JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris August 2001

20 OBJECT FILES

20.1 LEARNING OBJECTIVES

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

- write and test code to maintain a file of objects

20.2 PRE-REQUISITES

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

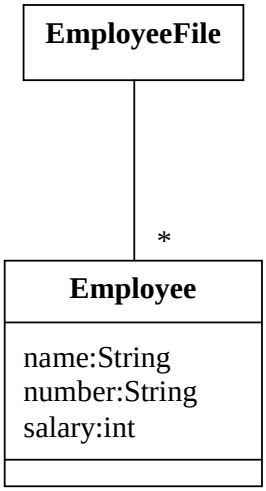
20.3 INTRODUCTION

In the last lesson we used a fixed-length record format to store records all of the same type. But where you have objects of several types, e.g. you might have the types *Student* and *Staff*, both inherited from *LibraryUser*, you need to save objects.

In this lesson we see how to store objects in files and how to retrieve objects from files.

20.4 CLASS DIAGRAM

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



20.5 THE EMPLOYEE CLASS

The ONLY change we make to the regular *Employee* class is to specify that it implements *Serializable*. *Serializable* has no methods that we need to implement. Simple.

When a *Serializable* object is written to an object stream, the file system automatically includes information that allow it to re-create the object when it is subsequently retrieved from an object stream. Brilliant.

```
/* Employee.java
   Terry Marris  10 August 2001
*/

import java.io.*;

public class Employee implements Serializable {
  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 toString()
  {
    return name + ", " + number + ", £" + salary;
  }
}
```

20.6 THE EMPLOYEE FILE CLASS

Perhaps the best way of managing a file of *Employee* objects is to store them in collection such as a *HashMap* and then store the *HashMap* object in the file. If we want to add new *Employee* objects to the file, or make changes to existing *Employee* objects, we first retrieve the *HashMap* from the file, make the required changes to its contents, then store the updated *HashMap* back into the file. We preserve the original *HashMap* and its contents in a back up file.

```
public class EmployeeFile {
  private String fileName = "Employee.data";
  private String backUp = "Employee.back";
  private Map map = new HashMap();
```

The constructor retrieves the *HashMap* object from the file with a call to *readObject()*.

```
      File file = new File(fileName);
      if (file.exists()) {
        FileInputStream inFile = new
                        FileInputStream(fileName);
        ObjectInputStream inStream = new
                          ObjectInputStream(inFile);
        map = (HashMap)inStream.readObject();
        inStream.close();
      }
```

The *update()* method writes the map back to the file. If a back up file already exists, it is deleted.

```
File backupFile = new File(backUp);
if (backupFile.exists())
  backupFile.delete();
```

If the employee file exists, it is renamed as the back up file.

```
File employeeFile = new File(fileName);
if (employeeFile.exists())
  employeeFile.renameTo(backupFile);
```

Then the map is written to the file with a call to *writeObject(map)*;

```
FileOutputStream outFile = new
                  FileOutputStream(fileName);
ObjectOutputStream outStream = new
                    ObjectOutputStream(outFile);
outStream.writeObject(map);
```

The entire file processing operations are confined to just the constructor and the update method. No other method, neither the *add()* nor the *toString()* accesses the file. Easy.

The entire class, together with some sample output, is shown below.

```
/* EmployeeFile.java
   Terry Marris 10 August 2001
*/

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

public class EmployeeFile {
  private String fileName = "Employee.data";
  private String backUp = "Employee.back";
  private Map map = new HashMap();

  public EmployeeFile()
  /* Retrieves the HashMap from the file,
     if the file exists.
  */
  {
    try {
      File file = new File(fileName);
      if (file.exists()) {
        FileInputStream inFile = new
                        FileInputStream(fileName);
        ObjectInputStream inStream = new
                          ObjectInputStream(inFile);
        map = (HashMap)inStream.readObject();
        inStream.close();
      }
    }
    catch(Exception 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,
     and adds it to the map.
  */
  {
    if (map.containsKey(number))
      return "failure - duplicate employee number";

    Employee employee = new Employee(name, number, salary);
    map.put(number, employee);
    return "success";
  }
```

```
public String toString()
/* Returns the contents of the map as a string with
   one record per line.
*/
{
  String string = "";
  Set entries = map.entrySet();
  Iterator it = entries.iterator();
  while (it.hasNext()) {
    Map.Entry entry = (Map.Entry)it.next();
    Object value = entry.getValue();
    Employee employee = (Employee)value;
    string += employee;
    if (it.hasNext())
      string += "\n";
  }
  return string;
}


public String update()
{
  try {
    File backupFile = new File(backUp);
    if (backupFile.exists())
      backupFile.delete();

    File employeeFile = new File(fileName);
    if (employeeFile.exists())
      employeeFile.renameTo(backupFile);

    FileOutputStream outFile = new
                     FileOutputStream(fileName);
    ObjectOutputStream outStream = new
                       ObjectOutputStream(outFile);
    outStream.writeObject(map);
    outStream.close();
  }
  catch(Exception e) {
    System.out.println("update(): " + e);
    System.exit(1);
  }
  return "success";
}
```

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

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

    employeeFile.update();  // must be the last message if the
                            // changes are to be preserved.
  }
}
```

Output when the program is run for the first time

```
                                  ← contents of an empty file
success
success                           } adding three employees to the file
success
bond, 007, £50000
moneypenny, 002, £65000           } contents of the file after adding three objects
Q, 001, £75000
```

Output when the program is run for the second time without first deleting the data file.

```
bond, 007, £50000
moneypenny, 002, £65000           } the original contents of the file
Q, 001, £75000
failure - duplicate employee number
failure - duplicate employee number  } adding objects already in the file
failure - duplicate employee number
bond, 007, £50000
moneypenny, 002, £65000           } file contents unchanged
Q, 001, £75000
```

If the order in which the employees appeared was important you would use a linear data structure such as an array (see § 2) rather than a set or a map.

20.7 REVIEW

20.8 FURTHER READING

HORSTMANN & CORNELL *Core Java 2 Volume 1* pp 664

20.9 EXERCISES

1 Write and test the method *String replace(String name, String number, int salary)* that replaces the employee with the given number with a new employee object with the given parameters. The method should return either success or failure - employee with the given number not found. Your test program should demonstrate that the file contents have changed.