Working with Files

18

Outcomes:

By the end of this chapter you should be able to:

- *explain the principles of* ***input*** *and* ***output*** *and identify a number of different input and output devices*;
- *explain the concept of an* ***I/O stream***;
- *describe the basic file-handling techniques used in the Java language*;
- *distinguish between* ***text***, ***binary*** *and* ***object*** *encoding of data*;
- *distinguish between* ***serial*** *access files and* ***random*** *access files*;
- *create and access files in Java using all the above encoding and access methods.*

18.1 Introduction

When we developed our case study in Chaps. 11 and 12 it became apparent that in reality an application such as that one wouldn't be much use unless we had some way of storing our data permanently—even when the program has been terminated and the computer has been switched off. You will remember in those chapters, because you had not yet learnt how to do this, we provided a special class called `TenantFileHandler` that enabled you to keep permanent records on disk.

Now it is time to learn how to do this yourself. As you are no doubt already aware, a named block of externally stored data is called a **file**.

When we are taking an object-oriented approach, as we have been doing, we tend not to separate the data from the behaviour; however, when it comes to storing information in files then of course it is only the data that we are interested in storing. When referring to data alone it is customary to use the terms **record** and **field**. A record refers to a single data instance—for example a person, a stock-item, a

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_18

student and so on; a **field** refers to what in the object-oriented world we would normally call an attribute—a name, a stock number, an exam mark etc.

In this chapter we will learn how to create files, and write information to them, and to read the information back when we need it. We start by looking at this process in the overall context of input and output, or I/O as it is often called; you will then go on to learn a number of different techniques for keeping permanent copies of your data.

18.2 Input and Output

Any computer system must provide a means of allowing information to come in from the outside world (**input**) and, once it has been processed, to be sent out again (**output**). The whole question of input and output, particularly where files are concerned, can sometimes seem rather complex, especially from the point of view of the programmer.

As with all aspects of a computer system, the processes of input and output are handled by the computer hardware working in conjunction with the system software —that is, the operating system (Windows™, macOS™ or Linux™ for example). The particular application program that is running at the time normally deals with input and output by communicating with the operating system, and getting it to perform these tasks in conjunction with the hardware.

All this involves some very real complexity and involves a lot of low-level details that a programmer is not usually concerned with; for example, the way in which the system writes to external media such as disks, or the way it reconciles the differences between the speed of the processor with the speed of the disk-drive.

18.3 Input and Output Devices

The most common way of getting data input from the outside world is via the keyboard; and the most common way of displaying output data is on the screen. Therefore, most systems are normally set up so that the *standard* input and output devices are the keyboard and the screen respectively. However, there are many other devices that are concerned with input and output: magnetic, solid state and optical disks for permanent storage of data (both local and remote); flash drives; network interface cards and modems for communicating with other computers; and printers for producing hard copies.

We should bear in mind that the process, in one sense, is always the same, no matter what the input or output device. All the data that is processed by the computer's central processing unit in response to program instructions is stored in the computer's main memory or RAM (Random Access Memory). Input is the transfer of data from some external device to main memory whereas output is the

transfer of data from main memory to an external device. In order for input or output to take place, it is necessary for a channel of communication to be established between the device and the computer's memory. Such a channel is referred to as a **stream**. The operating system will have established a **standard input stream** and a **standard output stream**, which will normally be the keyboard and screen respectively. In addition to this, there is usually a **standard error stream** where error messages can be displayed; this is normally also set to the screen. All of these default settings for the standard streams can be changed either via the operating system or from within the program itself.

In previous chapters you have seen that the `System` class has two attributes called `in` and `out`. In addition to this, it has an additional attribute called `err`; these objects are already set up to provide access to the standard input, output and error streams. The attribute `in` is an object of a class called `InputStream`. This class provides some low-level methods to deal with basic input—they are low-level because they deal with sequences of bytes, rather than characters. A higher-level class, `InputStreamReader` can be wrapped around this class to deal with character input; `InputStreamReader` objects can subsequently be wrapped by another class, `BufferedReader`, which handles input in the form of strings.

This rather complex way of reading from the keyboard is how things were done before Java 5.0 provided the `Scanner` class. The following program illustrates how you would get keyboard input in this manner.

KeyboardInput

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class KeyboardInput
{
   public static void main(String[] args)
   {
      InputStreamReader input = new InputStreamReader(System.in); // to handle low-level details
      BufferedReader reader = new BufferedReader(input); // to handle high-level details

      try
      {
         System.out.print("Enter a string: ");
         String test = reader.readLine();  // gets a string of characters from the keyboard
         System.out.println("You entered: " + test);
      }

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

You can see that the `readLine` method of `BufferedReader` is used to get a string of characters from the keyboard; as you would expect, the method reads characters from the keyboard until the user presses the enter key. It throws an `IOException` if an error (such as a keyboard lock) occurs during the process, and this has to be handled.

In this chapter, instead of dealing with input and output to the standard streams, we are going to be dealing with the input and output of data to external disk drives in the form of files—but, as you will see, the principles are the same.

18.4 File-Handling

The output process, which consists of transferring data from memory to a file, is usually referred to as **writing**; the input process, which consists of transferring data from a file to memory, is referred to as **reading**. Both of these involve some low-level detail to do with the way in which data is stored physically on the disk. As programmers we do not want to have to worry more than is necessary about this process—which, of course, will differ from one machine to the next and from one operating system to the next. Fortunately, Java makes it quite easy for us to deal with these processes. As we shall see, Java provides low-level classes which create **file streams**—input or output streams that handle communication between main memory and a named file on a disk. It also provides higher-level classes which we can "wrap around" the low-level objects, enabling us to use methods that relate more closely to our logical way of thinking about data. In this way we are shielded from having to know too much detail about the way our particular system stores and retrieves data to or from a file.

As we shall see, this whole process enables us to read and write data in terms of units that we understand—for example, in the form of strings, lines of text, or basic types such integers or characters; Java even allows us to store and retrieve whole objects.

18.4.1 Encoding

Java supports three different ways of **encoding** data—that is, representing data on a disk. These are **text**, **binary** and **object**.

Text encoding means that the data on the disk is stored as characters in the form used by the external system (often ASCII). Java, as we know, uses the Unicode character set, so, depending on the form used by the external system, some conversion might take place in the process, but fortunately the programmer does not have to worry about that. As an example, consider saving the number 107 to a text file—it will be saved as the character '1' in ASCII code (or whatever is used by the system) followed by the character '0', followed by the character '7'. A text file is therefore readable by a text editor (such as Windows™ Notepad).

Binary encoding, on the other hand, means that the data is stored in the same format as the internal representation of the data used by the program to store data in memory. So the number 107 would be saved as the binary number 1101011. A binary file could not be read properly by a text editor as we shall see in Sect. 18.6.

Finally, there is object-encoding which is a powerful mechanism provided by Java whereby a whole object can be input or output with a single command.

You are probably asking yourself which is the best method to use when you start to write applications that read and write to files. Well, if your files are going to be read and written by the same application, then it really makes very little difference how they are encoded. Just use the method that seems the easiest for the type of data you are storing. However, do bear in mind that if you wanted your files to be read by a text editor then you must, of course, use the text encoding method.

18.4.2 Access

The final thing that you need to consider before we show you how to write files in Java is the way in which files are accessed. There are two ways in which this can take place—**serial** access and **random** access. In the first (and more common) method, each item of data is read (or written) in turn. The operating system provides what is known as a **file pointer**, which is really just a location in memory that keeps track of where we have got to in the process of reading or writing to a file.

Another way to access data in a file is to go directly to the record you want—this is known as random access, and is a bit like going straight to the clip you want on a DVD; whereas serial access is like using an old fashioned video tape, where you have to work your way through the entire tape to get to the bit you want. Java provides a class (`RandomAccessFile`) that we can use for random access. We will start, however, with serial access.

18.5 Reading and Writing to Text Files

In this and the following section we are going to use as an example a very simple class called `Car`; the code for this class is given below:

The *Car* class

```
public class Car
{
    private String registration;
    private String make;
    private double price;

    public Car(String registrationIn, String makeIn, double priceIn)
    {
        registration = registrationIn;
        make = makeIn;
        price = priceIn;
    }

    public String getRegistration()
    {
        return registration;
    }

    public String getMake()
    {
        return make;
    }

    public double getPrice()
    {
        return price;
    }
}
```

The program below, `TextFileTester`, is a very simple menu-driven program that manipulates a list of cars, held in memory as a `List`; it provides the facility to add new cars to the list, to remove cars from the list and to display the details of all the cars in the list. As it is a demonstration program only, we have not bothered with such things as input validation, or checking if the list is empty before we try to remove an item.

The difference between this and other similar programs that we have discussed before, is that the list is kept as a permanent record—as we mentioned before, we did a similar thing in our case study in Chap. 12, but there the process was hidden from you.

The program is designed so that reading and writing to the file takes place as follows: when the quit option is selected, the list is written as a permanent text file called `Cars.txt`; each time the program is run, this file is read into memory.

The program is presented below; notice that we have provided two helper methods, `writeList` and `readList` for the purpose of accessing the file; as we shall explain, the `writeList` method also deals with creating the file for the first time. Notice also that, for convenience, we are making use of the `EasyScanner` class that we developed in Chap. 8.

TextFileTester

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

public class TextFileTester
{
   public static void main(String[] args)
   {
       char choice;

       // create an empty list to hold Cars
       List<Car> carList = new ArrayList<>();

       // read the list from file when the program starts
       readList(carList);

       // menu options
       do
       {
           System.out.println("\nText File Tester");
           System.out.println("1. Add a car");
           System.out.println("2. Remove a car");
           System.out.println("3. List all cars");
           System.out.println("4. Quit\n");
           choice = EasyScanner.nextChar();
           System.out.println(); switch(choice)
           {
               case '1' :addCar(carList);
                         break;
               case '2' :removeCar(carList);
                         break;
               case '3' :listAll(carList);
                         break;
               case '4' :writeList(carList); // write to the file
                         break;
               default : System.out.print("\nPlease choose a number from 1 - 4 only\n ");
           }
       }while(choice != '4');
   }

   // method for adding a new car to the list
   static void addCar(List<Car> carListIn)
   {
       String tempReg;
       String tempMake;
       double tempPrice;

       System.out.print("Please enter the registration number: ");
       tempReg = EasyScanner.nextString();
       System.out.print("Please enter the make: ");
       tempMake = EasyScanner.nextString();
       System.out.print("Please enter the price: ");
       tempPrice = EasyScanner.nextDouble();
       carListIn.add(new Car(tempReg, tempMake, tempPrice));
   }

   /* method for removing a car from the list - in a real application this would need to include
      some validation */
   static void removeCar(List<Car> carListIn)
   {
       int pos;
       System.out.print("Enter the position of the car to be removed: ");
       pos = EasyScanner.nextInt();
       carListIn.remove(pos - 1);
   }

   // method for listing details of all cars in the list
   static void listAll(List<Car> carListIn)
```

```
    {
        for(Car item : carListIn)
        {
            System.out.println(item.getRegistration()
            + " "
            + item.getMake()
            + " "
            + item.getPrice());
        }
    }

    // method for writing the file
    static void writeList(List<Car> carListIn)
    {
        // use try-with-resources to ensure file is closed safely
        try(
               /* create a FileWriter object, carFile, that handles the low-level details of writing
               the list to a file which we have called "Cars.txt" */
               FileWriter carFile = new FileWriter("Cars.txt");
               /* now create a PrintWriter object to wrap around carFile; this allows us to user
               high-level functions such as println */
               PrintWriter carWriter = new PrintWriter(carFile);
           )
        {
            // write each element of the list to the file
            for(Car item : carListIn)
            {
               carWriter.println(item.getRegistration());
               carWriter.println(item.getMake());
               carWriter.println(item.getPrice()); // println can accept a double, and write it as a string
            }
        }
        // handle the exception thrown by the FileWriter methods
        catch(IOException e)
        {
            System.out.println("There was a problem writing the file");
        }
    }

    // method for reading the file
    static void readList(List<Car> carListIn)
    {
        String tempReg;
        String tempMake;
        String tempStringPrice;
        double tempDoublePrice;

        // use try-with-resources to ensure file is closed safely
        try(
                /* create a FileReader object, carFile, that handles the lowlevel details of reading
                the list from the "Cars.txt" file */
                FileReader carFile = new FileReader("Cars.txt");
                /* now create a BufferedReader object to wrap around carFile; this allows us to user
                high-level functions such as readLine */
                BufferedReader carStream = new BufferedReader(carFile);
            )
        {
            // read the first line of the file
            tempReg = carStream.readLine();
            /* read the rest of the first record, then all the rest of the records until the end of
               the file is reached */
            while(tempReg != null) // a null string indicates end of file
            {
                tempMake = carStream.readLine();
                tempStringPrice = carStream.readLine();
                // as this is a text file we have to convert the price to double
                tempDoublePrice = Double.parseDouble(tempStringPrice);
                carListIn.add(new Car(tempReg, tempMake, tempDoublePrice));
                tempReg = carStream.readLine();
            }
        }

        // handle the exception that is thrown by the FileReader constructor if the file is not found
        catch(FileNotFoundException e)
        {
            System.out.println("\nNo file was read");
        }

        // handle the exception thrown by the FileReader methods
        catch(IOException e)
        {
            System.out.println("\nThere was a problem reading the file");
        }
    }
}
```

It is only the `writeList` and `readList` methods that we need to analyse here—none of the other methods involves anything new. Let's start with `writeList`:

```
// method for writing the file
static void writeList(List<Car> carListIn)
{
   // use try-with-resources to ensure file is closed safely
   try(
         /* create a FileWriter object, carFile, that handles the low-level details of writing
         the list to a file which we have called "Cars.txt" */
          FileWriter carFile = new FileWriter("Cars.txt");
         /* now create a PrintWriter object to wrap around carFile; this allows us to user
         high-level functions such as println */
          PrintWriter carWriter = new PrintWriter(carFile);
      )
   {
      // write each element of the list to the file
      for(Car item : carListIn)
      {
         carWriter.println(item.getRegistration());
         carWriter.println(item.getMake());
         carWriter.println(item.getPrice()); // println can accept a double, and write it as a string
      }
   }
   // handle the exception thrown by the FileWriter methods
   catch(IOException e)
   {
      System.out.println("There was a problem writing the file");
   }
}
```

The first thing to notice is that we have enclosed everything in a **`try`** block. Here we are creating our files using the *try-with-resources* mechanism that was introduced to you in Chap. 14. We use this because the file has to be closed after we finish using it. Closing the file achieves two things. First, it ensures that a special character, the end-of-file marker,[1] is written at the end of the file. This enables us to detect when the end of the file has been reached when we are reading it—more about this when we explore the `readList` method. Second, closing the file means that it is no longer accessible by the program, and is therefore not susceptible to being written to in error.

Using *try-with-resources* ensures that the file is always closed, no matter what other errors or exceptions may have occurred—before the advent of *try-with-resources* we would have had to specifically close the file by calling the `close` method of `PrintWriter` (this would have been done in a **`finally`** clause). As you can see, the instructions for opening the file have been placed in the brackets after the **`try`** keyword. We will use *try-with-resources* to create and open files throughout this chapter.

The file is opened by using a class called `FileWriter`; this is one of the classes we talked about earlier that provide the low-level communication between the program and the file. By opening a file we establish a *stream* through which we can output data to the file. We create a `FileWriter` object, `carFile`, giving it the name of the file to which we want to write the data:

```
FileWriter carFile = new FileWriter("Cars.txt");
```

In this case we have called the file `Cars.txt`.[2] Creating the new `FileWriter` object causes the file to be opened in output mode—meaning that it is ready to receive data; if no file of this name exists then one will be created. Opening the file in this way (in output mode) means that any data that we write to the file will

[1] Most systems use Unicode character 26 as the end-of-file marker.
[2] As we have not supplied an absolute pathname, the file will be saved in the current directory.

wipe out what was previously there. That is what we need for this particular application, because we are simply going to write the entire list when the program terminates. Sometimes, however, it is necessary to open a file in **append** mode; in this mode any data written to the file would be written after the existing data. To do this we would simply have used another constructor, which takes an additional (**`boolean`**) parameter indicating whether or not we require append mode:

```
FileWriter carFile = new FileWriter("Cars.txt", true);
```

The next thing we do is create an object, `carWriter`, of the `PrintWriter` class, sending it the `carFile` object as a parameter.

```
PrintWriter carWriter = new PrintWriter(carFile);
```

This object can now communicate with our file via the `carFile` object; `PrintWriter` objects have higher level methods than `FileWriter` objects (for example `print` and `println`) that enable us to write whole strings like we do when we output to the screen.

Now we are ready to write each `Car` in the list to our file—we can use a **`for`** loop for this:

```
for(Car item : carListIn)
{
    carWriter.println(item.getRegistration());
    carWriter.println(item.getMake());
    carWriter.println(item.getPrice());
}
```

On each iteration we use the `println` method of our `PrintWriter` object, `carWriter`, to write the registration number, the make and the price of the car to the file; `println` converts the price to a `String` before writing it. Notice also that the `println` method inserts a newline character at the end of the string that it prints; if we did not want the newline character to be inserted, we would use the print method instead.

Finally we have to handle any `IOExceptions` that may be thrown by the `FileWriter` methods:

```
catch(IOException e)
{
    System.out.println("There was a problem writing the file");
}
```

In a moment we will explore the code for reading the file. But bear in mind that if we were to run our program and add a few records, and then quit the program we would have saved the data to a text-file called `Cars.txt`, so we should be able to read this file with a text editor. When we did this, we created three cars, and then looked inside the file using Windows™ Notepad. Figure 18.1 shows the result.

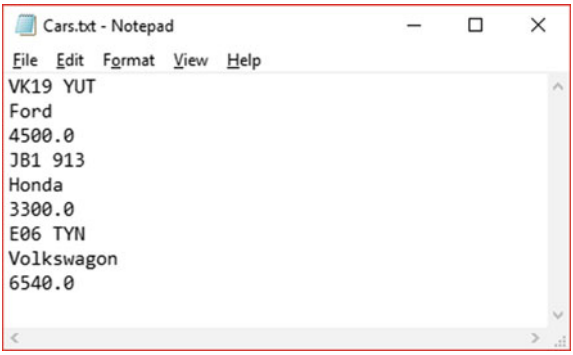


Fig. 18.1 Viewing a text file with windows notepad

As we have written each field using the `println` statement, each one, as you can see, starts on a new line. If our aim were to view the file with a text editor as we have just done, then this might not be the most suitable format—we might, for example, have wanted to have one record per line; we could also have printed some headings if we had wished. However, it is actually our intention to make our program read the entire file into our list when the program starts—and as we shall now see, one field per line makes reading the text file nice and easy. So let's take a look at our `readList` method:

```
// method for reading the file
static void readList(List<Car> carListIn)
{
    String tempReg;
    String tempMake;
    String tempStringPrice;
    double tempDoublePrice;

    // use try-with-resources to ensure file is closed safely
    try(
            /* create a FileReader object, carFile, that handles the lowlevel details of reading
            the list from the "Cars.txt" file */
            FileReader carFile = new FileReader("Cars.txt");
            /* now create a BufferedReader object to wrap around carFile; this allows us to user
            high-level functions such as readLine */
            BufferedReader carStream = new BufferedReader(carFile);

       )
    {
       // read the first line of the file
       tempReg = carStream.readLine();
       /* read the rest of the first record, then all the rest of the records until the end of
          the file is reached */
       while(tempReg != null) // a null string indicates end of file
       {
           tempMake = carStream.readLine();
           tempStringPrice = carStream.readLine();
           // as this is a text file we have to convert the price to double
           tempDoublePrice = Double.parseDouble(tempStringPrice);
           carListIn.add(new Car(tempReg, tempMake, tempDoublePrice));
           tempReg = carStream.readLine();
       }
    }

    // handle the exception that is thrown by the FileReader constructor if the file is not found
    catch(FileNotFoundException e)
    {
        System.out.println("\nNo file was read");
    }

    // handle the exception thrown by the FileReader methods
    catch(IOException e)
    {
        System.out.println("\nThere was a problem reading the file");
    }
}
```

First, we have declared some variables to hold the value of each field as we progressively read through the file. Remembering that this is a text file we have declared three `String`s:

```
String tempReg;
String tempMake;
String tempStringPrice;
```

The last of these will have to be converted to a **`double`** before we store it in the list so we also need a variable to hold this value once it is converted:

```
double tempDoublePrice;
```

Now, as before, we put everything into a **`try`** block, as we are going to have to deal with the exceptions that may be thrown by the various methods we will be using. Again we are using *try-with-resources*, so the process of opening the file is place in the brackets after the **`try`**. As you can see from the code, we start by creating an object—`carFile`—of the class `FileReader` which deals with the low-level details involved in the process of reading a file. The name of the file, `Cars.txt`, that we wish to read is sent in as a parameter to the constructor; this file is then opened in read mode.

```
FileReader carFile = new FileReader("Cars.txt");
```

Now, in order that we can use some higher-level read methods, we wrap up our `carFile` object in an object of a class called `BufferedReader`. We have called this new object `carStream`.

```
BufferedReader carStream = new BufferedReader(carFile);
```

Now we are going to read each field of each record in turn, so we will need some sort of loop. The only problem is to know when to stop—this is because the number of records in the file can be different each time we run the program. There are different ways in which to approach this problem. One very good way (although not the one we have used here), if the same program is responsible for both reading and writing the file, is simply to write the total number of records as the first item when the file is written. Then, when reading the file, this item is the first thing to be read —and once this number is known a **`for`** loop can be used to read the records.

However, it may well be the case that the file was written by another program (such as a text editor). In this case it is necessary to check for the end-of-file marker that we spoke about earlier. In order to help you understand this process we are using this method here, even though we could have used the first (and perhaps simpler) method.

This is what we have to do: we have to read the first field of each record, then check whether that field began with the end-of-file marker. If it did, we must stop reading the file, but if it didn't we have to carry on and read the remaining fields of that record. Then we start the process again for the next record.

Some pseudocode should make the process clear; we have made this pseudocode specific to our particular example:

```
BEGIN
    READ the registration number field of the first record
    LOOP while the field just read does not contain the end-of-file marker
    BEGIN
        READ the make field of the next record
        READ the price field of the next record
        CONVERT the price to a double
        CREATE a new car with details just read and add it to the list
        READ the registration number field of the next record
    END
END
```

The code for this is shown below:

```
tempReg = carStream.readLine();
while(tempReg != null) // a null string indicates end of file
{
    tempMake = carStream.readLine();
    tempStringPrice = carStream.readLine();
    tempDoublePrice = Double.parseDouble(tempStringPrice);
    carListIn.add (new Car(tempReg, tempMake, tempDoublePrice));
    tempReg = carStream.readLine();
}
```

Notice that we are using the `readLine` method of `BufferedReader` to read each record. This method reads a line of text from the file; a line is considered anything that is terminated by the newline character. The method returns that line as a `String` (which does not include the newline character). However, if the line read consists of the end-of-file marker, then `readLine` returns a **`null`**, making it very easy for us to check if the end of the file has been reached. In Sect. 18.7 you will be able to contrast this method of `BufferedReader` with the `read` method, which reads a single character only.

Finally, we must handle any exceptions that may be thrown by the methods of `FileReader`; first, the constructor throws a `FileNotFoundException` if the file is not found:

```
catch(FileNotFoundException e)
{
    System.out.println("\nNo file was read");
}
```

All the other methods may throw `IOExceptions`:

```
catch(IOException e)
{
    System.out.println("\nThere was a problem reading the file");
}
```

18.6 Reading and Writing to Binary Files

In many ways, it makes little difference whether we store our data in text format or binary format; but it is, of course, important to know the sort of file that we are dealing with when we are reading it. For example, in the previous section you saw that we needed to convert a `String` to a **`double`** when it came to handling the price of a car. However, it is important for you to be familiar with the ways of handling both types of file, so now we will show you how to read and write data to a binary file using exactly the same example as before.

The only difference in our program will be the `writeList` and `readList` methods. First let's look at the code for the new `writeList` method:

```
static void writeList(List<Car> carListIn)
{
       // use try-with-resources to ensure file is closed safely
       try(
             FileOutputStream carFile = new FileOutputStream("Cars.bin");
             DataOutputStream carWriter = new DataOutputStream(carFile);
          )
       {

          for(Car item : carListIn)
          {
             carWriter.writeUTF(item.getRegistration());
             carWriter.writeUTF(item.getMake());
             carWriter.writeDouble(item.getPrice());
          }

       }

       catch(IOException e)
       {
          System.out.println("There was a problem writing the file");
       }
```

You can see that the process is similar to the one we used to write a text file, but here the two classes that we are using are `FileOutputStream` and `DataOutputStream` which deal with the low-level and high-level processes respectively. The `DataOutputStream` class provides methods such as `writeDouble`, `writeInt` and `writeChar` for writing all the basic scalar types, as well a method called `writeUTF` for writing strings. UTF stands for *Unicode Transformation Format*, and the method is so-called because, when it writes the string to a file, it converts the Unicode characters (which are used in Java) to the machine-specific format.

Before moving on to the `readList` method it is worth reminding ourselves that a file written in this way—that is, a binary file—cannot be read by a text editor. And to prove the point, Fig. 18.2 shows the result of trying to read such a file in Windows™ Notepad.

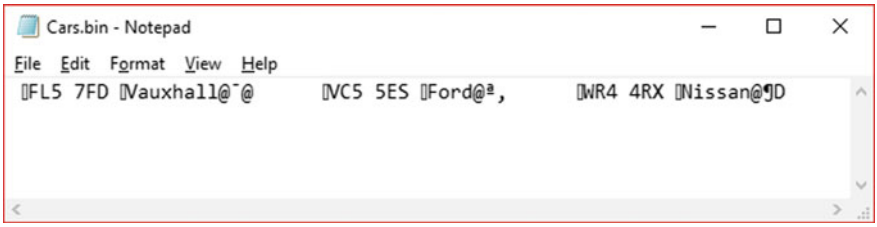


Fig. 18.2 Trying to read a binary file with a text editor

So now we can look at the `readList` method:

```
static void readList(List<Car> carListIn)
{
    String tempReg;
    String tempMake;
    double tempPrice;
    boolean endOfFile = false;

    // use try-with-resources to ensure file is closed safely
    try(
           FileInputStream carFile = new FileInputStream("Cars.bin");
           DataInputStream carStream = new DataInputStream(carFile);
       )
    {
       while(endOfFile == false)
       {
           try
           {
              tempReg = carStream.readUTF();
              tempMake = carStream.readUTF();
              tempPrice = carStream.readDouble();
              carListIn.add(new Car(tempReg, tempMake, tempPrice));
           }
           catch(EOFException e)
           {
              endOfFile = true;
           }
       }

    }
    catch(FileNotFoundException e)
    {
       System.out.println("\nThere are currently no records");
    }

    catch(IOException e)
    {
       System.out.println("There was a problem reading the file");
    }
}
```

You can see that the two classes we use for reading binary files are `FileInputStream` for low-level access and `DataInputStream` for the higher-level functions; they have equivalent methods to those we saw previously when writing to files.

The most important thing to observe in this method is the way we test whether we have reached the end of the file. In the case of a binary file we can do this by making use of the fact that the `DataInputStream` methods throw `EOFExceptions` when an end of file marker has been detected during a read operation. So all we have to do is declare a **`boolean`** variable, `endOfFile`, which we initially set to **`false`**, and we use this as the termination condition in the **`while`** loop. Then we enclose our read operations in a **`try`** block, and, when an exception is thrown, `endOfFile` is set to **`true`** within the **`catch`** block, causing the **`while`** loop to terminate.

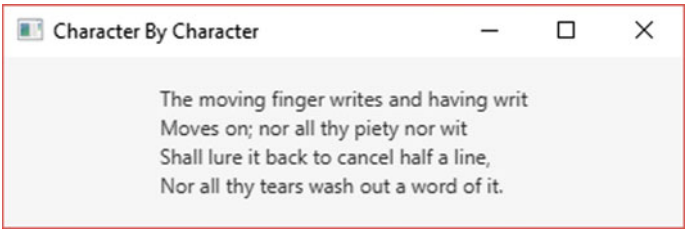


Fig. 18.3 Reading a file character by character

18.7 Reading a Text File Character by Character

As you will have realized by now, there are many ways in which we can deal with handling files, and the methods we choose will depend largely on what it is we want to achieve.

In this section we will show you how to read a text file character by character—this is a useful technique if we do not know anything about the structure of the file.

We have written a JavaFX application which reads a text file, `Poem.txt`, character by character, and builds a string by appending each character as it is read. The process continues until the end of the file is reached, or until a stipulated number of characters have been read. We put this last condition in as a safeguard in case the user should try to display a very large file by mistake.

Once the reading process has finished, the string is displayed by adding it to a label, as shown in Fig. 18.3.

Here is the code for the application:

CharacterByCharacter

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CharacterByCharacter extends Application
{
    private Label viewArea = new Label();

    @Override
    public void start(Stage stage)
    {
        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(viewArea);
        Scene scene = new Scene(root, 400, 100);
        stage.setScene(scene);
        stage.setTitle("Character By Character");
        stage.show();

        readAndWrite();
    }
    private void readAndWrite()
```

```
    {
          // use try-with-resources to ensure file is closed safely
          try(
                FileReader testFile = new FileReader("Poem.txt");
                BufferedReader textStream = new BufferedReader(testFile);
              )
          {
              String str = new String();
              final int MAX = 1000;

              int ch; // to hold the integer (Unicode) value of the character
              char c; // to hold the character when type cast from integer
              int counter = 0; // to count the number of characters read so far
              ch = textStream.read(); // read the first character from the file
              c = (char) ch; // type cast from integer to character
              /* continue through the file until either the end of the file is
                 reached (in which case -1 is returned) or the maximum number of
                 characters stipulated have been read */
              while( ch != -1 && counter <= MAX)
              {
                  counter++; // increment the counter
                  str = str + c;
                  ch = textStream.read(); // read the next character
                  c = (char) ch;
              }

              str = str + "\n";
              viewArea.setText(str);
          }

          catch(IOException ioe)
          {
              viewArea.setText("There was a problem reading the file\n");
          }
    }

    public static void main(String[] args)
    {
          launch(args);
    }

}
```

As you can see, we have placed the functionality in a helper method, `readAndWrite`. The main thing to notice here is that we are using the `read` method of `BufferedReader`; this method reads a single character from the file and returns an integer, the Unicode value of the character read. If the character read was the end-of-file marker then it returns −1, making it an easy matter for us to check whether the end of the file has been reached. In the above example, as explained earlier, we stop reading the file if we have reached the end or if more than the maximum number of characters allowed has been read; here we have set that maximum to 1000. You can see that in the above method, after each read operation, we type cast the integer to a character, which we then append to the string.

18.8 Object Serialization

If you are going to be dealing with files that will be accessed only within a Java program, then one of the easiest ways to do this is to make use of two classes called `ObjectInputStream` and `ObjectOutputStream`. These classes have methods called, respectively, `readObject` and `writeObject` that enable us to read and write whole objects from and to files. The process of converting an object into a stream of data suitable for storage on a disk is called **serialization**.

Any class whose objects are to be read and written using the above methods must implement the interface `Serializable`. This is a type of interface that we have not actually come across before—it is known as a **marker** and in fact contains

no methods. Its purpose is simply to make an "announcement" to anyone using the class; namely that objects of this class can be read and written as whole objects. In designing a class we can, then, choose not to make our class `Serializable`—we might want to do this for security reasons (for example, to stop whole objects being transportable over the Internet) or to avoid errors in a distributed environment where the code for the class was not present on every machine.

In the case of our `Car` class, we therefore need to declare it in the following way before we could use it in a program that handles whole objects:

```
public class Car implements Serializable
```

The `Serializable` interface resides within the `java.io` package.

Now we can re-write the `writeList` and `readList` methods of `TextFileTester` so that we manipulate whole objects. First the `writeList` method:

```
static void writeList(List<Car> carListIn)
{
     // use try-with-resources to ensure file is closed safely
     try(
             FileOutputStream carFile = new FileOutputStream("Cars.dat");
             ObjectOutputStream carStream = new ObjectOutputStream(carFile);
        )
     {

        for(Car item : carListIn)
        {
            carStream.writeObject(item);
        }

     }
     catch(IOException e)
     {
        System.out.println("There was a problem writing the file");
     }
}
```

You can see how easy this is—you just need one line to save a whole object to a file by using the `writeObject` method of `ObjectOutputStream`.

Now the `readList` method:

```
static void readList(List<Car> carListIn)
{
     boolean endOfFile = false;
     Car tempCar;

     // use try-with-resources to ensure file is closed safely
     try(
           // create a FileInputStream object, carFile
           FileInputStream carFile = new FileInputStream("Cars.dat");
           // create an ObjectInputStream object to wrap around carFile
           ObjectInputStream carStream = new ObjectInputStream(carFile);
        )
     {

        // read the first (whole) object with the readObject method
        tempCar =  (Car) carStream.readObject();
        while(endOfFile != true)
        {
            try
             {
                carListIn.add(tempCar);
               // read the next (whole) object
               tempCar = (Car) carStream.readObject();
            }

            /* use the fact that readObject throws an EOFException to
               check whether the end of the file has been reached */
```

```
        catch(EOFException  e)
        {
            endOfFile = true;
        }
    }

}
        catch(FileNotFoundException e)
        {
           System.out.println("\nNo file was read");
        }

        catch(ClassNotFoundException e) // thrown by readObject
        {
           System.out.println("\nTrying to read an object of an unknown class");
        }

        catch(StreamCorruptedException e) // thrown by the constructor
        {
           System.out.println("\nUnreadable file format");
        }

        catch(IOException e)
        {
           System.out.println("There was a problem reading the file");
        }
```

Again you can see how easy this is—a whole object is read with the `readObject` method.

We should draw your attention to a few of the exception handling routines we have used here—first notice that we have once again made use of the fact that `readObject` throws an `EOFException` to check for the end of the file. Second, notice that `readObject` also throws a `ClassNotFoundException`, which indicates that the object just read does not correspond to any class known to the program. Finally, the constructor throws a `StreamCorruptedException`, which indicates that the input stream given to it was not produced by an `ObjectOutputStream` object—underlining the fact that reading and writing whole objects are complementary techniques that are specific to Java programs.

One final thing to note—if an attribute of a `Serializable` class is itself an object of another class, then that class too must be `Serializable` in order for us to be able to read and write whole objects as we have just done. You will probably have noticed that in the case of the `Car` class, one of its attributes is a `String`—fortunately the `String` class does indeed implement the `Serializable` interface, which is why we had no problem using it in this way in our example.

Before moving on, it is worth noting that all the Java collection classes such as `HashMap` and `ArrayList` are themselves `Serializable`.

18.9 Random Access Files

All the programs that we have looked at so far in this chapter have made use of serial access. For small applications this will probably be all you need—however, if you were writing applications that handled very large data files it would be desirable to use random access methods. Fortunately Java provides us with this facility.

Table 18.1 Size of the primitive types

byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
boolean	1 bit[3]

The class that we need is called `RandomAccessFile`. This enables us to open a file for random access. Random access files can be opened in either read–write mode or read-only mode; the constructor therefore takes, in addition to the name of the file, an additional `String` parameter which can be either "`rw`" or "`r`", indicating the mode in which the file is to be opened.

In addition to methods similar to those of the `DataOutputStream` class (such as `writeUTF`, `readDouble` and so on), `RandomAccessFile` has a method called `seek`. This takes one attribute, of type `long`, which indicates how many bytes to move the file-pointer before starting a read or write operation.

So now we have the question of how far to move the pointer—we need to be able to calculate the size of each record. If we are dealing only with primitive types, this is an easy matter. These types all take up a fixed amount of storage space, as shown in Table 18.1 overleaf.

The difficulty comes when a record contains `String`s, as is commonly the case. The size of a `String` object varies according to how many characters it contains. What we have to do is to restrict the length of each string to a given amount; let's take the `Car` class as an example. The data elements of any `Car` object consist of two `String`s and a **`double`**. We will make the decision that the two `String` attributes —registration number and make—will be restricted to 10 characters only. Now, any `String` variable will always take up one byte for each character, plus two extra bytes (at the beginning) to hold an integer representing the length of the `String`. So now we can calculate the maximum amount of storage space we need for a car as follows:

registration (String)	12 bytes
make (String)	12 bytes
price (double)	8 bytes
Total	**32 bytes**

This still leaves us with one problem—what if the length of one of the `String` attributes entered is actually *less* than 10? The best way to deal with this is to pad the string out with spaces so that it always contains *exactly* 10 characters. This

[3] Allow for 1 byte when calculating storage space.

means that the size of every `Car` object will always be exactly 32 bytes—you will see how we have done this when you study program below, `RandomFileTester`. This program uses a rather different approach to the one we have used so far in this chapter. Two options (as well as a *Quit* option) are provided. The first, the option to add a car, simply adds the car to the end of the file. The second, to display the details of a car, asks the user for the position of the car in the file then reads this record directly from the file. You can see that there is now no need for a `List` in which to store the cars.

Study the program carefully—then we will discuss it. Note that we have made use of our `EasyScanner` class here.

RandomFileTester

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomFileTester
{
   static final int CAR_SIZE = 32; // each record will be 32 bytes

   public static void main(String[] args)
   {
      char choice;
      do
      {
         System.out.println("\nRandom File Tester");
         System.out.println("1. Add a car");
         System.out.println("2. Display a car");
         System.out.println("3. Quit\n");
         choice = EasyScanner.nextChar();
         System.out.println();
         switch(choice)
         {
            case '1' :addCar();
                      break;
            case '2' :displayCar();
                      break;
            case '3' :break;
            default : System.out.print("\nChoose 1 - 3 only please\n ");
         }
      }while(choice != '3');
   }

    static void addCar()
   {
      String tempReg;
      String tempMake;
      double tempPrice;
      System.out.print("Please enter the registration number: ");
      tempReg = EasyScanner.nextString();

      if(tempReg.length() > 10) //limit the registration number to 10 characters
      {
         System.out.print("Ten characters only - please re-enter: ");
         tempReg = EasyScanner.nextString();
      }
      // pad the string with spaces to make it exactly 10 characters long
      for(int i = tempReg.length() + 1 ; i <= 10 ; i++)
      {
         tempReg = tempReg.concat(" ");
      }

      // get the make of the car from the user
      System.out.print("Please enter the make: ");
      tempMake = EasyScanner.nextString();

      // limit the make number to 10 characters
      if(tempMake.length() > 10)
      {
         System.out.print("Ten characters only - please re-enter: ");
         tempMake = EasyScanner.nextString();
      }
      // pad the string with spaces to make it exactly 10 characters long
      for(int i = tempMake.length() + 1; i <= 10; i++)
      {
         tempMake = tempMake.concat(" ");
      }
```

```
        // get the price of the car from the user
        System.out.print("Please enter the price: ");
        tempPrice = EasyScanner.nextDouble();

        // write the record to the file
        writeRecord(new Car(tempReg, tempMake, tempPrice));
    }

    static void displayCar()
    {
        int pos;
        // get the position of the item to be read from the user
        System.out.print("Enter the car's position in the list: ");
        pos = EasyScanner.nextInt(); // read the record requested from file
        Car tempCar = readRecord(pos);
        if(tempCar != null)
        {
            System.out.println(tempCar.getRegistration().trim()
            + " "
            + tempCar.getMake().trim()
            + " "
            + tempCar.getPrice());
        }
        else
        {
            System.out.println("Invalid postion") ;
        }
    }

    static void writeRecord(Car tempCar)
    {
        // use try-with-resources to ensure file is closed safely
        try(
              // open a RandomAccessFile in read-write mode
              RandomAccessFile carFile = new RandomAccessFile("Cars.rand", "rw");
          )
       {

          // move the pointer to the end of the file
          carFile.seek(carFile.length());
          // write the three fields of the record to the file
          carFile.writeUTF(tempCar.getRegistration());
          carFile.writeUTF(tempCar.getMake());
          carFile.writeDouble(tempCar.getPrice());
       }
       catch(IOException e)
       {
          System.out.println("There was a problem writing the file");
       }
    }
    static Car readRecord(int pos)
    {
         String tempReg;
         String tempMake;
         double tempPrice;
         Car tempCar = null; // a null value is returned if there is a problem reading the record

         // use try-with-resources to ensure file is closed safely
         try(
               // open a RandomAccessFile in read-only mode
               RandomAccessFile carFile = new RandomAccessFile("Cars.rand","r");
            )
         {

            // move the pointer to the start of the required record
            carFile.seek((pos-1) * CAR_SIZE);
            // read the three fields of the record from the file
            tempReg = carFile.readUTF();
            tempMake = carFile.readUTF();
            tempPrice = carFile.readDouble();
            // use the data just read to create a new Car object
            tempCar = new Car(tempReg, tempMake, tempPrice);
         }
         catch(FileNotFoundException e)
         {
            System.out.println("\nNo file was read");
         }

         catch(IOException e)
         {
            System.out.println("There was a problem reading the file");
         }
         // return the record that was read
         return tempCar;
    }
}
```

You can see that in the `addCar` method we have called `writeRecord` with a `Car` object as a parameter. Let's take a closer look at the `writeRecord` method. First the line to open the file in read-write mode:

```
RandomAccessFile carFile = new RandomAccessFile("Cars.rand", "rw");
```

Now the instruction to move the file pointer:

```
carFile.seek(carFile.length());
```

You can see how we use the `seek` method to move the pointer a specific number of bytes; here the correct number of bytes is the size of the file (as we want to write the new record at the end of the file), so we use the `length` method of `RandomAccessFile` to determine this number.

Now we can move on to look at the `readRecord` method. You can see that this is called from within the `displayCar` method, with an integer parameter, representing the position of the required record in the file.

The file is opened in read-only mode:

```
RandomAccessFile carFile = new RandomAccessFile("Cars.rand","r");
```

Then the `seek` method of `RandomAccessFile` is invoked as follows:

```
carFile.seek((pos-1) * CAR_SIZE);
```

You can see that the number of bytes through which to move the pointer has been calculated by multiplying the size of the record by one less than the position. This is because in order to read the first record we don't move the pointer at all; in order to read the second record we must move it 1×32 bytes; for the third record 2×32 bytes; and so on.

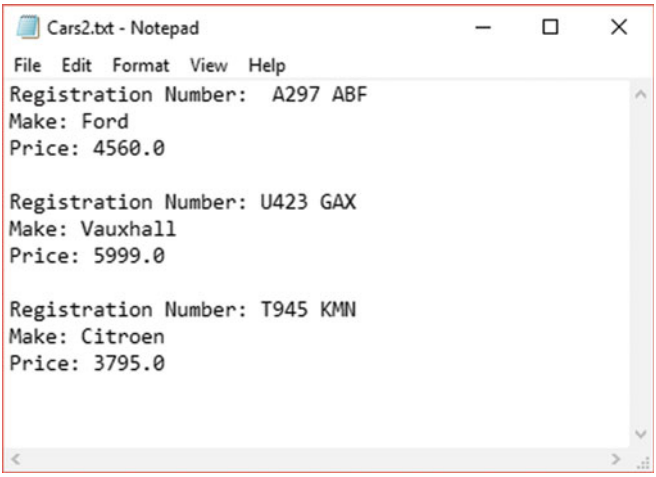
The final thing to note about the program is that in the `displayCar` method we have used the `trim` method of `String` to get rid of the extra spaces that we used to pad out the first two fields of the record.

Here is a test run from the program (starting off with an empty file):

```
Random File Tester
1. Add a car
2. View a car
3. Quit

1

Please enter the registration number: R54 HJK
Please enter the make: Vauxhall
Please enter the price: 7000

Random File Tester
1. Add a car
2. View a car
3. Quit

1

Please enter the registration number: T87 EFU
Please enter the make: Nissan
Please enter the price: 9000

Random File Tester
1. Add a car
2. View a car
3. Quit

2

Enter the car's position in the list: 2
T87 EFU Nissan 9000.0
```

18.10 Self-test Questions

1. Explain the principles of *input* and *output* and identify different input and output devices.
2. What is meant by the term *input/output stream?*
3. Distinguish between *text*, *binary* and *object encoding* of data.
4. Explain why we have used the *try-with-resources* construct throughout this chapter to create and open files.
5. The `TextFileTester` of Sect. 18.5 is to be adapted so that the user is simply asked to enter a number of cars, which, when that process is finished, saves those cars to a text file. The program then terminates. The file does not have to be read from within the program, but should be able to be read by a text editor such as Windows™ Notepad. The format should be as follows:



Adapt the `writeList` method accordingly.

Hint: remember that a blank line is obtained by calling `println` *with no parameters.*

6. What is the difference between *serial access* files and *random access* files?

7. Explain the purpose of the `Serializable` interface.

8. Calculate the number of bytes required to store an object of a class, the attributes of which are declared as follows:

```
private int x;
private char c;
private String s;
```

You can assume that the `String` attribute will always consists of exactly 20 characters.

18.11 Programming Exercises

You will need to have access to the `Car` *class, the source code for which is available on the website. Or you can simply copy it from this chapter. The source code for the programs in the chapter is also on the website.*

1. Run the `TextFileTester` from Sect. 18.5 then adapt it so that it handles binary files, as described in Sect. 18.6.

2. Adapt the `TextFileTester` so that it behaves in the way described in question 5 of the self-test questions.

3. Implement the `CharacterByCharacter` program from Sect. 18.7, using a text file that you have created.

4. Adapt the `TextFileTester` so that it uses object encoding, as explained in Sect. 18.8 (don't forget that the `Car` class must implement the `Serializable` interface).

5. Adapt the `Bank` application of Chap. 8 so that it keeps permanent records using text encoding.

6. In Chap. 12 the case study made use of a file called `TenantFileHandler` that we wrote for you. This is available on the website. Study this class carefully, so that you are sure you understand it, then modify it so that it uses:

   (a) text encoding;

   (b) object encoding.

7. Adapt the `Library` application of Chap. 15 so that it keeps permanent records using object encoding.