MT201

Unit 8

Basic input/output

Course team

Developer: Herbert Shiu, Consultant

Designer: Dr Rex G Sharman, OUHK

Coordinator: Kelvin Lee, OUHK

Member: Dr Vanessa Ng, OUHK

External Course Assessor

Professor Jimmy Lee, Chinese University of Hong Kong

Production

ETPU Publishing Team

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong

Contents

Introduction

Up to this point in *MT201*, you have learned computing fundamentals about hardware/software, the software development cycle with respect to the object oriented paradigm, the syntax and various constructs of the Java programming language and the ways the language supports the three key concepts of an object oriented programming language: encapsulation, inheritance and polymorphism.

In previous units, you wrote class definitions in the Java programming language to solve various problems. From now on, the units concentrate on ways to develop programs for solving problems in particular areas, such as performing data input/output operations, creating graphical user interfaces and so on. The Java standard software library provides plenty of classes that you can use to perform such operations. What you have to do is to extend or use existing classes provided by the software library.

Unit 8 discusses how to perform input/output operations with software applications written in the Java programming language. As files stored on secondary storage such as a computer drive are common data stores, we discuss how to manipulate files.

The memory of the JVM stores all objects and data to be manipulated by the software. Therefore, reading and writing data concerns the flow of data between the JVM and the outside world, such as a file. A flow of data is considered a data stream. Basically, the JVM supports two types of stream, byte-based and character-based streams, corresponding to two primitive types `byte` and `char` in the Java programming language. We discuss the significance of these two stream types in detail.

Reading and writing operations are not necessarily successful every time. For example, if a hard disk is full, subsequent writing of data to a file on the hard disk becomes impossible. This is a runtime problem that an enterprise-level and robust software application should handle. Such runtime problems (due to the occurrence of exceptional conditions that a software application may encounter) are known as *exceptions* in the Java programming language. We discuss how to handle exceptions at runtime and you will then realize why exception handling has become an indispensable facility in software development, because it enables the language to be an excellent tool for developing enterprise-level or mission-critical software applications.

While you execute a software application written in the Java programming language, the keyboard and the screen of your computer are considered the standard input stream and standard output stream of the software application. The standard input stream and standard output stream can act as a data source (if you type a key on your keyboard) and a data destination (if your program calls the `System.out.println()` method) respectively. Therefore, you can apply your understanding of manipulating data streams to handle the keyboard and the screen. You can then read the keystrokes typed by the user and display the output on the screen. Besides the standard output stream, a software application has a stream known as standard error stream. We discuss how and why the standard error stream can be used for displaying errors or debugging messages.

While we are discussing how to manipulate data streams, sample programs are provided to integrate the classes for stream manipulations with some classes we discussed in previous units so that you can understand how input/output operations can be practically applied. Furthermore, the unit provides appendices containing supplementary materials for you to acquire a more complete understanding of the topics introduced in *Unit 8*.

Objectives

On completing *Unit 8*, you should be able to:

1. *Manipulate* files in Java.
2. *Classify* byte stream and character streams.
3. *Explain* Java exceptions and their handling.
4. *Apply* Java streams.
5. *Manipulate* the standard streams.

Using files in Java

A file is an entity on a secondary storage device, such as the hard disk in your computer or a floppy diskette, which enables you to store data. The contents of a file can be plain text, image data or compiled executable binary data. The storage is not concerned with the contents. You can write programs to read data from an existing file for processing or to write operation results to a file so that the data can be read for further processing at a later time.

For example, after you have written a class definition and saved it to a file, probably on your computer hard disk, you can execute compiler software (`javac`) to read it and compile it, and the resultant class file is saved in the computer hard disk. With respect to the compiler software, the class definition file (with the file extension `.java`) is the source of the data to be handled. After processing (compilation), the result (the compiled class definition in Java bytecodes) is written to a compiled class file (with the file extension `.class`). Later on, when you execute the JVM and specify a class name, the JVM reads the compiled class file and performs operations according to the contents of the class file.

Some programming languages provide functions and statements for performing input/output operations. The Java programming language provides no built-in facility for manipulating files, but the well-rounded Java standard software library that comes with the JRE provides the necessary classes for performing such operations. In this unit, the related classes are introduced one by one so that you can gradually understand how Java programs perform these operations.

Most classes for performing input/output operations are in the `java.io` package. Therefore, you should notice that all example programs and exercises that perform input/output operations need suitable `import` statements, such as

```
import java.io.*;
```

or

```
import java.io.File;
import java.io.FileInputStream;
```

for resolving the classes used in the class definitions. The former `import` statement instructs the compiler software to resolve all classes in the `java.io` package. The latter resolves the classes `File` and `FileInputStream` to be `java.io.File` and `java.io.FileInputStream` respectively. In most cases, the former is simpler and is therefore used throughout the class definitions in this unit.

The discussions on using the mentioned classes in this and subsequent units are not exhaustive, because we can only cover the common uses of their attributes and methods. If you want to know the complete list of methods, attributes and their detailed operations, you should refer to the documentation prepared by the developers of the classes. Please refer to

Appendix A at the end of the unit for a discussion on how to use the documentation.

Creating and using file objects

The Java standard software library provides the `java.io.File` class (hereafter the `File` class, for simplicity) that enables you to manipulate a file, such as for getting its information or attributes. Please use the following reading to learn how to use `File` objects.

Reading

King, section 14.2, pp. 608–13

From the above reading, you know that the `File` class defines several overloaded constructors and the following is the most common:

```
public File(String pathname)
```

The constructor needs the reference of a `String` object containing the path name of a file. Table 8.1 provides some path name examples for your reference. Please notice that the conventions for different operating systems, mainly the Microsoft Windows family and UNIX, are different.

Table 8.1 Example path name for `File(String)` constructor

Supplied path name	**The referred file**
`"file1"`	The file with filename `file1` in the current directory in which you execute the program
`"C:\\file2.txt"`	The file with filename `file2.txt` in the root directory of drive `C` (for Windows family)
`"A:\\text\\file3.txt"`	The file with filename `file3.txt` in the sub-directory `text` in the root directory of drive `A`, which is probably a floppy diskette (for Windows family)
`"/tmp/file4.doc"`	The file with filename `file4.doc` in the `tmp` sub-directory of the root directory (for machines running UNIX as the operating system)

The path separator (for delimiting the directory and file name in a path name) of the Windows family is a single '`\`' character and is written as '`\\`' in a `String` literal as shown in the example, `"C:\\file2.txt"`, in Table 8.1. The reason is that the '`\`' character and the subsequent character constitute a special character in `String` literals, such as '`\n`', and you therefore need to use '`\\`' to denote a single '`\`' character.

For example, to create a `File` object that refers to a file named `file.txt` in the current directory, the following statement can be used:

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

Creating a `File` object does not imply a file is created physically on your computer drive. It can refer to a non-existing file and you can use the `exists()` method to determine whether the file exists or not. You can also use other methods defined in the `File` class as listed in Table 8.2 to get further information about an existing file.

Table 8.2 The methods defined in the `File` class for getting information about a file/directory

Method name	**Return value**
`public boolean canRead()`	`true` if the file can be read; `false` otherwise
`public boolean canWrite()`	`true` if the file can be written; `false` otherwise
`public boolean exists()`	`true` if the file exists; `false` otherwise
`public String getAbsolutePath()`	the reference of a `String` object containing the absolute full path name of the file
`public String getName()`	the reference of a `String` object containing the file name
`public String getParent()`	the reference of a `String` object containing the name of the parent directory; `null` if no parent directory is supplied to the constructor
`public String getPath()`	the reference of a `String` object containing the full path name of the file supplied to the constructor
`public boolean isAbsolute()`	`true` if the path supplied to the constructor is absolute; `false` otherwise
`public boolean isDirectory()`	`true` if the name supplied to the constructor is referring to a physical directory; `false` otherwise
`public boolean isFile()`	`true` if the name supplied to the constructor is referring to a physical file; `false` otherwise
`public boolean isHidden()`	`true` if the file is hidden; `false` otherwise
`public long lastModified()`	the time when the file is last modified; the time is specified in number of seconds since 00:00:00 GMT, January 1, 1970
`public long length()`	the size of the file

Please note the following points:

1 A `File` object can refer to a file or a directory. The methods `isDirectory()` and `isFile()`can be used for the determination.

2 The parent directory of a file is the directory in which the file resides.

3 The full path is absolute if the full path name starts with a drive name for the Microsoft Windows family (such as `"C:\\file2.txt"`) or with a `/` character for UNIX (such as `"/tmp/file4.doc"`).

4 The `lastModified()` method returns the time the file was last modified as an integer of type `long` and is specified in the number of seconds since 00:00:00 GMT, January 1, 1970. We can use the `java.util.Date` class provided by the Java standard software library to determine the actual date and time from the number. The class `FileExpert` (Figure 8.1) (discussed very soon) illustrates how to interpret the time returned by the method `lastModified()`.

5 The methods shown in Table 8.2 above are those that are commonly used. For the complete method list, read the documentation of the `File` class.

Showing file information

With the methods shown in Table 8.2, it is possible to write a class, `FileExpert` as shown in Figure 8.1, to show the information about a file.

```
// Resolve I/O related classes and java.util.Date class
import java.io.*;
import java.util.Date;

// Definition of class FileExpert
public class FileExpert {

    // Show the information on a file
    public static void showFileInfo(String filename) {
        // Create a file object referring to the physical file
        File file = new File(filename);

        // Check if the file exists
        if (file.exists()) {
            // If the file exists, show its details
            System.out.println("Details of " + filename);
            System.out.println("Readable = " + file.canRead());
            System.out.println("Writeable = " + file.canWrite());
            System.out.println("Absolute path = " +
                file.getAbsolutePath());
            System.out.println("Name = " + file.getName());
            System.out.println("Parent directory = " +
                file.getParent());
            System.out.println("Path = " + file.getPath());
            System.out.println("Absolute file = " + file.isAbsolute());
```

```
            System.out.println("Refer to a directory = " +
                file.isDirectory());
            System.out.println("Refer to a file = " + file.isFile());
            System.out.println("Hidden = " + file.isHidden());
            System.out.println("Last modified = " +
                new Date(file.lastModified()));
            System.out.println("File size = " + file.length());
        }
        else {
            // Show the user the file does not exist
            System.out.println("The file " + filename +
                " does not exist.");
        }
    }
}
```

Figure 8.1 FileExpert.java

Calling the `lastModified()` method of a `File` object returns the last modified date/time of a file in number of seconds since 00:00:00 GMT, January 1, 1970. Then, with such a value, we can create a `java.util.Date` (or simply `Date`) object by supplying the value to its constructor, such as:

```
new Date(file.lastModified())
```

The created `Date` object represents that last modified date/time. In the definition of the class `FileExpert`, the `toString()` of the `Date` object is called implicitly (due to `String` concatenation) giving descriptive textual representation of the time.

To use the `FileExpert` class, a driver program is written as the `TestFile` class shown in Figure 8.2.

```
// Definiton of class TestFile
public class TestFile {

    // Main executive method
    public static void main(String args[]) {
        // Check if the user supplied the file name as program parameter
        if (args.length == 0) {
            // Show usage information
            System.out.println("Usage: java TestFile <file>");
        }
        else {
            // Create the FileExpert object
            FileExpert expert = new FileExpert();
            // Supply the file name to the FileExpert to show
            // its details
            expert.showFileInfo(args[0]);
        }
    }
}
```

Figure 8.2 TestFile.java

The `TestFile` class simply verifies whether a program parameter is provided, creates the `FileExpert` object and calls its `showFileInfo()` method with the program parameter as supplementary data.

Compile the classes and execute the `TestFile` class with a program parameter of a file name. The program will show the information of the specified file. For example, if the file `abc.txt` does not exist and the `TestFile` program is executed with `abc.txt` as the program parameter, that is

`java TestFile abc.txt`

the following output will be shown on the screen:

```
The file abc.txt does not exist.
```

To show the information of the compiled `TestFile` class, enter the following command in the Command Prompt:

`java TestFile TestFile.class`

The program shows the following output:

```
Details of TestFile.class
Readable = true
Writeable = true
Absolute path = C:\OUHK\Unit08\TestFile.class
Name = TestFile.class
Parent directory = null
Path = TestFile.class
Absolute file = false
Refer to a directory = false
Refer to a file = true
Hidden = false
Last modified = Thu Apr 03 11:53:26 CST 2003
File size = 520
```

Another example shows the information of the root directory of drive `C` of the computer by using the following command:

`java TestFile C:\`

A possible output of the program is:

```
Details of C:\
Readable = true
Writeable = true
Absolute path = C:\
Name =
Parent directory = null
Path = C:\
Absolute file = true
Refer to directory = true
Refer to file = false
Hidden = true
```

```
Last modified = Thu Apr 03 08:36:56 CST 2003
File size = 0
```

Please experiment with the `TestFile` program by executing it with a file or a directory on your computer drive. Based on the output of the program, you can develop a better understanding of the methods shown in Table 8.2.

File and directory information

If the `File` object refers to a physical file on your disk, it is possible to manipulate it further for reading. We discuss how to perform such operations in the subsequent subsections. If the `File` object refers to a directory on a computer drive, however, you can call its `list()` method to get an array object that maintains `String` objects with contents of the file names in the directory. For example, a class `DirLister1` is written in Figure 8.3 to show the file names of a specified directory.

```
// Resolve java.io.File class and java.util.Arrays class
import java.io.File;
import java.util.Arrays;

// Definition of class DirLister1
public class DirLister1 {

    // Show the files in the supplied directory
    public void showFilesInDir(String dirName) {
        // Create a File object to refer to the directory
        File dir = new File(dirName);

        // Check if the File object is referring to a directory
        if (dir.isDirectory()) {
            // If the File object is referring to a directory

            // Get the filenames in the directory
            String[] filenames = dir.list();

            // Sort the file names
            Arrays.sort(filenames);

            // Show a heading
            System.out.println("Directory : " + dirName +
                "\n----------------------------------------");

            // Show each file
            for (int i=0; i < filenames.length; i++) {
                System.out.println((i + 1) + ".\t" + filenames[i]);
            }

            // Show a footer
            System.out.println(
                "----------------------------------------" +
                "\nThere are totally " +
                filenames.length + " file(s)");
        }
```

```
        else {
            // Show the user the supplied directory is not a directory
            System.out.println("Sorry, the " + dirName +
                " is not referring to a directory.");
        }
    }
}
```

Figure 8.3 DirLister1.java

The array object returned by the `list()` method of the `File` class refers to `String` objects containing the file names, but the names are in an arbitrary order. Therefore, a utility method `sort()` of the class `java.util.Arrays` is used to sort the file names. The method `sort()` accepts the reference to an array object with element type `Object`. In *Unit 7*, we mentioned that all classes in the Java programming language are subclasses of the `Object` class. The `String` class is therefore a specific type of the general type `Object` and an array object with element type `String` can be treated as an array object with the element type `Object` to be supplied to the `sort()` method for sorting, such as

```
Arrays.sort(filenames);
```

in the `showFilesInDir()` method of the `DirLister1` class. You can experiment by removing the above statement from the method to observe the new output.

To test the `DirLister1` class, a driver program, the `TestDirLister1` class, is written as shown in Figure 8.4.

```
// Definiton of class TestDirLister1
public class TestDirLister1 {

    // Main executive method
    public static void main(String args[]) {
        // Check if the user supplied the directory name as
        // program parameter
        if (args.length == 0) {
            // Show usage information
            System.out.println("Usage: java TestDirLister1 <dir>");
        }
        else {
            // Create the DirLister1 object
            DirLister1 lister = new DirLister1();
            // Supply the directory name to the DirLister1 object
            // to show the files in the specified directory
            lister.showFilesInDir(args[0]);
        }
    }
}
```

Figure 8.4 TestDirLister.java

Compile the `DirLister1` and `TestDirLister1` classes, and execute the `TestDirLister1` with a program parameter that is the name of a directory, such as:

```
java TestDirLister1 C:\OUHK\Unit08
```

The program shows the files (sorted) in the directory:

```
Directory : C:\OUHK\Unit08
----------------------------------------
1. DirLister1.class
2. DirLister1.java
3. FileExpert.class
4. FileExpert.java
5. TestDirLister.class
6. TestDirLister.java
7. TestFile.class
8. TestFile.java
----------------------------------------
There are totally 8 file(s)
```

Detailed information of files

Now you know how to obtain the names of the files in a directory. What you might want to do next is to show detailed information about the files in the directory, such as their types (file or directory), file sizes and last modified dates. We can use a `FileExpert` object to show the file information by supplying the file names to it one by one in a `for` loop. As it is more presentable to show the file information in columns, the loop body of the `for` loop in the `showFilesInDir()` method of the class `DirLister1` is enhanced so that `File` objects are created in the method for showing the file information.

In the `for` loop, the expression `filenames[i]` refers to a particular `String` object containing the name of a file in the directory. Then, we can create a `File` object to refer to that physical file by supplying the full path name of the file to the constructor, such as:

```
File file = new File(dirName + "\\" + filenames[i]);
```

The variable `dirName` is the parameter of the `showFilesInDir()` method referring to a `String` object containing the directory name. The `String "\\"` is required in the `String` concatenation so that a proper full path name to the file can be derived for the Windows family. However, the default path separator for the UNIX machines is a single `/` character instead, which means that the above statement for UNIX should be modified as

```
File file = new File(dirName + "/" + filenames[i]);
```

so that a proper file name can be constructed and the program can execute correctly for computers running UNIX as the operating system.

Although the reading suggests that both slash (/) and backslash (\) can be used as path separators in a Java program, it is preferable not to handle such platform dependent issues in the program. Therefore, it is not desirable to prepare a `String` object of a full path name for creating the `File` object. In view of this, we can use other overloaded constructors of the `File` class to create the `File` object instead. The definitions of these constructors are:

```
public File(File parent, String child)
public File(String parent, String child)
```

The parameter `parent` specifies the parent directory of the file and the parameter `child` specifies the file name. You can use either one of the above two constructors; the difference between them is the type of the parameter `parent` to be supplied.

We can create a `File` object that refers to a file in the directory with either one of the following two statements:

```
File file = new File(dir, filenames[i]);
```

or

```
File file = new File(dirName, filenames[i]);
```

Variables `dir` and `dirName` are of type `File` and `String` respectively. Then, with the `File` object referred by the variable `file`, we can use the methods `length()`, `lastModified()`, and `isDirectory()` (or `isFile()`) to obtain its information, such as:

```
System.out.println(
  file.length() + "\t" +
  new Date(file.lastModified()) + "\t" +
  file.isDirectory() ? "Dir" : "File") + "\t" +
  filenames[i]);
```

A class `DirLister2` is derived based on `DirLister1` as shown in Figure 8.5. The `for` loop in the `showFilesInDir()` method is modified as mentioned above.

```
// Resolve File, Arrays and Date classes
import java.io.File;
import java.util.Arrays;
import java.util.Date;

// Definition of class DirLister2
public class DirLister2 {

    // Show the files in the supplied directory
    public void showFilesInDir(String dirName) {
        // Create a File object to refer to the directory
        File dir = new File(dirName);

        // Check if the File object is referring to a directory
        if (dir.isDirectory()) {
```

```
            // If the File object is referring to a directory

            // Get the filenames in the directory
            String[] filenames = dir.list();

            // Sort the file names
            Arrays.sort(filenames);

            // Show a heading
            System.out.println("Directory : " + dirName +
                "\nSize\tDate\t\t\t\tType\tName" +
                "\n----\t----\t\t\t\t----\t----");

            // Show each file
            for (int i=0; i < filenames.length; i++) {
                File file = new File(dir, filenames[i]);
                System.out.println(
                    file.length() + "\t" +
                    new Date(file.lastModified()) + "\t" +
                    (file.isDirectory() ? "Dir" : "File") + "\t" +
                    filenames[i]);
            }

            // Show a footer
            System.out.println(
                "\nThere are totally " +
                filenames.length + " file(s)");
        }
        else {
            // Show the user the supplied directory is not a directory
            System.out.println("Sorry, the " + dirName +
                " is not referring to a directory.");
        }
    }
}
```

Figure 8.5 DirLister2.java

To test `DirLister2`, a driver program `TestDirLister2` is written. As its definition is pretty much the same as that of the `TestDirLister1`, except the object to be created is a `DirLister2` object

```
// Create the DirLister2 object
   DirLister2 lister = new DirLister2();
```

the definition is not shown in the unit. You can find it on the course CD-ROM or course website.

Compile the classes and execute the `TestDirLister2` program with a program parameter of a directory name, such as:

```
java TestDirLister2 C:\OUHK\unit08\
```

It shows the files with their information in the specified directory. A possible output is:

```
Directory : C:\OUHK\unit08\
Size    Date                            Type    Name
----    ----                            ----    ----
1145    Thu Apr 03 13:21:08 CST 2003    File    DirLister1.class
1558    Thu Apr 03 13:14:58 CST 2003    File    DirLister1.java
1433    Thu Apr 03 13:21:08 CST 2003    File    DirLister2.class
1806    Thu Apr 03 13:21:04 CST 2003    File    DirLister2.java
1786    Thu Apr 03 13:21:10 CST 2003    File    FileExpert.class
1665    Thu Apr 03 11:50:46 CST 2003    File    FileExpert.java
538     Thu Apr 03 13:21:10 CST 2003    File    TestDirLister1.class
699     Thu Apr 03 13:14:54 CST 2003    File    TestDirLister1.java
538     Thu Apr 03 13:21:10 CST 2003    File    TestDirLister2.class
699     Thu Apr 03 13:18:12 CST 2003    File    TestDirLister2.java
519     Thu Apr 03 13:21:10 CST 2003    File    TestFile.class
622     Thu Apr 03 12:17:28 CST 2003    File    TestFile.java
0       Thu Apr 03 13:21:36 CST 2003    Dir     answers

There are totally 13 file(s)
```

You can experiment with the program with a directory on your computer drive. Please use the following self-test to test yourself on the use of the `File` class.

Self-test 8.1

Based on the definition of class `DirLister2` or otherwise, write a class named `DirUsage` with a `showDirUsage()` method, that is:

```
public void showDirUsage(String dirName) {
    ...
}
```

The parameter `dirName` of the method specifies the path name of the directory. The method finds and displays the number of files in the directory and the total sizes of all files.

Write the driver program `TestDirUsage` based on the definition of class `TestDirLister2` that accepts a program parameter of the path name of a directory. A sample execution of the `TestDirUsage` program is:

```
java TestDirUsage C:\

There are totally 15 file(s) (12351225 byte(s)) in the
directory C:\.
```

Now, we can proceed to the next subsection for performing reading and writing operations with files.

Reading and writing files

Performing reading and writing operations on a file can be considered the flow of data between two entities. For example, when the JVM is launched and is executing a Java program, we can imagine the JVM is a black box that is executing the statements of a program. Whenever a statement performs a reading operation, it is considered that there is a flow of data from the outside world into the JVM. Performing a writing operation, however, can be considered a flow of data from the JVM to the outside world. With respect to a file, a reading operation transfers the file contents (data) to the JVM, and a writing operation refers to the data flow out of the JVM and the data are written to a file as its contents.

The pattern of data flow mentioned above can be considered a stream of data, like the flow of water in a stream. The flow of water (data) is sequential and is aimed in a unique direction.

There are two fundamental types of data unit. They are the 8-bit byte and 16-bit (2-byte) Unicode characters, which correspond to the primitive types `byte` and `char` in the Java programming language. If an object supports a flow of data in units of `byte`, it is known as a *byte stream*, whereas a *character stream* supports a flow of data in units of `char`. Byte streams and character streams are usually used to handle (byte-oriented) binary files and (character-based) textual data respectively.

According to the directions of the data flow, they can be further sub-categorized as an *input stream* and an *output stream* that correspond to the flow of data into the JVM and the flow of data out of the JVM as shown in Figure 8.6. Input streams and output streams are usually nicknamed *source streams* and *destination* (or *sink*) *streams* respectively.

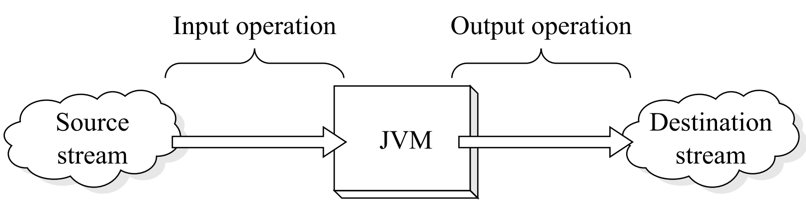


Figure 8.6 The input/output operations with respect to the JVM

In the following subsections, we discuss how to apply the idea of data streams to perform file reading and writing operations.

Byte streams

As the storage units of most storage media or communication channels are natively byte, byte-based streams are the fundamental way to perform input/output operations. Please use the following reading to learn how to perform input/output operations with byte streams.

Reading

King, ‘Reading and writing bytes’, pp. 613–18

A byte stream is a flow of data in units of `byte`, and it can be a byte-based input stream or output stream. In the Java software library, they correspond to two general (more exactly, `abstract`) classes `java.io.InputStream` and `java.io.OutputStream`. They define general types of how an input byte stream and an output byte stream should behave respectively. The Java standard software library defines some concrete subclasses of these general superclasses for particular storage media or communication channels.

The `OutputStream` class

The abstract superclass `java.io.OutputStream` (hereafter `OutputStream` for simplicity) defines overloaded `write()` methods so that the data in units of `byte` supplied to the method are written to a particular data destination, depending on the data destination the object associates with. After all data have been written to the data destination, it is generally preferable to call the `close()` method of the object to release the resources it acquired, such as the allocated memory, and to make sure the data supplied to the `write()` method calls are properly written to the data destination. Figure 8.7 visualizes the concept behind using an `OutputStream` object.

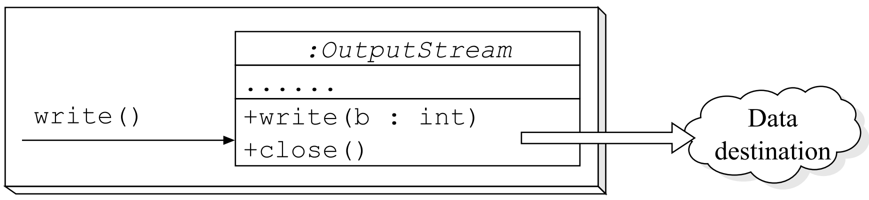


Figure 8.7 The scenario of writing a byte to a data destination via an `OutputStream` object

In Figure 8.7, the class name `OutputStream` is set in italics to highlight that there are no objects of the class `OutputStream` because it is an `abstract` class. Instead, it is an object of a concrete subclass of the superclass `OutputStream` that associates with a particular data destination at runtime (such as a file if the `OutputStream` object is actually a `FileOutputStream` object). Calling the `write()` method of the object with supplementary data in units of type `byte` will output the data to its associated destination (such as the data are written to a file for a `FileOutputStream` object).

You should notice that the parameter type of the `write()` method is `int` but the data to be written to the data destination are `byte`. Therefore, only the least significant eight bits of the supplied data of type `int` are written to the associated data destination.

Subsequent calls to the `write()` method of an `OutputStream` object will append the supplied data at the end of the associated data destination in units of `byte`. After writing the data to the `OutputStream`, it is usually good practice to call its `close()` method so that it can release the resources it acquires and the data are written to the associated data destination properly.

At runtime, for performing actual output operations, it is necessary to prepare a suitable concrete subclass object of the `abstract` superclass `OutputStream`; it can then be used as a general `OutputStream` object.

`FileOutputStream` — the concrete subclass of `OutputStream` for file

The class `java.io.FileOutputStream` (or simply `FileOutputStream`) is a specific type of the general type `OutputStream` for writing byte-based data to files. It therefore has all behaviours an `OutputStream` class defines. To create a `FileOutputStream`, it is necessary to specify the file to be written. The two common constructors of the class are:

`public FileOutputStream(File file)`, and
`public FileOutputStream(String name)`

In the former constructor, a `File` object that specifies a file is required. The latter one requires the reference of a `String` object with contents of the file path name. If the specified file exists on the disk, it is overwritten and the content of the original file is lost. Therefore, you should create a `FileOutputStream` object with care. If necessary, you can create a `File` object that specifies the file beforehand and call its `exists()` method to determine whether it is an existing file or not.

To experience writing bytes to a file on a disk, a class `BinaryFileCreator` is written as shown in Figure 8.8.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class BinaryFileCreator
public class BinaryFileCreator {

    // Create a binary file according to the file name and size
    public void create(String name) throws IOException {
        // Create a File object associated with physical file
        // specified by the file name
        File file = new File(name);

        // Create a FileOutputStream object and is handled as
        // if an OutputStream object
        OutputStream os = new FileOutputStream(file);

        // A for loop is used to write the byte to the
        // file via the FileOutputStream object
```

```
        for (int i = 0; i < 100; i++) {
            os.write(i);
        }
    }
}
```

Figure 8.8 BinaryFileCreator.java

To test the `BinaryFileCreator` class, a driver program `TestBinaryFileCreator` is written as shown in Figure 8.9.

```
// Resolve class in package java.io.
import java.io.*;

// Definition of class TestBinaryFileCreator
public class TestBinaryFileCreator {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage: java TestBinaryFileCreator <file>");
        }
        else {
            // Create a BinaryFileCreator object
            BinaryFileCreator creator = new BinaryFileCreator();

            // Create the binary file
            creator.create(args[0]);
        }
    }
}
```

Figure 8.9 TestBinaryFileCreator.java

You may have observed the following points from the definitions of the classes `BinaryFileCreator` and `TestBinaryFileCreator` as shown in Figure 8.8 and Figure 8.9.

1 There is a single `import` statement that helps resolve all classes in the `java.io` package. The classes to be resolved for the `BinaryFileCreator` class are `java.io.IOException`, `java.io.File`, `java.io.OutputStream`, and `java.io.FileOutputStream`. If individual `import` statements are used instead, the single `import` statement can be replaced by the following:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
```

You can easily figure out the classes to be used in the program with individual `import` statements. In contrast, a single `import` statement is a handier way for class name resolutions.

2 The `create()` method of the `BinaryFileCreator` and the `main()` method of the `TestBinaryFileCreator` are written as:

```
public void create(String name, int size) throws
IOException {
    ......
}

public static void main(String args[]) throws
IOException {
    ......
}
```

The `throws` clause '`throws IOException`' is added to fulfil the requirements of handling runtime errors. The issue is discussed in detail later in the unit. For the time being, you can treat it as a necessity for the method that performs input/output operations.

3 The `FileOutputStream` object is created by supplying a `File` object that refers to a physical file on the disk, so that the `FileOutputStream` associates with the file specified by the `File` object. As the `FileOutputStream` class defines another overloaded constructor

```
public FileOutputStream(String name)
```

it is possible to create the `FileOutputStream` object with the following statement instead:

```
FileOutputStream fos = new FileOutputStream(name);
```

The variable `name` is the parameter of the `create()` method, and the statement for creating the `File` object can be omitted.

4 In the `create()` method of the `BinaryFileCreator`, the `FileOutputStream` object created is referred by the variable `os` of type `OutputStream`.

```
OutputStream os = new FileOutputStream(file);
```

This is possible because `FileOutputStream` is a specific type (subclass) of the general type (superclass) `OutputStream`. Therefore, the `FileOutputStream` object can be treated as an `OutputStream` object to be referred by the variable `os` of type `OutputStream`.

Once the `OutputStream` object (actually a `FileOutputStream` object) is created, a `for` loop is executed to send a message `write` with a value in the sequence of bytes with values `0` to `99` repeatedly to the `OutputStream` object.

```
for (byte i=0; i < 100; i++) {
      os.write(i);
}
```

When the `FileOutputStream` object receives a message `write` with supplementary data of the type `byte`, it writes the data to its associated file. Because the parameter type of the `write()` method is `int`, the type of control variable in the above `for` loop can be of type `int` as well.

```
for (int i=0; i < 100; i++) {
      os.write(i);
}
```

Compile the `TestBinaryFileCreator` class and execute it, such as:

`java TestBinaryFileCreator C:\bytes.dat`

A file named `bytes.dat` is created in the root directory of drive `C` of the computer. If you examine its properties with Windows Explorer, the file size is 100 (bytes) as shown in Figure 8.10.

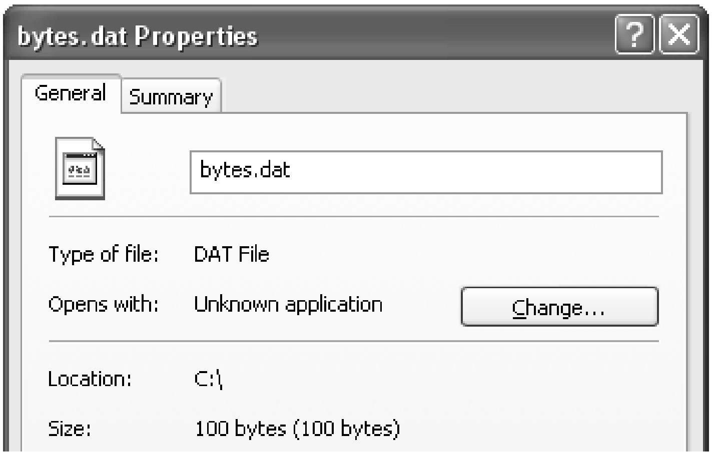


Figure 8.10 The property of the file `bytes.dat` shown by Windows Explorer

The overloaded `write()` methods of the `OutputStream` class

In addition to the `write()` method that accepts a parameter of type `int`, there are two overloaded `write()` methods. There are therefore three overloaded `write()` methods in total:

```
public void write(int b)
public void write(byte[] b)
public void write(byte[] b, int off, int len)
```

You are reminded once again that the parameter list of the first `write()` method is `int` rather than `byte` that only outputs an 8-bit single `byte` to the data destination. That is, only the least significant eight bits of the supplied 32-bit value of type `int` are written to the data destination.

The second and third overloaded `write()` methods accept an array object with elements of type `byte`. The bytes maintained by the array object that are supplied to the `write()` methods are written to the data destination. The difference between the second and third `write()` methods is that the second one writes all bytes maintained by the array object to the data destination, and the third one writes a portion of bytes (elements `b[off]` to `b[off+len-1]`), to the data destination.

The `InputStream` class

Now we have created a byte-based data file. We can experiment with reading the data (in bytes) from it by another program and show the data on the screen.

The concept behind performing a reading operation is similar to the writing operation, but the direction of the data flow is reversed. We need an object that can retrieve data from a particular data source and the data read are returned in units of `byte`. Such a type of object is modelled by the abstract class `java.io.InputStream` (or simply `InputStream`). The idea is visualized in Figure 8.11.

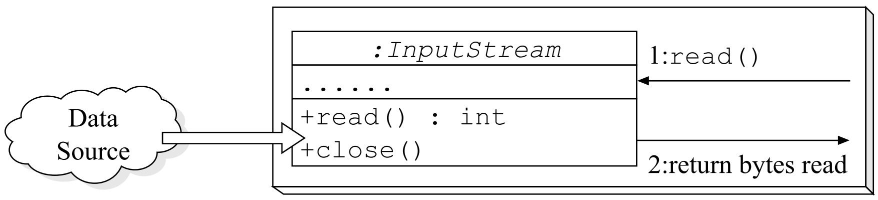


Figure 8.11 The scenario of reading a byte from a data source via an `InputStream` object

In Figure 8.11, as the class `InputStream` is `abstract`, there is no object of the `InputStream` class, so the name of `InputStream` object is set in italics. At runtime, the `InputStream` object is actually an object of a particular subclass of the `InputStream` that associates with a particular data source (for example, the `InputStream` object can be a `FileInputStream` object that associates with a file).

The behaviour `read` of an `InputStream` object returns a value of type `byte` from its associated data source one byte at a time. Subsequent calls to the `read()` method return the data from the data source in the order the data are maintained in the data source. When all data have been read from the data source, the `read()` method returns a value of `-1`. To release the resources associated for reading the `InputStream` object, you can call its `close()` method.

`FileInputStream` — the concrete subclass of `InputStream` class for file

The `FileInputStream` class is a concrete subclass of the abstract superclass `InputStream`; its `read()` method reads the associated file and returns a value of type `byte` from it. The `FileInputStream` provides three overloaded constructors. The two common ones are:

```
public FileInputStream(File file)
public FileInputStream(String name)
```

They are similar to the `FileOutputStream` class in that they accept a parameter of the references of a `File` object and a `String` object respectively.

The steps in reading data in units of `byte` from a file are as follows:

1 Creating a `File` object that refers to a file.

2 Creating a `FileInputStream` object by supplying the `File` object to the constructor so that the `FileInputStream` object is associated with the file. It is also possible to create a `FileInputStream` object by supplying the reference of a `String` object that contains the file name. Then, step 1 can be omitted.

3 Repeatedly calling the `read()` method of the `FileInputStream` object to read the data from the file in units of type `byte`. Then, the program can carry out its operations based on the data that have been read.

4 If the `read()` method returns a value of `-1`, all data in the file associated with the `FileInputStream` object (the one referred by the `File` object that is supplied to the `FileInputStream` constructor) have all been read. It is then preferable to call the `close()` method to release all allocated resources.

The above steps are implemented in the `show()` method of the `BinaryFileViewer` class shown in Figure 8.12.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class BinaryFileViewer
public class BinaryFileViewer {
    // The value to be returned for end of file
    private static final int EOF = -1;

    // Show the contents of a binary file
    public void show(String name) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);

        // Create a FileInputStream object specified by the File
        // object and is treated as a general InputStream object
        InputStream is = new FileInputStream(file);

        // Read the data from the input stream and show the data
        // on the screen
        System.out.println("Data read from file (" +
            file.getPath() + ")");
```

```
        int byteRead;
        int byteCount = 0;
        while ((byteRead = is.read()) != EOF) {
            byteCount++;
            System.out.print("Byte (#" + byteCount + ") = " +
                byteRead);

            // Treat the byte read as an ASCII character and show
            // it on the screen if its code is greater than 31
            if (byteRead > 31) {
                System.out.println(" <" + (char) byteRead + ">");
            }
            else {
                System.out.println();
            }
        }

        // Close the stream and release all corresponding resources
        is.close();
    }
}
```

Figure 8.12 BinaryFileViewer.java

The `show()` method of the `BinaryFileViewer` class is defined above. You should have no difficulty understanding it, except the condition of the `while` loop:

```
while ((byteRead = is.read()) != EOF) {
  byteCount++;
  System.out.println("Byte (#" + byteCount +
     ") = " + byteRead);
  }
```

For the condition of the `while` loop

```
(byteRead = is.read()) != EOF)
```

the parentheses govern the order of evaluating the expression. First of all, the expression enclosed in the inner parentheses is evaluated first, that is:

```
byteRead = is.read()
```

The `read()` method of the `InputStream` object (a `FileInputStream` object) referred by the variable `is` is called to read a datum of type `byte` from the associated data source (the file). Te data read are assigned to the variable `byteRead` by the = operator. The side effect of the = operator is the overall value of the expression (`byteRead = is.read()`) is the value assigned to the variable `byteRead`. Therefore, if the datum read from the data source is `10`, the overall value of the expression (`byteRead = is.read()`) is `10` as well.

Then, the overall value of the expression (`byteRead = is.read()`), for example `10`, is compared with the `final` variable `EOF` (with value

-1) by the != operator. As the values on both sides are unequal, the result is evaluated to be true, and the while loop continues.

When the reading operation has reached the end of the file, the data read from the input stream is -1 , the variable byteRead is assigned the value -1 and the comparison with the variable EOF using the != operator becomes false. As a result, the while loop terminates immediately.

To test class BinaryFileViewer, a driver program TestBinaryFileViewer is written in Figure 8.13.

```
// Resolve class in package java.io.
import java.io.*;

// Definition of class TestBinaryFileCreator
public class TestBinaryFileViewer {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage: java TestBinaryFileViewer <file>");
        }
        else {
            // Create a BinaryFileViewer object
            BinaryFileViewer viewer = new BinaryFileViewer();

            // Show the contents of the file
            viewer.show(args[0]);
        }
    }
}
```

Figure 8.13 TestBinaryFileViewer.java

Compile the classes and execute the TestBinaryFileViewer class with the file byte.dat we created earlier, that is:

java TestBinaryFileViewer c:\byte.dat

The following output is shown on the screen.

```
Data read from file (c:\bytes.dat)
Byte (#1) = 0
Byte (#2) = 1
Byte (#3) = 2
Byte (#4) = 3
......
Byte (#98) = 97 <a>
Byte (#99) = 98 <b>
Byte (#100) = 99 <c>
```

(Most of the output is omitted to save space.)

If the bytes read are treated as characters, those with values greater than `31` are displayable characters and are shown on the screen. For example, the values `97`, `98` and `99` correspond to the characters `a`, `b` and `c` respectively.

The overloaded `read()` methods of the `InputStream` class

Like the `write()` method of the `OutputStream` class, there are three overloaded `read()` methods defined by the `InputStream` class. They are:

```
public int read();
public int read(byte[] b);
public int read(byte[] b, int off, int len);
```

The first `read()` method is the one used in the `show()` method of the `BinaryFileViewer` class. It reads a datum (an 8-bit `byte`) from the data source and returns it as a value of type `int`. As the type of the return value is 32-bit `int` and the data read from the data source is of type `byte` (8-bit), only the least significant 8 bits of the return value is read as the datum from the data source. (All remaining 24 bits of the return value of type `int` are zeros if a byte is properly read from the data source.) Therefore, to restore the datum (a byte) read from an `InputStream` object, a casting operation is required, such as:

```
byte byteRead = (byte) is.read();
```

The second and third `read()` methods retrieve the data (bytes) from the data source and store them in the supplied array object. The return values of the methods indicate the numbers of bytes read. The second `read()` method stores the data read from the data source to the array object starting from the first array element, and the maximum number of bytes to be read is limited by the size of the supplied array object. The third `read()` method stores the bytes read starting at the array element with the subscript specified by the parameter `off` and the maximum number of bytes to be read is specified by the parameter `len`.

Please use the following self-test to experiment with writing some data — a sequence of random numbers — to a data file so that the numbers can be read and displayed by another program.

Self-test 8.2

Based on the class `BinaryFileCreator` or otherwise, write a class `RandomFileCreator` that writes random numbers ranging from `0` to `127` to a data file as specified by the program parameter. The number of random numbers to be written to the file is a random number between `50` and `200` inclusive. Write a driver program `TestRandomFileCreator` based on the `TestBinaryFileCreator`. Execute the `TestRandomFileCreator` program to generate the file, and use the `TestBinaryFileViewer` class as shown in Figure 8.12 to show the data that have been written.

A generic class for copying data from an `InputStream` object to an `OutputStream` object

In the last reading, a class `CopyFile1` was written to make a copy of a file. Here, an alternative approach is provided. For copying a file, it is necessary to create a source stream and a destination stream that are associated with the source file and destination file respectively. Then, it is possible to copy the contents from the source stream to the destination stream byte by byte. A class `StreamCopier` is defined in Figure 8.14 for performing the copying operations.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class StreamCopier
public class StreamCopier {
    // Constant for end of file
    private static final int EOF = -1;

    // Copy the contents from the source stream to
    // destination stream
    public void copy(InputStream in, OutputStream out)
        throws IOException {

        // The byte read during copying
        int byteRead;

        // Read a byte from the source stream and write it to the
        // destination stream as long as it is not end of file
        while ((byteRead = in.read()) != EOF) {
            out.write(byteRead);
        }
    }
}
```

Figure 8.14 StreamCopier.java

The `copy()` method of the `StreamCopier` class accepts the reference of an `InputStream` object and an `OutputStream` object. As these two classes are `abstract`, the references supplied to the constructor must be the references to objects of concrete subclasses of the two abstract superclasses `InputStream` and `OutputStream`.

To use the above `StreamCopier` class for copying a file, it is necessary to prepare an `InputStream` and an `OutputStream` object that are associated with the source file and destination file respectively. Therefore, we can use the two concrete subclasses of the `InputStream` and `OutputStream` classes that are for file input/output operations; that is, a `FileInputStream` object and a `FileOutputStream` object.

Once the `FileInputStream` and the `FileOutputStream` object are created and are associated with the source file and destination file, it is possible to use a `StreamCopier` object to copy the contents from the source file to the destination file. These steps are used in the definition of class `CopyFile1` in Figure 8.15.

```
// Resolve classes in the java.io package
import java.io.*;

// Definition of class CopyFile1
public class CopyFile1 {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if two files are specified as program parameters
        if (args.length < 2) {
            // If two files are not specified, show usage information
            System.out.println(
                "Usage: java CopyFile1 <source> <destination>");
        }
        else {
            // Create File objects to refer to the physical files
            File inFile = new File(args[0]);
            File outFile = new File(args[1]);

            // Create InputStream/OutputStream that associate to the
            // two File objects and hence the physical files
            InputStream in = new FileInputStream(inFile);
            OutputStream out = new FileOutputStream(outFile);

            // Create a StreamCopier object
            StreamCopier copier = new StreamCopier();

            // Copy the contents from the InputStream object to the
            // OutputStream object
            copier.copy(in, out);

            // Close the InputStream/OutputStream and release all
            // related resources
            in.close();
            out.close();
        }
    }
}
```

Figure 8.15 CopyFile1.java

The core part of the `StreamCopier` class `copy()` method is the `while` loop in which a datum of type `byte` is read from the data source by calling the `read()` method of the `InputStream` (a `FileInputStream`) object, which is then written to the data destination by calling the `write()` method of the `OutputStream` object (a `FileOutputStream` object). The underlying operations can be visualized in Figure 8.16.

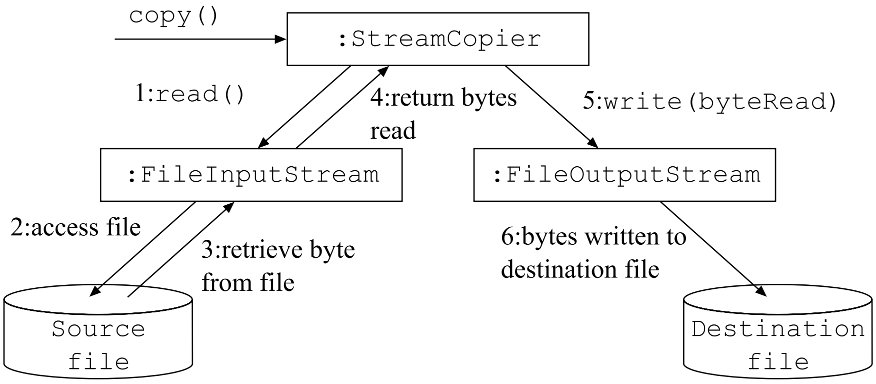


Figure 8.16 The operation sequences behind the `copy()` method of the `StreamCopier` class

Compile and execute `CopyFile1` with two file names as program parameters. The first program parameter is the source file name and the second one is the destination file. If the source file does not exist or the destination file is either invalid or non-writable, the program will cause runtime errors and error messages will be shown on the screen.

Please use the following self-test to experiment with using the `StreamCopier` class for retrieving a resource from the World Wide Web.

Self-test 8.3

The Java standard software library provides the `java.net.URL` class (or simply `URL` class) for associating a resource on the World Wide Web (WWW). For example, the following URL, http://www.ouhk.edu.hk/WEB/images/testing/top_logo.gif specifies a resource of an image file (a GIF file). Once a `URL` object is created successfully, it is possible to call its `openStream()` method to obtain the reference of an `InputStream` object associating with the contents of the resource. You can then call the `read()` method of the `InputStream` object for getting the data from the WWW resource.

Based on the definition of the class `CopyFile1` shown in Figure 8.15, write a `URLGetter` class that retrieves a resource from the WWW and stores the contents in a file on a computer drive. For example, the following command

```
java URLGetter http://www.ouhk.edu.hk/WEB/images/
testing/top_logo.gif top_logo.gif
```

retrieves the resource from the WWW and stores the resource to a file `top_logo.gif`.

Hints:

1 To create a `URL` object, the constructor accepts the reference of a `String` as the hyperlink, such as:

```
URL url = new
URL("http://www.ouhk.edu.hk/WEB/images/testing/
top_logo.gif");
```

2 The first program parameter that is supplied to the `URLGetter` program is the `URL` for retrieval, and it is therefore supplied to the constructor of the `URL` object.

3 To get the `InputStream` object from the `URL` object, call its `openStream()` method, that is:

```
InputStream in = url.openStream();
```

4 Like the definition of the class `CopyFile1`, the second program parameter is the target file. The way to create the `OutputStream` object is the same.

5 Once the `InputStream` object and the `OutputStream` object are ready, it is then possible to use a `StreamCopier` object as in the definition of the `CopyFile1` class (Figure 8.15) to copy the contents from the WWW resource to the specified file.

Compare the definitions of the classes `CopyFile1` and `URLGetter`. You can see that their structures are similar. An even more important observation is that `StreamCopier` handles a generic `InputStream` and a generic `OutputStream` object. Therefore, provided that there are objects of concrete subclasses of abstract superclasses `InputStream` and `OutputStream`, the `StreamCopier` class needs no modification and can work properly. Otherwise, if `StreamCopier` is written to handle a `FileInputStream` and `FileOutputStream` explicitly, that is, the method `copy()` is written to be:

```
public void copy(FileInputStream in, FileOutputStream out) {
    ...
}
```

the `URLGetter` class cannot use the `StreamCopier` class because the `InputStream` object obtained from the `URL` object is not a `FileInputStream` object. Then, it is necessary to modify the definition of class `StreamCopier` so that the `URLGetter` class can use it.

The `StreamCopier` class illustrates a benefit of writing a class definition that handles the generic or abstract classes that we discussed in *Unit 7*. It leaves room for the developers to prepare objects of concrete subclasses (such as `FileInputStream` and `FileOutputStream`) of those abstract superclasses (`InputStream` and `OutputStream`). The class definition, `StreamCopier` in this case, can be used without any modification. For example, the data sources for `CopyFile1` and `URLGetter` are file and WWW resource respectively, but once an `InputStream` object is associated with a data source, the subsequent operations are the same.

Character streams

Another type of data stream is the character-based (16-bit Unicode) stream (corresponding to the Java primitive type `char`). Compared with byte (8-bit) streams, every reading/writing operation handles the data in units of `char` (16-bit or 2-byte). Therefore, character streams are usually used for handling textual data. In the Java programming language, any object that can act as an input character stream is generally considered a `Reader` object, and any object that can act as an output character stream is generally considered a `Writer` object.

`Reader` and `Writer` are abstract classes defining the common methods that an input character stream and output character stream object should process. There are concrete subclasses that enable reading/writing data in units of character with respect to particular storage media and communication channels, such as the classes `FileReader` and `FileWriter` for reading and writing files in units of character.

Writing character-based data with `Writer` objects

First of all, let's investigate how to write data in units of character to a data destination. To write character data to a data destination, we need a `Writer` object. At runtime, it can be, for example, a `FileWriter` object. Like the `OutputStream` class, the `Writer` class defines overloaded `write()` methods for writing the supplied characters to the data destination:

```
public void write(int c);
public void write(char[] cbuf);
public void write(char[] cbuf, int off, int len);
```

The first `write()` method accepts a parameter value of type `int` and the supplied value is written to the data destination. As data of type `int` and `char` take 32 bits and 16 bits respectively, only the least significant 16 bits of the supplied data of type `int` are written to the data destination.

The second and third overloaded `write()` methods write the characters maintained by the supplied array object to the data destination. The second `write()` method writes all characters maintained by the supplied array object to the data destination, whereas the third one writes the characters maintained by the array object specified by the parameters `off` (the starting subscript of the array element to be written) and `len` (the number of characters to be written) to the data destination.

Besides supplying the sequence of characters by an array object with element type `char`, the content of a `String` object can be considered a sequence of `char`. Therefore, the `Writer` class defines two more overloaded `write()` methods that accept supplementary data of a `String` object.

```
public void write(String str);
public void write(String str, int off, int len);
```

The first one writes the sequence of characters maintained by the `String` object to the data destination. The second one writes a portion of the character sequences to the data destination as specified by the parameters `off` and `len`.

Like the use of `OutputStream`, after the data are written to the data destination, it is a good programming practice to call the `close()` method of a `Writer` object to release all allocated resources and make sure the data supplied to the `Writer` object are properly stored or sent.

`FileWriter` — a concrete subclass of the `Writer` class for files

The `FileWriter` class is a concrete subclass of the `Writer` class for performing the writing of character-based data to files. To create a `FileWriter` object, it is necessary to supply the file name to the constructor. There are three overloaded constructors, but the following two are commonly used:

```
public FileWriter(File file)
public FileWriter(String name)
```

The first constructor accepts a `File` object that specifies a physical file. The second constructor accepts a `String` object with contents of the name of the physical file. Similar to create a `FileOutputStream`, if the specified file exists, it is overwritten. Otherwise, the specified file is created. If the specified file is invalid, or the file exists but cannot be overwritten, runtime errors will occur.

To experiment with using a `FileWriter` object, which is a concrete subclass of the abstract superclass `Writer`, a class `TextFileCreator1` is shown in Figure 8.17.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TextFileCreator1
public class TextFileCreator1 {

    // Create the file with the char array as its contents
    public void create(String name, char[] message) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);

        // Create a FileWriter object specified by the File
        // object and is treated as a general Writer object
        Writer writer = new FileWriter(file);

        // Send the character data to the Writer one by one
        for (int i = 0; i < message.length; i++) {
            writer.write(message[i]);
        }

        // Close the Writer and release all corresponding resources
        writer.close();
    }
}
```

Figure 8.17 TextFileCreator1.java

The `create()` method of the `TextFileCreator1` class accepts a `String` name and an array object with element of type `char` for the file name and the characters to be written. Compared with the `BinaryFileCreator`, the differences are a `FileWriter` object is created instead of `FileOutputStream` and the data to be written are of the type `char` instead of `byte`.

A corresponding driver program `TestTextFileCreator1` is shown in Figure 8.18.

```
// Resolve classes in java.io. package
import java.io.IOException;

// Definition of class TestTextFileCreator1
public class TestTextFileCreator1 {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage : java TestTextFileCreator1 <file>");
        }
```

```
        else {
            // The array of characters to be written
            char[] buffer = {
                'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};

            // Create a TextFileCreator1 object
            TextFileCreator1 creator = new TextFileCreator1();

            // Create the file and write the characters to it
            creator.create(args[0], buffer);
        }
    }
}
```

Figure 8.18 TestTextFileCreator1.java

Compile the classes and execute the `TestTextFileCreator1` with a file name as the program parameter, such as:

`java TestTextFileCreator1 out.txt`

The program will create a file `out.txt` and the message `"Hello World"` is written to the file. You can then use a common editor, such as Notepad, to view its content as in Figure 8.19.

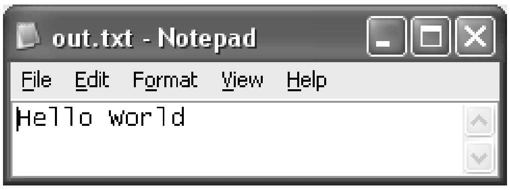


Figure 8.19 Viewing the file `out.txt` with Notepad

Reading character-based data with `Reader` objects

To read data from a character data stream, we need a `Reader` object. The abstract superclass `Reader` defines the following overloaded `read()` methods for reading characters from a character data stream:

```
public int read();
public int read(char[] cbuf);
public int read(char[] cbuf, int off, int len);
```

You can see that the `read()` methods are similar to those defined by the `InputStream`, except the data read are in units of `char` instead of `byte`. The first `read()` method with an empty parameter list returns a single character from the data source, and the type of return value is `int` instead of `char`. Therefore, only the least significant 16 bits of the return value represent the character that has been read. If necessary, the return value can be converted to a datum of type `char` by casting, such as:

```
char charRead = (char) reader.read();
```

If no character is available, such as all available data provided by the data source have been read, a value of `-1` is returned as the return value.

In the second and third `read()` methods, the characters read from the character stream are stored in the array object supplied to the methods. The maximum number of characters to be stored in the array objects by the second `read()` method is determined by the size of the array object supplied. In the third `read()` method, the parameter `off` specifies the starting array element subscript for storing the characters read, and the parameter `len` specifies the maximum number of characters to be read. The return values of both the second and third `read()` methods indicate the number of characters read.

After completing reading the data stream, it is also preferable to call the `close()` method of the `Reader` object to release all allocated resources.

`FileReader` — the concrete subclass of the `Reader` class for files

With the knowledge of using a `Reader` object, we can experiment with using an object of a concrete subclass, `FileReader`, of the abstract superclass `Reader`. A `FileReader` object is an input character stream that associates with a file on a computer drive. Like `FileInputStream`, there are two common ways to specify the physical file to be read by the `FileReader` object with the following two overloaded constructors:

```
public FileReader(File file)
public FileReader(String name)
```

The first constructor expects the reference of a `File` object that specifies a physical file. The second one accepts the reference of a `String` object that contains the full path name of a file. If the specified file does not exist, a runtime error will occur.

Once a `FileReader` object is successfully created, it is possible to call its `read()` method to retrieve the characters from the data stream that is associated with the file. A class `TextFileViewer1` is written in Figure 8.20 to illustrate how to use `FileReader` to read the contents as a sequence of characters from a file.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TextFileViewer1
public class TextFileViewer1 {
    // Final variable for end of file
    private static final int EOF = -1;

    // Show the contents of a text file
    public void show(String name) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);
```

```
        // Create a FileReader object specified by the File
        // object and is treated as a general Reader object
        Reader reader = new FileReader(file);

        // Read the data from the reader and show the data
        // on the screen
        System.out.println("Data read from file (" +
            file.getPath() + ")");
        int charRead;
        int charCount = 0;
        while ((charRead = reader.read()) != EOF) {
            charCount++;
            System.out.print("Char (#" + charCount + ") = " + charRead);
            // If the character is displayable characters
            // (the code is >= 32), show it
            if (charRead >= 32) {
                System.out.println(" ('" + (char) charRead + "')");
            } else {
                System.out.println();
            }
        }

        // Close the reader and release all corresponding resources
        reader.close();
    }
}
```

Figure 8.20 TextFileViewer1.java

A driver program `TestTextFileViewer1` is written to test the `TextFileViewer1`. As its definition is similar to the `TestBinaryFileViewer`, it is not shown here. You can find the complete definition from the course CD-ROM or website.

The condition of the `while` loop in the `show()` method reads a character from the `Reader` object (actually a `FileReader` object) and checks whether it equals `-1`. If so, all data have been read and the `while` loop terminates. Otherwise, the value (or code) of the character read is shown. If the code of a character is less than `32`, it is a special character that cannot be shown properly on the screen. Otherwise, the return value of type `int` is cast to type `char` to be shown on the screen.

Compile the classes and execute the `TestTextFileViewer1` class with the file `out.txt` as the program parameter, that is:

`java TestTextFileViewer1 out.txt`

The following output is shown on the screen:

```
Data read from file (out.txt)
Char (#1) = 72 ('H')
Char (#2) = 101 ('e')
Char (#3) = 108 ('l')
Char (#4) = 108 ('l')
Char (#5) = 111 ('o')
```

```
Char (#6) = 32 (' ')
Char (#7) = 87 ('W')
Char (#8) = 111 ('o')
Char (#9) = 114 ('r')
Char (#10) = 108 ('l')
Char (#11) = 100 ('d')
```

To summarize, the steps in performing input/output operations with classes provided by the Java software library are:

1 Determine whether the data to be handled are in units of byte or character for choosing byte stream or character stream. Byte stream and character stream are usually used for binary data and textual data respectively.

2 Determine whether the direction of the data flow is input or output. Depending on the determined unit of data in step 1, either byte or character, choose whether it is an `InputStream` (input byte stream), `Reader` (input character stream), `OutputStream` (output byte stream) or `Writer` (output character stream).

3 According to the type of media or storage, choose a class from the Java standard software library that is dedicated for manipulating that type of media or storage. For example, if the data source/destination is a file, the class to be chosen is a `FileInputStream` (input byte stream for file), `FileReader` (input character stream for file), `FileOutputStream` (output byte stream for file) or `FileWriter` (output character stream for file).

4 Create an object of the class determined in step 3 to associate with the specified media or storage. The ways to create the object depend on the constructors defined by the class. To find out the constructors defined by the class, you can read the documentation presented in Appendix A of the unit.

5 Call the `read()` or `write()` method of the object created in step 4 to read data from the source stream or write data to the destination stream.

6 After completing the reading or writing operation, call the `close()` method of the object to release the acquired resources.

The above steps describe a general way to manipulate a data source or data destination directly with byte streams or character streams. Usually, the data to be read or written are not necessarily in units of byte or character. The Java software library provides other classes so that it is possible to perform input/output operations with the structured data. We discuss some of them later in the unit.

Self-test 8.4

Please categorize the following files into binary and textual.

1 Image files (with extension `.gif` or `.jpg`)

2 Microsoft Word documents (with file extension `.doc`)

3 Java source code files (with file extension `.java`)

4 Java compiled class files (with file extension `.class`)

5 Web pages (with file extension `.htm` or `.html`)

Exception handling

For most class definitions we have seen so far, the methods that perform input/output operations need an extra `throws` clause in the method definitions, such as:

```
public static void main(String args[]) throws IOException {
    ......
}
```

Such a `throws` clause is mandatory, or a compile-time error will occur. It is related to the issue of exception handling. Please use the following reading to learn about exceptions. Afterwards, we elaborate on the concepts introduced in the reading, such as their importance in software development.

> ***Reading***
>
> King, section 8.1, ‘Exceptions’, pp. 300–6

What is an exception?

The reading suggests that an exception happens when a Java program performs an illegal operation, such as performing a division operation by zero at runtime. Exceptions may also occur from experiencing exceptional conditions, such as creating a `FileInputStream` object for a non-existing file. Therefore, the occurrence of an exception indicates a runtime error or problematic situation that the statement cannot handle. Then, the exception is generated to tell the program that such a problem has happened and the program should perform suitable remedial action to handle it. For example, if the program opens a file for reading but the file does not exist, the program should prompt the user for the file name again so that it can try to create a `FileInputStream` object for another file.

When a program encounters a problem or performs an illegal operation at runtime, an exception occurs and an exception object is created containing the information about the exceptional condition or the problem. Therefore, there are various exception classes indicating various runtime problems or errors. With the exception object, the program can determine what has happened and can then react appropriately.

Common exceptions

In the following subsections, some common exception types are introduced one by one with examples, so that you can understand the reasons why they occur and gain insight into preventing their occurrences.

The `NumberFormatException` exception

While you were executing the example programs discussed in the previous units, you may have encountered exceptions. For example, if you execute the `TestGreeting` program we discussed in *Unit 4*, a dialog box is shown on the screen for you to enter a number as the hour for determining the greeting to be shown (as in Figure 8.21).

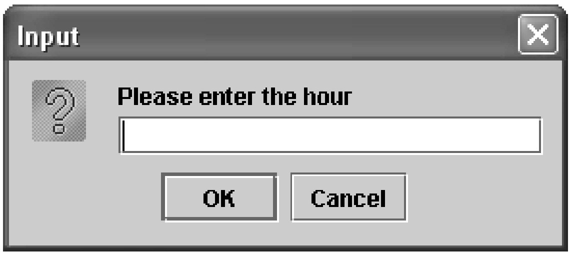


Figure 8.21 The dialog box shown by executing the `TestGreeting` program

If you enter a number as the hour, a dialog box with the proper greeting message will be shown. However, if you type nothing and click **OK** to complete the dialog, the dialog box disappears and the following message is shown in the Command Prompt of your computer:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: ""
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:447)
        at java.lang.Integer.parseInt(Integer.java:476)
        at TestGreeting.main(TestGreeting.java:20)
```

The message that looks like the above indicates an exception, or runtime error, has occurred. The occurrence of an exception usually terminates the program immediately. (For the `TestGreeting` program, because a dialog box is used and the method `System.exit()` has not been executed yet, the JVM does not terminate and it is necessary to terminate the JVM by pressing the keystroke <Ctrl-C>.)

The first line of the message shown includes the class name of the exception. In the above example, the class name of the exception is `java.lang.NumberFormatException` (or simply `NumberFormatException`) that indicates a runtime problem related to an invalid number format. The message that follows the class name provides further information. In the above example, the message discloses that the input string to be converted into an integer is an empty string, which is unacceptable; hence the `NumberFormatException` exception is displayed. The message starting from the second line indicates the sequence of method calls starting from the `main()` method of the `TestGreeting` class, so that you can trace the execution path.

The `ArrayIndexOutOfBoundsException` exception

The second common exception you have possibly encountered is the `java.lang.ArrayIndexOutOfBoundsException` (or simply `ArrayIndexOutOfBoundsException`) exception. To illustrate the exception, a class `ShowFirstArgs` is written in Figure 8.22 that accesses the first element of the array object that maintains the program parameters.

```
// Definition of class ShowFirstArgs
public class ShowFirstArgs {

    // Main executive method
    public static void main(String args[]) {
        // Show the first argument maintained by the array object
        System.out.println("The first program parameter is [" +
            args[0] + "].");
    }
}
```

Figure 8.22 ShowFirstArgs.java

Compile and execute the `ShowFirstArgs` with a program parameter, such as:

```
java ShowFirstArgs Hello
```

The following message is shown on the screen:

```
The first program parameter is [Hello].
```

However, if no program parameter is supplied and you execute the `ShowFirstArgs` as

```
java ShowFirstArgs
```

the `main()` method of the `ShowFirstArgs` intends to access the first element of the array object that maintains the program parameters, but the array object maintains no `String` object as no program parameter is provided. The size of the array object is zero. Therefore, accessing the first element of the array object, `args[0]`, causes an exception and the program terminates with the following message shown on the screen:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at ShowFirstArgs.main(ShowFirstArgs.java:7)
```

The message indicates the name of the exception is `ArrayIndexOutOfBoundsException` and it occurs while the statement in the seventh line of the source file `ShowFirstArgs.java` is executing. The message after the exception name indicates that the subscript of the array object to be accessed was zero.

The `ClassCastException` exception

Another runtime error that may occur is the improper casting of an object reference. For example, if the statement explicitly casts an object reference of superclass type to a subclass type but the object reference is not actually referring to that subclass object, it will cause an exception known as `java.lang.ClassCastException` (or simply `ClassCastException`). For example, the object reference referring to an `Object` class object is cast to an object reference of type `String`.

```
Object obj = new Object();
String str = (String) obj;
```

The compiler software accepts the statement because the statement explicitly performs the casting operation. However, when the second statement is executed at runtime, the object reference (referring to an `Object` class object) is verified whether it is actually a `String` object or not. If so, the casting operation succeeds. However, the variable `obj` refers to an `Object` class object, which is not a `String` object. Therefore, a runtime error or an exception will occur.

A class `TestCCException` is written in Figure 8.23, to illustrate.

```
// Definition of class TestCCException
public class TestCCException {

    // Main executive method
    public static void main(String args[]) {
        // Create an Object class object and is referred by a
        // variable of type Object
        Object obj = new Object();

        // Cast the reference to String type explicitly
        String str = (String) obj;

        // Show the textual representation of the object
        System.out.println(str);
    }
}
```

Figure 8.23 TestCCException.java

Compile and execute the `TestCCException` class, the following message is shown on the screen:

```
Exception in thread "main" java.lang.ClassCastException
     at TestCCException.main(TestCCException.java:11)
```

Your program will not be written in the same way as `TestCCException,` which was written for illustration only. Such an exception probably occurs in a method; an object reference is supplied via a parameter and is cast without prior verification with the `instanceof` operator.

The `ArithmeticException` exception

Performing mathematical operations is not necessarily successful at runtime, such as performing a division operation by zero. To illustrate such an exception, a class `FindAverage` is written in Figure 8.24 to find the average of the program parameters.

```
// Definition of class FindAverage
public class FindAverage {

    // Main executive method
    public static void main(String args[]) {
        // Initialize variables for finding the average
        int sum = 0;
        int count = 0;

        // Iterate each number for finding the total
        for (int i=0; i < args.length; i++) {
            sum += Integer.parseInt(args[i]);
            count++;
        }

        // Show the average of the numbers on the screen
        System.out.println(
            "The average of the numbers is " + sum / count);
    }
}
```

Figure 8.24 FindAverage.java

Compile the class and execute it with program parameters of some integer values, such as:

`java FindAverage 10 20`

The following message is shown on the screen:

```
The average of the numbers is 15
```

However, if the program is executed without any program parameters, the following message is shown on the screen:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at FindAverage.main(FindAverage.java:17)
```

The exception occurs because no program parameters are supplied when the program is executed. The entire `for` loop in the `main()` method is therefore skipped and the values of the variables `sum` and `count` are kept at zero. Finally, when the expression `sum / zero` is performed, the `java.lang.ArithmeticException` (or simply `ArithmeticException`) exception occurs.

The `NullPointerException` exception

The last common exception that a Java program may encounter is the `java.lang.NullPointerException` (or simply `NullPointerException`) exception. To illustrate a possible scenario of such occurrence, a class `GuessACard` is written in Figure 8.25.

```
// Definition of class GuessACard
public class GuessACard {

    // Main executive method
    public static void main(String args[]) {
        // Verify the number of program parameters
        if (args.length < 1) {
            // If no program parameter is supplied, show usage message
            System.out.println(
                "Usage: java GuessACard <Spade|Heart|Club|Diamond>");
        }
        else {
            // Get a random number
            int suitDrawn = (int) (Math.random() * 4);

            // Derive a suit name according to the random number
            String suitName = null;
            switch (suitDrawn) {
                case 1:
                    suitName = "Spade";
                    break;
                case 2:
                    suitName = "Heart";
                    break;
                case 3:
                    suitName = "Club";
                    break;
                case 4:
                    suitName = "Diamond";
                    break;
            }

            // Check whether the program parameter is the same as
            // the suit drawn and show message accordingly
            if (suitName.equals(args[0])) {
                System.out.println(
                    "Yes, my card is also a " + suitName);
            }
            else {
                System.out.println("Sorry, my card is a " + suitName);
            }
        }
    }
}
```

Figure 8.25 GuessACard.java

The `main()` method of the class `GuessACard` treats the first program parameter as the suit name chosen by the user; it randomly chooses a suit according to a random number. The `equals()` method of the `String` class is used to check the user's and the program's choices, and the appropriate message will be shown on the screen. (The `equals()` method of the `String` class is discussed in detail in *Unit 10*.)

For example, if you execute the `GuessACard` program and guess the suit drawn is a Club

```
java GuessACard Club
```

the program may show either one of the following messages:

```
Yes, my card is also a Club
```

or

```
Sorry, my card is a Spade
```

However, if you execute the `GuessACard` repeatedly, the following message will be shown on the screen:

```
Exception in thread "main" java.lang.NullPointerException
        at GuessACard.main(GuessACard.java:25)
```

The output shown on the screen discloses a `NullPointerException` exception occurred. Please take a few minutes to review the definition of the class `GuessACard`. Can you figure out the reason why this exception occurs?

The core reason is the possible values obtained from the expression

```
(int) (Math.random() * 4)
```

are `0`, `1`, `2` and `3` respectively. Therefore, if the number obtained from the expression is `0`, the value `0` is assigned to the variable `suitDrawn`. According to the `switch/case` statement, no statements in the `switch/case` construct are executed and the content of the variable `suitName` is kept `null`. Therefore, when the program executes the statement

```
suitName.equals(args[0])
```

the message `equals` is sent via a reference variable with value `null`. As the reference `null` refers to nothing, no object will receive the message and hence the `NullPointerException` is displayed.

The `IOException` and `FileNotFoundException` exceptions

The class definitions we discussed in this unit may encounter other runtime problems, which are modelled by `IOException` and `FileNotFoundException`. In a few words, an `IOException`

exception occurs when an input/output operation fails at runtime and a `FileNotFoundException` is a specific type of `IOException` that denotes a file cannot be found or cannot be accessed.

For example, execute the `TestBinaryFileViewer` or `TestTextFileViewer1` program with a file name that does not exist as the program parameter, such as:

`java TestBinaryFileViewer abc.txt`

If the file `abc.txt` does not exist in the current directory, the following message will be shown on the screen:

```
Exception in thread "main" java.io.FileNotFoundException: abc.txt
(The system cannot find the file specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:103)
        at BinaryFileViewer.show(BinaryFileViewer.java:16)
        at TestBinaryFileViewer.main(TestBinaryFileViewer.java:20)
```

The message indicates a runtime error was encountered and the exception type was `FileNotFoundException`. The message following the exception class name specifies that the program cannot find the file. Another scenario that can cause a `FileNotFoundException` exception is the supplied file cannot be read from or written to. For example, if the file name supplied via a program parameter is a directory, it is not possible to perform reading or writing operations on it. Furthermore, if the file on the disk is set to be non-readable or non-writable, performing reading or writing operations on it will cause the exception as well.

To illustrate, execute the `TestBinaryFileViewer` program with program parameter `C:` that is:

`java TestBinaryFileViewer C:`

This command intends to read the contents from `C:\`, which is actually a directory. Therefore, it will cause an exception and the following message is shown on the screen:

```
Exception in thread "main" java.io.FileNotFoundException: C: (Access
is denied)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:103)
        at BinaryFileViewer.show(BinaryFileViewer.java:16)
        at TestBinaryFileViewer.main(TestBinaryFileViewer.java:20)
```

The output shown suggests the exception is the same `FileNotFoundException`, and the message that follows the exception class name discloses the reason for causing the exception is that accessing the specified file (actually a directory) is denied.

The above `FileNotFoundException` may happen during the creation process of a stream object (such as `FileInputStream`/`FileOutputStream`, `FileReader`/`FileWriter`). After the stream is ready, runtime input/output errors may still occur, such as the file being read is corrupted or no free disk space is available while writing data to a file on that disk. A class `FileCreator1` is written in Figure 8.26 to illustrate the exception.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator1
public class FileCreator1 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(args[0]);

        // Determine the number of KB to be written to the file
        int size = Integer.parseInt(args[1]) * 1024;

        // Create a FileOutputStream for writing bytes to the file
        OutputStream out = new FileOutputStream(file);

        // Write the specifed number of bytes (with values 0)
        // in KB to the file
        for (int i = 0; i < size; i++) {
            out.write(0);
        }

        // Close the OutputStream object and release all related
        // resources
        out.close();
    }
}
```

Figure 8.26 FileCreator1.java

The corresponding driver program `TestFileCreator1` is shown in Figure 8.27.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestFileCreator1
public class TestFileCreator1 {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length < 2) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage: java TestFileCreator1 <file> <size in KB>");
        }
        else {
```

```
            // Create a FileCreator1 object
            FileCreator1 creator = new FileCreator1();

            // Create a file by specifying the name and size
            creator.create(args);
        }
    }
}
```

Figure 8.27 TestFileCreator1.java

The `TestFileCreator1` program expects two program parameters. The first program parameter is the name of the file to be created and written. The second one specifies the size in kilobytes of the file to be created and that the data are to be converted into type `int`. After the `FileCreator1` object is created, the reference of the array object that maintains the program parameters is supplied to the `FileCreator1` object while calling its `create()` method. For example, executing the `TestFileCreator1` program with program parameters `A:\test.dat` and `100` respectively gives:

```
java TestFileCreator1 A:\test.dat 100
```

A file named `test.dat` is created in the floppy diskette in drive `A` of the computer. The file size is 100 kilobytes.

However, if there is not sufficient free space on the disk to accommodate the file, the following message is shown:

```
Exception in thread "main" java.io.IOException: There is not enough
space on the disk
        at java.io.FileOutputStream.write(Native Method)
        at FileCreator1.create(FileCreator1.java:21)
        at TestFileCreator1.main(TestFileCreator1.java:20)
```

The exception type is `IOException`, and the message following the class name states that the reason for causing the exception was lack of space on the disk.

You can experiment with the `TestFileCreator1` by running it on your computer, but it is recommended you experiment with a file on a floppy disk and not create a huge file on your computer hard disk because, if your hard disk is running out of free space, the normal operations of your computer may be affected.

The importance of exception handling

Although many programming languages, especially the traditional ones, do not feature exception handling, the exception-handling feature has become a typical facility of modern programming languages.

A programmer implements specific exception handling in a program or class definition to specify the remedial operations to be executed if a particular runtime error occurs, so that the software can perform remedial actions and keep on executing. Otherwise, the software terminates immediately, which usually upsets the software users. What's your typical reaction to something like '*this program has performed an illegal operation and will be shut down immediately*'? We can't put in print what our reaction usually is! Therefore, the software should be built to handle the problems that could potentially occur. As a result, exception handling has become an indispensable feature of a modern programming language like the Java programming language.

Let's review the evolution of handling runtime errors in a program. Suppose there is a program segment like:

```
operation1();
operation2();
operation3();
......
```

If any one of the operations in the above program segment fails, the program may terminate immediately or may cause execution failures in the subsequent operations. Therefore, a serious programmer might consider a way to improve the reliability of the program segment. A possible way is to modify the methods, `operation1()`, `operation2()`, `operation3()`, and so on to return a `boolean` value to indicate the operation result. For example, the return values `true` and `false` indicate successful execution and failed execution respectively. Then, the program segment could be modified to be:

```
if (operation1()) {
    if (operation2()) {
        if (operation3()) {
            ......
        }
        else {
            // perform remedial action if operation3() fails
        }
    }
    else {
        // perform remedial action if operation2() fails
    }
}
else {
    // perform remedial action if operation1() fails
}
```

If the return value of any method is `false`, all subsequent method calls are skipped and the corresponding `else` part of the multi-level `if/else` statements will be executed to perform the corresponding remedial operation. The reliability of the program segment is greatly improved at the expense of the program segment being greatly complicated. Furthermore, the program statement could become unmanageable, and it becomes a nightmare to maintain a program segment written in this way.

Mechanisms for handling runtime errors evolved, and some programming languages started supporting a feature that executes a specified block of statements. If any statement fails, the subsequent statements are skipped and a supplementary block of statements is executed instead. That is:

```
{   // block of statements for actual operations
    operation1();
    operation2();
    operation3();
    ......
}
{   // block of statements to be executed if the execution of any
    // statement for actual operations fails
    ......
}
```

The above program construct provides a better mechanism of handling runtime errors, because it is not necessary to implement the program segment with multi-level `if/else` statements and the reliability of the program segment is guaranteed. Example programming languages that use such an approach are the Microsoft VBScript and Visual Basic version 6.0 and the prior versions.

Software developers soon found that they were still dissatisfied with the above program construct because different statements in the statement block for actual operations may cause different runtime errors. The program construct for handling runtime errors therefore evolved to be:

```
{   // block of statements for actual operations
    operation1();
    operation2();
    operation3();
    ......
}
{   // block of statements to be executed if the execution of any
    // statement for actual operations fails for reason1
    ......
}
{   // block of statements to be executed if the execution of any
    // statement for actual operations fails for reason2
    ......
}
```

The programming languages that handle runtime errors in this way include C++, Oracle PL/SQL, Microsoft Visual Basic.Net and C#. The above construct is also the blueprint of the way the Java programming language implements exception handling. We will discuss it soon. The mechanism for handling exceptions in the Java programming language is even more advanced. Runtime errors or exceptions in the Java programming language are objects of a well-organized class hierarchy. They can be classified as checked exceptions or unchecked exceptions, as mentioned in your last reading.

As your last reading also mentions, unchecked exceptions include runtime errors that are difficult for the program to handle, such as being out of memory. Such an unchecked exception is considered an `Error`.

Another type of unchecked exception represents the runtime errors that are caused by imperfect program design. Examples of such types are `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, `NumberFormatException` and `ClassCastException`, which are subclasses of the class `java.lang.RuntimeException` (or simply `RuntimeException`). In most cases, if your program causes such exceptions, you should review the program. You will usually find a program design problem, such as a class definition that cannot perform properly with input of extreme values.

The above two types of unchecked exception are due to the faults of resource exhaustion in the operating system or program design. Therefore, they cannot and should not be anticipated. The programmers are therefore not required to handle them.

However, checked exceptions are runtime errors or problems that the program should be able to handle and remedy, such as `FileNotFoundException` and `IOException`. They are subclasses of the class `Exception` other than `RuntimeException` or its subclasses. If a statement may cause a checked exception, the programmer must provide a way to handle it. The compiler can determine whether a statement may cause a checked exception(s). It must be enclosed in the above program construct and a corresponding remedial statement block must be provided, or a compile-time error will occur.

All exceptions are subclasses of the superclass `Exception`. Runtime errors that are difficult for the program to handle are subclasses of the superclass `Error`. Furthermore, the classes `Exception` and `Error` are subclasses of the class `Throwable`. The inheritance relationships among the above mentioned classes are visualized in Figure 8.28.

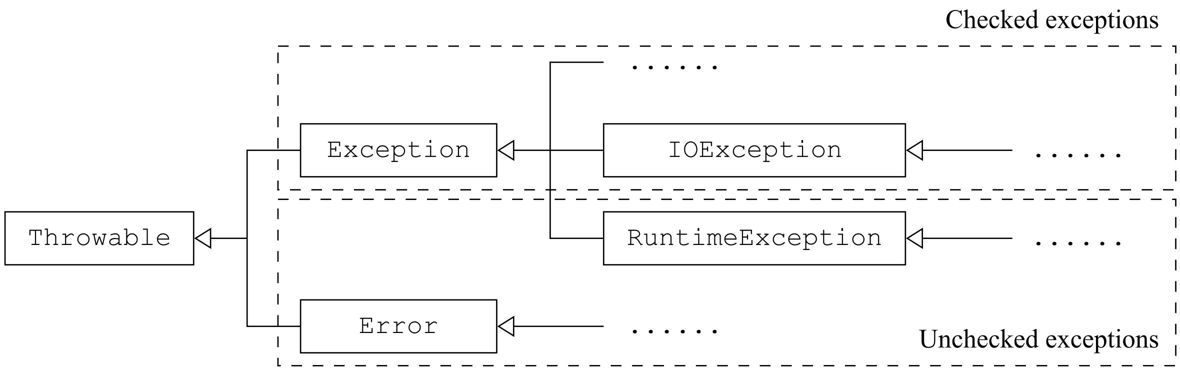


Figure 8.28 The classification of checked exceptions and unchecked exceptions

You are now aware that a program that causes an unchecked exception is mostly due to imperfect program design. Please use the following self-test to review the sample programs that caused unchecked exceptions.

Self-test 8.5

Review the sample programs that illustrate various unchecked exceptions (Figure 8.21–8.25). Suggest how to modify them so that those potential unchecked exceptions will not occur.

In a method definition, if a statement may cause checked exceptions, such as creating a `FileInputStream` object or calling the `read()` method of a `InputStream`, and no exception handling is implemented, the compiler software will give you a compile-time error. The reason is the compiler software cannot 'tolerate' your program that potentially causes checked exceptions for which no remedial measures have been implemented.

There are two possible ways to handle exceptions. They are discussed in the following two subsections.

Handling exceptions with `try/catch`

A program segment that can cause exceptions can be handled by using the keywords `try` and `catch`. A general format for using `try/catch` is:

```
try {
    // The program segment causes a potential exception
    ......
} catch (Exception-class-name variable) {
    // The program segment if an exception of type
    // specified by the catch clause occurs
    ......
}
```

The pair of curly braces that follows the keyword `try` encloses the statements that may cause exceptions. This is usually known as the `try` block. For the subsequent keyword `catch`, the parentheses enclose an exception class name and a variable, and the following pair of curly braces specifies the remedial action to be taken. This is usually known as the `catch` block. The *exception class name* specifies the type of exception this `catch` block can handle. In other words, if the type of an exception caused in the `try` block is an exception specified by the `catch` block, the `catch` block will be executed.

An exception is an object that maintains the information that causes the exception. It is created when software encounters an exceptional condition and the reference of the created exception object will be assigned to the *variable* that follows the exception class name. Finally, the following pair of curly braces encloses the statements for performing the remedial action for the exception type. The variable that follows the exception class name is local to the `catch` block. The statements in the `catch` block can therefore access the exception object with the variable.

If a statement causes an exception in the `try` block, all subsequent statements in the `try` block are skipped. If the type of caused exception object matches the exception type of a `catch` block, the statements in that `catch` block are executed and the exception is handled. After the execution of the `catch` block, the flow of control will continue after the last `catch` block of the entire `try/catch` construct.

For example, we can enhance the `FileCreator1` class so that it will show a proper message if any exception occurs. In an earlier section, we said that the statement that creates a `FileInputStream`/`FileOutputStream` object may cause a `FileNotFoundException` exception, and the `read()`/`write()` method may cause an `IOException` exception. Therefore, the `FileCreator1` is enhanced as `FileCreator2` in Figure 8.29.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator2
public class FileCreator2 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) {
        // Determine the number of KB to be written to the file
        int size = Integer.parseInt(args[1]) * 1024;

        // Create a File object that refers to the physical file
        File file = new File(args[0]);

        OutputStream out = null;
        try {
            // Create a FileOutputStream for writing bytes
            // to the file
            out = new FileOutputStream(file);
        } catch (FileNotFoundException fnfe) {
            System.out.println(
                "Failed in opening the file "
                    + file.getPath()
                    + " for writing.");
            System.out.println(
                "Please specify another file for writing.");
            System.out.println("Problem encountered: " +
                fnfe.getMessage());
        }

        // If the OutputStream object cannot be successfully
        // created, the variable out is null and the loop
        // for writing data to the file can be skipped
        if (out != null) {
            // Write the specifed number of bytes (with values 0)
            // in KB to the file
            try {
                for (int i = 0; i < size; i++) {
                    out.write(0);
                }
```

```
                // Close the OutputStream object and release all
                // related resources
                out.close();
            } catch (IOException ioe) {
                System.out.println(
                    "Failed in writing data to the file "
                        + file.getPath()
                        + ".");
                System.out.println("Problem encountered: " +
                    ioe.getMessage());
            }
        }
    }
}
```

Figure 8.29 FileCreator2.java

Like the `FileCreator1` class, it is necessary to write a driver program `TestFileCreator2` as shown in Figure 8.30 to test the `FileCreator2` class.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestFileCreator2
public class TestFileCreator2 {

    // Main executive method
    public static void main(String args[]) {
        // Check if a file is specified by program parameter
        if (args.length < 2) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage: java TestFileCreator2 <file> <size in KB>");
        }
        else {
            // Create a FileCreator2 object
            FileCreator2 creator = new FileCreator2();

            // Create a file by specifying the name and size
            creator.create(args);
        }
    }
}
```

Figure 8.30 TestFileCreator2.java

Compared with the definitions of the `create()` methods of the class `FileCreator1` and `FileCreator2` and the `main()` methods of the class `TestFileCreator1` and `TestFileCreator2`, the first difference is that the `create()` and `main()` methods of `FileCreator1` and `TestFileCreator1` need a `throws` clause, whereas those of the `FileCreator2` and `TestFileCreator2` do not. The reason is that all statements that can potentially cause checked exceptions are enclosed in `try` blocks. We elaborate the issue in detail in the next subsection. In the `catch` blocks of the `FileCreator2` `create()` method, the `getMessage()` method of the exception object is called to get the underlying reason for causing the exception.

In the subsection 'What is an exception?', you learned that the statement

```
OutputStream out = new FileOutputStream(file);
```

may cause a `FileNotFoundException` exception. Therefore, if the program intends to handle the exception explicitly, the statement must be enclosed in a `try` block with a `catch` block for `FileNotFoundException`, such as:

```
try {
    OutputStream out = new FileOutputStream(file);
} catch (FileNotFoundException fnfe) {
    ......
}
```

You should notice that if the variable `out` is declared in the `try` block, it is local to the `try` block and can only be accessed in the `try` block. In order to enable the variable `out` to be accessible by statements after the `try/catch` construct, the variable must be declared before the `try` block, such as:

```
OutputStream out;
try {
 out = new FileOutputStream(file);
} catch (FileNotFoundException fnfe) {
    ......
}
```

This is not the end of the story. The variable `out` is involved in the condition of the `if` statement.

```
if (out != null) {
    ......
}
```

If an exception occurs while creating the `FileOutputStream` object in the `try` block, the assignment operation is not executed and the content of the variable `out` has not been initialized. Then, the condition of the `if` statement causes a compile-time error because the condition of the `if` statement tries to compare a value with a variable that is not initialized. Therefore, it is necessary to declare the variable `out` with initialization, and it is preferable to initialize it to be `null`. As a result, the program segment is further modified to be:

```
OutputStream out = null;
try {
    out = new FileOutputStream(file);
} catch (FileNotFoundException fnfe) {
    ......
}
```

Compile the classes `FileCreator2` and `TestFileCreator2` and execute the `TestFileCreator2` with program parameters `C:\` and `100`, that is:

```
java TestFileCreator2 C:\ 100
```

The statement:

```
new FileOutputStream(file)
```

in the `main()` method will cause a `FileNotFoundException` exception that matches the exception class name of the `catch` block, and the statements in the `catch` block are executed. The following output is shown on the screen as the result:

```
Failed in opening the file C:\ for writing.
Please specify another file for writing.
Problem encountered: C:\ (The system cannot find the path specified)
```

If the file specified as the program parameter is not writable, such as the file `C:\IO.SYS`

```
java TestFileCreator2 C:\IO.SYS 100
```

the following output is shown on the screen:

```
Failed in opening the file C:\IO.SYS for writing.
Please specify another file for writing.
Problem encountered: C:\IO.SYS (Access is denied)
```

Even though the program cannot continue its usual operations, it provides non-technical information or instructions to the user, so that he or she is notified of what has happened and the instructions to follow, if any.

If the `FileOutputStream` object is successfully created, its reference is assigned to the variable `out`. The `catch` block for `FileNotFoundException` is skipped and the program continues. Then, the `if` part of the `if` statement will be executed:

```
if (out != null) {
    ......
}
```

In the `if` part of the `if` statement, the number of bytes to be written to the file is obtained, and a `for` loop is used to write the bytes (with value `0`) to the file associated through the `FileOutputStream` object. You learned earlier that the `write()` method of the `FileOutputStream` might cause an `IOException` exception if the method fails to write a byte to the file, such as being out of free disk space. Therefore, the statement

```
out.write(0);
```

has to be enclosed in a `try` block with a `catch` block dedicated for `IOException`. That is:

```
......
try {
    ......
    out.write(0);
    ......
} catch (IOException ioe) {
    ......
}
......
```

Here, we have two options to define the above program segment. They are shown in Table 8.3.

Table 8.3 The two approaches to handle the `IOException` exception caused by the `write()` method

<table>
<tr><th>Approach A</th><th>Approach B</th></tr>
<tr><td><pre>......
try {
    for (i = 0; i < size; i++) {
        out.write(0);
    }
} catch (IOException ioe) {
    ......
}
......</pre></td><td><pre>......
for (i = 0; i < size; i++) {
    try {
        out.write(0);
    } catch (IOException ioe) {
        ......
    }
}
......</pre></td></tr>
</table>

In Approach A presented in Table 8.3, the entire `for` loop is enclosed in the `try` block. If the `write()` method causes an exception, all statements in the `try` block are skipped and the `catch` block for `IOException` is executed, which implies the `for` loop is terminated. After the statements in the `catch` block are executed, the flow of control continues at the statement that follows the `catch` block. Therefore, the `catch` block is executed once.

In Approach B, the `for` loop encloses the entire `try/catch` construct. If the `write()` method causes an `IOException`, the statements in the `catch` block are executed and the exception is handled. Afterwards, the flow of control continues after the `try/catch` construct and before the end of the loop body of the `for` loop. The `for` loop has not terminated yet and it will continue to execute. If the reason that caused the `IOException` was no free disk space, the subsequent calls to the `write()` method of the `OutputStream` object will probably fail as well and `IOException` exceptions will occur repeatedly. As a result, the `catch` block will be executed again and again.

In the `FileCreator2` class definition, once a call to the `write()` method fails, it is only necessary to execute the `catch` block once to notify the user and subsequent calls to the `write()` method are unnecessary. Therefore, Approach A is implemented in the `FileCreator2` class definition.

Now, execute the `FileCreator2` with program parameters that specify a file on a disk and the size (in kilobytes) of the file to be created, such as:

```
java TestFileCreator2 A:\test.dat 2048
```

A file named `test.dat` is created with size 2048KB on drive `A` of the computer. If there is not sufficient disk-space on drive `A`, the following message will be shown on the screen:

```
Failed in writing data to the file A:\test.dat.
Problem encountered: There is not enough space on the disk
```

Hierarchy of exception classes and the matching of `catch` blocks

We said that exceptions in the Java programming language are implemented as objects and hence there are class definitions for the exceptions. Furthermore, some exceptions are a specific type of a general type exception. Exception `FileNotFoundException` is a specific type of `IOException` exception. Therefore, it makes sense for the class `FileNotFoundException` to be defined as a subclass of the superclass `IOException`. Furthermore, `IOException` is a particular type of `Exception`, and `IOException` is therefore defined as a subclass of the superclass `Exception`.

The Java software library defines many exception classes. They are categorized and the relationships among them are identified. As a result, the exceptions in the Java programming language are defined as a family tree in the Java software library as shown in Figure 8.31.

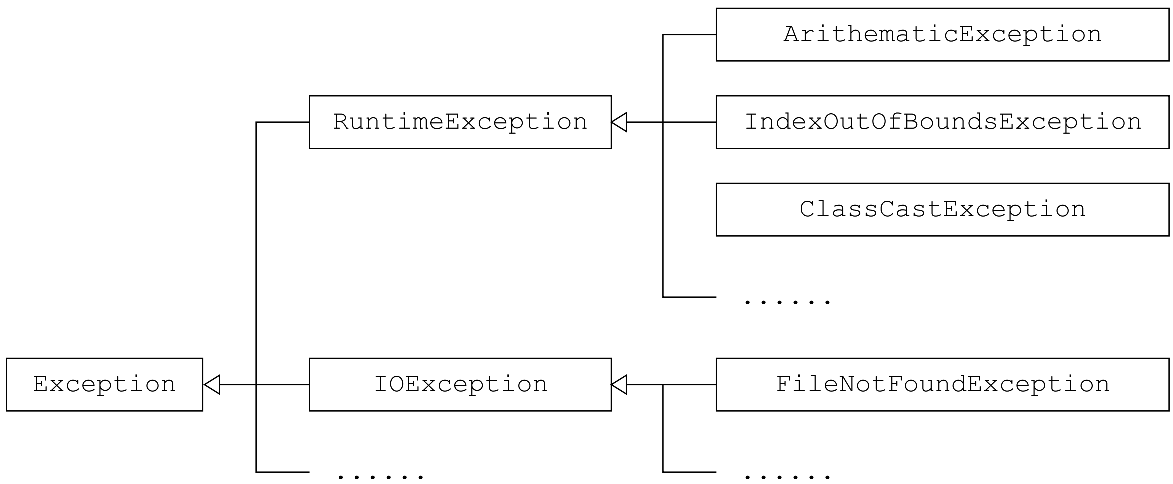


Figure 8.31 The inheritance relationships among the exception classes

Due to the 'is a' (inheritance) among the exception classes, a `FileNotFoundException` exception can be considered an `IOException` exception. Therefore, a `FileNotFoundException` exception matches the `catch` block for an `IOException` exception as well. With respect to the `create()` method of the `FileCreator2` class, the `FileNotFoundException` exception that may be caused by the statement

```
new FileOutputStream(file)
```

can be handled by the `catch` block for `IOException` as well. As a result, most statements in the `create()` method can be enclosed in a single `try` block as in the definition of `FileCreator3` class shown in Figure 8.32.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator3
public class FileCreator3 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) {
        // Determine the number of KB to be written to the file
        int size = Integer.parseInt(args[1]) * 1024;

        // Variable file has to be defined here so that it can be
        // accessed in the catch block
        File file = null;
        try {
            // Create a File object that refers to the physical file
            file = new File(args[0]);

            // Create a FileOutputStream for writing bytes
            // to the file
            OutputStream out = new FileOutputStream(file);

            // Write the specifed number of bytes (with values 0)
            // in KB to the file
            for (int i = 0; i < size; i++) {
                out.write(0);
            }

            // Close the OutputStream object and release all related
            // resources
            out.close();
        } catch (IOException ioe) {
            System.out.println(
                "Failed in an input/output operation with file "
                    + file.getPath()
                    + ".");
            System.out.println("Problem encountered: " +
                ioe.getMessage());
        }
    }
}
```

Figure 8.32 FileCreator3.java

(The driver programs for the class `FileCreator3` and the subsequent versions in this section are similar to `TestFileCreator2`. They can be found on the course CD-ROM and course website.)

Compared with the `FileCreator2` class definition, the `create()` method of the `FileCreator3` class does not need the `if` statement to verify whether the `FileOutputStream` object is created successfully because, if the `FileOutputStream` object cannot be created, a `FileNotFoundException` exception will occur, all subsequent statements in the `try` block are skipped and the `catch` block for `IOException` (as a `FileNotFoundException` exception is an `IOException` exception) will be executed.

The way to handle exceptions used by the `FileCreator3` class is simpler, but the same `catch` block is executed if either a `FileNotFoundException` exception or an `IOException` exception occurs. If specific remedial operations are required for a `FileNotFoundException` exception, it is possible to use the `instanceof` operator in the `catch` block for `IOException` to determine whether the exception is actually a `FileNotFoundException` exception. This is used in the `FileCreator4` class definition as shown in Figure 8.33.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator4
public class FileCreator4 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) {
        // Determine the number of KB to be written to the file
        int size = Integer.parseInt(args[1]) * 1024;

        // Variable file has to be defined here so that it can be
        // accessed in the catch block
        File file = null;
        try {
            // Create a File object that refers to the physical file
            file = new File(args[0]);

            // Create a FileOutputStream for writing bytes
            // to the file
            OutputStream out = new FileOutputStream(file);

            // Write the specifed number of bytes (with values 0)
            // in KB to the file
            for (int i = 0; i < size; i++) {
                out.write(0);
            }
```

```
            // Close the OutputStream object and release all related
            // resources
            out.close();
        } catch (IOException ioe) {
            if (ioe instanceof FileNotFoundException) {
                System.out.println(
                    "Failed in opening the file "
                        + file.getPath()
                        + " for writing.");
                System.out.println(
                    "Please specify another file for writing.");
            } else {
                System.out.println(
                    "Failed in writing data to the file "
                        + file.getPath()
                        + ".");
            }
            System.out.println("Problem encountered: " +
                ioe.getMessage());
        }
    }
}
```

Figure 8.33 FileCreator4.java

In the `catch` block for `IOException` exception in the `create()` method of `FileCreator4` class definition, the condition of the `if` statement uses an `instanceof` operator to determine whether the exception object is a `FileNotFoundException` exception object and shows the corresponding messages on the screen. Otherwise, the `else` part of the `if` statement is executed to show a general message for an `IOException` exception.

The Java programming language provides an even more elegant way to handle multiple exception types in that one exception (`FileNotFoundException`) is a specific type of another general exception (`IOException`), as in the `FileCreator5` class definition shown in Figure 8.34.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator5
public class FileCreator5 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) {
        // Determine the number of KB to be written to the file
        int size = Integer.parseInt(args[1]) * 1024;
```

```
        // Variable file has to be defined here so that it can be
        // accessed in the catch block
        File file = null;
        try {
            // Create a File object that refers to the physical file
            file = new File(args[0]);

            // Create a FileOutputStream for writing bytes
            // to the file
            OutputStream out = new FileOutputStream(file);

            // Write the specifed number of bytes (with values 0)
            // in KB to the file
            for (int i = 0; i < size; i++) {
                out.write(0);
            }

            // Close the OutputStream object and release all related
            // resources
            out.close();
        } catch (FileNotFoundException fnfe) {
            System.out.println(
                "Failed in opening the file "
                    + file.getPath()
                    + " for writing.");
            System.out.println(
                "Please specify another file for writing.");
            System.out.println("Problem encountered: " +
                fnfe.getMessage());
        } catch (IOException ioe) {
            System.out.println(
                "Failed in writing data to the file " +
                file.getPath() + ".");
            System.out.println("Problem encountered: " +
                ioe.getMessage());
        }
    }
}
```

Figure 8.34 FileCreator5.java

The definitions of classes `FileCreator4` and `FileCreator5` are practically equivalent. However, the definition of the `FileCreator5` class is preferable because there is a `catch` block dedicated to the `FileNotFoundException` exception, and further modification and maintenance is easier. If a `FileNotFoundException` exception occurs, the `catch` block for `FileNotFoundException` is executed and all subsequent `catch` blocks will not be executed, even though a `FileNotFoundException` exception can be considered an `IOException` exception. That is, only the first `catch` block that matches the exception that occurred will be executed.

Another reason why `FileCreator5` class definition is preferable is that if the `FileCreator4` class needs to use the specific functionality of the `FileNotFoundException` object, it is necessary to perform a casting operation on the reference stored in the variable `ioe`, whereas the type of variable `fnfe` in the `FileCreator5` is `FileNotFoundException` requires no casting operation to access the specific members defined in the `FileNotFoundException` class.

You should notice that the `catch` block for the `FileNotFoundException` exception must precede that for the `IOException` exception. Otherwise, the `try/catch` construct would be written as:

```
try {
    ......
} catch (IOException ioe) {
    ......
} catch (FileNotFoundException fnfe) {
    ......
}
```

If a `FileNotFoundException` occurred in the `try` block, the `catch` block for `IOException` would have been executed and the `catch` block for `FileNotFoundException` would never be executed. Therefore, the compiler software will give you a compile-time error.

You learned that the method:

```
Integer.parseInt(args[1])
```

may give a `NumberFormatException` if the `String` object supplied to the method cannot be converted into a value of type `int`, such as executing the `FileCreator5` with the second parameter as `1.5` that intends to create a file of size 1.5 kilobytes:

```
java TestFileCreator5 test.dat 1.5
```

The program shows the screen output in Figure 8.35:

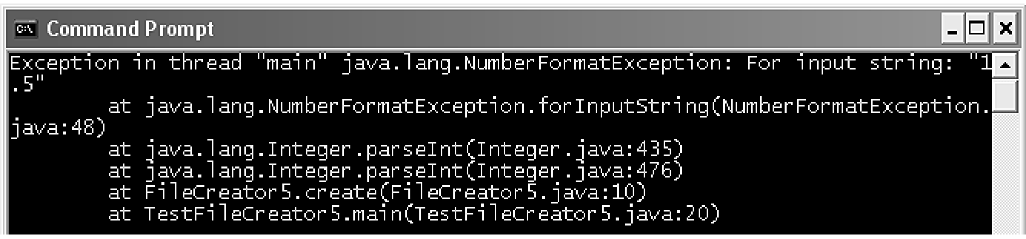


Figure 8.35 The output shown by providing an invalid second program parameter to `TestFileCreator5`

In order to show a proper message to the user if the user supplies an invalid program parameter for the file size, it is preferable to add a `catch` block for the `NumberFormatException` exception as in the definition of class `FileCreator6` shown in Figure 8.36.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator6
public class FileCreator6 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) {
        // Variable file has to be defined here so that it can be
        // accessed in the catch block
        File file = null;
        try {
            // Determine the number of KB to be written to the file
            int size = Integer.parseInt(args[1]) * 1024;

            // Create a File object that refers to the physical file
            file = new File(args[0]);

            // Create a FileOutputStream for writing bytes
            // to the file
            OutputStream out = new FileOutputStream(file);

            // Write the specifed number of bytes (with values 0)
            // in KB to the file
            for (int i = 0; i < size; i++) {
                out.write(0);
            }

            // Close the OutputStream object and release all related
            // resources
            out.close();
        } catch (NumberFormatException nfe) {
            System.out.println(
                "The second program parameter is not an integer.");
            System.out.println(
                "Please provide a valid integer as the file size.");
        } catch (FileNotFoundException fnfe) {
            System.out.println(
                "Failed in opening the file "
                    + file.getPath()
                    + " for writing.");
            System.out.println(
                "Please specify another file for writing.");
            System.out.println("Problem encountered: " +
                fnfe.getMessage());
        } catch (IOException ioe) {
            System.out.println(
                "Failed in writing data to the file " +
                file.getPath() + ".");
            System.out.println("Problem encountered: " +
                ioe.getMessage());
        }
    }
}
```

Figure 8.36 FileCreator6.java

Compile the classes `FileCreator6` and `TestFileCreator6`, and execute the `TestFileCreator6` class definition with `1.5` as the second program parameter, such as:

```
java TestFileCreator6 test.dat 1.5
```

The following output will be shown on the screen.

```
The second program parameter is not an integer.
Please provide a valid integer as the file size.
```

In the definition of class `FileCreator6`, the `catch` block for the `NumberFormatException` exception can be placed arbitrarily among the three because there is no 'is a' relationship between the `NumberFormatException` exception and the `FileNotFoundException` exception, or between the `NumberFormatException` and the `IOException` individually.

Self-test 8.6

Modify the definition of the class `TestFileCreator6` (you can find it on the course CD or course website) so that no validation on the number of program parameters is performed and the reference of the array object that maintains program parameters is supplied to the `create()` method of the `FileCreator6` object. Then, if there are not enough program parameters, using an array element (`args[0]` or `args[1]`) will cause an `ArrayIndexOutOfBoundsException` exception. To handle the exception, add a `catch` block for `ArrayIndexOutOfBoundsException` exception in the `create()` method of the `FileCreator6` class.

Discuss the original and modified definitions of the `FileCreator6`.

Declaring methods that throw exceptions

In the previous subsection, all input/output operations are performed in the `create()` method, and we used `try/catch` constructs to handle the exceptions that can occur during software execution. Therefore, because all statements that may cause exceptions are enclosed in `try/catch` constructs, the `create()` methods of classes `FileCreator2` through `FileCreator6` do not need a `throws` clause. In this subsection, we discuss this issue in greater detail.

A sample flow of control sequence is shown in Figure 8.37.

```
... main(String args[])
{
  ......
  method1();
  ......
}
```

```
void method1() {
  ......
  method2();
  ......
}
```

```
void method2() {
  ......
  // Actual I/O operations
  ......
}
```

Figure 8.37 A sample sequence of method calls

The methods `method1()` and `method2()` can be any class methods or the methods of objects in the JVM. As previously mentioned, input/output operations may cause `IOException` exceptions; it is therefore necessary to use `try/catch` construct in the method `method2()` to handle the exception caused or the compiler software will give compile-time errors. As a result, the method `method2()` should be modified to be as shown in Figure 8.38.

```
... main(String args[])
{
  ......
  method1();
  ......
}
```

```
void method1() {
  ......
  method2();
  ......
}
```

```
void method2() {
  ......
  try {
    ......
    // I/O operations
    ......
  } catch (IOException e) {
  ......
  }
}
```

Figure 8.38 A `try/catch` construct is implemented in the `method2()` method

Suppose that the method `method1()` calls another method `method3()` that also performs some input/output operations. The method `method3()` has to enclose the statements that may cause exceptions in a `try` block with a suitable `catch` block as well. That is:

```
... main(String args[])
{
  ......
  method1();
  ......
}
```

```
void method1() {
  ......
  method2();
  ......
  method3();
  ......
}
```

```
void method2() {
  ......
  try {
    ......
    // I/O operations
    ......
  } catch (IOException e) {
  ......
  }
}
```

```
void method3() {
  ......
  try {
    ......
    // I/O operations
    ......
  } catch (IOException e) {
  ......
  }
}
```

Figure 8.39 Both `method2()` and `method3()` methods need a `try/catch` construct for exception handling

The above flow of control pattern works, but there is still some room for improvement.

1 First, the `catch` blocks of the methods `method2()` and `method3()` may perform similar remedial operations and the statements in the `catch` blocks may be the same. If possible, such duplicated statements should be avoided.

2 In some cases, an input/output operation failure in method `method2()` may imply it is no longer necessary to call the method `method3()` by the method `method1()`.

The Java programming language allows a method that executes statements that may cause exceptions not to handle the caused exception by itself, which means that it is not necessary to enclose those statements in a `try` block with suitable `catch` blocks. Then, if a statement in the method causes an exception, all subsequent statements in the method are skipped and the flow of control will be returned to the caller method with the caused exception. To induce the compiler software to accept a method that does not handle the caused exception by itself and the caused exception is returned to the caller method, the method must be defined with a `throws` clause to indicate the method may cause such an exception.

Therefore, the general format of a method definition is:

`[public | private | protected]` *return-type method-name(parameter-list)* `throws` *exception-list* `{`
    `// statements`
`}`

The `throws` clause declares all possible checked exceptions that may cause exceptions while the method is executing. If the method can throw more than one checked exception, the class names of the exceptions are listed in the `throws` clause and are separated by commas. It indicates that if a statement in the method causes a checked exception that is one of the exceptions in the `throws` clause, the method will not handle it, and the subsequent statements in the method are skipped and the flow of control will immediately be returned to the caller method with the caused exception. The scenario of the method calls is shown in Figure 8.40.

```
... main(String args[])
{
  ......
  method1();
  ......
}
```

```
void method1() {
  ......
  method2();
  ......
  method3();
  ......
}
```

```
void method2()
  throws IOException {
  ......
  // I/O operations
  ......
}
```

```
void method3()
  throws IOException {
  ......
  // I/O operations
  ......
}
```

Figure 8.40 Both `method2()` and `method3()` methods are defined with a `throws` clause for `IOException` and any caused `IOException` will not be handled in those methods

Then, the compiler software determines that method `method1()` calls methods (`method2()` and `method3()`) that may cause `IOException` exceptions. Therefore, method `method1()` should use a `try/catch` block to enclose the method calls `method2()` and `method3()` as shown in Figure 8.41.

```
... main(String args[])
{
  ......
  method1();
  ......
}
```

```
void method1() {
  ......
  try {
    ......
    method2();
    ......
    method3();
    ......
  }
  catch (IOException e)
  {
    ......
  }
  ......
}
```

```
void method2()
  throws IOException {
  ......
  // I/O operations
  ......
}
```

```
void method3()
  throws IOException {
  ......
  // I/O operations
  ......
}
```

Figure 8.41 Method `method1()`needs a `try/catch` construct as it calls other methods that may throw an `IOException`

If any statement in the `try` block of method `method1()` and any statement in methods `method2()` and `method3()` causes an `IOException` exception, all subsequent statements are skipped and the statements in the `catch` block in method `method1()` are executed and the exception is considered handled. After executing the `catch` block in method `method1()`, the flow of control will continue after the `catch` block (or `catch` blocks if there is more than one `catch` block in method `method1()`).

Furthermore, it is even possible for method `method1()` not to handle the caused exception, and it is then necessary to add a `throws` clause to method `method1()` as shown in Figure 8.42.

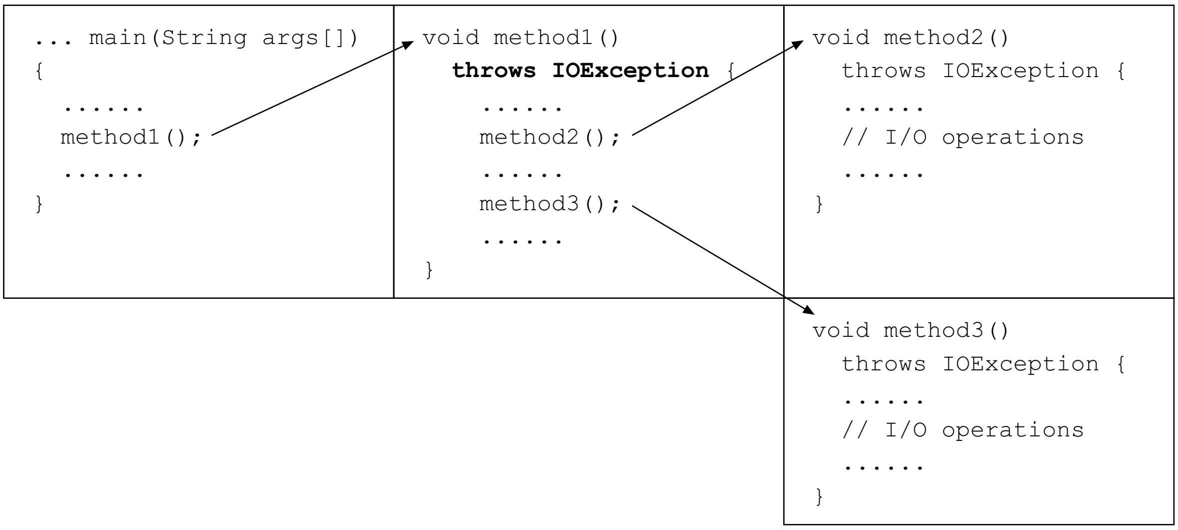


Figure 8.42 Method `method1()` declares a `throws` clause to signify that it may cause an `IOException` and that it will not handle it by itself

The compiler software then determines that the `main()` method calls method `method1()`which may cause an `IOException` exception. Then, the developer is required to use a `try/catch` construct to enclose the method call to method `method1()`, as shown in Figure 8.43.

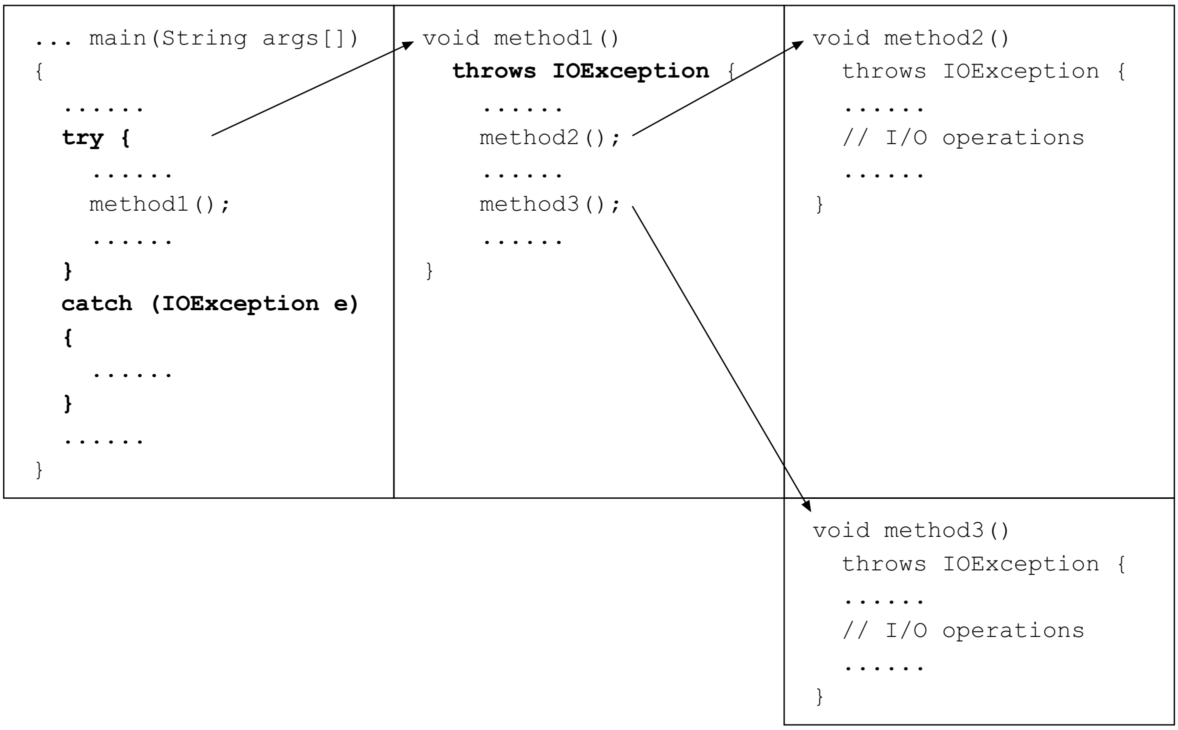


Figure 8.43 A suitable `try/catch` construct is mandatory in the `main()` method

In this instance, if any statement in the methods `method1()`, `method2()` and `method3()` and the `try` block in the `main()` method causes an `IOException` exception, the `catch` block of the `main()` method will be executed.

Another approach is that the `main()` method can be defined with a `throws` clause to declare that it may cause a checked exception, as shown in Figure 8.44.

```
... main(String args[])
  throws IOException {
  ......
  method1();
  ......
}
```

```
void method1()
  throws IOException {
    ......
    method2();
    ......
    method3();
    ......
}
```

```
void method2()
  throws IOException {
  ......
  // I/O operations
  ......
}
```

```
void method3()
  throws IOException {
  ......
  // I/O operations
  ......
}
```

Figure 8.44 All methods are declared with a `throws` clause for an `IOException`

If the method `main()` declares it may cause the checked exception, the compiler software will accept the class definition, as it is the first method to be executed. If an exception occurs anywhere during the execution, there is no `try/catch` block to handle the caused exception and all subsequent statements of the statement that causes the exception are skipped. The flow of control is returned to the caller method. Ultimately, the `main()` method will terminate immediately. Furthermore, the JVM will show the exception details in it own way.

In Figure 8.44, all methods are defined with a `throws` clause, which implies that any exception caused will terminate the software execution immediately and no remedial operation is carried out. The class definitions are simpler, but it is not good programming practice to develop software in the Java programming language in this way.

To summarize, any method that may cause a checked exception can either use a `try/catch` construct to handle the caused exception itself or declare a `throws` clause so that the caused exception is handled by the caller method. Sometimes, it is preferable to handle the exception by performing remedial operations as soon as possible, so the method should use a `try/catch` construct to handle the exception in the method. It is sometime preferable, however, to stop further operations in the method if an exception is caused and notify the caller method immediately. In this instance, the method that may cause an exception should declare the `throws` clause so that the exception is handled by the caller method.

Methods of classes that perform input/output operations, such as `TestBinaryFileViewer`, `TestBinaryFileCreator`, `TestTextFileViewer1` and `TestTextFileCreator`, were defined with a `throws` clause because the methods are simpler for illustration and discussion. However, for serious software applications, a proper exception handling mechanism, such as `try`/`catch` blocks, should be implemented appropriately in the class definitions, so that the exceptions caused are properly handled.

Self-test 8.7

Two lazy Java programmers are writing classes in the Java programming language to perform input/output operations. In order to 'comfort' the Java compiler software that statements that may cause checked exceptions must be enclosed in a `try/catch` construct or the method containing the statements must declare a `throws` clause, they use two different approaches:

1 Programmer A declares all methods that perform input/output operations with a `throws` clause `throws Exception`, for example:

```
public void performIO() throws Exception {
    ......
}
```

Subsequently, all methods that call the methods performing input/output operations, such as the above `performIO()` method, are declared with the same `throws` clause. Such an approach is used for all related methods including the main method, `main()`.

2 Programmer B adds a `try/catch` block to each method that performs input/output operations, but there are no statements in the `catch` block, that is:

```
public void performIO() {
    try {
        // IO operations
        ......
    } catch (Exception e) {}
}
```

Discuss the following questions:

1 Why can the two approaches above 'comfort' the Java compiler software and no compilation errors will occur?

2 What are the problems with the approaches used by the two programmers?

Filter streams

Earlier in the unit, we said that there were basically two stream categories — byte streams and character streams. According the directions of the data flow in the streams, they were further classified into input stream and output stream, as shown in Table 8.4.

Table 8.4 The different categories of stream classes

	Byte stream	**Character stream**
Input stream	`InputStream` (e.g., `FileInputStream`)	`Reader` (e.g., `FileReader`)
Output stream	`OutputStream` (e.g., `FileOutputStream`)	`Writer` (e.g., `FileWriter`)

The classes shown in parentheses in Table 8.4 are concrete subclasses of the general classes `InputStream`, `Reader`, `OutputStream` and `Writer` and are dedicated to file input/output operations.

Besides concrete classes for performing the input/output operations with a particular data source or data destination, the Java standard software library provides classes that act as an intermediary providing extra functionalities. It is like a filter lens for a camera, as shown in Figure 8.45.

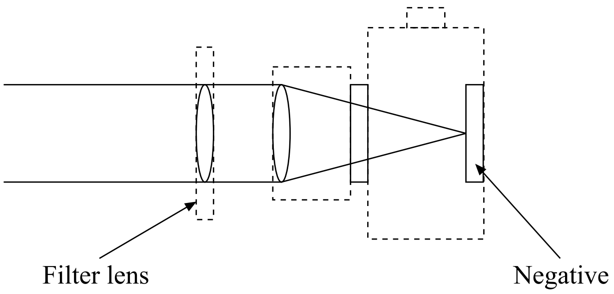


Figure 8.45 A camera with an add-on filter lens for a special effect

In Figure 8.45, the negative plays the role of a data destination. The built-in lens focuses the incoming light onto the surface of the negative, which is like an output data stream sending the supplied data to the data destination for storage. If a special effect is required, it is possible to add an add-on filter lens on top of the camera's built-in lens. The stream based input/output operations used by the Java programming language can use a similar mechanism for extra functionalities.

The Java standard software library provides some classes that do not associate with particular storage media or communication channels but with the general input stream and output stream. For an input filter stream object, it reads data from its associated input stream to be returned with extra functionality. Similarly, the output filter stream object manipulates the data supplied to it for extra functionality, and the resultant data are sent to its associated output stream object. Figure 8.46 can help you to visualize the relationship between filter streams and general input/output streams.

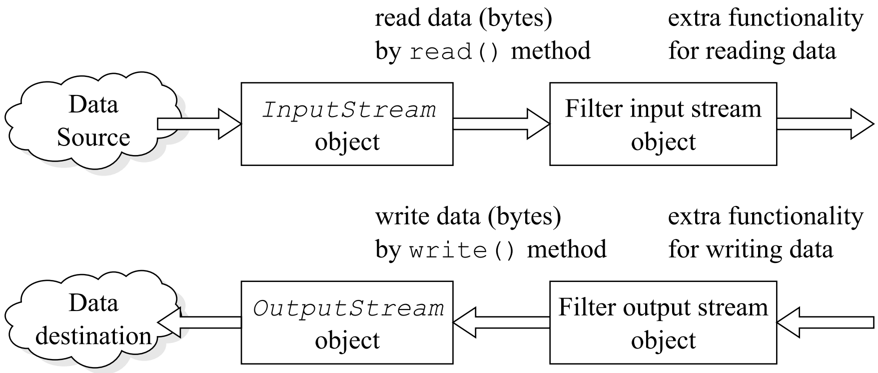


Figure 8.46 The functionalities of filter streams

The Java software library provides many filter classes. The commonly used filter classes are for data conversion and improving execution performance. In the next subsection, we discuss possible filter streams for data conversions. To improve input/output operating performance, a possible way is to use the computer's memory as a temporary data source and data destination, because accessing computer memory is much faster than accessing secondary memory such as a file. Such an approach is known as *buffering* and it is discussed in detail in Appendix D.

Filter streams for conversion

A common unit of data flow in a computer is an 8-bit `byte`. Therefore, the basic units of data for most storage media and communication channels are of type `byte` as well. However, a software application may require a collection of bytes to represent some more structured data, such as it takes 4 bytes and 8 bytes in a storage device to represent a 4-byte integer value of type `int` and an 8-byte datum of type `double`. With an `InputStream` object, it is possible to read 4 bytes and 8 bytes in the program and convert them programmatically into data of `int` and `double` respectively. It is a common but tedious task. Therefore, the Java standard software library provides classes for conversions between bytes and the structured data based on the idea of filter streams.

The necessity of data conversions

Among all data conversions, the conversion between 8-bit bytes and 16-bit characters is the most common.

You should note the JVM handles characters in Unicode, which is a 16-bit coding system. That is, each character needs 16 bits to represent it. For a storage medium such as a file on a disk, a character in a text file takes 8 bits or 16 bits to represent it depending on the code of the character and the default coding system of the platform. Furthermore, the codes for the same character in the default coding system of the platform (mostly an 8-bit coding system) and Unicode may differ. Therefore, a translation between 8-bit byte and 16-bit Unicode character is required in reading and writing character data to and from a data source/destination, as shown in Figure 8.47.

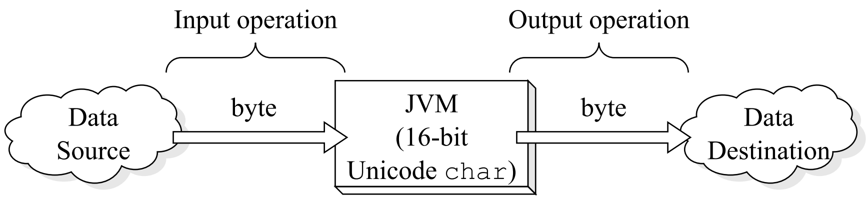


Figure 8.47 The input/output operations with respect to the JVM

Whenever a character datum flows across the boundary of the JVM, it is usually necessary to perform a translation between byte and character. For example, if the contents of a file are

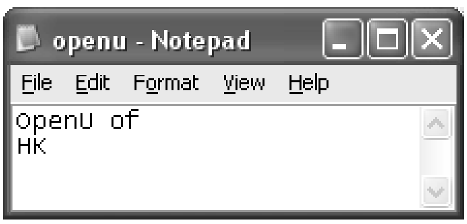
O	P	E	N		U	N	I	V	E	R	S	I	T	Y		......

it is expected that when the data are read from the file, they are converted into a sequence of characters such as:

```
'O', 'P', 'E', 'N', ' ', 'U', 'N', 'I', 'V', 'E', 'R','S', 'I', 'T', 'Y', ...
```

You should note that every single byte in the file is converted into a two-byte Unicode character. Going the other way, you would expect that when a sequence of characters is written to a byte-oriented destination stream, the data would be converted into bytes and stored properly.

To illustrate the necessity of data conversion, let's create a text file with an editor software such as Notepad. The `TestBinaryFileViewer` program will read the file. Type something in the file, such as:



Save the file with a name, say `openu.txt`. Then, we can use the `TestBinaryFileViewer` class to show its contents. The output shown on the screen is:

```
Data read from file (openu.txt)
Byte (#1) = 79 <O>
Byte (#2) = 112 <p>
Byte (#3) = 101 <e>
Byte (#4) = 110 <n>
Byte (#5) = 85 <U>
Byte (#6) = 32 < >
Byte (#7) = 111 <o>
```

```
   Byte (#8) = 102 <f>
   Byte (#9) = 13
   Byte (#10) = 10
   Byte (#11) = 72 <H>
   Byte (#12) = 75 <K>
```

The output shown on the screen discloses there are 12 bytes in the files, and the values of the bytes with the corresponding characters (if the bytes are treated as characters) in the file are shown. Each byte shown represents a symbol according to the American Standard Code for Information Interchange (ASCII).

The bytes with values `13` and `10` correspond to the new-line character (`\n`) and carriage return character (`\r`) respectively. A combination of these two bytes is the default byte pattern for Windows platforms to indicate the end of line and that the following bytes are to be in the next line. The byte with value `32` represents a blank space.

We can read the contents of the `openu.txt` text file with the `TestTextFileViewer1` program that we discussed earlier in this unit. By executing the program with `openu.txt` as the program parameter, it shows the following output.

```
Data read from file (f:openu.txt)
   Char (#1) = 79 ('O')
   Char (#2) = 112 ('p')
   Char (#3) = 101 ('e')
   Char (#4) = 110 ('n')
   Char (#5) = 85 ('U')
   Char (#6) = 32 (' ')
   Char (#7) = 111 ('o')
   Char (#8) = 102 ('f')
   Char (#9) = 13
   Char (#10) = 10
   Char (#11) = 72 ('H')
   Char (#12) = 75 ('K')
```

From the output of the execution, you can see that each byte in the file `openu.txt` is translated into a corresponding Unicode character and returned by the `read()` method of the `Reader` object. As all characters in the file are letters that are defined in ASCII, the conversion simply extends an 8-bit `byte` to a 16-bit `int` and most significant 8 bits filled with zeroes.

To reverse the process, a class named `TextFileWriter` is written as shown in Figure 8.48. Its `create()` method accepts two parameters and writes the second program parameter to the file specified by the first program parameter.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TextFileWriter
public class TextFileWriter {

    // Create a file with the specified message as content
    public void create(String name, String message) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);

        // Create a FileWriter object specified by the File
        // object and is treated as a general Writer object
        Writer writer = new FileWriter(file);

        // Send the second program parameter to the destination file
        writer.write(message);

        // Call the Writer close() method to release all
        // corresponding resources
        writer.close();

    }
}
```

Figure 8.48 TextFileWriter.java

To use a `TextFileWriter` object to create a text file with a string as its contents, a driver program `ArgsToFile` is written in Figure 8.49.

```
// Resolve classes in the java.io package
import java.io.*;

// Definition of class ArgsToFile
public class ArgsToFile {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length < 2) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage : java ArgsToFile <file> <message>");
        }
        else {
            // Create a TextFileWriter object
            TextFileWriter creator = new TextFileWriter();

            // Create the file with the message according to the
            // program parameter
            creator.create(args[0], args[1]);
        }
    }
}
```

Figure 8.49 ArgsToFile.java

Compile the classes and execute the `ArgsToFile` program, such as:

```
java ArgsToFile out.txt "Hello World"
```

It creates a file `out.txt` and the `String` object `"Hello World"` is written to it. (If the message supplied via the second program parameter contains a single word, the pair of double quotation marks is optional.) The file size of the created file `out.txt` is `11`. Execute the `TestBinaryFileViewer` with `out.txt` as program parameter. The following output is shown:

```
Data read from file (out.txt)
Byte (#1) = 72 <H>
Byte (#2) = 101 <e>
Byte (#3) = 108 <l>
Byte (#4) = 108 <l>
Byte (#5) = 111 <o>
Byte (#6) = 32 < >
Byte (#7) = 87 <W>
Byte (#8) = 111 <o>
Byte (#9) = 114 <r>
Byte (#10) = 108 <l>
Byte (#11) = 100 <d>
```

You can see that every time the `write()` method is called and a character (a datum of primitive type `char`) is supplied, the data eventually written to the data destination (the file in this case) are a single byte for all characters defined in ASCII. This confirms that a translation process is involved, and it determines that a two-byte character that corresponds to a character defined in ASCII is translated and written to the data destination as a single byte.

More information on conversion by filter streams can be found in Appendix D.

The standard input and standard output

We have been using statements that look like

```
System.out.println(var);
```

to display the contents of the variable on the screen since the first program you encountered in the course. But we haven't discussed why and how it works. The variable `out` is a class (`static`) variable of the `java.lang.System` (or simply `System`) class. Also, the class defines two more class variables, `in` and `err`, which can be used for input/output purposes. Please use the following reading to get a basic understanding of these class variables.

> ***Reading***
>
> King, 'Class variables in the `System` class', pp. 275–77

The reading explains that the three class variables `in`, `out`, and `err` represent the standard *input* stream, standard *output* stream and standard *error* stream respectively. The type of the class variable `in` is `InputStream` and the types of the class variables `out` and `err` are both `PrintStream`. As they are `public` class variables, it is possible to access them directly, such as:

```
System.out.println();
System.err.println();
```

When the JVM is launched, an `InputStream` object attached to the computer keyboard is prepared and is referred by the class variable `in` of the `System` class. Two `PrintStream` objects are prepared and are made accessible by class variables `out` and `err` of the `System` class. The `PrintStream` objects that are accessible by the class variables `out` and `err` of the `System` class are usually used for displaying application related messages and error/debugging messages respectively.

In the following subsection, we discuss how to manipulate these streams for performing input/output operations.

Reading entries from a keyboard

We mentioned that when the JVM is started, an `InputStream` object that is attached to the keyboard of the computer is prepared by default, and it is made accessible via the class variable `in` of the `System` class. Therefore, it is possible to use the `read()` method of the `InputStream` object to retrieve the byte obtained from the attached keyboard.

Reading bytes from a keyboard

A class `Keyboard1` is written in Figure 8.50 to illustrate how to retrieve a byte from the computer keyboard.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class Keyboard1
public class Keyboard1 {
    // Constant for end of file
    private static final int EOF = -1;

    // Show the entries from keyboard
    public void show() throws IOException {
        int byteCount = 0; // The count of byte read
        int byteRead; // The byte read from keyboard
        // A while loop to read a byte from the keyboard
        while ((byteRead = System.in.read()) != EOF) {
            // Increase the byte count
            byteCount++;
            // Show the byte read
            System.out.print("Byte (#" + byteCount +
                ") = " + byteRead);

            // If the byte read is treated as a character and its code
            // is greater than 31, show it. Otherwise, just skip to
            // next new line.
            if (byteRead > 31) {
                System.out.println(" <" + (char) byteRead + ">");
            } else {
                System.out.println();
            }
        }
    }
}
```

Figure 8.50 Keyboard1.java

The driver program, `TestKeyboard1`, is shown in Figure 8.51.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestKeyboard1
public class TestKeyboard1 {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Show message for terminating the program
        System.out.println(
            "Enter Ctrl-Z/Ctrl-D to terminate the program");

        // Create a Keyboard1 object
```

```
        Keyboard1 keyboard = new Keyboard1();

        // Show the entry from the keyboard
        keyboard.show();
    }
}
```

Figure 8.51 TestKeyboard1.java

Compile the classes and execute the `TestKeyboard1` class. The following message

```
Enter Ctrl-Z/Ctrl-D to terminate the program
```

is shown on the screen, and the program is waiting for your entry from the keyboard. The program will repeatedly get the codes of the keystrokes from the keyboard. To terminate the program, it is necessary to enter a special keystroke representing the end of entry. For the Windows family and UNIX platforms, the keystrokes are `Ctrl-Z` (or `F6`) and `Ctrl-D` respectively.

When you type a non-control key, such as a letter, a number or a symbol on your keyboard, the character is shown on the screen. Before pressing the <Enter/Return> key, you can even use the <arrow> keys, <backspace> and <delete> key to navigate among the characters typed and modify them. For example, the keys <a> to <g> are typed, that is:

```
Enter Ctrl-Z/Ctrl-D to terminate the program
abcdefg
```

Then, you can press the <Enter/Return> key to terminate a line of entry, and the program will show the following message according to the entry:

```
Enter Ctrl-Z/Ctrl-D to terminate the program
abcdefg
Byte (#1) = 97 <a>
Byte (#2) = 98 <b>
Byte (#3) = 99 <c>
Byte (#4) = 100 <d>
Byte (#5) = 101 <e>
Byte (#6) = 102 <f>
Byte (#7) = 103 <g>
Byte (#8) = 13
Byte (#9) = 10
```

You can see that the line of entry contains seven letters, and the output message shows that nine bytes are read. The last two bytes are the default byte pattern of the default end of line of the platform, which is the same as the default end of line for text files.

You can enter another line of entry to the program. If the first keystroke of the line is the default end-of-entry keystroke of the platform, that is, `Ctrl-Z` (or `F6`) for the Windows family and `Ctrl-D` for UNIX, the program will terminate.

More information on standard input can be found in Appendix E.

Displaying messages to standard output and standard error

We have been using the statement

```
System.out.println(var);
```

to display the contents of the variable `var` to the screen without being concerned how the statement works. First, even if you did not know the term `System` in the above statement refers to the class `java.lang.System`, you should notice that according to the naming convention of the Java programming language, the term `System`, which starts with a capital letter, is not a variable name but a class name. Then, the term `System.out` indicates that the variable `out` is a class variable of the class `System`. Furthermore, as it is possible to refer to the variable `out` in any class, the class variable `out` is a `public` class variable of the class `System`. The type of the `public` class variable `out` of the `System` class is `java.io.PrintStream` (or simply `PrintStream`).

When the JVM is started, a `PrintStream` object that attaches to the screen is created automatically and can be accessed by the class variable `out` of the class `System`. Such a stream is known as standard output stream. You can consider a `PrintStream` object a combination of a `PrintWriter` object and an `OutputStreamWriter` object. Like the class `PrintWriter`, the class `PrintStream` defines nine `print()` and ten `println()` methods, so that it is possible to show data of various types to the associated `OutputStream` object with these methods.

The `System` class defines another public class variable `err`, which refers to another `PrintStream` object that attaches to the standard error stream. The standard output stream is usually used for displaying application related messages and the standard error stream is preferably used for displaying errors or debugging messages. By default, the contents sent to the standard error stream are shown on the same screen as the standard output stream. Therefore, the contents sent to the standard output stream and standard error stream are mixed together. You may wonder why there seems to be no difference in sending contents to either the standard output stream or standard error stream. We discuss how to divert the messages sent to the two streams to different destinations very soon.

We can now experiment with using the standard error stream for showing error or debugging messages. For example, we can modify the definition of the class `InsertionSorter` (see Appendix D) so that debugging messages are sent to the standard error stream. The modified version, `InsertionSorter2`, is shown in Figure 8.52.

```
// Definition of class InsertionSorter2
public class InsertionSorter2 {
    // Attributes
    // Array object for storing the numbers
    private int[] numbers = new int[100];
    // Specifies the amount of numbers stored in the array
    private int total;

    // Store a number in the array
    public void storeNumber(int theNumber) {
        // If all array elements are used, use an array of
        // larger size
        if (total == numbers.length) {
            // Create an array with larger capacity
            int[] newArray = new int[numbers.length + 100];
            // Copy the contents from the original array
            for (int i=0; i < total; i++) {
                newArray[i] = numbers[i];
            }
            numbers = newArray;
        }

        // Store the number in the array
        numbers[total++] = theNumber;
    }

    // Get the array object maintained by this object
    public int[] getNumbers() {
        return numbers;
    }

    // Get the total of number stored
    public int getTotal() {
        return total;
    }

    // Set an array object to be the array to be sorted
    public void setNumbers(int[] theNumbers) {
        // Store the array object
        numbers = theNumbers;
        // Update the number of array elements to be sorted
        total = theNumbers.length;
    }

    public void sort() {
        for (int unsorted=1; unsorted < total; unsorted++) {

            // Display the current contents of the array
            // for tracing the operations
            System.err.print("DEBUG: unsorted=" + unsorted);
            System.err.print("\tarray:\t");
            for (int i=0; i < total; i++) {
                System.err.print(numbers[i] + "\t");
            }
            System.err.println();
```

```
            // Declare local variable for temporary numbers of the
            // number to be inserted to the sorted region
            int number = numbers[unsorted];
            // Declare and initialise the first subscript of the
            // sorted region
            int sorted = unsorted - 1;

            // For each number in the sorted region
            while (sorted >= 0) {
                // If the number is greater than the number to be
                // inserted
                if (numbers[sorted] > number) {
                    // Show debug message
                    System.err.println("DEBUG: Copy numbers[" + sorted +
                        "]" + " to numbers[" + (sorted + 1) + "]");

                    // Copy its value to the one with subscript that
                    // is higher by 1
                    numbers[sorted + 1] = numbers[sorted];
                }
                else {
                    // A number is found not to be greater than the
                    // number to be inserted, terminate the process
                    // of checking, i.e., the while loop.
                    break;
                }

                // Decrease the value of sorted, so that the next number
                // in the sorted region is processed
                sorted--;
            }

            // Show debug message
            System.err.println("DEBUG: numbers[" + (sorted + 1) +
                "] = " + number);

            // Insert the number to the array element with subscript
            // that is one higher than the variable sorted.
            numbers[sorted + 1] = number;
        }
    }
}
```

Figure 8.52 InsertionSorter2.java

The way to send messages to the standard error stream is exactly the same as it is to send to the standard output stream. A driver program, the class `SortIntegersFromKeyboard2` is written to sort the numbers entered by an `InsertionSorter2` object. As most parts of the definition of the class `SortIntegersFromKeyboard2` are the same as that of `SortIntegersFromKeyboard1`, the definition of the `SortIntegersFromKeyboard2` is not shown here. Please refer to the course CD-ROM or website for the class definition.

Compile the classes `InsertionSorter2` and `SortIntegersFromKeyboard2`, and execute the `SortIntegersFromKeyboard2` program. You can first enter the number of numbers to be sorted, followed by the numbers to be sorted. The following is a sample execution of the program:

```
Number of integers to be sorted : 5
Input number (#1) = 10
Input number (#2) = 20
Input number (#3) = 15
Input number (#4) = 8
Input number (#5) = 30
DEBUG: unsorted=1      array:  10      20      15      8       30
DEBUG: numbers[1] = 20
DEBUG: unsorted=2      array:  10      20      15      8       30
DEBUG: Copy numbers[1] to numbers[2]
DEBUG: numbers[1] = 15
DEBUG: unsorted=3      array:  10      15      20      8       30
DEBUG: Copy numbers[2] to numbers[3]
DEBUG: Copy numbers[1] to numbers[2]
DEBUG: Copy numbers[0] to numbers[1]
DEBUG: numbers[0] = 8
DEBUG: unsorted=4      array:  8       10      15      20      30
DEBUG: numbers[4] = 30
Sorted numbers:
8       10      15      20      30
```

In the above execution output, the lines starting with the word `"DEBUG"` are shown on the screen via the standard error stream.

It is possible to redirect the standard output stream to a file so that the output sent to the standard output stream is redirected to a file instead of being shown on the screen. For example, the following command

```
java SortIntegersFromKeyboard2 > stdout.txt
```

sends all outputs that are otherwise sent to the standard output stream to the file `stdout.txt`. (The file name that follows the > character in the command specifies the file to which the standard output stream is redirected.) Therefore, all messages shown on the screen for requesting the user to enter the numbers are not shown. We know that the program first requests the number of numbers to be sorted, followed by the numbers to be sorted. The following is a sample execution output:

```
5
10
20
15
8
30
DEBUG: unsorted=1      array:  10      20      15      8       30
DEBUG: numbers[1] = 20
DEBUG: unsorted=2      array:  10      20      15      8       30
DEBUG: Copy numbers[1] to numbers[2]
DEBUG: numbers[1] = 15
```

```
DEBUG: unsorted=3       array:  10      15      20      8       30
DEBUG: Copy numbers[2] to numbers[3]
DEBUG: Copy numbers[1] to numbers[2]
DEBUG: Copy numbers[0] to numbers[1]
DEBUG: numbers[0] = 8
DEBUG: unsorted=4       array:  8       10      15      20      30
DEBUG: numbers[4] = 30
```

The outputs that were originally shown on the screen via the standard output stream are not shown. Instead, the file `stdout.txt` is created and the output that is sent to the standard output stream is stored in the file. The content of the file is:

```
Number of integers to be sorted : Input number (#1) = Input number
(#2) = Input number (#3) = Input number (#4) = Input number (#5) =
Sorted numbers:
8       10      15      20      30
```

Also, if your computer is running Windows NT/2000/XP or UNIX as the operating system, it is possible to redirect the output that is sent to the standard error stream to a file. You can execute a Java program with a command that looks like:

`java SortIntegersFromKeyboard2 2> stderr.txt`

The `2>` characters and the following file name define the destination file of the standard error stream. You can then execute the program as usual. The following is a sample execution output:

```
Number of integers to be sorted : 5
Input number (#1) = 10
Input number (#2) = 20
Input number (#3) = 15
Input number (#4) = 8
Input number (#5) = 30
Sorted numbers:
8       10      15      20      30
```

The execution output is the same as if there are no statements for displaying debugging messages. The outputs that are sent to the standard error stream are not shown on the screen but are redirected to a file `stderr.txt`. If the file `stderr.txt` existed before executing the program, it is overwritten. Otherwise, the file is created. You can now view the content of the file `stderr.txt`:

```
DEBUG: unsorted=1       array:  10      20      15      8       30
DEBUG: numbers[1] = 20
DEBUG: unsorted=2       array:  10      20      15      8       30
DEBUG: Copy numbers[1] to numbers[2]
DEBUG: numbers[1] = 15
DEBUG: unsorted=3       array:  10      15      20      8       30
DEBUG: Copy numbers[2] to numbers[3]
DEBUG: Copy numbers[1] to numbers[2]
DEBUG: Copy numbers[0] to numbers[1]
DEBUG: numbers[0] = 8
```

```
DEBUG: unsorted=4       array:  8       10      15      20      30
DEBUG: numbers[4] = 30
```

Displaying debugging message via the standard error stream gives you the flexibility to separate the normal application messages and debugging messages. If you want to read debugging messages while you are executing the program, you can execute the program without any redirection of the standard error stream. Otherwise, you can redirect the standard error stream so that you can inspect its contents for tracing the program execution afterwards.

You learned that three standard streams are created automatically by the JVM that are attached to the keyboard and the screen for input/output purposes. Furthermore, all standard streams support redirection. The extreme situation is that the standard input stream is redirected so that the keyboard entries come from a text file and the output sent to the standard output stream and standard error stream are redirected to files. For example, the following command

```
java SortIntegersFromKeyboard2 < forSort.txt >
stdout.txt 2> stderr.txt
```

redirects the standard input stream so that the keyboard entries come from the file `forSort.txt`. Furthermore, the output sent to the standard output stream is redirected to the file `stdout.txt` and the output sent to the standard error stream is stored in the file `stderr.txt`. When you enter the above command to execute the program, the program terminates immediately without showing any message on the screen. The program read the inputs from the standard input stream from the file `forSort.txt` and the files `stdout.txt` and `stderr.txt` are created for storing the outputs of the standard output stream and standard error stream respectively. You can then view the contents of the files `stdout.txt` and `stderr.txt` for the execution results and the trace of program execution.

Summary

This unit discusses how a program written in the Java programming language performs input/output operations. As a file is a common storage medium used for data storage, the examples and discussions were mostly file related. However, once you have become familiar with the approach of performing stream-based input/output operations, you can apply the knowledge you acquired in this unit to other stream-enabled storage media or communication channels.

A file is identified by its file name and path name. To manipulate a file, such as inspecting its attributes, you can create a `java.io.File` (or simply `File`) object by providing a `String` object specifying the full path of the file, including the path name and file name. Furthermore, a `File` object in the Java programming language can associate not only with a file but also with a directory. Then, you can call methods of the `File` object to obtain the information about the file.

The Java standard software library provides classes for performing stream-based input/output operations. Stream classes are categorized according to the type of data they process and the direction of data flow. They are summarized in the following table.

	Byte stream	**Character stream**
Input stream	`InputStream` (e.g., `FileInputStream`)	`Reader` (e.g., `FileReader`)
Output stream	`OutputStream` (e.g., `FileOutputStream`)	`Writer` (e.g., `FileWriter`)

If you need other functionalities, such as buffering and data conversion, you can create filter streams. Simple data conversion filter stream has been discussed and extra materials can be found in Appendix D.

Once you are familiar with the stream-based input/output approach, you may find such an approach flexible, as you can replace or insert a stream object for various behaviours. You can then further study the uses of other classes provided by the `java.io` package that are specific to other storage media or communication channels.

During the discussion of performing input/output operations with the Java programming language, we discussed input/output operations that may fail due to an exceptional condition. Then, the statement that performs the operation fails, and it is known as a runtime error or exception. In the Java programming language, an exception is an object that encapsulates the information of the runtime error that occurred.

There are checked and unchecked exceptions. The types of unchecked exception are `Error` and `RuntimeException` and their subclasses. An `Error` is an exceptional condition that is difficult to recover. A `RuntimeException` usually refers to an error that is caused by badly designed programs or logic. The exceptions of types other than `RuntimeException` and `Error` are considered checked exceptions. If a statement or a method call may cause a checked exception, it must be enclosed in a `try/catch` construct or the method has to declare a `throws` clause with the exception type(s) that may occur.

If the statement causes an exception in a `try` block, the statements following the statement are skipped and the `catch` block is checked one by one for remedial operations. If a matching `catch` block is found, the statements in the `catch` block are executed and the exception is considered handled. Afterwards, the execution resumes after the last `catch` block.

If the statement that causes an exception is in a method that is declared with a `throws` clause, however, the statements following the statement are skipped. The flow of control is returned to the caller of the method and the exception handling mechanism of the caller method is carried out. If no exception-handling mechanisms are implemented and the flow of control returns eventually to the `main()` method and no exception handling is implemented, the program will terminate immediately and an error message will be shown on the screen.

Exception handling is an indispensable facility in enterprise or commercial software application developments, because it enables software developers to define remedial operations to be carried out if specific runtime errors occur. Then, the software can try to resume usual operations if a runtime error occurs or at least perform housekeeping operations to keep track of the error that occurred so that software developers can enhance the software applications.

Appendix A: using the Java standard software library documentation

The Java standard software library provides a lot of classes that target the development of different aspects of software applications, such as performing input/output operations and creating graphical user interfaces. As the number of classes and their attributes and methods is huge, it is almost impossible for any reference book to provide details. Therefore, software developers usually prepare the documentation for the classes they developed. To help the reader navigate the documentation, it is presented in HTML format so that it can be viewed and navigated by any Web browser.

The documentation for the Java standard software library is known as the Application Programming Interface (API) specification. It can be accessed through the following URL:
http://java.sun.com/j2se/1.4.1/docs/api/index.html

Figure 8.53 shows a snapshot of the API specification.

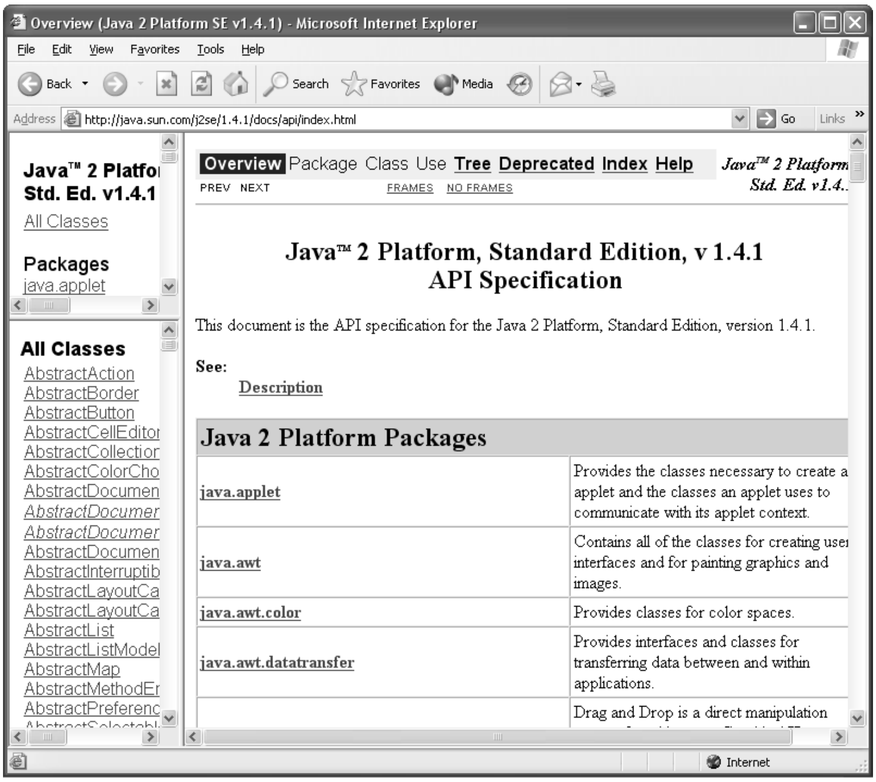


Figure 8.53 The first page of the Java API specification

As the API specification provides detailed information on each class provided in the library, the sizes of the web pages are large. It may take a long time to download a web page with a slow connection to the Internet. Therefore, it is recommended you install the API specification to your computer so that you can read the web pages without accessing the Internet. The API specification can be downloaded from the official Java website, http://java.sun.com/docs/.

Installation of the documentation

The documentation contains plenty of HTML pages and it will occupy up to 180 MB of your computer hard disk. Therefore, you should first verify that your computer hard disk has sufficient free space for the installation.

The steps in installing the documentation are:

1 Download or locate the compressed version of the documentation. All documentation pages are consolidated and compressed becoming a file with `.zip` extension, such as `j2sdk-1_4_0-doc.zip`.

2 Start a Command Prompt with your computer.

3 Change the current directory to the directory for the Java SDK, such as, `C:\j2sdk1.4.0_01\` by command:

```
C:
cd \j2sdk1.4.0_01
```

4 Enter the following command to extract the compressed pages:

```
jar xvf <full path of compressed documentation file>
```

For example, if the full path of the compressed documentation file is `C:\download\j2sdk-1_4_0-doc.zip`, the command is:

```
jar xvf C:\download\j2sdk-1_4_0-doc.zip
```

The above command may take a minute or two to complete.

5 You can now browse the documentation with your web browser at the following link: C:\j2sdk1.4.0_01\docs\api\index.html

Using the API specification

The web browser window is partitioned into three frames — the package frame, class frame and description frame — as shown in Figure 8.54.

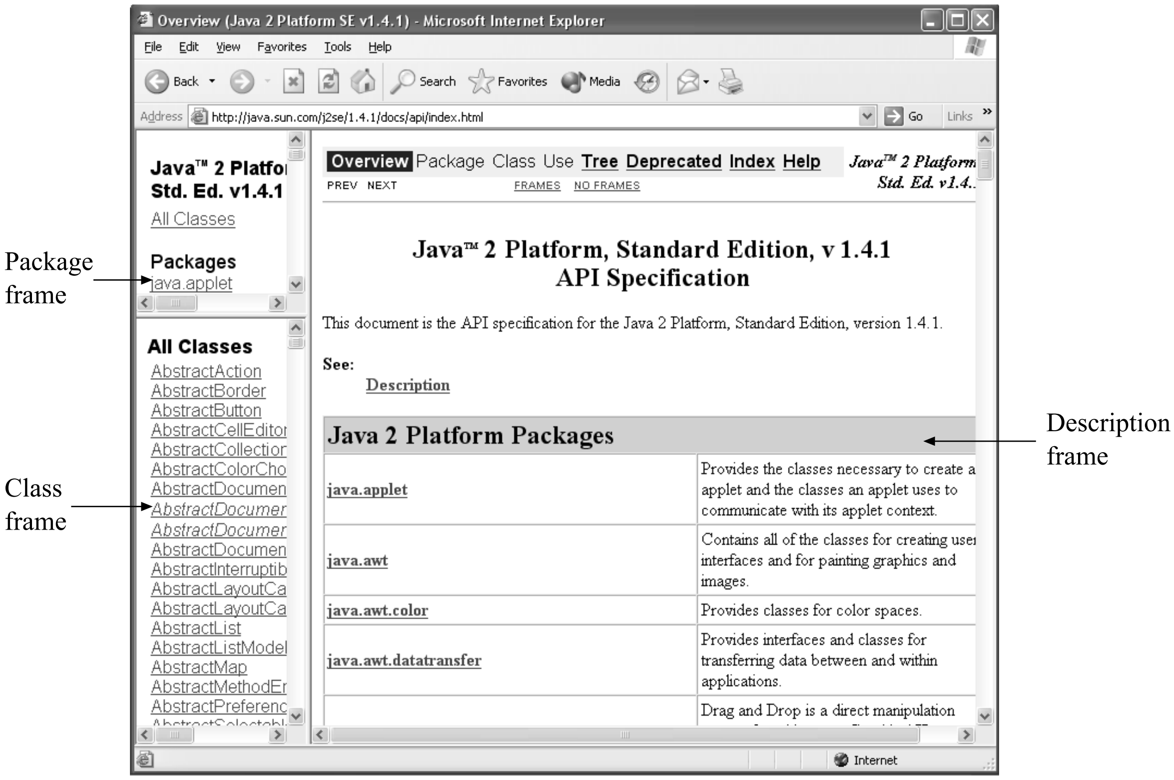


Figure 8.54 The three frames of the web browser window

The functionalities of the frames are:

1 The package frame shows all package names in the Java standard software library. Click a package name with your mouse pointer in the package frame. The class frame will show all the class names in the selected package.

2 The class frame shows all class names in a package, if you click that package name in the package frame. Initially, the class frame shows all class names in all packages. At any time if you want to read all class names in all packages in the class frame, you can click the link All Classes in the package frame.

3 The description frame shows the details of a class. To view the description of a class, click the class name in the class frame. Furthermore, the class may be referred to in the description of other classes. Clicking the class name will show the description of the class in the description frame as well.

The following sections give you some sample ways to navigate the API specification.

Find the description of a class but the package name is unknown

For example, if you want to find the description of the class `System` but you do not know its package name, you can follow the following steps:

1 Click the link All Classes in the package frame.

2 Use the scroll bar of the class frame to locate the name `System`.

3 Click the link System. The description of the class `System` will be shown in the description frame.

Find the description of a class and the package name is known

Even if you know the package name of the desired class, you can navigate the API specification as presented in the previous section. However, as there are plenty of class names in the class frame, it takes a long time to download the web page from the Internet and locate the name of the desired class in the class frame.

You may follow the following steps to find the description for the class `System` in the package `java.lang` and it may take less time.

1 Click the link for the package of the class, the `java.lang` package in this example, in the package frame, so that the package frame shows the names for the classes in that package only.

2 Use the scroll bar of the class frame to locate the name `System`.

3 Click the link System. The description of the class `System` will be shown in the description frame.

Find the description of a method if you do not know the class that defines it

Once you have navigated to the description for the desired class, you can read the details such as constructors, methods and so on. However, if you want to use a method but you do not know the class that defines it, say the `parseInt()` method, you can follow the following steps:

Click the link Index at the top of the description frame. An index-like page is shown.

1 Click the link P at the top of the page in the description frame. The description frame will load the page listing all classes, methods, and class variables and instance variables with names starting with the letter p or P.

2 Use the scroll bar of the description frame (or otherwise). Find the link for the method `parseInt()`. Click the link. The description frame will load the page for the class (`java.lang.Integer`) that defines the `parseInt()` method.

The description of a class

The following is a list of the aspects covered by the description of a class:

1 The parent classes of the class and its subclasses. For example, the description of `java.io.InputStreamReader` shows that it is a subclass of `Reader` class and `Reader` class is a subclass of the `Object` class. Furthermore, the `FileReader` class is a known subclass of the `InputStreamReader` class.

```
java.io
Class InputStreamReader
   java.lang.Object
     |
     +-- java.io.Reader
           |
           +--java.io.InputStreamReader
```

Direct Known Subclasses:

`FileReader`

2 A general description of the class. You can get a basic idea of the class, such as its uses and limitations.

3 Field summary. You can read the usage of the variables defined by the class. By default, only class variables of `public` and `protected` access privilege are shown. You can read the detailed description of a variable by clicking its link. For example, the following is the field summary for the `java.lang.System` class.

Field Summary	
static `PrintStream`	**`err`** The 'standard' error output stream.
static `InputStream`	**`in`** The 'standard' input stream.
static `PrintStream`	**`out`** The 'standard' output stream.

4 The constructor summary is a brief description of the constructor(s). Click the link of a constructor to show its detailed description. For example, the constructor summary of the `InputStreamReader` shows that there are four overloaded constructors.

Constructor Summary
`InputStreamReader`(`InputStream` in) Create an InputStreamReader that uses the default charset.
`InputStreamReader`(`InputStream` in, `Charset` cs) Create an InputStreamReader that uses the given charset.
`InputStreamReader`(`InputStream` in, `CharsetDecoder` dec) Create an InputStreamReader that uses the given charset decoder.
`InputStreamReader`(`InputStream` in, `String` charsetName) Create an InputStreamReader that uses the named charset.

5 Method summary. The methods defined by the class and the methods inherited from its superclasses are shown. For example, the following is the method summary for the `InputStreamReader` class. You can click the link of a method to show its detailed description.

Method Summary	
void	**`close`**() Close the stream.
`String`	**`getEncoding`**() Return the name of the character encoding being used by this stream.
int	**`read`**() Read a single character.
int	**`read`**(char[] cbuf, int offset, int length) Read characters into a portion of an array.
boolean	**`ready`**() Tell whether this stream is ready to be read.

Methods inherited from class java.io.Reader
`mark`, `markSupported`, `read`, `reset`, `skip`

Methods inherited from class java.lang.Object
`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

6 Field Details. You can read the detailed descriptions and uses of the variables defined by the class.

7 The detailed descriptions of the constructor(s) and method(s). You can read the detailed description and use of the constructor(s) and method(s) defined by the class, such as the use of each parameter and the return value of the method under different conditions. Furthermore, it shows the exception(s) that may be caused if the method is called.

Appendix B: modifying the font settings for showing Chinese characters

This appendix gives the procedure to update the settings of the JVM so that Chinese characters can be shown in a graphical user interface, such as a dialog box, by running a software application written in the Java programming language on a computer with a non-Chinese Microsoft Windows. As Chinese processing is only mentioned in this unit of the course, it is recommended you *not change the settings unless you are really interested in experimenting with the `TestInputStreamReader2` program with the file `chinese.txt`, and you are an expert Windows user*.

Suppose that the Java SDK is installed in the directory `C:\j2sdk1.4.0_01` of your computer. You can enter the following commands in sequence to change the font settings:

```
ren c:\j2sdk1.4.0_01\jre\lib\font.properties font.properties.orig
ren c:\j2sdk1.4.0_01\jre\lib\font.properties.zh_TW font.properties
```

You can now experiment with the `TestInputStreamReader2` to see whether Chinese characters can be shown properly. If the Chinese characters still cannot be shown properly, you can try entering the following commands in the Command Prompt. It is assumed that the system directory of the Microsoft Windows is `C:\windows`:

```
ren c:\windows\system32\java.exe java-old.exe
ren c:\windows\system32\javaw.exe javaw-old.exe
```

The program `TestInputStreamReader2` should work properly and the Chinese characters in the `chinese.txt` file should be shown properly now. If you experience any problem in running Java programs, you should restore the changes you have made by entering the following commands:

```
ren c:\windows\system32\javaw-old.exe javaw.exe
ren c:\windows\system32\java-old.exe java.exe
ren c:\j2sdk1.4.0_01\jre\lib\font.properties font.properties.zh_TW
ren c:\j2sdk1.4.0_01\jre\lib\font.properties.orig font.properties
```

Appendix C: getting web pages from the Internet

With what you have learned in the section 'Handling text files with `BufferedReader` and `PrintWriter`' earlier in the unit, we can further enhance the class `URLGetter` mentioned in Self-test 8.3 for retrieving resources from the WWW. The `URLGetter` program accepts two program parameters. The first one is a URL, and the second one is the file name of the target file for storing the retrieved resource. Executing the program, an exact copy of the resource specified by the first program parameter is stored in a file. Such behaviour is acceptable for binary data, such as image files. If you use the `URLGetter` program to retrieve a web page, such as, http://www.ouhk.edu.hk/index.html by the command

```
java URLGetter http://www.ouhk.edu.hk/index.html index.html
```

an exact copy of the file `index.html` that is stored in the web server is saved to a file named `index.html` on the computer drive. A problem is that the `index.html` file stored on your computer drive may not be encoded in the default coding system of your computer but in the default coding system of the web server instead. Furthermore, the end-of-line delimiters used in the file may not be the default delimiters of your computer's operating system. Then, other software applications may not read the file content properly.

Fortunately, most web pages specify the coding systems used, and we can obtain the coding system used for creating an `InputStreamReader` object with a suitable coding system to be involved in the translation. The steps in preparing a character stream from a web page are:

1 Create a `URL` object referring to a web page on the World Wide Web by providing the URL of the web page as the supplementary data to the constructor. That is:

```
URL url = new URL(...);
```

2 Get the reference of the `URLConnection` object associated with the `URL` object.

```
URLConnection conn = url.openConnection();
```

3 Get the coding system used by the web page.

```
String encoding = conn.getContentEncoding();
```

If the resource does not specify the encoding system used, a value of `null` is returned.

4 Get the `InputStream` object from the `URLConnection` object.

```
InputStream is = conn.getInputStream();
```

5 Prepare an `InputStreamReader` object based on the `InputStream` object that is associated with the resource and the coding system used.

```
Reader reader = new InputStreamReader(is, encoding);
```

Then, you can consider it a generic `Reader` object that is a character input stream. For a web page that is basically a plain text file, it is preferable to read the contents line by line. Furthermore, it is possible to create a `BufferedReader` object based on the above `Reader` object, so that it is possible to use its `readLine()` method to read a line from the resource, such as:

```
BufferedReader br = new BufferedReader(reader);
```

For illustration, the lines are simply shown on the screen. The above steps are used in the definition of the class `WebPageGetter` shown in Figure 8.55.

```
// Resolve classes for performing input/output
import java.io.*;
// Resolve URL related classes
import java.net.*;

// The definition of class WebPageGetter
public class WebPageGetter {
    // Attribute
    private BufferedReader in;    // The BufferedReader object for
                                  // reading lines from the URL

    // Constructor
    public WebPageGetter(String urlString) {
        try {
            // Create a URL object associated with the URL address
            URL url = new URL(urlString);

            // Obtain the URLConnection object from the URL object
            URLConnection conn = url.openConnection();

            // Obtain the name of the coding used by the URL resource
            String encoding = conn.getContentEncoding();

            // Obtain the InputStream object from the URLConnection
            InputStream is = conn.getInputStream();

            // Create a InputStreamReader object based on the
            // InputStream obtained from the URLConnection. If encoding
            // is specified, it is supplied to the InputStreamReader for
            // translation. Otherwise, the default coding system of the
            // computer is used.
            Reader reader = null;
```

```
            if (encoding != null) {
                reader = new InputStreamReader(is, encoding);
            } else {
                reader = new InputStreamReader(is);
            }

            // Create a BufferedReader object based on the
            // InputStreamReader object
            in = new BufferedReader(reader);
        } catch (IOException ioe) {
            // Show error message if there is any I/O runtime error
            System.out.println("Error in performing I/O operation");
        }
    }

    // Show the content of the web resource
    public void showContent() {
        // If the BufferedReader is initialized
        if (in != null) {
            try {
                // Read the lines from the URL resource and show them on
                // the screen
                String line = null;
                while ((line = in.readLine()) != null) {
                    System.out.println(line);
                }
            } catch (IOException ioe) {
                System.out.println("Error in performing I/O operation");
            }
        } else {
            System.out.println("Cannot read from the URL");
        }
    }
}
```

Figure 8.55 WebPageGetter.java

To test the `WebPageGetter` class, a driver program `TestWebPageGetter` is written in Figure 8.56.

```
// The definition of class TestWebPageGetter
public class TestWebPageGetter {

    // Main executive method
    public static void main(String args[]) {
        // Verify the number of program parameter
        if (args.length < 1) {
            // If no program parameter is given, show usage message
            System.out.println("Usage: java TestWebPageGetter <url>");
        }
        else {
            // Create a WebPageGetter object that is associated with
            // the URL
            WebPageGetter getter = new WebPageGetter(args[0]);
```

```
            // Show the contents of the resource specified by the URL
            getter.showContent();
        }
    }
}
```

Figure 8.56 TestWebPageGetter.java

Compile the classes and execute the `TestWebPageGetter` program by specifying the URL of a web page as the program parameter, such as:

`java TestWebPageGetter http://www.ouhk.edu.hk/index.html`

The contents of the file `index.html` stored on the web server will be shown on the screen.

The flow of data among the stream objects involved in the `WebPageGetter` is visualized in Figure 8.57.

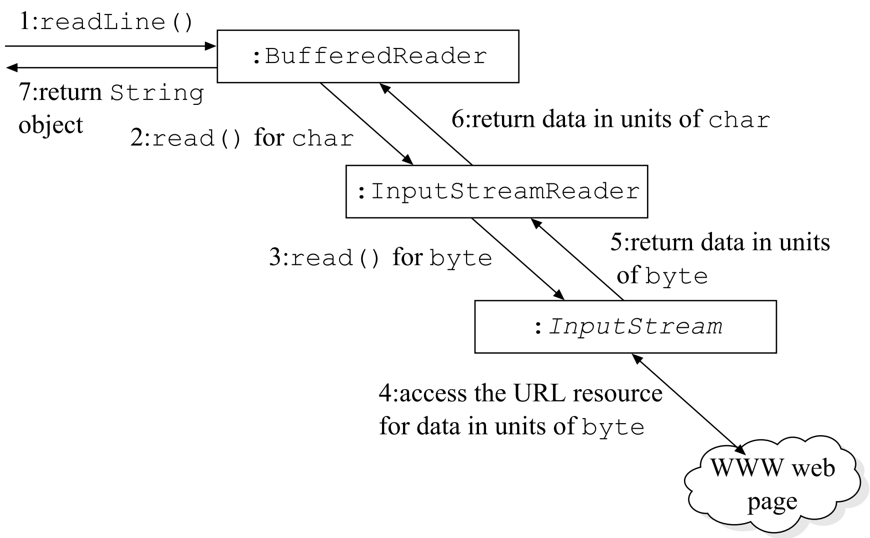


Figure 8.57 The sequence of operations behind calling the `readLine()` method by a `WebPageGetter` object

Please use the following self-test to experiment with the use of `InputStreamReader`, `OutputStreamWriter`, `BufferedReader` and `PrintWriter` discussed in this appendix.

Self-test 8.12

Modify the `WebPageGetter` class by adding a `copyContentToFile()` method,

```
public void copyContentToFile(String filename) {
    ...
}
```

Then, the retrieved is web page is written to the specified file line by line.

Modify the `TestWebPageGetter` class so that it can accept an optional second program parameter for the file name to store the web page by calling the `copyContentToFile()` method. If the second program parameter is missing, the lines read from the web page are shown on the screen by calling the `showContent()` method.

Appendix D: more on filter streams

The JVM performs the translation between bytes and characters according to the default coding system of the operating system. For some coding systems, however, the translation is not simply between a single byte and a two-byte character. For example, as there are more than 10,000 Chinese characters, it is necessary to use two bytes to represent a single Chinese character. Therefore, every Chinese character is stored in a byte-oriented storage with two bytes, that is:

香	港	公	開	大	學	O	P	E	N	U

Each Chinese character occupies two bytes in the file, whereas each ASCII character takes one byte only. When the content is read from the file, the combination of two bytes that constitutes a Chinese character is translated into the single corresponding two-byte Unicode character. However, if the byte by itself represents a character in ASCII, such as an English letter, the byte is translated into a two-byte Unicode character. When the contents in the above file is read, it is expected that the bytes read are translated to a sequence of characters as in the following:

```
'香','港','公','開','大','學','O','P','E','N','U'
```

Seventeen bytes in the files are translated into 11 Unicode characters in the JVM. The `Reader` and `Writer` classes provided by the Java software library support the translation transparently.

Now, let’s examine executing the `TestBinaryFileViewer` (using a byte-based `InputStream`) and `TestTextFileViewer1` (using a character-based `Reader`) with a file containing Chinese characters. The content of the file, `chinese.txt`, is prepared on a computer running a Chinese version of Microsoft Windows as shown in Figure 8.58.

Figure 8.58 A text file with Chinese characters

The file size of the `chinese.txt` file is `17` bytes. Execute the `TestBinaryFileViewer` program with the file `chinese.txt`. The output is shown in Figure 8.59.

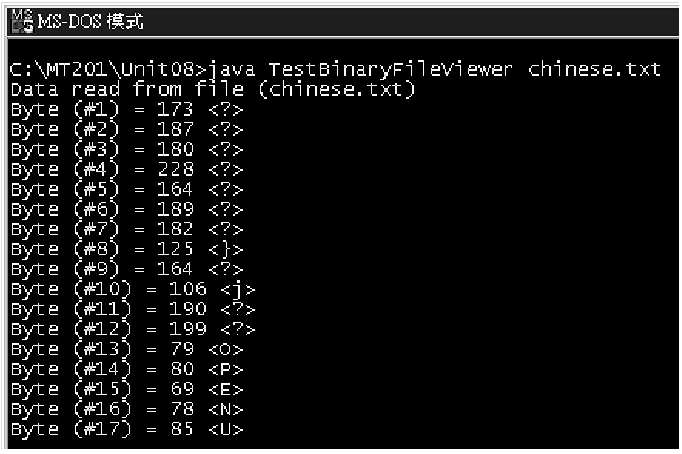


Figure 8.59 The contents of the chinese.txt file is shown as a sequence of bytes

The value of each byte in the file is shown by executing the `TestBinaryFileViewer` with the `chinese.txt` file. Now execute the `TestTextFileViewer1` with the `chinese.txt` file. Figure 8.60 shows the output:

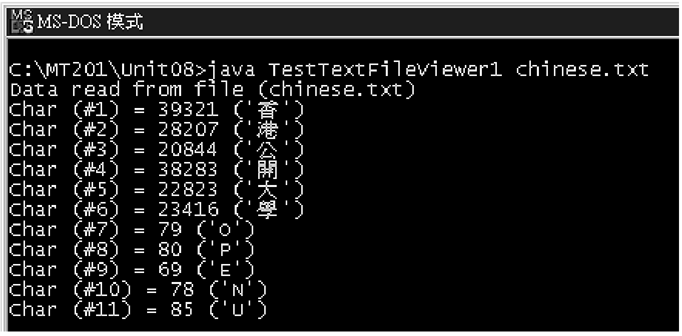


Figure 8.60 The contents of the chinese.txt file is shown as a sequence of characters

From the output, you can see that the `FileReader` object (a `Reader` object) properly identifies the Chinese characters in the sequence of bytes and converts the two bytes into a single character (in Unicode) representing a Chinese character.

`Reader`/`Writer` objects help us to resolve the issue of translation between byte and Unicode character according to a particular coding system. By default, the coding system involved in the translation is the default coding system of the operating system of the computer. If you run the above program on a computer running a traditional Chinese version of Microsoft Windows, the default coding system of the machine is probably Big5, a common coding system for traditional Chinese characters. The bytes from the file are considered encoded in the Big5 coding system, and the translation between bytes and characters is determined to be between the Big5 coding system and Unicode.

Going the other way, if a character is written by a `Writer` object, the 2-byte character (in Unicode) will be translated to either one or two bytes according to the coding system involved. Similarly, the translation of the character is, by default, from Unicode to the default coding system of the operating system.

The conversions between Unicode characters and bytes

As a file is commonly used as a data source and data destination, the Java standard software library provides the `FileInputStream` and `FileReader` classes for reading the data from a file. The returned values are in units of `byte` and `char` respectively. For some storage or communication media, there may be no character-based classes. Only byte-based class input stream or output stream are available. To bridge the expectations of the program for handling data in units of `char` and the media for storing data in units of `byte`, the Java standard software library provides the classes `OutputStreamWriter` and `InputStreamReader` for handling the translation. The scenario is visualized in Figure 8.61.

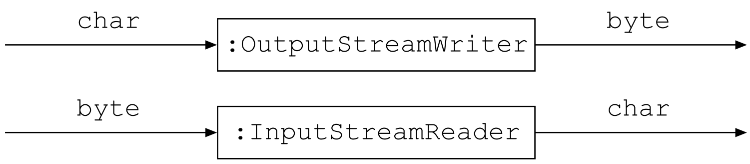


Figure 8.61 The functionality of an `OutputStreamWriter` object and an `InputStreamReader` object

The `OutputStreamWriter` class for the conversion from Unicode characters to bytes

The `java.io.OutputStreamWriter` (or simply `OutputStreamWriter`) class is a concrete subclass of the `Writer` class. Its constructor expects supplementary data as the reference to a byte-based `OutputStream` object, that is:

```
public OutputStreamWriter(OutputStream out)
```

Therefore, an `OutputStreamWriter` object always refers to a byte based `OutputStream` object that is supplied to the constructor.

As it is a subclass of the `Writer` class, it possesses `write()` methods. The characters supplied to the methods via the parameter are translated into bytes according to the conversion between Unicode and the default coding system of the computer. Then, the `write()` method of the associated byte-based `OutputStream` object is called with supplementary data of the translated bytes, so that the bytes are written to its associated medium. Figure 8.62 visualizes the scenario and the relationship between the `OutputStreamWriter` and `OutputStream` objects.

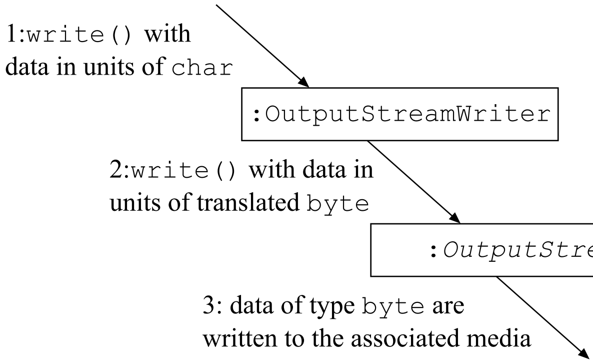


Figure 8.62 Operations behind calling the `write()` method of an `OutputStreamWriter` object

You should notice that the class name `OutputStream` in Figure 8.62 is set in italics, as it is an `abstract` class, which will be replaced by an object of a concrete subclass of the `abstract` class `OutputStream`, such as a `FileOutputStream` object that is associated with a file. Then, the runtime scenario is visualized as shown in Figure 8.63.

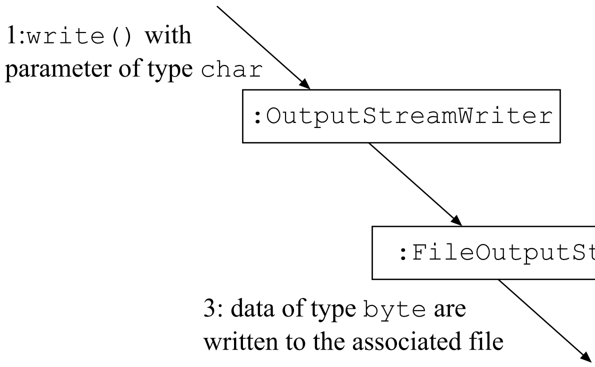


Figure 8.63 Storing data of type `char` to a file with combination of `OutputStreamWriter` and `FileOutputStream` objects

To construct the scenario as shown in Figure 8.63, it is necessary to create a `FileOutputStream` object first (an object of subclass `OutputStream`) and then an `OutputStreamWriter` object by supplying the reference of the `FileOutputStream` object. That is

```
OutputStream os = new FileOutputStream(...);
Writer out = new OutputStreamWriter(os);
```

or simply:

```
Writer out = new OutputStreamWriter(new FileOutputStream(...));
```

Then, it is possible to call the `write()` method of the `OutputStreamWriter` object with supplementary character data. The translated bytes will then be sent to the associated `FileOutputStream` (an object of the concrete subclass of `OutputStream`) object and hence written to the associated file.

The above implementation is applied in the definition of class `TextFileCreator2` shown in Figure 8.64.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TextFileCreator2
public class TextFileCreator2 {

    // Create the file with the char array as its contents
    public void create(String name, char[] message) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);

        // Create a FileOutputStream object specified by the File
        // object and is treated as a general OutputStream object
        OutputStream os = new FileOutputStream(file);

        // Create a OutputStreamWriter by supplying the reference
        // of an existing OutputStream object
        Writer writer = new OutputStreamWriter(os);

        // Send the character data to the Writer one by one
        for (int i=0; i < message.length; i++) {
            writer.write(message[i]);
        }

        // Close the stream and release all corresponding resources
        writer.close();
    }
}
```

Figure 8.64 TextFileCreator2.java

A driver program `TestTextFileCreator2` similar to `TestTextFileCreator1` (Figure 8.18) is written and is available from the course CD-ROM and course website. Compile the classes and execute the `TestTextFileCreator2` program with a file name, say `osw.txt`, as program parameter:

```
java TestTextFileCreator2 osw.txt
```

The file `osw.txt` is created or overwritten with the contents "`Hello World`".

Compared with the definition of class `TestFileCreator1` shown in Figure 8.17, you can see that the following two program segments are practically equivalent

```
Writer out = new FileWriter(file);
```

and:

```
OutputStream os = new FileOutputStream(file);
Writer out = new OutputStreamWriter(os);
```

The `InputStreamReader` class for the conversion from bytes to Unicode characters

To read Unicode characters from a byte-oriented data source stream, it is possible to use an `InputStreamReader` object associated with an existing `InputStream` object. Like `OutputStreamWriter`, a `java.io.InputStreamReader` (`InputStreamReader`) object is created by supplying the reference of an existing `InputStream` object that provides a byte-based input stream such as a `FileInputStream` object. The class defines overloaded constructors, and the following is usually used:

```
public InputStreamReader(InputStream in)
```

`InputStreamReader` is a subclass of the class `Reader` and it therefore possesses `read()` methods that return data from the associated data source in units of Unicode character. The scenario of calling the `read()` method of an `InputStreamReader` object is visualized in Figure 8.65.

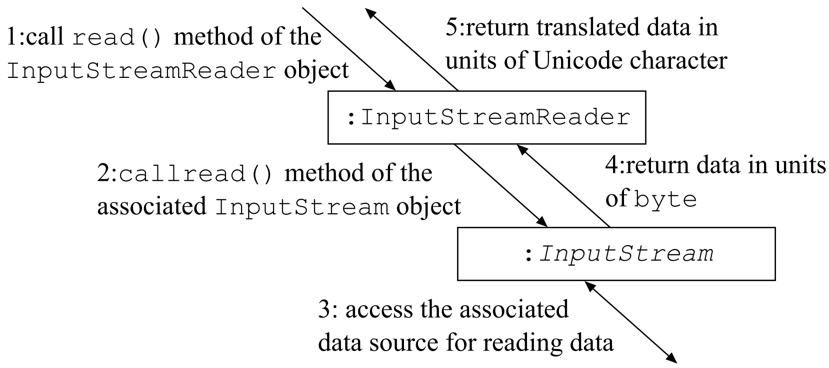


Figure 8.65 The scenario of calling the `read()` method of an `InputStreamReader` object

In Figure 8.65, the name `InputStream` is set in italics, as the class is `abstract` and it must become an object of a concrete subclass of the `abstract` superclass `InputStream` at runtime.

The sequence of operations involved in calling the `read()` method of an `InputStreamReader` object is:

1 The `read()` method of an `InputStreamReader` object is called for reading data in units of Unicode character.

2 In order to return data in units of Unicode character, the `InputStreamReader` object calls the `read()` method of its associated `InputStream` object for reading data in units of `byte`.

3 The `InputStream` object accesses its associated data source, such as a file for a `FileInputStream` object, and returns the data in units of `byte`.

4 The `InputStream` object returns the data in units of `byte` to the `InputStreamReader` object.

5 Once the `InputStreamReader` object obtains the data in units of `byte` from the associated `InputStream` object, it translates the sequence of bytes into a sequence of Unicode characters that are returned.

To illustrate the use of the `InputStreamReader` class, a `TextFileViewer2` class is written in Figure 8.66.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TextFileViewer2
public class TextFileViewer2 {
    // Final variable for end of file
    private static final int EOF = -1;

    // Show the contents of a text file
    public void show(String name) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);

        // Create a FileReader object specified by the File
        // object and is treated as a general InputStream object
        InputStream is = new FileInputStream(file);

        // Create an InputStreamReader by supplying the reference
        // of the InputStream object and is treated as a general
        // Reader object
        Reader reader = new InputStreamReader(is);

        // Read the data from the reader and show the data
        // on the screen
        System.out.println("Data read from file (" +
            file.getPath() + ")");
        int charRead;
        int charCount = 0;
        while ((charRead = reader.read()) != EOF) {
            charCount++;
            System.out.print("Char (#" + charCount + ") = " + charRead);
            // If the character is displayable characters
            // (the code is >= 32), show it
            if (charRead >= 32) {
                System.out.println(" ('" + (char) charRead + "')");
            } else {
                System.out.println();
            }
        }
        // Close the reader and release all corresponding resources
        reader.close();
    }
}
```

Figure 8.66 TextFileViewer2.java

A driver program `TestTextFileViewer2` similar to `TestTextFileViewer1` is written and is available from the course CD-ROM and website. Compile the classes and execute the `TestTextFileViewer2` program with a program parameter of a file name, such as:

```
java TestTextFileViewer2 out.txt
```

The following output is shown on the screen:

```
Data read from file (out.txt)
Char (#1) = 72 ('H')
Char (#2) = 101 ('e')
Char (#3) = 108 ('l')
Char (#4) = 108 ('l')
Char (#5) = 111 ('o')
Char (#6) = 32 (' ')
Char (#7) = 87 ('W')
Char (#8) = 111 ('o')
Char (#9) = 114 ('r')
Char (#10) = 108 ('l')
Char (#11) = 100 ('d')
```

Compare the classes `TextFileCreator1` (Figure 8.17) and `TextFileViewer1` (Figure 8.20) with `TextFileCreator2` and `TextFileViewer2`. You can see that the parts for reading or writing the data are the same. The differences are the ways to create the `Reader` and `Writer` object, as summarized in Table 8.5.

Table 8.5 The two ways of creating `Reader` and `Writer` objects for file accesses

`InputStream is = new FileInputStream(...); Reader reader = new InputStreamReader(is);`	`Reader reader = new FileReader(...);`
`OutputStream os = new FileOutputStream(...); Writer writer = new OutputStreamWriter(os);`	`Writer writer = new FileWriter(...);`

As files are frequently accessed for retrieving data in units of Unicode characters, the Java standard software library provides the `FileReader` and `FileWriter` classes as shorthand so that it is not necessary to prepare the byte stream (a `FileInputStream` or `FileOutputStream`) and the character stream (a `InputStreamReader` or `OutputStreamWriter` object).

Although using the `FileReader` and `FileWriter` classes is handier, the use of `InputStreamReader` and `OutputStreamWriter` gives you extra flexibility. You can choose the coding system to be involved in the translation to Unicode characters.

The translation used by a `FileReader` or `FileWriter` object is the translation between the default coding system of the operating system and Unicode. They provide no methods for changing the involved coding system. However, it is possible to specify the coding system involved in the translation for an `InputStreamReader` or `OutputStreamWriter` by supplying a `String` object with contents of the coding system as the second parameter to their constructors, such as:

```
public InputStreamReader(InputStream in, String charsetName)
public OutputStreamWriter(OutputStream out, String charsetName)
```

For example, the file `chinese.txt` contains traditional Chinese characters encoded in Big5. If you want to read it by running a program written in the Java programming language on a computer that does not use Big5 as the default coding system, you can supply the `String` object `"BIG5"` as the second parameter to the `InputStreamReader` constructor to specify the translation is between Unicode characters and traditional Chinese characters. That is:

```
InputStream is = new FileInputStream(...);
Reader reader = new InputStreamReader(is, "BIG5");
```

The above two statements are used in the definition of the class `TextDialog` shown in Figure 8.67 for showing the contents of a text file in a dialog box on the screen.

```
// Resolve classes in java.io and javax.swing packages
import java.io.*;
import javax.swing.*;

// Definition of class TextDialog
public class TextDialog {

    public void show(String name, String charSet) throws IOException {
        // Create a File object referring to the physical file
        File file = new File(name);

        // Create a FileInputStream object based on the File object
        InputStream is = new FileInputStream(file);

        // Prepare the InputStreamReader object. If there is a
        // second program parameter, it is used as the coding system
        // involved in translation. Otherwise, just create a
        // InputStreamReader using the default coding system.
        Reader reader = null;
        if (charSet != null) {
            reader = new InputStreamReader(is, charSet);
        }
        else {
            reader = new InputStreamReader(is);
        }

        // Read the character from the Reader (InputStreamReader)
        // object one by one and concatenate them
        String content = "";
```

```
        int charRead;
        while ((charRead = reader.read()) != -1) {
            content += (char) charRead;
        }

        // Use a dialog to show the String
        JOptionPane.showMessageDialog(null, content);
    }
}
```

Figure 8.67 TextDialog.java

The corresponding driver program `TestTextDialog` is written in Figure 8.68 to obtain a file name and an encoding system name from the program parameters that are supplied to the `show()` method of a `TextDialog` object.

```
// Resolve classes in java.io package
import java.io.*;

// The class definition of TestTextDialog
public class TestTextDialog {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Validate the number of program parameters.
        if (args.length < 1) {
            // If no parameter is provided, show usage message
            System.out.println(
                "Usage: java TestTextDialog filename> [<coding>]");
        }
        else {
            // Create a TextDialog object
            TextDialog dialog = new TextDialog();

            // Show the dialog with the contents of the file
            // and the encoding system name, if provided
            dialog.show(args[0],
                (args.length >= 2) ? args[1] : null);

            // Explicitly terminate the program
            System.exit(0);
        }
    }
}
```

Figure 8.68 TestTextDialog.java

Compile the classes and execute the `TestTextDialog` program on a non-Chinese operating system with the file name `chinese.txt` as the program parameter. That is:

`java TestTextDialog chinese.txt`

A dialog box that looks like Figure 8.69 will be shown.

Figure 8.69 The contents of a file that contains traditional Chinese characters on a non-traditional Chinese version of an operating system

The contents of the file `chinese.txt` cannot be shown properly in the dialog box because the Chinese characters in the file are encoded in Big5, but the translation is not translating the bytes from Big5 to Unicode. As a result, the characters are not properly translated and cannot be shown properly in the box.

In order to show the contents of the file `chinese.txt` properly on the screen, it is necessary to provide the coding system involved in the translation, such as:

```
java TestTextDialog chinese.txt Big5
```

Then, the dialog box in Figure 8.70 is shown on the screen:

Figure 8.70 The traditional Chinese characters are properly shown on the screen by providing the appropriate coding system involved in the translation

If you experiment with the `TestInputStreamReader2` program by executing the above command and the Chinese characters still cannot be shown properly, such as the dialog shown in Figure 8.71, it may be that the fonts of the JVM are not properly set. Appendix B gives the procedure to change the font settings of the JVM for displaying Chinese characters.

Figure 8.71 The Chinese characters cannot be shown due to improper font settings

In conclusion, the classes `InputStreamReader` and `OutputStreamWriter` always associate with existing objects of concrete classes of `InputStream` and `OutputStream` respectively, such as `FileInputStream` and `FileOutputStream` objects. They bridge a character stream with a byte stream by performing translation between characters in Unicode and bytes in a particular encoding system. The encoding system in the translation is, by default, the default coding system of the operating system. If necessary, it is possible to specify the coding system involved in the translation by providing a second parameter as the reference of a `String` object that contains a coding system name.

The `DataInputStream` and `DataOutputStream` classes

We mentioned that most storage media and communication channels are byte oriented. In the previous subsection, we discussed the use of `InputStreamReader` and `OutputStreamWriter` that are associated with an `InputStream` object and an `OutputStream` object respectively to perform the translation between Unicode characters and bytes.

Besides the conversion between Unicode characters and bytes involved in reading and writing textual data, it is usually necessary to read and write primitive values with respect to a storage medium or communication channel. However, the concrete subclasses of `InputStream` and `OutputStream` classes for particular storage media or communication channels, such as the classes `FileInputStream` and `FileOutputStream` for files, can handle reading and writing data in units of `byte` only. Therefore, it is necessary to translate primitive values, such as 32-bit values of type `int` and 64-bit values of type `long` or `double`, to a sequence of bytes so that they can be supplied to the specific `InputStream` and `OutputStream` objects for performing actual reading and writing operations.

The Java standard software library provides the classes `DataInputStream` and `DataOutputStream` for handling the conversion between primitive types and bytes to be handled by objects of the concrete subclasses of the classes `InputStream` and `OutputStream`. Please use the following reading to learn the uses of these two classes.

> ***Reading***
>
> King, section 14.5, ‘Reading and writing data types’, pp. 625–28

The reading informs you that `DataInputStream` and `DataOutputStream` objects are created based on `FileInputStream` and `FileOutputStream` objects respectively. Actually, the definitions of their constructors are:

```
public DataInputStream(InputStream in)
```

```
public DataOutputStream(OutputStream out)
```

Therefore, it is possible to create objects of the two classes that associate with objects of any concrete subclasses of the abstract superclasses `InputStream` and `Outputstream`. Objects of the classes `FileInputStream` and `FileOutputStream` are typical in that they are associated with files.

The classes `DataInputStream` and `DataOutputStream` are subclasses of `FilterInputStream` and `FilterOutputStream` respectively. Furthermore, `FilterInputStream` and `FilterOutputStream` are subclasses of `InputStream` and `OutputStream` respectively. The relationships among these classes are visualized in Figure 8.72.

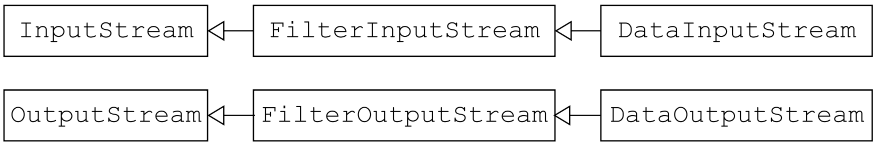


Figure 8.72 The class hierarchies of the classes `DataInputStream` and `DataOutputStream`

As `DataInputStream` and `DataOutputStream` are indirect subclasses of `InputStream` and `OutputStream`, they possess the methods `read()` and `write()`. Also, they define their own methods for performing reading and writing operations as listed in Table 8.6.

Table 8.6 Methods defined by the classes `DataInputStream` and `DataOutputStream`

	`DataInputStream`	`DataOutputStream`

Primitive type related	`boolean readBoolean()`
	`byte readByte()`
	`char readChar()`
	`double readDouble()`
	`float readFloat()`
	`int readInt()`
	`long readLong()`
	`short readShort()`
		`void writeBoolean(boolean v)`
		`void writeByte(int v)`
		`void writeChar(int v)`
		`void writeDouble(double v)`
		`void writeFloat(float v)`
		`void writeInt(int v)`
		`void writeLong(long v)`
		`void writeShort(int v)`

`String` related	`String readUTF()`	`void writeBytes(String s)`
		`void writeChars(String s)`
		`void writeUTF(String s)`

Calling the `writeXXX()` method of a `DataOutputStream`, such as the `writeInt()` method, with supplementary data of the corresponding primitive value, the supplied primitive value will be converted into a corresponding sequence of bytes to be written to the associated `OutputStream` object. For example, calling the `writeInt()` with supplementary data of a value of type `int`, the primitive value is converted into 4 bytes and the 4 bytes are sent to the associated `OutputStream` object for actual writing.

In reading by calling the `readXXX()` method of a `DataInputStream` object, the `DataInputStream` object will read the necessary bytes from the associated `InputStream` object for conversion to be returned as the primitive type. For example, calling the `readInt()` method of a `DataInputStream` object, it will read 4 bytes from the associated `InputStream` object. The 4 bytes are converted into a single value of type `int` to be returned as the return value of the `readInt()`.

The numbers of bytes to be read from an `InputStream` object or written to an `OutputStream` object are not our concern. However, it is clear that the order of primitive values sent to a `DataOutputStream` object determines the order of the corresponding bytes to be sent to the ultimate associated storage medium or communication channel. Therefore, it is also the order of the primitive values to be read if the storage media or communication channel is used as the data source.

To illustrate the use of `DataInputStream` and `DataOutputStream`, two classes `DataFileCreator` and `DataFileViewer` are shown in Figure 8.73 and Figure 8.74.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class DataFileCreator
public class DataFileCreator {

    // Create a data file with the specified file name
    public void create(String name) throws IOException {
        // Create a File object associated with the file
        File file = new File(name);

        // Create a FileOutputStream associated with the File
        // and used as a general OutputStream object
        OutputStream os = new FileOutputStream(file);

        // Create a DataOutputStream object associated with the
        // OutputStream object (actually a FileOutputStream object)
        DataOutputStream dos = new DataOutputStream(os);

        // Call the writeXXX() methods to write the primitive values
        // to the associated OutputStream (and hence the file)
        dos.writeBoolean(true);
        dos.writeByte((byte) 13);
        dos.writeChar('A');
        dos.writeDouble(3.1415);
        dos.writeFloat(1.414F);
        dos.writeInt(100);
        dos.writeLong(2L);
        dos.writeShort((short) 20);
        dos.writeUTF("MT201");

        // After writing operation, close the stream
        dos.close();
    }
}
```

Figure 8.73 DataFileCreator.java

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class DataFileViewer
public class DataFileViewer {

    // Show the contents of a data file
    public void show(String name) throws IOException {
        // Create a File object associated with the file
        File file = new File(name);

        // Create a FileInputStream associated with the File
        // and used as a general InputStream object
        InputStream is = new FileInputStream(file);

        // Create a DataInputStream object associated with the
        // InputStream object (actually a FileInputStream object)
        DataInputStream dos = new DataInputStream(is);

        // Call the readXXX() methods to read the primitive values
        // to the associated InputStream (and hence the file)
        System.out.println("boolean = " + dos.readBoolean());
        System.out.println("byte = " + dos.readByte());
        System.out.println("char = " + dos.readChar());
        System.out.println("double = " + dos.readDouble());
        System.out.println("float = " + dos.readFloat());
        System.out.println("int = " + dos.readInt());
        System.out.println("long = " + dos.readLong());
        System.out.println("short = " + dos.readShort());
        System.out.println("String = " + dos.readUTF());

        // After reading operation, close the stream
        dos.close();
    }
}
```

Figure 8.74 DataFileViewer.java

Two driver programs `TestDataFileCreator` and `TestDataFileViewer` are written in Figure 8.75 and Figure 8.76 for using a `DataFileCreator` object and a `DataFileViewer` object respectively.

```
// Resolve classes in the java.io package
import java.io.*;

// Definition of class TestDataFileCreator
public class TestDataFileCreator {

    // Main execute method
    public static void main(String args[]) throws IOException {
        // Validate the number of program parameter provided
        if (args.length < 1) {
            // If no program parameter is provided, show usage message
            System.out.println(
                "Usage: java TestDataFileCreator <file>");
        }
        else {
            // Create a DataFileCreator object
            DataFileCreator creator = new DataFileCreator();

            // Create the data file by supplying the file name
            creator.create(args[0]);
        }
    }
}
```

Figure 8.75 TestDataFileCreator.java

```
// Resolve classes in the java.io package
import java.io.*;

// Definition of class TestDataFileViewer
public class TestDataFileViewer {

    // Main execute method
    public static void main(String args[]) throws IOException {
        // Validate the number of program parameter provided
        if (args.length < 1) {
            // If no program parameter is provided, show usage message
            System.out.println("Usage: java TestDataFileViewer <file>");
        }
        else {
            // Create a DataFileViewer object
            DataFileViewer viewer = new DataFileViewer();

            // Show the contents of a data file specified by
            // the program parameter
            viewer.show(args[0]);
        }
    }
}
```

Figure 8.76 TestDataFileViewer.java

Compile the classes `DataFileCreator` and `TestDataFileCreator`, and execute the `TestDataFileCreator` with a program parameter of a file name, such as:

```
java TestDataFileCreator data.dat
```

A file `data.dat` is created. The `create()` method of the class `DataFileCreator` calls the `writeXXX()` methods of the `DataOutputStream` object, which is associated with a `FileOutputStream` object and hence with the file for converting the primitive values provided. The converted bytes are then supplied to the `FileOutputStream` object for writing to the file.

To read the data stored in the `data.dat` file, the `show()` method of the class `DataFileViewer` uses a `FileInputStream` to associate with the file and uses a `DataInputStream` object to read and convert the bytes obtained from the `FileInputStream` object.

Compile the classes `DataFileViewer` and `TestDataFileViewer`, and execute the `TestDataFileViewer` program with the file `data.dat` as the program parameter; that is:

```
java TestDataFileViewer data.dat
```

The following output is shown on the screen:

```
boolean = true
byte = 13
char = A
double = 3.1415
float = 1.414
int = 100
long = 2
short = 20
String = MT201
```

The operations behind the classes `DataFileCreator` and `DataFileViewer` are visualized in Figure 8.77.

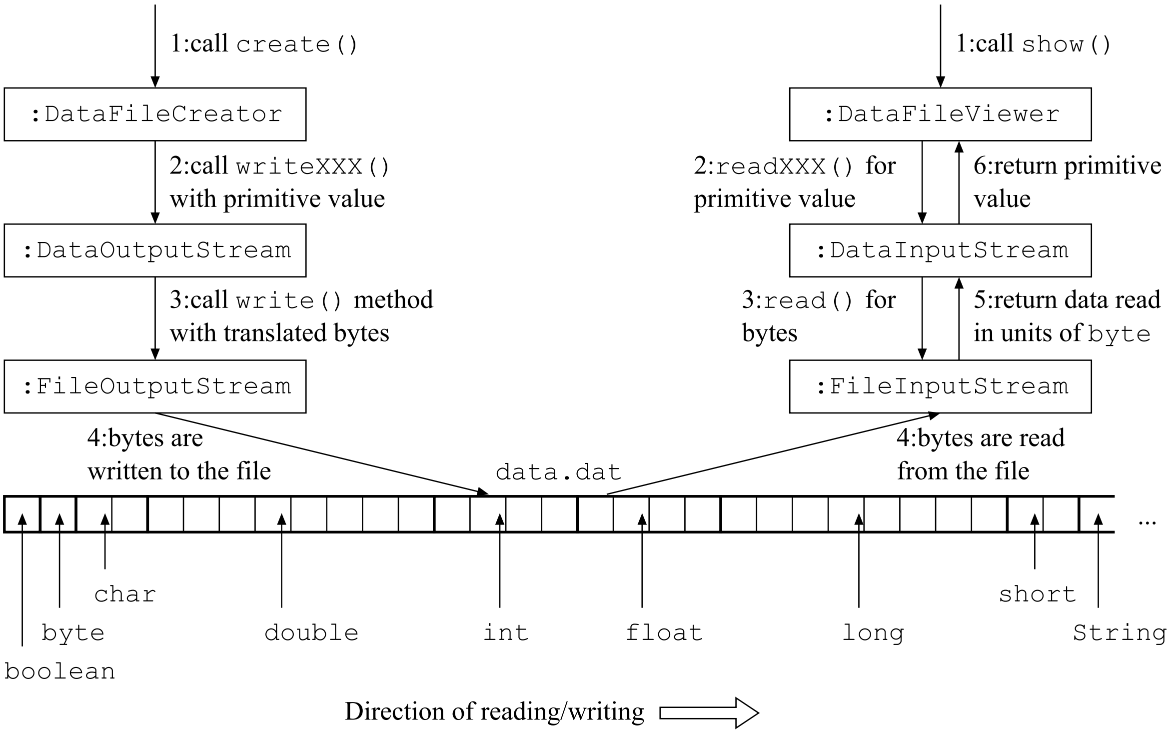


Figure 8.77 The operations behind the `DataFileCreator` and `DataFileViewer` and the contents of the file data.dat

In Figure 8.77, the contents of the file `data.dat` are shown and the bytes in the files are partitioned according to the values written to it. The bytes for the `String` object are not completely shown, to save space.

You may notice that although both the `write()` method of the class `OutputStreamWriter` and the `writeChar()` method of the class `DataOutputStream` accept a character to be sent to the associated `OutputStream`, their behaviours are different. The `write()` method of the class `OutputStreamWriter` translates the characters from Unicode to byte(s) of the default encoding system of the operating system for sending to the associated `OutputStream` object. The `writeChar()` method of the `DataOutputStream` simply sends the two bytes to the associated `OutputStream` object. Therefore, writing Unicode characters by an `OutputStream` involves no translation, and the Unicode character is written as is. When the character is read from the storage medium or communication channel, it is not necessary to be concerned about the problem of translation of the coding system.

However, as mentioned in your last reading, it is preferable to use the `writeUTF()` method instead of the methods `writeBytes()` and `writeChars()` because they do not indicate the numbers of written characters. Unless the number of units is understood or has been obtained or derived, it is troublesome to use these two methods to read characters.

By using the `writeUTF()` and `readUTF()` methods of `DataOutputStream` and `DataInputStream` respectively, the Unicode characters are encoded in a universal coding system, UTF-8. Therefore, calling the `readUTF()` method to read a `String` object that was written by method `writeUTF()` prevents you from being concerned about the problem of different coding systems.

The `ObjectOutputStream` and `ObjectInputStream` classes

In the previous subsection, primitive values and `String` objects can be sent to a `DataOutputStream` object or can be read from a `DataInputStream` object. In some instances, it is preferable to handle input/output operations in units of objects. That is, an object is sent to a destination stream for storage or read/reconstituted from a source stream in a single operation.

For example, the class `Staff` defines two attributes, `name` and `basicSalary`.

```
// Definition of abstract class Staff
public abstract class Staff implements Serializable {
    // Attributes
    private String name;
    private double basicSalary;

    ......
}
```

(The extra clause `implements Serializable` is mandatory here.)

If it is necessary to write the data possessed by a `Staff` object to a destination stream, a possible way is to use a `DataOutputStream` object and call its `writeUTF()` and `writeDouble()` method with supplementary data of the attribute values. It is tedious and error-prone to perform input/output operations with respect to each attribute, especially for objects with plenty of attributes. Even when the object attributes are stored in data storage, it is up to the program that reads the file to interpret its contents. For example, the program may misinterpret that the `String` object and the `double` value are the account name and the balance respectively of a bank account. Therefore, the Java standard software library provides two classes, `ObjectOutputStream` and `ObjectInputStream` for handling writing and reading objects.

Please use the following reading to learn how to perform input/output operations with respect to objects by using the `ObjectOutputStream` and `ObjectInputStream` class.

> ***Reading***
>
> King, section 14.7, 'Reading and writing objects', pp. 636–41

Like the `DataInputStream` and `DataOutputStream` objects, the behaviours of `ObjectInputStream` and `ObjectOutputStream` objects are quite similar, except the conversions are between objects and bytes instead of between primitive values and bytes.

Whenever the `writeObject()` method of an `ObjectOutputStream` class is called with the supplementary data of the reference of an object, the supplied object is converted into a sequence of bytes to be written to the data destination (an `OutputStream` object). As the parameter type of the `writeObject()` method is `Object` and `Object` is the parent class of all classes in the Java programming language, the method can accept a reference of any object.

However, whenever the `readObject()` method of an `ObjectInputStream` object is called, the `ObjectInputStream` object reads bytes from its associated data source (an `InputStream` object) and the corresponding object is reconstructed and returned. The return type of `readObject()` method is `Object` and it is usually necessary to use casting before the reference is assigned to a reference variable of a suitable type other than `Object`. For example, if the object read from the data source is a `String` object, the following statement is required to assign the reference to the `String` object to a variable of type `String`,

```
String message = (String) in.readObject();
```

The variable `in` refers to an `ObjectInputStream` object.

The operations behind `ObjectOutputStream` and `ObjectInputStream` objects are visualized in Figure 8.78.

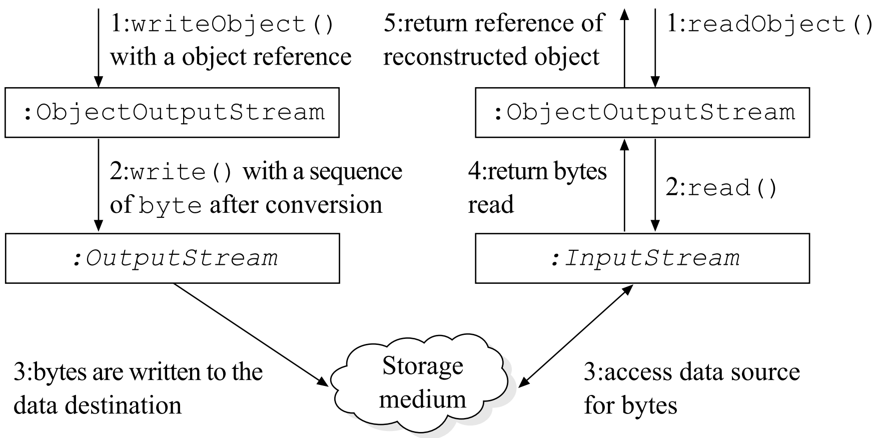


Figure 8.78 Operations behind `ObjectOutputStream` and `ObjectInputStream` objects

An advantage of the using `ObjectOutputStream` and `ObjectInputStream` is that it is not necessary to handle each the attributes of each object. Furthermore, if the type of object attribute is not primitive, it stores a reference to another object. Then, when such an object is sent to an `ObjectOutputStream` for writing to a data destination, all associated objects are written to the associated data destination of the `ObjectOutputStream`. Conversely, when the object is read from an `ObjectInputStream`, the object with all associated objects is reconstructed and returned.

Self-test 8.8

Write a DateWriter class with a single create() method that accepts a String object as the file name. Save a java.util.Date (or simply Date) object representing the current date/time to the specified file.

Write another DateReader class with a single show() method that accepts a String object as the file name and reads a Date object from the specified file to be shown on the screen.

Write suitable driver programs, TestDateWriter and TestDateReader that accept a program parameter of a file name to be supplied to the DateWriter create() method and DateReader show() method respectively.

Hint: Use the expression new Date() to create a Date object representing the current date/time.

Handling text files with BufferedReader and PrintWriter

Usually, we expect to handle a line in a text file as a String object one at a time. For example, for reading a line from a text file as a String object, a possible way to do so is to use a Reader object with a loop to read characters from a data source for concatenating a String object until the end of a line is reached.

However, the execution of the TestBinaryFileViewer and TestTextFileViewer1 with openu.txt as the program parameter disclosed that the combination of bytes with values 13 and 10 indicates the end of a line in the file. Therefore, the condition of the loop for reading a line character by character has to detect the combination of these two special bytes as the end of line. For writing characters to a text file, if you want to indicate the subsequent characters to be written to the file start on the next line, it is necessary to write two bytes with values 13 and 10 (or two characters '\n' and '\r') to the file.

It is even more troublesome that for most UNIX platforms, a line in a text file is terminated (or delimited) by a single byte of value 10 (or the character '\r') only. The implication is that if you are using Reader/Writer objects for performing reading/writing operations with characters, the programs have to determine whether the combination of characters '\n' and '\r', or a single '\r' character is used as line delimiters according to the operating system in which the programs are executing.

Reading text files with `BufferedReader` objects

Fortunately, the Java software library provides classes to help us resolve the problem. With the class `java.io.BufferedReader`, it is possible to create a `BufferedReader` object by supplying the reference of a `Reader` object, and the `Reader` object is treated as the data source of the created `BufferedReader` object. The `BufferedReader` class provides a `readLine()` method that calls the `read()` method of the associated `Reader` object to read sufficient characters until the end of a line and the characters in the line are concatenated to obtain a `String` object to be returned. The `BufferedReader` object transparently handles the line delimiter for various platforms. The `readLine()` method of the `BufferedReader` class returns `null` if all data available by the `Reader` have been read.

As the `Reader` class is abstract, a `BufferedReader` object must be associated with an object of the concrete subclass of `Reader`. A typical concrete subclass is a `FileReader` object. The above scenario is visualized in Figure 8.79.

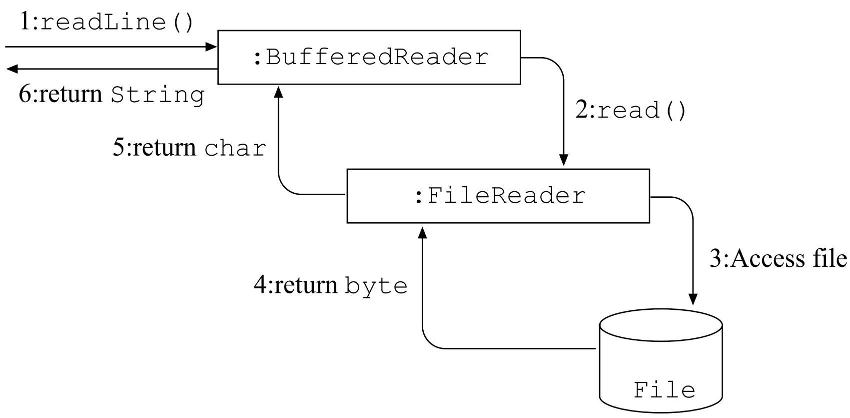


Figure 8.79 The operation sequences behind calling the `readLine()` method of a `BufferedReader` object with the associated `FileReader`

First of all, a `FileReader` object is created and is associated with a physical file on the disk. Then, the `FileReader` object is supplied to the constructor of the `BufferedReader` class for creating a `BufferedReader` so that the `BufferedReader` is associating with the `FileReader` and hence the physical file. The steps in reading a line from the file as a `String` object are:

1 Your program calls the `readLine()` method of a `BufferedReader` object for reading a line, which is returned from the associated physical file as a `String` object.

2 The `BufferedReader` object needs some characters to compose the `String` object; it therefore calls the `read()` method of the associated `Reader` object. The `BufferedReader` object may call the `read()` method of the associated `Reader` object repeatedly.

3 As the `read()` method of the `FileReader` object is called, it accesses the physical file to retrieve the bytes from it.

4 The bytes retrieved from the file are returned to the `Reader` object.

5 Based on the default coding system of the operating system, the bytes obtained from the file are translated to Unicode characters and are returned as the return values of the `read()` method of the `FileReader` object.

6 Based on the sequence of characters obtained by calling the `read()` method of the `FileReader` object, the `BufferedReader` object identifies the pattern of the end of line in the sequence of characters and returns the sequence of characters preceding the end of line pattern as a `String` object as the return value of the `readLine()` method.

The scenario of the application of `BufferedReader` and `FileReader` is termed a *layered approach*. It is a software design methodology that every object is dedicated to a particular operation and they cooperate to complete a task. For example, the `FileReader` is dedicated to reading data as bytes from the file and translating them from bytes to characters, whereas the `BufferedReader` object converts a sequence of characters to a `String` object.

The sequence of operations mentioned above is implemented in the `FileLister` class shown in Figure 8.80.

```
// Resolve classes in the java.io package

import java.io.*;

// Definition of class FileLister
public class FileLister {

    public void show(String name) throws IOException {
        // Create a File object that refers to the file
        File file = new File(name);

        // Create a FileReader object specified by the File
        // object and is treated as a general Reader object
        Reader reader = new FileReader(file);

        // Create a BufferedReader object specified by the
        // Reader object
        BufferedReader br = new BufferedReader(reader);

        // Declare a variable for storing the line read
        String lineRead;

        // The loop for reading each line from the file
        while ((lineRead = br.readLine()) != null) {
            // If the line read is not null, print it
            System.out.println(file.getName() + ":" + lineRead);
```

```
        }

        // Close the BufferedReader and hence all related
        // resources
        br.close();
    }
}
```

Figure 8.80 FileLister.java

The corresponding driver program is written as `TestFileLister` shown in Figure 8.73.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestFileLister
public class TestFileLister {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println("Usage : java FileLister <file>");
        } else {
            // Create a FileLister object
            FileLister lister = new FileLister();

            // Show the contents of the file
            lister.show(args[0]);
        }
    }
}
```

Figure 8. 81 TestFileLister.java

Compile the classes and execute the `TestFileLister` with a program parameter of `FileLister.java.` The contents of the file `FileLister.java` are shown on the screen. That is, execute the program with the following command:

`java TestFileLister FileLister.java`

The following output is shown on the screen:

```
FileLister.java: // Resolve classes in the java.io package
FileLister.java: import java.io.*;
FileLister.java:
FileLister.java: // Definition of class FileLister
FileLister.java: public class FileLister {
FileLister.java:
FileLister.java:    public void show(String name) throws IOException {
FileLister.java:        // Create a File object that refers to the file
```

```
FileLister.java:        File file = new File(name);
FileLister.java:
FileLister.java:        // Create a FileReader object specified by the File
FileLister.java:        // object and is treated as a general Reader object
FileLister.java:        Reader reader = new FileReader(file);
FileLister.java:
FileLister.java:        // Create a BufferedReader object specified by the
FileLister.java:        // Reader object
FileLister.java:        BufferedReader br = new BufferedReader(reader);
FileLister.java:
FileLister.java:        // Declare a variable for storing the line read
FileLister.java:        String lineRead;
FileLister.java:
FileLister.java:        // The loop for reading each line from the file
FileLister.java:        while ((lineRead = br.readLine()) != null) {
FileLister.java:           // If the line read is not null, print it
FileLister.java:        System.out.println(file.getName() + ":" + lineRead);
FileLister.java:        }
FileLister.java:
FileLister.java:        // Close the BufferedReader and hence all related
FileLister.java:        // resources
FileLister.java:        br.close();
FileLister.java:    }
FileLister.java: }
```

The source code file of the class `FileLister` is a text file. Therefore, it is possible to read the contents of the file a line at a time and show them on the screen.

A text file is a good way for storing data to be processed by a software application, because it is possible to use a text editor, such as Notepad of the Windows family, to edit its contents. For example, some settings or raw data that a software application need can be stored in text files, and the software application can read them to retrieve the necessary settings or raw data for processing.

For example, a class `SortIntegersInFile` is written in Figure 8.82. It reads a text file with an integer in each line and uses an `InsertionSorter` object (discussed in *Unit 6*) to sort the integers. Finally, the numbers are shown on the screen.

```
// Resolve classes in the java.io package
import java.io.*;

// Definition of class SortIntegersInFile
public class SortIntegersInFile {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage : java SortIntegersInFile <file>");
        }
```

```
        else {
            // Create a File object that refers to the file
            File file = new File(args[0]);

            // Create a FileReader object specified by the File
            // object and is treated as a general Reader object
            Reader reader = new FileReader(file);

            // Create a BufferedReader object specified by the
            // Reader object
            BufferedReader br = new BufferedReader(reader);

            // Create a InsertionSorter object for sorting
            InsertionSorter sorter = new InsertionSorter();

            // Declare a variable for storing the line read
            String lineRead;

            // The loop for reading each line from the file
            while ((lineRead = br.readLine()) != null) {
                // If the line read is not null, translate it
                // into integer and store it to the InsertionSorter
                sorter.storeNumber(Integer.parseInt(lineRead));
            }

            // Close the BufferedReader and hence all related
            // resources
            br.close();

            // sort the integers
            sorter.sort();

            // Get the array object of the sorter after sorting
            // and print the numbers
            int[] numbers = sorter.getNumbers();
            int total = sorter.getTotal();
            for (int i=0; i < total; i++) {
                System.out.print(numbers[i] + "\t");
            }
        }
    }
}
```

Figure 8.82 SortIntegersInFile.java

Once you can read the contents from a text file, you can convert the contents and process them any way you want. For example, the program `SortIntegersInFile` converts the `String` objects to equivalent values of type `int` to be supplied to the `InsertionSorter` object for sorting.

To test the `SortIntegersInFile` class, a text file `numbers.txt` is prepared using Notepad with contents as shown in Figure 8.83.

```
100
20
50
79
234
23
53
0
```

Figure 8.83 numbers.txt

Execute the `SortIntegersInFile` with program parameter `numbers.txt`, that is:

`java SortIntegersInFile numbers.txt`

The following output is shown on the screen:

```
0       20      23      50      53      79      100     234
```

It is more flexible to store the data to be processed by a software application in a text file and a software application reads it for the raw data at runtime. Please use the following self-test to experiment with using the `BufferedReader` with `FileReader` for searching for a number in the supplied text file.

Self-test 8.9

Based on the `FileLister` class or otherwise, write a class `SearchIntegerInFile` that accepts a file name and an integer as the program parameters. For example, the command

`java SearchIntegerInFile numbers.txt 23`

supplies the program the file name `numbers.txt` and an integer `23`. In the supplied file, every line stores an integer, and the `SearchIntegerInFile` searches the supplied file to determine whether the supplied integer, `23` in the above example, is found in the file. If the supplied integer is found, a message that looks like

```
The number 23 is found in line #6 of numbers.txt.
```

should be shown on the screen. Otherwise, a message that looks like the following should be shown:

```
The number 23 is not found in numbers.txt
```

We have discussed how to read and process text files with the `FileReader` and `BufferedReader` objects. Now, we are going to discuss how to write the contents to a text file so that we can use a common editor, such as Notepad, to read the contents.

Creating text files with `PrintWriter` objects

To write some textual data to a file so that a common text editor can read the file, the data must be written in a textual representation format and the corresponding bytes are encoded according to the default coding of the machine.

We said that we could use a `FileWriter` object to output character data to a file. However, preparing the sequence of characters to be supplied to the `Writer` object is tedious. For example, it is tedious to convert a value of type `double` into a textual representation as a sequence character. Furthermore, the issue of different line delimiters for different platforms further complicates the process. Therefore, we usually do not use the `Writer` object directly. Instead, we will use a `java.io.PrintWriter` (or simply `PrintWriter`) object that is associated with a `Writer` object.

The `PrintWriter` object accepts data in various structured formats (with its methods `print()` and `println()`) including all primitive types and non-primitive types, translates them into sequences of characters and calls the `write()` method of the associated `Writer` object to write the characters. Figure 8.84 may help you to visualize the scenario.

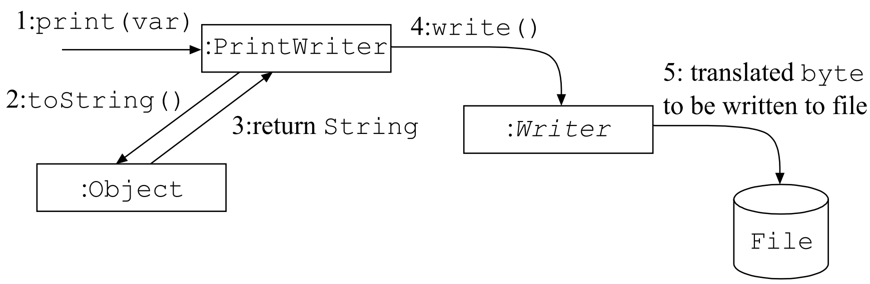


Figure 8.84 The operation sequences behind calling the `print()` method of a `PrintWriter` with the associated `Writer`

(The class name `Writer` is set in italics to highlight it is an abstract class and the object is a concrete subclass object of the `Writer` class, such as a `FileWriter` object.)

The steps in writing data to a physical file in textual formats are:

1 The `print()` method (or `println()` method) of the `PrintWriter` is called with supplementary data supplied. The `PrinterWriter` class defines nine overloaded `print()` methods and ten overloaded `println()` methods that can accept various primitive types and common non-primitive types — `char[]`, `String` and `Object`.

`print()`methods	`println()`methods

`public void print(boolean b)`
`public void print(char c)`
`public void print(char[] s)`
`public void print(double d)`
`public void print(float f)`
`public void print(int i)`
`public void print(long l)`
`public void print(Object obj)`
`public void print(String s)`
	`public void println(boolean b)`
	`public void println(char c)`
	`public void println(char[] s)`
	`public void println(double d)`
	`public void println(float f)`
	`public void println(int i)`
	`public void println(long l)`
	`public void println(Object obj)`
	`public void println(String s)`
	`public void println()`

The difference between the `print()` method and the `println()` method is that the `println()` method appends a character sequence of the default line delimiter of the operating system to the sequence of characters of the textual representation of the data to be sent to the associated `Writer` object, so that the subsequent written characters are considered written in the next line. In short, the `println()` method writes the supplied data in the current line and makes sure that subsequent data to be written to the data destination start with a new line. A `println()` method with an empty parameter list solely sends the character sequence of the default line delimiter to the associated `Writer` object, just to make sure the subsequent written characters start with a new line.

1 The data supplied to the `print()` or `println()` method will be converted into a proper textual representation in a sequence of characters.

2 If the type of supplementary data supplied to the `print()` or `println()` method called is neither a primitive type, `String` nor `char[]`, the one with parameter type `Object` is called and the `toString()` method of the supplied `Object` object is called to obtain the textual representation. Converting primitive values to their corresponding textual representations is discussed in *Unit 10*.

3 A `String` object that contains the textual representation of the supplied `Object` object is returned to the `PrintWriter` object.

4 The sequence of characters is supplied to the associated `Writer` object of the `PrintWriter` object. It will translate the characters into a sequence of bytes in the default coding system of the operating system.

5 The sequence of translated bytes is finally written to the physical file on a disk.

To test the use of `PrintWriter` class, a class `TestPrintWriter` is written in Figure 8.85.

```
// Resolve classes in the java.io package
import java.io.*;
// Resolve Date class
import java.util.Date;

public class TestPrintWriter {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println("Usage: java TestPrintWriter <file>");
        }
        else {
            // Create a File object that refers to the physical file
            File file = new File(args[0]);

            // Create a FileWriter object specified by the File
            // object and is treated as a general Writer object
            Writer writer = new FileWriter(file);

            // Create a PrintWriter object that associated with the
            // Writer object
            PrintWriter out = new PrintWriter(writer);

            // Prepare an array object with element char
            char[] word = {'H', 'e', 'l', 'l', 'o'};

            // Use the println() method of the PrintWriter to write
            // the data to the ultimate file
            out.println(true); // boolean
            out.println('A'); // char
            out.println(word); // char[]
            out.println(3.1415); // double
            out.println(0.0F); // float
            out.println(100); // int
            out.println(2L); // long
            out.println(new Date()); // Object (a Date object)
            out.println("Open University"); // String

            // Close the reader and release all corresponding resources
            out.close();
        }
    }
}
```

Figure 8.85 TestPrintWriter.java

Compile and execute the `TestPrintWriter` class with a program parameter, say `pw.txt`:

```
java TestPrintWriter pw.txt
```

The specified file, pw.txt, will be created. View the file with an editor, such as Notepad. You will find the contents shown as in Figure 8.86.

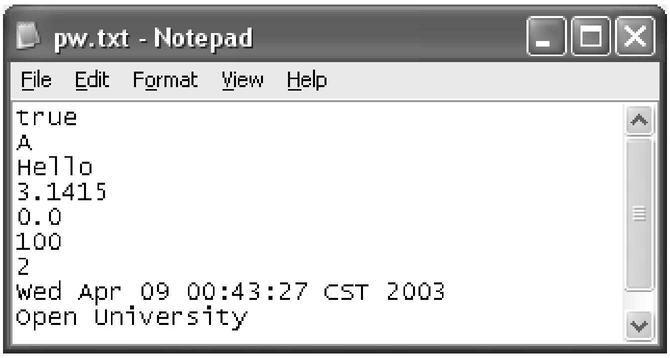


Figure 8.86 The contents of the file, pw.txt, created by executing the `TestPrintWriter` program

The `print()` and `println()` methods of a `PrintWriter` object enable us to output the data to a text file as if we are showing messages on the screen. You should notice that for the eighth `println()` method in the `main()` method of `TestPrintWriter` class, a `Date` object is created with no parameter. By default, it represents the current date/time. Therefore, the `println()` method calls the `toString()` method of the `Date` object to obtain the textual representation and writes the sequence of characters to the file. If you execute the `TestPrintWriter` program on your computer, the date/time stored in the text file is the one when you execute the program.

Since the `println()` method of the `PrintWriter` object is used to output the data by the `TestPrintWriter` program, the default line delimiter of the platform is appended at the end of each set of data. Therefore, you are freed from handling the issue of different line delimiters for different platforms.

Up to now, you have learned many classes for reading and writing data in various formats. By making use of the classes introduced, it is possible to get a web page from the Internet to be shown on the screen or written to a file. Please refer to Appendix C for a sample implementation.

Please use the following self-test to verify your understanding on the use of `PrintWriter` class.

Self-test 8.10

Enhance the `PayrollCalculator1` class discussed in *Unit 7* by modifying the method `showReport()` to be

```
public void showReport(String filename) {
    ......
}
```

so that the payroll report is written to the file specified. Modify the `TestPayrollCalculator1` class as well so that the first program parameter is supplied to the `showReport()` method of the `PayrollCalculator1` object.

Filter streams for buffering

The operations of reading and writing physical files are comparatively slower than reading and storing data in the memory. Therefore, it is preferable to write a large block of data to a file in a single output operation instead writing data to it byte by byte. The memory block for storing the temporary data is usually known as a buffer, and the technique of speeding up the reading/writing with buffers is known as *buffering*.

When performing input/output operations with a Java program, it is possible to implement kinds of buffering to improve operation efficiency by adding filter stream objects for buffering capability.

Improving input/output efficiency with buffering

The definition of class `CopyFile1` reads the source file byte by byte and each byte read is written to the destination file one byte at a time. In order to improve the performance of copying a file, it is possible to read a large block of data from the source file; the block of data is then written to the destination. The class `CopyFile` presented in the textbook (pp. 617–18) takes such an approach. The Java software library provides two classes `BufferedInputStream` and `BufferedOutputStream` that feature buffering capability.

By studying the documentation of `BufferedInputStream` and `BufferedOutputStream`, you can see that they are subclasses of `FilteredInputStream` and `FilteredOutputStream` respectively. Furthermore, `FilteredInputStream` and `FilteredOutputStream` are subclasses of `InputStream` and `OutputStream` respectively. The relationships among the classes are visualized in Figure 8.87.

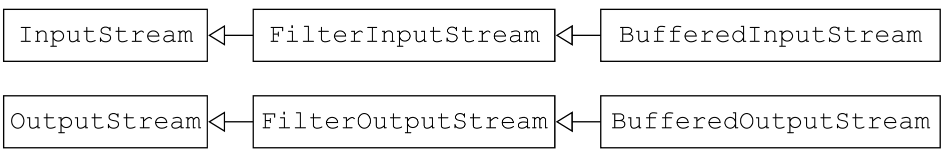


Figure 8.87 The relationship among the Filter streams and `InputStream/OutputStream`

Since `BufferedInputStream` and `BufferedOutputStream` are subclasses of the `InputStream` and `OutputStream` respectively, they can be used as if they are general `InputStream` and `OutputStream` objects. With filter streams `BufferedInputStream` and

`BufferedOutputStream`, the definition of class `CopyFile1` is enhanced to become the `CopyFile2` class shown in Figure 8.88.

```
// Resolve classes in the java.io package

import java.io.*;

// Definition of class CopyFile2
public class CopyFile2 {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if two files are specified as program parameters
        if (args.length < 2) {
            // If not two files are specified, show usage information
            System.out.println(
                "Usage: java CopyFile2 <source> <destination>");
        }
        else {
            // Create File objects to refer to the physical files
            File inFile = new File(args[0]);
            File outFile = new File(args[1]);

            // Create InputStream/OutputStream that associate to the
            // two File objects and hence the physical files
            InputStream in = new FileInputStream(inFile);
            OutputStream out = new FileOutputStream(outFile);

            // Create BufferedInputStream and BufferedOutputStream
            // objects that associate to FileInputStream and
            // OutputStream objects
            InputStream bin = new BufferedInputStream(in);
            OutputStream bout = new BufferedOutputStream(out);

            // Create a StreamCopier object
            StreamCopier copier = new StreamCopier();

            // Copy the contents from the InputStream object to the
            // OutputStream object
            copier.copy(bin, bout);

            // Close the InputStream/OutputStream and release all
            // related resources
            bin.close();
            bout.close();
        }
    }
}
```

Figure 8.88 CopyFile2.java

Compared with the definition of class `CopyFile1`, two filter stream objects of classes `BufferedInputStream` and `BufferedOutputStream` are created that are associated with the

`FileInputStream` and `FileOutputStream` objects. The two filter buffer stream objects are used as if they are general `InputStream` and `OutputStream` objects. The rest of the definition of class `CopyFile2` is therefore similar to that of class `CopyFile1`. The sequence of data flow can be visualized in Figure 8.89.

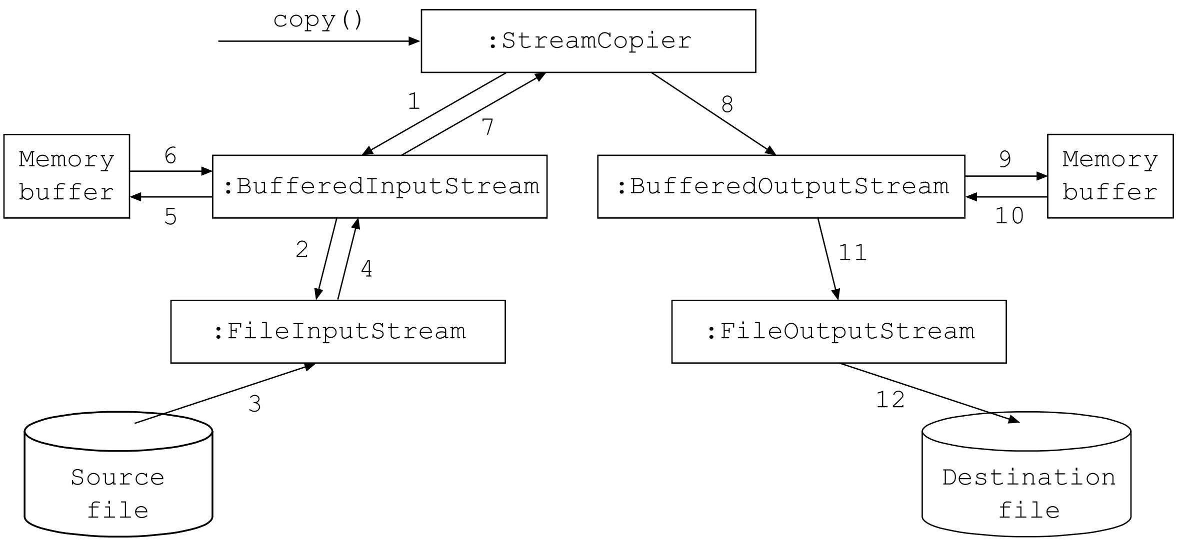


Figure 8.89 The operation sequences behind using `FileInputStream` / `FileOutputStream` objects with `BufferedInputStream` / `BufferedOutputStream` objects

The following describes the events numbered in Figure 8.89.

1 The `copy()` method of the class `StreamCopier` calls the `read()` method of the `BufferedInputStream` object as if it is a general `InputStream` object for getting a byte.

2 The `BufferedInputStream` object calls the `read()` method of associated `InputStream` object (actually a `FileInputStream` object) for reading a large block of data.

3 The `FileInputStream` object receives the message `read` from the `BufferedInputStream` object, and it reads a large block of data from the file.

4 The block of data in units of byte is returned to the `BufferedInputStream` object.

5 The block of data returned from the `FileInputStream` object is stored in a memory block that is accessible by the `BufferedInputStream` object.

6 A byte is read from the memory buffer to be returned by the `BufferedInputStream` object.

7 The `BufferedInputStream` object returns a byte as the return value of the method `read()`.

8 The `copy()` method calls the `write()` method of the `BufferedOutputStream` object with a supplied byte as if it is a general `OutputStream` object.

9 The `BufferedOutputStream` object stores the byte to a block of memory used as a buffer for writing data. The execution of the method `write()` completes and returns immediately.

10 Whenever necessary, for example the memory buffer for writing is full, the `BufferedOutputStream` object retrieves the bytes in the memory blocks.

11 The `write()` method of the associated `OutputStream` (actually a `FileOutputStream`) object is called with the data in the memory block.

12 The data received are permanently written to the associated physical file.

If the `copy()` method calls the `read()` method of the `BufferedInputStream` again, the byte to be returned is retrieved from the memory buffer and no physical file operation is involved. Therefore, the efficiency of reading a byte from the file is greatly improved. But, when the `copy()` method calls the `write()` method of the `BufferedOutputStream` again, the byte supplied to the object is stored in the memory buffer for most of the time, and it similarly involves no physical file operation. Whenever the buffer is full or the file is to be closed, the data in the memory buffer are written to the physical file by the `FileOutputStream` object.

Four core objects are involved when executing the `CopyFile2` program: `FileInputStream`, `BufferedInputStream`, `FileOutputStream` and `BufferedOutputStream`. When the process of copying the bytes from the source file to the destination file completes, it is necessary to call the `close()` method of these objects to release the resources and the data are properly stored in the destination file. It is not necessary to call the `close()` methods of all four objects, but instead to call the `close()` methods of the `BufferedInputStream` and `BufferedOutputStream` objects. They will call the `close()` method of the associated `InputStream` (actually a `FileInputStream`) object and `OutputStream` (actually `FileOutputStream`) object respectively and release the resources they acquired.

Appendix E: more on standard input and output

Reading characters from a keyboard

As the line of entry is textual, it is preferable to handle the line of entry as a sequence of characters. It is especially important if the platform enables an input method for double-byte characters, such as Chinese characters. Then, a translation between bytes and the default coding system of the platform is required. Therefore, we can use an `InputStreamReader` object to translate the bytes obtained from the line of entry into a sequence of Unicode characters.

The class `Keyboard2` shown in Figure 8.90 is based on `Keyboard1` — the bytes read from the `InputStream` object that is attached to the computer keyboard are consumed by an `InputStreamReader` object. Then, the `InputStreamReader` object, which is a character input stream, returns Unicode characters after translation.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class Keyboard2
public class Keyboard2 {
    // Constant for end of file
    private static final int EOF = -1;

    // Show the entries from keyboard
    public void show() throws IOException {
        // Create an InputStreamReader that associated with the
        // InputStream for the keyboard
        Reader reader = new InputStreamReader(System.in);

        int charCount = 0; // The count of character read
        int charRead; // The char read from keyboard
        // A while loop to read a char from the keyboard
        while ((charRead = reader.read()) != EOF) {
            // Increase the character count
            charCount++;
            // Show the value of the character read
            System.out.print("Character (#" + charCount +
                ") = " + charRead);

            // If the character read is not a special character,
            // show it. Otherwise, just skip to next line
            if (charRead >= 32) {
                System.out.println(" ('" + (char) charRead + "')");
            } else {
                System.out.println();
            }
        }
    }
}
```

Figure 8.90 Keyboard2.java

The `InputStreamReader` object in the `show()` method of `Keyboard2` is created without supplying a `String` for the coding system involved in translation. The default coding system of the platform is therefore used.

When the `read()` method of the `InputStreamReader` object is called, the object will call the `read()` method of the associated `InputStream` object, which is attached to the computer keyboard, to get a sequence of bytes. Once the `InputStream` returns the bytes to the `InputStreamReader` object, the `InputStreamReader` object translates the bytes with respect to the default encoding system of the platform to Unicode characters. Finally, the `read()` method of the `InputStreamReader` object returns the translated Unicode characters.

A driver program `TestKeyboard2` is written for testing the `Keyboard2` class. As it is similar to the `TestKeyboard1` class, except that the object to be created is `Keyboard2` instead of `Keyboard1`, the definition is not shown here. Please refer to the course CD-ROM or website. When you compile the classes and execute the `TestKeyboard2` class, the same message is shown on the screen:

```
Enter Ctrl-Z/Ctrl-D to terminate the program
```

Then, you can use it like the `TestKeyboard1` class and enter a line of entry with the keyboard, such as:

```
Enter Ctrl-Z/Ctrl-D to terminate the program
abcdefg
```

The program will show the following output message:

```
Enter Ctrl-Z/Ctrl-D to terminate the program
abcdefg
Character (#1) = 97 ('a')
Character (#2) = 98 ('b')
Character (#3) = 99 ('c')
Character (#4) = 100 ('d')
Character (#5) = 101 ('e')
Character (#6) = 102 ('f')
Character (#7) = 103 ('g')
Character (#8) = 13
Character (#9) = 10
```

The bytes obtained from the keyboard are now translated into the corresponding Unicode characters. If the operating system of your computer enables you to enter Chinese characters in the Command Prompt, you will find out that the two bytes that correspond to a Chinese character are translated into a single Chinese character and displayed.

Reading lines from a keyboard as `String` objects

Even though we can now read a sequence of characters from the keyboard, it is preferable to obtain a `String` object that corresponds to a line of entry from the keyboard. Of course, we can use a `while` loop to read the characters until the pattern for end of line is reached. Then, we face the same problem that the patterns for end of line for different platforms are different. Like handling a text file, we can use a `BufferedReader` object to help us to resolve the issue.

The sequence of characters returned by the `InputStreamReader` is processed by a `BufferedReader` object so that a sequence of characters is used to construct a `String` object to be returned by the `readLine()` method of the `BufferedReader` object.

Based on `Keyboard2`, a class `Keyboard3` is written in Figure 8.91 using a `BufferedReader` object that is associated with an `InputStreamReader.` The `InputStreamReader` is associated with the `InputStream` attached to the keyboard.

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class Keyboard3
public class Keyboard3 {

    // Show the entries from keyboard
    public void show() throws IOException {
        // Create an InputStreamReader that associates with the
        // InputStream for the keyboard
        Reader reader = new InputStreamReader(System.in);

        // Create a BufferedReader that associates with the
        // InputStreamReader and subsequently associates with
        // the keyboard
        BufferedReader in = new BufferedReader(reader);

        int lineCount = 0; // The count of character read
        String lineRead; // The char read from keyboard
        // A while loop to read a char from the keyboard
        while ((lineRead = in.readLine()) != null) {
            // Increase the character count
            lineCount++;
            // Show the value of the character read
            System.out.println("Line (#" + lineCount +
                ") = " + lineRead);
        }
    }
}
```

Figure 8.91 Keyboard3.java

The sequence of operations behind `Keyboard3` can be visualized in Figure 8.92.

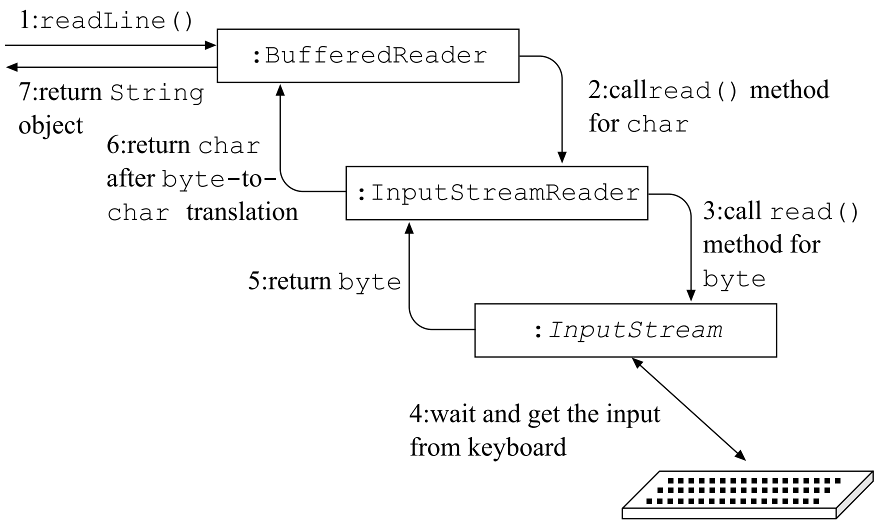


Figure 8.92 The sequence of operations behind calling the `readLine()` method for getting a `String` from the keyboard

The statements for creating the `BufferedReader` object can be combined into a single statement for simplicity:

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(System.in));
```

Please refer to the course CD-ROM or website for the driver program, `TestKeyboard3`. Compile the classes `Keyboard3` and `TestKeyboard3`, and execute the `TestKeyboard3` class. Whatever you type in a line of entry that is terminated by pressing the <Enter/Return> key is shown on the screen as a `String`, for example:

```
java TestKeyboard3
Enter Ctrl-Z/Ctrl-D to terminate the program
abcdefg
Line (#1) = abcdefg
hijklmn
Line (#2) = hijklmn
```

The previous reading mentions that operating systems allow the user to redirect the standard input stream so that the data returned from the `InputStream` object referred by `System.in` come from a file instead. We can experiment with such a redirection by the following command:

```
java TestKeyboard3 < pw.txt
```

The bytes returned by the `InputStream` object referred by the class variable `System.in` come from the file `pw.txt`. Therefore, the following output is shown on the screen:

```
Line (#1) = true
Line (#2) = A
Line (#3) = Hello
Line (#4) = 3.1415
Line (#5) = 0.0
Line (#6) = 100
```

```
Line (#7) = 2
Line (#8) = Wed Apr 09 00:43:27 CST 2003
Line (#9) = Open University
```

After the `InputStream` object referred by `System.in reads` all lines in the file, the end of file is reached and the program terminates.

With the knowledge of reading a line of entry as a `String` object, we can further manipulate the `String` object, such as converting the `String` object that has been read to a value of primitive types like `int` and `double`.

A class `Console` is written in Figure 8.93 to ease reading data from the keyboard. Like the definition of the class `TestKeyboard3`, a `BufferedReader` object is prepared that associates with an `InputStreamReader` object that subsequently associates with the `InputStream` object for the computer keyboard. In different methods, the `String` object returned by the `BufferedReader` object is converted into the desired data.

```
// Import statement for resolving classes in java.io package
import java.io.*;

// Definition of class Console
public class Console {
    // Create a BufferedReader which indirectly attaches the keyboard
    private static BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    // Show the supplied prompt to the screen
    private static void showPrompt(String prompt) {
        // If the supplied prompt is not null, show it
        if (prompt != null) {
            System.out.print(prompt);
        }
    }

    // Read a line from the keyboard and return as a String object
    private static String readStringFromKeyboard() {
        String input = null;
        try {
            input = in.readLine();
        } catch (IOException ioe) {
            // It is almost impossible for causing an
            // IOException, but the catch block is needed
            System.out.println("Unexpected I/O error");
        }
        return input;
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of int
    public static int readInt(String prompt) {
        showPrompt(prompt);
        return Integer.parseInt(readStringFromKeyboard());
    }
```

```
    // Read a line from the keyboard and return as a data of int
    public static int readInt() {
        return readInt(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of long
    public static long readLong(String prompt) {
        showPrompt(prompt);
        return Long.parseLong(readStringFromKeyboard());
    }

    // Read a line from the keyboard and return as a data of long
    public static long readLong() {
        return readLong(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of double
    public static double readDouble(String prompt) {
        showPrompt(prompt);
        return Double.parseDouble(readStringFromKeyboard());
    }

    // Read a line from the keyboard and return as a data of double
    public static double readDouble() {
        return readDouble(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of float
    public static float readFloat(String prompt) {
        showPrompt(prompt);
        return Float.parseFloat(readStringFromKeyboard());
    }

    // Read a line from the keyboard and return as a data of float
    public static float readFloat() {
        return readFloat(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return the String object right away
    public static String readString(String prompt) {
        showPrompt(prompt);
        return readStringFromKeyboard();
    }

    // Read a line from the keyboard and return it right away
    public static String readString() {
        return readString(null);
    }
}
```

Figure 8.93 Console.java

As the methods defined in the `Console` class are all utility methods and they access no object attribute, they are defined as class methods so that the methods can be called without creating an object of the class. For example, to read a value of `int` from the keyboard, a statement that looks like the following can be used:

```
int num = Console.readInt("Input a number");
```

Two methods `showPrompt()` and `readStringFromKeyboard()` are marked `private` because they are to be used by the methods in the same class only. The `readStringFromKeyboard()` method is called instead of calling the `readLine()` method of the `BufferedReader` object in each method, because calling the `readLine()` method may cause an exception. It is preferable to define a single method for exception handling instead of duplicating the `try/catch` block in every method.

The class defines methods enable the user to enter data of type `int`, `long`, `float`, `double` and `String` through the keyboard. There are two overloaded methods for each type of data. If the reference of a `String` object is supplied to the method, the `String` object is shown on the screen as a prompt.

For getting primitive type data, the line of entry obtained from the keyboard is converted into the equivalent values with the class methods, `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()` and `Double.parseDouble()` respectively. These classes are provided in the Java standard software library in the package `java.lang` and they are commonly known as *wrapper* classes for the corresponding primitive types. We discuss the uses of these classes in detail in *Unit 10*. If the type of data to be read is `String`, the `String` object obtained by the `readStringFromKeyboard()` method is returned right away.

Now, we can use the `Console` class to obtain data to be processed. For example, we can use the `readInt()` method of the `Console` class to get the integers to be sorted. A class `SortIntegersFromKeyboard1` is shown in Figure 8.94.

```
// Definition of class SortIntegersFromKeyboard1
public class SortIntegersFromKeyboard1 {

    // Main executive method
    public static void main(String args[]) {
        // Create an InsertionSorter object for sorting
        InsertionSorter sorter = new InsertionSorter();

        // Obtain the amount of numbers to be sorted
        int numTotal = Console.readInt(
            "Number of integers to be sorted : ");

        // Get a number from keyboard and store it to the
        // InsertionSorter object
        for (int i=1; i <= numTotal; i++) {
            // Get a number from keyboard
            int num = Console.readInt("Input number (#" + i + ") = ");
```

```
                // Store the number to the InsertionSorter object
                sorter.storeNumber(num);
            }

            // Sort the numbers
            sorter.sort();

            // Get the sorted numbers from the InsertionSorter object
            int[] sorted = sorter.getNumbers();

            // Show the sorted numbers
            System.out.println("Sorted numbers:");
            for (int i=0; i < numTotal; i++) {
                System.out.print(sorted[i] + "\t");
            }
        }
    }
```

Figure 8.94 SortIntegersFromKeyboard1.java

Compile the classes `Console` and `SortIntegersFromKeyboard1`, and execute the `SortIntegersFromKeyboard1` program. You will be prompted with the message

```
Number of integers to be sorted :
```

requesting you enter the amount of numbers to be entered to the program for sorting. For example, if you enter `5`, you will be prompted five more times for the numbers to be sorted. The `InsertionSorter` object stores the entered numbers. After all the numbers are entered and stored by the `InsertionSorter` object, the `InsertionSorter` object sorts the numbers and the sorted numbers are retrieved from the `InsertionSorter` object to be shown. A sample execution is:

```
Number of integers to be sorted : 5
Input number (#1) = 50
Input number (#2) = 80
Input number (#3) = 78
Input number (#4) = 23
Input number (#5) = 10
Sorted numbers:
10      23      50      78      80
```

The definition of the `SortIntegersFromKeyboard1` class `main()` method is quite straightforward. The core part of the program is the uses of the `Console` class that performs input operations with respect to the keyboard.

Please use the following self-test to enhance the `Console` class. Use the `Console` class for getting the coefficients to solve a quadratic equation.

Self-test 8.11

1 The methods defined in the `Console` class use the class methods `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()` and `Double.parseDouble()` for converting a `String` object to the corresponding primitive value. However, if the format of the `String` object contents is invalid, a `NumberFormatException` will occur.

Enhance the `Console` class by implementing proper exception handling so that the user is prompted with a warning message such as, `"Invalid numeric format. Please re-enter."` The user can re-enter the numbers until no exception occurs.

2 In *Unit 4*, you learned how to solve a quadratic equation with the class `QuadraticEquation`. The class `EquationSolver` is used as the driver class to use a `QuadraticEquation` object, and it shows dialog boxes for getting the coefficients.

Modify the `EquationSolver` class so that it uses the `Console` class for getting the coefficients of a quadratic equation from the keyboard instead.

Suggested answers to self-test questions

Self-test 8.1

DirUsage.java

```
// Resolve File, Arrays and Date classes
import java.io.File;
import java.util.Arrays;
import java.util.Date;

// Definition of class DirUsage
public class DirUsage {

    // Show the files in the supplied directory
    public void showFilesInDir(String dirName) {
        // Create a File object to refer to the directory
        File dir = new File(dirName);

        // Check if the File object is referring to a directory
        if (dir.isDirectory()) {
            // If the File object is referring to a directory

            // Get the filenames in the directory
            String[] filenames = dir.list();

            // Declare and initialize file count and total file sizes
            int fileCount = 0;
            long totalSize = 0L;

            // Process each file
            for (int i=0; i < filenames.length; i++) {
                File file = new File(dir, filenames[i]);
                if (file.isFile()) {
                    fileCount++;
                    totalSize += file.length();
                }
            }

            // Show a footer
            System.out.println(
                "\nThere are totally " + fileCount +
                " file(s) ("+ totalSize +
                " byte(s)) in the directory " + dirName + ".");
        }
        else {
            // Show the user the supplied directory is not a directory
            System.out.println("Sorry, the " + dirName +
                " is not referring to a directory.");
        }
    }
}
```

TestDirUsage.java

```
// Definiton of class TestDirUsage
public class TestDirUsage {

    // Main executive method
    public static void main(String args[]) {
        // Check if the user supplied the directory name as
        // program parameter
        if (args.length == 0) {
            // Show usage information
            System.out.println("Usage: java TestDirUsage <file>");
        }
        else {
            // Create the DirUsage object
            DirUsage lister = new DirUsage();
            // Supply the directory name to the DirUsage object
            // to show the files in the specified directory
            lister.showFilesInDir(args[0]);
        }
    }
}
```

Self-test 8.2

RandomFileCreator.java

```
// Resolve the classes in the java.io package
import java.io.*;

// Definition of class RandomFileCreator
public class RandomFileCreator {

    // Main executive method
    public void create(String name) throws IOException {
        // Create a File object that refers to the physical file
        File file = new File(name);

        // Create a FileOutputStream object specified by the File
        // object and is treated as a general OutputStream object
        OutputStream os = new FileOutputStream(file);

        // The file size is a random number in range 50 to 200
        // inclusive
        int fileSize = (int) (Math.random() * 151) + 50;

        // Write the data to the output stream
        for (int i=0; i < fileSize; i++) {
            os.write((int) (Math.random() * 128));
        }

        // Close the stream and release all corresponding resources
        os.close();
    }
}
```

TestRandomFileCreator.java

```
// Resolve class in package java.io.
import java.io.*;

// Definition of class TestRandomFileCreator
public class TestRandomFileCreator {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length == 0) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage: java TestRandomFileCreator <file>");
        }
        else {
            // Create a RandomFileCreator object
            RandomFileCreator creator = new RandomFileCreator();

            // Create the binary file
            creator.create(args[0]);
        }
    }
}
```

Self-test 8.3

URLGetter.java

```
// Resolve classes in the java.io package
import java.io.*;
import java.net.*;

// Definition of class URLGetter
public class URLGetter {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a URL and a file are specified as program parameters
        if (args.length < 2) {
            // If there are less than two program parameters,
            // show usage information
            System.out.println(
                "Usage: java URLGetter <URL> <file>");
        }
        else {
            // Create a URL object that refer to the URL resource
            URL url = new URL(args[0]);

            // Obtain the InputStream object from the URL object
            InputStream in = url.openStream();

            // Create File objects to refer to the target files
            File file = new File(args[1]);
```

```
            // Create InputStream/OutputStream that associate to the
            // two File objects and hence the physical files
            OutputStream out = new FileOutputStream(file);

            // Create a StreamCopier object
            StreamCopier copier = new StreamCopier();

            // Copy the contents from the InputStream object to the
            // OutputStream object
            copier.copy(in, out);

            // Close the InputStream/OutputStream and release all
            // related resources
            in.close();
            out.close();
        }
    }
}
```

Self-test 8.4

If you are not sure whether a file is binary or textual, you can use the `TestBinaryFileViewer` to show the contents of a file of the desired type. If the contents of the file shown contain mostly readable and meaningful characters, it probably is a textual file. Otherwise, it is usually a binary file.

The five file types are categorized into the following two types — binary and textual.

Binary: image files, Microsoft Word document files and Java compiled class files

Textual: Java source code file and web pages

Self-test 8.5

`NumberFormatException`

The cause of the `NumberFormatException` results from calling the `parseInt()` method with supplementary data of an empty `String` object or a value of `null`. Therefore, it is necessary to make sure that the supplementary data supplied to the `parseInt()` method are a non-empty `String` object.

A possible way to handle the problem in the `TestGreeting` class is to use a `while` loop to repeatedly prompt the user for the hour until the entry is neither a non-empty `String` object nor `null`.

TestGreeting.java

```
// Import statement for resolving the class JOptionPane
import javax.swing.JOptionPane;

// The class definition of TestGreeting for setting up the
// environment and test the GreetingChooser object
public class TestGreeting {

    // The main executive method
    public static void main(String args[]) {
        // Create the GreetingChooser object and use variable chooser
        // to refer to it
        GreetingChooser chooser = new GreetingChooser();

        // Show a dialog for getting the hour from user
        // The obtained hour is returned as a String object
        String inputHour;
        while ((inputHour =
                JOptionPane.showInputDialog("Please enter the hour"))
                == null || inputHour.length() == 0) {
            JOptionPane.showMessageDialog(
                null, "Invalid input. Please enter again.");
        }

        // Obtain the integer value from the input String object

        int hour = Integer.parseInt(inputHour);

        // Get the greeting from the GreetingChooser object
        String greeting = chooser.getGreeting(hour);

        // Determine the message to be shown
        String output = "Good " + greeting;

        // Show the greeting to the user with a dialog
        JOptionPane.showMessageDialog(null, output);

        // Terminate the program explicitly
        System.exit(0);
    }
}
```

ArrayIndexOutOfBoundsException

The `ArrayIndexOutOfBoundsException` occurs because the first element of the array object referred by the variable `args` is used without checking its size. To prevent the program from causing the exception, an `if` statement can be used and the first element of the array object is used only if there is at least one program parameter.

ShowFirstArgs.java

```
// Definition of class ShowFirstArgs
public class ShowFirstArgs {

    // Main executive method
    public static void main(String args[]) {
        if (args.length < 1) {
            System.out.println("Usage: java ShowFirstArgs <parameter>");
        }
        else {
            // Show the first argument maintained by the array object
            System.out.println("The first program parameter is [" +
                args[0] + "].");
        }
    }
}
```

ClassCastException

Casting operation should be carried out only if the reference is referring to an object of the target class of the casting operation. Therefore, it is preferable to enclose the casting operating in an `if` statement with `instanceof` operator as the condition.

TestCCException.java

```
// Definition of class TestCCException
public class TestCCException {

    // Main executive method
    public static void main(String args[]) {
        // Create an Object class object and is referred by a
        // variable of type Object
        Object obj = new Object();

        // Check if the object referred by the variable obj is
        // a String object
        if (obj instanceof String) {
            // Cast the reference to String type explicitly
            String str = (String) obj;
            // Show the textual representation of the object
            System.out.println(str);
        }
        else {
            // If the object referred by the variable obj is not
            // a string, show a message
            System.out.println("It is not a String object.");
        }
    }
}
```

`ArithmeticException`

This exception occurs as a result of division by a value of zero. With respect to the `FindAverage` program, if no program parameters are provided, it is unreasonable to find the average. Therefore, the program segment for finding the average should only be executed if program parameters are provided.

FindAverage.java

```
// Definition of class FindAverage
public class FindAverage {

    // Main executive method
    public static void main(String args[]) {
        // Check if at least one program parameter is provided
        if (args.length < 1) {
            // If no program parameter provided, prompt the user
            System.out.println(
                "Usage: java FindAverage <number> {<number>}");
        }
        else {
            // Calculate the average

            // Initialize variables for finding the average
            int sum = 0;
            int count = 0;

            // Iterate each number for finding the total
            for (int i=0; i < args.length; i++) {
                sum += Integer.parseInt(args[i]);
                count++;
            }

            // Show the average of the numbers on the screen
            System.out.println(
                "The average of the numbers is " + sum / count);
        }
    }
}
```

`NullPointerException`

This exception usually occurs because, if the random number obtained from the expression is `0`, the content of the variable `suitName` is kept to be `null`. This reveals that constants used in the `switch/case` structure do not match the random number obtained. To amend the program, it is necessary to modify either the expression for getting a random number or the constants used in the `switch/case` construct.

GuessACard.java

```
// Definition of class GuessACard
public class GuessACard {

    // Main executive method
    public static void main(String args[]) {
        // Verify the number of program parameter
        if (args.length < 1) {
            // If no program parameter is supplied, show usage message
            System.out.println(
                "Usage: java GuessACard <Spade|Heart|Club|Diamond>");
        }
        else {
            // Get a random number
            int suitDrawn = (int) (Math.random() * 4);

            // Derive a suit name according to the random number
            String suitName = null;
            switch (suitDrawn) {
                case 0:
                    suitName = "Spade";
                    break;
                case 1:
                    suitName = "Heart";
                    break;
                case 2:
                    suitName = "Club";
                    break;
                case 3:
                    suitName = "Diamond";
                    break;
            }

            // Check whether the program parameter is the same as
            // the suit drawn and show message accordingly
            if (suitName.equals(args[0])) {
                System.out.println(
                    "Yes, my card is also a " + suitName);
            }
            else {
                System.out.println("Sorry, my card is a " + suitName);
            }
        }
    }
}
```

Self-test 8.6

FileCreator6.java

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class FileCreator6
public class FileCreator6 {

    // Create a file according to the name and size specified by
    // the String array elements
    public void create(String[] args) {
        // Variable file has to be defined here so that it can be
        // accessed in the catch block
        File file = null;
        try {
            // Determine the number of KB to be written to the file
            int size = Integer.parseInt(args[1]) * 1024;

            // Create a File object that refers to the physical file
            file = new File(args[0]);

            // Create a FileOutputStream for writing bytes
            // to the file
            OutputStream out = new FileOutputStream(file);

            // Write the specifed number of bytes (with values 0)
            // in KB to the file
            for (int i = 0; i < size; i++) {
                out.write(0);
            }

            // Close the OutputStream object and release all related
            // resources
            out.close();
        } catch (NumberFormatException nfe) {
            System.out.println(
                "The second program parameter is not an integer.");
            System.out.println(
                "Please provide a valid integer as the file size.");
        } catch (FileNotFoundException fnfe) {
            System.out.println(
                "Failed in opening the file "
                    + file.getPath()
                    + " for writing.");
            System.out.println(
                "Please specify another file for writing.");
            System.out.println("Problem encountered: " +
                fnfe.getMessage());
        } catch (IOException ioe) {
            System.out.println(
                "Failed in writing data to the file " +
                file.getPath() + ".");
            System.out.println("Problem encountered: " +
                ioe.getMessage());
```

```
        } catch (ArrayIndexOutOfBoundsException ofbe) {
            // The ArrayIndexOutOfBoundsException occurs most probably
            // because no program parameters are supplied. Prompt the
            // the user the usage of the program.
            System.out.println(
                "Usage: java FillFile6 <file> <size in KB>");
        }
    }
}
```

TestFileCreator6.java

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestFileCreator6
public class TestFileCreator6 {

    // Main executive method
    public static void main(String args[]) {
        // Create a FileCreator6 object
        FileCreator6 creator = new FileCreator6();

        // Create a file by specifying the name and size
        creator.create(args);
    }
}
```

The modified definition of the class `FileCreator6` presented here is not recommended. The reasons are:

1 The `ArrayIndexOutOfBoundsException` exception is an unchecked exception, and its occurrence indicates a design or implementation problem. Therefore, using a `try/catch` block to handle the exception just hides the design deficiency.

2 Exception handling should be used for handling exceptional conditions instead of for implementing kinds of program logic or affecting the flow of control. Classes written using this approach are difficult to maintain.

Self-test 8.7

1 The class `Exception` is the parent class of all exceptions. Therefore, the implication of a method defined with a `throws` clause for `Exception` is that all exceptions will not be handled by the method. As a result, the compiler software will accept the method even though it contains statements that may cause an exception, no matter what type of exception it is.

Similarly, if all statements in a method are enclosed in a `try` block with a `catch` block for `Exception`, any exception caused in the `try` block will be handled by the `catch` block for `Exception`. Therefore, the compiler software will accept the method as well.

2 For Programmer A, as all methods are defined with a `throws` clause for `Exception`, any exception caused will not be handled by the method and is handled by the caller of the method. However, the caller method is defined with a `throws` clause for `Exception` as well; the exception will be handled by its caller method. Such propagation of the burden of handling the exception will continue until the main executive method `main()`. As the `main()` method is defined with a `throws` clause for `Exception` as well, it will not handle the exception and the program will terminate immediately if an exception occurs.

For Programmer B, as all statements that may cause exceptions are enclosed in a `try` block with an empty `catch` block for `Exception`, any exception caused will be handled by the `catch` block for `Exception`. As, there is no statement in the `catch` block, no remedial action would be taken. However, the exception is considered handled and the flow of control will continue after the `catch` block. As a result, the program will continue as if no exception occurred at all.

The problem with the approach used by Programmer A is that any exception, including a checked exception, will terminate the software execution immediately. As mentioned, most checked exceptions are due to some exceptional conditions that the program should be able to handle, a serious software application should define appropriate remedial operations to resume normal operations instead of immediate termination.

The problem with the approach taken by Programmer B is that the `catch` block for `Exception` will handle all checked exceptions and `RuntimeException` exceptions and its subclasses, because `RuntimeException` is a subclass of the `Exception` class. As mentioned, exceptions of type `RuntimeException` and its subclasses are unchecked exceptions, and the occurrence of unchecked exceptions reveals an imperfect program design or implementation. Therefore, the approach taken by Programmer B actually hides the program design and implementation faults. That is, the reliability of the software is questionable.

In a few words, the software built by Programmer A terminates even though it is possible to perform remedial operations. The software built by Programmer B does not terminate even though there are imperfect program design or implementation faults.

Self-test 8.8

DateWriter.java

```
// Resolve classes in java.io package
import java.io.*;
// Resolve the class Date
import java.util.Date;
```

```
// Definition of class DateWriter
public class DateWriter {

    // Create a file with a Date object as its content
    public void create(String name) throws IOException {
        // Create a file object referring to the specified file
        File file = new File(name);

        // Create a OutputStream associated with the file
        OutputStream os = new FileOutputStream(file);

        // Create a ObjectOutputStream associated with the OutputStream
        ObjectOutputStream out = new ObjectOutputStream(os);

        // Write the Date object
        out.writeObject(new Date());

        // Close the ObjectOutputStream and hence the associated
        // resources
        out.close();
    }
}
```

DateReader.java

```
// Resolve classes in java.io package
import java.io.*;
// Resolve the class Date
import java.util.Date;

// Definition of class DateReader
public class DateReader {

    // Create a file with a Date object as its content
    public void show(String name) throws IOException {
        // Create a file object referring to the specified file
        File file = new File(name);

        // Create a InputStream associated with the file
        InputStream is = new FileInputStream(file);

        // Create a ObjectInputStream associated with the InputStream
        ObjectInputStream in = new ObjectInputStream(is);

        // Read the Date object
        Date date = null;
        try {
            date = (Date) in.readObject();
        }  catch (ClassNotFoundException e) {
            // Execute if the class definition of the object
            // read cannot be found by the JVM
            System.err.println(
                "Cannot find the necessary class definition.");
        }
```

```
        // Show the Date object
        System.out.println(date);

        // Close the ObjectInputStream and hence the associated
        // resources
        in.close();
    }
```

TestDateWriter.java

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestDateWriter
public class TestDateWriter {

    // Main executive method
    public static void main(String args[]) throws IOException {
        if (args.length < 1) {
            System.out.println("Usage: java TestDateWriter <file>");
        }
        else {
            // Create a DateWriter object
            DateWriter writer = new DateWriter();

            // Create the file with a Date object as contents
            writer.create(args[0]);
        }
    }
}
```

TestDateReader.java

```
// Resolve classes in java.io package
import java.io.*;

// Definition of class TestDateReader
public class TestDateReader {

    // Main executive method
    public static void main(String args[]) throws IOException {
        if (args.length < 1) {
            System.out.println("Usage: java TestDateReader <file>");
        }
        else {
            // Create a DateReader object
            DateReader Reader = new DateReader();

            // Read the file with a Date object as contents
            // to be shown on the screen
            Reader.show(args[0]);
        }
    }
```

Self-test 8.9

SearchIntegerInFile.java

```
// Resolve classes in the java.io package
import java.io.*;

// Definition of class SearchIntegerInFile
public class SearchIntegerInFile {

    // Main executive method
    public static void main(String args[]) throws IOException {
        // Check if a file is specified by program parameter
        if (args.length < 2) {
            // If no file is supplied, show usage information
            System.out.println(
                "Usage : java SearchIntegerInFile <file> <int>");
        }
        else {
            // Create a File object that refers to the file
            File file = new File(args[0]);

            // Create a FileReader object specified by the File
            // object and is treated as a general Reader object
            Reader reader = new FileReader(file);

            // Create a BufferedReader object specified by the
            // Reader object
            BufferedReader br = new BufferedReader(reader);

            // Get the integer to be searched
            int target = Integer.parseInt(args[1]);

            // Initialize a variable for the searching result
            boolean found = false;

            // Initialize the variable for the line number
            int lineNumber = 0;

            // Declare a variable for storing the line read
            String lineRead;

            // The loop for reading each line from the file while
            // the number is still not found and not the end of file
            while (!found && (lineRead = br.readLine()) != null) {
                lineNumber++;
                found = (Integer.parseInt(lineRead) == target);
            }

            // Close the BufferedReader and hence all related
            // resources
            br.close();

            // Show the searching result
            if (found) {
                System.out.println(
                    "The number " + args[1] +
```

```
                    " is found at the line #" + lineNumber +
                    " of " + args[0] + ".");
            }
            else {
                System.out.println(
                    "The number " + args[1] +
                    " is not found in " + args[0] + ".");
            }
        }
    }
}
```

Self-test 8.10

PayrollCalculator1.java

```
// Resolve classes in java.io classes
import java.io.*;

// Definition of class PayrollCalculator1
public class PayrollCalculator1 {
    // Attribute
    private Staff[] staffList;    // The staff list to be processed

    // Constructor
    public PayrollCalculator1(Staff[] staffList) {
        this.staffList = staffList;
    }

    public void showReport(String filename) {
        try {
            // Create a File object referring to the file
            File file = new File(filename);

            // Create a FileWriter for writing to the file
            Writer writer = new FileWriter(file);

            // Create a PrintWriter for using the println() method
            PrintWriter out = new PrintWriter(writer);

            // Declare and initialize running totals
            double totalSalary = 0.0;
            double totalMPFByStaff = 0.0;
            double totalMPFByCompany = 0.0;

            // Show the listing title
            out.println("Staff\tRaw\tNet\tMPF by\tMPF by");
            out.println("Name\tSalary\tSalary\tStaff\tCompany");
            out.println("------- ------- ------- ------- -------");

            // Iterate each staff to show his/her details
            for (int i=0; i < staffList.length; i++) {
                // Get the staff payroll details
                double salary = staffList[i].findSalary();
                double netSalary = staffList[i].findNetSalary();
                double mpfByStaff = staffList[i].findMPFByStaff();
```

```
                double mpfByCompany = staffList[i].findMPFByCompany();

                // Show the details
                out.println(
                    staffList[i].getName() +
                    "\t" + salary +
                    "\t" + netSalary +
                    "\t" + mpfByStaff +
                    "\t" + mpfByCompany);

                // Update the running totals
                totalSalary += netSalary;
                totalMPFByStaff += mpfByStaff;
                totalMPFByCompany += mpfByCompany;
            }

            // Show the grand totals
            out.println("------- ------- ------- ------- -------");
            out.println(
                "Total" +
                "\t\t" + totalSalary +
                "\t" + totalMPFByStaff +
                "\t" + totalMPFByCompany);

            // Close the stream
            out.close();
        } catch (IOException ioe) {
            // Problem encountered in performing I/O operations
            // Show message to the user
            System.out.println(
                "A problem encountered in performing I/O operation.");
            System.out.println("Reason : " + ioe.getMessage());
        }
    }
}
```

TestPayrollCalculator1.java

```
// Definition of class TestPayrollCalculator1 {
public class TestPayrollCalculator1 {

    // Main exeuctive method
    public static void main(String args[]) {
        // Verify the number of program parameters
        if (args.length < 1) {
            System.out.println(
                "Usage: java TestPayrollCalculator1 <file>");
        }
        else {
            // Create an array of staff in the company
            Staff[] staff = {
                new Clerk("Mary", 4000.0),
                new Clerk("Peter", 6000.0),
                new SalesPerson("Joe", 4000.0, 100000.0, 0.05),
                new SalesPerson("Amy", 5000.0, 200000.0, 0.08),
                new Manager("John", 20000.0, 10000.0)
```

```
            };

            // Create a PayrollCalculator1 object by supplying an array
            // of Staff objects
            PayrollCalculator1 calculator =
                new PayrollCalculator1(staff);

            // Show the payroll report
            calculator.showReport(args[0]);
        }
    }
}
```

Self-test 8.11

1 **Console.java**

```
// Import statement for resolving classes in java.io package
import java.io.*;

// Definition of class Console
public class Console {
    // Create a BufferedReader which indirectly attaches the keyboard
    private static BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    // Show the supplied prompt to the screen
    private static void showPrompt(String prompt) {
        // If the supplied prompt is not null, show it
        if (prompt != null) {
            System.out.print(prompt);
        }
    }

    // Read a line from the keyboard and return as a String object
    private static String readStringFromKeyboard() {
        String input = null;
        try {
            input = in.readLine();
        } catch (IOException ioe) {
            // It is almost impossible for causing an
            // IOException, but the catch block is needed
            System.out.println("Unexpected I/O error");
        }
        return input;
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of int
    public static int readInt(String prompt) {
        // A infinite loop for prompting and accepting user entry.
        // If no NumberFormatException occur, the return statment
        // returns the converted value
        while (true) {
            try {
                showPrompt(prompt);
```

```
                return Integer.parseInt(readStringFromKeyboard());
            } catch (NumberFormatException nfe) {
                System.out.println(
                    "Invalid number format. Please re-enter.");
            }
        }
    }

    // Read a line from the keyboard and return as a data of int
    public static int readInt() {
        return readInt(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of long
    public static long readLong(String prompt) {
        // A infinite loop for prompting and accepting user entry.
        // If no NumberFormatException occur, the return statment
        // returns the converted value
        while (true) {
            try {
                showPrompt(prompt);
                return Long.parseLong(readStringFromKeyboard());
            } catch (NumberFormatException nfe) {
                System.out.println(
                    "Invalid number format. Please re-enter.");
            }
        }
    }

    // Read a line from the keyboard and return as a data of long
    public static long readLong() {
        return readLong(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of double
    public static double readDouble(String prompt) {
        // A infinite loop for prompting and accepting user entry.
        // If no NumberFormatException occur, the return statment
        // returns the converted value
        while (true) {
            try {
                showPrompt(prompt);
                return Double.parseDouble(readStringFromKeyboard());
            } catch (NumberFormatException nfe) {
                System.out.println(
                    "Invalid number format. Please re-enter.");
            }
        }
    }
```

```
// Read a line from the keyboard and return as a data of double
    public static double readDouble() {
        return readDouble(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return as a data of float
    public static float readFloat(String prompt) {
        // A infinite loop for prompting and accepting user entry.
        // If no NumberFormatException occur, the return statment
        // returns the converted value
        while (true) {
            try {
                showPrompt(prompt);
                return Float.parseFloat(readStringFromKeyboard());
            } catch (NumberFormatException nfe) {
                System.out.println(
                    "Invalid number format. Please re-enter.");
            }
        }
    }

    // Read a line from the keyboard and return as a data of float
    public static float readFloat() {
        return readFloat(null);
    }

    // Show the supplied prompt and read a line from the keyboard
    // and return the String object right away
    public static String readString(String prompt) {
        // A infinite loop for prompting and accepting user entry.
        // If no NumberFormatException occur, the return statment
        // returns the converted value
        while (true) {
            try {
                showPrompt(prompt);
                return readStringFromKeyboard();
            } catch (NumberFormatException nfe) {
                System.out.println(
                    "Invalid number format. Please re-enter.");
            }
        }
    }

    // Read a line from the keyboard and return it right away
    public static String readString() {
        return readString(null);
    }
}
```

2 EquationSolver.java

```
// The class definition of EquationSolver for setting up the
// environment and test the QuadraticEquation object
public class EquationSolver {

    // The main executive method
    public static void main(String args[]) {
        // Create a QuadraticEquation object and use variable equation
        // to refer to it
        QuadraticEquation equation = new QuadraticEquation();

        // Get the coefficient A and set it to the quadratic equation
        double coeffA = Console.readDouble(
            "Please enter coefficient A : ");
        equation.setCoeffA(coeffA);

        // Get the coefficient B and set it to the quadratic equation
        double coeffB = Console.readDouble(
            "Please enter coefficient B : ");
        equation.setCoeffB(coeffB);

        // Get the coefficient C and set it to the quadratic equation
        double coeffC = Console.readDouble(
            "Please enter coefficient C : ");
        equation.setCoeffC(coeffC);

        // Check if the quadratic equation has real roots based on the
        // determinant, and prepare the message to be shown to the user
        String message;
        if (equation.getDeterminant() >= 0.0) {
            double firstRoot = equation.getFirstRealRoot();
            double secondRoot = equation.getSecondRealRoot();
            message = "The roots of the quadratic equation are:" +
                "\nFirst root = " + firstRoot +
                "\nSecond root = " + secondRoot;
        }
        else {
            message = "No real roots";
        }

        // Show the result to the user with a dialog
        System.out.println(message);
    }
}
```

Self-test 8.12

WebPageGetter.java

```
// Resolve classes for performing input/output
import java.io.*;
// Resolve URL related classes
import java.net.*;

// The definition of class WebPageGetter
public class WebPageGetter {
    // Attribute
    private BufferedReader in;    // The BufferedReader object for
                                  // reading lines from the URL

    // Constructor
    public WebPageGetter(String urlString) {
        try {
            // Create a URL object associated with the URL address
            URL url = new URL(urlString);

            // Obtain the URLConnection object from the URL object
            URLConnection conn = url.openConnection();

            // Obtain the name of the coding used by the URL resource
            String encoding = conn.getContentEncoding();

            // Obtain the InputStream object from the URLConnection
            InputStream is = conn.getInputStream();

            // Create a InputStreamReader object based on the
            // InputStream obtained from the URLConnection. If encoding
            // is specified, it is supplied to the InputStreamReader for
            // translation. Otherwise, the default coding system of the
            // computer is used.
            Reader reader = null;
            if (encoding != null) {
                reader = new InputStreamReader(is, encoding);
            } else {
                reader = new InputStreamReader(is);
            }

            // Create a BufferedReader object based on the
            // InputStreamReader object
            in = new BufferedReader(reader);
        } catch (IOException ioe) {
            // Show error message if there is any I/O runtime error
            System.out.println("Error in performing I/O operation");
        }
    }

    // Show the content of the web resource
    public void showContent() {
        // If the BufferedReader is initialized
        if (in != null) {
            try {
```

```
                // Read the lines from the URL resource and show them on
                // the screen
                String line = null;
                while ((line = in.readLine()) != null) {
                    System.out.println(line);
                }
            } catch (IOException ioe) {
                System.out.println("Error in performing I/O operation");
            }
        } else {
            System.out.println("Cannot read from the URL");
        }
    }

    // Copy the content of the web resource to the file line by line
    public void copyContentToFile(String filename) {
        // If the BufferedReader is initialized
        if (in != null) {
            try {
                // Create a File object referring to the supplied file
                File file = new File(filename);

                // Create a Writer object that translate the characters
                // and output to the file in units of byte
                Writer writer = new FileWriter(file);

                // Create a PrintWriter object to prepare the characters
                // to be written by the Writer object
                PrintWriter pw = new PrintWriter(writer);

                // Read the lines from the URL resource and show them on
                // the screen
                String line = null;
                while ((line = in.readLine()) != null) {
                    // Write a String to the file with a default
                    // end of line pattern
                    pw.println(line);
                }

                // Close the PrintWriter object
                pw.close();
            } catch (IOException ioe) {
                System.out.println("Error in performing I/O operation");
            }
        } else {
            System.out.println("Cannot read from the URL");
        }
    }
}
```

TestWebPageGetter.java

```
// The definition of class TestWebPageGetter
public class TestWebPageGetter {

    // Main executive method
    public static void main(String args[]) {
        // Verify the number of program parameter
        if (args.length < 1) {
            // If no program parameter is given, show usage message
            System.out.println(
                "Usage: java TestWebPageGetter <url> [<file>]");
        }
        else {
            // Create a WebPageGetter object that is associated with
            // the URL
            WebPageGetter getter = new WebPageGetter(args[0]);

            // Check if the second program parameter is provided
            if (args.length > 1) {
                // Copy the contents to the specified file
                getter.copyContentToFile(args[1]);
            }
            else {
                // Show the contents of the resource specified by
                // the URL
                getter.showContent();
            }
        }
    }
}
```