23 Random Access Files

1 Random access files

Random access is a more flexible way of handling files than sequential access. It allows reading and writing to be intermixed, and records to be read and written in any order. For example, we may read the 9th record, then write a new version of the 7th record, and finally append an extra record. Unlike sequential access, random access allows us to overwrite existing records with new data: we just position the file pointer at an existing record and then write data. Remember that all binary files can be processed randomly, even if they were created sequentially (as they usually will be). If sequential processing is adequate for the job, however, it is usually to be preferred because most operating systems implement it more efficiently.

A random access file is handled by creating an instance of class **`RandomAccessFile`**:

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

`new RandomAccessFile(s,md)` makes the file named `s` on the disk available to the program in *mode* `md`. The mode is one of `"rw"` (which stands for "read-write") or `"r"` (which stands for "read-only"). When opened with mode `"rw"`, the file may be read or written to, whereas mode `"r"` limits file access to reading only. Two examples follow:

```
RandomAccessFile myFile = new RandomAccessFile("diskData", "rw");
RandomAccessFile myFile = new RandomAccessFile("diskData", "r");
```

If a file opened in read-write mode does not exist, an empty one is created. If it is opened in read-only mode it must exist in advance. If you intend only to read the file then you can still open it with mode `"rw"`, but demand on system resources will be greater. 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 `diskData` above we write `C:/myDirectory/diskData` (running under Windows) it refers to a file called `diskData` in folder `myDirectory` on drive `C`. Instances of `RandomAccessFile` contains, amongst other things, a file pointer (its initial value is not specified, but see `seek()` below). After either of the declarations 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. To use `RandomAccessFile` your program should import the `java.io` package.

The methods provided by `RandomAccessFile` for reading files are those we already know from `DataInputStream`:

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

The methods for writing are the following we know from `DataOutputStream`:

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

All the above methods behave as they do in `DataInputStream` and `DataOutputStream`. As with sequential processing, reading and writing takes place at the location indicated by the file pointer, and the file pointer is advanced by the length of the data read or written. Consequently, processing files sequentially that have been opened for random access imposes no extra burden on the programmer, although it is computationally less efficient. There is no method to read a string that has been written using `writeChars()`. As with `DataInputStream`, we have to read the string character-by-character into an array, and then convert the array contents to a `String` type. `RandomAccessFile` also provides

void close() throws IOException

`f.close()` closes file `f`. You should close random access files when you have finished with them.

The method that sets `RandomAccessFile` apart is the following:

void seek(*long*) throws IOException

`f.seek(n)` causes the file pointer of file `f` to be positioned `n` bytes from the start of the file. An `IOException` will thrown if n<0. If you want to read or write data other than at the offset indicated by the file pointer, you must first invoke `seek()`. For example, suppose `myFile` references a `RandomAccessFile` where the records are integers. Then the following reads the first and third integer from `myFile`:

```
myFile.seek(0);  int j = myFile.readInt();
myFile.seek(8);  int k = myFile.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. It is not forbidden to seek to an offset that exceeds the length of the file and then write to that location, but the intervening locations will contain garbage until such time as meaningful data is written to them.

`RandomAccessFile` also provides the following methods:

long length() throws IOException
long getFilePointer() 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). For example, with `myFile` referencing a `RandomAccessFile` whose records are integers, the following code overwrites the integer at the current location of the file pointer with the final integer in `myFile` (we assume the file is not empty):

```
long mark = myFile.getFilePointer();    // remember current offset in myFile
myFile.seek(myFile.length()-4);         // position file pointer at final integer
int k = myFile.readInt();               // read final integer in myFile into k
myFile.seek(mark);                      // re-position file pointer at mark
myFile.writeInt(k);                     // overwrite with k (= final integer)
```

Fixed size records

When records in a file are all of the same size we can easily calculate the location of the *n*'th record (it begins at offset *n* times the record size in bytes, assuming we start the record count at 0). It is therefore quite common to design the file so that the records are identically shaped: each record has a fixed number of fields, and each constituent field is of a fixed size. The size of each field is chosen so that it is large enough to accommodate all likely eventualities. In the case of entries in fields of type `String`, shorter entries are artificially lengthened by appending padding characters (such as 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 have decided on a length of 40 because that is sufficiently large to accommodate the longest name we think likely to arise. Added blanks can be written to the file using a loop, or by executing

the following smart piece of code:

```
out.writeChars("                    ".substring(s.length()));
```

Here, `out` refers to the sequential or random file, `s` references the string that has just been written and is to be padded with blanks (to length 40, say). There are 40 blanks in the string of blanks. You should convince yourself the statement writes precisely 40-*n* blanks where *n* denotes the value of `s.length()`, and that is just as we wish. 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, then the padding character must be something other than a blank. A common choice is the null character (i.e. `'\0'`) because this is exceedingly rare in strings that arise in practice.

It is not always practical to design a fixed uniform shape for the constituent records in a file, and so variable-sized records are much used in practice. Randomly accessing a file with variable-sized records requires the use of directories, as discussed later.

2 Large example: Club membership

We make a small database of members for a racquet club. The database contains one record per member, consisting of the member's name (forename plus surname), preferred sport, and membership number. Club members play one of two sports: tennis or squash. 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 details are displayed. The following is a typical interaction:

```
Welcome to the membership data base!

Jill Sixpack tennis
New member: Jill Sixpack (tennis) Membership number: 1000

Willy Smasher squash
New member: Willy Smasher (squash) Membership number: 1001

1000
Jill Sixpack (tennis) Membership number: 1000

Nick Rusher tennis
New member: Nick Rusher (tennis) Membership number: 1002

1001
Willy Smasher (squash) Membership number: 1001

1003
Can't find that membership number
```

The database resides in a file and so it carries over from run to run of the program.

The notion of a club member is central to the problem and is encapsulated in a class called `Member`:

```
import java.io.*;
class Member {
    private String name;      // member's name (max nameSize characters)
    private char sport;   // 't': tennis, 's': squash
    private int number;   // membership number

    private final static int nameSize = 40;  // max number of characters in name
    final static int recordSize = nameSize*2 + 2 + 4; // bytes per Member record

    Member() {
    }

    Member(String n, char s, int num) {
    // assume n.length()<=nameSize, and s either 't' or 's'
        name = n; sport = s; number = num;
    }

    void append(RandomAccessFile f) { // append attributes to f
        try {
            f.seek(f.length());  // move file pointer to end of file
            f.writeChars(name);
            for (int i=0; i<nameSize-name.length(); i++)
                f.writeChar(' ');  // pad name with spaces
            f.writeChar(sport);
            f.writeInt(number);
        }
        catch (IOException e)  {
            e.printStackTrace();
        }
    }

    void get(RandomAccessFile f, int n) {
    // read attributes from n'th record of f, 0 <= n < number of records
        try {
            f.seek(n*recordSize);  // move file pointer to offset of record n
            char[] b = new char[nameSize];
            for (int i=0; i<nameSize; i++)
                b[i] = f.readChar();
            name = String.valueOf(b).trim();  // name with trailing blanks deleted
            sport = f.readChar();
            number = f.readInt();
        }
```

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

        void put() { // display member on screen
            System.out.print(name);
            if (sport=='t') System.out.print(" (tennis)");
            else System.out.print(" (squash)");
            System.out.println(" Membership number: " + number);
        }
    }
```

A member has three attributes: name, preferred sport, and membership number. To facilitate random processing, the collected attributes will occupy a fixed size record on disk. We decide that each member's name will be at most 40 characters long, and that we will pad it with blanks to length 40 when we write it to the file. It will become clear shortly why variable `recordSize` is needed and why it is not private.

Class `Membership` encapsulates the notion of a collection of members:

```
import java.io.*;
class Membership {

    private RandomAccessFile file;  // file of Members
    private int numMembers;  // number of members (= number of records)

    private static final int firstMember = 1000;  // first membership number

    Membership(String fileName) {
        try {
            file = new RandomAccessFile(fileName, "rw");
            numMembers = (int)file.length()/Member.recordSize;
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    Member getByNumber(int num) {
    // retrieve member with membership number num (return null if num invalid)
        if (num<firstMember || num>=firstMember+numMembers) return null;
        Member p = new Member();
        p.get(file, num-firstMember);
        return p;
    }

    Member addMember(String name, char sport) {
```

```
    // add member where name & sport are given by parameters, and
    // membership number is generated.  New member is returned.
        Member member = new Member(name, sport, firstMember+numMembers);
        member.append(file);
        numMembers++;
        return member;
    }

    void close() { // close file
        try { file.close(); }
        catch (IOException e) { e.printStackTrace(); }
    }
}
```

`Membership` has two attributes: a file which contains the members' details, and the current number of members. The number of members is of course equal to the number of records in the file, and that is calculated by dividing the length of the file by the record size. The record size is available in variable `recordSize` in class `Members`, notwithstanding the fact that it is used in class `Membership`. This is as it should be because the record size is directly related to the instance variables of class `Member`.

Class `Club` contains just method `main()` whose only role is to interact with the user at the keyboard:

```
class Club {
    public static void main(String[] args)  {
    // Handle queries and additions entered by user at keyboard
        Membership members = new Membership("MembersFile");
        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))) { // handle a query
                int n = Integer.parseInt(token);
                Member p = members.getByNumber(n);
                if (p != null) p.put();
                else System.out.println("Can't find that membership number");
            }
            else { // handle an addition
                String name = token + " " + Console.readToken();  // name
                char sport = (Console.readToken()).charAt(0);  // tennis or squash
                Member p = members.addMember(name, sport);
                System.out.print("New member: ");  p.put();
            }
        }
        members.close();
```

```
        }
    }
```

3 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, and in income tax files they may be social security numbers. Perhaps the most common operation on files is to display the details of a record identified by its key. 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, and there is likely to be more than one.

Random access files are advantageous only if we can quickly locate any particular record of interest to us. Sometimes we can calculate the location of a record from the information supplied, as in the example of the preceding section 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 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 Doe	2351	Maths	18	Al Dean	1756	Arts	19	Jo Ryan	2311	Arts	21
0				26				51			

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

```
2351        0
1756       26
2311       51
.....
```

– 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 from Java's Collections Framework). 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 51, execute a seek on the file to location 51, 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 Large example: fast file lookup ADVANCED

In on-line query systems, a user typically enters a word identifying some person or item stored in a database, and the system displays the details of the person or item. Because disk accessing is very slow, we usually use directories (or index files) to allow fast searching. We illustrate how this is done in a program which looks up the details of students, where students are identified by student numbers. The following is a typical interaction at the keyboard:

```
Welcome to the student records enquiry system.

2351
Name: Ian Doe
Student number: 2351
Course: Maths
Age: 18

6546
Sorry -- no info on that student.
```

So as not to encumber the code with detail, the information stored about each student is small – just name, student number, course, and age. In these circumstances the file will be small and could well fit in memory. You must imagine, however, that in practice the data stored about

each student runs into many thousands of bytes (even tens of thousands if graphic data such as photographs are stored), and the use of directories is vital.

Naturally the notion at the core of the design is that of a student, and we start by encapsulating it in a class called `Student`:

```
import java.io.*;
class Student {

    private String name;          // name
    private String idNumber;      // student number
    private String course;        // course taken
    private int age;              // age in years

    Student(String who, String id, String what, int years) {
        name = who; idNumber = id; course = what; age = years;
    }

    Student() {
    }

    String theIdNumber() {
        return idNumber;
    }

    void put(RandomAccessFile f) {
    // Write attributes to f (as indicated by file pointer)
        try {
            f.writeUTF(name);
            f.writeUTF(idNumber);
            f.writeUTF(course);
            f.writeInt(age);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    void get(RandomAccessFile f) {
    // Read attributes from f (as indicated by file pointer)
        try {
            name = f.readUTF();
            idNumber = f.readUTF();
            course = f.readUTF();
            age = f.readInt();
        }
        catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }

    void get(RandomAccessFile f, long n) {
    // Read attributes from f at offset n
        try {
            f.seek(n); // move file pointer to offset n
            get(f);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    void put() { // display attributes on screen
        System.out.println("Name: " + name);
        System.out.println("Student number: " + idNumber);
        System.out.println("Course: " + course);
        System.out.println("Age: " + age);
    }
}
```

Note that we store the student number in an instance variable `idNumber` of type `String`, not `int`. Although we conventionally use the word "number" in "student number", a student number is just a string which happens to consist of digits. If `idNumber` was declared to be of type `int`, it would be limited in size (variables of type `int` are 4 bytes long), and any leading zeros would not appear when it was printed. The methods in `Student` are simple. In writing them, the only design decision of consequence is that we have elected to carry out all file input/output using class `RandomAccessFile`. `RandomAccessFile` is preferred because it gives us access to the file pointer, and that is useful when building the directory.

Now let us think about encapsulating a directory as a class called `Directory`. The class implements a collection of student number-offset pairs, and needs to provide only two methods: one to insert a number-offset pair, and another to retrieve the offset associated with a given number.

```
class Directory {
    ......
    void add(String s, long n) {
    // Associate offset n with student number s
    ......
    }

    long lookUp(String s) {
    // Return offset associated with student number s  (-1 if s absent)
```

```
        .....
        }
}
```

How we fill in the details is a matter entirely local to `Directory`, without any bearing on the coding of other classes. An unsophisticated implementation is to store the pairs in arrays. A more efficient alternative in this instance is Java's `HashMap` class. We will present an array implementation now, and leave the more sophisticated implementation as an exercise. For simplicity, we will use linear search when looking up an entry; it is left as an exercise to replace it with binary search.

```
class Directory {

    private String[] id = new String[10000];     // student numbers (max 10000)
    private long[] offset = new long[id.length];  // id[i] is in record at offset[i]
    private int numIds = 0;  // first numIds entries in id[] & offset[] significant

    void add(String s, long n) {
    // Associate offset n with student number s
        id[numIds] = s;
        offset[numIds] = n;
        numIds++;
    }

    long lookUp(String s) {
    // Return offset associated with student number s  (-1 if s absent)
        int i = 0;
        while (i<numIds && !id[i].equals(s)) {
            i++;
        }
        if (i<numIds) return offset[i];
        else return -1;
    }
}
```

An *indexed file* is a file which has an auxiliary directory (or several of them) to aid locating particular records. Class `IndexedStudentFile` encapsulates a student file indexed on student numbers:

```
class IndexedStudentFile {

    private RandomAccessFile theFile;      // the file of students
    private Directory numberDirectory;    // directory for student numbers

    IndexedStudentFile(String fileName) {
    // Initialse theFile using the file named fileName, and create
```

```
        // and initialise numberDirectory
            .....
        }

        Student getByNumber(String id) {
        // Return a Student object whose attributes are taken from the
        // record with id in its student number field (return null if absent)
            .....
        }

        void closeFile() { // close theFile
            .....
        }
    }
```

All access to the file of student records will be done via method `getByNumber()` in `IndexedStudentFile`. Note that the directory is constructed when the file is opened, i.e. as part of the operation to construct an object type `IndexedStudentFile`. The complete coding of `IndexedStudentFile` follows:

```
import java.io.*;
class IndexedStudentFile {

    private RandomAccessFile theFile;      // the file of students
    private Directory numberDirectory;    // directory for student numbers

    IndexedStudentFile(String fileName) { // fileName is name of file of students
        openFile(fileName);            // initialse theFile
        makeNumberDirectory();     // initialise numberDirectory
    }

    private void openFile(String fileName) {
    // Open file fileName (if it doesn't exist create a small test file).
        // First check for file's existence, & create one if necessary
        File studentFile = new File(fileName);
        if (!studentFile .exists()) {
            try {
                RandomAccessFile out = new RandomAccessFile(fileName, "rw");
                new Student("Ian Doe", "2351", "Maths", 18).put(out);
                new Student("Al Dean", "1756", "Arts", 19).put(out);
                new Student("Jo Ryan", "2311", "Arts", 21).put(out);
                new Student("Pat Baker", "2073", "Science", 19).put(out);
                new Student("Pete Rice", "2249", "Maths", 20).put(out);
                out.close();
            }
            catch (IOException e) {
                e.printStackTrace();
```

```
            }
        }
        // Open the file for reading
        try {
            theFile = new RandomAccessFile(fileName, "r");
        }
        catch (IOException e) { e.printStackTrace(); }
    }

    private void makeNumberDirectory() {
    // Make directory for student numbers (theFile has been initialised)
        try {
            numberDirectory = new Directory();
            Student p = new Student();
            long fileLength = theFile.length();
            theFile.seek(0);
            long filePointer = 0;  // current location of file pointer
            // retrieve each record and record its student number & offset
            while (filePointer<fileLength) {
                p.get(theFile);  // next student record
                numberDirectory.add(p.theIdNumber(), filePointer);
                filePointer = theFile.getFilePointer();
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    Student getByNumber(String id) {
    // Return a Student object whose attributes are taken from the
    // record with id in its student number field (return null if absent)
        long n = numberDirectory.lookUp(id);
        if (n>=0) {
            Student p = new Student();
            p.get(theFile, n);
            return p;
        }
        else return null;
    }

    void closeFile() { // close the file
        try {
            theFile.close();
        }
        catch (IOException e) { e.printStackTrace(); }
```

```
    }
}
```

Note that constructing the directory imposes little demand on memory – there is never more than one record of the file in memory at any given time.

Finally class `StudentEnquires` contains just method `main()` which interacts with the user at the keyboard:

```
class StudentEnquiries {
    public static void main(String[] args) {
        IndexedStudentFile indexedFile = new IndexedStudentFile("StudentFile");
        System.out.println("Welcome to the student records enquiry system");
        System.out.println();
        while (!Console.endOfFile()) {
            String id = Console.readString();  // student number
            Student p = indexedFile.getByNumber(id);
            if (p!=null) p.put();
            else System.out.println("Sorry -- no info on that student.");
            System.out.println();
        }
        indexedFile.closeFile();
    }
}
```