Chapter 11 Input/Output and Exception Handling

CHAPTER GOALS

- To be able to read and write text files
- To learn how to throw exceptions
- To be able to design your own exception classes
- To understand the difference between checked and unchecked exceptions
- To learn how to catch exceptions
- To know when and where to catch an exception

This chapter starts with a discussion of file input and output. Whenever you read or write data, potential errors are to be expected. A file may have been corrupted or deleted, or it may be stored on another computer that was just disconnected from the network. In order to deal with these issues, you need to know about exception handling. The remainder of this chapter tells you how your programs can report exceptional conditions, and how they can recover when an exceptional condition has occurred.

11.1 Reading and Writing Text Files

We begin this chapter by discussing the common task of reading and writing files that contain text. Examples are files that are created with a simple text editor, such as Windows Notepad, as well as Java source code and HTML files.

The simplest mechanism for reading text is to use the `Scanner` class. You already know how to use a `Scanner` for reading console input. To read input from a disk file, first construct a `FileReader` object with the name of the input file, then use the `FileReader` to construct a `Scanner` object:

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```

This `Scanner` object reads text from the file `input.txt`. You can use the `Scanner` methods (such as `next`, `nextLine`, `nextInt`, and `nextDouble`) to read data from the input file.

When reading text files, use the `Scanner` class.

To write output to a file, you construct a `PrintWriter` object with the given file name, for example

```
PrintWriter out = new PrintWriter("output.txt");
```

If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.

When writing text files, use the `PrintWriter` class and the `print/println` methods.

Use the familiar `print` and `println` methods to send numbers, objects, and strings to a `PrintWriter`:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

The `print` and `println` methods convert numbers to their decimal string representations and use the `toString` method to convert objects to strings.

498

499

When you are done processing a file, be sure to *close* the `Scanner` or `PrintWriter`:

```
in.close();
out.close();
```

If your program exits without closing the `PrintWriter`, not all of the output may be written to the disk file.

You must close all files When you are done processing them.

The following program puts these concepts to work. It reads all lines of an input file and sends them to the output file, preceded by *line numbers*. If the input file is

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

then the program produces the output file

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

The line numbers are enclosed in /* */ delimiters so that the program can be used for numbering Java source files.

There one additional issue that we need to tackle. When the input or output file doesn't exist, a `FileNotFoundException` can occur. The compiler insists that we tell it what the program should do when that happens. (In this regard, the `FileNot-FoundException` is different from the exceptions that you have already encountered. We will discuss this difference in detail in Section 11.3.) In our sample program, we take the easy way out and acknowledge that the `main` method should simply be terminated if the exception occurs. We label the `main` method like this:

```
public static void main(String[] args) throws
FileNotFoundException
```

You will see in the following sections how to deal with exceptions in a more professional way.

ch11/fileio/LineNumberer.java

```
1  import java.io.FileReader;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  import java.util.Scanner;
5
6  public class LineNumberer
7  {
```

```
 8    public static void main(String[] args)
 9          throws FileNotFoundException
10   {
11       Scanner console = new Scanner(System.in);
12       System.out.print("Input file: ");
13       String inputFileName = console.next();
14       System.out.print("Output file:");
15       String outputFileName = console.next();
16                                                          499
17       FileReader reader = new                            500
FileReader(inputFileName);
18       Scanner in = new Scanner(reader);
19       PrintWriter out = new
PrintWriter(outputFileName);
20       int lineNumber = 1;
21
22       while (in.hasNextLine())
23       {
24          String line = in.nextLine();
25          out.println("/* " + lineNumber + " */
" + line);
26          lineNumber++;
27       }
28
29       out.close();
30    }
31 }
```

Self Check

1. What happens when you supply the same name for the input and output files to the `LineNumberer` program?
2. What happens when you supply the name of a nonexistent input file to the `LineNumberer` program?

Common Error 11.1: Backslashes in File Names

When you specify a file name as a constant string, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
in = new FileReader("c:\\homework\\input.dat");
```

Recall that a single backslash inside quoted strings is an *escape character* that is combined with another character to form a special meaning, such as `\n` for a newline character. The \\ combination denotes a single backslash.

When a user supplies a file name to a program, however, the user should not type the backslash twice.

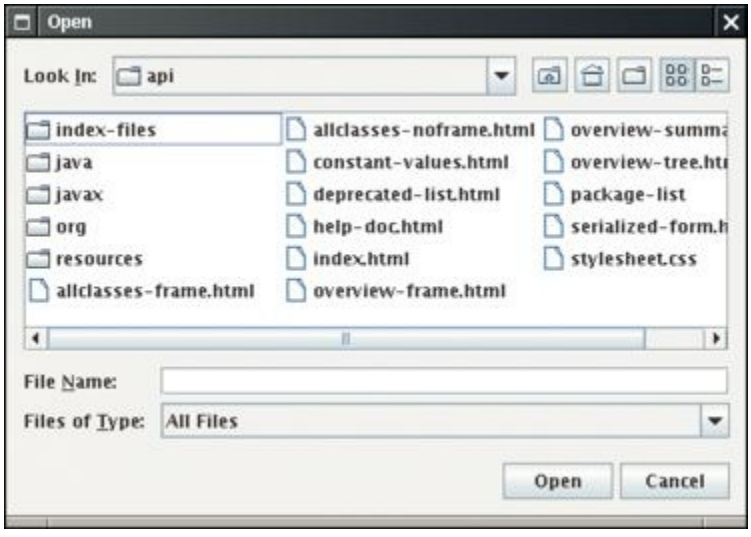
ADVANCED TOPIC 11.1: File Dialog Boxes

In a program with a graphical user interface, you will want to use a file dialog box (such as the one shown in the figure below) whenever the users of your program need to pick a file. The `JFileChooser` class implements a file dialog box for the Swing user interface toolkit.

The `JFileChooser` dialog box allows users to select a file by navigating through directories.

500

501



A `JFileChooser` Dialog Box

The `JFileChooser` class relies on another class, `File`, which describes disk files and directories. For example,

```
File inputFile = new File("input.txt");
```

describes the file `input.txt` in the current directory. The `File` class has methods to delete or rename the file. The file does not actually have to exist—you may want to pass the `File` object to an output stream or writer so that the file can be created. The `exists` method returns `true` if the file already exists.

> A `File` object describes a file or directory.

You cannot directly use a `File` object for reading or writing. You still need to construct a file reader or writer from the `File` object. Simply pass the `File` object in the constructor.

```
FileReader in = new FileReader(inputFile);
```

The `JFileChooser` class has many options to fine-tune the display of the dialog box, but in its most basic form it is quite simple: Construct a file chooser object; then call the `showOpenDialog` or `showSaveDialog` method. Both methods show the same dialog box, but the button for selecting a file is labeled “Open” or “Save”, depending on which method you call.

> You can pass a `File` object to the constructor of a file reader, writer, or stream.

For better placement of the dialog box on the screen, you can specify the user interface component over which to pop up the dialog box. If you don't care where the dialog box pops up, you can simply pass `null`. These methods return either `JFileChooser.APPROVE_OPTION`, if the user has chosen a file, or `JFileChooser.CANCEL_OPTION`, if the user canceled the selection. If a file was chosen, then you call the `getSelectedFile` method to obtain a `File` object that describes the file. Here is a complete example:

```
JFileChooser chooser = new JFileChooser();
FileReader in = null;
if (chooser.showOpenDialog(null) ==
JFileChooser.APPROVE_OPTION)
{
   File selectedFile = chooser.getSelectedFile();
```

```
      reader = new FileReader(selectedFile);
      . . .
   }
```

501

502

ADVANCED TOPIC 11.2: Command Line Arguments

Depending on the operating system and Java development system used, there are different methods of starting a program—for example, by selecting "Run" in the compilation environment, by clicking on an icon, or by typing the name of the program at a prompt in a terminal or shell window. The latter method is called "invoking the program from the command line". When you use this method, you must type the name of the program, but you can also type in additional information that the program can use. These additional strings are called *command line arguments*.

For example, it is convenient to specify the input and output file names for the `Line-Numberer` program on the command line:

```
java LineNumberer input.txt numbered.txt
```

The strings that are typed after the Java program name are placed into the `args` parameter of the `main` method. (Now you finally know the use of the `args` parameter that you have seen in so many programs!)

> When you launch a program from the command line, you can specify arguments after the program name. The program can access these strings by processing the `args` parameter of the `main` method.

For example, with the given program invocation, the `args` parameter of the `LineNumberer.main` method has the following contents:

- `args[0]` is "input.txt"
- `args[1]` is "output.txt"

The `main` method can then process these parameters, for example:

```
if (args.length >= 1)
   inputFileName = args[0];
```

It is entirely up to the program what to do with the command line argument strings. It is customary to interpret strings starting with a hyphen (-) as options and other strings as file names. For example, we may want to enhance the `LineNumberer` program so that a `-c` option places line numbers inside comment delimiters; for example

```
java LineNumberer -c HelloWorld.java HelloWorld.txt
```

If the `-c` option is missing, the delimiters should not be included. Here is how the `main` method can analyze the command line arguments:

```
for (String a : args)
{
   if (a.startsWith("-")) // It's an option
   {
       if (a.equals("-c")) useCommentDelimiters =
true;
   }
   else if (inputFileName == null) inputFileName =
a;
   else if (outputFileName == null) outputFileName
= a;
}
```

Should you support command line interfaces for your programs, or should you instead supply a graphical user interface with file chooser dialog boxes? For a casual and infrequent user, the graphical user interface is much better. The user interface guides the user along and makes it possible to navigate the application without much knowledge. But for a frequent user, graphical user interfaces have a major drawback—they are hard to automate. If you need to process hundreds of files every day, you could spend all your time typing file names into file chooser dialog boxes. But it is not difficult to call a program multiple times automatically with different command line arguments. Productivity Hint 7.1 discusses how to use shell scripts (also called batch files) for this purpose.

502
503

11.2 Throwing Exceptions

There are two main aspects to exception handling: *reporting* and *recovery*. A major challenge of error handling is that the point of reporting is usually far apart from the point of recovery. For example, the `get` method of the `ArrayList` class may detect

that a nonexistent element is being accessed, but it does not have enough information to decide what to do about this failure. Should the user be asked to try a different operation? Should the program be aborted after saving the user's work? The logic for these decisions is contained in a different part of the program code.

In Java, *exception handling* provides a flexible mechanism for passing control from the point of error reporting to a competent recovery handler. In the remainder of this chapter, we will look into the details of this mechanism.

When you detect an error condition, your job is really easy. You just `throw` an appropriate exception object, and you are done. For example, suppose someone tries to withdraw too much money from a bank account.

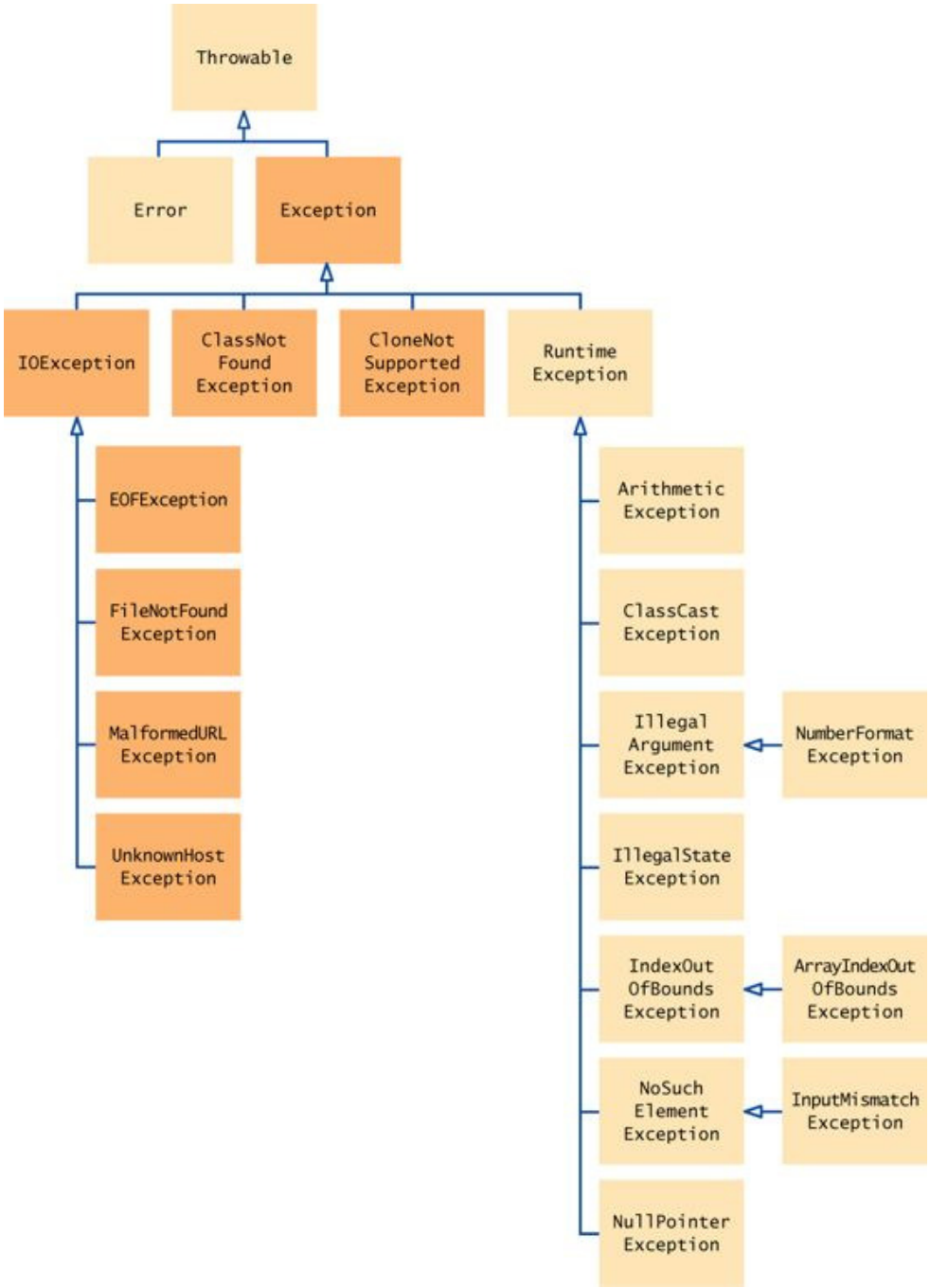
```
public class BankAccount
{
  public void withdraw(double amount)
  {
    if (amount > balance)
      // Now what?
    . . .
  }
  . . .
}
```

First look for an appropriate exception class. The Java library provides many classes to signal all sorts of exceptional conditions. Figure 1 shows the most useful ones.

To signal an exceptional condition, use the `throw` statement to throw an exception object.

Look around for an exception type that might describe your situation. How about the `IllegalStateException`? Is the bank account in an illegal state for the `withdraw` operation? Not really—some `withdraw` operations could succeed. Is the parameter value illegal? Indeed it is. It is just too large. Therefore, let's throw an `illegalArgumentException`. (The term *argument* is an alternative term for a parameter value.)

Figure 1



The Hierarchy of Exception Classes

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception
               = new
IllegalArgumentException("Amount exceeds balance");
         throw exception;
      }
     balance = balance - amount;
  }
  . . .
}
```

Actually, you don't have to store the exception object in a variable. You can just throw the object that the new operator returns:

```
throw new IllegalArgumentException("Amount exceeds
balance");
```

When you throw an exception, execution does not continue with the next statement but with an *exception handler*. For now, we won't worry about the handling of the exception. That is the topic of Section 11.4.

When you throw an exception, the current method terminates immediately.

SYNTAX 11.1 Throwing an Exception

`throw` *exceptionObject*;

Example:

```
throw new IllegalArgumentException();
```

Purpose

To throw an exception and transfer control to a handler for this exception type

SELF CHECK

3. How should you modify the `deposit` method to ensure that the balance is never negative?
4. Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`. What is the value of balance afterwards?

505

506

11.3 Checked and Unchecked Exceptions

Java exceptions fall into two categories, called *checked* and *unchecked* exceptions. When you call a method that throws a checked exception, the compiler checks that you don't ignore it. You must tell the compiler what you are going to do about the exception if it is ever thrown. For example, all subclasses of `IOException` are checked exceptions. On the other hand, the compiler does not require you to keep track of unchecked exceptions. Exceptions, such as `NumberFormatException`, `IllegalArgumentException`, and `NullPointerException`, are unchecked exceptions. More generally, all exceptions that belong to subclasses of `RuntimeException` are unchecked, and all other subclasses of the class `Exception` are checked. (In Figure 1, the checked exceptions are shaded in a darker color.) There is a second category of internal errors that are reported by throwing objects of type `Error`. One example is the `OutOfMemoryError`, which is thrown when all available memory has been used up. These are fatal errors that happen rarely and are beyond your control. They too are unchecked.

There are two kinds of exceptions: checked and unchecked. Unchecked exceptions extend the class `RuntimeException` or Error.

Why have two kinds of exceptions? A checked exception describes a problem that is likely to occur at times, no matter how careful you are. The unchecked exceptions, on the other hand, are your fault. For example, an unexpected end of file can be caused by forces beyond your control, such as a disk error or a broken network connection. But you are to blame for a `NullPointerException`, because your code was wrong when it tried to use a `null` reference.

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

The compiler doesn't check whether you handle a `NullPointer-Exception`, because you should test your references for `null` before using them rather than install a handler for that exception. The compiler does insist that your program be able to handle error conditions that you cannot prevent.

Actually, those categories aren't perfect. For example, the `Scanner.nextInt` method throws an unchecked `InputMismatchException` if a user enters an input that is not an integer. A checked exception would have been more appropriate because the programmer cannot prevent users from entering incorrect input. (The designers of the `Scanner` class made this choice to make the class easy to use for beginning programmers.)

As you can see from Figure 1, the majority of checked exceptions occur when you deal with input and output. That is a fertile ground for external failures beyond your control—a file might have been corrupted or removed, a network connection might be overloaded, a server might have crashed, and so on. Therefore, you will need to deal with checked exceptions principally when programming with files and streams.

You have seen how to use the `Scanner` class to read data from a file, by passing a `FileReader` object to the `Scanner` constructor:

```
String filename = . . .;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

506

507

However, the `FileReader` constructor can throw a `FileNotFoundException`. The `FileNotFoundException` is a checked exception, so you need to tell the compiler what you are going to do about it. You have two choices. You can handle the exception, using the techniques that you will see in Section 11.4. Or you can simply tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. The method that reads input rarely knows what to do about an unexpected error, so that is usually the better option.

To declare that a method should be terminated when a checked exception occurs within it, tag the method with a `throws` specifier.

```
public class DataSet
{
     public void read(String filename) throws
FileNotFoundException
     {
          FileReader reader = new
FileReader(filename);
          Scanner in = new Scanner(reader);
          . . .
     }
     . . .
}
```

The `throws` clause in turn signals the caller of your method that it may encounter a `FileNotFoundException`. Then the caller needs to make the same decision—handle the exception, or tell its caller that the exception may be thrown.

Add a `throws` specifier to a method that can throw a checked exception.

If your method can throw checked exceptions of different types, you separate the exception class names by commas:

```
public void read (String filename)
        throws IOException, ClassNotFoundException
```

Always keep in mind that exception classes form an inheritance hierarchy. For example, `FileNotFoundException` is a subclass of `IOException`. Thus, if a method can throw both an `IOException` and a `FileNotFoundException`, you only tag it as `throws IOException`.

It sounds somehow irresponsible not to handle an exception when you know that it happened. Actually, though, it is usually best not to catch an exception if you don't know how to remedy the situation. After all, what can you do in a low-level `read` method? Can you tell the user? How? By sending a message to `System.out`? You don't know whether this method is called in a graphical program or an embedded system (such as a vending machine), where the user may never see `System.out`. And even if your users can see your error message, how do you know that they can understand English? Your class may be used to build an application for users in another country. If you can't tell the user, can you patch up the data and keep going?

How? If you set a variable to zero, `null` or an empty string, that may just cause the program to break later, with much greater mystery.

Of course, some methods in the program know how to communicate with the user or take other remedial action. By allowing the exception to reach those methods, you make it possible for the exception to be processed by a competent handler.

507

508

SYNTAX 11.2 Exception Specification

```
accessSpecifier returnType
methodName(parameterType parameterName,. . .)
        throws ExceptionClass, ExceptionClass, . .
.
```

Example:

```
public void read(FileReader in)
      throws IOException
```

Purpose:

To indicate the checked exceptions that this method can throw

SELF CHECK

5. Suppose a method calls the `FileReader` constructor and the read method of the `FileReader` class, which can throw an `IOException`. Which `throws` specification should you use?

6. Why is a `NullPointerException` not a checked exception?

11.4 Catching Exceptions

Every exception should be handled somewhere in your program. If an exception has no handler, an error message is printed, and your program terminates. That may be fine for a student program. But you would not want a professionally written program to die just because some method detected an unexpected error. Therefore, you should install exception handlers for all exceptions that your program might throw.

In a method that is ready to handle a particular exception type, place the statements that can cause the exception inside a `try` block, and the handler inside a `catch` clause.

You install an exception handler with the `try/catch` statement. Each `try` block contains one or more statements that may cause an exception. `Each catch` clause contains the handler for an exception type. Here is an example:

```
try
{
   String filename = . . .;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
  catch (NumberFormatException exception)
  {
   System.out.println("Input was not a number");
  }
```

508

509

Three exceptions may be thrown in this `try` block: The `FileReader` constructor can throw a `FileNotFoundException`, `Scanner.next` can throw a `NoSuchElementException`, and `Integer.parseInt` can throw a `NumberFormatException`.

SYNTAX 11.3 General `try` Block

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
```

```
      statement
      statement
      . . .
   }
   catch (ExceptionClass exceptionObject)
   {
      statement
      statement
      . . .
   }
   . . .
```

Example:

```
   try
   {
      System.out.println("How old are you?");
      int age = in.nextInt();
      System.out.println("Next year, you'll be " +
   (age + 1));
   }
   catch (InputMismatchException exception)
   {
       exception.printStackTrace();
   }
```

Purpose:

To execute one or more statements that may generate exceptions. If an exception occurs and it matches one of the `catch` clauses, execute the first one that matches. If no exception occurs, or an exception is thrown that doesn't match any `catch` clause, then skip the `catch` clauses.

509

510

If any of these exceptions is actually thrown, then the rest of the instructions in the `try` block are skipped. Here is what happens for the various exception types:

- If a `FileNotFoundException` is thrown, then the `catch` clause for the `IOException` is executed. (Recall that `FileNotFoundException` is a subclass of `IOException`.)
- If a `NumberFormatException` occurs, then the second `catch` clause is executed.

- A `NoSuchElementException` is *not caught* by any of the `catch` clauses. The exception remains thrown until it is caught by another `try` block.

When the `catch (IOException exception)` block is executed, then some method in the `try` block has failed with an `IOException`. The variable `exception` contains a reference to the exception object that was thrown. The `catch` clause can analyze that object to find out more details about the failure. For example, you can get a printout of the chain of method calls that lead to the exception, by calling

```
exception.printStackTrace()
```

In these sample `catch` clauses, we merely inform the user of the source of the problem. A better way of dealing with the exception would be to give the user another chance to provide a correct input—see Section 11.7 for a solution.

It is important to remember that you should place `catch` clauses only in methods in which you can competently handle the particular exception type.

Self Check

7. Suppose the file with the given file name exists and has no contents. Trace the flow of execution in the `try` block in this section.
8. Is there a difference between catching checked and unchecked exceptions?

Quality Tip 11.1 Throw Early, Catch Late

When a method notices a problem that it cannot solve, it is generally better to throw an exception rather than try to come up with an imperfect fix (such as doing nothing or returning a default value).

> It is better to declare that a method throws a checked exception than to handle the exception poorly.

Conversely, a method should only catch an exception if it can really remedy the situation. Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

These principles can be summarized with the slogan “throw early, catch late”.

510

511

QUALITY TIP 11.2 Do Not Squelch Exceptions

When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains. In your eagerness to continue your work, it is an understandable impulse to shut the compiler up by squelching the exception:

```
try
{
   FileReader reader = new FileReader(filename);
   // Compiler complained about FileNotFoundException
   . . .
}
catch (Exception e) {} // So there!
```

The do-nothing exception handler fools the compiler into thinking that the exception has been handled. In the long run, this is clearly a bad idea. Exceptions were designed to transmit problem reports to a competent handler. Installing an incompetent handler simply hides an error condition that could be serious.

11.5 The Finally Clause

Occasionally, you need to take some action whether or not an exception is thrown. The `finally` construct is used to handle this situation. Here is a typical situation.

It is important to close a `PrintWriter` to ensure that all output is written to the file. In the following code segment, we open a writer, call one or more methods, and then close the writer:

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // May never get here
```

Now suppose that one of the methods before the last line throws an exception. Then the call to `close` is never executed! Solve this problem by placing the call to `close` inside a `finally` clause:

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

511

512

In a normal case, there will be no problem. When the `try` block is completed, the `finally` clause is executed, and the writer is closed. However, if an exception occurs, the `finally` clause is also executed before the exception is passed to its handler.

> Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown.

Use the `finally` clause whenever you need to do some clean up, such as closing a file, to ensure that the clean up happens no matter how the method exits.

It is also possible to have a `finally` clause following one or more `catch` clauses. Then the code in the `finally` clause is executed whenever the `try` block is exited in any of three ways:

1. After completing the last statement of the `try` block
2. After completing the last statement of a `catch` clause, if this `try` block caught an exception
3. When an exception was thrown in the `try` block and not caught

SYNTAX 11.4 `finally` Clause

```
try
{
```

```
      statement
      statement
      . . .
   }
   finally
   {
      statement
      statement
      . . .
   }
```

Examt:

```
   PrintWriter out = new PrintWriter(filename);
   try
   {
      writeData(out);
   }
   finally
   {
      out.close();
   }
```

Purpose:

To ensure that the statements in the `finally` clause are executed whether or not the statements in the `try` block throw an exception

512

513

However, we recommend that you don't mix `catch` and `finally` clauses in the same `try` block—see Quality Tip 11.3.

SELF CHECK

9. Why was the out variable declared outside the `try` block?

10. Suppose the file with the given name does not exist. Trace the flow of execution of the code segment in this section.

QUALITY TIP 11.3 Do Not Use `catch` and `finally` in the Same `try` Statement

It is tempting to combine `catch` and `finally` clauses, but the resulting code can be hard to understand. Instead, you should use a `try/finally` statement to close resources and a separate `try/catch` statement to handle errors. For example,

```
try
{
   PrintWriter out = new PrintWriter(filename);
   try
   {
   // Write output
   }
   finally
   {
      out.close();
   }
}
catch (IOException exception)
{
  // Handle exception
}
```

Note that the nested statements work correctly if the call `out.close()` throws an exception—see Exercise R11.18.

11.6 Designing your Own Exception Types

Sometimes none of the standard exception types describe your particular error condition well enough. In that case, you can design your own exception class. Consider a bank account. Let's report an `InsufficientFundsException` when an attempt is made to withdraw an amount from a bank account that exceeds the current balance.

```
if (amount > balance)
{                                                    513
throw new InsufficientFundsException(                514
```

```
                "withdrawal of " + amount + " exceeds
      balance of " + balance);
      }
```

Now you need to define the `InsufficientFundsException` class. Should it be a checked or an unchecked exception? Is it the fault of some external event, or is it the fault of the programmer? We take the position that the programmer could have prevented the exceptional condition—after all, it would have been an easy matter to check whether `amount <= account.getBalance()` before calling the `withdraw` method. Therefore, the exception should be an unchecked exception and extend the `RuntimeException` class or one of its subclasses.

> You can design your own exception types—subclasses of `Exception` or `RuntimeException`.

It is customary to provide two constructors for an exception class: a default constructor and a constructor that accepts a message string describing the reason for the exception. Here is the definition of the exception class.

```
public class InsufficientFundsException
         extends RuntimeException
{
    public InsufficientFundsException() {}
    public InsufficientFundsException(String message)
    {
       super(message);
     }
}
```

When the exception is caught, its message string can be retrieved using the `getMessage` method of the `RunTimeException` class.

Self Check

11. What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?

12. Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to

implement a `BadData-Exception`. Which exception class should you extend?

QUALITY TIP 11.4 Do Throw Specific Exceptions

When throwing an exception, you should choose an exception class that describes the situation as closely as possible. For example, it would be a bad idea to simply throw a `Runtime-Exception` object when a bank account has insufficient funds. This would make it far too difficult to catch the exception. After all, if you caught all exceptions of type `Runtime-Exception`, your `catch` clause would also be activated by exceptions of the type `NullPointer-Exception`, `ArrayIndexOutOfBoundsException`, and so on. You would then need to carefully examine the exception object and attempt to deduce whether the exception was caused by insufficient funds.

514

If the standard library does not have an exception class that describes your particular error situation, simply define a new exception class.

515

11.7 Case Study: A Complete Example

This section walks through a complete example of a program with exception handling. The program asks a user for the name of a file. The file is expected to contain data values. The first line of the file contains the total number of values, and the remaining lines contain the data. A typical input file looks like this:

```
3
1.45
-2.1
0.05
```

What can go wrong? There are two principal risks.

- The file might not exist.
- The file might have data in the wrong format.

Who can detect these faults? The `FileReader` constructor will throw an exception when the file does not exist. The methods that process the input values need to throw an exception when they find an error in the data format.

What exceptions can be thrown? The `FileReader` constructor throws a `FileNot-FoundException` when the file does not exist, which is very appropriate in our situation. The `close` method of the `FileReader` class can throw an `IOException`. Finally, when the file data is in the wrong format, we will throw a `BadDataException`, a custom checked exception class. We use a checked exception because corruption of a data file is beyond the control of the programmer.

Who can remedy the faults that the exceptions report? Only the `main` method of the `DataAnalyzer` program interacts with the user. It catches the exceptions, prints appropriate error messages, and gives the user another chance to enter a correct file.

ch11/data/DataAnalyzer.java

```
 1  import java.io.FileNotFoundException;
 2  import java.io.IOException;
 3  import java.util.Scanner;
 4
 5  /**
 6     This program reads a file containing numbers and analyzes its
   contents.
 7     If the file doesn't exist or contains strings that are not numbers, an
 8     error message is displayed.
 9  */
10 public class DataAnalyzer
11 {
12    public static void main(String[] args)
13    {                                                                    515
14    Scanner in = new Scanner(System.in);                                 516
15    DataSetReader reader = new DataSetReader();
16
17    boolean done = false;
18    while (!done)
19    {
20       try
21       {
22          System.out.println("Please enter the
   file name: ");
23          String filename = in.next();
24
```

```
25            double[] data = reader.
readFile(filename);
26            double sum = 0;
27            for (double d : data) sum = sum + d;
28            System.out.println("The sum is " +
sum);
29            done = true;
30         }
31         catch (FileNotFoundException exception)
32         {
33            System.out.println("File not found.");
34         }
35         catch (BadDataException exception)
36         {
37            System.out.println("Bad data: " +
exception.getMessage());
38         }
39         catch (IOException exception)
40         {
41            exception.printStackTrace();
42         }
43      }
44   }
45 }
```

The first two `catch` clauses in the `main` method give a human-readable error report if the file was not found or bad data was encountered. However, if another `IOException` occurs, then we print the stack trace so that a programmer can diagnose the problem.

The following `readFile` method of the `DataSetReader` class constructs the `Scanner` object and calls the `readData` method. It is completely unconcerned with any exceptions. If there is a problem with the input file, it simply passes the exception to its caller.

```
public double[] readFile(String filename)
      throws IOException, BadDataException
{
   FileReader reader = new FileReader(filename);
   try
   {
      Scanner in = new Scanner(reader);
```

```
            readData(in);
         }
         finally
         {                                                    516
            reader.close();                                   517
         }
         return data;
      }
```

Note how the `finally` clause ensures that the file is closed even when an exception occurs.

Also note that the `throws` specifier of the `readFile` method need not include the `FileNotFoundException` class because it is a subclass of `IOException`.

Next, here is the `readData` method of the `DataSetReader` class. It reads the number of values, constructs an array, and calls `readValue` for each data value.

```
private void readData(Scanner in) throws
BadDataException
{
   if (!in.hasNextInt())
      throw new BadDataException("Length expected");
   int numberOfValues = in.nextInt();
   data = new double [numberOfValues];
   for (int i = 0; i < numberOfValues; i + +)
         readValue(in, i);
   if (in.hasNext())
      throw new BadDataException("End of file
expected");
}
```

This method checks for two potential errors. The file might not start with an integer, or it might have additional data after reading all values.

However, this method makes no attempt to catch any exceptions. Plus, if the `readValue` method throws an exception—which it will if there aren't enough values in the file—the exception is simply passed on to the caller.

Here is the readVal ue method:

```
private void readValue(Scanner in, int i) throws
BadDataException
{
```

```
      if (!in.hasNextDouble())
         throw new BadDataException("Data value
   expected");
      data[i] = in. nextDouble();
   }
```

To see the exception handling at work, look at a specific error scenario.

1. `DataAnalyzer.main calls DataSetReader. read File.`
2. `readFile calls readData.`
3. `readData calls readValue.`
4. `readValue` doesn't find the expected value and throws a `BadDataException`.
5. `readValue` has no handler for the exception and terminates immediately.
6. `readData` has no handler for the exception and terminates immediately.
7. `readFile` has no handler for the exception and terminates immediately after executing the `finally` clause and closing the file.
8. `DataAnalyzer.main` has a handler for a `BadDataException`. That handler prints a message to the user. Afterwards, the user is given another chance to enter a file name. Note that the statements computing the sum of the values have been skipped.

517

518

This example shows the separation between error detection (in the `DataSetReader. readValue` method) and error handling (in the `DataAnalyzer.main` method). In between the two are the `readData` and `readFile` methods, which just pass exceptions along.

ch11/data/DataSetReader.java

```
1  import java.io. FileReader;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  /**
6     Reads a data set from a file. The file must have the format
```

```
7     numberOfValues
8     value1
9     value2
10     ...
11  */
12  public class DataSetReader
13  {
14     /**
15        Reads a data set.
16        @param filename the name of the file holding the data
17        @returnthe data in the file
18     */
19     public double[] readFile(String filename)
20           throws IOException, BadDataException
21     {
22        FileReader reader = new Fi leReader(fil
ename);
23        try
24        {
25           Scanner in = new Scanner(reader);
26           readData(in);
27        }
28        finally
29        {
30           reader.close();
31        }
32        return data;
33     }
34
35     /**
36        Reads all data.
37        @param in the scanner that scans the data
38     */
39     private void readData(Scanner in) throws
BadDataException
40     {
41        if (!in.hasNextInt())
42           throw new BadDataException("Length
expected");
43        int numberOfValues = in.nextInt();
44        data = new double[numberOfValues];
```

518

519

```
45
46          for (int i = 0; i < numberOfValues; i++)
47             readValue(in, i);
48
49          if (in.hasNext())
50             throw new BadDataException("End of
file expected");
51       }
52
53       /**
54          Reads one data value.
55          @param in the scanner that scans the data
56          @param i the position of the value to read
57       */
58       private void readValue(Scanner in, int i)
throws BadDataException
59       {
60          if (!in.hasNextDouble())
61             throw new BadDataException("Data value
expected");
62          data[i] = in.nextDouble();
63       }
64
65       private double[] data;
66    }
```

ch11/data/BadDataException.java

```
 1  /**
 2     This class reports bad input data.
 3  */
 4  public class BadDataException extends Exception
 5  {
 6     public BadDataException()
 7     public BadDataException(String message)
 8     {
 9        super(message);
10     }
11  }
```

Self Check

13. Why doesn't the `DataSetReader.read File` method catch any exceptions?

14. Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

519

520

Random Fact 11.1: The Ariane Rocket Incident

The European Space Agency (ESA), Europe's counterpart to NASA, had developed a rocket model called Ariane that it had successfully used several times to launch satellites and scientific experiments into space. However, when a new version, the Ariane 5, was launched on June 4, 1996, from ESA's launch site in Kourou, French Guiana, the rocket veered off course about 40 seconds after liftoff. Flying at an angle of more than 20 degrees, rather than straight up, exerted such an aerodynamic force that the boosters separated, which triggered the automatic self-destruction mechanism. The rocket blew itself up.

The ultimate cause of this accident was an unhandled exception! The rocket contained two identical devices (called inertial reference systems) that processed flight data from measuring devices and turned the data into information about the rocket position. The onboard computer used the position information for controlling the boosters. The same inertial reference systems and computer software had worked fine on the Ariane 4.

However, due to design changes to the rocket, one of the sensors measured a larger acceleration force than had been encountered in the Ariane 4. That value, expressed as a floating-point value, was stored in a 16-bit integer (like a `short` variable in Java). Unlike Java, the Ada language, used for the device software, generates an exception if a floating-point number is too large to be converted to an integer. Unfortunately, the programmers of the device had decided that this situation would never happen and didn't provide an exception handler.

When the overflow did happen, the exception was triggered and, because there was no handler, the device shut itself off. The onboard computer sensed the failure and switched over to the backup device. However, that device had shut itself off for

exactly the same reason, something that the designers of the rocket had not expected. They figured that the devices might fail for mechanical reasons, and the chances of two devices having the same mechanical failure was considered remote. At that point, the rocket was without reliable position information and went off course.

Perhaps it would have been better if the software hadn't been so thorough? If it had ignored the overflow, the device wouldn't have been shut off. It would have computed bad data. But then the device would have reported wrong position data, which could have been just as fatal. Instead, a correct implementation should have caught overflow exceptions and come up with some strategy to recompute the flight data. Clearly, giving up was not a reasonable option in this context.

The Explosion of the Ariane Rocket

520

521

The advantage of the exception-handling mechanism is that it makes these issues explicit to programmers—something to think about when you curse the Java compiler for complaining about uncaught exceptions.

CHAPTER SUMMARY

1. When reading text files, use the `Scanner` class.
2. When writing text files, use the `PrintWriter` class and the `print/println` methods.
3. You must close all files when you are done processing them.
4. The `JFileChooser` dialog box allows users to select a file by navigating through directories.

5. A `File` object describes a file or directory.
6. You can pass a `File` object to the constructor of a file reader, writer, or stream.
7. When you launch a program from the command line, you can specify arguments after the program name. The program can access these strings by processing the `args` parameter of the `main` method.
8. To signal an exceptional condition, use the `throw` statement to throw an exception object.
9. When you throw an exception, the current method terminates immediately.
10. There are two kinds of exceptions: checked and unchecked. Unchecked exceptions extend the class `RuntimeException` or `Error`.
11. Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.
12. Add a `throws` specifier to a method that can throw a checked exception.
13. In a method that is ready to handle a particular exception type, place the statements that can cause the exception inside a `try` block, and the handler inside a `catch` clause.
14. It is better to declare that a method throws a checked exception than to handle the exception poorly.
15. Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown.
16. You can design your own exception types—subclasses of `Exception` or `Runtime-Exception`.

521

522

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.io.EOFException
java.io.File
```

```
    exists
  java.io.FileNotFoundException
  java.io.FileReader
  java.io.IOException
  java.io.PrintWriter
     close
     print
     println
  java.lang.Error
  java.lang.IllegalArgumentException
  java.lang.IllegalStateException
  java.lang.NullPointerException
  java.lang.NumberFormatException
  java.lang.RuntimeException
  java.lang.Throwable
    getMessage
    printStackTrace
  java.util.InputMismatchException
  java.util.NoSuchElementException
  java.util.Scanner
    close
  javax.swing.JFileChooser
    getSelectedFile
    showOpenDialog
    showSaveDialog
```

REVIEW EXERCISES

★★ **Exercise R11.1.** What happens if you try to open a file for reading that doesn't exist?

What happens if you try to open a file for writing that doesn't exist?

★★★ **Exercise R11.2.** What happens if you try to open a file for writing, but the file or device is write-protected (sometimes called read-only)? Try it out with a short test program.

★ **Exercise R11.3.** How do you open a file whose name contains a backslash, like `c:\temp\output.dat`?

★★★ **Exercise R11.4.** What is a command line? How can a program read its command line arguments?

★★ **Exercise R11.5.** Give two examples of programs on your computer that read arguments from the command line.

★★ **Exercise R11.6.** If a program Woozle is started with the command

```
java Woozle-Dname=piglet -I\eeyore -v heff.txt
a.txt lump.txt
```

what are the values of `args[0]`, `args[1]`, and so on?

★★ **Exercise R11.7.** What is the difference between throwing an exception and catching an exception?

★★ **Exercise R11.8.** What is a checked exception? What is an unchecked exception? Is a `NullPointerException` checked or unchecked? Which exceptions do you need to declare with the `throws` keyword?

★ **Exercise R11.9.** Why don't you need to declare that your method might throw a `NullPointerException`?

522

523

★★ **Exercise R11.10.** When your program executes a `throw` statement, which statement is executed next?

★ **Exercise R11.11.** What happens if an exception does not have a matching `catch` clause?

★ **Exercise R11.12.** What can your program do with the exception object that a `catch` clause receives?

★ **Exercise R11.13.** Is the type of the exception object always the same as the type declared in the `catch` clause that catches it?

★ **Exercise R11.14.** What kind of values can you throw? Can you throw a string? An integer?

★★ **Exercise R11.15.** What is the purpose of the `finally` clause? Give an example of how it can be used.

★★★ **Exercise R11.16.** What happens when an exception is thrown, the code of a `finally` clause executes, and that code throws an exception of a

different kind than the original one? Which one is caught by a surrounding `catch` clause? Write a sample program to try it out.

★★ **Exercise R11.17.** Which exceptions can the `next` and `nextInt` methods of the `Scanner` class throw? Are they checked exceptions or unchecked exceptions?

★★★ **Exercise R11.18.** Suppose the `catch` clause in the example of Quality Tip 11.3 had been moved to the inner `try` block, eliminating the outer `try` block. Does the modified code work correctly if (a) the `FileReader` constructor throws an exception and (b) the `close` method throws an exception?

★★ **Exercise R11.19.** Suppose the program in Section 11.7 reads a file containing the following values:

```
0
1
2
3
```

What is the outcome? How could the program be improved to give a more accurate error report?

★★ **Exercise R11.20.** Can the `readFile` method in Section 11.7 throw a `NullPointer-Exception`? If so, how?

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★★ **Exercise P11.1.** Write a program that asks a user for a file name and prints the number of characters, words, and lines in that file.

523

524

★★ **Exercise P11.2.** Write a program that asks the user for a file name and counts the number of characters, words, and lines in that file. Then the program asks for the name of the next file. When the user enters a file that doesn't exist, the program prints the total counts of characters, words, and lines in all processed files and exits.

★★ **Exercise P11.3.** Write a program `CopyFile` that copies one file to another. The file names are specified on the command line. For example,

```
java CopyFile report.txt report.sav
```

★★ **Exercise P11.4.** Write a program that *concatenates* the contents of several files into one file. For example,

```
java CatFiles chapter1.txt chapter2.txt
chapter3.txt book.txt
```

makes a long file, `book.txt`, that contains the contents of the files `chapter1.txt`, `chapter2.txt`, and `chapter3.txt`. The output file is always the last file specified on the command line.

★★ **Exercise P11.5.** Write a program `Find` that searches all files specified on the command line and prints out all lines containing a keyword. For example, if you call

```
java Find ring report.txt address.txt
Homework.java
```

then the program might print

```
report.txt: has broken up an international ring
of DVD bootleggers that
address.txt: Kris Kringle, North Pole
address.txt: Homer Simpson, Springfield
Homework.java: String filename;
```

The keyword is always the first command line argument.

★★ **Exercise P11.6.** Write a program that checks the spelling of all words in a file. It should read each word of a file and check whether it is contained in a word list. A word list is available on most UNIX systems in the file `/usr/dict/words`. (If you don't have access to a UNIX system, your instructor should be able to get you a copy.) The program should print out all words that it cannot find in the word list.

★★ **Exercise P11.7.** Write a program that replaces each line of a file with its reverse. For example, if you run

```
java Reverse HelloPrinter. java
```

then the contents of `HelloPrinter.java` are changed to

```
retnirPolleH ssalc cilbup
{
)sgra ] [gnirtS(niam diov citats cilbup
{
wodniw elosnoc eht ni gniteerg a yalpsiD //
;)"!dlroW ,olleH"(nltnirp.tuo.metsyS
}
}
```

Of course, if you run `Reverse` twice on the same file, you get back the original file.

524

525

★★★ **Exercise P11.8.** Write a program that replaces all tab characters `'\t'` in a file with the *appropriate* number of spaces. By default, the distance between tab columns should be 3 (the value we use in this book for Java programs) but it can be changed by the user. Expand tabs to the number of spaces necessary to move to the next tab column. That may be *less* than three spaces. For example, consider the line containing `"\t|\t||\t|"`. The first tab is changed to three spaces, the second to two spaces, and the third to one space. Your program should be executed as

```
java TabExpander filename
```

or

```
java -t tabwidth filename
```

★ **Exercise P11.9.** Modify the `BankAccount` class to throw an `IllegalArgumentException` when the account is constructed with a negative balance, when a negative amount is deposited, or when an amount that is not between `0` and the current balance is withdrawn. Write a test program that causes all three exceptions to occur and that catches them all.

★★ **Exercise P11.10.** Repeat Exercise P11.9, but throw exceptions of three exception types that you define.

★★ **Exercise P11.11.** Write a program that asks the user to input a set of floating-point values. When the user enters a value that is not a number, give the user a second chance to enter the value. After two chances, quit reading input. Add all correctly specified values and print the sum when the user is done entering data. Use exception handling to detect improper inputs.

★★ **Exercise P11.12.** Repeat Exercise P11.11, but give the user as many chances as necessary to enter a correct value. Quit the program only when the user enters a blank input.

★ **Exercise P11.13.** Modify the `DataSetReader` class so that you do not call `hasNextInt` or `hasNextDouble`. Simply have `nextInt` and `nextDouble` throw an `InputMismatchException` or `NoSuchElementException` and catch it in the `main` method.

★★ **Exercise P11.14.** Write a program that reads in a set of coin descriptions from a file. The input file has the format

```
coinName1 coinValue1
coinName2 coinValue2
. . .
```

Add a method

```
void read(Scanner in) throws IOException
```

to the `Coin` class. Throw an exception if the current line is not properly formatted. Then implement a method

```
static ArrayList<Coin> readFile(String filename)
throws IOException
```

In the `main` method, call `readFile`. If an exception is thrown, give the user a chance to select another file. If you read all coins successfully, print the total value.

525

526

★★★ **Exercise P11.15.** Design a class Bank that contains a number of bank accounts. Each account has an account number and a current balance. Add an `accountNumber` field to the `BankAccount` class. Store the

bank accounts in an array list. Write a `readFile` method of the `Bank` class for reading a file with the format

```
accountNumber1 balance1
accountNumber2 balance2
. . .
```

Implement `read` methods for the `Bank` and `BankAccount` classes. Write a sample program to read in a file with bank accounts, then print the account with the highest balance. If the file is not properly formatted, give the user a chance to select another file.

- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 11.1.** You can read the contents of a web page with this sequence of commands.

```
String address =
"http://java.sun.com/index.html";
URL u = new URL(address);
URLConnection connection = u.openConnection();
InputStream stream = connection.getInputStream();
Scanner in = new Scanner(stream);
. . .
```

Some of these methods may throw exceptions—check out the API documentation. Design a class `LinkFinder` that finds all hyperlinks of the form

```
<a href="link">link text</a>
```

Throw an exception if you find a malformed hyperlink. Extra credit if your program can follow the links that it finds and find links in those web pages as well. (This is the method that search engines such as Google use to find web sites.)

ANSWERS TO SELF-CHECK QUESTIONS

1. When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the `while` loop exits immediately.
2. The program throws and catches a `FileNotFoundException`, prints an error message, and terminates.
3. Throw an exception if the amount being deposited is less than zero.
4. The balance is still zero because the last statement of the `withdraw` method was never executed.
5. The specification throws `IOException` is sufficient because `FileNotFound-Exception` is a subclass of `IOException`.
6. Because programmers should simply check for `null` pointers instead of trying to handle a `NullPointerException`.
7. The `FileReader` constructor succeeds, and in is constructed. Then the call `in.next()` throws a `NoSuchElementException`, and the `try` block is aborted. None of the catch clauses match, so none are executed. If none of the enclosing method calls catch the exception, the program terminates.
8. No—you catch both exception types in the same way, as you can see from the code example on page 508. Recall that `IOException` is a checked exception and `NumberFormatException` is an unchecked exception.
9. If it had been declared inside the `try` block, its scope would only have extended to the end of the `try` block, and the `finally` clause could not have closed it.
10. The `FileReader` constructor throws an exception. The `finally` clause is executed. Since `reader` is `null`, the call to `close` is not executed. Next, a `catch` clause that matches the `FileNotFoundException` is located. If none exists, the program terminates.

11. To pass the exception message string to the `RuntimeException` superclass.

12. `Exception` or `IOException` are both good choices. Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException`.

13. It would not be able to do much with them. The `DataSetReader` class is a reusable class that may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

14. `DataAnalyzer.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught.