24 Reading Binary Files

1 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()`.

`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 earlier:

```
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.