4
5 class FullStackException : Exception
6 {
7     // parameterless constructor
8     public FullStackException() : base( "Stack is full" )
9     {
10        // empty constructor
11    } // end FullStackException constructor
12
13    // one-parameter constructor
14    public FullStackException( string exception ) : base( exception )
15    {
16        // empty constructor
17    } // end FullStackException constructor

```

**Fig. 27.6** | FullStackException indicates a stack is full. (Part 1 of 2.)



## Outline

### FullStackException .cs

(2 of 2)

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

**Fig. 27.6** | FullStackException indicates a stack is full. (Part 2 of 2.)



## Outline

### EmptyStack Exception. cs

(1 of 2)

```
1 // Fig. 27.7: EmptyStackException.cs
2 // EmptyStackException indicates a stack is empty.
3 using System;
4
5 class EmptyStackException : Exception
6 {
7     // parameterless constructor
8     public EmptyStackException() : base( "Stack is empty" )
9     {
10         // empty constructor
11     } // end EmptyStackException constructor
12
13     // one-parameter constructor
14     public EmptyStackException( string exception ) : base( exception )
15     {
16         // empty constructor
17     } // end EmptyStackException constructor
```

**Fig. 27.7** | EmptyStackException indicates a stack is empty. (Part 1 of 2.)



## Outline

### EmptyStack Exception. cs

(2 of 2)

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

**Fig. 27.7** | EmptyStackException indicates a stack is empty. (Part 2 of 2.)



## 27.6 Generic Classes (Cont.)

- The compiler performs type checking on the class's type parameters to ensure that they can be used with the code in the generic class.
- The constraints determine the operations that can be performed on the type parameters.
- The runtime system replaces the type parameters with the actual types at runtime.
- The scope of a generic class's type parameter is the entire class.



## Outline

- Now, let's consider an application (Fig. 27.8) that uses the Stack generic class.

### StackTest.cs

(1 of 8)

```

1 // Fig. 27.8: 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         new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
10    private static int[] intElements =
11        new int[]{ 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 int objects
15
16    public static void Main( string[] args )
17    {
18        doubleStack = new Stack< double >( 5 ); // stack of doubles
19        intStack = new Stack< int >( 10 ); // stack of ints
20

```

declare variables of type Stack< double > and Stack< int >. The types double and int are the Stack's type arguments.

**Fig. 27.8** | Testing generic class Stack. (Part 1 of 8.)



## Outline

### StackTest.cs

(2 of 8)

```
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 } // end Main
26
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.WriteLine( "{0:F1} ", element );
39             doubleStack.Push( element ); // push onto doubleStack
40         } // end foreach
41     } // end try
```

**Fig. 27.8** | Testing generic class Stack. (Part 2 of 8.)



## Outline

### StackTest.cs

(3 of 8)

```
42     catch ( FullStackExcepti on excepti on )
43     {
44         Consol e. Error. Wri teLi ne();
45         Consol e. Error. Wri teLi ne( "Message: " + excepti on. Message );
46         Consol e. Error. Wri teLi ne( excepti on. StackTrace );
47     } // end catch
48 } // end method TestPushDoubl e
49
50 // test Pop method wi th doubl eStack
51 private static void TestPopDoubl e()
52 {
53     // pop el ements from stack
54     try
55     {
56         Consol e. Wri teLi ne( "\nPoppi ng el ements from doubl eStack" );
57
58         doubl e popVal ue; // store el ement removed from stack
59
```

**Fig. 27.8** | Testing generic class Stack. (Part 3 of 8.)



## Outline

### StackTest.cs

(4 of 8)

```
60     // remove all elements from stack
61     while ( true )
62     {
63         popValue = doubleStack.Pop(); // pop from doubleStack
64         Console.WriteLine( "{0:F1} ", popValue );
65     } // end while
66 } // end try
67 catch ( EmptyStackException exception )
68 {
69     Console.Error.WriteLine();
70     Console.Error.WriteLine( "Message: " + exception.Message );
71     Console.Error.WriteLine( exception.StackTrace );
72 } // end catch
73 } // end method TestPopDouble
74
75 // test Push method with intStack
76 private static void TestPushInt()
77 {
78     // push elements onto stack
79     try
80     {
81         Console.WriteLine( "\nPushing elements onto intStack" );
82
```

**Fig. 27.8** | Testing generic class Stack. (Part 4 of 8.)



## Outline

### StackTest.cs

(5 of 8)

```
83     // push elements onto stack
84     foreach ( var element in intElements )
85     {
86         Console.Wri te( "{0} ", el ement );
87         i ntStack. Push( el ement ); // push onto i ntStack
88     } // end foreach
89 } // end try
90 catch ( Ful lStackExcepti on excepti on )
91 {
92     Console. Error. Wri teLi ne();
93     Console. Error. Wri teLi ne( "Message: " + excepti on. Message );
94     Console. Error. Wri teLi ne( excepti on. StackTrace );
95 } // end catch
96 } // end method TestPushI nt
97
98 // test Pop method wi th i ntStack
99 private static void TestPopI nt()
100 {
101     // pop el ements from stack
102     try
103     {
104         Console. Wri teLi ne( "\nPoppi ng el ements from i ntStack" );
```

**Fig. 27.8** | Testing generic class Stack. (Part 5 of 8.)



## Outline

### StackTest.cs

(6 of 8)

```
105
106     int popValue; // store element removed from stack
107
108     // remove all elements from stack
109     while ( true )
110     {
111         popValue = intStack.Pop(); // pop from intStack
112         Console.WriteLine( "{0} ", popValue );
113     } // end while
114 } // end try
115 catch ( EmptyStackException exception )
116 {
117     Console.Error.WriteLine();
118     Console.Error.WriteLine( "Message: " + exception.Message );
119     Console.Error.WriteLine( exception.StackTrace );
120 } // end catch
121 } // end method TestPopInt
122} // end class StackTest
```

**Fig. 27.8** | Testing generic class Stack. (Part 6 of 8.)



## Outline

### StackTest.cs

(7 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:\Examples\ch27\Fig27\_05\_08\Stack\Stack\Stack.cs:line 33

at StackTest.TestPushDouble() in

C:\Examples\ch27\Fig27\_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:\Examples\ch27\Fig27\_05\_08\Stack\Stack\Stack.cs:line 45

at StackTest.TestPopDouble() in

C:\Examples\ch27\Fig27\_05\_08\Stack\Stack\StackTest.cs:line 63

*(continued on next page...)*

**Fig. 27.8** | Testing generic class Stack. (Part 7 of 8.)



## Outline

### StackTest.cs

*(continued from previous page...)*

(8 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:\Examples\ch27\Fig27\_05\_08\Stack\Stack\Stack.cs:line 33

at StackTest.TestPushInt() in

C:\Examples\ch27\Fig27\_05\_08\Stack\Stack\StackTest.cs:line 87

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:\Examples\ch27\Fig27\_05\_08\Stack\Stack\Stack.cs:line 45

at StackTest.TestPopInt() in

C:\Examples\ch27\Fig27\_05\_08\Stack\Stack\StackTest.cs:line 111

**Fig. 27.8** | Testing generic class Stack. (Part 8 of 8.)



Outline

StackTest.cs

(1 of 6)

*Creating Generic Methods to Test Class Stack< T >*

- The code in methods TestPushDouble and TestPushInt is almost identical for pushing values onto a Stack< double > or a Stack< int >, respectively.
- Similarly methods TestPopDouble and TestPopInt are almost identical for popping values from a Stack< double > or a Stack< int >, respectively.

```

1 // Fig. 27.9: 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         new double[] { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
11     private static int[] intElements =
12         new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

```

**Fig. 27.9** | Testing generic class Stack. (Part 1 of 6.)

## Outline

### StackTest.cs

(2 of 6)

```
13
14 private static Stack< double > doubleStack; // stack stores doubles
15 private static Stack< int > intStack; // stack stores int objects
16
17 public static void Main( string[] args )
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( "doubleStack", doubleStack, doubleElements );
24     // pop doubles from doubleStack
25     TestPop( "doubleStack", doubleStack );
26     // push ints onto intStack
27     TestPush( "intStack", intStack, intElements );
28     // pop ints from intStack
29     TestPop( "intStack", intStack );
30 } // end Main
31
```

**Fig. 27.9** | Testing generic class Stack. (Part 2 of 6.)



## Outline

```

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.WriteLine( "{0} ", element );
45             stack.Push( element ); // push onto stack
46         } // end foreach
47     } // end try
48     catch ( FullStackException exception )
49     {
50         Console.Error.WriteLine();
51         Console.Error.WriteLine( "Message: " + exception.Message );
52         Console.Error.WriteLine( exception.StackTrace );
53     } // end catch
54 } // end method TestPush

```

StackTest.cs

(3 of 6)

Generic method TestPush uses type parameter T to represent the data type stored in the Stack.

**Fig. 27.9** | Testing generic class Stack. (Part 3 of 6.)



## Outline

### StackTest.cs

(4 of 6)

```
55
56 // test Pop method
57 private static void TestPop< T >( string name, Stack< T > stack )
58 {
59     // push elements onto stack
60     try
61     {
62         Console.WriteLine( "\nPoppi ng elements from " + name );
63
64         T popVal ue; // store element removed from stack
65
66         // remove all elements from stack
67         while ( true )
68         {
69             popVal ue = stack.Pop(); // pop from stack
70             Console.Wri te( "{0} ", popVal ue );
71         } // end while
72     } // end try
```

**Fig. 27.9** | Testing generic class Stack. (Part 4 of 6.)



## Outline

### StackTest.cs

(5 of 6)

```

73     catch ( EmptyStackExcepti on excepti on )
74     {
75         Consol e. Error. Wri teLi ne();
76         Consol e. Error. Wri teLi ne( "Message: " + excepti on. Message );
77         Consol e. Error. Wri teLi ne( excepti on. StackTrace );
78     } // end catch
79 } // end TestPop
80 } // end class StackTest

```

Pushi ng el ements onto doubl eStack

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:\Exampl es\ch27\Fi g27\_09\Stack\Stack\Stack.cs:Li ne 33

at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerabl e`1

el ements) in C:\Exampl es\ch27\Fi g27\_09\Stack\Stack\StackTest.cs:Li ne 45

Poppi ng el ements from doubl eStack

5.5 4.4 3.3 2.2 1.1

*(continued on next page...)*

**Fig. 27.9** | Testing generic class Stack. (Part 5 of 6.)



Outline

## StackTest.cs

(6 of 6)

*continued from previous page...*

```

Message: Stack is empty, cannot pop
  at Stack`1.Pop() in
    C:\Examples\ch27\Fig27_09\Stack\Stack\Stack.cs:line 45
  at StackTest.TestPop[T](String name, Stack`1 stack) in
    C:\Examples\ch27\Fig27_09\Stack\Stack\StackTest.cs:line 69

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:\Examples\ch27\Fig27_09\Stack\Stack\Stack.cs:line 33
  at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1
    elements) in C:\Examples\ch27\Fig27_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:\Examples\ch27\Fig27_09\Stack\Stack\Stack.cs:line 45
  at StackTest.TestPop[T](String name, Stack`1 stack) in
    C:\Examples\ch27\Fig27_09\Stack\Stack\StackTest.cs:line 69

```

**Fig. 27.9** | Testing generic class Stack. (Part 6 of 6.)