22 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:

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 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

```
0000000000000000000000000001011100000000000000000000000000010001000000000000
0000000000000001011000000000000000000000000000011100000000000000000000000000
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:

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.

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):

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.

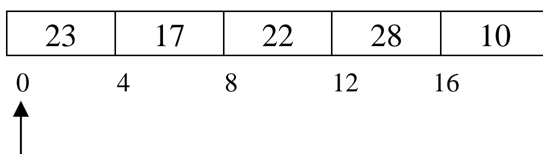
If you are presented with a binary file prepared by another programmer, it is not possible to discover its structure by examining its contents. You have to be told. For example, if the file is 40 bytes long, it might consist of 10 integers (of type `int`), or 5 reals (of type `double`), or 4 records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

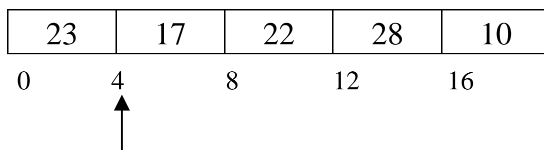
In computational terms, reading and writing files is a complex and potentially slow process. Fortunately, nearly all the work is done for us behind the scenes by the operating system. To help the operating system work efficiently, the program is expected to announce when processing of a particular file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

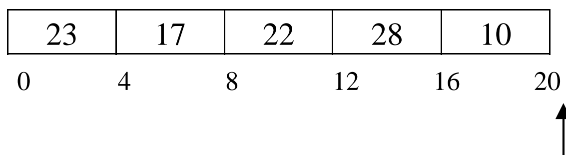


Activity on a binary file, whether reading or writing, always takes place at the location indexed by the file pointer. When an item is read from a file (into a program variable), the item is retrieved from the position indexed by the file pointer, and the file pointer is advanced by the length of the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

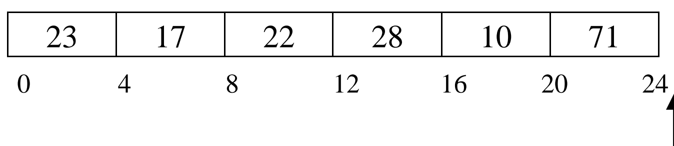
value 23 is read and the file pointer is positioned as follows:



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

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. We explain this further below. Strings are usually 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"); }
   }
}
```

5 Reading binary files

To open a binary file for reading, we create an instance of class **`RandomAccessFile`** in the

following form:

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

new RandomAccessFile(s,"r") makes the file named s on the disk or external medium available to the program for reading. Actually, we can use "rw" instead of "r" above, but if we only intend to read then "r" is best as it protects the file from accidental writing, and it is more efficient. The rules governing file names are as given above. The file pointer is initially set at offset 0. RandomAccessFile 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(). As a string written by writeUTF() includes within it an encoding of its length, readUTF() knows just how many bytes to retrieve. 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(). 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.

RandomAccessFile provides the following methods for managing files:

long length() throws IOException
void setLength(*long*) throws IOException
long getFilePointer() throws IOException
void seek(*long*) throws IOException

f.length() returns the current size of the file in bytes, and **f.getFilePointer()** returns the current value of the file pointer (the return type is long in each case because very large files may have a length greater than can be expressed in 32 bits). There is no simple

method to determine if we've reached the end of the file when we are reading all the records of file `f` one after the other. So we code it ourselves as `f.getFilePointer()<f.length()` (or, less attractively, we could explicitly catch an end-of-file exception). **`f.setLength(n)`** truncates the file to the first `n` bytes; its use is limited mostly to `f.setLength(0)` to make the file empty.

`f.seek(n)` causes the file pointer of file `f` to be positioned `n` bytes from the start of the file. If you want to read or write at an offset different from that of the current value of the file pointer, you must first invoke `seek()`. For example, suppose `f` references a `RandomAccessFile` where the records are integers. Then the following reads the first and third integer from `f`:

```
f.seek(0);  int j = f.readInt();
f.seek(8);  int k = f.readInt();
```

We write 8 in `f.seek(8)` above because integers occupy 4 bytes, and hence the third integer is located 8 bytes from the start. `f.seek(f.length());` positions the file pointer just after the end of the file, ready to append new data.

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 `ints.dat` created in an example above. 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 `ints.dat`.

```
import java.io.*;

class IntsLookup {
   public static void main(String[] args) {
      try {
         RandomAccessFile in  = new RandomAccessFile("ints.dat", "r");
         int x = Integer.parseInt(args[0]);  // x is the search value
         int count = 0;  // number of occurrences of x
         while (in.getFilePointer()<in.length()) {
            int k = in.readInt();
            if (k==x) count++;
         }
         in.close();
         System.out.println(count + " occurrences of " + x);
      }
      catch (IOException e) { System.out.println("Could not read file"); }
   }
}
```

Although we happen to know the number of integers in `ints.dat` (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 integer files of any size.

Example 2: Interrogating a file of complex records

The following program calculates the average age of people in the persons file `persons.dat` we created above:

```
import java.io.*;

class AverageAge {
   public static void main(String[] args) {
      try {
         RandomAccessFile in  = new RandomAccessFile("persons.dat","r");
         int totalAge = 0;  // total of ages
         int numRecs = 0;  // number of records read
         while (in.getFilePointer()<in.length()) {
            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) { System.out.println("Could not read file"); }
   }
}
```

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.

Some pitfalls

Opening a *text file* for writing in the standard way deletes an existing file of the same name if one happens to exist. However, that is not the case when using `RandomAccessFile`; in this case we can use `setLength(0)` to erase the file contents.

If we use `RandomAccessFile` with the intention of creating a new file, but coincidentally a file of exactly the same name exists, we will end up inadvertently overwriting the contents of the existing file. If necessary, we can use class `File` (see below) to check for the existence of a file of the same name.

When we open an existing *text file* for the purposes of appending new data, the file pointer is automatically positioned just after the end of the file. Using `RandomAccessFile` to open a

binary file for the purposes of appending new data, however, leaves the file pointer positioned at the start of the file. We may need to use `seek()` to position the file pointer as we wish.

Finally, the Java documentation does not actually specify the initial value of the file pointer when using `RandomAccessFile`. However, the current implementation positions it at the start of the file. If you needed to be really sure of that for all possible future implementations of Java, you might take the precaution of executing `seek(0)` after the file has been opened. But such caution is not needed for most applications.

6 Reading and writing binary files

Many on-line applications require us to read and write to the file arbitrarily during the program. For example, the membership database of a club must allow the user to key in a membership number and have his or her details displayed on the screen, and also allow a member's details to be changed. Locating a particular record in a file is often made easier by ensuring that all the records are of the same size. The *n*'th record then begins at offset *n* times the record size in bytes, assuming the record count starts at 0. We achieve this by ensuring that each record has a fixed number of fields, and each constituent field is of a fixed size.

In the case of entries in fields of type `String`, we fix on some maximum string size and pad shorter entries artificially with additional blanks. For example, suppose a particular field in the record of a personnel file is to contain an employee's name. We decide on a maximum length for persons' names (40 characters, say) and instead of inserting `"Bill Smith"`, say, we insert `"Bill Smith                              "`. In this case we decide on a length of 40 because that is sufficiently large to accommodate the longest name we think likely to arise.

Method `format()` in class `String` provides an easy way to pad a string with blanks:

static String format(*String*, *Object, ...*)

It works exactly like. `System.out.printf()` except that it *computes* a string rather than prints it. For example, `String.format("%-40s",name)` returns a new string by padding string `name` on the right with as many blanks as needed for a length of 40 (`name` itself is not changed, of course). When the string is subsequently read from the file, any trailing blanks are easily removed using `trim()`. If it is possible for a string to have trailing blanks as part of the string proper, some extra care is needed.

Example: club membership

We make a small database of members belonging to a racquet club. The database contains one record per member, consisting of the member's name (forename plus surname), preferred sport (tennis or squash), and membership number. Membership numbers are allocated sequentially starting at 1000 (so the 5th person to join, say, will have membership number 1004). The user interacts with the database by typing either a *query* or an *addition* on a line. A query consists of

just a membership number and results in the member's details being displayed. An addition consists of a name (forename plus surname) and preferred sport, and results in that person being admitted to membership and added to the database. The new member's membership number is displayed. The following is a typical interaction:

```
Welcome to the membership data base!

Jill Ryan tennis
Membership number: 1000

Willy Smith squash
Membership number: 1001

1000
Jill Ryan (tennis)

Fred Jones tennis
Membership number: 1002

1001
Willy Smith (squash)

1003
Can't find that membership number
```

The database resides in a file (rather than in an array, say) to ensure it carries over from run to run of the program. Although the program is complex enough to warrant the use of classes and methods in its design, we don't use them so that the file handling code is easier to pick out.

```
import java.io.*;

class Club {
    public static void main(String[] args)  {
        final String fileName = "members.dat"; // members file
        final int nameLen = 40; // max length of member's name;
        final int recordSize = 2+nameLen/*name*/+2/*sport*/+4/*membership no.*/;
        final int firstMember = 1000; // first membership number
        try {
            RandomAccessFile file = new RandomAccessFile(fileName, "rw");
            int nextMember = firstMember+(int)file.length()/recordSize;
                                                // next available membership number
            System.out.println("Welcome to the membership data base!");
            while (!Console.endOfFile()) {
                String token = Console.readToken();  // first token on line
                if (Character.isDigit(token.charAt(0))) { //  have a query
                    int num = Integer.parseInt(token);
                    if (num<firstMember||num>=nextMember)
                        System.out.println("Can't find that membership number");
                    else {
                        file.seek((num-firstMember)*recordSize);  // locate record
```

```
                    String name = file.readUTF().trim();  // trailing blanks deleted
                    char sport = file.readChar();
                    System.out.print(name);
                    if (sport=='t') System.out.println(" (tennis)");
                    else System.out.println(" (squash)");
                }
            }
            else { // have a new member
                String name = token + " " + Console.readToken();
                char sport = (Console.readToken()).charAt(0);
                int number = nextMember; nextMember++;
                file.seek(file.length()); // move to end of file
                file.writeUTF(String.format("%-"+nameLen+"s",name));  // pad
                file.writeChar(sport); file.writeInt(number);
                System.out.println("New membership number: " + number);
            }
        }
        file.close();
    }
    catch(IOException e) { System.out.println("File error!"); }
  }
}
```

7 Directories OPTIONAL

It is common in the design of files to include in each record a field (called a *key* field) with the special property that no two records have the same values in the key field. For any record, the value of its key field is called its *key*. Keys don't usually change over the lifetime of the file. For example, in a student data base, the keys are usually student numbers; in income tax files they may be social security numbers. One of the most common operation on files is to display the details of a record identified by its key supplied by the user at the keyboard. However, we may also be expected to identify records by values in other fields whose values are not guaranteed to be unique. For example, we may be asked to display the record of a student with a certain name. In that case we should display all the records that match the name supplied, as there is likely to be more than one.

Random access processing is particularly advantageous in quickly locating any particular record of interest to us. Sometimes we can calculate the location of a record from the information supplied, as in the example above where the location of the record for any club member can be deduced from the membership number. But that is not typical. It is extremely slow to search a file record by record because reading from or writing to a disk is orders of magnitude slower than accessing data held in variables in main memory. If the file is small then it can be read in its entirety into memory, in which case subsequent searching will be very fast. But many files are much too large to fit in memory.

For fast searching of large files we employ what is called a *directory* or *index* for each field on which we may need to search the file. For example, if we intend to search an income tax file for records with a particular social security number, then we maintain a directory for social security numbers. A directory (for a key field, say) consists of a collection of items, one item per record in the file. Each item has two components: the key value *v* for the record, and the byte offset in the file of the record. For example, suppose each record in a student database has four fields: the student's name (a string), his or her student number (a string), the course he or she is taking (a string) and the student's age (an integer). Suppose the first three records in the file are as follows, with offsets written underneath (note that we haven't assumed that all records are of equal size):

Ian Lee	2351	Law	17	Al Doe	1756	Law	19	Joe Rice	2311	Arts	21
0				24				47			

Then the directory for the student number field will contain the following information:

2351	0
1756	24
2311	47
.....	

– each item in the left hand column is a student number occurring in the file, and the associated item in the right-hand column is the offset of the record containing that number.

The directory may be stored in an array or (more likely) in a fancier data structure for which look-ups are very fast (such as a `HashMap` object to be described later). In typical industrial applications, the size of a directory will be less than 1% of the size of the file, perhaps much less, and hence more likely to be of a size that can be accommodated comfortably in memory. The directory can be created when the file is opened. This requires us to read the entire file, but once we have paid that price we can locate particular records very fast indeed. For example, if we are asked to display the details of the student whose student number is 2311 we just search the directory in memory for 2311, extract the offset 47, execute a seek on the file to location 47, and retrieve the record at that location (and of course that contains the record of a student whose number is 2311).

If the file can be changed in the same session as it is being queried (for example, if new records can be added), then of course the directory has to be updated to reflect the changes.

For large files, it may be too expensive to generate the directory each time the file is opened. In that case, the directory can be stored as an auxiliary file, and read into memory when the file is opened. These auxiliary files are called *index files*. If the file is very very large, then the directory may itself be too large to fit in memory. In that case, the directory has to be handled using sophisticated techniques that we will not go into here.

8 Sequential versus random file processing

If a program reads and/or writes individually selected records scattered throughout the file we say that it is processing the file *randomly* or by *random access*. For example, the club membership example earlier uses random file 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. This was the case in the integer lookup example earlier. 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. This was the case in the example where we created a file of persons.

If the run-time system knows when the file is opened that it will be only read sequentially, or only written to sequentially, it can mange the file operations significantly cheaper. Java provides classes especially for sequential processing called `DataInputStream` (for input) and `DataOutputStream` (for output). If efficiency is not a concern then we can equally well use `RandomAccessFile`.

The most computationally efficient technique for updating all (or most) of the records in a large binary file in some uniform way is, oddly enough, not to update the file but to create an entirely new one. Suppose, for example, that we have a personnel file and we want to increase the salaries of, say, all non-manual workers by 5%. This is done by reading each record of the personnel file in turn into a program variable, and inspecting it. If the record 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 sequentially, and the new file which is being written to sequentially. 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. Deleting records from a file is similar: we construct a new file with the chosen records deleted.

8 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. Class `File` resides in the `java.io` package which must be imported

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()
String getName()
File[] listFiles()
```

`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. **`f.getName()`** returns the name of teh file. **`f.listFiles()`** returns an array of `File` objects, one for each file or directory in the directory identified by `f`; `null` is returned if `f` does not identify a directory or if an i/o error occurs. 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
      }
    }
}
```

Example 2: generating a unique file name

The following code generates a file name that is guaranteed to be different from that of any other file in the directory.

```
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);
}
```