CHAPTER

15
EXCEPTION HANDLING

In this chapter, you will:

- ♦ Learn about exceptions
- ♦ Try code and catch Exceptions
- ♦ Use the Exception getMessage() method
- ♦ Throw and catch multiple Exceptions
- ♦ Use the `finally` block
- ♦ Understand the limitations of traditional error handling
- ♦ Specify the Exceptions a method can throw
- ♦ Handle Exceptions uniquely with each `catch`
- ♦ Trace Exceptions through the call stack
- ♦ Create your own Exceptions

You're muttering to yourself at your desk at Event Handlers Incorporated.

"Anything wrong?" Lynn Greenbrier asks as she passes by.

"It's these errors!" you complain.

"Aren't you going overboard?" Lynn asks. "Everyone makes errors when they code programs."

"Oh, I expect typos and compiler errors while I'm developing my programs," you say, "but no matter how well I write my code, the user can still mess everything up by inputting bad data. The Event Planning Department told me that it has events planned for the 32nd day of the month and for negative five people. Even if my code is perfect, the user can enter mistakes."

"Then your code isn't perfect yet," Lynn says. "Besides writing programs that can handle ordinary situations, you must enable your programs to handle exceptions."

Learning About Exceptions

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

- You issue a command to read a file from a disk, but the file does not exist there.
- You write data to a disk, but the disk is full or unformatted.
- Your program asks for user input, but the user enters invalid data.
- The program attempts to divide a value by zero, access an array with a subscript that is too large, or calculate a value that is too large for the answer's variable type.

These errors are called exceptions because, presumably, they are not usual occurrences; they are "exceptional". The object-oriented techniques to manage such errors comprise the group of methods known as **exception handling**.

Providing for exceptions involves an oxymoron; you must expect the unexpected.

Like all other classes in the Java programming language, exceptions are Objects; their class name is Exception. In Java, there are two basic classes of errors: Error and Exception. Both of these classes descend from the Throwable class, as shown in Figure 15-1.

```
Throwable
¦
+--Exception
|   ¦
|   +--IOException
|   ¦
|   +--RuntimeException
|   |   ¦
|   |   +--ArithmeticException
|   |   ¦
|   |   +--ArrayIndexOutOfBoundsException
|   |   ¦
|   |   +--Others..
|   ¦
|   +--Others..
¦
+--ErrorException
    ¦
    +--OutOfMemoryException
    ¦
    +--InternalErrorException
    ¦
    +--Others..
```

Figure 15-1 Structure of the Exception class

With JDK version 1.4.0, Java acknowledges over 40 categories of Exceptions with unusual names such as ActivationException, AlreadyBoundException, AWTException, CloneNotSupportedException, PropertyVetoException, and UnsupportedFlavorException. See the *http://java.sun.com* Web site for more details about these and other Exceptions.

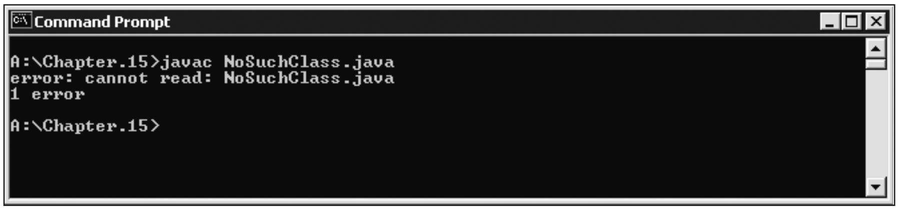
The **Error class** represents more serious errors from which your program usually cannot recover. These errors are the ones you probably have made in your own programs when you spelled a class name incorrectly or stored a required class in the wrong folder. When a program cannot locate a required class or your system runs out of memory, an Error condition occurs. Of course, a person can recover from such errors by spelling a name correctly, moving a file to the correct folder, or by physically installing more memory, but a program cannot recover from these kinds of mistakes.

The **Exception class** comprises less serious errors that represent 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.

When your code causes a program error, whether inadvertently or purposely, you can determine whether the type of Throwable object generated is an Error or an Exception by examining the message that you receive from Java after the error occurs. Next you will generate an unrecoverable Error.

To purposefully cause an unrecoverable Error:

1. Go to the command prompt for Chapter 15.
2. Type **`javac NoSuchClass.java`**, and then press **[Enter]**. Unless you have created a file named NoSuchClass.java in the current directory, the error message you receive should look like Figure 15-2.



```
A:\Chapter.15>javac NoSuchClass.java
error: cannot read: NoSuchClass.java
1 error

A:\Chapter.15>
```

Figure 15-2 Error message generated by missing file

Tip: Even though you generated an Error with an uppercase E (that is, you generated an instance of the Error class), the error message displays with a lowercase e.

The "cannot read" Error in Figure 15-2 must be remedied by typing a different class name, or storing a file with the name NoSuchClass.java in the Chapter.15 folder. In other words, a person must take action before the command can successfully execute; there is no program code you could write that would prevent the Error message. However, when you generate a recoverable Exception, which is less severe than an Error, you see a different type of message. An Exception message indicates that you could have prevented the message by using specific code within your program. Next you will generate a recoverable Exception.

To purposefully cause an Exception:

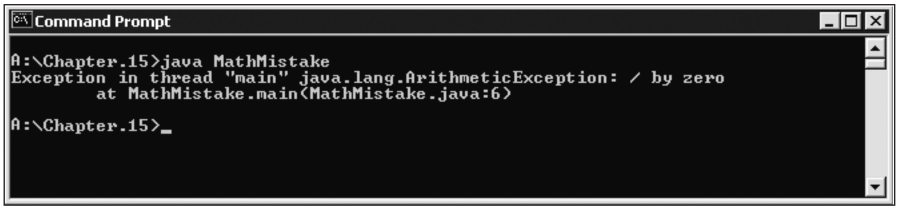
1. Open a new file in your text editor, and then type the following MathMistake class that attempts to divide by zero:

```
public class MathMistake
{
       public static void main(String[] args)
       {
              int num = 13, denom = 0, result;
              result = num / denom;
       }
}
```

Tip

You never should write a program that purposefully divides a value by zero. However, this situation certainly could occur if a variable used as a denominator gets its value as the result of user input.

2. Save the file as **MathMistake.java** in the Chapter.15 folder on your Student Disk, and then compile using the `javac MathMistake.java` command.
3. After the program compiles successfully, run the program using the `java MathMistake` command. Your result should look like Figure 15-3. You can see that the Exception is a java.lang. ArithmeticException, which is a subclass of Exception. You also get some information about the error ("/ by zero"), the method that generated the error (MathMistake.main), and the file and line number for the error (MathMistake.java, line 6); your line number might be different if you include comment lines at the beginning of your program.



```
A:\Chapter.15>java MathMistake
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MathMistake.main(MathMistake.java:6)

A:\Chapter.15>_
```

Figure 15-3 Exception generated by the MathMistake class

Just because an Exception occurs, you don't necessarily have to deal with it. In the MathMistake class, you simply let the offending program terminate. However, the program termination is abrupt and unforgiving. When a program divides two numbers (or even performs a less trivial task such as playing a game with the user or balancing a checkbook), the user might get annoyed if the program ends abruptly. However, if the program is used for air-traffic control or to monitor a patient's vital statistics during surgery, an abrupt conclusion could be disastrous. Object-oriented error-handling techniques provide more elegant (and safer) solutions.

Programmers had to deal with error conditions long before object-oriented methods were conceived. Probably the most-often-used error-handling solution has been to terminate the offending program. For example, you can change the main() method of the MathMistake class to halt the program before dividing by zero, as follows:

```
public class MathMistake
{
        public static void main(String[] args)
        {
               int num = 13, denom = 0, result;
               if(denom == 0)
                      System.exit(1);
                      result = num / denom;
        }
}
```

Tip

You first used the System.exit() method in Chapter 5 when you wrote code to close a dialog box. At that time, you used 0 as the argument to System.exit() to indicate the program was ending normally. Here, you use 1, which conventionally indicates a problem or error situation.

When you use the System.exit() method, the current application ends and control returns to the operating system. The convention is to return 1 if an error is causing program termination, or 0 if the program is ending normally. Using this exit() method circumvents displaying the error message because the program ends before the error occurs.

Exception handling provides a more elegant solution for handling error conditions. In object-oriented terminology, you "try" a procedure that might cause an error. A method that detects an error condition or Exception "throws an Exception", and the block of code that processes the error "catches the Exception".

TRYING CODE AND CATCHING EXCEPTIONS

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
- Statements that might cause Exceptions
- A closing curly brace

You must code at least one `catch` block 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. Each `catch` block can "catch" one type of Exception. 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
- Statements that take the action you want to use to handle the error condition
- A closing curly brace

If a method `throws` an Exception that will be caught, you must also use the keyword `throws`, followed by an Exception type in the method header. Figure 15-4 shows the general format of a `try...catch` pair.

```
public class someMethod throws someException
{
   try
   {
         //Statements that might cause an Exception
   }
   catch(someExceptionan ExceptionInstance)
   {
         //What to do about it
   }
//Statements here execute even if there was no Exception
}
```

Figure 15-4 General format of a `try...catch` pair

Tip: A `catch` block looks a lot like a catch() method 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.

Tip: Some programmers refer to a `catch` block as a `catch` clause.

In Figure 15-4, someException represents the Exception class or any of its subclasses. If an Exception occurs during the execution of the `try` block, then the statements in the `catch` block will execute. If no Exception occurs within the `try` block, then the `catch` block will not execute. Either way, the statements following the `catch` block execute normally. Next you will alter the MathMistake class so it catches the division-by-zero Exception.

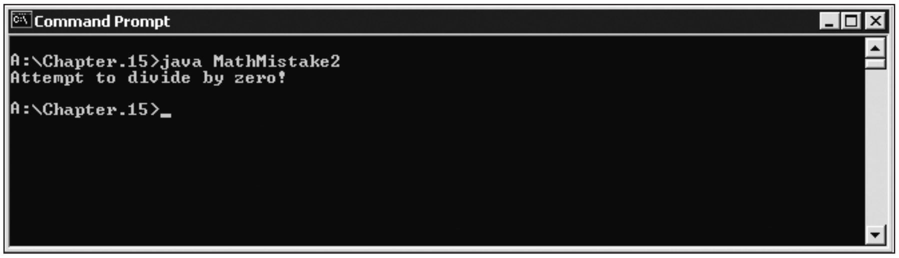
To catch an ArithmeticException:

1. Use your text editor's Save As command to save the MathMistake.java file as **MathMistake2.java**, and then change the class name in the class header from MathMistake to **`MathMistake2`**.
2. Position your insertion point at the end of the main() method header (`public static void main(String[] args)`), press the **Spacebar**, and then type **`throws ArithmeticException`**.
3. Position your insertion point at the end of the line that declares the three integer variables, and then press **[Enter]** to start a new line.
4. Type **`try`**, press **[Enter]**, and then type an opening curly brace.
5. Position your insertion point at the end of the line that performs division (`result = num / denom;`), press **[Enter]**, and then type a closing curly brace for the `try` block.
6. Press **[Enter]**, and then type a `catch` block that sends a message to the command line, as follows:

```
catch(ArithmeticException error)
{
      System.out.println("Attempt to divide by zero!");
}
```

Tip: If you want to send error messages to a different location from "normal" output, you can use System.err instead of System.out. For example, if a program writes a report to a specific disk file, you might want errors to write to a different disk file or to the screen.

7. Save the file, compile it, and then execute the program. The output looks like Figure 15-5, which shows that the Exception was caught successfully and your error message printed.



```
A:\Chapter.15>java MathMistake2
Attempt to divide by zero!

A:\Chapter.15>_
```

Figure 15-5 Output of MathMistake2

USING THE EXCEPTION GETMESSAGE() METHOD

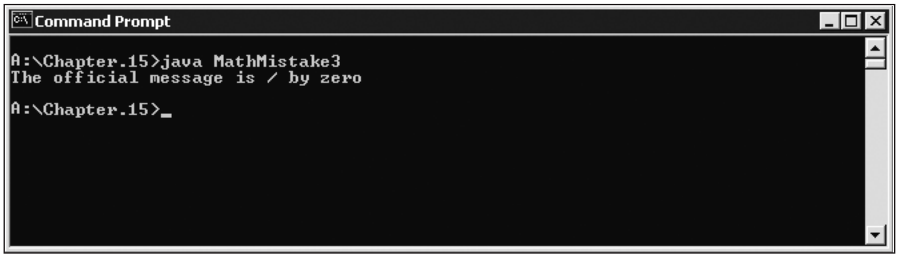
When the MathMistake2 program prints the error message ("Attempt to divide by zero!"), you cannot confirm that division by zero was the source of the error. In reality, *any* ArithmeticException generated within the `try` block in the program would be caught by the `catch` block in the method. Instead 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 Throwable Exception named someException, you code `someException.getMessage()`. In the next steps, you will use the getMessage() method instead of creating your own.

To use the getMessage() method with the MathMistake class:

1. Use your text editor's Save As command to save the MathMistake2.java file as **MathMistake3.java**.
2. Change the class header to **`public class MathMistake3`**.
3. Within the `catch` block, remove the existing println() statement and replace it with the following:

   ```
   System.out.println("The official message is " + error.get
   Message());
   ```

4. Save the file, and then compile and run it. Figure 15-6 shows the output. Java's analysis of the situation prints instead of your own.



```
A:\Chapter.15>java MathMistake3
The official message is / by zero

A:\Chapter.15>_
```

Figure 15-6 Output of MathMistake3

Of course, you might want to do more in a `catch` block than print an error message; after all, Java did that for you without catching any Exceptions. You also might want to add code to correct the error; such code would force the arithmetic to divide by one rather than by zero. Next you will add code to the `catch` block to catch the exception.

To add corrective code to the `catch` block of the MathMistake class:

1. Save the MathMistake3 file as **MathMistake4.java**.
2. Change the class header to reflect the new class name.
3. Position your insertion point at the end of the println() statement within the `catch` block of the main() method of the class, and then press **[Enter]** to

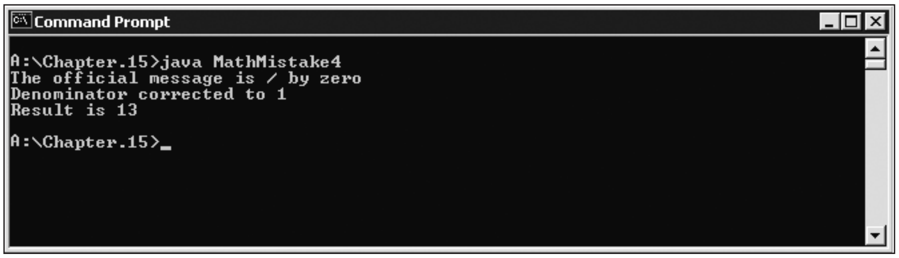
start a new line. Type the following message to indicate the action you are taking: **`System.out.println("Denominator corrected to 1");`**. Follow this statement with a recalculation of the result variable by typing the following code on a new line: **`result = num / 1;`**.

Tip: You can achieve the same result by coding `result = num;` instead of `result = num / 1;`. Explicitly dividing by one simply makes the code's intention clearer.

4. Position your insertion point after the closing curly brace of the `catch` block, and then press **[Enter]** to start a new line. Add the following println() statement to show the result:

   ```
   System.out.println("Result is " + result);
   ```

5. Save the program, and then compile and execute it. Figure 15-7 shows the result, which confirms that the `catch` block provides you with a usable result.



```
A:\Chapter.15>java MathMistake4
The official message is / by zero
Denominator corrected to 1
Result is 13

A:\Chapter.15>_
```

Figure 15-7 Output of MathMistake4

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 program in Figure 15-8. The main() method in the TwoMistakes class `throws` two types of Exceptions: ArithmeticExceptions and IndexOutOfBoundsExceptions. (An IndexOutOfBoundsException occurs when an array subscript is not within the allowed range.)

```
public class TwoMistakes
{
   public static void main(String[]args)
         throws ArithmeticException, IndexOutOfBoundsException
   {
         int num= 13,denom= 0,result;
         int[] array= {22,33,44};
         try
         {
               result= num/denom;//First try
               result= array[num];//Second try
         }
         catch(ArithmeticExceptionerror)
         {
               Systemoutprintln("Arithmetic error");
         }
         catch(IndexOutOfBoundsException error)
         {
               System.out.println("Index error");
         }
   }

}
```

Figure 15-8 TwoMistakes class

The TwoMistakes class declares three integers and an integer array with three elements. In the main() method, the `try` block executes, and at the first statement within the `try` block, an Exception occurs because the denom in the division problem is zero. The `try` block is abandoned, and the logic transfers to the first `catch` block. Division by zero causes an ArithmeticException, and because the first `catch` block receives an ArithmeticException, the message "Arithmetic error" prints. In this example, the second `try` statement is never attempted, and the second `catch` block is skipped.

If you make one minor change to the class in Figure 15-8, the process changes. You can force the division in the `try` block to succeed by substituting a constant value for the denom variable or by reversing the positions of num and denom in the line containing the `// First try` comment. With either of these changes, division by zero does not take place. The line containing `// First try` succeeds, and the program proceeds to the `// Second try` statement. This statement attempts to access element 13 of a three-element array, so it `throws` an IndexOutOfBoundsException. The `try` block is abandoned, and the first `catch` block is examined and found unsuitable because it does not catch an IndexOutOfBoundsException. The program logic proceeds to the second catch block whose Exception argument type is a match for the thrown Exception, so the message "Index error" prints.

Sometimes you want to execute the same code no matter which Exception type occurs. For example, within the TwoMistakes program in Figure 15-8, each of the two catch blocks prints a unique message. Instead, you might want both the ArithmeticException's **catch** block and the IndexOutOfBoundsException's **catch** block to use the getMessage() method. Because ArithmeticExceptions and IndexOutOfBoundsExceptions are both subclasses of Exception, you can rewrite the TwoMistakes class as shown in Figure 15-9, using a single generic **catch** block that can **catch** any type of Exception.

```
public class TwoMistakes
{
   public static void main(String[]args) throws
           ArithmeticException, IndexOutOfBoundsException
   {
         int num= 13,denom= 0,result;
         int[] array= {22,33,44};
         try
         {
               result= num/denom;//First try
               result= array[num];//Second try
         }
         catch(Exceptionerror)
         {
               System.out.println("Error is"+ error.getMessage());
         }
   }

}
```

Figure 15-9 TwoMistakes class using a single, generic `catch` block

The **catch** block in Figure 15-9 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 or array error occurs, the thrown error is "promoted" to an Exception error in the **catch** block. Through inheritance, ArithmeticExceptions and IndexOutOfBoundsExceptions are Exceptions, and an Exception is Throwable, so you can use the Throwable class getMessage() method.

When you list multiple **catch** blocks following a **try** block, you must be careful that some **catch** blocks don't become unreachable. For example, if successive **catch** blocks catch an IndexOutOfBoundsException and an ordinary Exception, then IndexOutOfBoundsException errors will cause the first **catch** to execute and other Exceptions will "fall through" to the more general Exception **catch** block. However, if you reverse the sequence of the **catch** blocks, then even IndexOutOfBoundsExceptions will be caught by the Exception **catch**. The IndexOutOfBoundsException **catch** block is unreachable because the Exception **catch** block is in its way and the class will not compile.

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 whether or not the preceding `try` block identifies Exceptions. Usually, you use a `finally` block to perform clean-up tasks that must happen whether or not any Exceptions occurred, and whether or not any Exceptions that occurred were caught. Figure 15-10 shows the format of a `try…catch` sequence that uses a `finally` block.

```
public class someMethod throws someException
{
   try
   {
         //Statements that might cause an Exception
   }
   catch(someExceptionan ExceptionInstance)
   {
         //What to do about it
   }
   finally
   {
         //Statements here always execute
   }

}
```

Figure 15-10 General form of a `try...catch` block with a `finally` block

Compare Figure 15-10 to Figure 15-4 shown earlier in this chapter. With the program in Figure 15-4, when the `try` code works without error, control passes to the statements at the end of the method. Also, when the `try` code fails and `throws` an Exception, if the Exception is caught, then the `catch` block executes, and again, control 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 always execute. However, the last set of statements might never execute for at least two reasons:

- It is possible that an unplanned Exception will occur.
- The `try` or `catch` block might contain a `System.exit();` statement.

Any `try` block might throw an Exception 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 MathMistake program 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. Likewise, if the `try` block contains an exit() statement, execution stops 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 Chapter 16. For now, however, consider the pseudocode that represents part of the logic for a typical file-handling program:

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

The preceding pseudocode represents a program that opens a file; if the file does not exist or is empty, an input/output exception, or IOException, is thrown and the `catch` block handles the error. However, because the program uses an array, it is possible that even though the file opened successfully, an uncaught ArrayIndexOutOfBoundsException might occur. In such an event, you 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 Exception causes the method to abandon prematurely—perhaps the array is not large enough to hold the data, or calculating the average results in division by zero. These Exceptions would not allow the `try` block to finish, nor would they cause the `catch` block to execute.

Tip

If a `try` block calls the System.exit() method, and the `finally` block calls the same method, it is the exit() method in the `finally` block that will actually execute. The `try` block's exit() method call will be abandoned.

UNDERSTANDING THE LIMITATIONS OF TRADITIONAL ERROR HANDLING

Before the conception 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 15-11.

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

Figure 15-11 Pseudocode representing traditional error checking

The program in Figure 15-11 performs methodA; it then performs methodB only if methodA is successful. Similarly, methodC executes only when methodA and methodB are successful. When any method fails, the program sets an appropriate errorCode to A, B, or C. (Presumably, the errorCode is used later in the program.) The program is difficult to follow, and the purpose of the program (and its presumed outcome when there are no errors)—to print the 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 15-12. Using the `try…catch` object-oriented technique provides the same results as the traditional method, but the real statements of the program (calling methods A, B, and C, and printing 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.

```
try
{
   methodA(and maybe throw anException)
   methodB(and maybe throw anException)
   methodC(and maybe throw anException)
   everything's okay so print 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 15-12 Pseudocode representing object-oriented Exception handling

Specifying the Exceptions a Method Can Throw

When you write a method that might throw an Exception, you can type the clause `throws <name>Exception` after the method header to indicate the type of Exception that might be thrown. Every Java method you write has the potential to throw an Exception. Some Exceptions, such as an InternalErrorException, can occur anywhere, at any time. However, for most Java methods that you write, you do not use a `throws` clause. For example, you have used a `throws` clause only a few times in the many programs you have written while working through this book. 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 for every possible error, including equipment failures and memory problems. Most exceptions never have to be explicitly thrown or caught.

Tip You never have to throw Error or RuntimeException exceptions explicitly. Most of the errors you received when you made mistakes in your Java programs are RuntimeExceptions—unplanned exceptions that occur during a program's execution.

One exception to the rule of not throwing Exceptions involves the IOException. You learned in Chapter 5 that you must include a `throws` clause in the method header of programs that allow keyboard input. In Chapter 16 you will discover that you also must include a `throws` clause in programs that use file input. However, even when you are not required to handle an Exception, you might choose to do so. When your method will throw an Exception that you want to handle, you must include the `throws` clause in the method header.

Tip: InterruptedException is another example of a `throws` clause that Java requires that you use when you are working with threads. You will learn about threads in Chapter 17.

When a method you write `throws` an Exception, the method can `catch` the Exception, although it is not required to do so. There are many times when you won't want a method to handle its own Exception. With many methods, 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. The calling program might need to handle the error differently, depending on its purpose. For example, one program that divides values might need to terminate if division by zero occurs. A different program simply might want the user to reenter the data to be used. The method that contains the division statement can throw the error, and the calling program can assume the responsibility for handling the error detected by the method.

Tip: You know a method can throw without catching because you have written methods that use keyboard input. With those methods, you threw an Exception, but you did not provide a `catch` block.

Java requires that you use the `throws` clause in the header of a method that might `throw` an Exception so that programs that use your methods are notified of the potential for an Exception. When you use any method, to be able to use the 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 first know what types of arguments the method that you send it requires. You can call a method without knowing its return type, but if you do so, 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. A method's header, including its name, any arguments, and any `throws` clause, is called the **method's signature**.

Next you will create a class that contains two methods that throw Exceptions but don't `catch` them. The PickMenu class allows Event Handlers Incorporated customers to choose a dinner menu selection as part of their event-planning process. Before you create PickMenu, you will create the Menu class that lists dinner choices for customers and allows them to make a selection.

To create the Menu class:

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

   ```
   public class Menu
   {
   ```

2. Type the following String array for three entrée choices:

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

3. Add the displayMenu() method, which lists each entrée 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` in the println() prompt.

   ```
   public void displayMenu()
   {
          System.out.println
            ("Type your selection, then
          press [Enter].");
          for(int x = 0; x < entreeChoice.length; ++x)
                  System.out.println("Type " + (x + 1) +
                     " for " + entreeChoice[x]);
   }
   ```

4. Create the following getSelection() method, which requires an integer argument and returns the name of one of the selected menu items. Because the user enters a value that is one higher than the actual subscript, you need to subtract one from `x` when accessing the array. Finally, include the closing curly brace for the Menu class.

   ```
      public String getSelection (int x)
      {
          return(entreeChoice[x - 1]);
      }
      }
   ```

5. Save the file as **Menu.java** in the Chapter.15 folder on your Student Disk.

6. Compile the class using the **`javac`** command.

Next you can create the PickMenu class, which lets the customer choose from the available dinner entrée options. The PickMenu class declares a Menu, an integer that holds the user's numeric menu choice, and a String named guestChoice that holds the name of the entrée that the customer selects.

To enable the PickMenu class to operate with different kinds of Menus in the future, you 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 any 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 will appear.

Tip

You have written many programs using GUI dialog boxes for input and output. However, you will use command-line prompts and input here to better illustrate Exception handling.

To create the PickMenu class:

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

```
public class PickMenu
{
       Menu briefMenu;
       int choice;
       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 setChoice() method, which prompts the user to select from the available menu. The PickMenu() constructor method must `throw` an Exception because it contains the setGuestChoice() method, which uses keyboard input, and any method that uses keyboard input or calls a method that uses keyboard input must `throw` the potential Exception.

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

3. The following setGuestChoice() method displays the menu and reads keyboard data entry, so the method `throws` an Exception. Start the method by declaring a character and String for input:

```
public void setGuestChoice() throws Exception
{
       char newChar;
       String inputString = new String();
```

4. Add the following data-entry procedure, which is similar to others that you have written:

```
System.out.println("Choose from the following menu:");
briefMenu.displayMenu();
newChar = (char)System.in.read();
```

```
while(newChar >= '0' && newChar <= '9')
{
      inputString = inputString + newChar;
      newChar = (char)System.in.read();
}
System.in.read();
```

5. Add the following code to convert the entered String to an integer, and then use it as an argument to the getSelection() method that you wrote in the Menu class. Because briefMenu is a Menu (an instance of the Menu class), it has access to the getSelection() method. When you pass an integer to the getSelection() method, it returns one of the Strings in the menu. Here, you assign the returned String to the guestChoice field. Finally, you end the setGuestChoice() method with a closing curly brace.

```
      choice = Integer.parseInt(inputString);
      guestChoice = briefMenu.getSelection(choice);
}
```

6. Add the following getGuestChoice() method. This method is simpler; it returns the String that represents the customer's menu selection. Finally, include the closing curly brace for the PickMenu class.

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

7. Save the file as **PickMenu.java** in the Chapter.15 folder on your Student Disk, and then compile it using the `javac` command.

You created a Menu class that simply holds a list of food items, displays itself, and allows you to retrieve a specific item. You also created a PickMenu class that has 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 contains two methods that throw Exceptions, but no methods that contain ways to `catch` those Exceptions. Next you will write a program that uses the PickMenu class. This program can `catch` Exceptions that PickMenu `throws`.

To write the PlanMenu class:

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

```
public class PlanMenu
{
       public static void main(String[] args)
       {
```

2. Construct the following Menu named briefMenu, and also declare a PickMenu object that you name **entrée**. 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 will just declare entrée and assign it `null`. Also, you will declare a String that will hold the customer's menu selection.

```
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 entrée object. Because entrée 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();
}
```

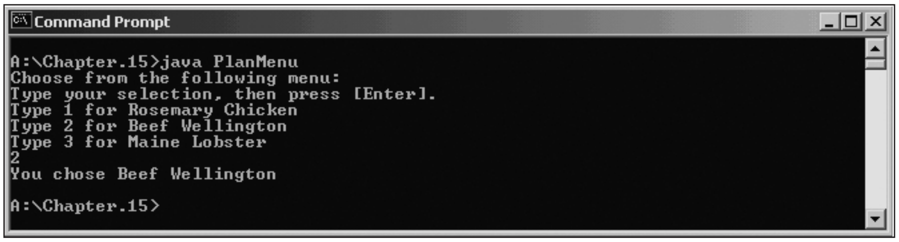
4. The `catch` block must immediately follow the `try` block. When the `try` block fails, guestChoice will not have a valid value, so recover from the Exception by assigning a value to guestChoice within the following `catch` block:

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

5. Use the following code to print the customer's choice at the end of the PlanMenu program, and then add closing curly braces for the main() method and the class:

```
          System.out.println("You chose " + guestChoice);
     }
}
```

6. Save the file as **PlanMenu.java** in the Chapter.15 folder on your Student Disk, and then compile and execute it. Choose an entrée selection by typing its number from the menu, and then compare your results to Figure 15-13.

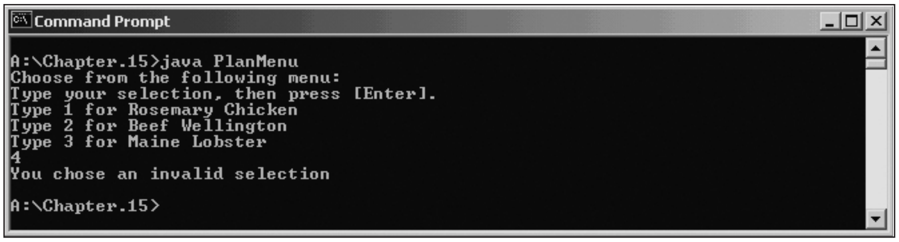


```
A:\Chapter.15>java PlanMenu
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Rosemary Chicken
Type 2 for Beef Wellington
Type 3 for Maine Lobster
2
You chose Beef Wellington

A:\Chapter.15>
```

Figure 15-13 Sample run of the PlanMenu program

7. The PlanMenu program works well when you enter a valid menu selection. One way that you can force an Exception to take place is to enter an invalid menu selection at the prompt. Run the PlanMenu program 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 PickValue class `throws` either type of Exception and the PlanMenu program catches it, guestChoice takes on the value "an invalid selection" and the program ends smoothly, as shown in Figure 15-14.



```
A:\Chapter.15>java PlanMenu
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Rosemary Chicken
Type 2 for Beef Wellington
Type 3 for Maine Lobster
4
You chose an invalid selection

A:\Chapter.15>
```

Figure 15-14 Exceptional run of the PlanMenu program

HANDLING EXCEPTIONS UNIQUELY WITH EACH `catch`

An advantage to using object-oriented exception-handling techniques is that you gain the ability to appropriately deal with Exceptions as you make conscious decisions about how to handle them. When you use a class and one of its methods `throws` an Exception, the class that `throws` the Exception does not have to `catch` it. Instead, your calling program can `catch` the Exception, and you can decide what you want to do. Just as a police officer can deal with a speeding driver differently depending on circumstances, you can react to Exceptions specifically for your current purposes. For example, the PickMenu class you created `throws` Exceptions. When you write new programs that use the PickMenu class, you can decide to handle error conditions differently within each program you write.

Next you will extend the Menu class to create a new class named VegetarianMenu. Subsequently, when you write a program that uses PickMenu with the VegetarianMenu, you can deal with any Exception differently than you did when you wrote the PlanMenu program.

To create the VegetarianMenu class:

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

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

2. Provide new menu choices for the VegetarianMenu as follows:

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

3. 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];
     }
     }
   ```

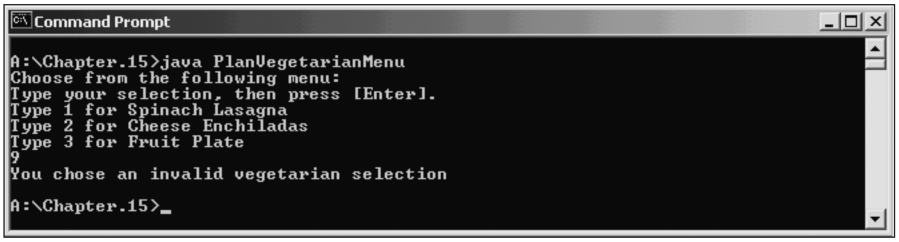
4. Save the class as **VegetarianMenu.java** in the Chapter.15 folder on your Student Disk, and then compile it.

5. Now write a program 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, and then immediately save it as **PlanVegetarianMenu.java**.

6. Change the class name in the header to **`PlanVegetarianMenu`**.
7. Change the first statement within the main() method as follows so it declares a VegetarianMenu instead of a Menu:

   ```
   VegetarianMenu briefMenu = new VegetarianMenu();
   ```
8. 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";
   ```
9. Save the file in the Chapter.15 folder on your Student Disk, compile it, and then run the program. When you see the vegetarian menu, enter a valid selection and confirm that the program works correctly. Run the program again and enter an invalid selection. The error message, shown in Figure 15-15, identifies your invalid entry as "an invalid vegetarian selection". Remember that you did not change the PickMenu class. Your new PlanVegetarianMenu program 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 programs in which you `catch` them.



```
A:\Chapter.15>java PlanVegetarianMenu
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Spinach Lasagna
Type 2 for Cheese Enchiladas
Type 3 for Fruit Plate
9
You chose an invalid vegetarian selection

A:\Chapter.15>_
```

Figure 15-15 Exceptional run of the PlanVegitarianMenu program

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, then while methodC executes, the computer must "remember" that it is going to return to methodB and, eventually, to return methodA. The memory location known as the **call stack** is where the computer stores the list of locations to which the system must return.

When a method `throws` an Exception, and if the method does not `catch` the Exception, then 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 15-16 shows how the call stack works. If methodA calls methodB, and methodB calls methodC, and methodC `throws` an Exception, then Java first looks for a `catch` block in methodC. If none exists, then Java looks for the same thing in methodB. If methodB does not have a `catch` block, then Java looks to methodA. If methodA cannot `catch` the Exception, then it is thrown to the operating system.

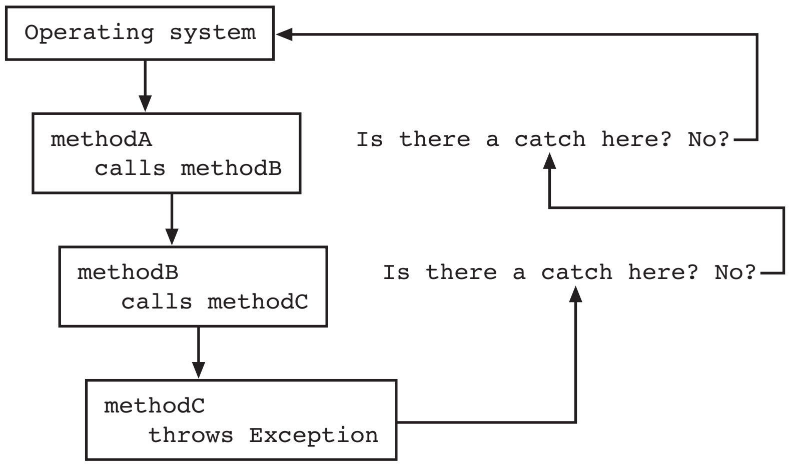


Figure 15-16 Cycling through the call stack

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. However, when a program uses several classes, this system's disadvantage is that it is very difficult for the programmer to locate the original source of an Exception.

You have already used the Throwable method getMessage() to obtain information about an Exception. Another useful Exception method is the printStackTrace() method. 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.

To use the printStackTrace() method:

1. Open the **PlanMenu.java** file in your text editor, and then save it as **PlanMenuWithStackTrace.java**.
2. Change the class header to **`PlanMenuWithStackTrace`**.

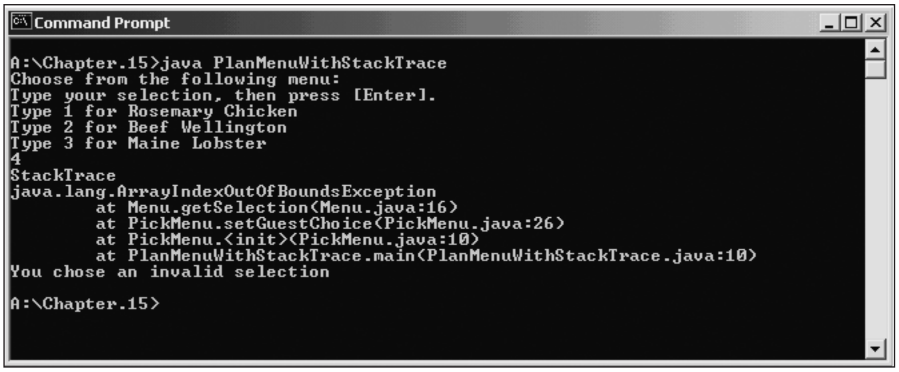
3. Position your insertion point within the `catch` block after the statement `guestChoice = "an invalid selection";`, and then press **[Enter]** to start a new line.

4. Type the following two new statements to identify and print the stack trace:

```
System.out.println("Stack Trace");
error.printStackTrace();
```

5. Save the file in the Chapter.15 folder on your Student Disk, and then compile and execute it. After the menu appears, enter an invalid selection. If you entered 4, your screen looks like Figure 15-17. If you read the list that follows the Stack Trace heading, you see that an ArrayIndexOutOfBoundsException occurred in the method Menu.getSelection(). That method was called by the PickMenu.setGuestChoice() method, which in turn was initiated by the PickMenu constructor. The PickMenu constructor was called from the PlanMenuWithStackTrace.main() method. You see the line number as additional information within each method where the Exception occurred (your line numbers might be different). If you did not understand why entering 4 caused an error, you would use the stack trace information to examine the Menu.getSelection() method as the first source of the error. Using printStackTrace() can be a helpful debugging tool.

6. Run the **PlanMenuWithStackTrace** program again, and then enter **A** for the user selection. You can see from the stack trace that this time the Exception does not originate in the Menu.getSelection() method. This program stops at the parseInt() method before the program attempts getSelection().



```
A:\Chapter.15>java PlanMenuWithStackTrace
Choose from the following menu:
Type your selection, then press [Enter].
Type 1 for Rosemary Chicken
Type 2 for Beef Wellington
Type 3 for Maine Lobster
4
StackTrace
java.lang.ArrayIndexOutOfBoundsException
        at Menu.getSelection(Menu.java:16)
        at PickMenu.setGuestChoice(PickMenu.java:26)
        at PickMenu.<init>(PickMenu.java:10)
        at PlanMenuWithStackTrace.main(PlanMenuWithStackTrace.java:10)
You chose an invalid selection

A:\Chapter.15>
```

Figure 15-17 Exceptional run of the PlanMenuWithStackTrace program

Tip

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

CREATING YOUR OWN EXCEPTIONS

Java provides over 40 categories of Exceptions that you can throw in your programs. However, Java's creators could not predict every condition that might be an Exception in your programs. 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 Exceptions.

To create your own throwable Exception, you must extend a subclass of Throwable. Recall from Figure 15-1 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, you should extend your Exceptions from the Exception class. You can extend any existing Exception subclass, such as ArithmeticException or NullPointerException, but usually you want to extend directly from Exception.

Tip

When you create an Exception, it's conventional to end its name with Exception.

Next you will create a PartyException class for Event Handlers Incorporated. The PartyException class has just one method—a constructor. You can include data fields and other methods within the PartyException class if you want. For example, you might want the PartyException class to contain a customized toString() method that you can use to display party details. To keep this example simple, however, you will include only the constructor. The constructor will take a String argument representing the name of the party, such as the *Jones* party. You will pass this String to the Exception superclass so you can use the String within superclass methods, such as getMessage().

To write your own Exception:

1. Open a new file in your text editor, and then type the following PartyException class:

```
public class PartyException extends Exception
{
       public PartyException(String s)
       {
```

```
                super(s);
        }
}
```

2. Save the file as **PartyException.java** in the Chapter.15 folder on your Student Disk, and then compile it.

Next you will create a Party class that holds information about any party hosted by Event Handlers Incorporated. The Party class holds two fields—the name of the party host and the number of guests—and it contains a constructor that requires values for both fields. Event Handlers does not host parties with fewer than 10 guests. Therefore, you want to test the guest number in the constructor, and throw a PartyException when the guest number is less than 10. The PartyException class constructor requires a String argument, so pass the name of the party host to the Exception. That way, you can use the host's name in error messages generated by the Exception class.

Tip

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.

To create the Party class:

1. Open a new file in your text editor, and then type the following Party class:

```
public class Party
{
        String host = new String();
        int guests;
        public Party(String hst, int gst) throws
        PartyException
        {
                host = hst;
                guests = gst;
                if(gst < 10)
                         throw(new PartyException(hst));
        }
}
```

2. Save the file as **Party.java** in the Chapter.15 folder on your Student Disk, and then compile it.

Next you will write a program that instantiates a few Party objects. When you run the program, you can observe which objects generate PartyExceptions.

To write the ThrowParty program:

1. Open a new text file, and then type the following first few lines of the ThrowParty program and its main() method:

```
public class ThrowParty
{
       public static void main(String[] args)
```

2. Enter the following code to attempt to construct three Party objects in a `try` block:

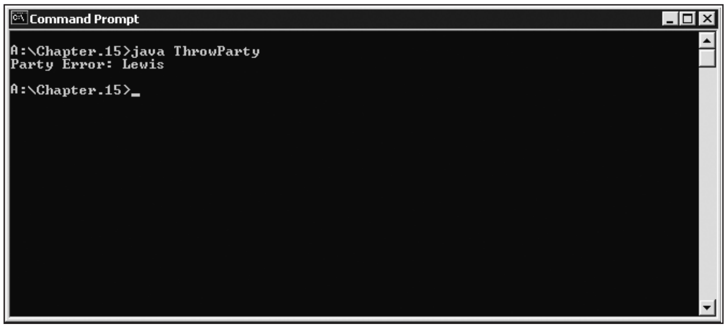
```
try
{
       Party first = new Party("Jones",15);
       Party second = new Party("Lewis",5);
       Party third = new Party("Newman",10);
}
```

3. Enter the following code to `catch` any PartyExceptions and use the Exception class getMessage() method to display a message, and then add two closing curly braces:

```
  catch(PartyException error)
  {
   System.out.println("Party Error: " +
      error.getMessage ());
  }
 }
}
```

4. Save the file as **ThrowParty.java** in the Chapter.15 folder on your Student Disk.

5. Compile **ThrowParty.java**, and then run the program. Compare your results to Figure 15-18. The two Party objects constructed with 10 or more guests compiled successfully, but the Party object with only 5 guests generated a PartyException.



```
A:\Chapter.15>java ThrowParty
Party Error: Lewis

A:\Chapter.15>_
```

Figure 15-18 Output of the ThrowParty program

You should not create an excessive number of special Exception types for your classes, especially if the Java development environment already contains an Exception that will `catch` the error. Extra Exception types add complexity for other programmers who will 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 also allow for errors to be passed up the stack and traced.

Chapter Summary

- An exception is an unexpected or error condition. The object-oriented techniques to manage such errors comprise the group of methods known as exception handling. In Java, there are two basic classes of errors—Error and Exception. Both of these classes descend from the Throwable class.
- In object-oriented terminology, you "try" a procedure that might not complete correctly. A method that detects an error condition or Exception "throws an Exception", and the block of code that processes the error "catches the Exception".
- Exceptions inherit the getMessage() method from the Throwable class.
- You can place as many statements as you need within a `try` block; only the first error-generating statement `throws` an Exception. You can catch as many Exceptions as you want, or you can let the operating system handle them.
- When you have actions that always must occur at the end of a `try...catch` sequence, you use a `finally` block.
- In a traditional, non-object-oriented program, error-checking code is complex. Object-oriented exception-handling allows you to isolate error-handling code.
- You can use the clause `throws <name>Exception` after the method header to indicate the type of Exception that might be thrown.
- An advantage of using object-oriented exception-handling techniques is they afford you the ability to make your reaction to Exceptions specific for your current purposes—you can handle the same Exception differently in every application, or choose not to handle it at all.
- The call stack is the memory location where the computer stores the list of locations to which the system must return after method calls. When a method `throws` an Exception, and if the method does not `catch` the Exception, then the Exception is thrown to the next method "up" the call stack. You can display this list using the printStackTrace() method.
- You can create your own Exceptions by extending the Exception class.

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. tries
   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
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 `try` three statements, and include three `catch` blocks, and the second `try` statement `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
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. When you use any method, you must know three pieces of information to use the method to its full potential; but you don't need to know ________________.
    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 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. To create your own Exception that you can throw, you must extend a subclass of ________________.
    a. Object
    b. Throwable
    c. Exception
    d. Error

EXERCISES

1. Write a program named GoTooFar in which you declare an array of five integers and store five values in the array. Initialize a subscript to zero. Write a `try` block in which you access each element of the array, subsequently increasing the subscript by 1. Create a `catch` block that catches the eventual ArrayIndexOutOfBoundsException, and then print to the screen the message, "Now you've gone too far." Save the program as **GoTooFar.java** in the Chapter.15 folder of your Student Disk.
2. The Integer.parseInt() method requires an integer argument. Write a program in which you try to parse a String. Catch the NumberFormatExceptionError that is thrown, and then display an appropriate error message. Save the program as **TryToParseString.java** in the Chapter.15 folder of your Student Disk.
3. Write an application program that prompts the user to enter a number to use as an array size, then attempt to declare an array using the entered size. If the array is created successfully, display an appropriate message. Use a `catch` block that executes if the array size is non-numeric or negative. Save the program as **NegativeArray.java** in the Chapter.15 folder of your Student Disk.
4. Write a program that throws and catches an ArithmeticException. Declare a variable and assign it a value. Test the variable, and if it is negative, throw an ArithmeticException. Otherwise, use the Math.sqrt() method to determine the square root. Save the program as **SqrtError.java** in the Chapter.15 folder of your Student Disk.
5. Create an EmployeeException class whose constructor receives a String that consists of an employee's ID and pay rate. Create an Employee class with two fields, idNum and hourlyWage. The Employee constructor requires values for both fields. Upon construction, throw an EmployeeException if the hourlyWage is less

than $6.00 or over $50.00. Write a program that establishes at least three Employees with hourlyWages that are above, below, and within the allowed range. Save the program as **ThrowEmployee.java** in the Chapter.15 folder of your Student Disk.

6. a. Create an IceCreamConeException class whose constructor receives a String that consists of an ice cream cone's flavor and number of scoops. Create an IceCreamCone class with two fields—iceCreamFlavor and scoops. The IceCreamCone constructor calls two data-entry methods—getFlavor() and getScoops(). The getScoops() method `throws` an IceCreamConeException when the scoop quantity exceeds 3. Write a program that establishes several IceCreamCone objects and handles the Exception. Save the program as **ThrowIceCream.java** in the Chapter.15 folder of your Student Disk.

   b. Modify the IceCreamCone getFlavor() 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 a program that establishes several IceCreamCone objects and handles the new Exception. Save the program as **ThrowIceCream2.java** in the Chapter.15 folder of your Student Disk.

7. Write a program that displays a student ID number and asks the user to enter a numeric test score for the student. Create a ScoreException class, and throw a ScoreException for that 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. Save the program as **TestScore.java** in the Chapter.15 folder of your Student Disk.

8. Write a program that displays a student ID number and asks the user to enter a test letter grade for the student. Create an Exception class named GradeException, and throw a GradeException if the user does not enter a valid letter grade. `Catch` the GradeException and then display an appropriate message. Save the program as **TestGrade.java** in the Chapter.15 folder of your Student Disk.

9. Write an applet that prompts the user for a color name. If it is not red, white, or blue, throw an Exception. Otherwise, change the applet's background color appropriately. Save the program as **RWBApplet.java** in the Chapter.15 folder of your Student Disk.

10. Write an applet that prompts the user for an ID number and an age. Create an Exception class and throw an Exception of that class if the ID is not in the range of valid ID numbers (zero through 899), or if the age is not in the range of valid ages (0 through 89). Catch the Exception and then display an appropriate message. Save the program as **BadIDAndAge.java** in the Chapter.15 folder of your Student Disk.

11. Each of the following files in the Chapter.15 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugFifteen1.java will become FixDebugFifteen1.java. You will also use a file named DebugEmployeeIDException.java with the DebugFifteen4.javafile.
    a. DebugFifteen1.java
    b. DebugFifteen2.java
    c. DebugFifteen3.java
    d. DebugFifteen4. java

CASE PROJECT

Gadgets by Mail sells many interesting items through its catalogs. Write a program that prompts the user for an item number and quantity ordered. Create an Exception class named OrderException, and throw an OrderException if the user does not enter a valid item number (at least 111, but no more than 999) or quantity (at least 1, but no more than 50). Catch the OrderException and display an appropriate message. If the item number and quantity are valid, display the final price, which is $2.00 per item, no matter what the item number is.