WA1755 Introduction to Java and Java Enterprise Using Eclipse and JBoss

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - The HelloWorld Class

Time for lab: 40 minutes

In this lab, you will use Eclipse to write a very simple "Hello World" Java class.

Part 1 - Start Eclipse and Explore the IDE

Eclipse is a **Java Integrated Development Environment (IDE)**. It is a GUI-driven and tool-assisted program that we can use to develop Java code in an effective and efficient manner. While developing Java code does not necessarily require an IDE (all the exercises in this lab guide could be achieved by using a simple text editor and the command line compiler), using an IDE will greatly simplify development and decrease coding time. As such, we will use Eclipse to develop all our code in this class.

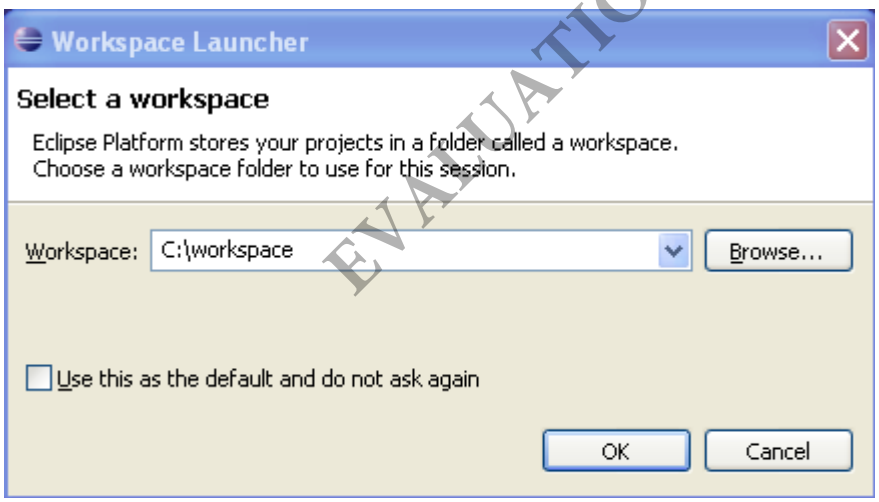
We will begin by launching Eclipse and exploring some of its facets.

__1. Open a file browser and navigate to **C:\Software\eclipse**.

__2. Run **eclipse.exe** by double clicking in the file.

The *Workspace Launcher* window will appear.

__3. Change the **Workspace** to **C:\workspace** as shown below.

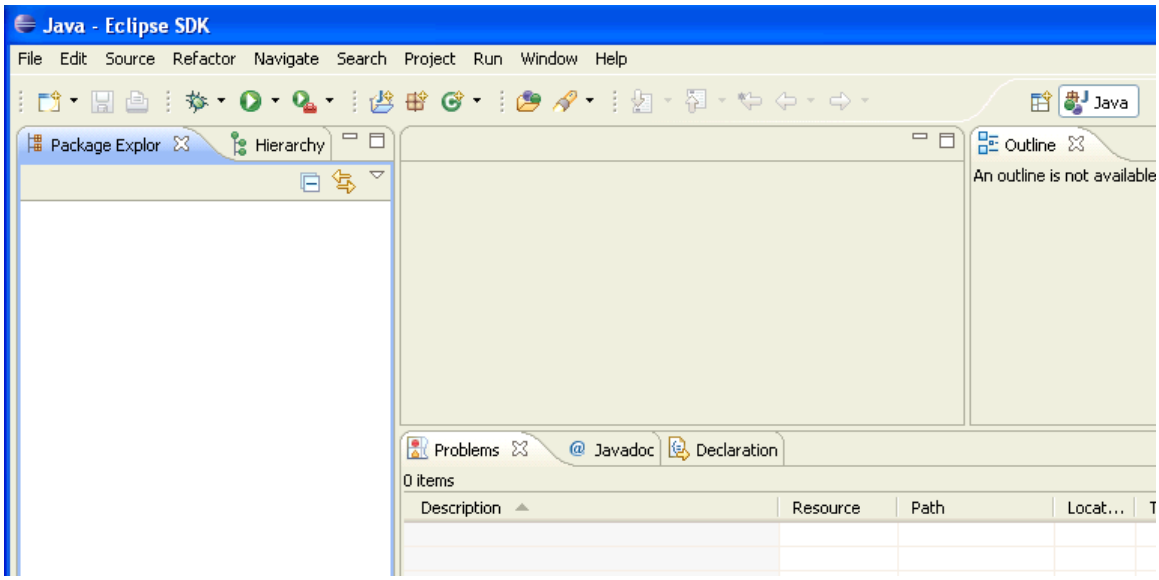


__4. Click **OK**.

__5. Close the Welcome screen if opens.

__6. From the menu bar, select **Window | Open Perspective | Other**.

__7. Select **Java** and click **OK**.



What has happened? Simple: you have changed *perspective*. A perspective in Eclipse is a collection of views chosen to achieve a specific task. We are currently looking at the **Java** perspective, for which Eclipse opens the appropriate *views*.

A view is simply a tiled window in the Eclipse environment. For example, in the screen shot above, you can see the *Package Explorer* view, and part of the *Outline* view.

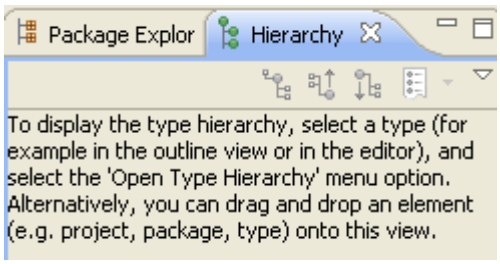
Look at your Eclipse environment and locate the following *views*:

The *Problems* view. This should be at the bottom of the Eclipse environment. Any coding/compile errors you make will be displayed here and (in a fit of unbridled pessimism) you will most likely be referring to this view quite a bit.

The *Package Explorer*. This is on the left of the IDE.

This will allow us to navigate through the Java classes we will be creating shortly.

__8. Locate the *Hierarchy* view. To find it, click the *Hierarchy* tab next to the *Package Explorer* tab.



__9. You can switch between the two views by clicking the appropriate tabs.

__10. Similarly, locate the *Javadoc* and *Declaration* views.

The big empty space in the middle of the IDE does not look very interesting right now; that is because it is the *Code* view – and we currently do not have any code to display. Do not worry about it for now.

__11. Each view can be moved and resized. Click on the *Package Explorer* tab and drag it over the area where the *Problems* view is. Note that the view "docks". Experiment with resizing and moving. Don't worry about "messing up" the perspective, because we will *reset* it in a moment.

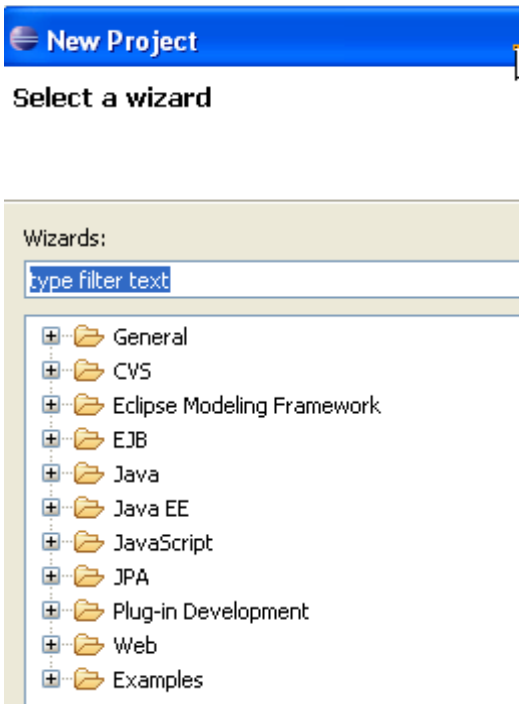
__12. From the menu bar, select **Window** | **Reset Perspective** and click **OK** in the box that appears. All the views should "reset" to their original layouts. Note that it is possible to *Save* a perspective that you have changed. This allows you to customize exactly which views you want open and how you want them laid out. This is something you will probably do as you become more experienced with Eclipse.

Part 2 - Create A New Java Project

Before we can create any Java code, we first need to create a *Java Project*. A project in Eclipse is essentially a folder for related files. We will create a new project and then create our classes in that project. Later on, if we work on a completely unrelated Java class, we could then create a separate, new project and create those classes in there. This helps keep our code environment organized.

__1. From the menu bar select **File** | **New** | **Project...**

When you do this, the *New Project* window will appear.



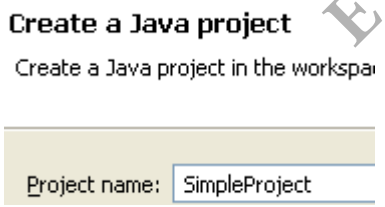
Eclipse can actually handle many different types of projects, depending on the code that needs to be developed. In our case, we will create a simple Java project.

__2. Expand **Java** and then select **Java Project** and click **Next**.

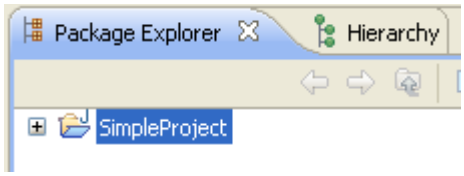
The *New Java Project* wizard will begin.

Here, we will describe the details of the Java project that will contain our simple HelloWorld Java.

__3. Set the *Project name:* to be **SimpleProject** and click **Finish**.



The project will be created. How do you know the project was created? Simple. Look in the *Package Explorer* view.



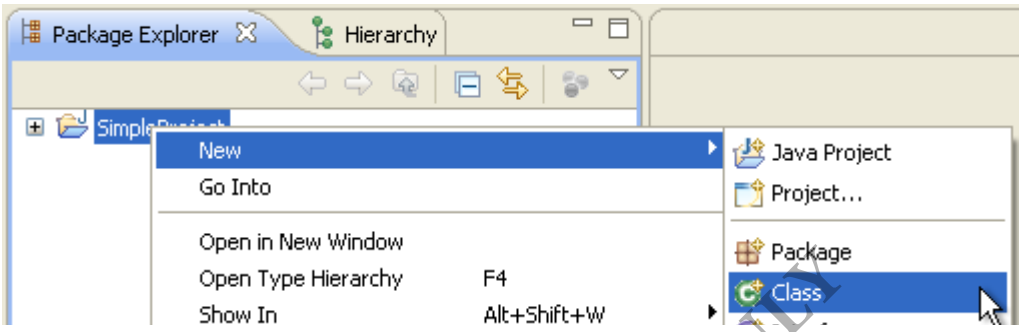
You should see the newly created project listed there. Our Java code will go in this project.

Part 3 - Create A Java Class

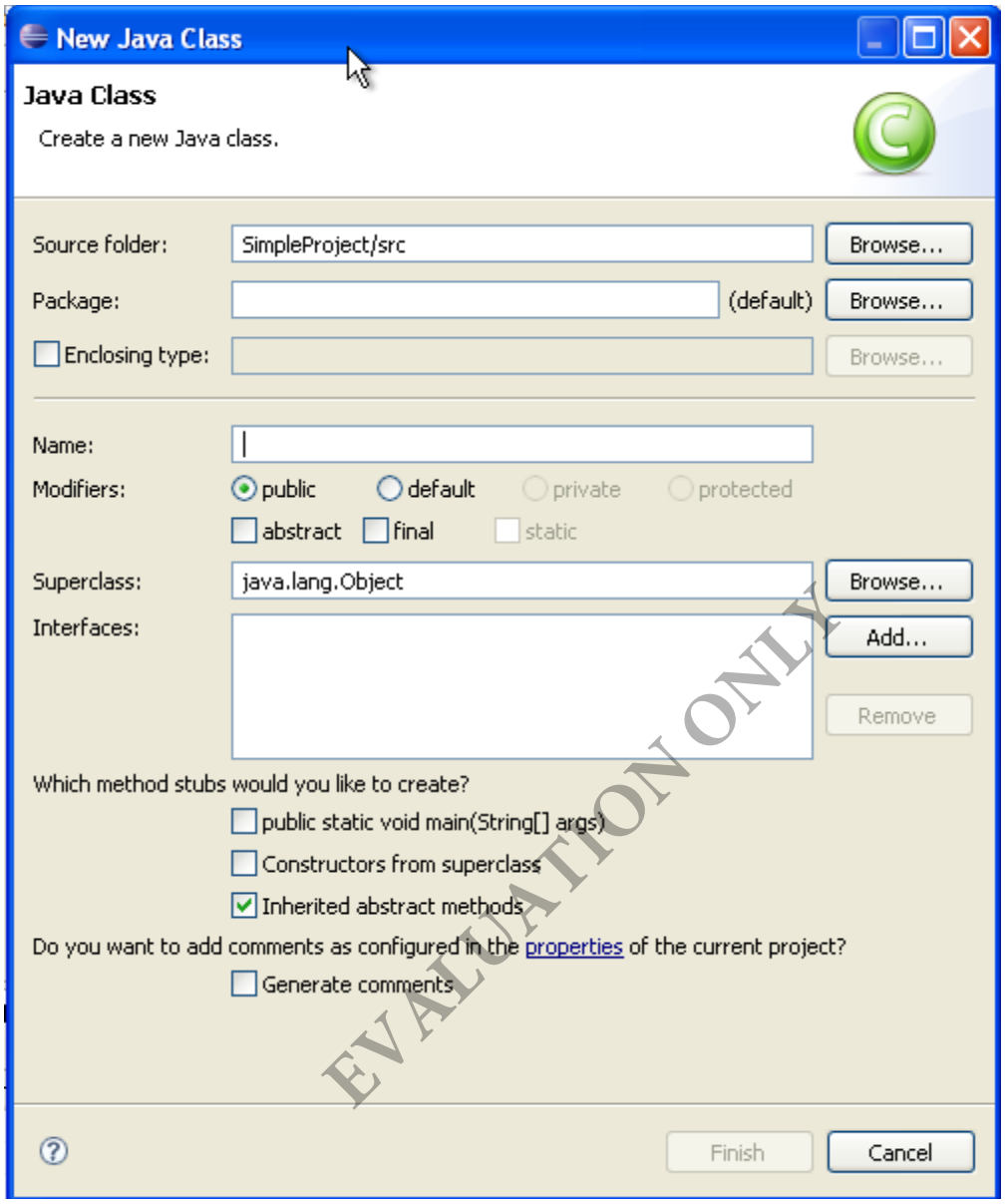
Recall that in Java, all code must be contained within a Java class. Within a class, code is typically placed within *methods*, and any class can have a **main** method which will serve as its "executable" method.

We will now create a Java class called "HelloWorld" that will have a single **main** method, and this method will simply print out "Hello World" to the *console*.

__1. In the *Package Explorer* view, right click on **SimpleProject** and select **New | Class**



The *New Java Class* window will appear.



This wizard will create a new Java class for us, and allows us to specify some options on the class. When we click the *Finish* button on this wizard, Eclipse will generate a new Java class according to the options we specify here. Wizards like this are one of the reasons that Eclipse is so useful; while it would have been possible to generate the class ourselves, Eclipse does it much faster.

Most importantly, we need to give the class a **name**.

Java classes should also belong to a **package**, for naming convention reasons. Think of a package as a name prefix. Imagine you are working at an insurance company in a Claims department, and you create a class called **Account**. However, imagine that another developer that you are working with (same company, but in the Billing division) also creates a class called **Account**. These two classes are completely different, but have the

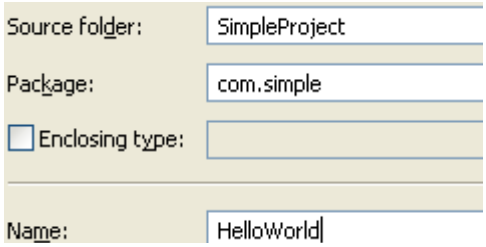
same name. How would we know which one to use?

The answer is that each class can be placed in a *package*. A package is a string of characters separated by periods (e.g. com.mycompany.www). The characters can be anything (although typical Java naming convention usually has your company name or URL as a part of the package name), and it is pre-pended to the class name. So a "fully qualified" class name is the name of the class preceded by its package.

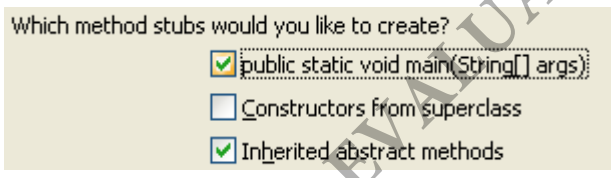
In the example above, the Billing department could be assigned a different package name from the Claims department; so there could be **com.insuranceco.billing.Account** and **com.insuranceco.claims.Account** as two separate entities. Even though the classes are both called **Account**, they belong to unique packages, thus allowing for differentiation.

In any case, we will use the *New Java Class* dialog to specify a class name and package name for our new class.

__2. For the *Package* enter **com.simple** and for the name, enter **HelloWorld**



__3. In the section *What method stubs would you like to create?* make sure that the box for **public static void main(String[] args)** is checked.

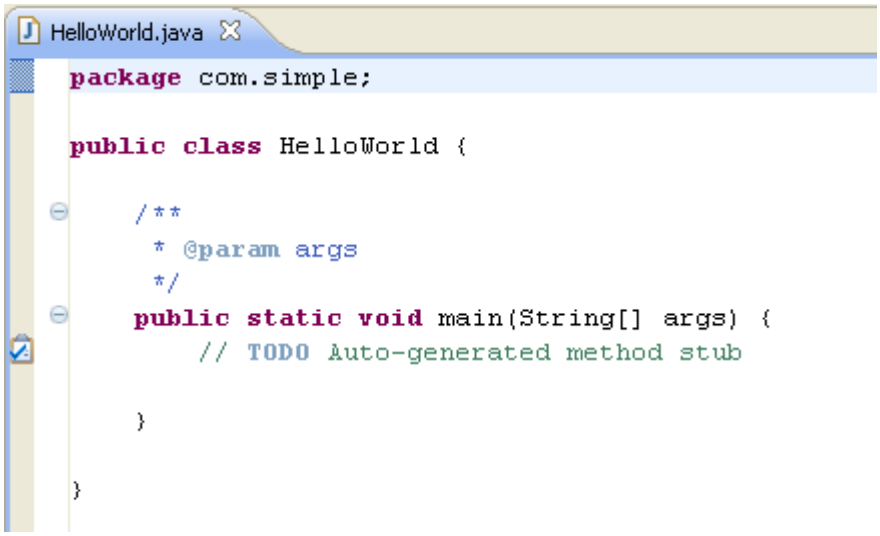


This means that when the class is created, Eclipse will automatically generate the **main** method for us.

You can ignore the rest of the options for now.

__4. Click **Finish**

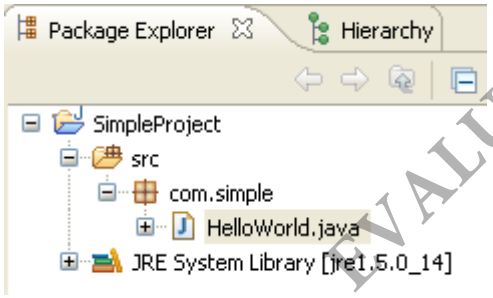
Eclipse will now generate the Java class. A lot has now changed in the various views. Firstly, the empty space that was in the center of the environment is now showing Java code. This is the Code Editor view, and it is showing the results of the *New Java Class* wizard's generation. Examine it.



Firstly, notice that the class this view is currently showing is called **HelloWorld.java**. You can tell this by looking at the name of the tab.

This Code Editor view is *editable*. You can place the cursor inside this view and type away; this is where you will do your coding. However, before we examine the code, let us examine some of the other views.

__5. Examine the *Package Explorer* view.



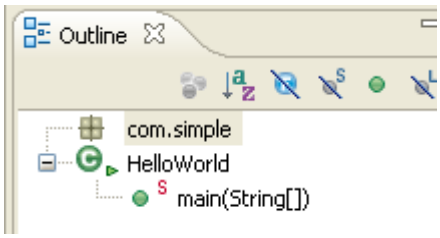
The *Package Explorer* provides us with a high-level navigable view of each project and their contents. At the moment, the only project is our **SimpleProject**

Note the tree structure. The top level is our project (**SimpleProject**) and immediately underneath that is the package **com.simple**

Beneath the package is our class, **HelloWorld.java**. If you double-clicked on it (or any other class that might be listed here), Eclipse would open it in the code editor window.

If we created new packages/classes in the project, they would similarly appear here in the tree structure.

__6. Examine the *Outline* view.



This view presents a summary of the class that is currently open in the code editor view. At the moment, it is examining the HelloWorld class (as represented by the green C) that belongs to the **com.simple** package. The outline is also showing that the class currently has one method: a *static public* method called **main(String[])**. We know it is a *public* method because of the green circle next to it, and we know that the method is *static* because of the **S**. (We will discuss the concept of static and public modifiers later on).

Remember that this view is just a summary of the Java class. As we add/remove code from the actual Java class, this view will update accordingly.

Turn your attention back to the code editor window. Let us examine the actual Java code now. All this code has been generated for us by Eclipse as a convenience.

Notice that the code is color coded. *Keywords* are in purple, *comments* are in blue and the remaining text is in black. Also notice the use of curly braces '{' and '}' to denote where classes and methods end and begin.

__7. Look at the first line:

```
package com.simple;
```

This the package declaration. This should always be the first line in a Java class. This line simply states that the following class belongs to the **com.simple** package.

__8. Examine the next line:

```
public class HelloWorld {
```

Here, we *declare* the class. We are saying that this is a new class that is called **HelloWorld** and that it is **public** (we will discuss **public** modifiers later).

This is followed by an opening curly brace '{'. Later, at the end of the file, this should be matched with a corresponding closing curly brace '}'. Locate this closing brace now.

Any code in between these braces will be considered belonging to this class. This shows the use of braces to indicate *scope*.

An unmatched curly brace pair will be flagged as an error and the code will refuse to compile. An important part of Java programming is learning how to properly open/close scope.

__9. Examine the next few lines:

```
/**
```

```
    * @param args
    */
```

These are special **JavaDoc** tags. JavaDoc is a tool that allows developers to place formatted comments inside a Java class. These comments can then be pooled together to form a good basis for documentation. We will not worry about this here. You can ignore this for now.

__10. Examine the next few lines:

```
public static void main(String[] args) {
   // TODO Auto-generated method stub
}
```

This is a **method**. The first line declares the method. It is a **public** method that is **static**, and its *return type* is **void**. It also takes one argument: an array of String objects. This argument is called *args*.

Remember that **main** is a special method. Since this HelloWorld class has such a main method, it can be "run". When the class is run, any code inside this method will be executed. Not all classes will necessarily have main methods, however.

Again, note the use of curly braces to denote where the method begins and ends. Just like the class declaration, any code that is in between these braces will be a part of this method.

Currently, the body of the method only has one line and that line is a comment that was placed there by Eclipse. A comment is a piece of **non-code text** that is placed within code as a note to the person reading the code. It can form the basis of documentation, or simply serve as helpful reminders as to what the code is doing for anyone reading the code. Comments in Java can either be preceded with a // sequence, or surrounded by a /* and */ sequence. Remember that this is **non-code** which means the compiler will not attempt to process it.

Keeping this in mind, we can see that the method currently does nothing. Let us change this now.

__11. Using the editor, delete the line

```
// TODO Auto-generated method stub
```

__12. Replace it with the following line:

```
System.out.println("Hello World");
```

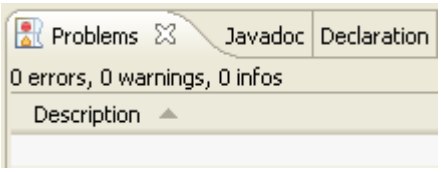
What is this line doing? This line is invoking the **println** method of **System.out** and passing the string "**Hello World**" to it. **System.out** is an object that is provided by the Java language itself and represents the "console" (the default location where Java outputs to; typically, the screen).

println is a method that is provided for **System.out** and it takes one argument: a **String object** representing what needs to be printed. (Objects will be discussed in more detail in class)

Pay special close attention to the brackets '(' and ')' as well as the trailing semi-colon. Every Java statement should end with a semi-colon.

__13. Your code is complete. Save your code by going to **File | Save** or by typing **Ctrl-S**

__14. Check the *Problems* view. It should be empty.



If you had made any code mistakes (e.g. typos, syntax errors, etc), an item would have appeared in the list.

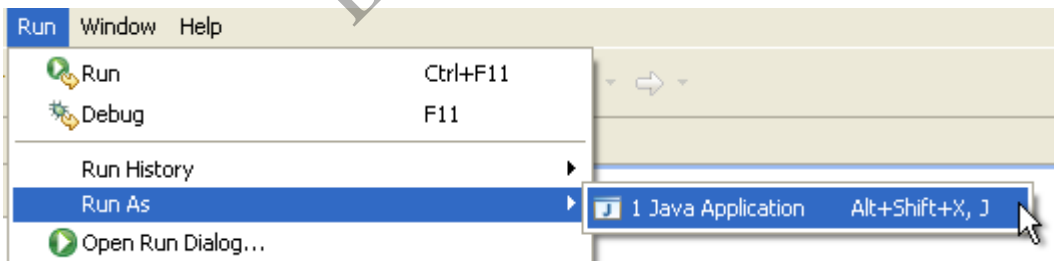
If any errors are there right now, go back and double check the code you typed in. Correct any mistakes you find. Do not proceed until there are no problems listed here.

__15. Your code is now complete. The next step is to run it.

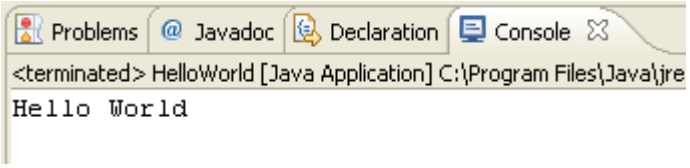
Part 4 - Run the Code

Now that we have completed writing the code, we can run it to see the results. Running Java code implies launching an instance of the Java Virtual Machine (JVM), and then loading the appropriate Java class and executing its **main** method. Fortunately for us, Eclipse makes that simple.

__1. From the menu, select **Run** | **Run As** | **Java Application**



A new view (the *Console* view) should appear in the same area where the *Problems* view is. Any text sent to the console (e.g. by a call to **System.out.println()**) will appear in this view.



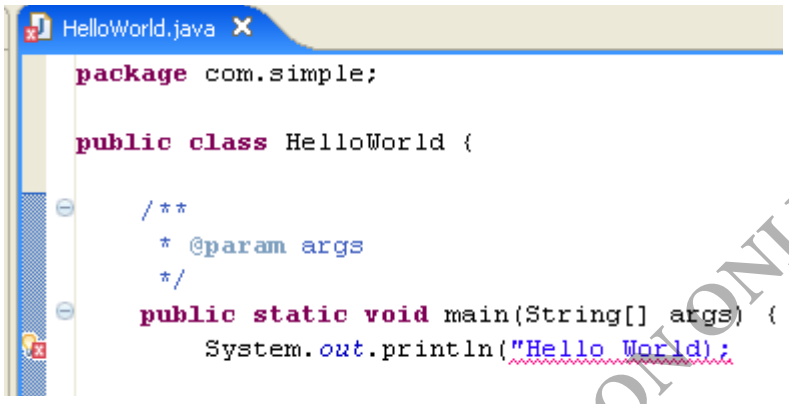
Voila! Our Java code has executed and Hello World has been printed. Congratulations. You have written and run your first Java class.

Part 5 - Change the Code

Just as an exercise, let us change the code a little.

__1. Let us now introduce an error to see what sort of error handling features that Eclipse offers. Delete the closing " mark from the **println** statement and save your code.

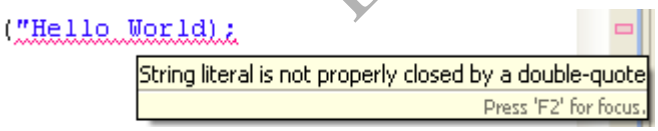
What happens?



Note that a red X has appeared in the tab title (next to "HelloWorld.java"). This indicates there is an error in the class.

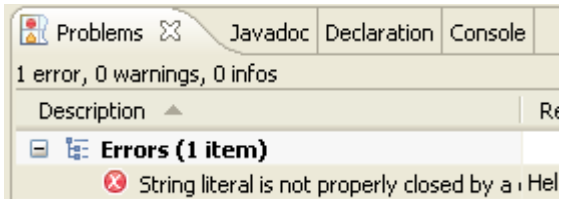
__2. Also note that in the left margin a similar red X has appeared on the same line that the error occurs on. Finally, note that the un-closed string ("Hello World); is itself underlined in red.

__3. Float the mouse cursor over the red underline.



A little pop up appears with a description of the problem. This is a hint to tell you what the corrective action should be. Do no fix it yet though!

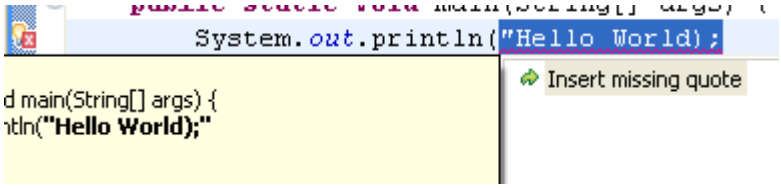
__4. Locate the *Problems* view.



Note that the error in our code has been detected. You may have to resize the

Description field to see the whole text. This view shows what resource (Java file) the error occurs in, and even shows the line number. This becomes invaluable if you are working on multiple Java files. Finally, if you double click on the error listed doing so will open the editor on the exact spot where the error is. Again, this is invaluable if you are working with multiple Java files.

__5. Finally, click on the red X in the left margin.



__6. Two boxes pop-up. Ignore the box in yellow; it is the box in white that is interesting. Eclipse has located the error and is actually suggesting a fix! Double click where it says *Insert missing quote*

What happens?

```
public static void main(String[] args) {
    System.out.println("Hello World);"
```

Eclipse has indeed added a closing quote mark, but unfortunately in the *wrong* location (after the semi-colon, instead of before the closing bracket). Let us see what this "fix" does.

__7. Save the file.

__8. Look at the *Problems* view. There are now two errors! So while the fix suggested by Eclipse was on the right track, it ended up doing more good than bad.

__9. Fix the errors properly (by putting the quote in the right place, immediately after **World** and save the file. Make sure all errors are gone.

From this, you should see that the error handling features of Eclipse are quite thorough; however, you should also see that it is not perfect. Additionally, Eclipse can only catch *compile time* errors (i.e. syntax errors and errors in the actual code) and not *run time* errors. The difference should become clear to you as work through the rest of these lab exercises.

__10. Immediately after the existing **System.out** line, add another line of code as follows.

```
System.out.println("Goodbye, cruel world!");
```

If you are not particularly conscious about using tabs, spacing, and line breaks properly in your code (i.e. code style), it can end up looking a little disorganized. Fortunately, Eclipse can format code for us so it looks neatly arranged.

__11. Right click anywhere in the code editor and select **Source | Format**.

If your code was not "neatly" written, it will be miraculously re-formatted to look perfectly neat and organized.

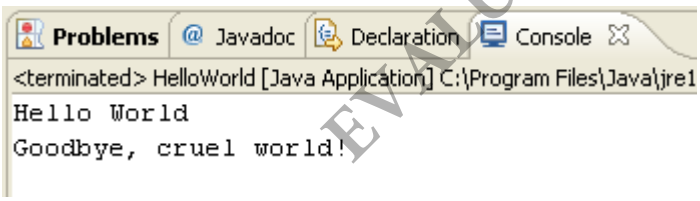
Note that in the tab for the code editor (where the file name **HelloWorld.java** is displayed) an asterisk has appeared. This is to indicate that the file was changed, but has not been saved.

__12. Save the file. Notice that the asterisk in the code editor tab disappears, indicating the file has been saved. This means the file is "current". There should be no errors. It should look like the following.

```
package com.simple;

public class HelloWorld {

    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Hello World");
        System.out.println("Goodbye, cruel world!");
    }

}
```

__13. Run the code again. (To do this, go to the menu bar and select **Run | Run As | Java Application**

The console window should show your new updated output.



__14. Close Eclipse by select **File | Exit**.

Don't worry about losing the state of your project; the next time you open Eclipse, it will remember what your last state was.

Part 6 - Review

In this lab exercise, you explored Eclipse and coded your first Java class.

You saw that a Java class has a name and (usually) a package. It has methods and its method/class scope is determined by braces. You learned how to print something to the console by invoking the **System.out.println** command.

Using Eclipse , you created a Java project to contain the HelloWorld.java class. You

learned about perspectives and views, and used wizards to generate some code for you. Finally, you also saw some of the error-handling features of Eclipse.

Lab 2 - Refining The HelloWorld Class

Time for Lab: 30 minutes

In this lab exercise, you will make a few changes to the HelloWorld class that you created in the last lab exercise. Specifically, you will make it a little more interactive. By using the **Scanner** class, you will actually prompt the user for a name which will then be printed back out.

By doing this, you will gain more exposure to both Eclipse and the Java language. You should start to build a better understanding of the syntax of Java.

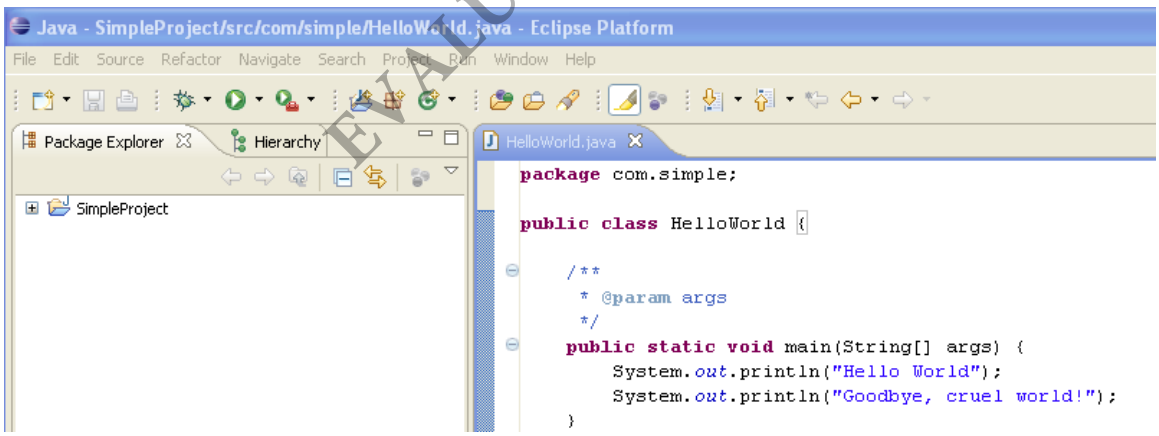
Part 1 - Edit the Class

Right now, our code gives out a generic "Hello World" greeting, which is a tad on the mundane side. We will edit the class to print out a request for a name, and then use the **Scanner** class to extract input from the user. A more personalized greeting will then be delivered to the user.

__1. You should have shut down Eclipse at the end of the last lab exercise, so now we will restart it. Launch Eclipse. **(C:\Software\eclipse\eclipse.exe)**

__2. When prompted to *Select a workspace* just click **OK**

When Eclipse starts up, it should look exactly like it did when you shut it down. Eclipse "remembers" the last state of the various views so it is easy to keep working where you left off.



__3. In the **HelloWorld.java** class, **remove** the lines:

```
System.out.println("Hello World");
System.out.println("Goodbye, cruel world!");
```

__4. Then **replace** it with the following code:

```
System.out.println("Hello.  Please enter your name:");
```

Nothing interesting here; we are merely changing the greeting text.

__5. Immediately after that code, enter the following:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

You will notice that as you were typing this, Eclipse will pop up a box underneath the cursor listing various class and package names; this is a convenience that Eclipse is trying to offer; specifically, it is trying to give you a *code assist* (i.e. guess what class/method you are trying to invoke, allowing you to click it instead of typing the entire thing). This can be handy, but is an advanced feature; ignore those pop up boxes for now.

What is this line doing? Well, we need to use the **Scanner** object, and doing so will require us to declare and initialize an *instance* of the **Scanner**. This is precisely what we are doing here. **Scanner** is a class provided by Java that is in the **java.util** package.

The first part of the statement (**`java.util.Scanner scan`**) is the *declaration* and the second part (**`new java.util.Scanner(System.in)`**) is the *initialization* of the object.

Note that the initialization uses the **new** keyword, which implies that we are using a *constructor* on the **Scanner** class. This creates a *new instance* of the class.

Also note that the reference to **`java.util.Scanner`** is a *fully qualified* reference, meaning the entire package/class name has been specified. This can be clunky, and we will see how to shorten this later in this lab.

Finally, notice that the *constructor* of the **Scanner** takes an argument which in this case is **System.in**. **System.in** represents an 'input stream' that Java can obtain input from. By default, this is the keyboard.

So, put together, this line creates a new instance of a Scanner class, and sets it up to read data from the keyboard.

__6. Immediately after that code, enter the following:

```
String name = scan.nextLine();
```

This line declares a new String variable (called **`name`**) and assigns it to the result of a call to `scan.nextLine()`. In effect, when the user types something into the prompt and hits Enter, the text that is entered will be placed inside the `name` variable.

__7. Finally, enter the following line of code:

```
System.out.println("Hello, " + name);
```

This is similar to the usual System.out.println code we have used before, but with a twist; we are now using a variable inside the print argument.

The code

"Hello, " + name

Uses the + symbol as a *concatenation* tool. Basically, it combines the string `"Hello, "` with **`name`**. A string can be concatenated with another string, and the result is a larger string. It is this larger string that is actually sent to **println**.

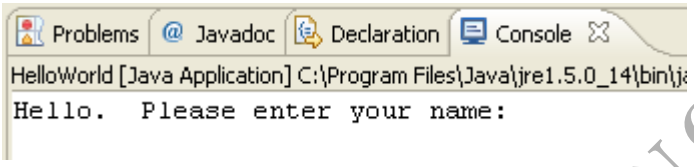
__8. Our code is done! Save your code. There should be no errors. If there are, fix them before proceeding.

Part 2 - Run the Code

We can now execute our new, more interactive, HelloWorld class.

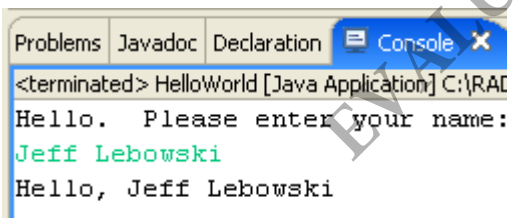
__1. Run the code as you did before. (Click anywhere on the code editor and then go to **Run | Run As | Java Application**)

Examine the console view.



This looks the same as before; our standard **println** commands are in use. However, what is different is that you can now enter text into the console.

__2. Click anywhere on the console view and type your name, followed by the Enter key. The Java class will then respond with an appropriate message.



Note that the code you type is in a different color.

__3. Run the code multiple times to make sure it works.

Congratulations! You have managed to add some interactivity to your simple HelloWorld program.

Part 3 - Import Packages

Earlier, when you entered the code to declare the **Scanner** object, you saw how the fully qualified class name was used. This is because, by default, Java does not "know" about the **Scanner** class. In order to locate and use it, the fully qualified package name had to be provided.

If you know that the **Scanner** class will be used multiple times, it may be a good idea to **import** the class. Importing a class will allow the code to use the class without having to fully qualify it. Typically, a Java program will import all the appropriate classes to simplify coding. We will do this now.

__1. In the HelloWorld.java class, add the following statement immediately after the **package** statement:

```
import java.util.Scanner;
```

__2. Now, change the line where the scanner is initialized to the following:

```
Scanner scan = new Scanner(System.in);
```

Note that we no longer need the fully qualified **Scanner** package name because it has been imported.

__3. Save the code. There should be no problems.

__4. Run the code. It should work the same as before.

Part 4 - Review

In this lab, you edited your HelloWorld class to add some interactivity. You saw the use of a constructor to initialize a new class instance, and you saw how a string can be assigned to the result of a method call.

You also saw how importing of packages can simplify coding.

Lab 3 - The Arithmetic Class

Time for Lab: 45 minutes

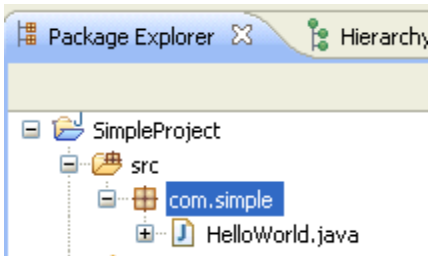
In this lab, you will build a slightly more complex class than the HelloWorld class; it will perform some basic arithmetic operations, and print out the results of the operations.

This lab is designed to give you yet more experience with the Java syntax. Additionally, you will see an **if** statement to handle *branching logic* for the first time.

Part 1 - Create The Arithmetic Class

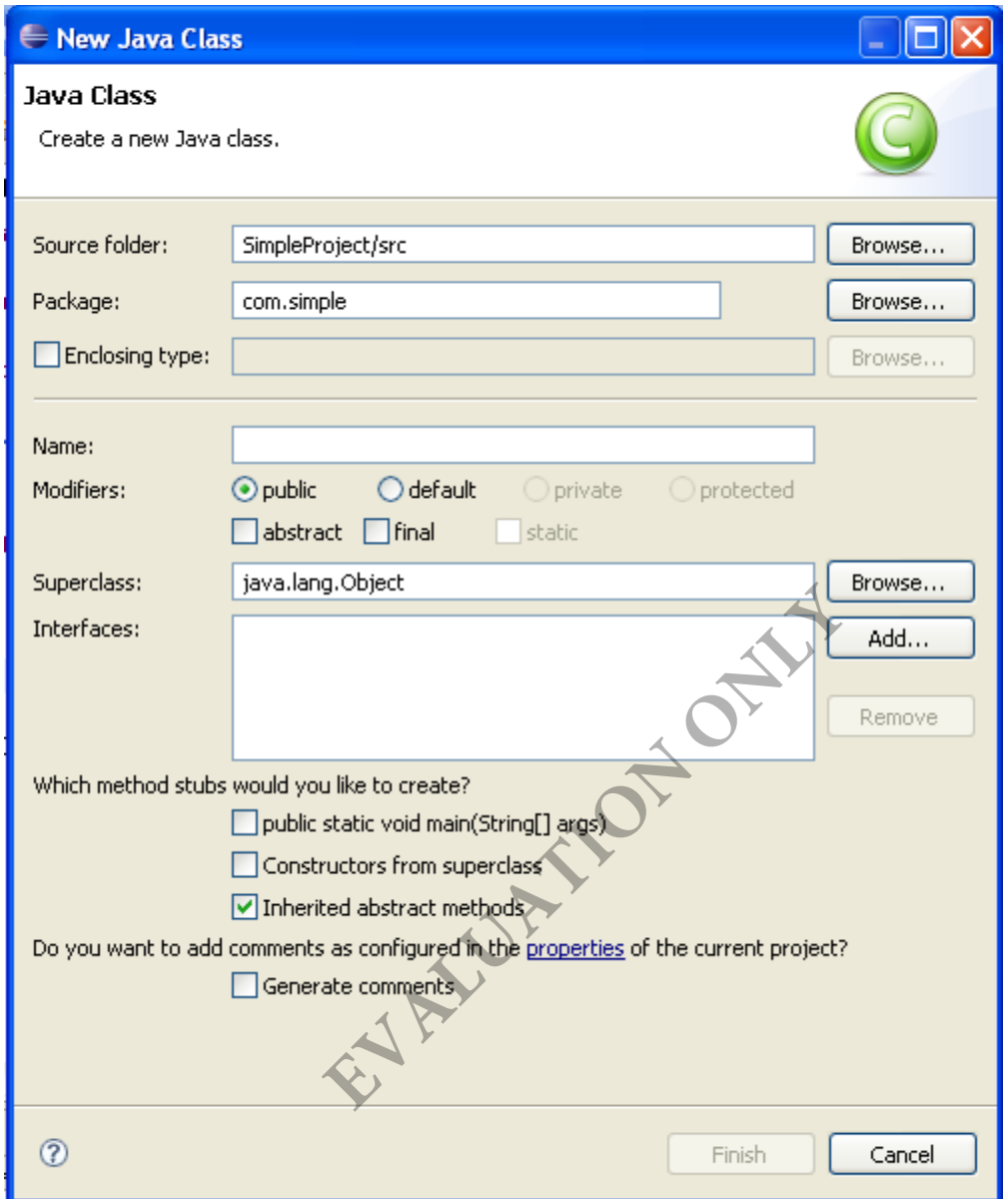
Instead of using the same HelloWorld.java class, we will create an all new class file to contain our code. We will create this class in the same project (**SimpleProject**) and package (**com.simple**) as our existing class.

__1. In the *Package Explorer,* expand the **SimpleProject > src > com.simple** package.



__2. Right click on the **com.simple** package and select **New | Class**

The *New Java Class* wizard will appear.



This should look familiar; we used the same wizard to create the HelloWorld class. As before, we should specify a name for our new class using this wizard.

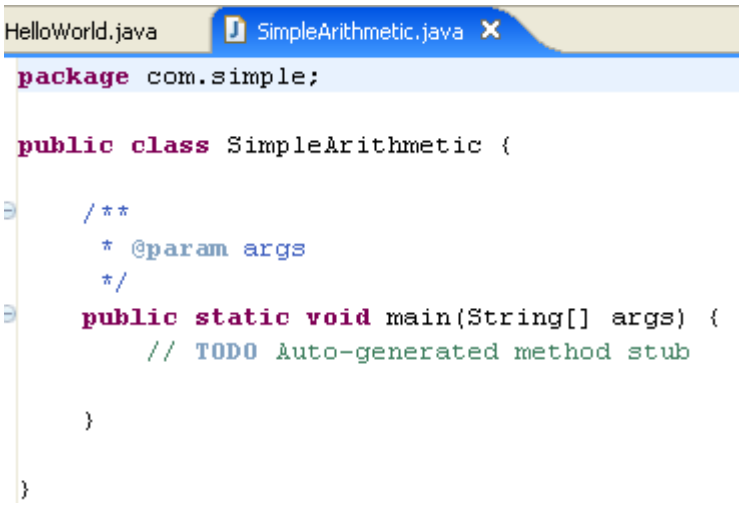
__3. Enter **SimpleArithmetic** as Name.

Notice that the *Package* field is already populated with **com.simple** saving us the trouble of typing it in. This is because we right-clicked on the **com.simple** package before creating this new class.

__4. Check the box marked *public static void main(String[] args)*. We will want Eclipse to generate this method for us.

__5. Click **Finish**.

Eclipse will generate the class and open an editor on it.



Part 2 - Edit the Class

Now we can begin coding the class. It will be very simple in design; it will first prompt the user to enter a number, followed by a second number. It will then display the results of the two numbers being added, subtracted, multiplied and divided by each other.

__1. Locate the **main** method and delete the `// TODO` comment line inside it.

__2. Let us start by prompting the user to enter two numbers. Enter the following code inside the **main** method.

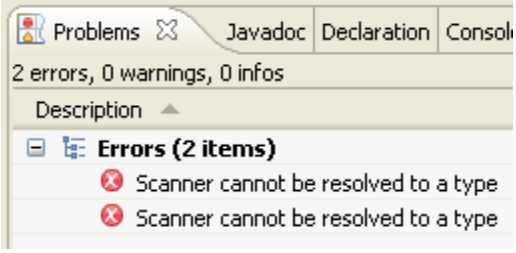
```
System.out.println("Please enter two numbers:");
```

__3. Now, set up the Scanner class (as we did in the previous lab) with the following code:

```
Scanner scan = new Scanner(System.in);
```

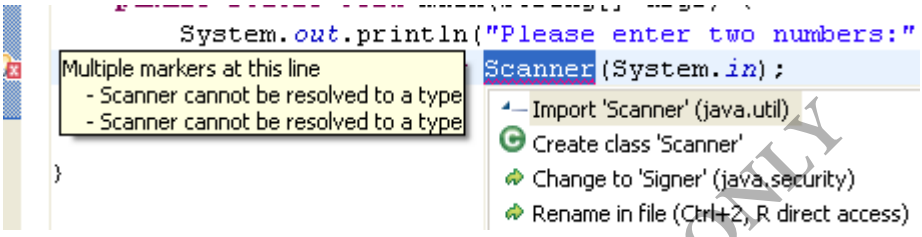
__4. Save the file.

Note that Eclipse will complain that there are errors on this line. What is the error? Look at the *Problems* view.



Eclipse does not understand what the **Scanner** class is. This is because we did not use the fully qualified name of the class (**java.util.Scanner**), nor did we import the package that the class belongs to. Fortunately, we can get Eclipse to fix this for us automatically.

__5. Click on the red X on the left margin of the line where the error is. Some pop up boxes will appear.



__6. In the white box, double-click the line **Import 'Scanner' (java.util)**.

The boxes, and errors should disappear. What happened? Take a look at the top of the source file and notice that the **import** statement has been automatically added for us by Eclipse, thus solving our problem! Eclipse is fairly good at solving these kind of **import** problems.

__7. Save the file and all the errors should go away.

__8. We can continue coding the file. Immediately following the scan initialization line, enter the following:

```
int x = scan.nextInt();
int y = scan.nextInt();
```

These two lines are quite straightforward. They declare **int** variables (**x** and **y**) and assign them to the result of **`scan.nextInt()`** which will retrieve input from the user. After these lines are executed, **x** will be the first number that was entered, and **y** will be the second number that was executed.

__9. We can now print the results of the arithmetic operations. Continue by adding the following code:

```
System.out.println("You entered " + x + " for x");
System.out.println("You entered " + y + " for y");
```

Here we simply print out what the values entered were. Note the use of concatenation to append the **int**s to the output string.

__10. Continue by adding the following code.

```
System.out.println("The sum of x and y is " + (x+y));
```

This seems like a simple concatenation, but notice the difference; we are adding an operation to the concatenation! The expression **`(x+y)`** will be evaluated first since it is in parentheses; that evaluated sum will then be concatenated to the string and **println**-ed

Note that since x and y are both **int**s, Java understands that **`(x+y)`** should invoke the *addition* operation.

__11. Continue with the subtraction and multiplication results.

```
System.out.println("The difference of x and y is " + (x-y));
System.out.println("The product of x and y is " + (x*y));
```

These work in the same way as before. (Note that clever use of the copy-and-paste mechanism of Eclipse can greatly speed up typing those lines...)

__12. Now, things become a little more complex. We have yet to handle division but there is a problem. What happens if the user has entered 0 for the value of **y**? Java will complain with a "division by 0" error. We should prevent this from ever happening and we can do that by entering the following code:

```
if(y == 0) {
   System.out.println("Cannot divide by 0.");
} else {
   System.out.println("The division of x and y is " + (x / y));
}
```


Here, we see a traditional **if-else** statement. The syntax can seem confusing at first (those pesky curly braces are everywhere), but is actually quite simple.

Immediately following the **if** keyword is the *test condition* (which is in parenthesis). In our case, we test to see if y is equal to 0. (Notice that the condition uses a **double equal** sign; == is a test for equivalence; putting a single equal sign in there would imply that we are assigning the value of y to 0!)

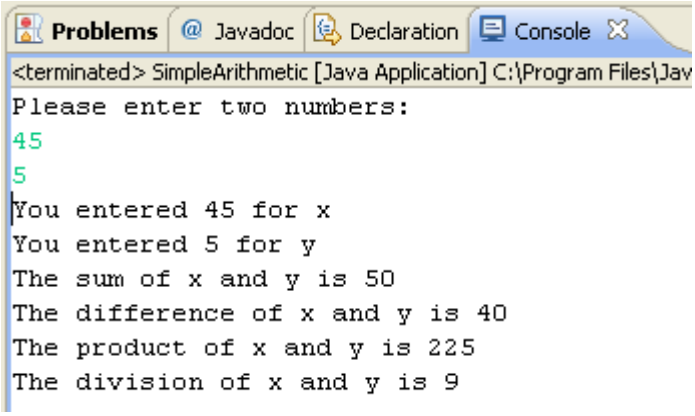
Immediately after the condition, we open a curly brace and put our *true* clause code in. This is the code that will be executed if the condition evaluates to true. In our case, we simply print out a message saying that division by 0 is not possible. This clause is open and terminated using the traditional curly braces { }

After the true clause, we use the keyword **else** to begin the *false* clause. The code contained within this set of curly braces will be executed in the event that the condition does **not** evaluate to true. In this case, if y is not equal to zero, we simply print out the corresponding division result.

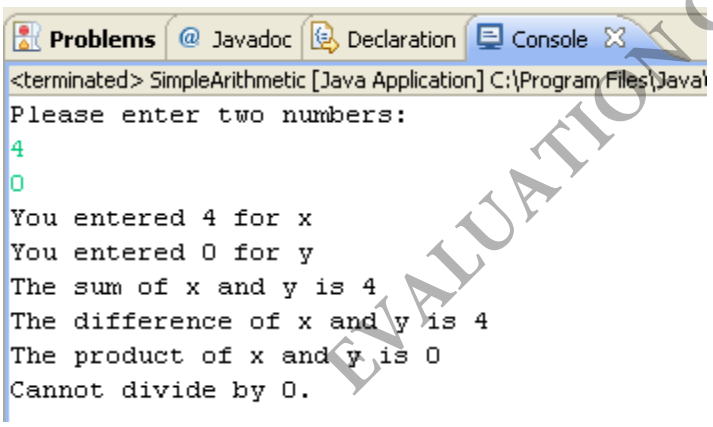
__13. We are done. Save the code and make sure there are no errors.

Part 3 - Run the Code

__1. Run the code as you have done so in previous labs. Use the console and enter two positive non-zero numbers. Make sure you get expected results! (You may need to resize the console view to see all the text at once)

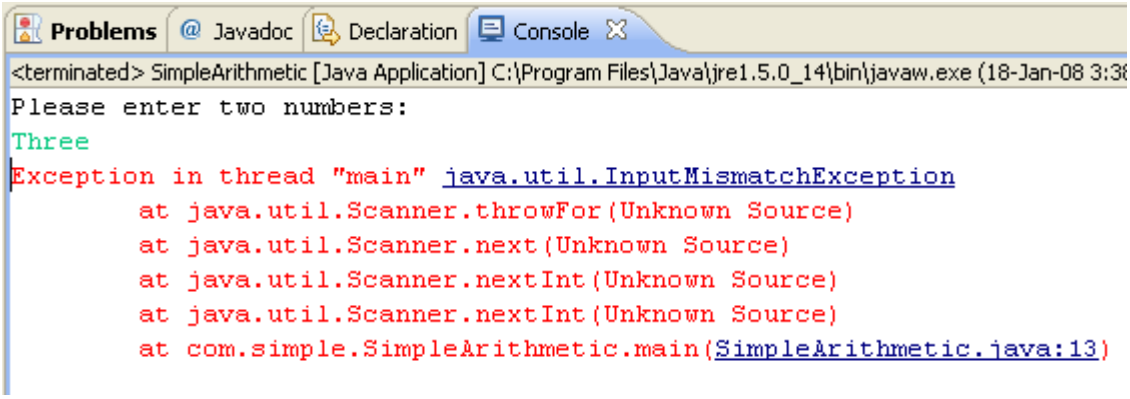


__2. Try entering a zero for the second number. Make sure the appropriate "Cannot divide by 0" message is displayed.



Part 4 - Break the Application

__1. Now, try entering non-numbers (like characters and punctuation marks). What happens?



A little on the ugly side. Java has encountered a problem and is throwing an *exception*. Because the **scan** was expecting an **int**, entering character data instead has caused the **scan** object to complain. The text in red is an exception – which is simply Java's way of telling you that something has gone wrong. Exceptions will be discussed in more detail later on.

This has happened because our code does not check for all error conditions – such as entering character data instead of numeric data. Currently, our code **does** check for one error condition: division by zero.

Knowing this, you could go back to the code and write **if** statements to check to see if **x** and **y** are indeed numeric values; doing so would be good error handling practice. Roughly sketch out (but do not code) what the **if** statements might look like if you were, in fact handling these types of errors.

Part 5 - Review

In this exercise, you wrote another Java class and saw more detail of how to use Java statements, including arithmetic operations. You also saw how Eclipse can assist with importing of appropriate packages. Finally, you saw how to use an **if** statement to control logical flow in Java code.