

# Chapter-2 File Handling

Serial Access Files, File Methods, Redirection, Command Line Parameters, Random Access Files.

After reading this chapter, you should:

- know how to create and process serial files in Java;
- know how to create and process random access files in Java;
- know how to redirect console input and output to disc files;
- know how to construct GUI-based file-handling programs;
- know how to use command line parameters with Java programs;

# Serial Access Files

- Serial access files are files in which data is stored in physically adjacent locations
- With no particular logical order
- With each new item of data being added to the end of the file.
- A sequential (a ordered) file is a serial file, but a serial file is not necessarily a sequential file.
- They are often used only to hold relatively **small amounts** of data or for **temporary storage**, prior to processing.
- But such files are simpler to handle and are in quite common usage.
- The internal structure of a serial file can be either **binary** (i.e., a compact format determined) or **text** (human-readable format, almost invariably using ASCII).
- **The former** stores data more efficiently, but the latter is much more convenient for human beings.

Use just a **File** object for either input or output (though not for both at the same time).

**The File constructor** takes a String argument that specifies the name of the file as it appears in a directory listing.

Examples

(i)        `File inputFile = new File("accounts.txt");`

(ii)       `String fileName = "dataFile.txt";`

.....

`File outputFile = new File(fileName);`

The full path name may be included, but double backslashes are then required in place of single backslashes

For example:

```
File resultsFile = new File("c:\\data\\results.txt");
```

Class **File** is contained within package java.io

We can wrap a **Scanner** object around a **File** object for input and a **PrintWriter** object around a **File** object for output.

(The PrintWriter class is also within package java.io .)

Examples

**(i) Scanner input = new Scanner(new File("inFile.txt"));**

**(ii) PrintWriter output = new PrintWriter(new File("outFile.txt"));**

We can then make use of methods *next*, *nextLine*, *nextInt*, *nextFloat*, ... *for input* and methods *print* and *println* for output.

Examples (using objects *input* and *output*, as declared above)

- (i) `String item = input.next();`
- (ii) `output.println("Test output");`
- (iii) `int number = input.nextInt();`

Note that we need to know the type of the data that is in the file before we attempt to read it!

*Anonymous File objects in above example.*

Named *File* objects.

Examples

- (i)                    `File inFile = new File("inFile.txt");`  
`Scanner input = new Scanner(inFile);`
- (ii)                   `File outFile = new File("outFile.txt");`  
`PrintWriter output = new PrintWriter(outFile);`

When the processing of a file has been completed, the file should be closed via the `close` method, which is a member of both the `Scanner` class and the `PrintWriter` class. For example:

`input.close();`

Now for a simple example program to illustrate file output...

## Example

Writes a single line of output to a text file.

```
import java.io.*;
public class FileTest1
{
    public static void main(String[] args) throws IOException
    {
        PrintWriter output =
            new PrintWriter(new File("test1.txt"));
        output.println("Single line of text!");
        output.close();
    }
}
```

If you need to add data to the contents of an existing file, you still need to use a **FileWriter** *object*, employing either of the following constructors with a second argument of true :

- `FileWriter(String <fileName>, boolean <append>)`
- `FileWriter(File <fileName>, boolean <append>)`

For example:

```
FileWriter addFile = new FileWriter("data.txt", true);
```

In order to send output to the file, a **PrintWriter** would then be wrapped around the `FileWriter` :

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



These two steps may, of course, be combined into one:

```
PrintWriter output = new PrintWriter(new FileWriter("data.txt", true);
```

Often, we will wish to accept data from the user during the running of a program.

In addition, we may also wish to allow the user to enter a name for the file.

It is good programming practice to use File method flush to empty the file output buffer.

The next example illustrates both of these features.

```
import java.io.*;
import java.util.*;
public class FileTest2
{
    public static void main(String[] args) throws IOException
    {
        String fileName;
        int mark;

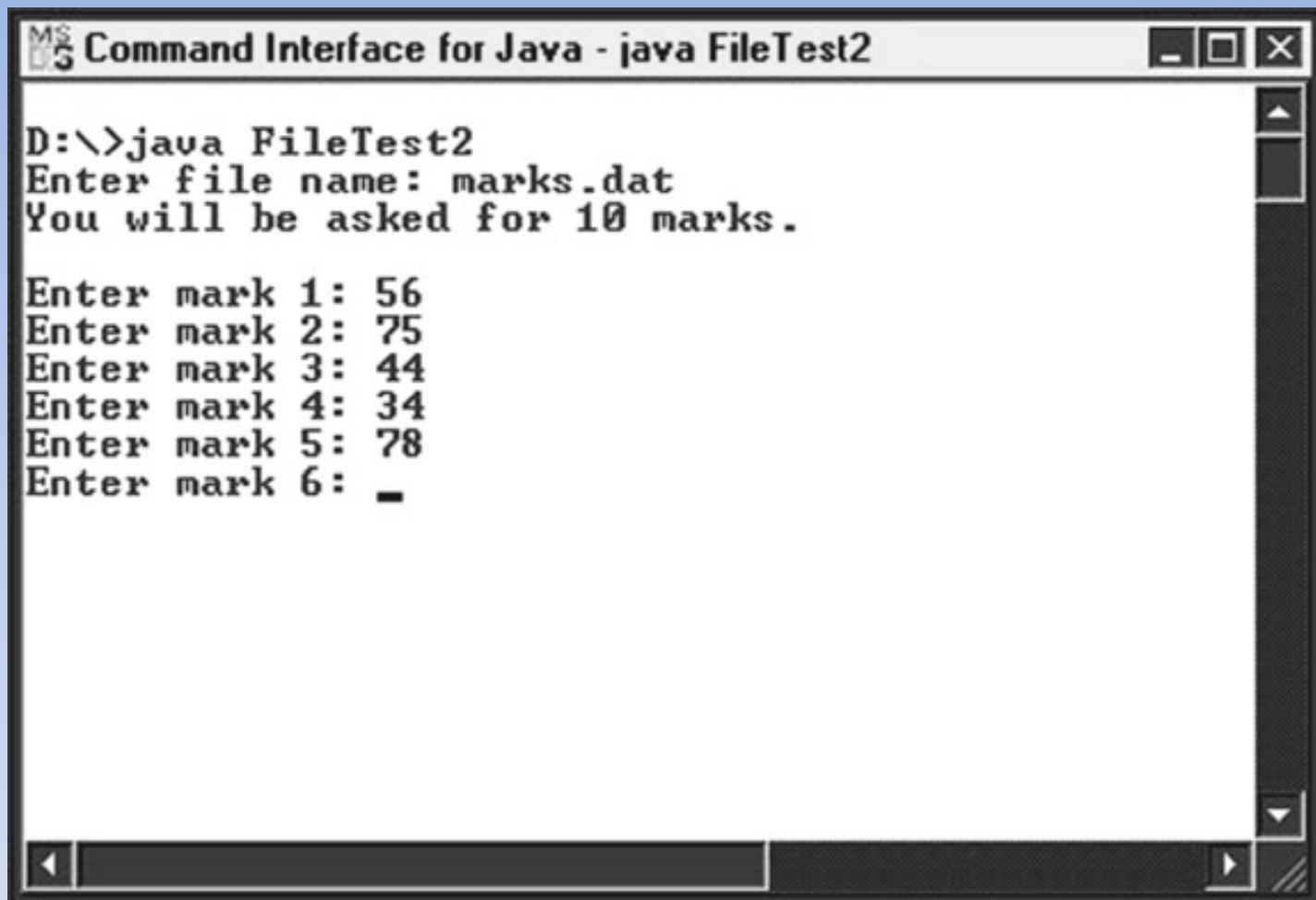
        Scanner input= new Scanner(System.in);

        System.out.print("Enter file name: ");

        fileName = input.nextLine();

        PrintWriter output = new PrintWriter(new File(fileName));
```

```
System.out.println("Ten marks needed.\n");  
for (int i=1; i<11; i++)  
{  
    System.out.print("Enter mark " + i + ": ");  
    mark = input.nextInt();  
    /* Should really validate entry! */  
    output.println(mark);  
    output.flush();  
}  
output.close();  
}  
}
```



```
MS Command Interface for Java - java FileTest2
D:\>java FileTest2
Enter file name: marks.dat
You will be asked for 10 marks.

Enter mark 1: 56
Enter mark 2: 75
Enter mark 3: 44
Enter mark 4: 34
Enter mark 5: 78
Enter mark 6: _
```

**NoSuchElementException** will be thrown when try to read beyond end-of-file

Example

```
import java.io.*;
import java.util.*;
public class FileTest3
{
    public static void main(String[] args) throws IOException
    {
        int mark, total=0, count=0;
        Scanner input = new Scanner(new File("marks.txt"));
        while (input.hasNext())
        {
            mark = input.nextInt();
            total += mark;
            count++;
        }
        input.close();
        System.out.println("Mean = " + (float)total/count);
    }
}
```

- Programmer's responsibility to impose any required **logical structuring**
  - account number;
  - customer name;
  - account balance.

Remaining...

- **File Methods**
- **Redirection**
- **Command Line Parameters**
- **Random Access Files.**

# File Methods

Class ***File*** has a large number of methods, the most important are shown below.

<b><i>boolean canRead()</i></b>	Returns <i>true</i> if file is readable and <i>false</i> otherwise.
<b><i>boolean canWrite( )</i></b>	Returns <i>true</i> if file is writeable and <i>false</i> otherwise.
<b><i>boolean delete()</i></b>	Deletes file and returns <i>true/false</i> for success/failure.
<b><i>boolean exists()</i></b>	Returns <i>true</i> if file exists and <i>false</i> otherwise.
<b><i>String getName()</i></b>	Returns name of file.
<b><i>boolean isDirectory()</i></b>	Returns <i>true</i> if object is a directory/folder and <i>false</i> otherwise.
<b><i>boolean isFile()</i></b>	Returns <i>true</i> if object is a file and <i>false</i> otherwise.
<b><i>long length()</i></b>	Returns length of file in bytes.

***String[] list()*** If object is a directory, array holding names of files within directory is returned.

***File[] listFiles()*** Similar to previous method, but returns array of *File objects*.

***boolean mkdir()*** Creates directory with name of current *File object*. Return value indicates success/failure.

```
import java.io.*;
import java.util.*;
public class FileMethods
{
    public static void main(String[] args) throws IOException
    {
        String filename;
        Scanner input = new Scanner(System.in);
        System.out.print("Enter name of file/directory ");
        System.out.print("or press <Enter> to quit: ");
        filename = input.nextLine();
    }
}
```



```

while (!filename.equals("")) //Not <Enter> key.
{
    File fileDir = new File(filename);
    if (!fileDir.exists())
    {
        System.out.println(filename+ " does not exist!");
        break; //Get out of loop.
    }
    //.....
    System.out.print(filename + " is a ");
    if (fileDir.isFile())
        System.out.println("file.");
    else
        System.out.println("directory.");
    //.....
    System.out.print("It is ");
    if (!fileDir.canRead())
        System.out.print("not ");
    System.out.println("readable.");
}

```

```

System.out.print("It is ");
if (!fileDir.canWrite())
    System.out.print("not ");
System.out.println("writeable.");
//.....
if (fileDir.isDirectory())
{
    System.out.println("Contents:");
    String[] fileList = fileDir.list();
    for (int i=0;i<fileList.length;i++)
        System.out.println(" "+ fileList[i]);
}
else
{
    System.out.print("Size of fi le: ");
    System.out.println(fi leDir.length()+ " bytes.");
}
//.....

```

```
System.out.print("\nEnter name of next file/directory ");
```

```
System.out.print("or press <Enter> to quit: ");
```

```
filename = input.nextLine();
```

```
}//End of while
```

```
input.close();
```

```
} //End of main() method
```

```
} //End of FileMethods
```

```
MS Command Interface for Java - java FileMethods
D:\>java FileMethods
Enter name of file/directory or press <Enter> to quit: cms215
cms215 is a directory.
It is readable.
It is not writeable.
Contents:
    Assessment
    B-Toolkit.doc
    B_lface.doc
    B_lface_Ex
    B_Impl_Steps.doc
    B_Proofs.doc
    GolfClub2.mch
    Lectures
    Non-B Re-Specification.doc
    Non-B Specification.doc
    Sequences.doc
Enter name of next file/directory or press <Enter> to quit: 
```

# Redirection

- **System.in** is associated with the **keyboard**
- **System.out** is associated with the **VDU**
- Input to come from some other source (such as a text file) or output to go to somewhere other than the VDU screen, then we are **redirecting the input/output**.
- Useful when debugging a program that requires anything more than a couple of items of data from the user.
- Saves time consumption
- Can be used to avoid tedious and error-prone re-entry of data when debugging a program.

We use ' < ' to specify the new source of input and ' > ' to specify the new output destination.

Examples

```
java ReadData < payroll.txt
```

```
java WriteData > results.txt
```

File input statement (via Scanner method next , nextLine , nextInt , etc.), it will now take as its input the next available item of data in file 'payroll.txt'.

Similarly, program 'WriteData(.class)' will direct the output of any print and println statements to file 'results.txt'.

We can use redirection of both input and output with the same program, as the example below shows. For example:

```
java ProcessData < readings.txt > results. txt
```

# Command Line Parameters

- It is possible to supply values (data) in addition to the name of the program during execution.
- These values are called **command line parameters** and are **values** that the program may make use of.
- Values are received by method *main* as an **array of Strings**.
- If this argument is called **arg** then the elements may be referred to as **arg[0] , arg[1] , arg[2] , etc.**

## Example

*Suppose a compiled Java program called **Copy.class** copies the contents of one file into another the program may allow the user to specify the names of the two files as **command line parameters**:*

```
C:\>java Copy source.dat dest.dat
```

Method main would then access the file names through arg[0] and arg[1] :

```
public class Copy {
    public static void main(String[] arg) throws IOException {
        //First check that 2 file names have been supplied...
        if (arg.length < 2) {
            System.out.println("You must supply TWO file names.");
            System.out.println("Syntax:");
            System.out.println(" java Copy <source> <destination>");
            return;
        }
        Scanner source = new Scanner(new File(arg[0]));
        PrintWriter destination = new PrintWriter(new File(arg[1]));
        //.....
        String input;
        while (source.hasNext()) {
            input = source.nextLine();
            destination.println(input);
        }
        source.close();
        destination.close();
    } //End of main() method
} //End of class Copy
```

Command Line Parameters



# Random Access Files

- Serial access files are simple to handle
- Are quite widely used in small-scale applications
- As a means of providing temporary storage in larger-scale applications.
- However, they do have two distinct disadvantages,
  - (i) We can't go directly to a specific record.
  - (ii) It is not possible to add or modify records within an existing file.  
(The whole file would have to be re-created!)

However, the **speed** and **flexibility** of random access files often greatly outweigh the above disadvantages.

**Random access files** (more meaningfully called **direct access files**) overcome both of these problems, but do have some disadvantages of their own...

- (i) In common usage, all the (logical) records in a particular file must be of the same length.
- (ii) Again in common usage, a given string field must be of the same length for all records on the file.
- (iii) Numeric data is not in human-readable form.

- To create a random access file in Java, we create a **RandomAccessFile** object.
- The constructor takes **two arguments**:
  - a **string** or **File object** identifying the file
  - a **string** specifying the file's **access mode**.
- The access mode argument may be either “r” (for read-only access) or “rw” (for read-and-write access).
- For example:

```
RandomAccessFile ranFile = new RandomAccessFile("accounts.dat","rw");
```

Before reading or writing a record, it is necessary to position the **file pointer** .

We do this by calling method **seek** , which requires a single argument specifying the byte position within the file.

Note that the first byte in a file is byte **0** .

For example:

```
ranFile.seek(500);
```

```
//Move to byte 500 (the 501st byte).
```

In order to move to the correct position for a particular record, we need to know two things:

- the size of records on the file;
- the algorithm for calculating the appropriate position.

The second of these two factors will usually involve some kind of **hashing function** that is applied to the key field.

We shall avoid this complexity and assume that records have keys 1, 2, 3,... and that they are stored sequentially.

However, we still need to calculate the record size.

- For numeric fields, though, the byte allocations are fixed by Java.

**int        4 bytes**

**long      8 bytes**

**Float     4 bytes**

**double   8 bytes**

- Class `RandomAccessFile` provides the following methods for manipulating the above types:

***readInt, readLong, readFloat, readDouble***

***writeInt, writeLong, writeFloat, writeDouble***

- In addition, it provides a method called ***writeChars*** for writing a (variable length) string.
- No methods for reading/writing a string of fixed size are provided.
- So we need to write our own code for this using methods ***readChar*** and ***writeChar*** for reading/writing the primitive type `char`.
- What about string ?

## Example

Suppose we wish to set up an accounts file with the following fields:

- account number ( *long* );
- surname ( *String* );
- initials ( *String* );
- balance ( *float* ).

Java is based on the **Unicode** character set, in which each character occupies **two bytes**.

To allocate 15 (Unicode) characters to surnames and 3 (Unicode) characters to initials.

For surnames 30 (i.e.,  $15 \times 2$ ) bytes and for initials 6 (i.e.,  $3 \times 2$ ) bytes allocated. Long 8 bytes and a float occupies 4 bytes,

then record size =  $(8 + 30 + 6 + 4)$  bytes = **48 bytes**.

Consequently, we shall store records starting at byte positions 0, 48, 96, etc.

The formula for calculating the position of any record on the file is then:

$(\text{Record No.} - 1) \times 48$ .

For example, let *ranAccts* be a object of *RandomAccessFile* for the above accounts file.

Then the code to locate the record with account **number 5** is:

**ranAccts.seek(192);**  $//(5-1) \times 48 = 192$

Since method *length* returns the number of bytes in a file, we can always work out the number of records in a random access file by dividing the size of the file by the size of an individual record.

Consequently, the number of records in file *ranAccts* at any given time = ***ranAccts.length()/48*** .



```

import java.io.*;
import java.util.*;
public class RanFile1
{
    private static final int REC_SIZE = 48;
    private static final int SURNAME_SIZE = 15;
    private static final int NUM_INITS = 3;
    private static long acctNum = 0;
    private static String surname, initials;
    private static float balance;
    //.....
    public static void main(String[] args) throws IOException
    {
        RandomAccessFile ranAccts =
            new RandomAccessFile("accounts.dat", "rw");
        Scanner input = new Scanner(System.in);
        String reply = "y";
    }
}

```

```

do
{
    acctNum++;
    System.out.println("\nAccount number " + acctNum + ".\n");
    System.out.print("Surname: "); surname = input.nextLine();
    System.out.print("Initial(s): "); initials = input.nextLine();
    System.out.print("Balance: "); balance = input.nextFloat();
    //.....
    //Now get rid of carriage return(!)...
    input.nextLine();
    writeRecord(ranAccts); //Method defined below.
    System.out.print("\nDo you wish to do this again (y/n)? ");
    reply = input.nextLine();
} while (reply.equals("y") || reply.equals("Y"));

```

```

System.out.println();
showRecords(ranAccts);

```

```

}

```

```
public static void writeRecord(RandomAccessFile file) throws IOException
{
    long filePos = (acctNum-1) * REC_SIZE; //First find starting byte for current record...
    file.seek(filePos);                    //Position file pointer...
    file.writeLong(acctNum);
    writeString(file, surname, SURNAME_SIZE);
    writeString(file, initials, NUM_INITS);
    file.writeFloat(balance);
}
```

```
public static void writeString(RandomAccessFile file, String text, int fixedSize)
    throws IOException
{
    int size = text.length();
    if (size<=fixedSize)
    {
        file.writeChars(text);

        for (int i=size; i<fixedSize; i++)
            file.writeChar(' ');
    }
    else //String is too long!
        file.writeChars(text.substring(0,fixedSize));
}
```

```
public static void showRecords(RandomAccessFile file) throws IOException
{
    long numRecords = file.length()/REC_SIZE;
    file.seek(0); //Go to start of file.
    for (int i=0; i<numRecords; i++)
    {
        acctNum = file.readLong();
        surname = readString(file, SURNAME_SIZE);
        initials = readString(file, NUM_INITS);
        balance = file.readFloat();
        System.out.printf("" + acctNum+ " " + surname+ " " + initials + " "+ "%.2f
                               %n",balance);
    }
}
```

```
public static String readString(RandomAccessFile file, int fixedSize) throws
    IOException
{
    String value = ""; //Set up empty string.
    for (int i=0; i<fixedSize; i++)
        value+=file.readChar();
    return value;
}
} // End of class RanFile1
```

```
MS-DOS Command Interface for Java
Account number 3.
Surname: Andrews
Initials: PMD
Balance: 1296.43
Do you wish to do this again (y/n)? y
Account number 4.
Surname: Daniels
Initials: RM
Balance: 52.45
Do you wish to do this again (y/n)? n
1      Black      AJ      2310.75
2      Jenkinson P       675.92
3      Andrews   PMD     1296.43
4      Daniels   RM       52.45
D:\>
```