7.4 Exception Handling

There are two aspects to dealing with program errors: *detection* and *handling*. For example, the `Scanner` constructor can detect an attempt to read from a non-existent file. However, it cannot handle that error. A satisfactory way of handling the error might be to terminate the program, or to ask the user for another file name. The `Scanner` class cannot choose between these alternatives. It needs to report the error to another part of the program.

In Java, *exception handling* provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error. In the following sections, we will look into the details of this mechanism.

7.4.1 Throwing Exceptions

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

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.

```
if (amount > balance)
{
   // Now what?
}
```

FULL CODE EXAMPLE
Go to `wiley.com/go/bjlo2code` to download a program that demonstrates throwing an exception.

First look for an appropriate exception class. The Java library provides many classes to signal all sorts of exceptional conditions. Figure 2 shows the most useful ones. (The classes are arranged as a tree-shaped hierarchy, with more specialized classes at the bottom of the tree. We will discuss such hierarchies in more detail in Chapter 9.)

Look around for an exception type that might describe your situation. How about the `ArithmeticException`? Is it an arithmetic error to have a negative balance? No—Java can deal with negative numbers. Is the amount to be withdrawn illegal? Indeed it is. It is just too large. Therefore, let's throw an `IllegalArgumentException`.

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
```

Syntax 7.1 Throwing an Exception

Syntax `throw` *exceptionObject*;

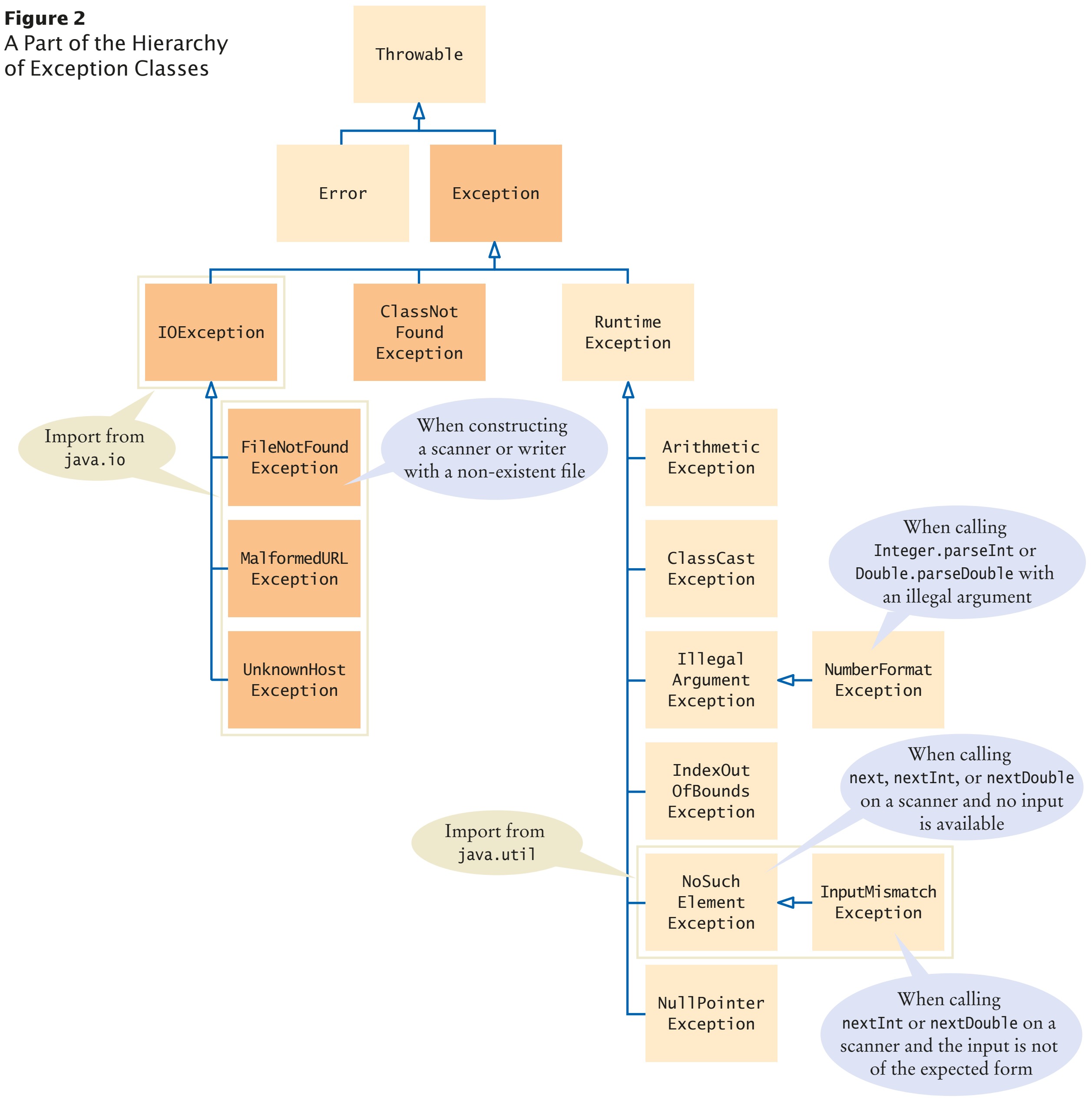
```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

A new exception object is constructed, then thrown.

This line is not executed when the exception is thrown.

Figure 2
A Part of the Hierarchy of Exception Classes


Throwable
Error
Exception
IOException
ClassNot Found Exception
Runtime Exception
Import from java.io
FileNotFound Exception
When constructing a scanner or writer with a non-existent file
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
When calling Integer.parseInt or Double.parseDouble with an illegal argument
Illegal Argument Exception
NumberFormat Exception
IndexOut OfBounds Exception
When calling next, nextInt, or nextDouble on a scanner and no input is available
Import from java.util
NoSuch Element Exception
InputMismatch Exception
NullPointer Exception
When calling nextInt or nextDouble on a scanner and the input is not of the expected form

When you throw an exception, processing continues in an exception handler.

When you throw an exception, execution does not continue with the next statement but with an **exception handler**. That is the topic of the next section.

© Lisa F. Young/iStockphoto.

When you throw an exception, the normal control flow is terminated. This is similar to a circuit breaker that cuts off the flow of electricity in a dangerous situation.

7.4.2 Catching Exceptions

Place the statements that can cause an exception inside a try block, and the handler inside a catch clause.

Every exception should be handled somewhere in your program. If an exception has no handler, an error message is printed, and your program terminates. Of course, such an unhandled exception is confusing to program users.

You handle exceptions with the `try/catch` statement. Place the statement into a location of your program that knows how to handle a particular exception. The `try` block contains one or more statements that may cause an exception of the kind that you are willing to handle. Each `catch` clause contains the handler for an exception type. Here is an example:

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println(exception.getMessage());
}
```

Syntax 7.2 Catching Exceptions

Syntax

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

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage());
}
```

This constructor can throw a `FileNotFoundException`.

This is the exception that was thrown.

When an `IOException` is thrown, execution resumes here.

A `FileNotFoundException` is a special case of an `IOException`.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

Three exceptions may be thrown in this try block:

- The Scanner constructor can throw a FileNotFound-Exception.
- Scanner.next can throw a NoSuchElementException.
- Integer.parseInt can throw a NumberFormatException.

© Andraz Cerar/iStockphoto.

You should only catch those exceptions that you can handle.

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. (If you look at Figure 2, you will note that FileNotFoundException is a descendant of IOException.) If you want to show the user a different message for a FileNotFoundException, you must place the catch clause *before* the clause for an 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 statement.

Each catch clause contains a handler. When the catch (IOException exception) block is executed, then some method in the try block has failed with an IOException (or one of its descendants).

In this handler, we produce a printout of the chain of method calls that led to the exception, by calling

```
exception.printStackTrace()
```

In the second exception handler, we call exception.getMessage() to retrieve the message associated with the exception. When the parseInt method throws a NumberFormat-Exception, the message contains the string that it was unable to format. When you throw an exception, you can provide your own message string. For example, when you call

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

the message of the exception is the string provided in the constructor.

In these sample catch clauses, we merely inform the user of the source of the problem. Often, it is better to give the user another chance to provide a correct input—see Section 7.5 for a solution.

FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download a program that demonstrates catching exceptions.

7.4.3 Checked Exceptions

In Java, the exceptions that you can throw and catch fall into three categories.

- Internal errors are reported by descendants of the type Error. One example is the OutOfMemoryError, which is thrown when all available computer memory has been used up. These are fatal errors that happen rarely, and we will not consider them in this book.
- Descendants of RuntimeException, such as as IndexOutOfBoundsException or Illegal-ArgumentException indicate errors in your code. They are called **unchecked exceptions**.

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

- All other exceptions are **checked exceptions**. These exceptions indicate that something has gone wrong for some external reason beyond your control. In Figure 2, the checked exceptions are shaded in a darker color.

Why have two kinds of exceptions? A checked exception describes a problem that can occur, no matter how careful you are. For example, an `IOException` can be caused by forces beyond your control, such as a disk error or a broken network connection. The compiler takes checked exceptions very seriously and ensures that they are handled. Your program will not compile if you don't indicate how to deal with a checked exception.

The unchecked exceptions, on the other hand, are your fault. The compiler does not check whether you handle an unchecked exception, such as an `IndexOutOfBoundsException`. After all, you should check your index values rather than install a handler for that exception.

FULL CODE EXAMPLE
Go to `wiley.com/go/bjlo2code` to download a program that demonstrates throwing and catching checked exceptions.

If you have a handler for a checked exception in the same method that may throw it, then the compiler is satisfied. For example,

```
try
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile); // Throws FileNotFoundException
   . . .
}
catch (FileNotFoundException exception) // Exception caught here
{
   . . .
}
```

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

However, it commonly happens that the current method *cannot handle* the exception. In that case, you need to tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. You supply a method with a `throws` clause.

```
public static String readData(String filename) throws FileNotFoundException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   . . .
}
```

The `throws` clause 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 declare that the exception may be thrown.

It sounds somehow irresponsible not to handle an exception when you know that it happened. Actually, the opposite is true. Java provides an exception handling facility so that an exception can be sent to the *appropriate* handler. Some methods detect errors, some methods handle them, and some methods just pass them along. The `throws` clause simply ensures that no exceptions get lost along the way.

© tillsonburg/iStockphoto.

Just as trucks with large or hazardous loads carry warning signs, the `throws` *clause warns the caller that an exception may occur.*

Syntax 7.3 The throws Clause

Syntax *modifiers returnType methodName*(*parameterType parameterName*, . . .)
    throws *ExceptionClass*, *ExceptionClass*, . . .

```
public static String readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

7.4.4 Closing Resources

When you use a resource that must be closed, such as a `PrintWriter`, you need to be careful in the presence of exceptions. Consider this sequence of statements:

```
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! This is a problem—data that was written to the stream may never end up in the file.

The try-with-resources statement ensures that a resource is closed when the statement ends normally or due to an exception.

The remedy is to use the **try-with-resources statement**. Declare the `PrintWriter` variable in a `try` statement, like this:

```
try (PrintWriter out = new PrintWriter(filename))
{
   writeData(out);
} // out.close() is always called
```

When the `try` block is completed, the `close` method is called on the variable. If no exception has occurred, this happens when the `writeData` method returns. However, if an exception occurs, the `close` method is invoked before the exception is passed to its handler.

Syntax 7.4 The try-with-resources Statement

Syntax try ($Type_1$ $variable_1$ = $expression_1$; $Type_2$ $variable_2$ = $expression_2$; . . .)
{
   . . .
}

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

This code may throw exceptions.

Implements the `AutoCloseable` interface.

At this point, `out.close()` is called, even when an exception occurs.

You can declare multiple variables in a try-with-resources statement, like this:

```
try (Scanner in = new Scanner(inFile); PrintWriter out = new PrintWriter(outFile))
{
   while (in.hasNextLine())
   {
      String input = in.nextLine();
      String result = process(input);
      out.println(result);
   }
} // Both in.close() and out.close() are called here
```

Use the try-with-resources statement whenever you work with a Scanner or PrintWriter to make sure that these resources are closed properly.

More generally, you can declare variables of any class that implements the AutoCloseable interface in a try-with-resources statement. The classes in the Java library that you use for working with files, network connections, and database connections all implement the AutoCloseable interface.

FULL CODE EXAMPLE
Go to wiley.com/go/bjlo2code to download a program that demonstrates closing resources.

All visitors to a foreign country have to go through passport control, no matter what happened on their trip. Similarly, the try-with-resources statement ensures that a resource is closed, even when an exception has occurred.

© archives/iStockphoto.

SELF CHECK

16. Suppose balance is 100 and amount is 200. What is the value of balance after these statements?

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

17. When depositing an amount into a bank account, we don't have to worry about overdrafts—except when the amount is negative. Write a statement that throws an appropriate exception in that case.

18. Consider the method

```
public static void main(String[] args)
{
   try
   {
      Scanner in = new Scanner(new File("input.txt"));
      int value = in.nextInt();
      System.out.println(value);
   }
   catch (IOException exception)
   {
      System.out.println("Error opening file.");
   }
}
```

Suppose the file with the given file name exists and has no contents. Trace the flow of execution.

19. Why is an `ArrayIndexOutOfBoundsException` not a checked exception?

20. Is there a difference between catching checked and unchecked exceptions?

21. What is wrong with the following code, and how can you fix it?

```
public static void writeAll(String[] lines, String filename)
{
   PrintWriter out = new PrintWriter(filename);
   for (String line : lines)
   {
      out.println(line.toUpperCase());
   }
   out.close();
}
```

Practice It Now you can try these exercises at the end of the chapter: R7.8, R7.9, R7.10.

Programming Tip 7.1

Throw Early, Catch Late

When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix. For example, suppose a method expects to read a number from a file, and the file doesn't contain a number. Simply using a zero value would be a poor choice because it hides the actual problem and perhaps causes a different problem elsewhere.

Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.

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".

Programming Tip 7.2

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
{
   Scanner in = new Scanner(new File(filename));
   // Compiler complained about FileNotFoundException
   . . .
}
catch (FileNotFoundException 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.

Programming Tip 7.3

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 `RuntimeException` 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 `RuntimeException`, your `catch` clause would also be activated by exceptions of the type `NullPointerException`, `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.

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

Special Topic 7.6

Assertions

An **assertion** is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true. Here is a typical assertion check:

```
public double deposit(double amount)
{
    assert amount >= 0;
    balance = balance + amount;
}
```

In this method, the programmer expects that the quantity `amount` can never be negative. When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, and assertion checking is enabled, then the `assert` statement throws an exception of type `AssertionError`, causing the program to terminate.

However, if assertion checking is disabled, then the assertion is never checked, and the program runs at full speed. By default, assertion checking is disabled when you execute a program.

To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MainClass
```

You can also use the shortcut `-ea` instead of `-enableassertions`. You should turn assertion checking on during program development and testing.

Special Topic 7.7

The `try/finally` Statement

You saw in Section 7.4.4 how to ensure that a resource is closed when an exception occurs. The `try`-with-resources statement calls the `close` methods of variables declared within the statement header. You should always use the `try`-with-resources statement when closing resources.

It can happen that you need to do some cleanup other than calling a `close` method. In that case, use the `try/finally` statement:

```
public double deposit (double amount)
try
{
   . . .
}
finally
{
   Cleanup. // This code is executed whether or not an exception occurs
}
```

If the body of the `try` statement completes without an exception, the cleanup happens. If an exception is thrown, the cleanup happens and the exception is then propagated to its handler.

The `try/finally` statement is rarely required because most Java library classes that require cleanup implement the `AutoCloseable` interface.

Computing & Society 7.2 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 chance 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 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.

© AP/Wide World Photos.

The Explosion of the Ariane Rocket

7.5 Application: Handling Input Errors

This section walks through an example program that includes exception handling. The program, `DataAnalyzer.java`, asks the user for the name of a file. The file is expected to contain data values. The first line of the file should contain 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
```

When designing a program, ask yourself what kinds of exceptions can occur.

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 Scanner 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 Scanner constructor throws a FileNotFoundException when the file does not exist, which is appropriate in our situation. When there are fewer data items than expected, or when the file doesn't start with the count of values, the program will throw an NoSuchElementException. Finally, when there are more inputs than expected, an IOException should be thrown.

For each exception, you need to decide which part of your program can competently handle it.

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

```
// Keep trying until there are no more exceptions
boolean done = false;
while (!done)
{
   try
   {
      Prompt user for file name.

      double[] data = readFile(filename);

      Process data.

      done = true;
   }
   catch (FileNotFoundException exception)
   {
      System.out.println("File not found.");
   }
   catch (NoSuchElementException exception)
   {
      System.out.println("File contents invalid.");
   }
   catch (IOException exception)
   {
      exception.printStackTrace();
   }
}
```

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

The following readFile method constructs the Scanner object and calls the readData method. It does not handle any exceptions. If there is a problem with the input file, it simply passes the exception to its caller.

```
public static double[] readFile(String filename) throws IOException
{
   File inFile = new File(filename);
   try (Scanner in = new Scanner(inFile))
   {
      return readData(in);
      }
   }
```

Note how the try-with-resources statement ensures that the file is closed even when an exception occurs.

Also note that the throws clause of the readFile method need not include the FileNotFoundException class because it is a special case of an IOException.

The readData method reads the number of values, constructs an array, and fills it with the data values.

```
   public static double[] readData(Scanner in) throws IOException
   {
      int numberOfValues = in.nextInt(); // May throw NoSuchElementException
      double[] data = new double[numberOfValues];

      for (int i = 0; i < numberOfValues; i++)
      {
         data[i] = in.nextDouble(); // May throw NoSuchElementException
      }

      if (in.hasNext())
      {
         throw new IOException("End of file expected");
      }
      return data;
   }
```

As discussed in Section 7.2.7, the calls to the nextInt and nextDouble methods can throw a NoSuchElementException when there is no input at all or an InputMismatchException if the input is not a number. As you can see from Figure 2 on page 340, an InputMismatchException is a special case of a NoSuchElementException.

You need not declare the NoSuchElementException in the throws clause because it is not a checked exception, but you can include it for greater clarity.

There are three potential errors:

- The file might not start with an integer.
- There might not be a sufficient number of data values.
- There might be additional input after reading all data values.

In the first two cases, the Scanner throws a NoSuchElementException. Note again that this is *not* a checked exception—we could have avoided it by calling hasNextInt/hasNextDouble first. However, this method does not know what to do in this case, so it allows the exception to be sent to a handler elsewhere.

When we find that there is additional unexpected input, we throw an IOException. To see the exception handling at work, look at a specific error scenario.

1. main calls readFile.
2. readFile calls readData.
3. readData calls Scanner.nextInt.
4. There is no integer in the input, and Scanner.nextInt throws a NoSuchElementException.
5. readData has no catch clause. It terminates immediately.
6. readFile has no catch clause. It terminates immediately when leaving the try-with-resources statement.

7. The first catch clause in `main` is for a `FileNotFoundException`. The exception that is currently being thrown is a `NoSuchElementException`, and this handler doesn't apply.
8. The next `catch` clause is for a `NoSuchElementException`, and execution resumes here. That handler prints a message to the user. Afterward, the user is given another chance to enter a file name. Note that the statements for processing the data have been skipped.

This example shows the separation between error detection (in the `readData` method) and error handling (in the `main` method). In between the two is the `readFile` method, which simply passes the exceptions along.

sec05/DataAnalyzer.java

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;
import java.util.NoSuchElementException;

/**
   This program processes a file containing a count followed by data values.
   If the file doesn't exist or the format is incorrect, you can specify another file.
*/
public class DataAnalyzer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      // Keep trying until there are no more exceptions

      boolean done = false;
      while (!done)
      {
         try
         {
            System.out.print("Please enter the file name: ");
            String filename = in.next();

            double[] data = readFile(filename);

            // As an example for processing the data, we compute the sum

            double sum = 0;
            for (double d : data) { sum = sum + d; }
            System.out.println("The sum is " + sum);

            done = true;
         }
         catch (FileNotFoundException exception)
         {
            System.out.println("File not found.");
         }
         catch (NoSuchElementException exception)
         {
            System.out.println("File contents invalid.");
         }
         catch (IOException exception)
         {
            exception.printStackTrace();
         }
      }
   }

   /**
      Opens a file and reads a data set.
      @param filename the name of the file holding the data
      @return the data in the file
   */
   public static double[] readFile(String filename) throws IOException
   {
      File inFile = new File(filename);
      try (Scanner in = new Scanner(inFile))
      {
         return readData(in);
      }
   }

   /**
      Reads a data set.
      @param in the scanner that scans the data
      @return the data set
   */
   public static double[] readData(Scanner in) throws IOException
   {
      int numberOfValues = in.nextInt(); // May throw NoSuchElementException
      double[] data = new double[numberOfValues];

      for (int i = 0; i < numberOfValues; i++)
      {
         data[i] = in.nextDouble(); // May throw NoSuchElementException
      }

      if (in.hasNext())
      {
         throw new IOException("End of file expected");
      }
      return data;
   }
}
```

SELF CHECK

22. Why doesn't the `readFile` method catch any exceptions?

23. What happens to the `Scanner` object if the `readData` method throws an exception?

24. What happens to the `Scanner` object if the `readData` method doesn't throw an exception?

25. Suppose the user specifies a file that exists and is empty. Trace the flow of execution in the `DataAnalyzer` program.

26. Why didn't the `readData` method call `hasNextInt`/`hasNextDouble` to ensure that the `NoSuchElementException` is not thrown?

Practice It Now you can try these exercises at the end of the chapter: R7.16, R7.17, E7.12.