Introduction

In this lab, you will implement the functionality of the Pause, Continue, and Restart buttons on the FactoryController. You will also make the buttons respond to the JSlider and pause the simulation when the ClientGUI is minimized.

Part 1 – Initial set-up

Remember creating these buttons?

```java
private JButton pauseButton;
nprivate JButton continueButton;
private JButton resetButton;
```

You already have these buttons in the createTimePanel() method. Turn them into member variables.

At the start of the simulation, the simulation will be running. The continue button should be disabled.

```java
continueButton = new JButton("Continue");
continueButton.setEnabled(false);
```

When the GUI is displayed, you should notice the continue button is grayed out.
Part 2 – Adding ActionListeners

On each of the buttons, add an ActionListener. Since each of the ActionListeners are unique to each button, we will use an anonymous inner class for each button.

```java
pauseButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        //TODO: Insert code here to pause
    }
});
continueButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        //TODO: Insert code here to continue
    }
});
resetButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        //TODO: Insert code here to reset
    }
});
```

When the pause button is pressed, we want to disable it. Secondly, the continue button should be enabled so the user can resume.

We want the continue button to do the exact opposite.

```java
pauseButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        continueButton.setEnabled(true);
        pauseButton.setEnabled(false);
    }
});
continueButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        continueButton.setEnabled(false);
        pauseButton.setEnabled(true);
    }
});
```

Notice the behavior of the buttons in the GUI when clicked. They should toggle each other.
Now, we want the slider to update correctly. When the pause is pressed, remember the current slider position, and adjust the slider to “Paused”. When continue is pressed, adjust the slider to the remembered position.

Create a member variable to store the speed.

```java
int speed = Constants.simulation_1x;
```

Update the ActionListeners as shown:

```java
pauseButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        continueButton.setEnabled(true);
        pauseButton.setEnabled(false);
        speed = simulationSpeedController.getValue();
        simulationSpeedController.setValue(Constants.simulation_0x);
    }
});
```

```java
continueButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        continueButton.setEnabled(false);
        pauseButton.setEnabled(true);
        simulationSpeedController.setValue(speed);
    }
});
```

Now, the slider should move properly. When the slider moves, the factory will also update properly.
Part 3 – Implementing the ChangeListener

You will notice that the buttons do not update properly if the user moves the slider. We need our FactoryController to respond to the change of the slider.

We can do this easily by implementing the ChangeListener interface.

class FactoryController extends JFrame implements ChangeListener

We will override the stateChanged(ChangeEvent) method to manage the buttons based on the slider’s position.

@Override
public void stateChanged(ChangeEvent ce) {
    //TODO: Update the buttons based on the slider's value
}

If the slider’s value is 0, enable the continue button, and disable the pause button. Otherwise, do the opposite.

@Override
public void stateChanged(ChangeEvent ce) {
    int state = ((JSlider)ce.getSource()).getValue();
    if (state == Constants.simulation_0x) {
        continueButton.setEnabled(true);
        pauseButton.setEnabled(false);
    } else {
        continueButton.setEnabled(false);
        pauseButton.setEnabled(true);
    }
}

Lastly, we need to add this class to the list of ChangeListeners that the JSlider will notify when it is altered.

simulationSpeedController.addChangeListener(this);

Or

simulationSpeedController.addChangeListener(factoryController);

Note: Use the first if you want to add the factoryController to the simulationSpeedController in its constructor. Use the second if you want to add the factoryController to the simulationSpeed controller in the clientGUI constructor. Either will work.

You will now notice when the slider is moved, the buttons will update properly.
Part 4 – Implementing the WindowStateListener

When the ClientGUI is minimized while the FactoryController is open, the FactoryController is still displayed. We want the FactoryController to hide. Also, when minimized, the FactorySimulation should be paused and resume once the user opens the window back up.

Implement the WindowStateListener interface.

```java
class FactoryController extends JFrame implements ChangeListener, WindowStateListener
```

We will override the windowStateChanged(WindowEvent) method to manage what happens when the ClientGUI window is minimized and maximized.

```java
@Override
public void windowStateChanged(WindowEvent we) {
    //TODO: Update the simulation.
    //If minimized, pause
    //Else, resume
}
```

First, get the state of the WindowEvent.

```java
int state = we.getNewState();
```

We will check if the state is equal to Frame.ICONIFIED to see if the window is minimized.

```java
if(((state & Frame.ICONIFIED) == Frame.ICONIFIED))
```

We must use a bitwise ‘AND’ since windows can have multiple states. The states are simply represented as a bit.

If the window is minimized, we will pause the factory and hide the FactoryController. Otherwise, we will resume the factory at its previous state.

We will need to save the speed of the FactorySimulation before the window is minimized. Create another member variable to keep track of this.

```java
int windowSaveSpeed;
```

```java
@Override
public void windowStateChanged(WindowEvent we) {
    int state = we.getNewState();
    if(((state & Frame.ICONIFIED) == Frame.ICONIFIED)){
        setVisible(false);
        windowSaveSpeed = simulationSpeedController.getValue();
        simulationSpeedController.setValue(Constants.simulation_0x);
    } else {
        simulationSpeedController.setValue(windowSaveSpeed);
    }
}
```

We will have to add the WindowStateListener to the clientGUI’s list so it can pass on the events.
factoryController = new FactoryController();
addWindowStateListener(factoryController);

Now the FactorySimulation will be paused while the window is minimized and resume to its original state when maximized.
Part 5 - Restarting the factory

The last button is fairly easy to implement. We want everything to go back to how it was at the start. Disable the continue button, enable the pause button, and set the slider back to 1x speed.

```java
resetButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        continueButton.setEnabled(false);
        pauseButton.setEnabled(true);
        speed = Constants.simulation_1x;
        simulationSpeedController.setValue(Constants.simulation_1x);
        //TODO: Reset the simulation
    }
});
```

Remember to set the speed back to 1x. Otherwise, if you pause the simulation and press reset, the continue button won’t do anything.

Implement a reset() method in the FactoryManager class.

In the loadFactory(Factory,JTable) method, we stop, load, and start a simulation. Save the inFactory and inTable as member variables in the FactoryManager class.

```java
private Factory mFactory;
private JTable mTable;

public void loadFactory(Factory inFactory, JTable inTable) {
    //must stop the current animation, load in the factory, and start the new factorySimulation
    mFactory = inFactory;
    mTable = inTable;
    stop();
    mFactorySimulation = new FactorySimulation(mFactory, mTable);
    mRenderPanel.refresh();
    start();
}
```

Now make the reset() method, and reload the factory.

```java
public void reset() {
    loadFactory(mFactory, mTable);
}
```

Lastly, call this reset() method in the reset button’s ActionListener.
resetButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        continueButton.setEnabled(false);
        pauseButton.setEnabled(true);
        speed = Constants.simulation_1x;
        simulationSpeedController.setValue(Constants.simulation_1x);
        factoryManager.reset();
    }
});

Now the factory can be run multiple times without quitting out of the program or reloading the factory on the server.

*Note: You will notice that the resource amounts do not reload – this is fine and will be handled in another lab.*
Questions:
Answer the following questions concerning some of the theory covered in this lab.

1. What code do you write so that an action is executed when a button is clicked?
2. What code does this in the lab?
3. What does the windowStateChanged method do?
4. How do you override the windowStateChanged method?

Expand on This
Opening the controller can take a bit of time! Although the GUI is nice for new users, try adding some shortcuts.

Before we get to the requirements, let's introduce you to a new type of listener - the KeyListener. Pay attention to this section, because this listener is a bit finicky to use[1].

First and foremost, you can add a KeyListener to just about any Swing component as long as:

1. It has the property setFocusable() AND it is set to TRUE. Some components have this defaulted to true.
2. The focus is on the component, which means it is either tabbed over, shift-tabbed over, or clicked on. However, just like bacon, just because you can add it to any Swing component, it does not mean that you should add it to every component.

In the lab, the main window of the Factory is a JFrame, and we will be clicking a lot in there. Therefore, it would be a good idea to implement the KeyListener in there. Once it is imported, generate the three methods that it will need. keyTyped() is called when a Unicode character is typed in. keyPressed() is called when the key goes down, and keyReleased() is called when the key goes up. For our lab, since we want to deal with both arrow keys (not unicode) and normal letter keys (unicode), you should put your button code into keyPressed(). With that in mind, here are your tasks:

1. When the user presses ‘P’, the simulation should pause.
2. When the user presses ‘C’, the simulation should continue.
3. When the user presses ‘R’, the simulation should reset.
4. When the user presses the right or left arrow keys, the simulation should move faster or slower, respectively.

It might also be a good idea to move the code done in the original lab's buttons into their own methods so they can be called outside of the Controller class.

Also, do not implement the KeyListener in the Controller class or it might not work.

Good luck!

[1] https://docs.oracle.com/javase/tutorial/uiswing/events/keylistener.html