EECS 2030: Lab 0

(1 % of the final grade, may be done in groups of up to three students)

Motivation

This lab will allow you to practice using an Eclipse IDE, and review some basic Java features.
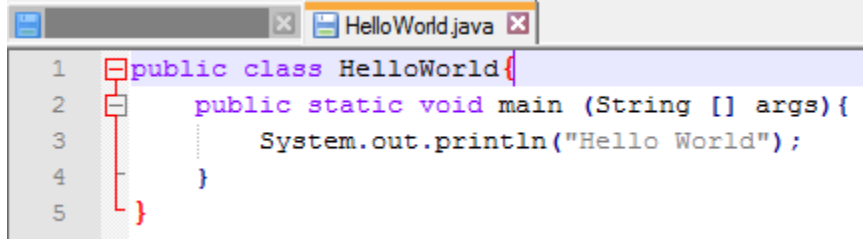
Part 1: Getting Started

Prerequisites: JDK and Eclipse

Developing Java applications requires a Java Development Kit (JDK). In this course we will also use Eclipse – an integrated development environment. Note that some JDK must be installed *before* one can install and use Eclipse. Please follow the installation instructions online[1]. As a backup, you may also use the EECS lab remote access (the link is posted on the course page).

Java: Command-Line

One way to compile and run Java code is to use a command line. For that, one writes the Java code using any plain-text editor (preferably an editor capable of at least highlighting the language's syntax, such as Visual Studio Code, Atom, Notepad++, then compiles and runs the application that was compiled to byte-codes.

Let's assume the following content is saved to a file called *HelloWorld.java*:

```
1  public class HelloWorld{
2      public static void main (String [] args){
3          System.out.println("Hello World");
4      }
5  }
```

In order to open the command line tool in the location where the file had been saved (and not have to use a *cd* command), in Windows, one can right-click on the directory containing the file while holding the Shift key:

---

[1] https://wiki.eclipse.org/Eclipse/Installation
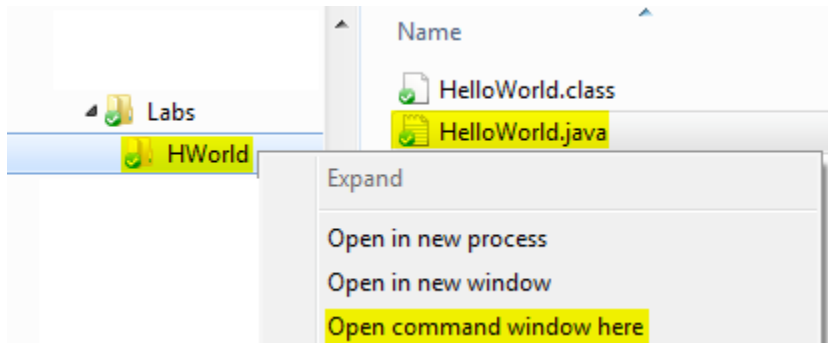
Similar context-menu choices exist for other operating systems, e.g., in Linux (CentOS):
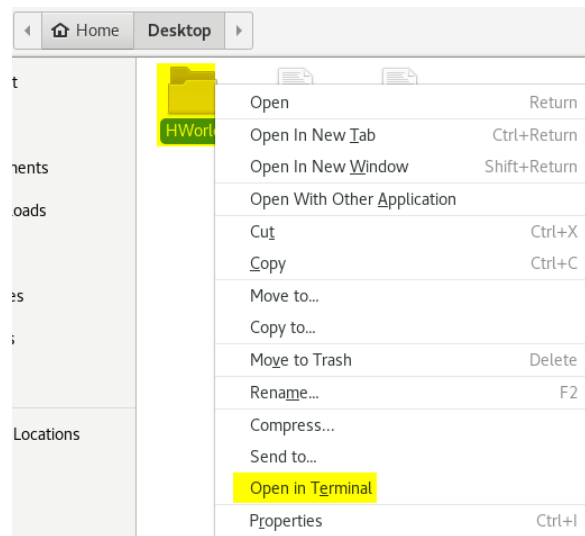
Then, the following commands compile and run the code (the full path is greyed out):
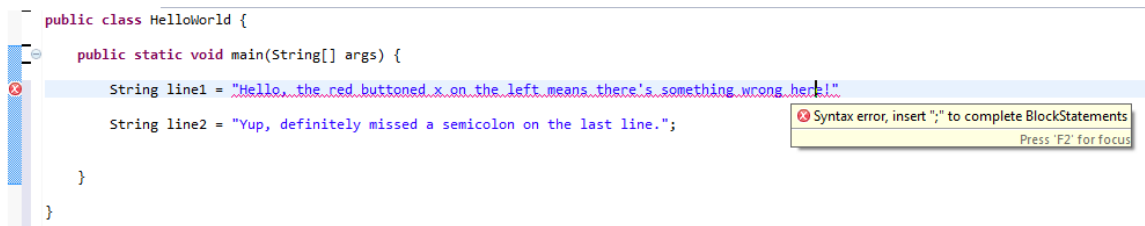
```
HWorld>javac HelloWorld.java

HWorld>java HelloWorld
Hello World

HWorld>
```

In case there are compilation or runtime errors, various error messages will be printed. Try introducing errors and see what kind of messages you receive.
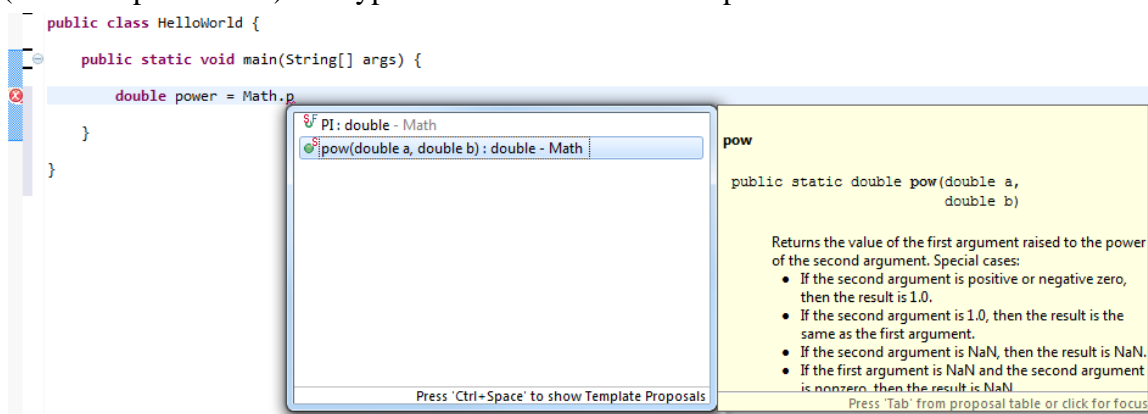
Java: Eclipse

Eclipse is an IDE (an Integrated Development Environment), meaning that it is a program *made* to provide you with all the tools you need to code, wrapped up in a nice user interface. Here are some of the top reasons for using an IDE:

Spot compiler errors quickly - if you've made some small mistake like forgetting a semicolon, or spelling a variable type wrong, the IDE will spot it quickly and let you know:

```
  public class HelloWorld {

      public static void main(String[] args) {

          String line1 = "Hello, the red buttoned x on the left means there's something wrong here!"
                                                                    ⊗ Syntax error, insert ";" to complete BlockStatements
          String line2 = "Yup, definitely missed a semicolon on the last line.";
                                                                                        Press 'F2' for focus
      }

  }
```
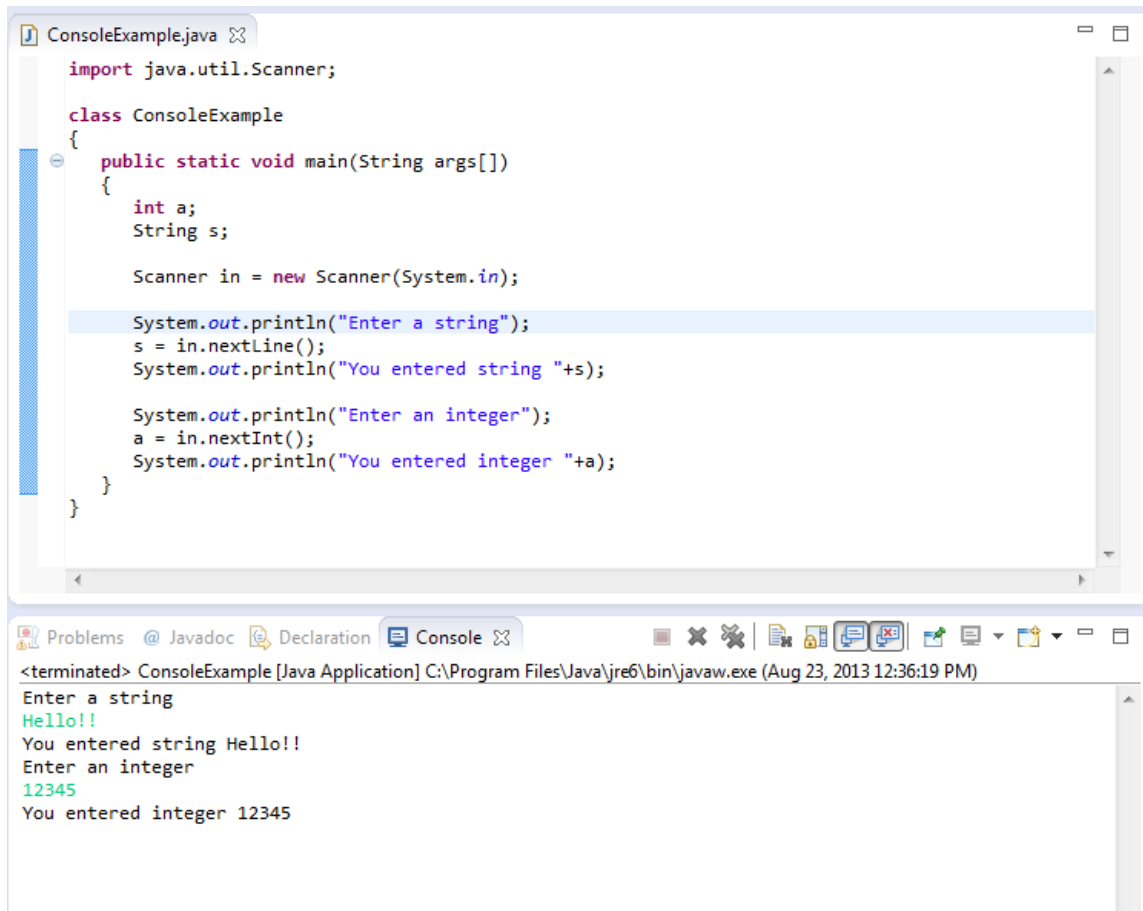
Autocomplete/Content Assist - this feature will save you time and effort by suggesting methods (and their parameters) and types. See below for an example:

```
  public class HelloWorld {

      public static void main(String[] args) {

          double power = Math.p
      }
```

```
 ⨍ PI : double - Math
 ⊙ pow(double a, double b) : double - Math
```

```
pow

public static double pow(double a,
                                double b)

Returns the value of the first argument raised to the power
of the second argument. Special cases:
  • If the second argument is positive or negative zero,
    then the result is 1.0.
  • If the second argument is 1.0, then the result is the
    same as the first argument.
  • If the second argument is NaN, then the result is NaN.
  • If the first argument is NaN and the second argument
    is nonzero, then the result is NaN.
                Press 'Tab' from proposal table or click for focus
```

Press 'Ctrl+Space' to show Template Proposals

Debug Mode - allows you to watch your variables and source code at any point in your program so you can find out exactly where your program is going wrong.

Built in console and compiler access - saves you the time from going back and forth from editor to console and vice versa.

```java
import java.util.Scanner;

class ConsoleExample
{
    public static void main(String args[])
    {
        int a;
        String s;

        Scanner in = new Scanner(System.in);

        System.out.println("Enter a string");
        s = in.nextLine();
        System.out.println("You entered string "+s);

        System.out.println("Enter an integer");
        a = in.nextInt();
        System.out.println("You entered integer "+a);
    }
}
```

Problems   @ Javadoc   Declaration   Console ⊠

<terminated> ConsoleExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Aug 23, 2013 12:36:19 PM)
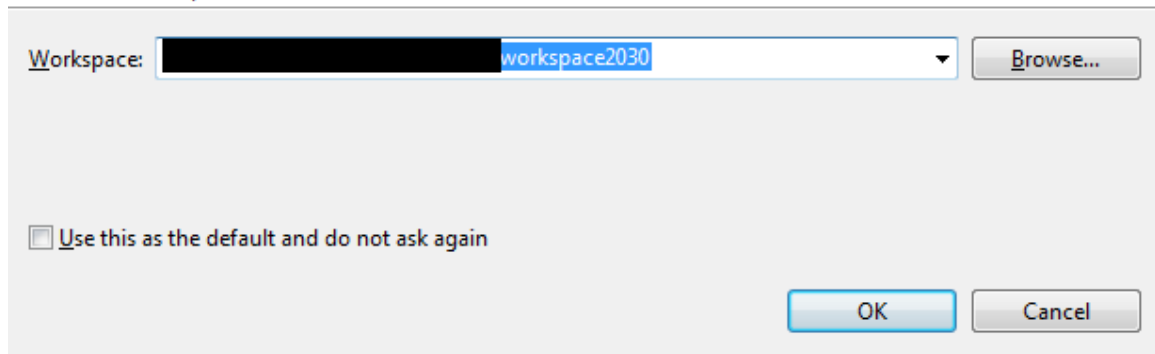
```
Enter a string
Hello!!
You entered string Hello!!
Enter an integer
12345
You entered integer 12345
```

A sample program in Eclipse:

- Start Eclipse
 The following window appears (with some variations)

you will be asked to select a workspace. This is important! This is the root directory to which all of your programs will be saved to, so make sure you know where it is. One may click *Browse...* and create a directory called **eecs2030, workspace2030 –**use any directory name that you like (including the default one **workspace**); try to avoid spaces in the directory names.

- If the following window appears:
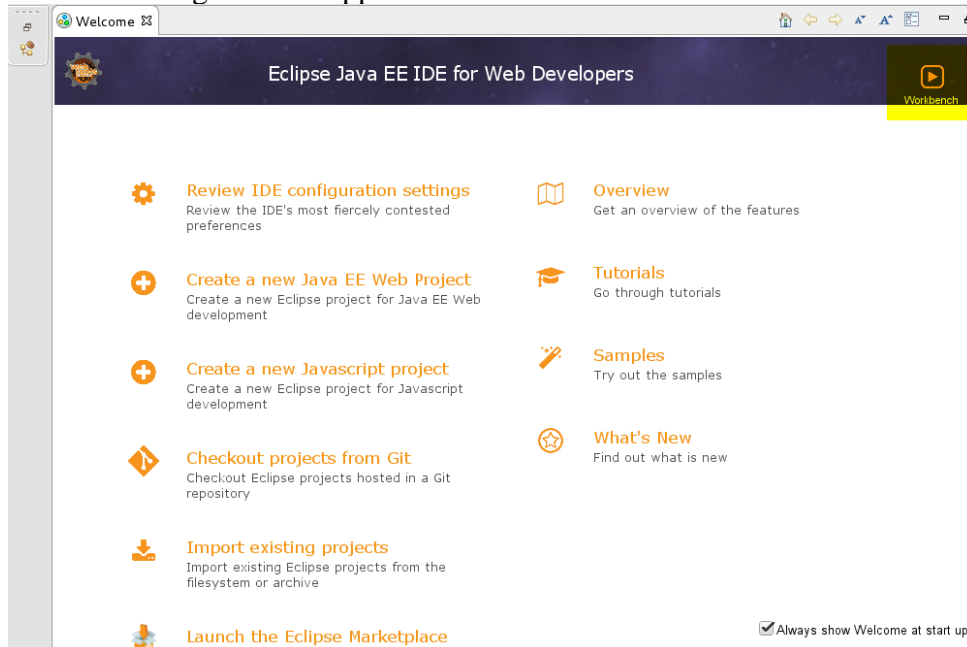
click on the orange **Workbench** button found near the top right corner.

- create a Java Project. You can do this by doing: *File -> New... -> Project...*. The project wizard looks like this - click on *Java Project*.
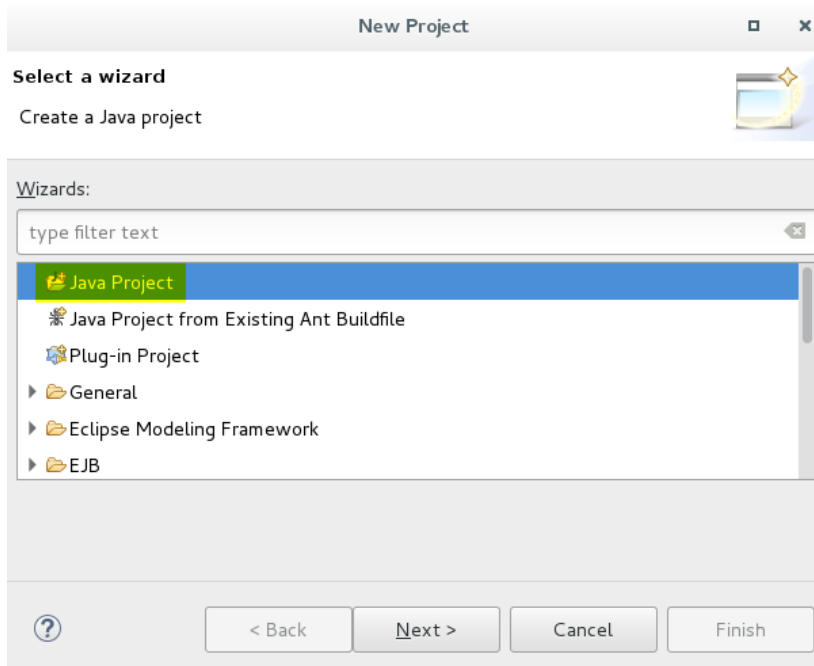
- Now, you will see a window like the one shown. Give your project a name, e.g., 'Tutorial'. You can name your project whatever you like, but *avoid using spaces in the project name* because this complicates navigating the directory structure of your project. Note that doing this now creates a directory in your previously created workspace directory. Keep the default settings and click finish.
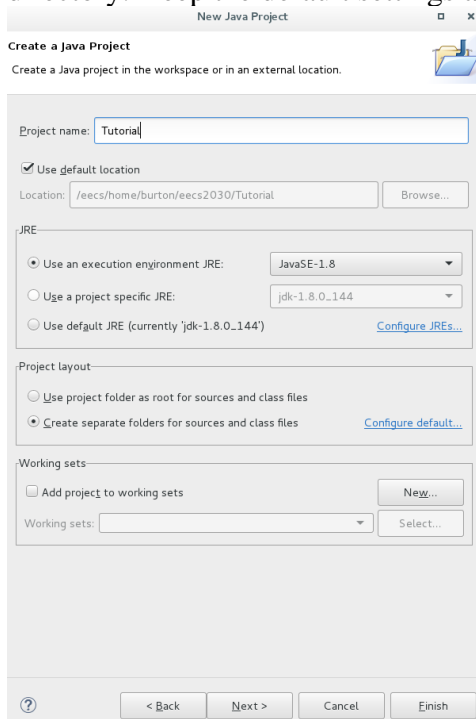
- If you see a popup window like the one showed here. Click *Yes* if that is the case. This will configure eclipse so that it enables Java specific features.


- create a package. To do this, right-click the new project you just created that will now appear in the *Package Explorer* on the left side, and then click *New -> Package*. Name it **eecs2030**. Now if you look in the Package Explorer, our **eecs2030** package is under *Tutorial -> src -> eecs2030*. *src* is the source folder. Hence, when it comes time to submit your files and you want to locate your source file, it will be in *workspace->Project->src->package->file*. So in our case, it would be: *eecs2030/Tutorial/src/eecs2030/HelloWorld.java*
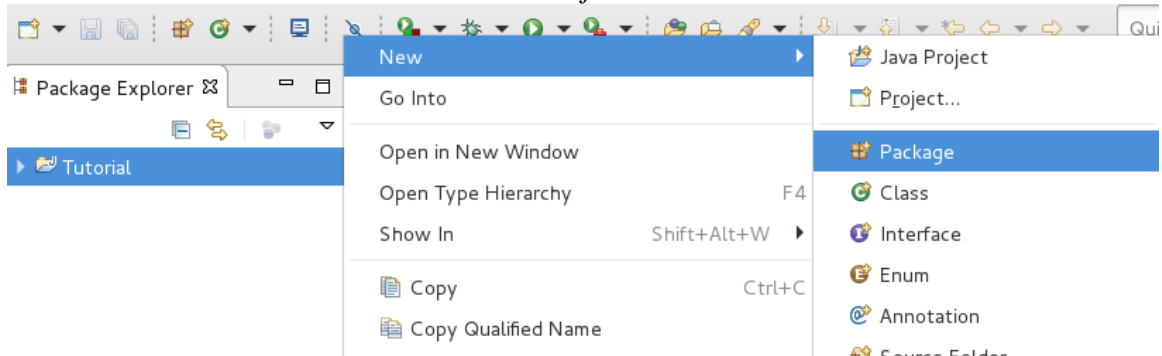
- create a HelloWorld.java. You can do this by right-clicking the package, then clicking *New*, and then *Class*. Enter the name of your class, in this case **HelloWorld**, and check the *public static void main(String[] args)* box - this will create the main method in your class for you. Our simple program does not inherit from any other program, so the other two boxes don't really matter.
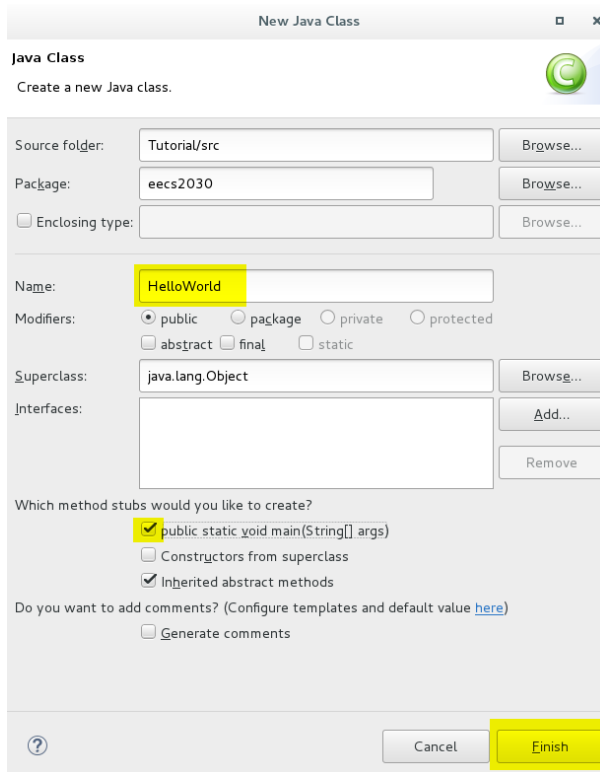
- Add **System.out.print("Hello, world.");** into the main body in the TODO section and then save your program by pressing *Ctrl+s* on your keyboard.
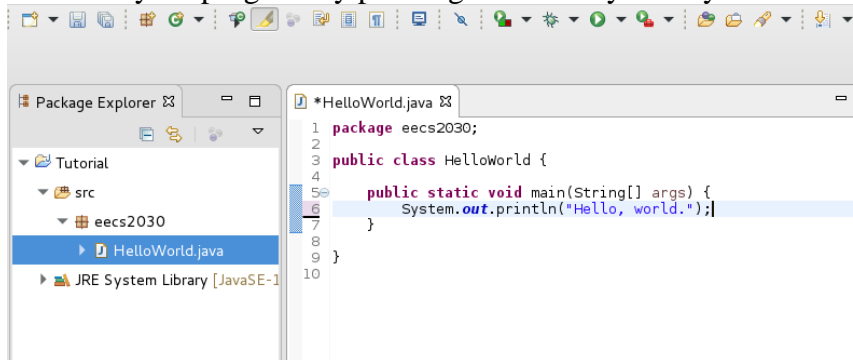
- Now you can run your program by either clicking the green run button on the toolbar, or by right-clicking your package in the package explorer and selecting *Run As...->Java Application* (find a keyboard shortcut for doing it quickly!). Note that the output appears in the console window at the bottom.

- Submit that **HelloWorld.java** file via Moodle and continue with Part 2.

Part 2: Java Review

The purpose of this lab is to review the following basic Java concepts that should have been covered in your previous courses:

- style
- using `int` and `double` values and variables
- arithmetic and the methods provided in `java.lang.Math`
- `boolean` expressions
- using objects
- `if` statements
- using `String`s
- `for` loops

This lab also introduces code testing using JUnit. This lab will be graded for style and correctness.

Style Rules

The style rules are not overly restrictive in EECS2030 but it is important that you follow them; **we use a style checker for all of your submitted lab work**. Most jobs that involve programming will require their employees to follow a rigid set of style rules.

1. Your programs should use the normal Java conventions (class names begin with an uppercase letter, variable names begin with a lowercase letter, `public static final` constants should be in all caps, etc.).

2. In general, use short but descriptive variable names. There are exceptions to this rule; for example, traditional loop variables are often called i, j, k, etc.

Avoid very long names; they are hard to read, take up too much screen space, and are easy to mistype.

3. Use a consistent indentation size. Beware of the TAB vs. SPACE problem: Tabs have no fixed size; one editor might interpret a tab to be 4 spaces and another might use 8 spaces. If you mix tabs and spaces, you will have indenting errors when your code is viewed in different editors.

4. Use a *consistent* brace style:

```
// left aligned braces

class X
{
  public void someMethod()
  {
    // ...
  }
```

```
  public void anotherMethod()
  {
    for (int i = 0; i < 1; i++)
    {
      // ...
    }
  }
}
```

or

```
// ragged braces

class X {
  public void someMethod() {
    // ...
  }

  public void anotherMethod() {
    for (int i = 0; i < 1; i++) {
      // ...
    }
  }
}
```

5. Insert a space around operators (except the period/dot ".").

The following

```
    // some code somewhere
    boolean isBetween = (x > MIN_VALUE) && (x > MAX_VALUE);
    int someValue = x + y * z;
```

is much easier to read than this

```
    // AVOID DOING THIS
    // some code somewhere
    boolean isBetween=(x>MIN_VALUE)&&(x>MAX_VALUE);
    int someValue=x+y*z;
```

Note that in HTML the convention is the opposite (no spaces around `attribute="value"`)

6. Avoid using "magic numbers". A magic number is a number that appears in a program in place of a named constant. For example, consider the following code:

```
int n = 7 * 24;
```

What do the numbers `7` and `24` mean? Compare the code above to the following:

```
final int DAYS_PER_WEEK = 7;
```

```
final int HOURS_PER_DAY = 24;
int n = DAYS_PER_WEEK * HOURS_PER_DAY;
```

In the second example, the meaning of 7 and 24 is now clear (better yet would be to also rename n).

Not all numbers are magic numbers. You can usually use the values 0, 1, and 2 without creating a named constant. If you ever find yourself doing something like:

```
final int TEN = 10;
```

then you are probably better off using 10 and explaining its meaning in a comment.

7. A good IDE (integrated development environment) such as Eclipse will correct many style errors for you. In eclipse, you can select the code that you want to format, right click to bring up a context menu, and choose *Source -> Format* to automatically format your code. You can also correct indentation only.

## *Getting started*

These instructions assume that you have completed the previous parts of this lab, and have Eclipse running. In this lab, you will import an existing project rather than starting everything from scratch. In the eclipse *File* menu, choose the *Import...* menu item.

In the *Import* dialog box that appears, choose the *Existing Projects into Workspace* item and click *Next*:
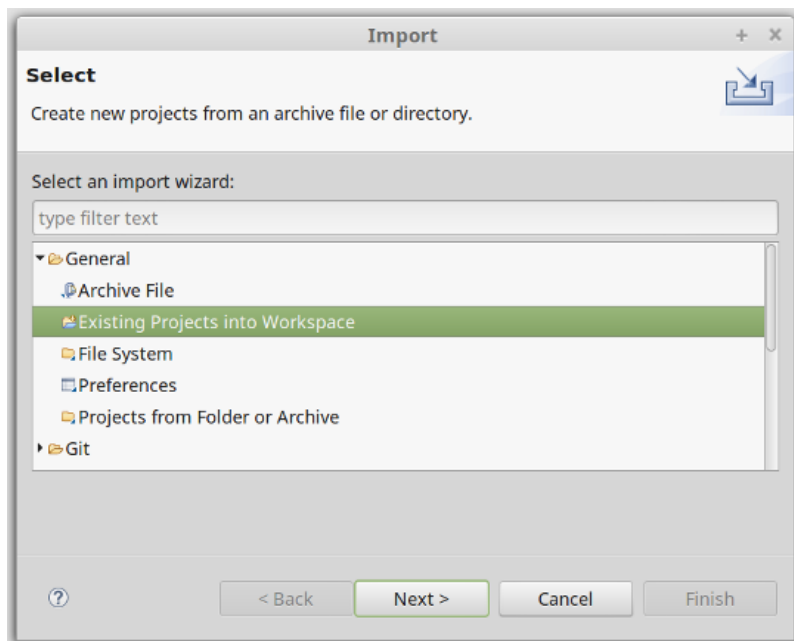
Use the included zip file. Click on the *Select archive file* radio button. Click on the *Browse...* button and select the file that you just downloaded. Click the *Finish* button to import the project. Note that there is no need to unpack the zip file.

On the left-hand side of the eclipse window, you will see a tab labelled *Package Explorer*. Use the small triangles to expand the *lab1* contents, then the *src* contents, and finally the *eecs2030.lab1* contents. Double-click on `Lab1.java` and `TestLab1.java` to open these files in the editor:
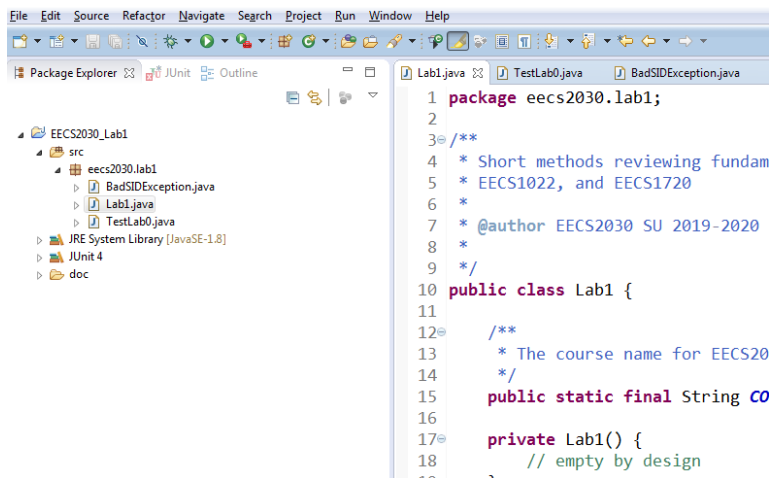
Click on the `Lab1.java` tab. `Lab1.java` is the Java source code file that you need to edit to complete this lab. It contains several methods that you should be able to complete if you have mastered the material from your previous Java programming course or courses.

Find the method with the header `public static int minInt()`. Reading the Javadoc comment preceding the method, tells us that the method should return the smallest value that can be represented by the type `int`. Examining the body of the method in eclipse, we see that the method is implemented like so:

```
public static int minInt() {
    return 0;
}
```

which is clearly incorrect. DON'T FIX THE METHOD YET; continue following the lab document.

`TestLab1.java` is a test class that contains unit tests for all of the methods that you will implement in this lab. You will learn more about unit tests in your next lecture. For now, all you need to know is that you can use the test class to check for errors in the methods in `Lab1.java`.

Click on the `TestLab1.java` tab in the editor window to view the contents of `TestLab1.java`. Run the test class by pressing the green run button indicated by the red arrow in the figure below:
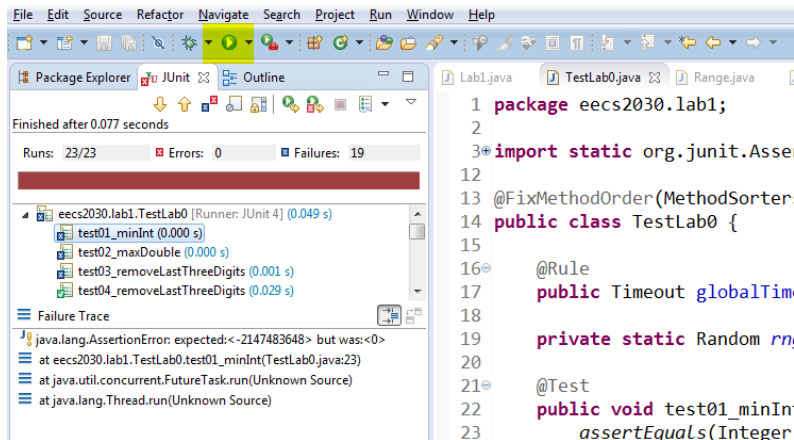
The results of running the tests are shown to you in the *JUnit* tab located on the left-hand side of the eclipse window (see figure above). Notice that all of the tests have a blue x beside them; the blue x indicates that the test has failed. In the *Failure Trace* panel, some diagnostic information is shown to you. For the `test01_minInt` test, the diagnostic information is indicating that the test expected a value of `-2147483648` but received a value of `0`. It seems like there is something wrong with our implementation of the `minInt` method.

Click on the `Lab1.java` tab in the editor window to view the contents of `Lab1.java`. Scroll down to the `minInt` method (the first method in the class). Edit the return value of the method so that it returns the correct value as shown below:
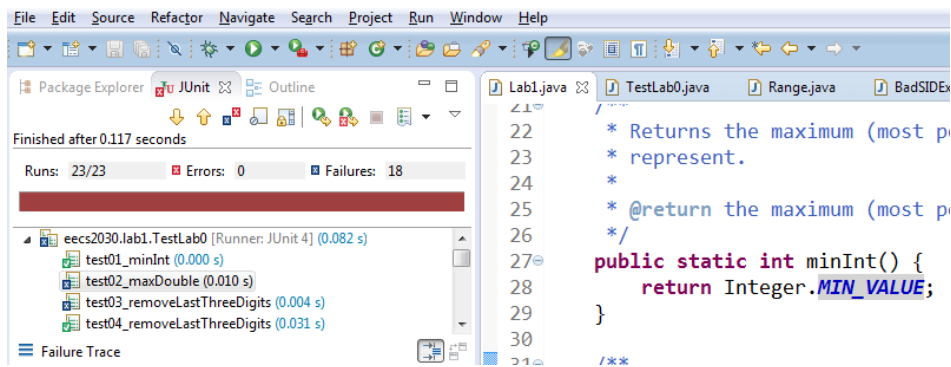
Save the edited file (use the *File* menu or type *Ctrl-s*). When you re-run the test class; you should see that the test `test01_minInt` now has a green check mark beside it indicating that the test has passed. The remaining tests are still failing. In the remainder of this lab, you will use the test class to help you fix the remaining methods in the `Lab1` class. Follow the remainder of the lab to complete the exercises below. Most of the methods can be completed with a single line of code.

13

Primitive types

In Java, all values have a *type*. A type defines a set of values and the operations that can be performed using those values.

Java's *primitive types* are those types that are predefined by the Java language and are named by a reserved keyword. The primitive types are all numeric types and one type representing true/false values.

int

The `int` type represents integer values in the range -2^{31} to $(2^{31} - 1)$. Java will interpret any number not having a decimal as being an `int` value.

Constant values important to the `int` type can be found in the class `java.lang.Integer`.

double

The `double` type represents real values in the approximate range of -1.7×10^{308} to 1.7×10^{308}. Java will interpret any number having a decimal as being a `double` value.

Exercise 1

Complete the method `maxDouble()`. Refer to the method's description in the JavaDoc in-code comments.

Run the JUnit tester after you complete each method to check your work.

Exercise 2

Complete the methods `removeLastThreeDigits(int n)` and `lastThreeDigits(int n)`.

Note that in `removeLastThreeDigits (int n)` and `lastThreeDigits (int n)` you do not need to create named constants for any magic numbers you might need.

`removeLastThreeDigits (int n)` can be completed using integer division. `lastThreeDigits (int n)` can be completed using integer remainder. You do not (and should not) need to use an `if` statement in either method.

Run the JUnit tester after you complete each method to check your work.

Exercise 3

Complete the method `avg(int a, int b, int c)`. Be aware of the possibility of incorrectly using integer division instead of floating-point division.

Run the JUnit tester after you complete each method to check your work.

Exercise 4

Refer to the following link: https://en.wikipedia.org/wiki/Pressure_altitude if you want more detail.

Pressure altitude is the altitude in the International Standard Atmosphere (ISA) with the same atmospheric pressure as that of the part of the atmosphere in question.

The National Oceanic and Atmospheric Administration (NOAA) published the following formula for directly converting atmospheric pressure in millibars to pressure altitude in feet:

$$145366.45 \left[1 - \left(\frac{\text{Station pressure in millibars}}{1013.25} \right)^{0.190284} \right]$$

Complete the method `pressureAltitude(double pressure)` using named constants to represent the magic numbers in the formula.

Exercise 5

Complete the methods `isEven(int x)` and `isUnitVector(double x, double y)`. Recall that the unit vector is a vector with a magnitude of 1, so that $x^2 + y^2 = 1$.

Run the JUnit tester after you complete each method to check your work.

Exercise 6

Complete the method `enrolStudent(int studentNumber)`. Note that the method can throw a `BadSIDException` which is an exception class that has been included for you in the lab project.

All the method should do is return true. However, the method should throw an exception if the number is not a 9-digit number (at York there are other factors that would allow for checking the number validity; we use a simple hypothetical rule here).

Run the JUnit tester after you complete each method to check your work.

Exercise 7

Complete the method `getCourseName()`.

To complete this method, you should first find the `Lab1` class constant that contains the course name. You should then simply return the constant to complete the method; in other words, you should *not* duplicate string containing the course name in your method.

Run the JUnit tester after you complete each method to check your work.

Exercise 8

Complete the method `middleChar(String s).` See the code comments for the method to see the more formal definition of the middle character of a string.

Run the JUnit tester after you complete each method to check your work.

Exercise 9

Complete the method `alternatingCaps(String s).` The API for the method defines what the method should return and contains several examples.

To convert characters to lowercase, use the method `Character.toLowerCase.` To convert characters to uppercase, use the method `Character.toUpperCase.` The API for the `Character` class can be found here.

Run the JUnit tester after you complete each method to check your work.

If you have questions, don't hesitate to post your questions on the course forum on Moodle, or contact the instructor directly (andriyp@cse.yorku.ca).

Grading

The assignment will be graded using *the Common Grading Scheme for Undergraduate Faculties*[2]. We look at whether the code passes the unit tests, and whether it conforms to the code style rules.

Submission

Find the `Lab1.java` file in your project and submit it electronically via Moodle.

If working in a group, make only one submission and include a **group.txt** file containing the names and the student numbers of the group members. The deadline is firm. Contact the instructor *in advance* if you cannot meet the deadline explaining your circumstances.

Academic Honesty

Direct collaboration (e.g., sharing your work results across groups) is not allowed (plagiarism detection software may be employed). However, you're allowed to discuss the assignment requirements, approaches you take, etc. Also, make sure to state any sources you use (online sources – including old solutions, books, etc.). Although using outside sources is allowed – with proper citing, if the amount of non-original work is excessive, your grade may be reduced.

---

[2] https://secretariat-policies.info.yorku.ca/policies/common-grading-scheme-for-undergraduate-faculties/