JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris August 2001

18 TEXT FILES

18.1 LEARNING OUTCOMES

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

- explain the roles of *FileWriter* and *PrintWriter* objects
- explain the role of *FileReader* and *BufferedReader* objects
- explain the concepts of field, record and file
- create a line sequential text file
- retrieve the contents of a line sequential text file
- find a given record in a text file

18.2 PRE-REQUISITES

The student should be comfortable with using *StringTokenizer* methods and should appreciate that *Integer.parseInt(String)* converts a string to an *int*.

18.3 INTRODUCTION

Data stored in a computer's memory is temporary - it is lost when the Java program stops running or when the computer is switched off. We need a structure that stores data on a more persistent, permanent basis. We need to store data in files on disk.

In a text file, data is stored sequentially, one line after another. This sequential order is maintained - data is retrieved from the file in the same order it was written.

We see how to create a file in text format where each field is separated from the next by a comma and each record in on a new line. We see how to retrieve records from a text file and how to add new records onto the end of a file.

18.4 FIELDS, RECORDS AND FILES

Part of a file of data on employees might look something like this

```
bond,007,50000
Q,001,80000
moneypenny,009,85000
```

Each line represents a *record* - the data for just one employee.

```
Q,001,80000
```


a record

Each record contains *fields*. We have the name field.

```
Q,001,80000
```

name field

We have the employee number field.

```
Q,001,80000
```

number field

We have the salary field.

```
Q,001,80000
```

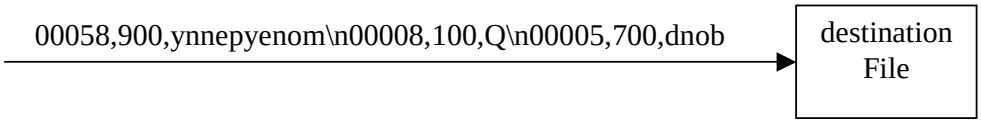
salary field

Notice that a comma is used to separate one field from the next.

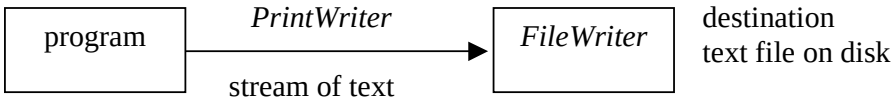
In computing, a file contains records. A file may contain many records, one for each employee. An employee does not have a file. But an employee does have a record in a file.

18.5 STREAMS

We can think of a stream as just a sequence of characters. A stream can have a destination such as a printer or a file on disk.



A *PrintWriter* object represents a stream of text coming out of a program and heading for a text file on disk. A *FileWriter* object represents the text file on disk that is the destination for a stream of text.

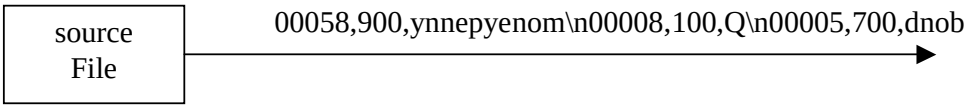


We write text into an output stream using the *println()* method.

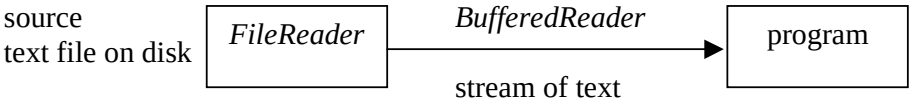
```
PrintWriter outStream;
...
outStream.println("bond,007,50000");
```

println() works in much the same way as it does in *System.out.println()*.

A stream can have a source such as a keyboard or a file on disk.



A *BufferedReader* represents a stream of text coming from a text file on disk and *in* to a program. A *FileReader* represents the text file that is the source of the stream of text.



We read text from an input stream with the *readLine()* method.

```
BufferedReader inStream;
...
String line = inStream.readLine();
```

readLine() returns a line of text from an input stream, or *null* if it fails. A line ends with the carriage-return/newline combination *"\r\n"*.

To summarise, for text files we have

	output		input	
Class	*PrintWriter*	*FileWriter*	*BufferedReader*	*FileReader*
Object	output stream	destination file	input stream	source file

18.6 THE EMPLOYEE CLASS

An employee has a name, a number and a salary. An employee has a constructor and a *toString()* method.

```
public class Employee {
  private String name;
  private String number;
  private int salary;

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


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

In addition an employee has two methods, one to write itself into a text stream and one to read itself from a text stream.

The *write(PrintWriter)* method shown below will write this employee's fields into an output text stream.

```
  public void write(PrintWriter outStream) throws IOException
  {
    outStream.println(name + "," + number + "," + salary);
  }
```

The phrase *throws IOException* says that an input-output error, *such as disk drive not ready* or *file is read only* could occur. At this point, *Employee* does not know (nor does it care) where the output stream is going to. We rely on the users of the *Employee* class to catch and deal with input-output errors.

We use a *BufferedReader* for reading text from a stream. A *BufferedReader* has a method, *readLine()*, which retrieves the next line of text from a stream whose source could be a text file on disk. With repeated calls to *readLine()* you eventually reach the end of the stream. *readLine()* returns *null* if the end of the stream is reached and there are no more lines of text to retrieve.

```
BufferedReader inStream;
...
String line = inStream.readLine();
if (line == null)
  ...
```

Having got a line of text we have to break it up into its component fields. We use a *StringTokenizer* to do this. Recall that when a record was written to the output stream, a comma was used to separate one field from the next. So we specify a comma to be the token separator.

```
StringTokenizer st = new StringTokenizer(line, ",");
```

Before we update this employee's private fields we have to ensure that there are enough tokens in the line just retrieved from the stream.

```
if (st.countTokens() == 3) {  // one token for each field
  name = st.nextToken();
  number = st.nextToken();
  String salaryString = st.nextToken();
```

Now, *nextToken()* always returns a string. So if its an *int* field that we need to put a value in to, we must convert the string to an *int*. We use the *Integer* wrapper class method *parseInt()* to do this.

```
salary = Integer.parseInt(salaryString);  // string to int
```

Here is the entire method.

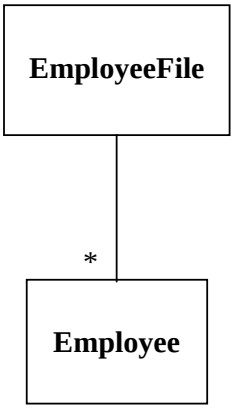
```
public int read(BufferedReader inStream) throws IOException
{
  String line = inStream.readLine();
  if (line == null)
    return -1;               // end of stream encountered

  StringTokenizer st = new StringTokenizer(line, ",");

  if (st.countTokens() == 3) {   // one token for each field
    name = st.nextToken();
    number = st.nextToken();
    String salaryString = st.nextToken();
    salary = Integer.parseInt(salaryString);  // to int
  }
  return line.length();
}
```

The entire class is shown below.

```
/* Employee.java
   Terry Marris  7 August 2001
*/

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

public class Employee {
  private String name;
  private String number;
  private int salary;

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


  public void write(PrintWriter outStream) throws IOException
  {
    outStream.println(name + "," + number + "," + salary);
  }


  public int read(BufferedReader inStream) throws IOException
  {
    String line = inStream.readLine();
    if (line == null)
      return -1;                  // end of stream encountered

    StringTokenizer st = new StringTokenizer(line, ",");

    if (st.countTokens() == 3) {   // one token for each field
      name = st.nextToken();
      number = st.nextToken();
      String salaryString = st.nextToken();
      salary = Integer.parseInt(salaryString);  // to int
    }
    return line.length();
  }


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

18.7 THE EMPLOYEE FILE CLASS

An *EmployeeFile* has a collection of *Employee* objects.



An *EmployeeFile* has a name on disk. We use the convention that data files have the suffix *.data*.

```
private String fileName = "Employee.data";
```

The constructor creates a new empty file, but only if it does not already exist.

```
public EmployeeFile() throws IOException
{
  File file = new File(fileName);
  if (!file.exists()) {
```

In the *Employee* class we used a *PrintWriter* to represent the stream of characters that we are sending to the file on disk. Now we use a *FileWriter* to represent this destination. We associate the text file with its file name.

```
FileWriter destFile = new FileWriter(fileName);
```

We associate the stream with its destination.

```
PrintWriter outStream = new PrintWriter(destFile);
```

Finally, we break the connection between output stream and its destination with the *close()* message.

```
outStream.close();
```

We *try* to execute some Java file-handling code, and *catch* and deal with any errors that might occur.

```
try {
  File-handling code goes here
}
catch(IOException e) {
  Error-handling code goes here
}
```

For example

```
try {
  File file = new File(fileName);
  if (!file.exists()) {
    FileWriter destFile = new FileWriter(fileName);
    PrintWriter outStream = new PrintWriter(destFile);
    outStream.close();
  }
}
catch (IOException e) {
  System.out.println(e);
  System.exit(1);
}
```

The Java run-time system will place details of any file handling errors in the *IOException* object, *e*. We print the error messages on the console with *System.out.println(e)*. Then we choose to halt program execution with *System.exit(1)*. With a *try...catch....* combination we programmers remain in control.

The entire constructor is shown below.

```
public EmployeeFile()
/* Creates an empty file if it does not already exist.
*/
{
  try {
    File file = new File(fileName);
    if (!file.exists()) {
      FileWriter destFile = new FileWriter(fileName);
      PrintWriter outStream = new PrintWriter(destFile);
      outStream.close();
    }
  }
  catch (IOException e) {
    System.out.println(e);
    System.exit(1);
  }
}
```

The *add(String, String, int)* method creates a new *Employee* object from the given parameters.

```
public String add(String name, String number, int salary)
{
  Employee employee = new Employee(name, number, salary);
```

We want to add the new employee onto the end of the file on disk. We assume that the file with the given file name exists.

```
  boolean append = true;
  FileWriter destFile = new FileWriter(fileName, append);
```

We associate an output stream with the destination file.

```
  PrintWriter outStream = new PrintWriter(destFile);
```

And then send the message to write the employee into the output stream.

```
  employee.write(outStream);
```

Finally, we close the output stream and return *success.*

```
  outStream.close();
  return "success";
```

As before, we wrap the file handling code in a *try...catch* combination.

```
  try {
    boolean append = true;
    FileWriter destFile = new FileWriter(fileName, append);
    PrintWriter outStream = new PrintWriter(destFile);
    employee.write(outStream);
    outStream.close();
  }
  catch (IOException e) {
    System.out.println(e);
    System.exit(1);
  }
  return "success";
```

The entire *add()* method is shown below.

```
public String add(String name, String number, int salary)
/* Creates a new Employee instance from the given fields,
   adds it onto the end of the file.
*/
{
  Employee employee = new Employee(name, number, salary);

  try {
    boolean append = true;
    FileWriter destFile = new FileWriter(fileName, append);
    PrintWriter outStream = new PrintWriter(destFile);
    employee.write(outStream);
    outStream.close();
  }
  catch (IOException e) {
    System.out.println(e);
    System.exit(1);
  }
  return "success";
}
```

Before and after every change we make to a file, we print out its contents. This assures us that our methods work as we want them to. We need a method that enables us to view the contents of a file. *toString()* returns the contents of the file as a string with one record per line.

We create an employee with no name, no number and no salary.

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

We create an empty string that will eventually contain the entire contents of the file.

```
String string = "";
```

We associate the source file with a file name on disk.

```
FileReader sourceFile = new FileReader(fileName);
```

We associate the source file with its stream.

```
BufferedReader inStream = new BufferedReader(sourceFile);
```

We loop for as long as there is another line of text to be retrieved from the input stream.

```
while (employee.read(inStream) >= 0) {
```

Remember that *employee.read()* places values into this employee's fields (§18.6).

Having retrieved the next employee from the input stream we add it to the growing string and append the new line character (so we get one employee per line).

```
string += employee + "\n";
```

As before, we enclose the file handling operations in a *try...catch...* combination. Here is the entire method.

```
public String toString()
/* Returns the contents of the file as a string with
   one record per line.
*/
{
  Employee employee = new Employee("", "", 0);
  String string = "";
  try {
    FileReader sourceFile = new FileReader(fileName);
    BufferedReader inStream = new
                              BufferedReader(sourceFile);
    while (employee.read(inStream) >= 0) {
      string += employee + "\n";
    }
    inStream.close();
  }
  catch (IOException e) {
    System.out.println(e);
    System.exit(1);
  }
  return string;
}
```

The entire class, and a small test method, are shown below.

```
/* EmployeeFile.java
   Terry Marris 7 August 2001
*/

import java.io.*;


public class EmployeeFile {
  private String fileName = "Employee.data";

  public EmployeeFile()
  /* Creates an empty file if it does not already exist.
  */
  {
    try {
      File file = new File(fileName);
      if (!file.exists()) {
        FileWriter destFile = new FileWriter(fileName);
        PrintWriter outStream = new PrintWriter(destFile);
        outStream.close();
      }
    }
    catch (IOException e) {
      System.out.println(e);
      System.exit(1);
    }
  }


  public String add(String name, String number, int salary)
  /* Creates a new Employee instance from the given fields,
     adds it onto the end of the file.
  */
  {
    Employee employee = new Employee(name, number, salary);

    try {
      boolean append = true;
      FileWriter destFile = new FileWriter(fileName, append);
      PrintWriter outStream = new PrintWriter(destFile);
      employee.write(outStream);
      outStream.close();
    }
    catch (IOException e) {
      System.out.println(e);
      System.exit(1);
    }
    return "success";
  }
```

```
  public String toString()
  /* Returns the contents of the file as a string with
     one record per line.
  */
  {
    Employee employee = new Employee("", "", 0);
    String string = "";
    try {
      FileReader sourceFile = new FileReader(fileName);
      BufferedReader inStream = new
                                BufferedReader(sourceFile);
      while (employee.read(inStream) >= 0) {
        string += employee + "\n";
      }
      inStream.close();
    }
    catch (IOException e) {
      System.out.println(e);
      System.exit(1);
    }
    return string;
  }


  public static void main(String[] s)
  {
    EmployeeFile employeeFile = new EmployeeFile();

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

    System.out.println(employeeFile);
  }
}
```

Output

```
bond, 007, £50000
Q, 001, £75000
moneypenny, 002, £65000
```

18.8 REVIEW

18.9 FURTHER READING

HORSTMANN & CORNELL Core Java 2 Volume 1 pp 638, 652

In the next section we look at random access files.

18.10 EXERCISES

1 Explain the meaning of the terms field, record and file and how they are inter-related. Illustrate your explanation with examples.

2 Explain the role of *FileWriter, PrintWriter, BufferedReader* and *FileReader* classes.

3 Try out the *EmployeeFile* class described in §18.7 above. If you do not want the same data to be repeatedly added to your file, remember to delete the data file (use the dos delete command) before each program run.

4 In the *toString()* method of the *EmployeeFile* class, change the statement

```
FileReader sourceFile = new FileReader(fileName);
```

to

```
FileReader sourceFile = new FileReader("OldEmployee.data");
```

Compile, run and report what happens. Explain the program output.

5 Implement and test a method *boolean containsEmployeeWithName(String aName)* that returns *true* if an employee with the given name is found in the *Employee.data* file, *false* otherwise. Remember to check that your method successfully finds the *first* and the *last* records in the file; the file should contain at least five records.

6 Implement and test a method, *boolean employee(String number)* that returns the employee with the given number if found in the *Employee.data* file, and returns *null* if no such employee is found in the file.