GUI - Graphics

Buttons Revisited

- To add functionality to our buttons we set onAction to a function that takes an ActionEvent and returns Unit
  - Let's take a closer look at this

```
import javafx.event.ActionEvent

...

val button: Button = new Button {
  minWidth = 100
  minHeight = 100
  style = "-fx-font: 28 ariel;"
  text = "F to C"
  onAction = (event: ActionEvent) => buttonPressed()
}
```

Buttons Revisited

- onAction is of type EventHandler of ActionEvent
  - EventHandler[ActionEvent]
  - Event handler is a Java Interface (Abstract Class with no state and only abstract methods)
    - Extend and implement the handle method

```
public interface EventHandler<T extends Event> extends EventListener {
    /**
     * Invoked when a specific event of the type for which this handler is
     * registered happens.
     *
     * @param event the event which occurred
     */
    void handle(T event);
}
```

Buttons Revisited

- Create a class that inherits this interface
  - Since ButtonListener is-an EventHandler[ActionEvent] it can be used for onAction

```
import javafx.event.{ActionEvent, EventHandler}

class ButtonListener extends EventHandler[ActionEvent]{

  override def handle(event: ActionEvent): Unit = {
    println("The button was pressed")
  }

}
```

```
public interface EventHandler<T extends Event> extends EventListener {
    /**
     * Invoked when a specific event of the type for which this handler is
     * registered happens.
     *
     * @param event the event which occurred
     */
    void handle(T event);
}
```

Buttons Revisited

- To add functionality to our buttons we set onAction to a function that takes an ActionEvent and returns Unit
  - Let's take a closer look at this

```
import javafx.event.ActionEvent

...

val button: Button = new Button {
  minWidth = 100
  minHeight = 100
  style = "-fx-font: 28 ariel;"
  text = "Just a Button"
  onAction = new ButtonListener()
}
```

Buttons Revisited

- We don't always want to create a new class for every button
- Can use anonymous classes
  - This is the Java way

```
import javafx.event.ActionEvent

...

val button: Button = new Button {
  minWidth = 100
  minHeight = 100
  style = "-fx-font: 28 ariel;"
  text = "Just a Button"
  onAction = new EventHandler[ActionEvent]{
    override def handle(event: ActionEvent): Unit = {
      println("The button was pressed")
    }
  }
}
```

Buttons Revisited

- We don't always want to create a new class for every button
- Scala takes this a step further
  - Assign a method with the correct types and the anonymous class is created implicitly
  - Just make sure the types match (javafx ActionEvent, not scalafx in this case)

```
import javafx.event.ActionEvent

...

val button: Button = new Button {
  minWidth = 100
  minHeight = 100
  style = "-fx-font: 28 ariel;"
  text = "Just a Button"
  onAction = (event: ActionEvent) => {
    println("The button was pressed")
  }
}
```

Buttons Revisited

- However, creating a new type can help with complex programs
- Can use everything we know about classes
  - State, methods, constructors

```
import javafx.event.{ActionEvent, EventHandler}
import scalafx.scene.control.TextField

class ButtonListener(inputDisplay: TextField, outputDisplay: TextField) extends EventHandler[ActionEvent] {

  override def handle(event: ActionEvent): Unit = {
    val fahrenheit: Double = inputDisplay.text.value.toDouble
    val celsius = this.fahrenheitToCelsius(fahrenheit)
    outputDisplay.text.value = f"$celsius%1.2f"
  }

  def fahrenheitToCelsius(degreesFahrenheit: Double): Double = {
    val degreesCelsius = (degreesFahrenheit - 32.0) * 5.0 / 9.0
    degreesCelsius
  }

}
```

```
val alternateButton: Button = new Button {
  minWidth = 100
  minHeight = 100
  style = "-fx-font: 28 ariel;"
  text = "F to C"
  onAction = new ButtonListener(inputDisplay, outputDisplay)
}
```

Buttons Are Cool

But How Does This Help With Our Project?

Graphics - 2D

- Coordinate System has inverted y-axis
- Upper left corner is the origin for an element (screen/ window)

Graphics - 2D

- Add Shapes to a GUI instead of buttons/text fields
- Circle and Rectangle both extend Shape

```
new Circle {
  centerX = 20.0
  centerY = 50.0
  radius = 20.0
  fill = Color.Green
}
```

```
new Rectangle {
  width = 60.0
  height = 40.0
  translateX = 60.0
  translateY = 10.0
  fill = Color.Blue
}
```

Graphics - 2D

- Can add shapes directly to the Scene
  - Better organization to add graphics to a new element and add that element to the Scene

```
var sceneGraphics: Group = new Group {}

val circle: Circle = new Circle {
  centerX = 20.0
  centerY = 50.0
  radius = 20.0
  fill = Color.Green
}
sceneGraphics.children.add(circle)

val rectangle: Rectangle = new Rectangle {
  width = 60.0
  height = 40.0
  translateX = 60.0
  translateY = 10.0
  fill = Color.Blue
}
sceneGraphics.children.add(rectangle)

...

scene = new Scene(windowWidth, windowHeight) {
  content = List(sceneGraphics)
}
```

Graphics - Animation

- Make shapes move/rotate with an ActionTimer
- ActionTimer constructor takes a function (or method) as an argument of type (Long) => Unit
  - This function is called 60 times per second (If possible)
  - The long is the current epoch time in nanoseconds
- This is used in the Clicker GUI to call your update method

```
// define a function for the action timer (Could also use a method)
// Rotate all rectangles (relies on frame rate. lag will slow rotation)
val update: Long => Unit = (time: Long) => {
  for (shape <- allRectangles) {
    shape.rotate.value += 0.5
  }
}

// Start Animations. Calls update 60 times per second (takes update as an argument)
AnimationTimer(update).start()
```

Graphics - User Inputs

- Add functionality with EventHandlers
  - Same idea as for Buttons
- Add handlers to the scene to enable inputs whenever window is in focus

```
scene = new Scene(windowWidth, windowHeight) {
  content = List(sceneGraphics)

  // add an EventHandler[KeyEvent] to control player movement
  addEventHandler(KeyEvent.KEY_PRESSED, (event: KeyEvent) => keyPressed(event.getCode))

  // add an EventHandler[MouseEvent] to draw a rectangle when the player clicks the screen
  addEventHandler(MouseEvent.MOUSE_CLICKED, (event: MouseEvent) => drawRectangle(event.getX, event.getY))
}
```

Graphics - User Inputs

- Inherit the EventHandler[KeyEvent] class for keyboard inputs
    - Listen for key events {KEY_PRESSED, KEY_RELEASED, KEY_TYPED}
- Each event has a key code identifying which key was used

```
def keyPressed(keyCode: KeyCode): Unit = {
  keyCode.getName match {
    case "W" => player.translateY.value -= playerSpeed
    case "A" => player.translateX.value -= playerSpeed
    case "S" => player.translateY.value += playerSpeed
    case "D" => player.translateX.value += playerSpeed
    case _ => println(keyCode.getName + " pressed with no action")
  }
}
```

```
scene = new Scene(windowWidth, windowHeight) {
  // add an EventHandler[KeyEvent] to control player movement
  addEventHandler(KeyEvent.KEY_PRESSED, (event: KeyEvent) => keyPressed(event.getCode))
}
```

Graphics - User Inputs

- Use match/case to react to different keys
  - Similar to switch/case in other languages
- Use underscore for a default case

```
def keyPressed(keyCode: KeyCode): Unit = {
  keyCode.getName match {
    case "W" => player.translateY.value -= playerSpeed
    case "A" => player.translateX.value -= playerSpeed
    case "S" => player.translateY.value += playerSpeed
    case "D" => player.translateX.value += playerSpeed
    case _ => println(keyCode.getName + " pressed with no action")
  }
}
```

```
scene = new Scene(windowWidth, windowHeight) {
  // add an EventHandler[KeyEvent] to control player movement
  addEventHandler(KeyEvent.KEY_PRESSED, (event: KeyEvent) => keyPressed(event.getCode))
}
```

Graphics - User Inputs

- Inherit the EventHandler[MouseEvent] class for mouse inputs
    - Many mouse events including click, enter an element, exit an element, move, drag
- Each event has the (x, y) coordinated of the event

```
def drawRectangle(centerX: Double, centerY: Double): Unit = {
  val newRectangle = new Rectangle() {
    width = rectangleWidth
    height = rectangleHeight
    translateX = centerX - rectangleWidth / 2.0
    translateY = centerY - rectangleHeight / 2.0
    fill = Color.Blue
  }
  sceneGraphics.children.add(newRectangle)
  allRectangles = newRectangle :: allRectangles
}
```

```
scene = new Scene(windowWidth, windowHeight) {
  // add an EventHandler[MouseEvent] to draw a rectangle when the player clicks the screen
  addEventHandler(MouseEvent.MOUSE_CLICKED, (event: MouseEvent) => drawRectangle(event.getX, event.getY))
}
```

Graphics - User Inputs

- Here we use the coordinates of a mouse click to add a rectangle at that location

```
def drawRectangle(centerX: Double, centerY: Double): Unit = {
  val newRectangle = new Rectangle() {
    width = rectangleWidth
    height = rectangleHeight
    translateX = centerX - rectangleWidth / 2.0
    translateY = centerY - rectangleHeight / 2.0
    fill = Color.Blue
  }
  sceneGraphics.children.add(newRectangle)
  allRectangles = newRectangle :: allRectangles
}
```

```
scene = new Scene(windowWidth, windowHeight) {
  // add an EventHandler[MouseEvent] to draw a rectangle when the player clicks the screen
  addEventHandler(MouseEvent.MOUSE_CLICKED, (event: MouseEvent) => drawRectangle(event.getX, event.getY))
}
```

Graphics - User Inputs

- Use match/case to react to different keys
  - Similar to switch/case in other languages
- Use underscore for a default case

```
def keyPressed(keyCode: KeyCode): Unit = {
  keyCode.getName match {
    case "W" => player.translateY.value -= playerSpeed
    case "A" => player.translateX.value -= playerSpeed
    case "S" => player.translateY.value += playerSpeed
    case "D" => player.translateX.value += playerSpeed
    case _ => println(keyCode.getName + " pressed with no action")
  }
}
```

```
scene = new Scene(windowWidth, windowHeight) {
  // add an EventHandler[KeyEvent] to control player movement
  addEventHandler(KeyEvent.KEY_PRESSED, (event: KeyEvent) => keyPressed(event.getCode))
}
```

Lecture Question

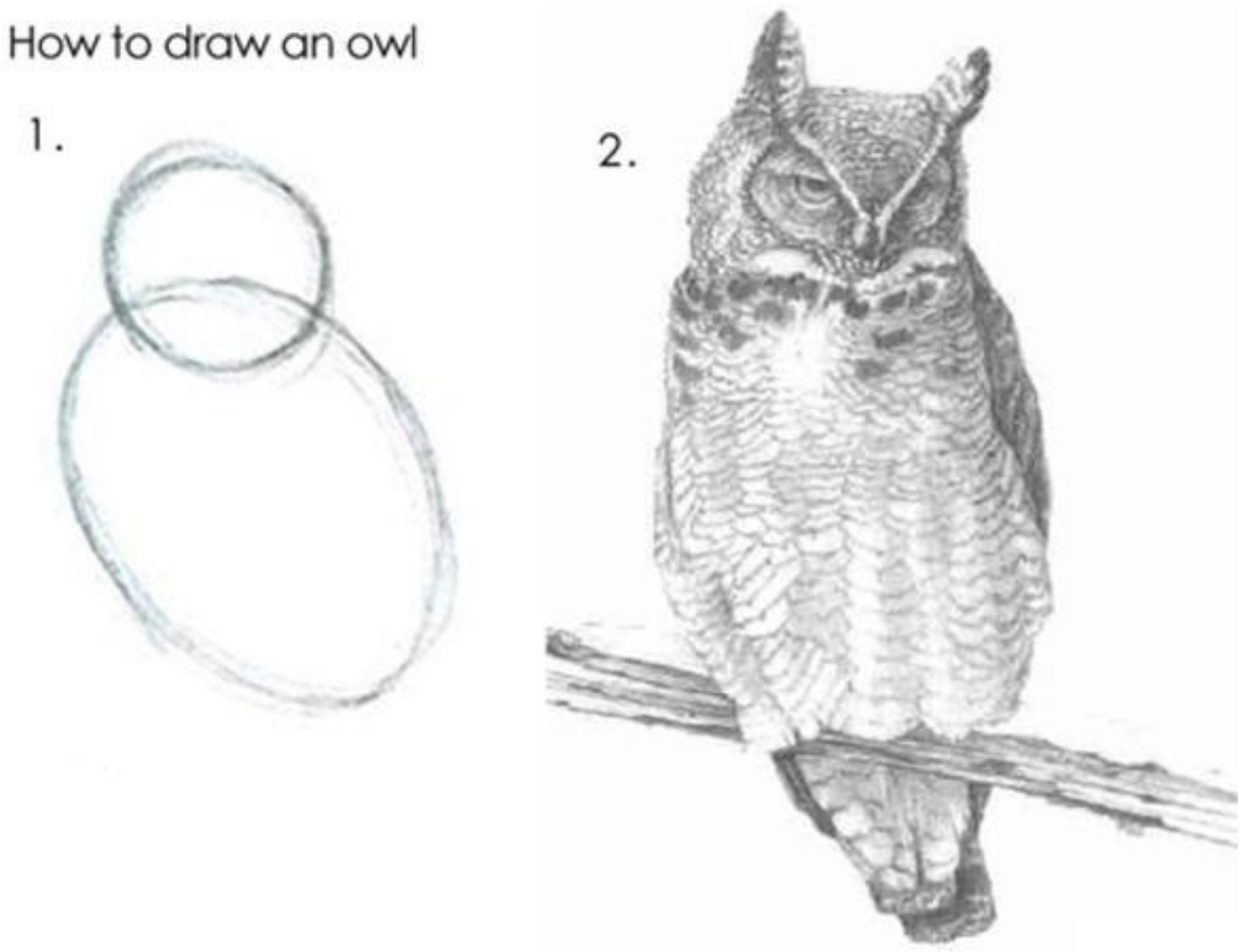
Question: Make pong

- No Grader for this. Submit whatever you have for credit
- Note: If there were a grader, the lecture question would not be this difficult. Try to do as much as you can especially if you're looking for a challenge

If the full game is too challenging, try to setup the graphics (2 rectangles and a circle) and a way for the player to move one of the rectangles (paddle)

* This question will be open until midnight

Graphics - 3D



1. Draw some circles

2. Draw the rest of the focking owl

Graphics - 3D

- ScalaFX has 3D shapes
  - Add to group same as 2D shapes
  - Add a material to set the color

- Create a 5x5x5 grey cube at (0, -2.5, -10)
  - Coordinates
    - Up is negative y (Gravity is positive)

```
new Box(5, 5, 5) {
  material = new PhongMaterial(Color.Grey)
  drawMode = DrawMode.Fill
  transforms.add(new Translate(0, -2.5, -10))
}
```

Graphics - 3D

- Add a camera to the scene to control the player's view

```
var camera_ = new PerspectiveCamera(true) {
  transforms.addAll(new Rotate(0, Rotate.YAxis), new Rotate(0, Rotate.XAxis), new Translate(-5, -3, -50))
}

...

this.scene = new Scene(root, windowWidth, windowHeight) {
  fill = Color.AliceBlue
  camera = camera_
}
```