**Lab 3 - Game Flow & Input**

**Part 1 – Game Flow:**

Let’s quickly cover the flow of a game in Mercury. There are 4 main methods that are used in our game that control how the game functions:

* **init()**: Called right as the game is starting, is used to set up any objects or resources that are needed before the game can start.
* **update()**: Called every “**frame**” while the game is running, is used to update all active objects or game state, the main work horse method of your game.
* **render(Graphics g):** Called immediately after every **update()** method call, is in charge of “rendering” or drawing the **“frame”** that the user sees.
* **cleanup():** Called right before the game shuts down, handles cleaning up any objects or data before the program exits.

Before we talk any further, let’s quickly go through what a “**frame**” is. Movies consist of hunderds of thousands of individual pictures or **frames**, which are played very quickly to present the idea that is actively happening, like a flip book. Video games work very similarly, the game is in charge of showing several **frames** every second to the user to give the idea of the game actively happening. You can equate this to how the **render()** method works, it’s in charge of showing the **frame** to the user.

However, unlike a movie which has all of its **frames** already created and put in the reel, a game has to dynamically create each **frame** while the game is playing. If you’re playing Super Mario Bros, the **frame** that is **rendered** to the screen will be different if you press the left button versus the right button. So the game has to **update** its state first, then determine what it has to **render** to the screen. The **update()** method is in charge of running any code necessary to update the state of the game (player location, enemies, lives, points, etc). Then the **render()** method determines what it is we want to **render** to the screen based off of its new state.

For the most part, we will only be using **init()**, **update()**, and **render()**; with the majority of our code residing in the latter two. Here’s the sort of flow of how the methods will be called during your game:

**update()**

**init()**

**render()**

**cleanup()**

On Game Exit

**Part 2 – Game Entities:**

Just like with our regular programming, we want to think of game development in terms of objects. For a game of Pong, what different objects would exist in the game? To keep it basic, we’ll have the two paddles and the ball. Each of these are conceptually an object, but we’re going to call them **game entities** (also called **game objects**). You can think of a **game entity** as anything in a game that is either **updated** as time goes on or **rendered** to the screen.

In the case of Pong, both of the paddles and the ball are **game entities**. Both of the paddles must move according to the player input and the ball moves and changes direction over time; with all three of them needing to be rendered to the screen. For the rest of this labs, we’ll be designing our own version of Pong.

To start off, let’s create a new class called **PongGame** and use the template below to get started:

import com.radirius.mercury.framework.\*;  
import com.radirius.mercury.graphics.\*;  
  
public class PongGame extends Core {  
  
 public PongGame(CoreSetup coreSetup) {  
 super(coreSetup);  
 }  
  
 public static void main(String[] args) {  
 CoreSetup coreSetup = new CoreSetup("Pong");  
 coreSetup.width = 800;  
 coreSetup.height = 600;  
  
 new PongGame(coreSetup).start();  
 }  
  
 public void init() {  
 }  
  
 public void update() {  
 }  
  
 public void render(Graphics g) {  
 }  
  
 public void cleanup() {  
 }  
}

Compile and run your game to make sure everything is working fine. Once that’s done, let’s work on our first **game entity**, the Pong paddles. Create a new class called **PongPaddle**; this class will represent an individual Pong paddle. Add the following import statements to the top of your **PongPaddle** class:

import com.radirius.mercury.framework.\*;  
import com.radirius.mercury.graphics.\*;  
import com.radirius.mercury.math.geometry.\*;

Most **game entity** classes will have at least 3 methods: constructor, **update(), render()**. Notice how these match up nicely with the 3 methods that our game class will usually use! Our game’s **update()** method will want to call every **game entity’s update()** method, just as our game’s **render()** method will want to call every **game entity’s render()** method.

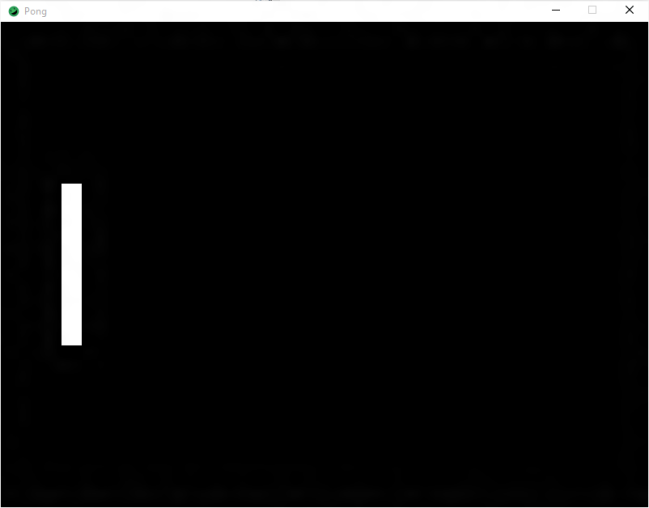
Our paddle will be represented by a white rectangle, so we’ll use a **Rectangle** object to store our paddle’s position and render our paddle to the screen. We want to do the following:

* Create a private **Rectangle** variable for our **PongPaddle** class
* Create a constructor for **PongPaddle** that takes 2 int parameters; **x** and **y**
  + Create a new **Rectangle** object and set our private variable to it.
  + The **Rectangle**’s X and Y coordinates should be the **x** and **y** parameters that were passed in and it should be 25 wide by 200 tall. *(See our 1st & 2nd labs for reminders on this)*
* Create a **render( Graphics g )** method
  + Render our **Rectangle** object to the screen by setting the color to WHITE and then drawing it. *(See our 1st & 2nd labs for reminders on this)*

Once the above is done, compile the **PongPaddle** class to make sure there are no issues. Now, let’s go back to our **PongGame** class. Now that we have a **PongPaddle** class that represents a paddle **game entity**, let’s create an instance of our **PongPaddle** class. In our **PongGame** class, do the following:

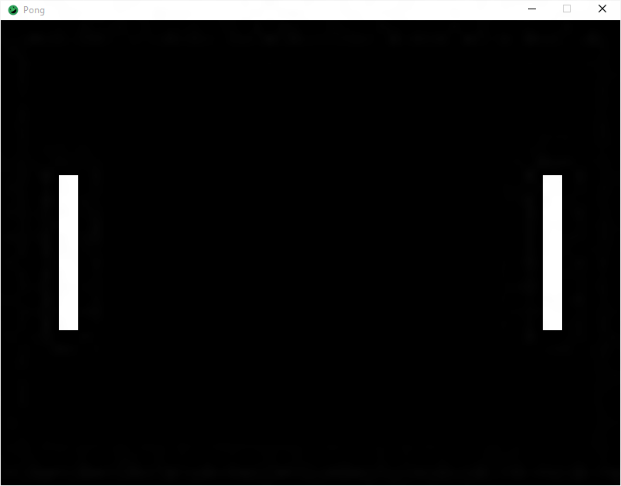
* Create a private **PongPaddle** variable that will be our left paddle
* In the **init()** method
  + Create a new **PongPaddle** object and set our left paddle variable to it
  + Pass in the coordinates (75, 200) for our X and Y values
* In the **render(Graphics g)** method:
  + Call the **render(Graphics g)** method on the **PongPaddle** class for our left paddle, making sure to pass the **Graphics g** variable from the **PongGame** method into it.

Compile and run your code, you should see the following:



Look at that, we’ve got out first pong paddle done! Let’s create another one that will be our right paddle. Create another private **PongPaddle** variable for our right paddle, following the same steps above as we did for the left paddle; except use the coordinates (700, 200) for this one.

Compile and run your code, you should see the following:



Alright, two paddles! But, they don’t do anything…yet.

**Part 3 – Input:**

Let’s make these paddles actually do some interesting stuff using our keyboard as user input. Let’s go back to our **PongPaddle** class. In order to read user input, we’ll need to add the following import statement to the top of our **PongPaddle** class:

import com.radirius.mercury.input.\*;

In order to check for input, we use a **static class** that Mercury provides us called **Input**. The **Input** class provides us with the **Input.keyDown( int key )** method to determine if a specific key is down (being pressed). It takes as a parameter an int representing the key we are looking for. To make our lives easier, all of these int values are already stored for us in the **Input** class. We can get their value by writing **Input.KEY\_\*\*\*** where **\*\*\*** is the name of the key. Here are some examples:

Input.KEY\_R Input.KEY\_8 Input.KEY\_N Input.KEY\_ENTER Input.KEY\_SPACE

There is a full list of all the keys available on the Mercury wiki page. We are going to use the **W** and **S** keys for the paddles. Once we know what key we want to use, we simply pass that value in to the **Input.keyDown()** method, which will return true if the key is currently being pressed; like so:

**Input.keyDown( Input.KEY\_R )**

The other piece we’ll need to accomplish this is the ability to move our **Rectangle** object that represents our paddle. The **Rectangle** class has a method called **translate( int x, int y )**. This method moves the **Rectangle** by the amount specified as **x** and **y**. Positive and negative values can be used for both **x** and **y**; think about how the **Rectangle** would move for both positive and negative values of **x** and **y**.

To get started, let’s create an **update()** method inside of our **PongPaddle** class. The **update()** method will do the following:

* If the **W** key is being pressed, move the rectangle up by 5 pixels
* Else if the **S** key is being pressed, move the rectangle down by 5 pixels

Compile the **PongPaddle** class to make sure there are no errors, then go back to the **PongGame** class. In here, we simply need to add some code to the **update()** method for our **PongGame** class to call the **update()** methods for both of our **PongPaddle** objects.

Once that is done, compile and run your program. You should now be able to move both paddles up and down using the **W** and **S** keys! Eventually we want each of our paddles to move separately, but this works for right now.