CHAPTER 12

Exception Handling

In this chapter, you will:

- Learn about exceptions
- Try code and catch exceptions
- Throw and catch multiple exceptions
- Use the `finally` block
- Understand the advantages of exception handling
- Specify the exceptions that a method can throw
- Trace exceptions through the call stack
- Create your own `Exception` classes
- Use an assertion

Unless noted otherwise, all images are © 2014 Cengage Learning

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

Learning About Exceptions

An **exception** is an unexpected or error condition. The programs you write can generate many types of potential exceptions:

- A program might issue a command to read a file from a disk, but the file does not exist there.
- A program might attempt to write data to a disk, but the disk is full or unformatted.
- A program might ask for user input, but the user enters an invalid data type.
- A program might attempt to divide a value by 0.
- A program might try to access an array with a subscript that is too large or too small.

These errors are called exceptions because, presumably, they are not usual occurrences; they are "exceptional." **Exception handling** is the name for the object-oriented techniques that manage such errors. Unplanned exceptions that occur during a program's execution are also called **runtime exceptions**, in contrast with syntax errors that are discovered during program compilation.

Java includes two basic classes of errors: `Error` and `Exception`. Both of these classes descend from the `Throwable` class, as shown in Figure 12-1. Like all other classes in Java, `Error` and `Exception` originally descend from `Object`.

```
java.lang.Object
|
+--java.lang.Throwable
   |
   +--java.lang.Exception
   |  |
   |  +--java.io.IOException
   |  |
   |  +--java.lang.RuntimeException
   |  |  |
   |  |  +--java.lang.ArithmeticException
   |  |  |
   |  |  +-- java.lang.IndexOutOfBoundsException
   |  |  |  |
   |  |  |  +--java.lang.ArrayIndexOutOfBoundsException
   |  |  |
   |  |  +-- java.util.NoSuchElementException
   |  |  |   |
   |  |  |   +--java.util.InputMismatchException
   |  |  |
   |  |  +--Others..
   |  |
   |  +--Others..
```

Figure 12-1 The `Exception` and `Error` class inheritance hierarchy *(continues)*

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

```
   |  |
      +--java.lang.Error
      |
      +-- java.lang.VirtualMachineError
         |
         +--java.lang.OutOfMemoryError
         |
         +--java.lang.InternalError
         |
         +--Others...
```

Figure 12-1 The `Exception` and `Error` class inheritance hierarchy

The **`Error` class** represents more serious errors from which your program usually cannot recover. For example, there might be insufficient memory to execute a program. Usually, you do not use or implement `Error` objects in your programs. A program cannot recover from `Error` conditions on its own.

The **`Exception` class** comprises less serious errors representing unusual conditions that arise while a program is running and from which the program *can* recover. Some examples of `Exception` class errors include using an invalid array subscript or performing certain illegal arithmetic operations.

Java displays an `Exception` message when the program code could have prevented an error. For example, Figure 12-2 shows a class named `Division` that contains a single, small `main()` method. The method declares three integers, prompts the user for values for two of them, and calculates the value of the third integer by dividing the first two values.

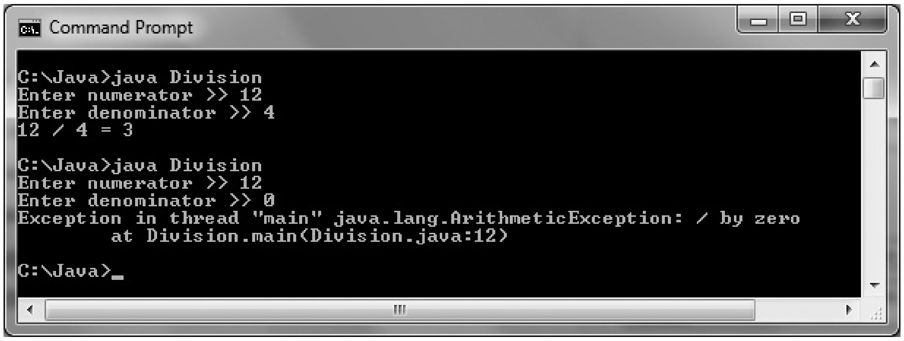
```java
import java.util.Scanner;
public class Division
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      System.out.print("Enter numerator >> ");
      numerator = input.nextInt();
      System.out.print("Enter denominator >> ");
      denominator = input.nextInt();
      result = numerator / denominator;
      System.out.println(numerator + " / " + denominator +
         " = " + result);
   }
}
```

Figure 12-2 The `Division` class

Not For Sale

Not For Sale

Figure 12-3 shows two typical executions of the `Division` program. In the first execution, the user enters two usable values and the program executes normally. In the second execution, the user enters 0 as the value for the denominator and an `Exception` message is displayed. (Java does not allow integer division by 0, but floating-point division by 0 is allowed—the result is displayed as `Infinity`.) In the second execution in Figure 12-3, most programmers would say that the program experienced a **crash**, meaning that it ended prematurely with an error. The term *crash* probably evolved from the hardware error that occurs when a read/write head abruptly comes into contact with a hard disk, but the term has evolved to include software errors that cause program failure.



```
C:\Java>java Division
Enter numerator >> 12
Enter denominator >> 4
12 / 4 = 3

C:\Java>java Division
Enter numerator >> 12
Enter denominator >> 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Division.main(Division.java:12)

C:\Java>_
```

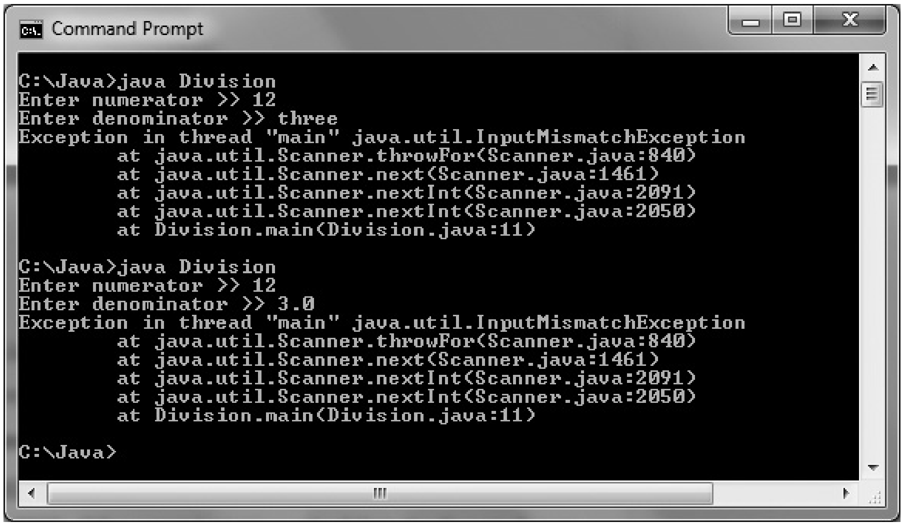
Figure 12-3 Two typical executions of the `Division` application

In Figure 12-3, the `Exception` is a `java.lang.ArithmeticException`. `ArithmeticException` is one of many subclasses of `Exception`. Java acknowledges more than 75 categories of `Exceptions` with unusual names such as `ActivationException`, `AlreadyBoundException`, `AWTException`, `CloneNotSupportedException`, `PropertyVetoException`, and `UnsupportedFlavorException`.

Besides the type of `Exception`, Figure 12-3 also shows some information about the error ("/ by zero"), the method that generated the error (`Division.main`), and the file and line number for the error (Division.java, line 12).

Figure 12-4 shows two more executions of the `Division` class. In each execution, the user has entered noninteger data for the denominator—first a string of characters, and second, a floating-point value. In each case, a different type of `Exception` occurs. You can see from both sets of error messages that the `Exception` is an `InputMismatchException`. The last line of the messages indicates that the problem occurred in line 11 of the `Division` program, and the second-to-last error message shows that the problem occurred within the call to `nextInt()`. Because the user did not enter an integer, the `nextInt()` method failed. The second-to-last message also shows that the error occurred in line 2050 of the `nextInt()` method, but clearly you do not want to alter the `nextInt()` method that resides in the `Scanner` class—you either want to rerun the program and enter an integer, or alter the program so that these errors cannot occur in subsequent executions.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.



```
C:\Java>java Division
Enter numerator >> 12
Enter denominator >> three
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextInt(Scanner.java:2091)
        at java.util.Scanner.nextInt(Scanner.java:2050)
        at Division.main(Division.java:11)

C:\Java>java Division
Enter numerator >> 12
Enter denominator >> 3.0
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextInt(Scanner.java:2091)
        at java.util.Scanner.nextInt(Scanner.java:2050)
        at Division.main(Division.java:11)

C:\Java>
```

Figure 12-4 Two executions of the `Division` application in which the user entered noninteger values

The list of error messages after each attempted execution in Figure 12-4 is called a **stack trace history list**, or more simply, a **stack trace**. (You might also hear the terms *stack backtrace* or *stack traceback*.) The list shows each method that was called as the program ran. You will learn more about tracing the stack later in this chapter.

Just because an exception occurs, you don't necessarily have to deal with it. In the `Division` class, you can simply let the offending program terminate as it did in Figure 12-4. However, the program termination is abrupt and unforgiving. When a program divides two numbers (or performs a less trivial task such as balancing a checkbook), the user might be annoyed if the program ends abruptly. However, if the program is used for a mission critical task such as air-traffic control or to monitor a patient's vital statistics during surgery, an abrupt conclusion could be disastrous. (The term **mission critical** refers to any process that is crucial to an organization.) Object-oriented error-handling techniques provide more elegant and safer solutions for errors.

Of course, you can write programs without using exception-handling techniques—you have already written many such programs as you have worked through this book. Programmers had to deal with error conditions long before object-oriented methods were conceived. Probably the most common error-handling solution has been to use a decision to avoid an error. For example, you can change the `main()` method of the `Division` class to avoid dividing by 0 by adding the decision shown in the shaded portion of Figure 12-5:

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

```
import java.util.Scanner;
public class Division2
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      System.out.print("Enter numerator >> ");
      numerator = input.nextInt();
      System.out.print("Enter denominator >> ");
      denominator = input.nextInt();
      if(denominator == 0)
        System.out.println("Cannot divide by 0");
      else
      {
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
   }
}
```

Figure 12-5 The `Division2` application using a traditional error-handling technique

The application in Figure 12-5 displays a message to the user when 0 is entered for a denominator value, but it is not able to recover when noninteger data such as a string or floating-point value is entered. Object-oriented exception handling provides a more elegant solution for handling error conditions.

Programs that can handle exceptions appropriately are said to be more fault tolerant and robust. **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 if you choose never to use exception-handling techniques in your own programs, you must understand them because built-in Java methods will throw exceptions to your programs.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

TWO TRUTHS & A LIE

Learning About Exceptions

1. Exception handling is the name for the object-oriented techniques used to manage runtime errors.
2. The `Error` class represents serious errors from which your program usually cannot recover; the `Exception` class comprises less serious errors representing unusual conditions that occur while a program is running and from which the program *can* recover.
3. When an exception occurs, your program must handle it using object-oriented exception-handling techniques.

The false statement is #3. Just because an exception occurs, you don't necessarily have to deal with it. You have already written many programs that do not handle exceptions that arise.

Trying Code and Catching Exceptions

In object-oriented terminology, you "try" a procedure that might cause an error. A method that detects an error condition "throws an exception," and the block of code that processes the error "catches the exception."

When you create a segment of code in which something might go wrong, you place the code in a **`try` block**, which is a block of code you attempt to execute while acknowledging that an exception might occur. A `try` block consists of the following elements:

- The keyword `try`
- An opening curly brace
- Executable statements, including some that might cause exceptions
- A closing curly brace

To handle a thrown exception, you can code one or more `catch` blocks immediately following a `try` block. A **`catch` block** is a segment of code that can handle an exception that might be thrown by the `try` block that precedes it. The exception might be one that is thrown automatically, or you might explicitly write a `throw` statement. A **`throw` statement** is one that sends an `Exception` object out of a block or a method so that it can be handled elsewhere. A thrown `Exception` can be caught by a `catch` block. Each `catch` block can "catch" one type

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

of exception—that is, one object that is an object of type `Exception` or one of its child classes. You create a `catch` block by typing the following elements:

- The keyword `catch`
- An opening parenthesis
- An `Exception` type
- A name for an instance of the `Exception` type
- A closing parenthesis
- An opening curly brace
- The statements that take the action you want to use to handle the error condition
- A closing curly brace

Figure 12-6 shows the general format of a method that includes a shaded `try`...`catch` pair. A `catch` block looks a lot like a method named `catch()` that takes an argument that is some type of `Exception`. However, it is not a method; it has no return type, and you can't call it directly. Some programmers refer to a `catch` block as a *catch clause*.

```
returnType methodName(optional arguments)
{
   // optional statements prior to code that is tried
   try
   {
      // statement or statements that might generate an exception
   }
   catch(Exception someException)
   {
      // actions to take if exception occurs
   }
   // optional statements that occur after try,
   // whether or not catch block executes
}
```

Figure 12-6 Format of `try`...`catch` pair

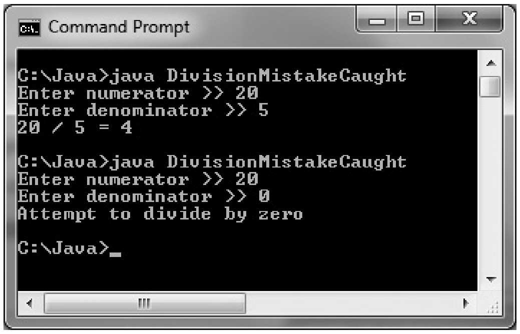
In Figure 12-6, `someException` represents an object of the `Exception` class or any of its subclasses. If an exception occurs during the execution of the `try` block, the statements in the `catch` block execute. If no exception occurs within the `try` block, the `catch` block does not execute. Either way, the statements following the `catch` block execute normally.

Figure 12-7 shows an application named `DivisionMistakeCaught` that improves on the `Division` class. The `main()` method in the class contains a `try` block with code that attempts division. When illegal integer division is attempted, an `ArithmeticException` is automatically created and the `catch` block executes. Figure 12-8 shows two typical executions, one with a generated `Exception` and one without.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

```
import java.util.Scanner;
public class DivisionMistakeCaught
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      System.out.print("Enter numerator >> ");
      numerator = input.nextInt();
      System.out.print("Enter denominator >> ");
      denominator = input.nextInt();
      try
      {
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(ArithmeticException mistake)
      {
         System.out.println("Attempt to divide by zero");
      }
   }
}
```

Figure 12-7 The `DivisionMistakeCaught` application



```
C:\Java>java DivisionMistakeCaught
Enter numerator >> 20
Enter denominator >> 5
20 / 5 = 4

C:\Java>java DivisionMistakeCaught
Enter numerator >> 20
Enter denominator >> 0
Attempt to divide by zero

C:\Java>_
```

Figure 12-8 Two executions of the `DivisionMistakeCaught` application

In the application in Figure 12-7, the `throw` and `catch` operations reside in the same method. Later in this chapter, you will learn that `throws` and their corresponding `catch` blocks frequently reside in separate methods.

If you want to send error messages to a location other than “normal” output, you can use `System.err` instead of `System.out`. For example, if an application writes a report to a specific disk file, you might want errors to write to a different location—perhaps to a different disk file or to the screen.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

Although the DivisionMistakeCaught application displays the error message ("Attempt to divide by zero"), you cannot be sure that division by 0 was the source of the error. In reality, *any* ArithmeticException generated within the try block in the program would be caught by the catch block in the method. Instead of writing your own message, you can use the getMessage() method that ArithmeticException inherits from the Throwable class. To retrieve Java's message about any ThrowableException named someException, you code someException.getMessage().

As an example of another condition that could generate an ArithmeticException, if you create an object using Java's BigDecimal class and then perform a division that results in a nonterminating decimal division such as 1/3 but specify that an exact result is needed, an ArithmeticException is thrown. As another example, you could create your own class containing a method that creates a new instance of the ArithmeticException class and throws it under any conditions you specify.

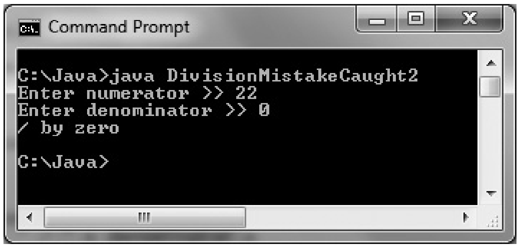
For example, Figure 12-9 shows a DivisionMistakeCaught2 class that uses the getMessage() method (see the shaded statement) to generate the message that "comes with" the caught ArithmeticException argument to the catch block. Figure 12-10 shows the output; the message is "/ by zero".

```
import java.util.Scanner;
public class DivisionMistakeCaught2
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      System.out.print("Enter numerator >> ");
      numerator = input.nextInt();
      System.out.print("Enter denominator >> ");
      denominator = input.nextInt();
      try
      {
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(ArithmeticException mistake)
      {
         System.out.println(mistake.getMessage());
      }
   }
}
```

Figure 12-9 The DivisionMistakeCaught2 application

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.



```
C:\Java>java DivisionMistakeCaught2
Enter numerator >> 22
Enter denominator >> 0
/ by zero

C:\Java>
```

Figure 12-10 Output of the `DivisionMistakeCaught2` application

It should be no surprise that the automatically generated error message in Figure 12-10 is "/ by zero"; you saw the same message in Figure 12-3 when the programmer provided no exception handling, the exception was automatically thrown, and its message was automatically supplied.

Of course, you might want to do more in a `catch` block than display an error message; after all, Java did that for you without requiring you to write the code to catch any exceptions. You also might want to add code to correct the error; for example, such code could force the arithmetic to divide by 1 rather than by 0. Figure 12-11 shows `try`…`catch` code in which the `catch` block computes the result by dividing by 1 instead of by the `denominator` value. After the `catch` block, the application could continue with a guarantee that `result` holds a valid value—either the division worked in the `try` block and the `catch` block did not execute, or the `catch` block remedied the error.

```
try
{
    result = numerator / denominator;
}
catch(ArithmeticException mistake)
{
    result = numerator / 1;
}
// program continues here; result is guaranteed to have a valid value
```

Figure 12-11 A `try`…`catch` block in which the `catch` block corrects the error

In the code in Figure 12-11, you can achieve the same result in the `catch` block by coding `result = numerator;` instead of `result = numerator / 1;`. Explicitly dividing by 1 simply makes the code's intention clearer, but it does require a small amount of time to execute the instruction. As an alternative, you could make the program more efficient by omitting the division by 1 and adding clarity with a comment.

Not For Sale

Not For Sale

Using a `try` Block to Make Programs "Foolproof"

One of the most common uses for a `try` block is to circumvent user data entry errors. When testing your own programs throughout this book, you might have entered the wrong data type accidentally in response to a prompt. For example, if the user enters a character or floating-point number in response to a `nextInt()` method call, the program crashes. Using a `try` block can allow you to handle potential data conversion exceptions caused by careless users. You can place conversion attempts, such as calling `nextInt()` or `nextDouble()`, in a `try` block and then handle any generated errors.

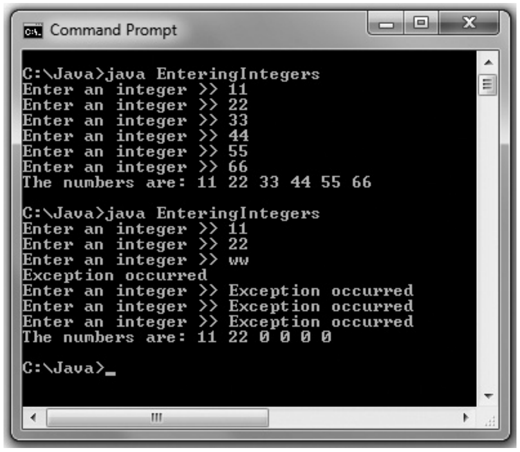
In Chapter 2, you learned to add a `nextLine()` call after any `next()`, `nextInt()`, or `nextDouble()` call to absorb the Enter key remaining in the input buffer before subsequent `nextLine()` calls. When you attempt to convert numeric data in a `try` block and the effort is followed by another attempted conversion, you also must remember to account for the potential remaining characters left in the input buffer. For example, Figure 12-12 shows a program that accepts and displays an array of six integers. The shaded and commented line is not part of the program when it is executed twice in Figure 12-13.

```
import java.util.Scanner;
public class EnteringIntegers
{
   public static void main(String[] args)
   {
      int[] numberList = {0, 0, 0, 0, 0, 0};
      int x;
      Scanner input = new Scanner(System.in);
      for(x = 0; x < numberList.length; ++x)
      {
         try
         {
            System.out.print("Enter an integer >> ");
            numberList[x] = input.nextInt();
         }
         catch(Exception e)
         {
            System.out.println("Exception occurred");
         }
         // input.nextLine();
      }
      System.out.print("The numbers are: ");
      for(x = 0; x < numberList.length; ++x)
         System.out.print(numberList[x] + " ");
      System.out.println();
   }
}
```

Figure 12-12 The `EnteringIntegers` program

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.



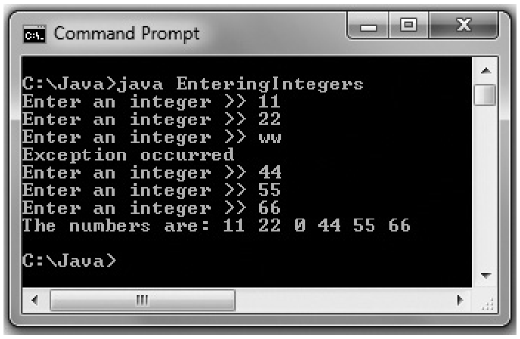
```
C:\Java>java EnteringIntegers
Enter an integer >> 11
Enter an integer >> 22
Enter an integer >> 33
Enter an integer >> 44
Enter an integer >> 55
Enter an integer >> 66
The numbers are: 11 22 33 44 55 66

C:\Java>java EnteringIntegers
Enter an integer >> 11
Enter an integer >> 22
Enter an integer >> ww
Exception occurred
Enter an integer >> Exception occurred
Enter an integer >> Exception occurred
Enter an integer >> Exception occurred
The numbers are: 11 22 0 0 0 0

C:\Java>_
```

Figure 12-13 Two typical executions of the `EnteringIntegers` program without the extra `nextLine()` call

In Figure 12-13, you can see that when a user enters valid data in the first execution, the program runs smoothly. However, in the second execution, the user enters some letters instead of numbers. The program correctly displays *Exception occurred*, but the user is not allowed to enter data for any of the remaining numbers. The problem can be corrected by uncommenting the shaded `nextLine()` call in the program in Figure 12-12. After the program is recompiled, it executes as shown in Figure 12-14. Now, the data entry exception is noted, but the user can continue entering data for the remaining array elements.



```
C:\Java>java EnteringIntegers
Enter an integer >> 11
Enter an integer >> 22
Enter an integer >> ww
Exception occurred
Enter an integer >> 44
Enter an integer >> 55
Enter an integer >> 66
The numbers are: 11 22 0 44 55 66

C:\Java>
```

Figure 12-14 A typical execution of the `EnteringIntegers` program with the extra `nextLine()` call

Not For Sale

Not For Sale

Declaring and Initializing Variables in try…catch Blocks

You can include any legal Java statements within a try block or catch block, including declaring variables. However, you must remember that a variable declared within a block is local to that block. In other words, the variable goes out of scope when the try or catch block ends, so any variable declared within one of the blocks should serve only a temporary purpose.

If you want to use a variable both with a try or catch block and afterward, then you must declare the variable before the try block begins. However, if you declare a variable before a try block but wait to assign its initial usable value within the try…catch block, you must be careful that the variable receives a useful value; otherwise, when you use the variable after the try…catch pair ends, the program will not compile.

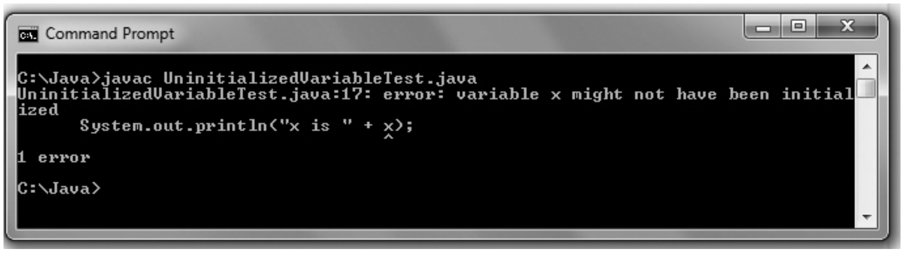
Figure 12-15 illustrates this scenario. In the UninitializedVariableTest program, x is declared and its value is received from the user in a try block. Because the user might not enter an integer, the conversion to an integer might fail, and an exception might be thrown. In this example, the catch block only displays a message and does not assign a useful value to x. When the program attempts to display x after the catch block, an error message is generated, as shown in Figure 12-16. You have three easy options for fixing this error:

- You can assign a value to x before the try block starts. That way, even if an exception is thrown, x will have a usable value to display in the last statement.
- You can assign a usable value to x within the catch block. That way, if an exception is thrown, x will again hold a usable value.
- You can move the output statement within the try block. If the conversion of the user's entry to an integer is successful, the try block finishes execution and the value of x is displayed. However, if the conversion fails, the try block is abandoned, the catch block executes, the error message is displayed, and x is not used.

```
import java.util.Scanner;
public class UninitializedVariableTest
{
   public static void main(String[] args)
   {
      int x;
      Scanner input = new Scanner(System.in);
      try
      {
         System.out.print("Enter an integer >> ");
         x = input.nextInt();
      }
      catch(Exception e)
      {
         System.out.println("Exception occurred");
      }
      System.out.println("x is " + x);
   }
}
```

Figure 12-15 The UninitializedVariableTest program

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.



```
C:\Java>javac UninitializedVariableTest.java
UninitializedVariableTest.java:17: error: variable x might not have been initial
ized
        System.out.println("x is " + x);
                                     ^
1 error

C:\Java>
```

Figure 12-16 The error message generated when compiling the `UninitializedVariableTest` program

Watch the video *Exceptions*.

TWO TRUTHS & A LIE

Trying Code and Catching Exceptions

1. A `try` block is a block of code you attempt to execute while acknowledging that an exception might occur.
2. You usually code at least one `catch` block immediately following a `try` block to handle an exception that might be thrown by the `try` block.
3. A `throw` statement is one that sends an `Exception` object to a `try` block so it can be handled.

The false statement is #3. A `throw` statement sends an `Exception` object to a `catch` block.

You Do It

Throwing and Catching an Exception

In this section, you create an application in which the user enters two values to be divided. The application catches an exception if either of the entered values is not an integer.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

(continued)

1. Open a new file in your text editor, and type the first few lines of an interactive application named **ExceptionDemo**.

```
import javax.swing.*;
public class ExceptionDemo
{
   public static void main(String[] args)
   {
```

2. Declare three integers—two to be input by the user and a third to hold the result after dividing the first two. The numerator and denominator variables must be assigned starting values because their values will be entered within a `try` block. The compiler understands that a `try` block might not complete; that is, it might throw an exception before it is through. Also declare an input `String` to hold the return value of the `JOptionPane showInputDialog()` method.

```
int numerator = 0, denominator = 0, result;
String inputString;
```

3. Add a `try` block that prompts the user for two values, converts each entered `String` to an integer, and divides the values, producing `result`.

```
try
{
   inputString = JOptionPane.showInputDialog(null,
      "Enter a number to be divided");
   numerator = Integer.parseInt(inputString);
   inputString = JOptionPane.showInputDialog(null,
      "Enter a number to divide into the first number");
   denominator = Integer.parseInt(inputString);
   result = numerator / denominator;
}
```

4. Add a `catch` block that catches an `ArithmeticException` object if division by 0 is attempted. If this block executes, display an error message, and force `result` to 0.

```
catch(ArithmeticException exception)
{
   JOptionPane.showMessageDialog(null, exception.getMessage());
   result = 0;
}
```

5. Whether or not the `try` block succeeds, display the result (which might have been set to 0). Include closing curly braces for the `main()` method and for the class.

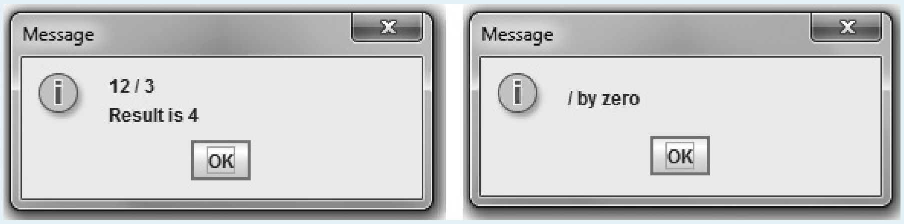
(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

```
            JOptionPane.showMessageDialog(null, numerator + " / " +
               denominator + "\nResult is " + result);
        }
    }
```

6. Save the file as **ExceptionDemo.java**, and then compile and execute the application. Enter two nonzero integer values. For example, the first execution in Figure 12-17 shows the output when the user enters *12* and *3* as the two input values. The application completes successfully. Click **OK** to end the application, and execute the `ExceptionDemo` application again. This time, enter **0** for the second value; the output looks like the second part of Figure 12-17. Click **OK** to end the application.



```
12 / 3
Result is 4
```

```
/ by zero
```

Figure 12-17 Output of two executions of the `ExceptionDemo` application

Throwing and Catching Multiple Exceptions

You can place as many statements as you need within a `try` block, and you can catch as many exceptions as you want. If you try 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.

When a program contains multiple `catch` blocks, they are examined in sequence until a match is found for the type of exception that occurred. Then, the matching `catch` block executes, and each remaining `catch` block is bypassed.

For example, consider the application in Figure 12-18. The `main()` method in the `DivisionMistakeCaught3` class throws two types of `Exception` objects: an `ArithmeticException` and an `InputMismatchException`. The `try` block in the application surrounds all the statements in which the exceptions might occur.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

```
import java.util.*;
public class DivisionMistakeCaught3
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      try
      {
         System.out.print("Enter numerator >> ");
         numerator = input.nextInt();
         System.out.print("Enter denominator >> ");
         denominator = input.nextInt();
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(ArithmeticException mistake)
      {
         System.out.println(mistake.getMessage());
      }
      catch(InputMismatchException mistake)
      {
         System.out.println("Wrong data type");
      }
   }
}
```

Figure 12-18 The `DivisionMistakeCaught3` class

The program in Figure 12-18 must import the `java.util.InputMismatchException` class to be able to use an `InputMismatchException` object. The `java.util` package is also needed for the `Scanner` class, so it's easiest to import the whole package.

If you use the `getMessage()` method with the `InputMismatchException` object, you see that the message is `null`, because `null` is the default message value for an `InputMismatchException` object.

In the `main()` method of the program in Figure 12-18, the `try` block executes. Several outcomes are possible:

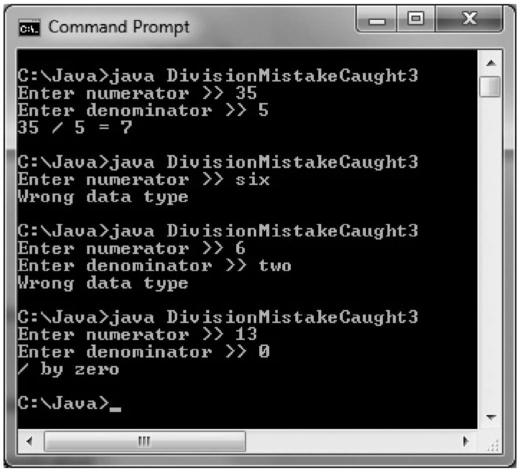
- If the user enters two usable integers, `result` is calculated, normal output is displayed, and neither `catch` block executes.
- If the user enters an invalid (noninteger) value at either the first or second shaded statement, an `InputMismatchException` object is created and thrown. When the program encounters the first `catch` block (that catches an `ArithmeticException`), the block is

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

bypassed because the `Exception` types do not match. When the program encounters the second `catch` block, the types match, and the "Wrong data type" message is displayed.

- If the user enters 0 for `denominator`, the division statement throws an `ArithmeticException`, and the `try` block is abandoned. When the program encounters the first `catch` block, the `Exception` types match, the value of the `getMessage()` method is displayed, and then the second `catch` block is bypassed.

Figure 12-19 shows the output of four typical program executions.



```
C:\Java>java DivisionMistakeCaught3
Enter numerator >> 35
Enter denominator >> 5
35 / 5 = 7

C:\Java>java DivisionMistakeCaught3
Enter numerator >> six
Wrong data type

C:\Java>java DivisionMistakeCaught3
Enter numerator >> 6
Enter denominator >> two
Wrong data type

C:\Java>java DivisionMistakeCaught3
Enter numerator >> 13
Enter denominator >> 0
/ by zero

C:\Java>_
```

Figure 12-19 Four executions of the `DivisionMistakeCaught3` application

When you list multiple `catch` blocks following a `try` block, you must be careful that some `catch` blocks don't become unreachable. Unreachable statements are program statements that can never execute under any circumstances. For example, if two successive `catch` blocks catch an `ArithmeticException` and an ordinary `Exception`, respectively, the `ArithmeticException` errors cause the first `catch` to execute and other types that derive from `Exception` "fall through" to the more general `Exception catch` block. However, if you reverse the sequence of the `catch` blocks so the one that catches general `Exception` objects is first, even `ArithmeticExceptions` would be caught by the `Exception catch`. The `ArithmeticException catch` block therefore is unreachable because the `Exception catch` block is in its way, and the class does not compile. Think of arranging your `catch` blocks so that the "bigger basket" is always below a smaller one. That is, each `Exception` should "fall through" as many `catch` blocks as necessary to reach the one that will hold it.

You first learned about unreachable statements in Chapter 3. For example, statements that follow a method's `return` statement are unreachable. Creating an unreachable `catch` block causes a compiler error that generates a message indicating that the exception "has already been caught."

Not For Sale

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

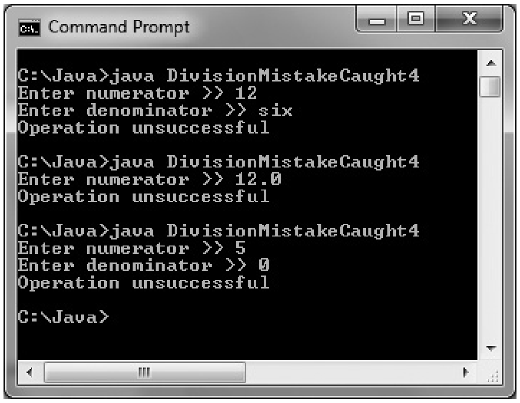
Sometimes, you want to execute the same code no matter which Exception type occurs. For example, within the DivisionMistakeCaught3 application in Figure 12-18, each of the two catch blocks displays a unique message. Instead, you might want both catch blocks to display the same message. Because ArithmeticExceptions and InputMismatchExceptions are both subclasses of Exception, you can rewrite the program as shown in Figure 12-20, using a single generic catch block (shaded) that can catch any type of Exception object.

```
import java.util.*;
public class DivisionMistakeCaught4
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      try
      {
         System.out.print("Enter numerator >> ");
         numerator = input.nextInt();
         System.out.print("Enter denominator >> ");
         denominator = input.nextInt();
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(Exception mistake)
      {
         System.out.println("Operation unsuccessful");
      }
   }
}
```

Figure 12-20 The DivisionMistakeCaught4 application

The catch block in Figure 12-20 accepts a more generic Exception argument type than that thrown by either of the potentially error-causing try statements, so the generic catch block can act as a “catch-all” block. When either an arithmetic error or incorrect input type error occurs, the thrown exception is “promoted” to an Exception error in the catch block. Figure 12-21 shows several executions of the DivisionMistakeCaught4 application. Notice that no matter which type of mistake occurs during execution, the general “Operation unsuccessful” message is displayed by the generic catch block.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.



```
C:\Java>java DivisionMistakeCaught4
Enter numerator >> 12
Enter denominator >> six
Operation unsuccessful

C:\Java>java DivisionMistakeCaught4
Enter numerator >> 12.0
Operation unsuccessful

C:\Java>java DivisionMistakeCaught4
Enter numerator >> 5
Enter denominator >> 0
Operation unsuccessful

C:\Java>
```

Figure 12-21 Several executions of the `DivisionMistakeCaught4` application

As a new feature in Java 7, a `catch` block can also be written to catch specific multiple exception types. For example, the following `catch` block catches two `Exception` types. When either is caught, its local identifier is `e`.

```
catch(ArithmeticException, InputMismatchException e)
{
}
```

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

- Perhaps the method is trying to accomplish too many diverse tasks and should be broken up into smaller methods.
- Perhaps the `Exception` types thrown are too specific and should be generalized, as they are in the `DivisionMistakeCaught4` application in Figure 12-20.

Watch the video *Catching Multiple Exceptions*.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

TWO TRUTHS & A LIE

Throwing and Catching Multiple Exceptions

1. When multiple `try` block statements throw exceptions, multiple `catch` blocks might execute.
2. As soon as an exception occurs, the `try` block that contains it is abandoned and the rest of its statements are unexecuted.
3. When a program contains multiple `catch` blocks, the first one that matches the thrown `Exception` type is the one that executes.

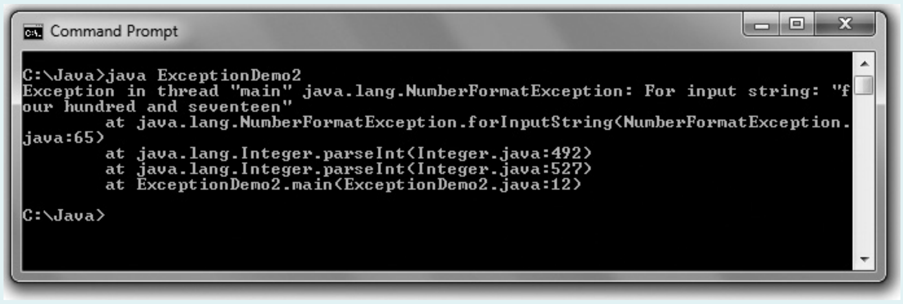
The false statement is #1. If you try more than one statement, only the first error-generating statement throws an exception, and then the rest of the `try` block is abandoned.

You Do It

Using Multiple `catch` Blocks

In this section, you add a second `catch` block to the `ExceptionDemo` application.

1. Open the **ExceptionDemo.java** file. Change the class name to **`ExceptionDemo2`**, and save the file as **ExceptionDemo2.java**.
2. Execute the program, and enter a noninteger value at one of the prompts. Program execution fails. For example, Figure 12-22 shows the error generated when the user types the string "four hundred and seventeen" at the first prompt.



```
C:\Java>java ExceptionDemo2
Exception in thread "main" java.lang.NumberFormatException: For input string: "f
our hundred and seventeen"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:65)
        at java.lang.Integer.parseInt(Integer.java:492)
        at java.lang.Integer.parseInt(Integer.java:527)
        at ExceptionDemo2.main(ExceptionDemo2.java:12)

C:\Java>
```

Figure 12-22 Error message generated by the current version of the `ExceptionDemo2` application when a user enters a noninteger value

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

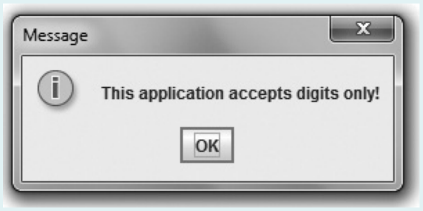
© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

3. After the existing `catch` block that catches an `ArithmeticException` object, add a `catch` block that catches a `NumberFormatException` object if neither user entry can be converted to an integer. If this block executes, display an error message, set `numerator` and `denominator` to a default value of 999, and force `result` to 1.

```
catch(NumberFormatException exception)
{
   JOptionPane.showMessageDialog(null,
      "This application accepts digits only!");
   numerator = 999;
   denominator = 999;
   result = 1;
}
```

4. Save, compile, and execute the program. This time, if you enter a noninteger value, the output appears as shown in Figure 12-23. Click **OK** to end the application.



This application accepts digits only!

Figure 12-23 Error message generated by the improved version of the `ExceptionDemo2` application when a user enters a noninteger value

5. Execute the application a few more times by entering a variety of valid and invalid data. Confirm that the program works appropriately whether you type two usable integers, an unusable 0 for the second integer, or noninteger data such as strings containing alphabetic characters or punctuation.

Using the `finally` Block

When you have actions you must perform at the end of a `try...catch` sequence, you can use a **`finally` block**. The code within a `finally` block executes regardless of whether the preceding `try` block identifies an exception. Usually, you use a `finally` block to perform cleanup tasks that must happen whether or not any exceptions occurred, and whether or not any exceptions that occurred were caught. Figure 12-24 shows the format of a `try...catch` sequence that uses a `finally` block.

Not For Sale

Not For Sale

```
try
{
   // statements to try
}
catch(Exception e)
{
   // actions that occur if exception was thrown
}
finally
{
   // actions that occur whether catch block executed or not
}
```

Figure 12-24 Format of `try`…`catch`…`finally` sequence

Compare Figure 12-24 to Figure 12-6 shown earlier in this chapter. When the `try` code works without error in Figure 12-6, control passes to the statements at the end of the method. Also, when the `try` code fails and throws an exception, and the `Exception` object is caught, the `catch` block executes, and control again passes to the statements at the end of the method. At first glance, it seems as though the statements at the end of the method in Figure 12-6 always execute. However, the final set of statements might never execute for at least two reasons:

- Any `try` block might throw an `Exception` object for which you did not provide a `catch` block. After all, exceptions occur all the time without your handling them, as one did in the first `Division` application in Figure 12-2 earlier in this chapter. In the case of an unhandled exception, program execution stops immediately, the exception is sent to the operating system for handling, and the current method is abandoned.
- The `try` or `catch` block might contain a `System.exit();` statement, which stops execution immediately.

When you include a `finally` block, you are assured that the `finally` statements will execute before the method is abandoned, even if the method concludes prematurely. For example, programmers often use a `finally` block when the program uses data files that must be closed. You will learn more about writing to and reading from data files in the next chapter. For now, however, consider the format shown in Figure 12-25, which represents part of the logic for a typical file-handling program:

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

```
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
   // System exit
}
finally
{
   // If the file is open, close it
}
```

Figure 12-25 Pseudocode that tries reading a file and handles an `IOException`

The pseudocode in Figure 12-25 represents an application that opens a file; in Java, if a file does not exist when you open it, an input/output exception, or `IOException`, is thrown and a `catch` block can handle the error. However, because the application in Figure 12-25 uses an array, an uncaught `IndexOutOfBoundsException` might occur even though the file opened successfully. (An `IndexOutOfBoundsException` occurs, as its name implies, when a subscript is not in the range of valid subscripts for an array.) The `IndexOutOfBoundsException` would not be caught by the existing `catch` block. Also, because the application calculates an average, it might divide by 0 and an `ArithmeticException` might occur; it also would not be caught. In any of these events, you might 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 control returns 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 `catch` executes.
- An uncaught exception causes the method to abandon prematurely. An uncaught exception does not allow the `try` block to finish, nor does it cause the `catch` block to execute.

If an application might throw several types of exceptions, you can try some code, catch the possible exception, try some more code and catch the possible exception, and so on. Usually, however, the superior approach is to try all the statements that might throw exceptions, and then include all the needed `catch` blocks and an optional `finally` block. This is the approach shown in Figure 12-25, and it usually results in logic that is easier to follow.

Not For Sale

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

You 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 12-25, 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 `System exit`. However, writing code just once in a `finally` block is clearer and less prone to error.

If a `try` block calls the `System.exit()` method and the `finally` block calls the same method, the `exit()` method in the `finally` block executes. The `try` block's `exit()` method call is abandoned.

C++ programmers are familiar with `try` and `catch` blocks, but C++ does not provide a `finally` block. C# and Visual Basic contain the keywords `try`, `catch`, and `finally`.

TWO TRUTHS & A LIE

Using the `finally` Block

1. The code within a `finally` block executes when a `try` block identifies an exception that is not caught.
2. Usually, you use a `finally` block to perform cleanup tasks that must happen whether or not any exceptions occurred, and whether or not any exceptions that occurred were caught.
3. It's possible that the code that follows a `try...catch...finally` sequence might never execute—for example, if a `try` block throws an unhandled exception.

The false statement is #1. The code within a `finally` block executes whether or not the preceding `try` block identifies an exception, and whether or not an exception is caught.

Understanding the Advantages of Exception Handling

Before the inception of object-oriented programming languages, potential program errors were handled using somewhat confusing, error-prone methods. For example, a traditional, non-object-oriented procedural program might perform three methods that depend on each other using code that provides error checking similar to the pseudocode in Figure 12-26.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

```
call methodA()
if methodA() worked
{
   call methodB()
   if methodB() worked
   {
      call methodC()
      if methodC() worked
         everything's okay, so display finalResult
      else
         set errorCode to 'C'
   }
   else
      set errorCode to 'B'
}
else
   set errorCode to 'A'
```

Figure 12-26 Pseudocode representing traditional error checking

The pseudocode in Figure 12-26 represents an application in which the logic must pass three tests before `finalResult` can be displayed. The program executes `methodA()`; it then calls `methodB()` only if `methodA()` is successful. Similarly, `methodC()` executes only when `methodA()` and `methodB()` are both successful. When any method fails, the program sets an appropriate `errorCode` to 'A', 'B', or 'C'. (Presumably, the `errorCode` is used later in the application.) The logic is difficult to follow, and the application's purpose and intended usual outcome—to display `finalResult`—is lost in the maze of `if` statements. Also, you can easily make coding mistakes within such a program because of the complicated nesting, indenting, and opening and closing of curly braces.

Compare the same program logic using Java's object-oriented, error-handling technique shown in Figure 12-27. Using the `try...catch` object-oriented technique provides the same results as the traditional method, but the statements of the program that do the "real" work (calling methods `A`, `B`, and `C` and displaying `finalResult`) are placed together, where their logic is easy to follow. The `try` steps should usually work without generating errors; after all, the errors are "exceptions." It is convenient to see these business-as-usual steps in one location. The unusual, exceptional events are grouped and moved out of the way of the primary action.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

```
try
{
   call methodA() and maybe throw an exception
   call methodB() and maybe throw an exception
   call methodC() and maybe throw an exception
   everything's okay, so display finalResult
}
catch(methodA()'s error)
{
   set errorCode to "A"
}
catch(methodB()'s error)
{
   set errorCode to "B"
}
catch(methodC()'s error)
{
   set errorCode to "C"
}
```

Figure 12-27 Pseudocode representing object-oriented exception handling

Besides clarity, an advantage to object-oriented exception handling is the flexibility it allows in the handling of error situations. When a method you write throws an exception, the same method can catch the exception, although it is not required to do so, and in most object-oriented programs it does not. Often, you don't want a method to handle its own exception. In many cases, you want the method to check for errors, but you do not want to require a method to handle an error if it finds one. Another advantage to object-oriented exception handling is that you gain the ability to appropriately deal with exceptions as you decide how to handle them. When you write a method, it can call another, catch a thrown exception, and you can decide what you want to do. Just as a police officer has leeway to deal with a speeding driver differently depending on circumstances, programs can react to exceptions specifically for their current purposes.

Methods are flexible partly because they are reusable—that is, a well-written method might be used by any number of applications. Each calling application might need to handle a thrown error differently, depending on its purpose. For example, an application that uses a method that divides values might need to terminate if division by 0 occurs. A different program simply might want the user to reenter the data to be used, and a third program might want to force division by 1. The method that contains the division statement can throw the error, but each calling program can assume responsibility for handling the error detected by the method in an appropriate way.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

TWO TRUTHS & A LIE

Understanding the Advantages of Exception Handling

1. An advantage to using object-oriented error-handling techniques is that programs are clearer and more flexible.
2. An advantage to using object-oriented error-handling techniques is that when a method throws an exception, it will always be handled in the same, consistent way.
3. 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.

The false statement is #2. A well-written method might be used by any number of applications. An advantage of object-oriented exception-handling techniques is that each calling application can handle thrown errors differently, depending on its purpose.

Specifying the Exceptions That a Method Can Throw

If a method throws an exception that it will not catch but will be caught by a different method, you must use the keyword `throws` followed by an `Exception` type in the method header. This practice is known as **exception specification**.

For example, Figure 12-28 shows a `PriceList` class used by a company to hold a list of prices for items it sells. For simplicity, there are only four prices and a single method that displays the price of a single item. The `displayPrice()` method accepts a parameter to use as the array subscript, but because the subscript could be out of bounds, the method contains a shaded `throws` clause, acknowledging it could throw an exception.

```
public class PriceList
{
   private static final double[] price = {15.99, 27.88, 34.56, 45.89};
   public static void displayPrice(int item) throws IndexOutOfBoundsException
   {
      System.out.println("The price is $" + price[item]);
   }
}
```

Figure 12-28 The `PriceList` class

Not For Sale

Not For Sale

Figures 12-29 and 12-30 show two applications in which programmers have chosen to handle the potential exception differently. In the first class, `PriceListApplication1`, the programmer has chosen to handle the exception in the shaded `catch` block by displaying a price of $0. In the second class, `PriceListApplication2`, the programmer has chosen to handle the exception by using the highest price in the array. Figure 12-31 shows several executions of each program. Other programmers writing other applications that use the `PriceList` class could choose still different actions, but they all can use the flexible `displayPrice()` method because it doesn't limit the calling method's choice of recourse.

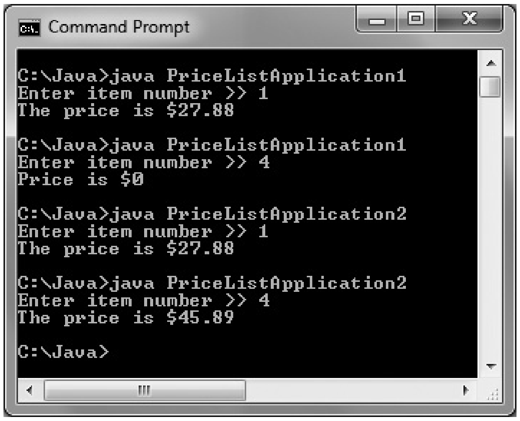
```
import java.util.*;
public class PriceListApplication1
{
   public static void main(String[] args)
   {
      int item;
      Scanner input = new Scanner(System.in);
      System.out.print("Enter item number >> ");
      item = input.nextInt();
      try
      {
         PriceList.displayPrice(item);
      }
      catch(IndexOutOfBoundsException e)
      {
         System.out.println("Price is $0");
      }
   }
}
```

Figure 12-29 The `PriceListApplication1` class

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

```
import java.util.*;
public class PriceListApplication2
{
   public static void main(String[] args)
   {
      int item;
      Scanner input = new Scanner(System.in);
      final int MAXITEM = 3;
      System.out.print("Enter item number >> ");
      item = input.nextInt();
      try
      {
         PriceList.displayPrice(item);
      }
      catch(IndexOutOfBoundsException e)
      {
         PriceList.displayPrice(MAXITEM);
      }
   }
}
```

Figure 12-30 The `PriceListApplication2` class



```
C:\Java>java PriceListApplication1
Enter item number >> 1
The price is $27.88

C:\Java>java PriceListApplication1
Enter item number >> 4
Price is $0

C:\Java>java PriceListApplication2
Enter item number >> 1
The price is $27.88

C:\Java>java PriceListApplication2
Enter item number >> 4
The price is $45.89

C:\Java>
```

Figure 12-31 Several executions of `PriceListApplication1` and `PriceListApplication2`

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

For most Java methods that you write, you do not use a `throws` clause. For example, you have not needed to use a `throws` clause in any of the many programs you have written while working through this book; however, in those methods, if you divided by 0 or went beyond an array's bounds, an exception was thrown nevertheless. Most of the time, you let Java handle any exception by shutting down the program. Imagine how unwieldy your programs would become if you were required to provide instructions for handling every possible error, including equipment failures and memory problems. Most exceptions never have to be explicitly thrown or caught, nor do you have to include a `throws` clause in the headers of methods that automatically throw these exceptions. The only exceptions that must be caught or named in a `throws` clause are the type known as *checked* exceptions.

Java's exceptions can be categorized into two types:

- **Unchecked exceptions**—These exceptions inherit from the `Error` class or the `RuntimeException` class. Although you *can* handle these exceptions in your programs, you are not required to do so. For example, dividing by zero is a type of `RuntimeException`, and you are not required to handle this exception—you can simply let the program terminate.
- **Checked exceptions**—These exceptions are the type that programmers should anticipate and from which programs should be able to recover. All exceptions that you explicitly throw and that descend from the `Exception` class are checked exceptions.

Java programmers say that checked exceptions are subject to the **catch or specify requirement**, which means if you throw a checked exception from a method, you must do one of the following:

- Catch it within the method.
- Specify the exception in your method header's `throws` clause.

Code that uses a checked exception will not compile if the catch or specify rule is not followed.

If you write a method with a `throws` clause in the header, then any method that uses your method must do one of the following:

- Catch and handle the possible exception.
- Declare the exception in its `throws` clause. The called method can then rethrow the exception to yet another method that might either catch it or throw it yet again.

In other words, when an exception is a checked exception, client programmers are forced to deal with the possibility that an exception will be thrown.

Some programmers feel that using checked exceptions is an example of "syntactic salt." **Syntactic sugar** is a term coined by Peter J. Landin to describe aspects of a computer language that make it "sweeter," or easier, for programmers to use. For example, you learned in Chapter 1 that you do not have to write `import java.lang;` at the top of every Java program file because the package is automatically imported for you. The metaphor has been extended by the term **syntactic salt**, which is a language feature designed to make it harder to write bad code.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

If you write a method that explicitly throws a checked exception that is not caught within the method, Java requires that you use the `throws` clause in the header of the method. Using the `throws` clause does not mean that the method *will* throw an exception—everything might go smoothly. Instead, it means the method *might* throw an exception. You include the `throws` clause in the method header so applications that use your methods are notified of the potential for an exception.

In Chapter 3, you learned that a method's signature is the combination of the method name and the number, types, and order of arguments. Some programmers argue that any `throws` clause is also part of the signature, but most authorities disagree. You cannot create a class that contains multiple methods that differ only in their `return` types; such methods are not overloaded. The same is true for methods with the same signatures that differ only in their `throws` clauses; the compiler considers the methods to have an identical signature. Instead of saying that the `throws` clause is part of the method's signature, you might prefer to say that it is part of the method's interface.

A method that overrides another cannot throw an exception unless it throws the same type as its parent or a subclass of its parent's thrown type. These rules do not apply to overloaded methods. Any exceptions may (or may not) be thrown from one version of an overloaded method without considering what exceptions are thrown by other versions of an overloaded method.

To be able to use a method to its full potential, you must know the method's name and three additional pieces of information:

- The method's `return` type
- The type and number of arguments the method requires
- The type and number of exceptions the method throws

To use a method, you must know what types of arguments are required. You can call a method without knowing its `return` type, but if you do, you can't benefit from any value that the method returns. (Also, if you use a method without knowing its return type, you probably don't understand the purpose of the method.) Likewise, you can't make sound decisions about what to do in case of an error if you don't know what types of exceptions a method might throw.

When a method might throw more than one exception type, you can specify a list of potential exceptions in the method header by separating them with commas. As an alternative, if all the exceptions descend from the same parent, you can specify the more general parent class. For example, if your method might throw either an `ArithmeticException` or an `ArrayIndexOutOfBoundsException`, you can just specify that your method throws a `RuntimeException`. One advantage to this technique is that when your method is modified to include more specific `RuntimeExceptions` in the future, the method header will not change. This saves time and money for users of your methods, who will not have to modify their own methods to accommodate new `RuntimeException` types.

An extreme alternative is simply to specify that your method throws a general `Exception` object, so that all exceptions are included in one clause. Doing this simplifies the exception

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

specification you write. However, using this technique disguises information about the specific types of exceptions that might occur, and such information usually has value to users of your methods.

Usually, you declare only checked exceptions. Remember that runtime exceptions can occur anywhere in a program, and they can be numerous. Programs would be less clear and more cumbersome if you had to account for runtime exceptions in every method declaration. Therefore, the Java compiler does not require that you catch or specify runtime exceptions.

Watch the video *Specifying Exceptions*.

TWO TRUTHS & A LIE

Specifying the Exceptions That a Method Can Throw

1. Exception specification is the practice of listing possible exceptions in a `throws` clause in a method header.
2. Many exceptions never have to be explicitly thrown or caught, nor do you have to include a `throws` clause in the headers of methods that automatically throw these exceptions.
3. If you write a method with a `throws` clause for a checked exception in the header, then any method that uses your method must catch and handle the possible exception.

The false statement is #3. If you write a method with a `throws` clause for a checked exception in the header, then any method that uses your method must catch and handle the possible exception *or* declare the exception in its `throws` clause so the exception can be rethrown.

Tracing Exceptions 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 completed. For example, if `methodA()` calls `methodB()`, the operating system has to "remember" to return to `methodA()` when `methodB()` ends. Likewise, if `methodB()` calls `methodC()`, the computer must "remember" while `methodC()` executes to return to `methodB()` and eventually to `methodA()`. The memory location known as the **call stack** is where the computer stores the list of method locations to which the system must return.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

When a method throws an exception and the method does not catch it, the exception is thrown to the next method up the call stack, or in other words, to the method that called the offending method. Figure 12-32 shows how the call stack works. If `methodA()` calls `methodB()`, and `methodB()` calls `methodC()`, and `methodC()` throws an exception, Java first looks for a `catch` block in `methodC()`. If none exists, Java looks for the same thing in `methodB()`. If `methodB()` does not have a `catch` block, Java looks to `methodA()`. If `methodA()` cannot catch the exception, it is thrown to the Java Virtual Machine, which displays a message at the command prompt.

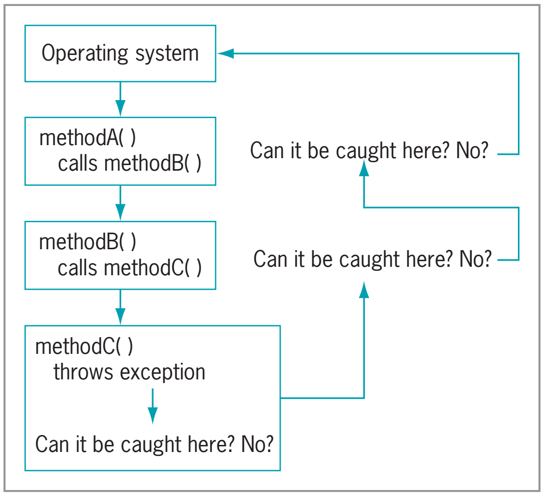

Operating system
methodA( ) calls methodB( )
methodB( ) calls methodC( )
methodC( ) throws exception
Can it be caught here? No?
Can it be caught here? No?
Can it be caught here? No?

Figure 12-32 Cycling through the call stack

For example, examine the application in Figure 12-33. The `main()` method of the application calls `methodA()`, which displays a message and calls `methodB()`. Within `methodB()`, another message is displayed and `methodC()` is called. In `methodC()`, yet another message is displayed. Then, a three-integer array is declared, and the program attempts to display the fourth element in the array. This program compiles correctly—no error is detected until `methodC()` attempts to access the out-of-range array element. In Figure 12-33, the comments indicate line numbers so you can more easily follow the sequence of generated error messages. You probably would not add such comments to a working application. Figure 12-34 shows the output when the application executes.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

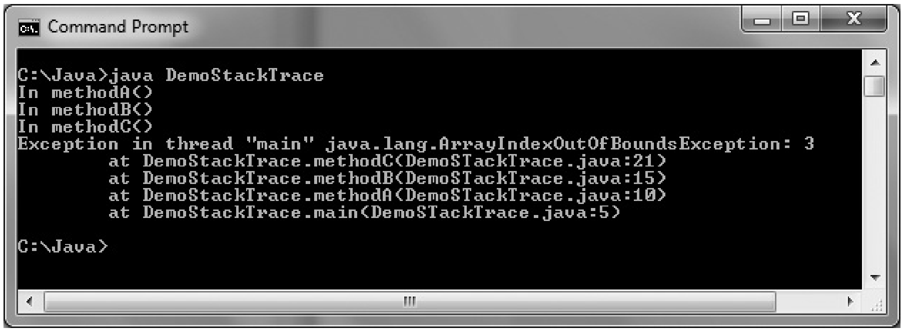
Not For Sale

Not For Sale

```
public class DemoStackTrace
{
   public static void main(String[] args)
   {
      methodA();  // line 5
   }
   public static void methodA()
   {
      System.out.println("In methodA()");
      methodB();  // line 10
   }
   public static void methodB()
   {
      System.out.println("In methodB()");
      methodC();  // line 15
   }
   public static void methodC()
   {
      System.out.println("In methodC()");
      int [] array = {0, 1, 2};
      System.out.println(array[3]);  // line 21
   }
}
```

Don't Do It
You never would purposely use an out-of-range subscript in a professional program.

Figure 12-33 The `DemoStackTrace` class



```
C:\Java>java DemoStackTrace
In methodA()
In methodB()
In methodC()
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at DemoStackTrace.methodC(DemoSTackTrace.java:21)
        at DemoStackTrace.methodB(DemoSTackTrace.java:15)
        at DemoStackTrace.methodA(DemoSTackTrace.java:10)
        at DemoStackTrace.main(DemoSTackTrace.java:5)

C:\Java>
```

Figure 12-34 Error messages generated by the `DemoStackTrace` application

As you can see in Figure 12-34, three messages are displayed, indicating that `methodA()`, `methodB()`, and `methodC()` were called in order. However, when `methodC()` attempts to access the out-of-range element in the array, an `ArrayIndexOutOfBoundsException` is automatically thrown. The error message generated shows that the exception occurred at line 21 of the file in `methodC()`, which was called in line 15 of the file by `methodB()`, which was

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

called in line 10 of the file by `methodA()`, which was called by the `main()` method in line 5 of the file. Using this list of error messages, you could track down the location where the error was generated. Of course, in a larger application that contains thousands of lines of code, the stack trace history list would be even more useful.

The technique of cycling through the methods in the stack has great advantages because it allows methods to handle exceptions wherever the programmer has decided it is most appropriate—including allowing the operating system to handle the error. However, when a program uses several classes, the disadvantage is that the programmer finds it difficult to locate the original source of an exception.

You have already used the `Throwable` method `getMessage()` to obtain information about an `Exception` object. Another useful `Exception` method is the `printStackTrace()` method. When you catch an `Exception` object, you can call `printStackTrace()` to display a list of methods in the call stack and determine the location of the statement that caused the exception.

For example, Figure 12-35 shows a `DemoStackTrace2` application in which the `printStackTrace()` method produces a trace of the trail taken by a thrown exception. The differences in the executable statements from the `DemoStackTrace` application are shaded. The call to `methodB()` has been placed in a `try` block so that the exception can be caught. Instead of throwing the exception to the operating system, this application catches the exception, displays a stack trace history list, and continues to execute. The output of the list of methods in Figure 12-36 is similar to the one shown in Figure 12-34, but the application does not end abruptly.

```
public class DemoStackTrace2
{
   public static void main(String[] args)
   {
      methodA();  // line 5
   }
   public static void methodA()
   {
      System.out.println("In methodA()");
      try
      {
         methodB();  // line 12
      }
      catch(ArrayIndexOutOfBoundsException error)
      {
         System.out.println("In methodA() - The stack trace:");
         error.printStackTrace();
      }
      System.out.println("methodA() ends normally.");
      System.out.println("Application could continue " +
         "from this point.");
   }
```

Figure 12-35 The `DemoStackTrace2` class *(continues)*

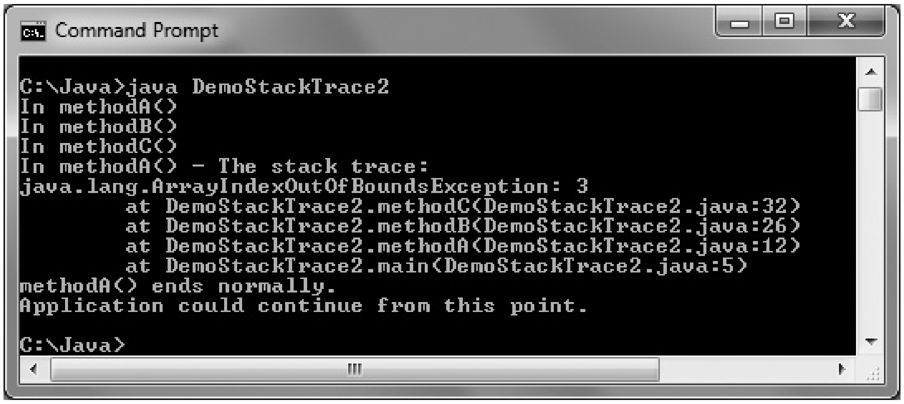
Not For Sale

Not For Sale

(continued)

```
    public static void methodB()
    {
       System.out.println("In methodB()");
       methodC();  // line 26
    }
    public static void methodC()
    {
       System.out.println("In methodC()");
       int[] array = {0, 1, 2};
       System.out.println(array[3]);  // line 32
    }
 }
```

Figure 12-35 The `DemoStackTrace2` class



```
C:\Java>java DemoStackTrace2
In methodA()
In methodB()
In methodC()
In methodA() - The stack trace:
java.lang.ArrayIndexOutOfBoundsException: 3
        at DemoStackTrace2.methodC(DemoStackTrace2.java:32)
        at DemoStackTrace2.methodB(DemoStackTrace2.java:26)
        at DemoStackTrace2.methodA(DemoStackTrace2.java:12)
        at DemoStackTrace2.main(DemoStackTrace2.java:5)
methodA() ends normally.
Application could continue from this point.

C:\Java>
```

Figure 12-36 Output of the `DemoStackTrace2` application

Usually, you do not want to place a `printStackTrace()` method call in a finished program. The typical application user has no interest in the cryptic messages that are displayed. However, while you are developing an application, `printStackTrace()` can be a useful tool for diagnosing your class's problems.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

TWO TRUTHS & A LIE

Tracing Exceptions Through the Call Stack

1. The memory location known as the call stack is where the computer stores the list of locations to which the system must return after each method call.
2. When a method throws an exception and the method does not catch it, the exception is thrown to the next method down the call stack, or in other words, to the next method that the offending method calls.
3. When you catch an exception, you can call `printStackTrace()` to display a list of methods in the call stack and determine the location of the statement that caused the exception. However, usually you do not want to place a `printStackTrace()` method call in a finished program.

The false statement is #2. When a method throws an exception and the method does not catch it, the exception is thrown to the next method up the call stack, or in other words, to the method that called the offending method.

Creating Your Own `Exception` Classes

Java provides over 40 categories of `Exceptions` that you can use in your programs. However, Java's creators could not predict every condition that might be an exception in your applications. 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; for example, 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 Java also allows you to create your own `Exception` classes.

To create your own throwable `Exception` class, you must extend a subclass of `Throwable`. Recall from Figure 12-1 that `Throwable` has two subclasses, `Exception` and `Error`, which are used to distinguish between recoverable and nonrecoverable errors. Because you always want to create your own exceptions for recoverable errors, your classes should extend the `Exception` class. You can extend any existing `Exception` subclass, such as `ArithmeticException` or `NullPointerException`, but usually you want to inherit directly from `Exception`. When you create an `Exception` subclass, it's conventional to end the name with *`Exception`*.

The `Exception` class contains four constructors as follows:

- `Exception()`—Constructs a new `Exception` object with `null` as its detail message
- `Exception(String message)`—Constructs a new `Exception` object with the specified detail message

Not For Sale

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

- `Exception(String message, Throwable cause)`—Constructs a new `Exception` object with the specified detail message and cause
- `Exception(Throwable cause)`—Constructs a new `Exception` object with the specified cause and a detail message of `cause.toString()`, which typically contains the class and the detail message of `cause`, or `null` if the cause argument is `null`

For example, Figure 12-37 shows a `HighBalanceException` class. Its constructor contains a single statement that passes a description of an error to the parent `Exception` constructor. This `String` would be retrieved if you called the `getMessage()` method with a `HighBalanceException` object.

```
public class HighBalanceException extends Exception
{
   public HighBalanceException()
   {
      super("Customer balance is high");
   }
}
```

Figure 12-37 The `HighBalanceException` class

Figure 12-38 shows a `CustomerAccount` class that uses a `HighBalanceException`. The `CustomerAccount` constructor header indicates that it might throw a `HighBalanceException` (see the first shaded statement); if the balance used as an argument to the constructor exceeds a set limit, a new, unnamed instance of the `HighBalanceException` class is thrown (see the second shaded statement).

```
public class CustomerAccount
{
   private int acctNum;
   private double balance;
   public static double HIGH_CREDIT_LIMIT = 20000.00;
   public CustomerAccount(int num, double bal) throws HighBalanceException
   {
      acctNum = num;
      balance = bal;
      if(balance > HIGH_CREDIT_LIMIT)
         throw(new HighBalanceException());
   }
}
```

Figure 12-38 The `CustomerAccount` class

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

In the CustomerAccount class in Figure 12-38, you could choose to instantiate a named HighBalanceException and throw it when the balance exceeds the credit limit. By waiting and instantiating an unnamed object only when it is needed, you improve program performance.

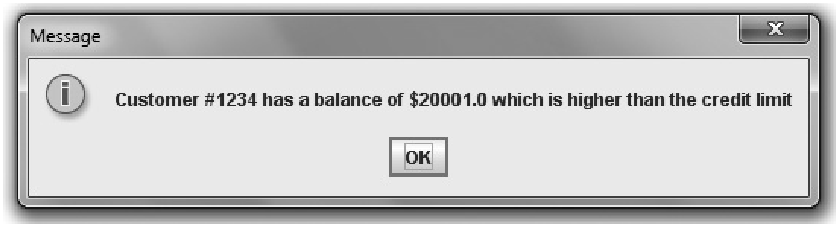
Figure 12-39 shows an application that instantiates a CustomerAccount. In this application, a user is prompted for an account number and balance. After the values are entered, an attempt is made to construct a CustomerAccount in a try block (as shown in the first shaded section). If the attempt is successful—that is, if the CustomerAccount constructor does not throw an Exception—the CustomerAccount information is displayed in a dialog box. However, if the CustomerAccount constructor does throw a HighBalanceException, the catch block receives it (as shown in the second shaded section) and displays a message. A different application could take any number of different actions; for example, it could display the return value of the getMessage() method, construct a CustomerAccount object with a lower balance, or construct a different type of object—perhaps a child of CustomerAccount called PreferredCustomerAccount that allows a higher balance. Figure 12-40 shows typical output of the application in a case in which a customer's balance is too high.

```
import javax.swing.*;
public class UseCustomerAccount
{
   public static void main(String[] args)
   {
      int num;
      double balance;
      String input;
      input = JOptionPane.showInputDialog(null,
         "Enter account number");
      num = Integer.parseInt(input);
      input = JOptionPane.showInputDialog(null, "Enter balance due");
      balance = Double.parseDouble(input);
      try
      {
         CustomerAccount ca = new CustomerAccount(num, balance);
         JOptionPane.showMessageDialog(null, "Customer #" +
            num + " has a balance of $" + balance);
      }
      catch( HighBalanceException hbe)
      {
         JOptionPane.showMessageDialog(null, "Customer #" +
            num + " has a balance of $" + balance +
            " which is higher than the credit limit");
      }
   }
}
```

Figure 12-39 The UseCustomerAccount class

Not For Sale

Not For Sale



Customer #1234 has a balance of $20001.0 which is higher than the credit limit

Figure 12-40 Typical output of the `UseCustomerAccount` application

Instead of hard coding error messages into your exception classes, as shown in Figure 12-39, you might consider creating a catalog of possible messages to use. This approach provides several advantages:

- All the messages are stored in one location instead of being scattered throughout the program, making them easier to see and modify.
- The list of possible errors serves as a source of documentation, listing potential problems when running the application.
- Other applications might want to use the same catalog of messages.
- If your application will be used internationally, you can provide messages in multiple languages, and other programmers can use the version that is appropriate for their country.

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 `throw(new RuntimeException());`. Of course, you would want to do so only with good reason because Java handles `RuntimeExceptions` for you by stopping the program. Because you cannot anticipate every possible error, Java's automatic response is often the best course of action.

You should not create an excessive number of special `Exception` types for your classes, especially if the Java development environment already contains an `Exception` class that will catch the error. Extra `Exception` types add complexity for other programmers who use your classes. However, when appropriate, specialized `Exception` classes provide an elegant way for you to handle error situations. They enable you to separate your error code from the usual, nonexceptional sequence of events; they allow errors to be passed up the stack and traced; and they allow clients of your classes to handle exceptional situations in the manner most suitable for their application.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

TWO TRUTHS & A LIE

Creating Your Own `Exception` Classes

1. You must create your own `Exception` classes for your programs to be considered truly object oriented.
2. To create your own throwable `Exception` class, you should extend the `Exception` class.
3. The `Exception` class contains four constructors, including a default constructor and one that requires a `String` that contains the message that can be returned by the `getMessage()` method.

The false statement is #1. You are not required to throw exceptions in object-oriented programs. However, Java does provide many built-in categories of `Exceptions` that you can use, and you also can create your own `Exception` classes.

Using Assertions

In Chapter 1, you learned that you might inadvertently create syntax or logical errors when you write a program. Syntax errors are mistakes using the Java language; they are compile-time errors that prevent a program from compiling and creating an executable file with a .class extension.

In Chapter 1, you also learned that a program might contain logical errors even though it is free from syntax errors. Some logical errors cause runtime errors, or errors that cause a program to terminate. In this chapter, you learned how to use exceptions to handle many of these kinds of errors.

Some logical errors do not cause a program to terminate but nevertheless produce incorrect results. For example, if a payroll program should determine gross pay by multiplying hours worked by hourly pay rate, but you inadvertently divide the numbers, no runtime error occurs and no exception is thrown, but the output is wrong. An **assertion** is a Java language feature that can help you detect such logic errors and debug a program. You use an **`assert` statement** to create an assertion; when you use an `assert` statement, you state a condition that should be true, and Java throws an `AssertionError` when it is not.

The syntax of an `assert` statement is:

```
assert booleanExpression : optionalErrorMessage
```

The Boolean expression in the `assert` statement should always be `true` if the program is working correctly. The `optionalErrorMessage` is displayed if the `booleanExpression` is `false`.

Not For Sale

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

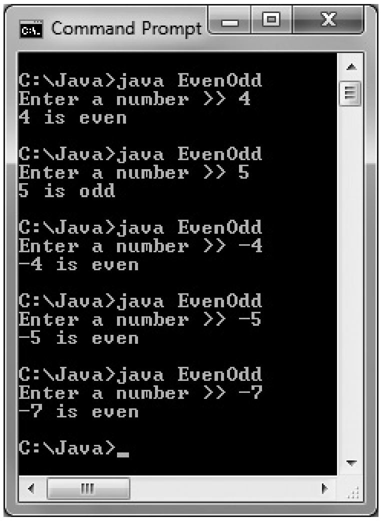
Figure 12-41 contains an application that prompts a user for a number and passes it to a method that determines whether a value is even. Within the `isEven()` method, the remainder is taken when the passed parameter is divided by 2. If the remainder after dividing by 2 is 1, `result` is set to `false`. For example, 1, 3, and 5 all are odd, and all result in a value of 1 when % 2 is applied to them. If the remainder after dividing by 2 is not 1, `result` is set to `true`. For example, 2, 4, and 6 all are even, and all have a 0 remainder when % 2 is applied to them.

```
import java.util.Scanner;
public class EvenOdd
{
   public static void main(String[] args)
   {
       Scanner input = new Scanner(System.in);
       int number;
       System.out.print("Enter a number >> ");
       number = input.nextInt();
       if(isEven(number))
          System.out.println(number + " is even");
       else
          System.out.println(number + " is odd");
   }
   public static boolean isEven(int number)
   {
      boolean result;
      if(number % 2 == 1)
         result = false;
      else
         result = true;
      return result;
   }
}
```

Figure 12-41 The flawed `EvenOdd` program without an assertion

Figure 12-42 shows several executions of the application in Figure 12-41. The output seems correct until the last two executions. The values –5 and –7 are classified as even although they are odd. An assertion might help you to debug this application.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.



```
C:\Java>java EvenOdd
Enter a number >> 4
4 is even

C:\Java>java EvenOdd
Enter a number >> 5
5 is odd

C:\Java>java EvenOdd
Enter a number >> -4
-4 is even

C:\Java>java EvenOdd
Enter a number >> -5
-5 is even

C:\Java>java EvenOdd
Enter a number >> -7
-7 is even

C:\Java>_
```

Figure 12-42 Typical executions of the `EvenOdd` application

Figure 12-43 contains a new version of the `isEven()` method to which the shaded `assert` statement has been added. The statement asserts that when the remainder of a number divided by 2 is not 1, it must be 0. If the expression is not true, a message is created using the values of both `number` and its remainder after dividing by 2.

```
public static boolean isEven(int number)
{
   boolean result;
   if(number % 2 == 1)
      result = false;
   else
   {
      result = true;
      assert number % 2 == 0 : number + " % 2 is " + number % 2;
   }
   return result;
}
```

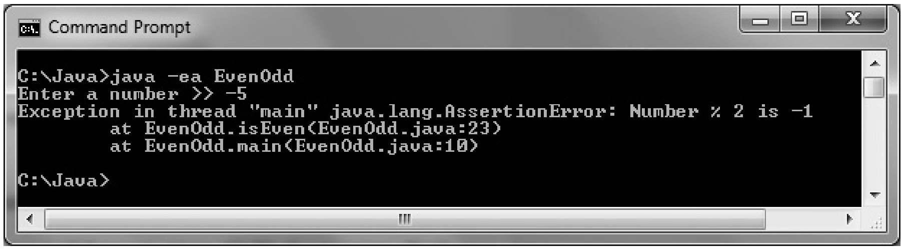
Figure 12-43 The flawed `isEven()` method with an assertion

If you add the assertion shown in Figure 12-43 and then compile and execute the program in the usual way, you get the same incorrect output as in Figure 12-42. To enable the assertion, you must use the `-ea` option when you execute the program; *ea* stands for *enable assertion*. Figure 12-44 shows the command prompt with an execution that uses the `-ea` option.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale



Command Prompt

```
C:\Java>java -ea EvenOdd
Enter a number >> -5
Exception in thread "main" java.lang.AssertionError: Number % 2 is -1
        at EvenOdd.isEven(EvenOdd.java:23)
        at EvenOdd.main(EvenOdd.java:10)

C:\Java>
```

Figure 12-44 Executing an application using the enable assertion option

When the `EvenOdd` program executes and the user enters –5, the program displays the messages in Figure 12-44 instead of displaying incorrect output. You can see from the message that an `AssertionError` was thrown and that the value of –5 % 2 is –1, not 1 as you had assumed. The remainder operator results in a negative value when one of its operands is negative, making the output in this program incorrect.

When the programmer sees that –5 % 2 is –1, the reasonable course of action is to return to the source code and change the logic.

Several adjustments are possible:

- The programmer might decide to convert the parameter to the `isEven()` method to its absolute value before using the remainder operator, as in the following:

  ```
  number = Math.abs(number);
  ```

- Another option would be to change the `if` statement to test for even values by comparing `number % 2` to 0 first, as follows:

  ```
  if(number % 2 == 0)
     result = true;
  else
     result = false;
  ```

  Then values of both 1 and –1 would be classified as not even.

- Other options might include displaying an error message when negative values are encountered, reversing the result values of `true` and `false` when the parameter is negative, or throwing an exception.

An experienced programmer might have found the error in the original `EvenOdd` application without using an assertion. For example, the programmer might have previously used the remainder operator with a negative operand, remembered that the result might be negative, and changed the code accordingly. Alternatively, the programmer could have inserted statements to display values at strategic points in the program. However, after the mistake is found and fixed, any extra display statements should be removed when the final product is ready for distribution to users. In contrast, any `assert` statements can be left in place, and if the user does not use the `-ea` option when running the program, the user will see no evidence

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

that the `assert` statements exist. Placing `assert` statements in key program locations can reduce development and debugging time.

You do not want to use assertions to check for every type of error that could occur in a program. For example, if you want to ensure that a user enters numeric data, you should use exception-handling techniques that provide the means for your program to recover from the mistake. If you want to ensure that the data falls within a specific range, you should use a decision or a loop. Assertions are meant to be helpful in the development stage of a program, not when it is in production and in the hands of users.

TWO TRUTHS & A LIE

Using Assertions

1. All logical errors cause a program to terminate, and they should be handled by throwing and catching exceptions.
2. The Boolean expression in an `assert` statement should always be `true` if the program is working correctly.
3. To enable an assertion, you must use the `-ea` option when you execute the program.

The false statement is #1. Many logical errors do not cause program termination—they simply produce incorrect results.

You Do It

Creating a Class That Automatically Throws Exceptions

Next, you create a class that contains two methods that throw exceptions but don't catch them. The `PickMenu` class allows restaurant customers to choose from a dinner menu. Before you create `PickMenu`, you will create the `Menu` class, which lists dinner choices and allows a user to make a selection.

1. Open a new file in your text editor, and then enter the following import statement, class header, and opening curly brace for the `Menu` class:

```
import javax.swing.*;
public class Menu
{
```

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

(continued)

2. Type the following `String` array for three entree choices. Also include a `String` to build the menu that you will display and an integer to hold the numeric equivalent of the selection.

```
private String[] entreeChoice = {"Rosemary Chicken",
   "Beef Wellington", "Maine Lobster"};
private String menu = "";
private int choice;
```

3. Add the `displayMenu()` method, which lists each entree option with a corresponding number the customer can type to make a selection. Even though the allowable `entreeChoice` array subscripts are 0, 1, and 2, most users would expect to type 1, 2, or 3. So, you code `x + 1` rather than `x` as the number in the prompt. After the user enters a selection, convert it to an integer. Return the `String` that corresponds to the user's menu selection—the one with the subscript that is 1 less than the entered value. After the closing curly brace for the `displayMenu()` method, add the closing curly brace for the class.

```
   public String displayMenu()
   {
      for(int x = 0; x < entreeChoice.length; ++x)
      {
         menu = menu + "\n" + (x + 1) + " for " +
            entreeChoice[x];
      }
      String input = JOptionPane.showInputDialog(null,
         "Type your selection, then press Enter." + menu);
      choice = Integer.parseInt(input);
      return(entreeChoice[choice - 1]);
   }
}
```

The curly braces are not necessary in the `for` loop of the `displayMenu()` method because the loop contains only one statement. However, in a later exercise, you will add another statement within this block.

4. Examine the code within the `displayMenu()` method. Consider the exceptions that might occur. The user might not type an integer, so the `parseInt()` method can fail, and even if the user does type an integer, it might not be in the range allowed to access the `entreeChoice` array. Therefore, the `displayMenu()` method, like most methods in which you rely on the user to enter data, might throw exceptions that you can anticipate. (Of course, any method might throw an unanticipated exception.)

5. Save the file as **Menu.java**, and compile the class using the **javac** command.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

Creating a Class That Passes on an `Exception` Object

Next, you create the `PickMenu` class, which lets a customer choose from the available dinner entree options. The `PickMenu` class declares a `Menu` and a `String` named `guestChoice` that holds the name of the entree the customer selects.

To enable the `PickMenu` class to operate with different kinds of `Menus` in the future, you will pass a `Menu` to `PickMenu`'s constructor. This technique provides two advantages: First, when the menu options change, you can alter the contents of the Menu.java file without changing any of the code in programs that use `Menu`. Second, you can extend `Menu`, perhaps to `VegetarianMenu`, `LowSaltMenu`, or `KosherMenu`, and still use the existing `PickMenu` class. When you pass any `Menu` or `Menu` subclass into the `PickMenu` constructor, the correct customer options appear.

The `PickMenu` class is unlikely to directly generate any exceptions because it does not request user input. (Keep in mind that any class might generate an exception for such uncontrollable events as the system not having enough memory available.) However, `PickMenu` declares a `Menu` object; the `Menu` class, because it relies on user input, is likely to generate an exception.

1. Open a new file in your text editor, and then add the following first few lines of the `PickMenu` class with its data fields (a `Menu` and a `String` that reflect the customer's choice):

```
import javax.swing.*;
public class PickMenu
{
  private Menu briefMenu;
  private String guestChoice = new String();
```

2. Enter the following `PickMenu` constructor, which receives an argument representing a `Menu`. The constructor assigns the `Menu` that is the argument to the local `Menu` and then calls the `setGuestChoice()` method, which prompts the user to select from the available menu. The `PickMenu()` constructor might throw an exception because it calls `setGuestChoice()`, which calls `displayMenu()`, a method that uses keyboard input and might throw an exception.

```
public PickMenu(Menu theMenu)
{
    briefMenu = theMenu;
    setGuestChoice();
}
```

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

(continued)

3. The following `setGuestChoice()` method displays the menu and reads keyboard data entry (so the method throws an exception). It also displays instructions and then retrieves the user's selection.

```
public void setGuestChoice()
{
  JOptionPane.showMessageDialog(null,
    "Choose from the following menu:");
  guestChoice = briefMenu.displayMenu();
}
```

4. Add the following `getGuestChoice()` method that returns a guest's `String` selection from the `PickMenu` class. Also, add a closing curly brace for the class.

```
    public String getGuestChoice()
    {
       return(guestChoice);
    }
}
```

5. Save the file as **PickMenu.java**, and compile it using the **javac** command.

Creating an Application That Can Catch Exceptions

You have created a `Menu` class that simply holds a list of food items, displays itself, and allows the user to make a selection. You also created a `PickMenu` class with fields that hold a user's specific selection from a given menu and methods to get and set values for those fields. The `PickMenu` class might throw exceptions, but it contains no methods that catch those exceptions. Next, you write an application that uses the `PickMenu` class. This application can catch exceptions that `PickMenu` throws.

1. Open a new file in your text editor, and start entering the following `PlanMenu` class, which has just one method—a `main()` method:

```
import javax.swing.*;
public class PlanMenu
{
  public static void main(String[] args)
  {
```

2. Construct the following `Menu` named `briefMenu`, and declare a `PickMenu` object that you name `entree`. You do not want to construct a `PickMenu` object yet because you want to be able to catch the exception that the `PickMenu` constructor might throw. Therefore, you want to wait and construct the `PickMenu` object within a `try` block. For now, you just declare `entree` and assign it `null`. Also, you declare a `String` that holds the customer's menu selection.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

```
Menu briefMenu = new Menu();
PickMenu entree = null;
String guestChoice = new String();
```

3. Write the following `try` block that constructs a `PickMenu` item. If the construction is successful, the next statement assigns a selection to the `entree` object. Because `entree` is a `PickMenu` object, it has access to the `getGuestChoice()` method in the `PickMenu` class, and you can assign the method's returned value to the `guestChoice String`.

```
try
{
   PickMenu selection = new PickMenu(briefMenu);
   entree = selection;
   guestChoice = entree.getGuestChoice();
}
```

4. The `catch` block must immediately follow the `try` block. When the `try` block fails, `guestChoice` will not have a valid value, so recover from the exception by assigning a value to `guestChoice` within the following `catch` block:

```
catch(Exception error)
{
  guestChoice = "an invalid selection";
}
```

5. After the `catch` block, the application continues. Use the following code to display the customer's choice at the end of the `PlanMenu` application, and then add closing curly braces for the `main()` method and the class:

```
         JOptionPane.showMessageDialog(null,
            "You chose " + guestChoice);
     }
}
```

6. Save the file as **PlanMenu.java**, and then compile and execute it. Read the instructions, click **OK**, choose an entree by typing its number from the menu, and click **OK** again. Confirm that the menu selection displayed is the one you chose, and click **OK** to dismiss the last dialog box. Figure 12-45 shows the first dialog box of instructions, the menu that appears, and the output when the user selects option 3.

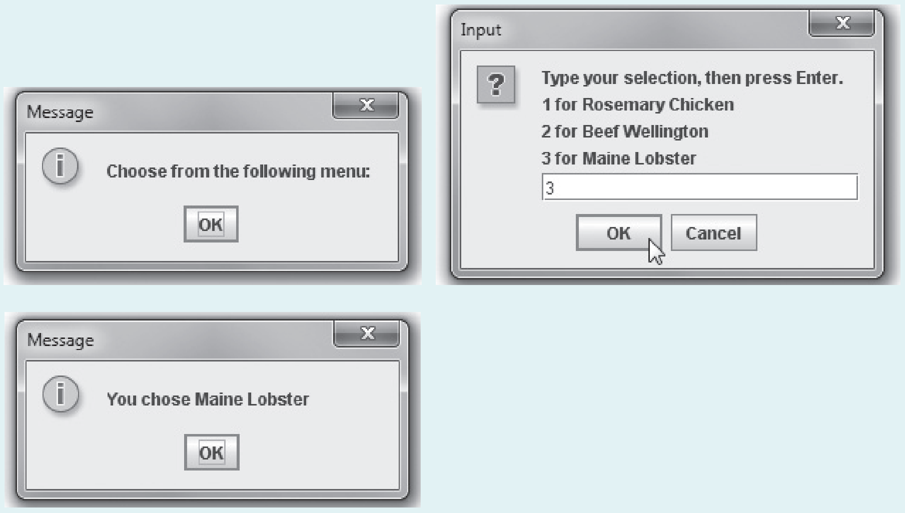
(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

(continued)


Message

Choose from the following menu:

Input

Type your selection, then press Enter.
1 for Rosemary Chicken
2 for Beef Wellington
3 for Maine Lobster

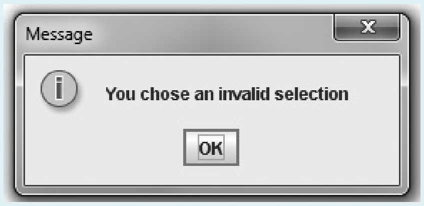
3

Message

You chose Maine Lobster

Figure 12-45 Typical execution of the `PlanMenu` application

7. The `PlanMenu` application works well when you enter a valid menu selection. One way that you can force an exception is to enter an invalid menu selection at the prompt. Run the `PlanMenu` application again, and type **4**, **A**, or any invalid value at the prompt. Entering *4* produces an `ArrayIndexOutOfBoundsException`, and entering *A* produces a `NumberFormatException`. If the program lacked the `try…catch` pair, either entry would halt the program. However, because the `setGuestChoice()` method in the `PickMenu` class throws the exception and the `PlanMenu` application catches it, `guestChoice` takes on the value "an invalid selection" and the application ends smoothly, as shown in Figure 12-46.


Message

You chose an invalid selection

Figure 12-46 Exceptional execution of the `PlanMenu` application

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

Extending a Class That Throws Exceptions

An advantage to using object-oriented exception handling techniques is that you gain the ability to handle error conditions differently within each program you write. Next, you extend the `Menu` class to create a class named `VegetarianMenu`. Subsequently, when you write an application that uses `PickMenu` with a `VegetarianMenu` object, you can deal with any thrown exception differently than when you wrote the `PlanMenu` application.

1. Open the **Menu.java** file in your text editor, and change the access specifier for the `entreeChoice` array from `private` to `protected`. That way, when you extend the class, the derived class will have access to the array. Save the file, and recompile it using the **javac** command.

2. Open a new file in your text editor, and then type the following class header for the `VegetarianMenu` class that extends `Menu`:

```
public class VegetarianMenu extends Menu
{
```

3. Provide new menu choices for the `VegetarianMenu` as follows:

```
String[] vegEntreeChoice = {"Spinach Lasagna",
   "Cheese Enchiladas", "Fruit Plate"};
```

4. Add the following constructor that calls the superclass constructor and assigns each vegetarian selection to the `Menu` superclass `entreeChoice` array, and then add the closing curly brace for the class:

```
   public VegetarianMenu()
   {
      super();
      for(int x = 0; x < vegEntreeChoice.length; ++x)
         entreeChoice[x] = vegEntreeChoice[x];
   }
}
```

5. Save the class as **VegetarianMenu.java**, and then compile it.

6. Now write an application that uses `VegetarianMenu`. You could write any program, but for demonstration purposes, you can simply modify PlanMenu.java. Open the **PlanMenu.java** file in your text editor, then immediately save it as **PlanVegetarianMenu.java**.

7. Change the class name in the header to **`PlanVegetarianMenu`**.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

(continued)

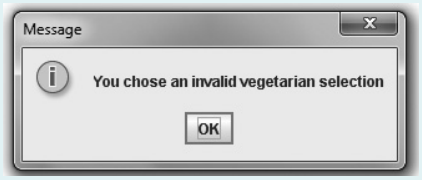
8. Change the first statement within the `main()` method as follows so it declares a `VegetarianMenu` instead of a `Menu`:

```
VegetarianMenu briefMenu = new VegetarianMenu();
```

9. Change the `guestChoice` assignment statement in the `catch` block as follows so it is specific to the program that uses the `VegetarianMenu`:

```
guestChoice = "an invalid vegetarian selection";
```

10. Save the file, compile it, and run the application. When you see the vegetarian menu, enter a valid selection and confirm that the program works correctly. Run the application again, and enter an invalid selection. The error message shown in Figure 12-47 identifies your invalid entry as "an invalid vegetarian selection". Remember that you did not change the `PickMenu` class. Your new `PlanVegetarianMenu` application uses the `PickMenu` class that you wrote and compiled before a `VegetarianMenu` ever existed. However, because `PickMenu` throws uncaught exceptions, you can handle those exceptions as you see fit in any new applications in which you catch them. Click **OK** to end the application.



You chose an invalid vegetarian selection

Figure 12-47 Output of the `PlanVegetarianMenu` application when the user makes an invalid selection

Creating an `Exception` *Class*

Besides using built-in classes that derive from `Exception`, such as `NumberFormatException` and `IndexOutOfBoundsException`, you can create your own `Exception` classes. For example, suppose that although you have asked a user to type a number representing a menu selection, you realize that some users might mistakenly type the initial letter of an option, such as *R* for *Rosemary Chicken*. Although the user has made an error, you want to treat this type of error more leniently than other errors, such as typing a letter that has no discernable connection to the presented menu. In the next section, you create a `MenuException` class that you can use with the `Menu` class to represent a specific type of error.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

1. Open a new file in your text editor, and enter the `MenuException` class. The class extends `Exception`. Its constructor requires a `String` argument, which is passed to the parent class to be used as a return value for the `getMessage()` method.

```
public class MenuException extends Exception
{
   public MenuException(String choice)
   {
      super(choice);
   }
}
```

2. Save the file as **MenuException.java**, and compile it.

Using an `Exception` You Created

Next, you modify the `Menu`, `PickMenu`, and `PlanMenu` classes to demonstrate how to use a `MenuException` object.

1. Open the **Menu** class in your text editor, and immediately save the file as **Menu2.java**.
2. Change the class name to **`Menu2`**.
3. At the end of the list of class data fields, add an array of characters that can hold the first letter of each of the entrees in the menu.

```
protected char initial[] = new char[entreeChoice.length];
```

4. At the end of the method header for the `displayMenu()` class, add the following clause:

```
throws MenuException
```

   You add this clause because you are going to add code that throws such an exception.

5. Within the `displayMenu()` method, just before the closing curly brace of the `for` loop that builds the menu `String`, add a statement that takes the first character of each `entreeChoice` and stores it in a corresponding element of the `initial` array. At the end of the `for` loop, the `initial` array holds the first character of each available entree.

```
initial[x] = entreeChoice[x].charAt(0);
```

6. After displaying the `JOptionPane` dialog box that displays the menu and receives the user's input, add a loop that compares the first letter of the user's choice to each of the initials of valid menu options. If a match is found, throw a new

(continues)

Not For Sale

Not For Sale

(continued)

instance of the `MenuException` class that uses the corresponding entree as its `String` argument. In other words, when this thrown `MenuException` is caught by another method, the assumed entree is the `String` returned by the `getMessage()` method. By placing this test before the call to `parseInt()`, you cause entries of *R*, *B*, or *M* to throw a `MenuException` before they can cause a `NumberFormatException`.

```
for(int y = 0; y < entreeChoice.length; ++y)
   if(input.charAt(0) == initial[y])
      throw (new MenuException(entreeChoice[y]));
```

7. Compare your new class with Figure 12-48, in which all of the changes to the `Menu` class are shaded.

```
import javax.swing.*;
public class Menu2
{
   protected String[] entreeChoice = {"Rosemary Chicken",
      "Beef Wellington", "Maine Lobster"};
   private String menu = "";
   private int choice;
   protected char initial[] = new char[entreeChoice.length];
   public String displayMenu() throws MenuException
   {
      for(int x = 0; x < entreeChoice.length; ++x)
      {
         menu = menu + "\n" + (x + 1) + " for " +
            entreeChoice[x];
         initial[x] = entreeChoice[x].charAt(0);
      }
      String input = JOptionPane.showInputDialog(null,
         "Type your selection, then press Enter." + menu);
      for(int y = 0; y < entreeChoice.length; ++y)
         if(input.charAt(0) == initial[y])
            throw(new MenuException(entreeChoice[y]));
      choice = Integer.parseInt(input);
      return(entreeChoice[choice - 1]);
   }
}
```

Figure 12-48 The Menu2 class

8. Save the class, and compile it.

9. Open the **PickMenu** file in your text editor, and immediately save it as **PickMenu2.java**.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

10. Change the class name to **PickMenu2**, and change the declaration of the Menu object to a **Menu2** object. Change the constructor name to **PickMenu2** and its argument to type **Menu2**. Also add a throws clause to the PickMenu2 constructor header so that it throws a MenuException. This constructor does not throw an exception directly, but it calls the setGuestChoice() method, which calls the displayMenu() method, which throws a MenuException.

11. Add the following throws clause to the setGuestChoice() method header:

    throws MenuException

12. Compare your modifications to the PickMenu2 class in Figure 12-49, in which the changes from the PickMenu class are shaded. Save your file, and compile it.

```
import javax.swing.*;
public class PickMenu2
{
   private Menu2 briefMenu;
   private String guestChoice = new String();
   public PickMenu2(Menu2 theMenu) throws MenuException
   {
      briefMenu = theMenu;
      setGuestChoice();
   }
   public void setGuestChoice() throws MenuException
   {
      JOptionPane.showMessageDialog(null,
         "Choose from the following menu:");
      guestChoice = briefMenu.displayMenu();
   }
   public String getGuestChoice()
   {
      return(guestChoice);
   }
}
```

Figure 12-49 The PickMenu2 class

13. Open the **PlanMenu.java** file in your text editor, and immediately save it as **PlanMenu2.java**.

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

(continued)

14. Change the class name to **PlanMenu2**. Within the main() method, declare a **Menu2** object and a **PickMenu2** reference instead of the current Menu object and PickMenu reference.

15. Within the try block, change both references of PickMenu to **PickMenu2**.

    Using Figure 12-50 as a reference, add a catch block after the try block and before the existing catch block. This catch block will catch any thrown MenuExceptions and display their messages. The message will be the name of a menu item, based on the initial the user entered. All other Exception objects, including NumberFormatExceptions and IndexOutOfBoundsExceptions, will fall through to the second catch block and be handled as before.

```
import javax.swing.*;
public class PlanMenu2
{
   public static void main(String[] args)
   {
      Menu2 briefMenu = new Menu2();
      PickMenu2 entree = null;
      String guestChoice = new String();
      try
      {
         PickMenu2 selection = new PickMenu2(briefMenu);
         entree = selection;
         guestChoice = entree.getGuestChoice();
      }
      catch(MenuException error)
      {
         guestChoice = error.getMessage();
      }
      catch(Exception error)
      {
         guestChoice = "an invalid selection";
      }
      JOptionPane.showMessageDialog(null,
         "You chose " + guestChoice);
   }
}
```

Figure 12-50 The PlanMenu2 class

(continues)

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

(continued)

16. Save the file, then compile and execute it several times. When you are asked to make a selection, try entering a valid number, an invalid number, an initial letter that is part of the menu, and a letter that is not one of the initial menu letters, and observe the results each time. Whether or not you enter a valid number, the application works as expected. Entering an invalid number still results in an error message. When you enter a letter or a string of letters, the application assumes your selection is valid if you enter the same initial letter, using the same case, as one of the menu options.

Don't Do It

- Don't forget that all the statements in a `try` block might not execute. If an exception is thrown, no statements after that point in the `try` block will execute.
- Don't forget that you might need a `nextLine()` method call after an attempt to read numeric data from the keyboard throws an exception.
- Don't forget that a variable declared in a `try` block goes out of scope at the end of the block.
- Don't forget that when a variable gets its usable value within a `try` block, you must ensure that it has a valid value before attempting to use it.
- Don't forget to place more specific `catch` blocks before more general ones.
- Don't forget to write a `throws` clause for a method that throws a checked exception but does not handle it.
- Don't forget to handle any checked exception thrown to your method either by writing a `catch` block or by listing it in your method's `throws` clause.

Key Terms

An **exception** is an unexpected or error condition.

Exception handling is an object-oriented technique for managing errors.

Runtime exceptions are unplanned exceptions that occur during a program's execution. The term is also used more specifically to describe members of the `RuntimeException` class.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

The **`Error` class** represents more serious errors than the `Exception` class—those from which your program usually cannot recover.

The **`Exception` class** comprises less serious errors than those from the `Error` class; the `Exception` class represents unusual conditions that arise while a program is running and from which the program can recover.

A **crash** is a premature, unexpected, and inelegant end to a program.

A **stack trace history list**, or more simply a **stack trace**, displays all the methods that were called during program execution.

The term **mission critical** refers to any process that is crucial to an organization.

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.

A **`try` block** is a block of code that might throw an exception that can be handled by a subsequent `catch` block.

A **`catch` block** is a segment of code that can handle an exception that might be thrown by the `try` block that precedes it.

A **`throw` statement** is one that sends an exception out of a block or a method so it can be handled elsewhere.

A **`finally` block** is a block of code that holds statements that must execute at the end of a `try...catch` sequence, whether or not an exception was thrown.

Exception specification is the practice of using the keyword `throws` followed by an `Exception` type in the method header. If a method throws a checked `Exception` object that it will not catch but will be caught by a different method, you must use an exception specification.

Unchecked exceptions are those from which an executing program cannot reasonably be expected to recover.

Checked exceptions are those that a programmer should plan for and from which a program should be able to recover.

The **catch or specify requirement** is the Java rule that checked exceptions require catching or declaration.

Syntactic sugar is a term to describe aspects of a computer language that make it “sweeter,” or easier, for programmers to use.

Syntactic salt describes a language feature designed to make it harder to write bad code.

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

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

An **assertion** is a Java language feature that can help you detect logic errors and debug a program.

An **`assert` statement** creates an assertion.

Chapter Summary

- An exception is an unexpected or error condition. Exception handling is the name for the object-oriented techniques that manage such errors. In Java, the two basic classes of errors are `Error` and `Exception`; both descend from the `Throwable` class.
- In object-oriented terminology, a `try` block holds code that might cause an error and throw an exception, and a `catch` block processes the error.
- You can place as many statements as you need within a `try` block, and you can catch as many exceptions as you want. If you try 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. When a program contains multiple `catch` blocks, the first matching `catch` block executes, and each remaining `catch` block is bypassed.
- When you have actions you must perform at the end of a `try`...`catch` sequence, you can use a `finally` block that executes regardless of whether the preceding `try` block identifies an exception. Usually, you use a `finally` block to perform cleanup tasks.
- Besides clarity, an advantage to object-oriented exception handling is the flexibility it allows in the handling of error situations. Each calling application might need to handle the same error differently, depending on its purpose.
- When you write a method that might throw a checked exception that is not caught within the method, you must type the clause `throws <name>Exception` after the method header to indicate the type of `Exception` that might be thrown. Methods in which you explicitly throw a checked exception require a catch or a declaration.
- The memory location known as the call stack is where the computer stores the list of method locations to which the system must return. When you catch an exception, you can call `printStackTrace()` to display a list of methods in the call stack so you can determine the location of the exception.
- Java provides over 40 categories of `Exceptions` that you can use in your programs. However, Java's creators could not predict every condition that might be an `Exception` in your applications, so Java also allows you to create your own `Exceptions`. To create your own throwable `Exception` class, you must extend a subclass of `Throwable`.
- An assertion is a Java language feature that can help you detect logic errors and debug a program. When you use an assertion, you state a condition that should be true, and Java throws an `AssertionError` when it is not.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

Review Questions

1. In object-oriented programming terminology, an unexpected or error condition is a(n) ____________.

    a. anomaly
    b. aberration
    c. deviation
    d. exception

2. All Java `Exceptions` are ____________.

    a. `Errors`
    b. `RuntimeExceptions`
    c. `Throwables`
    d. `Omissions`

3. Which of the following statements is true?

    a. `Exceptions` are more serious than `Errors`.
    b. `Errors` are more serious than `Exceptions`.
    c. `Errors` and `Exceptions` are equally serious.
    d. `Exceptions` and `Errors` are the same thing.

4. The method that ends the current application and returns control to the operating system is ____________.

    a. `System.end()`
    b. `System.done()`
    c. `System.exit()`
    d. `System.abort()`

5. In object-oriented terminology, you ____________ a procedure that might not complete correctly.

    a. try
    b. catch
    c. handle
    d. encapsulate

6. A method that detects an error condition or `Exception` ____________ an `Exception`.

    a. throws
    b. catches
    c. handles
    d. encapsulates

7. A `try` block includes all of the following elements except ____________.

    a. the keyword `try`
    b. the keyword `catch`
    c. curly braces
    d. statements that might cause `Exceptions`

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

8. The segment of code that handles or takes appropriate action following an exception is a __________ block.

   a. `try`
   b. `catch`
   c. `throws`
   d. `handles`

9. You __________ within a `try` block.

   a. must place only a single statement
   b. can place any number of statements
   c. must place at least two statements
   d. must place a `catch` block

10. If you include three statements in a `try` block and follow the block with three `catch` blocks, and the second statement in the `try` block throws an `Exception`, then __________.

    a. the first `catch` block executes
    b. the first two `catch` blocks execute
    c. only the second `catch` block executes
    d. the first matching `catch` block executes

11. When a `try` block does not generate an `Exception` and you have included multiple `catch` blocks, __________.

    a. they all execute
    b. only the first one executes
    c. only the first matching one executes
    d. no `catch` blocks execute

12. The `catch` block that begins `catch(Exception e)` can catch `Exceptions` of type __________.

    a. `IOException`
    b. `ArithmeticException`
    c. both of the above
    d. none of the above

13. The code within a `finally` block executes when the `try` block __________.

    a. identifies one or more `Exceptions`
    b. does not identify any `Exceptions`
    c. either a or b
    d. neither a nor b

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

14. An advantage to using a `try...catch` block is that exceptional events are ___________.

    a. eliminated
    b. reduced
    c. integrated with regular events
    d. isolated from regular events

15. Which methods can throw an `Exception`?

    a. methods with a `throws` clause
    b. methods with a `catch` block
    c. methods with both a `throws` clause and a `catch` block
    d. any method

16. A method can ___________.

    a. check for errors but not handle them
    b. handle errors but not check for them
    c. either of the above
    d. neither of the above

17. Which of the following is least important to know if you want to be able to use a method to its full potential?

    a. the method's return type
    b. the type of arguments the method requires
    c. the number of statements within the method
    d. the type of `Exceptions` the method throws

18. The memory location where the computer stores the list of method locations to which the system must return is known as the ___________.

    a. registry
    b. call stack
    c. chronicle
    d. archive

19. You can get a list of the methods through which an `Exception` has traveled by using the ___________ method.

    a. `getMessage()`
    b. `callStack()`
    c. `getPath()`
    d. `printStackTrace()`

20. A(n) ___________ is a statement used in testing programs that should be true; if it is not true, an `Exception` is thrown.

    a. assertion
    b. throwable
    c. verification
    d. declaration

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Exercises

Programming Exercises

1. Write an application named `BadSubscriptCaught` in which you declare an array of 10 first names. Write a `try` block in which you prompt the user for an integer and display the name in the requested position. Create a `catch` block that catches the potential `ArrayIndexOutOfBoundsException` thrown when the user enters a number that is out of range. The `catch` block should also display an error message. Save the file as **BadSubscriptCaught.java**.

2. The `Double.parseDouble()` method requires a `String` argument, but it fails if the `String` cannot be converted to a floating-point number. Write an application in which you try accepting a `double` input from a user and catch a `NumberFormatException` if one is thrown. The `catch` block forces the number to 0 and displays an appropriate error message. Following the `catch` block, display the number. Save the file as **TryToParseDouble.java**.

3. In Chapter 9, you wrote a program named `MeanMedian` that allows a user to enter five integers and then displays the values, their mean, and their median. Now, modify the program to throw an exception if an entered value is not an integer. If an exception is thrown, display an appropriate message and allow the user to reenter the value. Save the file as **MeanMedianHandleException.java**.

4. Write an application that prompts the user to enter a number to use as an array size, and then attempt to declare an array using the entered size. If the array is created successfully, display an appropriate message. Java generates a `NegativeArraySizeException` if you attempt to create an array with a negative size, and it creates a `NumberFormatException` if you attempt to create an array using a nonnumeric value for the size. Use a `catch` block that executes if the array size is nonnumeric or negative, displaying a message that indicates the array was not created. Save the file as **NegativeArray.java**.

5. Write an application that throws and catches an `ArithmeticException` when you attempt to take the square root of a negative value. Prompt the user for an input value and try the `Math.sqrt()` method on it. The application either displays the square root or catches the thrown `Exception` and displays an appropriate message. Save the file as **SqrtException.java**.

6. Create a `ProductException` class whose constructor receives a `String` that consists of a product number and price. Save the file as **ProductException.java**. Create a `Product` class with two fields, `productNum` and `price`. The `Product` constructor requires values for both fields. Upon construction, throw a `ProductException` if the product number does not consist of three digits, if the price is less than $0.01, or if the price is over $1,000. Save the class as **Product.java**. Write an application that establishes at least four `Product` objects with valid and

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

invalid values. Display an appropriate message when a `Product` object is created successfully and when one is not. Save the file as **ThrowProductException.java**.

7. a. Create an `IceCreamConeException` class whose constructor receives a `String` that consists of an ice cream cone's flavor and an integer representing the number of scoops in the `IceCreamCone`. Pass this `String` to the `IceCreamConeException`'s parent so it can be used in a `getMessage()` call. Save the class as **IceCreamConeException.java**. Create an `IceCreamCone` class with two fields—`flavor` and `scoops`. The `IceCreamCone` constructor calls two data-entry methods—`setFlavor()` and `setScoops()`. The `setScoops()` method throws an `IceCreamConeException` when the scoop quantity exceeds three. Save the class as **IceCreamCone.java**. Write an application that establishes several `IceCreamCone` objects and handles the `Exceptions`. Save the file as **ThrowIceCream.java**.

   b. Create an `IceCreamCone2` class in which you modify the `IceCreamCone setFlavor()` method to ensure that the user enters a valid flavor. Allow at least four flavors of your choice. If the user's entry does not match a valid flavor, throw an `IceCreamConeException`. Write an application that establishes several `IceCreamCone` objects and demonstrates the handling of the new `Exception`. Save the new class file as **IceCreamCone2.java**, and save the new application file as **ThrowIceCream2.java**.

8. Write an application that displays a series of at least five student ID numbers (that you have stored in an array) and asks the user to enter a numeric test score for the student. Create a `ScoreException` class, and throw a `ScoreException` for the class if the user does not enter a valid score (less than or equal to 100). Catch the `ScoreException`, and then display an appropriate message. In addition, store a *0* for the student's score. At the end of the application, display all the student IDs and scores. Save the files as **ScoreException.java** and **TestScore.java**.

9. Write an application that displays a series of at least 10 student ID numbers (that you have stored in an array) and asks the user to enter a test letter grade for the student. Create an `Exception` class named `GradeException` that contains a `static public` array of valid grade letters ('A', 'B', 'C', 'D', 'F', and 'I') you can use to determine whether a grade entered from the application is valid. In your application, throw a `GradeException` if the user does not enter a valid letter grade. Catch the `GradeException`, and then display an appropriate message. In addition, store an 'I' (for Incomplete) for any student for whom an exception is caught. At the end of the application, display all the student IDs and grades. Save the files as **GradeException.java** and **TestGrade.java**.

10. Create a `DataEntryException` class whose `getMessage()` method returns information about invalid integer data. Write a program named `GetIDAndAge` that continually prompts the user for an ID number and an age until a terminal *0* is entered for both. Throw a `DataEntryException` if the ID is not in the range of

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

valid ID numbers (0 through 999), or if the age is not in the range of valid ages (0 through 119). Catch any `DataEntryException` or `InputMismatchException` that is thrown, and display an appropriate message. Save the files as **DataEntryException.java** and **GetIDAndAge.java**.

11. A company accepts user orders by part numbers interactively. Users might make the following errors as they enter data:

    - The part number is not numeric.
    - The quantity is not numeric.
    - The part number is too low (less than 0).
    - The part number is too high (more than 999).
    - The quantity ordered is too low (less than 1).
    - The quantity ordered is too high (more than 5,000).

    Create a class that stores an array of usable error messages; save the file as **DataMessages.java**. Create a `DataException` class; each object of this class will store one of the messages. Save the file as **DataException.java**. Create an application that prompts the user for a part number and quantity. Allow for the possibility of nonnumeric entries as well as out-of-range entries, and display the appropriate message when an error occurs. If no error occurs, display the message “Valid entry”. Save the program as **PartAndQuantityEntry.java**.

12. A company accepts user orders for its products interactively. Users might make the following errors as they enter data:

    - The item number ordered is not numeric.
    - The quantity is not numeric.
    - The item number is too low (less than 0).
    - The item number is too high (more than 9999).
    - The quantity ordered is too low (less than 1).
    - The quantity ordered is too high (more than 12).
    - The item number is not a currently valid item.

    Although the company might expand in the future, its current inventory consists of the items listed in Table 12-1.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

Item Number	Price ($)
111	0.89
222	1.47
333	2.43
444	5.99

Table 12-1 Item numbers and prices

Create a class that stores an array of usable error messages; save the file as **OrderMessages.java**. Create an `OrderException` class that stores one of the messages; save the file as **OrderException.java**. Create an application that contains prompts for an item number and quantity. Allow for the possibility of nonnumeric entries as well as out-of-range entries and entries that do not match any of the currently available item numbers. The program should display an appropriate message if an error has occurred. If no errors exist in the entered data, compute the user's total amount due (quantity times price each) and display it. Save the program as **PlaceAnOrder.java**.

13. a. Gadgets by Mail sells many interesting items through its catalogs. Write an application that prompts the user for order details, including item numbers and quantity of each item ordered, based on the available items shown in Table 12-2.

Item #	Description	Price ($)
101	Electric hand warmer	12.99
124	Battery-operated plant waterer	7.55
256	Gerbil trimmer	9.99
512	Talking bookmark	6.89

Table 12-2 Items offered by Gadgets by Mail

The shipping and handling fee for an order is based on the total order price, as shown in Table 12-3.

Price of Order ($)	Shipping and Handling ($)
0–24.99	5.55
25.00–49.99	8.55
50.00 or more	11.55

Table 12-3 Shipping and handling fees charged by Gadgets by Mail

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Create the following classes:

- `Gadget`, which contains an item number, description, and price for a gadget; a constructor that sets all the fields; and get methods to retrieve the field values.
- `Order`, which contains an order number, customer name, and address (assume you need just a street address, not city, state, and zip code); a list of item numbers ordered (up to four); the total price of all items ordered; and a shipping and handling fee for the order. Include a constructor to set the field values and get methods to retrieve the field values.
- `GadgetOrderTaker`, which is an interactive application that takes four customer orders. The class contains an array of the four `Gadget` objects offered (from Table 12-2). The application prompts each user for a name and street address and assigns a unique order number to each customer, starting with 101. The application asks each user to enter an item number and quantity wanted. When the user enters 999 for the item number, the order is complete, and the next customer can enter data. Each customer can order up to four item numbers. When a customer's order is complete (the customer has entered 999 for an item number, or has ordered four different items), calculate the shipping and handling charges. After four customers have placed `Order`s, display each `Order`'s data, including the order number, the name and address of the customer, and the list of items ordered, including the item number, description, and price of each `Order`, the total price for the order, and the shipping and handling charge. The `GadgetOrderTaker` class handles all thrown `Exceptions` by displaying an explanatory message and ending the application.
- `OrderException`, which is an `Exception` that is created and thrown under any of the following conditions:
  - A customer attempts to order more than four different items.
  - A customer orders more than 100 of any item.
  - A customer enters an invalid item number.
- Also, catch the `Exception` generated by either of these conditions:
  - A customer enters a nonnumeric character as the item number.
  - A customer enters a nonnumeric character as the quantity.

Save the files as **Gadget.java**, **Order.java**, **GadgetOrderTaker.java**, and **OrderException.java**.

b. The `GadgetOrderTaker` class handles all thrown `Exceptions` by displaying an explanatory message and ending the application. Create a new application that handles all `Exceptions` by requiring the user to reenter the offending data. Save the file as **GadgetOrderTaker2.java**.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

Debugging Exercises

1. Each of the following files in the Chapter12 folder of your downloadable student files has syntax and/or logic 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 *Fix*. For example, DebugTwelve1.java will become FixDebugTwelve1.java. You will also use a file named DebugEmployeeIDException.java with the DebugTwelve4.java file.

   a. DebugTwelve1.java
   b. DebugTwelve2.java
   c. DebugTwelve3.java
   d. DebugTwelve4.java

Game Zone

1. In Chapter 1, you created a class called `RandomGuess`. In this game, the application generates a random number for a player to guess. In Chapter 5, you improved the application to display a message indicating whether the player's guess was correct, too high, or too low. In Chapter 6, you further improved the game by adding a loop that continually prompts the user to enter the correct value, if necessary. As written, the game should work as long as the player enters numeric guesses. However, if the player enters a letter or other nonnumeric character, the game throws an exception. Discover the type of `Exception` thrown, then improve the game by handling the exception so that the user is informed of the error and allowed to attempt to enter the correct data again. Save the file as **RandomGuess4.java**.

2. In Chapter 8, you created a `Quiz` class that contains an array of 10 multiple-choice questions to which the user was required to respond with an *A*, *B*, or *C*. At the time, you knew how to handle the user's response if an invalid character was entered. Rerun the program now to determine whether an exception is thrown if the user enters nothing—that is, the user just presses the Enter key without making an entry. If so, improve the program by catching the exception, displaying an appropriate error message, and presenting the same question to the user again. Save the file as **QuizWithExceptionsCaught.java**.

Case Problems

1. In Chapter 11, you created an interactive `StaffDinnerEvent` class that obtains all the data for a dinner event for Carly's Catering, including details about the staff members required to work at the event. Now, modify the class so that it becomes immune to user data entry errors by handling exceptions for each numeric entry. Each time the program requires numeric data—for example, for the number of guests, selected menu options, and staff members' salaries—continuously prompt

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

the user until the data entered is the correct type. Save the revised program as **StaffDinnerEvent.java**.

2. In Chapter 11, you created an interactive `RentalDemo` class that obtains all the data for four rentals from Sammy's Seashore Rentals, including details about the contract number, length of the rental, and equipment type. Now, modify the class so that it becomes immune to user data entry errors by handling exceptions for each numeric entry. Each time the program requires numeric data—for example, for the rental period—continuously prompt the user until the data entered is the correct type. Save the revised program as **RentalDemo.java**.

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.

Not For Sale

Not For Sale

© Cengage Learning. All rights reserved. No distribution allowed without express authorization.