25 Random File Accessing

1 Reading and writing binary files

Many on-line applications require us to read and write to files 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 or a new member added. Retrieving the record associated with an individual or item demands careful design of the file. 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 fast. But many files are much too large to fit in memory.

A common approach in the design of files is 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. The core task in file searching is to display the details of a record identified by a key supplied by the user (at the keyboard, typically).

2 Using fixed size records

Locating a particular record in a file is made easier if we can ensure that all records are of the same size. If we can easily map each key to its record number, then we can easily calculate the offset of the record in the file: the *n*'th record begins at offset *n* times the record size in bytes, assuming the record count starts at 0. We achieve fixed size records 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 can 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. We design the file so that records are of fixed length; we can ensure this by padding names to length 40, say. 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!"); }
    }
}
```

3 Directories OPTIONAL

For fast searching of large files where fixed sized records are inconvenient, 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 class `HashMap` provided by the Java library). 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.

4 Changing and deleting records

If a file is merely read uniformly from start to finish, or written uniformly from start to finish, then the system can handle if very efficiently. In fact, the most computationally efficient technique for updating all (or most) of the records in a large binary file in some uniform way is 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 uniformly, and the new file which is being written to uniformly. 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.