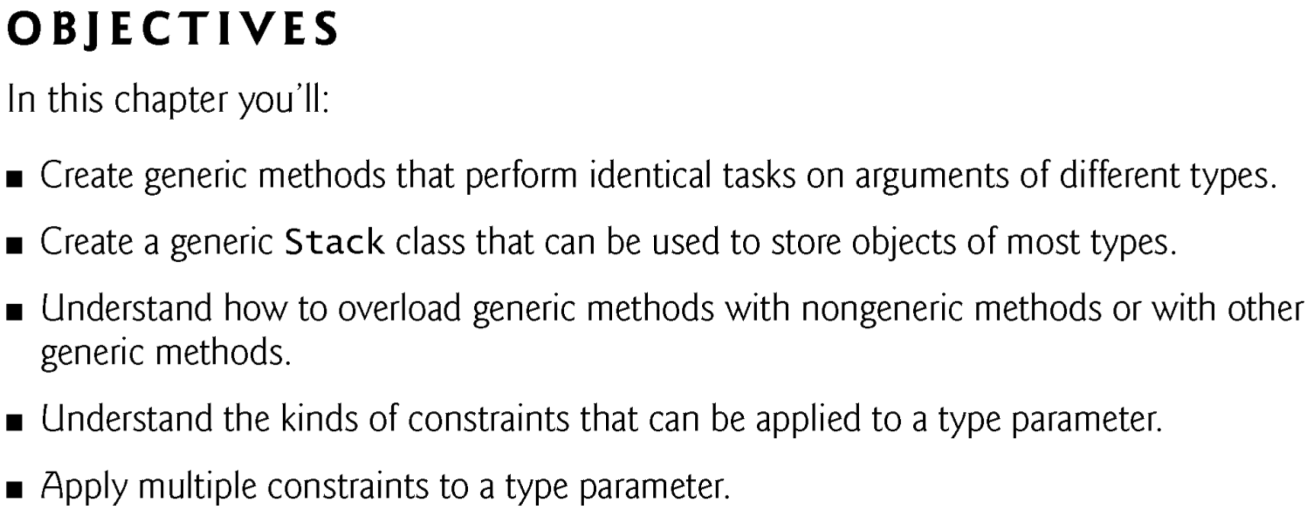
Chapter 20
Generics

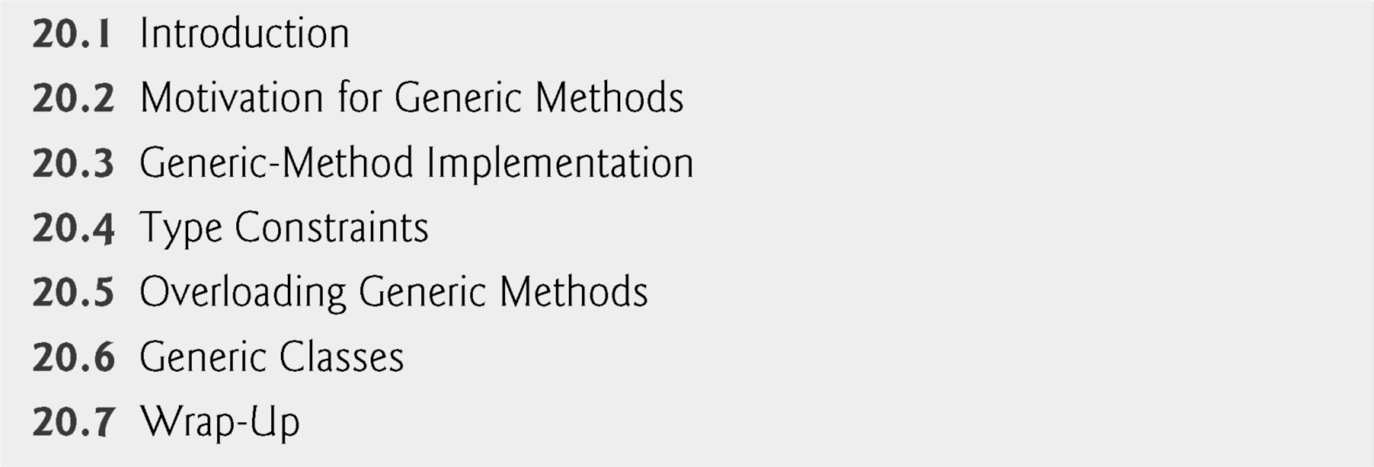
Visual C# 2012 How to Program



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 most types.
- 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
20.5 Overloading Generic Methods
20.6 Generic Classes
20.7 Wrap-Up

20.5 Overloading Generic Methods

- A generic method may be overloaded.
- Each overloaded method must have a unique signature (as discussed in Chapter 7).
- A class can provide two or more generic methods with the same name but *different* method parameters.
- A generic method can be overloaded by nongeneric methods with the same method name.
- When the compiler encounters a method call, it searches for the method declaration that best matches the method name and the argument types specified in the call.

20.6 Generic Classes

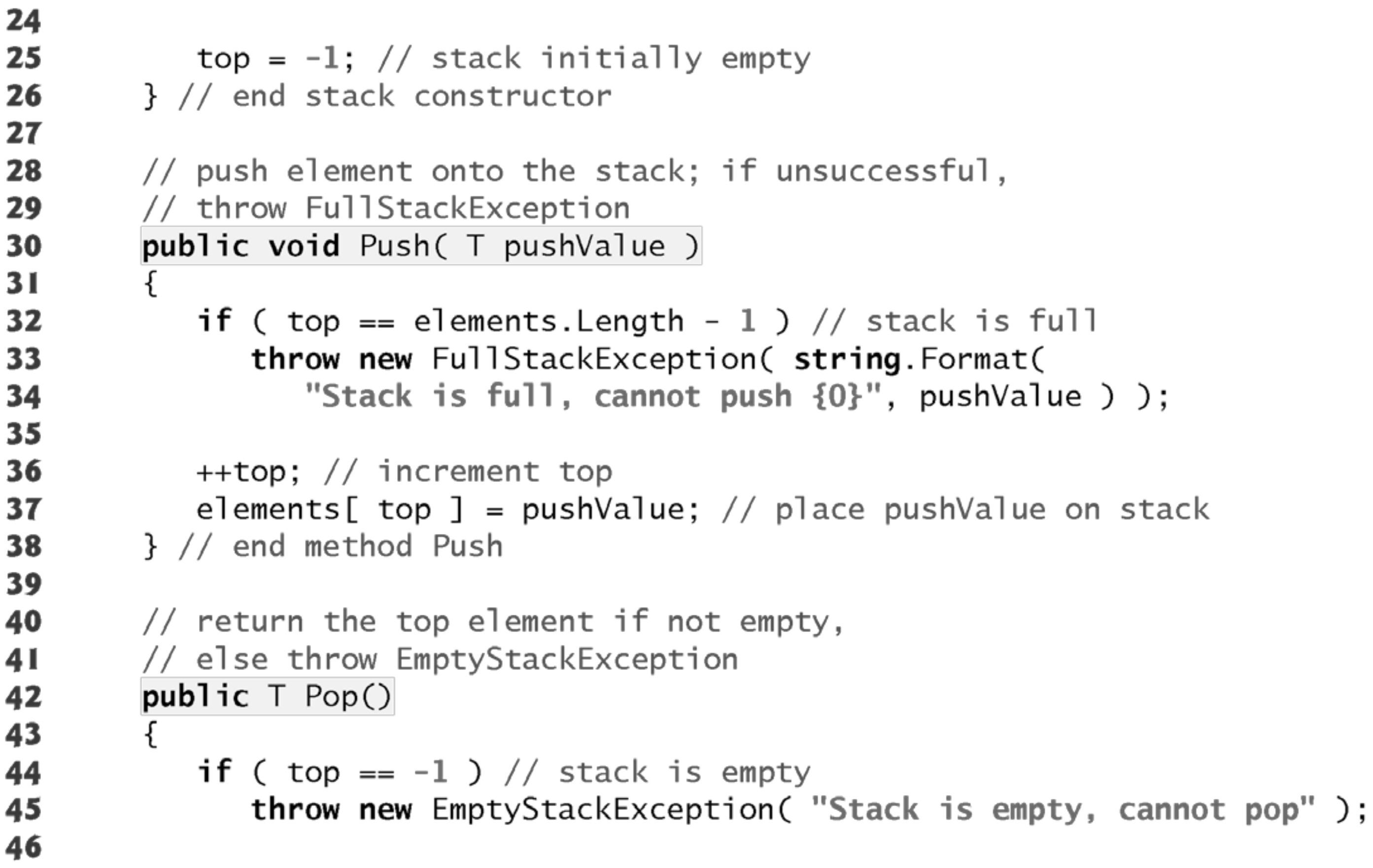
- With a generic class, you can use a simple, concise notation to indicate the actual type(s) that should be used in place of the class’s type parameter(s).
- At compilation time, the compiler ensures your code’s type safety, and the runtime system replaces type parameters with type arguments to enable your client code to interact with the generic class.

20.6 Generic Classes (cont.)

- One generic `Stack` class, for example, could be the basis for creating many `Stack` classes (e.g., “`Stack` of `double`,” “`Stack` of `int`,” “`Stack` of `char`,” “`Stack` of `Employee`”).
- Figure 20.5 presents a generic `Stack` class declaration.
- This class should not be confused with the class `Stack` from namespace `System.Collections.Generics`.

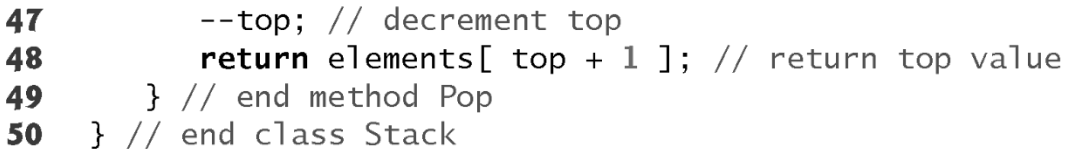
```
 1  // Fig. 20.5: Stack.cs
 2  // Generic class Stack.
 3  using System;
 4
 5  class Stack< T >
 6  {
 7     private int top; // location of the top element
 8     private T[] elements; // array that stores stack elements
 9
10     // parameterless constructor creates a stack of the default size
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
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." );
```

Fig. 20.5 | Generic class Stack. (Part 1 of 3.)



```
      top = -1; // stack initially empty
   } // end stack constructor

   // 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( string.Format(
            "Stack is full, cannot push {0}", pushValue ) );

      ++top; // increment top
      elements[ top ] = pushValue; // place pushValue on stack
   } // end method Push

   // 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" );

```

Fig. 20.5 | Generic class Stack. (Part 2 of 3.)

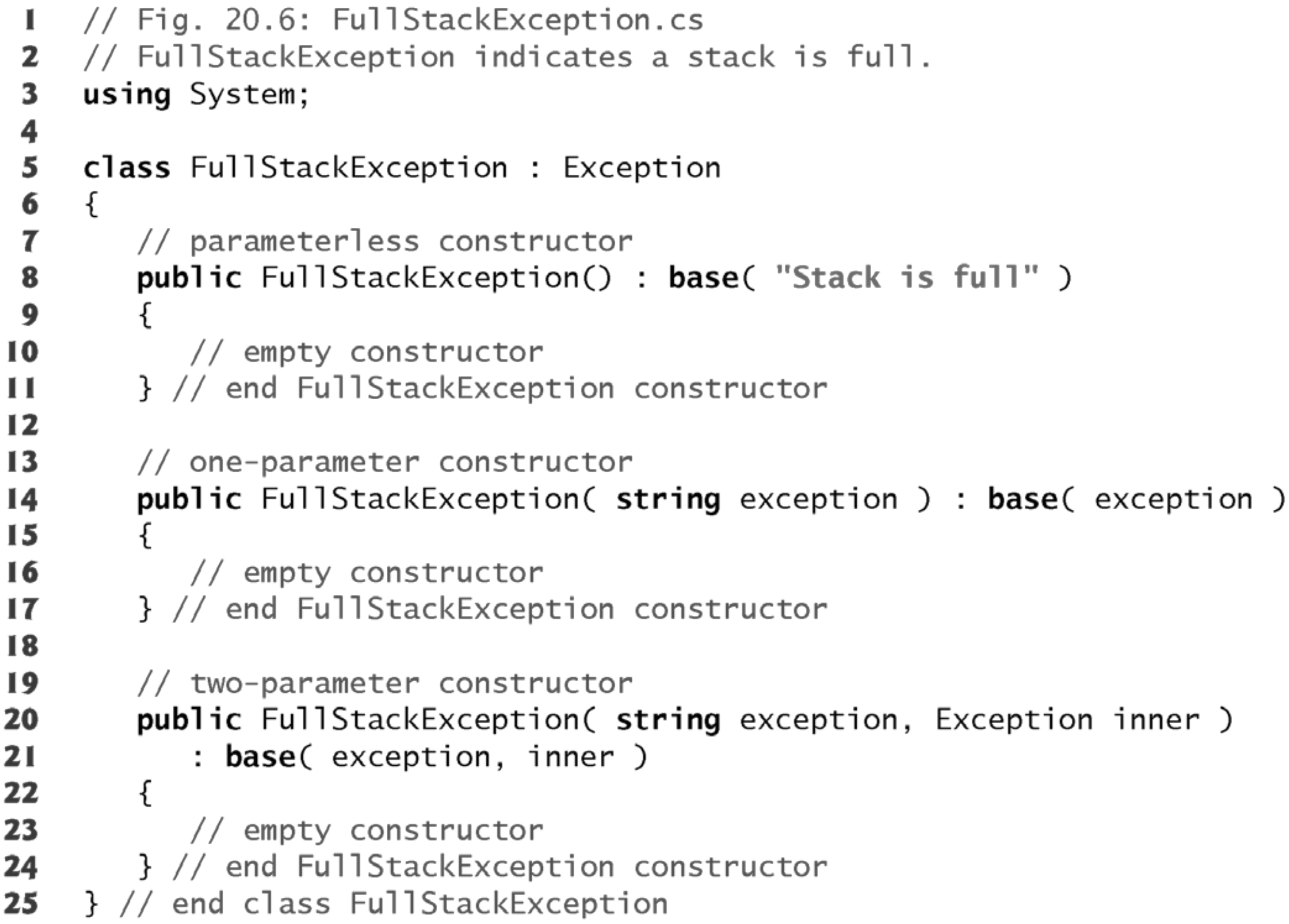


```
      --top; // decrement top
      return elements[ top + 1 ]; // return top value
   } // end method Pop
} // end class Stack
```

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

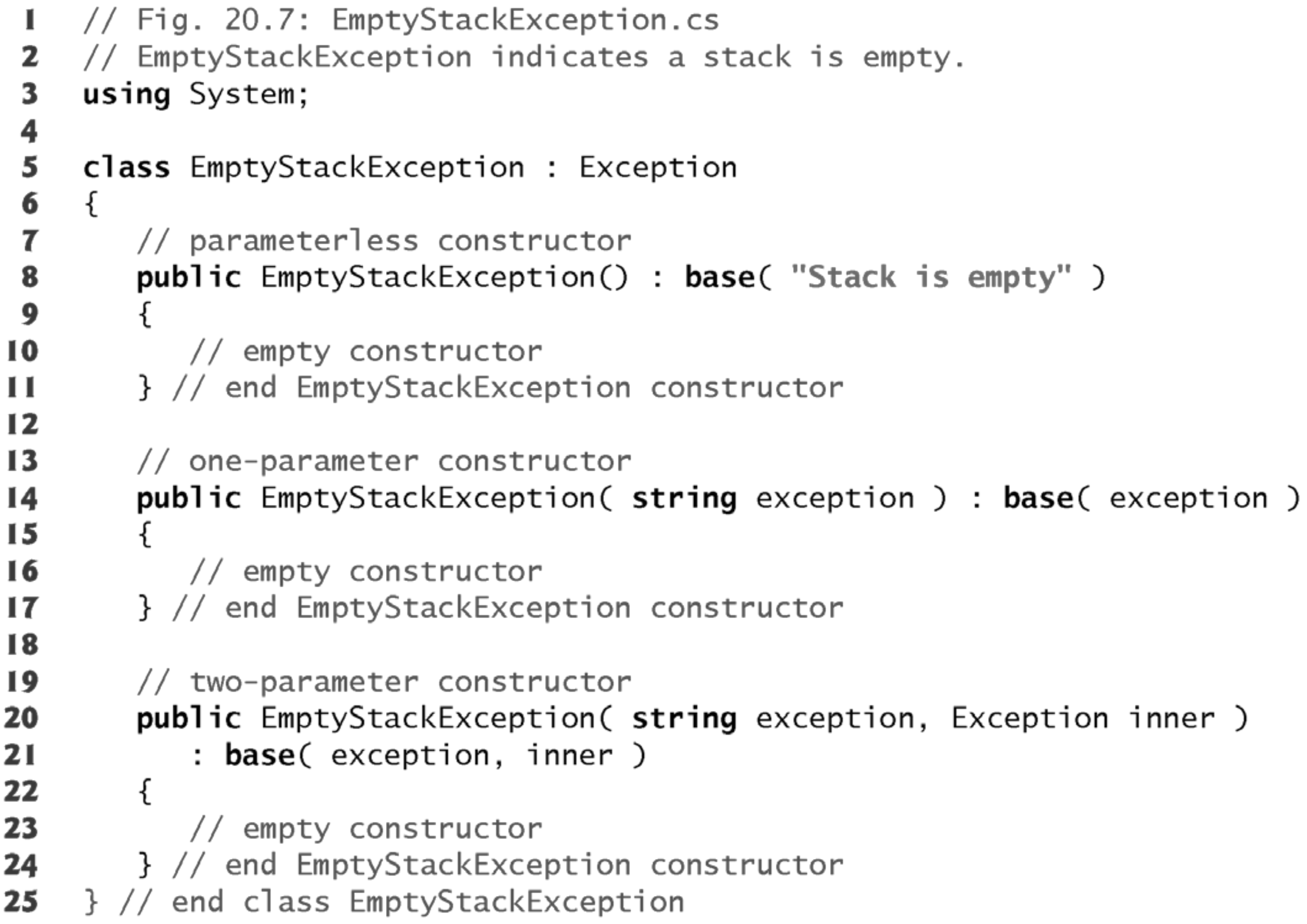
20.6 Generic Classes (cont.)

- Classes `FullStackException` (Fig. 20.6) and `EmptyStackException` (Fig. 20.7) each provide a parameterless constructor, a one-argument constructor of exception classes (as discussed in Section 13.8) and a two-argument constructor for creating a new exception using an existing one.
- The parameterless constructor sets the default error message while the other two constructors set custom error messages.



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

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

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

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

Fig. 20.6 | `FullStackException` indicates a stack is full.

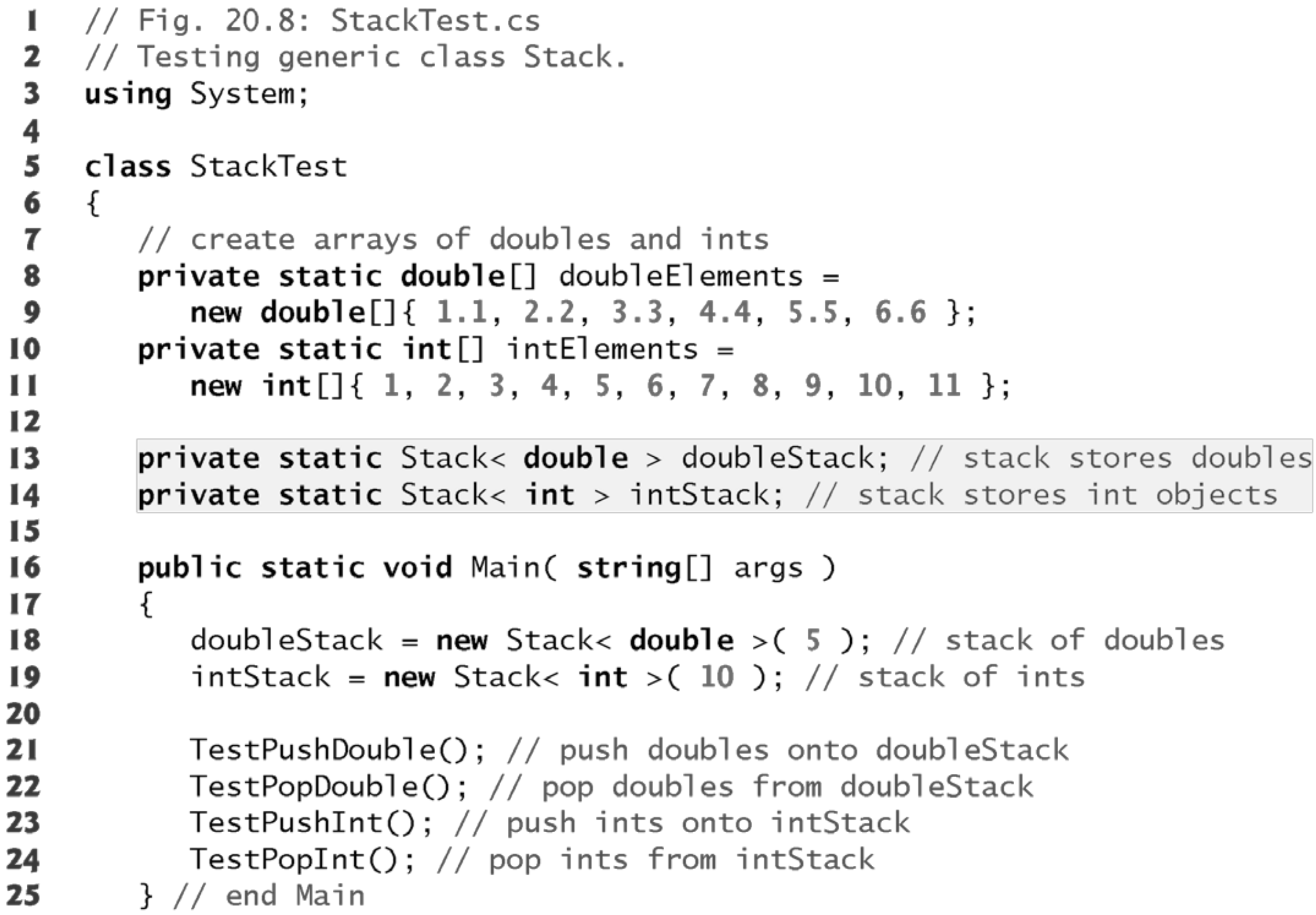


```
// Fig. 20.7: EmptyStackException.cs
// EmptyStackException indicates a stack is empty.
using System;

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

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

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

Fig. 20.7 | EmptyStackException indicates a stack is empty.

20.6 Generic Classes (cont.)

- Now, let’s consider an app (Fig. 20.8) that uses the `Stack` generic class.



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

class StackTest
{
   // create arrays of doubles and ints
   private static double[] doubleElements =
      new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
   private static int[] intElements =
      new int[]{ 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

   public static void Main( string[] args )
   {
      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
   } // end Main
```

Fig. 20.8 | Testing generic class Stack. (Part 1 of 7.)

```
   // 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( "{0:F1} ", element );
            doubleStack.Push( element ); // push onto doubleStack
         } // end foreach
      } // end try
      catch ( FullStackException exception )
      {
         Console.Error.WriteLine();
         Console.Error.WriteLine( "Message: " + exception.Message );
         Console.Error.WriteLine( exception.StackTrace );
      } // end catch
   } // end method TestPushDouble

```

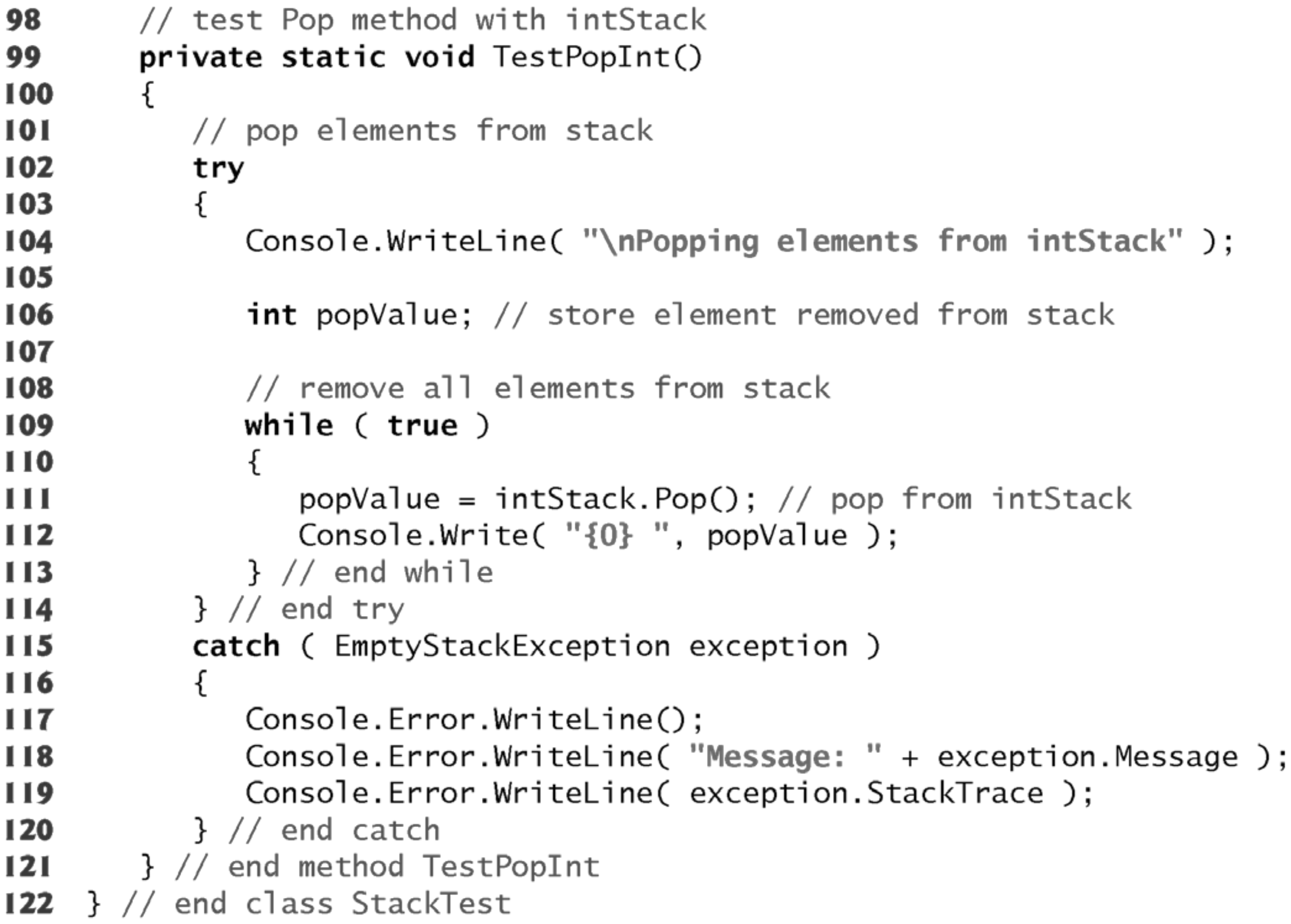
Fig. 20.8 | Testing generic class Stack. (Part 2 of 7.)

```
// 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( "{0:F1} ", popValue );
      } // end while
   } // end try
   catch ( EmptyStackException exception )
   {
      Console.Error.WriteLine();
      Console.Error.WriteLine( "Message: " + exception.Message );
      Console.Error.WriteLine( exception.StackTrace );
   } // end catch
} // end method TestPopDouble
```

Fig. 20.8 | Testing generic class Stack. (Part 3 of 7.)

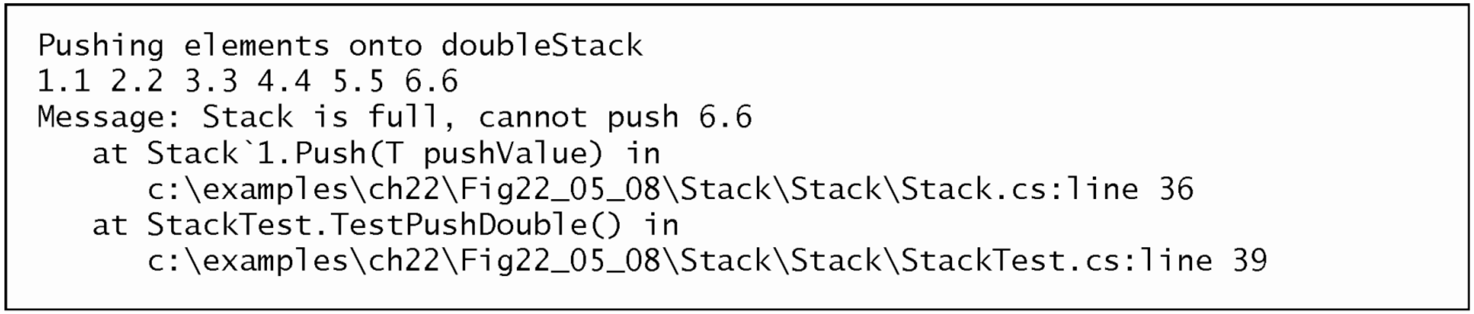
```
   // test Push method with intStack
   private static void TestPushInt()
   {
      // push elements onto stack
      try
      {
         Console.WriteLine( "\nPushing elements onto intStack" );

         // push elements onto stack
         foreach ( var element in intElements )
         {
            Console.Write( "{0} ", element );
            intStack.Push( element ); // push onto intStack
         } // end foreach
      } // end try
      catch ( FullStackException exception )
      {
         Console.Error.WriteLine();
         Console.Error.WriteLine( "Message: " + exception.Message );
         Console.Error.WriteLine( exception.StackTrace );
      } // end catch
   } // end method TestPushInt

```

Fig. 20.8 | Testing generic class Stack. (Part 4 of 7.)



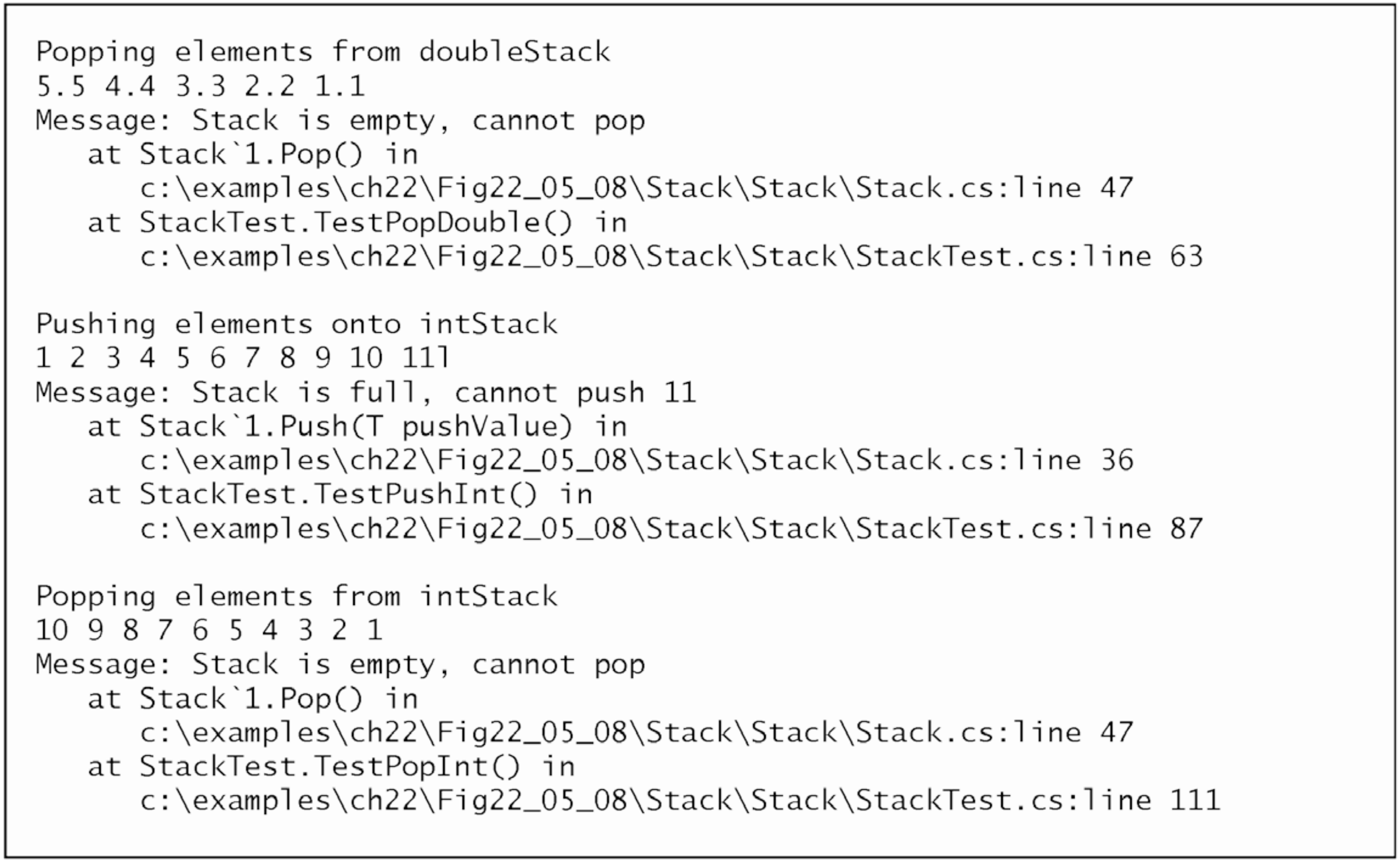
```
   // test Pop method with intStack
   private static void TestPopInt()
   {
      // pop elements from stack
      try
      {
         Console.WriteLine( "\nPopping elements from intStack" );

         int popValue; // store element removed from stack

         // remove all elements from stack
         while ( true )
         {
            popValue = intStack.Pop(); // pop from intStack
            Console.Write( "{0} ", popValue );
         } // end while
      } // end try
      catch ( EmptyStackException exception )
      {
         Console.Error.WriteLine();
         Console.Error.WriteLine( "Message: " + exception.Message );
         Console.Error.WriteLine( exception.StackTrace );
      } // end catch
   } // end method TestPopInt
} // end class StackTest
```

Fig. 20.8 | Testing generic class Stack. (Part 5 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(T pushValue) in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPushDouble() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 39
```

Fig. 20.8 | Testing generic class `Stack`. (Part 6 of 7.)

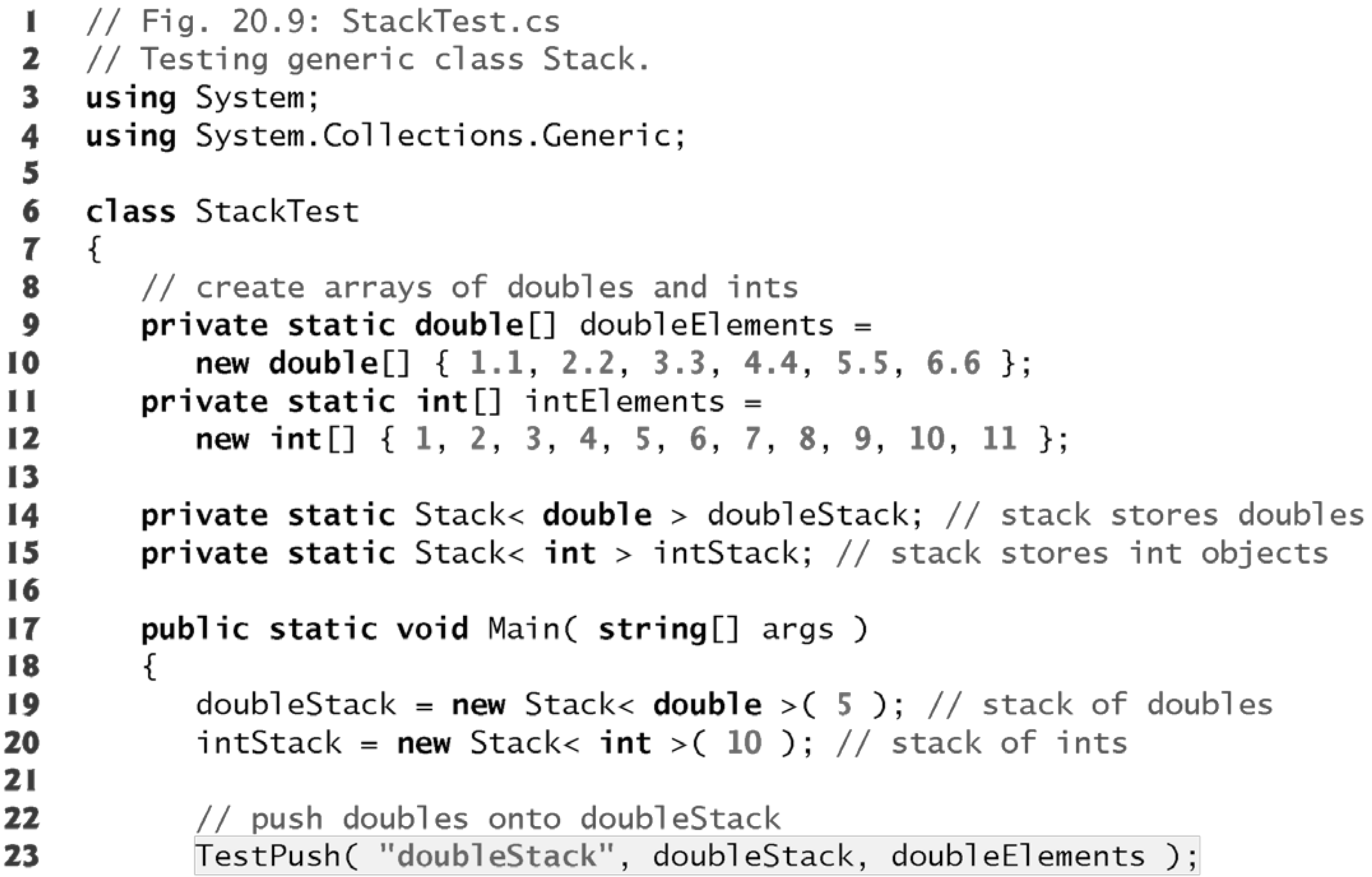


```
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPopDouble() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 63

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 111
Message: Stack is full, cannot push 11
   at Stack`1.Push(T pushValue) in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPushInt() in
      c:\examples\ch22\Fig22_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\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPopInt() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 111
```

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

20.6 Generic Classes (cont.)

- Figure 20.9 declares generic method `TestPush` (lines 33–54) to perform the same tasks as `TestPushDouble` and `TestPushInt` in Fig. 20.8—that is, `Push` values onto a `Stack<T>`.
- Similarly, generic method `TestPop` (lines 57–79) performs the same tasks as `TestPopDouble` and `TestPopInt` in Fig. 20.8—that is, `Pop` values off a `Stack<T>`.

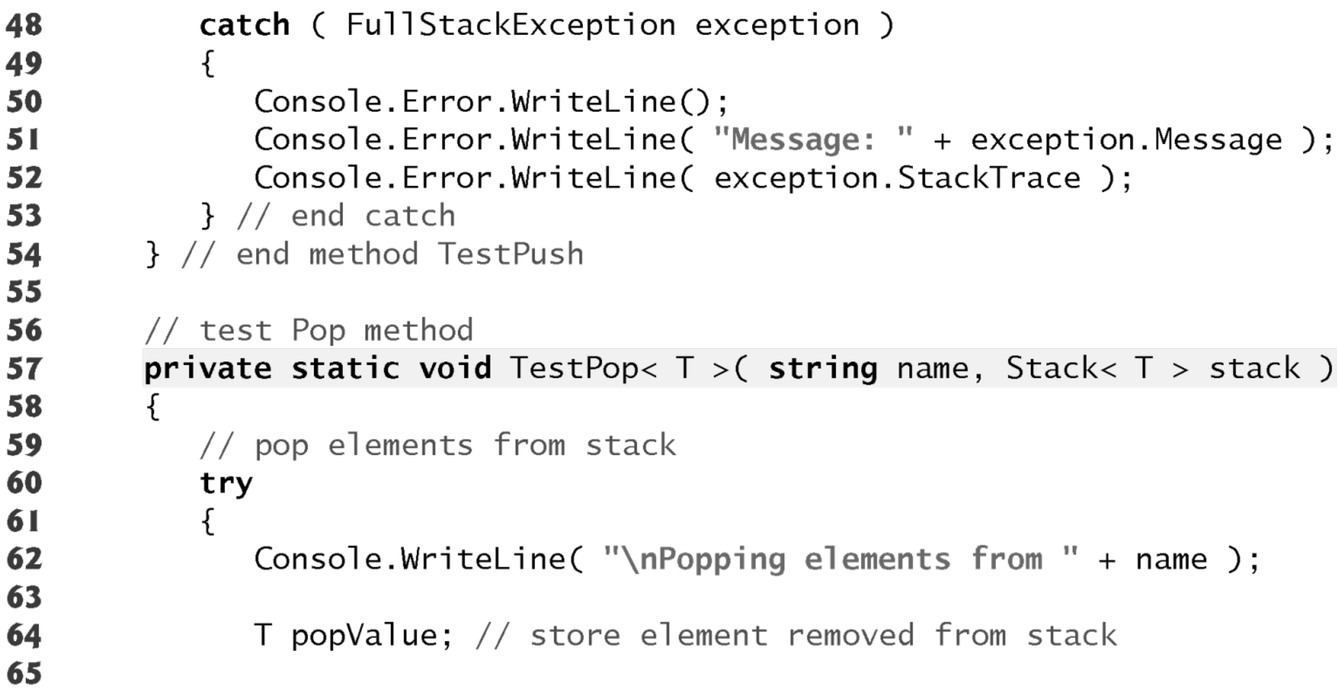


```
// Fig. 20.9: StackTest.cs
// Testing generic class Stack.
using System;
using System.Collections.Generic;

class StackTest
{
   // create arrays of doubles and ints
   private static double[] doubleElements =
      new double[] { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
   private static int[] intElements =
      new int[] { 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

   public static void Main( string[] args )
   {
      doubleStack = new Stack< double >( 5 ); // stack of doubles
      intStack = new Stack< int >( 10 ); // stack of ints

      // push doubles onto doubleStack
      TestPush( "doubleStack", doubleStack, doubleElements );
```

Fig. 20.9 | Testing generic class Stack. (Part 1 of 5.)

```
      // pop doubles from doubleStack
      TestPop( "doubleStack", doubleStack );
      // push ints onto intStack
      TestPush( "intStack", intStack, intElements );
      // pop ints from intStack
      TestPop( "intStack", intStack );
   } // end Main

   // 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( "{0} ", element );
            stack.Push( element ); // push onto stack
         } // end foreach
      } // end try
```

Fig. 20.9 | Testing generic class Stack. (Part 2 of 5.)



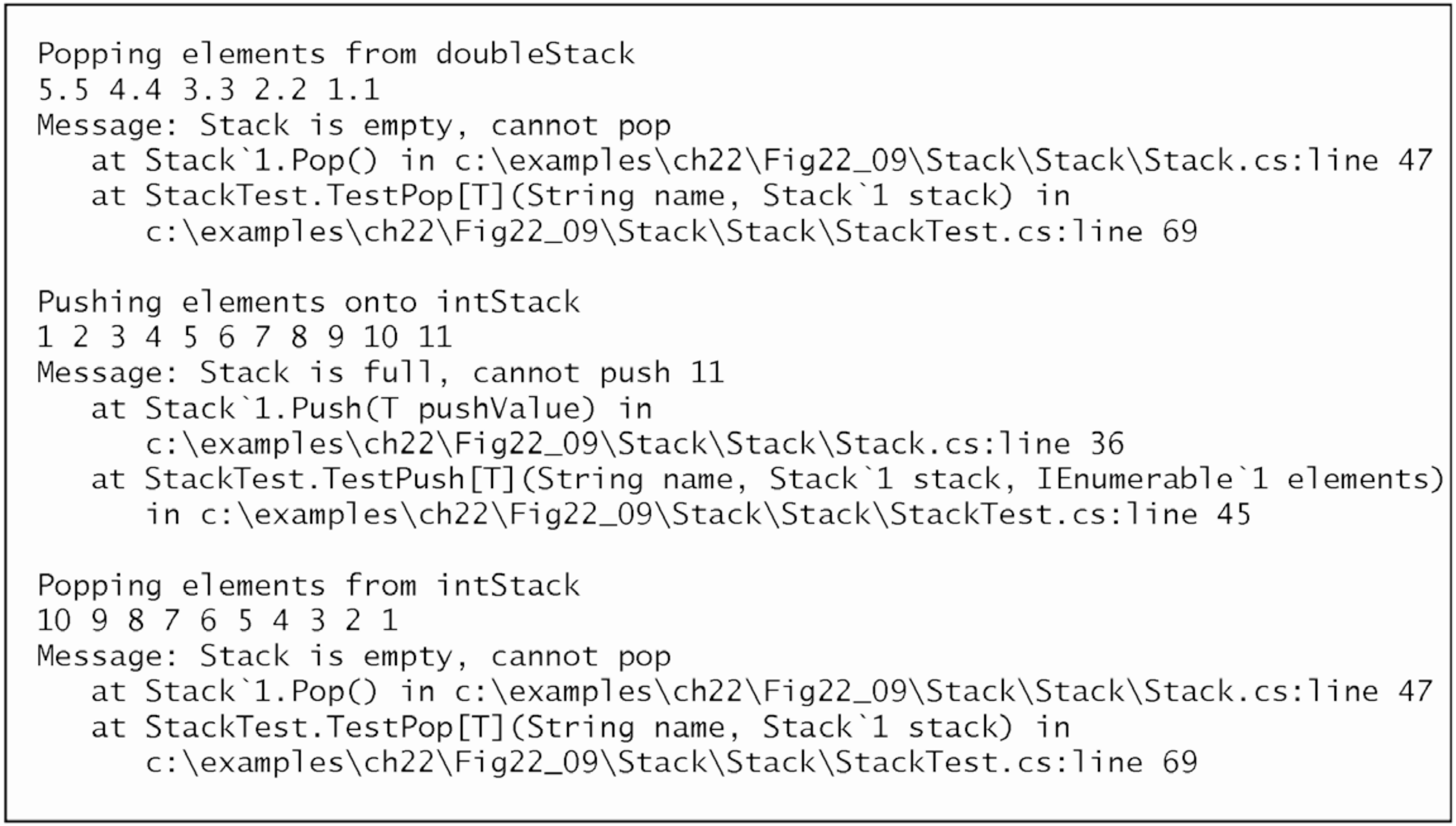
```
      catch ( FullStackException exception )
      {
         Console.Error.WriteLine();
         Console.Error.WriteLine( "Message: " + exception.Message );
         Console.Error.WriteLine( exception.StackTrace );
      } // end catch
   } // end method TestPush

   // 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

```

Fig. 20.9 | Testing generic class Stack. (Part 3 of 5.)

```
         // remove all elements from stack
         while ( true )
         {
            popValue = stack.Pop(); // pop from stack
            Console.Write( "{0} ", popValue );
         } // end while
      } // end try
      catch ( EmptyStackException exception )
      {
         Console.Error.WriteLine();
         Console.Error.WriteLine( "Message: " + exception.Message );
         Console.Error.WriteLine( exception.StackTrace );
      } // end catch
   } // end TestPop
} // end class StackTest
```

```
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\ch22\Fig22_09\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1 elements)
      in c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 45
```

Fig. 20.9 | Testing generic class Stack. (Part 4 of 5.)



```
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\ch22\Fig22_09\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPop[T](String name, Stack`1 stack) in
      c:\examples\ch22\Fig22_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\ch22\Fig22_09\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1 elements)
      in c:\examples\ch22\Fig22_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\ch22\Fig22_09\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPop[T](String name, Stack`1 stack) in
      c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 69
```

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