Inheritance

Part 1 – FactoryWall Creation

As was explained in Lab 1, all of the labs in this class will build off the Factory code that was provided to you. You imported the Factory project in Lab 1, so open that project.

In this lab, you will 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. Once this is done, you will create a FactoryReporter interface. This interface will force child objects to be able to report information. You will make the FactoryWorker class and the FactoryResource class inherit from this.

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.

Once the class is created you will notice that there is an error.

```java
public class FactoryWall extends FactoryObject {
    // Implicit super constructor FactoryObject() is undefined for default constructor. Must define an explicit constructor
    public FactoryWall(Rectangle inDimensions) {
        super(inDimensions);
        // TODO Auto-generated constructor stub
    }
}
```

This is because we only have one constructor for the FactoryObject class, and it takes a Rectangle as an argument. Go ahead and add the FactoryWall(Rectangle) constructor.

```java
public FactoryWall(Rectangle inDimensions) {
    super(inDimensions);
    // TODO Auto-generated constructor stub
}
```
Since the goal is to draw the wall onto the screen, we need to set the image of the FactoryObject. The FactoryObject class already has an Image member variable.

```java
public abstract class FactoryObject
protected Image mImage;
```

Simply use the ImageLibrary class to load an image. The wall.png file is provided on the course web site for this lab.

```java
mImage = ImageLibrary.getImage("resources/img/Wall.png");
```

The resulting class is then:

```java
public class FactoryWall extends FactoryObject{

    public FactoryWall(Rectangle inDimensions) {
        super(inDimensions);
        mImage = ImageLibrary.getImage("resources/img/Wall.png");
    }
}
```

Simply click and drag Wall.png to the resources/img folder to add it to the project. You may use the Wall.png provided, or use any other image.

Luckily for us, the FactoryObject class already contains a draw method that will draw the image for us.

```java
public void draw(Graphics g, Point mouseLocation) {
    g.setColor(Color.BLACK);
    g.drawImage(mImage, renderBounds.x, renderBounds.y, renderBounds.width, renderBounds.height, null);
    drawMouseover(g, mouseLocation);
}
```

We don’t need to touch anything there. Our FactoryWall class is now fully implemented! Let’s add some walls into the factory simulation.
Part 3 – Adding FactoryWalls to the FactorySimulation

Let's take a look at the FactorySimulation(Factory, Table) constructor in FactorySimulation.java. The FactorySimulation constructor creates FactoryNodes and places FactoryObjects onto them. We will just hard-code the walls into the Factory, but walls could just as well be specified from the FactoryServer through the configuration file.

Let's create a single wall at location (7, 8) in the factory.

First, create a FactoryWall object.

```java
FactoryWall factoryWall = new FactoryWall(new Rectangle(7, 8, 1, 1));
```

The Rectangle specifies the x, y coordinates of the FactoryWall, and its width and height. Set the x value to 7, and the y value to 8. Make sure the height and width are both 1.

Next, we need to make sure to add this wall to the FactorySimulation. The FactorySimulation has an ArrayList of FactoryObjects named mFObjects. Since FactoryWall is a FactoryObject, the Simulation will take care of everything for us.

```java
mFObjects.add(factoryWall);
```

*Note: Adding the FactoryWall won't cause the FactoryWall be drawn. Since draw-order matters, there is a separate ArrayList for each type of FactoryObject. However, adding the Object to mFObjects will ensure the image is scaled properly, which is very important.*

Lastly, we need to set the corresponding FactoryNode’s object to this new FactoryWall.

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

*Note: FactoryNodes will draw their Object instead of themselves when they own an object. So this step will effectively allow workers to navigate around the walls and ensure that the walls will be drawn.*

*Note: See FactoryPanel.java for more insight about how the FactorySimulation is drawn.*

If you run a FactoryClient, you will see the workers navigating around the newly created wall.

![Image of Factory with a wall]

Now that one wall works, create a bunch of walls for the factory.
//Create the walls
for(int i = 0; i < 10; i++) {
    FactoryWall fw = new FactoryWall(new Rectangle(7, i, 1, 1));
    mFObjects.add(fw);
    mFNodes[fw.getX()][fw.getY()].setObject(fw);
}

for(int i = 0; i < 6; i++) {
    FactoryWall fw = new FactoryWall(new Rectangle(i, 9, 1, 1));
    mFObjects.add(fw);
    mFNodes[fw.getX()][fw.getY()].setObject(fw);
}
Part 4 – Adding A FactoryReporter Interface

Create a new FactoryReporter Interface in the client package.

Add a single public method named report.

```java
package client;

public interface FactoryReporter {
    public void report();
}
```

Now make FactoryWorker and FactoryResource implement FactoryReporter.

```java
public class FactoryWorker extends FactoryObject implements Runnable, FactoryReporter {
    // The type FactoryWorker must implement the inherited abstract method FactoryReporter.report()
    public void report() {
        // TODO Auto-generated method stub
    }
}
```

You will need to add the unimplemented method “report”.

```java
@Override
public void report() {
    // TODO Auto-generated method stub
}
```
Part 5 – When to report

Now that we have the FactoryReporter Interface, let's put it to use.

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.

Add two integer instance variables to keep track of the total number of products, and the number of products completed.

```java
// Products the workers must build
private Queue<Product> mProducts;
int totalProducts;
int productsMade;

Initialize these integers inside the TaskBoard constructor, not the instance constructor.
```

```java
protected FactoryTaskBoard(JTable inTable, Vector<Product> inProducts, int x, int y) {
    super(new Rectangle(x, y, 1, 1));
    // Add the information to the task board
    mTable = inTable;
    for (Product product : inProducts) {
        for (int i = 0; i < product.getQuantity(); i++) {
            mProducts.add(product);
        }
    }
    for (int i = 0; i < Constants.tableColumnNames.length; i++) {
        workerTableColumnNames.add(Constants.tableColumnNames[i]);
    }
    for (Product product : inProducts) {
        Vector<Object> productRow = new Vector<Object>();
        synchronized(this) {
            productRow.add(product.getName()); // Name of product
            productRow.add(product.getQuantity()); // How many to make
            productRow.add(0); // None in progress
            productRow.add(0); // None completed
            workerTableDataVector.add(productRow);
            updateWorkerTable();
        }
    }
    productsMade = 0;
    totalProducts = mProducts.size();

Next, whenever endTask() gets called, we can say that a product has been made by a worker. Increment productsMade in here. Make sure to place the statement between the lock statements.

```java
mLock.lock();
productsMade++;
mLock.unlock();
```
Now, add a method that tells the caller if all the products have been made.

```java
public boolean isDone() {
    return (productsMade == totalProducts);
}
```

Now, in FactorySimulation we can check if all the tasks have been completed in the update(double) method.

```java
if(mTaskBoard.isDone())
```

When the FactorySimulation is over, we want to report everything. Iterate through all of the mFObjects, and check if it is an instance of a FactoryReporter. If it is, we can then call its report method.

```java
if(mTaskBoard.isDone()) {
    for(FactoryObject object : mFObjects) {
        if(object instanceof FactoryReporter) {
            ((FactoryReporter) object).report();
        }
    }
}
```

Lastly, make sure that nothing is updated after the factory ends. Add a boolean to the FactorySimulation, and return out of the update(double) method if it is true.

```java
private boolean isDone = false;

public void update(double deltaTime) {
    isDone = false;
    //Update all the objects in the factory that need updating each tick
    for(FactoryObject object : mWorkers) object.update(deltaTime);
    if(mTaskBoard.isDone()) {
        isDone = true;
        for(FactoryObject object : mFObjects) {
            if(object instanceof FactoryReporter) {
                ((FactoryReporter) object).report();
            }
        }
    }
}
```

Now, the FactoryReporter will call its report(double) method after the simulation ends. You can test this by adding print statements inside the report methods.
Part 6 – Making meaningful reports

For the FactoryResource 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 this value in the constructor that takes parameters, not in the instance constructor.

```java
int startAmount;
startAmount = mResource.getQuantity();
```

Add the following to the FactoryResource’s report() method:

```java
@Override
public void report() {
    System.out.println("Total resources: "+mResource.getQuantity()+"/"+startAmount
        + "Taken:"+(startAmount-mResource.getQuantity()));
}
```

Now in the FactoryWorker add a TimeStamp.

**Note: import java.sql.Timestamp;**

```java
private Timestamp finished;
```

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

```java
//update table
{
    #DestinationNode = mFactorySimulation.getNode("Task Board");
    #ShortestPath = mCurrentNode.findShortestPath(#DestinationNode);
    #NextNode = #ShortestPath.pop();
    @location.await();
    finished = new Timestamp(System.currentTimeMillis());
    #FactorySimulation.getTaskBoard().endTask(#ProductToMake);
    #ProductToMake = null;
}
```

Lastly, add the following to the FactoryWorker’s report() method:

```java
@Override
public void report() {
    System.out.printIn(mNumber +" finished at "+ finished);
}
```

Once the factory completes, something similar to the following should print out:

```
Total resources: 473/500Taken:27
Total resources: 672/800Taken:128
Total resources: 826/900Taken:74
Total resources: 470/800Taken:330
Total resources: 473/500Taken:27
0 finished at 2015-07-05 17:49:27.412
1 finished at 2015-07-05 17:49:08.589
2 finished at 2015-07-05 17:49:10.49
3 finished at 2015-07-05 17:49:09.562
4 finished at 2015-07-05 17:49:03.893
5 finished at 2015-07-05 17:49:07.353
```
Part 7 – Questions

If you have sufficiently understood this lab, you should be able to answer these two questions fairly easily.

1. What classes use inheritance in the factory code?
2. What methods are overridden?
Expand on This

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.