**CSCI 201 Factory Code Walkthrough**

Programming Exam Tips:

Review the Google Chrome DevTools Guide on the course site for useful debugging tips. Check out the Common Problems Guide on the course site if you experience any issues during the exam.

Note: I use a Mac so apologies for the shortcuts being just for Mac.

Note: method\_name() should be interpreted as method\_name(…), i.e. the methods referenced may or may not have parameters.

For Java files:

* Use Search > File (Command+H) to search functions or variables in the project
* Double-click a function > References > Project to see where it was called
* Command-click a function to see its declaration
* Any time you update anything on the backend, you have to Run on Server again. This includes Java files, which need to recompile. For HTML files, CSS files, and **sometimes** JSP files you can just (force) refresh the page because they’re client-side and just non-data-dependent visual elements. Always force refresh (Command+Shift+R) to be safe, and re-Run on Server JSP files if you’re not seeing expected updates.

The following is a high-level walkthrough of the functionalities of the factory base code (available from Lab 2). It will explain the events (in order) that eventually deliver the factory simulation to your front-end browser, how the back-end simulation updates the browser dynamically, and how the front-end signals the back-end when events occur. Anything not mentioned does not imply that it is not important, but most likely it is self-explanatory and you should be able to understand the code yourselves. We will omit the low-level networking details which you do not need to know for the programming exam. Check out Andrew’s Piazza post “Understanding WebSockets!” for some more details if you please (@409).

index.html:

* We run index.html on the server, everything starts here.
* This is just the skeleton of elements we KNOW will be on the page (e.g. a container for the factory grid) but not anything that is data-dependent (e.g. the factory worker – we don’t know how many there will be, if any).
* We include the JavaScript files in <head>, but they are NOT automatically executed, they are just referenced (think “import”)

WebSocketEndpoint.js

* Notice how index.html did not explicitly invoke any methods in this file to execute, but it “instantiated” it when it included it. The top of this file creates a new WebSocket, and the file’s **socket.onopen() function is called when the socket is open, which then triggers further functions**. This is the answer to “how does everything get started?”
* socket.onopen() calls readTextFile(), which makes a GET request for the factory.txt file stored on the server. It then calls the callback that was passed into it to go back to socket.onopen(). This callback is an anonymous function that just calls socket.send(text).

WebSocketEndpoint.java

* The backend Java files are compiled when you run the project. Without getting into the details, @ServerEndpoint(value = "/ws") makes this WebSocketEndpoint.java connect on the project’s websocket, which WebSocketEndpoint.js connected to.
* Therefore, it’s constantly listening for messages on this socket (because that’s how WebSockets work; note this is not how servlets work). Its onMessage function goes into its else statement because it does not yet have a factory.
* “factories” is a HashMap so that all clients get the appropriate factory. If I connected 5 seconds ago, I would get a different instance of the factory from someone who connected 10 seconds ago because their workers, etc. should be 5 seconds ahead.
* We put a new Factory object into this HashMap, which we get by calling FactoryParser().factory. “new FactoryParser(session, this, is)” is an anonymous function; its constructor is called before moving onto the “.factory” part.

FactoryParser.java

* The constructor calls factory = readFile(br); to parse the received “factory.txt”
* readfile() calls a bunch of parser helper functions, but most importantly creates a new Factory(), calls factory.createNodes(), and calls factory.sendFactory().

Factory.java

* This class implements Serializable, which means it can be packaged and sent as data.
* At the top of the file are its data members. All members without the keyword “transient” are included in the object when it is sent. They are stored similar to key-value pairs, for example the key “width” will have a corresponding int value that is set somewhere in Factory.java (in this case, its constructor).
* The first constructor (there are two) is called.

Before continuing, here are summaries of relevant classes:

FactoryObject.java:

* Nothing that special, just an abstract class that hold a barebones skeleton for the objects in the factory. When its children classes set something like their name, they are setting the “public String image” from FactoryObject.
* **Note that it is Serializable**.

FactoryNode.java:

* Serializable by extension.
* Everything placed on the factory grid is a FactoryNode – the FactoryWorkers, FactoryResource, FactoryProduct, FactoryTaskBoard.
* addNeighbor()/getNeighbors(): access mNeighbors, which is just a FactoryNode immediately next to this (the current object) FactoryNode on the factory grid. These will be assigned in Factory.java
* heuristicCostEstimate()/lowestFScore()makePath()/containsNode()/findShortestPath(): just helpers to figure out and assign the shortest path to another FactoryNode. Don’t worry about these, but you **should** know what a path is.
* A path consists of the FactoryNodes (grid cells, if you want to think of it visually) from one FactoryNode to another. For example there is a single shortest path from the TaskBoard to the Motherboard.

When FactoryParser.java calls factory.createNodes():

* createNodes() calls some obvious functions to initialize the FactoryNodes, including calling createResources() and createWorkers(). It doesn’t initialize any FactoryProducts because none have been made yet.

createResources():

* The following three calls are common in other create methods, such as createTaskBoard()
* mFObjects: just a list of all the FactoryObjects that get added to the factory
* mFNodes: a 2D-array representing the factory grid. When you add a FactoryNode to the factory, it’s added to mFNodes[x][y] corresponding to its x- and y-position.
* mFNodeMap: a convenient HashMap where you can access FactoryNodes by their name. For example if you know there is a resource “Motherboard”, you can get the corresponding FactoryNode (which holds data like its x- and y-position, etc.) by using “Motherboard” as the key to mFNodeMap.

FactoryWorker.java:

* createWorkers() only does three things, one of which is create a new FactoryWorker.
* The important thing here is the run() method. You will learn about threads, but basically each FactoryWorker is its own thread, which means they can all execute their run() methods at what appears to be the same time.
* Note: you will not be responsible for threading, however you should be able to manipulate the code inside thread-related code. For example, you have edited code inside of locks (myLock.lock() -> some code -> myLock.unlock(). Therefore you should be able to manipulate code inside of run() and similar methods, excluding atLocation.await(), Thread.sleep(1), etc.

run():

* Notice the try block is a big while loop. It may seem that the loop will execute very frequently, but there are conditions inside the loop that the worker will wait on (again, outside of your scope right now).
* Assigning the worker a task:
  + The if statement ensures you only assign the worker a task if they don’t have a product to make.
  + mDestinationNode…: assigns the task board as the destination node
  + mShortestPath…: calls FactoryNode’s findShortestPath() method to find the shortest path to the task board
  + mFactory.sendWorkerMoveToPath(): calls Factory’s sendWorkerMoveToPath() method to send a message to the front-end client that the worker has been assigned a path (see sendWorkerMoveToPath() below).
  + atLocation.await(): the worker waits for a signal from the front-end client that it has finished animating/moving the worker to its destination. This happens in JavaScript, remember the MVC design pattern.
  + while (!mDestinationNode.aquireNode()) { Thread.sleep(1); }: the worker waits in line for the task board to be free if there are other workers at the task board.
  + mProductToMake…: since we have exited the above while loop, we know we are at the task board. The task board assigns the worker a task.
  + Thread.sleep(1000): the task board takes one second to assign a task to a worker.
  + mDestinationNode.releaseNode(): the task board is now free so the next worker can use it (outside of your scope).
* Building the product:
  + A product is composed of multiple resources. Just like assigning a task, a path to each resource in the product is sent to the client (mDestinationNode() -> mShortestPath() -> mFactory. sendWorkerMoveToPath()).
  + atLocation.await() ensures that the code will not continue until the worker receives a signal that it is at the desired resource.
  + FactoryResource toTake…: gets resource at the destination node (which we know is a resource node)
  + toTake.takeResource(): calls FactoryResource’s takeResource() method to decrement the number of that resource still available.
  + mFactory.sendResources(): calls Factory’s sendResources() method (see sendWorkerMoveToPath()).
* Updating the task board table:
  + mDestinationNode… -> mShortestPath -> mFactory. sendWorkerMoveToPath() -> atLocation.await() tells the front-end to send the worker to the table and waits for a signal that this has happened.
  + mFactory.getTaskBoard().endTask(mProductToMake): calls the FactoryTaskBoard’s endTask() method which updates
  + mFactory.getTaskBoard().endTask(): worker no longer has a product to make, so we set it to null to prepare for the next loop of the while loop.

sendWorkerMoveToPath():

* All the send\_\_\_() methods in the factory code all do the same thing, which is send a message to the client.
* send(new WorkerMoveToPathMessage(worker, pathStack)): calls Factory’s send() method, which just takes a Message.

Message:

* Serializable abstract class (it should be abstract in the base code, either way it is never directly instantiated). Its sole purpose is to hold data.
* Holds “String action,” an identifier which will act as a key when we send the message to the client. For example WebSocketEndpoint.js will receive this message and if (action == ‘WorkerMoveToPath’), it will do move the worker elements, but if (action == 'UpdateResources') it will update the resource elements. **Note: the JavaScript cannot interact with any Java except indirectly either through a Servlet or by sending messages over the WebSocket.**

WorkerMoveToPathMessage:

* Like all Messages, its only method is its constructor, which takes in data as its parameters (in this case, a FactoryWorker and a Stack holding its path). All the constructor does is set the corresponding data members, so that when the message is serialized the receiver can access it.

send(Message message):

* String json = mGson.toJson(message): the message passed into send() is converted into JSON format by Google’s convenient Gson library. Web protocols can only send certain data formats and browsers can parse only these formats. JSON is the one (the most popular), XML is another.
* mWebSocketEndpoint.sendToSession(mSession, json): calls WebSocketEndpoint.java’s sendToSession() method to send the JSON message over the WebSocket to be received by WebSocketEndpoint.js.

WebSocketEndpoint.js:

* socket.onmessage() receives the JSON data

socket.onmessage():

* var msg = JSON.parse(event.data): calls JavaScript’s built-in JSON parser
* var action = msg.action: reads in the message’s action attribute (Message’s String action)
* if (action == 'Factory'): If the message was of type FactoryMessage (with String action = “Factory”)
* factory = new Factory(msg['factory']): creates a new Factory object - **Factory.js object NOT Factory.java object** - with the message’s “factory” attribute (FactoryMessage has a data member “private Factory factory”).
* The other else if statements call methods from Factory.js depending on the value of the action in the received message.

Factory.js:

* It’s confusing, but objects in JavaScript can be defined as functions. So when we do “factory = new Factory(msg['factory']),” we are calling “function Factory(factoryData),” where factoryData is the value of msg[‘factory’] – **which if you recall is a Factory.java object!** You may think of “function Factory(factoryData)” as a constructor if it helps, just know that is not accurate terminology.
* Now any time we write something like factoryData.resources, we are accessing the data member named “resources” from this Factory.java object (private Vector<FactoryResource> resources). **Note: the factoryData.resources received by the client is already stored as an array (not a Vector) in the JSON. This was likely done by the Gson library.**
* If we take “this.drawResources(factoryData.resources)” as an example, the function loops through the array of FactoryResources. **FactoryResource is also Serializable because it extends FactoryNode which implements FactoryObject which implements Serializable**. Therefore we can access its data members just like we can access Factory.java’s. The function simply generates some HTML, but notice how it creates a “new FactoryResource(cell, resource).” Again, this is a new FactoryResource.js object, NOT a FactoryResource.java object. The FactoryResource.js object then does its own thing to generate HTML for the resources.

FactoryWorker.js:

* function Factory(factoryData) calls “this.drawWorkers(factoryData.workers),” which calls “Factory.prototype.drawWorkers,” which pushes a new FactoryWorker.js object into the Factory.js’s array of workers.
* You can look through FactoryWorker.js on your own, but let’s take a look at FactoryWorker.prototype.move.

Before continuing, a note on how the animation gets constantly updated by the browser. This is not crucial for the exam.

* FactoryWorker.prototype.move defines function tick(), which is the callback for window.requestAnimationFrame(tick). Side note: on the first pass of FactoryWorker.prototype.move, just because we define function tick() does not mean we call it, it is only called at the bottom by the second window.requestAnimationFrame(tick). Then once inside the function tick() callback, it calls window.requestAnimationFrame(tick) again which starts a recursive-like back-and-forth between the two.
* window.requestAnimationFrame(tick): requests your browser to call a specified animation function before it performs the next repaint, usually at a rate of 60 callbacks per second, but this depends on the browser.

tick():

* if (self.moveTowards(self.nextNode, delta)): moveTowards returns true if the worker is close enough to the next node in its path within a certain tolerance.
* self.currentNode = self.nextNode: if the worker has arrived, the current node is the node it was moving to.
* if (self.pathStack.length > 0): if there are more nodes in the path, i.e. the worker is not at its destination node, the worker will update its nextNode
* else: we are at our destination node.
* socket.send(JSON.stringify(): sends a string containing an action, the worker, and its currentNode over the socket to be received by WebSocketEndpoint.java’s onMessage() method. As it stands now, the factory code only supports one action being sent from the front-end client to the back-end, which is “WorkerArrivedAtDestination”.

WebSocketEndpoint.java:

* Enters “if (factory != null)” because there is already a factory in the session.
* factory.listen(message): calls Factory.java’s listen() method

listen():

* JsonObject msg…: creates a JSONObject out of the action string received from socket.send().
* Action action…: calls ActionFactory.java’s getAction() method
* if (action != null)…: if the getAction() method returned an Action, the action exists so call its execute() method. We check if it exists because the client can easily write their own JavaScript to send a non-existent Action.

ActionFactory.java:

* Adds Actions to its actionMap, a HashMap of supported actions. Pretty straightforward especially since “WorkerArrivedAtDestination” is the only supported Action.

Action.java:

* Abstract class that just declares an abstract execute() method. Similar to the abstract Message.java class.

WorkerArrivedAtDestinationAction.java:

* Defines its abstract execute() method.
* int workerIndex…: gets the index of the worker that sent the message. Recall that the FactoryWorker.js sent its corresponding FactoryWorker.java object (from factoryData.workers) in the message, and FactoryWorker.java has a data member “public final int number.”
* int x…: gets the x value of the sent currentNode which is a FactoryNode.java.
* factory.getWorker(workerIndex).atLocationSignal(x, y); signals the FactoryWorker that it has arrived at its destination, which if you recall allows the while loop in their run() method to continue and get a new destination.

That was all one pass of the factory code! Hopefully you understand what’s going on at least at a high level so that you know generally where to look for a class or method that you may need for the exam. Be sure to look at the rest of the code as this was of course not totally comprehensive.

Good luck!!!