Chapter 9: Writing to and Reading from Text Files

Learning Outcomes:

- ✓ Identify the importance of being able to store data into a non-volatile format
- ✓ Implement appropriate classes into a program that enable data to be written to a text file and read from a text file

Prerequisite Knowledge:

- ✓ Complete Chapter 8
- ✓ Be able to confidently implement arrays and array lists in conjunction with iteration (loops)
- ✓ Understand the fundamental principles of instantiating objects from classes

Keywords: FileWrite, PrintWriter, Dispose, BufferedReader, Directory

9.1 The Theory: Writing to a Text File

Executed programs run in the random-access memory (RAM) of a computer; RAM is referred to as 'volatile memory', because a small trickle of electricity is required to preserve the retained information. All the programs written in the previous chapters executed in volatile memory and thus at the end of their execution, none of the information was saved for later use. As programmers, being able to save information is essential; no customer would be happy with the loss of their data! Thankfully, there are many artefacts available for programmers that can be used to store data to the hard disk of a computer (non-volatile memory); such as databases, serialised files and text files, all of which can be manipulated by programming languages. One of the easiest ways to save data is to write it to a text file.

Using FileWriter and PrintWriter

The Java API can be used to achieve almost anything! Thankfully, there are two classes in the Java library that make saving information to a text file a 'doddle'. The `FileWriter` class contains the necessary methods to print simple characters to a text file; the `PrintWriter` class can be used to extend the functionality of the `FileWriter` class, by allowing the use of additional methods, in particular the [`println()`] method that is used to print entire strings to a single line of a text file. Below is an example of how `FileWriter` and `PrintWriter` can be used in a program:

```
import java.io.FileWriter; //import classes
import java.io.PrintWriter;
public class MyTextWriter{

    public static void main(String[] args){
              FileWriter writeObj; //declare variables (uninstantiated)
              PrintWriter printObj;
              try{ //risky behaviour – catch any errors
                 writeObj = new FileWriter("C:\\Documents\\myText.txt" , true);
                 printObj = new PrintWriter(writeObj); //instantiate both objects
                 printObj.println("Hello Text file");  //print to file
                 printObj.println("A good day to be texting!");
                 printObj.close(); //close stream
              }catch(Exception ex){
                 System.out.println("Error: " + ex);
              }

       }
}
```

 © ZigZag Education, 2014

Writing Data to File

The `FileWriter` class requires two arguments [`FileWriter("C:\\Documents\\myText.txt", true);`]; the first is the directory of the text file to write to (if the file does not exist, it will be automatically created). Note that two backslashes are used in the directory to 'escape' the use of the backslash character. The second argument is a boolean value; the value true indicates to append the file (add to it), whereas false indicates to overwrite the current file content. Working with files is considered 'risky behaviour' by the Java compiler, and thus must be safely handled using the [`try/catch`] technique. When the [`printObj`] is finished with, the [`close()`] method is called to close the stream and release any associated system resources. The example below demonstrates how the [`println()`] method, from the `PrintWriter` class, can be passed a string variable, as opposed to a string literal. In theory, the string variable can be loaded with values obtained by a [`readLine()`] method, before printing the variable's content to a text file.

Try the code

```
import java.io.FileWriter; //import classes
import java.io.PrintWriter;
public class MyTextWriter{

    public static void main(String[] args){
             FileWriter writeObj; //declare variables (uninstantiated)
             PrintWriter printObj;
             String myText = "Hello Text file";
             try{ //risky behaviour - catch any errors
                writeObj = new FileWriter("C:\\Documents\\myText.txt" , true);
                printObj = new PrintWriter(writeObj);//create both objects
                printObj.println(myText);  //print to file
                printObj.close(); //close stream
             }catch(Exception ex){
                System.out.println("Error: " + ex);
             }
      }
}
```

Activity 9.1

Write a small utility program that prompts the user to enter their name, address and telephone number. The program should then write all of these details to a text file.

Activity 9.2

The Subway Program – write a program that can be used to calculate customer orders. The application should allow the user to pick a bread type, size and filling from a set menu, before summarising the order and calculating the order total. The program should be written into independent methods and append the summary to a text file. The set menu, and prices, can be improvised.

Programming Tips!

There is a common problem with 'hard-coding' directories, the problem being that directories can change from computer to computer. For example, some computers may not even have a 'C drive'. Alternatively, it is much safer to work with relative paths. The code below can be used to find the directory of where the Java file was executed from; once the directory is located it can be used to write a text file to the same location. This method will execute on all computers.

```
String myDirectory = System.getProperty("user.dir");
System.out.println("My Current Directory: " + myDirectory);
writeObj = new FileWriter(myDirectory + "\\myText.txt", true);
```

 © ZigZag Education, 2014

9.3 The Theory: Reading from a Text File

Using BufferedReader

In the previous steps, the `FileWriter` and `PrintWriter` classes were used to write information into a text file. Just as important is the technique of being able to read information from a text file into a program. Once information is loaded into a program, usually in the form of data structure, it can be manipulated.

Fortunately in Java, there is a useful class called `BufferedReader` which holds useful methods that enable a programmer to easily load data stored inside of a text file into a program. The `BufferedReader` class can be imported by adding the following line of code at the top of a program [`import java.io.*;`]. Like other file-handling exercises, a `BufferedReader` is considered risky, and thus must be appropriately handled inside of a [`try/catch`] technique.

The `BufferedReader` class must be instantiated before the non-static [`readLine()`] method can be used; the [`readLine()`] method is used to read each individual line saved inside of a text file. The Java syntax used to instantiate the object is [`BufferedReader re = new BufferedReader(new FileReader(fullDirectory));`]. Notice that the directory of the text file to be read is given as an argument to an instantiated `FileReader` object – a convenient object used to read character-based files.

Once an object has been created (from the `BufferedReader` class), a loop can be used to read each line of text stored inside of the text file. The condition of the WHILE loop [`(input_line = re.readLine()) != null`] consists of two parts; firstly, the code inside of the brackets is responsible for loading the next available line of text, from the text file, into the string variable `input_line`. The second part of the condition is used to check if the `input_line` variable is not equal [`!=`] to null (loaded with a value). In other words, the WHILE loop will continue to iterate as long as the next line of text exists in the text file. Finally, for each successful iteration of the WHILE loop, the line read from the text file (and stored into the `input_line` variable) is written to the command-line.

```
import java.io.*; //import class
public class MyTextReader{

   public static void main(String[] args){
       String myDirectory = System.getProperty("user.dir"); //get current directory
       String fullDirectory = myDirectory + "\\myText.txt"; //full path
       String input_line = null;
       try{ //risky behaviour
           BufferedReader re = new BufferedReader(new FileReader(fullDirectory));
           while((input_line = re.readLine()) != null){ //loop while NOT empty
                    System.out.println(input_line); //print to command-line
           }
       }catch(Exception ex){
            System.out.println("Error: " + ex);
       }
   }
}
```

 © ZigZag Education, 2014

Reading Data from File

The syntax below demonstrates how items stored inside of a text file can be loaded into an array list, ready for manipulation by a program. In this particular case the items, once loaded into an array list, are then printed to the command-line:

Try the code

```
import java.io.*; //import classes
import java.util.ArrayList;
import java.util.Iterator;
public class MyTextReader{

   public static void main(String[] args){
       String myDirectory = System.getProperty("user.dir");
       String fullDirectory = myDirectory + "\\myText.txt";
       String input_line = null;
        ArrayList<String> textItems = new ArrayList<String>(); //create array list
       try{
           BufferedReader re = new BufferedReader(new FileReader(fullDirectory));
           while((input_line = re.readLine()) != null){
                    textItems.add(input_line); //add item to array list
           }
       }catch(Exception ex){
            System.out.println("Error: " + ex);
       }
        Iterator myIteration = textItems.iterator(); //use Iterator to cycle list
        while(myIteration.hasNext()){ //while items exist
             System.out.println(myIteration.next()); //print item to command-line
        }
   }
}
```

Programming Tips!

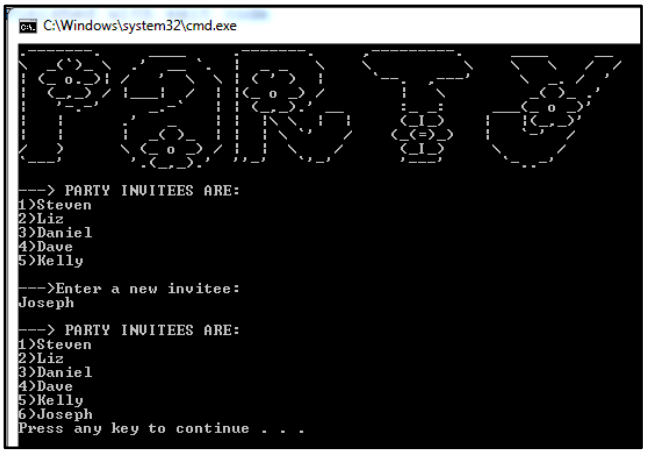
To complete the party program, create a global array list (of type string) and two other methods: [`party_invitees()`] and [`add_new_person()`].

The [`party_invitees()`] method should use the [`clear()`] method to clear all items in the global array list, before loading the names from the text file into it. The method should then cycle through all items (using the `Iterator` class, see Chapter 8) in the list and print them to the command-line.

The [`add_new_person()`] method should prompt the user for a name value; the method should then add the name to the array list, before writing the entire list back to the text file. Ensure that the `FileWriter` argument is set to false so that the text file is overwritten, as opposed to being appended. Finally, call the [`party_invitees()`] method again to show the updated list.

Activity 9.3

The Party Program – write a program that reads all invitees to a party, from a text file, and prints them to the command-line. The program should then ask the user to add another guest, and print the new list to the same text file. An updated version of the list should then be printed to the command-line.



 © ZigZag Education, 2014