**CSCI 201 Lab 2**

**Inheritance**

“*Programming is a skill best acquired by practice and example rather than from books.” – Alan Turing*

Introduction

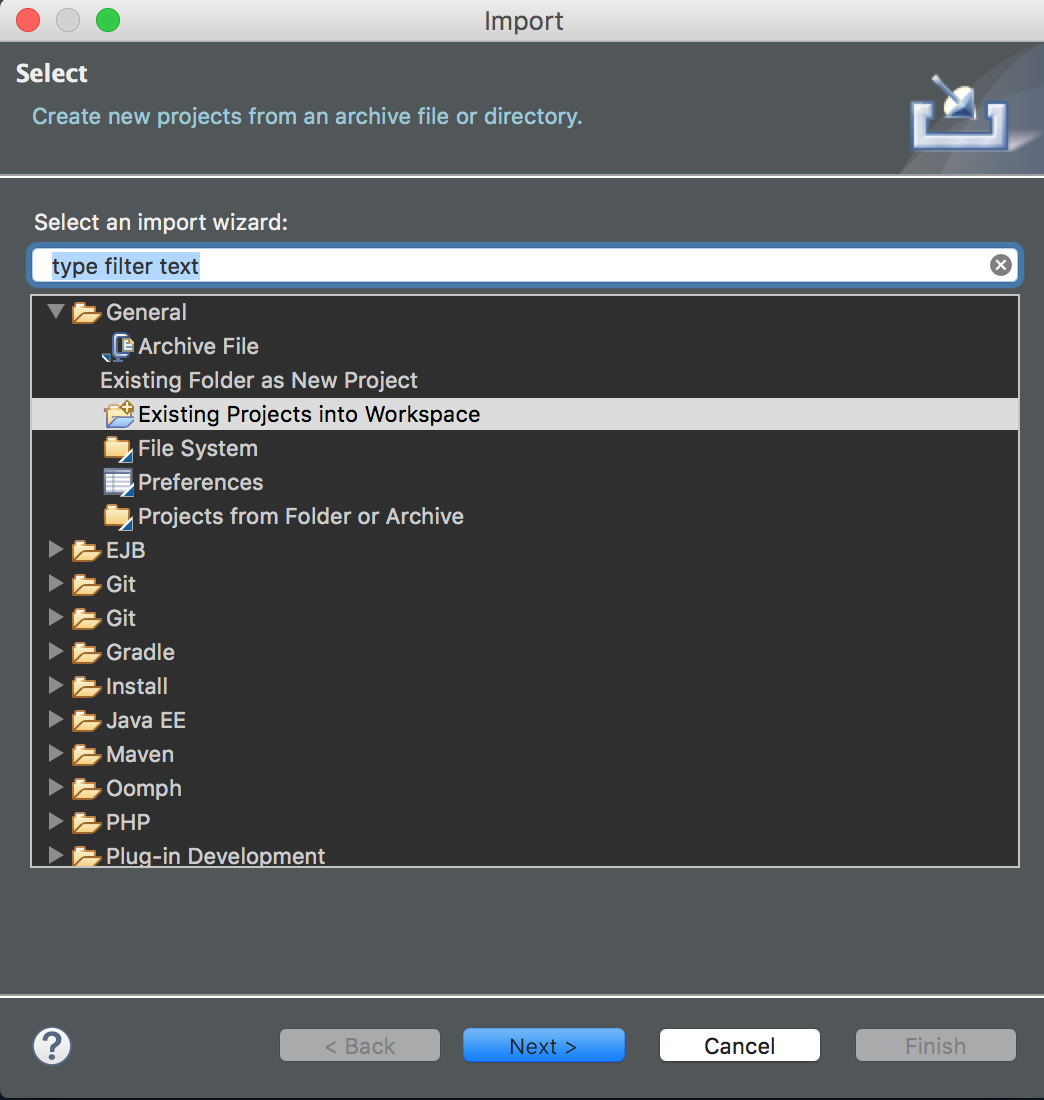
The Factory code is the basis of all labs this semester. It runs a factory simulation complete with workers, products, resources, and more. The logic is written in Java and displayed with a web frontend, both of which you will add to. It is recommended that you take the time to look through all the code and make sure you understand how it works, as you will be modifying many of the classes within.

**Reference the Common Problems Guide and Google Chrome DevTools documents frequently as many of your environment-related issues will be solved there. Keep in mind they will be updated as the semester continues.**

Part 1 – Importing a Java Project

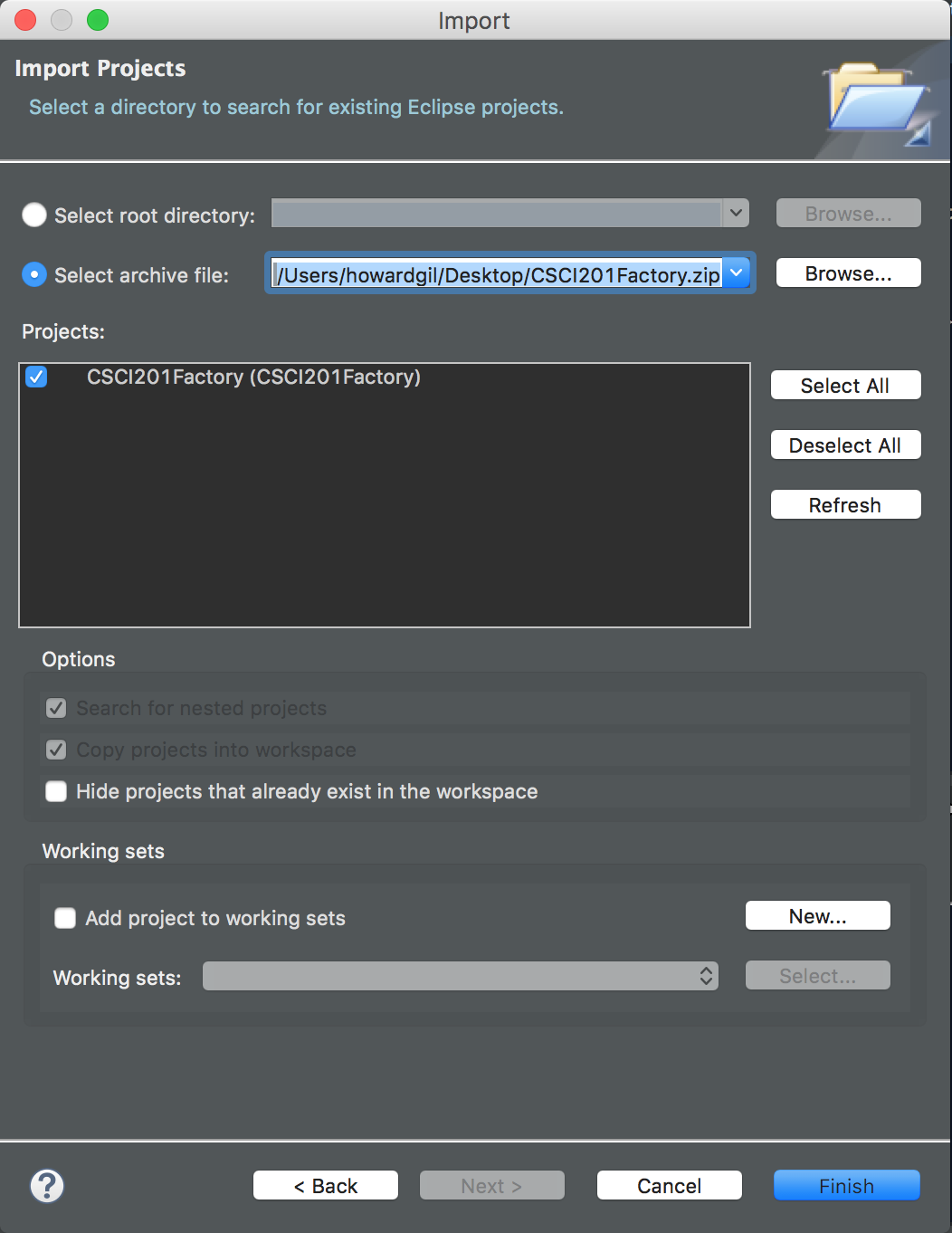
Let’s import the Factory code.

1. Download the project from the course website.
2. Go to File > Import > General > Existing Projects into Workspace

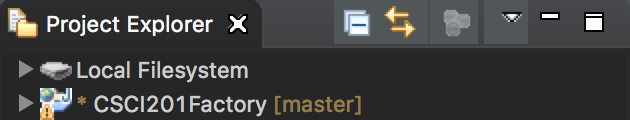


1. Select ‘Select archive file:’, and the Press ‘Browse…’ Select the zip file that you downloaded.

* See the Common Problems Guide for problems you may encounter here

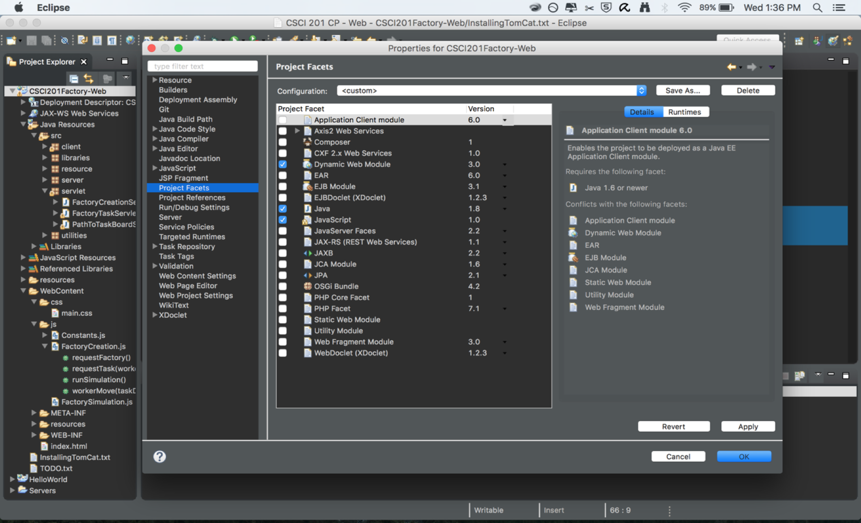


1. Press Finish. You should now see your project in your workspace



**Note:** if the project has a red “x”, refer to “Project not building…” in the Common Problems Guide

1. Right-click the project
   1. Properties > Project Facets
   2. Make sure these are selected:
      1. Dynamic Web Module
      2. Java
      3. JavaScript



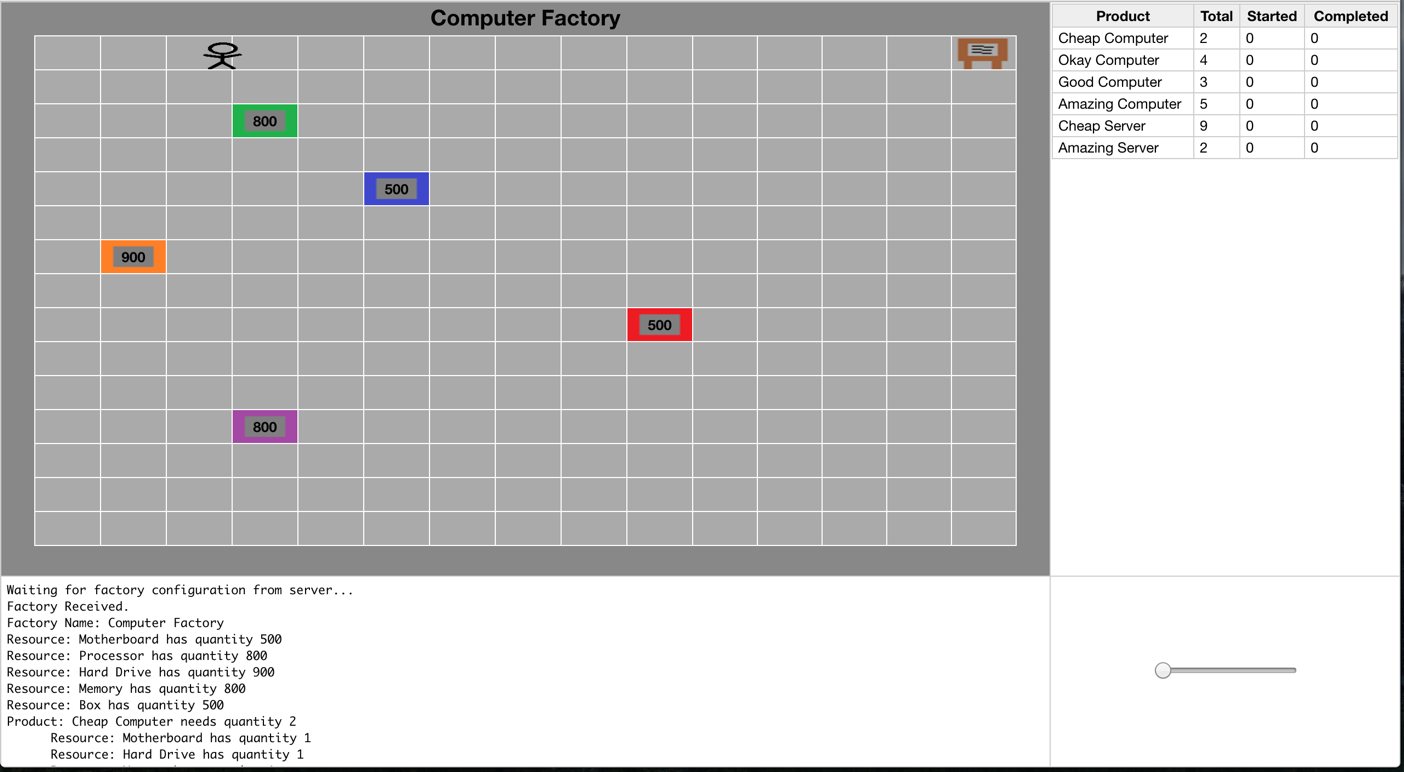
* 1. Apply > OK

1. Right click the index.html file in the WebContent folder
   1. Select: Run as… > Run on Server

**Reminder:** you should be in Java EE perspective

**Window > Perspective > Open Perspective > Other > Java EE**

The simulation will run in Eclipse, and if you go to localhost:8080/CSCI201Factory/index.html in your browser you should also see it.



You now have everything set up on your computer for all of the labs and assignments for the semester!

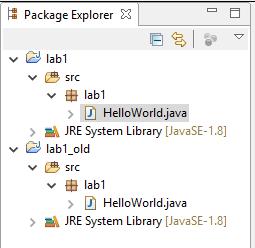
Part 2 – Building off a project

In order to build off of previous labs or assignments, you will first need to **duplicate** your project. In the future, when you (inevitably) request a regrade from the CP who graded you, you will need to duplicate your project. This is because when an in-person regrade is done, the CP will have you download a fresh copy of your submitted code off Blackboard and then import it into Eclipse before demoing it to them.

First, export your Lab1 code into a .zip file. Keep it somewhere safe in your computer.

Then, you will need to **rename** your original project, otherwise Eclipse will not allow you to import your project. You can do this by right clicking your project's name and then clicking “Refactor” and then “Rename.” Rename your project to something like “Lab1\_Expansion”.

Next follow the same steps as you did to import the Factory code except select the Lab1 zip file that you exported earlier.



Finally, do the next exercise IN YOUR Lab1\_Expansion PROJECT, NOT THE ORIGINAL Lab1 PROJECT.

Part 3 – Printing to the console

In C++, you used something like **cin** from the Standard Template Library (STL) to read in something from the console. In Java, there is no STL, so you will be using a Java object to do basic input/output (I/O). The **Scanner** behaves a lot like a basic buffer reader, such as cin, however, there are specific ways to use the Scanner that differs from cin.

In order to do so, you first need to **declare a new Scanner object.** This means that you will need to import java.util.Scanner; as part of your project's imports.

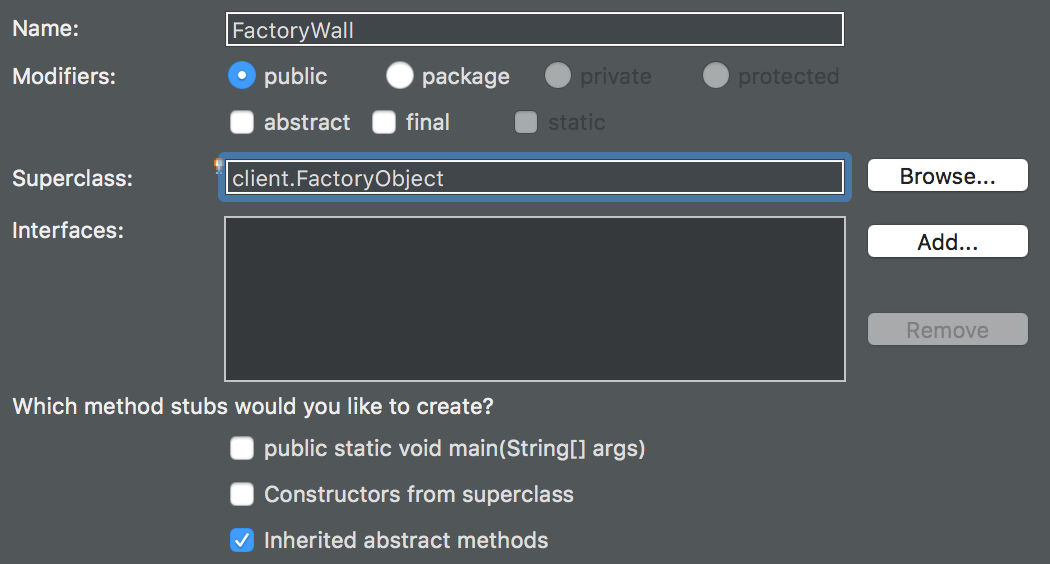
Once you declare a new Scanner object, you can start reading in items from the Java console. Your job is to make a simple program that will take in your username, and then print out your username with a nice message such as “Hi <username>!”

Part 4 – FactoryWall Creation

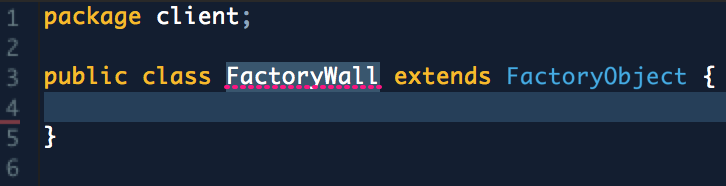
**Do the following in a new project – lab2 – that builds off CSCI201Factory. So export, rename and import CSCI201Factory.**

Let’s create a new type of FactoryObject, the FactoryWall. You will need to make sure your FactoryWalls are drawn in the factory, and that the FactoryWorkers navigate around them.

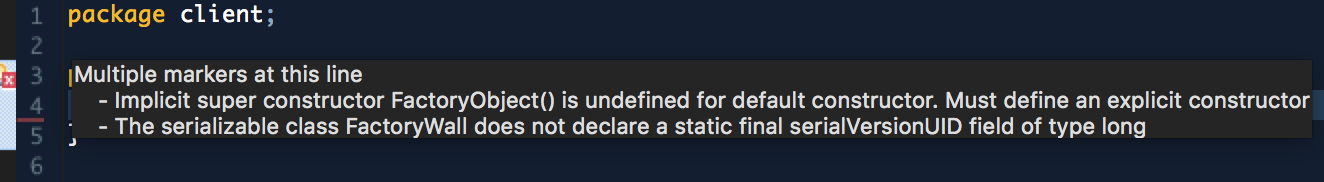
Start by creating a new Java class named FactoryWall that extends our client.FactoryObject class. Note that “client” is the package and “FactoryObject” is the name of the class.



Your class should look like this.



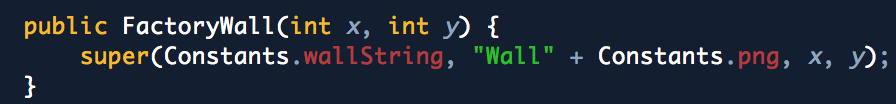
You will notice there is an error and a warning.



The error is because we have to implement one of FactoryObject’s two constructors. Both constructors take a name and image parameter, but we will implement the one that also takes in an x and y location. First, though, go to the client package’s Constants file and add a wallString constant. This is good practice, as later we may want to change the name from “wall” to, say, “panel,” and don’t want to go through the whole code and update all instances of “wall.”

../../../../Screen%20Shot%202017-01-16%20at%2010.45.58%20AM.png

Now implement super(…), which calls the superclass’s constructor.

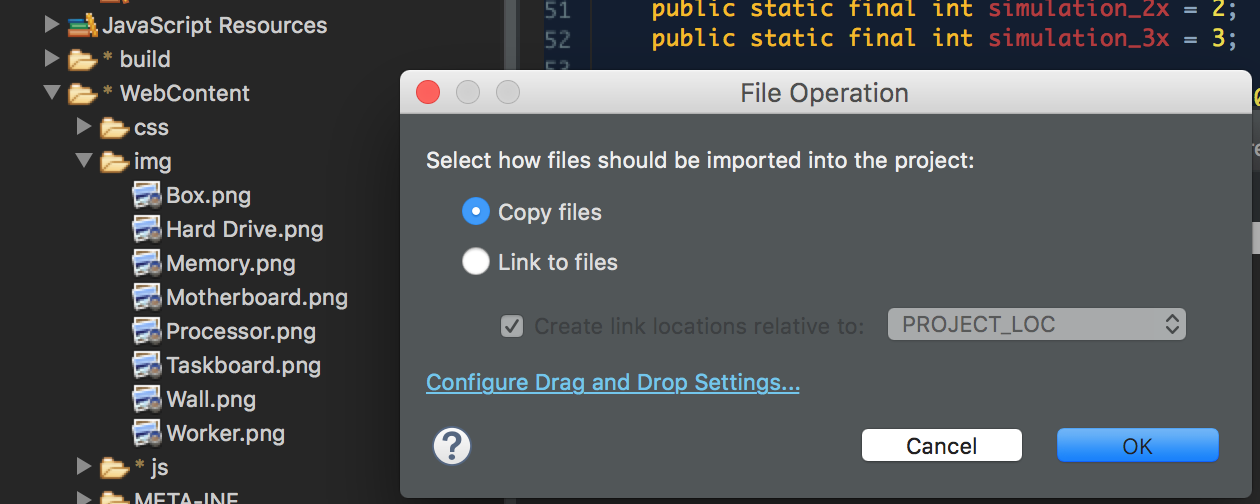


Now, the warning is because FactoryObject is serializable (you will learn about this, but basically this means a FactoryObject object can be packaged up and sent as data). Do the recommended fix and give this class a serialVersionUID.

Screen%20Shot%202017-01-14%20at%205.39.47%20PM.png

Now we want to add an image for our wall.

You may use the Wall.png provided, or use any other image. To add an image, simply click and drag your\_image.image\_extension to the WebContent/img folder to add it to the project

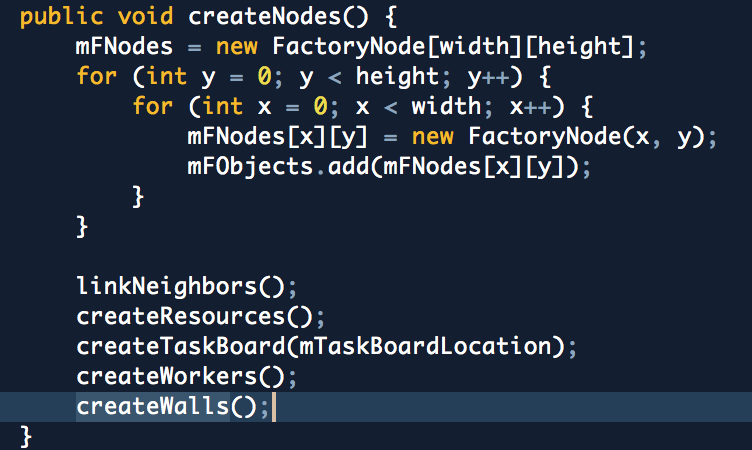


Our FactoryWall class is now fully implemented! Let’s add some walls into the factory simulation.

Part 5 – Adding FactoryWalls to the Factory

Let’s take a look at Factory.java.

The second constructor calls createNodes(), which instantiates a grid of FactoryNodes, links them together and calls on other functions to create resources, the task board, and workers. This is where we want to add another function call to createWalls(), which we will write.



createWalls() will:

* instantiate new FactoryWall objects, with locations we pass in

*FactoryWall factoryWall = new FactoryWall(7, i);*

* Add them to the Factory’s list of FactoryObjects, mFObjects

*mFObjects.add(factoryWall);*

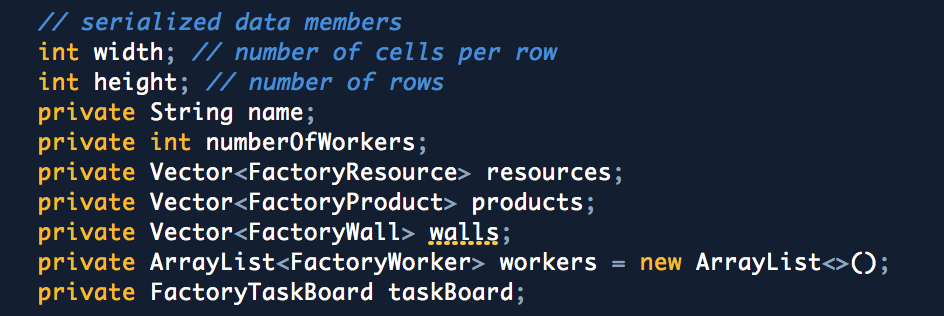
* Add them to the Factory’s map of FactoryNodes, mFNodeMap

*mFNodes[factoryWall.getX()][factoryWall.getY()].setObject(factoryWall);*

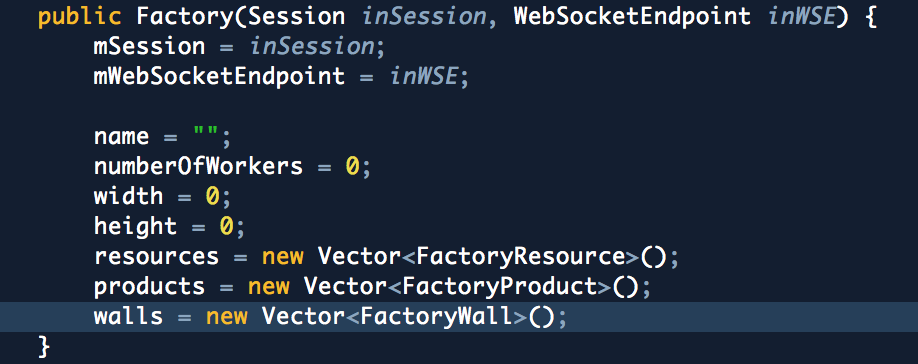
* Add them to our vector of FactoryWalls, which is passed as data when a Factory object is sent to the web client. This gives the web client the information it needs to draw the walls on its end.

*walls.add(factoryWall);*

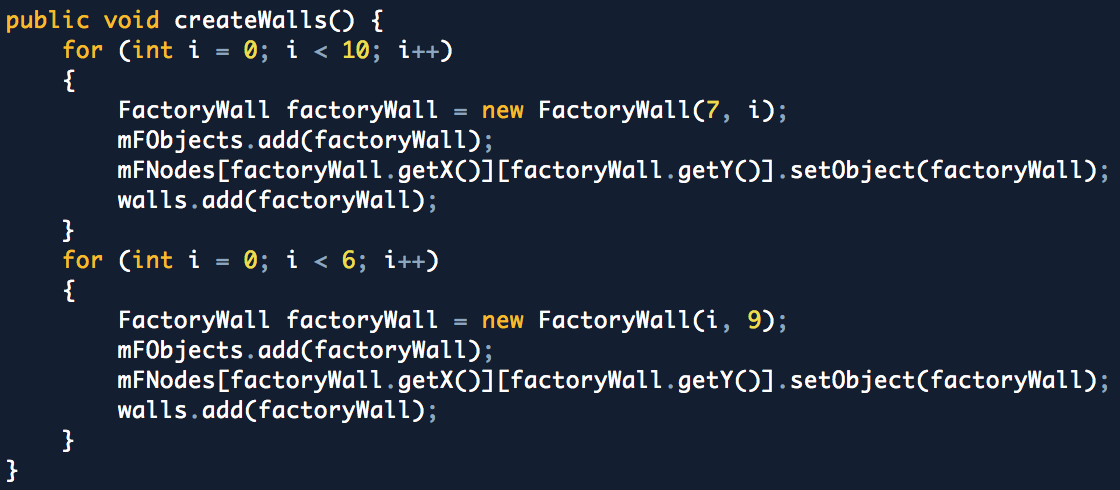
First, we must create this vector of FactoryWalls.



Then instantiate it.

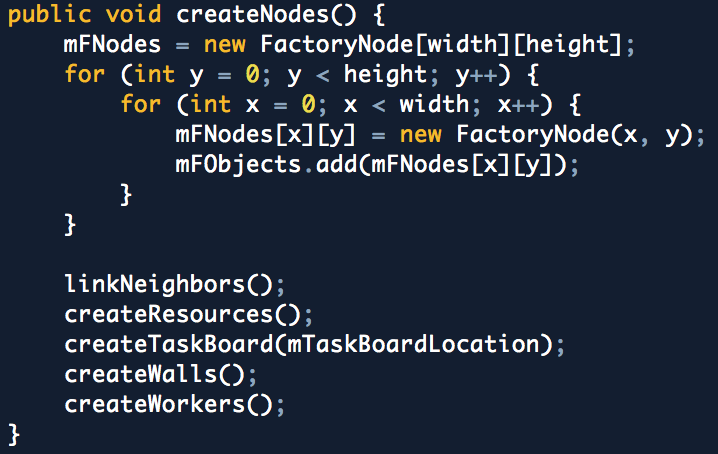


Now write createWalls() and have it loop to make two rows of walls.



*Note: We are hard-coding the walls into the Factory, but walls could just as well be specified from the configuration file, as we will do in the next assignment.*

Be sure to call createWalls() in createNodes(). We place it before createWorkers() so that the workers are aware of the walls before they start moving, else they would go through them when the program starts.

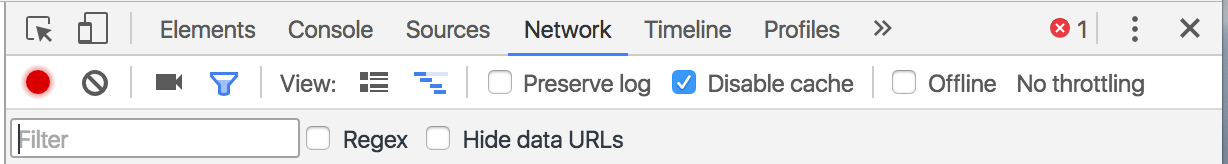


Now we have our walls in our factory simulation! If you run the project now (index.html > Run As… > Run on Server), you will see the workers going around an invisible set of walls (we still have to draw them).

Part 7 – Drawing the walls

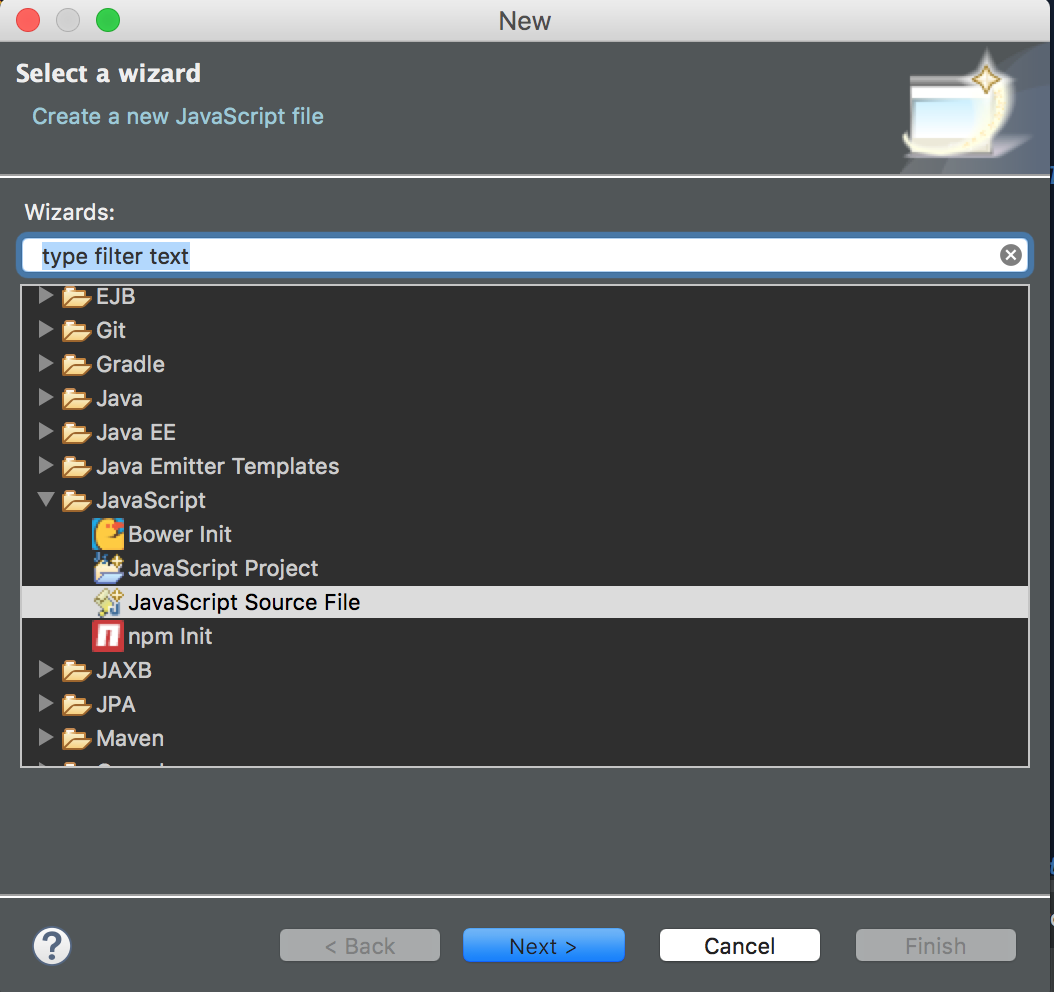
Though our FactoryWalls exist in our factory simulation, we haven’t written anything to draw them yet because this happens on the web side. This will require some JavaScript, which you will learn soon. For now just copy the code as you see it.

**First**, though, open Google Chrome > right-click > Inspect > Network > Disable cache. This will keep your browser up-to-date with all changes in your code. See the Google Chrome DevTools document for more information.

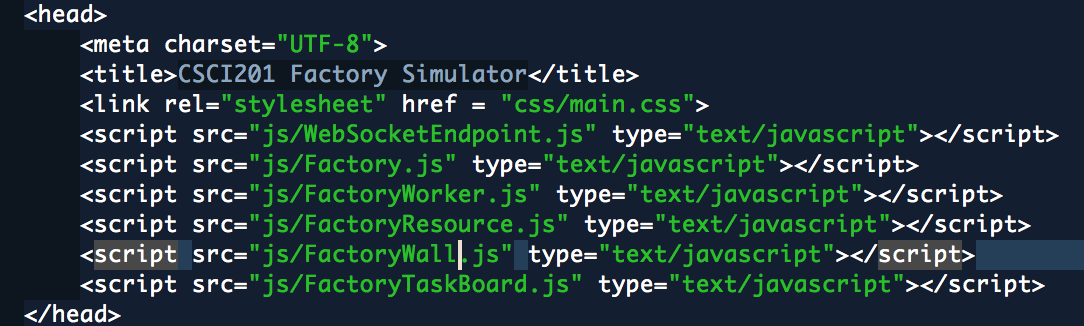


Now, create a JavaScript file FactoryWall.js in WebContent/js:

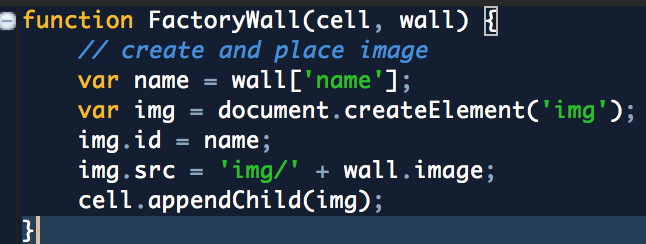
Right-click the js folder > New > Other > JavaScript Source File (in the JavaScript folder)



Since our code will reference this JavaScript file, we need to link it in our index.html

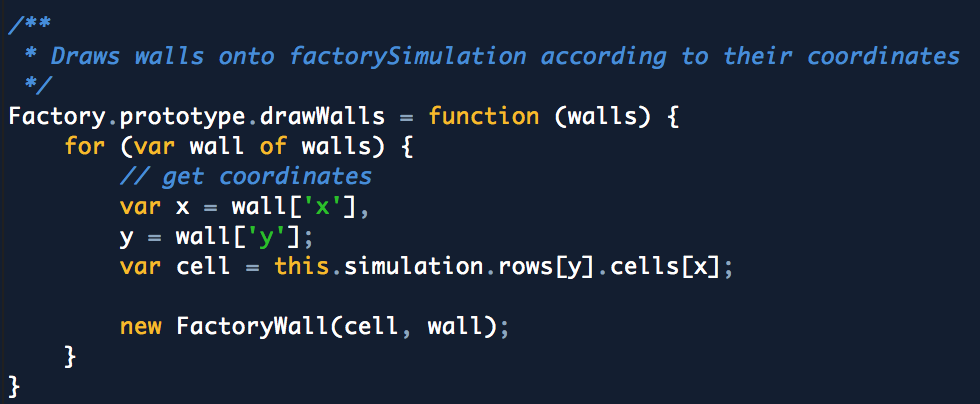


FactoryWall.js will take a cell (from the factory grid) and a wall as parameters. It will simply retrieve the name and image of the given wall and append an image element to the given cell.

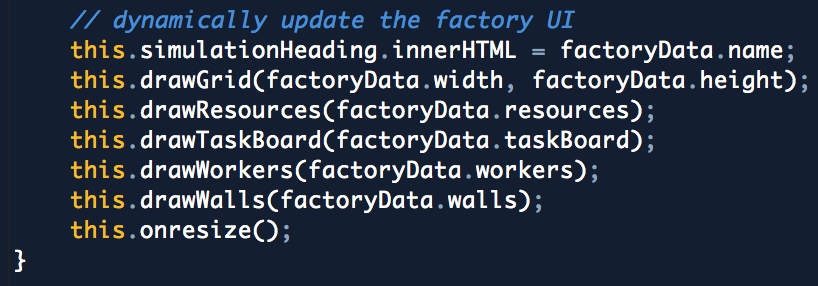


Factory.js is where we will write our drawWalls() function.

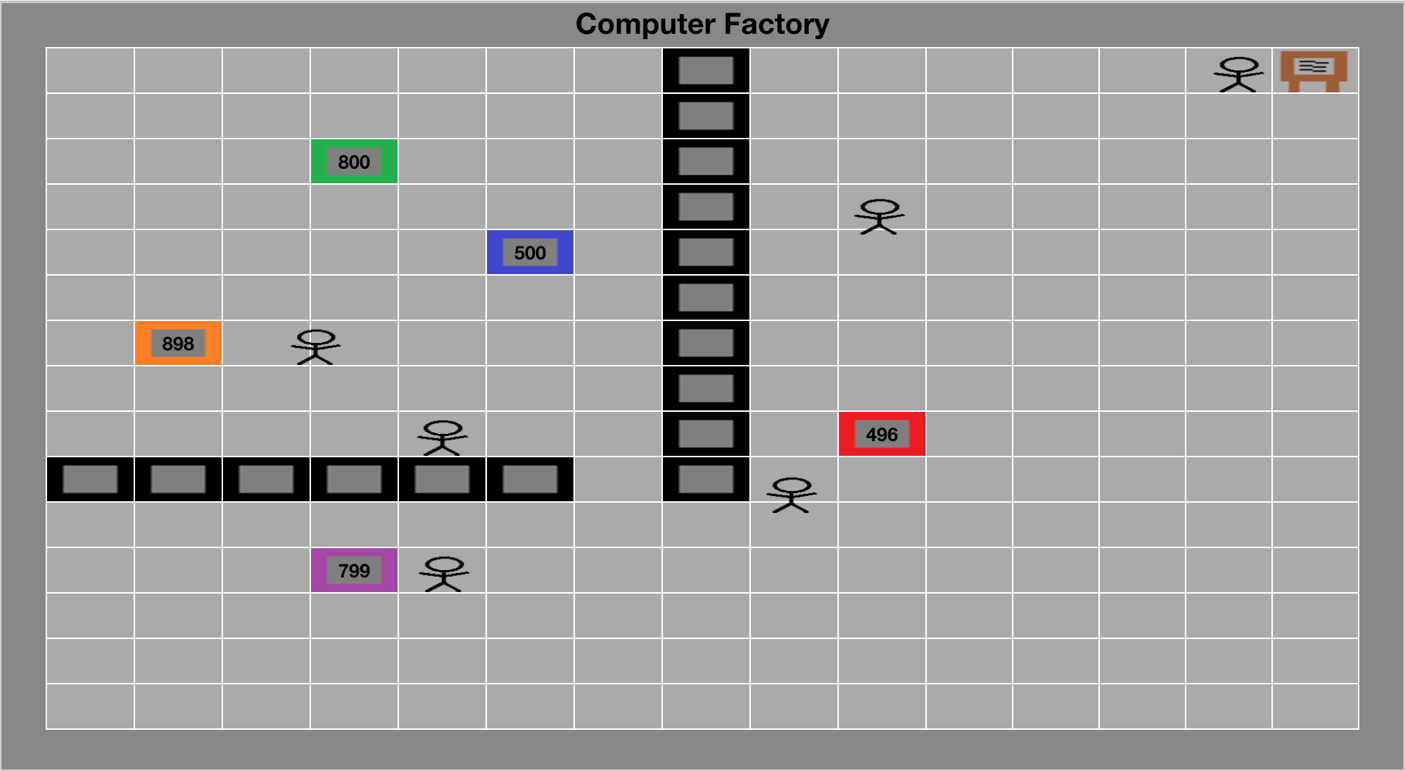
This function iterates over all the walls passed into it, parses their x and y location, retrieves the grid cell at this location, and finally creates a new FactoryWall (which just adds the image to the grid cell).



Let’s call the drawWalls function in the topmost function, Factory(factoryData).

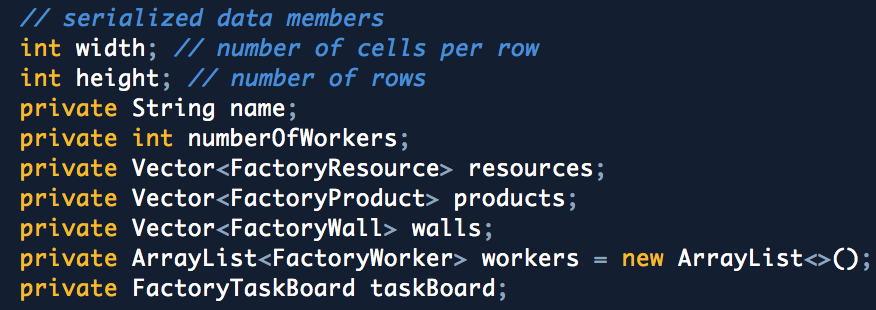


You should now see your walls!



Understanding

To give an overview of what’s going on here, factoryData is a representation of the Factory object the web client received when it started up. It contains selected data members from the Factory class, such as the factory’s width and height, and the vectors of resources, products, and walls.



FactoryWall.java is packaged in the same way. In order to get the x and y location of a given FactoryWall, we access the nodes of this factory data much in the same way we do in Assignment 1. There, data was stored as an XML while here it is stored as JSON (the “tree-like” structure is the same, it’s just how we choose to present the data).

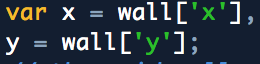
We get the “walls” data member with factoryData.walls.

Screen%20Shot%202017-01-14%20at%207.29.36%20PM.png

This is a vector of FactoryWalls, so we iterate through it.

Screen%20Shot%202017-01-14%20at%207.29.29%20PM.png

Now we have a single FactoryWall object, which has x and y as data members.



This is what we wanted! The rest of the data in factoryData is parsed in much the same way.

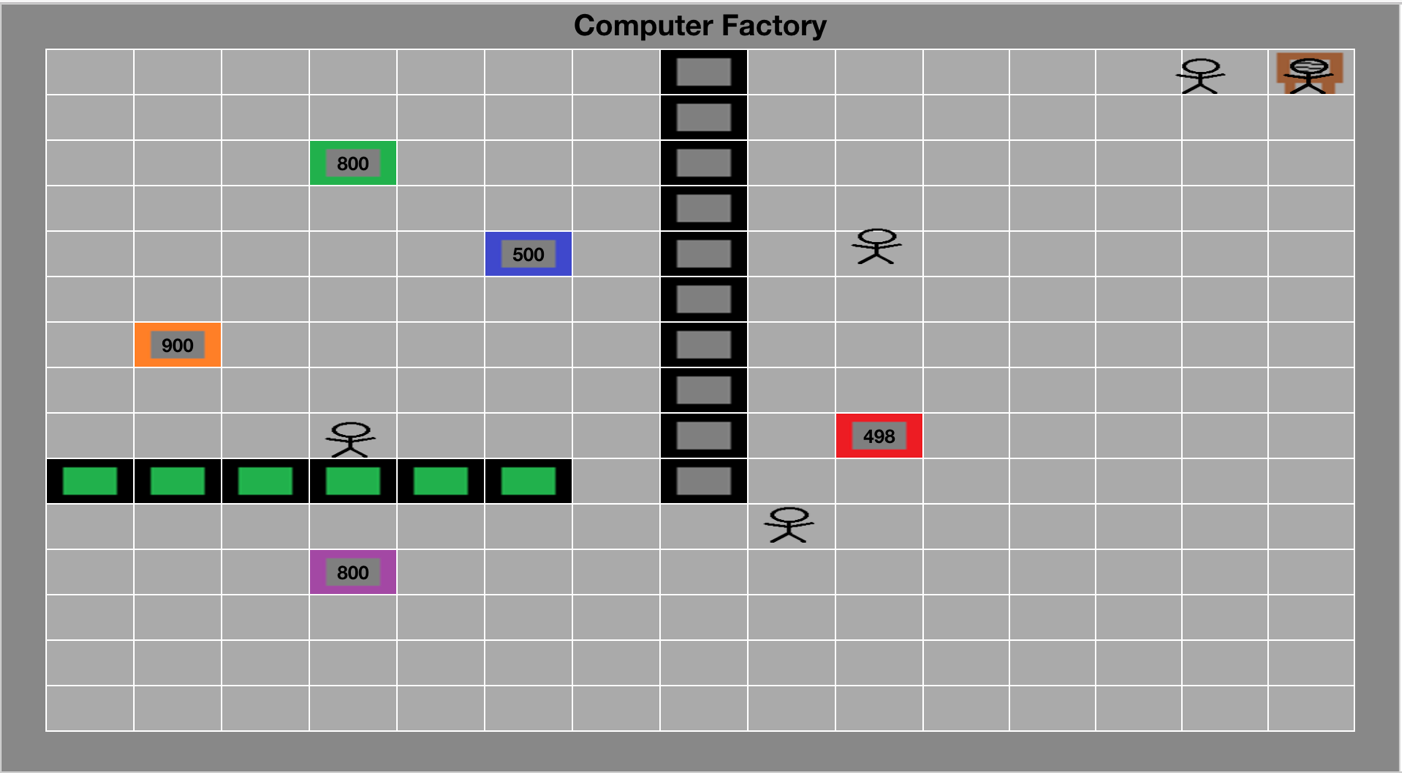
We’ve covered a lot of new material at once! It will make sense when it comes time in lecture, but hopefully you are getting an idea of how everything in the factory code ties together, aside from the different languages used to program it.

Expand on This

Expand on This sections are do-it-yourself sections that will require you to think about how to design a solution. There will be very little tutorial in this part, so use your creativity to complete this portion of the lab. Oftentimes, words that are in **bold** or sections IN ALL CAPS are very important to your understanding of this section. **SECTIONS IN BOLD AND CAPS ARE PROBABLY EVEN MORE SO.**

Let’s create a new type of wall. Create a new Java class and name it **FactoryWall2,** inheriting from FactoryObject**.** However, let's differentiate it from the normal FactoryWall. Use a program such as Paint or Gimp, and create a second wall image with a different color.

Modify your code so that the following walls are displayed.



**Things to know:**

In Chrome, paste localhost:8080/CSCI201Factory/index.html > Right-click anywhere > Inspect

**This will bring up useful information like the Console WHERE EVERYTHING ON THE WEB SIDE WILL PRINT (only code in your Java code will print to your console in Eclipse).**

**See Common Problems Guide/Google Chrome DevTools whenever you get stuck.**