CHAPTER 7

File Handling and Applications

After studying this chapter, you will be able to:

- Understand computer files and perform file operations
- Work with sequential files and control break logic

In this chapter, you learn how to open and close files in Java, how to use Java to read data from and write data to a file in a program, and how to work with sequential files in a Java program. You should do the exercises and labs in this chapter after you have finished Chapter 7 in *Programming Logic and Design, Eighth Edition*.

File Handling

Business applications are often required to manipulate large amounts of data that are stored in one or more files. As you learned in Chapter 7 of *Programming Logic and Design,* data is organized in a hierarchy. At the lowest level of the hierarchy is a **field**, which is a group of characters. On the next level up is a **record**, which is a group of related fields. For example, you could write a program that processes employee records, with each employee record consisting of three fields: the employee's first name, the employee's last name, and the employee's department number.

In Java, to use the data stored in a file, the program must first open the file and then read the data from the file. You use prewritten classes that are part of the Java Standard Edition Development Kit (JDK) to accomplish this. In the next section, you learn how to import packages and classes to make the `BufferedReader` and `FileReader` classes available in your programs. You will also learn how to use these classes to open a file, close a file, read data from a file, and write data to a file.

Importing Packages and Classes

A **package** is a group of related classes. The classes that you need in this chapter are part of a package named `java.io`. The JDK contains many classes that are prewritten for you by the Java development team. You can simplify your programming tasks by creating objects using these classes. You can then use the attributes and methods of those objects in your Java programs.

To use these prewritten classes, you must import them into your Java program. You use the `import` keyword to include a class from a Java package. The following code imports the `BufferedReader` class from the Java package named `java.io`.

```
import java.io.BufferedReader;
```

You can also use the * (asterisk) character in an import statement to import all classes from a package rather than specifying a single class. The following code imports all of the classes in the `java.io` package.

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

The Java programs in this chapter will use this style to import the classes needed to perform file input and output.

The `import` statement tells the Java compiler the name of the package and the name of the class (or classes) that contains the prewritten code you want to use. The Java compiler will automatically include this code.

Opening a File for Reading

To open a file and read data into a Java program, you instantiate a `FileReader` object and specify the name of the file to associate with the object. Look at the following example:

```
FileReader fr = new FileReader("inputFile.txt");
```

In the example, the `new` keyword instantiates a `FileReader` object. This new object is associated with the file named `inputFile.txt`. Notice that the name of the file is enclosed in double quotes and placed within parentheses. As a result of the assignment statement, this newly created `FileReader` object is assigned to a variable named `fr` and may now be referred to in your Java program using the name `fr`. In addition, the statement opens the file named `inputFile.txt` for reading. This means that the program can now read data from the file. In this example, the file named `inputFile.txt` must be saved in the same folder as the Java program that is using the file. To open a file that is saved in a different folder, a path must be specified as in the next example:

```
FileReader fr = new FileReader(
   "C:\myJavaPrograms\Chapter7\inputFile.txt");
```

Even though the program can now read from the file, it is usually more efficient to read from a buffered file. To do this, you need to create a `BufferedReader` object. A `FileReader` object reads data from a file one character at a time, whereas a `BufferedReader` object can read data a line at a time. To create a `BufferedReader` object, we decorate the `FileReader` object. **Decorating** is a way of adding functionality to objects in Java. Here is an example:

```
BufferedReader br = new BufferedReader(fr);
```

In this example, a new `BufferedReader` object is created by adding functionality to the `FileReader` object named `fr`. The name of the `BufferedReader` variable is `br`. You will use the name `br` to refer to the `BufferedReader` object in your Java program.

Reading Data from an Input File

Once you have created a new `BufferedReader` object that decorates a `FileReader` object, you are ready to read the data in the file. The `BufferedReader` class provides this functionality with the `readLine()` method. The `readLine()` method allows the program to read a line from an input file. A **line** is defined as all of the characters up to a **newline** character or up to the **End Of File (EOF)** marker. The newline character is generated when you press the Enter key on the keyboard. The EOF marker is automatically placed at the end of a file when it is saved.

Assume that the input file for a program is organized so an employee's first name is on one line, followed by his last name on the next line, followed by his salary on the third, as follows:

Tim

Moriarty

4000.00

To allow the program to read this data, you would write the following Java code:

```
String firstName, lastName, salaryString;
double salary;
firstName = br.readLine();
lastName = br.readLine();
salaryString = br.readLine();
```

Because the `readLine()` method always returns a `String`, the first line in the example declares three `String` variables named `firstName`, `lastName`, and `salaryString`. The next line declares a `double` named `salary`. Next, the `readLine()` method is used three times to read the three lines of input from the file associated with the `BufferedReader` object named `br`. After this code executes, the variable named `firstName` contains the value *Tim*, the variable named `lastName` contains the value *Moriarty*, and the variable named `salaryString` contains the value *4000.00*. As you have previously learned, if your program requires the use of an employee's salary in a numeric calculation, you must convert `salaryString` to a `double` as follows:

```
salary = Double.parseDouble(salaryString);
```

The next example illustrates how to read a salary and convert it to a `double` in one step. This technique allows you to omit declaring the `salaryString` variable.

```
salary = Double.parseDouble(br.readLine());
```

Reading Data Using a Loop and EOF

In a program that has to read large amounts of data, it is usually best to have the program use a loop. In the loop, the program continues to read from the file until EOF (end of file) is encountered. The `readLine()` method returns a `null` value when EOF is reached. The Java code that follows shows how to use the `readLine()` method as part of a loop.

```
while((firstName = br.readLine()) != null)
{
     // body of loop
}
```

In this example, the `readLine()` method is part of the expression to be tested. As long as the value returned by `readLine()` is not equal to `null`, the expression is `true`, and the loop is entered. As soon as EOF is encountered, the test becomes `false`, and the program exits the loop. The parentheses are used to control precedence.

Opening a File for Writing

To write data from a Java program to an output file, the program must first open a file. This is a two-step process: first, the program must instantiate a `FileWriter` object and then specify the name of the file to associate with the object. Look at the following example:

```
FileWriter fw = new FileWriter("outputFile.txt");
```

In this example, the `new` keyword is used to instantiate a `FileWriter` object. This object is associated with the file named `outputFile.txt`. Notice that the name of the file is enclosed in double quotes and placed within parentheses. As a result of the assignment statement, this newly created `FileWriter` object is assigned to a variable named `fw`. You can now refer to the object in your Java program using the name `fw`. In addition, the statement opens the file named `outputFile.txt` for writing. This means that the program can now write data to the file.

As with input files, it is a good idea to decorate the `FileWriter` object to add functionality. For example, you can add the functionality that is included in the `PrintWriter` class, which provides the ability to flush (that is, empty) and close an output file. In Java, a write operation is not complete until the buffer associated with an output file is **flushed** (emptied) and **closed** (made unavailable for further output). The following example shows how to decorate the `FileWriter` object by adding functionality from the `PrintWriter` class.

```
PrintWriter pw = new PrintWriter(fw);
```

In this example, a new `PrintWriter` object is created by adding functionality to the `FileWriter` object named `fw`. The name of the `PrintWriter` object is `pw`. From this point on, you can use the name `pw` to refer to the `PrintWriter` object in your Java program.

Writing Data to an Output File

Once you have decorated a `FileWriter` object with a `PrintWriter` object, the program is ready to write data to a file. You can use the `println()` method (which is included in the `PrintWriter` class) to write a line to an output file.

As an example, assume that an employee's `firstName`, `lastName`, and `salary` have been read from an input file as in the previous example and that the employee is to receive a 15 percent salary increase that is calculated as follows:

```
final double INCREASE = 1.15;
double newSalary;
newSalary = salary * INCREASE;
```

You now want to write the employee's `lastName`, `firstName`, and `newSalary` to the output file named `newSalary2015.txt`. The code that follows accomplishes this task.

```
FileWriter fw = new FileWriter("newSalary2015.txt");
PrintWriter pw = new PrintWriter(fw);
pw.println(lastName);
pw.println(firstName);
pw.println(newSalary);
pw.flush();
pw.close();
```

The Java program shown in Figure 7-1 implements the file input and output operations discussed in this section.

```
// EmployeeRaise.java - This program reads employee first
// and last names and salaries from an input file,
// calculates a 15% raise, and writes the employee's first
// and last name and new salary to an output file.
// Input:  employees.txt.
// Output: newSalary2015.txt

import java.io.*;  // Import class for file input.

public class EmployeeRaise
{
   public static void main(String args[]) throws Exception
   {

      String firstName, lastName, salaryString;
      double salary, newSalary;
      final double INCREASE = 1.15;

      // Open input file.
      FileReader fr = new FileReader("employees.txt");
      // Create BufferedReader object.
      BufferedReader br = new BufferedReader(fr);

      // Open output file
      FileWriter fw = new FileWriter("newSalary2015.txt");
      PrintWriter pw = new PrintWriter(fw);

      // Read records from file and test for EOF.
      while((firstName = br.readLine()) != null)
      {
         lastName = br.readLine();
         salaryString = br.readLine();
         salary = Double.parseDouble(salaryString);
         newSalary = salary * INCREASE;
         pw.println(lastName);
         pw.println(firstName);
         pw.println(newSalary);
         pw.flush();
      }

      br.close();
      pw.close();
      System.exit(0);
   } // End of main() method.
} // End of EmployeeRaise class.
```

Figure 7-1 Reading and writing file data

When writing code that opens files and writes to files, you need to be aware of potential problems. For example, the program might try to open a nonexistent file or it might try to read beyond the EOF marker. If these events occur, a Java program will generate an

exception. (An **exception** is an event that occurs that disrupts the normal flow of execution.) The Java compiler knows that certain methods are capable of causing an exception. If these methods are used in a program, it will fail to compile unless you include the words `throws Exception` as part of the header for the `main()` method, as shown in Figure 7-1. By including these words, you are telling the Java compiler that you know an exception could occur, and the compiler should assume that the program contains code that will handle the exception, should it occur. A section of code that is designed to solve problems related to exceptions is known as an **exception handler**.

There is much more to learn about the input and output classes in the `java.io` package, but you will be able to accomplish quite a lot using what you have learned in this section.

You need to know quite a bit about Java in order to write exception handlers. In this book, we will simply include `throws Exception` in headers to ensure that our programs compile.

Exercise 7-1: Opening Files and Performing File Input

In this exercise, you use what you have learned about opening a file and getting input into a program from a file. Study the following code, and then answer Questions 1–3.

```
1   FileReader fr = new FileReader(myDVDFile.dat);
2   BufferedReader br = new BufferedReader();
3   String dvdName, dvdPrice, dvdShelf;
4   dvdName = br.readLine();
5   dvdPrice = br.readLine();
6   dvdShelf = br.readLine();
```

Figure 7-2 Code for Exercise 7-1

1. Describe the error on line 1, and explain how to fix it.

   _____

2. Describe the error on line 2, and explain how to fix it.

   _____

3. Consider the following data from the input file `myDVDFile.dat`:

   ```
   Fargo 8.00 1A
   Amadeus 20.00 2C
   Casino 7.50 3B
   ```

a. What value is stored in the variable named `dvdName`?

b. What value is stored in the variable name `dvdPrice`?

c. What value is stored in the variable named `dvdShelf`?

d. If there is a problem with the values of these variables, what is the problem and how could you fix it?

Lab 7-1: Opening Files and Performing File Input

In this lab, you will open a file, `flowers.dat`, and read input from that file in a prewritten Java program. The program should read and print the names of flowers and whether they are grown in shade or sun.

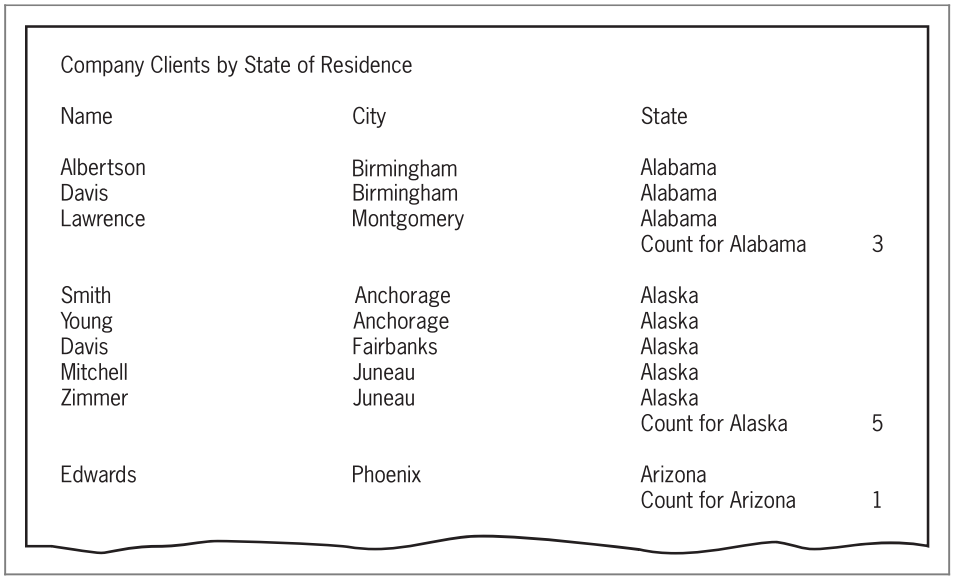
1. Open the source code file named `Flowers.java` using Notepad or the text editor of your choice.
2. Declare the variables you will need.
3. Write the Java statements that will open the input file, `flowers.dat`, for reading.
4. Write a `while` loop to read the input until EOF is reached.
5. In the body of the loop, print the name of each flower and where it can be grown (sun or shade).
6. Save this source code file in a directory of your choice, and then make that directory your working directory.
7. Compile the source code file `Flowers.java`.
8. Execute the program.

Understanding Sequential Files and Control Break Logic

As you learned in Chapter 7 of *Programming Logic and Design,* a **sequential file** is a file in which records are stored one after another in some order. The records in a sequential file are organized based on the contents of one or more fields, such as ID numbers, part numbers, or last names.

A **single-level control break** program reads data from a sequential file and causes a break in the logic based on the value of a single variable. In Chapter 7 of *Programming Logic and Design,* you learned about techniques you can employ to implement a single-level control break program. Be sure you understand these techniques before you continue on with this chapter. The program described in Chapter 7 of *Programming Logic and Design* that produces a report of customers by state is an example of a single-level control break program. This

program reads a record for each client, keeps a count of the number of clients in each state, and prints a report. As shown in Figure 7-3, the report generated by this program includes clients' names, cities, and states, along with a count of the number of clients in each state.



Company Clients by State of Residence

Name	City	State	
Albertson	Birmingham	Alabama	
Davis	Birmingham	Alabama	
Lawrence	Montgomery	Alabama	
		Count for Alabama	3
Smith	Anchorage	Alaska	
Young	Anchorage	Alaska	
Davis	Fairbanks	Alaska	
Mitchell	Juneau	Alaska	
Zimmer	Juneau	Alaska	
		Count for Alaska	5
Edwards	Phoenix	Arizona	
		Count for Arizona	1

Figure 7-3 Control break report with totals after each state

Each client record is made up of the following fields: `Name`, `City`, and `State`. Note the following example records, each made up of three lines:

```
Albertson
Birmingham
Alabama
Lawrence
Montgomery
Alabama
Smith
Anchorage
Alaska
```

Remember that input records for a control break program are usually stored in a data file on a storage device, such as a disk, and the records are sorted according to a predetermined control break variable. For example, the control break variable for this program is `state`, so the input records would be sorted according to `state`.

Figure 7-4 includes the pseudocode for the Client By State program, and Figure 7-5 shows the Java code that implements the program.

As you can see in Figure 7-5, the Java program begins on line 1 with comments that describe what the program does. (The line numbers shown in this program are not part of the Java

code. They are included for reference only.) The program also includes comments that describe the program's input and output. Next comes the Java code that defines the `ClientByState` class (line 9) and, within the class, the `main()` method (line 11).


```
Start
   Declarations
      InputFile inFile
      string TITLE = "Company Clients by State of Residence"
      string COL_HEADS = "Name   City   State"
      string name
      string city
      string state
      num count = 0
      String oldState
   getReady()
   while not eof
      produceReport()
   endwhile
   finishUp()
stop

getReady()
   output TITLE
   output COL_HEADS
   open inFile "ClientsByState.dat"
   input name, city, state from inFile
   oldState = state
return
produceReport()
   if state <> oldState then
      controlBreak()
   endif
   output name, city, state
   count = count + 1
   input name, city, state from inFile
return

controlBreak()
   output "Count for ", oldState, count
   count = 0
   oldState = state
return

finishUp()
   output "Count for ", oldState, count
   close inFile
return
```

Figure 7-4 Client By State program pseudocode

```
 1 // ClientByState.java - This program creates a report that
 2 // lists clients with a count of the number of clients for
 3 // each state.
 4 // Input:  client.dat
 5 // Output:  Report
 6
 7 import java.io.*;
 8
 9 public class ClientByState
10 {
11    public static void main(String args[]) throws Exception
12    {
13       // Declarations
14       FileReader fr = new FileReader("client.dat");
15       BufferedReader br = new BufferedReader(fr);
16       final String TITLE =
17             "\n\nCompany Clients by State of Residence\n\n";
18       String name = "", city = "", state = "";
19       int count = 0;
20       String oldState = "";
21       boolean done;
22
23       // Work done in the getReady() method
24       System.out.println(TITLE);
25       if((name = br.readLine()) != null)
26       {
27          city = br.readLine();
28          state = br.readLine();
29          done = false;
30          oldState = state;
31       }
32       else
33          done = true;
34       while(done == false)
35       {
36          // Work done in the produceReport() method
37          if(state.compareTo(oldState) != 0)
38          {
39             // Work done in the controlBreak() method
40             System.out.println("\t\t\tCount for " +
41                                oldState + " " + count);
42             count = 0;
43             oldState = state;
44          }
45          System.out.println(name + " " + city + " " +
46                             state);
47          count++;
```

Figure 7-5 Client By State program written in Java *(continues)*

(continued)

```
48          if((name = br.readLine()) != null)
49          {
50             city = br.readLine();
51             state = br.readLine();
52             done = false;
53          }
54          else
55             done = true;
56       }
57       // Work done in the finishUp() method
58       System.out.println("\t\t\tCount for " +
59                           oldState + " " + count);
60       br.close();
61       System.exit(0);
62
63    } // End of main() method
64 } // End of ClientByState class
```

Figure 7-5 Client By State program written in Java

Within the `main()` method, lines 14 through 21 declare variables and constants and initialize them when appropriate. Lines 14 and 15 declare variables as well as open the input file named `client.dat`. Notice that one of the declarations shown in the pseudocode, string `COL_HEADS = "Name City State"` is not included in the Java code. Since you have not yet learned about the Java statements needed to line up values in report format, the Java program shown in Figure 7-5 prints information on separate lines rather than in the column format used in the pseudocode.

Lines 24 through 33 include the work done in the `getReady()` method, which includes printing the heading for the report this program generates and performing a priming read. You learned about performing a priming read in Chapter 3 of this book and in Chapter 3 of *Programming Logic and Design*.

Notice that the Java code in the priming read (lines 25 through 28) is a little different than the pseudocode. An `if` statement is used on line 25 to test if a client's name was read from the input file or if EOF was encountered. If EOF is not encountered, the result of this test will be `true`, causing the execution of the input statements that read the `city` and `state` from the input file. The `boolean` value `false` is also assigned to the variable named `done` on line 29, followed by assigning the current value of `state` to the variable named `oldState` on line 30. Remember that the variable `state` serves as the control break variable. If EOF is encountered, the result of this test will be `false`, causing the `boolean` value `true` to be assigned to the variable named `done` on line 33. The `boolean` variable named `done` is used later in the program to control the `while` loop.

Next comes the `while` loop (line 34), which continues to execute as long as the value of the `boolean` variable `done` is `false`. The body of the `while` loop contains the work done in the

`produceReport()` method. First, an `if` statement uses the `compareTo()` method to test the control break variable `state` on line 37. The `compareTo()` method's job is to determine if the record the program is currently working with has the same state as the previous record's state. If it does not, this indicates the beginning of a new state. As a result, the program performs the work done in the `controlBreak()` method (lines 40 through 43). The work of the `controlBreak()` method does the following:

1. Prints the value of the variable named `count` that contains the count of clients in the current state (lines 40 and 41).
2. Assigns the value 0 to the variable named `count` to prepare for the next state.
3. Assigns the value of the variable named `state` to the variable named `oldState` to prepare for the next state.

If the record the program is currently working with has the same state as the previous record's state, the `controlBreak()` method's work is not performed. Whether or not the current record's state is the same state as the previous record's state, the next statement to execute (lines 45 and 46) prints the client's name, city, and state. Then the variable named `count` is incremented on line 47 followed by the program reading the next client's record on lines 48 through 55 using the same technique as the priming read.

The condition in the `while` loop on line 34 is then tested again, causing the loop to continue executing until the value of the variable named `done` is `true`. The variable named `done` is assigned the value `true` when the program encounters EOF when reading from the input file on line 55.

When the `while` loop is exited, the last section of the program executes. This consists of the work done in the `finishUp()` method:

- Printing the value of the variable named `count` (which is the count of the clients in the last state in the input file) on lines 58 and 59
- Closing the input file (line 60)

Exercise 7-2: Accumulating Totals in Single-Level Control Break Programs

In this exercise, you will use what you have learned about accumulating totals in a single-level control break program. Study the following code, and then answer Questions 1–4.

```
if(sectionNum != oldSectionNum)
{
     System.out.println("Section Number " + oldSectionNum);
     totalSections = sectionNum;
     oldSectionNum = sectionNum;
}
```

1. What is the control break variable?

_____

2. True or False? The value of the control break variable should never be changed.

3. Is `totalSections` being calculated correctly?

   If not, how can you fix the code?

4. True or False? In a control break program, it does not matter if the records in the input file are in a specified order.

Lab 7-2: Accumulating Totals in Single-Level Control Break Programs

In this lab, you will use what you have learned about accumulating totals in a single-level control break program to complete a Java program. The program should produce a report for a supermarket manager to help her keep track of the hours worked by her part-time employees. The report should include the day of the week, the number of hours worked by each employee for each day, and the total hours worked by all employees each day. The report should look similar to the one shown in Figure 7-6.

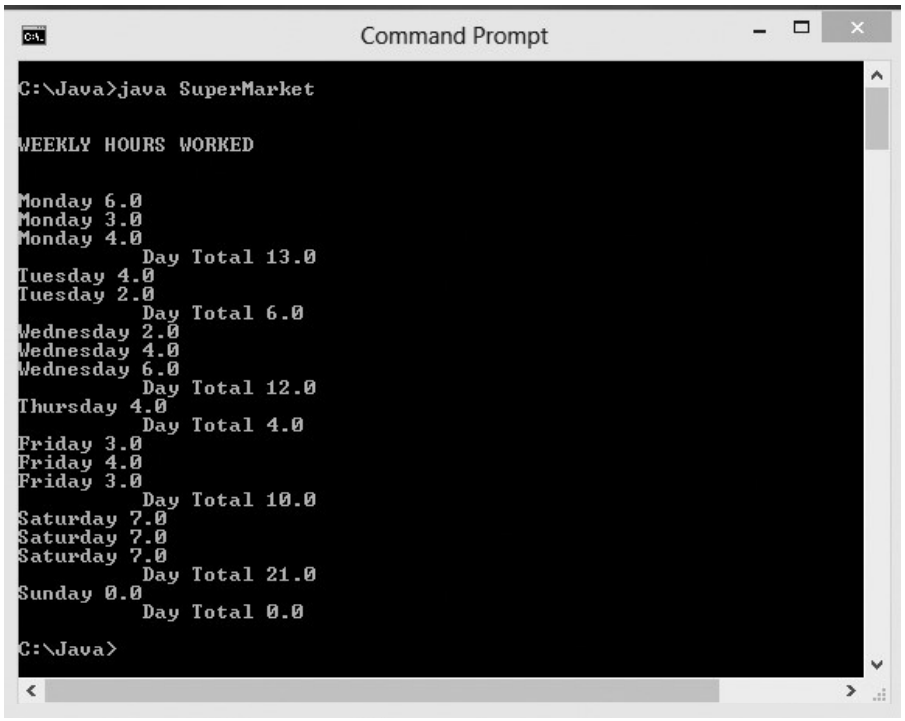


Figure 7-6 SuperMarket program report

The student file provided for this lab includes the necessary variable declarations and input and output statements. You need to implement the code that recognizes when a control break should occur. You also need to complete the control break code. Be sure to accumulate the daily totals for all days in the week. Comments in the code tell you where to write your code. You can use the Client By State program in this chapter as a guide for this new program.

1. Open the source code file named `SuperMarket.java` using Notepad or the text editor of your choice.
2. Study the prewritten code to understand what has already been done.
3. Write the control break code, including the code for the `dayChange()` method, in the `main()` method.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file, `SuperMarket.java`.
6. Execute this program using the following input values:

   `Monday` –**`6`** hours (employee 1), **`3`** hours (employee 2), **`4`** hours (employee 3)

   `Tuesday` – **`4`** hours (employee 1), **`2`** hours (employee 2)

   `Wednesday` – **`2`** hours (employee 1), **`4`** hours (employee 2), **`6`** hours (employee 3)

   `Thursday` – **`4`** hours (employee 1)

   `Friday` – **`3`** hours (employee 1), **`4`** hours (employee 2), **`3`** hours (employee 3)

   `Saturday` – **`7`** hours (employee 1), **`7`** hours (employee 2), **`7`** hours (employee 3)

   `Sunday` – **`0`** hours