

14.3 Handling files as binary files

Although all files on a computer's hard disk contain only bits, binary digits, we say that some files are text files while others are considered binary files. Categorizing files into text files and binary files is thus somewhat arbitrary. A binary file can be considered a series of bytes, and so can a text file. It is possible to open a text file as a binary file. The essential distinction between these two file types is that a binary file can contain any bytes, but a text file contains only character codes of visible characters, line termination characters, and such text formatting characters as tabulator characters. It is possible to view a text file with an editor program (e.g. JCreator, EditPad or Notepad). Binary files, on the other hand, usually contain data that is not readable by the human eye, and may even cause strange behavior in an editor program.

When we want to process binary files in a Java program, we use different standard classes depending on whether we want to read data from a file or to write data to a file. A file can be opened for binary reading operations with a statement like

```
FileInputStream file_to_read = new
    FileInputStream( file_name_as_string ) ;
```

In this statement the constructor of class `FileInputStream` opens a file with the given file name. After the statement is successfully executed, the file is represented by the `FileInputStream` object, and it can be read by using the file reading methods of class `FileInputStream`.

The basic `FileInputStream` method for reading a file in binary form has the name `read()`. Actually, there are several versions of the `read()` method. The version of the `read()` method which we use in this book can be called to read a certain number of bytes from a file. The method puts the bytes it reads to an array of type `byte[]`, and returns a value that indicates how many bytes it was able to read. Normally, the `read()` method reads as many bytes as it was asked to read, but when the end of the file has been reached, it is possible that it may not be able to read all the bytes that were requested. One possible way to call the `read()` method is the following:

```
byte[] bytes_from_file = new byte[ 50 ] ;
int number_of_bytes_read =
    file_to_read.read( bytes_from_file ) ;
```

This method call could be "translated" as follows: "Here is this array of bytes. Please, read bytes from the file and put them to the array, starting from the array position that has index 0. Read 50 bytes at maximum because that is the length of the array; it cannot hold any more bytes." After receiving these orders, the `read()` method would do its job and return the number of bytes it was able to read. The `read()` method checks the length of the given array of type `byte[]`, and automatically delivers as many bytes as is the length of the array.

Files are typically read with a loop, but the loop that uses method `read()` becomes somewhat complicated as the above method call must be fitted into it. One possibility to construct a loop that calls method `read()` is the following one which calculates the size of a file by counting how many bytes the file contains:

```

byte[] bytes_from_file = new byte[ 50 ] ;
int file_size_in_bytes = 0 ;
int number_of_bytes_read = 0 ;
while ( ( number_of_bytes_read =
        file_to_read.read( bytes_from_file ) ) > 0 )
{
    file_size_in_bytes = file_size_in_bytes +
                        number_of_bytes_read ;

    // This loop does not do anything to the read data.
}

```

In the above case, method `read()` is called in the somewhat complicated boolean expression of the `while` loop. When the boolean expression is evaluated, the `read()` method is called, and as a result variable `number_of_bytes_read` gets a new value, and that value is then compared to zero. The boolean expression is true as long as the `read()` method is able to read bytes from the file.

Method `read()` works with binary files in the same manner as method `readLine()` works with text files. In the loop above, when method `read()` is called for the first time, it reads the first 50 bytes from the file, and subsequent calls read the file from the position where the previous reading operation stopped. The above loop reads 50-byte data blocks from the file until it has read the last block. Only the last block is likely to be less than 50 bytes in length. In file reading operations it is usually not known when the final data at the end of a file is going to be read. Therefore, a loop that reads a file must always be prepared to encounter the end of file.

The above loop can be made easier to understand if a boolean variable is used in the following way:

```

byte[] bytes_from_file = new byte[ 50 ] ;
int file_size_in_bytes = 0 ;
boolean bytes_still_available_in_file = true ;
while ( bytes_still_available_in_file == true )
{
    int number_of_bytes_read =
        file_to_read.read( bytes_from_file ) ;

    if ( number_of_bytes_read > 0 )
    {
        file_size_in_bytes = file_size_in_bytes +
                            number_of_bytes_read ;

        // This loop does not do anything to the read data.
    }
    else
    {
        bytes_still_available_in_file = false ;
    }
}

```

This loop is, however, somewhat longer than the first loop. For this reason I ended up using a loop like the first one in program **FileToNumbers.java**.

Program **FileToNumbers.java** demonstrates how a file is read as a binary file. It also shows that a text file can be treated as a binary file. The program reads a file in 16-byte blocks and displays the bytes as hexadecimal numbers on the screen. Those bytes that represent character codes of visible characters are also shown as characters.

When you want to write data to a binary file, you can use class `FileOutputStream` to create an object that represents a binary file for writing operations. One possible file opening statement is the following

```
FileOutputStream binary_file_for_writing = new
    FileOutputStream( file_name_as_string ) ;
```

Here, the constructor of class `FileOutputStream` opens the file whose name it gets as a parameter. The file is opened so that it is created if it does not exist, and if it exists, it is overwritten with new data.

Just as class `FileInputStream` provides methods named `read()` to read bytes from binary files, the `FileOutputStream` class provides methods named `write()` for writing bytes to binary files. Supposing that a file is opened with the above statement, and that there is the array

```
byte[] some_array_of_bytes = new byte[ 300 ] ;
```

containing some data, the statement

```
binary_file_for_writing.write( some_array_of_bytes ) ;
```

would write all the 300 bytes from the array referenced by `some_array_of_bytes` to the file represented by the `FileOutputStream` object referenced by `binary_file_for_writing`, and the statement

```
binary_file_for_writing.write( some_array_of_bytes,
                               2,
                               5 ) ;
```

would use a different version of the `write()` method, and write the five bytes starting from the third array position that has index 2. The first parameter for the `write()` method specifies the byte array from which bytes should be copied to a file, the second parameter specifies the array position from which the writing operations should begin, and the third parameter says how many bytes should be written. As there are several versions of the `write()` method, there is also a version with which it is possible to write just a single byte to a file. This `write()` method could be called with a statement like

```
binary_file_for_writing.write( 0x22 ) ;
```

which would write the hexadecimal value `0x22` to the binary file that was opened above.

NumbersToFile.java is a program that demonstrates how data can be written to a binary file. The program uses class `FileOutputStream` to open a file for binary writing operations. In addition, the program uses a class named `DataOutputStream` in the writing operations. In program **NumbersToFile.java**, a file named **NumbersToFile_output.data** is first opened for binary writing operations with the statement

```
FileOutputStream file_output_stream =
    new FileOutputStream( "NumbersToFile_output.data" ) ;
```

and then the `FileOutputStream` object is used to construct a `DataOutputStream` object with the statement

```
DataOutputStream file_to_write =
    new DataOutputStream( file_output_stream ) ;
```

and it is the `DataOutputStream` object that is used later to write data to the file.

The `DataOutputStream` class provides many methods for writing operations, and it is therefore useful in a program like `NumbersToFile.java`. Figure 14-3 describes how a `DataOutputStream` object co-operates with a `FileOutputStream` object when data is written to a file. Class `DataOutputStream` has a "sister class" named `DataInputStream`. The methods of `DataInputStream` class are convenient when we need to read data that is written with the `DataOutputStream` methods. In the next section a program named `Collect.java` uses both `DataOutputStream` and `DataInputStream` objects in file operations.

Program `NumbersToFile.java` destroys the previous content of its output file if the file already exists. If you want to write data to a binary file without destroying the previously written data, you must open the file in appending mode. If the file opening statement of program `NumbersToFile.java` were written in the following way

```
FileOutputStream file_output_stream = new
    FileOutputStream( "NumbersToFile_output.data", true ) ;
```

where the constructor of class `FileOutputStream` is given two parameters, the program would append data to the end of the `NumbersToFile_output.data` file, and the file would grow always when the program is executed. When the second constructor parameter is the literal value `true`, or a `boolean` variable whose value is `true`, the file is opened in appending mode. Also in the appending mode the file is created if it does not yet exist.

These are sample pages from Kari Laitinen's book
"A Natural Introduction to Computer Programming with Java".
For more information, please visit
<http://www.naturalprogramming.com/javabook.html>

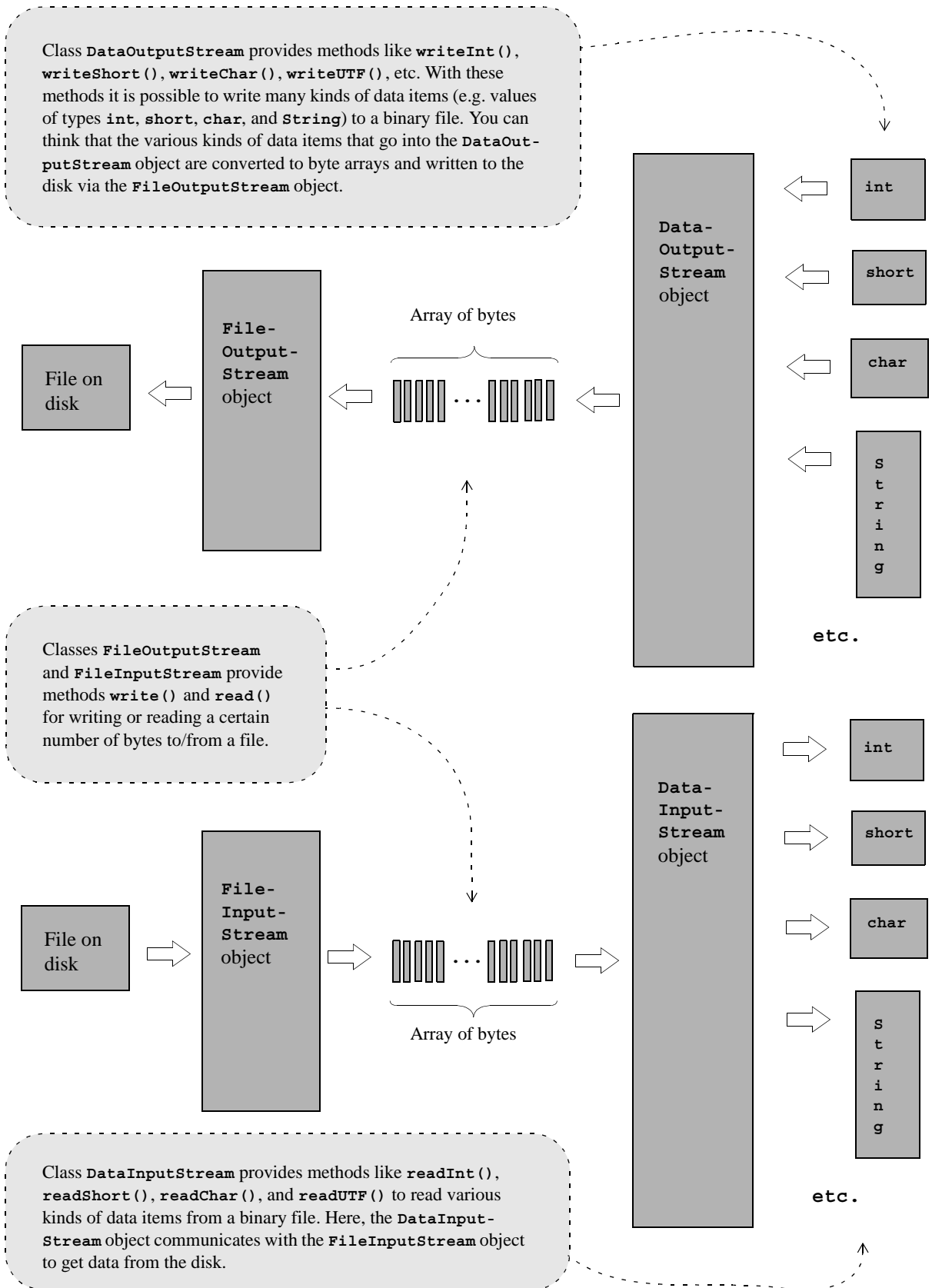


Figure 14-3. Co-operation of classes in binary file access.

```
// FileToNumbers.java
import java.io.* ;

class FileToNumbers
{
    public static void main( String[] command_line_parameters )
    {
        if ( command_line_parameters.length == 1 )
        {
            try
            {
                FileInputStream file_to_read =
                    new FileInputStream( command_line_parameters[ 0 ] ) ;

                byte[] bytes_from_file = new byte[ 16 ] ;

                int number_of_bytes_read ;

                while ( ( number_of_bytes_read =
                    file_to_read.read( bytes_from_file ) ) > 0 )
                {
                    StringBuilder line_of_bytes = new StringBuilder() ;
                    StringBuilder bytes_as_characters = new StringBuilder () ;

                    for ( int byte_index = 0 ;
                        byte_index < number_of_bytes_read ;
                        byte_index ++ )
                    {
                        line_of_bytes.append(
                            String.format( " %02X", bytes_from_file[ byte_index ] ) ) ;

                        char byte_as_character =
                            (char) bytes_from_file[ byte_index ] ;

                        if ( byte_as_character >= ' ' )
                        {
                            bytes_as_characters.append( byte_as_character ) ;
                        }
                        else
                        {
                            bytes_as_characters.append( ' ' ) ;
                        }
                    }

                    System.out.printf( "\n%-48s %s", line_of_bytes,
                        bytes_as_characters ) ;
                }
            }
        }
    }
}
```

The file is treated as a binary file. Method `read()` reads 16 bytes from the file at a time because the length of the array `bytes_from_file` is 16. The first call to method `read()` reads the first 16 bytes, the second call reads the following 16 bytes, and so on. This `while` loop is executed as long as there are bytes available in the file.

The non-printable characters are shown as spaces.

It is possible to print characters stored in `StringBuilder` objects with the `printf()` method. The format specifier `%-48s` stipulates that the `toString()` method is invoked for the object referenced by `line_of_bytes`, and the string is printed left-justified into a printing field that is 48 character positions wide.

FileToNumbers.java - 1: A program to show the contents of a file as hexadecimal bytes.

It is important that all files, both binary and text files, are closed with a `close()` method after they are not used any more.

`FileNotFoundException` is thrown by the constructor of class `FileInputStream` if it cannot find a file with the specified file name.

```

    file_to_read.close() ;
}
catch ( FileNotFoundException caught_file_not_found_exception )
{
    System.out.print( "\n Cannot open file "
        + command_line_parameters[ 0 ] ) ;
}
catch ( IOException caught_io_exception )
{
    System.out.print( "\n Error while processing file "
        + command_line_parameters[ 0 ] ) ;
}
}
else
{
    System.out.print( "\n You have to command this program as: \n"
        + "\n java FileToNumbers file.ext \n" ) ;
}
}
}

```

The only way to give a file name to this program is to write it on the command line.

FileToNumbers.java - 2. The last part of the program.

```

D:\javafiles3>type ministory.txt
1 == ministory.txt ==
2
3 aaa AAA bbb BBB
4
5 This is the end.

```

The contents of the `StringBuilder` object referenced by `bytes_as_characters` are printed after the contents of the object referenced by `line_of_bytes`.

```

D:\javafiles3>java FileToNumbers ministory.txt

```

```

31 20 3D 3D 20 6D 69 6E 69 73 74 6F 72 79 2E 74 1 == ministory.t
78 74 20 3D 3D 3D 0D 0A 32 0D 0A 33 20 61 61 61 xt === 2 3 aaa
20 41 41 41 20 62 62 62 20 42 42 42 0D 0A 34 0D AAA bbb BBB 4
0A 35 20 54 68 69 73 20 69 73 20 74 68 65 20 65 5 This is the e
6E 64 2E 0D 0A 0D 0A nd.

```

All bytes from the file are shown in hexadecimal form. A newline character is represented by two character codes, `0DH` and `0AH`, inside the file. There are two newlines at the end of the file.

FileToNumbers.java - X. Showing the contents of file "ministory.txt".

```
// NumbersToFile.java

import java.io.* ;

class NumbersToFile
{
    public static void main( String[] not_in_use )
    {
        try
        {
            FileOutputStream file_output_stream =
                new FileOutputStream( "NumbersToFile_output.data" ) ;

            DataOutputStream file_to_write =
                new DataOutputStream( file_output_stream ) ;

            int integer_to_file = 0x22 ;

            while ( integer_to_file < 0x77 )
            {
                file_to_write.writeInt( integer_to_file ) ;

                integer_to_file = integer_to_file + 0x11 ;
            }

            file_to_write.writeShort( (short) 0x1234 ) ;
            file_to_write.writeDouble( 1.2345 ) ;
            file_to_write.writeBoolean( true ) ;
            file_to_write.writeBoolean( false ) ;
            file_to_write.writeUTF( "aaAAbbBB" ) ;

            byte[] bytes_to_file = { 0x4B, 0x61, 0x72, 0x69 } ;

            file_to_write.write( bytes_to_file, 0, 4 ) ;

            file_to_write.close() ;
        }
        catch ( Exception caught_exception )
        {
            System.out.print( "\n File error. Cannot write to file." ) ;
        }
    }
}
```

A `FileOutputStream` object that represents a file on the disk is created first, and then that object is used to create a `DataOutputStream` object. You can think that the `FileOutputStream` object is controlled by the `DataOutputStream` object. The standard class `DataOutputStream` provides useful methods to write binary values to a file.

If a file with the specified name already exists, it will be written over.

Class `DataOutputStream` provides many methods that can write data items of different types to a file. Here, a method named `write()` writes four bytes from the array (referenced by `bytes_to_file`). The writing begins from the array position whose index is 0 (the first array position). The bytes that are put to the array `0x4B`, `0x61`, `0x72`, and `0x69` are the character codes of letters `K`, `a`, `r`, and `i`, respectively.

NumbersToFile.java - 1. A program that stores numerical values to a binary file.

The execution of the program does not produce any output to the screen. All output goes to a file whose contents are shown here by using the previous example program.

The `while` loop of the program puts five values of type `int` to the file. Each `int` value occupies four bytes in the file, and the bytes are in such an order that the most significant byte is the first in the file.

```
D:\javafiles3>java NumbersToFile
D:\javafiles3>java FileToNumbers NumbersToFile_output.data
00 00 00 22 00 00 00 33 00 00 00 44 00 00 00 55      " 3 D U
00 00 00 66 12 34 3F F3 C0 83 12 6E 97 8D 01 00      f 4???? n??
00 08 61 61 41 41 62 62 42 42 4B 61 72 69           aaAAbbBBKari
```

The contents of a string are stored in UTF-8 format. Each character code occupies only a single byte. Before the character codes, there is a two-byte (16-bit) value that tells how many bytes are stored. In this case there are 8 bytes. The UTF-8 encoding is such that the characters whose codes are less than 0x80 (i.e., the normal characters and symbols in the English alphabet) are represented by a single byte. The UTF-8 encoding used by the `writeUTF()` method differs slightly from the official UTF-8 format. (See page 456 for more information about UTF-8.)

The `double` value 1.2345 is stored in eight bytes, but the number is encoded so that it is not possible to read it. The boolean values `true` and `false` are stored as bytes containing 1 and 0, respectively.

NumbersToFile.java - X. Examining the contents of file "NumbersToFile_output.data".

Exercises related to binary files

- Exercise 14-6. Write a program that compares two files to find out if the files have the same contents. Two files are equal if their lengths are equal, and they contain the same bytes in the same order. You can use method `read()` of class `FileInputStream` to read equal size byte blocks from two files. After each reading operation you must test if the byte blocks are the same.
- Exercise 14-7. Write a program (e.g. `FilecopyBinary.java`) that makes a copy of a file by using binary file access. You can use the `read()` and `write()` methods of classes `FileInputStream` and `FileOutputStream` to move byte blocks from one file to another. You do not need to use the `DataOutputStream` class because class `FileOutputStream` has a suitable `write()` method.
- Exercise 14-8. Program `FileToNumbers.java` may be a useful program when you work with files, but the program is not convenient to use when large files are examined with it. The contents of a large file do not fit to a single screen. Improve the program so that when it prints the contents of a file it asks the user of the program to press the Enter key before it shows the following 20 or so lines on the screen.