26

Generics

...our special individuality, as distinguished from our generic humanity.

— Oliver Wendell Holmes, Sr.

Every man of genius sees the world at a different angle from his fellows.

— Havelock Ellis

Born under one law, to another bound.

— Lord Brooke

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 non-generic methods or with other generic methods.
- To understand the `new()` constraint of a type parameter.
- To apply multiple constraints to a type parameter.
- The relationship between generics and inheritance.

Outline

26.1 Introduction

- **Generics**
  - **New feature of C# 2.0**
  - **Provide compile-time type safety**
    - **Catch invalid types at compile time**
  - **Generic methods**
    - **A single method declaration**
    - **A set of related methods**
  - **Generic classes**
    - **A single class declaration**
    - **A set of related classes**
  - **Generic interfaces**
    - **A single interface declaration**
    - **A set of related interfaces**

26.2 Motivation for Generic Methods

- **Overloaded methods**
  - **Perform similar operations on different types of data**
  - **Overloaded `PrintArray` methods**
    - `int` **array**
    - `double` **array**
    - `char` **array**

Outline

```
1  // Fig. 26.1: OverloadedMethods.cs
2  // Using overloaded methods to print 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       PrintArray( intArray ); // pass an int array argument
16       Console.WriteLine( "Array doubleArray contains:" );
17       PrintArray( doubleArray ); // pass a double array argument
18       Console.WriteLine( "Array charArray contains:" );
19       PrintArray( charArray ); // pass a char array argument
20    } // end Main
21
22    // output int array
23    static void PrintArray( int[] inputArray )
24    {
25       foreach ( int element in inputArray )
26          Console.Write( element + " " );
27
28       Console.WriteLine( "\n" );
29    } // end method PrintArray
```

Declare three arrays of different types

Method calls that invoke three separate methods that has the same functionality

Accept an `int` array and output its elements

OverloadedMethods.cs

(1 of 2)

Outline

Overloaded Methods .cs

(2 of 2)

```
30
31    // output double array
32    static void PrintArray( double[] inputArray )
33    {
34       foreach ( double element in inputArray )
35          Console.Write( element + " " );
36
37       Console.WriteLine( "\n" );
38    } // end method PrintArray
39
40    // output char array
41    static void PrintArray( char[] inputArray )
42    {
43       foreach ( char element in inputArray )
44          Console.Write( element + " " );
45
46       Console.WriteLine( "\n" );
47    } // end method PrintArray
48 } // end class OverloadedMethods
```

Accept a `double` array and output its elements

Accept a `char` array and output its elements

```
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
```

26.2 Motivation for Generic Methods (Cont.)

- **Study each `PrintArray` method**
  - **Array element type appears in two location**
    - **Method header**
    - **`foreach` statement header**
- **Combine three `PrintArray` methods into one**
  - **Replace the element types with a generic name `E`**
  - **Declare one `PrintArray` method**
    - **Display the string representation of the elements of any array**

```
1 static void PrintArray( E[] inputArray )
2 {
3    foreach ( E element in inputArray )
4       Console.Write( element + " " );
5
6    Console.WriteLine( "\n" );
7 } // end method PrintArray
```

Outline

PrintArray Method

26.3 Generic Method Implementation

- **Reimplement Fig. 26.1 using a generic method**
  - **Method calls are identical**
  - **Outputs are identical**
- **Generic method declaration**
  - **Type parameter list**
    - **Delimited by angle brackets (< and >)**
    - **Precede the method's return type**
    - **Contain one or more type parameters**
      - **Separated by commas**

Outline

GenericMethod.cs

(1 of 2)

```
1  // Fig. 26.3: GenericMethod.cs
2  // Using overloaded methods to print arrays of different types.
3  using System;
4  using System.Collections.Generic;
5
6  class GenericMethod
7  {
8     static void Main( string[] args )
9     {
10       // create arrays of int, double and char
11       int[] intArray = { 1, 2, 3, 4, 5, 6 };
12       double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
13       char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
14
15       Console.WriteLine( "Array intArray contains:" );
16       PrintArray( intArray ); // pass an int array argument
17       Console.WriteLine( "Array doubleArray contains:" );
18       PrintArray( doubleArray ); // pass a double array argument
19       Console.WriteLine( "Array charArray contains:" );
20       PrintArray( charArray ); // pass a char array argument
21    } // end Main
```

Call generic method to print out elements of arrays of different type

```
22
23    // output array of all types
24    static void PrintArray< E >( E[] inputArray )
25    {
26       foreach ( E element in inputArray )
27          Console.Write( element + " " );
28
29       Console.WriteLine( "\n" );
30    } // end method PrintArray
31 } // 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
```

Generic method header

Use the type parameter as an identifier in place of actual type names

GenericMethod.cs

Type parameter list

Output the elements in an array of any type

26.3 Generica Method Implementation (Cont.)

- **Type parameter**
  - **An identifier that specifies a generic type name**
  - **Used to declare return type, parameter types and local variable types**
  - **Act as placeholders for the types of the argument passed to the generic method**
- **Type Inferencing**
  - **When complier determines that an exact match occurs if the type parameter E of a method is replaced with a type of the elements in method call's argument, then the complier sets up a call to the method with that type as the type argument for the parameter E**
  - **Use explicit type arguments to indicate the exact type that should be used to call a generic function**

```
PrintArray< int >( intArray );
```

Common Programming Error 26.1

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 are encountered in the method. This results in compilation errors.

Good Programming Practice 26.1

It is 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 26.2

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

26.4 Type Constraints

- **Type Constraints**
  - **Generic `Maximum` method**
    - **Determines and returns the largest of its three arguments (of the same type)**
    - **The type parameter declares both the method's return type and its parameters**
  - **Expressions like variable1 < variable2 is not allowed unless the compiler can ensure that the operator < is provided for every type that will ever be used in the generic code**
  - **Cannot call a method on a generic-type variable unless the compiler can ensure that all types that will ever be used in the generic code support that method**

26.4 Type Constraints (Cont.)

- `IComparable< E >` **Interface**
  - **Allow objects of the same type to be compared**
    - **Must implement the generic interface** `IComparable< T >`
  - `IComparable < T >` **objects can be used with the sorting and searching methods of classes in the** `System.Collections.Generic` **namespace**
  - **The structures that correspond to the simple types all implement this interface**
  - **Must declare a** `CompareTo` **method for comparing objects**
    - **Returns 0 if the objects are equal**
    - **Returns a negative integer if int1 is less than int2**
    - **Returns a positive integer if int1 is greater than int2**

26.4 Type Constraints (Cont.)

- **Specifying Type Constraints**
  - `IComparable` **objects cannot be used with generic code by default**
    - **Not all types implement interface** `IComparable<T>`
    - **Restrict the types that can be used with a generic method or class to ensure that they meet certain requirements**
      - **Known as a type constraint**
        - **Restricts the type of the argument supplied to a particular type parameter**
      - **The** `where` **clause specifies the type constraint for type parameter T**

26.4 Type Constraints (Cont.)

- C# provides several kinds of type constraints
  - Class constraint
    - Indicates that the type argument must be an object of a specific base class or one of its derived classes
      - Can specify that the type argument must be a reference type or a value type
        - Use the reference type constraint (`class`)
        - Use the value type constraint (`struct`)
  - Interface constraint
    - Indicates that the type argument's class must implement a specific interface
  - Constructor constraint `new()`
    - Indicate that the generic code can use operator `new` to create new objects of the type represented by the type parameter
      - Must provide `public` parameter-less or default constructor
        - Ensure that objects of the class can be created without passing constructor arguments
  - Multiple constraints to a type parameter
    - Provide a comma-separated list of constraints in the `where` clause
      - Class, reference type or value type constraint must be listed first
        - Only one of these types of constraints can be used for each type parameter
      - Interface constraints are listed next
      - The constructor constraint is listed last (if there is one)

Outline

MaximumTest.cs

(1 of 2)

```
1  // Fig 26.4: MaximumTest.cs
2  // Generic method maximum returns the largest of three objects.
3  using System;
4
5  class MaximumTest
6  {
7     static void Main( 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",
14          "pear", "apple", "orange",
15          Maximum( "pear", "apple", "orange" ) );
16    } // end Main
17
18    // generic function determines the
19    // largest of the IComparable objects
```

Call generic interface type constraint method `Maximum`

Outline

MaximumTest.cs

(2 of 2)

```
20    static T Maximum< T >( T x, T y, T z ) where T : IComparable < T >
21    {
22       T max = x; // assume x is initially the largest
23
24       // compare y with max
25       if ( y.CompareTo( max ) > 0 )
26          max = y; // y is the largest so far
27
28       // compare z with max
29       if ( z.CompareTo( max ) > 0 )
30          max = z; // z is the largest
31
32       return max; // return largest object
33    } // end method Maximum
34 } // 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
```

Indicates that this method requires the type arguments to implement interface `IComparable<T>`

Uses type parameter T as the return type of the method and as the type of the method parameters

26.5 Overloading Generic Methods

- **Generic method may be overloaded**
  - **By another generic method with:**
    - **Same method name**
    - **Different method parameters**
  - **By non-generic methods with:**
    - **Same method name**
    - **Any number of parameters**

- **Compiler encountering a method call**
  - **Searches for the method declaration that most precisely matches the method name and argument types**
  - **Generates an error:**
    - **Ambiguity due to multiple possible matches**
    - **No matches**

26.6 Generic Classes

- **Generic classes**
  - **Simple, concise notation to indicate the actual type(s)**
  - **At compilation time, C# compiler:**
    - **Ensures the type safety**
    - **Replaces type parameters with actual arguments**
      - **Enable client code to interact with the generic class**
  - **Generic class declaration:**
    - **Class name**
    - **Followed by a type parameter list**
      - **Optional: constraint on its type parameter**
  - **Type parameter E**
    - **Represents the element type that is manipulated**
    - **Used throughout the class declaration to represent the element type**

Outline

Stack.cs

```
1  // Fig. 26.5: Stack.cs
2  // Generic class Stack
3  using System;
4
5  class Stack< E >
6  {
7     private int top; // location of the top element
8     private E[] elements; // array that stores Stack elements
9
10    // parameterless constructor creates a Stack of the default size
11    public Stack() : this( 10 ) // default stack size
12    {
13       // empty constructor; calls constructor at line 17 to perform init
14    } // end Stack constructor
15
16    // constructor creates a Stack of the specified number of elements
17    public Stack( int stackSize )
18    {
19       if ( stackSize > 0 ) // validate stackSize
20          elements = new E[ stackSize ]; // create stackSize elements
21       else
22          elements = new E[ 10 ]; // create 10 elements
23
24       top = -1; // Stack initially empty
25    } // end Stack constructor
```

Type parameter `E` is used throughout the `Stack` class to represent the element type

Declare `elements` as an array of type `E`

Initialize `elements` to the appropriate size

Outline

Stack.cs

```
26
27    // push element onto the Stack; if successful, return true
28    // otherwise, throw FullStackException
29    public void Push( E pushValue )
30    {
31       if ( top == elements.Length - 1 ) // Stack is full
32          throw new FullStackException( String.Format(
33             "Stack is full, cannot push {0}", pushValue ) );
34
35       top++; // increment top
36       elements[ top ] = pushValue; // place pushValue on Stack
37    } // end method Push
38
39    // return the top element if not empty
40    // else throw EmptyStackException
41    public E Pop()
42    {
43       if ( top == -1 ) // Stack is empty
44          throw new EmptyStackException( "Stack is empty, cannot pop" );
45
46       top--; // decrement top
47       return elements[ top + 1 ]; // return top value
48    } // end method Pop
49 } // end class Stack
```

Throw exception if stack is full

Increment `top` counter to indicate the new top position and store the argument

Throw exception if stack is empty

Decrement `top` counter to indicate the new top position and return the original top element

```
1  // Fig. 26.6: FullStackException.cs
2  // Indicates a stack is full.
3  using System;
4
5  class FullStackException : ApplicationException
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
18 } // end class FullStackException
```

Create customize exception for when the stack is full

Outline

FullStack Exception.cs

```
1  // Fig. 26.7: EmptyStackException.cs
2  // Indicates a stack is empty
3  using System;
4
5  class EmptyStackException : ApplicationException
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
18 } // end class EmptyStackException
```

Create customize exception for when the stack is empty

Outline

EmptyStackException.cs

Outline

StackTest.cs

(1 of 7)

```
1  // Fig. 26.8: StackTest.cs
2  // Stack generic class test program.
3  using System;
4
5  class StackTest
6  {
7     // create arrays of doubles and ints
8     static double[] doubleElements =
9        new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
10    static int[] intElements =
11       new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
12
13    static Stack< double > doubleStack; // stack stores double objects
14    static Stack< int > intStack; // stack stores int objects
15
16    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
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
```

Create and instantiates a stack of `int` and a stack of `double`

Manipulate the two stacks

Outline

StackTest.cs

(2 of 7)

```
26
27    // test Push method with doubleStack
28    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 ( double element in doubleElements )
37          {
38             Console.Write( "{0:F1} ", element );
39             doubleStack.Push( element ); // push onto doubleStack
40          } // end foreach
41       } // end try
42       catch ( FullStackException exception )
43       {
44          Console.Error.WriteLine();
45          Console.Error.WriteLine( "Message: " + exception.Message );
46          Console.Error.WriteLine( exception.StackTrace );
47       } // end catch
48    } // end method TestPushDouble
```

Push the elements of `doubleElements` into `doubleStack`

Exception handling for when the stack is full

Outline

StackTest.cs

(3 of 7)

```
49
50    // test Pop method with doubleStack
51    static void TestPopDouble()
52    {
53       // pop elements from stack
54       try
55       {
56          Console.WriteLine( "\nPopping elements from doubleStack" );
57
58          double popValue; // store element removed from stack
59
60          // remove all elements from Stack
61          while ( true )
62          {
63             popValue = doubleStack.Pop(); // pop from doubleStack
64             Console.Write( "{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
```

Pop the elements of `doubleElements` off `doubleStack`

Exception handling for when the stack is empty

Outline

StackTest.cs

(4 of 7)

```
74
75    // test Push method with intStack
76    static void TestPushInt()
77    {
78       // push elements onto stack
79       try
80       {
81          Console.WriteLine( "\nPushing elements onto intStack" );
82
83          // push elements onto stack
84          foreach ( int element in intElements )
85          {
86             Console.Write( "{0} ", element );
87             intStack.Push( element ); // push onto intStack
88          } // end foreach
89       } // end try
90       catch ( FullStackException exception )
91       {
92          Console.Error.WriteLine();
93          Console.Error.WriteLine( "Message: " + exception.Message );
94          Console.Error.WriteLine( exception.StackTrace );
95       } // end catch
96    } // end method TestPushInt
```

Push the elements of `intElements` into the `intStack`

Exception handling for when the stack is full

```
97
98    // test Pop method with intStack
99    static void TestPopInt()
100   {
101      // pop elements from stack
102      try
103      {
104         Console.WriteLine( "\nPopping elements from intStack" );
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.Write( "{0} ", popValue );
113         } // end while
114      } // end try
```

Pop the elements of `intElements` off `intStack`

Outline

StackTest.cs

(5 of 7)

```
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
```

Exception handling for when the stack is empty

StackTest.cs

(6 of 7)

```
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(E pushValue) in
      C:\Examples\ch25\Fig25_05\Stack\Stack.cs:line 32
   at StackTest.TestPushDouble() in
      C:\Examples\ch25\Fig25_05\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\ch25\Fig25_05\Stack\Stack.cs:line 44
   at StackTest.TestPopDouble() in
      C:\Examples\ch25\Fig25_05\Stack\StackTest.cs:line 63
```

```
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(E pushValue) in
      C:\Examples\ch25\Fig25_05\Stack\Stack.cs:line 32
   at StackTest.TestPushInt() in
       C:\Examples\ch25\Fig25_05\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\ch25\Fig25_05\Stack\Stack.cs:line 44
  at StackTest.TestPopInt() in
     C:\Examples\ch25\Fig25_05\Stack\StackTest.cs:line 111
```

Outline

StackTest.cs

(7 of 7)

Outline

StackTest.cs

(1 of 5)

```
1  // Fig 26.9: StackTest.cs
2  // Stack generic class test program.
3  using System;
4  using System.Collections.Generic;
5
6  class StackTest
7  {
8     // create arrays of doubles and ints
9     static double[] doubleElements =
10       new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
11    static int[] intElements =
12       new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
13
14    static Stack< double >doubleStack; // stack stores double objects
15    static Stack< int >intStack; // stack stores int objects
16
17    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
```

Create and instantiates a stack of `int` and a stack of `double`

Call generic methods to test `Stack`

Outline

StackTest.cs

(2 of 5)

```
31
32    static void TestPush< E >( string name, Stack< E > stack,
33       IEnumerable< E > elements )
34    {
35       // push elements onto stack
36       try
37       {
38          Console.WriteLine( "\nPushing elements onto " + name );
39
40          // push elements onto stack
41          foreach ( E element in elements )
42          {
43             Console.Write( "{0} ", element );
44             stack.Push( element ); // push onto stack
45          } // end foreach
46       } // end try
47       catch ( FullStackException exception )
48       {
49          Console.Error.WriteLine();
50          Console.Error.WriteLine( "Message: " + exception.Message );
51          Console.Error.WriteLine( exception.StackTrace );
52       } // end catch
53    } // end method TestPush
```

Generic method to push element onto stack

Push the elements of an array into its corresponding stack

Exception handling for when the stack is full

```
54
55    static void TestPop< E >( string name, Stack< E > stack )
56    {
57       // push elements onto stack
58       try
59       {
60          Console.WriteLine( "\nPopping elements from " + name );
61
62          E popValue; // store element removed from stack
63
64          // remove all elements from Stack
65          while ( true )
66          {
67             popValue = stack.Pop(); // pop from stack
68             Console.Write( "{0} ", popValue );
69          } // end while
70       } // end try
```

Generic method to pop element off stack

Pop the elements of an array off its corresponding stack

Outline

StackTest.cs

(3 of 5)

```
71       catch ( EmptyStackException exception )
72       {
73          Console.Error.WriteLine();
74          Console.Error.WriteLine( "Message: " + exception.Message );
75          Console.Error.WriteLine( exception.StackTrace );
76       } // end catch
77    } // end TestPop
78 } // end class StackTest
```

Exception handling for when the stack is empty

StackTest.cs

(4 of 5)

```
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(E pushValue) in
      C:\Examples\ch25\Fig25_05\Stack\Stack.cs:line 35
   at StackTest.TestPush[E](String name, Stack`1 stack, IEnumerable`1
      elements) in C:\Examples\ch25\Fig25_05\Stack\StackTest.cs:line 44

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\ch25\Fig25_05\Stack\Stack.cs:line 49
   at StackTest.TestPop[E](String name, Stack`1 stack) in
       C:\Examples\ch25\Fig25_05\Stack\StackTest.cs:line 67
```

```
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(E pushValue) in
      C:\Examples\ch25\Fig25_05\Stack\Stack.cs:line 35
   at StackTest.TestPush[E](String name, Stack`1 stack, IEnumerable`1
      elements) in C:\Examples\ch25\Fig25_05\Stack\StackTest.cs:line 44

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\ch25\Fig25_05\Stack\Stack.cs:line 49
   at StackTest.TestPop[E](String name, Stack`1 stack) in
      C:\Examples\ch25\Fig25_05\Stack\StackTest.cs:line 67
```

Outline

StackTest.cs

(5 of 5)

26.7 Notes on Generics and Inheritance

- **Generics and inheritance**
  - **Generic classes can be derived from a non-generic classes**
  - **Generic classes can be derived from another generic classes**
  - **Non-generic classes can be derived from a generic classes**