CHAPTER 11

Exceptions and Input/Output Operations

CHAPTER CONTENTS

Introduction
11.1 Simple Exception Handling
11.1.1 Using *try* and *catch* Blocks
11.1.2 Catching Multiple Exceptions
11.1.3 User-Defined Exceptions
11.2 The *java.io* Package
11.3 Reading Text Files Using *Scanner*
11.4 Writing and Appending to Text Files
11.4.1 Writing to Text Files
11.4.2 Appending to Text Files
11.5 Reading Structured Text Files
11.5.1 Parsing a *String* Using *Scanner*
11.5.2 Reading Structured Data Using *Scanner*
11.6 **Programming Activity 1: Reading from a Structured Text File**
11.7 Reading and Writing Objects to a File
11.7.1 Writing Objects to Files
11.7.2 Reading Objects from Files
11.8 **Programming Activity 2: Reading Objects from a File**
11.9 Chapter Summary
11.10 Exercises, Problems, and Projects
11.10.1 Multiple Choice Exercises
11.10.2 Reading and Understanding Code
11.10.3 Fill In the Code
11.10.4 Identifying Errors in Code
11.10.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
11.10.6 Write a Short Program
11.10.7 Programming Projects
11.10.8 Technical Writing
11.10.9 Group Project

Introduction

Up to now, whenever our programs needed data, we have supplied that data using one of several different methods:

- Assigning values to variables in our program. This is known as **hard coding** the values of variables.
- Prompting the user for values using the *Scanner* class.
- Reading data from a text file using the *Scanner* class.
- Prompting the user for values using a *JOptionPane* dialog box.
- Generating random numbers using the *Random* class.

Typically, whenever we prompted the user to supply values, we asked for one value at a time. If a program needs a large number of values, however, prompting the user for each value will take a long time and will be impractical. Furthermore, as the number of values increases, the potential for input errors increases as well.

Programs often use existing data accumulated by an organization, such as a university, a government, or a corporation. Typically, the volume of data is significant, and again, data entry through the keyboard is impractical.

Furthermore, these large amounts of data typically reside in two types of storage:

- disk files
- databases

Working with databases is beyond the scope of this book. In most of this chapter, we concentrate on reading from and writing to files.

But there is a prerequisite to all this: understanding the concept of exceptions, their associated classes, and exception handling.

11.1 Simple Exception Handling

By now you should have discovered that sometimes your program doesn't work, even though you didn't get any compiler errors. At run time, logic errors can surface. For example, we might attempt to divide an integer by 0

or try to access the 11th element in a 10-element array. Java is a robust language and does not allow these "illegal" operations to occur unnoticed.

These illegal operations generate **exceptions**. Some exceptions are generated by the Java Virtual Machine, while others are generated by constructors or other methods. For example, a method might generate an exception when it detects an attempted illegal operation or an illegal parameter.

By default, when an exception is generated in an application that does not have a graphical user interface, the program will terminate. In many cases, however, we can attempt to recover from the exception and continue running the program. This is called **handling the exception**. For the programmer to handle an exception, Java provides two tools:

- exception classes
- the *try*, *catch*, and *finally* blocks

The *Exception* class is the superclass of all exception classes, which encapsulate specific exceptions, such as integer division by 0, attempting to access an out-of-bounds array index, an illegal number format, using a *null* object reference to call a method, trying to open a file that does not exist, and others.

Figure 11.1 is an inheritance hierarchy showing only a few of the Java exception classes. The *Exception* class and *RuntimeException* and its subclasses are in the *java.lang* package. The *IOException* class and its subclass, *FileNotFoundException*, are in the *java.io* package.

11.1.1 Using *try* and *catch* Blocks

Throughout this book, we have used dialog boxes to prompt the user for input values. Example 11.1 shows a program that prompts the user for an integer. The *showInputDialog* method of the *JOptionPane* class returns the user's input as a *String* (lines 12–13), which we then convert to an *int* using the *parseInt* method of the *Integer* wrapper class (line 17).

```
1 /*  An exception generated by the parseInt method
2 *   Anderson, Franceschi
3 */
4
5 import javax.swing.JOptionPane;
6
7 public class DialogBoxInput
8 {
```

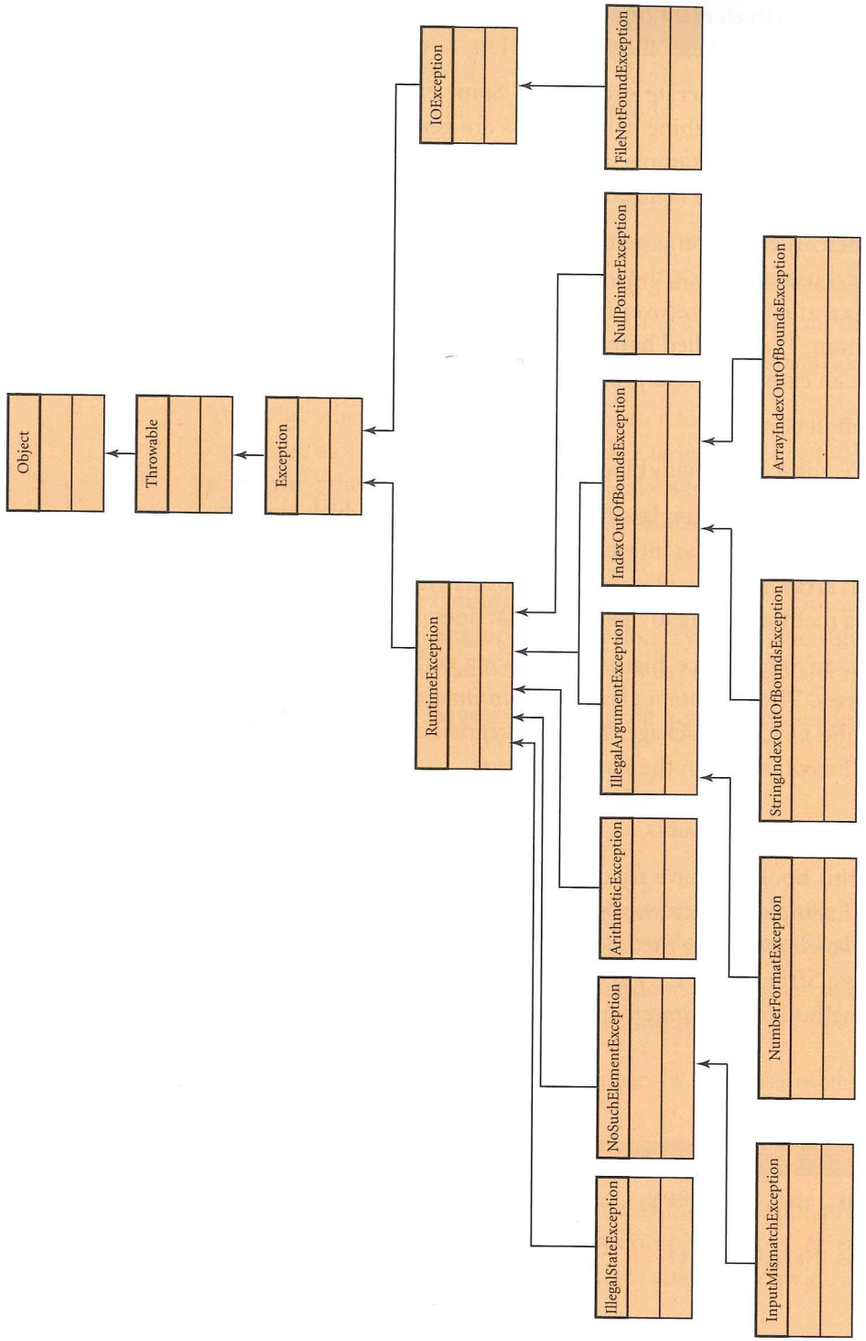


Figure 11.1 Inheritance Hierarchy for Various Exception Classes


```
 9   public static void main( String [ ] args )
10   {
11     // prompt for input; return value is a String
12     String s = JOptionPane.showInputDialog( null,
13                                 "Enter an integer" );
14     System.out.println( "You entered " + s );
15
16     // attempt to convert the String to an int
17     int n = Integer.parseInt( s );
18     System.out.println( "Conversion was successful. "
19                         + "The integer is " + n );
20   }
21 }
```

EXAMPLE 11.1 Generating an Exception

Figure 11.2 shows a successful run of the program with the user input being 45.

Everything goes well as long as the user's input can be converted to an *int*. But what happens when the user types characters other than digits? Figure 11.3 shows the output from the same program when the user enters "a."

At run time, the call to the *parseInt* method (line 17) generates a *NumberFormatException*, which terminates the program. Lines 18–19 of the program are never executed. We can determine the line that generated the exception by reading the last line of the console output, which identifies the

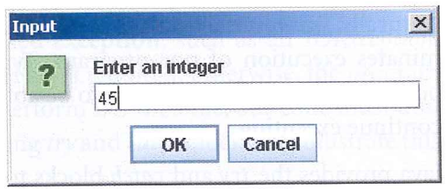
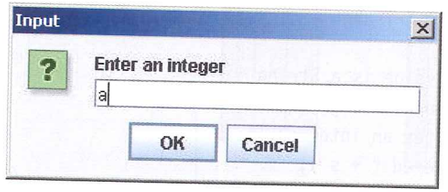


Figure 11.2 Successful Conversion of a *String* to an *int*

```
You entered 45
Conversion was successful. The integer is 45.
```

Figure 11.3
Output Showing a *NumberFormatException* at Run Time




```
You entered a
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
        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 DialogBoxInput.main(DialogBoxInput.java:17)
```

source of the exception: line 17 in the *main* method of the *DialogBoxInput* class.

Let's take a look at what happened here. The *parseInt* method has the following header:

```
public static int parseInt( String str ) throws NumberFormatException
```

The *throws NumberFormatException* clause indicates that the *parseInt* method may detect a situation for which it will generate a *NumberFormatException*. In fact, if the *String* argument cannot be converted to an *int*, the *parseInt* method will generate, or **throw**, a *NumberFormatException*.

Naturally, we would like to avoid the situation when invalid user input terminates execution of our program. Typos and invalid input values can occur in any program. We need to be able to recover from these errors and continue executing.

Java provides the *try* and *catch* blocks to allow us to handle exceptions so that our code can continue to run. We put the code that might generate an exception inside the *try* block, and we put the code to recover from the exception inside a *catch* block. If an exception is thrown by the code inside the *try* block, then execution will jump to the *catch* block, where we write code to handle that exception. If nothing illegal happens in the *try* block, the code in the *catch* block will be skipped.

The minimum syntax for a *try* and *catch* block is as follows:

```
try
{
  // code that might generate an exception
}
catch ( ExceptionClass exceptionObjRef )
{
  // code to recover from the exception
}
```

The curly braces are required for both the *try* body and the *catch* body even if the bodies have only one statement, or even no statements.

Note that the *ExceptionClass* parameter of the *catch* clause specifies one and only one *ExceptionClass*. Listing zero or two or more *ExceptionClasses* in the *catch* clause will generate a compiler error.

COMMON ERROR TRAP

Omitting curly braces around the *try* and *catch* blocks will generate a compiler error. A *catch* clause listing several *Exception* classes as parameters will also generate a compiler error.

Java distinguishes between two types of exceptions:

- unchecked, those that are subclasses of *Error* or *RuntimeException*
- checked, any other exception class

An **unchecked exception**, such as an *ArithmeticException* caused by attempting integer division by 0, a *NumberFormatException*, or a *NullPointerException*, does not have to be handled with a *try* and *catch* block. In other words, if you omit the *try* and *catch* blocks, your code will compile without an error. If one of these unchecked exceptions is generated at run time, however, the JVM will catch it and print output similar to that shown in Figure 11.3.

Code that could generate a **checked exception**, such as an *IOException*, must be coded within a *try* block. This is required; otherwise, the program will not compile. Thus, when we perform I/O on a file, our code must deal with a potential *IOException* by using *try* and *catch* blocks. We illustrate this later in the chapter.

In the *catch* block, we can use the *Exception* parameter as an object reference to get more information about what caused the exception. Table 11.1 shows three methods inherited by the *Exception* classes.

Example 11.2 shows Version 2 of the *DialogBoxInput* application, using a *try* and *catch* block to detect the exception if it occurs.

TABLE 11.1 Useful Methods of Exception Classes

Methods of Exception Classes	
Return value	**Method name and argument list**
String	getMessage( )
	returns a message indicating the cause of the exception
String	toString( )
	returns a *String* containing the exception class name and a message indicating the cause of the exception
void	printStackTrace( )
	prints the line number of the code that caused the exception, along with the sequence of method calls leading up to the exception

We prompt the user and echo the input on lines 14–17. Then we define a *try* block (lines 19–24) and insert the *parseInt* method call that performs the conversion. The *catch* block (lines 25–35) contains the code we want to execute if the *parseInt* method *throws* an exception.

```
1 /*  DialogBoxInput, Version 2
2 *   Catching the exception generated by the parseInt method
3 *   Anderson, Franceschi
4 */
5
6 import javax.swing.JOptionPane;
7
8 public class DialogBoxInput
9 {
10   public static void main( String [ ] args )
11   {
12     int n = 0;  // declare and initialize variable
13
14     // prompt for input
15     String s = JOptionPane.showInputDialog( null,
16                                 "Enter an integer" );
17     System.out.println( "You entered " + s );
18
19     try
```



```
20     {
21       // attempt to convert the String to an int
22       n = Integer.parseInt( s );
23       System.out.println( "Conversion was successful." );
24     }
25     catch ( NumberFormatException nfe )
26     {
27       System.out.println( "Sorry. incompatible data." );
28       System.out.println( "\nOutput from getMessage: \n"
29                           + nfe.getMessage( ) );
30
31       System.out.println( "\nOutput from toString: \n"
32                           + nfe.toString( ) );
33       System.out.println( "\nOutput from printStackTrace: " );
34       nfe.printStackTrace( );
35     }
36
37     System.out.println( "\nn is " + n );
38   }
39 }
```

EXAMPLE 11.2 *DialogBoxInput* with *try* and *catch* Blocks

Figure 11.4 shows the output when the user inputs a valid value. In this case, the *try* block completes without an exception being generated, and the *catch* block is not executed. Execution of the program skips to line 37, where we print the value of the converted integer.

In contrast, Figure 11.5 shows the output when the user inputs an invalid value. In this case, the *parseInt* method throws a *NumberFormatException*. As

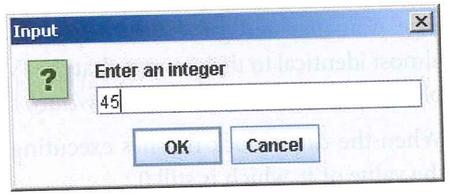
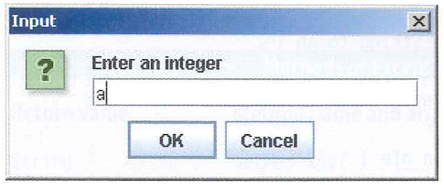


Figure 11.4
Output with No Exception Generated

```
You entered 45
Conversion was successful.

n is 45.
```

gure 11.5
tput When an ception Is Generated




```
You entered a
Sorry. incompatible data.

Output from getMessage:
For input string: "a"

Output from toString:
java.lang.NumberFormatException: For input string: "a"

Output from printStackTrace:
java.lang.NumberFormatException: For input string: "a"
        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 DialogBoxInput.main(DialogBoxInput.java:22)

n is 0
```

you can see, the *try* block stops executing at line 22, and we do not print the "Conversion was successful" message. Instead, the *catch* block is executed, and we print the values from the *getMessage*, *toString*, and *printStackTrace* methods of the *NumberFormatException* class. As you can see, the only difference between the return values from the *getMessage* and *toString* methods is that the *toString* method returns the exception class name, as well as the message. You may recognize the output of the *printStackTrace* method. It is almost identical to the message that the JVM printed to the console in Example 11.1, when we didn't have a *try/catch* block.

When the *catch* block finishes executing, we execute line 37, which prints the value of *n*, which is still 0.

COMMON ERROR TRAP

ailing to initialize a variable that is assigned a alue in a *try* block, then ccessed after the *try* lock, will generate a ompiler error.

Notice that we declare and initialize the variable *n* in line 12 before we enter the *try* block. If we do not initialize *n* and then try to access *n* after the *try/catch* blocks, we will receive the following compiler error:

```
variable n might not have been initialized
```

The error indicates that the only place where *n* is assigned a value is in the *try* block. If an exception occurs, the *try* block will be interrupted and we

might not ever assign a value to *n*. Initializing *n*'s value before entering the *try* block solves this problem.

Although Example 11.2 detects the exception, it merely prints a message that the exception occurred; it does nothing to recover from—or handle—the exception. Example 11.3 shows Version 3 of the *DialogBoxInput* class, which handles the exception by putting the *try* and *catch* blocks inside a *do/while* loop so that we continue to prompt the user for a value until the input is valid.

```
 1 /*  DialogBoxInput, Version 3
 2 *   Handling an exception
 3 *   Anderson, Franceschi
 4 */
 5
 6 import javax.swing.JOptionPane;
 7
 8 public class DialogBoxInput
 9 {
10   public static void main( String [ ] args )
11   {
12     // declare and initialize variables that will be
13     // assigned values in the try block
14     int n = 0;
15     boolean goodInput = false; // flag variable
16
17     // priming read
18     String s = JOptionPane.showInputDialog( null,
19                                "Enter an integer" );
20     do
21     {
22       try
23       {
24         // attempt to convert the String to an int
25         n = Integer.parseInt( s );
26         goodInput = true;
27       }
28       catch ( NumberFormatException nfe )
29       {
30         s = JOptionPane.showInputDialog( null,
31                                  s + " is not an integer. "
32                                  + "Enter an integer" );
33       }
34     } while ( !goodInput );
```

```
35
36      JOptionPane.showMessageDialog( null, "The integer is " + n );
37   }
38 }
```

EXAMPLE 11.3 Recovering from the Exception

On line 15, we declare a *boolean* variable, *goodInput*, which we will use as a flag variable to indicate whether the value entered by the user could be converted to an *int*. We initialize *goodInput* to *false*; its value will remain *false* until the *parseInt* method completes without an exception, at which time we will assign the value *true* (line 26). We use the value of *goodInput* in the *while* condition of the *do/while* loop (line 34) so that we continue to reprompt the user as long as the input is invalid. Remember that if an exception occurs in the *parseInt* method, line 26 will not be executed, so *goodInput* will be set to *true* only if no exception occurs. Now our application correctly detects and recovers from invalid user input and continues processing.

SOFTWARE ENGINEERING TIP

Write code to catch and handle exceptions generated by invalid user input. Although the methods of the *Exception* class are good debugging tools, they are not necessarily appropriate to use in the final version of a program. Always try to write code that is user-friendly.

In Figure 11.6, the user first enters an invalid value. We reprompt the user, and the user enters a valid value. Notice that the user doesn't see that the

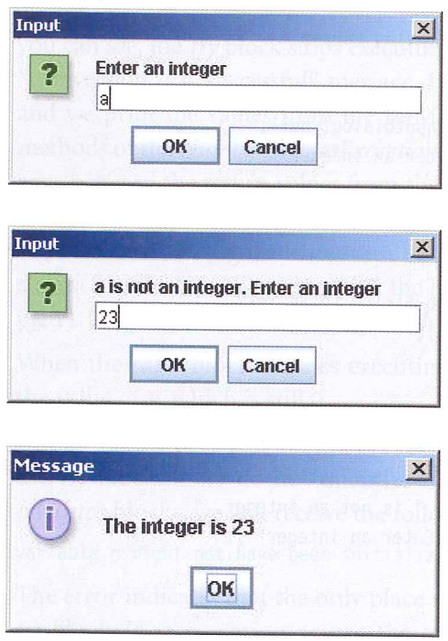


Figure 11.6
Handling the Exception

exception occurred, and also notice that our prompt clearly explains the problem to the user.

11.1.2 Catching Multiple Exceptions

If the code in the *try* block might generate multiple types of exceptions, we can provide multiple *catch* blocks, one for each possible exception. When an exception is generated, the JVM searches the *catch* blocks in order. The first *catch* block with a parameter that matches the exception thrown will execute; any remaining *catch* blocks will be skipped.

Remember that subclass objects are also objects of their superclasses, so an exception will match any *catch* block with a parameter that names any of its superclasses. For example, a *NumberFormatException* will match a *catch* block with a *RuntimeException* parameter, and all exceptions will match a *catch* block with an *Exception* parameter. Therefore, when coding several *catch* blocks, put the *catch* blocks for the specialized exceptions first, followed by more general exceptions.

SOFTWARE ENGINEERING TIP

Arrange *catch* blocks to handle the more specialized exceptions first, followed by more general exceptions.

Furthermore, after a *try* block and its associated *catch* blocks, you may optionally add a *finally* block, which will always be executed, whether an exception occurred or not. In the *finally* block, you can include some clean-up code. We will demonstrate a *finally* block when we read from a file later in this chapter.

Here is the syntax for using a *try* block, several *catch* blocks, and a *finally* block:

```
try
{
   // code that might generate an exception
}
catch ( Exception1Class e1 )
{
   // code to handle an Exception1Class exception
}
...
catch ( ExceptionNClass eN )
{
   // code to handle an ExceptionNClass exception
}
finally
{
   // code to execute regardless of whether an exception occurs
}
```

Again, the curly braces around the various blocks are required, whether these blocks contain zero, one, or more statements.

Example 11.4 shows a *Divider* class that catches two possible exceptions. First, we prompt the user for an integer (lines 20–22), then attempt to convert the input *String* to an *int* (lines 27–28). As we have seen, an invalid input value will generate a *NumberFormatException*. If we are successful in converting the input to an *int*, we use the *int* as the divisor in a division operation (lines 30–31). For simplicity, we use a predefined value (100) as the dividend. If the divisor is 0, line 31 will generate an *ArithmeticException*.

Thus, we put all this code into our *try* block (lines 25–34) and provide two *catch* blocks, one to handle the *NumberFormatException* (lines 35–40) and one to handle the *ArithmeticException* (lines 41–46).

```
 1 /* Divider
 2 *  Handling multiple exceptions
 3 *  Anderson, Franceschi
 4 */
 5
 6 import javax.swing.JOptionPane;
 7
 8 public class Divider
 9 {
10   public static void main( String [ ] args )
11   {
12     // declare and initialize variables
13     int divisor = 0;
14     int quotient = 0;
15     int dividend = 100;
16
17     // initialize flag variable
18     boolean goodInput = false;
19
20     // prompt for input
21     String s = JOptionPane.showInputDialog( null,
22                                 "Enter an integer divisor" );
23     do
24     {
25       try
26       {
```



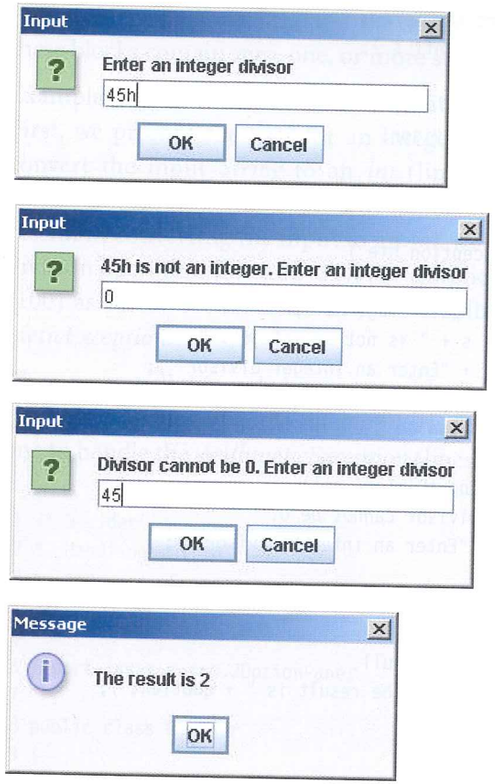
```
27         // attempt to convert the String to an int
28         divisor = Integer.parseInt( s );
29
30         // attempt the division
31         quotient = dividend / divisor;
32
33         goodInput = true;
34       }
35       catch ( NumberFormatException nfe )
36       {
37         s = JOptionPane.showInputDialog( null,
38                                 s + " is not an integer. "
39                                 + "Enter an integer divisor" );
40       }
41       catch ( ArithmeticException ae )
42       {
43         s = JOptionPane.showInputDialog( null,
44                                 "Divisor cannot be 0. "
45                                 + "Enter an integer divisor" );
46       }
47     } while ( !goodInput );
48
49     JOptionPane.showMessageDialog( null,
50                                 "The result is " + quotient );
51   }
52 }
```

EXAMPLE 11.4 The *Divider* Class

Figure 11.7 shows an execution of the program. At the first prompt, the user enters 45h, which causes the *parseInt* method to throw a *NumberFormatException*. The JVM starts scanning the *catch* blocks for a matching *Exception* parameter. The first *catch* block (lines 35–40) matches the exception thrown, so the code in that *catch* block is executed and we reprompt the user for a valid integer. We then skip the *catch* block for the *ArithmeticException* and line 47 is executed, which tests the value of *goodInput*. This will still be *false* at this point, so we repeat the *do/while* loop.

This time, the user enters 0, so the *parseInt* method successfully converts 0 to an *int*. Then line 31 is executed, where we attempt to divide by 0. This

Figure 11.7
Output from Example 11.4



causes an *ArithmeticException*. Again, the JVM starts scanning *catch* blocks for a matching exception parameter. The JVM skips the first *catch* block, but finds a match in the second block (lines 41–46), so we execute that *catch* block. This time we inform the user that the divisor cannot be 0 and reprompt for a new value. Finally, the user enters a valid, nonzero value; we are now able to both convert the input to an *int* and perform the division without generating any exceptions. In this case, both *catch* blocks are skipped, and we display the result in a dialog box (lines 49–50).

SOFTWARE ENGINEERING TIP

Whenever possible, use a simple *if/else* statement to detect an unchecked exception, rather than *try* and *catch* blocks. This will improve the performance of your code.

Having provided several examples of exceptions, we must also consider this: Not every problem needs to be addressed by generating an exception. As a matter of fact, generating and handling exceptions considerably slows down execution of your code due to the processing overhead. Often, for example when using Java's I/O classes, we will have no choice but to use *try* and *catch* blocks. In Example 11.4, however, we could have used a simple *if/else* statement to test the value of the divisor before attempting the

division. This would solve the problem simply and efficiently without generating an exception.

How do we know if a constructor or a method *throws* an exception and what type of exception it *throws*? As always, our best source of information is the Sun Microsystems website. After you have identified a constructor or a method that you would like to use, simply view its API in order to determine whether it *throws* any exceptions, and, if so, which ones.

REFERENCE POINT

Consult the Oracle Java website *www.oracle.com/technetwork/java* to see if a constructor or a method *throws* an exception and, if so, what type of exception.

11.1.3 User-Defined Exceptions

There will be times when we want to design our own exception class because the predefined Java exception classes do not fit our needs.

Suppose we are interested in designing a class encapsulating email addresses. We will call that class *EmailAddress*. To keep things simple, we will say that a legal email address is a *String* containing the @ character. In order to prevent instantiation of objects with illegal email addresses, we will design our *EmailAddress* constructor so that it throws an exception if its argument, a *String*, does not contain the @ character.

In order to do that, we first design an exception class that encapsulates an illegal email exception. We call our class *IllegalEmailException* and we will *throw* the exception when the argument to the *EmailAddress* constructor does not contain the @ character. Since Java already has an *IllegalArgumentException* class, we will define our *IllegalEmailException* class as a subclass of the *IllegalArgumentException* class.

More generally, when a user-defined exception class is defined as a subclass of an existing Java exception class, such as *Exception*, *NumberFormatException*, or *IllegalArgumentException*, our class inherits the functionality of the existing exception class, which simplifies coding the new class. We *extend* the *IllegalArgumentException* class so that we can associate a specific error message with the exception. We need to code only the constructor, and the constructor's job is to pass our message to the constructor of the superclass.

Thus, the general pattern of a user-defined exception class is:

```
public class ExceptionName extends ExistingExceptionClassName
{
  public ExceptionName( String message )
  {
    super( message );
  }
}
```

Example 11.5 shows our *IllegalEmailException* class.

```
1 /* The IllegalEmailException class
2    Anderson, Franceschi
3 */
4
5 public class IllegalEmailException extends IllegalArgumentException
6 {
7   public IllegalEmailException( String message )
8   {
9     super( message );
10  }
11 }
```

EXAMPLE 11.5 The *IllegalEmailException* Class

The constructor for the class is coded at lines 7 to 10; it takes a *String* parameter and simply passes it to the superclass constructor.

The pattern for a method that throws a user-defined exception is:

```
accessModifier dataType methodName( parameter list )
                        throws ExceptionName
{
  if ( parameter list is legal )
      // process the parameter list
  else
      throw new ExceptionName( "Some message here" );
}
```

The message we pass to the *ExceptionName* constructor will identify the type of error we detected. When a client program catches the exception, the client can call the *getMessage* method of the exception class in order to retrieve that message.

Example 11.6 shows our *EmailAddress* class.

```
1 /* The EmailAddress class
2    Anderson, Franceschi
3 */
4
5 public class EmailAddress
6 {
7   public static final char AT_SIGN = '@';
8   private String email;
9
```



```
10  public EmailAddress( String newEmail )
11                      throws IllegalEmailException
12  {
13    if ( newEmail.indexOf( AT_SIGN ) != - 1 )
14         email = newEmail;
15    else
16         throw new IllegalEmailException
17               ( "Email address does not contain " + AT_SIGN );
18  }
19
20  public String getHost( )
21  {
22    int index = email.indexOf( AT_SIGN );
23    return email.substring( index + 1, email.length( ) );
24  }
25 }
```

EXAMPLE 11.6 The *EmailAddress* Class

We coded the constructor at lines 10–18. We test if the constructor's argument, *newEmail*, contains the character *AT_SIGN* (a constant equal to @) at line 13. If it does, we proceed normally and initialize the instance variable *email* at line 14. If it does not, we throw an *IllegalEmailException* with the appropriate message at lines 16–17. In addition to the constructor, we coded the *getHost* method at lines 20–24. The *getHost* method returns the substring comprising the characters of *email* after *AT_SIGN*. Thus, for an email address of *myEmailAddress@yahoo.com*, the *getHost* method will return *yahoo.com*.

Now that we have built our own exception class and a class including a method that *throws* that exception, we are ready to use them in a client program. This is identical to using a predefined Java exception. Example 11.7 shows our *EmailChecker* class.

```
1 /* The EmailChecker class
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class EmailChecker
8 {
9   public static void main( String [ ] args )
10  {
```

```
11      Scanner scan = new Scanner( System.in );
12      System.out.print( "Enter your email address > " );
13      String myEmail = scan.next( );
14      try
15      {
16        EmailAddress address = new EmailAddress( myEmail );
17        System.out.println( "Your host is " + address.getHost( ) );
18      }
19      catch ( IllegalEmailException iee )
20      {
21        System.out.println( iee.getMessage( ) );
22      }
23    }
24  }
```

EXAMPLE 11.7 The *EmailChecker* Class

We ask the user to input an email address, *myEmail*, at lines 12–13. We then try to instantiate the *EmailAddress* object *address* at line 16, passing *myEmail* to the constructor. If *myEmail* does not contain the @ character, our *EmailAddress* constructor *throws* an *IllegalEmailException*, which this program *catches* at line 19. In this *catch* block, we print the message the *EmailAddress* constructor sent to the *IllegalEmailException* constructor. If *myEmail* contains the @ character, we continue executing inside the *try* block. Figure 11.8 shows two runs of this example;

Figure 11.8
Two Sample Runs of Example 11.7

```
Enter your email address > mary.jb.com
Email address does not contain @
```

```
Enter your email address > john@jb.com
Your host is jb.com
```


the first generates the exception, the second completes without generating an exception.

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of *try* and *catch* blocks. Click on the link for Chapter 11 to start the movie.

Skill Practice
with these end-of-chapter questions

11.10.1 Multiple Choice Exercises

Questions 1, 2, 3, 4

11.10.8 Technical Writing

Question 62

11.2 The *java.io* Package

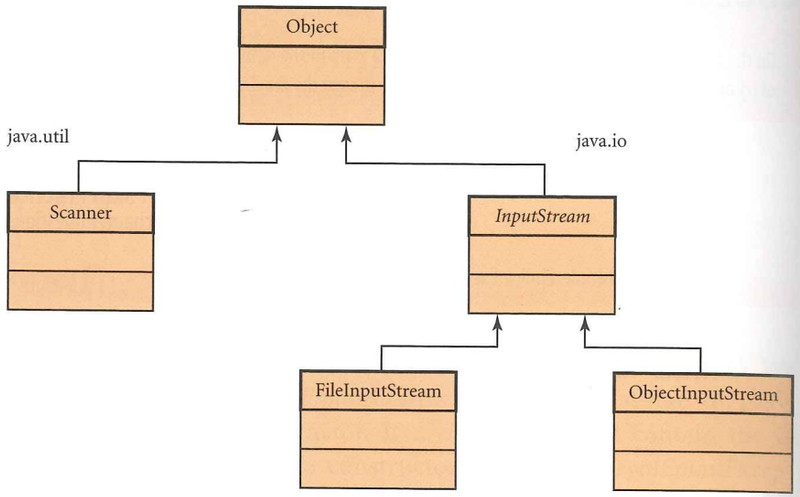
In addition to the *Scanner* class in the *java.util* package, Java provides a number of classes in the *java.io* package for reading from files and for writing to files. We will use only a few of those classes here. Table 11.2 describes a group of classes designed for data input.

TABLE 11.2 Selected Classes for Input

Input Classes	
Class	**Description**
`InputStream`	*Abstract* superclass representing an input stream of raw bytes
`FileInputStream`	Input stream to read raw bytes of data from files
`ObjectInputStream`	Class to read/recover objects from a file written using *ObjectOutputStream*

Figure 11.9 shows an inheritance hierarchy for the Java classes described in Table 11.2.

Figure 11.9
The Inheritance Hierarchy for Input Classes



REFERENCE POINT

The *Scanner* class is discussed in Chapter 3 and Chapter 6.

Table 11.3 describes a group of classes designed for data output, and Figure 11.10 shows an inheritance hierarchy for output Java classes in Table 11.3.

TABLE 11.3 Selected Output Classes

Output Classes	
Class	**Description**
`Writer`	*Abstract* superclass for output classes
`OutputStream`	*Abstract* superclass representing an output stream of raw bytes
`PrintWriter`	Convenient class to print basic data types, *Strings*, and objects
`FileOutputStream`	Output stream to write raw bytes of data to files
`ObjectOutputStream`	Class to write objects to a file

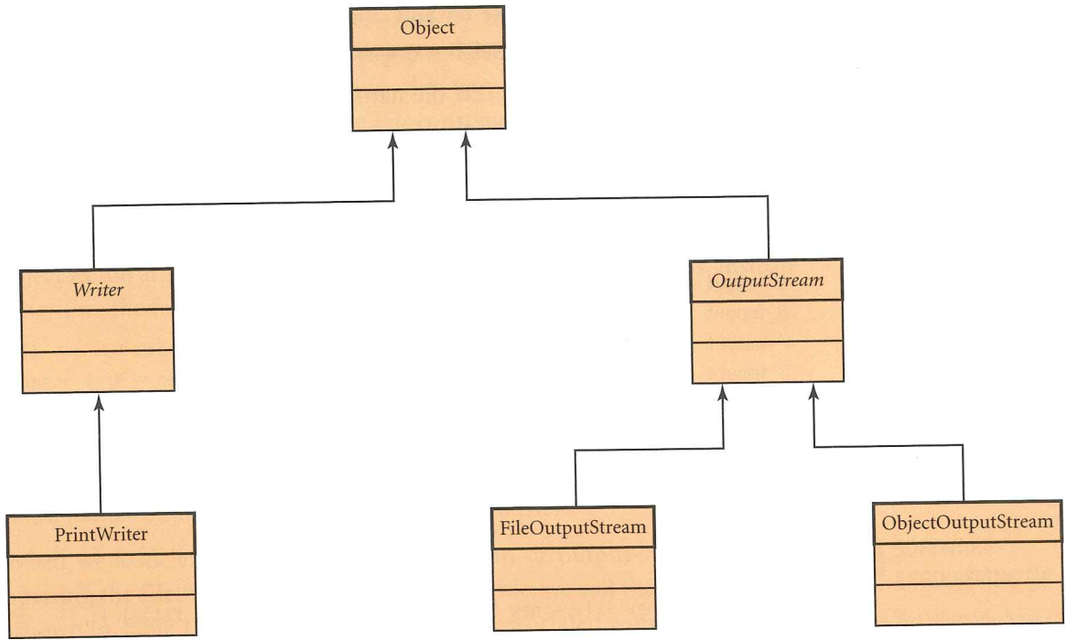


Figure 11.10
The Inheritance Hierarchy for Output Classes

11.3 Reading Text Files Using *Scanner*

Java supports two file types, text and binary. In text files, data is stored as characters; in binary files, data is stored as raw bytes. Different classes are used for writing and reading each file type. The type of a file is determined when the file is written to and depends on which classes were used to write to the file. Thus, to read from an existing file, you must know the file's type in order to select the appropriate classes.

In this section, we concentrate on text files.

In Chapter 6, we used the *Scanner* class to read from a text file. To avoid dealing with exceptions, we added the *throws IOException* clause to the header of *main*. If the file we wanted to read was not found, however, a *FileNotFoundException* was generated and the program terminated. Now

that we have seen how to use *try* and *catch* blocks, we can put the *Scanner* code inside a *try* block and catch any exceptions that may occur.

In Example 11.8, we read a text file named *datafile.txt*, one line at a time, and echo each line to the console.

```
1 /* Demonstrating how to read from a text file
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6 import java.io.File;
7 import java.io.FileNotFoundException;
8 import java.io.IOException;
9
10 public class ReadTextFile
11 {
12   public static void main( String [ ] args )
13   {
14     try
15     {
16        Scanner file = new Scanner( new File ( "dataFile.txt" ) );
17
18        while ( file.hasNext( ) ) // test for the end of the file
19        {
20           String stringRead = file.nextLine( );  // read a line
21           System.out.println( stringRead );  // print the line read
22        }
23
24        // release resources associated with dataFile.txt
25        file.close( );
26     }
27
28     catch ( FileNotFoundException fnfe )
29     {
30       System.out.println( "Unable to find dataFile.txt, exiting" );
31     }
32
33     catch ( IOException ioe )
34     {
35       ioe.printStackTrace( );
36     }
37   }
38 }
```

EXAMPLE 11.8 Demonstrating How to Read a Text File

Lines 5–8 import the four classes used by this program: *Scanner, File, FileNotFoundException,* and *IOException.* Line 16 instantiates a *Scanner* object from a *File* object, passing the name of the file to be read, *dataFile.txt*, as the argument to the *File* constructor.

The *File* object is used only as the argument of the *Scanner* constructor; it is not used anywhere else in the program. In this case, instead of creating a *File* object reference, many programmers prefer to use an anonymous *File* object as the argument of the *Scanner* constructor. In other words, instead of writing the following two statements:

```
File f = new File( "dataFile.txt" );
Scanner file = new Scanner( f );
```

we use the following single statement:

```
Scanner file = new Scanner( new File ( "dataFile.txt" ) );
```

It is a matter of preference as to which code you use.

When we associate a file with an input stream or output stream, we are **opening the file**. When we are finished with a file, we call the *close* method (shown in Table 11.4) to release the resources associated with the file. Calling the *close* method is optional. When the program finishes executing, all of its resources are released, including the resources of any unclosed files. Nevertheless, it is good practice, in general, to call the *close* method, especially if you will be opening a number of files (or opening the same file multiple times). In contrast, the standard input stream (*System.in*), the standard output stream (*System.out*), and the standard error stream (*System.err*) are open when the program begins. They are intended to stay open and should not be closed.

SOFTWARE ENGINEERING TIP

Close files when you have finished processing their data. Do not close the standard input, output, or error devices.

The *Scanner* constructor throws a *FileNotFoundException* if the file does not exist. We catch this exception at line 28. Since the program cannot continue without the file, we print a message to the user that we were not able to find the file and that the program is exiting (line 30). Thus, instead of

TABLE 11.4 The *close* Method of the *Scanner* Class

Return value	Method name and argument list
void	close( ) releases resources associated with an open input stream

facing a *FileNotFoundException* with a stack trace, the user now sees a friendly message explaining the problem.

If no exception is thrown, we begin to read the file. The *while* loop condition, at line 18, uses the *Scanner hasNext* method to test whether we have reached the end of the file. If not, *hasNext* returns *true*, and we execute the *while* loop body, which reads one line in the file by calling the *nextLine* method of the *Scanner* class, and echoes the *String* we read to the console.

We continue looping through the file until we receive a *false* return value from the *hasNext* method. We then exit the loop and call the *close* method on line 25 to release the file's resources.

Any attempt to access the file after it has been closed will generate an *IllegalStateException*. Also, if the *nextLine* method does not find data in the file, it throws a *NoSuchElementException*. We do not expect either of these exceptions to be thrown, and we would not be able to recover from those errors if they should occur, so we use *IOException* as the parameter of the *catch* block on lines 33–36, which will catch any I/O exception. Because we will not know the cause of the exception generated, we call the *printStackTrace* method, which outputs useful debugging information.

Note how we have ordered the two *catch* blocks. Remember that when an exception occurs, the *catch* blocks are scanned, in order, for a match between the *catch* block parameter and the type of exception that occurred. Because the *FileNotFoundException* is a subclass of *IOException*, a *FileNotFoundException* will also match a *catch* block for an *IOException*. Therefore, we need to put the *catch* block for the *FileNotFoundException* before the *catch* block for the *IOException*. This way, if the file *dataFile.txt* does not exist, the exception will match the first *catch* block, which handles the *FileNotFoundException*, and we will be able to print a meaningful message for the user.

Let's assume the file *dataFile.txt* contains the Gelett Burgess poem shown in Figure 11.11. When the program in Example 11.8 runs, it will produce the

Figure 11.11
Contents of *dataFile.txt*

```
I never saw a purple cow,
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one!
```



```
I never saw a purple cow,
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one!
```

Figure 11.12
Output of Example 11.8 When File is Found

```
Unable to find dataFile.txt, exiting
```

Figure 11.13
Output of Example 11.8 When File is Not Found

output shown in Figure 11.12 if the file is found, and the output in Figure 11.13 if the file is not found.

11.4 Writing and Appending to Text Files

In the previous section, we learned how to read data from a text file. But how did the data get into the file in the first place? It could be that someone put the data into the file using an editor, such as Notepad in Windows, or vi, pico, or emacs in Unix. Typing data into a file is convenient when the amount of data is small. But very often files contain a significant amount of data, typically written to the file by a computer program. For instance, a web server writes to log files to keep track of the visitors accessing the website, how they got to the website, the time they arrived, etc. If the web server comes under attack from a hacker, these log files can be consulted to determine where the hacker came from, who the hacker was, and other information.

In this section, we will learn how to write to a text file. But before going into the details, we must distinguish among several situations:

- Creating/writing to a new file, that is, the file does not exist.
- Writing to an existing file and replacing the contents of the file with new data.
- Writing to an existing file, but keeping the contents of the file and adding data at the end of the file. This is called **appending** to the file.

TABLE 11.5 Writing or Appending to an Existing or New File

Operation	If the file exists . . .	If the file does not exist . . .
write	the current contents of the file are deleted, and writing starts at the beginning of the file	the file is created and writing starts at the beginning of the file
append	data is added to the end of the file, keeping the existing contents	the file is created and writing starts at the beginning of the file

Java provides us with the necessary tools to perform all the preceding actions. Table 11.5 summarizes what will happen, depending on the action we perform.

11.4.1 Writing to Text Files

The data that we will need to write to files can be primitive data types, such as *ints*, *doubles*, or *booleans*, or even objects, such as *Strings*.

FileOutputStream, a subclass of the *OutputStream* class, is designed to write a stream of bytes to a file. It has several constructors, one of which takes a file name and a mode as its two arguments. The *boolean mode* variable specifies whether we are writing (*false*) or appending (*true*) to the file.

The *PrintWriter* class is designed for converting basic data types to characters and writing them to a text file. The *PrintWriter* class provides *print* and *println* methods for all primitive data types, as well as for *Strings* and objects. The *print* method writes the argument value to the file, whereas the *println* method writes the argument value to the file followed by a *newline* character. The constructors and method APIs are shown in Table 11.6.

Example 11.9 shows how the *PrintWriter* class can be used to write *Strings* and primitive data types to a text file named *grades.txt*.

```
/* Demonstrating how to write basic data types to a text file
   Anderson, Franceschi
*/

import java.io.FileOutputStream;
import java.io.PrintWriter;
```


TABLE 11.6 Useful Classes, Constructors, and Methods for Writing to a Text File

Classes, Constructors, and Methods for Writing to a Text File		
Class	**Constructors**	**Exceptions thrown**
`FileOutputStream`	`FileOutputStream( String filename, boolean mode )` constructs a *FileOutputStream* object from a *String* representing the name of a file; if *mode* is *false*, we will write to the file; if *mode* is *true*, we will append to the file	`FileNotFoundException`
`PrintWriter`	`PrintWriter( OutputStream os )` constructs a *PrintWriter* object from the *OutputStream* object	None
	Method APIs	

`PrintWriter`	`void print( int i )`
	`void print( double d )`
	`void print( char c )`
	`void print( boolean b )`
	`...`
	`void println( int i )`
	`void println( double d )`
	`...`
	writes the argument to a text file
		None

`PrintWriter`	`void close( )` releases the resources associated with the *PrintWriter* object	None

```
import java.io.FileNotFoundException;

public class WriteGradeFile
{
  public static void main( String [ ] args )
  {
    try
    {
      FileOutputStream fos = new FileOutputStream
              ( "grades.txt", false );
      // false means we will be writing to grades.txt,
```

```
18        // rather than appending to it
19
20        PrintWriter pw = new PrintWriter( fos );
21
22        // write data to the file
23        pw.print( "CS130: " );
24        pw.println( 95 );
25        pw.print( "Letter grade: " );
26        pw.println( 'A' );
27        pw.print( "Current GPA: " );
28        pw.println( 3.68 );
29        pw.print( "Successful student as of Spring semester: " );
30        pw.println( true );
31
32        // release the resources associated with grades.txt
33        pw.close( );
34      }
35
36      catch ( FileNotFoundException fnfe )
37      {
38        System.out.println( "Unable to find grades.txt" );
39      }
40    }
41  }
```

EXAMPLE 11.9 Writing Primitive Data Types and *Strings* to a Text File

Lines 15–16 instantiate a *FileOutputStream* object to write to the file *grades.txt*. The *FileOutputStream* constructor throws a *FileNotFoundException*. Our code catches that exception at lines 36–39. This is the only *catch* block because the methods of the *PrintWriter* class do not throw exceptions.

Line 20 instantiates a *PrintWriter* object, which we will use to write to the file. At lines 22 to 30, using the *PrintWriter* object *pw*, we call the methods *print* and *println*, passing various *String* and primitive data types arguments (*int*, *char*, *double*, and *boolean*) to be written to the file. When we want a *newline* character appended to the output, we call *println*, rather than *print*.

After this program is executed, the file *grades.txt* will contain the data shown in Figure 11.14. Because we wrote all the output to the file *grades.txt*, there is no output to the console.


```
CS130: 95
Letter grade: A
Current GPA: 3.68
Successful student as of Spring semester: true
```

Figure 11.14
Contents of *grades.txt*

11.4.2 Appending to Text Files

Appending text to a file is similar to writing text; the only difference is that the second argument of the *FileOutputStream* constructor is *true*, instead of *false*.

Example 11.10 shows how these *FileOutputStream* and *PrintWriter* classes can be used in a Java program to append text to our file named *grades.txt*.

```
 1 /* Demonstrating how to append basic data types to a text file
 2    Anderson, Franceschi
 3 */
 4
 5 import java.io.FileOutputStream;
 6 import java.io.PrintWriter;
 7 import java.io.FileNotFoundException;
 8
 9 public class AppendGradeFile
10 {
11   public static void main( String [ ] args )
12   {
13     try
14     {
15       FileOutputStream fos = new FileOutputStream
16                    ( "grades.txt", true );
17       // true means we will be appending to grades.txt,
18       // rather than writing to it
19
20       PrintWriter pw = new PrintWriter( fos );
21
22       // write data to the file
23       pw.print( "CS230: " );
24       pw.println( 87 );
25       pw.print( "Letter grade: " );
26       pw.println( 'B' );
```

```
27        pw.print( "Current GPA: " );
28        pw.println( 3.53 );
29        pw.print( "Successful student as of Fall semester: " );
30        pw.println( true );
31
32        // release the resources associated with grades.txt
33        pw.close( );
34      }
35
36      catch ( FileNotFoundException fnfe )
37      {
38        System.out.println( "Unable to find grades.txt" );
39      }
40    }
41 }
```

COMMON ERROR TRAP

Opening a file for writing will cause the existing file data to be deleted. If you intend to add new data to a file while maintaining the original contents, open the file for appending.

EXAMPLE 11.10 Appending Primitive Data Types to a Text File

Example 11.10 is similar to Example 11.9. The major difference is that when we instantiate the *FileOutputStream* object (lines 15–16), the second argument is *true*, which means that we will append to the file *grades.txt*, instead of writing to it. If the file *grades.txt* exists, we will start writing at the end of its current contents, whereas if the file does not exist, it will be created. In this program, we append four additional lines to the file (lines 21–29).

If before this program is executed the file *grades.txt* contains the data shown in Figure 11.14, then after this program is executed the file *grades.txt* will contain the data shown in Figure 11.15; again, there is no output to the console.

Figure 11.15
New Content of *grades.txt*

```
CS130: 95
Letter grade: A
Current GPA: 3.68
Successful student as of Spring semester: true
CS230: 87
Letter grade: B
Current GPA: 3.53
Successful student as of Fall semester: true
```


11.5 Reading Structured Text Files

Sometimes a text file is organized so that each line represents data related to a particular record or object. For instance, an airline company could have data stored in a file where each line represents a flight segment, with the following comma-separated data:

- flight number
- origin airport
- destination airport
- number of passengers
- average ticket price

Such a file could contain the following data:

```
AA123,BWI,SFO,235,239.5
AA200,BOS,JFK,150,89.3
AA900,LAX,CHI,201,201.8
...
```

As we read the file, we should **parse** each line, that is, separate the line into the individual pieces of data (flight number, origin airport, etc.), called **tokens**. In this case, the comma is the **delimiter**; that is, a comma separates one token from the next. We will store the tokens from each line into a corresponding *FlightRecord* object.

The *Scanner* class, in the *java.util* package, can be used to parse *Strings*, that is, to separate *Strings* into tokens.

11.5.1 Parsing a *String* Using *Scanner*

A constructor of the class *Scanner*, which takes a *String* to parse, is shown in Table 11.7.

The default delimiters are the whitespace characters. You can specify other delimiters through the *useDelimiter* method, which accepts as its argument a *String* representing a **regular expression**. Regular expressions allow you to specify a pattern against which to match sequences of characters using

TABLE 11.7 Useful Constructor and Methods of the *Scanner* Class for Parsing a *String*

Constructor		Exceptions Thrown
`Scanner( String source )`		None
constructs a *Scanner* object that produces tokens from the specified *String*		
Method APIs		
Return value	**Method name and argument list**	
`Scanner`	`useDelimiter( String pattern )`	None
sets this *Scanner* object's delimiting pattern to the specified pattern		
`boolean`	`hasNext( )`	`IllegalStateException`
returns *true* if this *Scanner* object has another token in its input		
`boolean`	`hasNextInt( )`	`IllegalStateException`
`boolean`	`hasNextDouble( )`	
`boolean`	`hasNextBoolean( )`	
.....		
return *true* if this *Scanner* object has another token in its input of the specified data type		
`String`	`next( )`	`NoSuchElementException` `IllegalStateException`
returns the next token for this *Scanner* object		
`int`	`nextInt( )`	`InputMismatchException`
`double`	`nextDouble( )`	`NoSuchElementException`
`boolean`	`nextBoolean( )`	`IllegalStateException`
.....		
return the next token for this *Scanner* object as the specified data type		

standard characters as well as **meta-characters**, which have special meanings. Further discussion of regular expressions is outside the scope of this book. For our purposes, we can specify a delimiter consisting of a single or multiple specific characters as a simple *String* argument for the *useDelimiter* method. For example, to parse a *String* like *AA123,BWI,SFO,235,239.5*, we call the *useDelimiter* method and pass a comma as its argument.

Some useful methods of the *Scanner* class for parsing a *String* are shown in Table 11.7.

Example 11.11 shows how the *Scanner* class can be used in a Java program to parse a *String*.

```
/* Demonstrating how to parse a String with the Scanner class
   Anderson, Franceschi
*/

import java.util.Scanner;
import java.util.InputMismatchException;
import java.util.NoSuchElementException;

public class UsingScannerToParseAString
{
  public static void main( String [ ] args )
  {
   String flightRecord1 = "AA123,BWI,SFO,235,239.5";
   Scanner parse = new Scanner( flightRecord1 );
   // set the delimiter to a comma
   parse.useDelimiter( "," );

   try
   {
     System.out.println( parse.next( ) );
     System.out.println( parse.next( ) );
     System.out.println( parse.next( ) );
     System.out.println( parse.nextInt( ) );
     System.out.println( parse.nextDouble( ) );
   }
   catch ( InputMismatchException ime )
   {
      System.out.println( "Error in data format" );
   }
   catch ( NoSuchElementException nse )
   {
      System.out.println( "No more tokens" );
   }
   catch ( IllegalStateException ise )
   {
      ise.printStackTrace ( );
   }
   parse.close( );
  }
}
```

EXAMPLE 11.11 Demonstrating How to Parse a *String* with the *Scanner* Class

Line 5 imports the *Scanner* class. The *flightRecord1 String* variable is declared and initialized at line 13. At line 14, the *Scanner* object *scan* is instantiated using the constructor with a *String* argument, *flightRecord1*, the *String* that we want to tokenize. At lines 15–16, we call the *useDelimiter* method to set the delimiter to a comma.

At lines 20–24, we call the *next*, *nextInt*, and *nextDouble* methods to retrieve the five tokens of *flightRecord1* and process them, echoing them to the console. Since we know the format of the *String flightRecord1*, we can call the appropriate method based on what data type we expect to find at that position in the *String*. Although these methods throw exceptions, these exceptions are unchecked exceptions and thus the *try* and *catch* blocks are not mandatory. The *nextInt*, *nextDouble*, … methods will *throw* an *InputMismatchException* if the token retrieved cannot be converted to the expected data type. They will *throw* a *NoSuchElementException* if there are no more tokens to retrieve. Finally, they will *throw* an *IllegalStateException* if the *Scanner* object has been closed. We *catch* all these exceptions at lines 26–37, from the most specific to the most general and in the order in which they are most likely to be thrown.

When the program in Example 11.11 runs, it will produce the output shown in Figure 11.16.

If we want to process all the tokens as *Strings* with a loop construct, we can call the *hasNext* method to check if there are more tokens to process as in the following:

```
while ( parse.hasNext( ) )
  System.out.println( parse.next( ) );
```

Eventually, when all tokens have been retrieved, the *hasNext* method returns *false*, which causes us to exit the *while* loop.

REFERENCE POINT

You can read more about the *Scanner* class on *www.oracle.com/technetwork/java*.

Figure 11.16
Output from Example 11.11

```
AA123
BWI
SFO
235
239.5
```


We can also test if the next token to be retrieved is of a given data type (before we attempt to retrieve it) with the methods *hasNextInt*, *hasNextDouble*, … as in the following:

```
if ( parse.hasNextInt( ) )
   int numPassengers = parse.nextInt( );
```

11.5.2 Reading Structured Data Using *Scanner*

Now we are ready to put the previous two ideas together: Let's say that we have a file named *flights.txt* containing many flight records, and we want to read the data into variables. Again, suppose that the file is in the same format as the *flightRecord1 String* in Section 11.4.1; that is, it looks like the following:

```
AA123,BWI,SFO,235,239.5
AA200,BOS,JFK,150,89.3
AA900,LAX,CHI,201,201.8
...
```

where each line represents a flight segment with the following comma-separated data: flight number, origin airport, destination airport, number of passengers, and average ticket price.

First, we will build a class called *FlightRecord*, encapsulating a flight record as reflected by the data in the file. Each line read from the file will be parsed and used to instantiate a *FlightRecord* object. Since we do not know how many lines (i.e., how many flight records) are in the file, we will place all the flight records into an *ArrayList* object as opposed to a fixed-length array.

Our simplified *FlightRecord* class is shown in Example 11.12. It has only a constructor (lines 17–35) and the *toString* method (lines 37–49).

```
 1 /* The FlightRecord class
 2    Anderson, Franceschi
 3 */
 4
 5 import java.text.DecimalFormat;
 6
 7 public class FlightRecord
 8 {
 9   public static final DecimalFormat MONEY
10                          = new DecimalFormat( "$###.00" );
11   private String flightNumber;      // ex. = AA123
12   private String origin;            // origin airport; ex. = BWI
13   private String destination;       // destination airport; ex. = SFO
14   private int numPassengers;        // number of passengers
```

```
15   private double avgTicketPrice;   // average ticket price
16
17   /** Constructor
18   *  @param  startFlightNumber   flight number
19   *  @param  startOrigin         origin airport
20   *  @param  startDestination    destination airport
21   *  @param  startNumPassengers  number of passengers
22   *  @param  startAvgTicketPrice average ticket price
23   */
24   public FlightRecord( String startFlightNumber,
25                        String startOrigin,
26                        String startDestination,
27                        int startNumPassengers,
28                        double startAvgTicketPrice )
29   {
30     flightNumber = startFlightNumber;
31     origin = startOrigin;
32     destination = startDestination;
33     numPassengers = startNumPassengers;
34     avgTicketPrice = startAvgTicketPrice;
35   }
36
37   /** toString
38   * @return flight number, origin, destination,
39   *         number of passengers, and average ticket price
40   */
41   public String toString( )
42   {
43     return "Flight " + flightNumber
44            + ": from " + origin
45            + " to " + destination
46            + "\n\t" + numPassengers + " passengers"
47            + "; average ticket price: "
48            + MONEY.format( avgTicketPrice );
49   }
50   // accessors, mutators, and other methods ...
51 }
```

EXAMPLE 11.12 The *FlightRecord* Class

Example 11.13 shows our client class, which will read the file *flights.txt*, parse each line using *Scanner*, then instantiate a *FlightRecord* object and add it to the *ArrayList* named *listFlightRecords*.


```
1 /* Reading structured data from a text file
2    Anderson, Franceschi
3 */
4
5 import java.io.File;
6 import java.io.FileNotFoundException;
7 import java.io.IOException;
8 import java.util.Scanner;
9 import java.util.InputMismatchException;
10 import java.util.ArrayList;
11
12 public class ReadFlights
13 {
14    public static void main( String [ ] args )
15    {
16      // instantiate ArrayList to hold FlightRecord objects
17      ArrayList<FlightRecord> listFlightRecords =
18                              new ArrayList<FlightRecord>( );
19
20      try
21      {
22        Scanner file = new Scanner( new File( "flights.txt" ) );
23
24        while ( file.hasNext( ) ) // test for the end of the file
25        {
26          // read a line
27          String stringRead = file.nextLine( );
28
29          // process the line read
30          Scanner parse = new Scanner( stringRead );
31          parse.useDelimiter( "," );
32          String flightNumber = parse.next( );
33          String origin = parse.next( );
34          String destination = parse.next( );
35
36          try
37          {
38            int numPassengers = parse.nextInt( );
39            double avgTicketPrice = parse.nextDouble( );
40
41            FlightRecord frTemp = new FlightRecord(
42                        flightNumber, origin, destination,
43                        numPassengers, avgTicketPrice );
44
```

```
45              // add FlightRecord obj to listFlightRecords
46              listFlightRecords.add( frTemp );
47           }
48
49           catch ( InputMismatchException ime )
50           {
51              System.out.println( "Error in flight record: "
52                       + stringRead + "; record ignored" );
53           }
54        }
55
56        // release resources associated with flights.txt
57        file.close( );
58     }
59
60     catch ( FileNotFoundException fnfe )
61     {
62        System.out.println( "Unable to find flights.txt" );
63     }
64
65     catch ( IOException ioe )
66     {
67        ioe.printStackTrace( );
68     }
69
70     // print the FlightRecords read
71     for ( FlightRecord flight : listFlightRecords )
72        System.out.println( flight );
73  }
74 }
```

EXAMPLE 11.13 Demonstrating How to Read Structured Data from a File

Lines 5–10 import the classes needed for input, parsing, and exception handling, as well as *ArrayList*. The *FlightRecord* class is also used in this program, but is assumed to be in the same directory as the *ReadFlights* class.

This example is similar to Example 11.8 in that we instantiate one *Scanner* object (line 22) to read from the file. We use the *nextLine* method to read a line at a time (line 27), and handle both *FileNotFoundException* (lines 60–63) and *IOException* (lines 65–68). The difference is that instead of just printing each line of text that we read from the file, we parse that line using


```
AA123,BWI,SFO,235,239.5
AA200,BOS,JFK,150,89.3
AA900,LAX,CHI,201,201.8
```

Figure 11.17
Contents of *flights.txt*

```
Flight AA123: from BWI to SFO
        235 passengers; average ticket price: $239.50
Flight AA200: from BOS to JFK
        150 passengers; average ticket price: $89.30
Flight AA900: from LAX to CHI
        201 passengers; average ticket price: $201.80
```

Figure 11.18
Output from *ReadFlights.java*

another *Scanner* object and pass the data we extracted as our arguments to the *FlightRecord* constructor to instantiate a *FlightRecord* object (lines 41–43). One intermediate step is to read the tokens representing the number of passengers and average ticket price as an *int* and a *double* using the *nextInt* and *nextDouble* methods, respectively. We catch any *InputMismatchException* that might occur during the conversion (lines 49–53). If an exception does occur, we print an error message and skip the record with the invalid data.

The *FlightRecord* object is then added to the *ArrayList listFlightRecords* at lines 45–46. The *ArrayList listFlightRecords* is declared and instantiated at lines 16–18, before the *try* block so that *listFlightRecords* is available for printing the *FlightRecord* objects at lines 70–72, after we finish reading the file. If the *flights.txt* file contains the data shown in Figure 11.17, the program will produce the output shown in Figure 11.18.

CODE IN ACTION

On the CD-ROM included with this book, you will find a Flash movie with a step-by-step illustration of reading from and writing to a text file. Click on the link for Chapter 11 to start the movie.

Skill Practice
with these end-of-chapter questions

11.10.1 Multiple Choice Exercises

Questions 5, 6, 7, 8, 9, 10

11.10.2 Reading and Understanding Code

Questions 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23

11.10.3 Fill In the Code

Questions 27, 28, 29, 30, 31, 32, 33

11.10.4 Identifying Errors in Code

Questions 34, 35, 36

11.10.5 Debugging Area

Questions 37, 38, 39, 40, 41, 42

11.10.6 Write a Short Program

Questions 43, 44, 45, 46, 47, 48, 49, 50, 51

11.6 Programming Activity 1: Reading from a Structured Text File

In this activity, you will read from a text file using an end-of-file controlled *while* loop performing this activity:

> Read a text file containing transaction items for a bank account. Loop through all the transaction items and calculate the new balance of the bank account. Assume that we do not know the number of transaction items (i.e., lines) in the file.

The framework will display the current transaction and current balance so that you can check the correctness of your code as the program executes.

For example, Figure 11.19 demonstrates the animation: We are currently scanning a check for the amount of $200.00. The original balance was $0.00 and the new balance is –$200.00. Ideally, this is not your bank account.

Instructions

In the Chapter 11 directory on the CD-ROM accompanying this book, you will find the Programming Activity 1 directory. Copy the contents of the directory onto a directory on your disk.

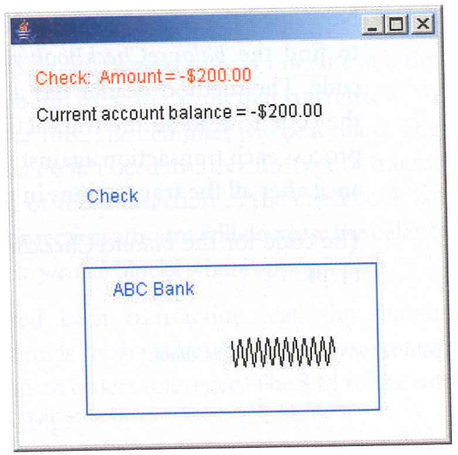


Figure 11.19
Animation Showing a $200 Check and the New Balance

```
Check # 13 :–200.00
Check # 14 :–100.00
Withdrawal June 12 :–200.00
Withdrawal June 17 :–400.00
Withdrawal June 23 :–100.00
Deposit :4000.00
Deposit :100.00
Something else :–1000.00
Check # 16 :–500.00
Check # 15 :–100.00
```

Figure 11.20
Contents of the *transactions.txt* File

- Open the file *transactions.txt* with a text editor. You will see that each line contains a transaction name and transaction amount separated by a colon, as shown in Figure 11.20.
- Note that the transaction amounts are positive or negative. For instance:
  - A check or a withdrawal has a negative amount.
  - A deposit has a positive amount.
  - An unknown transaction has either a positive or negative amount.

- Now open the *Accounting.java* file. Search for five asterisks (*****) to find the *balanceCheckBook* method where you will add your code. The method header has already been coded for you. Write the code to read all the transactions from the file *transactions.txt*, process each transaction against the account, and calculate the balance after all the transactions in that file have been processed.

  The code for the *balanceCheckBook* method is shown in Example 11.14.

```
public void balanceCheckBook( )
{
  // ***** Write the body of this method *****
  //
  //  Using a while loop, read the file transactions.txt
  // The file transactions.txt contains money
  // transactions between you and your bank
  //
  //  You will need to call the method animate inside
  //  the body of the loop reading the file contents
  //
  // The animate method takes 3 arguments:
  //    a String, representing the type of transaction
  //    a double, representing the transaction money amount
  //    a double, representing the new checkbook balance
  // So if these 3 variables are:
  //     transactionName, currentAmount, and balance,
  //  then the call to animate will be:
  //
  //  animate( transactionName, currentAmount, balance );
  //
  // You should make that call in the body of your while
  // loop, after you have updated the checkbook balance
  //


  //
  // end of student code
  //
}
```

EXAMPLE 11.14 ***balanceCheckBook* Method in *Accounting.java***

- Begin with a checkbook balance of 0.00.
- To process the transactions, you will need to read one line at a time from the *transactions.txt* file and parse the *String* that you retrieve. You can use the *Scanner* class for this. The delimiter will be a colon. Then process the transaction; you do not need to check the type of transaction. Just add the amount of the transaction to the checkbook balance. Adding a negative transaction amount will decrease the balance, as expected. Be sure to use *try/catch* blocks where appropriate.
- After you have processed each transaction, call the *animate* method. This method belongs to the *Accounting* class, so you will call *animate* without using an object reference. The API of the *animate* method is the following:

  ```
  public void animate( String currentTransaction,
                       double currentAmount,
                       double currentBalance )
  ```

  As you can see, the *animate* method takes three arguments: *currentTransaction* is the transaction name ("*Deposit*," for example), *currentAmount* is the amount of the transaction (*-45.00*, for example), and *currentBalance* is the current balance of the checkbook. Assuming that you have a *String* variable called *transactionName*, a *double* variable called *amount*, and another *double* called *balance*, a call to *animate* will look like the following:

  ```
  animate( transactionName, amount, balance );
  ```

- When you call *animate*, the window will display the current transaction graphically. It will also display the transaction amount (red if negative, blue if positive), and the current checkbook balance (in black). By adding the previous checkbook balance to the current transaction amount, you will be able to compute in your head what the current checkbook balance should be and determine if your program is working correctly.
- When you reach the end of the file, print the final balance and write it to a file named *balance.txt*.

To test your code, compile and run the *Accounting* application.

If you have time . . .

- Modify the file *transactions.txt* by deleting or adding transactions manually with a text editor. Run the program again and check that your code still gives the correct result.
- Using a text editor, modify the file *transactions.txt* by entering a positive amount to all transactions. Change the *balanceCheckBook* method so that it determines which transactions are positive and which are negative. Run the program again and check that your code still gives the correct result.

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you coded the priming read correctly.
- Verify that you coded the call to *animate* at the proper time.
- Verify that you coded the condition for exiting the loop correctly.
- Verify that you coded the body of the loop correctly.

DISCUSSION QUESTIONS ?

1. What exceptions can occur during this program?
2. Explain why we use the *Scanner* class.

11.7 Reading and Writing Objects to a File

Throughout this book, we have emphasized the benefits of object-oriented programming. Just as we can write text and primitive data types to a file and subsequently read them from the file, we can also write objects to a file and subsequently read them as objects. This is convenient for two reasons:

- We can write these objects directly to a file without having to convert the objects to primitive data types or *Strings*.
- We can read the objects directly from a file, without having to read *Strings* and convert these *Strings* to primitive data types in order to instantiate objects.

To read objects from a file, the contents of the file must have been written as objects. So our first order of business should be to learn how to write objects to a file.

11.7.1 Writing Objects to Files

The *ObjectOutputStream* class, coupled with the *FileOutputStream* class, provides the functionality to write objects to a file. We are familiar with the *FileOutputStream* class, which we used in the last section to write structured data to a text file.

The *ObjectOutputStream* class, which is a subclass of *OutputStream*, provides a convenient way to write objects to a file. Its *writeObject* method takes one argument—the object to be written.

The classes, constructors, and methods we will use are shown in Table 11.8.

We will use the *FlightRecord* class developed earlier in the chapter. However, in order for an object to be written to a file (and later to be read using the

TABLE 11.8 Useful Classes, Constructors, and Methods for Writing Objects to a File

Classes, Constructors, and Methods for Writing Objects to a File		
Class	**Constructor**	**Exceptions thrown**
`FileOutputStream`	`FileOutputStream( String filename, boolean mode )`	`FileNotFoundException`
	constructs a *FileOutputStream* object from a *String* representing the name of a file; if *mode* is *false*, we will write to the file; if *mode* is *true*, we will append to the file	
`ObjectOutputStream`	`ObjectOutputStream( OutputStream out )`	`IOException`
	creates an *ObjectOutputStream* that writes to the *OutputStream out*	
	Method API	
`ObjectOutputStream`	`void writeObject( Object o )`	`IOException`
	writes the object argument to a file. That object must be an instance of a class that implements the *Serializable* interface. Otherwise, a run-time exception will be generated	

ObjectInputStream class), that object must implement the *Serializable* interface. When an object implements the *Serializable* interface, its state can be converted to a byte stream to be written to a file, such that this byte stream can be converted back into a copy of the object when read from the file. Therefore, our modified *FlightRecord2* class will implement *Serializable*, which is in the *java.io* package.

The *Serializable* interface has no methods to implement. As a result, the only things we have to worry about when writing a class implementing *Serializable* are the following:

- the *import* statement
- the class header showing the class *implements Serializable*

Example 11.15 shows the *FlightRecord2* class. This class is identical to the *FlightRecord* class except that it *imports Serializable* (line 5) and *implements* the *Serializable* interface (line 8).

```
1 /* The FlightRecord2 class
2    Anderson, Franceschi
3 */
4
5 import java.io.Serializable;
6 import java.text.DecimalFormat;
7
8 public class FlightRecord2 implements Serializable
9 {
10   public static final DecimalFormat MONEY
11                        = new DecimalFormat( "$###.00" );
12   private String flightNumber;      // ex. = AA123
13   private String origin;            // origin airport; ex. = BWI
14   private String destination;       // destination airport; ex. = SFO
15   private int numPassengers;        // number of passengers
16   private double avgTicketPrice;    // average ticket price
17
18   /** Constructor
19   *  @param  startFlightNumber   flight number
20   *  @param  startOrigin         origin airport
21   *  @param  startDestination    destination airport
22   *  @param  startNumPassengers  number of passengers
23   *  @param  startAvgTicketPrice average ticket price
24   */
25   public FlightRecord2( String startFlightNumber,
26                         String startOrigin,
```



```
27                         String startDestination,
28                         int startNumPassengers,
29                         double startAvgTicketPrice )
30   {
31     flightNumber = startFlightNumber;
32     origin = startOrigin;
33     destination = startDestination;
34     numPassengers = startNumPassengers;
35     avgTicketPrice = startAvgTicketPrice;
36   }
37
38   /** toString
39   * @return flight number, origin, destination,
40   *         number of passengers, and average ticket price
41   */
42   public String toString( )
43   {
44     return "Flight " + flightNumber
45            + ": from " + origin
46            + " to " + destination
47            + "\n\t" + numPassengers + " passengers"
48            + "; average ticket price: "
49            + MONEY.format( avgTicketPrice );
50   }
51   // accessors, mutators, and other methods ...
52 }
```

EXAMPLE 11.15 The *FlightRecord2* Class

Example 11.16 shows how the *FileOutputStream* and *ObjectOutputStream* classes can be used in a Java program to write *FlightRecord2* objects to a file named *objects*.

```
1 /* Demonstrating how to write objects to a file
2    Anderson, Franceschi
3 */
4
5 import java.io.FileOutputStream;
6 import java.io.ObjectOutputStream;
7 import java.io.FileNotFoundException;
8 import java.io.IOException;
9
10 public class WritingObjects
11 {
```

```
12    public static void main( String [ ] args )
13    {
14     // instantiate the objects
15     FlightRecord2 fr1 = new FlightRecord2( "AA31", "BWI", "SFO",
16                                            200, 235.9 );
17     FlightRecord2 fr2 = new FlightRecord2( "CO25", "LAX", "JFK",
18                                            225, 419.9 );
19     FlightRecord2 fr3 = new FlightRecord2( "US57", "IAD", "DEN",
20                                            175, 179.5 );
21
22     try
23     {
24      FileOutputStream fos = new FileOutputStream
25                                  ( "objects", false );
26              // false means we will write to objects
27
28      ObjectOutputStream oos = new ObjectOutputStream( fos );
29
30      // write the objects to the file
31      oos.writeObject( fr1 );
32      oos.writeObject( fr2 );
33      oos.writeObject( fr3 );
34
35      // release resources associated with the objects file
36      oos.close( );
37     }
38
39     catch ( FileNotFoundException fnfe )
40     {
41       System.out.println( "Unable to write to objects" );
42     }
43
44     catch ( IOException ioe )
45     {
46       ioe.printStackTrace( );
47     }
48   }
49 }
```

EXAMPLE 11.16 Writing Objects to a File

Lines 14–20 declare and instantiate three *FlightRecord2* objects that we will write to the *objects* file.

Lines 24 and 25 instantiate a *FileOutputStream* object for writing to the *objects* file, then line 28 instantiates an *ObjectOutputStream* object, which we will use to write the *FlightRecord2* objects to the file.

At lines 30–33, using the *ObjectOutputStream* object *oos*, we call the *writeObject* method, passing the three *FlightRecord2* objects we instantiated. The *writeObject* method takes a *Serializable* object as its parameter, here a *FlightRecord2* object, and writes it to the file in such a way that the stream of bytes can be read using the *readObject* method from the *ObjectInputStream* class. Both the *ObjectOutputStream* constructor and the *writeObject* method can throw an *IOException*, which will be caught at line 44.

After this program is executed, the *objects* file will contain a representation of the three *FlightRecord2* objects.

One more note about writing objects to files: A file containing objects can be quite large. Not only does the object data get written to the file, but also the name of the class, a description of each data field, and other information needed to reconstruct the objects when the file is subsequently read.

The *writeObject* method, however, does not write any *static* class variables to the file. Thus, you may consider declaring any constants as *static*, if appropriate. For example, the object file we create in Example 11.16 by writing three *FlightRecord* objects is 241 bytes long. If we had not declared the constant *DecimalFormat* object *MONEY* as *static* in the *FlightRecord2* class, the size of the object file would be 1,945 bytes!

Similarly, the *writeObject* method does not write to the file any instance variable that is declared to be *transient*. Thus, you can also save space in the file by declaring an instance variable as *transient*. An instance variable is a good candidate to be declared *transient* if you can easily reproduce its value, or if the variable has a value of 0 at the time the file is created. For example, suppose our *FlightRecord* had an additional instance variable named *totalRevenue*, which stored a value we calculated by multiplying *avgTicketPrice* by *numPassengers*. Because we can easily recalculate the value for *totalRevenue*, we can declare it as *transient*; then, that instance variable will not be written to the object file.

You declare an instance variable as *transient* by inserting the keyword *transient* between the access modifier and the data type of the instance variable, as in the following syntax:

```
accessModifier transient dataType instanceVariableName
```

SOFTWARE ENGINEERING TIP

To save disk space when writing to an object file, declare the class data as *static* or *transient* where appropriate.

Thus, the following declaration would declare the *totalRevenue* instance variable as *transient*:

```
private transient double totalRevenue;
```

11.7.2 Reading Objects from Files

Reading objects from a file somewhat parallels writing objects to a file.

The class *ObjectInputStream*, a subclass of *InputStream*, coupled with *FileInputStream*, provides the functionality we need. We are already familiar with the *FileInputStream* class.

ObjectInputStream is designed to read objects from a file. The *readObject* method, which does not take any arguments, reads the next object from the file and returns it. Because the *readObject* method returns a generic *Object*, we must type cast the returned object to the appropriate class. When the end of the file is reached, the *readObject* method throws an *EOFException*. This is in contrast to the *Scanner* class, which provides the *hasNext* method to test whether the end of the file has been reached.

The classes, constructors, and methods discussed previously are shown in Table 11.9.

TABLE 11.9 Useful Classes, Constructors, and Methods for Reading Objects from a File

Classes, Constructors, and Methods for Reading Objects from a File		
Class	**Constructors**	**Exceptions thrown**
`FileInputStream`	`FileInputStream( String filename )` constructs a *FileInputStream* object from a *String* representing the name of a file	`FileNotFoundException`
`ObjectInputStream`	`ObjectInputStream( InputStream in )` constructs an *ObjectInputStream* from the *InputStream in*	`IOException`
	Method API	
`ObjectInputStream`	`Object readObject( )` reads the next object and returns it. The object must be an instance of a class that implements the *Serializable* interface. When the end of the file is reached, an *EOFException* is thrown	`IOException`, `ClassNotFoundException`, `EOFException`

Example 11.17 shows how these *FileInputStream* and *ObjectInputStream* classes can be used in a Java program to read objects from a file. We assume that the file *objects* contains *FlightRecord2* objects, as written in the previous section.

```
/* Demonstrating how to read objects from a file
   Anderson, Franceschi
*/

import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.FileNotFoundException;
import java.io.EOFException;
import java.io.IOException;

public class ReadingObjects
{
  public static void main( String [ ] args )
  {
   try
   {
     FileInputStream fis = new FileInputStream( "objects " );
     ObjectInputStream ois = new ObjectInputStream( fis );

     try
     {
       while ( true )
       {
         // read object, type cast returned object to FlightRecord
         FlightRecord2 temp = ( FlightRecord2 ) ois.readObject( );

         // print the FlightRecord2 object read
         System.out.println( temp );
       }
     } // end inner try block

     catch ( EOFException eofe )
     {
      System.out.println( "End of the file reached" );
     }

     catch ( ClassNotFoundException cnfe )
     {
      System.out.println( cnfe.getMessage( ) );
```

```
40        }
41
42        finally
43        {
44         System.out.println( "Closing file" );
45         ois.close( );
46        }
47     } // end outer try block
48
49     catch ( FileNotFoundException fnfe )
50     {
51        System.out.println( "Unable to find objects" );
52     }
53
54     catch ( IOException ioe )
55     {
56         ioe.printStackTrace( );
57     }
58  }
59 }
```

EXAMPLE 11.17 Reading Objects from a File

Lines 5 through 9 import the needed classes from the *java.io* package. The *ClassNotFoundException* class is part of the *java.lang* package and does not need to be imported.

Line 17 associates a *FileInputStream* object with the *objects* file, and line 18 instantiates an *ObjectInputStream* object for reading the objects from the file.

The *while* loop, from lines 22 to 29, reads and prints each object in the file. We will continue reading until the *readObject* method throws an *EOFException*, which will transfer control to the *catch* block (lines 32–35). Thus, our condition for the *while* loop is

```
while ( true )
```

In that *catch* block, we print a message that the end of the file was detected. Given this *while* loop construction, we do not need a priming read. Inside the *while* loop, we read an object, then print it. When the end of the file is detected, the output statement (line 28) will not be executed.

On line 25, we read an object from the file and assign it to the *FlightRecord2* object reference *temp*. Because the *readObject* method returns an *Object*, we

need to type cast the return value to a *FlightRecord2* object. The *readObject* method can also throw a *ClassNotFoundException* or an *IOException*, which will be caught at lines 37 or 54, respectively.

Because an *EOFException* will occur when the end of the file is reached, the *EOFException catch* block will always execute in a normal program run. Thus, any code following the *while* loop in the *try* block will not execute. In order to be able to close the *objects* file, we need to use nested *try/catch* blocks. The inner *try* block (lines 20–30) encloses the *while* loop; its associated *catch* blocks handle the *EOFException* and *ClassNotFoundException*. The outer *try* block (lines 15–47) encloses the instantiations of the *FileInputStream* and *ObjectInputStream* objects, the inner *try* block, and the *finally* block where we close the file (line 45). We can close the file in the *finally* block because the *ois* object reference, declared in the outer *try* block, is visible (that is, in scope) inside the *finally* block.

The *catch* blocks following the outer *try* block handle any *FileNotFoundException* and any other *IOExceptions* that occur in the inner or outer *try* blocks.

It is important to place the *catch* clause with the *EOFException* ahead of the *catch* clause with the *IOException*; otherwise, the *EOFException catch* block will never be reached because *EOFException* is a subclass of *IOException*, and therefore will match an *IOException catch* block.

Figure 11.21 shows the console output when this program is executed. Note that after we read the last object in the file and we try to read another

```
Flight AA31: from BWI to SFO
    200 passengers; average ticket price: $235.90
Flight CO25: from LAX to JFK
    225 passengers; average ticket price: $419.90
Flight US57: from IAD to DEN
    175 passengers; average ticket price: $179.50
End of the file reached
Closing file
```

Figure 11.21
Output from Example 11.17

object, the code executes the *catch* block for the *EOFException*, then the *finally* block.

Skill Practice
with these end-of-chapter questions

11.10.1 Multiple Choice Exercises
Question 11

11.10.6 Write a Short Program
Question 50

11.10.8 Technical Writing
Question 63

11.8 Programming Activity 2: Reading Objects from a File

In this activity, you will read objects from a file and perform this activity:

> Read an object file containing bank account transaction objects. Loop through all the objects and calculate the new balance of the bank account. Assume that we do not know the number of transaction items, that is, objects, in the file.

Notice that this activity is identical to Programming Activity 1, except that the transactions you will read are stored in the file as objects.

The framework will display the current transaction and current balance so that you can check the correctness of your code as the program executes.

For example, Figure 11.22 demonstrates the animation: We are currently scanning a check transaction for the amount of $500.00. The original balance was $0.00 and the new balance is now –$500.00.

Task Instructions: Reading from the transactions.obj File

In the Chapter 11 directory on the CD-ROM accompanying this book, you will find a Programming Activity 2 directory. Copy the contents of the directory onto a directory on your disk.

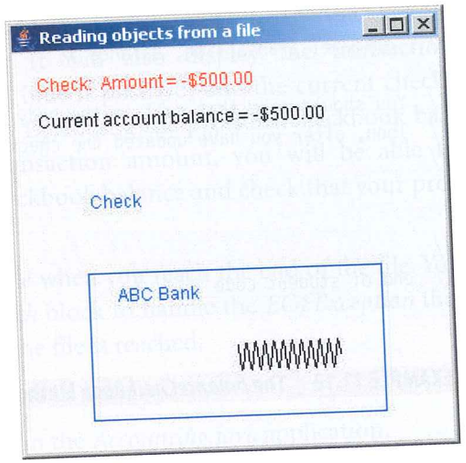


Figure 11.22
Animation of a $500 Check and the New Balance

- Open the *Accounting.java* file. Search for five asterisks (*****) to find the *balanceCheckBook* method where you will add your code. The method header has been coded for you. Write the code to read the transactions from the *transactions.obj* file, and calculate the balance after all the transactions in that file have been executed. This program first writes *Transaction* objects to the file *transactions.obj*; that code is provided. You need to code the body of the *balanceCheckBook* method in order to read that file. Example 11.18 shows the student code section of the *Accounting.java* file.

```
public void balanceCheckBook( )
{
//
// ***** Student writes the body of this method *****
//
// Using a while loop, read the file transactions.obj
// The file transactions.obj contains transaction objects
//
//  You will need to call the animate method inside
//  the body of the loop that reads the objects
//
// The animate method takes 2 arguments:
//     a Transaction object, representing the transaction
//     a double, representing the new checkbook balance
//  So if these two variables are transaction and balance,
//   then the call to animate will be:
```

```
//
//   animate( transaction, balance );
//
//  You should make that call in the body of your while
//  loop, after you have updated the checkbook balance
//


//
//  end of student code
//
}
```

EXAMPLE 11.18 The *balanceCheckBook* Method

- Begin with a checkbook balance of 0.00.
- To process the transactions, you will need to read one *Transaction* object at a time from the *transactions.obj* file; you will retrieve the transaction amount using the *getAmount* method of the *Transaction* class. The API for that method is:

```
public double getAmount( )
```

- Then process the transaction; you do not need to check the type of transaction. Just add the amount to the checkbook balance.
- After you have processed each transaction, call the *animate* method. This method belongs to the *Accounting* class, so you will call *animate* without using an object reference. The API of the *animate* method is the following:

```
public void animate( Transaction currentTransaction,
                     double currentBalance )
```

As you can see, the *animate* method takes two arguments:

- *currentTransaction* is the current *Transaction* object
- *currentBalance* is the current balance of the checkbook

Assuming that you have a *Transaction* object reference called *transactionObject* and a *double* called *balance*, a call to *animate* will look like the following:

```
animate( transactionObject, balance );
```


When you call *animate*, the window will display the current transaction graphically. It will also display the transaction amount (red if negative, blue if positive) and the current checkbook balance (in black). By adding the previous checkbook balance to the current transaction amount, you will be able to compute the current checkbook balance and check that your program is correct.

- Stop reading from the file when you reach the end of the file. You will need to set up a *catch* block to handle the *EOFException* that occurs when the end of the file is reached.
- Display the ending balance in a dialog box.

To test your code, compile and run the *Accounting.java* application.

If you have time . . .

- Modify the *main* method of the *Accounting* class, adding another transaction. Run the program again and verify that your code still yields the correct result. To add another transaction, you could, for instance, write this code:

```
Withdrawal w2 = new Withdrawal( -200.00 );
transactionList.add( w2 );
```

You can add a transaction of type *Check*, *Withdrawal*, *Deposit*, or *UnknownTransaction* (all of which are subclasses of the *abstract* class *Transaction*).

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you have coded the call to *animate* at the proper time.
- Verify that you have coded the condition for exiting the loop correctly.
- Verify that you have coded the body of the loop correctly.

? DISCUSSION QUESTIONS

1. Explain why we cannot simply read the *transactions.obj* file as a text file.
2. Explain why we need to type cast each object that we read from the file.

11.9 Chapter Summary

- Java provides exception classes so that unexpected, illegal operations at run time can be trapped and handled. This provides the programmer with a tool to keep the program running instead of terminating.
- When calling a constructor or method that *throws* a checked exception, you must use *try* and *catch* blocks; otherwise, the code will not compile.
- For calls to a constructor or method that *throws* an unchecked exception, *try* and *catch* blocks are optional. If *try* and *catch* blocks are not used, the exception will be caught at run time by the Java Virtual Machine.
- When a *try* block assigns a value to a variable and that variable is used after the *try/catch* block, the variable must be initialized before the *try* block is entered.
- A variable defined inside a *try* block is local to that block.
- To define your own exception, create a class that extends an existing exception class. This class will consist of a constructor that accepts a message and passes the message to the superclass constructor.
- The method that will generate the exception includes the *throws* clause in the method header. If the invalid condition is detected, the method *throws* a new object of the user-defined exception type, passing the appropriate message to the constructor.
- The *java.io* package contains classes for input and output operations.
- In order to read from a file, that file must exist; otherwise, a *FileNotFoundException* will be thrown.
- When we open a file for writing, the file is created if it does not exist. If the file already exists, the contents of the file are deleted.
- When we open a file for appending, the file is created if it does not exist. If the file already exists, we start writing at the end of the file.
- The *Scanner* class in the *java.util* package is helpful in parsing a *String* consisting of fields separated by one or more delimiters.

- The *FileOutputStream* and *PrintWriter* classes provide functionality to write primitive data types to a text file.
- Objects can be written to a file; they must be instantiated from a class that *implements* the *Serializable* interface.
- The *Serializable* interface has no methods; therefore, no additional methods need to be implemented in a class that *implements* the *Serializable* interface.
- The *FileOutputStream* and *ObjectOutputStream* classes provide functionality to write objects to a file.
- To avoid writing class data to a file of objects, declare the data as *static* or *transient*, where appropriate.
- The *FileInputStream* and *ObjectInputStream* classes provide the functionality to read objects from a file.
- The *readObject* method returns the object read as an *Object* class reference. That object reference must be type cast to the appropriate class.

11.10 Exercises, Problems, and Projects

11.10.1 Multiple Choice Exercises

1. Why are exceptions useful?
   - ❑ They can replace selection statements, thus saving CPU time.
   - ❑ Exceptions enable programmers to attempt to recover from illegal situations and continue running the program.
2. Some methods that *throw* an exception require *try* and *catch* blocks, while some do not.
   - ❑ true
   - ❑ false
3. What keyword is found in the header of a method that could detect an error and generate an appropriate exception?
   - ❑ *throw*
   - ❑ *throws*
   - ❑ *exception*
   - ❑ *exceptions*