Chapter 4
File Handling

Learning Objectives

After reading this chapter, you should:

- know how to create and process serial files in Java;
- know how to create and process random access files in Java;
- know how to redirect console input and output to disc files;
- know how to construct GUI-based file-handling programs;
- know how to use command line parameters with Java programs;
- understand the concept and importance of Java's serialisation mechanism and know how to implement it;
- know how to make use of *ArrayLists* for convenient packaging of serialised objects.

With all our programs so far, there has been a very fundamental limitation: all data accepted is held only for as long as the program remains active. As soon as the program finishes execution, any data that has been entered and the results of processing such data are thrown away. Of course, for very many real-life applications (banking, stock control, financial accounting, etc.), this limitation is simply not realistic. These applications demand **persistent** data storage. That is to say, data must be maintained in a permanent state, such that it is available for subsequent further processing. The most common way of providing such persistent storage is to use disc files. Java provides such a facility, with the access to such files being either **serial** or **random**. The following sections explain the use of these two file access methods, firstly for non-GUI applications and later for GUI applications. In addition, the important and often neglected topic of **serialisation** is covered.

4.1 Serial Access Files

Serial access files are files in which data is stored in physically adjacent locations, often in no particular logical order, with each new item of data being added to the end of the file.

J. Graba, *An Introduction to Network Programming with Java: Java 7 Compatible*,
DOI 10.1007/978-1-4471-5254-5_4, © Springer-Verlag London 2013

[Note that serial files are often mis-named *sequential* files, even by some authors who should know better. A sequential file is a serial file in which the data are stored in some particular order (e.g., in account number order). A sequential file is a serial file, but a serial file is not necessarily a sequential file.]

Serial files have a number of distinct disadvantages (as will be pointed out in 4.2), as a consequence of which they are often used only to hold relatively small amounts of data or for temporary storage, prior to processing, but such files are simpler to handle and are in quite common usage.

The internal structure of a serial file can be either **binary** (i.e., a compact format determined by the architecture of the particular computers on which the file is to be used) or **text** (human-readable format, almost invariably using ASCII). The former stores data more efficiently, but the latter is much more convenient for human beings. Coverage here will be devoted exclusively to **text** files.

Before Java SE 5, a text file required a *FileReader* object for input and a *FileWriter* object for output. As of Java 5, we can often use just a *File* object for either input or output (though not for both at the same time). The *File* constructor takes a *String* argument that specifies the name of the file as it appears in a directory listing.

Examples

```
(i)  File inputFile = new File("accounts.txt");
(ii) String fileName = "dataFile.txt";
     ..................................
     File outputFile = new File(fileName);
```

N.B. If a string literal is used (e.g., `"results.txt"`), the full pathname may be included, but double backslashes are then required in place of single backslashes (since a single backslash would indicate an escape sequence representing one of the invisible characters, such as tab). For example:

```
File resultsFile = new File("c:\\data\\results.txt");
```

Incidentally, we can (of course) call our files anything we like, but we should follow good programming practice and give them meaningful names. In particular, it is common practice to denote text data files by a suffix of `.txt` (for 'text').

Class *File* is contained within package *java.io*, so this package should be imported into any file-handling program. Before Java SE 5, it was necessary to wrap a *BufferedReader* object around a *FileReader* object in order to read from a file. Likewise, it was necessary to wrap a *PrintWriter* object around a *FileWriter* object in order to write to the file. Now we can wrap a *Scanner* object around a *File* object for input and a *PrintWriter* object around a *File* object for output. (The *PrintWriter* class is also within package *java.io*.)

Examples

```
(i) Scanner input =
                    new Scanner(new File("inFile.txt"));
```

```
(ii)   PrintWriter output =
                new PrintWriter(new File("outFile.txt"));
```

We can then make use of methods *next, nextLine*, *nextInt*, *nextFloat*, … for input and methods *print* and *println* for output.

Examples (using objects *input* and *output*, as declared above)

```
(i)    String item = input.next();
(ii)   output.println("Test output");
(iii)  int number = input.nextInt();
```

Note that we need to know the type of the data that is in the file before we attempt to read it! Another point worth noting is that we may choose to create anonymous *File* objects, as in the examples above, or we may choose to create named *File* objects.

Examples

```
(i)  File inFile = new File("inFile.txt");
     Scanner input = new Scanner(inFile);

(ii) File outFile = new File("outFile.txt");
     PrintWriter output = new PrintWriter(outFile);
```

Creating a named *File* object is slightly longer than using an anonymous *File* object, but it allows us to make use of the *File* class's methods to perform certain checks on the file. For example, we can test whether an input file actually exists. Programs that depend upon the existence of such a file in order to carry out their processing **must** use named *File* objects. (More about the *File* class's methods shortly.)

When the processing of a file has been completed, the file should be closed via the *close* method, which is a member of both the *Scanner* class and the *PrintWriter* class. For example:

```
input.close();
```

This is particularly important for output files, in order to ensure that the file buffer has been emptied and all data written to the file. Since file output is buffered, it is not until the output buffer is full that data will normally be written to disc. If a program crash occurs, then any data still in the buffer will not have been written to disc. Consequently, it is good practice to close a file explicitly if you have finished writing to it (or if your program does not need to write to the file for anything more than a very short amount of time). Closing the file causes the output buffer to be flushed and any data in the buffer to be written to disc. No such precaution is relevant for a file used for input purposes only, of course.

Note that we cannot move from reading mode to writing mode or vice versa without first closing our *Scanner* object or *PrintWriter* object and then opening a *PrintWriter* object or *Scanner* object respectively and associating it with the file.

Now for a simple example program to illustrate file output…

Example

Writes a single line of output to a text file.

```
import java.io.*;

public class FileTest1
{
   public static void main(String[] args)
                                    throws IOException
   {
      PrintWriter output =
               new PrintWriter(new File("test1.txt"));
      output.println("Single line of text!");
      output.close();
   }
}
```

Note that there is no 'append' method for a serial file in Java. After execution of the above program, the file 'test1.txt' will contain **only** the specified line of text. If the file already existed, its initial contents will have been overwritten. This may or may not have been your intention, so take care! If you need to add data to the contents of an existing file, you still (as before Java SE 5) need to use a *FileWriter* object, employing either of the following constructors with a second argument of `true`:

- `FileWriter(String <fileName>, boolean <append>)`
- `FileWriter(File <fileName>, boolean <append>)`

For example:

```
FileWriter addFile = new FileWriter("data.txt", true);
```

In order to send output to the file, a *PrintWriter* would then be wrapped around the *FileWriter*:

```
PrintWriter output = new PrintWriter(addFile);
```

These two steps may, of course, be combined into one:

```
PrintWriter output =
     new PrintWriter(
             new FileWriter("data.txt", true);
```

It would be a relatively simple matter to write Java code to read the data back from a text file to which it has been written, but a quick and easy way of checking that the data has been written successfully is to use the relevant operating system command. For example, on a PC, open up an MS-DOS command window and use the MS-DOS *type* command, as below.

```
type test1.dat
```

Often, we will wish to accept data from the user during the running of a program. In addition, we may also wish to allow the user to enter a name for the file. The next example illustrates both of these features. Since there may be a significant delay between consecutive file output operations while awaiting input from the user, it is good programming practice to use *File* method *flush* to empty the file output buffer. (Remember that, if the program crashes and there is still data in the file output buffer, that data will be lost!)

Example

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

public class FileTest2
{
    public static void main(String[] args)
                        throws IOException
    {
        String fileName;
        int mark;
        Scanner input= new Scanner(System.in);

        System.out.print("Enter file name: ");
        fileName = input.nextLine();
        PrintWriter output =
               new PrintWriter(new File(fileName));
        System.out.println("Ten marks needed.\n");
        for (int i=1; i<11; i++)
        {
            System.out.print("Enter mark " + i + ": ");
            mark = input.nextInt();
            //* Should really validate entry! *
            output.println(mark);
            output.flush();
        }
        output.close();
    }
}
```

Example output from this program is shown in Fig. 4.1.

When reading data from any text file, we should not depend upon being able to read a specific number of values, so we should read until the end of the file is reached. Programming languages differ fundamentally in how they detect an end-of-file situation. With some, a program crash will result if an attempt is made to read beyond the end of a file; with others, you **must** attempt to read beyond the end of the file in order for end-of-file to be detected. Before Java SE 5, Java fell into the latter category and it was necessary to keep reading until the string read (and **only** strings were read then)

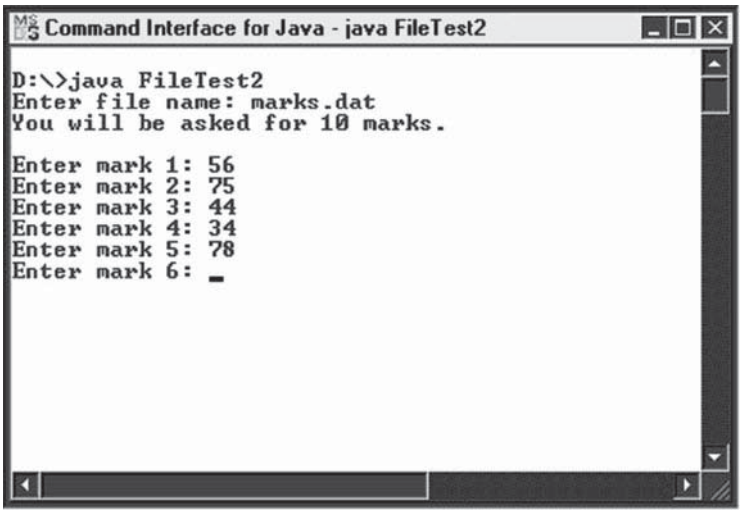

Fig. 4.1 Accepting serial file input from the user

had a `null` reference. As of Java 5, we must **not** attempt to read beyond the end-of-file if we wish to avoid the generation of a *NoSuchElementException*. Instead, we have to check ahead to see whether there is more data to be read. This is done by making use of the *Scanner* class's *hasNext* method, which returns a Boolean result indicating whether or not there is any more data.

Example

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

public class FileTest3
 {
    public static void main(String[] args)
                              throws IOException
    {
        int mark, total=0, count=0;
        Scanner input =
                  new Scanner(new File("marks.txt"));

        while (input.hasNext())
        {
            mark = input.nextInt();
            total += mark;
            count++;
        }
        input.close();
```

```
        System.out.println("Mean = "
                                + (float)total/count);
    }
}
```

Note that there is no structure imposed upon a file by Java. It is the programmer's responsibility to impose any required logical structuring upon the file. 'Records' on the file are not physical units determined by Java or the operating system, but logical units set up and maintained by the programmer. For example, if a file is to hold details of customer accounts, each logical record may comprise the following:

- account number;
- customer name;
- account balance.

It is the programmer's responsibility to ensure that each logical record on the file holds exactly these three fields and that they occur in the order specified.

4.2 File Methods

Class *File* has a large number of methods, the most important of which are shown below.

- ***boolean canRead()***
  Returns *true* if file is readable and *false* otherwise.
- ***boolean canWrite()***
  Returns *true* if file is writeable and *false* otherwise.
- ***boolean delete()***
  Deletes file and returns *true/false* for success/failure.
- ***boolean exists()***
  Returns *true* if file exists and *false* otherwise.
- ***String getName()***
  Returns name of file.
- ***boolean isDirectory()***
  Returns *true* if object is a directory/folder and *false* otherwise.
  (Note that *File* objects can refer to ordinary files or to directories.)
- ***boolean isFile()***
  Returns *true* if object is a file and *false* otherwise.
- ***long length()***
  Returns length of file in bytes.
- ***String[] list()***
  If object is a directory, array holding names of files within directory is returned.

- ***File[] listFiles()***
  Similar to previous method, but returns array of *File* objects.
- ***boolean mkdir()***
  Creates directory with name of current *File* object.
  Return value indicates success/failure.

The following example illustrates the use of some of these methods.

Example

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

public class FileMethods
{
    public static void main(String[] args)
                        throws IOException
    {
        String filename;
        Scanner input = new Scanner(System.in);

        System.out.print(
                    "Enter name of file/directory ");
        System.out.print("or press <Enter> to quit: ");
        filename = input.nextLine();

        while (!filename.equals("")) //Not <Enter> key.
        {
            File fileDir = new File(filename);

            if (!fileDir.exists())
            {
                System.out.println(filename
                            + " does not exist!");
                break; //Get out of loop.
            }

            System.out.print(filename + " is a ");
            if (fileDir.isFile())
                System.out.println("file.");
            else
                System.out.println("directory.");

            System.out.print("It is ");
            if (!fileDir.canRead())
                System.out.print("not ");
            System.out.println("readable.");

            System.out.print("It is ");
            if (!fileDir.canWrite())
                System.out.print("not ");
```

```
            System.out.println("writeable.");

            if (fileDir.isDirectory())
            {
                System.out.println("Contents:");
                String[] fileList   = fileDir.list();
                //Now display list of files in
                //directory…
                for (int i=0;i<fileList.length;i++)
                    System.out.println("     "
                                       + fileList[i]);
            }
            else
            {
                System.out.print("Size of file: ");
                System.out.println(fileDir.length()
                                           +  " bytes.");
            }

            System.out.print(
               "\n\nEnter name of next file/directory ");
            System.out.print(
                         "or press <Enter> to quit: ");
            filename   = input.nextLine();
        }
        input.close();
    }
}
```

Figure 4.2 shows example output from the above program.

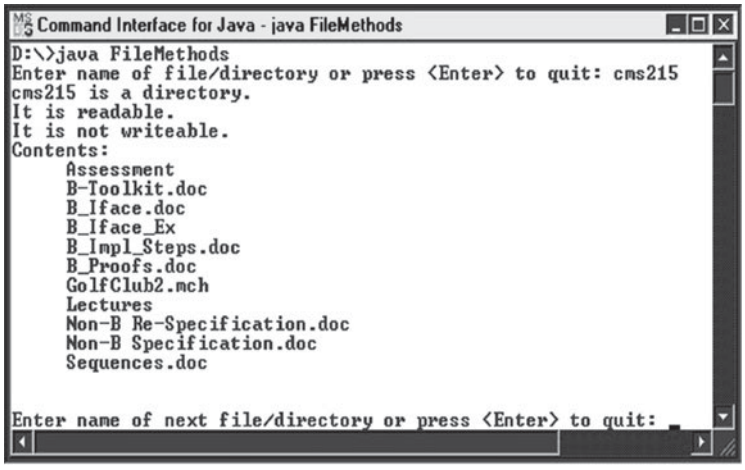

Fig. 4.2 Outputting file properties

4.3 Redirection

By default, the standard input stream *System.in* is associated with the keyboard, while the standard output stream *System.out* is associated with the VDU. If, however, we wish input to come from some other source (such as a text file) or we wish output to go to somewhere other than the VDU screen, then we can **redirect** the input/output. This can be **extremely** useful when debugging a program that requires anything more than a couple of items of data from the user. Instead of re-entering the data each time we run the program, we simply create a text file holding our items of data on separate lines (using a text editor or wordprocessor) and then re-direct input to come from our text file. This can save a **great deal** of time-consuming, tedious and error-prone re-entry of data when debugging a program.

We use '<' to specify the new source of input and '>' to specify the new output destination.

Examples

```
java ReadData < payroll.txt
java WriteData > results.txt
```

When the first of these lines is executed, program 'ReadData(.class)' begins execution as normal. However, whenever it encounters a file input statement (via *Scanner* method *next*, *nextLine*, *nextInt*, etc.), it will now take as its input the next available item of data in file 'payroll.txt'. Similarly, program 'WriteData(.class)' will direct the output of any *print* and *println* statements to file 'results.txt'.

We can use redirection of both input and output with the same program, as the example below shows. For example:

```
java ProcessData < readings.txt > results. txt
```

For program 'ProcessData(.class)' above, all file input statements will read from file 'readings.txt', while all *print*s and *println*s will send output to file 'results.txt'.

4.4 Command Line Parameters

When entering the *java* command into a command window, it is possible to supply values in addition to the name of the program to be executed. These values are called **command line parameters** and are values that the program may make use of. Such values are received by method *main* as an array of *String*s. If this argument is called *arg* [Singular used here, since individual elements of the array will now be referenced], then the elements may be referred to as *arg[0]*, *arg[1]*, *arg[2]*, etc.

Example

Suppose a compiled Java program called *Copy.class* copies the contents of one file into another. Rather than prompting the user to enter the names of the files (which would be perfectly feasible, of course), the program may allow the user to specify the names of the two files as command line parameters:

```
java Copy source.dat dest.dat
```

(Please ignore the fact that MS-DOS has a perfectly good *copy* command that could do the job without the need for our Java program!)

Method *main* would then access the file names through *arg[0]* and *arg[1]*:

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

public class Copy
{
    public static void main(String[] arg)
                                    throws IOException
    {
        //First check that 2 file names have been
        //supplied…
        if (arg.length < 2)
        {
            System.out.println(
                        "You must supply TWO file names.");
            System.out.println("Syntax:");
            System.out.println(
              "        java Copy <source> <destination>");
            return;
        }

        Scanner source = new Scanner(new File(arg[0]));
        PrintWriter destination =
                       new PrintWriter(new File(arg[1]));

        String input;
        while (source.hasNext())
        {
            input = source.nextLine();
            destination.println(input);
        }

        source.close();
        destination.close();
    }
}
```

4.5 Random Access Files

Serial access files are simple to handle and are quite widely used in small-scale applications or as a means of providing temporary storage in larger-scale applications. However, they do have two distinct disadvantages, as noted below.

(i) We can't go directly to a specific record. In order to access a particular record, it is necessary to physically read past all the preceding records. For applications containing thousands of records, this is simply not feasible.
(ii) It is not possible to add or modify records within an existing file. (The whole file would have to be re-created!)

Random access files (probably more meaningfully called **direct access** files) overcome both of these problems, but do have some disadvantages of their own…

(i) In common usage, all the (logical) records in a particular file must be of the same length.
(ii) Again in common usage, a given string field must be of the same length for all records on the file.
(iii) Numeric data is not in human-readable form.

However, the speed and flexibility of random access files often greatly outweigh the above disadvantages. Indeed, for many real-life applications, there is no realistic alternative to some form of direct access.

To create a random access file in Java, we create a *RandomAccessFile* object. The constructor takes two arguments:

- a string or *File* object identifying the file;
- a string specifying the file's access mode.

The latter of these may be either "r" (for read-only access) or "rw" (for read-and-write access). For example:

```
RandomAccessFile ranFile =
       new RandomAccessFile("accounts.dat","rw");
```

Before reading or writing a record, it is necessary to position the **file pointer**. We do this by calling method *seek*, which requires a single argument specifying the byte position within the file. Note that the first byte in a file is byte **0**. For example:

```
ranFile.seek(500);
//Move to byte 500 (the 501st byte).
```

In order to move to the correct position for a particular record, we need to know two things:

- the size of records on the file;
- the algorithm for calculating the appropriate position.

The second of these two factors will usually involve some kind of **hashing function** that is applied to the key field. We shall avoid this complexity and assume that

records have keys 1, 2, 3,… and that they are stored sequentially. However, we still need to calculate the record size. Obviously, we can decide upon the size of each *String* field ourselves. For numeric fields, though, the byte allocations are fixed by Java (in a platform-independent fashion) and are as shown below.

`int`	4 bytes
`long`	8 bytes
`float`	4 bytes
`double`	8 bytes

Class *RandomAccessFile* provides the following methods for manipulating the above types:

readInt, readLong, readFloat, readDouble
writeInt, writeLong, writeFloat, writeDouble

It also provides similarly-named methods for manipulating the other primitive types. In addition, it provides a method called *writeChars* for writing a (variable-length) string. Unfortunately, no methods for reading/writing a string of fixed size are provided, so we need to write our own code for this. In doing so, we shall need to make use of methods *readChar* and *writeChar* for reading/writing the primitive type *char*.

Example

Suppose we wish to set up an accounts file with the following fields:

- account number (*long*);
- surname (*String*);
- initials (*String*);
- balance (*float*).

N.B. When calculating the number of bytes for a *String* field, do not make the mistake of allocating only one byte per character. Remember that Java is based on the unicode character set, in which each character occupies **two** bytes.

Now let's suppose that we decide to allocate 15 (unicode) characters to surnames and 3 (unicode) characters to initials. This means that each surname will be allocated 30 (i.e., 15×2) bytes and each set of initials will be allocated 6 (i.e., 3×2) bytes. Since we know that a *long* occupies precisely 8 bytes and a *float* occupies precisely 4 bytes, we now know that record size = (8 + 30 + 6 + 4) bytes = 48 bytes. Consequently, we shall store records starting at byte positions 0, 48, 96, etc. The formula for calculating the position of any record on the file is then:

(Record No. –1) × 48

For example, suppose our *RandomAccessFile* object for the above accounts file is called *ranAccts*. Then the code to locate the record with account number 5 is:

```
ranAccts.seek(192);   //(5–1)x48 = 192
```

Since method *length* returns the number of bytes in a file, we can always work out the number of records in a random access file by dividing the size of the file by

the size of an individual record. Consequently, the number of records in file *ranAccts* at any given time = *ranAccts.length()/48*.

Now for the code…

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

public class RanFile1
{
   private static final int REC_SIZE = 48;
   private static final int SURNAME_SIZE = 15;
   private static final int NUM_INITS = 3;
   private static long acctNum = 0;
   private static String surname, initials;
   private static float balance;

   public static void main(String[] args)
                                    throws IOException
   {
      RandomAccessFile ranAccts =
          new RandomAccessFile("accounts.dat", "rw");

      Scanner input = new Scanner(System.in);

      String reply = "y";

      do
      {
          acctNum++;
          System.out.println(
             "\nAccount number " + acctNum + ".\n");
          System.out.print("Surname: ");
          surname = input.nextLine();
          System.out.print("Initial(s): ");
          initials = input.nextLine();
          System.out.print("Balance: ");
          balance = input.nextFloat();

          //Now get rid of carriage return(!)…
          input.nextLine();

          writeRecord(ranAccts); //Method defined below.

          System.out.print(
             "\nDo you wish to do this again (y/n)? ");
          reply = input.nextLine();
      }while (reply.equals("y")||reply.equals("Y"));

      System.out.println();
      showRecords(ranAccts);     //Method defined below.
```

```
}
public static void writeRecord(RandomAccessFile file)
                                    throws IOException
{
   //First find starting byte for current record…
   long filePos = (acctNum-1) * REC_SIZE;

   //Position file pointer…
   file.seek(filePos);

   //Now write the four (fixed-size) fields.
   //Note that a definition must be provided
   //for method writeString…
   file.writeLong(acctNum);
   writeString(file, surname, SURNAME_SIZE);
   writeString(file, initials, NUM_INITS);
   file.writeFloat(balance);
}
public static void writeString(RandomAccessFile file,
       String text, int fixedSize) throws IOException
{
   int size = text.length();

   if (size<=fixedSize)
   {
      file.writeChars(text);

      //Now 'pad out' the field with spaces…
      for (int i=size; i<fixedSize; i++)
          file.writeChar(' ');
   }
   else    //String is too long!
      file.writeChars(text.substring(0,fixedSize));
   //Write to file the first fixedSize characters of
   //string text, starting at byte zero.
}
public static void showRecords(RandomAccessFile file)
                                    throws IOException
{
   long numRecords = file.length()/REC_SIZE;

   file.seek(0);     //Go to start of file.
   for (int i=0; i<numRecords; i++)
   {
      acctNum = file.readLong();
      surname = readString(file, SURNAME_SIZE);
```

```
            //readString defined below.
            initials = readString(file, NUM_INITS);
            balance = file.readFloat();

            System.out.printf("" + acctNum
                              + "   " + surname
                              + "   " + initials + "   "
                              + "%.2f %n",balance);
         }
      }

      public static String readString(
                    RandomAccessFile file, int fixedSize)
                                       throws IOException
      {
         String value = "";    //Set up empty string.
         for (int i=0; i<fixedSize; i++)
         //Read character and concatenate it onto value…
             value+=file.readChar();

         return value;
      }
}
```

Note that methods *readString* and *writeString* above may be used *without modification* in any Java program that needs to transfer strings from/to a random access file.

The following screenshot demonstrates the operation of this program (Fig. 4.3).

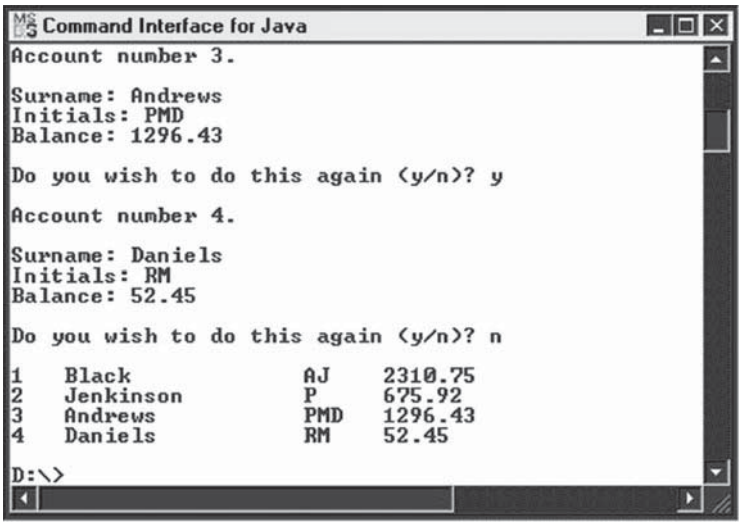

Fig. 4.3 Creating a simple random access file and displaying its contents

The above example does not adequately demonstrate the direct access capabilities of a *RandomAccessFile* object, since we have processed the whole of the file from start to finish, dealing with records in the order in which they are stored on the file. We should also be able to retrieve individual records from anywhere in the file and/or make modifications to those records. The next example shows how this can be done for our accounts file.

Example

```
//Allows the user to retrieve individual account
//records and modify their balances.

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

public class RanFile2
{
    private static final int REC_SIZE=48;
    private static final int SURNAME_SIZE=15;
    private static final int NUM_INITS=3;
    private static long acctNum=0;
    private static String surname, initials;
    private static float balance;

    public static void main(String[] args)
                                    throws IOException
    {
        Scanner input = new Scanner(System.in);
        RandomAccessFile ranAccts =
            new RandomAccessFile("accounts.dat", "rw");
        long numRecords = ranAccts.length()/REC_SIZE;
        String reply;
        long currentPos;      //File pointer position.

        do
        {
            System.out.print("\nEnter account number: ");
            acctNum = input.nextLong();
            while ((acctNum<1) || (acctNum>numRecords))
            {
                System.out.println(
                       "\n*** Invalid number! ***\n");
                System.out.print(
                            "\nEnter account number: ");
                acctNum = input.nextLong();
            }
```

```
            showRecord(ranAccts);     //Defined below.
            System.out.print("\nEnter new balance: ");
            balance = input.nextFloat();

            input.nextLine();
            //Get rid of carriage return!

            currentPos = ranAccts.getFilePointer();
            ranAccts.seek(currentPos-4); //Back 4 bytes.
            ranAccts.writeFloat(balance);
            System.out.print(
                     "\nModify another balance (y/n)? ");
            reply = (input.nextLine()).toLowerCase();
        }while (reply.equals("y"));
        //(Alternative to method in previous example.)

        ranAccts.close();
    }

    public static void showRecord(RandomAccessFile file)
                                      throws IOException
    {
        file.seek((acctNum-1)*REC_SIZE);
        acctNum = file.readLong();
        surname = readString(file, SURNAME_SIZE);
        initials = readString(file, NUM_INITS);
        balance = file.readFloat();

        System.out.println("Surname:  " + surname);
        System.out.println("Initials: " + initials);
        System.out.printf("Balance:  %.2f %n",balance);
    }

    public static String readString(
                RandomAccessFile file, int fixedSize)
                                      throws IOException
    {
        //Set up empty buffer before reading from file…
        StringBuffer buffer = new StringBuffer();

        for (int i=0; i<fixedSize; i++)
        //Read character from file and append to buffer.
            buffer.append(file.readChar());
       return buffer.toString(); //Convert into String.
    }
}
```

4.6 Serialisation [U.S. Spelling Serialization]

As seen in the preceding sections, transferring data of the primitive types to and from disc files is reasonably straightforward. Transferring string data presents a little more of a challenge, but even this is not a particularly onerous task. However, how do we go about transferring objects of classes? (*String* is a class, of course, but it is treated rather differently from other classes.) One way of saving an object to a file would be to decompose the object into its constituent fields (strings and numbers) and write those individual data members to the file. Then, when reading the values back from the file, we could re-create the original objects by supplying those values to the appropriate constructors. However, this is a rather tedious and long-winded method. In addition, since the data members of an object may themselves include other objects (some of whose data members may include further objects, some of whose members…), this method would not be generally applicable.

Unlike other common O-O languages, Java provides an inbuilt solution: **serialisation**. Objects of any class that implements the Serializable interface may be transferred to and from disc files as whole objects, with no need for decomposition of those objects. The *Serializable* interface is, in fact, nothing more than a marker to tell Java that objects of this class may be transferred on an object stream to and from files. Implementation of the *Serializable* interface need involve no implementation of methods. The programmer merely has to ensure that the class to be used includes the declaration '*implements Serializable*' in its header line.

Class *ObjectOutputStream* is used to save entire objects directly to disc, while class *ObjectInputStream* is used to read them back from disc. For output, we wrap an object of class *ObjectOutputStream* around an object of class *FileOutputStream*, which itself is wrapped around a *File* object or file name. Similarly, input requires us to wrap an *ObjectInputStream* object around a *FileInputStream* object, which in turn is wrapped around a *File* object or file name.

Examples

```
(i)  ObjectOutputStream outStream =
           new ObjectOutputStream(
               new FileOutputStream("personnel.dat"));
(ii) ObjectInputStream inStream =
           new ObjectInputStream(
               new FileInputStream("personnel.dat"));
```

Methods *writeObject* and *readObject* are then used for the actual output and input respectively. Since these methods write/read objects of class *Object* (the ultimate superclass), any objects read back from file must be **typecast** into their original class before we try to use them. For example:

```
Personnel person = (Personnel)inStream.readObject();
//(Assuming that inStream is as declared above.)
```

In addition to the possibility of an *IOException* being generated during I/O, there is also the possibility of a *ClassNotFoundException* being generated, so we must either handle this exception ourselves or throw it. A further consideration that needs to be made is how we detect end-of-file, since there is no equivalent of the *Scanner* class's *hasNext* method for use with object streams. We **could** simply use a `for` loop to read back the number of objects we believe that the file holds, but this would be very bad practice in general (especially as we may often not know how many objects a particular file holds).

The only viable option there appears to be is to catch the *EOFException* that is generated when we read past the end of the file. This author feels rather uneasy about having to use this technique, since it conflicts with the fundamental ethos of exception handling. Exception handling (as the term implies) is designed to cater for exceptional and erroneous situations that we do not expect to happen if all goes well and processing proceeds as planned. Here, however, we are going to be using exception handling to detect something that we not only know will happen eventually, but also are **dependent** upon happening if processing is to reach a successful conclusion. Unfortunately, there does not appear to be any alternative to this technique.

Example

This example creates three objects of a class called *Personnel* and writes them to disc file (as objects). It then reads the three objects back from file (employing a typecast to convert them into their original type) and makes use of the 'get' methods of class *Personnel* to display the data members of the three objects. We must, of course, ensure that class *Personnel* implements the *Serializable* interface (which involves nothing more than including the phrase *implements Serializable*). In a real-life application, class *Personnel* would be defined in a separate file, but it has been included in the main application file below simply for convenience.

```
import java.io.*;

public class Serialise
{
    public static void main(String[] args)
        throws IOException, ClassNotFoundException
    {
        ObjectOutputStream outStream =
           new ObjectOutputStream(
                new FileOutputStream("personnel.dat"));

        Personnel[] staff =
            {new Personnel(123456,"Smith", "John"),
             new Personnel(234567,"Jones", "Sally Ann"),
             new Personnel(999999,"Black", "James Paul")};

        for (int i=0; i<staff.length; i++)
            outStream.writeObject(staff[i]);
```

```
        outStream.close();

        ObjectInputStream inStream =
                new ObjectInputStream(
                 new FileInputStream("personnel.dat"));

        int staffCount = 0;

        try
        {
            do
            {
                Personnel person =
                      (Personnel)inStream.readObject();
                staffCount++;

                System.out.println(
                       "\nStaff member " + staffCount);
                System.out.println("Payroll number: "
                                   + person.getPayNum());
                System.out.println("Surname: "
                                   + person.getSurname());
                System.out.println("First names: "
                                 + person.getFirstNames());
            }while (true);
        }
        catch (EOFException eofEx)
        {
            System.out.println(
                "\n\n*** End of file ***\n");
            inStream.close();
        }
    }
}

class Personnel implements Serializable
//No action required by Serializable interface.
{
    private long payrollNum;
    private String surname;
    private  String firstNames;

    public Personnel(long payNum,String sName,
                                          String fNames)
    {
        payrollNum = payNum;
        surname = sName;
```

```
        firstNames = fNames;
    }
    public long getPayNum()
    {
        return payrollNum;
    }
    public String getSurname()
    {
        return surname;
    }
    public String getFirstNames()
    {
        return firstNames;
    }
    public void setSurname(String sName)
    {
        surname = sName;
    }
}
```

Output from the above program is shown in Fig. 4.4.

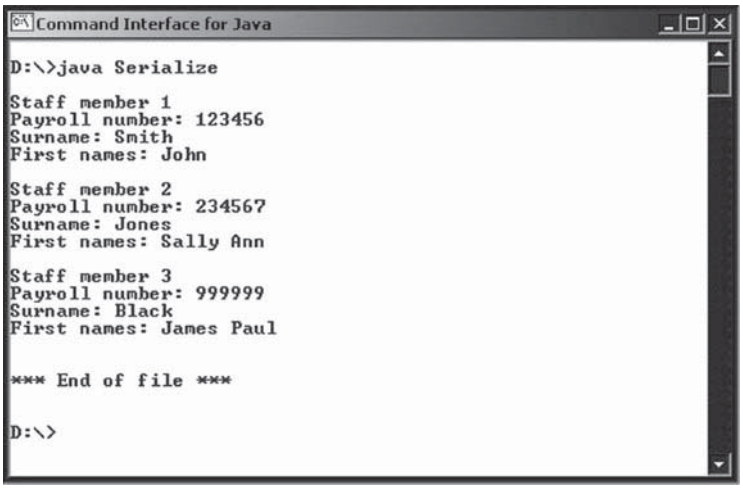

Fig. 4.4 Displaying the contents of a small file of serialised objects

4.7 File I/O with GUIs

Since the majority of applications nowadays have GUI interfaces, it would be nice to provide such an interface for our file handling programs. The reader will almost certainly have used file handling applications that provide such an interface. A particularly common feature of such applications is the provision of a dialogue box that allows the user to navigate the computer's file system and select either an existing file for opening or the destination directory for a file that is to be created. By employing Swing class *JFileChooser*, we can display a dialogue box that will allow the user to do just that.

A *JFileChooser* object opens a modal dialogue box ['Modal' means that the window must be dismissed before further processing may be carried out] that displays the system's directory structure and allows the user to traverse it. Once a *JFileChooser* object has been created, method *setFileSelectionMode* may be used to specify whether files and/or directories are selectable by the user, via the following constants:

JFileChooser.FILES_ONLY
JFileChooser.DIRECTORIES_ONLY
JFileChooser.FILES_AND_DIRECTORIES

Example

```
JFileChooser fileChooser = new JFileChooser();
fileChooser.setFileSelectionMode(
                          JFileChooser.FILES_ONLY);
```

We can then call either *showOpenDialog* or *showSaveDialog*. The former displays a dialogue box with 'Open' and 'Cancel' buttons, while the latter displays a dialogue box with 'Save' and 'Cancel' buttons. Each of these methods takes a single argument and returns an integer result. The argument specifies the *JFileChooser*'s parent component, i.e. the window over which the dialogue box will be displayed. For example:

```
fileChooser.showOpenDialog(this);
```

The above will cause the dialogue box to be displayed in the centre of the application window. If *null* is passed as an argument, then the dialogue box appears in the centre of the screen.

The integer value returned may be compared with either of the following inbuilt constants:

JFileChooser.CANCEL_OPTION
JFileChooser.APPROVE_OPTION

Testing against the latter of these constants will return 'true' if the user has selected a file.

Example

```
int selection = fileChooser.showOpenDialog(null);
if (selection == JFileChooser.APPROVE_OPTION)
   ..........................................
   //Specifies action taken if file chosen.)
```

If a file has been selected, then method *getSelectedFile* returns the corresponding *File* object. For example:

```
File file = fileChooser.getSelectedFile();
```

For serial I/O of strings and the primitive types, we would then wrap either a *Scanner* object (for input) or a *PrintWriter* object (for output) around the *File* object, as we did in 4.1.

Example

```
Scanner fileIn = new Scanner(file);
PrintWriter fileOut = new PrintWriter(file);
```

We can then make use of methods *next, nextInt*, etc. for input and methods *print* and *println* for output.

Similarly, for serial I/O of objects, we would wrap either an *ObjectInputStream* object plus *FileInputStream* object or an *ObjectOutputStream* object plus *FileOutputStream* object around the *File* object.

Example

```
ObjectInputStream fileIn =
          new ObjectInputStream(
                 new FileInputStream(file));

ObjectOutputStream fileOut =
          new ObjectOutputStream(
                 new FileOutputStream(file));
```

We can then make use of methods *readObject* and *writeObject*, as before. Of course, since we are now dealing with a GUI, we need to implement *ActionListener*, in order to process our button selections.

Example

This example is a simple application for reading a file of examination marks and displaying the results of individual students, one at a time. The file holding results will be a simple serial file, with each student's data held as three fields in the following sequence: surname, first name(s) and examination mark. We shall firstly allow the user to navigate the computer's file system and select the desired file (employing a *JFileChooser* object). Once a file has been selected, our program will open the file, read the first (logical) record and display its contents within the text fields of a panel we have set up. This panel will also contain two buttons, one to allow the user to open another file and the other to allow the user to move on to the next record in the file.

In order to read an individual record, we shall define a method called *readRecord* that reads in the surname, first name(s) and examination mark for an individual student.

Before looking at the code, it is probably useful to look ahead and see what the intended output should look like. In order that the *JFileChooser* object may be viewed as well, the screenshot in Fig. 4.5 shows the screen layout after one file has been opened and then the 'Open File' button has been clicked on by the user.

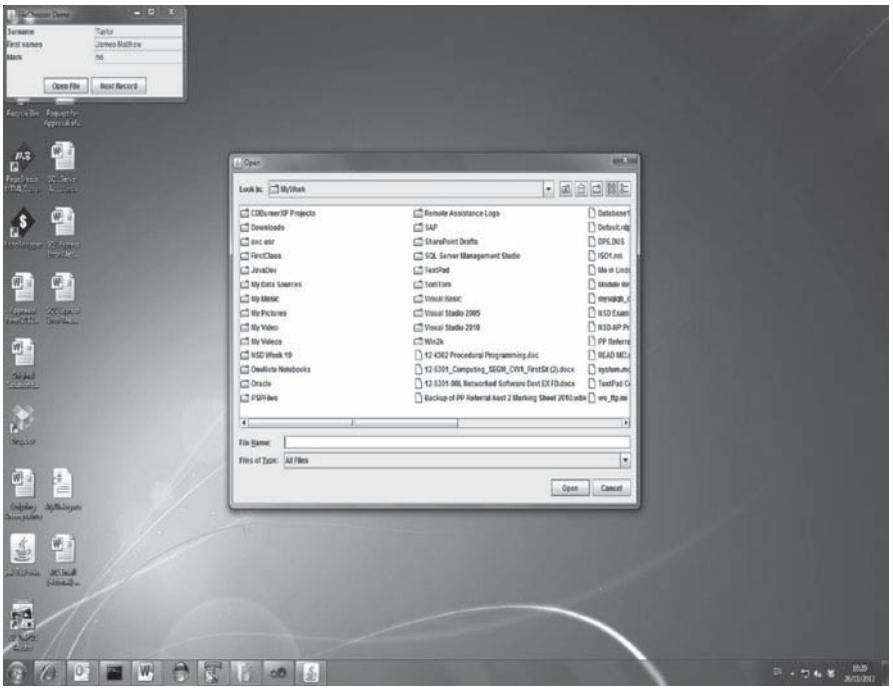

Fig. 4.5 A *JFileChooser* object being used to select a file

The code for this application is shown below. If the reader wishes to create a serial file for testing this program, this may be done very easily by using any text editor to enter the required three fields for each of a series of students (each field being followed by a carriage return).

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.util.*;

public class UseFileChooser extends JFrame
                            implements ActionListener
```

```
{
    private JPanel displayPanel, buttonPanel;
    private JLabel surnameLabel, firstNamesLabel,
                                          markLabel;
    private JTextField surnameBox, firstNamesBox,
                                          markBox;
    private JButton openButton, nextButton;
    private Scanner input;

    public static void main(String[] args)
    {
        UseFileChooser frame = new UseFileChooser();
        frame.setSize(350,150);
        frame.setVisible(true);
    }

    public UseFileChooser()
    {
        setTitle("FileChooser Demo");

        setLayout(new BorderLayout());

        displayPanel = new JPanel();
        displayPanel.setLayout(new GridLayout(3,2));

        surnameLabel = new JLabel("Surname");
        firstNamesLabel = new JLabel("First names");
        markLabel = new JLabel("Mark");
        surnameBox= new JTextField();
        firstNamesBox = new JTextField();
        markBox = new JTextField();

        //For this application, user should not be able
        //to change any records…
        surnameBox.setEditable(false);
        firstNamesBox.setEditable(false);
        markBox.setEditable(false);

        displayPanel.add(surnameLabel);
        displayPanel.add(surnameBox);
        displayPanel.add(firstNamesLabel);
        displayPanel.add(firstNamesBox);
        displayPanel.add(markLabel);
        displayPanel.add(markBox);

        add(displayPanel, BorderLayout.NORTH);

        buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout());

        openButton = new JButton("Open File");
```

```
        openButton.addActionListener(this);
        nextButton = new JButton("Next Record");
        nextButton.addActionListener(this);
        nextButton.setEnabled(false);
        //(No file open yet.)

        buttonPanel.add(openButton);
        buttonPanel.add(nextButton);

        add(buttonPanel, BorderLayout.SOUTH);

        addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing(
                                        WindowEvent event)
                {
                  if (input != null)    //A file is open.
                      input.close();
                  System.exit(0);
                }
            }
        );
    }

    public void actionPerformed(ActionEvent event)
    {
        if (event.getSource() == openButton)
        {
            try
            {
                openFile();
            }
            catch(IOException ioEx)
            {
                JOptionPane.showMessageDialog(this,
                               "Unable to open file!");
            }
        }
        else
        {
            try
            {
                readRecord();
            }
            catch(EOFException eofEx)
            {
                nextButton.setEnabled(false);
```

```
            //(No next record.)
            JOptionPane.showMessageDialog(this,
                            "Incomplete record!\n"
                          + "End of file reached.");
        }
        catch(IOException ioEx)
        {
            JOptionPane.showMessageDialog(this,
                            "Unable to read file!");
        }
    }
}

public void openFile() throws IOException
{
   if (input != null)    //A file is already open, so
                         //needs to be closed first.
    {
        input.close();
        input = null;
    }
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode(
                          JFileChooser.FILES_ONLY);

   int selection = fileChooser.showOpenDialog(null);
    //Window opened in centre of screen.

    if (selection == JFileChooser.CANCEL_OPTION)
        return;

    File results = fileChooser.getSelectedFile();
    if (results ==
              null||results.getName().equals(""))
    //No file name entered by user.
    {
        JOptionPane.showMessageDialog(this,
                            "Invalid selection!");
        return;
    }
    input = new Scanner(results);
    readRecord(); //Read and display first record.
    nextButton.setEnabled(true);   //(File now open.)
}

public void readRecord() throws IOException
{
    String surname, firstNames, textMark;
```

```
        //Clear text fields…
        surnameBox.setText("");
        firstNamesBox.setText("");
        markBox.setText("");

        if (input.hasNext()) //Not at end of file.
        {
            surname = input.nextLine();
            surnameBox.setText(surname);
        }
        else
        {
            JOptionPane.showMessageDialog(this,
                                "End of file reached.");
            nextButton.setEnabled(false);//No next record.
            return;
        }

        //Should cater for possibility of incomplete
        //records…

        if (!input.hasNext())
            throw (new EOFException());

        //Otherwise…
        firstNames = input.nextLine();
        firstNamesBox.setText(firstNames);

        if (!input.hasNext())
            throw (new EOFException());

        //Otherwise…
        textMark = input.nextLine();
        markBox.setText(textMark);
    }
}
```

Note that neither *windowClosing* nor *actionPerformed* can throw an exception, since their signatures do not contain any *throws* clause and we cannot change those signatures. Consequently, any exceptions that do arise must be handled explicitly either by these methods themselves or by methods called by them (as with method *closeFile*).

4.8 ArrayLists

An object of class *ArrayList* is like an array, but can dynamically increase or decrease in size according to an application's changing storage requirements and can hold only references to objects, not values of primitive types. As of Java

SE 5, an individual *ArrayList* can hold only references to instances of a single, specified class. (This restriction can be circumvented by specifying the base type to be *Object*, the ultimate superclass, if a heterogeneous collection of objects is truly required.) Class *ArrayList* is contained within package *java.util*, so this package should be imported by any program wishing to make use of *ArrayList*s.

Constructor overloading allows us to specify the initial size if we wish, but the simplest form of the constructor takes no arguments and assumes an initial capacity of ten. In common with the other member classes of Java's Collection Framework, the class of elements that may be held in an *ArrayList* structure is specified in angle brackets after the collection class name, both in the declaration of the *ArrayList* and in its creation. For example, the following statement declares and creates an *ArrayList* that can hold *String*s:

```
ArrayList<String> stringArray = new ArrayList<String>();
```

This syntax was applicable up to Java SE 6 and is accepted in Java SE 7, but the latter also allows a shortened version of the above syntax, in which the type of elements held inside the collection class is not repeated. Instead, the angle brackets immediately preceding the brackets for the collection class constructor are left empty:

```
ArrayList<String> nameList = new ArrayList<>();
```

Objects are added to the end of an *ArrayList* via method *add* and then referenced/retrieved via method *get*, which takes a single argument that specifies the object's position within the *ArrayList* (numbering from zero, of course). Since the elements of an *ArrayList* (and of any other collection class) are stored as *Object* references (i.e., as references to instances of class *Object*) whilst in the *ArrayList*, it used to be the case that elements of the *ArrayList* had to be typecast back into their original type when being retrieved from the *ArrayList*, but the 'auto-unboxing' feature introduced by Java SE 5 means that they can be retrieved directly now.

Example (Assumes that *ArrayList* is empty at start)

```
String name1 = "Jones";
nameList.add(name1);
String name2 = nameList.get(0);
```

After execution of the above lines, *name1* and *name2* will both reference the string 'Jones'.

An object may be added at a specific position within an *ArrayList* via an overloaded form of the *add* method that takes two arguments, the first of which specifies the position at which the element is to be added.

Example

```
nameList.add(2, "Patterson");  //3rd position.
```

4.9 ArrayLists and Serialisation

It is much more efficient to save a single *ArrayList* to disc than it is to save a series of individual objects. Placing a series of objects into a single *ArrayList* is a very neat way of packaging and transferring our objects. This technique carries another significant advantage: we shall have some form of **random access**, via the *ArrayList* class's *get* method (albeit based on knowing each element's position within the *ArrayList*). Without this, we have the considerable disadvantage of being restricted to serial access only.

Example

This example creates three objects of class *Personnel* (as featured in the example at the end of Sect. 4.6) and uses the *add* method of class `ArrayList` to place the objects into an `ArrayList`. It then employs a 'for-each' loop and the 'get' methods of class *Personnel* to retrieve the data properties of the three objects.

We could use the same `ArrayList` object for sending objects out to the file and for receiving them back from the file, but two `ArrayList` objects have been used below simply to demonstrate beyond any doubt that the values have been read back in (and are not simply the original values, still held in the `ArrayList` object).

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

public class ArrayListSerialise
{
    public static void main(String[] args)
           throws   IOException, ClassNotFoundException
    {
        ObjectOutputStream outStream =
            new ObjectOutputStream(
              new FileOutputStream("personnelList.dat"));
        ArrayList<Personnel> staffListOut =
                                    new ArrayList<>();
        ArrayList<Personnel> staffListIn =
                                    new ArrayList<>();

        Personnel[] staff =
            {new Personnel(123456,"Smith", "John"),
            new Personnel(234567,"Jones", "Sally Ann"),
            new Personnel(999999,"Black", "James Paul")};

        for (int i=0; i<staff.length; i++)
            staffListOut.add(staff[i]);

        outStream.writeObject(staffListOut);

        outStream.close();
        ObjectInputStream inStream =
```

```
            new ObjectInputStream(
              new FileInputStream("personnelList.dat"));

        int staffCount = 0;

        try
        {
            staffListIn =
              (ArrayList<Personnel>)inStream.readObject();
            //The compiler will issue a warning for the
            //above line, but ignore this!

            for (Personnel person:staffListIn)
            {
                staffCount++;
                System.out.println(
                       "\nStaff member " + staffCount);

                System.out.println("Payroll number: "
                                   + person.getPayNum());
                System.out.println("Surname: "
                                   + person.getSurname());
                System.out.println("First names: "
                             + person.getFirstNames());
            }
            System.out.println("\n");
        }
        catch (EOFException eofEx)
        {
            System.out.println(
                           "\n\n*** End of file ***\n");
            inStream.close();
        }
    }
}

class Personnel implements Serializable
 {
    private long payrollNum;
    private String surname;
    private String firstNames;

    public Personnel(long payNum,String sName,
                                          String fNames)
    {

        payrollNum = payNum;
        surname = sName;
        firstNames = fNames;
    }
```

```
    public long getPayNum()
    {
        return payrollNum;
    }
    public String getSurname()
    {
        return surname;
    }
    public String getFirstNames()
    {
        return firstNames;
    }
    public void setSurname(String sName)
    {
        surname = sName;
    }
}
```

Using methods covered in Chap. 2, the above code may be adapted very easily to produce a simple client–server application in which the server supplies personnel details in response to client requests. The only difference is that, instead of sending a series of strings from the server to the client(s), we shall now be passing an *ArrayList*. Consequently, we shall not be making use of a *PrintWriter* object in our server. Instead, we shall need to create an *ObjectOutputStream* object. We do this by passing the *OutputStream* object returned by our server's *Socket* object to the *ObjectOutputStream* constructor, instead of to the *PrintWriter* constructor (as was done previously).

Example

Suppose that the *Socket* object is called *socket* and the output object is called *out*. Then, instead of

```
PrintWriter out =
     new PrintWriter(socket.getOutputStream(),true);
```

we shall have:

```
ObjectOutputStream out =
  new ObjectOutputStream(socket.getOutputStream());
```

Though class *Personnel* was shown in previous examples as being in the same file as the main code, this was merely for convenience of reference. Normally, of course, we would hold this class in a separate file, in order to avoid code duplication and to allow the class's reusability by other applications. The code for the server (*PersonnelServer.java*) and the client (*PersonnelClient.java*) is shown below. You will find that the code for the server is an amalgamation of the first half of *MessageServer.java* from Chap. 2 and the early part of *ArrayListSerialise.java* from

this section, while the code for the client is an amalgamation of the first part of *MessageClient.java* (Chap. 2) and the remainder of *ArrayListSerialise.java*.

As with earlier cases, this example is unrealistically simple, but serves to illustrate all the required steps of a socket-based client–server application for transmitting whole objects, without overwhelming the reader with unnecessary detail. Upon receiving the message 'SEND PERSONNEL DETAILS' from a client, the server simply transmits the *ArrayList* containing the three *Personnel* objects used for demonstration purposes in this section.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class PersonnelServer
{
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;
    private static Socket socket;
    private static ArrayList<Personnel> staffListOut;
    private static Scanner inStream;
    private static ObjectOutputStream outStream;

    public static void main(String[] args)
    {
        System.out.println("Opening port…\n");

        try
        {
            serverSocket = new ServerSocket(PORT);
        }
        catch(IOException ioEx)
        {
            System.out.println(
                        "Unable to attach to port!");
            System.exit(1);
        }

        staffListOut = new ArrayList<Personnel>();

        Personnel[] staff =
          {new Personnel(123456,"Smith", "John"),
           new Personnel(234567,"Jones", "Sally Ann"),
           new Personnel(999999,"Black", "James Paul")};

        for (int i=0; i<staff.length; i++)
            staffListOut.add(staff[i]);
        startServer();
    }
```

```
    private static void startServer()
    {
        do
        {
            try
            {
                socket = serverSocket.accept();

                inStream =
                  new Scanner(socket.getInputStream());

                outStream =
                    new ObjectOutputStream(
                             socket.getOutputStream());
              /*
              The above line and associated declaration
              are the only really new code featured in
              this example.
              */

                String message = inStream.nextLine();
                if (message.equals(
                             "SEND PERSONNEL DETAILS"))
                {
                   outStream.writeObject(staffListOut);
                    outStream.close();
                }

                System.out.println(
                         "\n* Closing connection… *");
                socket.close();
            }
            catch(IOException ioEx)
            {
                ioEx.printStackTrace();
            }
        }while (true);
    }
}
```

The only new point worthy of note in the code for the client is the necessary inclusion of `throws ClassNotFoundException`, both in the method that directly accesses the *ArrayList* of *Personnel* objects (the *run* method) and in the method that calls this one (the *main* method)…

```
import java.io.*;
import java.net.*;
import java.util.*;
```

```
public class PersonnelClient
{
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args)
                          throws ClassNotFoundException
    {
        try
        {
            host = InetAddress.getLocalHost();
        }
        catch(UnknownHostException uhEx)
        {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        talkToServer();
    }

    private static void talkToServer()
                          throws ClassNotFoundException
    {
        try
        {
            Socket socket = new Socket(host,PORT);

            ObjectInputStream inStream =
                       new ObjectInputStream(
                              socket.getInputStream());

            PrintWriter outStream =
                new PrintWriter(
                         socket.getOutputStream(),true);

            //Set up stream for keyboard entry…
            Scanner userEntry = new Scanner(System.in);

            outStream.println("SEND PERSONNEL DETAILS");
            ArrayList<Personnel> response =
            (ArrayList<Personnel>)inStream.readObject();
            /*
            As in ArrayListSerialise, the compiler will
            issue a warning for the line above.
            Simply ignore this warning.
            */

            System.out.println(
                       "\n* Closing connection… *");
            socket.close();
```

```
            int staffCount = 0;

            for (Personnel person:response)
            {
                staffCount++;
                System.out.println(
                       "\nStaff member " + staffCount);
                System.out.println("Payroll number: "
                                   + person.getPayNum());
                System.out.println("Surname: "
                                  + person.getSurname());

                System.out.println("First names: "
                                + person.getFirstNames());
            }
            System.out.println("\n\n");
        }
        catch(IOException ioEx)
        {
            ioEx.printStackTrace();
        }
    }
}
```

The only change required to the code for the *Personnel* class (now in its separate file) will be the inclusion of `java.io.Serializable` in the header line (since it is no longer subject to the import of package *java.io*):

```
class Personnel implements java.io.Serializable
```

Figure 4.6 shows a client accessing the server, while Fig. 4.7 shows the corresponding output at the server end.

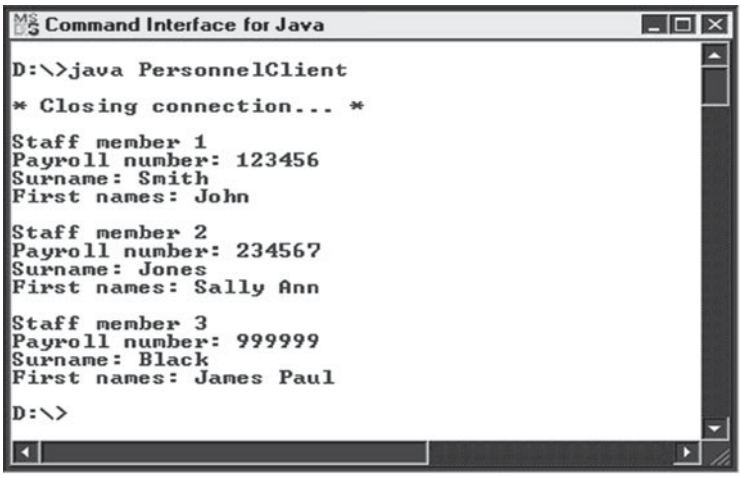

Fig. 4.6 Client using *ObjectInputStream*s to retrieve '*ArrayList*ed' data from a server

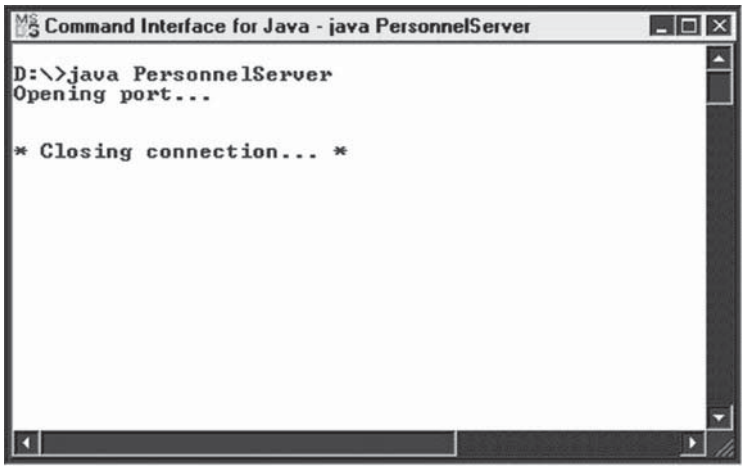

Fig. 4.7 Server providing '*Vector*ised' data to client in preceding screenshot

4.10 Vectors Versus ArrayLists

As an alternative to using an *ArrayList* to 'wrap up' our class objects, we could choose to use a *Vector*. This class existed in Java before the Collection Framework came into existence and was 'retro-fitted' to take its place alongside the other Collection classes. The only significant difference is that, instead of using method *get* to retrieve an object from a *Vector*, we need to use method *elementAt*.

Example

```
Vector<String> stringVector = new Vector<>();

stringVector.add("Example");
//Next step retrieves this element.
String word = stringVector.elementAt(0);
```

Another slight difference is that, as well as having method *add* to add objects to a *Vector*, there is also method *addElement*, principally to cater for older code.

The following question naturally arises: 'Which is it better to use—Vectors or ArrayLists?'. The answer is… it depends! The *ArrayList* is faster, but is not thread-safe, whereas the *Vector* **is** thread-safe. Consequently, if thread-safety is important to your program, you should use a *Vector*. If, on the other hand, thread-safety is not a factor, then you should use an *ArrayList*.