22 Binary Files

1 Binary versus text files

The information in a file may be encoded as either text or binary data. We have already studied text files; now we look at binary files. It is important to understand at the outset that text and binary files are distinguished only in the way that data is encoded and organised on the disk. Whenever we choose to store data as a text file, we might alternatively have chosen to store it as a binary file, and vice versa. We make our choice based on convenience and computational cost for the particular application we have in mind.

Text files have two advantages over binary files: they can be viewed using any text editor, and they are highly portable. By *portable* we mean that a file produced by a program running on a certain machine can be conveniently read from and written to by another program running on a different machine, even if that program has been written in a different language. (Of course, the second program must have physical access to the file, such as over a network.) Text files, however, have higher computational and storage costs. In particular, if they contain lots of non-text data (such as integers, reals, and booleans) they do not use disk space efficiently, and it takes longer to retrieve data. In binary files, data is stored in the same format as internally in the machine, and this brings a space gain: a binary integer occupies 32 bits whereas its decimal textual representation might occupy as many as 100 bits. And a time saving follows: the machine needs to transfer fewer bits when reading or writing information, and it does not need to translate the data into a different format.

Binary files are an alternative to text files. They are most appropriate when the file is composed of a collection of chunks of data, where each chunk consists of information under a

 © Joseph M. Morris Joseph.Morris@computing.dcu.ie

fixed set of headings. A chunk of data in this context is technically called a *record*. For example, a file of student data is composed of a collection of records, one record per student, with each record having information under the headings *name*, *student number*, *courses*, *fees*, etc.

2 Organisation of binary files

A binary file containing the integers 23, 17, 22, 28, and 10 (of type `int`) can be pictured as:

23	17	22	28	10

Each item of data in a file occupies space. Space is measured in either *bits* or *bytes*, where one byte is equivalent to eight bits. A single bit contains either a 1 or a 0. A single integer (of type `int`) occupies 4 bytes, i.e. 32 bits. For example, 23 (of type `int`) happens to be stored as the 32-bit sequence `00000000000000000000000000010111`. A real number (of type `double`) occupies 8 bytes, a character occupies 2 bytes, a boolean occupies 1 byte, and a string occupies twice as many bytes as there are characters in the string. The size of the above file, for example, is 20 bytes, made up of 5 integers at 4 bytes each. Looking microscopically at its contents, we would see the binary sequence

```
0000000000000000000000000001011100000000000000000000000000010001000000000000
0000000000000001011000000000000000000000000000011100000000000000000000000000
00001010
```

There is no limit on the size of a file, other than that imposed by the available space on the disk.

We can identify the position of any item in the file by giving its offset *in bytes* from the start of the file. The offsets are shown in the lower line of the picture below:

23	17	22	28	10
0	4	8	12	16

For example, number 22 in the file has offset 8 because it is preceded in the file by two integers each occupying 4 bytes.

A file created by your Java program can be read by any other Java program, even if it is running on a different machine under a different operating system. This pleasant attribute is not common among programming languages.

The items in a binary file need not all be of the same type. For example, the file depicted below has 5 items, of which four are of integer type integer and one is of string type.

23	17	John	28	10
0	4	8	14	18

However, it is uncommon that we mix types in this way.

Each record may be (and usually is) composed of several pieces of data, not all of the same type. For example, there are three records in the following file, each one consisting of a person's name, age, and sex (`true` for male):

Andre Agassi	29	true	Martina Hingis	18	false	Tim Henman	22	true

In each record, each component is called a *field*. The records in the file depicted above, for example, have three fields.

If you are presented with a binary file prepared by another programmer, it is not possible to discover its structure by examining its contents. You have to be told. For example, if the file is 40 bytes long, it might consist of 10 integers (of type `int`), or 5 reals (of type `double`), or 4 records each of which consists of a three-character string and an integer. If the file actually consists of say, 5 reals, and your program treats it as 10 integers, no program error will arise. However, the integers you read will not be meaningful, and neither will your results.

3 Sequential access

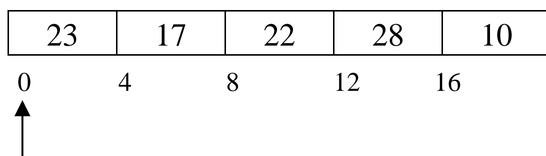
It is not possible to manipulate data directly in a file. If a file is small, we can read it in its entirety into memory, make any changes we want, and write it back to the file. However, many files are very large, much larger than would fit in main memory – think of a file of all the tax payers in a country, for example. We process larger files on a record by record basis such that there is never more than one (or at most a few) records in memory at any time.

If a program reads and/or writes a few individually selected records scattered throughout the file we say that it is processing the file *randomly* or by *random access*. Random processing is the most general file handling technique. However, it is commonly the case that the program simply reads all the records in the file, one after the other starting with the first and ending with the last. For example, we may read all the records in a file of student records in order to determine the average age of the students. We say that such a program reads the file *sequentially* or by *sequential access*. A program may also write to a file sequentially, as when it creates the file by writing record after record, appending each new record to the end of the file. Sequential processing of files is very common, and programming languages provide specially for it. For the present, we concern ourselves solely with sequential processing.

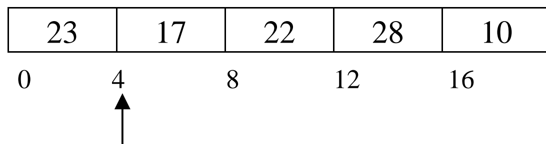
We add a word of caution regarding terminology. Some textbooks attribute the property of being sequential or random to *files* rather than to *file processing*. This is misleading: a binary files can be processed sequentially one day, randomly the next, and again sequentially the following day. There is nothing in a file that marks it as being sequential or random – these terms pertain to how we happen to be processing the file at the moment. In contrast, the property of being a text file or a binary file is fixed once and for all when the file is created.

In computational terms, reading and writing files is a complex and potentially slow process. Fortunately, nearly all the work is done for us behind the scenes by the operating system. To help the operating system work efficiently, the program is expected to announce when processing of a particular file begins – this is called *opening* the file, and when it ends – this is called *closing* the file.

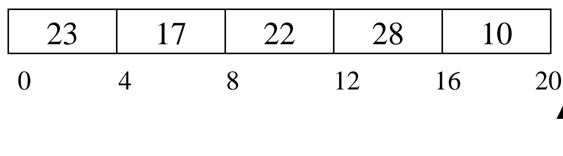
Every file has associated with it a hidden "file pointer" maintained by the run-time system. This is an integer variable (of type `long`) containing a byte offset in the file. The file pointer "points to" a location in the file (or possibly the very end of the file). There is a file pointer for each file being processed in the program. When the file is opened for reading (i.e. made available to the program for reading purposes), the file pointer indexes the start of the file:



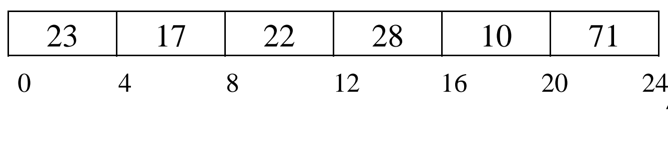
Activity on a binary file, whether reading or writing, always takes place at the location indexed by the file pointer. When an item is read from a file (into a program variable), the item is retrieved from the position indexed by the file pointer, and the file pointer is advanced by the length of the item. After a single integer read operation on the above file, for example, the value 23 is read and the file pointer is positioned as follows:



The next read will retrieve 17, and three further reads will retrieve 22, 28, and 10 in that order. When the last integer is read, the file pointer will have advanced to just after the end of the file:



Reading a file has no affect on its contents. When a file is opened for writing, the file pointer points to just after the end of the file, as in the immediately preceding picture. When we write a record, therefore, the effect is to append it to the file (and the file pointer is advanced by the length of the item written). For example, if we write 71, say, to the file depicted above, the result is:



It is an error to attempt to read a file when the file pointer is positioned beyond the end of the

file. The classes provided by Java for handling files are summarised in the following table

File activity	*Text files*	*Binary sequential files*
Input from file	`Scanner`	`DataInputStream`
Output to file	`PrintWriter`	`DataOutputStream`

To make these classes available to the program, the program should begin with `import java.io.*;`.

4 Writing to binary files: DataOutputStream

Java provides class **`DataOutputStream`** for writing to binary files sequentially:

DataOutputStream(new FileOutputStream(String)) throws IOException

`new DataOutputStream(new FileOutputStream(s))` causes an empty binary file called `s` to be created on the disk. For example, the following creates an empty binary file called `diskData` on the disk

```
DataOutputStream myFile = new DataOutputStream(
                                    new FileOutputStream("diskData"));
```

`FileOutputStream` has no use other than as an intermediary in the construction of a `DataOutputStream` object, and you need not concern yourself with it further. The name chosen for the file (above `diskData`) can be any name allowed by the operating system. If it is to be created in a folder other than that in which the program resides, a path name should be used (see text files). For example, if instead of `diskData` above we write `C:/myDirectory/diskData`, the file called `diskData` will be created in folder `myDirectory` on drive `C`. A word of warning: if a file called `diskData` already exists in the relevant folder it will be deleted and a new empty one created. After the declaration above, variable `myFile` references an object of type `DataOutputStream`, but we also loosely use the name `myFile` in explanatory text to describe the file itself. The `DataOutputStream` object that is created contains, amongst other things, a file pointer which is initialised to 0 (i.e. it points to the start of the file).

If you want to write to an existing binary file without first erasing its contents, you can use a `DataOutputStream` constructor that has an extra boolean parameter which is always `true`:

DataOutputStream(new FileOutputStream(String, boolean)) throws IOException

`new DataOutputStream(new FileOutputStream(s,true))` makes the file named `s` on the disk available for output. The inclusion of the boolean `true` informs the system that the file already exists – if you omit it, a *new* file called `s` will be created and the original one will be deleted! An example of its use is:

```
DataOutputStream myFile = new DataOutputStream(
```

```
                                  new FileOutputStream("diskData", true));
```

The file pointer initially indexes the end of the file and so all text written will be appended. Actually, it is safe to include the boolean `true` in the constructor even when creating new files – if the file does not already exist, an empty one of that name will be created.

`DataOutputStream` provides the following methods for writing data:

```
void writeInt(int) throws IOException
void writeDouble(double) throws IOException
void writeBoolean(boolean) throws IOException
void writeChar(int) throws IOException
void writeChars(String) throws IOException
void writeUTF(String) throws IOException
```

`f.writeInt(k)` appends `k` to file `f`, and advances the file pointer by 4 (4 being the length of a value of type `int`). **`f.writeDouble(x)`** appends `x` to file `f`, and advances the file pointer by 8. **`f.writeBoolean(b)`** appends `b` (in binary form) to file `f`, and advances the file pointer by 1. **`f.writeChar(c)`** appends character `c` to `f`, and advances the file pointer by 2. For a technical reason that we do not go into here, the argument of `writeChar()` is declared to be an integer although it is used to write a character. **`f.writeChars(s)`** appends `s` to `f`, and advances the file pointer by `s.length()`. **`f.writeUTF(s)`** also appends `s` to `myFile`, but the string is encoded in the file in an unusual way; we will discuss it further when we meet `readUTF()` in the next section.

`DataOutputStream` also provides:

```
void close() throws IOException
```

`f.close()` closes file `f`. It is important to close files at the end of output, as otherwise data may be lost.

Example 1: creating a file of integers

The program below creates a binary file called `IntFile` (on disk) which contains 50 random integers each in the range 0 to 19:

```
import java.io.*;
class SeqInts {
   public static void main(String[] args) {
      try {
         DataOutputStream out = new DataOutputStream(
                                     new FileOutputStream("IntFile"));
         for (int i=0; i<50; i++) {
            int n = (int)(Math.random()*20);
            out.writeInt(n);
         }
```

```
            out.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Note the importation of `java.io.*`. After the program has executed, you should see a file called `IntFile` in the same directory as the program. If you examine the properties of the file, you will see that its size is 200 bytes (50 integers at 4 bytes each). However, if you try to inspect the contents with a text editor you will see garbage. To display its contents you must write your own program to do so.

5 Reading binary files: DataInputStream

Java provides class **`DataInputStream`** for reading binary files sequentially. The `DataInputStream` constructor is:

```
DataInputStream(new FileInputStream(String)) throws FileNotFoundException
```

`new DataInputStream(new FileInputStream(s))` opens for reading an existing binary file called `s` on the disk. For example, the following opens a file called `diskData`:

```
DataInputStream myFile = new DataInputStream(
                                    new FileInputStream("diskData"));
```

`FileInputStream` has no use other than as an intermediary in the construction of a `DataInputStream` object, and you need not concern yourself with it further. If the file exists in a folder other than that in which the program resides, a path name should be supplied. After the declaration above, variable `myFile` references an object of type `DataInputStream`, but we also loosely use the name `myFile` in explanatory text to describe the file itself. The `DataInputStream` object that is created contains, amongst other things, a file pointer which is initialised to 0 (i.e. it points to the start of the file).

`DataInputStream` provides the following methods for reading data:

```
int readInt() throws IOException, EOFException
double readDouble() throws IOException, EOFException
boolean readBoolean() throws IOException, EOFException
char readChar() throws IOException, EOFException
String readUTF() throws IOException, EOFException
```

`f.readInt()` reads and returns an integer from file `f`. The integer is read from the file at the position indicated by the file pointer, and the file pointer is advanced by the length of an integer (4 bytes). It is the programmer’s responsibility to ensure that an integer was written to that location. If something other than an integer was written, the value returned is meaningless.

If fewer than four bytes remain from the file pointer to the end of the file, an end-of-file exception is generated. **f.readDouble()**, **f.readBoolean()**, and **f.readChar()** behave analogously as their name suggests. **f.readUTF()** reads a string provided it has been written to f using writeUTF(). A string as written by writeUTF()includes within it an encoding of its length so that readUTF() knows just how many bytes to retrieve; the encoding is called UTF encoding. If you use readUTF(), the string must have been written using writeUTF(), not writeChars(). Conversely, if a string is written to a file using writeUTF(), it can only be retrieved using readUTF(). UTF encoding is a little quirky in that not all characters are coded with the same number of bits and so two strings with the same number of characters might have different sizes in the file. This poses no problem for sequential file processing, but may slightly complicate random file processing. Actually, the UTF encoding of strings whose characters are drawn from the ASCII character set (which includes all the characters on a Western keyboard) are coded at one byte per character plus 2 bytes (an exception is the null character '\0' which occupies 2 bytes). For example, the string "horse" occupies 7 bytes when encoded in UTF: one byte for each of the 5 characters in "horse" plus two bytes to store the length of the string. We will give an example of using readUTF() and writeUTF() in the next section.

Rather than detecting the end of the file by trapping an end-of-file exception, we can discover whether the file pointer is positioned at the end of the file using method available() in DataInputStream:

> int available() throws IOException

When **f.available()**returns 0, the file pointer of f is at the end of the file.

DataInputStream provides:

> void close() throws IOException

f.close() closes file f after which it can no longer be read from (unless opened again). No harm is done if the file is not closed, but closing files reduces demand on system resources.

DataInputStream provides no method to read a string that has been written using writeChars(). Instead, we have to read the string character-by-character into an array, and then convert it to a String type. The following methods in class String create strings from a character array:

> static String valueOf(char[])
> static String valueOf(char[], int, int)

String.valueOf(b) yields the string obtained by concatenating the characters in array b, i.e. the string b[0]+b[1]+b[2]+.... **String.valueOf(b,i,j)** yields the string obtained by concatenating the characters b[i]+b[i+1]+...+b[j-1] (i and j denote integers in the range 0≤i≤j≤b.length). For example, the following code reads a 25-character string from file in that was written with writeChars():

```
char[] b = new char[25];
for (int i=0; i<b.length; i++)
    b[i] = in.readChar();
name = String.valueOf(b);
```

Example 1: searching an integer file

In the following program, we count the number of occurrences of a particular integer in the file of integers called `IntFile` created in the previous example. The integer being sought is passed as a command-line argument. For example, executing

```
java IntsLookup 27
```

will print the number of occurrences of 27 in `IntFile`.

```
import java.io.*;
class IntsLookup {
   public static void main(String[] args) {
      try {
         DataInputStream in  = new DataInputStream(
                        new FileInputStream("IntFile"));
         int x = Integer.parseInt(args[0]);  // x is the search value
         int count = 0;  // number of occurrences of x
         while (in.available()>0) {
            int k = in.readInt();
            if (k==x) count++;
         }
         in.close();
         System.out.println(count + " occurrences of " + x);
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Although we happen to know the number of integers in `IntFile` (because we created it ourselves), the program does not exploit that fact. Instead, it reads integers until it reaches the end of the file, and this makes the program applicable to files of any size. We could alternatively (but less attractively) have determined the end of the input by trapping an end-of-file exception – just replace the loop above with the following:

```
boolean moreData = true;
while (moreData) {
    try {
        int k = in.readInt();
        if (k==x) count++;
```

```
        }
        catch(EOFException e) {
            moreData = false;
        }
    }
```

6 Example: binary files of complex records

When we write an object to a file, it suffices to write just the values of its instance variables. Later when we read the file, we can reconstruct the object. We illustrate this on a data base of personnel records, each record consisting of a person's name, age, and sex (recorded as a boolean in which `true` represents male). We shall make a suite of two programs, one of which creates a personnel file, and the other of which displays the contents of the file. They both share class `Person`, and so we write this separately so it can be made available to both:

```
import java.io.*;
class Person {
   private String name;
   private int age;
   private boolean isMale;

   Person() {
   }

   Person(String s, int a, boolean m) {
      name = s; age = a; isMale = m;
   }

   void put(DataOutputStream out) {
   // write attributes to file
      try {
         out.writeUTF(name); out.writeInt(age); out.writeBoolean(isMale);
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }

   void get(DataInputStream in) {
   // read attributes from file (assume not at end of file)
      try {
         name = in.readUTF(); age = in.readInt(); isMale = in.readBoolean();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
```

```
    void put(){ // display attributes on screen
        System.out.println(name + " " + age + " " + isMale);
    }
}
```

Note that `put(DataOutputStream)` and `get(DataInputStream)` do not specify the record being read or written to -- this is determined by the file pointer. The class is designed to support a pattern of use where the file is accessed sequentially record by record during a writing phase, or sequentially record by record during a reading phase, without reading and writing phases overlapping. Observe also that `get()` retrieves the items in each record in precisely the same order in which they are written to the file, as it must.

Here is a program to create a trivial file of `Person` records:

```
import java.io.*;
class CreatePersons {
    public static void main(String[] args) {
        try {
            DataOutputStream out = new DataOutputStream(
                                new FileOutputStream("PersonFile"));
            (new Person("Andy Murray", 20, true)).put(out);
            (new Person("Ana Ivanovich", 23, false)).put(out);
            (new Person("Rafael Nadal", 22, true)).put(out);
            out.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

After the program has been executed, file `PersonFile` will exist in the same directory as the program. It will look like:

Andy Murray	20	true	Ana Ivanovich	23	false	Rafael Nadal	22	true

The following program displays its contents:

```
import java.io.*;
class DisplayPersons {
    public static void main(String[] args) {
        try {
            DataInputStream in = new DataInputStream(
                        new FileInputStream("PersonFile"));
```

```
            Person p = new Person();
            while (in.available()>0) {
               p.get(in);  p.put();
            }
            in.close();
         }
         catch (IOException e) {
            e.printStackTrace();
         }
      }
   }
```

Simple interrogation of complex files OPTIONAL

When we write objects to a file as above, the file retains no trace of the fact that the records derive from objects. Hence if we need to carry out some simple interrogation of the data, it is possible to do so without recreating the objects. To illustrate this, the following program calculates the average age of people in the personnel file we created above, without creating `Person` objects:

```
import java.io.*;
class AverageAge {
   public static void main(String[] args) {
      try {
         DataInputStream in  = new DataInputStream(
                              new FileInputStream("PersonFile"));
         int totalAge = 0; // total of ages
         int numRecs = 0; // number of records read
         while (in.available()>0) {
            in.readUTF(); // skip name
            int  k = in.readInt(); // read age
            in.readBoolean(); // skip sex
            totalAge = totalAge+k;
            numRecs++;
         }
         in.close();
         System.out.println("Average age: " + totalAge/numRecs);
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Each execution of the loop body retrieves a single record. The name and sex information is

read to advance the file pointer appropriately, but the value returned in each case is discarded.

For an alternative way to write objects to a file, see serialization.

7 File information: File

Files have properties, such as a name, a size (measured in bytes), whether or not it is readable and/or writable, whether it is a normal file or a folder (directory), and so on. Whether a file is readable or writable has nothing to do with the contents or the file or its structure or any error situations, but is a property of the file as an entity in the system. For example, a file may be unwritable if it exists on a floppy which is write-protected, or it may be unreadable because the user attempting to read it does not have permission to do so (it could be a file of passwords, for example). Java provides the class **`File`** for discovering and modifying properties of files. It has a simple constructor:

```
File(String)
```

`new File(s)` creates an object of type `File` pertaining to a file known to the operating system as `s`. For example, the declaration

```
File myInfo = new File("diskData");
```

creates and assigns to variable `myInfo` an object of type `File` pertaining to file `diskData`. A path name can be supplied in place of a simple file name. It is possible that a file called `diskData` does not exist; that is allowable, but a new file of that name is *not* created. Programs which use `File` should import `java.io.*`.

Class `File` includes the following methods for discovering the status of a file:

```
boolean exists()
boolean isDirectory()
boolean isFile()
boolean canRead()
boolean canWrite()
long length()
```

`f.exists()` returns a boolean indicating whether the file associated with `f` exists. **`f.isDirectory()`**, **`f.isFile()`**, **`f.canRead()`**, and **`f.canWrite()`**, return booleans indicating whether the file is a directory, a normal file, is readable, and is writable, respectively. A *normal* file is, for all practical purposes, a file that it is not a directory. **`f.length()`** returns the length of the file associated with `f` in bytes. All these methods can be invoked without error whether or not the file exists. The following methods of `File` are used to change the status of files:

```
boolean renameTo(File)
boolean delete()
```

`f.renameTo(fnew)` changes the name of the file associated with `f` to the name associated with `fnew`. Note that the new name must be supplied within an object `fnew` of type `File`. For example, `myFile.renameTo(new File("newFile"))` changes the name associated with `myFile` to `newFile`. **`f.delete()`** deletes the file associated with `f`. Both methods returns a boolean indicating whether they completed successfully.

Example 1: deleting files

The following program deletes files whose names are keyed in by the user. It takes care not to delete directories, and reports on the success or otherwise of each deletion.

```
import java.io.*;
class DeleteFiles {
   public static void main(String[] args) {
    System.out.print("Delete file: ");  // prompt user for file name or end of input
    while (!Console.endOfFile()) {
        String fileName = Console.readString(); // read name of file to be deleted
        File file = new File(fileName);
        if (!file.exists())
            System.out.println("Cannot find file " + fileName);
        else if (file.isDirectory())
            System.out.println("Cannot delete directory " + fileName);
        else {
            boolean ok = file.delete();
            if (ok) System.out.println(fileName + " deleted");
            else System.out.println("Cannot delete file " + fileName);
           }
           System.out.print("Delete file: "); // prompt for file name or end of input
     }
   }
}
```

8 Copying files OPTIONAL

We write a general program to copy a file, regardless of its structure. We will take care to check that the source and destination files are reasonable in all respects. For example, the program should check that the source file exists, and that if the destination file exists, it is acceptable to overwrite it. The source and destination files are supplied in the command line. A typical invocation is

```
java Copy source destination
```

This creates a file called `destination` that is a copy of `source` (which continues to exist afterwards). We have no interest in the structure of the source file, and for copying purposes view it only as a sequence of bytes. Java provides an integer type, called `byte`, for which the encoding is 8-bit. Apart from accommodating a narrower range of integers, `byte` is otherwise

identical to `int`. Variables of type `byte` can store integers in the range -2^7 to 2^7-1 (-128 to 127, inclusive). `DataInputStream` provides the following method for reading bytes:

int read(byte[]) throws IOException

`f.read(b)` fills array `b` with 8-bit integers read from file `f`, and advances the file pointer by the number of bytes read. If fewer than `b.length` bytes are available, it reads as many as are available. The number of bytes read is returned. If no more data is available, -1 is returned. `DataOutputStream` provides an analogous method for writing integers of type `byte` to a file:

void write(byte[], int, int) throws IOException

`f.write(b,j,k)` writes b[j..k-1], i.e. `b[j]`, `b[j+1]`, `b[j+2]`, ..., `b[k-1]` (`k-j` bytes in all). Neither `read()` nor `write()` take cognisance of the underlying structure of the data: even if the file consists of a collection of personnel records, say, `read()` will nevertheless read it as if it had been created as a file of 8-bit integers. The program to copy a file follows:

```
import java.io.*;
class Copy {
   public static void main(String[] args) {
      // Check for two arguments
      if (args.length != 2) {
         System.out.println("Usage: java Copy file1 file2");
         return;
      }
      // Check input file exists as a normal readable file
      File fin = new File(args[0]);
      if (!fin.exists()) {
         System.out.println("Cannot find file " + args[0]);
         return;
      }
      if (!fin.isFile() || !fin.canRead()) {
         System.out.println("Cannot copy file " + args[0]);
         return;
      }
      // Check output file does not exist, or is to be overwritten
      File fout = new File(args[1]);
      if (fout.exists()) {
         System.out.print("Overwrite existing file " + args[1] + "? (y/n) ");
         String r = Console.readString();
         if (r=="" || r.charAt(0)=='n' || r.charAt(0)=='N')
            return;
         if (!fout.canWrite()) {
            System.out.print("Cannot overwrite existing file " + args[1]);
            return;
         }
```

```
        }
        // Copy file
        try {
           DataInputStream in = new DataInputStream(
                                          new FileInputStream(args[0]));
           DataOutputStream out = new DataOutputStream(
                                          new FileOutputStream(args[1]));
           byte[] block = new byte[1024];
           while(in.available()>0) {
                 int k = in.read(block);
                 out.write(block, 0, k);
           }
           in.close(); out.close();
        }
        catch (IOException e) {
             System.out.println ("Error during copying");
        }
     }
  }
```

The program reads and writes in blocks of 1024 bytes (= 1K), but any size would work. The larger the block size, the fewer invocations of `read()` are needed. However, because of the way in which file systems are implemented, there is an upper limit beyond which no advantage is to be gained. A 1K block size works well in practice. The program copies any kind of file, whether a text file or a binary file. An alternative way to detect whether the end of the source file has been reached is to examine the value returned by `in.read(block)` – when this is less than 1024 all the data has been read.

9 Updating binary files sequentially OPTIONAL

Binary files are usually long-lived, and undergo many changes during their lifetime. For example, every time a student passes a course or changes address the file of student records has to be updated. The most commonly used technique for updating a large binary file sequentially is, oddly enough, not to update the file but to create an entirely new one. It turns out that this is more computationally efficient. Suppose, for example, that we have a personnel file and we want to increase the salaries of, say, non-manual workers by 5%. This is done by reading each record of the personnel file in turn into memory, and inspecting it. If it pertains to a non-manual worker we write it to the new file, having first increased the salary by 5% (in our copy of the record in memory). Otherwise we write it to the new file unchanged. When all the records have been processed in this way, we delete the original file and re-name the new one with the original name. Observe that during updating we are working simultaneously on two files: the original file which is being read, and the new file which is being written to. Operating systems can deal very efficiently with files which are being read only, or being written to only, but which do not intermix reading and writing. Observe also that the program need only ever

store a single record in memory at any one time, regardless of how large the file is. This technique is so space-efficient that it is usually used for all but the smallest files (very small files are just read into an array in their entirety, amended, and re-written).

Deleting records from a file is similar: we construct a new file with the chosen records deleted. If, for example, we want to delete from a file of student records all students who have not paid their fees, we create a new file that is the same as the original except that it omits the records of students who have not paid their fees. We do this by reading the records of the student file sequentially, one by one.

It is not efficient to create a new file if only a single record needs amending, and in that case we would prefer to update the record using random processing techniques.

Example 1: updating personnel file sequentially

We write a program to illustrate sequentially updating a binary file. The program modifies the database of persons we created above, by converting the names of all persons in the file to uppercase. This means that we will create a new file that is identical to the original, except that all names are in upper case. First we have to include in class `Person` (see above) a method which changes a name to upper case:

```
void upName() { // convert name to uppercase
  name = name.toUpperCase();
}
```

The program to update the database follows:

```
class UpdatePersons {
   public static void main(String[] args) {
      final String fileName = "PersonFile"; // name of original file
      final String tempName = "TempFile"; // temporary name of new file
      try {
          DataInputStream in = new DataInputStream(
                         new FileInputStream(fileName)); // original (input) file
          DataOutputStream out = new DataOutputStream(
                         new FileOutputStream(tempName)); // new (output) file
          Person p = new Person();
           // Read each record from original file, update it, and append to new file
          while (in.available()>0) {
             p.get(in);  p.upName();  p.put(out);
          }
          in.close(); out.close();
      }
      catch (IOException e) {
          System.out.println("Updating failed"); return;
```

```
        }
        // Delete original file, and rename new file
        File oldFile = new File(fileName);
        File newFile = new File(tempName);
        boolean ok1 = oldFile.delete();
        boolean ok2 = newFile.renameTo(oldFile);
        if (!ok1 || !ok2)
            System.out.println("Warning: New file may be called " + tempName);
    }
}
```

If a file called `TempFile` already exists, the program will inadvertently delete it. It would be better to generate a file name that is guaranteed to be new. We can do this by replacing the code `final String tempName = "TempFile"` with the code below. It generates an original name of the form `Temp452365`, where the final digits are randomly generated:

```
String tempName = "Temp" + (int)(Math.random()*1000000);
File theFile = new File(tempName);
while (theFile.exists()) { // bad luck -- try again!
    tempName = "Temp" + (int)(Math.random()*1000000);
    theFile = new File(tempName);
}
```