CHAPTER 11

# Exception Handling

In this chapter you will:

- Learn about exceptions, the `Exception` class, and generating `SystemExceptions`
- Compare traditional and object-oriented error-handling methods
- Use the `Exception` class's `ToString()` method and `Message` property
- Catch multiple exceptions
- Examine the structure of the `TryParse()` methods
- Use the `finally` block
- Handle exceptions thrown from outside methods
- Trace `Exception` objects through the call stack
- Create your own `Exception` classes
- Rethrow `Exceptions`

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

While visiting Web sites, you have probably seen an unexpected and cryptic message that announces an error and then shuts down your browser immediately. Perhaps something similar has happened to you while using a piece of application software. Certainly, if you have worked your way through all of the programming exercises in this book, you have encountered such errors while running your own programs. A program that just stops is aggravating, especially when you lose data, and the error message seems to indicate that the program "knows" what is causing the problem. You might grumble, "If the program knows what is wrong, why doesn't it just fix the problem?" In this chapter, you will learn how to handle these unexpected error conditions so your programs can be more user-friendly than those that simply shut down in the face of errors.

## Understanding Exceptions

An **exception** is any error condition or unexpected behavior in an executing program. Exceptions are caused by errors in program logic or insufficient system resources. The programs you write can generate many types of potential exceptions, including when:

- Your program asks for user input, but the user enters invalid data.
- The program attempts to divide an integer by zero.
- You attempt to access an array with a subscript that is too large or too small.
- You calculate a value that is too large for the answer's variable type.

These errors are called exceptions because presumably they are not usual occurrences; they are "exceptional." The object-oriented techniques used to manage such errors make up the group of methods known as **exception handling**. If you do not handle an exception, the running program terminates abruptly.

Managing exceptions involves an oxymoron; you must expect the unexpected.

Errors you discover when compiling a program are not exceptions; they are compiler errors. Only execution-time (also called *runtime*) errors are called exceptions.

In C#, all exceptions are objects that are instances of the `Exception` class or one of its derived classes. An exception condition generates an object that encapsulates information about the error. Like all other classes in the C# programming language, the `Exception` class is a descendent of the `Object` class. The `Exception` class has several descendent classes of its own, many with unusual names such as `CodeDomSerializerException`, `SUDSParserException`, and `SoapException`. Others have names that are more easily understood, such as `IOException` (for input and output errors), `InvalidPrinterException` (for when a user requests an invalid printer), and `PathTooLongException` (used when the path to a file contains more characters than a system allows). C# has more than 100 defined `Exception` subclasses; Table 11-1 lists just a few to give you an idea of the wide variety of circumstances they cover.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

| Class | Description |
|---|---|
| `System.ArgumentException` | Thrown when one of the arguments provided to a method is not valid |
| `System.ArithmeticException` | Thrown for errors in an arithmetic, casting, or conversion operation |
| `System.ArrayTypeMismatchException` | Thrown when an attempt is made to store an element of the wrong type within an array |
| `System.Data.OperationAbortedException` | Thrown when an ongoing operation is aborted by the user |
| `System.Drawing.Printing.InvalidPrinterException` | Thrown when you try to access a printer using printer settings that are not valid |
| `System.FormatException` | Thrown when the format of an argument does not meet the parameter specifications of the invoked method |
| `System.IndexOutOfRangeException` | Thrown when an attempt is made to access an element of an array with an index that is outside the bounds of the array; this class cannot be inherited |
| `System.InvalidCastException` | Thrown for an invalid casting or explicit conversion |
| `System.InvalidOperationException` | Thrown when a method call is invalid for the object's current state |
| `System.IO.InvalidDataException` | Thrown when a data stream is in an invalid format |
| `System.IO.IOException` | Thrown when an I/O error occurs |
| `System.MemberAccessException` | Thrown when an attempt to access a class member fails |
| `System.NotImplementedException` | Thrown when a requested method or operation is not implemented |
| `System.NullReferenceException` | Thrown when there is an attempt to dereference a null object reference |
| `System.OperationCanceledException` | Thrown in a thread upon cancellation of an operation that the thread was executing |
| `System.OutOfMemoryException` | Thrown when there is not enough memory to continue the execution of a program |
| `System.RankException` | Thrown when an array with the wrong number of dimensions is passed to a method |
| `System.StackOverflowException` | Thrown when the execution stack overflows because it contains too many nested method calls; this class cannot be inherited |

**Table 11-1** Selected C# `Exception` subclasses

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Table 11-1 uses the term *thrown*, which is explained later in this chapter.



Most exceptions you will use derive from three classes:

- The predefined Common Language Runtime exception classes derived from `SystemException`
- The user-defined application exception classes you derive from `ApplicationException`
- The `Exception` class, which is the parent of `SystemException` and `ApplicationException`

Microsoft previously advised that you should create your own custom exceptions from the `ApplicationException` class. They have revised their thinking and now advise that you should use the `Exception` class as a base because in practice, they have not found the previous approach to be of significant value. For updates, visit *http://msdn.microsoft.com*.

## Purposely Generating a `SystemException`

You can deliberately generate a `SystemException` by forcing a program to contain an error. As an example, in every programming language, it is illegal to divide an integer by zero because the operation is mathematically undefined. Consider the simple `MilesPerGallon` program in Figure 11-1, which prompts a user for two values and divides the first by the second. If the user enters nonzero integers, the program runs correctly and without incident. However, if the user enters 0 when prompted to enter gallons, division by 0 takes place, and an error is generated. Figure 11-2 shows two executions of the program.

When you divide a floating-point value by 0 in C#, no exception occurs. Instead, C# assigns the special value `Infinity` to the result. Because `float`s and `double`s are imprecise, dividing them by 0.000000. . . to an infinite number of decimal places approaches infinity mathematically.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
using System;
using static System.Console;
class MilesPerGallon
{
   static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      Write("Enter miles driven ");
      milesDriven = Convert.ToInt32(ReadLine());
      Write("Enter gallons of gas purchased ");
      gallonsOfGas = Convert.ToInt32(ReadLine());
      mpg = milesDriven / gallonsOfGas;
      WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

**Figure 11-1** The `MilesPerGallon` program

```
C:\C#\Chapter.11>MilesPerGallon
Enter miles driven 300
Enter gallons of gas purchased 12
You got 25 miles per gallon

C:\C#\Chapter.11>MilesPerGallon
Enter miles driven 300
Enter gallons of gas purchased 0

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at MilesPerGallon.Main()

C:\C#\Chapter.11>
```

**Figure 11-2** Two executions of the `MilesPerGallon` program

When the user enters a 0 for gallons in the MilesPerGallon program, a dialog box reports that the application has stopped working and needs to close. Because this error is intentional for demonstration purposes, you can ignore the message by clicking the Close program button in the lower-right corner of the dialog box.

In the first execution of the MilesPerGallon program in Figure 11-2, the user entered two usable integers, and division was carried out successfully. However, in the second execution, the user entered 0 for gallons of gas, and the error message indicates that an unhandled exception named `System.DivideByZeroException` was created. The message gives further information ("Attempted to divide by zero.") and shows the method where the exception occurred—in `MilesPerGallon.Main()`.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The `DivideByZeroException` object was generated automatically by C#. It is an instance of the `DivideByZeroException` class that has four ancestors. It is a child of the `ArithmeticException` class, which descends from the `SystemException` class. The `SystemException` class derives from the `Exception` class, which is a child of the `Object` class.



Just because an exception occurs and an `Exception` object is created, you don't necessarily have to deal with it. In the `MilesPerGallon` class, you simply let the offending program terminate; that's why the error message in Figure 11-2 indicates that the `Exception` is "Unhandled." However, the termination of the program is abrupt and unforgiving. When a program divides two numbers, or performs some other trivial task like playing a game, the user might be annoyed if the program ends abruptly. If the program is used for air-traffic control or to monitor a patient's vital statistics during surgery, an abrupt conclusion could be disastrous. With exception handling, a program can continue after dealing with a problem. This is especially important in mission-critical applications. The term **mission critical** refers to any process that is crucial to an organization. Object-oriented error-handling techniques provide more elegant solutions than simply shutting down.

Programs that can handle exceptions appropriately are said to be more fault tolerant and robust than those that do not. **Fault-tolerant** applications are designed so that they continue to operate, possibly at a reduced level, when some part of the system fails. **Robustness** represents the degree to which a system is resilient to stress, maintaining correct functioning.

Watch the video *System Exceptions*.

## TWO TRUTHS & A LIE

### Understanding Exceptions

1. An exception is any error condition or unexpected behavior in an executing program.
2. In C#, all exceptions are objects that are members of the `Exception` class or one of its derived classes.
3. In C#, when an exception occurs and an `Exception` object is created, you must employ a specific set of exception-handling techniques in order to manage the problem.

The false statement is #3. Just because an exception occurs and an `Exception` object is created, you don't necessarily have to deal with it.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

## You Do It

*Purposely Causing Exceptions*

C# generates `SystemExceptions` automatically under many circumstances. In the next steps, you purposely generate a `SystemException` by executing a program that provides multiple opportunities for `Exceptions`.

1. Start a new project named **ExceptionsOnPurpose**, and type the following program, which allows you to generate several different `Exceptions`.

```
using System;
using static System.Console;
class ExceptionsOnPurpose
{
   static void Main()
   {
      int answer;
      int result;
      int zero = 0;
      Write("Enter an integer >> ");
      answer = Convert.ToInt32(ReadLine());
      result = answer / zero;
      WriteLine("The answer is " + answer);
   }
}
```

2. You cannot divide by the unnamed constant 0 (by changing the dividing statement to `result = answer /0;`), nor can you change the variable `zero` to a named constant (by inserting `const` at the start of the `int zero` declaration). If you made either of these changes, the program would not compile because the compiler can determine that division by zero will occur. Using a variable `zero` instead of either an unnamed or named constant, however, the compiler "trusts" that a legitimate value will be provided for the variable before division occurs (although in this case, the trust is not warranted). Save the program, and compile it.

3. Execute the program several times using different input values, and observe the results. Two windows appear with each execution. The first reports that Windows is collecting more information about the problem. This window is soon replaced with the one shown in Figure 11-3, which repeats that

*(continues)*

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

*(continued)*

ExceptionsOnPurpose.exe has stopped working. You would never write a program that purposely divides by zero; you do so here to demonstrate C#'s exception-generating capabilities. If you were executing a professional application, you might be notified of a solution. Because you created this exception on purpose, just click the **Close program** button.

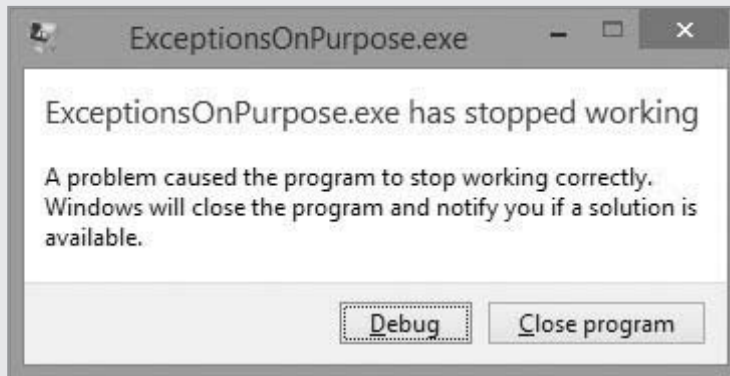


**Figure 11-3** The error report window generated by an unhandled exception

Figure 11-4 shows three executions of the program during which the user typed the following:

- **seven**—This generates a `System.FormatException`, which occurs when the program tries to convert the input value to an integer, because letters are not allowed in integers.
- **7.7**—This also generates a `System.FormatException` because the decimal point is not allowed in an integer.
- **7**—This does not generate a `System.FormatException` but instead causes a `System.DivideByZeroException` when the result is calculated.

*(continues)*

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

*(continued)*

```
C:\C#\Chapter.11>ExceptionsOnPurpose
Enter an integer >> seven

Unhandled Exception: System.FormatException: Input string was not in a correct f
ormat.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffe
r& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo in
fo)
   at System.Convert.ToInt32(String value)
   at ExceptionsOnPurpose.Main()

C:\C#\Chapter.11>ExceptionsOnPurpose
Enter an integer >> 7.7

Unhandled Exception: System.FormatException: Input string was not in a correct f
ormat.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffe
r& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo in
fo)
   at System.Convert.ToInt32(String value)
   at ExceptionsOnPurpose.Main()

C:\C#\Chapter.11>ExceptionsOnPurpose
Enter an integer >> 7

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at ExceptionsOnPurpose.Main()

C:\C#\Chapter.11>
```

**Figure 11-4** Error messages generated by successive executions of the `ExceptionsOnPurpose` program

# Comparing Traditional and Object-Oriented Error-Handling Methods

Programmers had to deal with error conditions long before object-oriented methods were conceived. For example, dividing by zero is an avoidable error for which programmers always have had to plan. If you simply check a variable's value with an `if` statement before attempting to divide it into another number, you can prevent the creation of an `Exception` object. For example, the following code uses a traditional `if` statement to check a variable to prevent division by zero.

```
if(gallonsOfGas != 0)
   mpg = milesDriven / gallonsOfGas;
else
   mpg = 0;
```

This code successfully prevents division by zero, but it does not really "handle an exception" because no `Exception` class object is created. The example code illustrates a perfectly legal and reasonable method of preventing division by zero, and it represents the most efficient method of handling the error if you think it will be a frequent problem. Because a program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

that contains this code does not have to instantiate an `Exception` object every time the user enters a 0 for the value of `gallonsOfGas`, the program saves time and computer memory. (Programmers say this program has little "overhead.") On the other hand, if you think dividing by zero will be infrequent—that is, the *exception* to the rule—then the decision will execute many times when it is not needed. In other words, if a user enters 0 for `gallonsOfGas` in only one case out of 1,000, then the `if` statement is executed unnecessarily 999 times. In that case, it is more efficient to eliminate the `if` test and instantiate an `Exception` object when needed.

The creators of C# define "infrequent" as an event that happens less than 30 percent of the time. That is, if you think an error will occur in less than 30 percent of all program executions, create an exception; if you think the error will occur more often, use traditional error checking. Of course, your boss or instructor might prefer a different percentage to use as a guideline.

Another advantage to using exception handling instead of decision making to control errors is that when a program's main logic is interrupted by a series of `if` statements and decisions that might detect errors, the program can become more complicated and difficult to understand. The object-oriented approach to error handling separates a program's main tasks from code that deals with rare exceptions.

Programmers use the phrase *sunny day case* to describe a scenario in which everything goes well and no errors occur.

## Understanding Object-Oriented Exception-Handling Methods

Object-oriented exception handling uses three unique terms: *try*, *throw*, and *catch*.

- You "try" a procedure that may not complete correctly.
- A method that detects an error condition "throws" an exception.
- The block of code that processes the error "catches" the exception.

When you write a block of code in which something can go wrong, you can place the code in a **`try` block**, which consists of the following elements:

- The keyword `try`
- A pair of curly braces containing statements that might cause exceptions

You must code at least one `catch` block or `finally` block immediately following a `try` block. (You will learn about `finally` blocks later in this chapter.) Each **`catch` block** can "catch" one type of exception—that is, an `Exception` object or an object of one of its child classes. Programmers sometimes refer to a `catch` block as a *catch clause*. You can create a `catch` block by typing the following elements:

- The keyword `catch`

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- Parentheses containing an `Exception` type and, optionally, a name for an instance of the `Exception` type
- A pair of curly braces containing statements that deal with the error condition

You also can create a `catch` block omitting the parentheses and `Exception` type. If you do, the `catch` block will execute when any `Exception` type is thrown. In other words `catch` and `catch(Exception)` are both correct ways to start a `catch` block to catch all `Exceptions` when you do not want to name the `Exception` object because you do not care about displaying its details.

Figure 11-5 shows the general format of a `try`...`catch` pair. The placeholder `XxxException` in the `catch` block represents the `Exception` class or any of its more specific subclasses. The placeholder `XxxException` is the data type of the `anExceptionInstance` object. A `catch` block looks a lot like a method named `catch()`, which takes an argument that is an instance of `XxxException`. However, it is not a method; you cannot call it directly. A method can accept multiple arguments, but a `catch` block can catch only one object, and, unlike a method, a `catch` block has no return type.

As `XxxException` implies, `Exception` classes typically are created using `Exception` as the second half of the name, as in `SystemException` and `ApplicationException`. The compiler does not require this naming convention, but the convention does make `Exception` descendents easier to identify.

```
try
{
   // Any number of statements;
   // some might cause an exception
}
catch(XxxException anExceptionInstance)
{
   // Do something about it
}
// Statements here execute whether there was an exception or not
```

**Figure 11-5** General form of a `try`...`catch` pair

You can place any number of statements in a `try` block, including those you know will never throw an exception. However, if any one of the statements you place within a `try` block does throw an exception, then the statements in the `catch` block that catches the exception will execute. If no exception occurs within the `try` block, then the `catch` block does not execute. Either way, the statements following the `catch` block execute normally.

For example, Figure 11-6 contains a program in which the statements that prompt for, accept, and use `gallonsOfGas` are encased in a `try` block. (The three variable declarations are made before the `try` block by convention. In this case, `mpg` must be declared outside the `try` block

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

to be recognized in the program's final output statement.) Figure 11-7 shows two executions of the program. In the first execution, a usable value is entered for `gallonsOfGas`, and the program operates normally, bypassing the `catch` block. In the second execution, however, the user enters 0 for `gallonsOfGas`. When division is attempted, an `Exception` object is automatically created and thrown. The `catch` block catches the `Exception`, where it becomes known as `e`. (The shaded arrow in Figure 11-6 extends from where the exception is created and thrown to where it is caught.) The statements in the `catch` block set `mpg` to 0 and display a message. Whether the `catch` block executes or not, the final `WriteLine()` statement that follows the `catch` block's closing curly brace executes.



```
using System;
using static System.Console;
class MilesPerGallon2
{
   static void Main( )
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      try
      {
         Write("Enter miles driven ");
         milesDriven = Convert.ToInt32(ReadLine( ));
         Write("Enter gallons of gas purchased ");
         gallonsOfGas = Convert.ToInt32(ReadLine( ));
         mpg = milesDriven / gallonsOfGas;
      }

      catch(Exception e)
      {
         mpg = 0;
         WriteLine("You attempted to divide by zero!");
      }
      WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

**Figure 11-6** The `MilesPerGallon2` program

```
C:\C#\Chapter.11>MilesPerGallon2
Enter miles driven 300
Enter gallons of gas purchased 12
You got 25 miles per gallon

C:\C#\Chapter.11>MilesPerGallon2
Enter miles driven 300
Enter gallons of gas purchased 0
You attempted to divide by zero!
You got 0 miles per gallon

C:\C#\Chapter.11>_
```

**Figure 11-7** Two executions of the `MilesPerGallon2` program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

When you compile the program in Figure 11-6, you receive a warning that `e` is declared but never used. You can declare a variable to hold the thrown `Exception` object, but you do not want to use it in this example, so you can safely ignore the warning. If you do not want to use the caught `Exception` object within a `catch` block, then you do not have to provide an instance name for it. For example, in Figure 11-6, the `catch` clause could begin as follows:

```
catch(Exception)
```

In later examples in this chapter, the `Exception` object will be used to provide information. In those cases, it is required to have an identifier.

In the `MilesPerGallon2` program, you could catch a more specific `DivideByZeroException` object instead of catching an `Exception` object. You will employ this technique later in the chapter. If you are working on a professional project, Microsoft recommends that you never use the general `Exception` class in a `catch` block because doing so can hide problems.

In the application in Figure 11-6, the source of the exception and the `catch` block reside in the same method. Later in this chapter, you will learn that exceptions and their corresponding `catch` blocks frequently reside in separate methods.

Watch the video *Throwing and Catching Exceptions*.

## TWO TRUTHS & A LIE

### Comparing Traditional and Object-Oriented Error-Handling Methods

1. Before object-oriented methods were conceived, programmers had no way of handling unexpected conditions.
2. Using object-oriented techniques, when you write a block of code in which something can go wrong, you can place the code in a `try` block.
3. If a `catch` block executes, then an `Exception` must have been thrown.

The false statement is #1. Programmers had to deal with error conditions long before object-oriented methods were conceived. They typically used decision-making techniques to check for errors.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

## Using the `Exception` Class's `ToString()` Method and `Message` Property

When the `MilesPerGallon2` program displays the error message ("You attempted to divide by zero!"), you actually cannot confirm from the message that division by zero was the source of the error. In reality, any `Exception` type generated within the `try` block of the program would be caught by the `catch` block in the method because the argument in the `catch` block is a general `Exception`, which is the parent class to all the `Exception` subtypes.

Instead of writing your own message, you can use the `ToString()` method that every `Exception` object inherits from the `Object` class. The `Exception` class overrides `ToString()` to provide a descriptive error message so a user can receive precise information about the nature of any `Exception` that is thrown. For example, Figure 11-8 shows a `MilesPerGallon3` program. The shaded areas show the only changes from the `MilesPerGallon2` program: the name of the class and the message that is displayed when an `Exception` is thrown. In this example, the `ToString()` method is used with the caught `Exception e`. Figure 11-9 shows an execution of the program in which the user enters 0 for `gallonsOfGas`. You can see in the output that the type of `Exception` caught is a `System.DivideByZeroException`.

You learned about overriding the `Object` class `ToString()` method in the chapter "Introduction to Inheritance."

```
using System;
using static System.Console;
class MilesPerGallon3
{
   static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      try
      {
         Write("Enter miles driven ");
         milesDriven = Convert.ToInt32(ReadLine());
         Write("Enter gallons of gas purchased ");
         gallonsOfGas = Convert.ToInt32(ReadLine());
         mpg = milesDriven / gallonsOfGas;
      }
```

**Figure 11-8** The `MilesPerGallon3` program *(continues)*

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

*(continued)*

```
      catch(Exception e)
      {
         mpg = 0;
         WriteLine(e.ToString());
      }
      WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

**Figure 11-8** The MilesPerGallon3 program

```
C:\C#\Chapter.11>MilesPerGallon3
Enter miles driven 300
Enter gallons of gas purchased 0
System.DivideByZeroException: Attempted to divide by zero.
   at MilesPerGallon3.Main()
You got 0 miles per gallon

C:\C#\Chapter.11>
```

**Figure 11-9** Execution of the MilesPerGallon3 program

The error message displayed in Figure 11-9 (*System.DivideByZeroException: Attempted to divide by zero.*) is the same message that appeared in Figure 11-2 when you provided no exception handling. Therefore, you can assume that the operating system most likely uses the same `ToString()` method you can use when displaying information about an `Exception` object. In the program in which you provided no exception handling, execution simply stopped; in this one, execution continues, and the final output statement is displayed whether the user's input was usable or not. Programmers would say this new version of the program ends more "elegantly."

The `Exception` class also contains a read-only property named `Message` that contains useful information about an `Exception` object. For example, the program in Figure 11-10 contains just two shaded changes from the `MilesPerGallon3` program: the class name and the use of the `Message` property in the statement that displays the error message in the `catch` block. The program produces the output shown in Figure 11-11. The value of `e.Message` is a `string` that is identical to the second part of the value returned by the `ToString()` method. You can guess that the `DivideByZeroException` class's `ToString()` method used in `MilesPerGallon3` constructs its string from two parts: the return value of the `getType()` method (that indicates the name of the class) and the return value from the `Message` property.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
using System;
using static System.Console;
class MilesPerGallon4
{
   static void Main()
   {
      int milesDriven;
      int gallonsOfGas;
      int mpg;
      try
      {
         Write("Enter miles driven ");
         milesDriven = Convert.ToInt32(ReadLine());
         Write("Enter gallons of gas purchased ");
         gallonsOfGas = Convert.ToInt32(ReadLine());
         mpg = milesDriven / gallonsOfGas;
      }
      catch(Exception e)
      {
         mpg = 0;
         WriteLine(e.Message);
      }
      WriteLine("You got {0} miles per gallon", mpg);
   }
}
```

**Figure 11-10** The `MilesPerGallon4` program

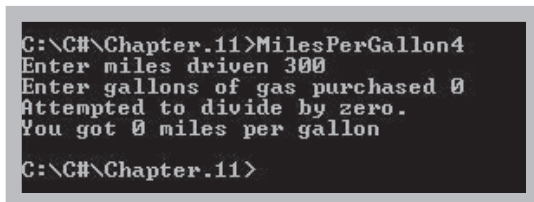


**Figure 11-11** Execution of the `MilesPerGallon4` program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

## TWO TRUTHS & A LIE

### Using the Exception Class's ToString() Method and Message Property

1. Any `Exception` type generated within a `try` block in a program will be caught by a `catch` block that has an `Exception` type argument.
2. The `Exception` class overrides the `Object` class `Description()` method to provide a descriptive error message so a user can receive precise information about the nature of any `Exception` object that is thrown.
3. The `Exception` class contains a property named `Message` that contains useful information about an `Exception`.

The false statement is #2. The `Exception` class overrides the `Object` class `ToString()` method to provide a descriptive error message so a user can receive precise information about the nature of any `Exception` object that is thrown.

## *You Do It*

*Handling Exceptions*

You can handle exceptions by placing them in a `try` block and then catching any exceptions that are thrown from it.

1. Open the **ExceptionsOnPurpose** program if it is not still open. Change the class name to **`ExceptionsOnPurpose2`**, and immediately save the project as **ExceptionsOnPurpose2**.
2. In the `Main()` method, after the three variable declarations, enclose the next three statements in a `try` block as follows:

```
try
{
   Write("Enter an integer >> ");
   answer = Convert.ToInt32(ReadLine());
   result = answer / zero;
}
```

3. Following the `try` block (but before the `WriteLine()` statement that displays `answer`), add a `catch` block that catches any thrown `Exception` object and displays its `Message` property:

*(continues)*

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

*(continued)*

```
catch(Exception e)
{
   WriteLine(e.Message);
}
```

4. Save the program, and compile it. You should receive a compiler error indicating that `answer` is an unassigned local variable. This error occurs at the last line of the program where `answer` is displayed. In the first version of this program, no such message appeared. However, now that the assignment to `answer` is within the `try` block, the compiler understands that an exception might be thrown, terminating the block before a valid value is assigned to `answer`. To eliminate this problem, initialize `answer` at its declaration:

```
int answer = 0;
```

5. Compile and execute the program again. Figure 11-12 shows three executions. The values typed by the user are the same as in Figure 11-4 in the previous "You Do It" section. However, the results are different in several significant ways:

- No error message window appears (as in Figure 11-3).
- The error messages displayed are cleaner and "friendlier" than the automatically generated versions in Figure 11-4.
- The program ends normally in each case, with the `answer` value displayed in a user-friendly manner.

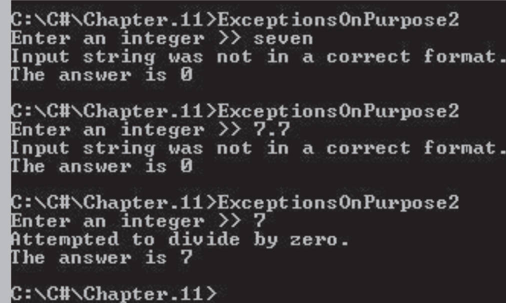


**Figure 11-12** Error messages generated by successive executions of the `ExceptionsOnPurpose2` program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

# Catching Multiple Exceptions

You can place as many statements as you need within a `try` block, and you can include as many `catch` blocks as you want to handle different `Exception` types. When a `try` block contains more than one statement, only the first error-generating statement throws an exception. As soon as the exception occurs, the logic transfers to the `catch` block, which leaves the rest of the statements in the `try` block unexecuted.

The set of `catch` blocks following a `try` block should cover all possible exceptions that the code in the `try` block might encounter. Assuming there is at least one `catch` block, if an exception is thrown without a matching `catch` block, the program will terminate abruptly. (It's okay if there are no `catch` blocks at all if there is a `finally` block, as you will see shortly.)

When an `Exception` object is thrown and multiple `catch` blocks are present, the `catch` blocks are examined in sequence until a match is found. The matching `catch` block then executes, and each remaining `catch` block is bypassed.

For example, consider the program in Figure 11-13. The `Main()` method in the `TwoErrors` class potentially throws two types of `Exceptions`—a `DivideByZeroException` and an `IndexOutOfRangeException`. (An `IndexOutOfRangeException` occurs when an array subscript is not within the allowed range. In the `TwoErrors` program, the array has only three elements, but 13 is used as a subscript.)

```
using System;
using static System.Console;
class TwoErrors
{
   static void Main( )
   {
      int num = 13, denom = 0, result;
      int[ ] array = {22, 33, 44};
      try
      {
         result = num / denom;
         result = array[num];
      }
      catch(DivideByZeroException error)
      {
         WriteLine("In first catch block: ");
         WriteLine(error.Message);
      }
      catch(IndexOutOfRangeException error)
      {
         WriteLine("In second catch block: ");
         WriteLine(error.Message);
      }
   }
}
```

**Figure 11-13** The `TwoErrors` program with two `catch` blocks

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The TwoErrors class declares three integers and an integer array with three elements. In the Main() method, the try block executes, and at the first statement within the try block, an exception occurs because the denom in the division problem is zero. The try block is abandoned, and control transfers to the first catch block. Integer division by zero causes a DivideByZeroException, and because the first catch block receives that type of Exception, the message *In first catch block* appears along with the Message value of the Exception object. In this example, the second try statement is never attempted, and the second catch block is skipped. Figure 11-14 shows the output.



```
C:\C#\Chapter.11>TwoErrors
In first catch block:
Attempted to divide by zero.

C:\C#\Chapter.11>
```

**Figure 11-14** Output of the TwoErrors program

If you reverse the two statements within the try block in the TwoErrors program, the process changes. If you use the try block in Figure 11-15, division by zero does not take place because the invalid array access throws an exception first.

```
using System;
using static System.Console;
class TwoErrors2
{
   static void Main( )
   {
      int num = 13, denom = 0, result;
      int[ ] array = {22, 33, 44};
      try
      {
         result = array[num];
         result = num / denom;
      }
      catch(DivideByZeroException error)
      {
         WriteLine("In first catch block: ");
         WriteLine(error.Message);
      }
      catch(IndexOutOfRangeException error)
      {
         WriteLine("In second catch block: ");
         WriteLine(error.Message);
      }
   }
}
```

**Figure 11-15** The TwoErrors2 program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The first statement within the `try` block in Figure 11-15 attempts to access element 13 of a three-element array, so the program throws an `IndexOutOfRangeException`. The `try` block is abandoned, and the first `catch` block is examined and found unsuitable because the `Exception` object is of the wrong type—it is not a `DivideByZeroException` object. The program logic proceeds to the second `catch` block, whose `IndexOutOfRangeException` argument type is a match for the thrown `Exception` type. The message *In second catch block* and the `Exception's Message` value are therefore displayed. Figure 11-16 shows the output.

```
C:\C#\Chapter.11>TwoErrors2
In second catch block:
Index was outside the bounds of the array.

C:\C#\Chapter.11>
```

**Figure 11-16** Output of the `TwoErrors2` program

Sometimes you want to execute the same code, no matter which `Exception` type occurs. For example, in each version of the `TwoErrors` program, each of the two `catch` blocks displays a unique message. Instead, you might want both the `DivideByZeroException catch` block and the `IndexOutOfRangeException catch` block to use the thrown `Exception's Message` field. Because `DivideByZeroException` and `IndexOutOfRangeException` both are subclasses of `Exception`, you can rewrite the `TwoErrors` class as shown in Figure 11-17 and include only one `Exception catch` block that catches any type of `Exception`.

```
using System;
using static System.Console;
class TwoErrors3
{
   static void Main()
   {
      int num = 13, denom = 0, result;
      int[] array = {22, 33, 44};
      try
      {
         result = array[num];
         result = num / denom;
      }
      catch(Exception error)
      {
         WriteLine(error.Message);
      }
   }
}
```

**Figure 11-17** The `TwoErrors3` class with one `catch` block

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

As an alternative to using a single `catch` block for multiple `Exception` types that require the same action, as in Figure 11-17, you could create the class to have separate `catch` blocks, each of which calls the same error-handling method.



The `catch` block in Figure 11-17 accepts a more generic `Exception` type than might be thrown by either statement in the `try` block, so the generic `catch` block can act as a "catch-all" block. That is, when either a division arithmetic error or an array error occurs, the thrown error is "promoted" to an `Exception` error in the `catch` block. Through inheritance, `DivideByZeroException`s and `IndexOutOfRangeException`s are `Exception`s.

As stated earlier, Microsoft recommends that a `catch` block should not handle general `Exceptions`. They say that if you cannot predict all possible causes of an exception, you should allow the application to terminate instead of handling the exception. Otherwise, it is easier for malicious code to exploit the resulting application state. However, many professional programmers disagree with this policy.

Although a block of code can throw any number of exceptions, many developers believe that it is poor style for a block or method to throw more than three or four types. If it does, one of the following conditions might be true:

- Perhaps the code block or method is trying to accomplish too many diverse tasks and should be broken up into smaller blocks or methods.
- Perhaps the `Exception` types thrown are too specific and should be generalized, as they are in the `TwoErrors3` program in Figure 11-17.

When you list multiple `catch` blocks following a `try` block, you must be careful that some `catch` blocks don't become unreachable. **Unreachable** blocks contain statements that can never execute under any circumstances because the program logic "can't get there." Programmers also call unreachable code **dead code**.

For example, if successive `catch` blocks catch an `Exception` followed by a `DivideByZeroException`, then even `DivideByZeroExceptions` will be caught by the `Exception catch`. The second, `DivideByZeroException catch` block is unreachable because the more general `Exception catch` block is in its way, and therefore the class will not compile.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

## TWO TRUTHS & A LIE

### Catching Multiple Exceptions

1. If you try more than one statement in a block, each error-generating statement throws an `Exception`.
2. You can write `catch` blocks that catch multiple `Exception` types.
3. When you list multiple `catch` blocks following a `try` block, you must be careful that some `catch` blocks don't become unreachable.

The false statement is #1. If you try more than one statement in a block, only the first error-generating statement throws an `Exception`.

## You Do It

*Catching Multiple* `Exception` *Types*

In this section, you instigate multiple outcomes by providing multiple `catch` blocks for code in a `try` block.

1. Open the **ExceptionsOnPurpose2** program if it is not still open. Change the class name to **`ExceptionsOnPurpose3`**, and immediately save the project as **ExceptionsOnPurpose3**.
2. Replace the existing generic `catch` block with two `catch` blocks. (The statement that displays the answer still follows these `catch` blocks.) The first catches any `FormatException` and displays a short message. The second catches a `DivideByZeroException` and displays a much longer message.

```
catch(FormatException e)
{
   WriteLine("You did not enter an integer");
}
catch(DivideByZeroException e)
{
   WriteLine("This is not your fault.");
   WriteLine("You entered the integer correctly.");
   WriteLine("The program divides by zero.");
}
```

*(continues)*

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

3. Save the program and compile it. When you execute the program and enter a value that is not an integer, the first `catch` block executes. When you enter an integer so that the program can proceed to the statement that divides by 0, the second `catch` block executes. Figure 11-18 shows two typical executions of the program.

```
C:\C#\Chapter.11>ExceptionsOnPurpose3
Enter an integer >> four
You did not enter an integer
The answer is 0

C:\C#\Chapter.11>ExceptionsOnPurpose3
Enter an integer >> 4
This is not your fault.
You entered the integer correctly.
The program divides by zero.
The answer is 4

C:\C#\Chapter.11>
```

**Figure 11-18** Error messages generated by successive executions of the `ExceptionsOnPurpose3` program

## Examining the Structure of the `TryParse()` Methods

In Chapter 8, you learned about the `TryParse()` methods that you can use to convert string data to another data type, such as `int`, `double`, or `bool`, without fear of generating an exception. A typical `TryParse()` statement might be as follows:

```
if(!int.TryParse(inputString, out number))
   number = DEFAULT_VALUE;
```

This statement uses the `int` version of the `TryParse()` method. The first argument is the string that you want to convert to a number, and the second argument is an `out` parameter that receives the result if the conversion is successful or 0 if it is not. Each of the `TryParse()` methods returns a Boolean value that indicates whether the conversion was successful. In the preceding example, the variable `number` is assigned a constant `DEFAULT_VALUE` if the method returns `false`—that is, if `inputString` is not an integer.

Suppose that no `TryParse()` method existed to convert a string to an integer safely. Using what you have learned so far in this chapter, you could now write your own method. The method handles the exception it might generate so that when a client uses the method, the client doesn't have to be concerned with exception handling. Figure 11-19 shows a usable method.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
public static bool TryParse(string inputString, out int number)
{
   bool wasSuccessful = true;
   try
   {
      number = Convert.ToInt32(inputString);
   }
   catch(FormatException e)
   {
      wasSuccessful = false;
      number = 0;
   }
   return wasSuccessful;
}
```

**Figure 11-19** A possible `TryParse()` method version

The `TryParse()` method in Figure 11-19 is `public` so that other classes can use it and `static` so that no object of its class needs to be instantiated. The method accepts a string and an `out` parameter that can hold the string's converted value. A `bool` variable named `wasSuccessful` is set to `true`, and then conversion is attempted. If the `Convert.ToInt32()` method works correctly, the variable `number` accepts the converted value. However, if a `FormatException` object is thrown, `wasSuccessful` is set to `false` and 0 is assigned to `number`. Whether an exception is thrown or not, the value of `wasSuccessful` is returned.

The creators of C# wrote code similar to that in Figure 11-19 when they created the `TryParse()` method for the `Integer` class. They may have included some differences—for example, they probably gave their variables different identifiers, and they might have reversed the positions of the statements in the `catch` block that set the Boolean value and the integer value, but the basic structure of the method must be similar. If no `TryParse()` method existed, you could write one for yourself, but the method is a convenience that already exists. As you continue to study programming, you might develop a method that would be convenient for others, and you might be rewarded in seeing your method used by other developers.

Your downloadable Student Data Files contain a program named TestTryParse.cs that can be used to test the `TryParse()` method.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

## TWO TRUTHS & A LIE

**Examining the Structure of the `TryParse()` Methods**

1. The `TryParse()` method uses an `out` parameter.
2. The `TryParse()` method employs exception handling.
3. The `TryParse()` method must be used within a `try` block.

The false statement is #3. The `TryParse()` method contains a `try` block, but you don't need to use the method in a `try` block because it handles its own exception.

# Using the `finally` Block

When you have actions to perform at the end of a `try...catch` sequence, you can use a **`finally` block**, which executes whether or not the `try` block identifies any exceptions. Typically, you use the `finally` block to perform clean-up tasks that must occur, regardless of whether any errors occurred or were caught. Figure 11-20 shows the format of a `try...catch` sequence that uses a `finally` block.

```
try
{
   // Statements that might cause an Exception
}
catch(SomeException anExceptionInstance)
{
   // What to do about it
}
finally
{
   // Statements here execute
   // whether an Exception occurred or not
}
```

**Figure 11-20** General form of a `try...catch` block with a `finally` block

At first glance, it seems as though the `finally` block serves no purpose. When a `try` block works without error, control passes to the statements that come after the `catch` block. Additionally, if the `try` code fails and throws an `Exception` object that is caught, then the `catch` block executes, and control again passes to any statements that are coded after the `catch` block. Therefore, it seems as though the statements after the `catch` block always

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

execute, so there is no need to place any statements within a special `finally` block. However, statements after the `catch` block might never execute for at least two reasons:

- An `Exception` for which you did not plan might occur, terminating the program.
- The `try` or `catch` block might contain a statement that terminates the application.

You can terminate an application with a statement such as `Environment.Exit(0);`. The `Environment.Exit()` method is part of the `System` namespace. It terminates a program and passes the argument (which can be any integer) to the operating system. You also might exit a `catch` block with a `break` statement or a `return` statement. You encountered `break` statements when you learned about the `switch` statement, and you learned about `return` statements when you studied methods.

The possibility exists that a `try` block might throw an `Exception` for which you did not provide a `catch`. After all, exceptions occur all the time without your handling them, as you saw in the first MilesPerGallon program at the beginning of this chapter. In the case of an unhandled exception, program execution stops immediately, sending the error to the operating system for handling and abandoning the current method. Likewise, if the `try` block contains an exit statement, execution stops immediately. When you include a `finally` block, you are assured that its enclosed statements will execute before the program is abandoned, even if the method concludes prematurely.

For example, the `finally` block is used frequently with file input and output to ensure that open files are closed. You will learn more about writing to and reading from data files in a later chapter. For now, however, consider the format shown in Figure 11-21, which represents part of the logic for a typical file-handling program. The `catch` block was written to catch an `IOException`, which is the type of `Exception` automatically generated if there is a problem opening a file, reading data from a file, or writing to a file.

```
try
{
   // Open the file
   // Read the file
   // Place the file data in an array
   // Calculate an average from the data
   // Display the average
}
catch(IOException e)
{
   // Issue an error message
   // Exit
}
finally
{
   // If the file is open, close it
}
```

**Figure 11-21** Format of code that tries reading a file and handles an exception

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The pseudocode in Figure 11-21 handles any file problems using the `catch` block. However, because the application uses an array (see the statement *Place the file data in an array*) and performs division (see the comment *Calculate an average...*), an uncaught exception could occur even though the file opened successfully. In such an event, you would want to close the file before proceeding. By using the `finally` block, you ensure that the file is closed, because the code in the `finally` block executes before the uncaught exception returns control to the operating system. The code in the `finally` block executes no matter which of the following outcomes of the `try` block occurs:

- The `try` ends normally.
- The `try` ends abnormally, and the `catch` executes.
- The `try` ends abnormally, and the `catch` does not execute. For example, an exception in the `try` block might cause the method to end prematurely—perhaps the array is not large enough to hold the data, or calculating the average results in division by 0. These exceptions do not allow the `try` block to finish, nor do they cause the `catch` block to execute because the `catch` block catches only `IOExceptions`.
- The `try` ends abnormally, the catch executes, but then the `catch` ends abnormally because an uncaught exception occurs.

If an application might throw several types of exceptions, you can try some code, catch the possible exception, try some more code, catch the possible exception, and so on. Usually, however, the superior approach is to try all the statements that might throw exceptions, then include all the needed `catch` blocks and an optional `finally` block. This is the approach shown in Figure 11-21, and it usually results in logic that is easier to follow.

You often can avoid using a `finally` block, but you would need repetitious code. For example, instead of using the `finally` block in the pseudocode in Figure 11-21, you could insert the statement *If the file is open, close it* as both the last statement in the `try` block and the second-to-last statement in the `catch` block, just before the program exits. However, writing code just once in a `finally` block is clearer and less prone to error.

Java, Visual Basic, and C++ provide `try` and `catch` blocks. Java and Visual Basic also provide a `finally` block, but C++ does not.

Many well-designed programs with one or more `try` blocks do not include `catch` blocks; instead, they contain only `try-finally` pairs. The `finally` block is used to release resources that other applications might be waiting for, such as database connections.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

**TWO TRUTHS & A LIE**

**Using the `finally` Block**

1. When a `finally` block follows a `try` block, it executes whether or not the `try` block identifies any `Exceptions`.
2. Typically, you use a `finally` block to perform clean-up tasks that must occur after an `Exception` has been thrown and caught.
3. Statements that follow a `try`…`catch` pair might never execute because an unplanned `Exception` might occur, or the `try` or `catch` block might contain a statement that quits the application.

The false statement is #2. Typically, you use a `finally` block to perform clean-up tasks that must occur, regardless of whether any errors occurred or were caught.

# Handling Exceptions Thrown from Outside Methods

An advantage of using object-oriented exception-handling techniques is the ability to deal with exceptions appropriately as you decide how to handle them. When methods from other classes throw exceptions, the methods don't have to catch them; instead, your calling program can catch them, and you can decide what to do.

For example, suppose that you prompt a user to enter a value at the console and then call the `Convert.ToInt32()` method to convert the input to an integer with a statement such as the following:

```
int answer = Convert.ToInt32(ReadLine());
```

If the user enters a noninteger value, a `FormatException` is generated from the `Convert.ToInt32()` method, but it is not caught and handled there. Instead, your program can catch the exception, and you can take appropriate action for your specific application. For example, you might want to force the user to reenter a value, but a different program might force the value to a default value or display an error message and terminate the program. This flexibility is an advantage when you need to create specific reactions to thrown exceptions. In many cases, you want a method to check for errors, but you do not want to require the method to handle an error if it finds one. Just as a police officer can deal with a speeding driver differently depending on circumstances, you can react to exceptions specifically for your current purposes.

When you design classes containing methods that have statements that might throw exceptions, you most frequently should create the methods so they throw the `Exception` object but do not handle it. Handling an exception usually should be left to the client—the program that uses your class—so the exception can be handled in an appropriate way for the application.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

For example, consider the very brief `PriceList` class in Figure 11-22. The class contains a list of prices and a single method that displays one price based on a parameter subscript value. Because the `DisplayPrice()` method uses an array, an `IndexOutOfRangeException` might be thrown. However, the `DisplayPrice()` method does not handle the potential exception.



```
class PriceList
{
   private static double[] price = {15.99, 27.88, 34.56, 45.89};
   public static void DisplayPrice(int item)
   {
      WriteLine("The price is " +
         price[item].ToString("C"));
   }
}
```

**Figure 11-22** The `PriceList` class

Figure 11-23 shows an application that uses the `DisplayPrice()` method. It calls the method in a `try` block and handles an `IndexOutOfRangeException` by displaying a price of $0. Figure 11-24 shows the output when a user enters an invalid item number.

```
using System;
using static System.Console;
class PriceListApplication1
{
   static void Main()
   {
      int item;
      try
      {
         Write("Enter an item number >> ");
         item = Convert.ToInt32(ReadLine());
         PriceList.DisplayPrice(item);
      }
      catch(IndexOutOfRangeException e)
      {
         WriteLine(e.Message + " The price is $0");
      }
   }
}
```

**Figure 11-23** The `PriceListApplication1` program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
C:\C#\Chapter.11>PriceListApplication1
Enter an item number >> 4
Index was outside the bounds of the array. The price is $0

C:\C#\Chapter.11>
```

**Figure 11-24** Output of the `PriceListApplication1` program when a user enters an invalid item number

Figure 11-25 shows a different application that uses the same `PriceList` class but handles the exception differently. In this case, the programmer wanted the user to keep responding until a correct entry was made. If an `IndexOutOfRangeException` is thrown from the `try` block, the `try` block is abandoned, `isGoodItem` is not set to `true`, and the `catch` block executes. Because `isGoodItem` remains `false`, the `while` loop continues to execute. Only when the first three statements in the `try` block are successful does the last statement execute, changing `isGoodItem` to `true` and ending the loop repetitions. Because the `DisplayPrice()` method in the `PriceList` class was written to throw an `Exception` but not handle it, the programmer of `PriceListApplication2` could handle the `Exception` in a totally different manner from the way it was handled in `PriceListApplication1`. Figure 11-26 shows a typical execution of this program.

```
using System;
using static System.Console;
class PriceListApplication2
{
   static void Main()
   {
      int item = 0;
      bool isGoodItem = false;
      while(!isGoodItem)
      {
         try
         {
            Write("Enter an item number >> "");
            item = Convert.ToInt32(ReadLine());
            PriceList.DisplayPrice(item);
            isGoodItem = true;
         }
```

**Figure 11-25** The `PriceListApplication2` program *(continues)*

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

*(continued)*

```
        catch(IndexOutOfRangeException e)
        {
           WriteLine("You must enter a number less " +
              "than 4");
           WriteLine("Please reenter item number ");
        }
     }
     WriteLine("Thank you");
  }
}
```

**Figure 11-25** The `PriceListApplication2` program

```
C:\C#\Chapter.11>PriceListApplication2
Enter an item number >> 12
You must enter a number less than 4
Please reenter item number
Enter an item number >> 4
You must enter a number less than 4
Please reenter item number
Enter an item number >> 3
The price is $45.89
Thank you

C:\C#\Chapter.11>
```

**Figure 11-26** Output of the `PriceListApplication2` program when a user enters an invalid item number several times

## TWO TRUTHS & A LIE

### Handling Exceptions Thrown from Outside Methods

1. When methods from other classes throw exceptions, the methods don't have to catch them; instead, your calling program can catch them, and you can decide what to do.
2. Often, you don't want a called method to handle its own exception; in many cases, you want the calling program to hande any exception in the way most appropriate for the application.
3. When you design classes containing methods that have statements that might throw exceptions, you almost always should make sure your class handles each exception appropriately.

The false statement is #3. When you design classes containing methods that have statements that might throw `Exception`s, you most frequently should create the methods so they throw the exception but do not handle it.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

# Tracing Exception Objects Through the Call Stack

When one method calls another, the computer's operating system must keep track of where the method call came from, and program control must return to the calling method when the called method is complete. For example, if MethodA() calls MethodB(), the operating system has to "remember" to return to MethodA() when MethodB() ends. Similarly, if MethodB() calls MethodC(), then while MethodC() is executing, the computer needs to "remember" that it will return to MethodB() and eventually to MethodA(). The memory location where the computer stores the list of locations to which the system must return is known as the **call stack**.

If a method throws an exception and does not catch it, then the exception is thrown to the next method "up" the call stack; in other words, it is thrown to the method that called the offending method. Consider this sequence of events:

- MethodA() calls MethodB().
- MethodB() calls MethodC().
- MethodC() throws an Exception.
- C# looks first for a catch block in MethodC().
- If none exists, then C# looks for the same thing in MethodB().
- If MethodB() does not have a catch block for the Exception, then C# looks to MethodA().
- If MethodA() doesn't catch the Exception, then the program terminates, and the operating system displays an error message.

This system of passing an exception through the chain of calling methods is called **propagating the exception**. It has great advantages because it allows your methods to handle exceptions more appropriately. However, a program that uses several classes makes it difficult for the programmer to locate the original source of an exception.

You already have used the Message property to obtain information about an Exception object. The StackTrace property is another useful Exception property. When you catch an Exception, you can display the value of StackTrace to provide a list of methods in the call stack so you can determine the location where the Exception object was created.

The StackTrace property can be a useful debugging tool. When your program stops abruptly, it is helpful to discover the method in which the exception occurred. Often, you do not want to display a StackTrace property in a finished program; the typical user has no interest in the cryptic messages that would appear. However, while you are developing a program, using StackTrace can help you diagnose problems.

## A Case Study: Using StackTrace

As an example of when StackTrace can be useful, consider the Tax class in Figure 11-27. Suppose that your company has created or purchased this class to make it easy to calculate tax rates on products sold. For simplicity, assume that only two tax rates are in effect—6 percent for

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

sales of $20 or less and 7 percent for sales over $20. The Tax class would be useful for anyone who wrote a program involving product sales, except for one flaw: in the shaded statement, the subscript is erroneously set to 2 instead of 1 for the higher tax rate. If this subscript is used with the taxRate array in the next statement, it will be out of bounds.



```
class Tax
{
   private static double[] taxRate = {0.06, 0.07};
   private static double CUTOFF = 20.00;
   public static double DetermineTaxRate(double price)
   {
      int subscript;
      double rate;
      if(price <= CUTOFF)
         subscript = 0;
      else
         subscript = 2;
      rate = taxRate[subscript];
      return rate;
   }
}
```

**Don't Do It**
This mistake is intentional.

**Figure 11-27** The Tax class

Assume that your company has also created a revised PriceList class, as shown in Figure 11-28. This class is similar to the one in Figure 11-22, except that it includes a tax calculation and calls the DetermineTaxRate() method in the shaded statement.

```
class PriceList
{
   private static double[] price = {15.99, 27.88, 34.56, 45.89};
   public static void DisplayPrice(int item)
   {
      double tax;
      double total;
      double pr;
      pr = price[item];
      tax = pr * Tax.DetermineTaxRate(pr);
      total = pr + tax;
      WriteLine("The total price is " + total.ToString("C"));
   }
}
```

**Figure 11-28** The PriceList class that includes a call to the Tax class method

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Suppose that you write the application shown in Figure 11-29. Your application is similar to the price list applications created earlier in this chapter, including a call to PriceList.DisplayPrice(). As in PriceListApplication1 and PriceListApplication2, your new program tries the data entry and display statement and then catches an exception. When you run the program using what you know to be a good item number, as in Figure 11-30, you are surprised to see the shaded "Error!" message you have coded in the catch block. In the earlier examples that used PriceList.DisplayPrice(), using an item number 1 would have resulted in a successful program execution.

```
using System;
using static System.Console;
class PriceListApplication3
{
   static void Main()
   {
      int item;
      try
      {
         Write("Enter an item number >> ");
         item = Convert.ToInt32(ReadLine());
         PriceList.DisplayPrice(item);
      }
      catch(Exception e)
      {
         WriteLine("Error!");
      }
   }
}
```

**Figure 11-29** The PriceListApplication3 class

When you compile the program in Figure 11-29, you receive a warning that variable e is declared but never used. If you omit the name e from the catch block, the application will execute identically without displaying a warning.

```
C:\C#\Chapter.11>PriceListApplication3
Enter an item number >> 1
Error!

C:\C#\Chapter.11>
```

**Figure 11-30** Execution of the PriceListApplication3 program when a user enters 1 for the item number

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

To attempt to discover what caused the *Error!* message, you can replace the statement that writes it as follows:

```
WriteLine(e.Message);
```

When the statement to display `e.Message` is added to the price list program, a warning that `e` is never used no longer appears when the program is compiled.

However, when you execute the program to display `e.Message`, you receive the output in Figure 11-31, indicating that the index is out of the bounds of the array. You are puzzled because you know 1 is a valid item number for the price array, and it should not be considered out of bounds.

```
C:\C#\Chapter.11>PriceListApplication3
Enter an item number >> 1
Index was outside the bounds of the array.

C:\C#\Chapter.11>
```

**Figure 11-31** Execution of the modified price list application in which `e.Message` is displayed in the `catch` block

Finally, you decide to replace the `WriteLine()` argument in the `catch` block statement with a `StackTrace` call, as follows:

```
WriteLine(e.StackTrace);
```

The output is shown in Figure 11-32. You can see from the list of methods that the error in your application came from `Tax.DetermineTaxRate()`, which was called by `PriceList.DisplayPrice()`, which was called by `Main()`. Notice that the last method executed is the first one dislayed in the list. Before using `StackTrace`, you might have not even considered that the `Tax` class could have been the source of the problem. If you work in a small organization, you can look at the code yourself and fix it. If you work in a larger organization or you purchased the class from an outside vendor, you can contact the programmer who created the class for assistance.

```
C:\C#\Chapter.11>PriceListApplication3
Enter an item number >> 1
   at Tax.DetermineTaxRate(Double price)
   at PriceList.DisplayPrice(Int32 item)
   at PriceListApplication3.Main()

C:\C#\Chapter.11>_
```

**Figure 11-32** Execution of the modified price list application in which `e.StackTrace` is displayed in the `catch` block

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The classes in this example were small to help you easily follow the discussion. However, a real-world application might have many more classes that contain many more methods, and so using `StackTrace` would become increasingly beneficial. In a large system of programs, you might find it useful to locate `StackTrace` calls strategically throughout the programs while testing them and then remove them or comment them out when the programs are complete.

Watch the video *Tracing Exceptions Through the Call Stack*.

**TWO TRUTHS & A LIE**

**Tracing `Exception` Objects Through the Call Stack**

1. The memory location where the computer stores the list of locations to which the system must return after a series of method calls is known as the stack trace.
2. If a method throws an exception and does not catch it, then the exception is thrown to the method that called the offending method.
3. When you catch an `Exception` object, you can output the value of `StackTrace` to display a list of methods in the call stack so you can determine the location of the error.

The false statement is #1. The memory location where the computer stores the list of locations to which the system must return after a series of method calls is known as the call stack.

## Creating Your Own `Exception` Classes

C# provides more than 100 categories of `Exceptions` that you can throw in your programs. However, C#'s creators could not predict every condition that might be an exception in the programs you write. For example, you might want to declare an `Exception` when your bank balance is negative or when an outside party attempts to access your e-mail account. Most organizations have specific rules for exceptional data, such as "an employee number must not exceed three digits" or "an hourly salary must not be less than the legal minimum wage." Of course, you can handle these potential error situations with `if` statements, but you also can create your own `Exception` subclasses.

To create your own `Exception` type from which you can instantiate throwable objects, you can extend the `ApplicationException` class, which is a subclass of `Exception`, or you can extend `Exception`. As you saw earlier in the chapter, Microsoft's advice on this matter has changed over time. Although you might see extensions of `ApplicationException` in classes written

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

by others, the current advice is simply to derive your own classes from `Exception`. Either approach will produce workable programs.

Figure 11-33 shows a `NegativeBalanceException` class that extends `Exception`. This class passes an appropriate `string` message to its parent's constructor. If you create an `Exception` object and display its `Message` property, you will see the message *Error in the application.* When the `NegativeBalanceException` constructor passes the string *Bank balance is negative.* to its parent's constructor, the `Message` property will hold this more descriptive message. The C# documentation recommends that you create all `Exception` subclass messages to be grammatically correct, complete sentences ending in a period.

```
class NegativeBalanceException : Exception
{
   private static string msg = "Bank balance is negative. ";
   public NegativeBalanceException() : base(msg)
   {
   }
}
```

**Figure 11-33** The `NegativeBalanceException` class

When you create a `BankAccount` class like the one shown in Figure 11-34, you can create the `Balance` property `set` accessor to throw a `NegativeBalanceException` when a client attempts to set the balance to be negative.

Instead of creating the `nbe` object in the `Balance` property in Figure 11-34, you could code the following statement, which creates and throws an anonymous `NegativeBalanceException` in a single step: `throw(new NegativeBalanceException());`

Figure 11-35 shows a program that attempts to set a `BankAccount` balance to a negative value in the shaded statement. When the `BankAccount` class's `Balance set` accessor throws the `NegativeBalanceException`, the `catch` block in the `TryBankAccount` program executes, displaying both the `NegativeBalanceException Message` and the value of `StackTrace`. Figure 11-36 shows the output.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
class BankAccount
{
   private double balance;
   public int AccountNum {get; set;}
   public double Balance
   {
      get
      {
         return balance;
      }
      set
      {
         if(value <0)
         {
            NegativeBalanceException nbe =
               new NegativeBalanceException();
            throw(nbe);
         }
         balance = value;
      }
   }
}
```

**Figure 11-34** The BankAccount class

```
using System;
using static System.Console;
class TryBankAccount
{
   static void Main()
   {
      BankAccount acct = new BankAccount();
      try
      {
         acct.AccountNum = 1234;
         acct.Balance = -1000;
      }
      catch(NegativeBalanceException e)
      {
         WriteLine(e.Message);
         WriteLine(e.StackTrace);
      }
   }
}
```

**Figure 11-35** The TryBankAccount program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
C:\C#\Chapter.11>TryBankAccount
Bank balance is negative.
   at BankAccount.set_Balance(Double value)
   at TryBankAccount.Main()

C:\C#\Chapter.11>
```

**Figure 11-36** Output of the `TryBankAccount` program

In the error message in Figure 11-36, notice that the `set` accessor for the `Balance` property is known internally as `set_Balance`. You can guess that the `set` accessor for the `AccountNum` property is known as `set_AccountNum`.

The `StackTrace` begins at the point where an `Exception` is thrown, not where it is instantiated. This consideration makes a difference when you create an `Exception` in a class and throw it from multiple methods.

`Exceptions` can be particularly useful when you throw them from constructors. Constructors do not have a return type, so they have no other way to send information back to a calling method.

In C#, you can't throw an object unless it is an `Exception` or a descendent of the `Exception` class. In other words, you cannot throw a `double` or a `BankAccount`. However, you can throw any type of `Exception` at any time, not just `Exceptions` of your own creation. For example, within any program you can code any of the following:

```
throw(new ApplicationException());
throw(new IndexOutOfRangeException());
throw(new Exception());
```

Of course, you should throw appropriate `Exception` types. In other words, you should not throw an `IndexOutOfRangeException` when you encounter division by 0 or data of an incorrect type; you should use it only when an index (subscript) is too high or too low. However, if a built-in `Exception` type is appropriate and suits your needs, you should use it. You should not create an excessive number of special `Exception` types for your classes, especially if the C# development environment already contains an `Exception` type that accurately describes the error. Extra `Exception` types add a level of complexity for other programmers who will use your classes. Nevertheless, when appropriate, creating a specialized `Exception` class is an elegant way for you to take care of error situations. `Exception`s allow you to separate your error code from the usual, nonexceptional sequence of events and allow for errors to be passed up the stack and traced.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

**TWO TRUTHS & A LIE**

**Creating Your Own Exception Classes**

1. To create your own Exception types whose objects you can throw, you can extend the ApplicationException class or the Exception class.
2. In C#, you can throw any object that you create if it is appropriate for the application.
3. You can throw any type of Exception at any time—both those that are already created as part of C# and those of your own creation.

The false statement is #2. In C#, you can't throw an object unless it is an Exception or a descendent of the Exception class.

## Rethrowing an Exception

When you write a method that catches an exception, your method does not have to handle it. Instead, you might choose to **rethrow the exception** to the method that called your method. Then you can let the calling method handle the problem.

Within a catch block, you can rethrow an unnamed exception that was caught by using the keyword throw with no object after it. For example, Figure 11-37 shows a class that contains four methods. In this program, the following sequence of events takes place:

1. The Main() method calls MethodA().
2. MethodA() calls MethodB().
3. MethodB() calls MethodC().
4. MethodC() throws an Exception.
5. When MethodB() catches the Exception, it does not handle the Exception; instead, it throws the Exception back to MethodA().
6. MethodA() catches the Exception but does not handle it either. Instead, MethodA() throws the Exception back to the Main() method.
7. The Exception is caught in the Main() method, where the message that was created in MethodC() is finally displayed.

Figure 11-38 shows the execution of the program.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
using System;
using static System.Console;
class ReThrowDemo
{
   static void Main()
   {
      try
      {
         WriteLine("Trying in Main() method");
         MethodA();
      }
      catch(Exception ae)
      {
      WriteLine("Caught in Main() method --\n {0}",
         ae.Message);
      }
      WriteLine("Main() method is done");
   }
   private static void MethodA()
   {
      try
      {
         WriteLine("Trying in method A");
         MethodB();
      }
      catch(Exception)
      {
         WriteLine("Caught in method A");
         throw;
      }
   }
   private static void MethodB()
   {
      try
      {
         WriteLine("Trying in method B");
         MethodC();
      }
      catch(Exception)
      {
         WriteLine("Caught in method B");
         throw;
      }
   }
   private static void MethodC()
   {
      WriteLine("In method C");
      throw(new Exception("This came from method C"));
   }
}
```

**Figure 11-37** The ReThrowDemo program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
C:\C#\Chapter.11>ReThrowDemo
Trying in Main() method
Trying in method A
Trying in method B
In method C
Caught in method B
Caught in method A
Caught in Main() method --
   This came from method C
Main() method is done

C:\C#\Chapter.11>
```

**Figure 11-38** Execution of the `ReThrowDemo` program

If you name the `Exception` argument in a `catch` block (for example, `catch(Exception e)`), then you should use that identifier when you use a `throw` statement in the block (for example, `throw e;`).

**TWO TRUTHS & A LIE**

**Rethrowing an `Exception`**

1. When you write a method that catches an exception, your method must handle it.
2. When a method catches an exception, you can rethrow it to a calling method.
3. Within a `catch` block, you can rethrow an unnamed caught `Exception` object by using the keyword `throw` with no object after it.

The false statement is #1. When you write a method that catches an `Exception`, your method does not have to handle the `Exception`.

## Chapter Summary

- An exception is any error condition or unexpected behavior in an executing program. In C#, all exceptions are objects that are instances of the `Exception` class or one of its derived classes. Most exceptions you will use are derived from three classes: `SystemException`, `ApplicationException`, and their parent `Exception`. You can purposely generate a `SystemException` by forcing a program to contain an error. Although you are not required to handle exceptions, you can use object-oriented techniques to provide elegant error-handling solutions.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- When you think an error will occur frequently, the most efficient approach is to handle the error in the traditional way, with `if` statements. If an error will occur infrequently, instantiating an `Exception` object when needed is more efficient. In object-oriented terminology, you "try" a procedure that may not complete correctly. A method that detects an error condition or exception "throws" an exception, and the block of code that processes the error "catches" the exception. You must include at least one `catch` block or `finally` block immediately following a `try` block.
- Every `Exception` object contains a `ToString()` method and a `Message` property that contains useful information about the `Exception`.
- You can place as many statements as you need within a `try` block, and you can catch as many different exceptions as you want. If you try more than one statement, only the first error-generating statement throws an exception. When multiple `catch` blocks are present, they are examined in sequence until a match is found for the `Exception` type that occurred. When you list multiple `catch` blocks after a `try` block, you must be careful about their order, or some `catch` blocks might become unreachable.
- The `TryParse()` methods that you can use to convert strings to other data types without causing exceptions use exception-handling techniques internally.
- A `finally` block performs actions at the end of a `try...catch` sequence, whether an exception was thrown or not.
- Exceptions do not have to be caught by the methods that generate them. Instead, a program that calls a method that throws an exception can catch and handle it. Often, this approach produces the best software design.
- If a method throws an exception and does not catch it, then the `Exception` object is thrown to the calling method. When you catch an `Exception` object, you can display the value of the `StackTrace` property, which provides a list of methods in the call stack, allowing you to determine the location where the error occurred.
- To create your own `Exception` classes from which you create throwable objects, you can extend the `ApplicationException` class or the `Exception` class. The current advice is to extend `Exception`.
- When you write a method that catches an exception, your method does not have to handle it. Instead, you might choose to rethrow the exception to the method that called your method and let that method handle it.

## Key Terms

An **exception** is any error condition or unexpected behavior in an executing program.

**Exception handling** is the set of object-oriented techniques used to manage unexpected errors in an executing program.

**Mission critical** describes any process that is crucial to an organization.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

**Fault-tolerant** applications are designed so that they continue to operate, possibly at a reduced level, when some part of the system fails.

**Robustness** represents the degree to which a system is resilient to stress, maintaining correct functioning even in the presence of errors.

A **`try` block** contains code that might create exceptions you want to handle.

A **`catch` block** can catch one type of `Exception`.

**Unreachable** blocks contain statements that can never execute under any circumstances because the program logic "can't get there."

**Dead code** is unreachable code.

A **`finally` block** can follow a `try` block; code within a `finally` block executes whether the `try` block identifies any `Exceptions` or not.

The **call stack** is the memory location where the computer stores the list of locations to which the system must return after method calls.

**Propagating an exception** is the act of transmitting it unchanged through the call stack.

To **rethrow an exception** is to throw a caught exception to a calling method instead of handling it within the method that generated it.

## Review Questions

1. Any error condition or unexpected behavior in an executing program is known as an ____________________.

    a. exception
    b. anomaly
    c. exclusion
    d. omission

2. Which of the following is *not* treated as a C# `Exception`?

    a. Your program asks the user to input a number, but the user enters a character.
    b. You attempt to execute a C# program, but the C# compiler has not been installed.
    c. You attempt to access an array with a subscript that is too large.
    d. You calculate a value that is too large for the answer's variable type.

3. Most exceptions you will use derive from three classes: ____________________.

    a. `Object`, `ObjectException`, and `ObjectApplicationException`
    b. `Exception`, `SystemException`, and `ApplicationException`
    c. `FormatException`, `ApplicationException`, and `IOException`
    d. `SystemException`, `IOException`, and `FormatException`

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

4. `Exception` objects can be ____________.
   a. generated automatically by C#
   b. created by a program
   c. both of these
   d. none of these



5. When a program creates an `Exception` object, you ____________.
   a. must handle it
   b. can handle it
   c. must not handle it
   d. none of these; programs cannot create `Exceptions`
6. If you do not use object-oriented techniques, ____________.
   a. there are no error situations
   b. you cannot manage error situations
   c. you can manage error situations but with great difficulty
   d. you can manage error situations
7. In object-oriented terminology, you ____________ a procedure that may not complete correctly.
   a. circumvent
   b. attempt
   c. catch
   d. try
8. In object-oriented terminology, a method that detects an error condition ____________ an exception.
   a. throws
   b. catches
   c. tries
   d. unearths
9. When you write a block of code in which something can go wrong, and you want to throw an exception if it does, you place the code in a ____________ block.
   a. `catch`
   b. `blind`
   c. `system`
   d. `try`
10. A `catch` block executes when its `try` block ____________.
    a. completes
    b. throws any `Exception` object
    c. throws an `Exception` of an acceptable type
    d. completes without throwing anything

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

11. Which of the following `catch` blocks will catch any `Exception` object?
    a. `catch(Any e) {}`
    b. `catch(Exception e) {}`
    c. `catch(e) {}`
    d. All of the above will catch any `Exception`.
12. Which of the following is valid within a `catch` block with the header `catch(Exception error)`?
    a. `WriteLine(error.ToString());`
    b. `WriteLine(error.Message);`
    c. `return(error.ToString());`
    d. two of these
13. You can place ____________________ statement(s) within a `try` block.
    a. zero
    b. one
    c. two
    d. any number of
14. How many `catch` blocks might follow a `try` block within the same method?
    a. only one
    b. any number as long as it is greater than zero
    c. any number as long as it is greater than one
    d. any number, including zero or one
15. Consider the following `try` block. If `x` is 15, what is the value of `a` when this code completes?

```
try
{
   a = 99;
   if(x > 10)
      throw(new Exception());
   a = 0;
   ++a;
}
```

    a. 0
    b. 1
    c. 99
    d. undefined

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

16. Consider the following `catch` blocks. The variable `b` has been initialized to 0. If a `DivideByZeroException` occurs in a `try` block just before this `catch` block, what is the value of `b` when this code completes?

```
catch(DivideByZeroException e)
{
   ++b;
}
catch(Exception e)
{
   ++b;
}
```

a. 0
b. 1
c. 2
d. 3

17. Consider the following `catch` blocks. The variable `c` has been initialized to 0. If an `IndexOutOfRangeException` occurs in a `try` block just before this `catch` block, what is the value of `c` when this code completes?

```
catch(IndexOutOfRangeException e)
{
   ++c;
}
catch(Exception e)
{
   ++c;
}
finally
{
   ++c;
}
```

a. 0
b. 1
c. 2
d. 3

18. If your program throws an `IndexOutOfRangeException`, and the only available `catch` block catches an `Exception`, ___________________.

a. an `IndexOutOfRangeException` `catch` block is generated automatically
b. the `Exception` `catch` block executes
c. the `Exception` `catch` block is bypassed
d. the `IndexOutOfRangeException` is thrown to the operating system

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

19. When you design your own classes that might cause exceptions, and other classes will use your classes as clients, you should usually create your methods to ____________________.
    a. neither throw nor handle exceptions
    b. throw exceptions but not handle them
    c. handle exceptions but not throw them
    d. both throw and handle exceptions

20. When you create an `Exception` subclass of your own, you should extend the ____________________ class.
    a. `SystemException`
    b. `PersonalException`
    c. `OverloadedException`
    d. `Exception`

# Exercises

## *Programming Exercises*

1. Write a program named **SubscriptExceptionTest** in which you declare an array of 20 `doubles` and store values in the array. Write a `try` block in which you place a loop that prompts the user for a subscript value and displays the value stored in the corresponding array position. Create a `catch` block that catches any `IndexOutOfRangeException` and displays an appropriate error message.

2. `ArgumentException` is an existing class that derives from `Exception`; you use it when one or more of a method's arguments do not fall within an expected range. Create an application for Merrydale Mortgage Company named **MortgageApplication** containing variables that can hold an applicant's name and credit score. Within the class, create a method that accepts a parameter for the credit score and returns `true` or `false`, indicating whether the applicant is eligible for a mortgage. If the score is not between 300 and 850, it is invalid, and the method should throw an `ArgumentException`. An application is accepted when the credit score is valid and at least 650. In the `Main()` method, continuously prompt the user for appplicant data, pass it to the method, and then display a message indicating whether the applicant is accepted, rejected, or has an invalid score.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

3. Create a program named **FindSquareRoot** that finds the square root of a user's input value. The `Math` class contains a static method named `Sqrt()` that accepts a `double` and returns the parameter's square root. If the user's entry cannot be converted to a `double`, display an appropriate message, and set the square root value to 0. Otherwise, test the input number's value. If it is negative, throw a `new ApplicationException` to which you pass the message "Number can't be negative." and again set `sqrt` to 0. If the input value is a `double` and not negative, pass it to the `Math.Sqrt()` method, and display the returned value.

4. a. Create a program named **OrderExceptionDemo** that attempts to create several valid and invalid `Order` objects. Immediately after each instantiation attempt, handle any thrown exceptions by displaying an error message. Create an `Order` class with three fields for an item number, quantity, and day ordered. The `Order` constructor requires values for each field. Upon construction, throw an `ArgumentException` if the item number is less than 100 or more than 999, or if the quantity ordered is less than 1 or more than 12, or the day ordered is not between 1 and 31, inclusive. Display the data if the instantiation is successful.

   b. Add `get` properties for each field in the `Order` class. Then, write an application named **OrderExceptionDemo2** that creates an array of five `Orders`. Prompt the user for values for each field; if the user enters improper or invalid data for any field, set item number, quantity, and day all to 0. (You will have to modify the `Order` constructor so that an object with all default values is legal.) At the end of the program, display all the records.

5. a. Create a program named **BookExceptionDemo** for the Peterman Publishing Company. Create a `BookException` class that is instantiated when a `Book's` price exceeds 10 cents per page and whose constructor requires three arguments for title, price, and number of pages. Create an error message that is passed to the `Exception` class constructor for the `Message` property when a `Book` does not meet the price-to-pages ratio. For example, an error message might be:

   *For Goodnight Moon, ratio is invalid.*

   *...Price is $12.99 for 25 pages.*

   b. Create a `Book` class that contains fields for title, author, price, and number of pages. Include properties for each field. Throw a `BookException` if a client program tries to construct a `Book` object for which the price is more than 10 cents per page. Create a program that creates at least four `Book` objects—some where the ratio is acceptable, and others where it is not. Catch any thrown exceptions, and display the `BookException Message`.

   c. Using the `Book` class, write an application named **BookExceptionDemo2** that creates an array of five `Book`s. Prompt the user for values for each `Book`. To handle any exceptions that are thrown because of improper or invalid data entered by the user, set the `Book`'s price to the maximum of 10 cents per page. At the end of the program, display all the entered, and possibly corrected, records.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

6. Create a GUI application named **ExceptionHandlingDemoGUI** that prompts a user to enter an integer. When the user clicks a button, try to convert the user's entry to an integer using the `Convert.ToInt32()` method, and display a message indicating whether the user was successful.



7. In Chapter 6, you created a game named `GuessAWord` in which a player guesses letters to try to replicate a hidden word. Now create a modified version of the program named **GuessAWordWithExceptionHandling** that throws and catches an exception when the user enters a guess that is not a letter of the alphabet. Create a `NonLetterException` class that descends from `Exception`. The `Message string` for your new class should indicate the nonletter character that caused the `Exception's` creation. When a `NonLetterException` is thrown and caught, the message should be displayed.

## *Debugging Exercises*

1. Each of the following files in the Chapter.11 folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, DebugEleven1.cs will become FixedDebugEleven1.cs.
   a. DebugEleven1.cs
   b. DebugEleven2.cs
   c. DebugEleven3.cs
   d. DebugEleven4.cs

## *Case Problems*

1. Modify the **GreenvilleRevenue** program created in the previous chapter so that it performs the following tasks:
   - The program prompts the user for the number of contestants in this year's competition; the number must be between 0 and 30. Use exception-handling techniques to ensure a valid value.
   - The program prompts the user for talent codes. Use exception-handling techniques to ensure a valid code.
   - After data entry is complete, the program prompts the user for codes so the user can view lists of appropriate contestants. Use exception-handling techniques for the code verification.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

2. Modify the **MarshallsRevenue** program created in the previous chapter so that it performs the following tasks:

   - Modify the program to use exception-handling techniques to ensure valid values for the month, the number of interior murals scheduled, and the number of exterior murals scheduled. In each case, the program continues to prompt the user until valid entries are made.
   - Modify the program to use exception-handling techniques to ensure valid mural codes for the scheduled jobs.
   - After data entry is complete, the program continuously prompts the user for codes and displays associated jobs. Use exception-handling techniques to verify valid codes.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.