S8354: Java From the Very Beginning Part II- Exercises

Ex. 1 Command Line Arguments

1) Open the **CommandLine.java** file which is in the **CommandLine** folder.

2) Your task in this exercise is to print out all the arguments passed to the program via the command line. Every Java application is passed its arguments as a parameter in its "main" method. In our program, these arguments are stored in the variable named "args":

```
public static void main(String [] args)
```

Recall from session 8352 that the [] brackets signify an array, in this case an array of strings. Can you remember the Java syntax to determine the length of an array, and to access the elements within an array? If not, you may need to look back at your notes from session 8352.

3) Find the place in the program where it says:

```
// TO DO:
// args is an array of Strings, use a for loop
// to iterate over the array and print its contents,
// one element on each line.
```

Use a **for loop** to iterate over the args array, printing out each argument in turn. The expression: **System.out.println(** ***string*** **);** can be used to write a string of text to the console.

4) Save your program.

5) Try running the application in the usual fashion. You should find that the application does not output any information in the console view. This is because we are not passing any arguments to the application at this point in time. Eclipse provides a mechanism for passing arguments to the application during runtime.

To attach arguments click the arrow on the "Run" icon  and select **Run**. A dialog box appears displaying the application's *runtime environment*. Ensure that **CommandLine** is highlighted (under the **Java Application** folder). If it does not exist, then select **Java Application** and click **New**. A new runtime configuration will appear for the **CommandLine** application.

6) Click the **Arguments** tab and enters some arguments in the **Program arguments** text area. By entering information here, you are effectively appending to the Java runtime command line. This information will be passed to the application and placed in the args array.
Click **Run** to start running the application.

7) Check the console view and ensure that the arguments were passed to the application correctly.

8) If you are familiar with the behaviour of C or C++, how does the output of your program differ from what you might have expected?

9 (optional)) Can you modify the program to display the text “You did not provide any arguments” if no arguments are provided on the command line?

Ex. 2 File Printer

In this exercise you will work with a slightly more complex program that prints out the contents of the files specified on its command line.

1) Open the **FilePrinter.java** file which is in the **FilePrinter** folder.

2) Run the existing FilePrinter application by passing the filename "FilePrinter\input.txt" as an argument. Use the procedure described in the previous exercise to add an argument to the command line.

3) Now try running the program again, but this time pass in the name of a file that doesn't exist, for example "FilePrinter\not_there.txt".
What happens?

4) Now we are going to modify the program so that it handles the error gracefully! The FilePrinter program uses a named block of code - called a "method" to do most of its work. We will learn more about methods in session 8358. In this case the method is called "displayFile", and we call it for each file passed to us on the command line.

Locate the **displayFile** method within the source code and examine the code within it. There are three possible places where errors may occur:

☒ when trying to open the file:
```
  fileReader = new FileReader(fileName);  // may throw a
FileNotFoundException
```

☒ when trying to read from the file:
```
  while ( (c = fileReader.read()) != -1) {  // may throw an IOException
```

☒ and when closing the file once we have finished with it:
```
  fileReader.close();  // may throw an IOException
```

5) Surround the code which may throw exceptions inside a try block.

6) Immediately after the try block, we need to add some catch handlers to deal with the errors that may occur. First add a catch block for the FileNotFoundException:
```
      catch ( FileNotFoundException fnfEx ) {
          // display error message here
      }
```
within the body of the catch block, display an error message saying that the file could not be opened - include the name of the file in the message.

7) Now add a catch block for the IOException:
```
      catch ( IOException ioEx ) {
         System.out.println( "Error reading from file." );
      }
```

8) Our program should always close any file handles it has open, regardless of what happens. This is a good choice for a finally clause. Create a finally block that closes the file if it is still open:
```
    finally {
```

```
    if (fileReader != null) {
      try { fileReader.close(); }
      catch (IOException ioEx) { ; } // nothing we can do now!
    }
  }
```

The call to close the file has been moved from inside the try block you created earlier to the finally block. The test of fileReader ensures that we only close a file that we previously opened. Note that the call to close the file can itself throw an exception, and this example illustrates the ability to nest try-catch blocks.

S8354: Java from the very beginning Part II - Sample Solutions

Ex. 1 Command Line

```
/**
 * A Java application to list the command line arguments
 */
class CommandLine {

  public static void main(String [] args) {

   if ( args.length > 0 ) {
     for (int i = 0; i < args.length; i++) {
        System.out.println("Argument " + i + " = " + args[i]);
     }
   }
   else {
     System.out.println( "You did not provide any arguments" );
   }

  } // end of main method

} // end of class
```

To test the program, refer to the notes from the exercise.

☒ In C and C++ the first element in the args array is the name of the program that was invoked ("CommandLine" in our case). Java does not provide access to the name of the program through the args array, and provides access to the program arguments only.

Ex. 2 File Printer

```
import java.io.*; // include Java's IO library

/**
 * FilePrinter application - displays the contents of the file or files
passed on
 * the command line to standard out.
 */
class FilePrinter {

  /**
   * Main entry point - display the contents of each file
   */
  public static void main(String[] args) {

    for (int f=0; f < args.length; f++) {
      displayFile(args[f]);
    }

  } // end of main method


  /**
   * Pretty print the contents of a file to the screen and handle any
   * exceptions
```

```
   */
  private static void displayFile(String fileName) {

    System.out.println(fileName + ":");
    System.out.println();

    FileReader fileReader = null; // declare outside of the scope of the
try block
    try {
      fileReader = new FileReader(fileName);
      int c;
      while ( (c = fileReader.read()) != -1) {
        System.out.print((char)c);
      }
    }
    catch (FileNotFoundException notFoundEx) {
      System.out.println("Could not open " + fileName);
    }
    catch (IOException ioEx) {
      System.out.println("Error reading from " + fileName);
    }
    finally {
      if (fileReader != null) {
        try { fileReader.close(); }
        catch (IOException ioEx) { ; } // nothing we can do now!
      }
    }

  } // end of displayFile method


} // end of class
```