Handout 6

File Input/Output: File, PrintWriter, FileWriter and Scanner classes.

In Java file i/o is supported by a wide variety of classes for a variety of specific purposes. We will use:

Input (Read) – use Scanner class
Output (Write/Append) - PrintWriter, FileWriter

File Output:

- Create a **`PrintWriter`** object, by passing it a String with a filename (absolute or relative).

- **`PrintWriter`**: enables writing values of **any** type using **`print()`, `println()`** methods

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

- **Must** be used within a **`try`** block with a **`catch`** clause, otherwise get a syntax error- Unhandled Exception, since **`PrintWriter`** constructor throws a **`FileNotFoundException`**

Example: **Create** a new file called `myfile.txt` and **write** a few lines into it.

```
/*  WriteFile.java - Demonstrates use of PrintWriter for simple
file output */
public class WriteFile {
     public static void main (String[] args) throws FileNotFoundException{

      //Create a PrintWriter object to write to myfile.txt
      // Opens file "myfile.txt" for writing
      // if myfile.txt does not exist in current directory
      //- create a new file. If exists - contents will be erased.
         PrintWriter pw = new PrintWriter("myfile.txt");
         pw.println("This is a new text file.");
                 // writes into file associated with pw
         for (int i = 1; i <= 13; i++){
             pw.print(i+", ");
         }
        pw.close();

    }}
```

To **append** instead of overwriting, use a slightly different mechanism: create a PrintWriter from a FileWriter as shown:

```
// open file for appending - second param must be true
FileWriter fw =  new FileWriter("myfile.txt", true);
PrintWriter pw = new PrintWriter(fw);
```

```
//AppendFile.java - append to a file
import java.io.*;

public class AppendFile {
      public static void main (String[] args) throws IOException{

                  // open file for appending - second param must be true
                  FileWriter fw =  new FileWriter("myfile.txt", true);
                  PrintWriter pw = new PrintWriter(fw);
                  pw.println("This line is added to an existing file.");
                  pw.close();

      }
}
```

File Input: Use **Scanner** class to read from file.

Scanner – a class for text input processing from **keyboard, file, or string**.
Must import java.util.Sacnner.
Constructing a scanner from file ***stringFileName***:

- **Scanner** *anyName* **= new Scanner (new File (*stringFileName*));**

To read input from the keyboard:

- **Scanner** *anyName* **= new Scanner (System.in);**

Scanner instance methods:
`anyName.nextInt();` - returns an integer. Skips all white space (space, tab, end-of-line characters) that appear **before** the integer.
`anyName.nextDouble();` - double
`anyName.next();`- returns the next token () , i.e. the String value consisting of characters up to, but not including the next white space or end of line (or another specified delimiter).
Essentially, reads one word, where word is a sequence of non-white space characters.
`anyName.nextLine();` - returns the String value consisting of characters up to, but not including the end of the current line.

For **checking** if the input stream contains unread data:

```
anyName.hasNext();
anyName.hasNextLine();
```

-return a Boolean value designating if more tokens or lines are left.

Example: **Using Scanner with File to read line by line**

```
/** Read file line by line */
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFileLineByLine {
  public static void main(String[] args) throws FileNotFoundException {
      String line;

            File inputfile = new File("sampleInput.txt");
            Scanner input =  new Scanner(inputfile);
             // read one line at a time
            while( input.hasNextLine()) // verify there are more lines
                                        // or will get an exception
            {
                  line = input.nextLine();
                  System.out.println(" read <" + line + ">");

            }
            // release resources to operating system when done
            input.close();

  }
}
```

Example: **Using Scanner with File to parse into words, separated by line breaks and commas:**

```
while( input.hasNext()) // verify there are more lines
                                        // or will get an exception
            {
                  line = input.next ();
                  System.out.println(" read <" + line + ">");

            }
```

Output:

Input file contents:	Output from first segment (using nextLine())	Output from second segment (using next())
One, two, three 45 4.67 four	read <One, two, three> read <> read <45 4.67 four>	read <One,> read <two,> read <three> read <45> read <4.67> read <four>

Practice problems:

File `menuitems.csv` contains data on restaurant menu items listed one dish per line, with the **type of the dish**, its **name** and its **price** separated by commas, s shown below.
Create programs for each of the problems below

```
appetizer,Bouillon in cup,6.7
appetizer,Queen olives,8
entrée,"Panfish, Meuniere",25
side-dish,German fried potatoes,5
entrée,Ribs of prime beef,19
dessert,Assorted cakes,15
entrée,"Scollops en caisse, Supreme",32
entrée,Irish stew,28
appetizer,"Marrow on toast, Bordelaise",18
salad,Lobster salad,14
…
```

1. Ask the user to enter dish type and calculate the average price of that type of a dish.
2. Ask the user to enter dish type and display the lowest-priced and the highest-priced dish of that type.
3. Ask the user to enter dish type and create a file into which you will record all information for the dishes of that type.
4. Create text a file that contains a menu, listing items of each type grouped together under an appropriate title as follows

```
              Appetizers
Bouillon in cup                $6.70
Queen olives                   $8.00
Marrow on toast, Bordelaise   $18.00
           .   .   .
               Entrée
Panfish, Meuniere             $25.00
Ribs of prime beef            $19.00
           .   .   .
```

Additional challenges:

- Make it look the best you can.
- Make the code as free of hardcoded values as you can.