CSC 260.002 Programming Assignment #1
Due Date: Thursday, September 1

NOTE: this assignment first steps you through how to use Eclipse. If you are already familiar with Eclipse, you can skip to part 4.

Part 1: Starting a Java Program in *Eclipse*

Start Eclipse. You will see a selection box called the "Workspace Launcher" as shown on the left side of figure 1. This box allows you to set the directory for your workspace files. You can set this location as the default so that this window does not appear in the future, otherwise it will appear every time you start Eclipse. Use the **`Browse`**... button to select an appropriate location. If you are using your home computer, set up a workspace directory that you will remember. If you are using a campus computer, use your flash drive or your student drive. You will probably want to use the same workspace for all of your programs this semester both for convenience

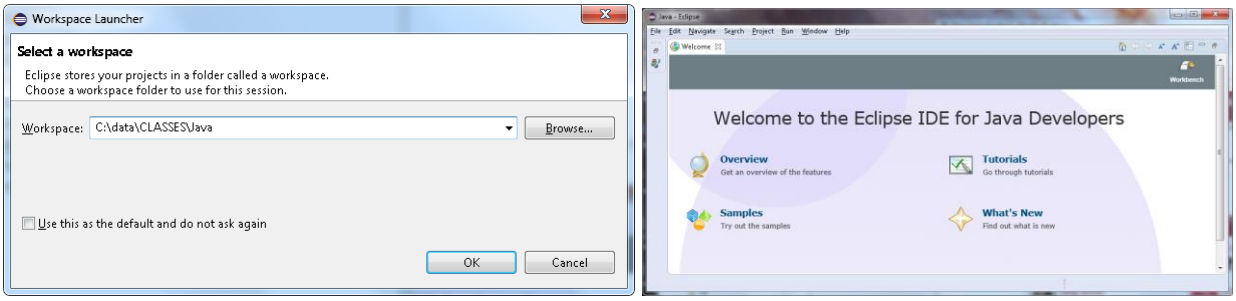

Figure 1 Workspace Launcher Window and Welcome Tab

The first time you run Eclipse, after setting up your workspace you will be taken to the Welcome tab (see the right side of figure 1). Close this (click on the X in the tab) and you will see the project interface window. Figure 2 shows the interface window which we explore shortly.

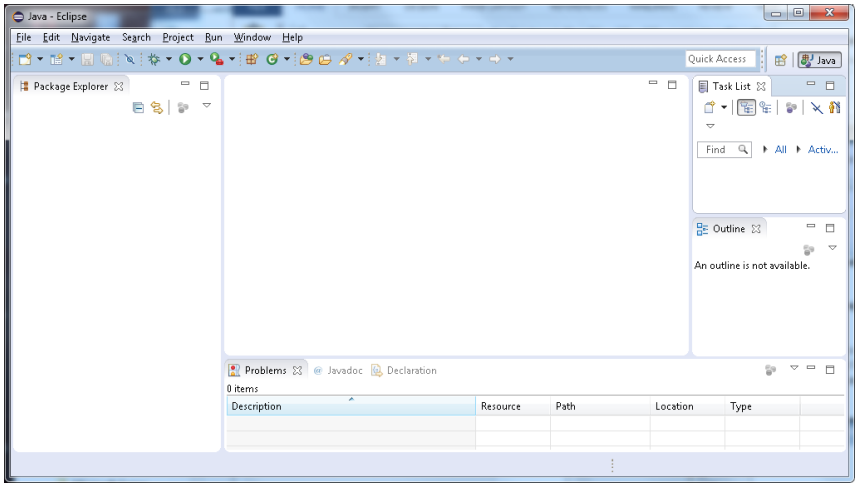

Figure 2: Eclipse Project Interface Window

Create a new project by selecting **`New`**... from the **`File`**... menu and then **`Java Project`** from the submenu. This brings up a New Java Project pop-up window, as shown in figure 3. From here, you will specify a project name which will then be placed in the workspace location you specified previously. You can change the default location by de-selecting the **`Use default location`** checkbox and then specifying a new location. For project name, use some meaningful name such as Program1 or something

descriptive of the assignment. For our first program, use **`HelloWorld`** (spell it exactly as shown here). The rest of the selections on this window default to what we want, so click on **`Finish`**.

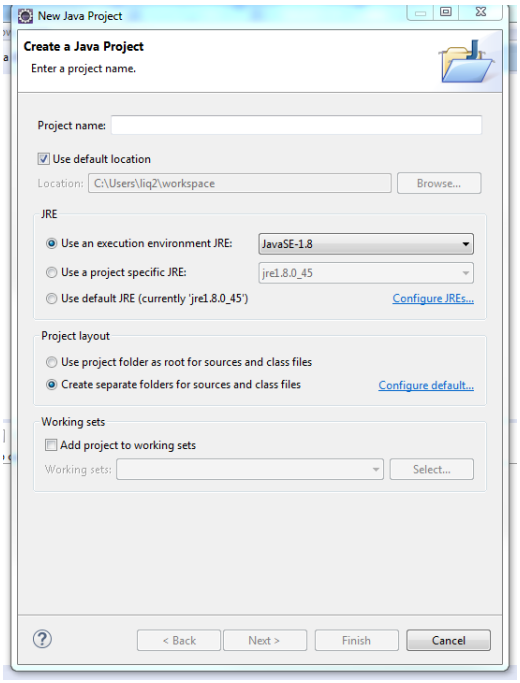

Figure 3: New Java Project Window

Let's explore the entire Project Interface window, as shown in figure 4. There are three panes of note. The leftmost pane is the Package Explorer. Although the figure shows a lot of packages, in your case there should only be one, HelloWorld (or whatever you named this project). As you add more and more projects to this workspace, you will find more listed. If you click on the triangle next to a package, it will expand to show you two items, src and JRE System Library. The src item is where your Java source code is placed. The JRE System Library lists all of the jar files that are available. You can see quite a few listed in figure 4. The upper right pane is your edit window where you will enter and edit your source code. The lower right pane is where error messages will appear when you compile your program and output will appear when you run your program.

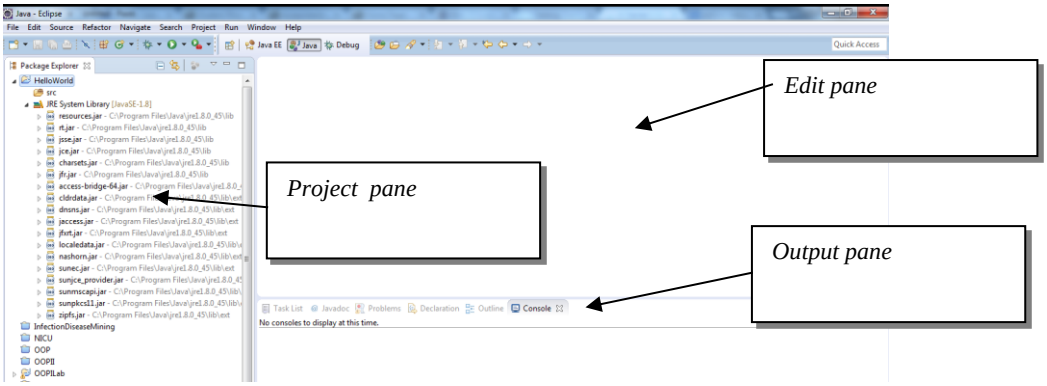

Figure 4: Java Project Interface Panes for HelloWord Project

Part 2: Writing, Compiling and Running a Program

Every project stores files which contain the Java source code of your project. For most of the semester, we will only have a single file although later in the semester we may have more. The single file will contain a

class definition. To create this file, make sure your project is selected in the Project Explorer window, for instance in figure 4 we see HelloWorld is selected. Select **`New`**... from the **`File`** menu and **`Class`** in the submenu. This brings up a New Java Class pop-up window, as shown in figure 5.

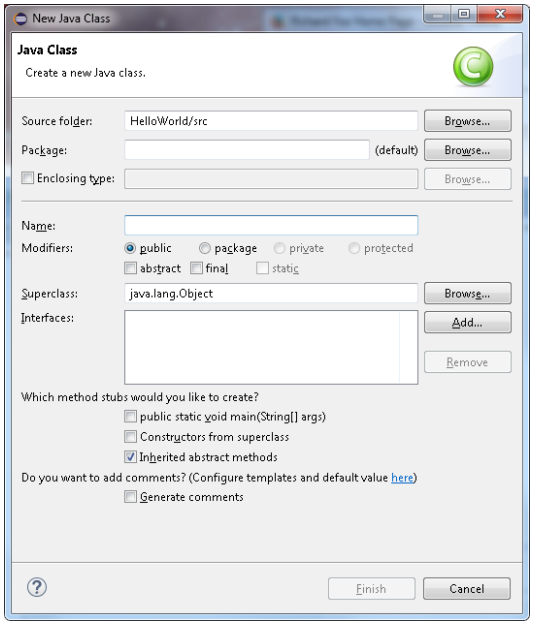

Figure 5: Creating a Class

In the New Java Class window, you will see the folder which will store this file and any package that you might want to place this file in. The next line is for a Package, if this is not empty delete whatever is there as we will not be creating packages. In the next part of the window you specify the name of your file, which will also be the name of your class. Eclipse will discourage you from using the same name as the project, but you can and in this case, do so by naming the class **`HelloWorld`**. Eclipse will automatically set up our class definition for us once we have established its modifiers. The rest of the items in the window allow Eclipse to start your class definition off with some default values. We want public (which should already be selected), java.lang.Object, and under method stubs, **`public static void main(String[] args)`**. Make sure these are selected (above you will see that the method stub is not correctly selected). We will not be dealing with abstract or final in this class, so leave those unchecked and similarly do not select Generate comments. Click on **`Finish`** which will create the new file, insert it into your project, and place into the edit pane code similar to what you see in figure 6.

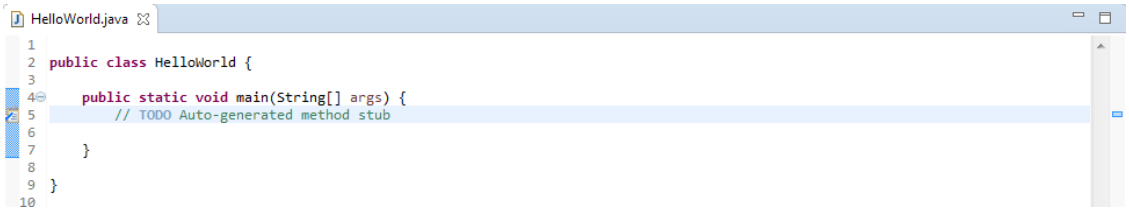

Figure 6: Automatically Generated Code for New Class

Where you see **`// TODO Auto-generated method stub`**, you will insert your code. The notation **`//`** means that what follows is a comment. Comments are ignored by the compiler being there for us humans to read. Delete this comment and enter the three lines of code in the tbox below (the three System statement). Also, above the public class HelloWorld line, add the 7 lines which start with /* and end with */. These are also comments. Where you see italics for your name and today's date, insert your name and today's date. Make sure you type the three System instructions exactly as shown.

```
/*
    Author: your name
    Course: CSC260.002
    Date: today's date
    Assignment: #1
    Instructor: Fox
*/

public class HelloWorld
{
     public static void main( String[] args )
     {
           System.out.println( "Hello World!" );
           System.out.println( "Welcome to the fun world of Java programming." );
           System.exit(0);
     }
}
```

Notes:

- The class name (HelloWorld in this case) **must** match the file name (excluding the .java part).
- The last instruction in the code above has a zero in parens.
- Spell all the words exactly as shown including upper and lower case letters (although you can change the capitalization of the text inside "" if you wish).
- Use the same punctuation including the parentheses, quote marks, semicolons, and brackets exactly as shown. DO NOT COPY AND PASTE from a document on-line into the editor because the quote marks ("") may copy incorrectly.
- Spacing does not have to be as shown above, but *Eclipse* automatically indents for you, so you should stick as closely to the above as possible.

This program, when run, executes the three System statements which output two lines of text and then terminates the program. Notice how all three executable instructions are placed between the { } of the main method. Nothing else in this program is actual "executable code".

If your program has syntax errors, they will be noted with a red circle with a white X in it to the left of the instruction that contains the error. You may also see certain words underlined in red. By hovering the mouse over the X you can see what Eclipse thinks is the reason for your error. In the next part, we explore types of errors and how to fix them. You must fix your errors before you can compile and run your program. So try to figure out, if you have syntax errors, how to resolve them.

Save your program (select **`Save`** from the **`File`** menu or select the **`Save button`** on the button bar, it looks like a floppy disk, it should be the third from the left). To run your program, you can either compile it first and then run it, or just run it in which case Eclipse compiles it first for you. To run it, select one of the following:

- the **`Run button`** (it's a white triangle pointing to the right in a green button)
- **`Run`** from the **`Run`** menu
- **`right click`** on **`HelloWorld.java`** in the **`Package Explorer pane`** and selecting **`Run As`** and from there **`Java Application`**

If you had not saved your file, you will be asked if you want to do so now.

NOTE: you should compile the program first before running it (that is, separate the two steps) because of a situation that may arise by letting Eclipse compile for you. The problem is this: assume you have written your program, compiled and run it. Now you make a change to the program. You go to run it again and

Eclipse will compile it for you. But if the revised version has syntax errors, then Eclipse will run the old version of the program, not the new, and you may not notice it. So its best to separate the steps. To do so, from the **Project** menu select **Build Automatically** (which is checked) to shut this feature off. This will make **Build Project** available from the menu and you will see **Build Automatically** no longer checked. Now, before running your program, select **Build Project** from the **Build** menu followed by running it using one of the three approaches given above. Don't forget to save your file first after making changes and before compiling.

Part 3. Debugging

There are three sources of errors for any program. The first is known as a *syntax* error. This error arises because there is something syntactically invalid with your program code. There are many reasons for having syntax errors such as misspelling words, forgetting to declare variables, having incorrect punctuation marks, or a common error in Java is not naming the file the same as the class name. The second source of error is called a *run-time* error. This error arises after successfully compiling the program but while it is executing something went wrong. For instance, if the program expects the user to input a number and the user types in a name instead, this results in a run-time error. Another source of run-time error is asking the computer to do something it cannot like store too large a number in an int variable or divide by 0. The final source of error is the hardest to catch and fix, a *logical* error. Here, your logic is incorrect. With very complex programs (consisting of hundreds of thousands or millions of instructions), finding logical errors is one of the biggest challenges that programmers will ever encounter. A logical error might be as simple as adding when you meant to subtract but it can also be an infinite loop so that your program never terminates. Here, we will explore syntax errors only but you will come across run-time and logical errors throughout the semester and quite likely your entire career.

Return to your Edit Pane and change the first word from **public** to **Public** as shown below.

```
Public class HelloWorld
```

This will cause the line to have an X next to it with Public being underlined. The error in this case is that public was expected instead of Public. If you try to run the program, you will find the old version running. Change your code back to what it was ("public" instead of "Public"). The X will disappear.

Now change the name of the class (same instruction) from **HelloWorld** to **Hello**. Again, you will see an error in this line. What is the error message this time? Correct the name. Now, remove the ; at the end of the first println statement. This instruction now has an error, what message do you get when you hover the mouse above the X? Add the ; and then remove one of the two } (either one) and you get another error. Fix it.

Let's explore a minor logical error. Your program contains two **System.out.println** statements. Change the first statement to **System.out.print** (remove the **ln** from println). Compile and run your program. No syntax error. Why is there a logical error? If you look at the output, you will see that the two output lines are now on one line. That in itself is not an error, but there is no space between them. You can fix this error by adding some blank spaces inside the quote marks either after the ! in World! or before Welcome. For instance, our first statement could become:

```
System.out.print("Hello World!  ");
```

What is the difference between print and println? With println, after the output a new line character is placed so that the next output statement will output on a new line. The print statement does not output the new line character so the next output statement will appear on the same line. By using a print instead of a println, while the output of the program is not incorrect, it does not look nice. You are done experimenting and may close this program. Part 4 is your actual assignment.

Part 4: Your Programming Assignment

Create a new project and new class, and write a program as follows. Our fastest spacecraft is the newly launched Juno, which with several gravity-assisted fly-bys will achieve a speed of around 165,000 miles per hour. All of our current spacecraft burn their fuel all at once at the beginning of their journey and then coast the rest of the way. At this speed, it would take over 17000 years to reach our nearest neighboring star, Alpha Centauri. By continually accelerating instead of coasting, our spacecraft can achieve a much faster speed and reduce the travel time greatly. For this assignment, you will compute the times it takes to reach different stars using coasting and using continual acceleration. Your program will input from the user the name of a target star (a String), the distance in light years (a double) and the amount of continual acceleration (an int in feet per second). Coasting speed will be 165000 (make this an int constant). Computing the two times is given below. First use step 1 to compute the distance in miles to the star.

1. convert distance from light years to miles:
   lightSeconds (a double) = lightYears * 365 * 24 * 60 * 60
   distanceInMiles (a long) = lightSeconds * 186000 (light travels 186,000 miles per hour)
   since distanceInMiles is a long, you need to perform a cast.
2. compute time for continual acceleration:

   $$\text{time}_{\text{method 1}} = 2 * \sqrt{\frac{distanceInMiles * 5280}{acceleration}}$$

   We multiply distanceInMiles by 5280 to convert from miles to feet. Use Math.sqrt for square root.
3. compute time for a constant velocity:
   $\text{time}_{\text{method 2}}$ = distance / velocity

The first time computed (equation 2) is in seconds. Convert it to years by dividing by 365 * 24 * 60 * 60. The second time computed (equation 3) is in hours. Convert it to years by dividing by 24 * 365. Both times should be int values so you will again have to perform a proper cast. Output the name of the star, the distance in light years, and the travel time in years using both methods. Run your program on the following four data sets (the table shows the expected time for the first two so that you can check your work). Sample input/output for the first data set is given below. Note that inputting a String using a Scanner restricts you to a String with no blanks, so if the star has a blank in its name, join the words with _ as in Alpha_Centauri.

Name	Distance	Acceleration (fps)	Time 1 (years)	Time 2 (years)
Alpha Centauri	4.37 light years	32	4	17734
Tau Ceti	11.9 light years	5	17	48292
Delta Eridani	29.5 light years	2		
Biham	96.6 light years	8		

NOTE: we are not restricting velocity to the speed of light so your output for time 1 will not be accurate

```
What is the name of the target star?  Alpha_Centauri
How many light years distant is it?  4.37
What is the acceleration in feet per second (an int)?  32

To reach Alpha_Centauri at 4.37 light years, it will take 4 years using
method 1 and 17734 years using method 2
```

Copy your input/output from the output pane for all four runs and paste this at the bottom of your source code inside of comments (/*…*/). Hand in your source code including the 4 input/outputs. Make sure you have commented your code well.