JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris August 2001

19 RANDOM ACCESS FILES

19.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- picture a random access file as an indexed array
- explain the concept of a file pointer
- implement and test methods that use RandomAccessFile methods

19.2 PRE-REQUISITES

The student should be comfortable with using *HashMaps* (Part A § 19).

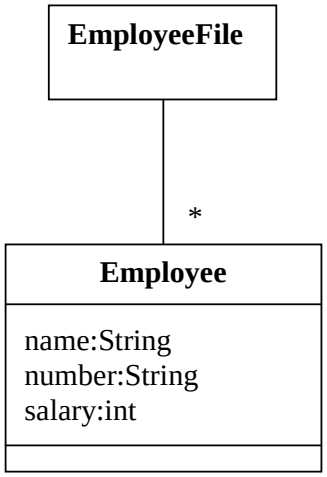
19.3 INTRODUCTION

In the last lesson we looked at text files. Files that have strict sequential access, such as text files for example, have limitations. To determine that a record is not in the file you have to search through it from beginning to end. To amend a record in the middle of the file you have to create, record by record, a copy of the original file, except that the copy contains the amended record and not the old record. With direct access files, supported by an index, you can determine almost instantly that a particular record is not in the file; and you can directly amend a record in the middle of a file without looking at all the preceding records.

We see how to create and maintain a file whose records can be accessed directly.

19.4 CLASS DIAGRAM

We illustrate the principles with a file of *Employee* objects.



An employee has a name, a number and a salary.

19.5 RANDOM ACCESS FILES

The records in a random access file are all the same length. The records in the file are indexed from zero upwards. Data is stored in the file as a sequence of bytes.

bond	007	50000	Q	001	75000	penny	002	85000
0			1			2		

Figure 19.1 *Records in a random access file are indexed from zero upwards and are all the same length*

A *RandomAccessFile* object maintains a file pointer. A read operation starts from the position in the file referred to by the pointer. After a read operation the file pointer is advanced to a point just past the last byte read.

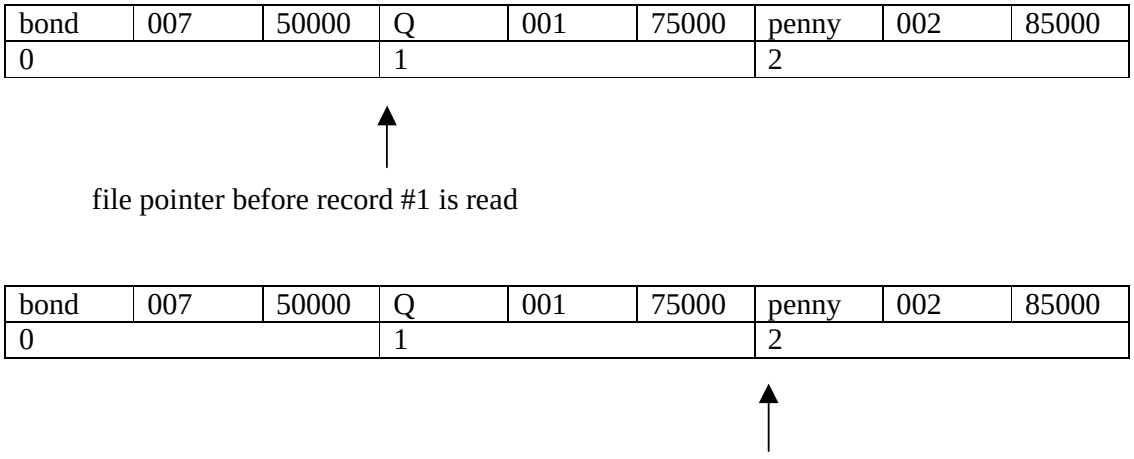


Figure 19.2 *After a read operation the file pointer is advanced to just beyond the last byte read*

A write operation starts from the position in the file referred to by the file pointer. After a write operation the file pointer is advanced to a point just beyond the last byte written.

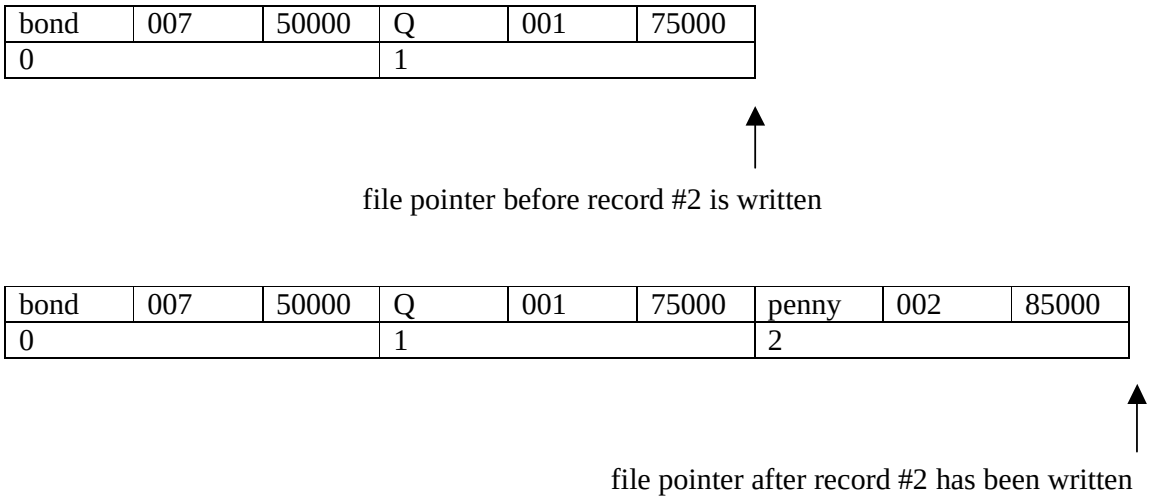


Figure 19.3 *After a write operation the file pointer is advanced to just beyond the last byte written*

To update Q's salary to 90000 for example we first position the pointer to where Q's record starts. Then from that point in the file we write an amended version of Q's record, thereby overwriting the original.

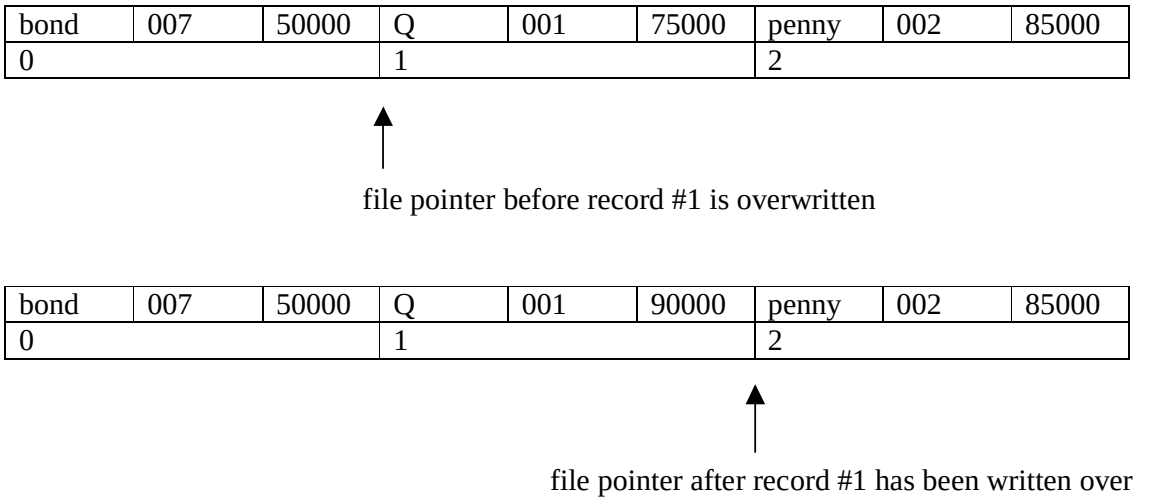


Figure 19.4 *After a write operation the file pointer is advanced to just beyond the last byte written*

19.6 STORAGE REQUIREMENTS

The *Employee* class differs from that described in §18 because we need to

- write and read fixed length strings and not the variable length strings we are accustomed to
- write bytes to streams and retrieve bytes from streams

The storage requirements for the commonly used primitive data types are shown below.

type	bytes
boolean	1
char	2
int	4
double	8

A *boolean* value requires just one byte of storage, a *char* two, an *int* four and a *double* eight.

We shall arbitrarily decide that an employee name may have at most fifteen characters and a number may have up to five characters. Now we can calculate the size of an employee record in bytes.

```
public class Employee {
  private static final int nameCapacity = 15;
  private static final int numberCapacity = 5;
  public static final int RECORD_SIZE = 2 * nameCapacity +
                                        2 * numberCapacity +
                                        4;  // for the int
  private String name;
  private String number;
  private int salary;
```

19.7 WRITING

We use a *DataOutput* object to represent an output byte stream. To write a *char* or an *int* to an output byte stream is easy.

```
DataOutput out;
char ch;
int i;
...
out.writeChar(ch);
out.writeInt(i);
```

We write a helper method to write a string of a fixed size to the output stream. *writeFixedString()* writes size characters to the file, either padding the string with characters whose Unicode value is zero to make up the required size or truncating the given string. (Unicode is a superset of ASCII, see Part A Fundamentals Appendix B.)

```
private void writeFixedString(String s, int size,
                              DataOutput out) throws IOException
{
  for (int i = 0; i < size; i++) {
    char ch = 0;
    if (i < s.length()) {
      ch = s.charAt(i);
    }
    out.writeChar(ch);
  }
}
```

We might make a call to *writeFixedString(String, int, DataOutput)* with the message

```
writeFixedString("789", 5, outputStream);
```

Exercise: dry run the *writeFixedString()* method shown above by completing the table shown below.

s	size	i	i<size	ch	i<s.length()	written
789	5	0	true	0	true	
				7		7

19.8 READING

We use a *DataInput* object to represent a byte-input stream. To retrieve an *int* or a *char* from a *DataInput* byte stream is easy.

```
DataInput inStream;
...
int i = inStream.readInt();
char ch = inStream.readChar();
```

We create a helper method to read a fixed number of bytes into a string. The *readFixedString(int, DataInput)* method reads characters from the input stream until it has consumed *size* characters. Only non-zero characters are added to the growing string, *s*.

```
private String readFixedString(int size, DataInput in)
                                    throws IOException
{
  String s = "";
  for (int i = 0; i < size; i++) {
    char ch = in.readChar();
    if (ch != 0)
      s += ch;
  }
  return s;
}
```

The entire *Employee* class is shown below.

```
/* Employee.java
   Terry Marris  9 August 2001
*/

import java.io.*;

public class Employee {
  private static final int nameCapacity = 15;
  private static final int numberCapacity = 5;
  public static final int RECORD_SIZE = 2 * nameCapacity +
                                        2 * numberCapacity +
                                        4;
  private String name;
  private String number;
  private int salary;

  public Employee(String name, String number, int salary)
  {
    this.name = name;
    this.number = number;
    this.salary = salary;
  }


  public String getNumber()
  {
    return number;
  }


  private void writeFixedString(String s, int size,
                            DataOutput out) throws IOException
  {
    for (int i = 0; i < size; i++) {
      char ch = 0;
      if (i < s.length()) {
        ch = s.charAt(i);
      }
      out.writeChar(ch);
    }
  }


  public void write(DataOutput outStream) throws IOException
  {
    writeFixedString(name, nameCapacity, outStream);
    writeFixedString(number, numberCapacity, outStream);
    outStream.writeInt(salary);
  }
```

```
  private String readFixedString(int size, DataInput in)
                                        throws IOException
  {
    String s = "";
    for (int i = 0; i < size; i++) {
      char ch = in.readChar();
      if (ch != 0)
        s += ch;
    }
    return s;
  }


  public void read(DataInput inStream) throws IOException
  {
    name = readFixedString(nameCapacity, inStream);
    number = readFixedString(numberCapacity, inStream);
    salary = inStream.readInt();
  }


  public String toString()
  {
    return name + ", " + number + ", £" + salary;
  }
}
```

19.9 EMPLOYEE FILE AS A RANDOM ACCESS FILE

An *EmployeeFile* object has a random access file of *Employee* objects and an index that maps employee numbers to record positions in the file.

index

employee number	record #
007	0
001	1
002	2

So, given an employee number we can immediately determine where its record is located in the file. The index is implemented as a *HashMap*. It is held in memory and created by the constructor from an existing file. Every time a new record is added to the file, the index is updated.

```
public class EmployeeFile {
  private String fileName = "Employee.data";
  private HashMap index = new HashMap();
```

The constructor creates an empty random access file if it does not already exist.

```
  public EmployeeFile()
  {
    ...
    File file = new File(fileName);
    if (!file.exists()) {
      RandomAccessFile outStream = new
                          RandomAccessFile(fileName, "rw");
      outStream.close();
    }
```

A *RandomAccessFile* object supports both reading and writing. The *rw* argument in its constructor specifies that the file is open for both reading and writing.

But if the file already exists, the *EmployeeFile()* constructor reads through the file record by record, creating the index as it does so.

First, we create an employee with no name, no number and no salary.

```
Employee employee = new Employee("", "", 0);
```

Then we create the *RandomAccessFile* instance.

```
RandomAccessFile inStream = new
                        RandomAccessFile(fileName, "r");
```

The *r* argument value specifies the file is open for reading only.

If we are going to repeatedly retrieve the next record from the file we need to know how many records there are to be retrieved. It would be an error to attempt to retrieve a record when you are past the end of the file.

A *RandomAccessFile* object has a method, *length()*, that returns the size of the file in bytes. If we know how many bytes make a record, we can calculate how many records there are in the file. We just take the file length in bytes and divide it by the record length in bytes to get the number of records in the file.

```
int fileSize=(int)(inStream.length() / Employee.RECORD_SIZE);
```

fileSize represents the number of records in the file.

Now we can iterate over the file.

```
for (int i = 0; i < fileSize; i++) {
  inStream.seek(i * Employee.RECORD_SIZE);
   ...
}
```

int i represents the record number. *inStream.seek(i * Employee.RECORD_SIZE)* positions the file pointer *i * Employee.RECORD_SIZE* bytes into the file. So if *i* is two, the file pointer is then positioned at the beginning of record #2 in the file.

Having positioned the file pointer we perform the read operation

```
employee.read(inStream);
```

and update the index with the employee number/record number pair.

```
index.put(employee.getNumber(), new Integer(i));
```

As usual, we wrap the file access operation in a *try...catch...* combination.

```
public EmployeeFile()
/* Creates an empty file if it does not already exist or
   creates the index from the existing file.
*/
{
  try {
    File file = new File(fileName);
    if (!file.exists()) {
      RandomAccessFile outStream = new
                       RandomAccessFile(fileName, "rw");
      outStream.close();
    }
    else {
      Employee employee = new Employee("", "", 0);
      RandomAccessFile inStream = new
                       RandomAccessFile(fileName, "r");
      int fileSize =
          (int)(inStream.length() / Employee.RECORD_SIZE);
      for (int i = 0; i < fileSize; i++) {
        inStream.seek(i * Employee.RECORD_SIZE);
        employee.read(inStream);
        index.put(employee.getNumber(), new Integer(i));
      }
      inStream.close();
    }
  }
  catch(IOException e) {
    System.out.println("EmployeeFile(): " + e);
    System.exit(1);
  }
}
```

The method to add a new employee to the file is shown next. You should notice that

- the index traps and rejects duplicate employee numbers
- the file pointer is positioned at the end of the file before the write operation takes place

```
public String add(String name, String number, int salary)
/* Creates a new Employee instance from the given fields,
   adds it onto the end of the file and updates the index.
   Returns failure if an employee already exists with the
   given number, otherwise returns success.
*/
{
  if (index.containsKey(number))
    return "failure - duplicate employee number";

  Employee employee = new Employee(name, number, salary);
  try {
    RandomAccessFile outStream = new
                     RandomAccessFile(fileName, "rw");
    int fileSize = (int)(outStream.length() /
                                    Employee.RECORD_SIZE);
    outStream.seek(fileSize * Employee.RECORD_SIZE);
    employee.write(outStream);
    index.put(number, new Integer(fileSize));
    outStream.close();
  }
  catch(IOException e) {
    System.out.println("add():" + e);
    System.exit(1);
  }
  return "success";
}
```

The entire implementation for the *EmployeeFile* class, along with some testing, is shown below. Notice that

- *toString()* was tested on an empty file
- the *replace()* method was tested on the first and last records in the file.
- the state of the file was printed before and after the changes were made to its contents

An empty file, and the first and last records in the file represent boundary conditions and should always feature in tests.

```
/* EmployeeFile.java
   Terry Marris 9 August 2001
*/

import java.util.*;
import java.io.*;

public class EmployeeFile {
  private String fileName = "Employee.data";
  private HashMap index = new HashMap();

  public EmployeeFile()
  /* Creates an empty file if it does not already exist or
     creates the index from the existing file.
  */
  {
    try {
      File file = new File(fileName);
      if (!file.exists()) {
        RandomAccessFile outStream = new
                        RandomAccessFile(fileName, "rw");
        outStream.close();
      }
      else {
        Employee employee = new Employee("", "", 0);
        RandomAccessFile inStream = new
                        RandomAccessFile(fileName, "r");
        int fileSize =
            (int)(inStream.length() / Employee.RECORD_SIZE);
        for (int i = 0; i < fileSize; i++) {
          inStream.seek(i * Employee.RECORD_SIZE);
          employee.read(inStream);
          index.put(employee.getNumber(), new Integer(i));
        }
        inStream.close();
      }
    }
    catch(IOException e) {
      System.out.println("EmployeeFile(): " + e);
      System.exit(1);
    }
  }
```

```
public String add(String name, String number, int salary)
/* Creates a new Employee instance from the given fields,
   adds it onto the end of the file and updates the index.
   Returns failure if an employee already exists with the
   given number, otherwise returns success.
*/
{
  if (index.containsKey(number))
    return "failure - duplicate employee number";

  Employee employee = new Employee(name, number, salary);
  try {
    RandomAccessFile outStream = new
                      RandomAccessFile(fileName, "rw");
    int fileSize = (int)(outStream.length() /
                                    Employee.RECORD_SIZE);
    outStream.seek(fileSize * Employee.RECORD_SIZE);
    employee.write(outStream);
    index.put(number, new Integer(fileSize));
    outStream.close();
  }
  catch(IOException e) {
    System.out.println("add():" + e);
    System.exit(1);
  }
  return "success";
}
```

```
  public String replace(
                String name, String number, int salary)
  /* Creates a new employee with the given parameters and
replaces the employee with the same number.
     Returns failure if no employee with the given number is
found, success otherwise.
  */
  {
    if (!index.containsKey(number))
      return "failure - no employee with the given number";

    Employee employee = new Employee(name, number, salary);
    Integer integerPosition = (Integer)index.get(number);
    int position = integerPosition.intValue();

    try {
      RandomAccessFile outStream = new
                      RandomAccessFile(fileName, "rw");
      outStream.seek(position * Employee.RECORD_SIZE);
      employee.write(outStream);
      outStream.close();
    }
    catch(IOException e) {
      System.out.println("replace():" + e);
      System.exit(1);
    }
    return "success";
  }
```

```
public String toString()
/* Returns the contents of the file as a string with
   one record per line.
*/
{
  Employee employee = new Employee("", "", 0);
  String string = "";
  try {
    RandomAccessFile inStream = new
                     RandomAccessFile(fileName, "r");
    int fileSize = (int)(inStream.length() /
                                  Employee.RECORD_SIZE);
    for (int i = 0; i < fileSize; i++) {
      inStream.seek(i * Employee.RECORD_SIZE);
      employee.read(inStream);
      string += employee + "\n";
    }
    inStream.close();
  }
  catch(IOException e) {
    System.out.println("toString(): " + e);
    System.exit(1);
  }
  return string;
}
```

```
  public static void main(String[] s)
  {
    EmployeeFile employeeFile = new EmployeeFile();
    System.out.println(employeeFile);

    employeeFile.add("bond", "007", 50000);
    employeeFile.add("Q", "001", 75000);
    employeeFile.add("moneypenny", "002", 65000);

    System.out.println(
               "Before amending first and last records ...");
    System.out.println(employeeFile);

    employeeFile.replace("bond", "007", 70000);
    employeeFile.replace("moneypenny", "002", 85000);

    System.out.println(
               "After amending first and last records ...");
    System.out.println(employeeFile);
  }
}
```

Output

```
Before amending first and last records ...
bond, 007, £50000
Q, 001, £75000
moneypenny, 002, £65000

After amending first and last records ...
bond, 007, £70000
Q, 001, £75000
moneypenny, 002, £85000
```

19.10 REVIEW

19.11 FURTHER READING

HORTSTMANN & CORNELL Core Java 2 Volume 1 pp 658

In the next lesson we look at object streams.

19.12 EXERCISES

1 Explain how the method *replace(String, String, int)* accomplishes its task.

2 Explain what each line of the *toString()* actually does.

3 Implement and test the method *Employee employee(String number)* that returns the employee with the given number from a file of *Employee* objects, or *null* if no such employee is found in the file.

4 Implement and test the method *int contains(String number)* that returns an integer >= 0 if the file contains an employee with the given number, -1 otherwise.