**CSCI 201 Lab 3**

**Inheritance, File I/O**

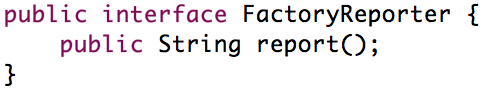
*We are all gifted. That is our inheritance. – Ethel Waters*

In this lab, you will create an interface for reporting the factory status at the end of the simulation and read in new walls from a file.

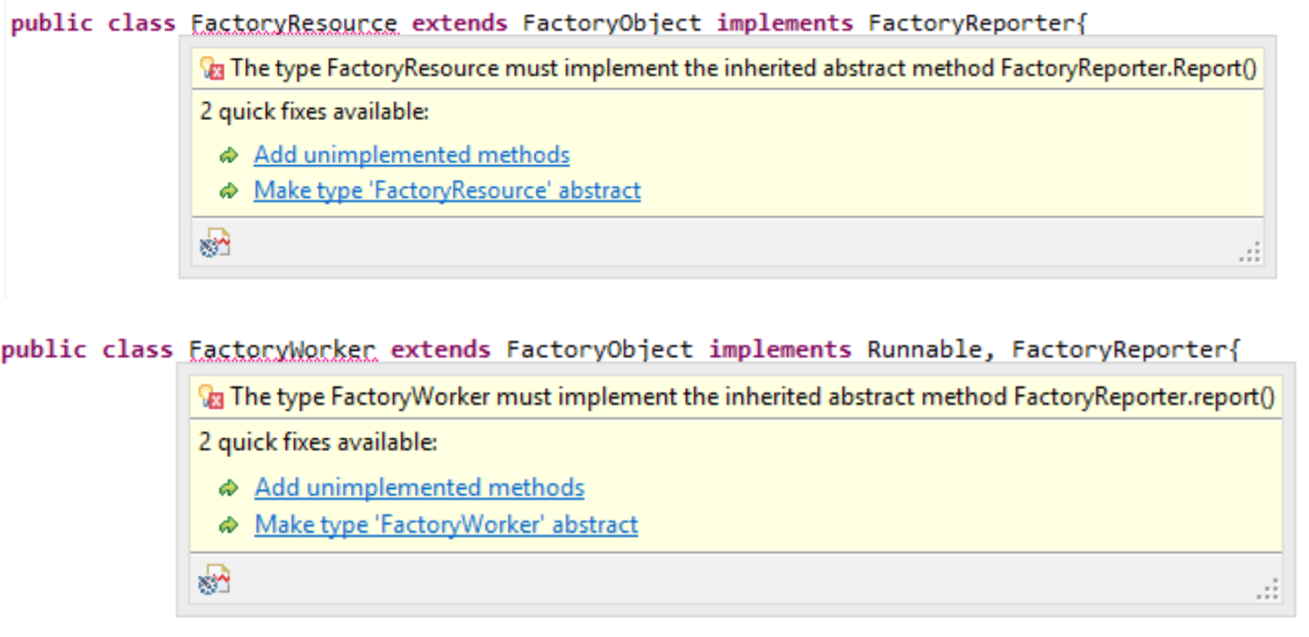
**Part 1 – Adding a FactoryReporter Interface**

Create a new FactoryReporter interface in the client package.

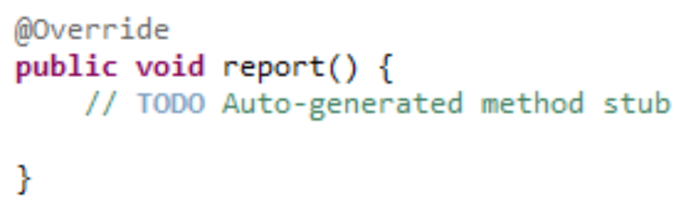
Add a single public method named report, which returns a String



Now make FactoryWorker and FactoryResource implement FactoryReporter.



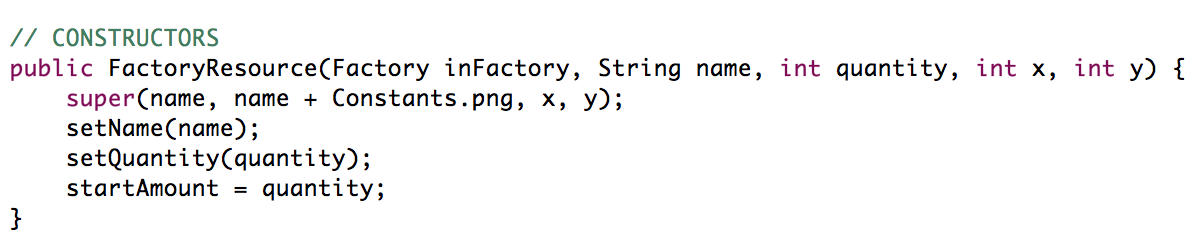
Add the unimplemented method “report().”



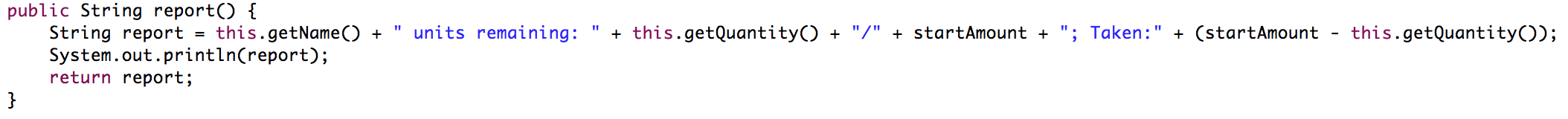
**Part 2 – Making meaningful reports**

For FactoryResource’s report, let’s report how many resources were taken in total. Create an integer member variable in FactoryResource to keep track of the starting amount. Set its value in the first constructor.





Now let’s write this report string and return it. Also print the report to the Eclipse console for debugging purposes.

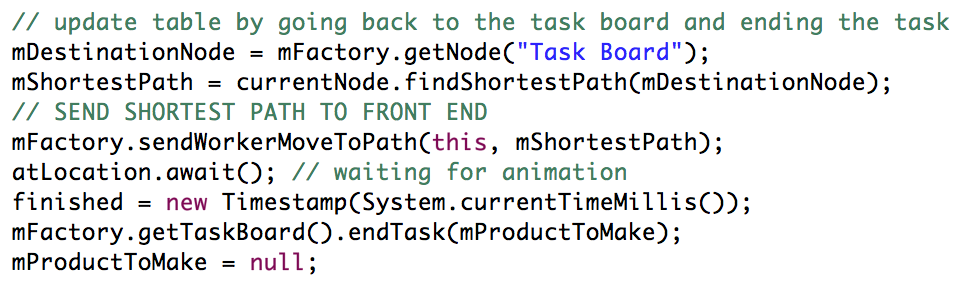


For FactoryWorker’s report, let’s report when the worker finished.

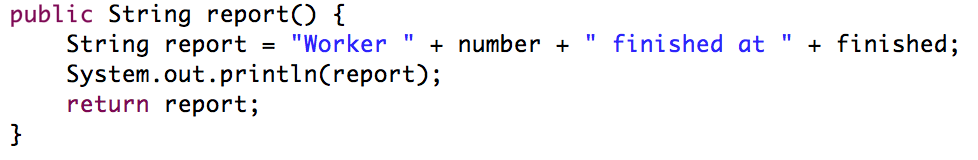
We need to add a Timestamp so import java.sql.Timestamp;



In FactoryWorker’s run() method, update the timestamp right before each task is submitted to the table.



Again let’s write this report, print it to the console, and return it.

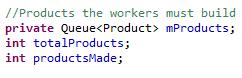


**Part 3 – When to report**

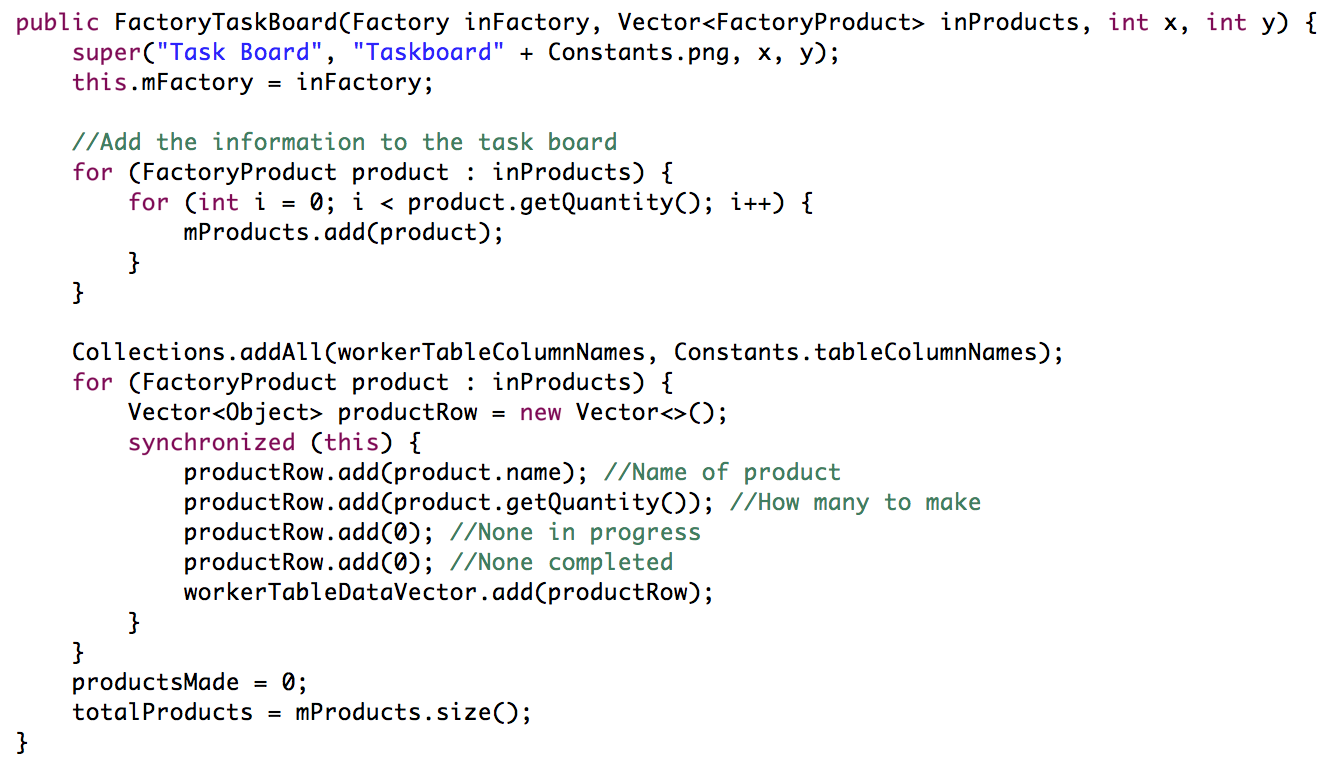
We want to report information about the results of the factory when all the products have been built by the FactoryWorkers. The TaskBoard will be a good place to do this.

The following code has been added to the base code in order to keep track of when all products have been made (i.e. the simulation is over).

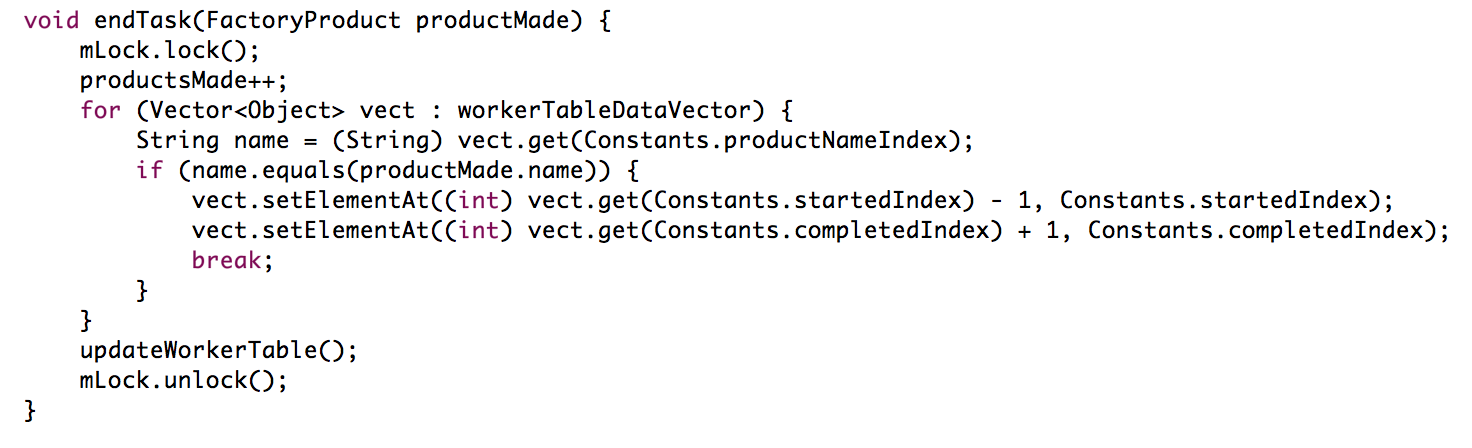
Two integer instance variables keep track of the total number of products needed and the number of products completed.



They are initialized inside FactoryTaskBoard’s constructor.



Each time FactoryTaskBoard’s endTask() is called, we know a product has been finished by a worker, so we increment productsMade.

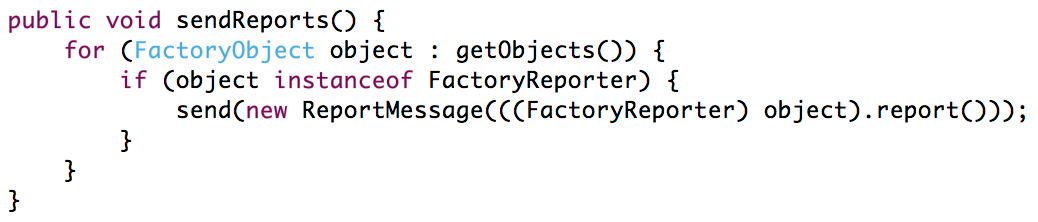


An isDone() method tells the caller if all products have been made.

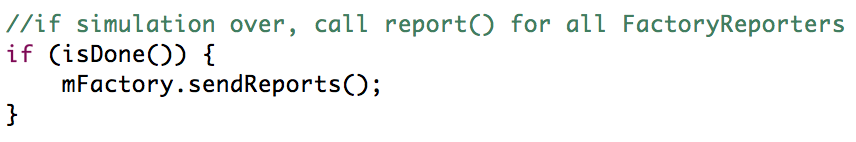


Now, when endTask() we must check whether that was the last task by calling isDone(). When isDone() is true, we want to call the report() method for every FactoryObject that implements the FactoryReporter interface. These reports must show up in our web front-end via a ReportMessage, which just holds a report String.

Factory is where we manage the simulation and send messages, so let’s make a public sendReports() function that FactoryTaskBoard will call. We loop through all FactoryObjects and check if they are instances of FactoryReporter, and send a ReportMessage with their report if they are.



Call it in endTask().



If you run the simulation now, you should see the following printed to console once it is over.

Motherboard units remaining: 473/500; Taken:27

Processor units remaining: 672/800; Taken:128

Hard Drive units remaining: 826/900; Taken:74

Memory units remaining: 470/800; Taken:330

Box units remaining: 473/500; Taken:27

Worker 0 finished at 2017-01-12 14:56:55.466

Worker 1 finished at 2017-01-12 14:56:55.466

Worker 2 finished at 2017-01-12 14:56:54.864

Worker 3 finished at 2017-01-12 14:56:55.465

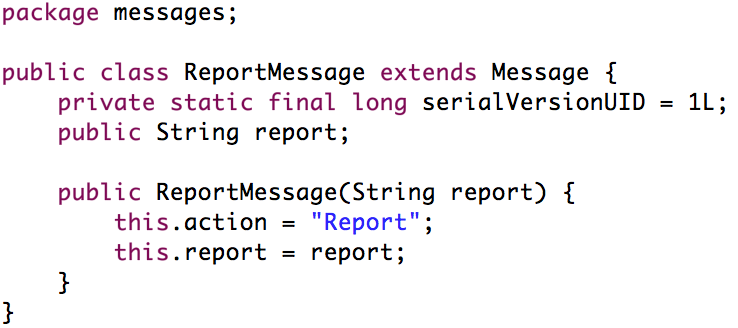
Worker 4 finished at 2017-01-12 14:56:55.466

Worker 5 finished at 2017-01-12 14:56:56.583

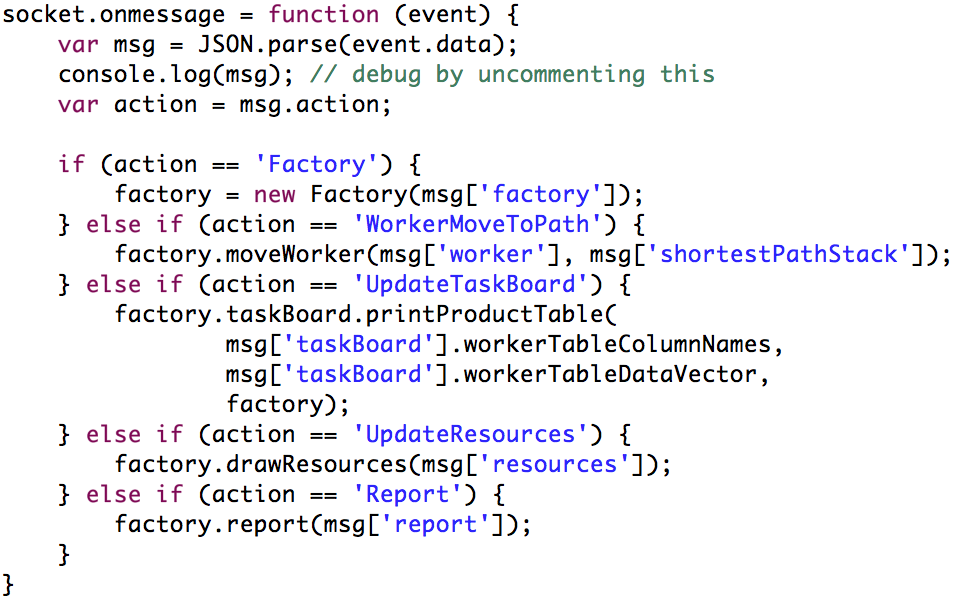
**Part 4 – Updating the front-end with reports**

We’ve sent the reports but haven’t done anything with them on the web-side.

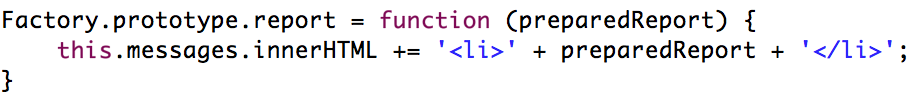
Add a ReportMessage class to the messages package.



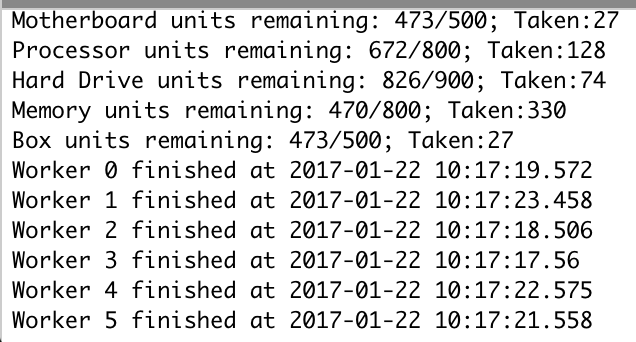
In WebContent/js/WebSocketEndpoint.js, add another else if statement for handling the reception of a ReportMessage. ReportMessage’s action variable is simply a string we use to describe what type of action this Message wants executed. If this action equals “Report” we pass ReportMessage’s report string variable to Factory.js’s report function, which we will write.



Factory.js’s report function will simply append a list item (<li>) with the report string to the messages element.



You should now see the reports in your messages text area.



**Part 5 – Reading from a file**

**In your src folder**, create a new file named ‘walls’. Be sure that it’s in src.

This file will contain the coordinates and images for additional walls we want to add to the factory. Copy the following:

1 1 http://grfx.cstv.com/story-nav/logos/usc.png

2 2 Wall.png

3 3 http://www-scf.usc.edu/~csci201/images/jeffrey\_miller.jpg

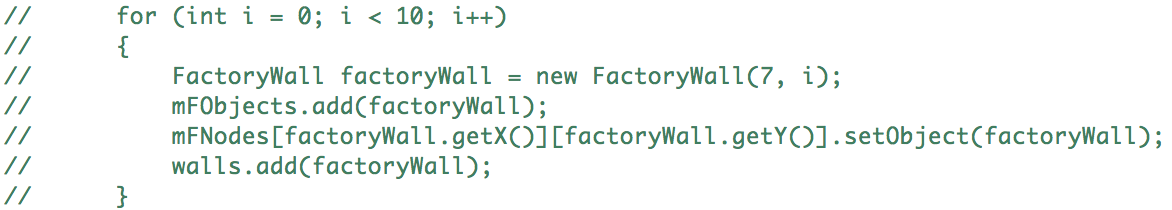
4 4 Wall.png

5 5 http://www-scf.usc.edu/~csci201/images/USC\_seal.gif

6 6 Wall.png

In Factory’s createWalls() we will open and parse this file and add its walls to the simulation.

First, though, let’s comment out our vertical row of FactoryWalls because otherwise the FactoryWorkers would be trapped without a path to the FactoryTaskboard.



Now, because this is a web app, we can’t access files as if it were a standalone Java project. We use this line to access our file as an input stream.

InputStream is = getClass().getClassLoader().getResourceAsStream("walls");

We import a file scanner

import java.util.Scanner;

and use it to scan our input stream.

Scanner reader = new Scanner(is);

We parse and add walls for as long as the reader has a next line.

while (reader.hasNext()) { }

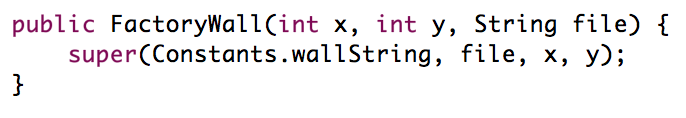
Each line, we parse the first int (x), the second int (y), and the rest of the string (file).

int x = reader.nextInt();

int y = reader.nextInt();

String file = reader.next();

Right now, our FactoryWall’s image file is pre-set in its constructor. We must add a new constructor that takes in a file name

.

Now we can add the factory walls as we did before, using this new constructor.

FactoryWall factoryWall = new FactoryWall(x, y, file);

mFObjects.add(factoryWall);

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

walls.add(factoryWall);

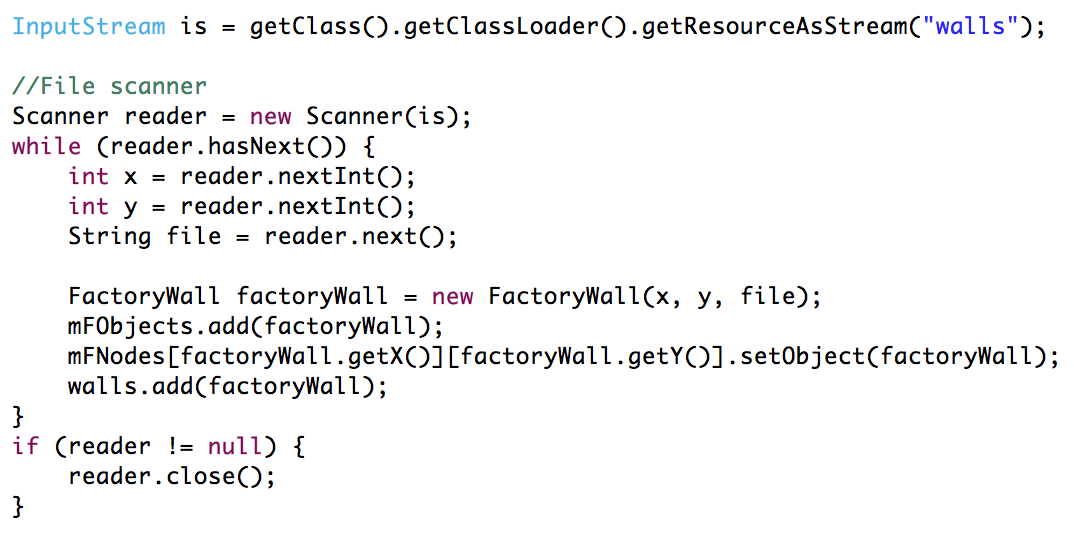
Finally, we close the reader.

if (reader != null) {

reader.close();

}

In all, we have:



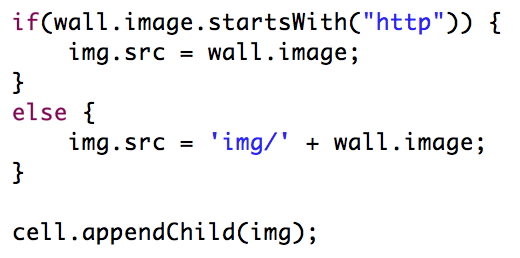
**Part 6 – Updating the front-end with walls**

These added walls get sent as factory data just like the ones we commented did, because we added them to Factory’s Vector<FactoryWall> walls;

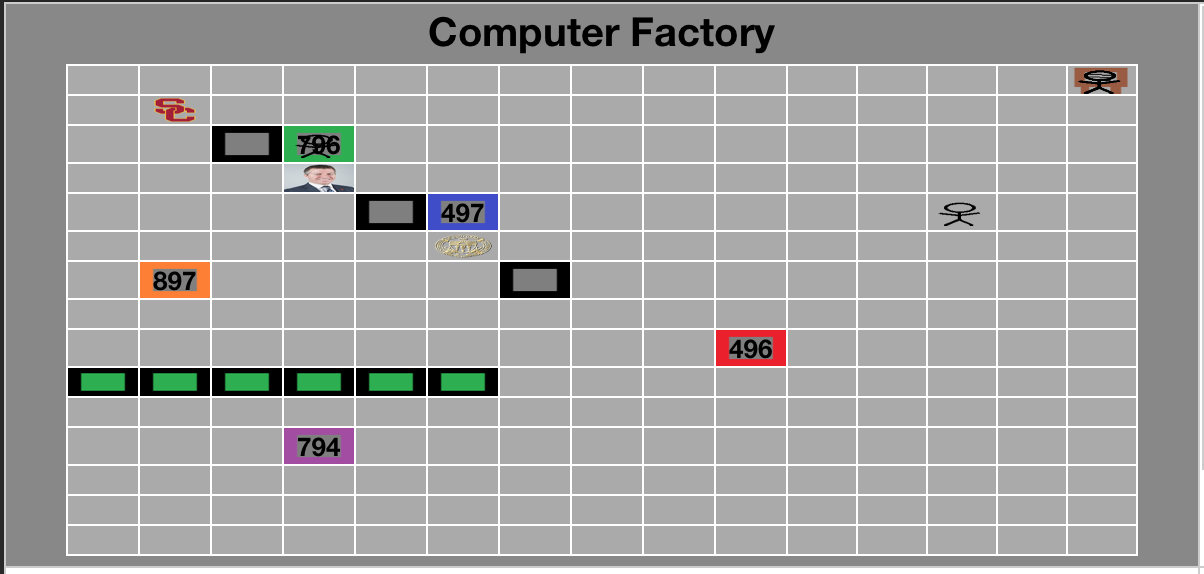
However, we must update our FactoryWall.js to account for some of the new walls image files being URLs as opposed to images in our project. The URLs all start with “http,” so we’ll just check if this wall’s image is a URL by using

if(wall.image.startsWith("http")) { }

In HTML, an img element’s src attribute can be a URL, so we set this attribute directly. Else, if it’s an image in our project we just look for it in the img folder.



Running your factory now will make the new walls appear!



**Note:** If your walls don’t show in your browser, restart your server or Eclipse. If you need to troubleshoot the walls, remember to uncomment console.log(msg) in WebSocketEndpoint.js. Then use the Chrome DevTools console to see if your walls made it properly to the front-end.

**Expand on This**

Report all the completed products at the end of the simulation. Your entire output should be:

Completed product: Cheap Computer

Completed product: Cheap Computer

Completed product: Okay Computer

Completed product: Okay Computer

Completed product: Okay Computer

Completed product: Okay Computer

Completed product: Good Computer

Completed product: Good Computer

Completed product: Good Computer

Completed product: Amazing Computer

Completed product: Amazing Computer

Completed product: Amazing Computer

Completed product: Amazing Computer

Completed product: Amazing Computer

Completed product: Cheap Server

Completed product: Cheap Server

Completed product: Cheap Server

Completed product: Cheap Server

Completed product: Cheap Server

Completed product: Cheap Server

Completed product: Cheap Server

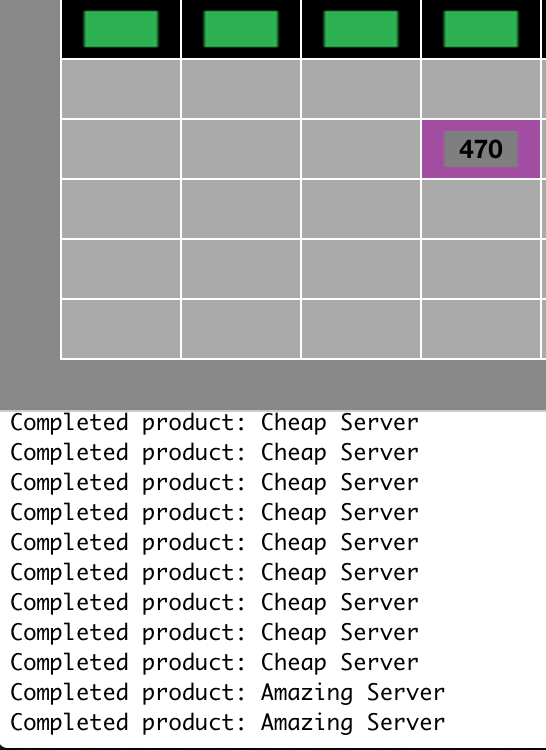
Completed product: Cheap Server

Completed product: Cheap Server

Completed product: Amazing Server

Completed product: Amazing Server

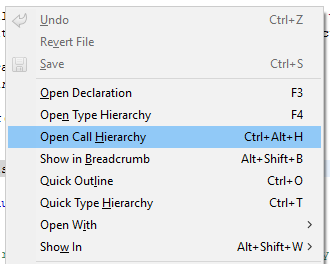
It must print in the message text area.



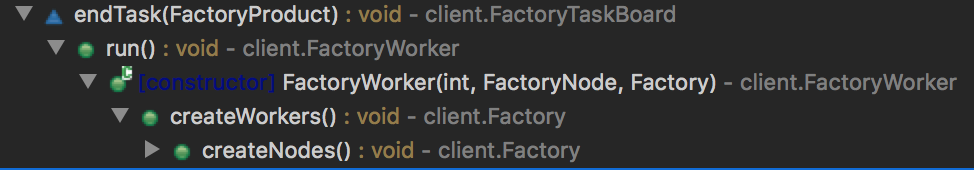
**Tips**

Get in the habit of restarting your server, deleting/re-adding Tomcat, and/or restarting Eclipse.

A great way to analyze any file is to right-click on a method and click “Open Call Hierarchy” or References > Project.



This should bring up this:



Follow the trail of methods to find what you are looking for. This is a great way to see how the factory works and how different classes interact with each other.