23 Writing Binary Files

1 Binary versus text files

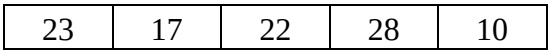
The information in a file may be encoded as either text or binary data. Text and binary files are distinguished only in the way that data is encoded and organised in the file. Whenever we choose to store data as a text file, we might alternatively have chosen to store it as a binary file, and vice versa. 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. 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 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:

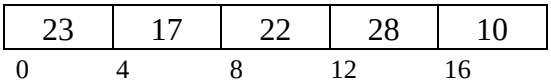


Each item of data in a file occupies space. Space is measured in either *bits* or *bytes*, where one byte is equivalent to eight bits. A single bit contains either a 1 or a 0. A single integer (of type `int`) occupies 4 bytes, i.e. 32 bits. For example, 23 (of type `int`) happens to be stored as the 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 two plus as many bytes as there are characters in the string (when encoded in a format called *UTF* which we explain later) . 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

```
00000000000000000000000000010111000000000000000000000000000010001000000000000
00000000000000010110000000000000000000000000000011100000000000000000000000000
00001010
```

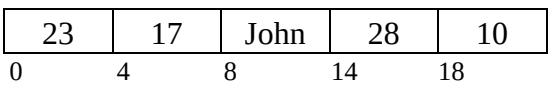
There is no limit on the size of a file, other than that imposed by the available space on the storage medium.

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:



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

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.



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):

Roger Federer	28	true	Venus Williams	29	false	Andy Murray	22	true

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

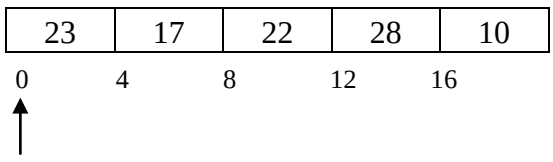
If you are presented with a binary file prepared by another programmer, it is not possible to discover its structure by examining its contents. You have to be told. For example, if the file is 40 bytes long, it might consist of 10 integers (of type `int`), or 5 reals (of type `double`), or 4 records 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 Navigating binary files

File contents are changed by reading records into variables in the program, making the change, and then writing the changed variables. If a file is small, we can read it in its entirety into memory, make any changes we want, and write it all 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 by keeping a relatively small number of records in the program's variables at any one time, often no more than one or two.

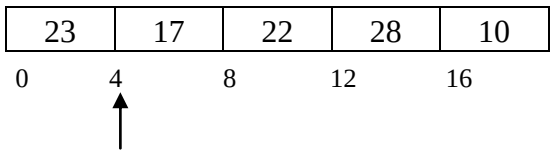
In computational terms, reading and writing files is a complex and potentially slow process. Fortunately, nearly all the work is done for us behind the scenes by the operating system. To help the operating system work efficiently, the program is expected to announce when processing of a particular file 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 just after the 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:

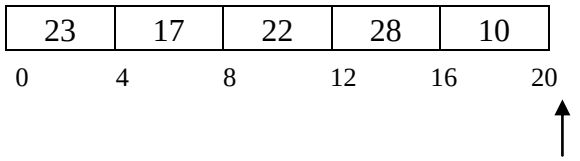


Activity on a binary file, whether reading or writing, always takes place at the location indexed by the file pointer. When an item is read from a file (into a program variable), the item is

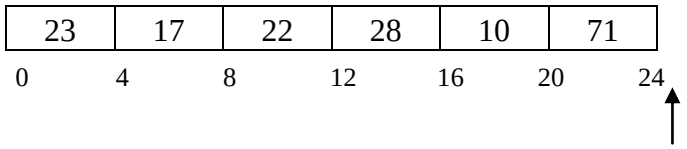
retrieved from the position indexed by the file pointer, and the file pointer is advanced by the length of the item. After a single integer read operation on the above file, for example, the value 23 is read and the file pointer is positioned as follows:



Reading a file has no affect on its contents. 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:



When we write a record when the file pointer is positioned just after the end, 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:



When a file is opened for writing, the file pointer is positioned either at the start of the file or just after the end (as in the immediately preceding picture), depending on the Java class used to open the file. We will be using class `RandomAccessFile` for which the pointer is positioned at the start.

4 Writing to binary files

Java provides a choice of classes for accessing binary files; we will use `RandomAccessFile` (an alternative is `SeekableByteChannel` which is more general but significantly less convenient). To open a binary file for writing we create an instance of class **`RandomAccessFile`**:

RandomAccessFile(*String*, "*rw*") throws FileNotFoundException

`new RandomAccessFile(s,"rw")` makes the file named `s` on the disk or external medium available to the program for writing or reading; for the moment we focus on writing. For example,

```
RandomAccessFile myFile = new RandomAccessFile("data.dat", "rw");
```

If a file of that name does not already exist, one will be created if possible. The name of the file

can be any name allowed by the operating system. A path name should be supplied for files that reside in a folder other than that in which the program resides (see text files). For example, if instead of `data.dat` above we write `C:/myDirectory/data.dat` (running under Windows) it refers to a file called `data.dat` in folder `myDirectory` on drive `C`. Instances of `RandomAccessFile` contains, amongst other things, a file pointer; it initially points to the start of they file, i.e. at offset 0. After the declaration above, variable `myFile` references an object of type `RandomAccessFile`, but we also loosely use the name `myFile` in explanatory text to describe the file itself. `RandomAccessFile` resides in the `java.io` package and must be imported.

`RandomAccessFile` 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)` writes `k` to file `f`, and advances the file pointer by 4 (4 being the length of a value of type `int`). **`f.writeDouble(x)`** writes `x` to file `f`, and advances the file pointer by 8. **`f.writeBoolean(b)`** writes `b` (in binary form) to file `f`, and advances the file pointer by 1. **`f.writeChar(c)`** writes 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)`** writes each character in `s` to `f` in turn, and advances the file pointer by `s.length()`. **`f.writeUTF(s)`** writes `s` to `f`, where the string is encoded in the file according to UTF encoding. UTF encoding encodes the length of the string as well as its constituent characters, making it easier to read from the file subsequently. In UTF, all the familiar Western characters (i.e. the ASCII characters) are stored as a single byte (an exception is `'\0'` which occupies two bytes), so the length of a UTF string for our purposes equals the number of characters in the string plus 2 for the encoding of the string length. For example, the string "horse" occupies 7 bytes when encoded in UTF. Strings are more commonly written using `writeUTF()` rather than `writeChars()`.

`RandomAccessFile` 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. Closing files also releases memory for re-cycling.

Example 1: creating a file of integers

The program below creates a binary file called `ints.dat` which contains 50 random integers each in the range 0 to 19:

```
import java.io.*;

class MakeIntsFile {
   public static void main(String[] args) {
      try {
         RandomAccessFile out = new RandomAccessFile("ints.dat", "rw");
         for (int i=0; i<50; i++)  {
            int n = (int)(Math.random()*20);
            out.writeInt(n);
         }
         out.close();
      }
      catch (IOException e) { System.out.println("Could not write to file.");}
   }
}
```

After the program has executed, you should see a file called `ints.dat` 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 what appears to be garbage. To display its contents you must write your own program to do so.

Example 2: creating a file of complex records

The program below creates a binary file called `persons.dat` containing the records of persons whose details are read from the standard input. Each line read on the standard input contains the person's name (forename only), age, and sex, such as

```
Bill 23 male
```

We choose to store the person's sex in the file as a boolean, because that occupies less space.

```
import java.io.*;

class CreatePersons {
   public static void main(String[] args) {
        try {
         RandomAccessFile out = new RandomAccessFile("persons.dat","rw");
         while (!Console.endOfFile()) {
            String name = Console.readToken();  int age = Console.readInt();
            String sex = Console.readToken();
            out.writeUTF(name));  out.writeInt(age);
            out.writeBoolean(sex.charAt(0)=='m');
         }
         out.close();
        }
```

```
            catch (IOException e) { System.out.println("Could not write to file"); }
        }
    }
```