

CHAPTER 12

EXCEPTION HANDLING AND TEXT I/O

Objectives

- To get an overview of exceptions and exception handling (§12.2).
- To explore the advantages of using exception handling (§12.2).
- To distinguish exception types: **Error** (fatal) vs. **Exception** (nonfatal) and checked vs. unchecked (§12.3).
- To declare exceptions in a method header (§12.4.1).
- To throw exceptions in a method (§12.4.2).
- To write a **try-catch** block to handle exceptions (§12.4.3).
- To explain how an exception is propagated (§12.4.3).
- To obtain information from an exception object (§12.4.4).
- To develop applications with exception handling (§12.4.5).
- To use the **finally** clause in a **try-catch** block (§12.5).
- To use exceptions only for unexpected errors (§12.6).
- To rethrow exceptions in a **catch** block (§12.7).
- To create chained exceptions (§12.8).
- To define custom exception classes (§12.9).
- To discover file/directory properties, to delete and rename files/directories, and to create directories using the **File** class (§12.10).
- To write data to a file using the **PrintWriter** class (§12.11.1).
- To use try-with-resources to ensure that the resources are closed automatically (§12.11.2).
- To read data from a file using the **Scanner** class (§12.11.3).
- To understand how data is read using a **Scanner** (§12.11.4).
- To develop a program that replaces text in a file (§12.11.5).
- To read data from the Web (§12.12).
- To develop a Web crawler (§12.13).



12.1 Introduction



Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution.

Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. For example, if you access an array using an index that is out of bounds, you will get a runtime error with an `ArrayIndexOutOfBoundsException`. If you enter a `double` value when your program expects an integer, you will get a runtime error with an `InputMismatchException`.

In Java, runtime errors are thrown as exceptions. An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. How can you handle the exception so the program can continue to run or else terminate gracefully? This chapter introduces this subject, and text input and output.

exception

12.2 Exception-Handling Overview



Exceptions are thrown from a method. The caller of the method can catch and handle the exception.

To demonstrate exception handling, including how an exception object is created and thrown, let's begin with the example in Listing 12.1, which reads in two integers and displays their quotient.

VideoNote

Exception-handling advantages

LISTING 12.1 Quotient.java

```
1 import java.util.Scanner;
2
3 public class Quotient {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         // Prompt the user to enter two integers
8         System.out.print("Enter two integers: ");
9         int number1 = input.nextInt();
10        int number2 = input.nextInt();
11
12        System.out.println(number1 + " / " + number2 + " is " +
13            (number1 / number2));
14    }
15 }
```

read two integers

integer division



Enter two integers: 5 2 ↵ Enter

5 / 2 is 2



Enter two integers: 3 0 ↵ Enter

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:13)

If you entered `0` for the second number, a runtime error would occur, because you cannot divide an integer by `0`. (Note a floating-point number divided by `0` does not raise an exception.) A simple way to fix this error is to add an `if` statement to test the second number, as shown in Listing 12.2.



Enter two integers: 5 3
5 / 3 is 1



Enter two integers: 5 0
Divisor cannot be zero

The method `quotient` (lines 4–11) returns the quotient of two integers. If `number2` is `0`, it cannot return a value, so the program is terminated in line 7. This is clearly a problem. You should not let the method terminate the program—the *caller* should decide whether to terminate the program.

How can a method notify its caller an exception has occurred? Java enables a method to throw an exception that can be caught and handled by the caller. Listing 12.3 can be rewritten, as shown in Listing 12.4.

LISTING 12.4 QuotientWithException.java

quotient method

throw exception

read two integers

try block
invoke method

catch block

```
1  import java.util.Scanner;
2
3  public class QuotientWithException {
4      public static int quotient(int number1, int number2) {
5          if (number2 == 0)
6              throw new ArithmeticException("Divisor cannot be zero");
7
8          return number1 / number2;
9      }
10
11     public static void main(String[] args) {
12         Scanner input = new Scanner(System.in);
13
14         // Prompt the user to enter two integers
15         System.out.print("Enter two integers: ");
16         int number1 = input.nextInt();
17         int number2 = input.nextInt();
18
19         try {
20             int result = quotient(number1, number2);
21             System.out.println(number1 + " / " + number2 + " is "
22                               + result);
23         }
24         catch (ArithmeticException ex) {
25             System.out.println("Exception: an integer " +
26                               "cannot be divided by zero ");
27         }
28
29         System.out.println("Execution continues ...");
30     }
31 }
```



Enter two integers: 5 3
5 / 3 is 1
Execution continues ...

```
Enter two integers: 5 0 
Exception: an integer cannot be divided by zero
Execution continues ...
```



If `number2` is 0, the method throws an exception (line 6) by executing

```
throw new ArithmeticException("Divisor cannot be zero");
```

throw statement

The value thrown, in this case `new ArithmeticException("Divisor cannot be zero")`, is called an *exception*. The execution of a `throw` statement is called *throwing an exception*. The exception is an object created from an exception class. In this case, the exception class is `java.lang.ArithmeticException`. The constructor `ArithmeticException(str)` is invoked to construct an exception object, where `str` is a message that describes the exception.

exception
throw exception

When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to “throw an exception” is to pass the exception from one place to another. The statement for invoking the method is contained in a `try` block. The `try` block (lines 19–23) contains the code that is executed in normal circumstances. The exception is caught by the `catch` block. The code in the `catch` block is executed to *handle the exception*. Afterward, the statement (line 29) after the `catch` block is executed.

handle exception

The `throw` statement is analogous to a method call, but instead of calling a method, it calls a `catch` block. In this sense, a `catch` block is like a method definition with a parameter that matches the type of the value being thrown. Unlike a method, however, after the `catch` block is executed, the program control does not return to the `throw` statement; instead, it executes the next statement after the `catch` block.

The identifier `ex` in the `catch`-block header

```
catch (ArithmeticException ex)
```

acts very much like a parameter in a method. Thus, this parameter is referred to as a `catch`-block parameter. The type (e.g., `ArithmeticException`) preceding `ex` specifies what kind of exception the `catch` block can catch. Once the exception is caught, you can access the thrown value from this parameter in the body of a `catch` block.

catch-block parameter

In summary, a template for a `try-throw-catch` block may look as follows:

```
try {
    Code to run;
    A statement or a method that may throw an exception;
    More code to run;
}
catch (type ex) {
    Code to process the exception;
}
```

An exception may be thrown directly by using a `throw` statement in a `try` block, or by invoking a method that may throw an exception.

The main method invokes `quotient` (line 20). If the `quotient` method executes normally, it returns a value to the caller. If the `quotient` method encounters an exception, it throws the exception back to its caller. The caller’s `catch` block handles the exception.

Now you can see the *advantage* of using exception handling: It enables a method to throw an exception to its caller, enabling the caller to handle the exception. Without this capability, the called method itself must handle the exception or terminate the program. Often the called method does not know what to do in case of error. This is typically the case for the library methods. The library method can detect the error, but only the caller knows what needs to be

advantage

done when an error occurs. The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).

Many library methods throw exceptions. Listing 12.5 gives an example that handles an `InputMismatchException` when reading an input.

LISTING 12.5 `InputMismatchExceptionDemo.java`

```

1  import java.util.*;
2
3  public class InputMismatchExceptionDemo {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6          boolean continueInput = true;
7
8          do {
9              try {
10                 System.out.print("Enter an integer: ");
11                 int number = input.nextInt();
12                 // Display the result
13                 System.out.println(
14                     "The number entered is " + number);
15
16                 continueInput = false;
17             }
18             catch (InputMismatchException ex) {
19                 System.out.println("Try again. (" +
20                     "Incorrect input: an integer is required)");
21                 input.nextLine(); // Discard input
22             }
23         } while (continueInput);
24     }
25 }
26

```

create a Scanner

try block

If an `InputMismatchException` occurs

catch block



```

Enter an integer: 3.5 Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 Enter
The number entered is 4

```

When executing `input.nextInt()` (line 11), an `InputMismatchException` occurs if the input entered is not an integer. Suppose `3.5` is entered. An `InputMismatchException` occurs and the control is transferred to the `catch` block. The statements in the `catch` block are now executed. The statement `input.nextLine()` in line 22 discards the current input line so the user can enter a new line of input. The variable `continueInput` controls the loop. Its initial value is `true` (line 6) and it is changed to `false` (line 17) when a valid input is received. Once a valid input is received, there is no need to continue the input.



12.2.1 What is the advantage of using exception handling?

12.2.2 Which of the following statements will throw an exception?

```

System.out.println(1 / 0);
System.out.println(1.0 / 0);

```

12.2.3 Point out the problem in the following code. Does the code throw any exceptions?

```
long value = Long.MAX_VALUE + 1;
System.out.println(value);
```

12.2.4 What does the JVM do when an exception occurs? How do you catch an exception?

12.2.5 What is the output of the following code?

```
public class Test {
    public static void main(String[] args) {
        try {
            int value = 30;
            if (value < 40)
                throw new Exception("value is too small");
        }
        catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
        System.out.println("Continue after the catch block");
    }
}
```

What would be the output if the line

```
int value = 30;
```

were changed to

```
int value = 50;
```

12.2.6 Show the output of the following code:

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
            System.out.print(i + " ");
            try {
                System.out.println(1 / 0);
            }
            catch (Exception ex) {
            }
        }
    }
}
```

(a)

```
public class Test {
    public static void main(String[] args) {
        try {
            for (int i = 0; i < 2; i++) {
                System.out.print(i + " ");
                System.out.println(1 / 0);
            }
        }
        catch (Exception ex) {
        }
    }
}
```

(b)

12.3 Exception Types

Exceptions are objects, and objects are defined using classes. The root class for exceptions is `java.lang.Throwable`.



The preceding section used the classes `ArithmeticException` and `InputMismatchException`. Are there any other types of exceptions you can use? Can you define your own exception classes? Yes. There are many predefined exception classes in the Java API. Figure 12.1 shows some of them, and in Section 12.9, you will learn how to define your own exception classes.

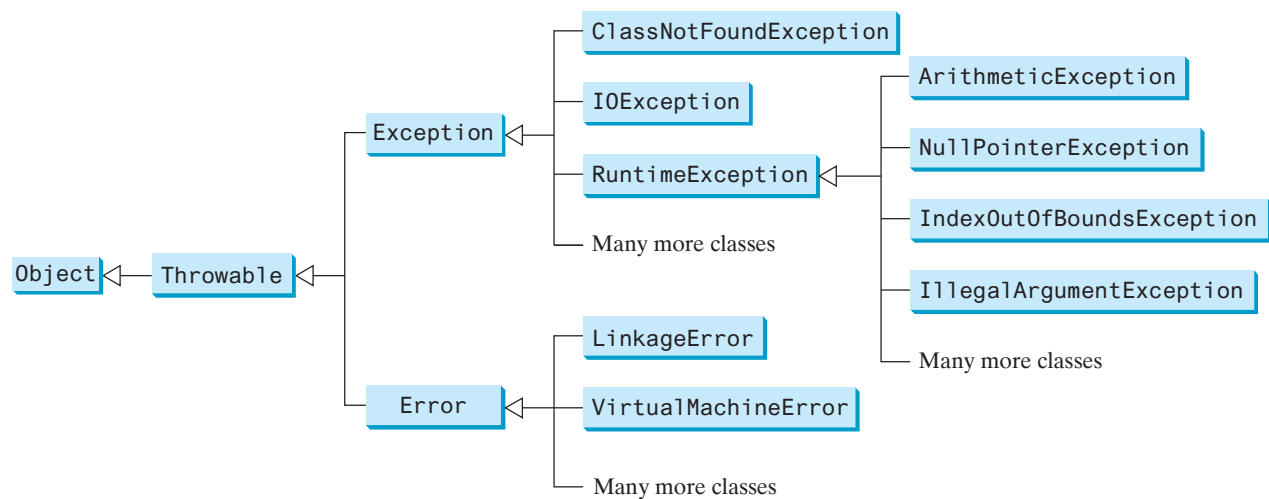


FIGURE 12.1 Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.



Note

The class names **Error**, **Exception**, and **RuntimeException** are somewhat confusing. All three of these classes are exceptions and all of the errors occur at runtime.

The **Throwable** class is the root of exception classes. All Java exception classes inherit directly or indirectly from **Throwable**. You can create your own exception classes by extending **Exception** or a subclass of **Exception**.

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

system error

- *System errors* are thrown by the JVM and are represented in the **Error** class. The **Error** class describes internal system errors, though such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully. Examples of subclasses of **Error** are listed in Table 12.1.

TABLE 12.1 Examples of Subclasses of **Error**

Class	Reasons for Exception
LinkageError	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
VirtualMachineError	The JVM is broken or has run out of the resources it needs in order to continue operating.

exception

- *Exceptions* are represented in the **Exception** class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program. Examples of subclasses of **Exception** are listed in Table 12.2.

TABLE 12.2 Examples of Subclasses of **Exception**

Class	Reasons for Exception
ClassNotFoundException	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the java command or if your program were composed of, say, three class files, only two of which could be found.
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of IOException are InterruptedException , EOFException (EOF is short for End of File), and FileNotFoundException .

- *Runtime exceptions* are represented in the `RuntimeException` class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions normally indicate programming errors. Examples of subclasses are listed in Table 12.3. runtime exception

TABLE 12.3 Examples of Subclasses of `RuntimeException`

Class	Reasons for Exception
<code>ArithmeticException</code>	Dividing an integer by zero. Note floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
<code>NullPointerException</code>	Attempt to access an object through a <code>null</code> reference variable.
<code>IndexOutOfBoundsException</code>	Index to an array is out of range.
<code>IllegalArgumentException</code>	A method is passed an argument that is illegal or inappropriate.

`RuntimeException`, `Error`, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning the compiler forces the programmer to check and deal with them in a `try-catch` block or declare it in the method header. Declaring an exception in the method header will be covered in Section 12.4. unchecked exception
checked exception

In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array. These are logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of `try-catch` blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.

12.3.1 Describe the Java `Throwable` class, its subclasses, and the types of exceptions.

12.3.2 What `RuntimeException` will the following programs throw, if any?



```
public class Test {
    public static void main(String[] args) {
        System.out.println(1 / 0);
    }
}
```

(a)

```
public class Test {
    public static void main(String[] args) {
        int[] list = new int[5];
        System.out.println(list[5]);
    }
}
```

(b)

```
public class Test {
    public static void main(String[] args) {
        String s = "abc";
        System.out.println(s.charAt(3));
    }
}
```

(c)

```
public class Test {
    public static void main(String[] args) {
        Object o = new Object();
        String d = (String)o;
    }
}
```

(d)

```
public class Test {
    public static void main(String[] args) {
        Object o = null;
        System.out.println(o.toString());
    }
}
```

(e)

```
public class Test {
    public static void main(String[] args) {
        System.out.println(1.0 / 0);
    }
}
```

(f)



12.4 More on Exception Handling

A handler for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method.

The preceding sections gave you an overview of exception handling and introduced several predefined exception types. This section provides an in-depth discussion of exception handling.

Java’s exception-handling model is based on three operations: *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 12.2.

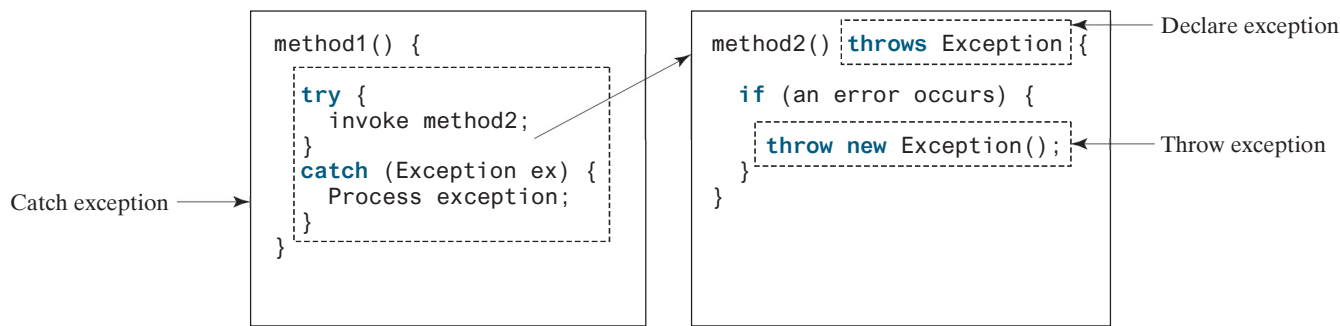


FIGURE 12.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

12.4.1 Declaring Exceptions

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the **main** method to start executing a program. Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*. Because system errors and runtime errors can happen to any code, Java does not require that you declare **Error** and **RuntimeException** (unchecked exceptions) explicitly in the method. However, all other exceptions thrown by the method must be explicitly declared in the method header so the caller of the method is informed of the exception.

To declare an exception in a method, use the **throws** keyword in the method header, as in this example:

```
public void myMethod() throws IOException
```

The **throws** keyword indicates **myMethod** might throw an **IOException**. If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after **throws**:

```
public void myMethod()  
    throws Exception1, Exception2, ..., ExceptionN
```



Note

If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

12.4.2 Throwing Exceptions

A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example: Suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument

must be nonnegative, but a negative argument is passed); the program can create an instance of `IllegalArgumentException` and throw it, as follows:

```
IllegalArgumentException ex =
    new IllegalArgumentException("Wrong Argument");
throw ex;
```

Or, if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```



Note

`IllegalArgumentException` is an exception class in the Java API. In general, each exception class in the Java API has at least two constructors: a no-arg constructor and a constructor with a `String` argument that describes the exception. This argument is called the *exception message*, which can be obtained by invoking `getMessage()` from an exception object.

exception message



Tip

The keyword to declare an exception is `throws`, and the keyword to throw an exception is `throw`.

throws vs. throw

12.4.3 Catching Exceptions

You now know how to declare an exception and how to throw an exception. When an exception is thrown, it can be caught and handled in a `try-catch` block, as follows:

catch exception

```
try {
    statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVarN) {
    handler for exceptionN;
}
```

If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped.

If one of the statements inside the `try` block throws an exception, Java skips the remaining statements in the `try` block and starts the process of finding the code to handle the exception. The code that handles the exception is called the *exception handler*; it is found by *propagating the exception* backward through a chain of method calls, starting from the current method. Each `catch` block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block. If so, the exception object is assigned to the variable declared and the code in the `catch` block is executed. If no handler is found, Java exits this method, passes the exception to the method's caller, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console. The process of finding a handler is called *catching an exception*.

exception handler
exception propagation

Suppose the `main` method invokes `method1`, `method1` invokes `method2`, `method2` invokes `method3`, and `method3` throws an exception, as shown in Figure 12.3. Consider the following scenario:

- If the exception type is `Exception3`, it is caught by the `catch` block for handling exception `ex3` in `method2`. `statement5` is skipped and `statement6` is executed.
- If the exception type is `Exception2`, `method2` is aborted, the control is returned to `method1`, and the exception is caught by the `catch` block for handling exception `ex2` in `method1`. `statement3` is skipped and `statement4` is executed.
- If the exception type is `Exception1`, `method1` is aborted, the control is returned to the `main` method, and the exception is caught by the `catch` block for handling exception `ex1` in the `main` method. `statement1` is skipped and `statement2` is executed.
- If the exception type is not caught in `method2`, `method1`, or `main`, the program terminates and `statement1` and `statement2` are not executed.

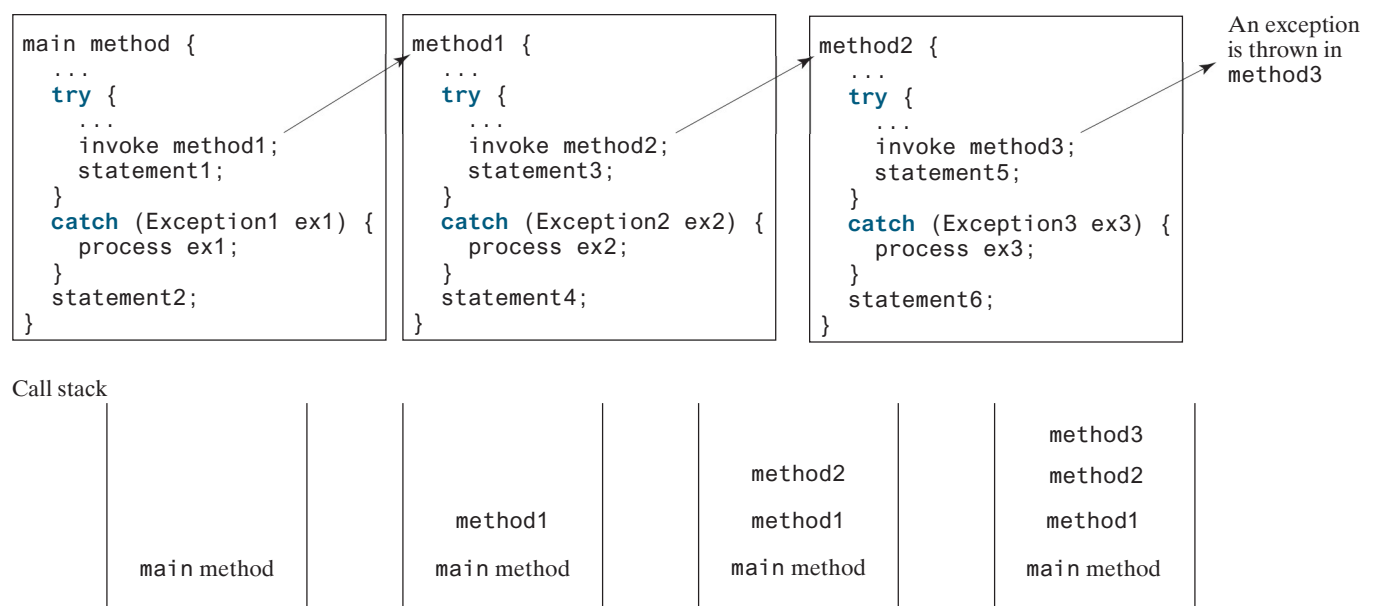




FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

catch block

 **Note** Various exception classes can be derived from a common superclass. If a `catch` block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

order of exception handlers

 **Note** The order in which exceptions are specified in `catch` blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type. For example, the ordering in (a) below is erroneous, because `RuntimeException` is a subclass of `Exception`. The correct ordering should be as shown in (b).

```
try {
    ...
}
catch (Exception ex) {
    ...
}
catch (RuntimeException ex) {
    ...
}
```

(a) Wrong order

```
try {
    ...
}
catch (RuntimeException ex) {
    ...
}
catch (Exception ex) {
    ...
}
```

(b) Correct order

**Note**

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than **Error** or **RuntimeException**), you must invoke it in a **try-catch** block or declare to throw the exception in the calling method. For example, suppose method **p1** invokes method **p2** and **p2** may throw a checked exception (e.g., **IOException**); you have to write the code as shown in (a) or (b) below.

catch or declare checked exceptions

```
void p1() {
    try {
        p2();
    }
    catch (IOException ex) {
        ...
    }
}
```

(a) Catch exception

```
void p1() throws IOException {
    p2();
}
```

(b) Throw exception

**Note**

You can use the new JDK 7 multcatch feature to simplify coding for the exceptions with the same handling code. The syntax is:

JDK 7 multcatch

```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
    // Same code for handling these exceptions
}
```

Each exception type is separated from the next with a vertical bar (**|**). If one of the exceptions is caught, the handling code is executed.

12.4.4 Getting Information from Exceptions

An exception object contains valuable information about the exception. You may use the following instance methods in the **java.lang.Throwable** class to get information regarding the exception, as shown in Figure 12.4. The **printStackTrace()** method prints stack trace

methods in Throwable

java.lang.Throwable	
<pre>+getMessage(): String +toString(): String +printStackTrace(): void +getStackTrace(): StackTraceElement[]</pre>	<p>Returns the message that describes this exception object.</p> <p>Returns the concatenation of three strings: (1) the full name of the exception class; (2) " : " (a colon and a space); and (3) the getMessage() method.</p> <p>Prints the Throwable object and its call stack trace information on the console.</p> <p>Returns an array of stack trace elements representing the stack trace pertaining to this exception object.</p>

FIGURE 12.4 **Throwable** is the root class for all exception objects.

information on the console. The stack trace lists all the methods in the call stack, which provides valuable information for debugging runtime errors. The `getStackTrace()` method provides programmatic access to the stack trace information printed by `printStackTrace()`.

Listing 12.6 gives an example that uses the methods in `Throwable` to display exception information. Line 4 invokes the `sum` method to return the sum of all the elements in the array. There is an error in line 23 that causes the `ArrayIndexOutOfBoundsException`, a subclass of `IndexOutOfBoundsException`. This exception is caught in the `try-catch` block. Lines 7, 8, and 9 display the stack trace, exception message, and exception object and message using the `printStackTrace()`, `getMessage()`, and `toString()` methods, as shown in Figure 12.5. Line 12 brings stack trace elements into an array. Each element represents a method call. You can obtain the method (line 14), class name (line 15), and exception line number (line 16) for each element.

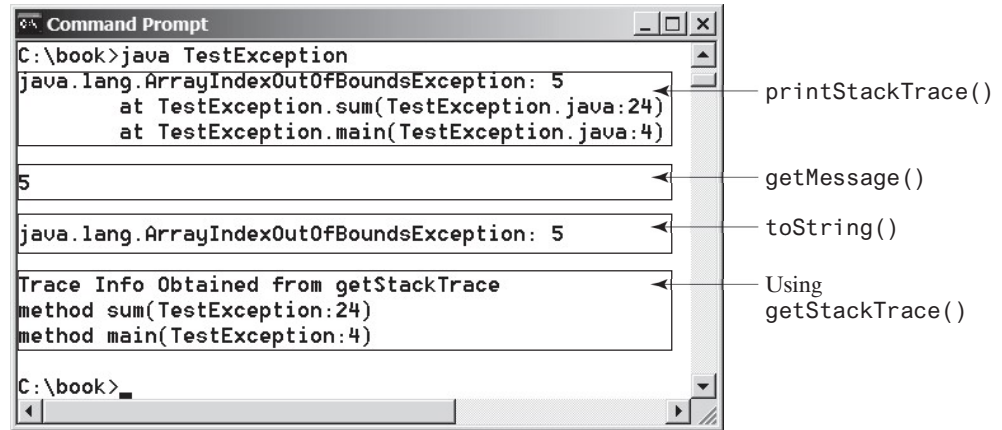


FIGURE 12.5 You can use the `printStackTrace()`, `getMessage()`, `toString()`, and `getStackTrace()` methods to obtain information from exception objects.

LISTING 12.6 TestException.java

```

1  public class TestException {
2      public static void main(String[] args) {
3          try {
4              System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
5          }
6          catch (Exception ex) {
7              ex.printStackTrace();
8              System.out.println("\n" + ex.getMessage());
9              System.out.println("\n" + ex.toString());
10
11              System.out.println("\nTrace Info Obtained from getStackTrace");
12              StackTraceElement[] traceElements = ex.getStackTrace();
13              for (int i = 0; i < traceElements.length; i++) {
14                  System.out.print("method " + traceElements[i].getMethodName());
15                  System.out.print("(" + traceElements[i].getClassName() + ":");
16                  System.out.println(traceElements[i].getLineNumber() + ")");
17              }
18          }
19      }
20
21      private static int sum(int[] list) {
22          int result = 0;
23          for (int i = 0; i <= list.length; i++)

```

invoke sum

printStackTrace()
getMessage()
toString()

getStackTrace()

cause an exception

```

24         result += list[i];
25     return result;
26 }
27 }

```

12.4.5 Example: Declaring, Throwing, and Catching Exceptions

This example demonstrates declaring, throwing, and catching exceptions by modifying the `setRadius` method in the `Circle` class in Listing 9.8, `Circle.java` (`CircleWithPrivateDataField`). The new `setRadius` method throws an exception if the radius is negative.

Listing 12.7 defines a new circle class named `CircleWithException`, which is the same as `Circle` in Listing 9.8 except that the `setRadius(double newRadius)` method throws an `IllegalArgumentException` if the argument `newRadius` is negative.

LISTING 12.7 `CircleWithException.java`

```

1  public class CircleWithException {
2      /** The radius of the circle */
3      private double radius;
4
5      /** The number of the objects created */
6      private static int numberOfObjects = 0;
7
8      /** Construct a circle with radius 1 */
9      public CircleWithException() {
10         this(1.0);
11     }
12
13     /** Construct a circle with a specified radius */
14     public CircleWithException(double newRadius) {
15         setRadius(newRadius);
16         numberOfObjects++;
17     }
18
19     /** Return radius */
20     public double getRadius() {
21         return radius;
22     }
23
24     /** Set a new radius */
25     public void setRadius(double newRadius)
26         throws IllegalArgumentException {                declare exception
27         if (newRadius >= 0)
28             radius = newRadius;
29         else
30             throw new IllegalArgumentException(            throw exception
31                 "Radius cannot be negative");
32     }
33
34     /** Return numberOfObjects */
35     public static int getNumberOfObjects() {
36         return numberOfObjects;
37     }
38
39     /** Return the area of this circle */
40     public double findArea() {
41         return radius * radius * 3.14159;
42     }
43 }

```


A test program that uses the new `Circle` class is given in Listing 12.8.

LISTING 12.8 TestCircleWithException.java

```

1  public class TestCircleWithException {
2      public static void main(String[] args) {
3          try {
4              CircleWithException c1 = new CircleWithException(5);
5              CircleWithException c2 = new CircleWithException(-5);
6              CircleWithException c3 = new CircleWithException(0);
7          }
8          catch (IllegalArgumentException ex) {
9              System.out.println(ex);
10         }
11
12         System.out.println("Number of objects created: " +
13             CircleWithException.getNumberOfObjects());
14     }
15 }

```



```

java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1

```

The original `Circle` class remains intact except that the class name is changed to `CircleWithException`, a new constructor `CircleWithException(newRadius)` is added, and the `setRadius` method now declares an exception and throws it if the radius is negative.

The `setRadius` method declares to throw `IllegalArgumentException` in the method header (lines 25–32 in Listing 12.7 `CircleWithException.java`). The `CircleWithException` class would still compile if the `throws IllegalArgumentException` clause (line 26) were removed from the method declaration, since it is a subclass of `RuntimeException` and every method can throw `RuntimeException` (an unchecked exception) regardless of whether it is declared in the method header.

The test program creates three `CircleWithException` objects—`c1`, `c2`, and `c3`—to test how to handle exceptions. Invoking `new CircleWithException(-5)` (line 5 in Listing 12.8) causes the `setRadius` method to be invoked, which throws an `IllegalArgumentException`, because the radius is negative. In the `catch` block, the type of the object `ex` is `IllegalArgumentException`, which matches the exception object thrown by the `setRadius` method, so this exception is caught by the `catch` block.

The exception handler prints a short message, `ex.toString()` (line 9 in Listing 12.8), about the exception, using `System.out.println(ex)`.

Note that the execution continues in the event of the exception. If the handlers had not caught the exception, the program would have abruptly terminated.

The test program would still compile if the `try` statement were not used, because the method throws an instance of `IllegalArgumentException`, a subclass of `RuntimeException` (an unchecked exception).



- 12.4.1** What is the purpose of declaring exceptions? How do you declare an exception and where? Can you declare multiple exceptions in a method header?
- 12.4.2** What is a checked exception and what is an unchecked exception?
- 12.4.3** How do you throw an exception? Can you throw multiple exceptions in one `throw` statement?
- 12.4.4** What is the keyword `throw` used for? What is the keyword `throws` used for?

12.4.5 Suppose `statement2` causes an exception in the following `try-catch` block:

```
try {
    statement1;
    statement2;
    statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}

statement4;
```

Answer the following questions:

- Will `statement3` be executed?
- If the exception is not caught, will `statement4` be executed?
- If the exception is caught in the `catch` block, will `statement4` be executed?

12.4.6 What is displayed when running the following program?

```
public class Test {
    public static void main(String[] args) {
        try {
            int[] list = new int[10];
            System.out.println("list[10] is " + list[10]);
        }
        catch (ArithmeticException ex) {
            System.out.println("ArithmeticException");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException");
        }
        catch (Exception ex) {
            System.out.println("Exception");
        }
    }
}
```

12.4.7 What is displayed when running the following program?

```
public class Test {
    public static void main(String[] args) {
        try {
            method();
            System.out.println("After the method call");
        }
        catch (ArithmeticException ex) {
            System.out.println("ArithmeticException");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException");
        }
        catch (Exception e) {
            System.out.println("Exception");
        }
    }

    static void method() throws Exception {
```

```
        System.out.println(1 / 0);
    }
}
```

12.4.8 What is displayed when running the following program?

```
public class Test {
    public static void main(String[] args) {
        try {
            method();
            System.out.println("After the method call");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException in main");
        }
        catch (Exception ex) {
            System.out.println("Exception in main");
        }
    }

    static void method() throws Exception {
        try {
            String s = "abc";
            System.out.println(s.charAt(3));
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException in method()");
        }
        catch (Exception ex) {
            System.out.println("Exception in method()");
        }
    }
}
```

12.4.9 What does the method `getMessage()` do?

12.4.10 What does the method `printStackTrace()` do?

12.4.11 Does the presence of a **try-catch** block impose overhead when no exception occurs?

12.4.12 Correct a compile error in the following code:

```
public void m(int value) {
    if (value < 40)
        throw new Exception("value is too small");
}
```

12.5 The **finally** Clause



*The **finally** clause is always executed regardless of whether an exception occurred or not.*

Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a **finally** clause that can be used to accomplish this objective. The syntax for the **finally** clause might look like this:

```
try {
    statements;
}
catch (TheException ex) {
    handling ex;
}
```

```
finally {
    finalStatements;
}
```

The code in the **finally** block is executed under all circumstances, regardless of whether an exception occurs in the **try** block or is caught. Consider three possible cases:

1. If no exception arises in the **try** block, **finalStatements** is executed and the next statement after the **try** statement is executed.
2. If a statement causes an exception in the **try** block that is caught in a **catch** block, the rest of the statements in the **try** block are skipped, the **catch** block is executed, and the **finally** clause is executed. The next statement after the **try** statement is executed.
3. If one of the statements causes an exception that is not caught in any **catch** block, the other statements in the **try** block are skipped, the **finally** clause is executed, and the exception is passed to the caller of this method.

The **finally** block executes even if there is a **return** statement prior to reaching the **finally** block.



Note

The **catch** block may be omitted when the **finally** clause is used.

omit catch block

12.5.1 Suppose **statement2** may cause an exception in the following statement:

```
try {
    statement1;
    statement2;
    statement3;
}
catch (Exception1 ex1) {
}
finally {
    statement4;
}
statement5;
```



Answer the following questions:

- a. If no exception occurs, will **statement4** or **statement5** be executed?
- b. If the exception is of type **Exception1**, will **statement4** or **statement5** be executed?
- c. If the exception is not of type **Exception1**, will **statement4** or **statement5** be executed?

12.6 When to Use Exceptions

A method should throw an exception if the error needs to be handled by its caller.

The **try** block contains the code that is executed in normal circumstances. The **catch** block contains the code that is executed in exceptional circumstances. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources, because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of method calls to search for the handler.



An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw or use exceptions.

In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled without throwing exceptions. This can be done by using **if** statements to check for errors.

When should you use a **try-catch** block in the code? Use it when you have to deal with unexpected error conditions. Do not use a **try-catch** block to deal with simple, expected situations. For example, the following code:

```
try {
    System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
    System.out.println("refVar is null");
}
```

is better replaced by

```
if (refVar != null)
    System.out.println(refVar.toString());
else
    System.out.println("refVar is null");
```

Which situations are exceptional and which are expected is sometimes difficult to decide. The point is not to abuse exception handling as a way to deal with a simple logic test.



12.6.1 The following method checks whether a string is a numeric string:

```
public static boolean isNumeric(String token) {
    try {
        Double.parseDouble(token);
        return true;
    }
    catch (java.lang.NumberFormatException ex) {
        return false;
    }
}
```

Is it correct? Rewrite it without using exceptions.

12.7 Rethrowing Exceptions



Java allows an exception handler to rethrow the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception.

The syntax for rethrowing an exception may look like this:

```
try {
    statements;
}
catch (TheException ex) {
    perform operations before exits;
    throw ex;
}
```

The statement **throw ex** rethrows the exception to the caller so other handlers in the caller get a chance to process the exception **ex**.

12.7.1 Suppose that `statement2` may cause an exception in the following code:



```
try {
    statement1;
    statement2;
    statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
    throw ex2;
}
finally {
    statement4;
}
statement5;
```

Answer the following questions:

- If no exception occurs, will `statement4` or `statement5` be executed?
- If the exception is of type `Exception1`, will `statement4` or `statement5` be executed?
- If the exception is of type `Exception2`, will `statement4` or `statement5` be executed?
- If the exception is not `Exception1` nor `Exception2`, will `statement4` or `statement5` be executed?

12.8 Chained Exceptions

Throwing an exception along with another exception forms a chained exception.



In the preceding section, the `catch` block rethrows the original exception. Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called *chained exceptions*. Listing 12.9 illustrates how to create and throw chained exceptions.

chained exception

LISTING 12.9 ChainedExceptionDemo.java

```
1 public class ChainedExceptionDemo {
2     public static void main(String[] args) {
3         try {
4             method1();
5         }
6         catch (Exception ex) {
7             ex.printStackTrace();
8         }
9     }
10
11     public static void method1() throws Exception {
12         try {
13             method2();
14         }
15         catch (Exception ex) {
16             throw new Exception("New info from method1", ex);
17         }
18     }
19 }
```

stack trace

chained exception

throw exception

```

19
20 public static void method2() throws Exception {
21     throw new Exception("New info from method2");
22 }
23 }

```



```

java.lang.Exception: New info from method1
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
    at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
    at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
    ... 1 more

```

The `main` method invokes `method1` (line 4), `method1` invokes `method2` (line 13), and `method2` throws an exception (line 21). This exception is caught in the `catch` block in `method1` and is wrapped in a new exception in line 16. The new exception is thrown and caught in the catch block in the `main` method in line 6. The sample output shows the output from the `printStackTrace()` method in line 7. The new exception thrown from `method1` is displayed first, followed by the original exception thrown from `method2`.



12.8.1 What would be the output if line 16 of Listing 12.9 is replaced by the following line?

```
throw new Exception("New info from method1");
```

12.9 Defining Custom Exception Classes

You can define a custom exception class by extending the `java.lang.Exception` class.



Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from `Exception` or from a subclass of `Exception`, such as `IOException`.

In Listing 12.7, `CircleWithException.java`, the `setRadius` method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler. In that case, you can define a custom exception class, as shown in Listing 12.10.

LISTING 12.10 `InvalidRadiusException.java`

extends Exception

```

1 public class InvalidRadiusException extends Exception {
2     private double radius;
3
4     /** Construct an exception */
5     public InvalidRadiusException(double radius) {
6         super("Invalid radius " + radius);
7         this.radius = radius;
8     }
9
10    /** Return the radius */
11    public double getRadius() {
12        return radius;
13    }
14 }

```

This custom exception class extends `java.lang.Exception` (line 1). The `Exception` class extends `java.lang.Throwable`. All the methods (e.g., `getMessage()`, `toString()`, and



VideoNote

Create custom exception classes

`printStackTrace()` in `Exception` are inherited from `Throwable`. The `Exception` class contains four constructors. Among them, the following constructors are often used:

<code>java.lang.Exception</code>	
<code>+Exception()</code>	Constructs an exception with no message.
<code>+Exception(message: String)</code>	Constructs an exception with the specified message.
<code>+Exception(message: String, cause: Exception)</code>	Constructs an exception with the specified message and a cause. This forms a chained exception.

Line 6 invokes the superclass's constructor with a message. This message will be set in the exception object and can be obtained by invoking `getMessage()` on the object.



Tip

Most exception classes in the Java API contain two constructors: a no-arg constructor and a constructor with a message parameter.

To create an `InvalidRadiusException`, you have to pass a radius. Therefore, the `setRadius` method in Listing 12.7 can be modified as shown in Listing 12.11.

LISTING 12.11 `TestCircleWithCustomException.java`

```

1  public class TestCircleWithCustomException {
2      public static void main(String[] args) {
3          try {
4              new CircleWithCustomException(5);
5              new CircleWithCustomException(-5);
6              new CircleWithCustomException(0);
7          }
8          catch (InvalidRadiusException ex) {
9              System.out.println(ex);
10         }
11
12         System.out.println("Number of objects created: " +
13             CircleWithCustomException.getNumberOfObjects());
14     }
15 }
16
17 class CircleWithCustomException {
18     /** The radius of the circle */
19     private double radius;
20
21     /** The number of objects created */
22     private static int numberOfObjects = 0;
23
24     /** Construct a circle with radius 1 */
25     public CircleWithCustomException() throws InvalidRadiusException {    declare exception
26         this(1.0);
27     }
28
29     /** Construct a circle with a specified radius */
30     public CircleWithCustomException(double newRadius)
31         throws InvalidRadiusException {
32         setRadius(newRadius);
33         numberOfObjects++;
34     }
35
36     /** Return radius */

```

throw exception

```

37     public double getRadius() {
38         return radius;
39     }
40
41     /** Set a new radius */
42     public void setRadius(double newRadius)
43         throws InvalidRadiusException {
44         if (newRadius >= 0)
45             radius = newRadius;
46         else
47             throw new InvalidRadiusException(newRadius);
48     }
49
50     /** Return numberOfObjects */
51     public static int getNumberOfObjects() {
52         return numberOfObjects;
53     }
54
55     /** Return the area of this circle */
56     public double findArea() {
57         return radius * radius * 3.14159;
58     }
59 }

```



```

InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1

```

The `setRadius` method in `CircleWithCustomException` throws an `InvalidRadiusException` when radius is negative (line 47). Since `InvalidRadiusException` is a checked exception, the `setRadius` method must declare it in the method header (line 43). Since the constructors for `CircleWithCustomException` invoke the `setRadius` method to set a new radius, and it may throw an `InvalidRadiusException`, the constructors are declared to throw `InvalidRadiusException` (lines 25 and 31).

Invoking `new CircleWithCustomException(-5)` (line 5) throws an `InvalidRadiusException`, which is caught by the handler. The handler displays the radius in the exception object `ex`.

checked custom exception

**Tip**

Can you define a custom exception class by extending `RuntimeException`? Yes, but it is not a good way to go because it makes your custom exception unchecked. It is better to make a custom exception checked, so the compiler can force these exceptions to be caught in your program.

**Check Point****12.9.1** How do you define a custom exception class?**12.9.2** Suppose that the `setRadius` method throws the `InvalidRadiusException` defined in Listing 12.10. What is displayed when running the following program?

```

public class Test {
    public static void main(String[] args) {
        try {
            method();
            System.out.println("After the method call");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException in main");
        }
    }
}

```



```

        catch (Exception ex) {
            System.out.println("Exception in main");
        }
    }

    static void method() throws Exception {
        try {
            Circle c1 = new Circle(1);
            c1.setRadius(-1);
            System.out.println(c1.getRadius());
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException in method()");
        }
        catch (Exception ex) {
            System.out.println("Exception in method()");
            throw ex;
        }
    }
}

```

12.10 The **File** Class

The **File** class contains the methods for obtaining the properties of a file/directory, and for renaming and deleting a file/directory.

Having learned exception handling, you are ready to step into file processing. Data stored in the program are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device. The file can then be transported and read later by other programs. Since data are stored in files, this section introduces how to use the **File** class to obtain file/directory properties, to delete and rename files/directories, and to create directories. The next section introduces how to read/write data from/to text files.

Every file is placed in a directory in the file system. An *absolute file name* (or *full name*) contains a file name with its complete path and drive letter. For example, **c:\book\Welcome.java** is the absolute file name for the file **Welcome.java** on the Windows operating system. Here, **c:\book** is referred to as the *directory path* for the file. Absolute file names are machine dependent. On the UNIX platform, the absolute file name may be **/home/liang/book/Welcome.java**, where **/home/liang/book** is the directory path for the file **Welcome.java**.

A *relative file name* is in relation to the current working directory. The complete directory path for a relative file name is omitted. For example, **Welcome.java** is a relative file name. If the current working directory is **c:\book**, the absolute file name would be **c:\book\Welcome.java**.

The **File** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The **File** class contains the methods for obtaining file and directory properties, and for renaming and deleting files and directories, as shown in Figure 12.6. However, the **File** class does not contain the methods for reading and writing file contents.

The file name is a string. The **File** class is a wrapper class for the file name and its directory path. For example, **new File("c:\\book")** creates a **File** object for the directory **c:\book** and **new File("c:\\book\\test.dat")** creates a **File** object for the file **c:\book\test.dat**, both on Windows. You can use the **File** class's **isDirectory()** method to check whether the object represents a directory, and the **isFile()** method to check whether the object represents a file.



why file?


absolute file name

directory path

relative file name


java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of <i>hidden</i> is system dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period (.) character.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\\book\\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\\book.
+lastModified(): long	Returns the time that the file was last modified.
+length(): long	Returns the size of the file, or 0 if it does not exist or if it is a directory.
+listFile(): File[]	Returns the files under the directory for a directory File object.
+delete(): boolean	Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames the file or directory represented by this File object to the specified name represented in dest. The method returns true if the operation succeeds.
+mkdir(): boolean	Creates a directory represented in this File object. Returns true if the the directory is created successfully.
+mkdirs(): boolean	Same as mkdir() except that it creates directory along with its parent directories if the parent directories do not exist.

FIGURE 12.6 The **File** class can be used to obtain file and directory properties, to delete and rename files and directories, and to create directories.

**Caution**

\ in file names

The directory separator for Windows is a backslash (\). The backslash is a special character in Java and should be written as \\ in a string literal (see Table 4.5).

**Note**

Constructing a **File** instance does not create a file on the machine. You can create a **File** instance for any file name regardless of whether it exists or not. You can invoke the **exists()** method on a **File** instance to check whether the file exists.

Do not use absolute file names in your program. If you use a file name such as `c:\\book\\Welcome.java`, it will work on Windows but not on other platforms. You should use a file name relative to the current directory. For example, you may create a **File** object using `new File("Welcome.java")` for the file **Welcome.java** in the current directory. You may create a **File** object using `new File("image/us.gif")` for the file **us.gif** under the **image** directory in the current directory. The forward slash (/) is the Java directory separator, which

relative file name

Java directory separator (/)

is the same as on UNIX. The statement `new File("image/us.gif")` works on Windows, UNIX, and any other platform.

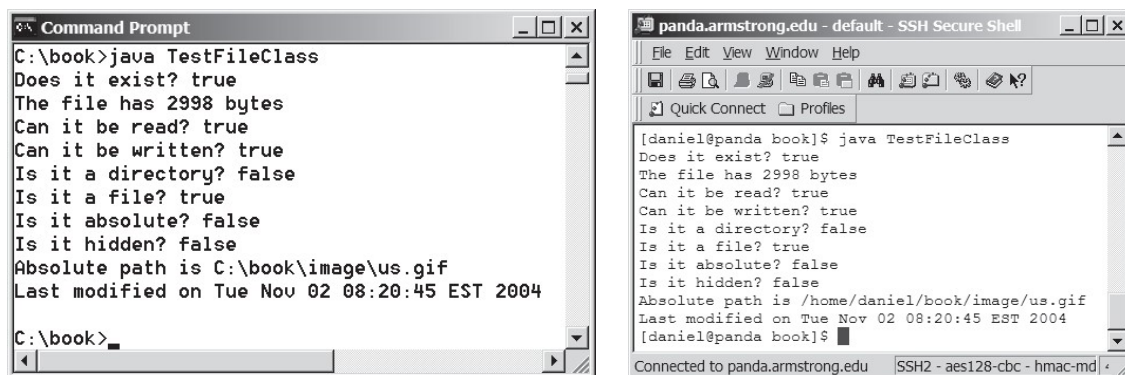
Listing 12.12 demonstrates how to create a `File` object and use the methods in the `File` class to obtain its properties. The program creates a `File` object for the file `us.gif`. This file is stored under the `image` directory in the current directory.

LISTING 12.12 TestFileClass.java

<pre> 1 public class TestFileClass { 2 public static void main(String[] args) { 3 java.io.File file = new java.io.File("image/us.gif"); 4 System.out.println("Does it exist? " + file.exists()); 5 System.out.println("The file has " + file.length() + " bytes"); 6 System.out.println("Can it be read? " + file.canRead()); 7 System.out.println("Can it be written? " + file.canWrite()); 8 System.out.println("Is it a directory? " + file.isDirectory()); 9 System.out.println("Is it a file? " + file.isFile()); 10 System.out.println("Is it absolute? " + file.isAbsolute()); 11 System.out.println("Is it hidden? " + file.isHidden()); 12 System.out.println("Absolute path is " + 13 file.getAbsolutePath()); 14 System.out.println("Last modified on " + 15 new java.util.Date(file.lastModified())); 16 } 17 }</pre>	<pre> create a File exists() length() canRead() canWrite() isDirectory() isFile() isAbsolute() isHidden() getAbsolutePath() lastModified()</pre>
--	--

The `lastModified()` method returns the date and time when the file was last modified, measured in milliseconds since the beginning of UNIX time (00:00:00 GMT, January 1, 1970). The `Date` class is used to display it in a readable format in lines 14 and 15.

Figure 12.7a shows a sample run of the program on Windows and Figure 12.7b, a sample run on UNIX. As shown in the figures, the path-naming conventions on Windows are different from those on UNIX.



(a) On Windows

(b) On UNIX

FIGURE 12.7 The program creates a `File` object and displays file properties.

12.10.1 What is wrong about creating a `File` object using the following statement? `new File("c:\book\test.dat");`

12.10.2 How do you check whether a file already exists? How do you delete a file? How do you rename a file? Can you find the file size (the number of bytes) using the `File` class? How do you create a directory?

12.10.3 Can you use the `File` class for I/O? Does creating a `File` object create a file on the disk?





VideoNote

Write and read data

Key
Point

12.1.1 File Input and Output

Use the **Scanner** class for reading text data from a file, and the **PrintWriter** class for writing text data to a file.

A **File** object encapsulates the properties of a file or a path, but it does not contain the methods for writing/reading data to/from a file (referred to as data *input* and *output*, or *I/O* for short). In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. There are two types of files: text and binary. Text files are essentially characters on disk. This section introduces how to read/write strings and numeric values from/to a text file using the **Scanner** and **PrintWriter** classes. Binary files will be introduced in Chapter 17.

12.1.1.1 Writing Data Using **PrintWriter**

The **java.io.PrintWriter** class can be used to create a file and write data to a text file. First, you have to create a **PrintWriter** object for a text file as follows:

```
PrintWriter output = new PrintWriter(filename);
```

Then, you can invoke the **print**, **println**, and **printf** methods on the **PrintWriter** object to write data to a file. Figure 12.8 summarizes frequently used methods in **PrintWriter**.

java.io.PrintWriter	
<pre>+PrintWriter(file: File) +PrintWriter(filename: String) +print(s: String): void +print(c: char): void +print(cArray: char[]): void +print(i: int): void +print(l: long): void +print(f: float): void +print(d: double): void +print(b: boolean): void</pre> <p>Also contains the overloaded println methods.</p> <p>Also contains the overloaded printf methods.</p>	<p>Creates a PrintWriter object for the specified file object.</p> <p>Creates a PrintWriter object for the specified file name string.</p> <p>Writes a string to the file.</p> <p>Writes a character to the file.</p> <p>Writes an array of characters to the file.</p> <p>Writes an int value to the file.</p> <p>Writes a long value to the file.</p> <p>Writes a float value to the file.</p> <p>Writes a double value to the file.</p> <p>Writes a boolean value to the file.</p> <p>A println method acts like a print method; additionally, it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.</p> <p>The printf method was introduced in §4.6, “Formatting Console Output.”</p>

FIGURE 12.8 The **PrintWriter** class contains the methods for writing data to a text file.

Listing 12.13 gives an example that creates an instance of **PrintWriter** and writes two lines to the file **scores.txt**. Each line consists of a first name (a string), a middle-name initial (a character), a last name (a string), and a score (an integer).

LISTING 12.13 WriteData.java

```
1 public class WriteData {
2     public static void main(String[] args) throws java.io.IOException {
3         java.io.File file = new java.io.File("scores.txt");
4         if (file.exists()) {
5             System.out.println("File already exists");
6             System.exit(1);
7         }
8     }
```

throws an exception
create File object
file exist?

```

9      // Create a file
10     java.io.PrintWriter output = new java.io.PrintWriter(file);           create PrintWriter
11
12     // Write formatted output to the file
13     output.print("John T Smith ");
14     output.println(90);
15     output.print("Eric K Jones ");
16     output.println(85);
17
18     // Close the file
19     output.close();                                                         close file
20 }
21 }

```

Lines 4–7 check whether the file **scores.txt** exists. If so, exit the program (line 6).

Invoking the constructor of **PrintWriter** will create a new file if the file does not exist. If the file already exists, the current content in the file will be discarded without verifying with the user. create a file

Invoking the constructor of **PrintWriter** may throw an I/O exception. Java forces you to write the code to deal with this type of exception. For simplicity, we declare **throws IOException** in the main method header (line 2). throws IOException

You have used the **System.out.print**, **System.out.println**, and **System.out.printf** methods to write text to the console output. **System.out** is a standard Java object for the console. You can create **PrintWriter** objects for writing text to any file using **print**, **println**, and **printf** (lines 13–16). print method

The **close()** method must be used to close the file (line 19). If this method is not invoked, the data may not be saved properly in the file. close file

12.11.2 Closing Resources Automatically Using try-with-resources

Programmers often forget to close the file. JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

```

try (declare and create resources) {
    Use the resource to process the file;
}

```

Using the try-with-resources syntax, we rewrite the code in Listing 12.13 as shown in Listing 12.14.

LISTING 12.14 WriteDataWithAutoClose.java

```

1  public class WriteDataWithAutoClose {
2      public static void main(String[] args) throws Exception {
3          java.io.File file = new java.io.File("scores.txt");
4          if (file.exists()) {
5              System.out.println("File already exists");
6              System.exit(0);
7          }
8
9          try (
10             // Create a file
11             java.io.PrintWriter output = new java.io.PrintWriter(file);           declare/create resource
12         ) {
13             // Write formatted output to the file
14             output.print("John T Smith ");
15             output.println(90);
16             output.print("Eric K Jones ");
17             output.println(85);
18         }
19     }
20 }

```

use the resource

A resource is declared and created followed by the keyword `try`. Note the resources are enclosed in the parentheses (lines 9–12). The resources must be a subtype of `AutoCloseable` such as a `PrinterWriter` that has the `close()` method. A resource must be declared and created in the same statement, and multiple resources can be declared and created inside the parentheses. The statements in the block (lines 12–18) immediately following the resource declaration use the resource. After the block is finished, the resource’s `close()` method is automatically invoked to close the resource. Using try-with-resources can not only avoid errors, but also make the code simpler. Note the catch clause may be omitted in a try-with-resources statement.

12.11.3 Reading Data Using Scanner

The `java.util.Scanner` class was used to read strings and primitive values from the console in Section 2.3, Reading Input from the Console. A `Scanner` breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a `Scanner` for `System.in`, as follows:

```
Scanner input = new Scanner(System.in);
```

To read from a file, create a `Scanner` for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

Figure 12.9 summarizes frequently used methods in `Scanner`.

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner’s delimiting pattern and returns this scanner.

FIGURE 12.9 The `Scanner` class contains the methods for scanning data.

Listing 12.15 gives an example that creates an instance of `Scanner` and reads data from the file `scores.txt`.

LISTING 12.15 ReadData.java

create a File

create a Scanner

```
1 import java.util.Scanner;
2
3 public class ReadData {
4     public static void main(String[] args) throws Exception {
5         // Create a File instance
6         java.io.File file = new java.io.File("scores.txt");
7
8         // Create a Scanner for the file
9         Scanner input = new Scanner(file);
```



```

10
11 // Read data from a file
12 while (input.hasNext()) {
13     String firstName = input.next();
14     String mi = input.next();
15     String lastName = input.next();
16     int score = input.nextInt();
17     System.out.println(
18         firstName + " " + mi + " " + lastName + " " + score);
19 }
20
21 // Close the file
22 input.close();
23 }
24 }

```

scores.txt

John	T	Smith	90
Eric	K	Jones	85

has next?
read items

close file

Note `new Scanner(String)` creates a `Scanner` for a given string. To create a `Scanner` to read data from a file, you have to use the `java.io.File` class to create an instance of the `File` using the constructor `new File(filename)` (line 6) and use `new Scanner(File)` to create a `Scanner` for the file (line 9).

Invoking the constructor `new Scanner(File)` may throw an I/O exception, so the `main` method declares `throws Exception` in line 4.

Each iteration in the `while` loop reads the first name, middle initial, last name, and score from the text file (lines 12–19). The file is closed in line 22.

It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file. You can rewrite this program using the try-with-resources syntax. See liveexample.pearsoncmg.com/html/ReadDataWithAutoClose.html.

12.11.4 How Does Scanner Work?

Section 4.5.5 introduced token-based and line-based input. The token-based input methods `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` read input separated by delimiters. By default, the delimiters are whitespace characters. You can use the `useDelimiter(String regex)` method to set a new pattern for delimiters.

How does an input method work? A token-based input first skips any delimiters (whitespace characters by default) then reads a token ending at a delimiter. The token is then automatically converted into a value of the `byte`, `short`, `int`, `long`, `float`, or `double` type for `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`, respectively. For the `next()` method, no conversion is performed. If the token does not match the expected type, a runtime exception `java.util.InputMismatchException` will be thrown.

Both methods `next()` and `nextLine()` read a string. The `next()` method reads a string separated by delimiters and `nextLine()` reads a line ending with a line separator.



Note

The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on UNIX. To get the line separator on a particular platform, use

```
String lineSeparator = System.getProperty("line.separator");
```

If you enter input from a keyboard, a line ends with the *Enter* key, which corresponds to the `\n` character.

The token-based input method does not read the delimiter after the token. If the `nextLine()` method is invoked after a token-based input method, this method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of the string returned by `nextLine()`.

File class

throws Exception

close file

change delimiter

InputMismatchException

next() vs. nextLine()

line separator

behavior of nextLine()

input from file

Suppose a text file named **test.txt** contains a line**34 567**

After the following code is executed,

```
Scanner input = new Scanner(new File("test.txt"));
int intValue = input.nextInt();
String line = input.nextLine();
```

intValue contains **34** and **line** contains the characters ' ', **5**, **6**, and **7**.

input from keyboard

What happens if the input is *entered from the keyboard*? Suppose you enter **34**, press the *Enter* key, then enter **567** and press the *Enter* key for the following code:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
String line = input.nextLine();
```

You will get **34** in **intValue** and an empty string in **line**. Why? Here is the reason. The token-based input method **nextInt()** reads in **34** and stops at the delimiter, which in this case is a line separator (the *Enter* key). The **nextLine()** method ends after reading the line separator and returns the string read before the line separator. Since there are no characters before the line separator, **line** is empty. For this reason, *you should not use a line-based input after a token-based input*.

scan a string

You can read data from a file or from the keyboard using the **Scanner** class. You can also scan data from a string using the **Scanner** class. For example, the following code:

```
Scanner input = new Scanner("13 14");
int sum = input.nextInt() + input.nextInt();
System.out.println("Sum is " + sum);
```

displays

Sum is 27

12.11.5 Case Study: Replacing Text

Suppose you are to write a program named **ReplaceText** that replaces all occurrences of a string in a text file with a new string. The file name and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of **StringBuilder** by **StringBuffer** in the file **FormatString.java** and saves the new file in **t.txt**.

Listing 12.16 gives the program. The program checks the number of arguments passed to the **main** method (lines 7–11), checks whether the source and target files exist (lines 14–25), creates a **Scanner** for the source file (line 29), creates a **PrintWriter** for the target file (line 30), and repeatedly reads a line from the source file (line 33), replaces the text (line 34), and writes a new line to the target file (line 35).

LISTING 12.16 ReplaceText.java

```
1 import java.io.*;
2 import java.util.*;
3
```



```

4 public class ReplaceText {
5     public static void main(String[] args) throws Exception {
6         // Check command line parameter usage
7         if (args.length != 4) {
8             System.out.println(
9                 "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10            System.exit(1);
11        }
12
13        // Check if source file exists
14        File sourceFile = new File(args[0]);
15        if (!sourceFile.exists()) {
16            System.out.println("Source file " + args[0] + " does not exist");
17            System.exit(2);
18        }
19
20        // Check if target file exists
21        File targetFile = new File(args[1]);
22        if (targetFile.exists()) {
23            System.out.println("Target file " + args[1] + " already exists");
24            System.exit(3);
25        }
26
27        try (
28            // Create input and output files
29            Scanner input = new Scanner(sourceFile);
30            PrintWriter output = new PrintWriter(targetFile);
31        ) {
32            while (input.hasNext()) {
33                String s1 = input.nextLine();
34                String s2 = s1.replaceAll(args[2], args[3]);
35                output.println(s2);
36            }
37        }
38    }
39 }

```

check command usage

source file exists?

target file exists?

try-with-resources

create a Scanner

create a PrintWriter

has next?

read a line

In a normal situation, the program is terminated after a file is copied. The program is terminated abnormally if the command-line arguments are not used properly (lines 7–11), if the source file does not exist (lines 14–18), or if the target file already exists (lines 22–25). The exit status codes 1, 2, and 3 are used to indicate these abnormal terminations (lines 10, 17, and 24).

12.11.1 How do you create a **PrintWriter** to write data to a file? What is the reason to declare **throws Exception** in the main method in Listing 12.13, **WriteData.java**? What would happen if the **close()** method were not invoked in Listing 12.13?



12.11.2 Show the contents of the file **temp.txt** after the following program is executed:

```

public class Test {
    public static void main(String[] args) throws Exception {
        java.io.PrintWriter output = new
            java.io.PrintWriter("temp.txt");
        output.printf("amount is %f %e\r\n", 32.32, 32.32);
        output.printf("amount is %5.4f %5.4e\r\n", 32.32, 32.32);
        output.printf("%6b\r\n", (1 > 2));
        output.printf("%6s\r\n", "Java");
        output.close();
    }
}

```

- 12.11.3** Rewrite the code in the preceding question using a try-with-resources syntax.
- 12.11.4** How do you create a **Scanner** to read data from a file? What is the reason to define **throws Exception** in the main method in Listing 12.15, ReadData.java? What would happen if the **close()** method were not invoked in Listing 12.15?
- 12.11.5** What will happen if you attempt to create a **Scanner** for a nonexistent file? What will happen if you attempt to create a **PrintWriter** for an existing file?
- 12.11.6** Is the line separator the same on all platforms? What is the line separator on Windows?
- 12.11.7** Suppose you enter **45 57.8 789**, then press the *Enter* key. Show the contents of the variables after the following code is executed:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

- 12.11.8** Suppose you enter **45**, press the *Enter* key, enter **57.8**, press the *Enter* key, and enter **789**, press the *Enter* key. Show the contents of the variables after the following code is executed:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

12.12 Reading Data from the Web



Just like you can read data from a file on your computer, you can read data from a file on the Web.

In addition to reading data from a local file on a computer or file server, you can also access data from a file that is on the Web if you know the file's URL (Uniform Resource Locator—the unique address for a file on the Web). For example, `www.google.com/index.html` is the URL for the file **index.html** located on the Google web server. When you enter the URL in a Web browser, the Web server sends the data to your browser, which renders the data graphically. Figure 12.10 illustrates how this process works.

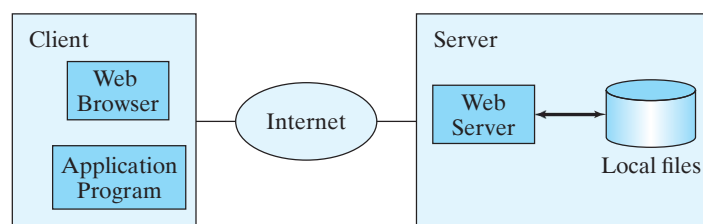


FIGURE 12.10 The client retrieves files from a Web server.

For an application program to read data from a URL, you first need to create a **URL** object using the **java.net.URL** class with this constructor:

```
public URL(String spec) throws MalformedURLException
```

For example, the following statement creates a URL object for `http://www.google.com/index.html`.

```
1 try {
2     URL url = new URL("http://www.google.com/index.html");
3 }
```

```

4 catch (MalformedURLException ex) {
5     ex.printStackTrace();
6 }

```

A **MalformedURLException** is thrown if the URL string has a syntax error. For example, the URL string `http:www.google.com/index.html` would cause a **MalformedURLException** runtime error because two slashes (`//`) are required after the colon (`:`). Note the `http://` prefix is required for the **URL** class to recognize a valid URL. It would be wrong if you replace line 2 with the following code:

```
URL url = new URL("www.google.com/index.html");
```

After a **URL** object is created, you can use the `openStream()` method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

```
Scanner input = new Scanner(url.openStream());
```

Now you can read the data from the input stream just like from a local file. The example in Listing 12.17 prompts the user to enter a URL and displays the size of the file.

LISTING 12.17 ReadFileFromURL.java

```

1 import java.util.Scanner;
2
3 public class ReadFileFromURL {
4     public static void main(String[] args) {
5         System.out.print("Enter a URL: ");
6         String urlString = new Scanner(System.in).next();
7
8         try {
9             java.net.URL url = new java.net.URL(urlString);
10            int count = 0;
11            Scanner input = new Scanner(url.openStream());
12            while (input.hasNext()) {
13                String line = input.nextLine();
14                count += line.length();
15            }
16
17            System.out.println("The file size is " + count + " characters");
18        }
19        catch (java.net.MalformedURLException ex) {
20            System.out.println("Invalid URL");
21        }
22        catch (java.io.IOException ex) {
23            System.out.println("I/O Errors: no such file");
24        }
25    }
26 }

```

enter a URL

create a URL object

create a Scanner object
more to read?
read a line

MalformedURLException

IOException

Enter a URL: `http://liveexample.pearsoncmg.com/data/Lincoln.txt`
The file size is 1469 characters

↵ Enter



Enter a URL: `http://www.yahoo.com`
The file size is 190006 characters

↵ Enter



MalformedURLException

The program prompts the user to enter a URL string (line 6) and creates a `URL` object (line 9). The constructor will throw a `java.net.MalformedURLException` (line 19) if the URL isn't formed correctly.

The program creates a `Scanner` object from the input stream for the URL (line 11). If the URL is formed correctly but does not exist, an `IOException` will be thrown (line 22). For example, `http://google.com/index1.html` uses the appropriate form, but the URL itself does not exist. An `IOException` would be thrown if this URL was used for this program.



12.12.1 How do you create a `Scanner` object for reading text from a URL?



12.13 Case Study: Web Crawler

This case study develops a program that travels the Web by following hyperlinks.

The World Wide web, abbreviated as WWW, W3, or Web, is a system of interlinked hypertext documents on the Internet. With a web browser, you can view a document and follow the hyperlinks to view other documents. In this case study, we will develop a program that automatically traverses the documents on the Web by following the hyperlinks. This type of program is commonly known as a *web crawler*. For simplicity, our program follows the hyperlink that starts with `http://`. Figure 12.11 shows an example of traversing the Web. We start from a Webpage that contains three URLs named `URL1`, `URL2`, and `URL3`. Following `URL1` leads to the page that contains three URLs named `URL11`, `URL12`, and `URL13`. Following `URL2` leads to the page that contains two URLs named `URL21` and `URL22`. Following `URL3` leads to the page that contains four URLs named `URL31`, `URL32`, `URL33`, and `URL34`. Continue to traverse the Web following the new hyperlinks. As you see, this process may continue forever, but we will exit the program once we have traversed 100 pages.

web crawler

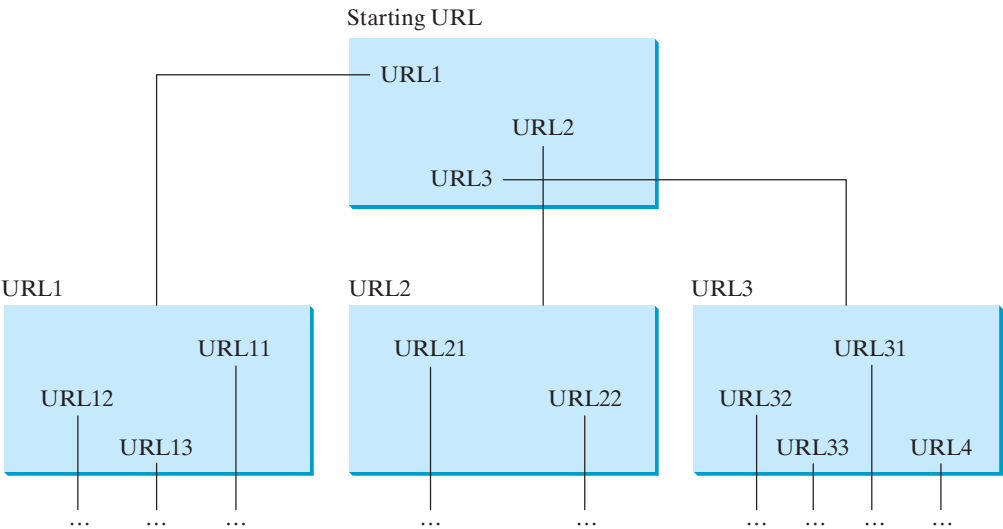


FIGURE 12.11 Web crawler explores the web through hyperlinks.

The program follows the URLs to traverse the Web. To ensure that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing, and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty and size of listOfTraversedURLs
<= 100 {
```

```

    Remove a URL from listOfPendingURLs;
    if this URL is not in listOfTraversedURLs {
        Add it to listOfTraversedURLs;
        Display this URL;
        Read the page from this URL and for each URL contained in the page {
            Add it to listOfPendingURLs if it is not in listOfTraversedURLs;
        }
    }
}

```

Listing 12.18 gives the program that implements this algorithm.

LISTING 12.18 WebCrawler.java

```

1  import java.util.Scanner;
2  import java.util.ArrayList;
3
4  public class WebCrawler {
5      public static void main(String[] args) {
6          Scanner input = new Scanner(System.in);
7          System.out.print("Enter a URL: ");
8          String url = input.nextLine();
9          crawler(url); // Traverse the Web from the a starting url
10     }
11
12     public static void crawler(String startingURL) {
13         ArrayList<String> listOfPendingURLs = new ArrayList<>();
14         ArrayList<String> listOfTraversedURLs = new ArrayList<>();
15
16         listOfPendingURLs.add(startingURL);
17         while (!listOfPendingURLs.isEmpty() &&
18             listOfTraversedURLs.size() <= 100) {
19             String urlString = listOfPendingURLs.remove(0);
20             if (!listOfTraversedURLs.contains(urlString)) {
21                 listOfTraversedURLs.add(urlString);
22                 System.out.println("Crawl " + urlString);
23
24                 for (String s: getSubURLs(urlString)) {
25                     if (!listOfTraversedURLs.contains(s))
26                         listOfPendingURLs.add(s);
27                 }
28             }
29         }
30     }
31
32     public static ArrayList<String> getSubURLs(String urlString) {
33         ArrayList<String> list = new ArrayList<>();
34
35         try {
36             java.net.URL url = new java.net.URL(urlString);
37             Scanner input = new Scanner(url.openStream());
38             int current = 0;
39             while (input.hasNext()) {
40                 String line = input.nextLine();
41                 current = line.indexOf("http:", current);
42                 while (current > 0) {
43                     int endIndex = line.indexOf("\\\"", current);
44                     if (endIndex > 0) { // Ensure that a correct URL is found
45                         list.add(line.substring(current, endIndex));
46                         current = line.indexOf("http:", endIndex);
47                     }

```

enter a URL
crawl from this URL

list of pending URLs
list of traversed URLs

add starting URL

get the first URL

URL traversed

add a new URL

read a line
search for a URL
end of a URL

URL ends with "
extract a URL
search for next URL

```

48         else
49             current = -1;
50     }
51 }
52 }
53 catch (Exception ex) {
54     System.out.println("Error: " + ex.getMessage());
55 }
56
57 return list;
58 }
59 }

```

return URLs



```

Enter a URL: http://cs.armstrong.edu/liang
Crawl http://www.cs.armstrong.edu/liang
Crawl http://www.cs.armstrong.edu
Crawl http://www.armstrong.edu
...

```

The program prompts the user to enter a starting URL (lines 7 and 8) and invokes the `crawler(url)` method to traverse the Web (line 9).

The `crawler(url)` method adds the starting url to `listOfPendingURLs` (line 16) and repeatedly process each URL in `listOfPendingURLs` in a while loop (lines 17–29). It removes the first URL in the list (line 19) and processes the URL if it has not been processed (lines 20–28). To process each URL, the program first adds the URL to `listOfTraversedURLs` (line 21). This list stores all the URLs that have been processed. The `getSubURLs(url)` method returns a list of URLs in the webpage for the specified URL (line 24). The program uses a foreach loop to add each URL in the page into `listOfPendingURLs` if it is not in `listOfTraversedURLs` (lines 24–27).

The `getSubURLs(url)` method reads each line from the webpage (line 40) and searches for the URLs in the line (line 41). Note a correct URL cannot contain line break characters. Therefore, it is sufficient to limit the search for a URL in one line of the text in a webpage. For simplicity, we assume that a URL ends with a quotation mark " (line 43). The method obtains a URL and adds it to a list (line 45). A line may contain multiple URLs. The method continues to search for the next URL (line 46). If no URL is found in the line, current is set to `-1` (line 49). The URLs contained in the page are returned in the form of a list (line 57).

The program terminates when the number of traversed URLs reaches 100 (line 18).

This is a simple program to traverse the Web. Later, you will learn the techniques to make the program more efficient and robust.



12.13.1 Before a URL is added to `listOfPendingURLs`, line 25 checks whether it has been traversed. Is it possible that `listOfPendingURLs` contains duplicate URLs? If so, give an example.

12.13.2 Simplify the code in lines 20–28 as follows: 1. Delete lines 20 and 28; 2. Add an additional condition `!listOfPendingURLs.contains(s)` to the if statement in line 25. Write the complete new code for the while loop in lines 20–29. Does this revision work?

KEY TERMS

absolute file name 499
 chained exception 495
 checked exception 483
 declare exception 484
 directory path 499

exception 476
 exception propagation 485
 relative file name 499
 throw exception 479
 unchecked exception 483

CHAPTER SUMMARY

1. Exception handling enables a method to throw an exception to its caller.
2. A Java *exception* is an instance of a class derived from `java.lang.Throwable`. Java provides a number of predefined exception classes, such as `Error`, `Exception`, `RuntimeException`, `ClassNotFoundException`, `NullPointerException`, and `ArithmeticException`. You can also define your own exception class by extending `Exception`.
3. Exceptions occur during the execution of a method. `RuntimeException` and `Error` are *unchecked exceptions*; all other exceptions are *checked*.
4. When *declaring a method*, you have to declare a checked exception if the method might throw it, thus telling the compiler what can go wrong.
5. The keyword for declaring an exception is `throws`, and the keyword for throwing an exception is `throw`.
6. To invoke the method that declares checked exceptions, enclose it in a `try` statement. When an exception occurs during the execution of the method, the `catch` block catches and handles the exception.
7. If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.
8. Various exception classes can be derived from a common superclass. If a `catch` block catches the exception objects of a superclass, it can also catch all the exception objects of the subclasses of that superclass.
9. The order in which exceptions are specified in a `catch` block is important. A compile error will result if you specify an exception object of a class after an exception object of the superclass of that class.
10. When an exception occurs in a method, the method exits immediately if it does not catch the exception. If the method is required to perform some task before exiting, you can catch the exception in the method and then rethrow it to its caller.
11. The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block, or whether an exception is caught if it occurs.
12. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
13. Exception handling should not be used to replace simple tests. You should perform simple test using `if` statements whenever possible and reserve exception handling for dealing with situations that cannot be handled with `if` statements.
14. The `File` class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.
15. You can use `Scanner` to read string and primitive data values from a text file and use `PrintWriter` to create a file and write data to a text file.
16. You can read from a file on the Web using the `URL` class.