Exception Handling: A Deeper Look

Chapter 13 of Visual C# How to Program, 6/e

OBJECTIVES

In this chapter you'll:

- Understand what exceptions are and how they're handled.
- Learn when to use exception handling.
- Use `try` blocks to delimit code that may throw exceptions.
- Use `throw` to indicate a problem at runtime.
- Use `catch` blocks to specify exception handlers.
- Understand what happens to uncaught exceptions.
- Understand the mechanics of the termination model of exception handling.

OBJECTIVES

- Use the `finally` block to release resources.
- See how a `using` statement can auto-release resources.
- Understand .NET exception class hierarchy.
- Use `Exception` properties.
- Create new exception types.
- Use C# 6's null-conditional operator (`?.`) to determine whether a reference is null before using it to call a method or access a property.
- Use nullable value types to specify that a variable may contain a value or `null`.
- Use C# 6 exception filters to specify a condition for catching an exception.

13.1 Introduction

- An exception is an indication of a problem that occurred during a program's execution.
- Exception handling enables you to create apps that can handle exceptions—in many cases allowing a program to continue executing as if no problems were encountered.
- Exception handling enables you to write clear, robust and more fault-tolerant programs.

13.2 Example: Divide by Zero without Exception Handling

- Figure 13.1's inputs two integers from the users, then divides the first integer by the second using integer division to obtain an int result.
- In this example, an exception is thrown (i.e., an exception occurs) when a method detects a problem.

```
1  // Fig. 13.1: DivideByZeroNoExceptionHandling.cs
2  // Integer division without exception handling.
3  using System;
4
5  class DivideByZeroNoExceptionHandling
6  {
7     static void Main()
8     {
9        // get numerator
10       Console.Write("Please enter an integer numerator: ");
11       var numerator = int.Parse(Console.ReadLine());
12
13       // get denominator
14       Console.Write("Please enter an integer denominator: ");
15       var denominator = int.Parse(Console.ReadLine());
16
17       // divide the two integers, then display the result
18       var result = numerator / denominator;
19       Console.WriteLine(
20          $"\nResult: {numerator} / {denominator} = {result}");
21    }
22 }
```

Fig. 13.1 | Integer division without exception handling. (Part 1 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Unhandled Exception: System.DivideByZeroException:
   Attempted to divide by zero.
   at DivideByZeroNoExceptionHandling.Main()
      in C:\Users\PaulDeitel\Documents\examples\ch13\Fig13_01\
      DivideByZeroNoExceptionHandling\DivideByZeroNoExceptionHandling\
      DivideByZeroNoExceptionHandling.cs:line 18
```

Fig. 13.1 | Integer division without exception handling. (Part 2 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Unhandled Exception: System.FormatException:
   Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
      NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style,
      NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at DivideByZeroNoExceptionHandling.Main()
      in C:\Users\PaulDeitel\Documents\examples\ch13\Fig13_01\
      DivideByZeroNoExceptionHandling\DivideByZeroNoExceptionHandling\
      DivideByZeroNoExceptionHandling.cs:line 15
```

Fig. 13.1 | Integer division without exception handling. (Part 3 of 3.)

13.2 Example: Divide by Zero without Exception Handling (cont.)

- The example in Fig. 13.1 might cause exceptions, depending on the user's input.
- Execute this app with **Debug > Start Without Debugging**.
- If an exception occurs during execution, a dialog appears indicating that the app "has stopped working."
- Click **Cancel** or **Close Program** to terminate the app.
- An error message describing the exception that occurred is displayed in the program's output.

13.2 Example: Divide by Zero without Exception Handling (cont.)

- A **stack trace** includes the exception class's name in a message indicating the problem that occurred and the path of execution that led to the exception, method by method.
- Stack traces help you debug a program.
- The first line of the error message specifies the exception that occurred.
- When a program divides an integer by 0, the CLR throws a **`DivideByZeroException`**.
- Division by zero is not allowed in integer arithmetic.

13.2 Example: Divide by Zero without Exception Handling (cont.)

- A **`FormatException`** occurs when `int.Parse` receives a `string` that does not represent a valid integer.
- This program terminates when an *unhandled* exception occurs.
- This does not always happen—sometimes a program may continue executing even though an exception has occurred and a stack trace has been displayed.
- In such cases, the app may produce incorrect results.

13.3 Example: Handling DivideByZeroExceptions and FormatExceptions

- The app in Fig. 13.2 uses exception handling to process `DivideByZeroExceptions` and `FormatExceptions` that might arise.
- This program demonstrates how to catch and handle exceptions—in this case, displaying an error message and allowing the user to enter another set of values.

```
1  // Fig. 13.2: DivideByZeroExceptionHandling.cs
2  // FormatException and DivideByZeroException handlers.
3  using System;
4
5  class DivideByZeroExceptionHandling
6  {
7     static void Main(string[] args)
8     {
9        var continueLoop = true; // determines whether to keep looping
10
11       do
12       {
13          // retrieve user input and calculate quotient
14          try
15          {
16             // int.Parse generates FormatException
17             // if argument cannot be converted to an integer
18             Console.Write("Enter an integer numerator: ");
19             var numerator = int.Parse(Console.ReadLine());
20             Console.Write("Enter an integer denominator: ");
21             var denominator = int.Parse(Console.ReadLine());
22
```

Fig. 13.2 | FormatException and DivideByZeroException handlers. (Part 1 of 4.)

```
23              // division generates DivideByZeroException
24              // if denominator is 0
25              var result = numerator / denominator;
26
27              // display result
28              Console.WriteLine(
29                 $"\nResult: {numerator} / {denominator} = {result}");
30              continueLoop = false;
31           }
32           catch (FormatException formatException)
33           {
34              Console.WriteLine($"\n{formatException.Message}");
35              Console.WriteLine(
36                 "You must enter two integers. Please try again.\n");
37           }
38           catch (DivideByZeroException divideByZeroException)
39           {
40              Console.WriteLine($"\n{divideByZeroException.Message}");
41              Console.WriteLine(
42                 "Zero is an invalid denominator. Please try again.\n");
43           }
44        } while (continueLoop);
45     }
46  }
```

Fig. 13.2 | FormatException and DivideByZeroException handlers. (Part 2 of 4.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Enter an integer numerator: 100
Enter an integer denominator: 0

Attempted to divide by zero.
Zero is an invalid denominator. Please try again.

Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 13.2 | FormatException and DivideByZeroException handlers. (Part 3 of 4.)

```
Enter an integer numerator: 100
Enter an integer denominator: hello

Input string was not in a correct format.
You must enter two integers. Please try again.

Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 13.2 | `FormatException` and `DivideByZeroException` handlers. (Part 4 of 4.)

13.3 Example: Handling DivideByZeroExceptions and FormatExceptions (cont).

- The `Int32.TryParse` method converts a `string` to an `int` value if possible.
- The method requires two arguments—one is the `string` to parse and the other is the variable in which the converted value is to be stored.
- The method returns a `bool` value that's `true` if the `string` was parsed successfully.
- If the `string` could not be converted, the value `0` is assigned to the second argument.

Error-Prevention Tip 13.1

Method `TryParse` *can be used to validate input in code rather than allowing the code to throw an exception—this technique is generally preferred.*

13.3.1 Enclosing Code in a `try` Block

- A `try` block encloses code that might throw exceptions, as well as the code that's skipped when an exception occurs.

13.3.2 Catching Exceptions

- When an exception occurs in a `try` block, a corresponding `catch` block catches the exception and handles it.
- At least one `catch` block must immediately follow a `try` block.
- A `catch` block specifies an exception parameter representing the exception that the `catch` block can handle.
- Optionally, you can include a `catch` block that does not specify an exception type to catch all exception types.

13.3.3 Uncaught Exceptions

- An uncaught exception (or unhandled exception) is an exception for which there is no matching catch block.
- If you run the app by using **Debug > Start Debugging** and the runtime environment detects an uncaught exception, the app pauses, and the **Exception Assistant** (Fig. 13.3) appears.

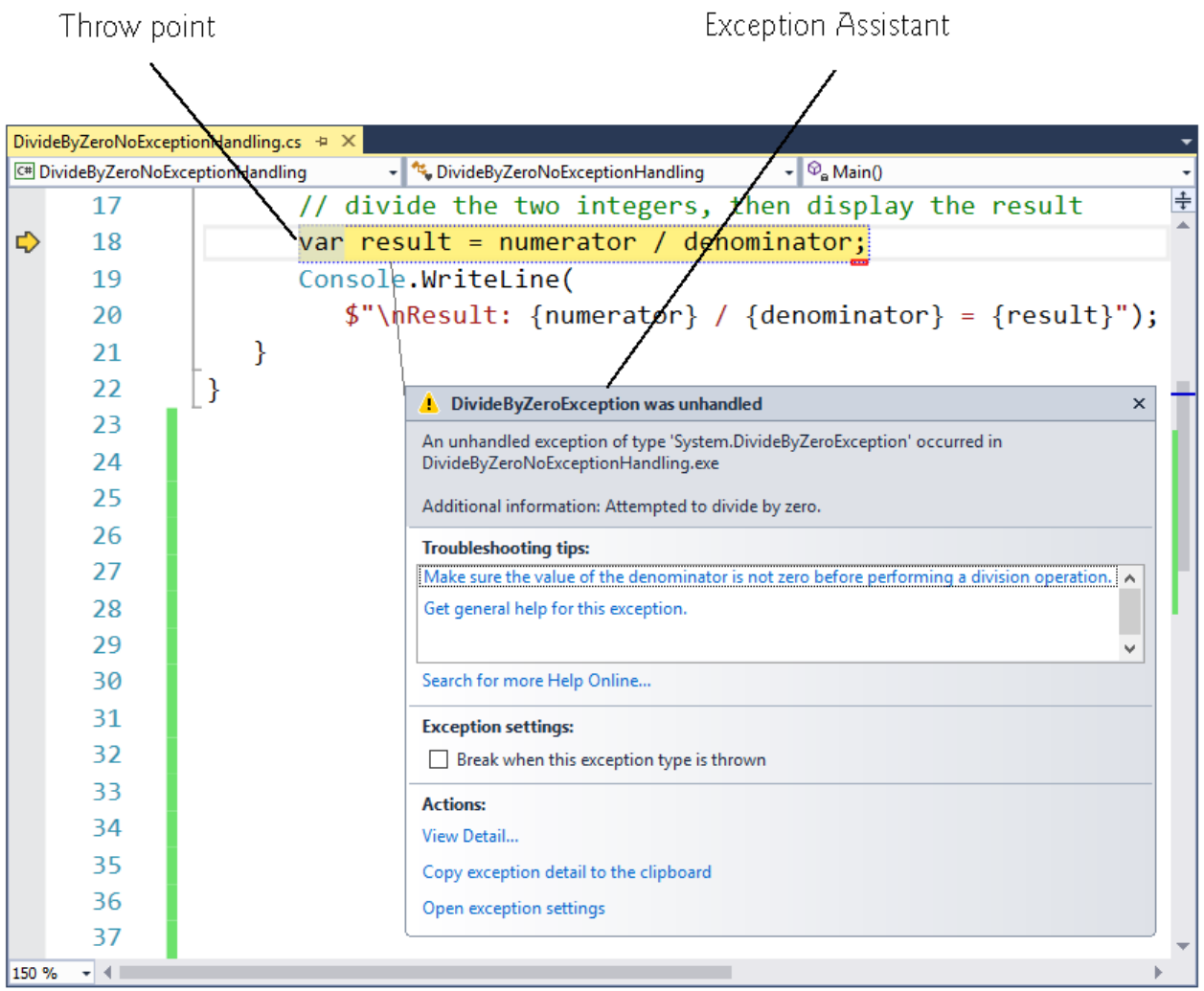

Fig. 13.3 | Exception Assistant.

13.3.4 Termination Model of Exception Handling

- The point at which an exception occurs is called the throw point.
- If an exception occurs in a `try` block, program control immediately transfers to the first `catch` block matching the type of the thrown exception.
- After the exception is handled, program control resumes after the last `catch` block.
- This is known as the termination model of exception handling.

Common Programming Error 13.1

Specifying a comma-separated list of parameters in a `catch` *block is a syntax error. A* `catch` *block can have at most one parameter. Section 13.10 shows how you can use exception filters to specify additional conditions for which an exception can be caught.*

13.4 .NET Exception Hierarchy

- In C#, only objects of class `Exception` and its derived classes may be thrown and caught.
- Exceptions thrown in other .NET languages can be caught with the general `catch` clause.

13.4.1 Class SystemException

- Class `Exception` is the base class of .NET's exception class hierarchy.
- The CLR generates `SystemExceptions`.
- If a program attempts to access an out-of-range array index, the CLR throws an exception of type `IndexOutOfRangeException`.
- Attempting to use a `null` reference causes a `NullReferenceException`.

13.4.1 Class SystemException

- A `catch` block can use a base-class type to catch a hierarchy of related exception types.
- A `catch` block that specifies a parameter of type `Exception` can catch all exceptions.
- This technique makes sense only if the handling behavior is the same for a base class and all derived classes.

Common Programming Error 13.2

The compiler issues an error if a `catch` *block that catches a base-class exception is placed* before *a* `catch` *block for any of that class's derived-class types. In this case, the base-class* `catch` *block would catch all base-class and derived-class exceptions, so the derived-class exception handler would never execute.*

13.4.2 Which Exceptions Might a Method Throw?

- Search for "`int.Parse` method" in the Visual Studio online documentation.
- The **Exceptions** section of this method's web page indicates that method `int.Parse` throws exception types
  - `ArgumentNullException`
  - `FormatException`
  - `OverflowException`
- and describes the reason for each.

Software Engineering Observation 13.1

If a method may throw exceptions, statements that invoke the method directly or indirectly should be placed in `try` *blocks, and those exceptions should be caught and handled.*

13.5 finally Block

- Programs frequently request and release resources dynamically.
- Operating systems typically prevent more than one program from manipulating a file.
- Therefore, the program should close the file (i.e., release the resource) so other programs can use it.
- If the file is not closed, a resource leak occurs.

Error-Prevention Tip 13.2

The CLR does not completely eliminate memory leaks. It will not garbage-collect an object until the program contains no more references to that object, and even then there may be a delay until the memory is required. Thus, memory leaks can occur if you inadvertently keep references to unwanted objects.

13.5.1 Moving Resource-Release Code to a `finally` Block

- Exceptions often occur when an app processes resources that require explicit release.
- Regardless of whether a program experiences exceptions, the program should close the file when it is no longer needed.
- C# provides the `finally` block, which is guaranteed to execute regardless of whether an exception occurs.
- This makes the `finally` block ideal to release resources from the corresponding try block.

13.5 finally Block (Cont.)

- Local variables in a `try` block cannot be accessed in the corresponding `finally` block, so variables that must be accessed in both should be declared before the `try` block.

Error-Prevention Tip 13.3

A `finally` *block typically contains code to release resources acquired in the corresponding* `try` *block, which makes the* `finally` *block an effective mechanism for eliminating resource leaks.*

Performance Tip 13.1

As a rule, resources should be released as soon as they're no longer needed in a program. This makes them available for reuse promptly.

13.5.2 Demonstrating the `finally` Block

- The app in Fig. 13.4 demonstrates that the `finally` block always executes, regardless of whether an exception occurs in the corresponding try block.

```
1  // Fig. 13.4: UsingExceptions.cs
2  // finally blocks always execute, even when no exception occurs.
3
4  using System;
5
6  class UsingExceptions
7  {
8     static void Main()
9     {
10       // Case 1: No exceptions occur in called method
11       Console.WriteLine("Calling DoesNotThrowException");
12       DoesNotThrowException();
13
14       // Case 2: Exception occurs and is caught in called method
15       Console.WriteLine("\nCalling ThrowExceptionWithCatch");
16       ThrowExceptionWithCatch();
17
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 1 of 9.)

```
18          // Case 3: Exception occurs, but is not caught in called method
19          // because there is no catch block.
20          Console.WriteLine("\nCalling ThrowExceptionWithoutCatch");
21
22          // call ThrowExceptionWithoutCatch
23          try
24          {
25             ThrowExceptionWithoutCatch();
26          }
27          catch
28          {
29             Console.WriteLine(
30                "Caught exception from ThrowExceptionWithoutCatch in Main");
31          }
32
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 2 of 9.)

```
33          // Case 4: Exception occurs and is caught in called method,
34          // then rethrown to caller.
35          Console.WriteLine("\nCalling ThrowExceptionCatchRethrow");
36
37          // call ThrowExceptionCatchRethrow
38          try
39          {
40             ThrowExceptionCatchRethrow();
41          }
42          catch
43          {
44             Console.WriteLine(
45                "Caught exception from ThrowExceptionCatchRethrow in Main");
46          }
47       }
48
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 3 of 9.)

```
49      // no exceptions thrown
50      static void DoesNotThrowException()
51      {
52         // try block does not throw any exceptions
53         try
54         {
55            Console.WriteLine("In DoesNotThrowException");
56         }
57         catch
58         {
59            Console.WriteLine("This catch never executes");
60         }
61         finally
62         {
63            Console.WriteLine("finally executed in DoesNotThrowException");
64         }
65
66         Console.WriteLine("End of DoesNotThrowException");
67      }
68
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 4 of 9.)

```
69      // throws exception and catches it locally
70      static void ThrowExceptionWithCatch()
71      {
72         // try block throws exception
73         try
74         {
75            Console.WriteLine("In ThrowExceptionWithCatch");
76            throw new Exception("Exception in ThrowExceptionWithCatch");
77         }
78         catch (Exception exceptionParameter)
79         {
80            Console.WriteLine($"Message: {exceptionParameter.Message}");
81         }
82         finally
83         {
84            Console.WriteLine(
85               "finally executed in ThrowExceptionWithCatch");
86         }
87
88         Console.WriteLine("End of ThrowExceptionWithCatch");
89      }
90
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 5 of 9.)

```
91      // throws exception and does not catch it locally
92      static void ThrowExceptionWithoutCatch()
93      {
94         // throw exception, but do not catch it
95         try
96         {
97            Console.WriteLine("In ThrowExceptionWithoutCatch");
98            throw new Exception("Exception in ThrowExceptionWithoutCatch");
99         }
100        finally
101        {
102           Console.WriteLine(
103              "finally executed in ThrowExceptionWithoutCatch");
104        }
105
106        // unreachable code; logic error
107        Console.WriteLine("End of ThrowExceptionWithoutCatch");
108     }
109
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 6 of 9.)

```
110     // throws exception, catches it and rethrows it
111     static void ThrowExceptionCatchRethrow()
112     {
113        // try block throws exception
114        try
115        {
116           Console.WriteLine("In ThrowExceptionCatchRethrow");
117           throw new Exception("Exception in ThrowExceptionCatchRethrow");
118        }
119        catch (Exception exceptionParameter)
120        {
121           Console.WriteLine("Message: " + exceptionParameter.Message);
122
123           // rethrow exception for further processing
124           throw;
125
126           // unreachable code; logic error
127        }
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 7 of 9.)

```
128         finally
129         {
130            Console.WriteLine(
131               "finally executed in ThrowExceptionCatchRethrow");
132         }
133
134         // any code placed here is never reached
135         Console.WriteLine("End of ThrowExceptionCatchRethrow");
136      }
137   }
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 8 of 9.)

```
Calling DoesNotThrowException
In DoesNotThrowException
finally executed in DoesNotThrowException
End of DoesNotThrowException

Calling ThrowExceptionWithCatch
In ThrowExceptionWithCatch
Message: Exception in ThrowExceptionWithCatch
finally executed in ThrowExceptionWithCatch
End of ThrowExceptionWithCatch

Calling ThrowExceptionWithoutCatch
In ThrowExceptionWithoutCatch
finally executed in ThrowExceptionWithoutCatch
Caught exception from ThrowExceptionWithoutCatch in Main

Calling ThrowExceptionCatchRethrow
In ThrowExceptionCatchRethrow
Message: Exception in ThrowExceptionCatchRethrow
finally executed in ThrowExceptionCatchRethrow
Caught exception from ThrowExceptionCatchRethrow in Main
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 9 of 9.)

13.5.3 Throwing Exceptions Using the `throw` Statement

- You create and throw an exception with the **`throw` statement**
- Executing the `throw` statement indicates that a problem has occurred in the code.
- A `throw` statement specifies an object to be thrown. The operand of a `throw` statement can be of type `Exception` or of any type derived from class `Exception`.

13.5.4 Rethrowing Exceptions

- You also can rethrow an exception with a version of the `throw` statement which takes an operand that's the reference to the exception that was caught.
- This form of throw statement *resets the throw point*, so the original throw point's stack-trace information is *lost*.
- Later, you'll see that after an exception is caught, you can create and throw a different type of exception object from the catch block and you can include the original exception as part of the new exception object.
- Class library designers often do this to *customize* the exception types thrown from methods in their class libraries or to provide additional debugging information

Software Engineering Observation 13.2

In general, it's considered better practice to throw a new exception and pass the original one to the new exception's constructor, rather than rethrowing the original exception. This maintains all of the stack-trace information from the original exception. We demonstrate passing an existing exception to a new exception's constructor in Section 13.7.3.

13.5.5 Returning After a `finally` Block

- If a `try` block successfully completes, or if a `catch` block catches and handles an exception, the program continues its execution with the next statement after the `finally` block.
- If an exception is not caught, or if a `catch` block rethrows an exception, program control continues in the next enclosing `try` block.
- The enclosing `try` could be in the calling method or in one of its callers.
- It also is possible to nest a `try` statement in a `try` block; in such a case, the outer `try` statement's `catch` blocks would process any exceptions that were not caught in the inner `try` statement.
- If a `try` block executes and has a corresponding `finally` block, the `finally` block executes even if the `try` block terminates due to a `return` statement.
- The return occurs after the execution of the `finally` block.

Common Programming Error 13.3

If an uncaught exception is awaiting processing when the `finally` *block executes, and the* `finally` *block throws a new exception that's not caught in the* `finally` *block, the first exception is* lost, *and the new exception is passed to the next enclosing* `try` *block.*

Error-Prevention Tip 13.4

When placing code that can throw an exception in a `finally` *block, always enclose the code in a* `try` *statement that catches the appropriate exception types. This prevents the loss of any uncaught and rethrown exceptions that occur before the* `finally` *block executes.*

Software Engineering Observation 13.3

Do not place `try` *blocks around every statement that might throw an exception—this can make programs difficult to read. Instead, place one* `try` *block around a significant portion of code, and follow this* `try` *block with* `catch` *blocks that handle each possible exception. Then follow the* `catch` *blocks with a single* `finally` *block. Use separate* `try` *blocks to distinguish between multiple statements that can throw the same exception type.*

13.6 The using statement

- The using statement simplifies writing code in which you obtain a resource.
- The general form of a using statement is:
  - ```
    using (var exampleObject = new ExampleClass())
    {
       exampleObject.SomeMethod();
    }
    ```
    ```

13.6 The using statement (Cont.)

- The using statement code is equivalent to
  - ```
    {
       var exampleObject = new ExampleClass();
       try
       {
          exampleObject.SomeMethod();
       }
       finally
       {
          if (exampleObject != null)
          {
             ((IDisposable) exampleObject).Dispose();
          }
       }
    }
    ```
    ```

13.7 Exception Properties

- Class Exception's properties are used to formulate error messages indicating a caught exception.
  - Property Message stores the error message associated with an Exception object.
  - Property StackTrace contains a string that represents the method-call stack.
- If the debugging information that's generated by the compiler for the method is accessible to the IDE, the stack trace also includes line numbers; the first line number indicates the throw point, and subsequent line numbers indicate the locations from which the methods in the stack trace were called.
- The IDE creates PDB files to maintain the debugging information for your projects.

13.7.1 Property `InnerException`

- When an exception occurs, a programmer might use a different error message or indicate a new exception type.
- The original exception object is stored in the `InnerException` property.

13.7.2 Other `Exception` Properties

- Class `Exception` provides other properties:
  - `HelpLink` specifies the location of a help file that describes the problem.
  - `Source` specifies the name of the app or object that caused the exception.
  - `TargetSite` specifies the method where the exception originated.

13.7.3 Demonstrating Exception Properties and Stack Unwinding

- Fig. 13.5 demonstrates properties of class `Exception`.

```
1  // Fig. 13.5: Properties.cs
2  // Stack unwinding and Exception class properties.
3  // Demonstrates using properties Message, StackTrace and InnerException.
4  using System;
5
6  class Properties
7  {
8     static void Main()
9     {
```

Fig. 13.5 | Stack unwinding and `Exception` class properties. (Part 1 of 6.)

```
10          // call Method1; any Exception generated is caught
11          // in the catch block that follows
12          try
13          {
14             Method1();
15          }
16          catch (Exception exceptionParameter)
17          {
18             // output the string representation of the Exception, then output
19             // properties Message, StackTrace and InnerException
20             Console.WriteLine("exceptionParameter.ToString: \n" +
21                exceptionParameter);
22             Console.WriteLine("\nexceptionParameter.Message: \n" +
23                exceptionParameter.Message);
24             Console.WriteLine("\nexceptionParameter.StackTrace: \n" +
25                exceptionParameter.StackTrace);
26             Console.WriteLine("\nexceptionParameter.InnerException: \n" +
27                exceptionParameter.InnerException);
28          }
29       }
30
```

Fig. 13.5 | Stack unwinding and `Exception` class properties. (Part 2 of 6.)

```
31      // calls Method2
32      static void Method1()
33      {
34         Method2();
35      }
36
37      // calls Method3
38      static void Method2()
39      {
40         Method3();
41      }
42
```

Fig. 13.5 | Stack unwinding and `Exception` class properties. (Part 3 of 6.)

```
43      // throws an Exception containing an InnerException
44      static void Method3()
45      {
46         // attempt to convert string to int
47         try
48         {
49            int.Parse("Not an integer");
50         }
51         catch (FormatException formatExceptionParameter)
52         {
53            // wrap FormatException in new Exception
54            throw new Exception("Exception occurred in Method3",
55               formatExceptionParameter);
56         }
57      }
58   }
```

Fig. 13.5 | Stack unwinding and `Exception` class properties. (Part 4 of 6.)

```
exceptionParameter.ToString:
System.Exception: Exception occurred in Method3 --->
   System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
      NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style,
      NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at Properties.Method3() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 49
   --- End of inner exception stack trace ---
   at Properties.Method3() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 54
   at Properties.Method2() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 40
   at Properties.Method1() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 34
   at Properties.Main() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 14

exceptionParameter.Message:
Exception occurred in Method3
```

Fig. 13.5 | Stack unwinding and `Exception` class properties. (Part 5 of 6.)

```
exceptionParameter.StackTrace:
   at Properties.Method3() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 54
   at Properties.Method2() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 40
   at Properties.Method1() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 34
   at Properties.Main() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 14

exceptionParameter.InnerException:
System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
       NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style,
      NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at Properties.Method3() in C:\Users\PaulDeitel\Documents\examples\
      ch13\Fig13_05\Properties\Properties\Properties.cs:line 49
```

Fig. 13.5 | Stack unwinding and `Exception` class properties. (Part 6 of 6.)

13.7.4 Throwing an Exception with an `InnerException`

- The Exception class's two-argument constructor receives
  - the custom error message and the
  - InnerException to provide as additional information

13.7.5 Displaying Information About the `Exception`

- `Exception` method `ToString` returns a string that begins with the name of the exception class followed by the `Message` property value.
- The next items present the stack trace of the `InnerException` object.
- The remainder of the block of output shows the `StackTrace` for the exception.
- The `StackTrace` represents the state of the method-call stack at the throw point of the exception, rather than at the point where the exception eventually is caught.
- Each `StackTrace` line that begins with "at" represents a method on the call stack.
- These indicate the method in which the exception occurred, the file in which the method resides and the line number of the throw point in the file. The inner-exception information includes the *inner-exception stack trace*.

Error-Prevention Tip 13.5

When catching and rethrowing an exception, provide additional debugging information in the rethrown exception. To do so, create an object of an `Exception` *subclass containing more specific debugging information, then pass the original caught exception to the new exception object's constructor to initialize the* `InnerException` *property.*

13.8 User-Defined Exception Classes

- In some cases, you might create exception classes specific to the problems that occur in your programs.
- User-defined exception classes should derive directly or indirectly from class `Exception`.
- Exceptions should be documented so that other developers will know how to handle them.

13.8 User-Defined Exception Classes (Cont.)

- ***Class NegativeNumberException***
- Figures 13.6–13.7 demonstrate a user-defined exception class. User-defined exceptions should define three constructors:
  - a parameterless constructor
  - a constructor that receives a `string` argument (the error message)
  - a constructor that receives a `string` argument and an `Exception` argument (the error message and the inner exception object)

Good Programming Practice 13.1

Associating each type of malfunction with an appropriately named exception class improves program clarity.

Software Engineering Observation 13.4

Before creating a user-defined exception class, investigate the existing exceptions in the .NET Framework Class Library to determine whether an appropriate exception type already exists.

13.8 User-Defined Exception Classes (Cont.)

- `NegativeNumberException` (Fig. 13.6) represents exceptions that occur when a program performs an illegal operation on a negative number, such as attempting to calculate its square root.
- Class `SquareRootTest` (Fig. 13.7) demonstrates our user-defined exception class.

```
1  // Fig. 13.6: NegativeNumberException.cs
2  // NegativeNumberException represents exceptions caused by
3  // illegal operations performed on negative numbers.
4  using System;
5
6  public class NegativeNumberException : Exception
7  {
8     // default constructor
9     public NegativeNumberException()
10       : base("Illegal operation for a negative number")
11    {
12       // empty body
13    }
14
```

Fig. 13.6 | `NegativeNumberException` represents exceptions caused by illegal operations performed on negative numbers. (Part 1 of 2.)

```
15      // constructor for customizing error message
16      public NegativeNumberException(string messageValue)
17         : base(messageValue)
18      {
19         // empty body
20      }
21
22      // constructor for customizing the exception's error
23      // message and specifying the InnerException object
24      public NegativeNumberException(string messageValue, Exception inner)
25         : base(messageValue, inner)
26      {
27         // empty body
28      }
29   }
```

Fig. 13.6 | `NegativeNumberException` represents exceptions caused by illegal operations performed on negative numbers. (Part 2 of 2.)

```
1  // Fig. 13.7: SquareRootTest.cs
2  // Demonstrating a user-defined exception class.
3  using System;
4
5  class SquareRootTest
6  {
7     static void Main(string[] args)
8     {
9        var continueLoop = true;
10
11       do
12       {
```

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 1 of 4.)

```
13            // catch any NegativeNumberException thrown
14            try
15            {
16               Console.Write(
17                  "Enter a value to calculate the square root of: ");
18               double inputValue = double.Parse(Console.ReadLine());
19               double result = SquareRoot(inputValue);
20
21               Console.WriteLine(
22                  $"The square root of {inputValue} is {result:F6}\n");
23               continueLoop = false;
24            }
25            catch (FormatException formatException)
26            {
27               Console.WriteLine("\n" + formatException.Message);
28               Console.WriteLine("Please enter a double value.\n");
29            }
30            catch (NegativeNumberException negativeNumberException)
31            {
32               Console.WriteLine("\n" + negativeNumberException.Message);
33               Console.WriteLine("Please enter a non-negative value.\n");
34            }
35         } while (continueLoop);
36      }
```

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 2 of 4.)

```
37
38      // computes square root of parameter; throws
39      // NegativeNumberException if parameter is negative
40      public static double SquareRoot(double value)
41      {
42         // if negative operand, throw NegativeNumberException
43         if (value < 0)
44         {
45            throw new NegativeNumberException(
46               "Square root of negative number not permitted");
47         }
48         else
49         {
50            return Math.Sqrt(value); // compute square root
51         }
52      }
53   }
```

```
Enter a value to calculate the square root of: 30
The square root of 30 is 5.477226
```

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 3 of 4.)

```
Enter a value to calculate the square root of: hello

Input string was not in a correct format.
Please enter a double value.

Enter a value to calculate the square root of: 25
The square root of 25 is 5.000000
```

```
Enter a value to calculate the square root of: -2

Square root of negative number not permitted
Please enter a non-negative value.

Enter a value to calculate the square root of: 2
The square root of 2 is 1.414214
```

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 4 of 4.)

13.9 Checking for null References; Introducing C# 6's ?. Operator

- Checking whether a reference is `null` before using it to call a method or access a property avoids `NullReferenceExceptions`.

Error-Prevention Tip 13.6

Always ensure that a reference is not `null` *before using it to call a method or access a property of an object.*

13.9.1 Null-Conditional Operator (?.)

- C# 6's new **null-conditional operator (?.)** provides an elegant way to check for `null`.
- The following statement calls `Dispose` *only* if `exampleObject` is not null
  - `exampleObject?.Dispose();`
- This eliminates the need to wrap the statement in and `if` statement that checks `exampleObject` to see if it's `null`

13.9.2 Revisiting Operators `is` and `as`

- Downcasting with operator `is` can cause `InvalidCastExceptions`.
- You can avoid the `InvalidCastException` by using the `as` operator:
  - `var employee = currentEmployee as BasePlusCommissionEmployee;`
  - If `currentEmployee` *is a* `BasePlusCommissionEmployee`, employee is assigned the `BasePlusCommissionEmployee`; otherwise, it's assigned `null`.
  - `employee` could be `null`, so you must check before using it.
  - For example, to give the `BasePlusCommissionEmployee` a 10% raise, we could use the following statement, which accesses and modifies the `BaseSalary` property *only* if employee is not `null`
    - `employee?.BaseSalary *= 1.10M;`

13.9.3 Nullable Types

- Suppose you'd like to capture the value of `employee?.BaseSalary`
  - `decimal salary = employee?.BaseSalary;`
- This statement actually results in a compilation error indicating that you cannot implicitly convert type `decimal?` to type `decimal`.
- Normally a value-type variable cannot be assigned `null`.
- Because the `employee` reference might be `null`, the expression returns a **nullable type**—a value type that also can be `null`.
- You specify a nullable type by following a value type's name with a question mark (`?`)—so `decimal?` represents a nullable `decimal`.
  - `decimal? salary = employee?.BaseSalary;`
  - indicates that `salary` either will be `null` or the `employee`'s `BaseSalary`.

13.9.3 Nullable Types

- Nullable types have the following capabilities:
  - The `GetValueOrDefault` method checks whether a nullable-type variable contains a value. If so, the method returns that value; otherwise, it returns the value type's default value.
  - An overload of this method receives one argument that enables you to specify a custom default value.
  - The `HasValue` property returns `true` if a nullable-type variable contains a value; otherwise, it returns `false`.
  - The `Value` property returns the nullable-type variable's underlying value or throws an `InvalidOperationException` if the underlying value is `null`.
- Variables of nullable types also may be used as the left operand of the null-conditional operator (`?.`) or the null coalescing operator (`??`).

Error-Prevention Tip 13.7

Before using a nullable-type variable's `Value` *property, use the* `HasValue` *property to check whether the variable has a value. If the nullable-type variable is* `null`*, accessing* `Value` *results in an* `InvalidOperationException`.

13.9.4 Null Coalescing Operator (??)

- C# also offers the **null coalescing operator (??)** for working with values that can be `null`.
- The operator has two operands. If the left operand is not `null`, the entire `??` expression evaluates to the left operand's value; otherwise, it evaluates to the right operand's value.
- `decimal salary = employee?.BaseSalary ?? 0M;`
  - if `employee` is not `null`, `salary` is assigned the `employee`'s `BaseSalary`; otherwise, `salary` is assigned `0M`.
  - The preceding statement is equivalent to
    - `decimal salary = (employee?.BaseSalary).GetValueOrDefault();`

13.9.4 Null Coalescing Operator (??)

- More elegant and more compact than writing the following equivalent code, which must explicitly test for `null`:
  - ```
    decimal salary = 0M;

    if (employee != null)
    {
       salary = employee.BaseSalary
    }
    ```
    ```

13.10 Exception Filters and the C# 6 when Clause

- Prior to C# 6, you could catch an exception based only on its type.
- C# 6 introduces **exception filters** that enable you to catch an exception based on a `catch`'s exeption type and a condition that's specified with a **`when` clause**, as in
  - `catch(`*`ExceptionType name`*`) when(`*`condition`*`)`
- You also can specify an exception filter for a general `catch` clause that does not provide an exception type. This allows you to catch an exception based only on a condition, as in
  - `catch when(`*`condition`*`)`
- The exception is caught only if the `when` clause's condition is `true`
- A typical use of an exception filter is to determine whether a property of an exception object has a specific value.

Common Programming Error 13.4

Following a `try` *block with multiple* `catch` *clauses for the same type results in a compilation error, unless they provide different* `when` *clauses. If there are multiple such* `catch`*es and one does not have a* `when` *clause, it must appear last; otherwise, a compilation error occurs.*